Case study 1: de-dispersion in astrophysics

Wes Armour, Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Overview

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

Physics

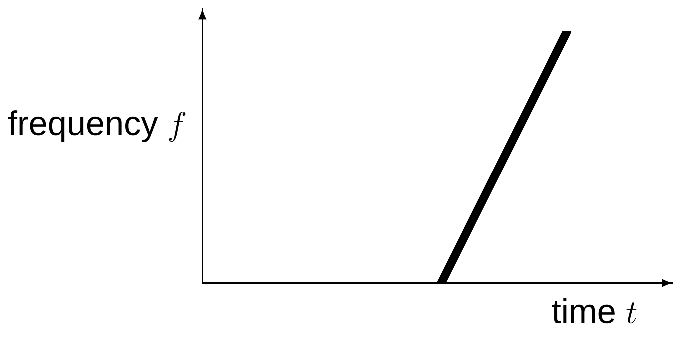
- Pulsars produce short bursts of electromagnetic radiation
- These are spread over a wide frequency range, and observed by radio-telescopes
- If space was an empty vacuum, all the signals would travel at the same speed; however, due to free electrons, different frequencies travel at different speeds (dispersion)
- The difference in travel time is proportional to distance, so the distance can be deduced from the relative time lag between different frequencies

Maths

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

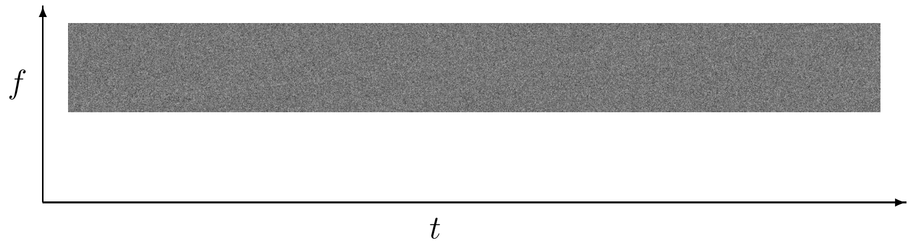
$$\tau = m\, d(f)$$

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



Physics

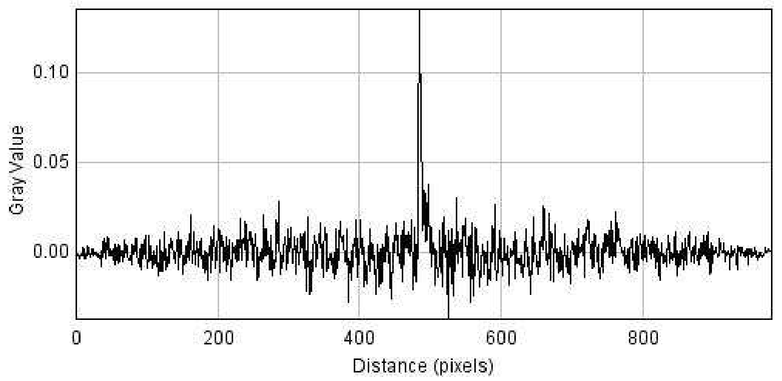
Problem: the signal is often very weak, barely distinguishable from the background noise



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 $m{+}1$:

$$\max_{m,f} \left| s(m{+}1,f) - s(m,f) \right| \leq 9$$

BoE Assessment

For each time slice t:

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

Typically $F,\,M\ \simeq 100-200$, so enough computation to hide communication cost of PCIe bus.

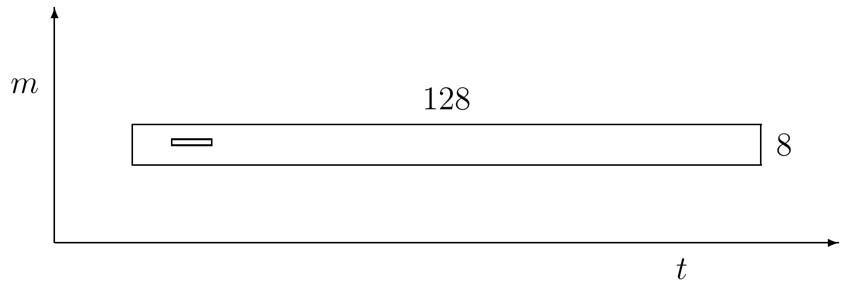
GPU memory bandwidth shouldn't be a problem either, provided each data item isn't transferred too many times

BoE 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

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

Software Design

Implementation 1:

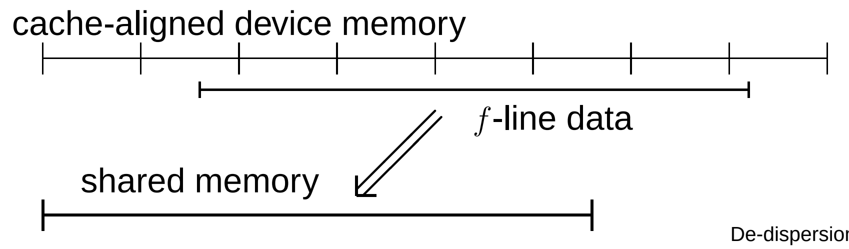
1. load f-line into shared memory
2. sync threads
3. each thread adds shifted values to 8 accumulators
4. sync threads
5. go back to step 1 and repeat for next f-line

- 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

Software Design

Implementation 1:

- each m-line needs 128 values
- at most a 9-shift between one m-line and the next
- at most a 63-shift for set of 8 m-lines, so at most 128+63 values required
- requires 7 cache lines, each holding 32 floats
- data reuse factor = $(8 \times 128) \;/\; (7 \times 32) \approx 4.5$



Software Design

Implementation 2:

1. load first f-line into shared memory
2. sync threads
3. load next f-line into registers
4. each thread adds shifted shared-memory values to 8 accumulators
5. sync threads
6. store new f-line register values into shared memory
7. go back to step 2 and repeat

Software Design

Implementation 2:

- steps 3 and 4 are overlapped: read into registers is started and then computation is performed in parallel
- if the read went straight into a different bit of shared memory, the compiler may worry about data dependencies
- this is similar to double-buffering in computer graphics – show one scene while drawing the next scene in a different buffer

BoE Assessment

- each thread handles 8 outputs – all use the same shift which minimises the integer operations per output
- factor 4.5 data reuse gives 4.5 flops and 4.5 integer ops per data transfer – still barely sufficient to avoid being bandwidth limited?
- might be better for block shape to be 64×16 to get more reuse – write parameterised code to find optimal shape
- PCIe bandwidth is 5GB/s, and GPU memory bandwidth is at least 100 GB/s – provided data is read into GPU no more than 20 times, then the PCIe bus will be the limit, not the GPU bus

Case study 2: stochastic simulation of chemical reactions

Guido Klingbeil, Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Overview

- chemistry and maths
- initial assessment
- one-off CUDA implementations
  - few species / reactions
  - more species / reactions
- performance assessment
- code generator

Chemistry

A chemical reaction involving 3 molecular species in a well-stirred solution

$$x_1 + 2\,x_2 \to x_3$$

is usually modelled by concentrations $c_1(t), c_2(t), c_3(t)$ with

$$\begin{aligned}\dot{c}_1 &= -R\\ \dot{c}_2 &= -2\,R\\ \dot{c}_3 &= +R\end{aligned}$$

The reaction rate is given by the “Law of Mass Action”:

$$R = \alpha\; c_1\; c_2$$

Chemistry

However, when the concentrations are extremely small one has to consider individual molecules, and the probability that they will come together and react.

This leads to a stochastic model with $x_1(t), x_2(t), x_3(t)$ molecules which react in the time interval $\mathrm{d}t$ with probability

$$\beta\; x_1\; x_2\; \mathrm{d}t$$

and if they react then

$$\begin{aligned}\Delta x_1 &= -1\\ \Delta x_2 &= -2\\ \Delta x_3 &= +1\end{aligned}$$

Chemistry

Generalising this, we have N species, $X = (x_1, \ldots, x_N)$, and M reactions.

For each reaction we have:

- propensity function $a_m(X)$
  (probability per unit time of a reaction taking place)
- reaction vector ν_{mn}
  (if it happens then $\Delta x_n = \nu_{mn}$)

Mathematics

- Combined propensity function

$$a_0(X) = \sum_m a_m(X)$$

- Time of next reaction is $(1/a_0) \log U$, where U is a $(0,1)$ uniformly distributed random number
- Probability that next reaction is reaction m is a_m/a_0

Mathematics

This leads to the Stochastic Simulation Algorithm (SSA)

1. compute propensity $a_m(X)$ for all reactions
2. generate $(0,1)$ uniform random numbers U_1, U_2
3. use U_1 to get next reaction time
4. use U_2 to get next reaction index i:
$$\sum_{m=1}^{i-1} \frac{a_m}{a_o} < U_2 \leq \sum_{m=1}^{i} \frac{a_m}{a_o}$$
5. update x_n
6. go back to step 1 and repeat

(One variation improves efficiency by only updating the propensities which have changed.)

Monte Carlo Simulation

We don't just do one simulation – we do lots, $10^4 - 10^6$, using independent random numbers for each one.

The natural independence makes this very well suited to parallel execution on GPUs.

Typically, what we are interested in is the mean and standard deviation of X at various monitoring times $T_1, T_2, \ldots$, with $100 - 1000$ reactions in each interval.

Initial Assessment

- natural parallel independence
- minimal bandwidth requirments, so compute-limited
- library can be used for parallel random number generation
- conditional branching could be a problem
- number of registers required could be a problem, depending on number of species and reactions

First Implementation

For applications with few species / reactions:

- each thread handles one simulation
- all propensities updated
- avoid conditional code as much as possible
- `case` statement used to handle updating of species, with reactions ordered roughly in decreasing probabilty
- `while` statement continues simulating to next output time – each warp continues until all of its threads have reached it
- random numbers can be generated as needed, or pre-computed to reduce registers required
- probably best with 128 threads per block, and 4-8 blocks on each SM

Second Implementation

For applications with more species / reactions:

- $32 \times K$ threads in each thread block, with $1 < K \leq 8$
- K threads cooperate to perform one simulation, with X and a stored in shared memory
- each thread/warp computes subset of propensities
- first warp determines which reaction takes place, and updates X

Generic Application Software?

Above implementations are OK for two specific applications, but what if you want to develop general software which can be used easily for new applications?

The usual solution is a generic application code:

- N, M, ν_{mn} and propensity functions all input parameters
- requires dynamic memory allocation
- code would loop over m and n

On a CPU, there would be some overhead in making it generic, but maybe a price worth paying.

On a GPU, the cost is much larger (factor $10\times$?)
– is there an alternative?

Code Generation

Yes – application-specific code generation!

- a high-level code takes application specification and produces a CUDA code specific to that application
- it sounds tough – but it's not!
- I use MATLAB because that's what I'm comfortable with, but you could use other languages (e.g. Python)
- all that is needed is ability to
  - process input specification
  - produce output code (by writing strings to a file)
- it doesn't require expert computer science knowledge – anyone can do it

Code Generation

How do you write a code generator?

- first, produce one or two hand-coded examples – experiment to find the best CUDA implementation
- second, look at your code – how would you change it for a new application? what mental rules would you apply?
- third, write a high-level code to implement your mental rules, taking the specification and producing lines of output code
- check that this reproduces your hand-coded examples
- add refinements to measure performance, try different optimisations, etc. – you might not do this for a one-off application, but with a generic tool it will repay the effort

Code Generation

I know of 3 CUDA code generators in Oxford:

- Stochastic simulation:
  - MATLAB prototype by me
  - extended by Guido Klingbeil (Maths)
  - high-level specification also in MATLAB
- OP2 – case study 3:
  - high-level specification in C++ / FORTRAN
  - MATLAB prototype by me – parses function calls
- DNA sequencing:
  - developed by Luke Cartey (Computer Science)
  - specification in his own domain specific language
  - code generaton using JastAdd (Java-based)