DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Implementation and Evaluation of Matrix Profile Algorithms on the Cerebras Wafer-Scale Engine

Vyas Giridharan

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Implementation and Evaluation of Matrix Profile Algorithms on the Cerebras Wafer-Scale Engine

Entwicklung und Evaluierung von "Matrix Profile"-Algorithmen auf der Cerebras Wafer-Scale-Engine

Author:	Vyas Giridharan
Supervisor:	PD Dr. rer. nat. Josef Weidendorfer
Advisor:	M.Sc. Amir Raoofy
Submission Date:	March 15, 2024

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

Munich, March 15, 2024 Vyas Giridharan

Acknowledgments

First and foremost, I thank my advisor, Amir Raoofy, for his continuous support throughout the whole process of creating this thesis, his wise advice and the many fruitful discussions. I must also express my profound gratitude to my parents and friends for their continuous love and support.

This work was supported by the Edinburgh International Data Facility (EIDF) and the Data-Driven Innovation Programme at the University of Edinburgh.

Abstract

Time series data mining has become indispensable across various disciplines, from astronomy to medicine. The utilization of matrix profiles has greatly facilitated target analyses, such as motif or discord discovery, making them considerably easier. While the initial methods for calculating the matrix profile were inefficient, significant efforts have been devoted to improving their computation efficiency. Among these, the SCAMP algorithm stands out for its simplicity, parallelizability, and effectiveness.

Wafer scale technology offers the promise of high performance and competitive energy efficiency for various HPC workloads. In this thesis, we introduce a Matrix Profiling Algorithm, based on the SCAMP algorithm, tailored for efficient computation on the Cerebras WSE-2. Given the novelty of Wafer Scale Computing, we explored the capabilities of the device and highlighted the challenges associated with implementing such an algorithm under memory and compute constraints. Finally, we propose potential optimizations to serve as a foundation for future research.

Die Auswertung von Zeitreihendaten ist in verschiedenen Disziplinen unverzichtbar geworden, von Astronomie bis zur Medizin. Die Nutzung von Matrixprofilen hat Zielanalysen ermöglicht wie z. B. die Entdeckung von Motiven oder Diskord, erheblich vereinfacht. Weil die ursprünglichen Methoden zur Berechnung des Matrixprofils ineffizient waren, wurden erhebliche Anstrengungen unternommen, um ihre Berechnungseffizienz zu verbessern. Unter diesen Methoden zeichnet sich der SCAMP Algorithmus durch seine Einfachheit, Parallelisierbarkeit und Effektivität aus.

Die Wafer-Scale-Technologie verspricht eine hohe Leistung und eine wettbewerbsfähige Energieeffizienz für verschiedene HPC-Workloads. In dieser Masterarbeit stellen wir einen Matrix Profiling-Algorithmus vor, der auf dem SCAMP-Algorithmus basiert und für effiziente Berechnungen auf dem Cerebras WSE-2 zugeschnitten ist. Angesichts der Neuartigkeit des Wafer Scale Computing haben wir die Fähigkeiten des Geräts untersucht und die Herausforderungen aufgezeigt, die mit der Implementierung eines solchen Algorithmus unter Speicher- und Rechenbeschränkungen. Schließlich empfehlen wir potenzielle Optimierungsmöglichkeiten, die als Grundlage für zukünftige Forschungen dienen können.

Contents

1 Introduction

In recent years, time series analysis has emerged as a vital tool across various fields including seismology [**10.1093/gji/ggy100**], medicine [**Fox_2017**], and music [**Silva2016SiMPleAM**]. A fundamental aspect of time series data mining involves uncovering motifs and discords. The computation of motifs, discords [**1**], semantic segmentation [**8215484**], rule discovery, or clustering has historically been challenging. However, a novel structure known as the matrix profile has simplified these tasks. Consequently, efficient algorithms have been developed to compute the matrix profile, such as STAMP [**1**], STOMP [**2**], SCRIMP++ [**8594908**], and more recently, SCAMP[**4**]. SCAMP, in particular, has gained popularity, particularly within the realm of High-Performance Computing (HPC). Simultaneously, Wafer-Scale Computing promises performance benefits and energy efficiency gains over traditional horizontal scaling infrastructures. Over the years, there have been many studies on the efficiency of Wafer-Scale computing [**10460211**] when compared to traditional GPUs and CPUs. The Cerebras WSE-2 is a state-of-the-art Wafer-Scale Engine built for High-Performance Computing needs including the ever-increasing need for computing resources for AI and other HPC problems [**257870**], [**6**], [**3**], [**5**].

In this work, we present an implementation of Matrix Profiling on the Cerebras Wafer-Scale Engine 2. We first describe the design and its subsequent implementation. First, the reader is introduced to the concept (and required notation) of a matrix profile and the algorithm (SCAMP) in Chapter 2. We then take the reader through the design we chose to implement in Chapter 3 with a hierarchical approach of a Wrapper and Host Application with an Accelerated Kernel. We also describe our Tiled Kernel derived from the SCAMP algorithm. We conclude the chapter by exploring different Tiling and Scheduling approaches. In Chapter 4, we delve into our implementation and the specifics of the Cerebras WSE-2. Subsequently, we present a theoretical execution model and empirical results from experimentation in Chapter 5, performed on the device provided by the EPIF[1], part of the University of Edinburgh. Specifically, we show the performance of Matrix Profiling on the Cerebras WSE-2 and talk about the limitations and capabilities of the device.

Chapter 6 finally highlights our experience porting the Matrix Profiling algorithm onto the Cerebras WSE-2 device and Chapter 7 talks about possible optimizations for

[1] https://edinburgh-international-data-facility.ed.ac.uk/

future work.

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 utilized 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 [**1**] and [**2**]. 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 serves as both input and output for matrix profile computation, the calculations necessitate a finer-grained data structure known as a *subsequence*. For a visual depiction of a time series and a subsequence (illustrated with a grey background), refer to Figure 2.1 In this thesis, we refer to the subsequence in our implementation as the `Window`.

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.

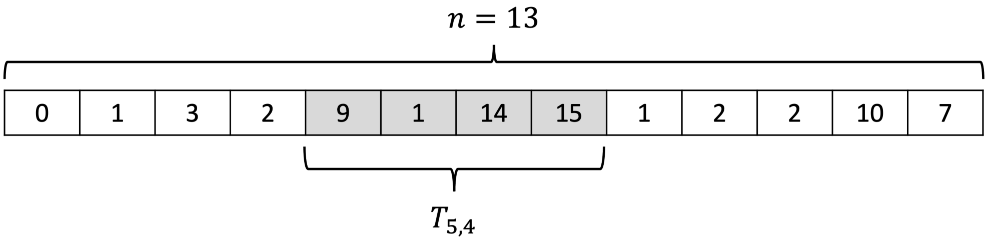

Figure 2.1: Time Series of length n = 13 and Subsequence $T_{5,4}$

Calculating distances between subsequences forms the crux of matrix profile computation. We define the (*z-normalized Euclidean*) *distance* between *time series*, and consequently *subsequences*, as follows:

Definition 3

The *z-normalized Euclidean distance* between two time series $X = x_1, x_2, \dots, x_m$ and $Y = y_1, y_2, \dots, y_m$ 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 $\bar{y}_i = \frac{y_i - \mu_Y}{\sigma_Y}$. μ_Z and σ_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 yields what is known as the *distance profile*[1]. This profile can then be employed to identify similar subsequences, those with a small distance, or those significantly different, with a large distance. Notably, the subsequence with the smallest distance is considered the closest match. If this distance is relatively small, the subsequence may represent a motif, as it appears multiple times within T.

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,

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

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.

[1] Although the concept of a matrix profile can be generalized to AB-Joins, in the following, we only regard the specialized AA-Join. In particular, we only compute distances between subsequences of the same time series T.

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

$$D_i \, (1 \leq i, j \leq n - m + 1)$$

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

where $d_{i,j}(1 \leq i, j \leq n - m + 1)$ denotes once again the z-normalized distance between subsequences $T_{i,m}$ and $T_{j,m}$. These definitions lead us to define the matrix profile, which can be interpreted as the column-wise minima, referred to as aggregates, of the distance matrix. Consequently, MP_i represents the minimal distance between subsequence $T_{i,m}$ and any other subsequence in T. It's worth noting that due to the symmetry of Euclidean distances ($d_{i,j} = d_{j,i}$), the matrix profile also represents the vector of row-wise minima.

However, the distance between any subsequence and itself is 0 ($d_{i,i} = 0$). Additionally, distances between adjacent subsequences are relatively small. These matches are commonly known as "trivial matches" and are often excluded as they are uninteresting for most use cases. The *exclusion zone* for a subsequence is defined as the set of indices resulting in a trivial match. The definition of a trivial match depends on the application domain. Typically, an exclusion zone of $m/2$ is used, meaning that 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 [3], is introduced.

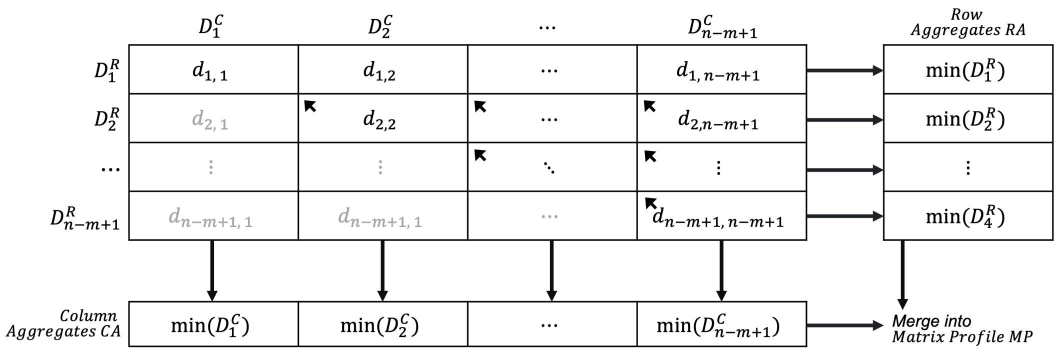

Figure 2.2: Matrix Profile *MP* of a Time series *T* as the column-wise minima of the Distance Matrix and the Matrix Profile Index *MPI* as the vector of the corresponding indices. In this example, $d_{2,j}$ represents a column-wise minimum and is therefore integrated into the Matrix Profile.

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,

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

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

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

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,

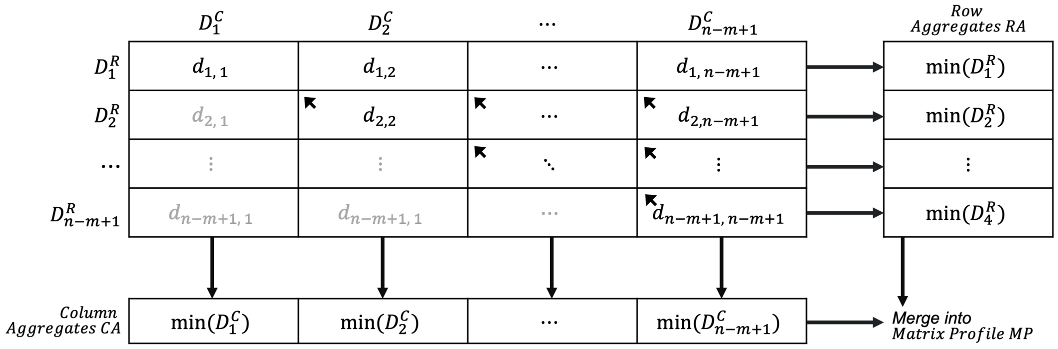

Figure 2.3: Computation performed by the SCAMP algorithm. In particular, only values above (and including) the main diagonal are computed. The diagonal dependency introduced through the updated formulation is visualized through upward-pointing arrows.

i.e., $\overline{QT}_{1,j}$ has to be calculated via the straightforward definition. This thesis focuses on calculating the pearson correlation and does not output Euclidean distance.

SCAMP does not compute the entire matrix but rather only elements above the main diagonal[2]. 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[3] to obtain the resulting matrix profile, as depicted in Figure 2.3.

2.2 Cerebras Wafer Scale Engine

Cerebras Systems' CS-2 represents a revolutionary hardware solution designed to accelerate deep learning tasks. At its core lies the Cerebras second-generation Wafer Scale Engine (WSE-2), a remarkable achievement boasting approximately 1.2 trillion transistors, making it the largest chip in the industry. The CS-2 wafer functions as a multiple instruction, multiple data (MIMD) distributed-memory system, interconnected via a 2D-mesh fabric. Each basic unit on the wafer, termed a tile, comprises 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.

[2] Typically, the main diagonal itself can be excluded as it is contained within the exclusion zone.

[3] By taking the minimum of both row- and column aggregate, i.e., $MP = min(CA, RA)$

Compared to graphical processing accelerators, the WSE-2 features over 100 times the compute cores, 1000 times more high-speed on-chip memory, and over 12,000 times more fabric bandwidth. Occupying approximately 46,000 mm^2, the WSE-2 employs Sparse Linear Algebra Compute (SLAC) cores for efficient computation, boasting over 850,000 of these cores tailored specifically for sparse linear algebra tasks critical in machine learning. This extensive 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.

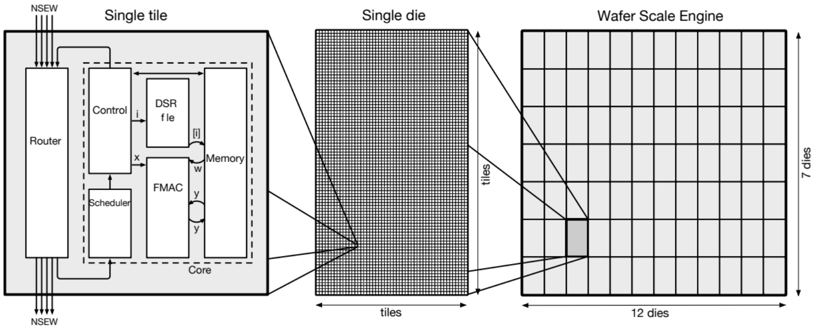

Figure 2.4: An overview of the Wafer Scale Engine (WSE). The WSE (to the right) occupies an entire wafer, and is a 2D array of dies. Each die is itself a grid of tiles (in the middle), which contains a router, a processing element and single-cycle access memory (to the left). In total, the WSE-2 embeds 2.6 trillion transistors in a silicon area of 46,225 mm2. [9]

The WSE-2's absence of off-chip memory distinguishes it from other architectures, as it does not 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 limited by memory bandwidth or latency. The theoretical bandwidth of the WSE-2 is remarkable, reaching 20 petabytes/second, surpassing the A100 GPU's bandwidth of 1.5–2.0 terabytes/second, 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 thesis, 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.

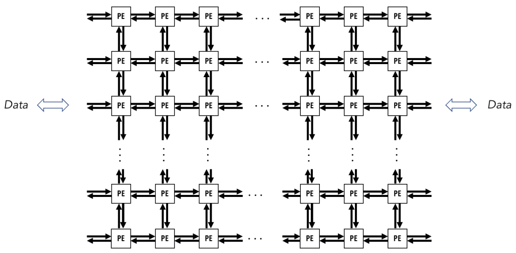

Figure 2.5: The PEs are interconnected cardinally allowing for communication between PEs in packets of size 32 bits.

The device allows programmers to interact with the cores using 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 with 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. A PE measures at 38,000 square microns, with half of its silicon allocated to 48 KB of memory and the remaining half dedicated to logic, comprising of 110,000 cells. Operating at 1.1 GHz, the entire core consumes just 30 milliwatts of peak power.

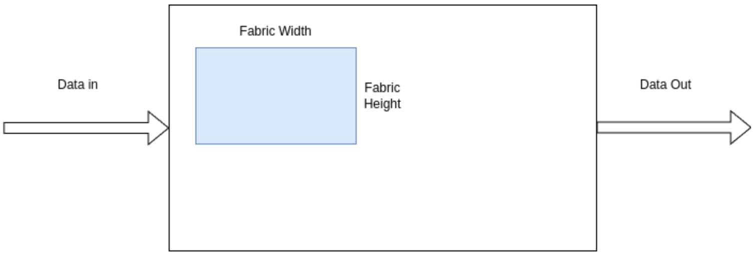

Figure 2.6: A `Resource Rectangle`, allocating a partial rectangle of the Wafer for computation

Figure 2.7 illustrates the memory architecture of a Processing Element (PE). Each individual core is equipped with 48 kilobytes of local SRAM, meticulously designed for optimal density and performance. This is achieved by partitioning the memory into eight single-ported banks, each 32 bits wide. The substantial degree of banking offers more raw memory bandwidth than required for the datapath, ensuring the maintenance of full datapath performance directly from memory. Specifically, this setup enables two full 64-bit reads and one full 64-bit write per cycle. Notably, each core possesses its own independently addressed memory, devoid of any shared memory in the conventional sense.

2.3 Related Works

The Cerebras WSE-2 has found applications across various organizations, including large global corporations, for accelerating machine learning and other tasks. Some noteworthy AI advancements include the adaptation of the GPT model to Cerebras [3]. While the benefits of accelerating machine learning workloads are well-established, there is relatively less research focused on utilizing the WSE for traditional computational tasks. Nevertheless, there have been significant achievements, such as running

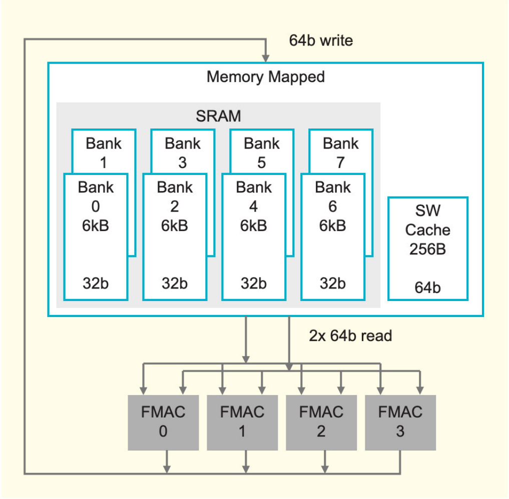

Figure 2.7: The Cerebras PE core memory design [12]

Stencil computations for solving the 3D wave equation using the finite-difference method in seismic imaging applications [5], fast fourier transforms for one, two, and three-dimensional arrays [6], and efficient algorithms for Monte Carlo particle transport [7]. Notably, Stencil computations on the WSE-2 have reached up to 503 TFLOPs, a level of performance typically achievable only by full clusters [8].

This work heavily draws inspiration from the SCAMP paper introduced by Z. Zimmerman et al. [4]. The parallelism approach introduced and optimized by the SCAMP algorithm can be easily mapped onto the distributed MIMD architecture of Cerebras (See Section 3.2). When computing the Matrix Profile aggregates of a row or column, the algorithm lacks data dependencies across rows or columns, allowing for the exploitation of weak scaling, which the Cerebras system is designed for. This algorithmic approach is prevalent in scientific computing and reflects the underlying computational pattern used by many HPC codes.

3 Design

The previous chapter introduced the reader to the concept of matrix profiles and their efficient computation, as well as key development concepts on the Cerebras Wafer Scale Engine. The following chapter presents the top-level architecture as a combination of a `Wrapper`, `Host Application` and an `Accelerated Kernel`. The presented design constitutes a tiled version of the algorithm described in Subsection 3.1.1, allowing the problem size to be decoupled from the concrete kernel implementation. We also explore the various types of tiling solutions that could be incorporated.

3.1 Architecture

The presented design is based on a architecture consisting of 3 distinct components, namely the `Wrapper Application`, `Host Application` and the `Accelerated Kernel`. The `Wrapper Application` computes the tiles required for the time series and passes the tiles to the `Host Application`. The `Host Application` loads the time series and prepares the data to be broadcasted to the *Fabric*. The `Accelerated Kernel` performs the actual computation described in Subsection 3.2.2 on the WSE. A visualization of this architecture can be found in Figure 3.1

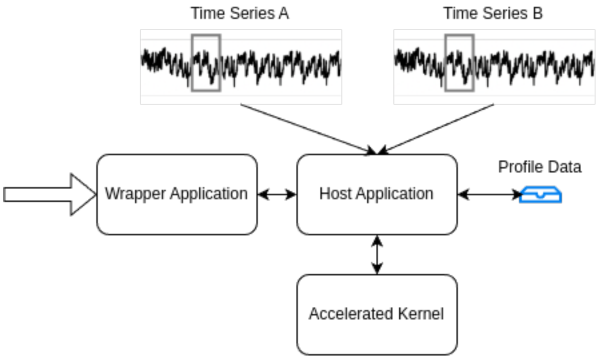

Figure 3.1: Architecture of our application

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 execution and splits the problem space into smaller chuncks which can be executed by the `Resource Rectangle` if the available resource is not sufficient.

2. **Pre-Computation** `Host Application`: It picks up the configuration from the `Wrapper` After the input time series are 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`. It then initializes the Simulator/Device and specifies the code to be pushed to the PEs that are part of the `Resource Rectangle`

3. **Computation** `Accelerated Kernel`: Once the data is transfered, The Host 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 device, The Host can read the row- and column-aggregates from the device.

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 Accelerated Kernel

Cerebras Software Language, or CSL, shares syntax similarities with languages like Go, Swift, and Scala. It is tailored for crafting high-performance programs for Processing Elements (PEs) on the Wafer Scale Engine (WSE). CSL operates as a medium-level language, providing access to both low and high-level programming features. At a low level, CSL exposes the functionalities of the instruction set and some hardware structures controlled by the Instruction Set Architecture (ISA). Conversely, CSL behaves as a high-level language, supporting structured control flow, loops, if-then-else constructs, and abstracting processor registers with automatic register allocation. Additionally, CSL facilitates compile-time execution of code blocks that take compile-time constant objects as input, a potent capability inherited from Zig, the language CSL is based on. Although CSL will be familiar to individuals proficient in C/C++, it incorporates new features atop the basics derived from C.

The 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. The language also provides the flexibility to allocate specific regions of the `Resource Rectangle` for different computational requirements. For our usecase, we have a kernel implementation that is pushed into every PE that is allocated in the `Resource Rectangle`. Figure 3.2 is an extract 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.

3.1.2 Host Application

The `Host Application` provides a path to the binaries produced by the CSL compiler, prepares the data required for execution and distributes the data to the allocated PEs. Once the kernel has finished computation on all the PEs, the host reads the profiles from the fabric and they are then reordered and reduced on the host. The Matrix Profile is then persisted onto a file.

The execution flow for the simulator and device are slightly different. The device compiler outputs an artifact id for the `SdkRuntime` to pick up and this is then executed. The simulator on the other hand picks up the binaries compiled by the `cslc` compiler and executes them on the CPU. The SDK handles the abstractions and configures the execution on the device/simulator.

```
@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,
        });
    }
}
```

Figure 3.2: Setting a `Resource Rectangle` of size `WIDTH` and `HEIGHT` and distributing `tile_kernel.csl` to each PE with params for `Tile Size` and `window`

3.1.3 Wrapper Application

The Wrapper Application is a wrapper of the `Host Application` and allows flexible execution of different tile sizes (t_s), partial matrix profiles and different *Fabric* configurations. This allowed us to test various optimizations and debug errors effectively.

3.2 Kernel Algorithm

The following section examines the Vanilla Kernel of the Matrix Profiling algorithm, followed by an exploration of the tiled version implemented by SCAMP. Finally, it delves into the tiled version that we implemented on the Cerebras WSE-2.

3.2.1 Vanilla Algorithm

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

2. Subsequently, the incremental update formulation, as described in Equation 2.6, is employed to compute successive QT rows (line 7), and Equation 2.3 is used to determine the Pearson correlation (line 8). Following that, the aggregates are updated accordingly (lines 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 1.

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

3.2.2 Tiled Algorithm

SCAMP improves on the vanilla kernel for exploiting the parallelizable nature of the problem by implementing a tiled architecture. As explained in Equation 2.6, the computation of $QT_{i,j}$ requires just the time series at that point, that and the precomputed statistics, enabling every tile to be computed independently. This can be exploited by instantiating multiple processing elements, of which each computes a single Tile of t_s (tile size) independently. 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.

Porting the SCAMP C++ Matrix Profiling Kernel to the Cerebras WSE engine posed significant challenges due to the fundamental differences in architecture between conventional CPUs/GPUs and the specialized wafer-scale integration of the Cerebras

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 $rowAggregates, columnAggregates$;
```

WSE-2. A key obstacle was defining appropriate parameters and efficiently distributing the workload across the processing elements (PEs) inherent to the Cerebras WSE architecture.

The SCAMP algorithm relied on specific compiler optimizations and architectural characteristics of conventional computing platforms achieving great performance, making direct translation to the Cerebras environment complex. Thus, extensive modifications and optimizations were necessary to ensure compatibility and performance efficiency on the novel hardware architecture. Since the latest version of SCAMP made use of multiple libraries that abstracted away optimizations, we focused on the commit hash `8aac4a1f3f64392d17be52576ad87c0521ebf56e` as our reference for our implementation.

To parallelize the Matrix Profile algorithm, the distance matrix calculated using Equation 2.9 is partitioned into square tiles of tile size (t_s). For one-dimensional time series where the point of interest lies only in the upper diagonal tiles, the number of tiles T in SCAMP is calculated using the formula:

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

where:

$$N = \frac{S - m + 1}{t_s}$$

where S is the size of the time series, m is the length of the subsequence/window and t_s is the maximum size of each tile.

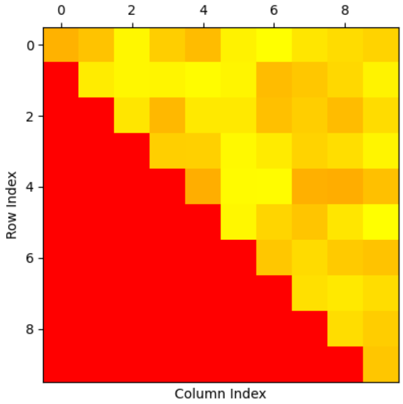

Figure 3.3: Tiling Representation

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 nodes. We finally settled on the following algorithm which is similar to the stripped down version of the SCAMP 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 PE has recieved data, then the kernel computation is triggered from the host. Each PE is only responsible to compute the aggregates for the particular tile since there are no data dependencies across tiles. The aggregates are then reduced to a Matrix Profile on the host device. This was a design decision made given the constraint on the Cerebras system for communication across PEs.

The following kernel described in Algorithm 3 supports t_s upto 835 due to memory limitations of a PE but we experienced higher error rates at that tile size, hence we decided on a maximum t_s of 830. It also supports window sizes of $3 < m < t_s/2$. We go deeper into the performance and limitations of a PE in Chapter 5.

Algorithm 3 is the final version of the Cerebras Tiled Kernel. The core algorithm computes the aggregates for the upper half of the triangle and the other half is computed by flipping the parameters hence computing the whole tile.

The following sections of the algorithm are the primary contributors to the performance.

- Compute initial $\overline{QT}$
- Compute pearson correlation for the current element
- Compute $\overline{QT}$ for the next row.

We also explored other version of tiling on the Cerebras to see if they offer any benefit in the following section.

Algorithm 3 Cerebras Tiled Matrix Profiling 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
    $rowAggregates_{ProcessingElement} \leftarrow (-\inf, -1);$
    $columnAggregates_{ProcessingElement} \leftarrow (-\inf, -1);$
    $QT \leftarrow computeQT(T_a, T_b);$
    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;
            $col \leftarrow row + diag;$
            $correlation \leftarrow QT_{diag} \cdot inv_{row} \cdot inv_{col};$
            $QT_{diag} \leftarrow QT_{diag} + df_{row} \cdot dg_{col} + df_{col} \cdot dg_{row}$
            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 $rowAggregates, columnAggregates;$
```

3.3 Tiling Options

The choice of tiling approaches plays a pivotal role in optimizing parallelization strategies for various computational tasks. Tiling, also known as loop blocking, partitions the computation space into smaller units, facilitating efficient data reuse and minimizing memory access overhead. Several research studies have investigated the impact of different tiling strategies on parallel performance across diverse architectures. [**10**], [**11**]

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 triangles leads to more jobs to be scheduled on 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 Scheduling Approaches

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. We also have a limitation on the maximum `Tile Size` that can be allocated on a single PE of 835. This creates a limitation on the size of time series we can process in one shot. Equation 3.1 provides the number of tiles a given series of size s will be decomposed into. since we have only $745,500$ PEs to work with and each can compute a tile of size 835, we arrive at a maximum time series of size $s = 1008940$ which decomposes into $744,810$ tiles where $t_s = 830$. Although this does not factor in the availability of Wafer Scale clusters with multiple Wafers available for execution, The Cerebras SDK at it's current stage, does not provide a simple way to integrate multiple wafers for our computation. This prompted us to solve the problem of large time series by breaking up the problem size since it is designed to be parallelizable. Real world time series data sets can go in order of billion entries [8], We explore here the different scheduling approaches to compute large scale time series data efficiently.

3.4.1 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* when the number of tiles is less than 745,500.

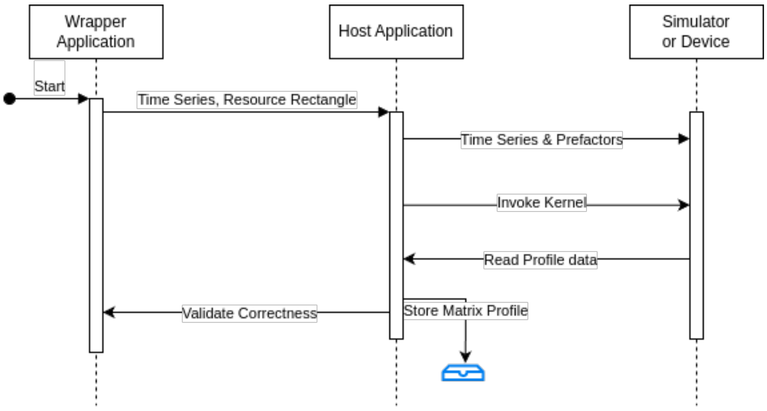

Figure 3.4: One Shot Execution of a small Time Series

The advantage of *One Shot* scheduling is the reduced overhead of PE allocation and fewer memory transfer. In Practice, we found that the One Shot execution of a time series to be more demanding on the device in comparison to *Iterative* execution due to the overhead of a larger `Resource Rectangle` configurations which involves larger memory transfers[1].

3.4.2 Iterative

Since the algorithm allows us to execute tiles out of order, It gives us the flexibility to schedule different tiles on any PE configuration, The following approach splits the execution into smaller chunks which are then scheduled for execution in smaller `Resource Rectangles`

[1] We ran into errors when moving data more than 40 GB through the interface, which ran into RPC timeouts to the device. The fix for this is upcoming in the next SDK release

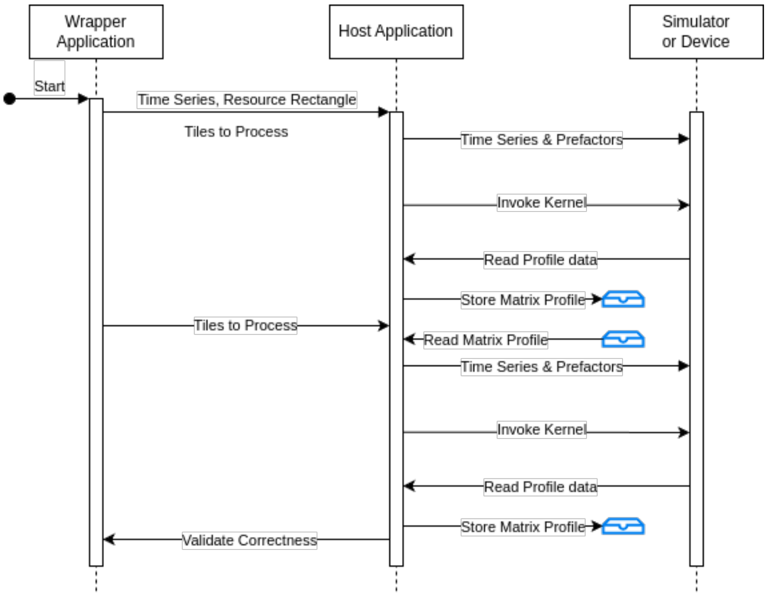

Figure 3.5: Iterative Execution of a large Time Series

When the total number of tiles is not enough to fit into a available `Resource Rectangle`, We chunk the tiles into smaller sections that are iteratively run on the Device/Simulator. This was useful to compute the Matrix Profile of very large data sets. The profile data is updated between iterations. This is possible due to the modularity of the algorithm, allowing us to split the computation into smaller steps.

4 Implementation

The following chapter describes how the design described in Section 3.1 is realized as an Cerebras application. This chapter uses the Cerebras SDK[1] conventions and syntax as the underlying work targets the Cerebras WSE-2 device. Before potraying concrete implementation choices, we briefly introduce the reader to the Cerebras SDK and the WSE programming paradigm. We then go through our execution model and take the reader through each component of the architecture, elaboriating on the steps involved in the Matrix Profile computation.

[1] https://sdk.cerebras.net/

4.1 WSE Programming Environment

To develop programs 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 functions on the device.

4.1.1 Build Process

The build process is resembles a standard compilation for the device code which is compiled using the `cslc` compiler toolchain[2]. It outputs binaries which are then picked up by the Python Host Code to be transfered to the device/simulator 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.

4.1.2 Host-Device Interaction

As described in Section 3.1, the design is split across a `Host Application`, `Accelerated Kernel` and a `Wrapper Application`. The `Host Code` is written in Python and uses the Cerebras SDK. The SDK provides a host runtime known as the *SdkRuntime*[3], and associated utility[4] and debug functionality[5] to load programs, launch functions, and transfer data to and from the device.

The interaction between the host and device consists of four distinct stages:

1. The Host prepares the data for the given time series, sets up the runtime, starts the simulator/connects to the device, setups up the required amount of computing resource (`Resource Rectangle`) and transfers data to the allocated PEs on the simulator/device.

2. It then triggers the kernel function on the PEs.

3. Once all the PEs have finished executing the required computation, The PEs are ready to accept commands from the host.

4. Finally, the Host copies back the result into host memory.

[2] https://sdk.cerebras.net/csl/csl-compiler

[3] https://sdk.cerebras.net/api-docs/sdkruntime-api

[4] https://sdk.cerebras.net/api-docs/sdkruntime-api#sdk-utils

[5] https://sdk.cerebras.net/api-docs/sdkruntime-api#debug-util

4.2 Host Application

In this section, we delve into the implementation specifics of the `Host Application` outlined in Section 3.1. As previously mentioned, the Host starts by loading the input time series from the files specified via command line arguments $file_a$ and $file_b$. It provides us the povision to process only specific tiles for computation or the entire time series. It performs the necessary pre-computation for the two time series and then initiates a `SdkRuntime`[6] from the Cerebras library, which connects to and initializes the device or starts the Cerebras fabric simulator. The `SdkRuntime` provides access to symbols exposed by the kernel program which are used to transfer data to the PEs. Although the SDK allows for asynchronous execution of commands, we choose to use synchronous operations.

The Host Code then prepares the data required for computation and transfers them to the device. Figure 4.2 shows the function call to transfer data to the device. The `memcpy_h2d` accpets a N-dimensional tensor and transfers it to a provided symbol to every PE in the `Resource Rectangle`. Each PE receives `MAX_TILE_SIZE - WINDOW + 1` elements of the prefactors and `MAX_TILE_SIZE + WINDOW` elements of both the time series. Large time series require large transfers to the device, and one approach is to split the transfer through multiple `memcpy_h2d` calls, targeting different sections of the wafer. One caveat here is that the arrays declared in the kernel are statically sized and therefore require padding from the host with precise sizes.

The `SdkRuntime` needs an artifact to run on the device or compiled binaries to run the simulator. Figure 4.1 shows the steps involved in invoking the `SdkRuntime`. Executing code on the hardware and simulator follows different execution paths and has varying compilation times[7].

Once the data is transferred, the kernel function $compute()$ is invoked as shown in Figure 4.3. Once the kernel function is complete and the device is ready to receive more commands, the tile aggregates are then transferred from the device as shown in Figure 4.4, following the same convention as the call to transfer memory to the device.

The received row- and column-aggregates are then merged into a unified matrix profile which is then persisted. The implementation allows executing tiles out of order or only specific tiles since the tiling and execution logic are decoupled.

[6] https://sdk.cerebras.net/api-docs/sdkruntime-api

[7] https://sdk.cerebras.net/appliance-mode.html

```
if on_device:
    # Read the artifact_id from the JSON file
    with open("artifact_id.json", "r", encoding="utf8") as f:
        data = json.load(f)
        artifact_id = data["artifact_id"]
    runner = SdkRuntime(artifact_id, simulator=False)
    runner.start()
else:
    runner = SdkRuntime(args.name, cmaddr=args.cmaddr)

    # Load and run the program
    runner.load()
    runner.run()
```

Figure 4.1: Initializing `SdkRuntime` on the device and for the simulator. Notice the missing `load()` call to the runner on the device. This is due to a dynamic compilation routine that outputs an artifact_id for the runtime to load the runtime from.

```
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 4.2: 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

4.3 Kernel Implementation

As elaborated in Section 3.2, 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. The kernel in addition, gets the following args for computation.

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

Figure 4.3: The SDK allows us to launch functions on the device synchronously or asynchronous

```
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 4.4: Memory Copy from the device, Note the explicit PEs width and height declaration

- n_x, n_y: The boundaries of the tile.
- $exclusion\_lower_u, exclusion\_upper_u, exclusion\_lower_b, exclusion\_upper_b$: The exclusion boundaries provide a exclusion boundaries for the two triangles that are part of the square tile.
- $full\_tile$: The specifies the kernel if it has to compute only the upper triangle or both upper and the bottom triangle of the square tile.

Figure 4.3 shows the entry point to the `Accelerated Kernel`, it decides if the PE is calculating only the upper triangle of the tile or the entire tile and calculates $\overline{QT}$ at that point and passes over the variables to the kernel function described in Figure 4.6. Notice the `@activate(EXIT)` at the end of the `compute` function, this signals the `SDKRuntime` that the PE is ready to accept calls from the device.

```
fn compute() void {
    if (full_tile == 0) {
        compute_cov(&T_A, &T_B);
        kernel(&NORM_A, &NORM_B, &DG_A, &DG_B, &DF_A, &DF_B, &MP_A, &MPI_A,
               &MP_B, &MPI_B, n_x, n_y, exclusion_lower_u, exclusion_upper_u);
    } else {
        compute_cov(&T_A, &T_B);
        kernel(&NORM_A, &NORM_B, &DG_A, &DG_B, &DF_A, &DF_B, &MP_A, &MPI_A,
               &MP_B, &MPI_B, n_x, n_y, exclusion_lower_u, exclusion_upper_u);
        // SWAP inputs for lower triangle
        compute_cov(&T_B, &T_A);
        kernel(&NORM_B, &NORM_A, &DG_B, &DG_A, &DF_B, &DF_A, &MP_B, &MPI_B,
               &MP_A, &MPI_A, n_y, n_x, exclusion_lower_b, exclusion_upper_b);
    }
    @activate(EXIT);
}
```

Figure 4.5: Entry point to the kernel, this controls the kernel execution on the upper triangle or complete tile

```
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, n_x: u16, n_y: u16,
          exclusion_lower: u16, exclusion_upper: u16) void {
    // Take into account exclusion zones while calculating
    var row_iters: u16 = math_lib.min(n_x - exclusion_lower, n_y);
    for (@range(u16, row_iters)) | row | {
        var diag_max: u16 = math_lib.min(n_x - exclusion_upper + 1, n_x - row);
        for (@range(u16, exclusion_lower, diag_max, 1)) | diag | {
            // Calculate Pearson correlation
            var corr: f32 = cov[diag] * norm_a[col] * norm_b[row];
            var coeff: f32 = df_a[col] * dg_b[row] +
                             dg_a[col] * df_b[row];
            // Update QT of the next diag.
            cov[diag] = cov[diag] + coeff;
            // Update column and row aggregates
            update_profile(corr, P_a, P_i_a, row, col);
            update_profile(corr, P_b, P_i_b, col, row);
        }
    }
}
```

Figure 4.6: *Matrix Profile* kernel. It computes the row- and column-aggregates for the upper triangle of the tile.

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 chapter, we systematically investigate the influence of four pivotal control variables: `Time Series`, `Tile Size`, `Window Size`, and `Resource Rectangle`, on the performance 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.

We also 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 Experimentation Model

In this chapter, we delve into a series of experiments aimed at elucidating the impact of various control variables on the performance of Matrix Profiling algorithms executed on the Cerebras Wafer Scale Engine (WSE), The implementation targeted the `Cerebras SDK 1.0.0`. These experiments are guided by the following control variables:

- **Time Series (s)**: The length of the time series data being analyzed.
- **Tile Size (t_s)**: The size of the tile allocated to a single PE, which directly influences the parallelism and performance.
- **Window Size (m)**: The size of the sliding window used for local pattern discovery within the time series data. It is fixed to 6 for all the experiments unless specified.
- **Resource Rectangle (r_r)**: The configuration of resources allocated within the Cerebras WSE for executing the Matrix Profiling algorithm. It consists of 2 variables, width w, height h.

These control variables serve as the foundation for our experimental design, allowing us to systematically explore their implications on the overall execution and performance of the Matrix Profiling algorithm on the Cerebras WSE-2. The time series used for the experiments are random walk time series generated uniquely for each test case.

The Matrix Profiling algorithm exhibits a time complexity of $O(s^2)$, where s represents the length of the time series data. The time complexity of the tiled kernel is $O(t_s^2)$ since it is localized to the tile. We will be looking at the performance of a single tile and the entire time series through this chapter.

Our design implicitly has 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-2. 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 empirical results from the simulator and hardware. All the following experiments were evaluated on a `Intel(R) Xeon(R) Platinum 8260 @ 2.40GHz` machine running `Red Hat Enterprise Linux 8` provided by the EPIF[1], part of the University of Edinburgh, The Machine also contains WSE-2 Cluster[2] on which the hardware experiments were run.

[1] https://epcced.github.io/eidf-docs/services/ultra2/run/

[2] https://epcced.github.io/eidf-docs/services/cs2/run/

5.2 Runtime Model

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

To determine the runtime of the kernel, the SDK exposes hardware unix timestamps of which PE which can be collected and queried at different parts of the kernel. While some degree of thermal throttling will occur, the WSE-2 implements throttling by injecting "nop" commands rather than adjusting the clock speed itself, such that any thermal "nop" cycles are included when measuring timestamps. This method of measuring kernel runtime performance by recording unix timestamp is used throughout this paper whenever runtime data is reported and is the standard method for recording performance data on Cerebras machines [9].

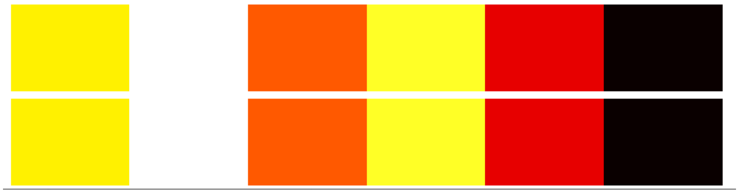

Figure 5.1: *Start* and *End Timestamps* of PEs for a 1 × 6 `Resource Rectangle`

Start Time	End Time
215471598885009	215471606612597
210033113389942	210033121118543
219951141230794	219951157273972
214324783517930	214324792236640
223219653123097	223219661842174
229732080439964	229732083089828

Table 5.1: Device Timestamps used in the Heatmap in Figure 5.1

Figure 5.1 is a heatmap representing the start and end timestamps on the fabric created from data in table 5.1, the lighter colors represent PEs that start first, the darker colors represent the PEs that started at a later point in time. Given the variation in timestamps across PEs which could depend on the hardware implementation of calls to the device. Our measurements calculate the min, mean and max differences between the start and end timestamps and uses the best measurement for the experiment since the tiling of the *Distance Matrix* leads to tiles of various sizes which impacts the execution time.

5.2.1 Low-Level (Kernel) Model

The algorithm to compute the Matrix profile as defined in Algorithm 3, is an $O(t_s^2)$ algorithm and computes the upper triangle of a given tile also called the half tile. The execution time is directly proportional to t_s. Our experiment look at the hardware and simulator runtimes on executions on a single PE solving a problem of size s = t_s. Figure 5.2 is a plot of execution times vs increasing tile sizes. We hence arrived at a Theoretical model for estimating runtime for a given tile size.

$$T(t_s) = 4.38e^{-5} * t_s^2 + 1 \tag{5.1}$$

5.2.2 Complete Tile Model

The complete tiles experiments were conducted by executing a complete tile in a single PE where $s > t_s$. The results from the hardware and simulator are depicted in Figure 5.2. Thus, we arive at the theoretical model for the runtime of a complete tile.

$$T(t_s) = 1.001e^{-4} * t_s^2 \tag{5.2}$$

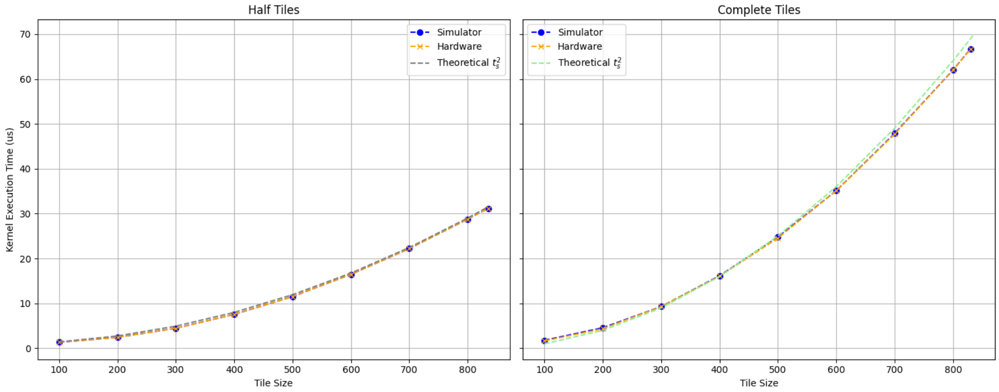

Figure 5.2: Comparison of *Kernel Execution Time* of increasing tile sizes t_s on a single PE computing a half and complete tile.

5.3 Memory Consumption Model

Matrix profiling is an memory intensive algorithm since it involves multiple variables and coefficients of data for 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 to consume on the PE. The memory consumption M for a single Processing Element (PE) in a Matrix Profiling computation of two time series a and b can be modeled using the following equation:

$$M = F(inputs_a, inputs_b, outputs_a, outputs_b)$$

$$M = F(T_a, inf_a, df_a, dg_a, T_b, inf_b, df_b, dg_b, args, MP_a, MPI_a, MP_b, MPI_b)$$

Upon further reduction, the equation becomes:

$$M = \text{sizeof}(2 \times (t_s + m) + 10 \times (t_s - m + 1) + 7)$$

Substituting the values for t_s (tile size), m (window size), size of each variable being 32 bits and simplifying, we get:

$$M = \frac{3 \times t_s - 2 \times m + 17}{256} \tag{5.3}$$

Here, t_s represents the tile size and m represents the window size.

The plots demonstrate the relationship between memory consumption (M) and tile size (t_s). We set the window size m to 6. Additionally, They include theoretical and practical limits for memory consumption. Figure 5.3 shows a plotted model using the Equation 5.3. We reaching a practical limit of 39.01 KB on a maximum tile size t_s of 835. We are able to allocate 81% of memory available in the PE. This does not account for the allocations required by the PE for execution since the global memory is also used for the program instructions and other runtime allocations. For the following experiments, we set t_s to 830 since we encountered higher errors when t_s = 835 which we could not explain.

5.4 Memory Bandwidth Experiments

The following section contains experiments that try to evaluate the memory bandwidth of the Cerebras WSE-2 by executing Matrix Profiles of time series of increasing sizes. This enables us to model the memory latency and transfer times of the Cerebras WSE-2.

5.4.1 Memory Bandwidth

Our first experiment looks at the memory bandwidth capabilities of the Cerebras WSE-2 and the impact of the size s of the time series. We fixed the tile size t_s to 830 and gradually increased the time series size s to transfer increasing quantity of data to the device. With a tile size of t_s = 830, Each PE in this experiment receives 39.01 KB, and

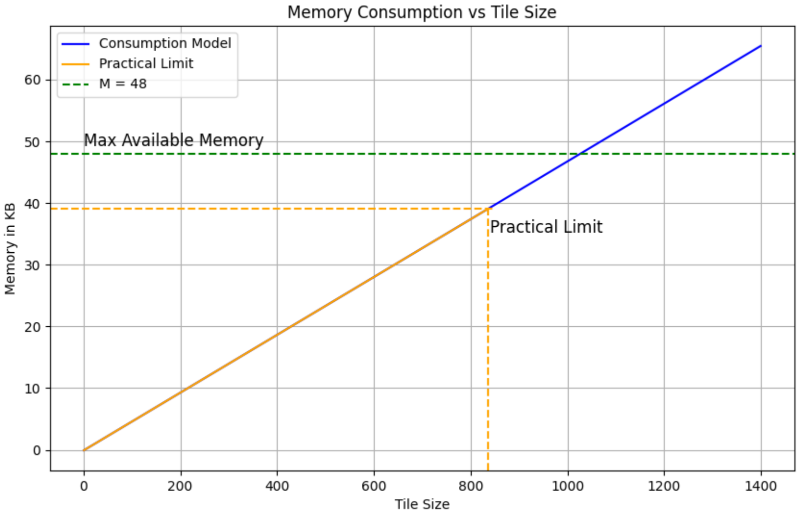

Figure 5.3: Memory Usage (M) of a single PE plotted against Tile Size t_s. The theoretical limit to t_s is 1010 but what is observed is a maximum limit of 835 per PE. In practice, we were able to use 39.01 KB of memory of the 48KB available in a PE.

the total transfers range from 622 to 15562.37 KB to the device and 207.4 to 5187.4 KB from the device. This enables us to look at concrete differences in transfer times of large time series.

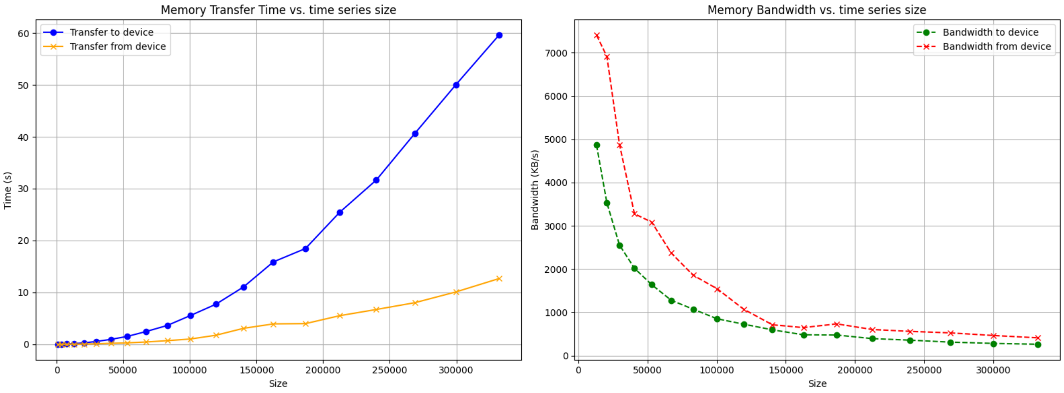

Figure 5.4: *Memory Transfer Time* in seconds and *Bandwidth* in KB/s of time series of increasing sizes.

In Figure 5.4, we notice a sharp decline in memory bandwidth of 27% and this can be explained by the exponential increase in number of PEs allocated and the complexity of the `Resource Rectangle`. The WSE-2 contains 66 memory channels spaced evenly at almost 16 intervals at the two edges of the wafer. This allows data to be streamed into the PEs through multiple channels across the wafer[3]. This experiment allocates large `Resource Rectangle` configurations of sizes (70×949), (101×806), etc. which uses almost the available memory channels on the wafer which could be a reason behind the decline in memory bandwidth.

We can then consider the memory transfer times M_t and M_f for a given time series of size s. We arrived at the following equations for transfer times to and from the device. This does not take into consideration the `Resource Rectangle` configurations since it also impacts the memory transfer times but as seen in Subsection 5.4.2, the impact is negligible.

$$M_t(s) = 5.354e^{-10} \times s^2 + 4.843e^{-06} \times s + -1.207e^{-01} \tag{5.4}$$

$$M_f(s) = 1.007e^{-10} \times s^2 + 4.400e^{-06} \times s + -1.014e^{-01} \tag{5.5}$$

5.4.2 Impact of spacial location of PEs on the Wafer

We further explored the implication of different `Resource Rectangle` on the latency of data transfer to and from the device. This is essential since, the Cerebras WSE-2 does not have a classic hierarchial memory structure with multiple layers of cache. Each PE has access only to its global memory of 48KB and the data is transmitted through a network connection to the fabric. We choose to execute different tile sizes on different configurations of `Resource Rectangle` to observe the affect on memory transfer times. We represent the `Resource Rectangle` configurations using the two dimensions w (Width) and h (Height) to study the impact on latency of data transfer. The experiment was conducted on the device since the simulator does not provide accurate data transfer results.

To study the impact of `Resource Rectangle` configurations on memory transfer, Our experiment consisted of s = 10000, t_s = 500, 600, 700 and m = 6 and various r_r configurations. These sizes allow us to evaulate transfer rates of large volumes of data to and from the device and also create large enough resource rectangles that have a noticable impact on the memory transfer times. We fixed the width to 1 and expanded along the height, e.g. $1 \times 66, 1 \times 105, 1 \times 410$ and fixed the height to 1 and expanded along the width, e.g. $66 \times 1, 105 \times 1, 410 \times 1$ to deleve into the impact of different

[3] https://sdk.cerebras.net/tensor-streaming

dimensions of the `Resource Rectangle`. As seen in Figure 2.6, the data to the device enters the fabric from the left end of the Wafer and exits from the right end. This has an influence on the memory transfer times to and from the PEs in the `Resource Rectangle` configurdation. This experiment transmits 468.62 KB of data to the fabric from the west edge.

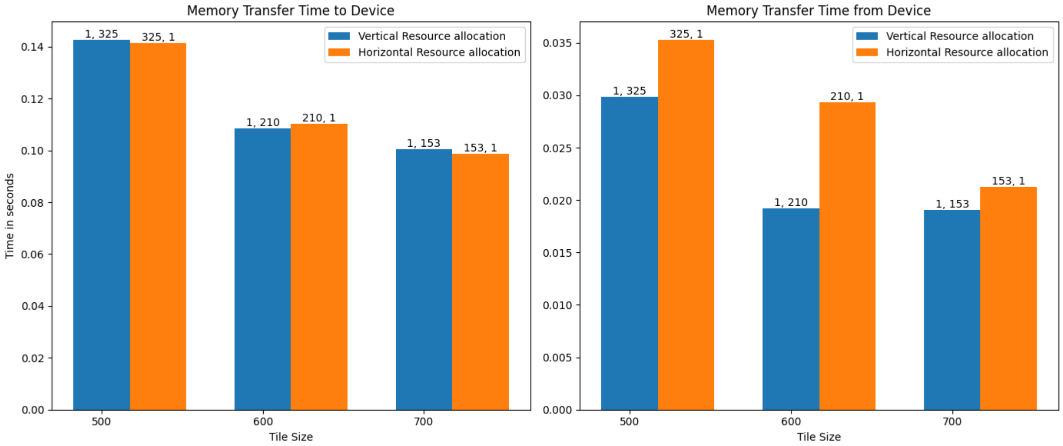

Figure 5.5: *Memory Transfer time* on different resource rectangle configurations allocated at an offset of 4, 4 on the wafer.

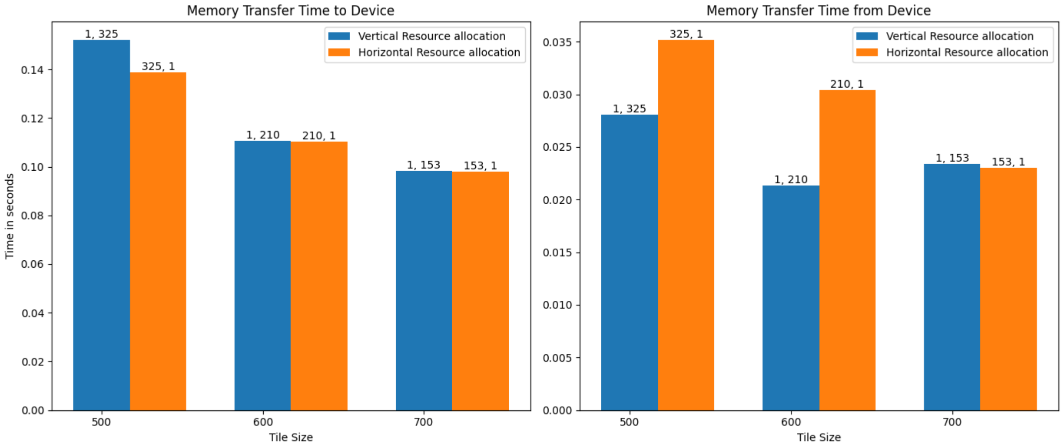

Figure 5.6: *Memory Transfer time* on different resource rectangle configurations allocated at an offset of 300, 300 on the wafer.

Figure 5.5 represents the transfer time to and from the device for a `r_r` allocated at an offset of 4, 4 on the wafer. The above results are from *r_r* allocations at the top right corner of the fabric. This leads to almost equal transfer times to the device in different allocation patterns. The transfer times from the device show a pattern of greater (4%) transfer times for horizontal resource allocations. This could be due to the overhead of data travelling through the same fabric since the PEs are allocated in the same row.

Figure 5.6 represents the transfer time to and from the device for a `r_r` allocated at an offset of 300, 300 the wafer. We see a trend of higher transfer times for all shapes even though the variation is about 5%. There is also a increase in transfer time from the device for horizontal resource allocations. The above figures show very slight variations (5%) in transfer times and does not show any concrete impact of different `Resource Rectangle` configurations since we are dealing with millisecond differences.

5.5 Time Series Execution Model

Now that we have a theoretical model for execution times for half and complete tiles for a given t_s and memory transfer times for time series of a given size s, we can derive a model to predict the runtime of any given time series of size s, tiled into tiles of size t_s. Figure 5.7 plots the total execution times of time series of increasing sizes. We notice a lower execution and memory transfer times on smaller time series and a gradual increase in memory transfer times to the device due to the increase in time series sizes and `Resource Rectangle` complexity.

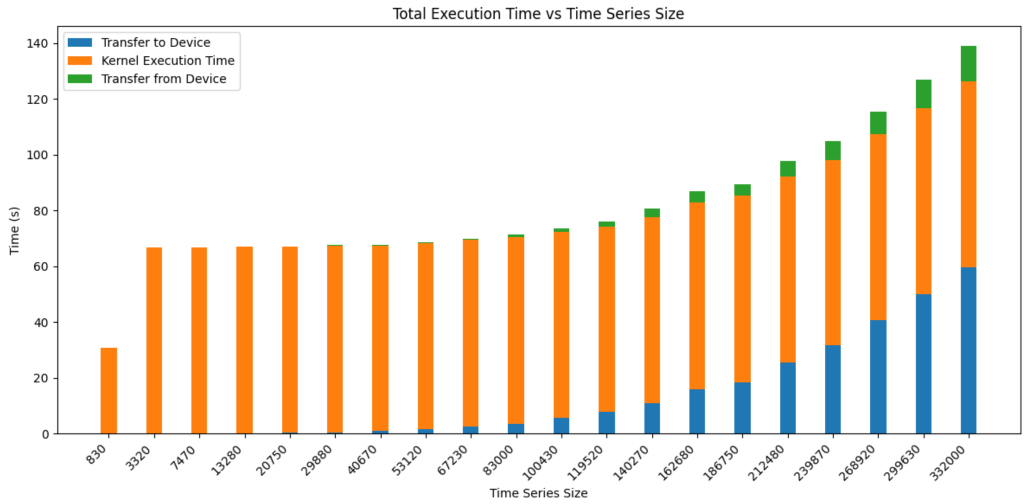

Figure 5.7: *Total Execution Times* for series of increasing sizes.

From Figure 5.7, we can derive that when $t_s > s/2$, we notice a higher performance, this can be explained by the tiling nature and the presence of more half tiles in the computation. With this information, We arrive at the below formulation for the kernel execution time of a series of size s and tile size t_s in seconds, considering Equation 5.2 and 5.1.

$$T_{kernel}(s, t_s) = \begin{cases} 4.38e^{-5} * t_s^2 + 1, & \text{if } t_s > s \times 0.7 \\ 1.001e^{-4} \times t_s^2, & \text{if } t_s \leq s/2 \end{cases} \tag{5.6}$$

We can simplify this for normal time series of larger sizes and arrive at a final equation.

$$T_{kernel}(t_s) = 1.001e^{-4} \times t_s^2 \tag{5.7}$$

Thus combining Equation 5.4, 5.5 and 5.6, we arrive at a model for the total execution time (T_{total}) for any given time series of size s and tile size t_s.

$$T_{total}(s, t_s) = M_t(s) + T(t_s) + M_f(s)$$

Substituting the equations for $M_{to\_device}(s)$, $T_{kernel}(s, t_s)$, and $M_{from\_device}(s)$, we get

$$T_{\text{total}} = M_t(s) + T(s, t_s) + M_f(s)$$

Substituting Equation 5.4 and 5.5, we arrive at Equation 5.8

$$\begin{aligned} T_{\text{total}} &= M_t(s) + T(s, t_s) + M_f(s) \\ &= (\underbrace{5.354e^{-10} \times s^2 + 4.843e^{-06} \times s + -1.207e^{-01}}_{\text{Memory Transfer Time}} \\ &\quad + \underbrace{1.001e^{-4} \times t_s^2}_{\text{Time for Compute}} \\ &\quad + \underbrace{1.007e^{-10} \times s^2 + 4.400e^{-06} \times s + -1.014e^{-01}}_{\text{Memory Fetch Time}}) \end{aligned} \tag{5.8}$$

This is the model to find out the execution time of a time series of any given size on the Cerebras WSE-2. Note that this does not take into account the resource availability on the wafer or the presence of clusters containing multiple such wafers.

Figure 5.8 presents the theoretical execution model vs total execution times of time series of increasing sizes. We see variations in the execution times which could be explained by the influence of different `Resource Rectangle` configurations on the transfer times.

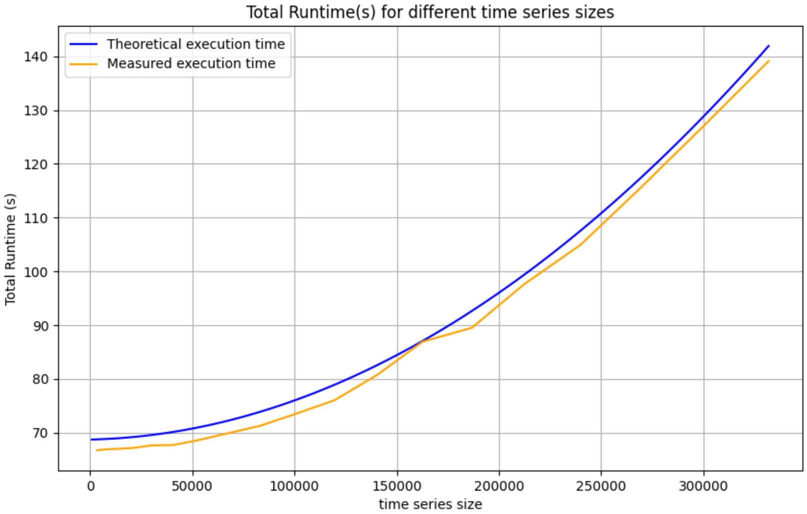

Figure 5.8: *Total Execution Time*: Aggregate execution times comprising both memory transfer and kernel execution times for time series of increasing sizes.

5.6 Weak Scaling Experiment

Since the WSE-2 primarily focuses on weak scaling, we examine the potential implications of weak scaling on the device using various configurations and analyze the execution times. We initiate our weak scaling experiments by gradually increasing the number of resources allocated for the time series. We use the following equation:

$$s = i^2 \times t_s$$

Where t_s = 830 and i ranges from 1 to 20. This provides us with an exponential increase in the sizes of the `Resource Rectangle` and a constant execution time for all PEs since $T_{kernel}(s) \propto t_s$.

The execution times represent the maximum execution time of all the PEs in the `Resource Rectangle` since there can be multiple PEs with lower workloads given the smaller tile dimensions. Weak scaling is evident when comparing the performance metrics across different problem sizes. Despite the increase in problem size, the performance metrics remains the same with increasing computational resources, maintaining a consistent workload per processing element.

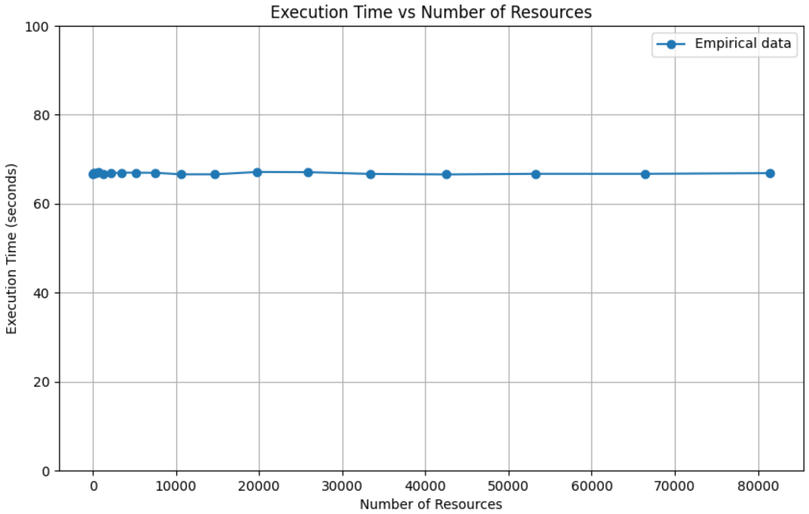

Figure 5.9: *Weak Scaling* effects of the Cerebras WSE-2 on large time series.

5.7 Strong Scaling Experiment

Although the Cerebras WSE-2 is not built with strong scaling in mind, we planned our strong scaling experiments in the following manner, We tested the *Tiled Kernel* with different number of PEs while keeping the problem size constant. We keep $s = 10000$ and decreasing t_s. t_s is continuously decreased from 830 to 100 which also leads to a increase in the amount of resource required for computation. In theory, We expect a decrease in the computation time given the increase in resources. The results of the experiment are depicted in Figure 5.10 with the execution time on logscale.

We notice a lower execution times on the smaller time series due to the decompositional behaviour of the *Distance Matrix*. Lower sizes of time series decompose into more half tiles which leads to lower mean execution times on the device. On the other hand, larger time series have more complete tiles which take around 66 seconds to compute, hence resulting in a almost straight line.

5.8 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

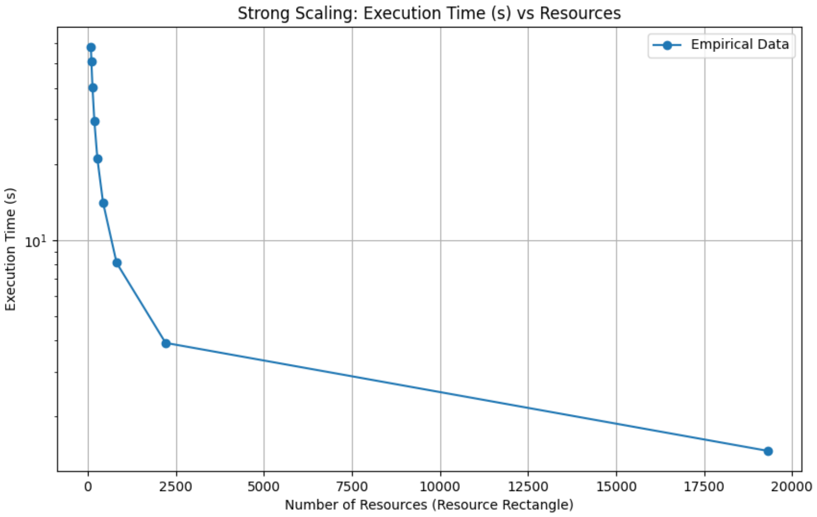

Figure 5.10: *Strong Scaling* - Kernel Execution time with increasing `Resource Rectangle` sizes

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. In the experiment, we created and executed time series using different random distributions of size s = 10000 and m = 400 on the device. These results are then compared to the double precision version of SCAMP. Although this does not reflect on the error rates of real life dataset, we wanted to look at anomalies in error rates across different distributions.

The errors observed in Figure 5.11 on different error distributions reveal varying challenges and successes in accurately capturing the characteristics of each distribution. These error rates are crucial in assessing the accuracy and reliability of the Matrixc Profile. For instance, when considering the Uniform distribution, which exhibits a constant probability density function within a defined range, the error rates are relatively low across the board. This indicates that the Matrix Profile closely align with the actual values for this distribution, likely because the uniform nature of the data allows for more straightforward motif discovery.

On the other hand, distributions such as Triangular, Gamma, and Exponential, which possess more complex shapes and variability in their probability density functions,

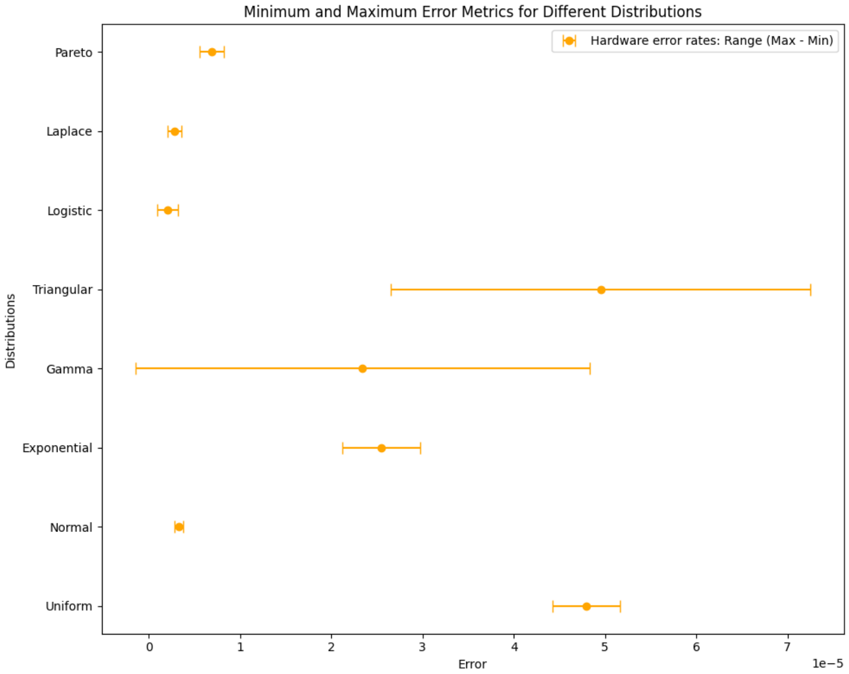

Figure 5.11: Error rate on different probabilistic distributions

exhibit higher error rates. This suggests that the algorithm may struggle to accurately capture the nuances and variability present in these distributions, leading to larger discrepancies between predicted and actual values.

5.9 Comparison to CPU & GPU Baseline

Now that experimental results from both the simulator and hardware are available, along with a theoretical model for estimating total execution time for any given time series size s and tile size t_s, a performance comparison between the Cerebras WSE-2 and traditional GPUs can be conducted. Table 5.2 presents an analysis of SCAMP performance on random walk datasets of increasing lengths [**4**], alongside extrapolated figures for the Cerebras WSE-2.

Notably, due to limitations in the number of tiles executable in a single iteration and the cost of memory transfers to and from the device, significant disparities in execution times are observed.

Table 5.2: Matrix Profiling Runtime on SCAMP on GPUs and our implementation on the Cerebras WSE-2 [**4**]

Implementation	SCAMP-GPU		Cerebras
Architecture	V100	V100	WSE-2
Precision	DP	SP	SP
2^{18}	3.04s	0.34s **(8.9x)**	1.9m **(3650.00x)**
2^{19}	11.4s	1.24s **(9.2x)**	4.14m **(2078.95x)**
2^{20}	44.1s	4.81s **(9.2x)**	12.96m **(1663.27x)**
2^{21}	174s	19.0s **(9.2x)**	48.09m **(1557.24x)**
2^{22}	629s	69.2s **(9.1x)**	188.29m **(1696.09x)**
2^{23}	2514s	277s **(9.1x)**	748.46m **(1686.30x)**

The execution times on the Cerebras WSE-2 are exponentially larger compared to SCAMP-GPU implementations. This is primarily due to the inherent latency introduced by memory transfers to and from the wafer, which significantly impacts the overall execution time.

It's worth noting that for time series larger than 2^{22}, the execution times become extremely long, with a time series of size 8M taking approximately 9 hours to execute without considering the overhead of the SLURM scheduler when running them on the EPIF. Hence, the derived execution times for larger datasets on the Cerebras WSE-2 are based on the theoretical model presented in Section 5.5.

5.10 Execution on Real World Datasets

The real world dataset used in our evaluation of the Cerebras WSE-2 is StarLightCurves[4], consisting of a time series depicting the brightness of a celestial object over time. The examination of light curves in astronomy is integral to understanding source variability. Comprising 1 million entries, this dataset is sourced from the UCR Time Series Classification Archive [**UCRArchive2018**]. Its distance matrix decomposes into 2,692,360 tiles, where $t_s = 830$ and $m = 400$. As this surpasses the 745,500 user-programmable PEs available on a single wafer, we segmented the problem and executed the dataset iteratively using our approach as described in Section 3.4.2.

We were unable to allocate the entire wafer for computation due to bandwidth limitations on Remote Procedure Calls (RPC) to the device for data transfers exceeding 2GB. Consequently, we allocated a maximum `Resource Rectangle` of 457×696 on the device, necessitating 8 iterations to compute the entire matrix profile. This limitation stems from the restricted memory availability of the PEs. With a maximum $t_s = 830$, larger time series require further segmentation, exacerbating overall time consumption due to average roundtrip execution time. An average execution time of $45s$ per PE was observed, lower than that for a tile size of $t_s = 830$, yet the rationale behind this discrepancy remains elusive. Figure 5.12 illustrates the total execution time for each iteration.

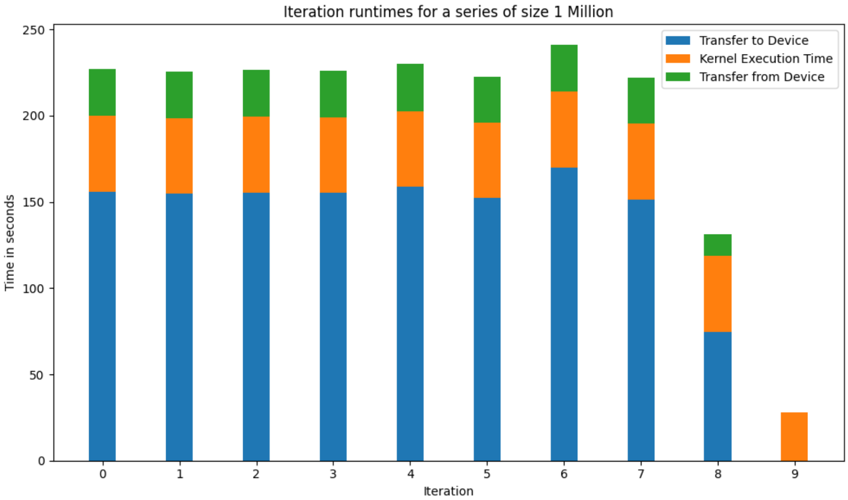

Figure 5.12: *Total Execution Time* of different iterations on a time series of size 1M

We notice varying memory transfer times for the same allocated `Resource Rectangle`

[4] https://www.cs.ucr.edu/%7Eeamonn/time_series_data_2018/

but see a similar execution time over all iterations. We achieved a absolute mean error rate of 1.865×10^{-04} when compared to the double precision profile computed by SCAMP. Table 5.3 compares the maximum absolute error of the Pearson Correlation for large datasets on SCAMP with Cerebras WSE-2. We were able to obtain comparable errors on the device.

Table 5.3: Maximum absolute error (Pearson Correlation) for various datasets/algorithms, compared to SCAMP double precision results, taken from [**4**]. StarLightCurves results were performed on the Cerebras WSE-2

Size (m)	SCAMP SP	STOMP SP	Cerebras WSE-2
Whitefly EPG (2.5M)	3.75×10^{-2}	1.89×10^{1}	N/A
ECG (8.4M)	3.14×10^{-4}	2.07×10^{-3}	N/A
Earthquake (1.7M)	6.35×10^{-1}	3.17×10^{3}	N/A
Power Demand (10M)	4.85×10^{-2}	2.22×10^{-1}	N/A
Chicken (9M)	4.92×10^{-2}	2.27×10^{1}	N/A
StarLightCurves (1M)	3.34×10^{-4}	2.17×10^{-2}	1.865×10^{-4}

The Cerebras WSE-2 performs well on the StarLightCurves dataset since the values were in the order of 10^{-10}. This could be represented precisely by single precision floats.

6 Experiences

Given the relatively nascent status of the Cerebras WSE-2 accelerator and its limited adoption, this chapter aims to succinctly outline our experiences concerning the available tools, workflow, and programming model, with the anticipation that future endeavors may benefit from these insights. We organize this chapter into four sections, beginning with an exploration of the Cerebras WSE-2 and proceeding to discuss the workflow involved in developing for the device.

6.1 Cerebras WSE-2

The Cerebras WSE-2 is still in its infancy regarding development. It offers a robust programming language and an SDK[1] for device development, along with comprehensive documentation to support beginners in their work with the device. The Multiple Instruction, Multiple Data (MIMD) programming paradigm employed differs from conventional paradigms in High-Performance Computing (HPC). Our endeavor to implement Matrix Profiling on the Cerebras represents a pioneering effort and is ripe for refinement, although we have yet to explore the full extent of the device's capabilities.

6.2 Workflow

The workflow for the Cerebras WSE-2 resembles that of working on an embedded device. The compiler[2] produces binaries that must be transferred to the Simulator/Device. However, the SDK is still in its nascent stage and lacks certain utilities, such as facilitating large data transfers to the wafer. Nonetheless, it includes multiple examples, such as the `GEMV` implementation[3], which serve as valuable points of reference. Subsequently, we developed a `Wrapper Application` enabling flexible execution of time series, result verification, and data collection from the device.

[1] https://sdk.cerebras.net/

[2] https://sdk.cerebras.net/csl/csl-compiler

[3] https://sdk.cerebras.net/csl/code-examples/benchmark-gemv-checkerboard

6.2.1 Transitioning from Simulator to Device

The Cerebras SDK provides clear guidelines on transitioning from the Simulator[4] to the device, making the switch seamless. Notably, the device imposes limitations on the number of Processing Elements (PEs) allocatable at any given time due to bandwidth-constrained memory transfer commands via a Remote Procedure Call (RPC) backend. This constraint is expected to be addressed in the forthcoming CSSoft 2.2 software release for WSC, which will streamline transitions between simulator programs and wafer-scale clusters. It will include the ability to pre-stage data on one of the cluster's worker nodes, thereby circumventing the need for gRPC transfers.

6.3 Build Process

Build times for the simulator are expedited since it directly invokes the `cslc` compiler. Conversely, device compilation is managed by the SLURM instance managing the Wafer cluster[5] in EIDF and is abstracted from the user. Our program's execution times on the device are depicted in Figure 6.1.

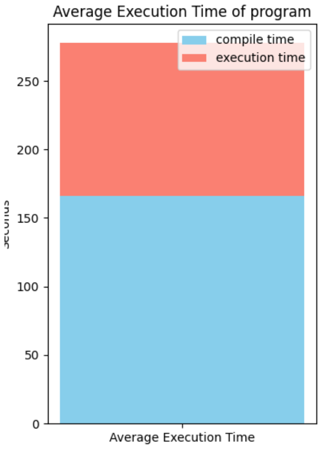

Figure 6.1: *Average Execution Times* of a Matrix Profiling program on the Cerebras

We initially tested our implementation on the simulator using small inputs. Once an acceptable error rate in the matrix profile was achieved, we ported the application to run on the wafer cluster setup at EDIF and the process was seamless.

[4] https://sdk.cerebras.net/appliance-mode

[5] https://docs.cerebras.net/en/latest/wsc/general/slurm-integration.html#slurm-integration

7 Conclusion

This work introduces an Matrix Profiling algorithm, based on the SCAMP algorithm, designed to compute matrix profiles on the Cerebras WSE-2. We provide an overview of concepts related to matrix profile computation and the architecture of the Cerebras WSE-2. Following this background, we delve into the specifics of our implementation, discussing the general architecture, including the division of computation between the Driver Application and Accelerated Kernel, as well as detailing two concrete kernel designs: the Vanilla and the Tiled Kernel.

We then address implementation considerations specific to the Cerebras WSE-2, outlining the steps involved in setting up and executing programs on the device. Subsequently, we conduct performance analyses using experiments performed on the Wafer Scale Cluster setup in the Edinburgh International Data Facility. These experiments include analyzing the performance capabilities of the Cerebras WSE-2 and deriving execution models for various tile sizes and memory consumption patterns. Notably, we highlight the challenge posed by the limited memory capacity of the processing elements (PEs) present in the device.

The insights derived from our experiments are valuable for understanding the potential of the Cerebras WSE-2 in practical applications. We also identify areas for future exploration, such as extending the implementation to handle row- and column-aggregates reduction on the wafer, optimizing inter-PE communication, implementing multi-stage `memcpy` to the device, and exploring strategies to utilize the entire wafer for calculation. Overall, we believe that the presented design can serve as a foundation for further research and development in this domain.

List of Figures

List of Tables