GENG555X

SEMINAR TALK

Thomas Hill Almeida (21963144)
Supervisor: Prof. Linqing Wen

2020-10-06

Contents

1 N-detector **2**
1.1 Gravitational Waves . . . 2
1.2 Processing Pipelines . . . 2
1.3 The power of powers of 2 . . . 3
1.4 The hunt for hardcoded detectors . . . 3

2 Complexity analysis **4**
2.1 Complexity Analysis . . . 4
2.2 So why do we care about this complexity analysis? . . . 4
2.3 CUDA . . . 5
2.4 Analysis . . . 5

3 Future work **5**

4 Et fin. **6**

Hello, I'm Tom Almeida, and I'm going to be presenting on code optimization of a gravitational wave detection pipeline — a project that I've been working on for the last six months.

But before we actually get into the content of my research, we should have a quick look at how this report is going to be structured.

I'm going to walk through each of the major areas that my research has focused on. These are — and I promise to expand on what these words actually mean once I get to them:

- N-detector implementation
- Complexity analysis; and
- CUDA `max_element` reduction

Finally, we'll finish by looking at the platform that this project gives for future research.

1 N-detector

1.1 Gravitational Waves

So let's begin by looking at some background information... what are gravitational waves?

You may have heard of a little thing called the "general theory of relativity", which Albert Einstein published in 1916 as a way to integrate gravity into the concept of space-time. Gravitational waves are 'ripples' in the fabric of space-time caused by some of the most powerful processes in the universe - things like colliding black holes, exploding stars and the birth of the universe are all things that emit gravitational waves. Their existence was predicted in 1916 as a derivation of Einstein's theory of general relativity, which showed that massive accelerating objects such as orbiting neutron stars or black holes would disrupt space-time in such a way that waves of distorted space would radiate from the source.

Detection of gravitational waves is done using interferometers, using the timing difference between massive lasers to determine if space-time has been distorted. Now the way that these detectors actually work and the physics behind all of this isn't really relevant to my research, so the most important thing to know about these detectors is that they emit a lot of data.

1.2 Processing Pipelines

To deal with all this data, processing pipelines were created to sift through the noise and find the gravitational waves.

One of these is the Summed Parallel Infinite Impulse Response pipeline (which I'm going to call SPIIR), which uses a combination of IIR filters (known as a "template") to approximate gravitational waves. It's currently thought to be one of the lowest latency pipelines of those existing, and is almost fast enough to enable prompt follow-up observation on gravitational wave events with traditional telescopes.

A major problem with the SPIIR pipeline (and all the other existing pipelines as well) is that only supports a fixed number of detectors, and the process of adding extra detectors to be processed is an incredibly time consuming process. A number of new detectors are coming online in the next few years, so figuring out a way to do this better is vital to ensuring that the pipeline can continue to operate.

This is the first part of my project, to allow the pipeline to run with any number of detectors with minimal change to the codebase.

1.3 The power of powers of 2

Internally, the pipeline uses an array to keep track of which detectors it is processing, and this array is called the IFOComboMap (which you can see on the screen). We can think of this map (in its current state) as being ordered as the following:

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

This gives us a problem when adding new detectors, because let's say we add one more detector, then suddenly the indexing would change for more than half of the combinations!

To solve this problem, we can think about how detectors are used in a slightly different way. Detectors (as you might be able to tell from the combination map) are either used or unused — and critically the ordering doesn't matter. We're talking about combinations here, and not permutations. One way to count the number of combinations with N objects is to count every number of to 2^N.

Let's use an example. Say we have 5 elements, and we want to represent every combination of those elements, we can represent it with a binary number with 5 bits. If I want to have only the first element selected in my combination, then I can simply flip it from 0 to 1. I can do the same if I want my combination to be the first and third elements, as well as the first, third and fifth elements.

We can use this information to make a table that doesn't change indexing at all when we add extra elements, by having each index be the number that would represent that combination if it were a power of two. It looks like this. And this is what it looks like now that we add detectors. Much better!

There's a few extra fun properties that we get from representing our combination like this. First, we can always tell how many detectors are being used by checking the number of flipped bits in our combination. We can do this super efficiently by using the x86 instruction "popcount", which should be even faster than a comparison (which you'd have to have done before). We can also check to see whether a particular detector is being used by checking to see if the bit corresponding to it has been flipped.

Making this change was actually relatively modest, with about 400 lines of code modified. Unfortunately, that's not quite the end of the game for letting the pipeline handle any number of detectors!

1.4 The hunt for hardcoded detectors

The pipeline passes around data structures with hardcoded names to each of its components. Fortunately, because C allows for fixed size arrays in data structures, this is quite a simple change, but unfortunately means that *everything* that references any of these tables needs to change in order to work with any number of detectors. So all in all, this ended up being around 1000 lines of code changed.

Unfortunately we don't have the time to dive into some of the technical things here about how this change interacts with the pipeline's Python or some of the other things, but feel free to come back to it in question time.

2 Complexity analysis

Now we're going to take a dive into a complexity analysis I did on one of the components of the pipeline.

We're going to start with some background information, and then I'll talk a bit about why this needed to be done before we get to the actually juicy things.

2.1 Complexity Analysis

Alright, so now let's actually have a look at what a complexity analysis is.

Complexity (or asymptotic) analysis is an analytical approach to understanding how the performance of an algorithm or data structure scales (or grows) with the size of its input. As UWA's very own CITS2200 (which this slide is ripped from) puts it, it allows us to compare algorithms and observe performance barriers. Complexity analysis of parallel algorithms is a little different to the sequential algorithms you may be used to, and so we need to define some terms.

First of all, we can consider a parallel algorithm to be a directed acyclic graph of sequential work, which we eventually combine to create some final answer. As our example up there, we have up there the "mergesort" algorithm and the work to complete each node up to the final combining node.

The "work" of a parallel algorithm is the same as though we attempted to execute the entire computation sequentially. So for this example, that would be $O(NlogN)$ (assuming we drew back enough nodes). We can find the "work" by summing up all the individual nodes together. A useful way to think about work is that it is equivalent to running the entire algorithm with just one processor — and thus it would be sequential.

The "span" of a parallel algorithm is the maximum time taken to complete any path in the DAG. So for this example, if we follow the path of the top edge, we can find that the "span" of mergesort would be $O(2N) = O(N)$.

Finally, we need to have some way of seeing practically what this means for us for runtimes. Brent's theorem is a handy way to combine our work and span with an actual number of P processors to turn it into an actual runtime. I've used it in my analysis, but we won't actually see it in action, just see the end result.

2.2 So why do we care about this complexity analysis?

Well now that we've successfully made it so we can have the SPIIR pipeline run any number of detectors, we need to figure out what that means for the speed of SPIIR. Remember that SPIIR is designed to be low-latency, so if adding too many more detectors slows things down, then it'll cause problems later on down the line.

We can see the components that we need to check for problems with extra detectors by taking another look at this data flow graph, which we can simplify further to this. Only the components after all the data flow combines need to be checked to make sure they'll deal with the extra detectors fine.

But before we get into the analysis, we need to look at CUDA, and how it plays with analysing parallel programs.

2.3 CUDA

The SPIIR pipeline uses NVIDIA's CUDA, a parallel programming interface for GPU acceleration, and the section of code that I'll be analysing today will be using CUDA so its worth taking a quick look at how it works so we can understand what's going on.

GPUs lend themselves really well to parallel code, as whilst they have very little memory per-processing core, they have orders of magnitude more processing cores than traditional CPUs. This heavy focus on parallel code results in a slightly different computational model.

In CUDA, each individual sequence of instructions being executed is called a thread. These threads are partitioned into warps, which are a group of 32 threads. Warps are the smallest unit of threads that a GPU can run, and all threads within a warp must execute the same instructions. Warps are then further organised into thread blocks, which are run on simultaneous multiprocessors and contain a small amount of fast memory that can be shared between the threads in the block. The region of data that is being processed can be further split by organising blocks into 'grids', although grids have no actual relation to a physical computation unit.

Conceptually speaking, every thread is a node that goes to every block, that goes to every grid. As such, we can "parallelise" things by splitting the work into many nodes by using threads, blocks or grids as our splitting mechanism.

2.4 Analysis

My first job was to figure out how and when all the functions were called, and by who. To do this, I wrote my own C/C++/CUDA compiler wrapper to generate a callgraph of which function calls what.

Now that I had my callgraph, I could use the techniques that I used above to determine what the total asymptotic complexity analysis was.

And this is what I ended up with. I'm sure that we can all stare at those final results all day and try to make heads and tails of it, but here's the important part — it turns out that SPIIR's post-processing deals with additional detectors kind of badly.

Because we currently use three detectors, the cubic term here means that we could potentially end up with around a 2.4x slow down of this component just for adding a detector.

This segues nicely into the future work that can be done to build on this research.

3 Future work

There are several future directions for this research to explore. One way to counter the potential slow down of adding new detectors is to simply have extra detectors skip the post-processing step.

This would change the structure of the pipeline from this, to this. So long as the pipeline is still able to detect gravitational waves with the remaining detectors, we can use these skipping detectors for localization — that is figuring out where the gravitational wave comes from. This is already being worked on, and should be ready to go in a few months time.

Another option is to take a further look at algorithms used in the post-processing. It may be possible that parallelisation around the number of detectors may be possible, which could potentially cause a large speedup, as even dropping from D^3 to D^2 brings the slowdown from extra detectors down significantly.

4 Et fin.

Anyway, that's about it from me, are there any questions?