

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).

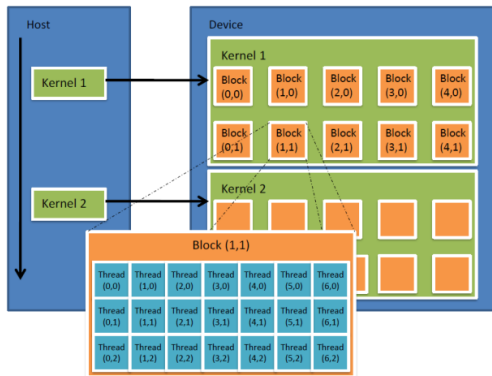
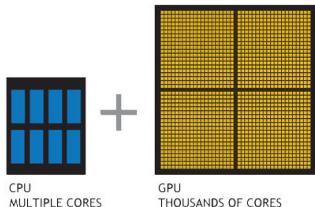
- ★ CPU vs GPU concept,
- ★ description of the all-sky F-stat search for candidate signals,
- ★ Implementation of the GPU version,
- ★ Performance testing

Central Processing Units vs Graphics Processing Units

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

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

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



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

C vs CUDA: Hello world! example

```
1  #include <stdio.h>
2  #define N 7
3
4  int main() {
5
6      char a[N] = "Hello ";
7
8      int b[N] = {15, 10, 6, 0, -11, 1, 0};
9
10     printf("%s", a);
11
12     // adding int to char
13     int i;
14     for (i=0; i<N; i++)
15         a[i] += b[i];
16
17     printf("%s\n", a);
18
19     return 0;
20
21     // in ASCII
22     // H 72, e 101, l 108, o 111
23     // W 87, r 114, d 100, ! 33
24 }
```

```
1  #include <stdio.h>
2  #define N 7
3
4  __global__ void add_arrays(char *a, int *b) {
5      a[threadIdx.x] += b[threadIdx.x];
6  }
7
8  int main() {
9
10     char a[N] = "Hello ";
11     int b[N] = {15, 10, 6, 0, -11, 1, 0};
12
13     char *ad; int *bd;
14     const int csize = N*sizeof(char);
15     const int isize = N*sizeof(int);
16
17     printf("%s", a);
18
19     cudaMalloc((void*)&ad, csize);
20     cudaMalloc((void*)&bd, isize);
21
22     cudaMemcpy(ad, a, csize, cudaMemcpyHostToDevice);
23     cudaMemcpy(bd, b, isize, cudaMemcpyHostToDevice);
24
25     dim3 dimBlock(N); dim3 dimGrid (1);
26     // adding int to char
27     add_arrays<<<dimGrid, dimBlock>>>(ad, bd);
28
29     cudaMemcpy(a, ad, csize, cudaMemcpyDeviceToHost);
30     cudaFree(ad);
31
32     printf("%s\n", a);
33     return EXIT_SUCCESS;
34 }
```

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),

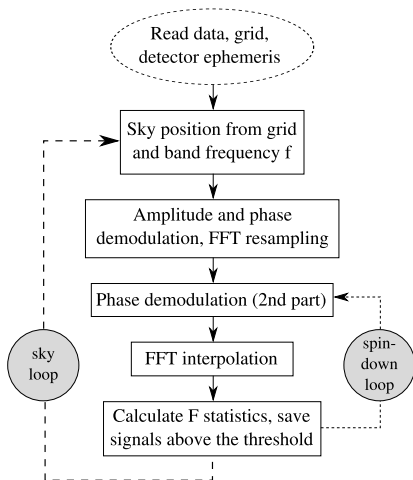
$$h_1(t) = a(t) \cos \phi(t), \quad h_2(t) = b(t) \cos \phi(t),$$

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

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

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

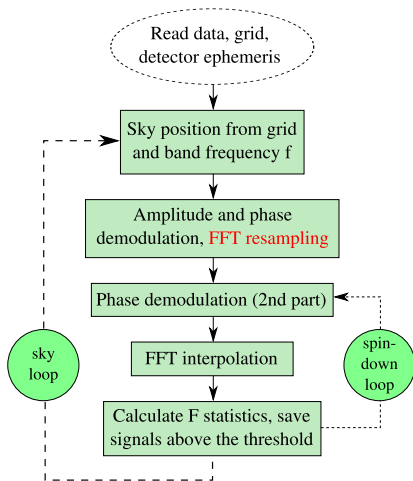
F-stat all-sky search description



Coherent search for CW signal; main parameters:

- ★ bandwidth 1Hz
 - ★ sampling time 0.5 s
 - ★ data length $N = 344656$ (two sidereal days)
- ★ 4D grid: $\alpha, \delta, f, \dot{f}$ - sky positions, frequency and spindown
- ★ Uses the F-statistic defined in [Jaranowski, Królak & Schutz \(1998\)](#), serial code 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 (α , δ) can be computed in parallel since the sky positions are independent of each other

The majority of computing is spent on

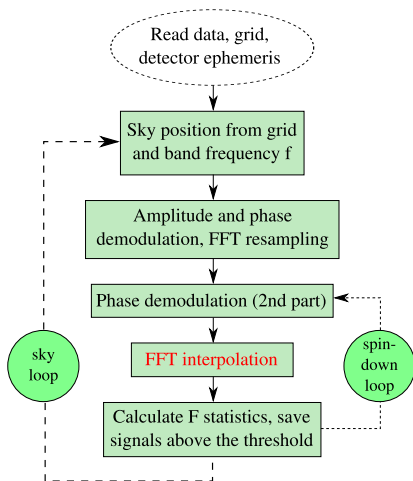
- ★ calculating the phase (trigonometric functions, $\approx 25\%$)
- ★ FFT ($\approx 60\%$)

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 ($\approx 5\%$ error),
 - ★ splines ($\approx 0.1\%$ error)

F-stat all-sky search description



The majority of computing is spent on

- ★ calculating the phase (trigonometric functions, $\approx 25\%$)
- ★ FFT ($\approx 60\%$)

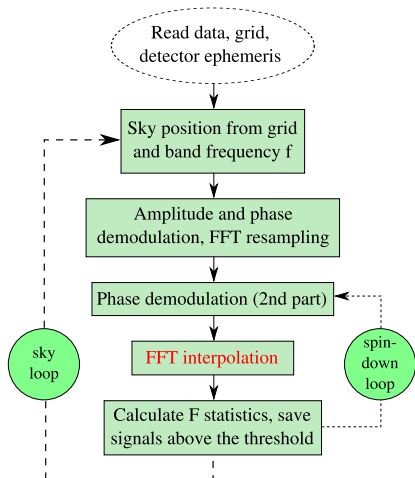
Efficient FFT requires 2^N data points
($N_{data} = 344656 < 2^{19}$) \rightarrow padding with zeros to $N = 2^{19}$

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

F-stat: parallelization strategy



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: ~ 50 speedup **with respect to the optimized CPU code**

The test case $f_{GW} = 1300$ Hz - 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).

Fast libraries for commonly used functions

- ★ Obvious choice is `icc` Intel compiler + Math Kernel Library (MKL).
- ★ We had 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.

References

- ▶ 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).