Tandem Processor: Grappling with Emerging Operators in Neural Networks

Soroush Ghodrati Sean Kinzer Hanyang Xu Rohan Mahapatra Yoonsung Kim[§]
Byung Hoon Ahn Dong Kai Wang[‡] Lavanya Karthikeyan Amir Yazdanbakhsh[♭]
Jongse Park[§] Nam Sung Kim[‡] Hadi Esmaeilzadeh

Alternative **C**omputing **T**echnologies (**ACT**) Lab
University of California, San Diego
[§]Korea Advanced Institute of Science and Technology [♭]Google DeepMind [‡]University of Illinois Urbana-Champaign

{soghodra, skinzer, hanyang, rohan, bhahn, lkarthik}@ucsd.edu
ayazdan@google.com {yskim, jspark}@casys.kaist.ac.kr {dwang47, nskim}@illinois.edu hadi@ucsd.edu

Abstract

With the ever increasing prevalence of neural networks and the upheaval from the language models, it is time to rethink neural acceleration. Up to this point, the broader research community, including ourselves, has disproportionately focused on GEneral Matrix Multiplication (GEMM) operations. The supporting argument was that the large majority of the neural operations are GEMM. This argument guided the research in Neural Processing Units (NPUs) for the last decade. However, scant attention was paid to non-GEMM operations and they are rather overlooked. As deep learning evolved and progressed, these operations have grown in diversity and also large variety of structural patterns have emerged that interweave them with the GEMM operations. However, conventional NPU designs have taken rather simplistic approaches by supporting these operations through either a number of dedicated blocks or fall back to general-purpose processors.

This work sets out to challenge the conventional wisdom in neural accelerator design and explore the architecture of an on-chip companion, dubbed Tandem Processor, that complements the rather optimized GEMM unit in neural accelerators. This processor needs to be specialized to keep up with the GEMM unit; and yet needs to be programmable to address the (1) structural and (2) operational variations. To strike a balance between specialization and programmability, on the one hand, we specialize its memory access logic with a novel ISA/microarchitecture that alleviates the register file and its associated load/store operations. On the other hand, the calculations of the non-GEMM layers are only supported through primitive arithmetic/logic vector operations. Therefore, programmability is offered at the mathematical level. The enhancements due to the specialization of the memory access logic in the Tandem Processor and its tight integration with the GEMM unit sustain the throughput and the utilization of the neural accelerator. Comprehensive evaluations of the proposed design based on the ***end-to-end*** execution of seven diverse DNNs including emerging language models show significant performance improvements and energy reduction enabled by leveraging the Tandem Processor. We provide the RTL code that is synthesizable both for FPGA and ASIC implementations in addition to the associated compiler as part of the open-source GeneSys project (https://actlab-genesys.github.io/). We also present the chip floorplan and post-layout analysis. This work is the result of 10 years of effort in building real NPUs that support end-to-end neural network execution.

1 Introduction

Deep Neural Networks (DNNs) have taken the IT industry and almost every computing research community by storm. Their compute intensity has heralded an era of neural accelerators or neural processing units [1–50]. These designs that include some of our own prior work [12, 24, 49, 51] have disproportionately focused on convolutions, then later on more broadly on GEneral Matrix Multiplication (GEMM) operations. The rationale was that more than 99% of operations of neural networks are of this type [12, 28, 30]. Researchers have focused on optimizing the design for these GEMM operations from various aspects including but not limited to sparsification [1–10, 51], bit-level flexibility [11–17, 52], use of resistive technologies [18–21, 53], analog computations [22–24], in/near memory computation [25–28], data flow optimizations [28–39], to name a few. These inspiring innovations have been effective in optimizing the runtime and energy efficiency of GEMM-based operations. *However, neural networks are not and were not just a series of matrix multiplications.* Yet, scant attention has been paid to the non-GEMM layers and neural networks have been treated as simply a sequence of GEMM operations even in commonly used neural accelerator simulators [12, 49, 54].

The scale has already shifted as new models and deep learning algorithms have emerged. Given the rising prevalence of neural networks and the transformative impact of language models in generative AI applications [55–61], it is timely to rethink neural accelerator design. As illustrated in Figure 1, non-GEMM operations have increased significantly in number, variety, and the structure of connectivity. For instance, VGG-16 [62], as the first generation of DNNs, includes non-GEMM operations from only three types. Whereas the types of non-GEMM operations have increased to ten for language models (e.g., BERT [63], GPT-2 [64]), as the current generation

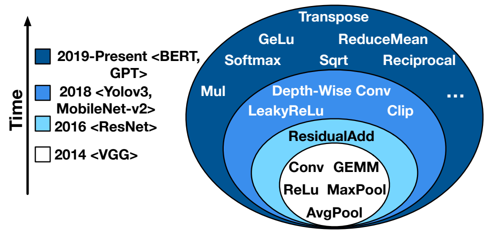


Figure 1. Neural operators in representative DNNs over the years.

of DNNs. This trend is expected to continue as DNNs enter more domains.

The non-GEMM operations are traditionally delegated to a few dedicated blocks (e.g., the ReLu/MaxPool units) [2, 4–6, 11, 12, 18–20, 22, 24, 26, 29, 31, 32, 34, 35, 38–40, 42–44, 65]. However, this approach is not sustainable as the variety of the non-GEMM operations and their structural connectivity to other layers increase. Clearly, there is a need for a rather significant degree of programmability. As such, alternative to or in addition to these blocks, an off-chip general-purpose processor [3, 7, 8, 13, 15, 16, 28, 30, 36, 37, 45–48, 66, 67] or an on-chip one [33, 68–73] is designated to handle non-GEMM operations. Through our evaluations, *we observe that runtime effects of the non-GEMM operations grow in dominance and they are not a rather small and limited minority.* Their runtime effects are amplified as the GEMM unit has been polished and optimized over the past decade. Due to these optimizations, Amdahl's bottleneck is shifting towards these non-GEMM operations. Moreover, the non-GEMM counterpart needs to keep up with this optimized GEMM unit to sustain both of their utilization levels.

To address these emerging challenges, this paper proposes a third alternative: a specialized, yet programmable processor, which acts as a companion to the GEMM unit. This specialized processor, named the Tandem Processor, not only handles the execution of the non-GEMM layers, but also orchestrates the end-to-end DNN execution and operand delivery between units. To strike a balance between specialization and programmability, on one hand, we specialize its ISA and memory semantics and alleviate the register file and its associated load/store operations. These specializations are derived from the common patterns of accesses in non-GEMM layers that rearrange and process data elements in a nested-loop fashion. On the other hand, the calculations of the non-GEMM layers are only supported through primitive arithmetic/logic vector operations. Therefore programmability is offered at the mathematical level.

The design of the Tandem Processor and the following contributions are the results of a decade-long endeavor to develop real NPUs capable of executing neural networks end-to-end.

Contributions:

(1) The paper explores the uncharted and rather ignored non-GEMM layers and their challenging structural and computational effects on the end-to-end DNN acceleration through a comprehensive analysis and characterization.

(2) We leverage the unique characteristics of non-GEMM layers and propose a new instruction execution semantic and architecture that does not adhere to the conventional Register-File-centric designs. This design enables a unique specialized data access semantic for the Tandem Processor.

(3) We exploit the common data manipulation patterns in non-GEMM DNN layers and offer a pipeline front-end that leverages microarchitectural mechanisms to keep track of strided iterators. This innovation minimizes the overhead of loop execution, address calculations, and memory accesses.

(4) We provide the RTL that is synthesizable on FPGA and ASIC implementation and the associated compiler as part of the open-source GeneSys project (https://actlab-genesys.github.io/) and present the floorplan and post-layout analysis.

We evaluate the Tandem Processor with respect to ***end-to-end*** execution of seven diverse models, ranging from rather classical DNNs to the emerging language models, when the Tandem Processor or the alternative design points augment the same GEMM unit. The results show that a balanced design offers significant advantages (2.7× speedup and 20.6× energy reduction) over the common practices of using dedicated blocks that may also require help from the off-chip host processor. In an iso-resource setting, we compare the Tandem Processor to Gemmini [72], a recent inspiring academic project [72], which uses an on-chip RISC-V processor in addition to the dedicated blocks. Utilizing the Tandem Processor outperforms the use of on-chip multi-core RISC-V processors by 5.9×. Compared to a TPU-like [33, 69] design that augments the GEMM unit with an on-chip general-purpose vector unit, leveraging the Tandem Processor offers 2.6× end-to-end speedup and 1.4× energy reduction. Comparison with NVIDIA's Jetson Xavier NX GPU that leverages NVDLA accelerator [65] shows 4.8× improvements in performance-per-Watt with ∼12× less resources. Finally, in an iso-TOPs setting, comparison to NVIDIA A100 GPU with TensorRT execution shows that the proposed design matches A100 performance, while the Tandem Processor provides 3.4× acceleration only for non-GEMM operations.

2 A Deep Dive into Non-GEMM Operations

2.1 Characteristics of Non-GEMM Operations

Non-GEMM operations are significantly diverse. Table 1 summarizes the non-GEMM operators used for inference across a set of diverse DNN models. We extract these operations from their corresponding ONNX implementations [78]. These layers can be categorized into five classes: (1) element-wise mathematical operations, (2) element-wise activation functions, (3) reduction-based operations, (4) data layout transformation operations, and (5) data type conversion operations. Non-GEMM operators fundamentally differ from

Table 1. Non-GEMM operators and their representative DNNs.

Non-GEMM Operator Classes	Operator Examples	Representative DNNs
Element-wise mathematical operators	Add, Sub, Mul, Exp, Sqrt, Floor, Ceil, Greater, Equal, Less, Pow, Reciprocal	ResNet [74], Yolov3 [75], MobileNetv2 [76], EfficientNet [77], BERT [63], GPT-2 [64]
Element-wise activation function	Relu, LeakyRelu, Clip, Tanh, Sigmoid, GeLU	VGG-16 [62], ResNet [74], Yolov3 [75], MobileNetv2 [76], EfficientNet [77], BERT [63], GPT-2 [64]
Reduction-based operators	Depth-wise Conv, MaxPool, GlobalAveragePool, ReduceMean, Softmax	VGG-16 [62], ResNet [74], MobileNetv2 [76], EfficientNet [77], BERT [63], GPT-2 [64]
Data layout transformation	Transpose, Reshape, Concat	Yolov3 [75], BERT [63], GPT-2 [64]
Type conversion	Cast, BitShift	Any Inference

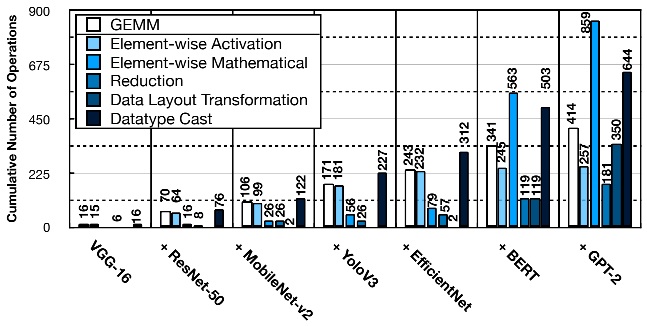


Figure 2. Cumulative number of GEMM and non-GEMM operations across benchmarks. Last bar covers the frequency of usage across all the models.

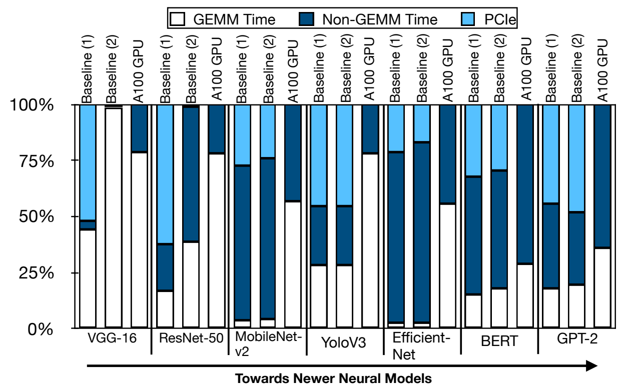


Figure 3. Runtime breakdown of benchmark DNNs across various platforms.

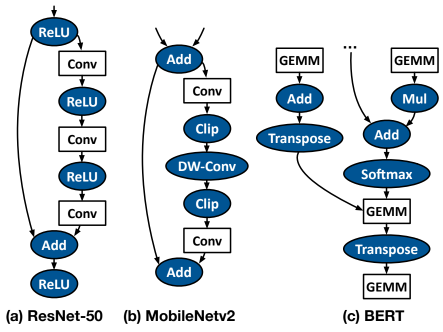


Figure 4. Repeated subgraphs of (a) ResNet-50 [74], (b) MobileNetv2 [76], and BERT [63]. The gray ovels illustrate the non-GEMM operations and white rectangles show the GEMM-based operations.

GEMM ones. They exhibit a wide diversity in terms of compute operations ranging from simple mathematical operations (e.g. Add, Mul, etc.) to complex ones (e.g. GeLU, Exp, etc.) as opposed to the commonly used multiply-accumulate in GEMM layers. Moreover, they require various patterns of mapping between input and output tensors, from one-to-one in element-wise operations to many-to-one in reduction-based ones.

Usage frequency of non-GEMM operations is continuously growing. Figure 2 shows the usage frequency of the GEMM and non-GEMM operators across the studied benchmarks. We extract this data from the ONNX graph representation of each model and categorize them with respect to the classification in Table 1. The y-axis shows the cumulative usage of these operators as additional models are taken into account[1]. The last group of bars show the total cumulative usage of operators across all benchmarks. As shown in Figure 2, as additional models are covered, the cumulative number of non-GEMM operations noticeably surges. Additionally, taking the entire benchmarks into account (last bar), merely 15% of total DNN operator nodes are GEMMs.

Non-GEMM operations impose non-trivial runtime overheads in newer DNNs. Figure 3 shows the runtime breakdown of benchmark DNNs for three design choices: (1) a GEMM unit with an off-chip CPU (Baseline (1) in Figure 3), (2) a GEMM unit coupled with a set of dedicated units and the same off-chip CPU (Baseline (2) in Figure 3), and NVIDIA A100 GPU that leverages tensor cores and INT8 execution mode. Section 7 describes the experimental methodology to obtain these results. Figure 3 reports the runtime breakdown across the time spent on GEMM layers, non-GEMM layers, and PCIe communications (for the case of Baselines (1) and (2)). As the non-GEMM layers become more diverse and complex in newer models such as EfficientNet, BERT, and GPT-2 they also become the main source of the execution bottleneck. For instance, the execution of non-GEMM layers take up 81% and 73% of the runtime for EfficientNet in Baseline (2) and GPU, respectively.

Non-GEMM operations are interspersed amongst GEMM operations. Figure 4 depicts the core and frequently used subgraphs of three representative DNNs. As shown, the non-GEMM operators are interspersed amongst the GEMM ones (e.g. Conv) with various forms of connectivity. This structure demands back-and-forth data exchange between GEMM and non-GEMM units through off-chip or on-chip memory. On top of this data exchange, tensor reformatting such as datatype

[1]The models are listed in chronological order.

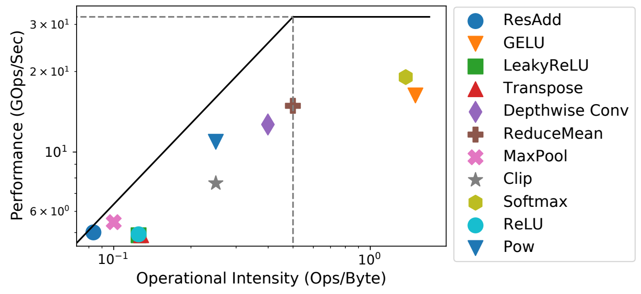


Figure 5. Roofline model for a number of prevalent non-GEMM operators.

casting and tensor layout transformations may be required. For instance the GEMM unit may operate with INT-8/16 mode, while the non-GEMM unit operates in FP32 mode.

The majority of non-GEMM operations are memory-bound. The majority of non-GEMM layers are element-wise operations (>80%). Moreover, the ones that are not element-wise exhibit low computational intensity and data reuse. Figure 5 shows a roofline [79][2] analysis for a set of prevalent non-GEMM operators. As shown, most of the analyzed operators (other than Softmax and GeLU) fall within the memory-bound region of the roofline. This is in contrast to Conv/GEMM operations that are generally compute-bound [80]. This distinction necessitates architecture design considerations.

2.2 Requirements for Executing Non-GEMM Operations

Inspired by the above characteristics, below we list three key requirements to efficiently execute non-GEMM operations.

R1: In-tandem execution of GEMM and non-GEMM operations. To reduce the data exchange among consecutive layers (GEMM and/or non-GEMMs) through off-chip memory, prior work suggests layer fusion [81–84]. Layer fusion preserves the intermediate activation values stationary on the chip for subsequent DNN operations. To leverage this technique, the intermediate activations ought to be communicated between GEMM and non-GEMM units via on-chip memory subsystem for a sequence of fused layers. However, this data communication at the granularity of entire layer outputs is neither trivial nor efficient, due to the limited on-chip memory of the accelerators and reduced utilization of GEMM and non-GEMM units (Figure 8 shows the impact on utilization). In essence, the data transfer ought to be performed at a finer granularity of a chunk of output tensor, a.k.a *tile. This fine granularity of coordination requires the non-GEMM unit to seamlessly work in tandem with the GEMM unit, while retaining minimal data transfer and reformatting overhead.*

R2: Balanced efficiency and programmability for the non-GEMM unit. The diversity of the non-GEMM operators calls for a degree of programmability in the hardware.

[2]We performed the experiments on the Tandem Processor with the configuration shown in Table 3 of Section 7.

Table 2. Comparison of prior approaches for supporting non-GEMM operators with this work. † indicates that these aspects are supported partially.

Design classes	Working in tandem with GEMM Unit	Specialization	Programmability	Execution Control
Offchip CPU fallback	✗	✗	✓	✓
Dedicated on-chip hardware units	✓	✓	✗	✗
Onchip RISC-V core (+ dedicated units)	✗†	✗†	✓	✓
General purpose vector unit	✓	✗†	✓	✗
This work (Tandem Processor)	✓	✓	✓	✓

Nonetheless, this should not emerge at the cost of noticeable efficiency reductions. This is important because the inefficiency of the non-GEMM unit can potentially make it the performance bottleneck and result in stalling the GEMM unit. Therefore, striking a balance between programmability and specialization is crucial.

R3: Orchestrating the execution across non-GEMM and GEMM units. Having both GEMM and non-GEMM acceleration units in one coherent system requires adequate support for execution orchestration. In particular, (1) DNN nodes need to be effectively dispatched to their pertinent processing units, (2) GEMM and non-GEMM units need to diligently synchronize and handshake together at the right time to realize in tandem execution and back-and-forth interactions.

2.3 Existing Approaches for Executing Non-GEMM Layers

Table 2 compares prior methods with respect to the aforementioned requirements. Below, we discuss them in details.

Class (1): Off-chip CPU fallback. This approach presumed by a large number of prior work [3, 7, 8, 13, 15, 16, 28, 30, 36, 37, 45–48, 66, 67] provides ultimate programmability and handles the end-to-end execution orchestration. However, it impedes the performance due to the lack of specialized execution and in tandem execution with the GEMM unit, which the latter is caused by the nontrivial back-and-forth data transfer between the GEMM unit and CPU over PCIe and required data conversions (e.g. integer to float and vice versa).

Class (2): Dedicated on-chip hardware units. An alternative strategy [2, 4–6, 11, 12, 18–20, 22, 24, 26, 29, 31, 32, 34, 35, 38–40, 42–44, 65] is to equip the GEMM unit with a set of dedicated units customized for specific non-GEMM operations. These dedicated units can often be tightly integrated with the GEMM unit (work in tandem), but do not offer execution orchestration. Another drawback is, it is not scalable to augment neural accelerators with dedicated units for each single type of non-GEMM operation. This also prohibits the accelerator to support emerging non-GEMM operations as a result of evolving DNNs. In the case of unsupported operations these accelerators *must* still fall back to an off-chip CPU.

Class (3): On-chip RISC-V core. The on-chip core in these designs [72, 73] executes the non-GEMM operators and controls on-chip resources. Gemmini [72] extends the RISC-V ISA with a set of dedicated units/instructions for a limited set of non-GEMM layers. Although this approach obviates off-chip CPU communication, but still the overheads of datatype casting and layout conversion remain, blocking in tandem execution. More importantly, the on-chip core that has a single ALU lacks in terms of compute power and efficiency to process the non-GEMM layers and can become the execution bottleneck.

Class (4): On-chip general-purpose vector unit. Nvidia Streaming Multiprocessor (SM) units [68] that consist of tensor cores (GEMM units) and CUDA cores (general-purpose vector units) belong to this design class. Another notable example is the Vector Processing Unit (VPU) in Google's TPU [33, 69, 85] and other industrial designs [35, 70, 71]. Vectorized execution leverages the inherent parallelism in non-GEMM layers for increased performance improvement. Additionally, these vector units often work in tandem with the GEMM units. However, these units do not handle the execution control [33, 68] and fall short in terms of specialization. Other related industry designs include SiFive x280 [86] and Meta MTIA v1 [87]. SiFive x260 is a multi-core vector processor with RISC-V vector extensions for deep learning workloads. The design does not include a GEMM unit but provides a set of communication protocols that can be leveraged to integrate this multi-core vector processor with a GEMM unit. Another design point is Meta's MTIA v1. This design comprises a grid of Processing Elements (PEs). Each PE comprises a GEMM unit and three other units to support non-GMEM operations: (1) a SIMD array of dedicated units to support activation functions and typecast operations, (2) a general-purpose core with RISC-V vector extensions to provide further programmability for more complex non-GEMM operations, and (3) a memory layout unit that support transpose/reshape types of operations. In a sense, this design follows both Class (2) and Class (4) of accelerators and includes both dedicated units and general-purpose vector cores.

2.4 Our Approach

In this paper we offer the Tandem Processor as a specialized companion SIMD processor that operates in tandem with the GEMM unit, while striking a balance between customization and programmability. The proposed processor orchestrates the end-to-end execution, eliminating the need for an additional companion CPU.

3 Design Considerations for the Tandem Processor

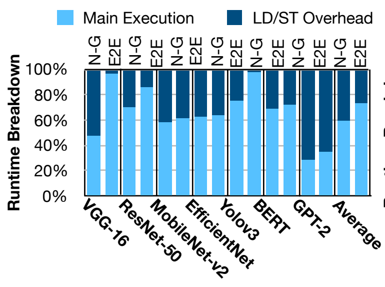
3.1 Memory Subsystem Design

The low computational intensity and the sizable tensor operands for non-GEMM operators prompt the memory subsystem to repeatedly *stream* data from off-chip memory. Thus, a locality-oriented hierarchical memory sub-system (i.e., vector register file and cache(s)) and conventional load/store data communication, necessitate an excessive number of memory instructions to deliver off-chip data to/from vector register files, funneling through the memory hierarchy. To address this, we use the following insight: Non-GEMM layers most often operate on statically-structured tensor operands with a-priori known dimensions in a streaming fashion. *The Tandem Processor replaces the entire vector register file and cache hierarchy with a collection of single-level software-managed on-chip scratchpads.* This design innovation is in contrast to all prior SIMD designs that rely on register file execution and memory semantics (e.g. Google's VPU [85]). As shown in Figure 6a, these load/store operations to vector register files on average impose 41% and 27% runtime overhead for non-GEMM operations and end-to-end execution, respectively. To manage data movements between off-chip/on-chip memories, we design a Data Access Engine. This unit can be configured and invoked by few explicit load/store instructions per tile to fetch entire tensors. Such data movement merely appears at the boundary of a tile, blocking any further intervention from the off-chip memory.

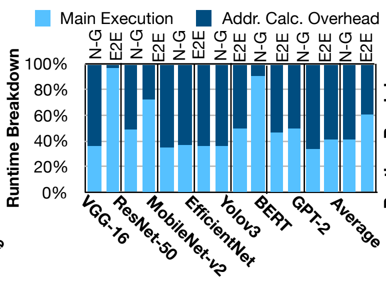
3.2 Specialized On-Chip Data Access Mechanism

Using large on-chip scratchpads submits a new challenge as fitting the scratchpad addresses in an Instruction Word as opposed to IDs of registers would require significant increase in instruction length. In addition, on-chip address calculations require excessive number of arithmetic instructions. For instance, per two-operand arithmetic/logic instruction, three extra instructions would be required solely for address calculation. As Figure 6b shows, this address calculation would impose runtime overheads: On average, 59% of the runtime for non-GEMM layers and 40% of end-to-end DNN runtime. To tackle this challenge, *we devise a dedicated pipeline stage for address calculation at the front-end*, relieving the burden of address calculation from compute units.

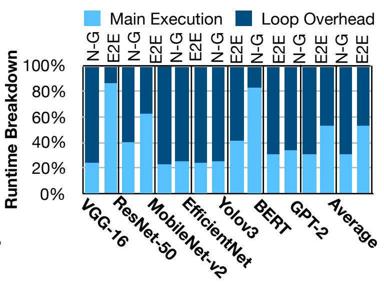
We regulate walking over each dimension of tensor operands by a tuple of ⟨Offset, Stride⟩. Hence, if these tuples can be embedded in a single instruction along with compute operations, upon being inferred at the decode stage, the scratchpad addresses can be calculated in parallel with compute operations. Yet, providing three such tuples for a non-GEMM layer would still require significant increase in instruction length. Instead, we forge scratchpad accesses through indirect strided address calculations. Figure 7 illustrates this feature. We formulate these strided accesses using ⟨Scratchpad ID, Iterator Index⟩ format. The Scratchpad ID is used to select the corresponding scratchpad iterator table and the Iterator Index points to an entry in the Iterator Table. Each entry in the Iterator Table stores a tuple of ⟨Offset, Stride⟩ for each operand. This design optimization realizes the embedding of strided addresses and compute operations into a single 32-bit instruction word (See Section 5). With this mechanism the Tandem Processor supports address calculation as well as compute operation *on the same pipeline path with shared control* and no extra



(a) LD/ST to/from register files.



(b) Address calculation.



(c) Loop execution.

Figure 6. Analyzing the overheads of non-GEMM execution eliminated by design considerations in the Tandem Processor, individually. "N-G" and "E2E" denote the runtime for Non-GEMM and End-to-End execution. These experiments are performed on the Tandem Processor + GEMM unit with Table 3 configurations with all hardware specializations and compiler optimizations, except the ones under evaluations.

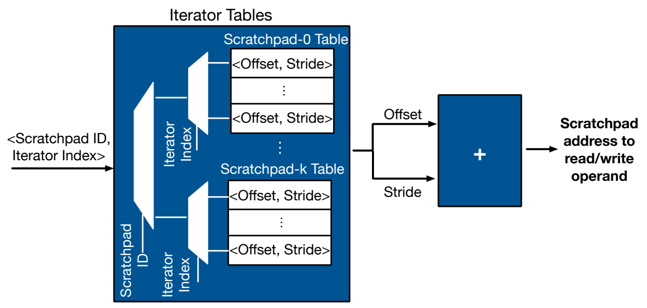


Figure 7. Indirect strided address calculation for scratchpad accesses.

runtime overhead. This is in contrast to prior work [88–90] which leverage decoupled access/execute engines with register files/FIFOs for data access and address generation.

3.3 Specialized Loop Execution

Non-GEMM layers are formed of nested loops of primitive operations with pre-determined iteration counts. As Figure 6c shows, using conventional loop logic (i.e. conditional branch) incurs on average 70% and 47% runtime overhead for non-GEMM layers and end-to-end DNN execution, respectively. To alleviate this, we devise specialized loop execution semantics, while removing the branch prediction logic.

To that end, the Tandem Processor uses software-managed tables in the fetch pipeline stage to orchestrate the execution of nested loop constructs in hardware. Prior to execution, these tables are configured *once* with the iteration counts and corresponding number of nested loop levels. Once configured, these specialized tables are used repeatedly in conjunction with the iterator tables to execute the loop body. This is crucial, since appropriate ⟨Offset, Stride⟩ tuples need to be employed at each level of loop nest to correctly calculate the scratchpad addresses. This specialized loop execution is unique to the Tandem Processor, as prior work [90, 91] leveraged hardware-managed loop logic with register-file based designs and did not offer mechanisms to combine it with address calculation.

3.4 Arithmetic Logic Units Design

ALU operations. To support a diverse set of non-GEMM layers, one approach would be to use dedicated specialized instruction for each layer. However, this would lead to a design similar to the second class in Section 2.3. We instead leverage the feasibility of implementing complex non-GEMM layers with a set of simple primitive operations [92, 93]. For instance, GeLU operator can be implemented using five multiplications, three additions, a sign, an absolute, and a minimum operations. We consider a union set of these primitives that is comprehensive enough to support non-GEMM layers shown in Table 1. Hence, the Tandem Processor offers better hardware resource utilization and reuse across a larger set of operations.

ALU precision and datatype. Prior works show that integer-only arithmetic can be used for inference execution of CNNs [94, 95] and transformers [92] with virtually no repercussions on accuracy. In addition, while GEMM layers and few non-GEMM layers such as Relu are amenable for low-precision INT8 implementation [94], some non-GEMM layers such as ResAdd and Softmax require INT32 precision [92, 95]. To provide sufficient precision for all non-GEMM operators, we use INT32 for the Tandem Processor ALUs. As a complementary benefit, additional data casting from GEMM to non-GEMM unit is not needed, since GEMM units typically accumulate the partial results in INT32 precision [12, 28, 30, 33, 69, 92]. However, since GEMM layers may use lower precision, a datatype casting instruction is required when activations move from non-GEMM to GEMM unit.

3.5 Integration with the GEMM Unit

Coordination granularity. We use tile (sub-tensor) granularity for software pipelining to facilitate execution overlap between GEMM and non-GEMM units, improve resource utilization, and better conform with limited on-chip memory capacity. As Figure 8 shows, the in tandem coordination of the GEMM unit and the Tandem Processor at tile granularity increases the compute resource utilization by 20% and 13% for the GEMM unit and the Tandem Processor, respectively.

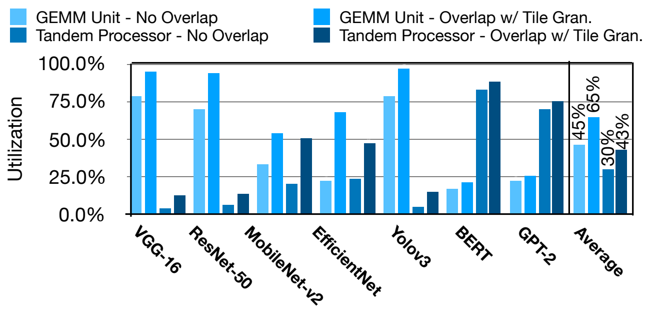


Figure 8. Resource utilization analysis.

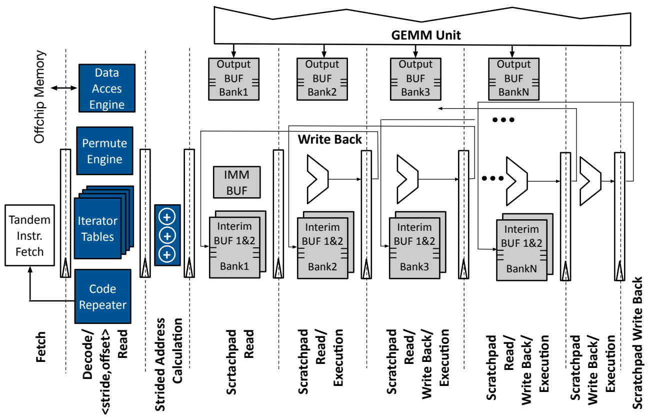


Figure 9. The Tandem Processor pipeline microarchitecture. The ALU and scratchpad reads/write stages are interleaved to improve frequency.

Note that an operand-level granularity is less efficient. This is because some non-GEMM operators, such as depthwise convolution, require arbitrary accesses to GEMM outputs for consecutive operations. This access pattern results in frequent stalls, curtailing the overall performance.

Communication mechanism. To enable tile-based coordination, one probable approach is to directly move/copy tiled data from the GEMM unit's Output BUF to the Tandem Processor's private scratchpads. However, this design decision incurs communication overhead at the boundary of each accelerator units, requiring complex coordination mechanism. Alternatively, we enable a fluid ownership of the GEMM unit's Output BUF for the Tandem Processor, obviating redundant data communications. After the GEMM unit completes storing the intermediate data in the Output BUF, the Tandem Processor takes the ownership of the buffer and directly executes its computations on the stored data.

Synchronization mechanism. To enable this fluid ownership while simplifying hardware, we leverage the compiler to weave a set of synchronization instructions (See Section 5) between GEMM and non-GEMM instructions. These synchronization instructions realize the following: (1) They identify the code regions for GEMM unit and the Tandem Processor, facilitating the instruction dispatch. (2) They define the flow of execution between GEMM and non-GEMM units. (3) They govern the handshaking mechanism between the acceleration units. For instance, enforcing the release of ownership of the Output BUF after the Tandem Processor completes the execution.

4 Microarchitecture Design for the Tandem Processor

4.1 Pipeline Design

In this section, we discuss the major aspects of the Tandem Processor's pipeline microarchitecture, illustrated in Figure 9.

On-chip memory organization. We refer to the Tandem Processor scratchpads as Namespaces, which are shown with gray colour in Figure 9. Interim BUF 1&2 namespaces represent the central Tandem Processor's on-chip scratchpads that operate as a storage medium for tensor operands as well as their intermediate results. These scratchpads, which bridge the off-chip memory and the Tandem Processor, are populated/drained by a Data Access Engine at a tile granularity. The Tandem Processor compiler configures the Data Access Engine by setting the base address of the off-chip source along with a series of stride values. Note that, the tiled data may be even dispersed across non-contiguous regions of memory lines, yet statically arranged in strided patterns. IMM BUF namespace serves as a small 32-slot scratchpad for immediate values in non-GEMM operations. This buffer is programmed with a series of customized instructions at the onset of non-GEMM layer execution. The last namespace is Output BUF, which serves as the GEMM Unit's buffer for output values.

Specialized on-chip data access. We place the Iterator Tables that are used to store the offset and stride information for scratchpad accesses at the decode stage of the Tandem Processor pipeline (see Figure 9). There is a dedicated Iterator Table for each namespaces of the Tandem Processor. Upon decoding one arithmetic/logic instruction, the ⟨Namespace ID, Iterator Index⟩ retrieves the address calculation information from the corresponding Iterator Table. The resulting outputs of accessing the Iterator Tables is a triplet address, two for source operands and one for destination operand. Each element of the triplet is a tuple of ⟨offset, stride⟩, indicating that target data resides in Scratchpad[offset + stride]. The triplet address is passed down to the subsequent pipeline stage (Strided Address Calculation) that repetitively assembles a series of scratchpad addresses, each as the result of offset + stride computation. The scratchpad indices propagate down the multi-staged execution pipeline to fetch the tiled operands, perform the non-GEMM operations, and write back the resulting data to the pipeline back-end.

Nested loop support. The Code Repeater module (see Figure 9) uses three tables: A table stores the compiler-defined iteration counts. Each entry of this table maintains the configuration of one of the loop nesting levels. The compiler organizes the loop configuration instructions from the outermost loop to the innermost one. At the Decode/<Stride,Offset>/Read pipeline stage, Code Repeater stores the number of iterations in each table entry, which is indexed using a pointer that keeps track of the number of nested loops. Once the Code Repeater is configured, it uses the second table with similar structure of entries to keep track of the current iteration of the loops. Whenever, the Code Repeater exhausts the iterations of a loop level, it decrements the pointer to update the iterations of the ensuing outer loop. Finally, the Code Repeater uses a collection of identical tables that store the information about what Iterator IDs need to be exercised for each operand at a certain loop level.

4.2 Overall Execution Flow and the GEMM-Unit-Tandem-Processor Synchronization Logic

Figure 10 illustrates the overall execution for a DNN subgraph on the NPU-Tandem. As shown, at a high level, the NPU-Tandem encompasses (1) a GEMM unit (including weight/input buffers), (2) Output Buf, which serves as a medium for communicating data from GEMM unit to the Tandem Processor, (3) an execution controller that orchestrates the overall execution and faciliates the synchronization between units, and (4) an instruction buffer that holds the instructions of the block. To execute DNNs, the compiler breaks the DNN graph into a set of execution blocks or subgraphs (step 0 in Figure 10). A block can be one of the followings: (1) a single GEMM layer, (2) a group of bundled non-GEMM layers, (3) a GEMM layer followed by a group of bundled non-GEMM layers (shown in this example). To realize the in tandem execution, a uniform tiling scheme is required across the inputs/outputs of fused layers in one block. Figure 10 shows four tiles of execution for fused GEMM (shown with square) and non-GEMM layers (shown with circles). As Section 5 discusses, the synchronization instructions mark the boundaries of GEMM and non-GEMM instructions (see the instruction block format in Step 0 of Figure 10). Figure 11 illustrates the high level view of the execution controller logic for the Tandem Processor. Below, we discuss the overall execution on the NPU-Tandem.

Instruction load and dispatch (Step 1 in Figure 10). First, the Tandem Processor loads the instructions of a block from off-chip memory into its Inst. BUF. Then, the FSM of the execution controller switches from Block Start to Inst. Dispatch state (see Figure 11). At this state, the Tandem Processor's Inst. Dispatch unit drives the Program Counter to walk over all the instructions of a block. Note that this is a lightweight decode state and does not invoke any execution on the GEMM unit or Tandem Processor. The Inst. Dispatch unit decodes the synchronization instructions to identify the topology of the block (GEMM only, non-GEMM only, or GEMM followed by non-GEMM). It then decodes the GEMM instructions and configures the GEMM unit, while writes back the non-GEMM instructions to the Inst. BUF. Note that GEMM units typically operate at macro operations level (e.g. Conv/Matmul instructions), when first a set of instructions are decoded to configure the GEMM unit. Based on the configuration, this unit then operates in a repetitive mode to fully execute the GEMM layers [12, 33, 72]. In contrast, the Tandem Processor is similar to von Neumann machines, where each instruction is decoded and executed through the processor pipeline. As such, at the end of this state, only non-GEMM instructions exist on Inst. Buf to be decoded and executed by the Tandem Processor. After the dispatch is done, based on the structure of the program block, the execution FSM switches to either of these three states: the GEMM state, the Tandem Processor state, and the GEMM-Tandem Processor state (see Figure 11). Below, we first discuss GEMM-Tandem Processor case as shown in the example of Figure 10 and then discuss the Tandem Processor only case.

GEMM-non-GEMM execution (Step 2 to Step 6 in Figure 10). If a GEMM layer is followed by a series of non-GEMM layers, the FSM transitions to the GEMM-Tandem Processor state after the instruction dispatch. GEMM unit first starts with executing the first tile (Step 2 in Figure 10). Whenever the GEMM unit finishes the tile, it sends a handshaking signal to the execution controller. If the Tandem Processor is idle, the execution controller invokes Tandem Inst. Fetch unit to start the execution of non-GEMM tile (see (FSM = Tandem | FSM = GEMM-Tandem) & GEMM_tile_done signal in Figure 11.). Utilizing a double-buffering scheme, the GEMM unit proceeds to the next tile, while the Tandem Processor takes the outputs of the GEMM-completed tile and performs the non-GEMM operations (Step 3 in Figure 10). To avoid stalls in the GEMM unit caused by the Output BUF being occupied by the Tandem Processor, the compiler inserts a synchronization instruction (see Section 5) right after the instructions consuming the data on the Output BUF. At this time, the Tandem Inst. Fetch sends a handshaking signal to the GEMM unit and Tandem Processor releases the Output BUF (OBUF_done -> GEMM Unit in Figure 11). the Tandem Processor may continue the computation using its private Interim BUFs. Once the Tandem Processor finishes a tile, it uses the synchronization instruction that marks the end of the non-GEMM program to alert the execution FSM (Tandem_done -> Exec. FSM in Figure 11). The execution FSM puts the Tandem Processor in the idle state until it receives the next tile from GEMM unit. After finishing the all tiles (Step 2 to 6 in Figure 10), the execution FSM transitions to the Block Done state (see Figure 11).

Non-GEMM only execution. The execution FSM transitions from Inst. Dispatch to Tandem state and triggers the Tandem Inst. Fetch to fetch the non-GEMM instructions and forward them to Tandem Processor pipeline. Once Tandem Processor completes executing all the instructions, the Tandem Inst. Fetch unit sends a handshaking signal to the execution FSM logic. The execution FSM loops back to this state if there are remaining tiles. To ensure the off-chip memory access instructions are updated for different tiles, the first tile is used to initialize configurations for the Data Access

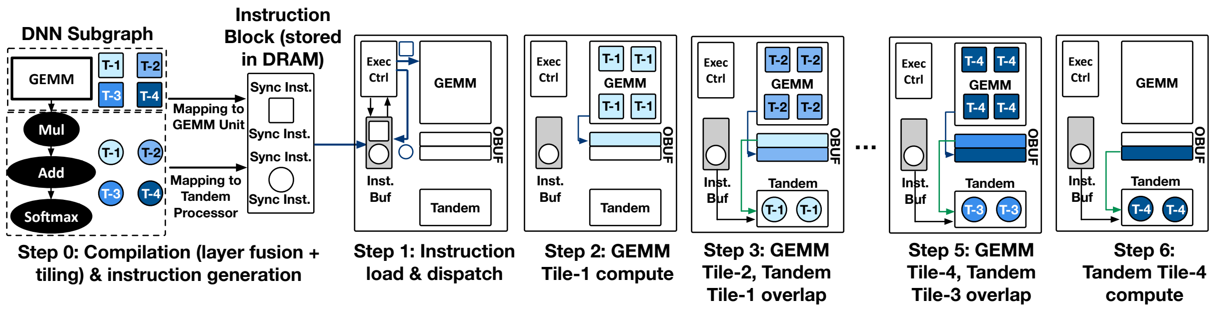

DNN Subgraph
GEMM
T-1
T-2
T-3
T-4
Mul
Add
Softmax
Instruction Block (stored in DRAM)
Mapping to GEMM Unit
Mapping to Tandem Processor
Sync Inst.
Exec Ctrl
GEMM
Inst. Buf
Tandem
OBUF
Step 0: Compilation (layer fusion + tiling) & instruction generation
Step 1: Instruction load & dispatch
Step 2: GEMM Tile-1 compute
Step 3: GEMM Tile-2, Tandem Tile-1 overlap
Step 5: GEMM Tile-4, Tandem Tile-3 overlap
Step 6: Tandem Tile-4 compute

Figure 10. An example of end-to-end execution on NPU-Tandem.

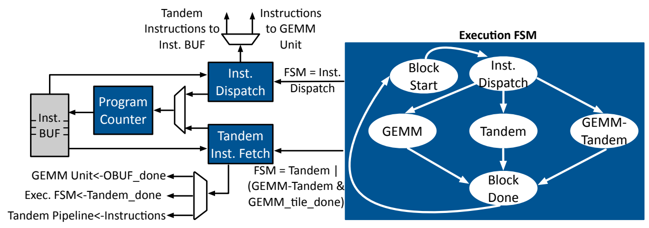


Figure 11. The execution controller.

	4 bits	4 bits	3 bits	5 bits	16 bits			
Synchronization	opcode	func	X	group ID	X			
	4 bits	4 bits	3 bits	5 bits	16 bits			
Configuration	opcode	func	ns id	iter idx	Immediate			
	4 bits	4 bits	3 bits	5 bits	3 bits	5 bits	3 bits	5 bits
Compute	opcode	func	dst ns	dst iter idx	src1 ns	src1 iter idx	src2 ns	src2 iter idx
	4 bits	4 bits	3 bits	5 bits	16 bits			
Loop	opcode	func	loop id	X	Immediate			
	4 bits	4 bits	3 bits	5 bits	16 bits			
Data Transformation	opcode	func	src/dst	dim idx	Immediate			
	4 bits	4 bits	3 bits	5 bits	16 bits			
Off-chip Data Movement	opcode	func1	func2	loop idx	Immediate			

Figure 12. The Tandem Processor instruction formats.

Engine. For rest of the tiles, the Data Access Engine reuses the initialized configurations and incrementally updates them.

5 ISA Design for the Tandem Processor

Figure 12 summarizes the instruction formats for the Tandem Processor. Below we discuss its instruction classes.

Synchronization instructions. The func bits are defined as ⟨GEMM/SIMD, START/END, EXEC/BUF, X⟩. The START/END along with EXEC bit identifies the regions of instructions that belong to the Tandem Processor and GEMM Unit (identified with GEMM/SIMD bit accordingly), which helps dispatch instructions to the appropriate unit. Also, this instruction can be used with EXEC bit to notify the GEMM Unit that the execution of non-GEMM operations of the running tile is completed, or with BUF bit to notify the GEMM Unit that the OUTPUT BUF is released and ready for the subsequent tile.

Configuration instructions. This class includes two opcodes. The ITERATOR_CONFIG opcode is used with three functions (func bits): (1) BASE_ADDR to fill the Iterator Tables with the base addresses for the scratchpads and (2) STRIDE to fill the Iterator Tables with strides for the scratchpad address calculation, and (3) IMM BUF to fill the immediate buffer with the immediate values needed for non-GEMM operations. The ns id and iter idx fields identify the target namespace and the index to its corresponding Iterator Table. Also, this instruction is used to set the immediate values in IMM BUF. Another opcode is DATATYPE_CONFIG, which is used for datatype casting.

Compute instructions. Opcode ALU is defined with various func bits to support Add, Sub, Mul, MACC, Div, Max, Min, Shift, Not, AND, OR operations on src1 and/or src2 operands. Additionally, this opcode supports MOVE/COND_MOVE instructions for scatter/gather operations. In case of COND_MOVE , the first source operand (src1) is moved predicated upon true/-false flags identified by the second operand (src2). Opcode CALCULUS consists of mathematical operations such as absolute value and sign. Opcode COMPARISON supports logical comparisons. The operands (src1/src2/dst) for each instruction are specified by using a 3-bit ns id to locate the buffer, and a 5-bit iter idx corresponding to the stride and offset.

Loop instructions. This class is used with the LOOP opcode to configure the Code Repeater. This opcode is used with SET_ITER function bits to specify the iterations for each loop identified by loop id. The SET_NUM_INST function is used to identify the number of instructions in the loop body. To cope with the customized on-chip memory accesses for each loop dimension, the SET_INDEX function is used, while the rest of the instruction bits are used to set the associated ⟨ns ID, iter idx⟩ for the three operands (similar to compute instructions). The loop instructions are designed to support arbitrary levels of nesting (up to eight, each of which is identified by loop id field) needed by non-GEMM operators.

Data transformation instructions. This class is used with two opcodes: (1) PERMUTE for permuting multi-dimensional tensors using the Permute Engine shown in Figure 9 and (2) DATATYPE_CAST for datatype casting. For PERMUTE opcode, SET_BASE_ADDR, SET_LOOP_ITER, and SET_LOOP_STRIDE functions configure the base addresses, shapes, and strides, respectively, for both the source and destination's tensor dimensions (identified by dim idx). Then, with the START function, the iterators start generating the address for the source and destination according to the desired permutation. Additionally, the LSB bit of the Immediate field while using the START function identifies if this permutation operation requires shuffling the data across the SIMD lanes/scratchpad banks or not. DATATYPE_CAST opcode is used to cast tensor elements to various fixed-point representations such as FXP32, FXP16, FXP8, and FXP4 needed by the GEMM unit.

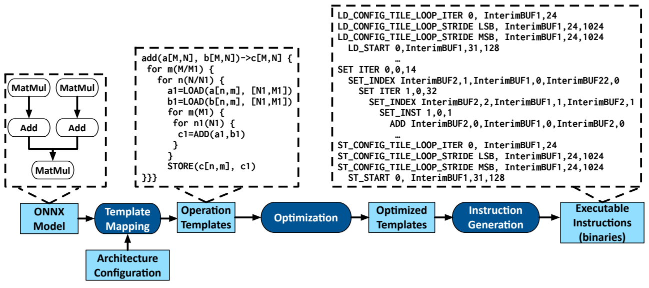
Off-chip data movement instructions. TILE_LD_ST opcode describes the data tile transfer between off-chip memory and Interim BUFs. The func1 field includes various fucntions: The LD/ST_CONFIG_BASE_ADDR function is used to generate the base addresses of each tile, then the shape and strides are configured using the LD/ST_CONFIG_BASE_LOOP_ITER/STRIDE functions. Also, LD/ST_CONFIG_TILE_LOOP_ITER/STRIDE functions are used to configure the Data Access Engine to generate the addresses required for each tile. Finally, LD/ST_START function triggers the Data Access Engine to start populating/draining the intermediate buffers. The func2 field is used to identify the target buffer between Interim BUF 1&2.

6 Compilation for the Tandem Processor

Tiling optimization. Compiler realizes software-pipelining by choosing the optimized tiling strategy. To improve the Tandem Processor's utilization, the compiler does not tile the reduction dimensions in GEMM operations. otherwise, the GEMM Unit produces partial results that would be insufficient for the Tandem Processor to perform its operations, causing it to stall. Additionally, the compiler finds the optimal sizes for tiles that are big enough to encompass all the adjacent elements of an input tensor for the non-GEMM operation, while small enough to fit on the limited on-chip scratchpads. For instance, to perform Depth-wise Conv operation with a kernel size 5×5, it would require the Tandem Processor to have access to all the elements in the 5×5 patch or it is inevitable to stall.

Dependency relaxation. The Tandem Processor leverages the regularity in the non-GEMM operations and eliminates the dependency check in the hardware, while shifting the burden to the compiler. The Tandem Processor compiler leverages *loop fission* [96] to remove dependencies among series of instructions. Additionally, some non-GEMM operations such as MaxPool has a long sequence of dependencies among instructions. For such cases, the compiler leverages *loop interchange* [96] to relax the dependencies.

Compilation workflow. Figure 13 describes the compilation workflow for the Tandem Processor. The compiler uses the ONNX format of DNNs and the architecture configuration of the Tandem Processor (e.g. number of lanes, Interim BUF) as its inputs. The compiler maps the ONNX node to pre-defined operation templates. However, as discussed in Section 3.4, not all non-GEMM operators are directly supported by the Tandem Processor. Therefore, for such complex operations (e.g., Softmax, Sqrt, Gelu) the compiler translates them to an integer-based counterpart [92, 93]. After mapping to the templates, the parameters of the operation templates are replaced with real values according to the ONNX layers. The compiler then performs the aforementioned optimizations. Finally, the compiler iterates the statements in the template and lowers them into instructions based on the Tandem Processor ISA.



```
add(a[M,N], b[M,N])->c[M,N] {
 for m(M/M1) {
   for n(N/N1) {
     a1=LOAD(a[n,m], [N1,M1])
     b1=LOAD(b[n,m], [N1,M1])
     for m(M1) {
      for n1(N1) {
       c1=ADD(a1,b1)
      }
     }
     STORE(c[n,m], c1)
}}}
```

```
LD_CONFIG_TILE_LOOP_ITER 0, InterimBUF1,24
LD_CONFIG_TILE_LOOP_STRIDE LSB, InterimBUF1,24,1024
LD_CONFIG_TILE_LOOP_STRIDE MSB, InterimBUF1,24,1024
  LD_START 0,InterimBUF1,31,128
           …
SET ITER 0,0,14
  SET_INDEX InterimBUF2,1,InterimBUF1,0,InterimBUF22,0
    SET ITER 1,0,32
      SET_INDEX InterimBUF2,2,InterimBUF1,1,InterimBUF2,1
        SET_INST 1,0,1
          ADD InterimBUF2,0,InterimBUF1,0,InterimBUF2,0
           …
ST_CONFIG_TILE_LOOP_ITER 0, InterimBUF1,24
ST_CONFIG_TILE_LOOP_STRIDE LSB, InterimBUF1,24,1024
ST_CONFIG_TILE_LOOP_STRIDE MSB, InterimBUF1,24,1024
  ST_START 0, InterimBUF1,31,128
```

Figure 13. Compilation workflow.

LOAD and STORE statements are lowered to TILE_LD_ST with BASE_LOOP_ITER/STRIDE functions for each LOOP variables, to set the number of iterations and strides in DRAM (a summarized version is shown in Figure 13). Then, the tile transfer instructions (LD/ST_CONFIG_TILE_LOOP_ITER/STRIDE) are generated using the tile shape in the LOAD and STORE statements. Compute operations are lowered to a set of inner LOOP instructions along with the pertinent compute instruction. For compute operations reading from Output BUF, the compiler generates additional synchronization instructions.

7 Evaluation Methodology

Benchmarks. To evaluate the efficacy of the Tandem Processor, we form our benchmark suite from domains of image classification (VGG-16 [62], ResNet-50 [74], MobileNetv2 [76], EfficientNet [77]), object detection (Yolov3 [75]), and emerging language models (BERT [63], GPT-2 [64]) with batch size 1 that is used for real-time AI [35], single-stream, and offline scenarios [97]. These DNN benchmarks constitute a diverse set of layers with various dimensions and types of operations (e.g. Relu/LeakyRelu/Clip, Maxpool/GlobalAveragePool, Depth-wise convolution, Residual Add, ReduceMean, Exp, Transpose, etc.)

Hardware implementation and synthesis. We implement the Tandem Processor in Verilog and synthesize it using Synopsys Design Compiler R-2020.09-SP4 with Global Foundries 65 nm library. We also perform place and route using Synopsys IC Compiler L-2016.03-SP1. Additionally, we synthesize the Tandem Processor with FreePDK 15nm open cell library and meet the 1 Ghz target frequency. To obtain power of the design, we use the synthesis results in FreePDK 15 nm for logic cells and model the on-chip memory energy using CACTI-P [98].

Table 3. Microarchitectural configurations for the NPU-Tandem.

Configs/Units	Systolic Array	Tandem Processor
Dimensions	32×32	32 Lane
Scratchpads	384 KB	128 KB (Interim BUF 1&2)
Accumulators	128 KB	N/A
Datatypes	INT8 (Mult) and INT32 (Acc)	INT32
Frequency	1 GHz	1 GHz

Simulation infrastructure. We develop a cycle-accurate simulator for the Tandem Processor that uses compiler-generated instructions and provides cycle counts and energy statistics. We validate the functionality of the simulator and RTL implementation by comparing the simulator/RTL-generated outputs with ground truth software implementation. These validations also show the closeness of the number of cycles by error margin of $\leq 5\%$. For end-to-end results, following the methodologies of [12, 54, 72, 99], we develop a cycle accurate simulator for a systolic array based GEMM Unit and integrate it with the Tandem Processor simulation infrastructure following the insights in Section 3.5.

Comparison to off-chip CPU fallback and dedicated units (Class (1) and (2) in Section 2.3) We compare the NPU-Tandem with the configurations listed in Table 3 to (1) a PCIe-attached (third generation with eight lanes) GEMM unit and an off-chip Intel Core i9-9980XE Extreme Edition CPU to support non-GEMM layers, (2) a GEMM unit augmented with a number of dedicated hardware blocks that support Relu, Clip, Residual Add, MaxPool, and scale & shift, similar to the design in [72]. This baseline still falls back to the CPU for unsupported layers. We measured the GEMM unit and dedicated units runtime using our aforementioned simulator and the CPU time using ONNX Runtime [100]. Finally, we measure the PCIe communication for all required data transfers in benchmarks using a Xilinx Alveo u280 FPGA connected to host CPU via PCIe. All baselines use the same frequency, number of PEs, and on-chip memories as in Table 3. For energy comparisons, we estimate the power of the GEMM unit using energy reports provided by prior works [69, 101], and model the energy of PCIe transactions according to [102].

Comparison to Gemmini [72] (Class (3) in Section 2.3). We compare the NPU-Tandem with Gemmini [72] that integrates a systolic array, a set of peripheral dedicated units (similar to those mentioned above), and a single RISC-V CPU core. We use ONNX Runtime [100] and cycle-accurate Firesim [103] simulator to obtain performance numbers for Gemmini. For a fair comparison, we exclude all the runtime/OS-related overheads. Additionally, for an iso-resource comparison, we use a scaled up Gemmini-like design that integrates the same number of cores as the number of ALU lanes in the Tandem Processor. To obtain the performance, we optimistically scale down the CPU runtime for Gemmini with the number of integrated cores.

Comparison to Google's VPU (Class (4) in Section 2.3). To eliminate the bias in comparisons due to the differences in the accelerators size, technology nodes, and the GEMM unit design and its optimizations, we model the behavior of Google's VPU within our simulation infrastructure according to Google's patent on VPU [85]. Concretely, we model the overheads of data communication (load/store instructions) between the scratchpads and vector register files in addition to nested loop execution due to the lack of specialized support for them on VPU. We modeled the benefits of using special functions in VPU for computing operations such as square root and exponential. TPU overlaps the execution of GEMM unit and VPU by forwarding the GEMM outputs through FIFOs to the VPU's scratchpads. We modeled this overhead.

Comparisons to GPUs (Class (4) in Section 2.3). We use NVIDIA's Jetson Xavier NX as a mobile GPU baseline and NVIDIA's RTX 2080 TI as a high-performance GPU baseline. We run all DNNs on GPU baselines using TensorRT v7.2.3. We also compare the performance of the NPU-Tandem to NVIDIA A100 GPU in an iso-TOPs (iso-resource) setting. We scale up both the GEMM (MAC units of GEMM unit) and non-GEMM (ALUs in the Tandem Processor) resources of the NPU-Tandem by 216× to match the TOPs of A100 for both GEMM and non-GEMM operations. As such, in this setting, both designs use the same amount of resources for GEMM and non-GEMM operations. We measure the runtime for A100 in two ways: (1) We use the TensorRT to obtain optimized end-to-end numbers for A100. However, TensorRT environment does not allow layer-wise DNN execution profiling to get the statistics on the breakdown of runtime across GEMM and non-GEMM operations. As such, (2) we also use ONNX Runtime with CUDA Execution Provider for runtime measurement and layer-wise profiling. We compare the NPU-Tandem end-to-end runtime to both measurements and use the CUDA-based results to further analyze the runtime based on its breakdown across GEMM and non-GEMM operations.

8 Experimental Results

Comparisons to offchip CPU fallback and dedicated units. Figure 14 compares performance of the NPU-Tandem with baselines (1) using offchip CPU fallback and (2) using dedicated units. The results are normalized to the baseline (1). On average, the NPU-Tandem provides 3.5× and 2.7× speedup compared to baseline (1) and baseline (2), respectively. The Tandem Processor not only eliminates the overheads of communication with offchip over PCIe and improving resource utilization, it also minimizes the overheads of instruction orchestration and data access compared to the general purpose CPU. The improvements provided by the Tandem Processor are more pronounced for MobileNet-v2 (5.9× over baseline (1) and 5.4× over baseline (2)) and BERT (5.4× over baseline (1) and 4.5× over baseline (2)) due to the use of more complex

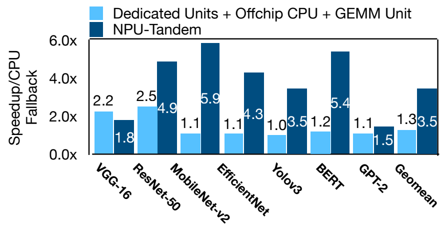


Figure 14. Performance comparison to offchip CPU fallback and dedicated units.

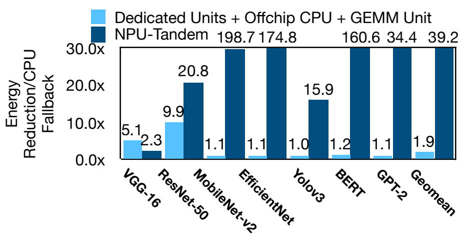


Figure 15. Energy reduction comparison.

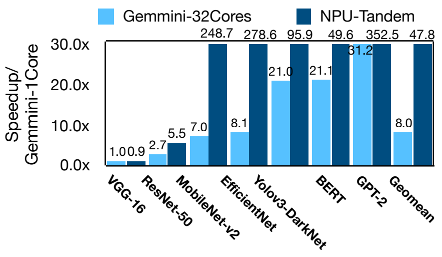


Figure 16. Comparison with Gemmini [72].

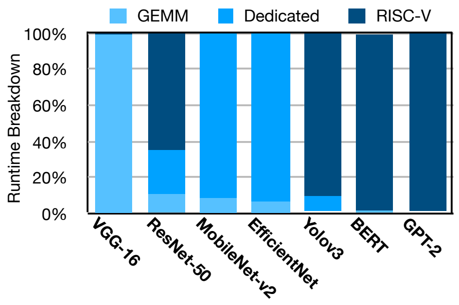


Figure 17. Gemmini time breakdown.

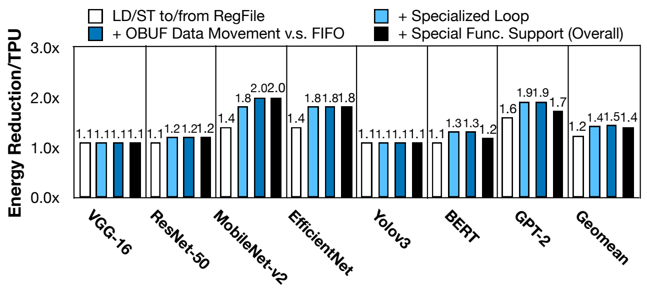


Figure 19. Energy reduction over TPU+VPU and analyzing the contribution of each Tandem Processor's specialization.

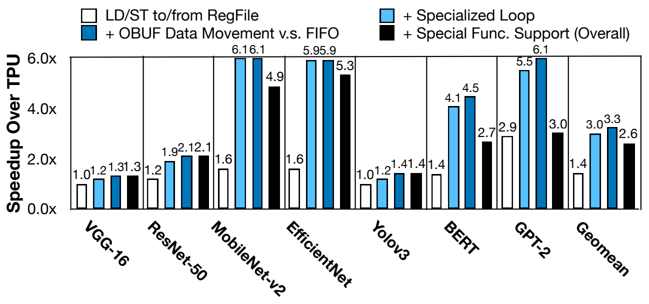


Figure 18. Performance comparison to TPU+VPU and analyzing the contribution of each Tandem Processor's specialization.

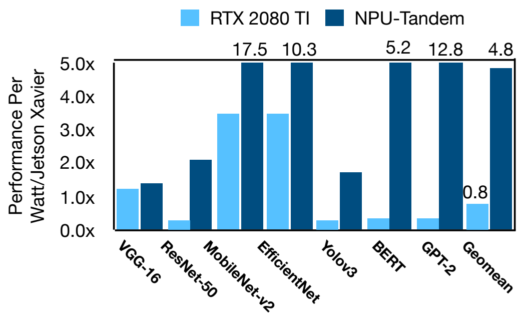


Figure 20. Comparisons to Jetson Xavier and RTX 2080-TI GPUs.

non-GEMM operations in their structure (depth-wise convolution in MobileNet-v2 and large number of mathematical and transpose operations in BERT) that significantly affect the total runtime. Figure 15 compares the energy reduction benefits of Tandem Processor. On average, the NPU-Tandem reduces the total energy consumption by 39.2× and 20.6× compared to baseline (1) and baseline (2), respectively. These large improvements are due to the significant time that baselines (1) and (2) spend on the power-hungry off-chip CPU (as shown in Figure 3) with a TDP of 165 Watts as opposed to 2.7 Watts in the Tandem Processor. *The results show that generally as DNNs evolve and use more complex structures and non-GEMM operations, the benefits of the Tandem Processor grow.*

Comparison to Gemmini. As Figure 16 shows, on average, the NPU-Tandem provides 47.8× performance improvements. Figure 16 also evaluates the improvements over an extended version of Gemmini that integrates the same number of RISC-V cores as the number of SIMD lanes in the Tandem Processor. On average, using multiple cores improves the performance of Gemmini by 8.0×. Compared to this design point, the NPU-Tandem provides 5.9× speedup, on average (with maximum of 35.3× for MobileNet-v2 and minimum of 0.9× for VGG-16).

To understand the sources of improvements, Figure 17[3] shows the runtime breakdown of Gemmini (default setting of one RISC-V core) across its three main components of GEMM unit, dedicated units, and RISC-V core. For MobileNet-v2 and EfficientNet, Gemmini spends a large amount of time (90% of runtime) on its im2col dedicated unit to convert the depth-wise convolutions to a series of GEMM operations. This not only requires a time-consuming im2col operation, but also results in additional GEMM operations with low resource utilization. On the other hand, the Tandem Processor executes

[3]Figure 24 shows the runtime breakdown for the NPU-Tandem.

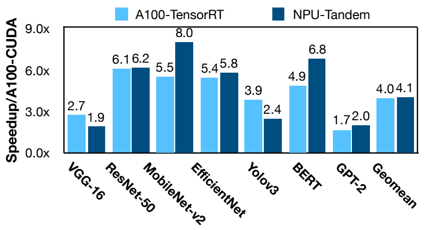


Figure 21. Performance comparison to A100 with CUDA and TensorRT execution, in iso-TOPs setting. Results are normalized to CUDA execution.

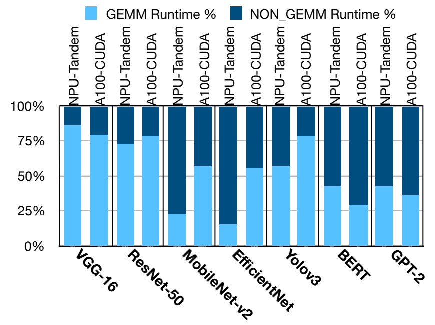


Figure 22. Runtime breakdown analysis for the scaled-up Tandem Processor and A100 GPU with CUDA execution in iso-TOPs setting.

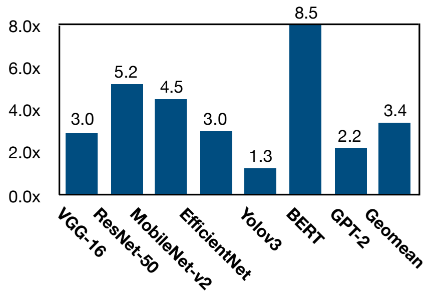


Figure 23. NPU-Tandem speedup for non-GEMM operations over A100 in iso-TOPs setting.

these operations natively and more efficiently without any need for im2col and overlaps them with other convolutions, as well. For YoloV3, BERT, and GPT-2 RISC-V core is the bottleneck. These DNNs require a significant number of complex mathematical operations such as Leaky ReLU in YoloV3 and GeLu, ReduceMean, Sqrt, Softmax, etc, in BERT and GPT-2, not supported by dedicated units. Note that, Gemmini uses one single RISC-V core (with 40% more area than the 32-lane Tandem Processor), which has one ALU to process all these operations on large tensors. For ResNet-50, still RISC-V core is the bottleneck, because of the last AveragePool layer (this layer takes the average of 7×7 feature maps for 2048 channels). In contrast, the Tandem Processor minimizes the cost of these operations and seeks to overlap them with GEMM ones. *These results show that for DNNs with more complex non-GEMM layers, paying the cost of PCIe and using a high-performance offchip CPU (and dedicated units) provides better performance than an on-chip CPU in Gemmini.*

Performance comparison to Google's TPU. Figure 18 compares the end-to-end performance of the NPU-Tandem to a TPU-like design that leverages the general-purpose VPU for non-GEMM layers. According to the Google's patent on VPU [85], we considered the following specializations for TPU: 1) strided address generation for LD/ST between DRAM and scratchpad, 2) strided address generation for LD/ST between scratchpad and vector register file, 3) software-pipelining of GEMM and non-GEMM through FIFOs, and 4) supporting specialized instructions for mathematical functions such as exp, sqrt, clip, etc.. As such, the benefits of Tandem Processor over VPU stem from 1) removing the vector register file and its LD/ST overheads, 2) supporting specialized nested loop execution, and 3) software-pipelining through reading from OBUF directly as opposed to FIFOs. On the other hand, supporting special functions in VPU can boost its performance over the NPU-Tandem. Figure18 analyzes the impacts of these four design decisions individually. For each benchmark four bars are reported. The first bar shows the speedup achieved only by removing RegFile and its LD/ST overheads, the second shows the impact of specialized loop execution on top of the RegFile LD/ST, the third shows speedup when the benefits of OBUF data movement is also considered on top of two previous decisions, and finally the last bar includes the slowdown impact of not supporting specialized functions as well. In another word, the last bar includes the impacts of the four design decisions and is the *final end-to-end speedup*. On average, the NPU-Tandem offers 2.6× speedup. Among the four design decisions, supporting specialized loop execution in the Tandem Processor provides the maximum speedup, 2.1× on average. The benefits due to this design decision are more pronounced for MobileNet-v2 and EfficientNet with depth-wise convolution layers, an operation with five nested loops. The second most effective technique is eliminating the register file and its associated LD/ST operations from/to scratchpad, providing 1.4× speedup on average. GPT-2 enjoys the maximum benefits from this specialization with 2.9× speedup. Direct data access through OBUF in the Tandem Processor as opposed to moving data through FIFOs across GEMM unit and VPU, provides 1.1× speedup on average, while not having hardware support for special functions causes 0.8× slowdown on average. Having the hardware support and dedicated instructions for special functions provides maximum benefits for VPU for BERT and GPT-2, since complex mathematical operations such as sqrt and exp (for softmax) are heavily used in their structure. Note that this speedup comes at the cost of extra area and design complexity for VPU which its quantification would require access to the exact hardware implementation that is not publicly available. Overall considering the impact of four design decisions, MobileNet-v2, EfficientNet, and GPT-2 show the most benefits for NPU-Tandem, while VGG-16 showing the least.

Energy comparison to Google's TPU. Figure 19 shows the end-to-end energy reduction achieved by the NPU-Tandem over TPU+VPU while analyzing the impact of the aforementioned design decisions individually. On average, the NPU-Tandem provides 1.4× energy reduction. Among the benchmarks the benefits are more pronounced for MobileNet-v2, EfficientNet, and GPT-2 (2.0×, 1.8×, and 1.7×, respectively), while VGG-16 and Yolov3 observes the minimum benefits

(1.1×). Eliminating the RegFile and its LD/ST overheads provides the maximum energy reduction with the average of 1.2×. Specialized support for nested loop is the second most effective technique. Although specialized loop management provides significant speedups, its energy benefits are less pronounced due to their amortization across the SIMD lanes of the VPU. Support for specialized functions in VPU realizes 7% lower energy for TPU, on average, by replacing several primitive operations with a single yet more complex instruction.

Comparison to Jetson Xavier and RTX 2080 TI GPUs. Figure 20 compares the performance-per-Watt benefits with Jetson Xavier NX and RTX 2080 TI GPUs, where the results are normalized to Jetson Xavier. RTX 2080 TI is less energy-efficient compared to mobile Jetson Xavier (20% lower on average). However, the NPU-Tandem provides 4.8× improvements, compared to Jetson Xavier. The trends in the results remain almost similar to the previous analyses with MobileNet-v2 exhibiting the maximum benefits. RTX 2080 TI is more efficient than Jetson Xavier for MobileNet-v2 and EfficientNet, because it can better parallelize the depth-wise convolutions across its abundant threads, compared to Jetson Xavier that employs relatively less number of threads.

Comparison to the A100 GPU. Figure 21 compares the end-to-end speedup of the NPU-Tandem to the A100 GPU with TensorRT and CUDA execution in an iso-TOPs setting. On average, the NPU-Tandem offers similar performance to A100 GPU with TensorRT execution (2.5% improvements) and 4.0× speedup compared to the A100 with CUDA execution. The NPU-Tandem outperforms A100 with TensorRT for ResNet-50, MobileNet, EfficientNet, BERT, and GPT-2, while A100 providing better performance for VGG-16 and Yolov3 that are mainly composed of heavy GEMM operations. Compared to A100 with CUDA execution, the NPU-Tandem provides maximum benefits for MobileNet-v2 and BERT.

Figure 22 shows the runtime breakdown across GEMM and non-GEMM operations for the NPU-Tandem and A100 GPU with CUDA execution. The NPU-Tandem accelerates both GEMM and non-GEMM operations compared to A100 with CUDA execution. However, for DNNs that have larger portion of non-GEMM runtime on A100 (e.g., MobileNet, EfficientNet, BERT, and GPT-2), NPU-Tandem provides larger end-to-end speedups, demystifying the impact of accelerating the non-GEMM operations using the Tandem Processor. This trend in speedups still holds while comparing to TensorRT as well, since the benchmarks mentioned above are those with the largest speedups by the NPU-Tandem with respect to this mode of execution (see Figure 21).

Figure 23 compares the performance of the Tandem Processor to A100 CUDA Cores for performing only non-GEMM operations of benchmark DNNs in an iso-TOPs/resources setting. The Tandem Processor accelerates the non-GEMM operations for all benchmarks and on average provides 3.4× speedup compared to A100 CUDA Cores. The benefits are more pronounced for BERT (8.0×), ResNet-50 (5.2×), and MobileNet-v2 (4.5×). Although GPT-2 comprise a large portion of non-GEMMs similar to BERT, the performance of scaled-up Tandem Processor is mainly bounded by the memory bandwidth for this DNN and hence showing relatively lower speedup compared to BERT. The results show that the Tandem Processor's unique microarchitectural design effectively reduces the overheads of data delivery (LD/ST to RegFile) and required address calculations and loop managements for non-GEMM operations.

Runtime breakdown analysis for the Tandem Processor. Figure 24 shows the runtime breakdown of the NPU-Tandem across GEMM and various non-GEMM layers. As the result show, non-GEMM layers are very diverse in terms of execution runtime. The proposed specializations in the Tandem Processor significantly reduce the overhead of non-GEMM layers in DNNs such as VGG-16, ResNet-50, and Yolov3. On the other hand, some DNN layers such as depthwise convolution in MobileNet-v2 and EfficientNet, GELU and transpose in BERT, and ReduceMean in GPT-2 still take a significant portion of runtime. Compared to the baselines, GEMM layers only become a more significant runtime component, when the highly specialized Tandem Processor is used to reduce the overheads of non-GEMM layers.

Energy breakdown analysis for the Tandem Processor. Figure 25 shows the energy breakdown of the Tandem Processor across off-chip memory accesses, on-chip memory (Interim BUF) accesses, ALU logic, loop + address calculation logic, and the rest of the Tandem Processor logic (decode, muxing logic, etc.) Although non-GEMM layers are memory bound operations, but off-chip memory accesses take about only 31% of the total energy on average, due to the seamless integration of Tandem Processor and the GEMM unit which minimizes the number of off-chip data transfer. The on-chip memory accesses take 13% of the total energy, on average, due to removing overheads of register files and associated memory hierarchy from the design. ALU logic takes 12% of the total energy because of leveraging integer primitive implementation philosophy in its design. Overall, the nested loop execution control and scratchpad address calculation logic takes the majority of the energy consumption in the Tandem Processor (40%), since they handle the heavy lifting portion of the overall execution.

The Tandem Processor layout. Fig. 26(a) shows the layout of the Tandem Processor occupying 1.02 mm^2. Fig. 26(b) shows the post-layout area breakdown. ALU logic occupies the largest area (56.6%), Interim BUF 1 & 2 is the second (29.2%) and the permute logic is the third (12.0%). The rest of the area is mainly for muxing logic, pipeline registers, Code Repeater and decode logic.

9 Related Work

Section 2.3 covers the related work on supporting non-GEMM layers. Below, we discuss the prior work on SIMD/vector units.

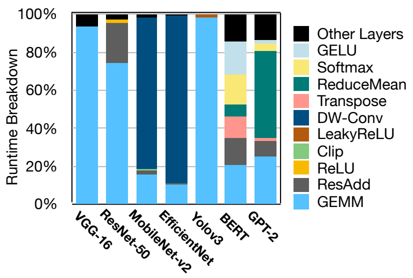


Figure 24. Runtime breakdown of the Tandem Processor.

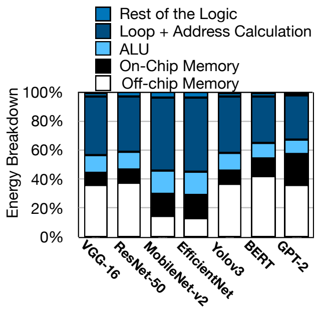


Figure 25. Energy breakdown of the Tandem Processor.

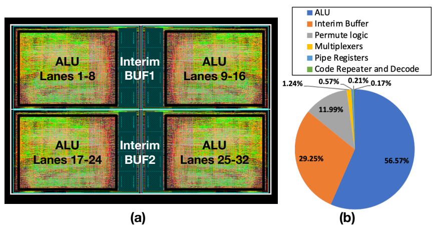


Figure 26. (a) Tandem Processor layout and (b) area breakdown in 65nm node.

Designing general-purpose SIMD units, vector ISA extensions, and compilation for them have been largely explored in academia [104–112] and industry products such as Intel AVX-512 [113], ARM SVE [114], RISC-V vector extensions [115], and etc.. Digital Signal Processors (DSPs) [90, 91, 116–120] are more specialized SIMD units that often come with VLIW architectures. Qualcomm Hexagon DSP [91] and MediaBreeze DSP [90] provide hardware-managed loop executions that work with their register file/FIFOs. MediaBreeze [90] also leverages a decoupled access-execute architecture to handle address generation for streams of data, which are fed into SIMD ALUs through FIFOs. ARM Helium [120] incorporates a set of DSP extensions such as low-overhead branch and scatter-store/gather-load instructions. In contrast, our design completely departures from register-file-memory semantics. This fundamental design choice enables Tandem Processor to eliminate explicit address calculations that are conventionally carried over registers and replace them with a customized loop logic. Additionally, the front-end of the pipeline in Tandem Processor handles memory access while in conventional designs this is normally in the back-end stages. This is also different from the prior SIMD units with Access-Execute architectures (e.g. MediaBreeze) that pass data to the execute units through FIFOs. In Tandem Processor, Access and Execute are part of the same pipeline and there are no FIFOs. Finally, the combination of not using register files/FIFOs with the loop logic is a new design feature in Tandem Processor.

10 Conclusion

The increasing prevalence of neural networks and advancements in language models prompt a reevaluation of neural accelerator design. In the last ten years, the research community has primarily concentrated on GEMM operations while overlooking non-GEMM operations. This has created a misconception that neural networks are solely composed of matrix multiplications. Furthermore, as deep learning has evolved and entered new domains, the non-GEMM operations have diversified and been interwoven in various structural patterns within neural networks. As such, to run neural networks end-to-end, there has been a need to consider non-GEMM layers as a first class citizen. To address this timely need, this paper proposes the Tandem Processor that brings forth a novel architecture along with a compiler and an innovative programmable ISA. Moreover, this architecture, which is the result of 10 years of research in building NPUs also enables adapting to the volatile landscape of deep learning algorithms. The Tandem Processor has become the heart of our open-source GeneSys project, a parametrizable NPU *generator* with a full-stack, multi-target compilation stack that goes from Python to accelerated execution of LLMs and other DNNs. GeneSys provides comprehensive NPU solutions for applications ranging from high-end datacenters to ultra-low-power brain-implantable devices and is publicly available at https://actlab-genesys.github.io/.

Acknowledgments

We thank Hyoukjun Kwon for shepherding our paper. This work was in part supported by generous gifts from Google, Samsung, Qualcomm, Microsoft, AMD Xilinx as well as the National Science Foundation (NSF) awards CCF#2107598, CNS#1822273, Defense Advanced Research Project Agency (DARPA) under agreement number #HR0011-18-C-0020, National Institute of Health (NIH) award #R01EB028350, and Semiconductor Research Corporation (SRC) award #2021-AH-3039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of Google, Qualcomm, Microsoft, Xilinx, Samsung, NSF, SRC, NIH, DARPA or the U.S. Government. Soroush Ghodrati was partly supported by a Google PhD Fellowship.

References

[1] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *MICRO*, 2016.

[2] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, 2016.

[3] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[4] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *ISCA*, 2017.

[5] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ASPLOS*, 2018.

[6] Vahide Aklaghi, Amir Yazdanbakhsh, Kambiz Samadi, Hadi Esmaeilzadeh, and Rajesh K. Gupta. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *ISCA*, 2018.

[7] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *JETCAS*, 2019.

[8] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. *HPCA*, 2020.

[9] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. Aˆ 3: Accelerating attention mechanisms in neural networks with approximation. In *HPCA*, 2020.

[10] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *HPCA*, 2021.

[11] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, 2016.

[12] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. *ISCA*, 2018.

[13] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *MICRO*, 2017.

[14] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *DAC*, 2018.

[15] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *ASPLOS*, 2019.

[16] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *ISCA*, 2019.

[17] Sungju Ryu, Hyungjun Kim, Wooseok Yi, and Jae-Joon Kim. Bitblade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation. In *DAC*, 2019.

[18] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ISCA*, 2016.

[19] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ISCA*, 2016.

[20] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, et al. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *ASPLOS*, 2019.

[21] Geng Yuan, Payman Behnam, Zhengang Li, Ali Shafiee, Sheng Lin, Xiaolong Ma, Hang Liu, Xuehai Qian, Mahdi Nazm Bojnordi, Yanzhi Wang, et al. Forms: fine-grained polarized reram-based in-situ computation for mixed-signal dnn accelerator. In *ISCA*, 2021.

[22] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. Redeye: analog convnet image sensor architecture for continuous mobile vision. In *ISCA*, 2016.

[23] Prakalp Srivastava, Mingu Kang, Sujan K Gonugondla, Sungmin Lim, Jungwook Choi, Vikram Adve, Nam Sung Kim, and Naresh Shanbhag. Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms. In *ISCA*, 2018.

[24] Soroush Ghodrati, Hardik Sharma, Sean Kinzer, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, Doug Burger, and Hadi Esmaeilzadeh. Mixed-signal charge-domain acceleration of deep neural networks through interleaved bit-partitioned arithmetic. In *PACT*, 2020.

[25] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *ISCA*, 2016.

[26] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ISCA*, 2018.

[27] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *ISCA*, 2019.

[28] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*, 2017.

[29] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[30] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*, 2016.

[31] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kim, Chenkai Shao, Asit Misra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *MICRO*, 2016.

[32] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*, 2017.

[33] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[34] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. *arXiv*, 2018.

[35] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.

[36] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *ASPLOS*, 2019.

[37] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W Fletcher. Morph: Flexible acceleration for 3d cnn-based video understanding. In *MICRO*, 2018.

[38] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *MICRO*, 2019.

[39] HT Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *ASPLOS*, 2019.
[40] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *MICRO*, 2017.
[41] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.
[42] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. DVAFS: Trading Computational Accuracy for Energy Through Dynamic-Voltage-Accuracy-Frequency-Scaling. In *DATE*, 2017.
[43] Daniel Bankman, Lita Yang, Bert Moons, Marian Verhelst, and Boris Murmann. An always-on 3.8 μj/86% cifar-10 mixed-signal binary cnn processor with all memory on chip in 28nm cmos. In *ISSCC*, 2018.
[44] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. Prediction based execution on deep neural networks. In *ISCA*, 2018.
[45] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*, 2015.
[46] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *ISCA*, 2018.
[47] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. Diffy: A déjà vu-free differential deep neural network accelerator. In *MICRO*, 2018.
[48] Jiaqi Zhang, Xiangru Chen, Mingcong Song, and Tao Li. Eager Pruning: Algorithm and Architecture Support for Fast Training of Deep Neural Networks. In *ISCA*, 2019.
[49] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks. In *MICRO*, 2020.
[50] Rohan Mahapatra, Soroush Ghodrati, Byung Hoon Ahn, Sean Kinzer, Shu ting Wang, Hanyang Xu, Lavanya Karthikeyan, Hardik Sharma, Amir Yazdanbakhsh, Mohammad Alian, and Hadi Esmaeilzadeh. In-storage domain-specific acceleration for serverless computing. *ASPLOS*, 2024.
[51] Zheng Li, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, and Mingu Kang. Accelerating Attention through Gradient-Based Learned Runtime Pruning. In *ISCA*, 2022.
[52] Soroush Ghodrati, Hardik Sharma, Cliff Young, Nam Sung Kim, and Hadi Esmaeilzadeh. Bit-parallel vector composability for neural acceleration. In *DAC*, 2020.
[53] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, 2017.
[54] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv*, 2018.
[55] OpenAI. Chatgpt. https://chat.openai.com, 2023.
[56] Google. Bard: A conversational ai tool by google. https://bard.google.com, 2023.
[57] Adobe. Your imagination's new best friend. https://www.adobe.com/products/firefly.html, 2023.
[58] Microsoft. Github copilot: Your ai pair programmer. https://github.com/features/copilot, 2023.
[59] Designs.ai. Ai-powered text-to-video - turn text into stunning videos. https://designs.ai/, 2023.
[60] Meta. Introducing audiocraft: A generative ai tool for audio and music. https://about.fb.com/news/2023/08/audiocraft-generative-ai-for-music-and-audio/, 2023.
[61] Microsoft. Reinventing search with a new ai-powered microsoft bing and edge, your copilot for the web. https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/, 2023.
[62] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.
[63] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv*, 2018.
[64] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 2019.
[65] Nvdla. http://nvdla.org/index.html.
[66] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training. In *MICRO*, 2020.
[67] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *MICRO*, 2020.
[68] NVIDIA. Nvidia turing architecture in-depth. https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/, 2022.
[69] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *ISCA*, 2021.
[70] Tesla. Dojo chip. https://www.tesla.com/AI, 2022.
[71] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, et al. RaPiD: AI Accelerator for Ultra-low Precision Training and Inference. In *ISCA*, 2021.
[72] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *DAC*, 2021.
[73] H Ekin Sumbul, Tony F Wu, Yuecheng Li, Syed Shakib Sarwar, William Koven, Eli Murphy-Trotzky, Xingxing Cai, Elnaz Ansari, Daniel H Morris, Huichu Liu, et al. System-Level Design and Integration of a Prototype AR/VR Hardware Featuring a Custom Low-Power DNN Accelerator Chip in 7nm Technology for Codec Avatars. In *CICC*, 2022.
[74] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
[75] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.
[76] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.
[77] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *ICML*, 2019.
[78] Facebook Research Microsoft. Onnx: an open format to represent deep learning models. http://onnx.ai/, 2017.
[79] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.
[80] Rui Xu, Sheng Ma, Yaohua Wang, Yang Guo, Dongsheng Li, and Yuran Qiao. Heterogeneous Systolic Array Architecture for Compact CNNs Hardware Accelerators. *IEEE Transactions on Parallel and Distributed Systems*, 2021.
[81] M. Abadi et al. TensorFlow: A system for large-scale machine learning. *OSDI*, 2016.
[82] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*, 2018.
[83] A. Paszke et al. PyTorch: An imperative style, high-performance deep learning library. *NeurIPS*, 2019.

[84] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *PLDI*, 2021.

[85] William Lacy, Gregory Michael Thorson, Christopher Aaron Clark, Norman Paul Jouppi, Thomas Norrie, and Andrew Everett Phelps. Vector Processing Unit. *U.S Patent 11520581*, 2022.

[86] SiFIve. Introduction to the sifive intelligence x280. https://www.sifive.com/blog/introduction-to-the-sifive-intelligence-x280, 2022.

[87] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, et al. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[88] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–151, 2019.

[89] Amir Yazdanbakhsh, Hajar Falahati, Philip J. Wolfe, Kambiz Samadi, Hadi Esmaeilzadeh, and Nam Sung Kim. GANAX: A Unified SIMD-MIMD Acceleration for Generative Adversarial Network. In *ISCA*, 2018.

[90] Deependra Talla, Lizy Kurian John, and Doug Burger. Bottlenecks in multimedia processing with simd style extensions and architectural enhancements. *IEEE Transactions on Computers*, 52(8):1015–1031, 2003.

[91] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. Hexagon dsp: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, 34(2):34–43, 2014. doi: 10.1109/MM.2014.12.

[92] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. In *ICML*, 2021.

[93] gemmlowp: a small self-contained low-precision gemm library, 2022. https://github.com/google/gemmlowp.

[94] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[95] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, et al. Hawq-v3: Dyadic neural network quantization. In *International Conference on Machine Learning*, pages 11875–11886. PMLR, 2021.

[96] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, 1997.

[97] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. *arxiv*, 2019.

[98] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *ICCAD*, 2011.

[99] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. A systematic methodology for characterizing scalability of dnn accelerators using scale-sim. In *ISPASS*, 2020.

[100] ONNX Runtime developers. ONNX Runtime, 11 2018. URL https://github.com/microsoft/onnxruntime.

[101] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[102] Noah Beck, Sean White, Milam Paraschou, and Samuel Naffziger. 'zeppelin': An soc for multichip architectures. In *ISSCC*, 2018.

[103] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *ISCA*, 2018.

[104] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[105] Basil Sh. Mahmood and Mamoon A. Al Jbaar. Design and implementation of simd vector processor on fpga. In *International Symposium on Innovations in Information and Communications Technology*, pages 124–130, 2011. doi: 10.1109/ISIICT.2011.6149607.

[106] A. Danysh and D. Tan. Architecture and implementation of a vector/simd multiply-accumulate unit. *IEEE Transactions on Computers*, 54(3):284–293, 2005. doi: 10.1109/TC.2005.41.

[107] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 36(5): C401–C423, 2014.

[108] Alexandre E Eichenberger, Peng Wu, and Kevin O'brien. Vectorization for simd architectures with alignment constraints. *Acm sigplan notices*, 39(6):82–93, 2004.

[109] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W Ueberhuber. Efficient utilization of simd extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005.

[110] Gaurav Mitra, Beau Johnston, Alistair P Rendell, Eric McCreath, and Jun Zhou. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1107–1116. IEEE, 2013.

[111] Amir Morad, Leonid Yavits, and Ran Ginosar. Gp-simd processing-in-memory. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2015.

[112] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. Spert-ii: a vector microprocessor system. *Computer*, 29 (3):79–86, 1996. doi: 10.1109/2.485896.

[113] Intel advanced vector extensions (avx). https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[114] Arm scalable vector extension (sve). https://developer.arm.com/documentation/102476/0100.

[115] Risc-v vector extensions. https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc.

[116] Jeff H Derby and Jaime H Moreno. A high-performance embedded dsp core with novel simd features. In *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03).*, volume 2, pages II–301. IEEE, 2003.

[117] Masaki Toyokura, Hisahi Kodama, Eiji Miyagoshi, Koyoshi Okamoto, Masahiro Gion, Takayuki Minemaru, A Ohtani, T Araki, H Takeno, T Akiyama, et al. A video dsp with a macroblock-level-pipeline and a simd type vector-pipeline architecture for mpeg2 codec. *IEEE journal of solid-state circuits*, 29(12):1474–1481, 1994.

[118] Huy Nguyen and Lizy Kurian John. Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology. In *Proceedings of the 13th international conference on Supercomputing*, pages 11–20, 1999.

[119] E Matus, Hendrik Seidel, Torsten Limberg, Pablo Robelly, and G Fettweis. A gflops vector-dsp for broadband wireless applications. In *IEEE Custom Integrated Circuits Conference 2006*, pages 543–546. IEEE, 2006.

[120] Joseph Yiu. Blending dsp and ml features into a low-power general-purpose processor – how far can we go? *White Paper*, 2020.