DNNDaSher: A Compiler Framework for Dataflow Compatible End-to-End Acceleration on IBM AIU

Sanchari Sen, Shubham Jain, Sarada Krithivasan, Swagath Venkataramani, Vijayalakshmi Srinivasan, *IBM Research, Yorktown Heights, NY*

Abstract—Artificial Intelligence Unit (AIU) is a specialized accelerator card from IBM offering state-of-the-art compute capabilities (100s of TOPS) through dataflow-driven compute arrays attached to a multi-level hierarchy of distributed memory elements. In mapping entire AI models, functional correctness hinges on maintaining dataflow compatibility between producer-consumer operations i.e., the element organization with which a tensor is produced in memory must match the organization expected by the consumer(s). This paper presents a key component in AIU's compiler stack, DNNDASHER[1]*, a systematic framework to analyze such dataflow incompatibilities and invoke an intermediate operation to shuffle tensor elements within and/or across memory elements to resolve the discrepancy. It targets opportunities to eliminate shuffles, increase granularity of memory accesses, and schedule transactions to maximize interconnect utilization. Compared to well-optimized baseline implementations of 4 CNN and Transformer benchmarks, DNNDaSher achieves of 1.27×-4.12× (average 2.3×) end-to-end latency improvement based on measured execution cycles on the AIU.*

Introduction

Recently, IBM announced its Artificial Intelligence Unit (AIU) [1], [2], a specialized PCIe attached accelerator card geared towards AI acceleration in enterprise workloads. The AIU chip with 23 billion transistors fabricated at 5nm technology offers state-of-the-art compute capabilities (100s of Tera-Operations or TOPs) supporting a spectrum of data precisions from FP16 to INT2. The AIU design targets high sustained end-to-end application throughput requiring software support to minimize data exchanges with the host CPU. This paper presents DNNDASHER, a key component in AIU's compiler stack, that enables entire AI models to be offloaded to the AIU.

The AIU is a *hybrid data-flow/control-flow* architecture consisting of 32 cores connected through a bi-directional ring [2]. Each core employs two 2D-systolic SIMDized *dataflow-driven* computation arrays [3] targeting batch Matrix-multiplications and convolutions. A 4-level memory hierarchy with distributed scratchpads and vector register files delivers data to the compute engines. Data accesses and transfers are *control-driven*, allowing complex tensor reuse patterns (*e.g.,* windowed access with strides and padding) within and across operations, reducing performance/energy penalty from data movement.

For each operation in a Deep Neural Network (DNN) model, AIU's graph compiler [3] generates spatio-temporal compute mappings which dictate the memory organization of the input/output tensor elements of the operation. Independently, the branchy topology of contemporary DNNs leads to the presence of *data operations* (*DataOp*) that intrinsically manipulate tensor element organization in memory by splitting (*e.g., Slice*), combining (*e.g., Concat*), permuting (*e.g., Transpose*) or re-interpreting (*e.g., Reshape*) elements. When offloading entire DNN models to the AIU, the aforementioned architecture-driven and workload-driven factors bring out the challenge of *dataflow compatibility*, *i.e.,* ensuring the tensor organization of producer/consumer operations match their respective expectations, to guarantee correct functionality. If ex-

XXXX-XXX © 2024 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

pectations in tensor organization differ, the tensor is deemed dataflow incompatible and an intermediate operation has to be executed to shuffle its elements and resolve the discrepancy.

In this paper, we present DNNDASHER, a compiler framework to detect and resolve dataflow incompatibilities achieving end-to-end mapping of DNNs on the AIU. DNNDASHER's optimization techniques minimize data-shuffles and mitigate performance overheads from dataflow incompatibilities. Prior efforts on architecture and compiler design for AI accelerators focus on hardware/software co-design techniques to improve compute efficiency, support new operations and dataflows, and generate optimized compute mappings [3]–[9]. To the best of our knowledge, this is the first work devoted to study dataflow incompatibility and its performance impact in modern DNNs. While we present and evaluate DNNDASHER in the context of the AIU, the key ideas can potentially benefit a broader set of AI accelerators.

DNNDASHER takes a DNN workload graph and computation mappings for its constituent operations as inputs. It identifies the tensor organization needs for the different operations to detect incompatibilities, and employs data-shuffle primitives to re-organize tensor elements ensuring correct functionality. Furthermore, DNNDASHER is equipped with a suite of performance optimization techniques geared towards: (a) eliminating the need for data-shuffle (or) minimizing the number of elements shuffled by adjusting tensor organization in memory, and (b) increasing granularity of memory accesses during data-shuffles.

In summary, we make the following contributions:

- We present DNNDASHER, a compiler framework to analyze and resolve dataflow incompatibilities in end-to-end execution of workloads on AIU. DNNDASHER addresses discrepancies stemming from both workload-driven (presence of data operations) and architecture-driven (computation mapping) source.
- We develop ɘ suite of new optimizations within DNNDASHER to reduce the performance overheads from data-shuffles.
- We evaluate DNNDASHER on a suite of 4 DNN benchmarks offloading them end-to-end on the AIU using TensorFlow deep learning framework. Relative to a well optimized baseline, DNNDASHER achieves 1.27×-4.12× (average 2.3×) improvement in inference latency measured on the AIU. The performance optimizations accelerate data-shuffles by ∼4.7× compared to baseline implementation.

Dataflow Compatibility in AI Acceleration

AIU architecture balances flexibility and efficiency by exercising both data-flow driven and control-flow driven components. As shown in Figure 1, each processing core [3] of the AIU has two 8x8 SIMD systolic arrays of Mixed-precision Processing Elements (MPEs) (supports FP16, FP8, INT8, INT4 and INT2) and two Special Function Units (SFUs). The operation of the MPE array and SFU is largely dataflow-based *i.e.,* data communication amongst the compute engines triggers execution. This leads to high sustained utilization when executing structured computations prevalent in AI models. The memory hierarchy is comprised of 4 levels: vector register file within each MPE/SFU engine, a L0 scratchpad private to each MPE array, a L1 scratchpad private to each core and shared by the 2 MPE arrays, and an external device memory accessible by all 32 cores connected through a bi-directional ring. The operation of the memory elements is control-flow based *i.e.,* data access/movement is explicitly managed. Doing so facilitates temporal reuse of tensor elements which in turn optimizes data communication costs (bandwidth) and enables end-to-end execution of DNN models within the accelerator.

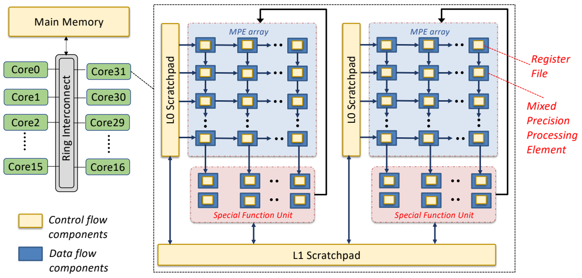
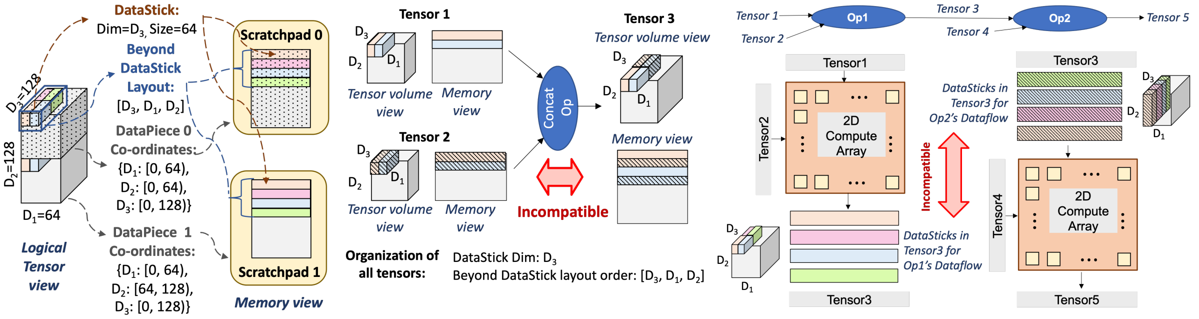


FIGURE 1: Block diagram of AIU's core [3]

The hybrid data-flow/control-flow nature of AIU poses an interesting challenge, which we refer to as *dataflow compatiblity*. In the simplest terms, dataflow compatiblity ensures that the producer operation generates the output in memory such that the consumer operation can directly use it. If incompatible, then tensor elements must be shuffled to re-arrange them in a consumable form. Needless to say, optimizing the cost of the data-shuffles—either completely eliminating through smart data allocation or realizing them by effectively utilizing the interconnect infrastructure—is crucial to the end-to-end performance of AI accelerators. More fundamentally, enforcing dataflow compatibility in a systematic manner using appropriate data-shuffle operations is a baseline requirement for correct


DataStick:
Dim=D3, Size=64
Beyond DataStick Layout: [D3, D1, D2]
DataPiece 0 Co-ordinates: {D1: [0, 64), D2: [0, 64), D3: [0, 128)}
DataPiece 1 Co-ordinates: {D1: [0, 64), D2: [64, 128), D3: [0, 128)}
D3=128
D2=128
D1=64
Logical Tensor view
Scratchpad 0
Scratchpad 1
Memory view
Tensor 1
Tensor 2
Tensor 3
D3
D2
D1
Tensor volume view
Memory view
Concat Op
Incompatible
Organization of all tensors:
DataStick Dim: D3
Beyond DataStick layout order: [D3, D1, D2]
Tensor 1
Tensor 2
Op1
Tensor 3
Tensor 4
Op2
Tensor 5
Tensor1
Tensor2
2D Compute Array
Tensor3
Tensor4
Tensor5
DataSticks in Tensor3 for Op2's Dataflow
DataSticks in Tensor3 for Op1's Dataflow
Incompatible

(a) Example tensor organization on AIU

(b) Workload-driven incompatibility due to presence of DataOps

(c) Architecture-driven incompatibility due to dataflow used in compute arrays

FIGURE 2: Tensor organization and different causes of dataflow incompatibilities.

functional execution.

Dataflow incompatibility could arise from both workload-driven and architecture-driven sources. Starting with a deep-dive into tensor organization on the AIU, the following sub-sections present sources of dataflow incompatibility, and the opportunity for performance optimization.

Tensor Organization in AIU

Figure 2(a) connects the logical view of the tensor to its memory view using 3 key components: (i) the organization of a DataStick, (ii) the organization beyond DataStick and (iii) the organization across scratchpads in different cores.

DataStick. The AIU has wide data buses to carry input/output operands between the memory and compute arrays. For example, the bidirectional ring offers a bandwidth of 128 bytes/cycle in each direction. To maximally utilize these bandwidths and sustain dataflow in the MPE arrays, the tensor elements must be packed cognizant of the dataflow to avoid scatter-gather accesses from memory in each cycle. We term this lowest-level packing of tensor elements as a *DataStick*. The DataStick shown in Figure 2(a) is comprised of 64 elements grouped along a single dimension D_3. In general, DataStick could be comprised of variable number of elements (based on their precision) and can include multiple dimensions (as dictated by the chosen dataflow). For example, an INT8 DataStick has 128 elements while an FP16 DataStick has 64 elements.

Beyond DataStick Layout Organization. We refer arranging multiple DataSticks in a memory element as "Beyond DataStick layout organization". In Figure 2(a), DataSticks along dimension D_3 are placed consecutive in memory, followed by D_1 and finally D_2. Accordingly, the Beyond DataStick layout is denoted as $[D_3, D_1, D_2]$ with dimensions ordered from innermost to outermost.

Across Memory Organization. Finally, the tensor is split across private scratchpad of each core based on the work assigned to their compute arrays. For example, in Figure 2(a), the tensor is split along dimension D_2 between the scratchpads of 2 cores. The range of coordinates in each dimension uniquely identifies the tensor elements stored in the memory.

Dataflow Incompatibility: Causes & Cost

Dataflow compatibility across operations is determined by examining the organization of tensor(s) connecting the operations. If the tensor organization required to execute the operation is different from its current form, the tensor needs to be re-organized by shuffling its data elements. Such tensor organization mismatch can arise intrinsically from the workload and/or from the way computations are mapped in the architecture. We describe each of these factors below.

Workload-Driven Incompatibility. Workload-driven incompatibility arises from the branchy-nature of modern DNN topologies. DNNs often include a sequence of data operations (DataOps) which are embedded within other deep learning (ComputeOps) operations. Such DataOps directly affect dataflow compatibility by intrinsically manipulating tensor element organization in memory by splitting (*e.g., Slice*), combining (*e.g., Concat*), permuting (*e.g., Transpose*) or re-interpreting (*e.g., Reshape*) elements.

Figure 2(b) shows a *Concat* operation on two 3-dimensional input tensors, *Tensor*1 and *Tensor*2. Concatenation is along dimension D_3, which is part of the DataStick and the innermost in the beyond DataStick layout. Hence *Tensor*3 needs to be formed by interleaving elements along D_1 and D_2 dimensions. Across 4 DNN benchmarks (details in the Results Section), we observed that the fraction of DataOps vary between 4% (InceptionV3) and 28% (YoloV3-Tiny).

The increasing complexity in DNNs topologies over the years has led to a larger fraction of *DataOps* resulting in increased cost from dataflow incompatibility. Growing compute capabilities to improve performance of ComputeOps further exacerbates the cost of dataflow incompatibility resulting from *DataOps*.

The use of different precisions by producer-consumer operations also triggers dataflow incompatibility. The number of elements packed within a DataStick varies across precisions. Hence, representing a tensor at different precisions necessitates changing its DataStick composition through a data-shuffle operation.

Architecture-Driven Incompatibility. Architecture-driven incompatibility arises from the way in which computations are mapped in the accelerator. Specifically, the dataflow used in the compute array dictates the tensor dimension(s) that are mapped spatially across its compute elements, and as a consequence determines the composition of the DataStick of the tensors used in the computation. In Figure 2(b), *Tensor*3 is *Op*1's output and *Op*2's input, and if different dimensions are mapped along the columns of the MPE array in the two operations, then DataStick produced by *Op*1 will be incompatible for consumption by *Op*2.

Execution Time for Data Shuffles. Data-shuffles across all three components of tensor organization *viz.*, the composition of DataStick, beyond DataStick layout, and tensor split across memory elements introduce performance overheads. The most significant overhead arises from DataStick incompatibility as the tensor needs to be re-organized in terms of individual elements. Beyond DataStick layout incompatibility is the least expensive, as the shuffles are limited to a given memory element and performed in the granularity of DataSticks. Across memory data-shuffles are also performed in the granularity of DataSticks but their cost depends on the interconnection fabric among the memory elements. Across 4 DNN benchmarks executing on AIU, we observe that data-shuffles account for 42.4%-80% (average of 61.8%) of the overall execution cycles. This strongly motivates compiler techniques to accelerate data-shuffles to achieve high end-to-end throughput.

DNNDaSher: Data Shuffle Primitives and Optimizations

We now present a compiler framework DNNDASHER, which uses a DNN graph and a spatio-temporal mapping of the workload onto IBM AIU to (i) analyze the dataflow/tensor organization requirements of the different operations and (ii) insert data shuffle operations to ensure functionality while maximizing performance. The following sections describe the primitives used to both resolve dataflow incompatibilities and optimize performance.

Ensuring Functionality with Data-shuffle

Incompatibility in tensor organization can stem from any of its 3 components *viz.* discrepancy in dimensions composing the DataStick, layout dimension order beyond DataStick and sub-tensor split across private scratchpads of each core. DNNDASHER employs 2 data-shuffle primitives to cure tensor organization discrepancies and render operations dataflow compatible. The primitives are:

- **Intra-DataStick shuffle.** Shuffle tensor at the granularity of individual elements for any given precision within a DataStick. This is geared towards curing discrepancies in the composition of DataStick.
- **Inter-DataStick shuffle.** Shuffle tensor at the granularity of full DataSticks within or across multiple memory elements. This helps resolve discrepancies in beyond DataStick layout and across-memory tensor split.

Tensor organization mismatch could occur simultaneously in multiple components. In the example shown in Figure 3, *TensorX* used in *Op*1 and *Op*2 has a mismatch is both the DataStick composition and beyond DataStick layout. In this case, DNNDASHER introduces a Intra-DataStick shuffle and Inter-DataStick shuffle primitive in sequence to cure each discrepancy respectively. Sometimes, performing a Intra-DataStick shuffle necessitates an Inter-DataStick shuffle first to gather the necessary DataSticks from different memory elements and then another Inter-DataStick shuffle at the end to re-distribute DataSticks.

AIU is equipped with programmable load/store units that can realize the aforementioned primitives. First, DNNDASHER steps through each tensor in the DNN graph and uses the workload mapping information to identify dataflow incompatibilities between producer-consumer operations. Next, it provides an analytical schedule of the desired shuffle, which is then translated into programs of the load/store units. We now discuss both shuffles in detail.

Intra-DataStick shuffle & Valid-Pad Tracking. One key concept in swapping dimension(s) within DataStick is that the tensor size in the newly introduced dimension

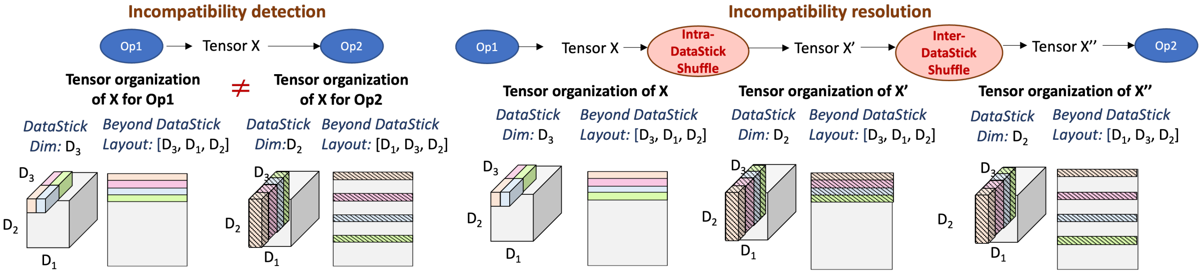

Incompatibility detection
Op1
Tensor X
Op2
Tensor organization of X for Op1
≠
Tensor organization of X for Op2
DataStick Dim: D3
Beyond DataStick Layout: [D3, D1, D2]
DataStick Dim:D2
Beyond DataStick Layout: [D1, D3, D2]
D3
D2
D1
Incompatibility resolution
Op1
Tensor X
Intra-DataStick Shuffle
Tensor X'
Inter-DataStick Shuffle
Tensor X''
Op2
Tensor organization of X
Tensor organization of X'
Tensor organization of X''
DataStick Dim: D3
Beyond DataStick Layout: [D3, D1, D2]
DataStick Dim: D2
Beyond DataStick Layout: [D3, D1, D2]
DataStick Dim: D2
Beyond DataStick Layout: [D1, D3, D2]

FIGURE 3: Detecting and resolving dataflow incompatibilities through Intra-DataStick and Inter-DataStick shuffles.

must be a multiple of desired DataStick size along that dimension. This is required because the overall tensor is organized in granularity of DataSticks and hence cannot contain partial *undefined* memory regions within a DataStick. To this end, the dimension introduced within the DataStick is *padded* at the end to make it a multiple of the DataStick size. Overall, the Intra-DataStick shuffle swaps out the elements of the original DataStick dimension and introduces elements from the new DataStick dimension alongside any required padding into the DataStick.

Another important aspect is that we track padding information through the DNN graph to ensure all operations utilizing the tensor are aware of the padding. We represent the span along each dimension using a tuple of $(Valid, Pad)$ values. Computations using a tensor and other tensor(s) interacting with it need to be potentially padded to match dimension spans. Further, the data-value padded is operation dependent. For example, accumulation could use a value of 0, whereas *Max* operation would require $-\infty$ for correctness. Therefore, in rare use-cases, if different consumers of the same tensor require different padding values, it results in tensor duplication.

Inter-DataStick shuffle & DataPiece Tracking. The Inter-DataStick shuffle primitive addresses incompatibilities arising due to changes in beyond DataStick layout and/or tensor distribution across memory elements. To ascertain the shuffle required within or across memory element(s), we introduce the notion of a DataPiece. A DataPiece is a sub-tensor comprised using a group of DataSticks spanning a *continuous range of coordinates* in each dimension. For example, Figure 2(a) shows the DataPieces across 2 scratchpads. With this abstraction, shuffles both within and across memory can be captured through affine relationships between the desired input and output DataPieces.

Similar to $(Valid, Pad)$, the DataPiece is systematically tracked through the DNN graph. DNNDASHER is versatile to allow multiple DataPieces within a memory, and DataPieces to be contained within larger memory region allocations.

Data-shuffle Performance Optimizations

By systematically tracking DataSticks and DataPieces through the DNN graph and introducing intra/inter-DataStick shuffles, DNNDASHER ensures dataflow compatibility and correct functionality. To address the performance overheads due to data shuffles, new optimizations are introduced in DNNDASHER to eliminate/minimize data-shuffle requirements and increase the granularity of memory accesses in data-shuffles, as detailed below.

Anywhere Padding. The previous subsection presented the concept of $(Valid, Pad)$ tuple to meet DataStick requirements, wherein necessary padding at the end of the valid tensor region is used. We generalize the notion to *Anywhere Padding*, wherein padding is not restricted to occur only at the end of the valid region, but instead padding could be *apportioned to any point within the valid region*. With anywhere padding, the tensor in each dimension is represented using a list of tuples—$List\{(Valid, Pad)\}$. For example, $\{(40, 20), (60, 30)\}$ represents a tensor dimension with a span of 150 elements in total and 100 valid elements. A padding of 20 occurs after the first 40 valid elements and a padding of 30 at the end.

Anywhere padding is useful to eliminate expensive intra-DataStick shuffles, albeit sometimes at the cost of increased tensor volume. For example, consider a *Concat* operation shown in Figure 4, where 2 inputs are concatenated along dimension D_3, which is also part of the DataStick. The inputs themselves have padding, $(100, 28)$ and $(50, 14)$ respectively, to meet DataStick size (=64). Conventionally, when concatenated, the output ($Tensor3$) should span 192 elements with 150 valid elements and a padding of 42 at the end—$(150, 42)$. This would necessitate a rearrangement of elements along D_3, wherein padding from the first input is moved to the end after placing the

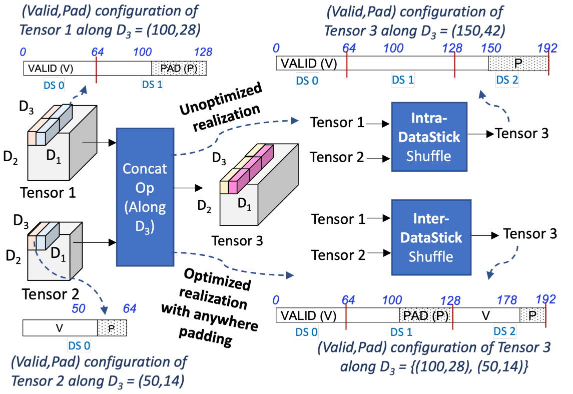


FIGURE 4: Optimizing *Concat* with anywhere padding.

valid elements from the second input, requiring an expensive intra-DataStick shuffle. Instead, with anywhere padding, we simply allow the padding to be present at the same position as the input tensors, making the output tensor to have apportioned padding—$\{(100, 28), (50, 14)\}$ and eliminate the expensive intra-DataStick shuffle.

Optimizing Intra-DataStick Re-interpretation. Tensor dimensions are often re-interpreted in DNN graphs using operations like *Reshape*. A common example is object detection models where feature pixels of tail convolutions are re-interpreted as object boxes. When the dimension constituting the DataStick is broken into multiple dimensions, it results in Intra-DataStick shuffle, as elements corresponding to the new dimension need to be moved out of the DataStick. Careful formulation of the anywhere padding avoids Intra-DataStick shuffle and introduces padding in a manner where element ranges that are part of the new dimension are separated into different DataSticks.

Figure 5 explains this concept using an example wherein dimension D_2 in the DataStick had a $(Valid, Pad)$ configuration of $(100, 28)$. Through a *Reshape*, D_2 is broken into D_{2_1} and D_{2_2} of size $50{\times}2$ respectively. The re-interpretation of D_2 triggers an intra-DataStick shuffle, to separate element ranges [0,49] and [50,99] into separate DataSticks. Instead, the *Reshape* can be turned into a *Nop*, by applying anywhere padding to D_2 and constructing its $(Valid, Pad)$ configuration as $\{(50, 14), (50, 14)\}$. Now when D_2 is split, valid element ranges [0,49] and [50,99] naturally fall in separate DataSticks as they are buffered with pad regions.

Intra-DataStick shuffle on 2D-Compute Array. A common form of intra-DataStick shuffle observed in DNN graphs is where the DataStick composition is completely changed from one dimension to another *i.e.* the original DataStick is made of a single dimension (say D_1) and the shuffle replaces it with another dimension (say D_2). Instead of using costly element-wise shuffles, the MPE arrays (which are *idle during the shuffle*) can be leveraged to effect the dimension swap. The first important observation is that if the DataStick comprises of S elements, then all element swaps are localized within an $S \times S$ slice of the tensor along D_1 and D_2. This enables the problem to be parallelized across the multiple MPE arrays in the accelerator. The next key insight is that, given the $S \times S$ slice of the tensor, changing the DataStick dimension can be iteratively achieved by transposing a 2D-plane of $S{\times}S$ elements along D_1 and D_2. This 2D-transpose operation maps well to the 2D MPE arrays.

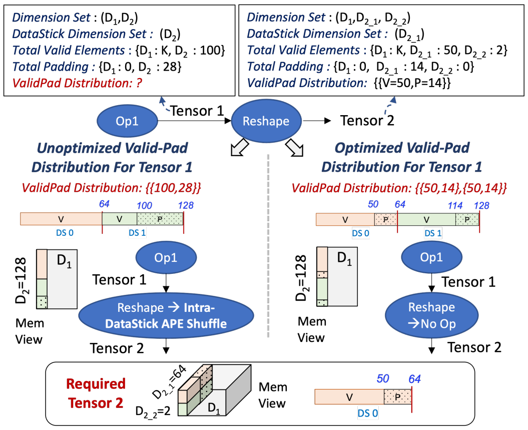

Dimension Set : (D_1,D_2)
DataStick Dimension Set : (D_2)
Total Valid Elements : {D_1 : K, D_2 : 100}
Total Padding : {D_1 : 0, D_2 : 28}
ValidPad Distribution: ?
Dimension Set : (D_1,D_{2_1}, D_{2_2})
DataStick Dimension Set : (D_{2_1})
Total Valid Elements : {D_1 : K, D_{2_1} : 50, D_{2_2} : 2}
Total Padding : {D_1 : 0, D_{2_1} : 14, D_{2_2} : 0}
ValidPad Distribution: {{V=50,P=14}}
Op1
Tensor 1
Reshape
Tensor 2
Unoptimized Valid-Pad Distribution For Tensor 1
ValidPad Distribution: {{100,28}}
64
100
128
V
V
P
DS 0
DS 1
D_2=128
D_1
Mem View
Op1
Tensor 1
Reshape → Intra-DataStick APE Shuffle
Tensor 2
Optimized Valid-Pad Distribution For Tensor 1
ValidPad Distribution: {{50,14},{50,14}}
50
64
114
128
V
P
V
P
DS 0
DS 1
D_2=128
D_1
Mem View
Op1
Tensor 1
Reshape →No Op
Tensor 2
Required Tensor 2
D_{2_1}=64
D_{2_2}=2
D_1
Mem View
50
64
V
P
DS 0

FIGURE 5: Optimizing re-interpretation of DataStick dims.

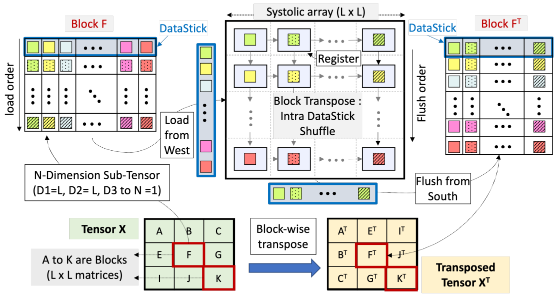


FIGURE 6: Intra-DataStick shuffles using 2D compute array

Figure 6 illustrates the overall procedure that effects transpose using an $L \times L$-sized MPE array. First, the tensor is broken to $L{\times}L$ sub-matrices. In granularity of a row, L elements of the sub-matrix are loaded from the *West* into register files in each column of the array. Thus register files along a *row* of the MPE array

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3423750

cumulatively contain elements from a single column of the sub-matrix, achieving the transpose. These elements are simply flushed *South* and stored back to memory. Thus an $L \times L$ transpose can be achieved with $2L$ transactions (load + flush) in contrast to L^2 element-wise shuffles. Finally, to achieve transpose on the full $S \times S$ matrix, the $L \times L$ transposed sub-matrices are block-wise re-arranged using the inter-DataStick shuffle. Thus element-wise shuffles are completely avoided in swapping dimensions within the DataStick.

Virtual Tensors and Smart Address Allocation. Operations such as *Slice* and *Concat* which extract/stitch sub-tensors from/into a larger tensor necessitate an inter-DataStick shuffle. To eliminate unnecessary inter-DataStick shuffles DnnDaSher exercises the concept of *Virtual Tensor* which is a sub-tensor whose elements are contiguous in memory within a larger tensor. In general, a sub-tensor formed by slicing the outermost dimension in the beyond DataStick layout, will yield a group of elements that are contiguous in memory and can form a virtual tensor. Inner dimensions can be sliced if all outer dimensions are sliced to size of 1.

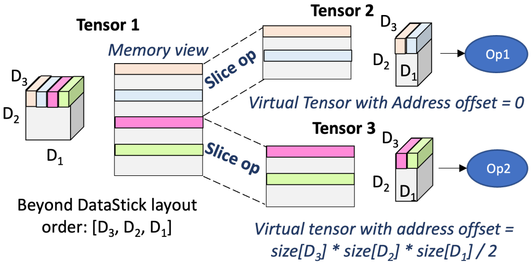

Tensor 1
Memory view
D3
D2
D1
Beyond DataStick layout order: [D3, D2, D1]
Slice op
Slice op
Tensor 2
D3
D2
D1
Op1
Virtual Tensor with Address offset = 0
Tensor 3
D3
D2
D1
Op2
Virtual tensor with address offset = size[D3] * size[D2] * size[D1] / 2

FIGURE 7: Realizing Slice operation using Virtual Tensors.

DnnDaSher extracts and stitches sub-tensors using virtual tensors by allocating the memory space for the larger tensor volume, and choosing the beyond DataStick layout to ensure that the concatenated/split dimension is outermost. Doing so enables each sub-tensor to become a virtual tensor of the larger volume and addresses can therefore be assigned corresponding to its index position in the outermost dimension. Figure 7 uses virtual tensor for a *Slice* operation wherein the big tensor (*Tensor*1) is sliced along its outer dimension D_1. *Tensor*2 and *Tensor*3 are virtual, which are used in *Op*1 and *Op*2 respectively by merely offsetting their start addresses, thereby not requiring data shuffles.

Results

Experimental Methodology

Benchmarks. Our benchmark suite consists of 4 DNN benchmarks from multiple domains of AI applications: (i) InceptionV3 network for image classification; (ii) YoloV3-Tiny network for object detection; (iii) BERT-Base and BERT-Large transformers for natural language processing with a sequence length of 384. We evaluate inference with batch sizes 1 and 4. The matrix multiply and convolution operations in the benchmarks are executed at 8-bit precision, with other auxiliary operations (activation, normalization, pooling) executing in 16-bit precision.

Software Stack. We exercise AIU's software stack [3] that facilitates end-to-end execution of DNN models from the framework onto the accelerator. DnnDaSher is a component within the AIU compiler and works in conjunction with the optimized mappings generated.

Performance Evaluation. The AIU is attached to a host CPU which runs TensorFlow deep learning framework. DNN graphs are executed end-to-end on the AIU with TensorFlow offloading all compute/data operations. We verify correct functionality across multiple inference inputs from the test dataset and obtain execution cycles across the full benchmark.

End-to-End Performance Improvement

Batch size=1 Inference Figure 8 shows the speedup achieved across the benchmark suite with batch size of 1. Overall, we observe a speedup of 1.27×-4.12× (average 2.3×). The Intra-DataStick optimizations (*e.g.,* anywhere padding) impact InceptionV3 and YoloTiny the most because of their branchy nature with several *Concat* and *Reshape* that affect DataStick. The *Reshape* and *Transpose* operations in multi-head attention of BERT-Base and BERT-Large benefit from 2D compute array based DataStick shuffle. Figure 8 also shows the theoretical ceiling on speedup by forcing no overhead cycles for data-shuffles. The achieved speedup is within 1.34× of this theoretical limit. Further, ignoring compute operations, the data shuffle operations alone accrue $\sim$4.7× runtime improvement.

Batch size=4 Inference Figure 8 also presents the speedup achieved by DnnDaSher in the case of inference at a larger batch size of 4. We observe a speedup of 1.3×-3.59× (average 2.2×) The trend is quite similar to batch size=1 case, where InceptionV3 and YoloTiny achieving the highest speedups, followed by the BERT models.The peak benefits achieved with batch size=4 is lower compared to batch size=1. Since

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3423750

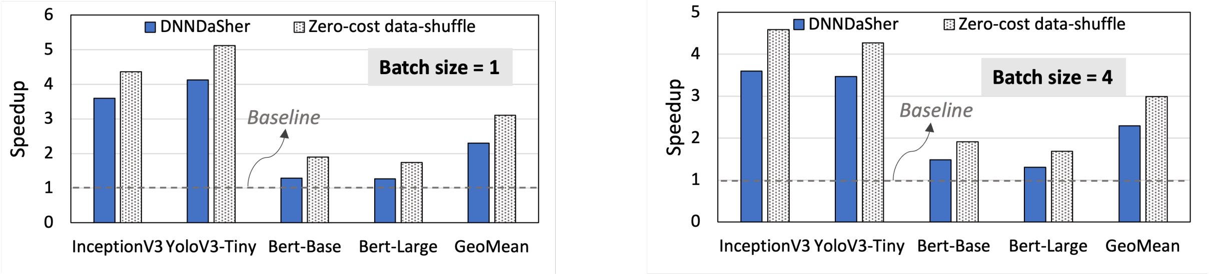


FIGURE 8: Performance benefits achieved for batch size = 1 and batch size = 4 inference

batch size dimension carries over all (compute and data-shuffle) operations in the DNN graph, it servers as common dimension to parallelize across the compute engines. In our case, a subset of cores process each input in a given batch making the baseline work mappings to be intrinsically more dataflow compatible across operations, lowering the opportunity to engage DNNDASHER's performance optimizations.

Conclusion

End-to-end execution of DNN workloads on any high TOPs AI accelerator necessitates consideration of dataflow compatibility *i.e.,* ensuring tensor organization in memory meets the expectations of both producer and consumer operations. Dataflow incompatibility arises from workload (presence of data operations) or architecture (computation mapping used in different operations). We propose DNNDASHER, a compiler framework to automatically detect and resolve dataflow incompatibilities achieving end-to-end mapping of DNNs on IBM AIU. We integrate DNNDASHER in the end-to-end software stack and evaluate across a suite of DNN benchmarks, and demonstrate 2.51×-27.3× reduction in data-shuffling time, which translates to 1.27×-4.12× improvement in overall inference latency.

REFERENCES

1. J. Burns and L. Chang, “Meet the ibm artificial intelligence unit.” *IBM Research Blog*, 2022. [Online]. Available: https://research.ibm.com/blog/ibm-artificial-intelligence-unit-aiu
2. M. Kar *et al.*, “14.1 a software-assisted peak current regulation scheme to improve power-limited inference performance in a 5nm ai soc,” in *Proc. ISSCC)*, 2024.
3. S. Venkataramani *et al.*, “Rapid: Ai accelerator for ultra-low precision training and inference,” in *Proc. ISCA*, 2021.
4. N. P. Jouppi *et al.*, “Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” 2023.
5. Z. Jia *et al.*, “Dissecting the graphcore ipu architecture via microbenchmarking,” *arXiv preprint arXiv:1912.03413*, 2019.
6. Y. S. Shao *et al.*, “Simba: Scaling deep-learning inference with multi-chip-module-based architecture,” in *Proc. MICRO*, 2019.
7. H. Kwon *et al.*, “Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects,” in *Proc. ASPLOS*, 2018.
8. A. Parashar *et al.*, “Timeloop: A systematic approach to dnn accelerator evaluation,” in *Proc. ISPASS*, 2019.
9. X. Yang *et al.*, “Interstellar: Using halide's scheduling language to analyze dnn accelerators,” in *Proc. ASPLOS*, 2020.