A GPU-based survey for millisecond radio transients using ARTEMIS


Oxford e-Research Centre

18th November 2011
GPUs

Why use GPUs?
Overview of current GPUs - Fermi

• Fermi released 2010

• 32/48 cores (stream processors or SP) per Streaming Multiprocessor (SM) (sm_20/sm_21)

• Configurable 16/48K or 48/16K L1 cache / shared memory (per SM)

• New L2 cache – 768K

• GigaThread – Concurrent kernel execution (16)

• Simple to use extensions - CUDA for C, C++ & FORTRAN
Fermi Streaming Multiprocessor (SM) design
Fermi’s 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).
Parallel granularity and data sharing.

- Each cuda core (SP) executes a sequential thread, in SIMT (Single Instruction, Multiple Thread) fashion - all cores in the same group execute the same instruction at the same time (like SIMD).

- Threads are executed in groups of 32 – a warp.

- To hide high memory latency, warps are executed in a time-multiplexed fashion - When one warp stalls on a memory operation, the multiprocessor selects another ready warp and switches to that one.
Parallel granularity and data sharing.

- Kernel launches a grid of (3,2) thread blocks...

Kernel<<< (3,2),(4,3) >>>(params)

- Each thread block consists of (4,3) unique threads.
Parallel granularity and data sharing.

• Single thread has its own (cannot be shared by other threads) per-thread very fast local memory (registers).

• A block of threads has its own per-block shared memory allowing for communication and data sharing between threads in a thread block.

• A grid of thread blocks can communicate via global memory.
Latest Fermi based cards

- GeForce GTX 590 – 2x GF110 GPUs
  - 6.8 GFLOPS / watt
  - Price point ~ £600

- M2090 – 1x GF110 GPU, 16 SMs, 512 cores.
  - 6.8 GFLOPS / watt
  - Price point ~ £3K
Influence and Take-up

TOP 500 – 3 out of top 5 utilise Fermi/Tesla

• Tianhe-1A 2.5 petaflops
• Based on 14336 Xeon and 7168 M2050
• To achieve same performance using only CPUs 50000 CPUs, 2x floor space and 3x power (Estimates made by NVIDIA)
• Uses Lustre :-s
Radio Astronomy

Radio Telescopes
LOFAR – Low Frequency Array for Radio Astronomy

- The largest radio telescope ever built - 10000+ omni-directional antennas and 77 larger stations.
- Distributed across Holland, Germany, UK, France and Sweden.
- Direction of observation is chosen by phase delays between antennas – “Aperture Synthesis”
- Can observe in several directions simultaneously - multi-user operation.
- “Path finder” for the SKA – Square Kilometre Array.
Dedispersion
Radio Astronomy and Radio Transients

**Quasars** – Energetic region of a distant galactic core, surrounding a supermassive black hole

![Quasar image](image)

NASA and J. Bahcall (IAS)

**Pulsars** – Magnetized, rotating neutron stars. Emit synchrotron radiation from the poles, e.g. Crab Nebula


Hester et al.
Dispersion

Chromatic dispersion is something we are all familiar with. A good example of this is when white light passes through a prism.

Group velocity dispersion occurs when pulse of light is spread in time due to its different frequency components travelling at different velocities. An example of this is when a pulse of light travels along an optical fibre.
Dispersion of Radio waves by the ISM

The interstellar medium (ISM) is the matter that exists between stars in a galaxy.

In warm regions of the ISM (~8000K) electrons are free and so can interact with and effect radio waves that pass through it.

Haffner et al. 2003
The Dispersion Measure - DM

The time delay, $\Delta \tau$, between the detection of frequency $f_{\text{high}}$ and $f_{\text{low}}$ is given by:

$$\Delta \tau = C_{DM} \times DM \times \left( \frac{1}{f_{\text{low}}^2} - \frac{1}{f_{\text{high}}^2} \right)$$

Where $C_{DM}$ is the dispersion constant. DM is the dispersion measure:

$$DM = \int_{0}^{d} n_e dl$$

This is the free electron column density between the radio source and observer.

We can measure $\Delta \tau$ and $f$ and so can study DM
• MOTIVATE is a pathfinder project and aims to investigate the latest many-core technologies with the aim of delivering energy and cost efficiency in the area of radio astronomy HPC.

• MOTIVATE stands for Many-cOre Technology Investigating Value, Application, deploymentT and Efficiency.

• The MOTIVATE project is funded by the Oxford-Martin School through the Institute for the Future of Computing

http://www.oerc.ox.ac.uk/research/many-core
Aims of the Astro–Acceleration project…

- Process a 3.2 Gb/s radio telescope data stream on a single GPU, de-dispersing and detecting transient events.

- Allowing for a vast reduction in the cost (capital/maintenance) of compute.

- Identify appropriate areas of parallelism on both CPU and GPU in the de-dispersion pipeline.

We have chosen to use the MDSM software as a wrapper to out GPU kernel

Experimental data

Most of the measured signals live in the noise of the apparatus.
Experimental data

Most of the measured signals live in the noise of the apparatus.

Hence frequency channels have to be “folded”
Brute force algorithm

Every DM is calculated to see if a signal is present.

• In a blind search for a signal many different dispersion measures are calculated.

• This results in many data points in the (f,t) domain being used multiple times for different dispersion searches.

• This allows for data reuse in a GPU algorithm.
MDSM 1st Stage – Memory access optimisation

- Began by profiling the code using the Cuda Profiler.
- Identified bottlenecks in memory access patterns.
- By moving the variable that holds the running total from shared memory to a register
  kernel execution was 2.6x faster.

```c
__device__ float localvalue[4096];
__global__ void opt_dedisperse_loop(…) {

  float localvalue;

  7: Performed Brute-Force Dedispersion 1: 1743.604004
  7: Performed Brute-Force Dedispersion 0: 1747.496582

  2.6x

  7: Performed Brute-Force Dedispersion 0: 675.611206
  7: Performed Brute-Force Dedispersion 1: 677.087769

  Produces a 5x speed increase in the less accurate “sub-band” method.
```
MDSM 2\textsuperscript{nd} Stage – Occupancy Optimisation

- Optimize GPU block size (number of threads) for each kernel to get the maximum occupancy of threads on the GPU.

- Investigated the optimal block size for GPU hardware versions sm\_13 (Tesla) and sm\_20 (Fermi).

- Found a block size of 192 on sm\_13 and 256 on sm\_20 produced best results.

- This produced a \textbf{1.25x} speed increase.

\textbf{Produce a 1.3x speed increase in the less accurate “sub-band” method.}
Timings for 1.8 seconds of telescope data, 496 channels, 800 dm’s, max dm 79.9

Original Kernel 1156 ms

Optimised Kernel 375 ms

371 ms reduces to 67 ms in the less accurate “sub-band” method.
A new GPU brute force algorithm

Three key features…

• Each thread processes a variable number of dispersion measures in local registers.

• Exploit the L1 Cache / Shared Memory present on the Fermi architecture.

• Optimise the region of dispersion space being processed (thread blocksize).

Web page :  http://www.oerc.ox.ac.uk/research/wes
New Algorithm works in the DM - t space rather than frequency – time space.

- Each thread processes a varying number of time samples for a constant dispersion measure.
- This ensures frequency - time data is loaded into fast L1 cache.
- Using registers ensures very quick memory access.
Exploiting the L1 cache…

Each dispersion measure for a given frequency channel needs a shifted time value.

Constant DM’s with varying time.

Incrementing all of the registers at every frequency step ensures a high data reuse of the stored frequency time data in the L1 cache.
Optimising the parameterisation.

The GPU block size of the new algorithm can take on any size that is integer multiples of the size of a “data chunk”…
Each thread loads a single element of (f,t) data into a shared memory array.

```c
// Calculate values for a normalised shift
int shift_one = (mstartdm + ((blockIdx.y*DIVINDM + threadIdx.y)*mdmstep));
int shift_two = (mstartdm + (blockIdx.y*DIVINDM*mdmstep));

// Calculate the threads index
int idx = (threadIdx.x + (threadIdx.y * DIVINT));

// Load single element of (f,t) into shared memory
f_line[idx] = buff[((c*(i_nsamp + i_maxshift)) + (blockIdx.x*NUMREG*DIVINT + threadIdx.x)) + __float2int_rz(dm_shifts[c]*shift_one)];

// Sync the threads to ensure complete (f,t) line is loaded into shared memory
__syncthreads();

// Calculate the local shift (within the loaded f,t line) then increment the set of accumulators
shift = __float2int_rz(dm_shifts[c]*shift_one) - __float2int_rz(dm_shifts[c]*shift_two);
for(i = 0; i < NUMREG; i++) local_kernel_t[i] += f_line[(shift + (i*DIVINT))];
```
A CPU brute force algorithm

Four key features…

• Aim to achieve full cache line utilization.

• Exploit the large (~375 GB/s) LLC bandwidth present on the new Intel Sand Bridge CPUs.

• Use the Intel Intrinsics to exploit the 16 AVX/SSE (YMM/XMM) SIMD registers (don’t rely on the Intel auto-vectorizer!)

• Use OpenMP to share work across the CPU cores.
• Process vectors of time, holding DM constant.

// Declare a local array AVX vectors
__m256 xmm[16]

// Loop over half of the 16 avx registers
for(i = 0; i < 8; i++) {

    // Unaligned load of 8 floats into AVX register i
    xmm[i] = _mm256_loadu_ps(input_buffer+shift+(i*SIMDWIDTH));

    // Add the loaded (f,t) values = xmm[i], to the accumulator register xmm[i+8]
    xmm[i + 8] = _mm256_add_ps(xmm[i], xmm[i + 8]);
}
// de-disperse each time sample
#pragma omp parallel for private(dm_count,i,c)
for(t = 0; t < (nsamp - maxshift); t = t+((NUMREG/2)*SIMDWIDTH)) {
    float *local_value;
    local_value = (float *) _mm_malloc(SIMDWIDTH*sizeof(float),ALIGNSIZE);
    for(dm_count = 0; dm_count < tdms; dm_count++) {
        float shift_temp = dm_low/tsamp + dm_count * (dm_step/tsamp);
        __m256 xmm[NUMREG];
        #pragma unroll
        for(i = 0; i < NUMREG; i++)
            xmm[i] = _mm256_setzero_ps();
        // Move through all frequency channels
        for(c = 0; c < nchans; c++) {
            // Calculate time independent shift
            register int shift = ((c * nsamp) + t) + floor(dmshifts[c] * shift_temp);
            #pragma unroll
            for(i = 0; i < (NUMREG/2); i++)
                xmm[i] = _mm256_loadu_ps(input_buffer + shift+(i*SIMDWIDTH));
            xmm[i + (NUMREG/2)] = _mm256_add_ps(xmm[i], xmm[i + (NUMREG/2)]);
        }
        for(i = 0; i < (NUMREG/2); i++)
            _mm256_store_ps(local_value, xmm[i+(NUMREG/2)]);
        #pragma unroll
        for(acc = 0; acc < SIMDWIDTH; acc++) {
            output_buffer[(dm_count)*(nsamp-maxshift) + (t) + acc + i*SIMDWIDTH] = local_value[acc];
        }
    }
    _mm_free(local_value);
}
Time binning

When scattering and dispersion effects are high, a radio signal can be spread over multiple time samples, all having the same frequency.

It makes sense to add the values of adjacent time data to increase the signal to noise. This reduces the amount of time samples to process at higher DMs (also increases the step size between DMs to achieve critical sampling).
Time binning...

1. Launch kernel to dedisperse data up to

2. Use CPU to bin data in time concurrently $+ = $

3. Use cudaThreadSynchronize() on host to ensure kernel execution has finished.

4. Transfer binned data to device.

5. Repeat
Time binning…

Has the effect of reducing the amount threads that are needed to process a region of (DM,t) space.

Utilizes the CPU and GPU at the same time.
Brute force results…

Results
Results...

A signal recovered with the correct DM parameters. (Matches that of the simulated data)
A signal recovered with the correct time parameters. (Matches that of the simulated data)
Results...

Comparison of different technologies

- New GPU Algorithm M2050
- 8x Xeon E7-8860 SSE (80 cores)
- 4x Xeon X7550 SSE (32 cores)
- Previous MDSM kernel M2050
- i7 2600K AVX (4 cores)

Number of channels = total number of DMs with Maximum DM = 200
Results…

![Graph showing the relationship between maximum dispersion measure and fraction of real-time for different processors: 4x Xeon X7550 SSE (red triangles), i7 2600K AVX (blue squares), NVidia M2050 (green circles). The graph indicates varying performance across different dispersion measures.]
Results...

Normalised run times (timed using omp) as a function of cores used
Conclusions and Future Work

- GPU wins hands-down. **At the moment!**
- AVX puts up a good fight.
- Watch out for Intels MIC (Many Integrated Core) chip – 32 in-order cores, 4 threads per core 512 bit SIMD units running a 1024 bit ring bus.

- Shared memory algorithm ensures more predictable data reuse and is about 15% quicker for some maximum DM and number of channels combinations – results to come.
- Ben, Mike and me are working on AVX vectorisation of the Polyphase filter.
Acknowledgments and Collaborators

**GPU de-dispersion**: [http://www.oerc.ox.ac.uk/research/wes](http://www.oerc.ox.ac.uk/research/wes)

**ARTEMIS**: [http://www.oerc.ox.ac.uk/research/artemis](http://www.oerc.ox.ac.uk/research/artemis)

**University of Malta**
Alessio Magro – MDSM

**University of Oxford**
Mike Giles (Maths) – Cuda, GPU algorithms.
Aris Karastergiou (Physics) – ARTEMIS, Astrophysics, Experimental Work.
Kimon Zagkouris (Physics) – Astrophysics, Experimental Work.
Chris Williams (OeRC) – RFI Clipper, Data pipeline.
Ben Mort (OeRC) – Data Pipeline, pelican.
Fred Dulwich (OeRC) – Data Pipeline, pelican.
Stef Salvini (OeRC) – Data Pipeline, pelican.
Steve Roberts (Engineering) – Signal processing/detection algorithms.