



UNIVERSIDAD  
DE MÁLAGA

Programa de Doctorado en Ingeniería Mecatrónica

Departamento de Arquitectura de Computadores

Escuela de Ingenierías Industriales

TESIS DOCTORAL

# **Accelerating Time Series Analysis via Near-Data-Processing Approaches**

Iván Fernández Vega


Abril de 2023

Dirigida por:  
Óscar Plata,  
Eladio Gutiérrez



UNIVERSIDAD  
DE MÁLAGA

AUTOR: Iván Fernández Vega

 <https://orcid.org/0000-0001-6133-5670>

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga



Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional:

<https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>

Cualquier parte de esta obra se puede reproducir sin autorización pero con el reconocimiento y atribución de los autores.

No se puede hacer uso comercial de la obra y no se puede alterar, transformar o hacer obras derivadas.

Esta Tesis Doctoral está depositada en el Repositorio Institucional de la Universidad de Málaga (RIUMA): [riuma.uma.es](http://riuma.uma.es)



## DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D. Iván Fernández Vega

Estudiante del programa de doctorado Ingeniería Mecatrónica de la Universidad de Málaga, autor de la tesis, presentada para la obtención del título de doctor por la Universidad de Málaga, titulada: Accelerating Time Series Analysis via Near-Data-Processing Approaches

Realizada bajo la tutorización de Óscar G. Plata González y dirección de Óscar G. Plata González y Eladio D. Gutiérrez Carrasco.

### DECLARO QUE:

La tesis presentada es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, conforme al ordenamiento jurídico vigente (Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), modificado por la Ley 2/2019, de 1 de marzo.

Igualmente asumo, ante la Universidad de Málaga y ante cualquier otra instancia, la responsabilidad que pudiera derivarse en caso de plagio de contenidos en la tesis presentada, conforme al ordenamiento jurídico vigente.

En Málaga, a 24 de abril de 2023

Fdo.: Iván Fernández Vega Doctorando	Fdo.: Óscar G. Plata González Tutor
Fdo.: Óscar G. Plata González Directores de tesis	Fdo.: Eladio D. Gutiérrez Carrasco



## CERTIFICADO AUTORÍA DIRECTORES

Dr. D. Óscar G. Plata González.  
Catedrático del Departamento de Ar-  
quitectura de Computadores de la Uni-  
versidad de Málaga.

Dr. D. Eladio D. Gutiérrez Carrasco.  
Profesor Titular del Departamento de  
Arquitectura de Computadores de la  
Universidad de Málaga.

### CERTIFICAN:

Que la memoria titulada “Accelerating Time Series Analysis via Near-Data-Processing Approaches”, ha sido realizada por D. Iván Fernández Vega bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería Mecatrónica.

En Málaga, a 24 de abril de 2023

Dr. D. Óscar G. Plata González.  
Codirector de la tesis.

Dr. D. Eladio D. Gutiérrez Carrasco.  
Codirector de la tesis.



UNIVERSIDAD  
DE MÁLAGA

## AUTORIZACIÓN LECTURA DE LA TESIS

Los abajo firmantes declaran, bajo su responsabilidad, que autorizan la lectura de la tesis del doctorando D. Iván Fernández Vega, con DNI , titulada “Accelerating Time Series Analysis via Near-Data-Processing Approaches”.

En Málaga, a 24 de abril de 2023

Dr. D. Óscar G. Plata González.  
Codirector de la tesis.

Dr. D. Eladio D. Gutiérrez Carrasco.  
Codirector de la tesis.



UNIVERSIDAD  
DE MÁLAGA



# INFORME DE UTILIZACIÓN DE PUBLICACIONES QUE AVALAN LA TESIS DOCTORAL

Dr. D. Óscar G. Plata González y Dr. D. Eladio D. Gutiérrez Carrasco, codirectores de la tesis doctoral de D. Iván Fernández Vega, con DNI , titulada “Accelerating Time Series Analysis via Near-Data-Processing Approaches”, declaran que ninguna de las publicaciones que avalan dicha tesis han sido utilizadas en tesis doctorales anteriores.

En Málaga, a 24 de abril de 2023

Dr. D. Óscar G. Plata González.  
Codirector de la tesis.

Dr. D. Eladio D. Gutiérrez Carrasco.  
Codirector de la tesis.



UNIVERSIDAD  
DE MÁLAGA

*A mi hermano Antonio*



UNIVERSIDAD  
DE MÁLAGA

# Agradecimientos

---

Sirvan estas líneas para agradecer a todas aquellas personas y entidades que de alguna manera han contribuido a que esta tesis haya sido posible.

En primer lugar, agradezco a mis directores de tesis, Óscar Plata y Eladio Gutiérrez por darme la posibilidad de realizar esta tesis con ellos y unirme a su grupo de investigación, Aldebarán. Este entorno ha servido para motivarme y aspirar a generar resultados de calidad. Gracias por ayudarme a mejorar mi autonomía en la generación de ideas y al mismo tiempo proveerme del feedback y directrices necesarias para tomar el camino más exitoso desde el punto de vista académico. Ha sido un verdadero placer compartir estos años con vosotros.

En segundo lugar, me gustaría agradecer a mis compañeros por haber contribuido a crear un ambiente ameno durante estos años. Especial mención merece Ricardo, ha sido muy enriquecedor compartir grupo de investigación y tu feedback ha sido muy valioso. Por otro lado, José Carlos, Andrés, Rubén, Bernabé, Fran, José Manuel, Denisa y Elena: los ratos que hemos pasado juntos y las discusiones que hemos tenido sin duda han contribuido a generar nuevas ideas y solventar los desafíos que han ido aconteciendo. No me olvido de los técnicos, Paco y Juanjo, ni de Carmen, cuyo soporte ha sido vital durante estos años.

Third, I would also like to really thank Prof. Onur Mutlu and from SAFARI research group at ETH Zürich for giving me the opportunity of joining his group. I find myself very fortunate of being part of this top-class environment. My two stays there were extremely productive not only from the academic point of view but also from a personal grow perspective. I really like that this collaboration is still going on and hope to continue contributing to the group. I would also like to thank Prof. Juan Gómez-Luna for mentoring me there over these years at SAFARI, and both Christina Giannoula and Aditya Manglik, who more than collaborators I consider as very good friends. I hope we continue collaborating in the future with you in exciting and top-notch projects!

En cuarto lugar, me gustaría agradecer a mis padres, Antonio y Victoria, el apoyo prestado durante estos años, el haberme inculcado el valor de las cosas y el hacerme ver que la perseverancia son claves para conseguir los objetivos que me propongo. Aunque mi hermano Antonio no pueda entender estas palabras, también le agradezco a él haber estado ahí y de alguna manera influenciarme positivamente. También agradezco a mi abuelo Cristóbal, por transmitirme el buen hacer, y al resto de mi familia en general. Por último, me gustaría agradecerle a Miriam estar a mi lado y apoyarme durante los episodios más difíciles, tu cariño ha sido clave y muy importante para mi.

Finalmente, también debo agradecer las fuentes de financiación que han contribuido a esta tesis. En concreto, a los proyectos TIN2016-80920-R, PID2019-105396RB-I00, UMA18-FEDERJA-197 y JA P18-FR-3433, y las Eurolab4HPC y HiPEAC collaboration grants.

# Abstract

---

The explosion of the Internet-Of-Things and Big Data era has resulted in the continuous generation of a very large amount of data, which is increasingly difficult to store and analyze [112]. Small sensors and devices produce a significant portion of this data [200], which includes observations (e.g., temperature, voltage, sound) sampled over time. Such a collection of data is also referred to as a time series, a common data representation in almost every scientific discipline and business application [167], for instance, epidemiology, genomics, neuroscience, environmental sciences, and stock markets. Time series analysis (TSA) is an useful set of tools that split the time series data into *subsequences* of consecutive data points to extract valuable information. This information can be used, for example, to filter out irrelevant subsequences or to find subsequences of interest. In real-world applications, TSA can perform heart beat analysis on a mobile medical device to predict a heart attack [118] or early earthquake detection [41]).

In this thesis, we characterize state-of-the-art time series analysis algorithms and find their bottlenecks in commodity architectures. Based on those bottlenecks, we propose software and hardware solutions to accelerate time series analysis and make its computation as energy-efficient as possible. To this end, we provide four contributions: PhiTSA, NATSA, MATSA and TraTSA.

First, we present *PhiTSA*. In this work, we optimize a state-of-the-art TSA algorithm in a many-core processor with integrated 3D-stacked RAM (Intel Xeon Phi). Our implementation exploits the underneath hardware as much as possible by adding vectorization and the use of two memory technologies (slower/faster). We use this implementation to identify the hotspots of the application, and we find that 1) the arithmetic intensity of the algorithm is very low and 2) when using the (faster) 3D-stacked memory, the performance increases significantly. Based on that, we conclude that the algorithm is memory bound.

Second, we present *NATSA*. In this work, we propose a novel Processing-Near-Memory accelerator based on 3D-stacked RAM that aims to reduce the memory

boundeness of TSA. The key idea is to place custom floating-point compute units close to the memory stack, improving the bandwidth and reducing latency. By reducing the generality of a CPU, NATSA provides the highest performance and best energy-efficiency when compared to its baselines.

Third, we propose *MATSA*. In this work, we propose a Processing-Using-Memory accelerator based on non-volatile memory cells that minimizes data movement by performing in-situ computation. The key idea is to use the actual memory cells in conjunction with reconfigurable sense amplifiers to perform the TSA computation. *MATSA* provides the highest performance and energy efficient solution so far for a wide range of time series applications.

Finally, we present *TraTSA*. In this work, we study the TSA algorithms from the software point of view, and find that floating-point operations are a big contributor to the energy consumption. Based on that, we evaluate the potentials of TSA when using a transprecision computing approach. Transprecision computing enables the possibility of reducing the number of bits of the floating-point operations while obtaining accurate results. To this end, we propose a transprecision framework that demonstrates that TSA algorithms are good candidates for reduced-precision computation.



# Contents

<b>Agradecimientos</b>	<b>I</b>
<b>Abstract</b>	<b>III</b>
<b>Contents</b>	<b>IX</b>
<b>List of Figures</b>	<b>XVI</b>
<b>List of Tables</b>	<b>XVII</b>
<b>1.- Introduction</b>	<b>1</b>
1.1. TSA Applications . . . . .	3
1.2. TSA Bottlenecks . . . . .	4
1.3. Thesis Motivation and Contributions . . . . .	8
1.4. Thesis Structure . . . . .	10
<b>2.- Background and Related Work</b>	<b>11</b>
2.1. Time Series Analysis . . . . .	11
2.1.1. SCRIMP Implementation . . . . .	14
2.1.2. SCAMP Implementation . . . . .	16
2.1.3. DTW Implementation . . . . .	17
2.2. Mitigating the Data Movement Bottleneck . . . . .	19

2.2.1. Processing-Near-Memory . . . . .	19
2.2.2. Processing-Using-Memory . . . . .	21
2.3. Transprecision Computing . . . . .	23
<b>3.- PhiTSA: Using a Xeon Phi to Optimize and Characterize TSA</b>	<b>27</b>
3.1. Motivation and Key Idea . . . . .	27
3.2. Intel Xeon Phi Knights Landing . . . . .	27
3.3. PhiTSA Optimizations of SCRIMP . . . . .	29
3.3.1. Updating P and I . . . . .	30
3.3.2. Increasing Arithmetic Intensity . . . . .	32
3.3.3. Memory Allocation Policy and Scalability . . . . .	34
3.4. Evaluation . . . . .	35
3.4.1. Methodology . . . . .	35
3.4.2. Results . . . . .	35
Speedups . . . . .	35
Memory Bandwidth Results . . . . .	38
Sensibility to Time Series Length and Window Size . . . . .	40
Floating Point Performance . . . . .	41
Real-World Applications . . . . .	41
<b>4.- NATSA: A PNM Accelerator for TSA</b>	<b>45</b>
4.1. Motivation and Key Idea . . . . .	45
4.2. NATSA Architecture . . . . .	46
4.2.1. NATSA Processing Units (PUs) . . . . .	47
4.2.2. Workload Partitioning Scheme . . . . .	48
4.2.3. Programming Interface . . . . .	49
4.3. Evaluation . . . . .	50
4.3.1. Methodology . . . . .	50
4.3.2. Workload . . . . .	52

4.3.3. Results . . . . .	52
Performance of NATSA . . . . .	53
Power, Energy and Area Consumption . . . . .	54
NATSA Design Space Exploration . . . . .	56
Performance of General-Purpose Cores . . . . .	56
Accuracy and Sensitivity to Subsequence Length . . . . .	58
<b>5.- MATSA: A PUM Accelerator for TSA</b>	<b>59</b>
5.1. Motivation and Key Idea . . . . .	59
5.2. MATSA Architecture . . . . .	60
5.2.1. Overview . . . . .	60
5.2.2. MATSA Subarrays . . . . .	61
5.2.3. sDTW Challenges in NVM-PUM . . . . .	62
5.2.4. Supported Operations . . . . .	63
5.2.5. Data Mapping . . . . .	64
5.2.6. Execution Flow . . . . .	65
5.2.7. System Integration . . . . .	66
5.3. Evaluation . . . . .	67
5.3.1. Methodology . . . . .	68
5.3.2. Workloads . . . . .	69
5.3.3. Results . . . . .	70
MATSA Characterization . . . . .	70
MATSA Comparison . . . . .	72
<b>6.- TraTSA: A Transprecision Framework for TSA</b>	<b>77</b>
6.1. Motivation and Key Idea . . . . .	77
6.2. TraTSA Framework . . . . .	78
6.2.1. Overview of TraTSA . . . . .	78
6.2.2. Transprecision SCRIMP-CPU (TranSCRIMP) . . . . .	80

6.2.3. Transprecision SCAMP-CPU (TranSCAMP) . . . . .	82
6.2.4. Transprecision SCAMP-FPGA (TranSCAMPfpga) . . . . .	83
6.3. Top-K Accuracy Metric . . . . .	87
6.9. Evaluation . . . . .	88
6.9.1. Methodology . . . . .	88
6.9.2. Results . . . . .	90
Short Time Series Accuracy . . . . .	90
Large Time Series Accuracy . . . . .	100
TraTSA Performance . . . . .	104
Energy Savings . . . . .	105
<b>7.- Conclusions</b>	<b>109</b>
7.1. Future Work . . . . .	111
<b>Appendices</b>	<b>113</b>
<b>A.- Resumen en español</b>	<b>113</b>
A.1. Introducción . . . . .	113
A.1.1. Aplicaciones de AST . . . . .	115
A.1.2. Cuellos de Botella de AST . . . . .	116
A.1.3. Motivación y Contribuciones de la Tesis . . . . .	118
A.2. PhiTSA: Usando un Xeon Phi para Optimizar y Caracterizar AST	120
A.2.1. Idea y Motivación . . . . .	120
A.2.2. Contribuciones . . . . .	121
A.2.3. Evaluación . . . . .	121
A.3. NATSA: Un Acelerador PNM para AST . . . . .	122
A.3.1. Idea y Motivación . . . . .	122
A.3.2. Propuesta . . . . .	123

A.3.3. Evaluación . . . . .	123
A.4. MATSA: Un Acelerador PUM para AST . . . . .	124
A.4.1. Contribuciones . . . . .	124
A.4.2. Evaluación . . . . .	124
A.5. TraTSA: Un Framework Transpreciso Para AST . . . . .	125
A.5.1. Idea y Motivación . . . . .	125
A.5.2. Contribuciones . . . . .	125
A.5.3. Evaluación . . . . .	126
A.6. Conclusiones y Trabajo Futuro . . . . .	127
<b>Bibliography</b>	<b>131</b>



UNIVERSIDAD  
DE MÁLAGA

# List of Figures

1.1. Example of TSA application. In this processing flow, TSA acts as a filter to avoid most of the computation by selecting the relevant subsequences (anomalies) and discarding the irrelevant ones (expected behavior). . . . .	2
1.2. Memory bandwidth usage (bars) and normalized performance (lines) of a parallel and vectorized version of SCRIMP [58] running on an Intel Xeon Phi 7210. . . . .	5
1.3. Roofline analysis of a parallel and vectorized version of SCRIMP [58] running on an Intel Xeon Phi 7210. . . . .	6
1.4. Roofline analysis of a parallel version of DTW running on an Intel Xeon Phi 7210. . . . .	6
2.1. Example of two subsequences, $T_{i,m}$ and $T_{j,m}$ , of a given time series $T$ . Distance $d_{i,j}$ is calculated from these subsequences. When calculating the matrix profile $P$ , subsequences in the exclusion zone of $T_{i,m}$ are ignored. . . . .	11
2.2. A time series with some anomalies (discords) and its matrix profile (MP). The anomalies appear as higher distance values in the profile.	12
2.3. Example of distance matrix (D), profile (P), and profile index (I). $P_i$ holds the minimum distance calculated in row $D_i$ , and $I_i$ holds the index $j$ of the subsequence that has such distance. Cells in the exclusion zone are coloured red. . . . .	13

2.4. Example of similarity calculation between two subsequences (blue and green). The Euclidean approach in a) provides a low similarity as it only compares each $i^{th}$ point of blue with each $i^{th}$ point of green. However, DTW in b) succesfully matches the points of the subsequences, thus finds the high similarity. . . . .	14
2.5. a) Warping matrix example for a reference time series $R$ and a query subsequence $Q$ . The DTW distance between $R$ and $Q$ is the minimum value of the last row of the matrix. b) Sakoe-Chiba band. c) Itakura parallelogram. . . . .	18
2.6. Sample PNM architecture (I). Compute Units are placed close to the HBM cube, connected via a short and wide interface. . . . .	20
2.7. Sample PNM architecture (II). UPMEM-based PNM system with a host CPU, standard main memory, and PNM-enabled memory (left), and internal components of a UPMEM PNM chip (right). Figure taken from [183]. . . . .	20
2.8. a) Crossbar organization. b) Magneto-resistive cell. c) Reconfigurable SA that performs in-memory operations based on the voltage across the bitline when two cells are activated. . . . .	22
2.9. Overview of the floating-point types used for energy evaluation. . .	24
3.1. Intel Xeon Phi KNL overview [93]. . . . .	28
3.2. Calculation of the Matrix Profile $P$ and the Matrix Profile index $I$ from the distance matrix. For each column (or row) $D_i$ we calculate the minimum distance and the index of the subsequence providing such minimum distance. . . . .	29
3.3. SCRIMP's lock-based implementation to update the Matrix Profile and the Matrix Profile index without potential conflicts. . . . .	31
3.4. SCRIMP's expanded structure for Matrix Profile distances (top) and indexes (bottom) in the privatization technique. . . . .	31
3.5. Privatization-based implementation of the SCRIMP computation. .	33
3.6. SCRIMP's inner loops unrolling and vectorization. . . . .	34
3.7. SCRIMP speedup in the Intel Xeon Phi KNL, calculated using a random time series of of $2^{17}$ (131072) elements and a fixed window size of 1024. . . . .	36



3.8. SCRIMP memory bandwidth utilization in the Intel Xeon Phi KNL, using different configurations, normalized as percentage of execution time. DDRo stands for DDR4 ordered and DDRr stands for DDR4 random, while HBMo means HBM+DDR4 ordered and HBMr means HBM+DDR4 random. . . . .	39
3.9. Average DDR4 bandwidth utilization, measured varying the number of threads in an execution of SCRIMP, using a random series of $2^{18}$ elements and a window of 1024. . . . .	39
3.10. Average HBM bandwidth utilization, measured varying the number of threads in an execution of SCRIMP, using a random series of $2^{18}$ elements and a window of 1024. . . . .	39
3.11. Seismology data. This dataset, which consists of approximately 40,000 elements, presents several peaks corresponding to earthquakes. The Matrix Profile is able to accurately identify them, which are represented as higher values of distance for the corresponding subsequences, as they are discords with respect to the whole time series. . . . .	42
3.12. Penguin data. This dataset, which consists of approximately 110,000 elements, presents several motifs that can help a biologist to identify when the penguin is diving, for example. . . . .	43
3.13. Neuroscience data. This dataset, which consists of approximately 1,030,000 elements, presents several discontinuities which are represented in the graph as values of 0 for readability. The Matrix Profile is able to obtain coherent results even with those discontinuities, proof of robustness of the algorithm. . . . .	43
4.1. NATSA design and integration next to HBM memory. NATSA is connected directly to the HBM interface. . . . .	46
4.2. Example of the diagonal scheduling scheme for two processing units, denoted as <i>PU0</i> (green) and <i>PU1</i> (white). Arrows show direction of computation. . . . .	49
4.3. Speedup with respect to the baseline platform (DDR4-OoO) using double precision data. . . . .	53
4.4. Dynamic power consumption for simulated and real hardware platforms. . . . .	54
4.5. Energy consumption for simulated and real hardware platforms. . .	55

4.6. Area comparison of different hardware platforms. . . . .	55
4.7. Speedup over the baseline DDR4-OoO and memory bandwidth usage for general-purpose platforms. . . . .	57
4.8. ECG (left) and seismology (right) data along with their profiles, using double and single precision, where events are easily visible. .	58
5.1. MATSA's high-level Architecture and Data Mapping. Note that modifications at <i>Inter-Bank</i> , <i>Inter-Mat</i> , and <i>Inter-Subarray</i> lev- els with respect to common NVM devices are negligible, easing fabrication and compatibility. . . . .	60
5.2. MATSA's Reconfigurable Sense Amplifier. Latch register is reused for Addition and Diagonal Copy operations. . . . .	62
5.3. Wavefront-based sDTW computation. In a, PEs are able to cal- culate their matrix elements in parallel. In b, the pipeline is full and PEs are also working on different queries. . . . .	66
5.4. MATSA integration. a) High-performance accelerator (PCIe card). b) Portable accelerator (USB stick). c) Embedded acceler- ator (small chip). . . . .	67
5.5. Execution time when varying cell read and write la- tencies (ref_size=128K, query_size=8K, num_queries=8K, matsa_cols=128K). . . . .	70
5.6. Execution energy when varying cell read and write en- ergies (ref_size=128K, query_size=8K, num_queries=8K, matsa_cols=128K). . . . .	71
5.7. Execution time when varying workload sizes (num_queries=8K, matsa_cols=128K). . . . .	71
5.8. Execution energy when varying problem sizes (num_queries=8K, matsa_cols=128K). . . . .	72
5.9. Execution time when varying MATSA sizes. . . . .	72
5.10. Execution times of MATSA-Embedded (num_cols=32K) and MATSA-Portable (num_cols=256K) versus baselines for different workload sizes (rd_latency=5ns, wr_latency=10ns). . . . .	73

5.11. Execution energy consumption of MATSA-Embedded (num_cols=32K) and MATSA-Portable (num_cols=256K) versus baselines for different workload sizes (rd_energy=50nJ, wr_energy=70nJ). . . . .	74
5.12. Execution times of MATSA-HPC versus baselines for different workload sizes (rd_latency=5ns, wr_latency=10ns, num_cols=1M). . . . .	74
5.13. Execution energy consumption of MATSA-HPC versus baselines for different workload sizes (rd_energy=50nJ, wr_energy=70nJ, num_cols=1M) . . . . .	75
6.1. TraTSA overview and its components. The user provides a time series file (.txt) and a configuration file (.cfg) to the wrapper. Then, the wrapper invokes matrix profile either in the CPU or in the FPGA. Finally, the wrapper provides the user the transprecision matrix profile (.csv) and some statistics (.stats). . . . .	79
6.2. Example of TraTSA's cfg file. . . . .	79
6.3. Transprecision SCRIMP (TranSCRIMP) algorithm (transprecision operations highlighted). . . . .	81
6.4. FPGA implementation overview. TranSCAMPfpga is composed of six kernels optimized for a Xilinx Alveo U50 FPGA that compute transprecision SCAMP algorithm using the data in the HBM. . . . .	84
6.5. TranSCAMPfpga systolic array example. . . . .	86
6.6. FPGA implementation diagram. . . . .	86
6.7. TranSCRIMP Top-100 motif accuracy with respect to double. . . . .	91
6.8. TranSCRIMP Top-100 discord accuracy with respect to double. . . . .	92
6.9. TranSCAMP Top-100 motif accuracy with respect to double. . . . .	93
6.10. TranSCAMP Top-100 discord accuracy with respect to double. . . . .	94
6.11. TranSCAMP <i>Song</i> profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal. . . . .	95
6.12. TranSCAMP <i>ECG_short</i> profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal. . . . .	96

6.13. TranSCAMP <i>Power_short</i> profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal. . . . .	96
6.14. TranSCAMP <i>Seismology_short</i> profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal. . . . .	97
6.15. TranSCAMP <i>Human Activity</i> profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal. . . . .	98
6.16. TranSCAMP <i>Penguin Behavior</i> profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal. . . . .	98
6.17. TranSCAMPfpga Top-1000 motif accuracy results with respect to double precision using a recalculation factor of 64K elements. . . .	101
6.18. TranSCAMPfpga Top-1000 discord accuracy results with respect to double precision using a recalculation factor of 64K elements. . . .	102
6.19. TranSCAMPfpga Top-1000 accuracy results with respect to double precision when varying the recalculation factor. . . . .	103
6.20. EPG matrix profile when varying the recalculation factor using TranSCAMPfpga for a given mantissa and exponent combination. .	104
6.21. Execution time for different platforms when computing <i>Seismology_short</i> , using a window size of 512 elements (exp. 7 man. 10). .	105
6.22. Normalized FPU energy using FPNew. . . . .	106

# List of Tables

1.1. Time Series Analysis application examples . . . . .	3
2.1. Overview of different NVM technologies [45]. . . . .	23
2.2. Floating-point bit counts, ranges and smallest numbers . . . . .	24
3.1. SCRIMP execution on Intel Xeon Phi KNL versus STOMP on NVIDIA Tesla K80 <sup>‡</sup> . . . . .	38
3.2. Time for SCRIMP using only DDR4 in random order. . . . .	40
3.3. Time for SCRIMP using only DDR4 in sequential order. . . . .	40
3.4. Time for SCRIMP using HBM plus DDR4 in random order. . . . .	40
3.5. Time for SCRIMP using HBM plus DDR4 in sequential order. . . . .	40
3.6. GFLOP/s for SCRIMP with different configurations. . . . .	41
4.1. Synthetic time series for performance evaluation. . . . .	52
4.2. Execution time (in seconds) for single-precision and double-precision. . . . .	54
4.3. NATSA design components for 48 PUs. . . . .	56
5.1. MATSA design exploration parameters. . . . .	69
5.2. Workload parameters used in our evaluation. . . . .	69
6.1. TranSCAMPfpga kernel resource utilization . . . . .	87
6.2. Time series dataset parameterization . . . . .	90
6.3. Mixed precision Top-100 accuracy results . . . . .	99
6.4. Energy consumption for different platforms when computing <i>Seis-</i> <i>mology_short</i> , using a window size of 512 elements (exp. 7 man. 10) . . . . .	105



UNIVERSIDAD  
DE MÁLAGA

# 1 Introduction

---

The explosion of the Internet-Of-Things and Big Data era has resulted in the continuous generation of a very large amount of data, which is increasingly difficult to store and analyze [112]. Small sensors and devices produce a significant portion of this data [200], which includes observations (e.g., temperature, voltage, sound) sampled over time. Such a collection of data is also referred to as a *time series*, a common data representation in almost every scientific discipline and business application [167], for instance, epidemiology, genomics, neuroscience, environmental sciences, and stock markets. *Time series analysis* (TSA) splits the time series into *subsequences* of consecutive data points to extract information.

To find subsequences of interest, TSA algorithms define a *distance metric*, e.g., Euclidean Distance (ED) or Dynamic Time Warping (DTW). This distance metric represents how similar are two given subsequences, meaning that the lower the distance value is, the more similar are them. Based on the distance metric, the algorithm classifies the subsequences that have low distance as *motifs* [148] (similarities) and high distance as *discords* [100] (anomalies). This classification is a critical step before further analysis via domain-specific algorithms or human experts, enabling ***TSA as a tool of vital importance and generality***.

TSA algorithms can be used to filter out those subsequences that match an expected behavior, leaving only anomalies for the computationally expensive domain-specific algorithm (e.g., [126]), i.e., only a small amount of critical data that need to be further analyzed. One example is SquiggleFilter [50], a TSA-based accelerator that processes the Mini-ION sequencer's output and filters everything except sequences of interest, pruning more than 85% of the computation. However, SquiggleFilter optimizations rely on a modified version of DTW that loses generality to other applications. Figure 1.1 describes the filtering process based on an example data flow for electrocardiogram data and a generic TSA algorithm.

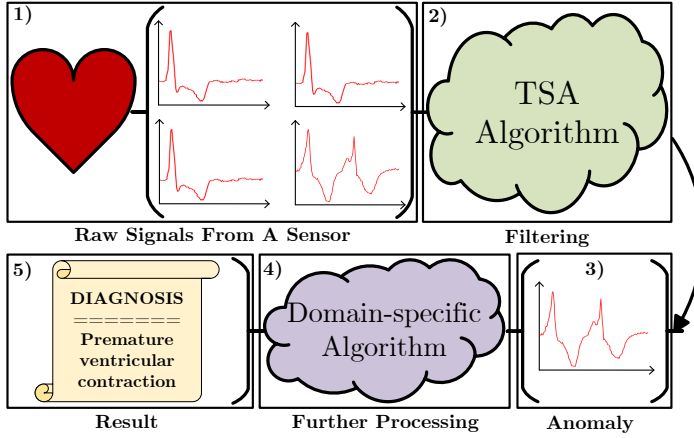


Figure 1.1: Example of TSA application. In this processing flow, TSA acts as a filter to avoid most of the computation by selecting the relevant subsequences (anomalies) and discarding the irrelevant ones (expected behavior).

**Key Motivation I:** TSA is useful in many domains, but prior acceleration proposals mostly rely on modifications of the algorithms that loose generality

Diving into the implementation point of view, we find that *Matrix Profile* [196] (MP) is the state-of-the-art set of tools to perform TSA, which comprises of both Euclidean Distance and Dynamic Time Warping algorithms. There are several CPU-based, GPU-based and even FPGA-based MP implementations in the literature [196, 86, 202, 204]. These CPU, GPU and FPGA approaches are embarrassingly parallelizable, relying on simple arithmetic operations over data to calculate distances. In this sense, the naive parallelization approach consists on spreading all the distances to calculate across the computational resources since—in principle—there are no dependencies between them. This scheme provides almost the size of the time series fully independent execution flows, much higher than typical core count in commodity CPUs and GPUs. In order to accelerate parallel implementations of TSA under those architectures, increasing the core count appears to be a promising approach.

With such opportunity in mind, we first characterize a MP implementation in a many-core machine (Intel Xeon Phi KNL) and find that core count is not the only bottleneck for TSA. Concretely, we observe that the scalability stops after a certain number of cores and conclude that *the performance and energy efficiency of TSA algorithms are heavily burdened by data movement*.



This observation is based on 1) the low arithmetic intensity of the algorithm and 2) the need to access large amounts of time series data from the memory. In other words, the amount of computation per data access is not enough to hide the memory latency, so the algorithm is memory bound in a conventional CPU architecture. Within this context, Processing-Near-Memory (PNM) and Processing-using-Memory (PUM) place computation closer to data, thus reducing the impact of data movement. Based on this, we consider PNM and PUM architectures as promising approaches to accelerate TSA.

**Key Motivation II:** TSA is bottlenecked in conventional architectures by 1) the low core count with respect to the potential parallelism and 2) data movement between memory and processing units

In the rest of this Introduction chapter, we further elaborate the two key motivation points and present the contributions of this thesis.

## 1.1. TSA Applications

The first point of motivation of this thesis is based on the fact that TSA constitutes one of the most important data mining primitives thanks to its generality in detecting anomalies and similarities for a wide range of applications. Note that this desirable feature is enabled by the domain-agnostic nature of the TSA algorithms, as their outputs are simple distance values. Table 1.1 presents a few examples of TSA applications.

Field	References	Field	Reference
Bioinformatics	[51]	Speech Recognition	[25]
Robotics	[175]	Weather Prediction	[131]
Neuroscience	[51]	Entomology	[170]
Machine Learning	[186]	Geophysics	[181]
Econometrics	[83]	Statistics	[162]
Finance	[182]	Control Engineering	[31]
Signal Processing	[109]	Pattern Recognition	[189]
Communication	[111]	Medicine	[88]
Astronomy	[184]	Social Networks	[144]
Clustering	[186]	Classification	[99]
Earthquakes	[181]	GPS Tracking	[108]
Virtual Reality	[169]	Gesture Recognition	[89]
Trajectories	[24]	Traffic Monitoring	[119]

Table 1.1: Time Series Analysis application examples

In statistics, econometrics, meteorology, and geophysics, the primary goal of time series analysis is prediction and forecasting. At the same time, in signal processing, control engineering, and communication engineering, it is used for signal detection and estimation. In data mining, pattern recognition, and machine learning, time series motif and discord discovery are used for clustering, classification, anomaly detection, and forecasting. Finally, the most important application of time series motif and discord discovery is clustering seismic data and discovering earthquake pattern clusters from the continuous seismic recording. Consequently, seismic clustering can be applied to earthquake relocation and volcano monitoring to help improve earthquake and volcanic hazard assessments.

**End-to-End Application Benefits of using TSA.** TSA is critical for real-life examples of end-to-end applications. For instance, [90] predicts circulatory failure in intensive care units. In this scenario, 90% of the execution time is dominated by TSA preprocessing, while the remainder 10% is used by the machine learning-based application to perform the classification. We also find many other real use case examples that can benefit from this approach, such as:

- **Earthquake detection** [41]. TSA can process the data from a seismograph and detect anomalies that can be further processed with complex algorithms.
- **Electroencephalography** [37]. Assuming an electroencephalograph that is monitoring a patient, TSA can be used to detect anomalies and trigger an alarm about that.
- **Virus Detection** [50]. During the genome assembly process, basecalling is a compute-intensive task that can be skipped for most queries using a TSA-based filter.

Based on that, we conclude that TSA 1) is widely applicable and 2) its acceleration significantly benefits the whole application execution time.

## 1.2. TSA Bottlenecks

The second point of motivation of this thesis is based on the fact that TSA is bottlenecked in commodity architectures. To identify those bottlenecks, we first develop an optimized and vectorized version of a state-of-the-art implementation of Matrix Profile. Then, we characterize it using a many-core machine (Intel Xeon Phi KNL) that also includes High-Bandwidth-Memory (HBM) in

conjunction with conventional DDR4. We first observe that while the hardware thread count (256) of the Intel Xeon Phi is relatively high compared to lower end machines, there are still way more independent executions flows (thousands, millions) than threads. However, thread availability is not the only bottleneck for TSA. In Figure 1.2, we present the performance results normalized to 1 thread (lines) and utilized memory bandwidth (bars) of this Euclidean Distance based implementation of MP, known as SCRIMP. We observe that, when using the DDR4 memory, the performance of SCRIMP does not scale beyond 32 threads, whereas the higher memory bandwidth provided by HBM enables SCRIMP to scale up to 128 threads. This shows that SCRIMP’s performance saturates on many-core architectures, because the achievable bandwidth saturates when the number of threads increases.

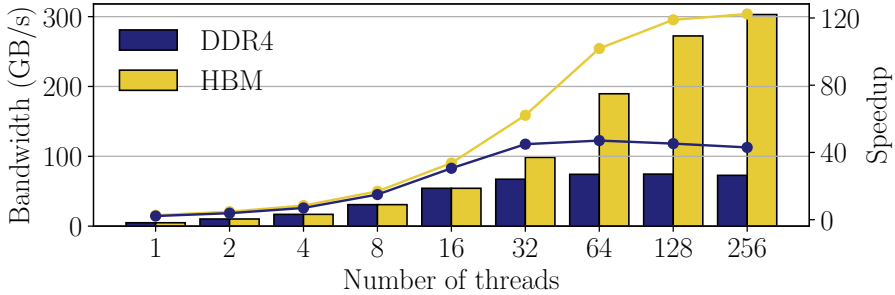


Figure 1.2: Memory bandwidth usage (bars) and normalized performance (lines) of a parallel and vectorized version of SCRIMP [58] running on an Intel Xeon Phi 7210.

To understand the cause for this memory boundedness, we perform the next experiment. We generate the roofline analysis for SCRIMP, as we show in Fig. 1.3, and observe that the arithmetic intensity of SCRIMP is significantly low. This fact confirms that the memory boundedness of SCRIMP is due to the low arithmetic intensity of the algorithm, which leads processing cores to be underutilized. Based on all these observations, we conclude that the performance of the state-of-the-art CPU-based implementation of SCRIMP is heavily bottlenecked by available memory bandwidth and data movement.

However, based on the similar operational simplicity of the TSA algorithms and the huge amount of data to process, we hypothesize that the bottlenecks are shared across other TSA approaches. To prove this statement, we also characterize a Dynamic-Time-Warping (DTW) implementation in the same architecture as SCRIMP and discuss the bottlenecks in other platforms.

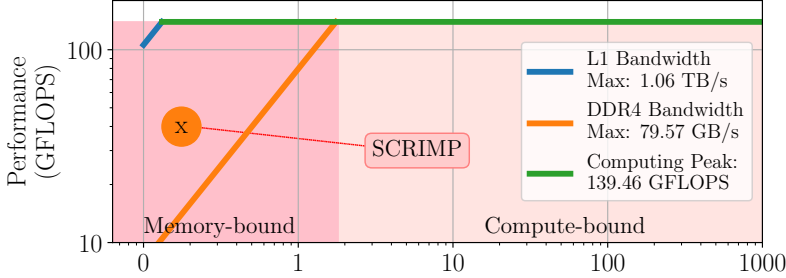


Figure 1.3: Roofline analysis of a parallel and vectorized version of SCRIMP [58] running on an Intel Xeon Phi 7210.

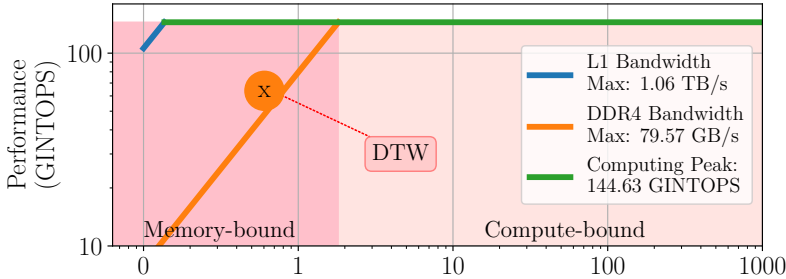


Figure 1.4: Roofline analysis of a parallel version of DTW running on an Intel Xeon Phi 7210.

**CPU platforms.** We profile the performance of DTW using the same many-core processor (Intel Xeon Phi 7210) as we used for SCRIMP, analyzing the execution of 16K queries of 8K elements each that are compared against a reference sequence of 32K elements. Figure 1.4 presents the roofline plot for the experiment.

First, we observe that DTW only exploits 41% of the system’s integer peak performance, i.e., 59 GINTOPS out of 145 GINTOPS, and it presents low arithmetic intensity (0.55 INTOP/Byte). Second, we find that the memory footprint of the execution is  $\approx 570\text{MB}$ . The total memory traffic generated during runtime is  $\approx 267\text{ GB}$ , putting the kernel above the DDR4 peak bandwidth most of the execution time.

**GPU platforms.** We find that several prior works [201, 124, 35, 42, 53]

propose to accelerate DTW using GPUs. However, these works rely on shared-memory optimizations that only perform for certain small query sizes. For large query sizes, these implementations either 1) do not work or 2) use high-latency global-memory to hold the main data structures of DTW, which results in large performance penalties. We develop a CUDA-based implementation that supports arbitrary sizes. Then, we analyze DTW using NVIDIA Visual Profiler and observe that its GPU core utilization is below 50% of the theoretical peak performance, and DTW execution is bottlenecked by a large number of accesses to the global memory.

**FPGA platforms.** Several prior works propose [97, 98, 16] to accelerate DTW using FPGAs. However, most of them have very limited onboard memory, and data has to be moved over narrow buses. We develop our own HLS-based FPGA implementation and find that 1) the number of computation units is not enough to exploit the inherent parallelism of DTW, and 2) the compute units spend most of their time waiting for the memory accesses to be served.

**PNM/PUM platforms.** One way of improving parallelism and reducing data movement costs is to perform computation near memory or using the actual memory cells. We analyze several approaches in those directions and detail their benefits and drawbacks:

- *General-purpose PNM.* This approach typically places small CPU cores in the same die as DRAM. The main benefit of this scheme is that the architecture can be potentially used for general-purpose applications. However, this comes at the cost of limited parallelism. We evaluate a general-purpose PNM (upmem baseline) in 5.3.3, and our evaluations show that this architecture is compute bound when performing the DTW computation.
- *Specialized PNM.* This approach typically places an ASIC accelerator in the same die as DRAM. The main benefit of this approach is that the processing elements are highly optimized for the target workload. However, the data still needs to be moved from memory to the accelerator. Even though an ASIC accelerator can be used in a PNM architecture, performance would still be bottlenecked by data movement between memory and the accelerator, similarly to general-purpose PNM.
- *SRAM-based PUM.* This approach uses SRAM-based memory arrays to perform in-situ computation (e.g., compute caches [10]). The main benefits of this approach are the high levels of parallelism and reduced data movement. Unfortunately, SRAM suffers from density and scalability issues [75], along with being radiation vulnerable.

- *DRAM-based PUM*. This approach uses DRAM-based memory arrays to perform in-situ computation (e.g., SIMDRAM [79]). However, this approach involves internal data movement to perform the operations, as data needs to be moved to specific compute-enabled rows before performing the actual operations. Moreover, DRAM suffers from data volatility and destructive read problems.

After identifying which compute paradigm fits better for TSA, the selection of the memory technology is crucial to ensure feasibility of the acceleration solutions. For example, the PNM approach is usually based on DRAM technologies, however high-bandwidth memories (e.g., HBM) are a must to ensure maximum occupancy of the processing elements. When moving to PUM and using the actual cells to perform the computation, it is possible to build accelerators using also DRAM technology, but in this case performance scaling is limited by 1) re-freshing times and specially by 2) the need of moving data to compute enabled rows. Because of that, crossbar-based NVM technologies can overcome those two issues, but it is challenging to support frequent write operations. This is because NVM-based architectures usually suffer from significant latency and energy penalties, and specially low endurance. However, emerging NVM technologies are a possible solution to overcome those drawbacks. We discuss deeper into the memory technology selection on the Background Chapter.

### 1.3. Thesis Motivation and Contributions

The increasing demand for processing data puts on the table the necessity of providing computing platforms that match the performance requirements. Moreover, energy consumption is a growing critical concern in those platforms. As we show in Section 1.1, TSA is a useful tool to preprocess the immense amount of data that are generated by contemporary devices, thus making the optimization and energy efficiency of TSA as a matter of vital importance. This motivates us to carefully study state-of-the-art TSA algorithms and propose solutions to mitigate their bottlenecks.

In recent times, computer architecture community has identified data movement between memory and processing units as the main bottleneck for scaling up performance. This fact is also applicable to TSA, as we show in Section 1.2. Based on that, platforms that place computation closer to data have been revisited and there are huge research efforts in this direction. We distinguish between Processing-Near-Memory, where compute units are placed physically closer

to memory, and Processing-using-Memory, where the actual memory cells are used to perform computation.

The main goal of this thesis is to study if PNM and PUM architectures are good candidates to accelerate TSA and make its computation energy-efficient. Additionally, we study TSA from the algorithmic point of view, reducing the number of floating-point bits required for computation and optimizing resources.

Our main contributions in this thesis and the related publications, which aim to fulfil the goal of it, are the following:

- **PhiTSA** [58]. We optimize and characterize state-of-the-art TSA algorithms in a many-core Intel Xeon Phi KNL platform. We identify data movement as a main bottleneck that 1) prevents from further acceleration and 2) consumes most of the execution energy.
- **NATSA** [57]. We propose a novel Processing-Near-Memory accelerator for TSA, known as NATSA. This accelerator places custom floating-point processing units close to High-Bandwidth-Memory, exploiting its memory channels and the lower latency of accesses. NATSA significantly improves performance and energy consumption with respect to commodity systems when performing Euclidean Distance based TSA computation.
- **MATSA** [Under Review]. We propose a novel Processing-Using-Memory accelerator for TSA, known as MATSA. The key idea is to exploit magnetoresistive memory crossbars to enable energy-efficient and fast time series computation in memory while overcoming endurance issues of other non-volatile memory technologies. MATSA: 1) leverages high levels of parallelism in the memory substrate by exploiting column-wise arithmetic operations, and 2) significantly reduces the data movement costs performing computation using the memory cells. We perform a design space exploration and demonstrate that our HPC version of MATSA greatly improves performance and energy efficiency over server CPU, GPU and PNM.
- **TraTSA** [56, 55]. We evaluate the benefits of applying Transprecision Computing to TSA, where the number of bits dedicated to floating-point operations is reduced. We develop TraTSA, a framework that enables tuning the precision of the operations for every use case, maximizing the resource utilization of the arithmetic units. Using this approach, it is possible to reduce the area required by the system, which enables the possibility of including more arithmetic units thus improve performance and reduce energy consumption.

Additionally, this thesis has contributed to several publications related to the mitigation of the data movement bottlenecks [69, 71, 82, 146, 68, 64].

## 1.4. Thesis Structure

The remainder of this thesis is structured in the following way. In Chapter 2 we provide some background and related work. In Chapter 3 we optimize and characterize a state-of-the-art TSA algorithm in a many-core machine. In Chapter 4 we present NATSA, a novel Processing-Near-Memory accelerator for TSA. In Chapter 5 we present MATSA, a novel Processing-Using-Memory accelerator for TSA. In Chapter 6 we present TraTSA, a framework to optimize the floating-point operations involved in TSA computation. Lastly, Chapter 7 presents the conclusions of this thesis.



# 2 Background and Related Work

---

In this chapter, we present the some background to support the rest of the thesis. First, we introduce time series analysis and its state-of-the art implementations. Second, we explain the key idea behind Processing Near/In Memory to mitigate the data movement bottleneck. Finally, we provide some insights on how transprecision computing can boost performance and energy efficiency.

## 2.1. Time Series Analysis

We begin our introduction to time series analysis by providing some basic definitions. A *time series*  $T$  is a sequence of  $n$  data points  $t_i$ , where  $1 \leq i \leq n$ , collected over time. Those data points are usually numeric values that can be represented in any format (e.g., integers or floats). A subsequence of  $T$ , also called a *window*, is denoted by  $T_{i,m}$ , where  $i$  is the index of the first data point, and  $m$  is the number of samples in the subsequence, with  $1 \leq i$ , and  $m \leq n - i$ . Figure 2.1 shows an example time series with two subsequences highlighted.

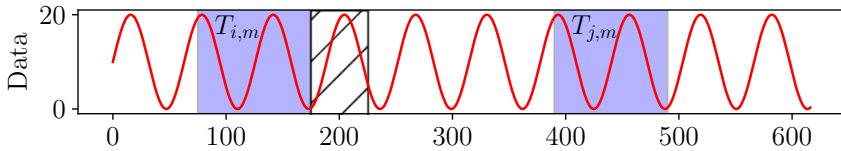


Figure 2.1: Example of two subsequences,  $T_{i,m}$  and  $T_{j,m}$ , of a given time series  $T$ . Distance  $d_{i,j}$  is calculated from these subsequences. When calculating the matrix profile  $P$ , subsequences in the exclusion zone of  $T_{i,m}$  are ignored.

Time series analysis algorithms usually try to find subsequences of interest, like *motifs* (repeated patterns) or *discords* (anomalies). Multiple techniques exist for time series motif and discord discovery [178, 148, 40, 174, 59, 194, 176, 135, 34, 142, 143, 123, 134, 197, 74, 180, 25]. In terms of precision, those approaches can be classified as approximate [179] or exact [196]. Approximate algorithms usually take less execution time than exact ones. However, for large time series, an approximate algorithm can provide inaccurate results as they are based on probabilistic assumptions. Moreover, the user has to set several (even not intuitive) parameters, trying to get results accordingly to the expected ones, a fact which is not always possible. Taking the best of those two approaches, *anytime* algorithms incrementally improve the result over the execution time. This means that if the execution is interrupted before its completeness, a partial solution is obtained. In case the anytime algorithm is executed until the end, the exact solution is obtained. Based on that, we focus our attention on anytime algorithms.

The state-of-the-art exact and anytime set of tools for time series analysis is *matrix profile* [196]. We show a naive example of a matrix profile execution in Figure 2.2, where a sinusoidal time series (red one) presents some anomalies. The output of the algorithm is another time series (the green one, known as *profile*), which contains the distance (similarity) for every subsequence indexed by its first element with respect to the most similar neighbor of the rest of the time series.

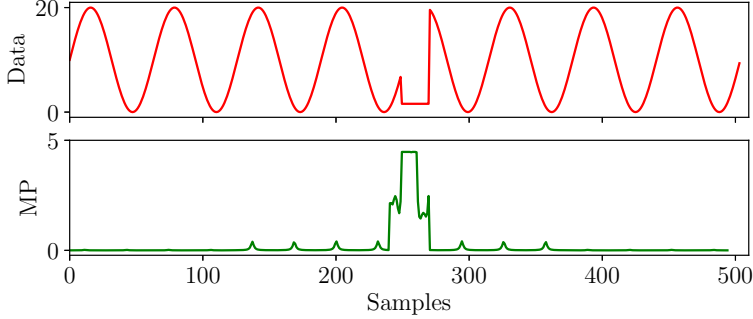


Figure 2.2: A time series with some anomalies (discords) and its matrix profile (MP). The anomalies appear as higher distance values in the profile.

The similarity between two subsequences  $T_{i,m}$  and  $T_{j,m}$  can be calculated using the *z-normalized Euclidean distance*, which is defined as follows:

$$d_{i,j} = \sqrt{2m \left( 1 - \frac{Q_{i,j} - m\mu_i\mu_j}{m\sigma_i\sigma_j} \right)} \quad (2.1)$$

where  $Q_{i,j}$  is the dot product of  $T_{i,m}$  and  $T_{j,m}$ ;  $\mu_x$  and  $\sigma_x$  are the mean and the standard deviation of the points in  $T_{x,m}$ , respectively. These statistics are calculated in  $O(n)$  time [151], taking negligible time of the whole computation.

Using the distance in Eq. 2.1 (or another one), the *matrix profile* algorithm solves the similarity search problem in three steps. First, it builds a symmetric  $(n - m + 1) \times (n - m + 1)$  matrix  $D$ , called *distance matrix*. Each cell in  $D$ ,  $d_{i,j}$ , stores the distance between two subsequences,  $T_{i,m}$  and  $T_{j,m}$ . Second, it creates an array  $P$  of size  $n - m + 1$ , called *profile*. Each cell  $P_i$  in  $P$  keeps the minimum distance recorded in the  $i^{th}$  row of  $D$ . Third, it allocates an array  $I$  that is of the same size as  $P$ , called *profile index*, such that  $I_i = j$  if  $P_i = d_{i,j}$ . This way,  $P$  contains the minimum distances between subsequences, while  $I$  is the vector of “pointers” to the location of these subsequences within the time series.

Fig. 2.3 depicts an example of the distance matrix  $D$ , the profile  $P$ , and the profile index  $I$ . The neighboring subsequences of  $T_{i,m}$  are highly similar to it (i.e.,  $d_{i,i+1} \approx 0$ ) due to overlapping between them. The algorithm excludes these subsequences from the computation to avoid false positives, by defining an exclusion zone (Figure 2.1) for each subsequence. It follows the approach in [202], where the exclusion zone of  $T_{i,m}$  is  $T_{i, \frac{m}{4}}$  (i.e., ends at  $t_{i+\frac{m}{4}}$  of the time series).

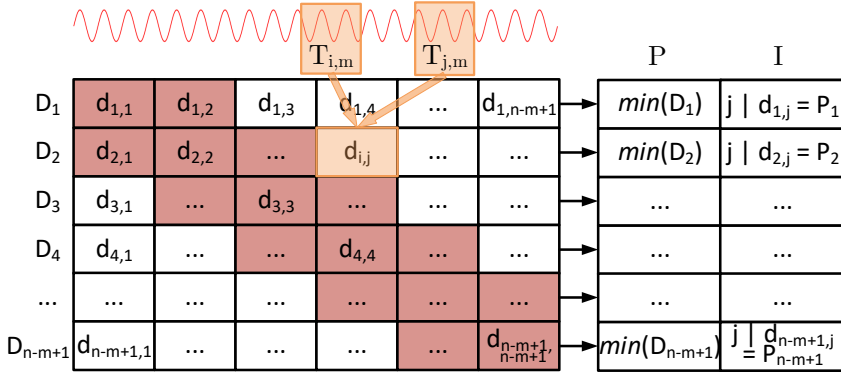


Figure 2.3: Example of distance matrix ( $D$ ), profile ( $P$ ), and profile index ( $I$ ).  $P_i$  holds the minimum distance calculated in row  $D_i$ , and  $I_i$  holds the index  $j$  of the subsequence that has such distance. Cells in the exclusion zone are coloured red.

There are several Euclidean-Distance-based implementations of *matrix profile*, including STAMP [196], STOMP [86], SCRIMP [202] and SCAMP [203]. SCRIMP is the state-of-the-art CPU-based implementation. Prior acceleration approaches to time series analysis [86, 202] mainly focus on accelerating STOMP

and PreSCRIMP [202] on GPUs. SCAMP [203] is a framework that combines a host (either a local machine or a server in a compute cluster) and workers that follow the directions from the host (either other CPUs in the cluster or accelerators such as GPUs). Recent proposals [13] have started to utilize Dynamic Time Warping (DTW)-based solutions because of higher precision. Figure 2.4 shows the key difference between the ED and DTW approaches, in which we compare two similar-shape subsequences that are offset with a fixed constant. It can be observed that the DTW algorithm offers better results as it compares a given point with respect to several potential candidates (i.e., determines the best alignment). In contrast, ED executes point-to-point alignment that cannot determine the best alignment in presence of an offset. ED can be considered as a special case of DTW where the *warping window* is set to one element.

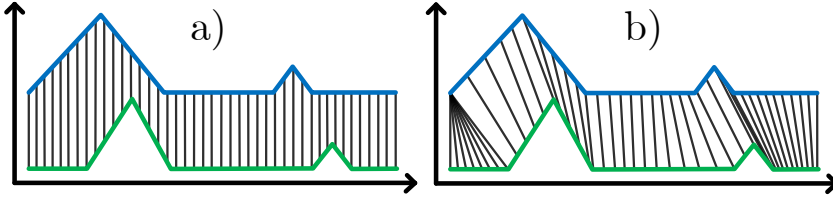


Figure 2.4: Example of similarity calculation between two subsequences (blue and green). The Euclidean approach in a) provides a low similarity as it only compares each  $i^{th}$  point of blue with each  $i^{th}$  point of green. However, DTW in b) successfully matches the points of the subsequences, thus finds the high similarity.

The time series analysis implementations covered so far are embarrassingly parallelizable, since the computation of different subsequence pairs is completely independent. In the next subsections, we provide some details about 1) SCRIMP, 2) SCAMP and 3) DTW.

### 2.1.1. SCRIMP Implementation

The state-of-the-art CPU-based implementation of the *matrix profile* algorithm is SCRIMP [202]. The key mechanism behind SCRIMP is that the dot product in Eq. 2.1 can be calculated incrementally in the diagonals of  $D$ :

$$Q_{i,j} = Q_{i-1,j-1} - t_{i-1}t_{j-1} + t_{i+m-1}t_{j+m-1} \quad (2.2)$$

This property enables a huge reduction of the computational cost of the algorithm, since it reduces the number of multiplications and additions needed.

According to Eq. 2.2, except for the first dot product, the remaining cells of a diagonal can be calculated using the values from the immediate upper left cells.

Algorithm 1, shows the pseudocode of a parallel version of SCRIMP. First, it precalculates the means and standard deviations of every subsequence of the time series (line 1), and initializes the profile vector (lines 2-3) to infinity. Second, it computes the diagonals in a parallel manner (see Fig. 2.3) using the loop in line 4. The variable **nDiag** is the number of diagonals of  $D$  assigned to each thread. These diagonals can be ordered in the **diag** vector (line 5) a) *randomly*, enabling the *anytime* property of the algorithm, or b) *sequentially*, discarding the *anytime* property but allowing for optimizations [202] (e.g., exploiting data locality of consecutive diagonals).

---

**Algorithm 1** SCRIMP

---

```

1:  $\mu, \sigma \leftarrow \text{precalculateMeansDevs}(T, m);$ 
2: for  $i \leftarrow 0$  to  $\text{size}(P) - 1$  do
3:    $P_i \leftarrow \infty;$ 
4: for  $\text{idx} \leftarrow \text{tid} * \text{nDiag}$  to  $(\text{tid} + 1) * \text{nDiag} - 1$  do
5:    $i \leftarrow 0; j \leftarrow \text{diag}_{\text{idx}};$ 
6:    $q \leftarrow \text{dotProduct}(T_{i,m}, T_{j,m});$ 
7:    $d \leftarrow \text{dist}(m, q, \mu_i, \sigma_i, \mu_j, \sigma_j);$ 
8:   if  $d < P_i$  then  $P_i \leftarrow d; I_i \leftarrow j;$ 
9:   if  $d < P_j$  then  $P_j \leftarrow d; I_j \leftarrow i;$ 
10:   $i \leftarrow i + 1;$ 
11:  for  $j \leftarrow \text{diag}_{\text{idx}} + 1$  to  $\text{size}(P)$  do
12:     $q \leftarrow q + t_{i+m-1+k}t_{j+m-1+k} - t_{i-1+k}t_{j-1+k};$ 
13:     $d \leftarrow \text{dist}(m, q, \mu_{i+k}, \sigma_{i+k}, \mu_{j+k}, \sigma_{j+k});$ 
14:    if  $d < P_i$  then  $P_i \leftarrow d; I_i \leftarrow j;$ 
15:    if  $d < P_j$  then  $P_j \leftarrow d; I_j \leftarrow i;$ 
16:     $i \leftarrow i + 1$ 

```

---

The size of the distance matrix  $D$  can be huge for large time series, so it is not convenient to store it in memory. For example, an earthquake sequence from a seismograph, consisting of 24,000 elements [185], needs approximately 1.3 GB of memory using double precision floating point representation. However, a 650,000 electrocardiogram (ECG) time series from the MIT-BIH arrhythmia database [133] requires about 850 GB of memory. Furthermore, we need to maintain the matrix profile  $P$ , the matrix profile index  $I$  and the time series  $T$ . For these reasons, matrix profile algorithms are designed to store only the profile, and the profile index arrays, and they compute the minimum  $d_{i,j}$  on the fly.

For each diagonal, the algorithm first computes the dot product of the first pair of subsequences in line 6 using the *dotProduct* function. Second, it calculates the distance according to Eq. 2.1 (line 7). Third, it checks and replaces the corresponding profile element with the new distance provided that the calculated one is smaller (lines 8-9). Then, the algorithm exploits the dot product update for the remaining elements in the diagonal with the following steps: 1) it adds the previous dot product to new calculated terms one (line 12); 2) it computes the new distance (line 13) and 3) performs the update if necessary (lines 14-15).

### 2.1.2. SCAMP Implementation

Whereas following a similar computation scheme than SCRIMP, SCAMP replaces the sliding dot product with a mean-centered-sum-of-products in order to reduce the floating-point rounding errors and the number of operations required [203]. The following equations can be precomputed in  $O(n - m + 1)$  time, with  $n - m + 1 = l$  being the length of the matrix profile vector:

$$df_i = \frac{T_{i+m-1} - T_{i-1}}{2}, \quad 0 < i < l \quad (2.3)$$

$$dg_i = T_{i+m-1} - \mu_i + T_{i-1} - \mu_{i-1}, \quad 0 < i < l \quad (2.4)$$

$$ssq_i = \begin{cases} \sum_{k=0}^{m-1} (T_k - \mu_0)^2, & i = 0 \\ ssq_{i-1} + (T_{i+m-1} - \mu_i + T_{i-1} - \mu_{i-1}) & \\ (T_{i+m-1} - T_{i-1}), & 0 < i < l \end{cases} \quad (2.5)$$

$$\sigma_i = \sqrt{ssq_i}, \quad 0 \leq i < l \quad (2.6)$$

Eqs. 2.3 and 2.4 are terms used in the covariance update of Eq. 2.7, and the standard deviation (L2-norm of subsequence  $T_{i,m} - \mu_i$ ) calculated in Eqs. 2.5 and 2.6 is used for the Pearson correlation coefficient depicted by Eq. 2.8. Note the exclusion zone in the limits of Eq. 2.7 given by  $\frac{m}{4}$ .

$$\sigma_{i,j} = \begin{cases} \sum_{k=0}^{m-1} (T_k - \mu_0)(T_{k+j} - \mu_j), & i = 0, \frac{m}{4} < j < l \\ \sigma_{i-1,j-1} + df_i dg_j + df_j dg_i, & i > 0, \frac{m+4}{4} < j < l \end{cases} \quad (2.7)$$

$$P_{i,j} = \frac{\sigma_{i,j}}{\sigma_i \sigma_j} \quad (2.8)$$

$$D_{i,j} = \sqrt{2m(1 - P_{i,j})} \quad (2.9)$$

The matrix profile can be derived incrementally for each diagonal of the distance matrix, Eq. 2.7, from the calculation of the covariance of two subsequences of the

first row (first piece in Eq. 2.7). The Pearson correlation coefficient in Eq. 2.8 can be computed in fewer operations and it is more robust than the Euclidean Distance used by SCRIMP. Eq. 2.9 calculates the distance from the Pearson coefficient in  $O(1)$ . Full derivations for Eq. 2.5 and Eq. 2.7 can be found in [203].

### 2.1.3. DTW Implementation

The Dynamic Time Warping (DTW) algorithm was first introduced by [26]. The key idea behind DTW is to compute the distance between a certain point in a subsequence and a set of points in another subsequence, considering the minimum distance found. This process is repeated for all the points of the first subsequence, and the sum of all distances provides a similarity measure between the subsequences. Let's assume that we have two time series subsequences,  $Q$  and  $R$ , of length  $n$  and  $m$ , respectively, where:

$$Q = q_1, q_2, \dots, q_i, \dots, q_n \quad (2.10)$$

$$R = r_1, r_2, \dots, r_j, \dots, r_m \quad (2.11)$$

To find how similar those two subsequences are (i.e., find the best alignment between them), DTW constructs an  $n$ -by- $m$  scoring matrix ( $S$ ). We show an example of this matrix in Figure 2.5 a. Each  $(i^{th}, j^{th})$  cell of the matrix is filled in two steps. First, the algorithm calculates the distance  $d(q_i, r_j)$  between the two corresponding points of the subsequences. There are several approaches to calculate such distance, while  $d(q_i, r_j) = \text{abs}(q_i - r_j)$  and  $d(q_i, r_j) = (q_i - r_j)^2$  are the most common ones [26]. Second, the distance value is added to the minimum of three neighboring cells, as follows:

$$s_{i,j} = d(q_i, r_j) + \min(s_{i-1,j-1}, s_{i-1,j}, s_{i,j-1}) \quad (2.12)$$

Once the matrix is computed using dynamic programming, the goal is to find the best alignment (i.e., minimum accumulated cost), known as the *warping path* ( $W$ ). The warping path is a contiguous set of matrix cells that defines the best mapping between  $Q$  and  $R$ , subject to some constraints [26]:

- *Boundary conditions*: the warping path must start and finish in diagonally opposite corner cells of the matrix.
- *Continuity*: the permissible steps in the warping path are restricted to adjacent cells.

- *Monotonicity*: the consecutive points in  $W$  must strictly increase with respect to the time they were collected.

Several DTW implementations try to reduce the computation cost and memory footprint by restricting the DTW score matrix calculation using different approaches, for instance, the Sakoe-Chiba band (Figure 2.5 b) and the Itakura Parallelogram (Figure 2.5 c). However, these approaches only apply to scenarios where the end-to-end sequences are quite similar and present small sequence compressions and attenuations (time warps). In other words, they do not fit well when the goal is to filter out irrelevant subsequences. In such scenarios, there are usually many small queries to compare against a longer reference (set of admissible patterns) and then detect the possible anomalies.

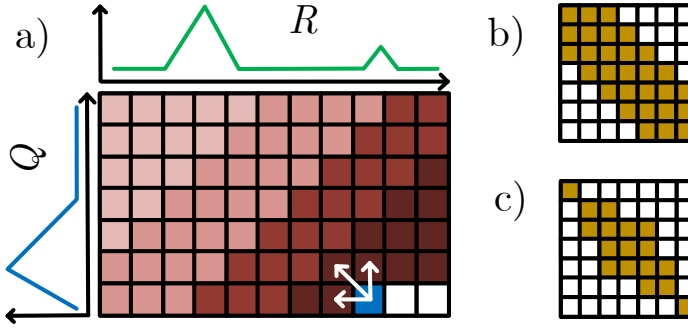


Figure 2.5: a) Warping matrix example for a reference time series  $R$  and a query subsequence  $Q$ . The DTW distance between  $R$  and  $Q$  is the minimum value of the last row of the matrix. b) Sakoe-Chiba band. c) Itakura parallelogram.

**Subsequence Dynamic Time Warping (sDTW).** sDTW is a slightly modified DTW algorithm that allows unbounded subsequence alignment. In the DTW domain, the unbounded alignments are often referred to as *open start* and *open end*, respectively. Using this approach, it is possible to align sequences of different lengths (e.g., a small query over a larger reference). Algorithm 2 presents the pseudocode of sDTW. First, it initializes the matrix  $S$  with zeros. Second, it calculates the distance value of the top-left corner and then the remaining elements of the first row, taking into account the previous values. Third, it fills the remaining elements of the matrix using dynamic programming row by row. Finally, it returns the minimum element of the last row of the  $S$  matrix, which indicates the similarity between the query and the best alignment with (part of) the reference.



**Algorithm 2** Subsequence DTW (sDTW)

---

```

1: procedure sDTW(Q,R)
2:    $S \leftarrow \text{zeros}(N, M)$ ;
3:    $S[0, 0] = \text{abs}(Q[0] - R[0])$ ;
4:   for  $i \leftarrow 1$  to  $N$  do
5:      $S[i, 0] \leftarrow S[i - 1, 0] + \text{abs}(Q[i] - R[0])$ ;
6:   for  $i \leftarrow 1$  to  $N$  do
7:     for  $j \leftarrow 1$  to  $M$  do
8:        $S[i, j] \leftarrow \text{abs}(Q[i] - R[j]) +$ 
9:          $\min(S[i - 1, j - 1], S[i, j - 1], S[i - 1, j])$ ;
10:  return  $\min(S[N, :])$ 

```

---

## 2.2. Mitigating the Data Movement Bottleneck

Processing-Near-Memory and Processing-Using-Memory are two promising hardware architectural paradigms to mitigate the data movement bottleneck. In the first one, compute units are placed close to memory arrays, while in the latter one compute units are the actual memory arrays. We discuss some details about them in the following subsections.

### 2.2.1. Processing-Near-Memory

Processing-Near-Memory (PNM) is a promising paradigm to reduce the data movement between CPUs and memory by placing simple general-purpose processors [84, 11, 28] or application-specific accelerators [85, 199, 106, 33, 28, 12] in or close to the logic layer of 3D-stacked memory. Generally, PNM can provide performance benefits for memory-bound applications when they exhibit one or more of the following major properties: 1) requiring higher memory bandwidth than available in the system, 2) being sensitive to memory access latency [138], or 3) performing irregular memory accesses, such that they cannot effectively benefit from cache hierarchy of conventional CPU architectures.

Recent advances in die-stacking technologies have enabled the integration of multiple layers of DRAM arrays in a single package. A 3D-stacked memory consists of several memory dies, one on top of each other, connected using Through-Silicon Vias (TSV) [114, 115]. PNM locates low-power processing units inside the logic layer of 3D-stacked memory, to harness the significantly higher bandwidth and the lower latency provided while consuming less energy. The most prominent

3D-stacked memory technologies are High Bandwidth Memory (HBM) [94] and Hybrid Memory Cube (HMC) [78], but there are several others [107, 67]. Figure 2.6 shows a typical PNM architecture where compute units are placed close to an HBM cube, exploiting the lower latency and higher bandwidth.

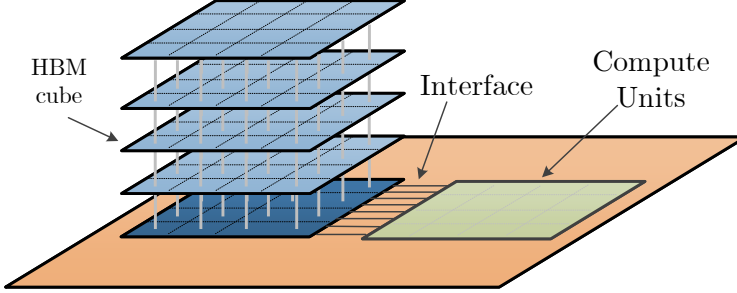


Figure 2.6: Sample PNM architecture (I). Compute Units are placed close to the HBM cube, connected via a short and wide interface.

Processing-Near-Memory can also happen inside the memory chips. The first commercially-available example is UPMEM [183] (Figure 2.7), where custom general-purpose cores are placed near memory banks.

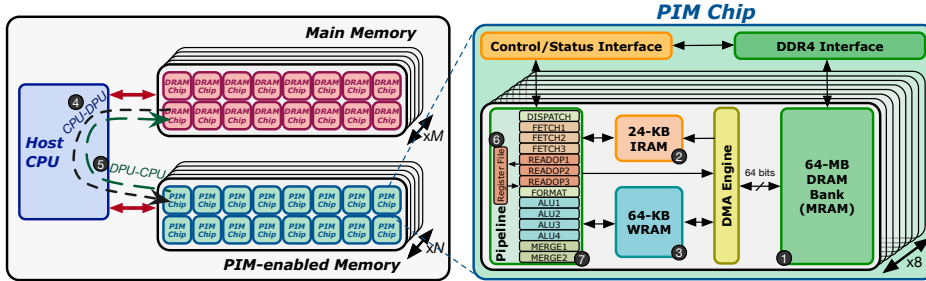


Figure 2.7: Sample PNM architecture (II). UPMEM-based PNM system with a host CPU, standard main memory, and PNM-enabled memory (left), and internal components of a UPMEM PNM chip (right). Figure taken from [183].

Inside each UPMEM PIM chip (Figure 2.7 (right)), there are 8 DPUs. Each DPU has exclusive access to (1) a 64-MB DRAM bank, called *Main RAM* (MRAM) ①, (2) a 24-KB instruction memory, called *Instruction RAM* (IRAM) ②, and (3) a 64-KB scratchpad memory, called *Working RAM* (WRAM) ③. MRAM is accessible by the host CPU (Figure 2.7 (left)) for *copying* input data (from main memory to MRAM) ④ and *retrieving* results (from MRAM to main

memory) ⑤. These CPU-DPU and DPU-CPU data transfers can be performed in parallel (i.e., concurrently across multiple MRAM banks), if the buffers transferred from/to all MRAM banks are of the same size. Otherwise, the data transfers happen serially (i.e., a transfer from/to another MRAM bank starts after the transfer from/to an MRAM bank completes). There is no support for direct communication between DPUs. All inter-DPU communication takes place through the host CPU by *retrieving* results from the DPU to the CPU and *copying* data from the CPU to the DPU.

There is a vast literature [136, 66, 65, 85, 187, 10, 23, 38, 81, 103, 125, 160, 12, 159, 62, 49, 11, 63, 116, 28, 29, 30, 84, 106, 33, 80, 156, 121, 164, 149, 158, 163, 80, 157, 150, 36, 104, 105] that follows the PNM approach to overcome the memory wall. In [49], ARM cores are used as PNM compute units to improve data analytics operators (e.g., group, join, sort). IMPICA [85] is a PNM pointer chasing accelerator. Tesseract [11] is a scalable PNM accelerator for parallel graph processing. TETRIS [63] is a PNM neural network accelerator. Lee et al. [116] propose a PNM accelerator for similarity search. GRIM-Filter [106] is a PNM accelerator for pre-alignment filtering [192, 18, 191, 20, 19] in genome analysis [17]. Boroumand et al. [28] analyze the energy and performance impact of data movement for several widely-used Google consumer workloads, providing PNM accelerators for them. CoNDA [29] provides efficient cache coherence support for PNM accelerators. Finally, a PNM architecture [77] has been proposed for MapReduce-style applications.

### 2.2.2. Processing-Using-Memory

Processing-Using-Memory architectures have been proposed to overcome the memory wall challenges in current von-Neumann architectures [136], as they are able to perform computation where data resides. Many prior works demonstrate significant performance and energy efficiency improvements for machine learning workloads using crossbars [132] by exploiting matrix-vector multiplication. Other application domains can exploit bitwise operations [127, 110] to obtain similar benefits. Figure 2.8 a shows a typical crossbar organization with memory cells connected using bitlines and wordlines.

**Bitwise PUM Mechanism.** Applications that are not based on matrix-vector multiplications (e.g., sDTW) require a flexible substrate to support the required operations. Moreover, off-the-shelf crossbar substrates offer limited support for other operations. To overcome this challenge, MAGIC [127, 110] proposes to decompose complex operations into simple Boolean functions (e.g., AND,

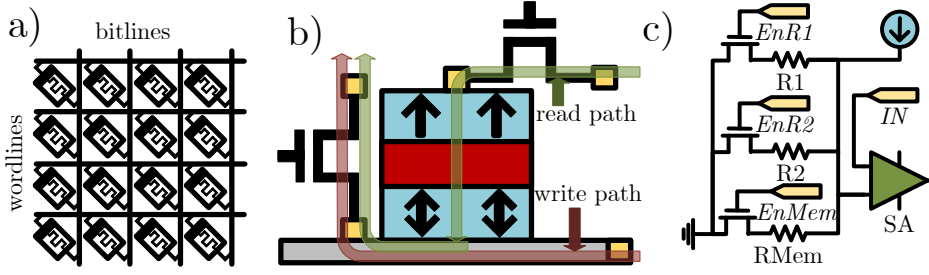


Figure 2.8: a) Crossbar organization. b) Magneto-resistive cell. c) Reconfigurable SA that performs in-memory operations based on the voltage across the bitline when two cells are activated.

NOR, XOR) and make them supported in the substrate. The key idea is to vertically map the operands (e.g., 32-bit integers) to the crossbars' columns using (typically) one bit per cell (e.g., each operand value takes 32 bits of a given column). Then, the desired operation (e.g., addition) is decomposed on simple bitwise ones (e.g., NOR) and performed bit by bit by sequentially activating one cell of each operand at the same time. This provokes a difference in the voltage over the bitline depending on the content of the activated cells. Then, a modified sense amplifier generates the result based on that voltage difference and thresholds, storing it in a cell of the same column. While this process is inherently sequential (i.e., the latency is higher than a CMOS-based approach), the 1) independence across columns and 2) the lack of data movement enable an unprecedented parallelism and thus an overall higher throughput. Figure 2.8 c shows an example of a modified sense amplifier (SA) with different voltage thresholds and the *regular* memory sensing threshold. While the throughput of performing a single operation is lower than a CMOS-based solution, this approach enables enormous parallelism as 1) all columns are working independently at the same time and 2) data movement is eliminated.

**Technology.** The selection of the cell technology is crucial to ensure feasibility of bitwise based accelerators. While it is possible to build them using DRAM technology (e.g., SIMDRAM [79]), performance scaling is limited by 1) refreshing times and specially by 2) the need of moving data to compute enabled rows. Crossbar-based NVM technologies can overcome those two issues, but it is challenging to support frequent write operations. This is because NVM-based architectures usually suffer from significant latency and energy penalties, and specially low endurance. However, emerging NVM technologies are a possible solution to overcome those drawbacks. Table 2.1 presents NVM characteristics.

Technology	Write/Read Energy (per bit)	Write/Read Time (per bit)	Endurance (Write Cycles)
NAND Flash [73]	470pJ / 46pJ	200 $\mu$ s / 25.2 $\mu$ s	10 <sup>5</sup>
FRAM [177]	1.4nJ / 1.4nJ	120ns / 120ns	10 <sup>15</sup>
STT-MRAM [39]	2nJ / 34pJ	250ns / 10ns	10 <sup>5</sup>
SOT-MRAM [22]	334pJ / 247pJ	1.4ns / 1.1ns	> 10 <sup>15</sup>
ReRAM [91]	1.1nJ / 525fJ	10 $\mu$ s / 5ns	10 <sup>5</sup>
PCM [113]	13.5pJ / 2pJ	150ns / 48ns	10 <sup>7</sup>

Table 2.1: Overview of different NVM technologies [45].

To help in the decision of what cell technology use for a potential accelerator, both endurance and latencies have to be taken into consideration. In terms of endurance, we note that both FRAM and SOT-MRAM are good candidates. However, if we add latencies to the equation, we observe that SOT-MRAM is the most promising one, as FRAM latencies are two orders of magnitude larger than the first one. Magneto-Resistive RAM (MRAM)-based substrates (e.g., SOT-MRAM) offer 1) higher density, 2) high endurance, 3) low latencies, and 4) they are CMOS compatible [72, 188, 117, 21, 87]. Based on that, MRAM technologies are promising architectural candidates for building PUM accelerators [193] where data changes frequently. Figure 2.8 b shows the basic structure of a Spin-Orbit Torque (SOT)-MRAM device, composed of a stack of Magnetic Tunnel Junctions (MTJs). The key mechanism is based on the change of orientation of one of the layers of the stack, which results in a variation in the device's electrical resistance. Several companies, like Samsung [96], are raising their interest in these architectures, and real devices might be commercially available shortly.

## 2.3. Transprecision Computing

Transprecision Computing aims to boost energy efficiency and performance by exploiting numeric truncation in both hardware and software. It enables fine control over the precision of floating-point arithmetic in space and time (where and when to use it). The key difference with approximate computing is that transprecision guarantees the error (as the numeric precision and range are known for a given configuration) while approximation provides uncertainty. One example of this uncertainty occurs when reducing the refresh interval of RAM to improve performance [95], since process variation makes the exact error to be unknown. Based on that, transprecision leads to significant energy savings and performance improvements without sacrificing overall quality of results.

Transprecision computing can be applied to the entire algorithm by setting fixed exponent and mantissa widths for every floating-point operation. However, it is possible to change the precision to different parts of the code to find a trade-off between the accuracy of the results and energy efficiency. Mixed precision of double and single floating-point operations has been successfully used in the past [32, 122] with a significant gain in performance.

Besides the IEEE-754 standard, transprecision computing allows the use of arbitrary exponent and mantissa bit combinations. However, the design of arbitrary precision floating-point units (FPUs) can be challenging and presents difficulties when integrating in computing platforms. Because of this reason, already designed FPUs typically support a fixed number of exponent and mantissa combinations. As an example, Fig. 2.9 shows the types [173] that we use in TraTSA for the energy evaluation.

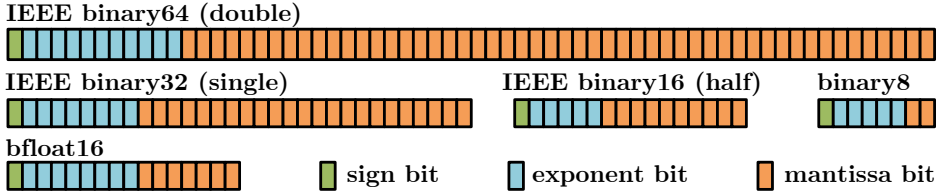


Figure 2.9: Overview of the floating-point types used for energy evaluation.

Table 2.2 summarizes the bit count for each floating-point datatype, along with the approximate range and smallest number that can be represented. Notice that, while *binary32* and *binary16* have approximately the same range, the first one provides narrower steps between numbers and more precision.

Table 2.2: Floating-point bit counts, ranges and smallest numbers

Type	Exp	Man	Range ( $\approx$ )	Smallest ( $\approx$ )
bin64	11	52	$\{-1.8 \cdot 10^{308}, 1.8 \cdot 10^{308}\}$	$2.2 \cdot 10^{-308}$
bin32	8	23	$\{-3.4 \cdot 10^{38}, 3.4 \cdot 10^{38}\}$	$1.2 \cdot 10^{-38}$
bin16	5	10	$\{-65504, 65504\}$	$6.1 \cdot 10^{-5}$
bfloat16	8	7	$\{-3.4 \cdot 10^{38}, 3.4 \cdot 10^{38}\}$	$1.2 \cdot 10^{-38}$
bin8	5	2	$\{-57344, 57344\}$	$6.1 \cdot 10^{-5}$

We find transprecision Floating-Point Units (FPU) already proposed in the literature, which aim to take advantage of transprecision computing in terms of energy, performance and area. The first silicon implementation of a 64-bit

transprecision FPU can be found in [129], using 22nm technology node. This FPU supports the floating-point types depicted in Fig. 2.9, and the key idea behind it is to operate in scalars (64-bit) or in SIMD vectors of 2 elements (32-bit), 4 elements (16-bit) or 8 elements (8-bit). The authors of such FPU evaluate it integrated into a RISC-V core via simulation (but can be used in FPGAs or ASICs) and obtain speedups up to  $7.3\times$  while reducing energy up to  $7.94\times$  with respect to *double* precision approaches. The source code of this FPU (known as FPnew) can be found in [8].

In contrast, transprecision libraries aim to enable transprecision software emulation in commodity architectures. An example of CPU-based transprecision emulation is FlexFloat [172], which is written in C. The main advantage of this library is that it can be executed in almost any commodity platform, but it presents a main drawback: the significant overhead introduced by the software emulation. In this sense, we observe that the FlexFloat execution time increases the native IEEE double execution time by  $200\times$  for a given computing platform. One way to overcome this issue is the use of an FPGA architecture, including custom-precision units from cfp-FPGA [47] library, which is intended for High-Level Synthesis (HLS). While not being a pure transprecision implementation from the architectural point of view (e.g., data is stored using 32 bits in memory and then converted to the desired width), it allows time-manageable evaluation of large time series (millions of elements). The key motivation to consider the FPGA acceleration is to avoid the emulation cost of the software-based one with the use of dedicated transprecision hardware. Then, once evaluated the potentials of a given algorithm using the FPGA, it is possible to design precision-tuned custom accelerators for a given application (e.g., TSA) and exploit the reduction in area, energy and execution time.



UNIVERSIDAD  
DE MÁLAGA



# 3 **PhiTSA: Using a Xeon Phi to Optimize and Characterize TSA**

---

## 3.1. Motivation and Key Idea

In this chapter we present PhiTSA, an optimized implementation of the SCRIMP algorithm tailored for Intel Xeon Phi Knights Landing (KNL) processors. This work is motivated by the opportunity that these architectures provide in terms of parallelism, vectorization and different memory technologies. Concretely, KNL includes two types of memory (i.e., DDR4 and HBM). This is useful to 1) study the impact of memory bandwidth in performance and 2) take advantage of the aggregated bandwidth. The key idea is to implement a privatization-based parallel and vectorized version of SCRIMP. Then, we use this optimized implementation to characterize the behaviour using Intel profiling tools and find its bottlenecks.

## 3.2. Intel Xeon Phi Knights Landing

The Intel Xeon Phi based on the Knights Landing architecture (KNL) includes 64 Airmont cores (Atom) and up to 256 hardware threads (4 threads per core). Figure 3.1 presents an overview of the platform. This processor features 16GB HBM integrated in the CPU chip, providing a much higher bandwidth

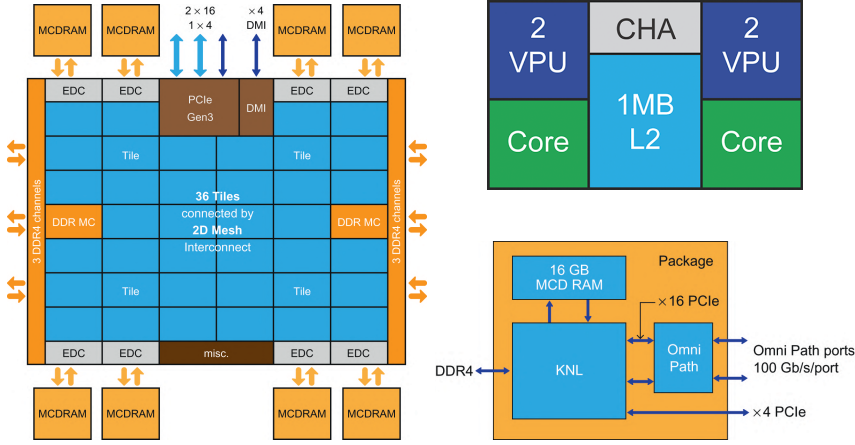


Figure 3.1: Intel Xeon Phi KNL overview [93].

than the external DDR4 modules. The HBM is based on Multi-Channel DRAM (MCDRAM) and consists of four memory stacks with an aggregated peak bandwidth of more than 450GB/s. The DDR4 memory, whose capacity is 192GB, provides 6 channels with a peak raw bandwidth of 80GB/s.

The MCDRAM can be configured in three modes: 1) Flat, 2) Cache or 3) Hybrid. The mode has to be selected at boot time. If the Flat mode is enabled, the MCDRAM acts as a separate addressable space. In the Cache mode, the MCDRAM is a transparent last-level cache. By setting the Hybrid mode, we can take advantage of both modes. Hybrid mode allows us to use part of the MCDRAM as addressable memory and the rest as cache. In this work, we select the Flat mode, as our goal is to find a suitable data mapping using both the MCDRAM and the DDR4 memories, and consequently aggregating bandwidth and size of both memory spaces.

The cores are bonded in pairs conforming 32 tiles. Both cores in each tile shares an L2-cache of 1MB. Tiles can be clustered in several NUMA configurations. In this work, we use the Quadrant cluster, in which the array of tiles is organized in four quadrants, each one connected to a MCDRAM controller. This configuration reduces the L2-cache miss latency and the four blocks of MCDRAM appear as contiguous memory spaces.

In terms of parallelism, KNL provides two approaches: 1) task/thread parallelism between the cores, and 2) data parallelism using the AVX-512 SIMD extensions. These extensions allow computing 16 single or 8 double precision

floating-point operations at the same time by means of two out-of-order vector processing units (VPUs) that are available per tile. Based on that, the theoretical peak performance of the KNL is 6 TFLOP/s in single precision and 3 TFLOP/s in double precision.

### 3.3. PhiTSA Optimizations of SCRIMP

The first part of this work comprises the optimization of SCRIMP to exploit the potential of the KNL platform as much as possible. We begin by analyzing the SCRIMP algorithm (described in Section 2.1.1) from the parallelism point of view. SCRIMP calculates distances between subsequences following diagonals of the matrix, which we show in Figure 3.2. This is explained because of two reasons: 1) elements in a diagonal can be computed from previous elements of the same diagonal with minimal calculations, and 2) calculating diagonals in random order enables the possibility of getting an accurate partial result of the Matrix Profile.

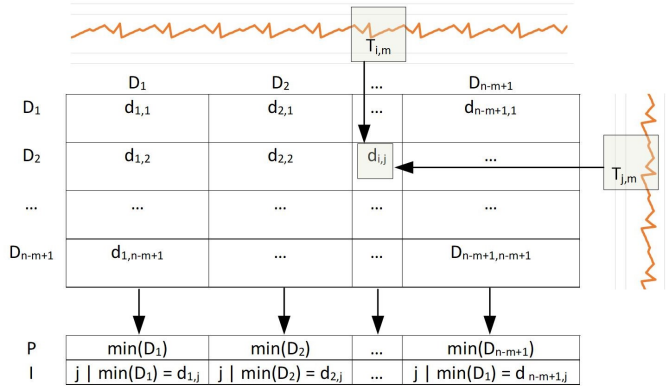


Figure 3.2: Calculation of the Matrix Profile  $P$  and the Matrix Profile index  $I$  from the distance matrix. For each column (or row)  $D_i$  we calculate the minimum distance and the index of the subsequence providing such minimum distance.

We find that SCRIMP is highly parallelizable because the computation of the different diagonals can be carried out independently. The simplest way for speeding up the algorithm in a many-core system is distributing the diagonals among the executing threads. However, as diagonals have different length, such assignment may lead to workload imbalance. We deal with this situation creating a pool of diagonals to be processed, which consists in a list of indexes and the

size depends on the time series length. From the thread scheduling point of view, it consumes 1% of the execution time. Every time a computing thread is idle, it retrieves a diagonal from this pool (task queue) and processes it. This way, threads are busy most of the time. We reduce the imbalance to those diagonals at a minimum (i.e., this only happens at the end of the computation when few of them remains to be processed).

When processing a diagonal, a thread calculates distances between subsequences and updates the Matrix Profile  $P$  and the Matrix Profile index  $I$ . Note that both  $P$  and  $I$  are to be calculated per column. This way, it could be possible that two threads are computing concurrently two different distances in two different diagonals, but with such distances belonging to the same column (i.e. the same element in the profile and profile index). Thus, both threads may try to update the same element in  $P$  and  $I$  simultaneously which may lead in inconsistencies if not managed properly (write-after-read data dependency). To deal with these potential data conflicts, we consider two solutions: lock-based mutual exclusion, and data privatization. We discuss them in the next subsection.

### 3.3.1. Updating $P$ and $I$

We find that the two more promising approaches to solve potential concurrent updates to the  $P$  and  $I$  data structures are 1) lock-based mutual exclusion and 2) privatizing the critical data. The performance of the first approach may be impacted strongly by the synchronization pressure. On the other hand, although the synchronization effort is negligible for privatization, it requires an extra amount of memory. We analyze both possibilities in more detail next.

**Atomics.** We first develop an implementation based on *atomics* as a fine-grained lock-based<sup>1</sup> solution. In this way, once a thread has calculated the distance  $d_{i,j}$ , we acquire two locks before updating the positions  $P[i]$ ,  $P[j]$ ,  $I[i]$ , and  $I[j]$ . As we show in Figure 3.3, if a thread needs to update the Matrix Profile  $P$  and the Matrix Profile index  $I$  (lines 12–15) with a new distance for a pair of subsequences, it has to acquire the two locks (lines 4–11), one for the first position and the other for the second position, releasing it after the update occurs (lines 16–17). In case both locks cannot be acquired, they are released (line 9) to prevent deadlocks. Note that this solution may lead to livelocks, but we rely on the fairness of the OS scheduler to prevent this situation.

<sup>1</sup>Our implementation uses the functions `test_and_set()` and `clear()` from `std::atomic` library available since C++11. We declare an array of type `std::atomic_flag`, corresponding to each position of the Matrix Profile.

```

void UpdateProfile(distance, i, j)
{
    int ack = 0;
    while(!ack) {
        if(!locks[j].test_and_set()) {
            if(!locks[i].test_and_set())
                ack = 1;
            else
                locks[j].clear();
        }
    }
    if (distance < profile[j])
        profile[j] = distance; index[j] = i
    if (distance < profile[i])
        profile[i] = distance; index[i] = j
    locks[j].clear();
    locks[i].clear();
}

```

Figure 3.3: SCRIMP's lock-based implementation to update the Matrix Profile and the Matrix Profile index without potential conflicts.

**Profile and index privatization.** Second, we develop an implementation based on privatizing the accesses to the shared data structures. Our goal is to avoid the use of mutual exclusion methods, looking for performance boosting. In this case, the use of synchronization between threads is minimal. This is achieved by expanding [54] the Matrix Profile, creating a replica (row) per thread as depicted in Figure 3.4. This is computationally safe as the performed operation per iteration is commutative and associative, i.e., it is a reduction loop [76].

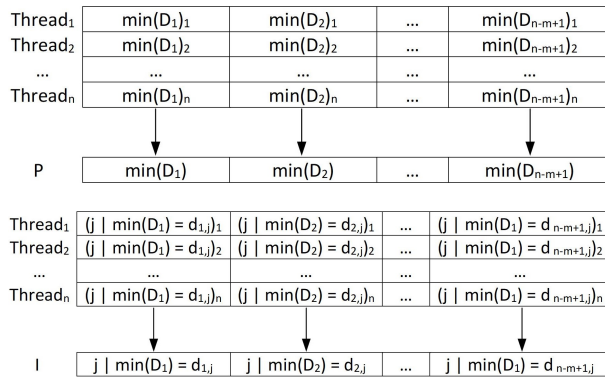


Figure 3.4: SCRIMP's expanded structure for Matrix Profile distances (top) and indexes (bottom) in the privatization technique.

Although shared, each replica can be seen as a private storage during the calculation of the Matrix Profile, in such a way that each thread computes its private Matrix Profile, storing the computed partial results in it. The only imperative synchronization is a barrier to wait for all threads to finish its private computation. Once all threads have calculated their private Matrix Profiles, a final reduction operation (per column) is carried out to compute the final results. No synchronization is necessary in this stage because each thread is in charge of a different column, so it can be done fully in parallel. Note that an analogous approach is applicable to the Matrix Profile index, as shown in Figure 3.4. The expansion involves to declare two matrices that can be accessed by threads: private profile and profile index, as the code in Figure 3.5 shows.

Each thread updates the new distances in the profile corresponding to a row based on their thread identifier (lines 16–24), and the final reduction stage is performed in parallel with no synchronization (lines 27–37). Overall, this technique increases the memory footprint linearly with the number of threads that we define for the computation. However, it enables to increase performance significantly.

### 3.3.2. Increasing Arithmetic Intensity

Our analysis using Intel profiling tools warn about the low arithmetic intensity exhibited by the SCRIMP algorithm. We tackle this by: 1) incrementing the arithmetic intensity of the algorithm via vectorization, and 2) selecting the most appropriate memory space (i.e., HBM or DDR4) for each variable in order to benefit from the aggregated bandwidth offered by the KNL architecture. In this sense, we find that the work of Khaldi et al. [101] explores the use of HBM and DDR4 memories from an automatic perspective depending on the application. However, our proposal focuses on the use of these memory spaces to solve memory-bounded problems, such as the Matrix Profile, profiling which data structures and variables fits better in each memory space.

Additionally to the thread parallelization of the algorithm, it is possible to reduce the execution time even more taking advantage of the SIMD support available in the KNL platform. For this purpose, several distance calculations and updates have to be packed into groups instead of calculating and updating them individually. In particular, we optimize the dot product calculation  $Q_{i,j}$  of Equation (2.1). It must be taken into account that there are data dependencies between the calculation of consecutive elements of a single diagonal. Consequently, this part of the computation is carried out in a non-vectorized unrolled loop, and later used to calculate the non-dependent part of the computation that

```

/* Expanding privatized structures */
double profile_exp[nThreads][ProfileLength];
int profileIndex_exp[nThreads][ProfileLength];
void SCRIMP() {
    #pragma omp for schedule(dynamic)
    for (j=0; j<numDiags; j++) {
        for (i=j; i<diagLength; i++) {
            /* distance is calculated here */
            UpdateProfile(distance,i,j,threadID);
        }
    }
    #pragma omp barrier
    FinalReduction();
}
void UpdateProfile(distance, i, j, threadID) {
    id = threadID;
    if (distance < profile_exp[id][j]) {
        profile_exp[id][j] = distance;
        profileIndex_exp[id][j] = i;
    }
    if (distance < profile_exp[id][i]) {
        profile_exp[id][i] = distance;
        profileIndex_exp[id][i] = j;
    }
}
void FinalReduction() {
    #pragma omp for schedule(static)
    for (col=0; col<ProfileLength; col++) {
        for (row=0; row<numThreads; row++) {
            if (profile_exp[row][col] < min_distance) {
                min_distance = profile_exp[row][col];
                min_index=profileIndex_exp[row][col];
            }
        }
        profile[col] = min_distance;
        profileIndex[col] = min_index;
    }
}

```

Figure 3.5: Privatization-based implementation of the SCRIMP computation.

can be vectorized inside another loop.

Figure 3.6 shows how we perform SCRIMP's computation in a vectorized manner. First we pre-calculate the component of the value with no data dependencies (lines 3–6) in a vectorized loop. After that, we can update the values, taking account of the dependencies and using an unrolled loop (lines 8–11). Finally, we obtain distance values using the values previously pre-calculated, again in a vectorized loop (lines 13–17). Note that the constant *ARIT\_FACT* is architecture-dependent and it guides the compiler when generating the machine

```

while(j < (ProfileLength - ARIT_FACT))
{
    #pragma omp simd
    for(int k=0; k<ARIT_FACT; k++) {
        Q[k] = /* Q value based on j */;
    }

    #pragma unroll (ARIT_FACT - 1)
    for(int k=1; k<ARIT_FACT; k++) {
        Q[k] += Q[k-1];
    }

    #pragma omp simd
    for(int k=0; k<ARIT_FACT; k++) {
        /*  $\xi$  = some calculations based on j */
        distances[k] = Q[k] +  $\xi$ ;
    }
    j+=ARIT_FACT;
}

```

Figure 3.6: SCRIMP’s inner loops unrolling and vectorization.

code. In our case, this constant is set to eight, in order to take advantage of the 512-bit SIMD instructions (eight 64-bit double precision floating-point numbers stored in a single vector register).

### 3.3.3. Memory Allocation Policy and Scalability

In order to gain advantage of the maximum memory bandwidth available (HBM plus DDR4), our approach allocates the most frequently accessed variables in the HBM space, where the highest bandwidth is available. Specifically, HBM stores the computed average and standard deviation for each subsequence (parameters  $\mu$  and  $\sigma$  in Equation (2.1)), as well as the privatized profile and profile indexes structures. Note that the code changes required to allocate HBM memory are minimal: Intel’s KNL libraries provides the *hbw\_malloc* functions to allocate HBM, while a regular *malloc* allocates DDR4 memory. However, the original time series and the final profile and profile indexes are allocated in the DDR4 memory.

Using this approach, the DDR4 memory serves the time series to the threads while the calculation of the diagonals is performed (only reads); and after that, the DDR4 is used again to keep the final results obtained in the final reduction stage. In this way, both the HBM and DDR4 bandwidths are aggregated, leading to an efficient memory exploitation, and saving space in the HBM. This



approach becomes more relevant in very large time series, because for shorter ones there is no significant improvement from the execution time point of view, and there is no need for saving space in the HBM space, being enough for those time series. Moreover, when processing very large time series, where the privatized structure of the whole series does not fit in the HBM space, we need to process the time series in several chunks of diagonals. In this sense, we take advantage of the DDR4 bandwidth and space for reading the original time series and storing the final result that is going to be updated after every chunk is processed.

## 3.4. Evaluation

### 3.4.1. Methodology

We conduct our experiments in an Intel Xeon Phi 7210, which is installed in a SuperMicro Superserver 5038K-i [7]. This architecture features a combination of both HBM and DDR4 memory technologies, allowing programmers to allocate data in any of them as required [166].

We compile the codes with the Intel C++ compiler version 18.0.2, enabling the flag `-O3` for the highest optimization and the flag `-xHost` to generate code with the widest available instruction set (in this case, the AVX-512, when possible). The results below correspond to the average of ten executions.

### 3.4.2. Results

#### Speedups

In this subsection, we present the experimental evaluation of our proposals. First, we perform experiments related to speedup and memory bandwidth usage with fixed time series length and fixed window size, using the different parallel implementations described in Section 3.3. After that, we measure the memory bandwidth usage varying the number of threads in the implementation that brought the best results from the performance point of view. Finally, we execute our private-structure based implementation varying time series length, window size and memory allocation policies allowing a comparison with previous works.

With respect to the speedup, we initially test our SCRIMP implementation based on software-atomics. In this case, our goal is to study the scalability with the number of threads. We calculate speedup with respect to the original

sequential version. Taking account of the Matrix Profile calculations, the nature of the data does not affect the execution times, so there is no need to particularize for a specific time series. In this sense, we use a random time series of  $2^{17}$  (131072) elements, and a fixed window size of 1024. Regarding the allocation of the data, we use only the DDR4 space, because using the HBM memory in this approach does not improve the performance of the execution. The explanation to this is that the memory bandwidth usage of this implementation is very low in comparison to the available bandwidth, as we show later.

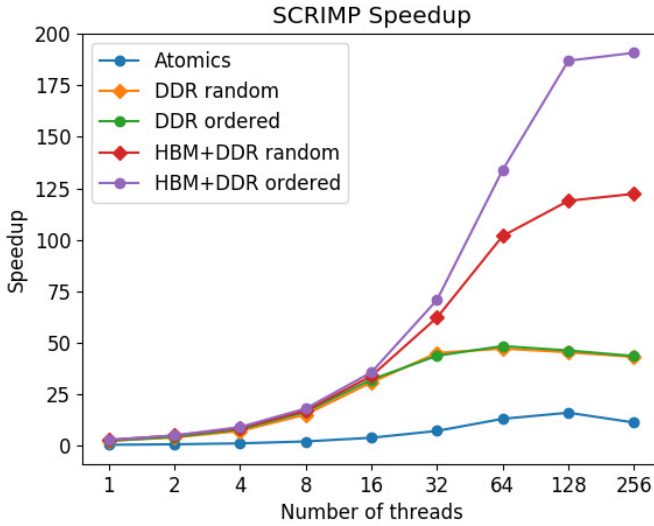


Figure 3.7: SCRIMP speedup in the Intel Xeon Phi KNL, calculated using a random time series of  $2^{17}$  (131072) elements and a fixed window size of 1024.

Results of this experiment correspond with the lowest plot in Figure 3.7. As expected, this implementation is far away from the ideal speedup. The maximum speedup with this approach is only about 16 times faster than the original and sequential version using 128 threads. Also, for obtaining similar execution times than the sequential version, the atomic-based implementation needs 4 threads. The explanation to this behaviour is that the atomic accesses to the common profile structure slows down the thread because such atomics are implemented in software. Also, this approach leads to a huge amount of cache traffic and waiting time for the threads, which also contribute to slow down the execution.

The second experiment tests our implementation based on privatizing the

calculation of the Matrix Profile. As in the previous experiment, we use a random time series of  $2^{17}$  elements and a fixed window size of 1024.

In this case, we evaluate four configurations. On the one hand, we test either the combination of the HBM plus the DDR4 memory, or using the DDR4 memory only. On the other hand, when computing the diagonals, we consider a random order or the sequential one.

We present the speedup of these four configurations in Figure 3.7. Setting the random order gives the advantage of the anytime property, which allows the user to stop the computation in any part of it, having a partial (not exact) result. The drawback of this approach is that it is expected a lower reuse of data in caches, and that explains why computing diagonals in the sequential order results in a better performance.

Furthermore, we can notice that both of the DDR4 cases (random and sequential order) obtain similar performance, growing significantly up to 16 threads and even starting to decrease from 64 threads. This is a scenario of over-threading, where increasing the non-payload part of the computation (more threads asking for memory requests) brings lower performance. Thus, the DDR4 bandwidth is not enough to serve the threads the data they need. In contrast, the HBM cases keep improving with the number of threads, and from 16 threads they start to be far from the DDR4 ones. Also, we have to take into account that the number of physical cores in the target platform is 64, and an amount of threads beyond this number imply the use of hyperthreading. This fact results in a lower performance increment if compared with the case of a thread per actual core. Finally, note that our single-thread execution is faster than the sequential implementation from the authors of Matrix Profile (concretely, 2.6x faster), due to the increase in the arithmetic intensity through taking advantage of vectorization (the use of AVX-512 instructions) in most of the computations, as explained in section 3.3.2.

Finally, we compare our implementation with the GPU-STOMP implementation from authors of Matrix Profile [86]. We present the results in Table 3.1, where we define a fixed window size of 256 elements and the same data type than the GPU implementation (single precision). We compute diagonals in order by 256 threads and the implementation uses the HBM space. Note that the speedup of SCRIMP in the Intel Xeon Phi tends to be lower with respect to GPU-STOMP in larger series than in shorter ones. This is explained by the relationship between the number of diagonals to compute and the number of processing elements. The GPU provides higher data parallelization lanes than the Xeon Phi, thus for large datasets it is desirable to have as much as processing elements as possible in conjunction with the high memory bandwidth.

Table 3.1: SCRIMP execution on Intel Xeon Phi KNL versus STOMP on NVIDIA Tesla K80<sup>‡</sup>.

	$2^{17}$	$2^{18}$	$2^{19}$	$2^{20}$	$2^{21}$
NVIDIA Tesla K80	10s	18s	46s	150s	555s
Intel Xeon Phi KNL 7210	0.98s	3.48s	13.51s	54.70s	222.61s
Speedup	10.20x	5.17x	3.40x	2.74x	2.49x

<sup>‡</sup>GPU results extracted from [86]

## Memory Bandwidth Results

We measure the memory bandwidth of our implementation by using the Intel VTune profiling tool [3]. Figure 3.8 shows the normalized execution time for every bandwidth usage given a random time series of  $2^{18}$  elements, which allows more accurate profiler results. We use a window size of 1024, 256 hardware threads and the same five configurations than in the previous subsection. In the atomic-based implementation, the memory usage is very low, since most of the time the threads are competing for the locks. The two cases corresponding to the DDR4-only implementations present a high memory bandwidth usage taking account of the maximum of this type of memory. This maximum is not enough for taking advantage of the sequential order of the diagonals.

With respect to the HBM plus DDR4 based implementations, we can notice how whereas a random order of the diagonals uses a high memory bandwidth, the sequential order nearly achieves the maximum for the HBM memory most of the time.

In another set of experiments, we measure the bandwidth utilization of our privatization-based implementations varying the number of threads, using the same time series and parameters than in the previous experiment. With these experiments we aim to test how is the memory bandwidth usage for different number of threads.

We show the first experiment in Figure 3.9, where we use only the DDR4 memory and a sequential order for computing the diagonals. We observe that the maximum bandwidth is obtained using 64 threads, which agrees with the point where the speedup graph becomes plain, as shown in Figure 3.7.

We present the second experiment in Figure 3.10, where we use the combination of the HBM plus the DDR4 bandwidth and a sequential order for the diagonals. In this case, the bandwidth starts to grow from the beginning until

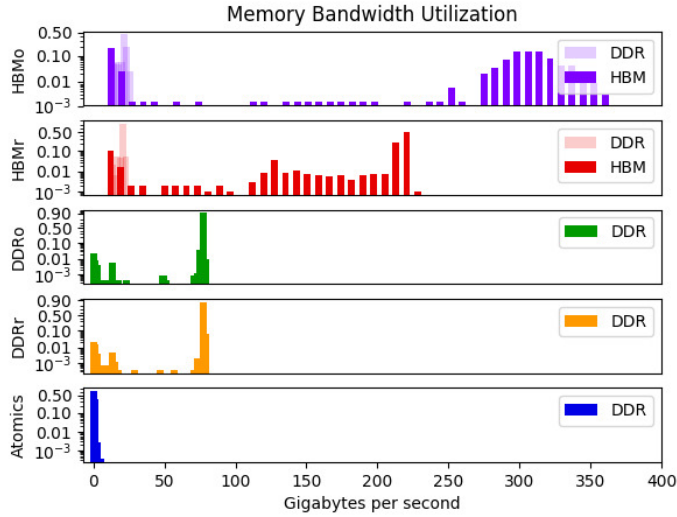


Figure 3.8: SCRIMP memory bandwidth utilization in the Intel Xeon Phi KNL, using different configurations, normalized as percentage of execution time. DDRo stands for DDR4 ordered and DDRr stands for DDR4 random, while HBMo means HBM+DDR4 ordered and HBMr means HBM+DDR4 random.

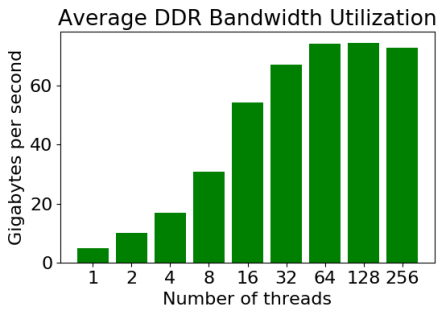


Figure 3.9: Average DDR4 bandwidth utilization, measured varying the number of threads in an execution of SCRIMP, using a random series of  $2^{18}$  elements and a window of 1024.

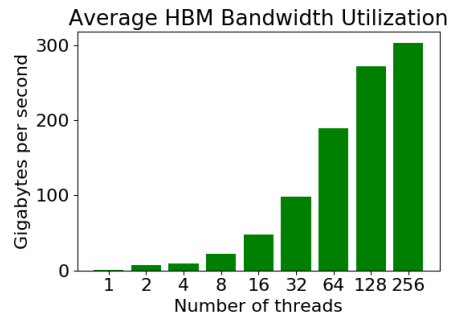


Figure 3.10: Average HBM bandwidth utilization, measured varying the number of threads in an execution of SCRIMP, using a random series of  $2^{18}$  elements and a window of 1024.

the maximum number of threads, where it achieves the theoretical maximum bandwidth for non sequential accesses.

### Sensibility to Time Series Length and Window Size

Tables from 3.2 to 3.5 show the sensibility of the implementations when varying the time series length and the window size. All of these tests use the number of threads that achieves the best performance. To conduct these experiments, we define random time series with representative sizes. In particular, we use time series of  $2^{17}$ ,  $2^{18}$ ,  $2^{19}$ ,  $2^{20}$  and  $2^{21}$  elements as they are common values used in the literature [202], as well as the most common window sizes ( $m$ ).

Table 3.2: Time for SCRIMP using only DDR4 in random order.

m	$2^{17}$	$2^{18}$	$2^{19}$	$2^{20}$	$2^{21}$
1024	5.65s	24.18s	119.70s	579.01s	2599.26s
2048	5.51s	23.84s	119.51s	590.99s	2615.70s
4096	5.32s	23.42s	118.71s	592.58s	2622.94s
8192	4.92s	22.54s	116.38s	586.66s	2637.59s
16384	4.21s	20.66s	111.77s	577.15s	2611.05s

Table 3.4: Time for SCRIMP using HBM plus DDR4 in random order.

m	$2^{17}$	$2^{18}$	$2^{19}$	$2^{20}$	$2^{21}$
1024	1.99s	8.38s	37.37s	153.61s	568.52s
2048	2.17s	9.22s	40.72s	169.43s	614.55s
4096	2.17s	9.35s	41.03s	173.04s	631.99s
8192	2.03s	9.07s	40.43s	173.02s	639.72s
16384	1.82s	8.48s	39.14s	167.54s	637.03s

Table 3.3: Time for SCRIMP using only DDR4 in sequential order.

m	$2^{17}$	$2^{18}$	$2^{19}$	$2^{20}$	$2^{21}$
1024	5.62s	23.49s	104.26s	468.75s	2063.70s
2048	5.51s	23.29s	104.09s	472.22s	2100.94s
4096	5.32s	22.82s	103.97s	473.78s	2148.62s
8192	4.93s	21.94s	101.25s	474.86s	2088.82s
16384	4.24s	20.24s	97.98s	464.04s	2072.78s

Table 3.5: Time for SCRIMP using HBM plus DDR4 in sequential order.

m	$2^{17}$	$2^{18}$	$2^{19}$	$2^{20}$	$2^{21}$
1024	1.28s	4.88s	20.71s	88.19s	380.82s
2048	1.27s	4.86s	21.13s	90.09s	397.05s
4096	1.24s	4.88s	20.77s	90.17s	406.58s
8192	1.18s	4.79s	20.46s	91.64s	402.10s
16384	1.03s	4.48s	19.88s	90.43s	411.40s

Table 3.2 shows the results of executing SCRIMP with the DDR4 memory and setting a random order for the diagonals. As the results suggest, there is not a clear correlation between the window size and the execution time, but depends on the number of elements that fits in caches.

In the case we use a sequential order for computing the diagonals, we obtain similar execution times than in the previous case, as shown in Table 3.3. Taking account that we use software prefetching for random and sequential order for the diagonals, there is not significant performance benefits for short series, but for bigger ones, we can achieve up to 25% of better execution times.

We also evaluate the combination of the HBM plus the DDR4 memories. If

we use the HBM memory for allocating the most used variables and a random order for the diagonals, we obtain from 2.8x to 4.6x of speedup with respect to only using the DDR4, depending on the window size and the time series length. Larger time series obtain more benefit from the use of HBM than the smaller ones, as shown in Table 3.4.

We show the experiments using the sequential order for the diagonals and the combination of the HBM plus the DDR4 memory in Table 3.5. In this case, we obtain up to 58% of better execution times than the same memory configuration with random order for the diagonals.

### Floating Point Performance

Table 3.6 summarizes the observed floating point operations per second that our implementation can achieve. We obtain such results using the Intel Advisor tool [1] using the same  $2^{18}$ -element random time series used previously with a window size of 1024 and 256 threads.

Table 3.6: GFLOP/s for SCRIMP with different configurations.

Configuration	Random order for diagonals	Sequential order for diagonals
DDR4 only	18.49 GFLOP/s	18.70 GFLOP/s
HBM + DDR4	47.79 GFLOP/s	60.18 GFLOP/s

We observe that there is not significant benefit in terms of GFLOP/s between computing in random or sequential order using only the DDR4 memory, because the bandwidth is already saturated even with the sequential order and data locality is not exploited. In contrast, if combined both the DDR4 and the HBM memory, there is a significant difference (around 20% of more throughput from the FLOP/s viewpoint) between using or not random order for the diagonals, as explained before. Nevertheless the results remain far away from the processor peak performance, which is a common issue in memory bounded problems.

### Real-World Applications

Lastly, we test our implementations taking as reference some of the datasets used in [196], [86] and [202], which allows us to compare and validate the results. These datasets come from real-world applications. We are interested in the exact

solution of the Matrix Profile, so our executions finish when SCRIMP converges (no partial solutions). Exact solutions are specially relevant in problems such as anomaly detection, which in some cases are detected at the final stage of the execution.

The first case study we evaluate is seismology. We present it in Figure 3.11, where the dataset (upper graph) consists of about 40,000 elements. Our multi-threaded and vectorized implementation of SCRIMP returns the Matrix Profile that we show in the lower graph, in which there are represented several peaks corresponding with the most significant discords, where the distance of the corresponding subsequence is higher (the similarity with respect to the whole time series is lower). Using a window size of 100 elements, our implementation of HBM plus DDR4 memory allocation policy and sequential order for the diagonals took only 0.25 seconds in obtaining the Matrix Profile, instead of 22.5 seconds of the original sequential implementation.

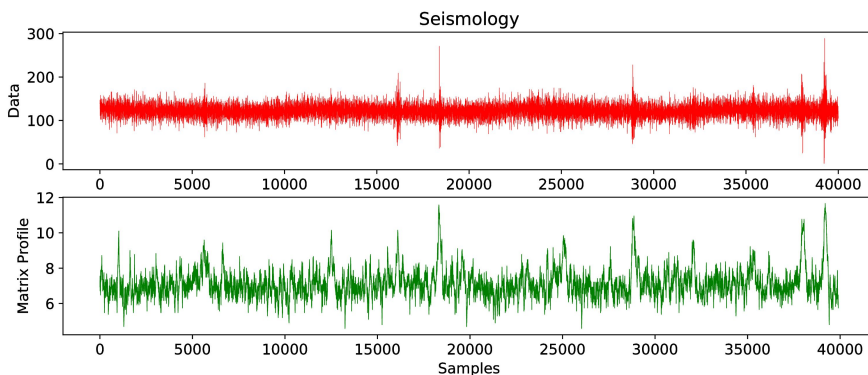


Figure 3.11: Seismology data. This dataset, which consists of approximately 40,000 elements, presents several peaks corresponding to earthquakes. The Matrix Profile is able to accurately identify them, which are represented as higher values of distance for the corresponding subsequences, as they are discords with respect to the whole time series.

Another application we use for testing our implementation is penguin data, which consists of a dataset of 110,000 elements approximately, obtained from an accelerometer. In Figure 3.12 we show the corresponding Matrix Profile of the dataset, from which a biologist could infer when the penguin is diving or walking, for example. In this case, the execution takes 0.93 seconds using our implementation of HBM plus DDR4 memory allocation policy and sequential



order for the diagonals, instead of 39.53 seconds of the original sequential version, using a window size of 800 elements.

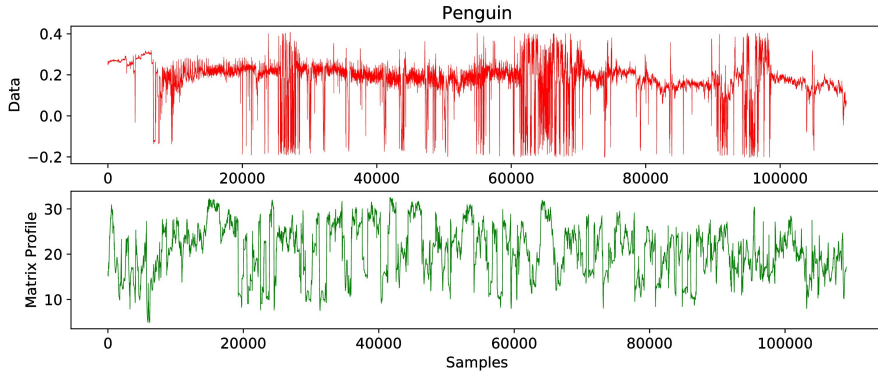


Figure 3.12: Penguin data. This dataset, which consists of approximately 110,000 elements, presents several motifs that can help a biologist to identify when the penguin is diving, for example.

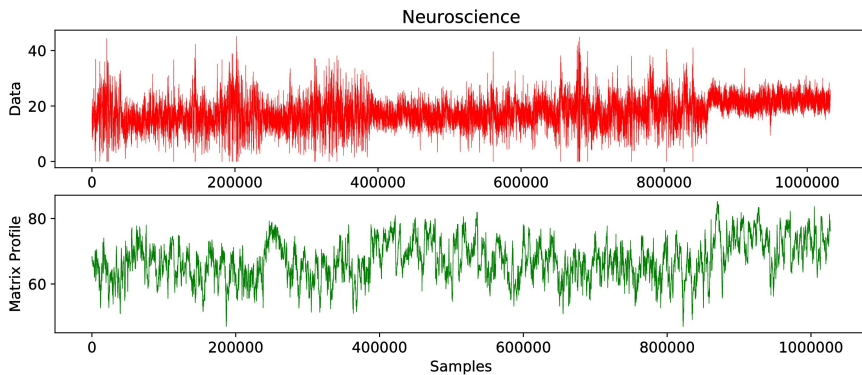


Figure 3.13: Neuroscience data. This dataset, which consists of approximately 1,030,000 elements, presents several discontinuities which are represented in the graph as values of 0 for readability. The Matrix Profile is able to obtain coherent results even with those discontinuities, proof of robustness of the algorithm.

Finally, we evaluate a neuroscience dataset, which contains 1,030,000 elements approximately, represented in the upper part of the Figure 3.13. Note that there are several discontinuities (missing values) in the dataset, which are represented

as a value of 0 in the graph for readability. The Matrix Profile algorithm can give us a coherent results even with this outliers, which is a proof of the robustness of the algorithm. Using our implementation of HBM plus DDR4 memory allocation policy and sequential order for the diagonals, the Matrix Profile takes 81 seconds to be calculated, whereas the original sequential implementation takes 15,108.12 seconds, using a window size of 5,400.

# 4 NATSA: A PNM Accelerator for TSA

---

## 4.1. Motivation and Key Idea

We find that several CPU and GPU implementations of *Matrix Profile* have been proposed in the literature. However, these acceleration efforts still require transferring the time series data from the main memory to the CPU/GPU cores, leading to the data movement bottleneck, as analyzed in the previous chapter using PhiTSA. Processing-Near-Memory (PNM) is a promising approach to alleviate data movement by placing processing units close to memory. As a result, PNM solutions have the potential to greatly improve system performance and energy efficiency when they are carefully designed with low-cost and low-overhead near data processing cores for memory-bound applications.

Our goal in this work is to enable high-performance and energy-efficient time series analysis for a wide range of applications, by minimizing the overheads of data movement. This can enable efficient time series analysis on large-scale systems as well as embedded and mobile devices, where power consumption is a critical constraint (e.g., heart beat analysis on a mobile medical device to predict a heart attack [118] or early earthquake detection [41]). To this end, we propose *NATSA*, the *first* Near-Data Processing *Accelerator* for *Time Series Analysis*. The key idea behind NATSA (Figure 4.1) is to exploit modern 3D-stacked High Bandwidth Memory (HBM) along with specialized custom processing units in the logic layer of HBM, to enable energy-efficient and fast *matrix profile* computation near memory, where time series data resides. NATSA supports a wide range of time series applications thanks to *matrix profile*'s generality and flexibility.

## 4.2. NATSA Architecture

Our Near-Data Processing Accelerator for Time Series Analysis, NATSA, is designed to 1) fully exploit the memory access parallelism and high memory bandwidth offered by HBM, and 2) employ the required amount of computing resources to provide a balanced solution. NATSA is built next to the HBM memory and exploits the full HBM bandwidth available. NATSA consists of multiple processing units (PUs) that efficiently compute the diagonals of *matrix profile* in a parallel fashion. The PUs are designed to compute diagonals using a vectorized approach to process a batch of elements of a diagonal at the same time. Each PU includes energy-efficient floating-point units [60], bitwise operators, and registers (See Table 4.3 in Section 4.3.3). Each PU communicates with HBM via a controller connected to one of the 8 memory channels provided by HBM.

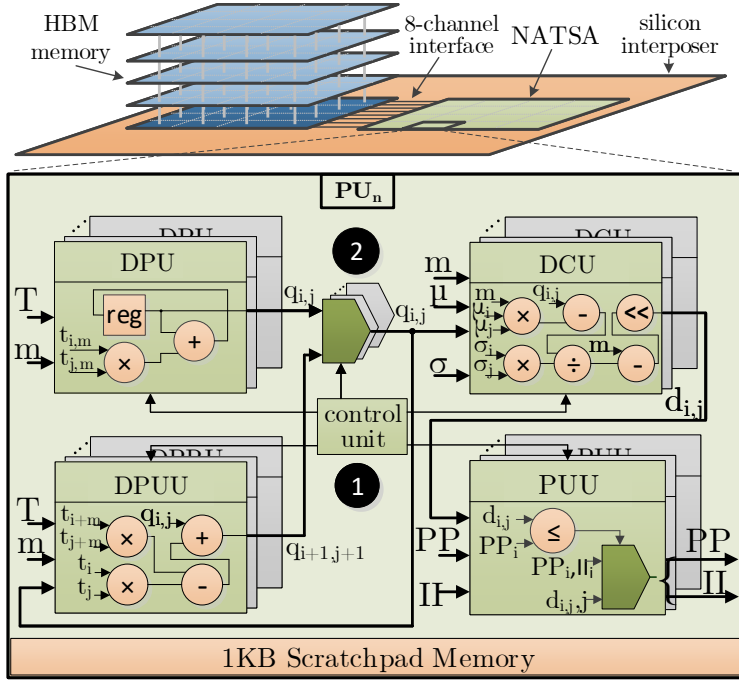


Figure 4.1: NATSA design and integration next to HBM memory. NATSA is connected directly to the HBM interface.

### 4.2.1. NATSA Processing Units (PUs)

Each NATSA PU consists of four hardware components: the *Dot Product Unit* (DPU), the *Distance Compute Unit* (DCU), the *Profile Update Unit* (PUU), and the *Dot Product Update Unit* (DPUU), as we show in Figure 4.1. We share the floating-point arithmetic operators (e.g., multipliers) among those hardware components to minimize idle cycles and enable reusability. The control unit (❶ in Figure 4.1) is a state machine that orchestrates the execution flow of a PU. The multiplexers (❷ in Figure 4.1) choose between the output of DPU and DPUU based on a signal from the control unit, so that the DCU can take advantage of Equation 2.2, starting from the second element of the diagonal all the way down to the last. We replicate those hardware components to compute different elements of a diagonal in parallel, using a vectorized approach. The diagonal assignment is pre-calculated in the host CPU, which sends the indices of the to-be-computed diagonals to each NATSA PU. Finally, each NATSA PU uses its own 1KB scratchpad memory to temporarily store fixed-size auxiliary data, such as the window size or configuration parameters.

The execution flow through the hardware components of a PU includes the following six steps:

1. **Dot product computation of the first element of the diagonal.** The DPU calculates the dot product between the first pair of subsequences of the diagonal ( $T_{i,m}$  and  $T_{j,m}$ ) by using the time series input  $T$ , and the window size,  $m$ , which is used to signal the end of each subsequence. This hardware component vectorizes the operation and outputs the result,  $q_{i,j}$ , for the next step.
2. **Euclidean distance computation of the first element of the diagonal.** The DCU computes the first Euclidean distance of each diagonal following Equation 2.1, using the dot product computed by the DPU  $q_{i,j}$ . The values of  $\mu$  and  $\sigma$  are precomputed by the host CPU in negligible time ( $O(n)$  [151]) with respect to the total execution time. This simplifies the design of the PU.
3. **First profile update.** If the Euclidean distance calculated in the DCU,  $d_{i,j}$ , is lower than that stored in the profile for both subsequences, the PUU updates the profile vector and index vector,  $PP$  and  $II$ .
4. **Dot product update.** The dot product of the second and successive cells in the diagonal is calculated from the previous cell. It is computed in the DPUU by subtracting the first product and adding the new one to  $q_{i,j}$ , as

shown in Equation 2.2. This hardware component is replicated to enable vectorization and is pipelined with the DCU and the PUU.

5. **Second and successive Euclidean distance computations.** The DCU computes again the Euclidean distance, but now it obtains  $q_{i,j}$  from the DPUU. The DPUU hardware component is replicated for vectorization of the dot product update calculations.
6. **Second and successive profile updates.** The PUU updates the profile vector and profile index vector, if needed. This hardware component is replicated to perform several updates at a time.

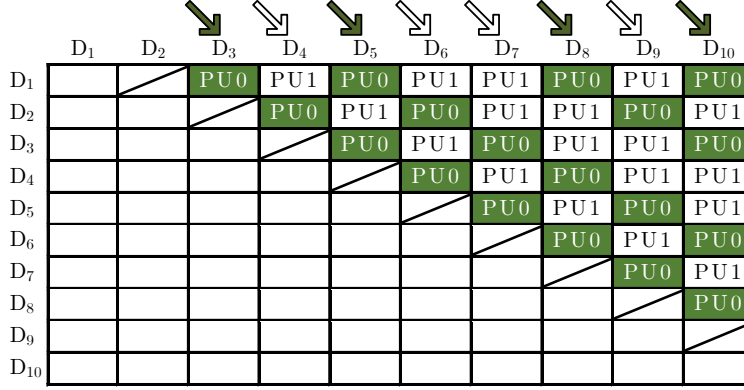
### 4.2.2. Workload Partitioning Scheme

Computing the diagonals of the distance matrix may lead to load imbalance among the PUs, because those diagonals have different lengths. To avoid this imbalance, we propose a static partition scheduling scheme which depends only on the size of the time series and the exclusion zone.

The way we tackle this problem is by assigning a set of pairs of diagonals to each NATSA PU such that the sum of their elements is equal to the number of cells of the main diagonal of the distance matrix minus the number of cells of the exclusion zone,  $(n - m + 1) - m/4$ .

Figure 4.2 illustrates an example with two PUs,  $PU0$  and  $PU1$ , a distance matrix for a time series of  $n = 13$  cells, a window size of  $m = 4$ , and an exclusion zone of 1 diagonal (crossed out rectangles). In this case, the number of elements that each pair of diagonals assigned to a PU should have is  $(n - m + 1) - m/4 = 10 - 1 = 9$ . Comparing a subsequence with itself gives zero distance value. As a consequence, the algorithm treats the main diagonal as exclusion zone and avoids computing it. The first diagonal of non-zero values, which starts in column  $D_2$  and is represented with crossed out rectangles, belongs to the exclusion zone (see Figure 2.3), so NATSA PUs also skip it.

Discarding the computation of the main diagonal and the diagonals in the exclusion zone, both PUs have to compute the diagonals from columns  $D_3$  to  $D_{10}$ . To perform this efficiently and maintain the *anytime* property of SCRIMP, in the first step,  $PU0$  is assigned the first and last diagonal (9 elements in total), and  $PU1$  is assigned the second and the penultimate diagonal (totalling 9 elements as well). In the second step,  $PU0$  computes the third and the third-to-last diagonal, whereas  $PU1$  computes the fourth and fifth diagonals.



The figure shows a 10x10 grid representing the scheduling of diagonals D<sub>1</sub> through D<sub>10</sub> for two processing units, PU0 (green) and PU1 (white). The grid is divided into two main sections by a diagonal line from D<sub>1</sub> to D<sub>10</sub>. The top section (D<sub>1</sub> to D<sub>10</sub>) contains the following assignments: D<sub>1</sub> (white), D<sub>2</sub> (white), D<sub>3</sub> (green), D<sub>4</sub> (white), D<sub>5</sub> (green), D<sub>6</sub> (white), D<sub>7</sub> (white), D<sub>8</sub> (green), D<sub>9</sub> (white), D<sub>10</sub> (green). The bottom section (D<sub>1</sub> to D<sub>10</sub>) contains the following assignments: D<sub>1</sub> (white), D<sub>2</sub> (white), D<sub>3</sub> (white), D<sub>4</sub> (white), D<sub>5</sub> (white), D<sub>6</sub> (white), D<sub>7</sub> (white), D<sub>8</sub> (white), D<sub>9</sub> (white), D<sub>10</sub> (white). Arrows above the grid indicate the direction of computation: green arrows point down-right, and white arrows point up-right.

	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	D <sub>8</sub>	D <sub>9</sub>	D <sub>10</sub>
D <sub>1</sub>			PU0	PU1	PU0	PU1	PU1	PU0	PU1	PU0
D <sub>2</sub>				PU0	PU1	PU0	PU1	PU1	PU0	PU1
D <sub>3</sub>					PU0	PU1	PU0	PU1	PU1	PU0
D <sub>4</sub>						PU0	PU1	PU0	PU1	PU1
D <sub>5</sub>							PU0	PU1	PU0	PU1
D <sub>6</sub>								PU0	PU1	PU0
D <sub>7</sub>									PU0	PU1
D <sub>8</sub>										PU0
D <sub>9</sub>										
D <sub>10</sub>										

Figure 4.2: Example of the diagonal scheduling scheme for two processing units, denoted as *PU0* (green) and *PU1* (white). Arrows show direction of computation.

Our proposed scheduling scheme can be used in two ways: 1) *Randomly ordering* the indices of diagonals that each PU has to compute. Using this approach, we are able to preserve the *anytime* property of the algorithm, since if the execution is interrupted, the user obtains a partial exploration of the whole time series (i.e., events from any point of the time series can be detected). 2) *Sequentially ordering* the indices of diagonals that each PU has to compute. This approach violates the *anytime* property (i.e., only events up to the interruption point can be detected), but allows for further optimizations (e.g., exploiting data locality between consecutive diagonals).

**Data mapping.** Each PU has access to its corresponding portion of the time series and statistic vectors, and works with replicated profile and profile index vectors. This approach simplifies the overall architecture, enabling the use of many PUs without having to synchronize between them. NATSA assigns multiple diagonals to each PU with the specific scheduling scheme described in this section.

### 4.2.3. Programming Interface

In this section, we introduce the API to invoke NATSA from a host processor. While conventional loosely-coupled accelerators (e.g., GPUs or FPGAs) have their own memory, where data must be transferred to from the host's memory, NATSA is a tightly-integrated PNM accelerator, located between the host CPU

and main memory. Thus, there is no need to transfer any data between the host memory and the accelerator memory, as loosely-coupled accelerators require. The user is responsible for 1) allocating the time series ( $T$ ) and 2) providing the window length ( $m$ ). NATSA will provide the user the profile vector ( $P$ ) and profile index vector ( $I$ ) in return. The size of the exclusion zone ( $\frac{m}{4}$  by default) can be also passed as a parameter ( $exc$ ).

Algorithm 3 outlines the NATSA API. First, **NATSA** function precalculates the statistics ( $\mu, \sigma$ ) (line 2) in the host CPU and allocates the private vectors ( $PP, II$ ) to NATSA's PUs (line 3).

---

**Algorithm 3** NATSA API

---

```

1: function  $P, I \leftarrow \text{NATSA}(T, m, exc, conf)$ 
2:    $\mu, \sigma \leftarrow \text{precalculateMeanDev}(T, m)$ 
3:    $PP, II \leftarrow \text{allocatePrivateProfiles}(T, m, exc)$ 
4:    $idx \leftarrow \text{diagonalScheduling}(T, m, exc)$ 
5:    $\text{START\_ACCELERATOR}(T, m, exc, conf, idx, PP, II)$ 
6:    $P, I \leftarrow \text{reduction}(PP, II)$ 

```

---

Second, **NATSA** function implements the diagonal scheduling scheme presented in the previous section, setting the diagonals to be computed by each PU in  $idx$  (line 4). Third, it initiates the accelerator (line 5), which starts the computation, and the host CPU waits for all the processing units to finish. Once the computation finishes, the host CPU performs the final reduction of the private vectors (line 6) and the user can find the results in the  $P$  and  $I$  vectors. The **conf** argument (line 1), besides holding configuration parameters for the accelerator, allows for future extensions, such as using other distance metrics (e.g., Pearson correlation [203]).

## 4.3. Evaluation

### 4.3.1. Methodology

In this subsection we describe the simulation environment and the workload we use to evaluate the performance of NATSA.

We simulate general-purpose cores using an in-house integration of *ZSim* [155], whose front-end is *Pin* [128], with *Ramulator* [107] [152]. *ZSim* is a simulator which can model 1) general purpose cores (both in-order and out-of-order cores),



and 2) the conventional cache hierarchy. Ramulator is a cycle-level and extensible DRAM simulator that provides a wide variety of memory models, including DDR4 [92] and HBM [114]. We use *McPAT* [120] for power estimations.

For the NATSA accelerator, we use the *gem5* [27] and *Aladdin* [161] integration developed in [168]. Aladdin provides performance, area, and power estimations for a system-on-chip accelerator by requiring the equivalent C implementation of the accelerator design. Aladdin estimates the performance, power, and area of the accelerator within 0.9%, 4.9%, and 6.6% compared to that provided by RTL flows, but over 100× faster [161]. As Aladdin does not model the memory subsystem, we need to simulate it using *gem5*.

For a fair comparison, we evaluate our baseline platform (see the evaluated platforms below) in both ZSim and *gem5* frameworks using the same workload (see Section 4.3.2). We obtain up to 10% simulated time reduction using ZSim with respect to *gem5* (i.e., the baseline system performs slightly better with ZSim). As a consequence, the performance benefits of NATSA with respect to the baseline simulated using *gem5*, would be even higher. However, we choose ZSim since simulations of manycore systems with ZSim are orders of magnitude faster than *gem5* simulations [155], and this allows for the evaluation of general-purpose core platforms with large time series. For both general-purpose cores and accelerators, we obtain the power consumption of the memory system using the Micron Power Calculator [4], which we feed with the bandwidth usage from Ramulator and *gem5*, respectively.

Using these simulation environments, we define several representative hardware platforms for the evaluation:

- **DDR4-OoO (*Baseline*):** A conventional DDR4-based system with eight four-wide out-of-order cores at 3.75GHz, intended for HPC. Each core has 32KB private L1 instruction/data caches and a private 256KB L2 cache. The cores share an 8MB L3 cache. The main memory is a dual channel 16GB DDR4-2400 with 38.4GB/s of memory bandwidth.
- **DDR4-inOrder:** A conventional architecture using 64 in-order cores at 2.5GHz, intended for power-constrained environments. Each core has only a single level of private 32KB instruction/data caches. The main memory is the same DDR4 as in the baseline system. We use this simple core-cache configuration to compare with the following PNM general-purpose-core system.
- **HBM-OoO:** An PNM architecture with eight four-wide out-of-order cores at 3.75GHz, intended for HPC. Each core has 32KB private L1 instruc-

tion/data caches and a private 256KB L2 cache. The main memory is a 4GB 3D-stacked HBM2 that provides a throughput of 256GB/s.

- **HBM-inOrder:** An PNM architecture with 64 in-order cores at 2.5GHz, intended for power-constrained environments. Each core has a single level of private 32KB instruction/data caches. The main memory is a 4GB 3D-stacked HBM2 that provides a throughput of 256GB/s.
- **NATSA:** Our PNM accelerator with 48 PUs at 1GHz. Each PU has access to a private scratchpad memory of 1KB. The main memory is the same 4GB 3D-stacked HBM2 as in the *HBM-OoO* and *HBM-inOrder* platforms.

### 4.3.2. Workload

We use two real datasets and five synthetic datasets to evaluate the performance of NATSA against state-of-the-art architectures. The two real datasets are electrocardiogram (ECG) and seismology data obtained from [171] and [195]. We use these real datasets to 1) verify the correctness of the *matrix profile* computed by NATSA (the same approach used in [195]) and 2) evaluate the effect of using single-precision versus double-precision (see Section 4.3.3). We generate the five synthetic datasets of different representative lengths [202] for performance evaluation using MATLAB, as shown in Table 4.1.

Table 4.1: Synthetic time series for performance evaluation.

Time Series	rand_128K	rand_256K	rand_512K	rand_1M	rand_2M
Length (n)	131072	262144	524288	1048576	2097152

### 4.3.3. Results

In this section, we first evaluate NATSA’s performance, comparing it to the general-purpose platforms (DDR4-OoO, DDR4-inOrder, HBM-OoO, and HBM-inOrder). Second, we compare NATSA to both simulated and real architectures (e.g., many-core CPUs and GPUs [86]) in terms of power consumption and area. Third, we present a design space exploration of NATSA. Fourth, we analyze the performance of general-purpose cores and their bottlenecks. Finally, we evaluate SCRIMP in terms of precision and sensitivity to subsequence lengths ( $m$ ).

### Performance of NATSA

We evaluate the performance of two NATSA designs using single-precision (SP) and double-precision (DP), respectively. We present normalized performance of NATSA-DP with respect to the baseline platform (DDR4-OoO) in Figure 4.3, using double-precision data. NATSA achieves significant performance improvements, up to  $14.2\times$  ( $9.9\times$  on average) over the baseline system for large time series, and  $6.3\times$  over HBM-inOrder for all sizes. We observe that NATSA’s speedup increases as the time series length becomes larger. This is because the arithmetic intensity decreases when the ratio of time series length ( $n$ ) to window size ( $m$ ) increases. Dot product update causes the first dot product to take a significant part of the computation for shorter diagonals (lower  $n$  to  $m$  ratio). The cache hierarchy of the baseline system accelerates the first dot product. Conversely, a greater  $n$  to  $m$  ratio results in longer diagonals with the first dot product being less significant with respect to the total execution time, reducing the observed benefits of a cache hierarchy.

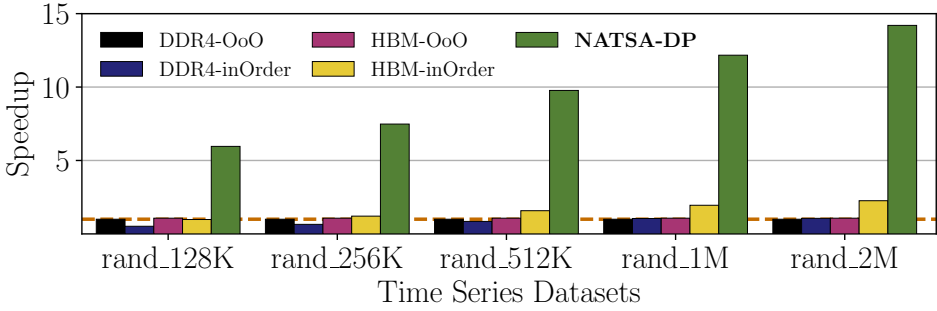


Figure 4.3: Speedup with respect to the baseline platform (DDR4-OoO) using double precision data.

We evaluate the performance of the single-precision NATSA design.<sup>1</sup> Table 4.2 presents the average execution time for the analyzed datasets. NATSA-SP, which provides higher performance with similar area cost to NATSA-DP, outperforms NATSA-DP by up to  $1.75\times$ , DDR4-OoO-DP by up to  $24.9\times$  and HBM-inOrder-DP by up to  $11.1\times$  for large time series.

We conclude that NATSA provides the highest performance compared to modern general-purpose platforms.

<sup>1</sup>We note that NATSA experiments are carried out with the gem5-Aladdin simulation framework, and the other platforms are evaluated with the ZSim-Ramulator framework (baseline system included). As mentioned in Section 4.3.1, simulated times are slightly shorter for ZSim, so the actual gains of NATSA would likely be even greater what we report.

Table 4.2: Execution time (in seconds) for single-precision and double-precision.

Config \ Dataset	rand_128K	rand_256K	rand_512K	rand_1M	rand_2M
DDR4-OoO-DP	14.72	77.55	414.55	2089.05	9810.30
DDR4-OoO-SP	6.46	44.47	207.85	1106.36	5206.75
HBM-inOrder-DP	14.95	64.20	262.33	1071.03	4347.38
HBM-inOrder-SP	8.16	35.68	130.23	625.27	2466.69
NATSA-DP	2.47	10.37	42.45	171.72	690.65
<b>NATSA-SP</b>	<b>1.41</b>	<b>5.91</b>	<b>24.19</b>	<b>97.84</b>	<b>393.45</b>

### Power, Energy and Area Consumption

**Power and Energy Consumption.** We compare the power and energy consumption of NATSA versus other existing hardware platforms in Figures 4.4 and 4.5. We use McPAT and Micron Power Calculators to evaluate energy consumption for the general-purpose platforms, getting the number of stalls and bandwidth usage from ZSim-Ramulator. For NATSA, we add Aladdin’s energy estimations to the values obtained from the Micron Power Calculator. We also obtain energy measurements from real executions on GPUs using NVVP [6] and on CPUs using PCM [2], to compare NATSA with real platforms.

Figure 4.4 shows the dynamic power consumption of each simulated or real hardware platform. We observe that NATSA has the lowest power consumption, and most of its power is consumed by memory.

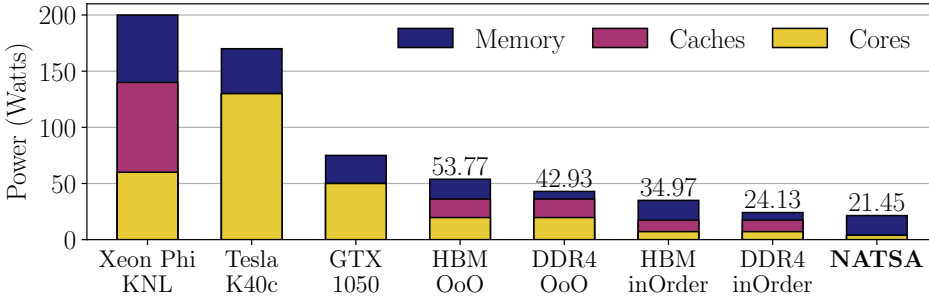


Figure 4.4: Dynamic power consumption for simulated and real hardware platforms.

Figure 4.5 shows the energy consumption of each simulated or real platform, for the computation of a time series of 524,288 elements (rand\_512K) using double-precision. To calculate the energy consumption, we compute the power-delay product with the measured instantaneous power consumption and the ex-

ecution time. NATSA reduces energy consumption by  $27.2\times$  ( $19.4\times$  on average) over the baseline platform (DDR4-OoO), and by  $10.2\times$  over an PNM architecture with general-purpose cores (HBM-inOrder). NATSA consumes  $1.7\times$ ,  $4.1\times$ , and  $11.0\times$  less energy than an NVIDIA Tesla K40c GPU [145], NVIDIA GTX 1050 GPU [5], and Intel Xeon Phi KNL [165], respectively. We conclude that NATSA is the most energy-efficient evaluated platform for *matrix profile*.

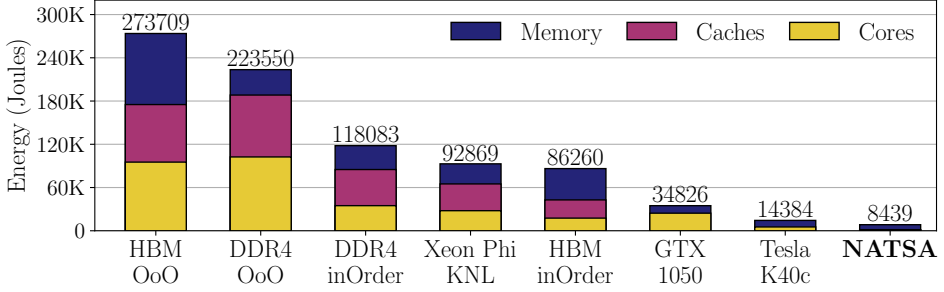


Figure 4.5: Energy consumption for simulated and real hardware platforms.

**Area.** We provide a scaled area comparison in Figure 4.6. We observe that NATSA requires  $9.6\times$ ,  $7.9\times$ ,  $3\times$ , and  $1.8\times$  less area than an Intel Xeon Phi KNL (14nm), NVIDIA Tesla K40c (28nm), Intel Core i7 (32nm), and NVIDIA GTX 1050 (14nm).

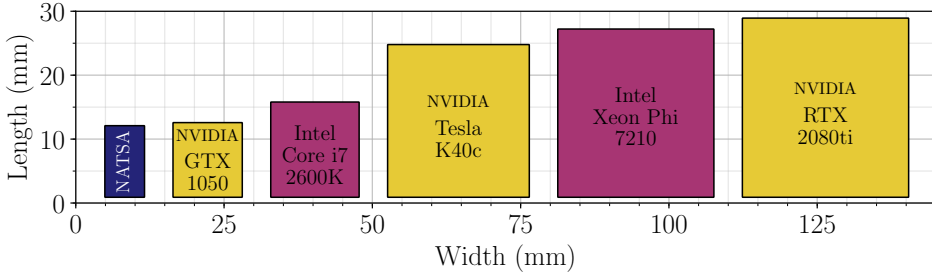


Figure 4.6: Area comparison of different hardware platforms.

We conclude that NATSA (at 45nm technology node) is the platform that requires the least area, while using the largest technology node (i.e., 45nm) compared to other evaluated architectures. Using a more recent and smaller technology node (e.g., 15nm instead of 45nm) could additionally reduce NATSA's energy consumption by  $4\times$  and area by  $3\times$  [154].

## NATSA Design Space Exploration

We explore the key design choices of NATSA so that we deploy the exact number of PUs that saturate the memory bandwidth available, while minimizing the area and power consumption of the accelerator. We evaluate the use of HBM memory,<sup>2</sup> where we find that 48 PUs make the accelerator balanced between memory bandwidth and compute parallelism, as 64 PUs result in a memory-bound accelerator, whereas 32 PUs a compute-bound one. Table 4.3 details the design parameters of NATSA for HBM. NATSA has 48 PUs which run at a frequency of 1GHz, fabricated at 45nm process. Implementations of NATSA with lower technology nodes would provide smaller area footprint and improved energy efficiency. Table 4.3 shows the components in a PU depending on the data precision: 1) double-precision (DP), and 2) single-precision (SP).

Table 4.3: NATSA design components for 48 PUs.

Parameter/Component	PU-DP	NATSA-DP	PU-SP	NATSA-SP
Mem. bandwidth (GB/s)	5	240	5	240
Peak power (W)	0.1	4.8	0.08	3.84
Area ( $mm^2$ )	1.62	77.76	1.51	72.48
FP Multipliers/Adders	16/14	768/672	64/36	3072/1728
Integer Adders	16	768	64	3072
Bitwise Operators	2	96	2	96
Registers	108	5184	267	12816

## Performance of General-Purpose Cores

We evaluate the speedup over the baseline (DDR4-OoO) and memory bandwidth usage of SCRIMP, calculated using the ZSim-Ramulator framework for the DDR4-OoO, DDR4-inOrder, HBM-OoO and HBM-inOrder platforms using double-precision time series of different lengths ( $n$ ), in Figure 4.7.

We report execution time of the baseline (DDR4-OoO) on top of the respective performance bars in Figure 4.7. Based on these results, we make three key observations. First, the DDR4-OoO platform does not use the peak available bandwidth of DDR4 (i.e., 38.4GB/s). We reinforce this observation with our HBM-OoO evaluation which replaces DDR4 with higher bandwidth HBM. HBM-OoO platform improves performance by only 7%, which means that provid-

<sup>2</sup>We also explore the use of DDR4 memory, where 8 PUs are enough to saturate the available memory bandwidth and the performance obtained is similar to the DDR4-inOrder platform (4% difference).

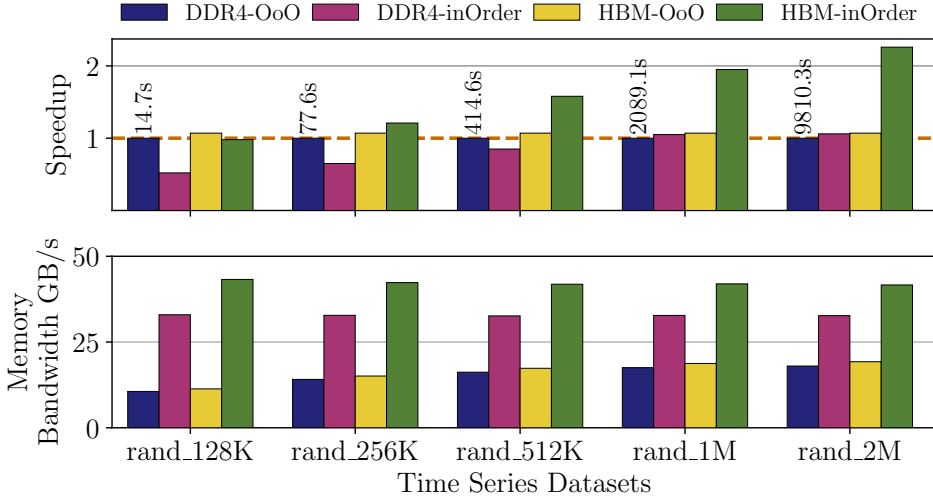


Figure 4.7: Speedup over the baseline DDR4-OoO and memory bandwidth usage for general-purpose platforms.

ing more bandwidth does not significantly affect performance. This is because both platforms are compute-bound when executing SCRIMP. Second, the 64 lightweight cores of DDR4-inOrder slightly outperform the 8 complex cores of DDR4-OoO when  $n \geq 1048576$  elements (i.e., rand\_1M dataset). This is because shorter time series can fit in the L3 cache. For long time series, the higher parallelism provided by the in-order platform enables higher memory-level parallelism [140, 137, 70, 139, 138, 141] and higher memory bandwidth demand, where DDR4 bandwidth becomes a bottleneck, resulting in a memory-bound system. Third, the HBM-inOrder platform provides up to  $2.25\times$  speedup over the baseline (DDR4-OoO), and consumes only 17% of the HBM’s peak bandwidth with the largest dataset evaluated. In this case, even though performance is improved, the application is still compute-bound and simple PNM general-purpose cores cannot fully exploit the bandwidth provided by HBM (256GB/s)<sup>3</sup> for the largest dataset we evaluate, which means that large datasets can be comfortably accommodated.

We conclude that general-purpose platforms provide less performance than NATSA’s design as they do not effectively exploit the memory bandwidth.

<sup>3</sup>Based on the memory bandwidth usage and McPAT, we estimate that a general-purpose based architecture would need 128 OoO cores (area 688mm<sup>2</sup>, TDP 1137W, 18nm) or 384 in-order cores (area 164mm<sup>2</sup>, TDP 126W, 18nm) to take full advantage of the maximum bandwidth provided by HBM.

### Accuracy and Sensitivity to Subsequence Length

**Accuracy.** We explore how the accuracy of the SCRIMP implementation is affected by changing the precision of the data representation. We use real data obtained from [171] and [195], as discussed in Section 4.3.2. Figure 4.8 presents the output obtained for an electrocardiogram (ECG) and for seismology data using two precision values. We observe that events are still detectable even when reducing the precision from *double* to *single* precision. This observation can be exploited to improve performance and reduce energy consumption, by operating on smaller arithmetic units and less memory footprint.

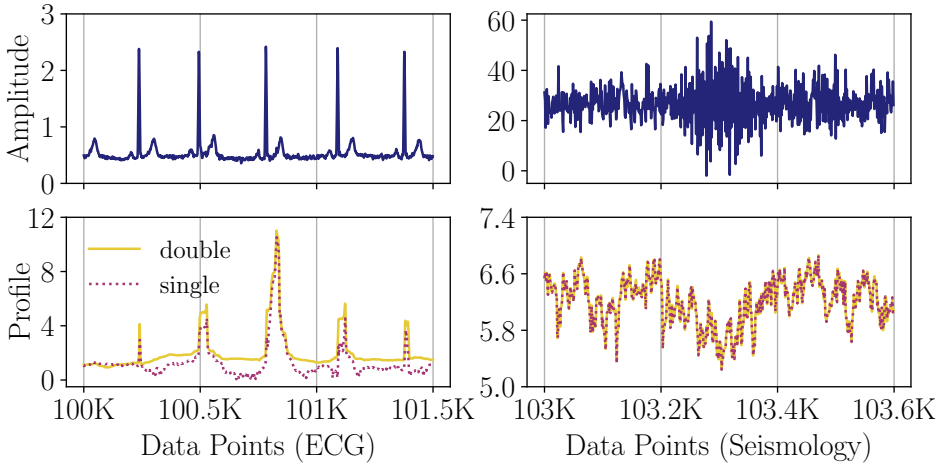


Figure 4.8: ECG (left) and seismology (right) data along with their profiles, using double and single precision, where events are easily visible.

**Sensitivity to the subsequence length.** We finally perform a sensitivity analysis to the subsequence length ( $m$ ). We observe that, when the proportion between  $m$  and  $n$  is less than two orders of magnitude, the performance of SCRIMP in all platforms is significantly affected by  $m$ . For example, when increasing  $m$  from 1,024 to 16,384 in a time series of 131,072 elements, the execution time of SCRIMP reduces by 41%. However, when the time series length is large enough compared to the subsequence length, performance of SCRIMP is affected by a smaller amount. For instance, when increasing  $m$  from 1,024 to 16,384 in a time series of 2,097,152 elements, the execution time of SCRIMP reduces by 13%. This is because the computation of the first element of each diagonal involves the dot product calculation without any reutilization.



# 5 MATSA: A PUM Accelerator for TSA

---

## 5.1. Motivation and Key Idea

TSA is widely applicable to many domains, while there is *no* previous proposal that satisfy the requirements of the application to provide a highly-efficient solution for power constraint scenarios. We conclude justifying the need for a NVM-based PUM TSA accelerator since 1) TSA is widely applicable to many domains and 2) there is no previous proposal that satisfy the requirements of TSA to enable an efficient solution.

In contrast to commodity architectures, a PUM architecture based on Non-Volatile Memories (NVM) constitutes a promising solution to accelerate TSA, since it since it 1) enables high parallelism, 2) reduces data movement costs, 3) can provide better scalability than its competitors due to massive parallelism, 4) is CMOS-compatible, and 5) overcomes the data volatility problem. We find that the main drawback of NVM technologies are their lack of mature, being difficult to evaluate. To tackle with this issue, we perform sensitivity studies in our evaluation for different NVM technologies.

In this chapter, we present MATSA, the *first* MRAM-based Accelerator for Time Series Analysis. The key idea is to exploit magneto-resistive memory cross-bars to enable energy-efficient and fast time series computation in memory while overcoming endurance issues of other non-volatile memory technologies. MATSA provides the following key benefits: 1) it leverages high levels of parallelism in the memory substrate by exploiting column-wise arithmetic operations, and 2) it

significantly reduces the data movement costs performing computation using the memory cells.

Compared to NATSA, which moves compute units closer to data, MATSA goes one step forward in the way to solve the memory wall by placing computation exactly where data resides.

## 5.2. MATSA Architecture

In this section, we present 1) the mechanism of MATSA, that enables high-performance and energy-efficient TSA, and 2) how MATSA is integrated in commodity systems.

### 5.2.1. Overview

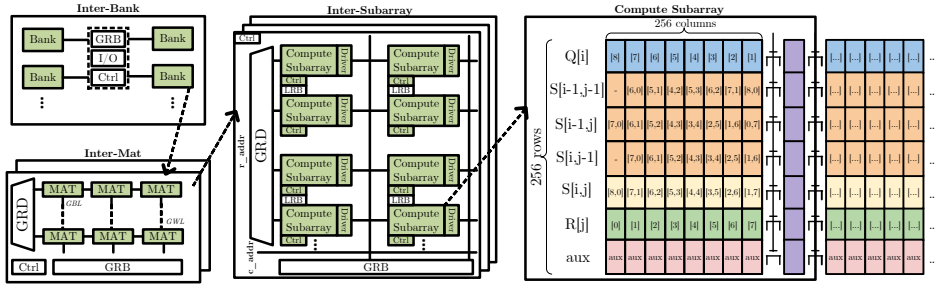


Figure 5.1: MATSA's high-level Architecture and Data Mapping. Note that modifications at *Inter-Bank*, *Inter-Mat*, and *Inter-Subarray* levels with respect to common NVM devices are negligible, easing fabrication and compatibility.

MATSA is an MRAM-based Accelerator for Time Series Analysis designed to substitute the main processing unit (e.g., CPU or  $\mu$ controller) when performing subsequence Dynamic Time Warping (sDTW). This way, MATSA improves the energy efficiency of the entire system. MATSA implements sDTW since 1) it is more robust than Euclidean Distance based ones for most of scenarios (See Background Chapter) and also 2) sDTW's computation flow fits better the PUM paradigm. Figure 5.1 presents an overview of our proposed architecture. MATSA is composed of one or several chips divided into multiple *banks*. Banks that belong to the same chip share buffers and I/O interfaces and work in a lock-step approach based on an integrated global controller (*Ctrl*) that orchestrates the flow. Each

bank is composed of several *Multiple Memory Matrices (MATs)*. The MATs share a *Global Row Buffer (GRB)* and are connected to a *Global Row Decoder (GRD)*. We place a *Local Row Buffer (LRB)* for every pair of subarrays. MATs are composed of several memory *subarrays*.

Each memory subarray is composed of magnetoresistive devices (e.g., SOT-MRAM cells) that are connected to the *Write Word Lines (WWL)*, *Write Bit Lines (WBL)*, *Read Word Lines (RWL)*, *Read Bit Lines (RBL)*, and *Source Lines (SL)*. The memory cells perform the sDTW computation in combination with reconfigurable Sense Amplifiers (SAs). We place a *MATSA controller (Ctrl)* next to each subarray, which is in charge of orchestrating the data flow and activating the subarrays to perform the sDTW computation properly. We discuss the system integration of MATSA in Subsection 5.2.7.

### 5.2.2. MATSA Subarrays

MATSA subarrays are comprised of non-volatile memory cells following a conventional crossbar organization that can work either in regular memory or compute mode. This is a desirable feature since our design consists of 1) subarrays that buffer the data pending to be processed and 2) subarrays that perform the actual computation. Adjacent subarrays are connected using pass gates and auxiliary columns (purple one in Figure 5.1) to enable the flow at this level of the hierarchy.

**Memory subarrays.** MATSA subarrays in *regular* memory mode support both read and write data operations. To perform a write operation, MATSA first invokes the *Memory Row Decoder (MRD)* to activate the proper WL. Second, the corresponding WD applies the voltage difference needed to switch the cell to the BL and SL (which induces a resistance change in the MJT accordingly). To perform a read operation, MATSA first invokes the MRD to activate the proper WL in the write operation, while the MCD is in charge of connecting the proper BL to its SA. Then, the sense amplifier interprets the voltage value across the path, compares it against a reference voltage, and produces the output result (logic '1' or '0').

**Compute subarrays.** MATSA subarrays working in *compute mode* perform bit-wise operations (e.g., NOR) for operands belonging to the same column. This enables the parallel execution of many operations at the same time since columns in the same subarray work synchronously. The key idea is to select two or three operands simultaneously using the MRD. This produces an equivalent resistance that depends on the content of the selected cells and modifies the sensing volt-

age across such column accordingly. We include a redesigned reconfigurable sense amplifier (Figure 5.2) per column, from which MATSA's Ctrls can select different thresholds depending on the operations. For example, assuming that the desired operation is the majority of three operands, Ctrl activates the three corresponding rows and sets *EnMaj* to logic '1'. Then, if the equivalent resistance of the activated cells in such column is below *RMaj*, the result produced by the SA will be logic '1', or logic '0' otherwise. Most operations are performed in a single cycle thanks to the gates added to SAs.

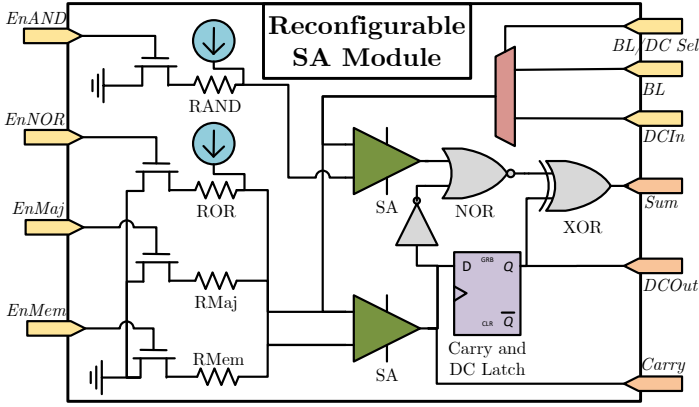


Figure 5.2: MATSA's Reconfigurable Sense Amplifier. Latch register is reused for Addition and Diagonal Copy operations.

The execution flow of MATSA is orchestrated by a hierarchy of controllers implemented as finite state machines (FSMs). In particular, MATSA comprises 1) a global controller that orchestrates inter-bank communication, 2) several inter-mat controllers that take care of the inter-mat communications, and 3) subarray controllers that activate the memory rows and drive Reconfigurable SAs according to sDTW's algorithm.

### 5.2.3. sDTW Challenges in NVM-PUM

Using an NVM-PUM architecture to accelerate sDTW raises two main challenges: 1) accessing neighboring columns and 2) column-wise parallelism.

**Accessing neighboring columns.** Each sDTW's calculated value depends on two cells from the  $j - i$  column ( $S[i - 1, j - 1]$  and  $S[i, j - i]$ ). We address this issue by introducing a novel single cycle and parallel *diagonal copy operation*.

**Column-wise parallelism.** Ensuring that columns can work independently and at the same time is critical for sDTW performance. We address this issue by introducing a novel data mapping and execution flow based on wavefront execution.

#### 5.2.4. Supported Operations

To support the sDTW algorithm, MATSA implements the following in-memory operations:

- **Row Copy.** The key idea behind this mechanism is to perform consecutive memory read and write operations in the same cycle. In the first half cycle, the corresponding subarray's MRD activates the source row that is read by the LRB. After that, the data is stored in the destination row in the second half cycle. This mechanism works at MAT and bank levels using the Global Row Buffer (GRB) to accelerate the copy operations across the hierarchy.
- **Diagonal Row Copy.** A diagonal copy occurs when the source and destination cells belong to adjacent columns. To enable such a feature, MATSA leverages the available registers in the sense amplifiers and the interconnections between them. The operation is performed in two steps. First, each SA reads the source value of its own column. Second, each SA takes the value of its left SA neighbor and writes it to its own column. This operation is performed in parallel when the source and destination rows are common for a given set of cells.
- **Addition/Subtraction.** MATSA implements bit-serial addition/subtraction across columns. It computes bit-by-bit operations from the LSB of the two operands until the MSB. Every bit position takes two memory cycles, further divided into four steps. In the first step, the SAs use the two bit cells activated in the same bit lines as input operands and calculate *Sum* taking into account the stored carry in the register available in the SA. In the second step, the SAs write back the *Sum* value to the corresponding cell. In the third step, the SAs calculate the new *Carry* based on a majority function. In step four, SAs write the result in a reserved cell in their register.
- **Absolute value.** To calculate the absolute value, MATSA first checks the sign bit, leading to two possible scenarios: 1) if the number is positive, no change is needed; otherwise, 2) if the number is negative, MATSA inverts

the bits of the number and adds '1' to the result using the addition operation (similar to 2's complement).

- **Minimum value.** To calculate the minimum value between three elements, MATSA performs two comparisons based on the subtraction operation. First, it calculates the difference between two of the numbers. Second, it checks the resulting sign from the previous step and selects one of the two numbers for comparison against the third. The final comparison sign determines the overall minimum.

### 5.2.5. Data Mapping

We design MATSA's data mapping to leverage the parallel column-wise computation support in MRAM. This organization requires that the operands have to be mapped to the same column. There are three basic structures involved in the sDTW computation: 1) reference, 2) queries, and 3) the warping matrix. The size of the warping matrix can be huge:  $O(NM)$ . However, we are interested in the distance value (i.e., no need to store the alignment), which can be computed by iterating over a single vector that holding the current row of the warping matrix. To do so, we define the *s\_vector*, but we note that each *s\_vector* element (which has to be mapped to different crossbar column) requires accessing previous *s\_vector* values that are mapped to another different columns (i.e.,  $S[i-1, j-1]$ ,  $S[i-1, j]$ ). To overcome this challenge, we propose to add three temporal *s\_vector* in the crossbar array, which are updated accordingly for each step of the computation:  $S[i-1, j-1]$ ,  $S[i-1, j]$  and  $S[i, j-i]$  (see Figure 5.1). Overall, our optimization reduces the memory footprint from  $O(MN)$  to  $O(4N)$ . Each subarray's column is composed of 256 cells sliced in the following way (e.g., using `int32` as datatype):

- **Reference elements ( $R[j]$ ).** We vertically map each reference element to 32 cells of a column. If 1) the number of available columns is bigger than the number of elements in reference, we replicate the reference to multiple columns to increase parallelism (distributing the queries between them). If 2) the number of available columns is lower than the number of elements in reference, we divide the sDTW matrix and complete processing in successive sequential batches.
- **Query elements ( $Q[i]$ ).** We vertically map each query element to 32 cells of a column. New query elements are introduced on the left side of the crossbar, and they are right-shifted in each successive step (see Section 5.2.6).

- **Current s\_vector** ( $S[i, j]$ ). We vertically map each element of the s\_vector to 32 cells of a column, being aligned with the reference. The query element is computed in a given step of the algorithm ( $i$  and  $j$  indexes).
- **Temporal s\_vectors** ( $S[i-1, j-1]$ ,  $S[i-1, j]$  and  $S[i, j-1]$ ). We vertically map the three temporal vectors that enable independence to 32 cells of a column, in alignment with the reference and query elements that are computed in a given step of the algorithm.
- **Aux Cells**. Each column is provided with a slice of 64 cells that are used to hold the partial results during the execution flow, as explained in the next subsection.

We note that MATSA's design requires internal data movement. For typical reference sizes, data movement happens at the inter-subarray level. If the reference is big enough, a hierarchy of paths enable its computation efficiently.

### 5.2.6. Execution Flow

MATSA's execution flow follows a wavefront [48] approach, which reflects the computation pattern in dynamic programming applications. The motivation is that sDTW's matrix has to be computed in the wavefront manner due to inter-cell dependencies. Figure 5.3 shows how we tackle this challenge.

The key idea is to perform the computation diagonally by assigning a diagonal element to each processing element (PE), and use the *diagonal row copy* operation to communicate between processing elements (columns) and get the surrounding values. This approach 1) enables parallelism while computing a given query and 2) creates a pipeline where several queries are being processed. The execution flow iterates over the following steps until queries remain.

1. **Distance calculation.** Calculation of  $dist(Q[i], R[j])$ , which provides the first partial result  $P1$ . This process implies several substeps depending on the selected distance metric, (e.g., subtraction  $\rightarrow$  absolute value).
2. **Minimum.** Calculation without storing the result of  $min(S[i-1, j-1], S[i-1, j], S[i, j-1])$ , which produces the value for the next step  $S1$ .
3. **Addition.** Calculation of the addition between the minimum value selected in the previous step ( $S1$ ) and the partial result  $P1$ .

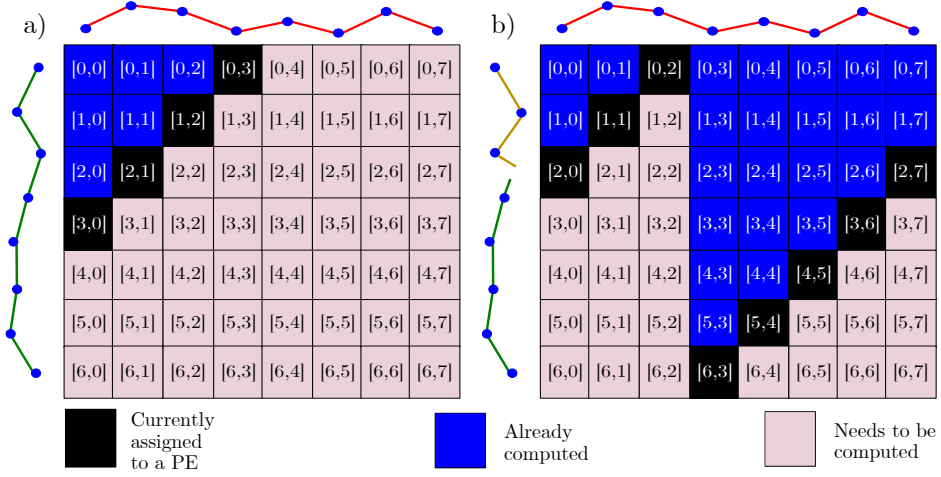


Figure 5.3: Wavefront-based sDTW computation. In a, PEs are able to calculate their matrix elements in parallel. In b, the pipeline is full and PEs are also working on different queries.

4. **Diagonal Copy.** Copying the  $S[i, j]$  vector into the  $S[i, j-1]$  vector shifted by one to the right.
5. **Diagonal Copy.** Copying the  $S[i-1, j]$  vector into the  $S[i-1, j-1]$  vector shifted by one to the right.
6. **Vertical Copy.** Copying the  $S[i, j]$  vector into the  $S[i-1, j]$  vector.
7. **Diagonal Copy.** Copying the  $Q[i]$  vector into the same  $Q[i]$  vector but shifted one position to the right.

### 5.2.7. System Integration

**Physical Device.** MATSA is designed to work synergistically with the main processing unit (e.g., CPU) to efficiently accelerate TSA. We propose three different versions of MATSA (see Figure 5.4) to meet the requirements of each environment:

1. **MATSA-Embedded.** A small chip intended to be integrated in edge devices (e.g., sensors).



2. **MATSA-Portable.** A USB-based accelerator intended for use in desktops and laptops computers, similar to Intel Neural Compute Stick [14].
3. **MATSA-HPC.** A high-performance PCIe-based accelerator intended to be integrated into servers.

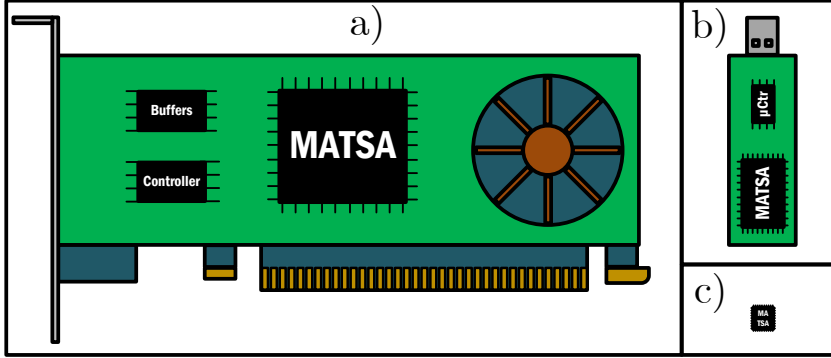


Figure 5.4: MATSA integration. a) High-performance accelerator (PCIe card). b) Portable accelerator (USB stick). c) Embedded accelerator (small chip).

**Programming Interface.** To enable efficient use of MATSA by the programmers, we expose an API (Listing 5.1) that invokes the accelerator based on the input data in a supported DTYPE (`int8`, `int16`, `int32`, `int64`, `fp32` or `fp64`), the selected mode (*query\_filtering* or *self\_join*) and the distance metric (*abs\_diff* or *square\_diff*). The user can optionally define an anomaly threshold, which provides a boolean array in return with the positions of the anomalies set to true.

```
void matsa(DTYPE *ref, DTYPE *queries, uint32_t *ref_size,
           uint32_t *query_sizes, uint32_t n_queries, char *mode,
           char *dist_metric, DTYPE anomaly_thres, bool *anomalies,
           DTYPE *distances)
```

Listing 5.1: MATSA's host interface function.

## 5.3. Evaluation

In this section, we present the evaluation of MATSA which comprises 1) a self-characterization and 2) a comparison with different baselines in terms of performance and energy.

### 5.3.1. Methodology

We compare several representative (*simulated* and *real*) hardware platforms for the evaluation.

- **(Simulated) CPU-ARM (*cpuarm*):** A portable-class platform based on an 4-core ARM CPU running at 2.5GHz, 32KB L1 caches and an 8GB LPDDR4 memory.
- **CPU-i7 (*cpui7*):** A desktop-class platform based on an 6-core (12 threads) Intel i7 x86 CPU running at 3.2GHz, 64KB L1 caches, 256KB L2 caches, 12MB L3 cache and a 64GB DDR4 memory.
- **CPU-Xeon (*cpuxeon*):** A server-class platform based on two 18-core (36 threads) Intel Xeon Gold x86 CPUs running at 3GHz, 32KB L1 caches, 1MB L2 caches, a 24.75 MB L3 cache and a 768GB DDR4 memory.
- **GPU (*gpu*):** An NVIDIA Tesla V100-SXM2 board equipped with 32GB of HBM memory.
- **FPGA (*fpga*):** An Alveo U50 board equipped with 872K LUTs and 8GB on-board HBM memory.
- **UPMEM (*upmem*):** A server-class platform based on PNM-enabled memory that is equipped with 2560 DPUs running at 425MHz [46, 71].
- **MATSA-Embedded (*matsa-embedded*):** An ultra low-power version of our accelerator, consisting of 128 compute-enabled crossbars.
- **MATSA-Portable (*matsa-portable*):** A balanced energy/performance version of our accelerator, consisting of 1024 compute-enabled crossbars.
- **MATSA-HPC (*matsa-hpc*):** A high-performance version of our accelerator, consisting of 4096 compute-enabled crossbars.

**Baselines.** To evaluate our *cpuarm* platform, we use ZSim [155] and Ramlator [107, 152] for performance and McPAT [120] for energy consumption evaluations. For the *cpui7* and *cpuxeon* platforms, we obtain the performance from the average of five executions (as we find that the divergence between them is negligible), and energy consumption using rapl-tools [102]. All the above CPU-based platforms execute the same parallel OpenMP C implementation of sDTW. To evaluate the performance of *upmem*, we implement sDTW for this platform. Further, we replicate the reference time series across all DPUs and distribute the

queries among them. For the energy measurements, we use an estimation tool provided by UPMEM [183]. To evaluate the performance of **fgpa**, we develop a custom HLS-based implementation of sDTW comprising six compute units that exploit the HBM bandwidth, and obtain the energy consumption using the *xbutil* tool [190]. Finally, we evaluate the performance of **gpu** by developing a custom CUDA-based implementation of sDTW that exploits the HBM bandwidth via coalescing, and obtain the energy consumption using the *nvidia-smi* tool.

Parameter	Values
Crossbar Size (cells)	256x256
Number of Crossbars	128, 256, 512, 1024, 2048, 4096
Read Latency (ns)	1, 3, 5, 10, 20
Write Latency (ns)	1, 3, 5, 10, 20
Read Energy (pJ)	20, 50, 100
Write Energy (pJ)	30, 70, 400

Table 5.1: MATSA design exploration parameters.

**MATSA.** To evaluate the performance and energy of MATSA, we perform a sensitivity analysis of the latency and energy of MRAM devices. We create a custom analytical-based simulator based on sDTW’s execution flow and its required operations. We feed this simulator with the workload parameters along with MATSA characteristics (memory cell latency/energy, etc.) and obtain the execution time and total energy from it. We present the parameter range we use in our evaluation in Table 5.1, going from conservative to optimistic ones based on the MRAM technology trends [153].

### 5.3.2. Workloads

We evaluate both MATSA and our baselines using the workload sizes listed in Table 5.2. We cover a wide combination of reference sizes and query sizes to better understand the tradeoffs in MATSA design aligned with real datasets parameters [179].

Parameter	Values
Reference Size	64K, 128K, 256K, 512K
Query Size	4K, 8K, 16K, 32K
Number of Queries	4K, 8K, 16K, 64K

Table 5.2: Workload parameters used in our evaluation.

### 5.3.3. Results

#### MATSA Characterization

We perform a design space exploration of MATSA taking into consideration performance parameters of the cells (i.e., read/write latencies and energies).

**Read/Write Latencies.** We evaluate how changing the read/write latencies affects the execution time and present the results in Figure 5.5. We observe that, increasing read latency by  $10\times$  incurs a  $4.7\times$  execution time penalty, while increasing the write latency incurs a  $6.5\times$  penalty.

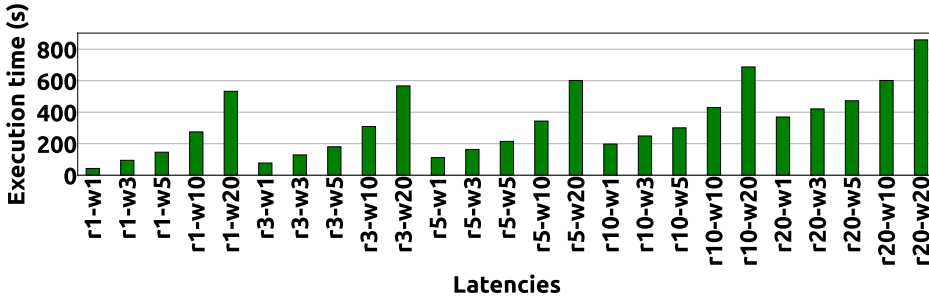


Figure 5.5: Execution time when varying cell read and write latencies (ref\_size=128K, query\_size=8K, num\_queries=8K, matsa\_cols=128K).

**Key Observation 1:** using a low write latency memory technology is crucial for MATSA's design

**Read/Write Energies.** We evaluate how the total execution energy varies with the per word write/read energy, and show the results in Figure 5.6. We observe here that the contributions of read energy and write energy are similar, thus both of them have to be carefully taken into consideration.

**Key Observation 2:** write energy contributes *only* 19% more than read energy to the total energy

**Workload sizes.** First, we evaluate how the execution time varies with different workloads sizes (i.e., ref\_size and query\_size) and present the results in Figure 5.7. Second, we evaluate how the execution energy varies with different workload sizes and present the results in Figure 5.8. We observe that both the reference size and the query size contribute equally to the execution time and energy. This happens because the total number of operations needed is directly

proportional to  $\text{ref\_size} \times \text{query\_size}$ .

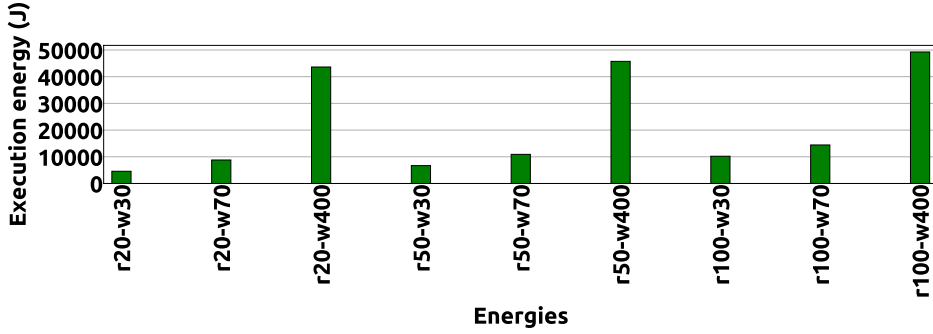


Figure 5.6: Execution energy when varying cell read and write energies ( $\text{ref\_size}=128\text{K}$ ,  $\text{query\_size}=8\text{K}$ ,  $\text{num\_queries}=8\text{K}$ ,  $\text{matsa\_cols}=128\text{K}$ ).

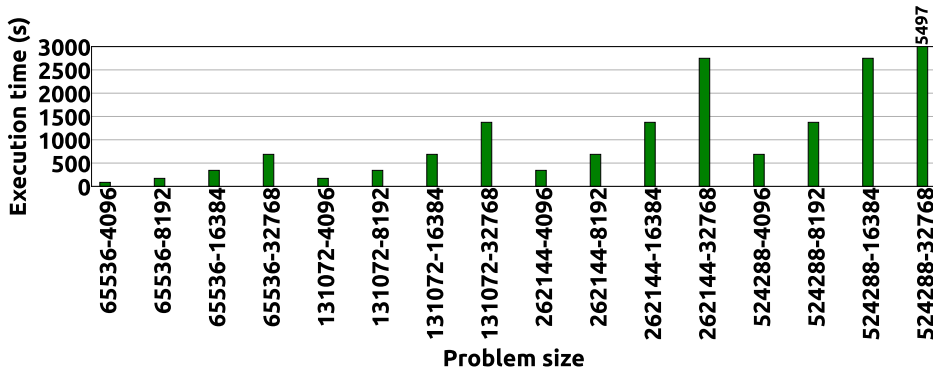


Figure 5.7: Execution time when varying workload sizes ( $\text{num\_queries}=8\text{K}$ ,  $\text{matsa\_cols}=128\text{K}$ ).

**Key Observation 3:** total execution time and energy are proportional both to the  $\text{ref\_size}$  and the  $\text{query\_size}$

**MATSA sizes.** We evaluate how the execution time varies when changing the number of MATSA's compute-enabled columns in Figure 5.9. MATSA provides almost-ideal scaling.

**Key Observation 4:** inherent independence across columns enables almost-ideal scaling when increasing # of columns

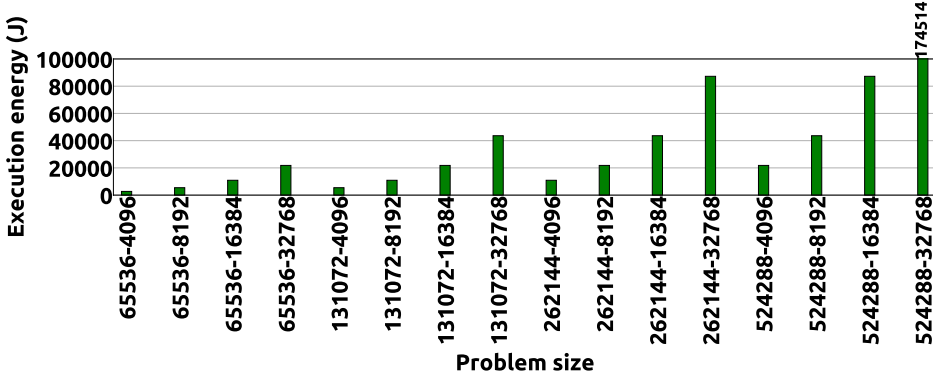


Figure 5.8: Execution energy when varying problem sizes (num\_queries=8K, matsa\_cols=128K).

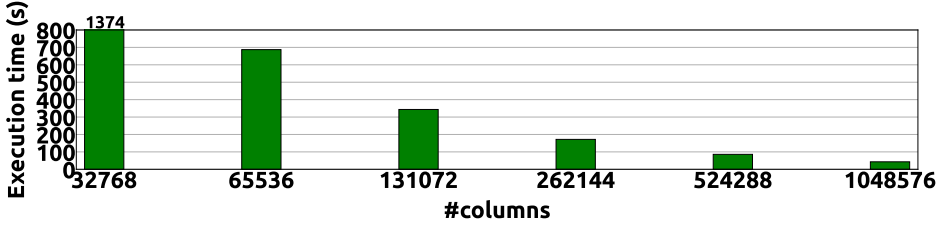


Figure 5.9: Execution time when varying MATSA sizes.

**Endurance.** Assuming that MATSA is built using 5ns rd/wr cells and runs 24/7 for 10 years, we estimate that (on average) each cell will be written on the order of  $4 \times 10^9$  times. Based on that, magnetoresistive technologies as STT-MRAM or SOT-MRAM are good candidates to build MATSA [52]. In contrast, ReRAM would not provide enough writing cycles.

## MATSA Comparison

In this section, we compare the performance and energy of three versions of MATSA with the corresponding baselines.

**MATSA-Embedded.** We first compare the performance of MATSA-Embedded (32K compute-enabled columns) with `cpuarm`, `cpui7` and `fpga` and present the results in Figure 5.10. We observe that MATSA-Embedded outper-

forms the `cpuarm` by  $30\times$ , the `cpui7` by  $1.3\times$  and the `fpga` by  $8\times$ <sup>1</sup>. Second, we compare the energy consumption of MATSA-Embedded with the baselines in Figure 5.11. We observe that MATSA-Embedded reduces the energy consumption by  $45\times$  with respect to the `cpuarm` baseline, by  $10\times$  with respect the `cpui7` baseline and by  $24\times$  with respect to the `fpga` baseline.

**MATSA-Portable.** We first compare the performance of MATSA-Portable (256K compute-enabled columns) with `cpuarm`, `cpui7` and `fpga` and show the results in Figure 5.10. We observe that MATSA-Embedded outperforms the `cpuarm` baseline by  $241\times$ , the `cpui7` baseline by  $10\times$  and the `fpga` baseline by  $64\times$ . Second, we compare the energy consumption of MATSA-Portable with the baselines in Figure 5.11.

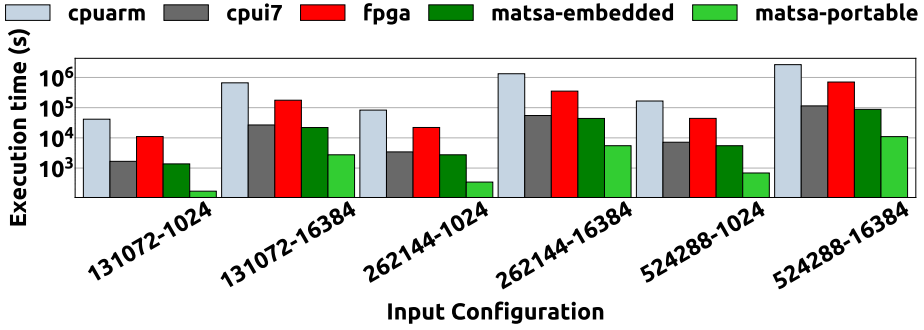


Figure 5.10: Execution times of MATSA-Embedded (num.cols=32K) and MATSA-Portable (num.cols=256K) versus baselines for different workload sizes (rd\_latency=5ns, wr\_latency=10ns).

**Key Observation 5:** Both MATSA-Embedded and MATSA-Portable improve performance and energy with respect to `cpuarm`, `cpui7` and `fpga`

**MATSA-HPC.** We first compare the performance of MATSA-HPC with `cpuxeon`, `gpu` and `upmem` and present the results in Figure 5.12. We observe that MATSA-HPC outperforms `cpuxeon` by  $7.3\times$ , `gpu` by  $6.15\times$  and `upmem` by  $6.3\times$ .

Second, we compare the energy consumption of MATSA-HPC with the baselines in Figure 5.13.

<sup>1</sup>The reader may wonder how the performance of the `cpui7` baseline can be better than the `fpga` one. The main reason for that is the cache memory in the CPU baseline, which is large enough to hold the main data structures of the algorithm, whereas the FPGA's scratchpad memory is smaller and the data structures have to be HBM located. Thus, HBM latency lags the FPGA behind the CPU baseline.

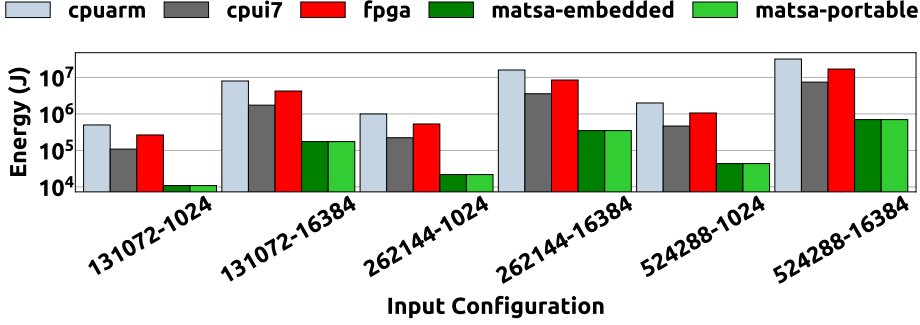


Figure 5.11: Execution energy consumption of MATSA-Embedded (num\_cols=32K) and MATSA-Portable (num\_cols=256K) versus baselines for different workload sizes (rd\_energy=50nJ, wr\_energy=70nJ).

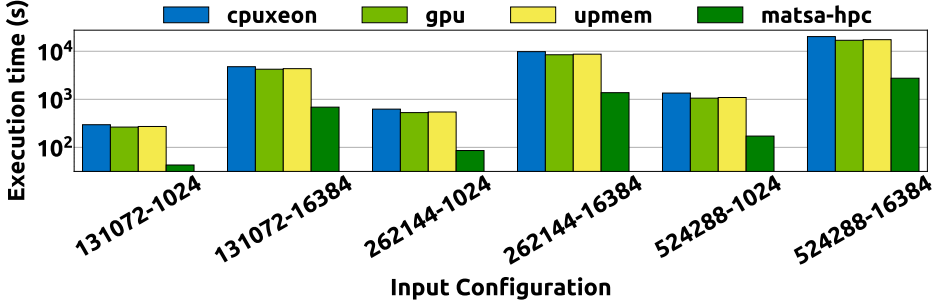


Figure 5.12: Execution times of MATSA-HPC versus baselines for different workload sizes (rd\_latency=5ns, wr\_latency=10ns, num\_cols=1M).

We note that MATSA-HPC is  $11\times$  more energy-efficient than `cpuxeon`,  $4.21\times$  more energy-efficient than `gpu` and  $2.6\times$  more energy-efficient than `upmem`. We note that `cpuxeon` is bottlenecked by 1) the limited parallelism (# of cores) and 2) memory hierarchy. Our `gpu` baseline provides high parallelism, but it is limited by data movement from and to memory. The PNM-based `upmem` baseline provides high parallelism and closeness to memory, but it is compute-bound for this application. Finally, MATSA-HPC provides huge parallelism, in-situ computation and high throughput, being a balanced solution.

**Key Observation 6:** MATSA-HPC provides better performance and energy consumption than `cpuxeon`, `gpu` and `upmem` baselines.



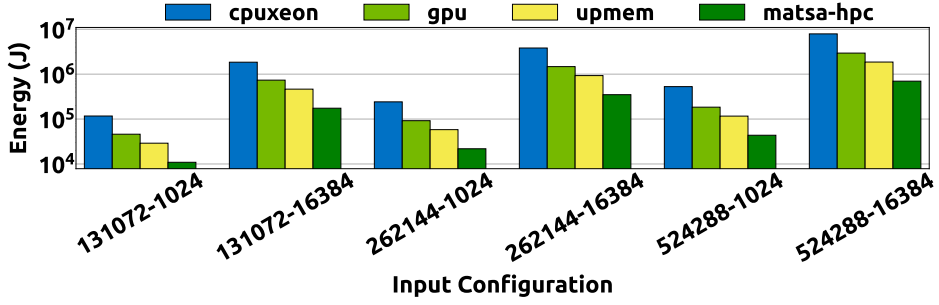


Figure 5.13: Execution energy consumption of MATSA-HPC versus baselines for different workload sizes (rd\_energy=50nJ, wr\_energy=70nJ, num\_cols=1M)

**Key Observation 7:** MATSA-Embedded, MATSA-Portable and MATSA-HPC reduce the energy consumption in a similar way with respect to a given baseline. This fact is explained because the main difference between the three MATSA versions is the increment in the number of columns (i.e., SIMD lanes). As a result, while execution time is reduced, the instantaneous power is incremented by the same factor, thus total energy consumed is similar.



UNIVERSIDAD  
DE MÁLAGA

# 6 TraTSA: A Transprecision Framework for TSA

---

## 6.1. Motivation and Key Idea

In this work, we evaluate the latest Euclidean-based implementations of matrix profile (SCRIMP [202] and SCAMP [203]) and find that a huge number of floating-point (FP) arithmetic operations are needed in order to analyze even short time series. In this sense, *transprecision computing* [130] has recently emerged as a promising approach to 1) improve energy efficiency, 2) provide better performance, 3) reduce area footprint, and 4) reduce memory bandwidth by tolerating some loss of accuracy in computed results. This paradigm reduces the number of bits for the exponent and the mantissa in FP operations in a flexible way, depending on the requirements of the application. It is well known that FP operations are a major contributor ( $\approx 50\%$ ) [173] to the energy consumption in modern computing platforms. Thus, transprecision has the potential to provide an efficient design with the required precision by the application.

Our goal in this work is to provide a set of tools to jumpstart the research niche of transprecision time series analysis, to achieve high-performance and energy-efficient computing for a wide range of applications. This way, new platforms can be designed that benefit from reduced-bit-count FP operations tailored to each application (e.g., using a transprecision Floating-Point-Unit (FPU), like FPNew [8]). This opens up the opportunity to detect important events on mobile and embedded devices, where energy is a critical concern. Those devices can be used, for example, to prevent ecological disasters or medical issues (e.g., for early earthquake detection [15] or to predict a heart attack [118]).

To this end, we introduce *TraTSA*, the *first* transprecision framework for time series analysis. TraTSA provides fast and user-friendly transprecision matrix profile computing thanks to its CPU and FPGA implementations. We evaluate TraTSA with use cases of real datasets from different domains and sizes, analyzing the trade-offs between arithmetic precision and result accuracy using a proposed metric. Additionally, we present the energy savings of a real transprecision FPU.

## 6.2. TraTSA Framework

In this section we present an overview of the TraTSA framework and then describe its main components.

### 6.2.1. Overview of TraTSA

TraTSA is a Transprecision Framework for Time Series Analysis developed as a tool 1) to perform design exploration of accelerators and 2) to tune current implementations. This way, computer architects can define the exact number of floating-point bits for exponent and mantissa, which potentially saves area and improves performance while reducing energy consumption. We build TraTSA framework based on matrix profile and using 1) the FlexFloat library to implement transprecision CPU versions of SCRIMP and SCAMP, 2) the cpfp-FPGA library to implement a transprecision FPGA implementation of SCAMP<sup>1</sup> and 3) Python to create a user-friendly wrapper.

We present TraTSA's overview in Figure 6.1. The *INPUT* and *OUTPUT* blocks in the figure represent TraTSA's wrapper. This wrapper is in charge of 1) interpreting the configuration file (i.e., exponent and mantissa widths, selected CPU or FPGA backend, window size, among others) and obtaining the time series file provided by the user; 2) invoking the corresponding execution backend, and 3) collecting the results providing them to the user after proper formatting. We present a simple example of TraTSA's configuration file in Figure 6.2, which is based on a custom format. The stats file follows a similar format to that provided by the FlexFloat [172] library.

---

<sup>1</sup>We do not consider implementing a transprecision FPGA version of SCRIMP since SCAMP provides better numeric stability and, as a consequence, more robustness to reduced precision (see Section 6.9.2), being more amenable to transprecision approaches. However, due to the high-similar computation schemes of both algorithms, it is feasible to implement *TransSCRIMPfpga* with modest effort.

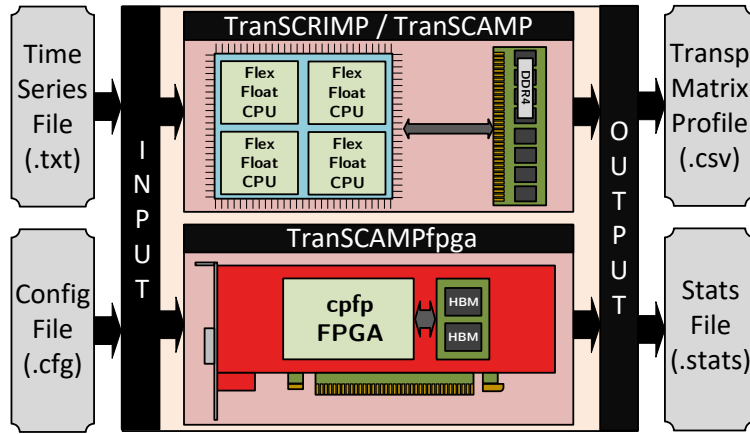


Figure 6.1: TraTSA overview and its components. The user provides a time series file (.txt) and a configuration file (.cfg) to the wrapper. Then, the wrapper invokes matrix profile either in the CPU or in the FPGA. Finally, the wrapper provides the user the transprecision matrix profile (.csv) and some statistics (.stats).

```

7 16 # distance_exponent distance_mantissa
7 16 # dotproduct_exponent dotproduct_mantissa
7 16 # statistics_exponent statistics_mantissa
7 16 # profile_exponent profile_mantissa
1024 # window_size
0    # backend (0=TranSCRIMP,1=TranSCAMP,2=TranSCAMPfpga)

```

Figure 6.2: Example of TraTSA's cfg file.

TraTSA, being easily extensible to support additional algorithms, includes the following backends:

- **TranSCRIMP.** TranSCRIMP is a CPU C++ parallel transprecision implementation of SCRIMP algorithm. This implementation provides configurable precision arithmetic which is emulated via software using the FlexFloat library.
- **TranSCAMP.** TranSCAMP is a CPU C++ parallel transprecision implementation of SCAMP algorithm. This implementation provides configurable precision arithmetic which is emulated via software using the FlexFloat library.
- **TranSCAMPfpga.** TranSCAMPfpga is an FPGA HLS-based implementation

of SCAMP algorithm. This implementation provides configurable precision arithmetic which is implemented via hardware using the `cpfp-FPGA` library. As we evaluate **TranSCAMPfpga** using a Xilinx Alveo U50 FPGA board, it is trivial to port it to other Alveo models.

The key benefit of TraTSA is to provide a transprecision framework being 1) portable enough to be executed in different execution environments according to the analysis requirements (i.e., length of the time series or if the user has access to an FPGA or not), and 2) flexible enough to allow the possibility of exploring a wide range of exponent and mantissa combinations for any dataset. In this sense, both **TranSCRIMP** and **TranSCAMP** are designed to be used with time series of modest sizes (below 200K elements) and executed in commodity CPUs (desktops or high-end servers) due to the overheads of custom-precision types in those platforms. In contrast, **TranSCAMPfpga** is able to compute series of up to several million elements in manageable time thanks to the transprecision hardware support, at the cost of requiring an FPGA. Both CPU and FPGA backends can work simultaneously and join compute power.

### 6.2.2. Transprecision SCRIMP-CPU (TranSCRIMP)

**TranSCRIMP** is a CPU transprecision implementation of SCRIMP based on FlexFloat library. The key idea of SCRIMP [202] is to minimize the computation by exploiting the fact that the dot product can be updated incrementally for subsequences in the diagonal of the distance matrix,  $D$  (see Figure 2.3). Consequently, the dot product can be expressed as follows:

$$Q_{i,j} = Q_{i-1,j-1} - T_{i-1}T_{j-1} + T_{i+m-1}T_{j+m-1} \quad (6.1)$$

The baseline implementation for our transprecision SCRIMP algorithm, **TranSCRIMP** in Figure 6.3, is a vectorized-parallel version presented in [58]. It first precomputes the mean and standard deviation of each time series subsequence (line 1), and initializes the matrix profile array (line 3). Then, the distances between pairs of subsequences are calculated following the diagonals of the distance matrix (lines 4-26). The for loop is fully parallelized, with each thread computing a random subset of diagonals provided by their indices in the *diag* array in line 5.

For the first element of the diagonal, we need to compute the dot product of the first pair of subsequences (line 6) in parallel. The rest are updated following Equation 6.1. For the proper vectorization of the dot product update, the algorithm separates the calculation of the diagonal in several steps: 1) the products

---

```

1:  $\mu, \sigma \leftarrow \text{computeMeanDev}(T, m);$   $\triangleright$  precalculation of statistics
2:  $\text{vectFact} \leftarrow 8/\text{sizeof}(\text{datatype});$   $\triangleright$  8 bytes (double) by the size of the
   transprecision datatype
3:  $P \leftarrow \infty;$   $\triangleright$  matrix profile array initialization
4: for  $\text{idx} \leftarrow \text{tid} * \text{numDiag}$  to  $(\text{tid} + 1) * \text{numDiag} - 1$  do  $\triangleright$  parallel for
5:    $i \leftarrow 0; j \leftarrow \text{diag}_{\text{idx}};$   $\triangleright$  diag array contains the diagonals assigned to each
   thread
6:    $q \leftarrow \text{dotProduct}(T_{i,m}, T_{j,m});$   $\triangleright$  compute dot product for the first diagonal
   element (SIMD)
7:    $d \leftarrow \text{dist}(m, q, \mu_i, \sigma_i, \mu_j, \sigma_j);$   $\triangleright$  compute distance for the first element of
   diagonal
8:   if  $d < P_i$  then  $\triangleright$  update matrix profile and index arrays
9:      $P_i \leftarrow d; I_i \leftarrow j;$ 
10:  if  $d < P_j$  then  $\triangleright$  distance matrix is symmetric
11:     $P_j \leftarrow d; I_j \leftarrow i;$ 
12:   $i \leftarrow i + 1;$ 
13:  for  $j \leftarrow \text{diag}_{\text{idx}}$  to  $\text{size}(P)$  do
14:    for  $k \leftarrow 0$  to  $\text{vectFact} - 1$  do  $\triangleright$  compute new dot product elements in
    parallel (SIMD)
15:     $qs_k \leftarrow t_{i+m-1+k}t_{j+m-1+k} - t_{i-1+k}t_{j-1+k};$ 
16:     $qs_0 \leftarrow qs_0 + q$   $\triangleright$  update with the first dot product of the diagonal
17:    for  $k \leftarrow 1$  to  $\text{vectFact} - 1$  do  $\triangleright$  update remaining dot products
    (sequentially)
18:     $qs_k \leftarrow qs_k + qs_{k-1};$ 
19:     $q \leftarrow qs_{\text{vectFact}-1};$   $\triangleright$  store last dot product for next iteration
20:    for  $k \leftarrow 0$  to  $\text{vectFact} - 1$  do  $\triangleright$  compute distances in parallel (SIMD)
21:     $ds_k \leftarrow \text{dist}(m, qs_k, \mu_{i+k}, \sigma_{i+k}, \mu_{j+k}, \sigma_{j+k});$ 
22:    if  $ds_k < P_{i+k}$  then  $\triangleright$  update matrix profile and index arrays
23:       $P_{i+k} \leftarrow ds_k; I_{i+k} \leftarrow j + k;$ 
24:    if  $ds_k < P_{j+k}$  then
25:       $P_{j+k} \leftarrow ds_k; I_{j+k} \leftarrow i + k;$ 
26:     $i \leftarrow i + \text{vectFact};$ 

```

---

Figure 6.3: Transprecision SCRIMP (TranSCRIMP) algorithm (transprecision operations highlighted).

in Equation 6.1 are calculated in parallel for *vectFact* elements of the diagonal (lines 14-15); 2) the previous dot product,  $q$ , is added to the element calculated in step 1) (line 16); 3) the subsequent dot products are updated sequentially using the previous ones (lines 17-18) saving the last one in  $q$  for the next iteration of the diagonal (line 19); 4) distances are calculated in parallel (lines 20-21); and 5) the profile is updated in parallel as well (lines 22-25).

The vectorization factor, *vectFact*, is given by the transprecision datatype width with respect to that of double precision (line 2). We highlight the lines of code which are to be executed using the transprecision approach. The algorithm is able to work in either one precision configuration or a mixed-precision one, thus the green (lower precision) and red (higher precision) marks.

Rather than using only double and single precision, we define a high and a low precision that can be set to any possible exponent and mantissa configuration, which provides further accuracy analysis opportunities. For the algorithms involved in this study, we store the time series codified with high precision and both the matrix profile and the precalculated statistics with low precision. For **TransSCRIMP**, Figure 6.3 shows the lines of code that works with high precision highlighted in red, and those which work with low precision highlighted in green. The dot product calculations use high precision as they may require a larger numeric range. Distance calculation as well as calculations with means and standard deviations are performed with less precision.

### 6.2.3. Transprecision SCAMP-CPU (TransSCAMP)

**TransSCAMP** is a CPU transprecision implementation of **SCAMP** based on **FlexFloat** library. Whereas following a similar computation scheme to **TransSCRIMP**, **TransSCAMP** replaces the sliding dot product with a mean-centered-sum-of-products in order to reduce the floating-point rounding errors and the number of operations required [203]. The following equations can be precomputed in  $O(n - m + 1)$  time, with  $n - m + 1 = l$  being the length of  $P$ :

$$df_i = \frac{T_{i+m-1} - T_{i-1}}{2}, \quad 0 < i < l \quad (6.2)$$

$$dg_i = T_{i+m-1} - \mu_i + T_{i-1} - \mu_{i-1}, \quad 0 < i < l \quad (6.3)$$



$$ssq_i = \begin{cases} \sum_{k=0}^{m-1} (T_k - \mu_0)^2, & i = 0 \\ ssq_{i-1} + (T_{i+m-1} - \mu_i + \\ + T_{i-1} - \mu_{i-1})(T_{i+m-1} - T_{i-1}) & 0 < i < l \end{cases} \quad (6.4)$$

$$\sigma_i = \sqrt{ssq_i}, \quad 0 \leq i < l \quad (6.5)$$

Equations 6.2 and 6.3 are terms used in the covariance update of Equation 6.6, and the standard deviation (L2-norm of subsequence  $T_{i,m} - \mu_i$ ) calculated in Equations 6.4 and 6.5 are used for the Pearson correlation coefficient depicted by Equation 6.7. Note the exclusion zone in the limits of Equation 6.6 given by  $\frac{m}{4}$ .

$$\sigma_{i,j} = \begin{cases} \sum_{k=0}^{m-1} (T_k - \mu_0)(T_{k+j} - \mu_j), & i = 0, \frac{m}{4} < j < l \\ \sigma_{i-1,j-1} + df_i dg_j + df_j dg_i, & i > 0, \frac{m+4}{4} < j < l \end{cases} \quad (6.6)$$

$$P_{i,j} = \frac{\sigma_{i,j}}{\sigma_i \sigma_j} \quad (6.7)$$

$$D_{i,j} = \sqrt{2m(1 - P_{i,j})} \quad (6.8)$$

The matrix profile can be derived incrementally for each diagonal of the distance matrix, Equation 6.6, from the calculation of the covariance of two subsequences of the first row (first piece in Equation 6.6). The Pearson correlation coefficient in Equation 6.7 can be computed in fewer operations and it is more robust than the Euclidean Distance used by SCRIMP [203]. Equation 6.8 calculates the distance from the Pearson coefficient in  $O(1)$ . For **TransSCAMP**, we compute the covariance in Equation 6.6 at high precision, which may have a large numeric range depending on the series. The correlation in Equation 6.7, which varies between -1 and 1, is computed at low precision.

#### 6.2.4. Transprecision SCAMP-FPGA (TransSCAMPfpga)

**TransSCAMPfpga** is an FPGA transprecision implementation of SCAMP based on the cpfp-FPGA library. We include **TransSCAMPfpga** as part of TraTSA's backend to speed up the evaluation of transprecision time series analysis in those research environments where FPGAs are available. **TransSCAMPfpga** is tuned for a Xilinx Alveo U50 FPGA board, which includes High-Bandwidth-Memory (HBM)

[94]. However, thanks to the C++-based HLS implementation, it is easy to port to higher-end Alveo boards or, with modest effort, to other FPGA platforms, as Intel Altera. We present an overview of **TranSCAMPfpga** in Figure 6.4, including the kernels inside the two SLRs (Super Logic Regions) of the FPGA and the HBM memory.

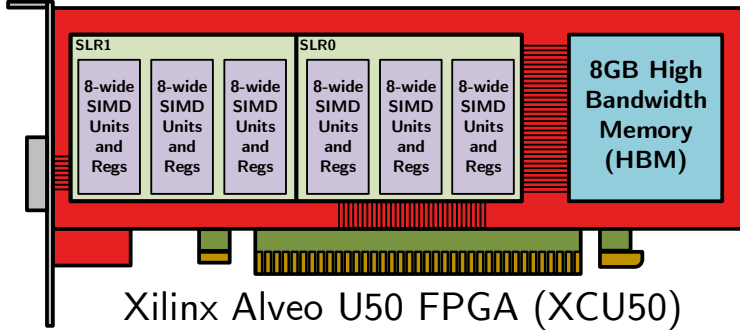


Figure 6.4: FPGA implementation overview. **TranSCAMPfpga** is composed of six kernels optimized for a Xilinx Alveo U50 FPGA that compute transprecision SCAMP algorithm using the data in the HBM.

We develop **TranSCAMPfpga** using Xilinx Vitis 2020.2 and a C++ HLS approach. Our implementation consists of 1) a host side code, and 2) an FPGA side code (kernels), that compute matrix profile in a parallel-vectorized manner. To avoid the use of synchronization primitives, each kernel has exclusive access to its private matrix profile.

On the one hand, the host code, which is executed in the host CPU, is in charge of 1) allocating the corresponding data structures in the FPGA and transferring the input vectors (i.e., time series and statistic data) from the host to the HBM cubes of the FPGA via PCI-Express, 2) defining the parallel scheduling partitioning, 3) invoking the FPGA, 4) transferring the output vectors (i.e., matrix profiles) from the HBM cubes of the FPGA to the host via PCI-Express, and 5) performing the final matrix profile reduction.

On the other hand, the FPGA code is an HLS-based implementation that performs the transprecision computation of SCAMP. This FPGA implementation, intended for the evaluation of large time series (i.e., millions of elements), includes the possibility of defining a recalculation interval to reduce the accumulated errors across diagonals due to lower precision arithmetic. Concretely, instead of reusing the previous covariance to calculate the current one, it performs the complete centered-sum-of-products at certain interval. Regarding to

performance, we tune this FPGA implementation to benefit as much as possible from the available resources of the FPGA, exploiting the following techniques:

- **Parallelization across kernels.** Instead of having a huge kernel that computes the whole matrix profile of the given time series, we create several kernels (six in the case of the Alveo U50) which allows performing diagonal-level parallelization. Additionally, the use of six kernels efficiently exploits the HBM memory as accesses can be overlapped.
- **Vectorization inside kernels.** Instead of calculating one diagonal at a time, each kernel computes a set of diagonals (512) which allows exploiting loop unrolling techniques via arithmetic operator replication, and reusing data via *systolic arrays*. Figure 6.5 illustrates an example of the main advantage of vectorization using five diagonals. We notice that for the first elements of all diagonals, values of data structures need to be brought from the HBM memory. However, for the rest of the rows of all diagonals (as all diagonals increase  $i$  index at the same time), we only need to bring from memory one data value (the last one), regardless of the vectorization width. This can be achieved thanks to the systolic array approach, where previously fetched data are pushed forward to the next row while the new data value is read from memory at the same time. This operation is performed in parallel and takes only one step for all elements of the systolic array.
- **Wide memory accesses.** We pack data that is requested from memory to improve the efficiency of the bus (i.e., bringing 512 bits in each request) while performing burst accesses at the same time.

We present `TransCAMPfpga`'s execution flowchart in Figure 6.6, which comprises an external loop that iterates over diagonals in batches of 512, and an inner loop to go over those diagonals following a SIMD approach. Notice that the *calculate\_covariances* module is shared among two steps of the execution, which allows to save hardware resources as it is not used in all iterations of the *diag-calculation* loop.

Finally, Table 6.1 shows the total resource utilization numbers by `TransCAMPfpga` kernels in the Alveo U50 board. According to our observations using Vitis Profiler, the main bottleneck for the performance of this implementation is data movement. This means that increasing the number of arithmetic operations does not improve performance because 1) global clock frequency is reduced to enable proper functionality, and 2) those additional arithmetic operations require increasing the number of memory ports, which leads to routing errors due to network congestion.

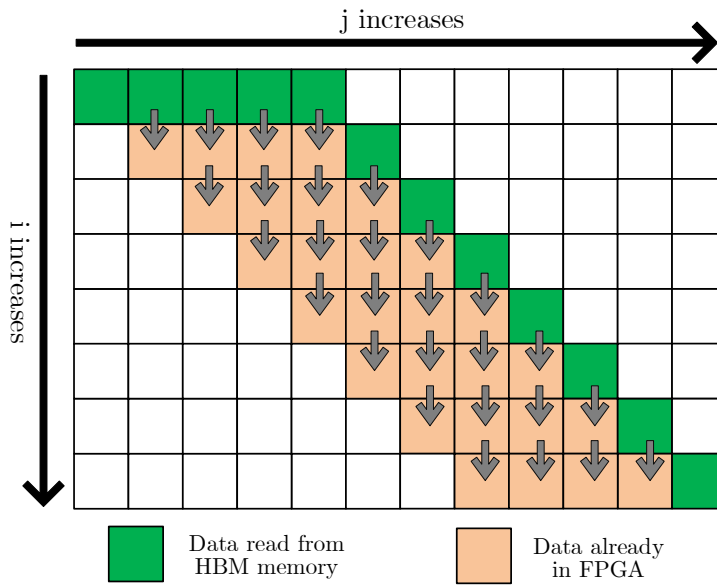


Figure 6.5: TransSCAMPfpga systolic array example.

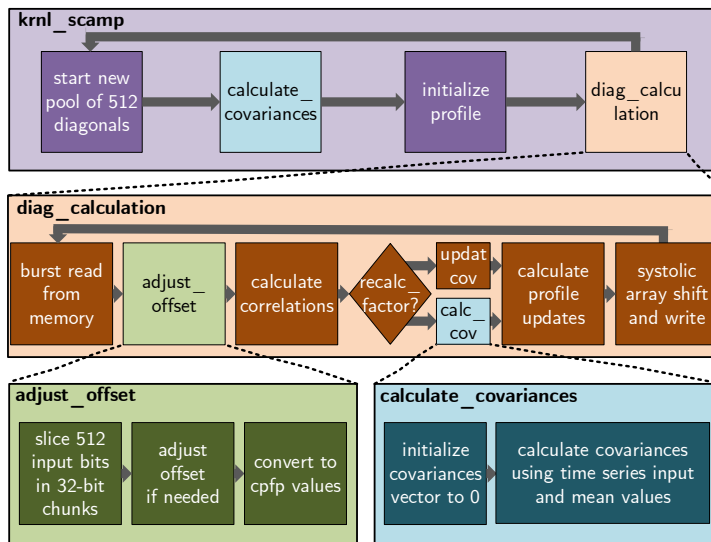


Figure 6.6: FPGA implementation diagram.

Table 6.1: TranSCAMPfpga kernel resource utilization

Parameter	Available	Used	Parameter	Available	Used
LUT	752672	308184 (41%)	URAM	640	0 (0%)
REG	1586939	494199 (31%)	BRAM	1164	714 (61%)
LUTm	389324	5802 (1.5%)	DSP	5936	828 (14%)

### 6.3. Top-K Accuracy Metric

Time series motifs [40] and discords [100] have been used for more than 15 years in the field of data mining for their capacity to find time series subsequences with special significance. In this section, we define these special subsequences and propose a metric to measure the accuracy in the detection of motifs and discords from two time series. Concretely, we next present the definitions of the motif and the Top-K motifs of a time series.

**Definition 6.4.** *The motif  $M_1$  of a time series  $T$  is the unordered pair of subsequences  $\{T_{i,m}, T_{j,m}\}$  which is the most similar among all possible pairs:*

$$M_1 = \{T_{i,m}, T_{j,m}\} \Leftrightarrow \text{dist}(T_{i,m}, T_{j,m}) \leq \text{dist}(T_{u,m}, T_{v,m})$$

$$\forall i, j, u, v; i \neq j, u \neq v.$$

■

**Definition 6.5.** *The Top-K motifs  $M_{1,K}$  of a time series  $T$  is the set of the first  $K$  motifs:*

$$M_{1,K} = \begin{cases} M_K \cup M_{1,K-1}, & K > 1 \\ M_1, & K = 1 \end{cases}$$

being  $M_K$  the motif ( $M_1$ ) of the time series  $T \setminus M_{1,K-1}, \forall K > 1$ .

■

We can define the discord and the Top-K discords of a time series in a similar way:

**Definition 6.6.** *The discord  $D_1$  of a time series  $T$  is the unordered pair of subsequences  $\{T_{i,m}, T_{j,m}\}$  which is the most dissimilar among all possible pairs:*

$$D_1 = \{T_{i,m}, T_{j,m}\} \Leftrightarrow \text{dist}(T_{i,m}, T_{j,m}) \geq \text{dist}(T_{u,m}, T_{v,m})$$

$$\forall i, j, u, v; i \neq j, u \neq v.$$

■

**Definition 6.7.** *The Top-K discords  $D_{1,K}$  of a time series  $T$  is the set of the first  $K$  discords:*

$$D_{1,K} = \begin{cases} D_K \cup D_{1,K-1}, & K > 1 \\ D_1, & K = 1 \end{cases}$$

being  $D_K$  the discord ( $D_1$ ) of the series  $T \setminus D_{1,K-1}, \forall K > 1$ . ■

Being  $MI_{1,K}$  and  $DI_{1,K}$  the Top-K sets of unordered pair of indices  $\{i, j\}$  of the unordered pair of subsequences in  $M_{1,K}$  and  $D_{1,K}$ , respectively, we define the Top-K Motif/Discord Accuracy in the following way:

**Definition 6.8.** *The Top-K Motif (Discord) Accuracy  $AM$  ( $AD$ ) of a time series  $T$  with respect to another time series  $TT$  is the number of coincidences among the unordered pairs in  $MI_{1,K}^T$  and  $MI_{1,K}^{TT}$  ( $DI_{1,K}^T$  and  $DI_{1,K}^{TT}$ ):*

$$AM_{1,K}^{T \rightarrow TT} = |MI_{1,K}^T \cap MI_{1,K}^{TT}|$$

$$AD_{1,K}^{T \rightarrow TT} = |DI_{1,K}^T \cap DI_{1,K}^{TT}|$$
■

Definition 6.8 is pointless when  $T$  and  $TT$  are time series from different applications. However, it can be useful when we want to know the degree of coincidence of the matrix profile of a time series and that of the same time series codified with less precision or affected by a transform operator. We can get to know if, after a transformation of the original time series, the matrix profile ends up unveiling the same motifs and discords, or a significant subset of them.

## 6.9. Evaluation

### 6.9.1. Methodology

We evaluate TraTSA's **TranSCRIMP**, **TranSCAMP** and **TranSCAMPfpga** backends in terms of accuracy, performance and energy savings. First, we compute the reference SCRIMP and SCAMP matrix profiles with double and single floating-point precisions using the native C++ implementations. Second, we use the FlexFloat [172] library on an Intel Xeon Phi 7210 “Knights Landing” manycore processor [93] with 64 cores and 256 threads to evaluate **TranSCRIMP** and **TranSCAMP** on relatively short<sup>2</sup> time series (<200K elements). Third, we

<sup>2</sup>Notice that both SCRIMP and SCAMP perform profile calculations ( $P_i$ ) based on a previous result along the diagonals (dot product and covariance, respectively). In such scenarios, it

use the `cpfp-FPGA` library on a Xilinx Alveo U50 FPGA board to evaluate `TransSCAMPfpga` for larger time series ( $>200K$  elements). Fourth, we present performance comparisons between `TransSCAMP` and `TransSCAMPfpga` using different computing platforms. Finally, we compare the energy consumption in two ways. On the one hand, we study the benefits of the hardware-based solution (`TransSCAMPfpga`) against the software emulation one (`TransSCAMP`). To do so, we obtain the Joules consumed by the FPGA using Xilinx’s *xbutil* [43] tool, and we use *rapl-tools* [102] to obtain the Joules consumed by the CPU implementations. On the other hand, we use a transprecision FPU [129], evaluated as part of RISC-V processors and suitable for FPGAs and ASICs, to estimate the potential benefits of a transprecision FPU with respect to a double-precision FPU, which are based on the operation breakdown statistics and the energy per operation numbers given in [129].

We use real-data time series and the Top-K Accuracy metric of Definition 6.8 to evaluate TraTSA. Table 6.2 summarizes the parameters of the time series we use for the experiments. *Song* corresponds to the song *London Bridge is Falling Down* [195] converted into Mel-frequency Cepstral Coefficients, which are commonly used in speech recognition [147]. *ECG\_short* and *ECG* are extracted from an electrocardiogram signal from the European ST-T Database [171]. We select the 180000 and 1800000 first samples of the V4 electrode from ECG 0103, respectively. *Power\_short* and *Power* are two time series extracted from fridge-freezer power consumption numbers collected over a whole year in a set of UK households [195]. *Seismology\_short* and *Seismology* are two time series of seismic data collected by a seismograph in a geologically active region of the Long Valley Caldera, California [195]. We analyze the first 180000 and 1727990 samples of the original series, respectively. *Human Activity* comprises a time series with information of the optical flow of an actor performing activities, from picking up an object to talking on a mobile phone [195]. *Penguin Behavior* is a time series of penguin magnetometer telemetry [9]. *Speech* is an speech recorded time series where Ivan Fernandez (myself) reads a fragment (2 mins) of *El Quijote* book sampled at 16KHz. *IMU* is a time series extracted from the calibration of an Inertial-Measurement-Unit used in robotics [205]. Finally, *EPG* is insect electrical penetration graph data from [13].

---

is possible that accumulated errors provide mismatching results for large time series. However, as production implementations of SCRIMP and SCAMP are based on a tiled approach where the dot product or covariance is calculated from scratch for each 128K-512K elements of the diagonal, our evaluation and conclusions are also valid for time series of larger sizes. This is the main reason we include a recalculation factor in `TransSCAMPfpga`.

Table 6.2: Time series dataset parameterization

Time series	n	m	Max	Min	Scale
Song	20234	200	6.69	-56.48	1.00
ECG_short	180000	500	2.60	0.32	1.00
Power_short	180000	1325	14.0	0.00	0.10
Seismology_short	180000	50	6.96	-1.86	0.01
Human Activity	7997	120	2.51	-1.90	1.00
Penguin Behavior	109842	800	0.52	-0.21	1.00
Speech	1933944	16384	0.98	-1.00	1.00
ECG	1800000	512	3.39	-1.64	1.00
Power	1754985	1536	14.00	0.00	0.10
Seismology	1727990	64	23.29	-23.34	0.10
IMU	1756230	256	1.65	-2.87	1.00
EPG	2000000	16384	72.26	-61.30	10.00

### 6.9.2. Results

#### Short Time Series Accuracy

We use the Top-K Accuracy metric of Definition 6.8 to evaluate our proposals. The value of K will depend on the number of significant events of a given time series, which will be eventually determined by a domain expert.

One way of setting K is defining a profile threshold for both motifs and discords. Figures 6.7 and 6.8 present the Top-100 motif/discord accuracy with respect to double for a wide range of configurations of exponent and mantissa in **TransSCRIMP**, respectively. In contrast, Figures 6.9 and 6.10 present the same metrics in **TransSCAMP**, respectively. In most cases, single precision (8,23) provides 100% accuracy with respect to double precision. As can be noticed, most of the plots follow a square-like shape where the accuracy decreases dramatically after a given combination of exponent and mantissa. This may occur whether one or both of the following scenarios appear: 1) the range of the exponent has been exceeded; 2) the precision provided by the mantissa is not enough for the calculations.

Comparing both algorithms, we can observe that SCAMP is more robust and presents a better numeric stability than SCRIMP for all the datasets. This fact can be noticed for *Penguin Behavior*, where a slight decrease in the length of the mantissa makes SCRIMP fail in detecting events, while SCAMP provides more margin in this reduction. We find an outlier scenario for the time series *Power\_short* in Figure 6.9, where accuracy seems to decrease when the number



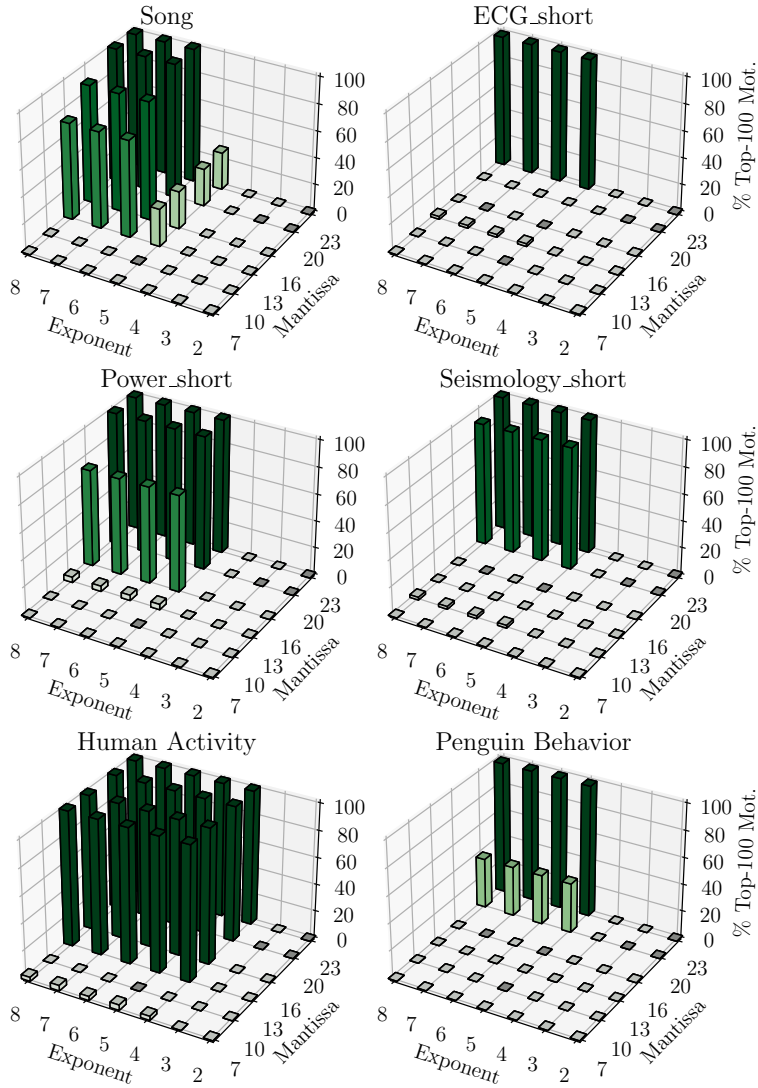


Figure 6.7: TranSCRIMP Top-100 motif accuracy with respect to double.

of mantissa bits increases from 20 to 23. The reason behind it is that, for this time series, there are several motifs with exactly the same profile value. Because of that, our sorting algorithm induces some order differences when comparing the transprecision version with respect to the reference one. However, from the

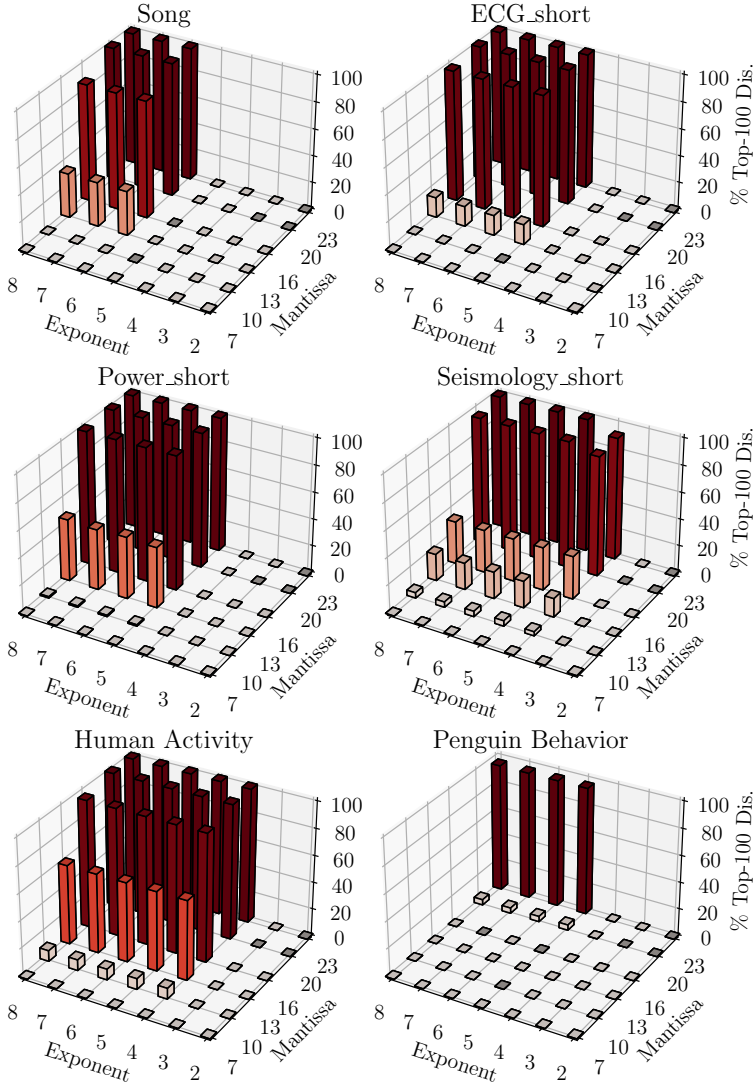


Figure 6.8: TranSCRIMP Top-100 discord accuracy with respect to double.

practical point of view, the accuracy of the mantissa 23 is as good as the mantissa 20 since all motifs are present in both of them. We conclude that SCAMP is a better candidate for transprecision computing since it provides similar accuracy with lower bit count requirements for the exponent and mantissa.

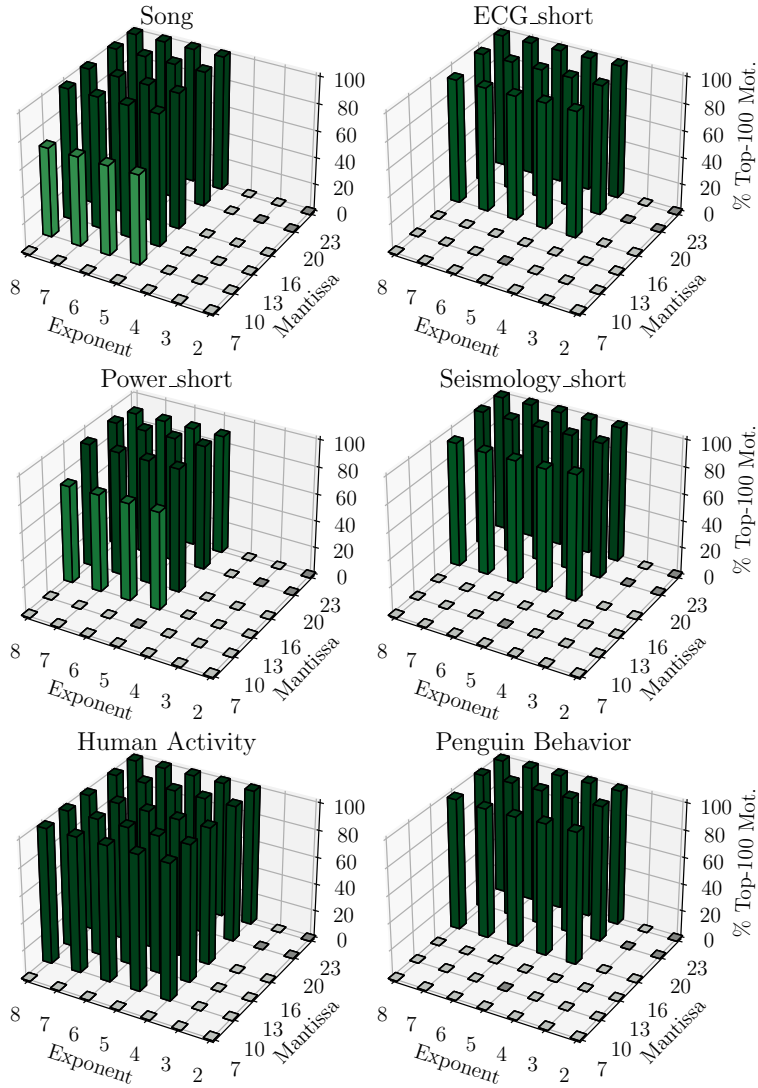


Figure 6.9: TranSCAMP Top-100 motif accuracy with respect to double.

We also evaluate how the profile is affected when reducing the exponent/mantissa bit count using TranSCAMP. Figure 6.11 presents the transprecision profile for the *Song* dataset and several exponent/mantissa combinations (i.e., (7,13), (7,10), (5,13), (5,7) in the figures) with respect to a double-precision profile.

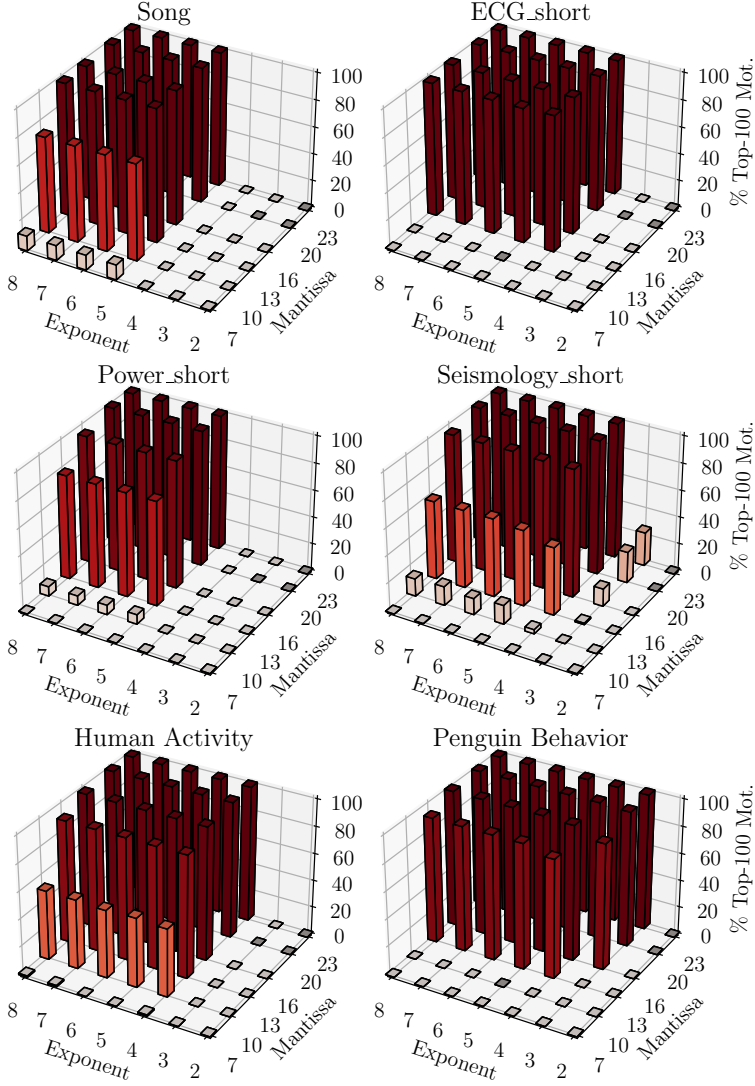


Figure 6.10: TranSCAMP Top-100 discord accuracy with respect to double.

We observe that for combinations of (5,13), (7,13) and (7,10) the profile is well-preserved (i.e., both curves overlap). We need to reduce the FP bit count to a lower value (e.g., (5,7)) to find mismatching results. We notice that, while in this (5,7) scenario the profile is somehow conserved with an offset, the profile

index is not providing the exact matches with respect to the reference (double) solution. This fact is explained because there are potentially many values very close to each other along the profile, which can lead to high chances of profile index interchanging even if profile values slightly differ from the double-precision reference.

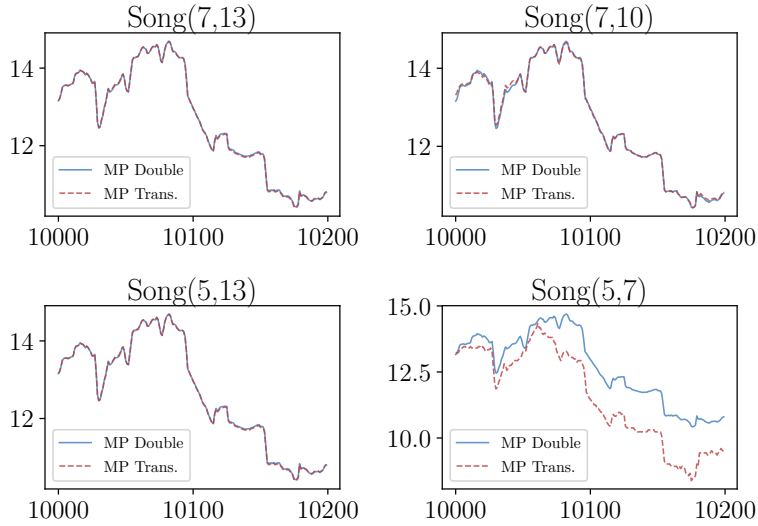


Figure 6.11: *TransSCAMP Song* profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal.

Figure 6.12 shows the transprecision profile with respect to double for the *ECG\_short* dataset. We observe that in this case the value of the mantissa plays a crucial role, since reducing it to a value lower than 13 bits leads to mismatching results (i.e., the transprecision curve is not shown at all since it contains NaN values). This fact is explained because the *ECG\_short* dataset is a very regular and monotonic time series, where profile values are smaller and mantissa becomes more relevant than exponent.

Figure 6.13 shows the transprecision profile with respect to double for the *Power\_short* dataset. We notice that the profile is well conserved for the (7,13), (7,10) and (5,13) exponent and mantissa combinations. However, the transprecision profile diverges from the oracle solution using lower bit counts, as shown in the (5,7) scenario. This dataset also benefits from larger mantissas since it follows a regular pattern, this explains why the (5,13) combination provides better results (i.e., the transprecision profile is better preserved) than the (7,10) one.

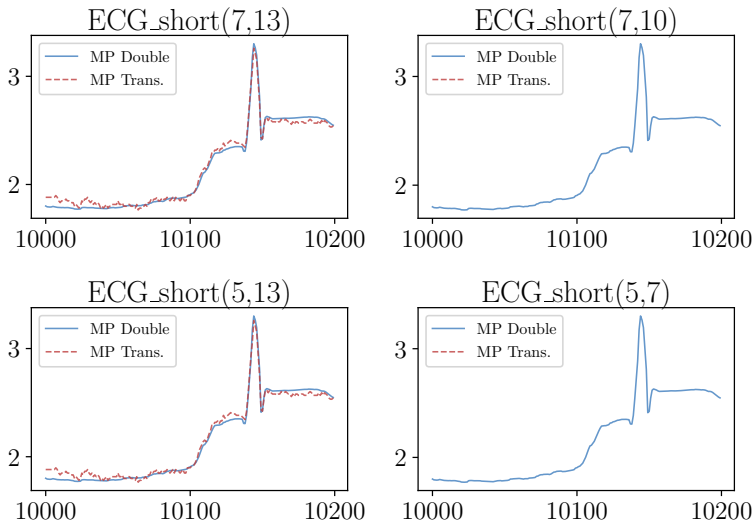


Figure 6.12: **TransSCAMP** *ECG\_short* profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal.

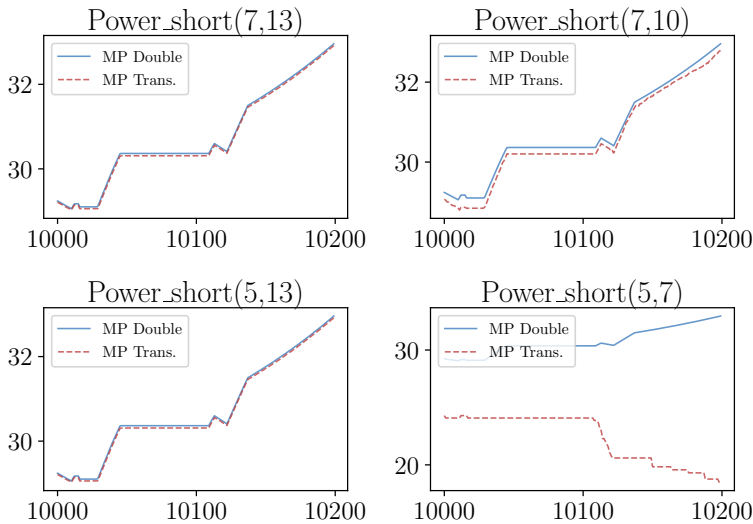


Figure 6.13: **TransSCAMP** *Power\_short* profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal.

Figure 6.14 shows the transprecision profile with respect to double for the *Seismology\_short* dataset. We observe a similar behavior to that in the *Song* dataset. While (7,13), (7,10) and (5,13) transprecision profiles are well-conserved with respect to double, lower mantissa bit count (e.g., 7 bits) leads to mismatching results and divergence.

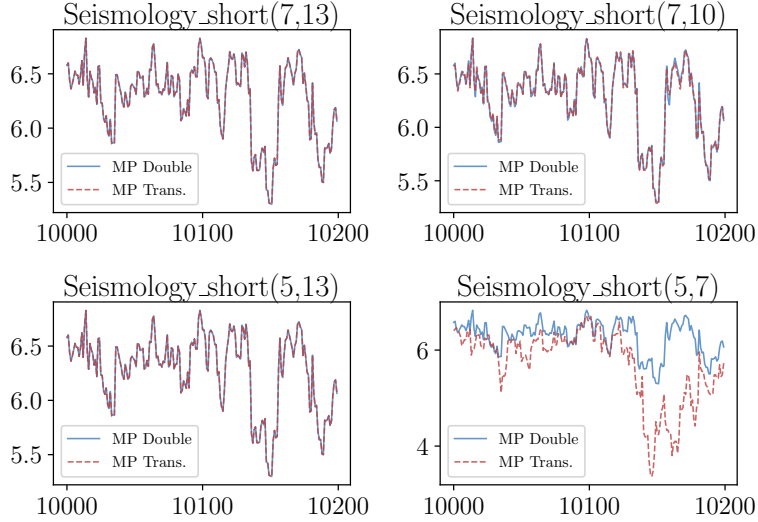


Figure 6.14: **TransSCAMP** *Seismology\_short* profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal.

Figure 6.15 shows the transprecision profile with respect to double for the *Human Activity* dataset. We notice that this dataset presents a higher sensitivity to mantissa bit count, since even with 13 bits the profile does not match perfectly. However, as evaluated with the Top-K metric, the overall accuracy of the results is not affected in a higher degree than the other datasets, since *peaks* in the profile are distinguishable among them.

Figure 6.16 shows the transprecision profile with respect to double for the *Penguin Behavior* dataset. This dataset presents a similar behavior to the *Power\_short* one, as they have similar numeric ranges (i.e., exponent/mantissa of (7,13), (7,10) and (5,13) provide well-conserved profiles while (5,7) fails).

We also evaluate a mixed-precision approach for **TransSCRIMP** and **TransSCAMP** to further tune the bit counts required in different parts of those algorithms. In this set of experiments, we restrict the exponent/mantissa configurations to the

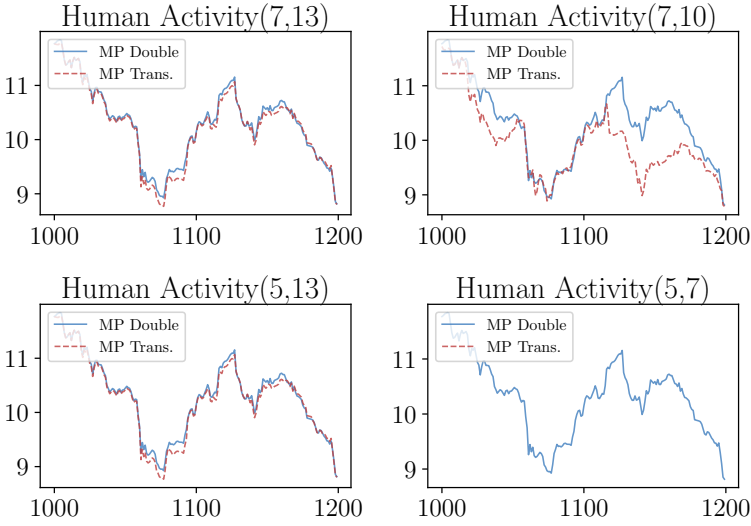


Figure 6.15: **TransSCAMP** *Human Activity* profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal.

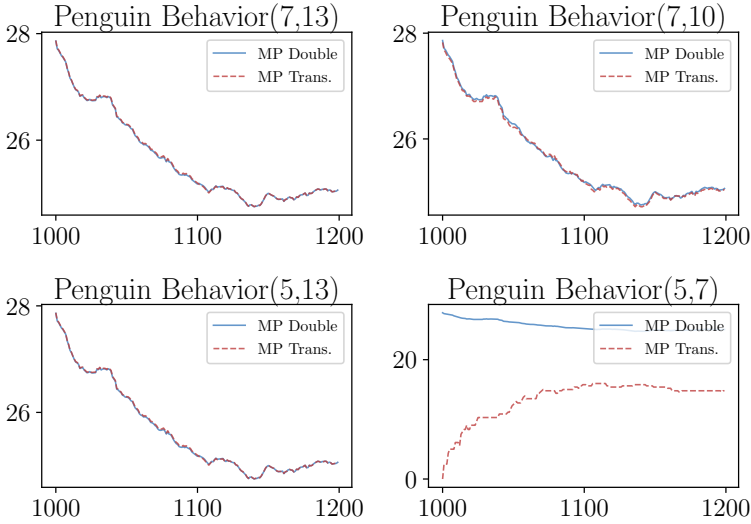


Figure 6.16: **TransSCAMP** *Penguin Behavior* profile with respect to double. The horizontal axis represents the index of the datapoints within the complete time series and the vertical axis represents the amplitude of the signal.



ones available in FPNew due to space limitations. Table 6.3 shows the results using mixed-precision for **TranSCRIMP** and **TranSCAMP**. We obtain better results with **TranSCAMP** for most datasets, as expected. It is worth noting the case of *Penguin Behavior*, where **TranSCRIMP** is not able to detect any of the events while **TranSCAMP** finds near 100% of them with the (8/23, 5/10) configuration.

Overall, our evaluation shows that the mixed-precision configurations provide better accuracy results than using only one reduced precision configuration throughout the code. The experiments suggest that a mixed configuration of (8/23, 5/10) can be used for the majority of the applications analyzed in this work. In this sense, we can observe that SCAMP results for *Song* present 70% accuracy for a 5/10 configuration (see Figures 6.9 and 6.10), whereas the accuracy increases to 95% using (8/23, 5/10) mixed precision (see Table 6.3). The rest of the series present a similar behavior with SCAMP: 0% accuracy for *ECG* with the 5/10 configuration, while up to 99% discord accuracy with the (8/23, 5/10)-mixed-precision configuration; 0% motif and 7% discord accuracy for *Power* with the 5/10 configuration, whereas 81% motif and 65% discord accuracy with mixed precision; and so on. SCRIMP presents a similar pattern as well.

Table 6.3: Mixed precision Top-100 accuracy results

T	High E/M	Low E/M	TranSCRIMP		TranSCAMP	
			Accuracy Mot/Disc	Accu. $\pm 10$ Mot/Disc	Accuracy Mot/Disc	Accu. $\pm 10$ Mot/Disc
Song	8/23	8/7	14/9	16/31	54/86	100/97
	"	5/10	38/0	99/0	95/99	100/100
	"	5/2	0/0	0/0	1/0	1/0
ECG	8/23	8/7	0/1	0/1	0/51	0/56
	"	5/10	10/57	10/60	25/99	30/100
	"	5/2	0/0	0/0	0/0	0/0
Power	8/23	8/7	47/31	67/95	39/25	78/99
	"	5/10	68/92	96/100	81/65	100/100
	"	5/2	0/0	0/0	0/0	0/0
Seis.	8/23	8/7	3/17	55/21	0/3	0/6
	"	5/10	7/68	86/70	12/40	15/45
	"	5/2	0/0	0/0	0/0	0/0
Hum.	8/23	8/7	72/24	80/63	91/84	99/92
	"	5/10	100/85	100/97	100/98	100/99
	"	5/2	0/3	0/4	0/0	0/1
Peng.	8/23	8/7	0/0	0/0	15/89	85/98
	"	5/10	0/0	0/0	81/99	100/99
	"	5/2	0/0	0/0	0/0	0/0

We observe that the  $(8/23, 5/2)$ -mixed-precision configuration does not yield meaningful results for any of the analyzed datasets, since the low bit count for the mantissa (only two bits) does not provide enough resolution for the correlation calculations. It can be noticed that detecting discords (anomalies) is more accurate than detecting motifs (similarities). This can be due to the fact that similarities are low values of the profile which require more precision than discords, that typically are higher ones. This fact takes more relevance in very monotonic time series, where most of the subsequences are similar to each other (e.g., the beats in an electrocardiogram – *ECG*).

The number of detected events and its significance must be eventually determined by a domain expert. Thus, we can think that presenting a time series subsequence to an expert, in its context, and moved slightly to the left/right might end up with the expert coming to the same conclusion as if the subsequence did not move. For that reason we introduce the concept of Accuracy  $\pm 10$  (see Table 6.3), which is the accuracy calculated for the motif/discord indices ranging in a  $\pm 10$  interval. Using this metric the accuracy peaks 100% for most of the datasets.

Mixed precision configurations are aimed at balancing the trade-off between accuracy and performance (time/energy) of the matrix profile algorithms. We can reach high peaks of accuracy with the  $(8/23, 5/10)$  configuration while reducing time and energy consumption as described in Section 6.9.2.

## Large Time Series Accuracy

We evaluate the accuracy of the larger time series using **TranSCAMPfpga**. To this end, we run the larger time series (between 1M and 2M elements) using the Alveo U50 FPGA, and evaluate the results using the Top-1000 accuracy metric (i.e., increasing K proportionally to time series length). We calculate time series statistics using double precision in the host side, which increases accuracy while it does not significantly affect the total execution time (statistics only take  $<1$  sec using 1 thread in an Intel Xeon Gold for 2M data points). This fact is particularly important in large time series, since statistics are calculated in an accumulative manner and single precision is not enough to avoid errors.

From the algorithmic point of view, **TranSCAMPfpga** differs from **TranSCAMP** in that 1) **TranSCAMPfpga** includes the possibility to recalculate the covariance value from scratch after certain number of elements of a diagonal, and 2) **TranSCAMP** allows us to define mixed-precision configurations. The first feature resets the accumulated error due to calculation reuse across the diagonal, which increases

accuracy in most datasets and allows to potentially evaluate time series of arbitrary length. We leave the mixed-precision evaluation of **TransSCAMPfpga** for future work.

Figures 6.17 and 6.18 show the results for motifs and discords, respectively.

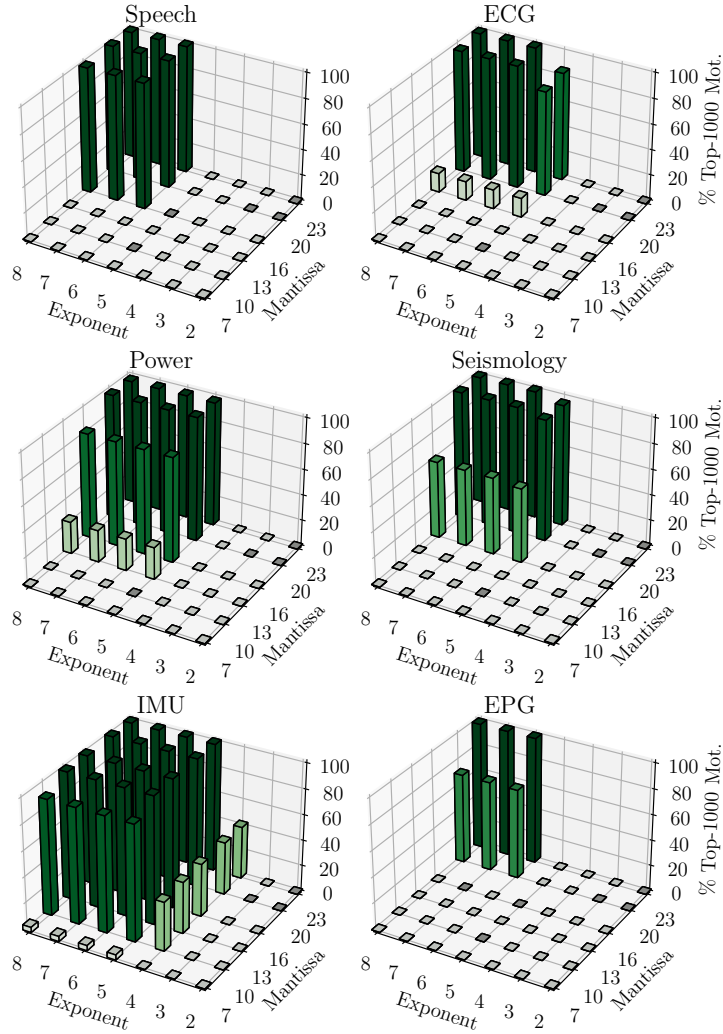


Figure 6.17: **TransSCAMPfpga** Top-1000 motif accuracy results with respect to double precision using a recalculation factor of 64K elements.

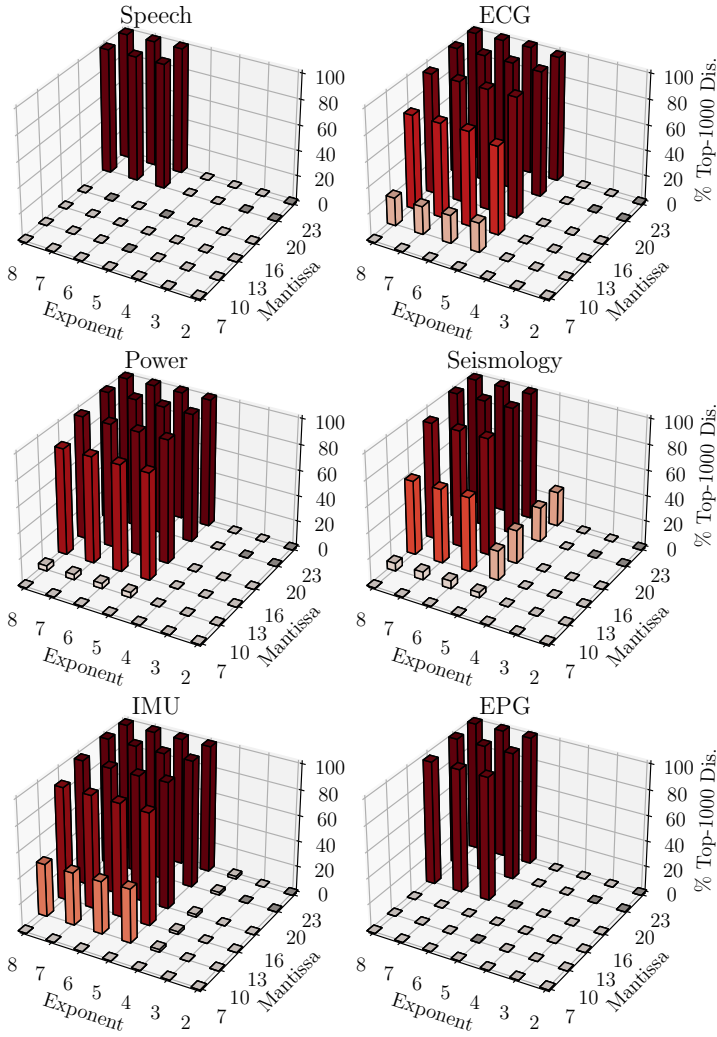


Figure 6.18: **TranSCAMPfpga** Top-1000 discord accuracy results with respect to double precision using a recalculation factor of 64K elements.

We observe that **TranSCAMPfpga** follows a similar behavior than the results obtained for its CPU counterpart implementation (**TranSCAMP** in Figures 6.9 and 6.10). We make several observations here. First, analyzing the larger time series, we observe that single precision still provides almost 100% accuracy for all

datasets when helped by covariance recalculation. Moreover, there is still margin to reduce the exponent and mantissa bit count below single precision depending on the application. Second, we find that the top part of the cubes are smaller in those series where the window size is larger (e.g., *Speech* or *EPG*). This fact is explained because larger window sizes have to deal with larger numbers, which may be out of the range of representation during calculations. And third, those datasets that present a randomness component (e.g., *IMU*) benefit from lower exponent and mantissa combinations, since the average distance is higher and there is no need for high precision to distinguish them.

We also evaluate the effect of changing the recalculation on the accuracy. Figure 6.19 shows the % of Top-1000 accuracy for the analyzed datasets with different exponent and mantissa combinations. We analyze *recalculation factors* of every 64K and 256K elements of the diagonal against turning this feature off.

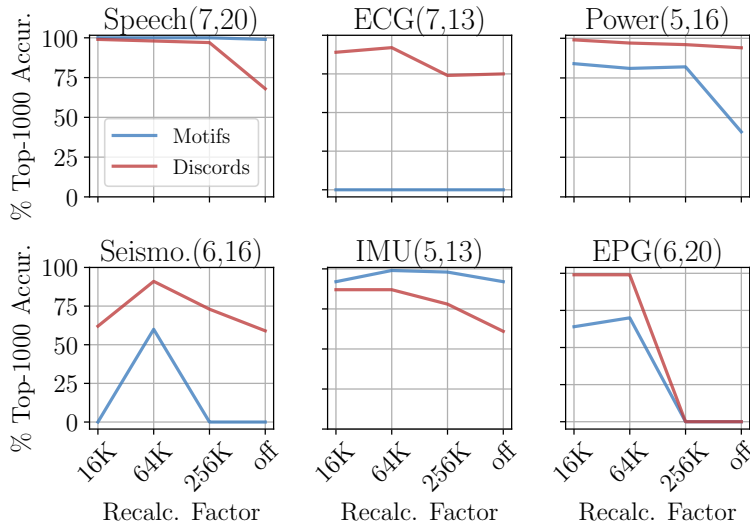


Figure 6.19: **TranSCAMPfpga** Top-1000 accuracy results with respect to double precision when varying the recalculation factor.

We observe that 64K elements is the sweet-spot since it provides even better results than lower ones (e.g., 16K) without having great impact on performance. Lower recalculation factors imply significant impact on performance as dot product has to be calculated more frequently. In contrast, turning this feature off for series of  $\approx 2$ M elements leads to mismatching results. We note that a recalculation factor of 16K for the *Seismology* time series leads to a counterintuitive result. The explanation behind it relies on the fact that the seismology time series is a

very periodic and regular time series without significant complexity (and, as a consequence, many motifs and discords are highly similar). This scenario, previously observed in [44], induces a random component when generating the Top-K comparison that takes special relevance when using the lower recalculation factor for this type of time series.

We show an example of modifying the recalculation factor in Figure 6.20, where we present the matrix profile of the *EPG* dataset with respect to the double-precision one. Notice that while recalculation factors of 16K and 64K provide accurate results (i.e., the curves overlap), a factor of 256K makes the profile always be close to 0 while turning this feature off leads to a NaN scenario.

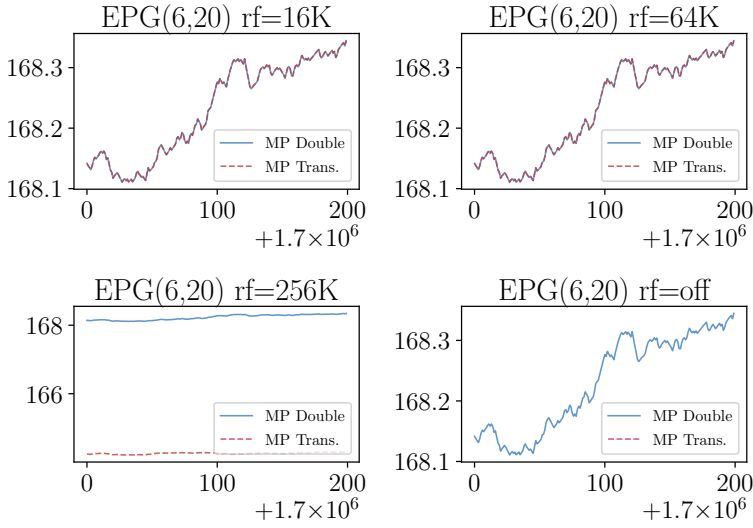


Figure 6.20: EPG matrix profile when varying the recalculation factor using *TransSCAMPfpga* for a given mantissa and exponent combination.

## TraTSA Performance

We evaluate the performance of TraTSA when running in different computing platforms. We focus our attention on comparing *TransSCAMP* versus *TransSCAMPfpga*, since SCAMP provides more accurate results than SCRIMP for a wider range of datasets.

Figure 6.21 presents the execution times obtained when computing the *Seis-mology\_short* series using a window of 512 elements and an exponent/mantissa

combination of 7/10. Our FPGA execution times also include the memory transfers from and to the FPGA. As can be noticed, **TranSCAMPfpga** in an Alveo U50 integrated in the Xeon Gold 6154 platform outperforms **TranSCAMP** in commodity servers by  $22.75\times$  when using a 72-core Xeon Gold, and by  $52.65\times$  when using a 64-core Intel Xeon Phi KNL. Compared to desktop computers, **TranSCAMPfpga** outperforms an Intel i7-8700 by  $126\times$  and by  $313\times$  an Intel i5-4570.

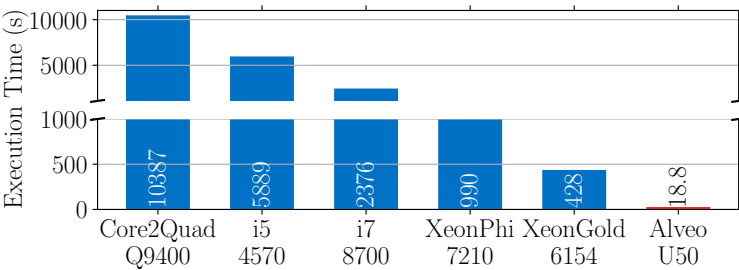


Figure 6.21: Execution time for different platforms when computing *Seismology\_short*, using a window size of 512 elements (exp. 7 man. 10).

Energy Savings

**Emulated vs Hardware Transprecision.** First, we show an energy consumption comparison in Table 6.4 for the platforms in Figure 6.21. As expected, the FPGA solution not only provides benefits in terms of performance but also reduced energy consumption. According to our measurements, the Alveo U50 consumes 0.47KJ when running **TranSCAMPfpga**, taking into account the memory transfers and idle time of the host CPU. This is way less than the CPU evaluated platforms, where, for example, the most efficient one, which is the Xeon Gold server, takes 82KJ ( $174\times$  more than the FPGA). This fact demonstrates that hardware-based solutions are way more efficient than the emulated ones.

Table 6.4: Energy consumption for different platforms when computing *Seismology\_short*, using a window size of 512 elements (exp. 7 man. 10)

Platform	i5 4570	i7 8700	XeonPhi 7210	XeonGold 6154	Alveo U50
Energy	180KJ	127KJ	132KJ	82KJ	0.47KJ

**Transprecision FPU.** Second, we present in Figure 6.22 the normalized energy consumption of a) **TranSCRIMP** with respect to SCRIMP double, b)

**TranSCAMP** with respect to **SCAMP** double and c) **TranSCAMP** with respect to **TranSCRIMP** with different precisions, respectively. As the energy is calculated with respect to the number of FPU operations performed by the algorithms, the proportion holds independently of the time series. We can observe a 60% energy reduction when using single precision instead of double for both algorithms. Furthermore, we can not only expect a reduction in time for this configuration due to an improved use of the memory hierarchy, but also because of SIMD capabilities, allowing two single precision elements computed at a time. We observe that single precision provides the same accuracy than double in the majority of cases (see Figures 6.7, 6.8, 6.9, 6.10, 6.17 and 6.18).

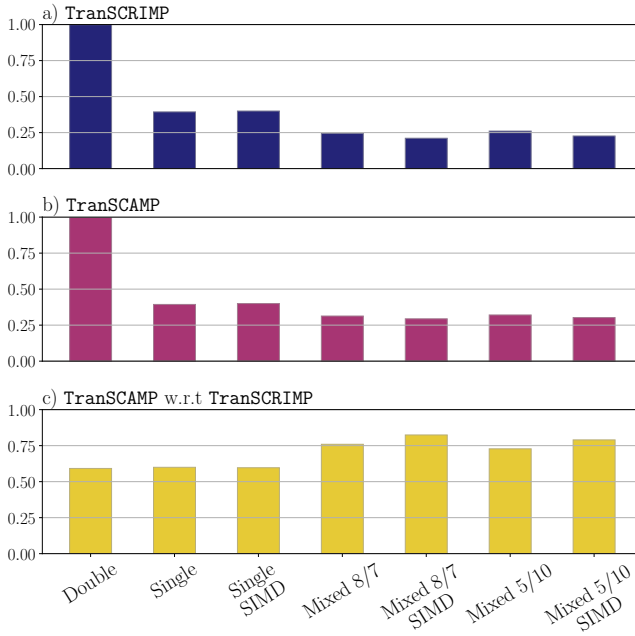


Figure 6.22: Normalized FPU energy using FPNew.

It is possible to reduce the energy consumption even further using mixed precision. Our **TransSCRIMP** and **TranSCAMP** mixed precision configurations can yield up to 50% and 25% energy reduction over the single precision approach respectively. The savings for **TransSCRIMP** are more pronounced since there are more operations computed in low precision. The dot product is computed in high precision but the distance in Equation 2.1 is calculated in low precision. However, the distance in **TranSCAMP** is given by the Pearson correlation coefficient



in Equation 6.7 which entails fewer operations. We confirm this fact when comparing **TranSCAMP** with respect to **TranSCRIMP** in plot c) of Figure 6.22, where **TranSCAMP** reduces the energy consumption between 18% and 40% for the same given precision. There are also a significant number of comparisons computed in low precision in both algorithms, although the energy cost of this operation is not as high as that of multiplications or sums, so the savings are not so high either. In contrast, the SIMD support of the FPU yields roughly the same energy numbers but opens the opportunity to improve the performance even more (up to 4 ops at a time).



UNIVERSIDAD  
DE MÁLAGA

# 7 Conclusions

---

Time series analysis algorithms are a critical workload for many important applications (e.g., to predict a heart attack or to analyze household power consumption). In this thesis, we study and characterize state-of-the-art time series analysis algorithms and find that they are bottlenecked by data movement in conventional architectures. Our final goal is to accelerate and make time series analysis computation as fast and energy-efficient as possible. To do so, we propose hardware and software solutions moving computation closer to memory, based on latest trends in computer architecture. This thesis makes four major contributions, as described next.

**① Optimization and Characterization.** Chapter 3 presents novel implementation of an state-of-the-art time series analysis algorithm tuned for an Intel Xeon Phi KNL architecture, provided with HBM. In this implementation we exploit multi-threading, vectorization, and the use of aggregated HBM plus DDR4 memory bandwidth. First, we propose the parallelization of the processing of diagonals in the algorithm, dynamically distributing their computation across the cores in a many-core machine. Secondly, we use privatization and reduction techniques to avoid unnecessary thread synchronization and improve the scalability. Finally, we propose the distribution of data across DDR4 and HBM, allocating private and more frequently accessed data in HBM and shared read-only data in DDR4. Our experiments show that our approach can improve performance in up to 190x with respect to sequential execution using the Intel Xeon Phi KNL. Furthermore, our techniques present better scalability than traditional lock-based synchronization mechanisms. This chapter is associated with the following publication:

Fernandez, I., Villegas, A., Gutierrez, E., & Plata, O. (2019). Accelerating time series motif discovery in the Intel Xeon Phi KNL processor. *The Journal of Supercomputing*, 75(11), 7053-7075.

**② Processing-Near-Memory Accelerator.** Chapter 4 presents NATSA, the first Near-Data-Processing (NDP) accelerator for time series analysis. NATSA 1) exploits the memory bandwidth of high-bandwidth memory (HBM) to analyze time series data at scale for a wide range of applications, 2) improves energy efficiency and execution time by using specialized low-power arithmetic units close to HBM memory, and 3) provides a novel workload scheduling scheme to prevent load imbalance and preserve the *anytime* property. NATSA outperforms the hardware platforms we evaluate in terms of performance, energy consumption and area requirements. This chapter is associated with the following publication:

Fernandez, I., Quislan, R., Gutiérrez, E., Plata, O., Giannoula, C., Alser, M., Gómez-Luna, J., & Mutlu, O. (2020, October). NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In 2020 IEEE 38th International Conference on Computer Design (ICCD) (pp. 120-129). IEEE.

**③ Processing-Using-Memory Accelerator.** Chapter 5 presents MATSA, the first MRAM-based Accelerator for Time Series Analysis. The key idea is to exploit magnetoresistive crossbars to enable energy efficient and fast time series computation in-memory. MATSA provides the following key benefits: 1) significantly increases parallelism exploiting column-level bitwise operations, and 2) reduces the overheads of data movement by performing computation in the memory cells. We evaluate three versions of MATSA to match the requirements of different environments and perform a design space exploration. MATSA significantly improves performance and energy over commodity and PNM platforms. This chapter is associated with the following under-review publication:

Fernandez, I., Manglik, A., Giannoula, C., Quislan, R., Ghiasi, N. M., Gómez-Luna, J., Gutiérrez, E., Plata, O., & Mutlu, O. (2022). Accelerating Time Series Analysis via Processing using Non-Volatile Memories. arXiv preprint arXiv:2211.04369.

**④ Transprecision Framework.** Chapter 6 studies how time series analysis benefits from a transprecision approach, and introduces TraTSA, a framework that allows defining the exact needed precision according to the requirements of the specific application. The proposed TranSCRIMP, TranSCAMP and TranSCAMPfpga implementations will help the community to design energy-efficient time series analysis solutions based on transprecision RISC-V processors, FPGAs or ASICs while minimizing area and power requirements. Our FPGA-based solution is  $22.75\times$  faster than a 72-core Xeon server thanks to the hardware transprecision support and the use of optimization techniques. Additionally, we study how matrix profile algorithms can benefit from an already

presented transprecision FPU. In this sense, our analysis reveals that, for a variety of applications, the energy consumption of the matrix profile algorithms is reduced up to  $3.3\times$  compared with double precision while obtaining accurate results. This chapter is associated with the following publications:

Fernandez, I., Quislan, R., Gutierrez, E., & Plata, O. (2020, September). Energy-Efficient Time Series Analysis Using Transprecision Computing. In 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (pp. 83-90). IEEE.

Fernandez, I., Quislan, R., Gonzalez-Navarro, S., Gutierrez, E., & Plata, O. (2022). TraTSA: A Transprecision Framework for Efficient Time Series Analysis. *Journal of Computational Science*, 63, 101784.

## 7.1. Future Work

To conclude this thesis, we would like to propose the following future research lines inspired on our work.

As mentioned in Chapter 3, the main bottleneck to accelerate time series analysis in commodity systems is data movement. This limitation occurs between main memory and processing units, and can be mitigated via Processing-Near-Memory or Processing-Using-Memory approaches. However, these scenarios assume that data is already located and properly arranged in main memory. Time series usually comprise large amounts of input data that can potentially be hosted in storage devices. The communication of those storage devices with main memory is usually slower than the one between main memory and the processing units. Based on that, *our first proposed research line is to explore Processing-In-Storage approaches to further optimize and accelerate the time series analysis computation.*

As mentioned in Chapter 6, another way to alleviate the impact of data movement is to reduce the amount of data to move. One way to do so is the use of transprecision computing, reducing the number of bits that operands take in memory and also when performing the actual operations. While we evaluate how SCRIMP and SCAMP algorithms benefit from this approach, there are many other algorithms (e.g., DTW) that could be good candidates. Based on that, *our second proposed research line is to explore transprecision computing approaches in other time series analysis algorithms, reducing the amount of data to move during computation.*



UNIVERSIDAD  
DE MÁLAGA

# Apéndice A

## Resumen en español

---

### A.1. Introducción

La explosión de la era del Internet de las Cosas y del Big Data ha dado lugar a la generación continua de una cantidad muy grande de datos, que cada vez es más difícil de almacenar y analizar [112]. Los pequeños sensores y dispositivos producen una parte importante de estos datos [200], que incluyen observaciones (por ejemplo, temperatura, voltaje o sonido) muestreadas a lo largo del tiempo. Esta colección de datos también se conoce como *serie temporal*, una representación de datos común en casi todas las disciplinas científicas y aplicaciones empresariales [167], por ejemplo, la epidemiología, la genómica, la neurociencia, las ciencias ambientales o los mercados de valores. El *análisis de series temporales* (AST) divide las series temporales en *subsecuencias* de puntos de datos consecutivos para extraer información valiosa.

Para encontrar subsecuencias de interés, los algoritmos de AST definen una métrica de distancia, por ejemplo, la distancia euclidiana (ED) o la deformación temporal dinámica (*dynamic time warping* en inglés, o DTW). Esta métrica de distancia representa el grado de similitud de dos subsecuencias dadas, lo que significa que cuanto menor sea el valor de la distancia, más similares son. Basándose en la métrica de distancia, el algoritmo AST clasifica las subsecuencias que tienen una distancia baja como *motivos* [148] (similitudes) y una distancia alta como *discordantes* [100] (anomalías). Esta clasificación es un paso crítico antes del análisis posterior mediante algoritmos específicos del dominio o expertos humanos, así que ***AST es una herramienta de vital importancia y generalidad.***

Los algoritmos de AST pueden utilizarse para filtrar aquellas subsecuencias que coincidan con un comportamiento esperado, dejando sólo las anomalías para el algoritmo específico del dominio, que es computacionalmente costoso (por ejemplo, [126]), es decir, sólo una pequeña cantidad de datos críticos que necesitan ser analizados más a fondo. Un ejemplo es SquiggleFilter [50], un acelerador basado en AST que procesa la salida del secuenciador Mini-ION y filtra todo excepto las secuencias de interés, reduciendo el cómputo en más de un 85 %. Sin embargo, las optimizaciones de SquiggleFilter se basan en una versión modificada de DTW que pierde generalidad para otras aplicaciones. La Figura 1.1 describe el proceso de filtrado basado en un flujo de datos de ejemplo para datos de electrocardiograma y un algoritmo AST genérico.

**Punto de Motivación I:** El AST es útil en muchos dominios, pero las propuestas de aceleración anteriores se basan principalmente en modificaciones de los algoritmos que pierden generalidad.

Profundizando en el punto de vista de la implementación, encontramos que *Matrix Profile* [196] (MP) es el conjunto de herramientas del estado del arte para realizar AST, que comprende tanto algoritmos de distancia euclidiana como de deformación temporal dinámica. Hay varias implementaciones de MP basadas en CPU, GPU e incluso FPGA en la literatura [196, 86, 202, 204]. Estos enfoques de CPU, GPU y FPGA son extremadamente paralelizables, usando simples operaciones aritméticas sobre los datos para calcular las distancias. En este sentido, el enfoque de paralelización más sencillo consiste en repartir todas las distancias a calcular entre los recursos computacionales ya que -en principio- no hay dependencias entre ellos. Este esquema proporciona tantos flujos de ejecución como subsecuencias tengan las series temporales de forma totalmente independiente, lo que es mucho más que el número de núcleos disponibles en las CPUs y GPUs comerciales. Para acelerar las implementaciones paralelas de AST en esas arquitecturas, aumentar el número de núcleos parece ser un enfoque prometedor.

Con esta oportunidad en mente, primero caracterizamos una implementación de MP en una máquina multinúcleo (Intel Xeon Phi KNL) y descubrimos que el número de núcleos no es el único cuello de botella para AST. Concretamente, observamos que la escalabilidad se detiene a partir de un cierto número de núcleos y concluimos que *el rendimiento y la eficiencia energética de los algoritmos de AST se ven fuertemente afectados por el movimiento de datos*. Esta observación se basa en 1) la baja intensidad aritmética del algoritmo y 2) la necesidad de acceder a grandes cantidades de datos de series temporales desde la memoria. En otras palabras, la cantidad de cálculo por acceso a los datos no es suficiente para ocultar la latencia de la memoria, por lo que el algoritmo está



limitado por la memoria en una arquitectura de CPU convencional. En este contexto, el Procesamiento-Cercano-Memoria (*Processing-Near-Memory* en inglés, o PNM) y el Procesamiento-Utilizando-Memoria (*Processing-Using-Memory* en inglés, o PUM) sitúan el cálculo más cerca de los datos, reduciendo así el impacto del movimiento de datos. Basándonos en esto, consideramos que las arquitecturas PNM y PUM son enfoques prometedores para acelerar el AST.

En las arquitecturas convencionales, el AST se ve limitado por: 1) el escaso número de núcleos con respecto al paralelismo potencial y 2) el movimiento de datos entre la memoria y las unidades de procesamiento.

### A.1.1. Aplicaciones de AST

El primer punto de motivación de esta tesis se basa en que el AST constituye una de las primitivas de minería de datos más importantes gracias a su generalidad en la detección de anomalías y similitudes para una amplia gama de aplicaciones. Nótese que esta característica deseable es posible gracias a la naturaleza agnóstica del dominio de los algoritmos de AST, ya que sus resultados son simples valores de distancia. La Tabla 1.1 presenta algunos ejemplos de aplicaciones de AST.

En estadística, econometría, meteorología y geofísica, el objetivo principal del análisis de series temporales es la predicción y el pronóstico. Al mismo tiempo, en el procesamiento de señales, la ingeniería de control y la ingeniería de comunicaciones, se utiliza para la detección y estimación de señales. En la minería de datos, en el reconocimiento de patrones y en el aprendizaje automático, el descubrimiento de motivos y de discordancias se utilizan para la agrupación, la clasificación, la detección de anomalías y la previsión. Por último, la aplicación más importante del descubrimiento de motivos y discordancias en las series temporales es la agrupación de datos sísmicos y el descubrimiento de conjuntos de patrones de terremotos a partir del registro sísmico continuo. En consecuencia, el agrupamiento sísmico puede aplicarse a la reubicación de terremotos y a la monitorización de volcanes para ayudar a mejorar las evaluaciones de riesgos sísmicos y volcánicos.

**Beneficios de usar AST en aplicaciones completas.** AST es fundamental para ejemplos reales de aplicaciones de extremo a extremo. Por ejemplo, [90] predice el fallo circulatorio en las unidades de cuidados intensivos. En este escenario, el 90 % del tiempo de ejecución está dominado por el preprocesamiento de el AST, mientras que el 10 % restante es utilizado por la aplicación basada

en el aprendizaje automático para realizar la clasificación. También encontramos muchos otros ejemplos de casos de uso reales que se benefician de AST.

- **Detección de terremotos** [41]. El AST puede procesar los datos de un sismógrafo y detectar anomalías que pueden ser procesadas posteriormente con complejos algoritmos.
- **Electroencefalografía** [37]. Suponiendo un electroencefalógrafo que está monitoreando a un paciente, el AST puede ser utilizada para detectar anomalías y activar una alarma al respecto.
- **Detección de virus** [50]. Durante el proceso de ensamblaje del genoma, el *basecalling* es una tarea intensiva de computación que puede ser omitida para la mayoría de las lecturas utilizando un filtro basado en AST.

### A.1.2. Cuellos de Botella de AST

El segundo punto de motivación de esta tesis se basa en el hecho de que el AST tiene cuellos de botella en las arquitecturas comerciales. Para identificar esos cuellos de botella, primero desarrollamos una versión optimizada y vectorizada de una implementación del estado del arte de Matrix Profile. A continuación, la caracterizamos utilizando una máquina multinúcleo (Intel Xeon Phi KNL) que también incluye memoria de alto ancho de banda (HBM) junto con DDR4 convencional. En primer lugar, observamos que, aunque el número de hilos hardware (256) del Intel Xeon Phi es relativamente alto en comparación con las máquinas de gama baja, sigue habiendo muchos más flujos de ejecuciones independientes (miles o millones) que hilos. Sin embargo, la disponibilidad de hilos no es el único cuello de botella de AST. En la Figura 1.2, presentamos los resultados de rendimiento normalizados a 1 hilo (líneas) y el ancho de banda de memoria utilizado (barras) de esta implementación de MP basada en la distancia euclidiana, conocida como SCRIMP. Observamos que, cuando se utiliza la memoria DDR4, el rendimiento de SCRIMP no escala más allá de 32 hilos, mientras que el mayor ancho de banda de memoria proporcionado por la HBM permite a SCRIMP escalar hasta 128 hilos. Esto demuestra que el rendimiento de SCRIMP se satura en arquitecturas multinúcleo, ya que el ancho de banda alcanzable se satura cuando aumenta el número de hilos.

Para entender la causa de esta limitación de la memoria, realizamos el siguiente experimento. Generamos el análisis de la línea de techo para SCRIMP como mostramos en la Figura 1.3. Observamos que la intensidad aritmética de SCRIMP es significativamente baja. Este hecho confirma que la limitación de memoria de

SCRIMP se debe a la baja intensidad aritmética del algoritmo, que hace que los núcleos de procesamiento estén infrautilizados. Basándonos en todas estas observaciones, llegamos a la conclusión de que el rendimiento de la implementación de SCRIMP basada en la CPU de última generación se ve fuertemente limitado por el ancho de banda de la memoria disponible y el movimiento de datos.

Sin embargo, basándonos en la similar simplicidad operativa de los algoritmos de AST y en la enorme cantidad de datos que hay que procesar, planteamos la hipótesis de que los cuellos de botella son compartidos por otros enfoques de AST. Para demostrar esta afirmación, también caracterizamos una implementación de DTW en la misma arquitectura que SCRIMP y analizamos los cuellos de botella en otras plataformas.

**Plataformas CPU.** Perfilamos el rendimiento de DTW utilizando el mismo procesador multinúcleo (Intel Xeon Phi 7210) que utilizamos para SCRIMP, analizando la ejecución de 16K consultas de 8K elementos cada una que se comparan con una secuencia de referencia de 32K elementos. La Figura 1.4 presenta el gráfico *roofline* del experimento. Observamos que DTW sólo explota el 41 % del rendimiento máximo de enteros del sistema, es decir, 59 GINTOPS de 145 GINTOPS, y presenta una baja intensidad aritmética (0.55 INTOP/Byte). En segundo lugar, comprobamos que la huella de memoria de la ejecución es de aproximadamente 570 MB. El tráfico total de memoria generado durante el tiempo de ejecución es de  $\approx 267$  GB, lo que sitúa al núcleo por encima del ancho de banda máximo de la DDR4 la mayor parte del tiempo de ejecución.

**Plataformas GPU.** Encontramos que varios trabajos anteriores [201, 124, 35, 42, 53] proponen acelerar DTW usando GPUs. Sin embargo, estos trabajos se basan en optimizaciones de memoria compartida que sólo funcionan para ciertos tamaños de consulta pequeños. Para tamaños de consulta grandes, estas implementaciones o bien 1) no funcionan o bien 2) utilizan memoria global de alta latencia para mantener las estructuras de datos principales de DTW, lo que resulta en grandes penalizaciones de rendimiento. Nosotros desarrollamos una implementación basada en CUDA que admite tamaños arbitrarios.

**Plataformas FPGA.** Varios trabajos anteriores proponen [97, 98, 16] acelerar el DTW utilizando FPGAs. Sin embargo, la mayoría de ellas tienen una memoria integrada muy limitada, y los datos tienen que moverse por buses estrechos. Desarrollamos nuestra propia implementación de FPGA basada en HLS y descubrimos que 1) el número de unidades de cálculo no es suficiente para explotar el paralelismo inherente de DTW, y 2) las unidades de cálculo pasan la mayor parte de su tiempo esperando a que se sirvan los accesos a la memoria.

**Plataformas PNM/PUM.** Una forma de mejorar el paralelismo y reducir

los costes de movimiento de datos es realizar el cálculo cerca de la memoria o incluso utilizándola para realizar los cálculos. Analizamos varios enfoques en esas direcciones y detallamos sus ventajas e inconvenientes.

- PNM de propósito general. Este enfoque suele colocar pequeños núcleos de CPU en la misma matriz que la DRAM. La principal ventaja de este esquema es que la arquitectura puede utilizarse potencialmente para aplicaciones de propósito general. Sin embargo, esto tiene el coste de un paralelismo limitado. Evaluamos una PNM de propósito general (*upmem* baseline) en la Subsección 5.3.3, y nuestras evaluaciones muestran que esta arquitectura está limitada por el cómputo cuando realiza el cálculo DTW.
- PNM especializado. Este enfoque suele colocar un acelerador ASIC en la misma matriz que la DRAM. La principal ventaja de este enfoque es que los elementos de procesamiento están altamente optimizados para la carga de trabajo objetivo. Sin embargo, los datos siguen teniendo que pasar de la memoria al acelerador. Aunque se puede utilizar un acelerador ASIC en una arquitectura PNM, el rendimiento seguiría sufriendo un cuello de botella por el movimiento de datos entre la memoria y el acelerador, de forma similar a la PNM de propósito general.
- PNM basado en SRAM. Este enfoque utiliza matrices de memoria basadas en SRAM para realizar cálculos *in situ* (por ejemplo, *compute caches* [10]). Las principales ventajas de este enfoque son los altos niveles de paralelismo y la reducción del movimiento de datos. Lamentablemente, la SRAM sufre problemas de densidad y escalabilidad [75], además de ser vulnerable a la radiación.
- PUM basada en DRAM. Este enfoque utiliza matrices de memoria basadas en DRAM para realizar cálculos *in situ* (por ejemplo, *SIMDRAM* [79]). Sin embargo, este enfoque implica el movimiento interno de datos para realizar las operaciones, ya que los datos deben moverse a filas específicas habilitadas para el cálculo antes de realizar las operaciones en sí mismas. Además, la DRAM sufre problemas de volatilidad de datos y de lectura destructiva.

### A.1.3. Motivación y Contribuciones de la Tesis

La creciente demanda de procesamiento de datos pone sobre la mesa la necesidad de proporcionar plataformas informáticas que se ajusten a los requisitos de rendimiento. Además, el consumo de energía es una preocupación crítica creciente en esas plataformas. Como se indica en la Sección 1.1, el AST es una herramienta

útil para preprocesar la inmensa cantidad de datos que generan los dispositivos contemporáneos, por lo que la optimización y la eficiencia energética del AST es una cuestión de vital importancia. Esto nos motiva a estudiar cuidadosamente los algoritmos AST más avanzados y a proponer soluciones para mitigar sus cuellos de botella.

En los últimos tiempos, la comunidad de arquitectos de ordenadores ha identificado el movimiento de datos entre la memoria y las unidades de procesamiento como el principal cuello de botella para aumentar el rendimiento. Este hecho también es aplicable a la AST, como mostramos en la Sección 1.2. A partir de ahí, se han revisado las plataformas que sitúan el cálculo más cerca de los datos y se están realizando grandes esfuerzos de investigación en esta dirección. Distinguimos entre Procesamiento-Cercano-Memoria (PNM, por sus siglas en inglés), donde las unidades de cálculo se sitúan físicamente más cerca de la memoria, y Procesamiento-Utilizando-Memoria (PUM, por sus siglas en inglés), donde las celdas de memoria en sí mismas se utilizan para realizar el cálculo.

El objetivo principal de esta tesis es estudiar si las arquitecturas PNM y PUM son buenas candidatas para acelerar el AST y hacer que su cálculo sea energéticamente eficiente. Además, estudiamos el AST desde el punto de vista algorítmico, reduciendo el número de bits de coma flotante necesarios para el cálculo y optimizando los recursos.

Nuestras principales aportaciones en esta tesis y en las publicaciones relacionadas, que pretenden cumplir el objetivo de la misma, son las siguientes:

- **PhiTSA** [58]. Optimizamos y caracterizamos los algoritmos de AST más avanzados en una plataforma KNL Xeon Phi multiúcleo. Identificamos el movimiento de datos como el principal cuello de botella que 1) impide una mayor aceleración y 2) consume la mayor parte de la energía de ejecución.
- **NATSA** [57]. Proponemos un novedoso acelerador de Procesamiento-Cercano-Memoria para AST, conocido como NATSA. Este acelerador coloca unidades de procesamiento de punto flotante personalizadas cerca de la memoria de gran ancho de banda, explotando sus canales de memoria y la menor latencia de los accesos. NATSA mejora significativamente el rendimiento y el consumo de energía con respecto a los sistemas básicos al realizar el cálculo del AST basado en la distancia euclidiana.
- **MATSA** [En revisión]. Proponemos un novedoso acelerador de Procesamiento-Usando-Memoria para AST, conocido como MATSA. La idea clave es explotar los *crossbars* de la memoria magnetorresistiva para permitir un cálculo de series temporales rápido y eficiente desde el punto

de vista energético en la memoria, superando al mismo tiempo los problemas de durabilidad de otras tecnologías de memoria no volátil. MATSA: 1) aprovecha los altos niveles de paralelismo en el sustrato de la memoria explotando las operaciones aritméticas por columnas, y 2) reduce significativamente los costes de movimiento de datos al realizar cálculos utilizando las celdas de memoria. Realizamos una exploración de los parámetros de diseño y demostramos que nuestra versión HPC de MATSA puede mejorar enormemente el rendimiento y la eficiencia energética con respecto a la CPU del servidor, la GPU y el PNM.

- **TraTSA** [56, 55]. Evaluamos las ventajas de aplicar el cálculo transpreciso a AST, donde se reduce el número de bits dedicados a las operaciones de punto flotante. Desarrollamos TraTSA, un *framework* que permite afinar la precisión de las operaciones para cada caso de uso, maximizando la utilización de recursos de las unidades aritméticas. Utilizando este enfoque, es posible reducir el área requerida por el sistema, lo que permite la posibilidad de incluir más unidades aritméticas y así mejorar el rendimiento y reducir el consumo de energía.

Además, esta tesis ha contribuido a varias publicaciones relacionadas con el tema principal de la misma [69, 71, 82, 146, 68, 64].

## A.2. PhiTSA: Usando un Xeon Phi para Optimizar y Caracterizar AST

### A.2.1. Idea y Motivación

En este apartado presentamos PhiTSA, una implementación optimizada del algoritmo SCRIMP adaptada a los procesadores Intel Xeon Phi Knights Landing (KNL). Este trabajo está motivado por la oportunidad que ofrecen estas arquitecturas en términos de paralelismo, vectorización y diferentes tecnologías de memoria. Concretamente, KNL incluye dos tipos de memoria (es decir, DDR4 y HBM). Esto es útil para 1) estudiar el impacto del ancho de banda de la memoria en el rendimiento y 2) aprovechar el ancho de banda agregado. La idea clave es implementar una versión paralela y vectorizada de SCRIMP basada en la privatización. A continuación, utilizamos esta implementación optimizada para caracterizar el comportamiento utilizando las herramientas de perfilado de Intel y encontrar sus cuellos de botella.

### A.2.2. Contribuciones

La primera parte de este trabajo comprende la optimización de SCRIMP para explotar al máximo el potencial de la plataforma KNL. Comenzamos analizando el algoritmo SCRIMP (descrito en la Sección 2.1.1) desde el punto de vista del paralelismo. SCRIMP calcula las distancias entre subsecuencias siguiendo las diagonales de la matriz, que mostramos en la Figura 3.2. Esto se explica por dos razones 1) los elementos de una diagonal se pueden calcular a partir de los elementos anteriores de la misma diagonal con un mínimo de cálculos, y 2) el cálculo de las diagonales en orden aleatorio permite la posibilidad de obtener un resultado parcial exacto del perfil de la matriz. Cuando se procesa una diagonal, un hilo calcula las distancias entre subsecuencias y actualiza el perfil de la matriz  $P$  y el índice del perfil de la matriz  $I$ . De esta manera, podría ocurrir que dos hilos estén calculando simultáneamente dos distancias diferentes en dos diagonales diferentes, pero con dichas distancias pertenecientes a la misma columna. Las posibles actualizaciones concurrentes de las estructuras de datos  $P$  e  $I$  pueden resolverse mediante la exclusión mutua o la privatización de los datos críticos. El rendimiento del primer enfoque puede verse muy afectado por la presión de la sincronización. Por otro lado, aunque el esfuerzo de sincronización es insignificante para la privatización, requiere una cantidad extra de memoria. Tras resolver las actualizaciones concurrentes de nuestra implementación, procedemos a aumentar la intensidad aritmética de nuestras implementaciones utilizando operaciones vectoriales. Por último, proponemos la distribución de los datos en los espacios de memoria DDR4 y HBM asignando datos privados y de acceso más frecuente en HBM y datos compartidos de sólo lectura en DDR4.

### A.2.3. Evaluación

En la evaluación de este trabajo realizamos experimentos relacionados con el aumento de velocidad y el uso del ancho de banda de memoria con una longitud de serie temporal y un tamaño de ventana fijos, utilizando las diferentes implementaciones paralelas descritas en la Sección 3.3. A continuación, medimos el uso del ancho de banda de memoria variando el número de hilos en la implementación que ha dado los mejores resultados de rendimiento. Finalmente, ejecutamos nuestra implementación basada en la estructura privada variando la longitud de la serie temporal, el tamaño de la ventana y las políticas de asignación de memoria, lo que permite una comparación con trabajos anteriores.

Los experimentos demuestran que nuestro enfoque basado en privatización puede mejorar el rendimiento en hasta  $190\times$  con respecto a la ejecución secuen-

cial (que no aprovecha la vectorización ni el espacio HBM) utilizando un Xeon Phi 7210 de 64 núcleos (KNL). Además, nuestras técnicas presentan una mejor escalabilidad que los mecanismos de sincronización basados en bloqueos. Por último, la implementación que utiliza tanto la HBM y DDR4 es capaz de superar en 5 veces a la solución basada únicamente en DDR4, lo que demuestra las ventajas de HBM para los problemas de ancho de banda limitado.

### A.3. NATSA: Un Acelerador PNM para AST

#### A.3.1. Idea y Motivación

En la literatura se han propuesto varias implementaciones de CPU y GPU de *matrix profile*. Sin embargo, estos esfuerzos de aceleración siguen requiriendo la transferencia de los datos de las series temporales desde la memoria principal a los núcleos de la CPU/GPU, lo que conduce al cuello de botella del movimiento de datos, como se analizó en la contribución anterior utilizando PhiTSA. El procesamiento cerca de la memoria (PNM) es un enfoque prometedor para aliviar el movimiento de datos colocando las unidades de procesamiento cerca de la memoria. Como resultado, las soluciones PNM tienen el potencial de mejorar en gran medida el rendimiento del sistema y la eficiencia energética cuando se diseñan cuidadosamente con núcleos de procesamiento cercanos a los datos de bajo coste y baja sobrecarga para aplicaciones ligadas a la memoria.

El objetivo de este trabajo es el de permitir un análisis de series temporales de alto rendimiento y eficiencia energética para una amplia gama de aplicaciones, minimizando los costes generales del movimiento de datos. Esto puede permitir un análisis eficiente de series temporales en sistemas a gran escala, así como en dispositivos embebidos y móviles, donde el consumo de energía es una restricción crítica (por ejemplo, el análisis de los latidos del corazón en un dispositivo médico móvil para predecir un ataque al corazón [118] o la detección temprana de terremotos [41]). Para ello, proponemos *NATSA*, el *primer* acelerador para el análisis de series temporales utilizando un enfoque cercano a la memoria. La idea clave de NATSA (Figura 4.1) es explotar la moderna memoria de alto ancho de banda (HBM) apilada en 3D junto con unidades de procesamiento especializadas en la capa lógica de la HBM, para permitir un cálculo rápido y eficiente energéticamente cerca de la memoria, donde residen los datos de las series temporales. NATSA admite una amplia gama de aplicaciones de series temporales gracias a la generalidad y flexibilidad de *matrix profile*.



### A.3.2. Propuesta

Nuestro acelerador de procesamiento de datos para el análisis de series temporales, NATSA, está diseñado para 1) explotar plenamente el paralelismo de acceso a la memoria y el gran ancho de banda de la memoria que ofrece la HBM, y 2) emplear la cantidad necesaria de recursos computacionales para proporcionar una solución equilibrada. NATSA se localiza junto a la memoria HBM y aprovecha todo el ancho de banda de la HBM disponible. NATSA consta de múltiples unidades de procesamiento (PUs) que computan eficientemente las diagonales del perfil de la matriz de forma paralela. Las PUs están diseñadas para calcular las diagonales utilizando un enfoque vectorial para procesar un lote de elementos de una diagonal al mismo tiempo. Cada PU incluye unidades de punto flotante energéticamente eficientes [60], operadores a nivel de bits y registros (véase la Tabla 4.3 en la Sección 4.3.3). Cada PU se comunica con la HBM a través de un controlador conectado a uno de los 8 canales de memoria de la HBM.

El cálculo de las diagonales de la matriz de distancia puede provocar un desequilibrio de la carga entre las PU, porque esas diagonales tienen longitudes diferentes. Para evitar este desequilibrio, proponemos un esquema de programación de particiones estáticas que sólo depende del tamaño de la serie temporal y de la zona de exclusión. La forma en que abordamos este problema es asignando un conjunto de pares de diagonales a cada PU NATSA tal que la suma de sus elementos sea igual al número de celdas de la diagonal principal de la matriz de distancia menos el número de celdas de la zona de exclusión,  $(n - m + 1) - m/4$ .

### A.3.3. Evaluación

**Rendimiento de NATSA.** NATSA consigue mejoras significativas en el rendimiento, hasta  $14,2\times$  ( $9,9\times$  de media) respecto al sistema de referencia para series grandes, y  $6,3\times$  respecto a HBM-inOrder para todos los tamaños.

**Potencia y consumo de energía.** NATSA reduce el consumo de energía en  $27,2\times$  ( $19,4\times$  de media) respecto a la plataforma de referencia (DDR4-OoO), y en  $10,2\times$  respecto a una arquitectura PNM con núcleos de propósito general (HBM-inOrder). NATSA consume  $1,7\times$ ,  $4,1\times$  y  $11,0\times$  menos energía que una GPU NVIDIA Tesla K40c [145], una GPU NVIDIA GTX 1050 [5] y un Intel Xeon Phi KNL [165], respectivamente. Llegamos a la conclusión de que NATSA es la plataforma evaluada más eficiente energéticamente para *perfil de matriz*.

## A.4. MATSA: Un Acelerador PUM para AST

La siguiente parte de esta tesis versa sobre el estudio de un acelerador para series temporales basado en el uso de las celdas de memoria (PUM). A diferencia de las plataformas comerciales, una arquitectura PUM constituye una solución prometedora para acelerar AST, ya que 1) permite un alto paralelismo, 2) reduce los costes de movimiento de datos, 3) puede proporcionar una mejor escalabilidad que sus competidores debido al paralelismo masivo, 4) es compatible con CMOS, y 5) supera el problema de la volatilidad de los datos. Encontramos que el principal inconveniente de esas tecnologías NVM es su falta de madurez, siendo difíciles de evaluar. Para solucionar este problema, realizamos estudios de sensibilidad en nuestra evaluación para diferentes tecnologías NVM.

### A.4.1. Contribuciones

MATSA es el primer Acelerador basado en MRAM para el análisis de series temporales. La idea clave es explotar las barras transversales de la memoria magnetorresistiva para permitir un cálculo de series temporales rápido y eficiente desde el punto de vista energético en la memoria, superando al mismo tiempo los problemas de resistencia de otras tecnologías de memoria no volátil. MATSA ofrece las siguientes ventajas clave: 1) aprovecha los altos niveles de paralelismo en el sustrato de la memoria explotando las operaciones aritméticas por columnas, y 2) reduce significativamente los costes de movimiento de datos al realizar el cálculo utilizando las celdas de memoria.

### A.4.2. Evaluación

En nuestra evaluación consideramos tres versiones de MATSA adaptadas a diferentes entornos de computación (dispositivos embebidos, ordenadores de sobremesa y clusters HPC) para mostrar la generalidad de nuestro diseño. Caracterizamos el rendimiento de MATSA utilizando la latencia de la celda y el consumo de energía basado en las tendencias del estado de la técnica en las características de los dispositivos MRAM [198, 61]. Nuestra evaluación muestra que MATSA-HPC mejora el rendimiento en  $7,35\times/6,15\times/6,31\times$  y la eficiencia energética en  $11,29\times/4,21\times/2,65\times$  respecto a las plataformas de clase servidor basadas en CPU, GPU y UPMEM, respectivamente.

## A.5. TraTSA: Un Framework Transpreciso Para AST

### A.5.1. Idea y Motivación

Tras proponer nuestros aceleradores cercanos o en la propia memoria, evaluamos las últimas implementaciones de Matrix Profile basadas en el método euclidiano (SCRIMP [202] y SCAMP [203]) y descubrimos que se necesita un gran número de operaciones aritméticas en coma flotante (FP) para analizar incluso series temporales cortas. En este sentido, la computación transprecisa ha surgido recientemente como un enfoque prometedor para 1) mejorar la eficiencia energética, 2) proporcionar un mejor rendimiento, 3) reducir la huella de área y 4) reducir el ancho de banda de la memoria tolerando cierta pérdida de precisión en los resultados calculados. Este paradigma reduce el número de bits para el exponente y la mantisa en las operaciones FP de forma flexible, dependiendo de los requisitos de la aplicación. Es bien sabido que las operaciones FP contribuyen en gran medida ( $\approx 50\%$ ) [173] al consumo de energía en las plataformas de cómputo modernas. Por lo tanto, la transprecisión tiene el potencial de proporcionar un diseño eficiente con la precisión requerida por la aplicación.

Nuestro objetivo en este trabajo es proporcionar un conjunto de herramientas para impulsar el nicho de investigación del análisis de series temporales basado en computación transprecisa, con el fin de lograr una computación de alto rendimiento y eficiencia energética para una amplia gama de aplicaciones. De este modo, se pueden diseñar nuevas plataformas que se beneficien de las operaciones FP de recuento reducido de bits adaptadas a cada aplicación (por ejemplo, utilizando una unidad de puntos flotantes (FPU) transprecisa, como FPNew [8]). Esto abre la oportunidad de detectar eventos importantes en dispositivos móviles y empujados, donde la energía es una preocupación crítica. Estos dispositivos pueden utilizarse, por ejemplo, para prevenir desastres ecológicos o problemas médicos (por ejemplo, para la detección temprana de terremotos [15] o para predecir un ataque al corazón [118]).

### A.5.2. Contribuciones

Para ello, introducimos *TraTSA*, el primer *framework* de transprecisión para el análisis de series temporales. TraTSA proporciona un cálculo de perfiles matriciales usando computación transprecisa rápido y fácil de usar gracias a sus implementaciones en CPU y FPGA. Evaluamos TraTSA con casos de uso de conjuntos

de datos reales de diferentes dominios y tamaños, analizando las compensaciones entre la precisión aritmética y la exactitud de los resultados utilizando una métrica propuesta. Además, presentamos el ahorro de energía de una FPU de transprecisión real.

TraTSA es un *framework* basado en Transprecision para Time Series Analysis desarrollado como una herramienta 1) para realizar la exploración del diseño de los aceleradores y 2) para afinar las implementaciones actuales. De este modo, los arquitectos de ordenadores pueden definir el número exacto de bits de punto flotante para el exponente y la mantisa, lo que potencialmente ahorra área y mejora el rendimiento, al tiempo que reduce el consumo de energía. Construimos el *framework* TraTSA basado en el perfil matricial y utilizando 1) la biblioteca FlexFloat para implementar versiones de transprecisión para CPU de SCRIMP y SCAMP, 2) la biblioteca cpfp-FPGA para implementar una versión transprecisa para FPGA de SCAMP, y 3) Python para crear un *wrapper* de fácil uso.

El beneficio clave de TraTSA es proporcionar un *framework* de desarrollo para transprecisión siendo 1) lo suficientemente portátil para ser ejecutado en diferentes entornos de ejecución de acuerdo con los requisitos de análisis (es decir, la longitud de la serie temporal o si el usuario tiene acceso a una FPGA o no), y 2) lo suficientemente flexible para permitir la posibilidad de explorar una amplia gama de combinaciones de exponente y mantisa para cualquier conjunto de datos. En este sentido, tanto **TranSCRIMP** como **TranSCAMP** están diseñados para ser utilizados con series temporales de tamaños modestos (por debajo de 200K elementos) y ejecutados en CPUs comerciales (ordenadores de sobremesa o servidores de gama alta) debido a los sobrecostos de los tipos de precisión personalizados en dichas plataformas. En cambio, **TranSCAMPfpga** es capaz de computar series de hasta varios millones de elementos en un tiempo manejable gracias al soporte hardware de transprecisión, a costa de requerir una FPGA. Tanto el *backend* de la CPU como el de la FPGA pueden trabajar simultáneamente y unir la potencia de cálculo.

### A.5.3. Evaluación

En nuestra evaluación comparamos los algoritmos de SCRIMP y SCAMP. Los resultados concluyen que SCAMP es más robusto y presenta una mejor estabilidad numérica que SCRIMP para todos los conjuntos de datos. Este hecho se aprecia para el conjunto de datos *Penguin Behavior*, donde una ligera disminución de la longitud de la mantisa hace que SCRIMP falle en la detección de eventos, mientras que SCAMP proporciona más margen en esta reducción. Encontramos

un escenario atípico para la serie temporal *Power\_short* en la Figura 6.9, donde la precisión parece disminuir cuando el número de bits de la mantisa aumenta de 20 a 23. La razón es que, para esta serie temporal, hay varios motivos con exactamente el mismo valor de perfil. Por ello, nuestro algoritmo de ordenación induce algunas diferencias de orden al comparar la versión de transprecisión con respecto a la de referencia. Sin embargo, desde el punto de vista práctico, la precisión de la mantisa 23 es tan buena como la de la mantisa 20, ya que todos los motivos están presentes en ambas. Concluimos que SCAMP es un mejor candidato para el cálculo de transprecisión, ya que proporciona una precisión similar con menores requisitos de recuento de bits para el exponente y la mantisa.

## A.6. Conclusiones y Trabajo Futuro

Los algoritmos de análisis de series temporales son una carga de trabajo crítica para muchas aplicaciones importantes (por ejemplo, para predecir un ataque al corazón o para analizar el consumo eléctrico de los hogares). En esta tesis, estudiamos y caracterizamos los algoritmos de análisis de series temporales más avanzados y descubrimos que el movimiento de datos en las arquitecturas convencionales les supone un cuello de botella. Nuestro objetivo final es acelerar y hacer que el cálculo de análisis de series temporales sea lo más rápido y eficiente energéticamente posible. Para ello, proponemos soluciones de hardware y software que acerquen el cálculo a la memoria, basándonos en las últimas tendencias en arquitectura de ordenadores. Esta tesis realiza cuatro contribuciones principales, que se describen a continuación.

**① Optimización y caracterización.** El Capítulo 3 presenta una novedosa implementación de un algoritmo de análisis de series temporales de última generación ajustado para una arquitectura Intel Xeon Phi KNL, provista de HBM. En esta implementación explotamos el multihilo, la vectorización y el uso de la HBM agregada más el ancho de banda de la memoria DDR4. En primer lugar, proponemos la paralelización del procesamiento de las diagonales en el algoritmo, distribuyendo dinámicamente su cómputo entre los núcleos de una máquina multinúcleo. En segundo lugar, utilizamos técnicas de privatización y reducción para evitar la sincronización innecesaria de hilos y mejorar la escalabilidad. Por último, proponemos la distribución de los datos entre DDR4 y HBM, asignando los datos privados y de acceso más frecuente en HBM y los datos compartidos de sólo lectura en DDR4. Nuestros experimentos muestran que nuestro enfoque puede mejorar el rendimiento hasta 190 veces con respecto a la ejecución secuencial utilizando el KNL Intel Xeon Phi. Además, nuestras técnicas presentan una

mejor escalabilidad que los mecanismos tradicionales de sincronización basados en bloqueos. Este capítulo está asociado a la siguiente publicación:

Fernandez, I., Villegas, A., Gutierrez, E., & Plata, O. (2019). Accelerating time series motif discovery in the Intel Xeon Phi KNL processor. *The Journal of Supercomputing*, 75(11), 7053-7075.

**② Acelerador de procesamiento en memoria cercana.** El Capítulo 4 presenta NATSA, el primer acelerador de procesamiento de datos cercanos (PNM) para el análisis de series temporales. NATSA 1) aprovecha el ancho de banda de la memoria de gran ancho de banda (HBM) para analizar datos de series temporales a escala para una amplia gama de aplicaciones, 2) mejora la eficiencia energética y el tiempo de ejecución mediante el uso de unidades aritméticas especializadas de bajo consumo cerca de la memoria HBM, y 3) proporciona un novedoso esquema de programación de la carga de trabajo para evitar el desequilibrio de la carga y preservar la propiedad *anytime*. NATSA supera a las plataformas de hardware que evaluamos en términos de rendimiento, consumo de energía y requisitos de área. Este capítulo está asociado a la siguiente publicación:

Fernandez, I., Quisilant, R., Gutiérrez, E., Plata, O., Giannoula, C., Alser, M., Gómez-Luna, J., & Mutlu, O. (2020, October). NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In 2020 IEEE 38th International Conference on Computer Design (ICCD) (pp. 120-129). IEEE.

**③ Acelerador de Procesamiento en Memoria.** El Capítulo 5 presenta MATSA, el primer acelerador basado en MRAM para el análisis de series temporales. La idea clave es explotar las barras transversales magnetorresistivas para permitir un cálculo de series temporales en memoria rápido y eficiente desde el punto de vista energético. MATSA ofrece las siguientes ventajas clave: 1) aumenta significativamente el paralelismo explotando las operaciones a nivel de columna y 2) reduce los gastos generales del movimiento de datos realizando el cálculo en las celdas de memoria. Evaluamos tres versiones de MATSA para ajustarnos a los requisitos de diferentes entornos y realizamos una exploración del espacio de diseño. MATSA mejora significativamente el rendimiento y la energía con respecto a las plataformas commodity y PNM. Este capítulo está asociado a la siguiente publicación en revisión:

Fernandez, I., Manglik, A., Giannoula, C., Quisilant, R., Ghiasi, N. M., Gómez-Luna, J., ... & Mutlu, O. (2022). Accelerating Time Series Analysis via Processing using Non-Volatile Memories. arXiv preprint arXiv:2211.04369.

④ **Framework de transprecisión.** El Capítulo 6 estudia cómo el análisis de series temporales se beneficia de un enfoque de transprecisión, e introduce TraTSA, un *framework* que permite definir la precisión exacta necesaria según los requisitos de la aplicación específica. Las implementaciones propuestas **TransSCRIMP**, **TransSCAMP** y **TransSCAMPfpga** ayudarán a la comunidad a diseñar soluciones de análisis de series temporales energéticamente eficientes basadas en procesadores RISC-V con soporte transpreciso, FPGAs o ASICs minimizando los requisitos de área y potencia. Nuestra solución basada en FPGA es  $22,75\times$  más rápida que un servidor Intel Xeon de 72 núcleos gracias al soporte de transprecisión por hardware y al uso de técnicas de optimización. Además, estudiamos cómo los algoritmos de Matrix Profile pueden beneficiarse de una FPU de transprecisión ya presentada. En este sentido, nuestro análisis revela que, para una variedad de aplicaciones, el consumo de energía de los algoritmos de Matrix Profile se reduce hasta  $3,3\times$  en comparación con la doble precisión mientras se obtienen resultados precisos. Este capítulo está asociado a las siguientes publicaciones.

Fernandez, I., Quislan, R., Gutierrez, E., & Plata, O. (2020, September). Energy-Efficient Time Series Analysis Using Transprecision Computing. In 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (pp. 83-90). IEEE.

Fernandez, I., Quislan, R., Gonzalez-Navarro, S., Gutierrez, E., & Plata, O. (2022). TraTSA: A Transprecision Framework for Efficient Time Series Analysis. *Journal of Computational Science*, 63, 101784.

Para concluir esta tesis, nos gustaría proponer las siguientes líneas de investigación futuras inspiradas en nuestro trabajo.

Como se menciona en el Capítulo 3, el principal cuello de botella para acelerar el análisis de series temporales en sistemas commodity es el movimiento de datos. Esta limitación se produce entre la memoria principal y las unidades de procesamiento, y puede mitigarse mediante enfoques de Procesamiento-Cercano-Memoria o Procesamiento-Usando-Memoria. Sin embargo, estos escenarios asumen que los datos ya están localizados y ordenados adecuadamente en la memoria principal. Las series temporales suelen incluir grandes cantidades de datos de entrada que pueden alojarse en dispositivos de almacenamiento. La comunicación de esos dispositivos de almacenamiento con la memoria principal suele ser más lenta que la existente entre la memoria principal y las unidades de procesamiento. Basándonos en esto, ***nuestra primera línea de investigación propuesta es explorar enfoques de Procesamiento en Almacenamiento para optimizar y acelerar aún más el cálculo del análisis de series temporales.***

Como se menciona en el Capítulo 6, otra forma de aliviar el impacto del movimiento de datos es reducir la cantidad de datos a mover. Una forma de hacerlo es el uso de la computación transprecisa, reduciendo el número de bits que toman los operandos en la memoria y también al realizar las operaciones reales. Aunque evaluamos cómo los algoritmos SCRIMP y SCAMP se benefician de este enfoque, hay muchos otros algoritmos (por ejemplo, DTW) que podrían ser buenos candidatos. En base a esto, *nuestra segunda línea de investigación propuesta es explorar enfoques de computación transprecisa en otros algoritmos de análisis de series temporales, reduciendo la cantidad de datos a mover durante el cálculo.*



# Bibliografía

- [1] Intel Advisor website. [software.intel.com/en-us/advisor](https://software.intel.com/en-us/advisor). Accessed 16 Mar. 2019. (Cited on page 41)
- [2] Intel Processor Counter Monitor. <https://github.com/opcm/pcm>. Accessed 23 September 2020. (Cited on page 54)
- [3] Intel VTune website. [software.intel.com/en-us/vtune](https://software.intel.com/en-us/vtune). Accessed 25 November 2019. (Cited on page 38)
- [4] Micron Power Calculator. [www.micron.com/support/tools-and-utilities/power-calc](https://www.micron.com/support/tools-and-utilities/power-calc). Accessed 23 September 2020. (Cited on page 51)
- [5] NVIDIA GTX 1050 Specs. <https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1050/>. Accessed 23 September 2020. (Cited on pages 55 and 123)
- [6] NVIDIA Visual Profiler. <https://developer.nvidia.com/nvidia-visual-profiler>. Accessed 23 September 2020. (Cited on page 54)
- [7] Superserver 5038k-i specs. [www.supermicro.com/products/system/tower/5038/SYS-5038K-I.cfm](http://www.supermicro.com/products/system/tower/5038/SYS-5038K-I.cfm). Accessed 22 November 2019. (Cited on page 35)
- [8] FPnew Source Code. <https://github.com/pulp-platform/fpnew>, 2021. (Cited on pages 25, 77 and 125)
- [9] Penguin Data. <https://www.cs.ucr.edu/~eamonn/MatrixProfile.html>, 2021. (Cited on page 89)

- [10] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. In *HPCA*, 2017. (Cited on pages 7, 21 and 118)
- [11] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoun Choi. A Scalable Processing-In-Memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015. (Cited on pages 19 and 21)
- [12] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoun Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-In-Memory Architecture. In *ISCA*, 2015. (Cited on pages 19 and 21)
- [13] S. Alaee, K. Kamgar, and E. Keogh. Matrix Profile XXII: exact discovery of time series motifs under DTW. In *IEEE 16th International Conference on Data Mining (ICDM)*, 2020. (Cited on pages 14 and 89)
- [14] Oleksandra Aleksandrova and Yevgen Bashkov. Face recognition systems based on Neural Compute Stick 2, CPU, GPU comparison. In *ATIT*, pages 104–107. IEEE, 2020. (Cited on page 67)
- [15] Richard M Allen, Qingkai Kong, and Robert Martin-Short. The MyShake Platform: a global vision for earthquake early warning. *Pure and Applied Geophysics*, 177:1699–1712, 2020. (Cited on pages 77 and 125)
- [16] Miquel L Alomar, Vincent Canals, Nicolas Perez-Mora, Víctor Martínez-Moll, and Josep L Rosselló. FPGA-based stochastic echo state networks for time-series forecasting. *Computational intelligence and neuroscience*, 2016, 2016. (Cited on pages 7 and 117)
- [17] Mohammed Alser, Zülal Bingöl, Damla Senol Cali, Jeremie Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. Accelerating Genome Analysis: A Primer on an Ongoing Journey. *IEEE Micro*, 2020. (Cited on page 21)
- [18] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. Shouji: A Fast and Efficient Pre-Alignment Filter for Sequence Alignment. *Bioinformatics*, 2019. (Cited on page 21)
- [19] Mohammed Alser, Hasan Hassan, Hongyi Xin, Oğuz Ergin, Onur Mutlu, and Can Alkan. GateKeeper: A New Hardware Arch. for Accelerating Pre-alignment in DNA Short Read Mapping. *Bioinformatics*, 2017. (Cited on page 21)

- [20] Mohammed Alser, Taha Shahroodi, Juan Gomez-Luna, Can Alkan, and Onur Mutlu. SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs. *arXiv*, 2019. (Cited on page 21)
- [21] Mohamed M Sabry Aly, Mingyu Gao, Gage Hills, Chi-Shuen Lee, Greg Pitner, Max M Shulaker, Tony F Wu, Mehdi Asheghi, Jeff Bokor, Franz Franchetti, et al. Energy-efficient abundant-data computing: The N3XT 1,000 x. *Computer*, 48(12):24–33, 2015. (Cited on page 23)
- [22] Antaios. Spin-Orbit Torque MRAM. Technical Report. Antaios, 51 Avenue Jean Kuntzmann, 38830 Montbonnot – France. 3 pages. [https://www.antaivos.fr/IMG/pdf/web\\_site\\_sot\\_whitepaper.pdf](https://www.antaivos.fr/IMG/pdf/web_site_sot_whitepaper.pdf). 2020. (Cited on page 23)
- [23] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and Practical Near-DRAM Acceleration Arch. for Large Mem. Sys. In *MICRO*, 2016. (Cited on page 21)
- [24] Samet Ayhan and Hanan Samet. Time series clustering of weather observations in predicting climb phase of aircraft trajectories. In *IWCTS*, pages 25–30, 2016. (Cited on page 3)
- [25] Arvind Balasubramanian, Jun Wang, and Balakrishnan Prabhakaran. Discovering Multidimensional Motifs in Physiological Signals for Personalized Healthcare. *JSTSP*, 2016. (Cited on pages 3 and 12)
- [26] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. Seattle, WA, USA:, 1994. (Cited on page 17)
- [27] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 Simulator. *Comp. Arch. News*, 2011. (Cited on page 51)
- [28] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. *ASPLOS*, 2018. (Cited on pages 19 and 21)
- [29] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar,

- Krishna T Malladi, Hongzhong Zheng, et al. CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators. In *ISCA*, 2019. (Cited on page 21)
- [30] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Nastaran Hajinazar, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. LazyPIM: Efficient Support for Cache Coherence in Processing-In-Memory Architectures. *arXiv*, 2017. (Cited on page 21)
- [31] George E. P. Box, Gwilym M. Jenkins, Gregory C. Reinsel, and Greta M. Ljung. *Time Series Analysis: Forecasting and Control*. Wiley-Interscience, 2015. (Cited on page 3)
- [32] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions on Mathematical Software*, 34, 2008. (Cited on page 24)
- [33] Damla Senol Cali, Gurpreet S Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, et al. GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *MICRO*, 2020. (Cited on pages 19 and 21)
- [34] Nuno Castro and Paulo Azevedo. Multiresolution motif discovery in time series. In *Proceedings of the 2010 SIAM international conference on data mining*, pages 665–676. SIAM, 2010. (Cited on page 12)
- [35] Kai-Wei Chang, Biplab Deka, Wen-Mei W Hwu, and Dan Roth. Efficient pattern-based time series classification on GPU. In *2012 IEEE 12th International Conference on Data Mining*, pages 131–140. IEEE, 2012. (Cited on pages 6 and 117)
- [36] Kevin K Chang, Prashant J Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K Qureshi, and Onur Mutlu. Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM. In *HPCA*, 2016. (Cited on page 21)
- [37] Guangyuan Chen, Guoliang Lu, Zhaohong Xie, and Wei Shang. Anomaly detection in eeg signals: a case study on similarity measure. *Computational intelligence and neuroscience*, 2020, 2020. (Cited on pages 4 and 116)

- [38] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A Novel Processing-In-Memory Arch. for Neural Network Computation in ReRAM-Based Main Memory. In *ISCA*, 2016. (Cited on page 21)
- [39] Yu-Der Chih, Yi-Chun Shih, Chia-Fu Lee, Yen-An Chang, Po-Hao Lee, Hon-Jarn Lin, Yu-Lin Chen, Chieh-Pu Lo, Meng-Chun Shih, Kuei-Hung Shen, et al. 13.3 a 22nm 32Mb embedded STT-MRAM with 10ns read speed, 1M cycle write endurance, 10 years retention at 150 c and high immunity to magnetic field interference. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 222–224. IEEE, 2020. (Cited on page 23)
- [40] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. Probabilistic Discovery of Time Series Motifs. In *SIGKDD*, 2003. (Cited on pages 12 and 87)
- [41] Annemarie Christophersen, Natalia I. Deligne, Anca M. Hanea, Lauriane Chardot, Nicolas Fournier, and Willy P. Aspinall. Bayesian network modeling and expert elicitation for probabilistic eruption forecasting: Pilot study for Whakaari/White Island, New Zealand. *Frontiers in Earth Science*, 6:211, 2018. (Cited on pages III, 4, 45, 116 and 122)
- [42] Igor M Coelho, Vitor N Coelho, Eduardo J da S Luz, Luiz S Ochi, Frederico G Guimarães, and Eyder Rios. A GPU deep learning metaheuristic based model for time series forecasting. *Applied Energy*, 201:412–418, 2017. (Cited on pages 6 and 117)
- [43] Xilinx Corporation. Xilinx board utility (xbutil). <https://xilinx.github.io/XRT/master/html/xbutil.html>, 2022. (Cited on page 89)
- [44] Hoang Anh Dau and Eamonn Keogh. Matrix profile V: A generic technique to incorporate domain knowledge into motif discovery. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 125–134, 2017. (Cited on page 104)
- [45] Tim Daulby, Anand Savanth, Alex S Weddell, and Geoff V Merrett. Comparing NVM technologies through the lens of Intermittent computation. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, pages 77–78, 2020. (Cited on pages XVII and 23)
- [46] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24. IEEE Computer Society, 2019. (Cited on page 68)

- [47] Roberto DiCecco, Lin Sun, and Paul Chow. FPGA-based training of convolutional neural networks with a reduced precision floating-point library. In *International Conference on Field Programmable Technology (ICFPT)*, 2017. (Cited on page 25)
- [48] Antonio J Dios, Angeles Navarro, Rafael Asenjo, Francisco Corbera, and Emilio L Zapata. A case study of the task-based parallel wavefront pattern. In *Applications, Tools and Techniques on the Road to Exascale Computing*, pages 65–72. IOS Press, 2012. (Cited on page 65)
- [49] Mario Paulo Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Us-tiugov, Javier Picorel Obando, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The Mondrian Data Engine. In *ISCA*, 2017. (Cited on page 21)
- [50] Tim Dunn, Harisankar Sadasivan, Jack Wadden, Kush Goliya, Kuan-Yu Chen, David Blaauw, Reetuparna Das, and Satish Narayanasamy. Squigglefilter: An accelerator for portable virus detection. In *MICRO*, pages 535–549, 2021. (Cited on pages 1, 4, 114 and 116)
- [51] André E X Brown, Eviatar Yemini, Laura J Grundy, Tadas Jucikas, and William Schafer. A dictionary of behavioral motifs reveals clusters of genes affecting *Caenorhabditis elegans* locomotion. *Proceedings of the National Academy of Sciences of the United States of America*, 110, 2012. (Cited on page 3)
- [52] Tetsuo Endoh, Hiroaki Honjo, Koichi Nishioka, and Shoji Ikeda. Recent progresses in STT-MRAM and SOT-MRAM for next generation MRAM. In *2020 IEEE Symposium on VLSI Technology*, pages 1–2. IEEE, 2020. (Cited on page 72)
- [53] Taban Eslami and Fahad Saeed. Fast-GPU-PCC: A GPU-Based Technique to Compute Pairwise Pearson’s Correlation Coefficients for Time Series Data-fMRI Study. *High-throughput*, 7(2):11, 2018. (Cited on pages 6 and 117)
- [54] P. Feautrier. Array expansion. In *Int’l. Conf. on Supercomputing (ICS’88)*, pages 429–441, 1988. (Cited on page 31)
- [55] Ivan Fernandez, Ricardo Quisilant, Sonia Gonzalez-Navarro, Eladio Gutierrez, and Oscar Plata. TraTSA: A transprecision framework for efficient time series analysis. *Journal of Computational Science*, page 101784, 2022. (Cited on pages 9 and 120)

- [56] Ivan Fernandez, Ricardo Quislan, Eladio Gutierrez, and Oscar Plata. Energy-efficient time series analysis using transprecision computing. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 83–90. IEEE, 2020. (Cited on pages 9 and 120)
- [57] Ivan Fernandez, Ricardo Quislan, Eladio Gutiérrez, Oscar Plata, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. NAT-SA: A near-data processing accelerator for time series analysis. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 120–129. IEEE, 2020. (Cited on pages 9 and 119)
- [58] Ivan Fernandez, Alejandro Villegas, Eladio Gutierrez, and Oscar Plata. Accelerating Time Series Motif Discovery in the Intel Xeon Phi KNL Processor. *The Journal of Supercomputing*, 2019. (Cited on pages XI, XI, 5, 6, 9, 80 and 119)
- [59] Pedro G Ferreira, Paulo J Azevedo, Cândida G Silva, and Rui MM Brito. Mining Approximate Motifs in Time Series. In *International Conference on Discovery Science*, 2006. (Cited on page 12)
- [60] Sameh Galal and Mark Horowitz. Energy-Efficient Floating-Point Unit Design. *IEEE Transactions on Computers*, 2010. (Cited on pages 46 and 123)
- [61] William J Gallagher, Eric Chien, Tien-Wei Chiang, Jian-Cheng Huang, Meng-Chun Shih, CY Wang, Christine Bair, George Lee, Yi-Chun Shih, Chia-Fu Lee, et al. Recent progress and next directions for embedded MRAM technology. In *2019 Symposium on VLSI Circuits*, pages T190–T191. IEEE, 2019. (Cited on page 124)
- [62] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *PACT*, 2015. (Cited on page 21)
- [63] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *ASPLOS*, 2017. (Cited on page 21)
- [64] Nika Mansouri Ghiasi, Nandita Vijaykumar, Geraldo F Oliveira, Lois Orosa, Ivan Fernandez, Mohammad Sadrosadati, Konstantinos Kanellopoulos, Nastaran Hajinazar, Juan Gómez Luna, and Onur Mutlu. ALP: Alleviating CPU-Memory Data Movement Overheads in Memory-Centric Systems. *IEEE Transactions on Emerging Topics in Computing*, 2022. (Cited on pages 10 and 120)

- [65] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. Processing-In-Memory: A Workload-Driven Perspective. *IBM Journal of Research and Development*, 2019. (Cited on page 21)
- [66] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungni-run, and Onur Mutlu. Enabling the Adoption of Processing-In-Memory: Challenges, Mechanisms, Future Research Directions. *arXiv*, 2018. (Cited on page 21)
- [67] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. In *SIGMETRICS*, 2019. (Cited on page 20)
- [68] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. SparseP: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–49, 2022. (Cited on pages 10 and 120)
- [69] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Synchron: Efficient synchronization support for near-data-processing architectures. In *HPCA*, pages 263–276. IEEE, 2021. (Cited on pages 10 and 120)
- [70] A. Glew. MLP yes! ILP no! In *ASPLOS*, 1998. (Cited on page 57)
- [71] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *arXiv preprint arXiv:2105.03814*, 2021. (Cited on pages 10, 68 and 120)
- [72] Alessandro Grossi, Cristian Zambelli, Piero Olivo, Paolo Pellati, Michele Ramponi, Christian Wenger, Jeremy Alvarez-Herault, and Ken Mackay. An automated test equipment for characterization of emerging MRAM and RRAM arrays. *IEEE Transactions on Emerging Topics in Computing*, 6(2):269–277, 2016. (Cited on page 23)
- [73] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–33, 2009. (Cited on page 23)



- [74] Sankalp Gulati, Joan Serra, Vignesh Ishwar, and Xavier Serra. Mining Melodic Patterns in Large Audio Collections of Indian Art Music. In *SITIS*, 2014. (Cited on page 12)
- [75] Saransh Gupta, Mohsen Imani, and Tajana Rosing. Exploring processing in-memory for different technologies. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 201–206, 2019. (Cited on pages 7 and 118)
- [76] E. Gutiérrez, O. Plata, and E. L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In *Int'l. Conf. on Supercomputing (ICS'00)*, pages 78–87, 2000. (Cited on page 31)
- [77] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. NDC: Analyzing the Impact of 3D-stacked Memory+Logic Devices on MapReduce Workloads. In *ISPASS*, 2014. (Cited on page 21)
- [78] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. Demystifying the Characteristics of 3D-stacked Memories: A case Study for Hybrid Memory Cube. In *IISWC*, 2017. (Cited on page 20)
- [79] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–345, 2021. (Cited on pages 8, 22 and 118)
- [80] Milad Hashemi, Eiman Ebrahimi, Onur Mutlu, Yale N Patt, et al. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *ISCA*, 2016. (Cited on page 21)
- [81] Milad Hashemi, Onur Mutlu, and Yale N Patt. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *MICRO*, 2016. (Cited on page 21)
- [82] Jose M Herruzo, Ivan Fernandez, Sonia González-Navarro, and Oscar Plata. Enabling fast and energy-efficient FM-index exact matching using processing-near-memory. *The Journal of Supercomputing*, 77(9):10226–10251, 2021. (Cited on pages 10 and 120)

- [83] E. Philip Howrey. The role of time series analysis in econometric model evaluation. *Evaluation of Econometric Models*, pages 275–307, 1980. (Cited on page 3)
- [84] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. TOM: Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*, 2016. (Cited on pages 19 and 21)
- [85] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016. (Cited on pages 19 and 21)
- [86] Yan Hu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk, and Eamonn Keogh. Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins. In *ICDM*, 2016. (Cited on pages 2, 13, 37, 38, 41, 52 and 114)
- [87] Yiming Huai, Yuchen Zhou, Ioan Tudosa, Roger Malmhall, Rajiv Ranjan, and Jing Zhang. Progress and outlook for STT-MRAM. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 235–235. IEEE, 2011. (Cited on page 23)
- [88] Lal Hussain, Wajid Aziz, Jalal S. Alowibdi, Nazneen Habib, Muhammad Rafique, Sharjil Saeed, and Syed Zaki Hassan Kazmi. Symbolic time series analysis of electroencephalographic (EEG) epileptic seizure and brain dynamics with eye-open and eye-closed subjects during resting states. *Journal of Physiological Anthropology*, 36(1), 2017. (Cited on page 3)
- [89] Shah Muhammed Abid Hussain and ABM Harun-ur Rashid. User independent hand gesture recognition by accelerated DTW. In *ICIEV*, pages 1033–1037. IEEE, 2012. (Cited on page 3)
- [90] Stephanie L Hyland, Martin Faltys, Matthias Hüser, Xinrui Lyu, Thomas Gumbsch, Cristóbal Esteban, Christian Bock, Max Horn, Michael Moor, Bastian Rieck, et al. Early prediction of circulatory failure in the intensive care unit using machine learning. *Nature medicine*, 26(3):364–373, 2020. (Cited on pages 4 and 115)
- [91] Pulkit Jain, Umut Arslan, Meenakshi Sekhar, Blake C Lin, Liqiong Wei, Tanaya Sahu, Juan Alzate-Vinasco, Ajay Vangapaty, Mesut Meterelliyozy,

- Nathan Strutt, et al. 13.2 A 3.6 Mb 10.1 Mb/mm<sup>2</sup> embedded non-volatile ReRAM macro in 22nm FinFET technology with adaptive forming/set/reset schemes yielding down to 0.5 V with sensing time of 5ns at 0.7 V. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 212–214. IEEE, 2019. (Cited on page 23)
- [92] JEDEC JESD79-4C. DDR4 SDRAM standard. [www.jedec.org/standards-documents/docs/jesd79-4a](http://www.jedec.org/standards-documents/docs/jesd79-4a). Accessed 23 September 2020. (Cited on page 51)
- [93] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016. (Cited on pages XII, 28 and 88)
- [94] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. HBM DRAM Technology and Architecture. In *IMW*, 2017. (Cited on pages 20 and 84)
- [95] Matthias Jung, Deepak M Mathew, Christian Weis, and Norbert Wehn. Approximate computing with partially unreliable dynamic random access memory—Approximate DRAM. In *2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–4. IEEE, 2016. (Cited on page 23)
- [96] Seungchul Jung, Hyungwoo Lee, Sungmeen Myung, Hyunsoo Kim, Seung Keun Yoon, Soon-Wan Kwon, Yongmin Ju, Minje Kim, Wooseok Yi, Shinhee Han, et al. A crossbar array of magnetoresistive memory devices for in-memory computing. *Nature*, 601(7892):211–216, 2022. (Cited on page 23)
- [97] Amin Kalantar, Zachary Zimmerman, and Philip Brisk. FA-LAMP: FPGA-accelerated learned approximate matrix profile for time series similarity prediction. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–49. IEEE, 2021. (Cited on pages 7 and 117)
- [98] Seongyoung Kang, Jinyeong Moon, and Sang-Woo Jun. FPGA-accelerated time series mining on low-power IOT devices. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 33–36. IEEE, 2020. (Cited on pages 7 and 117)
- [99] Eamonn Keogh and Shruti Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Min. Knowl. Discov.*, 7(4):349–371, 2003. (Cited on page 3)

- [100] Eamonn Keogh, Jessica Lin, Sang-Hee Lee, and Helga Van Herle. Finding the most unusual time series subsequence: algorithms and applications. *Knowledge and Information Systems*, 11(1):1–27, 2007. (Cited on pages 1, 87 and 113)
- [101] Dounia Khaldi and Barbara Chapman. Towards automatic HBM allocation using LLVM: a case study with knights landing. In *Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 12–20, 2016. (Cited on page 32)
- [102] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. *Rapl in Action: Experiences in Using RAPL for Power Measurements*, 2018. (Cited on pages 68 and 89)
- [103] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A Programmable Digital Neuromorphic Architecture with High-density 3D Memory. In *ISCA*, 2016. (Cited on page 21)
- [104] Jeremie S Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. The DRAM latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices. In *HPCA*, 2018. (Cited on page 21)
- [105] Jeremie S Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu. D-RaNGE: Using Com. DRAM Devices to Generate True Random Numb. with Low Lat. and High Throughput. In *HPCA*, 2019. (Cited on page 21)
- [106] Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. GRIM-Filter: Fast seed Location Filter. in *DNA Read Mapping Using PIM Technologies. BMC Genomics*, 2018. (Cited on pages 19 and 21)
- [107] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *CAL*, 2015. (Cited on pages 20, 50 and 68)
- [108] Anna Klos, Machiel S Bos, and Janusz Bogusz. Detecting time-varying seasonal signal in GPS position time series with different noise levels. *GPS solutions*, 22(1):1–11, 2018. (Cited on page 3)
- [109] S. A. P. Kumar and P. K. Bora. Time series analysis and signal processing. In *Conf. on Computational Intelligence and Signal Processing*, pages 24–24, 2012. (Cited on page 3)

- [110] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. Magic—memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014. (Cited on page 21)
- [111] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *IMC*, pages 201–206. ACM, 2004. (Cited on page 3)
- [112] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014. (Cited on pages III, 1 and 113)
- [113] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009. (Cited on page 23)
- [114] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong. 25.2 A 1.2V 8GB 8-channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods using 29nm Process and TSV. In *ISSCC*, 2014. (Cited on pages 19 and 51)
- [115] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Manabi Khan, and Onur Mutlu. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. *TACO*, 2016. (Cited on page 19)
- [116] V T Lee, A Mazumdar, C C del Mundo, A Alaghi, L Ceze, and M Oskin. Application Codesign of NDP for Similarity Search. In *IPDPS*, 2018. (Cited on page 21)
- [117] Haitong Li, Mudit Bhargav, Paul N Whatmough, and H-S Philip Wong. On-chip memory technology design space explorations for mobile deep neural network accelerators. In *2019 56th ACM/IEEE design automation conference (DAC)*, pages 1–6. IEEE, 2019. (Cited on page 23)
- [118] Ka Hou Christien Li, Francesca Anne White, Timothy Tipoe, Tong Liu, Martin CS Wong, Aaron Jesuthasan, Adrian Baranchuk, Gary Tse, and Bryan P Yan. The Current State of Mobile Phone Apps for Monitoring Heart Rate, Heart Rate Variability, and Atrial Fibrillation: Narrative Review. *JMIR Mhealth Uhealth*, 2019. (Cited on pages III, 45, 77, 122 and 125)

- [119] Li Li, Xiaonan Su, Yi Zhang, Yuetong Lin, and Zhiheng Li. Trend modeling for traffic time series analysis: An integrated study. *IEEE Transactions on Intelligent Transportation Systems*, 16(6):3430–3439, 2015. (Cited on page 3)
- [120] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*, 2009. (Cited on pages 51 and 68)
- [121] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-volatile Memories. In *DAC*, 2016. (Cited on page 21)
- [122] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28:152–205, 2002. (Cited on page 24)
- [123] Yuan Li, Jessica Lin, and Tim Oates. Visualizing Variable-Length Time Series Motifs. In *SDM*, 2012. (Cited on page 12)
- [124] Jia Liu, Yong Xue, Kaijun Ren, Junqiang Song, Christopher Windmill, and Patrick Merritt. High-performance time-series quantitative retrieval from satellite images on a GPU cluster. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 12(8):2810–2821, 2019. (Cited on pages 6 and 117)
- [125] Gabriel H Loh, Nuwan Jayasena, M Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. A Processing in Memory Taxonomy and a Case for Studying Fixed-Function PIM. In *WoNDP*, 2013. (Cited on page 21)
- [126] Nguyen Cong Long, Phayung Meesad, and Herwig Unger. A highly accurate firefly based algorithm for heart disease prediction. *Expert Systems with Applications*, 42(21):8221–8231, 2015. (Cited on pages 1 and 114)
- [127] Jeffry Louis, Barak Hoffer, and Shahar Kvatinisky. Performing memristor-aided logic (MAGIC) using STT-MRAM. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 787–790. IEEE, 2019. (Cited on page 21)

- [128] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005. (Cited on page 50)
- [129] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. A 0.80 pJ/flop, 1.24 Tflop/sW 8-to-64 bit transprecision floating-point unit for a 64 bit RISC-V processor in 22nm FD-SOI. In *IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SOC)*, 2019. (Cited on pages 25 and 89)
- [130] A Cristiano I Malossi, Michael Schaffner, Anca Molnos, Luca Gammaitoni, Giuseppe Tagliavini, Andrew Emerson, Andrés Tomás, Dimitrios S Nikolopoulos, Eric Flamand, and Norbert Wehn. The transprecision computing paradigm: concept, design, and applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018. (Cited on page 77)
- [131] Amy McGovern, Derek H. Rosendahl, Rodger A. Brown, and Kelvin K. Droegemeier. Identifying predictive multi-dimensional time series motifs: an application to severe weather prediction. *Data Mining and Knowledge Discovery*, 22(1):232–258, 2011. (Cited on page 3)
- [132] Sparsh Mittal. A survey of ReRAM-based architectures for processing-in-memory and neural networks. *Machine learning and knowledge extraction*, 1(1):75–114, 2018. (Cited on page 21)
- [133] George B Moody and Roger G Mark. The impact of the MIT-BIH arrhythmia database. *IEEE Engineering in Medicine and Biology Magazine*, 20:45–50, 2001. (Cited on page 15)
- [134] Abdullah Mueen and Nikan Chavoshi. Enumeration of Time Series Motifs of All Lengths. *Knowledge and Information Systems*, 2015. (Cited on page 12)
- [135] Abdullah Mueen, Eamonn J. Keogh, Qiang Zhu, Sydney Cash, and M. Brandon Westover. Exact Discovery of Time Series Motifs. In *SDM*, 2009. (Cited on page 12)
- [136] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing Data Where it Makes Sense: Enabling In-Memory Computation. *Microprocessors and Microsystems*, 2019. (Cited on page 21)
- [137] Onur Mutlu, Hyesoon Kim, and Yale N Patt. Techniques for Efficient Processing in Runahead Execution Engines. In *ISCA*, 2005. (Cited on page 57)

- [138] Onur Mutlu, Hyesoon Kim, and Yale N Patt. Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance. *IEEE Micro*, 2006. (Cited on pages 19 and 57)
- [139] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008. (Cited on page 57)
- [140] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *HPCA*, 2003. (Cited on page 57)
- [141] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead Execution: An Effective Alternative to Large Instruction Windows. *IEEE Micro*, 2003. (Cited on page 57)
- [142] Pawan Nunthanid, Vit Niennattrakul, and Chotirat Ann Ratanamahatana. Discovery of Variable Length Time Series Motif. In *ECTI-CON*, 2011. (Cited on page 12)
- [143] Pawan Nunthanid, Vit Niennattrakul, and Chotirat Ann Ratanamahatana. Parameter-Free Motif Discovery for Time Series Data. In *ECTI-CON*, 2012. (Cited on page 12)
- [144] Kajal Nusratullah, Shoab Ahmad Khan, Asadullah Shah, and Wasi Haider Butt. Detecting changes in context using time series analysis of social network. In *2015 SAI Intelligent Systems Conference (IntelliSys)*, pages 996–1001. IEEE, 2015. (Cited on page 3)
- [145] NVIDIA. Tesla K40 GPU Active Accelerator. *Board specification*, 2013. (Cited on pages 55 and 123)
- [146] Geraldo F Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. DAMOV: A new methodology and benchmark suite for evaluating data movement bottlenecks. *IEEE Access*, 9:134457–134502, 2021. (Cited on pages 10 and 120)
- [147] A. V. Oppenheim and R. W. Schaffer. From Frequency to Quefrency: a History of the Cepstrum. *IEEE Signal Processing Magazine*, 21:95–106, 2004. (Cited on page 89)
- [148] Pranav Patel, Eamonn Keogh, Jessica Lin, and Stefano Lonardi. Mining Motifs in Massive Time Series Databases. In *ICDM*, 2002. (Cited on pages 1, 12 and 113)



- [149] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities. In *PACT*, 2016. (Cited on page 21)
- [150] Ximing Qiao, Xiong Cao, Huanrui Yang, Linghao Song, and Hai Li. Atomlayer: a Universal Reram-based CNN Accelerator with Atomic Layer Computation. In *DAC*, 2018. (Cited on page 21)
- [151] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. Searching and Mining Trillions of Time Series Subsequences Under Dynamic Time Warping. In *KDD*, 2012. (Cited on pages 13 and 47)
- [152] SAFARI Research Group. Ramulator Source Code. <https://github.com/CMU-SAFARI/ramulator>. Accessed 23 September 2020. (Cited on pages 50 and 68)
- [153] Rajesh Saha, Yogendra Pratap Pundir, and Pankaj Kumar Pal. Comparative analysis of STT and SOT based MRAMs for last level caches. *Journal of Magnetism and Magnetic Materials*, 551:169161, 2022. (Cited on page 69)
- [154] Soheil Salehi and Ronald F DeMara. Energy and Area Analysis of a Floating-Point Unit in 15nm CMOS Process Technology. In *SoutheastCon*, 2015. (Cited on page 55)
- [155] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *ISCA*, 2013. (Cited on pages 50, 51 and 68)
- [156] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Fast Bulk Bitwise AND and OR in DRAM. *CAL*, 2015. (Cited on page 21)
- [157] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013. (Cited on page 21)
- [158] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory Accelerator for Bulk Bitwise

- Operations Using Commodity DRAM Technology. In *MICRO*, 2017. (Cited on page 21)
- [159] Vivek Seshadri and Onur Mutlu. Simple Operations in Memory to Reduce Data Movement. In *Advances in Computers*. Elsevier, 2017. (Cited on page 21)
- [160] Vivek Seshadri and Onur Mutlu. In-DRAM Bulk Bitwise Execution Engine. *arXiv*, 2019. (Cited on page 21)
- [161] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *ISCA*, 2014. (Cited on page 51)
- [162] R.H. Shumway. Applied statistical time series analysis. *Prentice-Hall, Englewood Cliffs.*, 1988. (Cited on page 3)
- [163] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gomez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In *FPL*, 2020. (Cited on page 21)
- [164] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NAPEL: Near-memory Computing Application Performance Prediction Via Ensemble Learning. In *DAC*, 2019. (Cited on page 21)
- [165] Avinash Sodani. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi Processor. In *HCS*, 2015. (Cited on pages 55 and 123)
- [166] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016. (Cited on page 35)
- [167] Hairong Song. Review of Time Series Analysis and Its Applications With R Examples , by Robert H. Shumway & David S. Stoffer: New York, NY: Springer, 2011, 596 pp., 2017. (Cited on pages III, 1 and 113)
- [168] Shao Yakun Sophia, Xi Sam Likun, Srinivasan Vijayalakshmi, Wei Gu-Yeon, and Brooks David. Co-Designing Accelerators and SoC Interfaces Using gem5-Aladdin. In *MICRO*, 2016. (Cited on page 51)

- [169] Robert Stoermer, Ralph Mager, Andreas Roessler, Franz Mueller-Spahn, and Alex H Bullinger. Monitoring human-virtual reality interaction: a time series analysis approach. *CyberPsychology & Behavior*, 3(3):401–406, 2000. (Cited on page 3)
- [170] Balázs Szigeti, Ajinkya Deogade, and Barbara Webb. Searching for motifs in the behaviour of larval drosophila melanogaster and caenorhabditis elegans reveals continuity between behavioural states. *Journal of The Royal Society Interface*, 12(113):20150899, 2015. (Cited on page 3)
- [171] A. Taddei, G. Distanti, M. Emdin, P. Pisani, G. B. Moody, C. Zeelenberg, and C. Marchesi. The European ST-T Database: Standard for Evaluating Systems for the Analysis of ST-T Changes in Ambulatory Electrocardiography. *European Heart Journal*, 1992. (Cited on pages 52, 58 and 89)
- [172] G. Tagliavini, A. Marongiu, and L. Benini. FlexFloat: a software library for transprecision computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39:145–156, 2020. (Cited on pages 25, 78 and 88)
- [173] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benin. A transprecision floating-point platform for ultra-low power computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018. (Cited on pages 24, 77 and 125)
- [174] Yoshiki Tanaka, Kazuhisa Iwamoto, and Kuniaki Uehara. Discovery of Time-Series Motif from Multi-Dimensional Data Based on MDL Principle. *Machine Learning*, 2005. (Cited on page 12)
- [175] Yoshiki Tanaka, Kazuhisa Iwamoto, and Kuniaki Uehara. Discovery of Time-Series Motif from MultiDimensional Data Based on MDL Principle. *Machine Learning*, 58:269–300, 2005. (Cited on page 3)
- [176] Heng Tang and Stephen Shaoyi Liao. Discovering Original Motifs with Different Lengths from Time Series. *Knowledge-Based Systems*, 2008. (Cited on page 12)
- [177] Texas Instruments. FRAM – New Generation of Non-Volatile Memory. [https://www.antaio.s.fr/IMG/pdf/web\\_site\\_sot\\_whitepaper.pdf](https://www.antaio.s.fr/IMG/pdf/web_site_sot_whitepaper.pdf). 2009. (Cited on page 23)
- [178] Sahar Torkamani and Volker Lohweg. Survey on Time Series Motif Discovery. *WIREs: Data Mining and Knowledge Discovery*, 2017. (Cited on page 12)

- [179] Sahar Torkamani and Volker Lohweg. Survey on time series motif discovery. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(2):e1199, 2017. (Cited on pages 12 and 69)
- [180] Sahar Torkamani, Volker Lohweg, F Hoffmann, and E Hüllermeier. Shift-Invariant Feature Extraction for Time-Series Motif Discovery. In *Workshop Computational Intelligence*, 2015. (Cited on page 12)
- [181] Daniel T. Trugman and Peter M Shearer. GrowClust: A hierarchical clustering algorithm for relative earthquake relocation, with application to the Spanish Springs and Sheldon, Nevada, earthquake sequences. *Seismological Research Letters*, 88(2A), 2017. (Cited on page 3)
- [182] Ruey S. Tsay. *Analysis of financial time series*. Wiley series in probability and statistics. Wiley-Interscience, 2005. (Cited on page 3)
- [183] UPMEM. *Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator (White Paper)*, 2018. (Cited on pages XII, 20 and 69)
- [184] R Vio, Niels Kristensen, Henrik Madsen, and W Wamsteker. Time series analysis in astronomy: limits and potentialities. *Astronomy and Astrophysics*, 435, 10 2004. (Cited on page 3)
- [185] EE Vogel, G Saravia, D Pastén, and V Muñoz. Time-series analysis of earthquake sequences by means of information recognizer. *Tectonophysics*, 712–713:723–728, 2017. (Cited on page 15)
- [186] T. Warren Liao. Clustering of time series data - A survey. *Pattern Recogn.*, 38(11):1857–1874, 2005. (Cited on page 3)
- [187] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. Metal-oxide RRAM. *Proceedings of the IEEE*, 2012. (Cited on page 21)
- [188] H-S Philip Wong and Sayeef Salahuddin. Memory leads the way to better computing. *Nature nanotechnology*, 10(3):191–194, 2015. (Cited on page 23)
- [189] Berlin Wu. Pattern recognition and classification in time series analysis. *Applied Mathematics and Computation*, 62(1):29 – 45, 1994. (Cited on page 3)
- [190] Xilinx. Xilinx runtime (xrt) documentation. <https://xilinx.github.io/XRT/>. Accessed Apr 21, 2022. (Cited on page 69)

- [191] Hongyi Xin, John Greth, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. Shifted Hamming Distance: A Fast and Accurate SIMD-friendly Filter to Accelerate Alignment Verification in Read Mapping. *Bioinformatics*, 2015. (Cited on page 21)
- [192] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Can Alkan, and Onur Mutlu. FastHASH: A New GPU-friendly Algorithm for Fast and Comprehensive Next-Generation Sequence Mapping. In *BMC Genomics*, 2013. (Cited on page 21)
- [193] Hao Yan, Hebin R. Cherian, Ethan C. Ahn, Xuehai Qian, and Lide Duan. iCELIA: A Full-Stack Framework for STT-MRAM-Based Deep Learning Acceleration. *IEEE Transactions on Parallel and Distributed Systems*, 31(2):408–422, 2020. (Cited on page 23)
- [194] Dragomir Yankov, Eamonn Keogh, Jose Medina, Bill Chiu, and Victor Zordan. Detecting Time Series Motifs Under Uniform Scaling. In *SIGKDD*, 2007. (Cited on page 12)
- [195] C M Yeh, H V Herle, and E Keogh. Matrix Profile III: The Matrix Profile Allows Visualization of Salient Subsequences in Massive Time Series. In *ICDM*, 2016. (Cited on pages 52, 58 and 89)
- [196] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. In *ICDM*, 2016. (Cited on pages 2, 12, 13, 41 and 114)
- [197] Sorrachai Yingchareonthawornchai, Haemwaan Sivaraks, Thanawin Rakthanmanon, and Chotirat Ann Ratanamahatana. Efficient Proper Length Time Series Motif Discovery. In *ICDM*, 2013. (Cited on page 12)
- [198] Shimeng Yu and Pai-Yu Chen. Emerging memory technologies: Recent trends and prospects. *IEEE Solid-State Circuits Magazine*, 8(2):43–56, 2016. (Cited on page 124)
- [199] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *HPDC*, 2014. (Cited on page 19)
- [200] Feichi Zhou and Yang Chai. Near-sensor and in-sensor computing. *Nature Electronics*, 3(11):664–671, 2020. (Cited on pages III, 1 and 113)

- [201] Huanzhou Zhu, Zhuoer Gu, Haiming Zhao, Keyang Chen, Chang-Tsun Li, and Ligang He. Developing a pattern discovery method in time series data and its GPU acceleration. *Big Data Mining and Analytics*, 1(4):266–283, 2018. (Cited on pages 6 and 117)
- [202] Yan Zhu, Chin-Chia Michael Yeh, Zachary Zimmerman, Kaveh Kamgar, and Eamonn Keogh. Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds. In *ICDM*, 2018. (Cited on pages 2, 13, 14, 15, 40, 41, 52, 77, 80, 114 and 125)
- [203] Zachary Zimmerman, Kaveh Kamgar, Nader Shakibay Senobari, Brian Crites, Gareth Funning, Philip Brisk, and Eamonn Keogh. Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond. In *SoCC*, 2019. (Cited on pages 13, 14, 16, 17, 50, 77, 82, 83 and 125)
- [204] Zachary Zimmerman, Kaveh Kamgar, Yan Zhu, N Shakibay Senobari, Brian Crites, Gareth Funning, Philip Brisk, and Eamonn Keogh. Scaling time series motif discovery with GPUs: breaking the quintillion pairwise comparisons a day barrier. In *Proceedings of the ACM Symposium on Cloud Computing*, 2018. (Cited on pages 2 and 114)
- [205] David Zuñiga-Noël, Alberto Jaenal, Ruben Gomez-Ojeda, and Javier Gonzalez-Jimenez. The UMA-VI dataset: visual-inertial odometry in low-textured and dynamic illumination environments. *The International Journal of Robotics Research*, 39(9):1052–1060, 2020. (Cited on page 89)