# An Introduction to OpenACC

Alistair Hart
Cray Exascale Research Initiative Europe

# Timetable

- **Monday 18th February 2013**
  - 10:10 Lecture 1: Introduction to the Cray XK7
  - 10:30 Lecture 2: Using the Cray XK7 Programming Env.
  - 10:50 Practical 0: Logging on, running a very simple job
  - *11:00* *break*
  - 11:30 Lecture 3: Basic OpenACC
  - 12:15 Practical 1: A first OpenACC code
  - 12:40 Lecture 4: Porting a simple example to OpenACC
  - *13:10* *lunch*
  - 14:10 Lecture 5: OpenACC debugging (CSCS)
  - 14:30 Practical 2: Porting the simple example yourself
  - 15:00 Lecture 6: Advanced OpenACC
  - 15:40 Lecture 7: Using Cray PE tools to port a code
  - 16:10 Practical 3: Using the tools to port a larger code
  - *16:30* *break*
  - 16:50 Lecture 8: Porting a parallel code to OpenACC
  - 17:20 Lecture 9: OpenACC interoperability and roadmap
  - 17:40 Practical 3 (continued) and discussions
  - *18:00* *close*

# Contents

- **The aims of this course:**
  - To motivate why directive-based programming of GPUs is useful
  - To introduce you to the OpenACC programming model
  - To give you some experience in using OpenACC directives
    - with some hints and tips along the way
  - To introduce you to profiling tools
    - to understand and tune OpenACC performance
    - to help you understand a real application to start OpenACC-ing...
- **The idea is to equip you with the knowledge to develop applications that run efficiently on parallel hybrid supercomputers**
  - not just on single GPUs

# Inside the Cray XK7 and the Nvidia Kepler K20X GPU

Alistair Hart
Cray Exascale Research Initiative Europe

# Contents of this talk

- **An overview of the Cray XK7**
  - The hardware
  - Why GPUs are interesting for Exascale research
  - Programming models for GPUs
- **A quick GPU refresher**
  - the hardware
  - how codes execute on the hardware
  - what this means for the programmer
- **Things to consider before starting an OpenACC port**

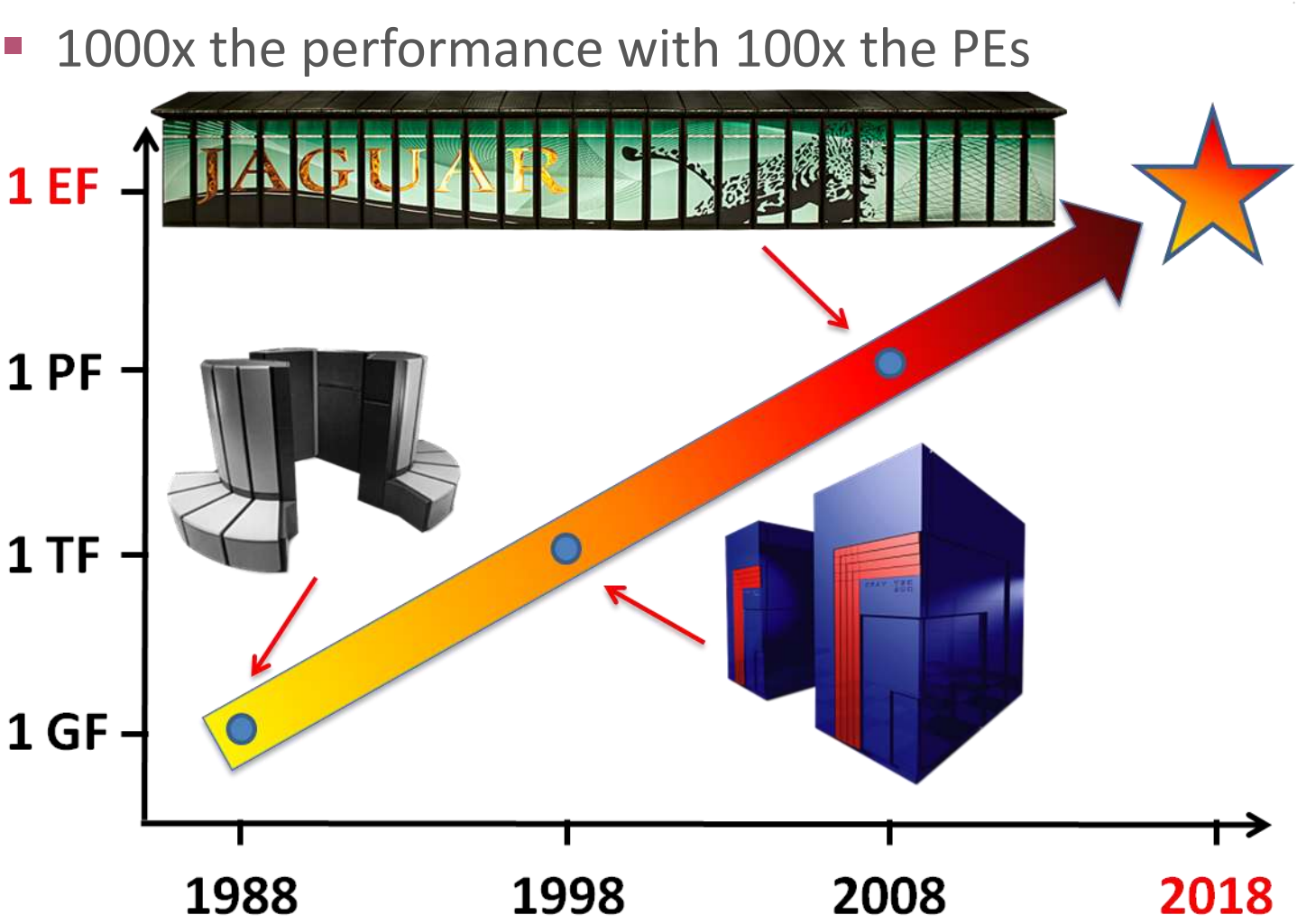
# "Accelerating the Way to Better Science"

## Cray XK7 supercomputer

- **Node architecture:**
  - One AMD Series 6200 Interlagos CPU (16 cores)
    - Cray XE6 (e.g. HECToR) has two of these per node
  - One Nvidia GPU
    - Kepler K20x
      - 2688 + 896 cores, 1.3 TFlop/s DP, 6GB memory
- **Cray Gemini interconnect**
  - shared between two nodes
  - high bandwidth/low latency scalability
- **Fully integrated/optimised/supported**
  - Tight integration of GPU and NIC drivers

# The Exascale is coming...

- Sustained performance milestones every 10 years...
  - 1000x the performance with 100x the PEs




(and they're all Crays)

# Exascale, but not exawatts

- **Power is a big consideration in an exascale architecture**
  - Jaguar XT (ORNL) draws 6MW to deliver 1PF
  - The US DoE wants 1EF, but using only 20MW...
- **A hybrid system is one way to reach this, e.g.**
  - $10^5$ nodes (up from $10^4$ for Jaguar)
  - $10^4$ FPUs/node (up from 10 for Jaguar)
    - some full-featured cores for serial work
    - a lot more cutdown cores for parallel work
  - Instruction level parallelism will be needed
    - continues the SIMD trend SSE → AVX → ...
- **This looks a lot like the current GPU accelerator model**
  - manycore architecture, split into SIMT threadblocks
  - Complicated memory space/hierarchy (internal and PCIe)
- **And this looks a lot like the old days**
  - welcome back to vectorisation, we kept the compiler ready for you

# The Exascale furrow is a hard one to plough...

Seymour Cray, the pioneer of supercomputing, famously once asked if you would rather plough a field with two strong oxen or five-hundred-and-twelve chickens.

Since then, the question has answered itself: power restrictions have driven CPU manufacturers away from “oxen” (powerful single-core devices) towards multi- and many-core “chickens”.

An exascale supercomputer will take this a step further, connecting tens of thousands of many-core nodes.

Application programmers face the challenge of harnessing the power of tens of millions of threads.

EPCC News, issue 70 (Autumn 2011)

- **EU FP7 Network: Collaborative Research into Exascale Software Tools and Applications**


- **Consortium has**
  - Leading European HPC centres
    - EPCC, HLRS, CSC, PDC
  - Hardware partner
    - Cray
  - Tools providers
    - TUD (Vampir), Allinea (DDT)
  - Codesign application owners, specialists
    - ABO, JYU, UCL, ECMWF, ECP, DLR

- **Codesign approach**
  - Focus on real problems in real applications
  - 6 apps: GROMACS, OpenFOAM, IFS, Elmfire, Nek5000, HemeLB

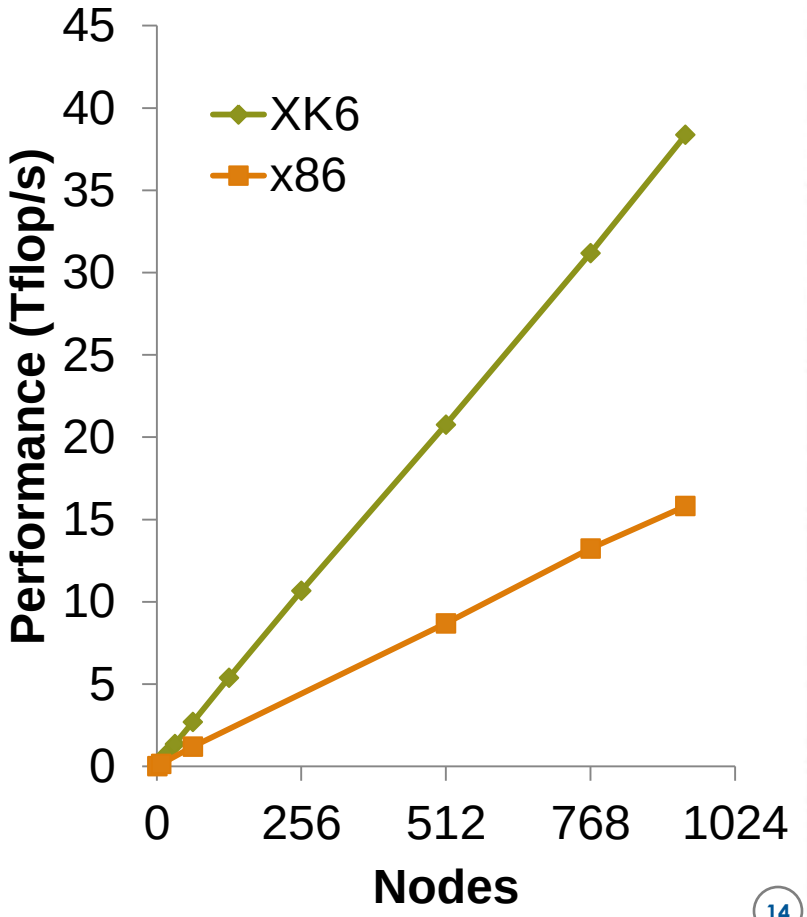
- **CRESTA and its two partner projects are the first Exascale development projects funded by Europe**
  - Runs from Oct. 2011-Sept. 2014
  - CRESTA workshop on Exascale codesign at SC12 now recruiting speakers

# Accelerator programming

- **Why do we need a new GPU programming model?**
- **Aren't there enough ways already?**
  - CUDA (incl. NVIDIA CUDA-C & PGI CUDA-Fortran)
  - OpenCL
  - Stream
  - hiCUDA ...
- **All are quite low-level and closely coupled to the GPU**
  - User needs to rewrite kernels in specialist language:
    - Hard to write and debug
    - Hard to optimise for specific GPU
    - Hard to port to new accelerator
  - Multiple versions of kernels in codebase
    - Hard to add new functionality

# CUDA for experts


- **If you work hard, you can get good parallel performance**
- **Ludwig Lattice Boltzmann code rewritten in CUDA**
  - Reordered all the data structures (structs of arrays)
  - Pack halos on the GPU
  - Streams to overlap compute, PCIe comms, MPI halo swaps
- **10 cabinets of Cray XK6**
  - 936 GPUs (nodes)
- **Only 4% deviation from perfect weak scaling between 8 and 936 GPUs.**
- **But...**
  - Most applications don't have this level of expert developer support



# Directive-based programming

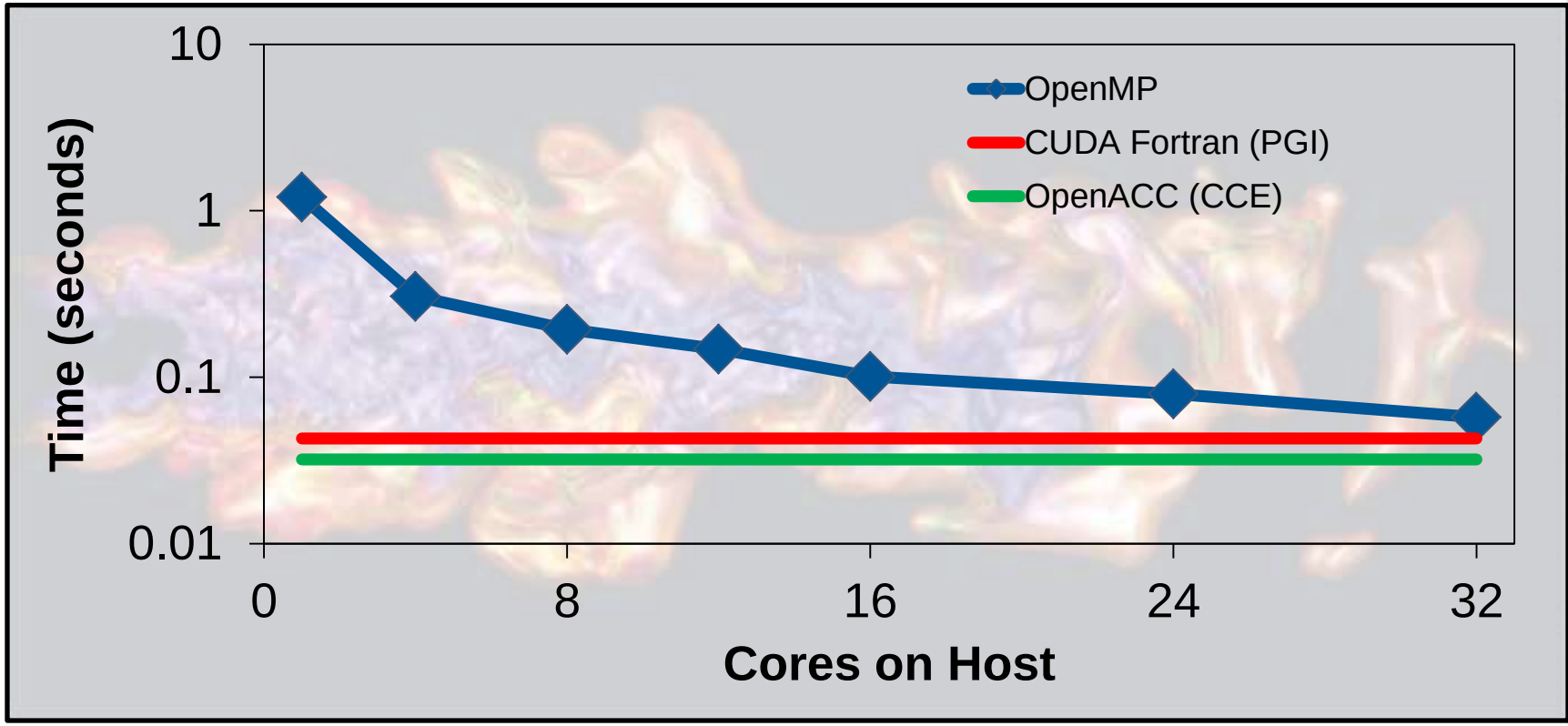
## Directives provide a high-level alternative

+ **Based on original source code (e.g. Fortran, C, C++)**
  + Easier to maintain/port/extend code
  + Users with (for instance) OpenMP experience find it a familiar programming model
  + Compiler handles repetitive boilerplate code (cudaMalloc, cudaMemcpy...)
  + Compiler handles default scheduling; user can step in with clauses where needed

– **Possible performance sacrifice**
  – Important to quantify this
  – Can then tune the compiler
  – Small performance sacrifice is an acceptable trade-off for portability and productivity
    – After all, who handcodes in assembler for CPUs these days?

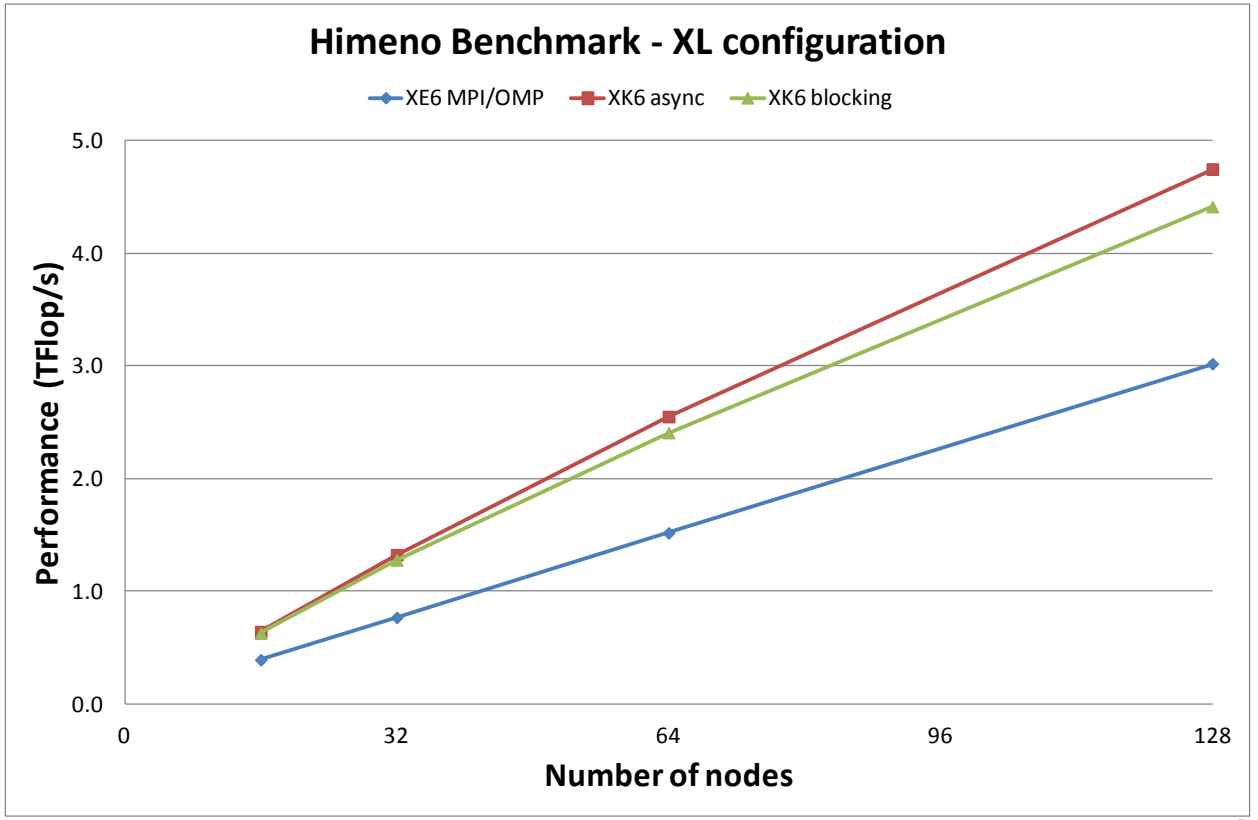
# Performance compared to CUDA

- Is there a performance gap relative to explicit low-level programming model? **Typically 10-15%, sometimes none.**
- Is the performance gap acceptable? **Yes.**
  - e.g. S3D comp_heat kernel (ORNL application readiness):



# Node-for-node performance comparison

- **Does accelerated parallel application performance justify buying a GPU on the node (e.g. Cray XK6)**
  - rather than another CPU (cf. Cray XE6)?
  - For many codes, yes.



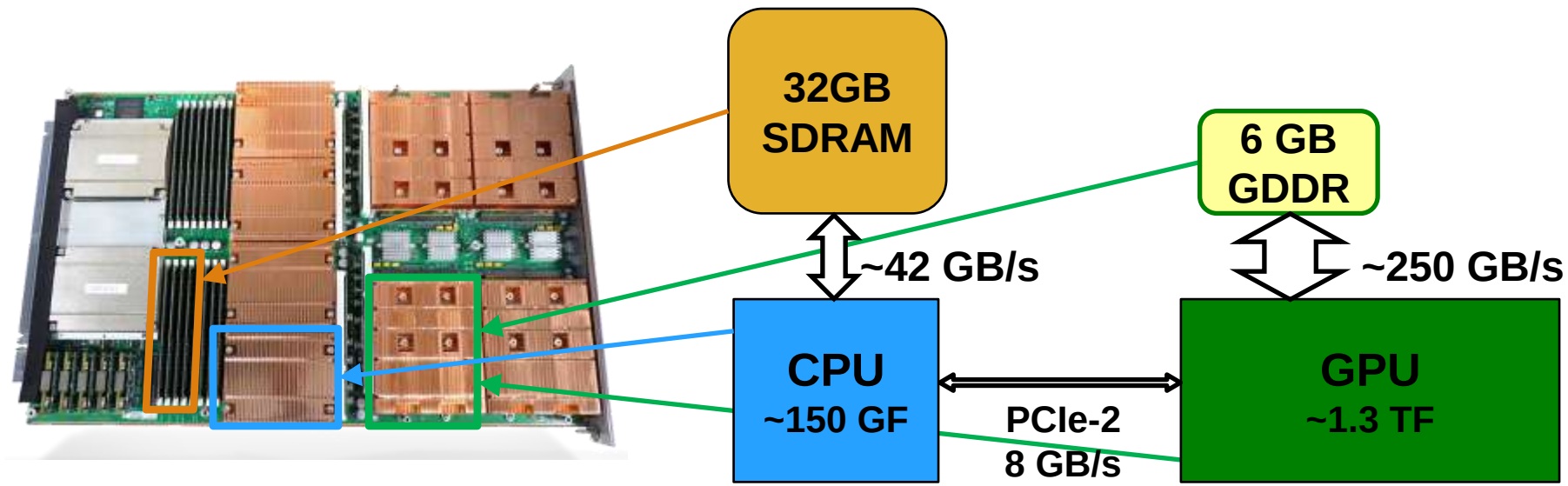
# Structure of this course

- **Aims to lead you through the entire development process**
  - What is OpenACC?
  - How do I use it in a simple code?
  - How do I port a real-sized application?
  - Performance tuning and advanced topics
- **It will assume you know**
  - A little bit about GPU architecture and programming
    - SMs, threadblocks, warps, coalescing
    - a quick refresher follows
- **It will help if you know**
  - The basic idea behind OpenMP programming
    - but this is not essential

# A quick GPU refresher

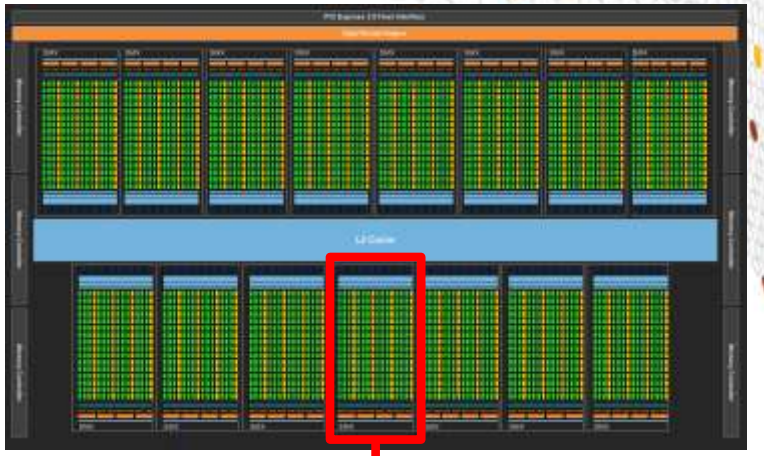
# How fast are current GPUs?

- **Beware the hype: "I got 1000x speed-up on a GPU"**
- **What should you expect?**
  - Cray XK7:
    - Flop/s: GPU ~9x faster than single, whole CPU (16 cores)
    - Memory bandwidth: GPU ~6x faster than CPU
  - These ratios are going to be similar in other systems
- **Plus, it is harder to reach peak performance on a GPU**
  - Your code needs to fit the architecture
  - You also need to factor in data transfers between CPU and GPU

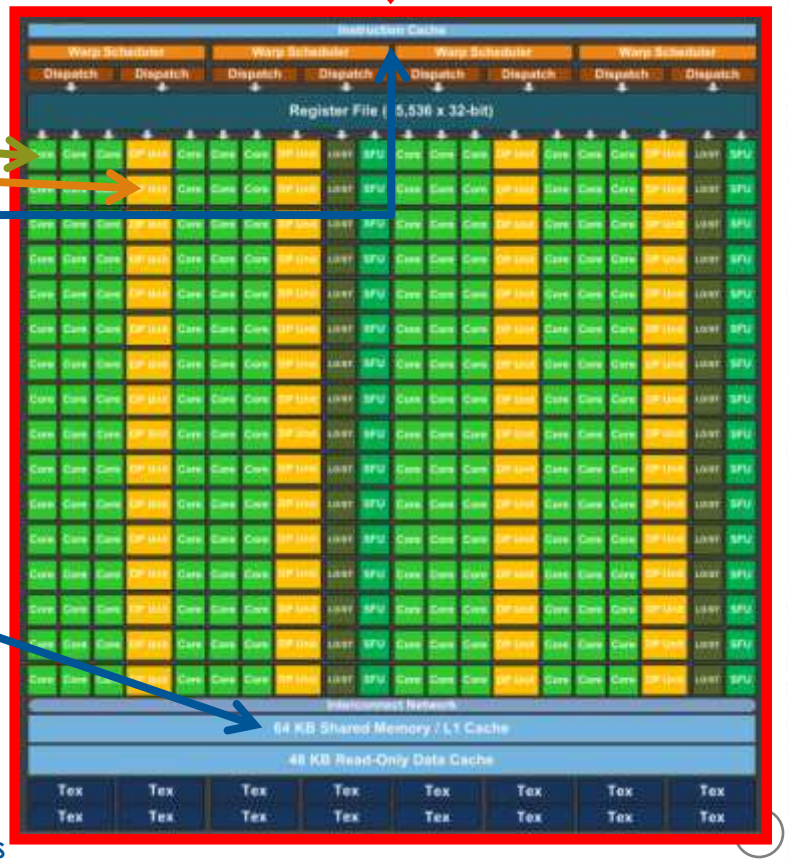


# Nvidia K20X Kepler architecture

- **Global architecture**
  - a lot of compute cores
    - 2688 SP plus 896 DP; ratio 3:1
  - divided into 14 Streaming Multiprocessors
  - these operate independently



- **SMX architecture**
  - many cores
    - 192 SP
    - 64 DP
  - shared instruction stream; same ops
    - lockstep, SIMT execution of same ops
    - SMX acts like vector processor




- **Memory hierarchy**
  - each core has private registers
    - fixed register file size
  - cores in an SM share a fast memory
    - 64KB, split between:
      - L1 cache and user-managed
  - all cores share large global memory
    - 6GB; also some specialist memory

# Program execution on a GPU

- **Kernels are launched by CPU to execute on GPU**
  - CUDA programming model:
    - parallel work divided between large number of lightweight threads
    - threads are divided into logical sets (known as threadblocks)
- **The GPU runtime schedules these on the GPU hardware**
  - each threadblock executes on an SM
    - threads in threadblock have some shared memory
    - you can synchronise the threads in a threadblock
  - different threadblocks execute on different SMs
    - runtime dynamically swaps threadblocks in and out of execution
      - to hide memory latency
    - there is no mechanism for synchronising between threadblocks within kernel
- **The kernels launch asynchronously**
  - global synchronisation is handled by the host at the end of the kernel
  - host also handles and synchronises memory copies between CPU and GPU

# What CUDA doesn't tell you (upfront)

- **Threads are not created equal**
  - The SM is really a vector processor of width 32
    - groups of 32 cores act in lockstep, rather than independently
    - shares a single instruction stream with single program counter
    - Threads within a threadblock are divided into sets of 32 (warps)
    - each warp executes in SIMT fashion using 32 cores of SM
    - if a threadblock contains multiple warps, these are executed in turn
      - i.e. if there are more than 32 threads in the threadblock
  - Memory loads/stores are also done on a per-warp basis
    - Loading/storing 32 consecutive memory addresses at once
- **So really, the compiler is implementing your code using vector instructions**
  - This is not explicit in the CUDA programming model, but is crucial to gaining good performance from a GPU
    - whichever programming model you are using (it's a hardware thing)

# What does this mean for the programmer?

- You need a lot of parallel tasks (i.e. loop iterations) to keep GPU busy
  - Not just 2688, but $10^4$ to $10^6$ to allow runtime to hide memory latency
- Your loop(s) must vectorise (at least with vector length of 32)
  - So we can use all 32 threads in a warp with shared instruction stream
  - Branches in loopnests are allowed, but too many will be an issue
- Global memory access is done with (sequential) vector loads
  - For good performance, want as few of these as possible
  - so all the threads in warp collectively load a contiguous block of memory at the same point in the instruction stream
  - This is known as "coalesced memory access"
  - So threads should access fastest-moving index of each array
- No internal mechanism for synchronising between threadblocks
  - So reduction operations are more complicated, for instance, even though all threadblocks share same global memory
- Data transfers between CPU and GPU are very expensive
  - You need to concentrate on "data locality" and avoid "data sloshing"
  - Keeping data in the right place for as long as it is needed is crucial
  - This probably means porting more of the application than you expected

# So...

- **GPUs are like race-horses**
  - can give very good performance
  - but it needs to be carefully coaxed from them
  - porting a real application to GPU(s) requires some hard work
    - Amdahl says you need to port a lot of the profile to see a speed-up
      - bad news: to see 10x speedup, need to port at least 90% of the application profile
      - good news: if profile very peaked, 90% of time may be spent in, say, 40% of code
    - even before you worry about data transfers

- **A good programming model and environment**
  - bridges the gap between peak performance and achievable performance

- **Directive-based programming can help a lot**
  - there is still much work to do, but you can spend more time thinking and less time on the donkey work

- **The good news:**
  - A lot of the work done in an OpenACC port increases performance on the CPU as well

# Strategic risk factors

- **Will there be machines to run my OpenACC code on?**
  - Now? Lots of Nvidia GPU accelerated systems
    - Cray XK7s: CSCS tödi, HLRS hermit, ORNL titan...
    - Lots of other GPU machines in Top100 (OpenACC is multi-vendor)
  - Future? OpenACC can be targetted at other accelerators
    - PGI and CAPS already target Intel Xeon Phi, AMD GPUs
  - Plus you can always run on CPUs using same codebase
- **Will OpenACC continue?**
  - Support? Cray and PGI (at least) are committed to support OpenACC
    - Lots of big customer pressure to continue to run OpenACC
  - Develop? OpenACC committee now finalising v2.0 of standard
    - Lots of new partners joined committee  at end of last year
- **Will OpenACC be superseded by something else?**
  - Auto-accelerating compilers? If only!
    - Never really managed it for threading real HPC applications on the CPU
    - Data locality adds to the challenge
  - OpenMP accelerator directives? OpenACC work not wasted
    - Very similar programming model; can transition when these release if wish
    - Cray (co-chair), PGI very active in OpenMP accelerator subcommittee