Universal Neural Network Acceleration via Real-Time Loop Blocking

Jiaqi Zhang, Xiangru Chen, Sandip Ray
Department of Electrical and Computer Engineering
University of Florida
{jiaqizhang, cxr1994816}@ufl.edu, sandip@ece.ufl.edu

***Abstract*—There is a recent trend that the DNN workloads and accelerators are increasingly heterogeneous and dynamic. Existing DNN acceleration solutions fail to address these challenges because they either rely on complicated ad hoc mapping or clumpy exhaustive search. To this end, this paper first proposes a formalization model that can comprehensively describe the accelerator design space. Instead of enforcing certain customized dataflows, the proposed model explicitly captures the intrinsic hardware functions of a given accelerator. We connect these functions with the data reuse opportunities of the DNN computation and build a correspondence between DNN loop blocking and accelerator constraints. Based on this, we implement an algorithm that efficiently and effectively performs universal loop blocking for various DNNs and accelerators without manual specifications. The evaluation shows that our results manifest 2.1x and 1.5x speedup and energy efficiency over dataflow-defined algorithm as well as significant improvement in blocking latency compared with search-based methods.**

Keywords-neural network; loop blocking; accelerator

I. INTRODUCTION

The revival of machine learning has led to an explosion of deep neural network (DNN) models. Compared with the general-purpose processors, customized accelerators demonstrated supreme energy efficiency in processing these both computation- and memory-intensive workloads. Consequently, the past few years also witnessed a burst of DNN accelerators [1]–[3]. With the massive and ever-emerging DNN models and accelerators deployed in our daily life, two challenges can be expected. First, both the DNNs and accelerators appear in various architectures, including the layer type, size, dimension, and accelerator shape, dataflow, etc. Second, since more AI tasks are executed on the edge, there will be more dynamic DNN acceleration requests. On one hand, different models are called on the fly when a certain function is evoked. On the other hand, the mobility of users makes it a common case to migrate the service among the edge and fog nodes.

Considering the heterogeneity and dynamic in DNN acceleration, there is a timely demand for a method that universally optimizes the acceleration of any DNN computation on a given accelerator in real time. There are existing works [4] that address the heterogeneity in DNN computation by uniformly parameterizing different layers. However, since there lacks a standard to regularize the heterogeneous accelerators and directly connect DNN computation with the accelerators, existing works [5]–[7] still require ad hoc optimization or time-consuming exploration of the large design space.

Driven by this motivation, this work proposes to address the need for a systematic approach to solving the universal DNN acceleration problem. Inspired by previous work [4] that generalized DNN layers into standard general convolution (GCONV) operations, we further propose to formalize the accelerators. Compared with other models that manually specify the dataflows, our formalization focuses on the intrinsic hardware functions of the accelerators. This further allows us to build a direct connection between DNN and the accelerator, which is invariable despite the heterogeneity. Therefore, the dataflow can be automatically and flexibly determined, and the loop blocking is explicitly dictated by the hardware function and resource constraints. Based on the proposed model, we implement a universal neural network acceleration algorithm that automatically fills the DNN computation loops into the accelerator with optimized data reuse on the fly.

In summary, this paper makes the following contributions:

(1) We propose a dataflow-independent but function-based DNN accelerator formalization that can comprehensively model the accelerator design space.

(2) We build a direct connection between DNN computation and the formalized accelerator model that explicitly describes the blocking constraints. Based on this, we implement a universal algorithm for DNN acceleration.

(3) The experiments on four popular DNNs and three various accelerators show that the proposed method is both effective and efficient in DNN acceleration.

II. BACKGROUND AND RELATED WORKS

A. DNN Computation

This work is based on the GCONV model proposed in [4]. A 1-D GCONV operation is described as a nest of four loops, i.e., *g* for groups of inputs, *op* for groups of kernels, *ks* for kernel size and *opc* for output size as in Fig. 1. This 1-D GCONV can be scaled up to multiple dimensions to represent all the computation in DNNs, even for traditionally non-convolution layers. For example, a convolution and a local response normalization layer are described by GCONV as in Fig. 2.

GCONV model preserves all the data reuse opportunities in DNNs. There are parallel and overlap reuses in each dimension. Specifically, the inputs and kernel parameters are reused

```
for g.i in range(N[dim][g]):
  for op.i in range(N[dim][op]):
    for opc.i in range(N[dim][opc]):
      for ks.i in range(N[dim][ks]):
        O[g.i][op.i][opc.i] += I[g.i][opc.i+ks.i] × K[g.i][op.i][ks.i]
```

Fig. 1. 1-D GCONV model which can be scaled up to multiple dimensions. N[dim][param]: the value of the parameter in the dimension. O: output. I: input. K: kernel parameter. The stride and padding are omitted.

(a) N[C][*ks*]=64, N[C][*op*]=128, N[H][*opc*]=28, N[H][*ks*]=3, N[W][*opc*]=28, N[W][*ks*]=3

(b) N[C][*opc*]=64, N[C][*ks*]=5, N[H][*opc*]=28, N[W][*opc*]=28

Fig. 2. **(a)** Convolution layer with 128 output channels and 3×3 kernels. **(b)** LRN layer that normalizes on 5 adjacent channels. There are 64 input channels and 28×28 outputs in both. C: channel. H: height. W: width. Parameters equal to 1 are omitted.

by computation in loops *op* and *opc* respectively and the outputs are reused through reduction in *ks* (parallel reuse). In addition, the computation of neighboring outputs share the inputs overlapped (when *ks* > *s* and *opc* > 1) in the convolution windows (overlap reuse). Most efficient NN accelerators are designed to maximize these reuses.

B. Loop Blocking in DNNs

It is demonstrated by previous works [6][5] that different accelerator dataflows can be represented by the loop blocking, i.e., how the nested loop is exchanged and blocked in different spatial (PE) and temporal (memory) dimensions. For example, the local response layer in Fig. 2 can be mapped to an accelerator with 2-level blocking as

$$opc_W\_opc_H\_ks_C\_opc_C \mid opc_W\_opc_H\_ks_C\_opc_C = 14\_14\_3\_4 \mid 2\_2\_2\_16.$$

The representation is similar as in [6], where | separates the different blocking dimensions and the inner loops are on the left. Different loop blocking strategies lead to different performance and energy efficiency. Since the accelerators exhibit various constraints in the computation and memory resources and on-chip communication patterns, choosing the optimal loop blocking is a major challenge in DNN acceleration.

C. Related Works

Early accelerators are bound to fixed dataflows. To improve the utilization, later works [3][2] introduce flexible dataflows to accommodate the various DNN computation but they still require manually optimized dataflow for each layer. This further motivates the automated mapping of the DNNs. A common approach to automation is to enumerate all the possible loop blocking strategies within the dataflow constraints and choose the optimum based on certain evaluation model [5]–[7]. This is not fully automated because it requires manual directives on the dataflow constraints for different layers and accelerators. For example, the row-stationary dataflow of convolution layers in Eyeriss [3] is represented as S_C | Q_K | R_C_P ($ks_H\_ks_C \mid opc_H\_op_C \mid ks_W\_ks_C\_opc_W$ in GCONV) in row | column | scratchpad in [5]. These works also suffer from long running time since they can only use the dataflow constraints to prune the search space but not directly dictate the loop blocking. In [4], a uniform algorithm to automatically map any layer is proposed, but there still lacks a formalization to make the algorithm universal even for various accelerators.

TABLE I. DATA REUSE FUNCTIONS. THE FUNCTIONS IN ITALIC ARE EXPLICITLY MODELED IN OUR FORMALIZATION.

Type	Data	Loop	Dimension	Function
Parallel	Input Kernel	*op* *opc*	Spatial	Broadcast
			Temporal	Stationary
	Output	*ks*	Spatial	*Reduction*
			Temporal	Stationary
Convolution	Input	*ks* *opc*	Spatial	*Diagonal*
			Temporal	Stationary
			Spatial+Temporal	*Shift*

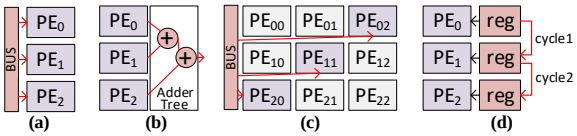

Fig. 3. Spatial reuse functions. **(a)** Broadcast. **(b)** *Reduction*. **(c)** *Diagonal*. **(d)** *Shift*.

III. DNN ACCELERATOR FORMALIZATION

Our accelerator formalization focuses on the PE array and memory characteristics that constrain the loop blocking.

A. PE Array Model

To reduce memory access, the DNN accelerators usually arrange a group of processing elements (PEs) into multiple dimensions and introduce exquisite interconnections among them to exploit data reuse. With little to no difference in the implementation of individual PEs, to model the PE array is indeed to abstract the interconnections for data reuse. Instead of binding the accelerator with a certain dataflow, we focus on the intrinsic functions of the interconnections. In Table I, we enumerate the possible hardware functions for data reuses discussed in Section II.A in a systematic manner.

First, the temporal reuse of all the data (both parallel and convolution reuse) can be intrinsically realized by keeping the data **stationary** in memory without any special hardware function. For example, *op* can be unrolled temporally to keep the input stationary in memory. The innermost temporal block keeps the data stationary in PEs. For output parallel reuse that requires reduction, it can also be performed in a stationary manner, i.e., read-reduce-write, using the reduction function within the PE. Therefore, the temporal data reuse functions are not explicitly modeled. Another function not explicitly modeled is the spatial parallel reuse of inputs and kernel parameters. It is realized by **broadcast** (Fig. 3(a)) through the data bus, which is commonly used in modern accelerators. We will clarify the enforced broadcast by the memory model in Section III.B. However, a ***reduction*** function (e.g., the adder tree in Fig. 3(b)) over a certain dimension needs to be clearly defined to dictate if the output parallel reuse can be exploited.

As for the convolution reuse, as shown in Fig. 4, the overlapping of convolutional windows provides reuse opportunities for the inputs, kernels, or outputs when the other two types

TABLE II. FORMALIZATION OF REPRESENTATIVE ACCELERATORS.

Accelerator	PE Model	Memory Model
Eyeriss [3]	**Dim1** (row): [12, A, A, N] **Dim2** (column): [14, N, A, N]	**Local storage:** K: [224, 1, False, False], I: [12, 1, False, False], O: [24, 1, False, False] **Global buffer:** K: [4096, 4, True, True], I: [51200, 1, True, True], O: [-2, 4, True, True]
Eager Pruning [2]	**Dim1** (PE array): [512, A, N, A] **Dim2** (subsystem): [4, A, N, N]	**Local storage:** K: [1, 1, False, False], I: [64, 512, True, False], O: [32, 32, True, False] **Global buffer:** K: [786432, 32, True, False], I: [786432, 32, True, False], O: [786432, 32, True, False]
TPU [1]	**Dim1** (row): [256, N, N, N] **Dim2** (column): [256, M, N, N]	**Local storage:** K: [1, 1, False, False], I: [1, 1, True, False], O: [1, 1, False, False] **Global buffer:** K: [2097152, 45, True, True], I: [12582912, 256, True, True], O: [-2, 256, True, True]

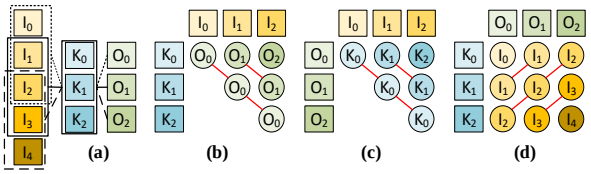

Fig. 4. Convolution reuse patterns for each type of data. **(a)** is an example of convolution. Each box on the inputs is the convolution window of each output. **(b)** to **(d)** indicate the convolution reuse (red line) of each type of data (round) when unrolling the other two (square).

of data are unrolled. Note that since not all the inputs are utilized for the computation of all the outputs (e.g., I0, I1, I3, I4), the unrolling of inputs (Fig. 4(b) and (c)) always results in ineffectual computation. Therefore, we focus on the unrolling of kernels and outputs with convolution reuse of inputs (Fig. 4(d)). To exploit the convolution reuse, both the kernels and outputs can be either spatially or temporally unrolled. When they are unrolled in different spatial dimensions, the inputs can be reused by ***diagonal*** broadcasting (Fig. 3(c)). And when one of them are unrolled temporally, the reuse of inputs should be implemented by ***shift*** function between the PEs (Fig. 3(d)), where the inputs are shared by different PEs in different cycles. Only accelerators with *diagonal* or *shift* functions implemented can exploit convolution reuse spatially.

With the data reuse functions defined, the PE array is described by a vector

[PE size, reduction, diagonal, shift]

in each dimension in our model, which clarifies the PE array size besides the three reuse functions. We explicitly indicate if the reuse functions are not-available (N), allowed (A) or mandatory (M). For example, although [2] provides an adder tree that connects all the PEs, *reduction* can be optionally performed, as long as the number of outputs does not exceed the capacity. This model is demonstrated to describe any multi-dimension rectangular PE array. Table II lists three of the most representative accelerators.

B. Memory Model

There are three kinds of data, i.e., inputs, kernel parameters and outputs (partial results) in the memory system. We model each level of memory for each kind of data as

[capacity (B), bandwidth (B/cycle), associativity in PE dim1, associativity in PE dim2, …]

as shown in Table II.

For simplicity, we assume all the accelerators implement 8-bit data and computation as in [1]. Minus values in the capacity and bandwidth represent sharing between different kinds of data. The associativity bits indicate if all the PEs in a certain dimension share the memory capacity and bandwidth. Usually, the local storage is exclusive and global buffers are shared by all. The association bit can also be utilized to model the mandatory broadcasting by forcing the PEs to share only one data. For example, the input is broadcast to the entire row of PEs (the first dimension) in [1]. Note that the capacity and bandwidth pertain to each associated memory.

Although Table II only lists the formalized model for local storage and global buffer, this model flexibly applies to different memory hierarchies. We include the entire memory hierarchy of each accelerator in the experiments.

IV. UNIVERSAL NEURAL NETWORK ACCELERATION

Based on the PE array and memory formalization models, we propose in this section a real-time universal loop blocking scheme.

A. Connection between the Formalized Accelerators and DNN Computation

Traditionally, there is no direct connection between the computation and the underlying hardware resources in the literature. Therefore, the accelerators are bound to a specific dataflow, i.e., fixed loop ordering and blocking, which are optimized based on the previously adopted stereotypes. In this work, we build a universal correspondence that systematically matches various DNN computation with various accelerators. With the GCONV model introduced in Section II.A and accelerator formalization proposed in Section III, the DNN workloads can be mapped to the accelerators with *rotatability*, and the goal is to *fill the accelerators with DNN loops to eliminate the idle resources*.

Rotatability. Normally, the accelerator possesses two or more spatial dimensions and a single hierarchical temporal dimension. The loops of any dimension of GCONV can be unrolled in any dimension of the accelerator. For instance, ks_C, ks_H and ks_W in Fig. 2 can all be unrolled in any blocking dimension with *reduction* function.

Filling blocking dimensions. Due to the different features of computation and memory resources, spatial and temporal blocking is not exactly the same. Spatially, each dimension is filled independently and the spare PEs will be left idle. Therefore, to improve the performance, the maximal number of PEs need to be occupied. Temporally, the dimension is unified though hierarchical. One temporal blocking can span multiple memory levels for different data types. For example, unrolling *op* may keep the input stationary in local scratchpad but need to fetch the kernel from the global buffer. For temporal blocking, it is not required to occupy all the available resources. Instead, it is preferred to reduce low-level memory access by fully exploiting the high-level memory.

Resource limitations. The data reuse functions in the accelerator PE array model determines whether a loop can be

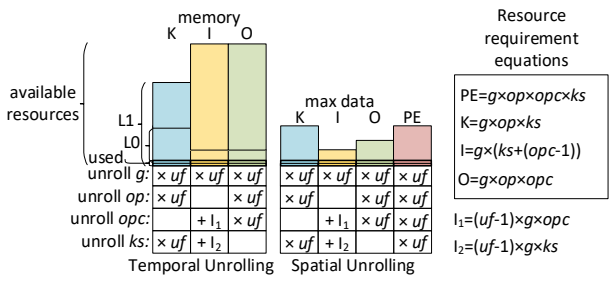

Fig. 5. Resource requirement in temporal and spatial unrolling. *uf*: unrolling factor. L0/L1: high/low memory levels.

unrolled in a certain dimension according to Table I. If the loop unrolling is allowed or mandatory, the key limitation for the blocking is the resource requirement. As shown in the "used" box in Fig. 5, when nothing is unrolled (unrolling factor is 1), only one PE and the storage for one data of each type are required.

For each blocking, the resource requirements expand with the unrolling factor differently based on the exact parameter. The temporal unrolling is constrained by the memory capacity and the spatial unrolling is constrained by both the PE size and the maximal data in that dimension, which is dictated by the available operator registers, i.e., the highest-level memory. Exploiting the data reuse opportunity can reduce the capacity requirement. The blank cells in Fig. 5 indicate the parallel reuse and +I1/I2 entries indicate the convolution reuse which is smaller than ×uf only when the unrolling factors of both *opc* and *ks* are above 1. Note that the resource requirements are multiplied on blockings of different parameters (e.g., *ks*, *op*, *opc*, *g*) and DNN dimensions (e.g., *C*, *H*, *W*).

Despite bandwidth's impact on the data loading time, it does not constrain the mapping of the network. Therefore, in the loop blocking, we only increase the data reuse in all the spatial dimensions to reduce the bandwidth requirement.

B. Loop Blocking Algorithm

With the accelerator formalization and resource requirement model, it is straightforward to design a loop blocking algorithm that universally applies to any DNN and any accelerator in real time.

For brevity, Fig. 6 only shows the critical steps of the algorithm. The key insight is to fill dimensions with the most exclusive functions first in case they will be left idle. **Step 1** first unrolls *opc* and *ks* of DNN dimensions with convolution reuse in spatial blocking dimensions with *diagonal* and *shift* functions. **Step 2** further unrolls *ks* in spatial dimensions with *reduction* functions. In these two steps, the mandatory functions are filled first to avoid underutilization. The blocking size is calculated directly by solving the maximal unrolling factor from the resource requirement in Fig. 5 based on the existing blockings and the constraints. Then in **Step 3**, if there is still DNN dimension with convolution reuse not unrolled, *ks* and *opc* of that dimension are both unrolled temporally for stationary reuse. In **Step 4** and **Step 5**, loops with reuse opportunities are further unrolled to fill the spare spatial dimensions and within the memory hierarchy (high-level first). For the unrolling in each step, there is no required order for the

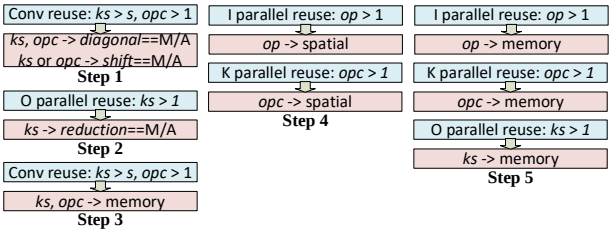

Fig. 6. Our implemented loop blocking algorithm. *g* is unrolled only when there is no other parameter.

TABLE III. BASELINES IN THE EVALUATION AND OUR METHOD. opc_B REPRESENTS THE LOOP FOR BATCH PROCESSING. THE DATAFLOW CONSTRAINTS SPECIFY THE LOOP ORDERINGS IN SPATIAL DIMENSION 1 | SPATIAL DIMENSION 2 | INNER TEMPORAL DIMENSION.

Method	Dataflow Constraints	Blocking
BL1	ER: $ks_H\_ks_C \mid opc_H\_op_C \mid ks_W\_ks_C\_opc_W$ ...	Search
BL2	EP: $ks_W\_ks_H\_op_C \mid ks_C \mid opc_W\_opc_H$ ... TPU: $op_C \mid ks_C\_ks_W\_ks_H \mid opc_B\_opc_W\_opc_H$ ...	Calculate
BL3	Reuse functions	Search
Our	Reuse functions	Calculate

dimensions or parameters. Our experiments show the order affects the average latency and data movement by less than 5%.

V. EVALUATION

A. Methodology

The algorithm in Section IV.B is implemented in Python to automatically perform loop blocking given a DNN workload and an accelerator defined using the formalization model. We build a simulator consistent with that in [4][3] to evaluate the running cycles and detailed data movement. The running cycles count in both the computation and data loading cycles. We generate the energy consumption using CACTI [8] for RAMs and Synopsys Design Compiler for registers based on the capacity and bandwidth.

We evaluate the loop blocking algorithm on the three accelerators in Table II. Four DNNs, i.e., AlexNet (AN) [9], ResNet-50 (R50) [10], YOLO [11], and Transformer (TFM) [12], are employed as the workloads. Although ER (Eyeriss) and EP (Eager Pruning) do not target DNNs with no convolution reuse opportunities, TFM is still evaluated on them for comprehensiveness. Since TPU is proposed for batch processing, we assume a mini-batch of 32 for the DNNs in TPU.

We adopt three baselines as listed in Table III. BL1 and BL2 rely on manually specified dataflow for each accelerator. BL1 searches the design space for the best loop blocking that obeys the dataflow, BL2 determines the unrolling factors based on the resource requirements as in our method. BL3 utilizes the reuse functions to dictate the search, which is expected to have a larger search space than BL1 and thus can result in the best result but the longest search time.

B. Experiment Results

Design space. Fig. 7 shows the performance and data movement energy of AN layer LRN1 on EP in various loop blockings. By comparing (a)/(c) with (b)/(d), it can be observed that the manual dataflow in BL1 and BL2 limits the design space. The function-defined constraint can generate

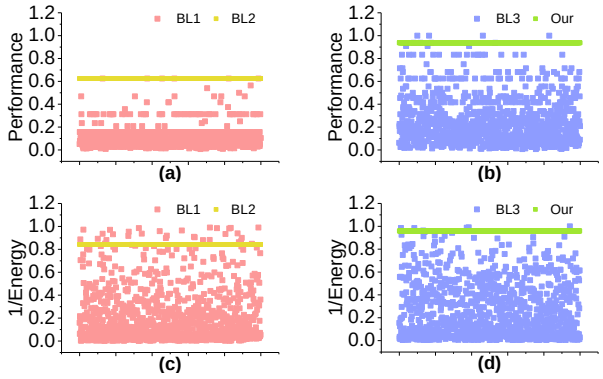

Fig. 7. Performance and data movement energy of AN Conv1 on EP. 1000 points are sampled for BL1 and BL3. The results in **(a)(b)** are normalized to the best in **(a)**. The results in **(c)(d)** are normalized to the best in **(c)**.

TABLE IV. TIME SPENT ON LOOP BLOCKING. THE RESULTS ARE AVERAGED ON THREE ACCELERATORS.

DNN	Our	BL3	BL3 Achieves Our
AN	0.05s	30m	4m
R50	0.14s	52m	18m
YOLO	0.09s	36m	14m
TFM	0.05s	19m	11m

loop blocking with better performance and energy efficiency. By comparing the results of BL3 and our method in (b) and (d), it shows that our method achieves near-optimal loop blocking.

Loop blocking results. To evaluate the loop blocking results of different methods, we focus on the performance and data movement energy. Fig. 8 shows the speedup results. Our method gains 1.7x and 2.1x speedup on average over BL1 and BL2 that rely on pre-defined dataflows. Compared with the optimum-guaranteed BL3, our method achieves 78% to 99% performance (an average of 88%). The results on the CNNs are slightly worse because they manifest more complicated reuse patterns. Fig. 9 shows the total data movement energy consumption by the entire system. Our method results in a 32% decrease and 14% increase in the data movement energy compared with BL2 and BL3.

Overhead. To emphasize the ability of real-time loop blocking, we list in Table IV the time spent on loop blocking by BL3 until convergence and our method. For fairness, we also include the time that BL3 takes to find a loop blocking with the performance and data movement similar to our results. The experiments are run on an Intel Xeon E5 CPU. For exact same layers, loop blocking is performed only once. As listed, our function-defined accelerator model and calculation-based loop blocking finishes all the DNNs within 0.2 seconds while the traditional search algorithm takes more than 1000x time to explore the design space and perform evaluation.

VI. CONCLUSION

In this era with increasing diversity and dynamic in both DNN computation and accelerators, a method that can efficiently and effectively map any DNN to a given accelerator is in timely demand. Driven by this need, we first build a model to formalize various DNN accelerators in a function-defined way and then a direct connection to eliminate the semantic gap between the DNN computation and accelerators. This allows uniform and fully automated real-time loop blocking algorithm for various workloads and accelerators.

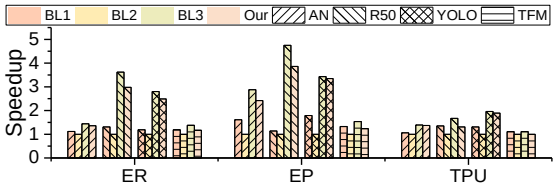

Fig. 8. The speedups of the entire DNNs, normalized to BL2.

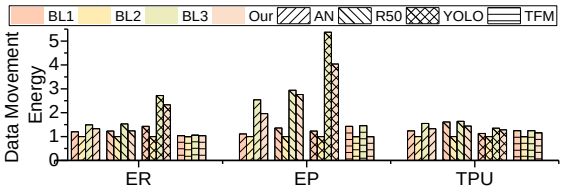

Fig. 9. Total data movement energy, normalized to BL2.

REFERENCE

[1] N. P. Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2017, pp. 1–12.

[2] J. Zhang, X. Chen, M. Song, and T. Li, "Eager Pruning: Algorithm and Architecture Support for Fast Training of Deep Neural Networks," in Proceedings of the 46th International Symposium on Computer Architecture (ISCA), 2019, pp. 292–303.

[3] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," IEEE SOLID-STATE CIRCUITS, vol. 1, 2016.

[4] J. Zhang, X. Chen, and S. Ray, "Optimizing the Whole-life Cost in End-to-end CNN Acceleration," arXiv:2104.05541, 2021.

[5] A. Parashar et al., "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019.

[6] X. Yang et al., "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators," in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020, pp. 369–383.

[7] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding Reuse, Performance, and Hardware Cost of DNN Dataflows: A Data-Centric Approach," in Proceedings of the International Symposium on Microarchitecture (MICRO), 2019, pp. 754–768.

[8] G. Reinman and N. P. Jouppi, "CACTI 2.0: An Integrated Cache Timing and Power Model."

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Commun. ACM, vol. 60, no. 6, pp. 84–90, Jun. 2017.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.

[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[12] A. Vaswani et al., "Attention is All You Need," Proceedings of the Conference on Neural Information Processing Systems (NIPS). 2017.