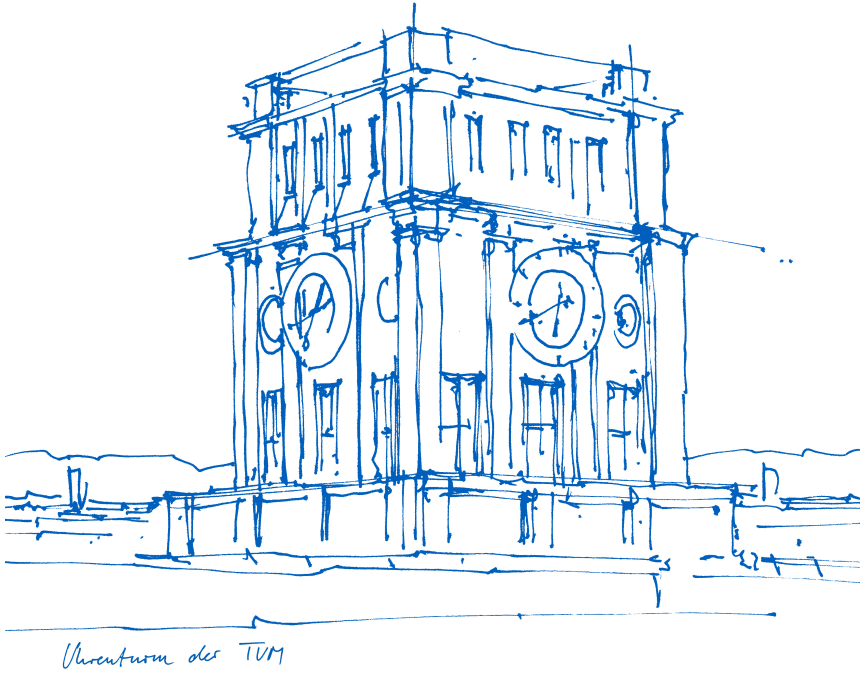
Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Exercises: Amir Raoofy
Technische Universität München
Fakultät für Informatik

Lecture 12:
Accelerators



Summary From Last Time

HPC Systems are increasingly concurrent, which we need to exploit
- Weak scaling: constant problem size per HW thread/core/node
- Strong scaling: constant total problem size

Impact on Programmability
- Work at scale is getting harder -> try to keep small reproducers
- Tool chain (e.g., debuggers) need to be adjusted as we scale

Impact on Algorithms and Data Structures
- Avoiding any algorithm or data structure that is O(N) or worse
- May have to rethink algorithms and/or data structures to be used

Impact on Mapping Code to Machines
- Mapping of applications to the topology of the machine can be crucial at scale
- MPI topologies (grids or graphs) can help, as they enable MPI to optimize

Impact on I/O
- I/O can be a large fraction of the execution time
- Need to exploit parallelism and tweak parameters (potentially on every system)

How to Scale Performance Further?

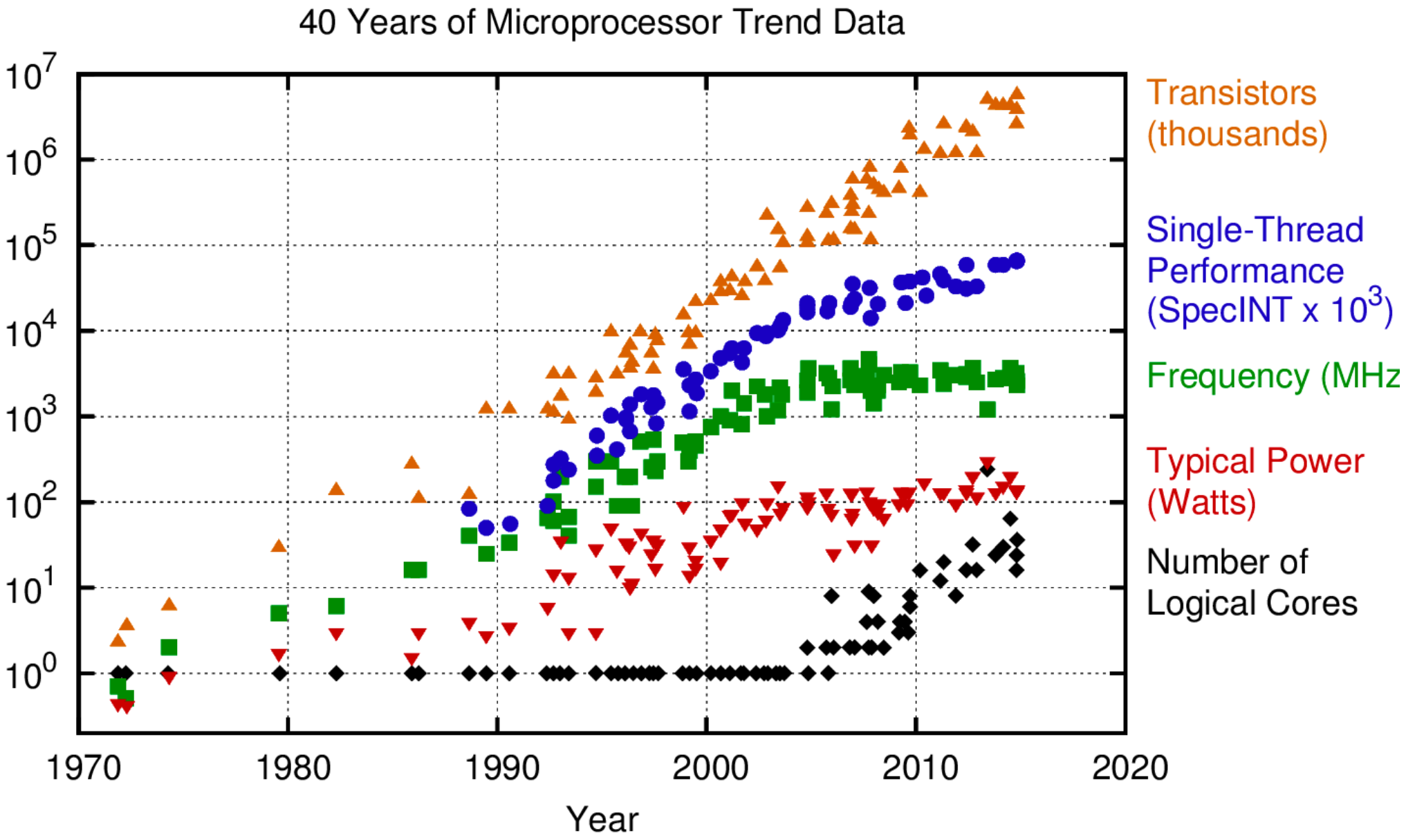
Performance scalability is getting more and more limited

- Single thread performance increases limited
- Multicore scaling limited by memory bandwidth
- Alternative: growing number of nodes

Technological limitations

- Floorspace
- Power, Energy, Heat
- Harder to use all transistors

Technology Trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

How to Scale Performance Further?

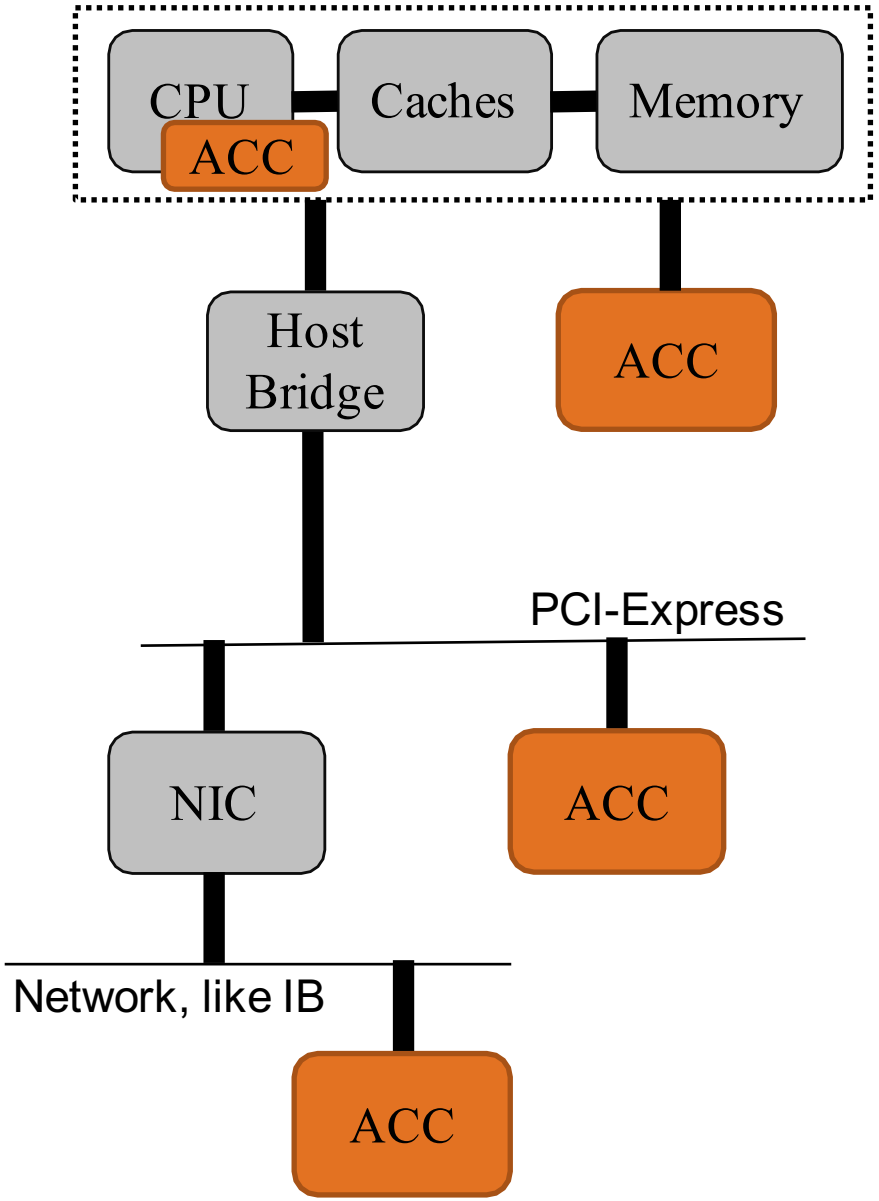
Performance scalability is getting more and more limited
- Single thread performance increases limited
- Multicore scaling limited by memory bandwidth
- Alternative: growing number of nodes

Technological limitations
- Floorspace
- Power, Energy, Heat
- Harder to use all transistors

Question: can we use transistors in a better way
- Is general purpose computing the right way?

Idea: extra hardware to accelerate computation
- Specialized operations
- No longer capable of (easily) sustaining general operations
- ”Hosted” by a (general purpose) host system
- Impact on programmability

Connecting an Accelerator



Specialized hardware attached to a host

Most common (as of now): PCIe
- Add-on cards
- Access and data transfer via PCI bus
- Advantage: easy to integrate
- Disadvantage: PCI can be a bottleneck

Alternative: direct processor connection
- Needs new interfaces
- Removes bottlenecks

One step further: CPU integration
- Direct access from within CPU
- Either fully integrated or with packaged dies

External booster concept

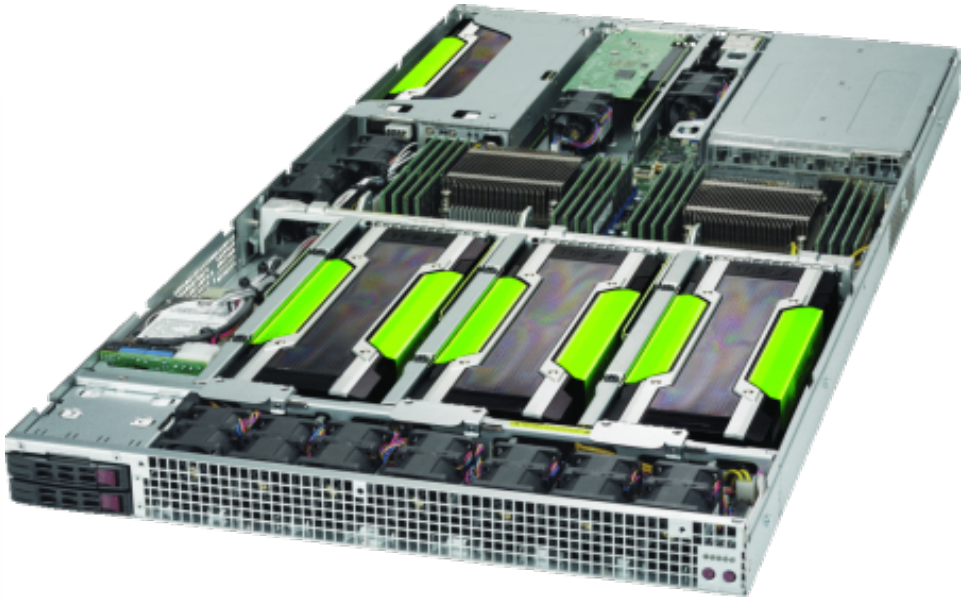
Examples of Accelerators

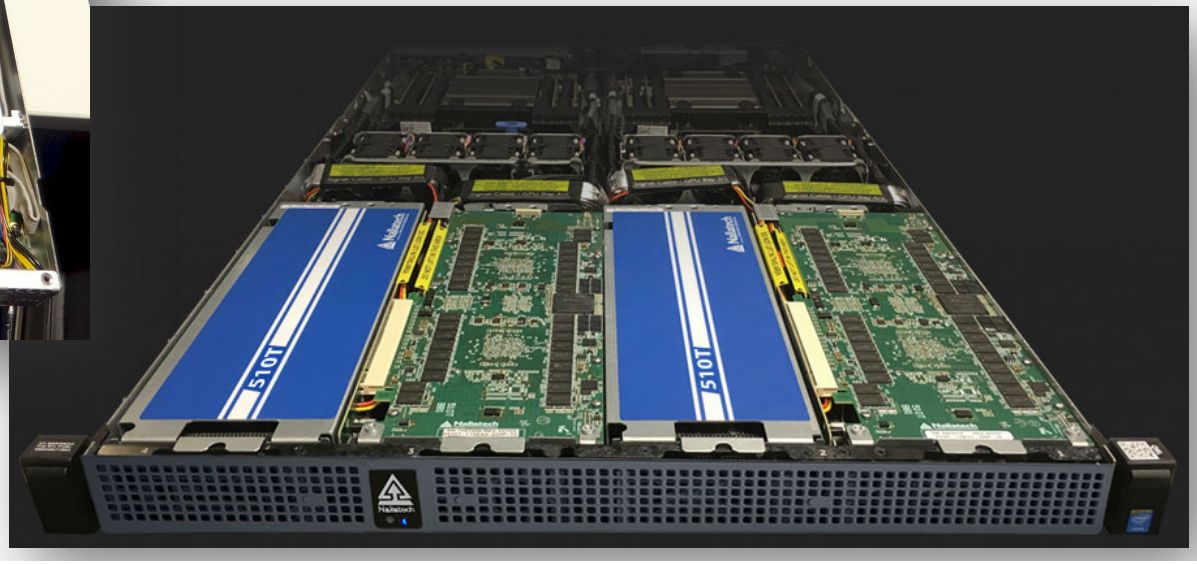
Graphics processing units
- Example: NVIDIA's Tesla V100 (Volta Generation)
- SIMD like operation

Field Programmable Gate Arrays (FPGAs)
- Example: board by Xilinx for Virtix7
- Reconfigurable logic
- Flexible data paths
- Programmed in VHDL or Verilog

Dataflow engines
- Example: Maxeler Galava PCI-e DFE Card
- Dataflow principle
- Programmed through separately developed language

System Integration





Graphics Processing Units (GPUs)

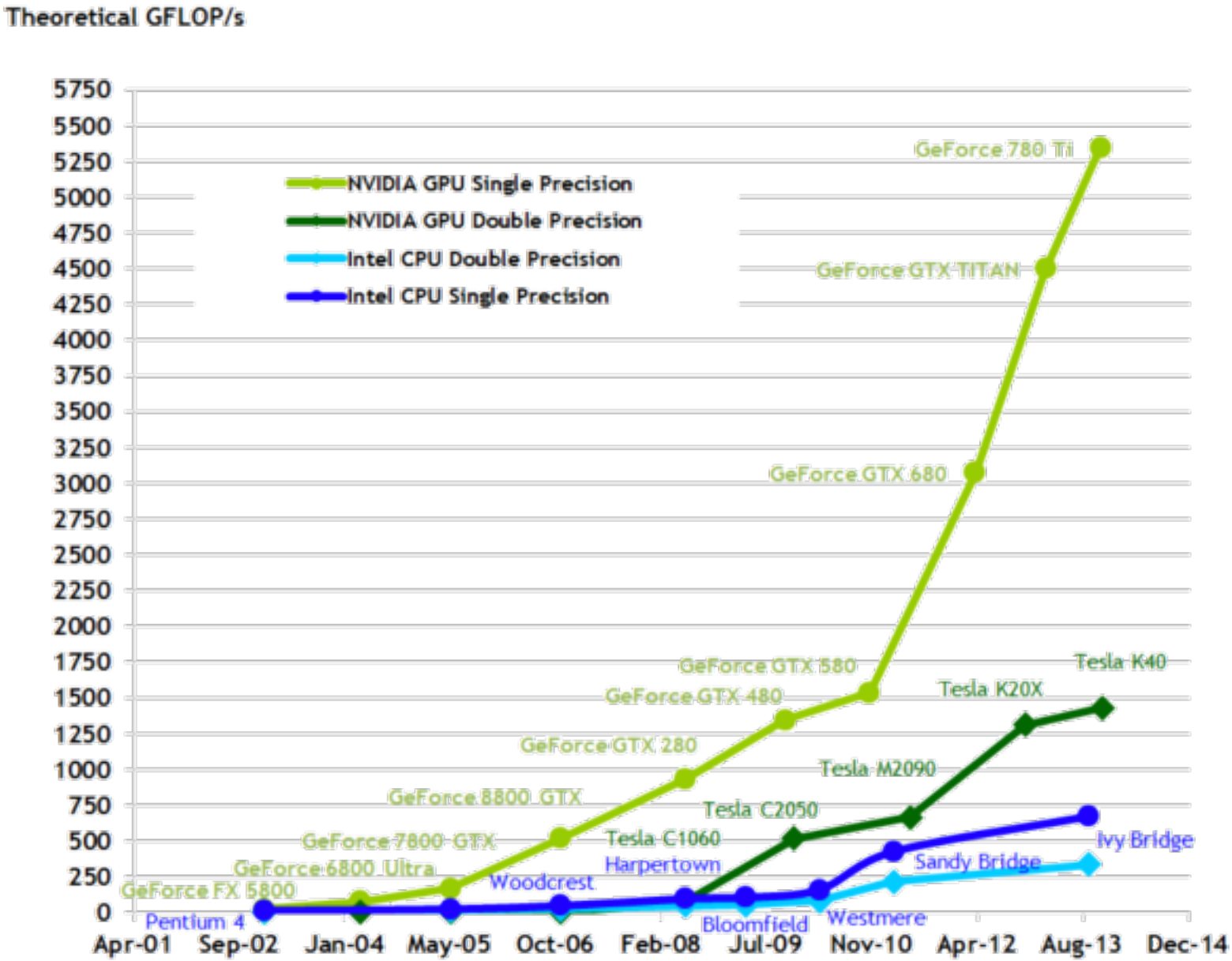
Originally developed to support fast graphics displays
- Triangle drawing, shading, texture mapping, …
- Paved the way for high-end graphics workstations

Evolution
- Originally hardwired pipeline
- Increasingly programmable over the years
- Led to usage of GPUs for general programming
  - Shader programming
  - “Re-Use” of languages/libraries like OpenGL

First fully programmable GPUs around 2006
- Aka. General Purpuse GPUs (GPGPUs)
- NVIDIA introduces CUDA to program GPUs

Today most GPUs are programmable
- From AMD GPUs to integrated SoCs as on the Raspberry Pi
- Programming varies and is often low-level

Potential of GPUs



From: NVIDIA, CUDA C Programming Guide

Concepts of a GPU

Key concept: vector / SIMD processing

- Massively parallel processing in a single chip

Advantages:

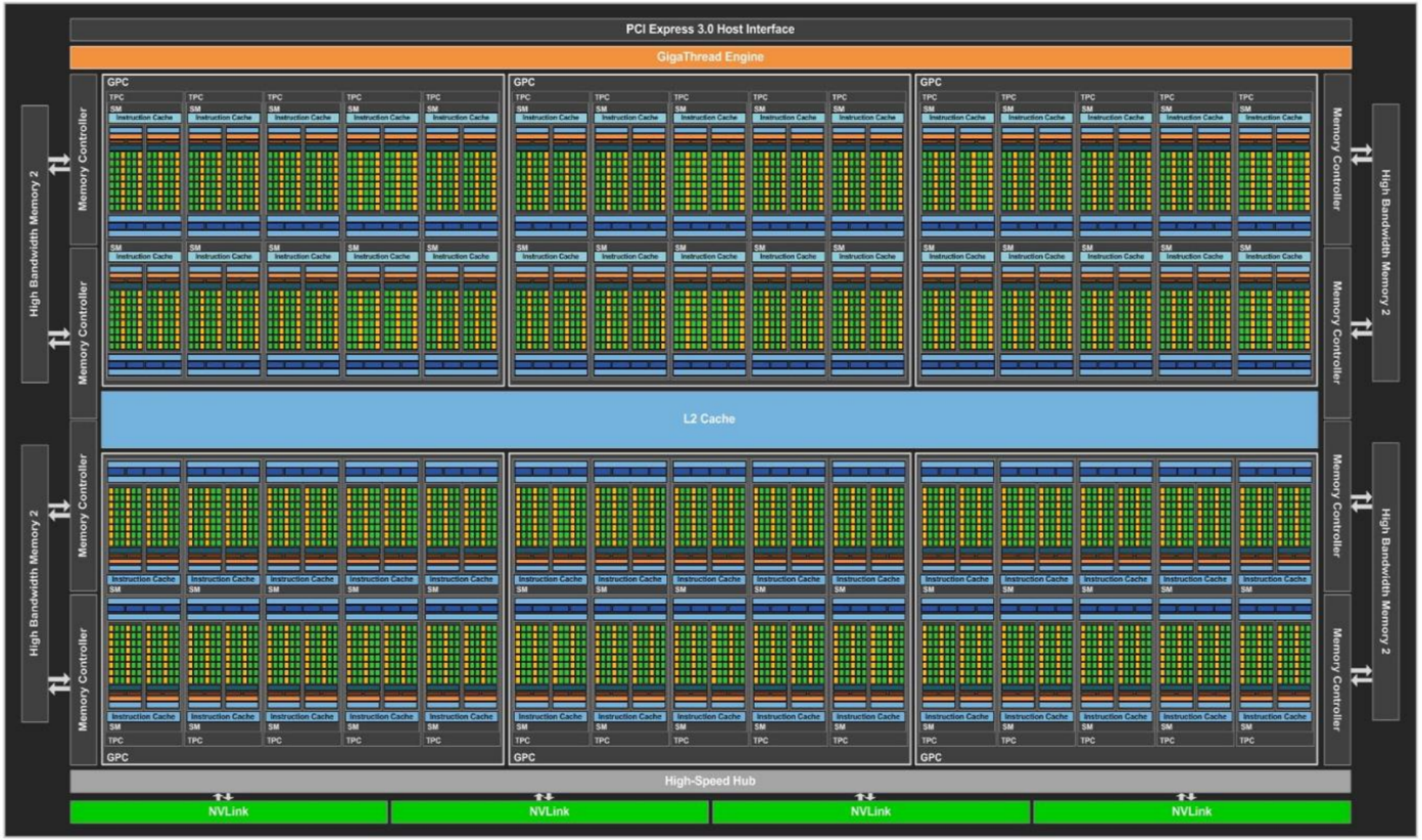
- Reduces complexity of the processor
- Increases energy efficiency

Disadvantages

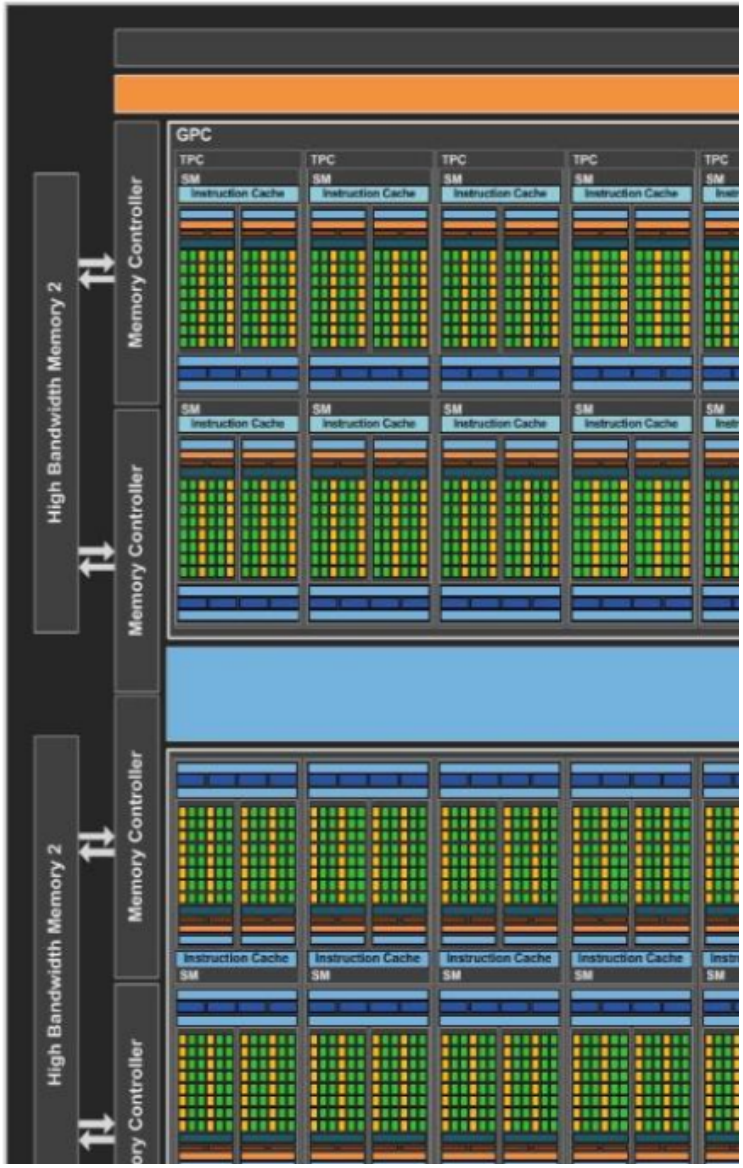
- Reduced flexibility (must vectorize problem)
- Additional programming complexity

GPUs are typically designed hierarchical

Architectural Diagram of NVIDIA's Pascal



Architectural Diagram of NVIDIA's Pascal



Compute hierarchy
- SM = Streaming Multiprocessors (60/TPC)
- TPC = Texture Processing Cluster (30/GPC)
- GPC = Graphics Processing Cluster (6)

Central L2 Cache

Connection to host via PCI Express

Multiple memory controllers
- HBM = High Bandwidth Memory

Gigathread Engine
- Schedule kernels to GPC/TPC/SM
- Enables sharing of GPU
- Further advanced in successor generations

Streaming Multiprocessor in Pascal

32 FP64 CUDA cores
64 K registers
L1 shared cache
64 KB shared memory

Thread blocks
as basic scheduling unit

Mapped to Warps with
execute a fixed number
of threads with single
Instruction counter

Nvidia Pascal Memory Structure

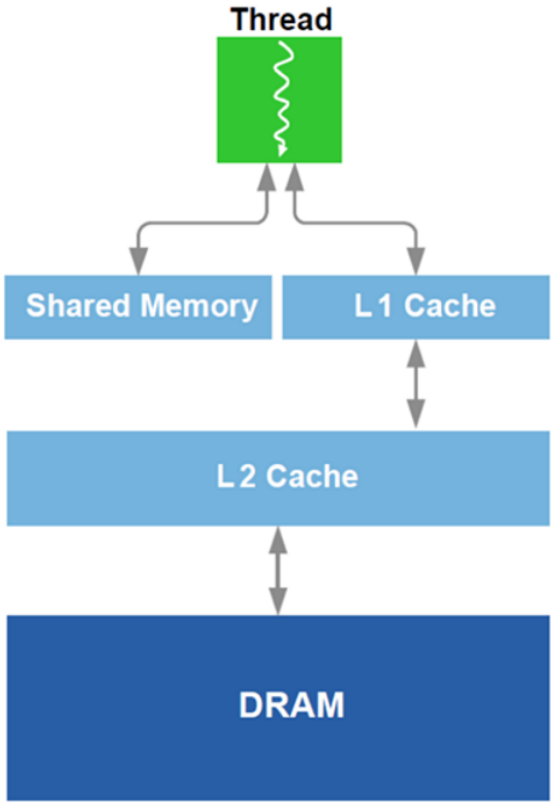
Registers
- CUDA threads have own registers.

L1 cache / texture cache
- 32 KB

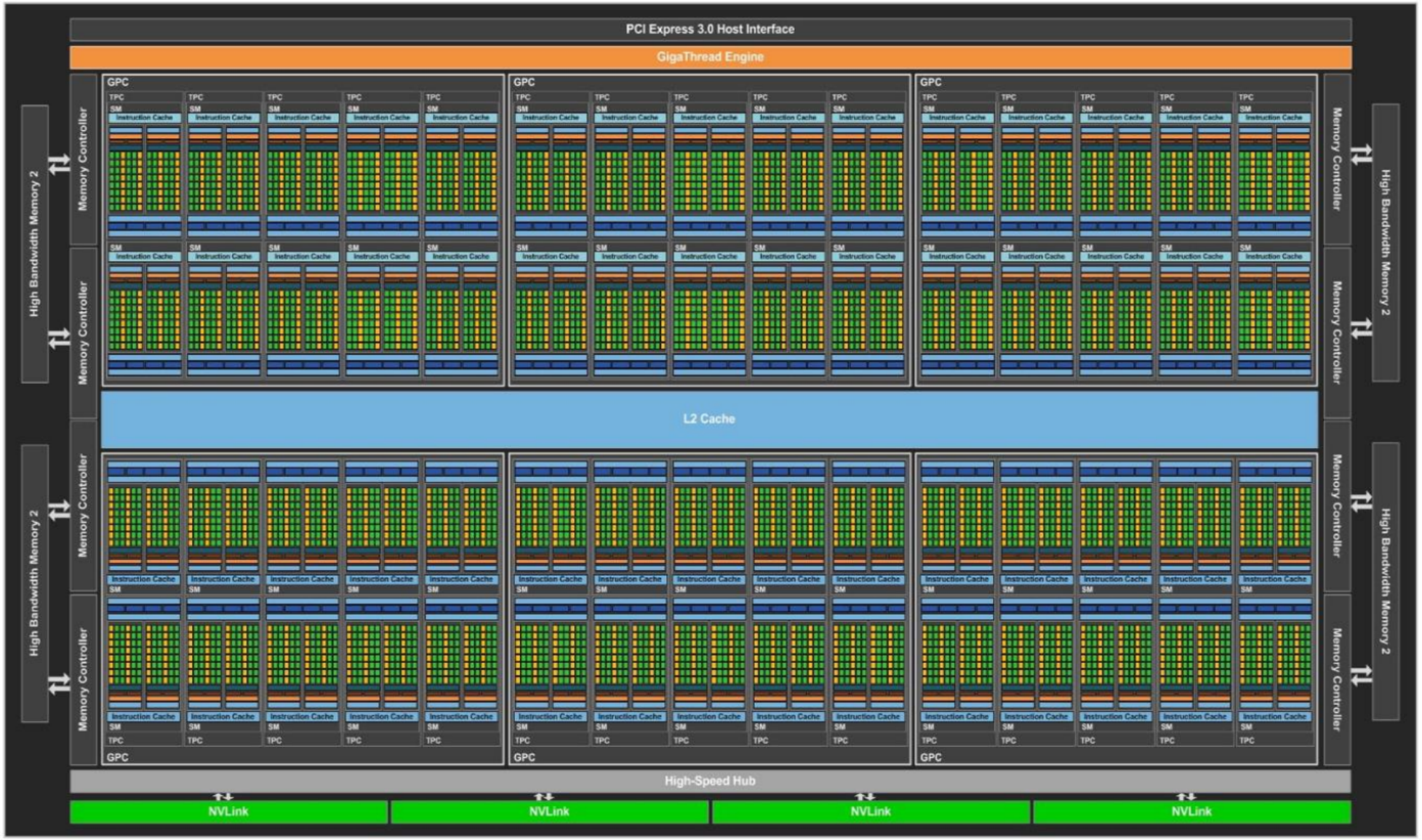
Shared memory
- 64 KB
- Small, but very high bandwidth

Global L2 Cache
- 4 MB

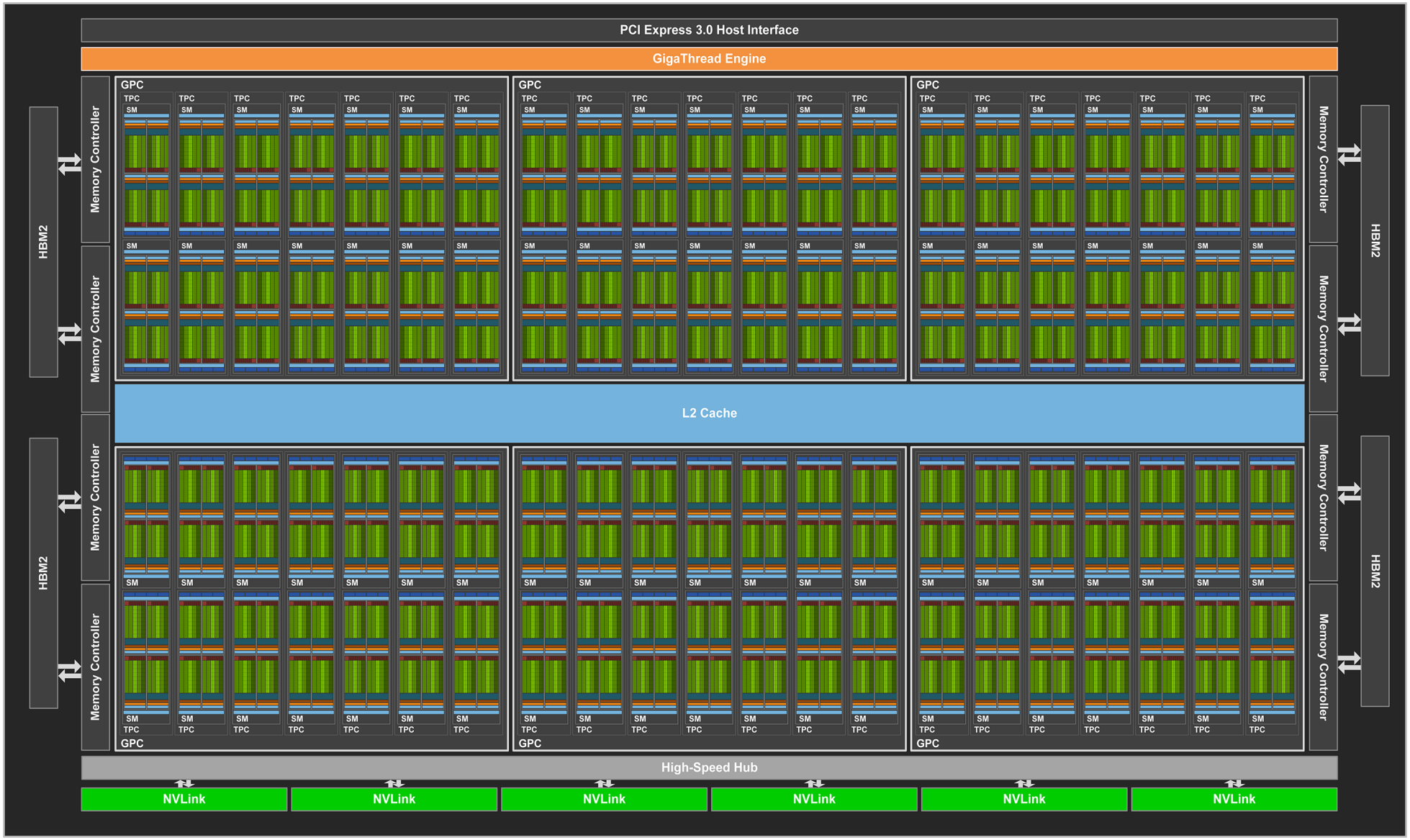
High Bandwidth Memory (HBM)
- Up to 16 GB
- 4096 bit, 8 memory controller



Architectural Diagram of NVIDIA's Pascal



Architectural Diagram of NVIDIA's Volta



SM Architecture of NVIDIA's Volta

SM Architecture of NVIDIA's Volta

Similar hierarchy as in Pascal
- Only needs one dispatch unit
- Mixed core options
  - Integer
  - FP32
  - FP64
- Includes FP16 support

New in Volta: Tensor-Cores
- Programmable matrix-multiply and accumulate
- Useful for AI/Deep Learning
- 125 Tflop/s (reduced precision)

SM Capabilities over NVIDIA Generations

	Kepler GK210	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max Regs / Thread Block	65536	32768	65536	65536
Max Regs / Thread	255	255	255	255
Max Thread Block Size	1024	1024	1024	1024
Shared Memory / SM	16/32/48 KB	96 KB	64KB	up to 96 KB

NVIDIA Volta Specs

Feature	Tesla V100 SXM2 16GB/32GB	Tesla V100 PCI-E 16GB/32GB
GPU Chip(s)	Volta GV100	
Integer Operations (INT8)*	62.8 TOPS	56.0 TOPS
Half Precision (FP16)*	125 TFLOPS	112 TFLOPS
Single Precision (FP32)*	15.7 TFLOPS	14.0 TFLOPS
Double Precision (FP64)*	7.8 TFLOPS	7.0 TFLOPS
On-die HBM2 Memory	16GB or 32GB	
Memory Bandwidth	900 GB/s	
L2 Cache	6 MB	
Interconnect	NVLink 2.0 + PCI-E 3.0	PCI-Express 3.0
Theoretical transfer bandwidth	150 GB/s	16 GB/s
# of SM Units	80	
# of single-precision FP32 CUDA Cores	5120	
# of double-precision FP64 CUDA Cores	2560	
GPU Boost Support	Yes – Dynamic	
GPU Boost Clock	1530 MHz	~1367 MHz

Memory

Generally GPUs have a separate memory
- Requires explicit data transfers, which complicates
- Typically expensive through PCI express

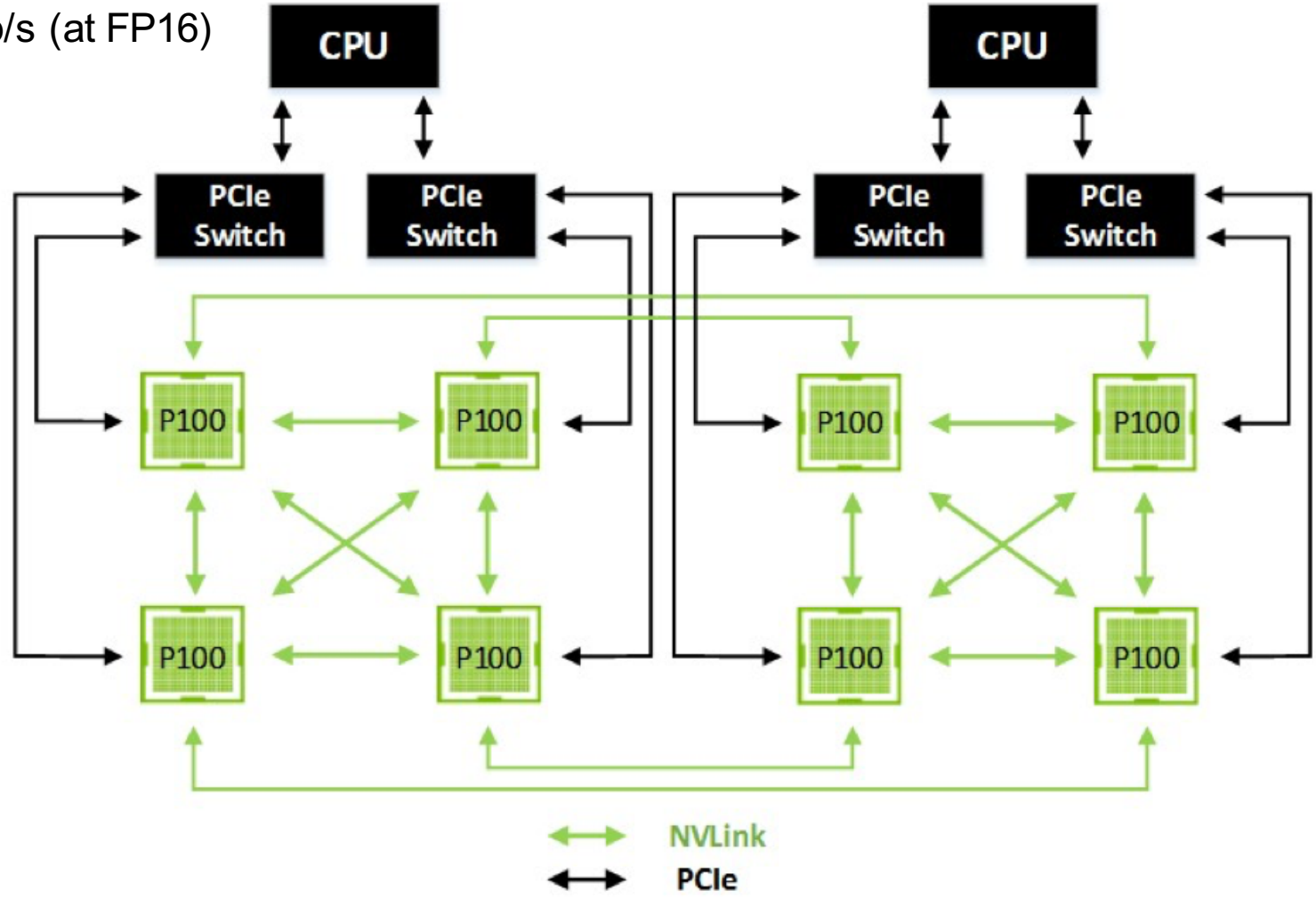
Newer GPU generations support a unified memory option
- Shared address space between CPU and GPU
- Allows direct access
- Note: not necessarily symmetric performance

Requires logical integration
- AMD Fusion approach integrates CPU and GPU in one package
- NVIDIA provided software version based on transparent paging
- Newer NVIDIA systems feature new HW link: NVLINK (1 and 2)

DGX Architecture

"Supercomputer in a Box"
Intended for Deep Learning

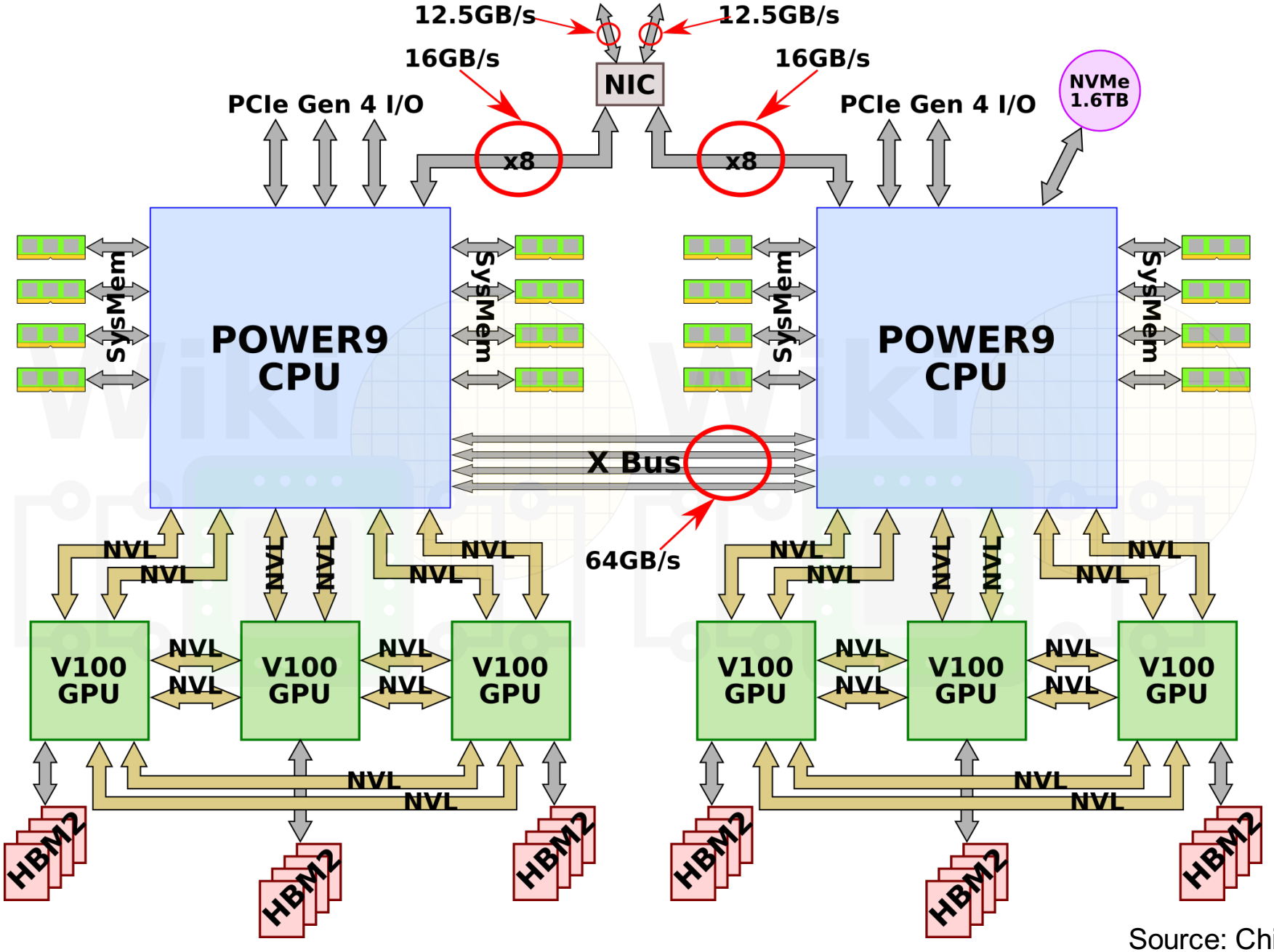
- 8 P100 GPUs with 16GB DRAM each
- Xeon host CPUs
- Peak at 170 TFlop/s (at FP16)



https://www.lrz.de/services/compute/special_systems/machine_learning/

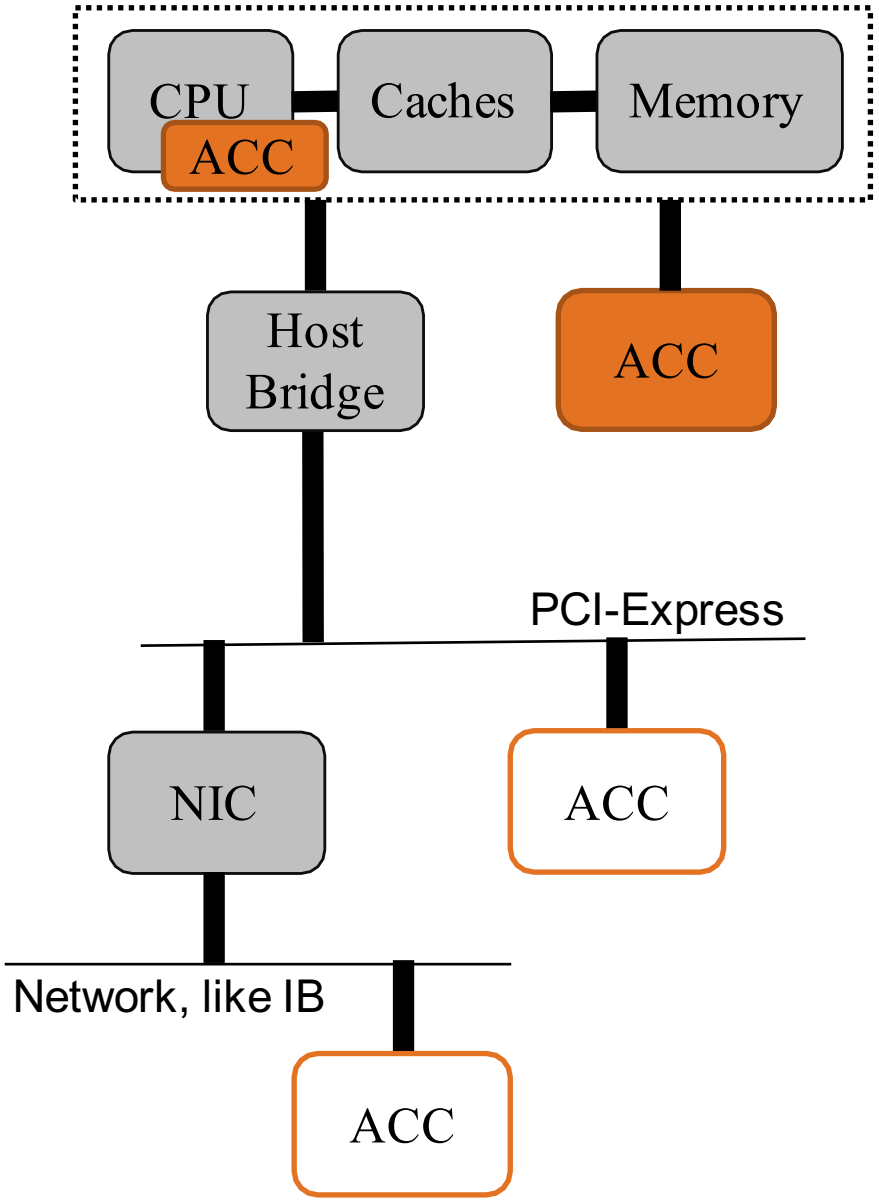
IBM Summit System @ ORNL

IBM Summit System @ ORNL



Source: ChipWiki

Connecting an Accelerator



Specialized hardware attached to a host

Most common (as of now): PCIe
- Add-on cards
- Access and data transfer via PCI bus
- Advantage: easy to integrate
- Disadvantage: PCI can be a bottleneck

Alternative: direct processor connection
- Needs new interfaces
- Removes bottlenecks

One step further: CPU integration
- Direct access from within CPU
- Either fully integrated or with packaged dies

External booster concept

Question: How to Program Accelerators

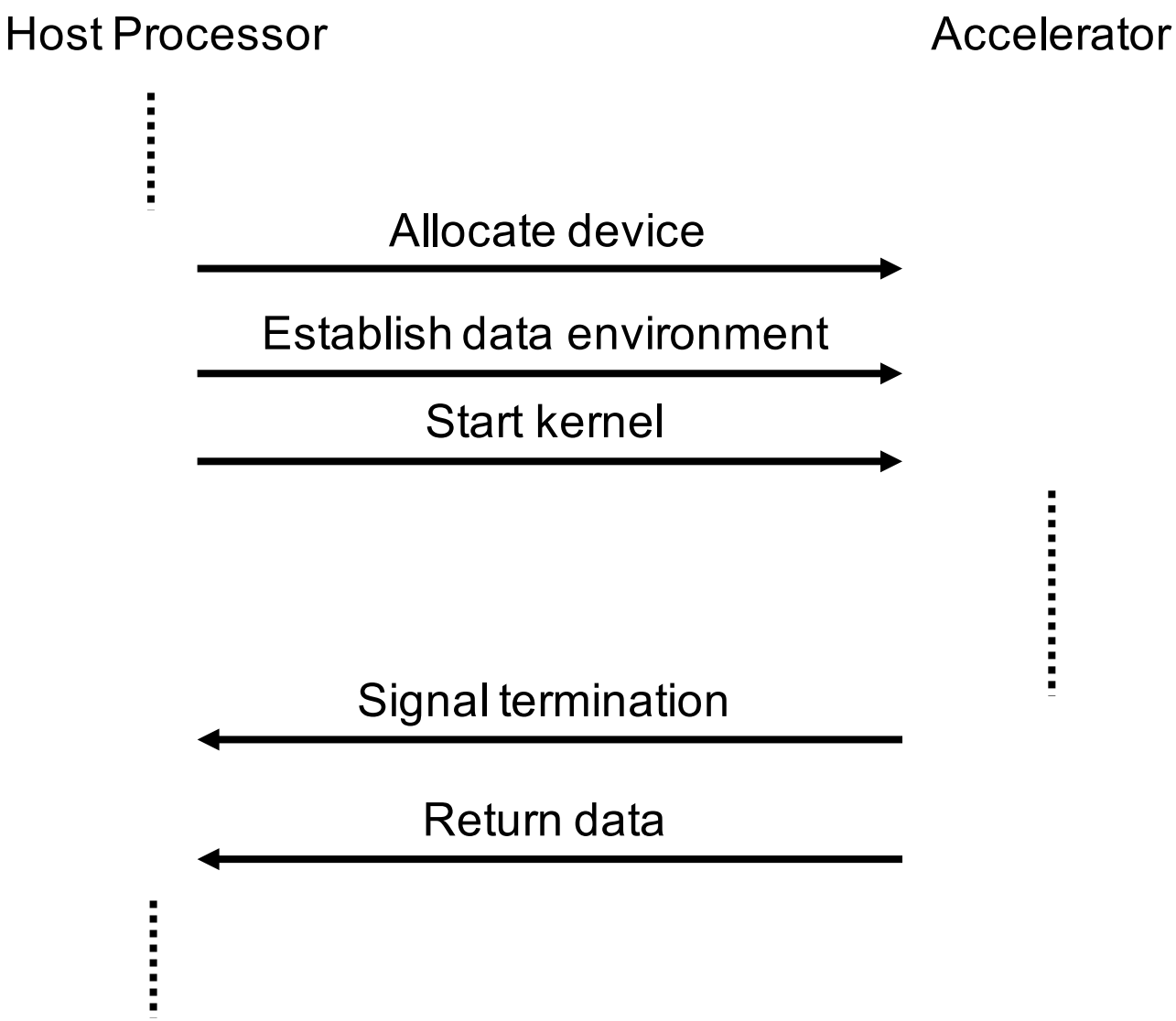
Differences to CPU programming

- Accelerators have no OS
- Computation in the form of kernels for the target architectures
- Requires a host program to run on host a system

Tasks

- Definition and programming of a kernel
- Setting up a data environment
- Dispatching a kernel
- Synchronization
- Collect results

Offloading of Kernels



Common Options

CUDA
- Targets NVIDIA GPUs
- Language extensions on top of C/C++
- Combined with runtime

OpenCL for any device
- Targets any kind of device
- Language extension on top of C/C++
- Combined with runtime

OpenMP
- Extensions in OpenMP since version 4.0
- General extension for any accelerator
- Accelerators are seens as “devices” and “targets”

OpenACC
- Specifically targets GPUs
- Grew out of OpenMP
- Focus on loop parallelism

Kernel Programming with NVIDIA‘s CUDA

Base concept: CUDA thread

- Used to define all forms of parallelism in the GPU

Thread Blocks: group of CUDA threads

- Executed by a single Streaming Multiprocessor
- As SIMD and multithreading execution mapped to Warps
- NVIDIA classifies CUDA as Single Instruction, Multiple Thread (SIMT)

Thread blocks are required to execute independently and in any order

- Different thread blocks cannot communicate directly
- Can be coordinated using atomic memory operations in the GPU memory

CUDA programs (host and kernel) are compiled with a separate compiler

- `nvcc`
- Includes language additions as well as access to CUDA runtime functions

Code and Data for the GPU

Functions can run on the host or the device (GPU)

Prefix for functions

- **`__global__`**
  - Kernels to be executed on the GPU
  - Kernels are configurable

- **`__device__`**
  - GPU functions that can be called from within kernels

- **`__host__`** (also default)
  - Functions for the CPU

Variables declared in __device__ or __global__ functions are allocated to the GPU memory accessible by all processors in the GPU.

Kernel Invocation

Function call syntax for the GPU

```
name<<<dimGrid, dimBlock>>>(… parameter list…)
```

With

`dimGrid`	number of thread blocks
`dimBlock`	number of threads per block

Allows control of concurrency

- Total concurrency is product of the two parameters
- Concrete numbers should be matched to archicture

Inside of a function the following variables are available for querying

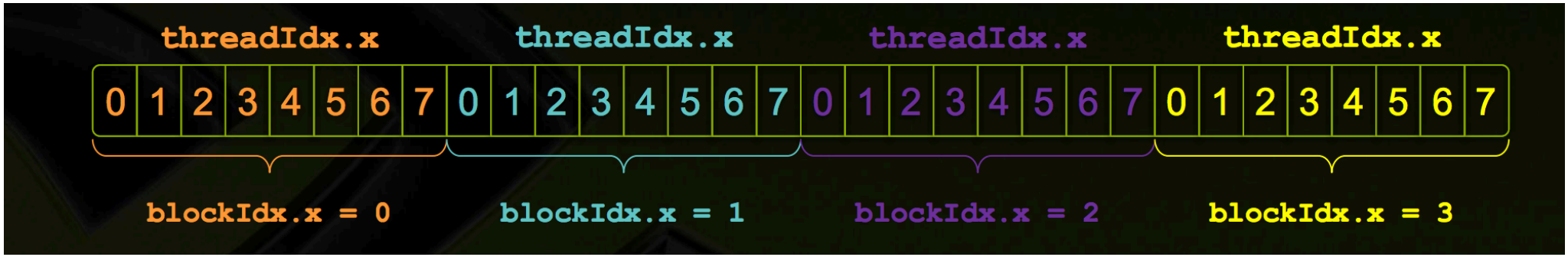
`blockIdx`	index of the thread block
`threadIdx`	id of a thread in a block

Example

```
//Invoke DAXPY
daxpy(n, 2.0, x, y)

//DAXPY in C
void daxpy(int n, double a, double *x, double *y){
   for (int i=0; i<n; ++i)
     y[i]=a*x[i]+y[i];
}
```

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



Example

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

```
//Copy data to device
cudaMalloc(&d_x, nbytes); cudaMalloc(&d_y, nbytes);
cudaMemcpy(d_x,h_x,nbytes,cudaMemocpyHostToDevice);
cudaMemcpy(d_y,h_y,nbytes,cudaMemocpyHostToDevice);

// invoke DAXPY with 256 threads per Thread Block
int nblocks = (n+255)/256;

daxpy<<<nblocks, 256>>>(n,2.0,d_x,d_y);

//Copy data from device
cudaMemocpy(h_y,d_y,nbytes,cudaMemcopyDevicetoHost);
```

Host Synchronization

All kernel launches are asynchronous

- Control returns to CPU immediately
- Kernel executes after all previous CUDA calls have completed

cudaMemcopy() is synchronous

- Control returns to CPU after copy completes
- Copy starts after all previous CUDA calls have completed

cudaThreadSynchronize()

- Blocks until all previous CUDA calls complete

CUDA Pros & Cons

CUDA enables straighforward programming of GPUs
- Direct mapping of computation to GPUs
- Exposes parallelism

Drawback 1
- Very low level
- Manual mapping and optimization
- Be aware of thread mapping and address alignments

Drawback 2
- Bound to one vendor (NVIDIA)

Drawback 3
- Want to use host system as well, which is also a parallel system
- Requires a separate programming system

Consequence: would like one, higher-level model that can cover both

OpenMP Target and Device Capabilities

Added in OpenMP 4.0, refined and extended since then

Host-centric execution model
- Offloading *target regions* to *target devices*
- *Target region* starts execution as a single thread on the device

Device is a generic concept
- Can be any kind of accelerator (different compilers will support different hardware)
- Can also be another CPU

Data environment on the device
- Directives define the data environment
  - Host variables called *original variables*
  - Device variables called *corresponding variables*
  - Host and device variables may share storage

- … define the mapping
  - To: copy from host to device
  - From: copy from device to host
  - FromTo: both

Target directive

```
#pragma omp target device(0)
        Structured-block
```

Creates a device data environment
Executes the target region (structured-block) on the specificed device

Synchronous
- The encountering threads waits until the device finished

Device clause
- Defines the target device

Target directive with Data Enviroment

```
#pragma omp target device(0),map(to:x,y)
        Structured-block
```

Variables referenced in target region
- Either are declared in the region
- Or have to be explicitly added to the data environment

`map` clause
- **`to, from, fromTo`**

`to`
- value copied to device at the beginning of the block

`from`
- value copied from device at the end of the block

Explicit Data Enviroment

```
#pragma omp target data device(0) map(to:x,y)
        Structured-block
```

Defines the data environment on a device
- Exists while the structure block is executed on the host

If (scalar-expression) clause
- false: The device is the host.

```
#pragma omp target upate device(0) map(to:x,y)
        Structured-block
```

Update corresponding or original variable
- from: GPU->CPU
- to: CPU->GPU

A list item can only appear in a *to* or *from* clause, but not both.

Declare Variables and Functions for Target

```
#pragma omp declare target
       block of definitions
#pragma omp end declare target
```

Specifies (static) variables and functions to be mapped to a device

Function:

- A device specific version is created by the compiler

Variable:

- Corresponding variable created in the initial device data environment
  - For all devices
  - Initialized variable: Value copied to the device.

Additional clauses available to map variables to specific devices

.

Example: daxpy

```
#pragma omp declare target
void daxpy(int n, double a, double *x, double *y){
   #pragma omp parallel for
   for (int i=0; i<n; ++i)
     y[i]=a*x[i]+y[i];
}
#pragma omp end declare target

...

#pragma omp target map(to:x,y),map(from:y)
daxpy(n,2.0,x,y);
```

Data transfer optimization

```
for (int i=0;i<m;i++){
  #pragma omp target map(to:x,y),map(from:y)
  daxpy(n,2.0,x,y);
}
```

```
#pragma omp target data map(to:x,y),map(from:y)
for (int i=0;i<m;i++){
  #pragma omp target
  daxpy(n,2.0,x,y);
}
```

Programming Challenges

OpenMP unifies host and accelerator programming in one model
- Still have to manage target device and host code separately
- Not as fine grained control as with CUDA

CUDA and OpenMP require careful memory management
- Data transfers are expensive
- Keep data on GPU as long as possible
  - Across kernel invocations
- Requires careful scheduling of kernels and data transfers

Higher level programming abstractions exist
- Can hide use of accelerators
- Often combined with automatic code generation
- Performance or generality is often problematic

Beyond the Single GPU or Accelerator

Goal: exploit the entire node
- Possibly multiple GPUs
- Host itself is typically a large NUMA system

Schedule work across all HW threads on CPU and GPU
- Neither CUDA nor OpenMP provide explicit support for this
- Manual chunking and load distribution
- Load balancing to compensate for heterogeneous performance

GPUs (Accelerators) in a larger HPC system
- Must cover multiple nodes by combining with MPI
  - MPI processes are CUDA or OpenMP with target/device codes
  - Additional complication for load balance
- Additional complication: #GPUs not equal #cores
  - Need to decide with MPI process handles a GPU
  - Recommendation is often: one MPI process for each GPU

Summary

Accelerators can help boost per node performance
- Dedicated hardware with specialized features
- Examples: FPGAs, Dataflow engines, GPUs

GPUs have evolved to massively parallel compute engines
- Hierarchical structure
- Inner elements are SIMT

Impact on programmability
- Accelerators rely on a host system
- Specialized language (extension) to cross-compile and control

CUDA targeting NVIDIA GPUs specifically
- Manual specification of data transfers through CUDA runtime
- Manual specification of thread mapping

OpenMP supports accelerators since OpenMP 4.0
- Allows specification of kernels on a device
- Pragmas to create the necessary data environment