RISC-V ISA Extensions with Hardware Acceleration for Hyperdimensional Computing

Rocco Martino, PhD Candidate
rocco.martino@uniroma1.it

RISC-V Summit Europe 2025. Paris, 12-15 May.

SAPIENZA
UNIVERSITÀ DI ROMA

Outline

Introduction
- Hyperdimensional Computing Paradigm
- Why a custom extension

Proposed Solution
- Klessydra T03 RISC-V Core
- The HDCU

Results
- Hardware resource usage
- Speed up performance

DIGITAL VLSI CIRCUITS AND SYSTEMS RESEARCH GROUP


Mauro Olivieri
Ordinary Professor


Saeid Jamili
PhD Candidate


Francesco Menichelli
Assistant Professor


Marco Angioli
PhD Candidate


Antonio Mastrandrea
Research Fellow


Andrea Marcelli
PhD Candidate


Abdallah Cheick
Research Fellow


Rocco Martino
PhD Candidate


Marcello Barbirotta
Research Fellow


Marco Pisani
PhD Candidate

The Hyperdimensional Computing paradigm for learning task

Hyperdimensional Computing (HDC) [1] is a consolidated computing paradigm that encode information through distributed high-dimensional representations called **hypervectors** (HVs).

The mathematical space where HVs are manipulated is characterized by a very small set of arithmetic vector operations:

- **Binding**
- **Bundling**
- **Permutation**
- **Similarity**

By appropriately combining these operations it is possible to perform various **learning tasks** such as classification, clustering and regression.

[1] Kanerva, P. (2009). Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation, 1*(2), 139–159. https://doi.org/10.1007/s12559-009-9009-8

The Hyperdimensional Computing paradigm for learning task

- **Computational and Energy Efficiency**
- **Scalability**
- **Resilience to Noise and Hardware Fault**
- **One-Shot Learning**
- **Extreme Parallelism**

Ideal for implementing AI algorithms on **resource-constrained** systems. A perfect candidate for **hardware acceleration.**

Why an HDC Extension to RISC-V?

Several hardware architectures have been proposed to accelerate HDC tasks. However, these designs are typically **task-specific**, making them difficult to adapt to different applications.

Common limitations include:

- No **general-purpose** hardware;
- Only accelerate **composite operations**;
- Fixed-length and fixed type **HVs**;

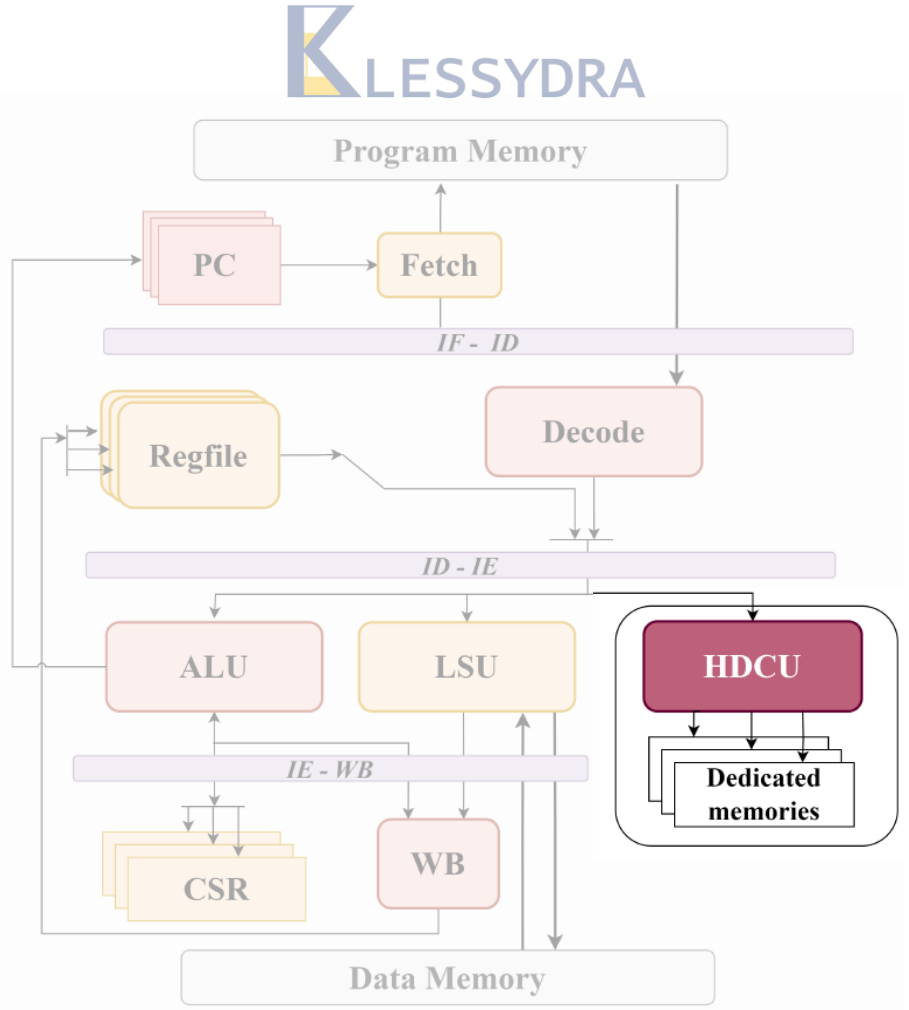
RISC-V allow us to overcome all these limitations!

Proposed solution

We designed a **highly flexible and reconfigurable hardware accelerator** to optimize the execution time of HDC learning tasks.

This architecture is integrated into the bare-metal RV32IM **Klessydra T03 core** [2], taken from a RISC-V open-source processing core family.

The HDCU accelerates all the main HDC operations and can be **configured at synthesis time**, enabling a trade-off between execution latency and hardware resource utilization.



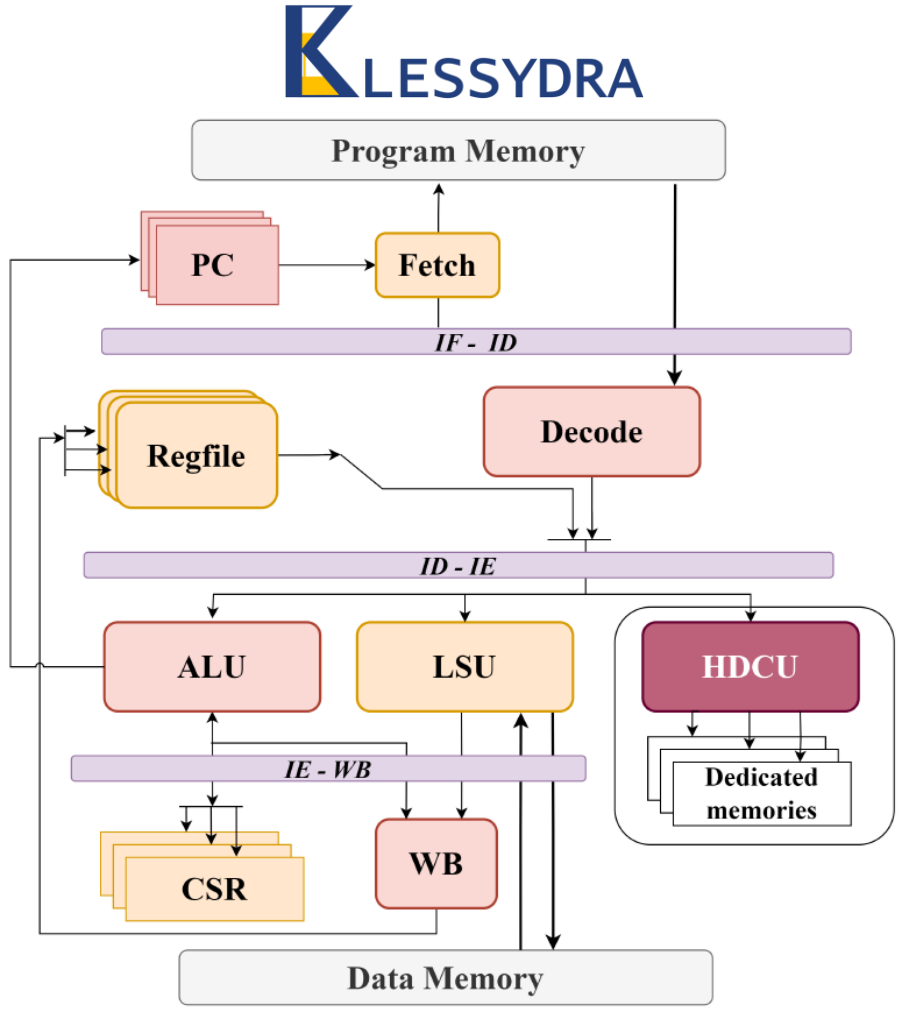
[2] A. Cheikh, S. Sordillo, A. Mastrandrea, F. Menichelli, G. Scotti and M. Olivieri, "Klessydra-T: Designing Vector Coprocessors for Multithreaded Edge-Computing Cores," in IEEE Micro, vol. 41, no. 2, pp. 64-71, March-April 2021, doi: 10.1109/MM.2021.3050962.

Proposed solution

We designed a **highly flexible and reconfigurable hardware accelerator** to optimize the execution time of HDC learning tasks.

This architecture is integrated into the bare-metal RV32IM **Klessydra T03 core** [2], taken from a RISC-V open-source processing core family.

The HDCU accelerates all the main HDC operations and can be **configured at synthesis time**, enabling a trade-off between execution latency and hardware resource utilization.



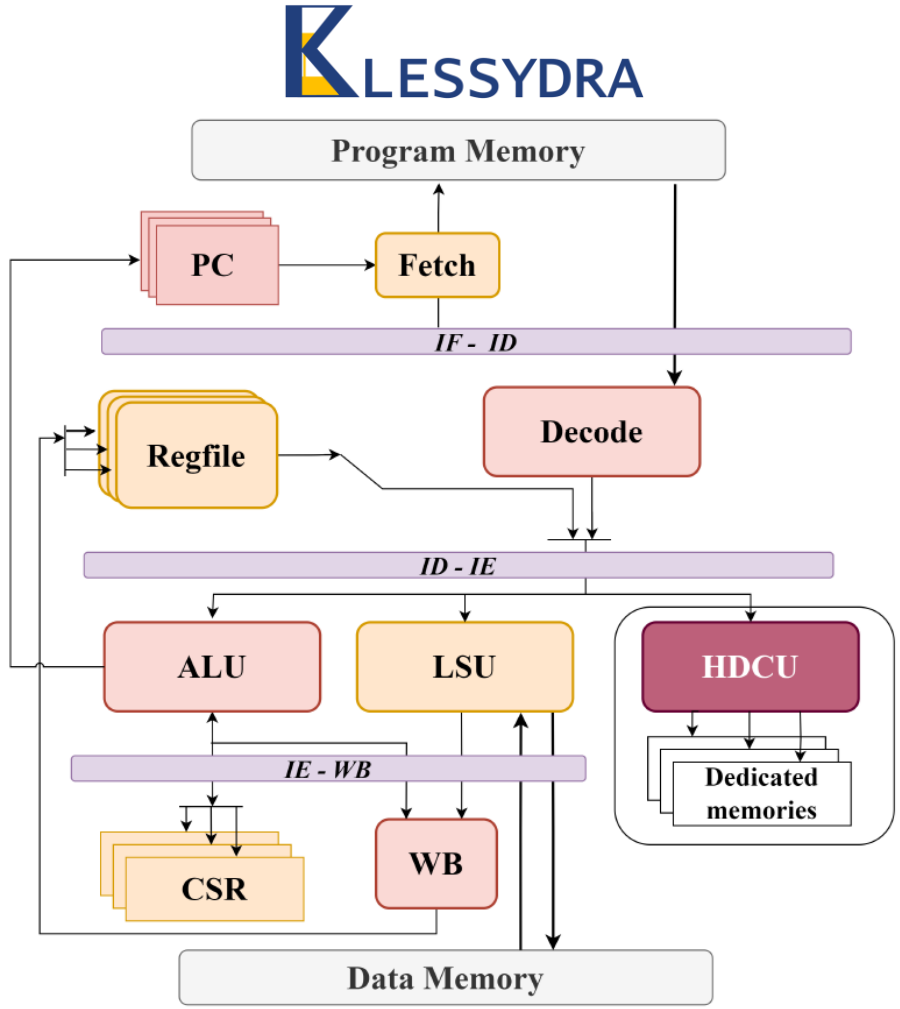
[2] A. Cheikh, S. Sordillo, A. Mastrandrea, F. Menichelli, G. Scotti and M. Olivieri, "Klessydra-T: Designing Vector Coprocessors for Multithreaded Edge-Computing Cores," in IEEE Micro, vol. 41, no. 2, pp. 64-71, March-April 2021, doi: 10.1109/MM.2021.3050962.

Proposed solution

We designed a **highly flexible and reconfigurable hardware accelerator** to optimize the execution time of HDC learning tasks.

This architecture is integrated into the bare-metal RV32IM **Klessydra T03 core** [2], taken from a RISC-V open-source processing core family.

The HDCU accelerates all the main HDC operations and can be **configured at synthesis time**, enabling a trade-off between execution latency and hardware resource utilization.



[2] A. Cheikh, S. Sordillo, A. Mastrandrea, F. Menichelli, G. Scotti and M. Olivieri, "Klessydra-T: Designing Vector Coprocessors for Multithreaded Edge-Computing Cores," in IEEE Micro, vol. 41, no. 2, pp. 64-71, March-April 2021, doi: 10.1109/MM.2021.3050962.

Hyperdimensional Computing Coprocessor

Main Features of the HDCU:

- **Custom RISC-V Instruction Set Extension**
- Specialized Functional Units
- Dedicated Local Memories
- Configurability

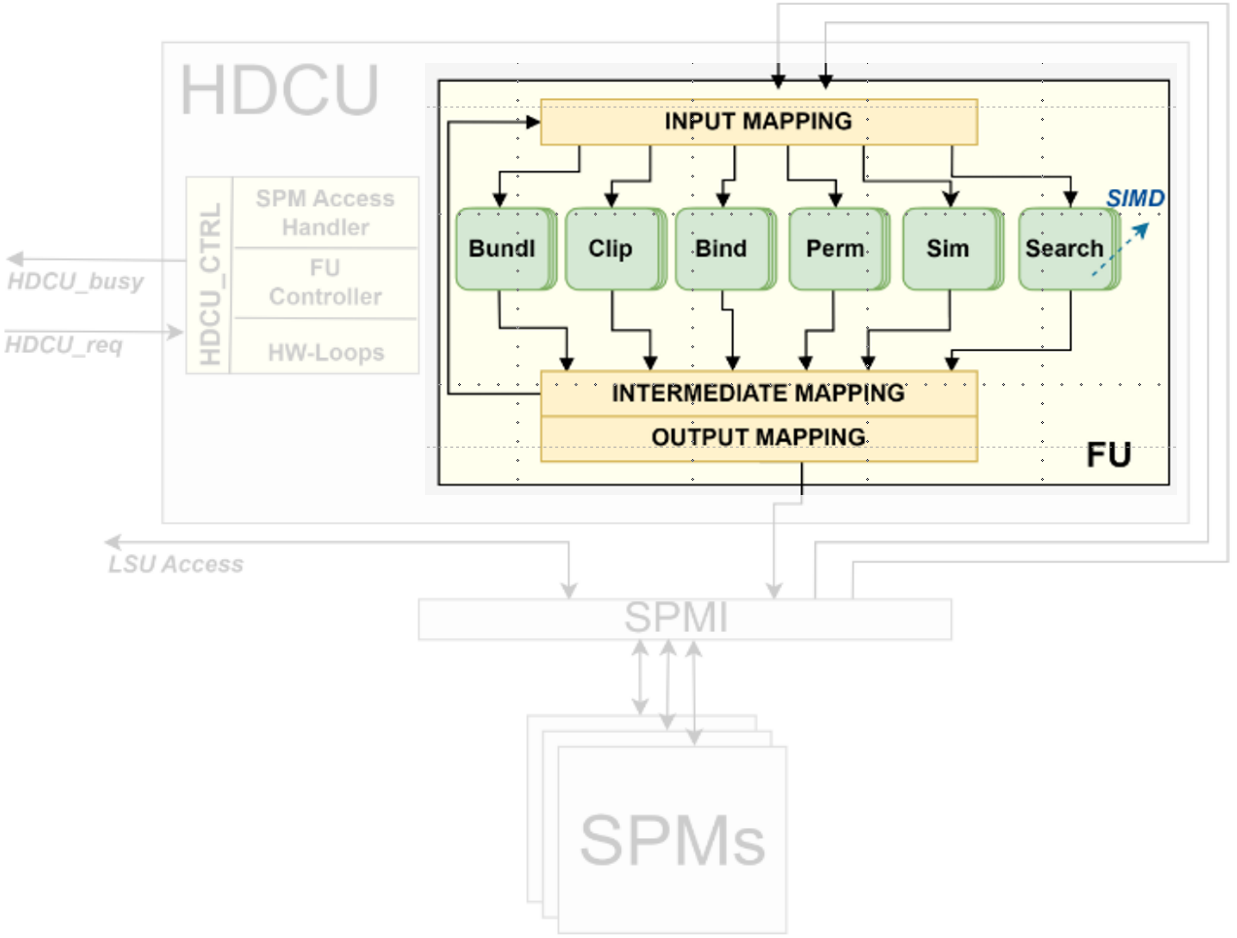
The instruction extension maps on a set of intrinsic functions in C, integrated in the GCC compilation flow.

Instruction	Description
hvbundle(void* rd, void* rs1, void* rs2)	Bundle the N-bit precision HV in `rs1` with the binary HV in `rs2` to create a new HV in `rd`.
hvbind(void* rd, void* rs1, void* rs2)	Binds the HVs in `rs1` and `rs2`, resulting in a new HV in `rd`.
hvperm(void* rd, void* rs1, int rs2)	Permute the HV in `rs1` by `rs2` positions storing the result in `rd`.
hvsim(void* rd, void* rs1, void* rs2)	Hamming distance between the HVs in `rs1` and `rs2`. The similarity is stored in `rd`.
hvclip(void* rd, void* rs1, int rs2)	Binarize the HV in `rs1` using the threshold in `rs2`. Result in `rd`.
hvsearch(void* rd, void* rs1, void* rs2)	Compare the HV in `rs1` with CSR_HVCLASS HVs in `rs2`. Stores the closest match in `rd`.
hvmemld(void* rd, void* rs1, int size)	Loads an HV from memory location `rs1` into the SPM memory location `rd`. The `size` specifies the number of bytes to load.
hvmemstr(void* rd, void* rs1, int size)	Stores an HV from the SPM memory location `rs1` into data memory location `rd`. The `size` specifies the number of bytes to store.

Hyperdimensional Computing Coprocessor

Main Features of the HDCU:

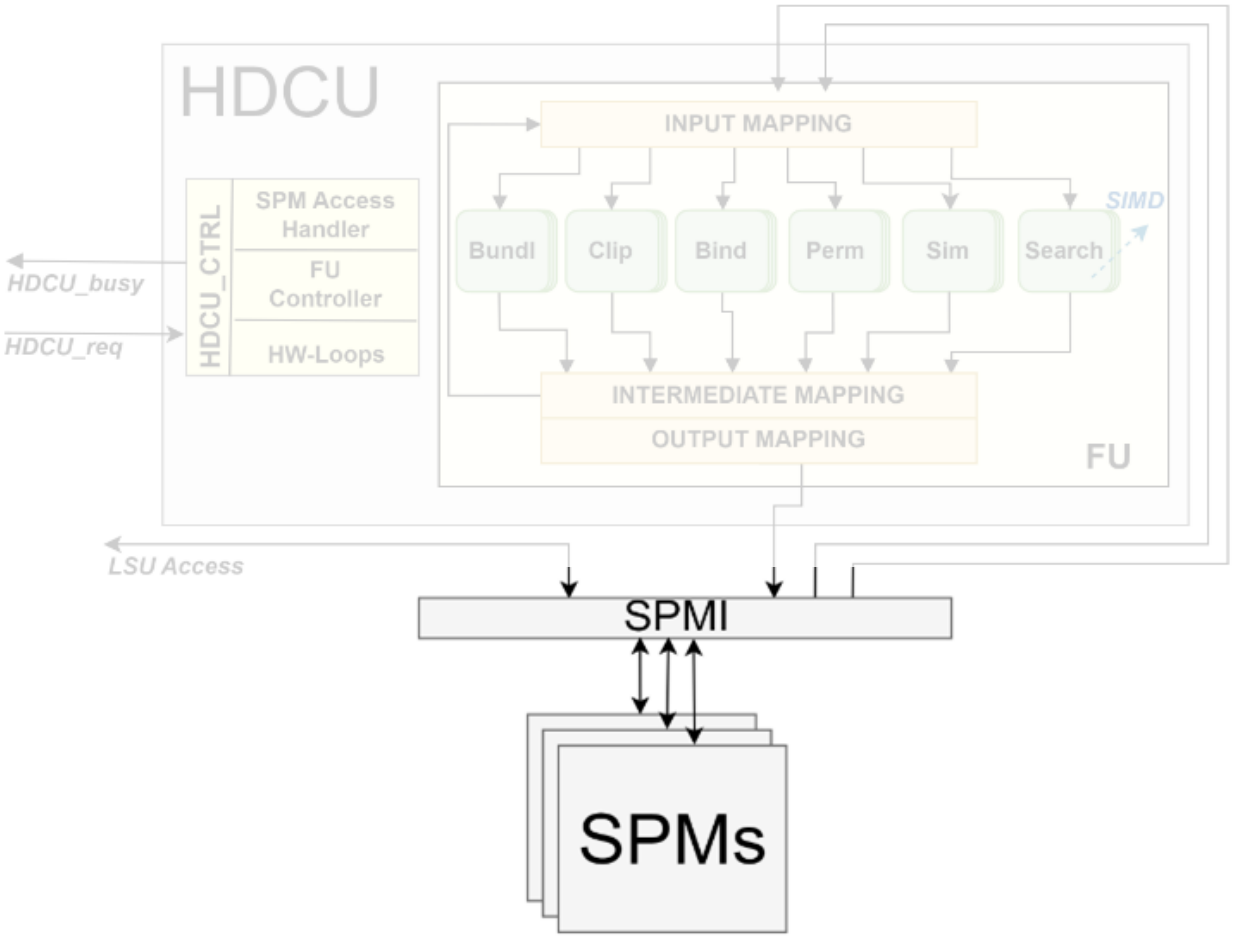
- Custom RISC-V Instruction Set Extension
- **Specialized Functional Units**
- Dedicated Local Memories
- Configurability



Hyperdimensional Computing Coprocessor

Main Features of the HDCU:

- Custom RISC-V Instruction Set Extension
- Specialized Functional Units
- **Dedicated Local Memories**
- Configurability




```
hvmemst(void* rd, void* rs1, void* rs2)

hvmemld(void* rd, void* rs1, void* rs2)
```

Hyperdimensional Computing Coprocessor

Main Features of the HDCU:

- Custom RISC-V Instruction Set Extension
- Specialized Functional Units
- Dedicated Local Memories
- **Configurability**

Parameter	Configuration Time
SPM Size	Synthesis
SPM Number	Synthesis
Hardware Parallelism (*SIMD*)	Synthesis
Functional Unit Enable/Disable	Synthesis
HVSIZE	Runtime
HVCLASS	Runtime

Hardware Results

To evaluate **the hardware requirements** of the proposed accelerator we used the **Xilinx Zynq UltraScale+ ZCU106** (EK-U1-ZCU106-G) FPGA:

- We synthesized and implemented the Klessydra-T03 core, including the designed HDCU.
- We analyzed how the hardware resource usage scales with increasing SIMD width, ranging from 32 to 1024 lanes, to evaluate the architectural flexibility of the proposed HDCU.

Device	LUTs	FF	DSPs	BRAM
Klessydra T03 Core	4281	1418	7	0
HDCU	1030	450	0	0
Scratchpad Memory Interface	384	151	0	2
Scratchpad Memory	268	0	0	2

Hyperdimensional Computing Unit				
Configuration	Synthesis Results			
SIMD	LUTs	FF	CARRY8	f_{max} [MHz]
32	1030	450	87	234
64	2040	651	101	221
128	3243	868	131	218
256	4016	1101	172	215
512	13811	3731	464	160
1024	34189	7478	1424	140

Hardware Results

To evaluate **the hardware requirements** of the proposed accelerator we used the **Xilinx Zynq UltraScale+ ZCU106** (EK-U1-ZCU106-G) FPGA:

- We synthesized and implemented the Klessydra-T03 core, including the designed HDCU.
- We analyzed how the hardware resource usage scales with increasing SIMD width, ranging from 32 to 1024 lanes, to evaluate the architectural flexibility of the proposed HDCU.

Device	LUTs	FF	DSPs	BRAM
Klessydra T03 Core	4281	1418	7	0
HDCU	1030	450	0	0
Scratchpad Memory Interface	384	151	0	2
Scratchpad Memory	268	0	0	2

Hyperdimensional Computing Unit				
Configuration	Synthesis Results			
SIMD	LUTs	FF	CARRY8	f_{max} [MHz]
32	1030	450	87	234
64	2040	651	101	221
128	3243	868	131	218
256	4016	1101	172	215
512	13811	3731	464	160
1024	34189	7478	1424	140

Hardware Results

To evaluate **the hardware requirements** of the proposed accelerator we used the **Xilinx Zynq UltraScale+ ZCU106** (EK-U1-ZCU106-G) FPGA:

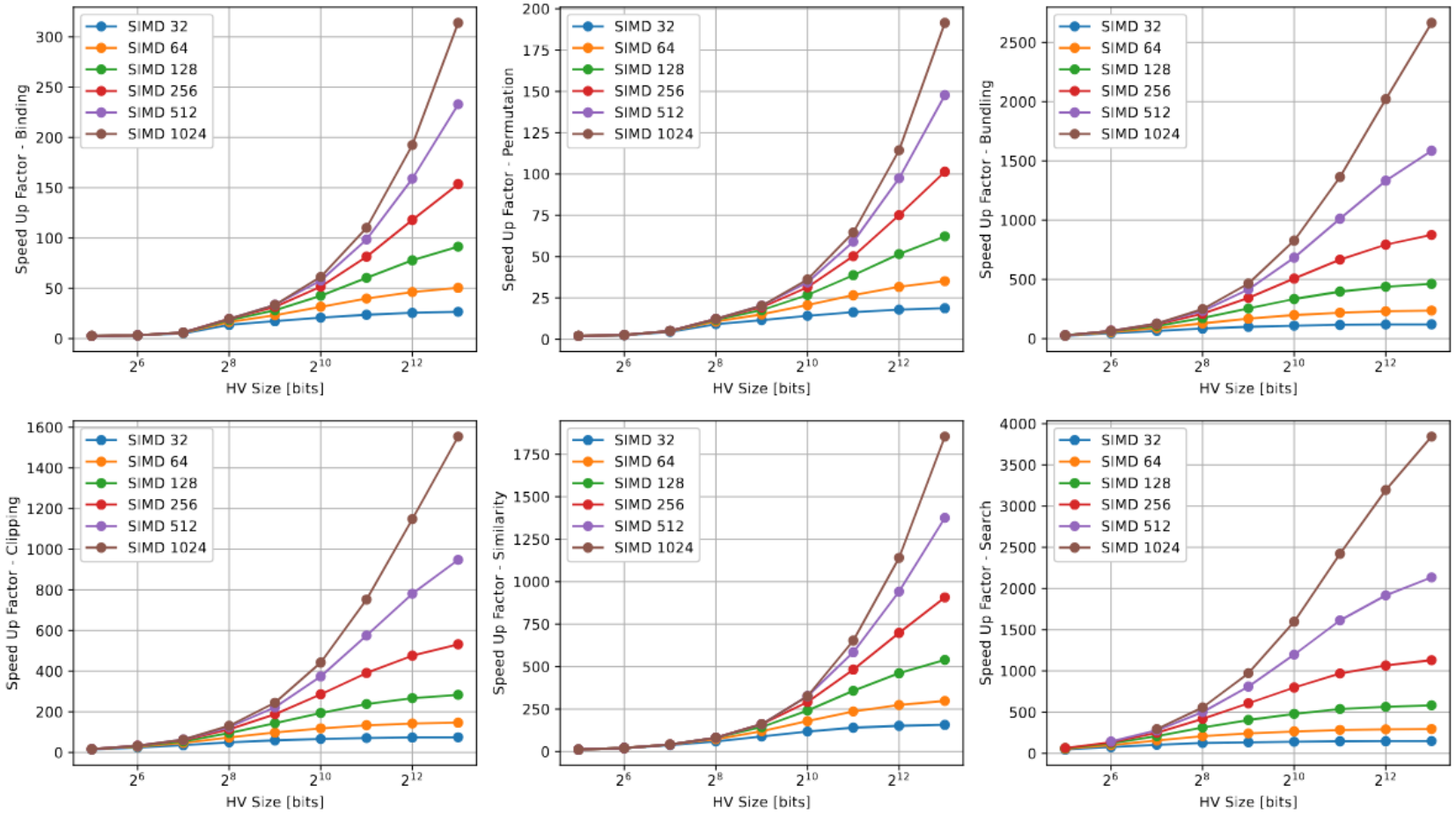
- We synthesized and implemented the Klessydra-T03 core, including the designed HDCU.
- We analyzed how the hardware resource usage scales with increasing SIMD width, ranging from 32 to 1024 lanes, to evaluate the architectural flexibility of the proposed HDCU.

Device	LUTs	FF	DSPs	BRAM
Klessydra T03 Core	4281	1418	7	0
HDCU	1030	450	0	0
Scratchpad Memory Interface	384	151	0	2
Scratchpad Memory	268	0	0	2

Hyperdimensional Computing Unit				
Configuration	Synthesis Results			
SIMD	LUTs	FF	CARRY8	f_{max} [MHz]
32	1030	450	87	234
64	2040	651	101	221
128	3243	868	131	218
256	4016	1101	172	215
512	13811	3731	464	160
1024	34189	7478	1424	140

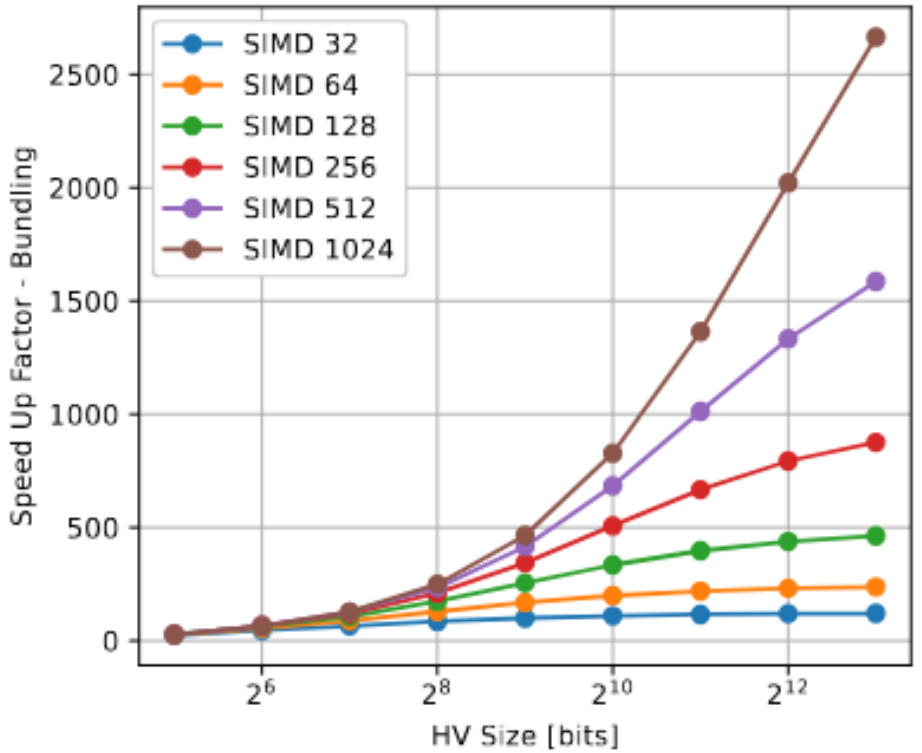
Performance Results – Speed Up Factor

- **Binding:**
  from 2.70× to 313.22×
- **Bundling:**
  from 23.36× to 2665.20×
- **Permutation:**
  from 2.06× to 191.48×
- **Clipping:**
  from 14.34× to 1554.16×
- **Similarity:**
  from 11.91× to 1854.44×
- **Associative Search:**
  from 40.21× to 3844.09×



Performance Results – Speed Up Factor

- **Binding:**
  from 2.70× to 313.22×
- **Bundling:**
  from 23.36× to 2665.20×
- **Permutation:**
  from 2.06× to 191.48×
- **Clipping:**
  from 14.34× to 1554.16×
- **Similarity:**
  from 11.91× to 1854.44×
- **Associative Search:**
  from 40.21× to 3844.09×



Conclusions

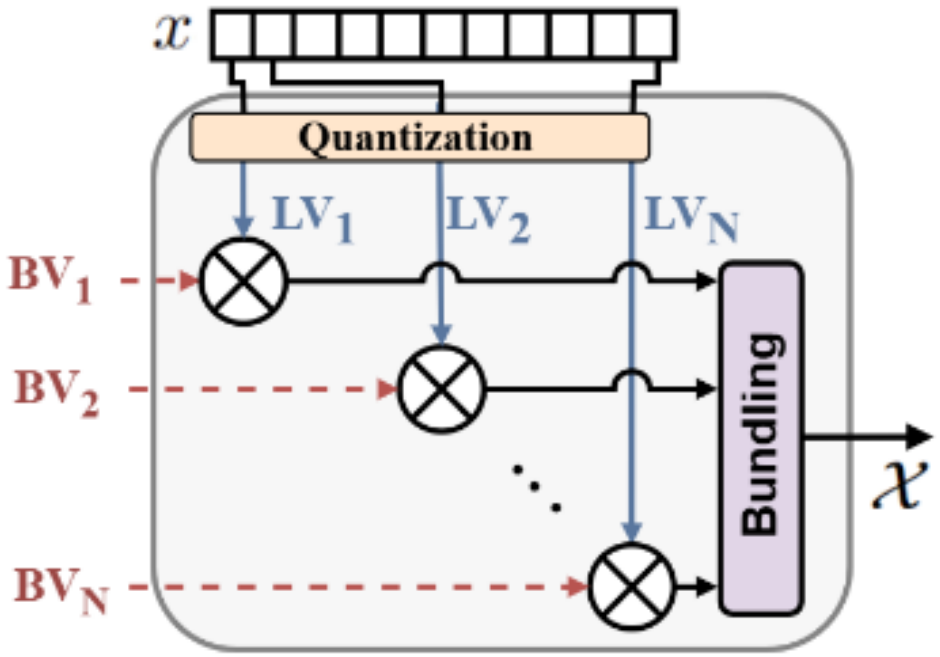
- We introduced the **HDCU**, a configurable and general-purpose coprocessor for accelerating the core operation of hyperdimensional computing paradigm.
- We extend the **RISC-V ISA** in order to give programmers an easy-to-use accelerators for speed-up they own algorithms.
- We validated our design by implementing it on an FPGA, demonstrating its **scalability** and achieving **consistent speed-up** across key HDC tasks.

Rocco Martino, Sapienza University Of Rome
rocco.martino@uniroma1.it

Find Us on **GitHub!**

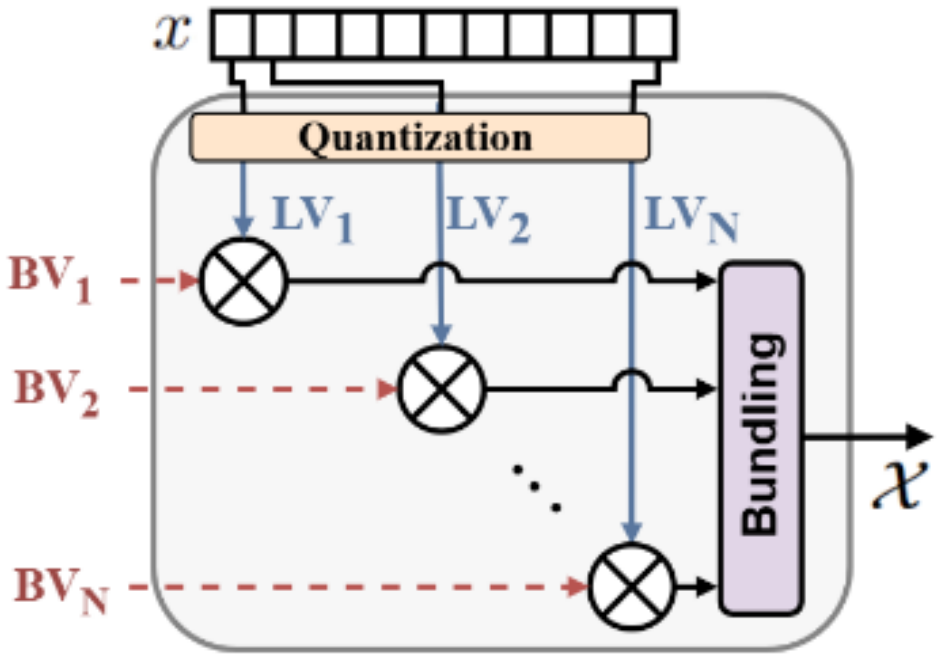
LESSYDRA

Code Example



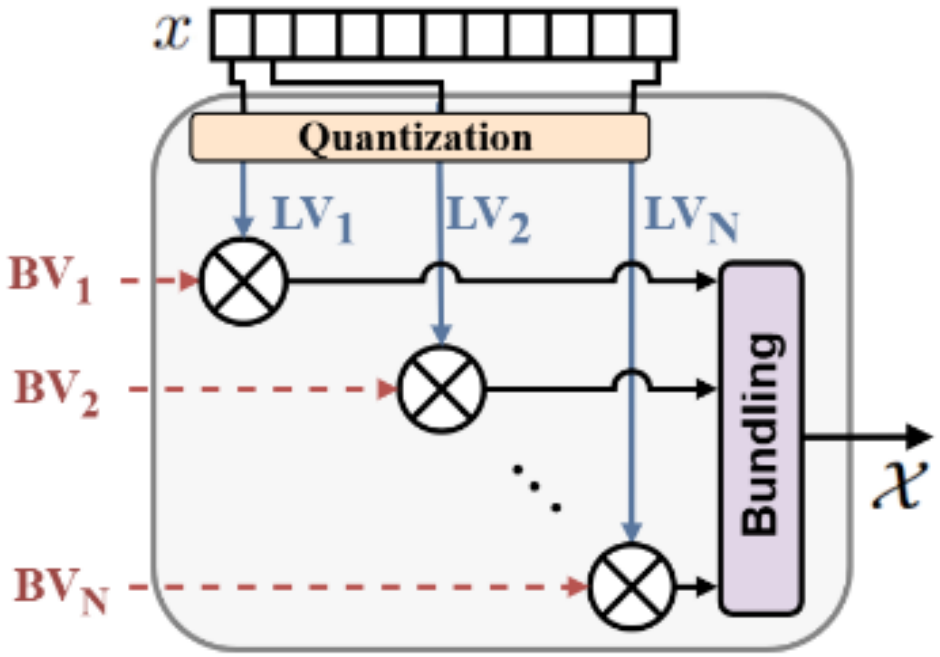

```
// ...Logic to generate the BV and LV

// Load the BV and LV into the SPMs and set the
// HVSIZE using the CSR. Perform these operations
// just once, before the first encoding
int HV_SIZE = 1024;
hvmemld((void*)((int*)spmA), &BV[0], FEATURE_NUM);
hvmemld((void*)((int*)spmB), &LV[0], LV_NUM);
CSR_HVSIZE(HV_SIZE);

// Iterative Record Based encoding
for (int i = 0; i < FEATURE_NUM; i++)
{
    // BIND the BV[i] with LV[quant_feature[i]]
    // quant_feature[i] is denoted as q[i]
    hvbind((void*)((int*)spmC +  i   * HV_SIZE),
           (void*)((int*)spmB + q[i] * HV_SIZE),
           (void*)((int*)spmA +  i   * HV_SIZE));

    // Bundle the feature-HVs
    hvbundle((void*)((int*)spmD),
             (void*)((int*)spmD),
             (void*)((int*)spmC + i * HV_SIZE));
}

// Optional: if you finished using the HDCU,
// you can store the HV in the main memory
hvmemstr(&out,(void*)((int*)spmC), sizeof(out));
```

Code Example



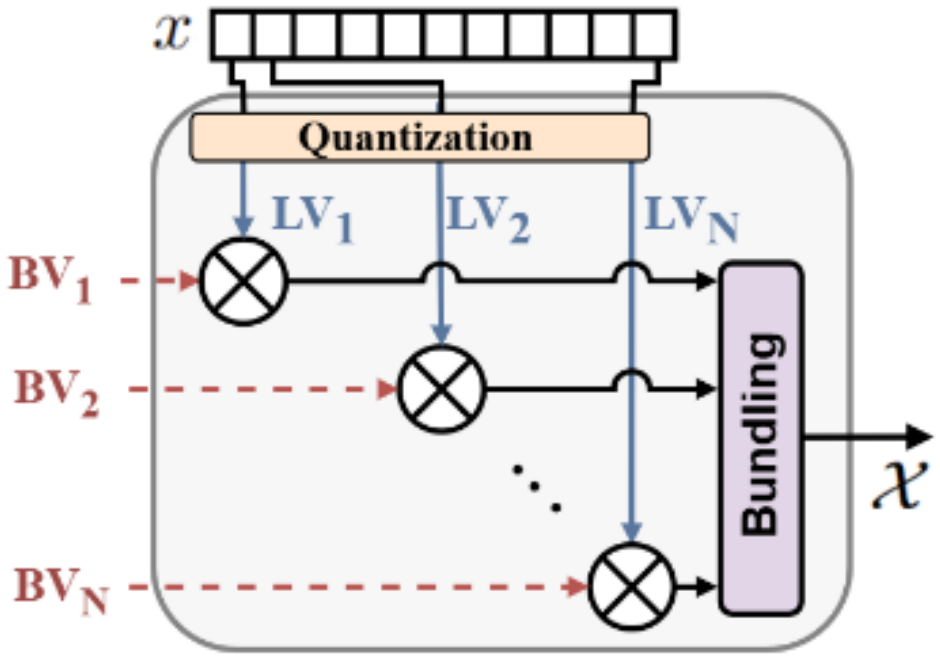

```
// ...Logic to generate the BV and LV

// Load the BV and LV into the SPMs and set the
// HVSIZE using the CSR. Perform these operations
// just once, before the first encoding
int HV_SIZE = 1024;
hvmemld((void*)((int*)spmA), &BV[0], FEATURE_NUM);
hvmemld((void*)((int*)spmB), &LV[0], LV_NUM);
CSR_HVSIZE(HV_SIZE);

// Iterative Record Based encoding
for (int i = 0; i < FEATURE_NUM; i++)
{
    // BIND the BV[i] with LV[quant_feature[i]]
    // quant_feature[i] is denoted as q[i]
    hvbind((void*)((int*)spmC +  i   * HV_SIZE),
           (void*)((int*)spmB + q[i] * HV_SIZE),
           (void*)((int*)spmA +  i   * HV_SIZE));

    // Bundle the feature-HVs
    hvbundle((void*)((int*)spmD),
             (void*)((int*)spmD),
             (void*)((int*)spmC + i * HV_SIZE));
}

// Optional: if you finished using the HDCU,
// you can store the HV in the main memory
hvmemstr(&out,(void*)((int*)spmC), sizeof(out));
```

Code Example




```
 // ...Logic to generate the BV and LV

// Load the BV and LV into the SPMs and set the
// HVSIZE using the CSR. Perform these operations
// just once, before the first encoding
int HV_SIZE = 1024;
hvmemld((void*)((int*)spmA), &BV[0], FEATURE_NUM);
hvmemld((void*)((int*)spmB), &LV[0], LV_NUM);
CSR_HVSIZE(HV_SIZE);

// Iterative Record Based encoding
for (int i = 0; i < FEATURE_NUM; i++)
{
    // BIND the BV[i] with LV[quant_feature[i]]
    // quant_feature[i] is denoted as q[i]
    hvbind((void*)((int*)spmC +  i   * HV_SIZE),
           (void*)((int*)spmB + q[i] * HV_SIZE),
           (void*)((int*)spmA +  i   * HV_SIZE));

    // Bundle the feature-HVs
    hvbundle((void*)((int*)spmD),
             (void*)((int*)spmD),
             (void*)((int*)spmC + i * HV_SIZE));
}

// Optional: if you finished using the HDCU,
// you can store the HV in the main memory
hvmemstr(&out,(void*)((int*)spmC), sizeof(out));
```

Code Example




```
// ...Logic to generate the BV and LV

// Load the BV and LV into the SPMs and set the
// HVSIZE using the CSR. Perform these operations
// just once, before the first encoding
int HV_SIZE = 1024;
hvmemld((void*)((int*)spmA), &BV[0], FEATURE_NUM);
hvmemld((void*)((int*)spmB), &LV[0], LV_NUM);
CSR_HVSIZE(HV_SIZE);

// Iterative Record Based encoding
for (int i = 0; i < FEATURE_NUM; i++)
{
    // BIND the BV[i] with LV[quant_feature[i]]
    // quant_feature[i] is denoted as q[i]
    hvbind((void*)((int*)spmC +  i   * HV_SIZE),
           (void*)((int*)spmB + q[i] * HV_SIZE),
           (void*)((int*)spmA +  i   * HV_SIZE));

    // Bundle the feature-HVs
    hvbundle((void*)((int*)spmD),
             (void*)((int*)spmD),
             (void*)((int*)spmC + i * HV_SIZE));
}

// Optional: if you finished using the HDCU,
// you can store the HV in the main memory
hvmemstr(&out,(void*)((int*)spmC), sizeof(out));
```