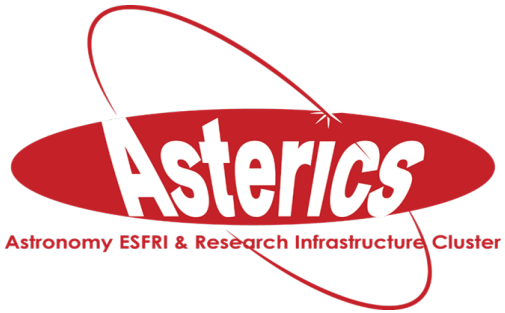
1st ASTERICS-OBELICS International School

6-9 June 2017, Annecy, France.



H2020-Astronomy ESFRI and Research Infrastructure Cluster
(Grant Agreement number: 653477).

GPU Programming

Valeriu Codreanu

SURFsara

Outline

Today's lecture

- Introduction to the GPU ecosystem
- The GPU HW architecture
- GPU programming
- GPUs & High-performance Libraries
- GPU Debugging & Profiling
- GPUs & Python

Tomorrow's lecture

- Slightly more advanced topics
- Recent GPU features (Unified memory, Cooperative threads, Tensor cores)
- Multi-GPU/GPUDirect RDMA

SURFsara

History:
1971: Founded by the VU, UvA, and CWI
2013: SARA (Stichting Academisch Rekencentrum A'dam) becomes part of SURF

Cartesius (Bull supercomputer):
40.960 Ivy Bridge / Haswell cores: 1327 TFLOPS
56GBit/s Infiniband

64 nodes with 2 K40m GPUs each: 210 TFLOPS

Broadwell & KNL extension (Nov 2016)
177 BDW and 18 KNL nodes: 284TFLOPS

7.7 PB Lustre parallel file-system

Top500 position
#45 2014/11
#97 2016/11

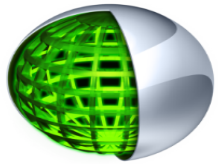
Introduction to the GPU ecosystem

CUDA Parallel Computing Platform

Programming Approaches

- **Libraries** — "Drop-in" Acceleration
- **OpenACC Directives** — Easily Accelerate Apps
- **Programming Languages** — Maximum Flexibility

Development Environment



Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

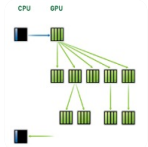
Open Compiler Tool Chain

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

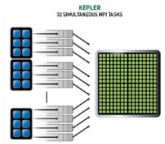
Hardware Capabilities

SMX

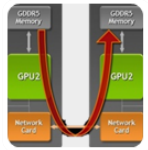
Dynamic Parallelism



HyperQ

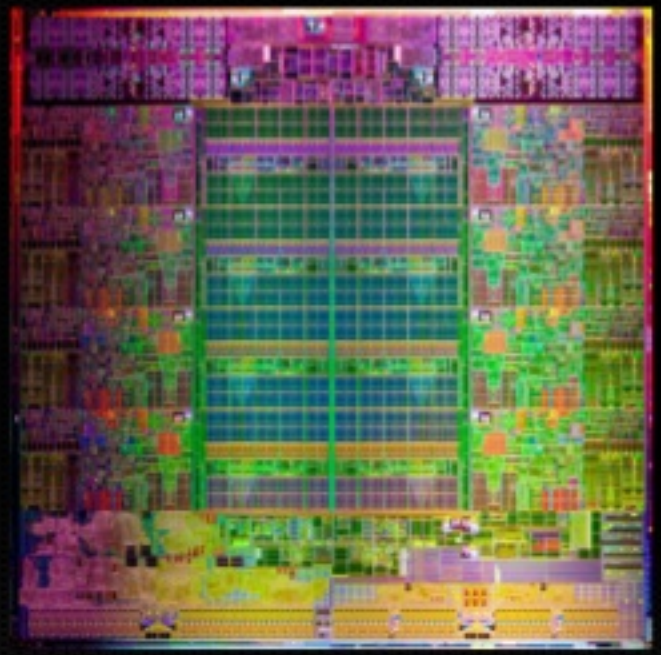


GPUDirect



CPU vs. GPU

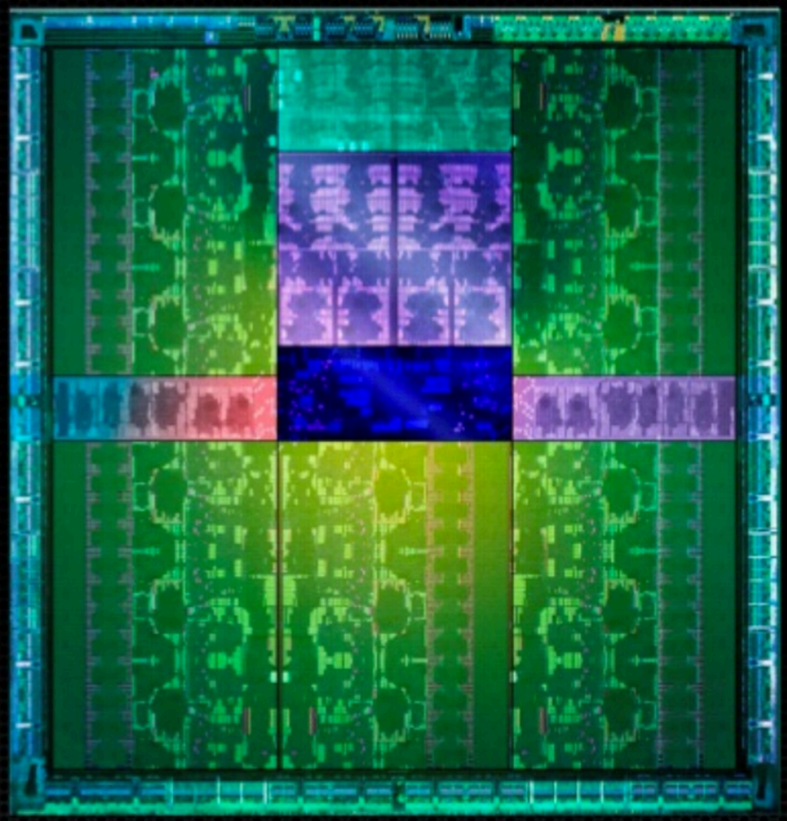
Example CPU: Xeon E5-2687W



- 2.27 B transistors
- 8 cores, 16 threads @ 3.1 GHz
  - 0.35 SP TFLOPS
  - 0.17 DP TFLOPS
- 256 GB DDR3 @1600 MHz
  - 51.2 GB/s
- 150 W
- 20 MB L3 cache
- Single thread Perf
  - branch prediction
  - out of order execution

NVIDIA

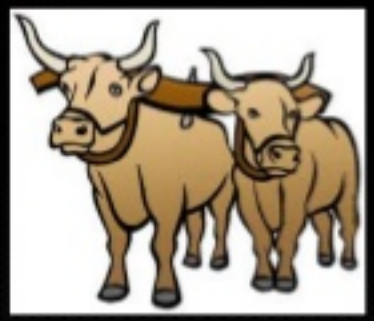
Example GPU: Tesla K40



- 7.1 B transistors
- 2880 cores, 30720 threads @ 745 MHz
  - 4.29 SP TFLOPS
  - 1.43 DP TFLOPS
- 12 GB GDDR5 @ 3GHz
  - 288 GB/s memory BW
- 235 W
- PCIE Gen3 x16
  - 12 GB/s

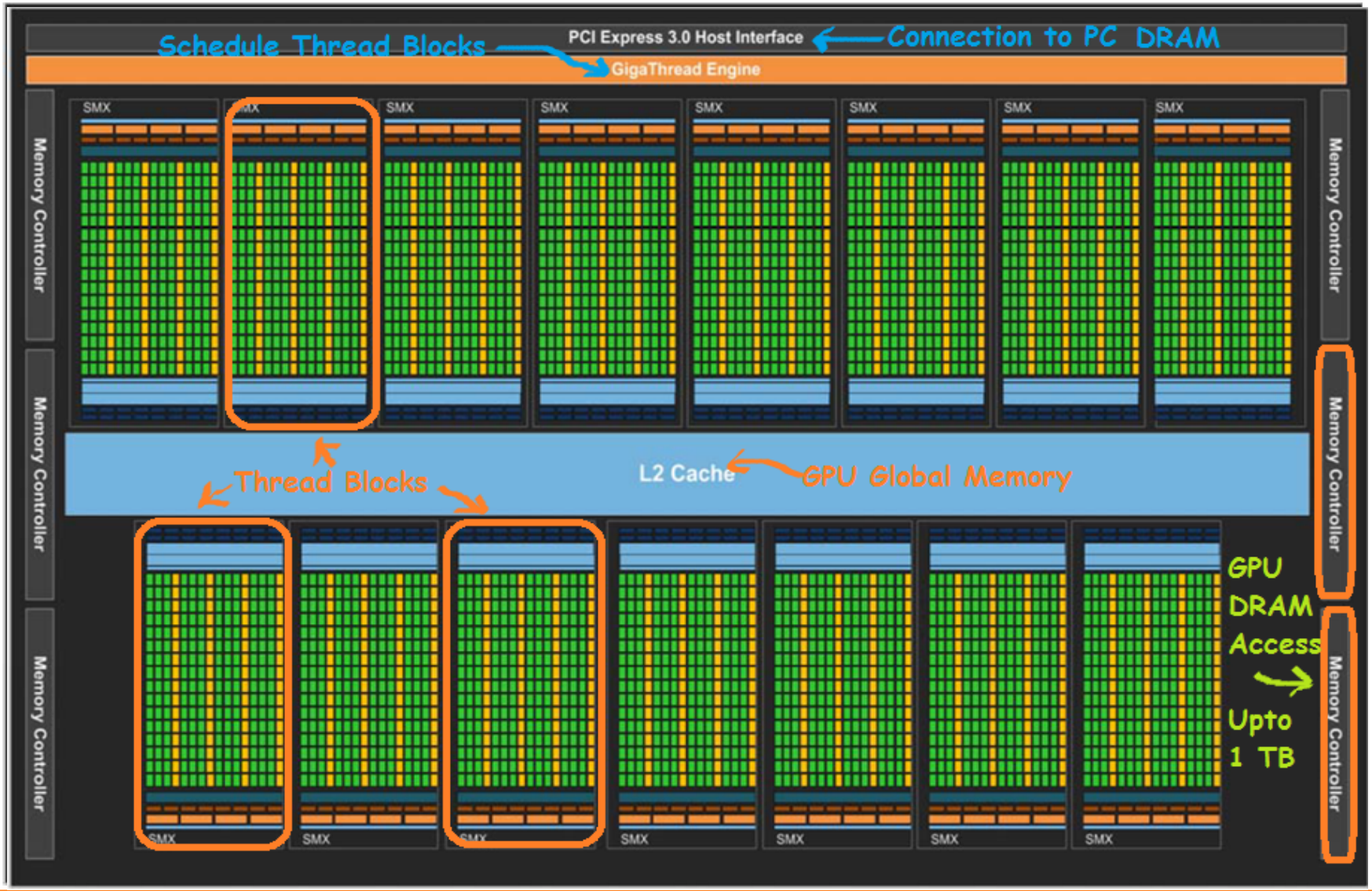
"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"

—Seymour Cray

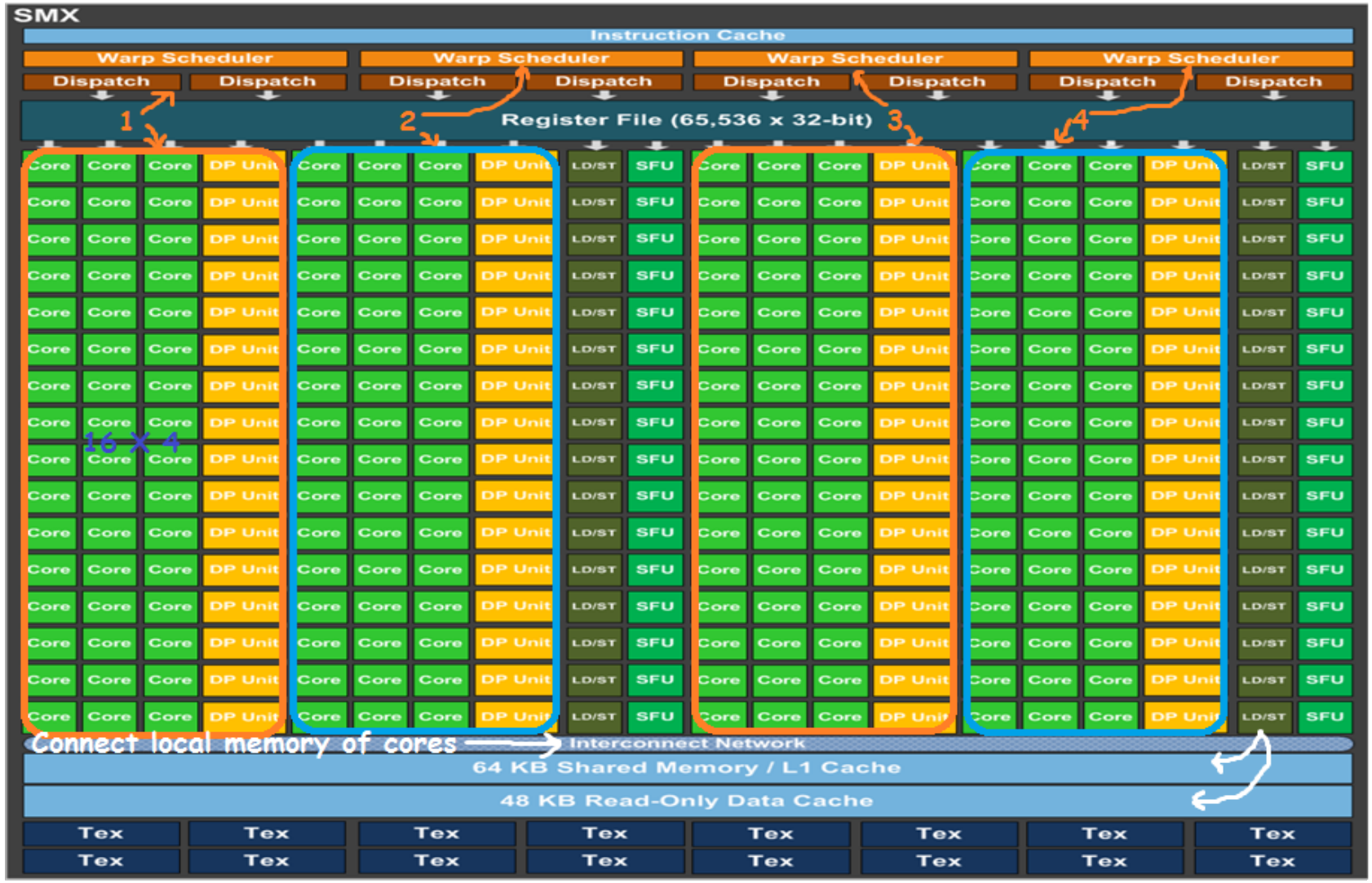


The GPU hardware architecture

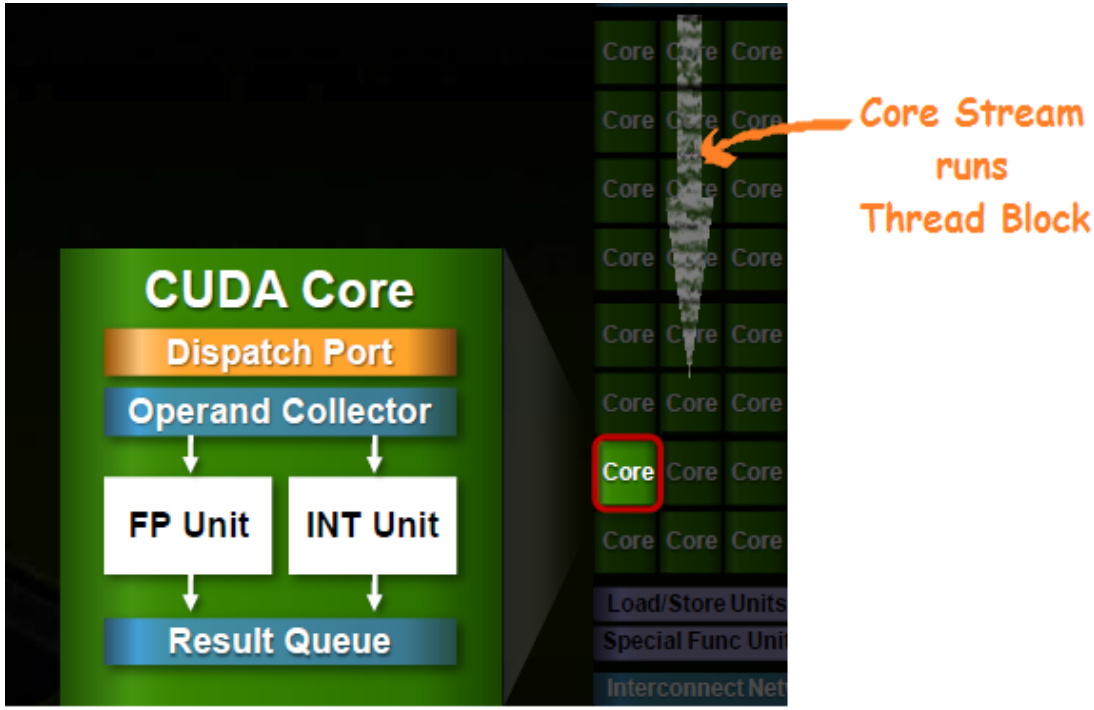
The Kepler GPU Architecture



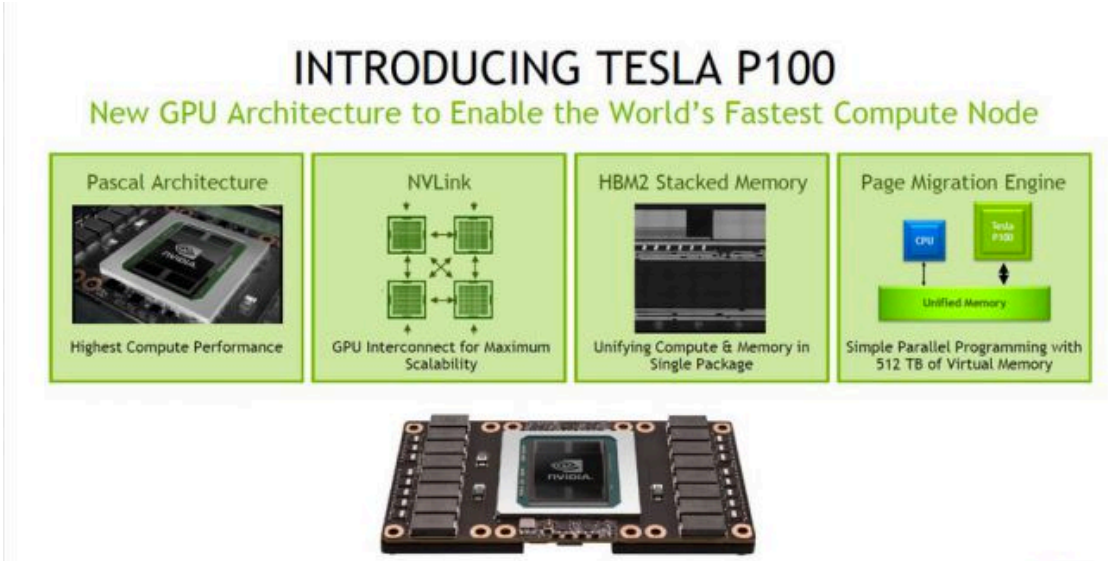
Kepler Streaming MultiProcessor architecture

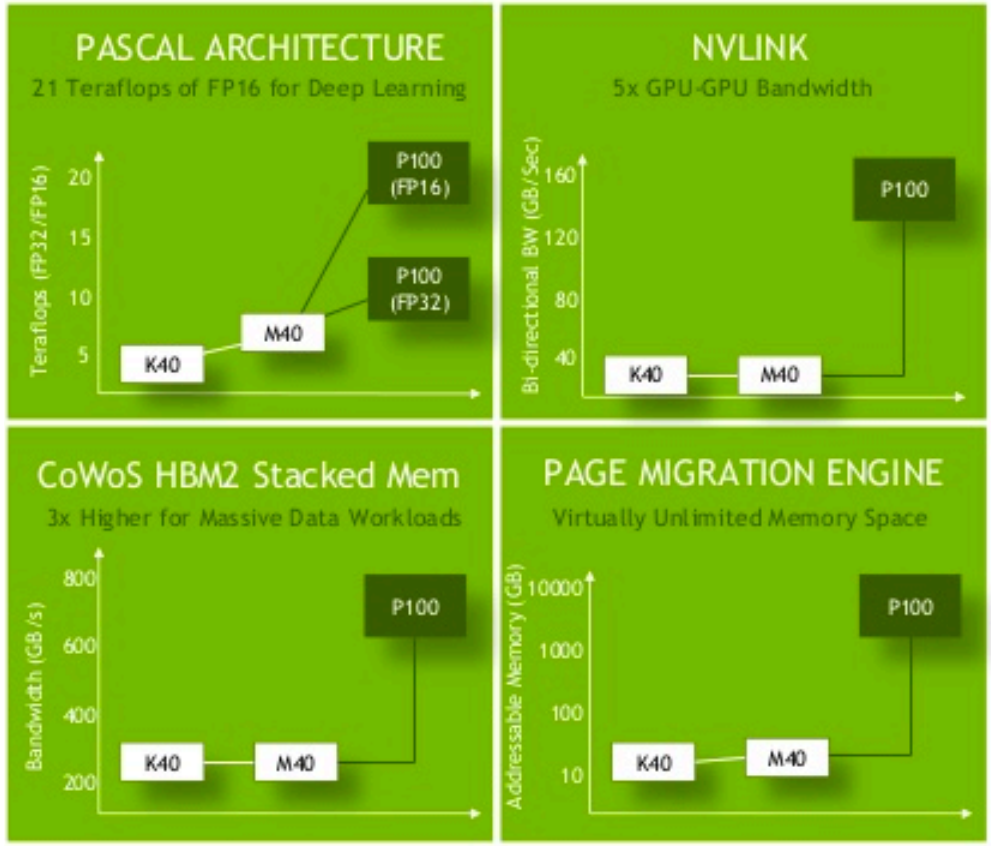


Kepler CUDA core

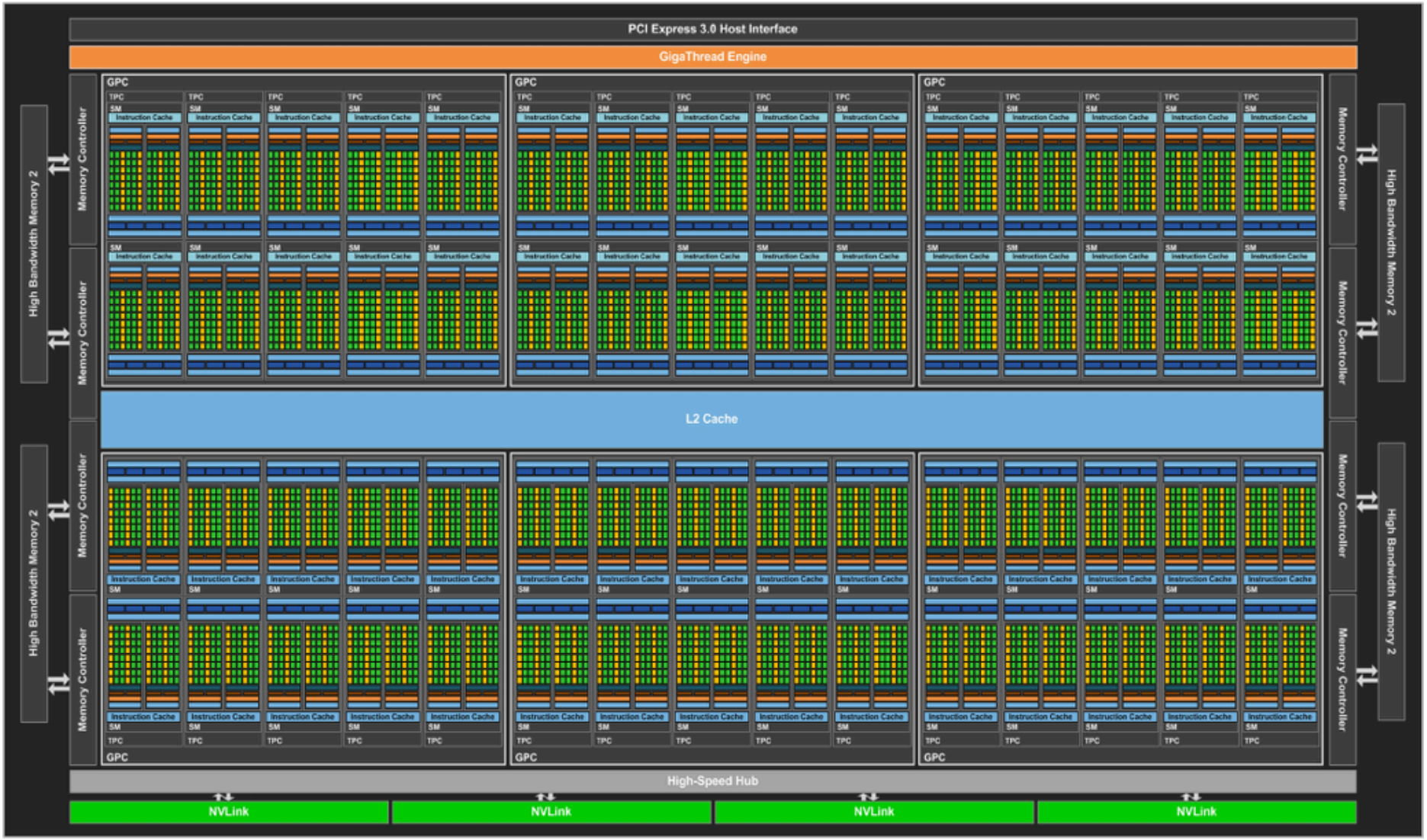


NVIDIA Pascal architecture



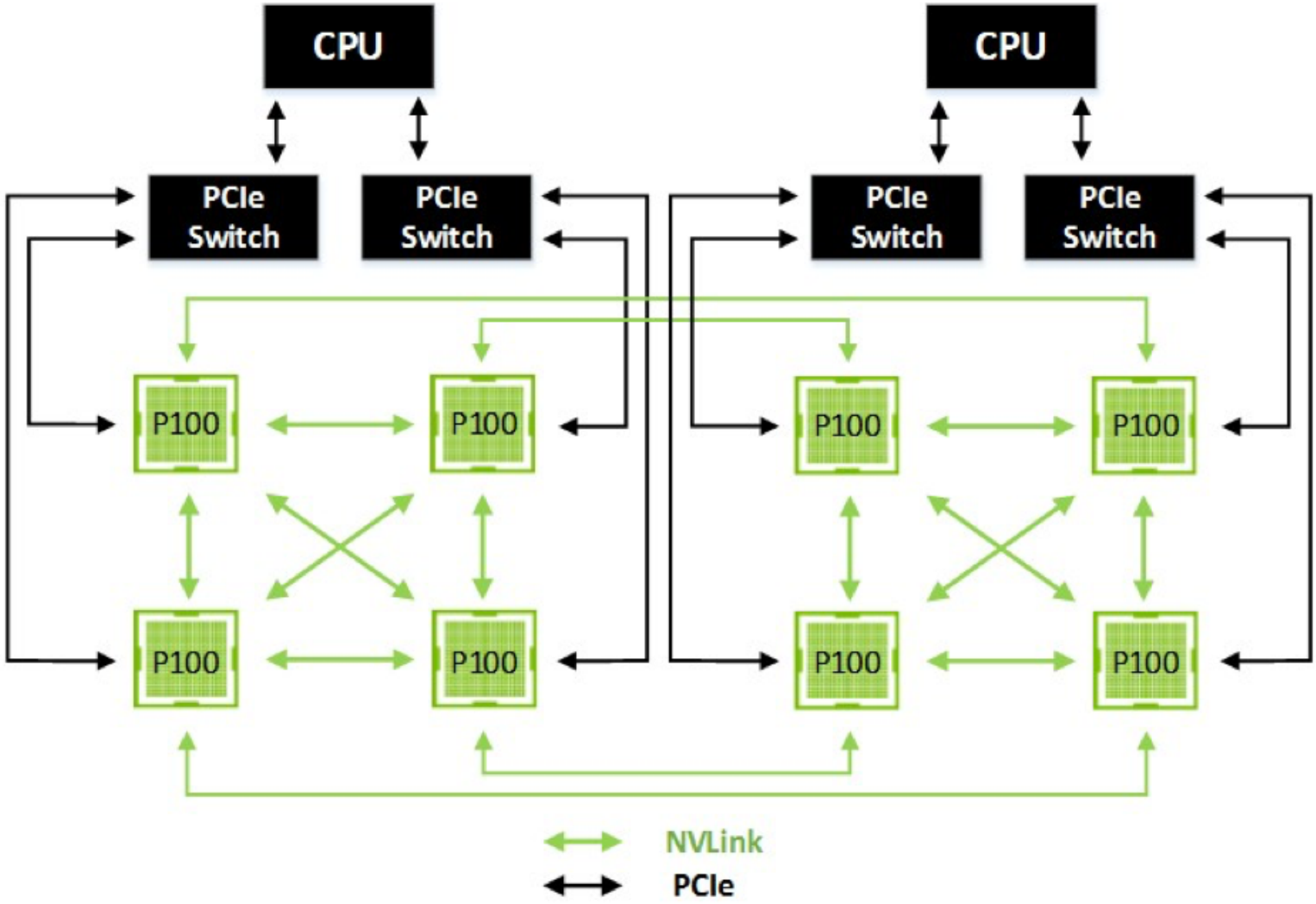


NVIDIA Pascal P100 architecture



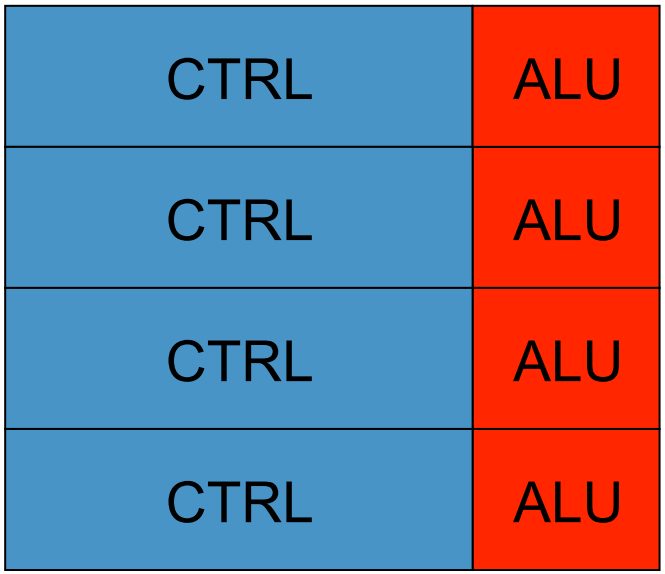
NVIDIA Pascal P100 SM architecture

NVIDIA Pascal P100 system architecture

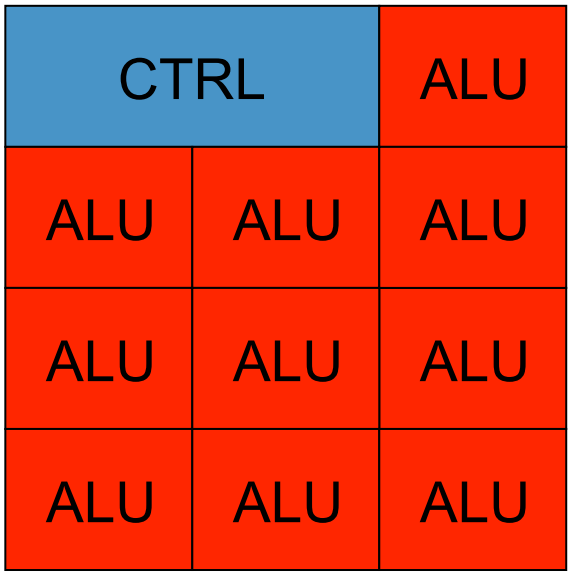


Architecture (MIMD vs SIMD)

MIMD(CPU-Like)



SIMD (GPU-Like)



Flexibility

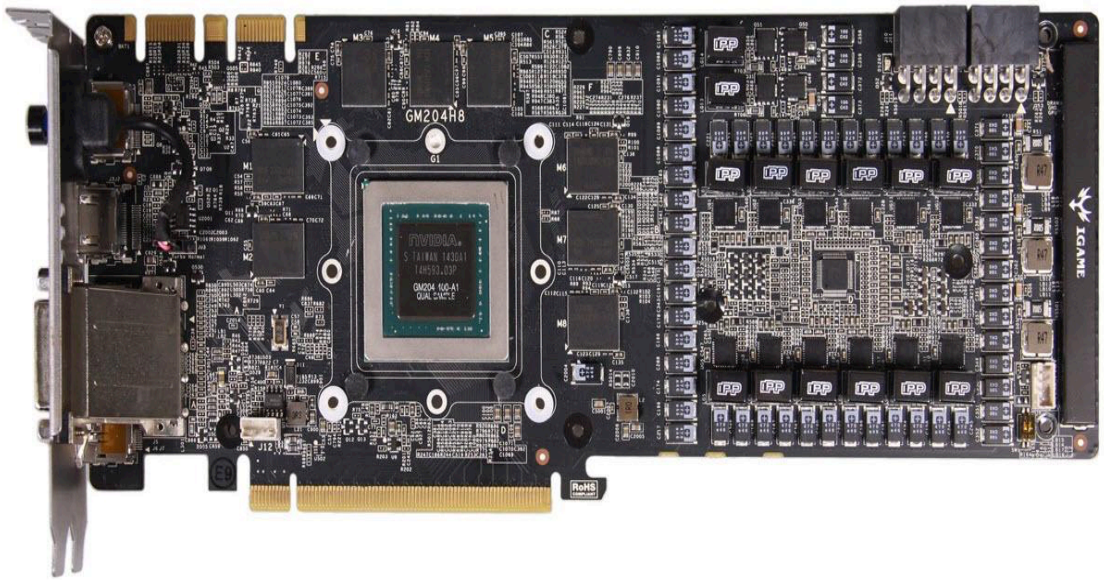
Ease of Use

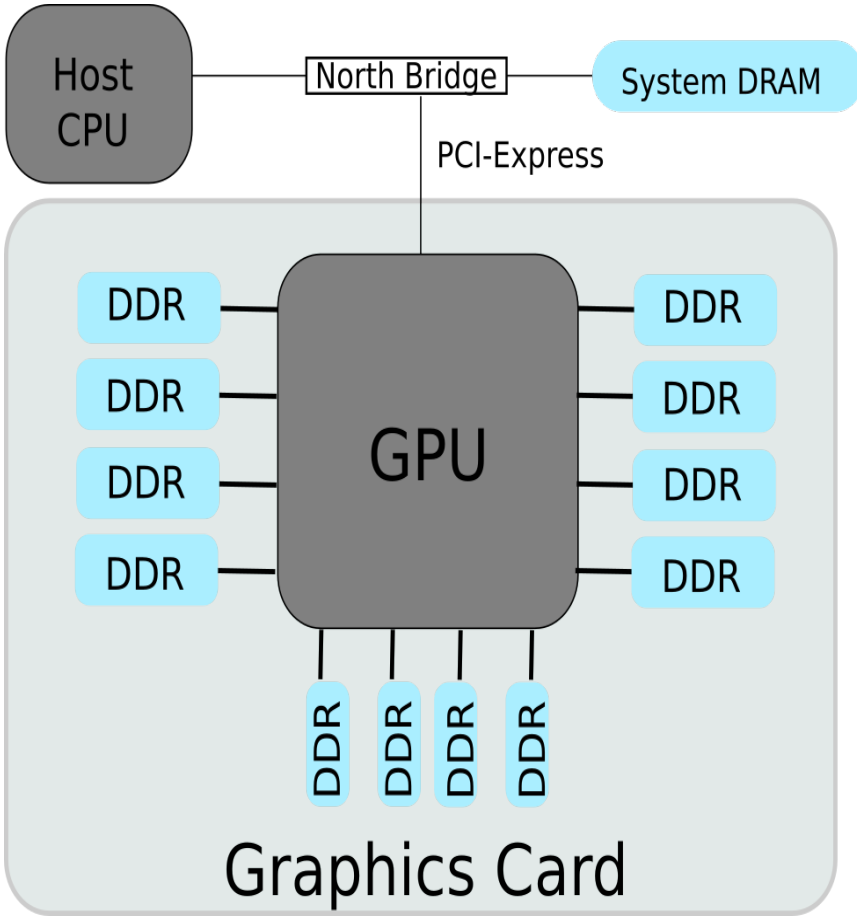
Horsepower

GPUs are everywhere

GPUs in Gaming Cards

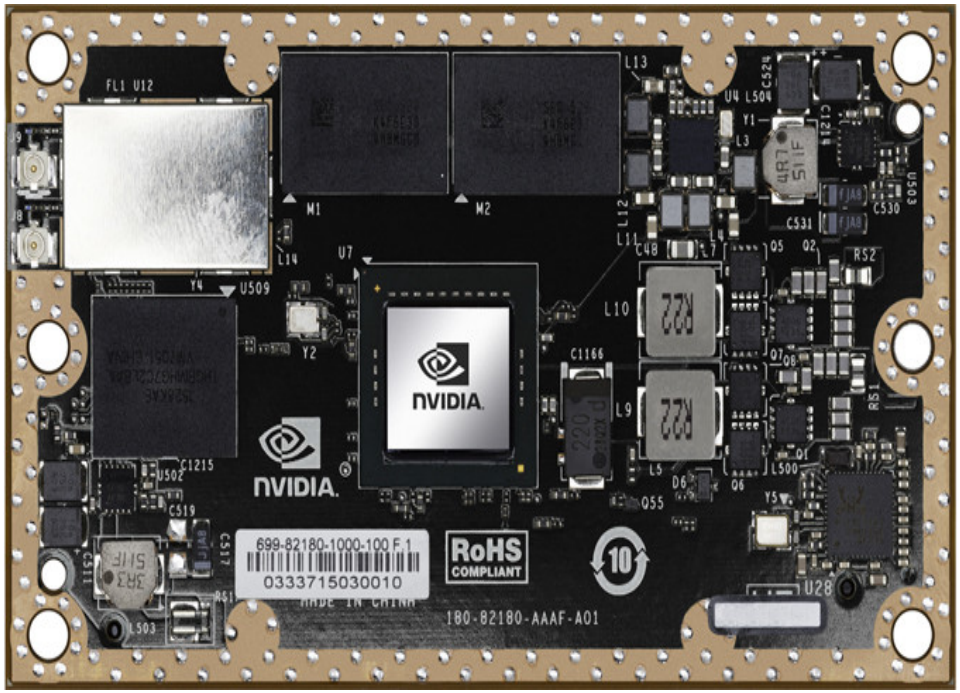
Image: Nvidia GTX 980

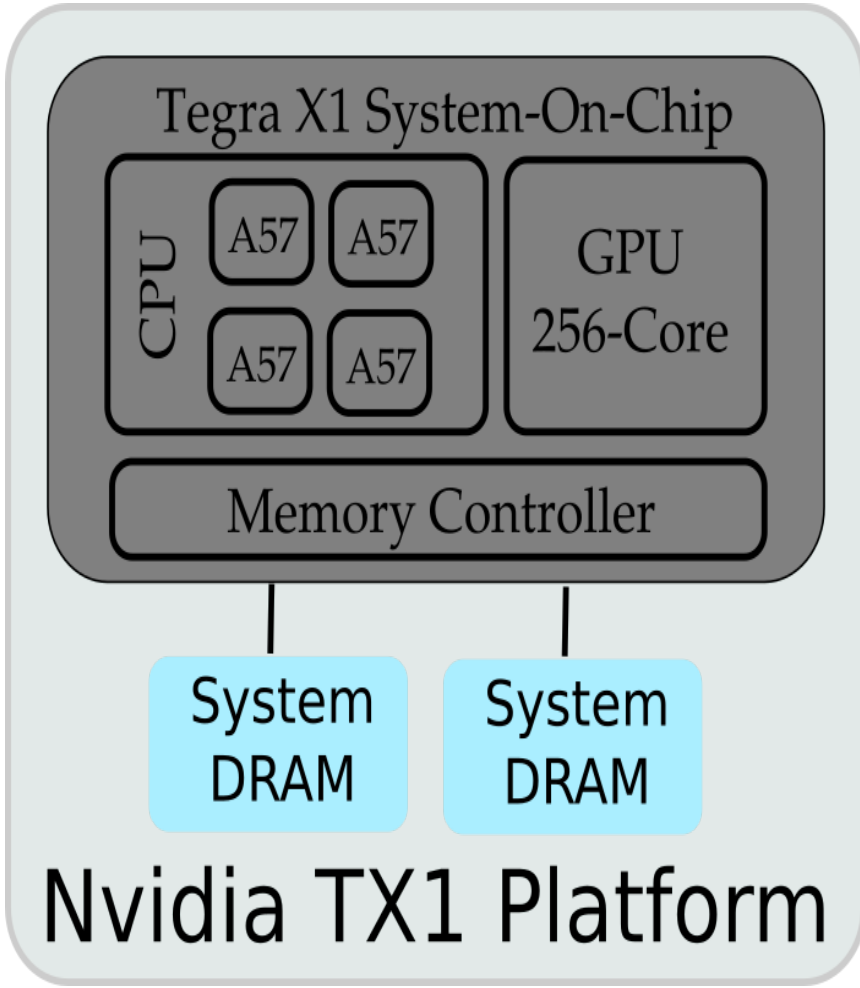




GPU in Mobile Processors

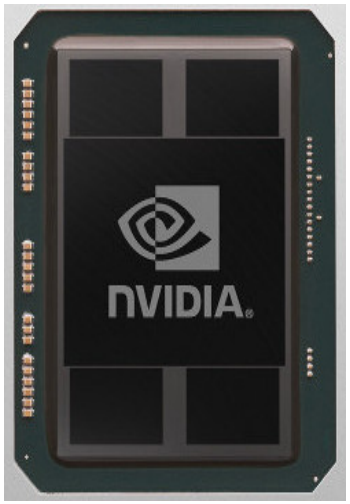
Image: Nvidia Jetson TX1 (Tegra X1 SOC)

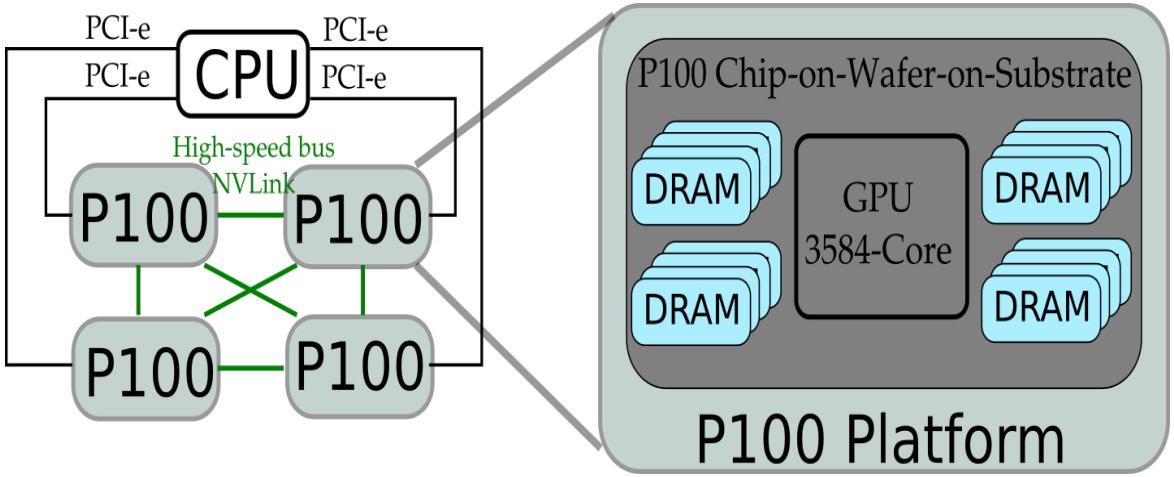




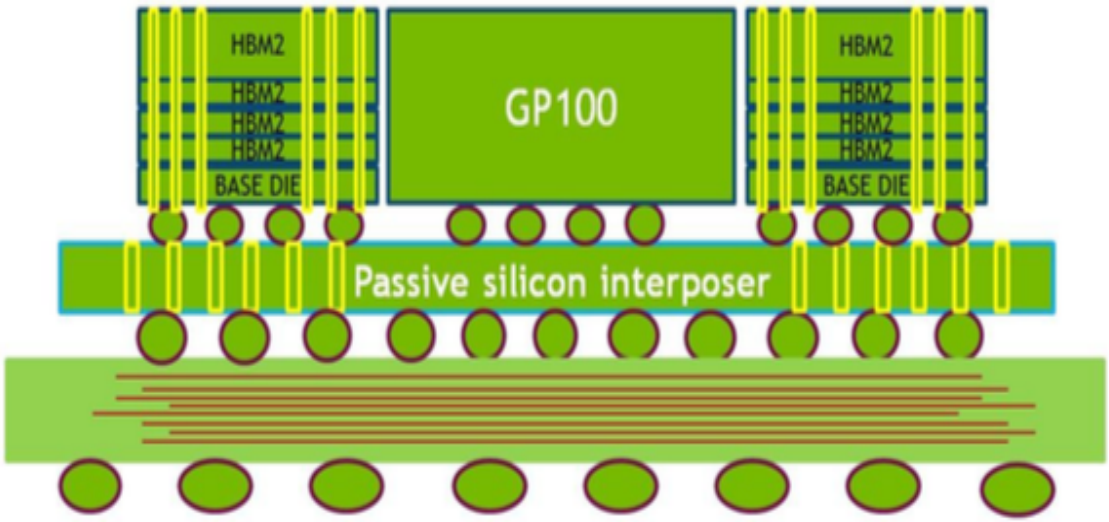
GPU in High-Performance Computers

Image: Nvidia P100
(Pascal Architecture)



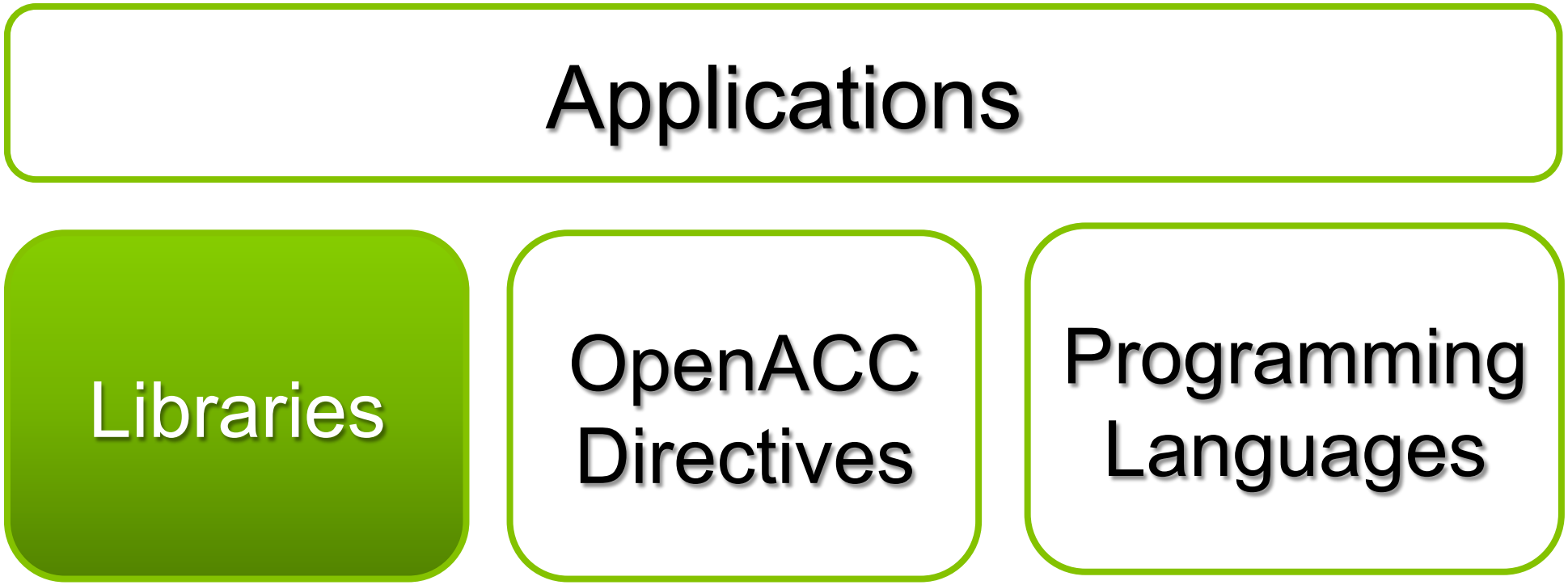


Chip-on-Wafer-on-Substrate



GPU programming

3 Ways to Accelerate Applications

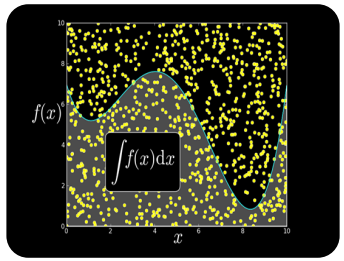


Libraries: Easy, High-Quality Acceleration

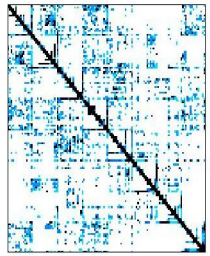
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes (replacing MKL/IPP/FFTW/…)
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** NVIDIA libraries are tuned by experts

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

CULA tools

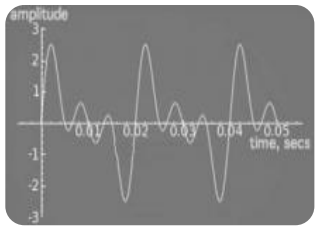
GPU Accelerated Linear Algebra

MAGMA

ICL UT

Matrix Algebra on GPU and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

open source initiative

C++ STL Features for CUDA

open source initiative

3 Steps to CUDA-accelerated application

- **Step 1:** Substitute library calls with equivalent CUDA library calls

  `saxpy ( … )` ➢ `cublasSaxpy ( … )`

- **Step 2:** Manage data locality

  - with CUDA: `cudaMalloc(), cudaMemcpy(), etc.`
  - with CUBLAS: `cublasAlloc(), cublasSetVector(), etc.`

- **Step 3:** Rebuild and link your CUDA Library-accelerated application

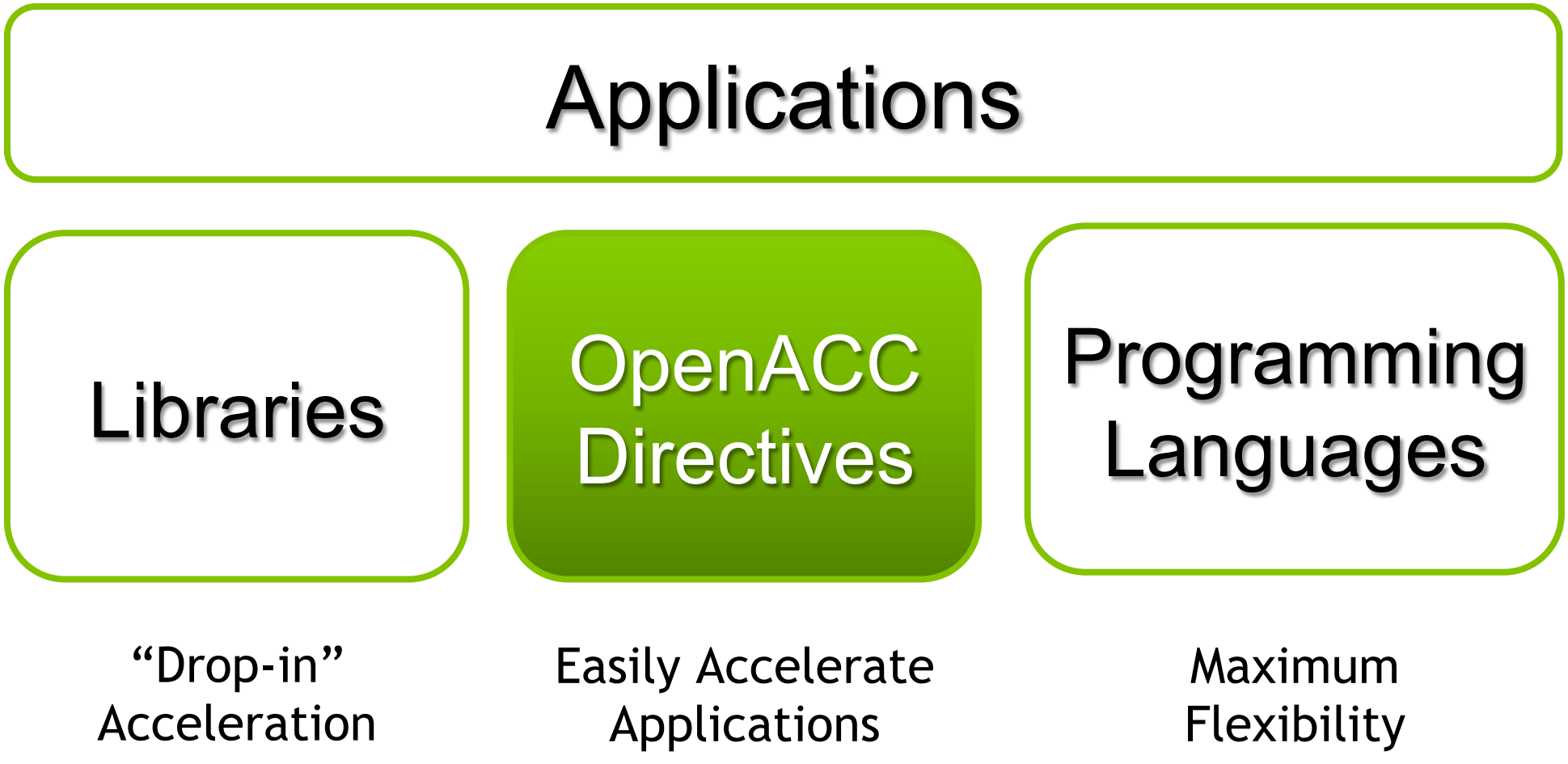
```
nvcc myobj.o –l cublas
```

Explore the CUDA (Libraries) Ecosystem

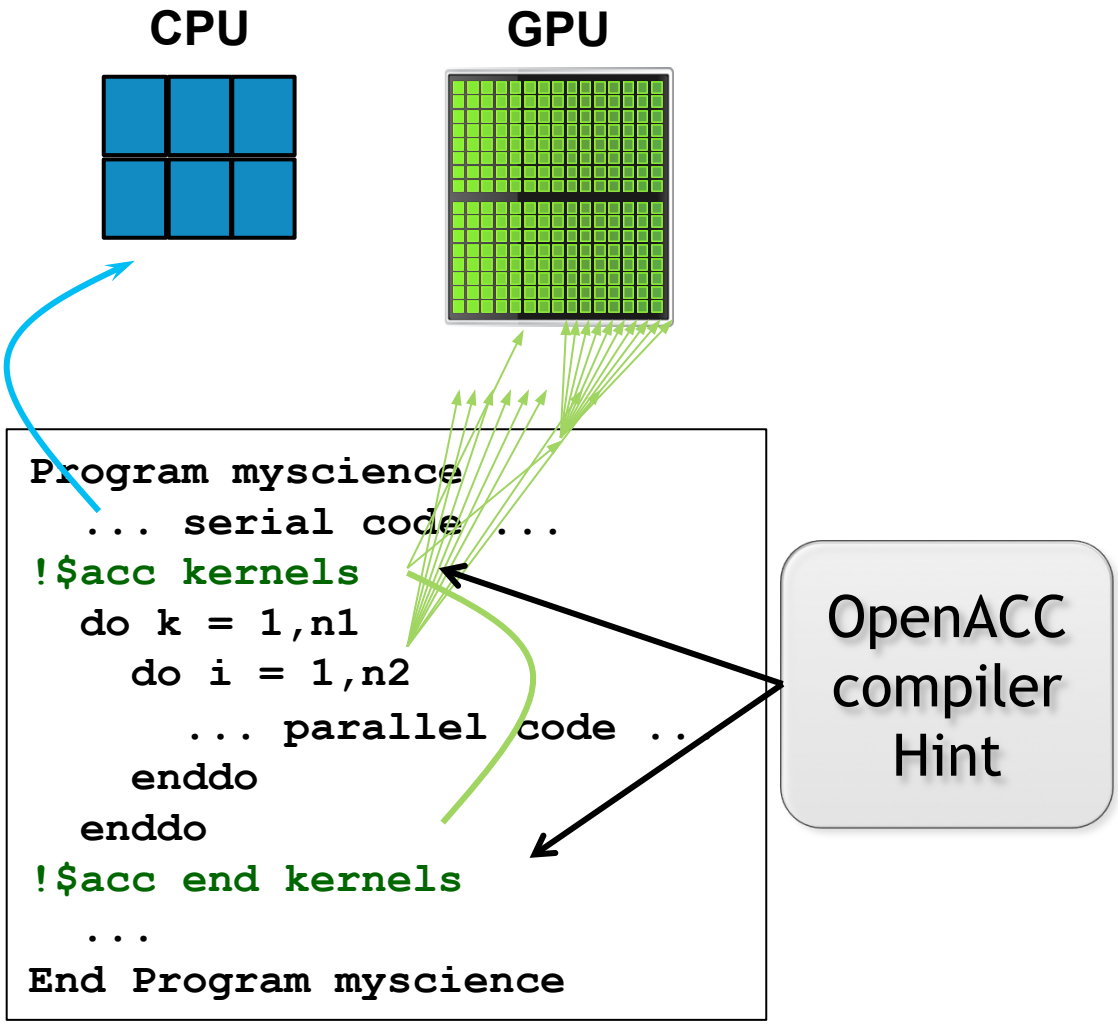
CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone:
developer.nvidia.com/cuda-tools-ecosystem

CUDA libraries described in:
developer.nvidia.com/gpu-accelerated-libraries

3 Ways to Accelerate Applications



OpenACC Directives



Simple Compiler hints

Compiler Parallelizes code

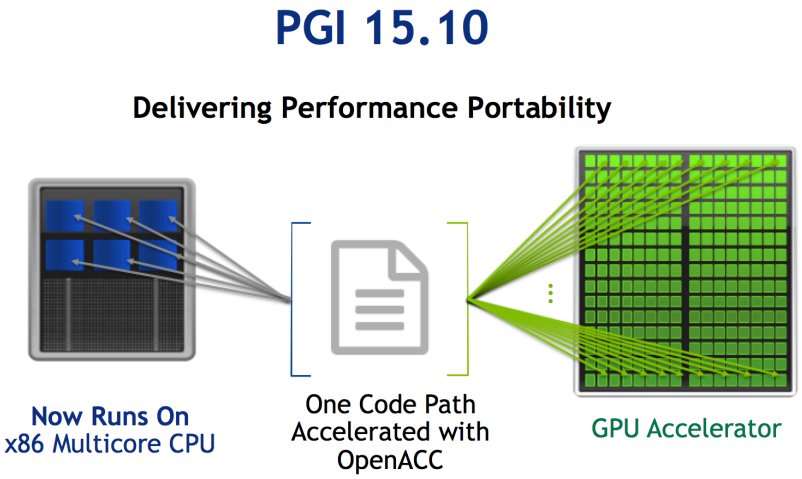
Works on many-core GPUs & multicore CPUs

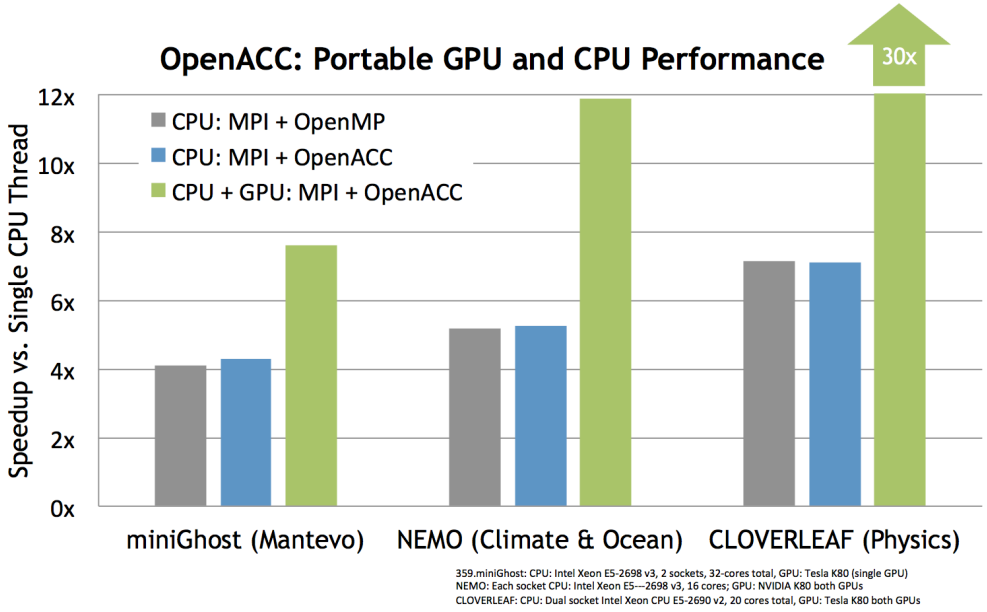
OpenACC: The Standard for GPU Directives

Easy: Directives are the easy path to accelerate compute intensive applications

Open: OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

Powerful: GPU Directives allow complete access to the massive parallel power of a GPU

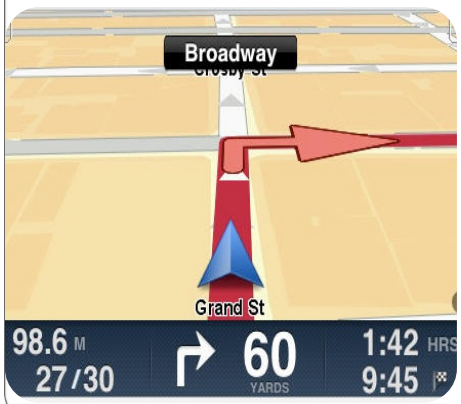




Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems



5x in 40 Hours

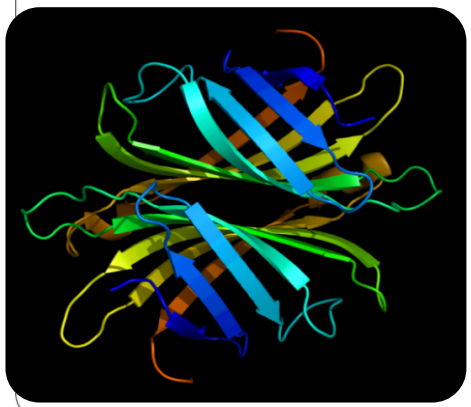
Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company

2x in 4 Hours

Interaction of Solvents and Biomolecules

University of Texas at San Antonio



5x in 8 Hours

"Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications."

-- Developer at the Global Manufacturer of Navigation Systems

3 Ways to Accelerate Applications

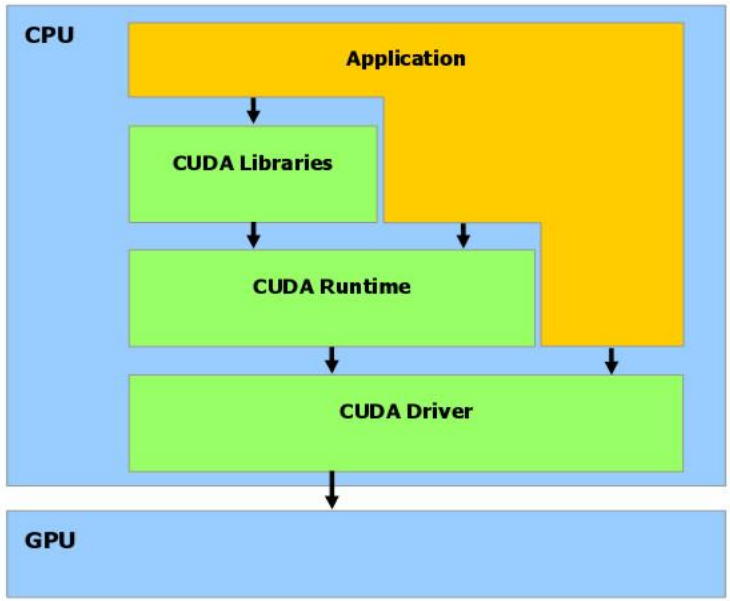
GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran, OpenACC, OpenMP4.5
C	CUDA C, OpenCL, OpenACC, OpenMP4.5
C++	CUDA C++, Thrust, OpenCL, OpenACC/OpenMP4.5
Python	PyCUDA/PyOpenCL, Numba, ...

CUDA

- "**C**ompute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor

- Targeted software stack
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Explicit GPU memory management



CUDA Programming Model

The GPU is viewed as a compute device that:

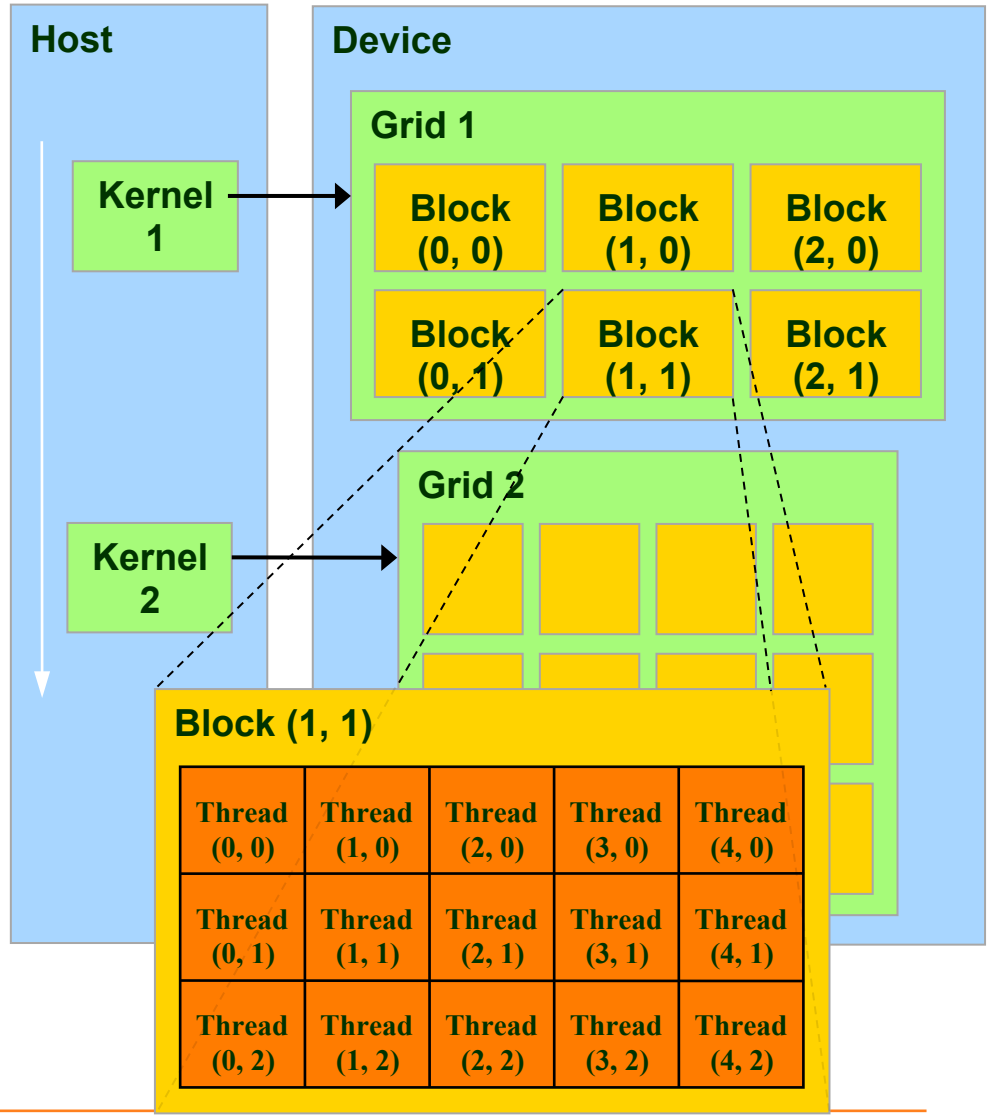
- Is a coprocessor to the CPU or host
- Has its own DRAM (device memory)
- Runs many threads in parallel
  - Hardware switching between threads (in 1 cycle) on long-latency memory reference
  - Overprovision (1000s of threads) → hide latencies

- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

- Kernel executed as a grid of thread blocks
  - All threads share data memory space
- Thread block is a batch of threads, can cooperate with each other by:
  - Synchronizing their execution:
    - For hazard-free shared memory accesses
  - Efficiently sharing data through the low latency shared memory
- Two threads from two different blocks cannot cooperate **(until CUDA8/9 and Volta)**
  - Unless thru slow global memory
- Threads and blocks have IDs



Extended C syntax

- Declspecs
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- Runtime API
  - Memory, symbol, execution management
- Function launch

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];


  __syncthreads()
  ...
  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	Host/device (DP,cc35)
`__host__ float HostFunc()`	host	Host

`__global__` defines a kernel function
- Must return `void`

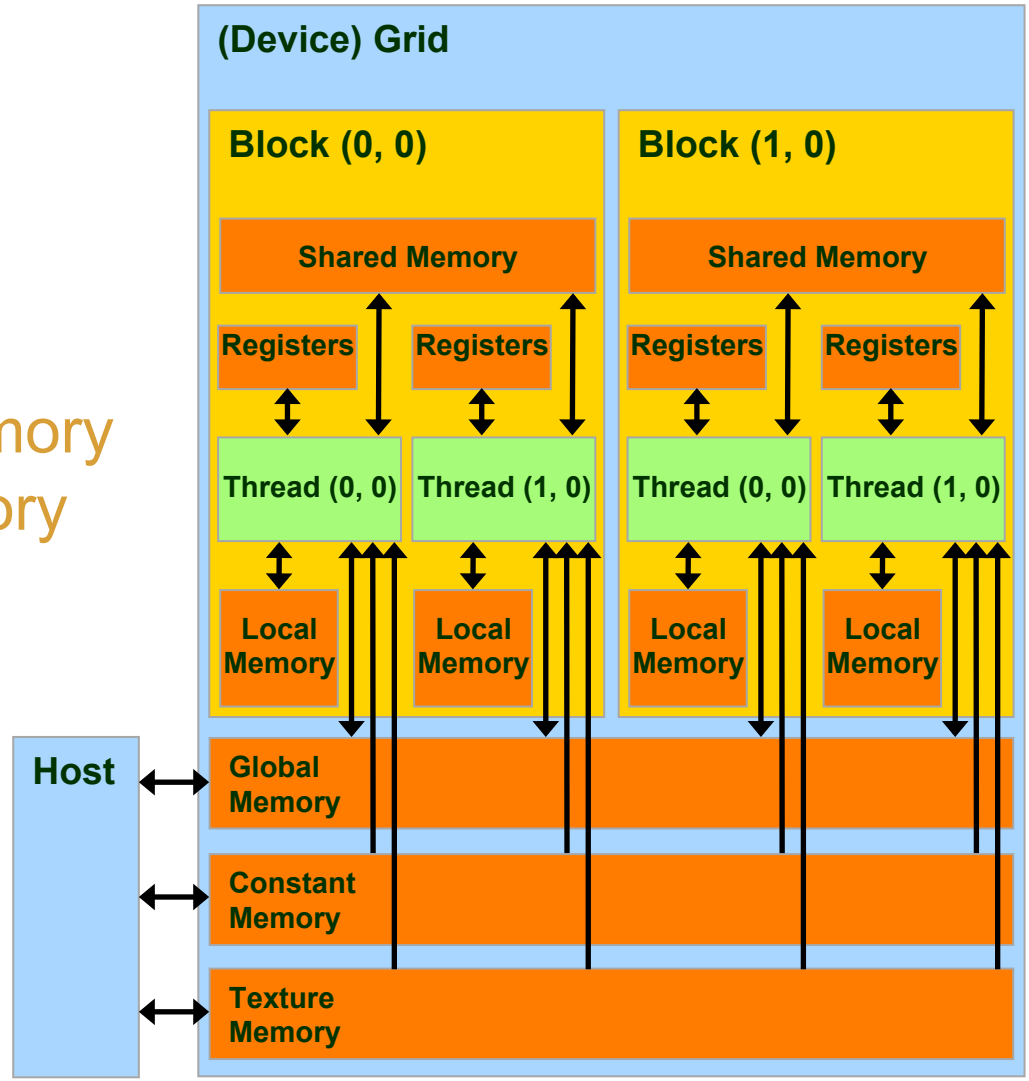
`__device__` and `__host__` define device and host functions respectively

CUDA Device Memory Space Overview

Each thread can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory
- Read only per-grid texture memory

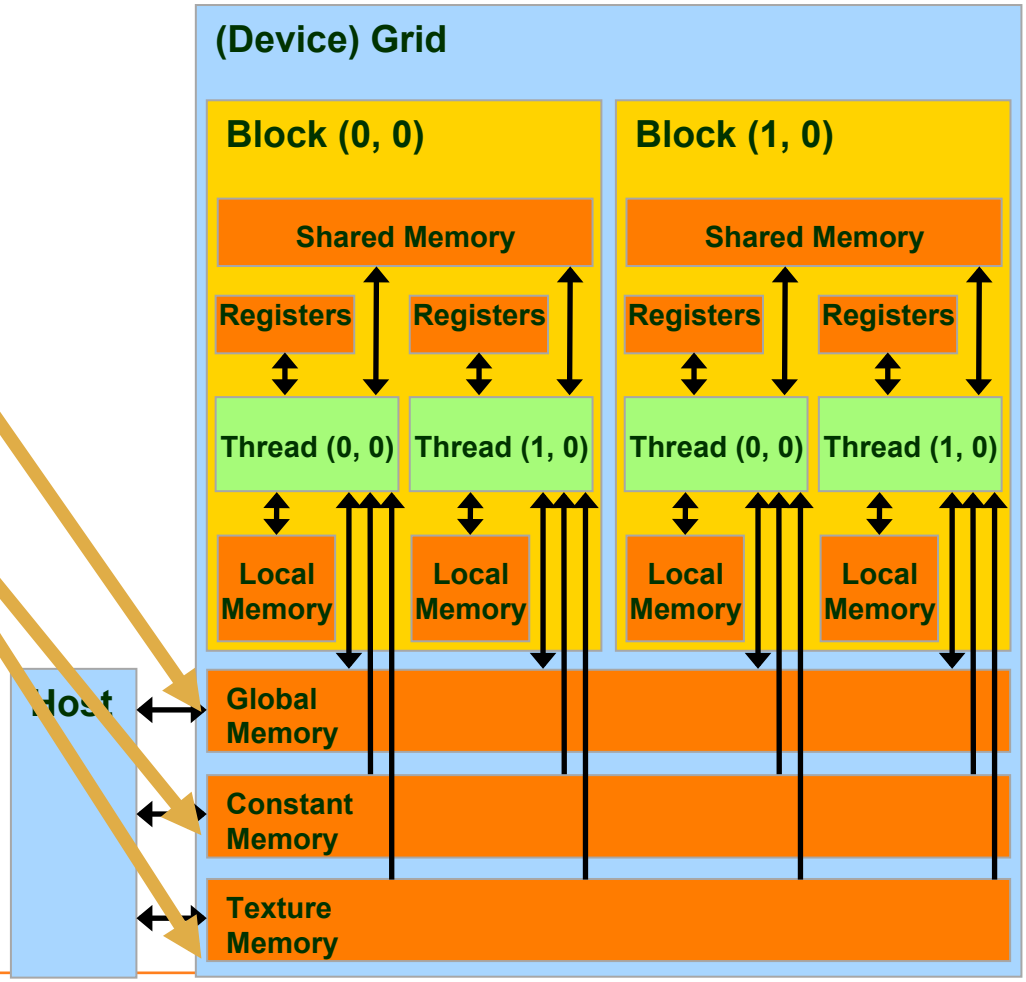
The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories

Global memory
- Main means of communicating R/W data between host and device
- Contents visible to all threads

Texture and Constant Memories
- Constants initialized by host
- Contents visible to all threads



Courtesy: NDVIA

Access times for various memories

Register – dedicated HW - single cycle
Shared Memory – dedicated HW - single cycle
Local Memory – DRAM, no cache - *slow*
Global Memory – DRAM, no cache - *slow*
Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
Instruction Memory (invisible) – DRAM, cached

Calling Kernel Function – Thread Creation

A kernel function must be called with an execution configuration:

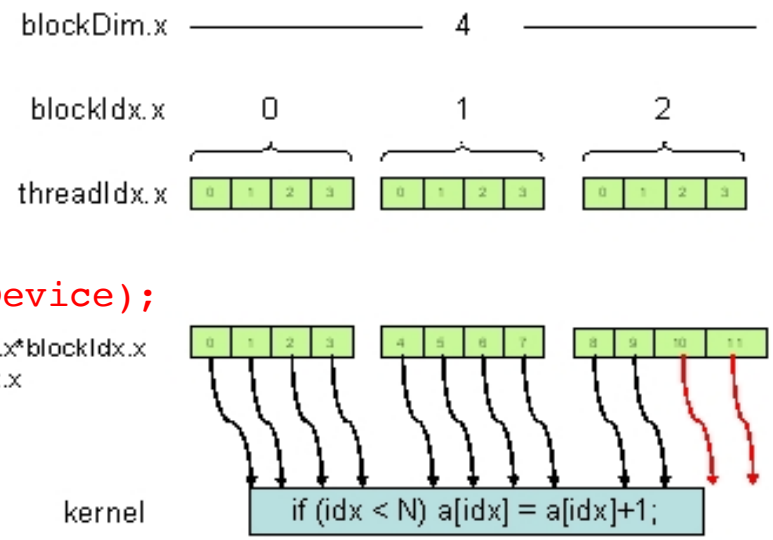
```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

Any call to a kernel function is asynchronous (CUDA 1.0 & later), explicit synchronization needed for blocking

Recursion in kernels supported (in 5.0/Kepler+)

Sample Code: Increment Array

```
main() { float *a_h, *a_d; int i, N=10; size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (i=0; i<N; i++) a_h[i] = (float)i;

  // allocate array on device
  cudaMalloc((void **) &a_d, size);

  // copy data from host to device
  cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);

  // do calculation on device:
  // Part 1 of 2. Compute execution configuration
  int blockSize = 4;
  int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
  // Part 2 of 2. Call incrementArrayOnDevice kernel
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);

  // Retrieve result from device and store in b_h
  cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

  // cleanup
  free(a_h);
  cudaFree(a_d);
}
```




```
__global__ void incrementArrayOnDevice(float *a,
int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx]+1.f;
}
```

Using per-block shared memory

Variables shared across block

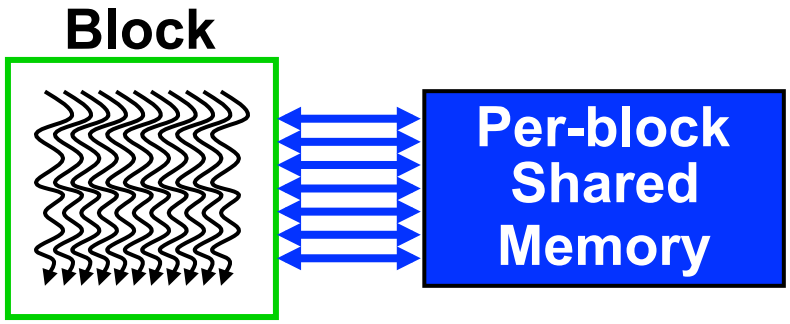
```
int *begin, *end;
```

Scratchpad memory

```
__shared__ int scratch[blocksize];
scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

Communicating values between threads

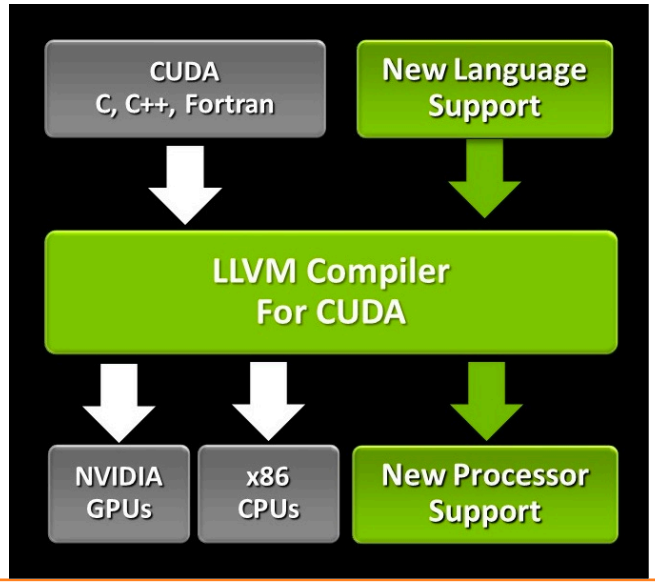
```
scratch[threadIdx.x] = begin[treadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

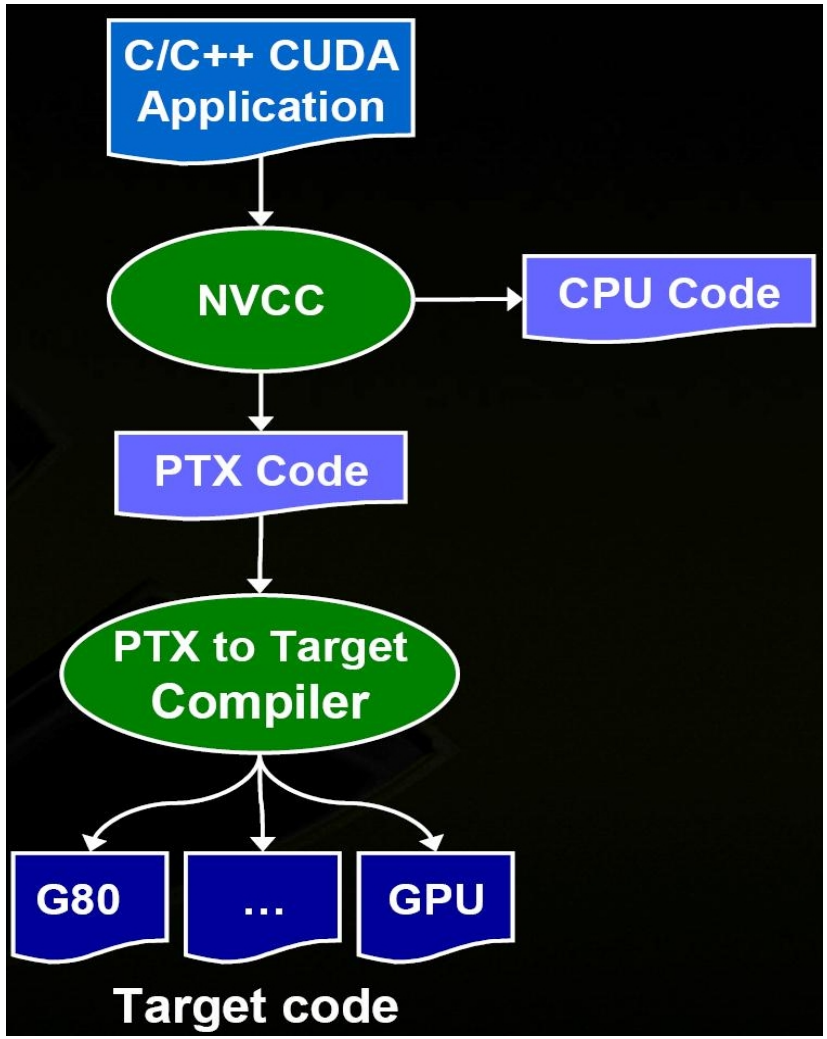


Compiling CUDA

- Call nvcc (driver) -- also C++/Fortran support
- LLVM front end
  - generates separate GPU & CPU code
- LLVM back end
  - generates GPU PTX assembly

- Parallel Threads eXecution (PTX)
  - virtual machine and ISA
  - gets assembled into actual machine code





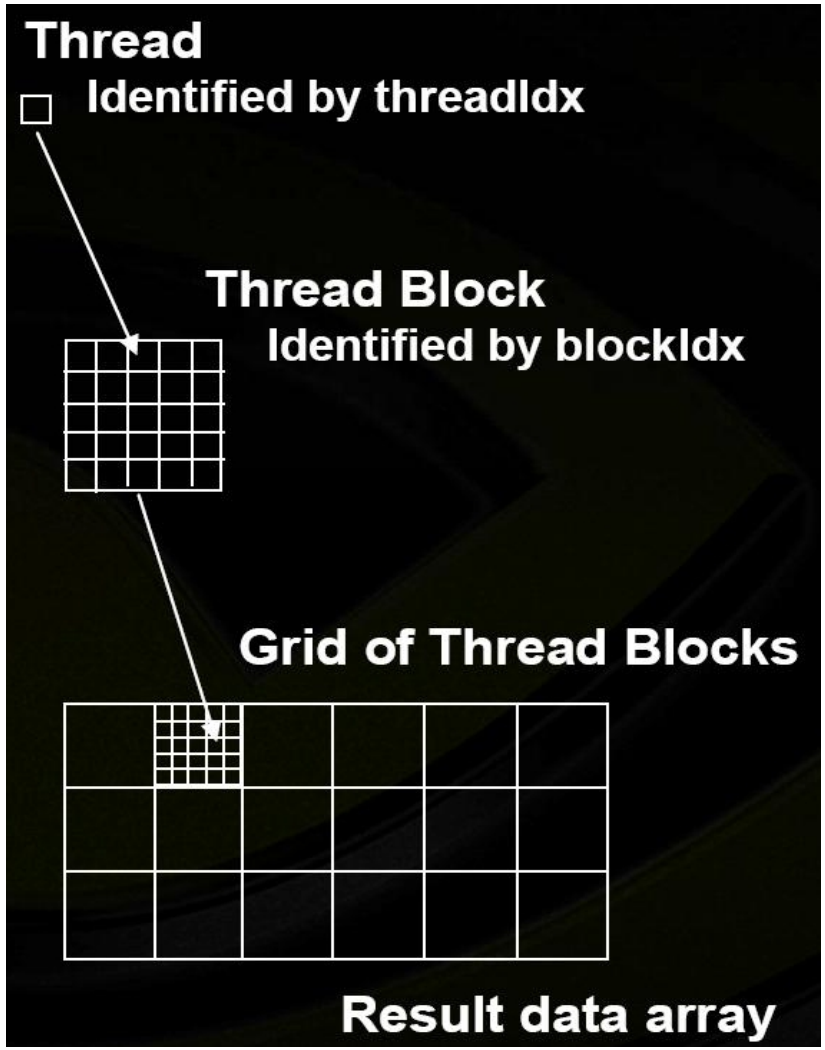
Execution model

Multiple levels of parallelism

- Thread block
  - max. 1024 threads/block
  - communication through shared memory (fast)
  - thread guaranteed to be resident
  - threadIdx, blockIdx
  - __syncthreads()
    → barrier for this block only!
    avoid RAW/WAR/WAW hazards when ref' shared/global memory

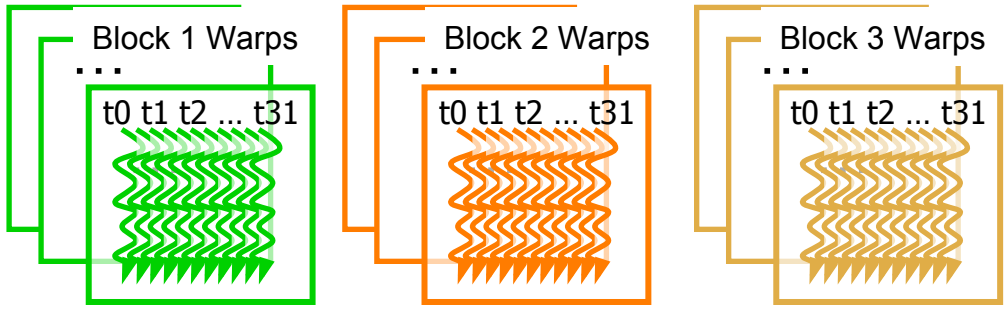
- Grid of thread blocks

  - F<<<nblocks, nthreads>>>(a, b, c)



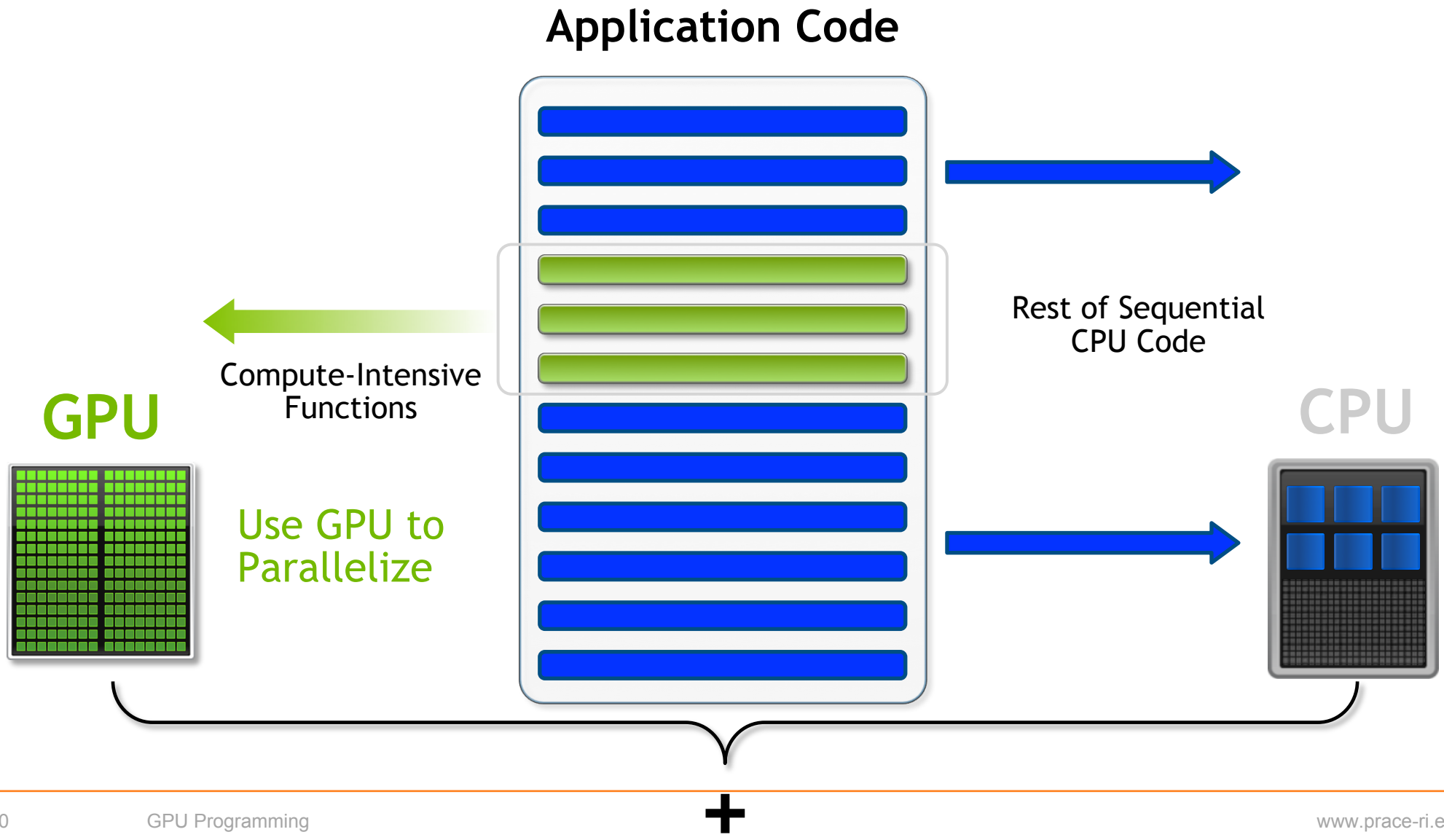
Execution model

- Each Block is executed as 32-thread “warps”
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps



- Warp divergence?
- Coalesced accesses?

Small Changes, Big Speed-up



Why GPU Computing Matters?

Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Cray Inc. 4.2MW

8.1 PFlops

4.2 Megawatts

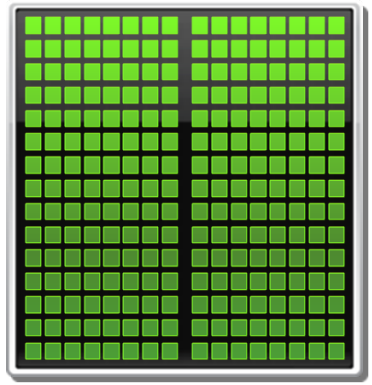
4200 homes

4.2 Megawatts

DGX SATURNV - NVIDIA DGX-1, XeonE5-2698v4 20C 2.2GHz + 8x Tesla P100, NVIDIA. 350KW

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Many Parallel Tasks



10x performance/socket

> 5x energy efficiency

3.3 PFlops

Traditional CPUs are not economically feasible

Era of GPU-accelerated computing is here

GPU architecture evolution

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
FP64 GFLOPs	1680	213	5304
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm²	601 mm²	610 mm²
Manufacturing Process	28-nm	28-nm	16-nm

GPUs & High-performance Libraries

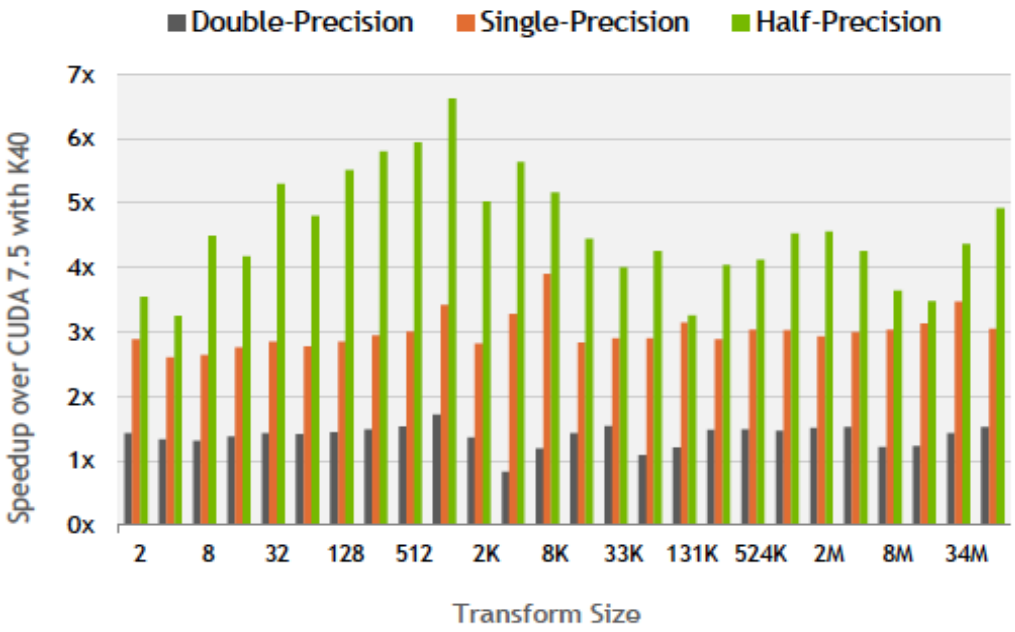
cuFFT

Complete Fast Fourier Transforms Library

Complete Multi-Dimensional FFT Library

- Simple “drop-in” replacement of a CPU FFTW library
- Real and complex, single- and double-precision data types
- Includes 1D, 2D and 3D batched transforms
- Support for half-precision (FP16) data types
- Supports flexible input and output data layouts
- XT interface now supports up to 8 GPUs

> 6x Speedup with Half-Precision on P100



- Speedup of P100 with CUDA 8 vs. K40m with CUDA 7.5
- cuFFT 7.5 on K40m, Base clocks, ECC on (r352)
- cuFFT 8.0 on P100, Base clocks, ECC on (r361)
- 1D Complex, Batched transforms on 28-33M elements
- Input and output data on device
- Host system: Intel Xeon Haswell single-socket 16-core E5-2698 v3 @ 2.3GHz, 3.6GHz Turbo
- CentOS 7.2 x86-64 with 128GB System Memory

cuBLAS

Dense Linear Algebra on GPUs

Complete BLAS Library Plus Extensions

Supports all 152 standard routines for single, double, complex, and double complex

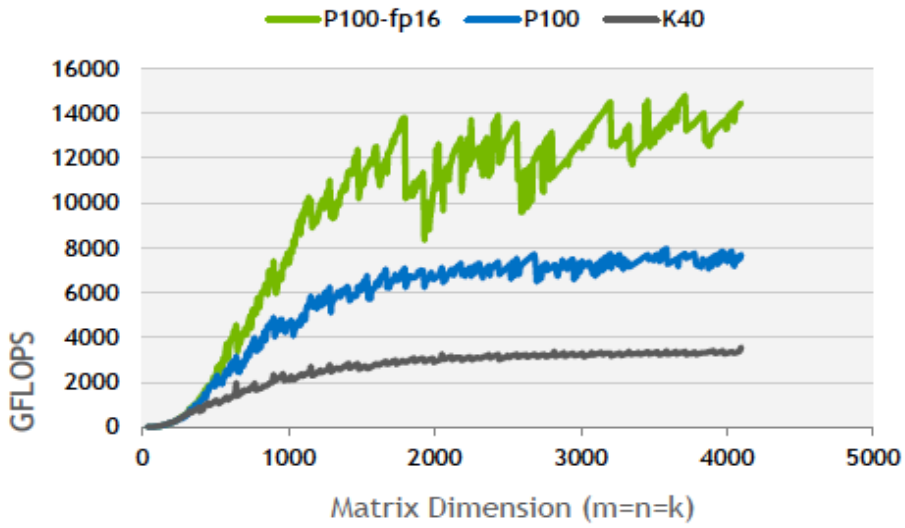
Supports half-precision (FP16) and integer (INT8) matrix multiplication operations

Batched routines for higher performance on small problem sizes

Host and device-callable interface

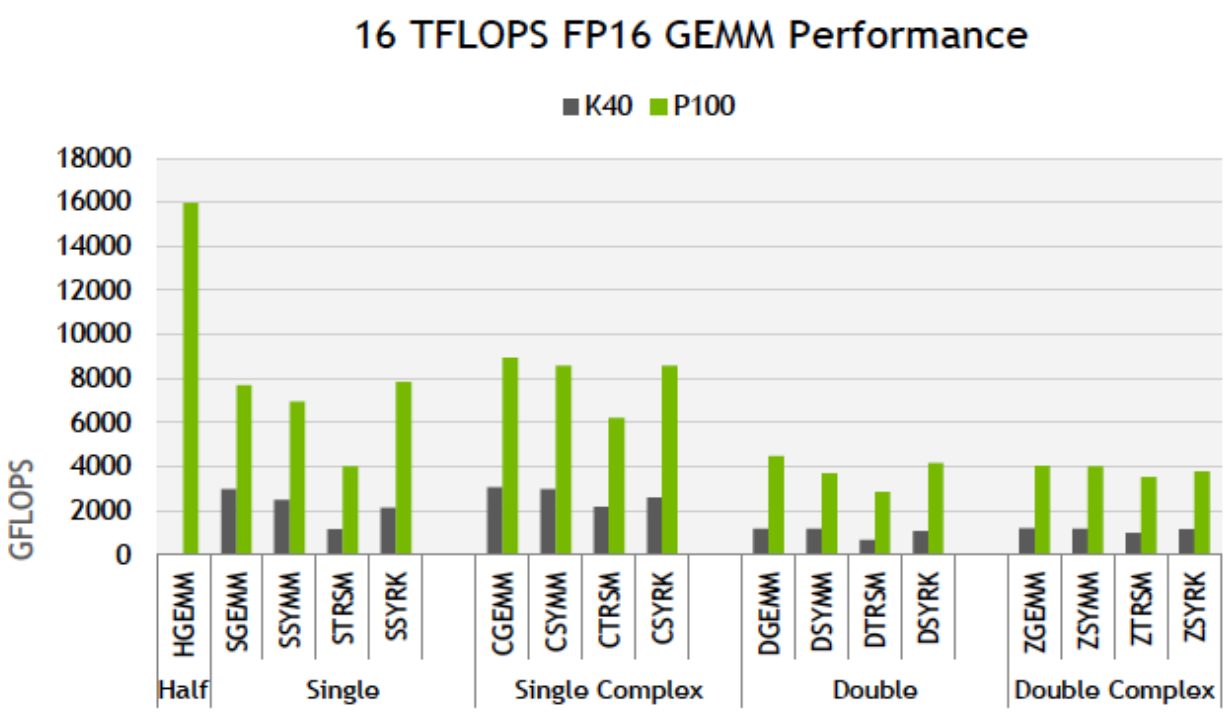
XT interface supports distributed computations across multiple GPUs

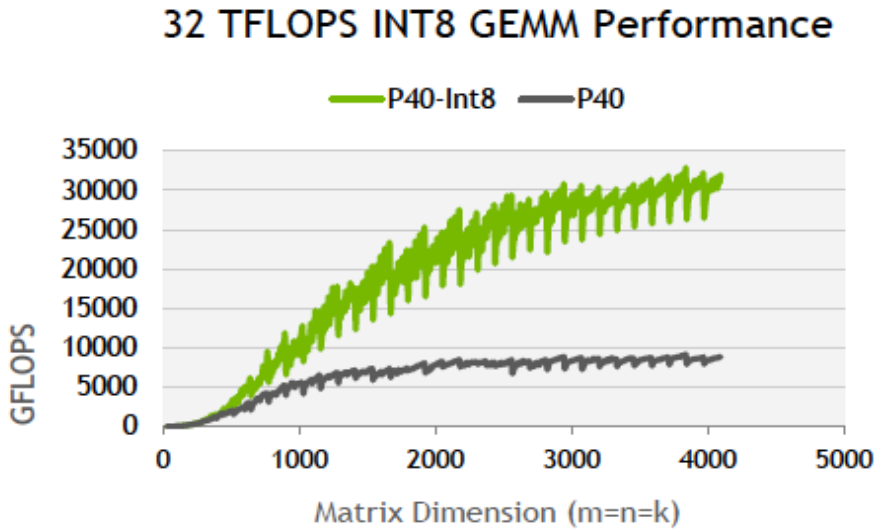
> 4x Faster GEMM Performance with FP16 on P100



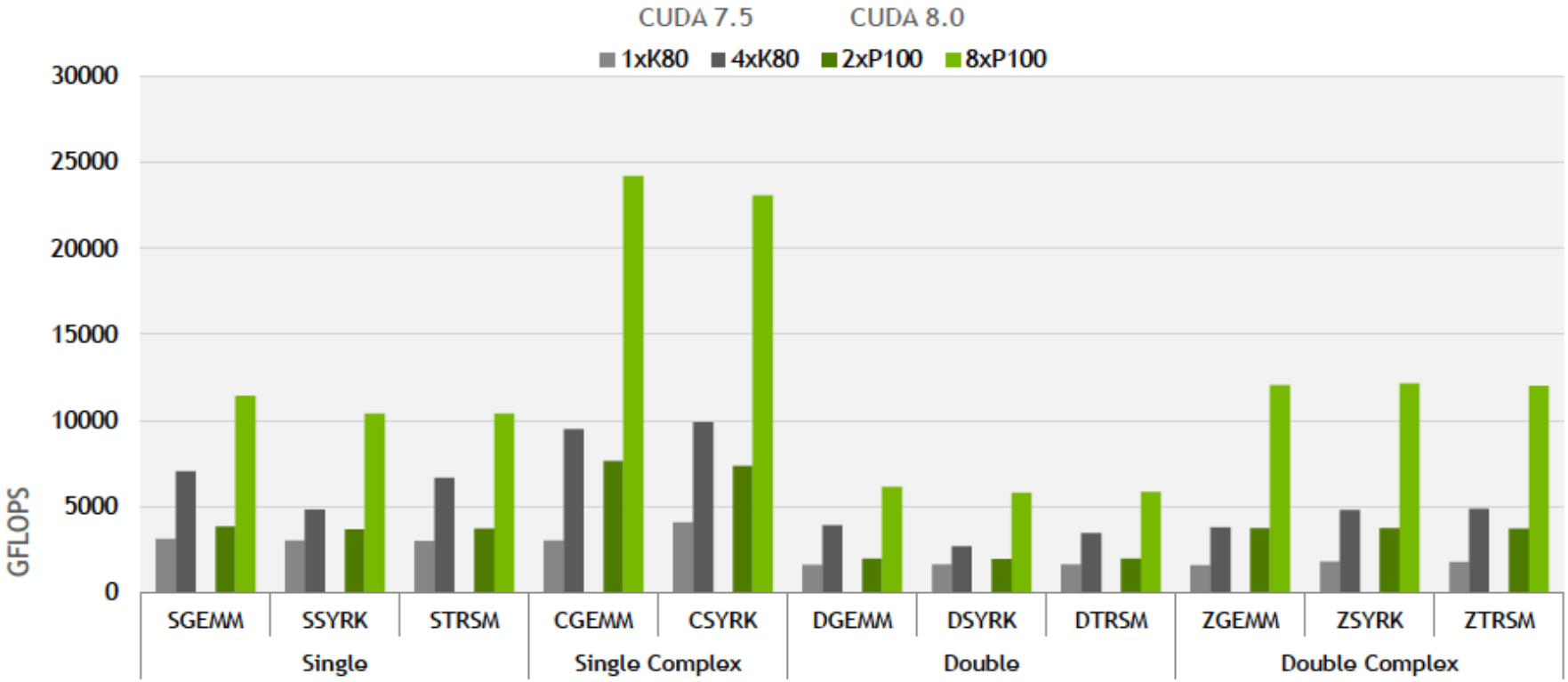
- Comparing GEMM performance on K40m (FP32) and P100 (FP32 and FP16)
- cuBLAS 8 on P100, Base clocks (r361)
- cuBLAS 8 on P40, Base clocks (r367)
- cuBLAS 7.5 on K40m, Base clocks, ECC ON (r352)
- Input and output data on device
- Host system: Intel Xeon Haswell single-socket 16-core E5-2698 v3@ 2.3GHz, 3.6GHz Turbo
- CentOS 7.2 x86-64 with 128GB System Memory
- m=n=k=4096

cuBLAS: > 8 TFLOPS SINGLE PRECISION





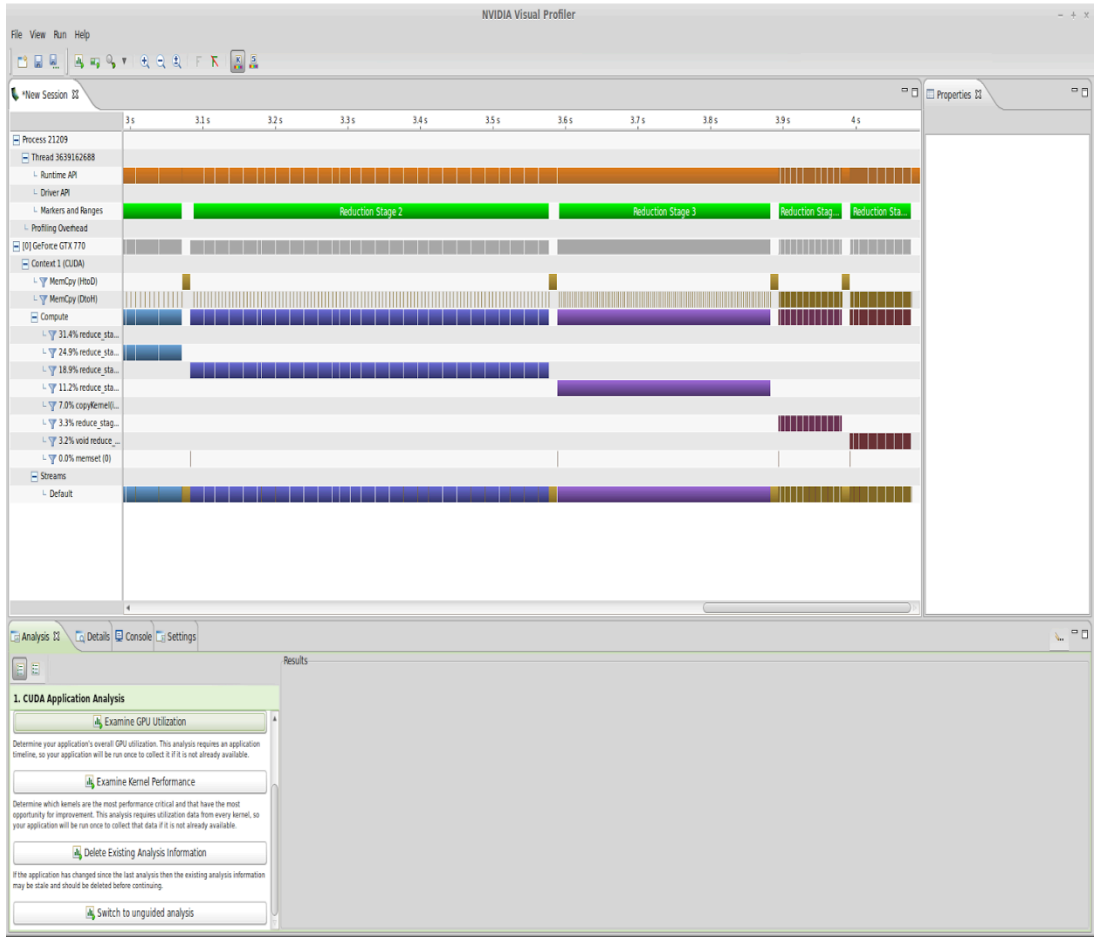
cuBLAS-XT: > 24 TFLOPS ON A SINGLE NODE



GPU Profiling and Debugging Tools

NVIDIA Visual Profiler

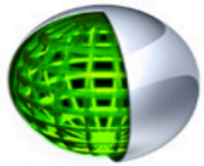
- Standalone application with CUDA Toolkit
- Visualize performance
- Timeline
- Power, clock, thermal profiling
- Concurrent profiling
- Profile activity on both GPU and CPU
- nvprof - command line tool



GPU Debugging Solutions

allinea DDT

TotalView®



Allinea DDT

Provides application developers with a single tool that can debug hybrid MPI, OpenMP, CUDA and OpenACC applications on a single workstation or GPU cluster.

TotalView

A GUI-based tool that allows you to debug one or many processes/threads with complete control over program execution, from basic debugging operations like stepping through code to concurrent programs that take advantage of threads, OpenMP, MPI, or GPUs.

NVIDIA® Nsight™

The ultimate development platform for heterogeneous computing. Work with powerful debugging and profiling tools. Find out about the Ecilipse Edition and the graphics debugging enabled Visual Studio Edition.

CUDA-GDB

Delivers a seamless debugging experience that allows you to debug both the CPU and GPU portions of your application simultaneously. Use CUDA-GDB on Linux or MacOS, from the command line, DDD or EMACS.

GPU Usage from Python

- PyCUDA/PyOpenCL
  - Developed by Andreas Klöckner
  - Built on top of CUDA Driver API
- Numba
- Anaconda Accelerate offers bindings to the important CUDA libraries:
  - cuBLAS
  - cuFFT
  - cuSPARSE
  - cuRAND
  - ….
- Many high-level libraries use these under the hood:
  - this abstracts complexity even more
  - take for example deep/machine learning frameworks:
    - seamless multi-GPU programming

PyCUDA example

```
1  import pycuda.driver as cuda
2  import pycuda.autoinit, pycuda.compiler
3  import numpy
4
5  a = numpy.random.randn(4,4).astype(numpy.float32)
6  a_gpu = cuda.mem_alloc(a.nbytes)
7  cuda.memcpy_htod(a_gpu, a)
```

- More examples in the Hands-on!

```
1  mod = pycuda.compiler.SourceModule("""
2      __global__ void twice(float *a)
3      {
4        int idx = threadIdx.x + threadIdx.y*4;
5        a[idx] *= 2;
6      }
7      """)
8
9  func = mod.get_function("twice")
10 func(a_gpu, block=(4,4,1))
11
12 a_doubled = numpy.empty_like(a)
13 cuda.memcpy_dtoh(a_doubled, a_gpu)
14 print a_doubled
15 print a
```

THANK YOU FOR YOUR ATTENTION

www.prace-ri.eu

Acknowledgement

H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).