Fast Calculation of the Lomb-Scargle Periodogram Using Graphics Processing Units

R. H. D. Townsend

Department of Astronomy, University of Wisconsin-Madison, Sterling Hall, 475 N. Charter Street, Madison, WI 53706, USA; townsend@astro.wisc.edu

Received _____; accepted _____

ABSTRACT

I introduce a new code for fast calculation of the Lomb-Scargle periodogram, that leverages the computing power of graphics processing units (GPUs). After establishing a background to the newly emergent field of GPU computing, I discuss the code design and narrate the key parts of the source. Benchmarking calculations indicate no significant differences in accuracy compared to an equivalent CPU-based code; however, the code is up to 200 times faster than the CPU equivalent. Possible applications include spectral analysis of long photometric time series obtained by ongoing satellite missions; and Monte-Carlo simulation of periodogram statistical properties.

Subject headings: methods: data analysis — methods: numerical — techniques: photometric

1. Introduction

Astronomical time series observations are often characterized by uneven temporal sampling (e.g., due to transformation to the heliocentric frame) and/or non-uniform coverage (e.g., from day/night cycles, or radiation belt passages). This complicates the search for periodic signals, as a fast Fourier transform (FFT) algorithm cannot be employed. A variety of alternatives have been put forward, the most oft-used being the Lomb-Scargle (L-S) periodogram (Lomb 1976; Scargle 1982). However, the L-S periodogram has the drawback of a computational cost that scales as N_t^2, where N_t is the number of measurements composing the time series; this contrasts with the far-more-efficient $N_t \log_2 N_t$ scaling of the FFT algorithm popularized by Cooley & Tukey (1965).

One approach to reducing this computational cost has been proposed by Press & Rybicki (1989), based on constructing a uniformly sampled approximation to the observations via 'extirpolation' and then evaluating the FFT. The present paper introduces a different approach, not through algorithmic development but rather by leveraging the computing power of graphics processing units (GPUs) — the specialized hardware at the heart of the display subsystem in personal computers and workstations. Modern GPUs typically comprise a number of identical programmable processors, and in recent years there has been significant interest in applying these parallel-computing resources to problems across a wide range of scientific disciplines. In the following section, I give a brief history of the newly emergent field of GPU computing. Then, Section 3 reviews the formalism defining the L-S periodogram, and Section 4 presents a GPU-based implementation of this formalism. Benchmarking calculations to evaluate the accuracy and performance of the GPU code are presented in Section 5. The findings and future outlook are then discussed in Section 6, and the paper is summarized in Section 7.

2. Background to GPU Computing

2.1. Pre-2006: Initial Forays

The past decade has seen remarkable increases in the ability of computers to render complex 3-dimensional scenes at movie frame-rates. These gains have been achieved by progressively shifting the graphics pipeline — the algorithmic sequence of steps that converts a scene description into an image — from the CPU to dedicated hardware within the GPU. To address the inflexibility that can accompany such hardware acceleration, GPU vendors introduced so-called *programmable shaders*, processing units that can apply a simple sequence of transformations to input elements such as image pixels and mesh vertices. NVIDIA Corporation were the first to implement programmable shader functionality, with their GeForce 3 series of GPUs (released March 2001) offering one vertex shader and four (parallel) pixel shaders. The release in the following year of ATI Corporation's R300 series brought not only an increase in the number of shaders (up to 4 vertex and 8 pixel), but also capabilities such as floating-point arithmetic and looping constructs that laid the foundations for what ultimately would become GPU computing.

Shaders are programmed using a variety of specialized languages, such as the OpenGL Shading Language (GLSL; e.g., Rost 2006) and Microsoft's High-Level Shading Language (HLSL). The designs of these languages are strongly tied to their graphics-related purpose, and thus early attempts at GPU computing using programmable shaders had to map each calculation into a sequence of equivalent graphical operations (see, e.g., Owens et al. 2005, and references therein). In an effort to abstract away this awkward aspect, Buck et al. (2004) developed BrookGPU — a compiler and run-time implementation of the Brook stream programming language for GPU platforms. With BrookGPU, the computational resources of shaders are accessed through a *stream processing* paradigm: a well-defined series of operations (the *kernel*) are applied to each element in a typically-large homogeneous set

of data (the *stream*).

2.2. Post-2006: Modern Era

GPU computing entered its modern era in 2006, with the release of NVIDIA's *Compute Unified Device Architecture* (CUDA) — a framework for defining and managing GPU computations without the need to map them into graphical operations. CUDA-enabled devices (see Appendix A of NVIDIA 2010) are distinguished by their general-purpose unified shaders, which replace the function-specific shaders (pixel, vertex, etc.) present in earlier GPUs. These shaders are programmed using an extension to the C language, which follows the same stream-processing paradigm pioneered by BrookGPU. Since the launch of CUDA, other vendors have been quick to develop their own GPU computing offerings, most notably Advanced Micro Devices (AMD) with their *Stream* framework, and Microsoft with their *DirectCompute* interface.

Abstracting away the graphical roots of GPUs has made them accessible to a very broad audience, and GPU-based computations are now being undertaken in fields as diverse as molecular biology, medical imaging, geophysics, fluid dynamics, economics and cryptography (see Pharr 2005; Nguyen 2007). Within astronomy and astrophysics, recent applications include N-body simulations (Belleman et al. 2008), real-time radio correlation (Wayth et al. 2009), gravitational lensing (Thompson et al. 2010), adaptive-mesh hydrodynamics (Schive et al. 2010) and cosmological reionization (Aubert & Teyssier 2010).

3. The Lomb-Scargle Periodogram

This section reviews the formalism defining the Lomb-Scargle periodogram. For a time series comprising N_t measurements $X_j \equiv X(t_j)$ sampled at times t_j ($j = 1, \ldots, N_t$), *assumed*

throughout to have been scaled and shifted such that its mean is zero and its variance is unity, the normalized L-S periodogram at frequency f is

$$P_{\rm n}(f) = \frac{1}{2}\left\{ \frac{\left[\sum_j X_j \cos\omega(t_j-\tau)\right]^2}{\sum_j \cos^2\omega(t_j-\tau)} + \frac{\left[\sum_j X_j \sin\omega(t_j-\tau)\right]^2}{\sum_j \sin^2\omega(t_j-\tau)} \right\}. \quad (1)$$

Here and throughout, $\omega \equiv 2\pi f$ is the angular frequency and all summations run from $j = 1$ to $j = N_t$. The frequency-dependent time offset τ is evaluated at each ω via

$$\tan(2\omega\tau) = \frac{\sum_j \sin 2\omega t_j}{\sum_j \cos 2\omega t_j}. \quad (2)$$

As discussed by Schwarzenberg-Czerny (1998), $P_{\rm n}$ in the case of a pure-noise time series is drawn from a beta distribution. Thus, for a periodogram comprising N_f frequencies[1], the *false-alarm probability* (FAP) — that some observed peak occurs due to chance fluctuations — is

$$Q = 1 - \left[1 - \left(1 - \frac{2P_{\rm n}}{N_t}\right)^{(N_t-3)/2}\right]^{N_f}. \quad (3)$$

Implemented directly, equations (1) and (2) require two complete iterations through the time series — the first to obtain τ, and the second $P_{\rm n}$. This is computationally expensive, requiring a total of $4N_t + 1$ trigonometric function evaluations at each frequency. Press et al. (1992) address the bottleneck by calculating the trig functions from recursion relations. However, this approach is difficult to map onto stream processing concepts, and moreover becomes inaccurate when the number of frequencies is large. An alternative, which avoids these difficulties while still offering improved performance, comes from refactoring

[1]The issue of 'independent' frequencies is briefly discussed in Section 6.2.

equations (1) and (2) as

$$P_{\rm n}(f) = \frac{1}{2}\left[\frac{(c_\tau XC + s_\tau XS)^2}{c_\tau^2 CC + 2c_\tau s_\tau CS + s_\tau^2 SS} + \frac{(c_\tau XS - s_\tau XC)^2}{c_\tau^2 SS - 2c_\tau s_\tau CS + s_\tau^2 CC}\right], \quad (4)$$

and

$$\tan(2\omega\tau) = \frac{2\,CS}{CC - SS}, \quad (5)$$

respectively, with

$$c_\tau = \cos\omega\tau, \qquad s_\tau = \sin\omega\tau. \quad (6)$$

The sums

$$\begin{aligned} XC &= \sum_j X_j \cos\omega t_j, \\ XS &= \sum_j X_j \sin\omega t_j, \\ CC &= \sum_j \cos^2\omega t_j, \\ SS &= \sum_j \sin^2\omega t_j, \\ CS &= \sum_j \cos\omega t_j \sin\omega t_j, \end{aligned} \quad (7)$$

appearing in these expressions can be evaluated together in a single run through the time series, for a total of $2N_t + 3$ trig-function evaluations per frequency.

4. culsp: a GPU Lomb-Scargle Periodogram Code

4.1. Overview

This section introduces CULSP, a Lomb-Scargle periodogram code implemented within NVIDIA's CUDA framework. Below, I provide a brief technical overview of CUDA.

Section 4.3 then reviews the general design of CULSP, and Section 4.4 narrates an abridged version of the kernel source. The full source, which is freely redistributable under the GNU General Public License, is provided in the accompanying on-line materials.

4.2. The CUDA Framework

A CUDA-enabled GPU comprises one or more streaming multiprocessors (SMs), themselves composed of a number[2] of scalar processors (SPs). Together, the SPs allow an SM to support concurrent execution of blocks of up to 512 program threads. Each thread applies the same computational kernel to an element of an input stream. Resources at a thread's disposal include its own register space; built-in integer indices uniquely identifying the thread; up to 16 KB of shared memory common to the thread block; and up to 4 GB of global memory. Accessing shared memory is typically as fast as accessing a register; however, global memory is two orders of magnitude slower.

CUDA programs are written in the C language with extensions that allow computational kernels to be defined and launched, and the differing types of memory be allocated and accessed. A typical program will transfer input data from CPU memory to GPU memory; launch one or more kernels to process these data; and then copy the results back from GPU to CPU. Programs are compiled using the `nvcc` tool from the CUDA software development kit (SDK).

A CUDA kernel has access to the standard C mathematical functions. In some cases, two versions are available ('library' and 'intrinsic'), offering different trade-offs between precision and speed (see Appendix C of NVIDIA 2010). For the cos() and sin() functions, the library versions are accurate to within 2 units of least precision, but are very slow

[2]Eight, for the GPUs considered in the present work.

because the range-reduction algorithm — required to bring arguments into the $(-\pi/4, \pi/4)$ interval — spills temporary variables to global memory. The intrinsic versions do not suffer this performance penalty, but become inaccurate as their arguments depart from the $(-\pi, \pi)$ interval. As discussed below, this inaccuracy can be ameliorated through a very simple range-reduction procedure.

4.3. Code Design

The CULSP code is a straightforward CUDA implementation of the L-S periodogram in its refactored form (equations 5–7). A uniform frequency grid is assumed,

$$f_i = i\,\Delta f \quad (i = 1, \ldots, N_f), \tag{8}$$

where the frequency spacing and number of frequencies are determined from

$$\Delta f = \frac{F_{\rm over}}{t_{N_t} - t_1} \tag{9}$$

and

$$N_f = \frac{F_{\rm high}}{2F_{\rm over}} N_t, \tag{10}$$

respectively. The user-specified parameters $F_{\rm over}$ and $F_{\rm high}$ control the oversampling and extent of the periodogram; $F_{\rm over} = 1$ gives the characteristic sampling established by the length of the time series, while $F_{\rm high} = 1$ gives a maximum frequency equal to the effective Nyquist frequency $f_{\rm Ny} = N_t/[2(t_{N_t} - t_1)]$.

The input time series is read from disk and pre-processed to have zero mean and unit variance, before being copied to GPU global memory. Then, the computational kernel is launched for N_f threads arranged into blocks of size $N_{\rm b}$[3]. Once all calculations are

[3]Set to 256 throughout the present work; tests indicate that larger or smaller values give a slightly reduced performance.

completed, the periodogram is copied back to CPU memory, and from there written to disk.

The sums in equation (7) involve the entire time series. To avoid a potential memory-access bottleneck, and to improve accuracy, CULSP partitions these sums into chunks equal in size to the thread block size N_b. The time-series data required to evaluate the sums for a given chunk are copied from (slow) global memory into (fast) shared memory, with each thread in a block transferring a single (t_j, X_j) pair. Then, all threads in the block enjoy fast access to these data when evaluating their respective per-chunk sums.

4.4. Kernel Source

Figure 1 lists abridged source for the CULSP computational kernel. This is based on the full version supplied in the on-line materials, but special-case code (handling situations where N_t is not an integer multiple of N_b) has been removed to facilitate the discussion.

The kernel accepts five arguments (lines 2–3 of the listing). The first three are array pointers giving the global-memory addresses of the time-series (`d_time` and `d_data`) and the output periodogram (`d_P`). The remaining two give the frequency spacing of the periodogram (`df`) and the number of points in the time series (`N_t`). The former is used on line 11 to evaluate the frequency, as specified by equation (8).

Lines 27–68 construct the sums defined by equation (7), following the partitioning approach described in the preceding section. The outer loop (line 27) iterates over chunks, while the inner loop (line 41) evaluates the per-chunk sums; in both loop definitions, the macro `BLOCK_SIZE` is expanded by the pre-processor to the thread block size N_b. The `SS` sum is not calculated explicitly, but reconstructed from the `CC` sum on line 70.

The sin() and cos() terms in the sums are evaluated simultaneously with a call to CUDA's intrinsic `__sincosf()` function (line 49). To maintain accuracy, a simple range

reduction is applied to the phase `ft`, by discarding its integer part (line 44). This brings the argument of the `__sincosf()` function into the interval $(-2\pi, 2\pi)$. It is possible to further reduce the argument into the $(-\pi, \pi)$ interval, but the very minor improvement in accuracy is offset by a 50% increase in execution time.

5. Benchmarking Calculations

5.1. Test Configurations

This section investigates the accuracy and performance of CULSP, running on a pair of GPUs summarized in Table 1. The GeForce 8400 GS is based on NVIDIA's G98 hardware architecture (released 2007), and is an entry-level GPU aimed at the budget segment of the market. The Tesla C1060 is based on the GT200 architecture (released 2008/2009), and was until recently the flagship in their GPU computing product line. Both GPUs are hosted in a Dell Precision 490 workstation, containing two Intel 2.33 GHz Xeon E5345 quad-core processors and running 64-bit Gentoo Linux. The 3.0 release of the CUDA SDK is used, with `gcc` 4.1.2 as the host compiler.

A benchmarking baseline is established using LSP, a CPU-based code that follows the same basic algorithm as CULSP. This code is written in ISO C99 and compiled using Intel's `icc` 11.1 compiler, with the `-O3` and `-xHost` optimization flags. Source for LSP is provided

Table 1. Specifications for the two GPU systems used in the benchmarking.

GPU	SMs	SPs	Clock (GHz)	Memory (MB)
GeForce 8400 GS	1	8	1.4	512
Tesla C1060	30	240	1.3	4096

in the accompanying on-line materials.

As the validation dataset on which to compare the two codes, I use the 150-day photometric time series of the β Cephei pulsator V1449 Aql (HD 180642) obtained by the *CoRoT* mission (Belkacem et al. 2009). The observations comprise 382003 flux measurements (after removal of points flagged as bad), sampled unevenly (in the heliocentric frame) with an average separation of 32 s.

5.2. Accuracy

Figure 2 plots the periodogram of V1449 Aql evaluated using LSP, over a frequency interval spanning the star's dominant 0.18 d pulsation mode (see Waelkens et al. 1998). Also shown in the figure is the absolute deviation $|P_{\rm n}^{\rm CULSP} - P_{\rm n}^{\rm LSP}|$ of the corresponding periodogram evaluated using CULSP. The figure confirms that, at least over this particular frequency interval, the two codes are in good agreement with one another; the relative error is on the order of 10^{-6}.

To explore accuracy over the full frequency range, Fig. 3 shows a scatter plot of $P_{\rm n}^{\rm LSP}$ against $P_{\rm n}^{\rm CULSP}$. Very few of the $N_f = 1,527,808$ points in this plot depart to any significant degree from the diagonal line $P_{\rm n}^{\rm LSP} = P_{\rm n}^{\rm CULSP}$. Those that do are clustered in the $P_{\rm n} \ll 1$ corner of the plot, and are therefore associated with the noise in the light curve rather than any periodic signal. Moreover, the maximum absolute difference in the periodogram FAPs (equation 3) across *all* frequencies is 3.3×10^{-5}, which is for all purposes negligible.

5.3. Performance

Table 2 lists the time $t_{\rm calc}$ taken to calculate the full V1449 Aql periodogram, for each combination of code and hardware platform. (The Intel Xeon X5550 and AMD Opteron 2387 platforms are included in the table in anticipation of the discussion in the following section; however, they are not considered in the present analysis). The values exclude the computational overhead from disk input/output, and from rectifying the light curve to zero mean and unit variance. Clearly, LSP is outperformed by CULSP on both GPU platforms; the speed-up is most pronounced on the Tesla C1060, but even on the GeForce 8400 GS, CULSP is over seven times faster than LSP.

The expressions given in Section 3 indicate a periodogram calculation time scaling as N_t per frequency. With the number of frequencies following $N_f \propto N_t$ (equation 10), the overall time should therefore scale as N_t^2. To test this expectation, Fig. 4 shows a log-log plot of $t_{\rm calc}$ as a function of N_t, for the three main code/platform combinations. The light curve for a given N_t is generated from the full V1449 Aql light curve by uniform down-sampling.

In the limit of large N_t, all curves asymptote toward a slope $\mathrm{d}\log t_{\rm calc}/\mathrm{d}\log N_t = 2$, confirming the hypothesized $t_{\rm calc} \propto N_t^2$ scaling. At smaller N_t, departures from this scaling arise from computational overheads that are not associated with the actual periodogram calculation. These are most clearly seen in the CULSP curves, which — on either GPU platform — asymptote toward a constant $t_{\rm calc} \approx 60\,\mathrm{ms}$ independent of N_t. This 'floor' on $t_{\rm calc}$ represents the cost of initializing the CUDA library; since this occurs only once per program execution, its impact can be minimized by processing a number of light curves in sequence.

As a final check on code performance, I use NVIDIA's `cudaprof` profiling tool to examine whether the CULSP is making efficient use of GPU resources. The results for the

Tesla C1060 indicate a total instruction throughput of 190 billion operations per second[4] — 61% of the device's single-instruction-issue maximum, indicating that good arithmetic intensity is being achieved. Memory access are moreover close to optimal, with `cudaprof` reporting that almost all reads from global memory are coalesced, and that no bank conflicts occur when reading from shared memory.

6. Discussion

6.1. Cost/Benefit Analysis

The foregoing sections indicate that CULSP outperforms LSP by a wide margin — 200 times faster when run on the Tesla C1060 GPU — with no significant loss of accuracy. However, to establish a practical context for this result, it is necessary to consider factors such as the cost and capabilities of the respective hardware platforms. At the time of writing, the Tesla GPU retails for around $1,300, while the manufacturer's bulk (1,000-unit) pricing for the Xeon E5345 CPU is $455, translating (naively) into $114 per core — over ten times cheaper than the Tesla. Nevertheless, even with these differences the Tesla offers the better price-performance ratio. The same is true of the GeForce 8400 GS, which currently retails for around $40.

A caveat here is that the hardware platforms considered are a couple of years old. To examine performance on more-recent CPUs, I have undertaken the same timing tests for LSP running on a Apple Mac Pro workstation containing two Intel 2.66 GHz Xeon X5550 quad-core processors (OS X 10.6.3), and on a Microway workstation containing two AMD

[4]This is can be thought of as broadly equivalent to GFLOPS (billions of floating-point operations per second), although it does not account for the fact that trig-function evaluations cost more than one floating-point operation.

2.8 GHz Opteron 2387 quad-core processors (64-bit Ubuntu Linux). The results, presented at the foot of Table 2, reveal a modest performance gain ($\sim 15\%$) over the Xeon E5345. The cost of the Xeon X5550 and Opteron 2387, based on the manufacturers' bulk pricing, is \$958 and \$523 respectively, translating into \$240 and \$131 per core; thus, on the CPU side the price-performance ratio has remained largely unchanged.

On the GPU side, NVIDIA's GT200 architecture has recently been superseded by their new GF100 'Fermi' architecture, which provides the basis for the new high-end Tesla C2050/C2070 products and the mainstream GeForce GTX 400 line. The GeForce GTX 480, offering nearly twice the single-precision floating-point performance of the older Tesla C1060, retails for around \$500. Combining these various figures, the best-case periodogram calculation throughput on a modern GPU (for the same $F_{\rm over} = 4$ and $F_{\rm high} = 2$ as before) should be around $10^8 N_t^{-1}$ frequencies per second per unit cost, compared to a corresponding CPU throughput of $10^6 N_t^{-1}$ frequencies per second per unit *core* cost. Hence, the price-performance ratio of CULSP remains around 100 times that of LSP.

6.2. Applications

The primary motivation for developing CULSP is to facilitate spectral analysis of the photometric time series obtained by ongoing satellite missions such as *MOST* (Walker et al. 2003), *CoRoT* (Auvergne 2009), and most recently, *Kepler* (Koch 2010). These datasets are typically very large ($N_t \gtrsim 10^5$), leading to a significant per-star cost for calculating a periodogram. When this cost is multiplied by the number of targets being monitored (in the cast of *Kepler*, again $\gtrsim 10^5$), the overall computational burden becomes very steep. It is hoped that CULSP, or an extension to other related periodograms (see below), will help resolve this issue.

An additional application of CULSP lies in the interpretation of periodograms. Equation (3) presumes that the $P_{\rm n}$ at each frequency in the periodogram is independent of the others. This is not necessarily the case, and the exponent in the equation should formally be replaced by some number $N_{f,\rm ind}$ representing the number of independent frequencies. Horne & Baliunas (1986) pioneered the use of simulations to estimate $N_{f,\rm ind}$ empirically, and similar Monte-Carlo techniques have since been applied to explore the statistical properties of the L-S periodogram in detail (see Frescura et al. 2008, and references therein). The bottleneck in these simulations are the many periodogram evaluations, making them strong candidates for GPU acceleration.

6.3. Further Work

Recognizing the drawbacks of being wedded to one particular hardware/software vendor, a strategically important future project will be to port the CULSP code to Open CL (Open Computing Language) — a recently developed standard for programming devices such as multi-core CPUs and GPUs in a platform-neutral manner (see, e.g., Stone et al. 2010). There is also considerable scope for applying the lessons learned herein to other spectral analysis techniques. Shrager (2001) and Zechmeister & Kürster (2009) generalize the L-S periodogram to allow for the fact that the time-series mean is typically not known *a priori*, but instead estimated from the data themselves. The expressions derived by these authors involve sums having very similar forms to equation (7); thus, it should prove trivial to develop GPU implementations of the generalized periodograms. The multi-harmonic periodogram of Schwarzenberg-Czerny (1996) and the SigSpec method of Reegen (2007) also appear promising candidates for implementation on GPUs, although algorithmically they are rather more-complex.

Looking at the bigger picture, while the astronomical theory and modeling communities

have been quick to recognize the usefulness of GPUs (see Section 1), progress has been more gradual in the observational community; radio correlation is the only significant application to date (Wayth et al. 2009). It is my hope that the present paper will help address this issue, by illustrating the powerful data-analysis capabilities of GPUs, and by demonstrating strategies for using these devices effectively.

7. Summary

I introduce a new code, CULSP, for calculating the Lomb-Scargle periodogram on graphics processing units. The code outperforms an equivalent CPU-based code by over two orders of magnitude. Possible applications include spectral analysis of long photometric time series obtained by ongoing satellite missions; and Monte-Carlo simulation of periodogram statistical properties.

I thank Dr. Gordon Freeman for providing the initial motivation to explore this line of research, and Ben Brown for the loan of computer resources for testing. I moreover acknowledge support from NSF *Advanced Technology and Instrumentation* grant AST-0904607. The Tesla C1060 GPU used in this study was donated by NVIDIA through their Professor Partnership Program, and I have made extensive use of NASA's Astrophysics Data System Bibliographic Services.

```
__global__ void
culsp_kernel(float *d_t, float *d_X, float *d_P,
             float df, int N_t)
{

  __shared__ float s_t[BLOCK_SIZE];
  __shared__ float s_X[BLOCK_SIZE];

  // Calculate the frequency

  float f = (blockIdx.x*BLOCK_SIZE+threadIdx.x+1)*df;

  // Calculate the various sums

  float XC = 0.f;
  float XS = 0.f;
  float CC = 0.f;
  float CS = 0.f;

  float XC_chunk = 0.f;
  float XS_chunk = 0.f;
  float CC_chunk = 0.f;
  float CS_chunk = 0.f;

  int j;

  for(j = 0; j < N_t; j += BLOCK_SIZE) {

    // Load the chunk into shared memory

    __syncthreads();

    s_t[threadIdx.x] = d_t[j+threadIdx.x];
    s_X[threadIdx.x] = d_X[j+threadIdx.x];

    __syncthreads();

    // Update the sums

    #pragma unroll
    for(int k = 0; k < BLOCK_SIZE; k++) {

      float ft = f*s_t[k];
      ft -= (int) ft;

      float c;
      float s;

      __sincosf(TWOPI*ft, &s, &c);

      XC_chunk += s_X[k]*c;
      XS_chunk += s_X[k]*s;
      CC_chunk += c*c;
      CS_chunk += c*s;

    }

    XC += XC_chunk;
    XS += XS_chunk;
    CC += CC_chunk;
    CS += CS_chunk;

    XC_chunk = 0.f;
    XS_chunk = 0.f;
    CC_chunk = 0.f;
    CS_chunk = 0.f;

  }

  float SS = (float) N_t - CC;

  // Calculate the tau terms

  float ct;
  float st;

  __sincosf(0.5f*atan2(2.f*CS, CC-SS), &st, &ct);

  // Calculate P

  d_P[blockIdx.x*BLOCK_SIZE+threadIdx.x] =
      0.5f*((ct*XC + st*XS)*(ct*XC + st*XS)/
            (ct*ct*CC + 2*ct*st*CS + st*st*SS) +
            (ct*XS - st*XC)*(ct*XS - st*XC)/
            (ct*ct*SS - 2*ct*st*CS + st*st*CC));

  // Finish

}
```

Fig. 1.— Abridged source for the CULSP computation kernel.

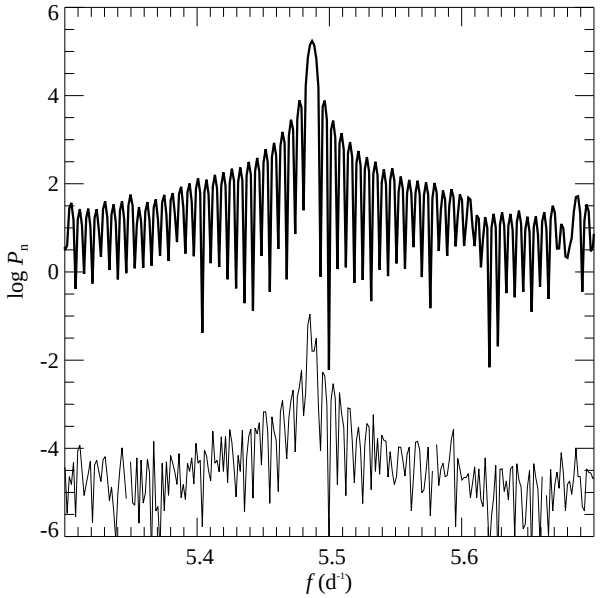

Fig. 2.— Part the L-S periodogram for V1449 Aql, evaluated using the LSP code (thick curve). The thin curve shows the absolute deviation $|P_{\rm n}^{\rm CULSP} - P_{\rm n}^{\rm LSP}|$ of the corresponding periodogram evaluated using the CULSP code. The strong peak corresponds to the star's dominant 0.18 d pulsation mode.

Table 2. Periodogram calculation times

Code	Platform	$t_{\rm calc}$ (s)
CULSP	NVIDIA GeForce 8400 GS	610.3
CULSP	NVIDIA Tesla C1060	21.7
LSP	Intel Xeon E5345	4406.0
LSP	Intel Xeon X5550	3760.1
LSP	AMD Opteron 2387	3779.0

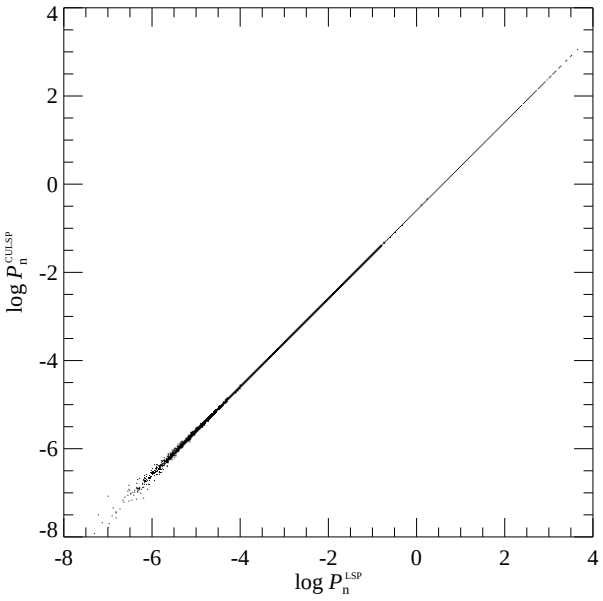

Fig. 3.— A scatter plot of the L-S periodogram for V1449 Aql, evaluated using the LSP (abscissa) and CULSP (ordinate) codes.

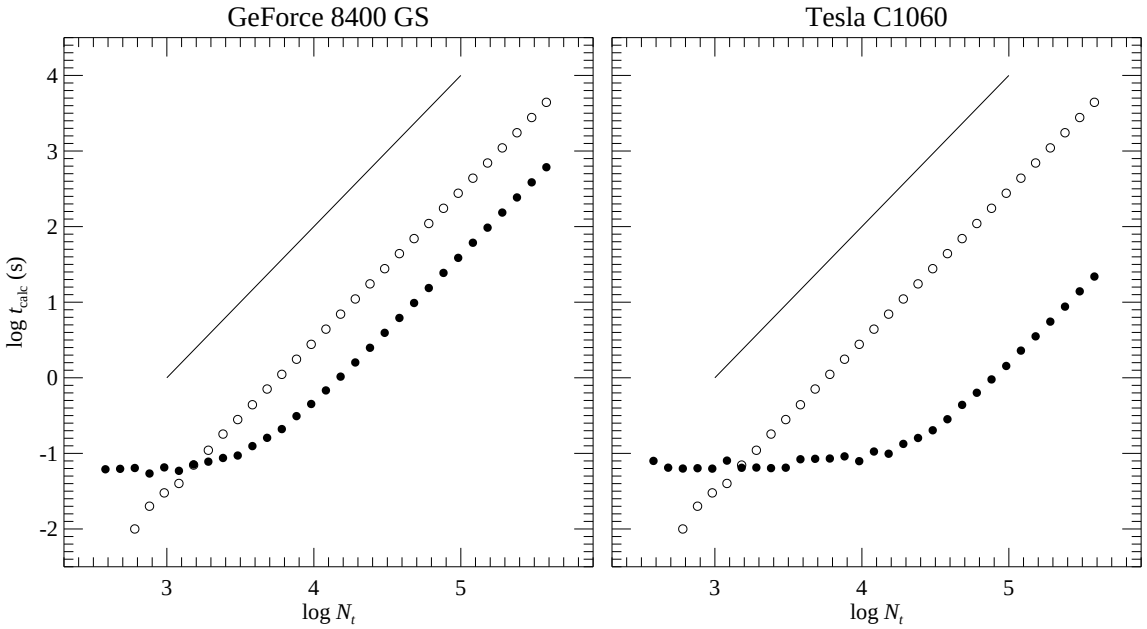

Fig. 4.— Calculation times $t_{\rm calc}$ for the L-S periodogram, evaluated using the LSP (open circles) and CULSP (filled circles) codes. The left and right panels correspond to the two GPUs considered; the diagonal line in each has a slope $\mathrm{d}\log t_{\rm calc}/\mathrm{d}\log N_t = 2$.

REFERENCES

Aubert D., Teyssier R., 2010, ApJ, submitted; arXiv: 1004.2503

Auvergne M. et al., 2009, A&A, 506, 411

Belkacem K., Samadi R., Goupil M., Lefèvre L., Baudin F., Deheuvels S., Dupret M., Appourchaux T., Scuflaire R., Auvergne M., Catala C., Michel E., Miglio A., Montalban J., Thoul A., Talon S., Baglin A., Noels A., 2009, Science, 324, 1540

Belleman R. G., Bédorf J., Portegies Zwart S. F., 2008, New Astronomy, 13, 103

Buck I., Foley T., Horn D., Sugerman J., Fatahalian K., Houston M., Hanrahan P., 2004, ACM Transactions on Graphics, 23, 777

Cooley J. W., Tukey J. W., 1965, Mathematics of Computation, 19, 297

Frescura F. A. M., Engelbrecht C. A., Frank B. S., 2008, MNRAS, 388, 1693

Horne J. H., Baliunas S. L., 1986, ApJ, 302, 757

Koch D. G. e. a., 2010, ApJ, 713, L79

Lomb N. R., 1976, Ap&SS, 39, 447

Nguyen H., 2007, GPU Gems 3. Addison-Wesley Professional

NVIDIA 2010, NVIDIA CUDA Programming Guide 3.0

Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E., Purcell T. J., 2005, in Eurographics 2005: State of the Art Reports, p. 21

Pharr M., 2005, GPU Gems 2. Addison-Wesley Professional

Press W. H., Rybicki G. B., 1989, ApJ, 338, 277

Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P., 1992, Numerical Recipes in Fortran, 2 edn. Cambridge University Press, Cambridge

Reegen P., 2007, A&A, 467, 1353

Rost R. J., 2006, OpenGL Shading Language, 2 edn. Addison-Wesley Professional

Scargle J. D., 1982, ApJ, 263, 835

Schive H., Tsai Y., Chiueh T., 2010, ApJS, 186, 457

Schwarzenberg-Czerny A., 1996, ApJ, 460, 107

Schwarzenberg-Czerny A., 1998, MNRAS, 301, 831

Shrager R. I., 2001, Ap&SS, 277, 519

Stone J. E., Gohara D., Shi G., 2010, Computing in Science and Engineering, 12, 66

Thompson A. C., Fluke C. J., Barnes D. G., Barsdell B. R., 2010, New Astronomy, 15, 16

Waelkens C., Aerts C., Kestens E., Grenon M., Eyer L., 1998, A&A, 330, 215

Walker G., Matthews J., Kuschnig R., Johnson R., Rucinski S., Pazder J., Burley G., Walker A., Skaret K., Zee R., Grocott S., Carroll K., Sinclair P., Sturgeon D., Harron J., 2003, PASP, 115, 1023

Wayth R. B., Greenhill L. J., Briggs F. H., 2009, PASP, 121, 857

Zechmeister M., Kürster M., 2009, A&A, 496, 577

This manuscript was prepared with the AAS LaTeX macros v5.2.