GPU-Enabled Searches for Periodic Signals of Unknown Shape

Michael Gowanlock[a,*], Nathaniel R. Butler[b], David E. Trilling[c], Andrew McNeill[c]

[a] *School of Informatics, Computing, and Cyber Systems, Northern Arizona University, Flagstaff, AZ, 86011, USA*
[b] *School of Earth & Space Exploration, Arizona State University, Tempe, AZ, 85287, USA*
[c] *Department of Astronomy & Planetary Science, Northern Arizona University, Flagstaff, AZ, 86011, USA*

Abstract

Recent and future generation observatories will enable the study of variable astronomical phenomena through their time-domain capabilities. High temporal fidelity will allow for unprecedented investigations into the nature of variable objects — those objects that vary in brightness over time. A major bottleneck in data processing pipelines is constructing light curve solutions for catalogs of variable objects, as it is well-known that period finding algorithms are computationally expensive. Furthermore, there are many period finding algorithms that are often suited for specific science cases. In this paper, we present the first GPU-accelerated Super Smoother algorithm. Super Smoother is general purpose and uses cross-validation to fit line segments to a time series, and as such, is more computationally expensive than other algorithms, such as Lomb-Scargle. Because the algorithm requires making several scans over the input time series for a tested frequency, we also propose a novel generalized-validation variant of Super Smoother that only requires a single scan over the data. We compare the performance of our algorithms to analogous parallel multi-core CPU implementations on three catalogs of data, and show that it is generally advantageous to use the GPU algorithm over the CPU counterparts. Furthermore, we demonstrate that our single-pass variant of Super Smoother is roughly equally as accurate at finding correct period solutions as the original algorithm. Our software supports several features, such as batching the computation to eliminate the possibility of exceeding global memory on the GPU, processing a single object or batches of objects, and we allow for scaling the algorithm across multiple GPUs.

Keywords: asteroids: general, massively parallel algorithms, methods: data analysis, methods: numerical, single instruction, multiple data, stars: variables

1. Introduction & Background

Finding the periodic signals of variable objects in astronomical catalogs is a computationally expensive task. New telescope facilities, such as the Rubin Observatory, will have significant time-domain capabilities, where the magnitudes of objects will be recorded with high frequency over long time scales. Consequently, the astronomy community will be largely interested in finding the periods of variable objects from near-future large synoptic surveys, such as the Vera Rubin Legacy Survey of Space and Time (LSST; LSST Science Collaboration, 2009).

The astronomy community has experience with processing large catalogs of light curve data, such as those produced by the Zwicky Transient Facility (ZTF; Graham et al., 2019), Asteroid Terrestrial-impact Last Alert System (ATLAS; Tonry et al., 2018), All Sky Automated Survey for Supernovae (ASAS-SN; Shappee et al., 2014), Catalina Real-Time Transient Survey (CRTS; Drake et al., 2014), and the Panoramic Survey Telescope and Rapid Response System (Pan-STARRS1; Chambers et al., 2016).

We summarize three factors will impact the computational cost of period finding on large astronomical catalogs: (*i*) there will be a large number of objects to consider; (*ii*) there will be potentially large frequency spaces, due to the vast range of scientific cases (searches may span hours to years); and, (*iii*) there will be a potentially large number of data points to consider in each object's light curve. The number of data points in a light curve and the number of frequencies searched directly impact the computational complexity of all period finding algorithms.

There are many period finding algorithms in the literature (see, for example, Lomb, 1976; Stellingwerf, 1978; Scargle, 1982; Dworetsky, 1983; Friedman, 1984; Schwarzenberg-Czerny, 1989; Reimann, 1994; Schwarzenberg-Czerny, 1996; Palmer, 2009; Zechmeister and Kürster, 2009; Townsend, 2010; McWilliam, 2011; Huijse et al., 2012; Graham et al., 2013). These algorithms target different aspects of period finding, such as reducing the computational complexity of the algorithm, parallelizing the algorithm, or improving the probability that an algorithm is able to correctly identify a periodic signal in a time series. One key difference among algorithms is that some are phased-based and require sorting the time series by a searched period, such as SuperSmoother (Friedman, 1984), whereas others such

*Corresponding author
Email address: michael.gowanlock@nau.edu (Michael Gowanlock)

as Lomb-Scargle (Lomb, 1976; Scargle, 1982), are not phased-based and thus do not require this pre-processing step.

Lomb-Scargle has a low computational complexity compared to other algorithms and has been widely utilized in scientific software packages, such as SciPy (Virtanen et al., 2020) and AstroPy (Price-Whelan et al., 2018). In contrast to Lomb-Scargle that fits a sinusoid to the data to find periodic signals, SuperSmoother uses cross-validation to fit the data using line segments. Consequently, periodic signals that have greater structure and that are not well-represented by a sinusoid may be better suited to smoothing algorithms, such as SuperSmoother. In addition, most ground-based observatories have a 24-hour observing cadence. This cadence can directly produce a periodic artifact in the periodogram, where period finding algorithms detect this 24-hour signature, which may falsely report that the period of an object is likely to be 24-hours (or an alias, such as 12 or 48 hours). An example of this is demonstrated in Figure 1, which shows the periodogram of an RR-Lyrae star derived by Lomb-Scargle and SuperSmoother. Here, we observe that the power is greater for SuperSmoother than Lomb-Scargle at the true frequency (1.9 day^{-1}), where the former algorithm has a higher probability of deriving the true period than the latter. Given a large catalog of objects from a ground-based observatory, it may be preferable to employ SuperSmoother over Lomb-Scargle despite its greater computational cost.

Similarly to other works that utilize GPUs to improve the performance of data processing pipelines for large-scale surveys (Katz et al., 2021; van Roestel et al., 2021; Coughlin et al., 2021), this paper proposes a GPU-accelerated SuperSmoother algorithm, and a more computationally efficient variant. Since each searched frequency can be computed in parallel, the GPU is an excellent architecture for carrying out this computation. In summary, this paper makes the following contributions.

- We propose the first GPU-accelerated SuperSmoother algorithm in the literature.
- To address the high computational complexity of SuperSmoother, we propose a single-pass variant of the algorithm. This algorithm uses generalized-validation instead of cross-validation, and makes fewer scans over an object's light curve. We denote this variant as SuperSmootherSP.
- To exploit parallel architectures, each frequency must be searched in parallel. This necessitates replicating the dataset for each searched frequency and sorting it by the phase derived from the searched period. This significantly increases the memory footprint of the algorithm which can exceed the GPU's global memory capacity. To address this limitation, we incorporate a batching scheme that splits the search frequencies into several discrete batches, which obviates GPU memory limitations.
- We optimize the original and single-pass variant for execution on the GPU. Optimizations include exploiting shared-memory and modifying the memory access patterns to exploit coalesced global memory accesses.
- Our software allows for scaling the computation of a catalog of objects, or a single object, across multiple GPUs.
- We evaluate our algorithm on two platforms, one with a single GPU, and one with four GPUs, and show performance across two single object datasets and three real-world catalogs, *Stripe82*, *TESS*, and *LINEAR*. These catalogs span a small, medium, and large number of objects, respectively.

The paper is organized as follows. Section 2 describes the original SuperSmoother algorithm and our associated GPU implementation, and Section 3 presents our single-pass variant of the algorithm. Section 4 summarizes the features of our software and supported modes of operation. Section 5 experimentally evaluates the algorithm on several datasets and experimental scenarios. Finally, Section 6 concludes the work and describes future research directions.

2. GPU Version of Friedman's Super Smoother

2.1. Overview and Challenges

The SuperSmoother algorithm performs local linear regression to fit line segments to data points. In the astronomical context, the data points are typically defined as time, magnitude, and photometric error. Using several bandwidths, it uses cross-validation to determine the best fitting line segment for a local subset of the data. In its original form, the SuperSmoother algorithm requires making nine passes (smooths) over the input dataset for a given frequency, which makes it much more computationally expensive than other period finding algorithms that derive periods in a single pass for a given frequency, such as the single pass Lomb-Scargle Algorithm (Lomb, 1976; Scargle, 1982; Press et al., 1992; Townsend, 2010).

Here, our objective is to search frequencies in parallel using a single GPU thread per frequency, thus exploiting the GPU's massive parallelism. However, a major concern for the parallelization and performance of the SuperSmoother algorithm on the GPU is the algorithm's space complexity. The memory footprint scales with frequency and number of data points for the following elements of the algorithm:

- *sc*- Scratch space: $8N_tN_f$
- *smo*- Output smooth array: N_tN_f

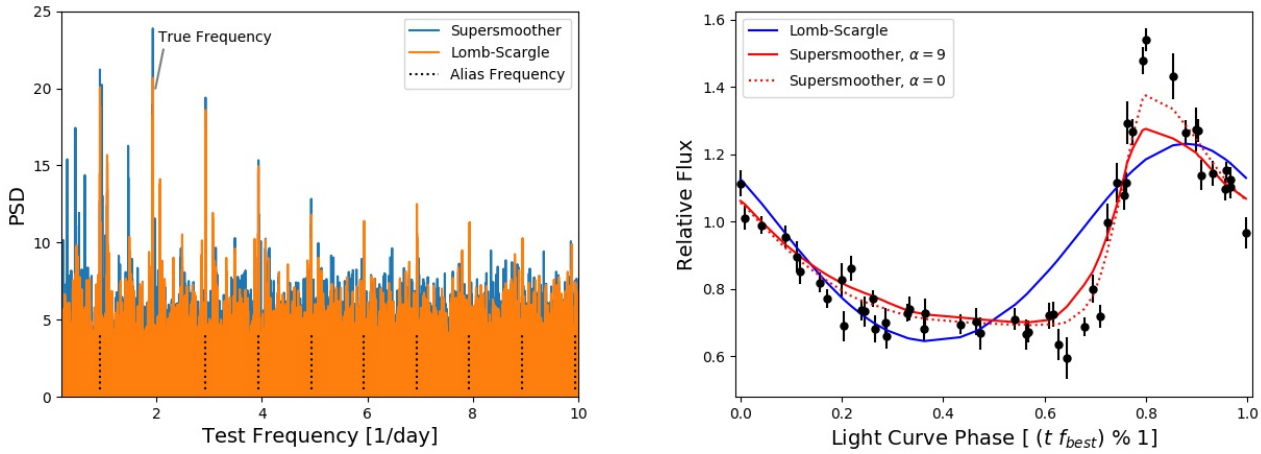

Figure 1: *Left:* The periodogram derived for an RR-Lyrae variable star from the SDSS survey. There are multiple candidate frequencies because the daily observing period is near the true source frequency (1.9 day^{-1}). *Right:* The identification of the true period is accomplished by fitting models that can account for the source flux variation, which is not simply sinusoidal. The algorithm uses a parameter α which can take a value in the range [0, 10], and we show $\alpha = 0$ and $\alpha = 9$.

- *freqarr*- Bookkeeping of frequency ids for the back-to-back sort: $N_t N_f$
- *argkeys*- Keys sorted by values in the back-to-back sort: $N_t N_f$
- *t1*- Time after sorting phased time series: $N_t N_f$
- *t1Asort*- Time after argsorting t1: $N_t N_f$
- *yAsort*- Data after argsorting by t1: $N_t N_f$
- *wAsort*- Weights after argsorting by t1: $N_t N_f$

Note that the largest contribution to the memory footprint is scratch space. The total space complexity of the above components with constant factors is: $15 N_t N_f$. With the exception of working memory for the back-to-back sorting operation, the other components of the algorithm require minimal additional space relative to the above factors.

As an illustrative example, consider an object with $N_t = 50$ data points, and a frequency grid with $N_f = 10^6$ searched frequencies (typical sizes for N_t and N_f). Assuming the data are stored in 64-bit floating point precision (FP64), the space complexity is 5.59 GiB. Modern GPUs have roughly 5–20 GiB of on-card memory, so period searches with a large number of frequencies and/or data points may be intractable due to memory limitations.

2.2. GPU Implementation Details

As a baseline for comparison, we directly implement Friedman's SuperSmoother algorithm (Friedman, 1984) using the GPU. Because the algorithm has to pre-process the data and perform the sorting procedure, the algorithm requires the execution of several kernels. Algorithm 1 provides an overview, and we present the main GPU kernel in Listing 1. In the pseudocode, parameters that are computed in a given GPU kernel are shown in bold face, whereas inputs to the kernel are shown without bold face. In Algorithm 1, only the compute_chi0_tt_weights and findPeriod functions are computed on the host. All other operations are computed on the GPU as CUDA kernels.

Algorithm 1 takes as input the length of the time series, N_t, the number of frequencies N_f, the time (t), magnitude (y), and photometric error (e). The algorithm returns the period corresponding to the frequency with the highest power. On line 2, the algorithm computes the `chi0` term which will be used to compute the periodogram, an array `tt` which is the time, t, after the minimum time has been subtracted, and a weights array (w) computed as e^{-2}. Next, on line 3, the algorithm transfers tt, y, and w to the GPU. Line 4 computes $t1$ as follows: `t1=(tt mod p)/p`. Line 5 initializes the `argkeys` and `freqArr` arrays, where for each frequency, `argkeys` is initialized with values $0, 1, \ldots, N_t - 1$, and `freqArr` is initialized using the frequency number for N_t elements. Next, line 6 performs an argsort on $t1$. Because we perform the argsort for all frequencies examined at once instead of executing several smaller independent sorts for each frequency, this requires a back-to-back sort. The sort uses $t1$, `argkeys`, and `freqArr` where `argkeys` is the ordered set of keys after argsorting by $t1$, and `freqArr` is used to map each index to its respective frequency in `argkeys`. The sort is implemented using two calls to the Thrust library's `thrust::stable_sort_by_key()` function (Bell and Hoberock, 2012), which is a state-of-the-art GPU sorting algorithm. Next, the main SuperSmoother kernel is called on line 8, which is outlined in Listing 1 below. The kernel takes as input several parameters described above, and outputs `smo`, an array of smoothed values. Using `smo`, the periodogram is computed on line 9, and is then returned to the host on line 10. Finally, the period is com-

Algorithm 1 Pseudocode overview of SuperSmoother.

```
1: procedure SuperSmoother(N_t, N_f, t, y, e)
2:     chi0, tt, w ← compute_chi0_tt_weights(N_f, t, y, e)
3:     transferDataHostToGPU(tt, y, w)
4:     computePhasedTime(N_t, N_f, f_min, df, tt, t1)
5:     initKeyArrays(argkeys, freqArr)
6:     argsort(t1, freqArr, argkeys)
7:     mapArgsort(argkeys, t1, y, w, t1Asort, yAsort, wAsort)
8:     supsmu(N_t, N_f, iper, span, alpha, t1Asort, dataAsort, weightsAsort, sc, smo)
9:     computePgram(N_t, N_f, chi0, smo, dataAsort, weightsAsort, pgram)
10:    transferDataGPUToHost(pgram)
11:    p ← findPeriod(N_f, pgram, Δf)
12:    return p
13: end procedure
```

puted on line 11, which performs an $\text{argmax}_x(pgram)$ to find the index, x, in the periodogram with the maximum power and the period is computed as $p = (f_{min} + x\Delta f)^{-1}$.

We briefly explain how all of the CUDA kernels have been parallelized. The `computePeriodModF`, `initKeyArrays`, and `mapArgsort` kernels all use $N_f N_t$ threads. The `computePgram` kernel uses $8N_f$ threads. Lastly, as a baseline, our global memory SuperSmoother kernel, `supsmu`, uses N_f threads.

Listing 1 presents the main SuperSmoother kernel which is directly ported from Friedman's algorithm (Friedman, 1984). Note that as a baseline we simply use global memory and do not store any information in shared memory. The reason for this is twofold. First, each frequency has a different set of data that it needs to iterate over; therefore, threads cannot share information, thus limiting the potential benefit of data reuse. Second, while we can cache data elements into shared memory for each thread, such as the `t1Asort`, `dataAsort`, and `weightAsort` arrays, this can utilize a significant amount of shared memory. Depending on the size of N_t, it is possible that too much shared memory could be requested which would result in a kernel launch failure.

In Listing 1, we show all floating point values being stored in 64-bit precision. However, the source code allows the user to toggle between using FP32 or FP64 using a C macro. For clarity, we did not present the macro in the listing.

In this paper, we assume that all objects are independent of each other, and that they do not share the same time sampling. Typically, for all sky surveys, which is the primary motivator for this work, there may be a low probability that objects share the same telescope pointings for their entire observational records. However, in the case where a targeted astronomical campaign were to observe objects in the same field and generate a single time sampling appropriate for all objects, it would be possible to reuse the same folded time intervals for all objects. We do not examine that optimization here, as it breaks the assumption that all objects are independent, and as we will show in Section 5, the folding step of the algorithm is not a bottleneck relative to the other algorithm components.

Listing 1: Listing of the SSO-Global global memory CUDA kernel.

```
1  __global__ void SSO−Global(const int N_t, const int N_f,
2  const int iper, const double span, const double alpha,
3  double * t1Asort, double * dataAsort,
4  double * weightsAsort, double * sc, double * smo)
5  {
6      unsigned int tid=threadIdx.x+(blockIdx.x*blockDim.x);
7      if (tid>=numThreads) return;
8
9      //offsets into arrays for the thread
10     const unsigned int dataOffset=tid*n;
11     const unsigned int scOffset=tid*n*8;
12
13     //pointers to time, data, weights
14     //offsets for smo and sc
15     double * x=t1_sortby_argkeys+dataOffset;
16     double * y=data_sortby_argkeys+dataOffset;
17     double * w=weights_sortby_argkeys+dataOffset;
18     double * smo_thread=smo+dataOffset;
19     double * sc_thread=sc+scOffset;
20     int i,j,jper;
21     double vsmlsq,sw,sy,a,scale,resmin,tmp,f;
22
23     //spans to be estimated: tweeter, midrange, and woofer
24     double spans[] = {0.05,0.2,0.5};
25
26     if (x[n−1]<=x[0]) {
27       sy=0.0;
28       sw=sy;
29       for (j=0;j<n;j++) {
30         sy=sy+w[j]*y[j];
31         sw=sw+w[j];
32       }
33       a=0.0;
34       if (sw>0) a=sy/sw;
35       for (j=0;j<n;j++) smo_thread[j] = a;
36       return;
37     }
38
39     i=n/4−1;
40     j=3*(i+1)−1;
41     scale=x[j]−x[i];
42     vsmlsq=1.e−6*scale*scale;
43
44     jper=iper;
45     if (iper==2 && (x[0]<0 || x[n−1]>1)) jper=1;
46     if (jper<1 || jper>2) jper=1;
47     if (span>0) {
48       //Fixed span
49       smoothkernel(n,x,y,w,span,jper,vsmlsq,
50       smo_thread,sc_thread);
51       return;
52     }
53
```

```
54      //If we made it here, the span will be
55      //estimated and variable
56      for (i=0;i<3;i++) {
57        smoothkernel(n,x,y,w,spans[i],jper,vsmlsq,
58        sc_thread+2*i*n,sc_thread+6*n);
59        smoothkernel(n,x,sc_thread+6*n,w,spans[1],-jper,
60        vsmlsq,sc_thread+(2*i+1)*n,sc_thread+7*n);
61      }
62
63      for (j=0;j<n;j++) {
64        resmin=1.e20;
65        for (i=0;i<3;i++) {
66          if (sc_thread[j+(2*i+1)*n]<resmin) {
67            resmin=sc_thread[j+(2*i+1)*n];
68            sc_thread[j+6*n]=spans[i];
69          }
70        }
71        if (alpha>0 && alpha<=10 &&
72        resmin<sc_thread[j+5*n] && resmin>0) {
73          tmp = resmin/sc_thread[j+5*n];
74          if (tmp<1.e-7) tmp=1.e-7;
75          sc_thread[j+6*n]+=(spans[2]-sc_thread[j+6*n])*
76          pow(tmp,10.0-alpha);
77        }
78      }
79
80      smoothkernel(n,x,sc_thread+6*n,w,spans[1],-jper,vsmlsq,
81      sc_thread+n,sc_thread+7*n);
82
83      for (j=0;j<n;j++) {
84        if (sc_thread[j+n]<=spans[0]) sc_thread[j+n]=spans[0];
85        if (sc_thread[j+n]>=spans[2]) sc_thread[j+n]=spans[2];
86        f=sc_thread[j+n]-spans[1];
87        if (f<0) {
88          f/=spans[0]-spans[1];
89          sc_thread[j+3*n]=(1.0-f)*sc_thread[j+2*n]+f*
90          sc_thread[j];
91        } else {
92          f/=spans[2]-spans[1];
93          sc_thread[j+3*n]=(1.0-f)*sc_thread[j+2*n]+
94          f*sc_thread[j+4*n];
95        }
96      }
97      smoothkernel(n,x,sc_thread+3*n,w,spans[0],-jper,vsmlsq,
98      smo_thread,sc_thread+7*n);
99      return;
100   } //end of GPU kernel
```

Now that we have outlined the baseline implementation of the original SuperSmoother algorithm, we describe three kernel designs, including the baseline global memory kernel and a shared memory kernel. As we will discuss, the shared memory kernel may fail to launch due to memory limitations. Therefore, we will show that both kernels are needed to make the original SuperSmoother algorithm achieve good performance and be robust to the size of the light curves, N_t. For brevity, we do not detail the code listings for each kernel here, but we refer the interested reader to our open source code repository for further information.

2.3. Global Memory Kernel: A Baseline

The global memory kernel, SSO-Global, was described in Listing 1. The kernel only uses global memory and is a baseline for which to compare to our other kernels.

2.4. Shared Memory Kernel: One Thread Per Frequency

The original SuperSmoother algorithm needs to make several passes over the input dataset and writes intermediate data to a scratch space buffer that is allocated for each frequency, N_f. This requires many accesses to global memory. Because the accesses to the scratch space are data-dependent and are not well constrained, it is not possible to achieve good coalesced memory accesses in a global memory kernel. To address this limitation, the shared-memory kernel caches data in shared-memory to reduce the latency of global memory accesses. Each thread is assigned a single frequency to process. For each thread in a CUDA block, we allocate scratch space for the time, data, and weights in shared memory (where the pointers `x, y, z` in Listing 1 will reference shared memory). In addition, we store the three spans in shared-memory to reduce register usage, but we do not store other temporary and constant variables so that we do not increase pressure on the capacity of shared memory. Since the original SuperSmoother algorithm requires making nine passes over the dataset, this optimization eliminates a significant number of (unconstrained) global memory accesses. We denote this CUDA kernel as SSO-Thread.

2.5. Cascade Mode: Addressing Shared-Memory Limitations

We will experimentally show that the best performance is achieved using the SSO-Thread kernel; however, as described above, the SSO-Thread can fail to execute due to insufficient shared memory. Therefore, we propose an execution option that attempts to execute the SSO-Thread kernel and then the SSO-Global kernel if SSO-Thread fails to execute[1]. This ensures that the SuperSmoother algorithm achieves the best performance possible while being robust to the size of the light curve, N_t. Additionally, since we check for CUDA kernel launch failures at runtime, the algorithm is also robust to differences in hardware platforms that have different shared memory sizes. This ensures that future generations of GPUs are supported by the software.

3. A Single-Pass Generalized Validation Variant of SuperSmoother

As described in Section 2.2, the original SuperSmoother algorithm requires a significant amount of memory for scratch space. To address this problem, we propose a variant of the SuperSmoother algorithm. In contrast

[1]Note that we attempted another kernel design that uses a single block to compute a frequency which could be executed between SSO-Thread and SSO-Global. However, this kernel performed significantly worse than the SSO-Thread and SSO-Global kernels. Therefore, we do not present its implementation in this paper.

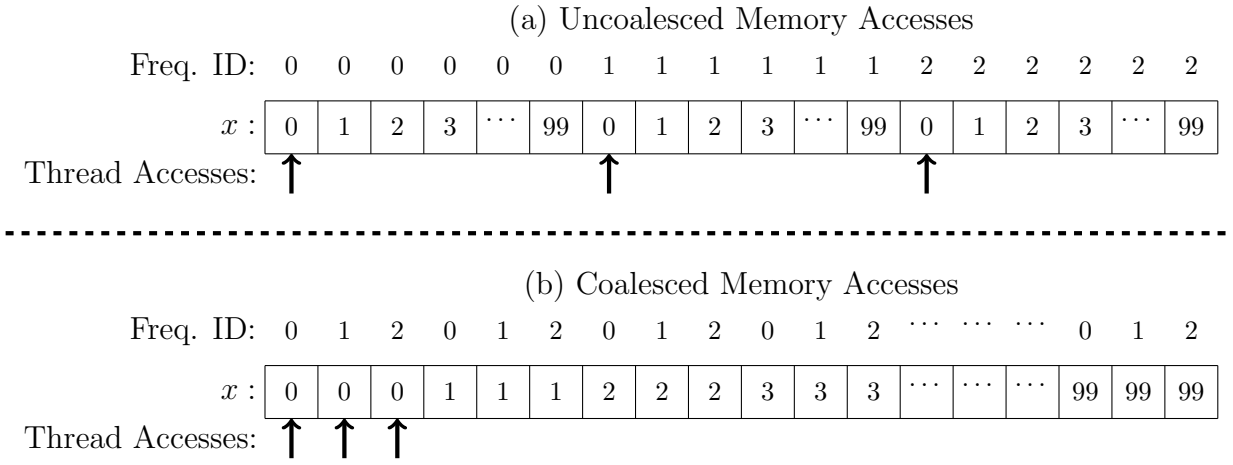

Figure 2: Example of transforming memory accesses to the time, magnitude, and weights arrays with $N_f = 3$ and $N_t = 100$. For illustrative purposes, we show a search using only $N_f = 3$ frequencies. (a) The data layout and memory access patterns where all data elements for a given frequency are stored contiguously. This memory access pattern leads to uncoalesced memory accesses to global memory. (b) Transforming the memory access pattern in (a) by storing the same data element IDs contiguously across frequencies to achieve coalesced memory accesses.

to using a cross-validation approach as in the original algorithm, our algorithm uses a *generalized validation* approach which only requires a single call to the smooth function and eliminates the use of the scratch space required of the original algorithm. While this approach yields a slightly worse fit to the data compared to the original SuperSmoother algorithm, it requires less computation and is still effective for period finding. We will demonstrate that our single-pass algorithm achieves roughly equivalent period finding accuracy as the original algorithm.

Similarly to the original algorithm, we propose several GPU kernel designs. All of the other steps in the algorithm are equivalent, with the exception of the kernel (line 8 in Algorithm 1). We outline the kernels as follows.

3.1. Single-pass Analogs of the Original Kernels

The proposed single-pass kernels are direct analogs of the original kernels outlined in Sections 2.3 and 2.4. In particular, the SSSP-Global kernel is a baseline that does not utilize any shared memory. The SSSP-Thread kernel assigns a single thread to parallelize each frequency. Each of these kernels have the same characteristics as the kernels in the original algorithms with the exception that only a single smooth operation is required for our generalized validation approach.

3.2. Global Memory Kernel: Exploiting Coalesced Memory Accesses

The original SuperSmoother algorithm requires nine calls to the smooth function. A drawback of the original algorithm's SSO-Global kernel is that it requires scratch space for each frequency that is accessed in a non-deterministic manner. It is well-known that to achieve good performance on the GPU, global memory accesses should be coalesced where possible (Bell and Hoberock, 2012). Otherwise, multiple transactions are required to retrieve data from global memory, which does not effectively utilize global memory bandwidth. Incorporating memory access patterns that guarantee coalesced memory accesses into the original SSO-Global kernel is not possible due to the abovementioned non-deterministic memory access patterns. In contrast, our single-pass variant has deterministic memory access patterns to the time, magnitude, and weights data (`x`, y, and w in Listing 1). We exploit these regularized memory access patterns to create a global memory kernel that has coalesced memory accesses.

Figure 2 illustrates transforming the non-coalesced memory accesses patterns into coalesced memory accesses, where $N_f = 3$ and $N_t = 100$. Figure 2(a) shows accessing an array (here, x refers to the time, magnitude, or weights arrays), where elements corresponding to the same frequency are stored contiguously. Therefore when the $N_f = 3$ threads access the data, the memory accesses occur with a stride of size N_t. This requires three transactions to global memory to occur. In contrast, Figure 2(b) shows accessing the same array, where data elements with the same ID are stored contiguously across the frequencies. Consequently, the $N_f = 3$ threads access the data contiguously, and this only requires a single coalesced load from global memory. This reduces the number of global memory loads that are required of the algorithm, thereby effectively utilizing global memory bandwidth.

We refer to the global memory kernel with coalesced memory accesses to the time, magnitude, and weights arrays as SSSP-Coalesced. Note that since we modify the data layout, the `mapArgsort` and `computePgram` kernels in Algorithm 1 on lines 7 and 9 need to be modified to use this transformation.

4. Optimizations and Summary of Modes

In this section, we outline optimizations common to both the original and single-pass algorithms and summarize the functionality of our software.

4.1. Batching Scheme

As described in Section 2.1, the SuperSmoother algorithm requires a significant amount of global memory; therefore, it may be intractable to perform a period search across all frequencies without exceeding global memory capacity. Since the frequencies can be searched independently, we divide the N_f frequencies into N_b batches. We compute N_b by estimating the total global memory footprint, r, divided by the global memory size of the device, s (in GiB). The software automatically detects the global memory size of the GPU such that it does not require any user intervention. Because r is an estimate of the total global memory footprint, and because we do not have direct control over when memory will be freed by the system, we underestimate the global memory capacity by a coefficient β where $\beta < 1$ to avoid exceeding global memory capacity. The number of batches is given as follows:

$$N_b = r \cdot (\beta s)^{-1}. \tag{1}$$

In this paper, we set $\beta = 0.75$[2]. Since the original SuperSmoother algorithm requires scratch space and the single-pass variant does not, we require two different values for r for a given level of floating point precision. We denote r_{orig} and r_{sp} as the estimate of the original SuperSmoother and the single-pass algorithm's memory footprints, respectively. We estimate these terms in GiB for both FP32 and FP64 precision floating point values as follows.

- $r_{orig}^{FP32} = 1024^{-3} \cdot [(4N_f) + (12N_t) + (84N_fN_t)]$
- $r_{orig}^{FP64} = 1024^{-3} \cdot [(8N_f) + (24N_t) + (160N_fN_t)]$
- $r_{sp}^{FP32} = 1024^{-3} \cdot [(4N_f) + (12N_t) + (52N_fN_t)]$
- $r_{sp}^{FP64} = 1024^{-3} \cdot [(8N_f) + (24N_t) + (96N_fN_t)]$

From the estimates of the FP64 memory footprints, the original SuperSmoother algorithm requires a factor 1.67 more memory on the N_fN_t terms than the single pass algorithm due to additional scratch space. This increases the number of batches and kernel invocations that need to be executed. To accommodate the batching scheme, Algorithm 1 is modified by including a loop around lines 4–9 that iterates over the batches from from $1, 2, \ldots, N_b$.

[2]Note that this default value is useful for a dedicated GPU where memory is not used for other purposes. In cases where the GPU is not dedicated, such as in a laptop that is also rendering a user interface, then it may be preferable to decrease the value of β. This parameter can be modified in the parameters file in the source code.

4.2. Multi-GPU Support

We add multi-GPU functionality to the algorithm. This allows us to distribute batches to GPUs, where N_{GPU} denotes the number of employed GPUs. When we use multi-GPU mode, to obtain low load imbalance at the end of the computation, we ensure that the number of batches evenly divides the number of GPUs. Therefore, when we estimate the number of batches when using multi-GPU mode, we use N_b' batches, where $N_b' = \lceil \frac{N_b}{N_{GPU}} \rceil \cdot N_{GPU}$. Therefore, each GPU will compute N_b'/N_{GPU} batches, where $N_b' \bmod N_{GPU} = 0$. To enable the GPUs to execute concurrently, we use N_{GPU} threads on the host, which are parallelized using OpenMP.

There may be the case where GPUs in a multi-GPU system have differing capabilities. In batch mode, each GPU is assigned a different object. In this case, the code dynamically assigns objects to the GPUs. This is because objects can have differing time series lengths, N_t, which directly impact execution time. Thus, even when using the same GPUs in a multi-GPU system, the GPUs are unlikely to process the exact same number of objects. Similarly, if using GPUs with varying capabilities, this mechanism will allow the GPUs to dynamically process the objects, where the slower GPU(s) will process fewer objects on average compared to the faster GPU(s). To do this, a minor modification to the code would be needed to accommodate the global memory capacity of each GPU such that the number of batches can be calculated (Equation 1). When computing the periodogram of a single object, the code could be modified to dynamically assign batches of work to each GPU where the number of batches (Equation 1) is computed using the GPU with the smallest global memory capacity.

4.3. Summary of Modes and Options

We summarize the modes and options provided by our software as follows.

- The software consists of GPU implementations of Friedman's SuperSmoother algorithm and our single-pass variant.
- The software supports both FP32 and FP64 floating point arithmetic modes. This allows the user to select an appropriate level of precision for their scientific investigation. GPUs designed for scientific computing have a significant amount of hardware dedicated to FP64 whereas consumer grade GPUs have minimal hardware dedicated to FP64; therefore, a user may also wish to select the precision level based on available hardware.
- The software also supports processing astronomical catalogs containing different objects. In this mode, we assume the same frequency grid for all objects, as we expect catalogs to target different science cases (e.g., asteroids compared to RR-Lyrae)

- As described in Sections 2.1 and 4.1, the SuperSmoother algorithm requires a significant amount of memory to enable parallelization across frequencies. We provide a batching scheme such that the GPU does not exceed global memory capacity. We automatically detect the GPU's global memory capacity, and estimate the algorithm's memory footprint to determine the number of batches that are required to process an object. This ensures that platform-specific details do not need to be entered by the user.

- Our software supports multi-GPU systems which are becoming increasingly ubiquitous in astronomical data processing pipelines.

5. Experimental Evaluation

5.1. Experimental Methodology

All host code is written in C/C++ and all GPU code is written in CUDA. The code is compiled using the O3 optimization flag, and unless otherwise noted, all reported response times are averaged over 3 time trials. SuperSmoother takes as input a value of α, and in this paper, for period finding purposes, we set $\alpha = 9.0$. The parameter is used to tune how sensitive the algorithm is to outlier data points, and we will discuss the selection of this parameter in Section 5.4.

Table 1 outlines the platforms used in our experimental evaluation. PlatformA contains Intel Xeon CPUs with 16 total physical cores and an Nvidia Quadro GP100 GPU (Pascal generation). PlatformB contains an Intel Xeon CPU with 18 total physical cores equipped with 4 Nvidia Quadro RTX 5000 GPUs (Turing generation). The Quadro GP100 in PlatformA supports both FP32 and FP64 floating point precision. Unless otherwise noted, when we use this platform, we execute the algorithm with FP64. In contrast, the Quadro RTX 5000 GPUs in PlatformB only contain 1/32 the hardware for FP64 as FP32[3]. The small hardware allocation for FP64 allows the GPU to be able to execute programs that contain FP64 instructions, but it is not designed to execute programs that require a significant amount of FP64 computation.

Due to the large range of GPU kernels described in Sections 2 and 3, we summarize the GPU kernels and CPU implementations below.

The kernels associated with the original SuperSmoother algorithm on the GPU are as follows:
•SSO-Global– Global memory baseline (Section 2.3).
•SSO-Thread– Assigns one thread per frequency and uses shared memory (Section 2.4).
•SSO-Cascade– Attempts to execute SSO-Thread and if it has insufficient shared memory, executes SSO-Global (Section 2.5).

The kernels associated with the SuperSmootherSP algorithm on the GPU are as follows:
•SSSP-Global– Global memory baseline (Section 3.1).
•SSSP-Thread– Assigns one thread per frequency and uses shared memory (Section 3.1).
•SSSP-Coalesced– Global memory kernel that exploits coalesced memory accesses (Section 3.2).

The parallel CPU implementations are denoted as follows:
•SSO-CPU– SuperSmoother algorithm configured with the same number of threads as physical CPU cores on a platform.
•SSSP-CPU– SuperSmootherSP algorithm configured the same as the above.

5.2. Datasets

We outline the datasets used in our experimental evaluation as follows. To evaluate executing our algorithms on a single light curve, we employ a dataset, *ObjSmall*, which is a short light curve of an RR-Lyrae star having $N_t = 53$ observations. We employ a synthetic light curve, denoted as *ObjLarge*, containing $N_t = 3,554$ observations corresponding to a large light curve of a synthetic asteroid. The observational record of this object has a cadence similar to that delivered by ZTF (Graham et al., 2019), where we define cadence to be the semi-regular observing pattern. Here, our cadence is based on the ZTF public survey, where data are collected around half of the nights. While we reproduce the semi-regular observing pattern, it does not impact the performance of the algorithms, so it is not important for the purposes of this paper. Lastly, to demonstrate how the algorithms perform as a function of N_t, we create a dataset denoted as *Syn3k*, which is the *ObjLarge* dataset that has been partitioned into several lightcurves having $N_t \in [25, 3000]$.

We employ three real-world catalogs of data for demonstrating the performance of the algorithm for batch processing. All of the objects that we examine may be periodic in their brightness. We use a subsample of 136 RR-Lyrae stars from Sesar et al. (2010) that have known period solutions, denoted as *Stripe82*. We employ a dataset of 37,965 asteroid light curves from TESS cycle 1, denoted as *TESS* (McNeill et al., 2021). In this dataset, asteroid magnitudes vary due to rotation. Finally, we use the *LINEAR* dataset, which contains 94,252 sidereal sources, where a subset are variable stars. A typical scenario for finding variable objects in the *LINEAR* dataset would be to derive the light curves of these objects and select those that have a sufficiently high normalized power. Afterwards, scanning (Burdge et al., 2019) and classifying the objects as being periodic or non-periodic sources (van Roestel et al., 2021) is needed. These three catalogs represent small, medium and large sizes, each of which have

[3] `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`

Table 1: Details of the platforms used in the experimental evaluation. The number of CUDA cores and global memory size are shown for each GPU in PlatformB. FP refers to the floating point precision capabilities of the GPUs.

	CPU				GPU		
Platform	Model	Cores	Clock	Memory	Model	Cores	Memory
PlatformA	2×E5-2620 v4	$2 \times 8 = 16$	2.1 GHz	128 GiB	Quadro GP100	3584	16 GiB
PlatformB	W-2295	18	3.0 GHz	256 GiB	4×Quadro RTX 5000	3072	16 GiB

Table 2: Practical values of N_f for the catalogs that will be processed using batch mode. Frequency ranges are selected based on scientific objective, and in the case of *TESS* cycle 1, the survey duration limits the light curve period range.

Dataset	$[f_{min}, f_{max})$ day^{-1}	Δf	N_f
Stripe82	$[0.1, 10.0)$	3.0×10^{-5}	330,000
TESS	$[0.1, 10.0)$	3.7×10^{-4}	26,757
LINEAR	$[0.01, 10.0)$	4.5×10^{-5}	222,000

different light curve length distributions, which will impact the execution time of the algorithms.

5.3. Selection of The Number of Frequencies

In the experiments that follow, we examine the performance of our algorithms, and this includes observing how performance varies as a function of N_f and N_t. It is well-known that in practice, when selecting N_f, it is important to select a value that will not miss the peaks in the periodogram (VanderPlas, 2018). Consequently, when we perform our experimental evaluation, we select values of N_f that at least bracket a value of this parameter. In what follows, we outline our procedure for selecting N_f when processing batches/catalogs of objects, which is the main objective of this paper.

We use a similar frequency grid prescription as Richards et al. (2011). Let a dataset of objects be denoted as O, where $|O|$ is the number of objects in the dataset. Let each object in a given dataset be denoted as o_i, where $i = 1, 2, \ldots, |O|$. Each object, $o_i \in O$, has a temporal extent denoted as $T_i = T_i^{end} - T_i^{start}$, where T_i^{start} and T_i^{end} refer to the time of the first and last observations of object o_i, respectively. We then compute the maximum temporal extent, T^{max}, in the dataset/catalog as $T^{max} = \max(T_i)$. We compute the frequency grid spacing as $\Delta f = 0.1(T^{max-1})$. We then compute N_f as follows: $N_f = f_{max} - f_{min}(\Delta f^{-1})$. Table 2 shows reasonable values for Δf and N_f using the frequency range, $[f_{min}, f_{max})$, where the frequency range is based on the expected periodic signal range for a given dataset which is driven by the properties of the objects. Note that the properties of the *TESS* dataset are such that we are able to detect light curve periods on the order of tens of days (McNeill et al., 2019), so the applicable period search range is small.

5.4. Selection of the α Parameter

The α parameter in SuperSmoother is used to penalize outliers in the fitting procedure, where $\alpha = 0$ and $\alpha = 10$ indicate low and high penalizations, respectively. The benefit of a high value of α is that it will not overfit outliers in the time series, and since astronomical data may have substantial error, these outliers can yield incorrect periods. Figure 1 demonstrated using two values of α. We find that in the case of asteroids and RR-Lyrae, $\alpha = 9$ yields good periods, and we use this value when processing catalogs of data with SuperSmoother.

We show the case where a low value of α is beneficial over a high value. Transiting exoplanet searches generally use the box least squares (BLS) approach, in which a box filter is scanned over the data to detect transits (Kovács et al., 2002). Here, we use SuperSmoother to derive the period of an exoplanet transit. Figure 3 shows the orbital periods of four exoplanets derived by SuperSmoother as a function of α[4]. The true period is shown by the horizontal red solid line. From the figure, we observe that selecting $\alpha = 0$ is able to correctly derive all of the orbital periods. This is because in the time series of a transiting exoplanet, the transiting data points are outliers, and we do not want to underfit those data points, otherwise the periodic signal will not be found by the algorithm. For comparison purposes, we also show that Lomb-Scargle is unable to derive any of the correct orbital periods. This is unsurprising, as a pure sinusoidal fit is unsuitable for fitting the asymmetric steep profile of these light curves; as mentioned above, these light curves are typically fit using BLS and not Lomb-Scargle.

5.5. Comparison of Frequency Histograms: Lomb-Scargle vs. SuperSmoother

As described in Section 1, one benefit of using SuperSmoother over Lomb-Scargle is that the algorithm is able to differentiate between periods that are aliases of the 24-hour ground-based telescope observing schedule. Consequently, given an input catalog of objects, we would expect that Lomb-Scargle would have a larger number of period solutions that are aliases of 24-hours. To demonstrate this, Figure 4 plots the derived frequency histograms for SuperSmoother and Lomb-Scargle on the *LINEAR* dataset. We find that SuperSmoother and SuperSmootherSP have fewer derived periods at frequencies that are multiples of days^{-1} compared to Lomb-Scargle. Since 24-hour aliases will be present for any period finding method, a typical approach is to remove the 24-hour aliased solutions (Coughlin et al., 2021), and instead select a solution that has a lower power. Another application

[4]These datasets were not described in Section 5.2, as we do not use them to evaluate the performance of the algorithm.

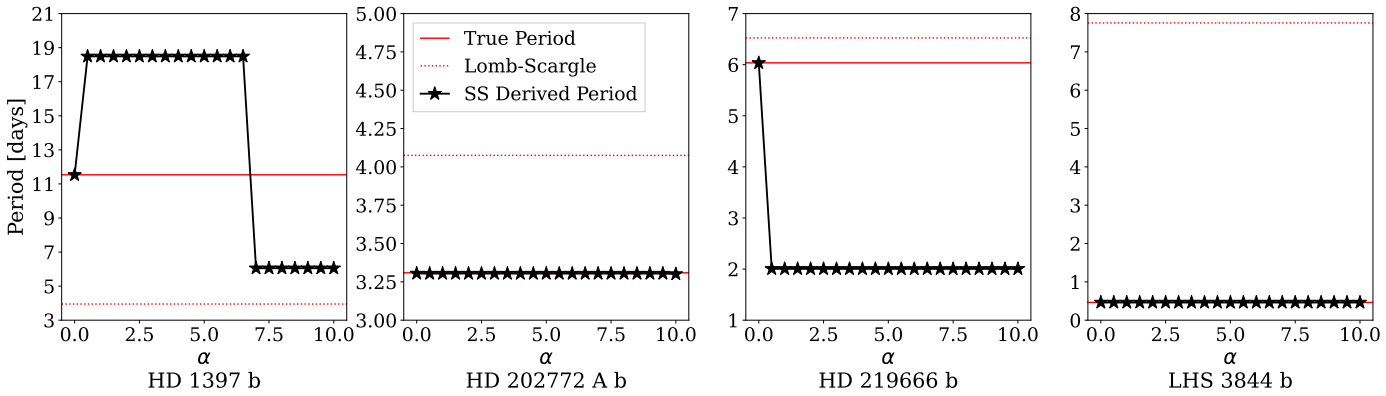

Figure 3: The SuperSmoother derived orbital period as a function of the α parameter for the following four exoplanets: HD 1397 b, HD 202772 A b, HD 219666 b, and LHS 3844 b. Values for the true periods (red solid horizontal line) are obtained from `http://exoplanet.eu/`. The dotted red horizontal line shows the periods derived by Lomb-Scargle, which yields incorrect periods in all four cases. The periods were derived using a uniform frequency grid having $N_f = 10^6$, and frequencies were searched in the range [0.05, 10.0) day^{-1}. Algorithms executed with FP64.

would be to use multiple period finding algorithms (e.g., SuperSmoother, Lomb-Scargle, and possibly others) on an input time series to increase confidence in derived period results. This may be another avenue of discerning between 24-hour solutions.

5.6. Single-pass Variant Accuracy

We use the *Stripe82* dataset of RR-Lyrae to examine the accuracy of SuperSmoother compared to SuperSmootherSP, and we also show the accuracy using Lomb-Scargle for comparison purposes. The periods for these RR-Lyrae were derived using light curve templates (Sesar et al., 2010).

Figure 5 plots the derived periods for SuperSmoother (top), SuperSmootherSP (middle), and Lomb-Scargle (bottom) compared to the real period for the RR-Lyrae in the *Stripe82* dataset. To compute the derived period for a given object, we simply select the frequency having the greatest power value in the periodogram, and do not manually inspect any of the light curves or periodograms. A perfect match between the derived and real solutions would be indicated by a diagonal line. We compute the fraction of matches between the real and derived solutions, where a match is defined as being within 1% of the real period. We find that 83.8% of objects match using SuperSmoother and 84.6% match using SuperSmootherSP, indicating that the faster single-pass variant does not decrease the ability of the algorithm to find the correct period. We find that Lomb-Scargle finds 62.5% of the correct periods, demonstrating that SuperSmoother and its single-pass variant are more effective at accurately deriving periods on this dataset. This motivates the use of SuperSmootherSP over SuperSmoother, as it is highly effective at deriving periods and has a much lower computational complexity.

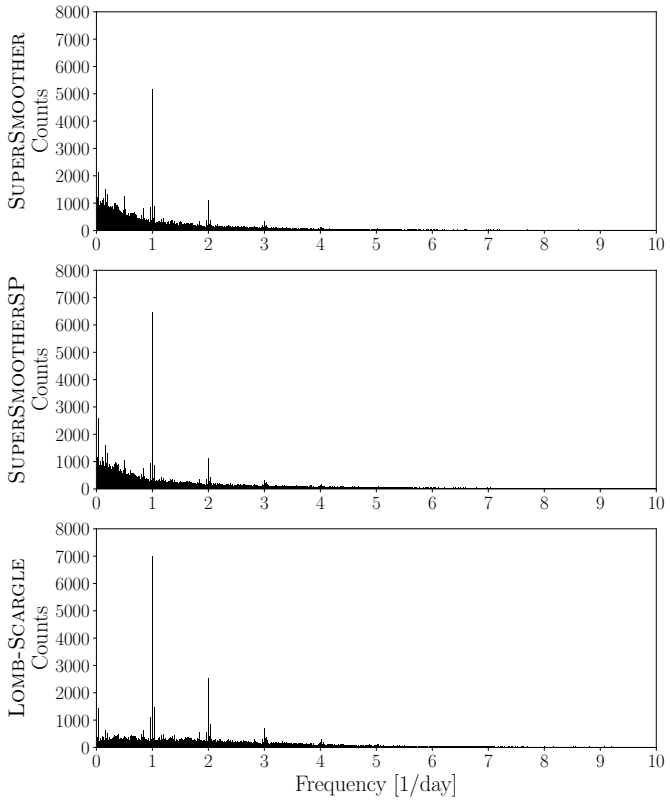

Figure 4: Histograms of the derived frequencies from top to bottom panels using SuperSmoother, SuperSmootherSP, and Lomb-Scargle on the *LINEAR* dataset. The periods were derived using a uniform frequency grid having $N_f = 222,000$ and frequencies were searched in the range [0.01, 10.0) day^{-1}. Algorithms executed with FP32.

5.7. Scalability of CPU Implementations

Before comparing the performance of the CPU and GPU implementations, we assess the scalability of the CPU

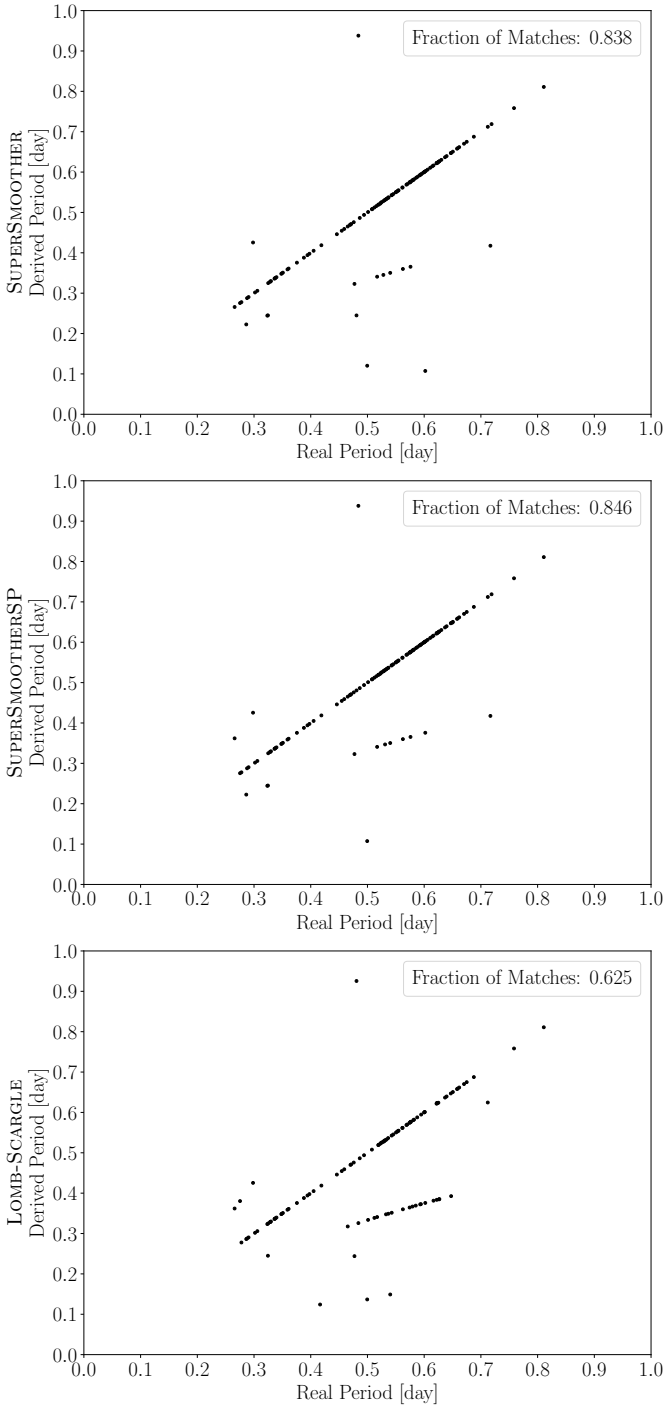

Figure 5: Scatter plots of the derived periods compared to the real periods on the *Stripe82* dataset containing RR-Lyrae stars. Panels from top to bottom are as follows: SuperSmoother, SuperSmootherSP, and Lomb-Scargle. The periods were derived using $N_f = 330,000$ in the frequency range [0.1, 10.0) day^{-1}. Algorithms executed with FP64.

implementations. In this experiment, we carry out the search on a single object, where the frequencies to be searched are partitioned between the CPU threads. Figure 6 plots the speedup compared to the number of threads

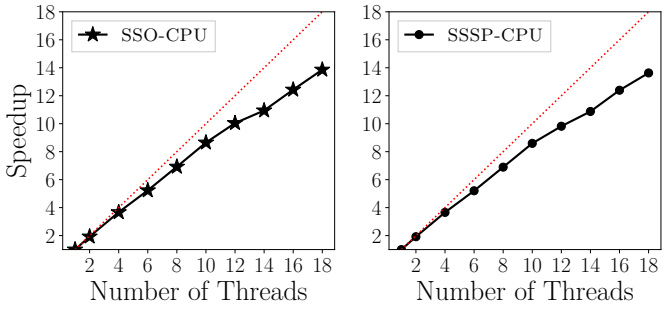

Figure 6: Scalability of the CPU implementations. Speedup as a function of the number of threads on the *ObjLarge* dataset. Frequencies are searched in the range [0.1, 10.0) day^{-1} using $N_f = 5 \times 10^5$. *Left:* SSO-CPU, and *Right:* SSSP-CPU. In each plot, a perfect speedup is indicated by the dashed line. Experiments performed on PlatformB with FP64.

on the *ObjLarge* dataset. We find that SuperSmoother (left panel) and SuperSmootherSP (right panel) achieve excellent scalability on the CPU yielding a speedup on 18 CPU cores of 13.86× and 13.63×, respectively.

Comparing the response times used to generate Figure 6, we find that on 18 cores, SuperSmoother executes in 19.89 s, whereas SuperSmootherSP executes in 17.12 s. Thus, the single-pass variant only provides a minor performance advantage on the CPU. Because the SuperSmoother algorithm makes several passes over the sorted time series for a given frequency, the time series can be read once from main memory and then stored in cache, and subsequent scans over the time series exploit high data reuse in the cache. In contrast, the single-pass variant does not make multiple scans over the time series, and so it is unable to exploit data reuse in the cache to the same degree as the original algorithm. This explains why the performance is similar between SuperSmoother and SuperSmootherSP on the CPU despite the lower complexity of the latter algorithm.

5.8. Performance of Cascade Mode

As described in Section 2.5, in the original SuperSmoother algorithm, we attempt to launch the kernel that executes a single thread per frequency (SSO-Thread). This kernel can fail to launch if too much shared memory is requested, which is a function of N_t. Consequently, we launch the global memory kernel (SSO-Global) if SSO-Thread fails to execute. Figure 7 plots the response time as a function of N_t, where we show the components of SSO-Cascade (SSO-Thread and SSO-Global), as the "x" and "diamond" markers, respectively. For comparison, the dashed black curve shows the execution time of independently launching SSO-Global. We observe that SSO-Thread is useful for small light curves, as it outperforms SSO-Global when $N_t \leq 50$. When $N_t \geq 75$, the cascading kernel will execute SSO-Global as there is insufficient shared memory to launch SSO-Thread. The

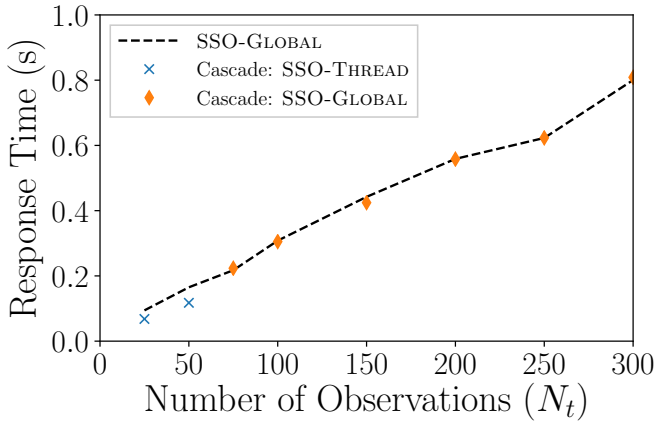

Figure 7: The response time as a function of N_t showing the performance of SSO-Cascade compared to SSO-Global when computing the period for a single object (*Syn3k*). The algorithms are executed using a frequency grid of size $N_f = 10^5$. The plot shows that the SSO-Thread kernel used by SSO-Cascade outperforms SSO-Global when $N_t \leq 50$. When $N_t \geq 75$, cascade mode reverts to SSO-Global due to insufficient shared memory to launch SSO-Thread. Experiments performed on PlatformA with FP64.

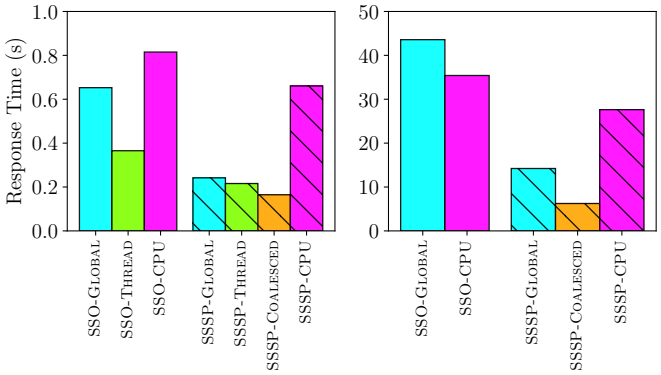

Figure 8: The response time of executing the various kernels on a small and large time series of a single object. *Left: ObjSmall*, and *right: ObjLarge*. Note that SSO-Thread and SSSP-Thread cannot be executed on *ObjLarge* because there is insufficient shared memory to launch the kernel. On both datasets, frequencies are searched in the range [0.1, 10.0) day^{-1} using $N_f = 5 \times 10^5$. SuperSmoother is shown by the plain bars, whereas SuperSmootherSP is denoted by the striped bars. Experiments performed on PlatformA with FP64.

convergence of the dashed curve and the diamond markers indicates that cascade mode does not degrade performance compared to simply executing SSO-Global. Thus, when configuring the software, SSO-Cascade can be used without any performance penalty relative to using the global memory baseline, SSO-Global.

5.9. Performance on the Single Object Datasets

Figure 8 examines the performance of the various GPU kernels on small (left panel) and large (right panel) time series datasets, denoted as *ObjSmall*, and *ObjLarge*. On the *ObjSmall* dataset, SSO-Thread and SSSP-Thread can be executed because there is sufficient shared memory; however, on the *ObjLarge* dataset, there is insufficient memory to launch these kernels.

From Figure 8 (left panel), using the original SuperSmoother algorithm on the *ObjSmall* dataset, SSO-Global performs worse than SSO-Thread. On this dataset, both SuperSmoother and SuperSmootherSP outperform the CPU implementation, SSO-CPU. However, on the larger dataset, *ObjLarge* (Figure 8, right panel), SSO-CPU outperforms SSO-Global, indicating that the GPU degrades performance relative to using the CPU when executing SuperSmoother.

Comparing the performance of the single pass variant kernels, we find that SSSP-Coalesced outperforms the global memory baseline, SSSP-Global and the shared-memory kernel SSSP-Thread on both datasets. Since SSSP-Coalesced is not limited by shared memory, and can execute the algorithm on any light curve size, N_t, it is preferable to select this GPU kernel over SSSP-Global and SSSP-Thread. We find that SSSP-Coalesced outperforms the parallel CPU implementations on both small and large datasets, achieving a speedup of 3.96× and 4.47× on the *ObjSmall* and *ObjLarge* datasets, respectively.

5.10. Kernel Time Breakdown

To understand the major bottlenecks in the SuperSmoother GPU implementations, we profile several kernels using the Nvidia Visual Profiler and report the time spent in major components of the algorithms. While the main SuperSmoother kernel is straightforward to measure, the Thrust sorting algorithm calls several independent kernels. Consequently, for clarity, when reporting the time spent sorting on the datasets that only require executing a single batch, we report the duration of time between when the first and last sorting kernel started and ended execution, respectively. In these experiments, since the standard deviation of the time trials is low, we only execute a single time trial.

Table 3 shows the percentage of time spent on different tasks on the two single object datasets comparing the two kernels for SuperSmoother and one kernel for SuperSmootherSP. Recall from Figure 8 that the SSSP-Coalesced SuperSmootherSP kernel performs best, so we do not examine the other kernels. Furthermore, as shown in the experiment illustrated in Figure 8, the SSO-Thread kernel can be executed on the *ObjSmall* dataset but not the *ObjLarge* dataset due to insufficient shared memory to process the larger light curve. From Table 3, we find that on the *ObjSmall* dataset, the SuperSmoother kernels, SSO-Global and SSO-Thread, both require the greatest fraction of time, and the back-to-back sort requires the second largest fraction of time. However, on SuperSmootherSP, the SSSP-Coalesced kernel requires less time than the sorting. Comparing SSSP-Coalesced to SSO-Thread and SSO-Global, we find

that the single-pass variant and the associated coalesced memory optimization is able to reduce the response time such that the smoothing function is no longer the bottleneck. Since the sorting function is state-of-the-art from the Thrust library, there is very little that can be optimized to further improve performance of SuperSmootherSP. We observe similar performance trends on the *ObjLarge* dataset.

Table 4 shows the same as Table 3, but on the *Stripe82* dataset using batch mode. On the *Stripe82* dataset, the SSO-Cascade kernel requires a majority of the response time, but sorting, freeing memory and the tasks outlined by "other" require a significant amount of work. The *Stripe82* dataset contains 136 RR-Lyrae stars. We find that the SSO-Cascade mode called the SSO-Thread kernel for 129 objects, and called the SSO-Global kernel for 7 objects. This further demonstrates that the cascade mode can be highly advantageous by executing the faster shared-memory kernel on smaller light curves rather than simply relying on the SSO-Global kernel that is slower but does not have any light curve size limitations.

We also find that compared to the single object experiments in Table 3, the "other" category in Table 4 requires a significant amount of time when executing SuperSmootherSP. Because we know that the light curves are small in the *Stripe82* dataset, it is likely that kernel invocations incur significant overhead in this experiment.

In both Tables 3 and 4, we find that freeing memory requires a non-negligible fraction of the total time. When processing batches of objects, we allocate only the memory required to store the data for that object and free the memory when we finish processing the object. While we could over-allocate memory and free it once, this would incur more kernel executions which would increase overhead. Furthermore, since this software will be used in production-grade settings, particularly for community LSST event brokers, we elect to free the memory after use which may prevent future memory leak bugs in our software as it evolves.

Overall, we find that SuperSmootherSP requires a minority fraction of the overall response time when processing a single object or a batch of objects. Since the other components of the algorithm cannot be optimized, this indicates that our GPU implementation is highly efficient.

5.11. Processing Catalogs of Objects in Batch Mode

In this section, we execute SuperSmoother on two real-world catalogs of astronomical data, *Stripe82* and *TESS*. From Figures 7 and 8, we observe that the original SuperSmoother algorithm should use cascade mode, and that the single pass variant should use the global memory kernel with the coalesced memory access optimization. In this section, we execute the algorithms with these optimizations. We execute SuperSmoother such that we use a value of N_f that captures the peaks in the periodograms based on typical science cases for these datasets as described in Section 5.3.

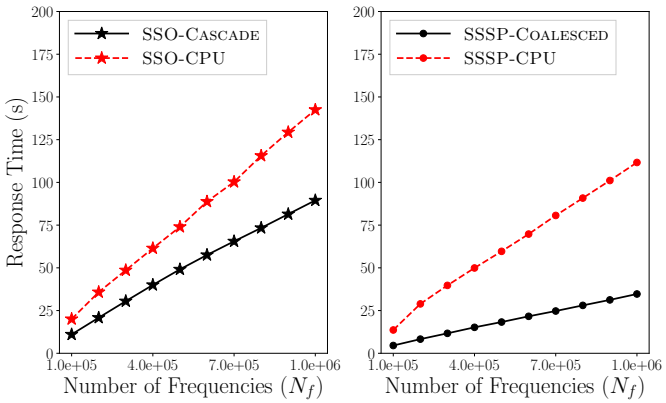

Figure 9: Response time as a function of N_f on the *Stripe82* dataset. *Left:* SuperSmoother, and *right:* SuperSmootherSP. Frequencies are searched in the range [0.1, 10.0) day^{-1} using $N_f = [10^5, 10^6]$. Experiments performed on PlatformA with FP64.

Figure 9 plots the response time for SuperSmoother (*Left*) and SuperSmootherSP (*Right*) as a function of the number of frequencies searched, N_f, on the *Stripe82* dataset. We find that when executing SuperSmoother, the GPU achieves a speedup over the CPU in the range 1.51–1.82×. On SuperSmootherSP, the speedup ranges from 3.00–3.49×. The speedup difference between SuperSmoother and SuperSmootherSP is interesting. Because the CPU processes a single frequency at a time, it can store the light curve of an object entirely in L3 cache and then reuse the data multiple times, as the algorithm performs several scans over the data. In contrast, the SuperSmootherSP algorithm only makes a single scan over the input dataset at each tested frequency; therefore, temporal locality on the CPU is exploited to a lesser degree when using the single-pass variant compared to the original algorithm. Consequently, this results in a larger speedup when using the GPU.

Figure 10 plots the response time as a function of N_f on the *TESS* dataset. We find that when comparing the CPU and GPU SuperSmoother algorithms (left panel), the GPU algorithm performs worse than the CPU algorithm, where the GPU yields a slowdown between 0.74–0.82×. In contrast to Figure 9 (left panel) on the *Stripe82* dataset, the number of data points (N_t) in each light curve from *TESS* is greater on average than in *Stripe82*. Consequently, the SSO-Cascade algorithm requires using the SSO-Global kernel more frequently than the SSO-Thread kernel, and as was demonstrated in Figure 8 (right panel), the CPU outperforms the GPU on large light curves due to high cache reuse that is possible on the CPU.

Figure 10 (right panel) illustrates the performance of SuperSmootherSP on the *TESS* dataset. We find that the speedup of the GPU over the CPU algorithm is in the range 1.78–2.91×, demonstrating that the single-pass

Table 3: The percentage of time spent on different tasks, comparing the SuperSmoother kernels: SSO-Global, SSO-Thread and the SuperSmootherSP kernel: SSSP-Coalesced. The "other" column refers to all other components not captured by the main tasks, such as: data transfers and memory allocation, kernel execution overhead, and GPU synchronization. The largest percentage is shown in bold face. Percentages may not add up to 1 due to rounding errors. This table highlights times on the single object datasets, *ObjSmall* and *ObjLarge*.

Component (line in Algorithm 1)	Dataset: *ObjSmall*			Dataset: *ObjLarge*	
	SSO-Global	SSO-Thread	SSSP-Coalesced	SSO-Global	SSSP-Coalesced
Main Smoother Kernel (line 8)	**78.4**	**63.6**	12.0	**90.0**	27.2
Sorting (line 6)	11.0	19.0	**45.8**	7.67	**53.8**
Map argsort (line 7)	0.30	0.60	6.75	0.320	6.25
Compute phased time (line 4)	0.30	0.60	1.50	0.241	1.70
Free memory	7.0	12.7	30.4	0.126	0.766
Other	3.0	3.50	3.55	1.65	10.28

Table 4: The same as Table 3 on the *Stripe82* dataset, demonstrating batch modes on both SuperSmoother and SuperSmootherSP.

Component	Dataset: *Stripe82*	
	SSO-Cascade	SSSP-Coalesced
Main Smoother Kernel	**53.0**	11.5
Sorting	13.9	24.0
Map argsort	0.556	5.47
Compute phased time	0.546	1.48
Free memory	10.7	21.1
Other	21.3	**36.5**

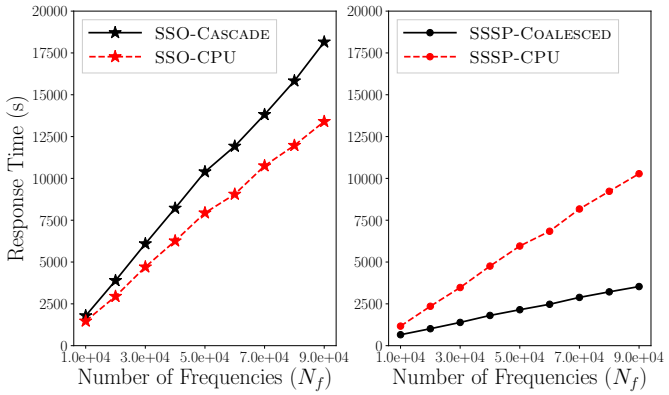

Figure 10: Response time as a function of N_f on the *TESS* dataset. *Left:* SuperSmoother, and *right:* SuperSmootherSP. Frequencies are searched in the range [0.1, 10.0) day^{-1} using $N_f = [10^4, 9 \times 10^4]$. Experiments performed on PlatformA with FP64.

variant is much more efficient on the GPU than the CPU.

We examine the performance of the batch modes as a function of N_t to examine the effect of this parameter on performance. Since the algorithm components do not all have a linear time complexity as a function of N_t, such as the sorting step, it is useful to examine how algorithmic performance may degrade with N_t. To carry out this experiment, we selected all objects in the *TESS* dataset with $N_t \geq 500$ which yields 3,150 total objects. We then created input datasets with these objects, where all objects in each dataset have $N_t = 50, 100, \ldots, 500$ observations.

Figure 11 plots the response time as a function of the input size (N_t) on the *TESS* dataset for the 3,150 objects described above. Similarly to the results in Figure 10, we find that the GPU performs worse than the CPU algorithm

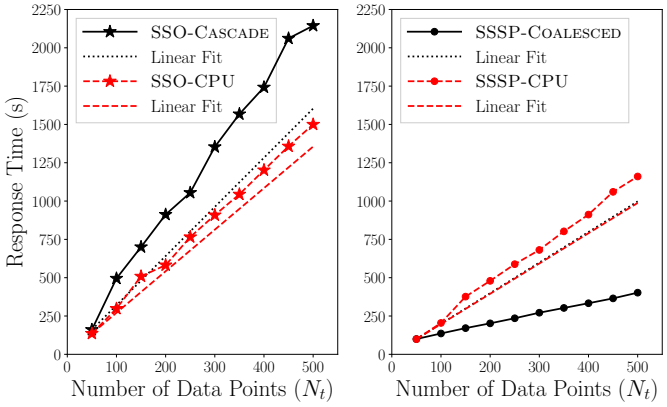

Figure 11: Response time as a function of N_t on the *TESS* dataset. *Left:* SuperSmoother, and *right:* SuperSmootherSP. Frequencies are searched in the range [0.1, 10.0) day^{-1} using $N_f = 5 \times 10^4$. The dataset consists of 3,150 objects from the *TESS* dataset that have $\geq$500 observations per object. The linear fit refers to extrapolating the response time at $N_t = 50$ for the CPU and GPU algorithms. Experiments performed on PlatformA with FP64.

for SuperSmoother (Figure 11, left panel), whereas the GPU outperforms the CPU on SuperSmootherSP (Figure 11, right panel). The linear fits shown assumes a linear extrapolation as a function of N_t, as extrapolated from the response time for the CPU and GPU algorithms at $N_t = 50$. We find that the original algorithm degrades with a superlinear profile (Figure 11, left panel), and interestingly, the SuperSmootherSP algorithm exhibits sublinear performance degradation (Figure 11, right panel).

The performance degradation profiles demonstrate why the original algorithm, SuperSmoother, is inefficient on the GPU compared to the CPU. The GPU experiences significant superlinear performance degradation, whereas the CPU algorithm degrades gracefully, exhibiting mild superlinear performance degradation. In contrast, the SuperSmootherSP algorithm exhibits sublinear performance degradation on the GPU, thus scaling to larger numbers of data points without a severe performance penalty, whereas the CPU algorithm exhibits mild superlinear performance degradation.

We omit showing results for the *LINEAR* dataset in this section, as PlatformA has insufficient main memory

Table 5: The ratio of the response time (s) (FP64/FP32) of SuperSmoother and SuperSmootherSP. The frequency grid uses the same default parameters as shown in Table 2. Experiments performed on PlatformB using four GPUs.

Dataset	SSO-Cascade Ratio: FP64/FP32	SSSP-Coalesced Ratio: FP64/FP32
Stripe82	2.92	1.89
TESS	1.25	1.73
LINEAR	1.15	2.05

to store the periodograms. We examine that dataset in Sections 5.12 and 5.13.

5.12. Performance and Accuracy of FP32 vs. FP64

Our software allows the user to select whether they would like to execute the algorithm using FP32 or FP64 precision. In this section, we examine the performance of executing SuperSmoother and SuperSmootherSP with FP32 and FP64, in addition to observing any potential discrepancy in derived solutions. We execute this experiment on PlatformB using four GPUs (we will present results for multi-GPU scalability experiments in Section 5.13). Recall from Section 5.1 that the GPUs in this platform have limited support for FP64; therefore, the response time ratio of FP64 to FP32 is expected to be high. Consequently, for GPUs with greater support for FP64, the ratios will be lower, and thus, the reported ratios represent an approximate upper bound on the performance degradation yielded by using FP64 over FP32.

Table 5 shows the response time ratios of executing SuperSmoother and SuperSmootherSP using FP32 and FP64. Beginning with SuperSmootherSP, we find that the response time ratios of FP64/FP32 across the three datasets are in the range 1.73–2.05, indicating that the algorithm has a similar performance penalty across the desperate datasets (e.g., recall that N_t is smaller on average for *Stripe82* than *TESS*). In contrast, SuperSmoother has response time ratios in the range 1.15-2.92 across the three datasets. The shared-memory kernel is exploited to a greater extent on *Stripe82* because N_t is smaller on average compared to *TESS* and *LINEAR*. When executing *Stripe82* with FP32, SSO-Cascade uses the shared-memory kernel to process all objects, but requires launching the slower global memory kernel to process 7 of 136 objects when using FP64. This explains why using FP64 causes a significant slowdown on *Stripe82*. In contrast, since most objects cannot be processed with the shared-memory kernel on the *TESS* and *LINEAR* datasets, the penalty for using FP64 is much smaller.

In summary, while FP64 has a larger memory footprint and increases the number of memory accesses compared to using FP32, we do not find that using FP64 yields a factor of 2 increase in the response time compared to FP32 in all instances. The slowdown is data dependent when using SuperSmoother, and is much more consistent across datasets with SuperSmootherSP. To reiterate, since the GPUs in this platform have minimal hardware dedicated to FP64 arithmetic, it is likely that these ratios will be lower on other GPUs, such as those designed for scientific computing.

Figure 12 plots the derived periods using FP32 compared to FP64 on *Stripe82* and *TESS* for (a)–(b) SuperSmoother and (c)–(d) SuperSmootherSP. Periods along the diagonal line indicate a perfect match. The percentage of periods that match, defined as being within 3% of each other are as follows: (a) 94.1%, (b) 93.0%, (c) 94.9%, and (d) 95.4%. From the *TESS* dataset (Figure 12(b) and (d)), we clearly observe that some of the mismatched periods are aliases, where the period generated by FP64 is either half or double the period generated when using FP32. This shows that it may be preferable to employ FP64 over FP32 when executing the algorithm. However, FP32 can capture the same period as that derived by FP64 in $\geq$93% of instances.

For consistency, our software allows for selecting either FP32 or FP64, and we do not employ mixed floating point precision. Since there may be cases where a user will want to hard code certain tasks with FP32 or FP64, such as using FP64 in the folding step where the time could be unbounded, and using FP32 for the rest of the tasks, we make our source code publicly available so this and other modifications can be carried out.

5.13. Multi-GPU Scalability

To assess the scalability of our algorithms on multiple GPUs, we use the *LINEAR* dataset, which is the largest catalog that we consider in this paper. There is insufficient main memory in PlatformA to store the periodograms for each object in the catalog using $N_f = 222,000$ (the minimum number of frequencies to search such that we do not miss peaks in the periodogram). Consequently, this experiment is performed on PlatformB, as it has 4 GPUs and sufficient main memory. The *LINEAR* dataset contains a large number of objects, and we estimated that to perform three time trials for each experiment would have required roughly a month of computation time. Therefore, in this experiment, we only perform a single time trial for each data point in the plots to limit the time required to run these experiments. However, we note that the standard deviation is very low between time trials[5]. Consequently, a single time trial is sufficient for this experiment.

Figure 13 plots the speedup as a function of N_{GPU} on the *LINEAR* dataset for SuperSmoother (left) and SuperSmootherSP (right). We find that the multi-GPU SuperSmoother implementation achieves a near-perfect speedup. We attribute this to the following factors: (1) The algorithm is compute-bound and there is little contention for PCIe bandwidth between the GPUs; therefore,

[5]We executed $N_f = 10^5$ frequencies for three time trials using $N_{GPU} = 4$ where the time measurements in seconds are as follows: 19,090.2, 19,078.8, 19,078.1, yielding a standard deviation of $\sigma = 5.55$.

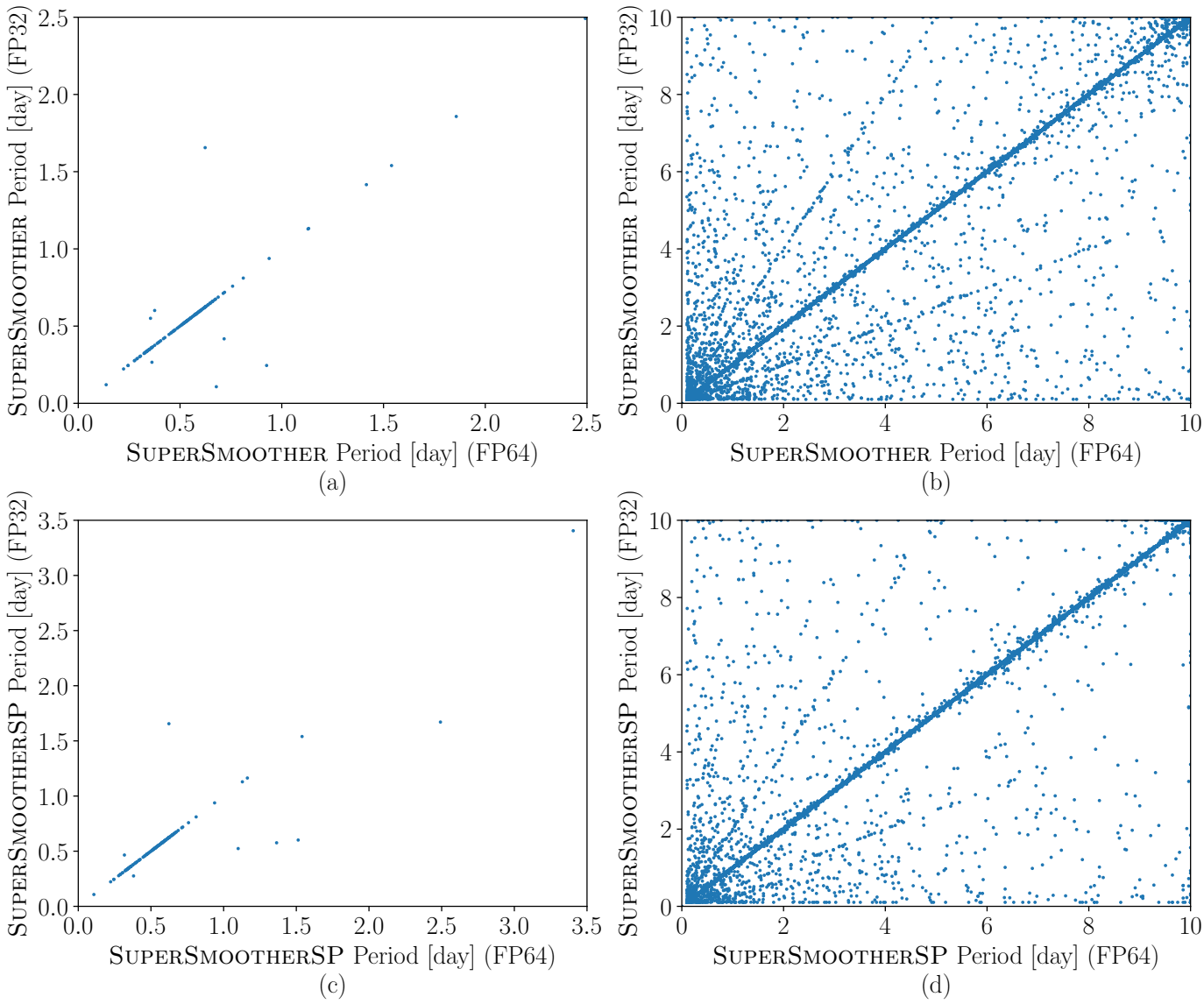

Figure 12: Comparing the periods derived using FP32 and FP64, for SuperSmoother on (a) *Stripe82*, and (b) *TESS*, and SuperSmootherSP on (c) *Stripe82*, and (d) *TESS*. Periods that fall on the diagonal line indicate agreement between the FP32 and FP64 solutions. The frequency grid uses the same default parameters as shown in Table 2.

the host-device interconnect does not limit performance; (2) An object is computed by a single GPU, and the execution time for each object varies with N_t. Thus, the probability that two or more GPUs require using the PCIe interconnect at the same time is low. (3) We assign objects to the GPUs using dynamic scheduling in OpenMP. Therefore, there is minimal load imbalance at the end of the computation between GPUs. In summary, multiple GPUs can be employed to significantly reduce the computation time and there is no source of performance degradation due to multi-GPU parallelization.

On SuperSmootherSP (Figure 13, right panel), we observe good scalability (a speedup of 3.61× on 4 GPUs), but the speedup is lower than SuperSmoother. This is because the algorithm performs less GPU computation, so there is more contention for host-side resources and the PCIe interconnect.

Comparing the response times used to derive Figure 13, SuperSmootherSP achieves a speedup of 10.38–11.46× over SuperSmoother.

5.14. Performance Comparison with Lomb-Scargle

In Section 5.5, we showed that on the *LINEAR* dataset, SuperSmoother and the single-pass variant are less likely than Lomb-Scargle to produce a 24- or 48-hour alias solution. Furthermore, in Section 5.6, we showed that the fraction of period matches for SuperSmoother and SuperSmootherSP is higher than for Lomb-Scargle on

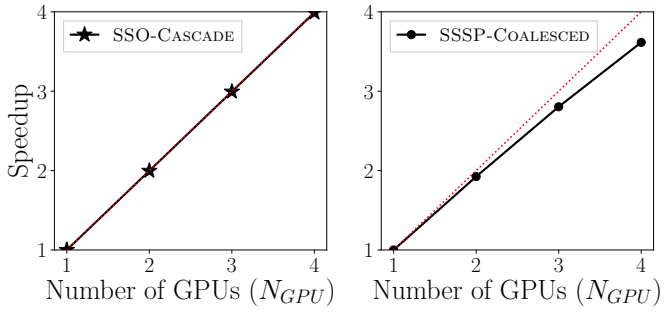

Figure 13: The response time as a function of the number of GPUs (N_{GPU}) on the *LINEAR* dataset for $N_f = 222,000$, and a frequency grid in the range [0.01, 10.0) day^{-1}. *Left:* SSO-Cascade, *Right:* SSSP-Coalesced. In each plot, a perfect speedup is indicated by the dashed red line. Experiments performed on PlatformB with FP32.

Table 6: Response time (s) comparison of the SuperSmoother algorithms compared to Lomb-Scargle. The frequency grid uses the same default parameters as shown in Table 2. *Stripe82* and *TESS* are executed on PlatformA using FP64, whereas *LINEAR* is executed on PlatformB using FP32.

Dataset	SSO-Cascade	SSSP-Coalesced	Lomb-Scargle
Stripe82	30.16	11.22	0.27
TESS	5,131.21	1,088.62	160.20
LINEAR	167,731.96	14,634.25	660.84

the *Stripe82* dataset. In this section, we compare SuperSmoother to Lomb-Scargle. We use the GPU-accelerated algorithm described in our prior work (Gowanlock et al., 2021). The GPU-accelerated Lomb-Scargle algorithm was shown to significantly outperform the parallel CPU algorithm, achieving speedups $> 100\times$ on several experimental scenarios (Gowanlock et al., 2021).

Table 6 shows the response time of deriving the periods using one GPU for the three datasets using the default grid search parameters in Table 2. We find that Lomb-Scargle achieves a speedup over SuperSmoother and SuperSmootherSP in the range of 32.03-253.82$\times$, and 6.80-41.56$\times$, respectively. The Lomb-Scargle algorithm is very well-suited to execution on the GPU. Compared to SuperSmoother, it does not have the same pre-processing requirements, and it also has a very small memory footprint. Lomb-Scargle is useful for period searches where the light curve can be best fit by a sinusoid; however, it is not suitable for light curves of unknown shape. Thus, there is an accuracy vs. performance trade-off between selecting Lomb-Scargle compared to an algorithm that is more robust to non-sinusoidal light curves.

To address the abovementioned trade-off, in Section 6 we outline two future research directions, one that takes advantage of the pre-processing step required of SuperSmoother and similar methods (described in Section 1), and the other proposes to combine Lomb-Scargle with SuperSmoother to limit the frequency search space.

6. Discussion & Conclusions

In this paper, we proposed the first GPU-accelerated SuperSmoother algorithm. Furthermore, we proposed a single-pass variant of SuperSmoother that uses generalized validation instead of the cross-validation approach used in the original algorithm. We find that the single-pass variant is largely efficient on the GPU, due to requiring fewer scans over the input dataset, and our optimization that reorders data access patterns that exploit coalesced memory accesses to global memory. We also find that the single-pass variant does not provide substantial performance gains over the original algorithm on the CPU due to the high cache reuse that is possible when making several scans over the time series when using the original algorithm. Therefore, for period finding purposes, it may be preferable to use the original algorithm over the single-pass algorithm when using CPU hardware.

SuperSmoother and our proposed single-pass variant are expensive algorithms, even with the significant parallelism offered by the GPU's architecture. To reduce the overhead of executing SuperSmoother, a future research direction is to employ a *search-and-refine* approach. Here, we will use an inexpensive algorithm, such as Lomb-Scargle, to *search* for potential peaks in the periodogram. Then, we will *refine* the search around the peaks using SuperSmoother. This will significantly reduce the cost of searching large frequency spaces with SuperSmoother. We can leverage our prior work in this area using our GPU-accelerated Lomb-Scargle algorithm (Gowanlock et al., 2021). This method would be similar to Shin and Byun (2004), which used a coarse-grained period search followed by a fine-grained search.

We found that the pre-processing requirements, such as folding the input time series by a given searched period and then sorting this time series requires non-negligible time (recall that without doing this, the frequencies cannot be searched in parallel). For example, sorting can require the largest fraction of the total time; see Table 3. Thus, to leverage this pre-processed data to a greater degree, another future work direction is to execute several period finding algorithms after the data has been pre-processed. This will make better use of the data, as it can be reused across multiple period finders. Using multiple algorithms are likely to yield greater confidence in derived period solutions.

SuperSmoother has several performance drawbacks when executed on modern GPU hardware, such as the requirement of sorting the input dataset for each searched frequency. Consequently, another future research direction is to use a hybrid approach that harnesses the capabilities of both the CPU and GPU to concurrently compute the periods of batches of objects.

Acknowledgment

This work has been supported in part by the Arizona Board of Regents, Regents' Innovation Fund. We thank Will Oldroyd for supplying us with the TESS exoplanet time series data. We thank the referee, Michael Coughlin, for his thorough review and feedback on our manuscript.

Appendix A. Open Source Code

The source code is publicly available at `https://github.com/mgowanlock/gpu_supersmoother`. Our code assumes an evenly spaced frequency grid in the range $[f_{min}, f_{max})$. The algorithm uses regular oscillating (not angular) frequencies, where the frequency is given by $f = 1/p$, where p is the period. Note that the units output by the code are equivalent to the units in the input dataset file (e.g., typically the time is given as the Julian date).

The same source code is used for both C and Python interfaces. The C version contains the code used to produce the results in this paper. The Python interface calls the C code using shared libraries. These C shared libraries should be compiled by the user so that they can input their machine-specific parameters via the Makefile. However, we also provide compiled shared libraries if the user does not want to compile the code. To reduce confusion with all of the parameters and optimizations used in the paper, the Python interface selects common default parameters for the user. In particular, in this version of the code, SuperSmoother uses cascade mode, and SuperSmootherSP uses the global memory kernel with the coalesced memory optimization. Additional default parameters are selected and more information can be found in the code repository. The Python interface will take as input the object IDs, time, magnitude, magnitude errors, frequency search ranges, and the selection of SuperSmoother algorithm, and will return for each object the object ID, the period with the greatest power for the object, and the periodogram.

References

Bell, N., Hoberock, J., 2012. Thrust: A productivity-oriented library for CUDA, in: GPU computing gems Jade edition. Elsevier, pp. 359–371.

Burdge, K.B., Coughlin, M.W., Fuller, J., Kupfer, T., Bellm, E.C., Bildsten, L., Graham, M.J., Kaplan, D.L., Roestel, J.v., Dekany, R.G., Duev, D.A., Feeney, M., Giomi, M., Helou, G., Kaye, S., Laher, R.R., Mahabal, A.A., Masci, F.J., Riddle, R., Shupe, D.L., Soumagnac, M.T., Smith, R.M., Szkody, P., Walters, R., Kulkarni, S.R., Prince, T.A., 2019. General relativistic orbital decay in a seven-minute-orbital-period eclipsing binary system. Nature 571, 528–531. doi:10.1038/s41586-019-1403-0, arXiv:1907.11291.

Chambers, K.C., Magnier, E.A., Metcalfe, N., Flewelling, H.A., Huber, M.E., Waters, C.Z., Denneau, L., Draper, P.W., Farrow, D., Finkbeiner, D.P., Holmberg, C., Koppenhoefer, J., Price, P.A., Rest, A., Saglia, R.P., Schlafly, E.F., Smartt, S.J., Sweeney, W., Wainscoat, R.J., Burgett, W.S., Chastel, S., Grav, T., Heasley, J.N., Hodapp, K.W., Jedicke, R., Kaiser, N., Kudritzki, R.P., Luppino, G.A., Lupton, R.H., Monet, D.G., Morgan, J.S., Onaka, P.M., Shiao, B., Stubbs, C.W., Tonry, J.L., White, R., Bañados, E., Bell, E.F., Bender, R., Bernard, E.J., Boegner, M., Boffi, F., Botticella, M.T., Calamida, A., Casertano, S., Chen, W.P., Chen, X., Cole, S., Deacon, N., Frenk, C., Fitzsimmons, A., Gezari, S., Gibbs, V., Goessl, C., Goggia, T., Gourgue, R., Goldman, B., Grant, P., Grebel, E.K., Hambly, N.C., Hasinger, G., Heavens, A.F., Heckman, T.M., Henderson, R., Henning, T., Holman, M., Hopp, U., Ip, W.H., Isani, S., Jackson, M., Keyes, C.D., Koekemoer, A.M., Kotak, R., Le, D., Liska, D., Long, K.S., Lucey, J.R., Liu, M., Martin, N.F., Masci, G., McLean, B., Mindel, E., Misra, P., Morganson, E., Murphy, D.N.A., Obaika, A., Narayan, G., Nieto-Santisteban, M.A., Norberg, P., Peacock, J.A., Pier, E.A., Postman, M., Primak, N., Rae, C., Rai, A., Riess, A., Riffeser, A., Rix, H.W., Röser, S., Russel, R., Rutz, L., Schilbach, E., Schultz, A.S.B., Scolnic, D., Strolger, L., Szalay, A., Seitz, S., Small, E., Smith, K.W., Soderblom, D.R., Taylor, P., Thomson, R., Taylor, A.N., Thakar, A.R., Thiel, J., Thilker, D., Unger, D., Urata, Y., Valenti, J., Wagner, J., Walder, T., Walter, F., Watters, S.P., Werner, S., Wood-Vasey, W.M., Wyse, R., 2016. The Pan-STARRS1 Surveys. arXiv e-prints , arXiv:1612.05560arXiv:1612.05560.

Coughlin, M.W., Burdge, K., Duev, D.A., Katz, M.L., van Roestel, J., Drake, A., Graham, M.J., Hillenbrand, L., Mahabal, A.A., Masci, F.J., Mróz, P., Prince, T.A., Yao, Y., Bellm, E.C., Burruss, R., Dekany, R., Jaodand, A., Kaplan, D.L., Kupfer, T., Laher, R.R., Riddle, R., Rigault, M., Rodriguez, H., Rusholme, B., Zolkower, J., 2021. The ztf source classification project–ii. periodicity and variability processing metrics. Monthly Notices of the Royal Astronomical Society 505, 2954–2965.

Drake, A.J., Graham, M.J., Djorgovski, S.G., Catelan, M., Mahabal, A.A., Torrealba, G., García-Álvarez, D., Donalek, C., Prieto, J.L., Williams, R., Larson, S., sen, E.C., Belokurov, V., Koposov, S.E., Beshore, E., Boattini, A., Gibbs, A., Hill, R., Kowalski, R., Johnson, J., Shelly, F., 2014. THE CATALINA SURVEYS PERIODIC VARIABLE STAR CATALOG. The Astrophysical Journal Supplement Series 213, 9. URL: https://doi.org/10.1088/0067-0049/213/1/9, doi:10.1088/0067-0049/213/1/9.

Dworetsky, M., 1983. A period-finding method for sparse randomly spaced observations or "How long is a piece of string?". Monthly Notices of the Royal Astronomical Society 203, 917–924.

Friedman, J.H., 1984. A variable span scatterplot smoother. Laboratory for Computational Statistics, Stanford University Technical Report No. 5. .

Gowanlock, M., Kramer, D., Trilling, D., Butler, N., Donnelly, B., 2021. Fast period searches using the Lomb–Scargle algorithm on Graphics Processing Units for large datasets and real-time applications. Astronomy and Computing 36, 100472. URL: https://www.sciencedirect.com/science/article/pii/S2213133721000263, doi:https://doi.org/10.1016/j.ascom.2021.100472.

Graham, M.J., Drake, A.J., Djorgovski, S.G., Mahabal, A.A., Donalek, C., 2013. Using conditional entropy to identify periodicity. Monthly Notices of the Royal Astronomical Society 434, 2629–2635. doi:10.1093/mnras/stt1206, arXiv:1306.6664.

Graham, M.J., Kulkarni, S.R., Bellm, E.C., Adams, S.M., Barbarino, C., Blagorodnova, N., Bodewits, D., Bolin, B., Brady, P.R., Cenko, S.B., Chang, C.K., Coughlin, M.W., De, K., Eadie, G., Farnham, T.L., Feindt, U., Franckowiak, A., Fremling, C., Gezari, S., Ghosh, S., Goldstein, D.A., Golkhou, V.Z., Goobar, A., Ho, A.Y.Q., Huppenkothen, D., Ivezić, Ž., Jones, R.L., Juric, M., Kaplan, D.L., Kasliwal, M.M., Kelley, M.S.P., Kupfer, T., Lee, C.D., Lin, H.W., Lunnan, R., Mahabal, A.A., Miller, A.A., Ngeow, C.C., Nugent, P., Ofek, E.O., Prince, T.A., Rauch, L., van Roestel, J., Schulze, S., Singer, L.P., Sollerman, J., Taddia, F., Yan, L., Ye, Q.Z., Yu, P.C., Barlow, T., Bauer, J., Beck, R., Belicki, J., Biswas, R., Brinnel, V., Brooke, T., Bue, B., Bulla, M., Burruss, R., Connolly, A., Cromer, J., Cunningham, V., Dekany, R., Delacroix, A., Desai, V., Duev, D.A., Feeney, M., Flynn, D., Frederick, S., Gal-Yam, A., Giomi, M., Groom, S., Hacopians, E., Hale, D., Helou, G., Henning, J., Hover, D., Hillenbrand, L.A., Howell, J., Hung, T., Imel, D., Ip, W.H., Jackson, E., Kaspi, S., Kaye, S., Kowalski, M., Kramer, E., Kuhn,

M., Landry, W., Laher, R.R., Mao, P., Masci, F.J., Monkewitz, S., Murphy, P., Nordin, J., Patterson, M.T., Penprase, B., Porter, M., Rebbapragada, U., Reiley, D., Riddle, R., Rigault, M., Rodriguez, H., Rusholme, B., van Santen, J., Shupe, D.L., Smith, R.M., Soumagnac, M.T., Stein, R., Surace, J., Szkody, P., Terek, S., Sistine, A.V., van Velzen, S., Vestrand, W.T., Walters, R., Ward, C., Zhang, C., Zolkower, J., 2019. The Zwicky Transient Facility: Science Objectives. Publications of the Astronomical Society of the Pacific 131, 078001. URL: https://doi.org/10.1088/1538-3873/ab006c, doi:10.1088/1538-3873/ab006c.

Huijse, P., Estevez, P.A., Protopapas, P., Zegers, P., Principe, J.C., 2012. An information theoretic algorithm for finding periodicities in stellar light curves. IEEE Transactions on Signal Processing 60, 5135–5145.

Katz, M.L., Cooper, O.R., Coughlin, M.W., Burdge, K.B., Breivik, K., Larson, S.L., 2021. GPU-accelerated periodic source identification in large-scale surveys: measuring P and P. Monthly Notices of the Royal Astronomical Society 503, 2665–2675.

Kovács, G., Zucker, S., Mazeh, T., 2002. A box-fitting algorithm in the search for periodic transits. Astronomy & Astrophysics 391, 369–377. doi:10.1051/0004-6361:20020802, arXiv:astro-ph/0206099.

Lomb, N.R., 1976. Least-Squares Frequency Analysis of Unequally Spaced Data. Astrophysics and Space Science 39, 447–462. doi:10.1007/BF00648343.

LSST Science Collaboration, 2009. LSST Science Book, Version 2.0. arXiv e-prints , arXiv:0912.0201arXiv:0912.0201.

McNeill, A., Mommert, M., Trilling, D.E., Llama, J., Skiff, B., 2019. Asteroid Photometry from the Transiting Exoplanet Survey Satellite: A Pilot Study. The Astrophysical Journal Supplement Series 245, 29. doi:10.3847/1538-4365/ab5223, arXiv:1911.01495.

McNeill, A., et al., 2021. Manuscript in preparation .

McWilliam, A., 2011. RR Lyrae Stars, Metal-Poor Stars, and the Galaxy. arXiv preprint arXiv:1109.1324 .

Palmer, D.M., 2009. A fast chi-squared technique for period search of irregularly sampled data. The Astrophysical Journal 695, 496.

Press, W.H., Teukolsky, S.A., Flannery, B.P., Vetterling, W.T., 1992. Numerical recipes in Fortran 77: volume 1 of Fortran numerical recipes: the art of scientific computing. Cambridge university press.

Price-Whelan, A.M., et al., 2018. The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package. The Astronomical Journal 156, 123. URL: https://doi.org/10.3847/1538-3881/aabc4f, doi:10.3847/1538-3881/aabc4f.

Reimann, J.D., 1994. Frequency estimation using unequally-spaced astronomical data. Ph.D. thesis. Citeseer.

Richards, J.W., Starr, D.L., Butler, N.R., Bloom, J.S., Brewer, J.M., Crellin-Quick, A., Higgins, J., Kennedy, R., Rischard, M., 2011. On Machine-learned Classification of Variable Stars with Sparse and Noisy Time-series Data. The Astrophysical Journal 733, 10. doi:10.1088/0004-637X/733/1/10, arXiv:1101.1959.

Scargle, J.D., 1982. Studies in astronomical time series analysis. II. Statistical aspects of spectral analysis of unevenly spaced data. The Astrophysical Journal 263, 835–853. doi:10.1086/160554.

Schwarzenberg-Czerny, A., 1989. On the advantage of using analysis of variance for period search. Monthly Notices of the Royal Astronomical Society 241, 153–165.

Schwarzenberg-Czerny, A., 1996. Fast and statistically optimal period search in uneven sampled observations. The Astrophysical Journal Letters 460, L107.

Sesar, B., Ivezić, Ž., Grammer, S.H., Morgan, D.P., Becker, A.C., Jurić, M., De Lee, N., Annis, J., Beers, T.C., Fan, X., Lupton, R.H., Gunn, J.E., Knapp, G.R., Jiang, L., Jester, S., Johnston, D.E., Lampeitl, H., 2010. Light Curve Templates and Galactic Distribution of RR Lyrae Stars from Sloan Digital Sky Survey Stripe 82. The Astrophysical Journal 708, 717–741. doi:10.1088/0004-637X/708/1/717, arXiv:0910.4611.

Shappee, B.J., Prieto, J.L., Grupe, D., Kochanek, C.S., Stanek, K.Z., Rosa, G.D., Mathur, S., Zu, Y., Peterson, B.M., Pogge, R.W., Komossa, S., Im, M., Jencson, J., Holoien, T.S., Basu, U., Beacom, J.F., Szczygieł, D.M., Brimacombe, J., Adams, S., Campillay, A., Choi, C., Contreras, C., Dietrich, M., Dubberley, M., Elphick, M., Foale, S., Giustini, M., Gonzalez, C., Hawkins, E., Howell, D.A., Hsiao, E.Y., Koss, M., Leighly, K.M., Morrell, N., Mudd, D., Mullins, D., Nugent, J.M., Parrent, J., Phillips, M.M., Pojmanski, G., Rosing, W., Ross, R., Sand, D., Terndrup, D.M., Valenti, S., Walker, Z., Yoon, Y., 2014. The Man Behind the Curtain: X-rays Drive the UV through NIR Variability in the 2013 AGN Outburst in NGC 2617. The Astrophysical Journal 788, 48. URL: https://doi.org/10.1088/0004-637x/788/1/48, doi:10.1088/0004-637x/788/1/48.

Shin, M.S., Byun, Y.I., 2004. Efficient Period Search for Time Series Photometry. Journal of Korean Astronomical Society 37, 79–85. doi:10.5303/JKAS.2004.37.2.079.

Stellingwerf, R.F., 1978. Period determination using phase dispersion minimization. The Astrophysical Journal 224, 953–960. doi:10.1086/156444.

Tonry, J.L., Denneau, L., Heinze, A.N., Stalder, B., Smith, K.W., Smartt, S.J., Stubbs, C.W., Weiland, H.J., Rest, A., 2018. ATLAS: A High-cadence All-sky Survey System. Publications of the Astronomical Society of the Pacific 130, 064505. URL: https://doi.org/10.1088/1538-3873/aabadf, doi:10.1088/1538-3873/aabadf.

Townsend, R.H.D., 2010. Fast Calculation of the Lomb-Scargle Periodogram Using Graphics Processing Units. The Astrophysical Journal Supplement Series 191, 247–253. doi:10.1088/0067-0049/191/2/247, arXiv:1007.1658.

van Roestel, J., Duev, D.A., Mahabal, A.A., Coughlin, M.W., Mróz, P., Burdge, K., Drake, A., Graham, M.J., Hillenbrand, L., Bellm, E.C., Kupfer, T., Delacroix, A., Fremling, C., Golkhou, V.Z., Hale, D., Laher, R.R., Masci, F.J., Riddle, R., Rosnet, P., Rusholme, B., Smith, R., Soumagnac, M.T., Walters, R., Prince, T.A., Kulkarni, S.R., 2021. The ztf source classification project. i. methods and infrastructure. The Astronomical Journal 161, 267.

VanderPlas, J.T., 2018. Understanding the Lomb-Scargle Periodogram. The Astrophysical Journals 236, 16. doi:10.3847/1538-4365/aab766, arXiv:1703.09824.

Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., et al., 2020. Scipy 1.0: fundamental algorithms for scientific computing in python. Nature methods 17, 261–272.

Zechmeister, M., Kürster, M., 2009. The generalised Lomb-Scargle periodogram-a new formalism for the floating-mean and Keplerian periodograms. Astronomy & Astrophysics 496, 577–584.