NeuroPIM: Felxible Neural Accelerator for Processing-in-Memory Architectures

Ali Monavari Bidgoli[1], Sepideh Fattahi[1], Seyyed Hossein Seyyedaghaei Rezaei[1], Mehdi Modarressi[1,2,] Masoud Daneshtalab[3,4]

[1]School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Iran
[2]School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Iran
[3]Tallinn University of Technology, Tallinn, Estonia
[4] Mälardalen University , Sweden
{a.monavari, fattahi.sepide, seyedaghaei, modarressi}@ut.ac.ir, masoud.daneshtalab@{mdu.se, taltech.ee}

Abstract— **The performance of microprocessors under many modern workloads is mainly limited by the off-chip memory bandwidth. The emerging process-in-memory paradigm present a unique opportunity to reduce data movement overheads by moving computation closer to memory. State-of-the-art processing-in-memory proposals stack a logic layer on top of one or multiple memory layers in a 3D fashion and leverage the logic layer to build near-memory processing units. Such processing units are either application-specific accelerators or general-purpose cores. In this paper, we present NeuroPIM, a new processing-in-memory architecture that uses a neural network as the memory-side general-purpose accelerator. This design is mainly motivated by the observation that in many real-world applications, some program regions, or even the entire program, can be replaced by a neural network that is learned to approximate the program's output. NeuroPIM benefits from both the flexibility of general-purpose processors and superior performance of application-specific accelerators. Experimental results show that NeuroPIM provides up to 41% speedup over a processor-side neural network accelerator and up to 8x speedup over a general-purpose processor**.

Keywords— Processing-in-memory, Hardware acceleration, Neural network.

I. INTRODUCTION

Processing-in-memory is generally considered as the most promising way to address the off-chip bandwidth limitations. This emerging technology enables tight integration of computation and memory by stacking a logic die on top of one or multiple memory dies in a 3D fashion [1]. Eliminating the need to get data via low-speed and power-hungry on-board wires, the computation implemented on the logic die benefits from unprecedented high internal bandwidth provided by vertical links, effectively pushing the bandwidth wall in traditional architectures.

Recent PIM proposals leverage the in-memory logic layer to implement different types of processing architectures. The architectures include simple application-specific accelerators [2][3], general-purpose low-power cores [4], and in-memory instructions [5].

Application-specific accelerator cores, with the datapath customized for the specific processing requirements and dataflow of a target application, provide superior performance and power-efficiency for that application. However, these fixed-function cores are not flexible (programmable) and the logic layer in a PIM system have to include a separate logic for each individual accelerated function. General-purpose cores and in-memory instructions, on the other hand, provide the flexibility of a conventional processor, but at the price of lower average performance and higher power consumption. The latter is mainly due to the associated software overheads (instruction fetch and decode, per-instruction datapath configuration) that dominate the power consumption in many processors [6].

To address this performance/flexibility challenge, this paper present NeuroPIM, a new processing-in-memory architecture that adopts neural network-based compute units as the memory-side accelerator.

Among different applications of neural networks, their capability to approximate function output has attracted considerable attention in the computer architecture community [7][8]. This capability suggests neural networks as a universal replacement for complex functions: a moderately-sized neural network can be trained to approximate a complex function and replace the original code [7]. It is likely for the execution time of the neural network on software to be comparable or even longer than the original code it mimics. However, running the neural network on a dedicated hardware unit, which leverages the abundant parallelism in the neural network dataflow, would boost the execution speed by tens to hundreds of times. Further, whereas software tools use floating-point representation for neural network weights and input data, the inherent error tolerance of neural networks allow quantizing these parameters to 8-bit or narrower fixed-point numbers with no accuracy loss [9]. This way, the neural equivalent of a function would run considerably faster than the case when the software code of that function runs on a general-purpose processor. Moreover, flexibility is the key benefit of NeuroPIM over the application-specific accelerators: a single neural logic can implement every function by just loading the weights related to that function.

Due to the approximate nature of neural networks, the generated results are not the exact result that the original function would produce. Rather, the output is an (often accurate-enough) approximation of the original result. Thus, the proposed architecture is beneficial to the complex functions that (1) need high memory bandwidth, in that they make abundant memory accesses, and (2) can tolerate some inaccuracy in their output. Fortunately, this covers a wide range of applications in many domains, including signal processing, multimedia, data analytics, and scientific computing, to name a few. By implementing dedicated neural network hardware at the logic layer of 3D memory architectures, a wide range of functions can benefit from both adaptable acceleration of neural networks and abundant memory-side bandwidth. Nonetheless, NeuroPIM can coexist with other in-memory compute units: a general-purpose memory-side processor, for

979-8-3503-3277-3/23/$31.00 ©2023 IEEE

example, would run the memory-intensive functions that are not too complex to justify neural acceleration and NeuroPIM accelerates memory-intensive complex functions.

While this work is not the first attempt to use neural networks as a general-purpose accelerator, it is the first to evaluate tight integration of neural-based accelerators within the logic layer of a 3D memory to make a flexibility-performance trade-off for PIM accelerator designs.

The rest of the paper is organized as follows. Section 2 covers important background. Section 3 describes the proposed PIM-based design. Section 4 outlines the implementation and evaluation methodology and experimental results. Finally, Section 5 concludes the paper.

II. BACKGROUND

A. Processing in memory

Several 3D-stacked DRAM implementations, such as High-Bandwidth Memory (HBM) [10] and Hybrid Memory Cube (HMC) [11], stack a logic layer, that interacts with both the host processor and the memory, on top of several DRAM layers. NeuroPIM uses the HMC architecture as the base, but it can be easily ported to the HBM architecture, as well.

In HMC, the memory device can have up to 8 DRAM layers for a total of 8GB capacity. Each DRAM layer, as well as the logic layer, is divided into 32 partitions and the vertically adjacent partitions (that form a column of vertically stacked partitions) are called a vault. Each vault has its own DRAM controller, or vault controller, implemented on its logic die. Vaults act as independent memory channels and can be accessed simultaneously. This way, each vault can roughly be considered as a DRAM channel, effectively providing a high level of parallelism inside the memory.

HMC is connected to the host processor through up to four 16-bit full-duplex packet-switched serial links, which provide up to 240 GB/s bandwidth. The logic layer provides a crossbar switch that connects the links to vault controllers.

Each vault provides 10 GB/s bandwidth between the DRAM and logic layer, for a total of 320 GB/s bandwidth for the entire logic layer. Thus, since up to half of the 240 GB/s off-chip bandwidth goes for packet-switching overhead, the internal bandwidth is much higher than the external bandwidth received by the host processor. Existing PIM research proposals augment this logic layer with compute units to exploit the high available bandwidth. This way, memory-intensive parts of a program are offloaded to the memory-side cores, leaving compute-intensive parts or those parts that exhibit high locality (which benefit from cache's high bandwidth at the processor side) to the host processor to maximize performance.

B. Neural acceleration

In this work, we employ the feedforward Multi-Layer Perceptron (MLP) model to replace the original function at run-time. The structure of an MLP is shown in Figure 1. Prior research shows that integrating CPUs and GPUs with a neural network unit as a general-purpose trainable accelerator for complex functions leads to considerable power/performance benefits [7][8]. Neural networks are approximate models in nature: unlike conventional computation models, neural networks do not produce exact results.

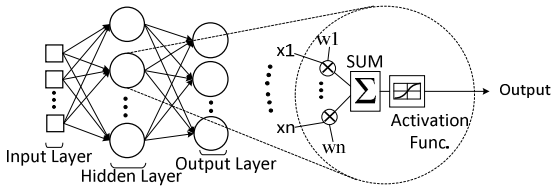

Figure 1. A 3-layer feedforward neural network

Rather, they make an approximation of what the function would generate. Therefore, they are applicable to functions that can tolerate some (small) accuracy loss at the application-level.

Once the right function(s) is selected by the programmer, the first step is to train a neural network to learn the behavior of the function and then replace the function with an invocation of the memory-side neural network accelerator. During the training, the neural network topology (the number of layers and the number of neurons per layer) and the neuron weights are calculated. The information of the equivalent neural network corresponding to each function is kept in memory. Once the function is called, the corresponding neural network parameters are loaded onto the memory-side neural network hardware to process the input and generate the output.

Finding the code regions or functions to accelerate can be done automatically by compiler or manually by programmer. This paper focuses on the architectural aspects of memory-side neural acceleration and is not restricted to a specific function selection mechanism.

III. NEUROPIM ARCHITECTURE

Figure 2 illustrates the architecture of the proposed PIM design. We equip each vault controller at the logic layer of a baseline HMC-like 3D memory with a neural network functional unit (NFU) that is specially designed to run neural networks.

NFUs at different vaults can be invoked and executed concurrently. Each neural network is mapped onto a single NFU and just communicates with the DRAM banks of its vault to read/write data. Thus, unlike some prior work [2] that implement large neural networks that span across multiple vaults, each NFU (and its vault) is independent and there is no need for inter-PE and inter-vault communication to exchange data or intermediate results. Consequently, there is no need for inter-vault network at the logic layer.

A. NFU architecture

NFU consists of a specialized neural network datapath and a sequencer unit (SeqU). SeqU schedules the execution of neural network neurons on the functional units and generates the sequence of addresses required by NFU to coordinate data movement between vault controller and the datapath. In order to generate the right sequence of memory addresses, SeqU is programmed by the host processor.

To this end, a special programming packet should be added to the host-memory communication protocol. This packet contains the ID of the target vault, the address of the locations where the corresponding neural network weights and input data are stored, and the neural network structure. Simply stated, the overall execution scenario of NFU is as follows. (1) Host processor sends programming packets to NFU. The packets are directed to the right vault based on the vault ID of the packet by the input crossbar of the memory.

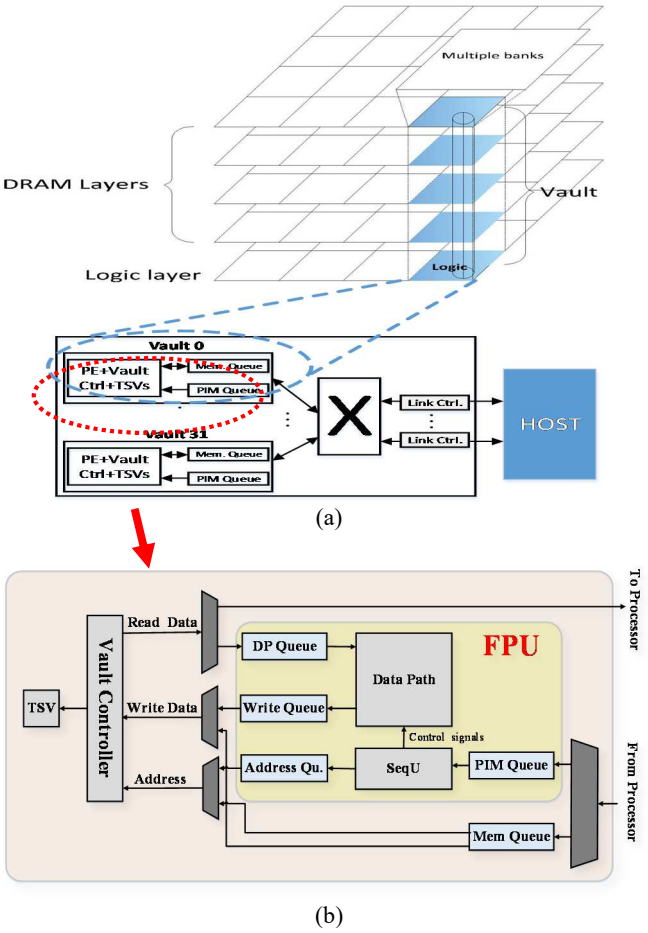

Figure 2. (a) 3D memory and memory-side PEs, (b) inside a vault logic layer

In each vault, the PIM packets are distinguished from regular read/write packets and are sent to a separate queue (*PIM Queue* in Figure 2.a) and from there, to NFU to initiate in-memory operations. Regular requests issued by the host processor are stored in *Mem Queue* in Figure 2.

(2) SeqU reads and decodes the packets of *PIM Queue* in order. It calculates the sequence of addresses from which the input data and weights should be fetched. (3) *SeqU* issues a sequence of read requests by writing the calculated address in *Address Queue*. A flag is kept for each address in *Address Queue* to distinguish between read and write requests. In his case, the flag is set to the read mode. (4) The requests at the *Address Queue* are multiplexed with the regular requests received from the host processor and are read and handled by vault controller. (5) When a read request is completed, the data is sent to *DP Queue*. (6) SeqU configures the datapath memory elements (via the control signals depicted in Figure 2.b) to write *DP Queue* contents to the right buffer slots in NFU. (7) Once all the required data are fetched, NFU starts running the neural network. (8) Upon completing the computation, the output of the neural network is written in *Write Queue*. *SeqU* generates the address at which the data should be written and queues it, with the write flag set, in *Address Queue*. (9) When the vault controller reaches to a write request in *Address Queue,* takes the data from the head of the *Write Queue* and writes it to the address picked up from *Address Queue*.

Regular accesses made by the host processor are multiplexed with NFU accesses, through the multiplexers and de-multiplexers that control the vault controller input and output signals (see Figure 2.b). Without loss of generality, we consider a priority-based multiplexing policy in which the NFU requests (in *Address Queue*) are prioritized over regular accesses (in *Mem Queue*).

B. Datapath structure

Neural network size. The NeuroPIM architecture replaces a bandwidth-intensive complex function by a neural network. However, a large neural network would require fetching many parameters from memory: this converts a bandwidth-hungry function to another bandwidth-hungry function and may offer no gain in terms of speedup and energy efficiency. However, a key observation about neural function approximation in many prior work shows that almost any kind of functions can be appropriately approximated by a small or moderately-sized neural network [7]. Particularly, our experimental results corroborate the observations in the prior work on neural network-based acceleration and shows a 2-layer MLP neural network with at most 32 neurons per hidden layer can approximate any function in our benchmarks with acceptable accuracy. Therefore, the datapath implements a two-layer neural network with 32 neurons per layer. In Section 3.d, we explain how larger neural networks are implemented.

Datapath design. Although any architecture can be used as the base of NFU architecture, we base our design on the accelerator architecture presents in [12]. The datapath structure consists of an arrays of Multiply-and-Accumulation (MAC) units and three memory elements that keep neuron's weights, inputs, and outputs. The MAC array is followed by an activation function array that generates the neuron output by a lookup table.

The weights of neurons are stored in a 2-D memory array (*Weight Buffer* in Figure 3). A *Weight Buffer* of size $n \times k$ has n rows, each of which is allocated to a single neuron with k weights (we set both k and n to 32). This way, the weights are stored in an input-major order, in which each Weight Buffer element (W_{ij}) keeps the j^{th} weight of the i^{th} neuron. As such, the j^{th} column of the *Weight Buffer* keeps the j^{th} weight of all neurons, which should be multiplied to the j^{th} input of the layer. The neural network layer is processed in k consecutive steps followed by one activation function call. At any given step j, the j^{th} weight of all neurons (stored in the j^{th} column of W*eight Buffer*) and the j^{th} input of the layer (stored in *Input Register*) enter the multiplier array. The multiplication outputs at each step are summed into their corresponding accumulator.

Once the final output values are generated, they are stored in *Output Buffer* and will be either written to memory or be used as input for the next layer.

Second layer support. To support two layers, we duplicate the *Weight Buffer*, so each instance of the buffers keeps the weights of one layer. Based on the layer that is being processed, SeqU selects the right weight through the multiplexers in front of the MAC units. The MAC array is not duplicated, because the two layers are processed sequentially one after the other, so do not use the MAC units at the same time.

The first layer receives its input from memory. However, the second layer uses the output of the first layer as input. To make proper input-output connections, during the execution of the second neural network layer, the *Output Buffer* elements, which contain the output of the first layer, are shifted to *Input Register* one cycle at a time to be used as the input. The multiplexer in front of the input register (which is controlled by SeqU) is responsible for selecting the right data. Finally, the output of the second layer should be written to memory.

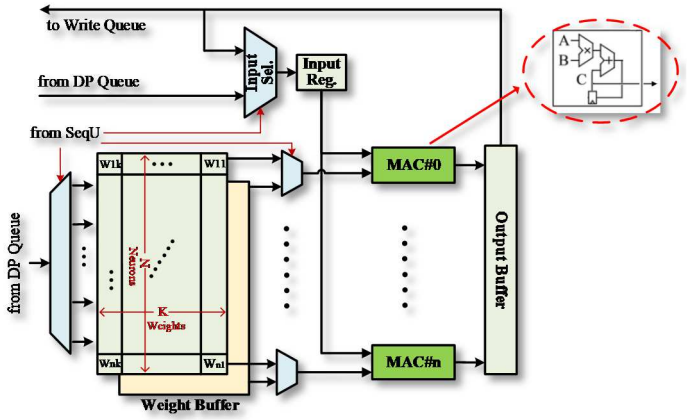

Figure 3. datapath architecture

C. Sequencer unit

Again SeqU generates the right signals to direct the *Output Buffer* data of the second layer to memory. The SeqU unit has three tasks: (1) loading the right neural network weights and data to NFU according to a pre-defined schedule (2) configuring NFU to write the fetched data onto the right buffer location and selecting the right operands for the MAC units, and (3) writing the generated result to *Input Register* to be used by the next layer or to a given memory location.

If multiple parts of a code are accelerated by different neural networks, invocation of each accelerated code segment enforces context switches between the corresponding neural networks. In this case, the weights of the invoked neural network should be loaded before data processing. However, if there is only a single neural network or one neural network is to be invoked several times consecutively, the weights are kept stationary in NFU and can be reused; hence there is no need to load them for each individual NFU call.

SeqU programming. As mentioned before, neural information processing at the memory side is initiated by receiving a special instruction from the host processor. The instruction carries some pieces of information, as follows.

- SrcAdr: the start address of the input data in memory
- WAdr: the start address of neural network weights in memory
- DestAdr: the start address for writing the neural network output
- InpCnt: the number of inputs
- LyrCnt: the number of layers (fixed to tow in our design)
- LyrSize[1..LyrCnt]: the number of neurons in each neural network layer.

The data structures are consecutively located in each vault, so that one read operation may bring multiple input data.

SeqU operation. Once SeqU receives the invocation packet from the host, it runs the neural network one layer at the time. For the first layer, it provides the required data and control signals for the input-major execution order of the datapath.

The first neural network layer has LyrSize[0] neurons, each with InpCnt weights (See Figure 1). The weights are stored in a region in memory starting from WAdr to WAdr+LyrSize[0]×InpCnt. Since the weights are used by the datapath in an input-major order, they are stored in the same way in memory. This way, the first weights of all LyrSize[0] neurons of layer 1 are stored in the address range of WAdr to WAdr+ LyrSize[0]-1, the second weights in the address range of WAdr+ LyrSize[0] to WAdr+ LyrSize[0]×2-1 and so forth. Finally, the last weights of all neurons are stored in the address range of WAdr+LyrSize[0]×(InpCnt-1) to WAdr+ LyrSize[0]×InpCnt-1.

Accordingly, the input is stored in a range of addresses from SrcAdr to SrcAdr+InpCnt-1.

The entire layer is processed in InpCnt steps. In each step i, SeqU takes a sequence of actions as follows.

(1) SeqU generates a sequence of read requests from WAdr+LyrSize[0]×(i-1) to WAdr+LyrSize[0]×i-1 to fetch LyrSize[0] elements from the neural network weights associated with the i^{th} input. For each data, SeqU sends a read request to the vault controller which is queued in *Address Queue* in the read mode. Once the request is replied by the vault controller, SeqU generates the right buffer address and asserts the write enable signal of *Weight Buffer* of the datapath to write the data.

(2) In the same way as it does for weights, SeqU generates a read request to fetch the i^{th} input element from the address SrcAdr+i-1 and moves it to *Input Register* in datapath.

(3) Once one input and one array of weights are fetched and written in the right buffer slots, the execution of the layer is triggered by SeqU. To this end, SeqU sends the contents of *Input Register* and the i^{th} column of *Weight Buffer* to the MAC array. This way, the *Input Buffer* that contains the i^{th} input of the layer is multiplied to the i^{th} weight of all neurons in parallel. the partial result obtained in this cycle are then accumulated in the accumulator.

When the layer is processed in InpCnt cycles, SeqU enables the activation function unit to process the content of the accumulators and asserts the write enable signal of *Output Buffer* to write the final result of the layer.

In parallel with the computations of the first layer, the SeqU continues with fetching the weights of the next layer to the second *Weight Buffer*. For the next layers, there is no need to fetch input data, since it is already available at the *Output Buffer*. Thus SeqU only configures the input-output connection of the datapath and shifts the contents of *Output Buffer* to the *Input Register* to be processed by the next layer.

When the neural network computation is completed, the SeqU issues a sequence of write requests to transfer *Output Buffer* contents to memory (in the address starts with DestAdr).

As mentioned before, if there is only one neural network for each vault or a neural network is called multiple times consecutively, the weights are only loaded at the first call and reused on the next calls. Therefore, SeqU should only deal with input data transfer on the second calls on.

D. Implementation issues

Data type. Although software tools (e.g. TensorFlow) originally use 32-bit floating-point numbers to represent neural network weights, most MLPs can work properly with narrower bit-widths. NeuroPIM stores weights and input data in 8-bit fixed-point numbers. This is the data format/size at which many feedforward neural networks (including all neural networks that we tested) work with no considerable accuracy loss. We use 16-bit for input data to provide sufficient resolution for function inputs that are fed to the neural network. The datapath, consequently, implements 8×16 fixed point multiplier and 16-bit adder units. Wider integer or floating-point numbers should be casted to this type.

This way, regarding the 64-bit bus size of HMC, each read and write transaction transfers 8 weights and 4 input data elements.

The overhead of NFU configuration. In many applications, the number of functions that are amenable to neural acceleration is limited to a few functions. Recall that the

functions should be memory-intensive, tolerate some output error, and be complex and called frequently in order to be amenable for neural acceleration. When the number of candidate functions, and hence the equivalent neural networks, are limited, there is no need to load the weights for each NFU invocation: rather the weights are fetched once at the first call and will be reused in consequent calls. If one application needs more NFUs to accelerate more functions, the data can be appropriately mapped across different (maximum 32 vaults) in such a way that the NFUs are utilized concurrently across all vaults. To this end, the input data of different accelerated functions and the corresponding neural network weights should be mapped onto different vaults to use different NFUs, effectively avoiding weight switch on the same NFU.

Note that although a few functions in a typical application hold these criteria, since they are called frequently, they account for a major part of the application's execution time and bandwidth usage, thereby the entire application highly benefits from the neural acceleration.

Large neural network support. We limit the number of hidden layer neurons to 32 and the number of layers to two. However, the number of inputs of the hidden layer neurons and the number of output layer neurons is determined by the number of neural network input and output sizes, respectively. To use NFUs to run larger neural networks, the execution is carried out in multiple steps: in each step, part of the neural network is processed. This requires slight modification to the controller.

Each MAC keeps a vector of length k (set to 32 in this paper) to keep neuron weights (Figure 3). If the neural network has more than k inputs, the weights of the first layer need a longer vector. In this case, the weights are partitioned into k-element groups and loaded into the datapath one by one. Note that the number of inputs of the second layer cannot exceed the number of neurons of the hidden layer (see Figure 1) that is set to 32, so cannot take unexpected values higher than 32.

Moreover, if the number of neurons of the second layer is higher than 32, which is the number of available neurons in a NFU core, the neurons will be divided into groups of 32 and are processed serially.

In both cases, we need to switch neural network parameters during the NFU execution. In this case, the programmer can compare the execution time of the original function and the neural accelerated version and decide whether or not to use NeuroPIM to accelerate the function.

Datapath partitioning. The datapath consists of n instances of the MAC unit, each running a single neuron. Since it is likely that some functions need fewer neurons, we modify the architecture to allow parallel execution of multiple neural networks on the same NFU.

To run C neural networks concurrently, the controller unit should maintain C copies of the neural network state registers. The registers, as mentioned before, keep the start addresses for the source and destination data, the number of inputs, and the number of neurons per each layer.

The control signal of the MACs is multiplexed among the C controller slices (through a multiplexer in front of o each control signal) and a global configuration controller set the multiplexers in such a way that each passes the control signals from the right controller slice to its associated MAC. In order to reduce the wiring overhead, MAC units are grouped into some clusters and are allocated to a neural network at the cluster granularity. Since we observed that most application have a limited number of functions for memory-side acceleration, implementing this feature is left for future work.

IV. Experimental Results

Simulation setup. To evaluate NeuroPIM, we use a set of benchmarks picked up from the AxBench suite [13]. All AxBench programs have one or more functions that are replaced with neural network [13], but we select the programs whose accelerated functions need many memory accesses and show poor caching behavior. In addition to the AxBench programs, we also use the SSIM program as benchmark: The structural similarity (SSIM) index is a metric to evaluate the quality of an encoded video. The SSIM of the 16-pixel blocks of an original video frame (x) and its encoded version (y) are calculated as:

$$\mathrm{SSIM}(x,y) = \frac{(2\mu_x\mu_y+c_1)+(2\sigma_{xy}+c_2)}{(\mu_x^2+\mu_y^2+c_1)(\sigma_x^2+\sigma_y^2+c_2)}$$

This function is complex, needs abundant data, and is not cachable. As a result, we accelerate it with NeuroPIM. Our implementation shows that a 2-layer neural network with 21 neurons (20 in the first layer and one in the second layer) can approximate the SSIM function with less than 5% error that translates to less than 0.005 MSE error in programs output.

The benchmarks, the structure or their equivalent neural network, and the accuracy of the neural network are listed in Table 1.

Table 1. Benchmarks

Benchmark	Neural Network Topology (input-hidden-output)	Benchmark Mean Square Error (MSE)
Blackscholes	6→4→1	0.00447
Inversek2j	2→8→2	0.00563
Jmeint	18→8→2	0.00530
JPEG encoder	32→16→32	0.00590
Sobel	9→8→1	0.00234
SSIM	32→20→1	0.00467

We implement NeuroPIM in the widely-used MARSSX86 cycle-accurate microprocessor simulator. We chain MARSSX86 with DRAMSim [14] to get realistic memory access numbers. The memory is configured as a 3D HMC-like device with the specification outlined in Section 2.a (32 vaults, 10 GB/s bandwidth per vault, 120 GB/s off-chip bandwidth). The working frequency of CPU and the logic layer of the 3D memory are set to 2.5GHz and 1.25GHz, respectively. 1.25GHz is the realistic working frequency that is reported for the logic layer in HMC specifications [15].

Simulation results. We compare NeuroPIM with two other configurations: when the programs run the original function on a processor and when the programs use a neural network accelerator, but at the processor side. The latter uses the same NFU architecture, but implements it in the processor-side, hence should use the off-chip bandwidth to read/write data.

Figure 4 shows the speedup of NeuroPIM over the two other designs. The numbers are normalized with respect to the original program results in order to get a better understanding on the improvements. For reference, the execution time of the original JPEG encoder on MARSSX86 is 2.9×10^{10} cycles to encode a single image. During the execution, the program spends 23% of the time waiting to receive data from memory. As the Figure shows, the speedup obtained for the processor-side neural network shows how much performance improvement would be obtained if the original functions are

replaced with neural networks. The results of NeuroPIM shows the extra performance improvement that the in-memory computation yields.

The obtained speedup depends on the complexity and memory access rate of the original accelerated functions. For some applications, such as Blackschols, where very complex and memory-intensive functions are accelerated, neural acceleration and in-memory computing yield 4.5x and 3x speedup, respectively.

We also compare the power consumption of the designs. Figure 5 shows the power reduction of the memory-side and processor-side neural accelerated designs over the baseline. The power consumption of those parts of the codes that run on the processor is calculated by McPAT [16]. The power consumption of the neural network-based accelerators is extracted from the hardware implementation of the designs in a 15nm ASIC library [17]. The memory access energy consumption for memory-side and processor-side designs are taken from [2] and set to 3.7 pJ/bit and 10 pJ/bit, respectively. These energy numbers include the vault controller DRAM array energy usages. For the processor-side design, the packetization and off-chip link traversal energy is also included.

As Figure 5 shows, NeuroPIM outperforms the processor-side designs in terms of power consumption. The main source of power reduction is the elimination of on-board data transfer that has a major contribution to the total power consumption of the programs.

Overhead analysis: To analyze the circuit-level overheads, i.e. area and temperature, the RTL code of the NeuroPIM in VHDL. The description is synthesized using a publically available 15nm technology library [17].

Based on the synthesis results, the area of the NeuroPIM architecture with 32 MACs and two 32×32 Weight Buffers is 980um^2. The area of the logic layer of HMC is around 2.5mm^2 [2][10]. Less than half of the vault logic layer's area is used to implement vault controller, leaving enough room to implement NeuroPIM.

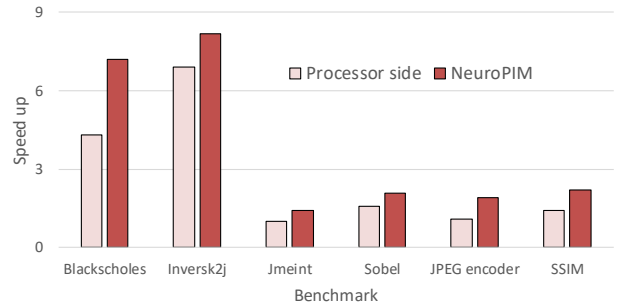

Figure 4. Speedup results

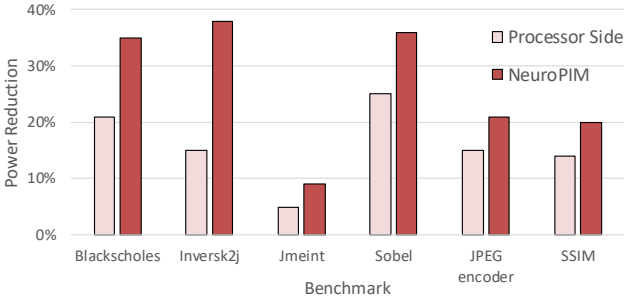

Figure 5. Power reduction results

To estimate the temperature of NeuroPIM, we use the Hotspot 6.0 temperature estimation tool [18] with its 3D stacking feature under the parameters picked up from [19]. The evaluations show that the peak temperature of the logic layer at 1.25GHz is 352˚K that lies within the safe temperature of HMC. According to the HMC specification [10], the maximum safe operating temperature of the logic layer is 383˚K.

V. Conclusion

In this paper, a new processing-in-memory architecture is proposed which uses a memory-side neural network as a reconfigurable general-purpose accelerator. This architecture makes an appropriate trade-off between the flexibility of general-purpose processors and superior performance of application-specific accelerators. The design is specifically beneficial to the applications with complex functions that make high memory access rates and exhibit poor caching behavior. The experiments on several benchmarks shows up to 41% speedup over processor-side acceleration.

Acknowledgement

This work was supported in part by the European Union through European Social Fund in the frames of the "Information and Communication Technologies (ICT) programme" and by the Swedish Innovation Agency VINNOVA project "FASTER-AI".

References

[1] O. Mutlu, et al., "A Modern Primer on Processing in Memory", in *Emerging Computing: From Devices to Systems. Computer Architecture and Design Methodologies- Chapter 4*, Springer Pubs, 2023.

[2] D. Kim, J. Kung, S. Chai, S. Yalamanchili and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory", *in Proc. of ISCA*, 2016.

[3] N. Akbari, M. Modarressi, M. Daneshtalab and M. Loni, "A Customized Processing-in-Memory Architecture for Biological Sequence Alignment", *in Proc. of ASAP,* 2018.

[4] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna and O. Mutlu, "Processing-in-memory: A workload-driven perspective," in *IBM Journal of Research and Development*, 2018.

[5] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *Proc. of HPCA*, 2017.

[6] I. Magaki, M. Khazraee, L. V. Gutierrez and M. B. Taylor, "ASIC Clouds: Specializing the Datacenter," *in Proc. of ISCA,* 2016.

[7] H.Esmaeilzadeh, et al., "Neural acceleration for general-purpose approximate programs", in *Proc. of MICRO*, 2012.

[8] A. Yazdanbakhsh, et al., "Neural acceleration for GPU throughput processors", in *Proc. of MICRO*, 2015.

[9] M. Daneshtalab and M. Modarressi. *Hardware Architectures for Deep Learning*. IET publishers, (2020).

[10] Hybrid Memory Cube Specification 2.1, Nov. 2015, Hybrid Memory Cube Consortium, Tech. Rep.

[11] H. Jun et al., "HBM DRAM Technology and Architecture", in *IEEE International Memory Workshop (IMW),* 2017.

[12] Ali Yasoubi, et al., "Power-Efficient Accelerator Design for Neural Networks Using Computation Reuse", in *IEEE Comput. Archit. Letters*, 2017.

[13] AxBench: Approximate Computing Benchmarks, http://axbench.org.

[14] DRAMSim, https://github.com/umd-memsys/DRAMSim2, 2023.

[15] R. Hadidi, et al., "Demystifying the characteristics of 3D-stacked memories: A case study for Hybrid Memory Cube", in *Proc. of IISWC, 2017.*

[16] McPAT, https://www.hpl.hp.com/research/mcpat/, 2023.

[17] NanGate 15nm Open Cell Library, [http://www.nangate.com], 2023.

[18] Hotspot, available: http://lava.cs.virginia.edu/HotSpot.

[19] M. Keramati, et al., "Thermal management in 3d networks-on-chip using dynamic link sharing," *Elsevier MICPRO*, 2017.