Masters Dissertation

IMPLEMENTATION OF N-DETECTORS IN A GRAVITATIONAL WAVE DETECTION PIPELINE

Thomas Hill Almeida (21963144)*

Supervisors: Prof. Linqing Wen†, Qi Chu†

2020-10-18

*Software Engineering, University of Western Australia

†Department of Physics, University of Western Australia

Contents

1 Introduction

Gravitational waves have been postulated to exist since Albert Einstein's publication of his general theory of relativity, as massive accelerating objects would cause 'ripples' in the curvature of spacetime [1]. Direct detection of gravitational waves, however, remained beyond the reach of the scientific community until 2015, when the Laser Interferometric Gravitational-Wave Observatory (LIGO [see 2]) reported a direct observation of a gravitational wave on the 14th of September [3, 4].

Due to their design, the detectors in use for gravitational wave detection are affected by a significant amount of noise from other sources, whilst the gravitational waves themselves have very weak signals. As such, a large amount of data processing must be done to the outputs produced by the detectors in order to filter and extract any possible gravitational waves. These data processors are known as "pipelines", and have historically been created by research groups that are a part of the LIGO Scientific Collaboration (LSC [see 5]), and are used throughout observation runs for real-time data analysis.

The Summed Parallel Infinite Impulse Response (SPIIR [see 6]) pipeline, based on the SPIIR method originally implemented by Shaun Hooper in 2012, uses a number of IIR (infinite impulse response) filters to approximate possible gravitational wave signals for detection [7]. The output of the ith IIR filter can be expressed with the equation [8]:

$$y_k^i = a_1^i y_{k-1}^i + b_0^i x_{k-d_i}, \tag{1}$$

where a_1^i and b_0^i are coefficients, k is time in a discrete form and x_{k-d_i} denotes input with some time delay d_i. After summing the output of the filters, the resulting signal undergoes coherent post-processing (see section 5.1 and [9, chapter 4]) to determine the likelihood of an event having occurred.

The pipeline is currently thought to be the fastest of all existing pipelines, is the only pipeline that implements coherent search, and has participated in every observation run since November 2015, successfully detecting most of the events seen in more than one detector.

The SPIIR pipeline uses GStreamer, a library for composing, filtering and moving around signals, in addition to the GStreamer LIGO Algorithm Library (`gstlal`) [10]. After receiving data from the detectors, the pipeline performs data conditioning and data whitening, followed by the usage of the IIR filters. The data is then combined for post-processing, where events are given sky localization and then inserted into the LIGO event database [8].

The structure of the SPIIR pipeline can be seen in figure 1.

At the time of the start of this research, the SPIIR pipeline supported the use of two or three detectors for gravitational wave detection — the two American LIGO detectors and the Italian Virgo detector — although additional interferometers are likely to be introduced soon. This presents several issues with the existing pipeline design. As with many of the other gravitational wave detection pipelines, providing support for additional detectors is a significant undertaking for the development team, with many hours of work and testing needing to be completed. As the number of available interferometers continues to grow, development work that could be spent on improving the optimisation, precision, or accuracy of the pipeline would instead have to be spent allowing for those detectors to be used.

Section 2 shall explore the existing literature on the implementation of N-detectors in other gravitational wave detection pipelines, in addition to exploring CUDA and complexity analysis, two tools that will be used in the analysis of the final design. Section 3 will look at the existing pipeline structure and its deficiencies for implementing N-detectors. It will also determine any constraints a design for implementing N-detectors would have, and will provide a framework for evaluating the success of the resulting design. The final design will be presented in section 4, and will explore the implementation details of the design and how it interacts with the existing programming interface that SPIIR provides. A discussion about the implications of this design will be done in section 5, with additional research on those implications being presented there. Finally, section 6 will provide some suggestions for future research that can be based on this research.

This dissertation aims to layout the design and implementation work done to provide the SPIIR pipeline with the ability to support any number of detectors without hindering the functionality of the pipeline (N-detectors).

2 Literature Review

This section explores the existing research that is relevant to this project. Section 2.1 shall explore the implementations that other gravitational wave detection pipelines have used to deal with with a growing number of detectors in their algorithms and interfaces, whilst section 2.2 will explore the computational model of CUDA, the library that SPIIR uses for parallelism. Finally, section 2.3 shall explore the literature of parallel complexity analysis, which will allow for the later analysis of the pipeline in section 5.1.

2.1 N-detector work in other gravitational wave detection pipelines

There are, of course, other gravitational wave detection pipelines that may also have to consider the issue of dealing with a growing number of detectors. As we are in the process of designing a new architecture to allow for any number of detectors to be used, it is well worth examining the methods that the other pipelines may use to shape our own design.

Most detection pipelines use a "coincidence" search to determine whether a gravitational wave event has occurred. Coincidence search, is defined as a process which includes finding candidates for gravitational waves from individual detectors, identifying temporal coincidences and producing measures to rank candidate events [9, chapter 3]. In more simple terms, a coincidence search considers events which can be seen in a single detector, and then sanity checks that other detectors may have seen them at the same time.

In contrast, the SPIIR pipeline uses a "coherent" search to determine whether a gravitational wave event has occurred, using the maximum likelihood ratio principle to consider specific parameters of a potential signal [9, chapter 4]. As such, it is unlikely that all of the principles for dealing with additional detectors may translate directly to being able to be used for the SPIIR pipeline, as the method of search differs.

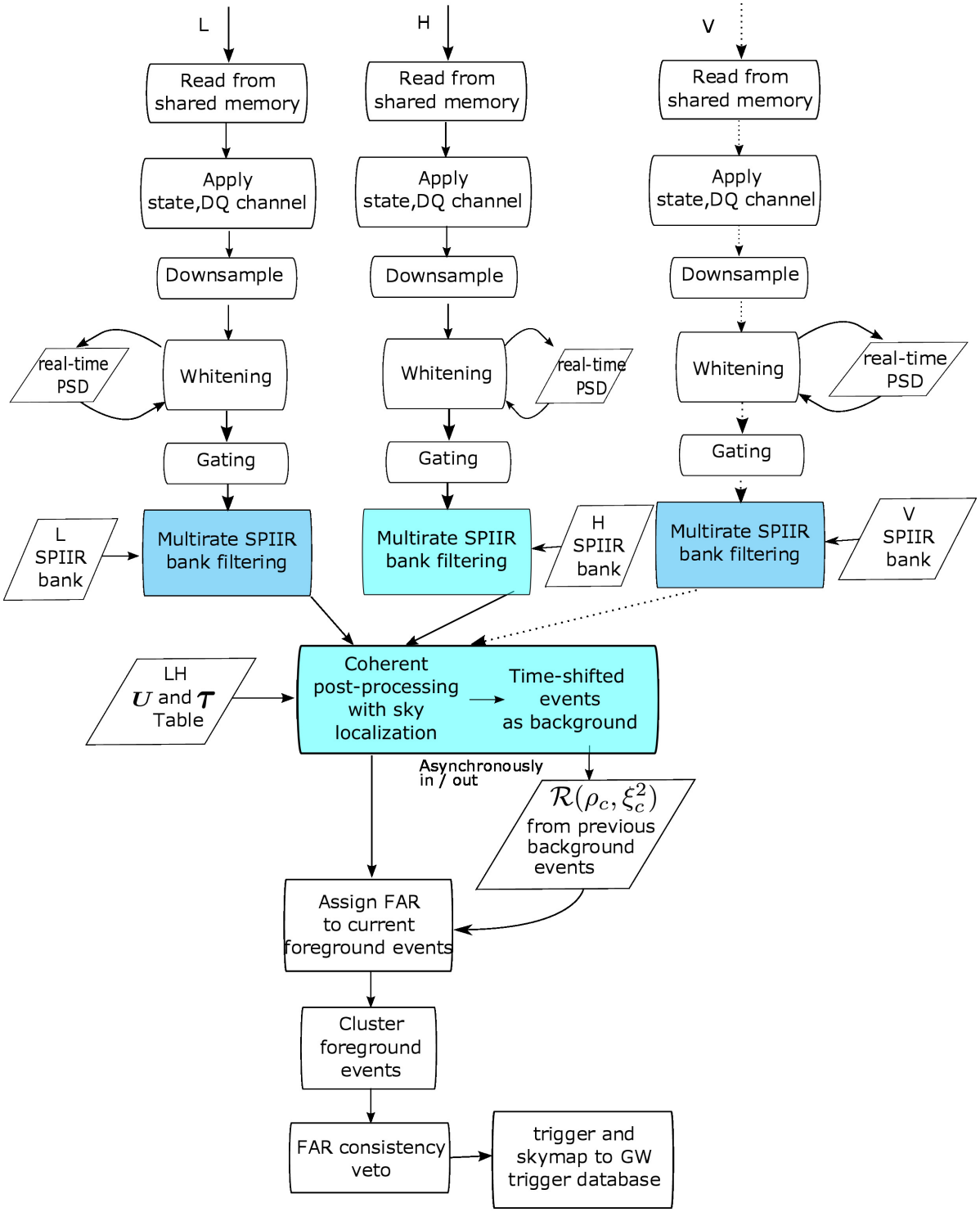

Figure 1: The structure of the SPIIR pipeline

PyCBC is one of the best known toolkits for gravitational wave astronomy, and was one of the pipelines used in the original 2015 gravitational wave detection [11]. From an examination of the codebase of the latest version of PyCBC [12, October 2020], it can be observed that the codebase itself makes no direct mention of detectors — instead it provides a generic `Detector` class as a wrapper around LALSuite's [13] `LALDetector` structure for validation, which in turns provides utilities for returning information about the detector as well as methods for getting readings from it.

This allows PyCBC to provide an entirely generic gravitational wave searching algorithm library for any input detectors, although the algorithm only supports using two detectors at a time, providing that the detectors are in the LALSuite library. Thus, we can note two things; for PyCBC to allow for additional detectors to be used, they simply update their dependency on the LALSuite library, and; PyCBC doesn't quite support N-detectors in the sense described in section **??**, instead it allows for any supported detectors to be used in a two-detector search. This means that whilst we cannot use PyCBC when considering how to support any number of detectors within a gravitational-wave searching algorithm nor its outputs, we can use PyCBC to consider a programming interface with which to support other detectors.

GstLAL is gravitational wave detection library, that exposes components of LALSuite [13] as GStreamer elements for use in other analysis pipelines — including the SPIIR pipeline — as well as providing its own pipeline for processing raw signals from detectors into lists of gravitational wave candidates [14]. The GstLAL's pipeline hard-codes the detector names into both its inputs and its outputs, however the algorithm used for detection itself is actually generic on which detectors are used [15, 16].

This means that the process of adding support for detectors involves changing a number of different files, as well as modifying several internal data structures [17] — which means that GstLAL does not support N-detectors.

Thus any work done to design N-detector support for the SPIIR pipeline will be novel.

2.2 CUDA

CUDA [18] is an extension of the C++ programming language created by NVIDIA that allows for the development of GPU-accelerated applications. In 2018, the SPIIR pipeline had multiple components rewritten in CUDA to take advantage of the high number of simultaneous threads available compared to CPUs [8]. As such, it is worth understanding the computational model of CUDA for the analysis of the SPIIR pipeline.

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 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 [19].

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 so that the number of registers and used memory is well balanced across threads.

2.3 Parallelised Complexity Analysis

As the pipeline changes to accommodate additional detectors, it is important that the impact that this has on the pipeline's runtime is considered. The SPIIR pipeline is designed to be as low latency as possible, and an asymptotic complexity analysis of components that may be impacted by the addition of new detectors allows for the measurement of the potential runtime cost of doing so. The SPIIR pipeline has been parallelised using CUDA [8], and thus determining the asymptotic complexity of components of the pipeline requires different considerations to that of a sequential program.

According to [20], 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. An example of ***work*** for a merge-sort like algorithm can be seen in figure 2, which has a work of $O(N \log N)$. In comparison, the ***span*** of a parallelised computation is the maximum time taken to complete any path in the DAG. An example of ***span*** for a merge-sort like algorithm can be seen in figure 3, which has a work of $O(N)$. 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 $S_P = \frac{T_1}{T_P}$. We can also then define ***parallelism*** as the maximum possible speed-up on an infinite number of processors, and thus as $p = \frac{T_1}{T_\infty}$.

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 [20]:

$$T_P \geq \frac{T_1}{P}. \tag{2}$$

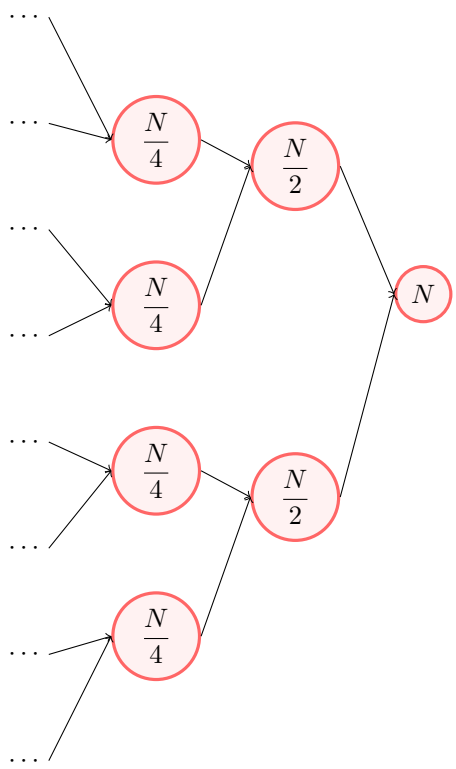

Figure 2: An example of calculating work for a merge-sort like algorithm. Summed nodes are in red.

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*** [20]:

$$T_P \geq T_\infty. \tag{3}$$

It is also useful to use the metrics of 'cost' and 'efficiency' when analysing parallel algorithms [21]. 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:

$$e_P = \frac{S_P}{P} = \frac{T_1}{C_P}. \tag{4}$$

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 [22]:

$$T_P \leq T_N + \frac{T_1 - T_N}{P}. \tag{5}$$

This can be approximated with the upper bound of $O(\frac{T_1}{P} + T_N)$ [21].

Determining the span, work, parallelism, efficiency and cost, and examining the application of Brent's

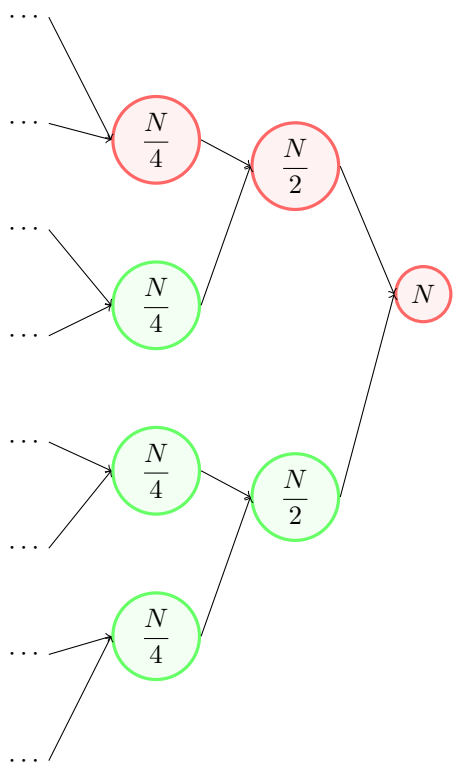

Figure 3: An example of calculating span for a merge-sort like algorithm. Summed nodes are in red.

theorem to the computations at hand will allow us to analyse the computational complexity of the SPIIR pipeline.

3 Design process

This section aims to layout and discuss the design process used to arrive at the final design of the N-detector implementation. Section 3.1 will discuss the various design constraints that should be factored in any design implementation, and section 3.2 will discuss the tools used throughout the design process. The code sections that will be impacted by the implementation will be explored in section 3.3, and a framework for evaluating the success of the implementation will be discussed in section 3.4.

3.1 Design Constraints

In the process of designing any system or any modifications thereof, constraints on that design must be considered and factored in. This section aims to lay out and discuss the various design constraints that should be imposed on any final design.

The below constraints are ordered in importance.

- **Output results must be unchanged**
  A major measure of a gravitational-wave detection pipeline is its accuracy. Any design should not impact or modify the output results of the existing pipeline on the same data.

- **The output table format should be unchanged**
  Events that are detected by the SPIIR pipeline are uploaded to the gravitational-wave candidate event database (GraceDB) using a unified table format to represent event data [23]. It is of critical importance that the SPIIR pipeline is still able to interact with GraceDB so that it can continue to have detected events considered by the wider gravitational-wave community. In addition, the pipeline outputs a number of files that are automatically scanned to ensure that the pipeline is functioning correctly. The format of these output files should not change.

- **Adding new detectors should be able to be done by a non-technical individual**
  The wider gravitational-wave detection community is largely composed of physicists as opposed to software engineers. As such, any changes to the number of detectors are likely to be able to be best tested by non-technical members of the gravitational-wave community, as the expertise to ensure that the values output by the pipeline are correct are more likely to come from physicists. Any design for N-detector support should ensure that the process of adding detectors should be easy enough to be completed and tested by a non-technical individual.

- **The external interface of the pipeline should not change**
  There are many developers and scientists that are using the existing SPIIR pipeline and use customised scripts that automate gravitational wave searches. The ability to support N-detectors should be a largely internal change, without needing to modify the way that other users interact with the pipeline and force modifications to existing search scripts.

- **Individual detectors should still be exposed in Python code**
  There are several reasons for this constraint to be factored into any implementation. First, exposing individual detectors in the Python code allows for easier programming debugging of any values that may be incorrect, as the values of each detector would be immediately evident by the variable names instead of having be indirectly referenced by knowledge of the data structure used to hold the group of detectors. In addition, the Python code that interacts with GraceDB iterates through the list of detectors and uses the individual detector names to collect the data required for each event. By exposing individual detectors to the Python code, none of the code that interacts with GraceDB would need to change.

3.2 Employed tools

There are a number of tools that were employed in the process of designing and testing the N-detector implementation. These were:

- **OzStar [24]**
  "OzStar" is a computing cluster hosted at the University of Swinburne for use in gravitational-wave discovery and theoretical astrophysics. Much of the SPIIR development process occurs on OzStar, and its head nodes were used for both building and testing the modifications made to the SPIIR pipeline.

- **LIGO DataGrid (LDG) [25]**
  The LDG is the collection of clusters that the LSC uses for running the various gravitational-wave

detection pipelines on live data. Testing and benchmarking (see section **??**) were performed on the LDG as the available nodes there closely mirror those that the SPIIR pipeline will run on.

- **SPIIR scripts [26]**
  "SPIIR scripts" is a collection of scripts maintained by Patrick Clearwater for use in the SPIIR development process. The `build_spiir` script from this collection was used to build the modifications made to the pipeline.

- **GWDC utils [27]**
  "GWDC utils" is a collection of utilities maintained by Alex Codoreanu that are written to perform various tasks for the SPIIR pipeline. The unit tests that are exposed as a part of GWDC utils were used for testing that the modifications made to the pipeline didn't affect its output (see section 3.1).

- **OzGrav Research utils [28, `utils/`]**
  "OzGrav Research utils" is a collection of utilities created by Thomas Almeida to automate the process of creating callgraphs from C or CUDA code on the OzStar cluster. These were used in the generation of callgraphs which were used for the complexity analysis in section 5.1.

3.3 Relevant code

The relevant code sections that may need to be modified to support any number of detectors can be observed in figure 1 as any step after the individual detector SPIIR filtering is combined. This includes:

- Coherent post-processing with sky localization
  [6, `gstlal-spiir/gst/cuda/postcoh/`]

- $\mathcal{R}(\rho_c, \xi_c^2)$ from previous background events
  [6, `gstlal-spiir/gst/cuda/cohfar/cohfar_accumbackground.c`]

- False Alarm Rate (FAR) assignment to foreground events
  [6, `gstlal-spiir/gst/cuda/cohfar/cohfar_assignfar.c`]

- Trigger and skymap to GW trigger database
  [6, `gstlal-spiir/python/pipemodules/postcoh_finalsink.py`]

In addition, the data structures that are passed between the components also need to be modified. This includes the `PostcohInspiralTable` [6, `gstlal-spiir/python/pipemodules/postcohtable/`] and the `PeakList` [6, `gstlal-spiir/gst/cuda/postcoh/postcoh.h`].

These files are mixed between several different languages; C, CUDA (see section 2.2), and Python. As such, any N-detector implementation must consider the ability to:

- **Move data structures easily between GPU and CPU memory**
  CUDA provides two interfaces for allocating and managing memory across a GPU and CPU; managed memory and memory copies. Managed memory automatically propagates changes to allocated memory in a GPU or CPU to the other address space, whilst memory copies copy a block of memory from one address space to another.

The existing SPIIR pipeline uses memory copies for transferring data to and from the GPU address space, and as such the implementation of N-detectors should ensure that the existing memory copies will still function as expected. This means that flat data structures would be generally preferred, as they can be copied with a single memory copy.

- **Interoperability between Python and C code**
  The `PostcohInspiralTable` is a C data structure that exposes a Python interface and the final trigger generation and uploading to GraceDB takes the data structure as an input. As such, it is vital that any changes to the `PostcohInspiralTable` are able to be used in the Python interface that it exposes.

3.4 Evaluation criteria

The success of a design and implementation needs to be evaluated according to some predefined criteria. The final design will be evaluated according to these criteria in section 4.3.

1. Output results must be unchanged
2. The output table format should be unchanged
3. Adding new detectors should be able to be done by a non-technical individual
   (a) Maximum number of required edited files: 3
4. Unchanged external interface
5. Exposing of individual detectors in Python
6. Changing number of detectors compiles correctly

4 Final Design

This section aims to describe and expand on the final design and implementation chosen for allowing the SPIIR pipeline to work with any number of detectors.

Section 4.1 discusses the major components and changes that each source code patch makes to the codebase, and the testing that the patches went through will be described in section 4.2. The final design and implementation will be evaluated according to the criteria laid out in section 3.4 in section 4.3.

4.1 Patches

The patches referred to in this section can be found in appendix A.

4.1.1 Making `IFOComboMap` be sums of powers of two

The patch for this section can be found in appendix A.1.

Internally, the pipeline uses a static constant array defined in `gstlal-spiir/include/pipe_macro.h` called the "`IFOComboMap`" to keep track of which detectors it is processing. The original ordering of the array is as follows:

- All single detectors
- All combinations of two detectors
- All combinations of three detectors

This presents a problem when adding detectors, as the indexing of the array would change such that; all combinations of two detectors are shifted by at least 1, and; all combinations of three detectors are shifted by at least $\binom{n}{2} - \binom{n-1}{2}$, where n is the number of detectors; and so on. If the number of detectors changes, then all of the locations that index the `IFOComboMap` will also need to change, to ensure that they continue to index the right combinations. For example, the `scan_trigger_ifos` function in `gstlal-spiir/gst/cuda/cohfar/background_stats_utils.c` determines which detectors should be active by checking the indexes of the `IFOComboMap`, and if the number of detectors change, all of the indexing done in the function will also have to change.

The `IFOComboMap` is, as the name suggests, a list of all of the combinations of detectors, not including combinations of zero detectors. As a special case of the binomial expansion; $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$ when $x = y = 1$, we can note that [29];

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n. \tag{6}$$

As per equation 6, the sum of the number of combinations of n elements is equal to 2^n. This lends itself well to binary numbers, as an 8-bit binary number can represent 2^8 numbers or all the combinations of 8 elements — which means that we can represent all combinations of N detectors with N bits.

By having each detector represented as a single bit in a N-bit number, there are a number of advantages across the codebase that are evident as well. `popcount` (or `POPCNT`) is an instruction for x86 processors and NVIDIA GPUs that returns the number of set (1) bits in an integer, and thus the number of detectors being used can be determined with a single instruction. In addition, whether a detector is being used can be determined using a bitwise `AND` of the combination and 2^i, where i is the index of the detector. These both simplify large areas of the codebase, and make up the majority of the patch in appendix A.1.

By implementing this change, adding a new detector does not modify the existing indexing into the `IFOComboMap`, significantly reducing the number of files and code locations that would need to be modified to support a new detector.

4.1.2 Removing hard-coded detector names

The patch for this section can be found in appendix A.2.

As mentioned in section 3.3, there are two data structures that are passed between the components of the pipeline that can be seen in figure 1 — the `PostcohInspiralTable` and the `PeakList`. These data structures include detector specific variables that are cloned for each detector. For example, both structures have `snglsnr_H` data members to refer to the signal-to-noise ratio from the Hanford LIGO detector.

Within the `PostcohInspiralTable` and `PeakList` themselves, these can simply be moved from detector specific variables to being arrays with a length equal to the number of detectors, however this change causes some issues with the way the C code interacts with both CUDA and Python.

Arrays in C are simply pointers to the first element of a range of contiguous memory [30], however the CUDA programming model assumes by default that both the host and the GPU maintain their own separate memory spaces as well as separate allocation and deallocation routines [19]. This means that any multi-dimensional arrays (such as the afore mentioned transformation to `snglsnr_H`, which was previously a pointer to a float in the `PeakList` structure), require a additional memory copies per element per dimension, which is both unergonomic and difficult to implement correctly due to each dimension needing to be synchronised before the next dimension can be copied. Since CUDA 6.0, CUDA also provides the `cudaMallocManaged` function, which is accessible from the CPUs and GPUs a system as a common address space. By using `cudaMallocManaged`, the complexity of handling multi-dimensional arrays can be largely elided and passed to the CUDA runtime without performance impairment.

There are also several challenges that using C arrays to hold detector specific variables presents when trying to provide interoperability with Python. First, Python does not have a native contiguous-memory array data type, and instead usually uses linked lists as its array-like data structure. This can be rectified by using the array API of NumPy, an open source project for numerical computing with Python, which allows for C arrays to be used within a Pythonic API [31]. Second, as per section 3.4 the individual detector variables should be exposed in Python, despite them no longer existing as singular variables. This can be resolved by using Python's attribute getters and setters, which allow for customized functions to be run for getting and setting class attributes [32]. When used on a data type, an array of getters and setters is used [33], and the modification of this array at runtime allows for the programmatic creation of the individual detector variables.

After implementing this change, the only modifications required to add support for new detectors are to the `IFOComboMap` mentioned in section 4.1.1, and to the list of data sources to ensure that the data from the new detector can be read.

4.2 Testing

As per section 3.2, the `build_spiir` script from "SPIIR scripts" was used for building the changes to the code, and the unit tests from "GWDC utils" were used to ensure that the output of the pipeline was not affected by the code modifications.

The unit test in "GWDC utils" runs a comparison on a known good output and the signal-to-noise ratio series for each detector in a new build. Both patches were built, run and passed the "GWDC utils" unit test, ensuring that there were no regressions in output. In addition, the outputs of existing combinations of detectors were manually checked against a build of the SPIIR pipeline without the patches to ensure that fields other than the signal-to-noise ratio series were consistent across pipeline versions.

The ease of supporting additional detectors was tested by doing the necessary modifications to support one additional detector, ensuring that the build process worked correctly, and then running the same unit tests as above to ensure no regressions.

All testing of the unit tests and of additional detector support were done successfully with no faults observed on the OzStar cluster.

Criteria	Success
Output results must be unchanged	✔
The output table format should be unchanged	✗
Adding new detectors should be able to be done by a non-technical individual	✔
Maximum number of required edited files	2
Unchanged external interface	✔
Exposing of individual detectors in Python	✔
Changing number of detectors compiles correctly	✔

Table 1: An evaluation of the implementation of N-detector against the criteria specified in section 3.4

4.3 Evaluation

A summary of this section can be found in table 1.

1. **Output results must be unchanged**
   As per section 4.2, unit testing and manual comparison of the pipeline showed that the output results were unchanged.

2. **The output table format should be unchanged**
   Whilst the table that is sent to GraceDB was unchanged, the files output by the pipeline had their header names modified as the full name of the detector was appended to detector specific variables instead of the one representative letter.

3. **Adding new detectors should be able to be done by a non-technical individual**
   This is subjective, but arguably achieved.

   (a) **Maximum number of required edited files: 3**
   The number of files that need to be edited to add new detectors is 2, `gstlal-spiir/include/pipe_macro.h` for the `IFOComboMap` and `gstlal/python/datasource.py` for getting detector data.

4. **Unchanged external interface**
   The API for the SPIIR pipeline was not modified, with the only visible changes being internal to the pipeline.

5. **Exposing of individual detectors in Python**
   As per section 4.1.2, this was done using Python's attribute getters and setters.

6. **Changing number of detectors compiles correctly**
   As per section 4.2, the modified pipeline compiles and runs correctly with an additional detector added.

As per the criteria laid out in section 3.4, the implementation of N-detectors has been successful, however there are still areas for improvement. These areas will be outlined in section 6.

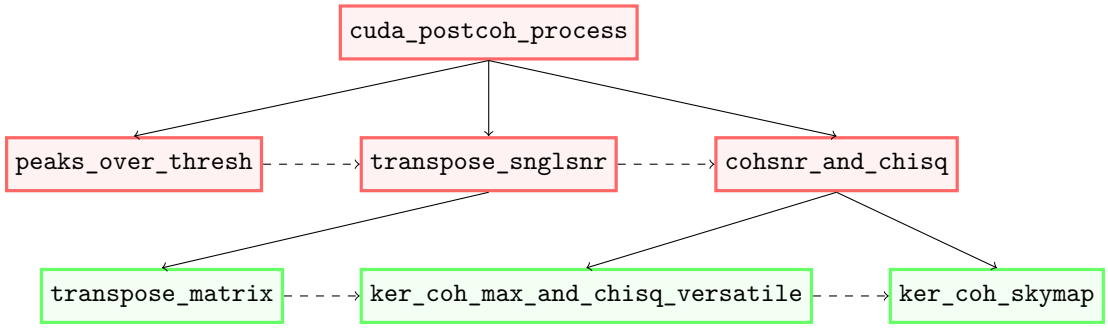

Figure 4: Simplified callgraph for coherent post-processing in the SPIIR pipeline.
Function calls are represented by solid arrows, the ordering of the calls is represented by dashed arrows.
GPU functions are in green boxes, and CPU functions are in red boxes.

5 Discussion

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

As discussed in 2.3, the SPIIR pipeline is designed to be as low latency as possible, and an asymptotic complexity analysis of components that may be impacted by the addition of new detectors allows for the measurements of the potential cost of doing so. This subsection aims to determine the complexity analysis of the coherent post-processing step of the SPIIR pipeline to understand the impact of increasing the number of detectors.

Coherent post-processing was introduced in 2017 by Qi Chu et al as an alternative to the coincidence post-processing used by all other pipelines (see section 2.1) [9]. In the original work, the computational cost of the coherent search is estimated to be $O(2N_d^3 N_m N_p)$, where N_d is the number of detectors, N_m is the number of sets of IIR filters (called templates), and N_p is the number of potential sky locations [9]. Further optimizations were made to the pipeline in 2018, including moving to using GPU acceleration, and whilst a paper outlining the changes made discusses a number of constant time optimizations made to the pipeline, the computational cost of the overall process is not discussed, despite the parallisation of GPU acceleration leading to additional changes to the overall potential cost [8].

"OzGrav Research utils" (see section 3.2) were used to generate a callgraph to determine the data flow for this analysis [see 28, `resources/callgraph.png`]. A simplified version of the callgraph can be seen in figure 4.

5.1.1 Maximum element reduction

One of the more common operations in the SPIIR pipeline is the concept of a "maximum element reduction". 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.

Harris 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 $O(\frac{N}{\log N} \times \log N) = O(N)$ [34].

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 3). 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 element reduction will be denoted for clarity with $M(x)$, where x is the size of the array being reduced.

5.1.2 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 element reduction to get the maximum SNR from the combined IIR filters (templates) for each sample. Recalling from section 5.1.1 that for maximum element reduction $T_1 = O(N)$, 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:

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

5.1.3 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 3 and 2), 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:

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

5.1.4 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 5.1.2) and splits the threads within the blocks for operations on that peak.

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 element reduction function discussed

in section 5.1.1.

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 element reduction function. We can further state that the ***parallelism*** of this is equivalent to the number of sky directions, $S + S/\log S$.

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^2_D 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:

$$\xi^2_D = \frac{\sum_{j=-m}^{m} |\varrho_D[j] - \varrho_D[0]A_D[j]|^2}{\sum_{j=-m}^{m}(2 - 2|A_D[j]|^2)}, \tag{7}$$

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 element reduction discussed in section 5.1.1 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, $O(N/\log N)$.

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.

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.

5.1.5 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 element reduction technique discussed in section 5.1.1. Following this, the function re-performs the process discussed in section 5.1.4 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 $T_1 = M_{T_1}(P) + S(D + D^2)$.

5.1.6 Overall complexity

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 5.1.2) 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 5.1.4 and 5.1.4) will be run for every detector.

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

$$\begin{aligned} T_\infty &= O(NT + N^2 + 1 + D(2(D + D^2 + \log S + D \log N) + \log P + D + D^2)) \\ &= O(NT + N^2 + D^3 + D^2 \log N + D \log S + D \log P), \end{aligned} \tag{8}$$

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:

$$\begin{aligned} T_1 &= O(NT + NT + S^2 + D(P + S(D + D^2) + P(S(D + D^2) + S + DN + B(S(D + D^2) + S)))) \\ &= O(NT + N^2 + SPD^3 + SPBD^3 + ND^2), \end{aligned} \tag{9}$$

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 $P = \max\{S, T\}$

5.1.7 Implications

6 Further work

The change to C arrays from individual detector variables in section 4.1.2 uses entirely fixed-size arrays, which means that the memory usage of the pipeline will be invariant to the number of detectors being used, instead it will depend on the number of detectors supported. This may present issues with a large number of detectors as GPU memory in particular is generally at a premium.

7 Conclusion

References

[1] *Introduction to LIGO & Gravitational Waves*. URL: https://www.ligo.org/science/GW-GW2.php.

[2] *LIGO Lab: Caltech: MIT*. URL: https://www.ligo.caltech.edu/.

[3] B. P. Abbott et al. "Observation of Gravitational Waves from a Binary Black Hole Merger". In: *Phys. Rev. Lett.* 116 (6 2016), p. 061102. DOI: 10.1103/PhysRevLett.116.061102. URL: https://link.aps.org/doi/10.1103/PhysRevLett.116.061102.

[4] *LIGO Detected Gravitational Waves from Black Holes*. URL: https://www.ligo.caltech.edu/detection.

[5] *LIGO Scientific Collaboration*. URL: https://ligo.org/.

[6] *Summed Parallel Infinite Impulse Response Pipeline Codebase*. URL: https://git.ligo.org/lscsoft/spiir/.

[7] Shaun Hooper et al. "Summed parallel infinite impulse response filters for low-latency detection of chirping gravitational waves". eng. In: *Physical Review D - Particles, Fields, Gravitation and Cosmology* 86.2 (2012). ISSN: 1550-7998.

[8] 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.

[9] Q. Chu. *Low-latency detection and localization of gravitational waves from compact binary coalescences*. eng. 2017.

[10] *LSC Algorithm Library for GStreamer*. URL: https://git.ligo.org/lscsoft/gstlal/.

[11] *PyCBC - Analyze gravitational-wave data, find signals, and study their parameters*. URL: https://pycbc.org/.

[12] Alex Nitz et al. "gwastro/pycbc: PyCBC release v1.16.11". In: (2020). DOI: 10.5281/zenodo.4075326.

[13] LIGO Scientific Collaboration. *LIGO Algorithm Library - LALSuite*. free software (GPL). 2018. DOI: 10.7935/GT1W-FZ16.

[14] *GstLAL documentation*. URL: https://lscsoft.docs.ligo.org/gstlal/.

[15] Cody Messick et al. "Analysis framework for the prompt discovery of compact binary mergers in gravitational-wave data". In: *Physical Review D* 95.4 (2017). ISSN: 2470-0029. DOI: 10.1103/physrevd.95.042001. URL: http://dx.doi.org/10.1103/PhysRevD.95.042001.

[16] *gstlal-inspiral/gst/lal/gstlal_itacac.c*. URL: https://git.ligo.org/lscsoft/gstlal/-/blob/master/gstlal-inspiral/gst/lal/gstlal_itacac.c.

[17] Cody Messick. *Added GEO600 support ("G1") to itacac (d43bcfd6)*. URL: https://git.ligo.org/lscsoft/gstlal/-/commit/d43bcfd6096ac4fab33114848b2d5f9ffaf6ca86.

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

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

[20] 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.

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

[22] 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.

[23] *GraceDB | The Gravitational-Wave Candidate Event Database*. URL: https://gracedb.ligo.org/.

[24] *OzStar – Supercomputing at Swinburne University of Technology*. URL: https://supercomputing.swin.edu.au/.

[25] *LIGO Data Grid*. URL: https://computing.docs.ligo.org/lscdatagridweb/.

[26] Patrick Clearwater. *Build SPIIR | SPIIR scripts*. URL: https://git.ligo.org/patrick.clearwater/spiir-scripts/.

[27] Alex Codoreanu. *GWDC Utils*. URL: https://git.ligo.org/alex.codoreanu/gwdc_utils/.

[28] Thomas Hill Almeida. "Tommoa/ozgrav-research". In: (2020). DOI: 10.5281/zenodo.4075219.

[29] Thomas H. Cormen et al. *Introduction to algorithms*. eng. 3rd ed. MIT electrical engineering and computer science series. Cambridge, Mass: MIT Press, pp. 1185–1186. ISBN: 0070131430.

[30] Bjarne Stroustrup. *The C++ Programming Language*. 4th ed. Addison-Wesley, 2013, pp. 179–182. ISBN: 0321563840.

[31] *Array API - NumPy v1.19 Manual*. URL: https://numpy.org/doc/stable/reference/c-api/array.html.

[32] *Common Object Structures - Python 2.7.18 documentation*. URL: https://docs.python.org/2.7/c-api/structures.html?highlight=pygetsetdef#c.PyGetSetDef.

[33] *Type Objects - Python 2.7.18 documentation*. URL: https://docs.python.org/2.7/c-api/typeobj.html#c.PyTypeObject.tp_getset.

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

A Patches

A.1 Making `IFOComboMap` be sums of powers of two

A.2 Removing hard-coded detector names