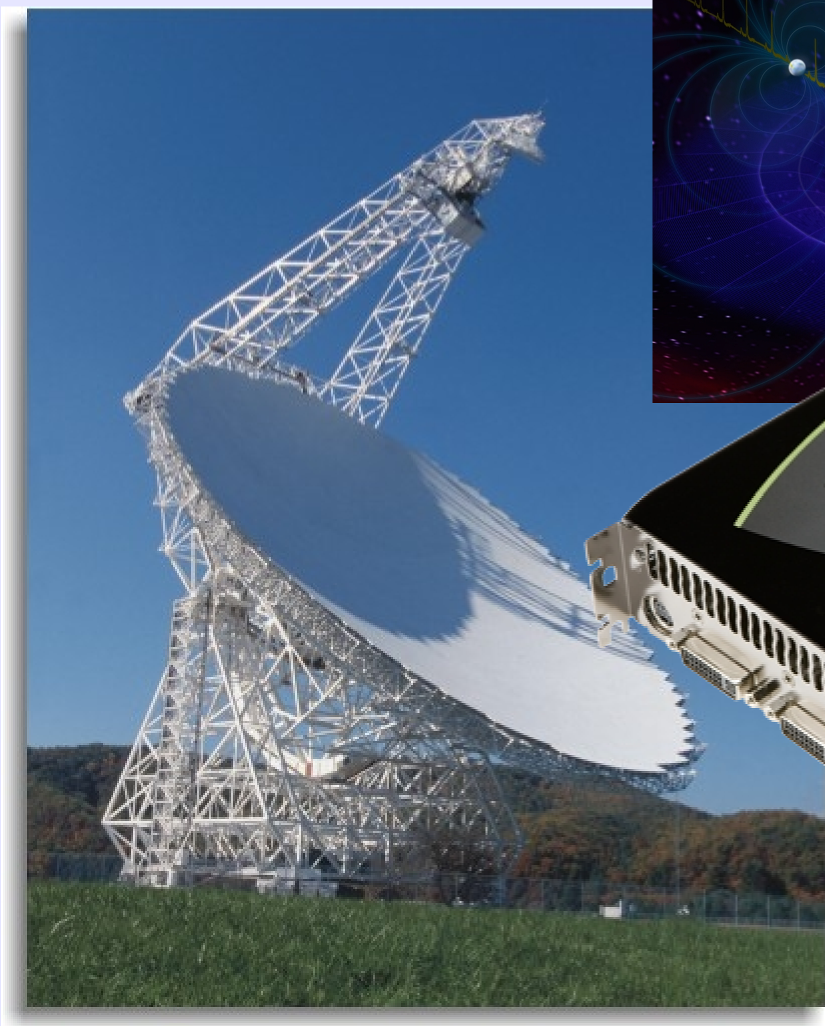
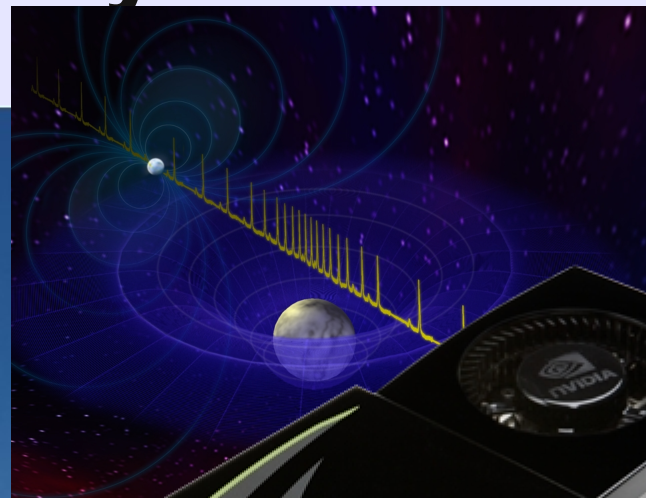
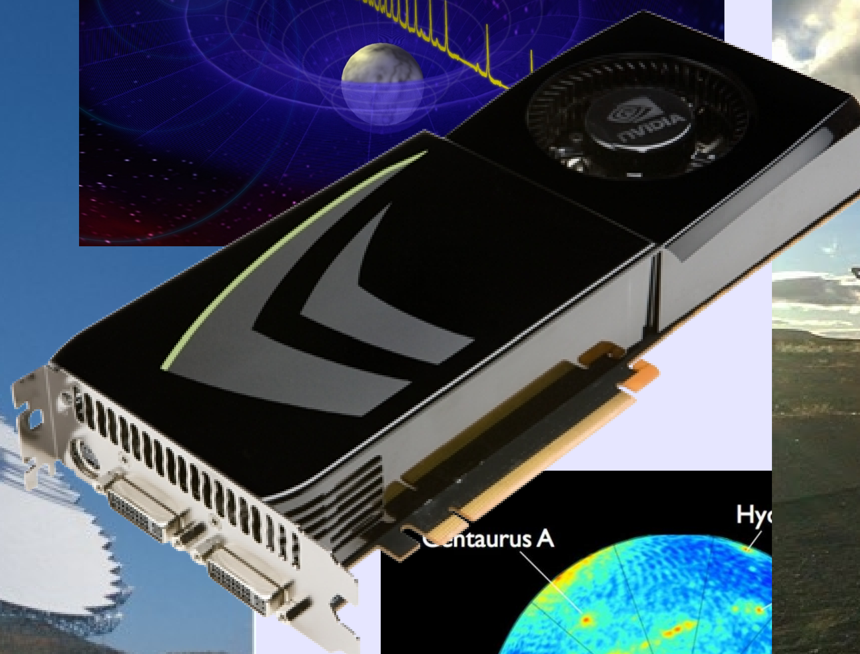
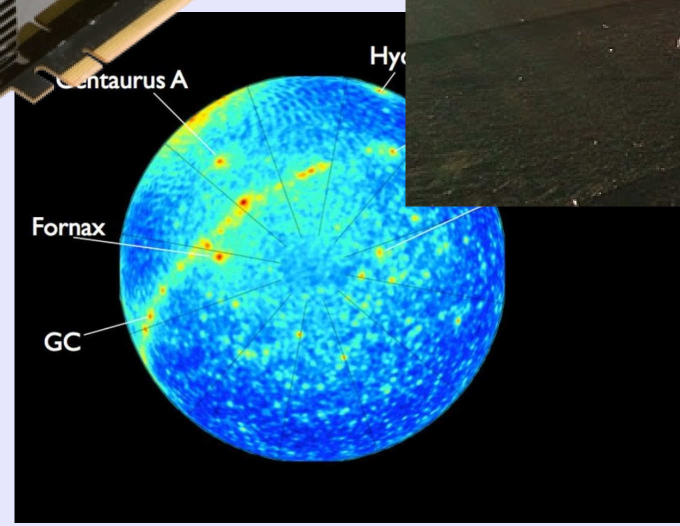
Graphical Processing Units (GPUs) in Radio Astronomy







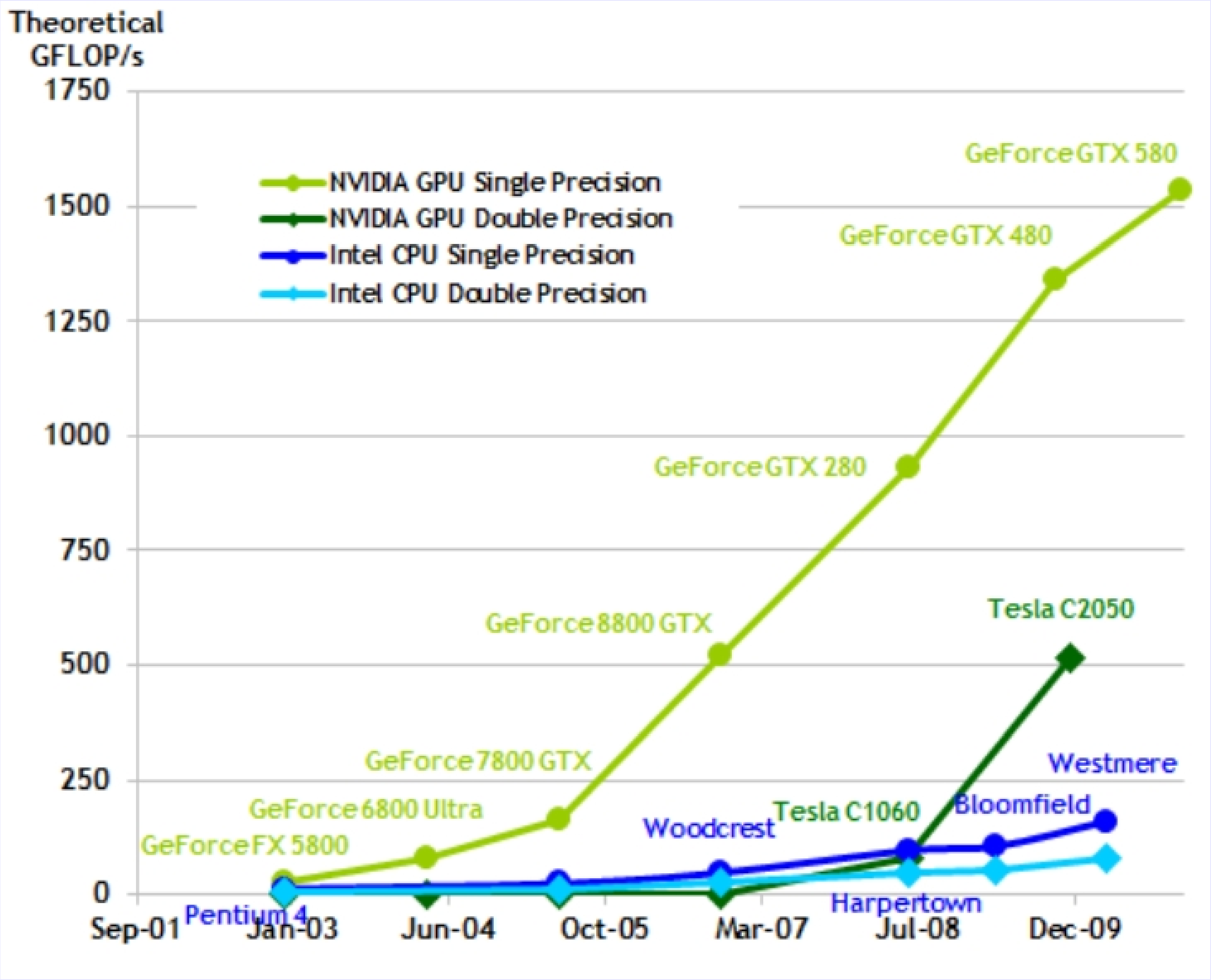


Paul Demorest (NRAO)

Outline

- General purpose computing on GPUs (GPGPU): History, motivations, device characteristics.
- How GPUs fit into radio astronomy instruments and signal processing pipelines.
- GPU programming basics: The devices, progamming languages/tools, useful concepts.
- Examples of GPUs in action!
- Semi-detailed examples: Pulsar instrumentation. (Yes, there will be code!)
- Comments from the audience?

Computing on GPUs - Motivations



(From NVIDIA CUDA Programming Guide)

Computing on GPUs - Motivations

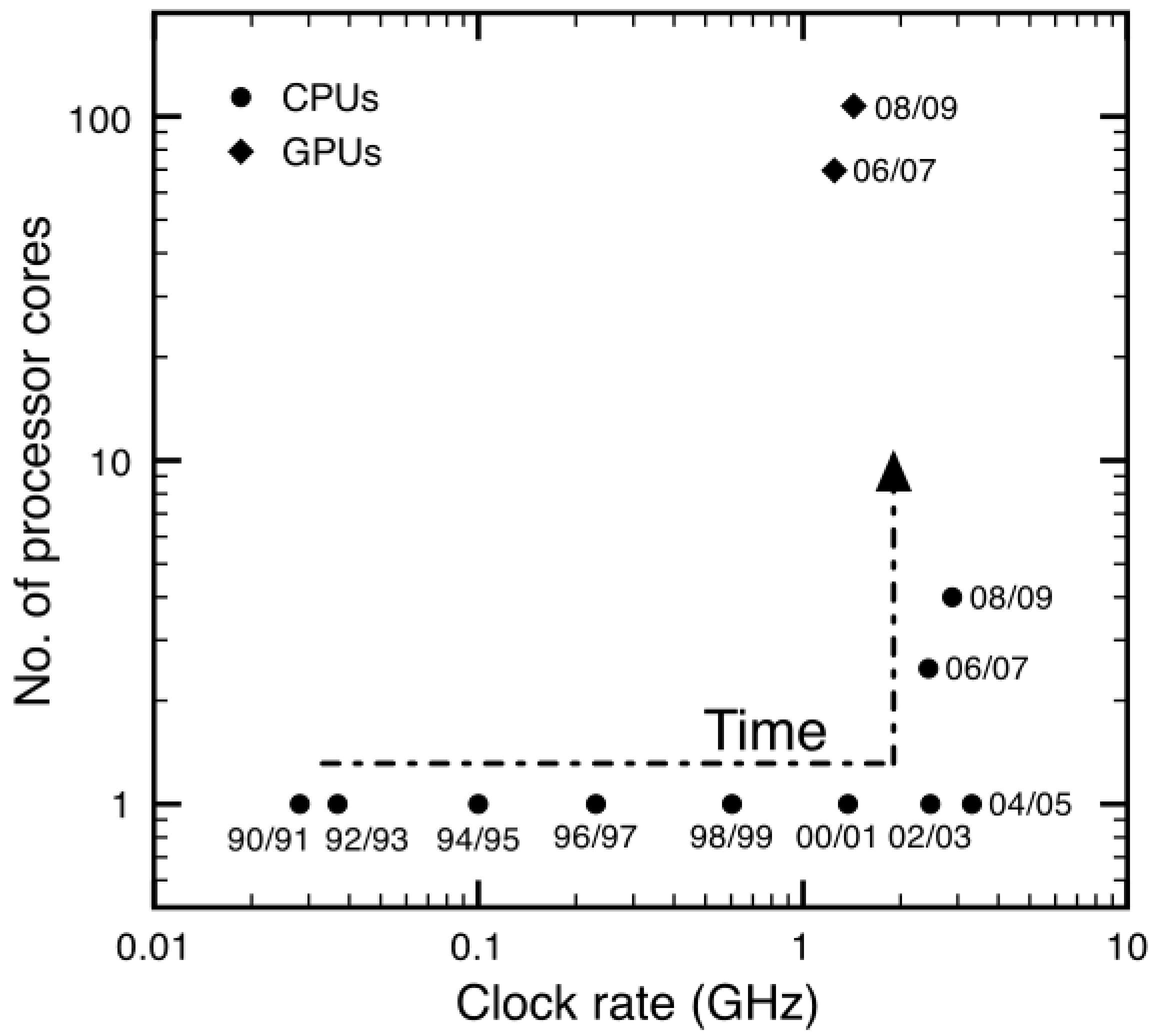
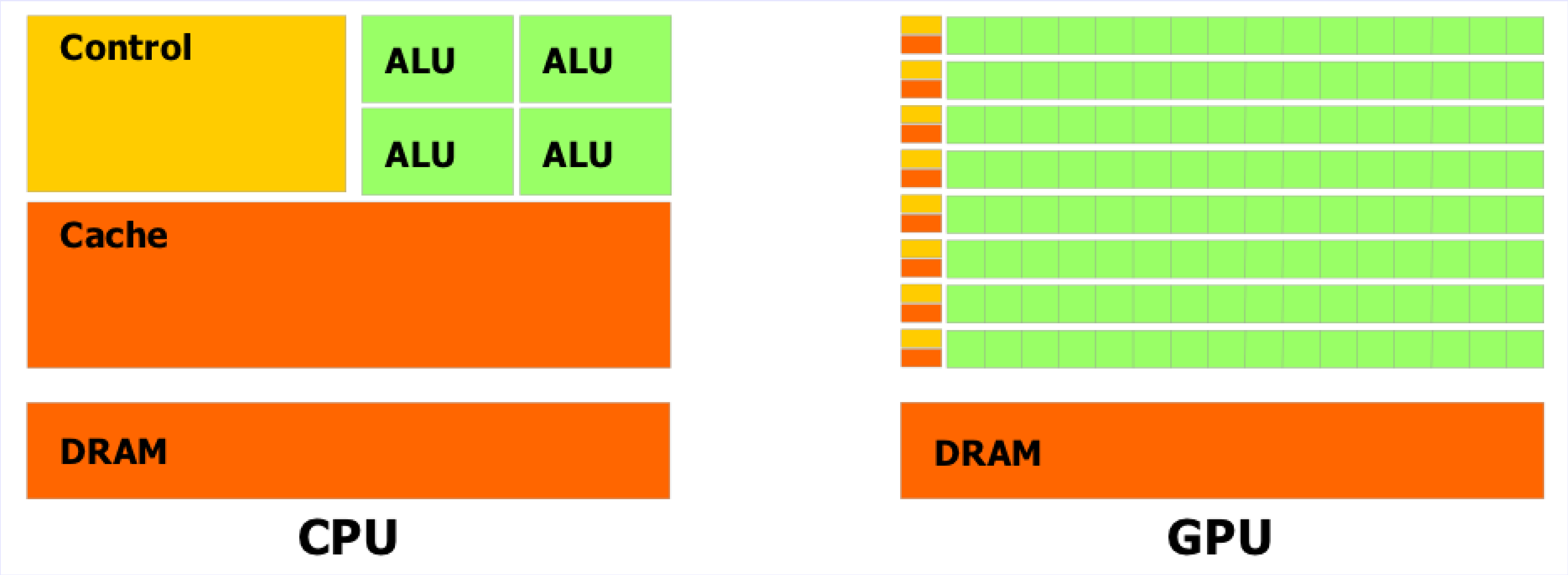


Figure 1. Clock-rate versus core-count phase space of Moore's law binned every 2 yr for CPUs (circles) and GPUs (diamonds). There is a general trend for performance to increase from bottom left to top right.

(Barsdell et al 2010)

GPU capabilities



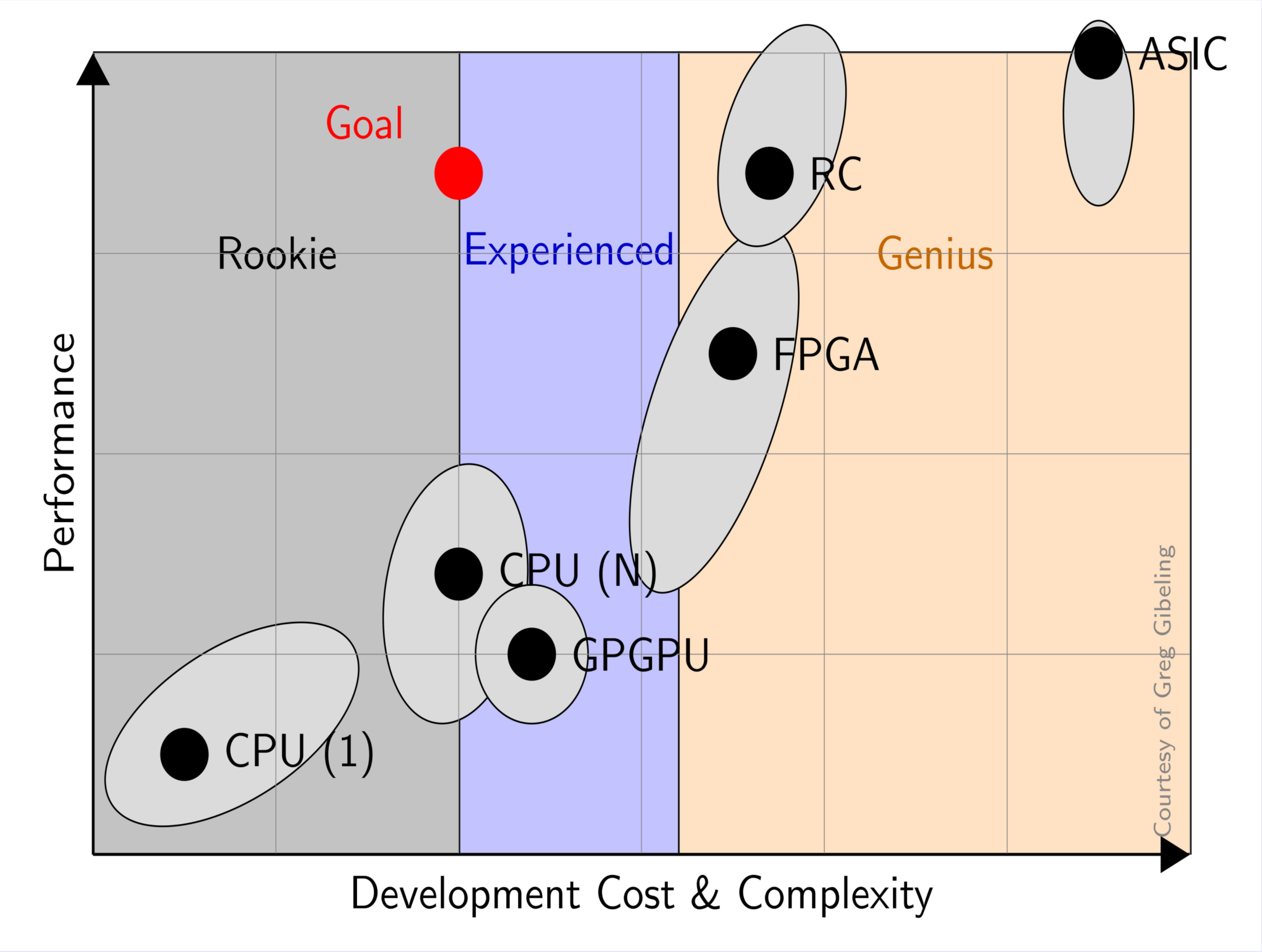
(From NVIDIA CUDA Programming Guide)

- Set of highly parallel (SIMD) “multiprocessors”.
- Best suited for parallel problems with high *arithmetic intensity* – roughly, # operations per sample (or per data transfer) should be in the 100s.

GPUs in radio astronomy

- Most current digital instrument designs for radio astronomy incorporate elements of FPGAs, GPUs, and CPUs, each with different strengths/uses:
  - FPGA – High data rate; small memory; simple algorithms; low power. ADC interfaces; high-BW coarse filterbanks; packetization. Still fairly hard to program!
  - CPU – Low/moderate data rate; large memory; complex algorithms. M&C code; file formatting; networking. Easy to program.
  - GPU – Low/moderate data rate; moderate memory; complex algorithms; high parallel ops/sec. High-res filterbank; coherent dedisp; correlator X-engine.

GPU design complexity



GPU programming tools

- Pre-2006, OpenGL/etc used directly (hard!)
- NVIDIA's Compute Unified Device Architecture (CUDA)
  - Provided free by NVIDIA.
  - Programmed mainly via “C for CUDA”
  - Comes with compiler, dev kit, code samples, good documentation, libraries (FFT, BLAS, etc).
  - First release Nov 2006, currently at v4.1
- Open Computing Language (OpenCL)
  - Industry-supported open standard.
  - Implementations exist for NVIDIA, AMD, Intel, Apple.
  - First release Dec 2008, currently at v1.2

CUDA vs OpenCL

- CUDA advantages:
  - More mature: Bigger userbase, codebase.
  - Supported libraries: CUFFT, etc.
  - New HW features supported quicker.
  - Faster code on NVIDIA HW (maybe)?
- OpenCL advantages:
  - Not vendor-specific.
  - Code is (in principle) portable between different devices.
  - Can be used on parallel CPU architectures also.

Bottom line: Almost all existing astronomy GPU projects are CUDA-based. OpenCL may be more “future-proof” but only if people start using it (chicken/egg)...

NVIDIA GPU devices

- Each line of GPU chips comes packaged as “gaming” (GeForce/GTX) and “computing” (Tesla) boards.
- In Fermi arch, compute-specific boards have fast double-precision floating point enabled.
- “Gaming” boards are entirely appropriate for many of our DSP applications (and cheaper!).

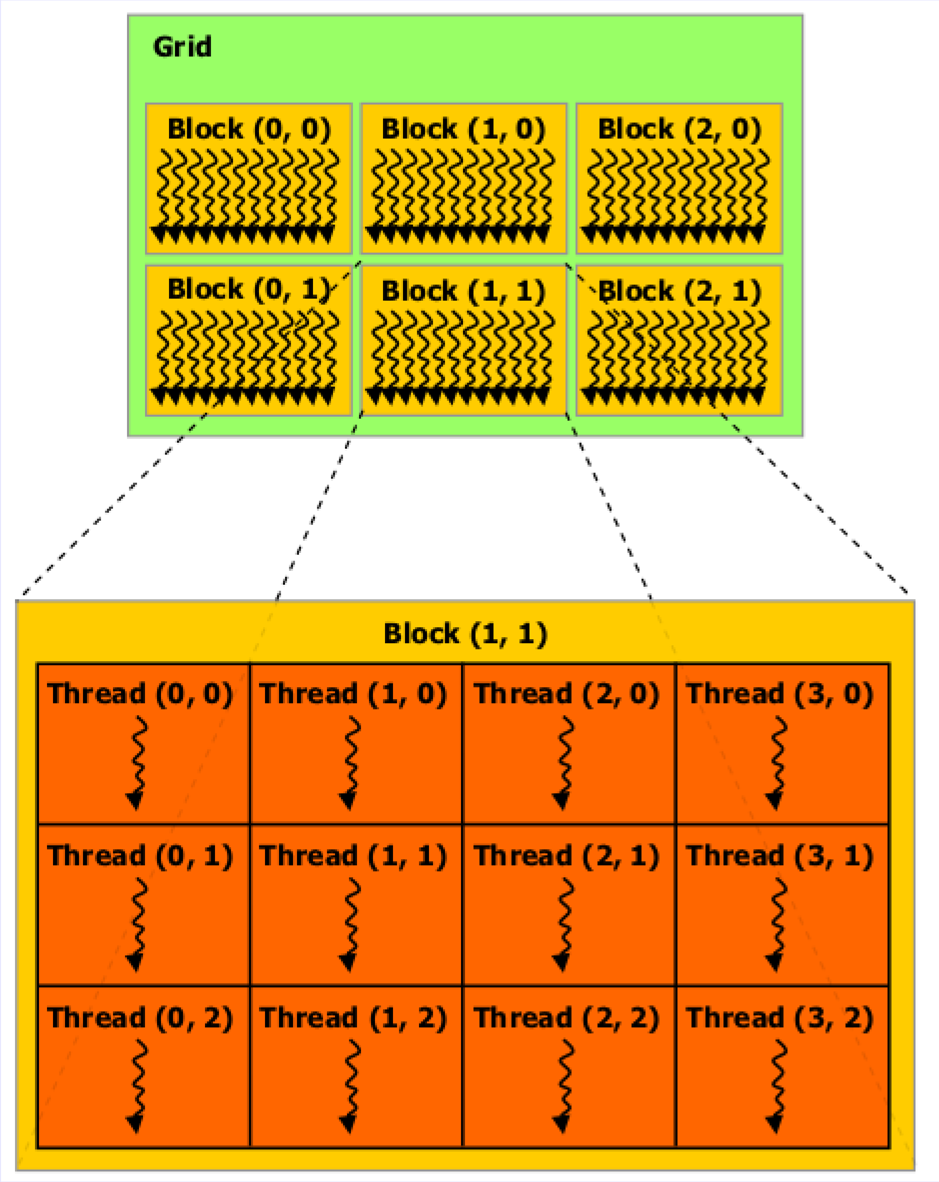
(From NVIDIA CUDA Programming Guide)

CUDA terminology

- “device” - the GPU board.
- “host” - the rest of the computer system (CPUs, memory, etc).
- “kernel” - parallel code that runs on the GPU.
- “thread” - unit of parallel instructions/data within a kernel.
- “stream” - a series of kernels, data transfers, that happen sequentially.

CUDA thread hierarchy

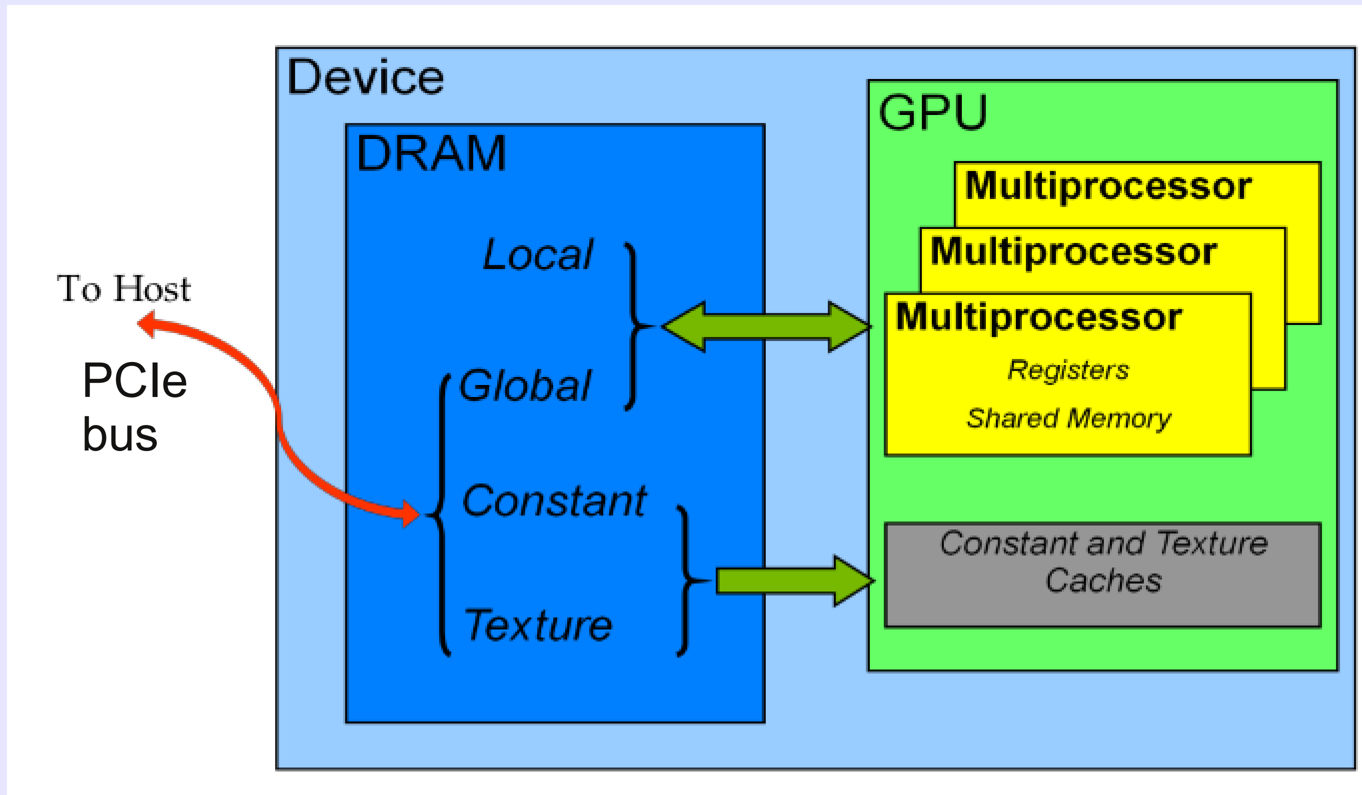
- A kernel is executed as a “grid” of independent thread blocks.
- Each thread block runs on a single multiprocessor unit, and should contain at least 32 threads.



CUDA memory hierarchy

Host

Main (CPU) system RAM



"Off-chip"

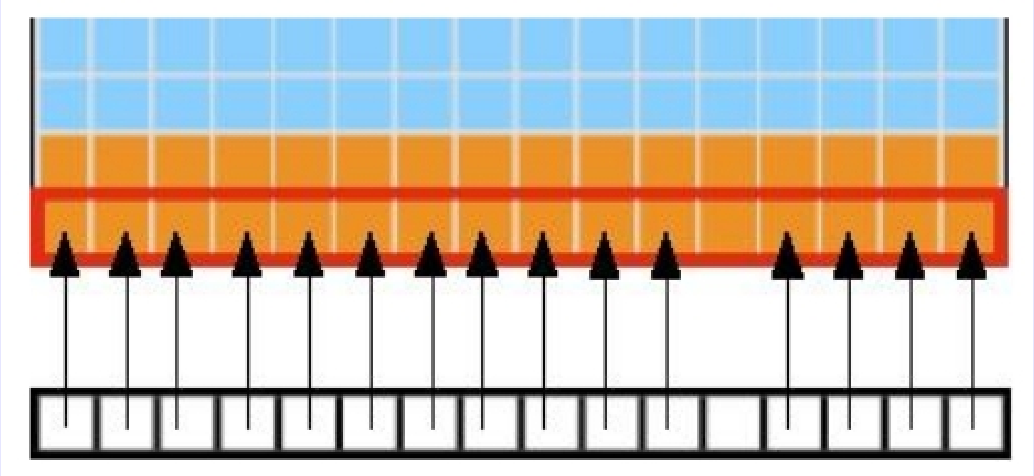
Slow
Large (~GB)

"On-chip" memory

Fast
Small (~16 kB / thread block)

Memory management

- Optimizing memory access has a *huge* effect on GPU code efficiency – a single global mem operation takes several hundred clock cycles.
- Requires much more thought than typical for CPU-based programming.
- Global GPU memory read/writes should be:
  - Minimized
  - “Coalesced”



Global memory

Threads

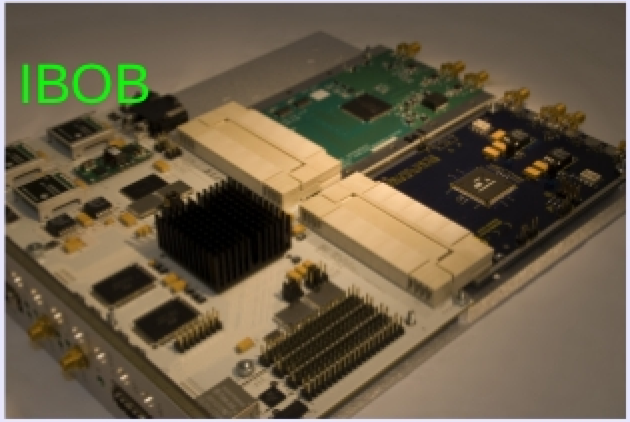
Existing GPU instruments

- Pulsar instruments (coherent dedisp)
  - GUPPI (Green Bank)
  - BON (Nancay)
  - CASPSR (Parkes)
- Transient detectors
  - ARTEMIS (LOFAR)
- Spectrometers
  - VEGAS (Green Bank)
- Array correlators
  - PAPER / LEDA
  - MWA
- … and probably many more!

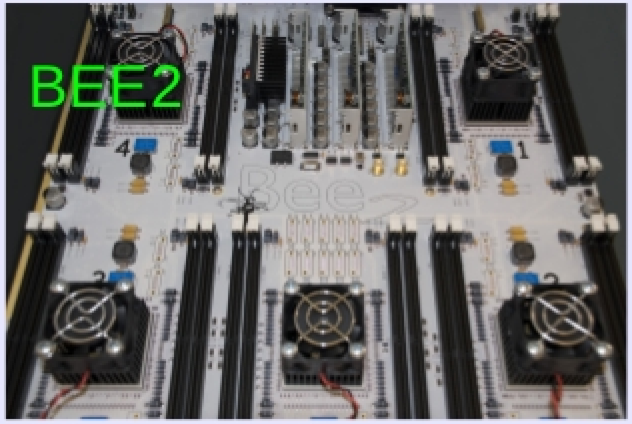
GUPPI pulsar backend

800 MHz total BW coherent dedispersion, 9-node GPU cluster (GTX 285)

GUPPI architecture:
~1 MHz PFB in FPGAs
Coherent dedisp in GPUs



XAUI



10 Ge switch;
24 Gb/s

LEDA / PAPER correlator

<512-input correlators.

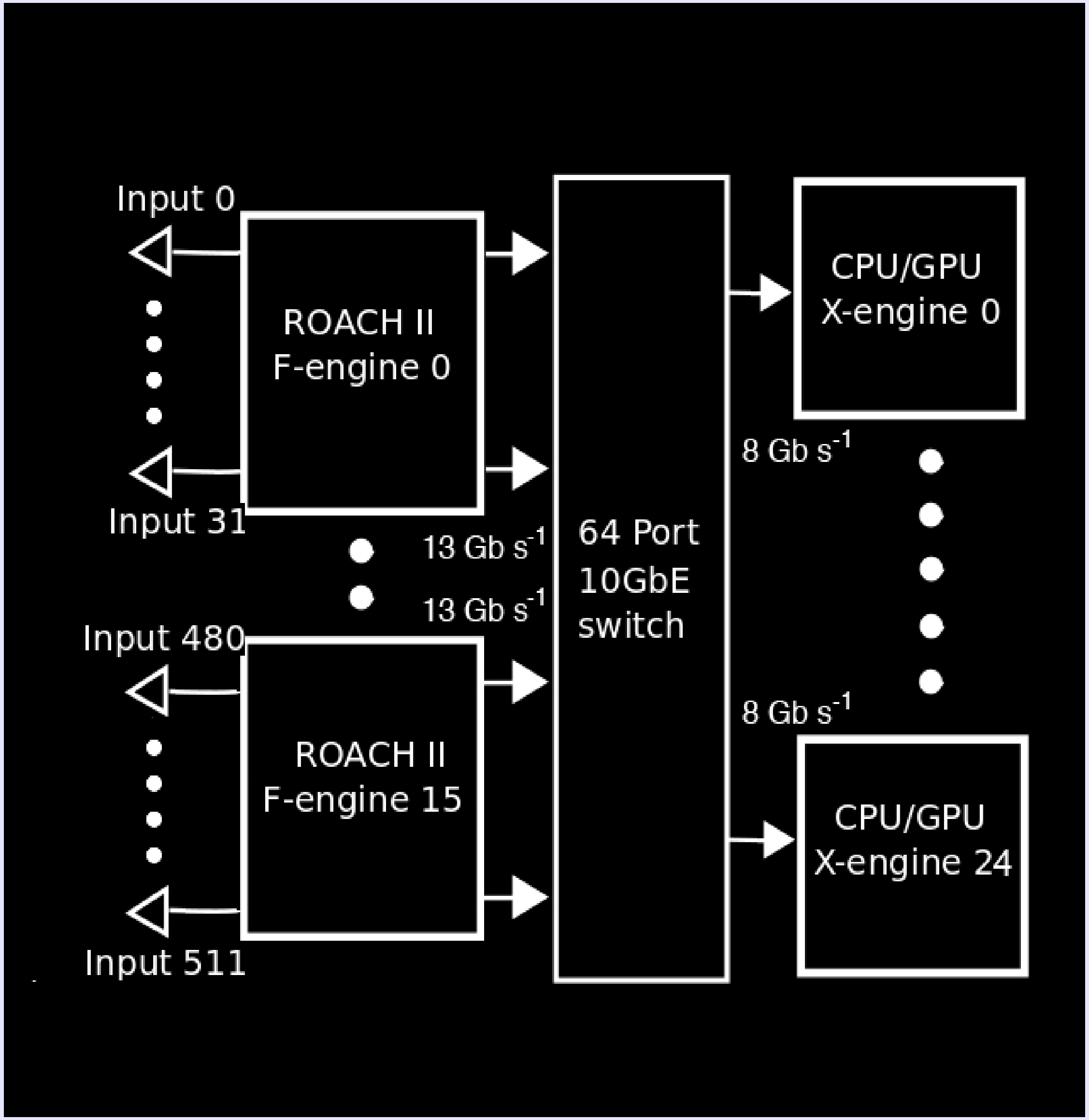
100 MHz BW.

F-engine in 16 roach boards, X-engine in 24 GPUs.

Coded in CUDA, based on work at CfA.

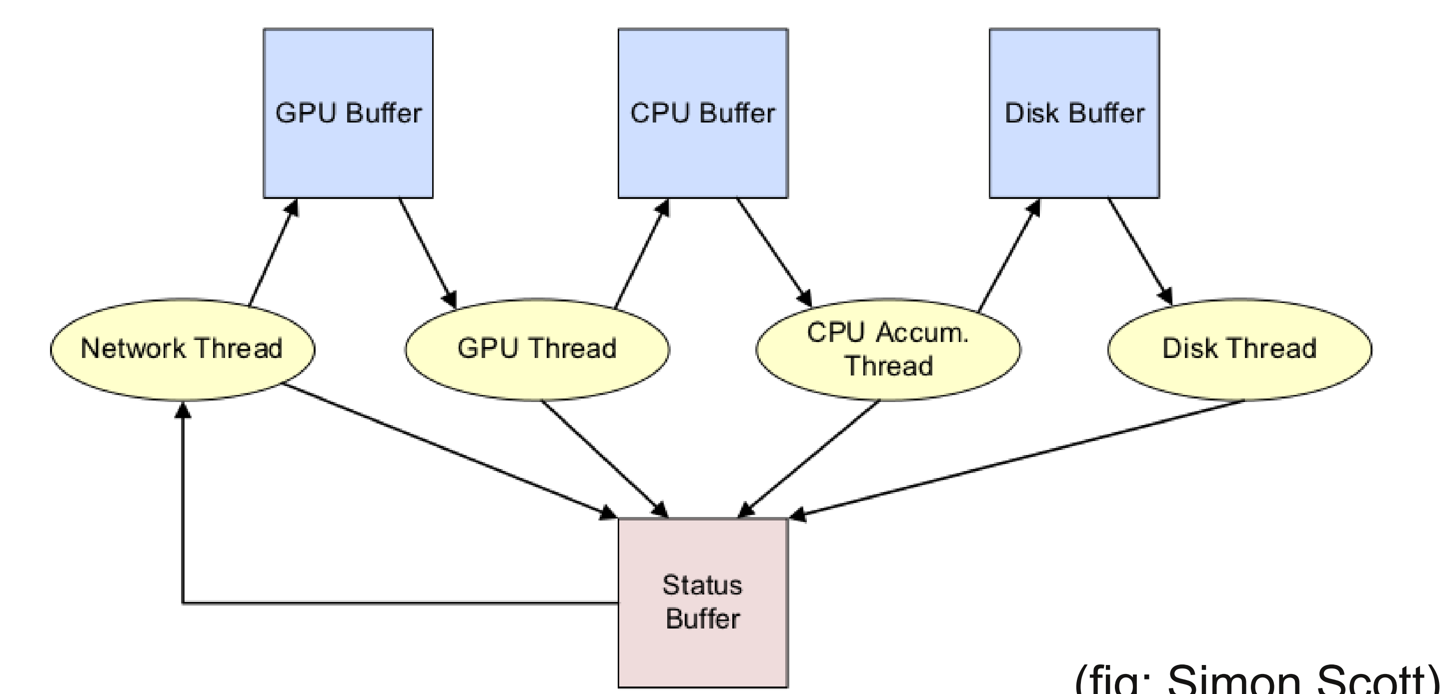
PAPER version to deploy summer 2012.

(Thanks Dan Werthimer and Aaron Parsons!)



VEGAS spectrometer for the GBT

- NRAO/Berkeley project to replace current (old) GBT Spectrometer.
- More ADC bits, more BW/beams, more flexibility.
- Digitally tunable sub-bands in Roach boards, high-res spectra in GPUs.
- Based on GUPPI software architecture ("guppi_daq").



(fig: Simon Scott)

ARTEMIS transient search

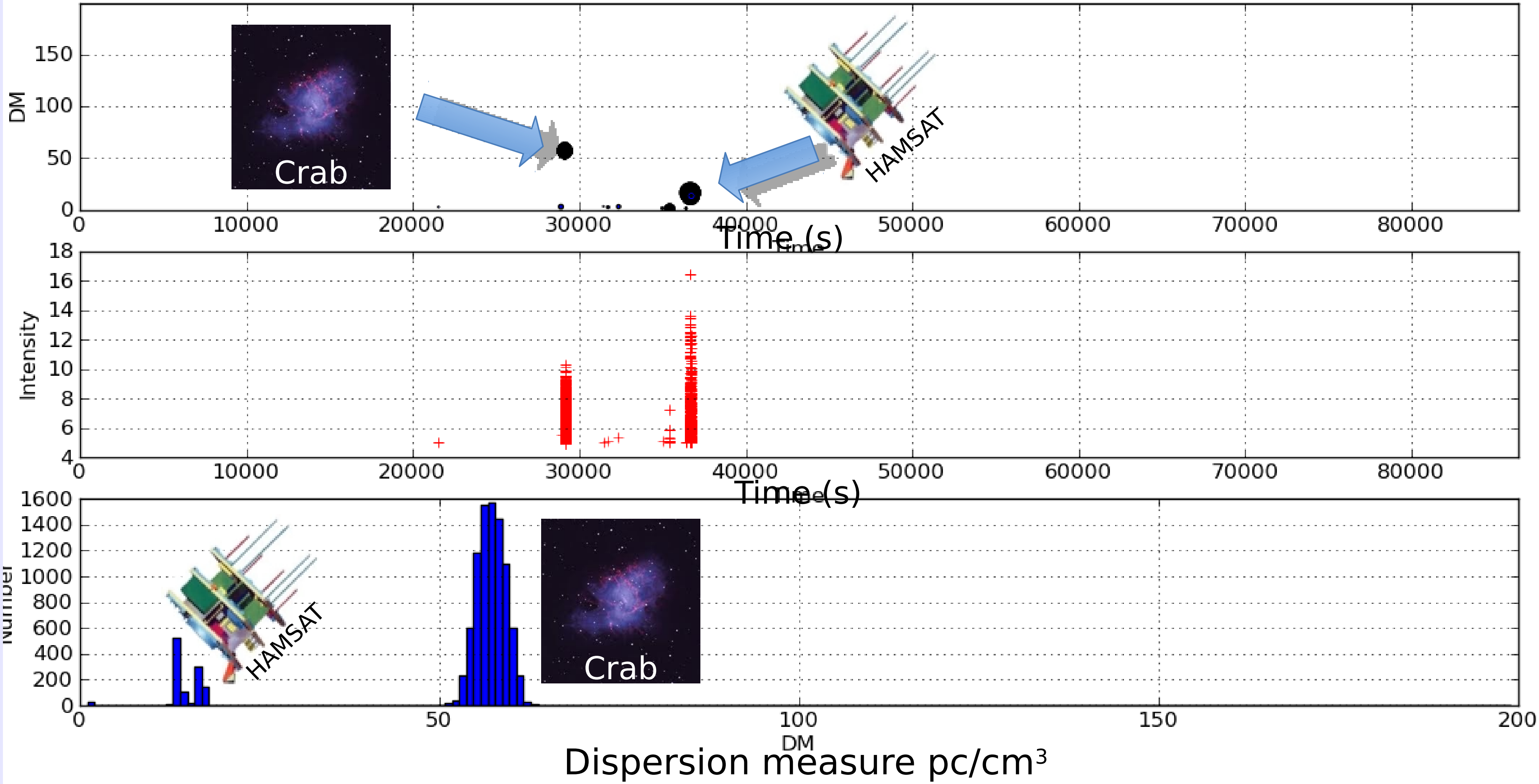
Thanks Aris Karastergiou!
(aris@astro.ok.ca.uk)

Receive udp stream → Polyphase filter → Raw to Stokes → RFI removal → Dedispersion → output

- Working prototype for SKA non-image processing.
- Real-time search for radio pulses using CPU/GPU.
- Dedispersion over 4000 DMs in real time in GPU.
- 150 MHz LOFAR station beam is 2.5° FWHM.
- Pilot survey 1: 6-8 beams, circumpolar targets
- Pilot survey 2: 6 beams fixed on meridian (8-28° dec)
- Each beam is 800 Mb/s (12.5 MHz) processed by 1 node = 12 Xeon cores + M2050/GTX

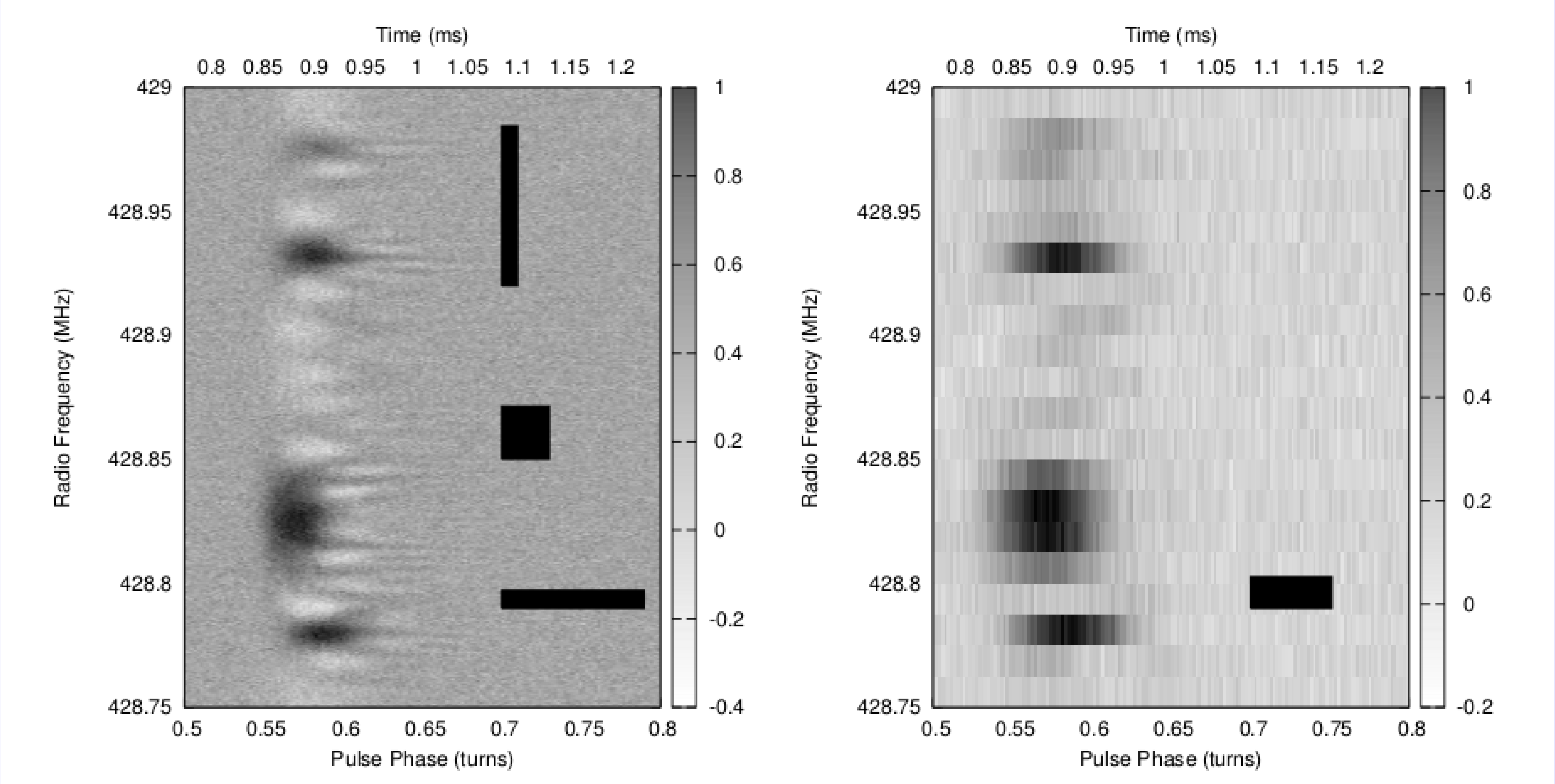
ARTEMIS transient search

Pipeline diagnostic plot from drift survey – daily summary from one of six beams



Cyclic spectroscopy for pulsars

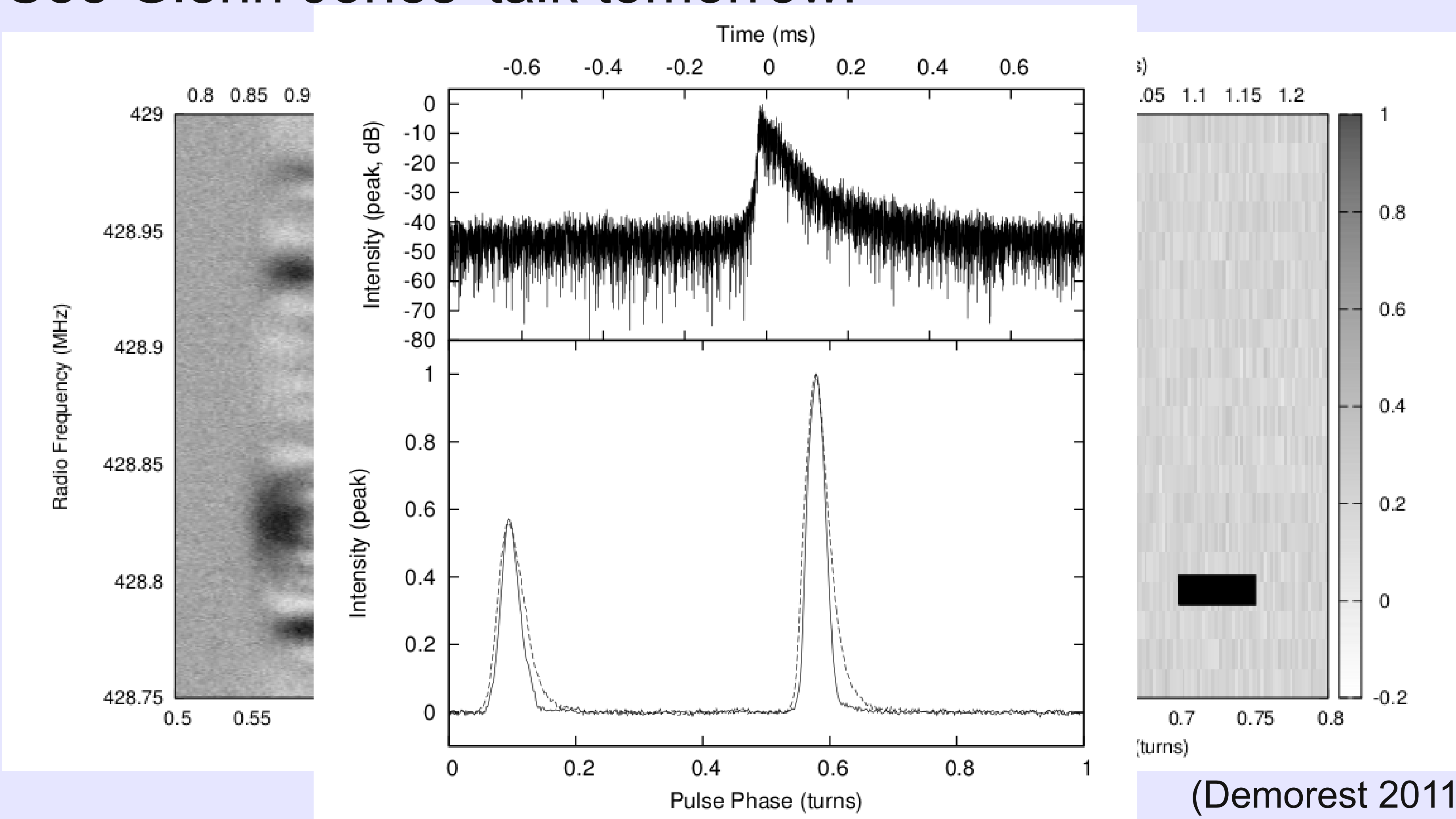
- Allows “de-scattering” of ISM response.
- Much more computation than coherent dedisp.
- See Glenn Jones' talk tomorrow!



(Demorest 2011)

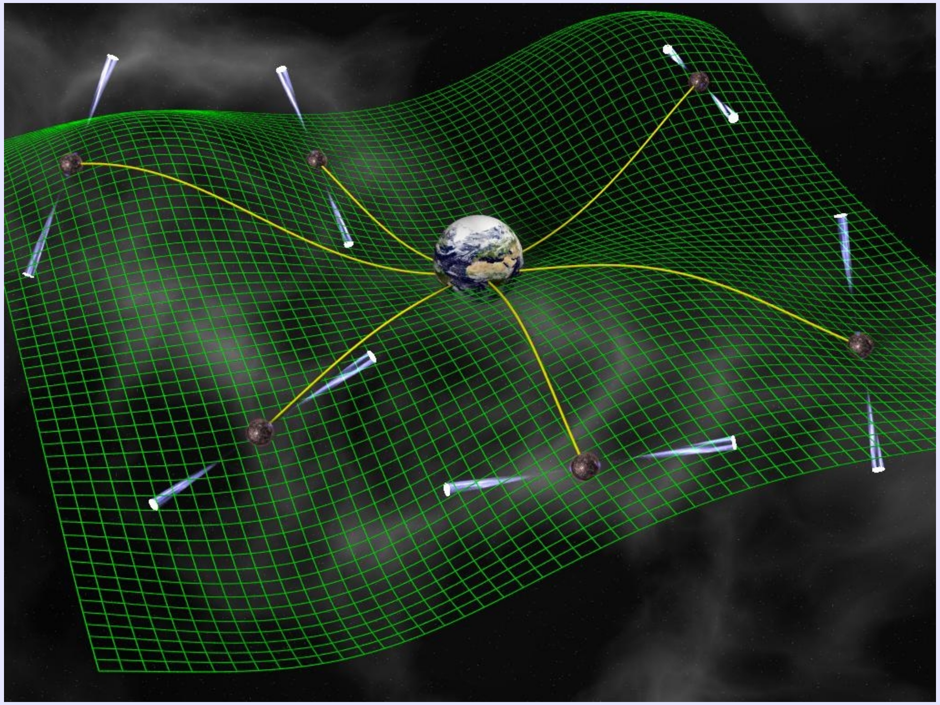
Cyclic spectroscopy for pulsars

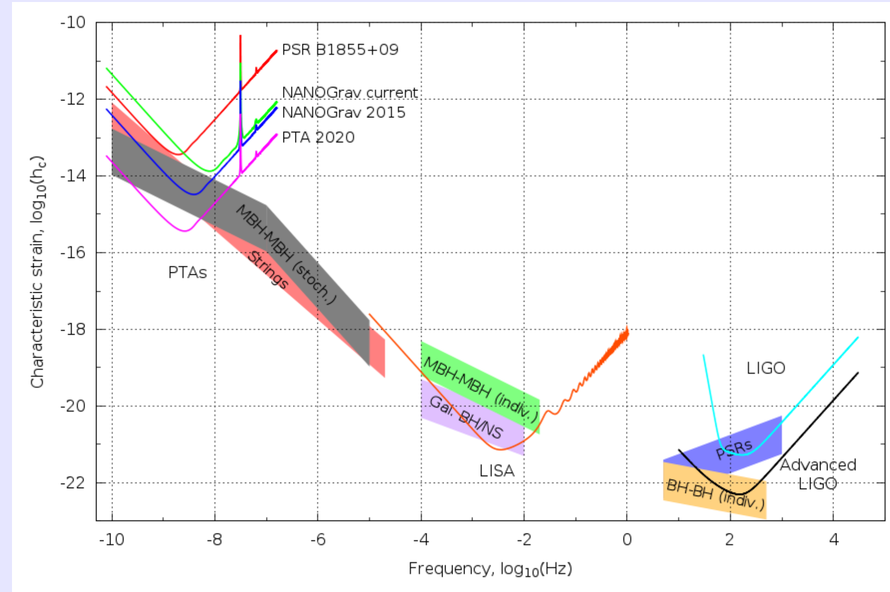
- Allows “de-scattering” of ISM response.
- Much more computation than coherent dedisp.
- See Glenn Jones' talk tomorrow!

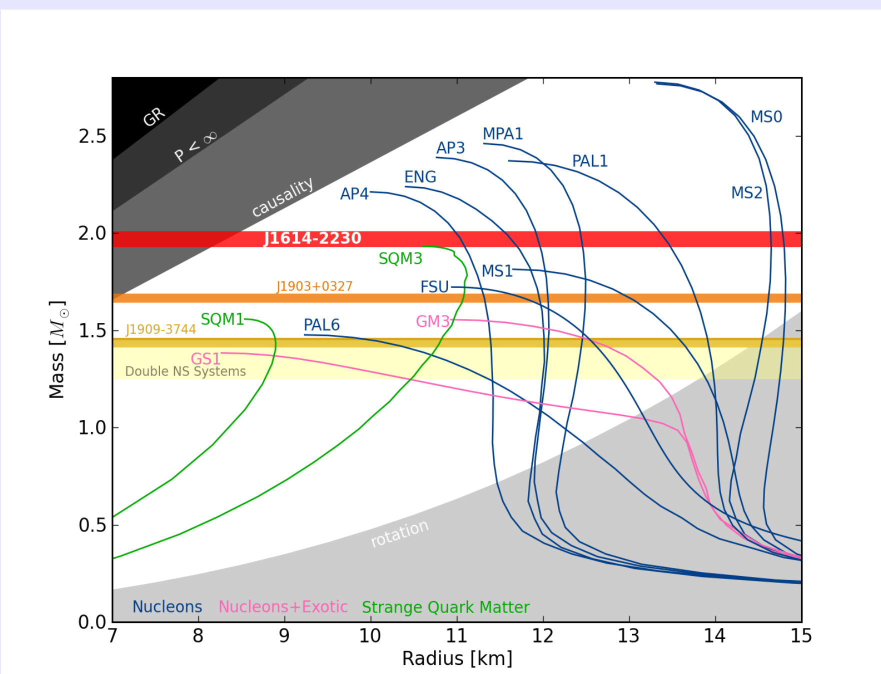


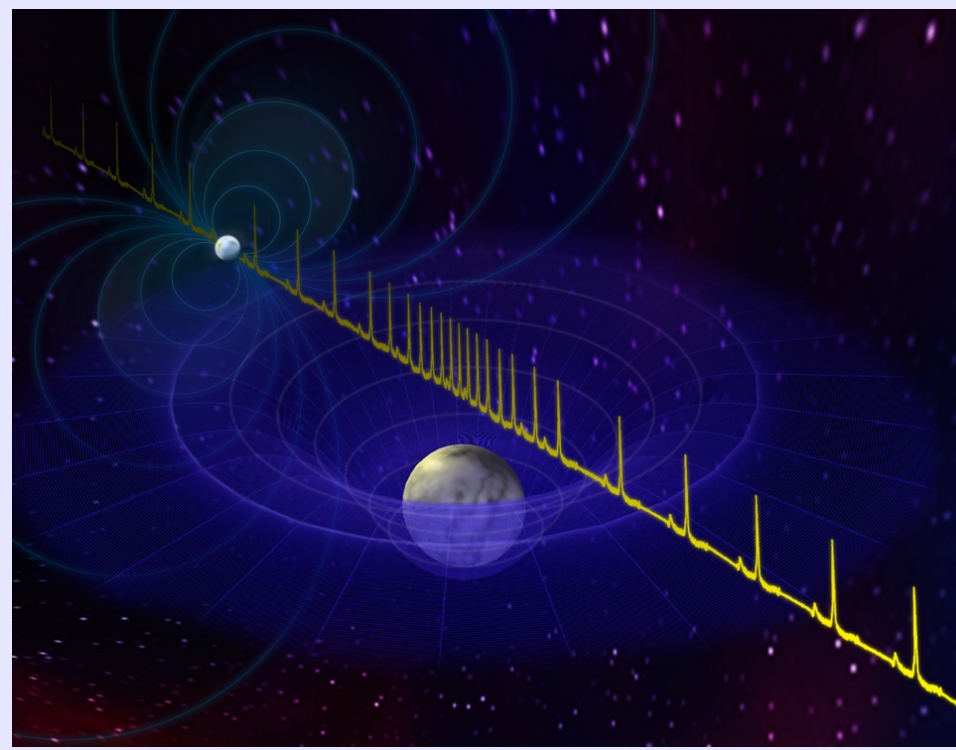
(Demorest 2011)

Instrumentation for pulsar timing - motivation



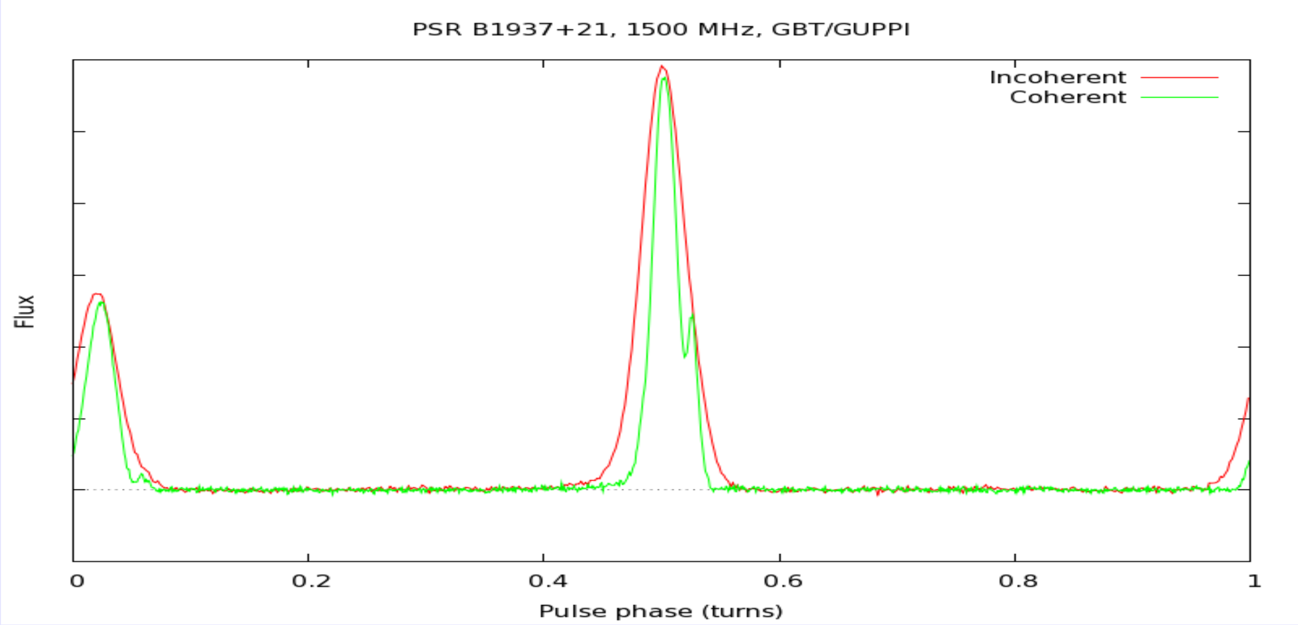






Design study: Pulsar instrumentation

- Want as much BW as possible (currently ~1 GHz BW at L-band); split into ~MHz channels for ~us time resoln.
- Need to do 100 MHz total BW per GPU.
- In GUPPI, single-channel data comes into GPU, then:
    - Unpack (8-bit to float)
    - Dedisperse (FFT-mult-IFFT; ~10k to 1M-points)
    - "Fold" modulo current pulse period:



First steps: FFT

CUFFT interface very similar to FFTW.

Plan:

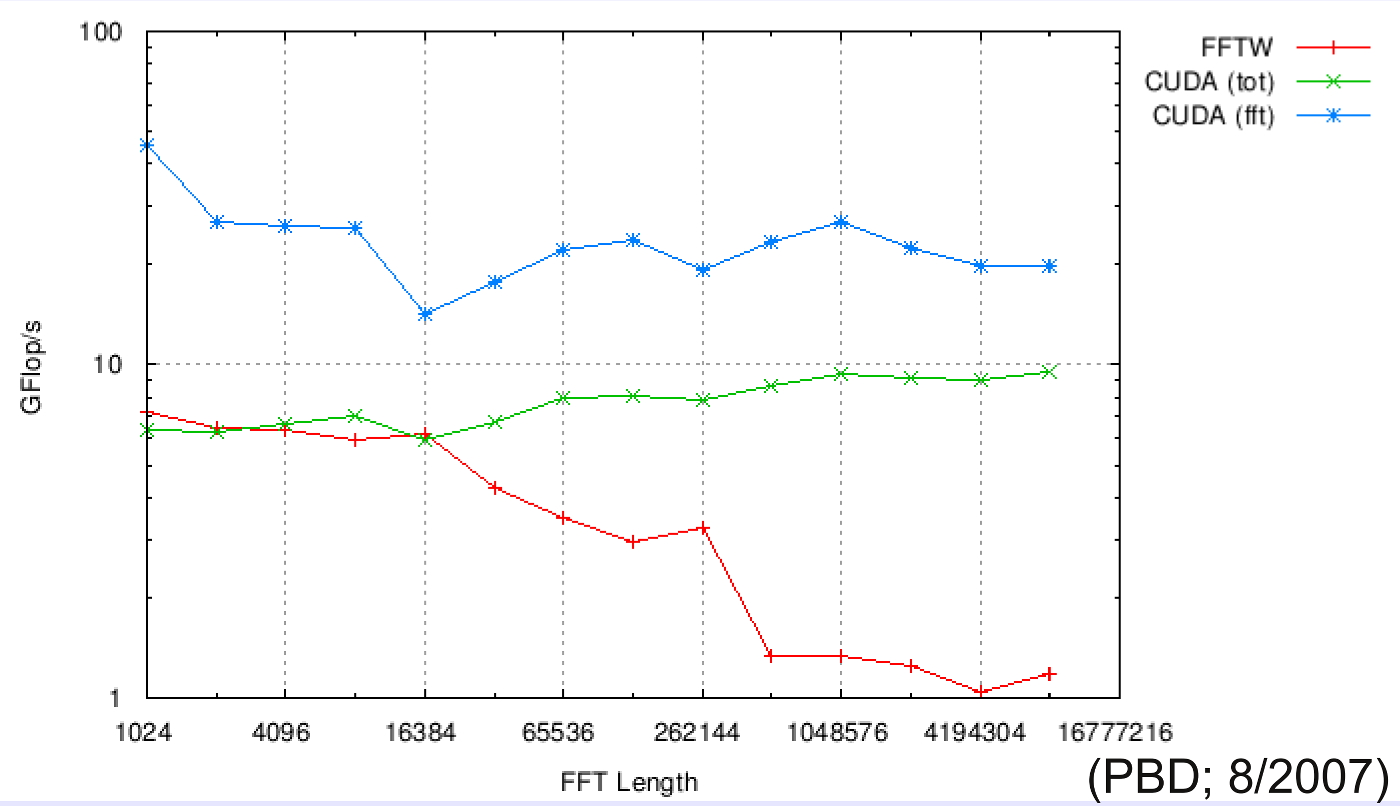
```
// Plan FFT
cufftResult fft_rv =
    cufftPlan1d(&s->plan, s->fft_len, CUFFT_C2C, 2*s->nfft_per_block);
```

Exec:

```
/* Forward FFT */
fft_rv = cufftExecC2C(s->plan, s->databuf0_gpu, s->databuf0_gpu,
        CUFFT_FORWARD);
cudaEventRecord(t[it], 0); it++;
```

Very easy introduction to using GPU/CUDA!

(Note: plot shows very old result!)



(PBD; 8/2007)

A simple kernel: 8-bit unpack

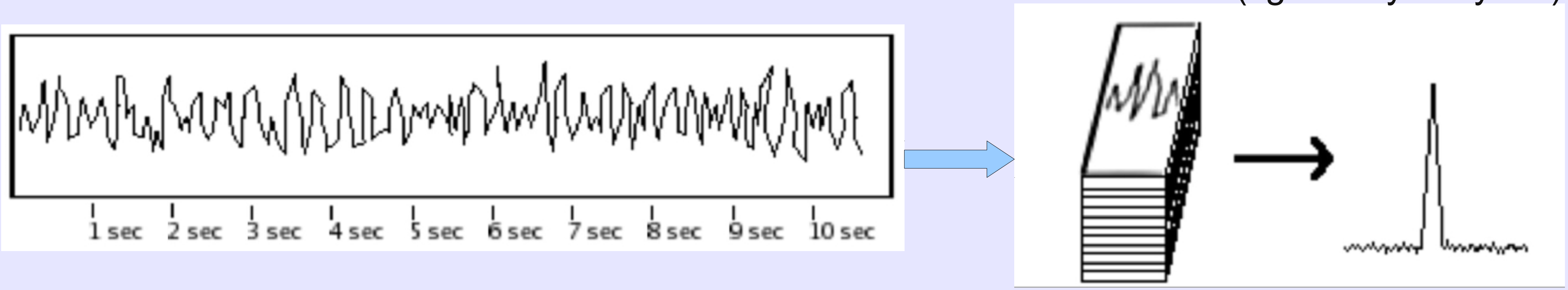
```
/* CUDA kernel to convert bytes to floats.  Also splits incoming
 * data into two polarizations (assuming polns are interleaved
 * in the raw data).
 */
__global__ void byte_to_float_2pol_complex(
        unsigned short *in, float2 *outx, float2 *outy,
        size_t n) {
    const int nt = blockDim.x * gridDim.x;
    const int tId = blockIdx.x * blockDim.x + threadIdx.x;
    char4 *in_8bit = (char4 *)in;
    for (int i=tId; i<n; i+=nt) {
        outx[i].x = __int2float_rn(in_8bit[i].x);
        outx[i].y = __int2float_rn(in_8bit[i].y);
        outy[i].x = __int2float_rn(in_8bit[i].z);
        outy[i].y = __int2float_rn(in_8bit[i].w);
    }
}
```

```
    /* Convert to floating point */
    byte_to_float_2pol_complex<<<16,128>>>((unsigned short *)s->overlap_gpu,
            s->databuf0_gpu, s->databuf1_gpu, npts_tot);
    cudaEventRecord(t[it], 0); it++;
```

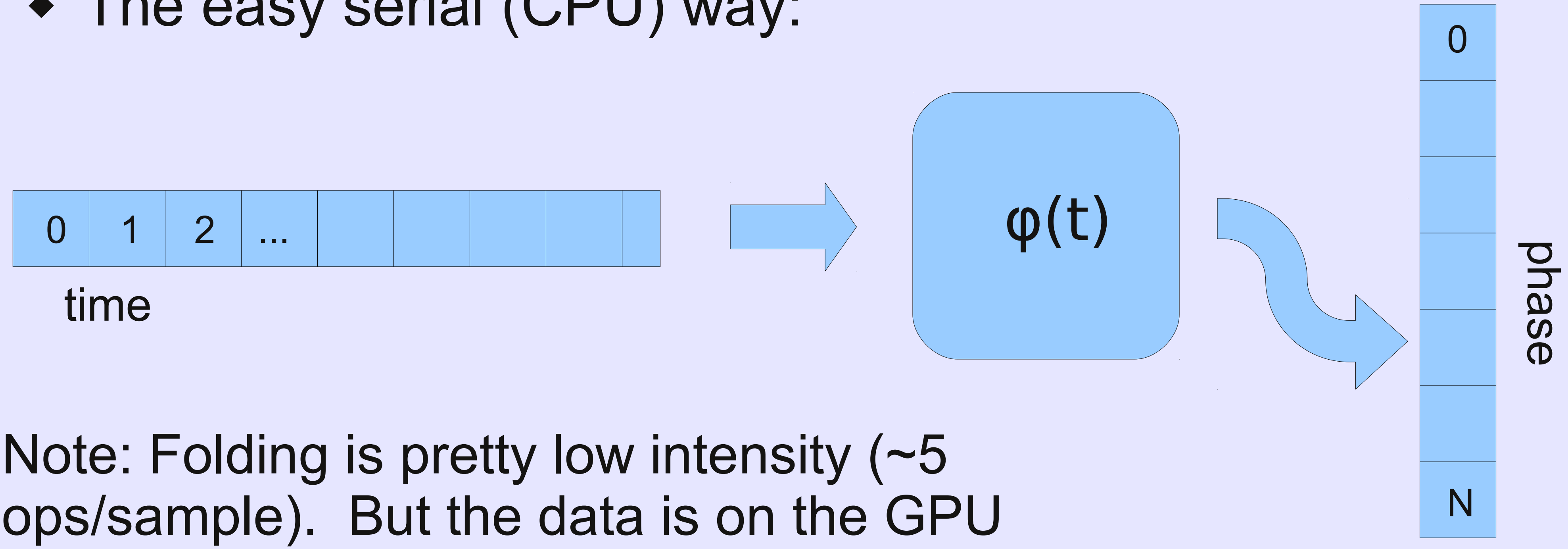
(Directly from guppi_daq code; dedisperse_gpu.cu)

Non-trivial: Pulse period folding

(figure: Ryan Lynch)

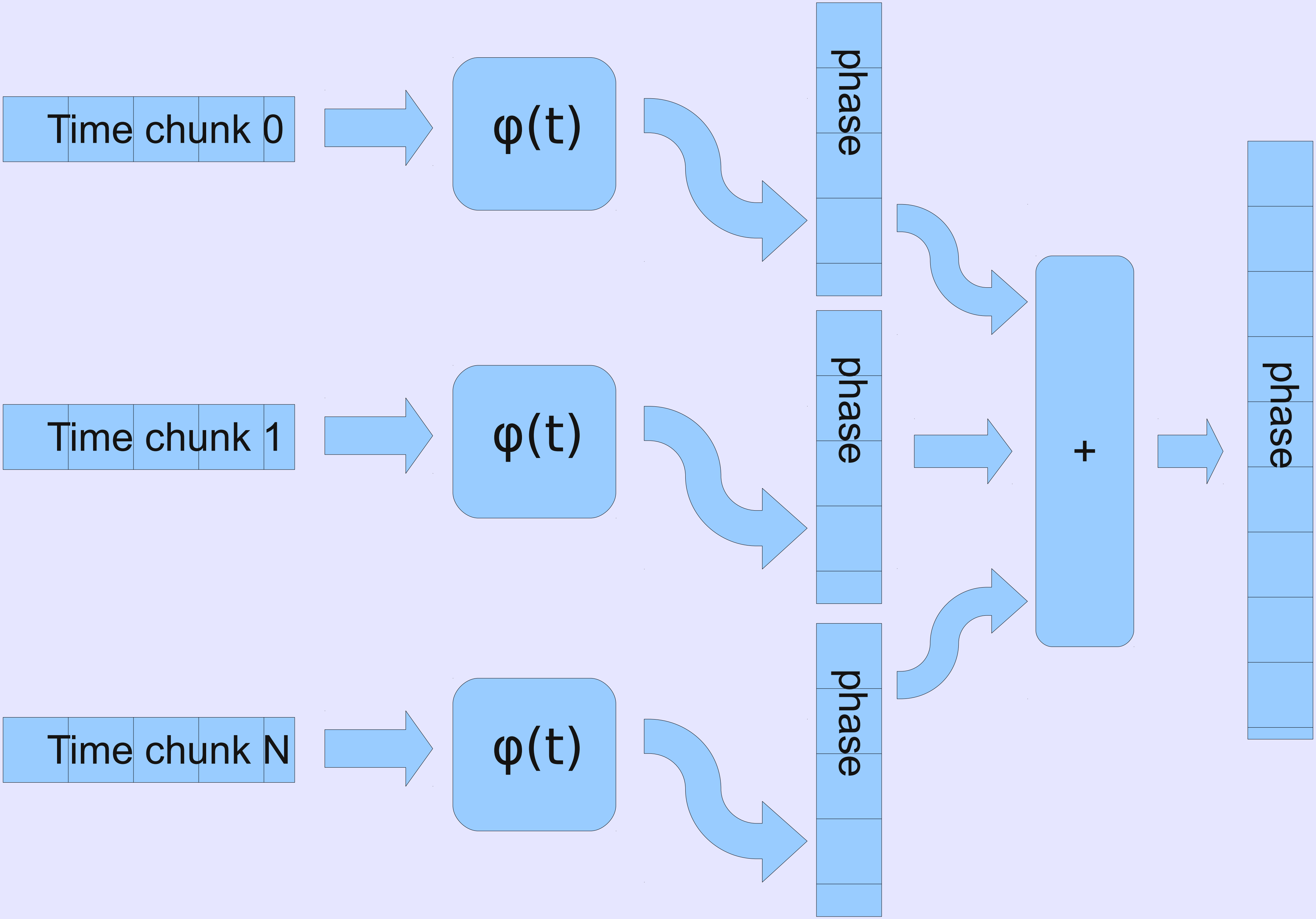


- The easy serial (CPU) way:

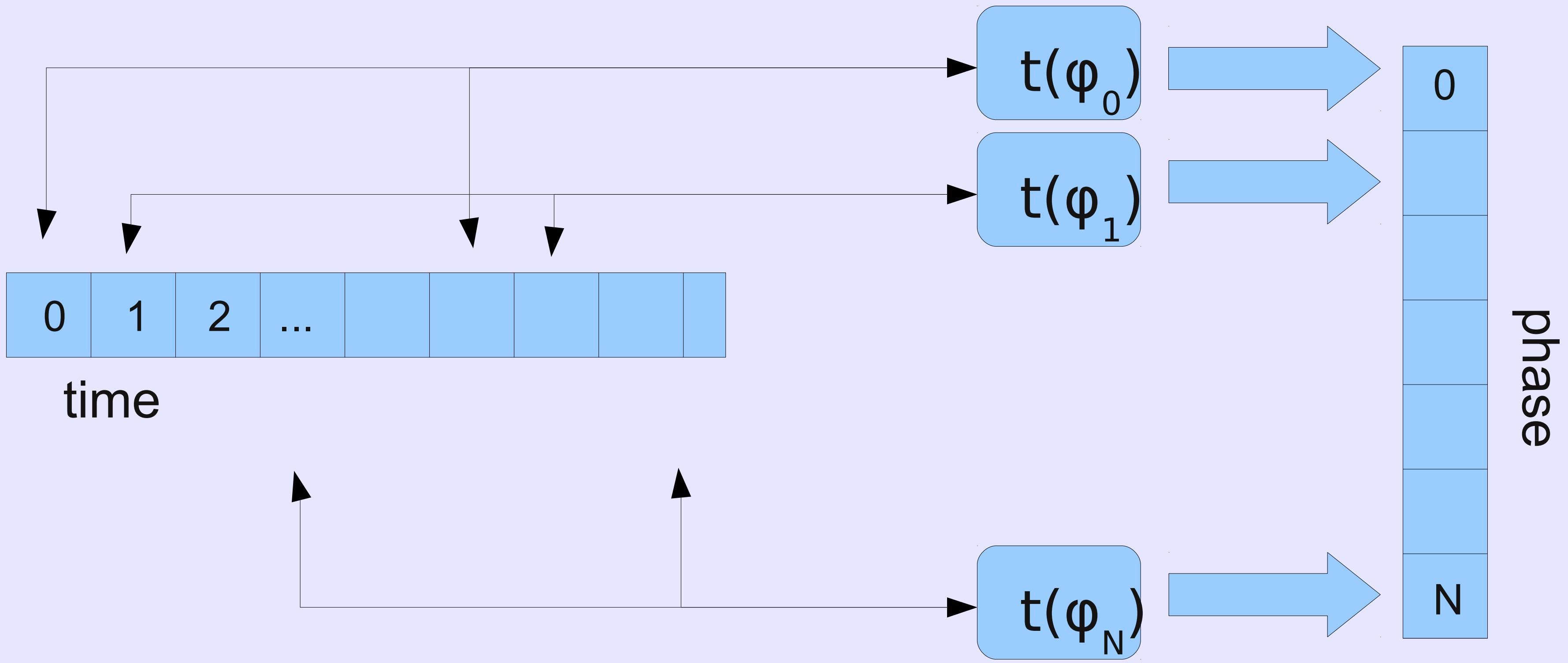


Note: Folding is pretty low intensity (~5 ops/sample). But the data is on the GPU already, so we want to reduce it there..

Folding, first GPU attempt:



GPU folding, a better way:



Each thread handles one pulse phase bin, pulling appropriate samples from input array.

Folded data accumulates in thread-local memory, resulting in less global mem writes = much faster!

GPU folding code

```
// Loop over number of pulse periods in data block
for (int iturn=0; iturn<nturn; iturn++) {

    // Determine range of samples needed for this bin, turn
    int samp0 = samp_bin*((double)bin_lo-bin0+(double)iturn*nbin)+0.5;
    int samp1 = samp_bin*((double)bin_lo-bin0+(double)iturn*nbin+1)+0.5;

    // Range checks
    if (samp0<0) { samp0=0; }
    if (samp1<0) { samp1=0; }
    if (samp0>nvalid) { samp0=nvalid; }
    if (samp1>nvalid) { samp1=nvalid; }

    // Read in and add samples
    for (int isamp=samp0; isamp<samp1; isamp++) {
        float2 p0 = ptr0[isamp];
        float2 p1 = ptr1[isamp];
        folddata.x += p0.x*p0.x + p0.y*p0.y;
        folddata.y += p1.x*p1.x + p1.y*p1.y;
        folddata.z += p0.x*p1.x + p0.y*p1.y;
        folddata.w += p0.x*p1.y - p0.y*p1.x;
        foldcount++;
    }
}

// Copy results into global mem
const unsigned prof_offset = ifft * nbin;
foldtmp[prof_offset + bin_lo].x = folddata.x;
foldtmp[prof_offset + bin_lo].y = folddata.y;
foldtmp[prof_offset + bin_lo].z = folddata.z;
foldtmp[prof_offset + bin_lo].w = folddata.w;
```

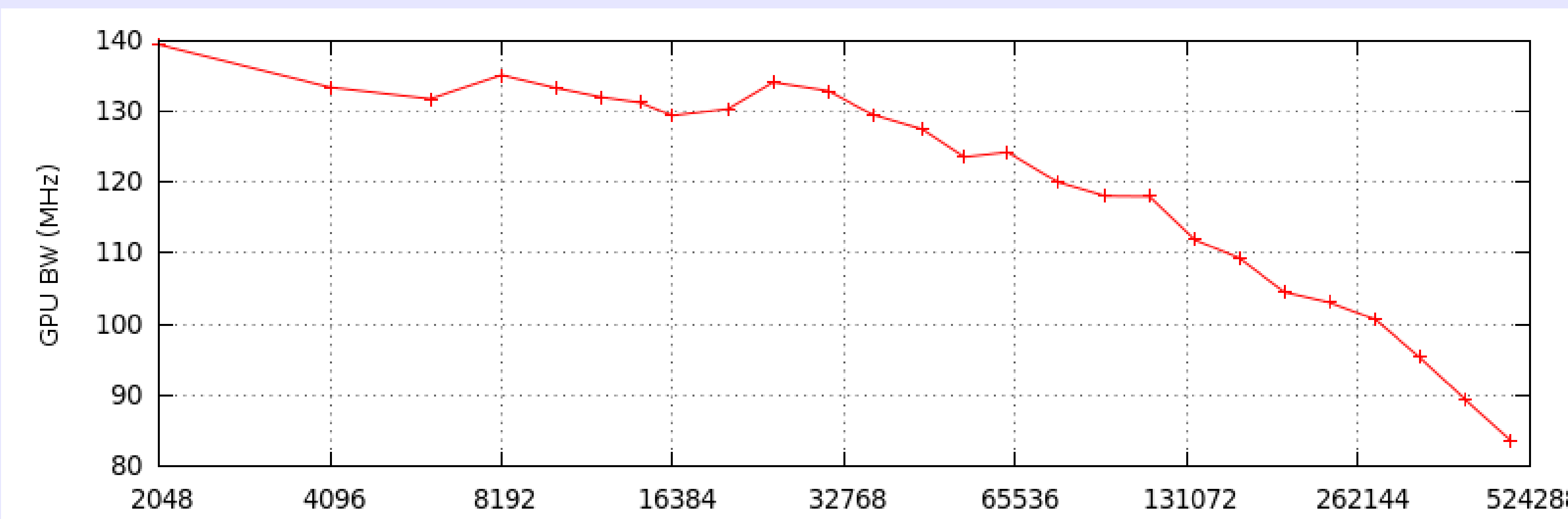
(guppi_daq/fold_gpu.cu)

Putting it all together

From recent GBT observation of B1937+21 at 820 MHz:

```
Total time  =  158.9 s (4.1900 ns/samp)
Total2 time =  158.7 s (4.1846 ns/samp)
  0.669 ns   15.97% transfer_to_gpu
  0.099 ns    2.35% overlap
  0.319 ns    7.62% bit_to_float
  2.199 ns   52.48% fft
  0.386 ns    9.22% xmult
  0.040 ns    0.96% fold_mem
  0.444 ns   10.59% fold_blocks
  0.009 ns    0.22% fold_combine
  0.000 ns    0.00% downsample
  0.019 ns    0.46% transfer_to_host
Closing file '/data/gpu/partial/gpu1/guppi_55919_B1937+21_0026_0001.fits'
```

Goal was 100 MHz BW per GPU (10 ns/samp).



Chirp length (proportional to DM)

The end!

- Questions or comments are of course welcome! Please share your own GPU experiences.
- Useful links:
  - http://developer.nvidia.com/cuda-downloads
  - http://www.khronos.org/opencl
  - http://github.com/demorest/guppi_daq
  - http://dspsr.sourceforge.net