HETEROGENEOUS **CPU**+**GPU** COMPUTING

Ana Lucia Varbanescu – University of Amsterdam

a.l.varbanescu@uva.nl

Significant contributions by: **Stijn Heldens** (U Twente), **Jie Shen** (NUDT, China), **Basilio Fraguela** (A Coruna University, ESP),

Today’s agenda

- Preliminaries

- Part I: Introduction to CPU+GPU heterogeneous computing
  - Performance promise vs. challenges
- Part II: Programing models
- Part III: Workload partitioning models
  - Static vs. Dynamic partitioning
- Part IV: Static partitioning and Glinda
- Part V: Tools for (programming) heterogeneous systems
  - Low-level to high-level

- Take home message

Goal

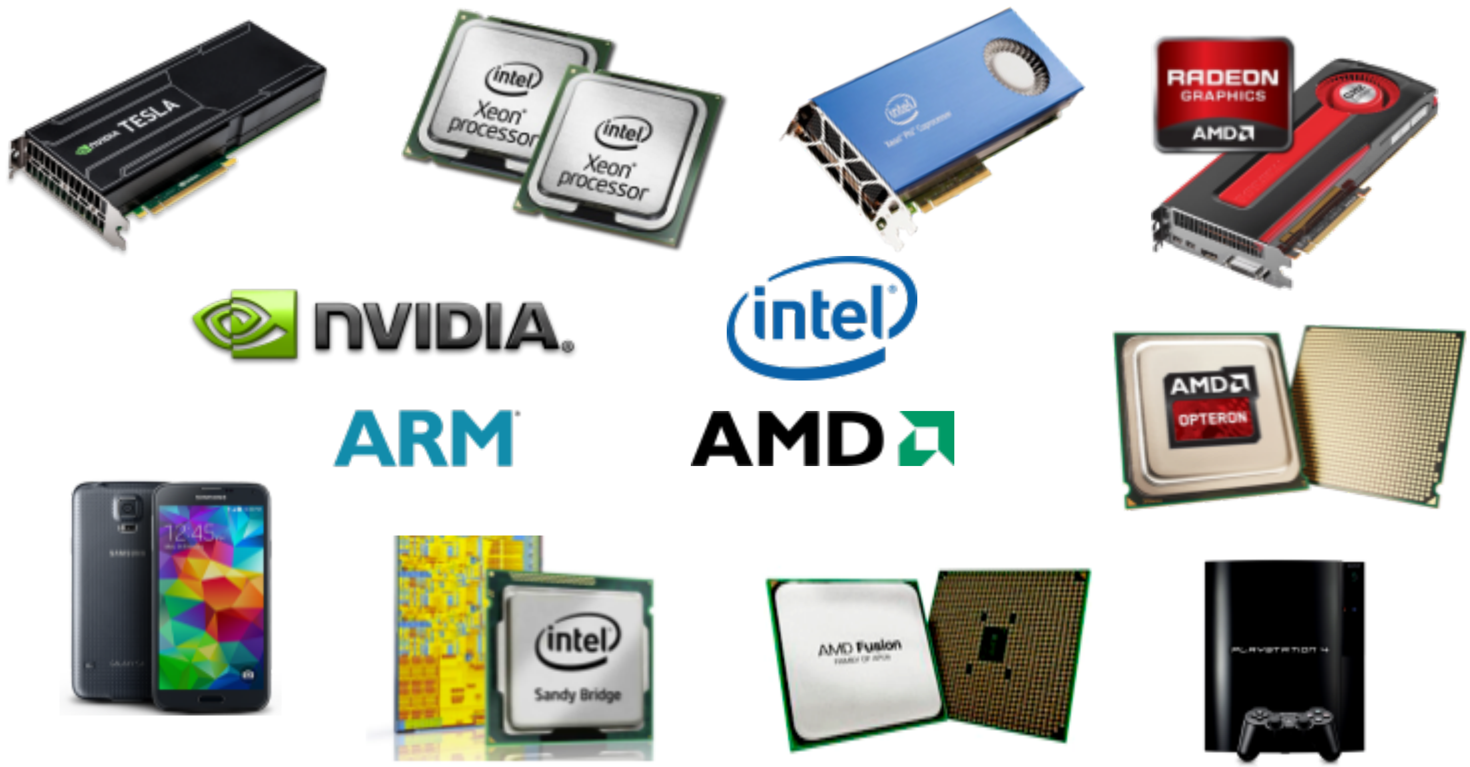
- Discuss heterogeneous computing as a promising solution for efficient resource utilization
  - And performance!
- Introduce methods for efficient heterogeneous computing
  - Programming
  - Partitioning
- Provide comparisons & selection criteria

- Current challenges and open research questions.

- Fair to others, but we advertise our research ☺

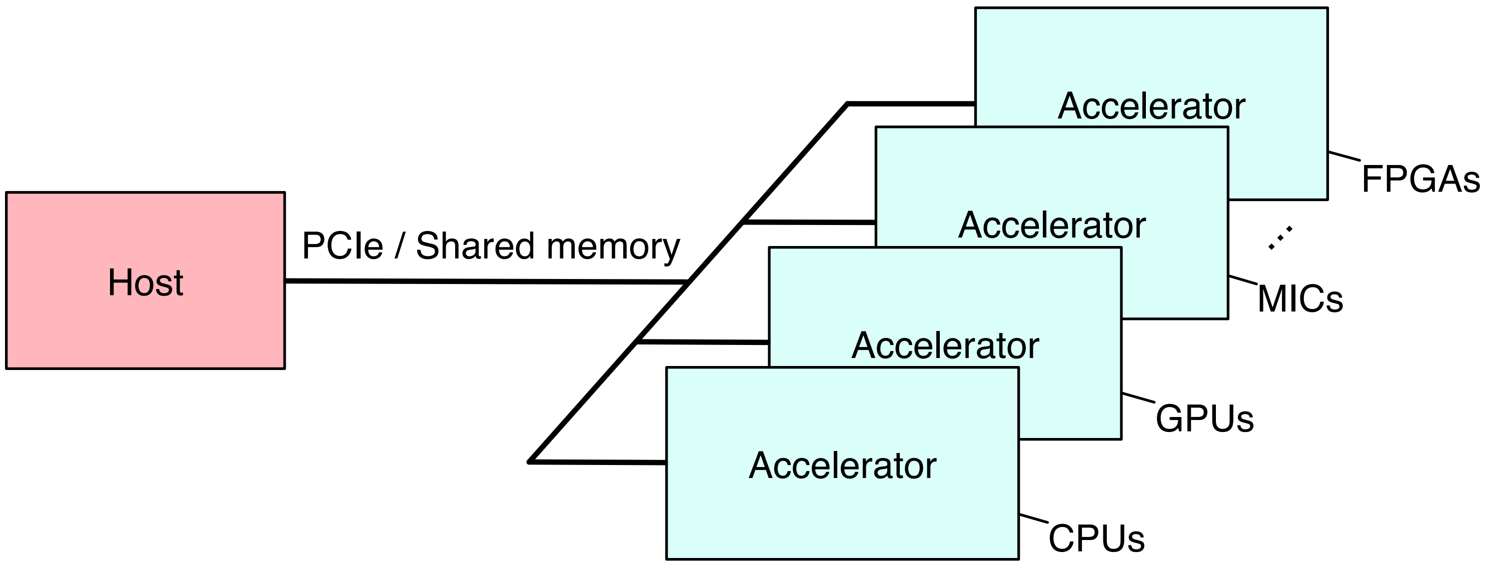
Heterogeneous platforms

- Systems combining main processors and accelerators
  - e.g., CPU + GPU, CPU + Intel MIC, AMD APU, ARM SoC
  - Everywhere from supercomputers to mobile devices



Heterogeneous platforms

- Host-accelerator hardware model



Heterogeneous platforms

- Top 500 (June 2015)

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom	1,572,864	17,173.2	20,132.7	7,890
4	RI[illegible] Co[illegible] Ja[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]0
5	[illegible] United States	[illegible] 1.60GHz, Custom IBM	[illegible]	[illegible]	[illegible]	[illegible]

195 cores/node

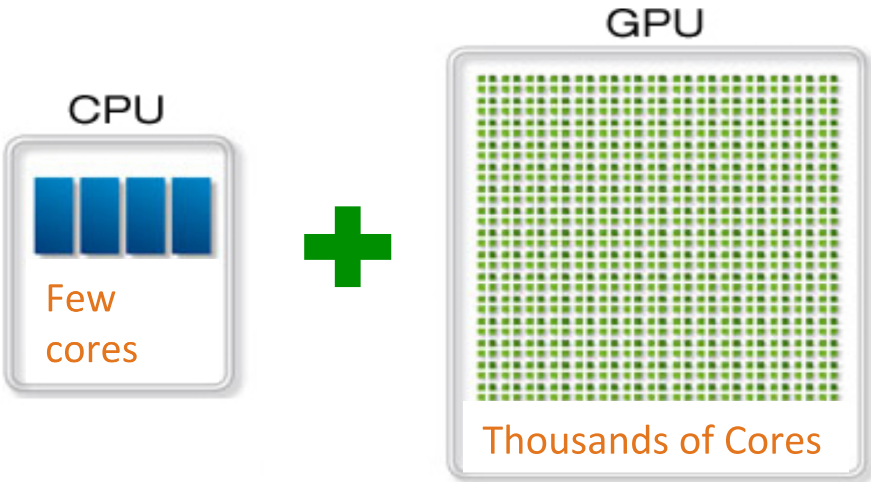
Accelerated!

Accelerated!

All systems are based on multi-cores.
90 systems have accelerators (18%).
Of those, 50% are NVIDIA GPUs, 30% are Intel MICs (Xeon Phi).

Our focus today …

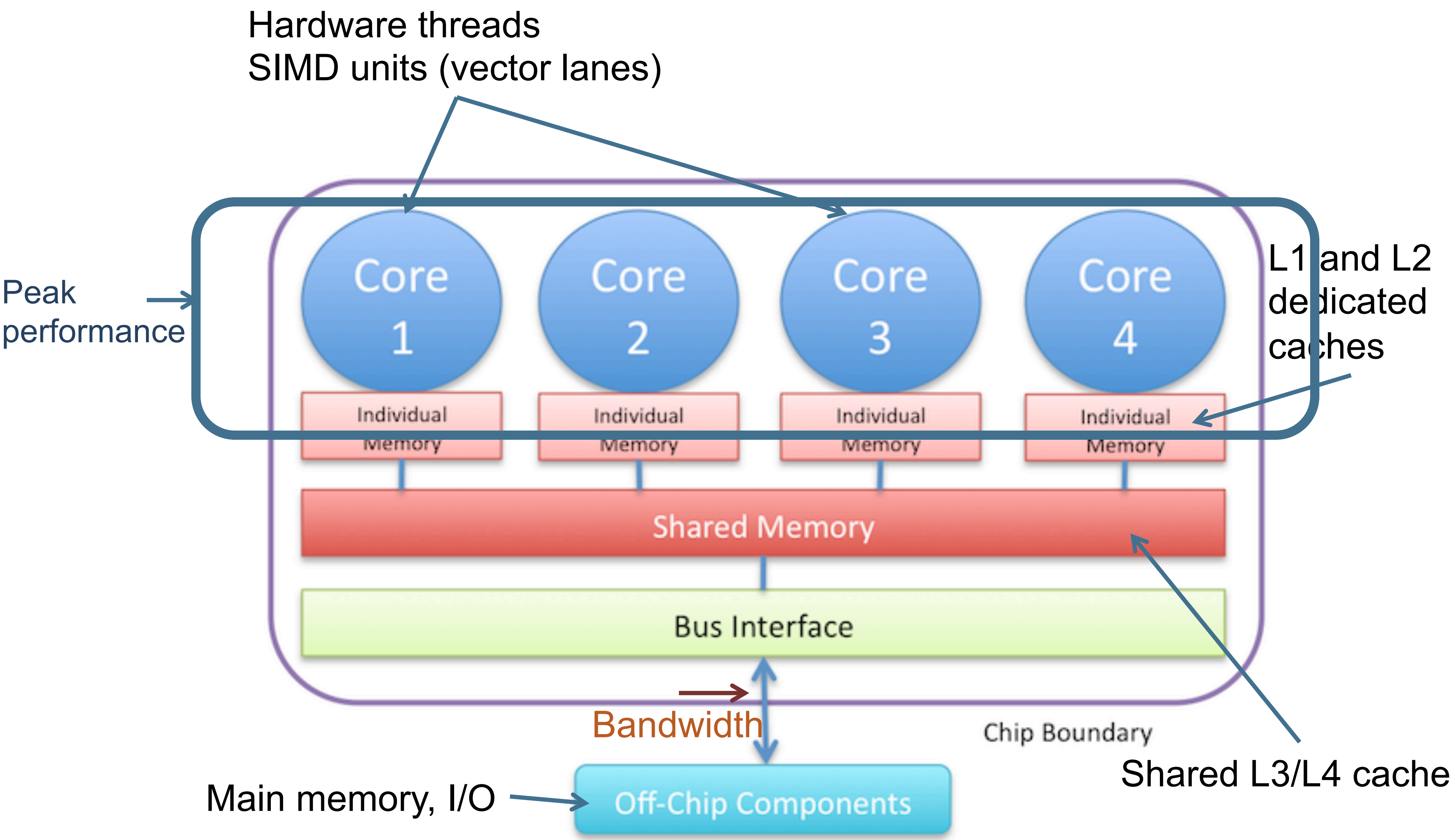
- A heterogeneous platform = **CPU** + **GPU**
  - Most solutions work for other/multiple accelerators
- An application workload = an application + its input dataset
- Workload partitioning = workload distribution among the processing units of a heterogeneous system

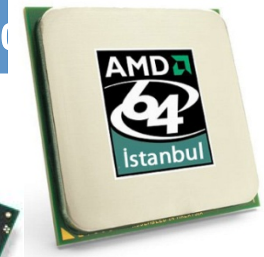


BEFORE WE START …

Basic knowledge about CPUs and GPUs

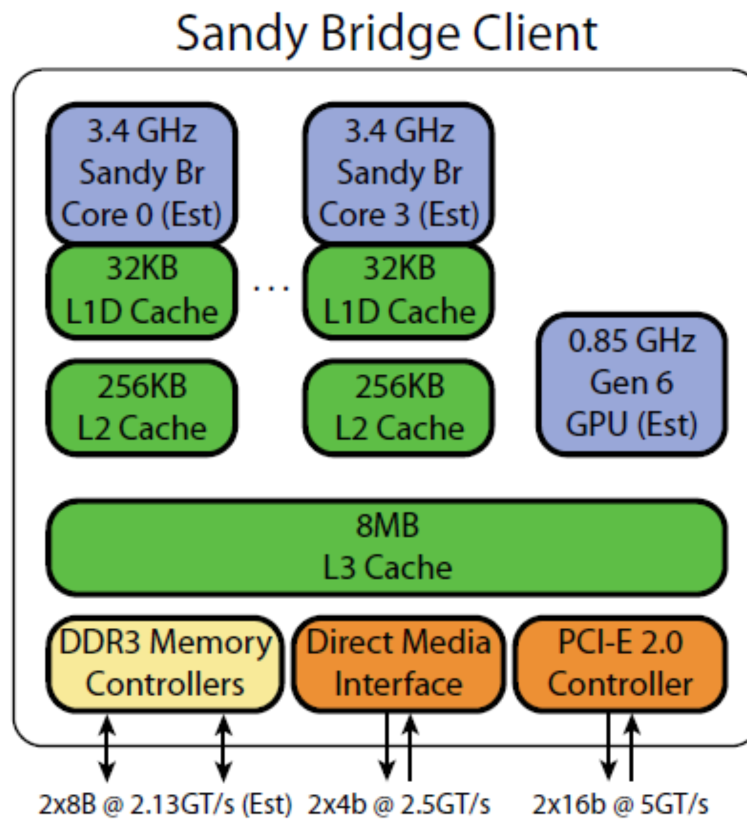
Generic multi-core CPU





Multi-core CPUs

- Architecture
  - Few large cores
  - (Integrated GPUs)
  - Vector units
    - Streaming SIMD Extensions (SSE)
    - Advanced Vector Extensions (AVX)
  - Stand-alone
- Memory
  - Shared, multi-layered
  - Per-core caches + shared caches
- Programming
  - Multi-threading
  - OS Scheduler



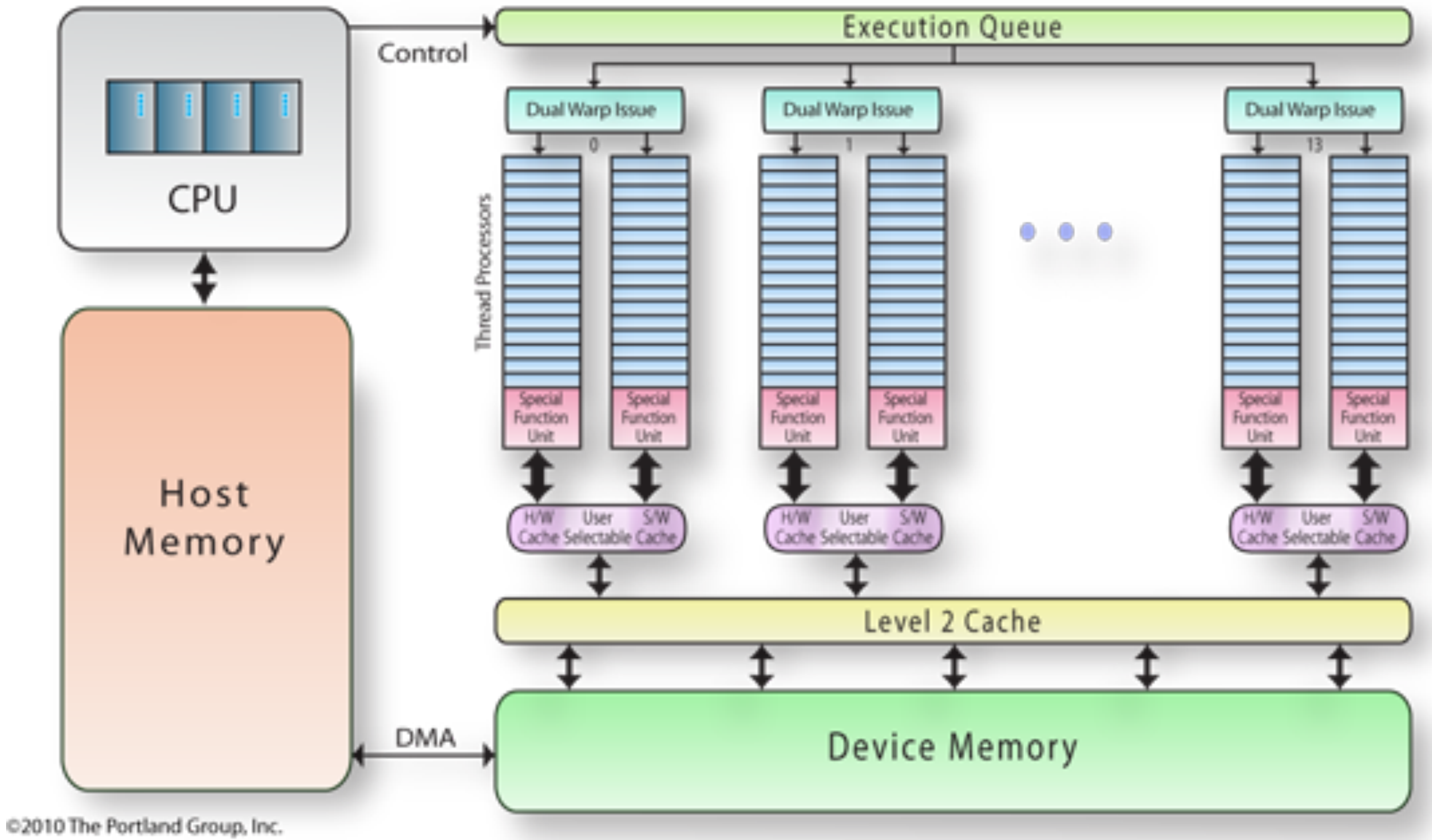
Parallelism

- Core-level parallelism ~ task/data parallelism (coarse)
  - 4-12 of powerful cores
    - Hardware hyperthreading (2x)
  - Local caches
  - Symmetrical or asymmetrical threading model
  - Implemented by programmer
- SIMD parallelism = data parallelism (fine)
  - 4-SP/2-DP floating point operations per second
    - 256-bit vectors
  - Run same instruction on different data
  - Sensitive to divergence
    - NOT the same instruction => performance loss
  - Implemented by programmer OR compiler

Programming models

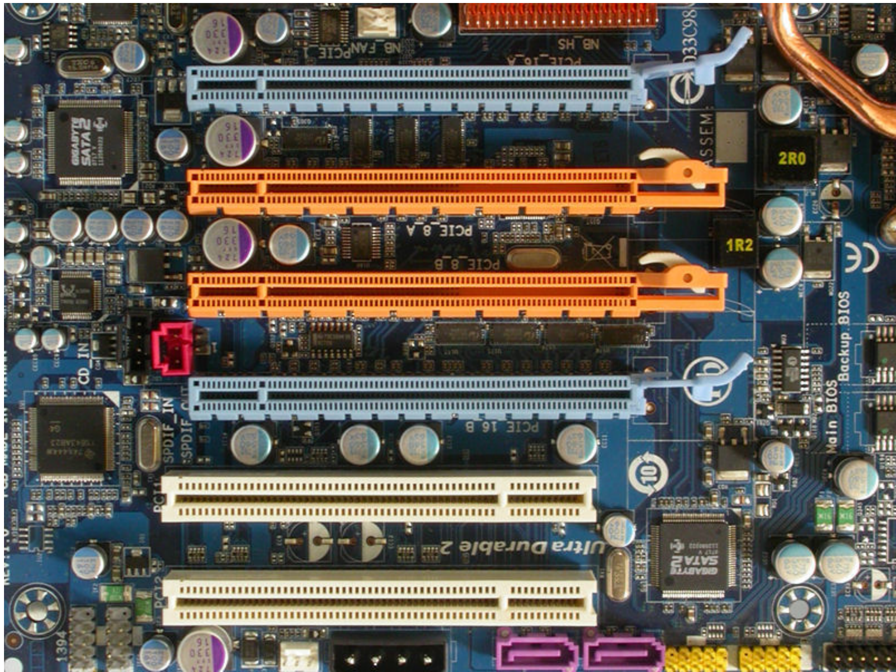
- Pthreads + intrinsics
- TBB – Thread building blocks
  - Threading library
- OpenCL
  - To be discussed …
- OpenMP
  - Traditional parallel library
  - High-level, pragma-based
- Cilk
  - Simple divide-and-conquer model

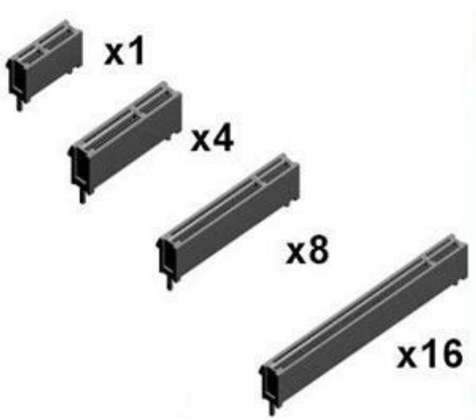
A GPU Architecture



Integration into host system

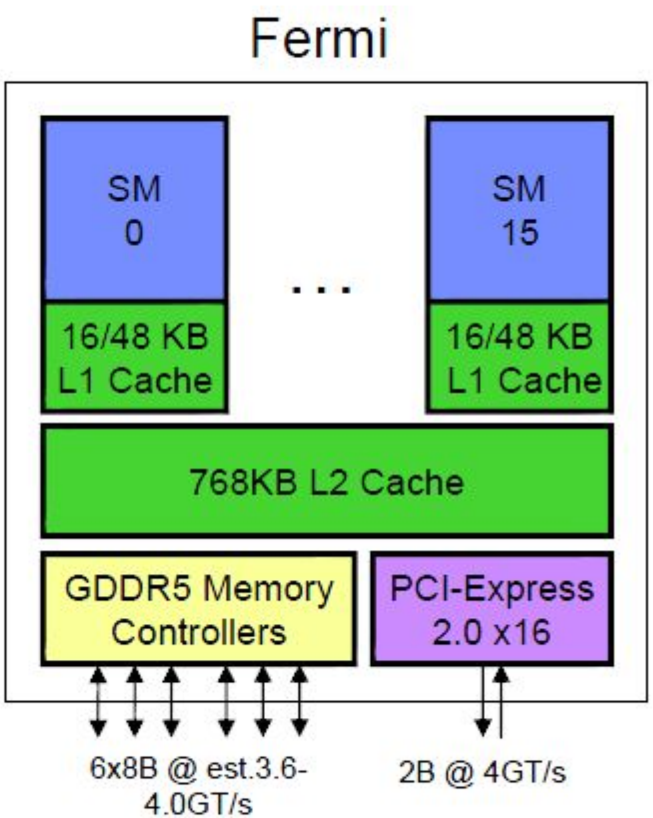
- Typically PCI Express 2.0
- Theoretical speed 8 GB/s
  - Effective ≤ 6 GB/s
  - In reality: 4 – 6 GB/s
- V3.0 recently available
  - Double bandwidth
  - Less protocol overhead





(NVIDIA) GPUs

- Architecture
  - Many (100s) slim cores
  - Sets of (32 or 192) cores grouped into “multiprocessors” with shared memory
    - SM(X) = stream multiprocessors
  - Work as accelerators
- Memory
  - Shared L2 cache
  - Per-core caches + shared caches
  - Off-chip global memory
- Programming
  - Symmetric multi-threading
  - Hardware scheduler



GPU Parallelism

- Data parallelism (fine-grain)
- **SIMT** (Single Instruction **Multiple Thread**) execution
  - Many threads execute concurrently
    - Same instruction
    - Different data elements
    - HW automatically handles divergence
  - Not same as SIMD because of multiple register sets, addresses, and flow paths*
- Hardware multithreading
  - HW resource allocation & thread scheduling
    - Excess of threads to hide latency
    - Context switching is (basically) free

*http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

Specific programming model: CUDA

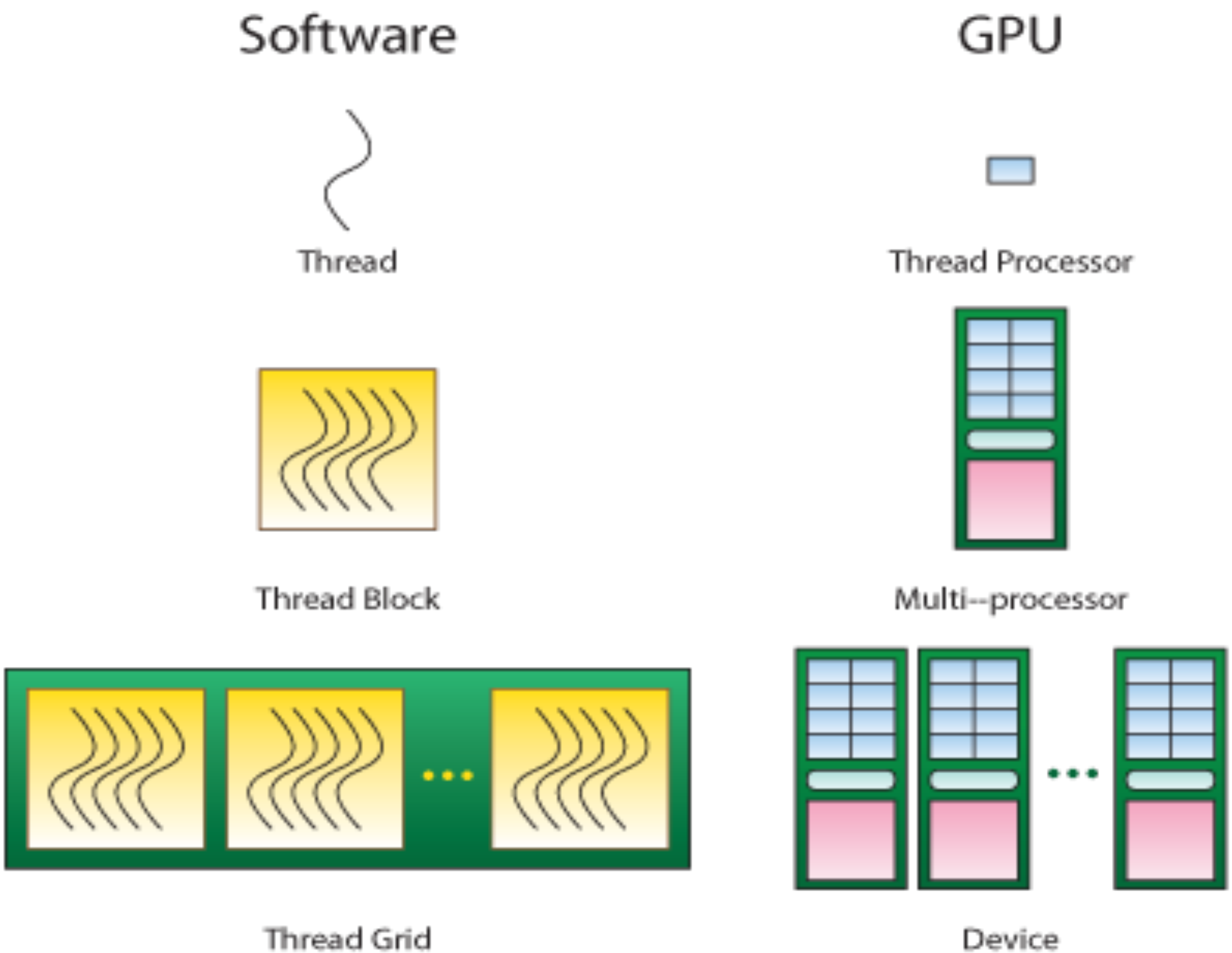
- CUDA: Compute Unified Device Architecture
  - C/C++ extensions
    - Other wrappers exist
- Straightforward mapping onto hardware
  - Hierarchy of threads (map to cores)
    - Configurable at logical level
  - Various memory spaces (map to physical mem. spaces)
    - Usable via variable scopes
- SIMT: single instruction multiple threads
  - Have 1000s threads running concurrently
  - Hardware multi-threading
    - GPU threads are lightweight

CUDA: Hierarchy of threads

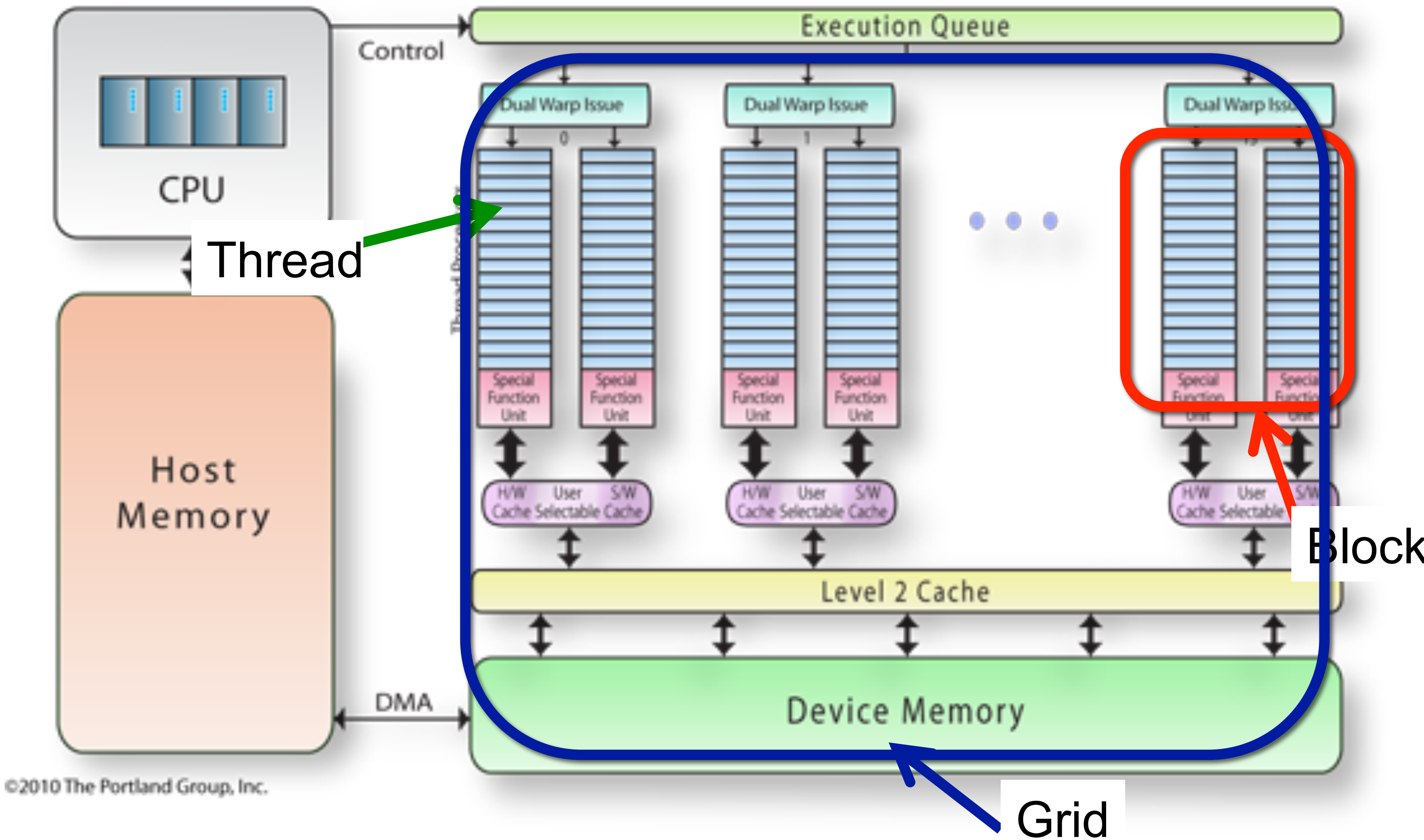
- Each thread executes the kernel code
  - One thread runs on one CUDA core
- Threads are logically grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot cooperate
- All thread blocks are logically organized in a Grid
  - 1D or 2D or 3D
  - Threads and blocks have unique IDs

- A grid specifies in how many instances the kernel is being run

CUDA Model of Parallelism



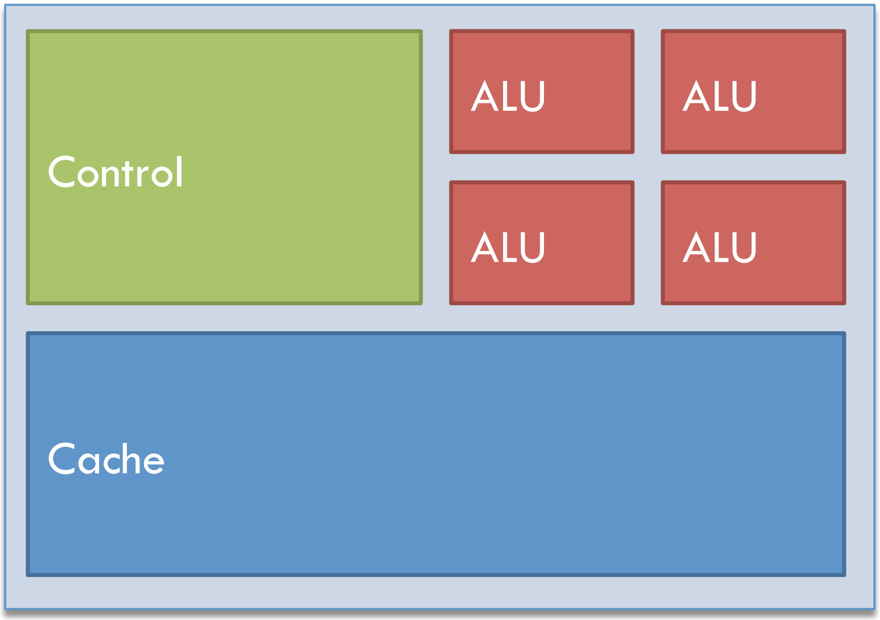
Hierarchy of threads



CUDA Model of Parallelism

- CUDA virtualizes the physical hardware
  - A block is a virtualized streaming multiprocessor
    - threads, shared memory
  - A thread is a virtualized scalar processor
    - registers, PC, state
- Execution model:
  - Threads execute in warps (32 threads per warp)
    - Called "wavefronts" by AMD (64 threads)
  - All threads in a warp execute the same code
    - On different data
  - Blocks = multiple warps
    - Scheduled independently on the same SM

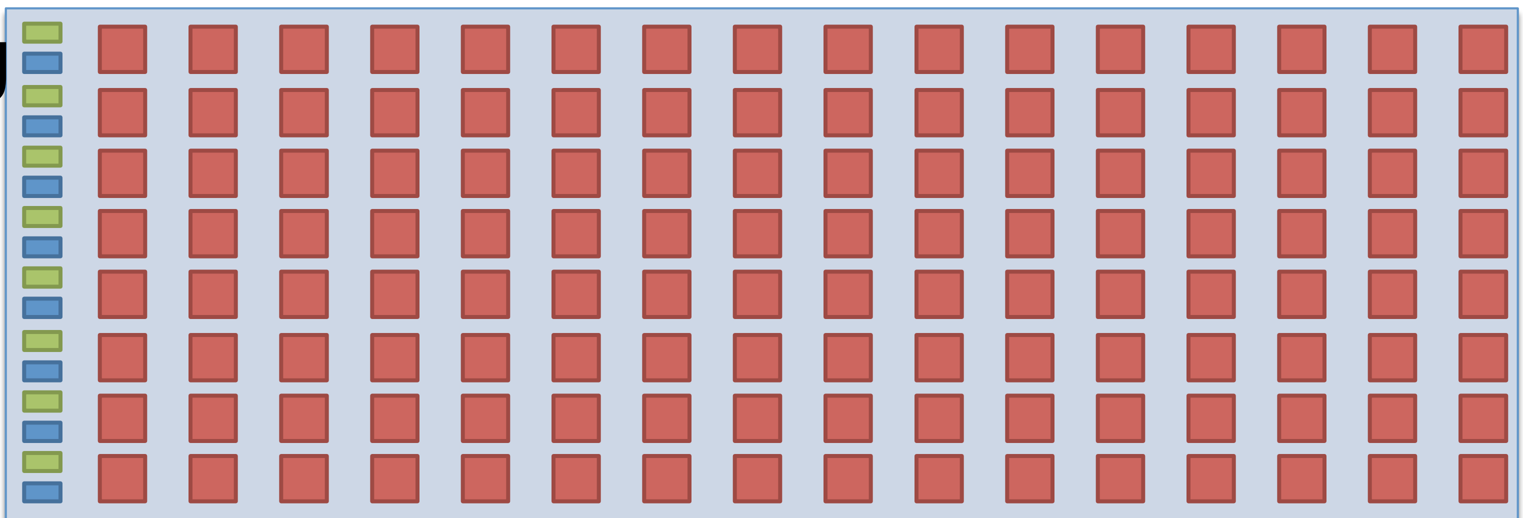
CPU vs. GPU



CPU

Low latency, high flexibility.
Excellent for irregular codes with limited parallelism.

GPU



High throughput.
Excellent for massively parallel workloads.

PART I

Heterogeneous processing: pro’s and con’s

Hardware Performance metrics

- Clock frequency [GHz] = absolute hardware speed
  - Memories, CPUs, interconnects
- Operational speed [GFLOPs]
  - Instructions per cycle + frequency
- Memory bandwidth [GB/s]
  - differs a lot between different memories on chip
- Power [Watt]
- Derived metrics
  - FLOP/Byte, FLOP/Watt

Theoretical peak performance

Peak = chips * cores * threads/core * vector_lanes * FLOPs/cycle * clockFrequency

- Some examples:
  - Intel Core i7 CPU

    2 chips * 4 cores * 4-way vectors * 2 FLOPs/cycle * 2.4 GHz = **154 GFLOPs**
  - NVIDIA GTX 580 GPU

    1 chip * 16 SMs * 32 cores * 2 FLOPs/cycle * 1.544 GhZ = **1581 GFLOPs**

Performance ratio (CPU:GPU): 1:10 !!!

DRAM Memory bandwidth

Bandwidth = memory bus frequency * bits per cycle
* bus width

- Memory clock != CPU clock!
- In bits, divide by 8 for GB/s

- Some Examples:
  - Intel Core i7 DDR3: 1.333 * 2 * 64 = **21 GB/s**
  - NVIDIA GTX 580 GDDR5: 1.002 * 4 * 384 = **192 GB/s**

Performance ratio (CPU:GPU): 1:8 !!!

Power

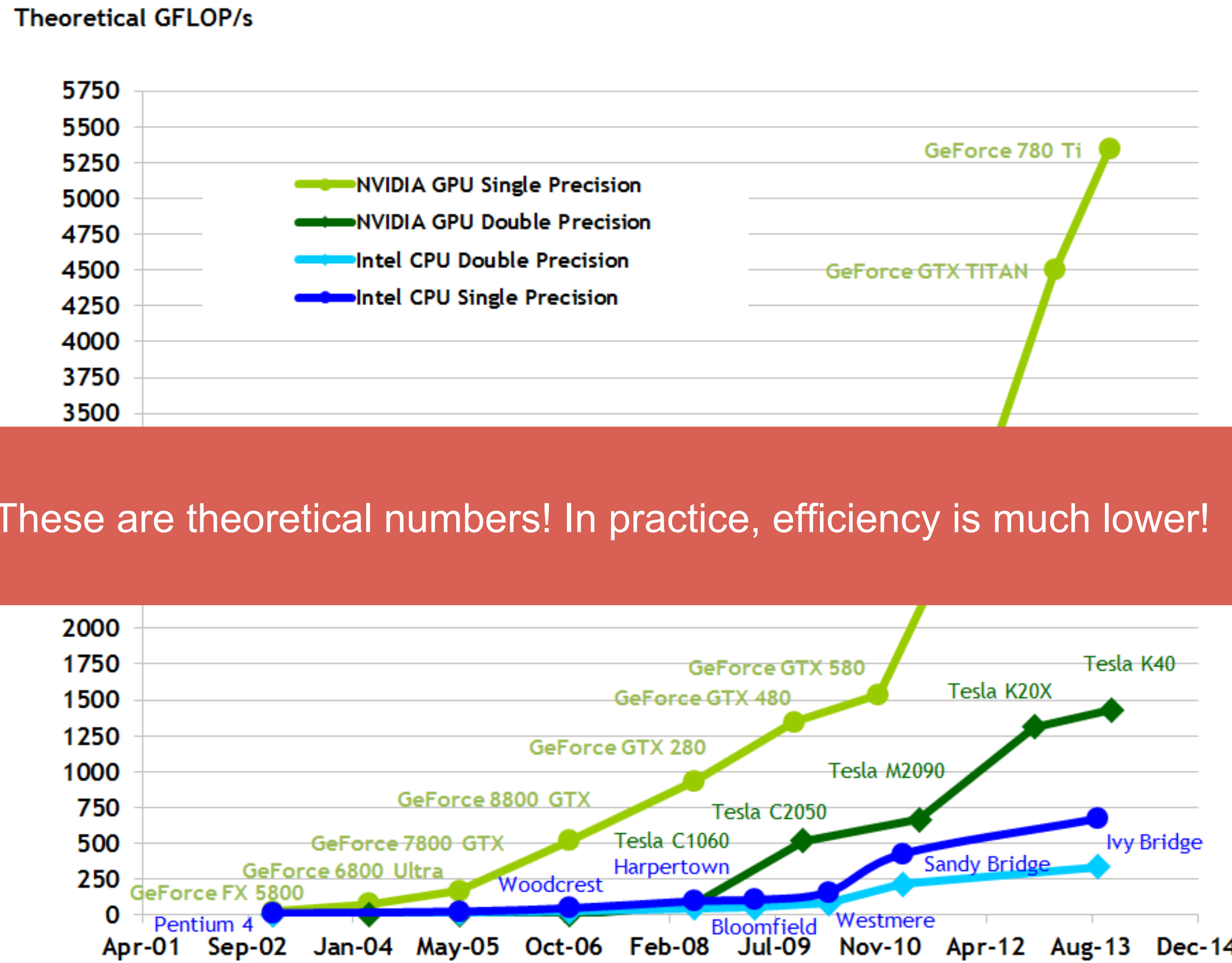
- Chip manufactures specify Thermal Design Power (TDP)
- We can measure dissipated power
  - Whole system
  - Typically (much) lower than TDP
- Power efficiency
  - FLOPS / Watt
- Examples (with theoretical peak and TDP)
  - Intel Core i7: 154 / 160 = **1.0 GFLOPs/W**
  - NVIDIA GTX 580: 1581 / 244 = **6.3 GFLOPs/W**
  - ATI HD 6970: 2703 / 250 = **10.8 GFLOPs/W**

Summary

	Cores	Threads/ALUs	**GFLOPS**	**Bandwidth**
Sun Niagara 2	8	64	**11.2**	**76**
IBM BG/P	4	8	**13.6**	**13.6**
IBM Power 7	8	32	**265**	**68**
Intel Core i7	4	16	**85**	**25.6**
AMD Barcelona	4	8	**37**	**21.4**
AMD Istanbul	6	6	**62.4**	**25.6**
AMD Magny-Cours	12	12	**125**	**25.6**
Cell/B.E.	8	8	**205**	**25.6**
NVIDIA GTX 580	16	512	**1581**	**192**
NVIDIA GTX 680	8	1536	**3090**	**192**
AMD HD 6970	384	1536	**2703**	**176**
AMD HD 7970	32	2048	**3789**	**264**
Intel Xeon Phi 7120	61	240	**2417**	**352**

GPU vs. CPU performance

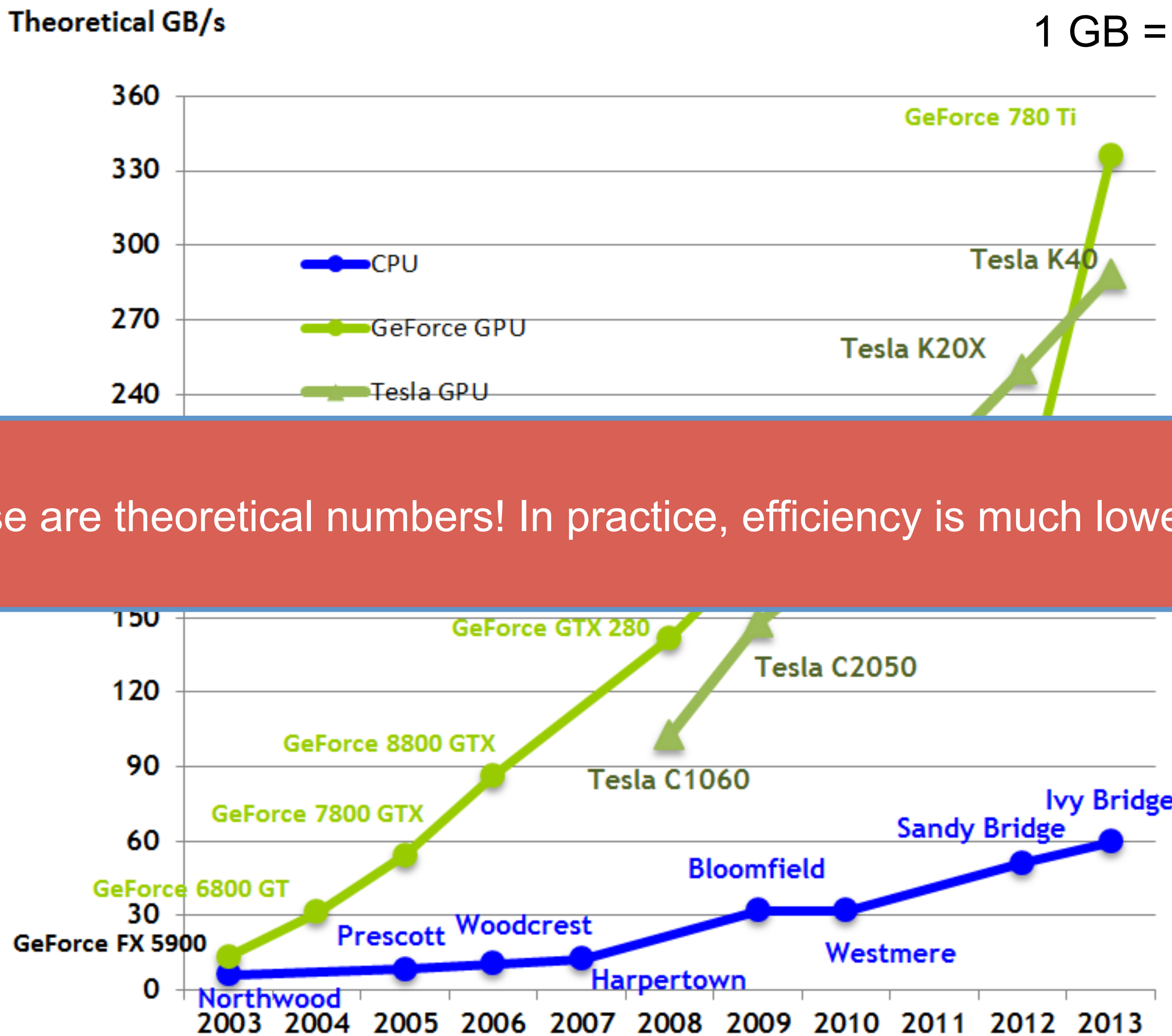
1 GFLOP = 10^9 ops



These are theoretical numbers! In practice, efficiency is much lower!

GPU vs. CPU performance

1 GB = 8 x10^9 bits



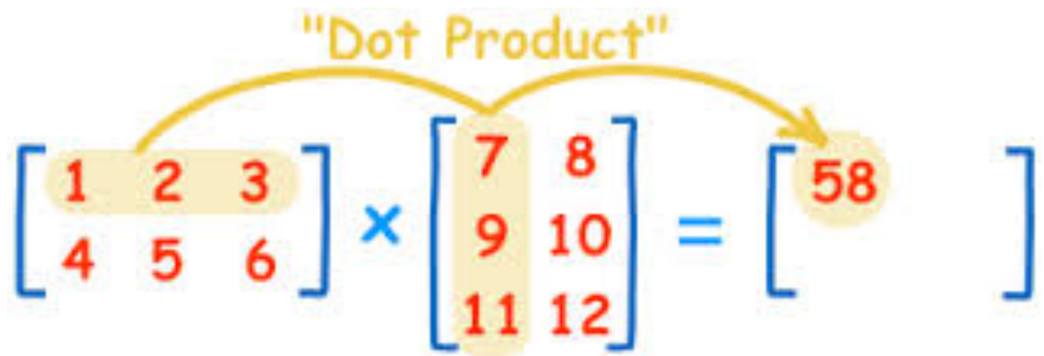
These are theoretical numbers! In practice, efficiency is much lower!

Heterogeneity vs. Homogeneity

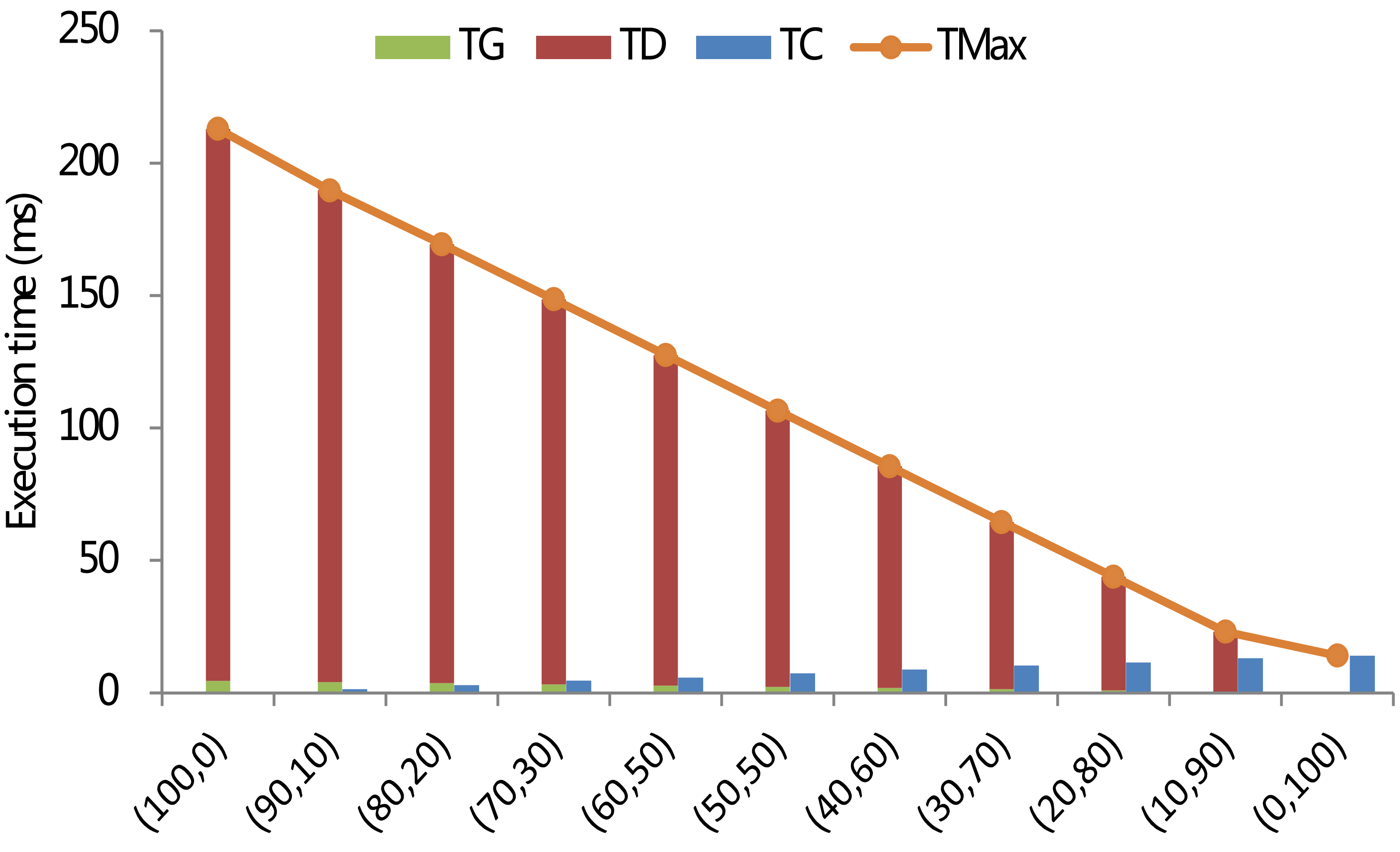
- Increase performance
  - Both devices work in parallel
    - Gain is much more than 10%
  - Decrease data communication
    - Which is often the bottleneck of the system
  - Different devices for different roles
- Increase flexibility and reliability
  - Choose one/all *PUs for execution
  - Fall-back solution when one *PU fails
- Increase power efficiency
- Cheaper per flop

Example 1: dot product

- Dot product
  - Compute the dot product of 2 (1D) arrays
- Performance
  - T_G = execution time on GPU
  - T_C = execution time on CPU
  - T_D = data transfer time CPU-GPU
- GPU best or CPU best?

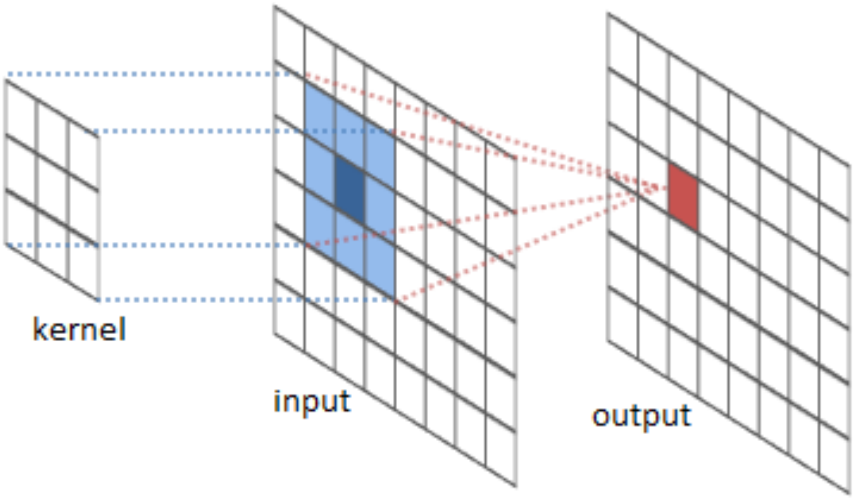


Example 1: dot product

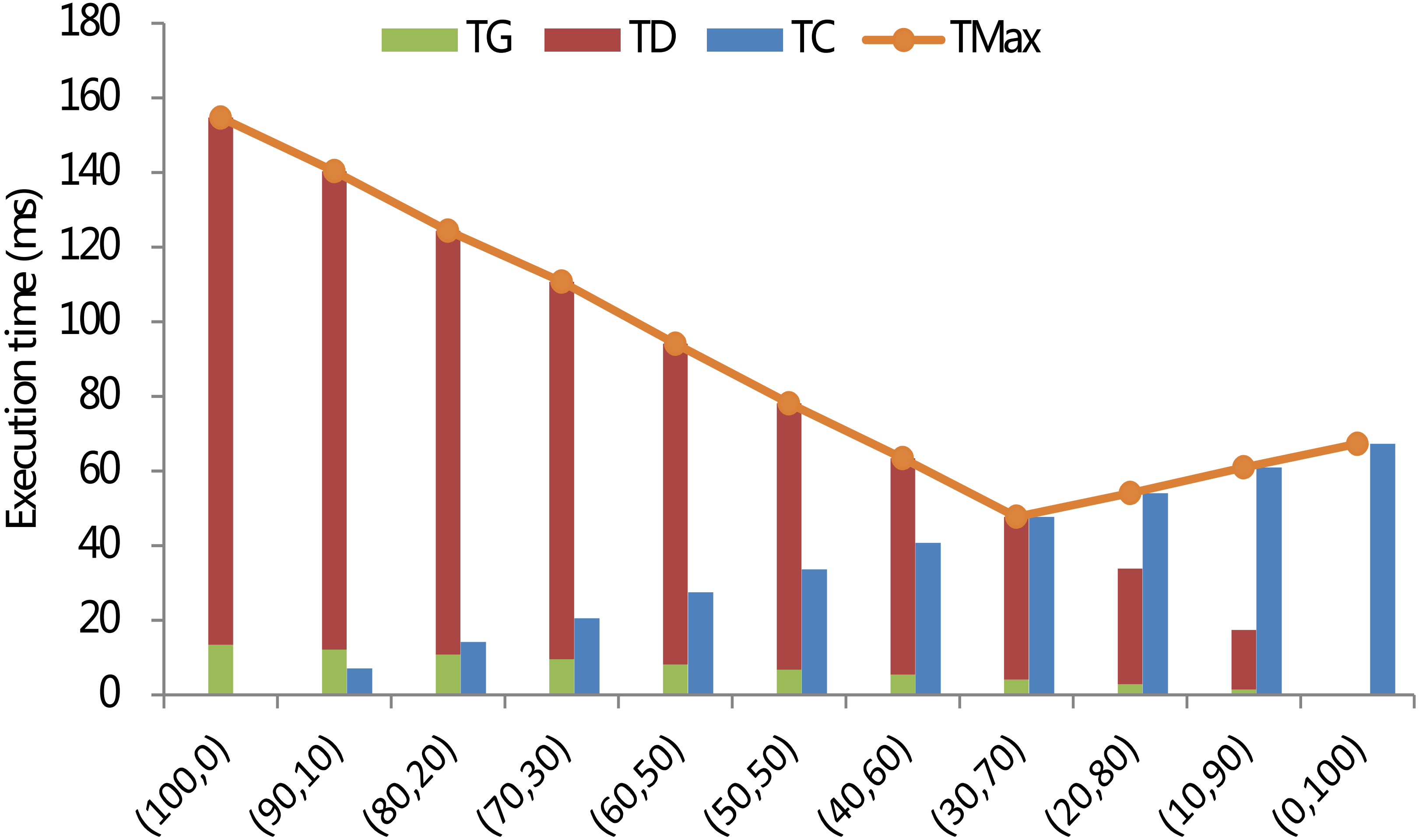


Example 2: separable convolution

- Separable convolution (CUDA SDK)
  - Apply a convolution filter (kernel) on a large image.
  - Separable kernel allows applying
    - Horizontal first
    - Vertical second
- Performance
  - T_G = execution time on GPU
  - T_C = execution time on CPU
  - T_D = data transfer time
- GPU best or CPU best?

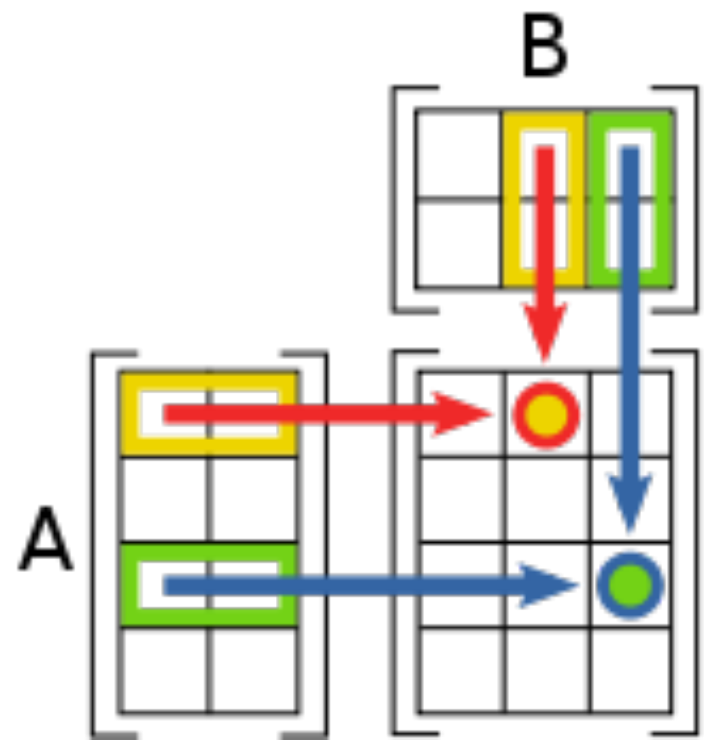


Example 2: separable convolution

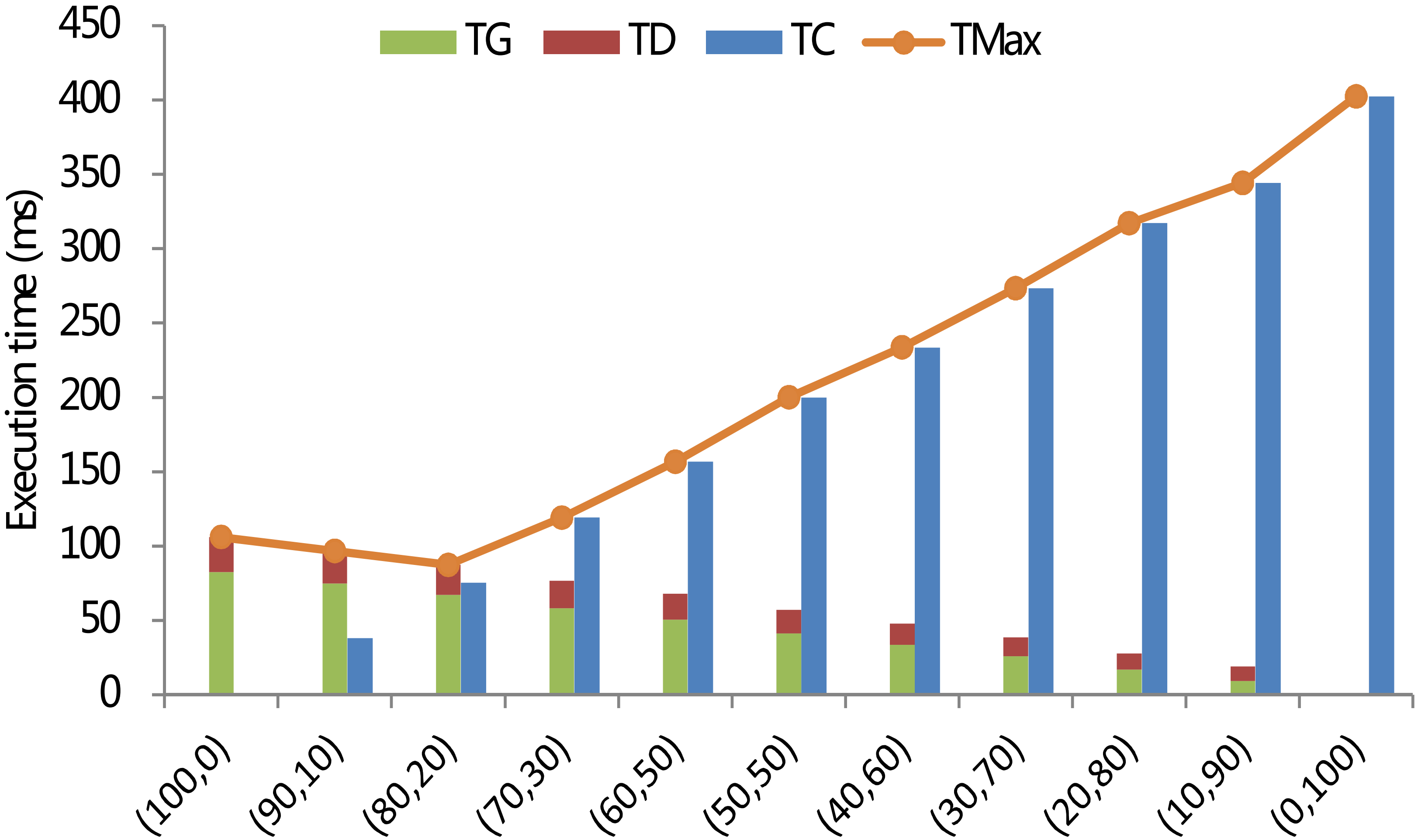


Example 3: matrix multiply

- Matrix multiply
  - Compute the product of 2 matrices
- Performance
  - T_G = execution time on GPU
  - T_C = execution time on CPU
  - T_D = data transfer time CPU-GPU
- GPU best or CPU best?

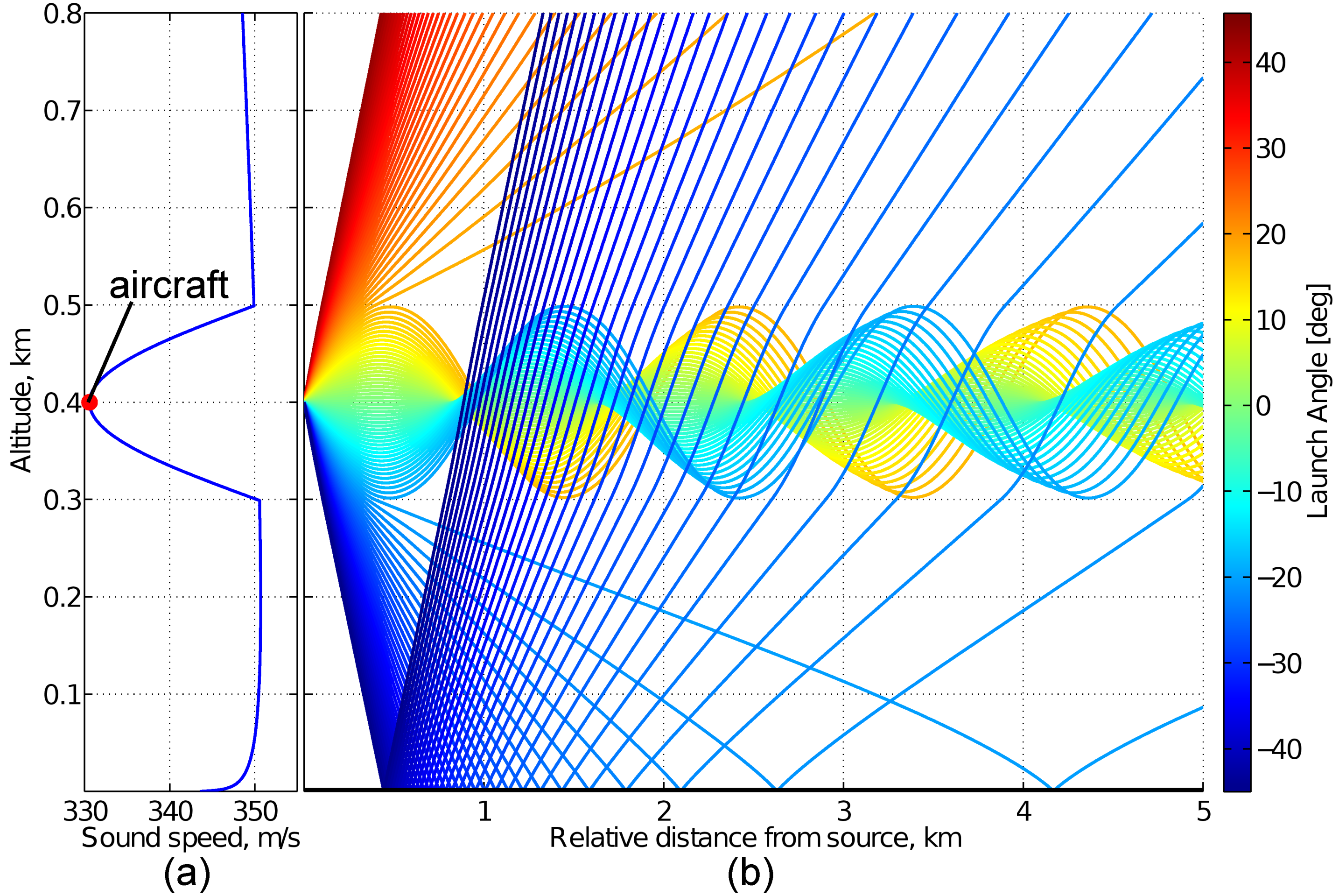


Example 3: matrix multiply



Example 4: Sound ray tracing

Example 4: Sound ray tracing

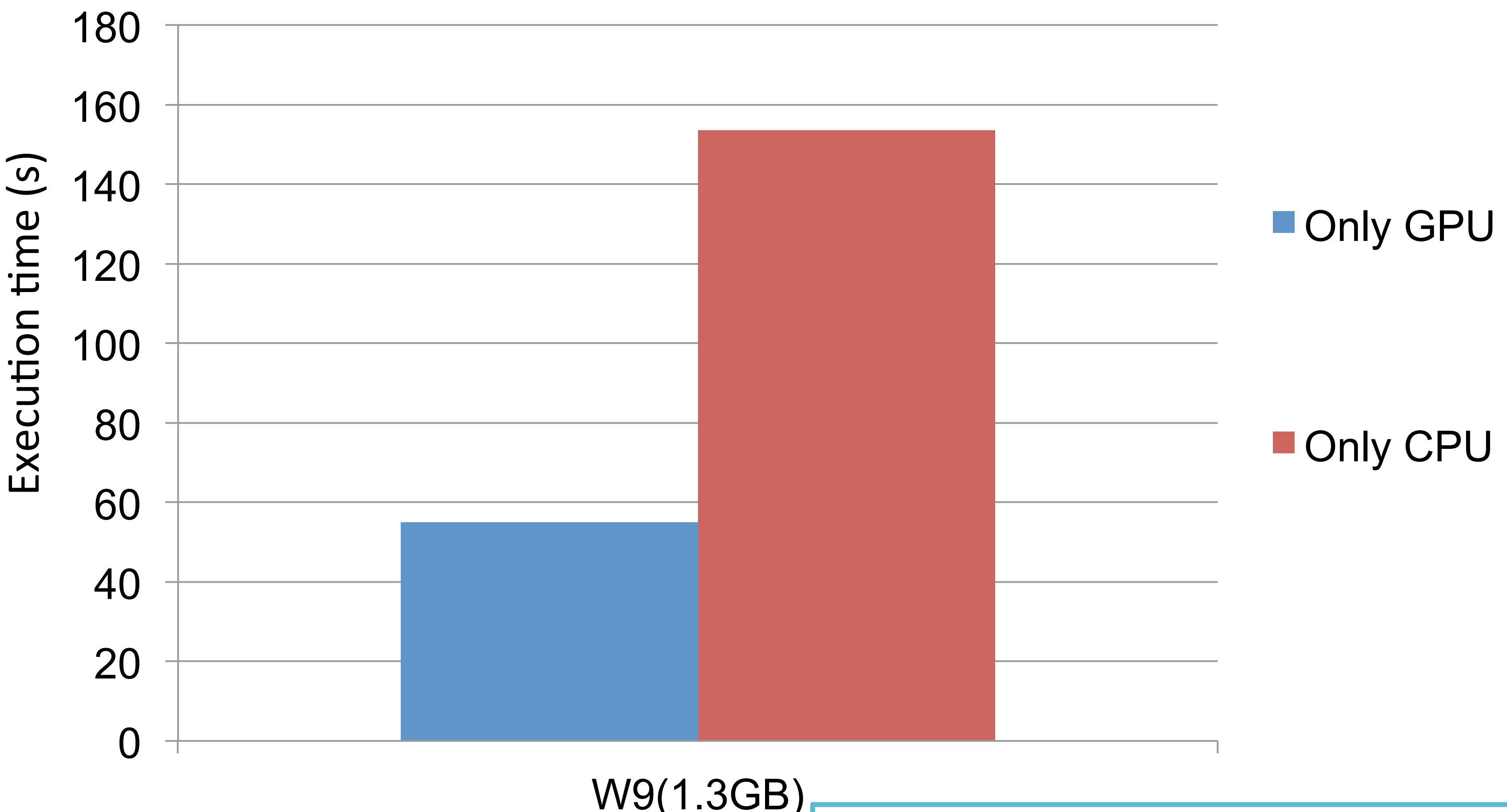


Which hardware?

- Our application has …
- Massive data-parallelism …
- No data dependency between rays …
- Compute-intensive per ray …

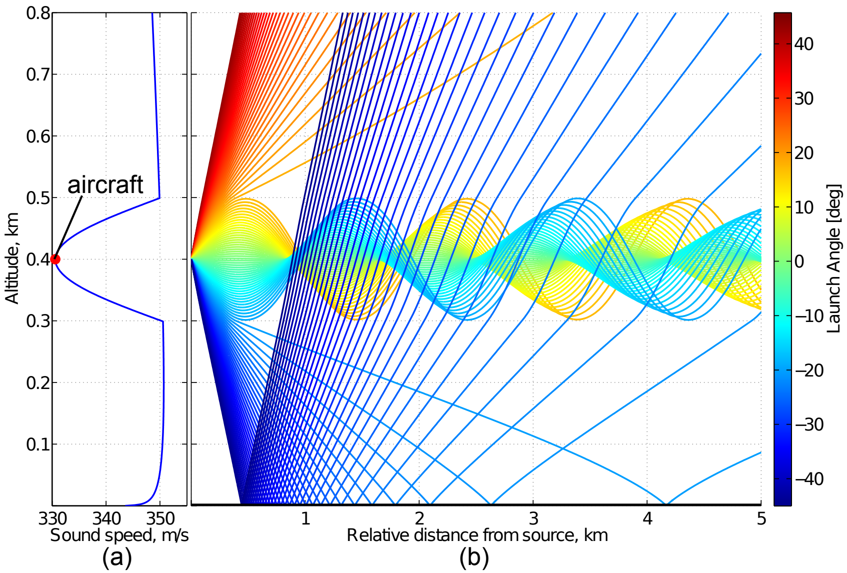
- … clearly, this is a perfect GPU workload !!!

Results [1]

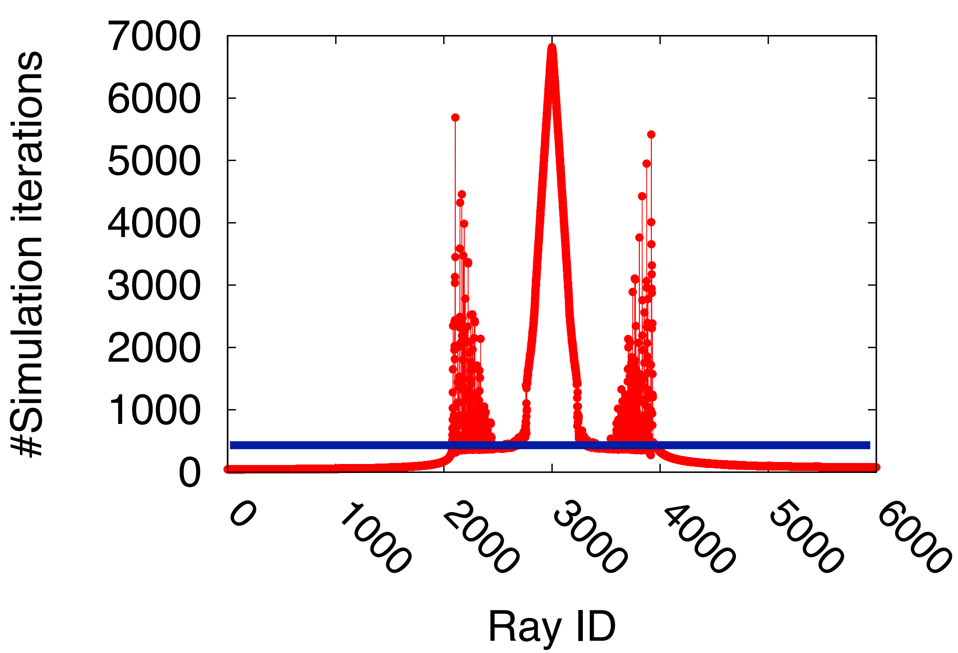


Only 2.2x performance improvement!
We expected 100x …

Workload profile

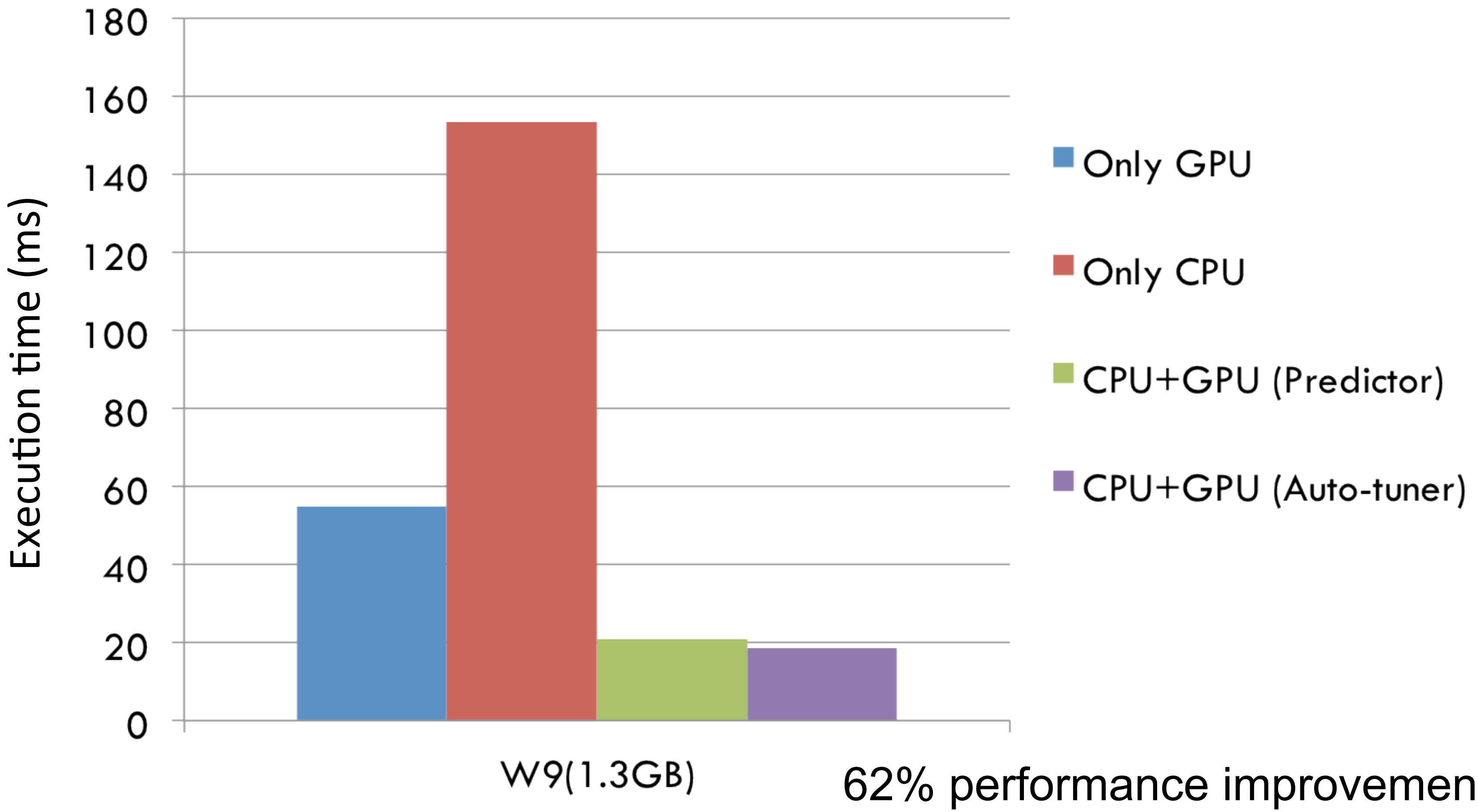


Peak
Processing iterations: ~7000



Bottom
Processing iterations: ~500

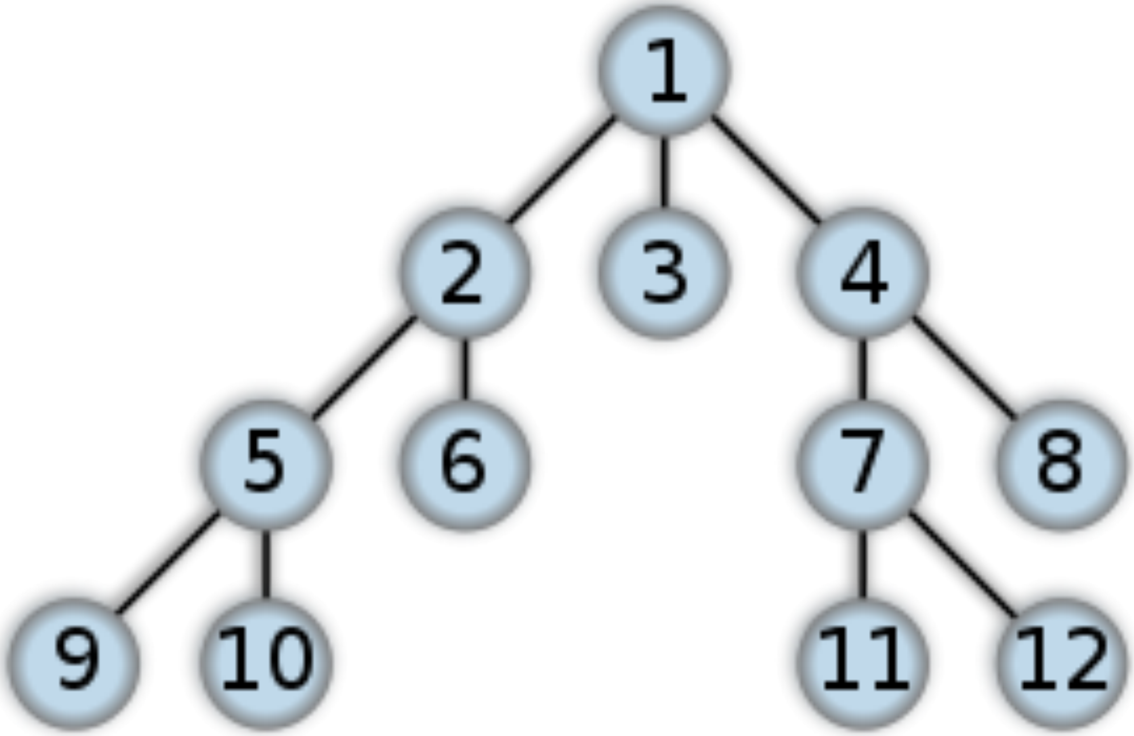
Results [2]



62% performance improvement compared to “Only-GPU”

Example 5: Graph processing (BFS)

- Graph traversal (Breadth First Search, BFS)
  - Traverses all vertices “in levels”

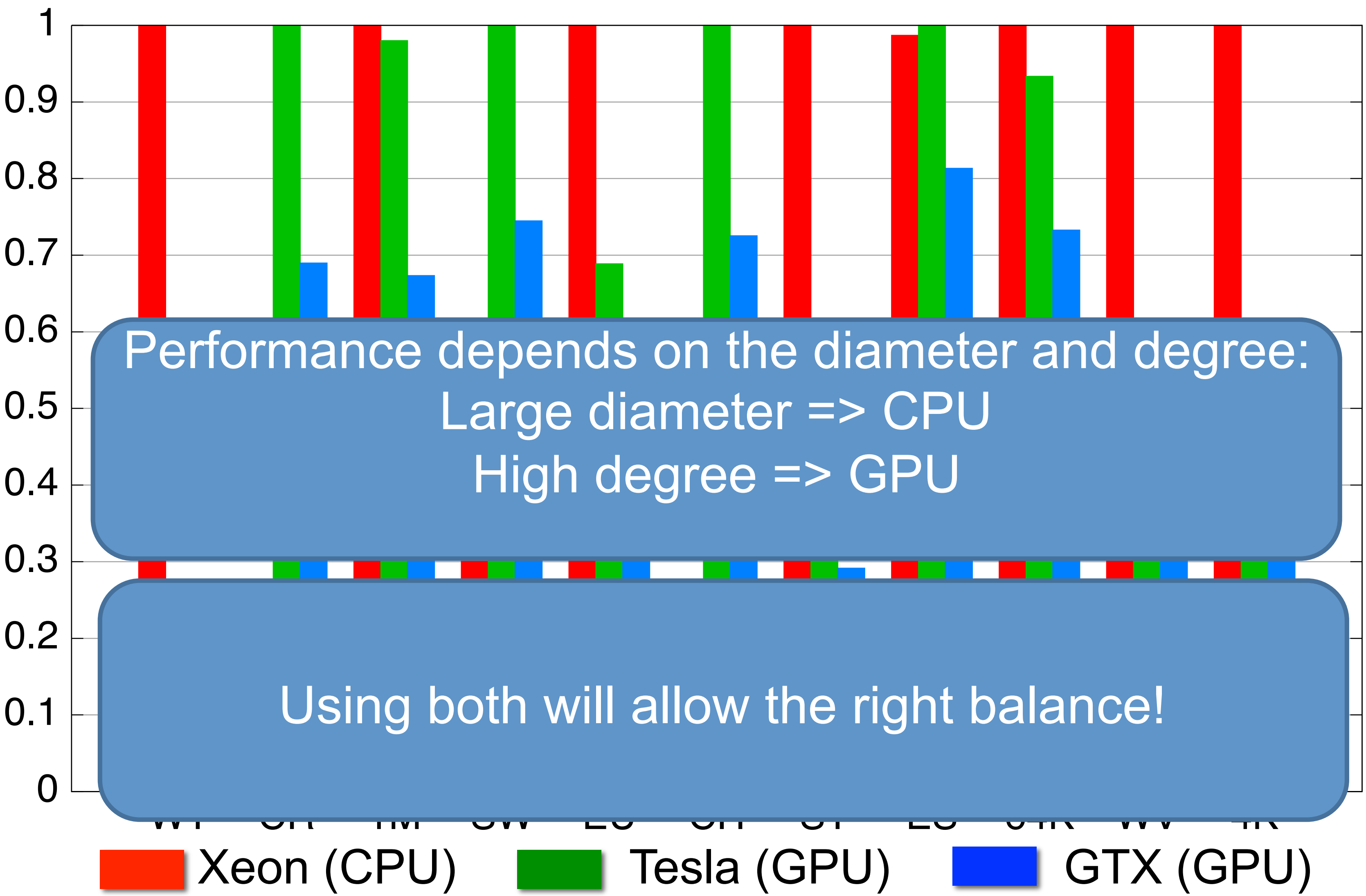


Graph processing

- … Is data-dependent
- … has poor locality
- … has low computation-to-memory-ops ratio …

- CPU or GPU?

BFS – normalized



Performance depends on the diameter and degree:
Large diameter => CPU
High degree => GPU

Using both will allow the right balance!

So …

- There are **very** few GPU-only applications
  - CPU – GPU communication bottleneck.
  - Increasing performance of CPUs
- A part of the computation can be done by the CPU.
  - How to program an application to enable this?
  - Which part?

Main challenges: **programming** and **workload partitioning!**

PART II

Challenge 1: Programming

Programming models (PMs)

- Heterogeneous computing = a mix of different processors on the same platform.
- Programming
  - Mix of programming models
    - One(/several?) for CPUs – OpenMP
    - One(/several?) for GPUs – CUDA
  - Single programming model (unified)
    - OpenCL is a popular choice

Low level

NVIDIA CUDA

OpenCL

High level

OpenACC Directives for Accelerators

OpenMP 4.0

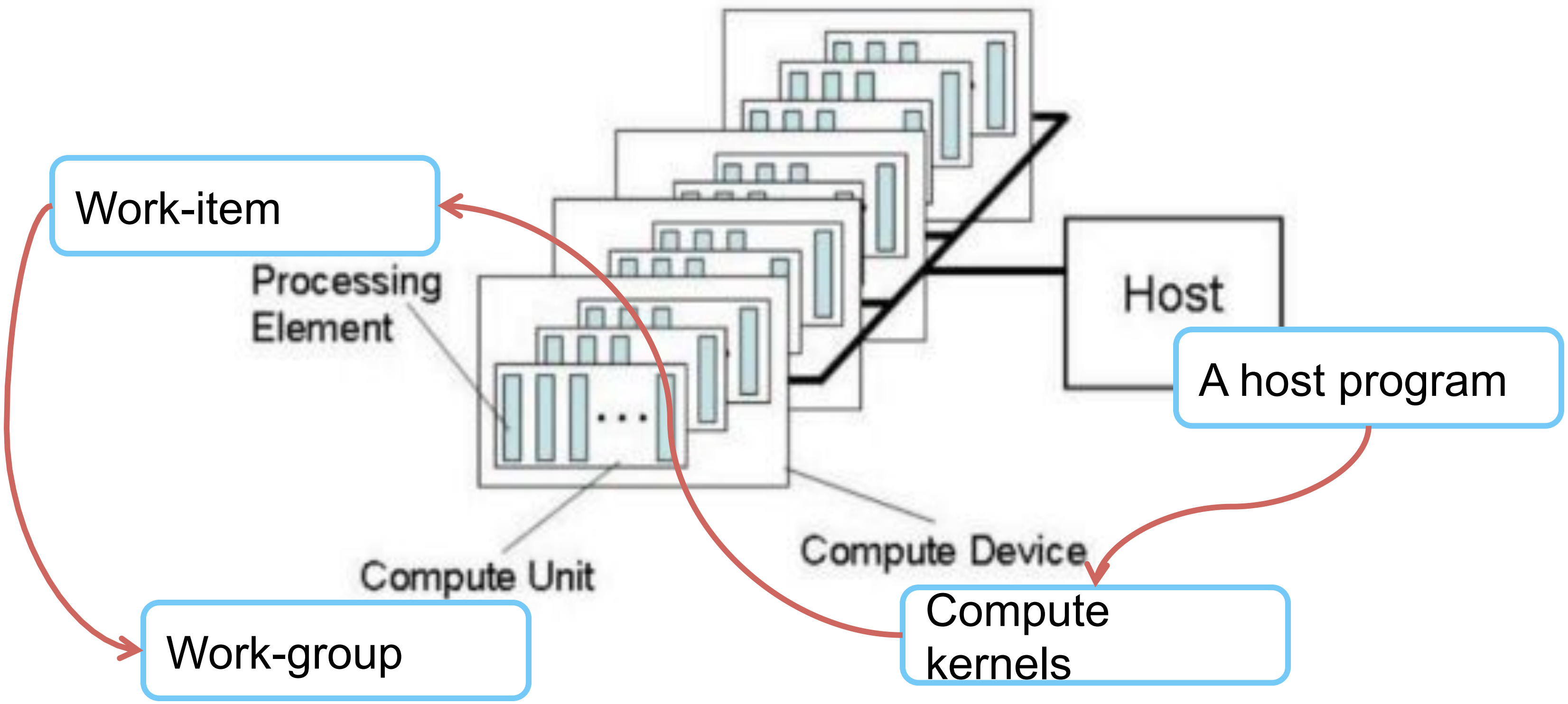
BSC OmpSs

Heterogeneous Programming Library

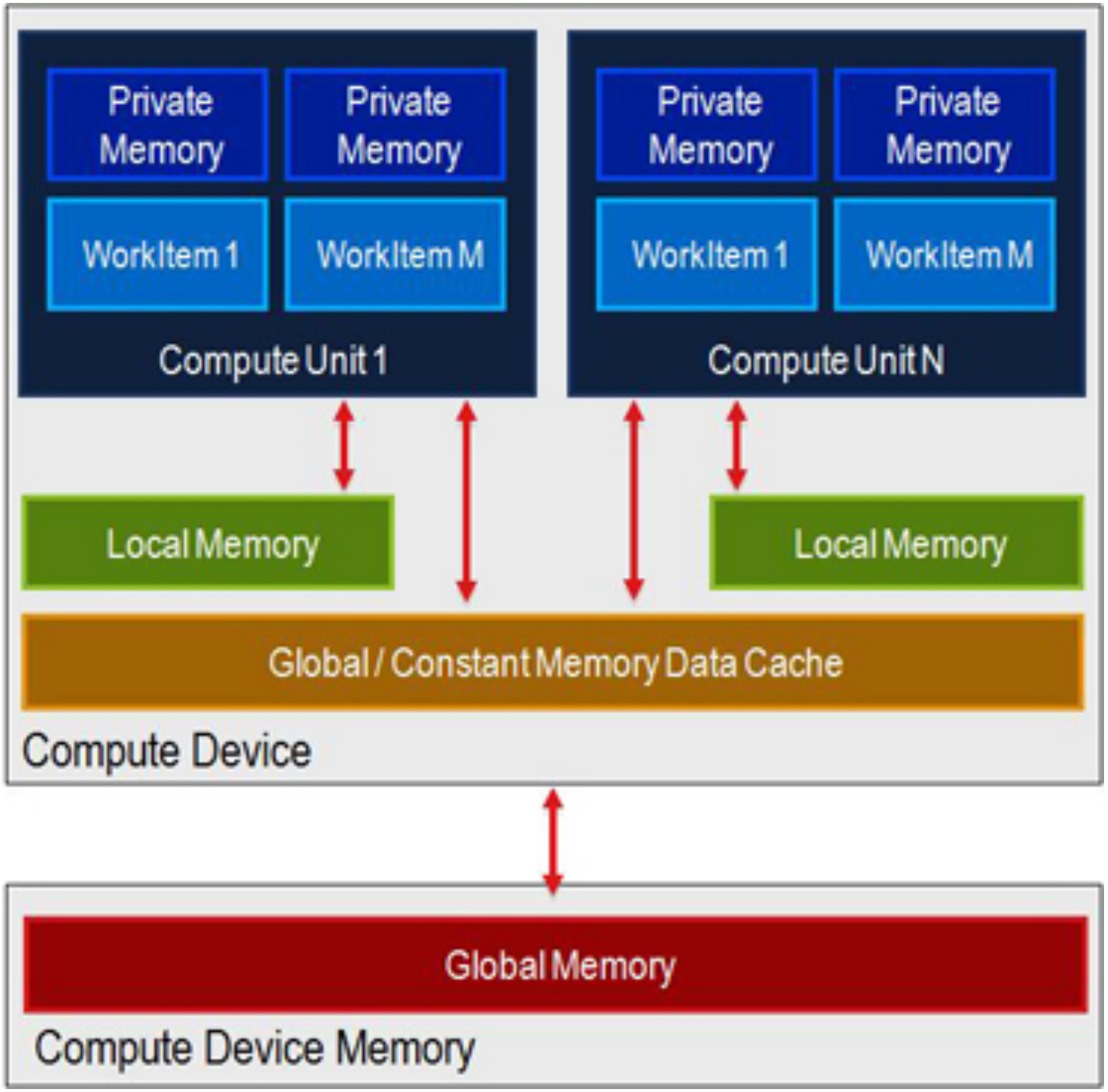
OpenCL in a nutshell

- Open standard for portable multi-core programming
- Architecture independent
  - Explicit support for multi-/many-cores
- Low-level host API
    - High-level bindings (e.g., Java, Python)
- Separate kernel language
- Run-time compilation
- Supports (some) architecture-dependent optimizations
  - Explicit & implicit

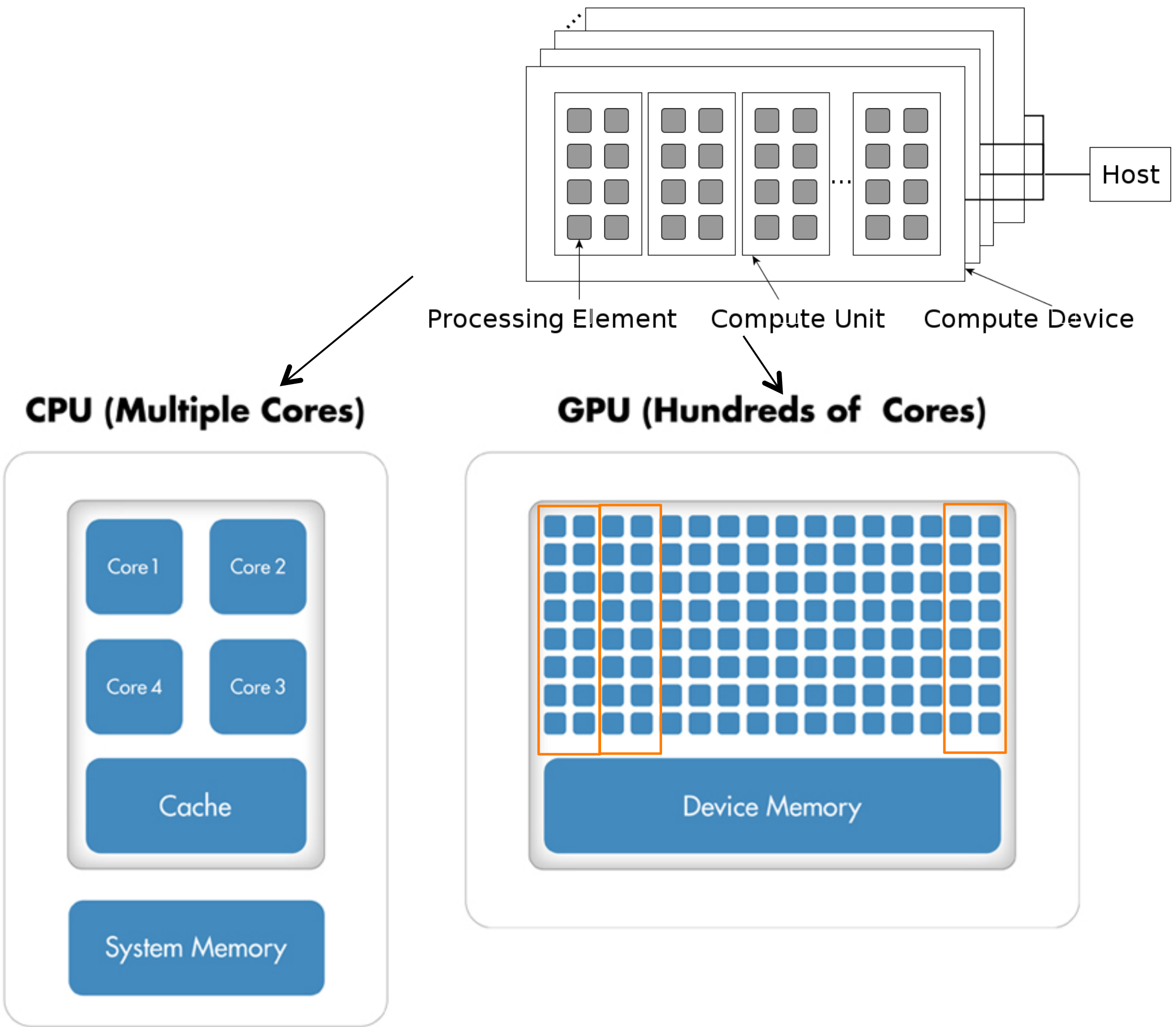
The OpenCL platform model



The OpenCL memory model



The OpenCL virtual platform



Programming in OpenCL

- Kernels are the main functional units in OpenCL
  - Kernels are executed by work-items
  - Work-items are mapped transparently on the hardware platform
- **Functional portability** is guaranteed
  - Programs run correctly on different families of hardware
  - Explicit platform-specific optimizations are dangerous
- **Performance portability** is **NOT** guaranteed
  - Performance portability is NOT guaranteed

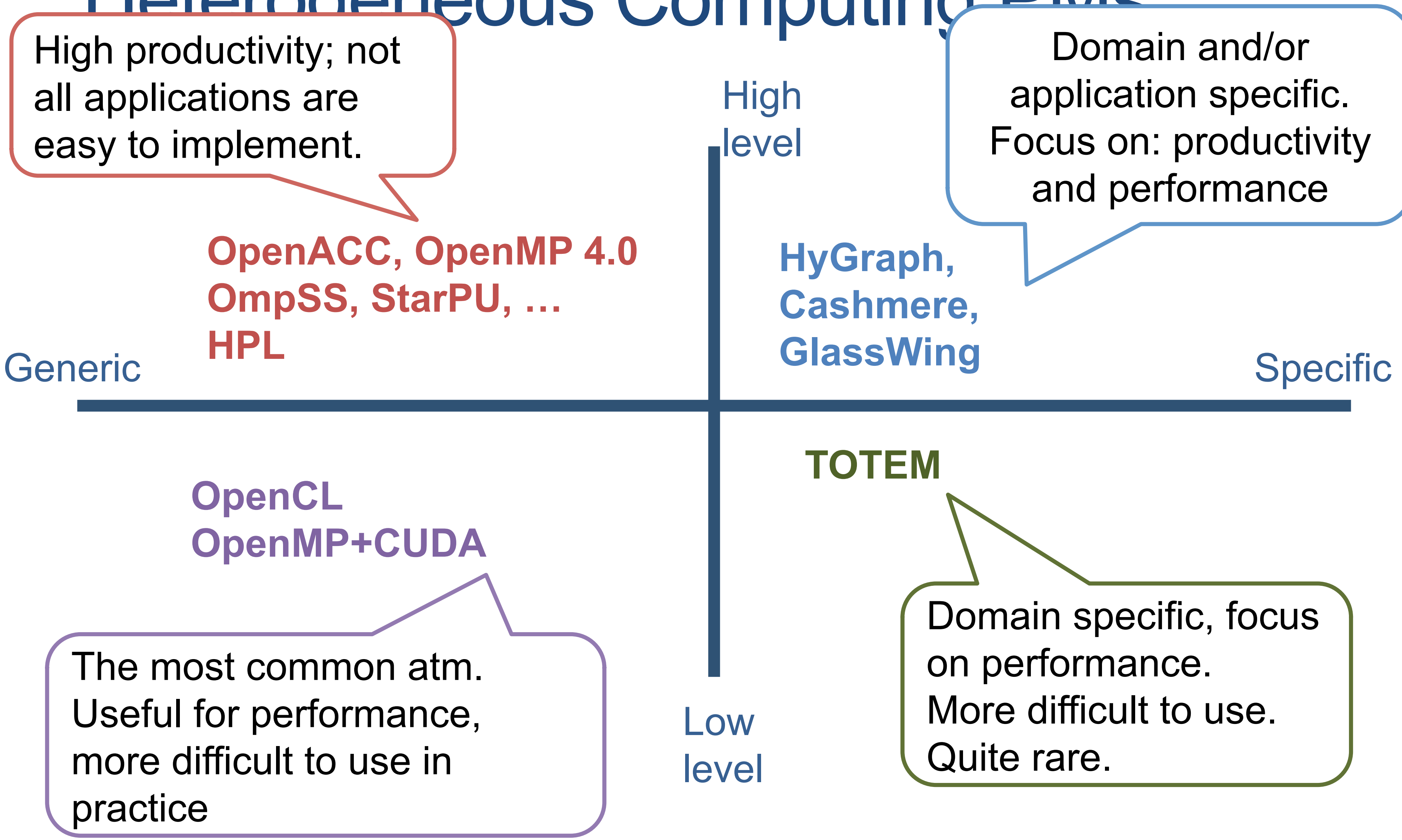
OpenCL is an efficient programming model for heterogeneous platforms iff we specialize the code to fit different processors.

OpenCL for heterogeneous platforms

- Functional portability guaranteed by the standard
- Performance portability is NOT guaranteed
  - vs. CUDA:
    - Used to be comparable (2012)
    - Lagging behind due to lack of support from NVIDIA
  - vs. OpenMP/other CPU models: 3 challenges
    - GPU-like programming styles

OpenCL is an efficient programming model for heterogeneous platforms iff we specialize the code to fit different processors.

Heterogeneous Computing PMs



Heterogeneous computing PMs

- CUDA + OpenMP/TBB
  - Typical combination for NVIDIA GPUs
  - Individual development per *PU
  - Glue code can be challenging
- OpenCL (KHRONOS group)
  - Functional portability => can be used as a unified model
  - Performance portability via code specialization
- HPL (University of A Coruna, Spain)
  - Library on top of OpenCL, to automate code specialization

Heterogeneous computing PMs

- StarPU (INRIA, France)
  - Special API for coding
  - Runtime system for scheduling
- OmpSS (UPC + BSC, Spain)
  - C + OpenCL/CUDA kernels
  - Runtime system for scheduling and communication optimization

Heterogeneous computing PMs

- Cashmere (VU Amsterdam + NLeSC)
  - Dedicated to Divide-and-conquer solutions
  - OpenCL backend.
- GlassWing (VU Amsterdam)
  - Dedicated to MapReduce applications
- TOTEM (U. of British Columbia, Canada)
  - Graph processing
  - CUDA+Multi-threading
- HyGraph (TUDelft, UTwente, UvA, NL)
  - Graph processing
  - Based on CUDA+OpenMP

End of part II

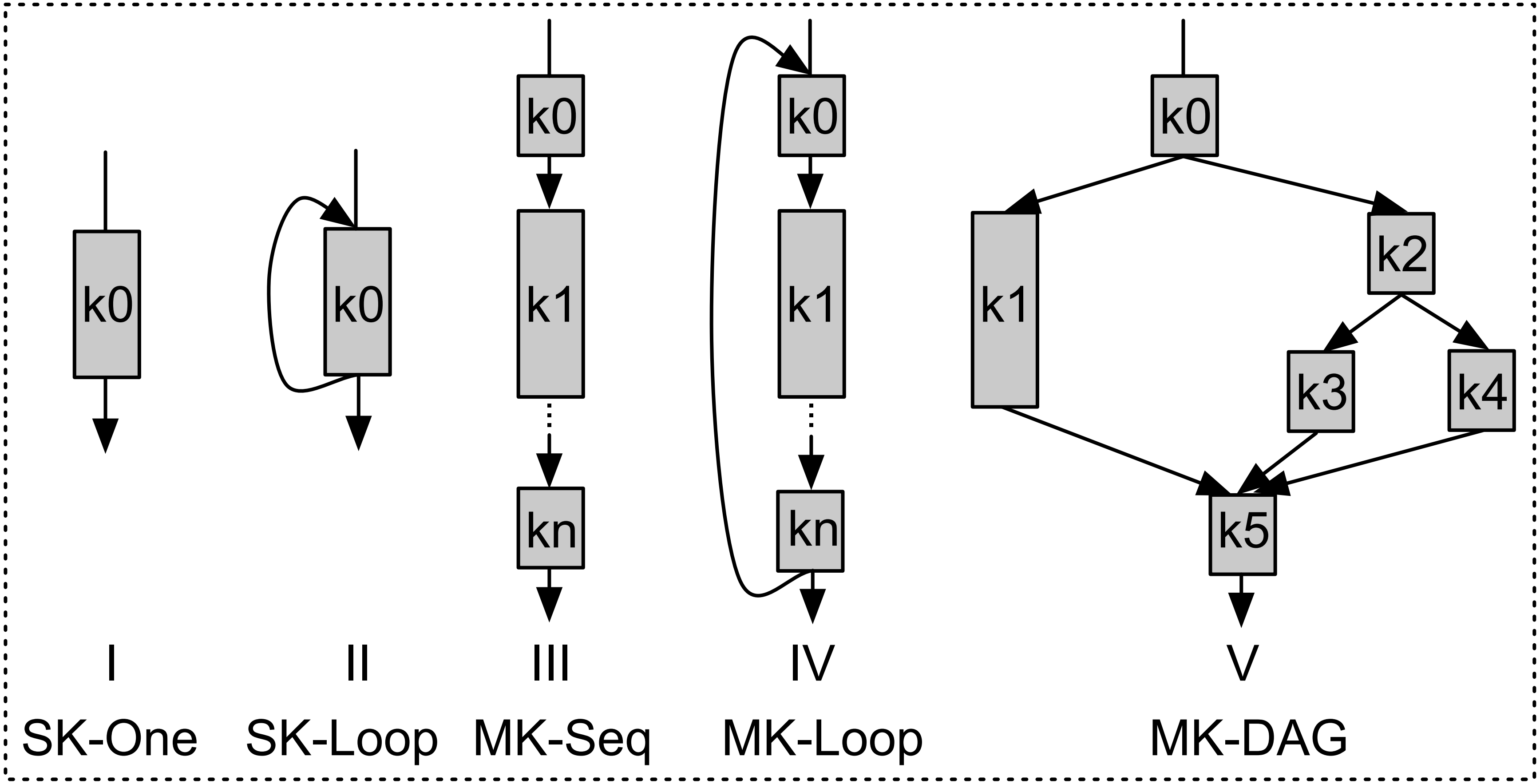
- Questions ?

PART III

Challenge 2: Workload partitioning

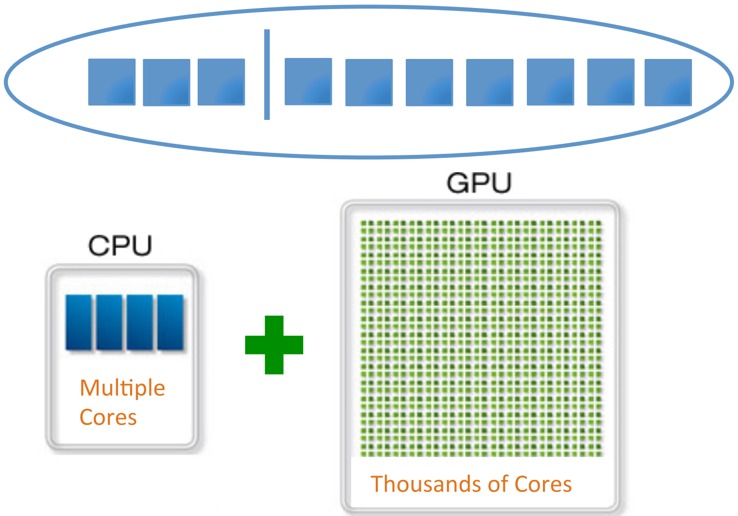
Workload

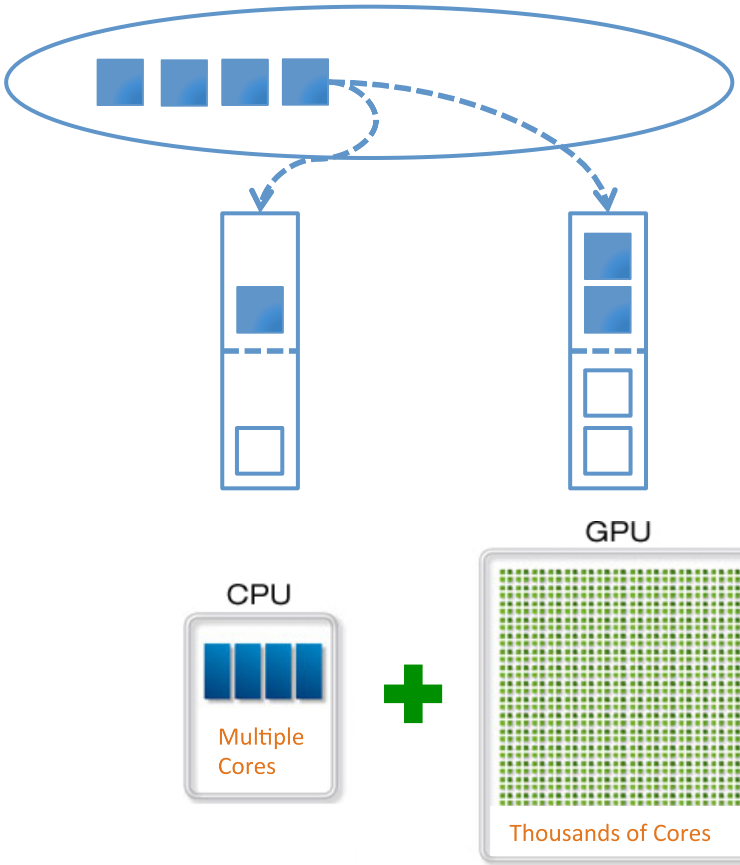
- DAG (directed acyclic graph) of “kernels”



Determining the partition

- Static partitioning (SP) vs. Dynamic partitioning (DP)



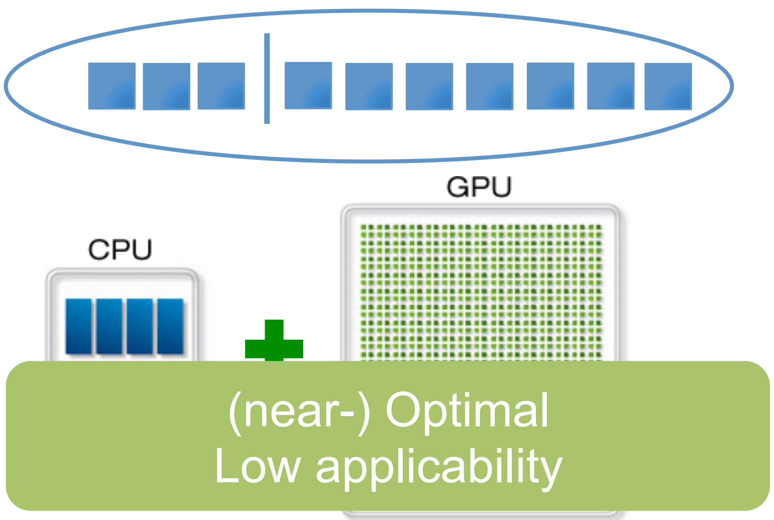


Static vs. dynamic

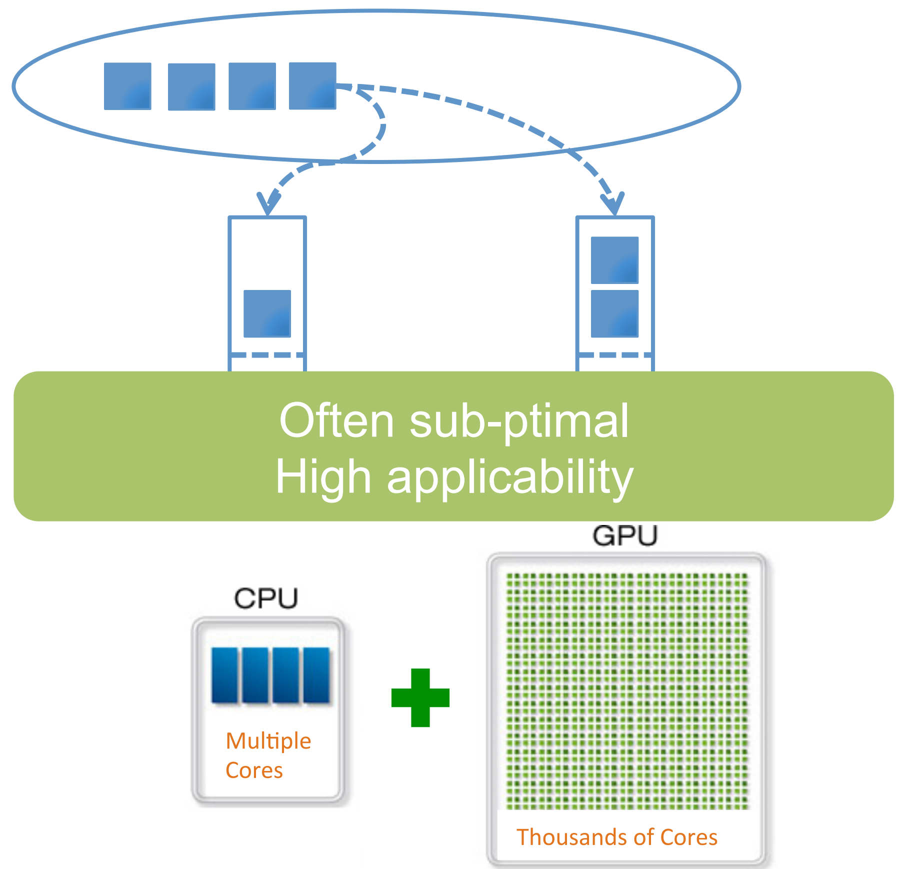
- Static partitioning
  - + can be computed before runtime => no overhead
  - + can detect GPU-only/CPU-only cases
  - + no unnecessary CPU-GPU data transfers
  - -- does not work for all applications
- Dynamic partitioning
  - + responds to runtime performance variability
  - + works for all applications
  - -- incurs (high) runtime scheduling overhead
  - -- might introduce (high) CPU-GPU data-transfer overhead
  - -- might not work for CPU-only/GPU-only cases

Determining the partition

- Static partitioning (SP) vs. Dynamic partitioning (DP)



(near-) Optimal
Low applicability



Often sub-ptimal
High applicability

Heterogeneous Computing today

Single kernel

Limited applicability.
Low overhead => high performance

Systems/frameworks:
Qilin, Insieme, SKMD,
Glinda, ...
Libraries: HPL, …

Not interesting, given that static & run-time based systems exist.

Sporradic attempts and light runtime systems

Static

Dynamic

Glinda 2.0

Low overhead => high performance
Still limited in applicability.

Run-time based systems:
StarPU
OmpSS
…

High Applicability, high overhead

Multi-kernel (complex) DAG

End of part II

- Questions ?