DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Thesis type (Master's Thesis in Informatics, ...)

Thesis title

Author

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Thesis type (Master's Thesis in Informatics, ...)

Thesis title

Titel der Abschlussarbeit

Author:	Author
Supervisor:	Supervisor
Advisor:	Advisor
Submission Date:	Submission date

I confirm that this thesis type (master's thesis in informatics, ...) is my own work and I have documented all sources and material used.

Munich, Submission date Author

Acknowledgments

Abstract

Time series data mining has become essential for a large variety of disciplines ranging from seismology to medicine. With the use of matrix profiles, most target analyses, especially motif or discord discovery, have become trivial to a large extend. As the original formulations for calculating the matrix profile were relatively ineffective, substantial effort has been invested in computing them efficiently. In particular, the state-of-the-art algorithm SCAMP has convinced with its simplicity, parallelizability, and efficiency.

As FPGA acceleration promises a combination of high performance and competitive energy efficiency, we present a - to the best of our knowledge first - systolic array-based design, based on the SCAMP algorithm, to compute matrix profiles efficiently on FPGAs. Our design mapped to a concrete architecture using a High-Level Synthesis tool. This approach allows us to maintain a high level of abstraction, use streaming abstractions, and enable maintainability and portability across FPGA devices. Kernels synthesized from our design are shown to offer competitive performance in practice, scaling with both compute and memory resources. Finally, we outline possible optimizations as a basis for future work.

Contents

1 Introduction

The understanding of the matrix profile, and its computation, is fundamental to the design of the underlying work. We briefly introduce the reader to concepts related to matrix profiles and their efficient computation, which are the basis for the formulations deployed on the Cerberas accelerator. Time series data mining, as data processing applications in general, greatly profits from streaming abstractions and the ability to express data transformations on a high level. We, therefore, employ a weak scaling approach to mapping the developed algorithm to a concrete hardware design running on the Cerebras accelerator. As the corresponding programming model is relatively novel and constitutes the basis of the introduced design and its subsequent implementation, the reader is introduced to fundamental concepts of the Accelerator and its abstractions. These concepts prove essential to address the complexities of the design while allowing the problem to be structured effectively. This chapter is split into two parts: first, the reader is introduced to the concept (and required notation) of a matrix profile and an algorithm (SCAMP) for its efficient computation; secondly, the chapter establishes fundamentals related to the target device, which make up the basic building blocks for the presented design and its implementation.

2 Theoretical Background and Related Works

Matrix profiling has emerged as a powerful technique for time series analysis, offering insights into underlying patterns, anomalies, and recurring motifs. This master thesis investigates the integration of the Cerebras Accelerator, a state-of-the-art hardware architecture designed for accelerating deep learning tasks, into the domain of matrix profiling. The Cerebras Accelerator's unique Wafer-Scale Engine (WSE) architecture, characterized by its massive scale and fine-grained parallelism, presents an intriguing opportunity to significantly enhance the efficiency of matrix profiling computations.

The research delves into the theoretical foundations of matrix profiling and explores the challenges associated with its computational demands. Leveraging the parallel processing capabilities of the Cerebras Accelerator, the study aims to optimize matrix profiling algorithms for enhanced speed and scalability. The thesis investigates how the WSE-2 architecture can be harnessed to efficiently perform essential matrix operations, such as sliding window calculations and motif discovery, crucial for time series analysis.

Furthermore, the research involves practical implementations and performance evaluations to assess the impact of Cerebras Accelerator on the overall efficiency of matrix profiling workflows. Comparative analyses against traditional hardware architectures and accelerators will provide insights into the unique advantages and potential limitations of employing Cerebras in this context.

The findings of this research not only contribute to the evolving field of time series analysis but also shed light on the adaptability and effectiveness of specialized accelerators, like the Cerebras Accelerator, in domains beyond their primary focus. Ultimately, this thesis seeks to bridge the gap between advanced hardware architectures and the demanding computational requirements of matrix profiling, offering a novel perspective on accelerating time series analysis for real-world applications.

2.1 Matrix Profile

While the matrix profile is a reasonably intuitive concept, a comprehensive set of definitions and notations is required to introduce and explain the design and optimization of the kernels responsible for its actual computation. This section aims to introduce those

fundamentals to the reader to express matrix profiles. The definitions are mainly in line with [7] and [8]. As the following subsections only briefly introduce the required concepts, the reader is referred to the references mentioned above for a more detailed explanation.

2.1.1 Definitions

We first introduce the core data structure associated with matrix profiles: *time series*.

Definition 1 A *time series* T of length $n \in \mathbb{N}$ is a *sequence* of real-valued numbers $t_i \in \mathbb{R}$:

$$T = t_1, t_2, \ldots, t_n$$

While time series data constitutes both input and output for matrix profile computation, we require a more granular data structure called a *subsequence* during the calculations. For a visual representation of a time series and a subsequence (depicted with a grey background), see Figure 2.1.

Definition 2 A *subsequence* $T_{i,m}$ of a time series T is a continuous subset of the values from T of length $m \in \mathbb{N}$ starting from position $1 \leq i \leq n$ - $m + 1$:

$$T_{i,m} = t_i, t_{i+1}, \ldots, t_{i+m-1}$$

Note: By this definition, every subsequence is itself a time series.

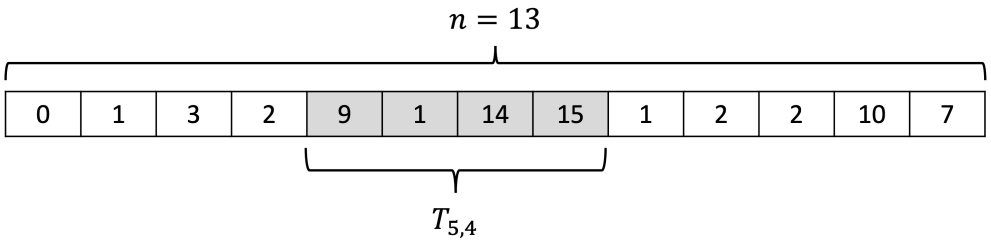

Figure 2.1: Time Series of length n = 13 and Subsequence $T_{5,4}$

Calculating distances between subsequences is the core of the computation of matrix profiles. We define the (*z-normalized Euclidean*) *distance* between *time series*, and therefore *subsequences*, as follows:

Definition 3 The *z-normalized Euclidean distance* between two time series $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_m$ of the same length $m \in \mathbb{N}$ is defined as

$$d(X, Y) = \sqrt{\sum_{i=1}^{m} (\bar{x}_i - \bar{y}_i)}$$

where $\bar{x}_i = \frac{x_i - \mu_X}{\sigma_X}$ and $\bar{y}_i = \frac{y_i - \mu_Y}{\sigma_Y}$. $\mu_Z and \sigma_Z$ denote the sample mean and standard deviation of a time series Z respectively. While this defenition is rather intuitive, we make use of the following reformation during the computation:

$$d(X, Y) = \sqrt{2m \left(1 - \frac{\sum_{i=1}^{m} x_i y_i - m\mu_X \mu_Y}{m\sigma_X \sigma_Y}\right)} = \sqrt{2m \left(1 - P\left(X, Y\right)\right)} \tag{2.1}$$

with $P(X, Y)$ denoting the Pearson correlation coefficient between X and Y:

$$P(X, Y) = \frac{\sum_{i}^{m} x_i y_i - m\mu_X \mu_Y}{m\sigma_X \sigma_Y} \tag{2.2}$$

Computing the distance between a single subsequence and every other possible subsequence of T results in the so-called *distance profile*. Thus, the distance profile can be utilized to find similar subsequences, i.e., subsequences with a small distance or subsequences that are substantially different, i.e., have a significant distance. In particular, the subsequence with the smallest distance is the closest match. If this distance is relatively small, the subsequence represents a motif, as it is similarly contained in T multiple times.

Definition 4 A *distance profile* D_i of a time series T is a vector of the Euclidean distances between a given query subsequence T_i, m and each subsequence of length m in T. Formally,

$$D_i = (d_{i,1} d_{i,2} \ldots d_{i,n-m+1})$$

where $d_{i,j}\, (1 \leq i, j \leq n - m + 1)$ is the z-nromalized distance between T_i, m and T_j, m.

By computing the distance profile for each subsequence of T, we obtain the so-called *distance matrix*. Therefore, the distance matrix contains the distance between every subsequence and every other subsequence in T.

Definition 5 A *distance matrix* D of a time series T is the vector of all distance profiles $D_i\, (1 \leq i, j \leq n - m + 1)$:

$$D = \begin{pmatrix} D_1 \\ D_2 \\ \vdots \\ D_{n-m+1} \end{pmatrix} = \begin{pmatrix} d_{1,1} & d_{1,2} & \dots & d_{1,n-m+1} \\ d_{2,1} & d_{2,2} & \dots & d_{2,n-m+1} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n-m+1,1} & d_{n-m+1,2} \dots & d_{n-m+1,n-m+1} & \end{pmatrix}$$

where $d_{i,j}$ $(1 \leq i, j \leq n - m + 1)$ once again denotes the z-normalized distance between subsequences $T_{i,m}$ and $T_{j,m}$. Finally, these definitions allow us to define the matrix profile, which can be expressed as the column-wise minima, henceforth referred to as aggregates, of the distance matrix. Therefore, MP_i is the minimal distance between subsequence $T_{i,m}$ and any other subsequence in T. Note that due to the symmetry of Euclidean distances $(d_{i,j} = d_{j,i})$, the matrix profile also represents the vector row-wise minima.

However, the distance between any subsequence and itself is 0 ($d_{i,i} = 0$). Furthermore, distances of adjacent subsequences are also relatively small. These matches are commonly referred to as (trivial matches) and are typically excluded, as they remain uninteresting for most use cases. The *exclusion zone* for a subsequence is defined as the set of indices that result in a trivial match. What is considered to be a trivial match depends on the application domain. Most commonly, an exclusion zone of $m/2$ is utilized, i.e., when computing the minima for subsequence $T_{i,m}$, the subsequences $T_{i-m/4,m}, \dots, T_{i+m/4,m}$ are ignored.

Definition 6 The *matrix profile MP* of a time series T is the vector corresponding to the column-wise minima of the distance matrix:

$$MP = \left(min_j\left(d_{1,j}\right) min_j\left(d_{2,j}\right) \dots min_j\left(d_{n-m+1,j}\right)\right)$$

wherein $min_j\left(d_{1,j}\right)$ is the minimum of D_i ignoring subsequences contained within the exclusion zone.

We are also interested in the index of the subsequence with the minimal distance, we introduce the *matrix profile index*, which represents the vector of indices corresponding to the entries in *MP*.

Definition 7 The *matrix profile MP* of a time series T is a vector of the corresponding indices of the matrix profile:

$$MPI = \left(argmin_j\left(d_{1,j}\right) argmin_j\left(d_{2,j}\right) \dots argmin_j\left(d_{n-m+1,j}\right)\right)$$

In the case of several minima, the one with the smallest index is to be chosen.

A graphical representation of the matrix profile and matrix profile index can be found in Figure 2.2 Elements contained within the exclusion zone are depicted with a grey background.

2.1.2 Computation

While the definitions in Subsection 2.1.1 help understand the concept of matrix profiles, their straightforward implementation is relatively inefficient. The performance is inherently limited by the interinsic dot product operations of Equation 2.1 and Equation 2.2. In the following, a more efficient way of computing the matrix profile, an algorithm called SCAMP in accordance with [2], is introduced.

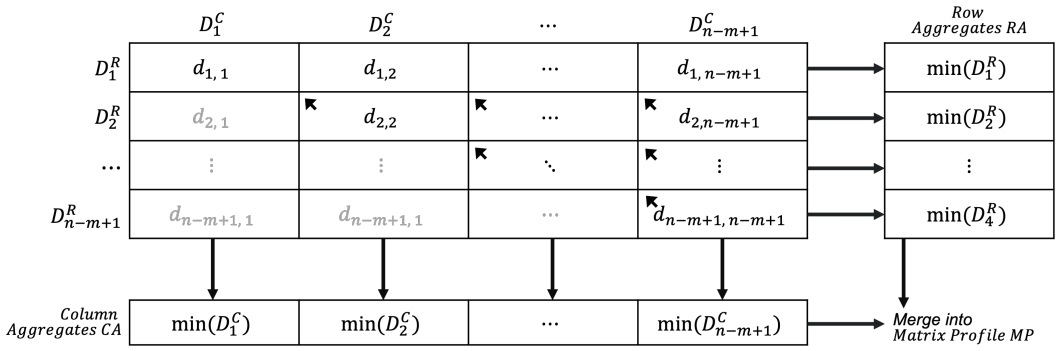

Figure 2.2: Matrix Profile MP of a Time series T as the column-wise minima of the Distance Matrix and the Matrix Profile Index MPI as the vector of the corresponding indices. In this example, $d_{2,j}$ represents a column-wise minimum and is therefore integrated into the Matrix Profile.

The Pearson correlation $P_{i,j}$ between two subsequences $T_{i,m}$ and $T_{j,m}$ of a fixed time series $T = t_1, t_2, \ldots, t_n$ and a common subsequence length $m \in \mathbb{N}$, as explicitly formulated in Equation 2.2, can be computed as follows:

$$P_{i,j} = \overline{QT}_{i,j} * inv_i * inv_j \tag{2.3}$$

where,

$$\overline{QT}_{i,j} = \sum_{k=0}^{m-1} \left(t_{i+k} - \mu_i\right)\left(t_{j+k} - \mu_j\right) \tag{2.4}$$

and inv_k denotes the inverse L2-Norm:

$$inv_k = \frac{1}{||T_{k,m} - \mu_k||} \tag{2.5}$$

SCAMP employes an optimization on $\overline{QT_{i,j}}$ by not implicitly calculation for all i, j by using a centered-sum-of-products formula:

$$\overline{QT}_{i,j} = \overline{QT}_{i-1,j-1} + df_i \cdot dg_j + df_j \cdot dg_i \tag{2.6}$$

where,

$$df_k = \begin{cases} 0, & \text{if } k = 1 \\ \frac{t_{k+m-1} - t_{k-1}}{2}, & \text{if } 2 \leq k \leq n - m + 1 \end{cases} \tag{2.7}$$

and, with μ_i representing the sample mean of the subsequence $T_{i,m}$,

$$dg_k = \begin{cases} 0, & \text{if } k = 1 \\ t_{k+m-1} + (t_{k-1} - \mu_{k-1}), & \text{if } 2 \leq k \leq n - m + 1 \end{cases} \tag{2.8}$$

Equations 2.7 and 2.8 are used to precompute the terms used in Equation 2.6 and incorporate incremental mean centering into the update. In addition to df and dg, the required L2-norm inverses are precomputed to avoid unnecessary recomputations.

As described in Equation 2.1, we can then convert the calculated Pearson correlation into Euclidean distance in $O(1)$ via:

$$d_{i,j} = \sqrt{2m\left(1 - P_{i,j}\right)} \tag{2.9}$$

Note, while the computation described in Equation 2.6 introduces a diagonal dependency between computations ($\overline{QT}_{i-1,j-1}$ is required to compute $\overline{QT}_{i,j}$), this dependency is circumvented at any point by explicitly calculating $\overline{QT}_{i-1,j-1}$ via the explicit dot product fomulation (Equation 2.4). The explicit caluclation is required for the first row, i.e., $\overline{QT}_{1,j}$ has to be calculated via the straightforward definition.

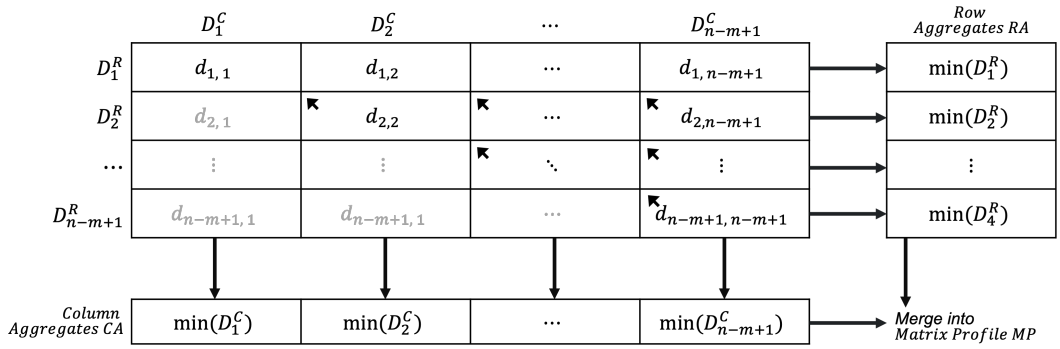

Figure 2.3: Computation performed by the SCAMP algorithm. In particular, only values above (and including) the main diagonal are computed. The diagonal dependency introduced through the updated formulation is visualized through upward-pointing arrows.

SCAMP does not compute the entire matrix but rather only elements above the main diagonal. This can be done due to the symmetric nature of the problem. We, therefore, store row- and column-wise aggregates, i.e., the minima of the considered values. These aggregates are merged subsequently to obtain the resulting matrix profile, as depicted in Figure 2.3.

2.2 Cerebras Wafer Scale Engine

Cerebras Systems' CS-2 stands as a groundbreaking hardware solution tailored for accelerating deep learning tasks. At its core lies the Cerebras second-generation Wafer Scale Engine (WSE-2), a monumental achievement boasting around 1.2 trillion transistors, making it the largest chip in the industry. The CS-2 wafer operates as a multiple instruction, multiple data (MIMD) distributed-memory system, interconnected via a 2D-mesh fabric. Each fundamental unit on the wafer, referred to as a tile, encompasses a processor core, its memory, and a router facilitating connections. These routers link with neighboring tiles, forming a 7×12 array housing 84 identical dies, each containing thousands of tiles. Notably, the system integrates 40 gigabytes of on-chip SRAM, offering rapid access within a single clock cycle, alongside staggering memory and interconnect bandwidths of 20 petabytes/sec and 220 petabits/sec, respectively. Comparatively, the WSE-2 dwarfs graphical processing accelerators with over 100 times the compute cores, 1000 times more high-speed on-chip memory, and over 12,000 times

more fabric bandwidth. Spanning approximately 46,000 mm^2 in size, the WSE-2 employs Sparse Linear Algebra Compute (SLAC) cores for efficient computation, boasting over 850,000 of these cores designed specifically for sparse linear algebra tasks crucial in machine learning. This massive design flexibility empowers the WSE-2 to adapt to various linear algebra paradigms, effectively catering to industry-grade applications. Leveraging SLAC cores enables the WSE-2 to bypass zero-to-zero multiplications in large datasets, optimizing compute resources and system efficiency. The Cerebras Software platform complements this hardware prowess by facilitating seamless machine learning model training, supporting popular frameworks such as TensorFlow and PyTorch. Its graph compiler translates ML models into optimized executables for the CS-2, ensuring smooth integration into researchers' workflows.

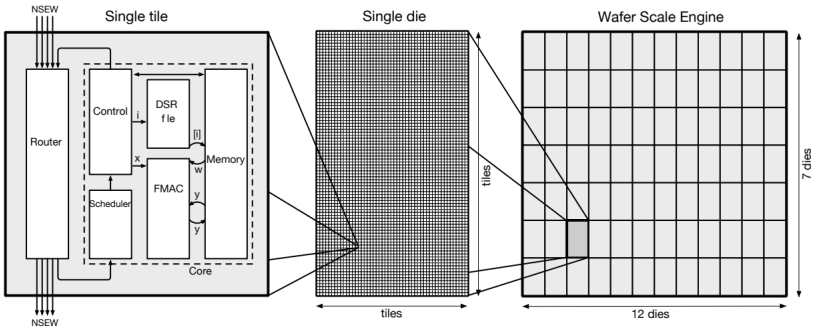

Figure 2.4: An overview of the Wafer Scale Engine (WSE). The WSE (to the right) occupies an entire wafer, and is a 2D array of dies. Each die is itself a grid of tiles (in the middle), which contains a router, a processing element and single-cycle access memory (to the left). In total, the WSE-2 embeds 2.6 trillion transistors in a silicon area of 46,225 mm2. [3]

The absence of off-chip memory is a distinctive trait of the WSE-2. Unlike other architectures, the WSE-2 doesn't rely on dynamic random access memory (DRAM). Instead, it exclusively employs single-cycle latency static random access memory (SRAM), providing approximately 40 GB of memory. This unique characteristic positions the architecture as a potentially excellent option for HPC simulation kernels that are limited by memory bandwidth or latency. The theoretical bandwidth of the WSE-2 is remarkable, reaching 20 petabytes/second, surpassing the A100 GPU, which typically offers 1.5–2.0 terabytes/second bandwidth depending on the specific model.

The Cerebras System (CS) is a self-contained rack-mounted system containing packaging, power supply, cooling and I/O for a single WSE. The CS communicates via parallel 100 Gigabit ethernet connections to a host CPU cluster. Throughout this documentation,

the CS is referred to as the "device," the host CPU cluster as the "host," and the ethernet connections connecting the two as "host I/O". The SDK provides mechanisms for using host I/O to move data between host and device or launch functions on the device.

The below figure gives a visual representation of the mesh of PEs that make up the WSE, and its connections to the outside world. Data is streamed onto the device via host I/O, and enters the WSE through a series of links along its edges. The programming model of the SDK abstracts away the details of these links, and allows the programmer to copy data from the host to arbitrary PEs on the device.

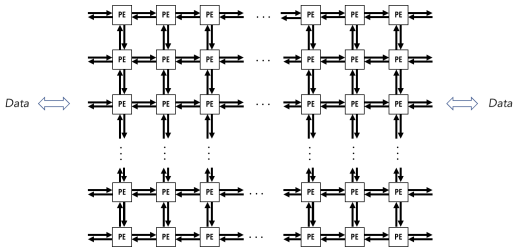

Figure 2.5: The PEs are interconnected cardinally allowing for communication between PEs in packets of size 32 bits.

The device allows programmers to interact with the cores using a lower-level code that targets the WSE's microarchitecture directly using a domain-specific programming language called the Cerebras Software Language, or CSL. In CSL, the cores on the WSE-2 are referred to as Processing Elements (PEs). The programmer can write code that targets every PE of the wafer such that compute and memory are optimally utilized. The programmer communicates to the device via a set of runtime APIs executed on one or more host computers. Although this device is primarily motivated towards AI workloads, This thesis explores the viability of applying the available hardware and compute power to other diverse workloads like Matrix Profiling.

2.2.1 Processing Elements

The 850,000 PEs on the WSE-2 are structured in a 2D grid. Each PE contains a general-purpose compute element (CE), a fabric router, and 48kB of local SRAM memory with single-cycle read/ write access latency. PE-to-PE communication latency is also one cycle. The PE contains a network router with links to the CE and to the routers of the four nearest PEs in the north, south, east, and west directions. Communication is integrated into the instruction set, at single 32-bit word granularity, and is accordingly as fast as arithmetic. Using the Cerebras SDK, each WSE-2 exposes up to 750×994 user

programmable PEs (i.e., 745,500 PEs). While the WSE-2 hardware actually contains about 850,000 PEs in total, some additional rows and columns around the user-space PEs are reserved for memory movement operations (to facilitate abstractions for moving data to/from the host) and other system functions The CE's instruction set supports FP32, FP16, and INT16 data types. The Cerebras ISA supports optimized vector operations for processing tensors as well as general-purpose control instructions. A CE can execute vector instructions that perform up to eight operations per clock for FP16 operands. Each PE is additionaly clocked at 850MHz. Each program is allowed to allocate a certain number of PEs for executing the workload, henceforth, we will refer to this as the `Resource Rectangle`.

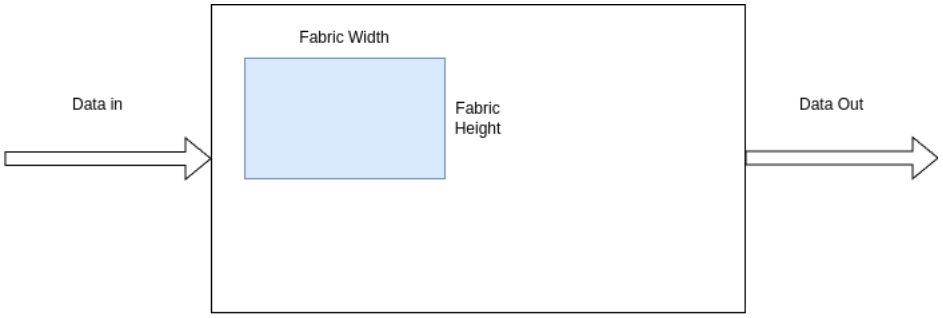

Figure 2.6: A `Resource Rectangle`, allocating a partial rectangle of the Wafer for computation

2.3 Related Works

The Cerebras WSE-2 has been used by various organisations, including large global corporations, for accelerating machine learning and other purposes. Some interesting AI breakthroughs, the porting of the GPT model to Cerebras [2], The benefits of accelerating machine learning workloads has been well proven, however there are far fewer studies concerned with using the WSE to run more traditional computational tasks. Already, there have been numerous notable successes from running Stencil computations for solving the 3D wave equation using finite-difference method in seismic imaging applications [4], fast Fourier transforms for one, two, and three-dimensional arrays [5], Efficient algorithm for Monte Carlo particle transport [6], Stencil computations reaching up to 503 TFLOPs on WSE-2, a figure that only full clusters can eventually yield [1]

This work is heavily based on the SCAMP paper introduce by Z. Zimmerman et a [9]. The approach to parallelism introduced and optimized by the SCAMP algorithm

could be easy mapped onto the distributed MIMD architecture of Cerebras (See Section 3). When calculating the Matrix Profile aggregates of a row or column, the algorithm does not contain any data dependency across rows or columns which allows us to exploit weak scaling which the Cerebras system is aimed at. This form of algorithm is widespread in scientific computing and hence represents the underlying computational pattern in use by a large number of HPC codes.

3 Design

The previous chapter has introduced the reader to the concept of matrix profiles and their efficient computation and key concepts of development on Cerebras Wafer Scale Engine. The following chapter presents the top-level architecture as a combination of a Host Application and an Accelerated Kernel by utilizing this background. The presented design constitutes a tiled version of the algorithm described in Subsection 2.1.2 and can, therefore, decouple the problem size from the concrete kernel implementation. Given the hardware capabilities of the WSE-2, a Tiled version of the Kernel is introduced in the chapter and explores the various types of tiling solutions that can be incorporated.

We then explore the various implications of resource allocation on the WSE-2 on the memory bandwidth, computation time and startup time.

We finally look at various scheduling approaches for large and small time series.

3.1 Architecture

The presented design is based on a architecture consisting of 3 distinct components, namely the `Wrapper Application`, `Host Application` and the `CSL Kernel`. The `Wrapper Application` computes the tiles required for the timeseries and passes the tiles to the `Host Application`. The `Host Application` loads the timeseries and prepares the data to be broadcasted to the *WSE Fabric*. The `CSL Kernel` performs the actual computation described in Subsection 2.1.2 on the WSE. A visualization of this architecture can be found in Figure 3.1.

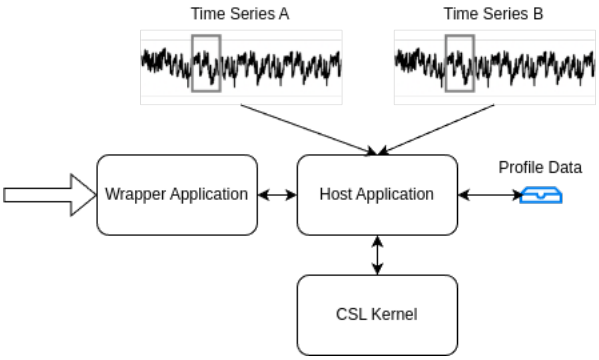

Figure 3.1: Architecture

The number of iterations of `Host Application` is dependant on the number of `Tiles` required to be processed and the size of the timeseries.

As outlined in Figure 3.1, the computation can be divided into four distinct phases:

1. **Configure Run** `Wrapper Application`: Configures the parameters for the run. Decides the `Tile Size` and `Window` and the dimensions of the `Resource Rectangle` to be allocated for the run and splits the problem space into smaller chuncks which can be executed by the `Resource Rectangle`.

2. **Pre-Computation** `Host Application`: The Host Application picks up the configuration from the Wrapper After the input time series is read, the statistics, i.e., df, dg, and the L2-norms inverses are computed. The Host Application also computes the kernel arguments for each `Tile`. The host application also initializes the Simulator/Hardware and specifies the code to be pushed to the PEs that are part of the `Resource Rectangle`

3. **Computation** `CSL Kernel`: Once the data is transfered, The Host Application calls the *compute* function which triggers the computation on all PEs allocated in the `Resource rectangle`. After all tiles have successfully been computed by the kernel, the data is returned to the Host Application.

4. **Post Computation** `Host Application` The next step in the computation is to merge the MP and MPI results from the tiles into a single Matrix Profile.

3.1.1 CSL Kernel

Cerebras Software Language, or CSL is similar to syntax of Go, Swift and Scala. It is designed for writing high-performance programs for PEs on the WSE. CSL is a

medium-level language, in that it exposes both low and high-level programming features. CSL is low-level in that it exposes the capabilities of the instruction set and some of the hardware structures that the ISA controls. On the other hand, CSL is a high-level language in that it supports structured control flow, loops, if-then-else constructs, and hides processor registers, performing automatic register allocation. CSL allows for compile time execution of code blocks that take compile-time constant objects as input, a powerful feature it inherits from Zig, on which CSL is based. CSL will be largely familiar to anyone who is comfortable with C/C++, but there are some new capabilities on top of the C-derived basics.

The main entry point to a CSL program is through defining the `Resource Rectangle` for the given program as depicted in Figure 3.2. This specifies the number of PEs that needs to allocated for computation as detailed in Figure. The language also allows us to flexibily specify the code to be pushed into a specific PE that can be executed during runtime of the program. For our usecase, we have a single kernel implementation that is being pushed into every single PE that is allocated for the given computation. Below is a piece of code that allocates a `Resource Rectangle` of size `WIDTH`, `HEIGHT` and pushes the code `tile_kernel.csl` to each of the PE. There is also the provision to pass params to each PE and the ability to modify them based on the requirements.

```
@set_rectangle(WIDTH, HEIGHT);
for (@range(i16, WIDTH)) | x | {
  for (@range(i16, HEIGHT)) | y | {
      @set_tile_code(x, y, "tile_kernel.csl", .{
          .memcpy_params = memcpy.get_params(x),
          .LEN = LEN,
          .WINDOW = WINDOW,
          .N = LEN - WINDOW + 1,
      });
  }
```

We will go over the main kernel algorithm in the upcoming Subsection 3.3 that is implemented on the device.

3.1.2 Host Application

The `Host Application` provides a path to the binaries created by the CSL compiler and specifies the name of the (simulated/real hardware) core that should be written to post simulation. Once the application has finished running on the simulator, the host reads the result from the core. The result is then stored onto a file for later comparing it's accuracy.

The Host Application also prepares the variables inv, df, dg for computation and concatenates the required data into a single array that is then pushed onto the device and spread across the allocated PEs.

3.1.3 Wrapper Application

The Wrapper Application is a wrapper of the `Host Application` and allows flexible execution of different tile sizes, partial matrix profiles and different *CSL Fabric* configurations

3.2 Kernel Design

3.2.1 Vanilla Kernel

The `Vanilla Kernel` employs the update formulation described in Equation 2.6 by iteratively computing parts of the distance matrix contained within the current diagonal chunk. The entire entire execution of the Vanilla Kernel is contained within a single unit of execution. Its concept is rather simple, and it provides a good foundation to

explain the `Tiled Kernel` in the following subsection. Subsequently, we describe the computation performed by the Vanilla Kernel:

1. First, the initial set of QT values is used to initialize the internal variables (line 1). Additionally, the local aggregate representations are initialized (lines 2 - 3). As we calculate Pearson correlations instead of the Euclidean distances, as described in the previous section, aggregates are initialized with $-\infty$ instead of ∞.

2. Next, the incremental update formulation, as described in Equation 2.6, is utilized to compute successive QT rows (line 7), and Equation 2.3 is employed to determine the Pearson correlation (line 8). Subsequently, the aggregates are updated accordingly (line 9 - 12).

3. Once all diagonals within the current chunk have been computed, the aggregates are returned (line 15).

Pseudocode for the Vanilla Kernel can be found in Algorithm 2.

Algorithm 1 Vanilla Kernel

Input : Time Series T of length $n \in \mathbb{N}$ and subsequence length $m \in \mathbb{N}$
Output : Matrix Profile MP and Matrix Profile Index MPI

1: $df, dg, inv \leftarrow PreComputeStatistics(T, m)$;
2: $QT_{init} \leftarrow PreComputeInitialQTRow(T, m)$;
3: $rowAggregates, columnAggregates \leftarrow (-\text{inf}, -1)$;
4: **for** $iteration \leftarrow 0$ **to** $\lceil \frac{n-m+1}{1} - 1 \rceil$ **do**
5: $\quad iteration_i \leftarrow MatrixProfileKernel(QT_{init}$,df,dg,inv$)$;
6: $\quad UpdateAggregates(rowAggregates, columnAggregates, iteration_i)$;
7: **end for**
8: $PostCompute(rowAggregates, columnAggregates)$;
9: **return** MP, MPI;

3.2.2 Tiled Kernel

SCAMP improves on the vanilla kernel for exploiting the parallelizable nature of the problem by implementing a systolic array architecture as described in Subsection 2.2.2. As explained in Equation 2.6, the computation of $QT_{i,j}$ solely requires the QT value diagonally above ($QT_{i-1,j-1}$) and precomputed statistics, enabling every diagonal to be computed independently. This can be exploited by instantiation multiple processing elements, of which each computes a set of t (tile size) diagonals independently.

Therefore, the `Tiled Kernel` subdivides the diagonal chunk once again. In constrast to the initial division, which aimed to decouple the problem size from the concrete kernel implementation, this subdivision exploits inherent parallelism. This division is visualized in Figure 3.3.

Every processing elements keeps track of it's row- and column-wise aggregates (lines 17 - 4) and update them accordingly, similar to the Vanilla Kernel (lines 10 - 13). After all processing elements have completed their computation, the individual aggregates can be reduced into a single set of aggregates (lines 16 - 22) which are subsequently returned.

Algorithm 2 SCAMP Tiled Algorithm

Input : The current Iteration i, $\overline{QT}_{init}$, as well as df, dg and inv.
Output : Row- and Column-Wise Aggregates for the current **Diagonal Chunk**

for $ProcessingElement \leftarrow 0$ **to** $\lceil \frac{w}{t} \rceil - 1$ **do**
  $rowAggregates_{ProcessingElement} \leftarrow (-\infty, -1)$;
  $columnAggregates_{ProcessingElement} \leftarrow (-\infty, -1)$;
  for $row \leftarrow 1$ **to** h_i **do**
    for $k \leftarrow 1$ **to** t **do**
      $column \leftarrow i \cdot w + ProcessingElement \cdot t + row + k - 1$;
      $QT_k \leftarrow QT_k + df_{row} \cdot dg_{column} + df_{column} \cdot dg_{row}$;
      $P \leftarrow QT_k \cdot inv_{row} \cdot inv_{column}$;
      if $P >$ is $rowAggregate_{ProcessingElement,row}.value$ **then**
        $rowAggregate_{ProcessingElement,row} \leftarrow (P, column)$;
      end if
      if $P >$ is $columnAggregate_{ProcessingElement,column}.value$ **then**
        $columnAggregate_{ProcessingElement,column} \leftarrow (P, row)$;
      end if
    end for
  end for
end for
$rowAggregates \leftarrow (-\infty, -1)$;
$columnAggregates \leftarrow (-\infty, -1)$;
for $ProcessingElement \leftarrow 0$ **to** $w * h$ **do**
  $Merge(rowAggregates, rowAggregates_{ProcessingElement})$;
  $Merge(columnAggregates, columnAggregate_{ProcessingElement})$;
end for
return $rowAggregates, columnAggregates$;

Porting the SCAMP C++ Matrix Profiling Kernel to the Cerebras WSE engine presented several significant challenges, primarily stemming from the fundamental differences in architecture between conventional CPUs/GPUs and the specialized wafer-scale integration of the Cerebras device. The initial hurdle we encountered was defining the appropriate parameters and effectively distributing the workload onto the smaller processing elements (PEs) inherent to the Cerebras WSE architecture.

One of the foremost challenges lay in reconciling the intricacies of the SCAMP codebase, originally optimized for CPUs and GPUs, with the unique features and constraints of the Cerebras WSE. The SCAMP algorithm was intricately tailored to leverage specific compiler optimizations and architectural characteristics of conventional

computing platforms, rendering direct translation to the Cerebras environment non-trivial. Thus, comprehensive modifications and optimizations were essential to ensure compatibility and performance efficiency on the novel hardware architecture.

In order to parallelize the Matrix Profile algorithm, the entire distance matrix calculated using Equation 2.9 is divided into square tiles of a maximum tile size. For a one-dimensional time series where the point of interest lies only in the tiles in the upper diagonal, the number of tiles T in SCAMP is calculated using the formula:

$$T = \frac{N \cdot (N+1)}{2} \tag{3.1}$$

where:

$$N = \frac{S}{T.S}$$

where S is the size of the time series and $T.S$ is the maximum size of the tile.

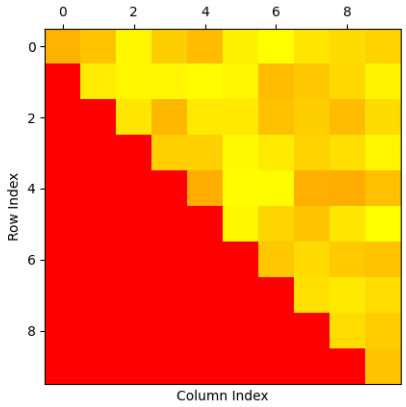

Figure 3.2: Tiling Representation

Rather than computing the entire distance matrix in one operation, we split it into tiles . Each tile independently computes an AB-join between two segments of the input time series. This allows the computation to scale to very large input sizes and distribute the work to many independent machines We finally settled on the following algorithm which is similar to the stripped down version of the SCAMP cpp implementation. Similar to the SCAMP version, The Statistics (df, dg, inv) are computed on the Host Device and then along with the Time series T_a, T_b, passed on to the Device and distributed across the allocated `Resource Rectangle`. Once each device has recieved data, then the kernel computation is triggered from the host. Each PE is responsible to compute the aggregates for the particular tile since there are no data

dependencies across tiles. The aggregates are then reduced on the host device. This was a design decision made given the constraint on the Cerebras system for communication across PEs.

Algorithm 3 Cerebras Tiled Algorithm

Input : Time series T_a, T_b, df, dg, inv and $args$ for current **Tile**.
Output : Row- and Column-Wise Aggregates for the current **Tile**

for $ProcessingElement \leftarrow 0$ **to** $w * h$ **do**
  $rowAggregates_{ProcessingElement} \leftarrow (-\inf, -1)$;
  $columnAggregates_{ProcessingElement} \leftarrow (-\inf, -1)$; ▷ Compute only top triangle
  $QT \leftarrow computeQT(T_a, T_b)$;
  for $row \leftarrow 0$ **to** $min(args.n_x - args.exclusion\_lower, args.n_y)$ **do**
    for $diag \leftarrow args.exclusion\_lower$ **to** $min(args.n_x - args.exclusion\_upper + 1, args.n_x - row)$ **do**;
      $col \leftarrow row + diag$;
      $correlation \leftarrow QT_{diag} \cdot inv_{row} \cdot inv_{col}$;
      if $correlation > rowAggregate_{ProcessingElement,row}.value$ **then**
        $rowAggregate_{ProcessingElement,row}.value \leftarrow (correlation, column)$;
      end if
      if $correlation > rowAggregate_{ProcessingElement,column}.value$ **then**
        $rowAggregate_{ProcessingElement,column}.value \leftarrow (correlation, row)$;
      end if
    end for
  end for
end for
$rowAggregates \leftarrow (-\inf, -1)$;
$columnAggregates \leftarrow (-\inf, -1)$;
for $ProcessingElement \leftarrow 0$ **to** $w * h$ **do**
  $Merge(rowAggregates, rowAggregates_{ProcessingElement})$;
  $Merge(columnAggregates, columnAggregate_{ProcessingElement})$;
end for
return $rowAggregates, columnAggregates$;

Above is the final version of the Cerebras Tiled Kernel. It only implements computing the aggregates for one half of the triangle but the other half is computed by flipping the parameters hence computing the whole tile. We also explored other version of tiling on the Cerebras to see if they offer any benefit

3.3 Tiling Approaches

3.3.1 Trapezoidal

While we were looking at different tiling approaches to experiment on its impact on the performance and efficiency, we looked at Trapezoidal tiles which in theory has the same area as a square but does not allow us to elegantly handle border conditions where there are a small section or the tiles on the diagonal which contain only a upper triangle.

3.3.2 Triangles

Since we already have squares, the natural next step was to explore the implication of Triangle tiles. Although the kernel already supports triangles, The deconstruction of squares into rectangles leads to more jobs to be distributed across the Cerebras WSE. This also has an implication on the size of the `Resource Rectangle` since the number of triangles in the upper diagonal of the *Distance Matrix* is more than the number of squares when deconstructed, this leads to more resource consumption but lesser execution time. This also has an impact on the memory transfer time to and from the device since a square tile and 2 triangles require the same amount of data and require reduction across two different PEs.

3.4 Resource Allocation

3.4.1 Rectangle

- Memory Bus on the right side of the wafer of size 16. spread across the height. We noticed a improvement in memory transfer speeds when the PEs were spread across the height.

3.4.2 Square

- Need to do more experiments to get values

3.5 Scheduling

Given the 745,500 available PEs on the Wafer, we are limited to the size of time series we can compute in a single run. This is also aggregevated by the fact that we have a soft limit on the maximum `Tile Size` that can be allocated on a single PE of 100.

Given that real world time series data sets can go in order of Billion entries [1], We explore here the different scheduling approaches to compute large scale time series data efficiently.

3.5.1 One Shot

From Equation 3.1, We arrive at the total number of tiles that need to be computed for a given matrix of size S, When it is lesser than the allocatable number of PEs on the wafer/simulator, we can execute the entire Matrix Profile in a single iteration or *One Shot*

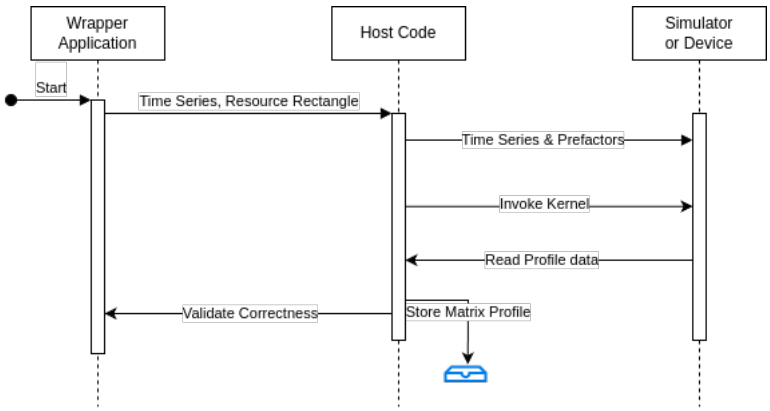

Figure 3.3: One Shot Execution of a small Time Series

The advantage of *One Shot* scheduling is the reduced overhead of PE allocation and fewer memory transfer. In Practice, we found

3.5.2 Iterative

- Time Series of lower sizes have a smaller matrix profile.
- Generates tiles of maximum size 100 which can fit in the available fabric.

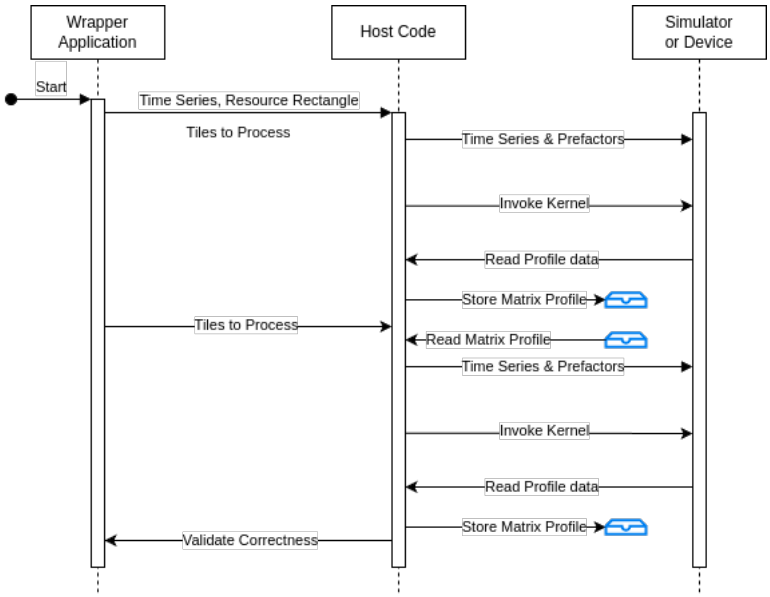

Figure 3.4: Iterative Execution of a large Time Series

4 Implementation

The following chapter describes how the design described in Section 3 is realized as an Cerebras accelerated application. This chapter uses the Cerebras SDK conventions and/or syntax as the underlying work targets the Cerebras WSE-2 hardware. Before potraying concrete implementation choices, we briefly introduce the reader to the Cerebras SDK and the WSE programming paradigm.

4.1 WSE Programming Environment

To develop code for the WSE, we write device code in CSL, and host code in Python. we then compile the device code, and run the program on either the Cerebras fabric simulator, or the actual network-attached device. The host code is responsible for copying data to and from the device, and launching discrete programs referred to as kernels.

4.1.1 Execution Model

As described in Section 3.1, the design is split between a `Host Code` and a `CSL Kernel`. The `Host Code` is written in Python and uses the Cerebras SDK. It provides a host runtime known as *SdkRuntime*, and associated functionality in the CSL memcpy library, to load programs, launch functions, and transfer data on and off the Wafer-Scale Engine.

The interaction between Host Code and kernel consists of four distinct stages:

1. The `Host Code` prepares the data for the given timeseries, sets up the kernel, connects to the Device/Simulator and setups up the required amount of Computing resource (`Resource Rectangle`) and transfers it to the device.
2. The execution of the kernel function on the PEs is triggered by the Host code.
3. Once all the PEs have performed the required computation, It triggers the host that it's ready to accept commands from the Host Code.
4. Finally, the Host code copies back the result into host memory.

4.1.2 Build Process

The build process is resembles a standard compilation for the device code which is compiled using the `cslc` compiler toolchain. It outputs binaries which are then picked up by the Python Host Code to be transfered to the device to all the allocated PEs.
The build process is quite straightforward but is accompanied with a build script written with GNU make-files. The underlying work provides a more convenient Python wrapper that builds and runs the code on the Simulator/Device and verify the results.

4.2 Host Code

In this section, we discuss the implementation specifics of the `Host Code` outlined in Section 3.1. As previously mentioned, the Host code starts by loading the input time series from a file specified via a command line argument $(--file_a)$. The Host code also allows us to process only specific tiles for computation or the entire time series. It then performs the required pre-computation for the two time series. The Code then initiates a `SdkRuntime` from the cerebras library which connects to and initializes the device or starts the Cerebras fabric simulator. The `SdkRuntime` populates the allocated PEs with the binaries and provides access to symbols exposed by the kernel program.

The Host Code then prepares the data required for computation

$$T_a, T_b, dg_a, dg_b, df_a, df_b, inv_a, inv_b, args$$

and pushes them to the device. One caveat here is the fact that the arrays declared in the kernel are sized statically and therefore require padding from the host with precise sizes.

```
runner.memcpy_h2d(T_A_symbol, np.array(T_a, dtype=np.float32),
                  0, 0, width, height,
                  MAX_TILE_SIZE + WINDOW, streaming=False,
                  order=MemcpyOrder.ROW_MAJOR,
                  data_type=MemcpyDataType.MEMCPY_32BIT,
                  nonblock=False)
runner.memcpy_h2d(T_B_symbol, np.array(T_b, dtype=np.float32),
                  0, 0, width, height,
                  MAX_TILE_SIZE + WINDOW, streaming=False,
                  order=MemcpyOrder.ROW_MAJOR,
                  data_type=MemcpyDataType.MEMCPY_32BIT,
                  nonblock=False)
```

Figure 4.1: Memory Copy to the device, The SDK allows us to define the order of exporting data along the PEs and the symbol to which the data needs to be transferred

```
runner.launch('compute', nonblock=False)
```

Once the data is transfered, The kernel function $compute()$ is invoked. Once the kernel function is complete and the device to recieve more commands, The tile aggregates are then extracted from the device and merged based on the tile positions.

```
runner.memcpy_d2h(MP_A_result, MP_A_symbol,
                  0, 0, width, height,
                  MAX_TILE_SIZE - WINDOW + 1, streaming=False,
                  order=MemcpyOrder.ROW_MAJOR,
                  data_type=MemcpyDataType.MEMCPY_32BIT,
                  nonblock=False)
runner.memcpy_d2h(MP_B_result, MP_B_symbol,
                  0, 0, width, height,
                  MAX_TILE_SIZE - WINDOW + 1, streaming=False,
                  order=MemcpyOrder.ROW_MAJOR,
                  data_type=MemcpyDataType.MEMCPY_32BIT,
                  nonblock=False)
```

Figure 4.2: Memory Copy from the device, Note the explicit PEs width and height declaration

4.3 Kernel Implementation

As elaborated in Section 3.4, The Kernel implements a tiled version of Matrix Profiling algorithm and each PE is capable of computing the row- and column-aggregates of a single tile with a upper bound on `Tile Size` of 100. Each kernel in addition gets the following args for computation.
n_x, n_y: They represent the actual boundaries of the tile and impact the runtime of the kernel.

$exclusion\_lower_u, exclusion\_upper_u, exclusion\_lower_b, exclusion\_upper_b$: The exclusion boundaries provide a exclusion boundaries for the two triangles that are part of the tile and are taken into account when setting the kernel

$full\_tile$: The specifies the kernel if it has to compute just the upper triangle or both upper and the bottom triangle of the square tile.

The Kernel then computes the current rows $\overline{QT}$ for the two time series at that particular point using the following function.

```
fn compute_cov(T_a: [*]f32, T_b: [*]f32) [*]f32
```

Once $\overline{QT}$ is calculated then the kernel function is launched which calculates the row- and column-aggregates with the appropriate precomputed factors dg, df and inv to the following kernel function.

```
fn kernel(norm_a: [*]f32, norm_b: [*]f32,
          dg_a: [*]f32, dg_b: [*]f32,
          df_a: [*]f32, df_b: [*]f32,
```

```
P_a: *[N]f32, P_i_a: *[N]i32,
P_b: *[N]f32, P_i_b: *[N]i32,
cov: [*]f32, n_x: u8, n_y: u8,
exclusion_lower: u8, exclusion_upper: u8) void
```

The precomputed factors are flipped based on weather it is computing the upper tile or the bottom tile.

Once the entire tile is completed, the `rowAggregates` and `columnAggregates` are then transfered to the host from all the PEs. The Host then reorders the aggregates based on tile information into a single Matrix Profile.

5 Experiments and Results

This chapter delves into the empirical exploration conducted to understand the interplay between various control variables and the performance of Matrix Profiling algorithms on the Cerebras Wafer Scale Engine (WSE).

In this study, we systematically investigate the influence of four pivotal control variables: `Time Series`, `Tile Size`, `Window Size`, and `Resource Rectangle`, on the efficiency and accuracy of Matrix Profiling computations on the Cerebras WSE. By meticulously varying these parameters and analyzing their impact on computational metrics such as execution time, memory utilization, and result fidelity, we aim to unravel insights crucial for optimizing Matrix Profiling workflows on this cutting-edge hardware platform.

Within this chapter, we outline our experimental methodology, detailing the setup, execution, and analysis of Matrix Profiling experiments on the Cerebras WSE. Through rigorous empirical investigation, we endeavor to shed light on the intricate relationship between control variables and Matrix Profiling performance, ultimately contributing to the advancement of time series analysis techniques on state-of-the-art computing architectures.

Our findings hold implications for researchers and practitioners seeking to harness the potential of novel hardware platforms like the Cerebras WSE for accelerating Matrix Profiling tasks. By elucidating the optimal configurations and parameters for Matrix Profiling experiments, we aim to facilitate the development of scalable, high-performance solutions tailored to the demands of modern data analysis workflows.

5.1 Model

In this chapter, we delve into a series of experiments aimed at elucidating the impact of various control variables on the performance and efficiency of Matrix Profiling algorithms executed on the Cerebras Wafer Scale Engine (WSE). These experiments are guided by the following control variables:

- **Time Series (*s*)**: The length and complexity of the time series data under analysis.
- **Tile Size (*t_s*)**: The size of the Tile allocated to a single PE, which directly influences the parallelism and computational efficiency.
- **Window Size (*m*)**: The size of the sliding window used for local pattern discovery within the time series data.
- **Resource Rectangle (*r_r*)**: The configuration of resources allocated within the Cerebras WSE for executing the Matrix Profiling algorithm.

These control variables serve as the foundation for our experimental design, allowing us to systematically explore their implications on the overall execution and efficiency of the Matrix Profiling algorithm on the Cerebras WSE.

The Matrix Profiling algorithm exhibits a time complexity of $O(n^2)$, where n represents the length of the time series data. This runtime complexity is directly influenced by the `Tile Size` parameter, as it dictates the granularity of computation performed by individual PEs within the Cerebras WSE.

Our experimental design implicitly assumes a one-to-one mapping between the number of tiles in the given time series data and the number of PEs allocated on the Cerebras WSE. This constraint arises from the memory limitations of individual PEs and the lack of support for streaming data in and out of the fabric within the Cerebras architecture.

Each of the following experiments introduces the theoretical model for the experiments first and the conclusions derived from them, and then the empherical results from the simulator and the hardware. The simulator provides only the number of cycles spent on each individual PE and we derive measurements from the same.

- Insert Simulator Hardware parameters here.

The Hardware experiemnts were run on the

- Insert Hardware execution parameters here.

5.2 Execution Time

The following section looks into the Execution time for the main kernel and overall execution on the Cerebras WSE-2.

To determine the runtime of the kernel, the CSL language exposes hardware clock cycle timer data that can be queried and saved over the runtime of a kernel and reported back to the host. The total wall time of a kernel can be computed by determining the maximum number of cycles used by any PE during the kernel and dividing it by the clock rate of the WSE-2 (850 MHz). While some degree of thermal throttling will occur, the WSE-2 implements throttling by injecting "nop" commands rather than by adjusting the clock speed itself, such that any thermal "nop" cycles are included when measuring kernel cycle counts. This method of measuring kernel runtime performance by recording clock cycles is used throughout this paper whenever runtime data is reported and is the standard method for recording performance data on Cerebras machines [3]

5.2.1 Kernel Execution Time

The algorithm to compute the Matrix profile as defined in Section 3.4, is an $O(n^2)$ algorithm and hence, the execution time is directly propotional to the tile size. We gradually increased the tile size (t_s) across the experiment and keeping all the other parameters constant. Since the *Distance Matrix* needs to be decomposed into smaller or larger tiles, this has a implication on the size of `Resource Rectangle`

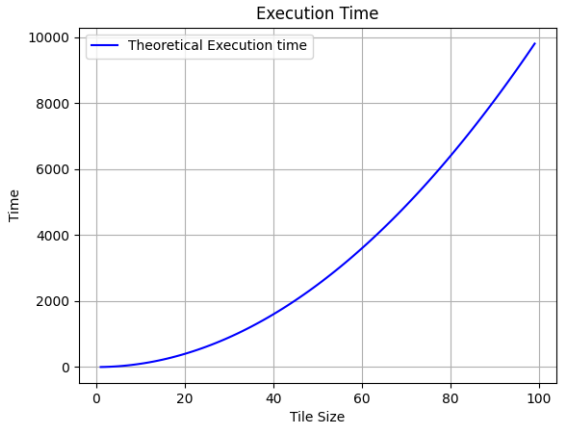

Figure 5.1: Plot of Observed Execution time of different tile sizes on the Simulator

Floating point operations per second

With the empherical values obtained from the simulator,

$$\text{FLOPs} = 2 \times (6 \times (t\_s \times t\_s) + 2 \times m + 7 \times t\_s)$$

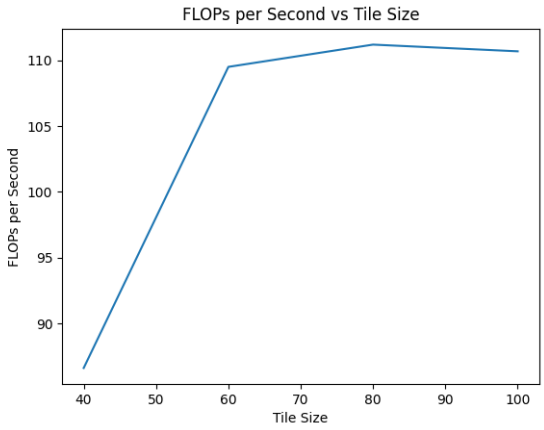

Figure 5.2: Floating point operations vs tile size

5.2.2 Overall Execution time

To look at the performance of the entire program and not just at the kernel execution time, we focused on small to larger datasets of $s = 1000 \ldots 100000$.

The following experiments are conducted with a random walk time series of varying sizes (s), a constant tile size (t_s) of 100 a constant window size (m) of 6. The increasing Time series sizes were chosen to capture the impact of memory transfers and kernel execution on larger datasets since it transmits a increasingly more data and computes on a greater `Resource Rectangle`. We reached a

5.3 Memory

Matrix profiling is an memory intensive algorithm since it involves a lot of prefactors in the computation, The Cerebras SDK does not expose any metrics on Memory consumption, Hence, we present our mathematical model on memory usage on a single PE and compare it to what was practically possible. In order to arrive at a model of

memory consumption for a single tile, we need to look at the variables involved in the computation of a single tile of size s and for a window of size m. therefore,

$$M = F\left(MP_a, MPI_a, MP_b, MPI_b, T_a, T_b, inf, df, dg, args\right); \tag{5.1}$$

$$M = sizeof(2 * (s + m) + 10 * (s - m + 1) + 7)$$
$$M = (12 * s - 8 * m + 17) * 32/8/1024$$

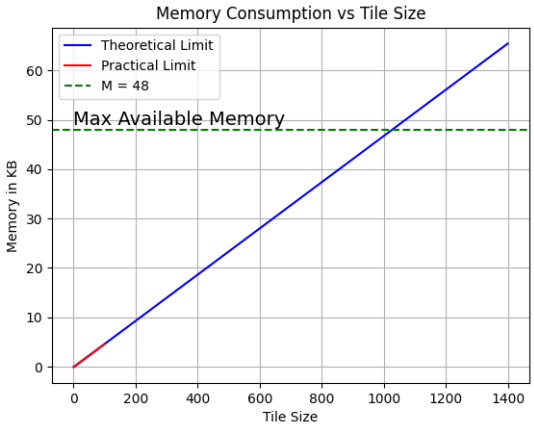

Figure 5.3: Memory Usage of a single PE plotted against Tile Size. The theoretical limit to the tile size is 1010 but what is observed is a maximum limit of 100 per PE. Practically, we were able to use only 4.566 KB of memory of the PE.

We reaching a practical limit of 4.566 KB. We encountered compilation issues on the Cerebras compiler that did not allow us to allocate larger arrays and led to a stack size issue. Sadly, we couldn't further investigate into the reason behind it due

5.4 Weak Scaling Experiments

Since the Cerebras WSE-2 is primarily focused on weak scaling, We look at the possible implications of Weak Scaling on the device using various configuration and analyse the implication on memory transfer speeds and execution time. We begin our weak scaling study by fixing the window size and tile size, m and t_s to 100 and 6 respectively and increase the time series size gradually. We choose to use to random walk algorithm to generate our time series. The sensitivity in performance to s is also investigated. We choose to create `Resource Rectangles` of different sizes to study the impact varous resource allocation schemes on the execution time of the algorithm.

We expect linear growth in the total throughput of the kernel

5.5 Precision Evaluation

Given the unpredictable nature of real-world data, which often includes numerous constant areas, numerical instability arises as a concern. The similarity of z-normalized subsequences is of particular interest to many researchers. In constant regions, the standard deviation is zero. Additionally, near-constant subsequences pose challenges as they may pass a bit-level test for two distinct values, yet result in division by a number nearly approaching zero.

In this section, we take a look at the impact of Cerebras lower precision floating point limitation. We compare the error study provided by SCAMP on single precision [9] to simulated random time series on different distribution. Since Cerebras works only with reduced floating point precision values, It leads to a loss of information in the MP when compared to a double precision algorithm. Although the difference is subtle, it is noticable on larger data sets.

Table 5.1: Maximum absolute error (Pearson Correlation) for various datasets on SCAMP in single precision. [9]

Maximum Absolute Error	Size (m)	SCAMP SP
Whitefly	2.5M (1000)	$3.75*10^{-2}$
ECG	8.4M (100)	$3.14*10^{-4}$
Earthquake	1.7M (200)	$6.35*10^{-1}$
Power Demand	10M (4000)	$4.85*10^{-2}$
Chicken	9M (1000)	$4.92*10^{-2}$

These results are comparitive to the double precision version of SCAMP, hence, the results are very similar. Since we did not have the compute power to execute such large datasets, we setup an simpler experiment involving different random distributions as shown in Figure 5.3. Although this does not provide a simulation of real life dataset, we wanted to look at anomolies in error rates across different distributions.

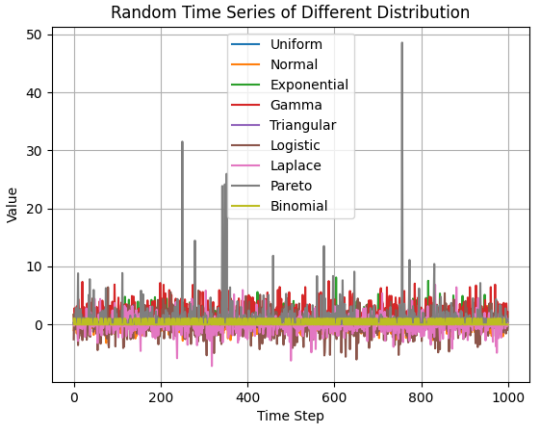

Figure 5.4: Time series of different random distributions

5.6 Comparison to CPU & GPU Baseline

Now that we have enough experimental results from the Hardware and Simulator, We can compare the performance of the Cerebras WSE-2 with traditional GPUs. In Table

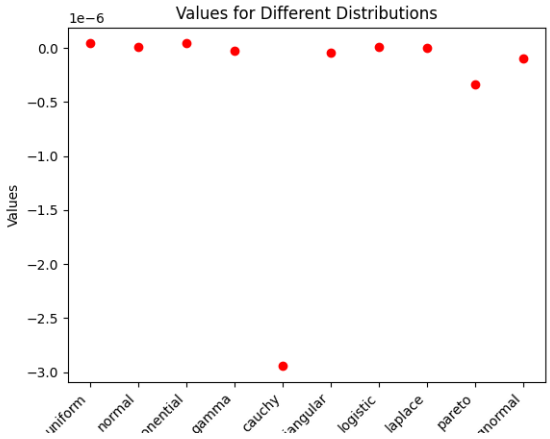

Figure 5.5: Error rate on different probabilistic distributions

4.1, we take a look at the SCAMP performance on random walk datasets of various lengths [9] and compare them to the runtime of similar random walk time series on the Cerebras WSE-2 simulator on a fabric of size $w = 500$ and $h = 500$ executed using iterative scheduling.

Table 5.2: Scamp Runtime

Implementation	SCAMP-GPU		Cerebras WSE-2
Architecture	V100	V100	
Precision	DP	SP	SP
2^{18}	0.28s	0.24s	
2^{19}	0.68s	0.57s	
2^{20}	2.05s	1.42s	
2^{21}	6.99s	4.38s	
2^{22}	25.8s	15.5s	
2^{23}	96.8s	52.5s	

5.7 Execution on Real world datasets

Time series	n	m	Max	Min	Up.%
5	88	788	6344		

We used a diverse set of time series for the experimental evaluation, whose parameters are shown in Table 1. We can see the number of samples, n, and the window size or subsequence length, m, used for each series, as well as the maximum and minimum values. The name of each series is descriptive of the type of content they contain. Note that all time series used in this work come from real data [33]. The last rows of the table identify the series used to evaluate the proposals with large data sets, including an extra-large one of nearly 11M samples [18]. The last column shows the percentage of all profile update attempts that indeed write the matrix profile and matrix profile index arrays, using the baseline SCAMP algorithm with one thread; i.e., the if statements in Lines 4, 4, 4 and 4 in Fig. 4 which become true.

6 Experiences

6.1 Cerebras WSE-2

6.2 Workflow

6.3 Build Process

6.4 Scaling

7 Conclusion

List of Figures

List of Tables

Bibliography

[1] H. A. Dau, E. Keogh, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, Yanping, B. Hu, N. Begum, A. Bagnall, A. Mueen, G. Batista, and Hexagon-ML. *The UCR Time Series Classification Archive*. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/. Aug. 2018.

[2] N. Dey, G. Gosal, Zhiming, Chen, H. Khachane, W. Marshall, R. Pathria, M. Tom, and J. Hestness. *Cerebras-GPT: Open Compute-Optimal Language Models Trained on the Cerebras Wafer-Scale Cluster*. 2023. arXiv: 2304.03208 [cs.LG].

[3] M. Jacquelin, M. Araya-Polo, and J. Meng. *Massively scalable stencil algorithm*. 2022. arXiv: 2204.03775 [cs.MS].

[4] M. Jacquelin, M. Araya-Polo, and J. Meng. "Scalable distributed high-order stencil computations." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022. ISBN: 9784665454445.

[5] M. Orenes-Vera, I. Sharapov, R. Schreiber, M. Jacquelin, P. Vandermersch, and S. Chetlur. "Wafer-Scale Fast Fourier Transforms." In: *Proceedings of the 37th International Conference on Supercomputing*. ICS '23. Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 180–191. ISBN: 9798400700569. DOI: 10.1145/3577193.3593708.

[6] J. Tramm, B. Allen, K. Yoshii, A. Siegel, and L. Wilson. "Efficient algorithms for Monte Carlo particle transport on AI accelerator hardware." In: *Computer Physics Communications* 298 (2024), p. 109072. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2023.109072.

[7] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. "Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets." In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 2016, pp. 1317–1322. DOI: 10.1109/ICDM.2016.0179.

[8] Y. Zhu, Z. Zimmerman, N. S. Senobari, C.-C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh. "Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins." In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 2016, pp. 739–748. DOI: 10.1109/ICDM.2016.0085.

[9] Z. Zimmerman, K. Kamgar, N. S. Senobari, B. Crites, G. Funning, P. Brisk, and E. Keogh. "Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 74–86. ISBN: 9781450369732. DOI: 10.1145/3357223.3362721.