Low power general purpose loop acceleration for NDP applications

Athanasios Tziouvaras
attziouv@uth.gr
Dept. of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

Georgios Dimitriou
dimitriu@uth.gr
Dept. of Computer Science
University of Thessaly
Lamia, Greece

Fotis Foukalas
f.foukalas@ucl.ac.uk
Brain Sciences
University College London
London, United Kingdom

Georgios Stamoulis
georges@uth.gr
Dept. of Electrical and computer engineering
University of Thessaly
Volos, Greece

ABSTRACT

Modern processor architectures face a throughput scaling problem as the performance bottleneck shifts from the core pipeline to the data transfer operations between the dynamic random access memory (DRAM) and the processor chip. To address such issue researchers have proposed the near-data processing (NDP) paradigm in which the instruction execution is moved to the DRAM die thus, lowering the data movement between the processor and the DRAM. Previous NDP works focus on specific application types and thus the general purpose application execution paradigm is neglected. In this work we propose an NDP methodology for low power general purpose loop acceleration. For this reason we design and implement a hardware loop accelerator from the ground up to improve the throughput and lower the power consumption of general purpose loops. We adopt a novel loop scheduling approach which enables the loop accelerator to take advantage of the dataflow parallelism of the executing loop and we implement our design on the logic layer of a hybrid memory cube (HMC) DRAM. Post-layout simulations demonstrate an average speedup factor of 20.5x when executing kernels from various scientific fields while the energy consumption is reduced by a factor of 9.3x over the host cpu execution.

CCS CONCEPTS

• **Computer systems organization** → **Interconnection architectures**; *System on a chip*; *Processors and memory architectures*.

PCI '20, Greece,

ACM ISBN 978-1-4503-8897-9. . . $15.00
https://doi.org/10.1145/1122445.1122456

KEYWORDS

Near data processing, mesh interconnection, system on chip, loop acceleration,dataflow

ACM Reference Format:
Athanasios Tziouvaras, Georgios Dimitriou, Fotis Foukalas, and Georgios Stamoulis. 2020. Low power general purpose loop acceleration for NDP applications. In *Proceedings of PCI 2020: Panhellenic Conference on Informatics (PCI '20).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Contemporary computer systems are bound by heavy energy constrains that impose penalties on their throughput scaling capabilities [1]. In order to alleviate such limitations researchers have proposed to execute the computations closer to the data, i.e. to the DRAM. Such an approach is mainly supported by the the appearance of through silicon vias (TSV) interconnections and 3D stacked memories which are key enablers for the NDP paradigm as mentioned in [2]. The NDP promises to alleviate the performance bottleneck imposed by the DRAM bus as a previous survey in [3] elaborates and also to increase the performance-to-power ratio of modern processing systems [4]. In previous work in [3] authors argue the existence of a performance wall due to the slow RAM - CPU communication and mention that such a wall leads to performance scaling problems. Under this premise, researchers mainly focus on specific application types to perform NDP optimizations such as graph processing [5], bitwise operations [6], big data applications[7] and neural network inference [8][9].

In this work we diverge from application specific application execution and we propose an NDP methodology for general purpose loop acceleration. Our design is optimized for general purpose instruction execution, in order to cover a wide range of application types. Our accelerator architecture consists of a number of processing elements (PEs) capable of executing simple arithmetic or logical operations, and of a mesh interconnection network that handles the communication between the deployed PEs. We also propose a novel

instruction issue technique which issues each loop instruction on a single PE in order to leverage the dataflow parallelism of the executing loop. We evaluate our methodology in a post-layout netlist of RISC-V out of order (OoO) BOOM core and in a post-layout implementation of a hybrid memory cube (HMC) DRAM.

The main contributions of this work to the current state of the art are the following:

- Design and implementation of an NDP methodology for general purpose loop execution. We employ a general purpose approach instead of focusing on an application specific methodology as the most of previous works do.
- An NDP methodology which schedules each loop instruction on a single PE. In this way each PE executes one instruction iteratively until the loop execution completes and thus, the system throughput is maximized.
- An evaluation process on post-layout netlists instead of relying on software simulations.

The rest of this paper is organized as follows. In sec.2, we provide the background of HMC DRAM used for this work. In sec.3, we present the system architecture for general purpose loop acceleration. In sec.4 we elaborate on the evaluation process and sec.5 concludes our work.

2 BACKGROUND

HMC architectures are 3D-stacked DRAMs which are widely adopted by NDP paradigms as previous work in [10] shows. An HMC DRAM consists of multiple DRAM layers and achieves much greater internal bandwidth than conventional DRAMs, due to the through-silicon vias (TSVs) it employs. TSVs are vertical silicon links that connect the multiple layers of a 3D-stack DRAM together. This enables the HMC to transfer data in parallel between the internal DRAM layers in high transfer rates. According to the HMC consortium specifications in [11], an HMC is organized in vertically structured memory vaults which are consisted of smaller partitions. Each partition contains a number of banks which are DRAM cells, storing the DRAM data. The lower DRAM layer is reserved for implementing custom logic and facilitates vault controllers that manage the data transfer process between the corresponding vaults. In this sense the vault controllers act as memory controllers for internal DRAM data transfer between the vertically deployed HMC layers. They also handle the refresh operations of each vault removing this responsibility from the host memory controller. In this work we employ the NDP paradigm and thus, we also adopt the aforementioned HMC hierarchy according to the industrial specifications issued by the HMC consortium in [11].

3 SYSTEM ARCHITECTURE

3.1 Hardware loop accelerator

The proposed loop acceleration architecture focuses on accelerating of general purpose loops while leveraging the dataflow parallelism of the executing instructions. Fig. 1 depicts the architecture of a $4x3$ PE network implemented on the logic

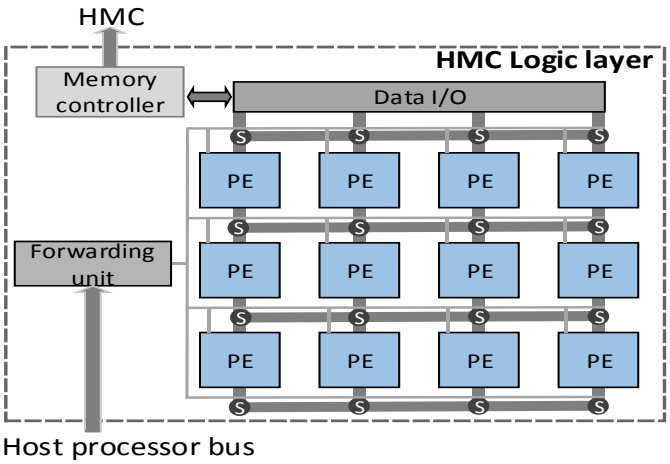

Figure 1: The architecture of the general purpose loop accelerator.

layer of the HMC DRAM. The network consists of multiple processing elements (PE) organized in a mesh-like structure and of on-chip interconnects capable of handling the communication between the PEs. Each PE is capable of executing arithmetic or logical operations and utilizes inputs either from the HMC DRAM or from the outputs of other PEs. The flow of data between the HMC and the PE network is controlled by the memory controllers (or vault controller as mentioned in sc.2) which execute memory requests heading to the HMC DRAM. In order to manage the data transfer within the mesh network we utilize switches that are designed to redirect data paths to the designated PEs. Also the communication between the host processor and the HMC DRAM is conducted via the host processor bus. Below we discuss the microarchitecture of the units deployed on the loop accelerator design.

Forwarding unit: The forwarding unit manages the forwarding and stalling processes by generating the necessary signals that propagate to the corresponding switches and PEs. To this end, such signals are used to eliminate the data dependencies of the executing instructions. As data dependencies do not change over consequent loop iterations, this process needs to be conducted at the beginning of the loop execution in order to open the corresponding data forwarding paths. Such data paths may change during the run time only when control statement evaluation results in changing the instruction execution sequence flow. In this case the forwarding unit redirects the corresponding control signals to forward the required data to the depended instructions issued on the PEs. As a result any unnecessary switching signals are omitted and dynamic power consumption is reduced. The forwarding unit also generates stalling signals freezing the instruction execution on specific PEs when necessary.

Processing element: The microarchitecture of a PE is depicted in fig. 2. Each PE is composed of a pipelined ALU or FPU unit capable of executing arithmetic, logical or comparison operations and of two multiplexers(switches) that

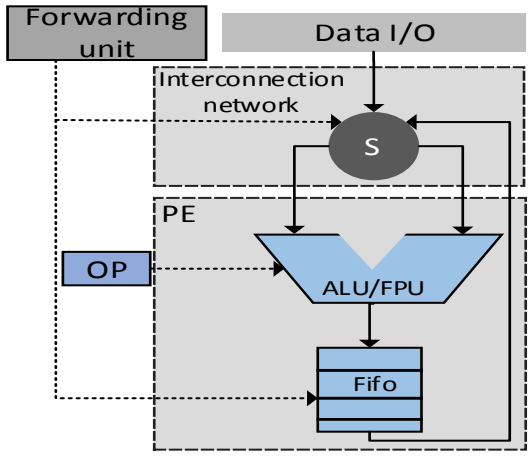

Figure 2: PE microarchitecture

control the unit's input operands. Such inputs may originate from the DRAM, from other PE outputs or from the output of the same PE, depending on the data dependencies of the executing loop. To resolve such dependencies we utilize the forwarding unit which selects the appropriate inputs for each PE as described above. Each PE is assigned one instruction and proceeds in executing it iteratively until the loop execution completes. We modify the PEs so that the outputs of such operations are temporarily stored to a fifo queue before propagated to the PE network. In case of a stalling action the forwarding unit evokes the write privileges of the ALU/FPU output to the queue and thus, no new entries are stored.

3.2 Loop execution in PE network

In this work we focus on improving the throughput and the area efficiency of the loop accelerator design and thus, we opt for an loop execution methodology that takes advantage of the dataflow execution paradigm. To this end each PE is assigned with the execution of one instruction only, as discussed in the previous subsection. As a result, each PE iteratively executes the same instruction for each loop iteration until the loop execution completes. According to this paradigm, when the accelerator pipeline is full we obtain a theoretical throughput of one loop iteration per clock cycle. More specific, algorithm 1 depicts the scheduling operation which is conducted by the host system and enables the loop to be executed on the PE network. The host processor fetches and decodes all the loop instructions as the PE network does not support any of the aforementioned operations. In the sequel the data dependencies of the loop instructions are analyzed and the host system checks if their amount is smaller or equal to the amount of the available PEs. If not, the loop fission operation takes place as in [12] which splits the large loop into groups of smaller loops which can be scheduled according to our technique. Then, the decoded instructions along with the data dependency information are dispatched to the PE network and the NDP execution commences. When the loop execution

Algorithm 1 The loop scheduling operation.

```
Fetch and decode the instructions of the loop.
Analyze loop instruction dependencies.
if Instr. No. >= PE amount then
    Perform loop fission.
    Save the resulting loops.
end if
while Remaining loops do
    Dispatch the next loop to the PE network.
    Dispatch the instruction dependencies to the forwarding
    unit.
    Initialize NDP processing.
    Wait for the execution to finish and collect the results.
    Remaining loops −−.
end while
```

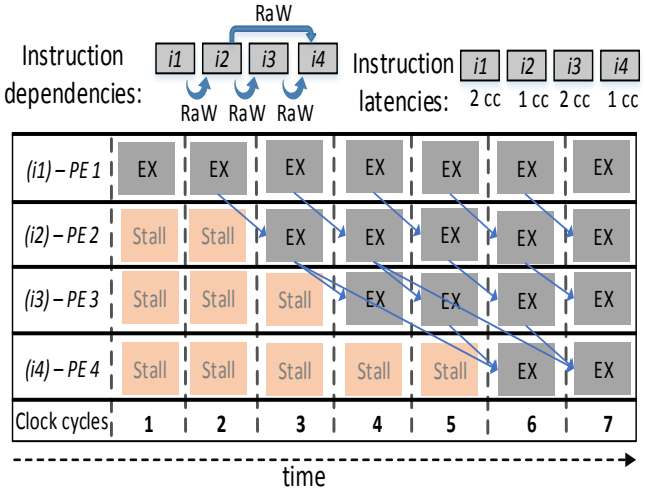

Figure 3: Loop execution instance on the PE network.

completes the results are collected by the host processor via the processor bush and the next NDP loop execution begins. This process is repeated until all the instruction loops are executed on the PE network. Our technique ensures that each loop is executed after a simultaneous dispatch of their instructions to the functional units. Consecutive iterations are completed at the rate of one iteration per clock cycle, by forwarding instruction results directly between the functional units. In this way, performance is limited primarily by the number of functional units, which are much higher than the width of the host processor pipeline.

Fig. 3 depicts a run time instance of a loop after being scheduled on the PE network. We note the read-after-write (RaW) data dependencies of the instructions as well as their latency in order to provide a thorough example of our methodology. The data dependency constrains would otherwise dictate that the instruction *i4* should wait for both *i2* and *i3* to finish their execution so that the inputs for *i4* to become available. As a result the *i2* would not execute during the clock cycles 4 and 5 due to the fact that the produced output

of the clock cycle 3 would be overwritten. Stalling the *i2* for 2cc creates a 2 cycle latency bubble that is propagated throughout the instruction sequence resulting in throughput decrease and PE under-utilization. We solve this problem by employing fifo queues capable of storing the outputs of each PE hence, enabling the PEs to continue executing instructions while previously generated outputs are not discarded. As a result the *i2* instruction is never stalled, instead it continues its iterative execution while its outputs are stored in the PE's fifo. In the sequel the *i4* instruction is forwarded the corresponding results from the *i2* and *i3* fifo queue and thus, eliminating the need for pipeline stalling. We mark the forwarding process with blue arrows that depict the flow of data from the fifo queues to the inputs of the corresponding PEs. The aforementioned technique requires the mesh pipeline to be full and the intermediate results to be stored in the fifo queues of the PEs. Considering that the whole loop body is scheduled on the PE network with each PE executing one instruction per clock cycle, we are able to execute one loop iteration per clock cycle, after the loop accelerator pipeline is full.

3.3 Implementation

The physical implementation of the design is carried out by following the CAD toolchain for application specific integrated circuits (ASICs) according to industry standards. To this end, we use verilog HDL to develop the HMC and host system descriptions while synopsys design compiler is employed for the synthesis operation, gate-level and retiming optimizations. In the sequel we employ the synopsys IC Compiler for place and route,clock tree synthesis and placement operation. For synthesis and physical realization we use the 15nm FreePDK library [13] while the post-layout netlist is generated by the synopsys IC compiler. In order to verify that our design meets the timing requirements and no timing errors occur we perform static timing analysis with the synopsys Primetime tool on the post-layout netlist. Finally functionality evaluation of the NDP design is conducted by performing gate level simulations on the back-annotated post-layout netlists using the Modelsim tool.

Table 1 depicts the host system and HMC design parameters. For the host system we implement a multi-core processor that consists of two identical Berkeley Out-of-Order Machine (BOOM) [14] cores which utilize the RISC-V instruction set architecture (ISA). BOOM is an open source out of order (OoO) core that facilitates an 10 stage execution pipeline and offers parameterized synthesis options. We tune such parameters to 32KB L1 and 512 KB L2 cache sizes, gshare branch prediction mechanism and 512 TLB entry size in order to resemble modern process designs. For the HMC DRAM implementation we use the openHMC netlist which is a configurable open source HMC architecture [15] developed by the Heidelberg University. We tune the HMC parameters to align its specifications according to the industry standards set by Hybrid Memory Cube Consortium (HMCC) in [11]. Specifically we opt for 8 GB memory size with 32 memory vaults each containing 16 memory banks, resulting in a total of 512 memory banks. We also use a 128-bit memory bus width and the GPI/O serial links that transfer data to the host system are capable of transmitting 480 GBps within an 8-cycle latency. The maximum bandwidth per vault is 16 GB per second for the HMC implementation.

Table 1: Key parameters of the host processor die and of the HMC DRAM.

Host processor	
Core	RiscV Boom OoO, 1 GHz, 64 bit
Amount of Cores	2
Pipeline	10 stages, 2 issue width
L1 cache	32 KB, 8-way, 4 cycle latency
L2 cache	512 KB, 8-way, 12 cycle latency
Branch prediction	gshare, 9-bit history, 512 entries
TLB size	512 entries
HMC 8 GB	
Memory vaults	32
Memory banks	512
Bus Width	128 bits
Timing	tCK = 1.2 ns, tRAS = 24 ns, tRCD = 11 ns, tCAS = 5.5 ns, tWR = 9 ns, tRP = 11 ns
Serial links	480 GBps, 8-cycle latency
BW per vault	16 GB/s

Table 2: Parameters of 32-bit and 64-bit NDP implementations.

NDP implementation parameters	**NDP-32 bit**	**NDP-64 bit**
ALU PEs	84	42
Mull PEs	40	20
Div PEs	6	3
FP ALU PEs	84	42
FP Mull PEs	40	20
FP Div PEs	6	3
Total number of PEs	180(12x15)	130(13x10)
Total power	4.7 W	4.5 W
Total area	1.7 mm^2	1.55 mm^2

We conduct a design space exploration of the proposed NDP methodology by implementing two different loop accelerator designs on the logic layer of the HMC that are depicted in table 2. Each NDP implementation consists of a number of PEs, switching elements and of an interconnection network as described above. Specifically we opt for 32-bit and 64-bit implementations, with both implementations having 6 fifo slots per PE and 800 MHz clock frequency in order to balance power requirements and performance goals. We select the amount of PEs of each implementation properly so that to satisfy a power budget of $5W$ and an area budget of $7mm^2$, as designated by the Hybrid Memory Cube Consortium (HMCC) in [11]. The 64 bit implementations require

significantly more power due to the increase of the functional unit size and thus, a lower amount of PEs is selected for such designs.

4 EVALUATION

In this section we discuss the experimentation process by which we evaluate our NDP methodology. For this purpose we have opted to use 7 kernels from the spec cpu 2017 benchmark suite [16] which are derived from various scientific fields in order to cover a wide range of applications. Due to the large benchmark size, the proposed scheduling methodology is not always able to schedule the kernel binary by mapping one instruction in one PE as discussed in section 3.2. To this end, we apply loop fission transformations during the reprocessing stage that split the kernels into groups of smaller loops. The loop fission is performed in the host processor and we include its execution time overhead to the total amount of time a kernel needs to execute. Table 3 depicts the benchmarks used for evaluation, their application areas and the amount of loop fission transformation required for each benchmark before it can be scheduled to the PE network. We quantitatively note with S (small), M (medium) and L (large) the loop fission count required for each benchmark.

Fig. 4 depicts the speedup of each NDP implementation normalized to the baseline execution time of the host system, i.e. the RISCV-V BOOM OoO dual core. To this end, we firstly run each kernel on the host processor with no NDP processing taking place. For this purpose we utilize all the available 2 RISC-V BOOM cores by constructing kernel threads which run in parallel in each BOOM core. In the sequel, we deploy the proposed NDP implementations along with the host system and we run the kernels again as our methodology dictates. Finally we compare the speedup we obtain for each NDP design over the host-only execution process. Below we discuss the results we obtain after the evaluation process is completed.

Due the the nature of NDP we observe high speedup values as the instruction execution takes place on the DRAM die and thus, the large data movement overhead between the host system and the DRAM is significantly reduced. Also, we have designed the loop accelerator to optimize general purpose loop execution thus, the benchmark execution is significantly accelerated. The speedup values we obtain range from 8.6x to 30.2x with an average of 19.3x and 21.7x for the corresponding 64-bit and 32-bit implementations. We observe that the 32-bit designs perform better due to the higher

Table 3: Benchmark characteristics.

Benchmark name	Application area	Loop fission count
bwaves	Explosion modeling	M
cactuBSSN	Relativity physics	L
leela	AI monte carlo tree search	L
x264	Video encoding	L
wrf	Weather forecasting	S
nab	Molecular dynamics	M
fotonik3d	Computational Electromagnetics	M

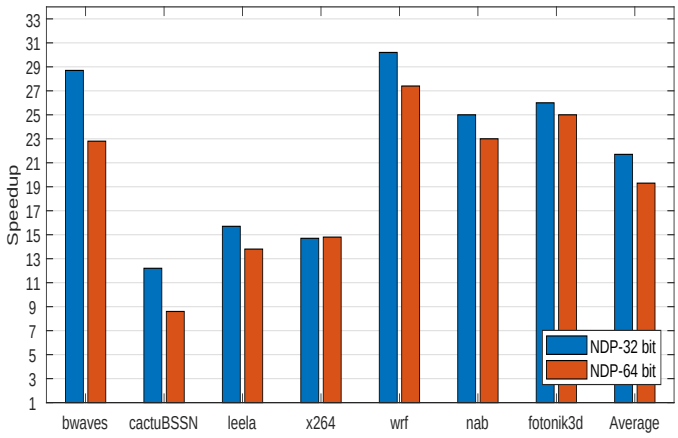

Figure 4: Normalized Speedup for different benchmarks per NDP implementation.

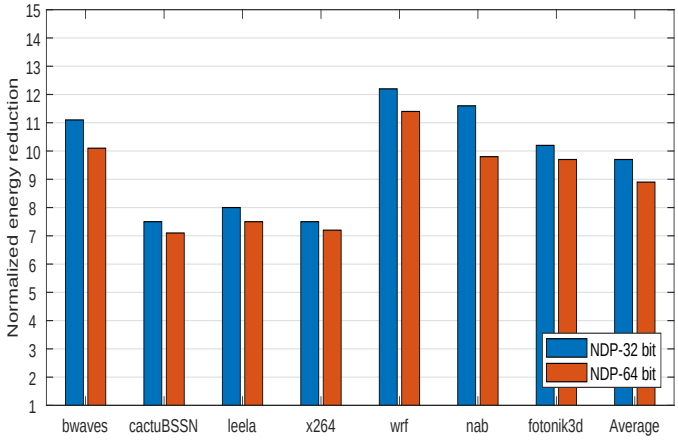

Figure 5: Normalized energy reduction for different benchmarks per NDP implementation.

amount of PEs compared to the 64-bit designs and thus, they achieve better speedup rates. Also benchmarks with lower fission counts such as bwaves, wrf, nab and fotonik3d tend to achieve higher speedups when compared with benchmarks with higher fission counts due to the fact that the former can be efficiently scheduled on the PE network.

Fig. 5 depicts the reduction of energy consumption levels for each benchmark execution, normalized to the host processor. We collect such results by averaging every implementation's normalized energy reduction for each executing benchmark. We observe an average reduction in energy consumption of 10.2x and 8.9x for the 32-bit and 64-bit implementation correspondingly. NDP achieves significantly faster execution times when compared to the host processor and also reduces the traffic between the DRAM and processor die. As a result the energy requirements of each benchmark are significantly reduced when employing the proposed NDP

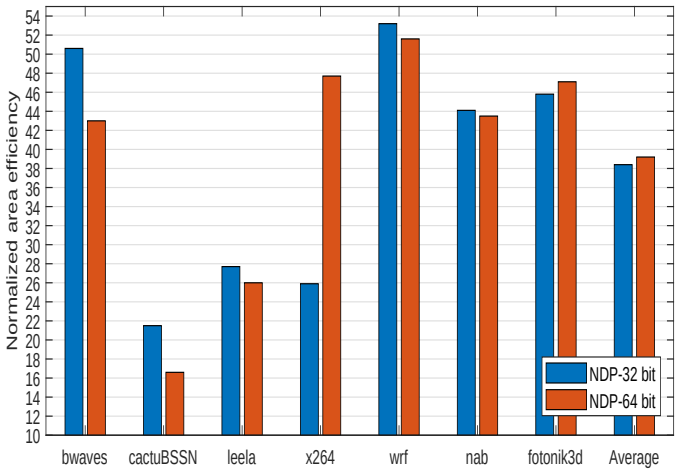

Figure 6: Normalized area efficiency of the NDP implementations.

methodology. Results vary among the executing kernels due to the wide range of requirements of the bench,arks used for evaluation. For example cactusBSSN achieves 7.5x while wrf achieves 12.2x reduction in energy consumption due to the fact that the cactusBSSN takes longer to execute while also having higher DRAM access energy overheads. The average energy reduction levels are very high for each NDP implementation which demonstrates the efficiency of our methodology.

Figure 6 depicts the area efficiency of both the 32-bit and the 64-bit NDP implementations normalized to the area efficiency of the host processor. We measure the area efficiency as the throughput achieved per mm^2 of the die area of the integrated circuit. Results indicate that the NDP designs are 38.4x to 39.2x more area efficient when compared to the host system. Such an improvement is expected due to the small die area of the NDP implementations and the high speedup rates they achieve. We also observe that the NDP-64 bit is more area efficient when compared to the NDP-re implementation despite the fact that the speedup improvement of the NDP-32 higher when compared to NDP-64. Such a difference is attributed to the die area difference of such designs as the NDP-64 is smaller than the NDP-32 implementation and thus, its area efficiency is higher.

5 CONCLUSION

In this work we have proposed a novel NDP methodology for loop scheduling and execution in the logic layer of an HMC DRAM. Our design utilizes a loop accelerator which is composed of several PEs connected with a mesh interconnection network and of a forwarding unit capable of managing the communication between the PEs. In order to exploit the capabilities of our design we employ an instruction scheduling technique that leverages the dataflow parallelism of the loop by scheduling one loop instruction per PE. In this sense the loop iterations are executed in a dataflow manner on the NDP hardware. We make a physical implementation of our design and conduct post-layout simulations on several benchmarks from various scientific field. Results indicate an average speedup factor of 20.5x while the energy consumption levels are reduced by a factor of 9.3x over the host cpu execution.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, "Dark silicon and the end of multicore scaling", in proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA), San Jose, CA, pp. 365-376, Jun. 2011.

[2] R. Balasubramonian et al., "Near-Data Processing: Insights from a MICRO-46 Workshop", in Proceedings of the IEEE Micro, vol. 34, no. 4, pp. 36-42,Aug. 2014.

[3] S. Patrick , B. Rainer and B. Mladen, "Data-Centric Computing Frontiers: A Survey On Processing-In-Memory", in proceedings of the Second International Symposium on Memory Systems, pp 295-308, Oct. 2016.

[4] Marko Scrbak, Mahzabeen Islam, Krishna M. Kavi, Mike Ignatowski, and Nuwan Jayasena. "Exploring the Processing-in-Memory design space". J. Syst. Archit. 75, 59–67,Apr. 2017.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing", in proceedings of theACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, pp. 105-117, Jun. 2015.

[6] V. Seshadri, et al., "Fast bulk bitwise AND and OR in DRAM", in IEEE Comput. Archit. Lett., vol. 14, no. 2, pp. 127–131,Dec. 2015.

[7] H. Zhang, G. Chen, B. C. Ooi, K. Tan and M. Zhang, "In-Memory Big Data Management and Processing: A Survey", in IEEE Transactions on Knowledge and Data Engineering, vol. 27, no. 7, pp. 1920-1948, Jul. 2015.

[8] F. Schuiki, M. Schaffner, F. K. Gürkaynak and L. Benini, "A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets", in IEEE Transactions on Computers, vol. 68, no. 4, pp. 484-497,Apr. 2019.

[9] S. Gupta, M. Imani, H. Kaur and T. S. Rosing, "NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration", in IEEE Transaction on Computers, vol. 68, no. 9, pp. 1325-1337,Sept. 2019.

[10] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun and Onur Mutlu, "Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions" in ArXiv, abs/1802.00320,Feb. 2018.

[11] Hybrid Memory Cube Consortium (HMCC), "Hybrid memory cube specification 2.1.", [Online]. Available: http://www.hybridmemorycube. org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf. ,2016.

[12] J.Lin, X.Tian and J Ng, "Mis-speculation-Driven Compiler Framework for Aggressive Loop Automatic Parallelization", in proceedings of IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, Cambridge, pp. 1159-1168, May 2013

[13] M. Martins, J. Maick Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open Cell Library in 15nm FreePDK Technology", in Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD '15). ACM, New York, NY, USA, pp. 171–178, Mar. 2015.

[14] C. Celio, P. Chiu, B. Nikolic, D.A Patterson, K.Asanović, "BOOM v2: an open-source out-of-order RISC-V core", Technical report in EECS Department, University of California, Berkeley, Sept. 2017.

[15] J. Schmidt and U. Bruning, "openHMC - a configurable open-source hybrid memory cube controller", in proceedings of International Conference on ReConFigurable Computing and FPGAs (ReConFig), Mexico City, pp. 1-6, Dec. 2015.

[16] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. "SPEC CPU2017: Next-Generation Compute Benchmark", in Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). Association for Computing Machinery, New York, NY, USA, 41–42, Apri. 2018.