MasterMind: Many-Accelerator SoC Architecture for Real-Time Brain-Computer Interfaces

Guy Eichler
Dept. of Computer Science
Columbia University
New York, New York, USA
guyeichler@cs.columbia.edu

Luca Piccolboni
Dept. of Computer Science
Columbia University
New York, New York, USA
piccolboni@cs.columbia.edu

Davide Giri
Dept. of Computer Science
Columbia University
New York, New York, USA
davide_giri@cs.columbia.edu

Luca P. Carloni
Dept. of Computer Science
Columbia University
New York, New York, USA
luca@cs.columbia.edu

***Abstract*—Hierarchical Wasserstein Alignment (HiWA) is one of the most promising Brain-Computer Interface algorithms. To enable its real-time communication with the brain and meet low-power requirements, we design and prototype a Linux-supporting, RISC-V based SoC that integrates multiple hardware accelerators. We conduct a thorough design-space exploration at the accelerator level and at the SoC level. With FPGA-based experiments, we show that one of our area-efficient SoCs provides 91x performance and 37x energy efficiency gains over software execution on an embedded processor. We further improve our gains (up to 3408x and 497x, respectively) by parallelizing the workload on multiple accelerator instances and by adopting point-to-point accelerator communication, which reduces memory accesses and software-synchronization overheads. The results include comparisons with multi-threaded software implementations of HiWA running on an Intel i7 and ARM A53 as well as a projection analysis showing that an ASIC implementation of our SoC would meet the needs of real-time Brain-Computer Interfaces.**

***Index Terms*—BCI, HLS, HiWA, SVD, Sinkhorn, P2P, FPGA**

I. Introduction

Brain-Computer Interfaces (BCI) enable the bidirectional interaction between brain and computers. The field of BCI has gained significant traction, in both academia [1]–[4] and industry [5]–[7]. The BCI literature stresses the importance of meeting hard real-time constraints and a tight power budget when recording brain signals [1], [3]. In a full BCI-based system, however, we should consider real-time, scalability, and power constraints also for the algorithms that analyze the brain signals. Performing brain-data analysis in real-time would allow the system to use its results to control an actuator or send feedback to the brain. To enable the realization of BCI devices that can operate in a body area network (BAN) [4], the system performing brain-data analysis should meet the challenging real-time goal imposed by the $0.18sec$ reaction time of the brain [8], and the ultra-low-power constraint of $200mW$ on an average workload for BCI wearables [9].

Towards this goal, we present MasterMind, a many-accelerator system-on-chip (SoC) specialized for BCI algorithms. As shown in Fig. 1, MasterMind consists of a RISC-V processor and multiple instances of accelerators. The implementations of the accelerators result from our proposed multi-objective design-space exploration (DSE) approach, which combines accelerator-level and SoC-level explorations. The number of instances of each accelerator in the SoC is configurable at design time, so that the workload can be parallelized on multiple accelerator instances. MasterMind uses advanced

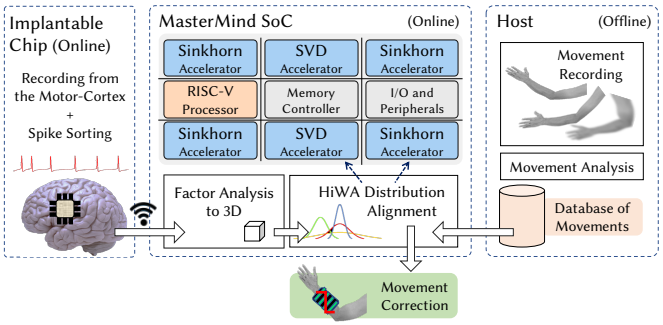

Fig. 1: MasterMind SoC in a Brain-Computer Interface system.

point-to-point (P2P) communication among accelerators that drastically reduces the memory accesses, with consequent performance and energy-efficiency benefits. Moreover, it allows the accelerators to work in a pipeline without any software synchronization overhead. Thanks to the energy efficiency and performance gains that hardware specialization provides, MasterMind represents a step forward in enabling the *online* deployment of the Hierarchical Wasserstein Alignment (HiWA), a promising machine-learning algorithm that can successfully decode brain signals into body movements [10]. To the best of our knowledge, MasterMind provides the first execution of HiWA that is based on hardware accelerators.

We deploy multiple versions of the MasterMind SoC on FPGA and evaluate them on executing the HiWA application. The FPGA-based experiments show the benefits of our solution over the RISC-V CVA6 processor [11], [12], a high-performance embedded processor from ARM (Cortex-A53), and a commercial processor from Intel (i7).

Our main contributions are:

1) A configurable accelerator-based SoC architecture with advanced support for parallelization and P2P communication to enable real-time and energy-efficient execution of BCI algorithms with a seamless flow for FPGA prototyping.
2) A broad DSE, and the resulting efficient implementation, of two hardware accelerators for Sinkhorn and SVD, the two main kernels of HiWA.
3) An innovative approach for the DSE of many-accelerator SoCs with P2P communication between accelerators.
4) A solution for a fast and energy efficient execution of the HiWA algorithm, as one of the first concrete steps towards the goal of real-time computation for BCI.
5) The open-source release of MasterMind.

```
1: global variables: p, q, m, γ, max_iter

2: function HiWA_core(X_{p×m}, Y_{q×m}, P_{1×1}, T_{m×m})
3:     Initialize Q_{p×q} = 1_{p×q}/(pq)
4:     Initialize R_{m×m} = 1_{m×m}
5:     while ||R − R_prev|| > 0.01 do
6:         R, XR, Y^T ← SVD(X, Y, T, P, Q)
7:         Q, sum(C ∗ Q) ← Sinkhorn(XR, Y^T)
8:     return R, sum(C ∗ Q)

9: function SVD(X, Y, T, P, Q)
10:    A = 2P(Y^T Q^T X) + T
11:    U, V, S = svd(A)
12:    return R = UV^T, XR, Y^T

13: function Sinkhorn(X, Y)
14:    C ← C(k, l) = ||X(k) − Y(l)||^2
15:    K = exp(−C/γ)
16:    Initialize b_{q×1} = 1_{q×1}/q
17:    for i from 0 to max_iter do
18:        a = 1_{p×1}/(p ⊙ Kb)
19:        b = 1_{q×1}/(q ⊙ K^T a)
20:        if (a, b) converged then
21:            break
22:    Q = diag(a) · K · diag(b)
23:    return Q, sum(C ∗ Q)
```

Notations:
$exp(\cdot), /, \odot, sum(\cdot), *$ are done elementwise
R_{prev} is the previous R
$diag(x)$ is a diagonal matrix with vector x as the diagonal
$svd(\cdot)$ is the original Singular Value Decomposition
$||\cdot||$ is $norm$

Fig. 2: The core of the HiWA algorithm.

II. Background

Fig. 1 shows the example of a full BCI-based system that is organized in two main phases. In an offline phase, body movements are recorded in 3D, analyzed, and stored in a database. In an online phase, electrical signals are recorded directly from the motor cortex of the brain by means of an implantable chip and sent to MasterMind. The data collected from the brain go through two processing steps, spike sorting [13] and factor analysis, before being decoded with HiWA, which produces a prediction of the subject's movement. This prediction is then sent outside of the SoC to be interpreted by a device that can help in cases of physical disabilities.

History of HiWA and Previous Results. HiWA was recently developed for machine-learning applications [10]. HiWA transforms a source dataset into a target dataset by minimizing a distance metric, and leverages the Sinkhorn algorithm [14] that enables highly efficient computation. HiWA was successfully applied for matching the neural activity from the brain of non-human primates to body movements in a process called Distribution Alignment [15]. *HiWA provides a proof of concept that the data recorded from the motor cortex of the brain can be matched to body movements recorded independently, and compared across time, space, and even different subjects.*

The Core of the HiWA Algorithm. Fig. 2 reports the inner loop of HiWA, which is the core part of the algorithm [10]. HiWA takes four main inputs (line 2). Matrix $X_{p\times m}$ represents a cluster of 3D points extracted from the multi-dimensional brain data (after applying Factor Analysis), while matrix $Y_{q\times m}$ is a cluster of 3D points from the database of body movements. Values p and q are the number of points in the two clusters and m is the number of dimensions represented in the datasets ($m = 3$). The scalar $P = 1/num\_total\_clusters$ is used as normalization factor in SVD. Matrix T is used as a step size, similar to the one used in gradient descent.

First, HiWA initializes Q (line 3) and R (line 4). Matrix Q encodes a correspondence between 3D points in X and Y (points $X(k)$ and $Y(l)$). Then, HiWA calls SVD and Sinkhorn in a loop that converges when a norm check on R is satisfied (line 5). R is used to rotate cluster X and to reduce its distance from Y. In SVD (line 9), matrix A represents the next rotation of cluster X towards cluster Y. It is calculated by multiplying the two clusters with their correspondence matrix Q, added with the step size T. Matrix A is then decomposed by the linear algebra method $svd()$ and assembled into matrix R by multiplying its singular vectors matrices U and V. By multiplying R with X, SVD rotates the cluster's data. Then, it returns R, XR, and Y^T. In Sinkhorn (line 14), matrix C represents the initial distances between 3D points in X and Y (points $X(k)$ and $Y(l)$). The global variable γ is a regularization parameter set by the user and used to produce matrix K from matrix C. The variable max_iter, which is also set by the user, determines the maximum number of iterations of Sinkhorn's loop. The loop (line 18) completes when a convergence check is satisfied on vectors a and b or the number of iterations reaches max_iter. The elementwise sum of $C*Q$ is the Sinkhorn distance between the two clusters. The Sinkhorn algorithm returns the realized sum, and the current correspondence matrix Q. The two outputs provide the current assessment of similarity between clusters X and Y. After the HiWA core computation completes on two clusters, matrix R and the distance $sum(C * Q)$ are returned and stored (line 8).

The algorithm of Fig. 2 is executed across all clusters pairs. Each result from a pair of clusters is completely independent from the other pairs, and thus can be analyzed in parallel. When all clusters have been analyzed, the results from all the combinations can be checked and the best match between the brain data and a movement can be determined (the best match is the shortest Sinkhorn distance between clusters).

III. SoC Architecture

Fig. 1 shows the tile-based architecture of the MasterMind SoC. The scalable architecture combines multiple accelerator tiles with one processor tile, one memory channel and one I/O tile to manage various peripherals. In particular, two accelerator tiles implement the two main computational kernels of the HiWA algorithm: Sinkhorn and SVD.

Our SoC comes with the full software stack for offloading the HiWA computation to the accelerators from a Linux user-space application. In Section VI, we show FPGA-based experiments on MasterMind SoCs with up to eight accelerators.

Target System. The MasterMind SoC is designed as the main computational component of a target BCI system (Fig. 1). On the left, an implantable chip, attached to the motor-cortex of the brain, records the neural activity using electrodes customized for the neural interface [16]. The recorded data are first processed with spike sorting [13] to clarify the occurrences

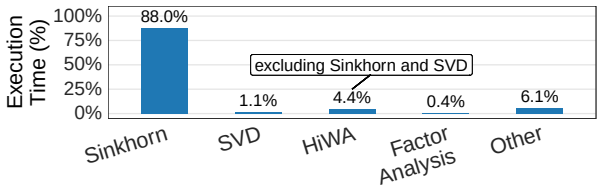

Fig. 3: Profiling of the HiWA application.

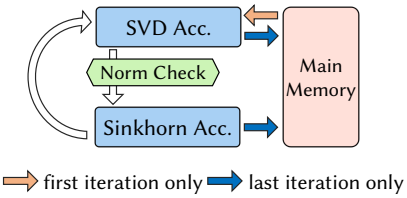

Fig. 4: Data-flow of the accelerators operation in the SoC.

of action potentials (spikes) in relevant neurons. In our target system, spike sorting takes place at the level of the implanted chip. MasterMind, however, can be scaled up to accommodate accelerators for spike sorting.

The implanted chip sends data through wireless communication to the device hosting the MasterMind SoC. A factor analysis application is executed by the processor tile on the SoC to transform the data from a multi-dimensional to a 3D representation. Then, MasterMind executes the HiWA application that compares the brain data to different body movements loaded from a database (Section II).

Eventually, this process finds the best matching movement, and the result can be forwarded to an external device with the goal of correcting disabilities. Alternatively, HiWA's results can be useful as part of a feedback loop that sends commands back to the implanted chip, which will actuate electrical stimulation through the electrodes attached to the brain. In this case, the stringent real-time constraint imposed by the reaction time of the brain [8] should be applied on the latency of the system.

SoC Design Platform. We designed MasterMind by leveraging ESP, an open-source platform for the agile design of heterogeneous SoCs and their prototyping on FPGA [17]–[19]. Given a set of hardware accelerators, ESP allows the designer to build a tile-based SoC with the desired mix and location of accelerator tiles, processor tiles, memory tiles (each containing a channel to main memory) and an I/O tile that manages various peripherals. ESP features a multi-plane network-on-chip (NoC) as the main on-chip interconnect. We selected the open-source 64-bit CVA6 RISC-V core [11], [12] as the main processor.

Software Profiling. We profiled the current state-of-the-art Python implementation of the HiWA application to find the best candidates for hardware acceleration [15]. In Section VI, we also evaluate our own optimized C++ implementation. Fig. 3 shows the execution time breakdown of running the HiWA application on the provided dataset. The Sinkhorn algorithm is the most time consuming task, accounting for 88% of the $12sec$ of total execution time. Instead, the SVD algorithm accounts for 1.1% of the execution time. The remaining part of the HiWA core function contributes an additional 4.4%, while the factor analysis step amounts to 0.4%. Finally, this application spends about 6% of the time for pre-processing tasks, extracting the data clusters and scaling them. The pre-processing tasks are not part of HiWA and can be tackled with various software implementations, which are not the focus of this work.

Based on the software profiling, we decided to implement hardware accelerators for Sinkhorn, the main bottleneck of the algorithm, and for SVD, a component that works in a loop with Sinkhorn. We embedded into our accelerators all the additional computation steps that contribute to the 4.4% of the execution time. Together, the accelerators allow us to isolate most of the computation and offload it into hardware. We designed a separate accelerator for SVD due to its computation complexity, short execution time, and reusability by other algorithms.

Data-Flow in the SoC. Fig. 4 depicts the data-flow of the HiWA algorithm running on the SoC. The SVD accelerator reads the data from the main memory and generates the inputs for the Sinkhorn accelerator, which in turn generates the next input for SVD. This process continues in a loop until matrix R, which is one of the outputs of SVD, satisfies a norm check. Finally, Sinkhorn stores the results back into main memory. This data-flow can be parallelized over multiple accelerators for a faster execution at the cost of larger energy consumption.

Accelerators Communication and Invocation. The accelerator tiles follow a loosely-coupled accelerator model [20], [21]. They are designed independently from the rest of the system, are connected directly to the system interconnect with a configurable socket, and execute coarse-grained tasks. The socket is configured through memory-mapped registers and handles tasks such as address translation, cache coherence, DMA, and P2P communication on behalf of the accelerators.

P2P communication allows direct data transfer between the accelerators and isolation of their computation from the processor and main memory [22]. On top of the basic P2P hardware support provided by ESP, we developed an advanced HW/SW P2P strategy to allow accelerators to work independently in a loop with minimum software assistance. At invocation time, the accelerators' configuration specifies across different iterations which data should be accessed with P2P and which data should be accessed regularly in main memory.

We leveraged the ESP software API to develop Linux device drivers for invoking our accelerators, and set up an on-chip direct P2P communication. The result is a major reduction of off-chip memory accesses, with consequent performance improvements and energy savings (Section VI).

The support for P2P in MasterMind can be very useful for a BCI system that executes long iterative computations involving multiple processing elements. For instance, Karageorgos *et al.* [1] suggest the main processing elements that are currently needed for common BCI tasks and their interactions with one another. These tasks in addition to HiWA, spike sorting, and factor analysis can be decomposed into several accelerators in MasterMind. The decomposition helps meeting low-power constraints for individual tasks, while direct data transfer significantly reduces interactions with the processor and main memory, and increases the general performance of the system.

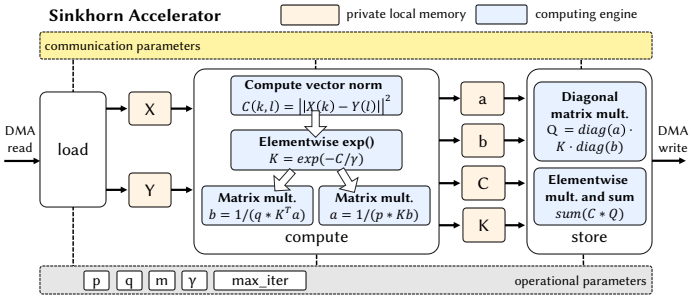

Fig. 5: The hardware architecture of the Sinkhorn accelerator.

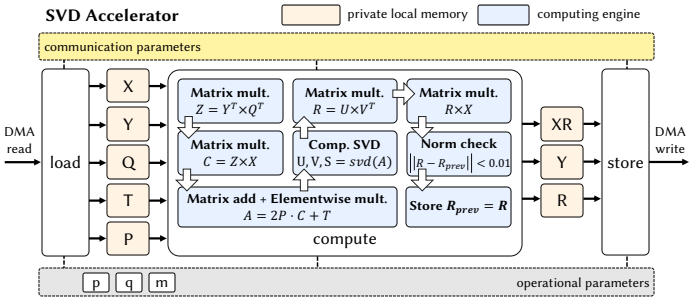

Fig. 6: The hardware architecture of the SVD accelerator.

IV. Accelerators Design

Fig. 5 and Fig. 6 show the architectures of our accelerators. We designed Sinkhorn in SystemC and synthesized it with Stratus HLS, while we designed SVD in C++ and synthesized it with Vivado HLS. The accelerators include private local memories (PLMs) to store inputs, outputs, and partial results [23]. Each accelerator can be configured to interact with main memory or directly with another accelerator for sending and receiving data. The PLMs are implemented as multi-bank memories that expose multiple read/write ports, and their size is configurable at design time.

The accelerators have several parameters, implemented as memory-mapped registers. They can be divided between operational parameters and communication parameters. The operational parameters are used as constants in the computation kernels. The communication parameters define how the accelerator communicates with the rest of the SoC and which functions must be executed in a particular invocation. In this way, we can handle dynamic changes in the communication at run-time and support the P2P feature of the SoC.

Each architecture consists of three main modules operating concurrently in pipeline [24]. The *load* function receives the input data into the PLMs; it uses a multiplexer to select the correct PLM according to the current offset in memory and the operational parameters. The *compute* function processes the data, and the *store* function emits the output data. Both accelerators use 32-bit fixed-point values that help improving performance and energy efficiency, without sacrificing the accuracy of the accelerators' computation ($\geq 97\%$).

Sinkhorn Accelerator. The accelerator implements the Sinkhorn algorithm that is responsible for most of HiWA's execution time. The operational parameters of the accelerator are: p, q, m, γ, and max_iter. They define the sizes of the matrices the accelerator takes as input, the regularization parameter (Section II), and the total number of iterations.

Fig. 5 shows the architecture of the Sinkhorn accelerator. The accelerator loads the input matrices X and Y. Matrix C is computed by parallelizing the computation of the vector norm between 3D-points in X and Y. To compute elements of matrix K, we implemented an efficient module for the $\exp(-x)$ operator as the approximation of the Taylor series. The module updates an initial value in a loop according to the recursive Taylor series formula, and uses only one 8-bit divisor to reduce area, a 32-bit multiplier, and a 32-bit adder.

The accelerator repeatedly uses K to calculate vectors a and b. The calculation of a depends on b and vice versa; it is done for a total of max_iter iterations, unless the vectors have converged. We managed to pipeline most of the computation by using multi-port memories to implement a, b, and K. The use of costly division operators was mostly avoided by reusing constants ($1/\gamma$), scheduling the division at the last stage of the computation, and implementing a Taylor series module for the inversion operator $1/x$. This module receives an initial value as input, and uses only one multiplier and one adder to calculate the output recursively. Compared to using a regular 32-bit divisor, this method significantly reduces area costs.

The final step is the calculation of matrix Q from K, a, and b, by transforming the vectors a and b into diagonal matrices and by multiplying them with K. Our diagonal matrix multiplication module multiplies three matrices, two of them diagonal, in a fully pipelined manner, while taking advantage of the unique structure of diagonal matrices. When an element of Q is ready, it is stored in a double buffer, multiplied by an element in C, and added to the total sum of the elementwise multiplication. The double buffer is used to preserve the full throughput of the pipeline. It masks the memory access latency in *store* by overlapping communication with computation, and it eliminates the need to store the entire output in a PLM.

SVD Accelerator. The accelerator rotates the cluster with the data from the brain before feeding it back to Sinkhorn. The operational parameters of the accelerator are m, p, and q, which define the sizes of its input matrices. To develop SVD, we used methods from the Vivado HLS Linear Algebra library.

Fig. 6 shows the architecture of the SVD accelerator. The accelerator loads as input four matrices (Q, X, Y, and T) and one scalar P. It calculates matrix A as the result of a chain of matrix operations: $2P(C) + T = 2P(ZX) + T = 2P(Y^T Q^T X) + T$, while the main bottleneck of the operations is the multiplication of matrix Y^T with matrix Q^T. For this operation, we used a large factor to unroll the main loop of the multiplication and gain a significant speedup. For other multiplications we used intermediate settings for a good balance on all resources, and used pipelined loops for the addition and elementwise multiplication.

The accelerator uses the linear algebra method $svd()$ on matrix A in order to generate its singular vectors matrices U and V, and it computes the rotation matrix $R = U \times V^T$. The $svd()$ method is another bottleneck of the accelerator

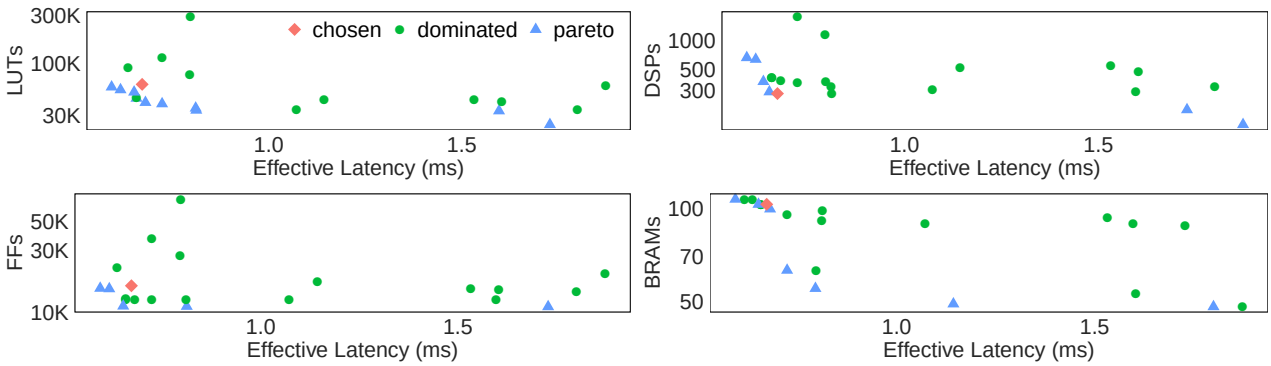

Fig. 7: Design-space exploration for the Sinkhorn accelerator from four different resource perspectives.

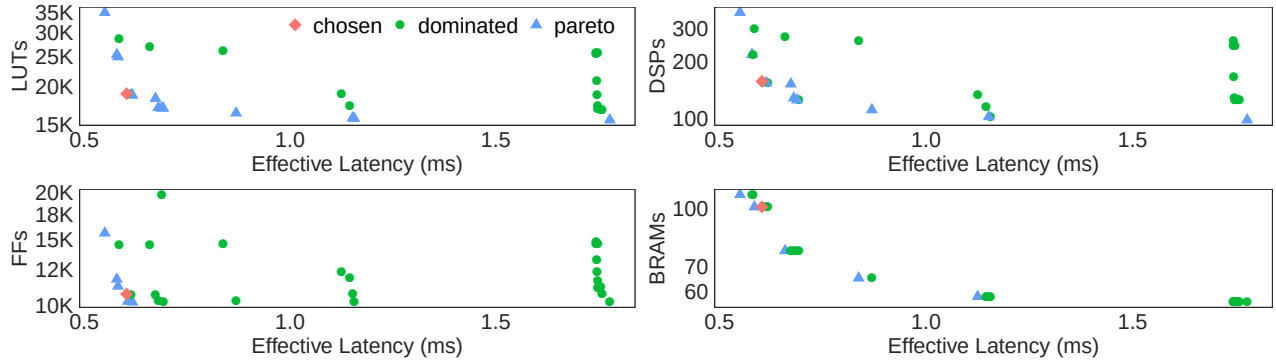

Fig. 8: Design-space exploration for the SVD accelerator from four different resource perspectives.

as it is an iterative operation with data dependencies. We customized $svd()$ by balancing the different initiation intervals of the module, in order to achieve an efficient pipelined implementation of the computation.

Finally, the accelerator rotates X by calculating $X \times R$. At the end of the computation, a norm check on R determines the convergence of R, and *store* writes the output into memory.

V. Design-Space Exploration

The exploration of different trade-offs in terms of performance and area is important for a BCI system with real-time performance requirements and strict area and power limitations. A broad DSE at the accelerator-level generates a group of implementations and increases the reusability of the accelerators under different constraints. A compositional DSE at the SoC-level produces a family of Pareto-optimal SoCs, revealing hidden elements in the coarse-grained computation and communication of the full system [24].

DSE at the Accelerators Level. In Sinkhorn, the *compute* function is divided into several computation modules that are executed in an efficient pipeline (Fig. 5). Inside the modules, we exposed additional parameters to determine the parallelism of read/write operations and multiplications.

After obtaining a baseline implementation for Sinkhorn, we explored the design-space of the accelerator. Specifically, we explored the application of loop unrolling, loop pipelining, function inlining, custom datapath components, different sizes for the local memories, and different implementations of the division operators in order to avoid resource overheads.

In SVD, we enriched the DSE of the accelerator by using the C++ traits made available by Vivado HLS, and we customized the implementations of the matrix operations modules.

In all the configurations, the Sinkhorn accelerator expects an input size of 5KB and an output size of 160KB. The SVD accelerator expects an input of 160KB and an output of 5KB. We recorded the latency as obtained by a cycle-accurate co-simulation with Stratus HLS for Sinkhorn, and Vivado HLS for SVD (with Xilinx Virtex UltraScale+ VCU118 FPGA as the target). We collected the resource usage reported by Vivado after synthesizing the accelerators.

Fig. 7 and Fig. 8 show the results of our DSE for Sinkhorn and SVD, respectively, where each design point is characterized in terms of effective latency and FPGA resources (LUTs, FFs, DSPs, and BRAMs). Each configuration for both SVD and Sinkhorn uses different combinations of knobs, each resulting in an implementation with unique performance and area trade-offs. *This aspect improves the reusability of our accelerators also in other BCI systems.*

For the experiments (Section VI), we chose a specific configuration for each of the accelerators that provides a good balance on FPGA resources (as marked in Fig. 7 and Fig. 8).

DSE at the SoC Level. The final goal is to integrate our accelerators in a complete SoC. Therefore, we propose a novel approach to evaluate all the possible combinations of the two accelerators. Our approach is based on simulating the computation time, while assuming the accelerators interact using P2P communication. For every possible pair of accelerators, i.e., a point of Fig. 7 and Fig. 8, we varied the number of accelerator instances that are deployed in the SoC. We focused

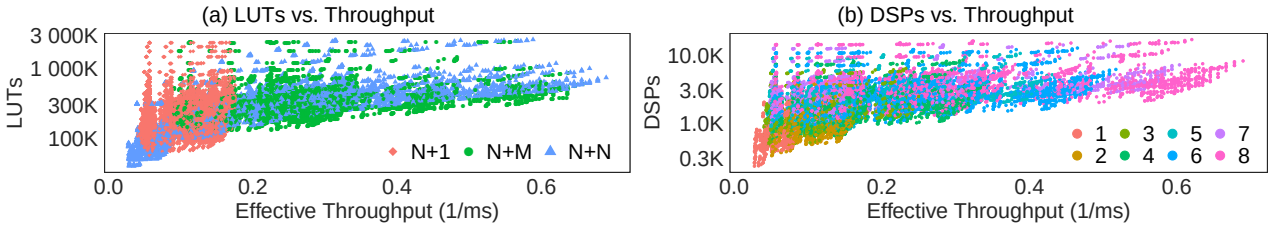

Fig. 9: Compositional design-space exploration.

only on the accelerators because our goal is to determine which combinations of accelerators are Pareto-optimal. We did not consider the area of the other SoC components, which remains constant across all the combinations.

To calculate the *area*, we summed the FPGA resources required by the selected accelerator implementations and we multiplied them by the number of accelerator instances. To calculate the *throughput*, we simulated the scheduling of the invocations of the accelerators in the SoC. As shown in Fig. 4, the data-flow of one task starts from one accelerator (SVD) and continues to the next (Sinkhorn), then it repeats for a certain number of iterations K. This loop can be parallelized if there are multiple instances of each accelerator, so that we can execute in parallel T tasks, each requiring K iterations. In the first and last iterations of the loop, the accelerators interact with the off-chip memory to read the input and store the output, respectively. Since these operations are performed by all possible combinations of the accelerators, we can assume that they have a fixed latency cost. We assume that the communication overhead when the accelerators exchange data directly with P2P is negligible. This approximation can be more precise by taking the traffic on the NoC into account.

We explored three types of SoC architectures. The first consists of one instance from one accelerator type and N instances of the other type. We call this architecture $N + 1$. The second architecture uses N instances of one type and M of the other type, assuming that N is larger than M. This architecture is called $N + M$. The third has N instances for each accelerator type and we refer to it as $N + N$. We reported values of N that vary from 1 to 8. For each configuration, we consider the total number of tasks T to complete to be equal to N, i.e., the maximum number of accelerators from one type, and we set the number of iterations for each task to 10. We decided to set the number of tasks to be equal to the maximum number of accelerators from one type (N) in order to be able to offload at least one task to each accelerator.

Fig. 9 reports the configurations of the simulated SoCs in terms of throughput and resource utilization. Each design point corresponds to a combination of accelerators with a given architecture (indicated by the color and the shape of the point).

Fig. 9(a) highlights the configurations according to the combination of accelerators, and reports the LUT utilization. The throughput for the $N + 1$ architectures is limited since one accelerator type is bounded to 1. When the throughput is low, these configurations dominate the Pareto curve. Except for the highest values of throughput, for any given value there is an $M + N$ architecture that achieves a lower resource utilization than the $N + N$ ones. In other words, the $M + N$ architectures dominate the Pareto curve. Fig. 9(b) highlights the configurations according to their maximum number of accelerators from one type (the value of N), and reports the DSP utilization. The results are similar for all resource types.

This information can be useful when designing BCI systems with a limit on the number of accelerators, for example when the communication bandwidth is bounded and the area is constrained.

VI. Evaluation

Experimental Setup. We evaluated our MasterMind SoCs on the Xilinx Virtex UltraScale+ VCU118 FPGA, with a clock frequency of $78MHz$, which is the frequency set by the ESP platform. We compared the FPGA prototypes of our SoCs, in terms of performance and energy efficiency, with three general-purpose 64-bit processors: Intel i7 8700K ($3.7GHz$), ARM Cortex-A53 ($1.2GHz$), and RISC-V CVA6 [11], [12]. We selected the Intel i7 as a high-performance core, the CVA6 soft core to represent the embedded systems domain (as part of the SoC prototype on FPGA), and the ARM core as an intermediate option in terms of performance and power consumption.

We synthesized MasterMind with Xilinx Vivado, which reports that the CVA6 processor tile consumes $0.14W$, while the SVD and Sinkhorn accelerator tiles have power consumptions of $0.11W$ and $0.09W$, respectively. We estimated a TDP of $78.6W$ (the nominal value is $95W$) for the Intel core and of $2.8W$ for the ARM core, based on their datasheets.

Our applications running on CVA6 leverage the ESP runtime API to invoke the accelerators. For each accelerator, the applications allocate buffers of the sizes that are expected by the accelerators.

Performance of the Accelerators. We first invoked our accelerators in isolation to validate them and to compare their execution time with the corresponding C++ implementations running on the three general-purpose processors. We developed software implementations of the Sinkhorn and SVD algorithms by leveraging the Eigen Linear Algebra library [25]. Fig. 10 reports in logarithmic scale the performance that our accelerators achieve compared to the corresponding software implementations running on the general-purpose processors. Compared to CVA6, the Sinkhorn accelerator achieves a $148\times$ speedup, while the SVD accelerator achieves a $844\times$ speedup. Compared to the other processors, Sinkhorn produces a $1.8\times$ slowdown with respect to Intel i7, and a $3\times$ speedup with respect to ARM A53. SVD produces slowdowns of $24\times$ and

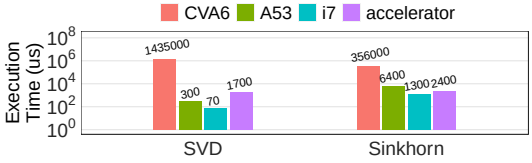

Fig. 10: Performance: accelerators vs. processors.

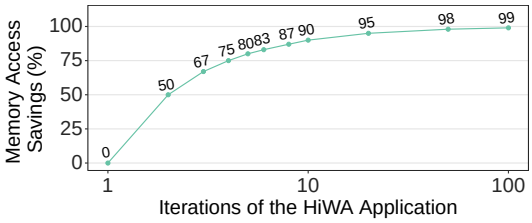

Fig. 11: Off-chip memory access savings with P2P.

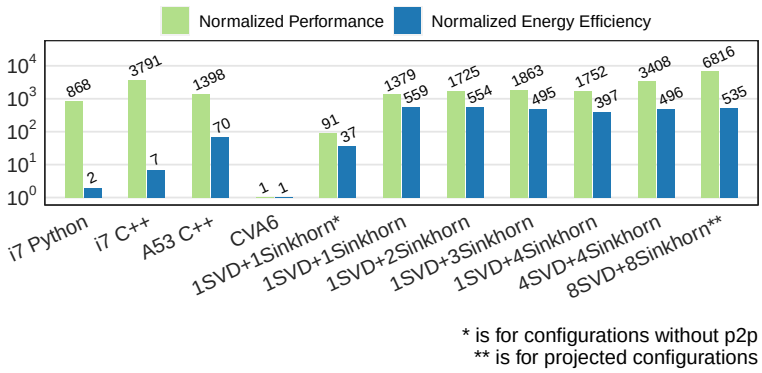

Fig. 12: Performance and energy efficiency: MasterMind SoCs vs. general-purpose processors.

$6\times$ compared to Intel i7 and ARM A53, respectively. The slowdowns of both accelerators are to be expected due to a significantly lower clock frequency on FPGA ($78MHz$) compared to the cores ($1.2 - 3.7GHz$).

Performance and Efficiency of the SoC. Our HiWA application invokes the accelerators in a loop on the MasterMind SoC (Fig. 4). We evaluated the execution of HiWA by scaling up the number of accelerators running in parallel and by experimenting with P2P communication. We ran our tests on SoCs containing up to eight accelerators, four per type, and compared our SoCs against the three processors. For Intel i7 and ARM A53, we developed multi-threaded software applications by using the POSIX Threads (Pthreads) library to exploit all available cores. For Intel i7, we also evaluated the original HiWA Python application [15].

Fig. 12 shows the performance and energy efficiency of running the HiWA applications on the three processors against multiple configurations of our MasterMind SoC. First, we evaluated two run-time configurations, one in which P2P communication is enabled and one in which it is disabled. Then, we increased the number of accelerators. The results across all four platforms (Intel i7, ARM A53, CVA6 and MasterMind SoC) have been normalized with respect to the CVA6. On Intel i7, our optimized C++ implementation is three orders of magnitude faster than the CVA6 core deployed on FPGA, while the Python implementation is two orders of magnitude faster. Even without P2P communication, the MasterMind SoC with one instance of each accelerator achieves $91\times$ speedup and $37\times$ on-chip energy saving over the CVA6 core. Enabling P2P improves these measures by 1500% ($1379\times$ and $559\times$, respectively), and the SoC surpasses the ARM core on both measures with $3\times$ speedup and $24\times$ energy efficiency.

Increasing the number of instances only for Sinkhorn helps us achieve a maximum performance gain of $1863\times$ with three instances, and a maximum energy efficiency improvement of $554\times$ with two instances. With four instances of Sinkhorn and one of SVD we encountered diminished returns. Adding more instances of SVD (for a total of four) increases the performance to $3408\times$. The MasterMind SoC with four instances for each accelerator achieves a similar performance to the Intel i7, with only 2% of the clock frequency, and $70\times$ better energy efficiency. The same SoC achieves an improvement of $2.4\times$ performance and $7\times$ energy efficiency with respect to the ARM core. In Fig. 12, we also report the projected performance of an SoC containing 8 SVD and 8 Sinkhorn accelerators. This SoC would be faster than the Intel i7 core even at $78MHz$ on FPGA, while still being $76\times$ more energy efficient.

Effectiveness of P2P Communication. Fig. 11 shows the reduction of off-chip memory accesses from enabling the P2P communication between the accelerators. The reduction is reported as a function of the number of HiWA iterations before convergence. In our experiments, we used a dataset for which the computation converges after nine iterations, resulting in 89% savings in memory accesses. Therefore, in addition to the gains in terms of on-chip energy efficiency, MasterMind can also minimize off-chip energy consumption, which is a major contributor to the overall energy dissipation of an SoC.

ASIC Projection. In our FPGA-based prototype setting, MasterMind achieves already an impressive improvement for HiWA compared to software-only executions on the Intel, ARM, and RISC-V cores. This result is obtained with a relatively low clock frequency ($78MHz$).

Although we prototyped the MasterMind SoCs on FPGA, the end goal is to build ASIC chips, which would have even better performance and energy efficiency than the FPGA prototypes we presented. We estimated the average power consumption of the Sinkhorn accelerator with Synopsys Design Compiler. For a $16nm$ LSTP ASIC technology node at $1V$, and with a clock frequency of $1GHz$, the Sinkhorn accelerator has an average dynamic power consumption of $14.64mW$, as opposed to the $90mW$ on FPGA. Combining this estimate with the increased clock frequency, we expect an ASIC implementation of a MasterMind SoC to achieve a further $12.8\times$ speedup and $78.8\times$ better energy efficiency over the FPGA prototype results in Fig. 12. In other words, the $2sec$ execution time of HiWA on Intel i7 will be shortened to roughly $0.15sec$ which is lower than the $0.18sec$ bound, and as much as 13 Sinkhorn tasks working in parallel will dissipate an average power under $200mW$ (Section I). *These further improvements reach the threshold for enabling real-time energy-efficient BCI systems.*

VII. Related Work

Multiple algorithms were proposed for analyzing brain-data [10], [26]–[28]. Lee *et. al* [10] present HiWA and provide its current state-of-the-art in software [15]. Its execution time with the available dataset is around $12sec$ on Intel i7. Our own software implementation takes $2sec$ on Intel i7 and $12sec$ on ARM A53. On FPGA, MasterMind achieves a performance similar to Intel i7, but with a much higher energy efficiency. As mentioned in Section VI, we expect even higher gains with ASIC technology. *To the best of our knowledge, MasterMind is the first hardware implementation of the HiWA algorithm.*

The SVD and Sinkhorn computation kernels are based on singular value decomposition and the Sinkhorn algorithm [14], respectively. Mohanty *et al.* [29] designed a fixed-point Jacobi SVD algorithm on a reconfigurable system. In their work, they analyzed the execution time and precision of a fixed-point type architecture called CORDIC on FPGA. They compared it to a SystemC simulation using both fixed-point and floating-point programs. They concluded that a future optimized implementation should be made using HLS on FPGA, as we did for the design of the SVD accelerator in MasterMind. *We are not aware of any hardware implementations for the Sinkhorn algorithm other than ours.*

In recent years, researchers have focused on the development of hardware-based BCI [1]–[3], [30]. The HALO project [1] uses a hardware-software co-design approach to design a low-power and general-purpose architecture for implantable BCI. This research development is part of the motivation of our efforts in developing MasterMind. Wahalla *et al.* [3] developed CereBridge, which is an FPGA-based real-time processing platform for mobile BCI. This paper shows a growing interest in hardware design for BCI systems, in order to meet the requirements of low power, real-time, and mobility.

There has been research on full BCI systems as well [4], [16], [31]. Miller *et al.* [16] reviewed the current state of BCI that aim to record reliable signals from the human brain surface by means of electrodes (ECoG). The review states that ECoG-based BCI may consist of the following components: (1) an implantable chip for real-time recording and digitization of brain signals; (2) wireless transmission of the digitized signal; (3) real-time analysis of the brain signals and translation into computer commands; and (4) a computerized device that receives these commands and performs an action on the patient. This vision matches the goals of our research work.

VIII. Conclusions

We designed MasterMind with the goal of advancing the system-level design research in SoC architectures for brain-computer interfaces (BCI), a field of computer engineering that is growing remarkably in importance. MasterMind is inherently flexible, as it can seamlessly accommodate the integration of many accelerators, and scalable, as it supports efficient point-to-point communication among accelerators that improves performance and energy efficiency. We released the contributions of this work in the public domain[1].

[1]https://github.com/GuyEichler/esp/tree/mastermind

IX. Acknowledgements

This work is partially supported by the DARPA NESD Program (C#: N6601-17-C-4002) and the National Science Foundation (A#: 1546296). The views and conclusions expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] I. Karageorgos et al. "Hardware-Software Co-Design for Brain-Computer Interfaces". In *Proc. of ISCA*, 2020.
[2] R. R. Shrivastwa et al. "An FPGA-Based Brain Computer Interfacing Using Compressive Sensing and Machine Learning". In *Proc. of ISVLSI*, 2018.
[3] M. N. Wahalla et al. "CereBridge: An Efficient, FPGA-based Real-Time Processing Platform for True Mobile Brain-Computer Interfaces". In *Proc. of EMBC*, 2020.
[4] G. Udovičić et al. "Wearable Technologies for Smart Environments: A Review with Emphasis on BCI". In *Proc. of SoftCOM*, 2016.
[5] "Neuralink". https://neuralink.com/.
[6] "CTRl-labs - Non-Invasive Neural Interfaces". https://www.ctrl-labs.com/.
[7] "OpenBCI - Open Source Tools for Neuroscience". https://openbci.com/.
[8] R. J. Kosinski. "A Literature Review on Reaction Time". *Clemson University*, 2008.
[9] A. Y. Dogan et al. "Multi-Core Architecture Design for Ultra-Low-Power Wearable Health Monitoring Systems". In *Proc. of DATE*, 2012.
[10] J. Lee et al. "Hierarchical Optimal Transport for Multimodal Distribution Alignment". In *Proc. of NeurIPS*, 2019.
[11] F. Zaruba et al. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". *ITVL*, 2019.
[12] "CVA6 RISC-V CPU". https://github.com/openhwgroup/cva6.
[13] M. S. Lewicki. "A Review of Methods for Spike Sorting: The Detection and Classification of Neural Action Potentials". *Network: Computation in Neural Systems*, 1998.
[14] M. Cuturi et al. "Sinkhorn Distances: Lightspeed Computation of Optimal Transport". In *Proc. of NeurIPS*, 2013.
[15] "Neuralign - All Things Neural Distribution Alignment". https://nerdslab.github.io/neuralign/.
[16] K. J. Miller et al. "The Current State of Electrocorticography-Based Brain-Computer Interfaces". *AANS: Neurosurgical focus*, 2020.
[17] P. Mantovani et al. "Agile SoC Development with Open ESP". In *Proc. of ICCAD*, 2020.
[18] "ESP - Open SoC Platform". https://esp.cs.columbia.edu/.
[19] L. P. Carloni. "The case for Embedded Scalable Platforms". In *Proc. of DAC*, 2016.
[20] J. Cong et al. "Architecture Support for Accelerator-Rich CMPs". In *Proc. of DAC*, 2012.
[21] E. G. Cota et al. "An Analysis of Accelerator Coupling in Heterogeneous Architectures". In *Proc. of DAC*, 2015.
[22] D. Giri et al. "ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning". In *Proc. of DATE*, 2020.
[23] C. Pilato et al. "System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip". *TCAD*, 2017.
[24] L. Piccolboni et al. "COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators". *TECS*, 2017.
[25] "Eigen API". http://eigen.tuxfamily.org/dox/index.html.
[26] E. L. Dyer et al. "A Cryptography-Based Approach for Movement Decoding". *Nature BME*, 2017.
[27] A. D. Degenhart et al. "Stabilization of a Brain-Computer Interface via the Alignment of Low-Dimensional Spaces of Neural Activity". *Nature BME*, 2020.
[28] K. Wang et al. "Enhance Decoding of Pre-Movement EEG Patterns for Brain-Computer Interfaces". *JNE*, 2020.
[29] R. Mohanty et al. "Design and Performance Analysis of Fixed-point Jacobi SVD Algorithm on Reconfigurable System". In *Proc. of ICACC*, 2013.
[30] M. Aravind et al. "Embedded Implementation of Brain Computer Interface Using FPGA". In *Proc. of ICETT*, 2016.
[31] R. P. N. Rao. "Towards Neural Co-Processors for the Brain: Combining Decoding and Encoding in Brain-Computer Interfaces". *Elsevier: Current opinion in neurobiology*, 2019.