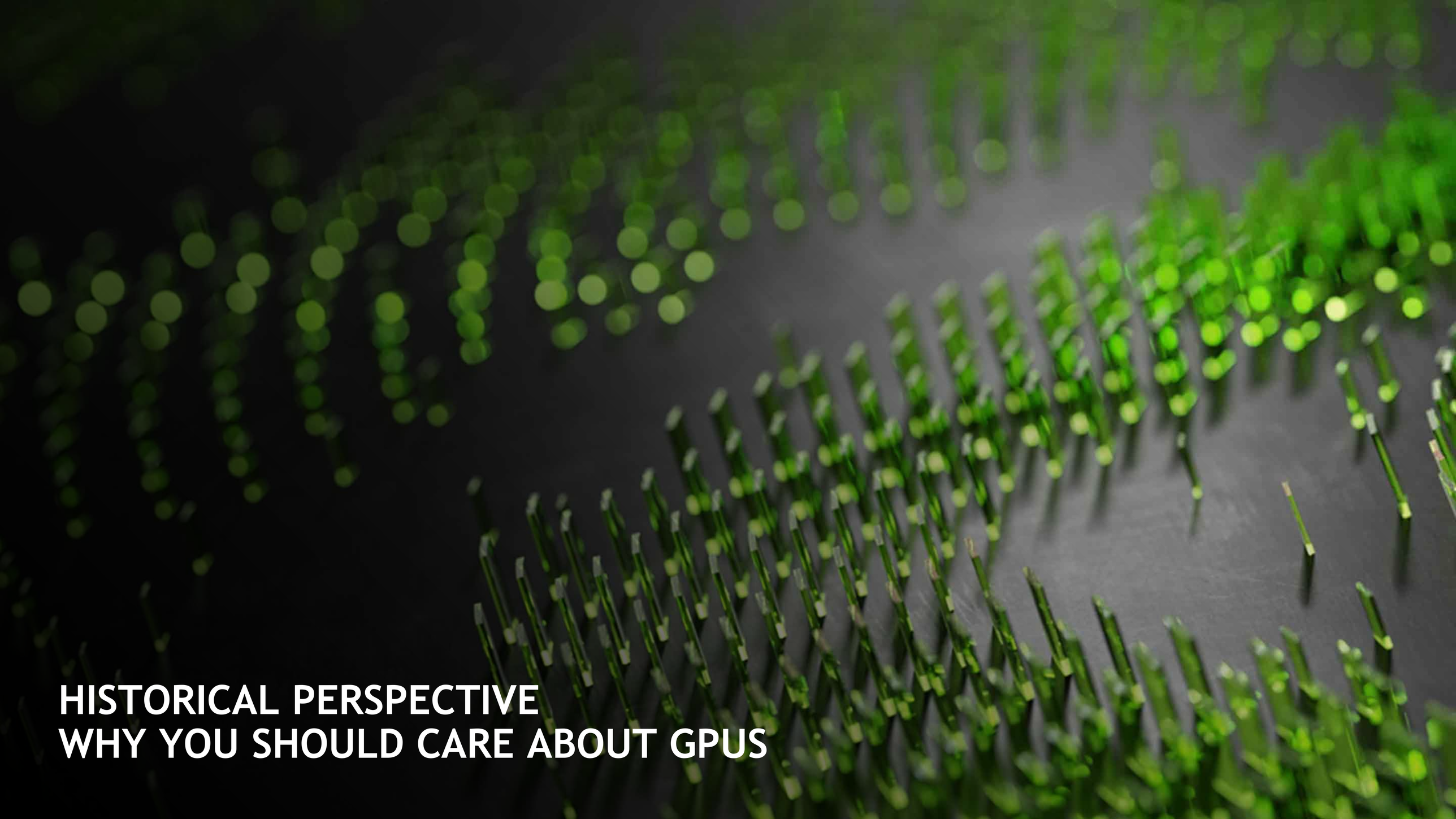
DEVELOPING CFD APPLICATIONS IN A WORLD FILLED WITH GPUS
WHAT YOU NEED TO KNOW

MATT BETTENCOURT, DEVTECH

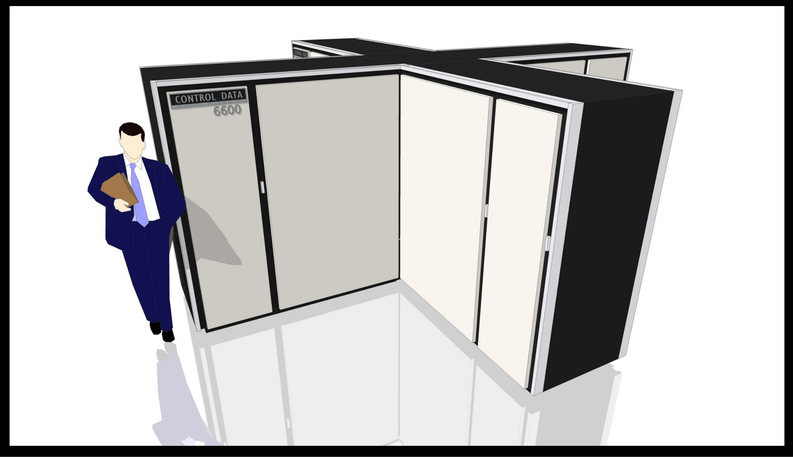


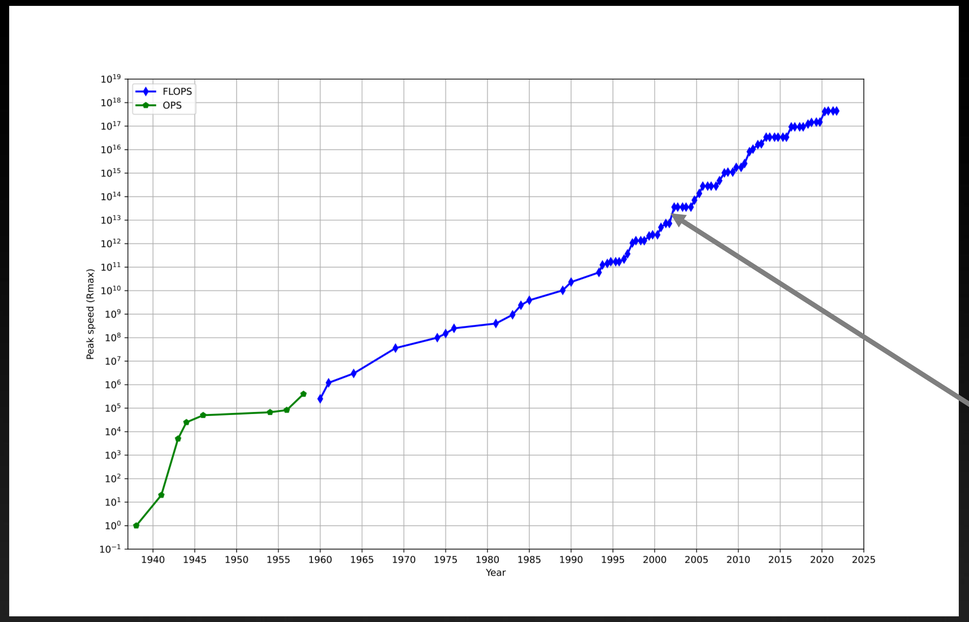
HISTORICAL PERSPECTIVE
WHY YOU SHOULD CARE ABOUT GPUS

A BRIEF HISTORY OF SUPERCOMPUTING

Three main generations of supercomputing

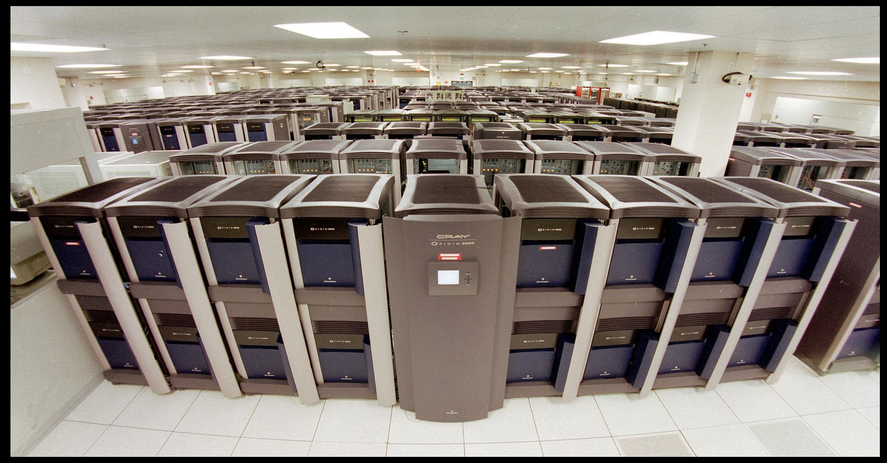
YOUR GRANDPARENTS





NVIDIA
A100 GPU
FP64

YOUR PARENTS



YOUR GENERATION

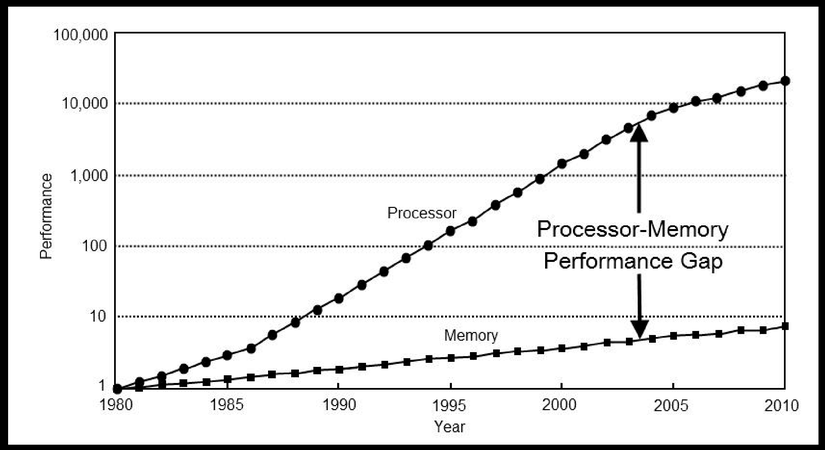
By Morn - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=32719361

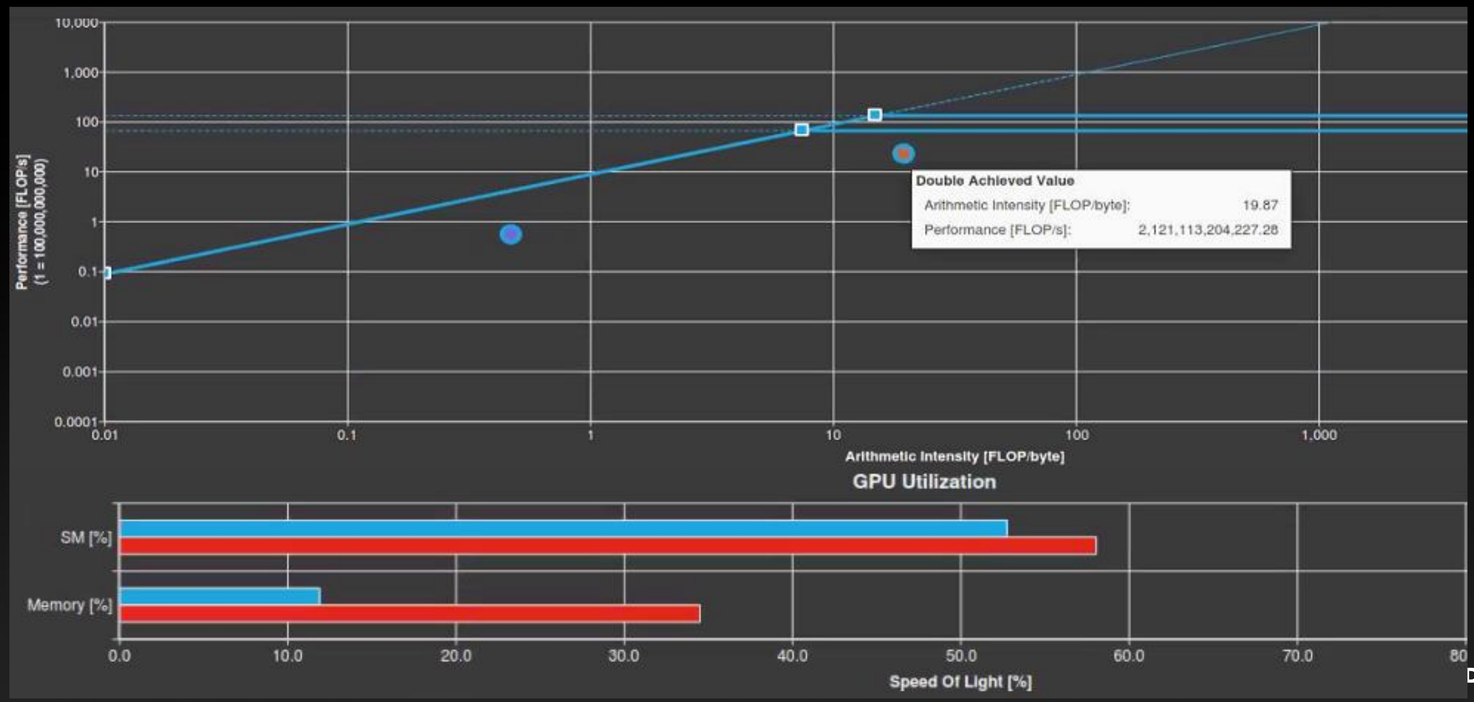
NVIDIA

A TREND THROUGH HISTORY

Roofline model as a way to understand performance

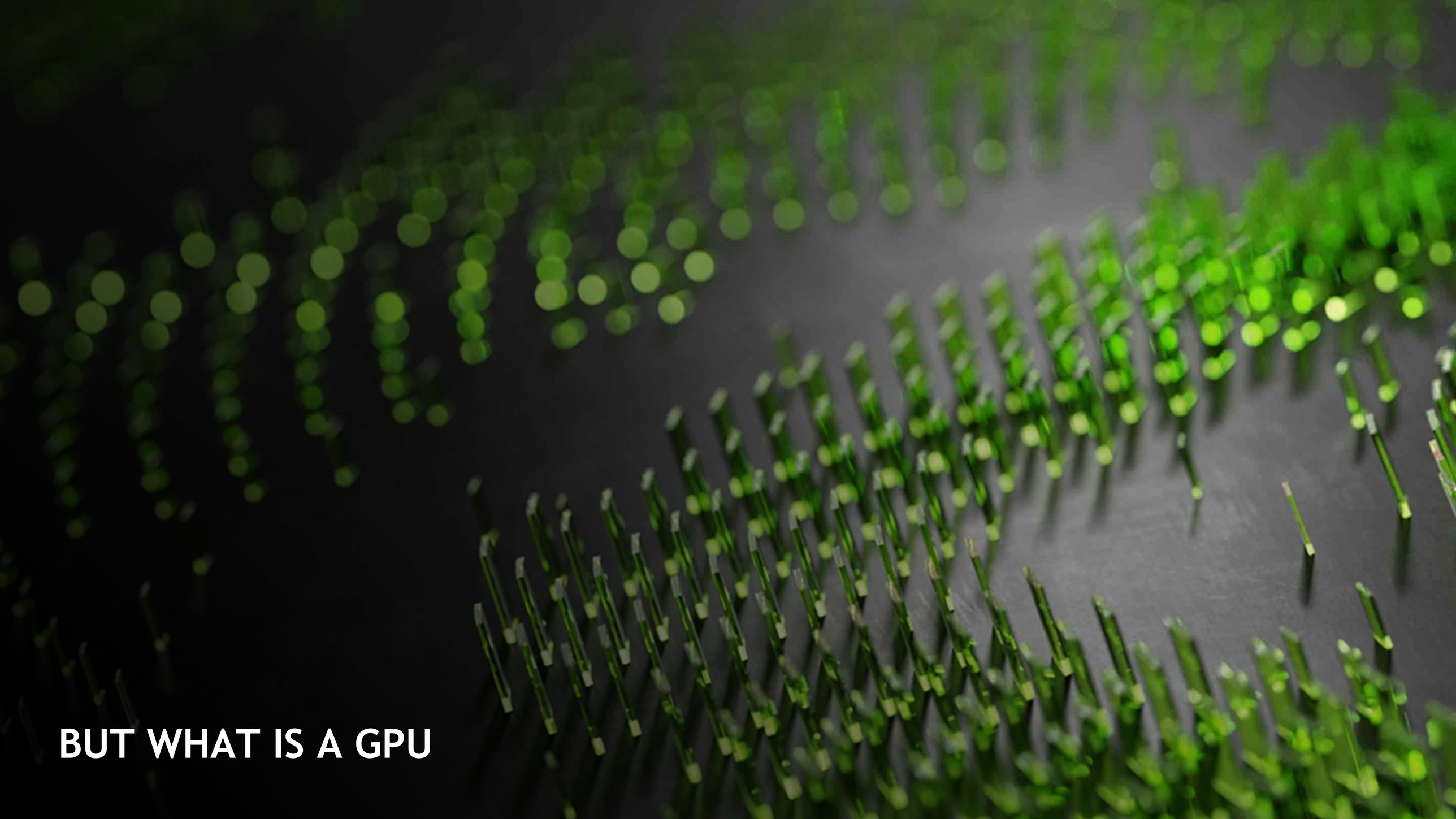
- In the origins of computing memory access was free, all the cost was in FLOPS
- Today, FLOPS are (mostly) free once you have the data on the CPU/GPU
- Roofline models are hardware specific plots of potential and achieved performance
  - Peak performance is plotted against “arithmetic intensity”
  - Arithmetic intensity is the number of floating point operations per byte loaded
  - $y = y + 10 * x + x * x + 0.5 * x * x * x$ has two loads of 8 bytes and 8 operations, intensity of 0.5
  - This would have a peak 400 GFLOP in the graph below
- In the olden days, an arithmetic intensity of 1, or less, would give you peak performance
- On an A100, you need 10-50 for peak performance
- How does one increase arithmetic intensity?
  - This is a function of the algorithm
  - Matrix-Matrix-Multiply (theoretical)
    - Operations are $2N^3$ memory accesses are $16N^2$
    - Intensity as high as $1/8^{th}$ the matrix dimension
  - Low order finite difference have compute intensities around 0.1 to 1.0
    - High order methods improve on this greatly
- Algorithms will have to change to be efficient on modern hardware





WHAT HAS HISTORY TAUGHT US

- Radical shifts in hardware will occur in your professional lifetime
  - I've developed software for all three generations listed here
- Complexity will increase
  - Every new generation will have to deal with the challenges of the previous generation
  - Tools, languages and libraries will help hide the complexity
- What we learn today will guide the way we solve things tomorrow
- As computers get faster, the speed of light doesn't change
  - Memory speed and latency become more and more important
  - Hardware folks will try to hide this latency by more cache and other tricks
  - One will have to reuse the data once it has been brought to the computing engine
- Algorithms will have to adjust to make the most from the new hardware
  - The "fastest" algorithm isn't always the fastest


BUT WHAT IS A GPU

SO, WHAT MAKES A GPU DIFFERENT?

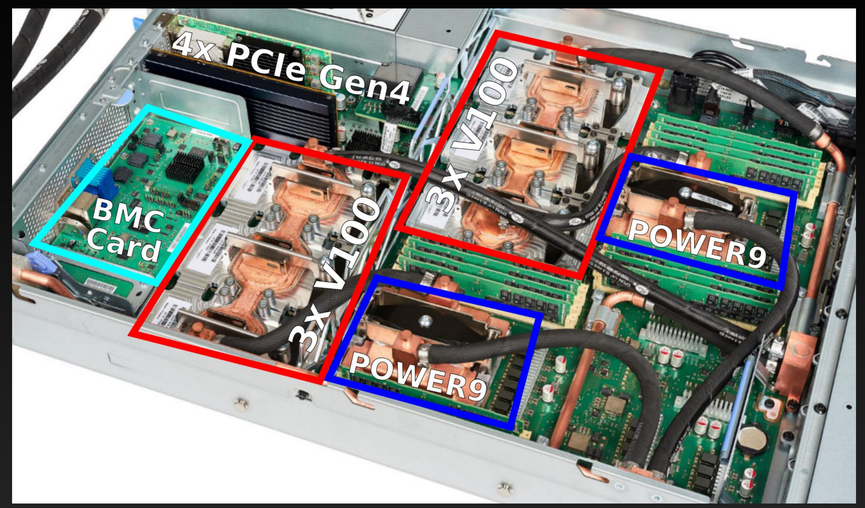
GPUs are about concurrency

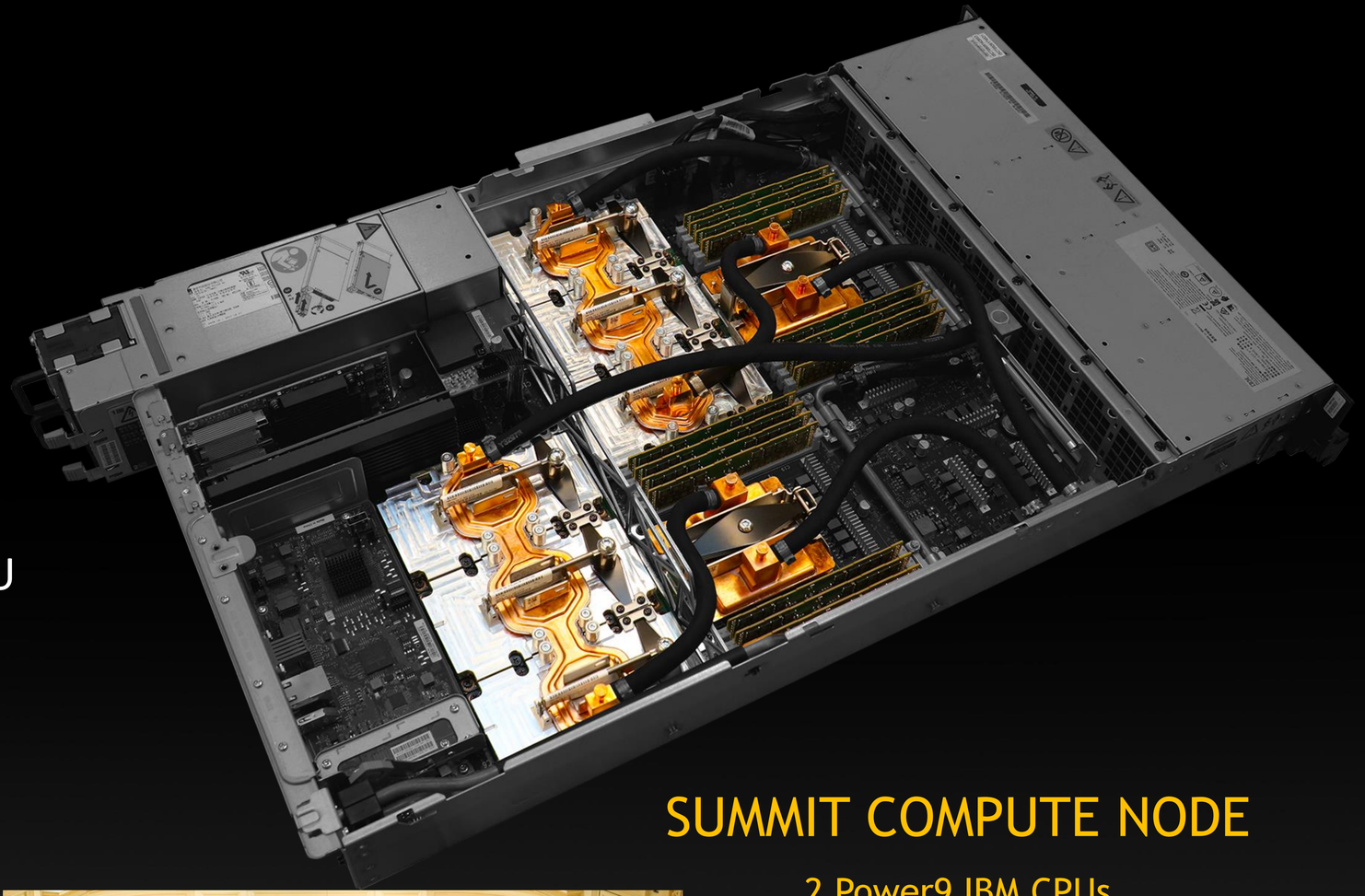
Many independent tasks operating at the same time

Many – 10s of thousands

INTEGRATION OF GPUS INTO SYSTEMS

- Systems have CPUs
  - 10% of the FLOPS
- Large system memory
- GPUs – 90% of the FLOPS
  - Small(er) memory – 80G
  - Low(er) bandwidth to system memory – 900GB/s
  - Each thread is slower than CPU





SUMMIT COMPUTE NODE

2 Power9 IBM CPUs

6 NVIDIA V100

NVLINK Interconnect

A LOOK INSIDE THE GPU

This is similar to what is in Leonardo

- What is inside of a GPU?
- Clock – 1410 MHz
- Processors
  - 108 SM – Streaming Multiprocessors
  - Basic unit of computing inside of a GPU
  - 32 FP64 computational threads
  - Can perform 32 FMA/cycle

FLOPS = 108 SMs*32 Threads*1.41GHz*2 = 9.7TFLOP

- Memory, different types of memory
  - HBW memory (16-80G), L2 Cache 40MB, L1/shared 164K/SM, Texture
  - Each thread can request 1 double per clock cycle

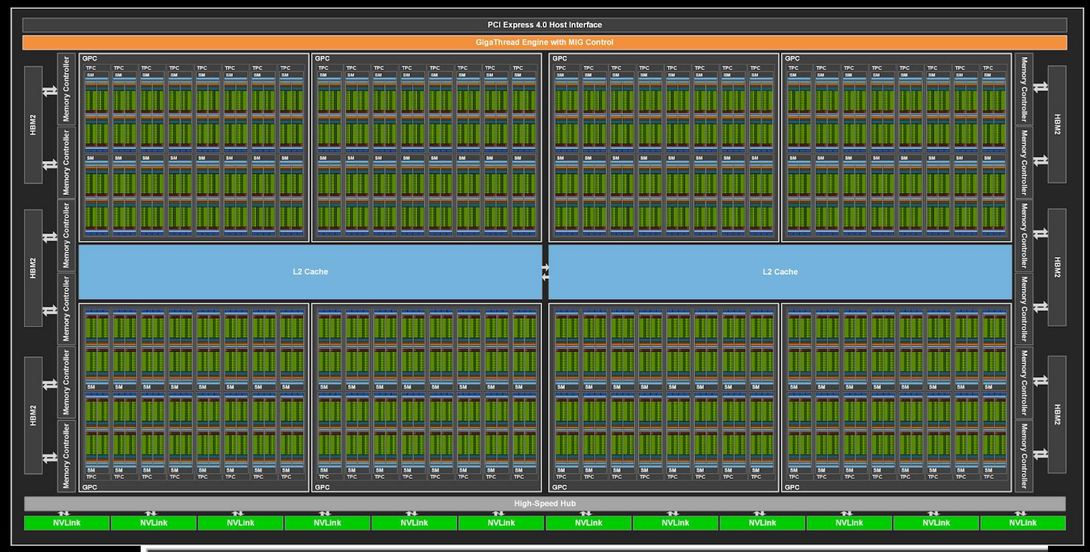
Mem = 108 SMs*32 Threads*1.41GHz *8B = 78TB/s

OK, that's what it can request, but

1.6TB/s is what it can deliver

- Schedulers – the unsung hero

GPUs - 5x FLOPS and 10x memory bandwidth CPUs




GPUS, WHAT ARE THEY GOOD FOR?

WITH GPUS, WHY DO WE HAVE CPUS

With the performance of GPUs, why do we still have CPUs?

GPUs have a much slower clock speed than CPUs

Streaming Multiprocessor (SM) are well, streaming

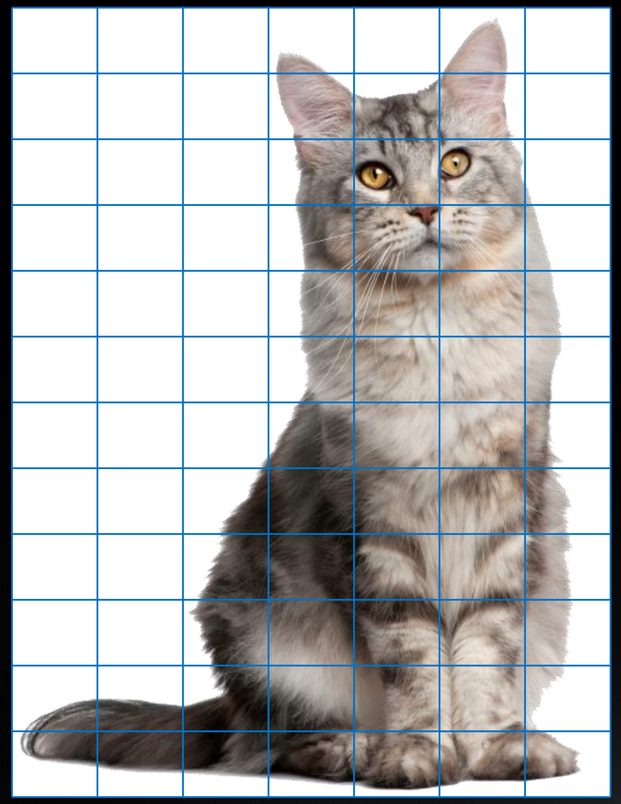
Designed for SIMT

GPUs are most efficient at a particular type of work

TAKE THIS CAT IMAGE

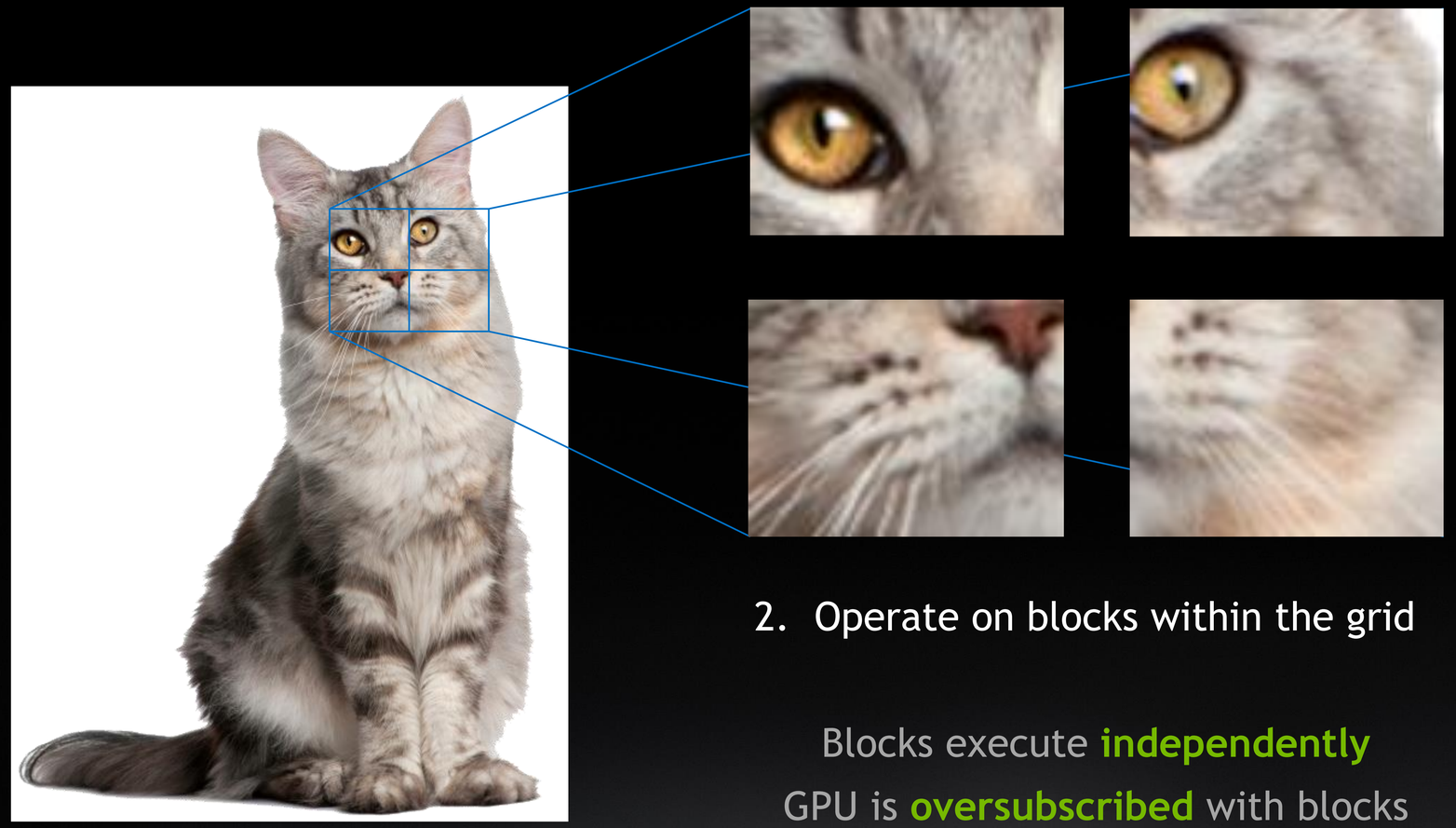
Let’s improve this image

FIRST WE BREAK IT UP ACROSS BLOCKS AND SEND TO SM



1. Overlay with a grid

EVERY PART OF THE IMAGE GETS A BUNCH OF THREADS



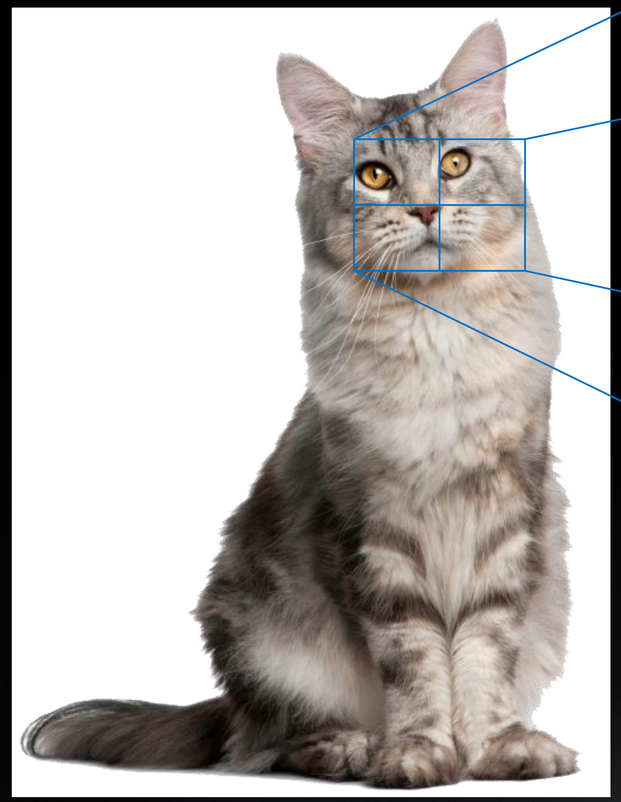
1. Overlay with a grid

2. Operate on blocks within the grid

Blocks execute **independently**

GPU is **oversubscribed** with blocks

EACH THREAD MODIFIES ITS PORTION

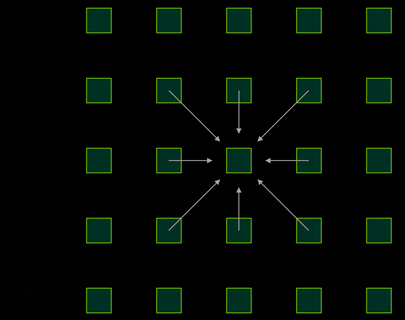


1. Overlay with a grid

2. Operate on blocks within the grid

Blocks execute **independently**

GPU is **oversubscribed** with blocks



3. Many threads work together in each block for **local** data sharing

THAT DATA IS WRITTEN BACK TO MEMORY

Now your cat image is a dog

or

Your CFD variables are updated

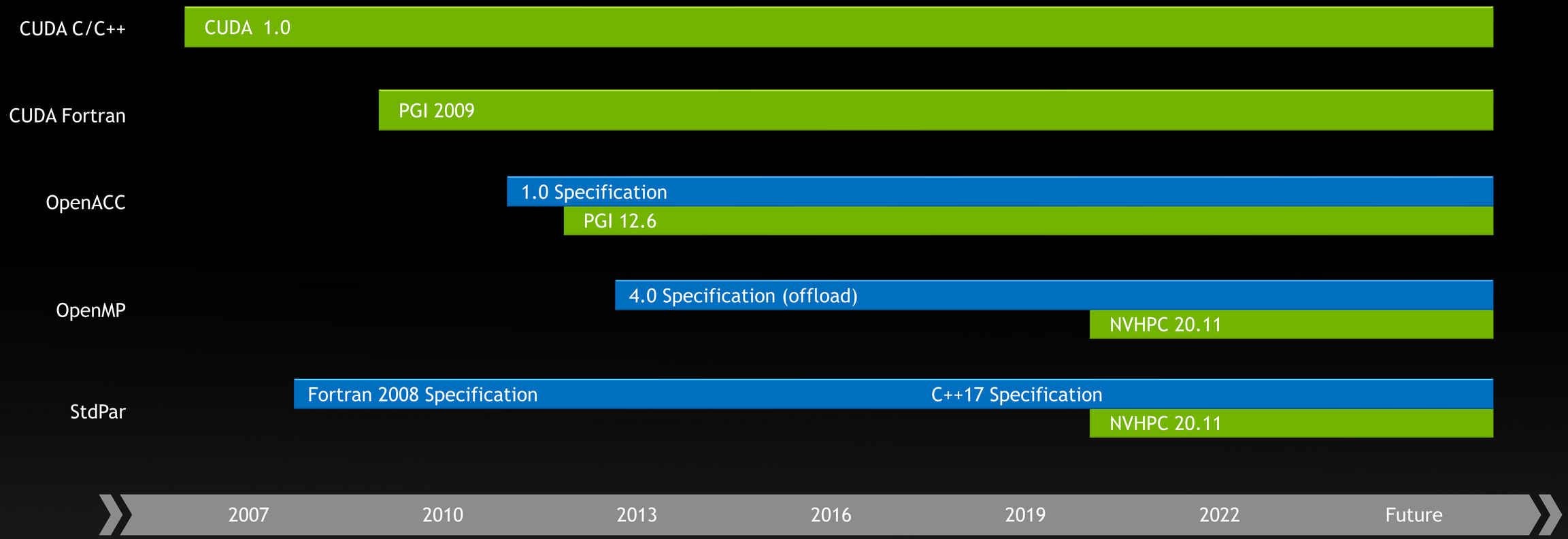

NOW THAT YOU CARE ABOUT GPUS
HOW TO USE THEM

YOU HAVE OPTIONS WHEN PROGRAMMING FOR A GPU

- When GPUs first came out you had Cuda and everything was manual
- Today – you still have Cuda and you can still do everything yourself
- However, today you have lots of options
  1. You can use language standard features
  2. You can use directive based languages
  3. You can use frameworks that abstract the hardware away
  4. You can use libraries
  5. You can write native Cuda
- Starting with a new code versus an existing code can really affect what path you take

GPU PROGRAMMING MODELS

A brief history



NVIDIA Compiler and Language Support

Accelerated Standard Languages

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y + a*x;
}
);
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

```
import legate.numpy as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

Incremental Portable Optimization

```
#pragma acc data copy(x,y) {
...
#pragma acc parallel loop
for (i=0; i<n; i++) {
  y[i] += a * x[i];
}
...
}
```

```
#pragma omp target data map(x,y) {
...
#pragma omp target teams loop
for (i=0; i<n; i++) {
  y[i] += a * x[i];
}
...
}
```

Platform Specialization

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

Core	Math	Communication	Data Analytics	AI	Quantum

Acceleration Libraries

https://developer.nvidia.com/nvidia-hpc-sdk-downloads

https://commons.wikimedia.org/wiki/File:Ferrari_F355_Spider_-_Flickr_-_The_Car_Spy_(16).jpg

WHAT IS THE GPU GEARBOX?

The GPU gearbox is a mental model for thinking about programming models, to deliver the best performance at different levels of developer effort and specialization.

Think about torque, not speed…

First Gear

ISO standard parallelism: Easiest to adopt. Maximum portability. Good performance in a subset of use cases.

Second Gear

Performance libraries: Peak performance for supported features, which include a wide range of common patterns in linear algebra, machine learning and data analysis.

Third Gear

Directives and Pragmas: Easy to adopt. Good portability. Great performance in many use cases.

Fourth Gear

CUDA languages: Exposes full hardware capability and enables maximum performance. Supported on all NVIDIA GPUs.

https://commons.wikimedia.org/wiki/File:Ferrari_F355_Spider_-_Flickr_-_The_Car_Spy_(16).jpg

WHAT IS THE GPU GEARBOX?

The GPU gearbox is a mental model for thinking about programming models, to deliver the best performance at different levels of developer effort and specialization.

Think about torque, not speed…

First Gear

ISO standard parallelism: Easiest to adopt. Maximum portability. Good performance in a subset of use cases.

Second Gear

Performance libraries: Peak performance for supported features, which include a wide range of common patterns in linear algebra, machine learning and data analysis.

Third Gear

Directives and Pragmas: Easy to adopt. Good portability. Great performance in many use cases.

Fourth Gear

CUDA languages: Exposes full hardware capability and enables maximum performance. Supported on all NVIDIA GPUs.

https://commons.wikimedia.org/wiki/File:Ferrari_F355_Spider_-_Flickr_-_The_Car_Spy_(16).jpg

WHAT IS THE GPU GEARBOX?

The GPU gearbox is a mental model for thinking about programming models, to deliver the best performance at different levels of developer effort and specialization.

Think about torque, not speed…

First Gear

ISO standard parallelism: Easiest to adopt. Maximum portability. Good performance in a subset of use cases.

Second Gear

Performance libraries: Peak performance for supported features, which include a wide range of common patterns in linear algebra, machine learning and data analysis.

Third Gear

Directives and Pragmas: Easy to adopt. Good portability. Great performance in many use cases.

Fourth Gear

CUDA languages: Exposes full hardware capability and enables maximum performance. Supported on all NVIDIA GPUs.

https://commons.wikimedia.org/wiki/File:Ferrari_F355_Spider_-_Flickr_-_The_Car_Spy_(16).jpg

WHAT IS THE GPU GEARBOX?

The GPU gearbox is a mental model for thinking about programming models, to deliver the best performance at different levels of developer effort and specialization.

Think about torque, not speed…

First Gear

ISO standard parallelism: Easiest to adopt. Maximum portability. Good performance in a subset of use cases.

Second Gear

Performance libraries: Peak performance for supported features, which include a wide range of common patterns in linear algebra, machine learning and data analysis.

Third Gear

Directives and Pragmas: Easy to adopt. Good portability. Great performance in many use cases.

Fourth Gear

CUDA languages: Exposes full hardware capability and enables maximum performance. Supported on all NVIDIA GPUs.

https://commons.wikimedia.org/wiki/File:Ferrari_F355_Spider_-_Flickr_-_The_Car_Spy_(16).jpg

WHAT IS THE GPU GEARBOX?

The GPU gearbox is a mental model for thinking about programming models, to deliver the best performance at different levels of developer effort and specialization.

Think about torque, not speed…

First Gear

ISO standard parallelism: Easiest to adopt. Maximum portability. Good performance in a subset of use cases.

Second Gear

Performance libraries: Peak performance for supported features, which include a wide range of common patterns in linear algebra, machine learning and data analysis.

Third Gear

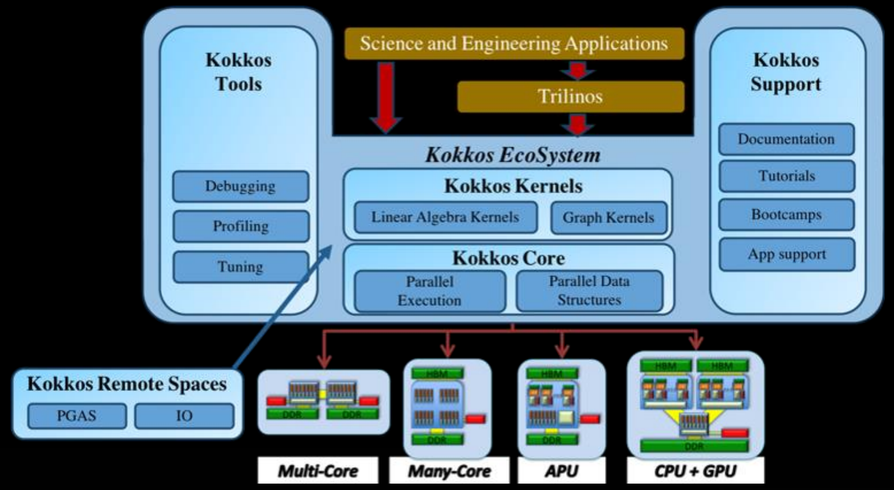
Directives and Pragmas: Easy to adopt. Good portability. Great performance in many use cases.

Fourth Gear

CUDA languages: Exposes full hardware capability and enables maximum performance. Supported on all NVIDIA GPUs.

AND THEN THERE ARE FRAMEWORKS

- Frameworks try to abstract the hardware from the application code
  - Kokkos is one such abstraction
- Frameworks can be difficult to retrofit into your application.
  - Does the framework manage the data for you?
  - Does the framework manage MPI for you, ghost exchanges?
  - Does the framework manage the discretization for you?
- Frameworks can disappear, it could have been a PhD project
- Frameworks can make your life much easier
  - But it can be hard to work outside what they intended you to do
- Frameworks can hide complexity
  - But can also inhibit performance
- Frameworks can let you code to any backend
  - Develop with CPU threads
  - Deploy on GPUs
- By this definition, the C++ stdpar is a framework




```
Kokkos::View<double*> x(“x”,n), y(“y”,n);
Kokkos::parallel_for(n,KOKKOS_LAMBDA(int i)
    { y(i) += a*x(i); }
);
```

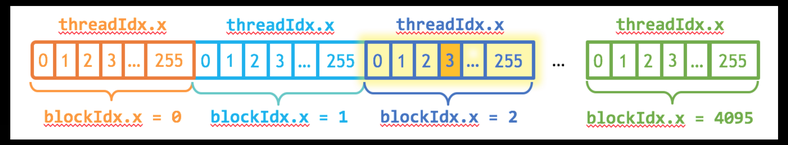
SHIFTING THROUGH THE GEARS

Experiments with linear algebra primitives

VECTOR ADDITION

Memory bandwidth

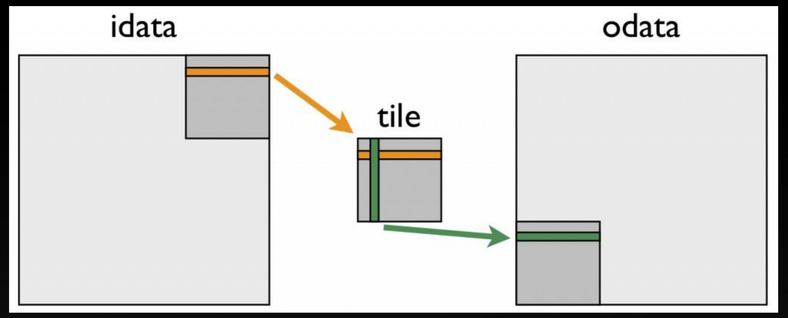
$$\forall i: \; Z_i = a \times X_i + Y_i$$



MATRIX TRANSPOSE

Memory bandwidth, shared memory, and coalescing

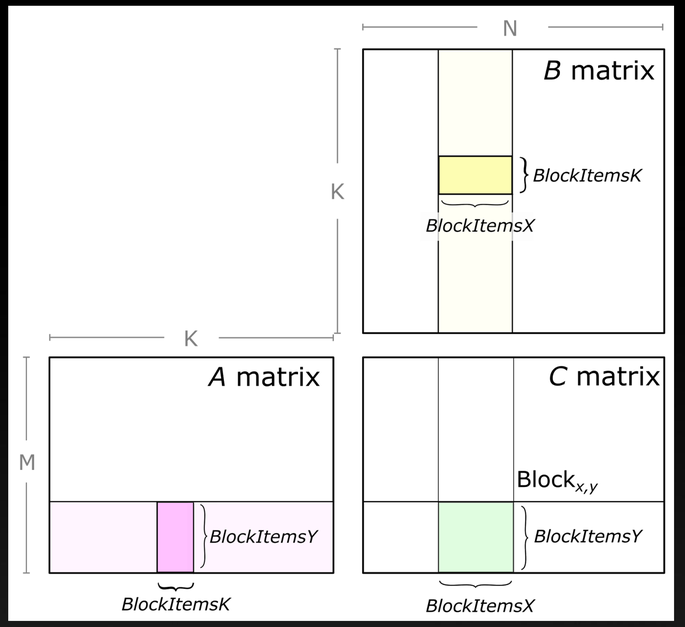
$$\forall i, j: \; B_{i,j} = B_{i,j} + A_{j,i}$$



MATRIX MULTIPLICATION

Optimize everything…

$$\forall i, j: \; C_{i,j} = \sum_k A_{i,k} \times B_{k,j}$$

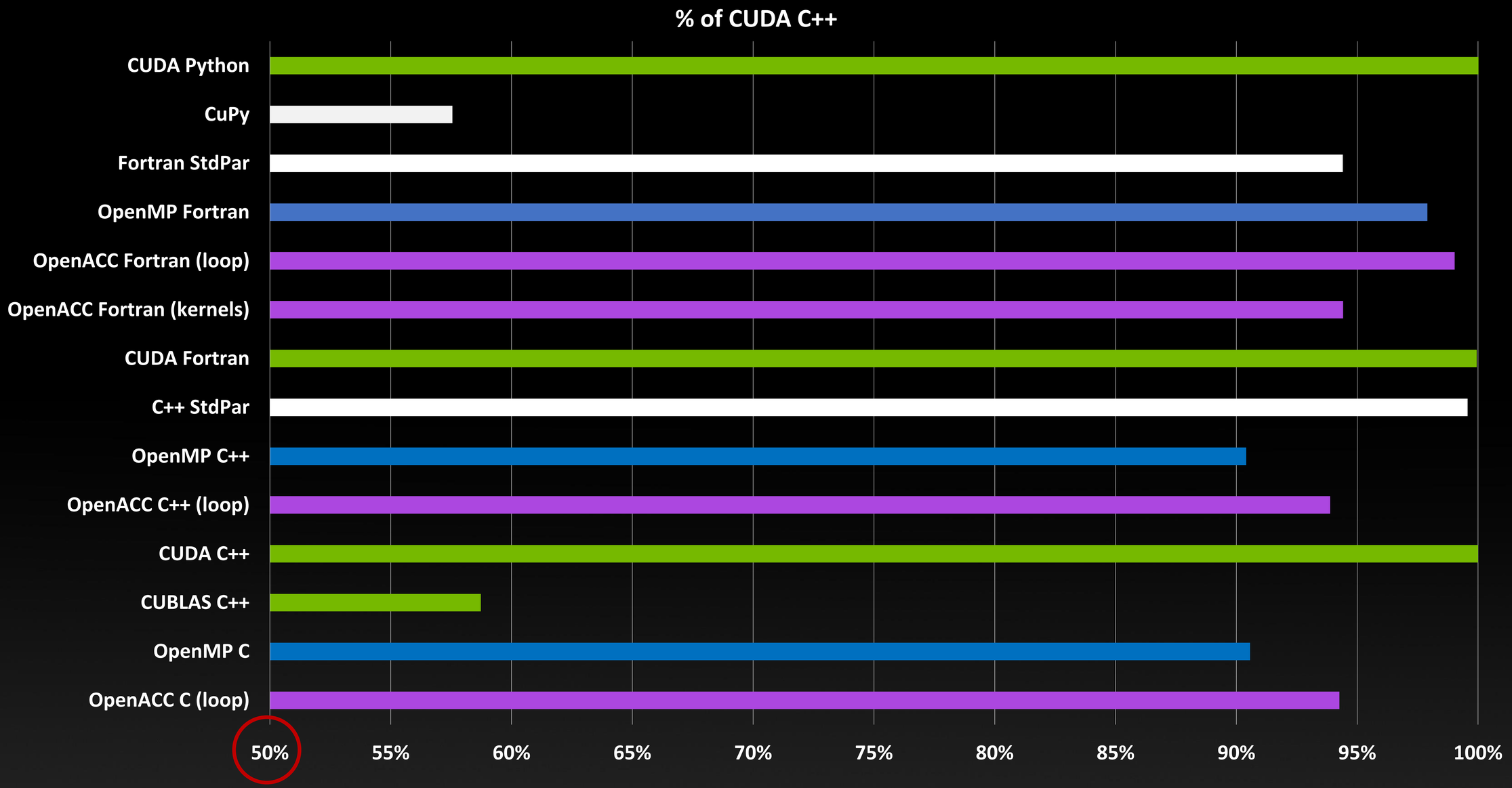


https://developer.nvidia.com/blog/even-easier-introduction-cuda/

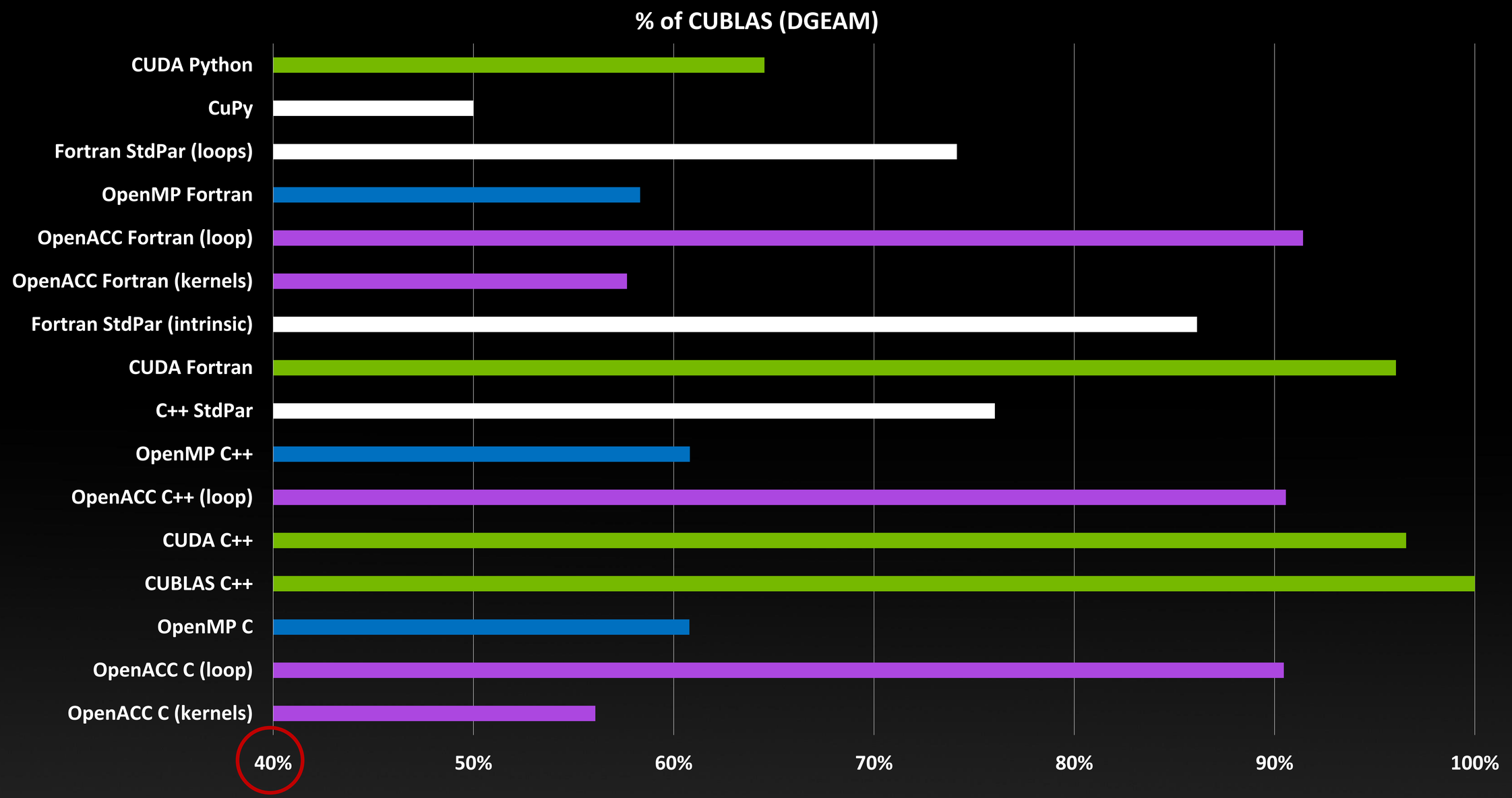
https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/

https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/

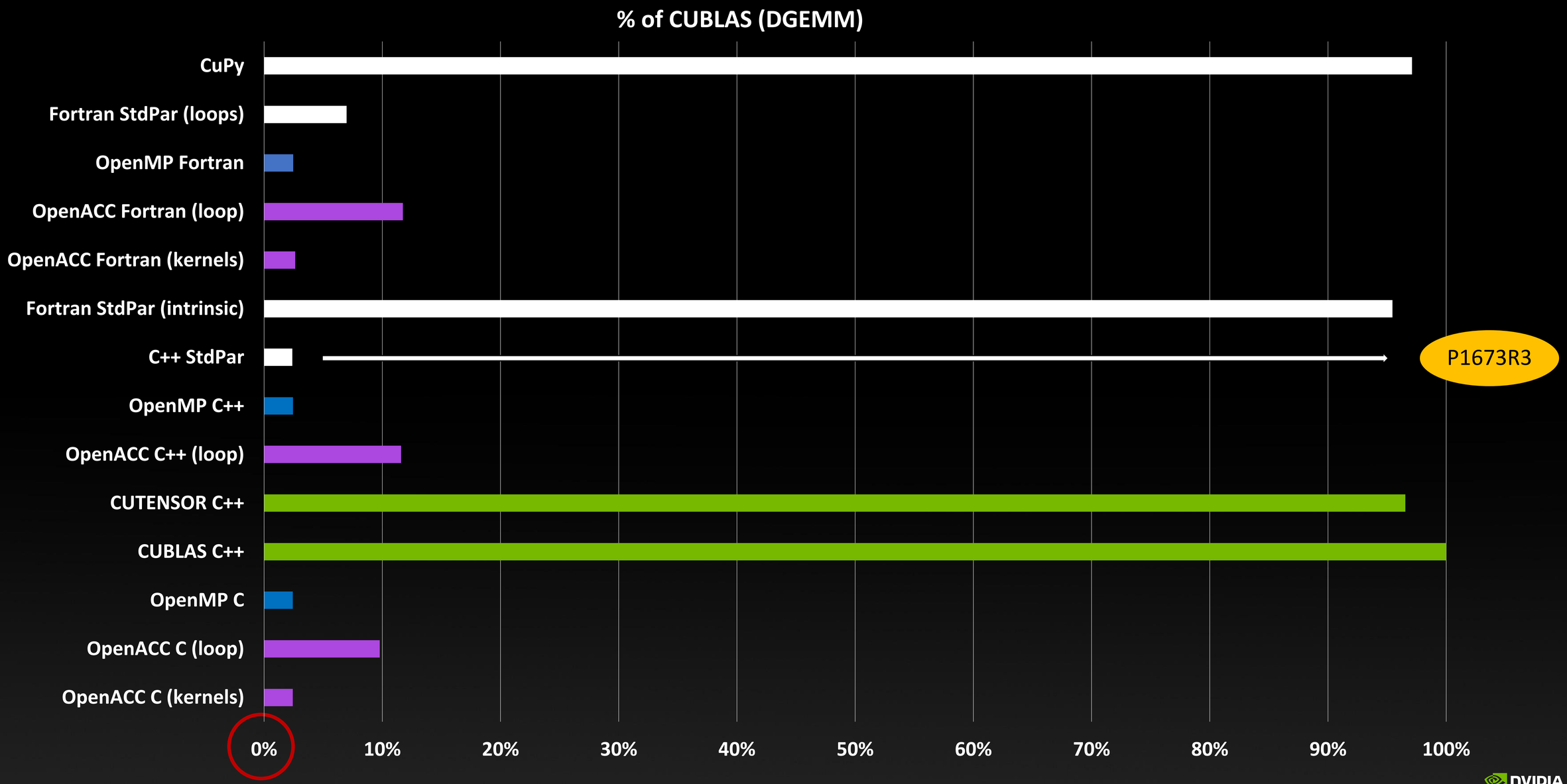
Vector Addition: Z = a * X + Y



Matrix Transpose: B = B + A^T



Matrix Multiplication: C = C + A * B



WHAT PARADIGM SHOULD YOU USE

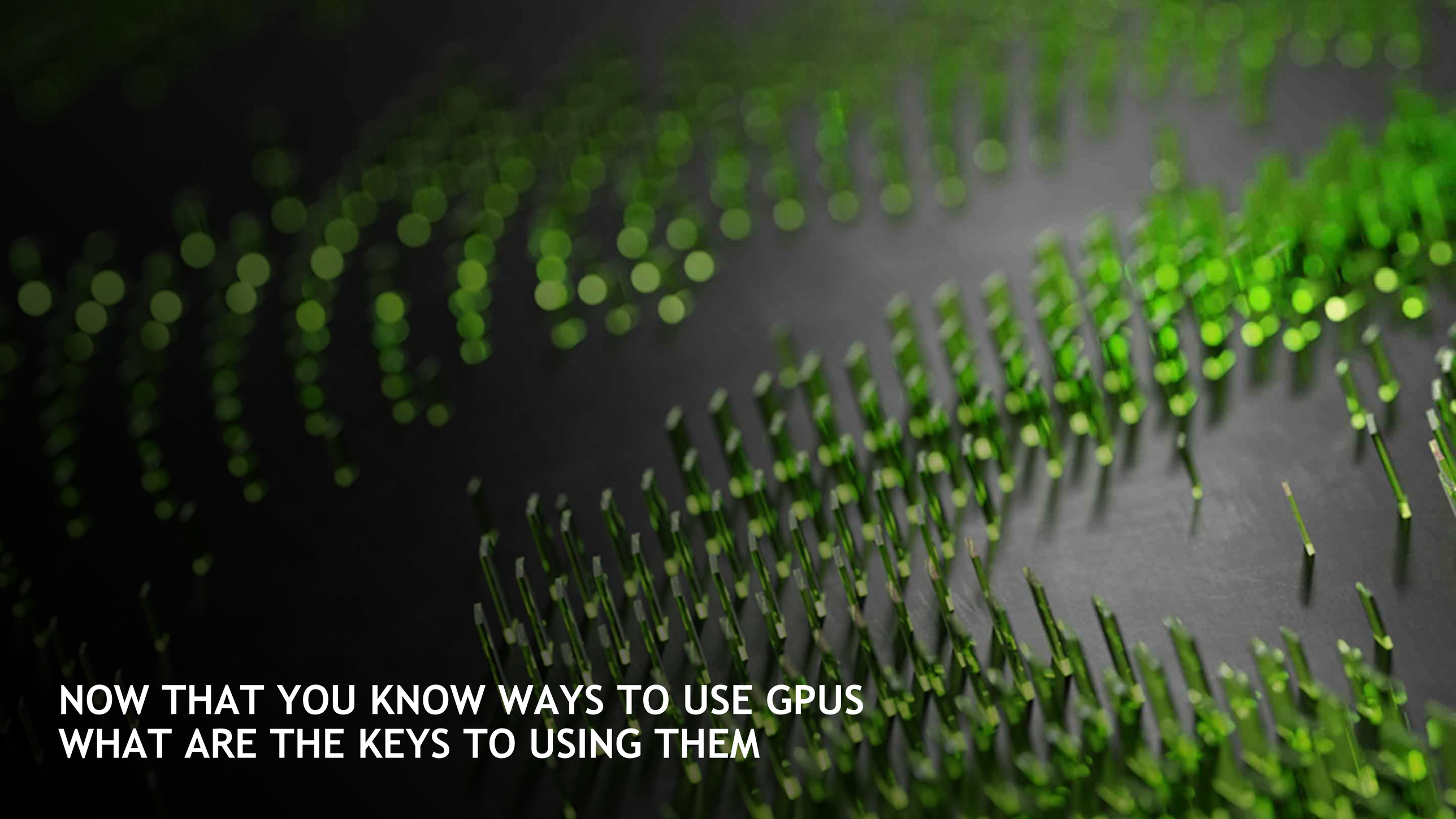
Well, it depends

For a lot of applications standard languages work very well

Specific kernels require special attention

Libraries - Matrix math, FFTs, tensor contractions and others

Using a mixture of different paradigms can give you the best of all worlds


NOW THAT YOU KNOW WAYS TO USE GPUS
WHAT ARE THE KEYS TO USING THEM

IT'S ALL ABOUT THE MEMORY

FLOPS are free

Simple definitions

FLOPS – Floating Point Operations Per Second

Memory Latency – Time between memory request and arrival

Memory Bandwidth – How much memory comes per second

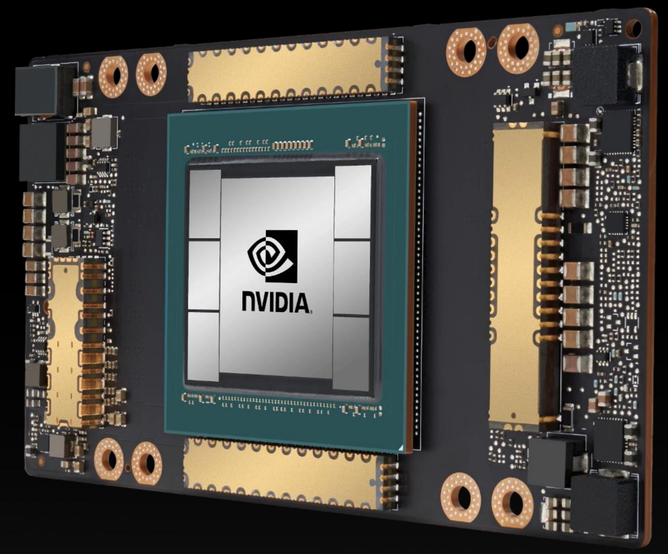
Shared Memory – Local fast shared memory to a SM

Compute Intensity – FLOPS/BYTE

THE NVIDIA AMPERE GPU ARCHITECTURE

These are the resources that are available

SMs	108
Total threads	221,184
Peak FP32 TFLOP/s	19.5
Peak FP64 TFLOP/s (non-tensor)	9.7
Peak FP64 TFLOP/s (tensor)	19.5
Tensor Core Precision	FP64, TF32, BF16, FP16, I8, I4, B1
Shared Memory per SM	160 kB
L2 Cache Size	40960 kB
Memory Bandwidth	1555 GB/sec
GPU Boost Clock	1410 MHz
NVLink Interconnect	600 GB/sec

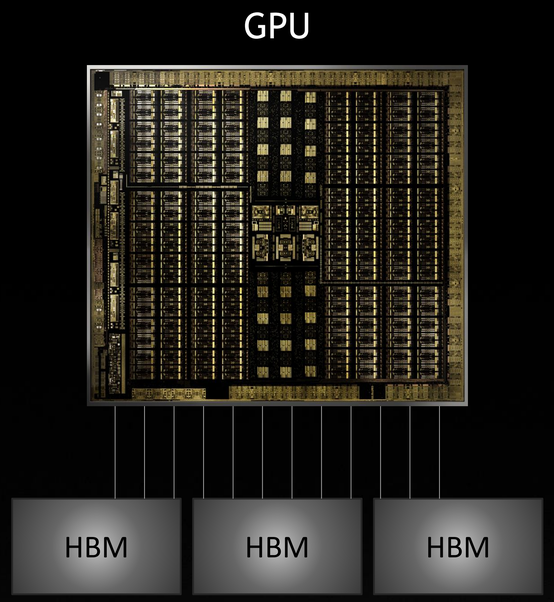


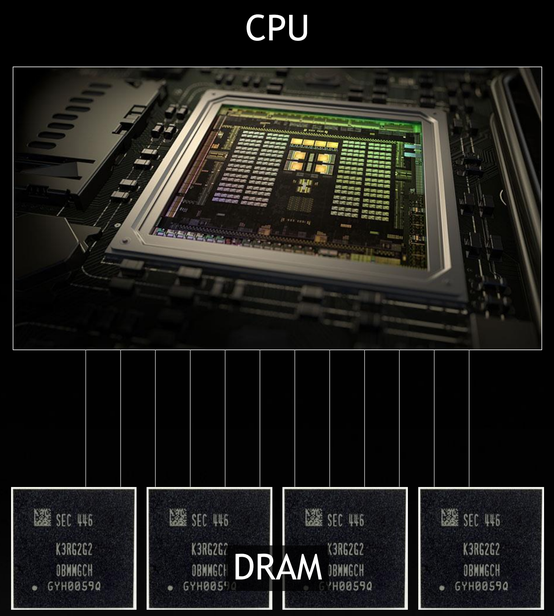
ARITHMETIC INTENSITY=9.7/1.555=6.25

Well, we want doubles, 8x!!

We need to use every load 50x

NOT JUST A GPU ISSUE





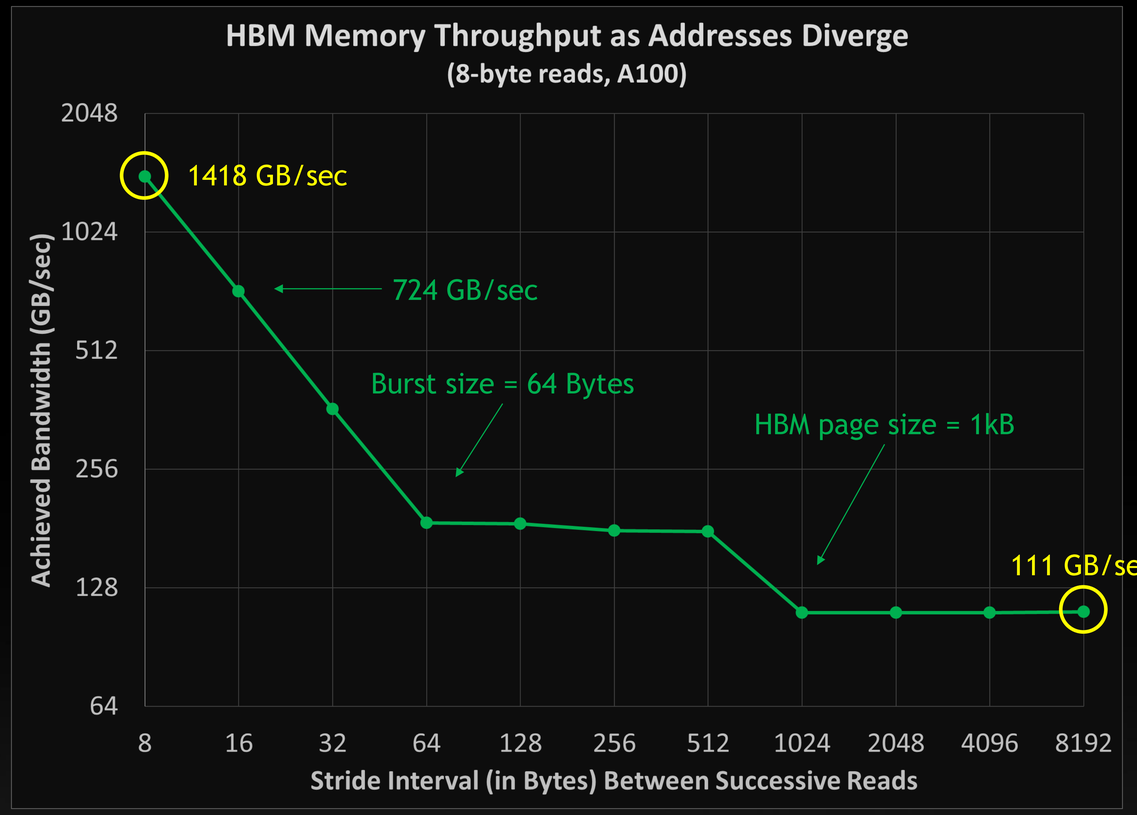
	NVIDIA A100	Intel Xeon 8280	AMD Rome 7742
Peak FP64 GigaFLOPs	9700	2190	2300
Memory B/W (GB/sec)	1555	131	204
Compute Intensity	50	134	90

THERE IS STILL A LOT OF MEMORY BANDWIDTH

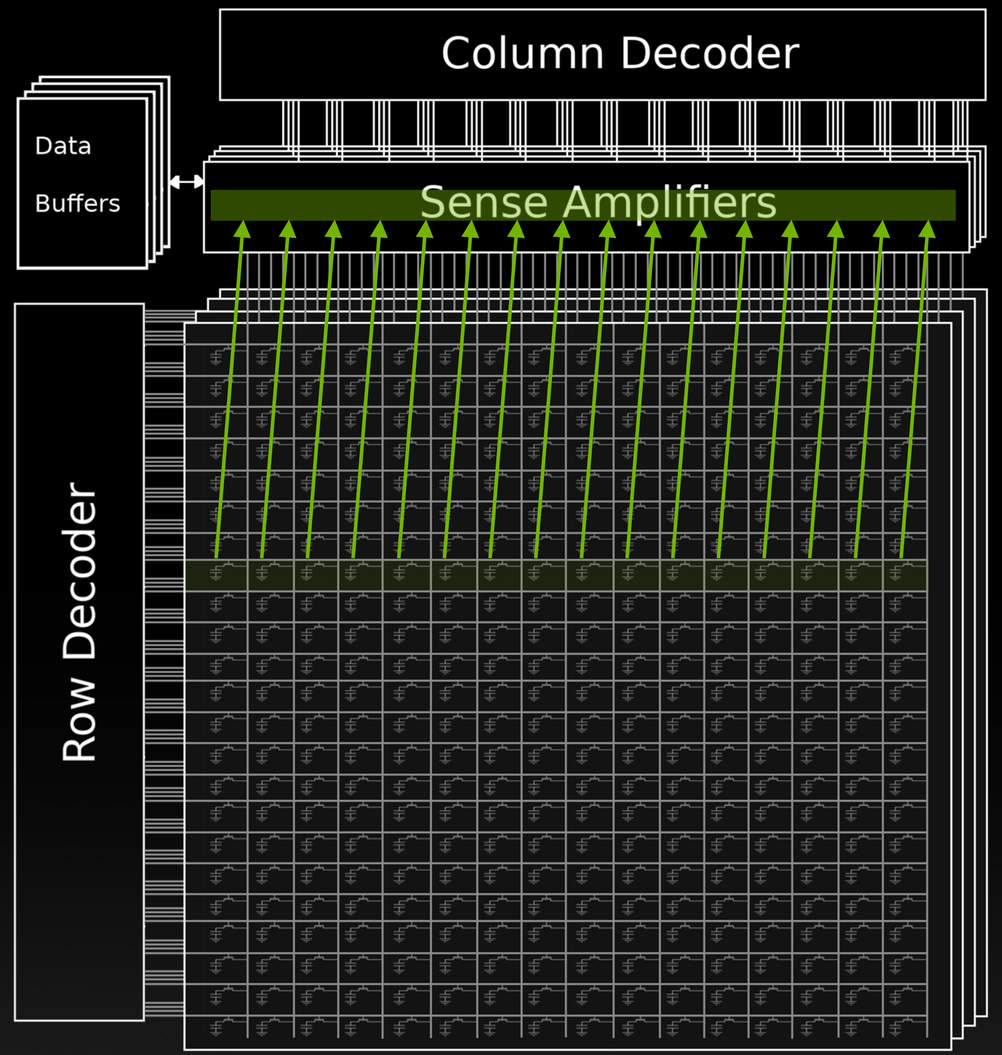
Your milage will very

Depending on how you access memory will greatly affect bandwidth!

Why?

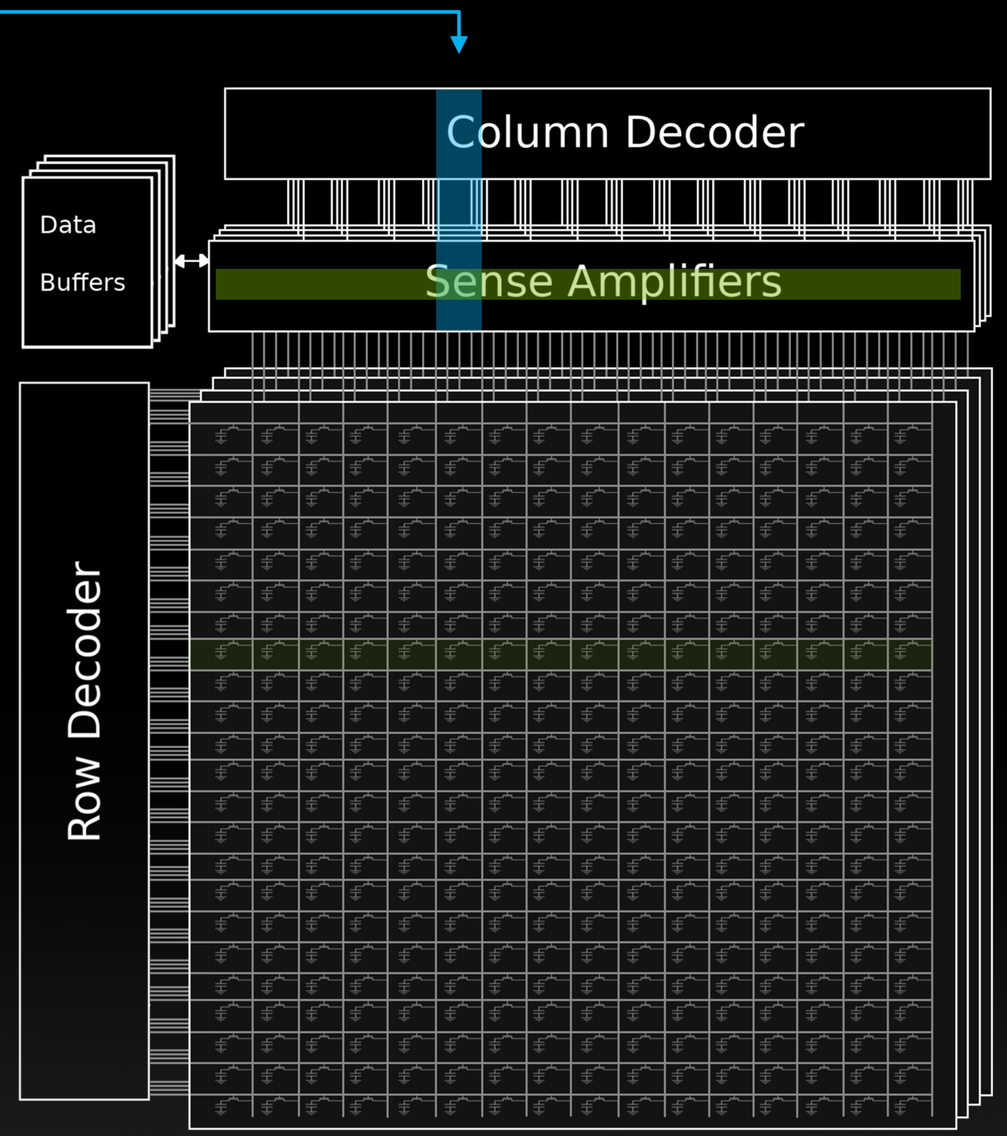


Read address: 00110001001001111010000110110111001 1



1 Activate row and pull data into sense amplifiers
This destroys data in the row as capacitors drain

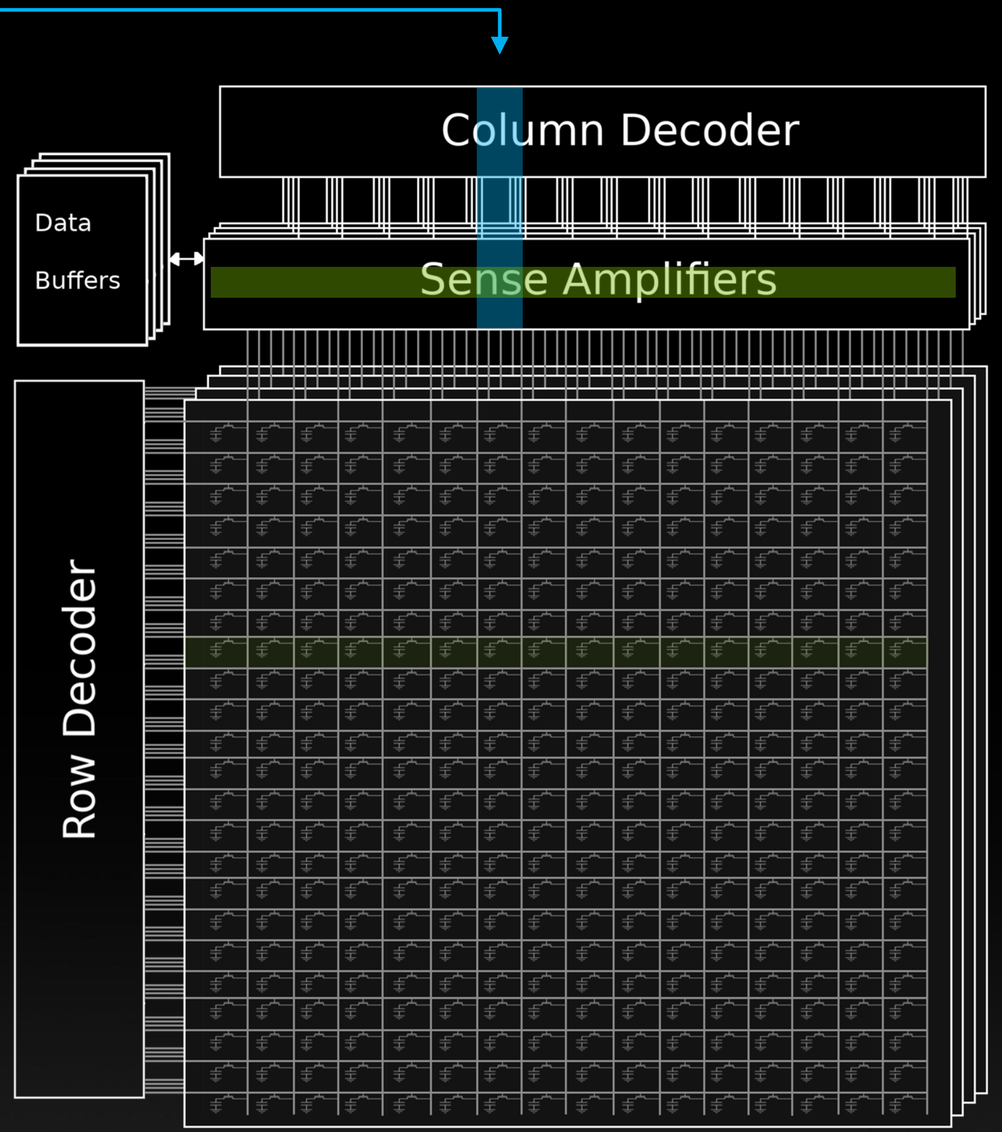
Read address: 00110001001001111010000110110111 0011



1. Activate row and pull data into sense amplifiers
   This destroys data in the row as capacitors drain

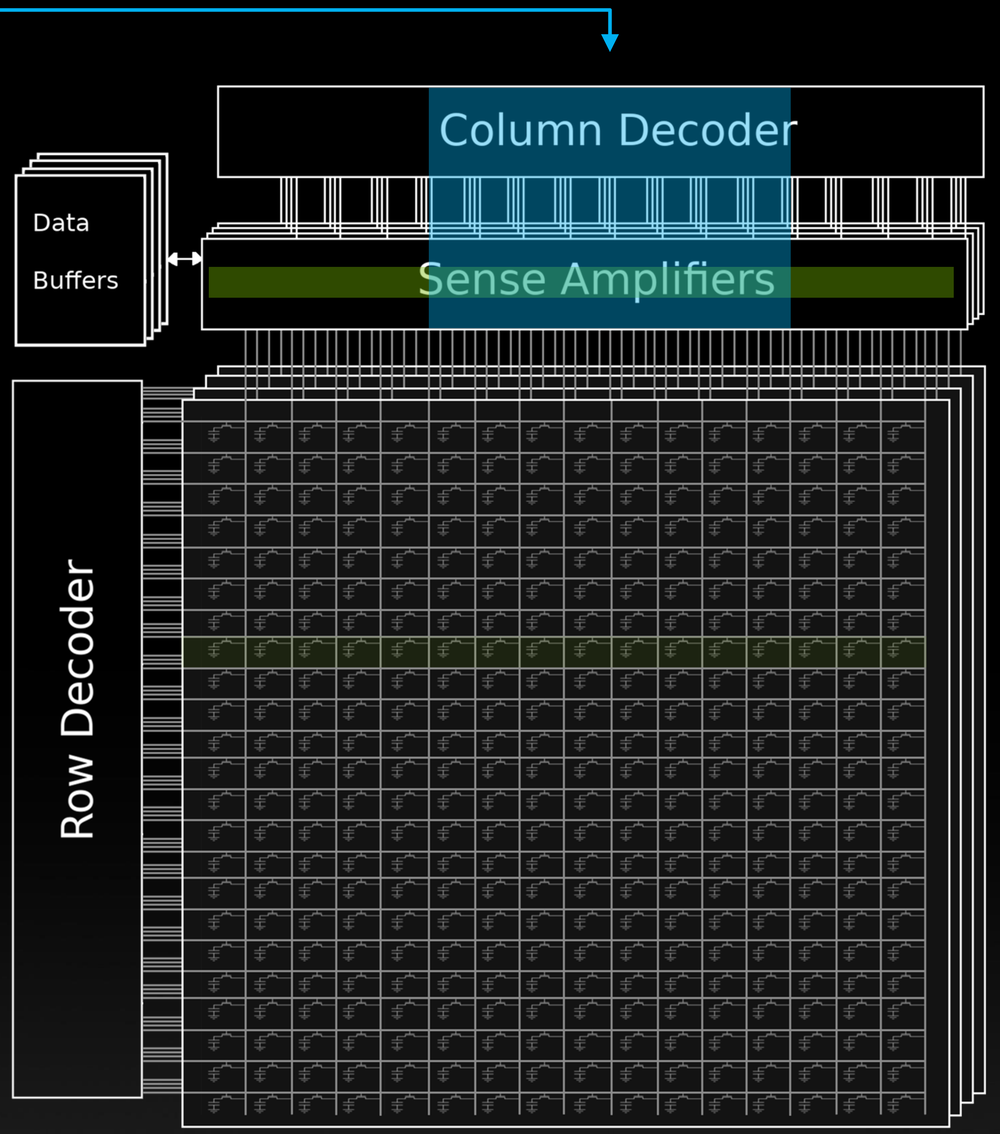
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers

Read address: 00110001001001111010000110110111010**0**



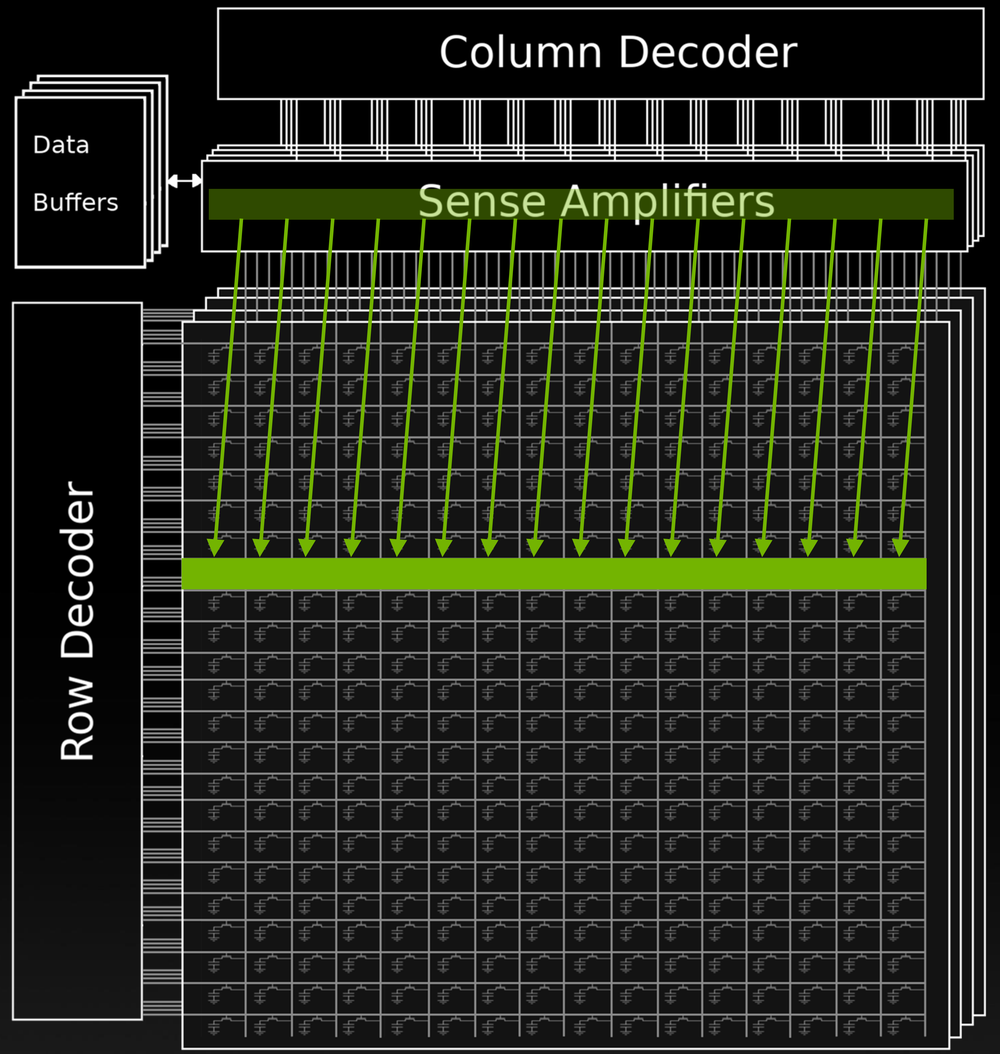
1. Activate row and pull data into sense amplifiers
   This destroys data in the row as capacitors drain
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers
3. May make repeated reads from the same page
   at different column indexes

Read address: 00110001001001111010000110110111010**0**



1. Activate row and pull data into sense amplifiers
   This destroys data in the row as capacitors drain
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers
3. May make repeated reads from the same page
   “Burst” reads load multiple columns at a time

Read address: 00110001001001111011000011011011110011



1. Activate row and pull data into sense amplifiers
   This destroys data in the row as capacitors drain
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers
3. May make repeated reads from the same page
   “Burst” reads load multiple columns at a time
4. Before a new page is fetched, old row must be
   written back because data was destroyed

SO WHAT DOES THIS ALL MEAN?

- We'd expect a significant performance difference for coalesced vs. scattered reads

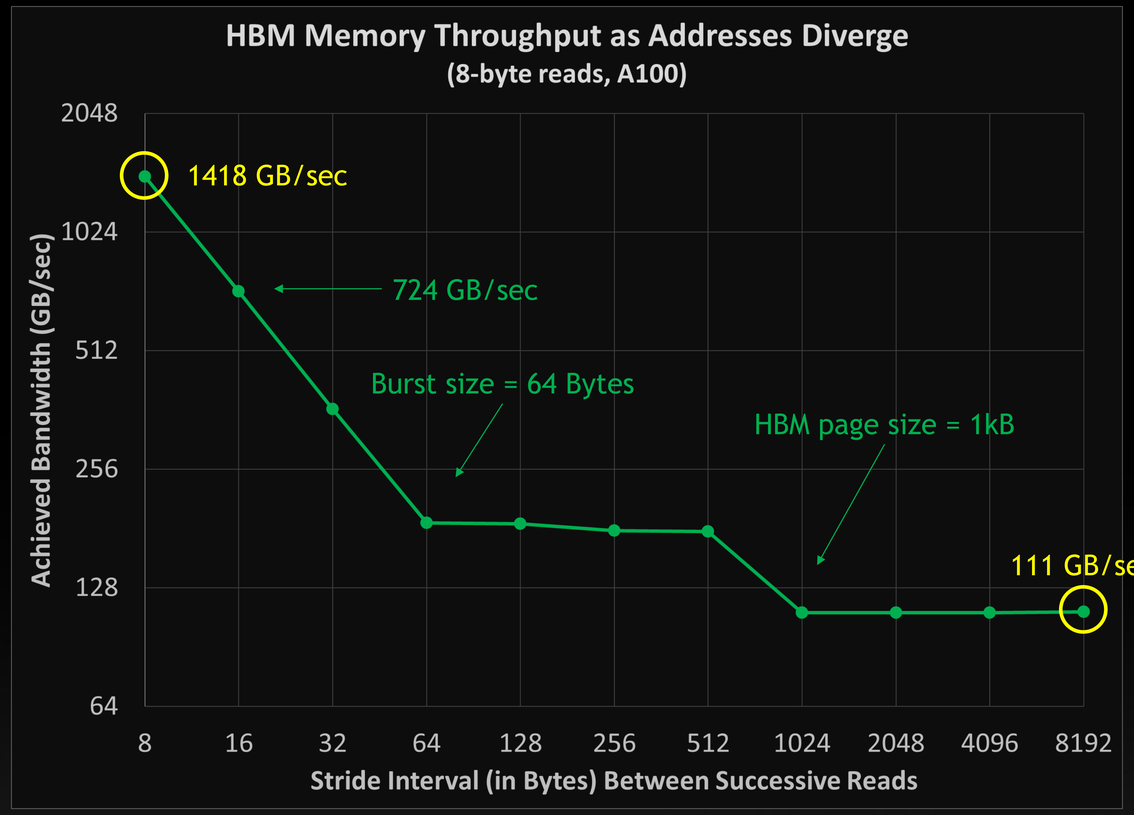
- On A100, memory bandwidth for widely-spaced reads is

$$\frac{111}{1418} = 8\% \text{ of peak bandwidth}$$

That's 1/13th of peak bandwidth!

ARITHMETIC INTENSITY=9.7/0.111=88

We need to use every load 700x

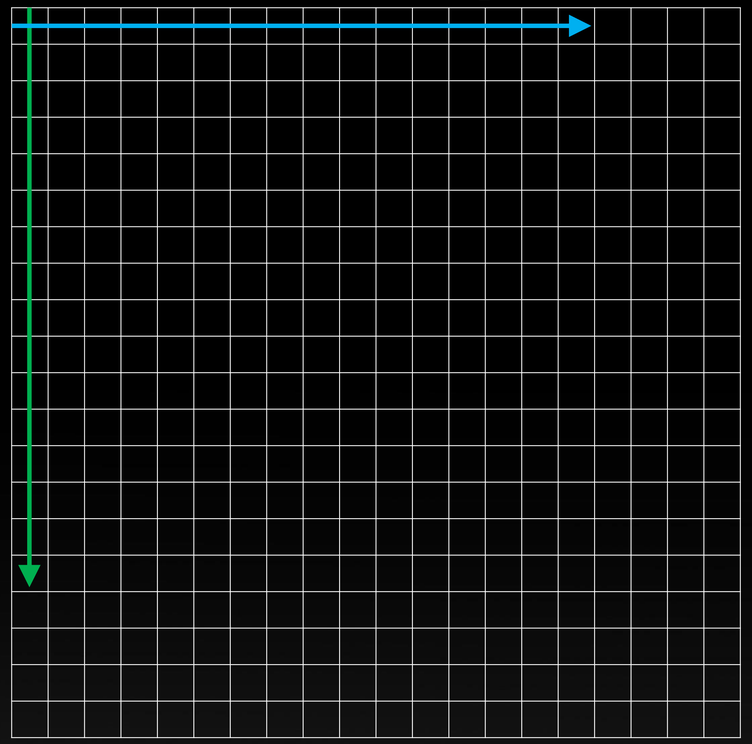


DATA ACCESS PATTERNS REALLY MATTER

```
for(y=0; y<M; y++) {
    for(x=0; x<N; x++) {
        load(array[y][x]);
    }
}
```

Row-major array traversal

Column read latency C_L



Row-major array layout

Row read latency

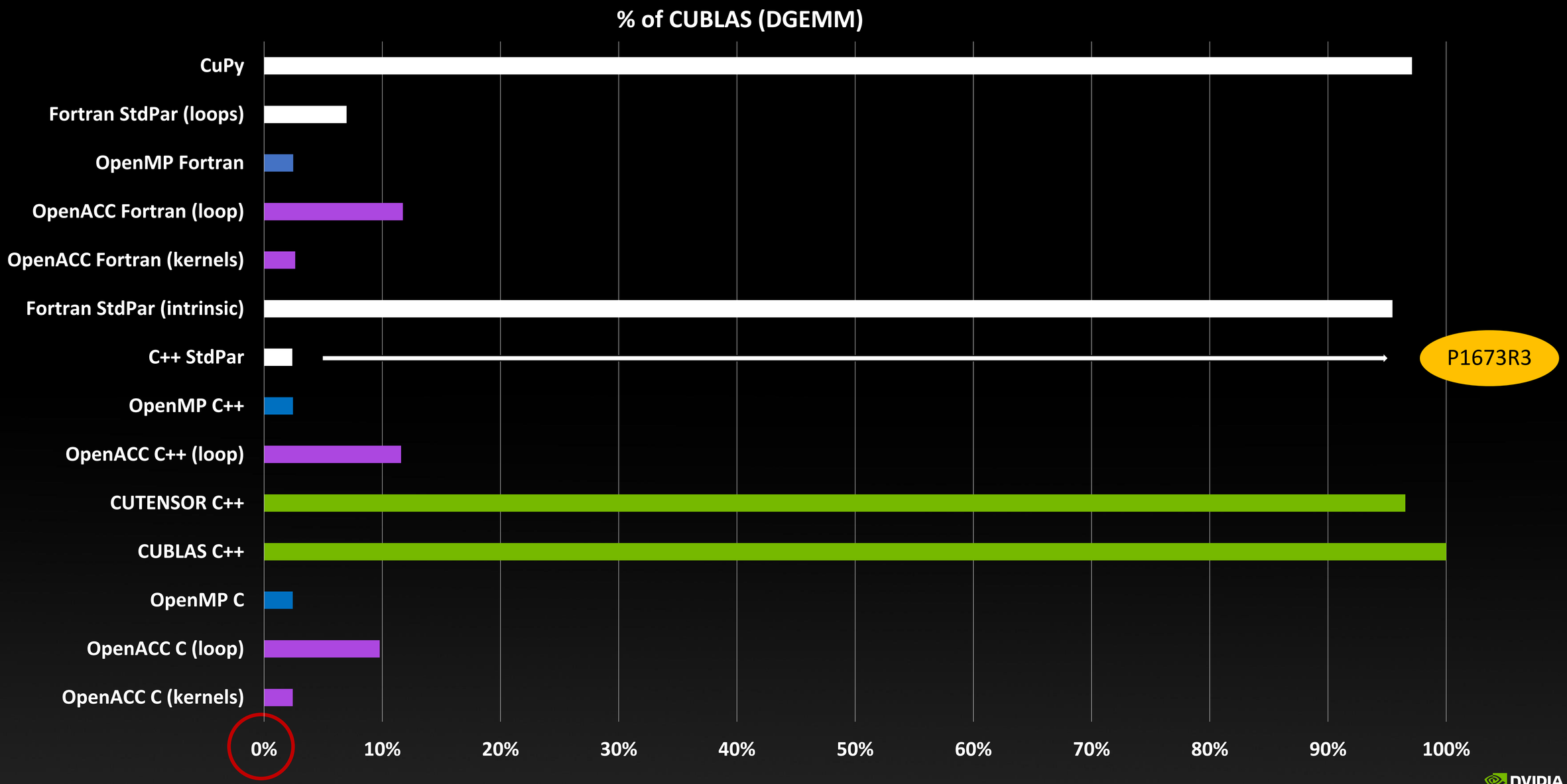
$T_{RAS} = T_{RP} + T_{RDC} + C_L$

13x slower than column access

```
for(x=0; x<N; x++) {
    for(y=0; y<M; y++) {
        load(array[y][x]);
    }
}
```

Column-major array traversal

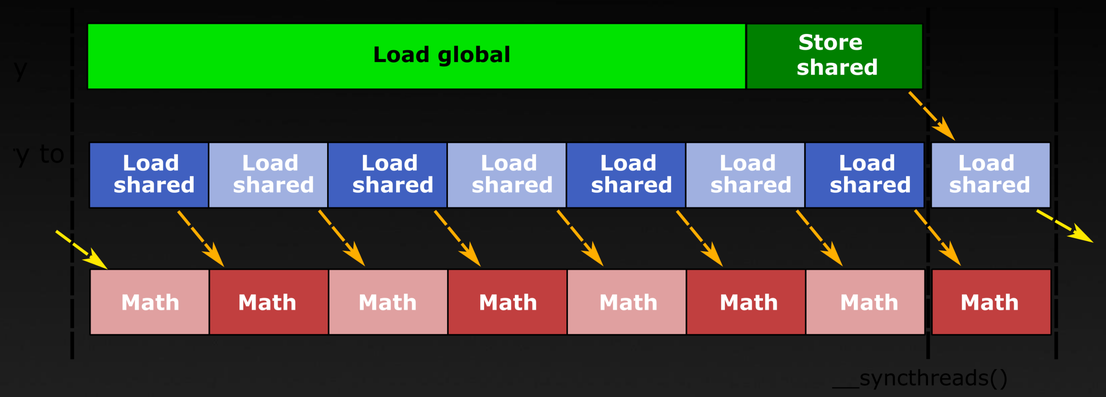
SO WHAT WENT WRONG?
Matrix Multiplication: C = C + A * B

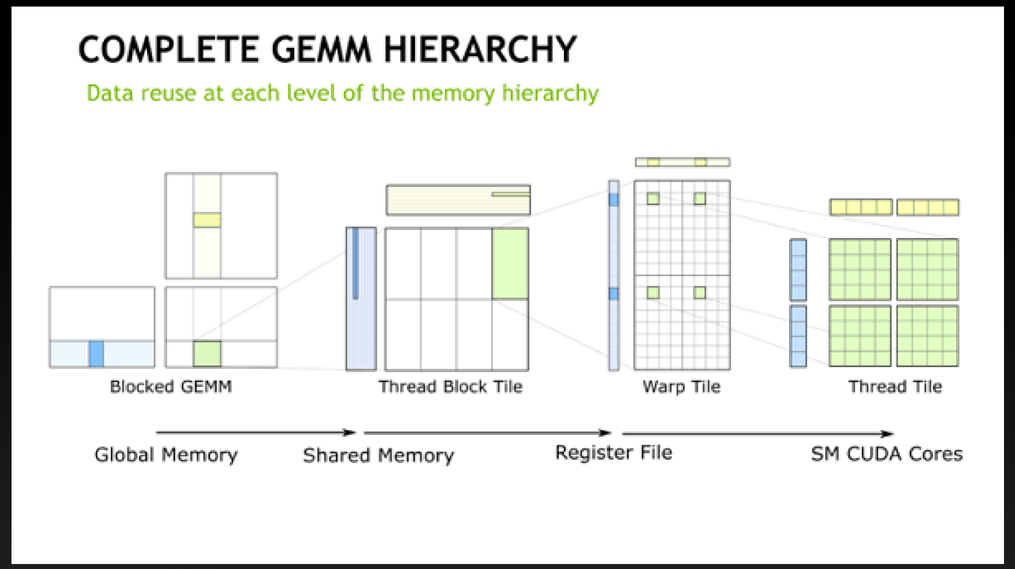


IT IS ABOUT MEMORY ACCESS

- The simple FORTRAN stdpar code is listed
  - The B matrix has good memory access
  - The A matrix has strided access
- What the cuBLAS library does for a matrix multiply
  - Divides up the A and B matrix into blocks
  - Loads these blocks into shared memory
  - Load from shared memory into registers
  - Perform unrolled math using registers
  - Store results
- Loads are all handled asynchronously
- Modern versions use tensor cores for the math

```
! Loop version
do concurrent (j=1:order, i=1:order) local(T)
  T = C(i,j)
  do concurrent (p=1:order) ! Implicit reduction
    T = T + A(i,p) * B(p,j)
   enddo
   C(i,j) = T
enddo
```





WHAT DO WE KNOW SO FAR

GPU Programming is easy, just…

Load as little data as possible

Access the data so it is adjacent for optimal bandwidth

Reuse the data a lot of times

i.e., Perform dense matrix-matrix multiplies

But my program isn't a matrix-matrix multiply

My mesh is unstructured or my data access is random

MEMORY LATENCY

Latency – Time between your first request and the data arrives

Bandwidth – How much data you get in a given time once the transfer starts

Low Latency (left)

Or

High bandwidth (right)

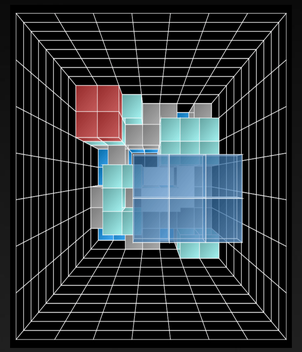
GPUs have very high bandwidth compared to CPUs (1.6GB/s vs 0.2 GB/s)

But also have higher latency than CPUs (400ns vs 100ns)

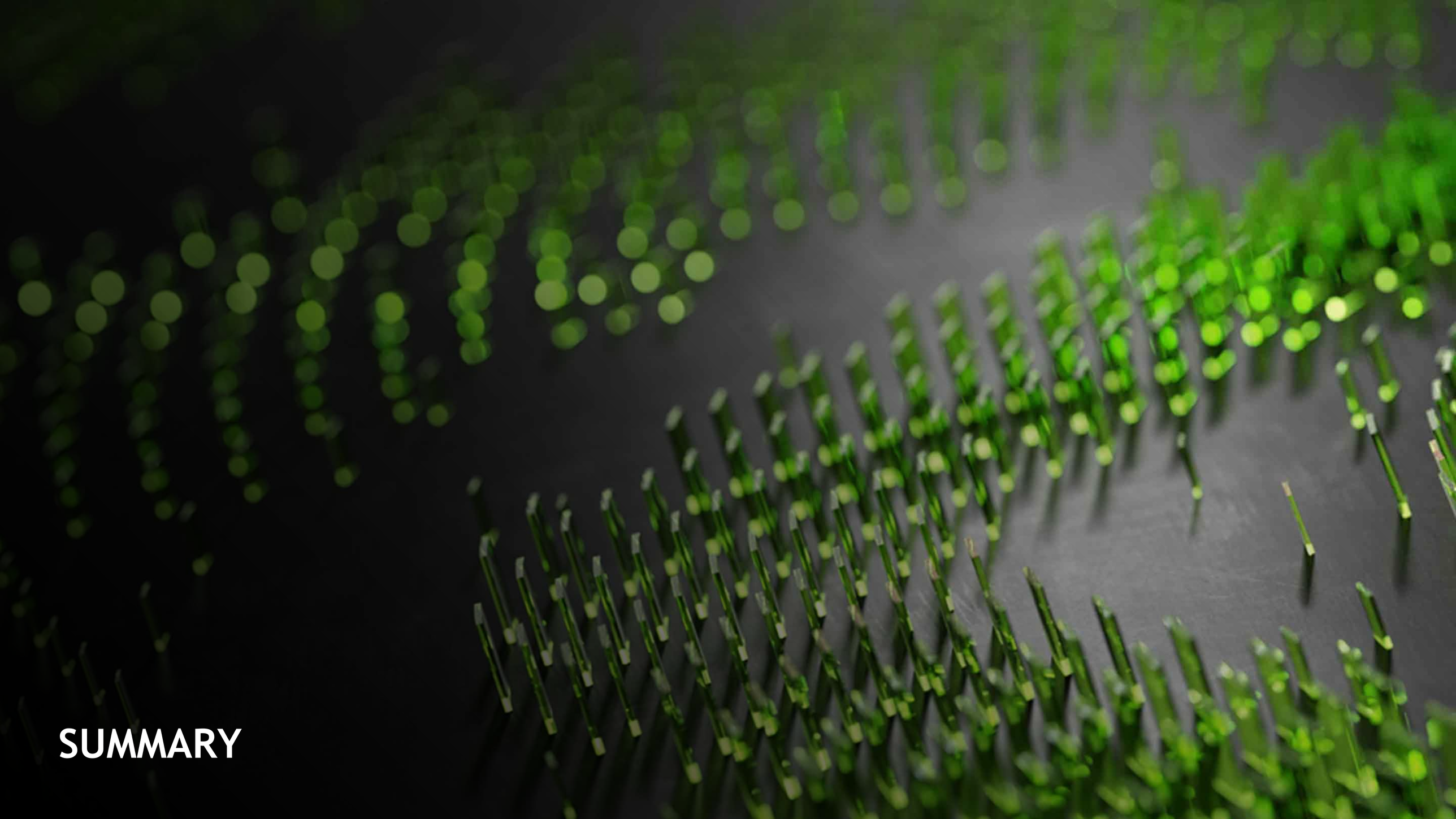
WORKING WITH HIGH LATENCY

Over-Subscription and Concurrency

- Remember that scheduler I mentioned??? It schedules work on an SM
- Fits as many blocks as it can based on resources. If it schedules 2048 threads occupancy is at 100%
- If a chunk of threads (warp) gets stalled while waiting for memory, another gets swapped in who is ready
- What can you do?
- Schedule multiple types of work
  - Fetch data and FLOPS
- Reduce resources
  - Registers and shared memory
- It is always a good idea to
  - Have multiple blocks on an SM
  - Ideally a mix of work
  - Use your shared memory wisely



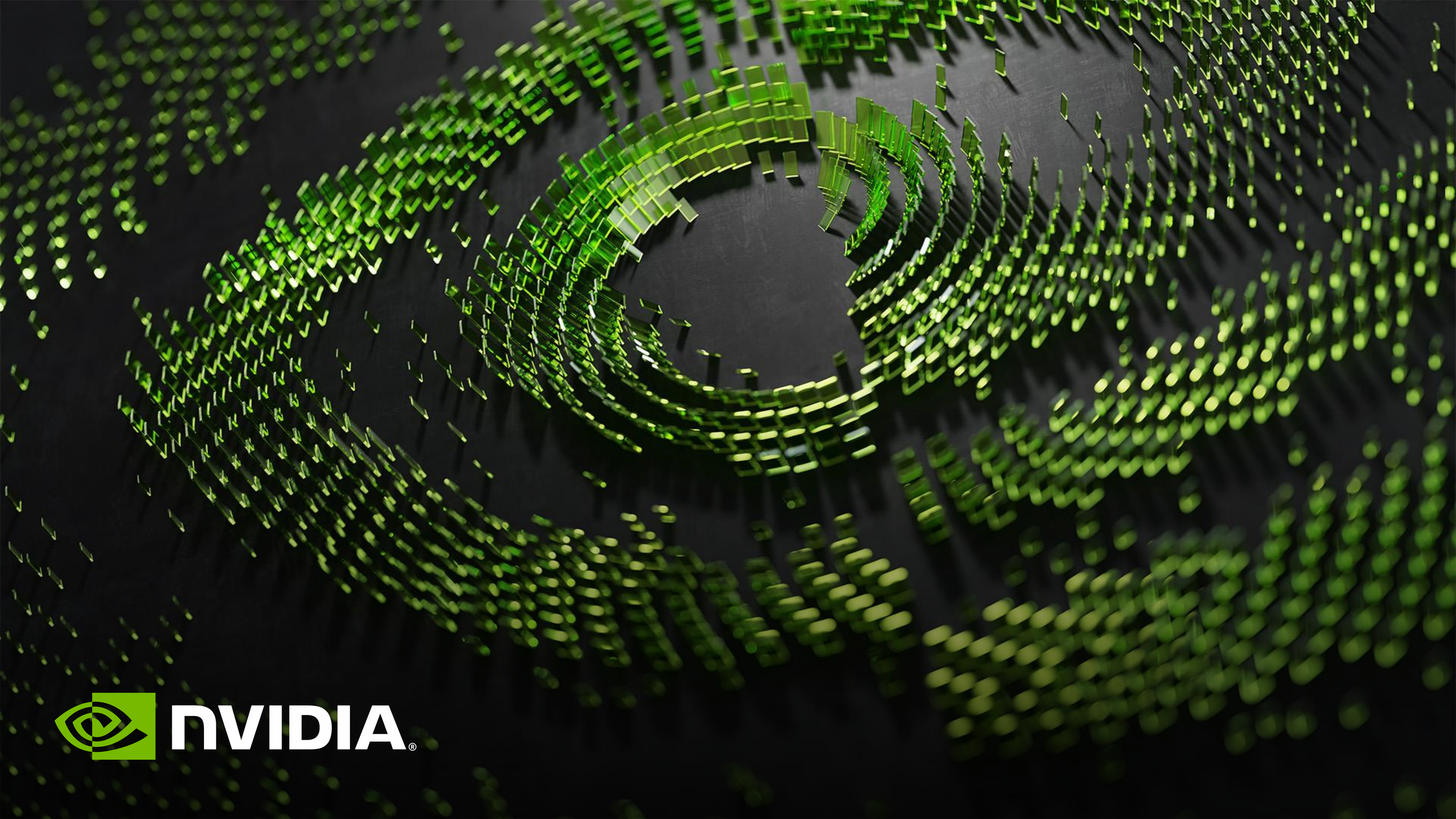
A100 SM Resources	
2048	Max threads per SM
32	Max blocks per SM
65,536	Total registers per SM
160 kB	Total shared memory in SM
32	Threads per warp
4	Concurrent warps active
64	FP32 cores per SM
32	FP64 cores per SM
192 kB	Max L1 cache size
90 GB/sec	Load bandwidth per SM
1410 MHz	GPU Boost Clock



SUMMARY

IN SUMMARY

- Computing has changed a lot in the last 50 years, and it will continue to change
- As computing has evolved, complexity has grown, and the tools have evolved to make this tractable
- GPUs are here with us, they are not going anywhere
- Programming GPUs (and CPUs really) one needs to focus on the memory access and use patterns
- Think about memory access patterns when you design your algorithm
- When choosing a programming model, one needs to balance flexibility with performance
- Use libraries when possible, the designers of these libraries focus on the details I'd rather ignore
- Profile your code often throughout the development process, optimize accordingly


NVIDIA