Description of Polgraw All-Sky Search GPGPU program

Jan Bolek

Contents

1 Introduction **2**

1.1 The algorithm 2

2 Code explanation **3**

2.1 Data structures 3

2.2 "main" function 3

2.3 Initialization 4

2.4 Main job 4

2.5 Sky loop internals 5

2.5.1 Preparing variables 5

2.5.2 Resampling and spline interpolation to baricentric time 6

2.6 Spin-down loop internals 8

2.7 Finding and saving candidates data 10

3 References **12**

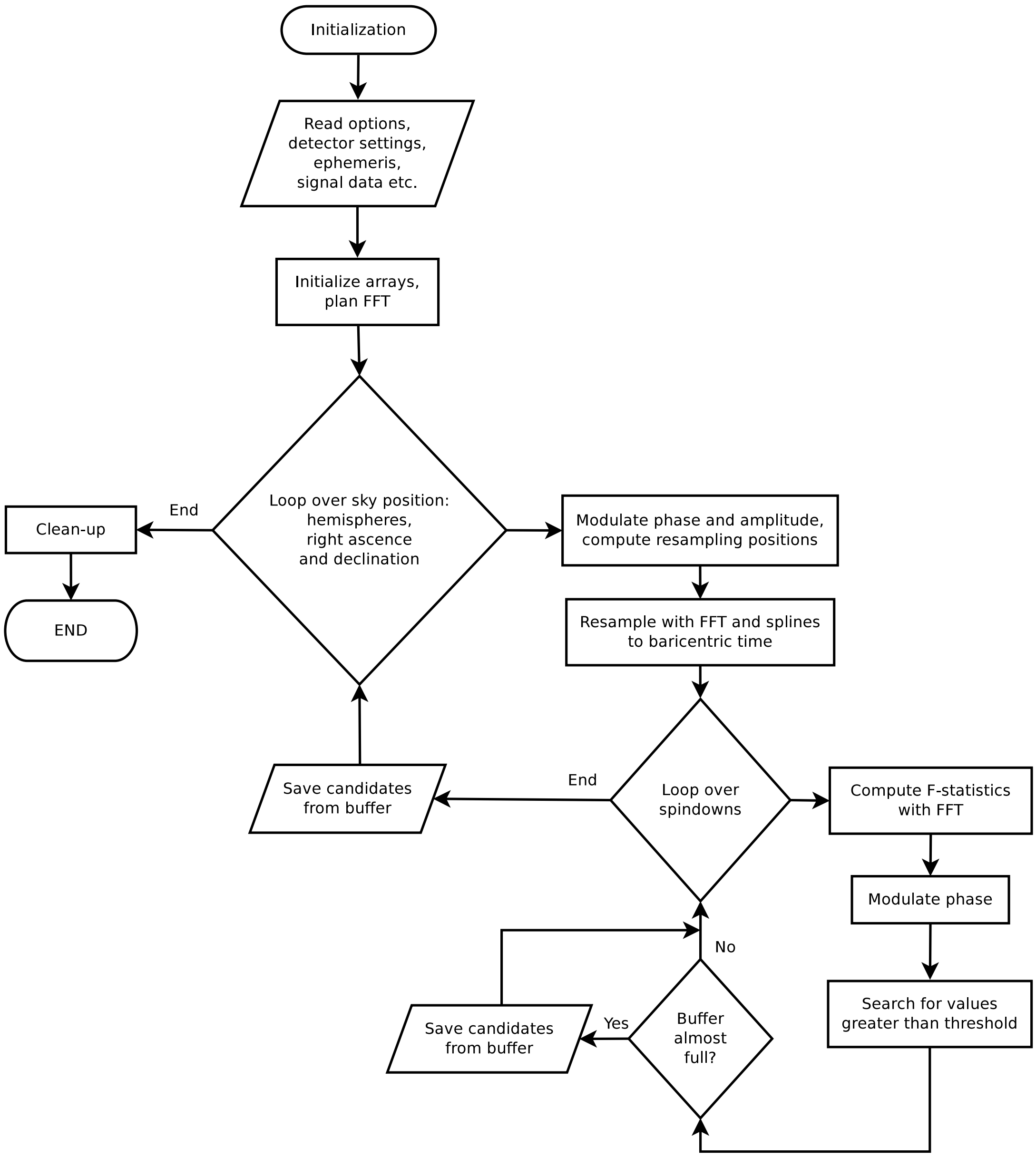
1 Introduction

The purpose of this article is to introduce one into structure, working principle and techniques used in gravitational waves (abbr. GW) All-Sky Search program.

The main purpose of described program is to find candidate GW signals parameters using $\mathcal{F}$-statistic. The parameters consist of frequency, spin-down (first derivative of frequency), positions on sky (e.g. right ascension and declination) and signal-to-noise ratio (SNR). Further theoretical explanation can be found in [1].

1.1 The algorithm

Figure 1: Flow-chart of the algorithm



The searching algorithm consists of iterating over sky positions (i.e. hemispheres, right ascension and declination transformed into integral-valued grid), frequency and one spindown parameter in order to modulate signal with appropriate function using this parameters and evaluate so-called $\mathcal{F}$*-statistic*. The trick is that $\mathcal{F}$-statistic may be computed using Fast Fourier Transform (FFT), thus iterating over frequency loop can be avoided, which results in great performance improvement.

In order to apply above method detector signal must be defined over baricentric time (i.e. time with respect to solar system baricenter), so there must be performed interpolation to precomputed baricentric time positions. Those positions are known from detector's and Earth's ephemeris.

Because of usage of large vectors ($\sim 2^{19}$ elements) and Fourier transform the program was written for GPUs using CUDA architecture. Thanks to use of massive parallelization there was achieved speed-up over CPU version of about 50-100 times, depending on GPU type.

2 Code explanation

2.1 Data structures

For convenience arrays and other variables, e.g. settings and detector constants are stored in structures defined in file `struct.h`. Basic structures are:

- `struct _comm_line_opts` (typedef `Command_line_opts`)
  options read from command-line
- `struct _detector_settings` (typedef `Detector_settings`)
  parameters and settings for detector and for computations (array lengths etc.)
- `struct _search_range` (typedef `Search_range`)
  range of search
- `struct _fft_plans` (typedef `FFT_plans`)
  plans (handles) for cuFFT
- `struct _ampl_mod_coeff` (typedef `Ampl_mod_coeff`)
  amplitude modulation function coefficients

2.2 "main" function

`main()` function consists of other functions calls – mostly initialization. The most important part is:

Listing 1: main.c

```
/*
############ Main job ################
```

```
*/

search(
        &sett,
        &opts,
        &s_range,
        &arr,
        &fft_plans,
        &amod,
        &Fnum,
        cu_F
       );
```

i.e. the worker function call.

2.3 Initialization

Initialization consists of the following steps:

1. handling command line options (`handle_opts()`)
2. checking if output directory exists (and creating it if needed)
3. reading grid generating matrix (`read_grid()`)
4. reading detector settings and physical constants (`settings()` from `settings.c`)
5. initializing amplitude modulation function coefficients for Virgo detector (`rogcvir()`)
6. allocating arrays and reading input arrays (`init_arrays()`)
7. determining searching range (`set_search_range()`)
8. planning FFT (`plan_fft()`)
9. reading checkpoints (for resuming execution at given point) (`read_checkpoints()`)

All those functions, except for `settings()`, are defined in `init.c`.

2.4 Main job

Function `search()` is defined in file `jobcore.cu`. It contains main loop over sky positions and jobcore() function call.

jobcore() function performs computations outside of spin-down loop and in the spin-down loop itself.

2.5 Sky loop internals

2.5.1 Preparing variables

First, we compute real angular coordinates from linear grid coordinates:

Listing 2: jobcore.cu

```
    //change linear (grid) coordinates to real coordinates
    lin2ast (al1/sett->oms, al2/sett->oms, pm, sett->sepsm, sett->cepsm,
             &sinalt, &cosalt, &sindelt, &cosdelt);

    // calculate declination and right ascencion
    // which will be written in file as candidate signal sky positions
    sgnlt[2] = asin (sindelt);
    sgnlt[3] = fmod (atan2 (sinalt, cosalt)+2.*M_PI, 2.*M_PI);
```

Then we compute amplitude modulation functions $a(t)$ and $b(t)$ (`arr->cu_aa` and `arr->cu_bb`, respectively) using GPU. In this step we need to normalize those function by square root of their sum of squares (i.e. $a_i = a'_i/\sqrt{\sum_t a'^2_i}$), which is performed using parallel reduction in kernels `reduction_sumsq()` and `reduction_sum()`.

Listing 3: jobcore.cu

```
    // amplitude modulation function
    modvir_gpu(sinalt, cosalt, sindelt, cosdelt, sett->sphir,
               sett->cphir, arr->cu_aa, arr->cu_bb, sett->N, arr);
```

Then kernel `shift_time_mod()` computes shifted (baricentric) time, modulates signal with $a(t)$, $b(t)$ and phase factor (thus obtaining two vectors, `xa` and `xb`), computes shifted time position `tshift[i]`) for spline interpolation and pads zeros to nearest power of two (`nfft`).

Listing 4: jobcore.cu

```
    //calculate detector positions with respect to baricenter
    nSource[0] = cosalt*cosdelt;
    nSource[1] = sinalt*cosdelt;
    nSource[2] = sindelt;

    shft1 = 0.;
    for (j=0; j<3; j++)
       shft1 += nSource[j]*arr->DetSSB[j];
    het0 = fmod (nn*sett->M[8]+mm*sett->M[12], sett->M[0]);

    //compute shifted time, modulate phase and pad zeros to size 'nfft'
    shift_time_mod<<<(sett->nfft + BLOCK_SIZE - 1)/BLOCK_SIZE, BLOCK_SIZE
       >>>(shft1, het0, nSource[0], nSource[1], nSource[2],
          arr->cu_xDat, arr->cu_xa, arr->cu_xb, arr->cu_shft,
          arr->cu_shftf, arr->cu_tshift, arr->cu_aa, arr->cu_bb, arr->
             cu_DetSSB,
          sett->oms, sett->N, sett->nfft, sett->interpftpad);
```

Listing 5: kernels.cu

```
__global__ void shift_time_mod(...) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if (i < N) {
        double S = ns0 * DetSSB[i*3+0] + ns1 * DetSSB[i*3+1] + ns2 *
            DetSSB[i*3+2];
        shft[i] = S;
        shftf[i]= S - shft1;

        /* phase mod */
        double phase = -het0*i - oms * S;
        double c = cos(phase), s = sin(phase);
        xa[i].x = xDat[i] * aa[i] * c;
        xa[i].y = xDat[i] * aa[i] * s;
        xb[i].x = xDat[i] * bb[i] * c;
        xb[i].y = xDat[i] * bb[i] * s;

        //calculate time positions for spline interpolation
        tshift[i] = interpftpad * ( i - shftf[i] );
    } else if (i < nfft) {
        xa[i].x = xa[i].y = xb[i].x = xb[i].y = 0;
    }
}
```

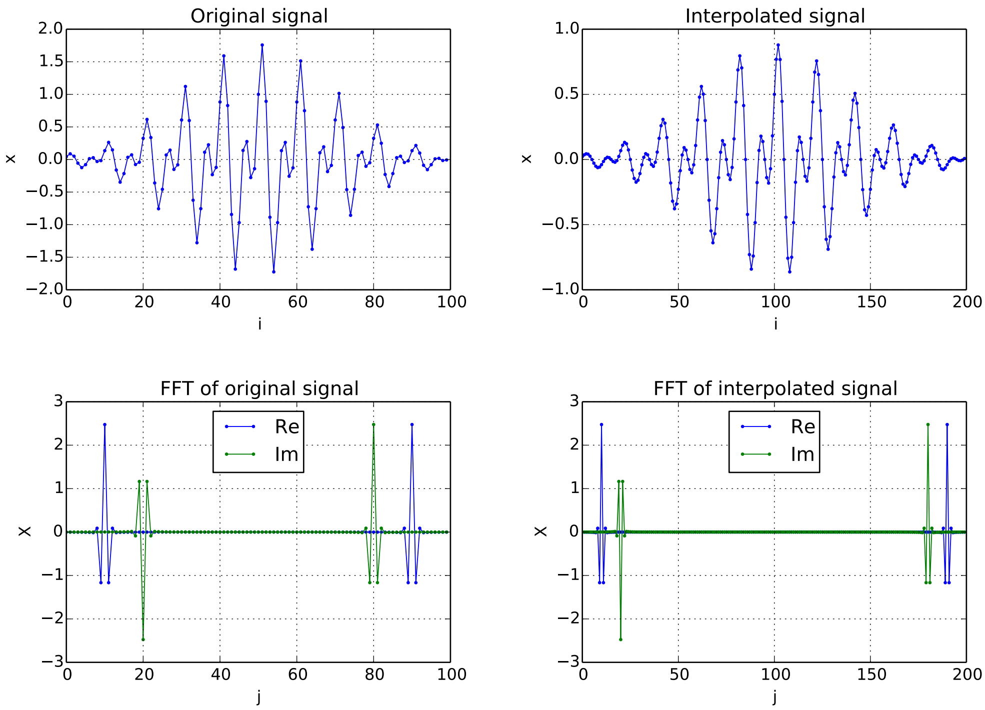
2.5.2 Resampling and spline interpolation to baricentric time

First we compute forward FFT of length `nfft` (with zeros padded from original length `N`). Then we pad Fourier transform with zeros to size 2*`nfft` (remembering that in unshifted FFT positive frequencies are placed from 0th bin to n/2-1, and negative are placed from n/2 to n-1, so we basically insert zeros "in-between") and take Fourier transform back to obtain signal samples at twice the original sampling frequency (see figure 2.) Then the vector is scaled because of property of FFT (i.e. elements after two FFTs are multiplied by size of vector).

Listing 6: jobcore.cu

```
    int nyqst = (sett->nfft+2)/2; // Nyquist frequency

    //fft
    cufftExecZ2Z(plans->pl_int, arr->cu_xa, arr->cu_xa, CUFFT_FORWARD);
    cufftExecZ2Z(plans->pl_int, arr->cu_xb, arr->cu_xb, CUFFT_FORWARD);

    //shift frequencies and remove those over Nyquist
    resample_postfft<<<(sett->Ninterp + BLOCK_SIZE - 1)/BLOCK_SIZE,
        BLOCK_SIZE>>>(arr->cu_xa, arr->cu_xb, sett->nfft, sett->Ninterp,
        nyqst);
    CudaCheckError();

    //fft back
    cufftExecZ2Z(plans->pl_inv, arr->cu_xa, arr->cu_xa, CUFFT_INVERSE);
    cufftExecZ2Z(plans->pl_inv, arr->cu_xb, arr->cu_xb, CUFFT_INVERSE);

    //scale fft
    ft = (double)sett->interpftpad / sett->Ninterp; //scale FFT
    scale_fft<<<(sett->Ninterp + BLOCK_SIZE - 1)/BLOCK_SIZE, BLOCK_SIZE
        >>>(arr->cu_xa, arr->cu_xb, ft, sett->Ninterp);
    CudaCheckError();
```

Figure 2: FFT interpolation example



Next step is spline interpolation in which we use cubic splines with natural boundary conditions, meaning that second derivatives at boundaries (x_0 and x_n) are set to zero. In our case we also have other assumption: $h = x_{i+1} - x_i = 1$.

Spline coefficients are computed by solving linear equation:

$$\begin{bmatrix} 4 & 1 & & & & \\ 1 & 4 & 1 & & & \\ & 1 & \ddots & \ddots & & \\ & & \ddots & \ddots & 1 & \\ & & & 1 & 4 & 1 \\ & & & & 1 & 4 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_{n-3} \\ z_{n-2} \\ z_{n-1} \end{bmatrix} = \begin{bmatrix} 6(b_1 - b_0) \\ 6(b_2 - b_1) \\ 6(b_3 - b_2) \\ \vdots \\ 6(b_{n-3} - b_{n-4}) \\ 6(b_{n-2} - b_{n-3}) \\ 6(b_{n-1} - b_{n-2}) \end{bmatrix}$$

where:

- n is number of intervals in interpolated function, so $n + 1$ is number of given points (knots)
- z_i are unknown coefficients (second derivatives at knots, to be precise),
- $n - 2$ is rank of set of equations (we know from natural boundary conditions that $z_0 = z_n = 0$)
- b_i are differences of function values in consecutive knots: $b_i = (y_{i+1} - y_i)$

Matrix of above shape is called *tridiagonal matrix*. We solve this set of equations using `cusparseZgtsv` function included in the `cuSPARSE` package.

Having solved above equations for z_i, we can compute interpolated value $y'(x)$ using following formula:

$$y(x) = S_i(x) = \frac{z_{i+1}}{6}(x-x_i)^3 + \frac{z_i}{6}(x_{i+1}-x)^3 + \left(y_{i+1} - \frac{z_{i+1}}{6}\right)(x-x_i) + \left(y_i - \frac{z_i}{6}\right)(x_{i+1}-x)$$

where i is number of spline interval S_i which contains point x: $i = \lfloor x \rfloor$.

Function `gpu_interp()` encapsulates above algorithm. It is defined in file `spline_z.cu`.

Listing 7: jobcore.cu

```
    //interpolation of xa
    gpu_interp(arr->cu_xa,      //input data
               sett->Ninterp,   //input data length
               arr->cu_tshift,  //output time domain
               arr->cu_xar,     //output values
               sett->N,         //output data length
               arr->cu_d,       //diagonal
               arr->cu_dl,      //upper diagonal
               arr->cu_du,      //lower diagonal
               arr->cu_B);      //coefficient matrix

    //interpolation of xb
    gpu_interp(arr->cu_xb,      //input data
               sett->Ninterp,   //input data length
               arr->cu_tshift,  //output time domain
               arr->cu_xbr,     //output values
               sett->N,         //output data length
               arr->cu_d,       //diagonal
               arr->cu_dl,      //upper diagonal
               arr->cu_du,      //lower diagonal
               arr->cu_B);      //coefficient matrix
```

2.6 Spin-down loop internals

In spin-down loop signal resampled in last step is phase-modulated by factor dependent on current spin-down parameter and shifted baricentric time.

Listing 8: jobcore.cu

```
        phase_mod_2<<<(sett->N + BLOCK_SIZE - 1)/BLOCK_SIZE, BLOCK_SIZE>>>(
            arr->cu_xa_f, arr->cu_xb_f,
            arr->cu_xar_f, arr->cu_xbr_f,
            het1, sgnlt[1], arr->cu_shft,
            sett->N);
```

Next step is the core of whole algorithm: we compute two integrals used in evaluating $\mathcal{F}$-statistic and at the same time perform interpolation to get twice (or more) denser sampling rate. Interpolation can be done with two methods:

1. *interbinning*, which consists in computing values in between the bins by approximation using following formula[1]:
   $x_{i+1/2} = (x_i - x_{i+1})/\sqrt{2}$

2. *FFT interpolation*, i.e. padding with zeros to size `fftpad`-times larger than original array. This method requires computing twice as long FFT.

Listing 9: jobcore.cu

```
        if (opts->fftinterp == INT) { //interpolation by interbinning
          //pad zeros from N to nfft
          pad_zeros<<<(sett->nfft - sett->N + BLOCK_SIZE - 1)/
             BLOCK_SIZE, BLOCK_SIZE>>>(
             arr->cu_xa_f+sett->N, arr->cu_xb_f+sett->N, sett->nfft -
                sett->N);
          CudaCheckError();

          // FFT length nfft
          CUFFT_EXEC_FFT(plans->plan, arr->cu_xa_f, arr->cu_xa_f,
             CUFFT_FORWARD);
          CUFFT_EXEC_FFT(plans->plan, arr->cu_xb_f, arr->cu_xb_f,
             CUFFT_FORWARD);

          // make a gap between every bin
          interbinning_gap<<<(sett->nfft/2 + BLOCK_SIZE - 1)/
             BLOCK_SIZE, BLOCK_SIZE>>>(
             arr->cu_xa_f, arr->cu_xb_f, arr->cu_xa2_f, arr->cu_xb2_f,
                 sett->nfft);
          CudaCheckError();

          //interpolate values between every bin
          interbinning_interp<<<( (sett->nfft-2)/2 + BLOCK_SIZE - 1)/
             BLOCK_SIZE, BLOCK_SIZE>>>(
             arr->cu_xa2_f, arr->cu_xb2_f, sett->nfft);
          CudaCheckError();

          cu_xa_final = arr->cu_xa2_f;
          cu_xb_final = arr->cu_xb2_f;

        } else { //interpolation by FFT (fftpad-times-longer FFT)
          //pad zeros from N to nfft*fftpad
          pad_zeros<<<(sett->nfft*sett->fftpad - sett->N + BLOCK_SIZE
             - 1)/BLOCK_SIZE, BLOCK_SIZE>>>(
             arr->cu_xa_f+sett->N, arr->cu_xb_f+sett->N, sett->nfft*
                sett->fftpad - sett->N);
          CudaCheckError();

          //fft length fftpad*nfft
          CUFFT_EXEC_FFT(plans->plan, arr->cu_xa_f, arr->cu_xa_f,
             CUFFT_FORWARD);
          CUFFT_EXEC_FFT(plans->plan, arr->cu_xb_f, arr->cu_xb_f,
             CUFFT_FORWARD);

          cu_xa_final = arr->cu_xa_f;
          cu_xb_final = arr->cu_xb_f;
        }
```

[1]I'm not sure about the sign, but later we use moduli of these values, so it's irrelevant.

Finally, having computed F_a and F_b integrals (as defined in [1], equation (5.5)) we can compute $\mathcal{F}$-statistic ([1], eq. (3.8)) and normalize it.

Listing 10: jobcore.cu

```
            compute_Fstat<<<(sett->nmax-sett->nmin + BLOCK_SIZE - 1)/BLOCK_SIZE, BLOCK_SIZE>>>(
                                   cu_xa_final + sett->nmin,
                                   cu_xb_final + sett->nmin,
                                   cu_F + sett->nmin,
                                   sett->crf0,
                                   sett->nmax - sett->nmin);
```

With assumption that noise present in signal is white noise, normalization can be done by simply dividing $\mathcal{F}$-statistic by signal's variance. Without that assumption $\mathcal{F}$-statistic is divided into blocks (size, for example, 4096 elements) and normalized by it's sum in every block[2]:

Listing 11: jobcore.cu

```
            if (sett->sig2 < 0.0) { // when noise is not white-noise
               FStat_gpu(cu_F+sett->nmin, sett->nmax - sett->nmin, NAV, arr->cu_mu, arr->cu_mu_t);
            } else { // when noise is white-noise
               kernel_norm_Fstat_wn<<<(sett->nmax - sett->nmin + BLOCK_SIZE - 1)/BLOCK_SIZE, BLOCK_SIZE>>>(
                                   cu_F + sett->nmin,
                                   sett->sig2,
                                   sett->nmax - sett->nmin);
            }
```

2.7 Finding and saving candidates data

Next step consist in finding $\mathcal{F}$-statistic values that exceed given threshold and save found parameters in output file.

Searching is performed in parallel: every thread checks if value lying under its index is greater than neighboring values. If so it increments buffer counter and saves parameters to local buffer (i.e. spin-down-loop-internal buffer).

Listing 12: jobcore.cu

```
            //zero candidates count
            cudaMemset(arr->cu_cand_count, 0, sizeof(int));

            /* FINDING AND SAVING CANDIDATES */
            find_candidates<<<(sett->nmax - sett->nmin + BLOCK_SIZE - 1)/BLOCK_SIZE, BLOCK_SIZE>>>(
               cu_F,                   // F-statistic array
               arr->cu_cand_params,    // local parameters buffer
               arr->cu_cand_count,     // local found candidates count
               opts->trl,              // threshold value
               sett->nmin,             // minimum F-statistic array index
```

[2] Important: 'block' length `NAV` described in this paragraph isn't the same as 'block' on the GPU grid length `BLOCK_SIZE`.

```
            sett->nmax,          // maximum -||-
            sett->fftpad,        // FFT padding for calculation of freq.
            sett->nfft,          // FFT length
            sgnl0,               // base frequency parameter
            sett->nd,            // number of degrees of freedom
            sgnlt[1],            // spindown
            sgnlt[2],            // sky\
            sgnlt[3]);           // positions
```

Listing 13: kernels.cu

```
// This may be confusing, but three "if"s in kernel below
// are faster than one "if" with one, big, combined condition.
// Operations in every "if" are the same, so they are wrapped in macro.

// parameters are:
// [frequency, spindown, position1, position2, snr]
#define ADD_PARAMS_MACRO \
      int p = atomicAdd(found, 1); \
      params[p*NPAR + 0] = 2.0*M_PI*(idx)/((double)fftpad*nfft)+sgnl0; \
      params[p*NPAR + 1] = sgnl1; \
      params[p*NPAR + 2] = sgnl2; \
      params[p*NPAR + 3] = sgnl3; \
      params[p*NPAR + 4] = sqrt(2*(F[idx]-ndf));


__global__ void find_candidates(...) {

   int idx = blockIdx.x * blockDim.x + threadIdx.x + nmin;

   if (idx > nmin && idx < nmax && F[idx] >= val && F[idx] > F[idx+1] && F[idx] > F[idx-1]) {
      ADD_PARAMS_MACRO
   } else if (idx == nmin && F[idx] >= val && F[idx] > F[idx+1]) {
      ADD_PARAMS_MACRO
   } else if (idx == nmax-1 && F[idx] >= val && F[idx] > F[idx-1]) {
      ADD_PARAMS_MACRO
   }
}
```

Finally found candidates parameters are copied to global buffer. If size left in global buffer is less than local buffer size (i.e. next searching could cause overflow) then we save data to output file. Saving is also performed in between first and second sky positions loop as there may not be enough found candidates to flush buffer to disk.

Listing 14: jobcore.cu

```
        if (cand_count>0) { //if something was found
            printf ("\nSome signals found in %d %d %d %d\n", pm, mm, nn, ss);
            cudaMemcpy(arr->cu_cand_buffer + *cand_buffer_count * NPAR,
                       arr->cu_cand_params,
                       sizeof(FLOAT_TYPE) * cand_count*NPAR,
                       cudaMemcpyDeviceToDevice);
            *cand_buffer_count += cand_count;

            // check if it's time to save results
            // (place left in buffer is less than one-spindown buffer length)
```

```
            if (*cand_buffer_count >= arr->cand_buffer_size - arr->
               cand_params_size) {
               save_candidates(arr->cu_cand_buffer, arr->cand_buffer,
                  cand_buffer_count, outname);
            }
         }
```

3 References

[1] P. Astone, K. Borkowski, P. Jaranowski, M. Pietka, and A. Królak. “Data analysis of gravitational-wave signals from spinning neutron stars. V. A narrow-band all-sky search”. In: *Physical Review D* 82, 022005 (2010).