CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction to GPUs in HPC

Ben Cumming, CSCS
July 27, 2015

Why GPUs?

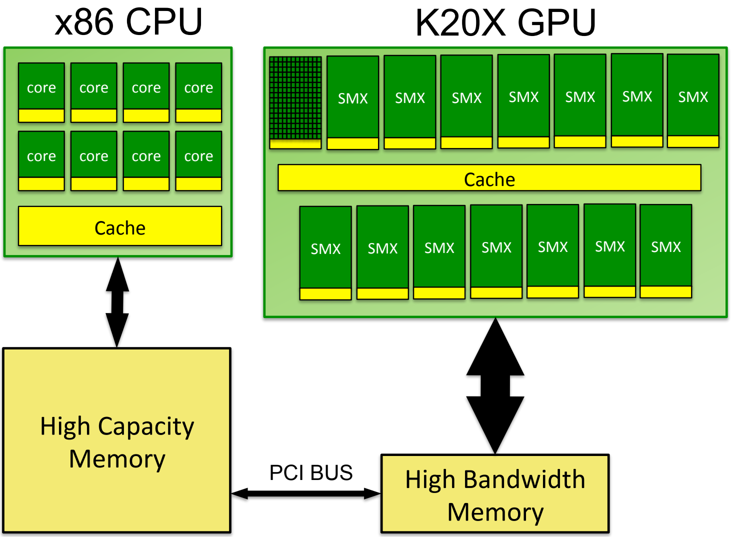
There is a trend towards more parallelism “on node”

Multi-core CPUs get more cores and wider vector lanes

- 16-core×2 thread Haswell processors from Intel
- 12-core×8 thread Power8 processors from IBM

Many-core Accelerators with many highly-specialized cores and high-bandwidth memory

- NVIDIA K20X GPUs with 2688 cores
- Intel KNL with 72 cores × 4 threads


x86 CPU
K20X GPU
core
core
core
core
core
core
core
core
Cache
SMX
SMX
SMX
SMX
SMX
SMX
SMX
Cache
SMX
SMX
SMX
SMX
SMX
SMX
SMX
High Capacity Memory
PCI BUS
High Bandwidth Memory

Most current applications are 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!

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

How to speed up an application

There are 3 ways to increase performance

1. increase clock speed
2. increase the number of operations per clock cycle
   - e.g. vectorization
   - e.g. multi-core
3. don't stall
   - e.g. cache to avoid waiting on memory requests
   - e.g. branch prediction to avoid pipeline stalls

What about just increasing clock frequency?

The number of operations per second that can be performed is directly proportional to CPU frequency

- increasing frequency is a great way to increase performance

Power consumption is a function of frequency f

$$P_{\text{dynamic}} = CV^2 f$$

However voltage V is proportional to frequency, so power increases **super-linearly** with clock frequency

- increasing frequency is an even better way to increase power consumption!

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*3.5=14 Gops/second) has the same power consumption as a 12-core Haswell at 2.6 GHz (12*2.6=31 Gops/second)
- a K20X GPU with 2688 CUDA cores runs at 800 MHz

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 AVX
  - increase to 8 double for AVX-3 (KNL and Skylake)
- the number of threads per core will increase
  - Haswell supports 2 threads/core
  - KNL supports 4 threads/core
  - Power-8 supports 8 threads/core

Memory is slow

- memory is much slower than processors
  - for both CPU and GPU the latency of fetching a cache-line from memory is 100s of cyles
  - that is 100s of cycles that the processor is stalled, unable to "work"
  - latency has to be hidden or reduced to minimise stalling

CPU solution: deep caches and prefetching

- use fast on-chip memory to **cache** frequently used data
- use hardware to **prefetch** data to cache before it is required

GPU solution: over subscribe threads to cores

- schedule many threads per core
- threads that are waiting for data are idle

TLDR: change because power

Writing good concurrent code for many-core is difficult

- but the days of easy speed up each generation of CPU are over
  - performance gains must not increase power consumption
- to continue improving performance many-core will be required
- this course will be about one type of many-core architecture NVIDIA GPUs
  - both CUDA and OpenACC are GPU-specific
  - but the concepts will be universally applicable to other many-core architectures (e.g. Xeon Phi)

Terminology

- the CPU and its memory are called the **host** (because GPUs require a host CPU to coordinate them)
- the GPU and its memory are referred to as the **device**

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Using GPUs in Your Application

Libraries

- use an off the shelf solution implemented in a library
- for specific tasks like dense linear algebra and FFTs, libraries are very hard to beat
- e.g. cublas, PETSc, cuFFT

Directives

- OpenACC and OpenMP 4 define **directives** that can be used to instruct the compiler how to generate GPU code
- in theory the easiest path for porting

GPU-specific Languages

- languages designed for GPU programming
- maximum flexibility and performance
- e.g. CUDA and OpenCL