TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

HTC with a Sprinkle of HPC
Finding Gravitational Waves with LIGO

ECSS Symposium

8/15/2017

PRESENTED BY:
Lars Koesterke

LIGO collaborators

Duncan Brown & Josh Willis

lars@tacc.utexas.edu

Progression

LIGO: Gravitational Waves

LIGO (Laser Interferometer Gravitational-Wave Observatory)

Why is this High-Throughput Computing (HTC)?

pycbc: Python Compact Binary Collision

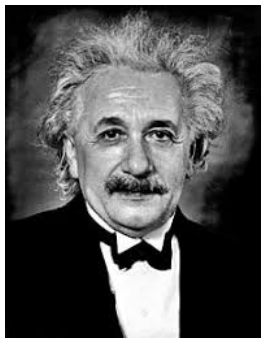
Progression in time:

- Adding HPC to HTC
- Code transformation
- Using GPU's
- Transition to Knights Landing

A little bit of Physics

how do we know that Gravitational Waves exist
… and advice for any 'betting man or woman'

Einstein
NP: 1921



Two theories from the early 20th century

1. A. Einstein (1915): General Relativity
2. E. Schroedinger and W. Heisenberg (mid- 1920's): Quantum Physics

Schroedinger
NP: 1933

Neither theory has been disproven by observations, yet

Both theories make **pretty weird predictions**

Only flaw: They don't mix and match

→ I recommend not to bet against either theory

Heisenberg
NP: 1932

Weird prediction by 'General Relativity'

Gravitational Waves

TACC

A little bit of Physics

how do we know that Gravitational Waves exist
… and advice for any 'betting man or woman'

Accelerated electrical charges (electrons, ions)

→ Electromagnetic waves

Accelerated masses

→ Gravitational waves

Coupling constant much (!!!) smaller
- → Masses have to be very large (not just a 'handful' of electrons)
- → Only waves from cosmic objects detectable
- → In this example: collision of massive compact (Black Hole, Neutron stars) objects

First 'indirect' detection in 1974:

Hulse-Taylor binary neutron star (NP: 1993)

Change of orbital period due to energy loss from 'Gravitational Waves'

LIGO measures length variation of 10^{-18} m (Size of atom: 10^{-10} m) with 4km arm length
- → Very sophisticated instrument/interferometer
- → Requires a lot of compute power to extract signal from raw data

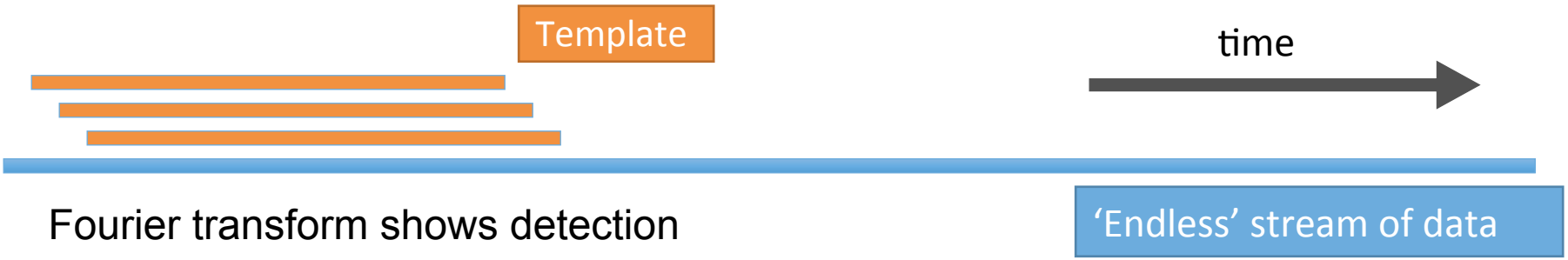
How is the Signal Detected? (1)

Signal templates from theoretical models (predictions)

Signal stream from the observatories

→ Correlation detected in Fourier space

Template ‘slides’ across the observations



Fourier transform shows detection

Why is this HTC? Large number of templates

Black hole masses/mass ratio
Spin ratio, spin axis orientation
Spin axis in relation to orbital plane
Direction of observation, etc.

How is the Signal Detected? (2)

Multi-tiered calculations

- Low-latency (quick response time) at telescope, LIGO labs, and at partner institutions
- High-latency (deep search) 'everywhere else' including XSEDE and 'Einstein@home'

pycbc is the largest (50%) consumer in the 'deep search' department (millions of SUs)

(There are literally dozens of other pipelines)

Two additional lines of work
(not discussed in this talk)

- Yaakoub "Yaakubski" El Kamra
- SDSC

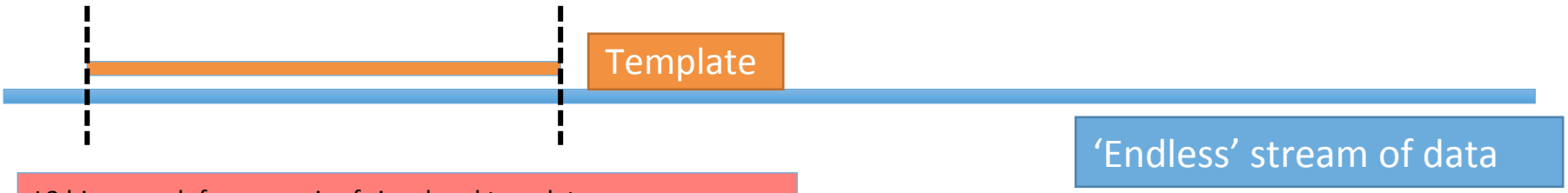
Method

- Template slides over data in small increments
- Fast-Fourier transformation at the heart of the detection
- Started as serial pipeline with single-core execution

High-Throughput Approach

1. Run many short a.out's simultaneously
2. Use the right libraries
3. Apply serial approach with 1 a.out per core unless …
    1. You run out of memory
    2. Or other concerns (in this case size of the L3 cache)

Template and signal fit neatly into the L3 cache of a CPU



L3 big enough for one pair of signal and template
L3 **not** big enough to host 8 (Stampede 1 with 8 cores per socket) datasets simultaneously

Timeline (1)

1. Serial, unoptimized
    1. Unoptimized C code
    2. Outdated libraries
    3. One **serial** a.out per **core**
    4. One **serial** fft per **core** → L3 **contention**
2. Parallel setup
    1. One **serial** a.out per **socket**
    2. 7 cores **idle** during non-fft stages
    3. **Thread-parallel** fft (fftw or MKL-fft): Speed-up for fft's > 4x (4x ~ bandwidth ratio L3 v. DDR3)
3. Code modification and parallelization
    1. One **parallel** a.,out per socket
    2. One **parallel** fft per socket
    3. Relatively small scaling losses in parallel C code
    4. Excellent overall performance

Status at the beginning
Python wrapper
Serial C code
Library calls for fft
No optimization at all

Stage 2
Leaving 7 cores idle during the C code execution
Using 8 cores during fft
→ Slower overall execution

Stage 3
After parallelizing 'host' code
using 8 cores per a.out was the fastest

Timeline (2)

Hosts are more expensive than GPUs
Pycbc cluster
Old hosts from a cluster ready for decommission

1. Use of GPUs
    1. Calculation in single precision
    2. Error correction (ECC) not needed
        1. GPUs at relatively low clock rate
        2. Templates have large overlap in parameter space and time
        1. Extensive verification tests
    3. → Consumer-grade GPU's (for graphics): best compute per dollar
    4. Slim host with 'lots' of GPUs
    5. Serial/single-core on host & one fft per SM (Streaming Multiprocessor)

GPU selection
Best consumer-grade GPUs
(consumer grade: single precision, no ECC)
Limited to power from a single PCI connector

Stage 4
Dedicated cluster with GPU's

Porting to KNL's → back to more serial

Recap Xeon considerations

Xeon host with limited L3 cache

→ Only 1 fft will fit

→ Parallel execution per socket

KNL architecture (many core)

No L3 cache

Large High-Bandwidth Memory (HBM)

68 cores and 272 hardware threads

KNL in HTC

Deploying one a.out per core is perfectly fine

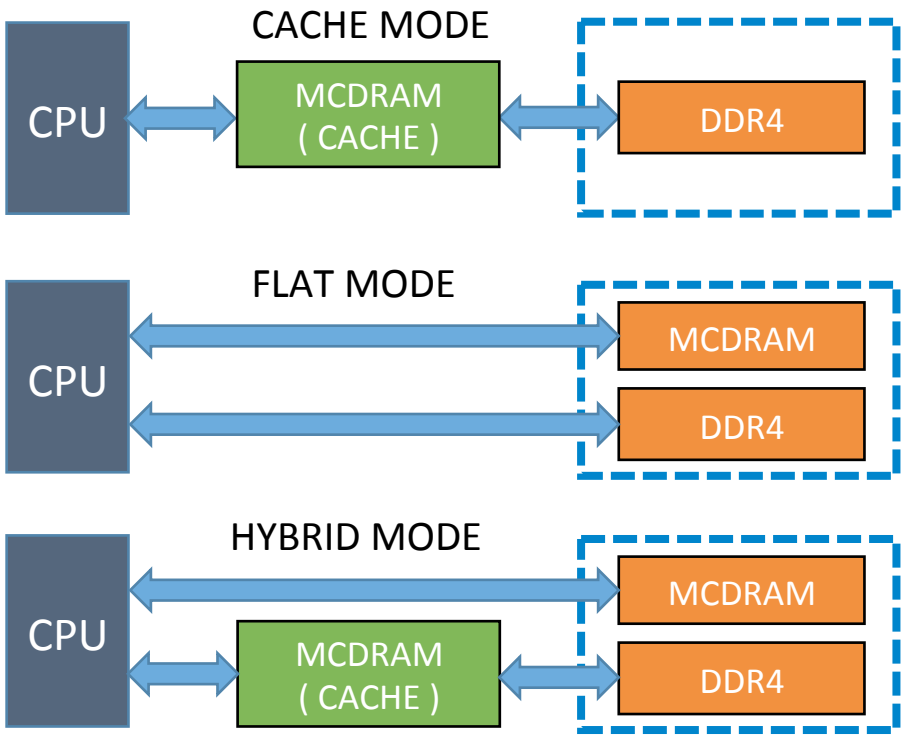
Bandwidth comparison

Xeon L3 (Sandy-Bridge)
roughly the same
KNL HBM

TACC

KNL Memory Architecture

- Two main memory types
  - DDR4
  - MCDRAM
- Three memory modes
  - Cache
  - Flat
  - Hybrid
- Hybrid mode
  - Three choices
  - 25% / 50% / 75%
  - 4GB / 8 GB / 12GB



Timeline (3)

Xeon Cache vs KNL HBM

HBM with16 GB, L3 with couple of MB

→ 16 GB >>> 68 x (couple of MB)

KNL's HBM works very well as a cache

No code modification necessary

How many a.out's?

Selection not limited by size of HBM

68 cores → 68 a.out's

272 hardware threads → 272 a.out's

Bandwidth comparison

Xeon L3 (Sandy-Bridge)

roughly the same

KNL HBM

High-Bandwidth Memory

Large: 16 GB

Works as a cache

'Obviously' high bandwidth

68 or 272 a.out's (or MPI tasks)?

More than one process per core causes problems

Speed-up not very good in many cases

TACC

Configuration matrix

Component testing

How many HT per fft are best

FFTW vs. MKL-fft

Scaling of the 'host' code

Matrix for fft and full code

a.out's	Threads per fft	Threads in C code
16	4, 8, 16	4, 8, 16
34	2, 4, 8	2, 4, 8
68	1, 2, 4	1, 2, 4

Minimum number of threads

At least one thread per core
68 a.out's → 1 thread
34 a.out's → 2 threads
16 a.out's → 4 threads

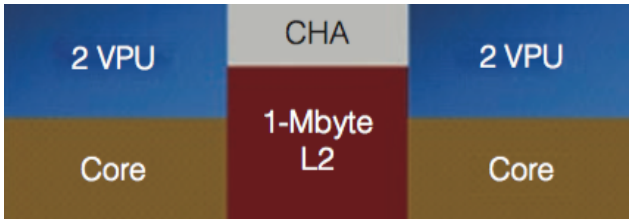
68 cores and 16 a.out's

4 cores left idle
Assigning HT's from a single core to multiple a.out's usually results in bad performance

TACC

Best candidates: 34 or 68?

Tile architecture: 2 cores share the L2 cache



Best candidate: 34 a.out's
Maximum number of threads: 8
One a.out per tile
a.out spans across a tile
L2 cache is not shared

Matrix for fft and full code

a.out's	Threads per fft	Threads in C code
16	4, 8, 16	4, 8, 16
34	2, 4, 8	2, 4, 8
68	1, 2, 4	1, 2, 4

Best candidate: 68 a.out's
Maximum number of threads: 4
One a.out per core
L2 cache is shared

TACC

Summary

- ECSS project has been very successful
- XSEDE provided essential guidance to ‘Compute LIGO’
- pycbc is now highly optimized
- pycbc works well on different systems
  - Slow CPUs (e.g. Einstein at home)
  - High-end CPUs (XSEDE resources)
  - GPU’s (particularly consumer grade)
  - KNL’s (ECSS project will finish soon)
- In general:
  - LIGO made very impressive progress
  - All relevant pipelines are well optimized
  - Unfeasible initial estimates much(!) reduced
  - LIGO on path to transition

from ‘Initial detection’ to ‘Observatory’