Background on HPC for Statistics

Louis J. M. Aslett (aslett@stats.ox.ac.uk)

Department of Statistics, University of Oxford

Young Researchers' Meeting
20 Ocotber 2015: University of Warwick

www.louisaslett.com

Background

Introduction

High performance computing is becoming increasingly important in both applied statistics and statistical methodology.

- Scaling existing methods to larger and more complex applications;
- Developing new methods which are amenable to scaling within the contraints that exist in modern HPC

Fundamentally it comes down to parallelism, which can be exploited using different (or ideally all) technologies:

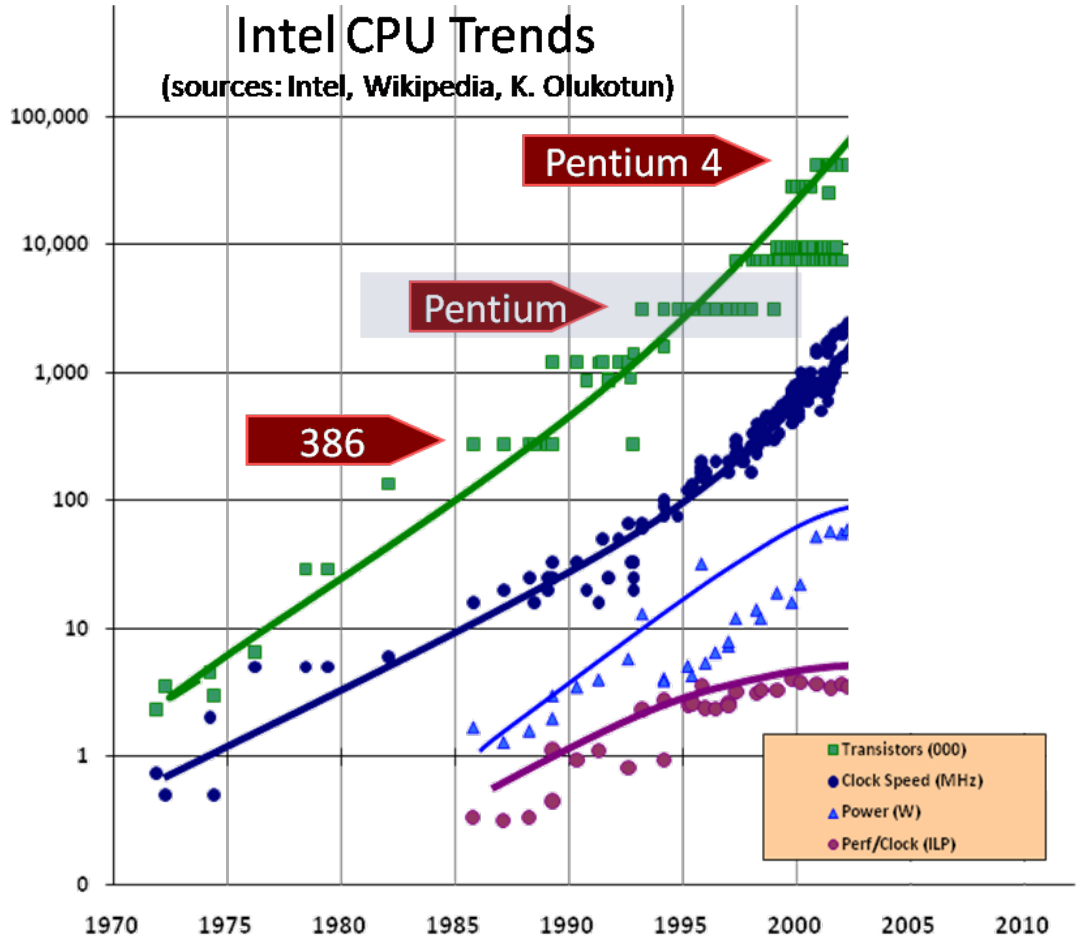
- CPU
- **GPU**
- Cluster
- **Cloud**

Bayesian statistics & computing

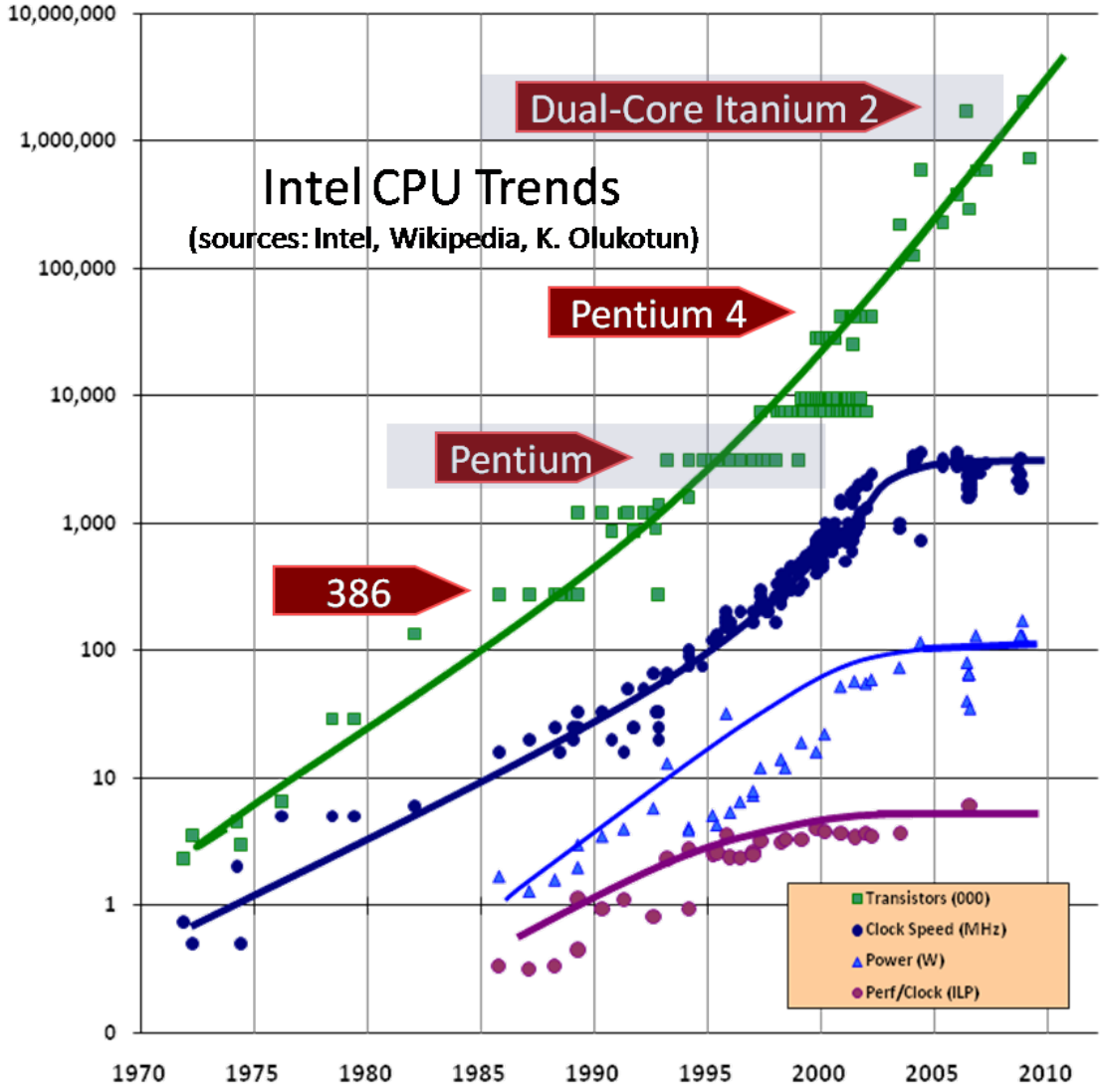
- 1970s reliance on conjugacy results abounds (e.g. skim classic Box and Tiao 1973)

Bayesian statistics & computing

- 1970s reliance on conjugacy results abounds (e.g. skim classic Box and Tiao 1973)
- 1990s MCMC techniques go mainstream opening up all sorts of models (sampler slow? Just wait a year!)



‘The free lunch is over’ — Herb Sutter



‘The free lunch is over’ — Herb Sutter

Bayesian statistics & computing

- 1970s reliance on conjugacy results abounds (e.g. skim classic Box and Tiao 1973)
- 1990s MCMC techniques go mainstream opening up all sorts of models (sampler slow? Just wait a year!)

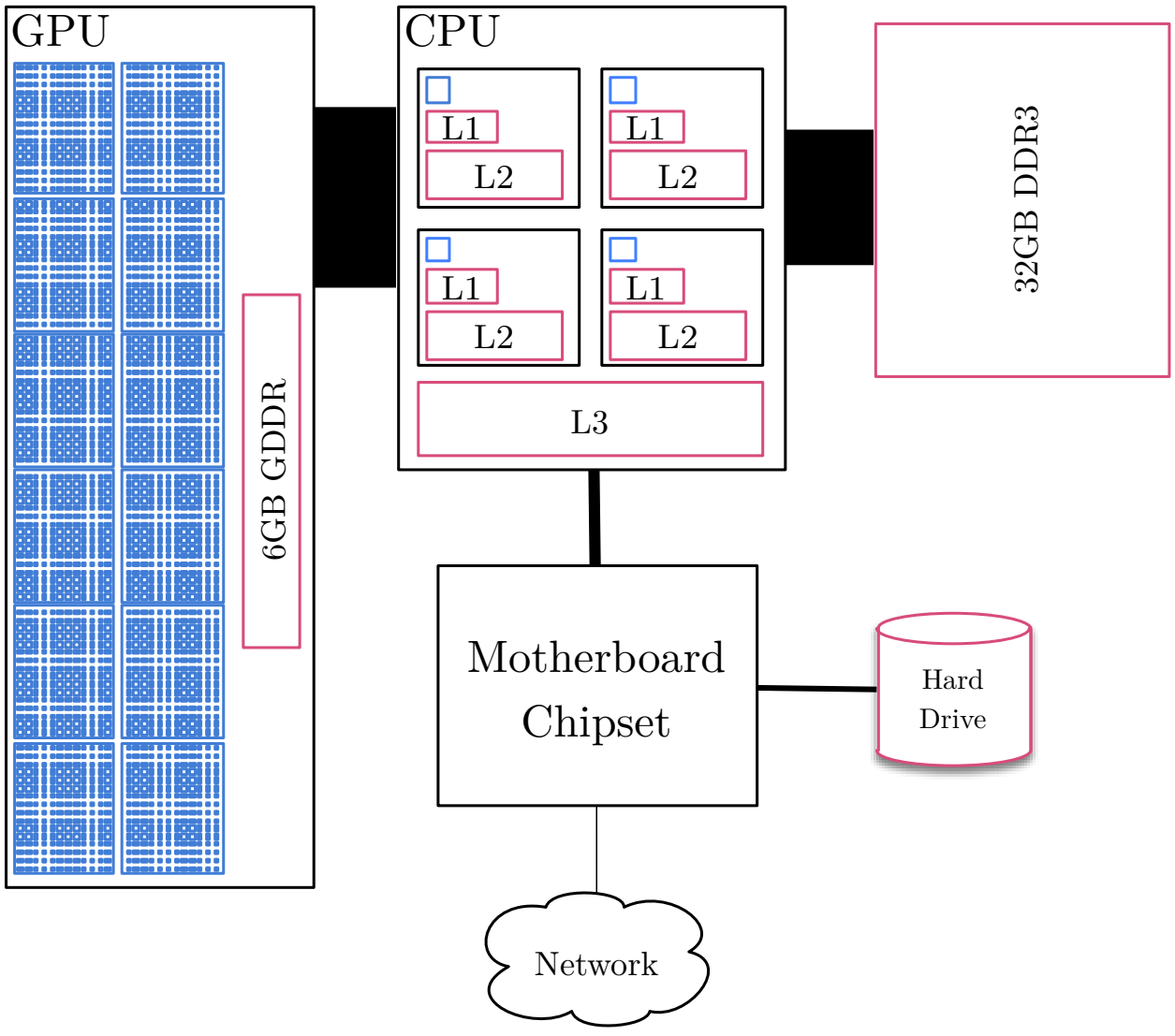
Bayesian statistics & computing

- 1970s reliance on conjugacy results abounds (e.g. skim classic Box and Tiao 1973)
- 1990s MCMC techniques go mainstream opening up all sorts of models (sampler slow? Just wait a year!)
- 2010s trying to make MCMC parallel friendly firmly embedded as an important research direction
  - July 2006 Intel ship first desktop class dual core CPU
  - August 2006 Amazon EC2 launches as a public beta (production in 2008)
  - November 2006 nVidia announce CUDA, first ever C development environment for GPUs
  - $\approx$ 9 years later:
    - 12 core Intel Xeon CPUs (/w 30MB cache)
    - $2 \times 2,496$ core Tesla K80 GPUs (/w 12GB GDDR)
    - $\geq 50,000$ core EC2 clusters launched (/w 29TB RAM)

Background reading

- 'The free lunch is over: A fundamental turn toward concurrency in software' — `http://www.gotw.ca/publications/concurrency-ddj.htm`
- 'Welcome to the jungle' — `http://herbsutter.com/welcome-to-the-jungle/`

Massively simplified architecture



Some reading pointers

- Goldberg, D. (1991). What every ~~computer~~ scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, **23**(1), p. 5-48.
- Drepper, U. (2007). 'What Every Programmer Should Know About Memory', §1 – §5

If you *really* enjoy this stuff, the following are as detailed and advanced as it gets outside Intel & AMD:

- Fog, A. (2014). 'The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers', `http://www.agner.org/optimize/microarchitecture.pdf`
- Fog, A. (2014). 'Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs', `http://www.agner.org/optimize/instruction_tables.pdf`

General (very subjective) comments on HPC

1. "Premature optimisation is the root of all evil" — old saying, but true
2. Don't guess or even excessively trust back-of-envelope theoretical calculations about where your bottleneck is … measure it!
3. Getting extreme parallel programming right is *tough*. Create a 'master' version which is purely serial and you know works first. Should do this anyway to satisfy comment 2!
4. Never lose sight of the programmer time -vs- run time tradeoff. Some things just aren't worth the effort for the small speedup payoff! "R when you can, C when you must"
5. Take modularising your code seriously — nothing like one huge spaghetti function for killing ability to tackle complex problems.

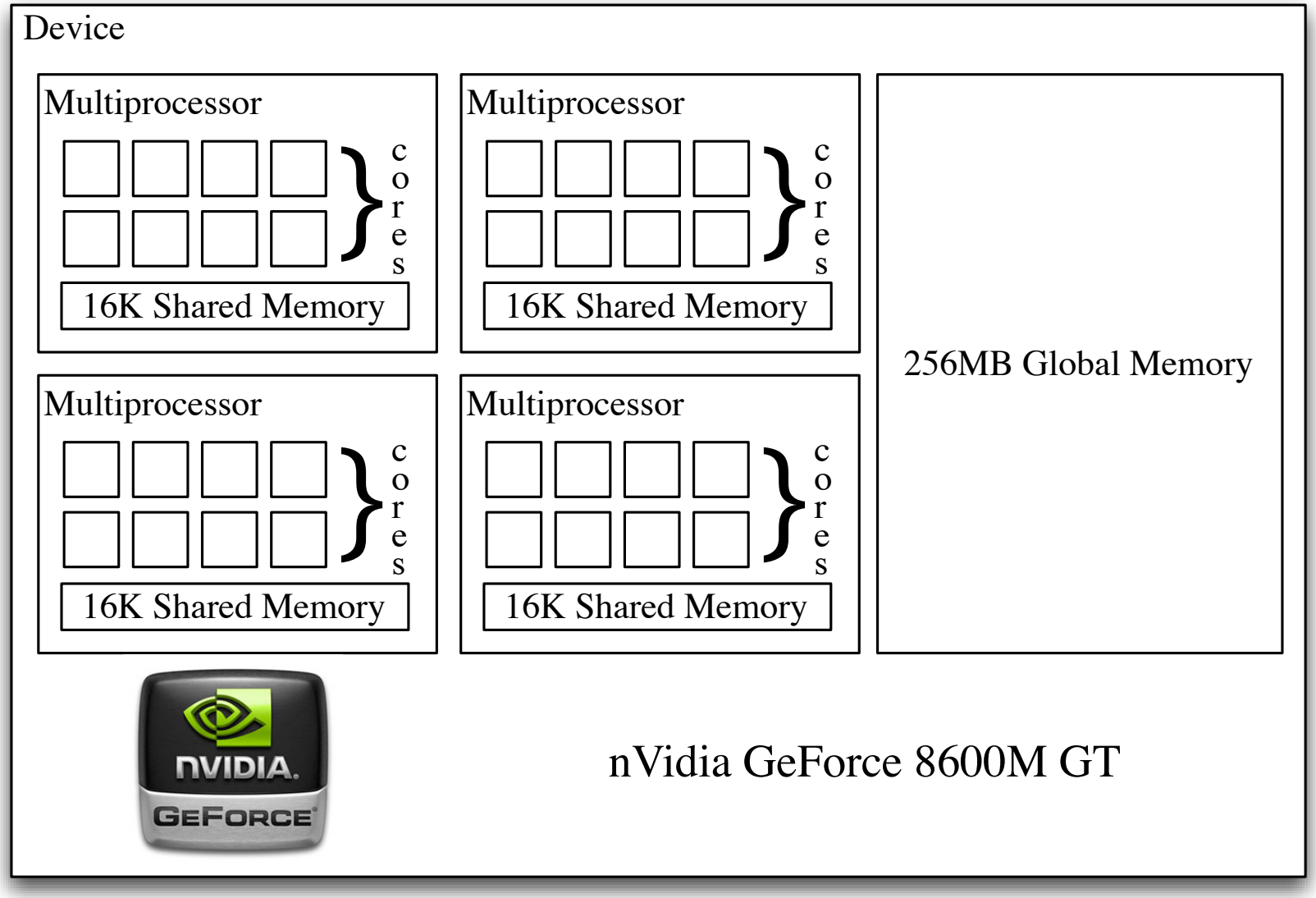
GPUs

Introduction

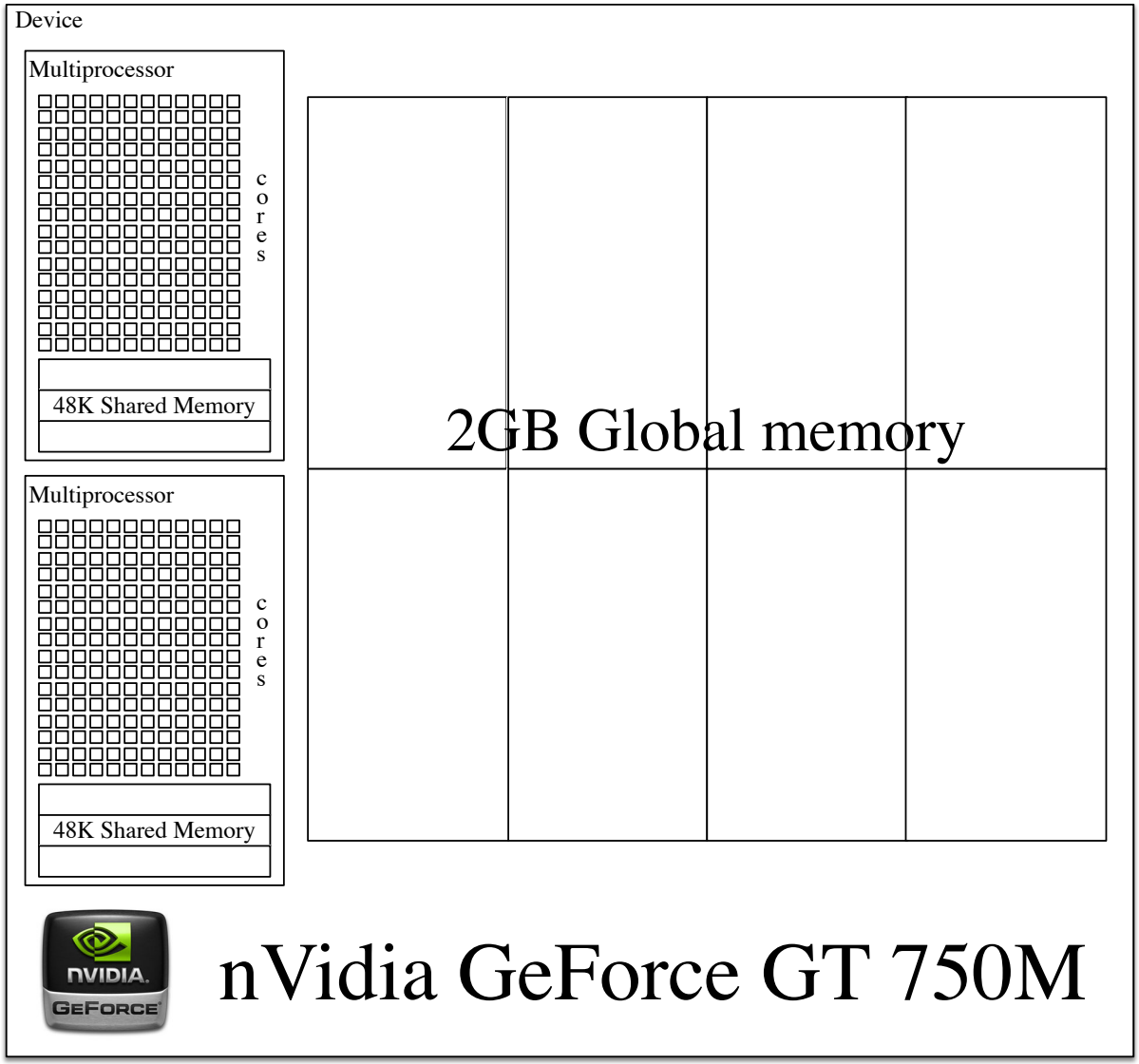
- GPUs are extraordinarily parallel devices (upto 4, 992 cores at present, Tesla K80)
- Usually programmed in C/C++ using CUDA
- Interfaces available in Python, R, Julia, ...
- Main mode of operation is SIMD
  - can now launch multiple independent kernels
- GPUs cannot directly access the system memory: you must copy data on and results off
  - CUDA 6 added 'unified memory', but this just hides what is happening anyway

Today: a code-free introduction to help you see whether your problem can map naturally onto a GPU given a few simple performance considerations.

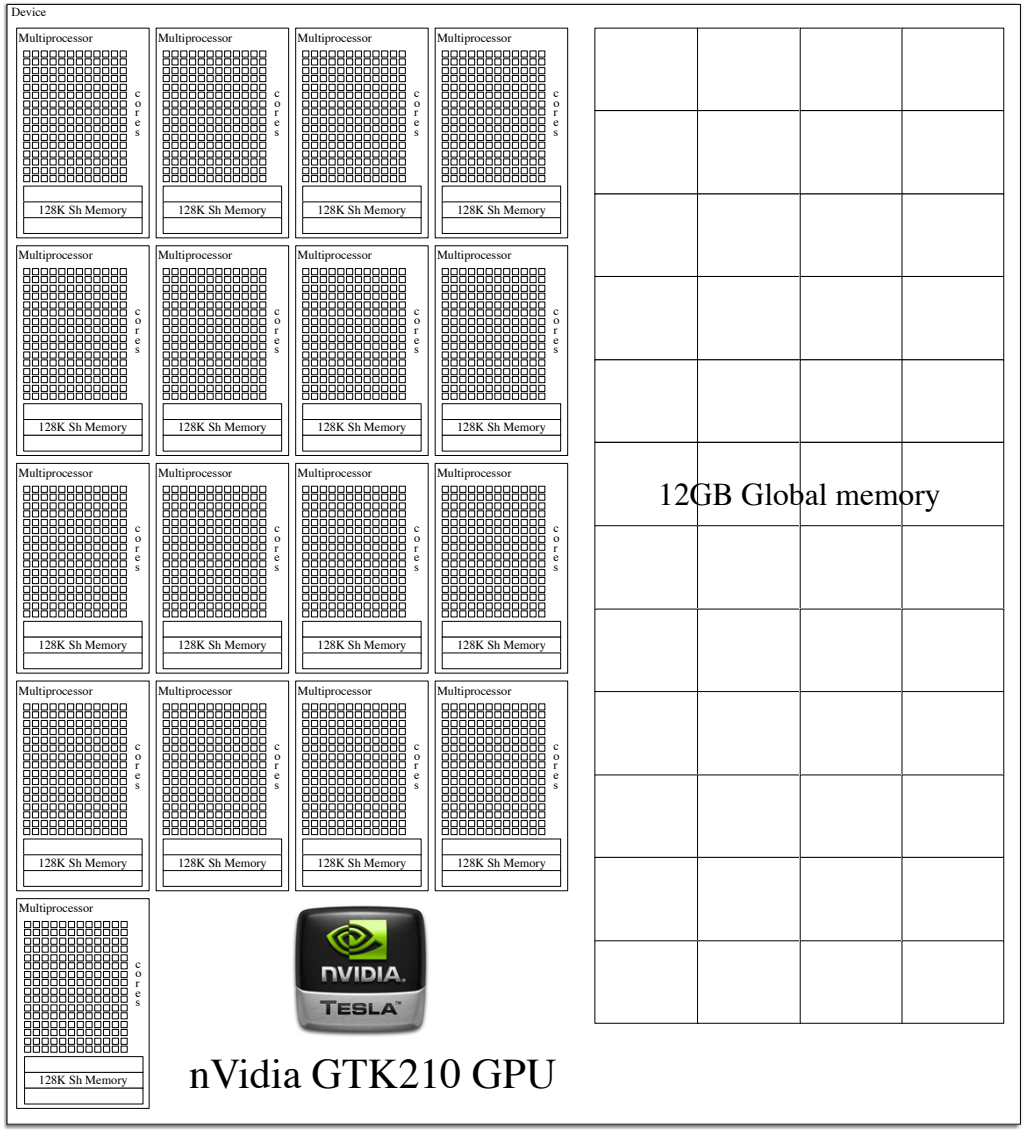
Highly simplified GPU architecture



Highly simplified GPU architecture



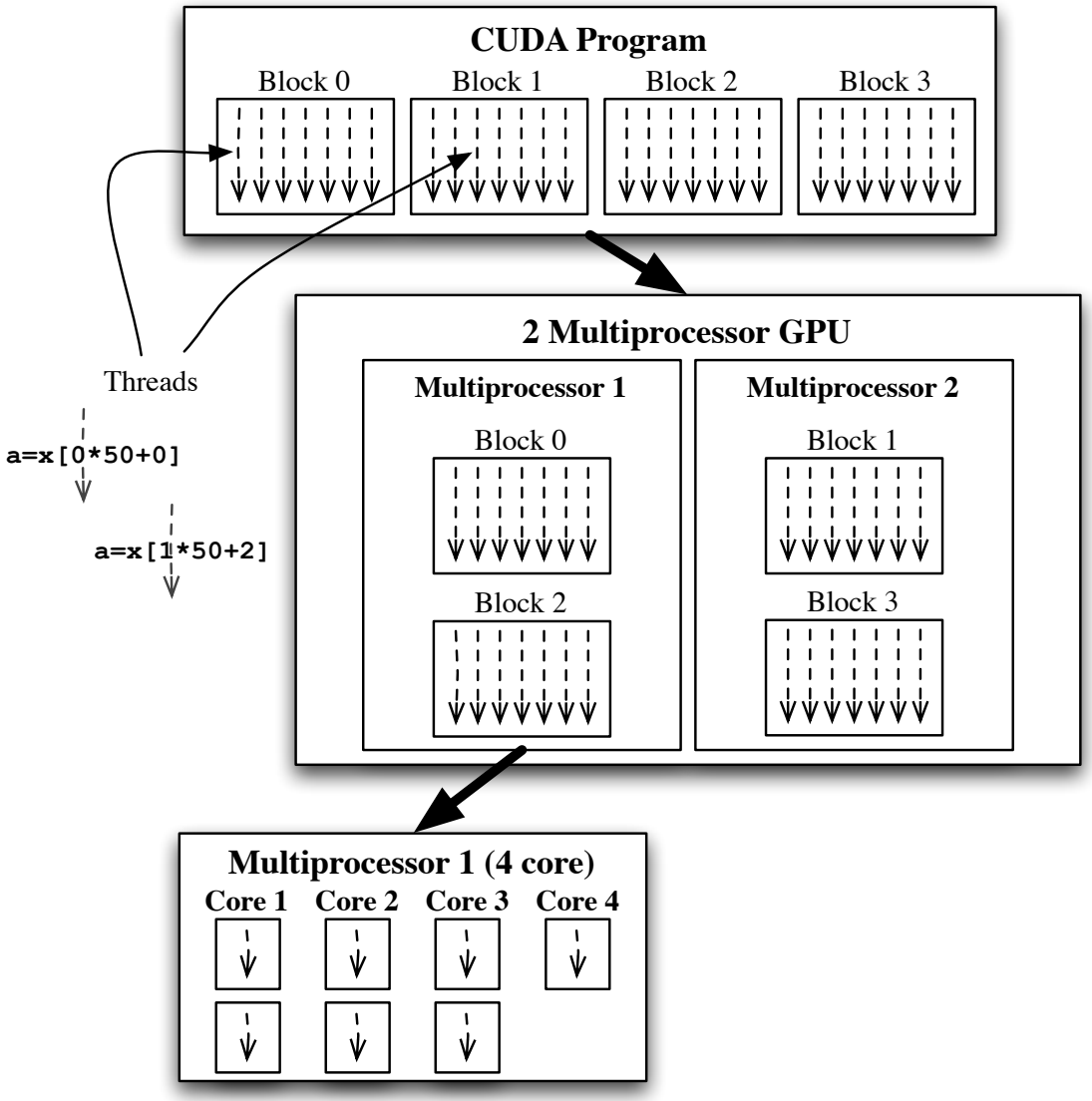
Highly simplified GPU architecture



CUDA Concepts (Oversimplified, *cf* dynamic parallelism)

- **Kernel:** a C function which is flagged to be run on a CUDA capable device
- A kernel is executed on the core of a multiprocessor inside a *thread*. A thread can be thought of as just an index $j \in \mathbb{N}$. V Loosely: a index of cores in multiprocessors
- At any given time, a *block* of threads is executed on a multiprocessor. A block can be thought of as just an index $i \in \mathbb{N}$. V Loosely: an index of multiprocessors in devices
- Together, (i, j) corresponds to exactly one kernel running on a core of a single multiprocessor.

i.e. Very simplistically speaking, think of how to parallelize your problem by how to split it into identical chunks indexed by a pair $(i, j) \in \mathbb{N} \times \mathbb{N}$


CUDA Program
Block 0
Block 1
Block 2
Block 3
2 Multiprocessor GPU
Multiprocessor 1
Multiprocessor 2
Block 0
Block 1
Block 2
Block 3
Threads
a=x[0*50+0]
a=x[1*50+2]
Multiprocessor 1 (4 core)
Core 1
Core 2
Core 3
Core 4

Some CUDA Rules

- There is a cap on the maximum number of blocks and threads (though both can – and should – exceed the physical number of multiprocessors and cores)
- Can't assume threads will complete in the order you index them.
- Can't assume blocks will complete in the order you index them.
- To deal with execution order dependency either:
  - run dependent items in the same block (`__syncthreads()`, beyond talk scope)
  - split into kernels which you call consecutively from C
- Don't write to the same memory location from different threads (proviso: shared memory, beyond talk scope)

Performance Considerations

1. Memory accesses are *slow* compared to the cores. Usually want many more total threads than cores to mask this.
2. Conditional sections of an algorithm can quickly kill performance.
3. Random or disorganised memory accesses will make a GPU under-perform a CPU!

Simple Performance Consideration #1

- Number of blocks can exceed number of multiprocessors
- Number of threads can exceed number of cores per multiprocessor

Worst case, at least both should equal the physical device sizes or else cores sit idle.

But in reality, rule of thumb is *ensure the thread figure exceeds the number of cores per multiprocessor* for performance reasons[1].

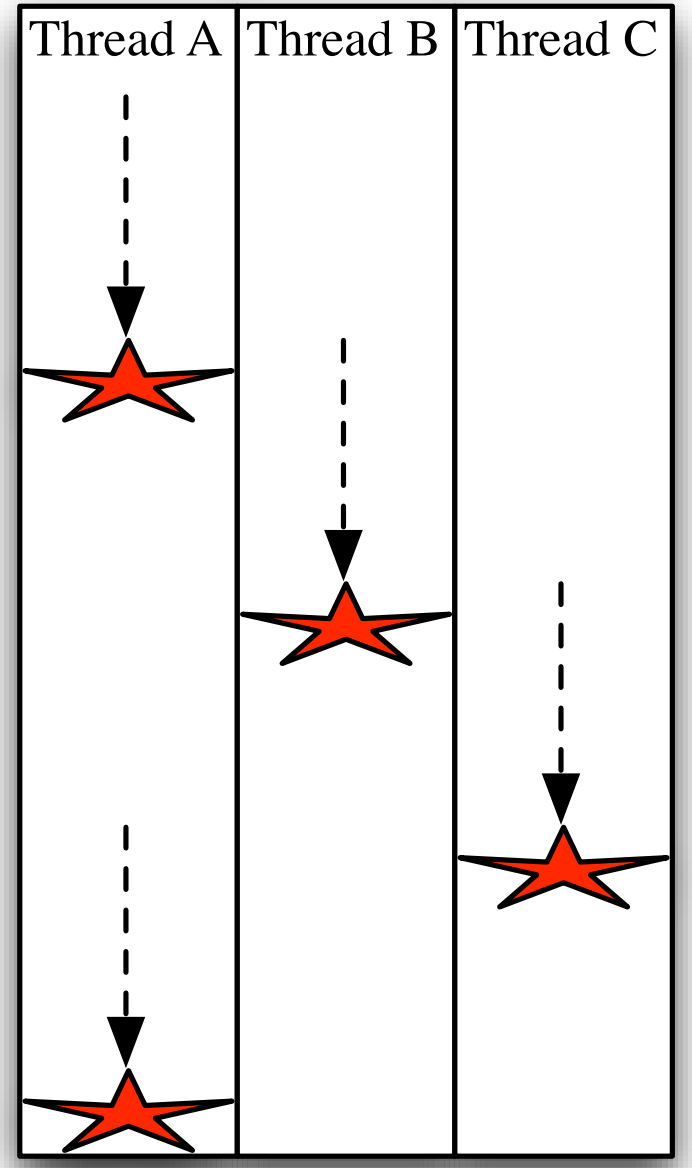
nVidia provide an 'occupancy calculator' in the form of an Excel spreadsheet which allows you to tune how many threads to choose for any given problem.
`http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

[1] this is a simplification ... ∃ occasions this is not true.

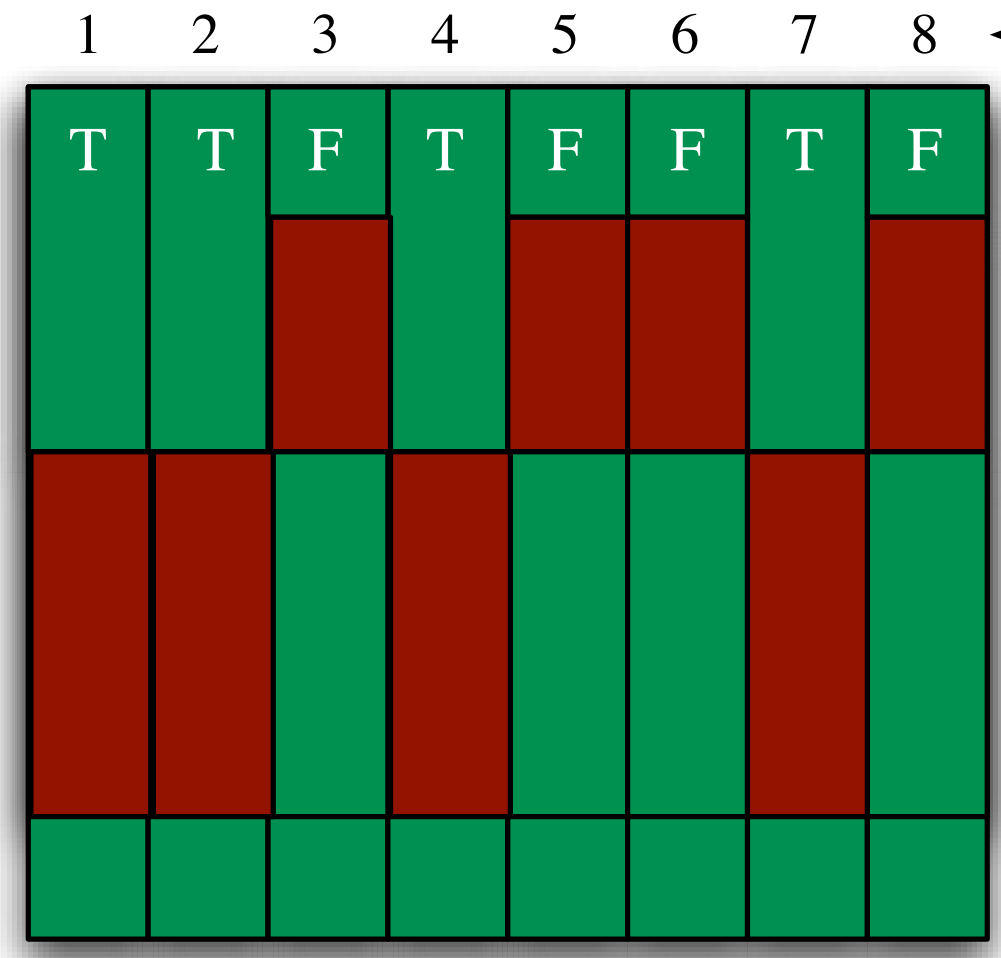
= Global memory access
=> Execution stall!

Global memory accesses are slow, so a core will stall when a request is made.

But, if # threads > # cores then another thread will be interleaved and run until the memory request is fulfilled and the first thread can run again.



Simple Performance Consideration #2



Threads execute in lock-step on the cores of a multiprocessor, so beware of very divergent code ... best to use block indices to separate highly divergent paths.

Simple Performance Consideration #3

The multiprocessors are able to pull in ranges of memory in large blocks rather than element by element as each core requires it (note also, due to the lock-step all cores will be ready for memory access at the same time).

When a floating point number is requested from memory, that number and the following 3 are loaded (128-bit memory bus) … *whether you asked for them or not!*

Simple Performance Consideration #3

The multiprocessors are able to pull in ranges of memory in large blocks rather than element by element as each core requires it (note also, due to the lock-step all cores will be ready for memory access at the same time).

When a floating point number is requested from memory, that number and the following 3 are loaded (128-bit memory bus) … *whether you asked for them or not!*

Thus, if consecutive threads require consecutive regions of memory, there are a quarter the number of memory transactions required: *coalesced* memory access.

If an algorithm requires random or disorganised memory access then this can reduce performance at least 4 fold compared to the intended GPU programming model.

i-like.org.uk

GPUs without CUDA/C/C++

Don't forget that there are many out-of-the-box solutions which enable you to leverage GPU power without writing a single line of a GPU kernel.

- R
  - gputools, gmatrix, HiPLARM
  - cudaBayesreg, WideLM, rpud (£)
  - Rth, RCUDA
- Python: theano, NumbaPro, PyCUDA, gnumpy
- C++: Thrust, NVBIO
- C: cuBLAS, cuSPARSE, cuRAND, cuDNN, cuFFT, Magma
- Diverse options like caffe (command line/C++/python/Matlab)

For more ...

This barely scratches the surface, but GPUs are arguably some of the most powerful compute devices available today and well worth the time investment (in my opinion).

To actually do direct GPU programming yourself, first learn C very well. Then, hard to beat Mike Giles summer CUDA course at University of Oxford:

`https://people.maths.ox.ac.uk/gilesm/cuda/`

£200 for academics, week long intensive course.

Cloud

Cloud — **buzz-word alert!**

By cloud, I mean an online service which allows users to create and destroy virtual servers remotely without having to worry about initial hardware and OS installation and where billing is in very small increments (e.g. hours).

There are several cloud providers, including:

- Amazon EC2
  - `http://aws.amazon.com/`
- Digital Ocean
  - `http://www.digitalocean.com/`
- Google Compute Engine
  - `http://cloud.google.com/`
- Rackspace Cloud Servers
  - `http://www.rackspace.co.uk/`
- Windows Azure VMs
  - `http://www.windowsazure.com/`

Why talk about Amazon today?

Today, AWS is the only the service which ticks all the following (subjectively) important boxes for scientific HPC, though this is a *fast* moving business:

- Repository for community development of images so users can boot ready-to-run machines with more than just bare operating system (bit like package system in R).
- A permanent storage medium for each machine which can persist independently of the running state of the server.
- Billing which suspends while the server is stopped, but where the above persistent storage remains alive.
- A free tier of instances so everyone can try it without cost.
- Everything from micro instances to the current state of the art in HPC, including machines with nVidia GPUs for CUDA support. (See also benchmarks)
- A 'stock-market' for unused compute capacity, enabling heavily discounted compute jobs.

Amazon Web Services (AWS)

Amazon used to buy in huge server capacity to keep their website up just for the Christmas shopping spree … rest of the year large parts of server farm sat mostly idle.

Launched 2006. By December 2014, $1,400,000$ servers operating in 28 data centres across 7 countries:

- Dublin, Ireland
- Frankfurt, Germany
- North Virginia, United States
- Oregon, United States
- Northern California, United States
- Singapore, Republic of Singapore
- Tokyo, Japan
- Sydney, Australia
- São Paulo, Brazil

AWS jargon

- EC2 (Elastic Compute Cloud)
  - is the service which enables launching virtual servers
- Instance
  - a virtual server running on EC2
- S3 (Simple Storage Service)
  - for resiliant storage of data independent of instances
- AMI (Amazon Machine Image)
  - a bundle of operating system and pre-loaded applications to boot on an instance
- Volume
  - a cloud ‘hard drive’ which is attached to an instance
- Spot instance
  - an ephemeral instance whose price follows the Amazon ‘stock-market’ price

Remainder of talk is a live demo

What could possibly go wrong?

- Intro to RStudio AMI
  `http://www.louisaslett.com/RStudioAMI/`
- Tour of EC2 console, including cryptographic login setup
- Bid on spot instance
- Show in action speeding up ABC for a discrete stochastic Lotka-Volterra preditor-prey model