Real-time de-dispersion in astrophysics

Wes Armour, Mike Giles

mike.giles@maths.ox.ac.uk

Oxford e-Research Centre

Overview

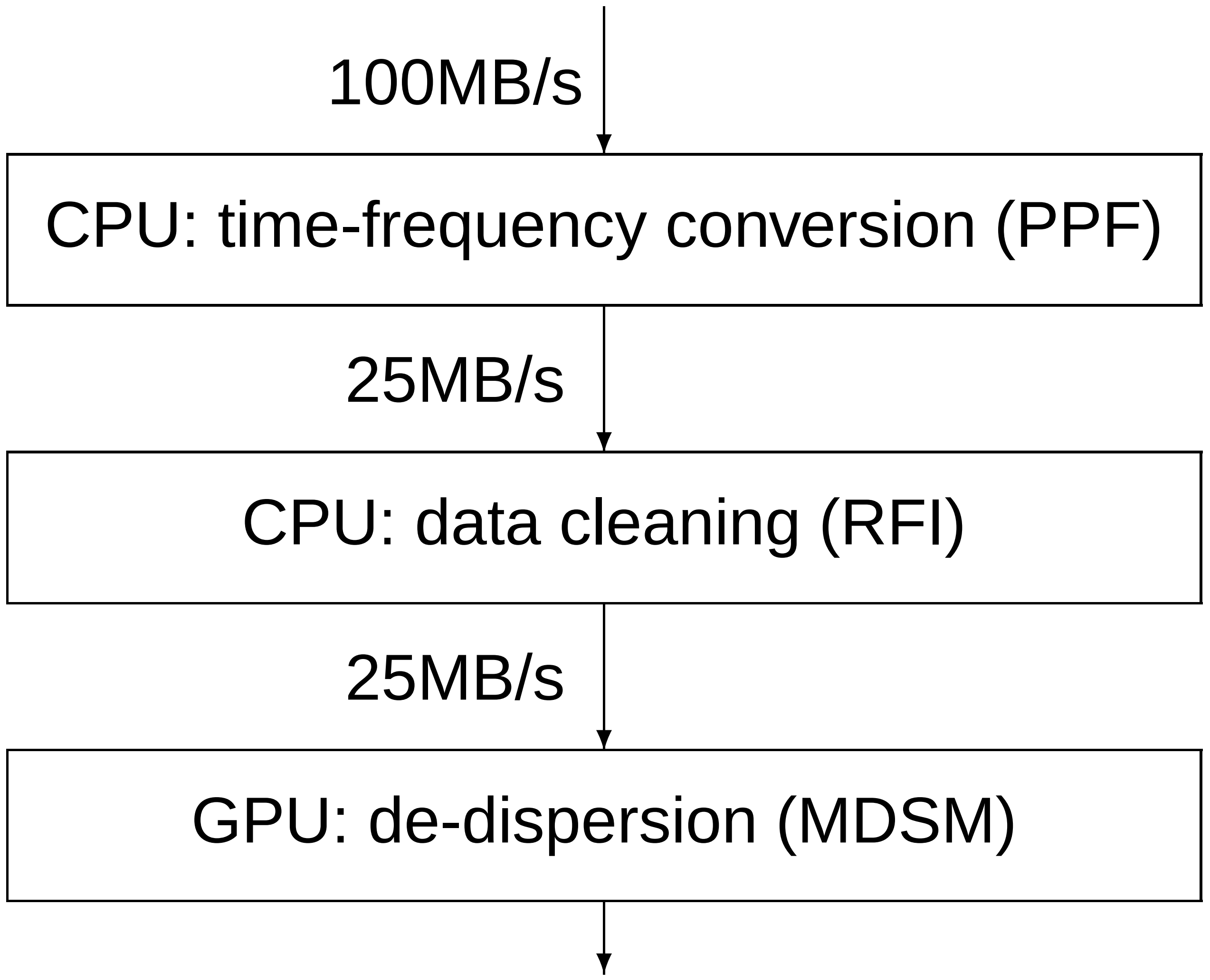
- radio-astronomy
- physics and maths
- back-of-the-envelope assessment
- software design
- performance

Radio-astronomy

- SKA – the big new international astronomy project of the next 20 years
- will use a huge number of radio-telescopes spread across southern Africa and/or Australia/NZ
- like military phased-array radar, adjusts the relative phase of signals from different receptors to “look” in different directions
- 3 major parts to SKA: receptors, data processing, power generation
- lots of pathfinder projects, including LOFAR, to test different technologies, including GPUs

LOFAR dataflow

400MB/s input split into 4 streams of 100MB/s:


100MB/s
CPU: time-frequency conversion (PPF)
25MB/s
CPU: data cleaning (RFI)
25MB/s
GPU: de-dispersion (MDSM)

LOFAR dataflow

- PPF and RFI are data intensive, not much compute required – since no need for high performance, simplest to do these on CPU
- MDSM de-dispersion is compute-intensive (for reasons to be explained) so this is the target for GPU implementation

Physics

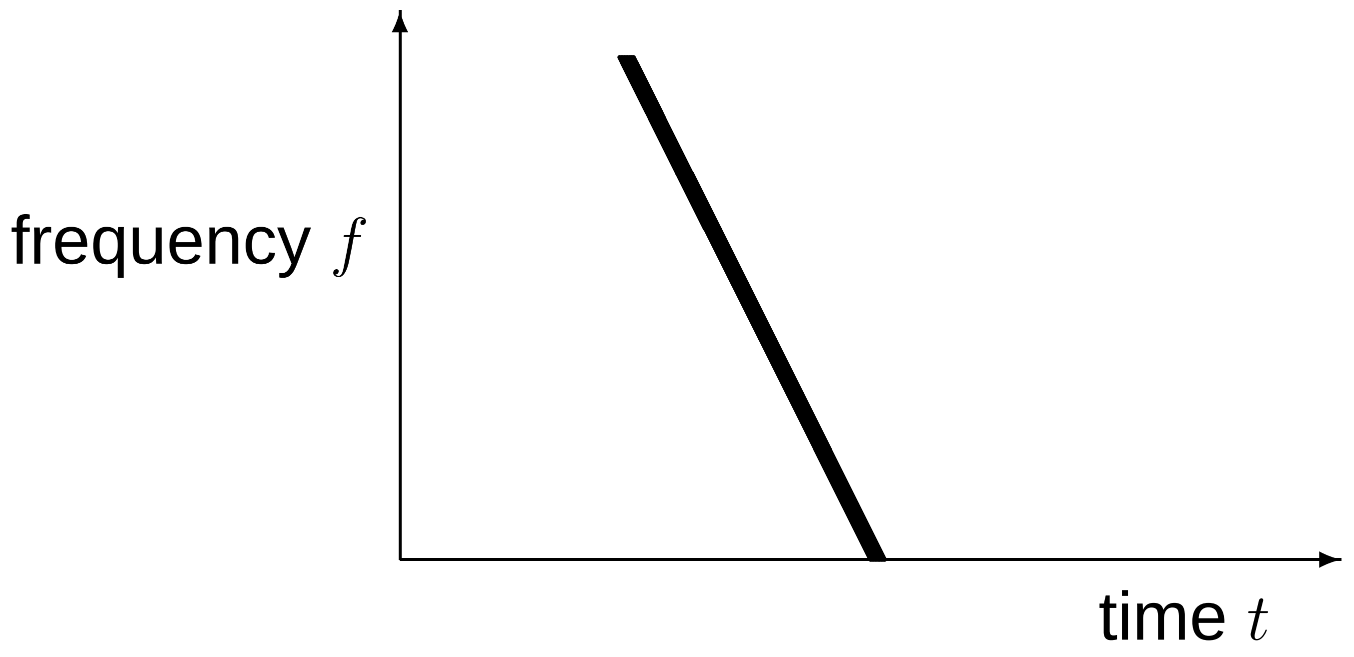
- Pulsars produce a narrow beam of electromagnetic radiation which rotates like a lighthouse beam, so a pulse is seen as it sweeps over a radiotelescope
- The signal is spread over a wide frequency range.
  If space was an empty vacuum, all the signals would travel at the same speed, but due to free electrons different frequencies travel at slightly different speeds (dispersion)
- The difference in travel time is proportional to distance, so the distance can be deduced from the relative time lag between different frequencies

Physics

The time delay depends on frequency f, and is proportional to the dispersion measure m which corresponds (roughly) to distance:

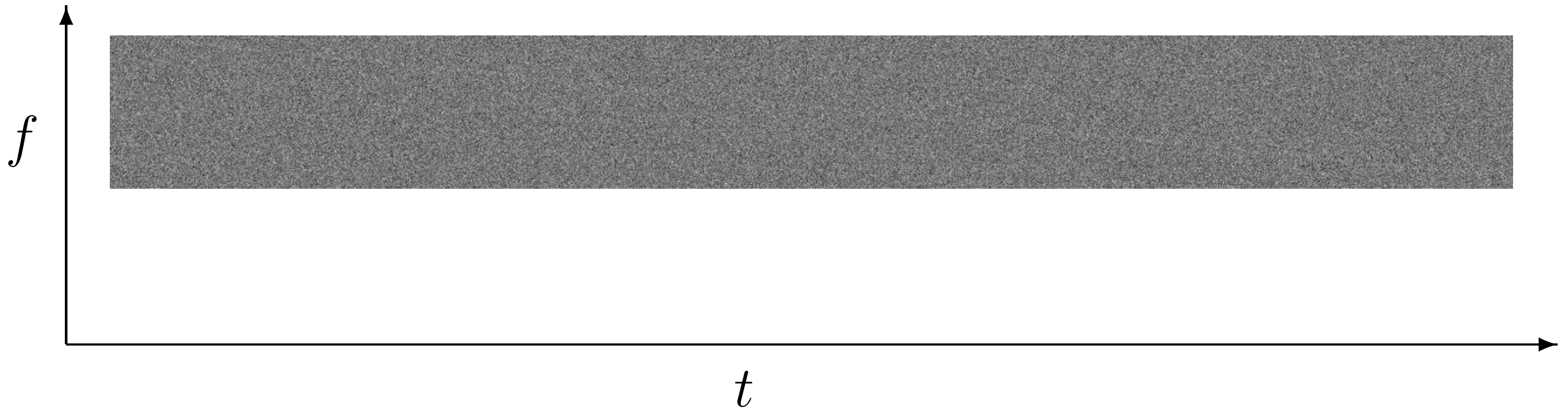
$$\tau = m\ d(f)$$

Since $d(f)$ is known, can work out m from signal data:



Physics

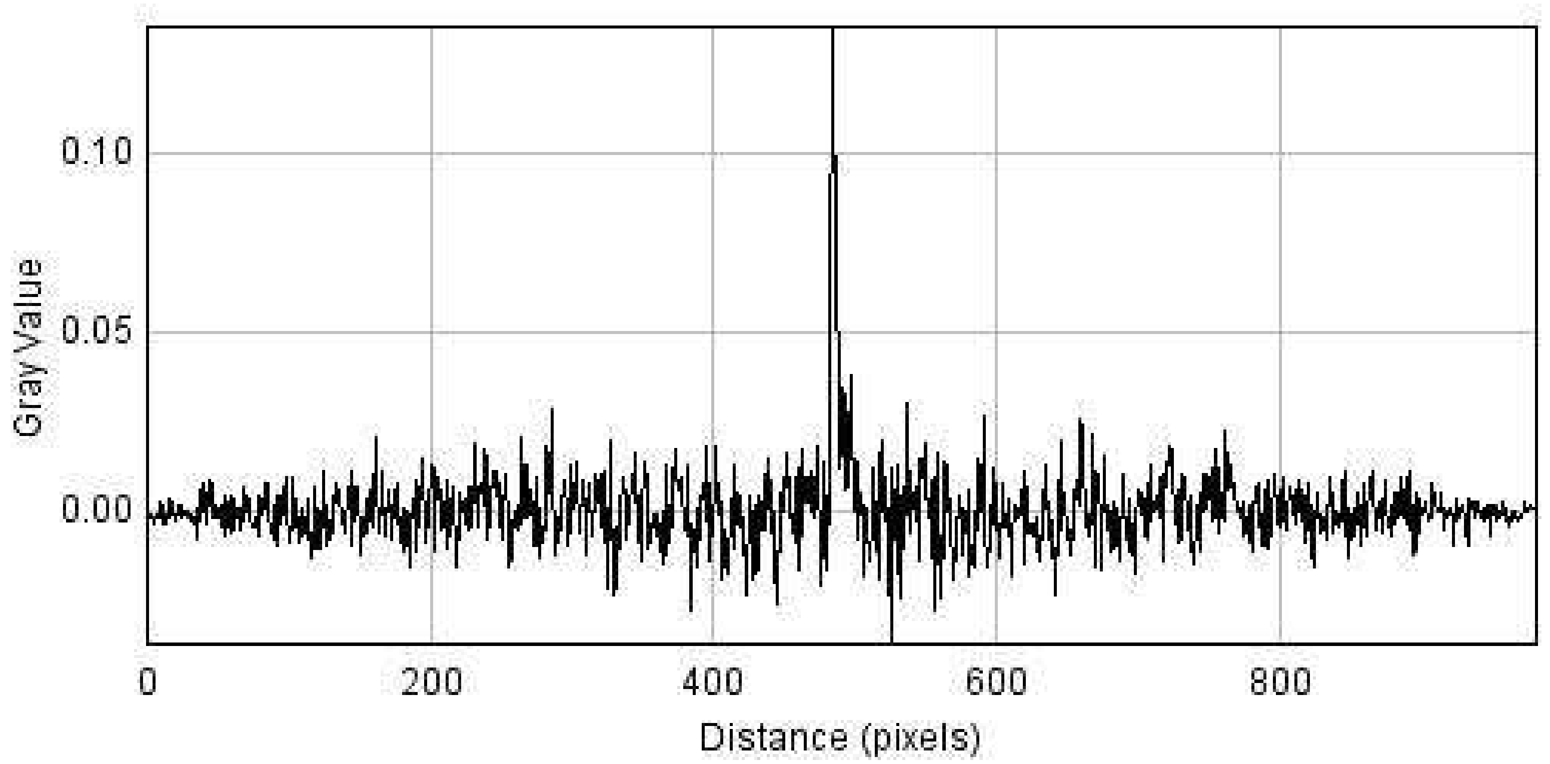
Problem: the signal is often very weak, barely distinguishable from the background noise



Physics

Solution: if we know the right value for m, then we can time-shift the data to correct for the dispersion (i.e. we can de-disperse the signal) then sum over the frequencies

This reinforces the signal relative to the background noise



Physics

New problem: we don’t know the right value for m

Solution: try lots of different values for m; the right one is the one that gives a clear signal!

This needs lots of computation – that’s why we are interested in using GPUs

Maths

Let

- f be integer frequency index, $0 \leq f < F$
- t be integer time index
- m be integer dispersion measure index, $0 \leq m < M$

Given input data $u(f, t)$, the objective is to compute the output

$$w(m, t) = \sum_f u\left(f,\, t - s(m, f)\right),$$

for an integer shift function $s(m, f)$ which is approximately linear in m, and varies little from m to $m{+}1$:

$$\max_{m,f} \left| s(m{+}1, f) - s(m, f) \right| \leq 5 \quad \text{(for our testcase)}$$

Back-of-envelope assessment

For each time slice t:

- F inputs
- M outputs
- $M\,F$ floating point operations

Typically $F,\ M\ \simeq 1000-2000$, so enough computation to hide communication cost of PCIe bus

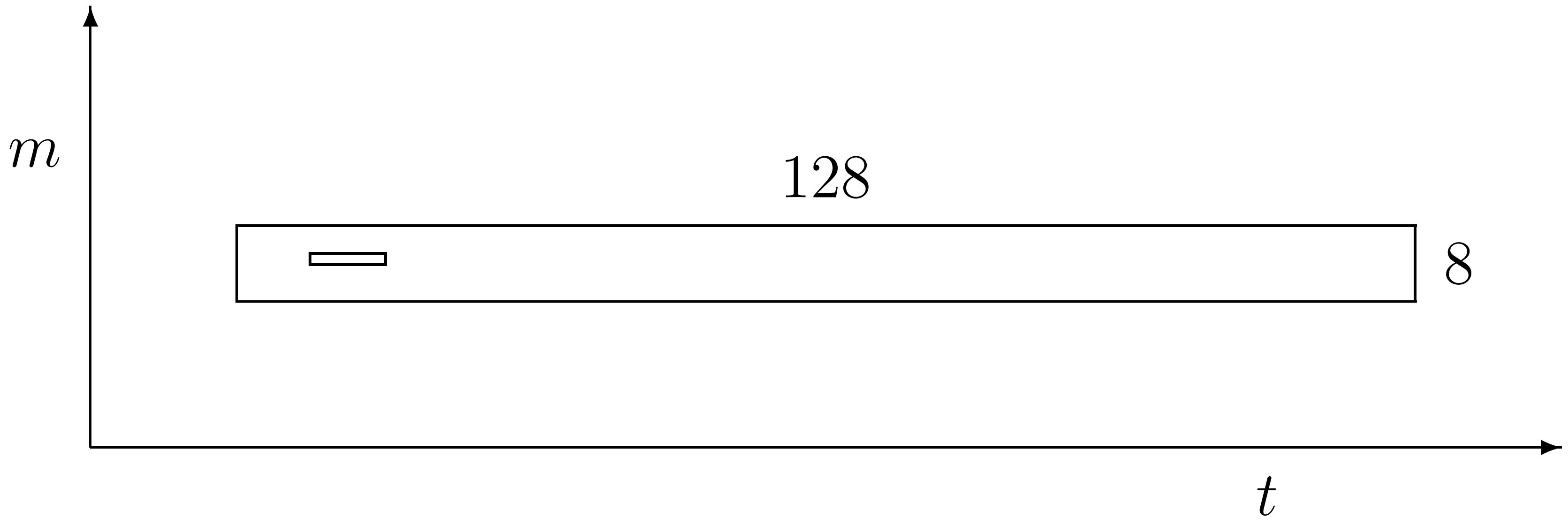
GPU memory bandwidth also not a problem, provided each input data item isn't transferred too many times

Back-of-envelope assessment

Other thoughts:

- no conditional code to worry about
- no relevant libraries
- looks a bit like matrix multiplication
  - key to performance will be blocking for data re-use
  - each output handled by one thread to avoid data dependencies
- should try to minimise the number of integer operations required

Initial software design



Region in dispersion space worked on by a single CUDA block with 128 threads – each will handle 8 points

Should get 8 blocks running on each SM if each thread needs at most 32 registers

Initial software design

Implementation:

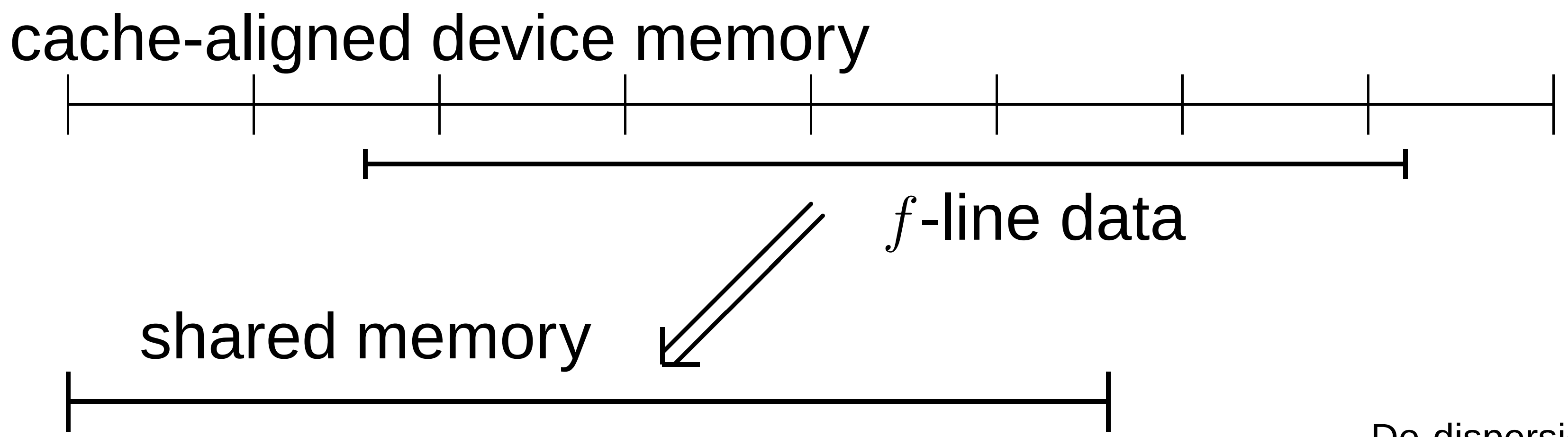
1. load f-line into shared memory
2. sync threads
3. each thread adds shifted values to 8 accumulators
4. sync threads
5. go back to step 1 and repeat for next f-line

- use of shared memory gives data reuse – most data items are used 8 times, once for each m-line
- this implementation alternates communication and computation – relies on multiple blocks for overlapping

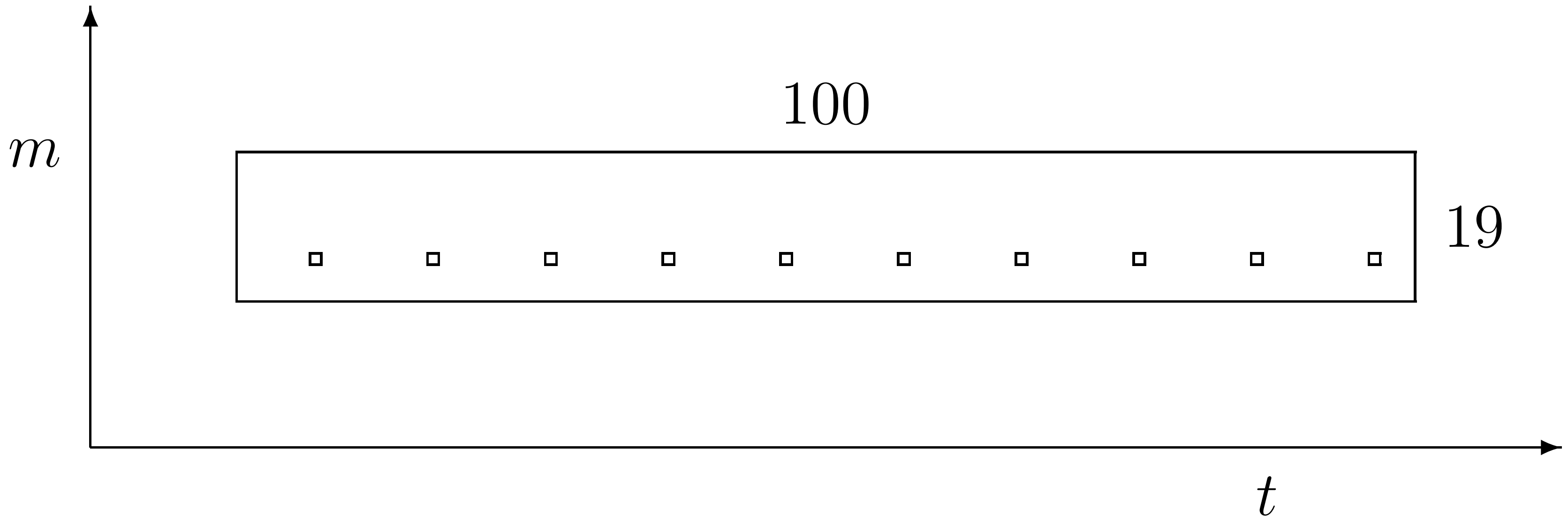
Initial software design

Implementation 1:

- each m-line needs 128 values
- at most a 5-shift between one m-line and the next
- at most a 35-shift for set of 8 m-lines, so at most 128+35 values required
- requires 6 cache lines, each holding 32 floats
- data reuse factor = $(8 \times 128) \;/\; (6 \times 32) \approx 5.3$



Optimised software design



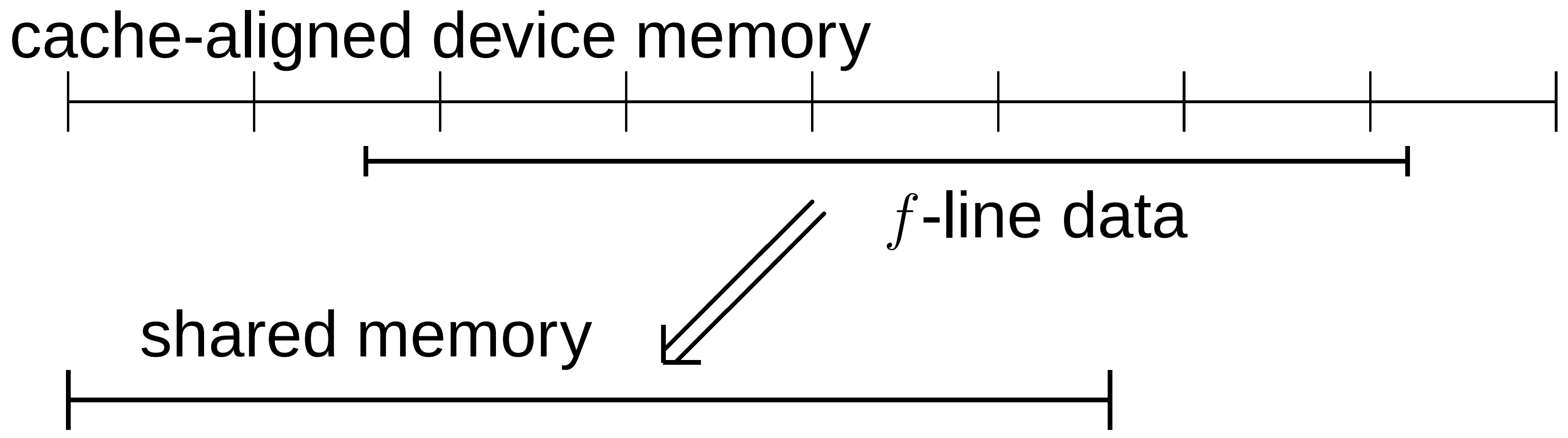
Rectangular region in dispersion space worked on by a block of 10×19 threads, each handling 10 points using 24 registers

7 thread blocks run simultaneously on each SM – 1330 threads on each SM, almost 20,000 on whole GPU

Optimised software design

Implementation:

- each m-line needs 100 values
- at most a 5-shift between one m-line and the next
- at most a 90-shift for set of 19 m-lines, so at most 100+90 values required (1 per thread)



Performance

Static testcase:

- 2 mins of test data: 3GB
- 0.6s transfer time to GPU: 5GB/s
- 15s processing time
  - approx 500 Gops, roughly evenly split between SP floating point, integer and shared memory reads
  - approx 40 GB/s bandwidth from graphics memory
- overall: achieving 40–50% of peak compute capability and communication

I'm very satisfied with performance, not much scope for improvement – 1 GPU could handle all 4 data streams in real-time

Two lessons learned

Auto-tuning is important:

- needs a lot of fiddling around to determine optimum parameters – not obvious even to an expert

Optimising data movement is key to performance:

- bandwidth struggling to match huge compute capability
- need to minimise the number of times data is moved
- applies also to CPU code – need good cache behaviour

Sandy Bridge AVX

The same algorithm has also been implemented on Intel’s new Sandy Bridge CPU with AVX vector units:

- 4 cores, 3.4 GHz
- AVX vector unit is 128 bits wide – 8 floats
- peak GFlop rate: $4 \times 3.4 \times 8 \approx 100$
  (not bad compared to GPU)
- auto-vectorising compiler does a very poor job
- to get good performance, need to hand-code using vector intrinsics (ugly!)
- final performance about 50% of peak
- currently working on improving the CPU performance for other elements of the dataflow

Final comments

I remain keen on GPUs, but not the only game in town:

- even longer 256-bit AVX vector units in new Intel MIC chip (successor to Larrabee, going into new 10 petaflop supercomputer in Texas)
- likely that these will also go into mainstream CPUs in future?
- key in both cases is idea of vector computing, and minimising data movement

Acknowledgements: funded by the Oxford Martin School
`http://www.futurecomputing.ox.ac.uk/`