VSC Users Day 2019

Start to GPU

Ehsan Moravveji

Outline

- A brief intro
- Available GPUs at VSC
- GPU architecture
- How to port CPU code to GPU
- Two benchmarking results

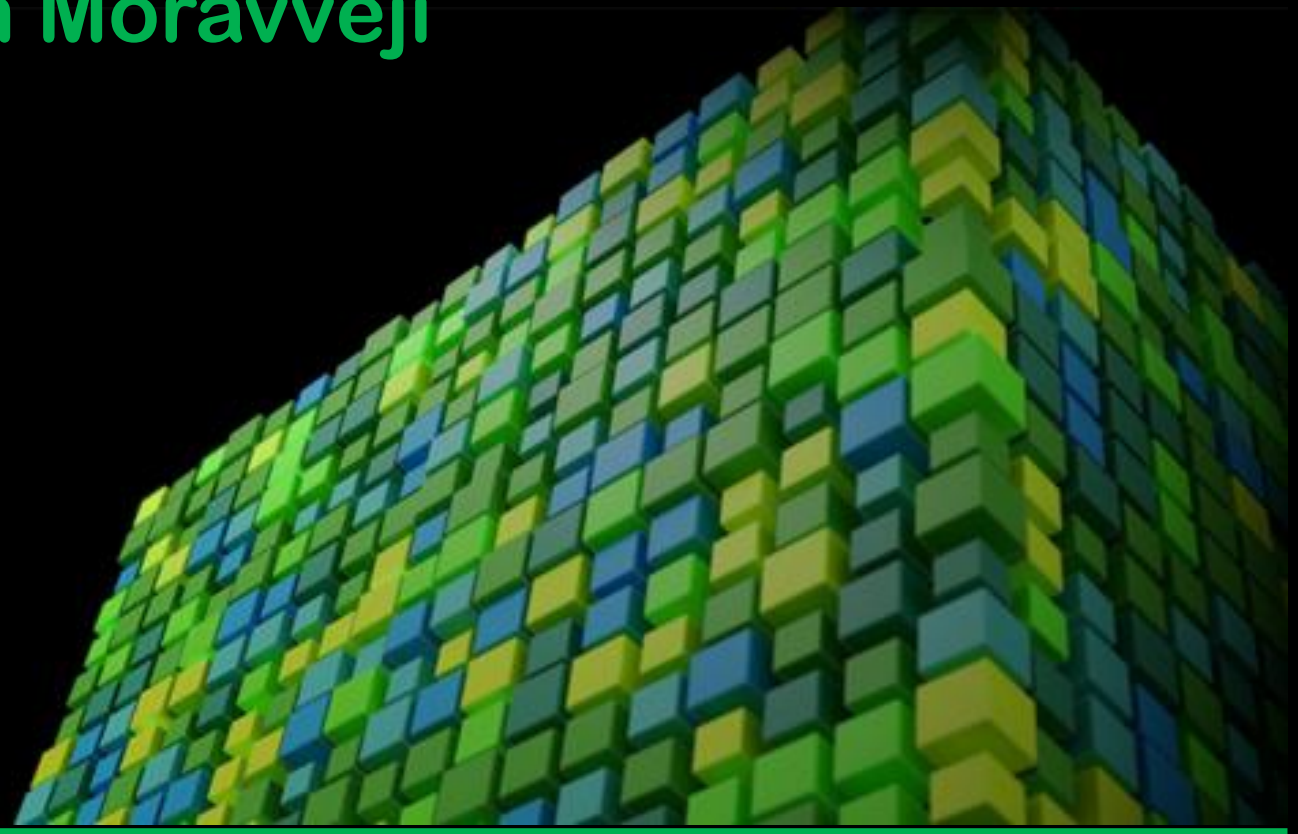
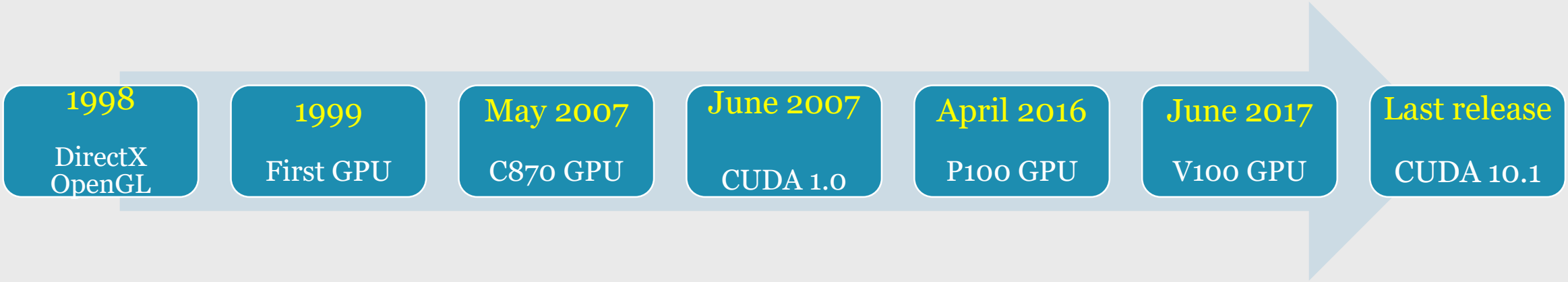


Image courtesy of Nvidia.com

Generally speaking …

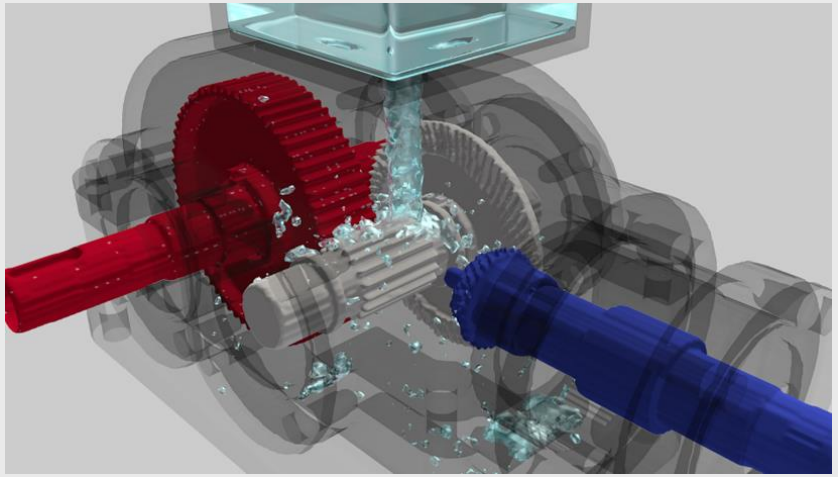


- Fast way to create images in frame buffers
- Main Suppliers: Nvidia and AMD
- Usage: (3D) Graphics rendering, Gaming, Virtual Reality, and Computation
- Used in: computers, self-driving cars , HPC clusters
- Revenue in 2017 (billion dollars): Nvidia 9.7 and AMD 5.3

From Graphics to Numerics

Pinterest: River Raid

innovationintelligence.com



- SIMT (single instruction multiple threads) architecture for
  + high-throughput analysis (e.g. ML, AI)
  + embarrassingly parallel tasks (e.g. Monte Carlo simulations, grids, FFT, …)
- Double-precision, single-precision and half-precision (tensor cores)
- Math function evaluation on the device, e.g. sin, cos, exp, …
- 1st, 2nd and 7th top supercomputers are Green for using GPUs (Top500, Nov. 2018).

Available GPUs at VSC

Ugent
No GPUs

UAntwerpen
2 nodes, each 2x P100

VUB
6 nodes, each 2x K20
4 nodes, each 2x P100
1 node, 4x GeForce GTX 1080

KU Leuven/UHasselt
8 nodes, each 2x K20
5 nodes, each 2x K40
20 nodes, each 4x P100

+ few visualization nodes at each site

Available GPUs at VSC

	Tesla K20 (Leu, Bru)	Tesla K40 (Leu)	GeForce GTX 1080 (Bru)	Pascal P100 (Leu, Ant, Bru)
SP cores	14x192=**2,688**	15x192=**2880**	28x128=**3584**	56x64=**3584**
DP cores	14x64=**896**	15x64=**960**	28x32=**896**	56x32=**1792**
Clock freq. (MHz)	732	745	1582	1481
DRAM (GB)	5.7	11.2	12	16
DRAM freq. (GHz)	2.6 (384-bit)	3.0 (384-bit)	5.5 (352-bit)	0.71 (4096-bit)
Compute capability	3.5	3.5	6.1	6.0
L2 cache (MB)	1.5	1.5	2.75	4.0
Constant mem. (KB)	64	64	64	64
Shared mem. per block (KB)	48	48	48	48
Registers per block (x1024)	64	64	64	64

GPU Architecture

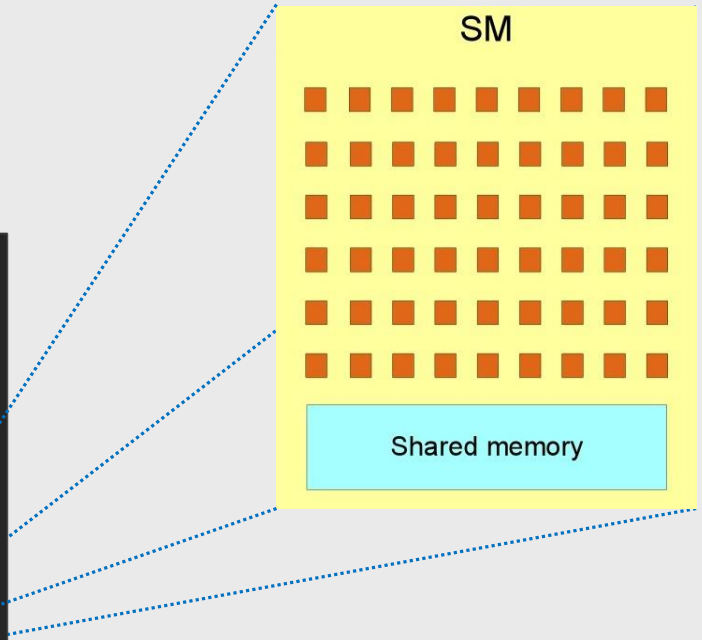
TESLA V100

21B transistors
815 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink

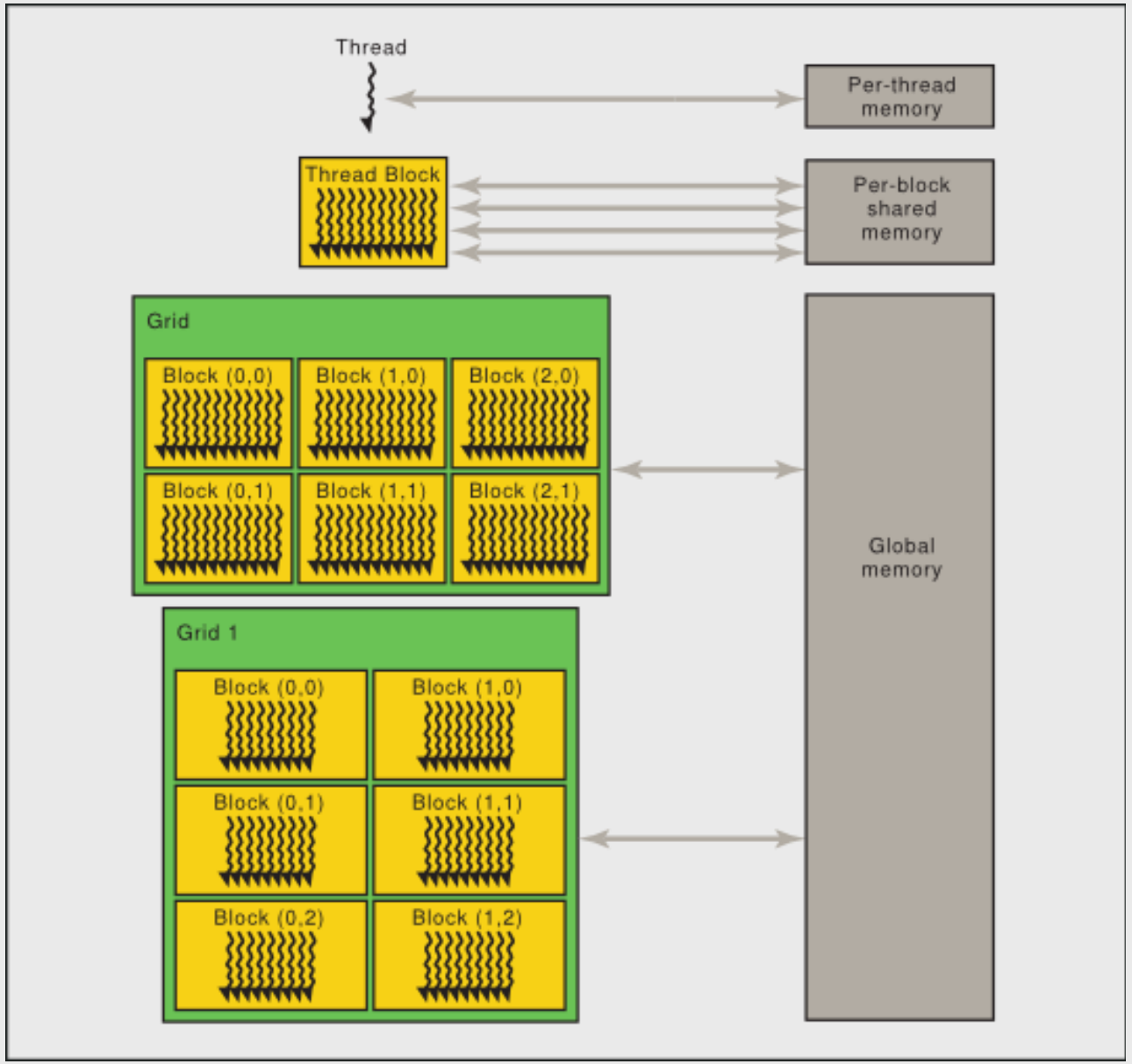
*full GV100 chip contains 84 SMs



https://www.servethehome.com/nvidia-v100-volta-update-hot-chips-2017/

Device Memory

Which?	Access
Register	Thread-private
Shared memory	Block of threads
Global memory	All threads
Texture memory	All threads



(Image from CUDA C Reference Guide)

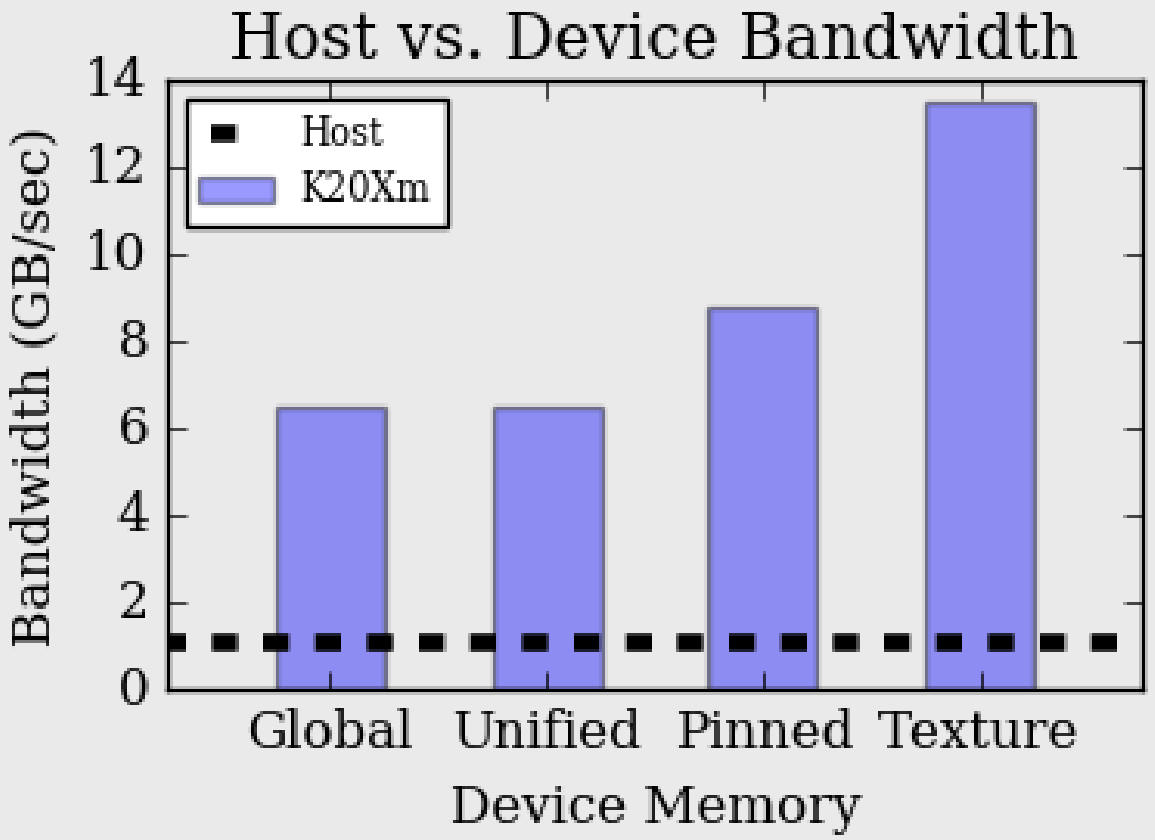
Device Memory

- Take an arbitrary function
  $y_k = b_k^2 \sin(a_k) + a_k \exp(\cos(b_k))$
- Timing: Host ←→ Device data transfer plus kernel launch

- Bandwidth (GB/sec):

$$BW = \frac{(R + W)/10^9}{\Delta t}$$

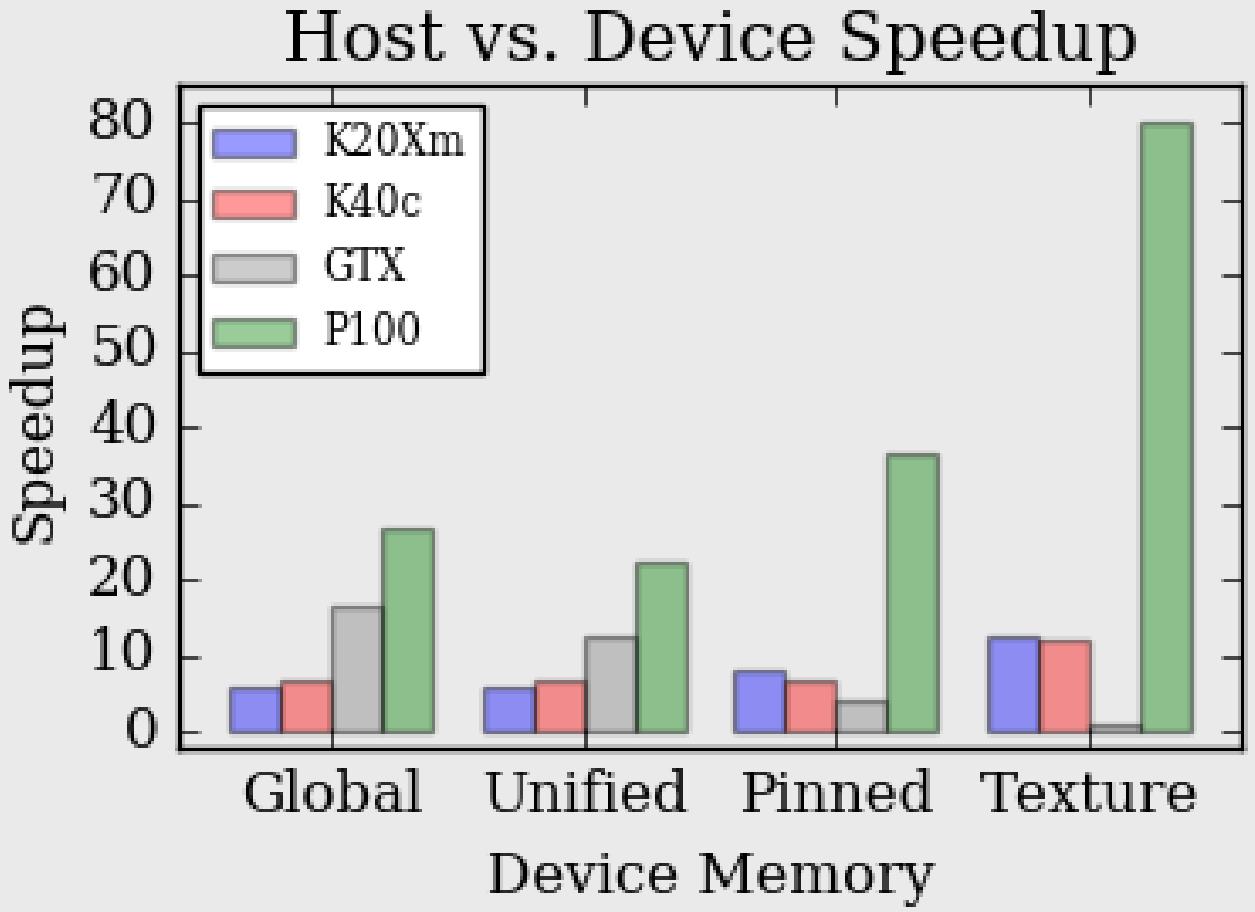
R: num. bytes to Read
W: num. bytes to Write
Δt: elapsed time



Data size: 160 MB, single-precision arrays
Host: Intel Xeon® E5-2630 @2.30GHz
PCI-Express SAS2308 Fusion-MPT SAS-2 (rev 01)

Device Memory

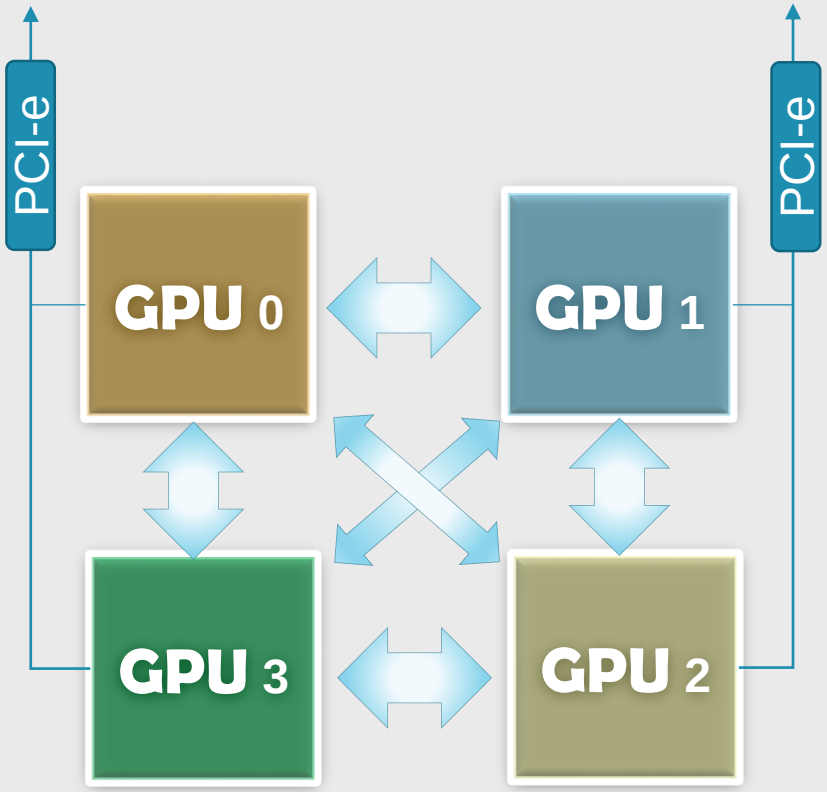
- Define Speedup: S= $\frac{T_{CPU}}{T_{GPU}}$
- Recent devices (P100) give 3x to 8x speedup.
- Unified memory has the same performance as of global memory, hence, cleaner code
- Transfer using pinned (page-locked) memory is faster than global memory
- For single-precision read-only data, texture memory offers the best speedup



K20Xm (IvyBridge); K40c (Haswell); P100 (SkyLake)

Peer-to-Peer Bandwidth

High Med Low



Dev. ID	0	1	2	3
0	509	10	19	18
1	10	508	18	18
2	19	18	508	10
3	18	18	10	507

Bi-directional P2P: **PCIe**

Dev. ID	0	1	2	3
0	508	37	37	61
1	37	507	61	37
2	36	61	508	37
3	62	37	37	506

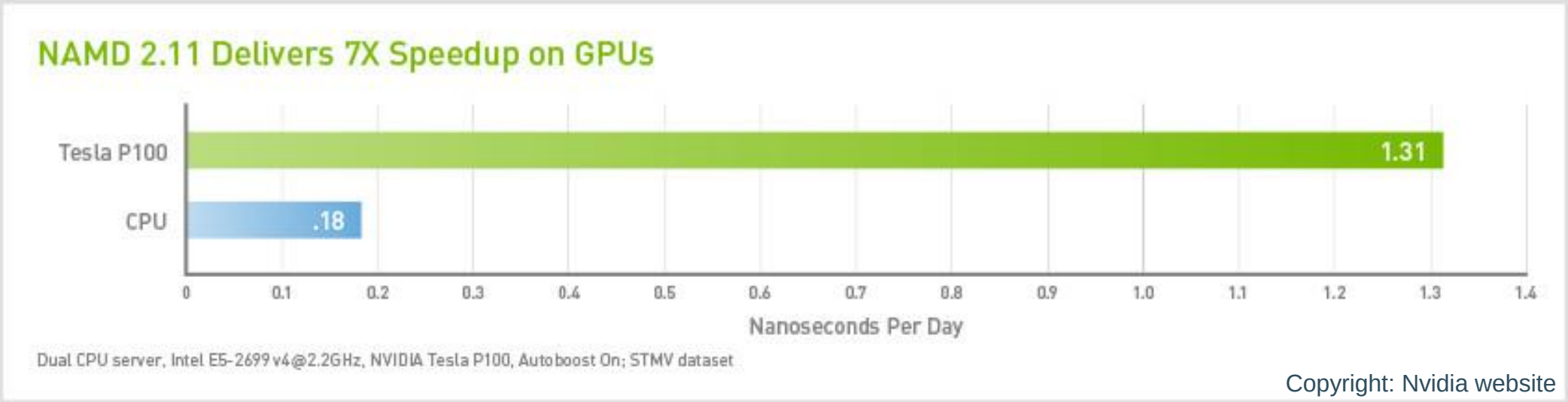
Bi-directional P2P: **NVLink**
(P100@Leuven)

How to Start-to-GPU?

Approach 1: Users

Does your software already use GPUs?

- Check Nvidia Application Catalog: (https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/)
- **Machine Learning:** Tensorflow, Keras, PyTorch, CAFFE2, …
- **Chemistry:** Abinit, BigDFT, CP2K, Gaussian, QuantumEspresso, BEAGLE-lib, VASP, …
- **Phys. & Eng.:** OpenFOAM, Fluent, COSMO, …
- **Biophysics:** NAMD, CHARM, GROMACS, …
- **Tools:** Alinea-Forge, Cmake, MAGMA,



How to Start-to-GPU?

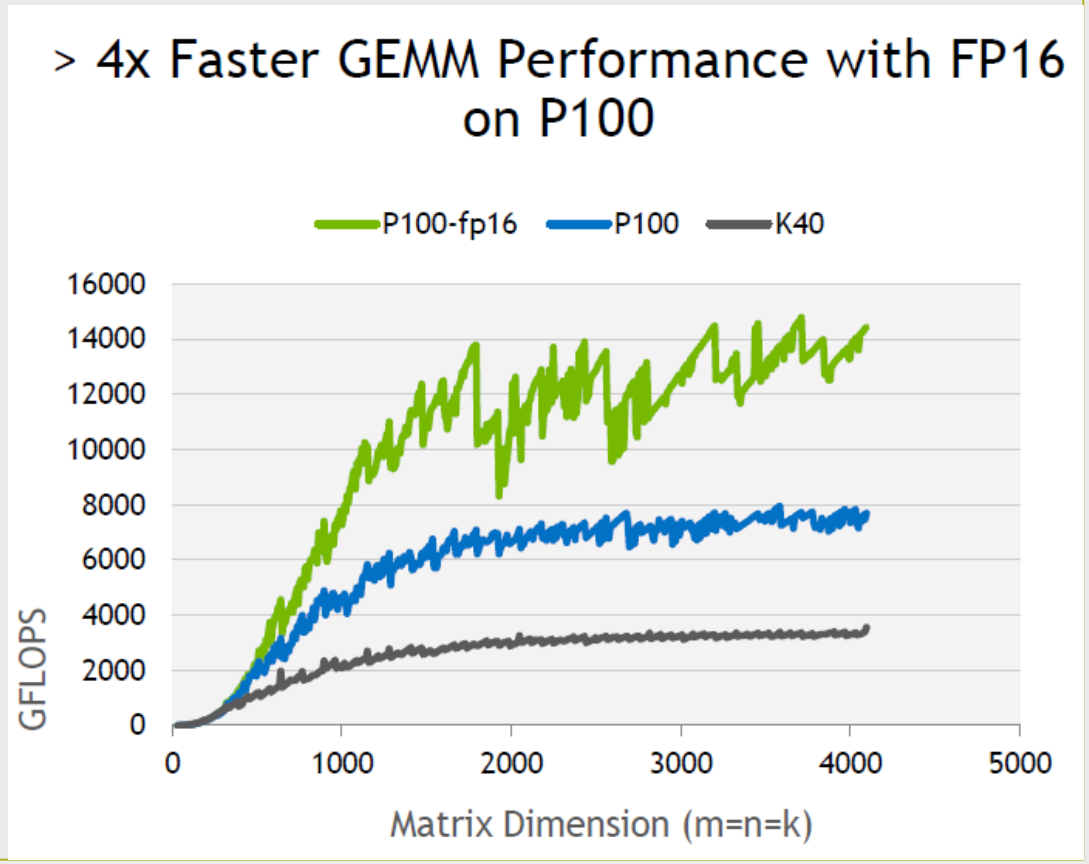
Approach 2: Porting

Incrementally porting your code to use GPUs!

- Check Nvidia Libraries: https://developer.nvidia.com/gpu-accelerated-libraries
- ✓ cuBLAS
- cuFFT
- cuSPARSE
- cuRAND
- THRUST

- Replace function calls in your application with one from the CUDA libraries. E.g. SGEMM(…) → cublasSgemm(…)

(Image taken from Nvidia CUDA 9.2 Libraries)



How to Start-to-GPU?

Approach 3: Developer

Tailor your software development to the GPU hardware!

High-level APIs

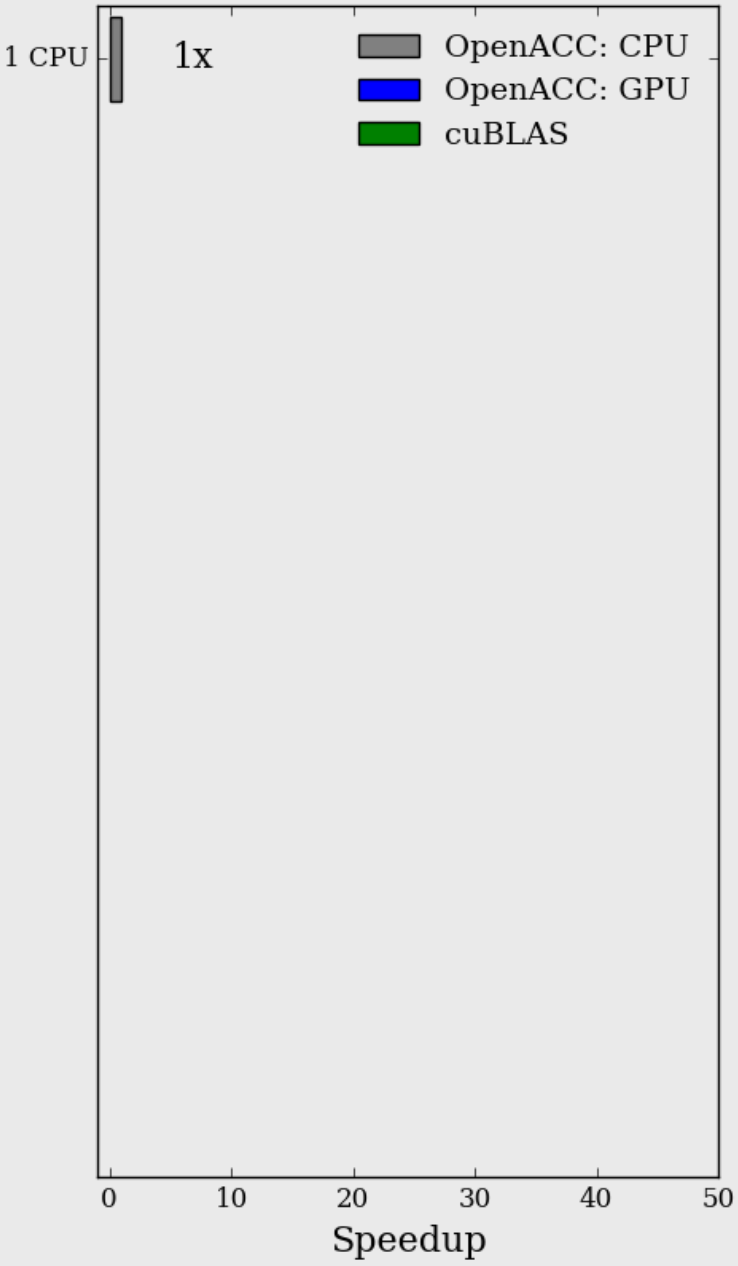
Python: Numba, pyCUDA, Quasar
Matlab: Overloaded functions and gpuArrays
R: rCUDA, rpud

Low-level APIs

Language Directives	**OpenACC**
	CUDA CUF Kernels
Programming Model	CUDA (C/C++/Fortran)
	OpenCL

Matrix Multiplication – CPU code

```
1
2     call cpu_time(tic)
3     do j = 1, p
4        do i = 1, n
5           val = 0d0
6           do k = 1, m
7              val = val + a(i, k) * b(k, j)
8           enddo
9           c(i, j) = val
10       enddo
11    enddo
12    call cpu_time(toc)
13    dt = toc - tic
14
```



$a(n,m);\ b(m,p); c(n,p);\ n = 5500; m = 3400, p = 4000$

$c_{i,j} = \sum_{k=1}^{N} a_{i,k}.b_{k,j};$

Single-core runtime (Intel SkyLake) about 53 seconds.

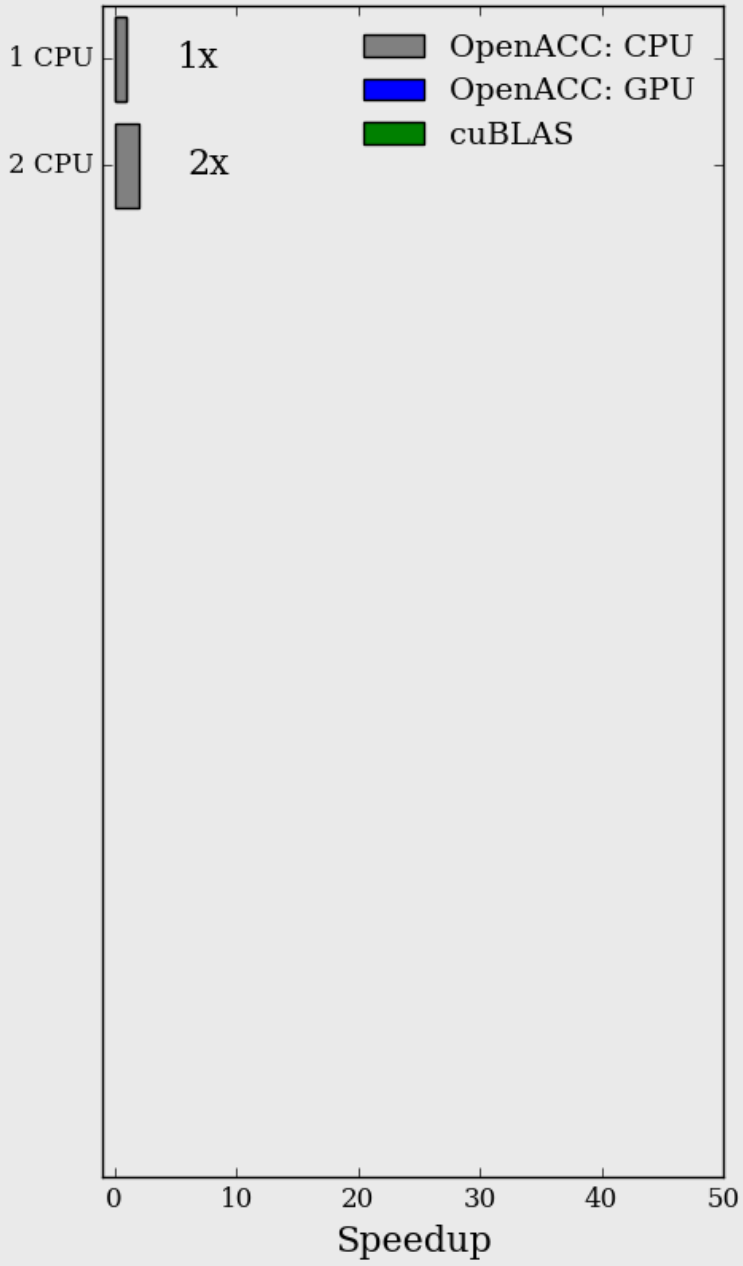
Matrix Multiplication – OpenACC

```
1   call cpu_time(tic)
2   !$acc kernels copyin(a, b) copyout(c)
3   do j = 1, p
4      do i = 1, n
5         val = 0d0
6         do k = 1, m
7            val = val + a(i, k) * b(k, j)
8         enddo
9         c(i, j) = val
10     enddo
11  enddo
12  !$acc end kernels
13  call cpu_time(toc)
14  dt = toc - tic
```

Multicore compilation

```
$> pgfortran -Mprof=ccff -acc -ta=multicore
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



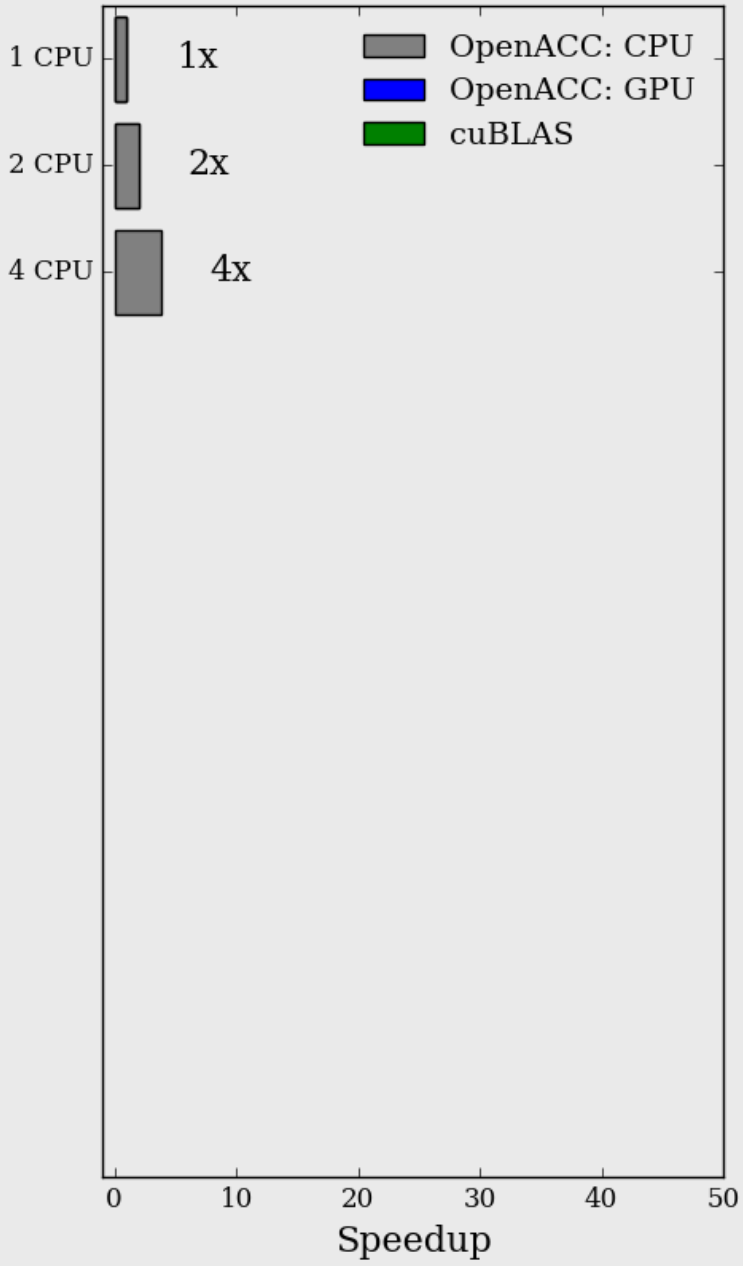
Matrix Multiplication – OpenACC

```
1   call cpu_time(tic)
2   !$acc kernels copyin(a, b) copyout(c)
3   do j = 1, p
4      do i = 1, n
5         val = 0d0
6         do k = 1, m
7            val = val + a(i, k) * b(k, j)
8         enddo
9         c(i, j) = val
10     enddo
11  enddo
12  !$acc end kernels
13  call cpu_time(toc)
14  dt = toc - tic
```

Multicore compilation

```
$> pgfortran -Mprof=ccff -acc -ta=multicore
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



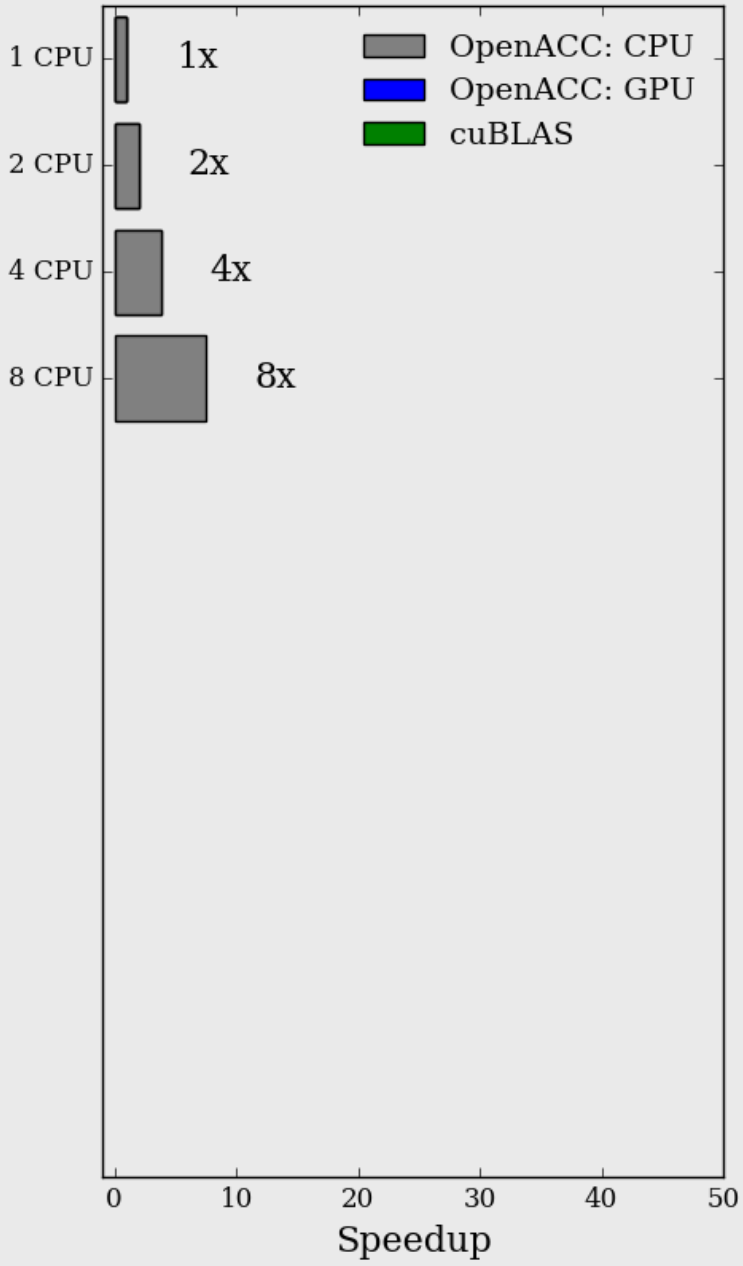
Matrix Multiplication – OpenACC

```
1   call cpu_time(tic)
2   !$acc kernels copyin(a, b) copyout(c)
3   do j = 1, p
4      do i = 1, n
5         val = 0d0
6         do k = 1, m
7            val = val + a(i, k) * b(k, j)
8         enddo
9         c(i, j) = val
10     enddo
11  enddo
12  !$acc end kernels
13  call cpu_time(toc)
14  dt = toc - tic
```

Multicore compilation

```
$> pgfortran -Mprof=ccff -acc -ta=multicore
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



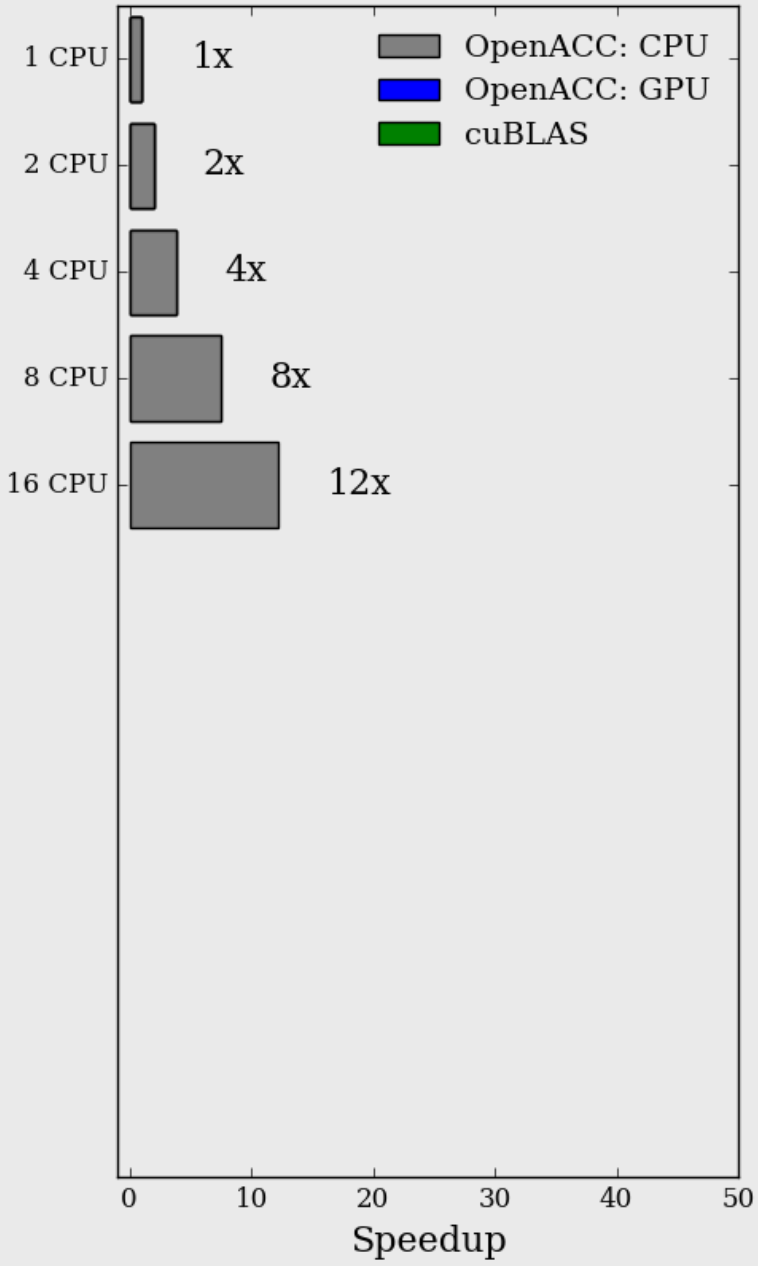
Matrix Multiplication – OpenACC

```
1   call cpu_time(tic)
2   !$acc kernels copyin(a, b) copyout(c)
3   do j = 1, p
4      do i = 1, n
5         val = 0d0
6         do k = 1, m
7            val = val + a(i, k) * b(k, j)
8         enddo
9         c(i, j) = val
10     enddo
11  enddo
12  !$acc end kernels
13  call cpu_time(toc)
14  dt = toc - tic
```

Multicore compilation

```
$> pgfortran -Mprof=ccff -acc -ta=multicore
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



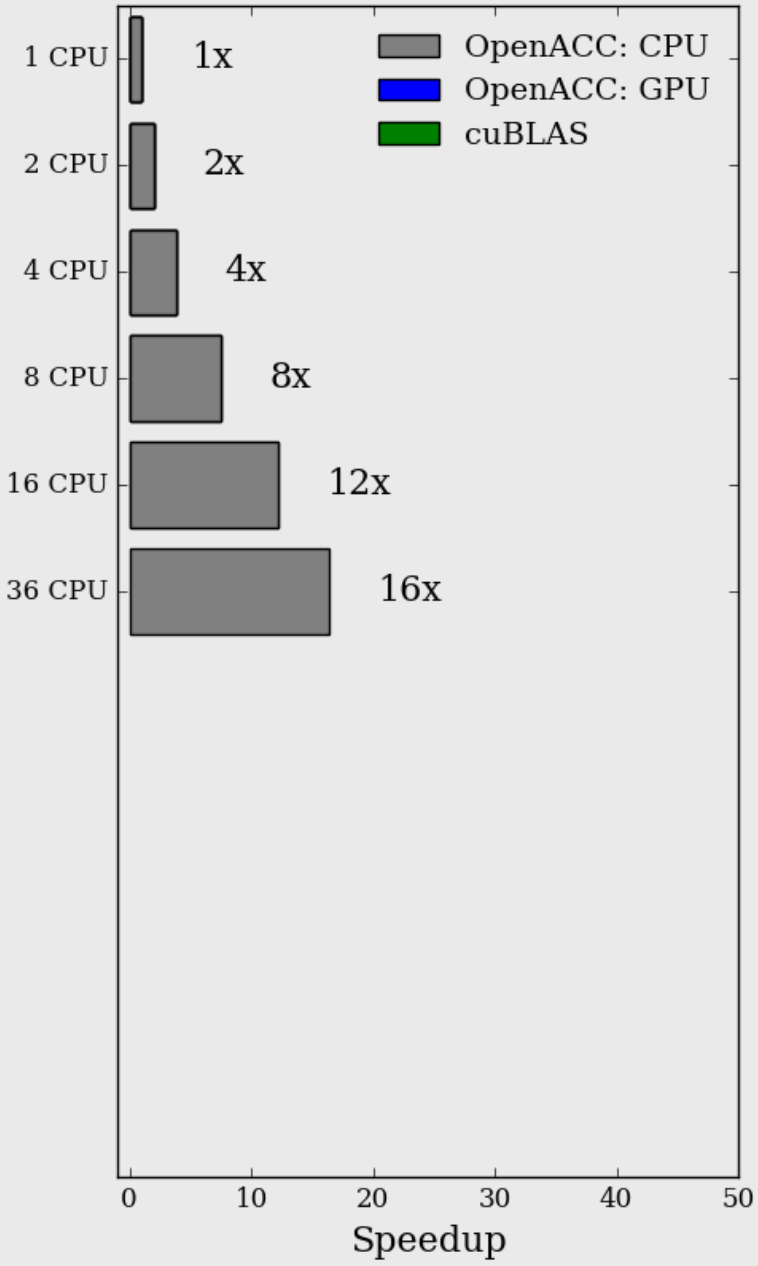
Matrix Multiplication – OpenACC

```
1   call cpu_time(tic)
2   !$acc kernels copyin(a, b) copyout(c)
3   do j = 1, p
4      do i = 1, n
5         val = 0d0
6         do k = 1, m
7            val = val + a(i, k) * b(k, j)
8         enddo
9         c(i, j) = val
10     enddo
11  enddo
12  !$acc end kernels
13  call cpu_time(toc)
14  dt = toc - tic
```

Multicore compilation

```
$> pgfortran -Mprof=ccff -acc -ta=multicore
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



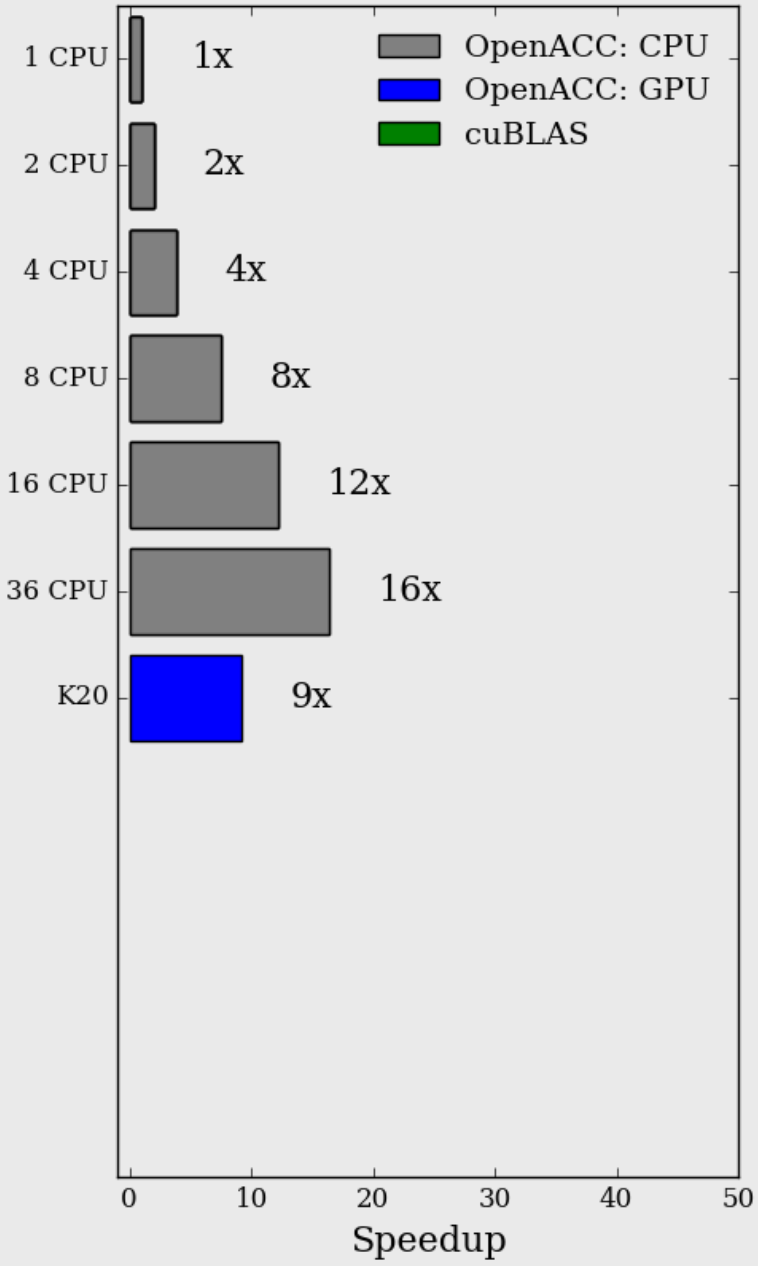
Matrix Multiplication – OpenACC

```
1   call cpu_time(tic)
2   !$acc kernels copyin(a, b) copyout(c)
3   do j = 1, p
4      do i = 1, n
5         val = 0d0
6         do k = 1, m
7            val = val + a(i, k) * b(k, j)
8         enddo
9         c(i, j) = val
10     enddo
11  enddo
12  !$acc end kernels
13  call cpu_time(toc)
14  dt = toc - tic
```

GPU compilation

```
$> pgfortran -Mprof=ccff -acc -ta=tesla:cuda8.0,cc35,llvm
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



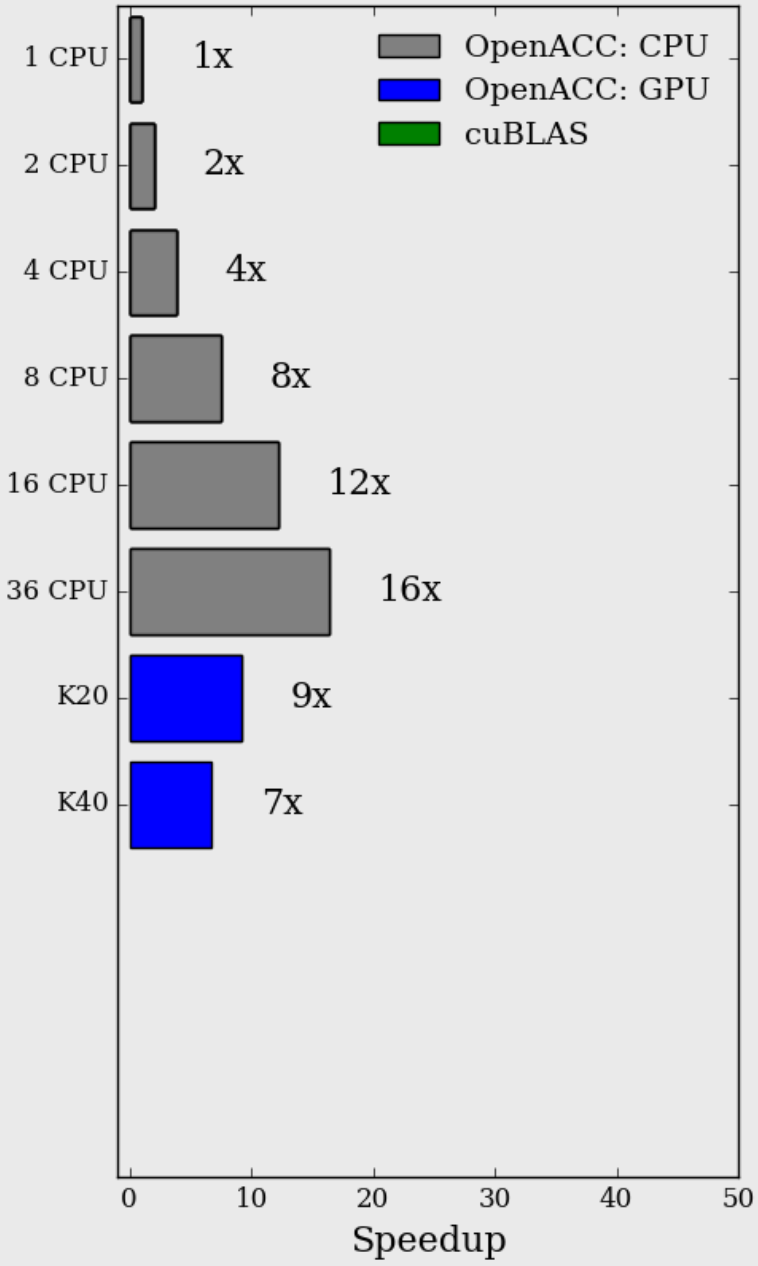
Matrix Multiplication – OpenACC

```
1   call cpu_time(tic)
2   !$acc kernels copyin(a, b) copyout(c)
3   do j = 1, p
4      do i = 1, n
5         val = 0d0
6         do k = 1, m
7            val = val + a(i, k) * b(k, j)
8         enddo
9         c(i, j) = val
10     enddo
11  enddo
12  !$acc end kernels
13  call cpu_time(toc)
14  dt = toc - tic
```

GPU compilation

```
$> pgfortran -Mprof=ccff -acc -ta=tesla:cuda8.0,cc35,llvm
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



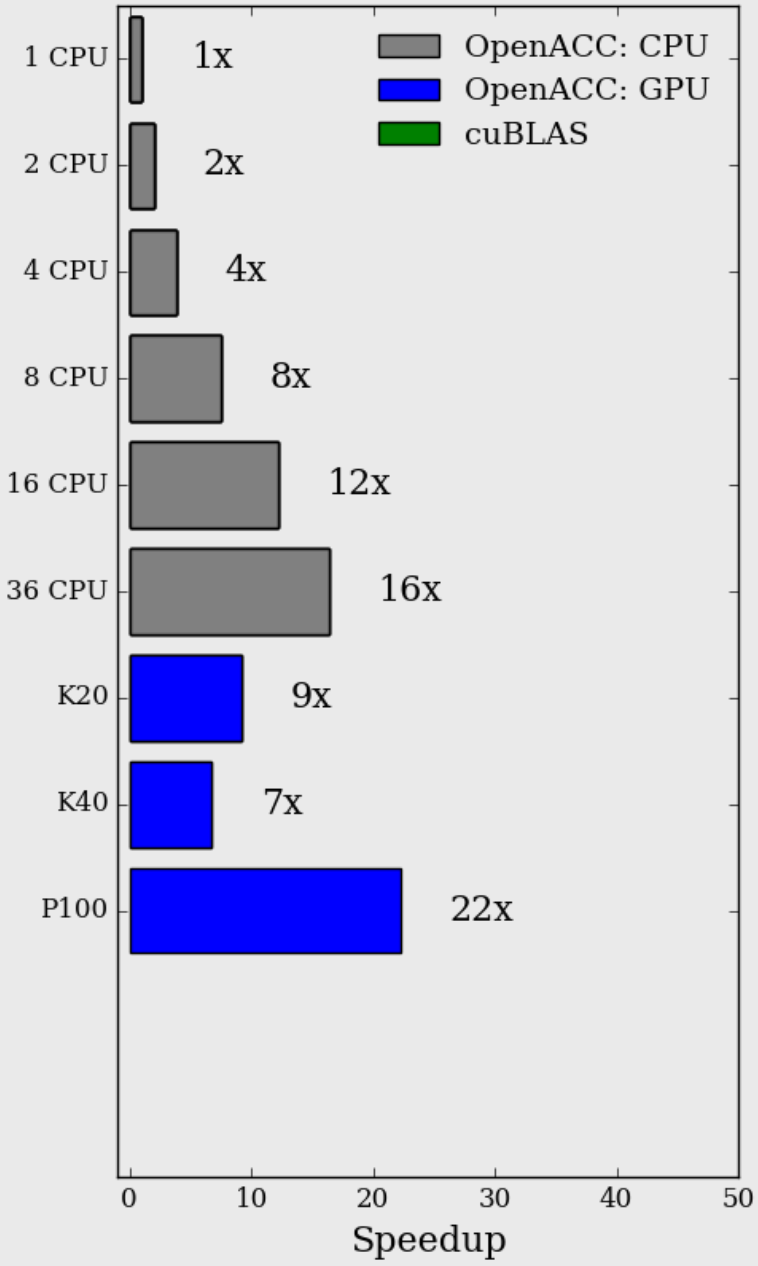
Matrix Multiplication – OpenACC

```
1   call cpu_time(tic)
2   !$acc kernels copyin(a, b) copyout(c)
3   do j = 1, p
4      do i = 1, n
5         val = 0d0
6         do k = 1, m
7            val = val + a(i, k) * b(k, j)
8         enddo
9         c(i, j) = val
10     enddo
11  enddo
12  !$acc end kernels
13  call cpu_time(toc)
14  dt = toc - tic
```

GPU compilation

```
$> pgfortran -Mprof=ccff -acc -ta=tesla:cuda8.0,cc35,llvm
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



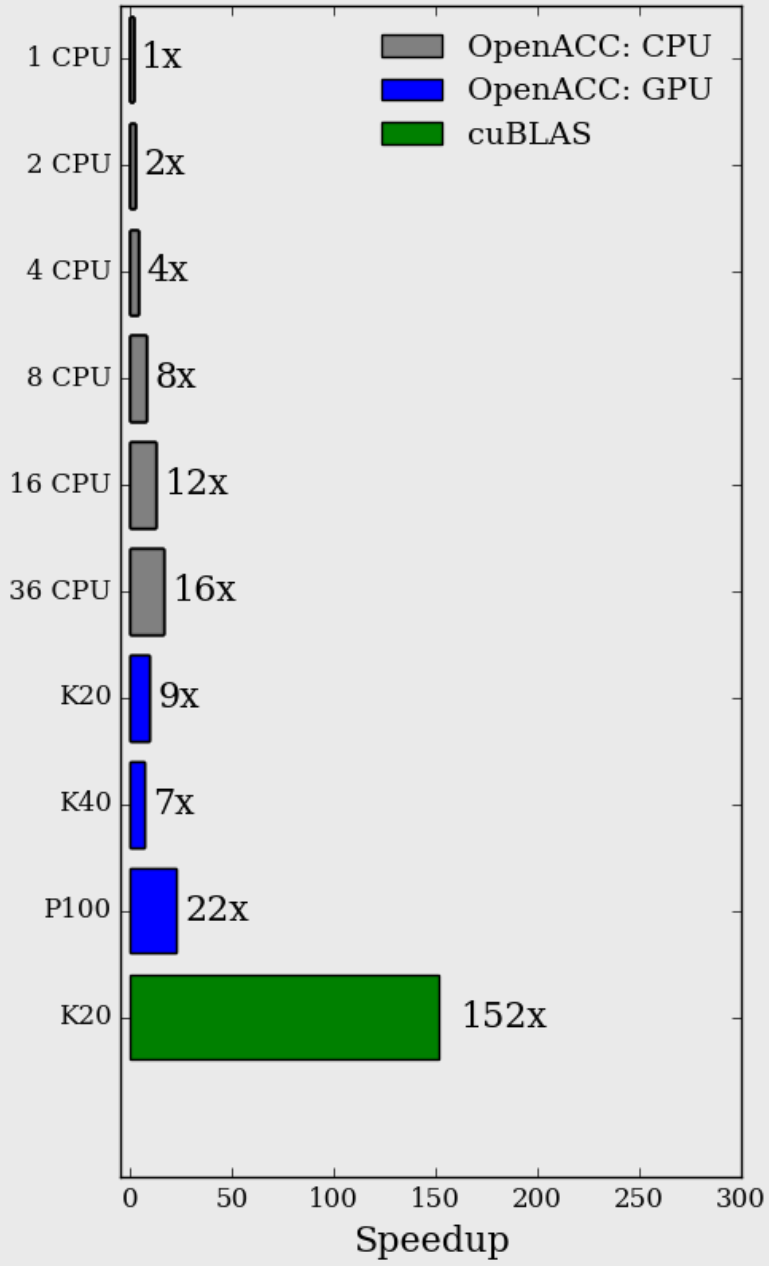
Matrix Multiplication – cuBLAS

```
1
2    ierr = cublasCreate(h)
3    call cpu_time(tic)
4    a_d  = a              ! H2D transfer
5    b_d  = b              ! H2D transfer
6
7    ierr = cublasDgemm_v2(h, CUBLAS_OP_N, &
8                          CUBLAS_OP_N, n, &
9                          p, m, 1.0d0,    &
10                         a_d, n, b_d, m, &
11                         0.0d0, c_d, n)
12   c    = c_d            ! D2H transfer
13   call cpu_time(toc)
14   ierr = cublasDestroy(h)
```

GPU compilation with cuBLAS

```
$> pgfortran -Mcuda -Mcudalib=cublas
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



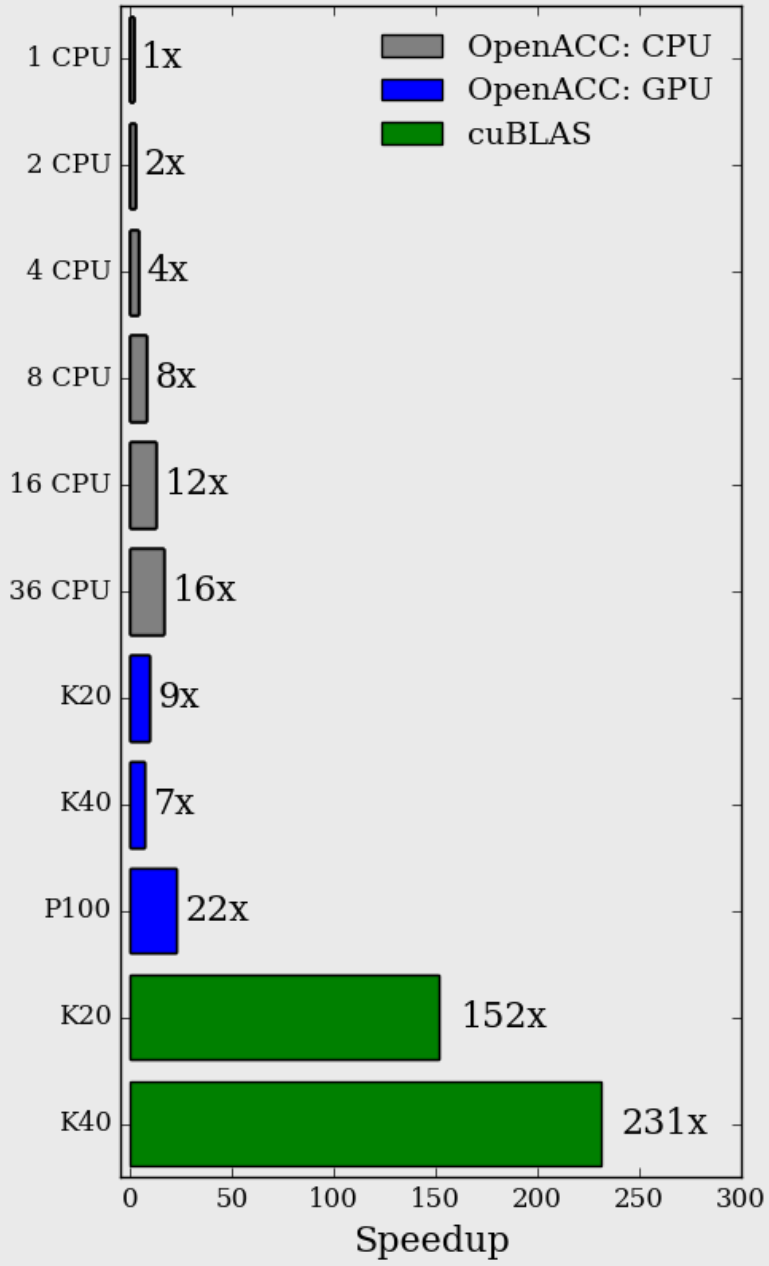
Matrix Multiplication – cuBLAS

```
1
2    ierr = cublasCreate(h)
3    call cpu_time(tic)
4    a_d  = a              ! H2D transfer
5    b_d  = b              ! H2D transfer
6
7    ierr = cublasDgemm_v2(h, CUBLAS_OP_N, &
8                          CUBLAS_OP_N, n, &
9                          p, m, 1.0d0,    &
10                         a_d, n, b_d, m, &
11                         0.0d0, c_d, n)
12   c    = c_d            ! D2H transfer
13   call cpu_time(toc)
14   ierr = cublasDestroy(h)
```

GPU compilation with cuBLAS

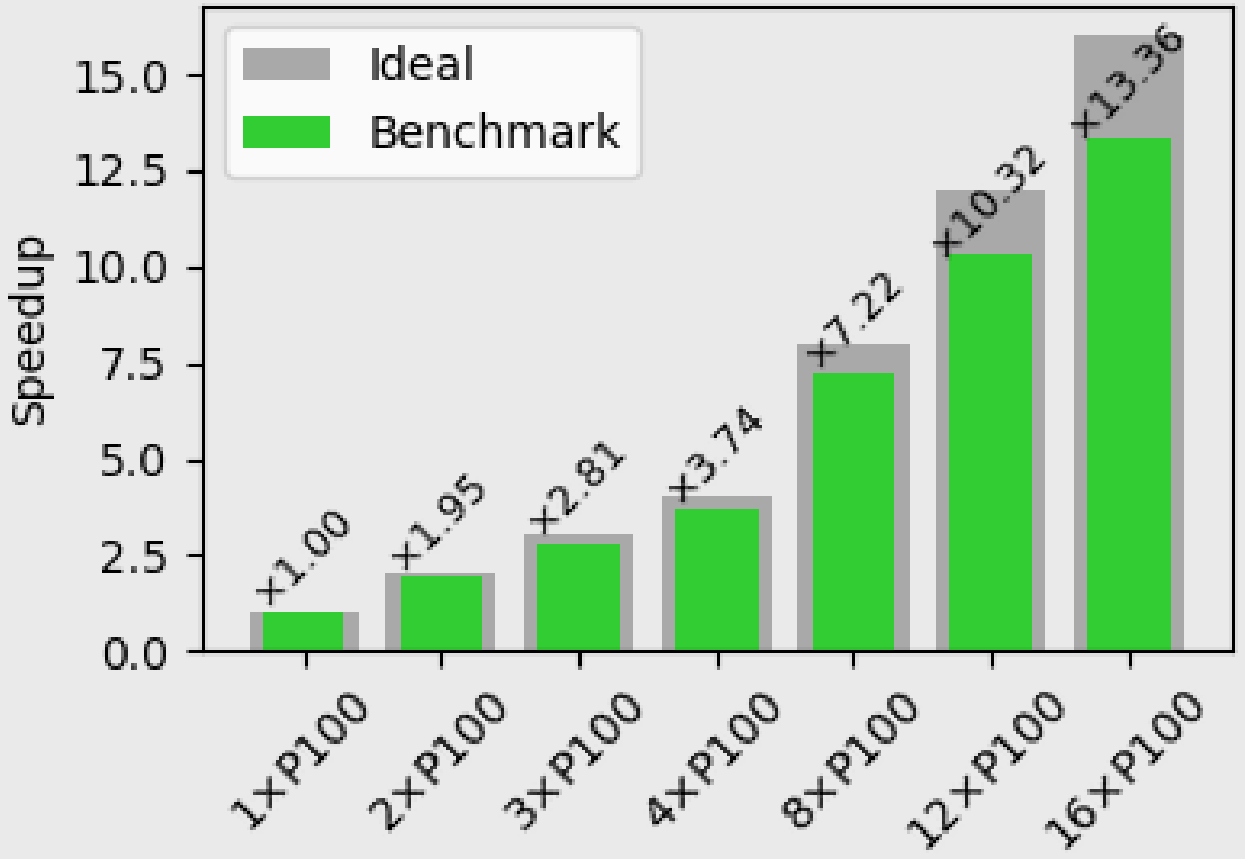
```
$> pgfortran -Mcuda -Mcudalib=cublas
```

Host: Intel Xeon 6140 (SkyLake) with 36 cores



Scaling Machine Learning Applications

- MNIST dataset
- 2-layer Conv. Neural Net
- PyTorch (distributed)
- Baseline: $T_{1\times P100} \approx 30$ min
- Speedup = $T_{N\times P100} / T_{1\times P100}$
- (almost) linear scaling within a node



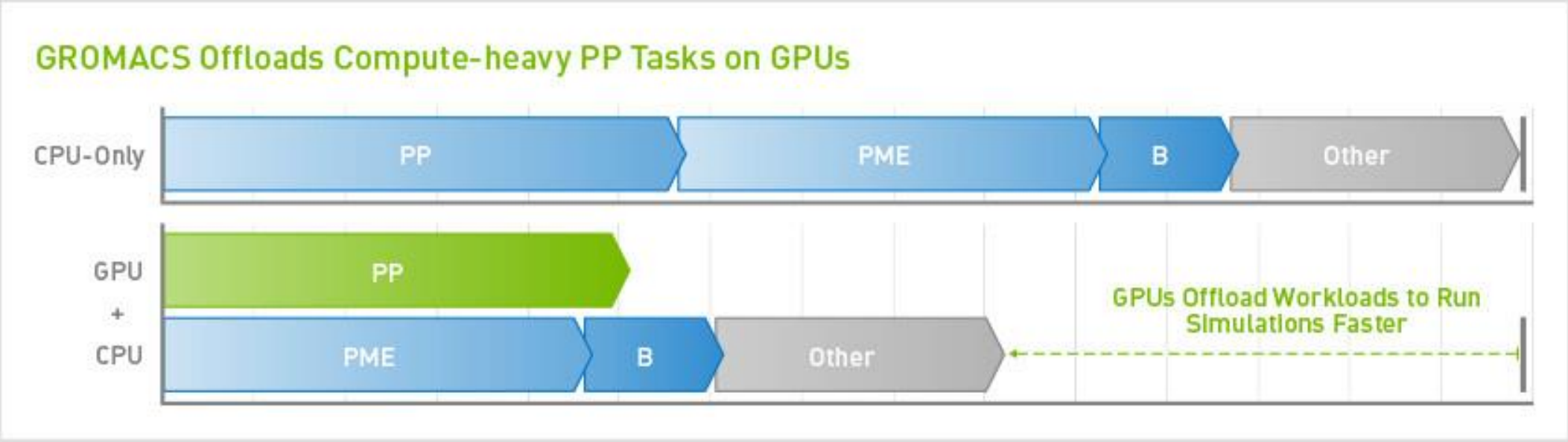
`~/.bashrc`

- For the setup and job template: Contact Us
- `export NCCL_SOCKET_IFNAME=ib1`
- `conda activate PyTorchEnv`

Credit: Martijn Oldenhof (KUL)

Scaling GROMACS

- GROMACS is a molecular dynamics software
- Particle-Particle (PP) tasks: highly parallel
- Overlap PP and PME computations on GPUs
- Data: `water-cut1.0_GMX50_bare/3072`
- Hybrid: MPI, OpenMP and GPU
- 1st: exploit MPI + OpenMP on host
- 2nd: compare with using GPUs



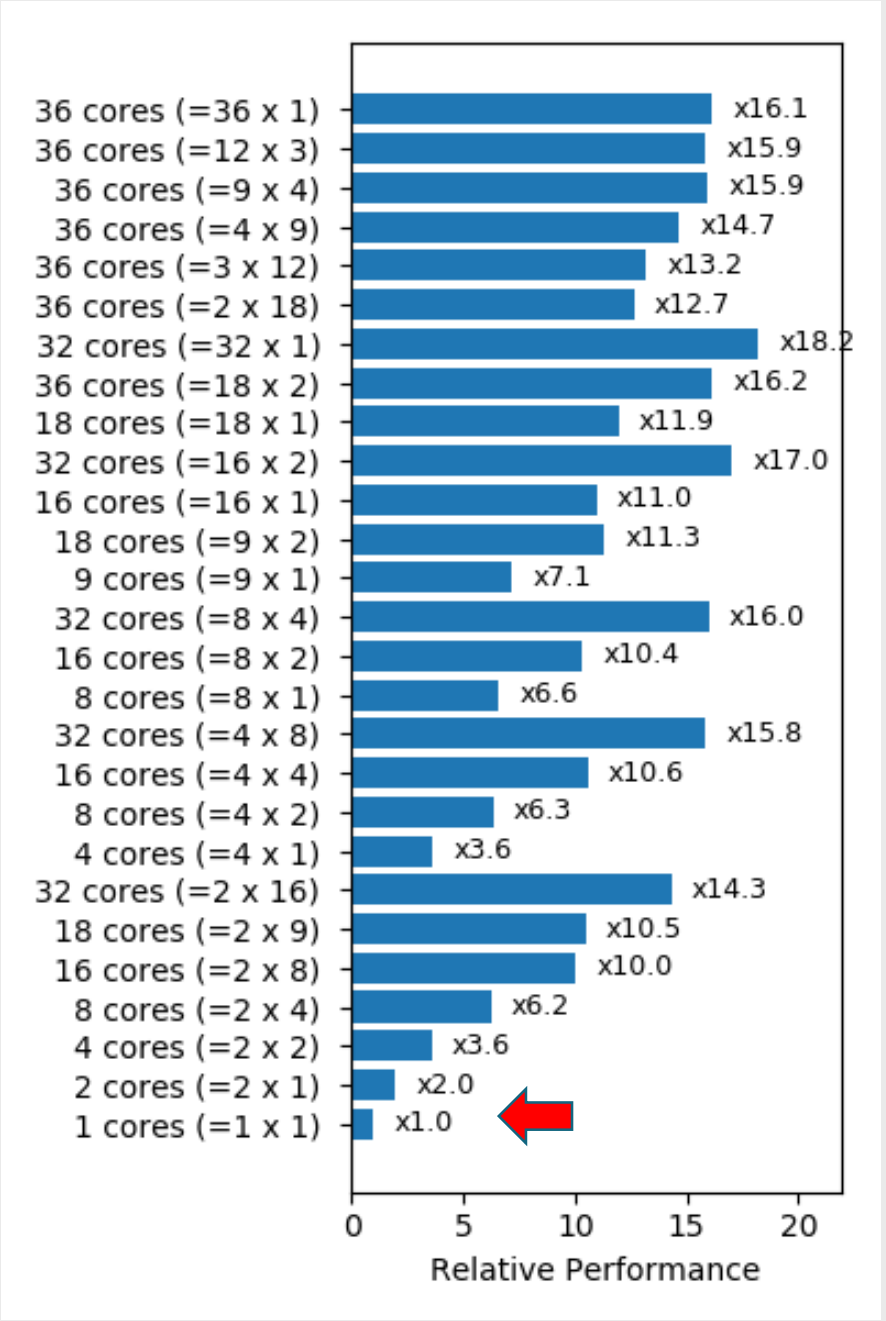
https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/gromacs/

Scaling GROMACS

- 1st: exploit MPI + OpenMP parallelism
- # cores = (# MPI ranks) × (# OpenMP threads)
- E.g. to fill up a SkyLake node (Genius machine):

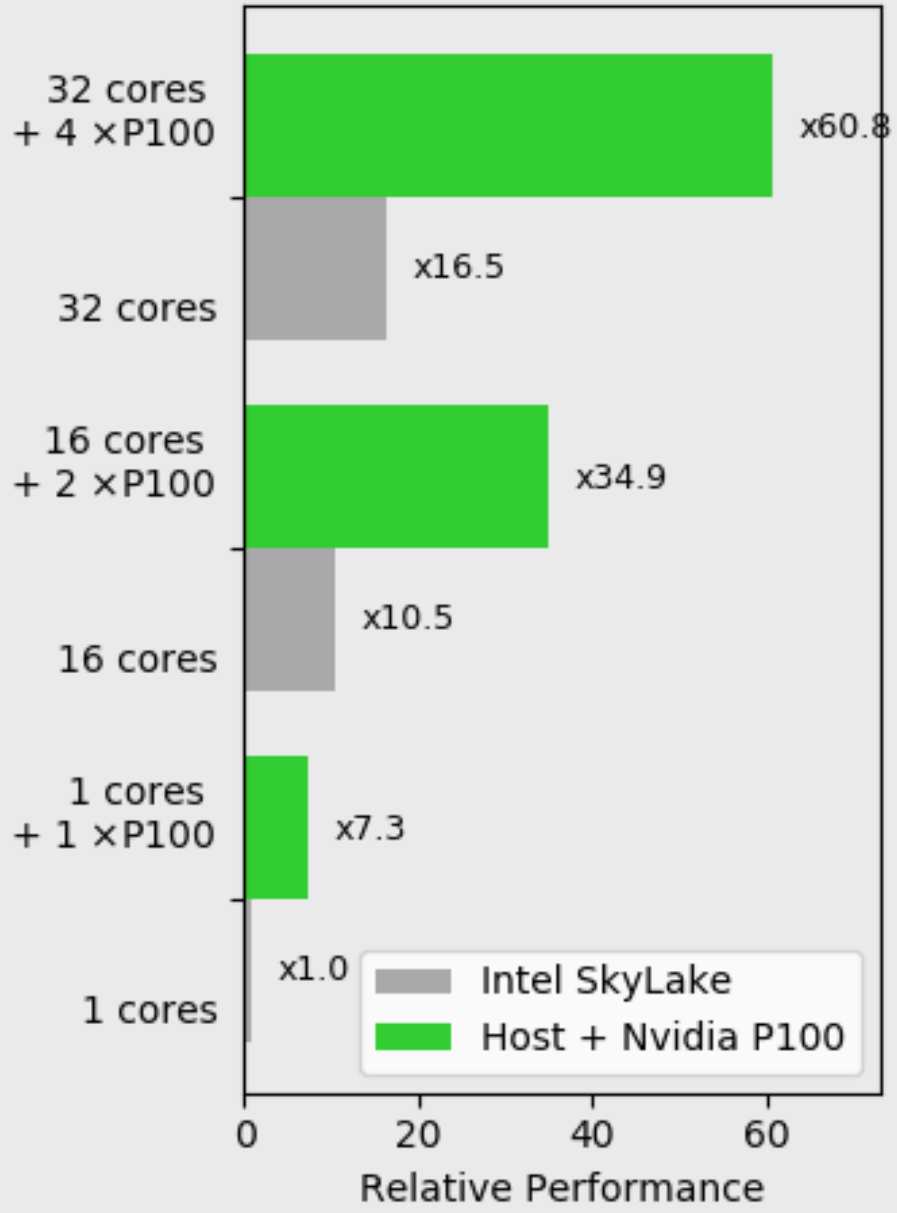
```
$> mpirun -np 4 gmx_mpi mdrun \
          -resethway -noconfout \
          -nsteps 4000 -v \
          -s topol.tpr -g result.log \
          -ntomp 9 -nb cpu
```

- Performance unit: nanoseconds / day
- Performance 1 core (=1 × 1) = 0.160 ns/day
- Relative performance w.r.t. that of 1 core (=1 × 1)



Scaling GROMACS

- 2nd: MPI + OpenMP + GPU
- Fixed problem size and configurations
- Only a single node is used
- Relative performance w.r.t. 1 core ~ **60x**
- Relative performance w.r.t. MPI/OpenMP: **4x to 7x**
- Small molecules: poor GPU performance
- **Large molecules: significant GPU performance**



Learning Resources

- CUDA C/C++ (6)
- CUDA Fortran (1)
- OpenACC (2)
- OpenCL (8)

- OpenACC Online Course
- Nvidia Quick Lab
- Udacity 458 (free): Intro to Parallel Programming CUDA
- CUDA Open Cloud Training Platform

YouTube

- OpenACC Channel
- Nvidia Intro to CUDA
- Few scattered OpenCL tutorials

- CUDA C Reference Guide
- PGI CUDA Fortran Reference Guide
- PGI CUDA Fortran Library Interfaces
- PGI OpenACC Reference Guide

Let's Start-to-GPU …

UAntwerpen

```
$>ssh vscXXXXX@login-Leibniz.uantwerpen.be
```
Access is on demand; contact: hpc@uantwerpen.be

VUB

```
$>ssh vscXXXXX@login.hpc.vub.ac.be
$>qsub -q gpu -l nodes=1:ppn=1:gpus=1:gpgpu -l feature=pascal
```

KU Leuven / UHasselt

Thinking Cluster

```
$>ssh vscXXXXX@login.hpc.kuleuven.be
$>qsub -l partition=gpu -l nodes=1:ppn=1
```

Genius Cluster

```
$>ssh vscXXXXX@login[1-4].hpc.kuleuven.be
```
You can request fraction of a node or full node(s):
```
$>qsub -l nodes=1:ppn=9:gpus=1,partition=gpu
$>qsub -l nodes=1:ppn=36:gpus=4,partition=gpu
```

All VSC Users

Evaluation of GPUs for future Tier-1: we invest only if users need. Thus, your inputs are very welcome. Contact Engelbert Tijskens

So, let's start to GPU

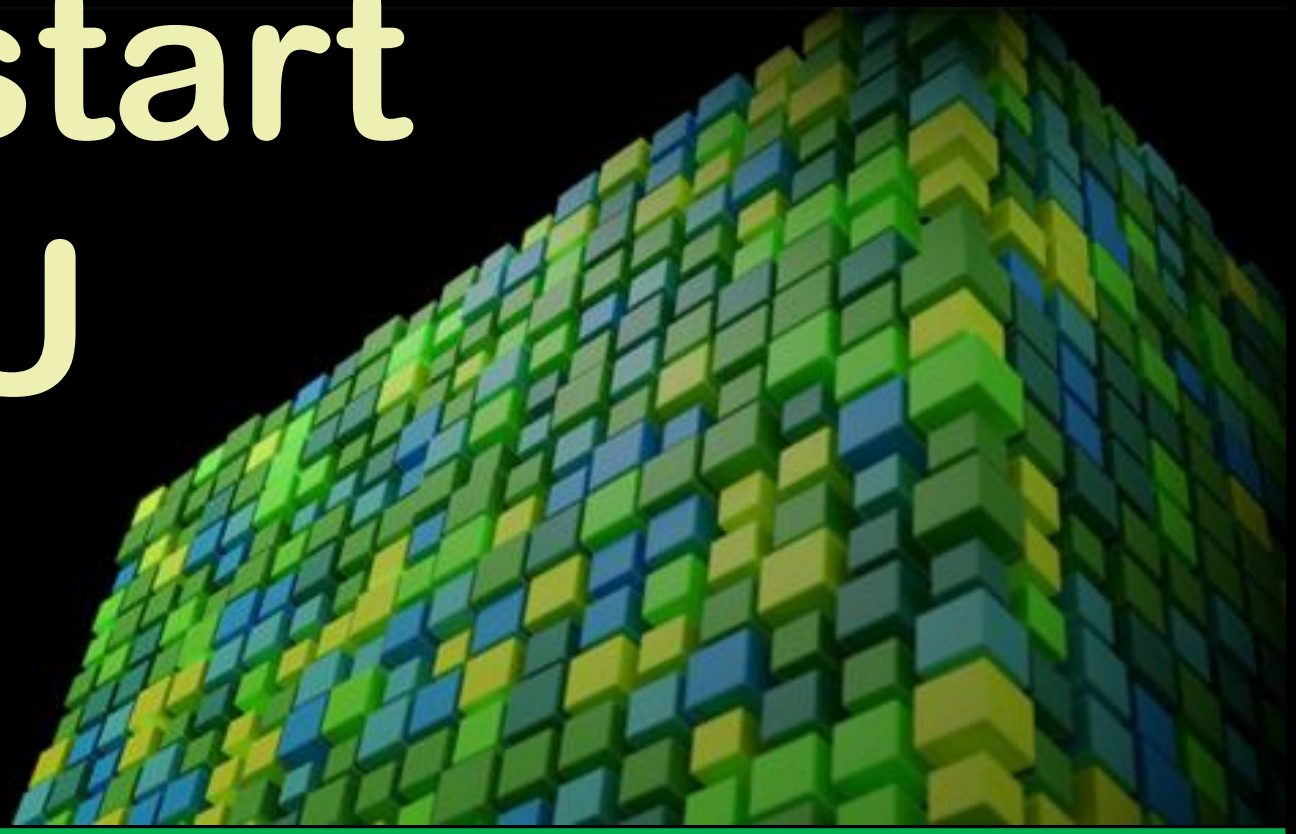


Image courtesy of Nvidia.com

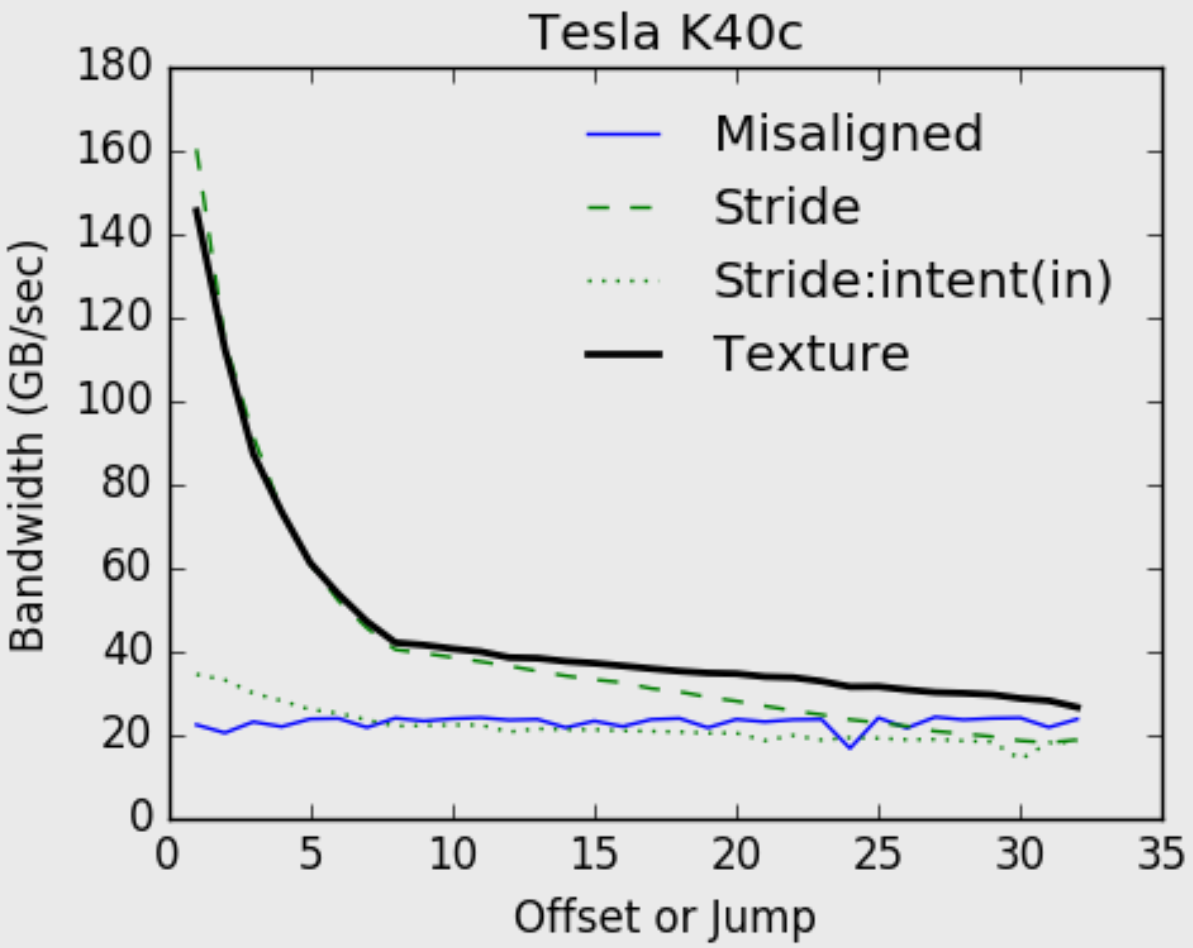
Strided and Misaligned Memory Access

Misaligned

```
y[k] = x[k - offset] + x[k + offset]
       for k = offset, …, N - offset
```

Strided

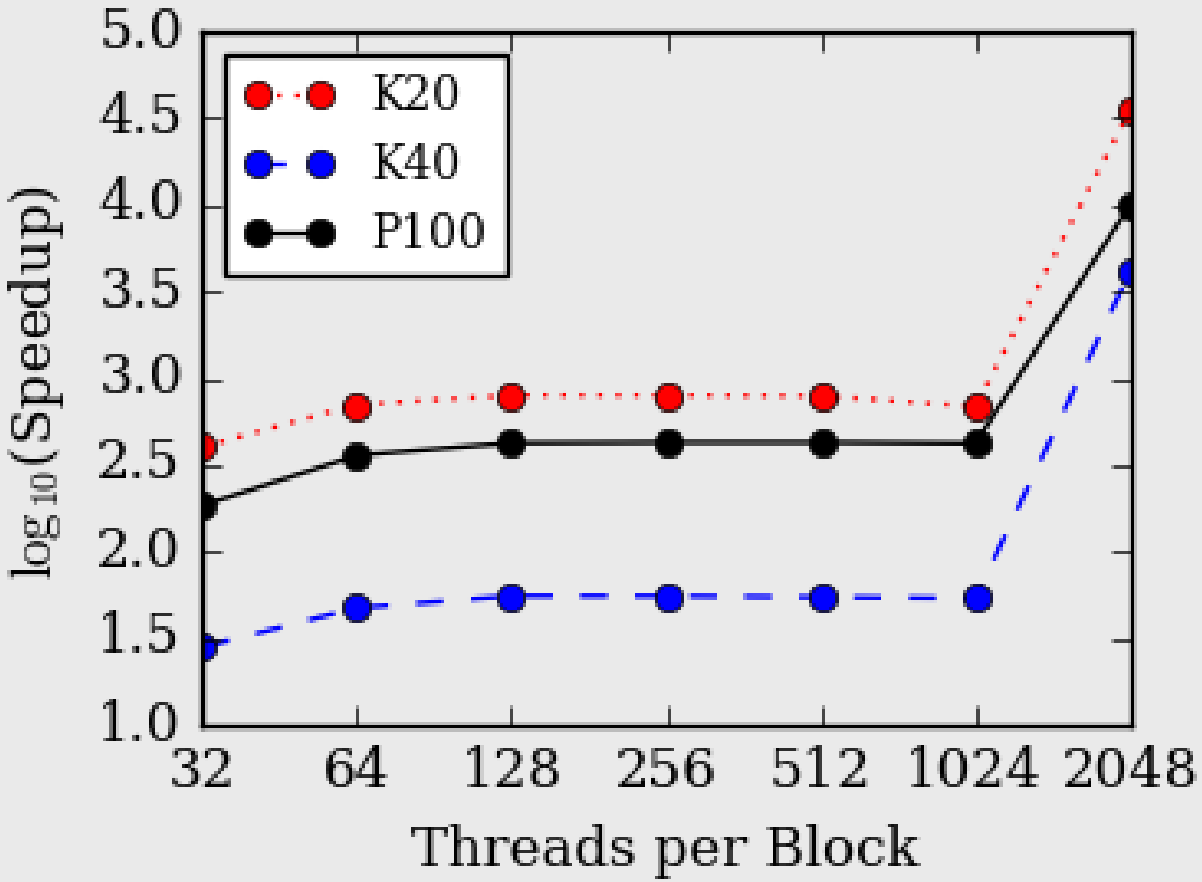
```
y[k] = x[ c * k] + 1.0
       where c = constant
```



Thread-Level Parallelism

- How to distribute jobs among available cores?
- `kernel<<<Nblocks, TpB>>>(…)`
- How does BW for kernel launch depends on Threads per Block (`TpB`)?

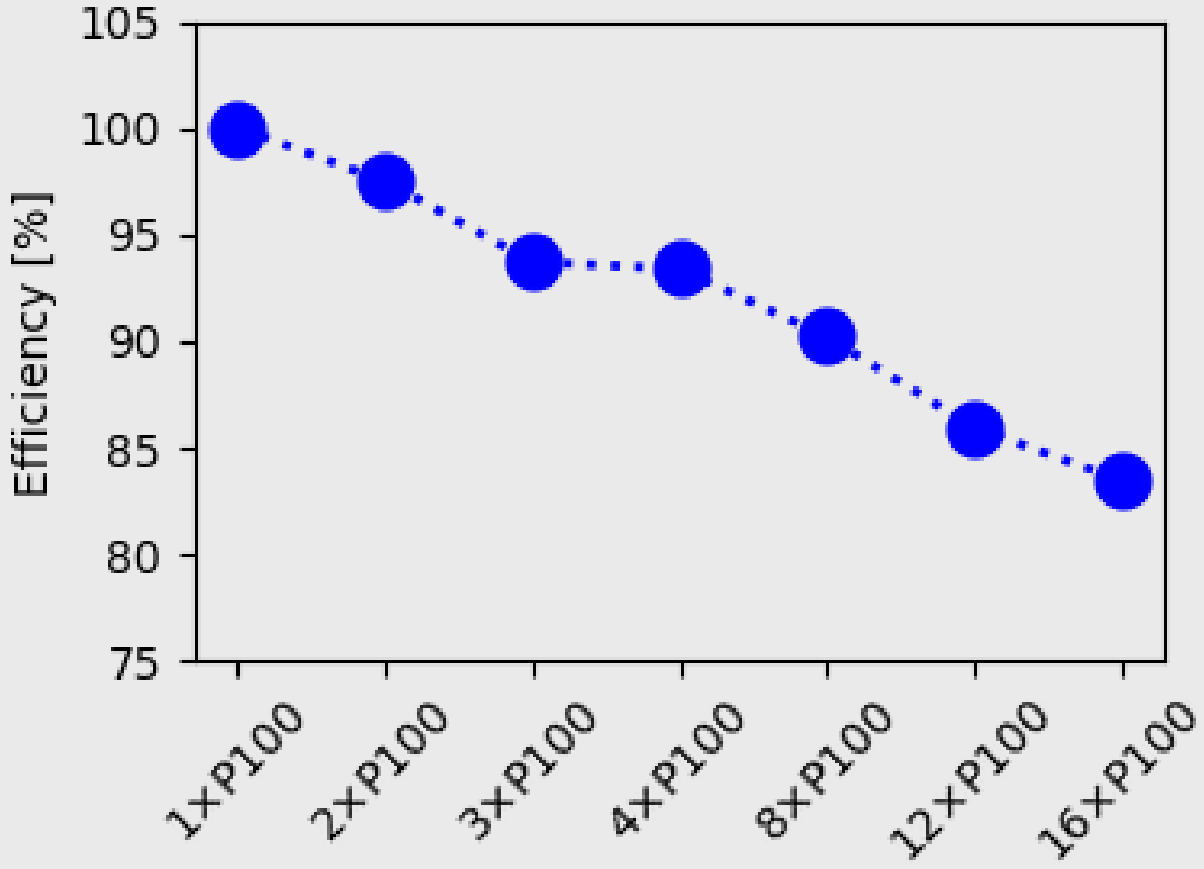
- The same arbitrary function
  $y_k = b_k^2 \sin(a_k) + a_k \exp(\cos(b_k))$
- H2D & D2H data transfer are **excluded** from BW measurement
- `TpB=2048` gives >70x speed-up (only for kernel launch)



160 MB single-precision array. Using texture memory.

Scaling Machine Learning Applications

- MNIST dataset
- 2-layer Conv. Neural Net
- PyTorch (distributed)
- Baseline: $T_{1\times P100} \approx 30$ min
- Efficiency = $T_{1\times P100} / (N \times T_{N\times P100})$
- Over 80% efficiency with 4 nodes



~/.bashrc

- For the setup and job template: Contact Us
- `export NCCL_SOCKET_IFNAME=ib1`
- `conda activate PyTorchEnv`