# Dedicated instruction set for pattern-based data transfers: an experimental validation on systems containing In-Memory Computing units

Kevin Mambu, Henri-Pierre Charles, Maha Kooli

# Dedicated Instruction Set for Pattern-based Data Transfers: an Experimental Validation on Systems Containing In-Memory Computing Units

Kévin Mambu, Henri-Pierre Charles, Maha Kooli
*Univ. Grenoble Alpes, CEA, LIST*
F-38000 Grenoble, France
firstname.name@cea.fr

***Abstract*—In-Memory Computing (IMC) aims at solving the performance gap between CPU and memories introduced by the memory wall. However, general-purpose IMC does not consider the optimization of data transfers for patterns such as stencils and convolutions. This paper proposes a new Instruction Set Architecture (ISA) and a novel pattern encoding for IMC to transfer and organize data streams in order to perform efficiently computation. This instruction set is implemented on the Data-locality Management Unit (DMU) as a subset of the Computational SRAM (C-SRAM) Instruction Set Architecture. A programming model to interact with the DMU at language-level is also presented in this paper. This DMU ISA is evaluated on six applications run on three different system nodes. These system nodes are based on existing RISC-V cores and range from embedded to high-performance computing domain. Experiments show on average a speed-up of $\times 8.81$, an energy reduction factor of $\times 6.81$ and an improvement of the number of operations per cycle of $\times 4.59$, for The C-SRAM architecture integrating the proposed ISA of the DMU compared to a reference implementation on embedded systems. Results also show an improvement of the number of operations per cycle of $\times 2.99$ compared to a reference implementation on all system nodes.**

***Index Terms*—Instruction Set Architecture; In-Memory Computing; Stencil; Convolution; Pattern; Programming Model; Non-Von Neumann; Performance Analysis**

## I. Introduction

Architectures based on the von Neumann model present performance limitations due to the main memory. This performance bottleneck manifests itself through two phenomena. The first is the *Memory Wall*, the performance gap in terms of latency between the Processing Elements (PEs) and the memories of a given architecture. The second is the *Energy wall*, the energy gap between the arithmetical operation of PEs and the data transfers required to perform it. The case studied in [1] shows that this gap can reach $\times 100$ for L1 cache memories and up to $\times 10000$ for DRAM memory. These bottleneck phenomena become critical for highly data-dependent applications such as Computer Vision (CV), Image Processing (IP) and numerical analysis. A paradigm shift in architecture is then necessary for better computation efficiency [2]. One of the proposed alternatives is In-Memory Computing (IMC), a solution to implement non-von Neumann architectures and mitigate the Memory Wall by integrating computation in memory units. By their design, bandwidth usage at destination for IMC units is reduced for an overall energy reduction. However, efficient implementation of data processing solutions on IMC requires input data structures to be transferred and organized with alignment constraints for correct computation. Classical memory operations in general-purpose computing architectures show limited functionality to transfer complex access patterns such as convolutions or stencil codes, which are common in data processing solutions. Even so, the efficiency of these instructions becomes critical for the performance of IMC at system level. While various works of the State of the art propose stencil-specialized IMC architectures with efficient bandwidth usage, their application-specific design make them unsuitable for current general-purpose computing and future, unexplored, workload classes. On the other hand, general-purpose computing architectures can benefit from software frameworks to efficiently program and implement stencil codes, but often at the cost of data redundancy and an inefficiency of bandwidth usage.

We propose in this paper an Instruction Set Architecture (ISA) to automatically transfer multiple data streams according to programmable memory access patterns such as stencils or convolutions, and automatically organize them upon arrival to efficiently perform vector IMC. Our proposed ISA is based on the specifications of the Data-locality Management Unit (DMU), a memory controller designed to be tightly coupled to IMC architectures and provide memory instructions. We also present a novel feature we call Pattern Stream Encoding (PSE), a compact format to generate multiple data streams and automatically leverage data reuse opportunities from a single instruction. In this paper, our DMU ISA is dimensionned and integrated as an instruction subset of the Computational SRAM (C-SRAM)[3], an IMC architecture based on SRAM technology. Our evaluation methodology bases itself on three different system nodes ranging from embedded to high-performance computing, and six applications in the scope of linear algebra, Computer Vision and Image Processing. The evaluation of DMU ISA shows on average a speed-up of $\times 8.81$, an energy reduction factor of $\times 6.81$ and an improvement of the number of operations per cycle of $\times 4.59$, compared to a reference implementation using the cache hierarchy to transfer data on embedded systems. Our results also show a $\times 2.99$ improvement of Matrix multiplication for all system nodes.

```
for i in 0 to height-1 by 1
do
  for j in 0 to width-1 by 1
  do
    G'[i][j] = -1*G[i-1][j  ]
               -1*G[i  ][j-1]
               +4*G[i  ][j  ]
               -1*G[i  ][j+1]
               -1*G[i+1][j  ];
  done
done
```

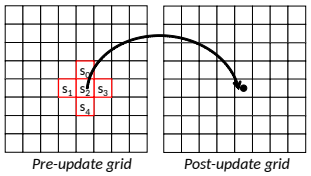



Fig. 1: Implementation of a discrete Laplacian operator as ISL. Left: pseudo-code of the ISL. Right: graphical representation of the stencil.

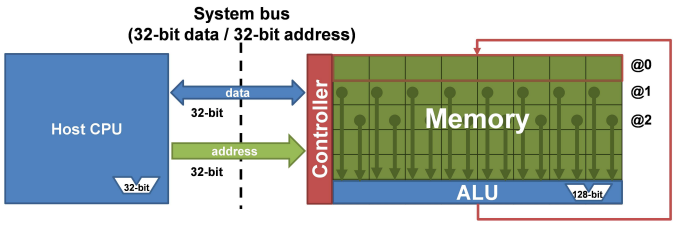



Fig. 2: Interface between C-SRAM and host CPU to exchange data and send instructions.

The rest of this paper is organized as follows. Section II presents the background and motivations behind this work. Section III presents the related work of the State of the art. Section IV introduces the DMU and presents our proposed instruction set. Section V presents a programming model for C-SRAM & DMU architectures. Section VI describes our evaluation methodology and our metrics of interest. Section VII presents and describes our obtained results. Finally, Section VIII concludes the paper and exposes future works.

## II. Background

### A. Iterative Stencil Loops

Iterative Stencil Loops (ISLs) are a class of data processing subprograms which solve numerical problems by the iterative update of regular data grids according to two main attributes: a stencil, *i.e* the pattern describing the neighborhood needed at each point for the latter to be updated and an update function taking as parameters the neighborhood mapped onto the stencil.

Implementing numerical problems using ISLs offers the benefit of easier discrete implementation of sometimes complex numerical solutions. They can be used to implement differential equation solvers in destination of physics modelling such as Navier-Stokes solvers[4] or Lattice-Boltzmann Methods[5]. Image processing applications also make use of ISLs to implement filters to perform tasks such as extracting edge features[6], detecting curves[7] or perform color interpolation on single-channel images[8]. Also, Convolutional Neural Networks (CNNs) are a popular class of artificial neural networks (ANNs), often used to extract features from an input image. The convolution operator CNNs are based on can be classified as an ISL. Furthermore we note that an important number of data processing operators, such as the Laplace operator shown on Figure 1, are part of standardized Application Programming Interfaces (APIs) such as OpenCV[9] and OpenVX[10].

### B. In-Memory Computing

In-Memory Computing (IMC) is an architectural paradigm where memories are enhanced with computation abilities. In the majority of the solutions presented in the literature, IMC performs parallel computation to improve the overall throughput achievable by its host CPU and reducing the bandwidth usage by substituting data exchanges with instructions. IMC can be based on various memory technologies such as DRAM[11][12], SRAM[13][14] or even ReRAM[15], and can implement either Single-Instruction Multiple-Data (SIMD) or Multiple-Instruction Multiple Data (MIMD) to achieve different levels of parallelism. The design of a given IMC architecture, *e.g* the integration of arithmetic operations to its base memory architecture, has a significant impact on how data must be arranged in the storage space dedicated to IMC to correctly compute data. Various publications present technological characterizations of IMC architectures with different memory technologies. Most of these papers describe accurately the physical implementation of arithmetical operations, but present no concrete Instruction Set Architecture (ISA) which can be used at system-scale. Furthermore, even when these solutions are effectively implemented on a test-chip, they primarily focus on the physical characterization of their design to evaluate and estimate its peak throughput without taking into account the data movements required across the system to fetch and send data. It is important to note, though, that most publications make use of IMC as application-specific accelerators[16][17], which justifies that such considerations are not addressed.

Our reference IMC architecture for the rest of this papier will be the Computational SRAM (C-SRAM) is an IMC architecture based on SRAM memory, with logical and 8-bit up to 32-bit integer vector operations. Figure 2 shows the interface model between C-SRAM and a 32-bit host CPU. The C-SRAM does not integrate an autonomous instruction flow. Instead, C-SRAM instructions are encoded and sent by the host CPU on both the data and address buses in form of "store-word" instructions, making them 64 bits wide. By this interface model, a single execution flow interleaves scalar CPU operations and C-SRAM vector operations. Moreover, the instruction set of the host architecture does not need to be modified for tightly-coupled system integration of C-SRAM, making it architecture-agnostic and easy to integrate in existing and future systems.

Figure 3 shows the instruction set encoding of C-SRAM architecture. The host CPU can use C-SRAM in two operating modes: computing mode and memory mode. Each mode of C-SRAM is toggled by the host CPU using the most significant bit of the address bus set to one. In consequence, C-SRAM owns two regions in the virtual address space: one for memory mode and the other for computing mode, *i.e* for encoding

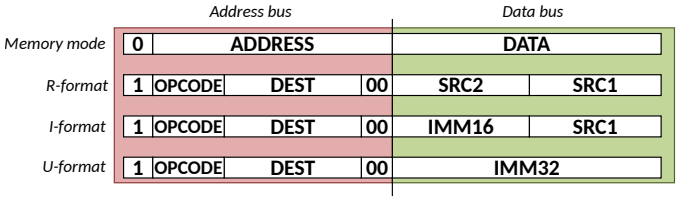



Fig. 3: Encoding of C-SRAM instructions on 32-bit host system.

C-SRAM instructions. In computing mode, three instruction formats are available: R-format with 2-rows operands, I-format for 1-row-1-immediate operands and U-format for 1-immediate operand. All three formats take a row as destination parameter.

## III. Related works

### A. Stencil computing on IMC architectures

Our overview of the State of the art focuses primarily on IMC solutions, with a particular interest to the following criteria: how these architectures are integrated to a host architecture, what interface model they use to for the former to interact with them, what programming model they provide and what memory instructions they implement. By our focus on ISLs, we are more particularly interested in whether or not these architectures can be programmed to issue intensive data transfers, and ideally with the ability to reorganize data transfers according to specific patterns.

Previous solutions of the literature proposed IMC architectures designed as application-specific accelerators for stencil computation. The Processing-In-Memory accelerator for Stencils (PIMS) for example is an IMC stencil accelerator implemented on a Hybrid Memory Cube to benefit from its 3D-stacked design and its high bandwidth[18]. According to the interface model presented in the original paper, each PE of PIMS is connected to a given memory bank, meaning that data needs to be stored in specific locations in memory to be computed. The PIMS architecture proposes no dedicated instructions for developers to manage data movements prior stencil computation, though elaborate communication and buffer mechanisms are implemented to seamlessly maximize transfer efficiency. The evaluation of PIMS showed interesting results regarding the pressure reduction on the memory banks of the Hybrid Memory Cube, by isolating metrics relevant to their architecture. CASPER, another stencil-specific IMC accelerator which shows promising performance results, proposed a programming model in the form of a dedicated API to issue data streams and trigger stencil computation[19]. In a manner similar to PIMS, seamless data reuse mechanisms are put into place to reuse data without any knowledge of developers. We note, however that the paper does not present the instruction set of this architecture, which is understandable as its design is specialized in any case for stencil computation.

These solutions show the interest of investigating stencil computation accelerator through IMC. However, we note that they may be too specialized to be programmed for other workloads. For these reasons, we also integrate in our overview general-purpose IMC architectures, and look at their provided transfer mechanisms. We retained from our analysis of the literature two architectures representative of the domain : the Duality Cache and the UPMEM architecture. The Duality Cache enhances the Last Level Cache (LLC) of its host architecture with a Very-Long Instruction Word (VLIW) execution model to perform MIMD programming[20]. Its DC-PTX ISA is based on a subset of Nvidia's PTX ISA to be retro-compatible with already existing code projects targeting CUDA applications and Nvidia Graphics Processing Units (GPUs). By this partial retro-compatibility, existing ISLs targeting Nvidia GPUs can be reused on the Duality Cache, with the condition of using compatible instructions. By its design specifications, the Duality Cache is more tightly coupled to the host architecture than a GPU, and takes advantages of its resources, *e.g* bandwidth, main memory, cache slices, to reduce data transfers compared to a conventional GPU. Load-store operations of PTX are redefined in DC-PTX to target cache slices and the main memory instead of an intermediate device-exclusive memory. Because the Duality Cache performs bit-serial computation, the input data need to be translated and organized along the same bit-lines in internal memory instead of along the same word-lines. To prevent the disruption of their ISA, a Transpose Memory Unit (TMU) is implemented to seamlessly perform data transposition and reorganization. The UPMEM architecture takes another approach by implementing Data-Processing Units (DPUs) directly in DRAM modules, making the execution model of this architecture MIMD. Each DPU owns a local memory to store instructions and data, and can also access a global memory shared with the host CPU. The UPMEM instruction set implements the specialized instructions `ldma`, `ldmai`, and `sdma` to exchange data between local and global memory. These instructions can be parameterized to transfer a single stream of contiguous bytes per instruction.

### B. Software frameworks for ISL programming

While not fit for stencil computation by design, these general-purpose IMC architectures could be programmed through high-level software frameworks to efficiently implement ISLs. The authors of the paper introducing the PIMS architecture mentioned Physis, an implicitly parallel programming model in destination of GPUs to implicitly describe parallel stencil computation[21]. We retained from our overview of the State of the art regarding Domain-Specific Languages (DSLs) and Application Programming Interfaces (APIs) multiple solutions specialized on implementing ISLs, for image processing pipelines[22][23][24] or general-purpose numerical solutions[25][26][27]. A consequential amount of software solutions to leverage the implementation of Machine Learning (ML) applications were also proposed. These solutions include high-level frameworks such as TensorFlow[28], Torch[29] and N2D2[30], low-level APIs such as FANN[31] and mlpack[32], and ML-specific compiler frameworks such

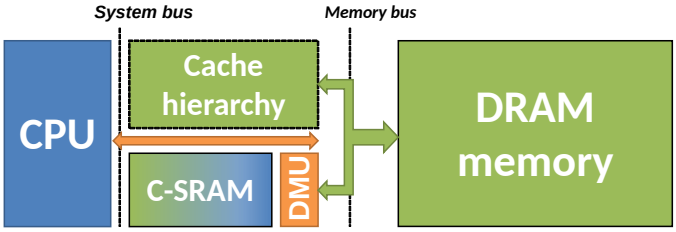



Fig. 4: System integration of DMU block to an IMC architecture, here C-SRAM.

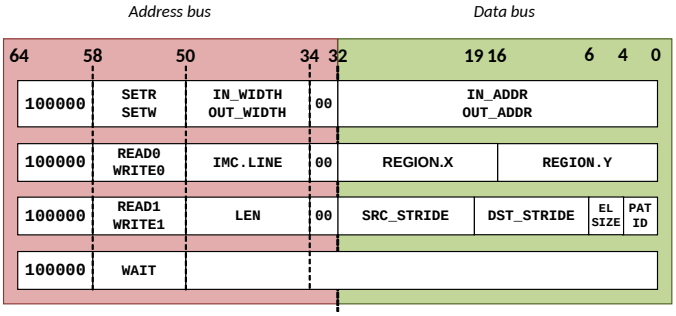



Fig. 5: Encoding of DMU instruction set on a 32-bit host system.

as TVM[33] and MLIR[34]. We note that the majority of these frameworks either targets GP-CPUs and GPUs, or application-specific accelerators in destination of ML. Regarding the former case, specific data layouts such as `im2col` are required to implement convolutional layers into matrix multiplications, which scales better with parallel computation architectures and high-bandwidth memory subsystems[35]. However, former investigations show that the overhead of `im2col` transformation induces a significant overhead in bandwidth pressure, thus aggravating the Energy Wall phenomenon due to the von Neumann bottleneck[36][37]. As discussed earlier, ML accelerators have hardware mechanisms to seamlessly perform data reuse and mitigate data redundancy, but they lack the flexibility to perform general-purpose stencil computation.

### C. Observations

To conclude our overview of the literature, we found no solution proposing an ISA compatible with general-purpose stencil computation and accessible to general-purpose IMC. We believe that the amount of software solutions using ISLs is a compelling motivation to propose a data transfer mechanism dedicated to efficient stencil computation. Our proposition for this challenge is an instruction set that can be programmed to transfer data according to any arbitrary access pattern. The stencil data are re-arranged, duplicated and organized upon arrival in the IMC storage to perform parallel energy-efficient stencil computation. This proposed ISA is implemented as a transfer, block tightly coupled to an IMC architecture and a bus between it and main memory, and as such can be integrated into any IMC architecture with minimal to no required modifications.

## IV. Data-locality Management Unit (DMU)

In this section, we introduce the Data-locality Management Unit (DMU), a memory controller designed to be tightly coupled to IMC architectures and provide memory instructions for efficient data transfers. In [38], the authors presented the base specification of the DMU architecture and evaluated its integration to an IMC architecture on an embedded system for three applications. The results showed that the system integration of DMU improves the performance of IMC on an embedded architecture implementing C-SRAM and, more specifically, can be used to reduce bandwidth usage on data-dependent applications such as stencils and convolutions. In this paper, we propose for the DMU an instruction set to automatically transfer multiple data streams according to a programmable memory access patterns, and organize them upon arrival to efficiently perform vector IMC. The ISA we propose in this paper rests on the primary specifications presented in [38] and adds a novel feature we call Pattern Stream Encoding (PSE), to generate multiple data streams from a single instruction and automatically leverage data reuse opportunities.

### A. DMU Instruction Set

By the definition of ISLs discussed in Section II-A, we identify the following parameters as necessary to implement the data transfers of a given ISL using the DMU instruction set: the *input references* containing the base address of the input data structure and $[X, Y]$ starting coordinates to read data from, and the *output references* containing the base address of the output data structure and $[X, Y]$ starting coordinates to write data at. Using relative coordinates instead of virtual addresses only is important to identify the elements mapped in the parameterized stencil shape for the upcoming transfer. From these observations, we identify two main operations to exchange data between C-SRAM and the main memory, labeled `read` and `write`. The `read` operation transfers streams of stencils according to the parameterized stencil shape, using respectively the input references and a C-SRAM row identifier to parameter where to start reading stencils from and writing them at. By symmetry, the `write` operation uses a C-SRAM row identifier and output references to parameter where to transfer data fro mC-SRAM in destination of DRAM.

We decide to implement the instruction set formats of our DMU instruction set using C-SRAM as a base, so that it can be integrated as an instruction subset in the C-SRAM ISA. This choice allows us to implement 64 bits wide instructions on 32-bit architecture systems to program the DMU. Figure 5 shows the DMU instruction set formats, obtained after performing parameter scaling according to the C-SRAM instruction set formats.

Because an ISL often targets a single pair of input/output data structures, certain parts of the input/output references such as their base address and their width are consistent for each data transfer generated during the execution. To benefit from this redundancy and reducing the quantity of bits required to

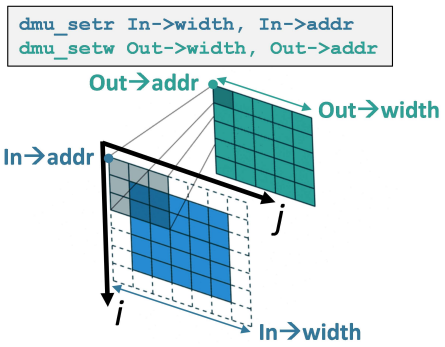



Fig. 6: Example usage of `setr` and `setw` instructions to target *In* and *Out* data structures, both stored in DRAM memory.

```
algorithm READ(IN_ADDR, IN_WIDTH,
               X, Y, IMC_LINE,
               SRC_STRIDE, DST_STRIDE,
               EL_SIZE, LEN)
begin
  for i := 0 to LEN-1
  do
    SRC = IN_ADDR+IN_WIDTH*ELSIZE*X
        + ELSIZE*Y*SRC_STRIDE;
    DST = ADDR_OF(IMC_LINE)+i*ELSIZE*DST_STRIDE;
    for j := 0 to EL_SIZE-1
    do
          DST[j] = SRC[j];
    done
  done
end
```

Fig. 7: Algorithm of the behaviour of `read` instruction with no parametered PSE, (*i.e* PSE identifier equals 0).

parameter data transfers, we implement the `setr` and `setw` instructions to persistently store the widths and base addresses of the input and output data structures respectively. Figure 6 shows an example usage of `setr` and `setw` instructions targeting "In" and "Out" to perform 2D filtering. The formats of the `setr` and `setw` instructions are based on the U-format of C-SRAM instructions, allowing 32-bit addresses and 16-bit width parameters. These parameters are identified as `IN_ADDR`/`IN_WIDTH` and `OUT_ADDR`/`OUT_WIDTH` for `setr` and `setw` respectively. By this scaling, the entirety of a 32-bit logical memory space can be addressed by the `set` instructions. However it is possible to implement platform-specific variants of these instructions, based on the actual size of the physical memory space for a given platform.

After scaling to the C-SRAM instruction set, the `read` and `write` operations are split into two instructions: `read0` (resp. `write0`) and `read1` (resp. `write1`). The parameters taken by `read0` and `write0` are the C-SRAM row index and $[x, y]$ coordinates relative to the targeted data structure. `read1` and `write1` take as parameters the source and destination stride to which data should be read and written, the PSE identifier,

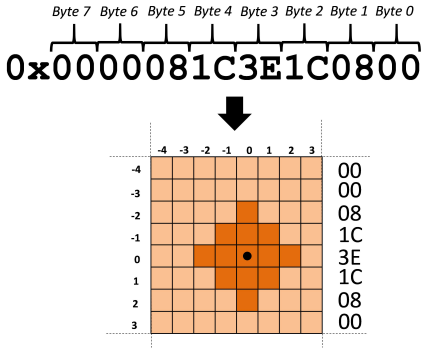



Fig. 8: Each bit of a PSE bitmap represents a neighbor according to a reference13-point 2D stencil encoded as a PSE bitmap.

| Byte# | Minimum base address | Maximum base address |
|---|---|---|
| 0 | $@IN\_ADDR[x+3][y-4]$ | $@IN\_ADDR[x+3][y+3]$ |
| 1 | $@IN\_ADDR[x+2][y-4]$ | $@IN\_ADDR[x+2][y+3]$ |
| 2 | $@IN\_ADDR[x+1][y-4]$ | $@IN\_ADDR[x+1][y+3]$ |
| 3 | $@IN\_ADDR[x][y-4]$ | $@IN\_ADDR[x][y+3]$ |
| 4 | $@IN\_ADDR[x-1][y-4]$ | $@IN\_ADDR[x-1][y+3]$ |
| 5 | $@IN\_ADDR[x-2][y-4]$ | $@IN\_ADDR[x-2][y+3]$ |
| 6 | $@IN\_ADDR[x-3][y-4]$ | $@IN\_ADDR[x-3][y+3]$ |
| 7 | $@IN\_ADDR[x-4][y-4]$ | $@IN\_ADDR[x-4][y+3]$ |

TABLE I: Table of generable source base addresses by a PSE bitmap.

the number of elements to transfer for each stream generated by the PSE and their size in bytes. The former provides the first part of the transfer parameters, while the latter provides the second part and triggers the DMU transfer. Figure 7 describes the behavior expected of `read` instruction. In this example, no particular PSE bitmap is programmed yet to transfer multiple streams at once, which is selected with the PSE identifier #0.

Since both instructions are non-blocking, a `wait` instruction is implemented to block the host CPU until completion of the overall transfer, *i.e* the entire set of streams generated from the PSE identifier.

### *B. Functionality of DMU instruction set*

*1) DMU Pattern Stream Encoding:* To program stencil neighborhoods in a reasonably compact manner, we propose an encoding format we call Pattern Stream Encoding (PSE). We implement PSE primarily to automatically align multiple data streams in C-SRAM according to its vector computation scheme, and improve the efficiency of data transfers triggered by `read` instructions. As of now, PSE is exclusively used for `read` instructions, as we the applications we investigate use reducers – such as convolutions – as ISL update functions. PSE identifier #0 being used by default to perform single-stream transfers. We plan on investigating on the interest of writing data from IMC to other memories, according to complex access patterns, in the near future. Figure 8 shows an example PSE bitmap to transfer multiple data streams according to the memory access pattern of a 13-point 2D stencil code.

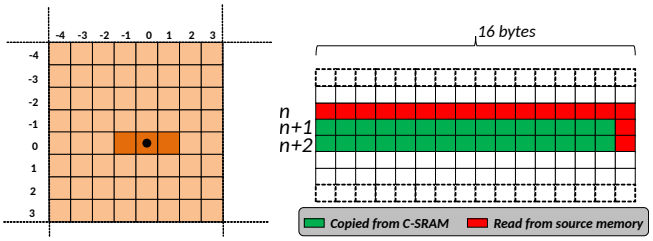



Fig. 9: Left : PSE bitmap to transfer 16 1-byte 1D stencil. Right : data layout obtained in C-SRAM storage, each stencil window has its elements aligned to correctly perform computation. Highlighted in green aer the elements which were copied directly from C-SRAM storage.

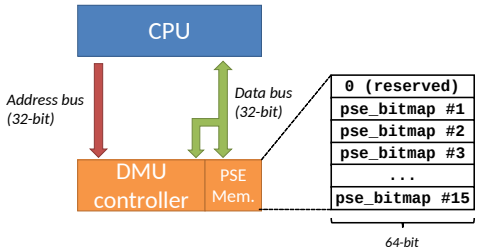



Fig. 10: Interface model between Host CPU and DMU, with its PSE memory.

Each bit in a PSE bitmap encodes a specific address relative to the $[x, y]$ coordinates, according to Table I. From a structural point of view, PSE bitmaps can be used to generate addresses in an $8 \times 8$ window around $[x, y]$. The address set generated by PSE bitmaps was defined to cover the access patterns encountered in our applicative study cases, but it also provides the flexibility to program it for arbitrary patterns.

*2) Automatic data reuse:* To seamlessly leverage data reuse opportunity when transferring data with DMU, streams belonging to the same PSE row – *aka* generated from the same byte of a given PSE bitmap – are partially copied directly from C-SRAM when possible, instead of read from the input data structure. The missing elements of the stream are then read from source memory, improving the latency and energy efficiency of the overall DMU transfer. To determine when data reuse possibilities are accessible, we use the PSE bitmap itself is used as a hint to detect for each stream of a given PSE row the initial stream, whose data have not been previously transferred in C-SRAM, and the subsequent streams, whose data are already partially transferred but need to be copied in the appropriate location in C-SRAM. Figure 9 shows as example the transfer of 16 3-point stencils of element size 1-byte. The PSE bitmap necessary for this access pattern will generate in consequence 3 data streams. The first stream is loaded in the $n^{th}$ row in C-SRAM with all its data read exclusively from the input data structure, while the $(n+1)^{th}$ and $(n+2)^{th}$ streams are partially copied from data already transferred in C-SRAM, then completed by reading the missing data from the input data structure. The resulting data layout in C-SRAM storage is each stencil window correctly aligned across the destination rows to perform stencil computation.

Figure 10 shows the interface model between the host CPU and DMU. The host CPU writes PSE bitmaps into a memory-mapped PSE buffer, indexed by the PSE identifier field of `read` instructions. This field being 4 bits wide, the PSE buffer is defined to store $16 \times 64$ bitmaps, `or` 128 bytes of raw data. The first double-word of this buffer is reserved to use single-stream DMU transfers, which is defined by the PSE bitmap of value 0, leaving 15 programmable bitmaps to the host CPU.

## V. C-SRAM & DMU PROGRAMMING MODEL

```
/* PSE buffer initialization */
#define 5PT_STEN_PSE 0x81C080000
#define 5PT_STEN_IDX 1
_pse_buffer[5PT_STEN_IDX] = 5PT_STEN_PSE;
/* Variable declarations */
unsigned short coef [1, 1, -4, 1, 1];
unsigned short *imc_coef, *imc_stenc, *imc_res;
/* (1) Alloc. / init. of C-SRAM memory */
imc_coef = imc_alloc(5);
imc_stenc = imc_alloc(5);
imc_res = imc_alloc(1);
imc_init_rows(imc_coef, coef, 5);
/* (2) Setting target regions */
dmu_setr(width, In);
dmu_setw(width, Out);
/* Main iteration loop */
for (int i = 1; i < height-1; i += 1)
{
  for (int j = 1; j < width-1; j += 8)
  {
       /* (3) Input transfer of 5-point stencil*/
    dmu_read0(imc_coef, i, j);
    dmu_read1(8, 1, 2, szof(char), 5PT_STENC_PSE);
       /* (4) In-Memory Computation */
    ....
       /* (5) Input transfer of single stream*/
    dmu_write0(imc_res, i, j);
    dmu_write1(8, 2, 1, szof(char), 0);
  }
}
```

Fig. 11: Implementation of discrete laplacian operator using DMU instruction set.

As of now, the programming models implemented for C-SRAM make use of explicit vectorization at high level[39], [40]. Following this, we implement for this paper the support of DMU instructions as an explicit programming model using C macros. Figure 11 shows an example code implementing a discrete laplacian operator using the resulting programming model. We define an `imc_alloc()` routine to implement stack allocation in C-SRAM, and an `imc_init_rows()` to initialize all the rows of an allocated region using the `_cm_bcast` C-SRAM instruction.

Figure 12 shows the data layout after completing the first iteration of the innermost loop in the example code shown in Figure 11. The convolution windows of center points $[i, j]$ up to $[i, j+7]$ are transferred according to the 5-point stencil pattern programmed in the PSE buffer. The streams generated by the `read` instructions are read from the target region at

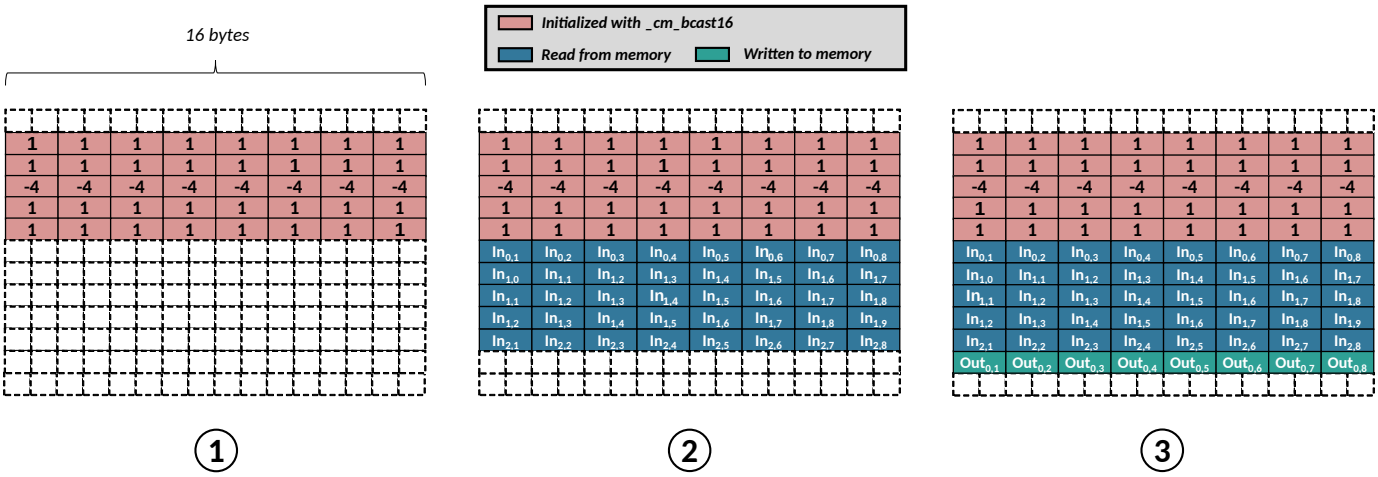



Fig. 12: Data layout obtained in C-SRAM architecture through the example code shown in Figure 11, obtained after $1^{st}$ iteration of the innermost loop.. (1) The weights of the convolution filter are initalized with `imc_init_rows()`. (2) Input data is transfered through the `read` instructions. (3) Atfer using IMC, the data t be written in Out are transfered using `write` instructions.

stride 1 and written in C-SRAM at stride 2 to perform byte extension for 16-bit computation. After that, IMC arithmetic operations are used to compute the convolution of the input windows and clip them to 8-bit values – *e.g* between 0 and 255. Respectively, the streams generated by `write` instructions are read from C-SRAM at stride 2 and written in the target region at stride 1 for proper data organization.

While this programming model is efficient at programming DMU, it requires developers to be familiar with internal and unconventional hardware features. We are currently working on a solution to implement the full support of DMU and PSE at high level, and plan soon on publishing a paper presenting our results.

## VI. Experimental methodology

In this section, we present our experimental methodology to evaluate the integration of C-SRAM/DMU ISA in a host architecture. Our goal is to evaluate the behavior of different applications implemented in this system, depending on the performance of its memory subsystem. We use an emulation framework based on QEMU[40] to evaluate the ISA at system-level, as opposed to micro-benchmarking. The energy and latency costs of cache memories and DRAM are obtained using [41] while the costs of C-SRAM are taken from [42].

### A. Experimental system nodes

We base our methodology around the RISC-V architecture, as its specification is open-source and backed by major industrial and academic actors. At the time of writing this paper, various cores designed by SiFive were implemented in various products. As such, we model our three system nodes based on three implemented SiFive cores to evaluate our proposed ISA in a realistic setting. Our selection includes the E31[43], E76[44] and U74[45] cores, each targeting different systems ranging from Internet-of-Things (IoT) to High-Performance Computing (HPC). However, none of these cores implement the RISC-V Vector (RVV) extension to support wide-SIMD computation. We plan on evaluating comparisons with RISC-V processors supporting RVV as soon as the State of the art will be able to provide performance characterizations.

| | E31-based *384MHz* | | E76-based *800MHz* | | U74-based *1GHz* | |
|---|---|---|---|---|---|---|
| | Latency (cycles) | Energy Cost | Latency (cycles) | Energy Cost | Latency (cycles) | Energy Cost |
| 8KB C-SRAM | 1 | 31.74pJ | 2 | 31.74pJ | 3 | 31.74pJ |
| L1 I$ | 1 | R: 19pJ W: 25pJ | 1 | R: 19pJ W: 25pJ | 1 | R: 24pJ W: 24pJ |
| L1 D$ | 1 | R: 34pJ W: 34pJ | 1 | R: 34pJ W: 34pJ | 1 | R: 24pJ W: 24pJ |
| L2 $ | | | | | 12 | R: 52pJ W: 52pJ |
| DRAM | 7 | R: 8.17nJ W: 8.04nJ | 24 | R: 14.45nJ W: 14.35nJ | 48 | R: 39nJ W: 37.5nJ |

TABLE II: Model parameters for experimental system nodes.

Figure 13 shows the memory hierarchies of the system nodes for our experiments. Both E31 and E76-based nodes have a single level of cache hierarchy, with the same 16KB instruction cache. The E31-based node uses a 32KB 2-way data cache, while the E76-based node implements a 16KB 4-way data cache. The U74-based node implements a two-level cache hierarchy, with 32KB 4-way L1 caches and a 128KB 8-way L2 cache. The E31-based node uses data write-through policy while both the E76 and U74 use data write-back policy. Table II shows the energy and latency parameters selected to model each system node.

### B. Applications

Table III presents the five applications implemented on the C-SRAM architecture. The criteria we retain to evaluate the results of our experimentation are their algorithmic complexity and their vector element size, as both have an impact on the

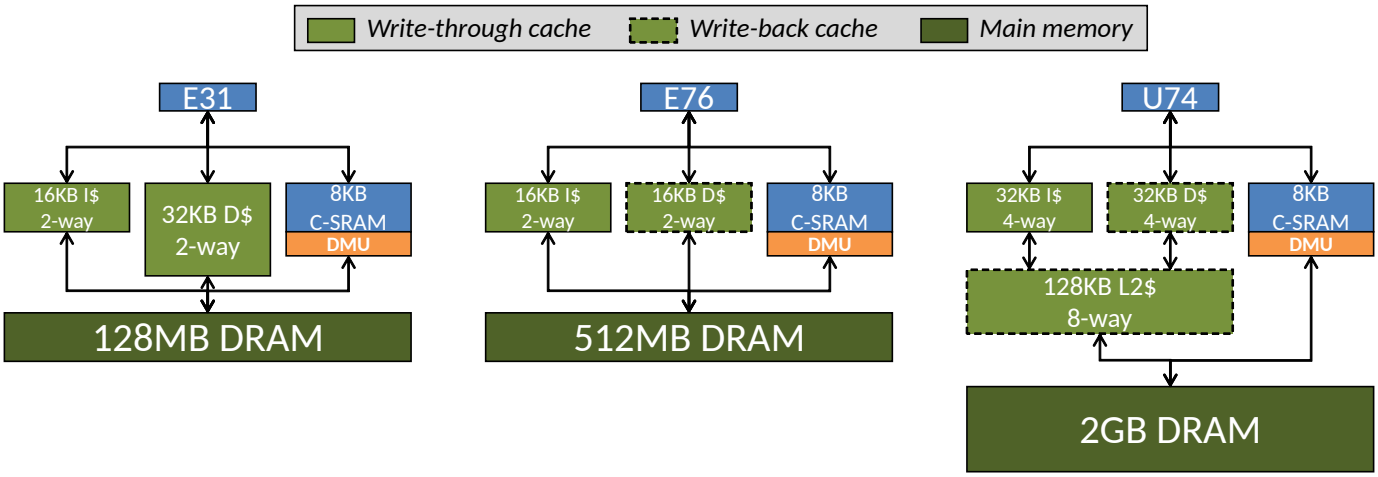



Fig. 13: Memory hierarchies of the system nodes modeled for our experimental methodology.

| Application | Vector element size | Complexity | Pattern type | Average data redundancy |
|---|---|---|---|---|
| **Frame difference [46]** | 8-bit | $\mathcal{O}(n)$ | On-load: row-major On-store: row-major | 1 |
| **Laplace filter [47]** | 16-bit | $\mathcal{O}(n)$ | On-load: stencil (5-pt) On-store: row-major | $\approx 5$ |
| **Sobel filter [6]** | 16-bit | $\mathcal{O}(n)$ | On-load: stencil (9-pt) On-store: row-major | $\approx 18$ |
| **Matrix multiplication (squares) [48]** | 32-bit | $\mathcal{O}(n^3)$ | On-load: row/col-major On-store: row-major | $n^2$ |
| **Demosaicking (AoS / SoA) [49]** | 16-bit | $\mathcal{O}(n)$ | On-load: stencil (13-pt) On-store: irr.→row-major | $\approx 26$ |

TABLE III: List of Applications with their Characteristics for Experimental Evaluation.

arithmetic complexity of the applications and whether or not the main generated access pattern is regular or complex. We also define for each application their average data redundancy factor, *e.g.* the number of times an input element is accessed during the entire life cycle of the application:

*a) Frame difference:* computes the saturated subtraction of two consecutive frames to highlight their differences. It is often used in CV for motion detection[46].

*b) Laplace filter:* is a discrete Laplacian operator implemented as a 5-point stencil code. It can be used in CV as an edge detection operator[47].

*c) Sobel filter:* is another edge detection operator, implemented as two $3 \times 3$ convolution kernels[6]. Both kernels are applied on the same inputs, and as such must be spanned across all compute lanes using broadcast instructions.

*d) Matrix-matrix multiplication:* is a cornerstone of linear algebra and various workloads in Computer Sciences, often used as the `gemm` operator[50]. We implement a tiled version of Matrix multiplication to maximize the data reuse and the throughput. To prevent the need for matrix transposition before computation, we use a vectorization scheme resting on broadcast instructions.

*e) Demosaicking:* is a digital image process to recolor an image sampled from an image sensor with a color filter array. The algorithm used as a baseline for the implementation is the Malvar-He-Cutler demosaicking algorithm[49]. This implementation uses eight 5x5 filters divided into two categories: red-row and blue-row filters. To maximize throughput, we compute multiple convolutions at once by arranging alternatively different stencils in C-SRAM. This data layout could be compared to weight-stationary layouts for Neural Network Computation[35]. We evaluate two variants: one targeting a Struct-of-Array (SoA) output, and the other targeting an Array-of-Struct (AoS) output. The impact of this difference lies in how to organize data after computation to efficiently write them to the output structure. In the SoA variant, bit masking can be used in C-SRAM to reorganize the color components and in the AoS variant, the CPU has to sequentially reorganize components as consecutive pixels using load and stores in C-SRAM.

The variety of this selection in terms of complexity and data reuse generates various distinct run-time behaviors to analyze. All three applications were retained for their importance and relevance to domains such as image processing and computer vision. Most are defined as standard functions in Domain-Specific APIs such as OpenVX [10].

### C. Performance metrics

For our experimental methodology, we use metrics commonly found in the State of the art as well as novel metrics to evaluate the quality of generated software. To compare the impact of integrating our DMU instruction set on a given for a given application, we measure the speed-up and the energy

reduction relative to the baseline execution of the application on the node by using its cache hierarchy instead of the DMU to transfer data. We also measure for each application and on each system node the average number of operations per cycle for each application on each system node. Finally, we study the percentage of C-SRAM usage time between actively waiting for transfers to finish and computing data, and the percentage of C-SRAM and CPU instructions.

## VII. Evaluation results

Figure 14 shows for each application on each system node the speed-up and the energy reduction of C-SRAM when transferring data through DMU, relative to a baseline code implementation. This baseline implementation uses the cache hierarchy of the host CPU to transfer data from DRAM to C-SRAM, to evaluate the impact of the integration of DMU to each system node. We note that both the performance and the energy efficiency of C-SRAM are improved on average by the integration of DMU on the E31-based node by $\times 8.81$ and $\times 6.81$ respectively, and every application was improved individually. On the E76 and U74-based nodes, these (speed-up/energy-reduction) factors are $(\times 1.51/\times 1.28)$ and $(\times 0.67/\times 1.175)$. Across all system nodes, the application that was consistently improved in terms of both energy reduction and speed-up is Matrix multiplication. We also note that starting a 2-level cache hierarchy – *e.g* on the U74-based node, the energy efficiency is more improved than the speed-up. Our conclusion is that the integration of DMU to C-SRAM with its current host/interface model is more interesting for embedded systems. The efficient integration of DMU and C-SRAM in higher-end nodes require different architectural approaches, such as increasing the degree of parallelism achievable while keeping C-SRAM tightly coupled to the host CPU[39], or moving C-SRAM further up in the memory hierarchy, past DRAM memory and close to lower-latency memories to improve C-SRAM-memory communication[51]. We plan to investigate and evaluate on the impact of DMU on these different integration paradigms.

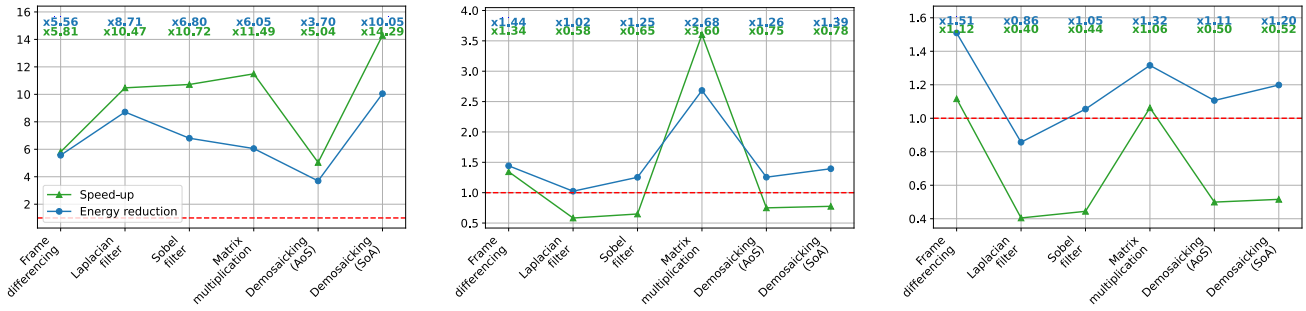
Figure 15 shows for each application on each system node the average number of operations per cycle measured by the end of execution. Next to each bar, an annotated star indicates for a given system node whether the variant with or without DMU is more efficient in terms of operations per cycle. We note that due to its low arithmetic intensity, Frame difference shows the lowest number of operations per cycle on every system node – $0.09$, $0.18$, and $0.37$ operations per cycle, respectively for the E31-based, E76-based and U74-based node. These results show that C-SRAM by itself amortizes the cost of memory access on most systems and applications. The first application to be amortized is the Laplacian filter with $2.72$ operations per cycle on the E31-based node and the last is Matrix multiplication with $10.15$ operations per cycle. This trend is also observed on the E76-based and U74-base nodes. We observe on average a $\times 4.59$ improvement of the numbers of operation for every application run on E31-based node, effectively improving its performance. On the E76 and U74-based nodes, the memory hierarchy of the host architecture is efficient enough to efficiently transfer data to C-SRAM without using DMU. The number of operations per cycle on Matrix multiplication is consistently improved by the integration of DMU on every system node by $\times 2.99$ on average, the improvement factors ranging from $\times 1.18$ on U74-based node to $\times 5.41$ on E31-based node. This is because in order to perform loop tiling of matrix multiplication, the no-DMU variant reads data of the first operand matrix to be broadcasted from the cache hierarchy instead directly transferring them to the C-SRAM, making the performance of matrix multiplication sensitive to cache misses and evictions.

Figure 16 shows for each application on E31-based node the distribution of C-SRAM usage time between computation and the active wait for data using the `_dmu_wait` instruction. We can observe that computation makes for $26.753\%$ of the active usage time of C-SRAM for matrix multiplication, and for $12.823\%$ on average for 2D filtering applications, *e.g* Laplacian filter, Sobel filter, and Demosaicking. As both these types of kernels are often used in applications such as Machine Learning, these data give us a preliminary estimation of how much C-SRAM can be actively used with DMU in a synchronous process. We are currently studying the impact of the asynchronous use of DMU to improve the percentage of active C-SRAM computation time, and plan to publish the results in the near future.

Figure 17 shows for each application the percentage of CPU and C-SRAM instructions executed at run-time. These results are identical for each system node since the number of executed instructions is only dependent on the compilers which generate the implemented code. We observe that C-SRAM instructions make on average $5\%$ of the executed instructions, meaning that the compiled software generates on average 1 C-SRAM instruction every 19 CPU instructions. The percentage of CPU instructions could be reduced by optimizing further the generation of C-SRAM instructions, which make for the majority of executed instructions in a C-SRAM-specialized code.

## VIII. Conclusion

We presented in this paper, a new instruction set for DMU, a memory controller designed to efficiently transfer data between main memory and C-SRAM. We implemented this instruction set to automatically organize and optimize multiple data streams thanks to a feature called Pattern Stream Encoding (PSE). The experimental methodology we used to evaluate the proposed approach is based on three different system nodes ranging from embedded to high-performance computing, and six applications in the scope of Linear Algebra, Computer Vision and Image Processing. For a system using proposed DMU instruction set, experiments show on average a $\times 8.81$ speed-up, a $\times 6.81$ energy reduction and a $\times 4.59$ improvement of the number of operations per cycle compared to a reference implementation using the cache hierarchy to transfer data on embedded systems. Our results also show on average a $\times 2.99$ improvement of Matrix multiplication for all system nodes.




(a) E31-based node. (b) E76-based node. (c) U74-based node.

Fig. 14: Speed-up and energy reduction relative to baseline implementation @$1280 \times 720$ image resolution and 4MB matrices on every node and application.

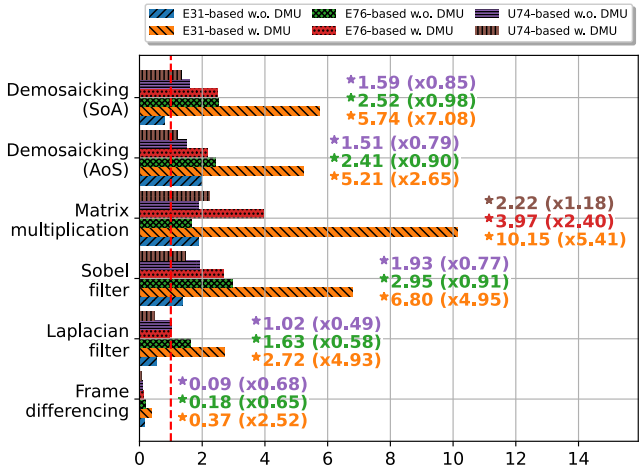



Fig. 15: Average number of operations per cycle @$1280 \times 720$ image resolution and 4MB matrices on every node and application.

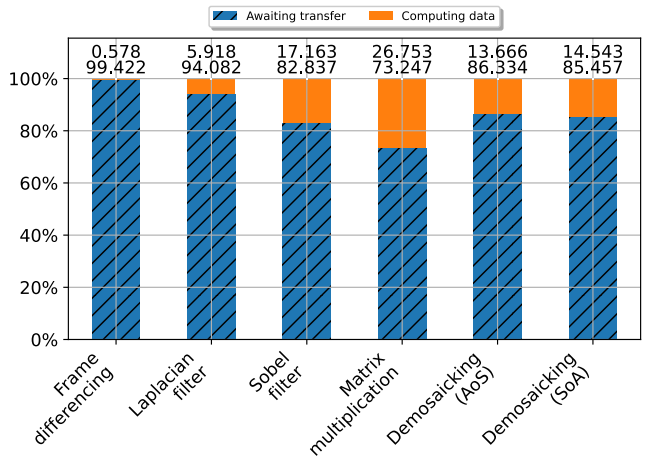



Fig. 16: Percentage of active C-SRAM usage time for every application on E31-based node.

The results obtained on high-performance system nodes are good insight for future IMC designs and integration. As future work, we aim to consider a system with asynchronous use of DMU instruction set to perform data transfers in parallel to computation in the C-SRAM. In addition we plan to implement a full-fledge programming model which efficiently leverages the features of C-SRAM and DMU.

## References

[1] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14.

[2] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[3] M. Kooli, H.-P. Charles, C. Touzet, B. Giraud, and J.-P. Noel, "Smart instruction codes for in-memory computing architectures compatible with standard sram interfaces," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1634–1639.

[4] N. Rusli, E. H. Kasiman, K. B. H. Ahmad, A. Y. M. Yassin, and N. Amin, "Numerical computation of a two-dimensional navier-stokes equation using an improved finite difference method," *MATEMATIKA: Malaysian Journal of Industrial and Applied Mathematics*, pp. 1–9, 2011.

[5] S. Guo, Y. Feng, J. Jacob, F. Renard, and P. Sagaut, "An efficient lattice boltzmann method for compressible aerodynamics on d3q19 lattice," *Journal of Computational Physics*, vol. 418, p. 109570, 2020.

[6] "The OpenVX Specification: Sobel 3x3." [Online]. Available: https://www.khronos.org/registry/OpenVX/specs/1.1/html/da/d4b/group__group__vision__function__sobel3x3.html

[7] P. Getreuer, "Contour stencils: Total variation along curves for adaptive image interpolation," *SIAM Journal on Imaging Sciences*, vol. 4, no. 3, pp. 954–979, 2011.

[8] ——, "Image demosaicking with contour stencils," *Image Processing On Line*, vol. 2, pp. 22–34, 2012.

[9] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[10] K. V. W. Group, "The OpenVX Specification version 1.2," Oct. 2017. [Online]. Available: https://www.khronos.org/registry/OpenVX/specs/1.2/OpenVX_Specification_1_2.pdf

[11] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 273–287.

[12] M. F. Ali, A. Jaiswal, and K. Roy, "In-memory low-cost bit-serial addition using commodity dram technology," *IEEE Transactions on*

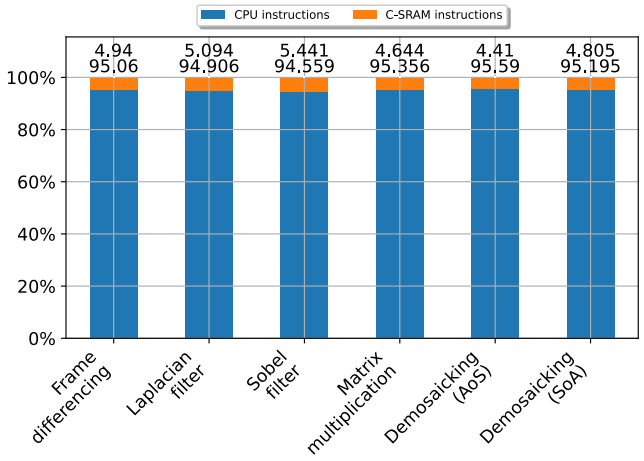



Fig. 17: Percentage of executed CPU and C-SRAM instructions during run-time on every node and application.

*Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 155–165, 2020.

[13] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester, "A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 76–86, 2020.

[14] S. Yin, Z. Jiang, J.-S. Seo, and M. Seok, "Xnor-sram: In-memory computing sram macro for binary/ternary deep neural networks," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 6, pp. 1733–1743, 2020.

[15] D. Bhattacharjee, Y. Tavva, A. Easwaran, and A. Chattopadhyay, "Crossbar-constrained technology mapping for reram based in-memory computing," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 734–748, 2020.

[16] Y. Long, T. Na, and S. Mukhopadhyay, "Reram-based processing-in-memory architecture for recurrent neural network acceleration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 12, pp. 2781–2794, 2018.

[17] K. Nagarajan, S. S. Ensan, M. Nasim Imtiaz Khan, S. Ghosh, and A. Chattopadhyay, "Shine: A novel sha-3 implementation using reram-based in-memory computing," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2019, pp. 1–6.

[18] J. Li, X. Wang, A. Tumeo, B. Williams, J. D. Leidel, and Y. Chen, "Pims: a lightweight processing-in-memory accelerator for stencil computations," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 41–52.

[19] A. Denzler, R. Bera, N. Hajinazar, G. Singh, G. F. Oliveira, J. Gómez-Luna, and O. Mutlu, "Casper: Accelerating stencil computation using near-cache processing," *CoRR*, vol. abs/2112.14216, 2021. [Online]. Available: https://arxiv.org/abs/2112.14216

[20] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 1–14.

[21] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[22] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[23] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 429–443, 2015.

[24] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines." *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144–1, 2014.

[25] M. Bianco and U. Varetto, "A generic library for stencil computations," *arXiv preprint arXiv:1207.1746*, 2012.

[26] T. Zhao, S. Williams, M. Hall, and H. Johansen, "Delivering performance-portable stencil computations on cpus and gpus using bricks," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2018, pp. 59–70.

[27] N. Zhang, M. Driscoll, C. Markley, S. Williams, P. Basu, and A. Fox, "Snowflake: A lightweight portable stencil dsl," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 795–804.

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[29] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS workshop*, no. CONF, 2011.

[30] O. Bichler, D. Briand, V. Gacoin, B. Bertelone, T. Allenet, and J. Thiele, "N2d2-neural network design & deployment," *Manual available on Github*, 2017.

[31] S. Nissen *et al.*, "Implementation of a fast artificial neural network library (fann)," *Report, Department of Computer Science University of Copenhagen (DIKU)*, vol. 31, no. 29, p. 26, 2003.

[32] R. R. Curtin, M. Edel, M. Lozhnikov, Y. Mentekidis, S. Ghaisas, and S. Zhang, "mlpack 3: a fast, flexible machine learning library," *Journal of Open Source Software*, vol. 3, p. 726, 2018. [Online]. Available: https://doi.org/10.21105/joss.00726

[33] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[34] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[35] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, 2017.

[36] M. Dukhan, "The indirect convolution algorithm," *arXiv preprint arXiv:1907.02129*, 2019.

[37] L. Lai, N. Suda, and V. Chandra, "Not all ops are created equal!" *arXiv preprint arXiv:1801.04326*, 2018.

[38] K. Mambu, H.-P. Charles, M. Kooli, and J. Dumas, "Towards integration of a dedicated memory controller and its instruction set to improve performance of systems containing computational sram," *Journal of Low Power Electronics and Applications*, vol. 12, no. 1, p. 18, 2022.

[39] R. Gauchi, V. Egloff, M. Kooli, J.-P. Noel, B. Giraud, P. Vivet, S. Mitra, and H.-P. Charles, "Reconfigurable tiles of computing-in-memory sram architecture for scalable vectorization," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 121–126.

[40] K. Mambu, H.-P. Charles, J. Dumas, and M. Kooli, "Instruction set design methodology for in-memory computing through qemu-based system emulator," in *32rd International Workshop on Rapid System Prototyping*, 2021.

[41] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to understand large caches," *University of Utah and Hewlett Packard Laboratories, Tech. Rep*, vol. 147, 2009.

[42] J.-P. Noel, M. Pezzin, R. Gauchi, J.-F. Christmann, M. Kooli, H.-P. Charles, L. Ciampolini, M. Diallo, F. Lepin, B. Blampey *et al.*, "A 35.6 tops/w/mm$^2$ 3-stage pipelined computational sram with adjustable form factor for highly data-centric applications," *IEEE Solid-State Circuits Letters*, vol. 3, pp. 286–289, 2020.

[43] I. SiFive, "SiFive FE310-G000 Manual v2p3." [Online]. Available: https://static.dev.sifive.com/FE310-G000.pdf

[44] B. Wheeler, "WD Rolls Its Own RISC-V Core," *The Linley Group Microprocessor Report*, vol. February 2019, p. 3, 2019. [Online]. Available: https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/analyst-report/WD_Rolls_Its_Own_RISC-V_Core.pdf

[45] I. SiFIve, "HF105 Datasheet." [Online]. Available: https://sifive.cdn.prismic.io/sifive/d0556df9-55c6-47a8-b0f2-4b1521546543_hifive-unmatched-datasheet.pdf
[46] N. Singla, "Motion detection based on frame difference method," *International Journal of Information & Computation Technology*, vol. 4, no. 15, pp. 1559–1565, 2014.
[47] "OpenCV: Laplace Operator." [Online]. Available: https://docs.opencv.org/3.4/d5/db5/tutorial_laplace_operator.html
[48] B. L. A. S. T. B. F. Standard, "Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard," Tech. Rep., Aug. 2001. [Online]. Available: http://www.netlib.org/blas/blast-forum/blas-report.pdf
[49] H. Malvar, L.-w. He, and R. Cutler, "High-quality linear interpolation for demosaicing of bayer-patterned color images," vol. 3, 06 2004, pp. iii – 485.
[50] B. L. A. S. T. B. F. Standard, "Basic linear algebra subprograms technical (blast) forum standard," 2001.
[51] V. Egloff, J.-P. Noel, M. Kooli, B. Giraud, L. Ciampolini, R. Gauchi, C. Fuguet, É. Guthmuller, M. Moreau, and J.-M. Portal, "Storage class memory with computing row buffer: A design space exploration," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1–6.