A constant-time complexity method for transit lightcurve detection with non-stationary noise

Jamila Taaki

January 2023

1 Introduction

Denote a vector of observations (time-series brightness observations of a star) as $\mathbf{y} \in \mathcal{R}^N$, where for a year of observations $N \sim 10^5$. Construct a noise covariance and denote the inverse of this covariance matrix as $\mathbf{K}$, a transit search space is denoted as $\mathbf{T}$ and comprises a collection of candidate transits $\mathbf{t} \in \mathcal{R}^N$. A transit search refers to computing detection test statistics for all $\mathbf{t} \in \mathbf{T}$ to find candidate transit statistics which exceed a threshold τ, a single detection test is calculated as:

$$L(\mathbf{y}) = \frac{\mathbf{y}^T\mathbf{K}\mathbf{t}}{\sqrt{\mathbf{t}^T\mathbf{K}\mathbf{t}}} \tag{1}$$

The candidate transit search space is parameterized by transit duration d ~hrs, the orbital period $P \in [1, \frac{N}{2}]$ days and the epoch/start time t_0 which for a candidate period P ranges from $t_0 \in [0, P]$. A transit event is often modeled as a fixed template parameterized by duration d, a candidate transit signal is modelled as periodic repetitions of a transit event determined by P, t_0. The use of a fixed template is justified since deviation in transit shape is small, and the error induced in transit statistic $L(\mathbf{y})$ is linearly proportional to this difference. Therefore $\mathbf{T}$ is usually gridded over the parameter ranges for P, d, t_0. In general there are ~ 10 physically plausible values for d, restricting the candidate search space to a fixed d, $\mathbf{T}_d$, the number of candidate transits is roughly upper bounded as $|\mathbf{T}_d| \approx \sum_{p\in P} p^2 \leq \frac{N^2}{4}$. Each calculation of a single transit detection test is $\mathrm{O}(N^2)$ in direct form, a transit search is therefore $\mathrm{O}(|T|N^2)$, $\sim \mathrm{O}(N^4)$. The form of search here is based on $|T|$ of order $\propto N^2$ (assuming a fixed step size in period P and epoch t_0), however by re-using partial computations achieves $\mathrm{O}(1)$ for any single transit test computation or $\mathrm{O}(N^2)$ overall.

Note: reductions can be obtained by obtaining simple forms of $\mathbf{K}$, if Toeplitz (stationary noise) the single transit detection test reduces to a Fourier-domain form, hence is $\mathrm{O}(N \log N)$. The standard box-least-squares is equivalent to assuming $\mathbf{K} = I$, and taking α the transit depth as the transit statistic $L(\mathbf{y})$. However it is desirable not to impose simplifications of this form; the rest of this document describes how to efficiently perform a transit search without any assumptions on the form of $\mathbf{K}$. This can also speed up box-least-squares (using the 1d numerator computation pattern described below).

First, the lightcurve $\mathbf{y}$ and inverse covariance $\mathbf{K}$ are fixed across the transit search, thus the vector-matrix calculation $\mathbf{y}^T\mathbf{K}$ need only be computed once per lightcurve.

Consider that for a fixed transit duration d, $\mathbf{t}$ consists of periodic repetitions of the transit template. For a pair P, t_0, this template is repeated at a set of locations given by an index vector $\mathbf{i}$. To calculate the denominator of a transit detection test for a single $\mathbf{t}$, is equivalent to the outer-product of the transit template of duration d (which I will denote as the transit kernel k_d), pointwise multiplying the covariance $\mathbf{K}$ at the indices given by the outer product of i and summing over these values:

$$\mathbf{t}^T\mathbf{K}\mathbf{t} = \sum_{i\in\mathbf{i}}\sum_{k\in\mathbf{i}} \mathbf{K}[i - \frac{d}{2} : i + \frac{d}{2}, j - \frac{d}{2} : j + \frac{d}{2}] \cdot \mathbf{k}_d \tag{2}$$

Therefore to compute transit detection tests for all $\mathbf{t} \in \mathbf{T}|_d$ it is efficient to first obtain a reduced form of $\mathbf{K}$ by convolving it with $\mathbf{k}_d$, denote this reduced form by $\mathbf{K}_d$ - note that if the step size in P, t_0 is greater than a single sample, a downsampled version of $\mathbf{K}_d$ can be computed/stored. The chosen discretization depends on the minimum acceptable mismatch for a chosen detection threshold, in general it should be a small fraction of the chosen duration

d. Then given $\mathbf{K}_d$, the denominator of a transit detection test is simply given by summing elements of $\mathbf{K}_d$ given by the outer product of the index vector $\mathbf{i}$:

$$\mathbf{t}^T\mathbf{K}\mathbf{t} = \sum_{i\in\mathbf{i}}\sum_{j\in\mathbf{i}} \mathbf{K}_d[i,j] \tag{3}$$

In general the number of elements of $\mathbf{i}$ is not the same between transits $\mathbf{t}$, however there is structure to the elements of $\mathbf{i}$. All denominator terms of the transit search, over $\mathbf{T}_d$, can be computed with a scatter/gather pattern. In the gather pattern, each thread sums the elements of $\mathbf{K}_d$ per transit $\mathbf{t}$ and writes the output of a single detection test. In the scatter pattern, each thread reads a single element of $\mathbf{K}_d$ and 'scatters' (cumulative adds) this into output detection test statistics. The scatter form minimizes global memory reads but performs many global writes (however with little output collision). In the scatter form the following occurs:

- Each thread loads a single element of $\mathbf{K}_d[i,j]$
- Loop through and cumulatively add this element to output transit detection statistics that would contain this value.
- This requires calculating output indices, to do this efficiently (few operations) and with minimal divergence among threads, I define a redundant output space $\geq \mathbf{T}_d$. This output space is defined as ranging over $P \in [1, \frac{N}{2}]$, and $t_0 \in [1, \frac{N}{2}]$ so that the output indices $\mathbf{K}_d[i,j]$ gets written to can be calculated from i,j as P all factors of $|i-j|$ and $t_0 = min(i,j)\%P$. Possible to change this/experiment with non grid-like space.
- Perform an atomic CAS to cumulatively update the output detection statistic, because looping through elements of $\mathbf{K}_d$ sequentially, threads in the same block should have different t_0, therefore little output write collision? Should be little thread locking?
- In general the number of writes per thread is not uniform/balanced, nor per output element.
- Possible that all threads within a block could perform a single write each, and the element $\mathbf{K}_d[i,j]$ is stored in shared memory. This may lower imbalanced writes/thread idling.
- Possible issues: atomic CAS worse than loading input from global memory, might be more benefital to use gather.

```
__device__ void collision_free_write(float* output, int output_idx, float value)
{
    while (true)
        {
            float old_value = output[output_idx];
            if (__int_as_float(atomicCAS((int *) &output[output_idx], __float_as_int(old_value
            {
                break;
            }
        }
}


__global__ void parallel_transit_den(float *output, float *input, int half_N)
{
    // For element of input K_d[i,j] calculate output test indices, by finding period as facto
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    float value = input[ty*(half_N*2) + tx];
    if (ty < (2*half_N) && tx < (2*half_N)) {
        if (tx == ty) {
            for (int i=1; i<=half_N; ++i){
                int output_idx = (half_N * (i-1)) + tx%i;
                collision_free_write(output, output_idx, value);
            }
        }
        else {
            for(int i=1; i<=min(int(abs(tx - ty)), half_N); ++i) {
                if (int(abs(tx-ty))%i == 0) {
                    int output_idx = (half_N*(i-1)) + tx%i;
                    collision_free_write(output, output_idx, value);
                }
            }
        }
    }
}
```

For the numerator, calculate $\mathbf{yK}$ once per lightcurve, and then reduce this vector by convolving it with 1D transit template of fixed duration d. Denote this reduced vector as $\mathbf{y}_d$. To calculate all numerator terms for all $\mathbf{t} \in \mathbf{T}|_d$ essentially use a similar pattern as described above and applied to $\mathbf{y}_d = \mathbf{yK} * k_d$:

```
__global__ void parallel_transit_num(float *output, float *input, int half_N)
{   // Each thread takes an element of input y_d and value P, write to output index P*half_N +
    int tx = blockIdx.x*blockDim.x + threadIdx.x;
    int ty = blockIdx.y*blockDim.y + threadIdx.y;
    int p = ty + 1; // period index
    float value = input[tx];
    int t0 = tx%p; // epoch
    if (p <= half_N && tx < 2*half_N) {
        int output_idx = (half_N * (p-1)) + t0;
        collision_free_write(output, output_idx, value);
    }
}
```

To calculate the transit detection tests, pointwise divide the output numerators by the square root of output denominators. This should be performed for all $d \in [1, 16]$ hours. Comparing this computational pattern on the GPU (GeForce RTX 2080, 3072 cores) /CPU (Intel(R) Core(TM) i9-9900K, however only using a single core) for a 8 days of data of observational data with $N \sim 10^4$, takes $5s$ on the GPU vs $250s$ CPU. Note that in this simple version, I am only computing the number of detection tests dependent on the observational length, however for a year of data I would combine these partial output statistics to compute detection tests for candidate transits up to the maximum orbital period (150 days).

The gather pattern is likely more efficient despite requiring a large number of reads from global memory, these can be cumulatively loaded to registers. Look into pre-compute for non-grid like transit search (grouping threads of the same orbital period into blocks so that there are not asymmetric thread wait times).