MC3A: Markov Chain Monte Carlo ManyCore Accelerator

Lahir Marni
Dept. of Computer Science and Electrical Engineering, University of Maryland, Baltimore County
Baltimore, Maryland
mlahir1@umbc.edu

Morteza Hosseini
Dept. of Computer Science and Electrical Engineering, University of Maryland, Baltimore County
Baltimore, Maryland
hs10@umbc.edu

Tinoosh Mohsenin
Dept. of Computer Science and Electrical Engineering, University of Maryland, Baltimore County
Baltimore, Maryland
tinoosh@umbc.edu

ABSTRACT

The paper presents "MC3A"- Markov Chain Monte Carlo ManyCore Accelerator, a high-throughput, domain-specific, programmable manycore accelerator, which effectively generates samples from a provided target distribution. MCMC samplers are used in machine learning, image and signal processing applications that are computationally intensive. In such scenarios, high-throughput samplers are of paramount importance. To achieve a high-throughput platform, we add two domain-specific instructions with dedicated hardware whose functions are extensively used in MCMC algorithms. These instructions bring down the number of clock cycles needed to implement the respective functions by 10× and 21×. A 64-cluster architecture of the MC3A is fully placed and routed in 65 nm, TSMC CMOS technology, where the VLSI layout of each cluster occupies an area of 0.577 mm^2 while consuming a power of 247 mW running at 1 GHz clock frequency. Our proposed MC3A achieves 6× higher throughput than its equivalent predecessor (PENC) and consumes 4× lower energy per sample. Also, when compared to other off-the-shelf platforms, such as, Jetson TX1 and TX2 SoC, MC3A results in 195× and 191× higher throughput, and consumes 808× and 726× lower energy per sample generation, respectively.

CCS CONCEPTS

• **Hardware** → **Application specific instruction set processors**; • **Computer systems organization** → *Multicore architectures*; • **Mathematics of computing** → Distribution functions;

KEYWORDS

Manycore accelerator, ASIC, VLSI, MCMC, Uniform Random Number Generator, PDF, Metropolis-Hastings (MH)

1 INTRODUCTION

Markov Chain Monte Carlo (MCMC) algorithms are used to obtain samples from any target probability distribution and are widely used in numerical approximations, processing multi-dimensional signals, Bayesian statistics, rare event simulations and most importantly various Machine Learning (ML) applications [7][18][14]. Most of these, find themselves as real-time applications in multi-dimensional sampling, image processing and machine learning algorithms which need results in very limited time and require high throughput [8][15][5][13]. Sampling from complex probability distributions and a large number of multiple channels is often a bottleneck that results in slowing down the speed of the application. In this paper, we propose a programmable MCMC hardware accelerator, MC3A, to achieve a very high-throughput performance by customization of an existing manycore architecture. Domain-specific accelerators consisting of dedicated processing hardware, provide high processing performance and high throughput/watt performance. Programmable domain specific manycore accelerators can outperform general purpose CPUs and GPUs in terms of energy efficiency, as they use low power cores with special instructions and hardware architectures [10][11][1][4][9][2][16]. The proposed MC3A allows programming the manycore to implement various MCMC samplers such as Parallel Tempering (PT), Multiple Parallel Tempering (MPT), Gibbs Sampler depending on the application's requirement. The high throughput sampling is achieved by adding dedicated hardware and instructions to an existing manycore architecture, PENC [17], for functions that are repeatedly used and cause the overhead in most MCMC algorithms. Main contributions of this paper include:

- Proposed MC3A, a low-power, programmable manycore accelerator to efficiently execute MCMC algorithms.
- Design and implement an optimized hardware for MCMC specific instructions without slowing the operating frequency.
- Fully synthesized, placed and routed VLSI layout of the MC3A in 65 nm CMOS technology.
- Evaluation and comparison analysis of MC3A with PENC manycore, embedded NVIDIA Jetson TX1 and TX2 CPU's.

2 BACKGROUND

MCMC sampling consists of two steps: Monte Carlo, that implies that the sampling is a stochastic process, and Markov Chain, that indicates the generation of the next sample is based on the current sample. The sampling is mainly used to convert a complicated integration over a continuous multi-dimensional Probability Distribution Function (PDF) to a discrete summation on the samples, the result of which is used to calculate an expected value. Samples $x^{(t)}$, where t=1 to N, are drawn out from a target distribution. Markov chain is an iterative stochastic process which is initialized to $x^{(1)}$, and determines the next states based on a transition probability, where the transition probability can be described by a first order Markov chain as shown in equation 1:

$$P\left(x^{(i)}|x^{(i-1)}, x^{(i-2)}, ..., x^{(1)}\right) = P\left(x^{(i)}|x^{(i-1)}\right) \quad (1)$$

GLSVLSI'18, May 23–25, 2018, Chicago, IL, USA

ACM ISBN ISBN 978-1-4503-5724-1/18/05…$15.00
https://doi.org/10.1145/3194554.3194577

Algorithm 1 Metropolis Hastings (MH) Algorithm

Initialize Samples:
x_0 = p(x)
for i in 1 to N **do**
  Update Samples:
  x_p = q(x(i) | x(i-1))
  Acceptance Function:
  $\alpha(x_p) = min\{1, \frac{p(x(i-1))}{p(x_p)}\}$ //Use of EXP instruction
  u = Uniform (u; 0, 1) //Use of RAN instruction
  if u < $\alpha(x_p)$ **then**
    Accept the proposed sample: x(i) ← x_p
  else
    Reject the proposed sample: x(i) ← x(i-1)
  end if
end for

MCMC contains iterative interdependent kernels which makes hardware acceleration challenging.

2.1 MH Algorithm

In this paper, we chose the Metropolis-Hastings (MH) algorithm to verify the results and evaluate the performance of our manycore accelerator. MH Algorithm is one of the MCMC samplers that takes random samples from a PDF, p(x), to calculate the value of the required function f(x) that is proportional to PDF. In order to generate samples from p(x), the first sample is always needed to be initialized and then, three steps are performed in each iteration to generate next samples [12]:

2.1.1 Update Samples. Propose a sample, x_p, from a proposal density, q(x), which should be a symmetrical function that suggests a next sample based on the current sample. We use a Gaussian distribution for q(x). In this step, we use the current sample, x, and the target PDF, p(x), to calculate the next suggested value, x_p.

2.1.2 Acceptance function. In this step we calculate the ratio that is used to decide if the sample we had selected in the previous step is accepted or rejected. This ratio is called the acceptance ratio, denoted in equation 2.

$$\alpha = f(x)/f(x_t) \tag{2}$$

Since the PDF, p(x), is proportional to the actual function, f(x), the acceptance ratio can be re-written as equation 3.

$$\alpha = f(x)/f(x_t) = p(x)/p(x_t) \tag{3}$$

2.1.3 Accept or Reject. A uniform random number, u, is generated in the range of [0,1] and the proposed sample is accepted or rejected based on the condition mentioned in equation 4.

$$Accepted : u \leq \alpha, Rejected : u > \alpha \tag{4}$$

The detailed pseudo-code of the MH algorithm is mentioned in the Algorithm 1.

2.2 PENC Manycore

The PENC manycore is composed of 64 processing clusters (192 cores) connected through routers in a two-tier hierarchy: each cluster consists of three processing cores called Nodes, a shared memory that consists of three block memories of size 1024x16 and a low latency bus. At the next tier, all clusters are connected to each other through a hierarchy of routers. A router with five ports connects four clusters and keeps one port open for communication to the next router [17][10]. Each core processor has 128 30-bit words for storing instructions and 128 16-bit words for data storage and a six-stage processor pipeline for all instructions.

IN and OUT instructions are used to pass data between different cores. The bus is used to read the data from output FIFO and to place data into the input FIFO of the destination core. Bus arbiter takes care of granting the bus to different cores for communication. The bus has five ports connecting three cores, one Distributed Cluster Memory (DCM) and the router for communication with cores outside the cluster. The processing core contains a 32-element content-addressable memory (CAM) to store packets from the bus and allows a finite state machine to find a word corresponding to the source core field in the IN instruction. Inside each cluster, the three cores have access to the DCM using load (LD) and store (ST) instructions.

3 MC3A IMPLEMENTATION

Two important instructions, RAN and EXP, are additionally implemented in the PENC manycore in order to boost the performance and increase the throughput, as all the MCMC methods use these functions extensively throughout their algorithms [3]. RAN is a Uniform Random Number Generator, and EXP is an Exponent Function Instruction. Figure 1(A) shows the cluster architecture of the MC3A integrating the three cores, shared memory and a low latency bus. Figure 1(B) is the Post-layout view of bus-based cluster implemented in 65nm, 1V TSMC CMOS technology using Cadence Encounter. Figure 1(C) shows the 6-stage pipeline. In the execution stage of the 6-stage pipeline, ALU instructions and the RAN instruction can be completed in a single clock cycle, whereas EXP instruction is executed in 20 clock cycles and has a dedicated hardware, explained in detail in section 3.2 of this paper. Figure 1(D) shows post-layout implementation results for a single cluster of MC3A.

3.1 Uniform Random Number Generator - RAN Instruction

Uniform Random Number Generator or RAN instruction is a new addition to the MC3A architecture. Uniform Random Number Generation (URNG) is a very pivotal and extensively used operation in MCMC algorithms. In order to implement a URNG hardware, Linear Feedback Shift Registers (LFSR) are of common practice. However, In order to reduce the hardware overhead, we take advantage of a new URNG schematic proposed by [6]. The implemented URNG, whose hardware is shown in Figure 2, eliminates the barrel-shifter by introducing a simpler bit-masking block, that results in a shrunk hardware, thereby reducing the total area of MC3A. The used URNG has a high repeating period of 2^{175}. RAN R1 instruction generates a 16-bit random number and stores it in the R1 register. The implementation of RAN takes advantage of 9 Shift Registers (that are practically wires and registers, unlike in a barrel-shifter that shifts the input dynamically), 3 AND and 9 XOR gates. RAN instruction

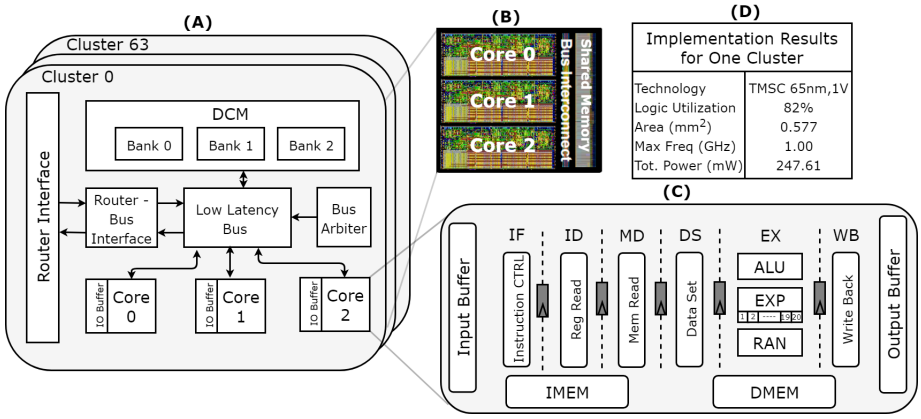

Figure 1: (A) 64 Clusters of MC3A are interconnected with routers. Each cluster consists of a distributed cluster memory (DCM), a low latency bus and three processing cores, bus arbiter and router-bus interface. (B) Post-layout view of MC3A implemented in 65nm, 1V TSMC CMOS technology. (C) Processing core architecture consisting of 6 pipeline stages where, IF, ID, MD, DS, EX and WB are instruction fetch, instruction decode, memory decode, date set, execution, and write-back pipeline stages respectively, Instruction memory(IMEM) and data memory(DMEM) are instruction and local data memory respectively. (D) Post Layout implementation results of MC3A single cluster.

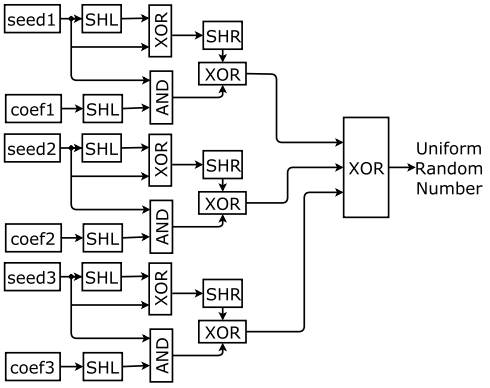

Figure 2: Block Diagram of Uniform Random Number Generator implemented using shift registers, XORs and ANDs

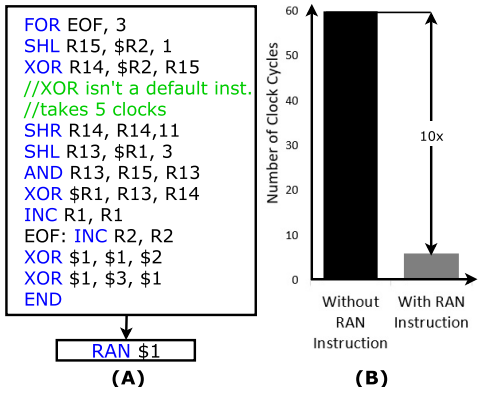

Figure 3: (A) On the above is an assembly program to implement uniform random number generator with PENC (All seeds are pre-loaded in data memory). Below is the equivalent assembly program with MC3A. (B) The graph showing the decrease in number of clock cycles with the addition of RAN instruction.

reduces the generation of a uniform random number from 60 clock cycles to 6 clock cycles with only one execution cycle. Figure 3 (A) Shows the alternate assembly instructions to generate the uniform random number with PENC and its equivalent instruction in MC3A. Figure 3 (B) shows the impact in the number of clock cycles with the addition of the RAN instruction.

3.2 Exponent Function - EXP Instruction

In addition, an EXP instruction to calculate the value of exp(-x) has been added to the execution stage of the manycore architecture. The EXP function is optimally implemented using CORDIC algorithm. CORDIC (Coordinate Rotation Digital Computer) is a simple and efficient algorithm to calculate hyperbolic and trigonometric functions such as exponent, logarithm, sine, and cosine, and is mainly used for hardware implementation. CORDIC uses pre-calculated coefficients which are stored in a look-up table, by means of which the function output converges to the actual value typically one bit per iteration. In our exponential function design, the input value is first read, and is assumed to have a temporary exponent value of 1. Then, a pre-calculated reference value is read from a ROM and is subtracted from the input value. If the result is negative, then it will replace the input value. Meanwhile, the temporary exponential value will be multiplied by another offset value, read from another ROM. Otherwise, both the input and the temporary exponential values will be passed to the next iteration without any manipulation. After a few iterations, the temporary value converges to the true exponent value. We calculated values for the reference and offset ROMs such that for an exponential function of a negative 16-bit value the process will complete after 20 iterations. If two of the iterations are merged into one clock cycle, then the critical path of the hardware exceeds 1ns, thus slowing down the system frequency below 1GHz. To eliminate this problem, a state machine is developed to implement the 20 iterations in 20 different clock cycles to maintain the minimum clock period of the manycore at 1 ns. "EXP R1, R2" reads the value from R2 in <8,8> binary fixed-point format,

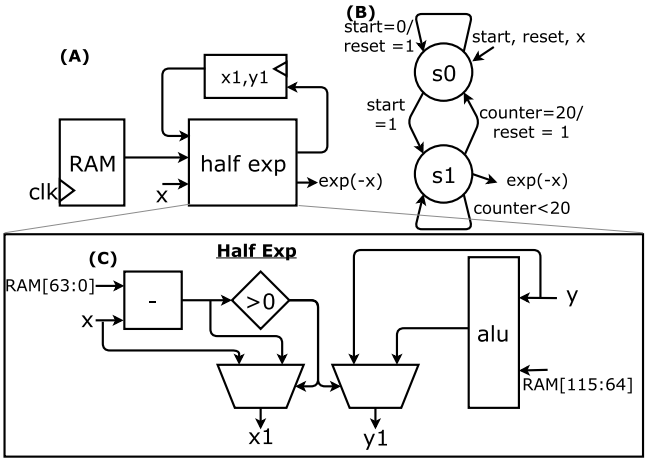

Figure 4: (A) Block diagram of exp(-x) implemented using half exp functions. All the coefficients and offset values are stored in RAM. The functionality of half exp is shown in Figure 4 (B) State Machine that implements exp(-x). (C) Block diagram for the single iteration of the half exp implemented using CORDIC.

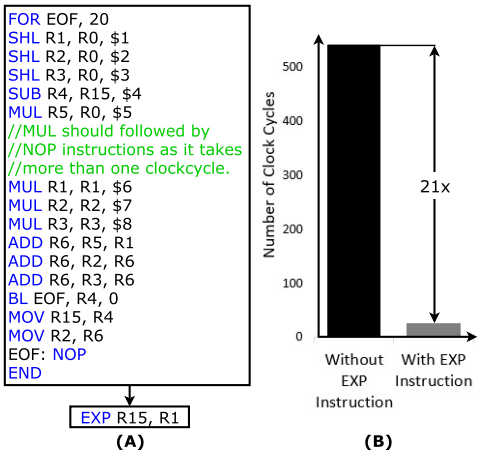

Figure 5: (A) On the above is an assembly program to implement exp(-x) in PENC (All coefficients are pre-loaded in data memory). Below is the equivalent assembly program with MC3A. (B) The graph showing the decrease in number of clock cycles with the addition of EXP instruction.

computes the value of exp(-x) and stores it in R1 as <0,16> binary fixed-point format. <0,16> binary point format is chosen because the values of exp(-x) always lie between 0 and 1. Figure 4 (A) Shows an iteration of the EXP function using Half Exp functions. Figure 4 (B) shows the state machine that divides the CORDIC exponent algorithm into sequential logic. Figure 4 (C) shows the combination logic of Half Exponent function. Implementing the EXP instruction reduces the calculation of exp(-x) from 542 clock cycles to 25 clock cycles. Figure 5 (A) Shows the alternate assembly instructions to calculate exp(-x) with PENC manycore and how the additional hardware reduces it in the proposed MC3A. Figure 5 (B) Shows the impact in the number of clock cycles with the addition of the EXP instruction.

3.3 Programming the Manycore

The manycore compiler, written in Python, converts the assembly language into binary files. It also reads in the initial values of registers and memories, and converts them into bin files for each core individually, which can be directly loaded on the manycore. A java simulator is written which is completely a software implementation of the manycore which assists with running, debugging, testing and refining the assembly code before loading it as binary files to the actual manycore. The simulator reads the assembly code and the initial state of data memory and registers, then models it as per assembly code. It then updates the final states of registers and memories. It also evaluates the summary of clock cycles it takes to execute the assembly program.

3.4 Hardware Evaluation

We implemented our algorithm on hardware for evaluating the energy consumption, execution time, area and power analysis of our proposed MC3A. The hardware model is simulated using NC-Verilog. The complete MC3A is placed and routed to generate a VLSI Layout in 65 nm 1V CMOS technology using the Cadence Encounter tool. The resulting VLSI layout is used to calculate the accurate area analysis, clock frequency and power consumption.

4 IMPLEMENTATION RESULTS AND PLATFORM COMPARISON

The MH algorithm of MCMC, presented in section 2.1, is implemented on both the PENC and MC3A manycore. The MH algorithm samples from a uni-variate Gaussian distribution as the target PDF. The results are evaluated in terms of throughput, execution time, energy and area metrics. To better gauge the performance of MC3A, the same algorithm is implemented on NVIDIA Jetson TX1 SoC CPU platform and NVIDIA Jetson TX2 SoC CPU. In-order to achieve a low power platform, the rule of thumb is to use the least number of clusters possible.

4.1 Implementation Results on NVIDIA Jetson TX1 and TX2 SoC

The MH algorithm is also implemented on the NVIDIA Jetson TX1 and the NVIDIA Jetson TX2 processors. The results of TX1 and Tx2 are scaled up (from 28nm) to match 65nm technology to be comparable with MC3A and PENC. Before collecting the power and timing analysis for the TX1 and TX2, we made sure that all the other components such as GPU, HDMI are turned off and there is no power consumption from any of these components.

4.2 Area Efficiency Analysis

To evaluate the area efficiency, we use the throughput per unit area (TPA) metric. TPA is the ratio between the throughput of each core to the total area of the core. There is a 4.7× performance increase in the TPA despite the increase in the area, because of the dedicated hardware to MC3A compared to PENC manycore. Figure 6 shows the graphical analysis of layout area of the complete cluster and the TPA metric.

Table 1: Hardware implementation results for MC3A compared with PENC manycore, NVIDIA Jetson TX1 and TX2 (CPU). For a fair comparison with Jetson TX1 and TX2 CPUs, the results are scaled to 65 nm as shown in the table

	MC3A	PENC (Base)	Improvement Over PENC	TX1 CPU	Improvement over TX1 CPU	TX2 CPU	Improvement over TX2 CPU
Technology (nm)	65	65	-	28	-	28	-
Frequency(MHz)	1000	1000	-	1224	-	1267.2	-
Cluster area (mm^2)	0.577	0.528	**0.9x**	-	-	-	-
Scaled Throughput/Area($Msps/mm^2$)	166.5	35.02	**5x**	-	-	-	-
Execution Time(nS)	42.9	247.6	**6x**	8400	**195x**	8200	**191x**
Throughput(Msps)	23.31	4.04	**6x**	0.11	**195x**	0.12	**191x**
Total Power (mW)	247.61	226.78	-	2373.33	-	2185.5	-
Scaled Power(mW)	247.61	226.78	**0.9x**	5508.75	**22x**	5072.32	**21x**
Scaled Energy Consumption(uJ)	10.6	41.29	**4x**	8587.8	**808x**	7719.85	**726x**

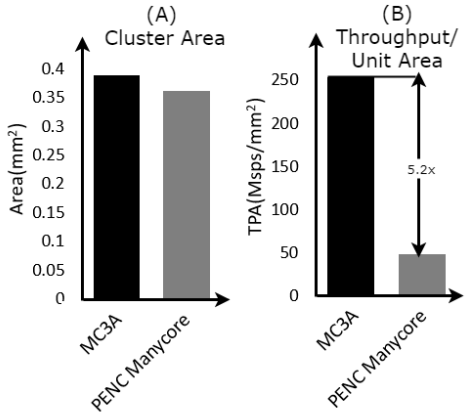

Figure 6: (A) Graph showing the areas for a single cluster of MC3A and the PENC manycore. (B) Graph showing the Throughout per unit area (TPA) of the core and the performance improvement of MC3A. The units of TPA are million samples/second/mm^2

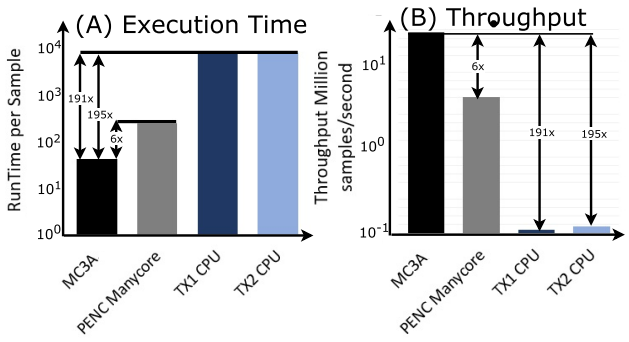

Figure 7: (A) Graph showing the Execution time and the amount of speed-up of MC3A compared to PENC manycore, TX1 CPU and TX2 CPU. (B) Graph showing the throughput (million samples per second) and the amount of increase in throughput of MC3A compared to PENC manycore, TX1 CPU and TX2 CPU. (All the graphs are plotted in logarithmic scale)

4.3 Execution Time and Throughput Analysis

It takes the MC3A 42.9 ns to generate a single sample using MH algorithm achieving a throughput of 23.3 million samples per second. A speed-up of around 6× is achieved compared to the PENC manycore. Figure 7 shows the execution time analysis per sample and throughput per second of the MC3A versus the PENC manycore. The results of this experiment show an improvement of 191× and 195× more throughput over Jetson TX1 and Jetson TX2 CPU platforms respectively.

4.4 Energy Efficiency Analysis

Although there is a considerable amount of hardware overhead to achieve great speed and high throughput, a significant decrease in the energy consumption was achieved. The power consumption per cluster has increased to 247.6 mW, but there is 4× decrease in the amount of energy consumed compared to PENC manycore. Figure 8 shows the power and energy analysis of the MC3A versus the PENC manycore and illustrates the energy efficiency it brings in the MCMC algorithms. The results of this experiment shows an

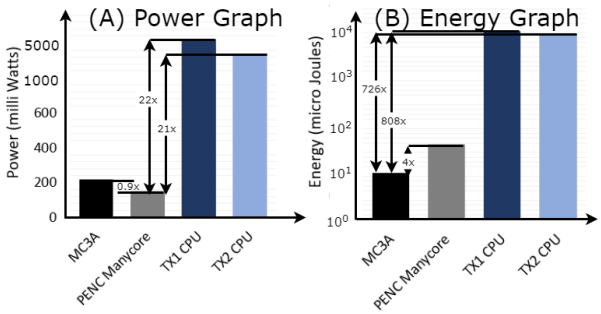

Figure 8: (A) Graph showing the Power in mW for MC3A compared to PENC manycore, TX1 CPU and TX2 CPU. (B) Graph showing the energy (μJ) in logarithmic scale and the amount of decrease in energy of MC3A compared to PENC manycore, TX1 CPU and TX2 CPU.

improvement of 808× and 726× over TX1 and TX2 CPU platforms respectively.

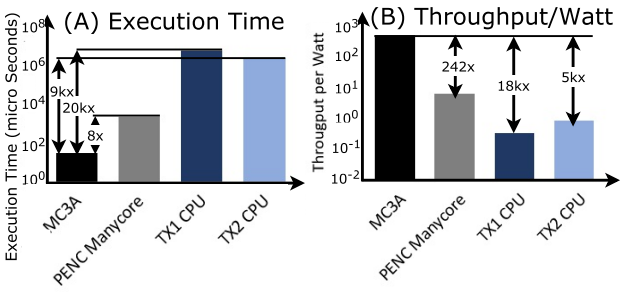

Figure 9: (A) Comparison graph of time taken to generate 10000 sample for 100 channels across MC3A, PENC, TX1 and TX2 CPUs. (B) Graph comparing efficiency metric (million samples per second/watt) to generate 10000 samples for 100 channels across MC3A, PENC, TX1 and TX2 CPUs (For both (A) and (B) quad-core CPU parallel, all channels evenly distributed among all the processes, and processes are distributed evenly among all the cores of the CPU)

Table 2: Compression of MC3A with existing work

	[7]	This Work
Implementation	Artix-7 FPGA	Domain-specific Manycore
Frequency (MHz)	200	1000
Throughput (Msps)	5.04	23.31
Improvement	Base	5x

4.5 Comparison with Parallel Multi-Core CPU

Applications that use MCMC algorithms usually need many samples from multiple channels following the distribution of data in the channel. So, the present MH algorithm is scaled to 10000 samples and 100 channels. The scaled results of the MC3A are compared across various platforms, PENC manycore, TX1 (quad-core) and TX2 (quad-core). Figure 9 (A) shows the performance increase in the execution time. Figure 9 (B) shows an increase in efficiency compared to other platforms. Over TX1 and TX2 each channel is run as a separate thread using pthreads, since there are 4 processor cores in TX1 and TX2 CPUs, 4 different processes are created using MPI (Message Paring Interface) and each process is mapped with 25 threads to ensure maximum parallelism that can be achieved over TX1 and TX2 CPUs.

4.6 Comparison with existing work

We compare our work with a similar existing work [7]. We consider only 1 chain Parallel Tempering (PT) from [7], which is equivalent to MH sampler algorithm. Table 2 summarizes the comparison, where MC3A achieves 4.6× higher throughput when compared with [7].

5 CONCLUSION

In this work, we proposed a programmable, high-throughput and energy-efficient manycore for MCMC algorithms - named "MC3A" that contains customized instructions, and an architecture that is modified to speed-up the MCMC algorithms. The proposed work is then fully placed and routed using 65nm 1V, TSMC CMOS technology. The VLSI post-layout results show that the proposed MC3A has a throughput of 23 Msps which is 6× higher over the PENC manycore, and consumes an energy of 10.6 μJ per sample generation per cluster which is a 4× improvement over PENC. Also, we implemented the MH algorithm on NVIDIA Jetson TX1 and TX2 CPUs where MC3A achieves 195× and 191× speedup respectively, and 808× and 726× better energy efficiency respectively.

REFERENCES

[1] T. Abtahi, A. Kulkarni, and T. Mohsenin. 2017. Accelerating convolutional neural network with FFT on tiny cores. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4. https://doi.org/10.1109/ISCAS.2017.8050588

[2] T. Abtahi, C. shea, A. Kulkarni, and T. Mohsenin. 2018. Accelerating Convolutional Neural Network with FFT on Embedded Hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2018).

[3] Narges Bani Asadi et al. 2008. Reconfigurable computing for learning Bayesian networks. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. ACM, 203–211.

[4] N. Attaran, A. Puranik, J. Brooks, and T. Mohsenin. 2018. Embedded Low-Power Processor for Personalized Stress Detection. *IEEE Transactions on Circuits and Systems II: Express Briefs* PP, 99 (2018), 1–1. https://doi.org/10.1109/TCSII.2018.2799821

[5] Ter Braak and Cajo JF. 2006. A Markov Chain Monte Carlo version of the genetic algorithm Differential Evolution: easy Bayesian computing for real parameter spaces. *Statistics and Computing* 16, 3 (2006), 239–249.

[6] R. Gutierrez, V. Torres, and J. Valls. 2012. Hardware Architecture of a Gaussian Noise Generator Based on the Inversion Method. *IEEE Transactions on Circuits and Systems-II* 8 (2012), 501–505.

[7] Morteza Hosseini et al. 2017. A Scalable FPGA-based Accelerator for High-Throughput MCMC Algorithms. In *IEEE Symposium on Field- Programmable Custom Computing Machines (FCCM)*.

[8] Z. Ji, Y. Xia, Q. Sun, Q. Chen, D. Xia, and D. D. Feng. 2012. Fuzzy Local Gaussian Mixture Model for Brain MR Image Segmentation. *IEEE Transactions on Information Technology in Biomedicine* 16, 3 (2012), 339–347. https://doi.org/10.1109/TITB.2012.2185852

[9] A. Kulkarni, T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin. 2017. PACENet: Energy efficient acceleration for convolutional network on embedded platform. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4.

[10] A. Kulkarni, T. Abtahi, E. Smith, and T. Mohsenin. 2016. Low Energy Sketching Engines on Many-Core Platform for Big Data Acceleration. In *Proceedings of the 26th Edition on Great Lakes Symposium on VLSI (GLSVLSI '16)*. ACM, New York, NY, USA, 57–62. https://doi.org/10.1145/2902961.2902984

[11] A. Kulkarni, A. Page, N. Attaran, A. Jafari, M. Malik, H. Homayoun, and T. Mohsenin. 2017. An Energy-Efficient Programmable Manycore Accelerator for Personalized Biomedical Applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* PP, 99 (2017), 1–14.

[12] Scott M. Lynch. 2007. Introduction to Applied Bayesian Statistics and Estimation for Social Scientists. (2007), 107–130.

[13] Alireza S. Mahani and Mansour T.A. Sharabiani. 2014. SIMD Parallel MCMC Sampling with Applications for Big-Data Bayesian Analytics. *Computational Statistics and Data Analysis* (2014), 1–41. https://doi.org/10.1016/j.csda.2015.02.010

[14] Lahir Marni, Morteza Hosseini, Hopp Jennifer, Mohseni Pedram, and Tinoosh Mohsenin. 2018. A Real-Time Wearable FPGA-based Seizure Detection Processor Using MCMC. In *IEEE proceedings of International Symposium on Circuits and Systems (ISCAS)*.

[15] G. Mingas and C. S. Bouganis. 2012. A Custom Precision Based Architecture for Accelerating Parallel Tempering MCMC on FPGAs without Introducing Sampling Error. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 153–156. https://doi.org/10.1109/FCCM.2012.34

[16] Grigorios Mingas and Christos-Savvas Bouganis. 2012. Parallel tempering MCMC acceleration using reconfigurable hardware. In *International Symposium on Applied Reconfigurable Computing*. Springer, 227–238.

[17] A. Page, N. Attaran, C. Shea, H. Homayoun, and T. Mohsenin. 2016. Low-Power Manycore Accelerator for Personalized Biomedical Applications. In *Proceedings of the 26th Edition on Great Lakes Symposium on VLSI (GLSVLSI '16)*. ACM, New York, NY, USA, 63–68. https://doi.org/10.1145/2902961.2902986

[18] Grigorios Mingas Shuanglong Liu and Christos-Savvas Bouganis. 2016. An Unbiased MCMC FPGA-based Accelerator in the Land of Custom Precision Arithmetic. *IEEE TRANSACTIONS ON COMPUTERS* PP, 99 (2016), 1–1.