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

Yanqi Gu[1], Sharon Brunett[2] and Zhihui Du[3]

[1] Beijing University of Posts and Telecommunications, Beijing 100876, China
gabrielgugyq@gmail.com
[2] California Institute of Technology, 1200 E. California Blvd, Pasadena, CA 91125, USA
sbrunett@ligo.caltech.edu
[3] Tsinghua University, Beijing 100084, China
duzh@tsinghua.edu.cn

Abstract. Gravitational wave(GW) signal reconstruction has been identified as a highly CPU intensive and time-consuming step in the coherent WaveBurst (cWB) data analysis pipeline. In this step, based on a likelihood maximization of coherent response for the detector network, cWB pipeline combines data streams from at least two arbitrarily aligned detectors, identifies GW events in collected data and establishes the event significance over three standard deviations. In this work we present a runtime profile for a cWB benchmark of interest, which influences our choice of routines benefiting from parallelization and optimization. We described 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. The performance and efficiency of the model is evaluated. We also implement further improvements to fully explore the acceleration of GPU on this model. Some discussions are also given based on experimental runs and timings results.

Keywords: Parallel Computing, GPU, MLR, Gravitational Waves.

1 Introduction

1.1 Gravitational Wave Detection

The 2017 Nobel Physics Prize winning Project Laser Interferometer Gravitational-Wave Observatory (LIGO)'s detections of gravitational waves(GW) have opened a new era in astrophysics. Until now there've been 5 events confirmed of gravitational waves being detected. Detectors are operated around the world, including LIGO in United States, Virgo and GEO 600 in Europe, and KAGRA in Japan [1].

From Initial LIGO to Advanced LIGO(aLIGO), LIGO project has walked through decades, from O1(first observing run) to O2(which ended on August 25, 2017), right now it's time for stepping into O3. The data processing pipelines used in LIGO, which in charge of processing the data detectors collected from sky and analyzing the possibility of existence of gravitational waves in bulks of data, have also evolved a lot during the development of LIGO.

Current data processing pipelines used in LIGO can be generally divided into two categories: online mode and offline mode. The online mode pipelines need to process data online, accepting data sent from detectors and do the first-round processing and judgment of existence of gravitational waves signal in data. Design of this kind of pipelines often focuses on low latency, usually in several minutes. If the online pipeline can't process the data in required time, scientists will risk missing observation of gravitational wave signals. The offline mode pipelines implement detailed analysis on candidate events, in order to get enough physics parameters for further study. This period of data processing could often cost up to days, decided by data scale. The offline mode pipelines are often with high computing cost, large demand of computing resources and strict standard for accuracy. The computing cost for identification of candidate event is negligible, while the analysis on the time-shifted detector data in offline mode should be repeated 10000 or 100000 times on the same data set due to the estimation of the background rates [2], so the time cost of offline pipelines is huge too. No matter which mode the pipeline is, time cost is always the most important issue of design and optimization, since the speed of pipeline can directly influence the science discovery. Usually the optimization of development of pipeline will include both online and offline modes, for example, cWB pipeline has both online all-sky search and all-sky long duration burst search(offline), and they require independent runs of cWB search pipeline (scientists can choose different modes when running cWB pipeline).

Although there're many pipelines being used during LIGO running in O1 and O2, there're difference in demand of use, before and in the future. Top 10 software pipelines by O3 demand are listed in Table 1. The computational need of LIGO data analysis varies enormously by pipelines, thus makes optimization effort largely prioritized by the estimated demands of pipelines. At $0.02/SU, the benefit of optimization is obvious.

Table 1. Top 10 Software Pipelines by O3 Demand [2]

Pipeline	Search Group	Est. O3 Demand(MSUs)	%Total O3 Demand
PyCBC	CBC	65	14.6%
gstlal	CBC	65	14.5%
PowerFlux	CW	48	10.8%
cWB	Burst	31	6.9%
Sky Hough	CW	29	6.5%
Tiger	CBC	25	5.7%
TwoSpect	CW	25	5.6%
E@H Fstat	CW	25	5.6%
LALinference	CBC	24	5.4%
MBTA	CBC	24	5.3%
Total Top 10		360	80.8%

Table 1 shows different pipelines and the research group they belong to. Est.O3 demand shows the estimated demand of computing power in O3 stage, in million Service Units. The last column shows the percentage of demand of specific pipelines to be used in O3.

The previous optimizations of pipelines used in gravitational search focus on cost of running time and computing resources [2]. The seeking of optimization potential usually lies in 6 areas, and the corresponding methods of optimization are listed in Table 2.

Table 2. Methods of Optimization

Name	Examples of implementation
Scientific	change the mathematical model of calculation
Algorithmic	use FFT, BLAS and other optimized code library
Compiler	gcc, icc or native high-performance instructions
Vectorization	use SIMD method, like AVX-512
Multi-core	OpenMP, MPI technique
GPU	use many-thread hardware and high-level libraries

In our work, we focus on the offline mode of cWB pipeline. cWB is an important all-sky search pipeline well known for its fast speed of processing data comparing with other pipelines. It combines data streams from arbitrarily aligned detectors, into a coherent statistic for analysis. It computes a constrained likelihood functional, which is dependent on the source sky position. Based on a constrained likelihood maximization of coherent response for all possible GW signals and for each point in the sky over the detector network, cWB pipeline identifies GW events in collected data and establishes the event significance over three standard deviations.

We consider all 6 aspects and propose a parallel model based on our analysis of cWB pipeline, needs from users and the overall progress of O3.

With much larger expected data to process in O3, the need to fasten cWB pipeline becomes urgent. After carefully analyzed all six aspects of optimization, we believe there's huge potential to optimize performance by releasing parallelism of the pipeline. Although current sequential implementation is nicely accelerated by the use of streaming SIMD extensions (SSE), which supports 4 parallelized operations, we see much larger space for further parallelizing. Attempts to use AVX to improve SSE technique have been proposed, however, we think porting to GPU would be a better choice to undercover the full parallelism of cWB pipeline since it supports much more threads than AVX. Also, since likelihood analysis is widely used in GW searching, our parallel model could be used not only in LIGO, which is ground-based observation, but also in space-based observation of GW, like eLISA in Europe or Taiji in China.

The outline of this paper is as follows. Section 2 describes background of cWB pipelines including mathematical and programming models, and presents our profile result and analysis of cWB pipeline. Section 3 shows features of the GPU hardware

and past studies of GPU acceleration. Section 4 proposes 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, and evaluate the performance of the model. In section 5 we evaluate the performance of the model and implement further improvements to fully explore the acceleration of GPU. The conclusion and future work are given in section 6.

2 cWB pipeline

2.1 Likelihood analysis in a coherent Waveburst pipeline

S. Klimenko *et al* [4] 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 [5], 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:

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

where K is the number of detectors in the network, in cWB pipeline K is not smaller than 2. $\omega_k[i,j]$ is the sampled detector data and $\xi_k[i,j]$ is the detector responses. Standard deviation $\sigma_k[i,j]$ is used to characterize the detector noise. The detection response can be written in standard notations as follow:

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

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

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

$$\omega[i,j] = \left(\frac{\omega_1[i,j]}{\sigma_1[i,j]}, \ldots, \frac{\omega_K[i,j]}{\sigma_K[i,j]} \right)$$

$$f_{+(\times)}[i,j] = \left(\frac{F_{1+(\times)}}{\sigma_1[i,j]}, \ldots, \frac{F_{K+(\times)}}{\sigma_K[i,j]} \right)$$

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

$(w \cdot f_+) = |f_+|^2 h_+$

$(w \cdot f_\times) = |f_\times|^2 h_\times$

where $|f_+|^2$ and $|f_\times|^2$ 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 $|f_\times|^2$. A specific class of constraints (regulators) arising from network responses to a generic GW signal is added to help solve this problem. In coherent WaveBurst analysis, we can change the norm of the $f\times$ vector using a regulator:

$$\left|f_\times^{'}\right|^2 = \left|f_\times\right|^2 + \delta$$

The regulator preserves the orthogonality of vectors f+ and f×. Thus the maximum likelihood statistic is written as:

$$L_{\max} = \sum_{\Omega_{TF}} \left[\frac{(w \cdot f_+)^2}{\left|f_+\right|^2} + \frac{(w \cdot f_\times^{'})^2}{\left|f_\times^{'}\right|^2} \right] = \sum_{\Omega_{TF}} \left[\left(w \cdot e_+\right)^2 + \left(w \cdot e_\times^{'}\right)^2 \right]$$

The solutions of the likelihood functional equation show the GW waveform as two GW components of the solutions can be calculated by:

$$h_+ = \frac{\left(w \cdot f_+\right)}{\left|f_+\right|^2}$$

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

In general, the likelihood function is calculated as a sum over the dataset selected for analysis, which depends on the selected time frequency(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 actually 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 cluster represents a group of pixels selected in a single detector while a group of LTF pixels repre-

sents a coherent trigger. The real data is combined with instrumental and environmental glitches, so additional operations are needed to distinguish 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 [4].

2.2 Profiling

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; (4) calculation of the detection statistic, sky location and reconstruction of burst waveforms [6].

We ran the pipeline on various platforms with different compilers, listing total runtimes, for a cWB benchmark in Table 3 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.

Table 3. Overall runtime comparisons on a single core

Platform	Compiler	Clock Speed	Runtime
Xeon Phi 7210	ICC	1.30GHz	4:47:31
Xeon Phi 7210	GCC	1.30GHz	4:34:06
Xeon E5-2670	ICC	2.60GHz	1:21:03
Xeon E5-2670	GCC,NVCC	2.60GHz	1:20:12

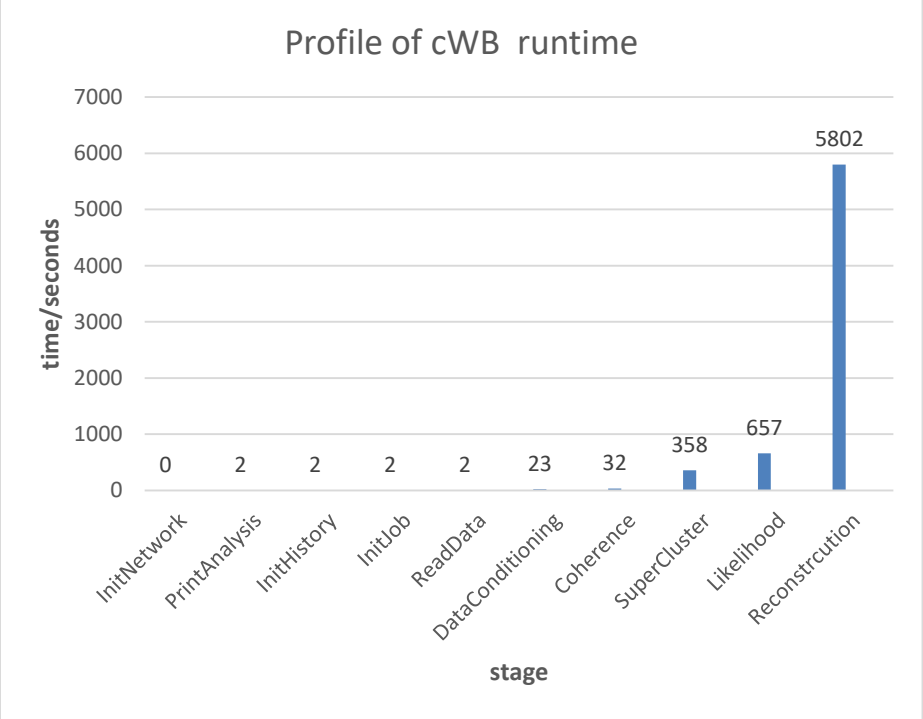

Fig. 1. Time cost of different stages during sample cWB run

Figure 1 shows the Likelihood and Reconstruction stage, which both are important roles in likelihood analysis, are the most time-consuming stages in the whole pipeline. Our optimization focuses on these two stages.

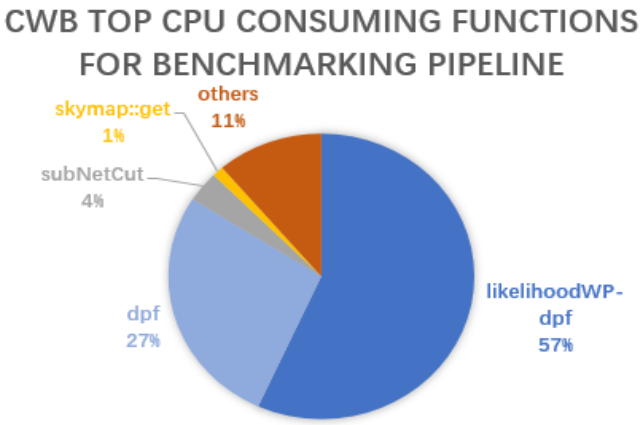

Fig. 2. Breakdown of top CPU consuming functions

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 likelihoodWP stage, where the time-frequency analysis, time-shift analysis and the search over nearly 200,000 sky locations are performed.

2.3 Reconstruction-Sequential MLR

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

Algorithm 1. loops of MLR calculation

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

In Algorithm 1, 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. From Algorithm 1 we can see that these three layers of loops are well prepared for parallel models. However, the scales of loops prevent us using same parallel strategy. The third layer has a fixed scale of 196608 sky locations, which suits for GPU porting, while the second layer has a varying scale of 10 to 50 clusters, which is too small for GPU porting, while not suitable for multi-core CPUs because the cost for communication and scheduling cost for the varying need of CPUs. We decide to only optimize the third layer in this project.

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 Algorithm 2, as follows:

Algorithm 2. Sequential MLR calculation

```
for l = 1,2, ..., 196,608 do          // 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])
```

The original algorithm for all-sky searching is sequential nature, not readily suitable for explicit data parallelism. An initial OpenMP port with 256 threads was attempted with bad performance achieved for bad communication cost and low level parallism. It seems significant data structure and implementation changes would have been necessary. Considering SSE structure embedded in original sequential algorithm, which is not supported on GPUs, it is necessary to transform SSE code back to a simple float version before porting this code to the GPU.

3 GPGPU computing

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 [8].

With the release of the compute unified device architecture (CUDA) [9] 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.

4 Parallel MLR model and GPU implementation

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 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. The original algorithm on a CPU is sequential, with one thread calculating for all sky locations in a huge loop and running at a time. In our parallel model, all threads run simultaneously, and one thread for one sky location. Then we compare the computed result and send back the max one with its index.

Significant changes to data structures were made in order to allow the parallelization of the MLR computation and release best performance. Algorithm 3 outlines the revised MLR computation, with routines running on the GPU designated in italics:

Algorithm 3. Parallel MLR

```
// Initialize GPU and copy input data to GPU
for L (l = 1,2…196,608) do
                                  // 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])
SelectCluster();
sum();
```

Algorithm 3 enables fully parallelism of the sky location layer. It removes dependencies between sky locations which exist in Algorithm 2, and reduce the memory cost by combining GPU operations together. In implementation, we considered several GPGPU computing techniques such as block design, CUDA stream, atomic operations, and we propose the final program with the best runtime performance.

5 Experiment and Analysis

The benchmark data set in this test is provided by S. Klimenko *et al* [], 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.

In detailed implementation of our model, we considered following aspects to achieve best performance with our Tesla P100 GPU.

1) Data access. To reduce times for data transmission between CPU and GPU, we allocate enough space on GPU in advance to transmit all data only once within one function. We also change the data structure to gather all functions implemented on GPU together, so we can reuse the memory on GPU.
2) According to Tesla architecture we carefully design the block size and thread size. In order to hide latency or delayed instructions, each SM needs to maintain at least 6 active warps. Also the thread size should be multiple of size of warp, which is 32. Luckily the location number is 196608 so we can easily achieve that without considering mod. Detailed analysis is shown in Table 3 for choosing the appropriate size.
3) We didn't use shared memory because we use 196608 threads to compute at the same time, and we can't afford such huge shared memory. Also, we didn't use CUDA streams because the order of operations on hosts(CPU) and device(GPU) is strictly sequential, we need all computation on GPU finished and then we can go back to CPU in each cluster. If we use CUDA streams, it would cause false dependencies problem and the task may block.
4) Atomic operations turn out unnecessary here, since our paralleled data structure solve the lock problem naturally.

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 4. SSE(sequential) times are for the SSE sequential version, running solely on the CPU.

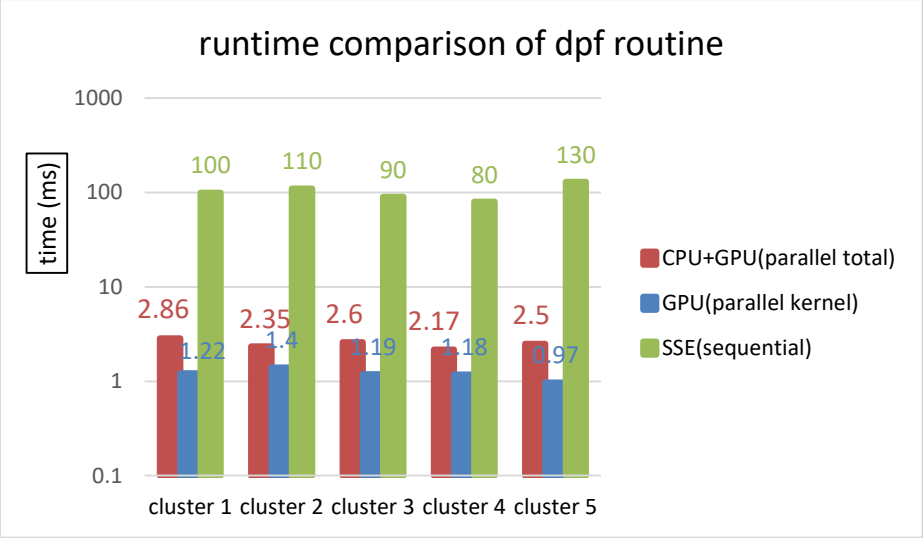

Fig. 4. Runtime comparison of dpf function, GPU vs CPU vs hybrid for lag 0

In Figure 4, 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 times improvement in cluster 5, 130s vs 2.5s. Worse case, a factor of 34 times 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 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.

We then continue to port the whole MLR calculation to the GPU, and we continue to see performance improvement with subsets of the calculation. The following profile of MLR calculation is on the first 3 lags of our dataset (each contains 16,15,12 clusters), and we tried different programs of filtering clusters to compare different design of dimension of blocks and threads. We record the start time and end time of processing clusters' data in these 3 lags with different number of threads per block, which includes the time cost for MLR calculation. We choose to test the number of threads per block with parameter 32,64,96,128 and 256(the ideal design of number of

threads per block is the multiple of 32 due to the structure of GPU), we also compare the time cost with SSE version, the result is shown in table 3:

Table 3. time for stages to finish with different number of threads per block

number of threads per block	MLRCalculationStart-toEnd(mm:ss)	time cost of MLR calculation(second)
256	09:49,10:00	11
128	09:44,09:55	11
96	09:39,09:50	11
64	09:39,09:50	11
32	10:13,10:25	12
SSE	9:27,10:01	34

From table 3, we can see there's no obvious difference between different settings of number of threads per block, which means the time for calculation of 3 lags on GPU is too small to tell the difference, and the level of difference is in millisecond level. After we run through the whole dataset, we choose to set 96 threads per block for its fast speed and stability, and we compare the performance between GPU version and the original SSE version, with detailed time cost on MLR calculation timed. We choose to show part of the timing data from the first lag in figure 5.

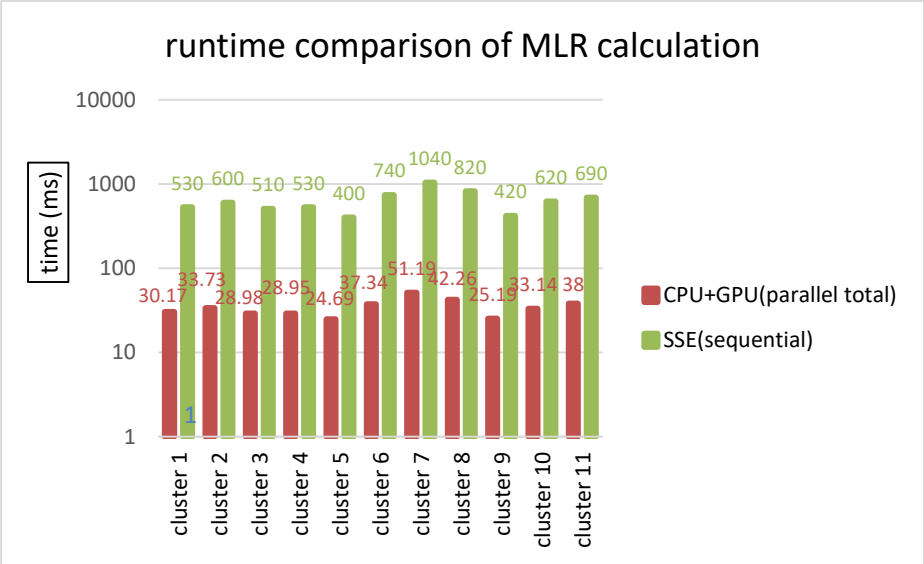

Fig. 5. Runtime comparison of MLR calculation, hybrid vs CPU

As we can see from Figure 5, parallel MLR model shows better performance than the original SSE version, a relatively stable 17.41x average speedup is achievable with Tesla P100 using our optimization method.

Implementation of other accelerating techniques are considered and tested on our model to fully explore performance of GPU. Since the sky location number is 196608, which is multiple of our setting of threads per block, the threads can avoid being idle, which reduces performance lost. The number of synchronization in calculation is largely reduced because of our batched-implemented computational model. We also take full advantage of parallel sum reduction to improve parallelism of our model and eliminate the explicit synchronization. Shared memory is properly used to accelerate memory access and improve the throughput of all threads.

6 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, the parallel MLR model is 1D. As the dataset size growing bigger, the number of lags and clusters will increase, we can modify our model to a 3D version to achieve more parallelism in the future.

References

1.Leo P.Singer, Larry R.Price, Ben Farr, et al, The First Two Years of Electromagnetic Follow-Up with Advanced LIGO and Virgo, 2014 APJ 795 105

2. Peter Couvares, Progress on Data Analysis Computing Optimization

3. Min-A Cho, Updates to advanced LIGO and Virgo's electromagnetic follow-up program APS April Meeting2017

4. S.Klimenko, I.Yakushin, A.Mercer, G.Mitselmakher, Coherent method for detection of gravi-tational wave bursts

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

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

7. SSE Guide, http://sci.tuomastonteri.fi/programming/sse8. CUDA Document, 8.

8. 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

9.CUDA Document, http://docs.nvidia.com/cuda/