GPU version of the Polgraw all-sky F-statistic pipeline

M. Bejger (Copernicus Center)

collaboration with

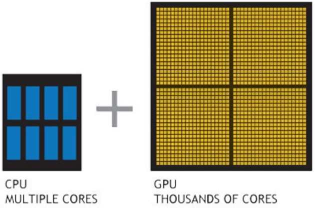
Jan Bolek (Warsaw Technical University),
Paweł Ciecieląg (Copernicus Center),
Aleksander Garus (ETH Zürich),
Andrzej Królak (IMPAN).

Outline

- ⋆ CPU vs GPU concept,
- ⋆ description of the all-sky F-stat search for candidate signals,
- ⋆ Implementation of the GPU version,
- ⋆ CPU version performance testing.

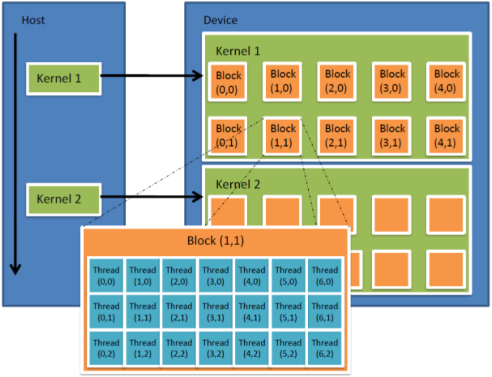
Central Processing Units vs Graphics Processing Units

CPU: a **few** cores optimized for **sequential serial** processing



GPU: **thousands** of smaller (⇏ more efficient) cores designed for handling **multiple tasks simultaneously**

⋆ Host (CPU) – Device (GPU) interaction, executing many kernels (device functions) in parallel



Platform & programming model for this project:
CUDA (Compute Unified Device Architecture) of NVIDIA

C vs CUDA: `Hello world!` example

```c
#include <stdio.h>
#define N 7

int main() {

    char a[N] = "Hello ";

    int b[N] = {15, 10, 6, 0, -11, 1, 0};

    printf("%s", a);

    // adding int to char
    int i;
    for (i=0; i<N; i++)
        a[i] += b[i];

    printf("%s\n", a);

    return 0;

    // in ASCII
    // H 72, e 101, l 108, o 111
    // W 87, r 114, d 100, ! 33
}
```

```c
#include <stdio.h>
#define N 7

__global__ void add_arrays(char *a, int *b)  {
    a[threadIdx.x] += b[threadIdx.x];
}

int main() {

    char a[N] = "Hello ";
    int b[N] = {15, 10, 6, 0, -11, 1,0};

    char *ad; int *bd;
    const int csize = N*sizeof(char);
    const int isize = N*sizeof(int);

    printf("%s", a);

    cudaMalloc((void**)&ad, csize);
    cudaMalloc((void**)&bd, isize);

    cudaMemcpy(ad, a, csize, cudaMemcpyHostToDevice);
    cudaMemcpy(bd, b, isize, cudaMemcpyHostToDevice);

    dim3 dimBlock(N); dim3 dimGrid (1);
    // adding int to char
    add_arrays<<<dimGrid, dimBlock>>>(ad, bd);

    cudaMemcpy(a, ad, csize, cudaMemcpyDeviceToHost);
    cudaFree(ad);

    printf("%s\n", a);
    return EXIT_SUCCESS;
}
```

Calculation of the F-statistic

To estimate how well the model matches with the data $x(t)$, we calculate $\mathcal{F}$,

$$\mathcal{F} = \frac{2}{S_0 T_0} \left(\frac{|F_a|^2}{\langle a^2 \rangle} + \frac{|F_b|^2}{\langle b^2 \rangle} \right)$$

where S_0 is the spectral density, T_0 is the observation time, and

$$F_a = \int_0^{T_0} x(t) a(t) \exp(-i\phi(t)) dt, F_b = \dots$$

and $a(t)$, $b(t)$ are amplitude modulation functions (depend on the detector location and sky position of the source),

$$h_1(t) = a(t) \cos\phi(t), \quad h_2(t) = b(t) \cos\phi(t),$$

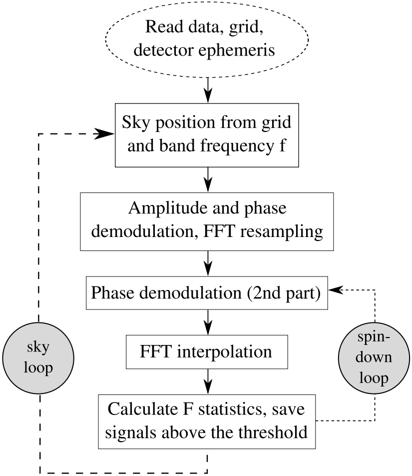
$$h_3(t) = a(t) \sin\phi(t), \quad h_4(t) = b(t) \sin\phi(t),$$

related to the model of the signal (h_i, $i = 1, \dots, 4$)

$$h(t) = \sum_{i=1}^{4} A_i \, h_i(t).$$

For triaxial ellipsoid model: dependence on extrinsic $(h_0, \psi, \iota, \phi_0)$ and intrinsic $(f, \dot{f}, \alpha, \delta)$ parameters.

F-stat all-sky search description



Main parameters in coherent search for continuous wave signals:

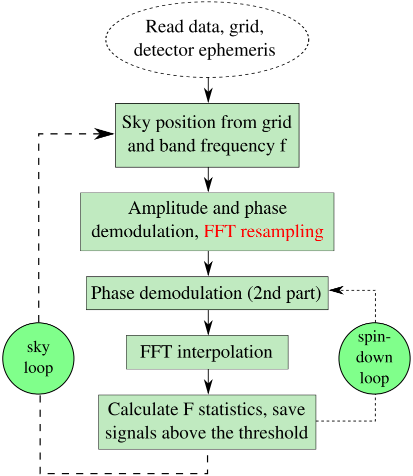
- ⋆ bandwidth 1Hz
- ⋆ sampling time 0.5 s
- ⋆ data length N = 344656 (two sideral days)

⋆ 4D grid: α, δ, f, $\dot{f}$ - sky positions, frequency and spindown

⋆ Uses the F-statistic defined in Jaranowski, Królak & Schutz (1998), algorithm described and tested in Astone et al. (2010)

⋆ No. of F-statistic evaluations $\propto f^3$ (no. of sky positions $\propto f^2$, spindown $\propto f$)

F-stat all-sky search description



Basically the whole loop over sky ($\alpha, \ \delta$) can be computed in parallel since the sky positions are independent of each other

The majority of computing is spent on

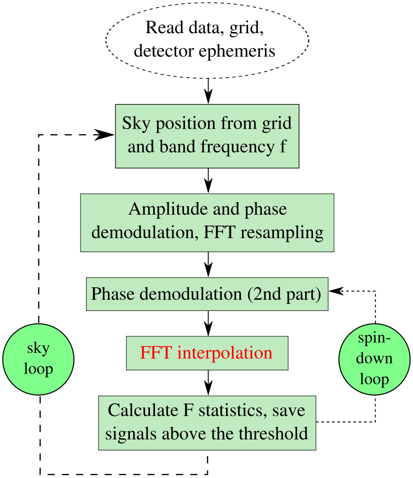
- ⋆ calculating the phase (trigonometric functions, $\gtrapprox$ 30%)
- ⋆ FFT ($\gtrapprox$ 50%)

Efficient FFT requires 2^N data points ($N_{data} = 344656 < 2^{19}$) $\rightarrow$ padding with zeros to $N = 2^{19}$

FFT: resampling

- ⋆ Resampling to barycentric time - FFT and inverse:
  - ⋆ nearest-neighbour ($\simeq$ 5% error),
  - ⋆ splines ($\simeq$ 0.1% error)

F-stat all-sky search description



The majority of computing is spent on

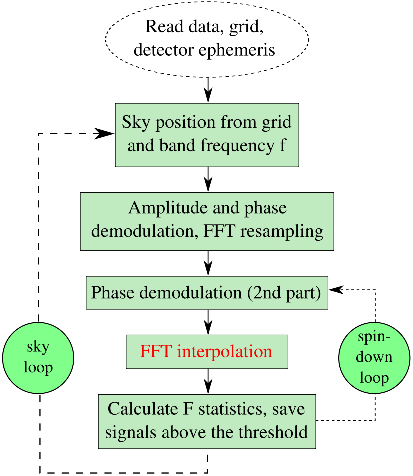
- ⋆ calculating the phase (trigonometric functions, $\gtrapprox$ 30%)
- ⋆ FFT ($\gtrapprox$ 50%)

Efficient FFT requires 2^N data points ($N_{data} = 344656 < 2^{19}$) $\rightarrow$ padding with zeros to $N = 2^{19}$

FFT: Interpolation

Grid coincides with Fourier frequencies - possible loss of signal (max. 36.3% when f is half way between the Fourier frequencies)

- ⋆ FFT (length N) & interbinning (max. $\simeq$ 13% error): DFT component in the middle of two Fourier frequencies approximated by $X((k+1/2) \simeq (X(k+1) - X(k))/\sqrt{2}$
- ⋆ FFT zero-padding (length $2N$, max. $\simeq$ 10% error)



How to do FFT with GPU:

- ⋆ use CUDA cuFFT library:
  - ☺ well-optimized (Cooley-Tukey, Bluestein), 1D/2D/3D double precision complex/real transforms, multiple transforms, in- and out-of-place transforms,
  - ☹ cannot launch many instances at the same time (at least not with every card/CUDA version).
- ⋆ write custom kernel for FFT, launch concurrently.

⋆ cuSPARSE (sparse matrix routines)

Results of implementation on GPUs

- ⋆ Input data loaded to device once,
  - ⋆ One detector version, but easy to generalize (CPU network-of-detectors version exists),
- ⋆ Sequence of kernels launched in a loop from CPU,
- ⋆ Time resampling done using double precision, everything else (main spindown loop) using single precision,
- ⋆ Asynchronous output transfer to host.

Current GPU results: $\sim \times 50$ speedup **with respect to the optimized CPU code**

Estimated time τ to match one template:

- ⋆ CPU (Intel(R) Xeon(R) CPU E5-2665 @ 2.40GHz) $\simeq 4 \times 10^{-2}$ s
- ⋆ GPU (GeForce GTX Titan) $\simeq 8 \times 10^{-4}$ s

Also testing on:

- ⋆ Intel(R) Core(TM) i5, 2.8GHz
- ⋆ GPUs:
  - ⋆ GeForce GTX 560 Ti
  - ⋆ GeForce GTX 480

Performance scaling - favorably for high frequencies (fast spindown loop on GPU).

Profiling the CPU version with `perf`

Initially we were using `gprof` and `Callgrind/KCachegrind`, but later learned about `perf` (of `linux-tools`) and found it much more useful to estimate performance in FLOPS:

- `perf stat -e r5300c0 -e r530110 -e r532010 -e r534010 -e r538010 -e r531010 -e r530111 -e r530211`, where the switches correspond to different operations on a Sandy Bridge processor:
  - r530111 SIMD_FP_256:PACKED_SINGLE
  - r530211 SIMD_FP_256:PACKED_DOUBLE
  - r530110 X87
  - r531010 SSE_FP_PACKED_DOUBL
  - r532010 SSE_FP_SCALAR_SINGLE
  - r534010 SSE_PACKED_SINGLE
  - r538010 SSE_SCALAR_DOUBLE

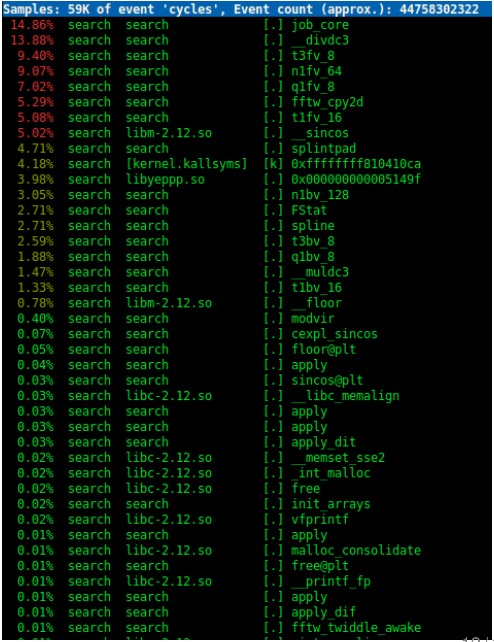
(SIMD - Single Instruction Multiple Data, SSE - Streaming SIMD Extensions)

Estimated performance is 25% of peak performance on Sandy Bridge

Profiling the CPU version with `perf`

Also useful to locate the time-expensive parts of the code (with a direct view into the assembly code):

- ⋆ `perf record -B -e task-clock:u,cycles:u, instructions:u`
- ⋆ `perf report`



```
Samples: 59K of event 'cycles', Event count (approx.): 44758302322
 14.86%  search  search              [.] job_core
 13.88%  search  search              [.] __divdc3
  9.40%  search  search              [.] t3fv_8
  9.07%  search  search              [.] n1fv_64
  7.02%  search  search              [.] q1fv_8
  5.29%  search  search              [.] fftw_cpy2d
  5.08%  search  search              [.] t1fv_16
  5.02%  search  libm-2.12.so        [.] __sincos
  4.71%  search  search              [.] splintpad
  4.18%  search  [kernel.kallsyms]   [k] 0xffffffff810410ca
  3.98%  search  libyeppp.so         [.] 0x000000000005149f
  3.05%  search  search              [.] n1bv_128
  2.71%  search  search              [.] FStat
  2.71%  search  search              [.] spline
  2.59%  search  search              [.] t3bv_8
  1.88%  search  search              [.] q1bv_8
  1.47%  search  search              [.] __muldc3
  1.33%  search  search              [.] t1bv_16
  0.78%  search  libm-2.12.so        [.] __floor
  0.40%  search  search              [.] modvir
  0.07%  search  search              [.] cexpl_sincos
  0.05%  search  search              [.] floor@plt
  0.04%  search  search              [.] apply
  0.03%  search  search              [.] sincos@plt
  0.03%  search  libc-2.12.so        [.] __libc_memalign
  0.03%  search  search              [.] apply
  0.03%  search  search              [.] apply
  0.03%  search  search              [.] apply_dit
  0.02%  search  libc-2.12.so        [.] __memset_sse2
  0.02%  search  libc-2.12.so        [.] _int_malloc
  0.02%  search  libc-2.12.so        [.] free
  0.02%  search  search              [.] init_arrays
  0.02%  search  libc-2.12.so        [.] vfprintf
  0.01%  search  search              [.] apply
  0.01%  search  libc-2.12.so        [.] malloc_consolidate
  0.01%  search  search              [.] free@plt
  0.01%  search  libc-2.12.so        [.] __printf_fp
  0.01%  search  search              [.] apply
  0.01%  search  search              [.] apply_dif
  0.01%  search  search              [.] fftw_twiddle_awake
```

Fast libraries for commonly used functions in CPU version

- ⋆ Obvious choice is `icc` Intel compiler + Math Kernel Library (MKL), with optimizing flags
  `-march=native -mtune=native -Ofast -unroll-agressive -ipo -use-intel-optimized-headers -opt-prefetch`

- ⋆ We also have a good experience with `gcc`, `FFTW3` and optimized math libraries (using latest SSE & AVX instructions):
  - ⋆ SLEEF (SIMD Library for Evaluating Elementary Functions) - trigonometric functions (among others) in double precision without table look-ups, conditional branches etc. `http://shibatch.sourceforge.net`
    or
  - ⋆ YEPPP - high-performance SIMD-optimized mathematical library for x86, ARM, and MIPS processors. `http://www.yeppp.info`
- ⋆ `FFTW3` Planner Flags - `FFTW_PATIENT` instead of `FFTW_MEASURE`
- ⋆ compiler flags: `-O3 -ffast-math -funsafe-loop-optimizations -funroll-loops -march=native -mtune=native -mavx`

Changing the libraries from standard math to optimized ones + remembering about FFTW3 planner flags - $>$ 30% speedup in case of CPU.

Summary/references

We have a quite well-optimized CPU code ($\pm$ memory access optimizations), and a working GPU code that may still need some optimization (+ extenstion to a network of detectors).

- P. Astone, K. M. Borkowski, P. Jaranowski, M. Piętka and A. Królak, PRD, **82**, 022005 (2010)
- `https://developer.nvidia.com/cuFFT`
- P. Jaranowski, A. Królak, and B. F. Schutz, PRD **58**, 063001 (1998).
- https://github.com/mbejger/polgraw-allsky.git