FPGA-Accelerated Supercomputing on the Edge for NCEM and ALS

Farzad Fatollahi-Fard, Anastasiia Butko,
Nirmalendu Patra, & John Shalf
Computational Research Division (CRD)

Yatish Kumar & Chin Guok
Energy Sciences Network (ESnet)

Peter Ercius
National Center for Electron Microscopy (NCEM)

David Shapiro
Advanced Light Source (ALS)

Chris Neely
Xilinx, Inc.

Abstract—With experimental facilities data production rates exponentially increasing, there is a greater need to move computation closer to the experimental source. To this end, we have developed the Berkeley eXtensible Processing Engine (BXPE) Framework, which allows experimentalists to utilize FPGAs stitched into the network pipeline to process data in real-time as it flows over the network. We provide a Python-tooling front-end to allow for development of algorithms using this framework. In this particular design, we ported the NCEM Center-of-Mass and the ALS convolution applications to this edge computing framework.

I. Motivation

Experimental facilities like the Advanced Light Source (ALS) and National Center for Electron Microscopy (NCEM) are experiencing double-exponential increases in data production rates from emerging detectors. The volume and rate of these increasingly large experimental dataset threatens to overwhelm both wide area networks and HPC center ingest rates, as shown in Fig. 1. For example, the 4D Camera at NCEM utilizes 48 10Gb/s interfaces for transmitting data acquired from the A/D over the link. Much like how experiments today benefit from custom sensors and associated accelerators/algorithms, future experiments will benefit from specialized compute to analyze and reduce the incoming data. Edge computing techniques offer an opportunity to reduce the load by moving some of the preprocessing closer to the detectors.

We asked our collaborators for examples of preprocessing currently in use. NCEM provided us a phase contrast electron microscopy imaging technique based on the Center-of-Mass of many detector frames, as shown in Fig. 2, as well as a sample of data obtained from their detector. ALS similarly provided us a Convolution

Fig. 1: Current HPC Centers cannot keep up with current data rates. [1]

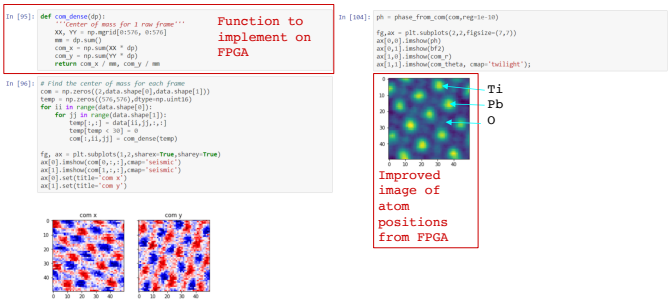

Fig. 2: Example Center-of-Mass Application Jupyter Notebook from NCEM.

application, as well as sample data. Both of these were implemented as Jupyter Notebooks, which provides a simple Python interface for their application. Since most of their development is in Python, maintaining that interface is critical for ease of use.

However, developing directly on FPGA hardware typically requires a hardware-design background, and it is outside the scope for domain scientists. There is great need to efficiently migrate algorithms developed in Python to the FPGA. Thus any developed framework must provide an easy interface for domain scientists to develop and to test their algorithms on this framework, as shown in Fig. 3.

We developed the Berkeley eXtensible Processing Engine (BXPE) Framework, an FPGA-based edge network processing pipeline for real-time data processing. Based on the Xilinx Alveo U250/U280 FPGA-based accelerator cards [2, 3], this edge computing framework provides two 100Gb/s Ethernet connections as well as up to ~10,000 DSP primitives. To provide access to these resources, we developed a Python-based tooling to allow users to port their algorithms to this framework with a software-based simulator and compiler, as well as configuring the FPGA-based hardware array. Using a co-design methodology, we developed a DSP processing engine with the BXPE framework based on the NCEM Center-of-Mass and ALS convolution applications.

II. State-of-the-Art FPGA Infrastructure

As previously mentioned, we are working closely with Xilinx to bring up the FPGA architecture needed for this project. Using SuperMicro compute boxes, we built a number of testbeds to test our FPGA designs, the chief of which being a server with up to eight U250/U280 FPGA cards. We currently have a number of these servers set up at ESnet, CRD (soon to move to NERSC), NCEM, and ALS. Utilizing the existing high-speed network links in the lab, we are able to closely couple our processing pipeline with the experimental pipeline. Fig. 4 diagrams the deployed hardware and the network nodes.

III. BXPE DSP Framework Description

A. Network System Architecture

In order to effectively utilize the resources on the FPGAs, we make use of the ESnet and Xilinx jointly developed OpenNIC Shell [4]. The OpenNIC Shell integrates the Network Processing Unit (NPU) with the two 100GbE interfaces on each FPGA card, allowing the FPGA to read Ethernet packets. In addition to providing the network interface, the OpenNIC Shell also provides a host sideband interface over PCIe for controlling the design contained within the User Logic Block. A detailed block diagram of the contents of OpenNIC Shell is shown in Fig. 5. This interface allows the user to read various control registers within the design, as well as to load static data and programs for the array of processors. The shell provides standard AXI4-Stream and AXI4-Lite interfaces for these connections, allowing for a modular design methodology. An example of how we used OpenNIC Shell is shown in Fig. 6.

This modular nature allows us to scale these processing arrays across multiple FPGA cards as well, allowing us to install the same array across multiple FPGAs.

B. Python Tooling Frontend

To facilitate easier use of the BXPE DSP Array, we developed an extensive Python-based Tooling Frontend

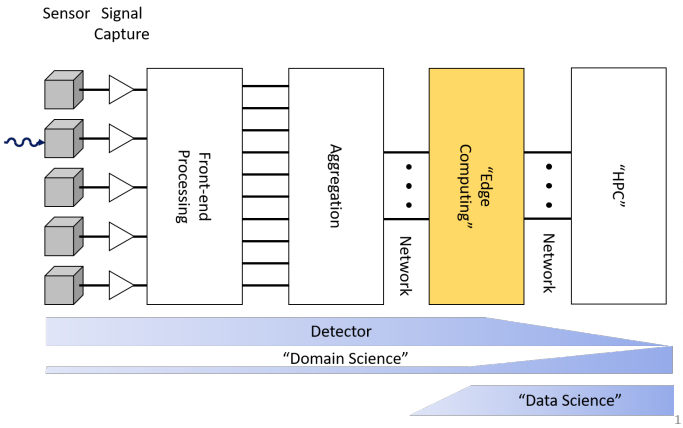

Fig. 3: The edge is part of a continuum of computing workflows. [1]

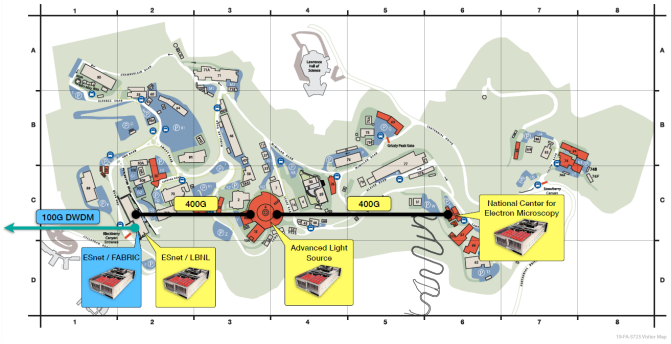

Fig. 4: Network Infrastructure of LBL FPGA deployment

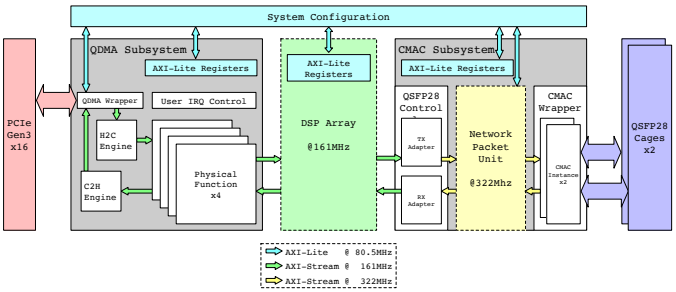

Fig. 5: Block diagram of the modified OpenNIC Shell

which provides three main components: (a) a functionally accurate simulator, (b) a compiler, and (c) a software frontend for the array.

1) Simulator & Compiler: Since the programming model of the array is very different from standard HPC systems, it was critical to build a simulator to easily demonstrate the architecture. As a result, we developed a functionally accurate Python simulator of the FPGA array. We analyzed the computation in each example application and ported them to this system using this simulator, comparing the simulators results with HPC results to ensure correctness. Being written in Python, we could demonstrate the use of the array in Jupyter Notebooks.

The DSP programs for the example applications are generated with Python functions. The multiply-accumulate DSP program is shown in Listing 1, which is the central part of the Center-Of-Mass algorithm. The convolution program is shown in Listing 2, which uses the multiply-accumulate function to generate the required instructions.

```
def macc_program(shape):
    prog = list()
    for j in range(shape[1]):
        for i in range(shape[0]):
            if (i, j) == (0, 0):
                prog.append(("MULT", (0, 0)))
            else:
                prog.append(("MACC", (i, j)))
    prog.append(("SAVE", (0, 0)))
    prog.append(("NOOP", (0, 0)))
    return prog
```

Listing 1: Multiply-Accumulate DSP Program Generator

```python
def convolution_program(rows_per_thread, image_len, ker_shape, bottom_threshold, top_threshold):
    prog = list()
    prog.append(("CLIP", 1))
    prog.append(("BTHS", bottom_threshold))
    prog.append(("TTHS", top_threshold))
    prog.append(("BPTR", (image_len, rows_per_thread)))
    prog += macc_program(ker_shape)
    return prog
```

Listing 2: Convolution DSP Program Generator

Once satisfied that the algorithm was working correctly on the simulator, we then compile the program and export the kernels for the given computation to be loaded on the FPGA array. We used this frontend to accelerate the development of the Center-of-Mass application for NCEM and a Convolution application for ALS. For these applications, we converted the data from floating point to fixed point, and compared the simulation results against the baseline Jupyter Notebook application, guaranteeing correctness of the simulator against the real application.

2) FPGA Frontend: When the FPGA is loaded with the DSP Array design, the program and kernel memories must be loaded in the array. This frontend allows users to load each tile of the array with the program to be run and corresponding the kernel. Once loaded, the user can then trigger the start of computation, and upon computation completion, the output memories of each tile are read back out. This data result is then compared against the simulator result, to again guarantee correctness of the computation against the Jupyter Notebook.

3) Packet Generation: Using the provided sample data, we generated PCAP files that sent the sample data over a 100GbE interface on a host to the FPGA. In the NCEM case, their data consists of a series of $576px \times 576px$ frames. As a result, the generated packets correspond to a full $576px \times 576px$ frame, as though it were in series with the detector.

C. Proof-of-Concept DSP Tile and Array

1) DSP Array Architecture: Housed inside the OpenNIC Shell, the architecture of the array consists of three main modules: the packet buffers, pixel addressable sequencers, and the DSP Tile itself, as shown in Fig. 6.

Sensor data are transmitted as a segmented series of network packets. Incoming packets get parsed to determine the relevant header information including network addresses and the description of the payload. For example, the payload description includes a frame number and a sequence number for the data. These incoming payloads get stored in a packet buffer using a number of Xilinx URAM memory primitives. On the ingress side, a full NCEM frame (5.0625Mb for each $576px \times 576px$ frame) is striped over nine banks of the packet buffer. Once a complete image frame has been buffered, the system then triggers the sequencer stage.

The pixel addressable sequencer is a programmable switch that reads individual pixels from an input memory and writes them to a given pixel addressable output memory. This allows us to pre-program read/write address patterns for the particular algorithm. For the first stage of the array pipeline, we exploit spatial parallelism for the computation. We assigned ranges of DSPs to 2D slices of the frame data, as well as a square raster of pixels assigned to each thread in the DSP, shown in Fig. 8. In this diagram the packet buffer has been simplified for illustration purposes to show coarse grain slices of the image frames.

For the NCEM Center-of-Mass application, we partition and distribute a full NCEM frame across 36 DSP Tiles; this is done to distribute the Center-of-Mass computation across the many DSP elements on the FPGA. To implement the parallel data movement between buffers and the DSP processing stages, we created modules called sequencers that move pixels and partial computation results between (a) buffers and DSP tiles and (b) between DSP tile stages, shown in Fig. 6. The pixel addressable sequencer partitions the full frame into $96px \times 96px$

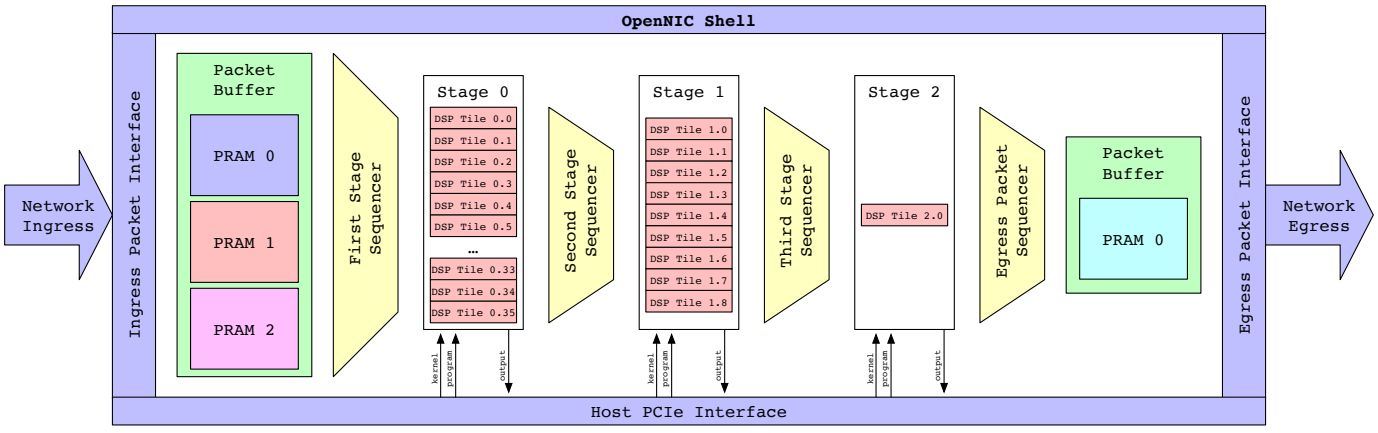

Fig. 6: Functional Diagram of DSP Array in an FPGA

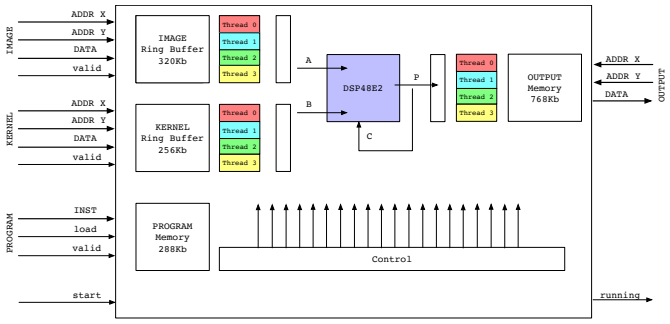

Fig. 7: Block Diagram of the DSP Tile.

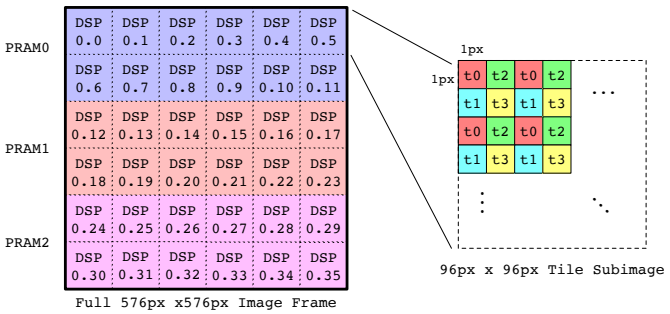

Fig. 8: Graphical demonstration of the NCEM image is buffered into 3 memories, then read out in the colored sequence to each tile thread.

sequential subframes and distributes them across the first stage of DSP tiles. These subframes are further distributed in a 2×2 raster across each thread of the tile, corresponding to $4 \times 48px \times 48px$ subsubframes. Fig. 8 describes how a frame is partitioned and distributed in the first stage.

Given that the Center-of-Mass is a reduction algorithm, each stage further reduces the data until in the final stage only a single tile remains. The final result of one frame calculation is contained in the output memory, which is buffered by the last sequencer into a packet buffer to be sent.

For the ALS Convolution application, the image size was smaller ($192px \times 484px$). Since the application treats this image as a series of 192 independent 1D vectors $484px$ long, we repack the 1D arrays to utilize 24 of the first stage DSPs. Each DSPs thread is loaded sequentially with 2 rows of the original image. The DSPs write an equally sized vector in it's output memory, which is read out by the host sideband interface.

2) DSP Tile Architecture: For our proof-of-concept design, we built a compact architecture centered on the Xilinx DSP48E2 [5] primitive on the FPGA. A block diagram of the design is shown in Fig. 7. Our development target is the Xilinx Alveo U250/U280 Acceleration Cards which provide up to ~10,000 DSP primitives. To maximize the three cycle pipeline of the DSP primitive, each tile has four independent threads operating on its ring buffers. There are three data ring buffers: Image, Kernel, and

TABLE I: DSP Tile Instructions

OPCODE Inst[23:20]	Immediate Inst[19:0]		
NOOP $4'h0$	Reserved $20b$		
SAVE $4'h1$	Reserved $8b$	X Address $6b$	Y Address $6b$
BPTR $4'h2$	Reserved $8b$	X Iterations $6b$	Y Iterations $6b$
MULT $4'h3$	Reserved $8b$	X Address $6b$	Y Address $6b$
MACC $4'h4$	Reserved $8b$	X Address $6b$	Y Address $6b$
BTHS $4'h5$	Bottom Threshold $20b$		
TTHS $4'h6$	Top Threshold $20b$		
CLIP $4'h7$	Reserved $19b$		True/False $1b$

TABLE II: OPCODE Description

NOOP	No Operation
SAVE	Save DSP Output to address (X, Y)
BPTR	Loop the loaded program by iterating the base pointer of the ring buffers in the given direction (required for convolution)
MULT	Multiply the Image and Kernel at the address (X, Y) (From Fig 7, $P = A \times B$)
MACC	Multiply the Image and Kernel at the address (X, Y), and accumulate with the previous result (From Fig 7, $P = A \times B + C$)
BTHS	Bottom threshold for Image values going into the DSP (Default: $20'b0$)
TTHS	Top threshold for Image values going into the DSP (Default: $20'h7FFF$)
CLIP	Clip the Image values to the threshold values

Output. All three memories are addressed by pixel location in a $64px \times 64px$ processed frame, which corresponds to the address being processed by each thread. The Image Ring Buffer supports 20-bit pixels, and when multiplied by the 4 threads, the total size is 320Kb. The Kernel Ring Buffer supports 16-bit pixels for a total size of 256Kb. The Output Ring Buffer has a 48-bit pixel width for a size of 768Kb. The control of the DSP is handled with instructions loaded into the Program memory. The instruction format is a 4-bit opcode followed by a 20-bit immediate. A detailed list of the instructions and their description is in Tab. I and Tab. II, respectively.

3) FPGA Layout: Fig. 9 shows the full system design run through Xilinx Vivado tools.

IV. Future Work

It should be noted that the design described here is a proof-of-concept design. This design was built for compact placement on the FPGA and has not been optimized for processing speed. Further architecture exploration would provide a more optimum design for different applications.

To address the flexibility requirement of a future architecture and extend the design for high-level operation parallelism, we consider so-called MOVE or Transport Triggered Architecture (TTA) [6]. The key concept of TTA is that it has only one type of operation, which

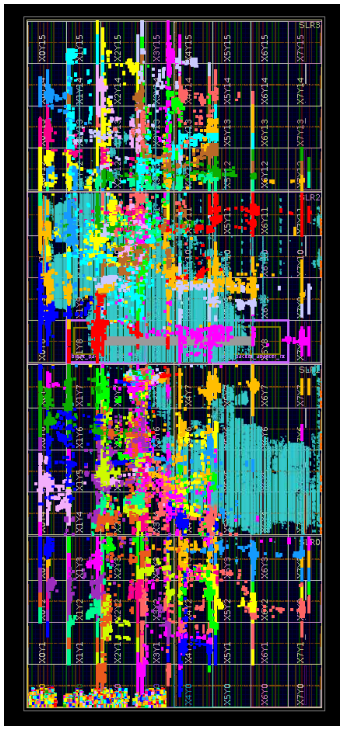

Fig. 9: FPGA Layout of 46 DSP Tile Array, with each tile color coded.

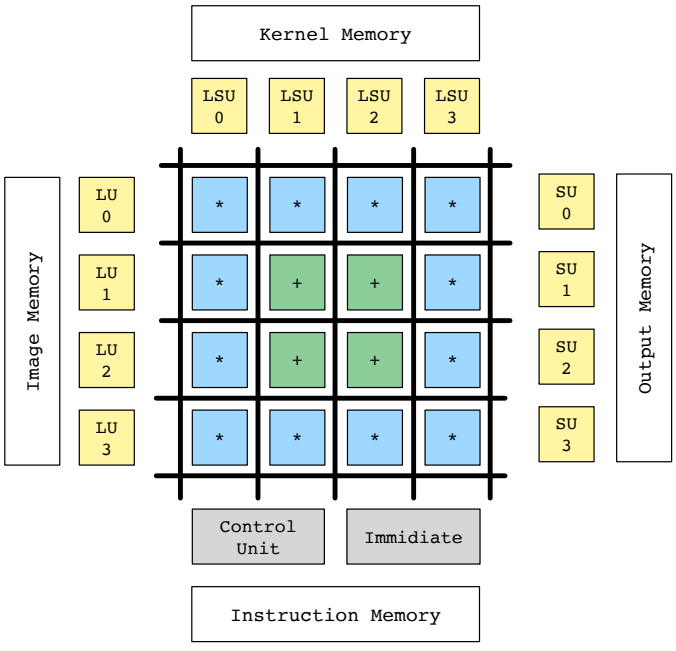

Fig. 10: Potential Transport Triggered Architecture (TTA) Exploration

transfers data from one Functional Unit (FU) to another triggering the computations on that data implicitly. The architecture can have a large number of simple functional units with different operations connected by a network-in-core. Here are the major characteristics of the TTA that make it a promising candidate for the DSP tile architecture exploration.

- The control over the data movement falls onto the compiler that schedules the data transfers with respect to dependencies and efficient resource utilization. That creates a straightforward interface in between the data-flow graph that represents the application and the architecture execution flow. With enough variability in operation types and a large number of FU's, such a DSP tile can be easily used for a large variety of DSP applications using the compiler to re-schedule the data movement.
- Since the instruction set is composed of a single type of operation, it has the potential to implement the VLIW approach. It is crucial to keep the DSP functional unit usage at a high rate to provide high throughput for the data processing.
- While the TTA can be seen as relatively simple compared to the traditional scalar and vector architectures, it heavily relies on the compiler and functional unit interconnect. The interconnect topology and reconfigurability will be a crucial components to make the design practically useful. Given the specifics of the FPGA technology, there is a potential for architecture research in this direction.
- There are previous works that explore TTA for real-time DSP applications [7, 8] that indicate it is a promising architecture concept.

An example of the DSP tile as a TTA is shown in Fig. 10. Instead of having a tile that can do multiple operations, we hard-code each tile to perform a specific operation (i.e. MULT, MACC, CLIP, etc.). The data is then read out from each corresponding memory and streamed through the interconnect among the tiles. This interconnect is configured for the desired application, filtering the data to correct tile. The resultant output would then be written to an output memory. It reduces the number of memory blocks within each tile, which has been a bottleneck for the current design. It also utilizes more of the DSP primitives on the FPGA.

V. Conclusion

In the project, we set out to create an edge-based FPGA framework to accelerate the sample applications of Center-of-Mass and Convolution. We have a scalable network shell called OpenNIC Shell, which provides the basics for connecting an FPGA to a network. We then build the BXPE DSP Array which performs compute for the given applications and a Python interface to control the engines. While this is a proof-of-concept and not very performant,

we would like to explore more performance driven designs, like transport triggered architectures.

References

[1] P. Denes, "Ascr co-design workshop," march 2021.
[2] Xilinx, "Alveo u250 data center accelerator card," 2021. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u250.html
[3] ——, "Alveo u280 data center accelerator card," 2021. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html
[4] ——, "Xilinx/open-nic," 2021. [Online]. Available: https://github.com/Xilinx/open-nic
[5] ——, UltraScale Architecture DSP Slice, Xilinx, August 2021, uG579 (v1.11). [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf
[6] P. Jääskeläinen, A. Tervo, G. Payá Vayá, T. Viitanen, N. Behmann, J. Takala, and H. Blume, "Transport-triggered soft cores," in 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2018, pp. 83–90.
[7] A. Bhagyanath and K. Schneider, "Tta as predictable architecture for real-time applications," in 2014 International Conference on Science Engineering and Management Research (ICSEMR), 2014, pp. 1–9.
[8] J. Heikkinen, J. Takala, A. Cilio, and H. Corporaal, "On efficiency of transport triggered architectures in dsp applications."