Dissertation

IMPLEMENTATION OF N-DETECTORS IN A GRAVITATIONAL WAVE DETECTION PIPELINE

Thomas Hill Almeida (21963144)

Supervisors: Prof. Linqing Wen, Qi Chu

2020-10-09

Contents

1 Introduction

2 Literature Review

2.1 Gravitational Wave Detection

2.2 CUDA

CUDA [6] is an extension of the C++ programming language created by NVIDIA that allows for the development of GPU-accelerated applications. In [3], the SPIIR pipeline had multiple components rewritten in CUDA to take advantage of the high number of simultaneous threads available compared to CPUs. As such, it is worth understanding the computational model of CUDA for the analysis of the coherent post-processing step of SPIIR.

In CUDA, each individual sequence of instructions being executed is called a *thread*. By its nature, a highly-parallelised environment such as GPUs will run many individual threads, which are partitoned into *warps*, a group of (typically 32) threads. Warps are the smallest unit that GPUs schedule, and all threads in a warp must execute the same instruction – although each thread maintains its own instruction pointer and can branch independently from the warp at a small performance cost. The performance cost of branching within a warp means that a major optimization that does not affect computational complexity in CUDA can be simply reducing the number of branches. Warps are further organised into thread blocks, which contain a small amount of fast memory shared between the threads in the block. Blocks in CUDA are typically executed on the same Simultaneous Multiprocessor (SM). The CUDA Programming Guide ([5]) states that the number of blocks and warps that can reside and be processed together on an SM depends on the number of registers and shared memory available on the SM, as well as on a CUDA defined maximum number of blocks and warps.

For the purpose of actual time-based computation, the maximum number of threads that can run at any given time is determined by a few factors of the CUDA runtime; the maximum number of resident warps per SM; the maximum number of resident threads per SM; the number of 32-bit registers per thread; the number of 32-bit registers per SM; the number of 32-bit registers per thread block; and the amount of shared memory in each of those divisions. Thus, one major determining factor in any speed-up given by a CUDA operation can be determined by the ability to split the workload across threads and thread blocks such that the number of registers and used memory is well balanced across threads.

2.3 Parallel Complexity Analysis

According to [4], the theoretical efficiency of a multi-threaded or parallelised algorithm can be measured using the metrics of ‘span’, ‘work’, ‘speed-up’ and ‘parallelism’, all of which should be considered in the context of a directed acyclic graph (DAG) of operations in the algorithm. The ***work*** of a parallelised computation is the total time to execute the entire computation sequentially on a single processor, and can be found by summing the total work of every vertex in the DAG. In comparison, the ***span*** of a parallelised computation is the maximum time taken to complete any path in the DAG. It should be noted that the actual running time of a parallelised computation also depends on the number of processors available for computation and how they are allocated to perform different tasks in the DAG, and thus denoting the running time of parallelised computation on P processors as T_P is also common practice. This leads to work being denoted at T_1 (the time taken to run on a single processor) and span being denoted as T_∞ (the time taken on an infinite number of processors). Another helpful metric is ***speed-up***, which shows how the algorithm scales with additional processors as $S_P = \dfrac{T_1}{T_P}$. We can also then define ***parallelism*** as the maximum possible speed-up on an infinite number of processors, and thus as $p = \dfrac{T_1}{T_\infty}$.

Using the above definitions, we can re-derive several laws that provide lower bounds on the running time of T_P.

In one step, a computer with P processors can do P units of work, and thus in T_P time can perform PT_P units of work. As the total work to be done as per above is T_1, the ***work law*** states that [4]:

$$T_P \geq \frac{T_1}{P}. \tag{1}$$

It is also evident that a computer with P processors cannot run any faster than a computer with an infinite number of processors, as the computer with an infinite number of processors can emulate a computer with P processors by using a subset of its processors, leading to the ***span law*** [4]:

$$T_P \geq T_\infty. \tag{2}$$

It is also useful to use the metrics of 'cost' and 'efficiency' when analysing parallel algorithms [1]. The ***cost*** of a parallel algorithm is minimised when all of processors are used at every step for useful computation and thus can be defined as $C_P = P \times T_P$. ***Efficiency*** is closely related to cost and describes speed-up per processor and can be defined as:

$$e_P = \frac{S_P}{P} = \frac{T_1}{C_P}. \tag{3}$$

Another helpful theorem for analysis is ***Brent's Theorem***, which states that for an algorithm that can run in parallel on N processors can be executed on $P < N$ processors in a time of approximately [2]

$$T_P \leq T_N + \frac{T_1 - T_N}{P}. \tag{4}$$

This can be approximated with the upper bound of $O(\frac{T_1}{P} + T_N)$ [1].

Determining the span, work, parallelism, efficiency and cost, and examining the application of Brent's theorem to the computations at hand will allow us to analyse the computational complexity of the SPIIR pipeline.

3 A complexity analysis of the parallel post-processing of the SPIIR pipeline

3.1 Motivation

3.2 Maximum reduction with index preservation

One of the more common operations in the SPIIR pipeline is the concept of a "maximum reduction with index preservation". Reduction is the idea of taking some array of data and producing a single summary output from that array, whether it is the total sum of the array or the maximum value of the array and its index in the array as it is in this case.

[7] discusses the computational complexity of reduction algorithms in a parallelised context, noting that the best complexity according to Brent's Law is $O(\frac{N}{\log N})$ threads each doing $O(\log N)$ sequential work, resulting in a total overall cost of $O(\frac{N}{\log N} \times \log N) = O(N)$.

We can note from our own analysis, that the process of reduction can be parallelised by the use of a binary tree of operations, where each vertex in the binary tree combines the results of the two parent vertices. In the case of determining the maximum of two numbers, each vertex is identical in the amount of work done, and thus we can determine each vertex to be a unit of work. As there are N elements in the original array, we can note that the height of the binary tree is $\log N$, and each level of the binary tree has $N_l/2$ vertices, thus the total number of vertices in the binary tree is $\sum_{i=0}^{\log N} 2 \times i = N$. Using this information, we can determine that the ***work*** of a parallelised reduction is $T_1 = O(N \times 1) = O(N)$, and that the ***span*** of the reduction is $T_\infty = O(\log N \times 1) = O(\log N)$. Thus, the ***parallelism*** of the reduction is:

$$p = \frac{N}{\log N}.$$

Using the span and work laws, we can observe that any algorithm using the above method is bounded by the inequalities $O(\log N) \leq T_P$, $\frac{O(N)}{P} \leq T_P$. This means that best possible time complexity with P processors is $O(\log N)$ (equation 2). We can determine the minimum number of processors required to achieve this runtime using the formula $T_P = O(\log N) = \frac{O(N)}{P}$, which can be rearranged to

$$P = \frac{N}{\log N},$$

thus the time complexity cannot improve past $P = N/\log N$ processors. We can also observe that using $P = N/\log N$ processors gives a ***cost*** of $C_P = N$, which is identical to the sequential algorithm.

Functions that include maximum reduction with index preservation will be denoted for clarity with $M(x)$, where x is the size of the array being reduced.

3.3 Determining the number of samples over a signal-to-noise threshold

The coherent post-processing in SPIIR determines the number of samples over a signal-to-noise (SNR) threshold in order to not do more work than is required. The function that is used for determining the number of samples over the threshold (`peaks_over_thresh`) is a sequential algorithm that runs on the CPU, and shall be analysed as such, although there is an alternative GPU-based implementation that is not used.

Initially, the function performs a maximum reduction with index preservation to get the maximum SNR from the combined IIR filters (templates) for each sample. Recalling from section 3.2 that for maximum reduction with index preservation $T_1 = O(N)$, and that this operation is performed S times, where S is the number of samples, we can determine that this initial reduction has a time complexity of $O(ST)$, where T is the number of templates.

The function then determines the maximum SNR across the templates found from the previous step by stepping through every combination of samples and removing SNR samples that are using the same template and have a lower SNR, resulting in a step with a time complexity of $O(S^2)$.

The function then determines the maximum overall SNR for the input samples ($O(S)$) and cycles through every maximum SNR to cluster maxima that are close together to be a single combined maximum. The number of maxima is bounded by ($O(\min\{S, T\})$) as there cannot be more maxima than there are samples or templates.

This gives the overall function a time complexity of $O(ST + S^2 + S + \min\{S, T\})$, which can be reduced to the dominating terms of:

$$O(ST + S^2).$$

3.4 Transposing the input matrices

The full post-processing function requires that the input matrix is transposed for better memory access such that each row is a different template, and each column is a different sample. To transpose the matrix, the GPU function `transpose_matrix` is used, thus this should be analysed as a parallel algorithm.

The algorithm in use works by breaking the original array into tiles of size 32×32, and then inserting the transpose of the tile into an output array. The tiles are further broken down eight processors per row, so each thread does four copies. We can conceptualise this as a DAG by observing that each tile does not depend on any other tile to be completed, and that each tile is composed of 32×8 interdependent processors, each doing 4 units of work.

Using this observation, we can see that the ***span*** of the algorithm is $T_\infty = (32 \times 8) \times 4 = O(1024) = O(1)$, and the ***work*** is $T_1 = O(ST)$, where S is the number of samples and T is the number of templates. Thus, the ***parallelism*** of the transpose is $p = ST$.

Using the span and work laws (equations 2 and 1), we can observe that the above method is bounded by the inequalities $O(1) \leq T_P$, $\frac{O(ST)}{P} \leq T_P$. Thus it can be determined that the best possible time complexity with P processors is bounded by ratio of available processors to the size of the transposed matrix (the work law). This gives the function an overall time complexity of:

$$O(\frac{ST}{P}).$$

3.5 Determining the coherent correlation and statistical value of data points

The scoring metric of different templates and times is determined using coherent correlation and determining their statistical value using a chi squared-based distribution. These scoring metrics are performed using the GPU function `ker_coh_max_and_chisq_versatile`, and thus should be analysed as a parallelised function.

In this function, each block looks at a different SNR maximum (as discussed in section 3.3) and splits the threads within the blocks for operations on that peak.

3.5.1 Determining the sky direction of the SNR maximum

Initially, each thread within a block looks at a different sky direction and determines the total signal-to-noise ratio (SNR) by summing the SNR of each of the detectors at that given sky direction with the relevant detector arrival time offsets. The time complexity for the calculation of SNR for a given time offset is $O(D + D^2)$, where D is the number of detectors. The maximum SNR for all the sky directions is then spread across each warp and placed into shared memory before being shared across every thread in the block, which is an application of the parallelised maximum reduction with index preservation function discussed in section 3.2.

Thus, the ***span*** of determining the sky direction with the highest signal to noise ratio is $T_\infty = O(D + D^2 + M_{T_\infty}(S))$ and the ***work*** is $T_1 = O(S(D + D^2) + M_{T_1}(S)$, where S is the number of sky directions and $M(x)$ is the complexity of the parallelised maximum reduction with index preservation function. We can further state that the ***parallelism*** of this is equivalent to the number of sky directions, $S + S/\log S$.

3.5.2 Calculating signal consistency statistics

After having determined the sky direction with the highest SNR for a given maximum, the function then calculates a signal-morphology based statistic ξ_D^2 for each detector D. The statistic is a reduced χ^2 distribution with $D \times 2 - 4$ degrees of freedom and a mean value of 1, and is given in the discrete form by:

$$\xi_D^2 = \frac{\sum_{j=-m}^{m} |\varrho_D[j] - \varrho_D[0]A_D[j]|^2}{\sum_{j=-m}^{m}(2 - 2|A_D[j]|^2)}, \tag{5}$$

where ϱ is the coherent SNR, A_D is the vector of the correlation of the given template with the output from the detector and $2 \times m$ is the number of samples.

The numerator of the statistic is calculated by splitting the number of samples between the threads of a block, followed by combining the results of the statistic across each warp and then each block. The combination of the statistic across each warp and block is a modification of the parallelised maximum reduction with index preservation discussed in section 3.2 that uses addition instead of maximum as the combining binary function. Thus the ***span*** of calculating the statistic is $T_\infty = O(D \times M_{T_\infty}(N))$ and its ***work*** is $T_1 = O(D \times M_{T_1}(N))$, where N is the number of samples. We can then state that the ***parallelism*** of calculating the statistic is equivalent to the parallelism of the reduction, $O(N/\log N)$.

3.5.3 Generating time-shifted background noise statistics

The function then performs a number of time shifts on background noise for use with the significance estimation. The generation of a single background statistical variant is equal to the total work of the function so far, save that instead of using blocks for every peak, each warp looks at a different time shift. Thus, whilst the theoretical time complexity does not change, the number of processors available is smaller, so the actual runtime each loop is approximately the warp size slower.

3.5.4 Overall computational cost

Overall, this function has a ***span*** of $T_\infty = 2(D + D^2 + M_{T_\infty}(S) + DM_{T_\infty}(N))$, and has $T_1 = P(S(D + D^2) + M_{T_1}(S) + DM_{T_1}(N) + B(S(D + D^2) + M_{T_1}(S)))$ ***work***, where P is the number of SNR maxima and B is the number of times shifts made to background noise.

3.6 Calculating heat skymaps

If the coherent SNR exceeds a threshold, the post-processing produces a skymap of the highest SNR in the GPU function `ker_coh_skymap`.

The function determines the highest maximum SNR by using the maximum reduction with index preservation technique discussed in section 3.2. Following this, the function re-performs the process discussed in section 3.5.1 with additional sky directions and without the reduction to generate the final skymap.

As such, this function has a ***span*** of $T_\infty = M_{T_\infty}(P) + D + D^2$ and total ***work*** of $T_1 = M_{T_1}(P) + S(D + D^2)$.

4 Conclusion

The total span and work of the coherent post-processing step in the SPIIR pipeline is the sum of the total spans and works of the internal functions. Conversely, we cannot determine the overall parallelism as the post-processing step spans a number individual functions that can each be run with a different set of processors. As the step to determine the number of peaks over a threshold (see section 3.3) is sequential, we can consider its time complexity as contributing to both the span and work of the total pipeline. Another thing to note is that the steps for determining the coherent correlation, statistic value and skymaps (sections 3.5 and 3.5.1) will be run for every detector.

With this in mind, we can determine that the ***span*** of the post-processing is:

$$\begin{aligned} T_\infty &= O(NT + N^2 + 1 + D(2(D + D^2 + \log S + D\log N) + \log P + D + D^2)) \\ &= O(NT + N^2 + D^3 + D^2\log N + D\log S + D\log P), \qquad (6) \end{aligned}$$

where D is the number of detectors, S is the number of sky directions, T is the number of templates, N is the number of samples and $P = \max\{S, T\}$.

The total ***work*** of the post-processing is:

$$T_1 = O(NT + NT + S^2 + D(P + S(D + D^2) + P(S(D + D^2) + S + DN + B(S(D + D^2) + S))))$$

$$= O(NT + N^2 + SPD^3 + SPBD^3 + ND^2), \tag{7}$$

where D is the number of detectors, S is the number of sky directions, T is the number of templates, N is the number of samples, B is the number of is the number of times shifts made to background noise and $P = \max\{S, T\}$

References

[1] Henri Casanova, Arnaud Legrand, and Yves Robert. *Parallel Algorithms*. eng. CRC Press, 2008, pp. 10–12. DOI: `10.1.1.466.8142`.

[2] John L. Gustafson. "Brent's Theorem". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 182–185. ISBN: 978-0-387-09766-4. DOI: `10.1007/978-0-387-09766-4_80`. URL: `https://doi.org/10.1007/978-0-387-09766-4_80`.

[3] Xiaoyang Guo et al. "GPU-Optimised Low-Latency Online Search for Gravitational Waves from Binary Coalescences". eng. In: vol. 2018-. EURASIP, 2018, pp. 2638–2642. ISBN: 9082797011. URL: `https://ieeexplore.ieee.org/document/8553574`.

[4] Thomas H. Cormen et al. *Introduction to algorithms*. eng. 3rd ed. MIT electrical engineering and computer science series. Cambridge, Mass: MIT Press, pp. 779–781. ISBN: 0070131430.

[5] *CUDA Programming Guide*. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-multithreading`.

[6] *CUDA Toolkit*. URL: `https://developer.nvidia.com/cuda-toolkit`.

[7] Mark Harris. *Optimizing Parallel Reduction in CUDA*. eng. URL: `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.