CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction to the GPU architecture

Directive Based GPU Programming

Vasileios Karakasis, CSCS

May 14-15, 2018

Schedule of the course

Monday, 14 May	
10:15–10:30	Introduce briefly yourself
10:30–11:30	Introduction to the GPU architecture and the Piz Daint environment
11:30–12:00	Introduction to the Piz Daint environment
LUNCH	
13:00–14:30	Introduction to OpenACC
COFFEE	
15:00–15:30	Profiling and debugging
15:30–17:00	Hands-on session
Tuesday, 15 May	
09:15–10:30	Advanced topics (activity queues)
COFFEE	
11:00–11:45	Advanced topics (interoperability with MPI and CUDA)
11:45–12:30	Advanced topics (deep copy)
LUNCH	
13:30–15:00	Hands-on session
COFFEE	
15:30–16:00	Hands-on session
16:00–16:30	OpenACC roadmap and success stories
16:30–16:45	Closing remarks

Overview

1. GPUs in HPC
   – Clock frequency vs. on-node parallelism
   – Differences with CPUs
   – Challenges for HPC applications

2. Basics of the GPU architecture
   – Execution model
   – Memory model

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

GPUs in HPC

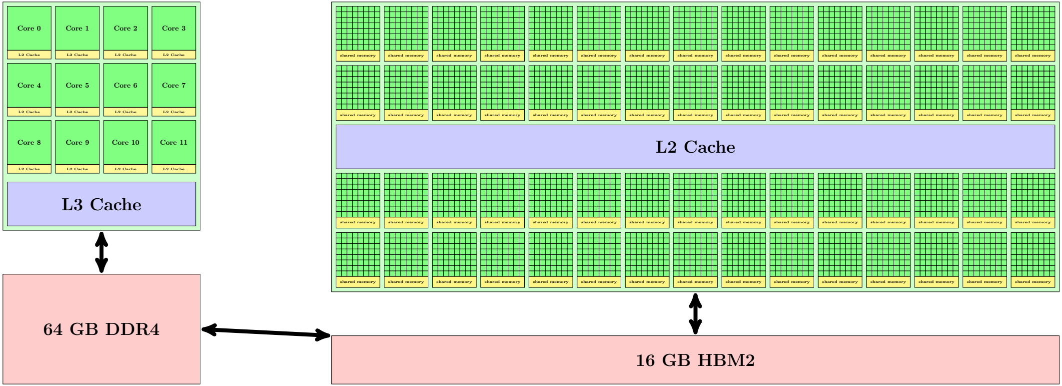
Why GPUs?

There is a trend towards more parallelism on the computing node

- Multi-core CPUs get more cores and wider vector lanes
  - 18-core 2 thread Broadwell processors from Intel
  - 12-core 8 thread Power8 processors from IBM

- Many-core Accelerators with many highly-specialized cores and high-bandwidth memory
  - NVIDIA P100 GPUs with 3582 cores
  - Intel KNL with 64 cores $\times$ 4 threads

A Piz Daint node



MPI and the free lunch

HPC applications were ported to use the message passing library MPI in the late 90s and early 2000s at great cost and effort

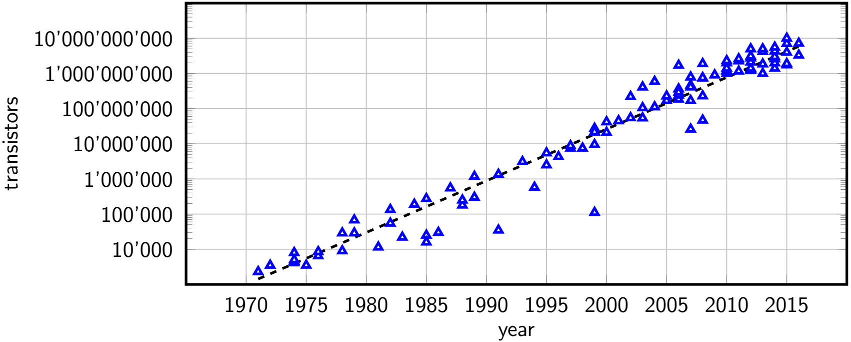
- Individual nodes with one or two CPUs
- Break problem into chunks/sub-domains
- Explicit message passing between sub-domains

The free lunch was the regular speedup in codes as CPU clock frequencies increased and as the number of nodes in systems increased

- With little/no effort, each new generation of processor bought significant speedups

…but there is no such thing as a free lunch

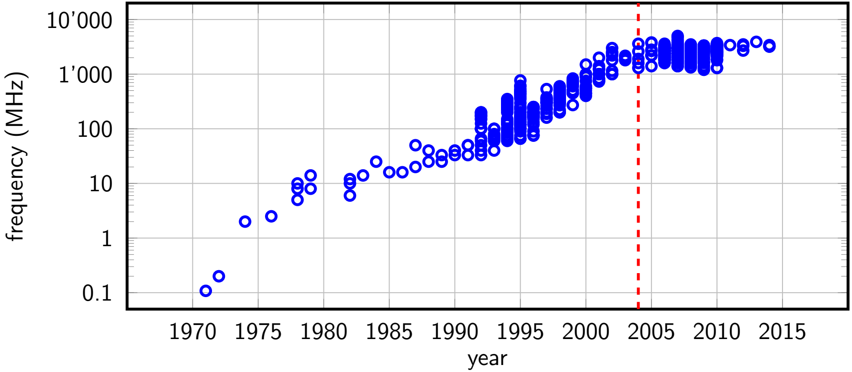
The Moore's Law



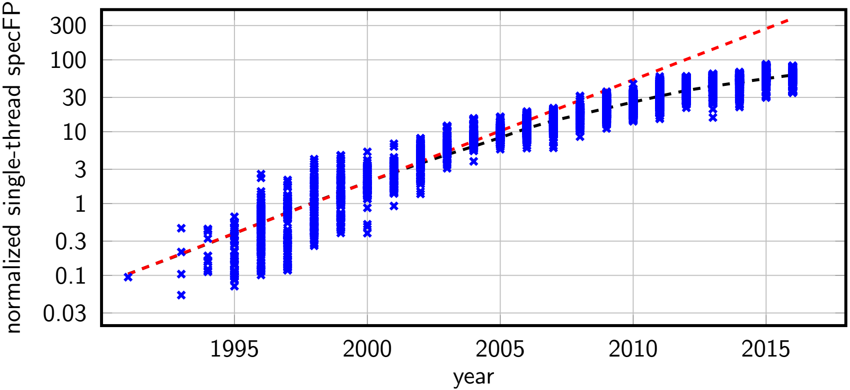
Transistor density doubles every 18 months.

The Power Wall

Frequency stopped scaling according to Moore's Law



The Power Wall



Floating point performance per core is not keeping up

The Power Wall

Why frequency could not scale?

1. $\mathbf{power} \propto \mathbf{frequency}^3$
   – Given the transistor shrinking, this was leading to very high power consumption densities!

2. Very deep pipelines were needed to keep high the instruction throughput
   – High penalties for branch mispredictions, instruction dependencies and cache misses.

How to speed up an application

There are three ways to increase performance:

1. Increase clock speed
2. Increase the number of operations per clock cycle:
   - Vectorization (SIMD)
   - Instruction-level parallelism (ILP)
   - Thread-level parallelism (TLP)
3. Don’t stall
   - e.g., increase cache reuse to avoid waiting on memory requests
   - e.g., branch prediction to avoid pipeline stalls

Clock frequency won’t increase

In fact, clock frequencies have been going down as the number of cores increases

- A 4-core Haswell processor at 3.5 GHz ($4 \times 3.5 = 14$ Gops/s) has the same power consumption as a 12-core Haswell at 2.6 GHz ($12 \times 2.6 = 31$ Gops/s)
- A P100 GPU with 3582 CUDA cores runs at 1.1 GHz

It is not reasonable to compare directly a CUDA core and an X86 core.

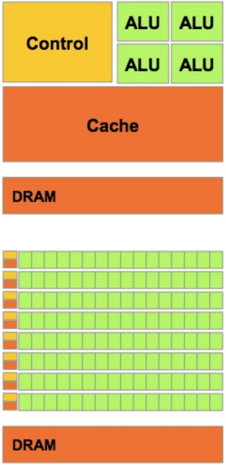
Parallelism will increase

- The number of cores in both CPUs and accelerators will continue to increase
- The width of vector lanes in CPUs will increase
  - Currently 4 doubles for AVX2
  - Increase to 8 double for AVX512 (KNL and Skylake)
- The number of threads per core will increase
  - Intel Haswell: 2 threads/core
  - Intel KNL: 4 threads/core
  - IBM Power8: 8 threads/core

Low latency or high throughput

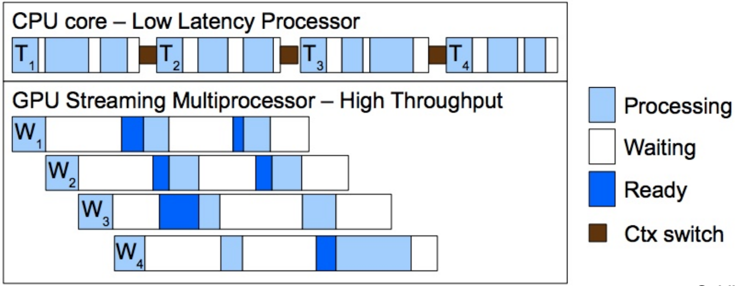
CPU
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation



GPUs are throughput devices

- CPU cores are optimized to minimize latency between operations
- GPUs aim to minimize latency between operations by scheduling multiple thread bundles (warps)



Current applications not designed for many-core

- Exposing sufficient fine-grained parallelism for multi- and many-core processors is hard
- New programming models are required
- New algorithms are required
- Existing code has to be rewritten or refactored

Current applications not designed for many-core

- Exposing sufficient fine-grained parallelism for multi- and many-core processors is hard
- New programming models are required
- New algorithms are required
- Existing code has to be rewritten or refactored

…and compute nodes are under-utilized

- Users are not getting the most out of allocations
- The amount of parallelism on-node is only going to increase!

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Understanding the GPU architecture

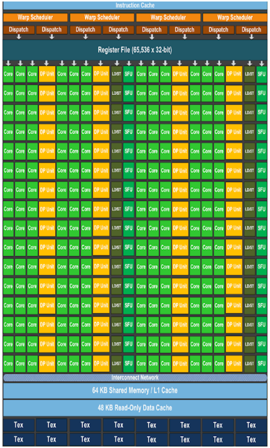
Architecture overview

The P100 GPU (Pascal architecture)

© NVIDIA Corp. 2016

The SM architecture

- Multiple lightweight single-threaded cores (64 on P100)
- Synchronous execution on groups of 32 threads/cores
  - All 32 threads execute the same instruction
- Very large register file partitioned per core (256 KB)
- Warp scheduler
  - Picks up the next ready warp
  - Very fast warp switching
- User-managed shared fast memory (64 KB on P100)



© NVIDIA Corp. 2012

Execution model

Host-directed execution

- CPU sets up and launches *kernels* on the GPU
- CPU manages the memory on the GPU
  - Allocations, transfers in and out of the GPU

Execution model

Host-directed execution

- CPU sets up and launches *kernels* on the GPU
- CPU manages the memory on the GPU
  - Allocations, transfers in and out of the GPU

Unified memory between CPU and GPU

- Virtual address space shared between CPU and GPU
- The CUDA driver and the hardware take care of the page migration
- Introduced with Kepler, significantly improved with Pascal

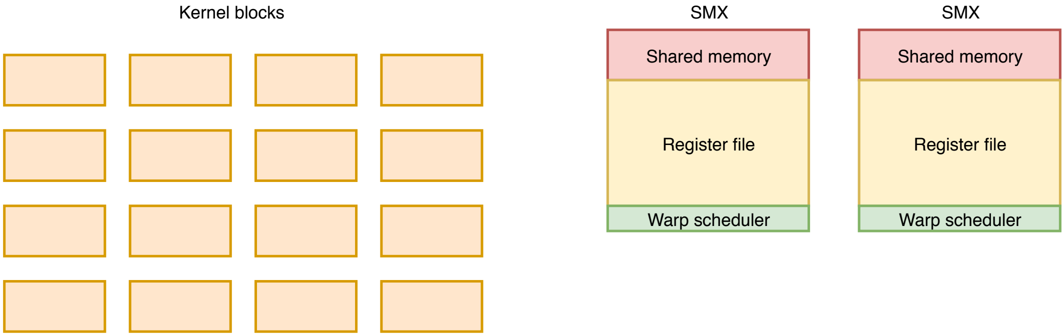
Execution model

How this huge parallelism is managed on the GPU?

- An application launches *kernels* to be executed on the GPU
- Each kernel comprises several *blocks* or *gangs* of threads
- A thread block may only run on a single SM
- Multiple thread blocks might be accomodated in a single SM, if…
  - there are enough registers,
  - there is enough shared memory or
  - hardware limits are not reached (active warps)
- Warps of any *active* block may be scheduled to run on the SM cores

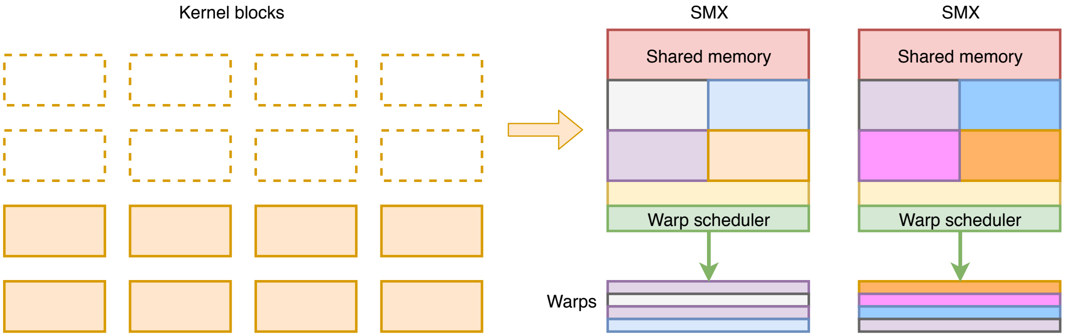
Execution model

How GPU threads are executed on the SMs



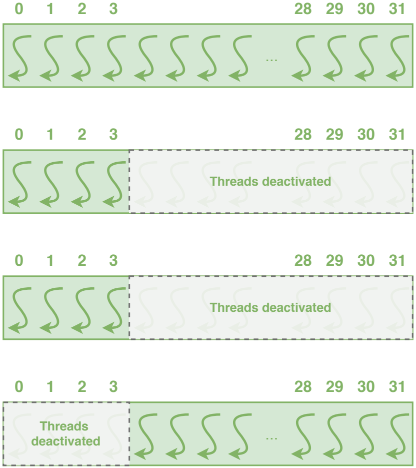
Execution model

How GPU threads are executed on the SMs



Execution model

Branching – Can individual threads execute different code?




```
if (tid < 4) {
    a;
    b;
} else {
    c;
}
```

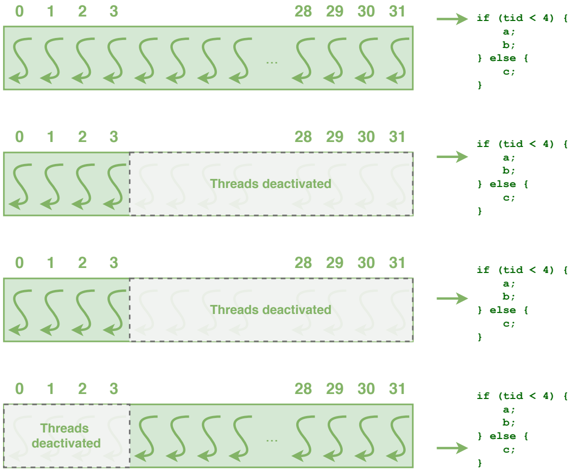
```
if (tid < 4) {
    a;
    b;
} else {
    c;
}
```

```
if (tid < 4) {
    a;
    b;
} else {
    c;
}
```

```
if (tid < 4) {
    a;
    b;
} else {
    c;
}
```

Execution model

Branching – Can individual threads execute different code?



- Each thread in a warp can take a different path, but…
- warp is executing all branches, deactivating the non participating threads.

Execution model

Implications

- Lots of parallelism is needed to cover execution latencies
  – Enough warps must be available for scheduling

- Global synchronization is not possible
  – Not all the blocks of a kernel run simultaneously
  – *Synchronization is only possible within the threads of a block*

- If program's control flow diverges within a warp → redundant execution
  – Both branches are executed by the warp redundantly

Memory model

Memory hierarchy

- Global high bandwidth memory (558 GB/s on P100)
  - Accessible from all thread gangs
  - Data persistent across kernel invocations
  - Memory accesses of warp threads are *coalesced* into one or two memory transactions if they are properly aligned and regular
- L1 cache/Shared memory
  - Shared among the threads of a single thread gang
  - One-cycle access latency, if warp threads access different locations
  - Software or hardware managed
  - No cache coherency across SMs
  - No sequential consistency → enforced by synchronization primitives

Memory model

Memory hierarchy (cont'd)

- Local memory
  - Not visible to the programmer
  - The compiler may place there automatic variables
- Constant memory
  - Cached part of the global memory used for storing constants
  - Visible to the programmer
- Texture and surface memory
  - Cached part of the global memory optimized for 2D accesses

How to program the GPUs?

- CUDA
  - C/C++ language extensions
  - Low-level, more code
  - Requires a good understanding of the GPU architecture
  - Can leverage every aspect of the architecture to get a fully optimized implementation

- Using directives OpenACC/OpenMP
  - Easy to use and productive
  - High-level, non-intrusive changes in the code
  - More on it, after lunch…

Thank you for your attention