LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO T1700206 *LIGO* September 22, 2017

An Optimization for Coherent WaveBurst – Enabling GPU Capable Maximum Likelihood Ratio Calculations

Yanqi Gu and Sharon Brunett

Introduction

Detections of gravitational waves have opened a new era in astrophysics. Detectors are operated around the world, including the Laser Interferometer Gravitational-Wave Observatory (LIGO) and Virgo. An all-sky search for transient gravitational waves (GW) from astrophysical burst sources is a searching method, used to successfully identify the first detected gravitational wave, GW150914.

cWB [1] is an important all-sky search pipeline combining data streams from at least two arbitrarily aligned detectors, into a coherent statistic for analysis based on maximum likelihood estimation (MLE). Based on a likelihood maximization of coherent response for the detector network, cWB pipeline identifies GW events in collected data and establishes the event significance over three standard deviations [2].

Gravitational wave signal reconstruction has been identified as a highly CPU intensive and time-consuming step in the coherent WaveBurst (cWB) data analysis pipeline. Here we present a runtime profile for a cWB benchmark of interest, which influences our choice of routines benefiting from parallelization and optimization. A parallel computing model for Maximum Likelihood Ratio (MLR) calculations in the all-sky search process and an efficient algorithm for porting the model onto a GPU are described. Lastly, experimental runs and timings are discussed.

Profiling cWB

Performance of the cWB pipeline can generally be analyzed in several stages: (1) data input/output and conditioning; (2) time-frequency (TF) transformation and selection of excess power samples; (3) TF pattern recognition analysis in clusters and identification of burst events at multiple TF resolutions; (4) calculation of the detection statistic, sky location and reconstruction of burst waveforms [2].

We ran the pipeline on various platforms with different compilers, listing total runtimes, for a cWB benchmark in Table 1 and a breakdown of times for various stages within the pipeline in Figure 1. Runtimes on the E5-2670 are very similar, building with gcc and the Intel C compiler, using standard optimization flags (e.g. –O3 -march=native -funroll-loops). We used modern compiler versions – icc v 17.0.2, gcc v 4.8.5, NVIDIA CUDA Compiler Driver (NVCC) v 8.0.

Platform	Compiler	Clock Speed	Runtime
XeonPhi7210	ICC	1.30GHz	4:47:31
XeonPhi7210	GCC	1.30GHz	4:34:06
XeonE5-2670	ICC	2.60GHz	1:21:03
XeonE5-2670	GCC, NVCC	2.60GHz	1:20:12

Table 1: Overall runtime comparisons on a single core

The profile below, Figure 1, shows the reconstruction stage is by far the most time consuming, a factor of 8.8 more than the likelihood stage.

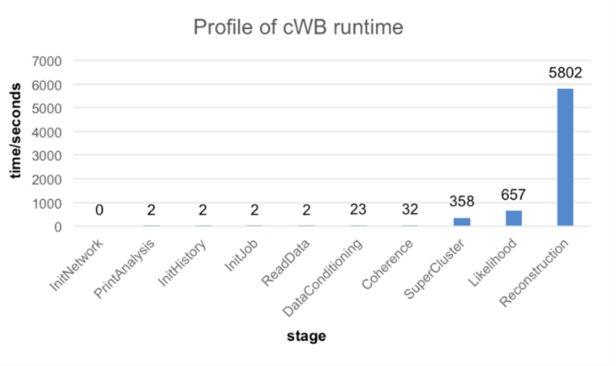

Figure 1: Time cost of different stages during sample cWB run

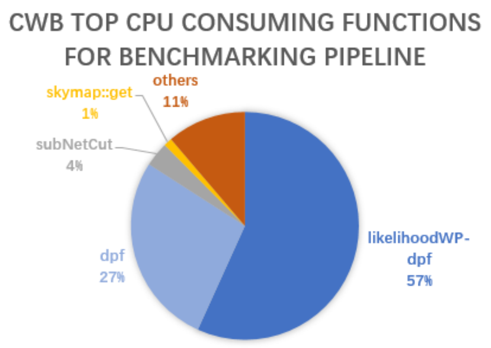

Figure 2: Breakdown of top CPU consuming functions

Figure 2 profiles functions, not stages, within a cWB run. Not surprisingly, the highest runtime cost, 57% of total runtime, is due to the likelihoodWP function, which occurs within the reconstruction stage. The dpf function accounts for 27% percent of the time spent in the likelihoodWP function.

Due to our profiling results, we decided to focus on optimizing stage 4, the dpf portion of the likelihoodWP stage, where the time-frequency analysis, time-shift analysis and the search over nearly 200,000 sky locations are performed.

Likelihood analysis in a coherent WaveBurst pipeline

S. Klimenko *et al* [1] proposed a coherent method for the detection and reconstruction of gravitational wave signals using a network of gravitational wave detectors. This method is derived using the likelihood functional for unknown signal waveforms [3], which allows reconstruction of the source coordinates and waveforms of two polarization components of a gravitational wave.

The whole reconstruction stage of coherent WaveBurst is based on the use of the global maximum of likelihood ratio function. For Gaussian quasi-stationary noise, the likelihood function in the wavelet domain can be written as:

$$\mathcal{L} = \sum_{k=1}^{K} \sum_{i,j=1}^{N} \left(\frac{w_k^2[i,j]}{\sigma_k^2[i,j]} - \frac{(w_k[i,j] - \xi_k[i,j])^2}{\sigma_k^2[i,j]} \right)$$

where K is the number of detectors in the network, $\omega_k[i,j]$ is the sampled detector data and $\xi_k[i,j]$ is the detector responses. Standard deviation $\sigma_k[i,j]$, which may vary over the time-frequency plane, characterizes the detector noise.

The detection response can be written in standard notations:

$$\xi_k[i,j] = F_{+k}h_+[i,j] + F_{\times k}h_\times[i,j],$$

where $F_{+k}(\theta,\phi)$,$F_\times(\theta,\phi)$ are antenna patterns of detector, which depend on coordinates θ and ϕ.

Once we know the variation of L, we can find GW waveforms. The maximum likelihood ratio statistic is obtained by substitution of the solutions into the function L, and waveforms in the time domain are reconstructed from the inverse wavelet transformation. For convenience, the data vector and antenna pattern vector are introduced:

$$\mathbf{w}[i,j] = \left(\frac{w_1[i,j]}{\sigma_1[i,j]}, \dots, \frac{w_K[i,j]}{\sigma_K[i,j]} \right)$$

$$\mathbf{f}_{+(\times)}[i,j] = \left(\frac{F_{1+(\times)}}{\sigma_1[i,j]}, \dots, \frac{F_{K+(\times)}}{\sigma_K[i,j]} \right)$$

Antenna pattern vectors are defined in the Dominant Polarization wave Frame (DPF), where the antenna pattern vectors are orthogonal to each other, $(\mathbf{f}_+ * \mathbf{f}_\times) = 0$. The estimators of the GW waveforms are the solutions of the equations:

$$(\mathbf{w} \cdot \mathbf{f}_+) = |\mathbf{f}_+|^2 h_+ \quad ,$$

$$(\mathbf{w} \cdot \mathbf{f}_\times) = |\mathbf{f}_\times|^2 h_\times \quad ,$$

where $|\mathbf{f}_+|^2$ and $|\mathbf{f}_\times|^2$, represent the sensitivity of the network to h_+ and $h_\times$.

Because of the effect of source location, the network can be less sensitive to $|\mathbf{f}_\times|^2$. To solve this problem, a specific class of constraints (regulators), which arise from network responses to a generic GW signal, is added. In coherent WaveBurst analysis, changing the norm of the f× vector uses a regulator

$$|\mathbf{f}'_\times|^2 = |\mathbf{f}_\times|^2 + \delta,$$

The regulator, for vectors f+ and f×, preserves the orthogonality and the maximum likelihood statistic is written as:

$$L_{\max} = \sum_{\Omega_{TF}} \left[\frac{(\mathbf{w} \cdot \mathbf{f}_+)^2}{|\mathbf{f}_+|^2} + \frac{(\mathbf{w} \cdot \mathbf{f}'_\times)^2}{|\mathbf{f}'_\times|^2} \right] = \sum_{\Omega_{TF}} \left[(\mathbf{w} \cdot \mathbf{e}_+)^2 + (\mathbf{w} \cdot \mathbf{e}'_\times)^2 \right],$$

The solutions of the likelihood functional equation show the GW waveform. Two GW components of the solutions calculated by:

$$h_+ = \frac{(\mathbf{w} \cdot \mathbf{f}_+)}{|\mathbf{f}_+|^2}$$

$$h_\times = \frac{(\mathbf{w} \cdot \mathbf{f}_\times)}{|\mathbf{f}'_\times|^2} \left(1 + \sqrt{1 - \frac{|\mathbf{f}_\times|^2}{|\mathbf{f}'_\times|^2}} \right)^{-1}$$

Calculating the likelihood function

In general, the likelihood function is calculated as a sum over the data samples selected for the analysis, depending on the selected TF area in the wavelet domain. The likelihood functional for a given TF location and point in the sky can be maximized over the source coordinates θ and φ:

$$L_m(i,j) = \max_{\theta,\varphi}\{L_p(i,j,\theta,\varphi)\}$$

The above equation gives us a likelihood time-frequency (LTF) map. A single data sample in the map is called the LTF pixel, which characterized by its TF location *(i,j)* and by the arrays of wavelet amplitudes $w_k(i,j,\tau_k(\theta,\varphi))$. A group of pixels selected in a single detector is called a cluster and refer to a group of LTF pixels as a coherent trigger. The real data is combined with instrumental and environmental glitches, so additional cuts are needed to distinguish genuine GW signals, and the maximum likelihood is then required for detection and selection of GW events, and the false alarm is controlled by its threshold.

Reconstruction – Sequential MLR

In the cWB pipeline, the MLR calculation is time-consuming because it runs over 3 layers of loops:

```
for lags
   for clusters
      for sky locations
            Calculations();
   getMLR();
   SelectOrRejectCluster();
sum();
```

A lag is a data packet sent from detectors, each lag includes several clusters. The second layer loops over all clusters not filtered in the prior phase while the third layer loops over all sky locations (coordinates in the sky). Lastly, each cluster not filtered, with a particular sky location, receives an MLR statistic for further calculations.

Runtime profiles are highly dependent on input parameters, so we use a collection specified by the code developers as being typical of prior and upcoming production jobs. In this case, we use 590 lags, 6,576 clusters (some of them are filtered before MLR calculation) and 196,608 sky locations. All sky locations must be searched, a limiting factor for creating a fast implementation of the cWB pipeline.

The sequential MLR calculation is described in figure 3, as follows:

```
for l = 1,2,...,196,608 {                                        // for all sky locations
    //apply sky mask
    pa,pA<- pnt(v00,v90)                                         // initialize pointers to first pixel 00
    and 90 data
    pd,pD<-loadata(v00,v90); dpf(FP,FX) // calc. data statistics, store & calc. DPF, norms
    ps,pS<-cpf(v00,v90,REG)                                      // copy data for GW reconstruction
    ort(ps,pS)                                                   // Orthogonalize signal
    amplitudes
    stat(pd,pD,ps,pS)                                            // get coherent statistics
    Cr,Ec,Mp,No,cc,Co                                            // extract coherent statistics
    AA<-StatisticsCalculation()
    STAT<-AA,lm<-l                                               // select maximum and store l
}                                          // end for all sky locations
calculations to select or reject cluster(lm,AA[lm])
```

Figure 3: Sequential MLR

The original algorithm for all-sky searching is sequential nature, not readily suitable, for explicit data parallelism. An initial OpenMP port was attempted, but significant data structure and implementation changes would have been necessary. Fortunately, the current sequential implementation is nicely accelerated by the use of streaming SIMD extensions (SSE). However, since SSE is not supported on GPUs, it is necessary to transform SSE code [4] [5] back to a simple float version before porting this code to the GPU.

Reconstruction – Parallel MLR

We designed a model for a portion of the reconstruction stage, suitable for parallelization and GPUs. Currently, we are unaware of previous efforts to port the reconstruction phase of cWB to GPUs, which nicely complements the sky loop stage already capable of using GPUs. We believe our work is relevant in providing a general parallel model for MLR calculations for massive data processing in all-sky searching routines. Other pipelines using the MLR method could also use this model.

We will use a typical hybrid CPU-GPU structure to implement our parallel MLR model. Stringent requirements are imposed for porting to GPUs, such as a high degree of parallelism, coalesced memory accesses and control divergence. We see an opportunity to design a parallel MLR calculation model by following a swimming competition model, in figure 3.

Figure 4: Parallel swimming lanes for simultaneous swimmers

In the figure above, one swimmer (player) represents one sky location, and one lane represents one computational thread. The original algorithm on a CPU is sequential, it has only one swimming lane. When the competition begins, the first player starts to swim in this lane. Only after he finishes can the next player start. Each player will get a point of likelihood ratio after swimming. Throughout the whole competition, the referee will update the best record, and the owner of that record, to date.

In our parallel model, each player has one lane, and many players swim simultaneously. The referees (each player gets a referee) record all swim times and compare results after the competition to pick the champion.

Significant changes to data structures were made to allow the parallelization of the MLR computation. Figure 5 outlines the revised MLR computation, with routines running on the GPU designated in italics:

```
_
    // Initialize GPU and copy input data to GPU
      for L (l = 1,2…196,608)                   // each with one thread, do in parallel:
                                              // apply sky mask
    pa,pA<- gpupnt(v00,v90,L)           // initialize pointers to first pixel 00 and 90 data
```

```
pd,pD<-gpuloadata(v00,v90,L)// calculate data statistics and store
gpudpf(FP,FX,L)                          // calculate DPF and norms
ps,pS<-cpf(v00,v90,REG,L)       // copy data for GW reconstruction
ort(ps,pS,L)                                  // orthogonalize signal amplitudes
stat(pd,pD,ps,pS,L)                      // get coherent statistics
Cr,Ec,Mp,No,cc,Co                       // extract coherent statistics
AA<-StatisticsCalculation(L)

   /// Store AA on GPU and transfer to CPU,
lm<-SelectMaxAA(L)

   // further calculations to select or reject cluster(lm,AA[lm])
```

Figure 5: Parallel MLR

The parallel model shows promise for improved performance, creating a need for a larger pool with more lanes (e.g. more computational cores for more groups of simultaneous swimmers)!

GPU Architecture and CUDA Programming Model

Two common choices when selecting a platform to implement parallel algorithms are multicore CPUs and GPU (Graphics Processing Units). A simple way to understand the difference between a GPU and a CPU is to compare how they process tasks. A CPU consists of small numbers of fast cores optimized for sequential processing while a GPU has a massively parallel architecture consisting of thousands of smaller, highly efficient cores designed for handling multiple tasks simultaneously. The CPU wins with general purpose and clock speed, but for select calculations in cWB we can utilize thousands of GPU cores. The target portion of the calculation we want to port to the GPU is not overly complex and the input can be compressed. Giving us confidence for porting a portion of cWB to a GPU is an early GPU implementation for GW searches, as presented by Chung [7].

With the release of the compute unified device architecture (CUDA) [6] in 2006, GPUs have been widely used in high performance computing. Modern GPUs can accommodate massive threads and perform general mathematical operations in a single-instruction, multiple-thread (SIMT) way. The use of CUDA is a powerful and cost-effective solution to computationally intensive problems in many areas, including gravitational wave searching. Three steps are needed in a successful CUDA implementation. First, input data must be copied from CPU to GPU. Second, calculations, also called kernels, are performed on the GPU. Finally, the results are copied from the GPU back to CPU.

In our work, we use an NVIDIA Tesla P100 GPU containing 3584 CUDA cores. Although GPU programs can typically be written using OpenCL or CUDA, we choose CUDA for performance and tool (profilers, debuggers) maturity reasons, .

Experiment and Analysis

The benchmark data set in this test is provided by S. Klimenko *et al* [1], based on O2 production runs. The reconstruction includes 590 lags, 196,608 sky locations and 6,576 clusters. The pipeline we use here is for cWB2G analysis; it searches for unmodeled MRA packets among data from September 12, 2016 to January 19, 2017.

Our test platform is an Intel Xeon E5-2670 (2.60GHz) hosting an Nvidia Tesla P100.

We run and time our parallel MLR implementation against the sequential version. Figure 6 shows the performance results of having ported the largest time-consuming function – dpf - in MLR,, to the GPU. The input data packet is random, we choose to show the first lag's timing result. The GPU (parallel kernel) timing represents the dpf calculation stage, done solely on the GPU. The CPU+GPU (parallel total) timing includes the runtime for the kernel and time for copying data between CPU and GPU, its value is bigger than GPU (parallel kernel), as shown in Figure 6. SSE(sequential) times are for the SSE sequential version, running solely on the CPU.

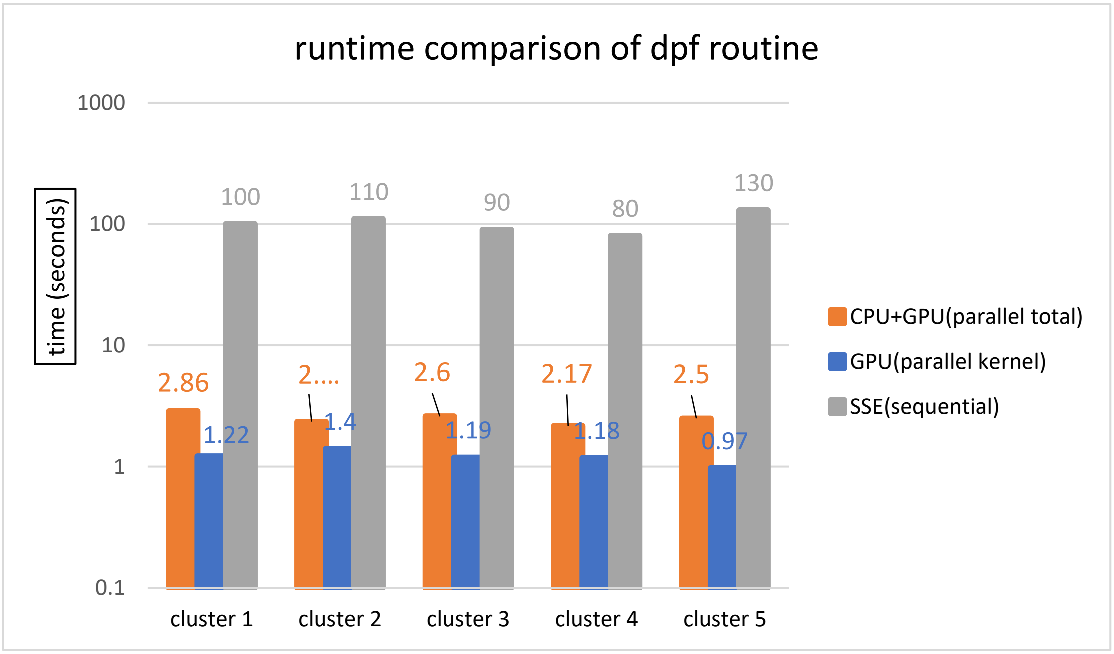

Figure 6. runtime comparison of dpf, GPU vs CPU vs hybrid for lag 0

In Figure 6, we can see very nice performance improvements, comparing the SSE dpf timings to the CPU+GPU dpf times. Best case, we see a factor of 52 improvement in cluster 5, 130s vs 2.5s. Worse case, a factor of 34 improvement for cluster3. Regardless of clusters timed, we see tremendous performance improvement for a given dpf invocation on the CPU+GPU, compared to the SSE implementation of the same routine and same data set.

The dpf function is called twice in function likelihoodWP, once in the preparation step before the MLR calculation (whose parallelized performance is shown in Figure 6), the other within MLR calculation porting the second call to the dpf function, requires slightly different data input/output storing/copying overhead than the first call, which is not difficult but would need to be done if the MLR calculation is ported to the GPU in pieces. A longer term goal is to port the whole MLR calculation to the GPU, as we continue to see performance improvement with subsets of the calculation**.**

Conclusion and future work

In this paper, we analyzed a typical cWB pipeline execution and identified major factors contributing to runtime costs. Based on our profile, we ported the time-consuming routine calculating MLR to a GPU for specific parallelization. We designed a parallel model for MLR calculation and built a prototype hybrid CPU-GPU implementation. We have shown our GPU capable implementation, with the provided data sets, can perform faster than the original CPU implementation.

Currently, only the dpf function is ported to run on the GPU. The next logical improvement is for the entire MLR calculation to reside on the GPU. Currently, the parallel MLR model is 1D. As the number of lags and clusters increase, we can modify our model to a 3D version to achieve more parallelism.

References

[1] S.Klimenko, I.Yakushin, A.Mercer, G.Mitselmakher, Coherent method for detection of gravitational wave bursts

[2] S.Klimenko, G.Vedovato, E.Lebigot, All-sky burst search: Performance and scaling.

[3] S.Klimenko, S.Mohanty, M.Rakhmanov and G.Mitselmakher, Constraint likelihood analysis for a network of gravitational wave detectors

[4] http://sci.tuomastonteri.fi/programming/sse

[5] https://msdn.microsoft.com/en-US/library/x8zs5twb(v=vs.80).aspx

[6] http://docs.nvidia.com/cuda/

[7] Chung S K, Wen L, et al, Application of graphics processing units to search pipelines for gravitational waves from coalescing binaries of compact objects *Class. Quantum Grav.* **27** 135009