LiteNoC: Developing Low-Cost Network-on-Chip for Deep Neural Networks

Khoa Ho*
khoaho@my.unt.edu
University of North Texas, USA

Siamak Biglari*
siamakbiglariardebili@my.unt.edu
University of North Texas, USA

Justin Garrigus
justingarrigus@my.unt.edu
University of North Texas, USA

Hui Zhao
hui.zhao@unt.edu
University of North Texas, USA

Saraju Mohanty
saraju.mohanty@unt.edu
University of North Texas, USA

Abstract

With the rapid advance in Deep Neural Networks (DNNs), GPU's role as a hardware accelerator becomes increasingly important. Due to the GPU's significant power consumption, developing high-performance and power-efficient GPU systems is a critical challenge. DNN applications need to move a large amount of data between memory and the processing cores which consumes a great amount of power in the on-chip network. Prior data compression techniques have been proposed for network-on-chips to reduce the size of data being moved and can thus save power, but these techniques are usually lossless because they target general-purpose applications that are not resilient to errors. On the other hand, DNN applications are well known to be error-resistant, which makes them good candidates for lossy compression.

In this work, we propose an NoC architecture that can reduce power consumption without compromising performance and accuracy. Our technique takes advantage of the error resilience of DNNs as well as the data locality in the exponent field of DNN's floating-point data. Each data packet is reorganized by grouping data with similar exponents, and redundant exponents are sent only once. We further compress the mantissa fields by appropriately selecting "proxy" values for data sharing the same exponent. Our evaluation results show that the proposed technique can effectively reduce the data transmissions and lead to better performance and power trade-offs without losing accuracy.

Keywords

GPU, NoC, Data Compression, Low-cost NoC, Machine Learning, DNN

ACM Reference Format:
Khoa Ho, Siamak Biglari, Justin Garrigus, Hui Zhao, and Saraju Mohanty. 2025. LiteNoC: Developing Low-Cost Network-on-Chip for Deep Neural Networks. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Both authors contributed equally to this research.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Originally developed for accelerating graphics processing, GPUs have been widely used today for speeding up machine learning applications such as deep neural networks. GPUs allow applications to run with extreme efficiency by executing thousands of threads in parallel. New GPU architectures such as the NVIDIA A100 Tensor Core GPU have been specifically developed to support deep learning, and several deep learning frameworks such as PyTorch and TensorFlow have been developed to support GPU's programming model. These frameworks abstract CUDA's programming complexity and make GPU programming more accessible to support deep learning applications. As a major accelerator, GPUs have made great contributions to the rapid advances in emerging technologies such as natural language processing, smart healthcare, and autonomous driving.

As the neural network models improve in their capabilities, the size of these models also grows exponentially. Some models had their size grow by hundreds of times in the last few years. Although the on-chip memory also grew larger and can accommodate more data, the on-chip network bandwidth is becoming a critical performance bottleneck [2, 5]. Unlike in many-core CPU systems that leverage banked last-level caches to increase memory bandwidth, all the processing cores in a GPU need to get their data from only a few memory controllers. Inefficient data communication can greatly degrade a GPU's performance, making it impossible to reach its peak computing power.

As a solution to the on-chip data communication challenge, network-on-chips (NoC) have been extensively studied in CPU-based many-core systems [10]. Prior work proposed designs for router micro-architectures, routing algorithms, flow control, and more. Several GPU NoC architectures have also been proposed [2, 5, 7, 8]; however, most of them target general-purpose computing rather than neural network models. There are several major differences in the data communication between neural networks and other general applications: first, data movement is significantly larger in neural networks; second, neural networks compute using floating-point data, while other GPU applications may run with integer types. Third, neural networks are more error resilient. It has been shown that neural networks can tolerate some errors and losing some data can still result in good accuracy [11, 16, 18].

Data compression is an ideal technique that can take advantage of the characteristics of neural networks [6, 14, 17, 18]. Compression can greatly reduce the data size and can thus improve the data communication throughput. However, conventional compression

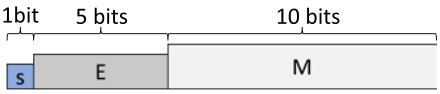

Figure 1: 16-bit floating point data representation.

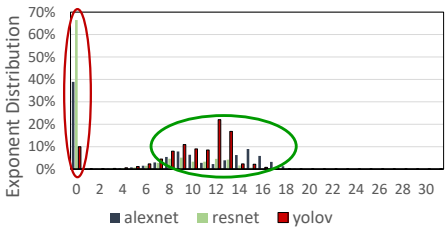

Figure 2: Exponent distribution.

techniques are usually very complicated and are not feasible for NoCs due to limited on-chip resources. Another design challenge is the compression and decompression latency, which needs to be small enough for the fast-running environment on-chip. Prior compression architectures have been proposed for NoCs. For example, a base-delta scheme was proposed to reduce packet sizes by sending the difference between consecutive words[19]. However, such a technique belongs to lossless compression and only works well with integer-type applications where regular patterns between consecutive words can be found. By far, cost-effective data compression targeting neural networks remains largely unexplored.

In this work, we propose to overcome the DNN communication bottleneck through a low-cost data compression scheme called LiteNoC. Our technique is based on the following observations: (1) floating-point data packets of neural networks exhibit different locality in the exponent fields and mantissa fields, as exponents are clustered around a small range of values while mantissas are randomly distributed; and (2) data of different magnitude has varying impact on the execution results, as data with larger magnitude is more important to model accuracy.

We make the following contributions:

- We develop a lightweight but effective compression technique tailored for neural network models. We cluster data words in each packet by their exponent values and remove redundant transmission of the same exponents.
- After compressing the exponent fields, we further compress the mantissa values through approximation. We select the mantissas with larger magnitude in a stochastic way as a proxy of a selected group.
- We performed a detailed evaluation of the proposed compression scheme using popular neural networks. Our evaluation results show that the proposed compression schemes can improve performance by a maximum of 61% and save energy by a maximum of 32% while maintaining the same accuracy.

2 LiteNoC compression mechanism

2.1 Compressing Shared Exponents

Today's DNN applications often use 16-bit (half-precision) floating-point formats to represent their data. This choice is driven by the

Figure 3: A packet demonstration with common exponents.

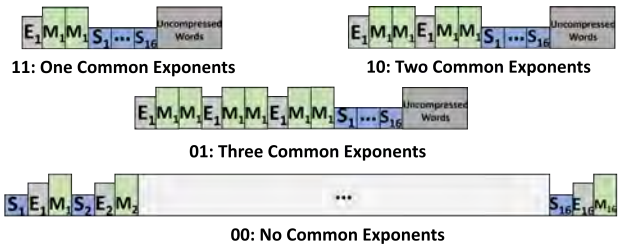

Figure 4: LiteNoC with different packet structure.

rapid growth in the size of DNN models, coupled with the critical need to efficiently manage storage [14, 15, 17, 18]. By reducing the data size, not only does the area overhead decrease, but there can also be significant savings in energy.

As shown in Figure 1, the IEEE754-style 16-bit half-precision floating-point format allocates 1 bit to the sign, 5 bits to the exponent, and 10 bits to the mantissa. In Figure 2, we characterized the complete range of exponents in three neural networks: Alexnet, ResNet, and YOLO. Since the half-precision floating-point format has an exponent field containing 5 bits, the exponent values range from 0 to 31. From this figure, we can observe that there are two regions that the exponent values cluster around. The red region on the left represents exponent values equal to zero, and we can see the three neural networks have 40%, 60% and 10% of exponent values match this. The green region on the right contains a significant portion of exponents that fall between 7 and 17. It is rare to see exponent values that fall outside of these two regions. This indicates that there is locality of the exponent values in packet data and it's highly likely the data in a given packet share some common exponents.

To take advantage of this locality, we can group data words inside a cache block if they share a common exponent. Rather than transmitting multiple copies of the same exponents, only one exponent needs to be sent. This motivated us to develop our first compression technique to extract the common exponents.

Figure 3 illustrates how our exponent compression technique works. The example packet contains sixteen 16-bit words. Among them, 12 words share a common exponent e_1 as follows:

$$e_1 \text{ and } M_1, \quad e_1 \text{ and } M_2, \quad \ldots, \quad e_1 \text{ and } M_{12}$$

12 redundant copies of e_1 in this packet are sent in every hop to the destination. To exploit this redundancy, we can only send one e_1 instead of twelve copies, saving more network bandwidth by sending fewer bits. The total number of bits in this packet is given by: 16 words × 16 bits = 256 bits. Using our technique, we only transmit 5 bits for e_1 instead of the original 60 bits.

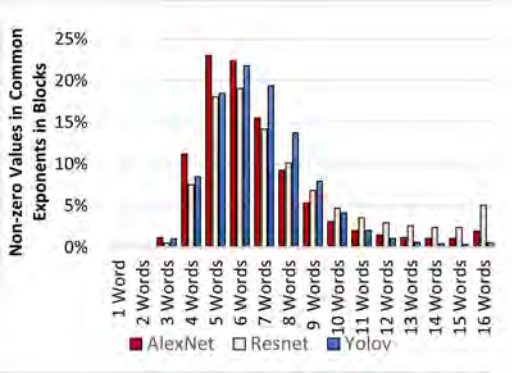

Figure 5: Number of values in Packets with same Exp(Non Zero).

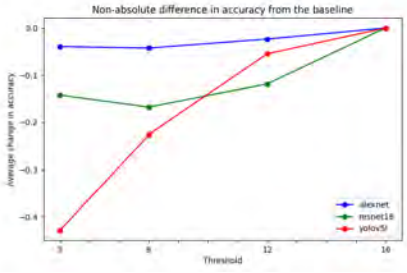

Figure 6: Threshold effect on accuracy.

During our compression, it is possible that the words in one packet sharing the same exponent are not adjacent to each other. In this case, we need to reorder the words to extract the common exponent. We use a 2-bit mask to keep track of which shared exponent one specific word is grouped with. We allow a maximum of 3 shared exponents to be extracted, with one bit used in the mask to represent the unextracted situation. In this way, we do not need to send redundant exponents, but only need to send this bitmask after compression. During decompression, we can recover which exponent a specific word will be associated with by using the bitmask. The design detail is illustrated in Figure 7, and we will discuss it in more detail in Section 3.

2.2 Compressing Mantissa through Stochastic Approximation with a Proxy

To further compress packet data, we investigate the mantissa, which it has heavy in the bits of FP16. In floating-point data, exponent values determine the scale or magnitude of a number, and thus, they are considered more critical to the accuracy. On the contrary, mantissa values are less important to accuracy, making them a candidate for truncating or rounding to save more bits. This characteristic provides us an opportunity to further compress packet data after we extract common exponents. Ideally, we want to send the fewest number of mantissa bits as a "proxy" within a group of words sharing one common exponent. It has been shown that data values with larger magnitude are more important to preserve the accuracy [11, 16]. If we can remove other mantissas and send only the mantissa with the maximum absolute value, and approximate other mantissas using this value, then we can further compress the data transmitted. We tried to apply this technique to packets that share one common exponent among 16 words, and we got 100 % accuracy. However, based on our observations shown in Figure 2, it is apparent that most of the packets with common exponents predominantly contain zero values in the left red region. Therefore, achieving perfect accuracy with a single-proxy mantissa is due to the zero-packets.

Figure 5 shows that, for non-zero shared exponents, the majority of packets have 4 to 10 words sharing an exponent. For instance, in AlexNet, more than 22% of the packets contain at least one group of 5 words sharing the same exponent. To effectively leverage our method, we need to find a threshold to filter packets containing enough words sharing one exponent. At the same time, we also need to consider the overhead of added control bits during the compression step.

As we modified the experiment to exclude packets with full zeroes or with a high number of zeroes, we also reduced the threshold of the required count of exponent-sharing words to include more compressible packets. That is because, with a smaller threshold, there is a higher likelihood of discovering more common exponents in packets, along with a higher number of non-zero values. However, during that process, the accuracy started to drop considerably from threshold=12, as depicted in Figure 6. And at that point, it still hasn't reached the point where the majority of packets can be compressed. With further testing for thresholds as low as 5-exponent-sharing-word, we can confirm that decreasing the threshold, while using a single proxy mantissa, would lead to a non-marginal decrease in accuracy. We investigated and deduced the reason for this loss of accuracy is primarily due to the non-uniformity of mantissa values. From our collected data and a bit-wise perspective, we observed that while exponent values tend to be local, mantissa values are more random. This variability in mantissa caused a high loss of accuracy when using one maximum value to represent other values.

Therefore, to address the accuracy issue, we must devise a less lossy method for compressing FP16 values. With that, we opted for two mantissas per group of shared exponent, namely mantissa with MSB=1 and MSB=0, instead of a single max mantissa. Simultaneously with this change, a smaller threshold would be required to save redundant bits. Consequently, we chose 5 as our threshold, after detailed calculation of bit-level cost and savings. A threshold equal to 5 would limit the number of sharing groups to 3, allowing for optimal use of the 2-bit mask for exponent. We have added a 1-bit mask for the mantissa since we use 2 mantissas per group now. So the total fixed overhead cost is 48 bits (16x 3-bit). Each group of 5 will cost 25 bits for one E-M-M group while saving a minimum of 75 bits. Therefore, with a threshold equal to 5, the compression system can at least break-even in the worst case, where there is only one group of 5 words sharing a common exponent and the others all have uncommon exponents. Reducing the threshold to 4 would cause the system to lose more bits than it can save during the worst-case scenario. Therefore, threshold=5 is the lower-bound limit for this compression system to achieve any potential savings.

Then, to optimize the system further to avoid wasting bits for group-place-holder and their bit-masks wherever there are fewer than three groups of exp-sharing-words, we added 2 control bits in

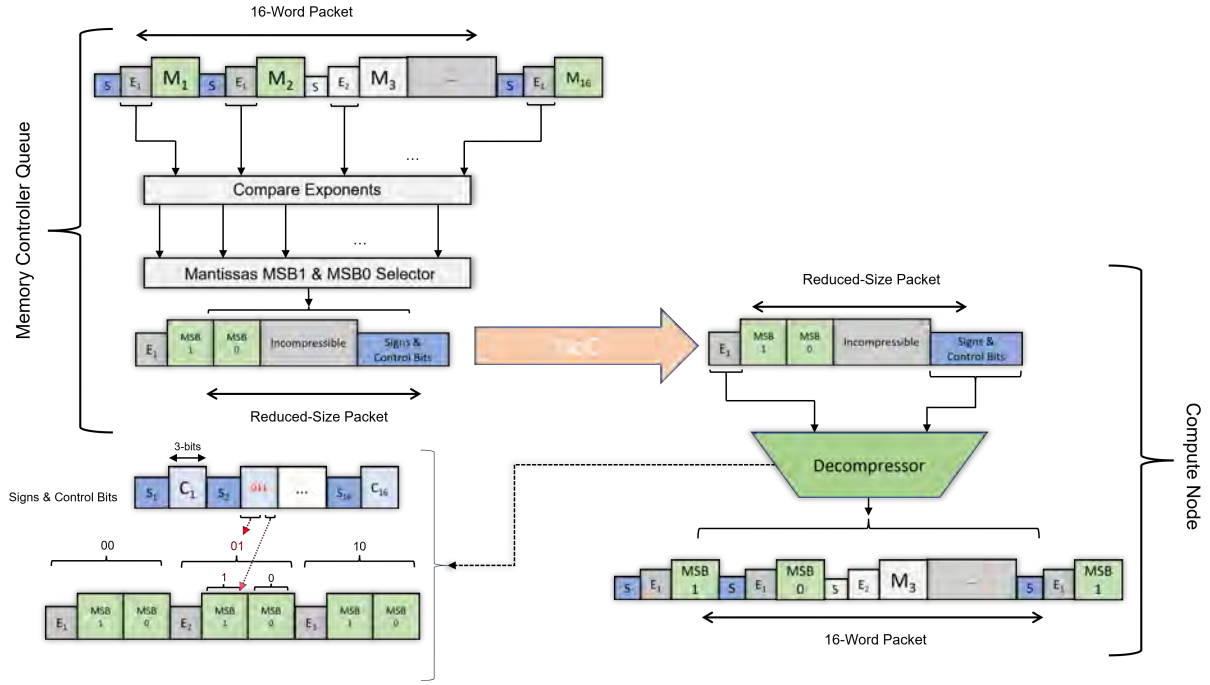

Figure 7: LiteNoC With Different Packet Structure.

each packet to specify what type of packet is this packet is, as shown in Figure 4 beside other details on the structure of the compressed packets.

In the compressed structure, we gathered all e_1s inside a packet that meets the sharing-word-count threshold and grouped them. Within each group, we place the maximum mantissa larger than 512 (MSB=1) in M_1, and for the other max-mantissa slot, we select the maximum mantissa among those with MSB=0. We use the M_1 and M_2 for e_1 for first group of common exponent. The same structure applies to the other two groups.

In the previous approach, where we selected one max mantissa, it is evident that more accuracy is lost. An example illustrating the loss of accuracy with one max mantissa is a scenario where, in a packet with 10 words sharing a common exponent, one mantissa is larger than 512, and the other 9 mantissas are much smaller than the maximum value. In this case, selecting the maximum mantissa as a proxy to represent others would result in significant accuracy loss due to the large difference between these values.

In our two maximum mantissa approach, we utilize a k-means approach to cluster the mantissas. We then add the appropriate control bits in the packet for each word to enable the selection of a mantissa that is close to its original value as the proxy mantissa in the compute node.

3 Architectural Support

3.1 Compression Module

As discussed previously, we first select a threshold as the minimum size of each sharing-group in each packet. To preserve accuracy, we employ 2 mantissas as deputies to represent other mantissa values within a group. Figure 4 illustrates four packet structures, and we use a 2-bit flag to represent the structure used by each packet.

The first packet structure is employed when there is only one exponent sharing group containing data words equal to or greater than the threshold. The best-case scenario is when we have 16 words sharing one exponent in one packet, and the worst case is when we have only 5 words sharing one exponent. The second structure is for the scenario with 2 exponent sharing groups, and the third is for 3 exponent sharing groups.

For example, to compress a packet with three exponent sharing groups equal to or exceeding the threshold, we utilize the third packet structure. In this case, we take the shared exponent and select two mantissas: the first one is the maximum mantissa among all mantissas that have a '1' in their MSB. The second mantissa is the maximum mantissa among all mantissas having a '0' in their MSB. The logic behind this operation is as follows: since magnitude affects the accuracy, only approximating the single MSB bit can cause more errors. Therefore, we split the words into two groups, MSB with 1 and 0. Then inside each group, we select the proxy with the largest magnitude, i.e., the maximum mantissa inside each group. In this way, we can lower the error injected by always selecting the maximum mantissa. We do the same for the other two groups of common exponents as well. If there are non-compressed words, they will be sent in their original form.

Our compressed packet contains the 2-bit control for packet-structure, followed by e_1, m_1, and m_2 in the compressed structure. The sign bits remain in their relative positions for easy decoding at the destination, and uncompressed words are placed in their original ordering. In addition to this structure, we add 48 additional bits to mask the words' group to select the appropriate exponents and mantissas at the destination.

Figure 8: Total Packet Latency.

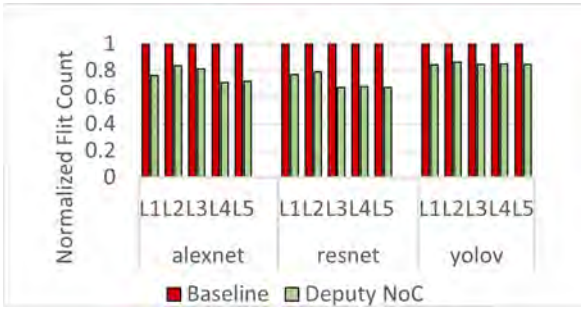

Figure 9: Total Flit.

For this compression, given the memory-intensive nature of DNN data, we leverage the already congested queues at the ejection port of all MCs to conceal the overhead of compression and compress the packets before entering the ejection buffer. In the next subsection, we will describe the decompression module.

3.2 Decompression Module

As depicted in Figure 7, our approach aims to further capitalize on the congested queues in Memory Controllers (MCs) and mitigate compression overhead. The compression hardware initiates by selecting exponents within the packet and identifying equal ones. Subsequently, mantissas with MSB equal to 0 and 1 are chosen to construct the compressed packet separately. Once compressed, the packets are transmitted to their respective destinations.

Upon reaching the destination, the compressed packet must undergo decompression to revert to the original structure shown in Figure 7. The decompressor relies on control bit-masks assigned to each word's position, which specify the exponents and mantissas associated with each word. This utilization of bit-masks ensures an effective and accurate decompression of the packet.

4 Experimental Evaluation

We use GPGPU-sim[3] as the simulator to evaluate LiteNoC. We use three well-known DNN applications: AlexNet[9], ResNet[4], and Yolov5l[13] as our benchmarks. Table 1 provides details of our system configurations.

From Figure 9, we can observe a reduction in the flit count by more than 30% in some cases. The flit compression ratio, as shown in Figure 10, exceeds 1.4 in many instances. This decrease in flit size results in reduced NoC traffic. Reducing data transmission not

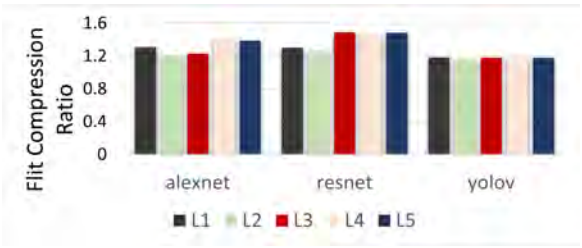

Figure 10: Flit Compression Ratio.

Table 1: System Configurations

Parameter	Value
Core Count	48
Clock Frequency	1.2GHz
L1 I & D Cache	256-way, 128KB
L2 Cache	32-way, 16 × 256KB,
# Memory Controllers	16
Network Topology	8 × 8 Mesh
Routing	Dimension-Order XY
Buffer	4 VCs per port, 8-flit depth per VC
Packet Length	5 flits, each flit 8 bytes

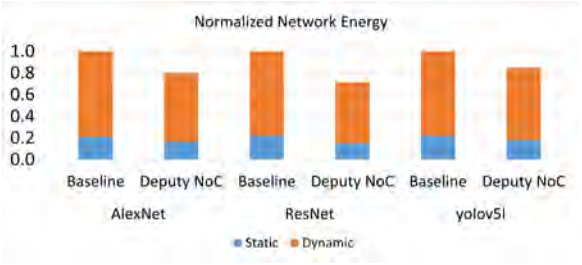

Figure 11: Total Dynamic and Static Energy Consumption.

only alleviates congestion in the NoC but also contributes to lower packet latency.

Sending fewer flits in the NoC results in less congested routers and smaller queues in each router's buffers. This reduction in congestion means that flits can reach their destinations in less time. As shown in Figure 8, packet latency in all our DNN applications has decreased, especially with a reduction of more than 30% in AlexNet and ResNet.

The NoC is a major consumer of energy in a GPU processor. Therefore, reducing the activity in the NoC by sending fewer flits will lead to increased energy savings. As shown in Figure 11, the LiteNoC achieves a reduction of approximately 30% in energy compared to the baseline in ResNet. Our technique can significantly reduce both dynamic and static energy.

Figure 12 illustrates the IPC in each layer of our DNN applications. In particular, our proposed compression method achieves an improvement in IPC across all layers. In certain ResNet layers, we observe that the IPC improvements slightly exceed 60 % compared to the baseline and BDI.

As mentioned above, the locality of the data in DNN applications makes them ideal candidates for investigating comparison

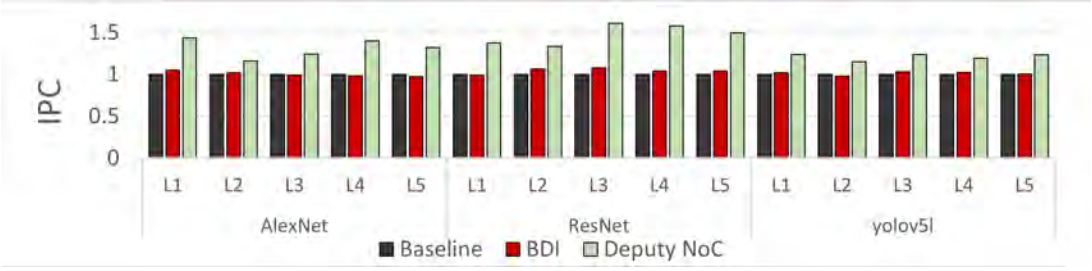

Figure 12: Normalized IPC Comparison between Baseline, BDI, and LiteNoC.

techniques. Additionally, their error-tolerant nature allows for a deeper exploration of methods to compress this data. Our proposed method leverages the data intensity in MCs' queues to compress local data words in packets, sending smaller flits in the NoC. This approach mitigates congestion in the reply network, resulting in improved overall performance. However, our compression is lossy, and we need to evaluate the accuracy results as well. Due to the extremely long execution time using the simulator, we were not able to test a large number of input images of these neural networks. We can only test 8 images to full completion, which can finish in a reasonable time. Our results show that all of the images tested were classified correctly. Therefore, within the practical scope of testing, our compression scheme achieved 100% accuracy.

5 related work

BDI is a cache compression technique that exploits the low dynamic range of cache data [12]. Cache lines are represented in the form of two base values and an array of differences from the base values. BDI can achieve low decompression latency while still achieving a high compression ratio. Global BDI further extended BDI to compress caches with more complicated schemes, trying to increase the compression ratio [1]. Zhan et. al. proposed a delta compression [19] for network-on-chips that conducts data encoding before injection and decoding before network ejection. It is an extension of the BDI compression to NoCs. However, these techniques rely on specific patterns in the packet data to work. Other than the conventional quantization and pruning techniques to compress neural network models, techniques on data representation format have been proposed [14, 17]. Even though our technique proposes to compress the original floating point data, our technique can also work with these data representations.

References

[1] Alexandra Angerd, Angelos Arelakis, Vasilis Spiliopoulos, Erik Sintorn, and Per Stenström. 2022. GBDI: Going Beyond Base-Delta-Immediate Compression with Global Bases. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA).*

[2] A. Bakhoda, J. Kim, and T.M. Aamodt. 2010. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software.* 163–174. doi:10.1109/ISPASS.2009.4919648

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[5] H. Jang, J. Kim, P. Gratz, K. Yum, and E. Kim. 2015. Bandwidth-Efficient On-Chip Interconnection Designs for GPGPUs. In *Proceedings of the 52nd Annual Design Automation Conference (DAC).*

[6] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A Study of BFLOAT16 for Deep Learning Training. arXiv:1905.12322 [cs.LG]

[7] H. Kim, J. Kim, Wong. Seo, Y. Cho, and S. Ryu. 2012. Providing Cost-effective On-Chip Network Bandwidth in GPGPUs. In *Proceedings of the IEEE International Conference on Computer Design (ICCD).*

[8] Kyung Hoon Kim, Rahul Boyapati, Jiayi Huang, Yuho Jin, Ki Hwan Yum, and Eun Jung Kim. 2017. Packet Coalescing Exploiting Data Redundancy in GPGPU Architectures. In *Proceedings of the ACM International Conference on Supercomputing (ICS).*

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS).*

[10] R. Mullins, A. West, and S. Moore. 2004. Low-latency virtual-channel routers for on-chip networks. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 188–197. doi:10.1109/ISCA.2004.1310774

[11] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. 2018. Value-aware Quantization for Training and Inference of Neural Networks.

[12] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT).*

[13] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[14] Bita Rouhani and et.al. 2020. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20).* Curran Associates Inc., Red Hook, NY, USA, Article 861, 11 pages.

[15] Bradley McDanel Sai Qian Zhang and H.T. Kung. 2022. FAST: DNN Training under Variable Precision Block Floating Point with Stochastic Rounding. In *Proceedings of the 28th International Symposium on High Performance Computer Architecture (HPCA).*

[16] Charbel Sakr, Steve Dai, Rangharajan Venkatesan, Brian Zimmer, William J. Dally, and Brucek Khailany. 2022. Optimal Clipping and Magnitude-aware Differentiation for Improved Quantization-aware Training.

[17] Thierry Tambe and et.al. 2020. AdaptivFloat: A Floating-point based Data Type for Resilient Deep Learning Inference. arXiv:1909.13271 [cs.LG]

[18] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training Deep Neural Networks with 8-Bit Floating Point Numbers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18).* Curran Associates Inc., Red Hook, NY, USA, 7686–7695.

[19] Jia Zhan, Matt Poremba, Yi Xu, and Yuan Xie. 2014. No: Leveraging Delta Compression for End-to-End Memory Access in NoC Based Multicores. In *19th Asia and South Pacific Design Automation Conference (ASP-DAC).*