Hardware and software trends in computing

Mike Giles

Oxford University Mathematical Institute

Module 6

Nov 29, 2017

Outline

Hardware:

- money + economics
- Moore's Law
- parallelism + vector processing

Software:

- OpenMP parallelism
- OpenMP vectorisation

Money and economics

Money and economics drive the computing industry – technology is just the way things get done!

Money: if there's a big enough market demand, someone will develop the technology / product.

Economics: designing a new chip is very expensive ($1bn?) so it's really important to minimise unit cost by producing lots of the same item.

A new fabrication plant is even more expensive (multi-$bn's) so there are now only 4 big fab companies doing the manufacturing: Intel, Samsung, TSMC, GlobalFoundries (ex-AMD + ex-IBM).

Computing markets

The computing market has several distinct categories:

- hyperscale servers (100k+ for Google, Amazon, Microsoft, Facebook, eBay, Baidu, Alibaba, TenCent) – historically dominated by Intel, but under threat from NVIDIA and perhaps ARM
- servers (10–20k) – historically dominated by Intel, but under threat from NVIDIA and perhaps ARM
- desktop (classic PC) – dominated by Intel
- mobile (phones + devices + cars?) – dominated by ARM, Qualcomm, Apple, Samsung but Intel and NVIDIA entering
- embedded (IoT) – dominated by ARM, Qualcomm

Big market forces

- AI / Deep Learning / Machine Learning applied to everything

  – this has two sides to it (training and execution) with slightly different needs

- Computing into everything – Internet of Things (IoT)

- Big focus on energy efficiency

Some market numbers

- Intel: stock up 30% in the past 3 years (mostly in last 2 months), market cap is now $210bn
- NVIDIA: stock up by factor 10 in 3 years, market cap is now $130bn
- Broadcom: stock up by 100% in the past 1.5 years, market cap $110bn
- Qualcomm: stock down 10% in the past 3 years, market cap $95bn
- ARM: bought by SoftBank a year ago for $32bn
- AMD: stock up by factor 3-4 in 3 years, market cap is now $10bn

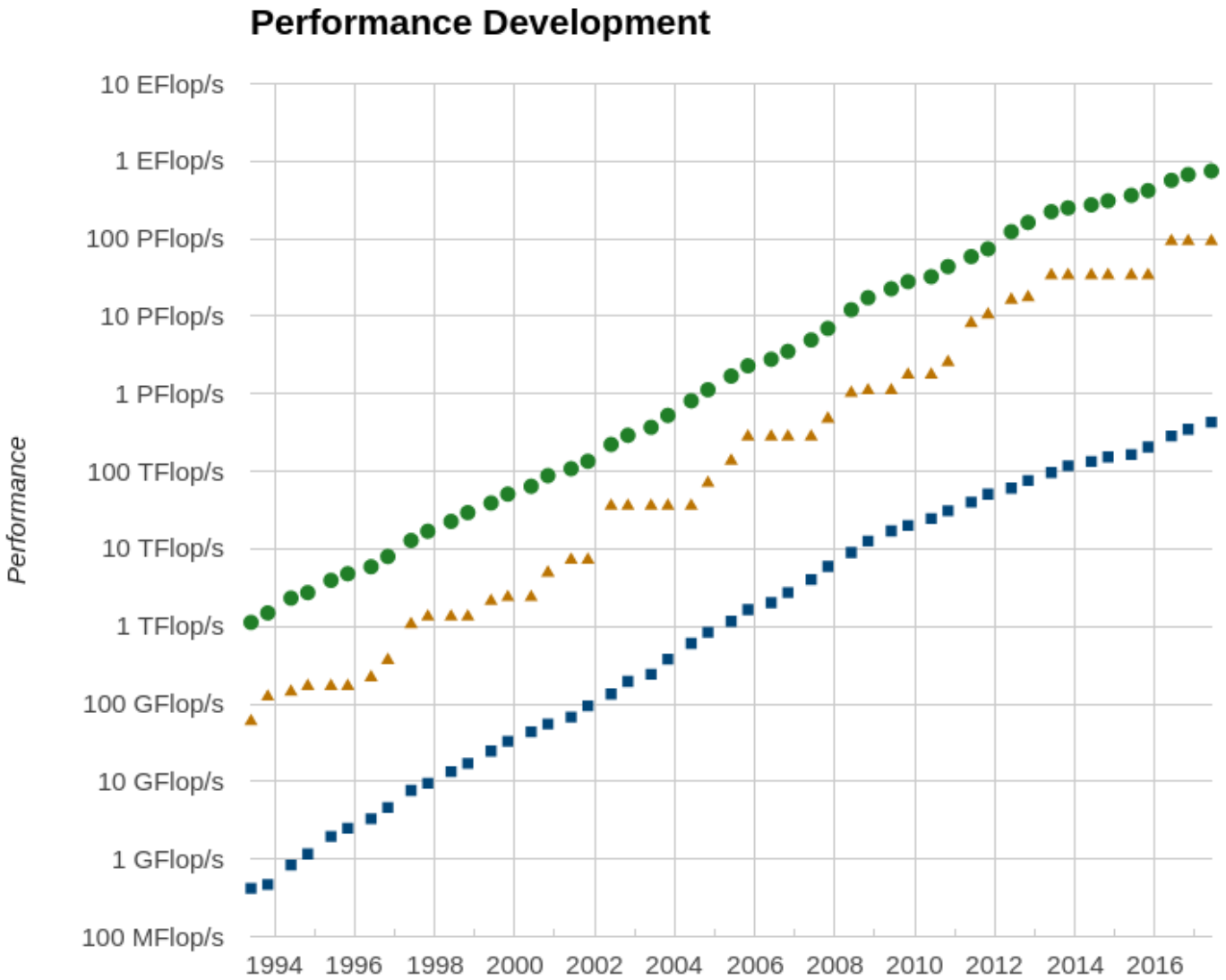
- Apple: stock up 60% in the past 3 years, market cap now $900bn
- Samsung: stock up 100% in the past 3 years, market cap $320bn
- IBM: market cap is now $140bn

NVIDIA QUARTERLY REVENUE TREND

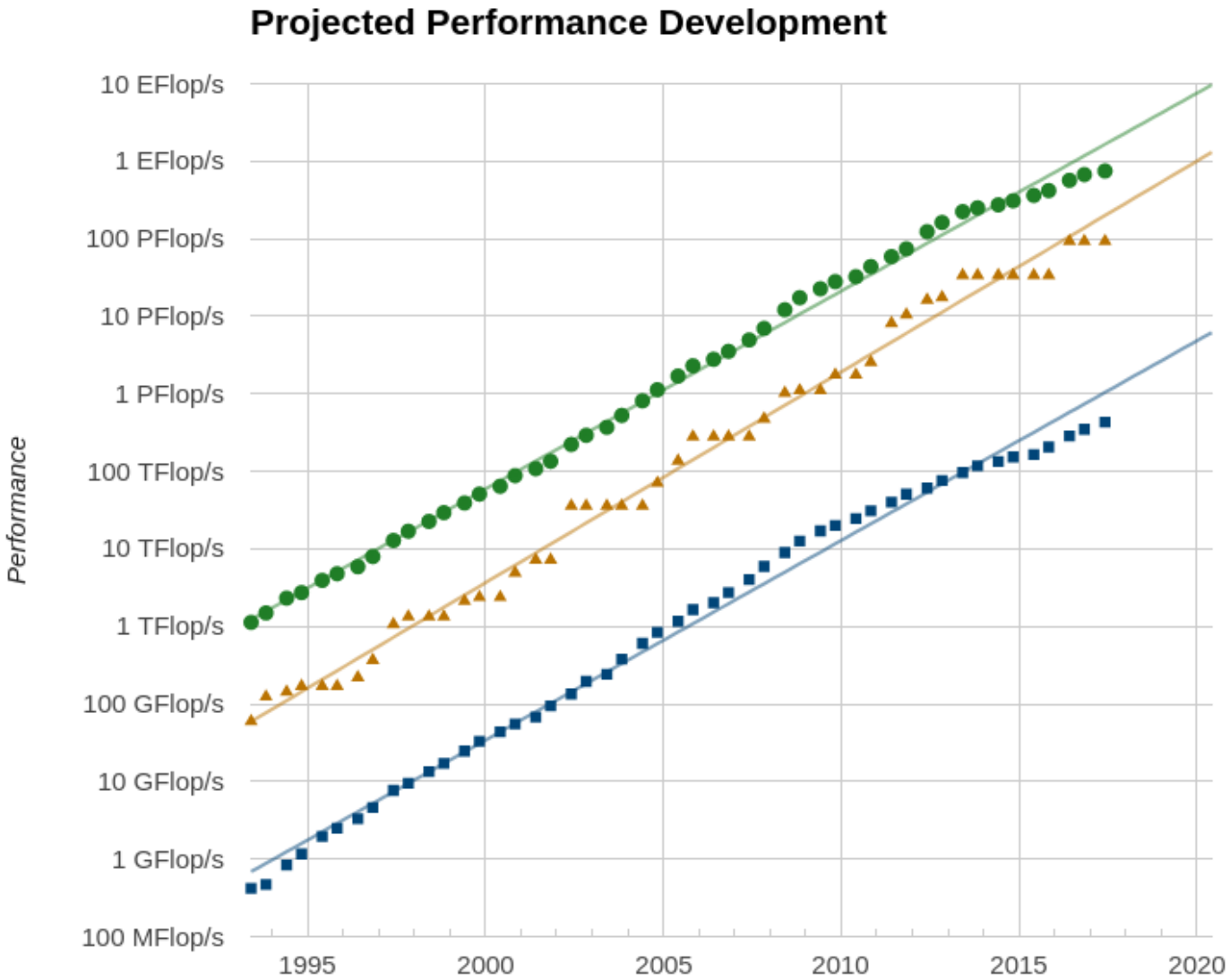
Revenue by Markets

($ in millions)	Q4 FY16	Q1 FY17	Q2 FY17	Q3 FY17	Q4 FY17	Q1 FY18	Q2 FY18	Q3 FY18
Gaming	$ 810	$ 687	$ 781	$ 1,244	$ 1,348	$ 1,027	$ 1,186	$ 1,561
Professional Visualization	203	189	214	207	225	205	235	239
Datacenter	97	143	151	240	296	409	416	501
Auto	93	113	119	127	128	140	142	144
OEM & IP	198	173	163	186	176	156	251	191
Total	**$ 1,401**	**$ 1,305**	**$ 1,428**	**$ 2,004**	**$ 2,173**	**$ 1,937**	**$ 2,230**	**$ 2,636**

Top500 supercomputers



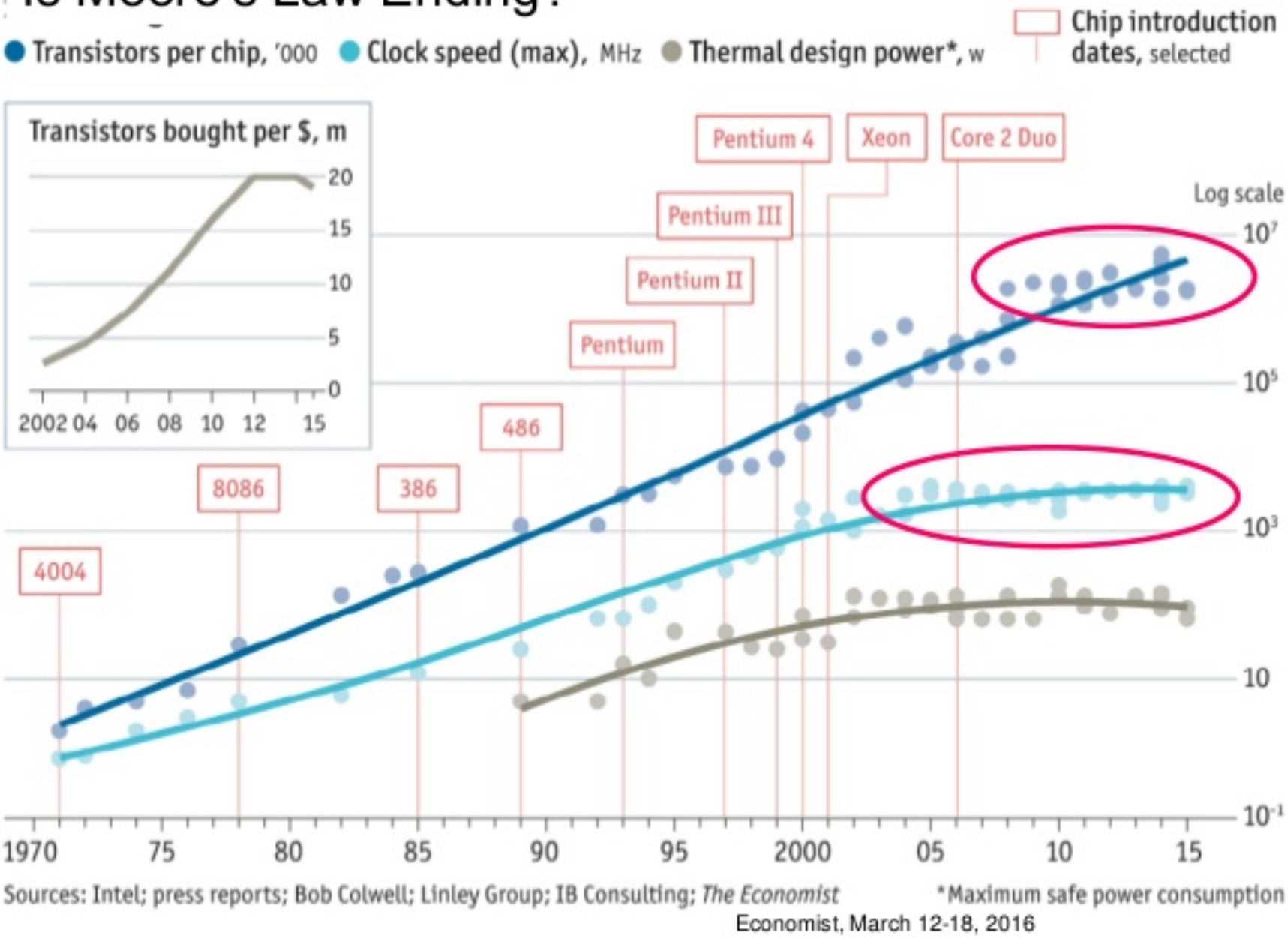
Top500 supercomputers



Top500 supercomputers

- very impressive growth, $10^5 \approx 2^{17}$ increase in 20 years, doubling roughly every 15 months
- seems to be slowing down in last few years
- power consumption of #1 system has also grown hugely:
  - 1977, Cray-1, 115kW
  - 1985, Cray-2, 150-200kW
  - 1993-6, Japanese Numerical Wind Tunnel, 500kW
  - 2005, IBM Blue Gene/L, 716kW
  - 2014, Tianhe-2, Intel Xeon Phi, 17MW
  - 2017, Sunway TaihuLight, 15MW

Is Moore's Law Ending?



Economist, March 12-18, 2016

Increasing parallelism

Chips typically have clock frequencies of 1-3 GHz, but the top GPU deliver up to 100 TFlops, and many CPUs can deliver over 1 TFlops – this is all due to parallelism at many levels:

- single instruction (addition or multiplication)
- instruction pipeline
- multiple pipelines (superscalar)

- vector units
- multiple cores
- server with multiple chips/sockets
- multiple servers

Vector processing

- don't want lots of chip dedicated to "command & control" – instead, cores work in small groups, all doing the same instruction at the same time, but on different data

  (similar to old days of vector computing on CRAY supercomputers)

- on NVIDIA GPUs, cores work in groups of 32 (a thread *warp*)

- CPUs also have vector units which are getting longer to compete with GPUs – old Intel Xeons have 256-bit AVX vector units, but the latest Xeons have 512-bit AVX-512 units (8 doubles or 16 floats), and each core has up to 2 of these

- tricky for algorithms with lots of conditional branching, but there are various algorithmic tricks that can be used

Data movement

Moving data to/from the main memory often limits execution performance

- 200-600 cycle delay in fetching data from main memory
- many applications are bandwidth-limited, not compute limited

  (in double precision, given 200 GFlops and 80 GB/s bandwidth, needs 20 flops/variable to balance computation and communication)
- takes much more energy / time even to move data across a chip than to perform a floating point operation
- often, true cost should be based on how much data is moved, and this is becoming more and more relevant over time
- in some cases, this needs a fundamental re-think about algorithms and their implementation

Latest GPU

NVIDIA's V100 GPU:

- 80 SM (streaming multiprocessor) units running at around 1.5 GHz
- each SM has
  - $64 = 2 \times 32$ single precision cores
  - 32 double precison cores
  - 64 32-bit integer cores
  - 8 special reduced-precision tensor cores for machine learning
  - 64k 32-bit registers
  - up to 96kB shared memory
- 16 TFlops in single precision / 8 TFlops in double precision
- 125 TFlops in reduced-precision machine learning application
- 900 GB/s bandwidth to 16GB HBM2 memory

Latest CPUs

New Intel Xeon Scalable Processors:

- up to 28 cores, each with either 1 or 2 AVX-512 vector units
- 1MB of L2 cache per core, and up to 38.5MB of shared L3 cache
- the most expensive costs $13k, more than an NVIDIA V100 GPU
- up to 120 GB/s bandwidth to main memory

Xeon Gold 6140:

- `https://en.wikichip.org/wiki/intel/xeon_gold/6140`
- $2.5k cost
- 18 cores, each with 2 AVX-512 units
- 2.3GHz, with single-core boost to 3.7GHz

Latest CPUs

Xeon Gold 6140:

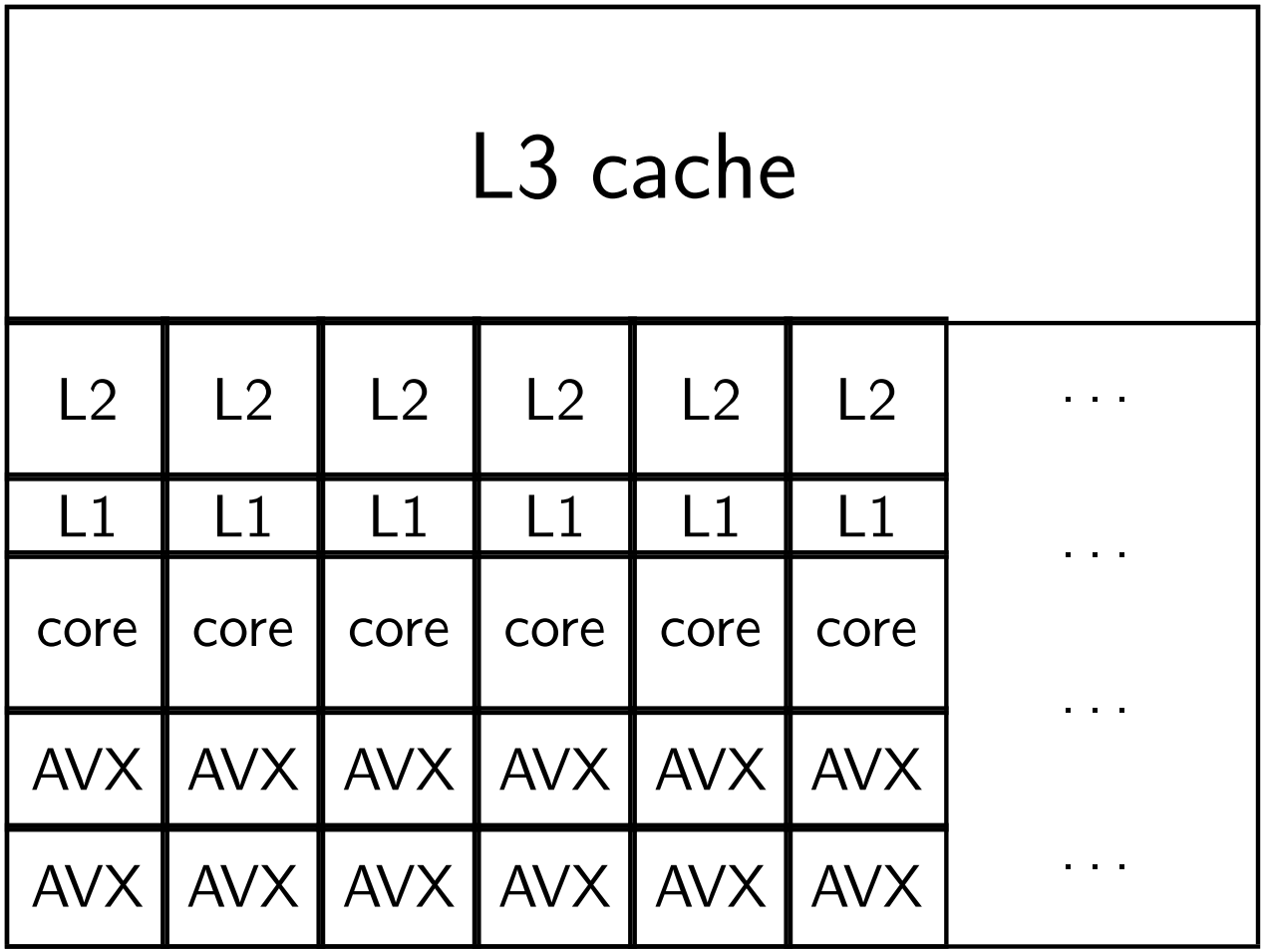
- `https://en.wikichip.org/wiki/intel/xeon_gold/6140`
- $2.5k cost
- 18 cores, each with 2 AVX-512 units
- 2.3GHz, with single-core boost to 3.7GHz

Importance of vectorisation:

- without, peak single precision performance is $18 \times 4 \times 2.3 \times 10^9 \approx 165$ GFlops
- with, peak single precision performance is $18 \times 2 \times 32 \times 2.3 \times 10^9 \approx 3$ TFlops

Latest CPUs

Xeon Gold 6140 structure: L1 (32kB), L2 (1MB), L3 (1.375MB/core)



Operating System Scheduler

Usually, there are many more jobs/tasks/processes running than available cores/threads, and hence a scheduler is required:

- it maintains a list of active processes
- in the simplest case, each process is allowed to run for 1ms then it is put to the back of the queue to wait for its next turn
- by default, Ubuntu uses the Completely Fair Scheduler (CFS): `https://en.wikipedia.org/wiki/Completely_Fair_Scheduler`
- important consequence: when the process gets its next turn, it may end up on a different core, so all of its data will have to be moved over from one L1/L2 cache to another – costly!
- alternatively, a process can insist that each of its threads always goes to the same core ("pinned" threads)

Software

What does a programmer need to know about developing high performance software?

Given what has been said about the hardware, the software needs to exploit parallelism, but there are lots of ways in which this can be done:

- task parallel: run lots of independent tasks at the same time – "trivial" but can be very effective
- multi-process: a task consists of several processes (on same system or multiple systems) which exchange information; usually implemented using MPI – classic HPC
- multi-threaded: a process uses multiple threads on a shared-memory multicore system – usually done using OpenMP or Posix threads
- vectorization: each thread exploits the AVX vector capabilities for maximum performance – can also be done using OpenMP

OpenMP

Strong view: expertise in using OpenMP is essential for a software developer interested in performance!

- simplest way to achieve both multi-threading and vectorization
- relatively easy to develop a parallel code
- BUT, must understand what is going on to get good performance (quite easy to get poor performance through "simple" mistakes)

OpenMP

OpenMP is an open standard and most compilers support it:

- defined for C/C++ and Fortran (the main HPC languages) it works mainly through "pragmas" (special comments within the code) and a header file `<omp.h>`
- there is also a run-time OpenMP library, and at run-time the OpenMP code checks a variety of environment variables
- gcc/g++: enabled by `-fopenmp` compiler flag
  – for environment variables see
  `https://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html`
- icc/icpc: enabled by `-qopenmp` compiler flag
  – for environment variables see
  `https://software.intel.com/en-us/node/522775`
  and look for OMP_* and KMP_*

OpenMP

Simple task parallelism within a single process:

```
#pragma omp parallel
  {
    // Code inside this region runs in parallel.
    printf("Hello! \n");
  }
```

How is the number of threads determined?

- function call `omp_set_num_threads`
- environment variable OMP_NUM_THREADS
- system default (often the number of cores)

OpenMP

More details:

- each thread can identify itself using function call `omp_get_thread_num`, and use this "rank" to decide what to do
- each thread will run on a different core, with its own L1/L2 cache
- any variables declared within the parallel region will be "private", i.e. each thread will only be able to read/write to its own variables
- any variables declared before the parallel region will be "shared", i.e. all threads can read/write to them – reading is usually not a problem, but writing can be a major problem if not done carefully

Writing to the same shared variable has unpredictable consequences; writing to different shared variables may give bad performance if they share a cache line. Write to private variables as much as possible.

OpenMP

A parallel `for` loop:

```
#pragma omp parallel for
  for (int i=0; i<1000; i++) {
    b[i] = (a[i] + a[i-1]) / 2.0;
  }
```

- `a, b` are shared
- `i` is private by default
- if there are 4 threads, then thread 0 will do first 250, thread 1 next 250, and so on
- the compiler will check that this is logically OK, that `i=0` can be done at the same time as `i=250` without affecting the final answer

OpenMP

The way in which iterations of a loop are allocated to threads can be controlled by a "clause":

- `#pragma omp parallel for schedule(static)`
  the default block schedule, good in most cases
- `#pragma omp parallel for schedule(dynamic)`
  the "cheese counter" scheduler: each thread does one, then gets a new one (good for small numbers with highly variable workloads)
- `#pragma omp parallel for schedule(dynamic,chunk)`
  similar, each does `chunk` iterations, then goes back for next chunk
- for more information on these and other options see
  `https://software.intel.com/en-us/articles/openmp-loop-scheduling`

Remember that each thread runs on a different core, so the schedule choice has implications for data movement and performance.

OpenMP

In many applications (e.g. Monte Carlo) we have code like this:

```
  float sum = 0.0f;
  for (int i=0; i<1000; i++) {
    float x = value_calc(i, ...);
    sum    += x;
  }
```

A naive use of OpenMP wouldn't work, because all threads would be updating the same sum. Instead, we have a special clause to handle it.

```
  float sum = 0.0f;
#pragma omp parallel for reduction(+:sum)
  for (int i=0; i<1000; i++) {
    float x = value_calc(i, ...);
    sum    += x;
  }
```

OpenMP

The compiler implementation of this first uses a private `sum_local` for each thread, then adds these onto `sum` once the loop is finished.

Other possible reduction operators include: *, - max, min and various logical operations.

For more details see:
`https://software.intel.com/en-us/node/608161`
or
`https://computing.llnl.gov/tutorials/openMP/#REDUCTION`

OpenMP

So far we have talked about multi-threaded parallelism.

OpenMP 4.0 expanded to include vectorisation (a.k.a. SIMD – Single Instruction, Multiple Data) although the Intel compiler may automatically vectorise the innermost loop with compiler flags `-O3` or `-vec`.

```
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int i=0; i<1000; i++) {
    float x = value_calc(i, ...);
    sum    += x;
  }
```

On AVX-512, with 16 floats per vector, it will do the loop in chunks of 16, and then cope with the remaining 8 at the end. The reduction is handled by using a vector of partial sums, then combining them at the end.

OpenMP

In this example, it would generate a vectorised version of the function call `value_calc`.

Remember: vectorisation is performed at the level of a single thread within a single core, so this should usually be innermost when there are nested loops.

Some slides:

https://doc.itc.rwth-aachen.de/download/attachments/28344675/SIMD%20Vectorization%20with%20OpenMP.pdf

https://www.nersc.gov/assets/Training-Materials/NERSC-VectorTrainingOct2014.pdf

OpenMP

Finite difference code example:

```
#pragma parallel for
  for (int j=0; j<J; i++) {
#pragma omp simd
    for (int i=0; i<I; i++) {
      arr[i + j*I] = ...;
    }
  }
```

Very important to order the loops correctly – the elements of `arr` are sequential in the `i` direction, so keeping the `i` loop innermost means the data is already contiguous for vectorisation **and** the different threads for the `j` loop are working on quite different parts of the array.

OpenMP

An experimental Monte Carlo test code to investigate OpenMP:
`http://people.maths.ox.ac.uk/gilesm/mlmc/c++/par_expts/GBM.c`
(slightly simplified version shown here)

```
#include <omp.h>
// vector length for vectorisation
#define VECTOR_LENGTH 8

// each OpenMP thread has its own VSL RNG and storage
double              *dW;
VSLStreamStatePtr stream;
#pragma omp threadprivate(stream, dW)
```

The last line ensures a private random number generator for each thread.

OpenMP

Top level multi-threaded parallelisation:

```
#pragma omp parallel shared(T,X0,mu,sigma,dt,M,N) \
                     reduction(+:sum1,sum2)
  {
    double sum1_t = 0.0, sum2_t = 0.0;
    int    tid    = omp_get_thread_num();
// create RNG, then give each thread a unique skipahead
    vslNewStream(&stream, VSL_BRNG_MRG32K3A,1337);
    long long skip = ((long long) (tid+1)) << 48;
    vslSkipAheadStream(stream,skip);
    dW = (double *)malloc(RV_BYTES);
    pathcalc(T,X0,mu,sigma,dt, M, N, &sum1_t, &sum2_t);
    sum1 += sum1_t;
    sum2 += sum2_t;
  }
```

OpenMP

Inside pathcalc

```
  /* loop over paths in increments of VECTOR_LENGTH */
  for (int n1=0; n1<N; n1+=VECTOR_LENGTH) {
    int offset = n1*M; /* random numbers already used */
    /* vectorised path calculation */
#pragma omp simd reduction(+:sum1,sum2)
    for (int n2=0; n2<VECTOR_LENGTH; n2++) {
      double X = X0;
      for (int m=0; m<M; m++) {
        double delW  = dW[offset+n2];
        X = X*(1.0 + mu*dt + sigma*delW);
      }
      sum1 += X; sum2 += X*X;
    }
  }
}
```

Compiler swaps innermost two loops to achieve vectorisation.

OpenMP

Finally, remembering the operating system scheduler, we don't want threads to hop around between different cores.

This is accomplished by "thread pinning" (or setting "thread affinity").

The OpenMP environment variable setting:
`OMP_PROC_BIND=true`
says that threads are to be bound to cores, i.e. not moved

The OpenMP environment variable `OMP_PLACES` can be used to specify which core each thread runs on.

Alternatively, the Intel compilers have a different environment variable:
`KMP_AFFINITY=type`
where type is one of `compact, scatter, explicit, ...`

OpenMP

With `compact` it tries to keep threads close together physically
one thread to core 0 in socket 0, then
one thread to core 1 in socket 0, then
one thread to core 2 in socket 0, . . . , then
one thread to core 0 in socket 1, . . .
– good when not using hyperthreading, but possibly bad with hyperthreading where you might want just one thread per physical core

With `scatter` I think it will assign
one thread to core 0 in socket 0, then
one thread to core 0 in socket 1, then
one thread to core 1 in socket 0, then
one thread to core 1 in socket 1, . . .
– good for hyperthreading, but maybe not good if neighbouring threads need to work on some neighbouring shared data, since that implies data traffic between the two sockets.

OpenMP

There is lots more to learn, but fortunately there's a lot of good information available:

- Intel: OpenMP support in C++ compiler 18.0

  `https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference-openmp-support`

- LLNL (Lawrence Livermore National Labs) tutorial:
  `https://computing.llnl.gov/tutorials/openMP/`

- MIT Press book: Using OpenMP (published in 2008 so possibly a bit dated – e.g. might not have new `simd` features)
  `https://mitpress.mit.edu/books/using-openmp`

- if you get the opportunity, you should take a course – 2 days is maybe about the right length

Final comments

- hardware is continuing to evolve rapidly, driven by applications such as machine learning
- parallelisation and vectorisation are critical to achieving good performance
- OpenMP is the simplest way to do both on shared-memory systems, and you can achieve good performance if you think clearly about the implications – i.e. how it will be implemented by the compiler
- alternatively, consider CUDA programming on NVIDIA GPUs