Bachelor's Thesis Computing Science

Radboud University Nijmegen

Efficient Elastic Alignment in CUDA

Author:
Robin Put
s1031986

First supervisor/assessor:
dr. Ileana Buhan

Daily supervisor:
MSc. Peter Horvath

Second assessor:
prof. dr. Lejla Batina

January 13, 2024

Abstract

Side-channel attacks require a set of traces to be aligned. Traces can be aligned by using static or elastic alignment. Static alignment is not always the best solution, elastic alignment may be better. However, elastic alignment takes a long time to align traces and is mainly done on the CPU. In this thesis, we aim to find out whether it is helpful to compute elastic alignment on the GPU and to find out what the challenges of porting to the GPU are. This is done by implementing elastic alignment in CUDA. We find that memory is the biggest limitation of elastic alignment and the GPU. When large trace sets are used, the implementation on the GPU can potentially be faster than the implementation on the CPU.

Contents

Chapter 1

Introduction

Cryptographic devices use a secret key to encrypt data. Side-channel attacks like power analysis attacks require aligned power traces to obtain this secret key [7]. Static alignment is a method to align all the traces by shifting them over the time axis. Countermeasures against these side-channel attacks involve a varying clock speed or random process interrupts [12][7]. Because of these countermeasures, static alignment is not always a solution. These countermeasures are mitigated if elastic alignment correctly aligns the traces. Elastic alignment dynamically aligns the traces by stretching or compressing them over the time axis.

Dynamic time warping (DTW) [9] is an elastic alignment algorithm that finds the optimal alignment between two traces, but DTW has an expensive quadratic time and space complexity. There are algorithms like FTW [10], FastDTW [11] and HybridFTW [6] that overcome this limitation to an extent. These algorithms put constraints on the search space to reduce the computations required. Despite these algorithms, elastic alignment still takes a long time to find an alignment.

Currently, in practice, a tool Riscure Inspector is being used to elastically align traces using fastDTW. The Riscure Inspector implementation computes on the CPU and it may be beneficial to let fastDTW run in parallel on the GPU. To find out whether it is beneficial to compute fastDTW on the GPU we implement fastDTW in CUDA to finally answer our research question: *Is it possible to implement elastic alignment in CUDA and can it be faster than the Riscure Inspector implementation?*

In this thesis, we use the open-source implementation of fastDTW in Julia by Riscure[8] and modify it to run with CUDA on the GPU. Next, a bench-

mark is performed to compare the CUDA implementation[1] with Riscure Inspector. Afterward, we estimate the performance when the GPU memory limitation is loosened and bigger trace sets are used.

This thesis is organized in the following manner:
Introduction, gives an overview of the research topic, why this thesis is relevant, and what has been done in this thesis.
Background, gives the reader some important knowledge, that does not belong to the basic knowledge of a computing science bachelor student, that could help to understand the research.
Related work, gives an overview of the related work on this topic.
Experimental setup, consists of the research that has been conducted and how we deal with the challenges faced.
Conclusion, this last chapter summarizes the results and contributions of this research. The conclusion mentions the further research that can be done and a reflection on this thesis.

[1]The CUDA implementation created in this thesis is available on `https://github.com/robintup598/ElasticAlignmentCUDA`.

Chapter 2

Background

2.1 Traces

A trace is a set of data points that represent samples measured or recorded during the execution of a cryptographic algorithm. One example of such a trace is a power trace which contains information about the power consumption of a device. The power usage is influenced by the data processed, this can be used in a power analysis to obtain secret information like an encryption key [4]. An example of how such a trace looks is given in Figure 2.2.

2.1.1 Power trace

A power trace consists of samples measured with an oscilloscope. One sample can be split up into different parts [7]. The total power of one sample consists of:

- P_{OP}, the power that is operation-dependent.
- P_{DATA}, the power that is data-dependent.
- $P_{electronic\_noise}$, the power that is different every time even though the same operation with the same data is performed.
- P_{const}, the power that occurs independently of the performed operation and data.

When P_{DATA} cannot be easily distinguished, more traces may be required for power analysis attacks.

2.2 Power analysis attacks

Power analysis attacks are used to reveal a secret key of a cryptographic device by analyzing its power consumption. Two dependencies of the power consumption that are exploited are data and operation dependencies. As seen in Figure 2.1 the peaks of a clock cycle look different because the power consumption depends on the data and operations. [7].

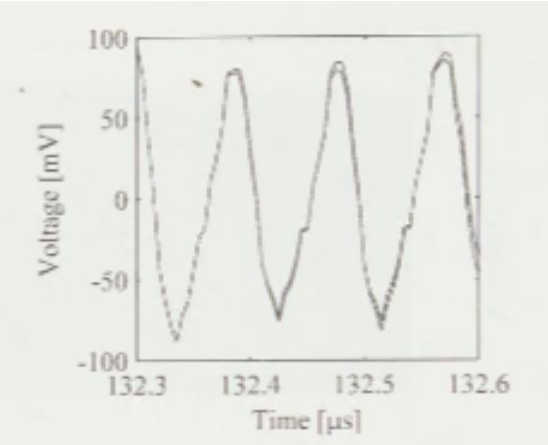

Figure 2.1. Two power traces of three clock cycles where the peaks differ slightly. [7]

2.2.1 Countermeasures

A countermeasure is implemented to make the power consumption of a cryptographic device independent of the intermediate values that are being processed by the encryption algorithm. There are two types of countermeasures: hiding and masking [7].
With hiding the data dependency of the power consumption is reduced. This means that the execution of the encryption algorithm is randomized or the power trace composition is changed in such a way that it gets harder for the attacker to find the data dependency. With masking the power consumption of randomized intermediate values that are being processed by the cryptographic device are independent of the actual intermediate values.

2.3 Static alignment

Static alignment [7] is a method to align misaligned traces to a reference trace. Static alignment works by first selecting a pattern in the reference trace. Subsequently, the other traces are searched for the chosen pattern and the part with the highest correlation is chosen. These traces are shifted

over the time axis such that the chosen patterns are aligned with each other. Static alignment works best when the following properties are taken into account when selecting the pattern: uniqueness, data dependency, length, and distance to the attacked intermediate result. Figure 2.2 contains two misaligned traces where $T1$ corresponds to the beginning of the chosen pattern and $T2$ to the end. As can be seen, trace 1 should be shifted to the left to be aligned with the reference trace 0.

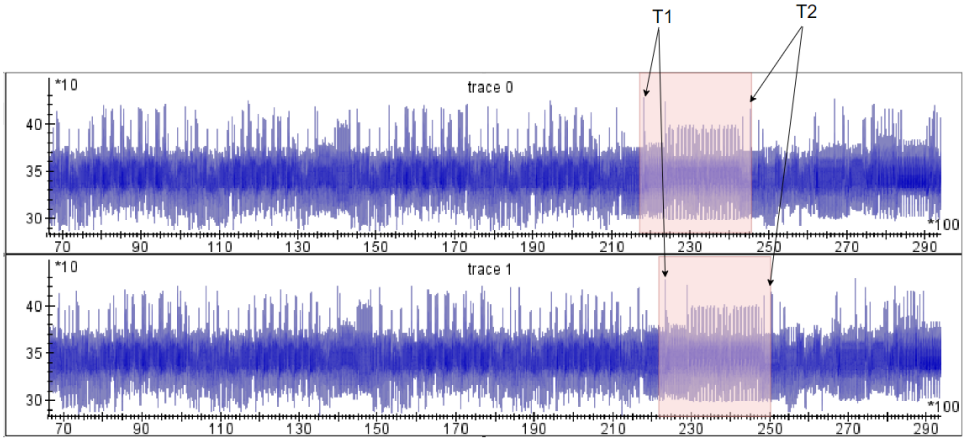

Figure 2.2. Two traces from the Riscure Inspector misalignment demo trace set, viewed in Riscure Inspector.

2.4 Elastic alignment

Elastic alignment is a method to create an optimal alignment of two traces. This is especially useful when the traces differ from each other as a result of countermeasures against side-channel analysis. Instead of moving the whole trace over the time axis like with static alignment, different parts of the trace are stretched or compressed to negate the countermeasures. [12]. In Figure 2.3 can be seen how the yellow trace is stretched and compressed to be aligned with the blue reference trace.

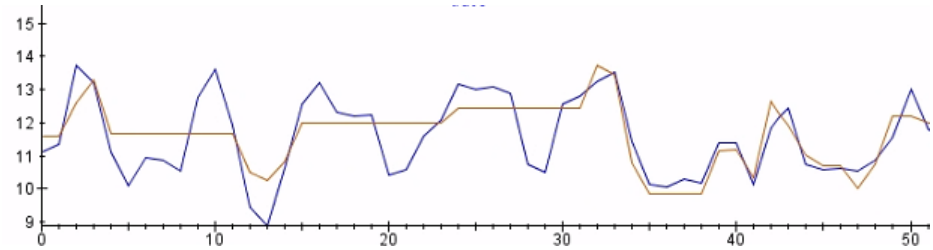

Figure 2.3. A zoomed-in view of the elastically aligned traces of Figure 2.2, viewed in Riscure Inspector.

2.4.1 Dynamic time warping

Dynamic time warping (DTW) [11][12] is an elastic alignment algorithm based on dynamic programming that finds the optimal alignment between reference trace X and Y. Every sample of trace Y needs to be linked to a sample of the reference trace X under the following constraints:

- **Monotonicity**: $x(i-1) \leq x(i)$ and $y(i-1) \leq y(i)$
- **Continuity**: $x(i) - x(i-1) \leq 1$ and $y(i) - y(i-1) \leq 1$
- **Boundary**: $x(1) = y(1), x(i) = T$ and $y(i) = T$, where T is the number of samples in X and Y

Each of these linked samples has a distance. For the optimal alignment, the sum of these distances is minimal. When all samples are linked together a *warp path* can be obtained, as seen in Figure 2.4.
A 2D matrix D, like in Figure 2.4, is constructed to compare two traces. The lengths of D are $|X|$ and $|Y|$. $D(i,j)$ is the minimum distance of the traces $X' = x_1..x_i$ and $Y' = y_1..y_i$.
The cells of matrix D are constructed by taking the current distance between two samples: $D(i,j)$, plus the minimum distance of the previous cells to the left, down or diagonal down: $min(D(i-1,j), D(i,j-1), D(i-1,j-1))$. To find the optimal warp path follow the shortest path from $D(T,T)$ back to $D(1,1)$.

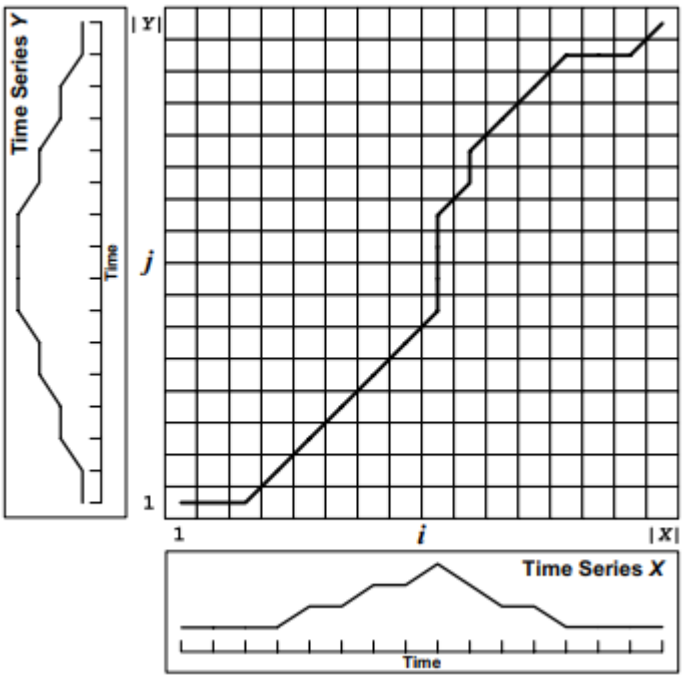

Figure 2.4. A matrix of DTW where an optimal warp path has been

found. [11].

Time and space complexity

The DTW algorithm has a time and space complexity of $O(N^2)$ because every cell in the matrix must be filled to find the optimal warp path.

2.4.2 FastDTW

FastDTW speeds up the DTW algorithm by approximating it [11].

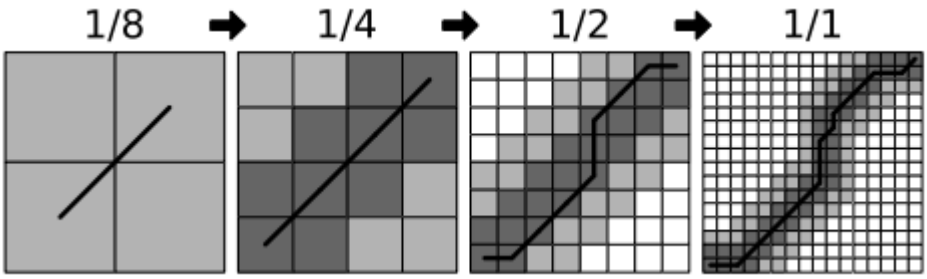

Figure 2.5. Four different resolutions during the fastDTW algorithm [11].

FastDTW has three key operations: Coarsening, Projection and Refinement.

Coarsening

During the coarsening phase, the trace is compressed by a factor of two. Until the length of the trace is smaller than 4 or the *radius*, then the original DTW is used to find a warp path.

Projection

During the projection phase, the warp path is found for the lower resolution. The warp path is then projected to a higher resolution, as can be seen in Figure 2.5 with the dark gray squares. With the projection, one single point maps to four higher-resolution points because of the resampling factor of 2.

Refinement

In the refinement phase, the projection constraint from the projection phase is loosened with the *radius* to get a more accurate solution. A higher radius results in better accuracy at the cost of speed.

Time and space complexity

The time and space complexity of fastDTW is $O(N(4r + 3))$ because the complexity grows linearly with the trace length. r is the radius input of the algorithm.

2.5 Riscure Inspector

Riscure Inspector is a powerful tool made by Riscure, a market-leading company in side-channel analysis. This tool is used to handle operations that are needed for side-channel analysis. These operations include inspecting, analyzing and modifying collected traces. Examples are resampling traces, static and elastic alignment.

2.6 NVIDIA GPU architecture and CUDA programming model

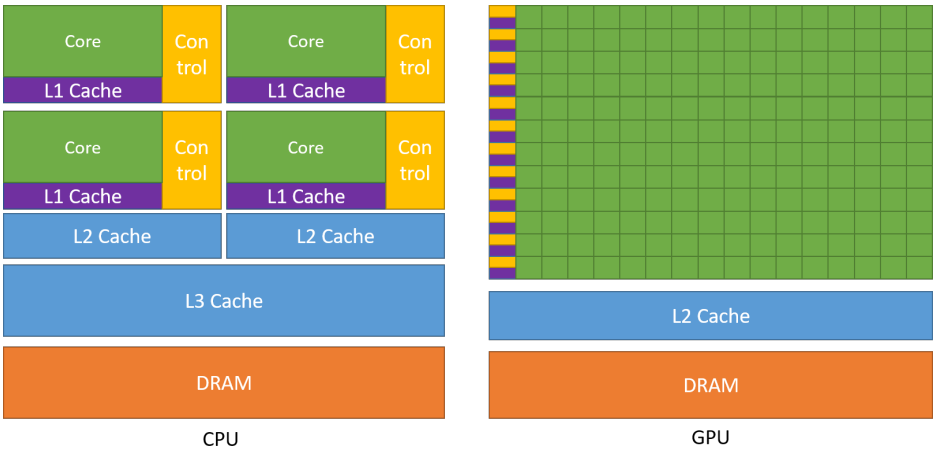

Figure 2.5: Comparison of the number of cores CPU vs GPU [1]

An NVIDIA graphics processing unit (GPU)[13][1] provides a high instruction throughput and memory bandwidth compared to the central processing unit (CPU) [1]. The CPU is designed to execute a (complex) task fast, while the GPU is designed to execute many more (simple) computations in parallel by utilizing its huge number of *CUDA cores* as seen in Figure 2.5. The GPU consists of multiple streaming multiprocessors (SMs), global memory, L2 cache and constant memory. Each SM contains its cluster of CUDA cores.
Different parts of the GPU contain memory. These parts are visually represented in Figure 2.6 by orange and yellow blocks. *Global memory*, which is the biggest block of memory available on the GPU, is accessible by all the

SMs. The downside of global memory is its high latency.
L2 cache is available to give lower latency access to frequently accessed data.
constant memory is a read-only block of memory that is allocated at compile time. Constant memory is accessible by all the SMs and benefits from a lower latency compared to global memory.
L1/Shared memory, a small memory block that is available in every SM, is shared between all threads in the same block. The latency of L1 memory is low compared to global memory.
Registers, every thread has its private registers, registers have the lowest latency of all memory in the GPU.

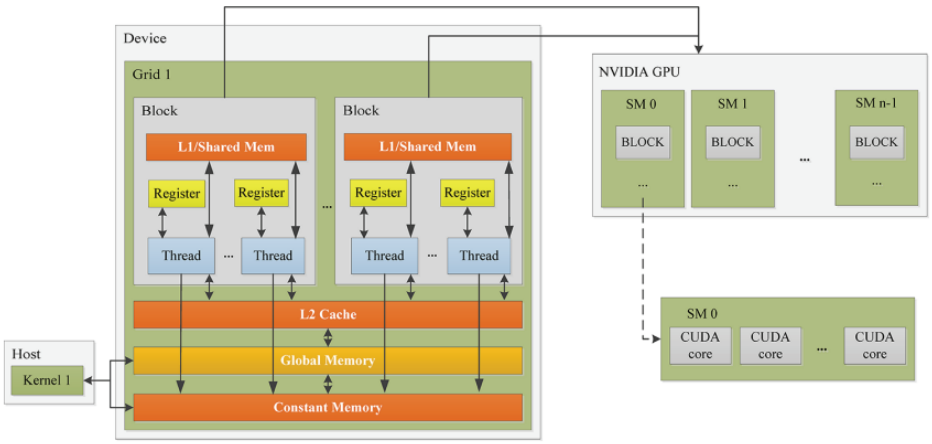

Figure 2.6: The CUDA programming model together with NVIDIA GPU architecture [13]

CUDA is a parallel computing model from NVIDIA that makes it easy for programmers to compute on the GPU. The model consists of grids, each grid consists of blocks, and each block consists of threads, as seen in Figure 2.6. The CUDA model maps the blocks of a grid and divides them over SMs. Multiple CUDA kernels can be launched, every kernel has one grid. The programmer specifies the number of threads per block and the number of blocks per grid. CUDA works by first allocating memory on the host and device (GPU), then copying initialized data from the host to the device, to in the end copy the relevant results back to the host.

Chapter 3

Related Work

There exists a book *power analysis attacks: revealing the secrets of smart cards* that discusses in-depth: cryptographic devices, power analysis attacks, countermeasures against power analysis attacks, and how to mitigate these countermeasures [7].

Aligning traces is not only relevant in the side-channel analysis domain but also used for other various applications like speech recognition, motion captures and many others [9, 5, 3] since 1978. All these alignment applications are based on dynamic time warping (DTW). DTW finds an optimal alignment between two traces, but has a quadratic time complexity [11]. Whenever traces are small the quadratic time complexity is not a problem but with larger ones, it becomes a limitation. In the last two decades, there has been quite some research done on improving the time complexity of DTW. All these approaches are based upon constraining the search space by a window to gain a lower time complexity.

In 2005 fast search method for dynamic time warping (FTW) was introduced which guarantees no false dismissals [10].
Later in 2007 $fastDTW$ [11] was introduced, which is another algorithm that approximates DTW in a linear time.
In 2022 these solutions were combined into $HybridFTW$ [6] to improve the performance of elastic alignment even further.

Another approach that has been investigated is computing DTW in parallel [14] by using multiple threads on the CPU. In 2022, a new algorithm was introduced that uses the GPU instead of the CPU, with the help of CUDA, to speed up the computation of a different algorithm called time-weighted dynamic warping (TWDTW) [2]. Although this solution shows promise, it is not intended for aligning traces.

Chapter 4

Experimental setup

We start with the open-source implementation of fastDTW available in the Julia programming language, from Riscure [8]. This Julia code takes as input two traces and outputs an array containing the indices of the cells where the warp path lays in the 2D matrix. Eventually, we want to align trace sets instead of only two traces.

4.1 Programming language

In this thesis, CUDA is used to compute on the GPU because of its flexibility which allows us to program complex algorithms. Because C code tends to perform better and the open-source code is in Julia we port the Julia code to Python, to then port it to C. Python is used for the host part of the code because Python is easy to use and has access to a library[1] that allows us to easily read and create trace set files. We make use of the *Cuda Python* library to work with CUDA through Python.

4.2 Preparing the algorithm for parallelization

The first limitation we come across on the GPU is the limited memory that is available for the threads. Because the thread memory is very limited in size, and the structure of the open-source implementation is recursive, our first idea is to call the recursive function calls as a new child thread. The child threads share memory with the parent thread so that does not help to solve our limited thread memory problem.

We look at making the space complexity of fastDTW linear by studying the 2D matrix that is being used. The matrix is created row per row and the hypothesis is to find the warp path while calculating the next row of the

[1] The trsfile library, available on `https://pypi.org/project/trsfile/`.

cost matrix. This gives either the not optimal solution or multiple paths are required to be explored as guesses which makes the implementation less efficient. The optimal warp path is guaranteed to be found in a single attempt when tracing back from the final cell because the final cell takes all the previously computed cells into account.

We make our implementation non-recursive to make it easier to read and work with and to limit the stack size. We make use of global memory instead of thread memory, for this purpose, the variables that are dependent on the input size are modified to pass by reference variables. The global memory has to be self-managed. Blocks of memory are preallocated for these variables, including traces, cost matrices, windows, and warp path results. Figure 4.1 shows a graph of the memory structure.

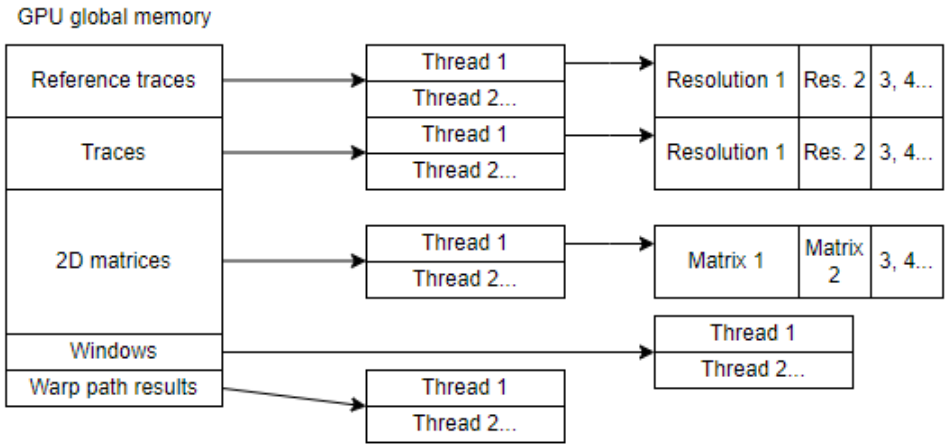

Figure 4.1 The structure of the preallocated memory on the GPU.

The main function of the code is $runFastDTW$, this function first creates all the lower resolutions required. The lower resolutions are stored in global memory after the initial trace, as can be seen in Figure 4.1. If there are no lower resolutions required then regular DTW is performed. The algorithm starts with the lowest resolution and uses DTW to obtain a warp path, this warp path is used to modify the window accordingly for the higher resolution.

While in theory, fastDTW requires significantly less memory for the 2D matrix thanks to using the window, in our implementation the 2D matrix memory allocation is still required to keep the structure.

Algorithm 1 CUDA C kernel pseudocode

```
1: procedure runFastDTW
2:     while length > radius + 4 do
3:         X* ← Resample X with factor 2
4:         Y* ← Resample Y with factor 2
5:     if length lowest resolution > radius + 4 then
6:         DTW on X and Y
7:     else
8:         for each resolution, starting from the lowest do
9:             DTW on X and Y
10:            Expand the window
```

4.3 Benchmarking CUDA with Riscure Inspector

4.3.1 Benchmark setup

We benchmark our CUDA implementation against Riscure Inspector[2] because both implementations align trace sets and Riscure Inspector is being used in practice. The CUDA implementation mainly uses the GPU[3] while Riscure Inspector solely computes on the CPU[4]. Because the GPU and CPU are variables in this benchmark the results are not perfect, a better GPU may let the CUDA implementation results look better compared to the Riscure Inspector implementation and visa versa. Nonetheless, the benchmark still indicates the potential speedup of using the GPU.

The number of traces and samples per trace influences how fast the trace set can be aligned. The GPU memory limitations restrict us to a certain combination of trace set size and trace size that can be done at once in parallel. To align bigger trace sets the algorithm essentially has to run again. The highest number of samples per trace in a trace set, marked bold in Table 4.1, requires approximately 4GB of memory.

The following trace subsets are used for the benchmark, they are taken from the *Misalignment example* demo trace set of Riscure Inspector. If a lower number of traces or samples is needed the trace set is truncated, if a higher number is required then the traces in the set are duplicated.

[2]Riscure Inspector version 4.12 is used for this benchmark.
[3]An NVIDIA 2060 GTX GTX 6GB mobile GPU is being used.
[4]An Intel i7-6700 3.4GHz CPU is being used.

traces per set:	1 trace	10 traces	100 traces	200 traces
samples per trace	1000	1000	1000	1000
	2000	2000	2000	**2000**
	5000	5000	**3000**	-
	10000	**10000**	-	-
	20000	-	-	-
	30000	-	-	-

Table 4.1: The first row states how many traces there are per trace set. The rows below indicate the number of samples in a trace.

4.3.2 Expectation

Because the CUDA implementation utilizes the parallelization capabilities of the GPU, it is expected that the CUDA implementation becomes faster relative to Riscure Inspector when the number of traces per trace set grows.

4.3.3 Results and discussion

Every benchmark ran three times and the time it took was consistent. Notably, as seen in Table 4.3, the CUDA implementation outperforms the Riscure Inspector implementation in the largest trace set, which includes 2000 traces with 2000 samples each. If we compare this result from Riscure Inspector with a smaller trace set, where the number of traces is halved, the eight seconds it takes to align 200 traces is unexpected because one would expect that doubling the number of traces from 100 to 200 with the same number of samples would approximately double the time from one second to two seconds. With our findings aligning the set of 200 traces takes eight times as long instead of the expected two times.

The following two tables contain the benchmark results in seconds:

1 trace	CUDA	RI	10 traces	CUDA	RI
1000 samples	0,5748	0,0039	1000	0,678	0,053
2000	0,6693	0,00348	2000	0,852	0,104
5000	0,9052	0,00385	5000	1,511	0,258
10000	1,5471	0,00418	10000	3,775	0,527
20000	4,3811	0,0036	-	-	-
30000	13,6790	0,00394	-	-	-

Table 4.2: The speed of the CUDA implementation and Riscure Inspector in seconds for 1 and 10 traces per trace set.

100 trace	CUDA	RI	200 traces	CUDA	RI
1000 samples	1,33	0,535	1000	2,12	1,07
2000	**2,44**	**1,051**	2000	**4,24**	**8,29**
3000	4,27	1,595	-	-	-

Table 4.3: The speed of the CUDA implementation and Riscure Inspector in seconds for 100 and 200 traces per trace set.

If a trace set consists of a single trace, the CUDA implementation is significantly slower than the CUDA implementation (Table 4.2). The CUDA implementation has a lot of overhead that Riscure Inspector does not have. For example, allocating all memory on the host and device, to then copy all memory from the host to the device, and then in the end copying the results back to the host.
With a trace set of 10 and 100, both implementations seem to have a linear time complexity.
It is likely the case that the quadratic space complexity of our implementation has a big impact on its performance since the speedup of our implementation is dependent on the trace size.

4.4 Estimation

Because the size of the trace sets being used in the benchmark is limited and we want to know what the speedup could be if the memory of the GPU was larger, we are going to estimate what the speedup could be. As the benchmark results are limited and somewhat surprising, our estimation will not be very precise:

1. If the memory of the GPU is completely taken up by a trace set, and we double the number of traces, the computation time doubles because the program essentially runs twice.

2. If we double the total memory of the GPU, and we assume that there are enough traces to make use of all the memory on the GPU, the computing time on the GPU halves while allocating time on the CPU stays the same. Because preparing all the memory cannot be done in parallel this time still takes the same amount of time, while working with the data can be done in parallel on the GPU.

3. We assume that allocating and initializing on the CPU takes up 50 percent of the total time it takes to compute. This matches with all the tested data sets of the benchmark.

With assumptions 2 and 3, if we double the amount of memory that the GPU has available for large trace sets this increases the performance by 33 percent. For example, if a trace set of 1000 traces is used with a GPU, the GPU can compute 500 traces in parallel. The total time it takes to align this trace set is 10 seconds, of which 5 are spent on allocating and initializing memory on the CPU (assumption 3). If we double the amount of memory the GPU has available, which lets us compute 1000 traces in parallel, this takes 2,5 seconds on the GPU and 5 seconds on the CPU (assumption 2). From doubling the GPU memory the computing time decreases from 10 seconds to 7,5, which results in a performance increase of 33 percent.

If we increase the number of traces per trace set, both the CUDA and Riscure Inspector implementation take twice as long. This means that percentage-wise the performance of both implementations stays the same.

We cannot make any sensible assumptions and conclusions for increasing the number of samples per trace. This is because there are not enough benchmark results where there is only an increase in samples per trace. The limited number of results we have are not consistent.

Chapter 5

Conclusions and future work

In this thesis, we looked at implementing elastic alignment in CUDA and we unveil the performance of CUDA with Riscure Inspector. We found challenges on the GPU while implementing fastDTW in CUDA and found ways to deal with them. A challenge of elastic alignment is its time and space complexity, this results in to a limitation of the GPU where global memory has to be used for the amount of memory needed. Because global memory is used all the memory needs to be self-managed.

We found that it is possible to implement elastic alignment in CUDA. Benchmarking CUDA against Riscure Inspector gives us an indication of the potential speedup by using the GPU. Our CUDA implementation seems to be twice as fast as Riscure Inspector whenever a trace set of 200 traces is used with at least 2000 samples per trace, however, this result is not what we expected because we expect that for Riscure Inspector a trace set of 200 traces takes twice the time of a trace set with 100 traces.

When fewer traces are used the parallelization capabilities do not overcome the efficiency of the Riscure Inspector implementation. If elastic alignment can be implemented in CUDA with a more efficient (space) complexity, then the speedup is even bigger because more traces can be done in parallel.

Future work could be done on efficiently implementing elastic alignment algorithms with CUDA, including efficient usage of memory. If less memory is required per trace this results in a significant speedup compared to Riscure Inspector because more traces can be aligned in parallel.

Reflection

Upon reflection, this thesis mostly focused on implementing elastic alignment in CUDA, which consumed a significant amount of time and left less room for optimization. Benchmarking the CUDA implementation against

Riscure Inspector could have been done more thoroughly if more time was available.

Bibliography

[1] NVIDIA Corporation and affiliates. *CUDA C++ Programming Guide Release 12.3*, 2023. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[2] Hengliang Guo, Bowen Xu, Hong Yang, Bingyang Li, Yuanyuan Yue, and Shan Zhao. Cuda-based parallelization of time-weighted dynamic time warping algorithm for time series analysis of remote sensing data. *Computers & Geosciences*, 164:105122, 2022.

[3] Eamonn Keogh and Shruti Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Mining and Knowledge Discovery*, 7(4):349–371, Oct 2003.

[4] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Apr 2011.

[5] Lucas Kovar and Michael Gleicher. Automated extraction and parameterization of motions in large data sets. *ACM Trans. Graph.*, 23(3):559–568, aug 2004.

[6] Minwoo Lee, Sanghun Lee, Mi-Jung Choi, Yang-Sae Moon, and Hyo-Sang Lim. Hybridftw: Hybrid computation of dynamic time warping distances. *IEEE Access*, 6:2085–2096, 2018.

[7] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[8] Riscure. Riscure fastdtw. https://github.com/Riscure/Jlsca/tree/master/src, 2023. Accessed on 20 September 2023.

[9] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):43–49, 1978.

[10] Yasushi Sakurai, Masatoshi Yoshikawa, and Christos Faloutsos. Ftw: Fast similarity search under the time warping distance. pages 326–337, 06 2005.

[11] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intell. Data Anal.*, 11(5):561–580, oct 2007.

[12] Jasper G. J. van Woudenberg, Marc F. Witteman, and Bram Bakker. Improving differential power analysis by elastic alignment. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, pages 104–119, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] Tianyi Wang and Qian Kemao. *GPU Acceleration for Optical Measurement*. 12 2017.

[14] Óscar Escudero-Arnanz, Antonio G. Marques, Cristina Soguero-Ruiz, Inmaculada Mora-Jiménez, and Gregorio Robles. dtwparallel: A python package to efficiently compute dynamic time warping between time series. *SoftwareX*, 22:101364, 2023.

Appendix A

Github repository

https://github.com/robintup598/ElasticAlignmentCUDA