C4CAM: A Compiler for CAM-based In-memory Accelerators

Hamid Farzaneh
TU Dresden
Dresden, Germany
hamid.farzaneh@tu-dresden.de

João Paulo C. de Lima
TU Dresden and ScaDS.AI
Dresden, Germany
joao.lima@tu-dresden.de

Mengyuan Li
University of Notre Dame
Notre Dame, Indiana, USA
mli22@nd.edu

Asif Ali Khan
TU Dresden
Dresden, Germany
asif_ali.khan@tu-dresden.de

Xiaobo Sharon Hu
University of Notre Dame
Notre Dame, Indiana, USA
shu@nd.edu

Jeronimo Castrillon
TU Dresden, ScaDS.AI and
Barkhausen Institut
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Abstract

Machine learning and data analytics applications increasingly suffer from the high latency and energy consumption of conventional von Neumann architectures. Recently, several in-memory and near-memory systems have been proposed to overcome this *von Neumann* bottleneck. Platforms based on *content-addressable memories* (CAMs) are particularly interesting due to their efficient support for the search-based operations that form the foundation for many applications, including K-nearest neighbors (KNN), high-dimensional computing (HDC), recommender systems, and one-shot learning among others. Today, these platforms are designed by hand and can only be programmed with low-level code, accessible only to hardware experts. In this paper, we introduce C4CAM, the first compiler framework to quickly explore CAM configurations and seamlessly generate code from high-level TorchScript code. C4CAM employs a hierarchy of abstractions that progressively lowers programs, allowing code transformations at the most suitable abstraction level. Depending on the type and technology, CAM arrays exhibit varying latencies and power profiles. Our framework allows analyzing the impact of such differences in terms of system-level performance and energy consumption, and thus supports designers in selecting appropriate designs for a given application.

ACM Reference Format:
Hamid Farzaneh, João Paulo C. de Lima, Mengyuan Li, Asif Ali Khan, Xiaobo Sharon Hu, and Jeronimo Castrillon. 2024. C4CAM: A Compiler for CAM-based In-memory Accelerators. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3620666.3651386

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651386

1 Introduction

Search operations come in numerous forms at the heart of many comparison-intensive applications. In the past decade, the revolution in machine learning, data analytics, and bioinformatics has played a significant role in driving the demand for efficient hardware acceleration of these operations. Domains such as network security [12], bioinformatics [45], data mining, and data analytics [53] heavily rely on *exact matching* of the query pattern with pre-stored patterns. In other applications, such as K-nearest neighbors (KNN) and genome analysis [15, 39], the emphasis lies on identifying similarities rather than exact pattern matching. In approximate search, when the dissimilarity between a stored pattern and the query pattern is within a predefined threshold, the stored pattern is regarded as a "match". From the computational standpoint, both exact and approximate search operations are time-consuming and are often bottlenecks in comparison-intensive kernels [50].

Recently, there has been a surge in the adoption of *content addressable memories* CAM-based system designs for efficient search operations. Originally employed in network routing and CPU caching [40], CAMs have now found applications in a wider range of data-intensive domains [15, 32, 39]. CAMs allow massively parallel search operations for an input query, enabling the search to be performed across the entire memory with a single operation. CAM's high-speed parallel search makes it a popular component for constructing cutting-edge *compute-in-memory* (CIM) systems, aiming to provide an energy-efficient alternative to the von Neumann bottleneck in terms of both latency and energy consumption.

CAM designs are broadly classified into binary, ternary, multi-state, and analog CAMs (BCAM, TCAM, MCAM, ACAM, respectively), with implementations based on either conventional CMOS or emerging non-volatile memory (NVM) technologies [6, 34, 39, 50]. Compared to CMOS technologies, NVM technologies, like magnetic RAM (MRAM), resistive RAM (ReRAM), or ferroelectric (FeFET), are denser and

more energy efficient, yielding more efficient CAM arrays [14, 16, 37]. BCAMs and TCAMs use a bit-wise Hamming distance (HD) to compare the query and stored data, whereas MCAMs and ACAMs apply a specific distance metric to compare the query with memory entries and determine which memory entries match the query based on the distance metric. In terms of match types, CAMs can be classified into the exact match (EX), best match (BE), and threshold match (TH) [16].

Although CAM designs have shown better performance than traditional methods for computing similarity in many domains [15, 32, 39], effectively mapping applications written in high-level programming languages onto CAM-based accelerators remains a challenge. This is due to the disparity in the abstractions of the applications (high-level) and the (low-level) set of commands needed to program the CAM arrays. Presently, CAM arrays are programmed manually with low-level code that only the device experts understand. Existing design automation and compilation tools for in-memory computing [46, 47] do not provide support for CAM primitives, highlighting the need for solutions that can support mapping a wider range of applications and accelerate the design process.

This paper proposes C4CAM, the first end-to-end automated retargetable framework that enables efficient mapping of applications from a higher TorchScript program onto CAM arrays. C4CAM leverages the multi-level intermediate representation (MLIR) framework to seamlessly optimize and offload comparison-intensive kernels to CAM-enabled systems. Concretely, we make the following contributions:

- An automated end-to-end compilation flow that (i) makes CAM accelerators accessible to non-experts and (ii) enables device/circuit/architecture experts to explore design trade-offs. C4CAM takes applications written in TorchScript along an architectural model for retargetability and generates code for the given architecture (see Section 4).
- An extension to the MLIR front-end to express search operations in PyTorch applications (see Section 4.3).
- Extension to the CIM abstraction from [26] to cater to CAM accelerators. Specifically, we propose analyses to detect computational primitives in applications that can be rewritten as search operations (see Section **4.4.1**).
- A novel CAM abstraction that supports different CAMs types and search operations (see Section **4.4.2**).
- Transformation passes to optimize for latency, power, and array utilization (see Section **4.4.2**).
- A comprehensive evaluation of the generated code, including validation and comparison to a GPU target and the hand-crafted implementations (see Section 5).

Our evaluation of C4CAM demonstrates that the generated code achieves comparable results to hand-crafted designs. We also showcase the capabilities of C4CAM in performing design space exploration on different CAM architectures.

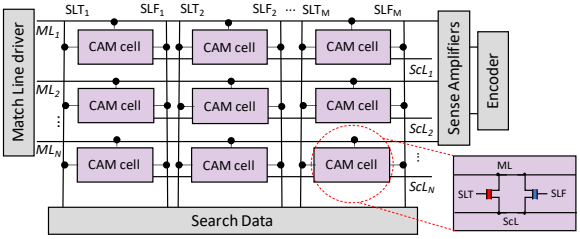


Figure 1. Structure of a FeFET-based CAM array [51]

2 Background

This section presents background on the MLIR framework and CAM-based structures and describes our proposed architecture. It also motivates the need for automatic compilation tools by explaining the challenges in the state-of-the-art programming models for CAMs.

2.1 MLIR compiler infrastructure

MLIR is a framework that enables representing and transforming intermediate representations (IR) at various abstraction levels, catering to diverse application domains and heterogeneous hardware targets [30]. It offers a customizable IR with minimal built-in features, enabling compiler developers to incorporate their own abstractions. This empowers them to optimize for specific domains or targets by leveraging matching techniques at the appropriate levels of abstraction.

MLIR consists of a collection of reusable abstractions organized into *dialects*. Each dialect incorporates custom types, operations, and attributes, which serve as fundamental building blocks of the IR. In MLIR, values are associated with compile-time known types, while attributes provide compile-time information linked to operations. Dialects in MLIR maintain preconditions for transformation validity within their IR, reducing the complexity and cost of analysis passes. Dialects are typically designed for specific domains (e.g., `linalg` for linear algebra, `TOSA` for tensor operations), representations (e.g., `affine` for the polyhedral model, `scf` for control flow), or targets (e.g., `gpu`, `cim`). The `llvm` dialect models LLVM IR constructs. Abstractions in MLIR can be progressively lowered (from high-level domain-specific dialects to low-level platform-specific dialects) and raised [9].

2.2 Content addressable memories

CIM fabrics are generally categorized into three classes: CIM-crossbars, renowned for their ability to compute matrix-vector multiplications in constant time; CIM-logic, facilitating the acceleration of bulk bitwise logic operations; and content addressable memories (CAMs), enabling fast and energy-efficient search operations [25].

CAMs support two main functions: search, which identifies the memory entries that match the input query, and write, which stores data entries in the memory cells. With CAMs, parallel searches can be performed on all stored data in

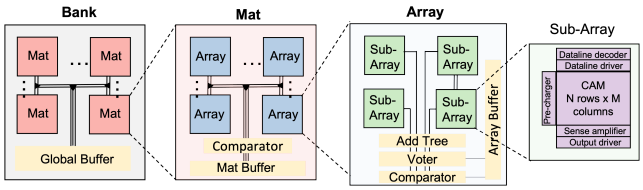


Figure 2. Hierarchical structure of a CAM-based accelerator

memory in constant time (O(1)). The most common type of CAM is the ternary CAM (TCAM), where the data elements can be either 0, 1, or don't care ('x'), which is a wildcard state matching both 0 and 1. Figure 1 illustrates a TCAM array with R rows and C columns. Each cell in a row is connected to a common match line (ML) and stores one of the tree states. During a search operation, each cell C_{ij} in row i performs an XNOR operation between its content and the query element q_j. The ML implements a logic OR operation of all the cells in the row to determine the result for that row.

Different sensing circuits can be designed to realize different match schemes, such as EX, BE, and TH. EX search is the fastest search type due to its simple sensing requirement, whereas best match search reports the row with the least number of mismatching cells and is widely used for nearest neighbor search. To find the best match, more sophisticated sensing circuits are needed, e.g., analog-digital-converters or a winner-take-all circuit, with the latter being more energy and area-efficient but limited to finding the best matches only within a certain number of mismatch cells [20].

2.3 Accelerator architecture

For this work, we consider a general CAM-accelerator design based on the state-of-the-art [23]. As illustrated in Figure 2, the CAM structure is organized into a four-level hierarchy comprising B banks, each bank containing T mats where each mat consists of A CAM arrays which are further partitioned into S subarrays. The subarrays can be operated and accessed independently. This hierarchical organization allows for scalable and flexible computation, as the number of banks, mats, and arrays can be allocated according to the computational requirements of each application. Within each bank, all mats and arrays can perform parallel search operations using the S CAM subarrays either in a sequential or parallel manner, providing further granularity for parallel processing and resource allocation. B banks operate independently to allow for task-level parallelism. RecSys [32], for instance, can profit from CAMs in both filtering and ranking stages, where each stage executes different tasks on different banks in parallel.

3 Related work

CAM's efficient data retrieval capabilities make it highly suitable for applications that rely heavily on large-scale matching or search operations. CAMs have been proposed based on various memory technologies including SRAM [5], resistive RAM [17], racetrack memory [11], and FeFET [39]. Recent works have demonstrated the use of CAMs in various fields, e.g., bioinformatics [8], high dimensional computing [23], reinforcement learning [31], few-shot learning [28] and recommender systems [32].

In terms of programmability, CAMs have received relatively little attention compared to other CIM paradigms. Prominent frameworks like TVM [10] and EXO [18] offer high-level programming abstractions and optimization passes for kernels, primarily from the machine learning domain, for CPU/GPU systems and traditional HW accelerators. Today, however, these high-level frameworks offer no support for CIM systems. In fact, CIM support in compilers and programming frameworks is scarce, with most approaches focused on accelerating neural networks on CIM crossbars. For example, OCC [46] automatically identifies the matrix multiplication (matmul) pattern, transforms kernels to match it, and offloads them to a PCM-crossbar. However, it does not support the diversity in crossbar technologies and architectures. To address this, CIM-MLC [43] is proposed, which considers the hierarchical structure of the CIM hardware and generates efficient code for it. For CIM-logic, Soeken et al. [47] introduced a compiler leveraging majority inverter graphs to enable operation rewriting and optimization for a given RRAM CIM accelerator. Recently, Jin et al.[21] proposed a compilation flow supporting both RRAM-based crossbar and logic architectures. Unlike CIM-logic and crossbars, there is currently no automatic general-purpose compilation framework targeting CIM-CAMs and generating efficient code for them.

The fundamental programming model of the CIM abstraction introduced in CINM [26] is generic and designed to be easily retargetable to different CIM architectures. CINM, however, primarily focuses on arithmetic and logic operations and does not support the search-based operations of CAMs (e.g., for computing distances, similarities, or comparisons). For CAM-based accelerators, frameworks such as DT2CAM [44] and X-TIME [41] exist to map and simulate decision trees onto TCAMs and ACAMs, respectively. However, these mapping tools do not generalize to other comparison-intensive kernels and require programmers to deeply understand both the application and the accelerator architecture. Therefore, there is a considerable demand for a generalized framework capable of efficiently lowering high-level language programs for diverse input applications. This framework should also incorporate CAM array optimizations, e.g., selective row pre-charging, to generate optimized code for the underlying architecture. In the following section, we introduce how the hierarchical C4CAM framework effectively addresses this gap.

4 The C4CAM framework

This section presents C4CAM, including the abstractions, lowerings, analysis, and optimization passes.

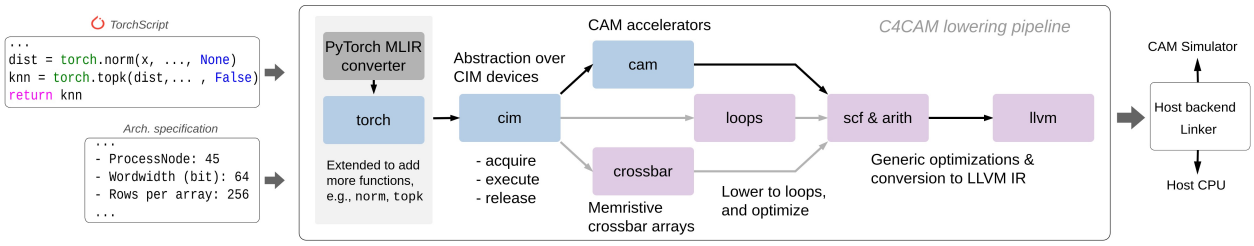


Figure 3. A high-level overview of C4CAM flow

4.1 An overview of the compilation flow

Figure 3 shows a high-level overview of C4CAM. The TorchScript functions chosen for offloading to the CAM accelerators are transformed into MLIR's representation using the PyTorch MLIR converter (see Section 4.3). This produces the `Torch` IR, which is the entry point into C4CAM, which includes ATen tensor library operations. Torch MLIR is then lowered to the `cim` abstraction, which is a comprehensive dialect for various CIM technologies, taking over the shared responsibilities of host-device interfacing and device mapping (see Section **4.4.1**). The `cim` abstraction has been previously investigated in [46] and [26], where a programming model for CIM devices was introduced. C4CAM extends this abstraction by incorporating the necessary analysis for CAM devices.

To enable application mapping, `cim` supports partitioning, rewriting, and kernel modifications. The latter transforms the code to use device-compatible sizes and operations, which the low-level dialects can then further process. Subsequently, the `cim` dialect is either lowered to `cam`, `loop`, or other device dialects. The `cam` dialect and other device dialects at the same level, such as `crossbar`, offer an abstraction for programming and executing functions on the target device (see Section **4.4.2**). The `cam` dialect also provides transformation passes that enable mapping and optimization of the selected kernel while accounting for the concrete hierarchy and other characteristics of CAM-based architectures.

4.2 Architecture specification

In addition to the input application, C4CAM takes the architectural configuration as input, as shown in Figure 3. This outlines the hierarchy of the proposed architecture (as discussed in Section 2.3), as well as the access mode for each level of the hierarchy, whether it supports sequential or parallel accesses. Note that all active rows within a subarray are accessed in parallel. However, through selective row accessing [55], it is possible to activate and pre-charge only a subset of rows within a subarray. This specification makes it easy to retarget C4CAM for different CAM accelerators. Apart from the architecture description, this input file specifies the optimization target, which can be set to latency, power, or array utilization.

4.3 C4CAM front end

The PyTorch MLIR converter [13] is responsible for converting Python code written in TorchScript. However, certain operations from the ATen library, particularly those used in search-based applications such as `norm` and `topk`, are not supported. From the CAM perspective, these are essential primitives in any input application. Since C4CAM is built upon the MLIR framework and this is the only available front-end that enables lowering TorchScript input to the MLIR torch dialect, we extend the front-end to support the `norm` and `topk` primitives that are commonly accelerated on CAM arrays.

To implement the benchmarks in Section 5, TorchScript was used. However, C4CAM is not confined only to TorchScript and the PyTorch MLIR Converter as its front-end. Representing every CAM-suitable application in TorchScript may not be straightforward (such as DNA mapping). For that, it is also possible to use alternative flows, such as *ONNX-MLIR* [4] and *supported importers in IREE [3]*. Such flows can also serve this purpose by implementing necessary conversions from their corresponding dialect to `cim`.

4.4 C4CAM progressive lowering

The compilation flow begins with the `Torch` dialect, as depicted in Figure 3. This dialect includes most of the ATen tensor library operations. To enable support for the `cim` abstraction, we have introduced a `torch-to-cim` conversion pass. This pass lowers the operations that are compatible with the `cim` abstraction. Examples of these operations include `topk`, `norm`, `sub`, and `matmul`, which can be executed individually or as part of a kernel on a CIM device.

To demonstrate how the IR of the application is transformed at each hierarchy level, we use the similarity kernel in hyperdimensional computing (HDC) as a running example. Figure 4a shows the TorchScript code of the input kernel. Figure 4b presents its MLIR representation at the `Torch` abstraction as produced by the MLIR PyTorch front-end. The conversion from `Torch` to `cim` is accomplished with target-agnostic transformations. The outcome of the conversion primarily showcases the interface with a generic CIM device.

4.4.1 The extended `cim` abstraction. `cim` is a high-level dialect for device-supported operations that includes essential transformations required to prepare kernels to run on target

```
def forward(self, input: Tensor, dot: bool = False)
    ↪ -> Tensor:
    others = self.weight.transpose(-2, -1)
    matmul = torch.matmul(input, (others))
    values, indices = torch.ops.aten.topk(matmul, 1,
        ↪ largest=False)
    return indices
```

(a) PyTorchScript code for HDC dot similarity

```
%1 = torch.aten.transpose.int %0, %int-2, %int-1 : !
    ↪ torch.vtensor<[10,8192],f32>, !torch.int, !
    ↪ torch.int -> !torch.vtensor<[8192,10],f32>
%2 = torch.aten.mm %arg0, %1 : !torch.vtensor
    ↪ <[10,8192],f32>, !torch.vtensor<[8192,10],f32
    ↪ > -> !torch.vtensor<[10,10],f32>
%values, %indices = torch.aten.topk %2, %int1, %int
    ↪ -1, %false, %true : !torch.vtensor<[10,10],
    ↪ f32>, !torch.int, !torch.int, !torch.bool, !
    ↪ torch.bool -> !torch.vtensor<[10,1],f32>, !
    ↪ torch.vtensor<[10,1],f32>
```

(b) `Torch` IR for HDC dot similarity

Figure 4. Python and MLIR representations of HDC similarity

devices. This abstraction is mainly responsible for: (i) analyzing the input code to identify CIM-amenable primitives that can be offloaded to the accelerator, (ii) if a CIM-executable pattern is identified but the operand sizes exceed the array sizes specified in the given architecture, dividing the input into smaller partitions to ensure compatibility with the accelerator, and (iii) providing an abstract programming model to enable the execution of kernels on a device.

The programming model of the `cim` abstraction is based on three main functions. To allocate an accelerator, `cim` uses the `cim.acquire` function that returns a handle to the device. The `cim.execute` function uses this handle and specifies the operations that are to be executed on this accelerator. Finally, the device is released using the `cim.release` function. For CAM architectures, we show how these functions are lowered to different CAM functions in Section **4.4.2**. Figure 5a shows the IR for the running example at the `cim` abstraction, which can be produced by running the conversion pass `torch-to-cim` at the `Torch` abstraction. As the `Torch` abstraction does not, and is not supposed to, specify the kernel type, the fundamental assumption of the `torch-to-cim` conversion is that each supported operation can be executed on a separate (non-)CIM device. Since all the torch operations are supported by the `cim` dialect (as they are part of the dot similarity), they are lowered to their corresponding `cim` versions.

Pattern matching and fusing: `cim` implements analysis and optimization passes to recover patterns that can be offloaded to a CIM accelerator and, when possible, optimizes them for the target. The analysis pass identifies blocks containing operations that cannot be directly lowered to the accelerator and fuses them. Once the code analysis is completed, the execution blocks can be transformed and offloaded to CIM accelerators, or they can follow the standard MLIR pipeline to generate `llvm` code for execution on the host processor.

Algorithm 1: Similarity search detection.

```
1  /* Pattern matching for dot product similarity */
2  Replace op<topk> (op<matmul> (arg2, op<transpose> (arg1)), arg3)
3    with op <similarity> (dot, arg1, arg2, arg3);
4  /* Pattern matching for Euclidean distance */
5  Replace op <topk>(op<norm> (op<sub> (arg1, arg2)), arg3)
6    with op <similarity> (euc, arg1, arg2, arg3);
7  /* Pattern matching for cosine similarity */
8  Replace op <smulmat>(op <div> (cons1, op<mul>(op<norm>(arg1),
     op<norm> (arg2))), op<matmul>(arg1, op<transpose>(arg2)))
9    with op <similarity> (cos, arg1, arg2);
10 /* Pattern matching for Hamming distance */
11 Replace op <nonzero>(op <cmp>(lt, op <popcount>(op <xor>(arg1,
     arg2), arg3)
12   with op <similarity> (ham, arg1, arg2, arg3);
```

Algorithm 1 illustrates the function designed for pattern matching within an execution block, tailored explicitly for identifying various similarity search operations. This function assesses whether a given data flow graph aligns with pre-defined supported patterns and replaces it with the corresponding similarity search operation.

The pattern matching for dot product, Euclidean norm, cosine, and Hamming similarity are defined in Lines 2, 5, 8, and 11, respectively. Algorithm 1 is the simplified illustration of `cim-fuse-ops` pass, which, when enabled with the similarity flag indicating the type of search, identifies code blocks that match the criteria and subsequently replace their operations with the `cim.similarity` operation. Figure 5a shows the base `cim` IR produced by the `torch-to-cim` conversion pass, while Figure 5c showcases the result obtained after applying the `cim-fuse-ops` pass to Figure 5b.

Compulsory partitioning: Kernels often require more space than what the processing elements (PE) of the target can support. To overcome this limitation, the kernel is partitioned according to the size supported by a PE. In a CAM system, the smallest block within the system is the subarray. Therefore, when partitioning the application, it is important to consider this level of granularity and divide it accordingly. To support this, C4CAM includes a partitioning transformation within the `cim` abstraction. This transformation can be likened to tiling in compiler terminology, with hardware-specific considerations. It enables the efficient partitioning of kernels to facilitate their execution on the device(s), but requires an abstraction to accumulate partial results. To this end, the `cim` dialect was extended with the `cim.merge_partial` operation. It considers both the type of operation for which partial results are generated and the direction in which these results are accumulated. The partitioned version of Figure 5c for a device of size 32x32 is shown in Figure 5d. Note that this partitioned code and the surrounding loop (`scf.for`) is still serial.

Our extension to the `cim` abstraction focuses on identifying operations that can be offloaded to the CAM accelerator and

```
...
%4 = cim.acquire : index
%5 = cim.execute(%4, %2) ({
  %11= cim.transpose %2: tensor<10x8192xf32>
    -> tensor<8192x10xf32>
  cim.yield %11 : tensor<8192x10xf32>
}) : (index, tensor<10x8192xf32>) -> tensor<8192x10xf32>
cim.release %4 : index
%6 = cim.acquire : index
%7 = cim.execute(%6, %0, %5) ({
  %11 = cim.matmul %0, %5 : tensor<10x8192xf32>,
    tensor<8192x10xf32> -> tensor<10x10xf32>
  cim.yield %11 : tensor<10x10xf32>
}) : (index, tensor<10x8192xf32>, tensor<8192x10xf32>) -> tensor
      ↪ <10x10xf32>
...
```

(a) cim IR

```
...
%4 = cim.acquire : index
%5:2 = cim.execute(%4, %2, %0, %3) ({
  %7 = cim.transpose %2 : tensor<10x8192xf32>
    -> tensor<8192x10xf32>
  %8 = cim.matmul %0, %7 : tensor<10x8192xf32>,
    tensor<8192x10xf32>
     -> tensor<10x10xf32>
  %values, %indices = cim.topk %8, %3 : tensor<10x10xf32>
    , i64 -> tensor<10x1xf32>, tensor<10x1xf32>
  cim.yield %values, %indices : tensor<10x1xf32>, tensor<10x1xf32>
}) : (index, tensor<10x8192xf32>, tensor<10x8192xf32>, i64)
  -> (tensor<10x1xf32>, tensor<10x1xf32>)
cim.release %4 : index
...
```

(b) cim IR after fusing execution blocks

```
...
%4 = cim.acquire : index
%5:2 = cim.execute(%4, %2, %0, %3) ({
  %values, %indices = cim.similarity dot %2,
    %0, %3 : tensor<10x8192xf32>, tensor<10x8192xf32>,
    i64 -> tensor<10x1xf32>, tensor<10x1xf32>
  cim.yield %values, %indices : tensor<10x1xf32>, tensor<10x1xf32>
}) : (index, tensor<10x8192xf32>, tensor<10x8192xf32>, i64)
  -> (tensor<10x1xf32>, tensor<10x1xf32>)
cim.release %4 : index
...
```

(c) cim IR with rewrites for CAM lowering

```
scf.for %arg1 = %c0 to %c8192 step %c32 {
  %extr_slice = tensor.extract_slice %2[0, %arg1] [10, 32]
    [1, 1] : tensor<10x8192xf32> to tensor<10x32xf32>
  %extr_slice_0 = tensor.extract_slice %0[0, %arg1] [10, 32]
    [1, 1] : tensor<10x8192xf32> to tensor<10x32xf32>
  %7 = cim.acquire : index
  %8:2 = cim.execute(%7, %extr_slice, %extr_slice_0, %3) ({
    %values, %indices = cim.similarity dot  %extr_slice,
      %extr_slice_0, %3 : tensor<10x32xf32>, tensor<10x32xf32>,
      i64 -> tensor<10x1xf32>, tensor<10x1xf32>
    cim.yield %values, %indices : tensor<10x1xf32>,
      tensor<10x1xf32>}) : (index, tensor<10x32xf32>,
      tensor<10x32xf32>, i64) -> (tensor<10x1xf32>,
      tensor<10x1xf32>)
  %9 = cim.merge_partial values  similarity dot  horizontal
    %7, %4, %8#0 : index, tensor<10x1xf32>, tensor<10x1xf32>
    -> tensor<10x1xf32>
  ...
  cim.release %7 : index
}
```

(d) cim IR after partitioning

Figure 5. cim IR of the HDC similarity function, after different analysis and optimization passes. Similar operations at different stages are highlighted using the same color.

on preparing the code via partitioning for further lowering to CAMs. It does not address the mapping of an input application or its partitions onto the CAM accelerator, nor does it incorporate any device-specific optimizations. Our novel cam abstraction takes on these responsibilities.

4.4.2 The cam abstraction. To convert the cim IR into the cam IR, C4CAM introduces the cim-to-cam conversion pass. This pass requires specifying the target CAM device type (e.g., ACAM, TCAM, or MCAM) in the architecture specification, which also determines the search type (EX, BE or TH) and the metric to be utilized during the conversion process. Additionally, it requires specifying the method for accumulating partial results. By default, this is handled by the CPU. Alternatively, a specialized method with HW support can be specified. This accounts for CAM arrays that include dedicated circuitry (e.g., an adder tree) or an extra CIM module to support typical accumulation functions. Depending on the merging circuitry and the mapping to the CAM arrays, the search and merge operations can be pipelined or executed sequentially. For example, in BioHD [54], search operations and the accumulation of partial Hamming Distance values are pipelined to improve throughput.

The cam dialect is responsible for mapping the high-level functions from the cim dialect to the CAM-device calls. After applying this conversion pass, occurrences of a sequence of cim.acquire, cim.execute, and cim.release working on the same device handle are substituted with allocation calls at bank, mat, and array-level. The allocated modules, i.e., banks, mats, arrays, and subarrays, then execute the search operations in parallel.

More concretely, the cam.alloc_bank function is used to allocate a CAM bank, taking the row and column sizes of the desired CAM size as parameters. Furthermore, allocating a mat from the bank, and a CAM array from the mat, and a subarray from an array is accomplished using the cam.alloc_mat, cam.alloc_array, and cam.alloc_subarray functions, respectively. Similarly, the cim.execute function is lowered into three CAM function calls: cam.write_value, cam.search, and cam.read_value. The write operation (line 16) programs the CAM arrays with the input data. The search operation (line 18) performs the actual search on the data based on the specified search type and metric. The supported search types include exact match, best match, and range match, while the available distance metrics are Euclidean and Hamming. The read operation (line 20) reads the values and indices of the search results from the device.

The original program underwent partitioning at the CIM dialect without considering the hierarchy. This approach was chosen because dealing with synchronization and accumulation of partial results across different levels of the hierarchy often requires hardware-specific information, which goes against the principles of the cim dialect. To map an application onto the CAM abstraction, the cam-map pass within the

```
%bank_values_buffer = memref.alloc() : memref<2x10x1xf32>
%bank_indices_buffer = memref.alloc() : memref<2x10x1xf32>
...
scf.parallel (%arg1) = (%c0) to (%c8192) step (%c4096) {
  %1 = cam.alloc_bank %c32, %c32 : index, index -> cam.bank_id
  ...
  scf.parallel (%arg2) = (%c0) to (%c4096) step (%c1024) {
    %4 = cam.alloc_mat %1 : cam.bank_id -> cam.mat_id
    ...
    scf.parallel (%arg3) = (%c0) to (%c1024) step (%c256) {
      %7 = cam.alloc_array %4 : cam.mat_id -> cam.array_id
      ...
      scf.parallel (%arg4) = (%c0) to (%c256) step (%c32) {
        ...
        %12 = cam.alloc_subarray %7 : cam.array_id -> cam.
            ↪ subarray_id
        cam.write_value %12, %subarray_data_buffer :
          cam.subarray_id, memref<10x32xf32>
        cam.search  exact  eucl %12, %subarray_query_buffer :
          cam.subarray_id, memref<1x32xf32>
        %13:2 = cam.read exact %12 : cam.subarray_id
          -> memref<10x1xf32>, memref<10x1xf32>
        ... }
      ...}
    ...}
  ...}
...
%value_res = cam.merge_partial bank values horizontal %1,
  %bank_values_buffer :cam.bank_id, memref<2x10x1xf32>
  -> memref<10x1xf32>
...
```

Figure 6. `cam` IR after mapping. Similar operations at `cim` and `cam` stages are highlighted using the same color.

`cam` dialect can be employed. This pass transforms the application into a nested loop structure according to the provided specifications, incorporating the required hardware calls at each loop level.

The code in Figure 6 shows the mapping of the code in Figure 5d to a system with two banks, four mats per bank, four arrays per bank, and eight subarrays per array. In this example we assume the method for merging partial results from [54], enabling parallel execution of operations. Consequently, the serial `scf.for` loop is substituted with a `scf.parallel` loop, and the `cim.merge_partial` operation is replaced with a corresponding `cam` operation. The required components of the CAM accelerator (bank, tile, array, subarray) and buffers for storing the results (values and indices) are allocated at each level of the loop. In cases where the system size precisely matches the data size, the levels of the nested loop, starting from the outermost level, iterate over the banks, mats within each bank, arrays within each bank, and subarrays within each array. However, if the data size exceeds the system's capacity, an additional loop is introduced. This loop includes iteration over banks, allowing the system to be called multiple times to process the data effectively.

The `cim-to-cam` conversion pass also performs bufferization of tensors involved in executing a kernel on the CAM. This process determines how the memory is handled between the host and the device. During the process of lowering from `cam` to `scf` and subsequently to `llvm`, the `cam` operations are mapped to function calls corresponding to the low-level API to access the CAM accelerator.

Built-in optimizations: C4CAM provides an extensible and flexible framework that enables future research in code optimizations and auto-tuning. Currently, the framework uses simple heuristics to optimize for different metrics, namely, for latency/performance, power consumption and device utilization. This is enabled by device-specific transformations that can be further composed by performance engineers. For example, in order to minimize latency, C4CAM prioritizes maximizing the utilization of parallel-executing arrays in the system. In contrast, C4CAM reduces the number of enabled subarrays at a time inside an array to minimize power consumption. For devices that support selective search and in cases where the standard data placement underutilizes an array due to the number of data entries being smaller than the number of rows in the memory, it is possible to place multiple batches of data on the same array. By utilizing selective search, different queries can be searched on corresponding rows of the same array in multiple cycles.

As demonstrated in Section **5.6.1**, employing the same hierarchy specification (mat, array, and subarray sizes) consistently leads to longer latencies. However, the impact on latency varies depending on the dimensions of the subarrays, which subsequently alters the number of banks. These design choices significantly impact energy consumption and are challenging to predict without an integrated framework that facilitates such optimizations and is supported by simulation. Thus, C4CAM enables quickly identifying the configurations that best meet workload requirements, while considering scalability and favorable compromise between latency and energy consumption.

5 Evaluation

This section presents our experimental setup and gives a detailed analysis of the code generated with C4CAM.

5.1 Experimental setup

5.1.1 System setup and technology parameters. For the CAM technology parameters, we consider the 2FeFET CAM design proposed in [51] at the 45 nm technology node. Energy and latency numbers for TCAM and MCAM operations were extracted from Eva-CAM [36], which is backed by experimental demonstrations of manufactured FeFET CAMs. Since we are varying the array size for design space exploration, the search latency can vary from 860 ps to 7.5 ns for array sizes of 16×16 and 256×256, respectively. For the GPU results, we use the NVIDIA RTX 3090 GPU (8 nm process) with a base clock speed of 1395 MHz. The power consumption is measured using the NVIDIA System Management Interface `nvidia-smi`, and energy is derived thereof.

5.1.2 Simulation infrastructure. For evaluation, we use the open-source CAM simulator CAMASim [33] and extend it with an interface to connect it to C4CAM. Table 1 lists our configuration parameters for all experiments in this section.

Table 1. Simulator configuration

Architecture & Circuit configuration			
Type	**HDC**	**KNN**	**DNA**
Horizontal merge	Voting	Voting	Counter
Vertical merge	Comparator	Comparator	Gather
Cell	TCAM	T/MCAM	TCAM
Sensing circuit	BE	BE	TH

Cost of additional circuits		
Type	**Latency**	**Energy**
Adder	0.25 ns	1.3 fJ/bit
Register	0.5 ns	4.5 fJ/bit
Comparator	0.25 ns	0.4 fJ/bit
Decoder/Encoder	0.25 ns	29 fJ

The simulator consists of two main modules: the *Functional Simulator* and the *Performance Evaluator*. The functional simulator module performs quantization and mapping to the subarrays of the simulated CAM. It also executes the search operation on each subarray and merges the partial results. The performance evaluator estimates device specifics, including the size of peripheral circuits at each hierarchy level, predicts the circuit types and sizes depending on the merge scheme for estimating merging costs and adjusts buffer size accordingly. The peripheral estimator manages the latency of merge operations generated at the CAM abstraction through the compiler flow. At the subarray level, CAMASim integrates external circuit-level CAM modeling tools like EvaCAM [36] or SPICE simulation results to generate performance values, ensuring compatibility with various CAM cell designs. To handle large data dimensions and entry sizes, the extended simulator allows for fine-grain control of the hierarchy, and models CAM queries to obtain energy and latency based on real hardware behavior.

5.2 Evaluated applications

In a real-world scenario, the data transfer for chosen benchmarks on an accelerator, like a GPU or CAM, would be amortized in the long run. For that, unless specified, reported latency and energy exclude preprocessing (e.g., hashing and quantization) and transfer of training data or reference genome, focusing only on the query processing for all applications.

5.2.1 K-nearest neighbors. KNN is a popular classification, regression, and anomaly detection algorithm. It identifies the K closest training examples in the feature space to a given test sample. It is especially interesting because of its versatility and explainability, with no training required. KNNs are both memory and computationally-intensive, making their scalability and performance strongly limited on conventional systems. Two versions of KNN were implemented, utilizing cosine similarity with `matmul` and Euclidean distance with `norm`, both followed by a `topk` operation. For a direct comparison with [24] in Section 5.3, we classify the top 4 most popular datasets: Iris, Wine, Breast Cancer, and Wine Quality [1], using an 80/20 training/test split on them. We also evaluated KNN on chest X-Ray images from the Pneumonia dataset [2] in Section 5.6, Table 2.

5.2.2 Hyperdimensional Computing. HDC is a framework inspired by the human brain's ability to process information. It utilizes high-dimensional vectors known as hypervectors as a fundamental building block. Hypervectors are large binary vectors with thousands of dimensions. We evaluated binary and multi-bit HDC models with 8k dimensions on the MNIST dataset following the approach outlined in [52]. The TorchScript implementation of the main kernel employs the same operations shown in Figure 4a.

5.2.3 DNA read mapping. Read mapping is a fundamental step in genome analysis, aiming to identify the origin of each read R from an input genome in a reference genome G. This task is computationally challenging due to the very large genome sizes and permissible minor differences in the form of insertions, deletions, and substitutions. One algorithm used for this purpose is known as the fast seed-and-vote algorithm (FVSA) [35]. This approach performs a read-by-read comparison with the reference genome and counts matching seeds (smaller substrings in a read). The block with the highest vote count in the reference genome is selected as the most likely location of a read R in the reference genome G. Unlike KNN and HDC, which is based on one-shot *similarity search*, read mapping finds high-similarity sections based on the Hamming distance of overlap positions of reads against G. Hamming distance is often used in prior CAM-based designs for read mapping [22, 27, 29].

We implemented the FVSA algorithm in TorchScript, drawing inspiration from SaVI [29] (seed-and-vote-based in-memory accelerator), where the seed-reference lookup and voting steps are solely performed by TCAM subarrays and shift-register counters, respectively. We encode base pairs using 4-bit encoding as in [29], which allows for representing low-quality bases (N), transversions, and transitions. Figure 7 illustrates the seed-reference lookup based on the Hamming distance between the reference genome and seeds of a read sequence implemented at the `cim` abstraction. The Hamming distance of two words, A and B, is calculated as A `xor` B (line 11) followed by a population count (line 14). The next step in FVSA involves counting votes based on the location of matched seeds in the reference genome through a threshold operation. This threshold accounts for partial matches, allowing for substitutions in the genome sequence. The `below_threshold` operation is achieved through a `cim.cmp` (compare) operation with `lt` (less than) argument (line 16), followed by a `nonzero` operation (line 18) to return the indices of the matched rows. We evaluated the FVSA use-case using the publicly available real human genome GRCh38 build (3 GB) [38] and the read sequence NA12878 (59 GB). This allowed us to stress-test our compilation framework and mapping algorithm for CAM arrays. We evaluated the accuracy, throughput and energy

```
...
scf.for %arg1 = %c0 to %read_count step %c1 {
  ...
  // 1. Seed generation
  %seeds = func.call @seed_generation(%read, %cst16) :
    (tensor<70xi8>, i64) -> tensor<1x110x16xi8>
  ...
  // 2. Seed-reference lookup
  %0 = cim.acquire : index
  %1 = cim.execute(%0, %ref_genome, %seeds, %cst3) ({
    %2 = cim.xor %ref_genome, %seeds :
      tensor<8192x1x16xi8>, tensor<1x110x16xi8> ->
      tensor<8192x110x16xi8>
    %3 = cim.popcount %2 : tensor<8192x110x16xi8>,
      -> tensor<8192x110xi64>
    %4 = cim.cmp lt %3, %cst3 : tensor<8192x110xi64>,
      i64 -> tensor<8192x110xi1>
    %5 = cim.nonzero %4 : tensor<8192x110xi1>,
      -> tensor<?x?xi64>
    cim.yield %5: tensor<?x?xi64>
  }) : (index, tensor<8192x1x16xi8>,
    tensor<1x110x16xi8>, i64) -> (tensor<?x?xi64>)
  // 3. Voting
  %6 = func.call @voting (%1) : (tensor<?x?xi64>)
    -> tensor<8192xi64>
  ...} ...
```

Figure 7. Seed-reference lookup in FVSA

efficiency across three array sizes and compared them to BWA-MEM implementation on GPUs [42], both with read sequences consisting of 100-bp reads.

5.3 Validation

To validate the C4CAM framework, we use the CAM-design and hand-optimized designs that for the HDC use case [23] and the DNA read mapping use case [29].

We use the manual designs from [23] as baseline for the HDC use case. We generate code for binary and multi-bit implementations of HDC for different CAM architectures, i.e., with array sizes of $32 \times C$ where C is varied to 16, 32, 64, and 128. The validation results for accuracy, latency and energy are shown in Figure 8a, Figure 8b, and Figure 8c respectively. For a fair comparison, we use the same system configuration as in the baseline, i.e., four mats per bank, four arrays per mat, eight sub-arrays per array, and as many banks as needed to store the whole dataset.

In this experiment, the observed deviation in the latency and the energy consumption is, on average (geomean), 0.9% and 5.5%, respectively (notice that the y-axes do not start at 0 for better visualization). These small deviations can be attributed to slight differences in the versions of the simulation environment rather than to fundamental differences in the implementations. Except for the 32-column design in the 1-bit implementation and the 128-column design in the 2-bit implementation, where the generated code shows a 1% higher accuracy, the accuracy is equal to the baseline. Hence, C4CAM effectively matches the quality of hand-tuned implementations by expert CAM designers.

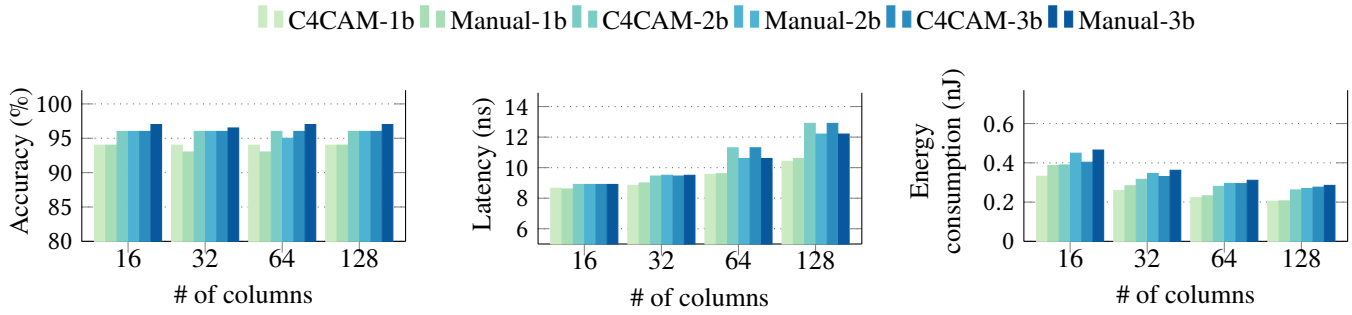
To understand the latency results in Figure 8b, it is important to note that all search operations happen in parallel, and the match line (ML) discharges more slowly for larger columns. As for the energy numbers shown in Figure 8c, larger C leads to lower energy consumption because fewer peripherals and fewer levels (arrays, mats, and banks) are required as C increases. Moreover, as observed in [23], we corroborate that 1-bit implementations are more energy efficient than multi-bit ones. This improvement is associated with the higher ML and data line voltages of the multi-bit implementations.

Figure 9 shows the versatility of C4CAM to support a fundamentally different and more complex algorithm, namely the FVSA inspired by [29]. The figure shows the accuracy, performance (as throughput in millions of reads per second) and energy efficiency (throughput per Watt) for both manual design and C4CAM-generated code, applied to 100-bp reads on the human genome [38] across three array sizes. These units are commonly used to compare the efficiency of accelerators due to the streaming nature of the read-mapping execution. Compared to the manual design, the C4CAM-generated codes achieve the same baseline accuracy for all setups. Regarding performance and energy estimation, our results vary by 7.8% and 6% on average for throughput and throughput per Watt, respectively. This variation might be attributed to minor architectural differences not presented in [29]. This example illustrates C4CAM's capability to handle the parallel operation of a substantial array workload totaling 1.5 GB of the encoded human reference genome. Specifically, this dataset requires from 805K to 12.8M subarrays in the 128×128 and 32×32 setups, respectively. Despite also managing a significant volume of encoded queries (29 GB of reads), the compiler does not significantly impede the processing speed per read. This is because the bottleneck of the FVSA lies in the seed-reference lookup, and the problem size is determined by the size of the reference genome.

5.4 Impact of CAM cell precision

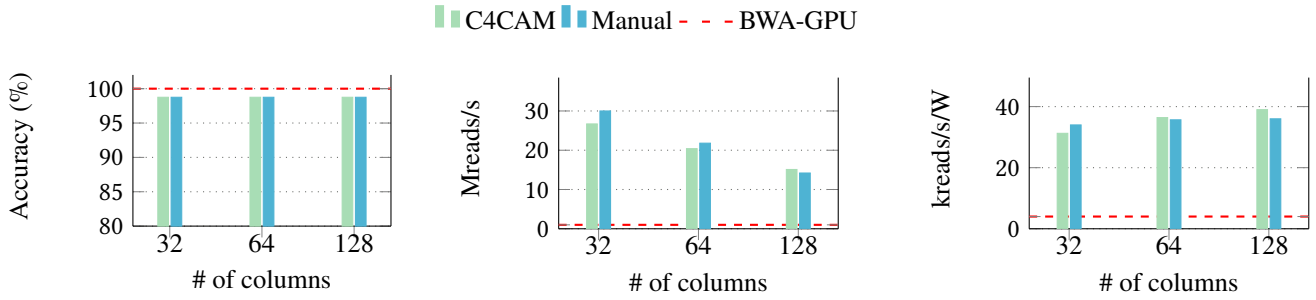
Due to limited CAM cell precision, the data entries and queries of applications must be encoded, quantized or hashed to be suitable for CAM search. The results for the HDC execution, as illustrated in Figure 8a, show that more bits per cell consistently improve accuracy. On the other hand, as depicted in Figure 8b, 2-bit and 3-bit CAMs have similar search latency, which is, on average, 12% higher than the 1-bit setting across various column counts. This latency increase is attributed to the greater complexity of the sensing circuit [23], and its impact is more pronounced in energy consumption. Across different column sizes, employing the 3-bit configuration, on average, results in a 4.5% and 29.1% increase in energy consumption in comparison to 2-bit and 1-bit configurations.

Increasing the number of bits per cell per feature would intuitively improve the accuracy of other applications that use the same distance metric. In KNN, the real-valued features of the query and memory entries can be simply quantized to match the same bit precision as the multi-bit CAM. Figures 10a, 10b, and 10c report the accuracy, latency, and energy



(a) Accuracy **(b)** Latency **(c)** Energy consumption

Figure 8. Validation results per query against manual designs [23] for HDC



(a) Accuracy over BWA-MEM **(b)** Throughput **(c)** Throughput/Watt

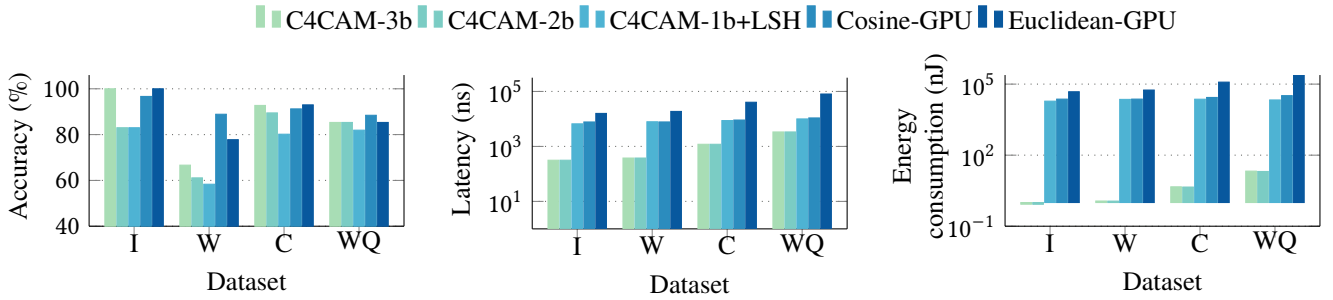
Figure 9. Comparison of accuracy, throughput and throughput per Watt against manual designs [29] and GPU implementation [42] for read mapping

consumption of CAM designs with various bit precision settings and different GPU implementations executing KNN on different datasets. A similar pattern to HDC is observed in KNN, where a 3-bit configuration yields, on average, 6.4% and 10.3% more accurate results compared to 2-bit and 1-bit designs, respectively. Similarly, the 3-bit configuration also incurs an average of 2% increase in the energy compared to the 2-bit configuration. However, with binary CAMs, real-valued query and memory entries must undergo a transformation using a Locality-sensitive Hashing (LSH) algorithm [7]. In this implementation, a CAM array stores the LSH signatures of the memory entries and computes the Hamming distance of the LSH signature of the query. The hashing step for queries is done using the GPU and results in a 7.3× and $\sim 12,000\times$ increase, on average, in latency and energy consumption, respectively, compared to the 3-bit configuration. Although the binary CAM+LSH approach can be more energy and time-efficient than the GPU-only approach, especially in handling larger and high-dimensional datasets, LSH sacrifices on accuracy.

5.5 Comparison to GPU

Due to their parallel processing capabilities, optimized architecture for matrix operations and high memory bandwidth and throughput, GPUs are often used in machine learning and similarity search domains. To compare CAM-based systems to GPUs, we also include results from CUDA implementations of cosine similarity and Euclidean norm in Figure 10. While it is known that cosine similarity performs better with higher dimensions than Euclidean distance, we demonstrate that 3-bit CAMs outperform GPU-based cosine similarity, operating $\sim 14\times$ faster with $\sim 13,700\times$ less energy consumption. The datasets used in Figure 10 have low dimensionality which leads to resource under-utilization in the GPU. The larger Pneumonia dataset provides a better perspective since these experiments can exhaust the computational power of the GPU. The results for the 3-bit CAM-based system on this large KNN classification are shown in Table 2 (under *cam-base*). As can be seen, the power consumption of the CAM-based implementation is considerably lower than that of the GPU (around 200 W).

We run the HDC implementation from [52] on the GPU, which achieves a 95.4% accuracy rate. On average, it takes 5× longer and consumes $14,900\times$ more energy compared to the implementations generated by C4CAM. Corroborating with prior works [19, 23, 24], CAM systems can outperform GPU energy by orders of magnitude while still being significantly faster. Note that the experiments on GPU and CAMs measure



(a) Accuracy **(b)** Execution time **(c)** Energy Consumption

Figure 10. Accuracy, latency, and energy consumption comparison of KNN implementations on 128×128 CAM arrays contrasting with GPU execution. I, W, C, and WQ represent Iris, Wine, Cancer, and Wine Quality datasets, respectively

Table 2. EDP and power analysis for KNN execution on the Pneumonia dataset

	EDP (nJ·s)					**POWER (W)**				
subarray size	16x16	32x32	64x64	128x128	256x256	16x16	32x32	64x64	128x128	256x256
cam-base	0.75	0.30	0.15	0.08	0.05	44.14	16.30	5.97	2.34	0.86
cam-power	1.32	0.61	0.44	0.29	0.23	25.23	8.15	2.10	0.66	0.19

solely the similarity search kernel. This kernel is by far the most time-consuming part in KNN and HDC. By Amdahl's law, the end-to-end performance can be improved by the fraction of the time that similarity search is actually used. For example, Euclidean distance dominates the execution time of standard KNN and takes 52 – 96% of the time in k-means algorithms [49]. In standard KNN, for instance, Euclidean distance dominates the execution time, while in k-means algorithms, it constitutes 52 – 96% of the total time [49]. In HDC, the associative search accounts for 51 – 85% of the GPU execution [23]. In Memory Augmented Neural Networks, similarity kernels contribute approximately 50 – 80% of the time, depending on the platform [48].

For the DNA read mapping use case, we compare the implementation of FVSA on TCAMs to the BWA-MEM implementation on GPUs from [42]. BWA-MEM remains the most efficient algorithm for GPUs and thus represents a strong baseline. It is important to note that while BWA-MEM and FVSA are distinct algorithms for the same problem, BWA-MEM solves it with slightly improved accuracy, as illustrated in Figure 9a. For aligning 100-bp reads, the GPU baseline achieves 1 million reads per second (Mreads/s) with an energy efficiency of 4 thousand reads per second per Watt (kreads/S/W), as depicted in Figure 9. This implies that CAM-based systems can potentially deliver up to 26× higher mapping throughput than a highly-optimized GPU implementation while consuming ∼ 9× less energy.

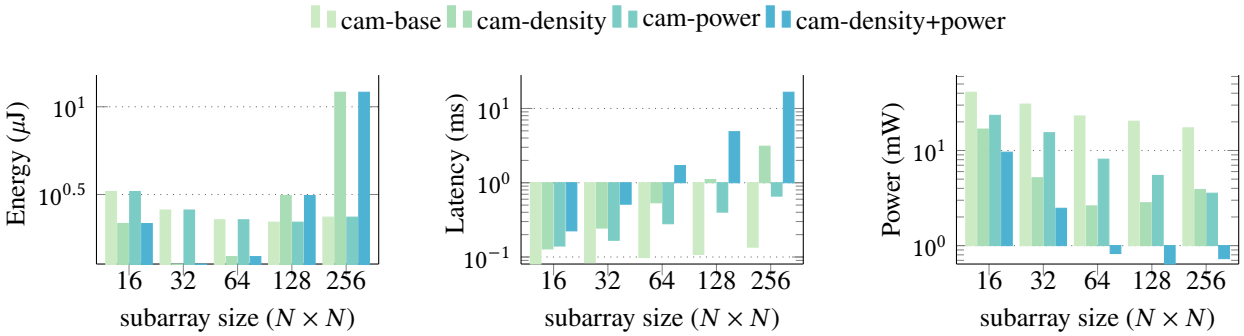
5.6 Design space exploration

Since C4CAM is retargetable, it can be used to quickly generate implementations for different system parameters and optimization targets. In this section, we explore the design-space exploration capabilities for hardware/software designs.

5.6.1 Fixed architectural parameters. As discussed above, C4CAM can reproduce the results of single manual designs. To demonstrate its retargetability, we evaluate systems consisting of sub-arrays with sizes of $R \times C$, where $C = R$ assuming values of 16, 32, 64, 128, and 256 with different configurations for the same, as outlined in Section **5.1.1**, namely:

- *cam-base*: In this configuration, applications are allocated to the CAM accelerator without incorporating the optimizations discussed in Section **4.4.2**. In this setup, parallel execution is enabled at each level.
- *cam-power*: This configuration implements a restriction on the maximum number of sub-arrays activated concurrently. Specifically, for each application, we have chosen to enable only one sub-array per array at a time.
- *cam-density*: This configuration demonstrates the impact of employing selective search [55] to enhance both the utilization of arrays and the system's overall capacity, as shown in Table 3.
- *cam-power+density*: This configuration imposes limitations on the number of enabled sub-arrays at a time. Simultaneously, it incorporates selective search technique to enhance the system's capacity.

For all sub-array sizes, the configuration remains consistent, with 4 mats per bank, 4 arrays per mat, and 8 sub-arrays per array. We use as many banks as needed to accommodate the input data. Figure 11a and Figure 11b illustrate the energy consumption and latency of the configurations mentioned above respectively when executing the HDC application on the MNIST dataset with 8k dimensions.



(a) Energy **(b)** Latency **(c)** Power

Figure 11. Impact of subarray size and C4CAM optimizations on latency, energy, and power

In the *cam-power* configuration, only one sub-array within the array is active at a time. With a sub-array of size 16×16, the power consumption is reduced to approximately 0.57× with respect to the base configuration (Figure 11c). Similarly, the power requirement for the largest array size is merely 20% of the base configuration. However, this reduction in power consumption results in increased latency. For instance, executing the application on a 32×32-sized subarray incurs a latency increase of approximately 2× compared to the baseline. As the array size increases, the latency rises, reaching up to 4.86× the baseline for the largest sub-array size. The overall energy consumption remains the same between the two configurations, *cam-power* and *cam-base*.

The analysis of the KNN benchmark is similar to the analysis for HDC. We summarize the results in Table 2 for EDP and power. The absolute values of energy and latency are considerably higher than in the HDC case. This is simply due to the sheer size of the Pneumonia dataset, requiring many banks in the CAM accelerator.

The *cam-density* configuration uses selective search to improve resource utilization, as shown in Table 3. In the case of the smallest array size (16×16), the execution time is less than twice compared to the base configuration. This trend scales further, and with the largest subarray size (256×256), the execution time is nearly 23× longer compared to the *cam-base* configuration. The energy consumption for subarray sizes ranging from 16×16 to 64×64 in the cam-density configuration is, on average, 0.6× that of the corresponding sub-array size in the baseline configuration. However, for sub-arrays of 128×128 or 256×256, the energy consumption increases compared to the baseline, reaching 1.4× and 5.1×, respectively. It is worth noting that by fixing the system configuration and enabling selective search, the number of banks required for application execution is reduced, thus reducing the overall power consumption.

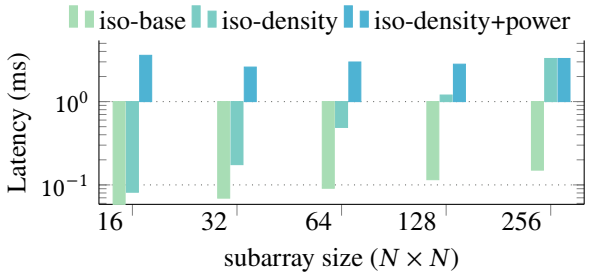
The *cam-power-density* configuration combines the approaches of both *cam-power* and *cam-density* to achieve the most significant reduction in power consumption. A 16×16-sized subarray utilizes only 23.4% of the base power, while the largest sub-array requires only 4.2% of the base power. However, this reduction in power consumption comes at the cost of significantly increasing the execution time. In the case of the largest subarray configuration, the execution time is approximately 121× longer compared to the base configuration.

Table 3. Number of subarrays used to implement HDC

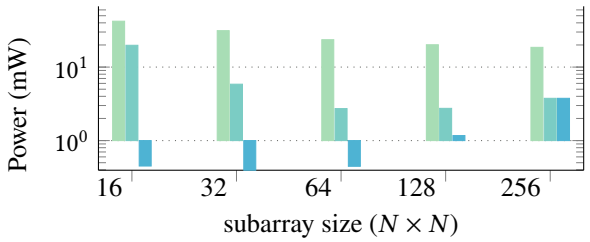
	16×16	32×32	64×64	128×128	256×256
cam-base	512	256	128	64	32
cam-density	512	86	22	6	2

5.6.2 Iso-capacity analysis. With the iso-capacity experiments, we investigate the relationship between energy consumption and latency by changing the size of subarrays and the number of subarrays per array while keeping the capacity fixed to 2^{16} TCAM cells per array. To achieve this, we modify the subarray size, starting from 256×256 which corresponds to one subarray per array, and gradually decrease it to 16×16, resulting in 256 subarrays per array. The numbers of arrays per mat and mats per bank are fixed as in the previous sections. It is important to note that these systems are not iso-area since each subarray has its own set of peripherals. This means that as the size of the subarrays is reduced, more peripherals are needed, and chip area increases.

Figure 12b shows that the energy consumption in *iso-base* remains nearly constant across subarray sizes. Moreover, *cam-density* and *cam-power+density*, on average, achieve 1.75× energy improvement over *iso-capacity-base*, except for large subarray sizes like 128×128 and 256×25. The total execution time across different subarray sizes also varies within a moderate range, i.e., from 58μs for 16×16 to 150μs for 256×256 , as shown in Figure 12b. Again, as the search latency increases for larger columns, the execution time also increases despite the consistent number of cells within an array. As for the *cam-density* and *cam-power+density* transformations,



(a) Latency



(b) Power

Figure 12. Impact of optimizations on iso-capacity setups

Figure 12b shows a significant decrease in power consumption, offering a potential CAM configuration that can be used in power-constrained system setups.

6 Conclusions

We present C4CAM, the first framework for programming and exploring trade-offs in CAM-based accelerators. We introduce a retargetable MLIR-based code transformation flow from high-level TorchScript code, featuring a novel `cam` abstraction that is specifically tailored for CAM-based accelerators. This abstraction provides control knobs that allow for the tuning of various metrics by adjusting the mapping of applications to the CAM arrays. To validate the effectiveness of C4CAM, we compare our results with those obtained from a hand-crafted designs and demonstrate that C4CAM produces comparable results. Moreover, we demonstrate C4CAM capabilities by automatically generating implementations optimized for performance, power and device utilization. Finally, we show how C4CAM retargetability facilitates design space exploration by varying architectural parameters without any application recoding effort. The architecture specification supported by C4CAM, along with its compilation flow, also enables the specification of heterogeneous systems. However, determining the optimal mapping strategy for heterogeneous systems based on different optimization targets remains a subject for future research.

Acknowledgments

This work was partially funded by the Center for Advancing Electronics Dresden (cfaed) and the German Research Council (DFG) through the HetCIM project (502388442) under the Priority Program on 'Disruptive Memory Technologies' (SPP 2377), the AI competence center ScaDS.AI Dresden/Leipzig in Germany (01IS18026A-D), the Semiconductor Research Corporation (SRC), the Logic and Memory Devices Program (LMD), and the AI Chip Center for Emerging Smart Systems (ACCESS) sponsored by InnoHK funding, Hong Kong SAR.

References

[1] Kaggle datasets, howpublished = https://www.kaggle.com/datasets, note = Accessed: 2023-11-20.

[2] Pneumonia dataset, howpublished = https://dzl.de/en/core-datasets/pneumonia/, note = Accessed: 2023-11-20.

[3] Intermediate representation execution environment. https://github.com/iree-org/iree/, 2021. Accessed: 2022-08-30.

[4] Onnx-mlir. https://github.com/onnx/onnx-mlir, 2024. Accessed: 2024-03-01.

[5] Ali Ahmed, Kyungbae Park, and Sanghyeon Baeg. Resource-efficient sram-based ternary content addressable memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(4):1583–1587, 2016.

[6] Mustafa Ali, Amogh Agrawal, and Kaushik Roy. Ramann: in-sram differentiable memory computations for memory-augmented neural networks. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 61–66, 2020.

[7] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122, 2008.

[8] Hamza Errahmouni Barkam et al. Hdgim: Hyperdimensional genome sequence matching on unreliable highly scaled fefetyperdimensional. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023.

[9] Lorenzo Chelini et al. Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 15–26, 2021.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018.

[11] João Paulo C. de Lima, Asif Ali Khan, Luigi Carro, and Jeronimo Castrillon. Full-stack optimization for cam-only dnn inference. In *2024 Design, Automation and Test in Europe Conference (DATE)*, DATE'24, pages 1–6. IEEE, March 2024.

[12] Paul Dlugosch et al. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[13] others. The torch-mlir project. https://github.com/llvm/torch-mlir, 2023. Accessed: 2023-11-20.

[14] Catherine E Graves et al. In-memory computing with memristor content addressable memories for pattern matching. *Advanced Materials*, 32(37):2003437, 2020.

[15] Robert Hanhan et al. Edam: edit distance tolerant approximate matching content addressable memory. In *49th Annual International Symposium on Computer Architecture*, pages 495–507, 2022.

[16] Xiaobo Sharon Hu et al. In-memory computing with associative memories: a cross-layer perspective. In *2021 IEEE International Electron Devices Meeting (IEDM)*, pages 25.2.1–25.2.4. IEEE, 2021.

[17] Li-Yue Huang, Meng-Fan Chang, Ching-Hao Chuang, Chia-Chen Kuo, Chien-Fu Chen, Geng-Hau Yang, Hsiang-Jen Tsai, Tien-Fu Chen, Shyh-Shyuan Sheu, Keng-Li Su, et al. Reram-based 4t2r nonvolatile tcam with 7x nvm-stress reduction, and 4x improvement in speed-wordlength-capacity for normally-off instant-on filter-based search engines used in big-data processing. In *2014 Symposium on VLSI Circuits Digest of Technical Papers*, pages 1–2. IEEE, 2014.

[18] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 703–718, New York, NY, USA, 2022.

[19] Mohsen Imani, Yeseong Kim, and Tajana Rosing. Nngine: Ultra-efficient nearest neighbor accelerator based on in-memory computing. In *2017 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8. IEEE, 2017.

[20] Mohsen Imani et al. Searchd: A memory-centric hyperdimensional computing with stochastic training. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2422–2433, 2019.

[21] Hai Jin, Bo Lei, Haikun Liu, Xiaofei Liao, Zhuohui Duan, Chencheng Ye, and Yu Zhang. A compilation tool for computation offloading in reram-based cim architectures. *ACM Transactions Archit. Code Optim.*, 20(4), oct 2023.

[22] Roman Kaplan, Leonid Yavits, and Ran Ginosar. Rassa: resistive prealignment accelerator for approximate dna long read mapping. *IEEE Micro*, 39(4):44–54, 2018.

[23] Arman Kazemi et al. Achieving software-equivalent accuracy for hyperdimensional computing with ferroelectric-based in-memory computing. *Scientific reports*, 12(1):19201, 2022.

[24] Arman Kazemi, Mohammad Mehdi Sharifi, Ann Franchesca Laguna, Franz Müller, Ramin Rajaei, Ricardo Olivo, Thomas Kämpfe, Michael Niemier, and X Sharon Hu. In-memory nearest neighbor search with fefet multi-bit content-addressable memories. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1084–1089. IEEE, 2021.

[25] Asif Ali Khan, João Paulo C De Lima, Hamid Farzaneh, and Jeronimo Castrillon. The landscape of compute-near-memory and compute-in-memory: A research and commercial overview. *arXiv preprint arXiv:2401.14428*, 2024.

[26] Asif Ali Khan, Hamid Farzaneh, Karl F. A. Friebel, Clément Fournier, Lorenzo Chelini, and Jeronimo Castrillon. Cinm (cinnamon): A compilation infrastructure for heterogeneous compute in-memory and compute near-memory paradigms. *arXiv preprint arXiv:2301.07486*, 2023.

[27] S Karen Khatamifard, Zamshed Chowdhury, Nakul Pande, Meisam Razaviyayn, Chris Kim, and Ulya R Karpuzcu. Genvom: Read mapping near non-volatile memory. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 19(6):3482–3496, 2021.

[28] Ann Franchesca Laguna et al. Ferroelectric fet based in-memory computing for few-shot learning. In *2019 on Great Lakes Symposium on VLSI*, pages 373–378, 2019.

[29] Ann Franchesca Laguna, Hasindu Gamaarachchi, Xunzhao Yin, Michael Niemier, Sri Parameswaran, and X Sharon Hu. Seed-and-vote based in-memory accelerator for dna read mapping. In *39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[30] Chris Lattner et al. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[31] Mengyuan Li et al. Associative memory based experience replay for deep reinforcement learning. In *41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[32] Mengyuan Li et al. imars: an in-memory-computing architecture for recommendation systems. In *59th ACM/IEEE Design Automation Conference*, pages 463–468, 2022.

[33] Mengyuan Li, Shiyi Liu, Mohammad Mehdi Sharifi, and X. Sharon Hu. Camasim: A comprehensive simulation framework for content-addressable memory based accelerators, 2024.

[34] Liu Liu et al. A reconfigurable fefet content addressable memory for multi-state hamming distance. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2023.

[35] Song Liu, Yi Wang, and Fei Wang. A fast read alignment method based on seed-and-vote for next generation sequencing. *BMC bioinformatics*, 17:193–203, 2016.

[36] Liu Liu et al. Eva-cam: a circuit/architecture-level evaluation tool for general content addressable memories. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1173–1176. IEEE, 2022.

[37] Siri Narla et al. Modeling and design for magnetoelectric ternary content addressable memory (tcam). *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 8(1):44–52, 2022.

[38] National Center for Biotechnology Information. Genome Reference Consortium Human Data. https://www.ncbi.nlm.nih.gov/grc/human/data, 2023. Accessed on: 28/11/2023.

[39] Kai Ni et al. Ferroelectric ternary content-addressable memory for one-shot learning. *Nature Electronics*, 2(11):521–529, 2019.

[40] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *IEEE journal of solid-state circuits*, 41(3):712–727, 2006.

[41] Giacomo Pedretti et al. X-time: An in-memory engine for accelerating machine learning on tabular data with cams. *arXiv preprint arXiv:2304.01285*, 2023.

[42] Minh Pham, Yicheng Tu, and Xiaoyi Lv. Accelerating bwa-mem read mapping on gpus. In *37th International Conference on Supercomputing*, pages 155–166, 2023.

[43] Songyun Qu, Shixin Zhao, Bing Li, Yintao He, Xuyi Cai, Lei Zhang, and Ying Wang. Cim-mlc: A multi-level compilation stack for computing-in-memory accelerators. *arXiv preprint arXiv:2401.12428*, 2024.

[44] Mariam Rakka et al. Dt2cam: A decision tree to content addressable memory framework. *IEEE Transactions on Emerging Topics in Computing*, 2023.

[45] Indranil Roy and Srinivas Aluru. Discovering motifs in biological sequences using the micron automata processor. *IEEE/ACM Transactions on computational biology and bioinformatics*, 13(1):99–111, 2015.

[46] Adam Siemieniuk et al. Occ: An automated end-to-end machine learning optimizing compiler for computing-in-memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[47] Mathias Soeken et al. An mig-based compiler for programmable logic-in-memory architectures. In *53rd Annual Design Automation Conference*, pages 1–6, 2016.

[48] Jacob R Stevens, Ashish Ranjan, Dipankar Das, Bharat Kaul, and Anand Raghunathan. Manna: An accelerator for memory-augmented neural networks. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 794–806, 2019.

[49] Fang Wang et al. Speeding up querying and mining operations on data with modern hardware. 2022.

[50] Xunzhao Yin, , et al. Deep random forest with ferroelectric analog content addressable memory. *arXiv preprint arXiv:2110.02495*, 2021.

[51] Xunzhao Yin et al. Fecam: A universal compact digital and analog content addressable memory using ferroelectric. *IEEE Transactions on Electron Devices*, 67(7):2785–2792, 2020.

[52] Tao Yu, Yichi Zhang, Zhiru Zhang, and Christopher M De Sa. Understanding hyperdimensional computing for parallel single-pass learning. *Advances in Neural Info. Processing Sys.*, 35:1157–1169, 2022.

[53] Xiaodong Yu et al. Robotomata: A framework for approximate pattern matching of big data on an automata processor. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 283–292, 2017.

[54] Zhuowen Zou, Hanning Chen, Prathyush Poduval, Yeseong Kim, Mahdi Imani, Elaheh Sadredini, Rosario Cammarota, and Mohsen Imani. Biohd: an efficient genome sequence search platform using hyperdimensional memorization. In *49th Annual International Symposium on Computer Architecture*, ISCA '22, page 656–669, New York, NY, USA, 2022. Association for Computing Machinery.

[55] Charles A Zukowski and Shao-Yi Wang. Use of selective precharge for low-power content-addressable memories. In *1997 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 1788–1791. IEEE, 1997.