TREX

Targeting Real chemical accuracy at the EXascale

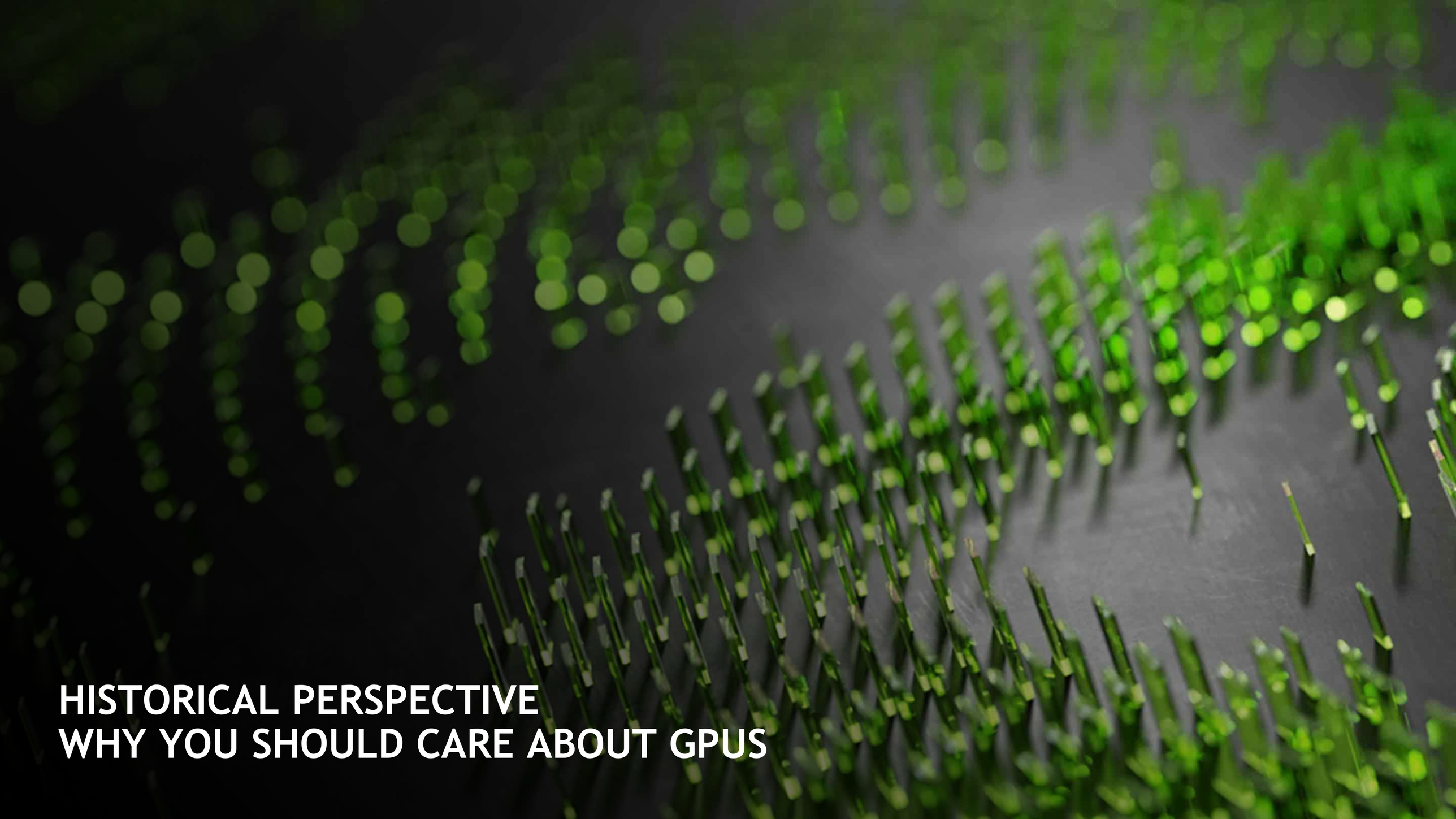
6-8 Mar 2023 | 9:00-17:00 CET

CINECA, Bologna, Italy

# Hackathon III at CINECA

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

MATT BETTENCOURT, DEVTECH

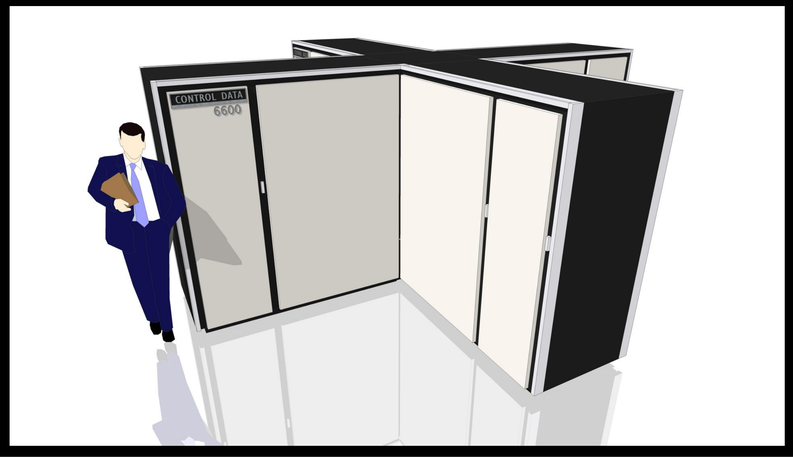


# HISTORICAL PERSPECTIVE
# WHY YOU SHOULD CARE ABOUT GPUS

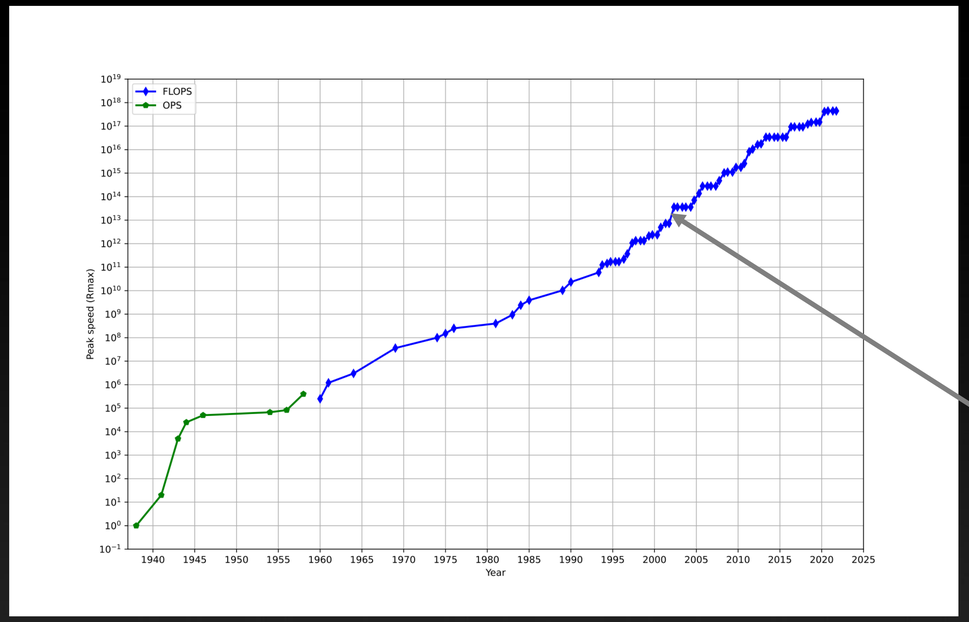
# A BRIEF HISTORY OF SUPERCOMPUTING

Three main generations of supercomputing

## YOUR GRANDPARENTS




## YOUR PARENTS




NVIDIA
A100 GPU
FP64


## YOUR GENERATION

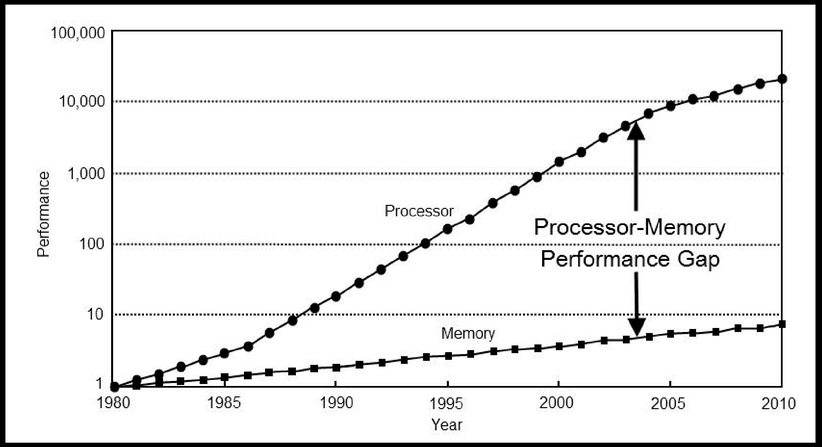
By Morn - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=32719361

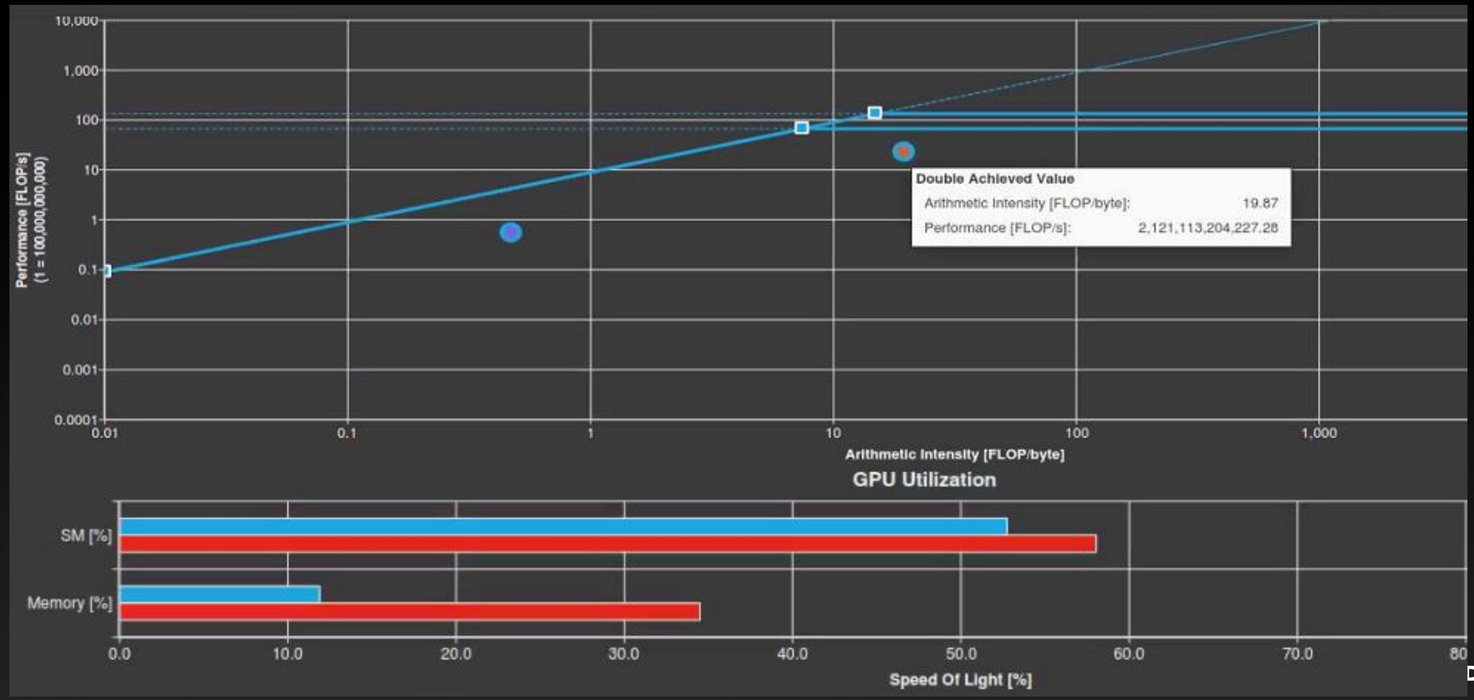
NVIDIA

# A TREND THROUGH HISTORY

Roofline model as a way to understand performance

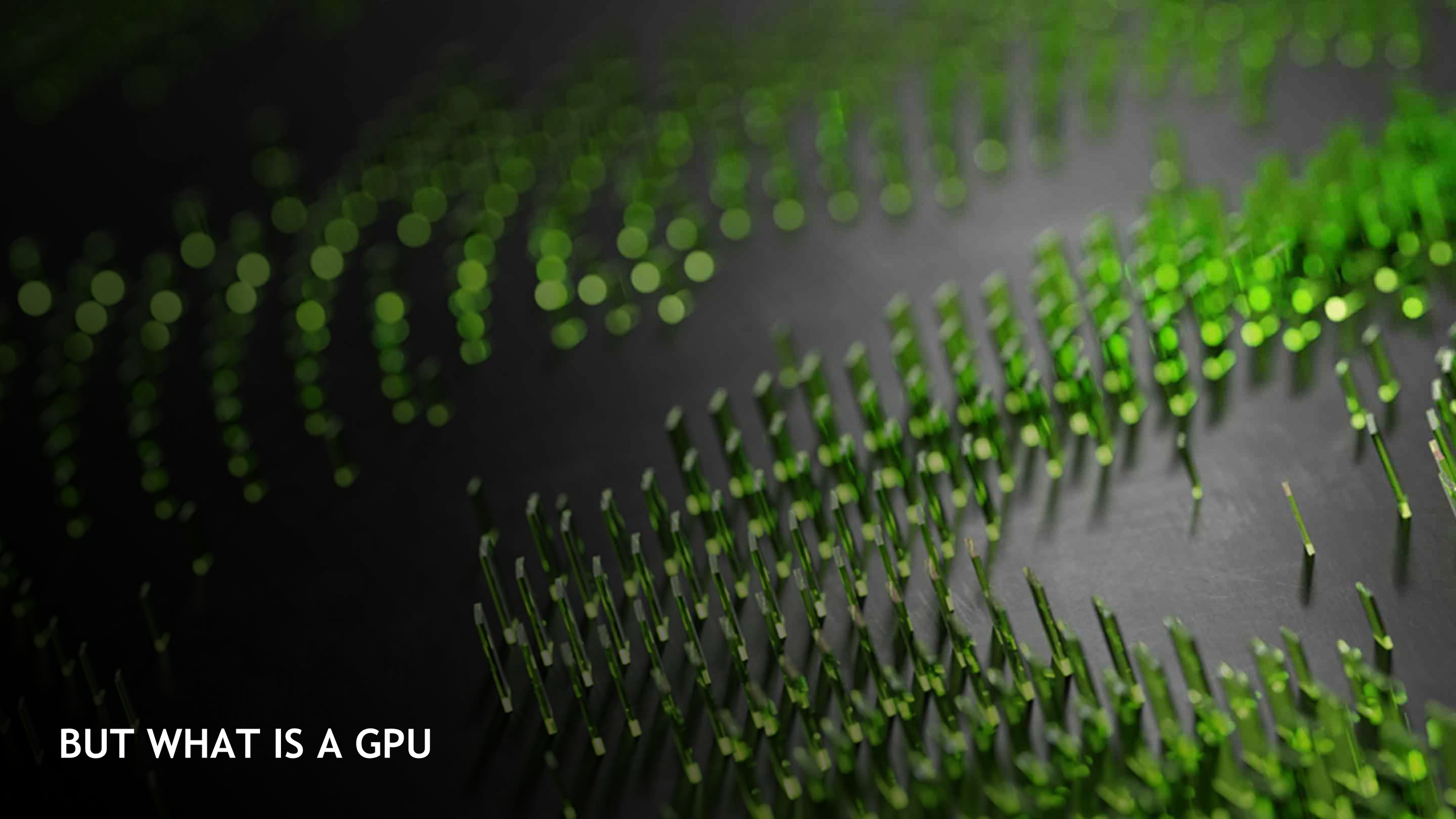
- In the origins of computing memory access was free, all the cost was in FLOPS
- Today, FLOPS are (mostly) free once you have the data on the CPU/GPU
- Roofline models are hardware specific plots of potential and achieved performance
  - Peak performance is plotted against “arithmetic intensity”
  - 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?

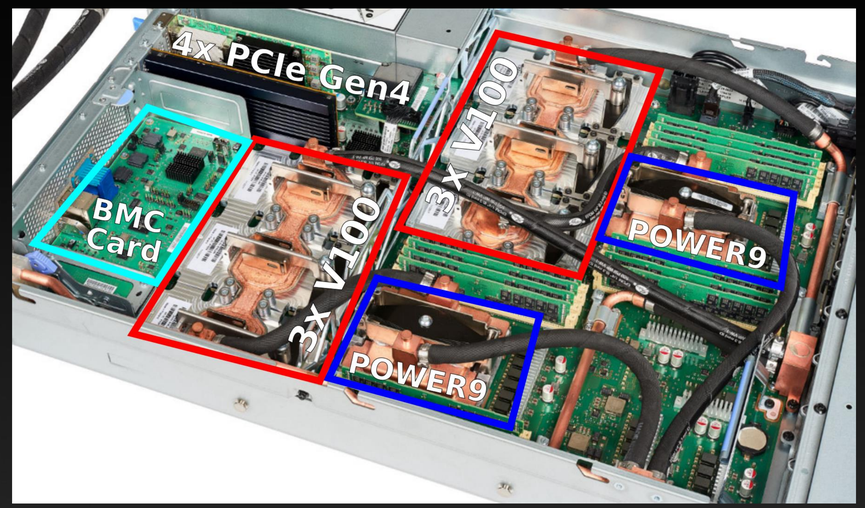
## GPUs are about concurrency

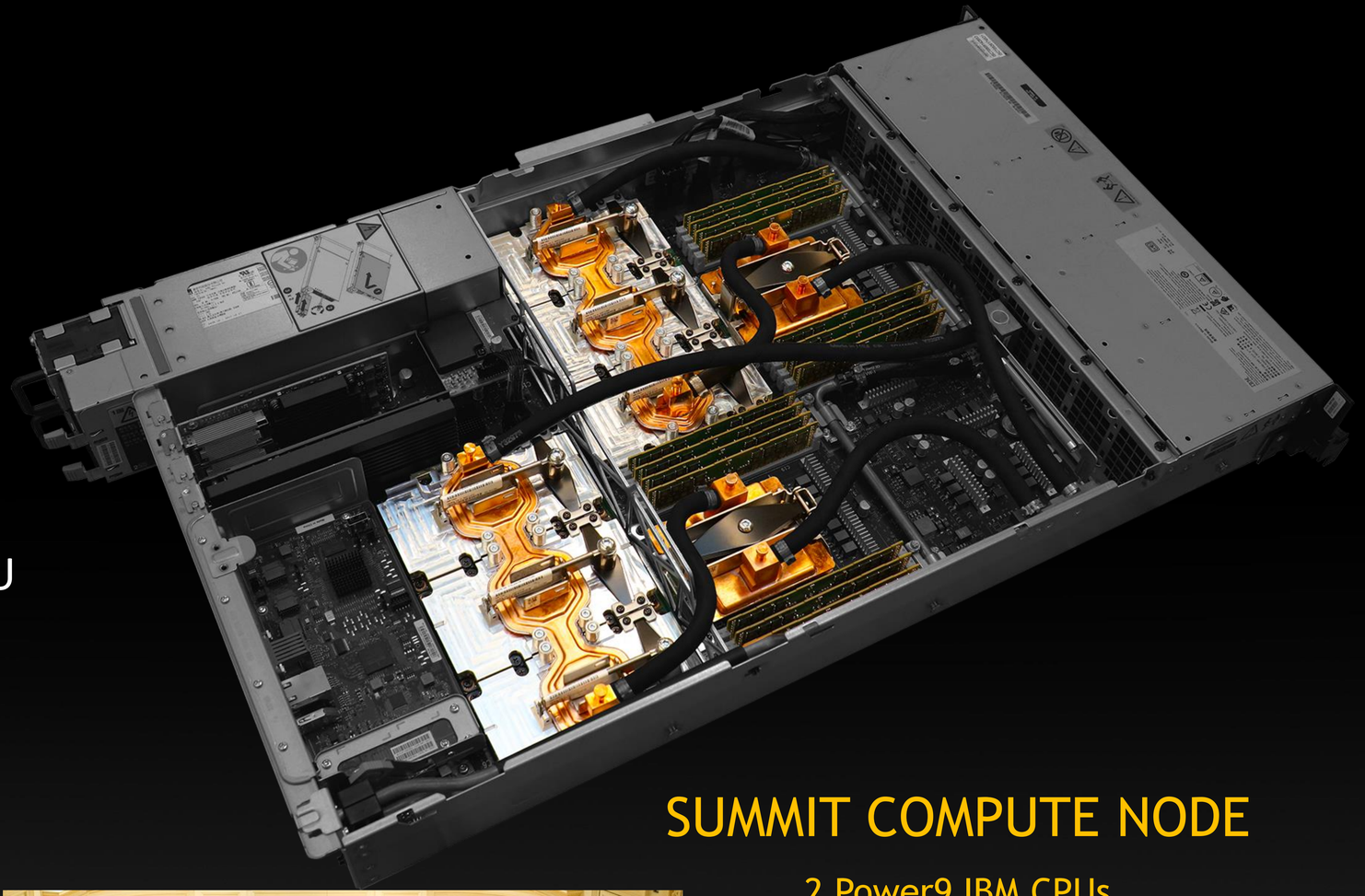
Many independent tasks operating at the same time

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

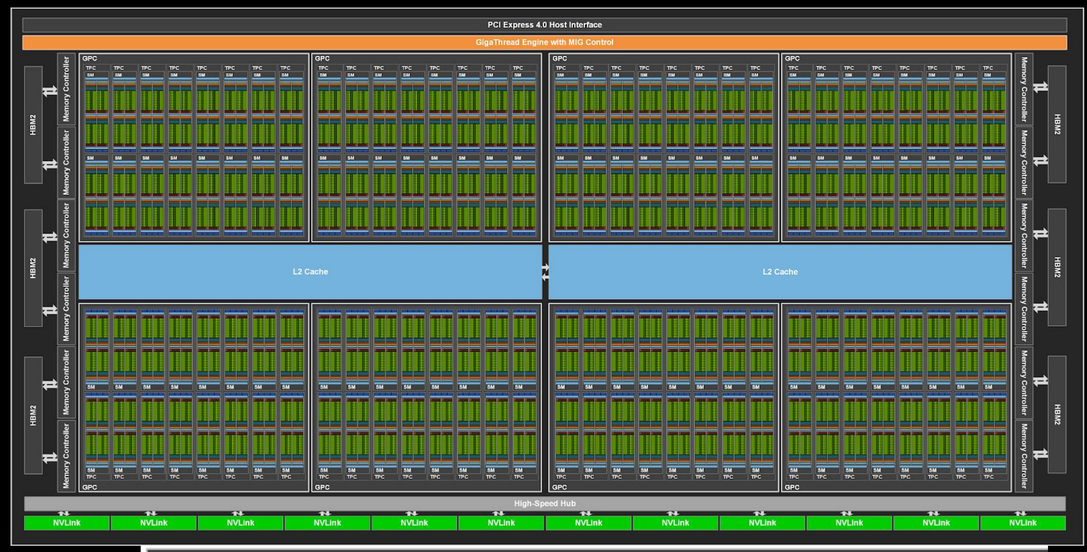
Mem = 108 SMs*32 Threads*1.41GHz *8B = 78TB/s

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

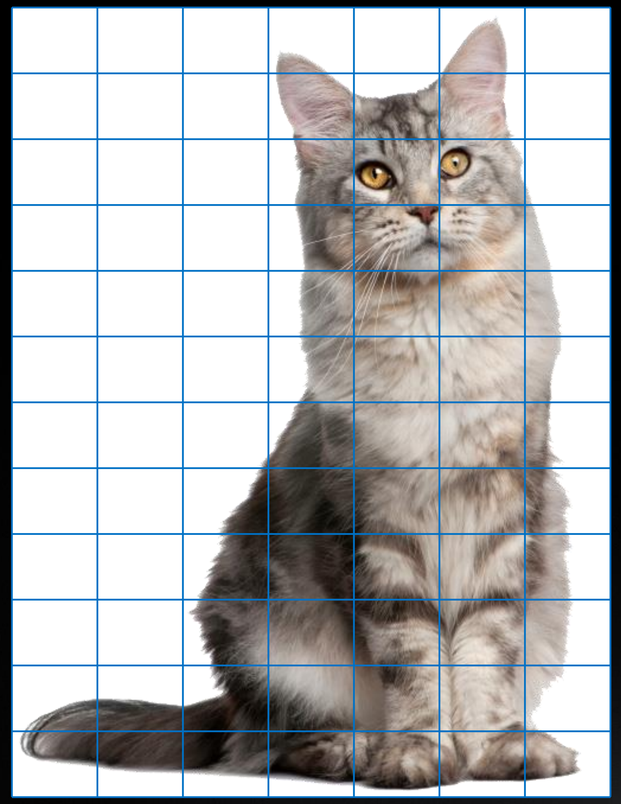
Designed for SIMT

GPUs are most efficient at a particular type of work

# TAKE THIS CAT IMAGE

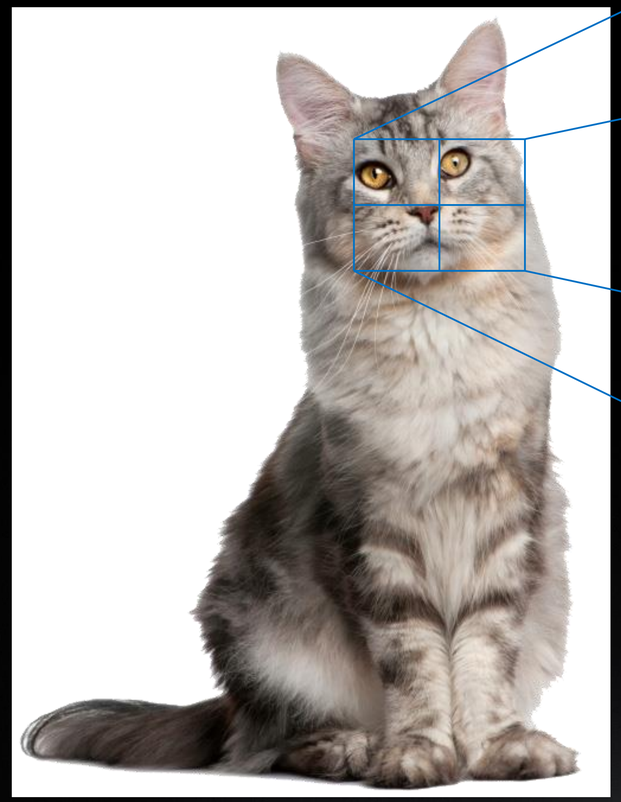
Let’s improve this image

# 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



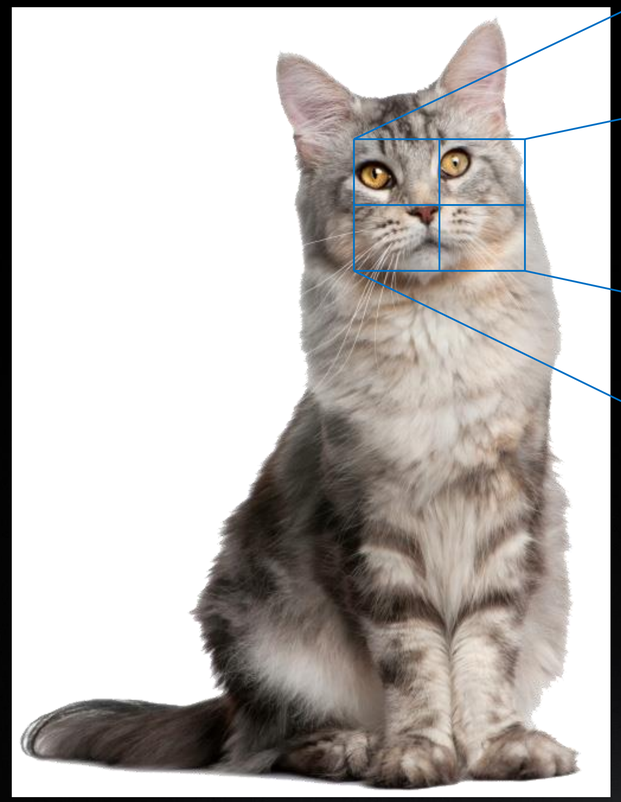
1. Overlay with a grid

2. Operate on blocks within the grid

Blocks execute **independently**

GPU is **oversubscribed** with blocks

# EACH THREAD MODIFIES ITS PORTION

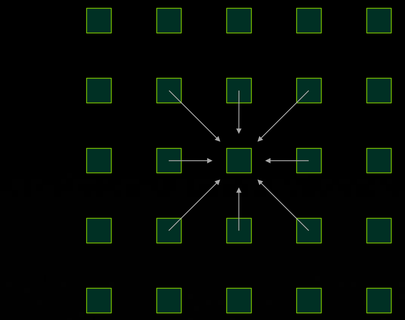


1. Overlay with a grid

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

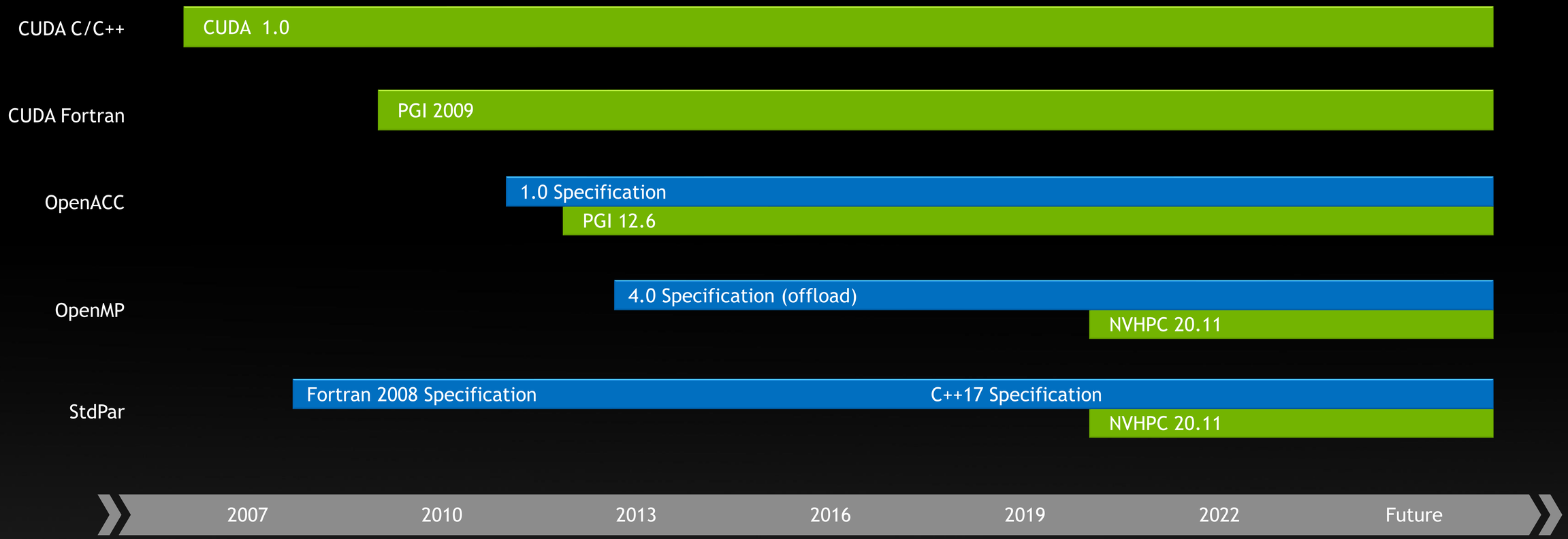

NOW THAT YOU CARE ABOUT GPUS
HOW TO USE THEM

# 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**

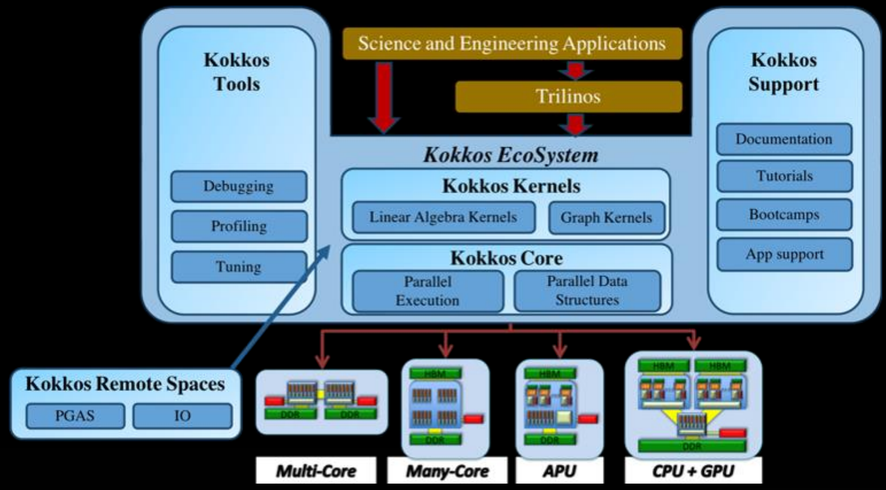
Directives and Pragmas: Easy to adopt. Good portability. Great performance in many use cases.

**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); }
);
```

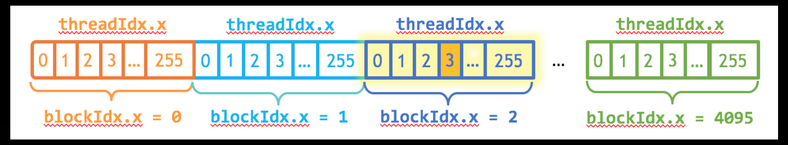
# SHIFTING THROUGH THE GEARS

Experiments with linear algebra primitives

## VECTOR ADDITION

Memory bandwidth

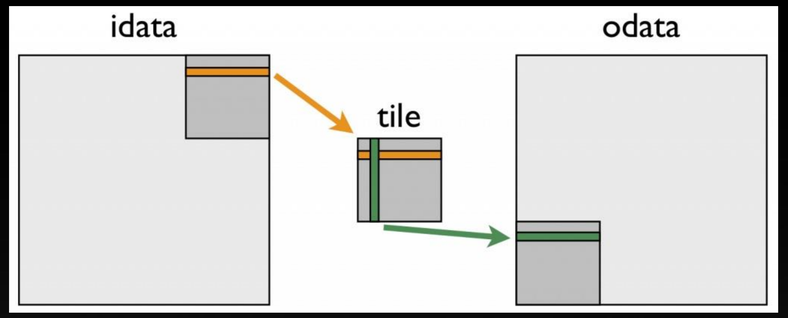
$$\forall i: \; Z_i = a \times X_i + Y_i$$




## MATRIX TRANSPOSE

Memory bandwidth, shared memory, and coalescing

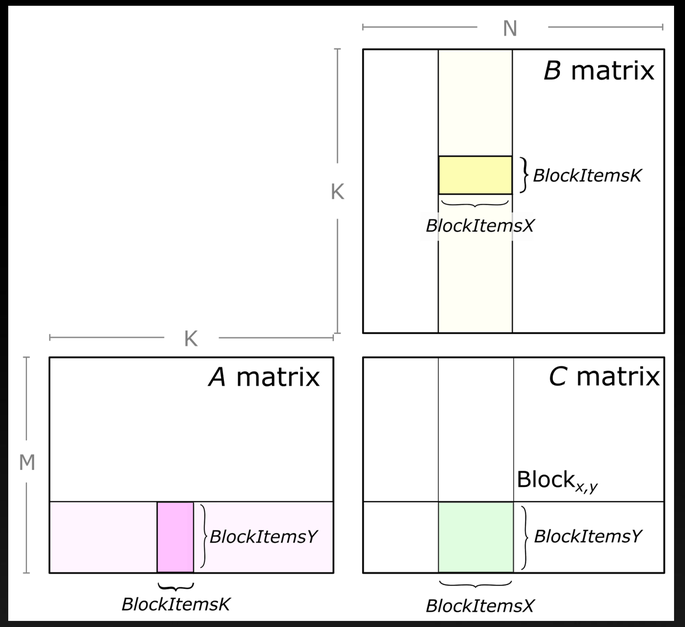
$$\forall i, j: \; B_{i,j} = B_{i,j} + A_{j,i}$$




## MATRIX MULTIPLICATION

Optimize everything…

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

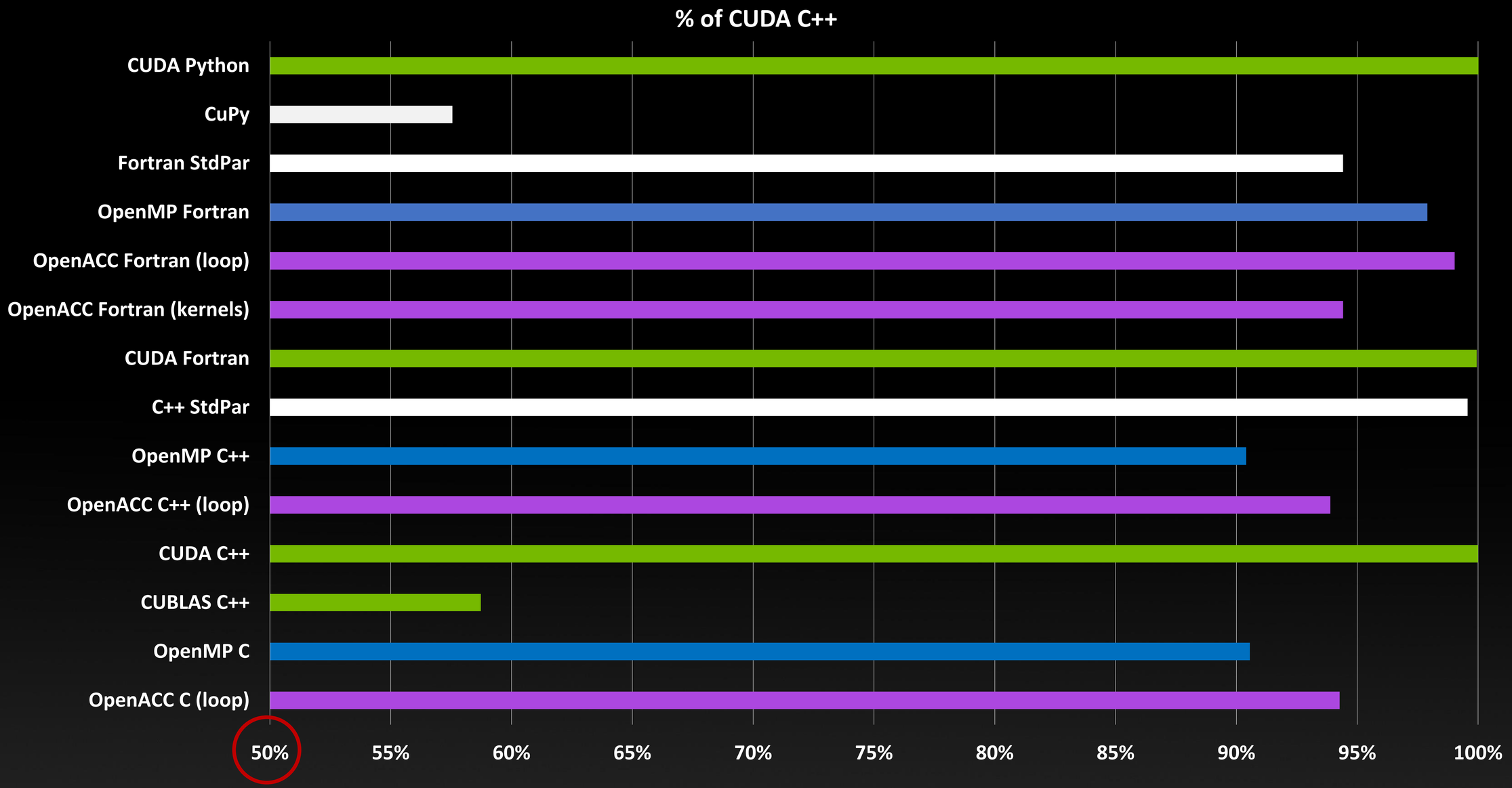



https://developer.nvidia.com/blog/even-easier-introduction-cuda/

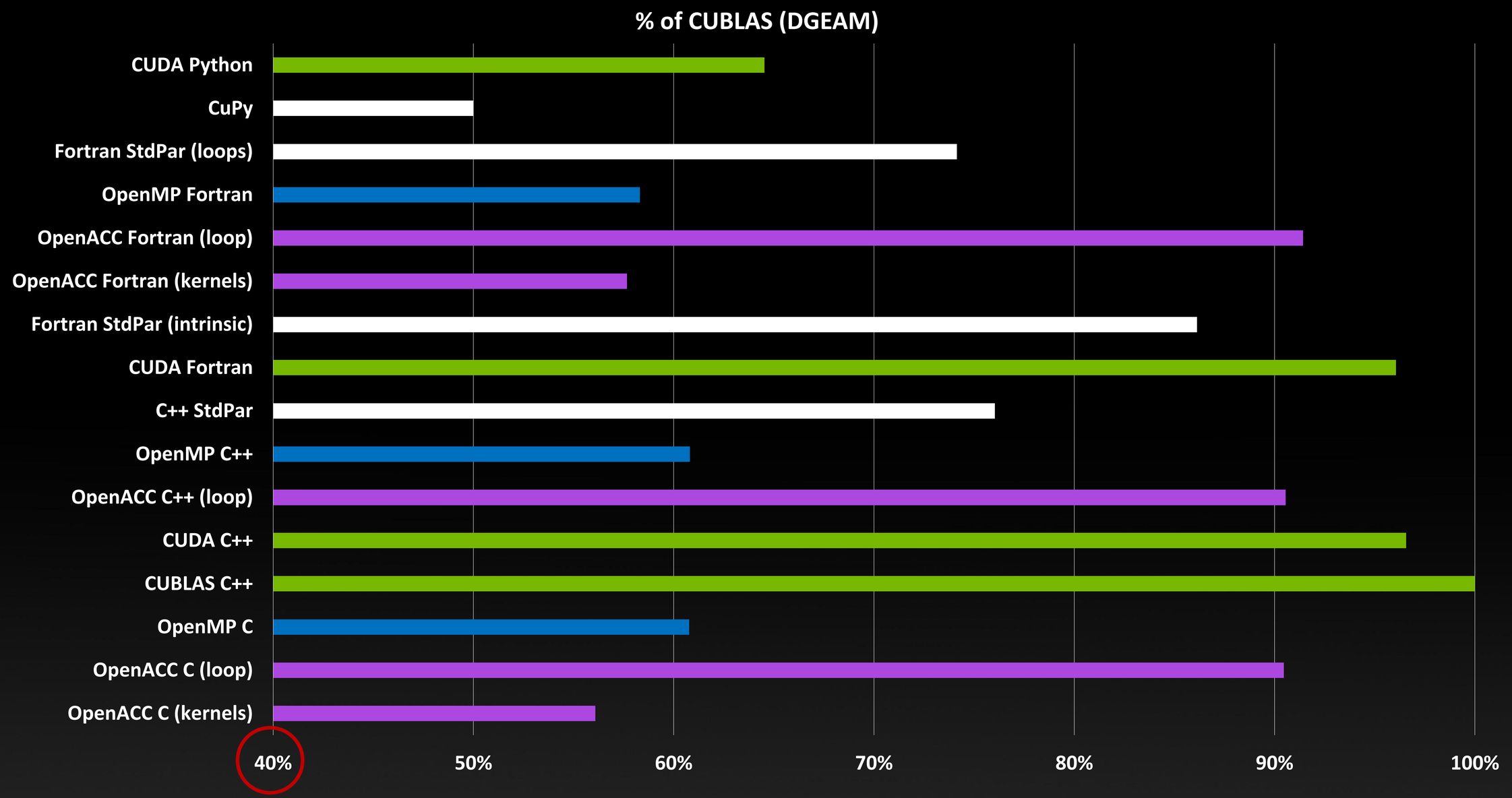
https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/

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

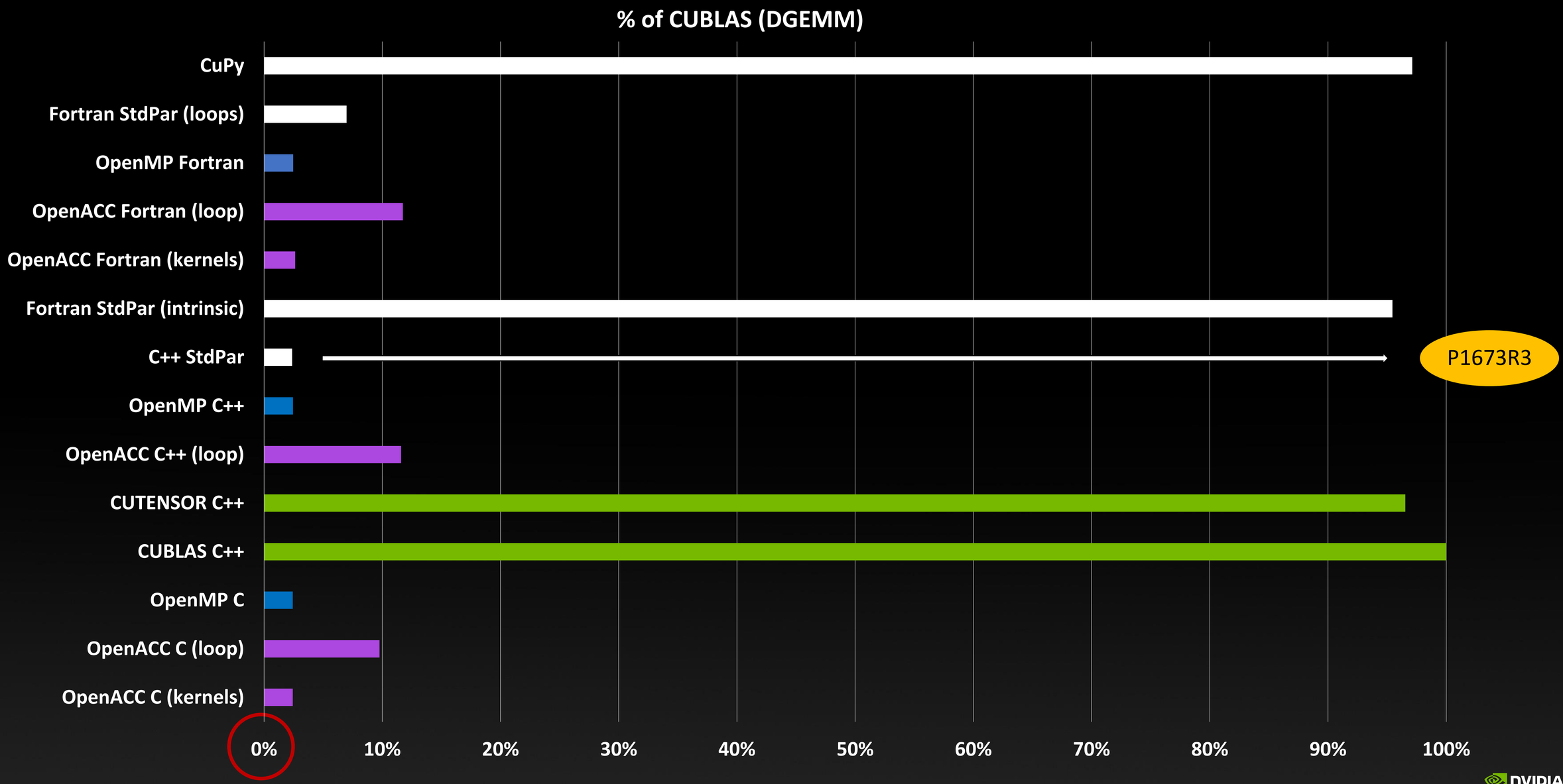
# Vector Addition: Z = a * X + Y



# Matrix Transpose: B = B + A^T



# Matrix Multiplication: C = C + A * B



# WHAT PARADIGM SHOULD YOU USE

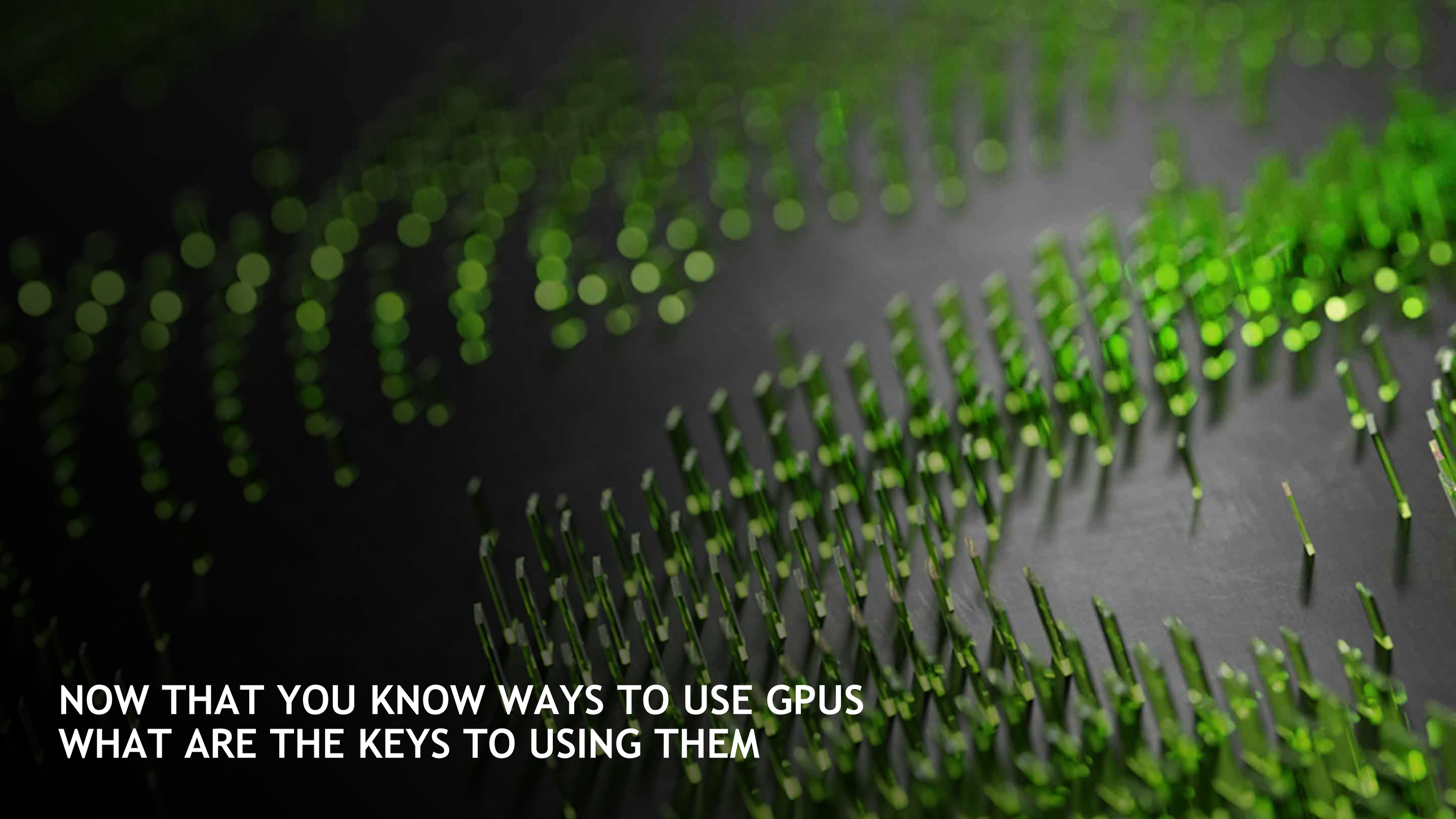
Well, it depends

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

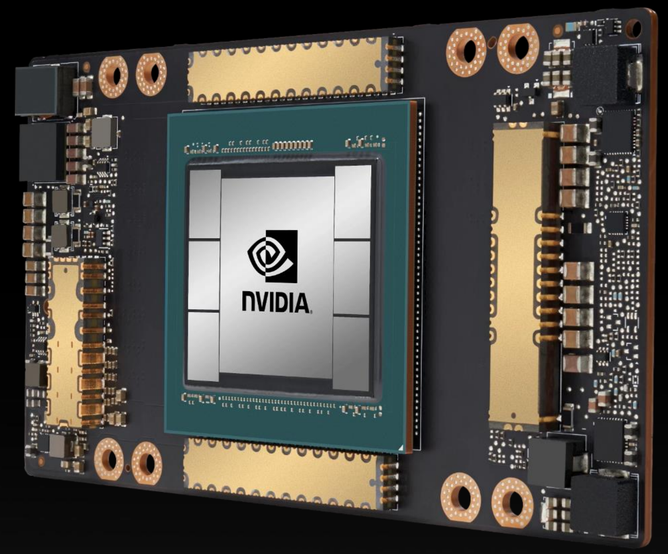
Shared Memory – Local fast shared memory to a SM

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 |

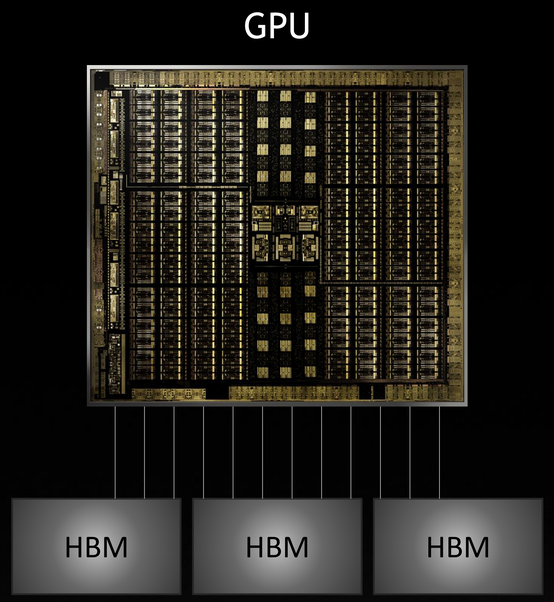


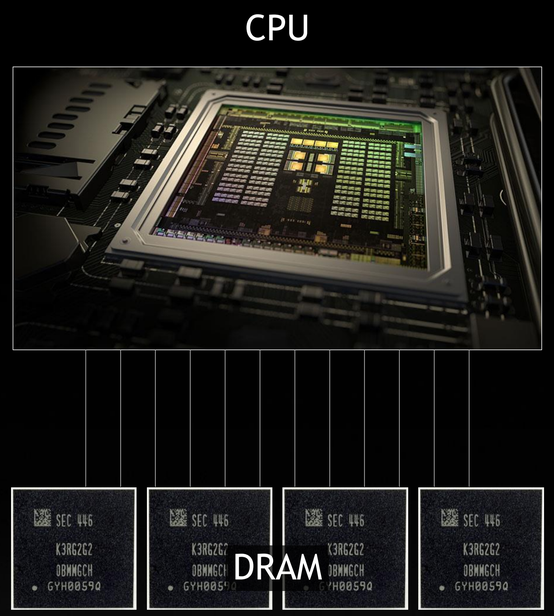

ARITHMETIC INTENSITY=9.7/1.555=6.25

Well, we want doubles, 8x!!

We need to use every load 50x

# NOT JUST A GPU ISSUE



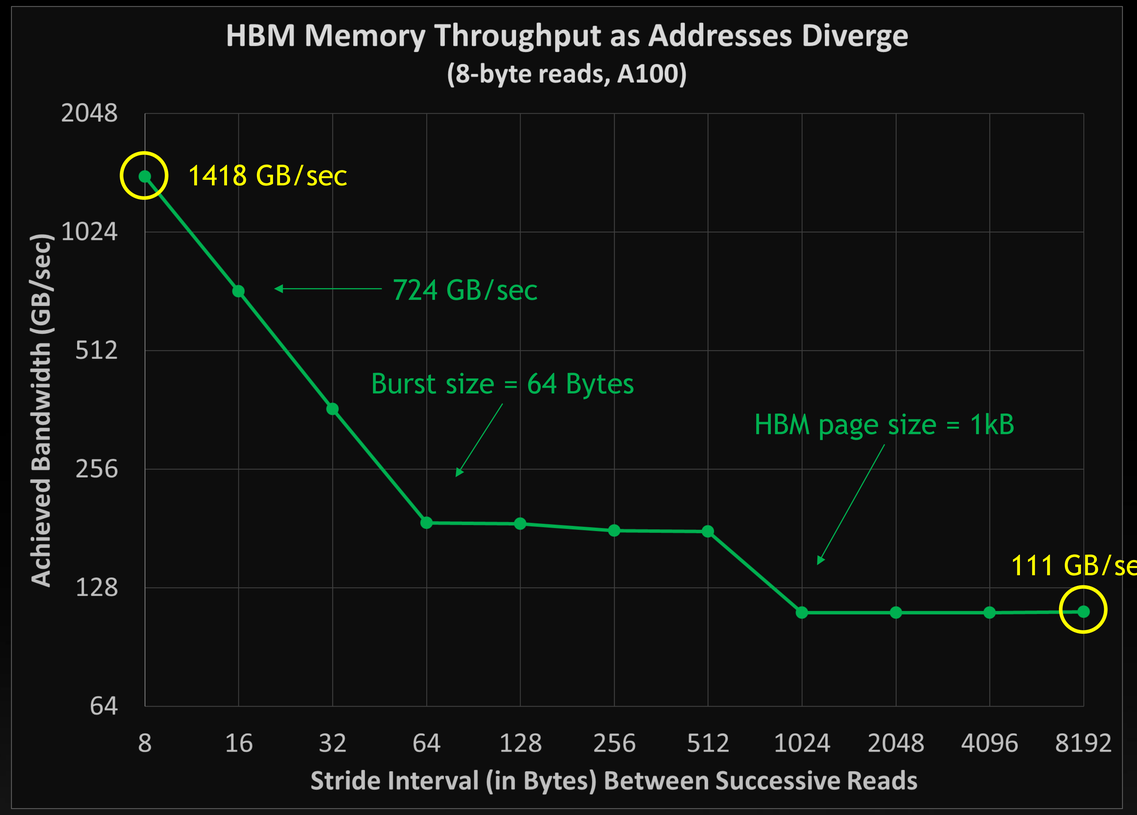

| | NVIDIA A100 | Intel Xeon 8280 | AMD Rome 7742 |
|---|---|---|---|
| Peak FP64 GigaFLOPs | 9700 | 2190 | 2300 |
| Memory B/W (GB/sec) | 1555 | 131 | 204 |
| Compute Intensity | 50 | 134 | 90 |

# THERE IS STILL A LOT OF MEMORY BANDWIDTH

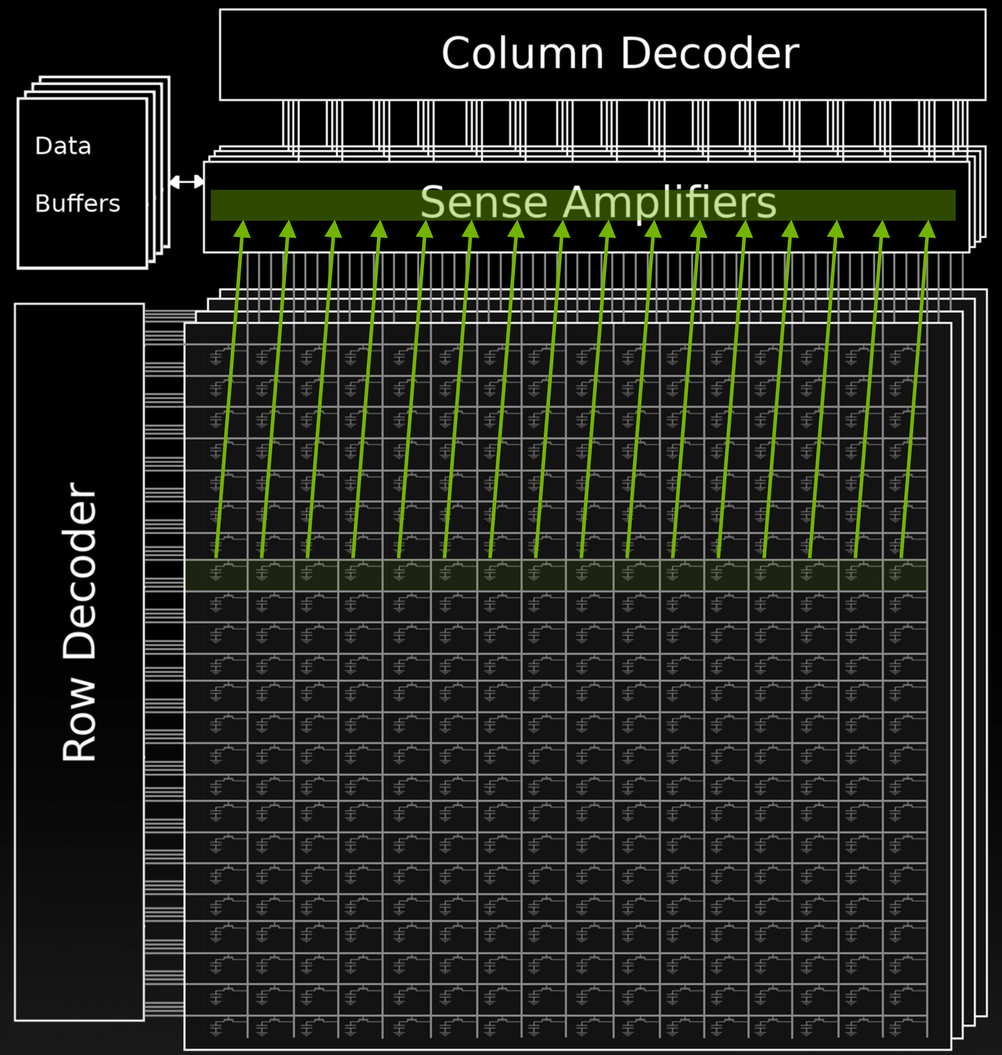
Your milage will very

Depending on how you access memory will greatly affect bandwidth!

Why?

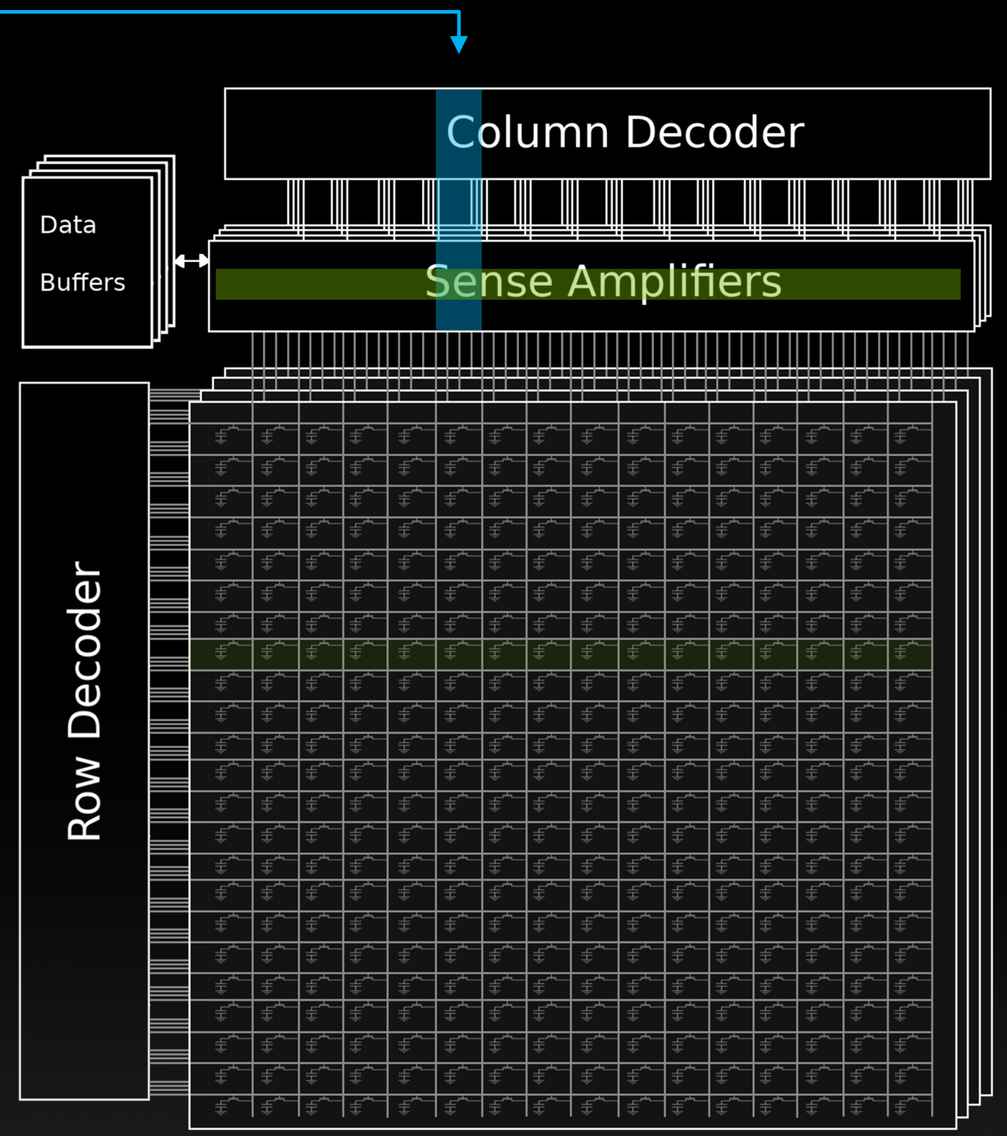


Read address: 00110001001001111010000110110111001 1




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

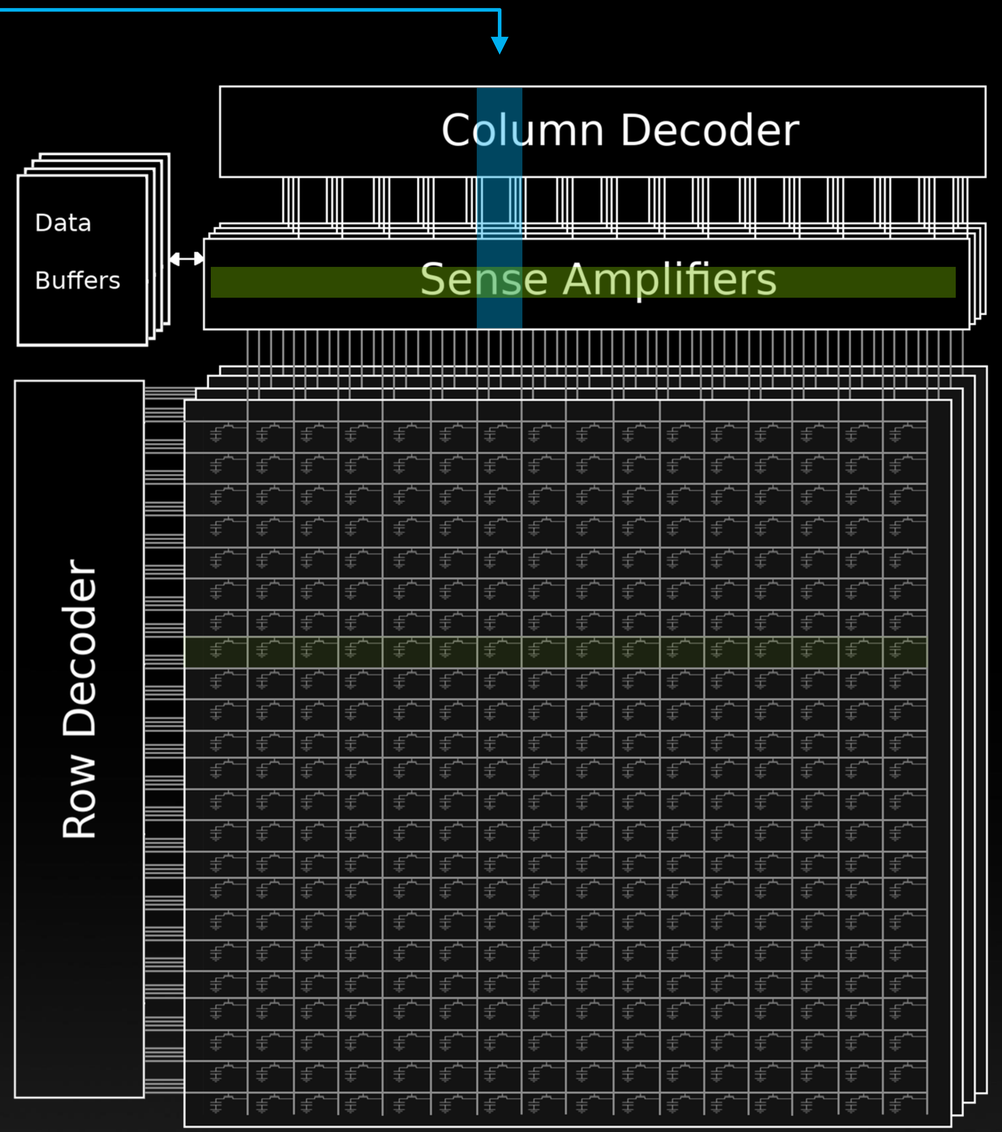
Read address: 00110001001001111010000110110111001 1




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

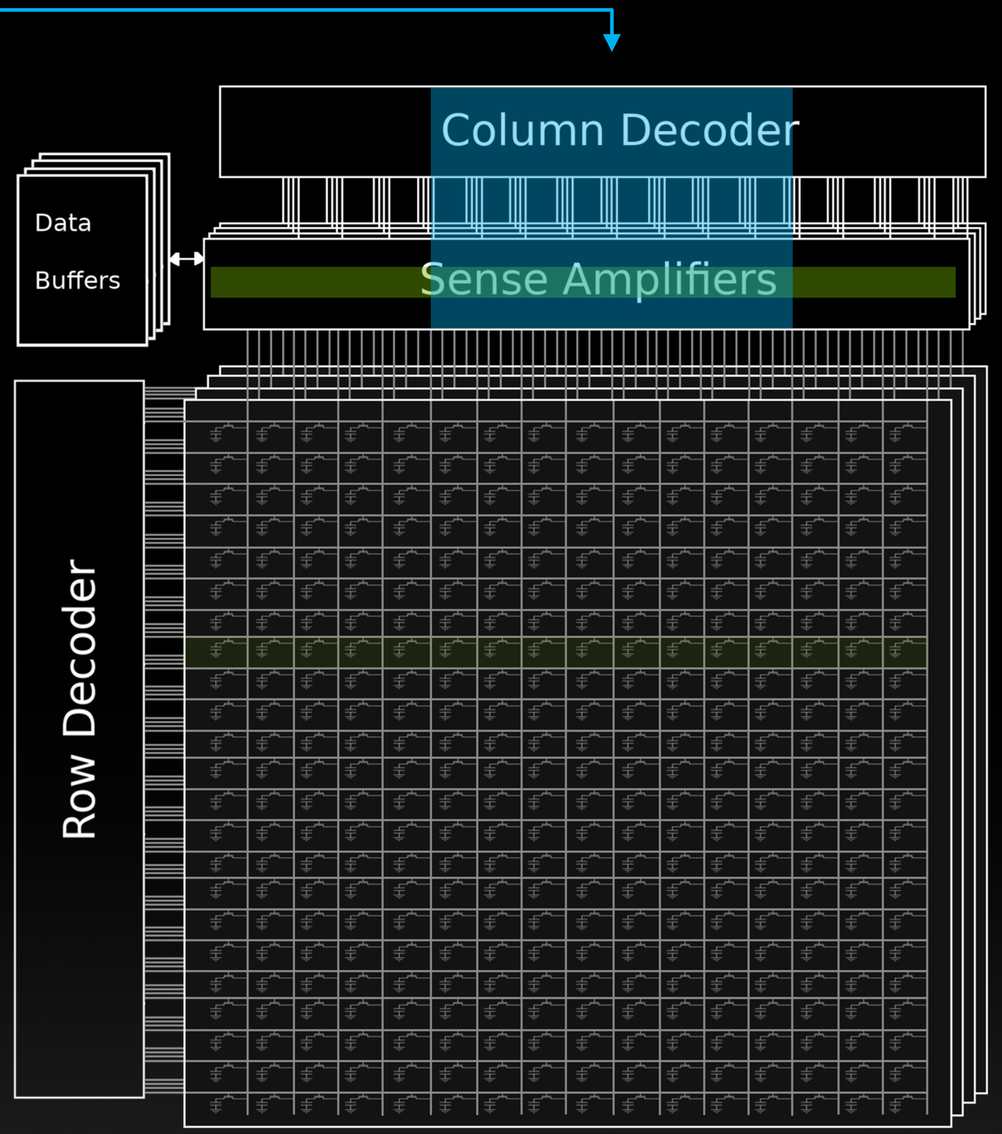
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers

Read address: 00110001001001111010000110110111010**0**



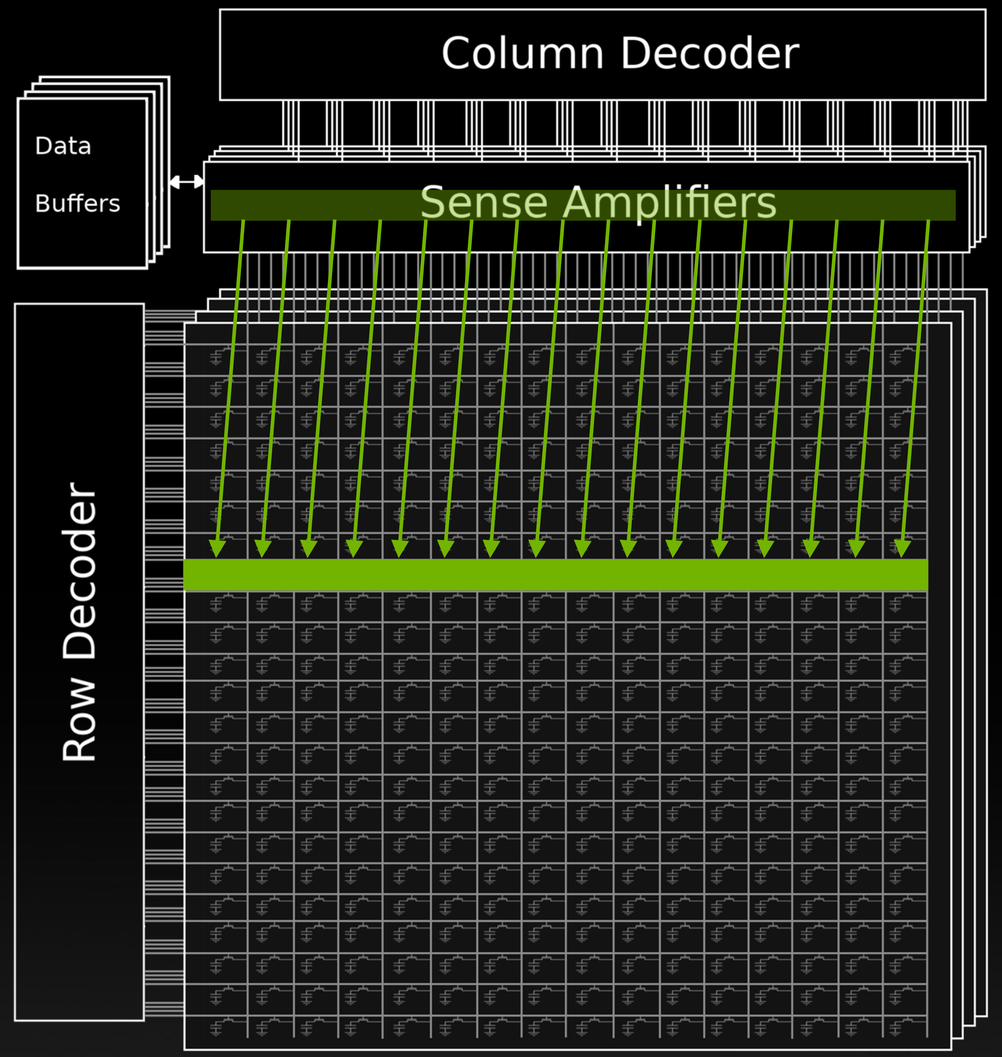

1. Activate row and pull data into sense amplifiers
   This destroys data in the row as capacitors drain
2. Read from page held in amplifiers at column index
   This does not destroy data in the amplifiers
3. May make repeated reads from the same page
   at different column indexes

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

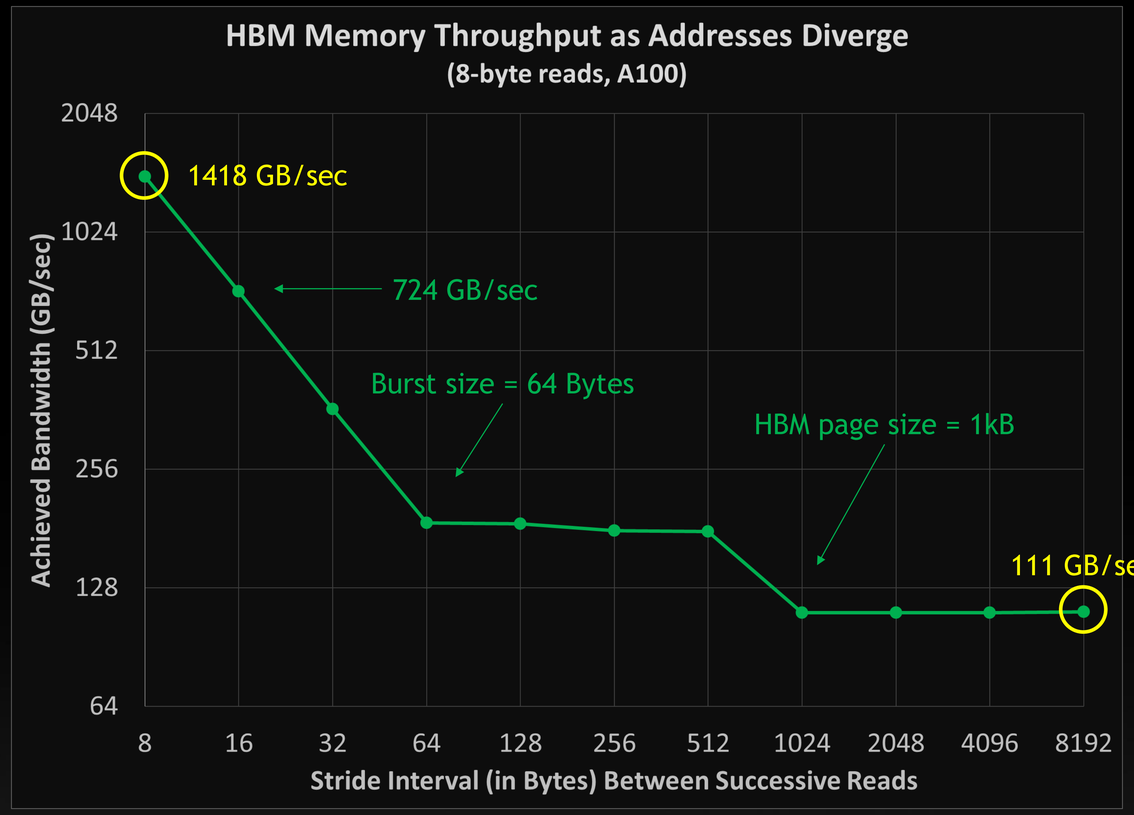
- On A100, memory bandwidth for widely-spaced reads is

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

That's 1/13th of peak bandwidth!

ARITHMETIC INTENSITY=9.7/0.111=88

We need to use every load 700x

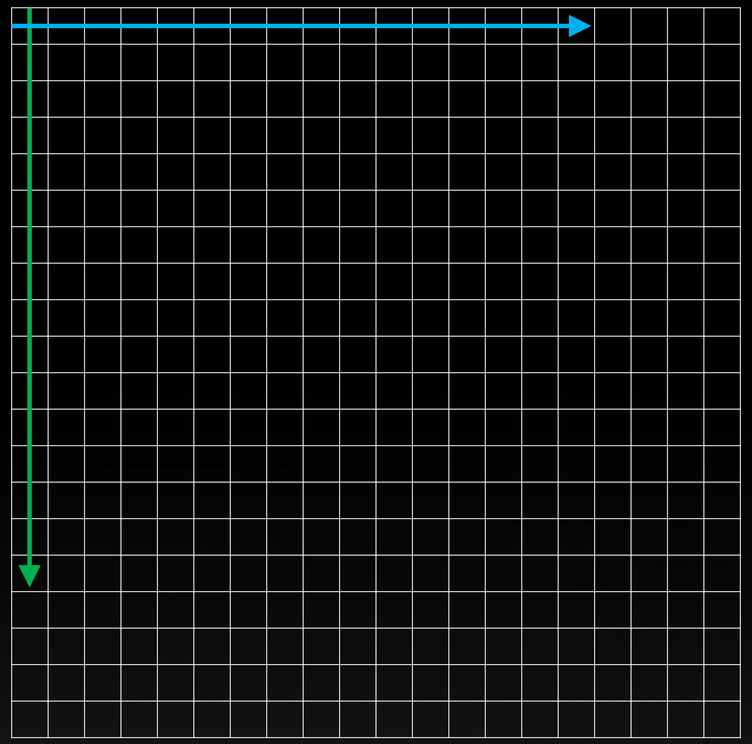


# DATA ACCESS PATTERNS REALLY MATTER

```
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

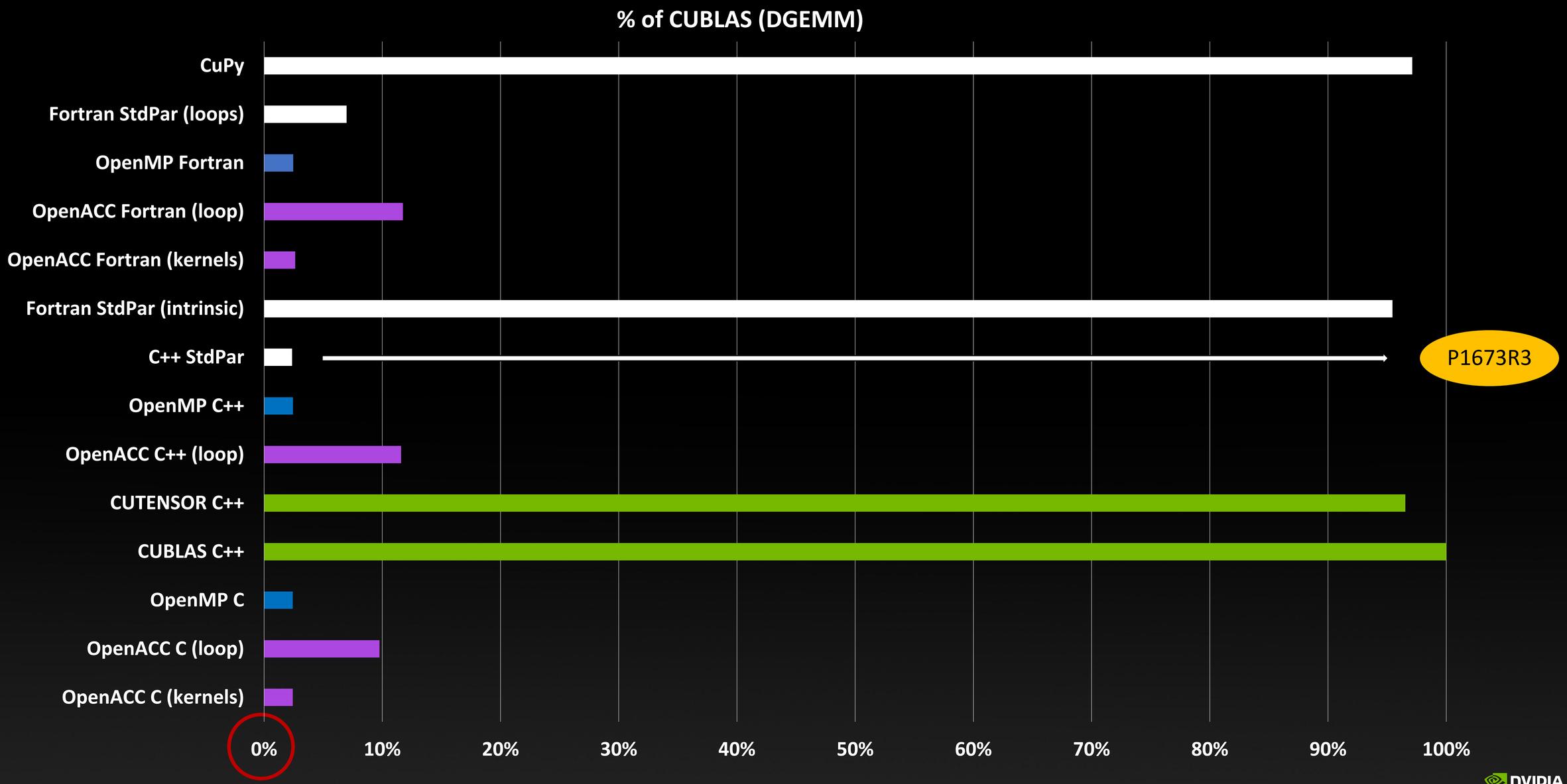
$T_{RAS} = T_{RP} + T_{RDC} + C_L$

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

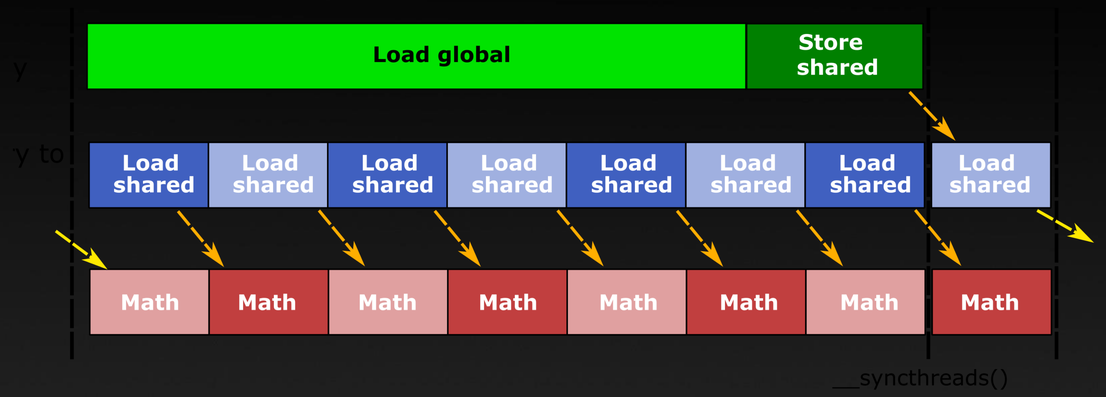
# SO WHAT WENT WRONG?
# Matrix Multiplication: C = C + A * B

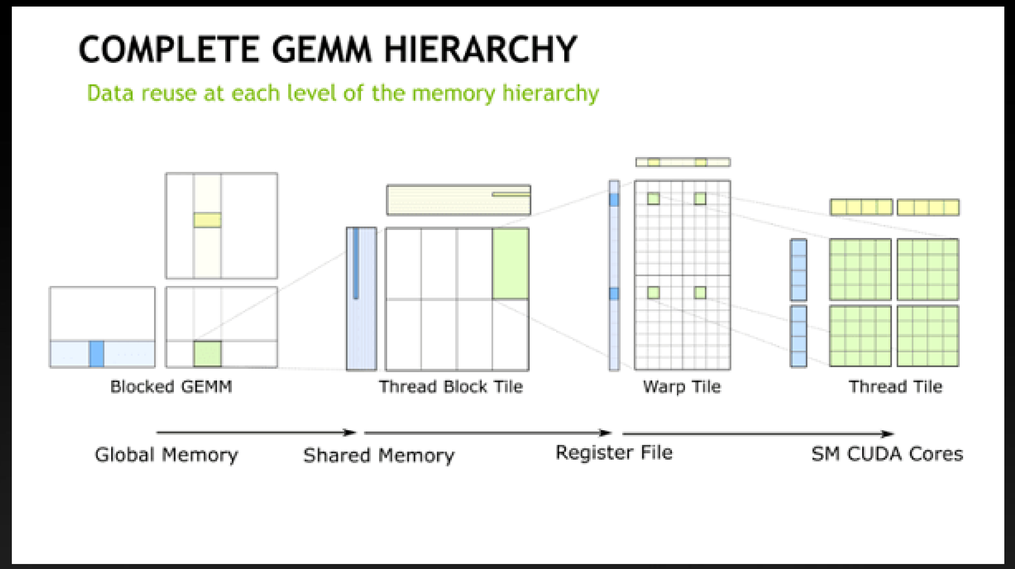


# IT IS ABOUT MEMORY ACCESS

- 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)

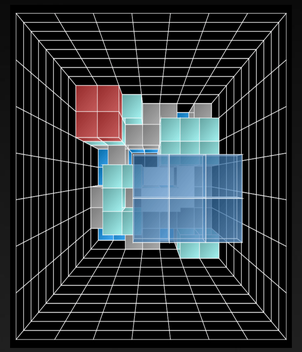
GPUs have very high bandwidth compared to CPUs ( 1.6GB/s vs 0.2 GB/s )

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

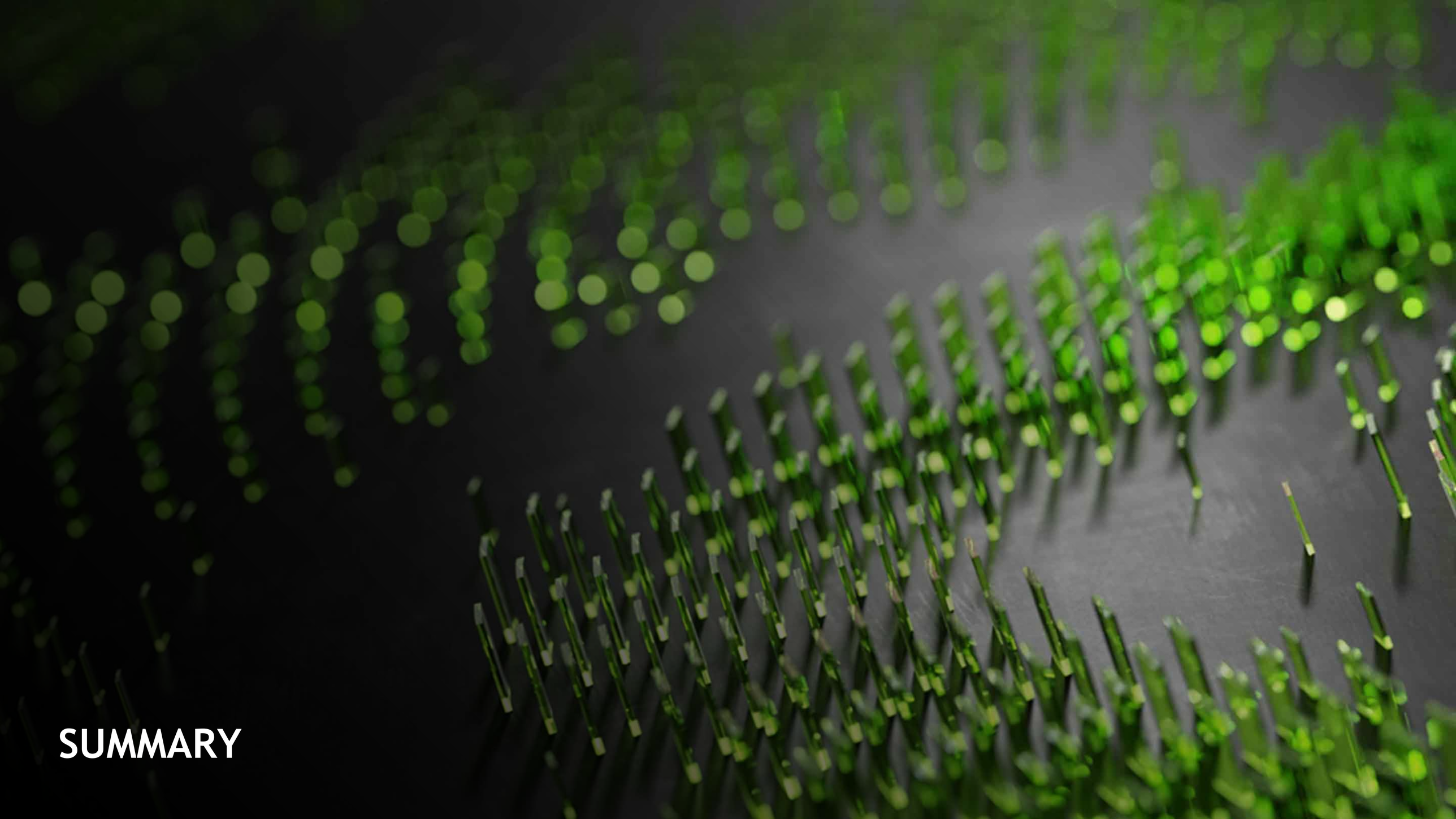
# WORKING WITH HIGH LATENCY

## 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



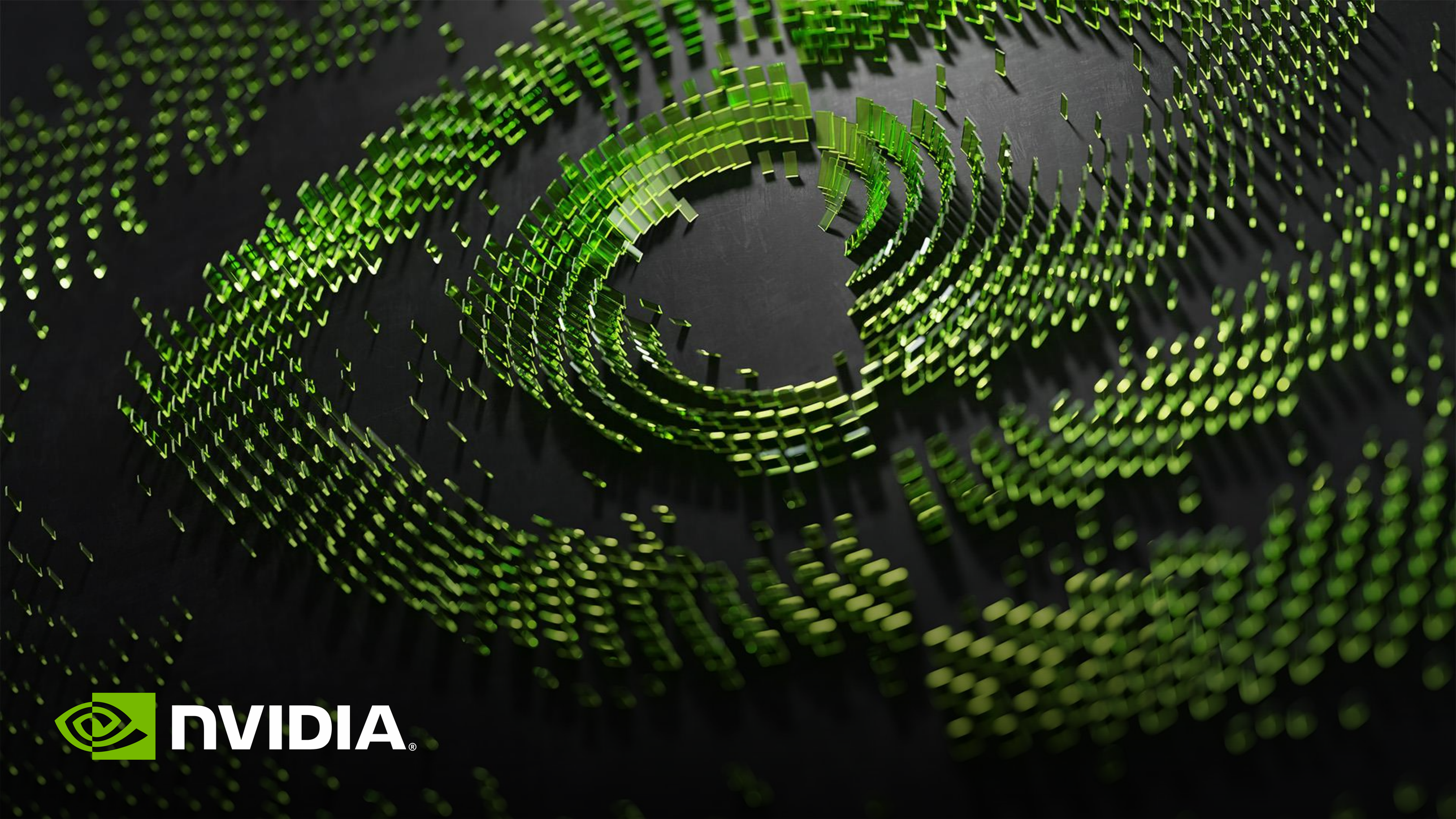

| A100 SM Resources | |
|---|---|
| 2048 | Max threads per SM |
| 32 | Max blocks per SM |
| 65,536 | Total registers per SM |
| 160 kB | Total shared memory in SM |
| 32 | Threads per warp |
| 4 | Concurrent warps active |
| 64 | FP32 cores per SM |
| 32 | FP64 cores per SM |
| 192 kB | Max L1 cache 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

# Introduction to the CINECA Marconi100 HPC system

TREX Hackathon III
March 06-08 2023

Diego Molinari
d.molinari@cineca.it

SuperComputing Applications and Innovations (SCAI) - High Performance Computing Dept

CINECA

# WELCOME TO CINECA

2022 OVERVIEW

## HPC SYSTEMS

CINECA

CINECA enables world-class scientific research by operating and supporting leading-edge supercomputing technologies and by managing a state-of-the-art and effective environment for the different scientific communities.


**LEONARDO** | 2022

4992 nodes
Booster Module:
32 core per node
4 GPU NVidia Ampere custom
Data Centric Module:
56 cores per node
110 PB Storage
250 PFlops

SOON IN PRODUCTION

**MARCONI** | 2016

3188 nodes
48 cores per node
612 TB RAM
10 PFlops

**MARCONI100** | 2020

980 nodes
32 cores per node
4 GPU Nvidia V100 per node
8 PB Storage
32 PFlops

**DGX** | 2021

3 nodes
128 cores per node
8 GPU NVIDIA A100 per node
100 TB Storage
15 PFlops

**GALILEO100** | 2021

564 nodes
48 cores per node
2 GPU NVIDIA V100 per node
~22 PB Storage
2 PFlops

# M100 Infrastructure: how to access

```
$ ssh -X username@login.m100.cineca.it
**************************************************************************
**
* Welcome to MARCONI100 Cluster /
*
*            IBM Power AC922 (Whiterspoon) -
*
*            Red Hat Enterprise Linux Server release 8.1 (Ootpa)
*
etc. etc.
```

- Short system description
- “In evidence” messages
- “Important messages” (changes of policies, maintenances, etc.)

Access by **public keys** (with the ssh keys generated on a local and **secure** environment, and protected via passphrase) is strongly recommended

**IMPORTANT**

Should the Hackathon activities (i.e., compilation) affect the M100 login nodes – it may happen – we will devote a login node to participants. Stay tuned!

# M100 Infrastructure: how to access

$ ssh -X username@login.m100.cineca.it
**********

You will receive personal username and password to connect to M100 for the Hackathon event:

Username: **a08traXX**

Password: **sent by email**
Account: **tra23_hackath** (needed to submit jobs to the queueing system: SLURM)

We also set up a set of reserved nodes for you. They are available using the following reservation:

Reservation: **s_tra_hackath** (valid from 2023-03-06 at 9:00 up to 2023-03-08 at 18:00)

There are 10 nodes in the reservation. If you realize you need more let us know.

At the first login, please **change the password** (you should be forced by the system to do that).
**Password Policy**
The new password has to be 10 characters long and contains at least 1 capital letter, 1 number, and 1 special character (!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~)

# Marconi100: the Power AC922 model

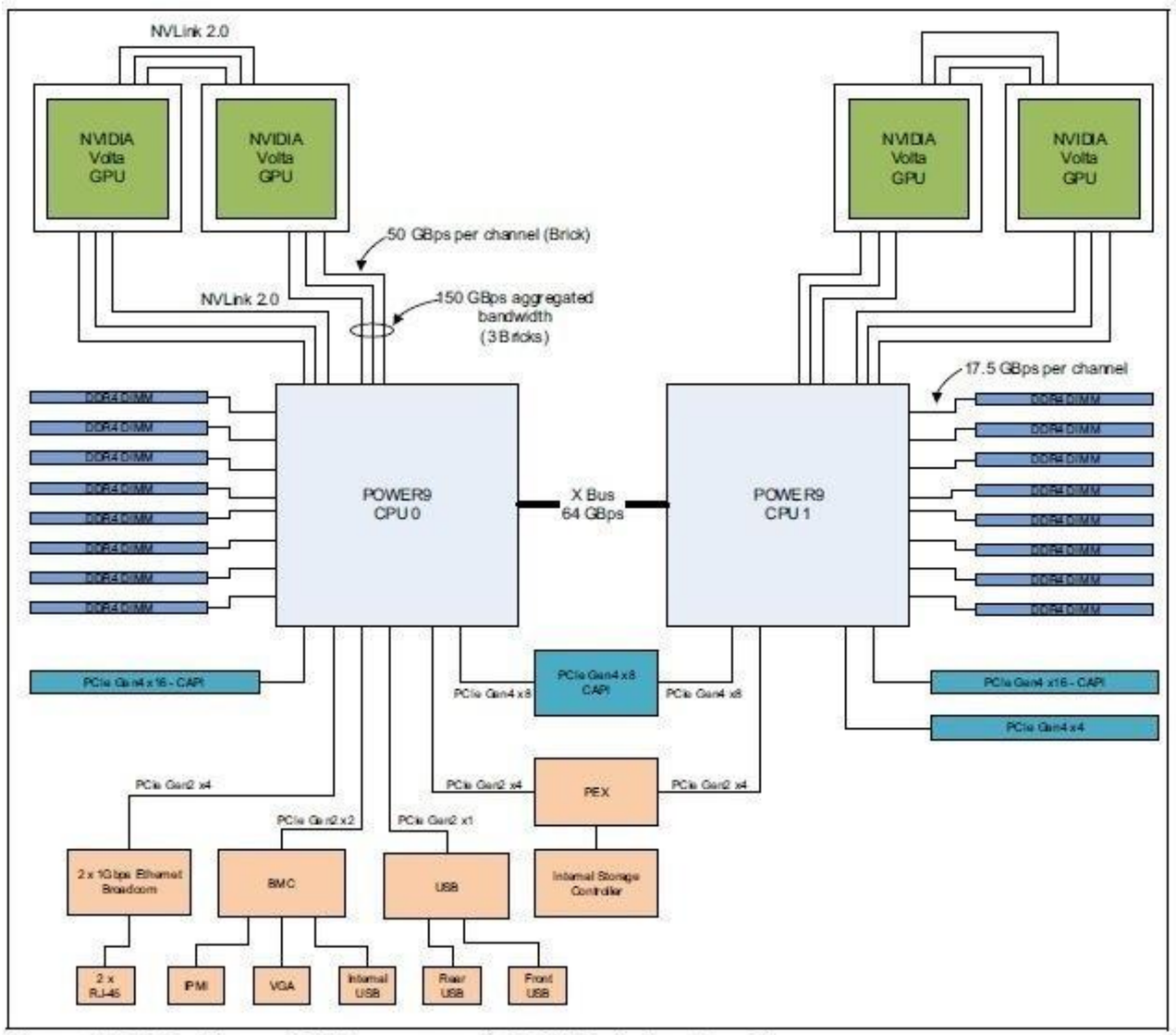
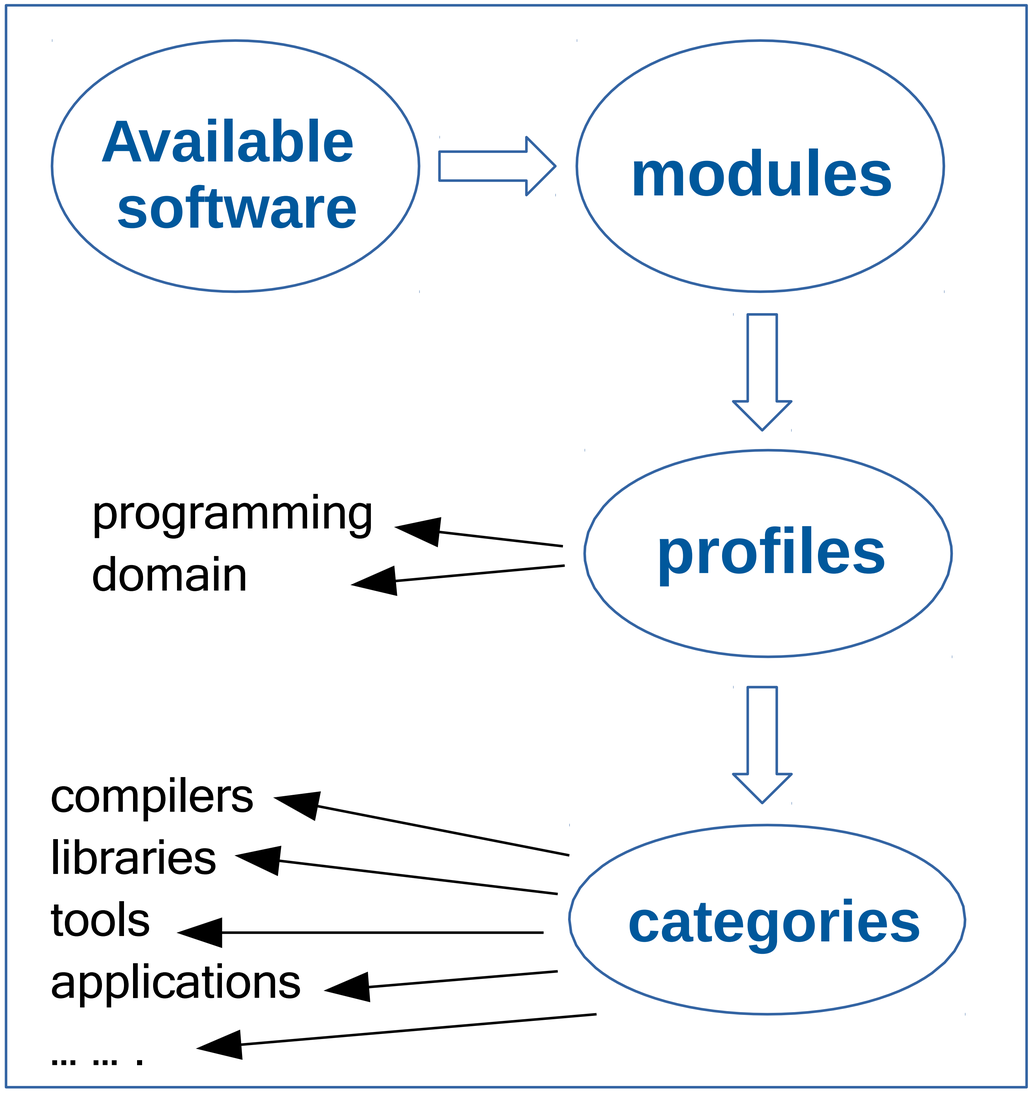



*Figure 2-5 The Power AC922 server model GTH logical system diagram*

- AC922 “Whiterspoon”
- **32 PFlops peak (9th on Top500 June 2020)**
- Nodes: 980 compute + 4 login nodes + 2 (for containers et all), 32 TFlops each
- Processors: 2x16 cores IBM 8335-GTG 2.6 (3.1) GHz
- Accelerators: **4xNVIDIA V100 GPUs**, Nvlink 2.0, 16GB
- RAM: 256 GB/node
- Local disk: 1.6TB NVMe
- Internal Network: Mellanox Infiniband EDR DragonFly+
- Disk Space: 8PB storage

# M100 Module Software Environment




The available software is offered in a module environment

The modules are collected in different profiles and organized in functional categories

**Profile types:**

- Programming (**base**, advanced): compilation, debugging, profiling, libraries
- Domain (chem-phys, lifesc, …) : production activities

The **base** profile is the default:

- automatically loaded after login
- contains basic modules for the programming activities

# M100 Module Software Env: base

```
$ module av

------------------- /cineca/prod/opt/modulefiles/profiles --------------------
profile/advanced  profile/base      profile/chem-phys  profile/bioinf  profile/archive   profile/candidate  profile/deeplm    profile/lifesc
.....
---------------- /cineca/prod/opt/modulefiles/base/environment ----------------
autoload

----------------- /cineca/prod/opt/modulefiles/base/libraries -----------------
blas/3.8.0--gnu--8.4.0    szip/2.1.1--gnu--8.4.0   boost/1.76.0--spectrum_mpi--10.4.0--binary    zlib/1.2.11--gnu--8.4.0
elsi/2.5.0--gnu--8.4.0    essl/6.2.1--binary       .........

----------------- /cineca/prod/opt/modulefiles/base/compilers -----------------
cuda/11.3 gnu/8.4.0  hpc-sdk/2022--binary  python/3.7.7  python/3.8.2  spectrum_mpi/10.4.0--binary  xl/16.1.1--binary

------------------- /cineca/prod/opt/modulefiles/base/tools --------------------
anaconda/2020.11 cmake/3.20.0 singularity/3.9.7 spack/0.14.2-prod
```

# M100 Module Software Env: domains

**“Domain” profiles**:

```
------------------------------------ /cineca/prod/opt/modulefiles/profiles ------------------------------------
profile/advanced profile/archive  profile/base  profile/candidate  profile/chem-phys  profile/deeplrn  profile/bioinf  profile/lifesc
```

To access a “domain” application, *e.g.* in the chemical physics scientific domain, you need to load the profile/chem-phys first:

```
$ module load profile/chem-phys
```

The profile chem-phys is **added** to the base profile

The domain profiles are all “additive”: you can load them together, adding them to the base profile

# M100 Module Software Env:
## autoload, modmap

Needing, *e.g.*, lammps?

```
$ module load profile/chem-phys
$ module load autoload lammps/22dec2022
$ module list
Currently Loaded Modulefiles:
 1) profile/base          4) spectrum_mpi/10.4.0–binary   7) cuda/11.0                    10) lammps/22dec2022
 2) profile/chem-phys     5) gnu/8.4.0                    8) lapack/3.9.0–gnu–8.4.0
 3) autoload              6) blas/3.8.0–gnu–8.4.0         9) fftw/3.3.8–spectrum_mpi–10.4.0
```

The **autoload** module takes care to load all the lammps dependencies

A better, easier way to know if an application is available on M100?

The **modmap** command!

```
$ modmap -m lammps
Profile: advanced
Profile: archive
Profile: base
Profile: chem-phys
                lammps
                 22dec2022
Profile: deeplrn
Profile: lifesc
```

**modmap** detects all the available profiles, categories, and modules
=> “map” of the available modules

modmap -h  # command help

# M100 Module Software Env:
## spack

$ module load spack/0.14.2-prod

- setup-env.sh file is sourced, $SPACK_ROOT is initialized to /cineca/prod/opt/tools/spack/<vers>/none, spack command is added to your PATH, and some nice command line integration tools too.

- A folder is created into your default $WORK space ($USER/spack-<vers>) with the subfolders created and used by spack during the phase of a package installation:

  - sources cache: $WORK/$USER/spack-<vers>/cache
  - software installation root: $WORK/$USER/spack-<vers>/install
  - module files location: $WORK/$USER/spack-<vers>/modulefiles

- You can define different paths for cache, installation and modules directories (please refer to the spack guide to find out how to customize these paths)

- Some softwares installed with spack are already available as modules or as spack packages:

$ module load spack/0.14.2-prod
$ module av
$ module av <module _name>

or

$ module load spack/0.14.2-prod
$ spack find
$ spack find <package_name>

Can't you find the software you need? Use the "**spack**" environment

# M100 Programming Environment

Available compilers in BASE profile:

| | |
|---|---|
| IBM XL C/C++ and Fortran | 16.1.1 |
| gnu | 8.4.0/10.3.0 |
| hpc-sdk | 2020/2021/2022 |
| cuda | 11.0 → 11.3 |

**hwloc** provides details about NUMA memory nodes, sockets, shared caches, cores and SMT, etc.

In addition:
- a gnu compiled **Open MPI 4.0.3** and **4.1.2** installations are available in **profile/advanced**
- more recent compilers (i.e. cuda/11.6 or hpc-sdk/2023 version 23.1) are available in **profile/candidate**

Available MPI environment in BASE profile:

**IBM Spectrum MPI** 10.4.0

- Based on Open MPI version 4.0.5, full MPI 3.2 standard
- FCA (hcoll) support (Mellanox Fabric Collective Accelerator on InfiniBand interconnect)
- Relies on hwloc to navigate the server hardware topology
- GPU support
  - NVIDIA GPUDirect RDMA
  - CUDA-aware MPI

Use **mpirun** (not srun, work in progress) to execute your MPI program
By default, GPUDirect support is disabled.
Run the "**mpirun -gpu**" command to enable it.

Use the --report-bindings option for an abbreviated image of the server's hardware and the binding of processes

# M100 data areas

## Login and Compute nodes

- $HOME  (personal, under back-up, shared GSS over IB)
- $CINECA_SCRATCH  (personal, no back-up, periodic cleaning for files older than 40 days, shared GSS)
- $WORK: common project area (/m100_work/tra23_hackath), 1 TB of quote, no back-up, available in the validity period of project + 6 months, shared GSS.
  Useful to share data among Project members (it is accessible by ALL Teams).
- /scratch_local (1TB NVMe),  local to nodes, not writable on compute nodes

## Compute nodes

- SLURM job TMPDIR:  /scratch_local/slurm_job.<jobid> (1TB NVMe for the scratch_local), local to nodes, created by slurmd prolog at the start of the job and removed at the end of the job

# M100 Environment

A RESERVATION ON **10 NODES** on the partition **m100_sys_test**
IS DEFINED from March, 6th up to March,, 8th

**s_tra_hackath**

In principle **1 node per participant**: PLEASE LET US KNOW IF YOUR ACTIVITY WOULD BENEFIT OF MORE THAN ONE NODE. We can increase the number of nodes in the reservation if needed.

THE RESERVATION IS AVAILABLE TO HACKATHON'S TRAINING USERNAMES

FORGOT IT? Write to us!

```
#!/bin/bash
...
#SBATCH -p m100_sys_test          # defines the partition
#SBATCH -q qos_test               # needed to access the m100_sys_test partition
#SBATCH --reservation=s_tra_hackath   # to access the reserved nodes
...
```

You can in principle use the production partition (m100_usr_prod), but you may end to wait because of a **long queue**.

# M100 Production Environment

M100 is a general purpose system used by hundreds of users.

## Compute nodes

- Production jobs must be submitted to M100 queueing system: batch jobs
- SLURM scheduler and resource manager
- Node sharing (but the allocated resources – cores, gpus, memory – are assigned in an exclusive way)

## Login nodes

A responsible use of the login nodes is crucial to ensuure the effective use of the infrastructure and the access to the computing resources.

- Protect your credentials and access from “safe” posts; opt for ssh keys with passphrase
- Interactive runs on login nodes are strongly discouraged and should be limited to short test runs
  - Per user limits on cpu-time (10 minutes) and memory (1 GB)
  - Avoid running large parallel applications on the front-ends.
- The variable TMPDIR is defined for all users to /scratch_local
  You can re-define it to $CINECA_SCRATCH or other areas.
  PLEASE DO NOT SET it to /tmp → critical!

# M100 Production Environment
## SLURM specs and Accounting

Each node “exposes itself” as having

- 128 (virtual) cpus [32 physical cores with 4 Hts each]
- 4 GPUs
- 246000 MB of memory

It is possible to ask up to
- 128 ntasks-per-node (1 cpus-per-task)
- 1 ntask-per-node (128 cpus-per-task)
- Or any combination of ntasks-per-node * cpus-per-task ≤ 128

BUT

SLURM has been configured so to assign a physical core with its 4 Hts

Asking for –ntasks-per-node=1 and –cpus-pet-task=1 corresponds to ask for --cpus-per-task=4

The accounting considers:

- The requested number of physical cpus
- The requested number of GPUs
- The amount of memory

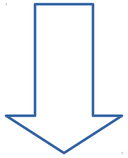
And calculates the number of equivalent cores taking the maximum among

- N physical cpus
- N GPUs * 8
- Memory / Memory-per-core

# M100 Production Environment
## Interactive batch jobs

In case you need to “interact” with your running job (tuning of input parameters, debugging, etc.)
And it needs more than 10 minutes, or many processes (not suitable on the login nodes)

**“Interactive” SLURM batch job**



- Ask for the needed resources (cores, gpus, memory, time) with srun or salloc
- The job is queued and scheduled as any other job, but, when executed, the standard input, output and error streams are connected to the reminal session from which srun or salloc were launched
- You can then run your application from the terminal

**NON MPI programs** (single process or multi-threaded programs using one or more GPUs)

$srun <options> –pty /bin/bash

The session starts on the compute node (look at the prompt!)

**MPI programs** using one or more GPUs

$salloc <options>

A new session is started on the login node
Remember to exit the session when you have finished.

# M100 Production Environment
## Interactive batch jobs for compilation

In case you need to **compile the code on a compute node**

**“Interactive” SLURM batch job**

- Ask for the minimal resources needed for the compilation with **salloc**
- When the job starts you can **ssh to the compute node** SLURM has granted you
- You can then compile your code
- Remember that you have set a **walltime** for the job

[...@login02] $ **ssh r206n11**
[...@**r206n11**] $ nvcc ... ..

When you have finished remember to **exit twice to stop the running job**
[...@**r206n11**] $ exit
[...@**login02**] $ exit

[...@login02] $ **salloc** -N1 --ntasks-per-node=1 -t 02:00:00 -A tra23_hackath
salloc: Pending job allocation 6802903
salloc: Job allocation 6802903 has been revoked.
[...@login02] $ salloc -N1 --ntasks-per-node=1 -t 02:00:00 -A tra23_hackath --partition=m100_usr_prod --qos=qos_test –reservation=s_tra_hackath
salloc: Pending job allocation 6802903
salloc: job 6802903 queued and waiting for resources
salloc: job 6802903 has been allocated resources salloc: Granted job allocation 6802903

# M100 Production Environment
## Non interactive batch jobs

As usual on HPC systems, the large production runs are executed in batch mode.
The user writes a list of the needed #SBATCH directives (resources, walltime, mail, jobname, etc. etc.) followed by the needed loading of modules, setting of variables, and launch of the executable.

```
#!/bin/bash
#SBATCH --nodes=1                    # Number of nodes
#SBATCH --ntasks-per-node=4          # Number of MPI ranks per node
#SBATCH --ntasks-per-socket=2        # Number of MPI ranks per socket
#SBATCH --cpus-per-task=32           # number of HW threads per task
#SBATCH --gres=gpu:4                 # Number of requested gpus per node, can vary between 1 and 4
#SBATCH --mem=230000MB               # Memory per node
#SBATCH --time 00:30:00              # Walltime, format: HH:MM:SS (max 24 hours)
#SBATCH -A tra23_hackath
#SBATCH -p m100_sys_test
#SBATCH -q qos_test
#SBATCH --reservation=s_tra_hackath

module load profile/chem-phys
Module load autoload yambo/4.5

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

mpirun -map-by socket:PE=8 -rank-by core -np ${SLURM_NTASK} yambo -F yambo.in -J yambo.out
```

# M100 Production Environment
## Profiling your code with Nvidia Nsight system

In the previous edition, the usage of the Nvidia profiler caused several compute node crashes due to the usage of the /tmp area of the node by the profiler itself.
In order to avoid such a problem, we suggest to **modify the sbatch script** as in the following example.

```
#!/bin/bash
#SBATCH directives
#SBATCH –exclusive        #to avoid superposition of profiling jobs on the same node

module load hpc-sdk
module load … …

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

rm -rf /tmp/nvidia
ln -s $TMPDIR /tmp/nvidia
mpirun ..... nsys profile ....
rm -rf /tmp/nvidia
```

In the worst case the node does not crash when filling the /tmp area, but the job simply stop producing output, but still running with a consequence lost of time and cpu-hours.

We kindly ask to follow these suggestions

**IMPORTANT:**
on M100 Nvidia Nsight system **GUI is not supported**.
Please run the profiler via **command line**, then you can download the .qdrep result on your **local PC for visualization**

# M100 HELP!

- We will be around for any kind of support needed!
- Ask superc@cineca.it (during and after hackathon. PLEASE: mention TREX Hackathon 2023 in the subject)
- Online guide: https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.2%3A+MARCONI100+UserGuide

**ENJOY TREX Hackathon III @ CINECA**

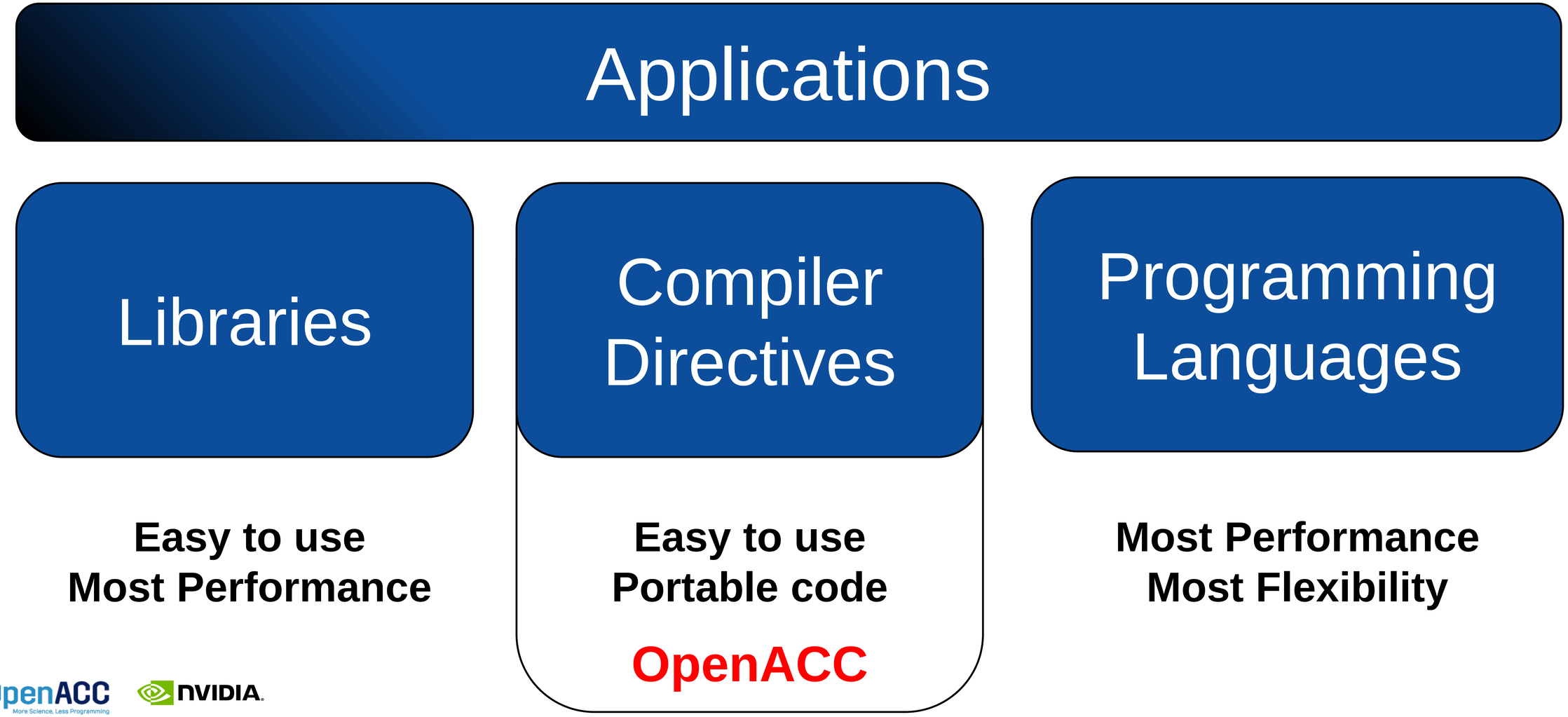
**from CINECA User Support Team!**

# INTRODUCTION TO OPENACC

OpenACC
More Science, Less Programming

NVIDIA

# 3 WAYS TO ACCELERATE APPLICATIONS




OpenACC More Science, Less Programming

NVIDIA

OPENACC DIRECTIVES

# a directive-based parallel programming model designed for usability, performance and portability

| APPLICATIONS | PLATFORMS SUPPORTED | COMMUNITY |
| --- | --- | --- |
| **250+**<br>3 out of Top 5 | NVIDIA GPU<br>X86 CPU<br>POWER CPU<br>Sunway<br>ARM CPU<br>AMD GPU<br>FPGA | **~3000**<br>Slack Members |

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          c[i] = a[i] + b[i];
          ...
      }
  }
  ...
}
```

OpenACC
Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

OpenACC More Science, Less Programming

NVIDIA

## Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

## Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

## Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

# OPENACC RESOURCES

Guides • Talks • Tutorials • Videos • Books • Spec • Code Samples • Teaching Materials • Events • Success Stories • Courses • Slack • Stack Overflow

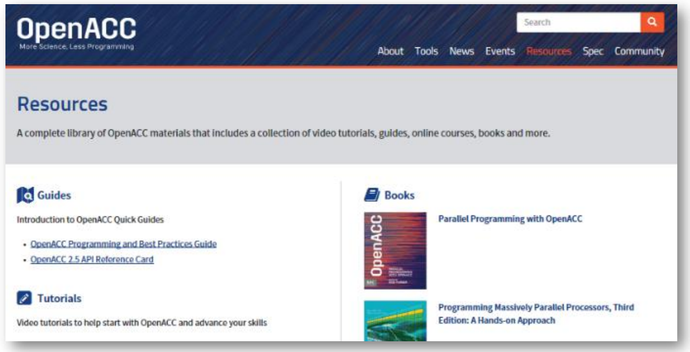
**FREE Compilers**

**NVIDIA HPC SDK**

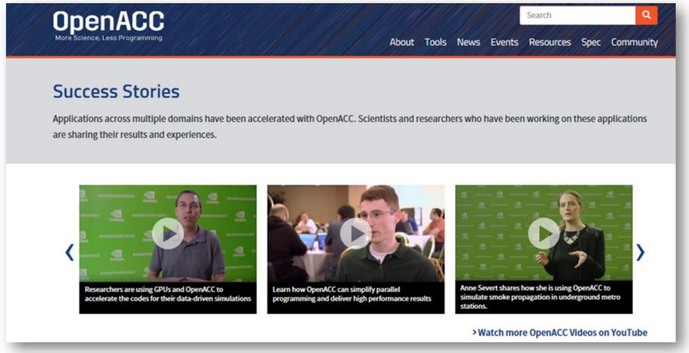
slack

https://www.openacc.org/community

## Resources

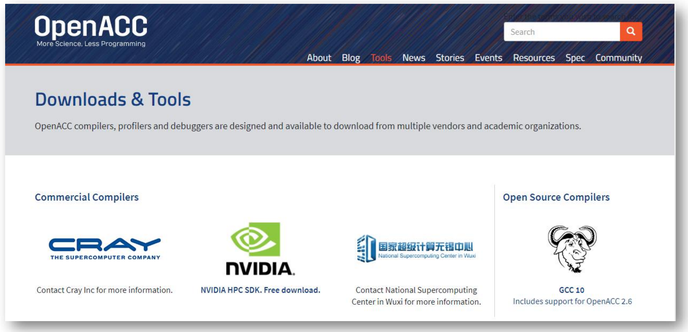



## Success Stories



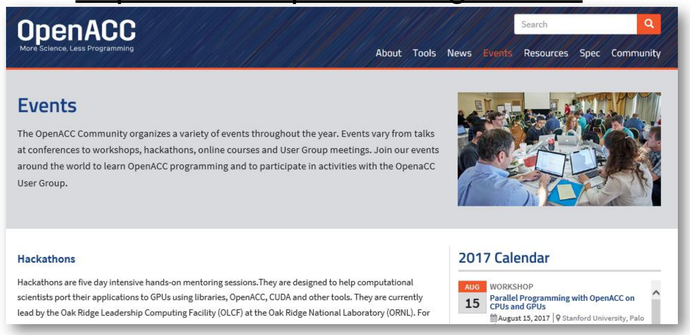

## Compilers and Tools

https://www.openacc.org/tools




## Events

https://www.openacc.org/events




OpenACC More Science, Less Programming NVIDIA

APPLY TO GPU HACKATHONS

# Accelerate your code on GPUs with mentors by your side

- Over 20 events globally.
- 4 full days over 2 weeks.
- Online or in-person.
- 2 mentors per team. Up to 10 teams.
- Free to participate.
- GPU resource is provided.


www.gpuhackathons.org/events

OpenACC More Science, Less Programming

NVIDIA

# OPENACC SYNTAX

OPENACC SYNTAX

# Syntax for using OpenACC directives in code

C/C++

```
#pragma acc directive clauses
<code>
```

Fortran

```
!$acc directive clauses
<code>
```

- A ***pragma*** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.
- A ***directive*** in Fortran is a specially formatted comment that likewise instructions the compiler in it compilation of the code and can be freely ignored.
- "***acc"*** informs the compiler that what will come is an OpenACC directive
- ***Directives*** are commands in OpenACC for altering our code.
- ***Clauses*** are specifiers or additions to directives.

EXAMPLE CODE

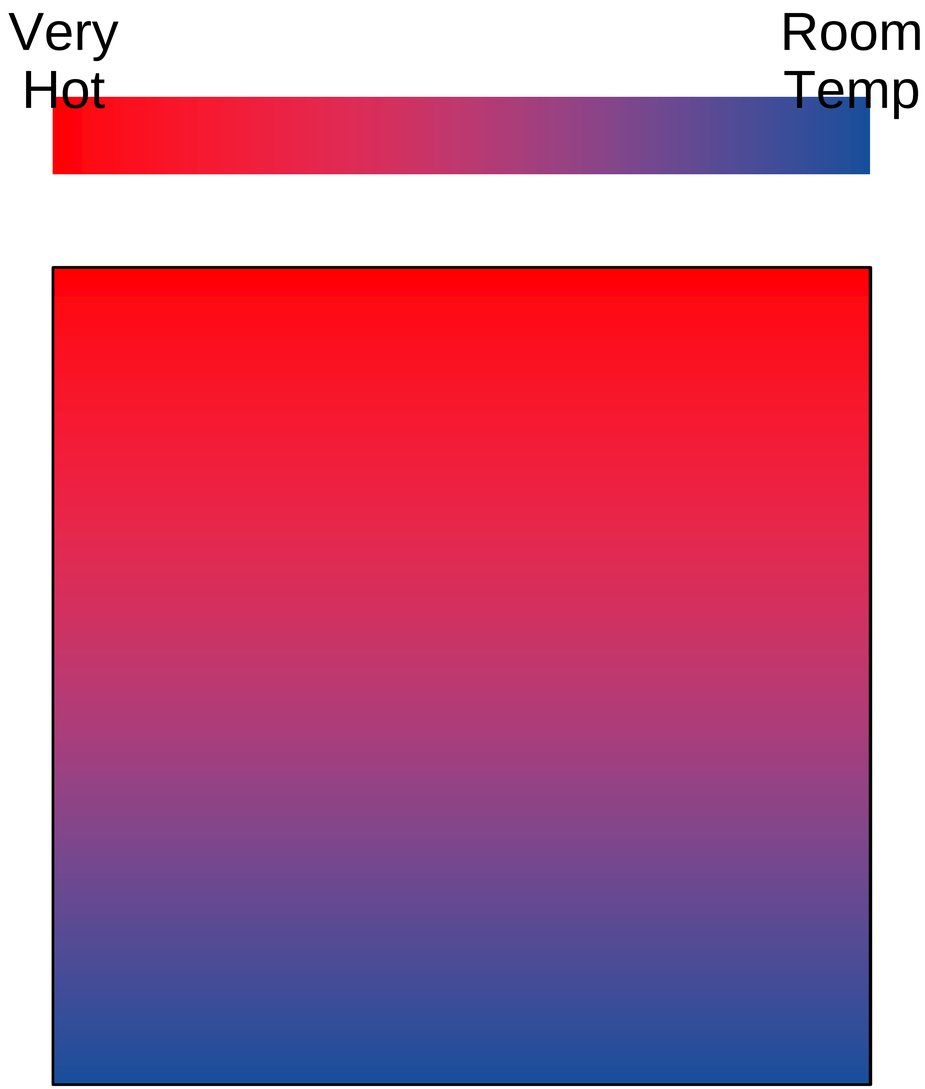
LAPLACE HEAT TRANSFER

# Introduction to lab code - visual

We will observe a simple simulation of heat distributing across a metal plate.

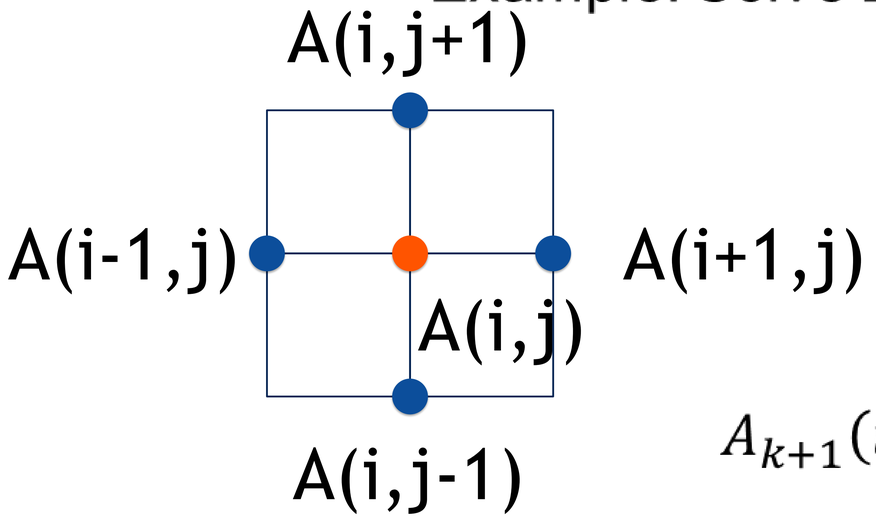
We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.



EXAMPLE: JACOBI ITERATION

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
- Common, useful algorithm
- Example: Solve Laplace equation in 2D: $\nabla^2 f(x,y) = 0$




$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

JACOBI ITERATION: C CODE

```
while ( err > tol && iter < iter_max ) {
  err=0.0;


  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }


  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

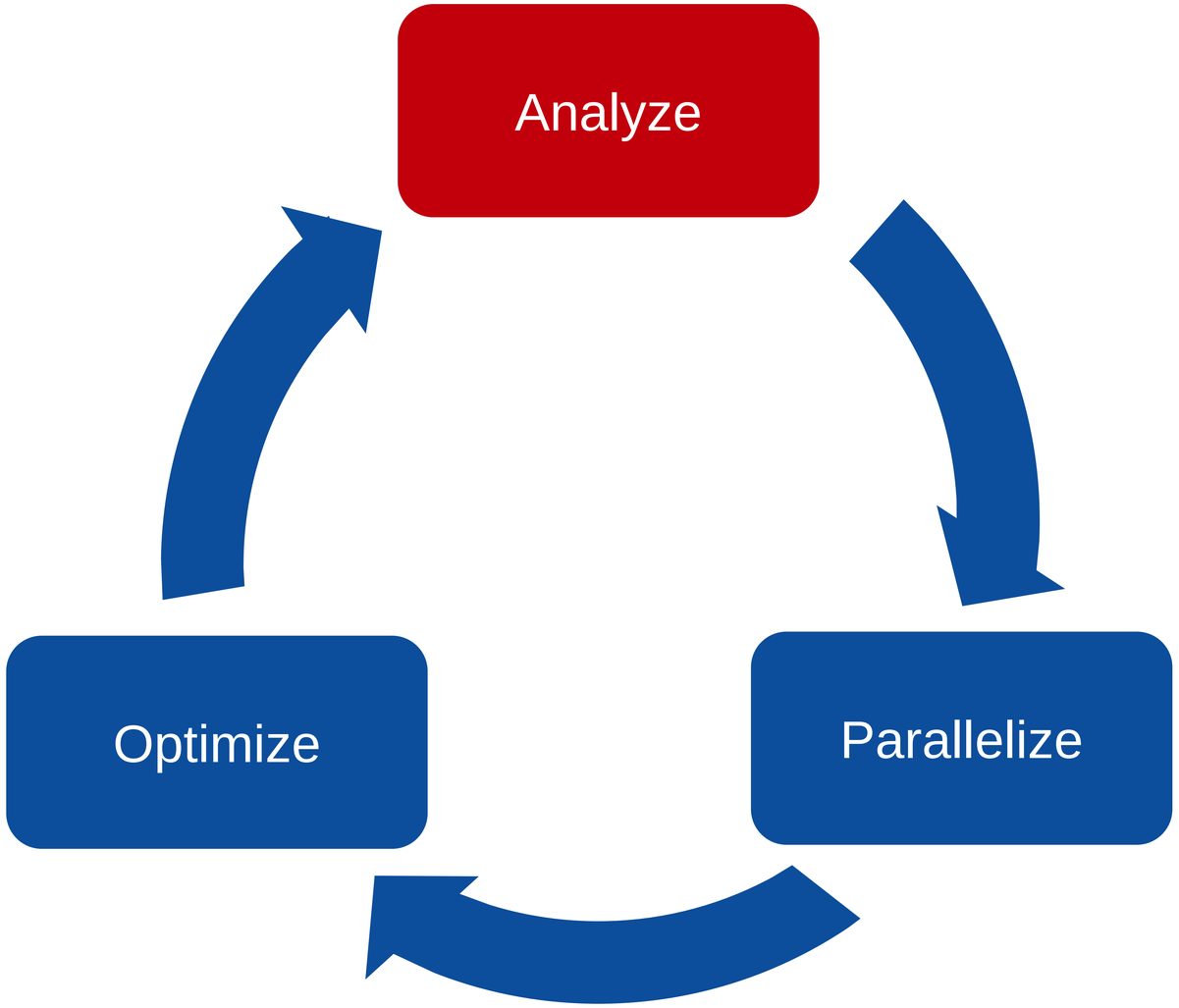
OpenACC More Science, Less Programming

NVIDIA

# PROFILE-DRIVEN DEVELOPMENT

OPENACC DEVELOPMENT CYCLE

- **Analyze** your code to determine most likely places needing parallelization or optimization.
- **Parallelize** your code by starting with the most time consuming parts and check for correctness.
- **Optimize** your code to improve observed speed-up from parallelization.



# OPENACC PARALLEL LOOP DIRECTIVE

OPENACC PARALLEL DIRECTIVE

# Parallelizing a single loop

C/C++

```
#pragma acc parallel
{
  #pragma acc loop
  for(int i = 0; j < N; i++)
        a[i] = 0;
}
```

Fortran

```
!$acc parallel
  !$acc loop
  do i = 1, N
        a(i) = 0
  end do
!$acc end parallel
```

- Use a **parallel** directive to mark a region of code where you want parallel execution to occur
- This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran
- The **loop** directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

OpenACC More Science, Less Programming NVIDIA

# PARALLELIZE WITH OPENACC PARALLEL LOOP

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

**Parallelize first loop nest, max *reduction* required.**

**Parallelize second loop.**

We didn't detail *how* to parallelize the loops, just *which* loops to parallelize.

OpenACC More Science, Less Programming

NVIDIA

# BUILD AND RUN THE CODE

# OPENACC DATA MANAGEMENT

OpenACC
More Science, Less Programming

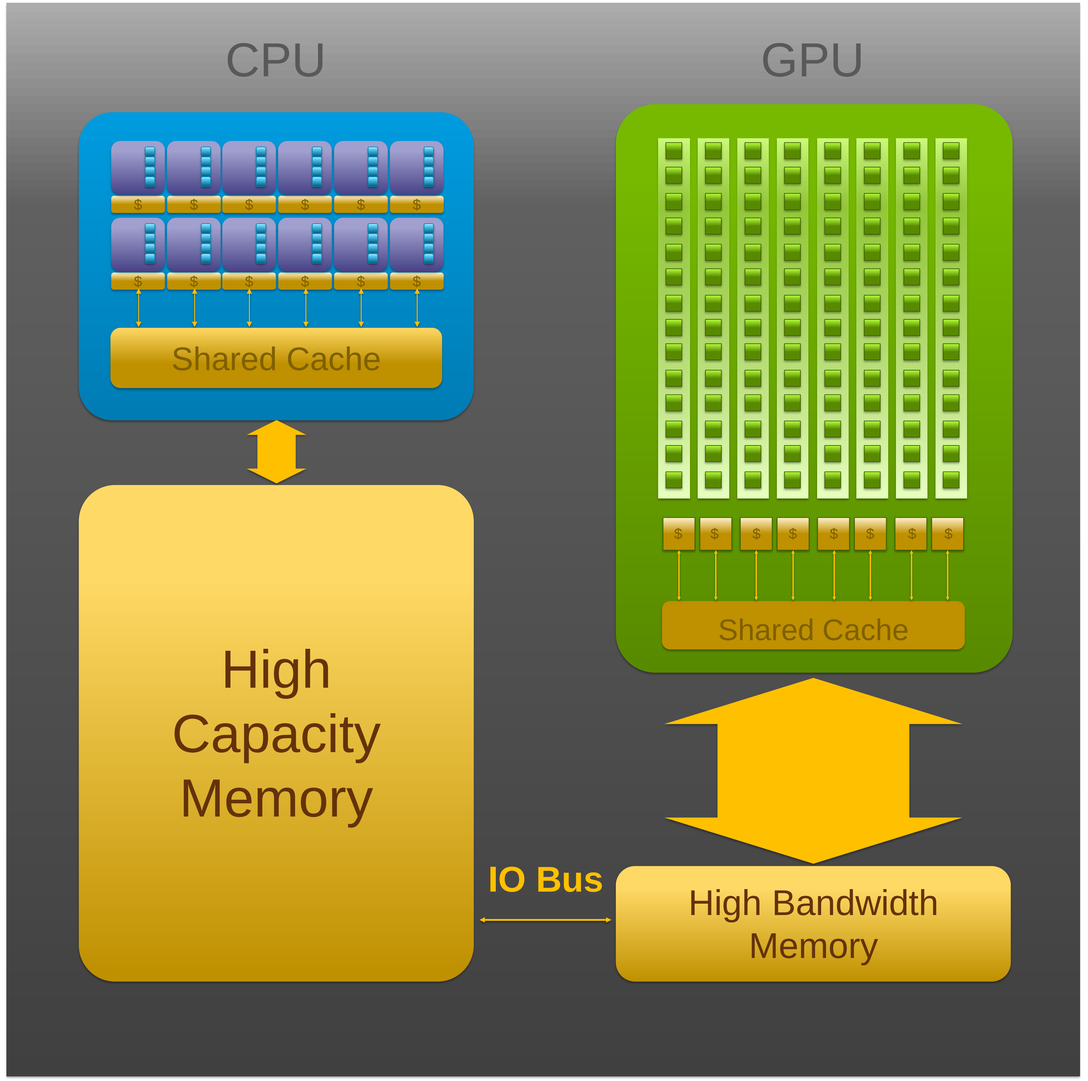
NVIDIA

# CPU AND GPU MEMORIES

CPU + GPU

# Physical Diagram

- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCI-e)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth
- The GPU cannot perform computation until the data is within its memory



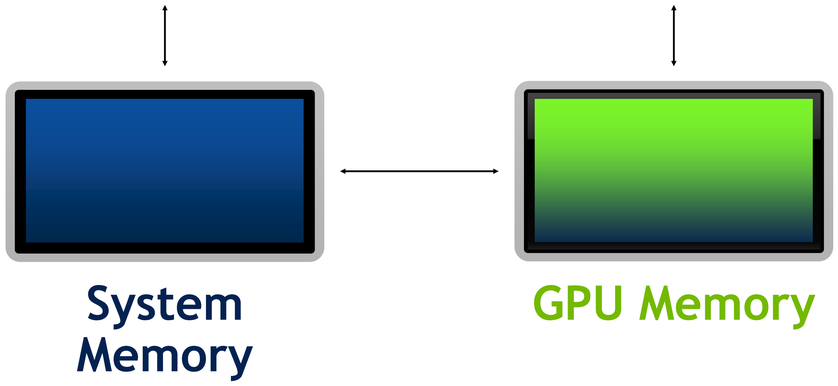
# CUDA UNIFIED MEMORY

CUDA UNIFIED MEMORY

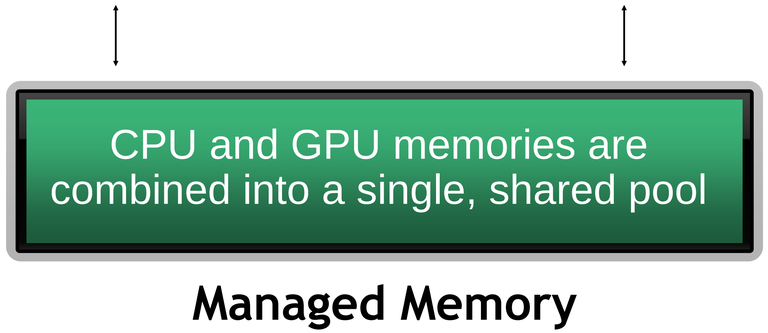
# Simplified Developer Effort

Commonly referred to as
*“managed memory.”*

## Without Managed Memory




## With Managed Memory




OpenACC More Science, Less Programming

NVIDIA

# Usefulness

- Handling explicit data transfers between the host and device (CPU and GPU) can be difficult
- The NVIDIA HPC compiler can utilize CUDA Managed Memory to defer data management
- This allows the developer to concentrate on parallelism and think about data movement as an optimization

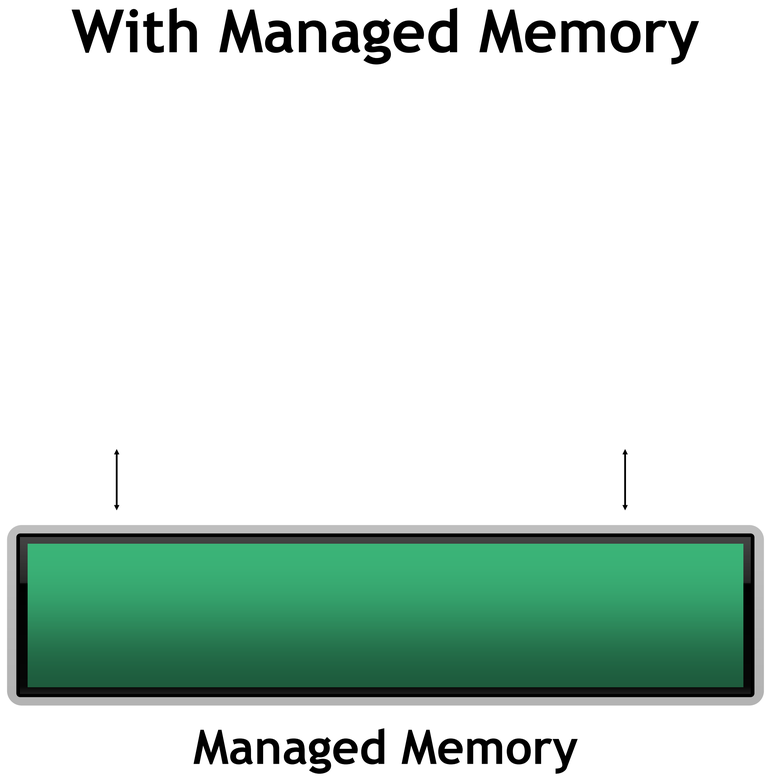
```
$       nvc –fast –acc –ta=tesla:managed –Minfo=accel main.c
```

```
$       nvfortran –fast –acc –ta=tesla:managed –Minfo=accel main.f90
```

MANAGED MEMORY

# Limitations

- The programmer will almost always be able to get better performance by manually handling data transfers
- Memory allocation/deallocation takes longer with managed memory
- Cannot transfer data asynchronously
- Currently only available on NVIDIA GPUs with NVIDIA HPC SDK.



# DATA SHAPING

DATA CLAUSES

`copy( list )` **Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.**

**Principal use:** For many important data structures in your code, this is a logical default to input, modify and return the data.

`copyin( list )` **Allocates memory on GPU and copies data from host to GPU when entering region.**

**Principal use:** Think of this like an array that you would use as just an input to a subroutine.

`copyout( list )` **Allocates memory on GPU and copies data to the host when exiting region.**

**Principal use:** A result that isn't overwriting the input data structure.

`create( list )` **Allocates memory on GPU but does not copy.**

**Principal use:** Temporary arrays.

## ARRAY SHAPING

- Sometimes the compiler needs help understanding the *shape* of an array
- The first number is the start index of the array
- In C/C++, the second number is how much data is to be transferred
- In Fortran, the second number is the ending index

```
copy(array[starting_index:length])
```

C/C++

```
copy(array(starting_index:ending_index))
```

Fortran

OpenACC More Science, Less Programming NVIDIA

# OPTIMIZED DATA MOVEMENT

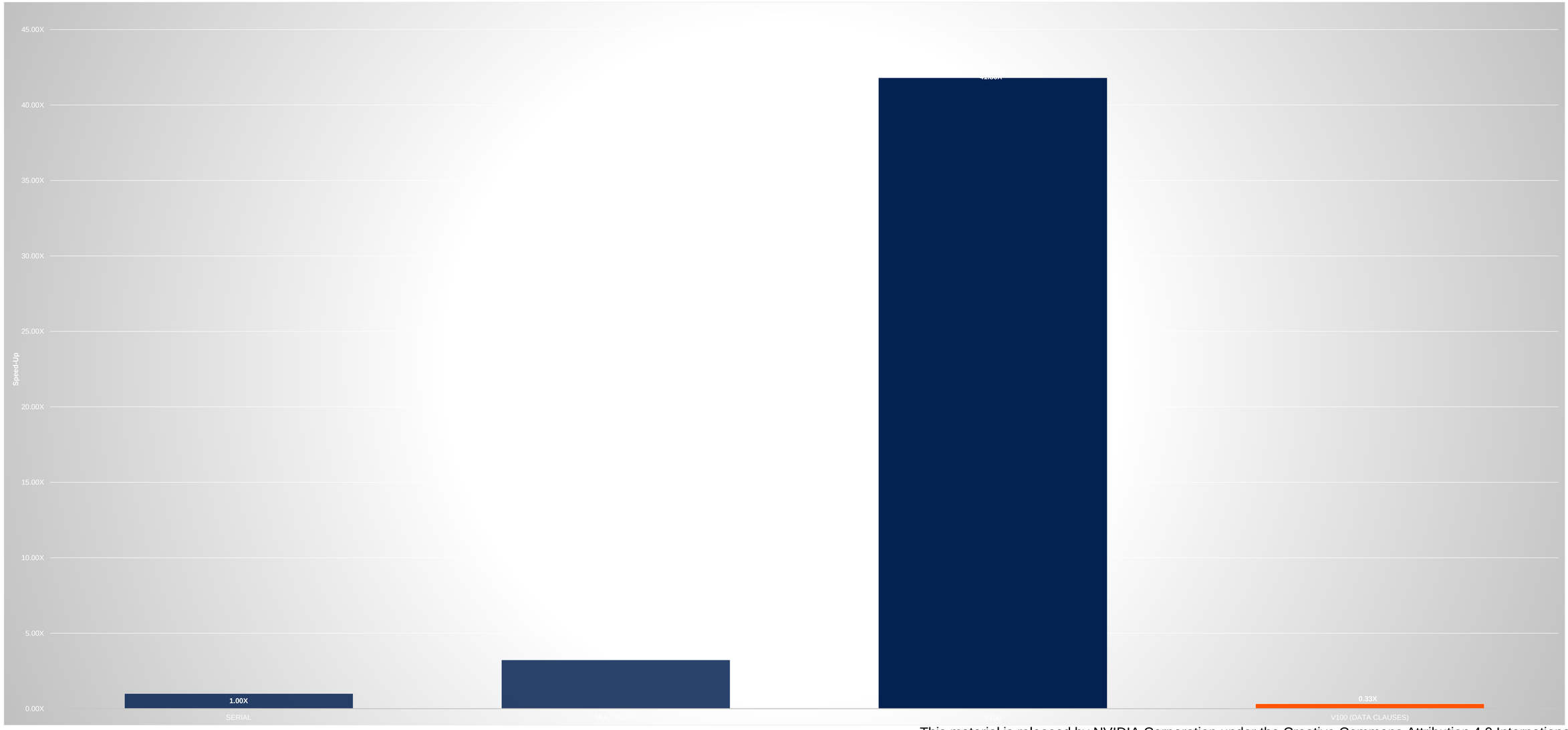
```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Data clauses provide necessary “shape” to the arrays.

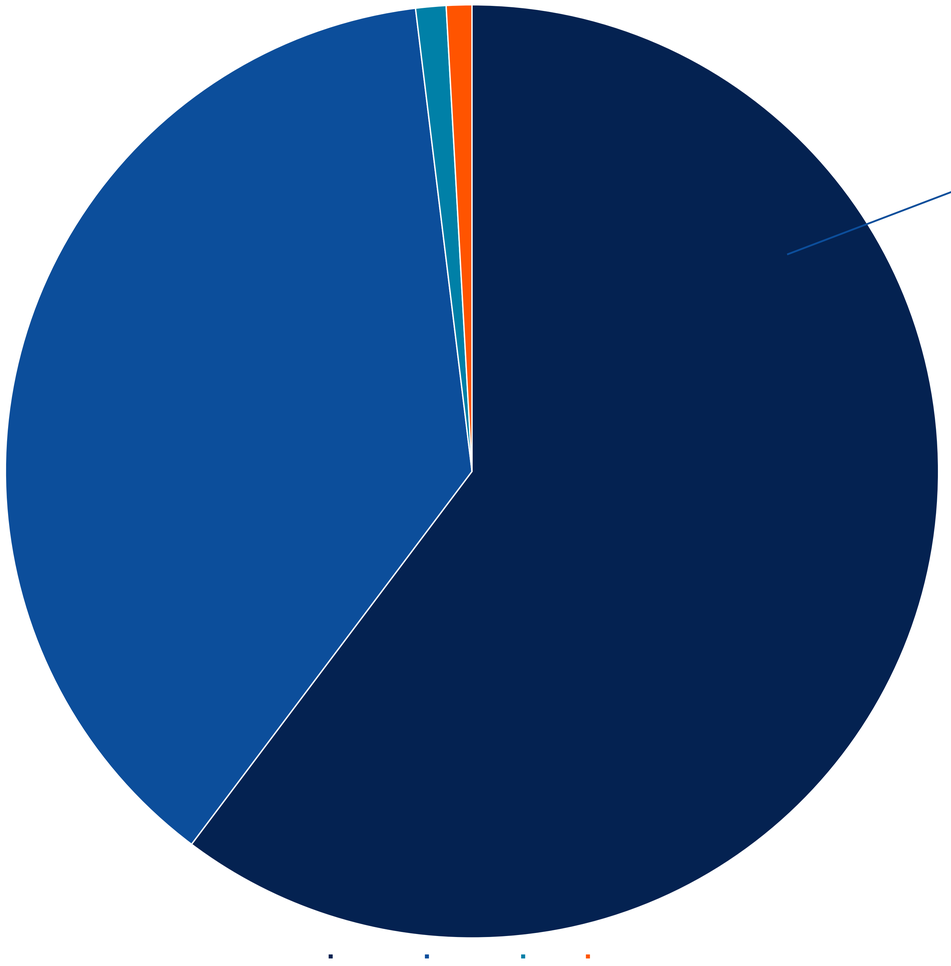
OpenACC More Science, Less Programming

NVIDIA

# OPENACC ~~SPEED-UP~~ SLOWDOWN



RUNTIME BREAKDOWN



Nearly all of our time is spent moving data to/from the GPU

# OPTIMIZED DATA MOVEMENT

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Currently we're copying to/from the GPU for each loop, can we reuse it?

OpenACC More Science, Less Programming

NVIDIA

# OPTIMIZE DATA MOVEMENT

OPENACC DATA DIRECTIVE

# Definition

- The data directive defines a lifetime for data on the device beyond individual loops
- During the region data is essentially “owned by” the accelerator
- Data clauses express shape and data movement for the region

```
#pragma acc data clauses
{

 < Sequential and/or Parallel code >

}
```

```
!$acc data clauses

 < Sequential and/or Parallel code >

!$acc end data
```

OpenACC More Science, Less Programming

NVIDIA

# OPTIMIZED DATA MOVEMENT

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

**Copy A to/from the accelerator only when needed.**

**Copy initial condition of Anew, but not final value**

# REBUILD THE CODE

```
nvc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c
main:
     60, Generating copy(A[:m*n])
         Generating copyin(Anew[:m*n])
     64, Accelerator kernel generated
         Generating Tesla code
         64, Generating reduction(max:error)
         65, #pragma acc loop gang /* blockIdx.x */
         67, #pragma acc loop vector(128) /* threadIdx.x */
     67, Loop is parallelizable
     75, Accelerator kernel generated
         Generating Tesla code
         76, #pragma acc loop gang /* blockIdx.x */
         78, #pragma acc loop vector(128) /* threadIdx.x */
     78, Loop is parallelizable
```

**Now data movement only happens at our data region.**

## WHAT WE'VE LEARNED SO FAR

- CUDA Unified (Managed) Memory is a powerful porting tool
- GPU programming without managed memory often requires data shaping
- Moving data at each loop is often inefficient
- The OpenACC Data region can decouple data movement and computation

# DATA SYNCHRONIZATION

## OPENACC UPDATE DIRECTIVE

**update:** Explicitly transfers data between the host and the device

Useful when you want to synchronize data in the middle of a data region

Clauses:

**self:** makes host data agree with device data

**device:** makes device data agree with host data

```
#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])
```

C/C++

```
!$acc update self(x(1:end_index))
!$acc update device(x(1:end_index))
```

Fortran

# OPENACC RESOURCES

Guides • Talks • Tutorials • Videos • Books • Spec • Code Samples • Teaching Materials • Events • Success Stories • Courses • Slack • Stack Overflow

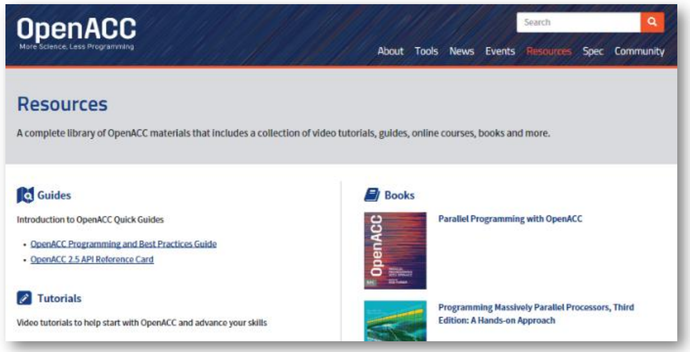
**FREE Compilers**

**NVIDIA HPC SDK**

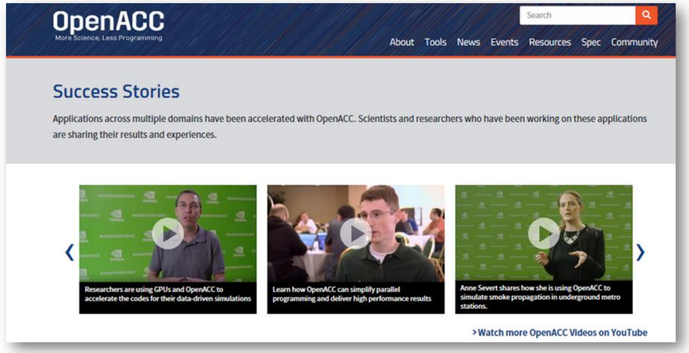
slack

https://www.openacc.org/community

## Resources

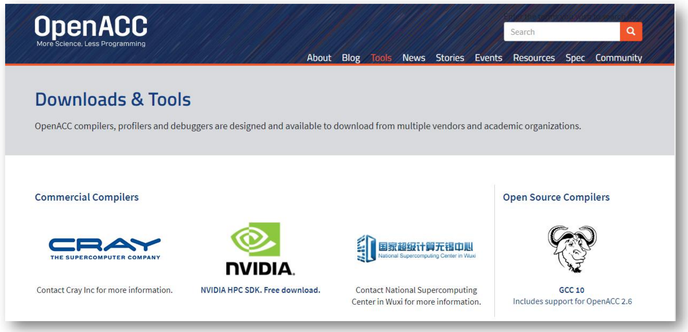



## Success Stories



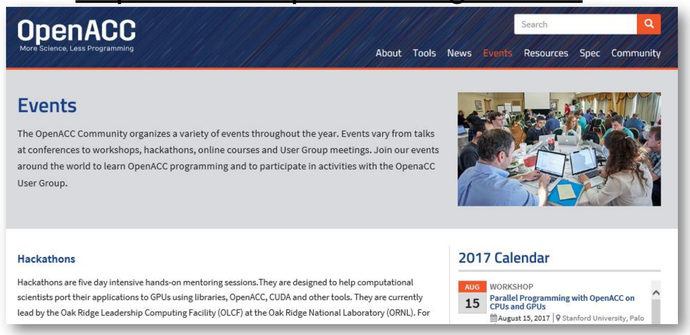

## Compilers and Tools

https://www.openacc.org/tools




## Events

https://www.openacc.org/events




OpenACC More Science, Less Programming NVIDIA

# OpenMP Offloading

Giacomo Rossi

Intel SCG/SCALE/TCE/XCSS

TREX Hackathon
March 7th 2023

# Agenda

- oneAPI
- OpenMP Offload
- oneMKL

# oneAPI

## Strategy

# What is oneAPI?

## Open-source software stack

Built with industry standard components (CLANG, LLVM, SPIR-V)

Intel LLVM | oneAPI Open-Source Projects

## An open community

github/oneAPI-TAB

| Technical Advisory Boards | Implementations | |
|---|---|---|
| SYCL | Intel | |
| oneMKL oneDNN | NVidia | AMD |
| | Xilinx | ARM |

## An open specification

Building on other open standards (SYCL 2020, OpenMP, BLAS, …)

oneAPI

oneAPI.io

creative commons

## Intel Products

Intel® oneAPI Toolkits

software.intel.com/oneapi

oneAPI Release 2023.1

## Available on

apt | GitHub | CentOS yum/dnf | intel | openSUSE zypper | nuget | CONDA | Maven | Spack

# oneAPI: the Big Picture

# OpenMP Offload


AMD
Argonne NATIONAL LABORATORY
arm
BSC
BERKELEY LAB
Lawrence Berkeley National Laboratory
Brookhaven National Laboratory
CSC
OMPcommunity
epcc
FUJITSU
Hewlett Packard Enterprise
IBM
intel
Los Alamos
lrz
MAUI HIGH PERFORMANCE COMPUTING CENTER
DSRC
DOD SUPERCOMPUTING RESOURCE CENTER
Micron
NASA
NEC
NVIDIA
OAK RIDGE National Laboratory
Pawsey
RWTH AACHEN UNIVERSITY
Sandia National Laboratories
SIEMENS
Stony Brook University
SUSE
We adapt. You succeed.
TACC
University of Basel
UNIVERSITY OF DELAWARE

# OpenMP for Accelerators

.Starting from OpenMP specifications 4.0, OpenMP introduced the "target" terminology: the programmer can offload portion of code to device other than the CPU (coprocessors, FPGAs, GPUs…)

.Version 4.5 of OpenMP specifications introduced device memory routines and a couple of constructs in order to control device data mapping

.OpenMP 5.0 specifications extended the device memory routines and improved the device support adding the declare variant construct

.OpenMP 5.1 specifications introduced Fortran interfaces to device memory routines and mainly focused on OpenMP usability on accelerators.

.OpenMP 5.2 is the last version of the specifications available.

# Introduction

```
subroutine vec_mult(p, v1, v2, n)
   double precision, intent(inout) :: p(:)
double precision, intent(in)       :: v1(:), v2(:)
integer, intent(in)                :: n
integer                            :: i

  !$omp target teams distribute parallel do simd map(to: v1(1:n), v2(1:n)) map(from: p(1:n))
  do i=1, n
    p(i) = v1(i) * v2(i)
  enddo
endsubroutine
```

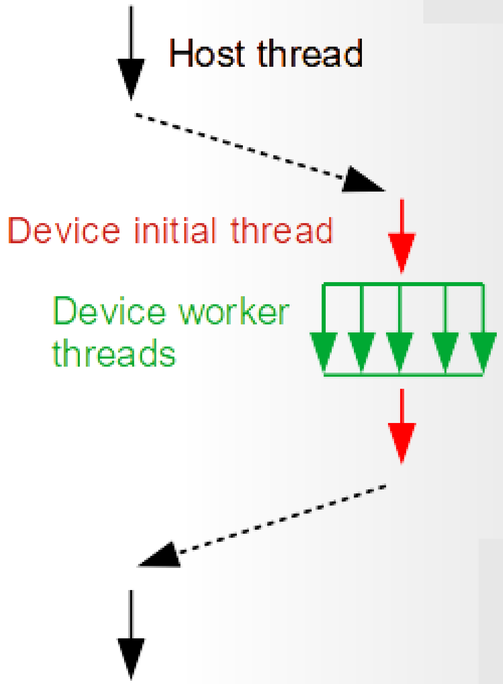
Controls data transfer between host and device

Distributes iterations to the threads and each thread use SIMD parallelism

Identifies the portion of the program that should be run in parallel on the device

# Execution model

- OpenMP heterogeneous execution model is host-centric
- The host thread that encounters a target region does not execute the target region; by defaults it waits for the execution of the target region
- A new initial thread is generated on the device
- The `target teams` construct starts a league of teams executing in parallel the subsequent code
- When the `parallel` construct is encountered by a league, each initial thread becomes the master of a new team of threads
- Each team is a contention group, so are restricted in how they can synchronize with each other



# Heterogeneous Memory Model

- OpenMP supports heterogeneous architectures by mapping variables from the host to a device
- The accelerator(s) has a device data environment that contains the set of the variables currently accessible by threads running on that device
- An original variable in host thread's data environment is mapped to a corresponding variable in the accelerator's data environment

# Device Execution Control – TARGET TEAMS Construct

`target teams` is a combined construct: it specifies that the subsequent code block should run in parallel

```
!$omp target teams[clause[[,]clause]…]
          structured block
!$omp end target teams
```

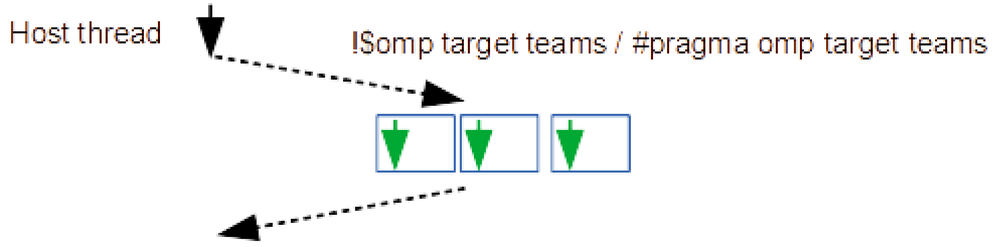
```
#pragma omp target teams[clause[[,]clause]…]
          structured block
```

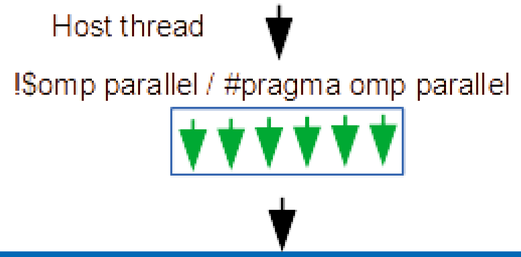
## `target teams`

- This construct starts a league of initial threads where each thread is its own team, and in its own contention group. Each initial thread executes the teams region in parallel
- Threads in different contention group cannot synchronize with each other

## `parallel`

- This construct creates a single team of threads, where each thread in the team executes the parallel region
- Threads can synchronize with each other



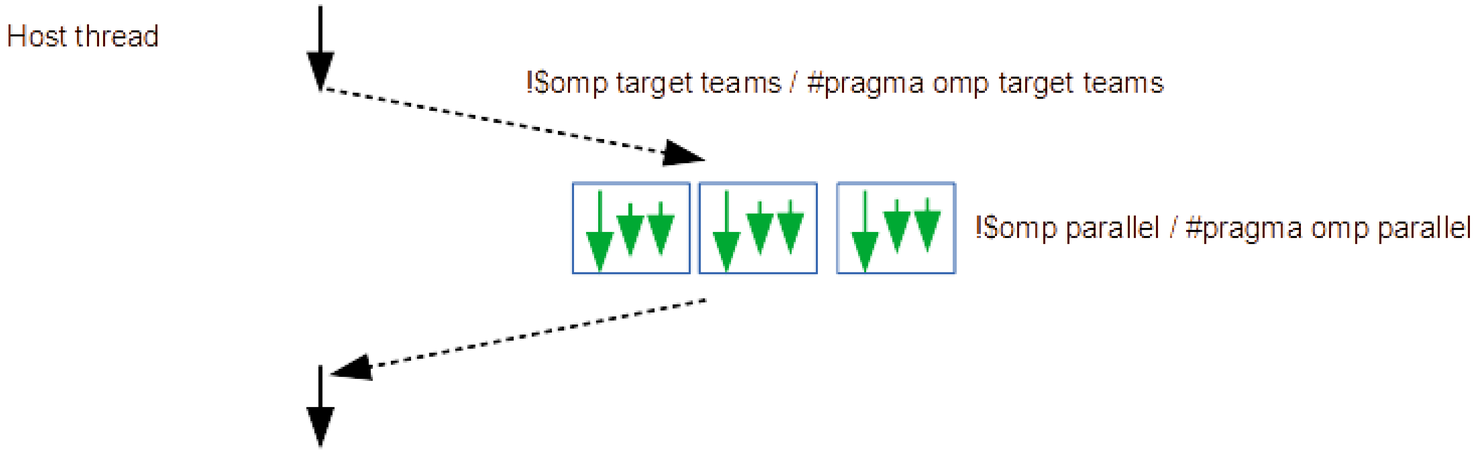



# Device Execution Control – TARGET TEAMS Construct

```
!$omp target teams[clause[[,]clause]…]
!$omp parallel
   structured block
!$omp end parallel
!$omp end target teams
```

```
#pragma omp target teams[clause[[,]clause]…]
#pragma omp parallel
          structured block
```

- When a parallel construct is encountered by a league, each thread in the league becomes the master of a new team of threads
- Each team of threads concurrently executes the parallel region



# Loop Related Directives: DISTRIBUTE

```
!$omp distribute[clause[[,]clause]…]
   do loops
!$omp end distribute
```

```
#pragma omp distribute[clause[[,]clause]…]
   do loops
```

- Distribute construct has the potential for better performance because of the restrictions on where it can be used and what other OpenMP constructs can appear inside the distribute region
- The do/for construct are more versatile but not perform as well

```
!$omp target teams num_teams(4)
!$omp distribute
do j=1, n, n/2
   !$omp parallel do
   do i=j, j+n/2
      y(i) = x(i)
   enddo
   !$omp end parallel do
Enddo
!$omp end distribute
```

Creates a league of 4 teams

Distributes the loop iterations across the master threads of each team

Activates the threads in the teams and distributes the loop iterations to the threads

# Loop Related Directives: LOOP

loop specifies that the logical iterations of the associated loops may execute concurrently

```
!$omp loop[clause[[,]clause]…]
   do loops
!$omp end loop
```

```
#pragma omp loop[clause[[,]clause]…]
   do loops
```

- Loop construct is a worksharing construct if its binding region is the innermost enclosing parallel region
- The directive asserts that the iterations of the associated loops may execute in any order, including concurrently. Each logical iteration is executed once per instance of the loop region that is encountered by exactly one thread that is a member of the binding thread set.

```
!$omp target teams num_teams(4)
!$omp loop collapse(2)
do j=1, n
   y(i) = x(i)
!$omp end parallel do
enddo
!$omp end distribute
```

Creates a league of 4 teams

Distributes the loop iterations across the master threads of each team

Activates the threads in the teams and distributes the loop iterations to the threads

# Combine and Composite Accelerated Worksha ring Constructs

The combined constructs has the same execution behavior of separate constructs, but in some instances, depending on the compiler, the combined constructs may achieve better performance than individual constructs and in such way is possible to distribute loop iterations across multiple levels of parallelism without needing multiple loop nests

```
!$omp target teams distribute parallel do [clause[[,] clause]…]
      do-loops
!$omp end target teams distribute parallel do


!$omp target teams distribute simd [clause[[,] clause]…]
      do-loops
!$omp end target teams distribute simd



!$omp target teams distribute parallel do simd [clause[[,] clause]…]
      do-loops
!$omp end target teams distribute parallel do simd



!$omp target teams loop [clause[[,] clause]…]
      do-loops
!$omp end target teams loop
```

# Data Mapping

- The accelerator has its own data environment which contains all set of all variables that are available to the threads executing on that accelerator
- When a host variable is mapped to an accelerator, a corresponding variable is allocated in the accelerator's device data environment
- Host and device variables may share the same storage location → synchronization and memory consistency are required to avoid data races
- Host and device may not share the same storage location → copy operations are required in order to make original and corresponding variable consistent

# map Clause

```
map([[map-type-modifier[,] [map-type-modifier[,] ...]] map-type: ] locator-list)
```

Map type:

- `to`
- `from`
- `tofrom`
- `alloc`
- `release`
- `delete`

Map type modifiers:

- `always`
- `close`
- `mapper`
- `present`
- `iterator`

Map clause on a target constructs:

- `target`
- `target data`
- `target enter data`
- `target exit data`

- Defaults:
- Arrays are `tofrom`
- Scalars are `firstprivate`
- Be careful about pointers: their memory address may not exist on the device!

BE AWARE OF FORTRAN DERIVED TYPES / C STRUCTS!!!

# always map modifier

This modifier forces OpenMP runtime to map data even if such data is already present in the device memory space:

- OpenMP runtime performs presence check and if data is present, mapping clause is most likely translated in a target update

```
!$omp target teams
distribute parallel do
map(always,to:a)
```

```
!$omp target update to(a)
!$omp target teams
distribute parallel do
```

```
!$omp target data
map(always,from:a)
...
!$omp end target data
```

```
...
!$omp target update fro m(a
                          )
```

# Mapping in `target` construct

- `map` clause on a target construct:

A. map variables for a single target region

B. enclosed region executes on device and maps data

```
!$omp target map ([[map-type-modifier[,]] map-type:] list)
...
!$omp end target map
```

```
#pragma omp target map ([[map-type-modifier[,]] map-type:] list)
...
```

# Incremental porting and data mapping

```
subroutine foo(x)
implicit none

integer, intent(inout) :: x(:)
integer :: n, i

   do i=1, n
      x(i) = x(i) + i
   enddo
   ...
endsubroutine foo
```

Original subroutine, no offload

Modified subroutine with OpenMP offload

```
subroutine foo(x)
implicit none

integer, intent(inout) :: x(:)
integer :: n, i

   !$omp target teams distribute parallel do map(tofrom:x)
   do i=1, n
      x(i) = x(i) + i
   enddo

endsubroutine foo
```

# Incremental porting and data mapping (continued)

```
subroutine foo_caller(x,a)
implicit none

integer, intent(inout) :: x(:), a(:)
integer :: n, i

   do i=1, n
      a(i) = x(i) + a(i)
   enddo
   call foo_second(x) !CPU routine that
                      !MODIFIES x ONLY on the
                      !CPU!!!
   call foo(x)
   ...
endsubroutine foo_caller
```

Modified subroutine with OpenMP offload: your program will give wrong results (or will crash)

Original subroutine, no offload: main program works also when subroutine foo is offloaded

```
subroutine foo_caller(x,a)
implicit none

integer, intent(inout) :: x(:), a(:)
integer :: n, i

   !$omp target data map(to:x) map(tofrom:a)

   !$omp target teams distribute parallel do
   do i=1, n
      a(i) = x(i) + a(i)
   enddo

   call foo_second(x) !CPU routine that MODIFIES x
                      !ONLY on the CPU!!!


   call foo(x)

   !$omp end target data
   ...
endsubroutine foo_caller
```

# defaultmap clause

```
!$omp target defaultmap(implicit-behavior[:variable-category])
...
!$omp end target
```

```
#pragma omp target defaultmap(implicit-behavior[:variable-category])
...
```

- explicitly determines the data-mapping attributes of variables referenced in the region

Implicit behavior
- alloc
- to, from, tofrom
- firstprivate
- none
- default
- present

Variable category
- scalar
- aggregate
- pointer
- allocatable (Fortran only)
- all

# Implicit vs Explicit Mapping

Default map for arrays is `tofrom`

```
double precision :: x(1:n), y(1:n)
integer :: i, n

!$omp target
do i=1, n
  y(i) = i+n
enddo

do i=1, n
  x(i) = y(i)
end do
!$omp end target
```

4 data transfers

```
double precision :: x(1:n), y(1:n)
integer :: i, n

!$omp target map(alloc:y) map(from:x)
do i=1, n
  y(i) = i+n
enddo

do i=1, n
  x(i) = y(i)
end do
!$omp end target
```

1 data transfer

# Mapping in `target data` construct

`target data`:

- map variables across multiple target regions in a structured block
- enclosed region does not execute on the device, only maps data from host to device

```
!$omp target data [clause[[,] clause]…]
structured block
!$omp end target data
```

```
integer :: i, n
double precision :: v1(1:n), v2(1:n)
!$omp target map(to: v1, v2) map(from: p)
!$omp parallel do
do i=1, n
  p(i) = v1(i) * v2(i)
enddo
!$omp end parallel do
!$omp end target

do other stuffs on v1 and v2

!$omp target map(to: v1, v2) map(tofrom: p)
!$omp parallel do
do i=1, n
  p(i) = p(i) + v1(i) * v2(i)
enddo
!$omp end parallel do
!$omp end target
```

```
integer :: i, n
double precision :: v1(1:n), v2(1:n)
!$omp target data map (from:p)
!$omp target map(to: v1, v2)
!$omp parallel do
do i=1, n
  p(i) = v1(i) * v2(i)
enddo
!$omp end parallel do
!$omp end target

do other stuffs on v1 and v2

!$omp target map(to: v1, v2)
!$omp parallel do
do i=1, n
  p(i) = p(i) + v1(i) * v2(i)
enddo
!$omp end parallel do
!$omp end target
!$omp end target data
```

# `declare target` construct

`declare target`:

- allows global variables to be mapped on an accelerator's device data environment for the whole execution of the program
- - if a function is called from a target region, then the name of the function must appear in a declare target directive

```
!$omp declare target (extended list)
or
!$omp declare target [clause[[,] clause]…]
```

```
#pragma omp declare target (extended list)
or
#pragma omp declare target [clause[[,] clause]…]
```

BE AWARE OF FORTRAN ALLOCATABLES / C POINTERS!!!

# declare target construct(II)

```
module my_arrays
!$omp declare target (N, p, v1, v2)
integer, parameter :: N=1000
real :: p(N), v1(N), v2(N)
end module

subroutine vec_mult()
use my_arrays
integer :: I
call init(v1, v2, N);
!$omp target update to(v1, v2)
!$omp target
!$omp parallel do
do i = 1,N
    p(i) = v1(i) * v2(i)
end do
!$omp end target
!$omp target update from (p)
call output(p, N)
end subroutine
```

# target enter / target exit data constructs

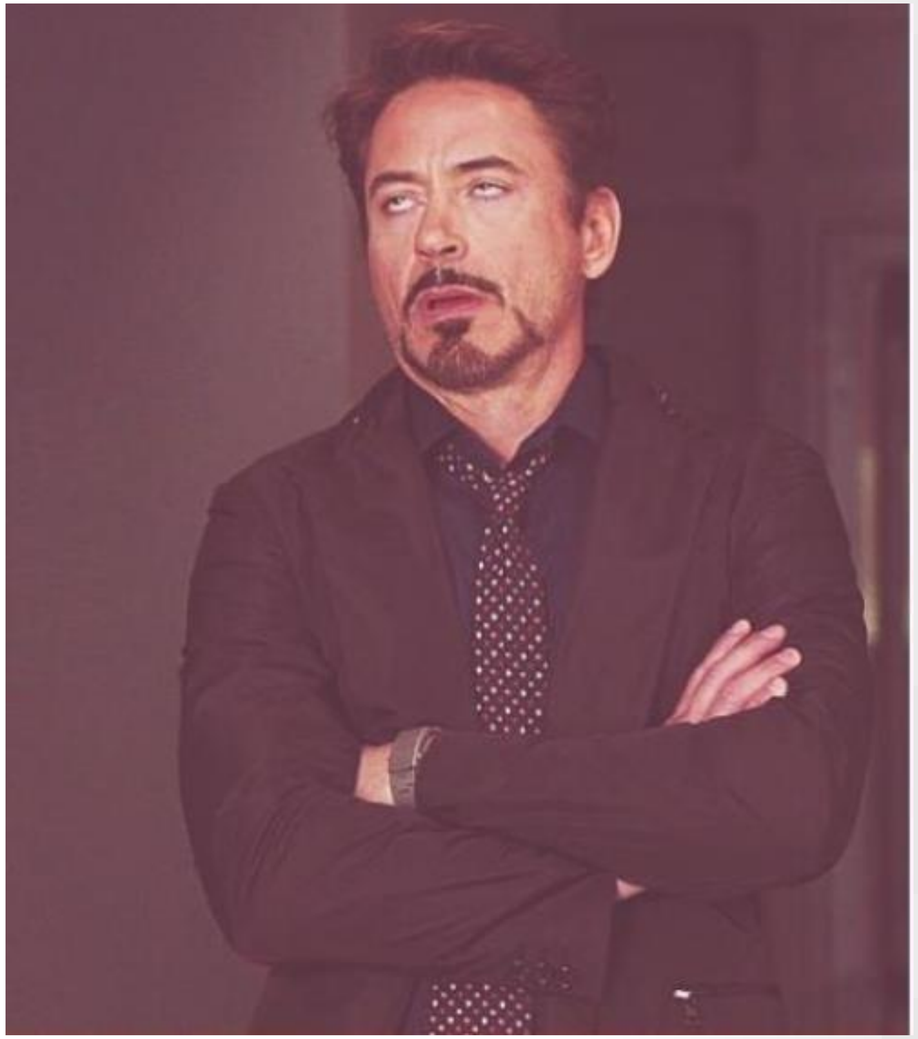
- map variables in stand alone clauses, not associated with a statement or a structured block

```
!$omp target enter data [clause[[,] clause]…]
or
!$omp target exit data [clause[[,] clause]…]
```

```
module example
real(8), allocatable :: A(:), B(:)
contains

subroutine initialize(N) integer :: N
allocate(A(N)); allocate(B(N))
!$omp target enter data map(alloc:A,B)
end subroutine initialize

subroutine finalize()
!$omp target exit data map(delete:A) map(from:B)
deallocate(A, B)
end subroutine finalize
end module example
```

# Avoid Unnecessary Data Transfers

- Don’t rely on implicit mapping! Use map clause to specify when a variable needs to be copied to or from the device.
- Use target data regions around structured blocks to avoid mapping variables unnecessarily
- Use target enter/exit data and target declare data to manage data transfer more explicitly



# Data-Sharing Attribute Clauses

- These clauses allow the user to control data-sharing attributes of variables referenced in a construct.
- Concerning offload, four clauses are available:

```
is_device_ptr

use_device_ptr

has_device_addr

use_device_addr
```

# is_device_ptr clause

- Can be used in `target` or `dispatch` directives
- It indicates that the list items are device pointers: so each list item is privatized inside the construct and the new list item is initialized to the device address to which the original list item refers.
- In Fortran, each list item should be of type C_PTR

```
...
  arr_host = (int *) malloc(N * sizeof(int));
  arr_device = (int *) omp_target_alloc(N * sizeof(int), omp_get_default_device());

  #pragma omp target is_device_ptr(arr_device) map(from: arr_host[0:N])
  {
    for (int i = 0; i < N; ++i) {
      arr_device[i] = i;
      arr_host[i] = arr_device[i];
    }
  }
...
```

# use_device_ptr clause

- Can be use in `target data` directive
- It indicates that each list item is a pointer to an object that has corresponding storage on the device or is accessible on the device.
- In Fortran, each list item should be of type C_PTR

```
...
  A = (double *) omp_target_alloc(bytes, device_id);
  if (A == NULL){
     printf(" ERROR: Cannot allocate space for A using omp_target_alloc_device().\n");
     exit(1);
  }

  B = (double *) omp_target_alloc(bytes, device_id);
  if (B == NULL){
     printf(" ERROR: Cannot allocate space for B using omp_target_alloc_device().\n");
     exit(1);
  }

  #pragma omp target data use_device_ptr(A,B)
  {
      #pragma omp target teams distribute parallel for
      for (size_t i=0; i<length; i++) {
          A[i] = 2.0;
          B[i] = 2.0;
  }
...
```

# has_device_addr clause

- Can be use in `target` directive
- It indicates that the list items already have valid device addresses, and therefore may be directly accessed from the device

```
...
  real(kind=REAL64), allocatable ::  A(:)
  real(kind=REAL64), allocatable ::  B(:)
  !
  ! Allocate arrays in device memory

  !$omp allocate allocator(omp_target_device_mem_alloc) !Intel Extension
  allocate(A(length))

  !$omp allocate allocator(omp_target_device_mem_alloc)
  allocate(B(length))

  !
  ! Initialize the arrays

  !$omp target teams distribute parallel do has_device_addr(A, B)
  do i = 1, length
      A(i) = 2.0
      B(i) = 2.0
  end do
...
```

# use_device_addr clause

- Can be use in `target data` directive
- It indicates that the list items already have valid device addresses, and therefore may be directly accessed from the device

```
...
  real(kind=real64), parameter :: aval = real(42, real64)
  real(kind=real64), allocatable :: array_d(:), array_h(:)
  integer :: i,err

  ! Allocate host data
  allocate(array_h(N1))

  !$omp target data map (from:array_h(1:N1)) map(alloc:array_d(1:N1))
  !$omp target data use_device_addr(array_d)
  !$omp target
      do i=1, N1
        array_d(i) = aval
        array_h(i) = array_d(i)
     end do
  !$omp end target
  !$omp end target data
  !$omp end target data
...
```

# declare variant directive

- Declare base function / routines to have the specified function variant
- The context selector in the `match` clause is associated with the variant

```
! In oneMKL OpenMP offload interface (file mkl_blas_omp_offload_lp64.f90)
module subroutine mkl_blas_dgemm_omp_offload_ilp64 ( transa, transb, m, n, k, alpha, &
   &a, lda, b, ldb, beta, c, ldc ) BIND(C)

   character*1,intent(in) :: transa, transb
   integer,intent(in) :: m, n, k, lda, ldb, ldc
   double precision,intent(in) :: alpha, beta
   double precision,intent(in) :: a( lda, * ), b( ldb, * )
   double precision,intent(inout) :: c( ldc, * )
end subroutine mkl_blas_dgemm_omp_offload_ilp64

subroutine dgemm ( transa, transb, m, n, k, alpha, a, lda, &
   &b, ldb, beta, c, ldc ) BIND(C)

   character*1,intent(in) :: transa, transb
   integer,intent(in) :: m, n, k, lda, ldb, ldc
   double precision,intent(in) :: alpha, beta
   double precision,intent(in) :: a( lda, * ), b( ldb, * )
   double precision,intent(inout) :: c( ldc, * )

   !$omp declare variant( dgemm:mkl_blas_dgemm_omp_offload_ilp64 ) match( construct={dispatch}, device={arch(gen)} ) &
   !$omp& append_args(interop(targetsync)) adjust_args(need_device_ptr:a,b,c)

end subroutine dgemm
```

# dispatch construct

- Controls whether variant substitution occurs for target call in the associated function dispatch structured block

```
include 'mkl_omp_offload.f90'
SUBROUTINE laxlib_cdiaghg_gpu( n, m, h, s, ldh, e, v, me_bgrp, root_bgrp,
intra_bgrp_comm )
!                                                                                !
USE laxlib_parallel_include
#if defined(MKL_ILP64)
   USE onemkl_lapack_omp_offload_ilp64
#else
   USE onemkl_lapack_omp_offload_lp64
#endif
IMPLICIT NONE
...
   !$omp target data map(to:h, s) map(from:m, e, v, lwork, rwork, ifail, info)
   !$omp dispatch is_device_ptr(h, s, m, e, v, lwork, rwork, ifail, info)
   CALL ZHEGVX( 1, 'V', 'I', 'U', n, h, ldh, s, ldh, &
                0.D0, 0.D0, 1, m, abstol, mm, e, v, ldh, &
                work, lwork, rwork, iwork, ifail, info )
   !$omp end target data
...
END SUBROUTINE laxlib_cdiaghg_gpu
```

# Device Memory Routines

OpenMP 4.5 introduced device memory routines for C/C++, to support pointers allocation/deallocation and management in the data environment of target devices:

```
void* omp_target_alloc(size_t size, int device_num);

void omp_target_free(void *device_ptr, int device_num);

int omp_target_is_present(void *ptr, int device_num);


int omp_target_memcpy(void *dst, void *src, size_t length, size_t dst_offset, size_t src_offset, int
dst_device_num, int src_device_num);


int omp_target_memcpy_rect( void *dst, void *src, size_t element_size, int num_dims, const size_t*
volume, const size_t* dst_offsets, const size_t* src_offsets, const size_t* dst_dimensions, const
size_t* src_dimensions, int dst_device_num, int src_device_num);


int omp_target_associate_ptr(void *host_ptr, void *device_ptr, size_t size, size_t device_offset,
int device_num);


int omp_target_disassociate_ptr(void *ptr, int device_num);
```

# Device Memory Routines (II)

OpenMP 5.1 added:

```
int omp_target_is_accessible( const void *ptr, size_t size, int
device_num);

int omp_target_memcpy_async( void *dst, const void *src, size_t length,
size_t dst_offset, size_t src_offset, int dst_device_num, int
src_device_num, int depobj_count, omp_depend_t *depobj_list );

int omp_target_memcpy_rect_async( void *dst, const void *src, size_t
element_size, int num_dims, const size_t *volume, const size_t
*dst_offsets, const size_t *src_offsets, const size_t *dst_dimensions,
const size_t *src_dimensions, int dst_device_num, int src_device_num, int
depobj_count, omp_depend_t *depobj_list );
```

OpenMP 5.1 added also Fortran interfaces to all device memory routines

# Runtime Routines and Environment Variables

- set the default device:

```
subroutine omp_set_default_device(device_num)
integer :: device_num
```

- get the default device:

```
integer function omp_get_default_device()
```

```
int omp_get_default_device(void);
```

- get the number of target devices:

```
integer function omp_get_num_devices()
```

```
int omp_get_num_devices(void);
```

- find out if we are on the host:

```
logical function omp_is_initial_device()
```

```
int omp_is_initial_device(void);
```

- find out the device number of the host:

```
integer function omp_get_initial_device()
```

```
int omp_get_initial_device(void);
```

# Runtime Routines and Environment Variables (II)

- get the total number of teams:

```
integer function omp_get_num_teams()
```

```
int omp_get_num_teams(void);
```

- get the team number:

```
integer function omp_get_team_num()
```

```
int omp_get_team_num(void);
```

**OMP_DEFAULT_DEVICE** → Set the default device

**OMP_TARGET_OFFLOAD={"mandatory" | "disabled" | "default" }** →

# Intel Offload Runtime Environment Variables

- **`LIBOMPTARGET_PLUGIN=<Name>:`** Designates offload plugin name to use. Offload runtime does not try to load other RTLs if this option is used.

```
<Name> := LEVEL0 | OPENCL | CUDA | X86_64 | NIOS2 |
          level0 | opencl | cuda | x86_64 | nios2
```

- **`LIBOMPTARGET_DEBUG:`** Control whether or not debugging information will be displayed

  1 → basic information (device detection, kernel compilation, memory copy operations, kernel invocations, and other plugin-dependent actions)

  2 → additionally displays which GPU runtime API functions are invoked with which arguments/parameters

- **`LIBOMPTARGET_INFO:`** Allows the user to request different types of runtime information from **`libomptarget`**
- **`LIBOMPTARGET_PLUGIN_PROFILE=<Enable>[,<Unit>]:`** Enables basic plugin profiling and displays the result when program finishes. Microsecond is the default unit if ``<Unit>`` is not specified.
- **`LIBOMPTARGET_DEVICETYPE=<Type>:`** Decides which device type is used. Only OpenCL plugin supports "CPU" device type.

```
<Type> := GPU | gpu | CPU | cpu
```

# LIBOMPTARGET_PLUGIN_PROFILE

```
================================================================================================================================
LIBOMPTARGET_PLUGIN_PROFILE(LEVEL0) for OMP DEVICE(0) Intel(R) Data Center GPU Max 1550, Thread 0
--------------------------------------------------------------------------------------------------------------------------------

Kernel 0 :___omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_c2psi_gamma_omp__l457
Kernel 1 :___omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_c2psi_gamma_omp__l463
Kernel 2 :___omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_c2psi_gamma_omp__l469
Kernel 3 :___omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_psi2c_gamma_omp__l948
Kernel 4 :___omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_psi2c_gamma_omp__l956
Kernel 5 :___omp_offloading_802_262342_qe_drivers_lda_lsda_mp_xc_lda__l95
Kernel 6 :___omp_offloading_802_822655_v_xc__l474
Kernel 7 :___omp_offloading_802_822655_v_xc__l488
Kernel 8 :___omp_offloading_802_885338_vloc_psi_gamma__l120
Kernel 9 :___omp_offloading_802_885338_vloc_psi_gamma__l134
--------------------------------------------------------------------------------------------------------------------------------

: Host Time (msec) Device Time (msec)
Name : Total Average Min Max Total Average Min Max Count
--------------------------------------------------------------------------------------------------------------------------------

Compiling : 193.28 193.28 193.28 193.28 0.00 0.00 0.00 0.00 1.00
DataAlloc : 7.12 0.00 0.00 2.16 0.00 0.00 0.00 0.00 3453.00
DataRead (Device to Host) : 42.46 0.02 0.01 0.30 3.03 0.00 0.00 0.03 2772.00
DataWrite (Host to Device): 75.48 0.02 0.01 0.52 8.62 0.00 0.00 0.05 4164.00
Kernel 0 : 2.50 0.04 0.02 0.21 0.71 0.01 0.01 0.08 56.00
Kernel 1 : 1.20 0.03 0.03 0.07 0.40 0.01 0.01 0.01 40.00
Kernel 2 : 0.54 0.03 0.03 0.09 0.18 0.01 0.01 0.01 16.00
Kernel 3 : 1.26 0.03 0.03 0.08 0.44 0.01 0.01 0.03 40.00
Kernel 4 : 0.51 0.03 0.02 0.08 0.15 0.01 0.01 0.03 16.00
Kernel 5 : 2.09 0.42 0.05 1.86 0.12 0.02 0.02 0.02 5.00
Kernel 6 : 4.00 0.80 0.11 3.55 0.21 0.04 0.03 0.06 5.00
Kernel 7 : 1.01 0.20 0.16 0.36 0.65 0.13 0.13 0.13 5.00
Kernel 8 : 1.77 0.03 0.02 0.10 0.50 0.01 0.01 0.01 56.00
Kernel 9 : 1.49 0.03 0.02 0.06 0.44 0.01 0.01 0.01 56.00
Linking : 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 1.00
OffloadEntriesInit : 8.96 8.96 8.96 8.96 0.00 0.00 0.00 0.00 1.00
================================================================================================================================
```

# LIBOMPTARGET_DEBUG

```
Libomptarget --> Launching target execution_omp_offloading_802_262342_qe_drivers_lda_lsda_mp_xc_lda_l95 with pointer 0x000000000bf358a0 (index=38).
Target LEVEL0 RTL --> Executing a kernel 0x000000000bf358a0...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred team size is multiple of 64
Target LEVEL0 RTL --> Loop 0: lower bound = 0, upper bound = 91124, Stride = 1
Target LEVEL0 RTL --> Team sizes = {64, 1, 1}
Target LEVEL0 RTL --> Number of teams = {1424, 1, 1}
Target LEVEL0 RTL --> Kernel Pointer argument 0 (value: 0xff00ffffff400000) was set successfully for device 0.
Target LEVEL0 RTL --> Kernel Scalar argument 7 (value: 0x00000000000163f5) was set successfully for device 0.
...
Target LEVEL0 RTL --> Kernel Scalar argument 9 (value: 0x00000000000163f5) was set successfully for device 0.
Target LEVEL0 RTL --> Kernel Scalar argument 26 (value: 0x00000000000163f5) was set successfully for device 0.
Target LEVEL0 RTL --> Kernel Scalar argument 27 (value: 0x00000000000163f5) was set successfully for device 0.
Target LEVEL0 RTL --> Submitted kernel 0x000000000c4fc420 to device 0
Target LEVEL0 RTL --> Executed kernel entry 0x000000000bf358a0 on device 0
...
Libomptarget --> Looking up mapping(HstPtrBegin=0x0000153919c6e900, Size=729000)...
Libomptarget --> Mapping exists with HstPtrBegin=0x0000153919c6e900, TgtPtrBegin=0xff00ffffff400000, Size=729000, DynRefCount=1 (decremented), HoldRefCount=0
Libomptarget --> There are 729000 bytes allocated at target address 0xff00ffffff400000 - is not last
Libomptarget --> Entering target region with entry point 0x0000000000ef14bc and device Id 0
Libomptarget --> Call to omp_get_num_devices returning 1
Libomptarget --> Call to omp_get_num_devices returning 1
Libomptarget --> Call to omp_get_initial_device returning 1
Libomptarget --> Checking whether device 0 is ready.
Libomptarget --> Is the device 0 (local ID 0) initialized? 1
Libomptarget --> Device 0 is ready to use.
Libomptarget --> Entry 0: Base=0x0000000007f790b0, Begin=0x0000000007f790b0, Size=8, Type=0x223, Name=v_xc_$ETXC
...
Libomptarget --> Entry 6: Base=0x00007ffdbac90900, Begin=0x00007ffdbac90908, Size=88, Type=0x5000000000001, Name=v_xc_$VX_dv_len
...
Libomptarget --> Entry 54: Base=0x0000000000000000, Begin=0x0000000000000000, Size=0, Type=0x120, Name=unknown
Libomptarget --> Looking up mapping(HstPtrBegin=0x0000000007f790b0, Size=8)...
Target LEVEL0 RTL --> Ptr 0x0000000007f790b0 requires mapping
Libomptarget --> Creating new map entry with HstPtrBegin=0x0000000007f790b0, TgtPtrBegin=0xff00ffffffffa0800, Size=8, DynRefCount=1, HoldRefCount=0, Name=v_xc_$ETXC
Libomptarget --> Moving 8 bytes (hst:0x0000000007f790b0) -> (tgt:0xff00ffffffffa0800)
Target LEVEL0 RTL --> Copied 8 bytes (hst:0x0000000007f790b0) -> (tgt:0xff00ffffffffa0800)
Libomptarget --> There are 8 bytes allocated at target address 0xff00ffffffffa0800 - is new
Libomptarget --> Looking up mapping(HstPtrBegin=0x0000000007f7aed0, Size=8)...
Libomptarget --> Moving 8 bytes (hst:0x0000000007f7aed0) -> (tgt:0xff00ffffffffa02c0)
Target LEVEL0 RTL --> Copied 8 bytes (hst:0x0000000007f7aed0) -> (tgt:0xff00ffffffffa02c0)
Libomptarget --> Mapping exists (implicit) with HstPtrBegin=0x0000153919d22980, TgtPtrBegin=0xff00ffffff700000, Size=729000, DynRefCount=2 (incremented), HoldRefCount=0, Name=v_xc_$V
Libomptarget --> There are 729000 bytes allocated at target address 0xff00ffffff700000 - is not new
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffdbac90900, Size=96)...
Libomptarget --> Mapping exists with HstPtrBegin=0x00007ffdbac90900, TgtPtrBegin=0xff00ffffffff80500, Size=96, DynRefCount=2 (incremented), HoldRefCount=0, Name=v_xc_$VX
Libomptarget --> There are 96 bytes allocated at target address 0xff00ffffffff80500 - is not new
```

# OpenMP 5.x: *standard ready to compete with OpenACC*

## Hierarchical parallelism

```
#pragma omp target teams distribute
for (...) {
  #pragma omp parallel for
  for (...) {
    #pragma omp simd
    for (...) {
    }
  }
}
```

## Unified Shared Memory

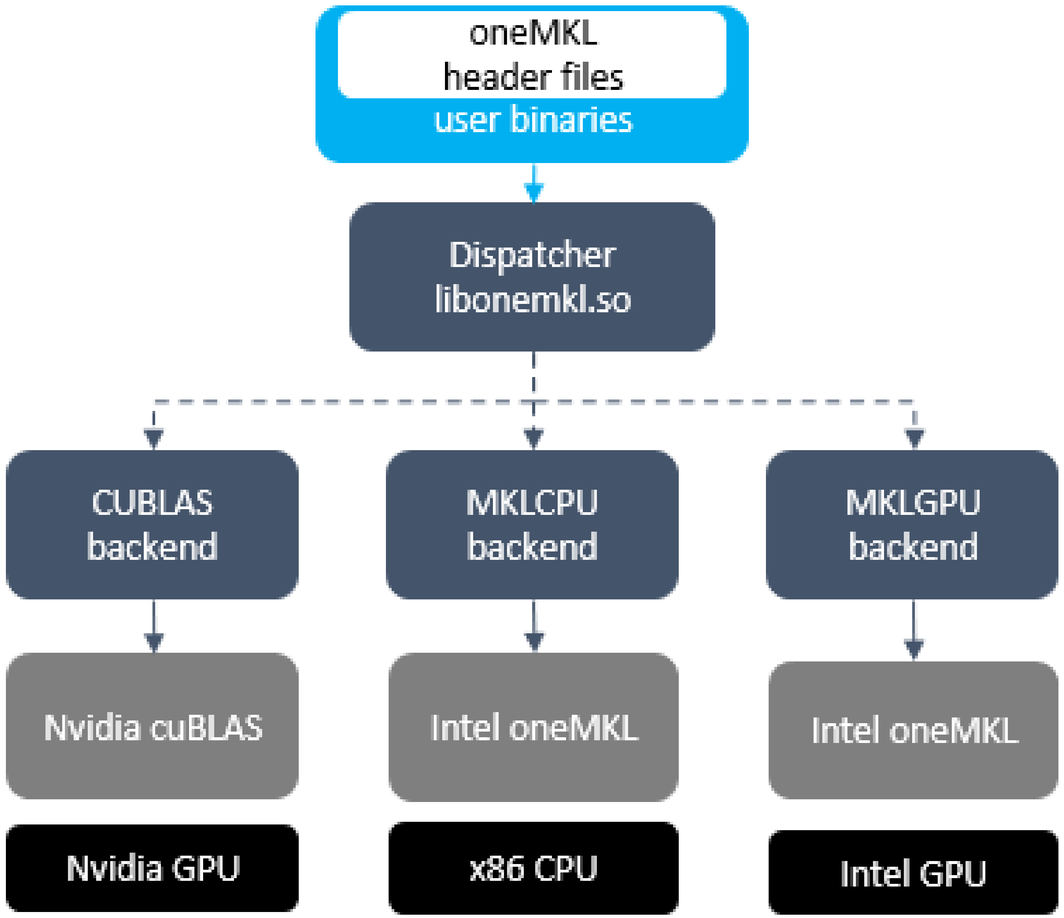
```
#pragma omp requires unified_address
A = omp_target_alloc_shared(...);
```

## Or explicit control of data movement

```
int *arr_host = malloc(...);
int *arr_device = omp_target_alloc_device(...);
#pragma omp target is_device_ptr(arr_device)
#pragma omp target map(tofrom: arr_host[0:N])
```

# OneMKL

# oneMKL defines common interface to multiple targets



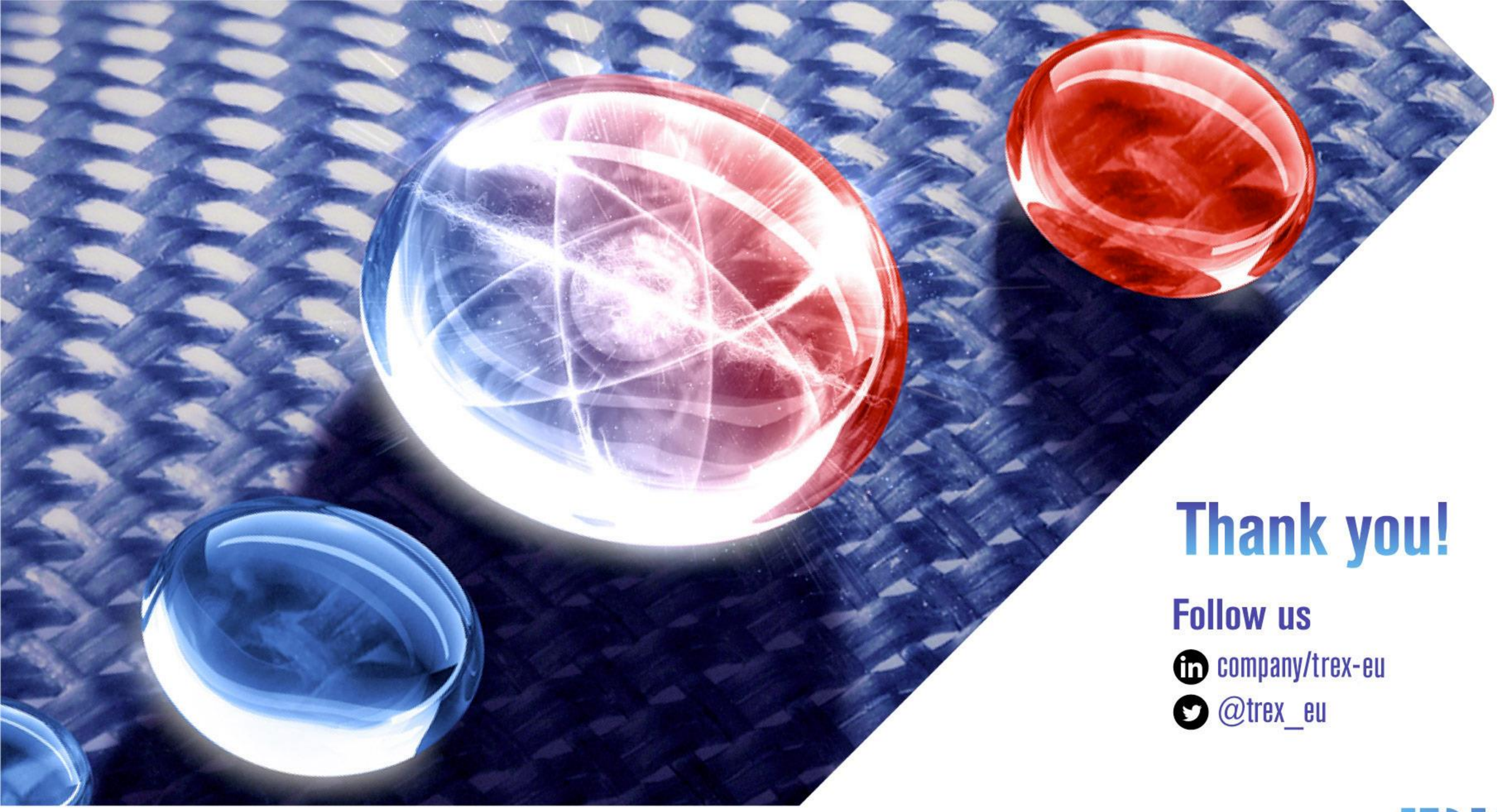

```
// C := alpha*(AxB)+ beta*C

gemm(q,
     transA, transB,
     m, n, k, alpha,
     A, ldA, B, ldB,
     beta, C, ldC);
```

GEMM is called from the host

Call is routed to appropriate library by the queue

Dispatch can be dynamic or determined statically

# Thank You



# Thank you!

## Follow us

company/trex-eu

@trex_eu

Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation programme under Grant Agreement **No. 952165.**

TREX
Targeting Real chemical accuracy at the EXascale