GPU Accelerated Transiting Exoplanet Detection

Jamila Taaki

1 Introduction

The primary method of exoplanet detection is the transit detection method and to-date has produced the majority of the 5200 + known exoplanets (https://exoplanetarchive.ipac.caltech.edu/). Transit detection is a method to find exoplanets in orbit around a star from time-series brightness measurements (lightcurve) of an unresolved star-planet system. A planet will produce a brief but detectable dip in the observed stellar brightness as it eclipses our line of sight to the star; this is referred to as a transit event. If periodic transit events are observed in a lightcurve, one can conclude that they are likely caused by a planet eclipsing a star and are not due to brightness fluctuations of the star. Example lightcurves and exoplanet transit signals are visualized here: https://twitter.com/exoplanet_day.

A typical transit detection code performs a brute force search over the space of plausible exoplanet transit signals $\mathbf{t} \in \mathbf{T}$ for a lightcurve $\mathbf{y}$, computing detection statistics of the form:

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

Where $\mathbf{K}$ is generally a non-sparse matrix which depends on $\mathbf{y}$. A transit signal $\mathbf{t}(P, d, e)$ has an orbital period P, transit duration d and epoch (time of first transit) e, the space of candidate transit signals $\mathbf{T}$ is generated by enumerating $\mathbf{t}$ over suitable P, d, e values. Transit detection instruments observe many stars ($\sim 10^6$), hence performing transit searches over many such targets is a data- and compute-intensive process.

The main bottleneck is the computation of the denominator $\mathbf{t}^T \mathbf{K} \mathbf{t}$ over $\mathbf{t} \in \mathbf{T}$, the goal of this project would be to improve the performance of the overall transit search focused on this part of the computation. This calculation is efficiently computed as simple gather type computation, for a transit $\mathbf{t}$, elements of $\mathbf{K}$ corresponding to the times of 'transit events' are read and summed by a thread, before performing a single output write. However the number of reads per $\mathbf{t}$ is highly variable and so the threads must be efficiently grouped to avoid idling. A sample kernel which calculates the denominator of $L(\mathbf{y})$ for a single $\mathbf{t}$ is shown below, where the input is $\mathbf{K}$. PyCUDA prototype has further details github.com/xiaziyna/CUDA-transit-detection/blob/main/transit_gather.py.

```
__global__ void gather_transit_den(float *output, float *input, int half_N)
{
    int tx = blockIdx.x*blockDim.x + threadIdx.x; // transit parameter: epoch
    int ty = blockIdx.y*blockDim.y + threadIdx.y; // transit parameter: period
    int p = ty + 1;
    int t0 = tx;
    int no_transit = floorf(((2*half_N)-t0-1)/p)+1;
    float value = 0.0;
    if (t0 < p && p <= half_N) {
        for (int i=0; i < no_transit; ++i) {
            for (int j=0; j < no_transit; ++j) {
                value += input[((i*ty)+tx)*(2*half_N) + (j*ty) + tx];
            }
        }
        output[(half_N * (p-1)) + t0] = value;
    }
}
```

This is one such simple prototype, in general $\mathbf{T}$ is highly structured and there are different optimizations that could be applied (data layout) etc. Another possibility is to extend this work over multiple data-segments and efficiently combine outputs.