Accelerating General-Purpose Linear Algebra on DNN Accelerators

Alon Amid†§, Hasan Genc*, Jerry Zhao*, Krste Asanović*, Borivoje Nikolić*, Yakun Sophia Shao*
*University of California, Berkeley, †Microsoft
{alonamid,hngenc,jhz,krste,bora,ysshao}@berkeley.edu

***Abstract*—Deep learning inference and training tasks are often accompanied by additional numerical data analysis tasks such as clustering, dimensionality reduction, data transformation, and linear modeling. While matrix engines are primarily designed with deep neural network workloads in mind, they have also been used to accelerate general-purpose matrix processing workloads. The matrix multiplication components of numerical data analysis workloads vary in matrix shapes, sizes, and layouts compared to deep neural network models. In this wide problem space, subtle static scheduling or system-level effects generate variable memory-latency behavior observed by the accelerator in small matrix size regimes, leading to up to a 30% degradation in accelerator utilization. We observe that minor modifications to a matrix accelerator's hardware controller can substantially improve the suitability of the accelerator for these problem types, and demonstrate up to a $1.25\times$ improvement in the utilization of a matrix engine on small matrices through hardware-managed static scheduling, and up to a $1.15\times$ improvement through dynamic scheduling and hardware-managed commutative micro-threading, helping improve the utilization of matrix engines for general purpose linear algebra workloads.**

I. Introduction

While deep neural networks (DNNs) have dominated the machine learning domain in the past decade, they have not completely displaced traditional numerical data modeling techniques. DNNs have captured much of the attention of the hardware community in recent years due to their high computational cost dominated by a small number of high-arithmetic-intensity kernels, but they are regularly accompanied with complete data processing pipelines [15], often consisting of additional data modeling and analysis techniques. Dimensionality reduction [16] and clustering [11], [22] during pre- or post-processing, exploratory or explainable modeling using linear models [6], and transformations based on system of equations, all integrate into data modeling and analyses pipelines.

The number of dedicated accelerators on SoCs has steadily increased in the past decade, with specialized accelerators accounting for over 60% of the area in recent SoC designs [17]. At the same time, these accelerators experience low, bursty utilization (with respect to absolute time), due to their single-function/few-function design goals. This work focuses on expanding the use of DNN accelerators for general-purpose linear algebra and numerical data analysis [7], [10], [23]–[25].

II. Matrix Engines for Data Analysis vs. DNNs

Numerical data analysis methods and tools typically rely on several core numerical linear algebra algorithms such as exact or least-squares solutions of linear systems, together with basic matrix operations such as matrix multiplications and matrix factorizations such as Choleksy, LU, QR and singular value decomposition (SVD). While both DNNs and the linear algebra kernels at the basis of numerical data analysis workloads are dominated by matrix-matrix operations, they differ in several key characteristics which limit the efficiency of DNN accelerators for such a mix of workloads. In particular, the spectrum and diversity of matrix shapes and sizes that need to be processed highlights differences between DNNs vs. the broader category of numerical data analysis.

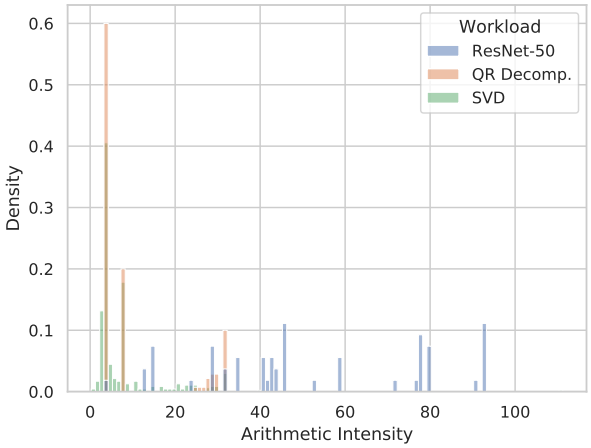

Fig. 1: Arithmetic intensity histogram of matrix-matrix operations (GEMM, TRMM) within a batch-1 ResNet-50 DNN forward pass, and a blocked-Householder QR decomposition and SVD of the UCI Human Activity Recognition training dataset with block-size 32 (the LAPACK default). Operands are 16-bit floating point datatypes, and results are 32-bit floating point datatypes.

In order to obtain high performance in systems with multi-level memory hierarchies, optimized linear algebra libraries use blocking techniques or recursive implementations to reduce communication across the memory hierarchy. Cache blocking through outer products in matrix multiplication is a popular blocking technique. However, blocking techniques and recursive implementations are also used in more complex matrix decompositions such as Cholesky, LU, and QR factorizations with the goal of operating on "blocks" using high-arithmetic-intensity matrix-matrix operations. LAPACK [2], a widely used open-source high-performance numerical computing library, uses blocking techniques to boost the arithmetic intensity of a series of matrix-vector operations by grouping them into matrix-matrix operations. However, the resulting matrix operations can differ in their shapes and sizes from matrices found in typical DNN models.

In particular, these blocking and reduction techniques can generate non-square matrices which cause numerical data analysis workloads to exhibit much lower arithmetic intensity than DNN workloads. Figure 1 illustrates the difference in arithmetic intensity between the matrix multiplications in ResNet-50, as a representative DNN, compared to common matrix decompositions used in data analysis applications, performed on a data matrix from the UCI Human Activity Recognition dataset [3]. Even though the DNN is run with the smallest possible batch size of 1, it still achieves a much larger arithmetic intensity than the matrix decompositions, which are

§Work done while at the University of California, Berkeley

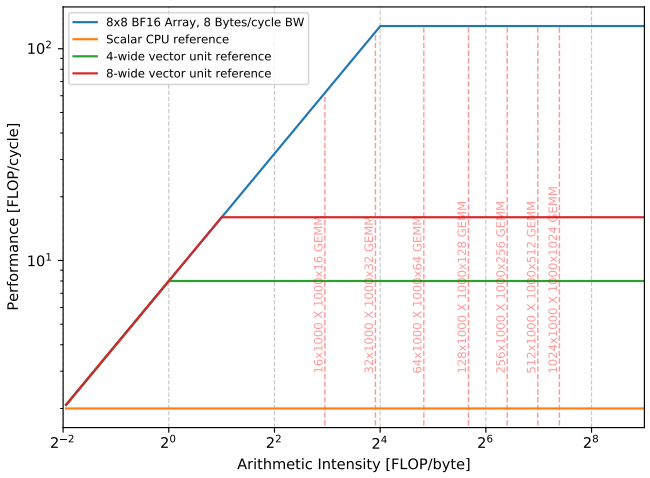

Fig. 2: Theoretical peak performance roofline model for a representative DNN matrix engine, in comparison to reference CPU and vector units. DRAM bandwidth for all design points is 8 bytes/cycle.

based on the default LAPACK implementations of SVD and QR decompositions with default block-sizes of 32.

A deeper analysis of these workloads reveals that matrix decompositions contain a diverse set of small and rectangular matrix shapes, while the DNN model's matrices are less rectangular. For example, using a notation of $M \times K$ times $K \times N$ matrix operations, in the QR decomposition we observe many operations with small values of M and N but large values of K, as well as operations with a large value of M but with small values of K and N. A collection of triangular matrix multiplication (`TRMM`) operations in the matrix decompositions particularly represent a series of smaller operations, with the triangular matrix dimensions being equal to the block size (32). In contrast, in DNN inference, the dimensions of many layers are of the same order of magnitude. This is the case for ResNet-50, except for the first few layers, which exhibit large values of M with small values of K and N, and the last layers which exhibit small values of M with larger values of K and N.

Increasing the block-size increases the arithmetic intensity of matrix factorizations, but in some cases can come at the cost of an increased number of floating-point operations that the blocked algorithms must perform on the exact same input. While some matrix factorizations, such as LU, can simply re-arrange operations to obtain higher arithmetic intensity, others, such as QR and SVD, require additional arithmetic steps in order to group operations into matrix-matrix procedures. This leads to a tradeoff between the desire to increase the block size in order to achieve higher utilization of the matrix engine in the DNN accelerator through higher arithmetic intensity vs. potentially increasing the operation count so much that it outweighs the benefits of faster and more efficient execution on the accelerator. However, these small block sizes can present a challenge for matrix engines in DNN accelerators, whose performance depends on the arithmetic intensity of *each* of the matrix operations executed on the accelerator. For matrix factorization algorithms, small block sizes generate small and non-square matrix shapes, which limit the arithmetic intensity and data re-use within a dedicated matrix multiplication accelerator. Zhang et al. [24] make a similar observation within the context of NVIDIA GPU tensor cores, and demonstrate the decrease in performance beyond a certain optimal block size tuned for NVIDIA GPUs.

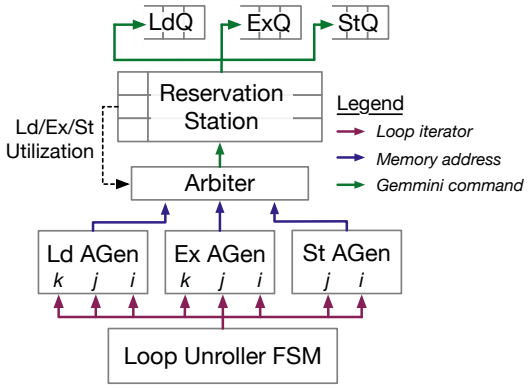

Fig. 3: Gemmini matrix multiplication hardware FSM controller.

Figure 2 illustrates the roofline model [21] of an 8×8 matrix engine (with bfloat16 operands and single-precision accumulators), compared to reference vector units (128-bit datapath, and 256-bit datapath) and a scalar CPU. The arithmetic intensity of matrix multiplication for several matrix shapes is noted on the diagram for reference, with the shared dimension remaining constant (1000), while the product dimensions are increased (similar to the increase of the block size). As Figure 2 indicates, while blocking factors of 32 may well be within the realm of compute-bound problems for traditional CPUs and vector units, modern DNN matrix engines require a higher level of arithmetic intensity for peak utilization. These heterogeneous matrix shapes and sizes come also into play within higher-level layers of the data analysis software stack, such as in k-means clustering, where "tall-and-narrow" data matrices can lead to low accelerator utilization.

In order to maximize performance and utilization using matrix engines within both compute-bound and memory-bound regimes across a diversity of matrix shapes and sizes, the scheduling of compute and memory operations on these accelerators needs to be customized to meet the workload requirements.

III. Matrix Engine Controller Scheduling

DNN accelerators typically utilize a controller to schedule memory transactions and compute resources within the accelerator. These controllers range from fully programmable processors to fixed hardware finite state machines (FSMs), including potential hierarchies of controllers within an accelerator, enabling different levels of programmability [4], [5], [14], [18], [20]. For example, the Gemmini DNN accelerator [8], which we focus on for the evaluation in this work, is equipped with a FSM which divides a large matrix multiplication problem (defined as $C = AB + D$), into a sequence of smaller matrix multiplications executed on a spatial array (of dimension $DIM \times DIM$). Each of the smaller operations, which we refer to as individual "Gemmini commands", can be at most $DIM \times DIM$ large, and is issued to either an execution queue, which performs these small multiplications, or a load or store queue, which performs DMA transactions. Figure 3 illustrates the high-level structure of the Gemmini matrix multiplication hardware controller.

The scheduling of memory and compute operations on accelerator resources has a direct impact on the overall utilization of the accelerator. The scheduling of matrix multiplication operations on CPUs has been extensively researched with evaluation choices and placements of stationary and streaming data across the memory hierarchy [9]. The scheduling of matrix multiplications on DNN accelerators exhibits similar characteristics, with the addition of constraints set by the

accelerator's spatial resources and fixed interconnects, as well as the private accelerator memory management policy [12]. Recent work by Jeong et al. [13] has identified some of the under-utilization challenges of systolic array matrix accelerators with small memory systems as a result of the matrix dimensions, memory latency, and pipeline fill/drain delays, with the first two dominated by control and scheduling within the accelerator. Unlike DNN models, where tensor shapes and weight values are known at compilation time and can therefore be optimized and scheduled based on static analysis, numerical computing libraries such as LAPACK and BLAS are often designed to assume runtime-dynamic matrix shapes and sizes. This runtime flexibility requirement entails that the DNN accelerator matrix engine controller needs to be able to make independent matrix compute and memory operation scheduling decisions, without relying on an optimal statically analyzed software schedule.

The Gemmini controller uses hardware-managed double buffering as a latency-hiding technique. As such, the private scratchpad memory and accumulators are split by the controller into two partitions, where the data in one partition gets used for computations while the data in the other partition gets used by the DMA for moving data to and from main memory. This approach can sustain full utilization of the accelerator as long as the number of cycles it takes to perform computation on the data in a partition is longer than the number of cycles it takes to move data from main memory into the second partition. However, this technique is not able to hide latency in cases where the operand matrices are smaller than the size of the accelerator private scratchpad memory. In such cases, there is not enough data to overlap the compute of one partition with the memory movement of the other partition. In those scenarios, the utilization of the accelerator becomes sensitive to the scheduling decisions made by the controller.

In Gemmini's weight-stationary (WS) dataflow configuration, $DIM \times DIM$ data elements of the second operand matrix (the B matrix, also referred to as a "weights" matrix in deep learning workloads) are resident within the systolic array processing units (PEs), while data elements of the first operand matrix (the A matrix, or "activations" matrix in deep learning workloads) stream into the array from the private scratchpad memory. The output data elements (matrix C) propagate through the systolic array and accumulate into a wide accumulator SRAM within the accelerator. As such, when the target matrix operation is of high arithmetic intensity, given this streaming schedule the controller should schedule more memory operations for the A matrix rather than the B matrix in order to minimize latency. However, when the matrix operation is of low arithmetic intensity, there is less data re-use within the accelerator memory hierarchy, and as the roofline model dictates, the operation may be bound by memory bandwidth. Hence, for example, for the case of a tall and narrow B matrix it would be more beneficial to schedule A and B memory operations at a "symmetric" rate, as opposed to the method of choice for high arithmetic intensity operations of an "asymmetric" rate with more A memory operations due to the streaming nature of A in a WS dataflow systolic array.

The accelerator controller must also be able to handle scheduling decisions impacted by dynamic properties of shared resources within the SoC. In our Chipyard SoC evaluation system [1], the main shared resource used by Gemmini is the SoC memory system. The Gemmini DMA can generate many load and store requests to the SoC memory system, and the responses to these requests can return in variable latency and out-of-order. The Gemmini DMA keeps track of these memory transactions and handles their ordering upon their return. Unlike the DMA, the Gemmini execution command queue supports only in-order execution. This is an efficient design point under the assumption of double-buffering and high data re-use within the scratchpad, which together mean that data should be readily available within the private scratchpad when commands are dispatched to the execution command queue. However, when the operand matrices are smaller than the size of the accelerator private memory and do not enable double-buffering, the variable latency of the shared memory system can impact the utilization of the compute array due to front-of-line blocking of the execution command queue.

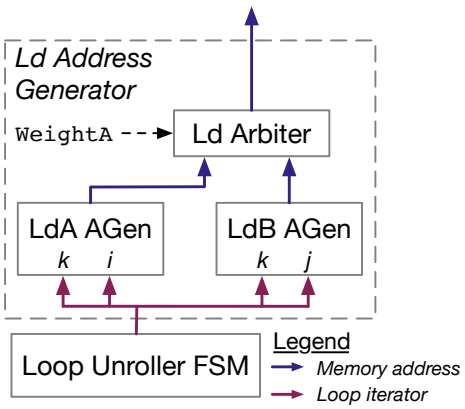

Fig. 4: Arbitration mechanism between A matrix and B matrix loads in Gemmini's load-address generator (which is part of the matrix multiplication FSM illustrated in Figure 3).

Variable DMA transaction tail latency can occur for a variety of reasons, including quality of service (QoS) policies across SoC buses and fabrics, as well as interrupts and other asynchronous events within the system. When the operand matrices are big enough to be double-buffered, this type of tail latency can be hidden. For this reason, an in-order execution command queue is generally a sufficient and efficient choice for DNN accelerators, but may be insufficient for broader classes of workloads.

IV. Matrix Engine Controller Adaptations

We demonstrate how simple and inexpensive improvements within DNN matrix controllers can make them more amenable for use for a broader class of matrix shapes and sizes.

A. Hardware-Managed Static Scheduling

Accelerator controllers with full processor-based software capabilities are flexible enough to enable any combination of static compute and memory scheduling decisions. However, performing scheduling operations using processor-based accelerator controllers comes with software overheads of computing addresses, strides, pointers, bound-checking and control flow, which can often be limited by instruction-issue bandwidth and the throughput of the control processor itself. In contrast, fixed hardware controllers, such as the one implemented in Gemmini, perform address calculations, bound-checking, and control flow, all in parallel to issuing operations, resulting in zero-overhead scheduling decisions. Zero-overhead hardware control can also better utilize feedback from the execution pipeline in order to assist in schedule decisions.

In most cases, such fixed hardware controllers in-fact have sufficient information to perform low-cost hardware-managed static scheduling decisions based on the shapes and sizes of the operand matrices. Specifically, we focus on data load scheduling arbitration within the Gemmini FSM controller. An arbiter, controlled by an arbitration parameter listed as `WeightA`, regulates a weighted arbitration

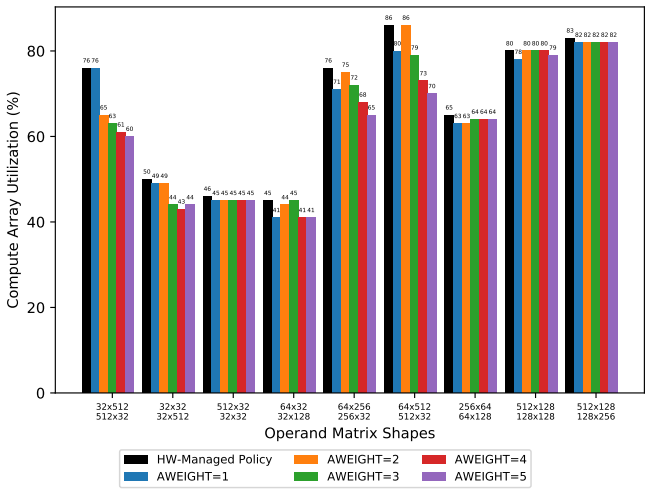

Fig. 5: Gemmini (8×8) utilization using a hardware-managed static scheduling policy in comparison to different hardware controller operand matrix arbitration parameter values

scheme for DMA transactions generated by two address generators for each of the operand matrices in the matrix multiplication: the A matrix address generator (LdA AGen) and the B matrix address generator (LdB AGen). For example, if the value of the `WeightA` parameter is 3, then for every one DMA transaction dispatched to the reservation station by the B matrix address generator, there will be three DMA transactions dispatched to the reservation station by the A matrix address generator. Figure 4 illustrates a more detailed schematic of the load address generator listed in Figure 3, highlighting the `WeightA` arbiter. While this arbitration parameter could be hard-coded in the hardware FSM, which would result in sufficiently high utilization for most large matrices that are double-buffered, it could also be a programmable parameter that is configured in software by the programmer for a particular matrix size and shape. Alternatively, this arbitration decision can also be set by the hardware controller FSM, based on monitoring the loop iterators generated by the FSM for a particular matrix shape. This approach, which we refer to as "hardware-managed static scheduling", due to the fact that a hardware-only control loop sets the arbitration decision based on the software-controlled shape of the matrix, would potentially make the DNN accelerator more robust to handling a diversity of small and rectangular-shaped matrices.

We set a relatively simple hardware-managed static scheduling policy within the Gemmini hardware controller. In this policy, the arbiter starts by issuing a load command for the first block of the second operand matrix (B). This is since the systolic array is a weight-stationary systolic array, in which the second operand matrix is the "static" (stationary) operand within the array. The controller policy will then continue to dispatch load commands based on the values of the k iterators within the address generator for the A matrix and the address generator of the B matrix.

The loop unroller FSM maintain independent copies of the loop iterators for the A address generator and B address generator, allowing each address generator to progress autonomously based on the independent iterator values issued to them by the FSM. As such, when the k iterator associated with the B address generator is greater than the value of the k iterator associated with the A address generator, this is an indication that the DMA transactions being issued are related to the inner most loop in the nested loops. Similarly, when the k iterator of the A address generator is greater than the value of the k iterator of the B address generator, this is an indication of a value increment in the middle loop of the nested loops. Therefore, in this hardware-managed static scheduling policy, the transaction arbiter will issue DMA transactions from the B address generator as long as the value of the k iterator associated with the A address generator is greater than the value of the k iterator of the B address generator.

Figure 5 presents a comparison of the utilization of the 8×8 Gemmini accelerator when using the hardware-managed static scheduling policy vs. using software programmable values of the `WeightA` parameter. Notably, the hardware-managed static scheduling policy demonstrates equal or better utilization compared to the best software programmable value in each of the evaluated cases. More importantly, the hardware-managed adaptive static scheduling policy achieves this utilization without additional programmer intervention or domain knowledge about the shape of the operand matrices. The hardware cost of this adaptive hardware-managed policy is relatively inexpensive, and is primarily reflected in wiring (since the iterator values need to be wired to the arbiter), and a pair of multiplexers and comparators used to implement the adaptive policy decision.

B. Dynamic Scheduling in Matrix Engines

In order to improve dynamic scheduling and alleviate variable-latency head-of-line blocking experienced by small matrix operations in Gemmini due to its integration with shared resources in the SoC, we add out-of-order execution support within Gemmini. Out-of-order execution helps unblock the execution pipeline when processing a long-latency operation by parallel scheduling of additional independent instructions on other available execution units. Execution of the instructions may be out-of-order, but the instructions commit and update the architectural state in-order. This type of ILP-extraction can be very beneficial in superscalar CPUs which have high diversity of instructions with variable latencies. In contrast, Gemmini has a very small instruction set, consisting primarily of two types of operations: fixed-latency execution operations, and variable latency DMA operations. DMA operations are variable latency due to Gemmini's integration with the coherent SoC memory system which includes a cache hierarchy and coherence protocols.

As such, out-of-order execution within Gemmini does not need to encompass the entire pipeline and all instruction types, but rather only those that may experience head-of-line blocking due to a variable-latency instruction and a data dependency. Specifically, we identify two operation types which would benefit from out-of-order execution within the Gemmini controller:

- Compute (matmul) - Reordering of independent or commutative matrix multiplication and accumulation operations, as a result of variable-latency operand load latency
- Store (mvout) - Reordering of DMA transactions from the Gemmini accumulator to main memory as a result of a reordering of compute operations.

Most importantly, unlike CPUs, the Gemmini matrix engine would not benefit from out-of-order execution of memory load commands, since the Gemmini hardware controller dictates a *static* schedule. The static schedule means that there are no dynamic address computations, which means there are no load-after-load dependencies within the instruction stream. Gemmini's decoupled access-execute design further supports this scheme of independent execution orders of memory and compute operations, allowing us to implement out-of-order execution only for the execution and store command queues.

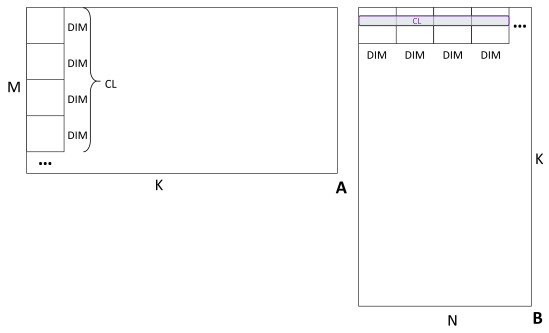

Fig. 6: Matrix operation blocks depending on a single cache line.

However, the out-of-order implementation exposes challenges at the intersection of static and dynamic scheduling within the matrix engines in the context of cache-based SoC memory systems. When the size of a shared cache line is greater than the dimension of the spatial array (and hence, the dimension of compute operations), a static schedule for matrix multiplication should be able to take advantage of spatial locality within the cache line for at least one of the two operand matrices. This advantage of spatial locality can also become a detriment when tail latencies are caused by the shared memory system.

If we assume the granularity of each controller command is a block of $DIM \times DIM$ elements, while a cache line contains CL elements, we can see that if an operand matrix is represented in a row-major layout, a long-latency arrival of data from single cache line could delay the arrival of approximately $\frac{CL}{DIM}$ blocks from that operand matrix. Specifically, if we assume that both operand matrices are represented in a row-major layout, we observe that a long-latency arrival of data from a single cache line would delay the arrival of approximately $\frac{CL}{DIM}$ blocks from the second operand matrix (B), depending on data alignment, since they are all resident in the same cache line (as illustrated in Figure 6). As a result, we see that a long-latency arrival of a single cache line would delay at least $\frac{CL}{DIM} \times \frac{CL}{DIM}$ compute commands, since outer products expect to re-use the same blocks. These blocked operations would consume precious slots within the out-of-order execution reservation station, effectively requiring very large reservation stations in order for out-of-order execution to be effective in hiding long-latency memory accesses through dynamic scheduling.

One solution would be to interleave commands which operate on different cache lines while maintaining the WS dataflow (hence, maintaining the same loop ordering and static schedule), and utilizing as much data locality as possible. We observe that we can take advantage of the commutative nature of accumulation, and the fact that accumulation in matrix multiplication is always performed across the shared dimension (the k dimension), which is the external most loop in our static schedule. We further note that by keeping the static schedule and load operations in their original order, we are able to maintain maximal use of data locality. Therefore, if we take advantage of commutative interleaving across the reduction dimension only within the execution queue, we can maintain both the WS dataflow and maximal data re-use, while providing a different mix of commands within the execution queue issue window. This can be incorporated into the controller in the form of hardware-controlled commutative micro-threading of the execution queue.

This idea is similar to an observation suggested by Shomron & Weiser [19] in the context of SMT processing on systolic arrays, in which they note that the SMT threads could be part of the same DNN

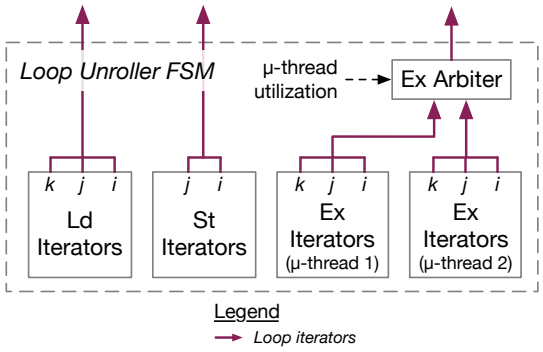

Fig. 7: Commutative micro-threads within the Loop Unroller FSM (illustrated in Figure 3). The "μ-thread utilization" is a feedback channel from the reservation station, and describes the number of instructions from each micro-thread which are stored within the reservation station but which have not yet been issued to the Execution Queue (ExQ).

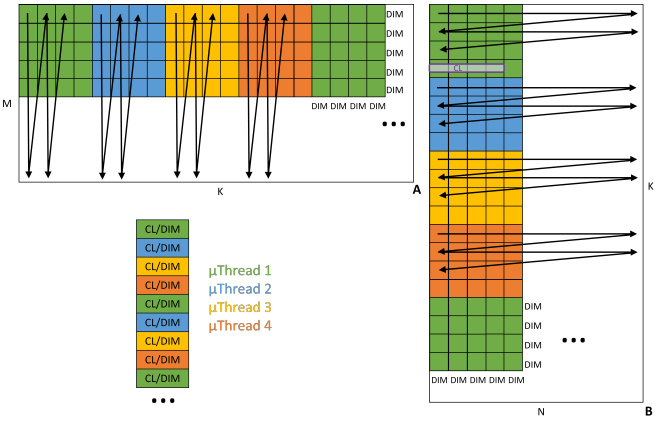

Fig. 8: Gemmini micro-threads fine-grained interleaving. Each thread processes CL/DIM consecutive commands, with thread assignment repeating in a periodic pattern.

execution flow, as opposed to independent threads of independent execution flows. Since our controller manages a single execution flow of matrix multiplication, it is able to split this execution flow into multiple hardware-managed micro-threads in an attempt to hide the latency generated by a sequence of data-dependent commands. Notably, these are not full-fledged threads, since memory load and store operations are still performed according to the original static schedule. Only compute execution commands can be interleaved using these micro-threads, therefore making them both opportunistic and inexpensive in terms of additional required state. The controller generates hardware-managed micro-threads by splitting the nested loops across the most *external* loop-level (the k reduction dimension). The controller maintains the loop iterator indices for each of the micro-threads, and can feed them into the execution address generator, as illustrated in Figure 7. Slots are allocated in the execution queue reservation station only for micro-threads for which the relevant memory load commands have already been issued and which are within a reservation station utilization bound in order to prevent a single thread from starving other threads. Equations 1 and 2 demonstrate the independence of the hardware-managed micro-threads (for the cases of T and 2 threads, respectively) from the perspective of the execution flow within a single controller-managed matrix multiplication instruction.

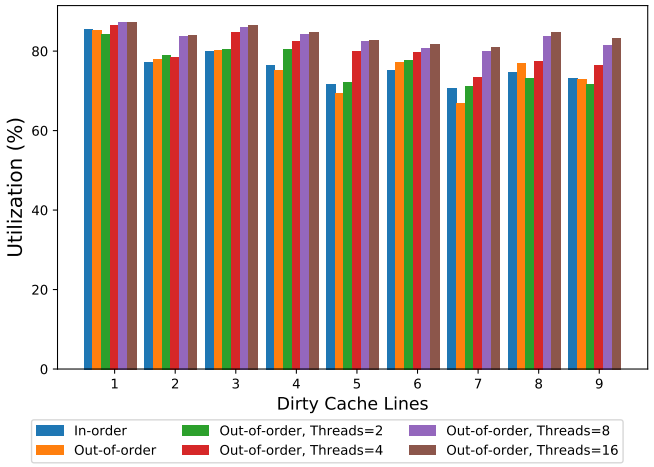

Fig. 9: Average utilization across different numbers of shared L2 dirty cache lines for a 32×1000 times 1000×32 matrix multiplication, comparing fine-grained interleaved commutative micro-threads vs. simple out-of-order and in-order execution in Gemmini (8×8).

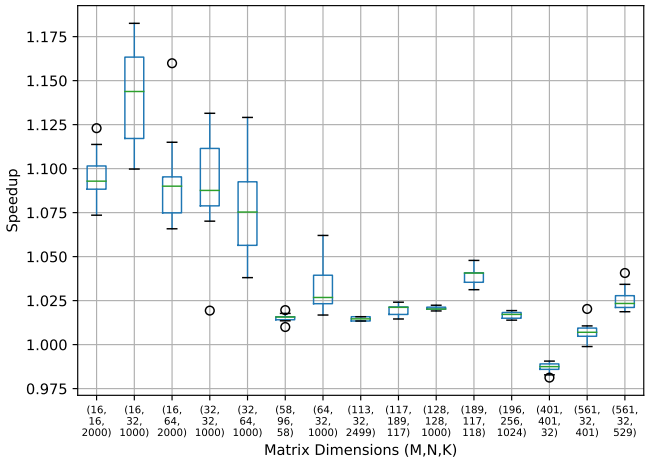

Fig. 10: Speedup distributions (box-plot) of an out-of-order 8×8 Gemmini controller with 8 commutative micro-threads vs. a baseline in-order controller across a collection of matrix shapes with different arithmetic intensities sampled from matrix decomposition workloads.

$$c_{ij} = \sum_{k=0}^{K}\sum_{i=0}^{M}\sum_{j=0}^{N} a_{ik}b_{kj} = \sum_{t=0}^{T}\sum_{k=t\frac{K}{T}}^{(t+1)\frac{K}{T}-1}\sum_{i=0}^{M}\sum_{j=0}^{N} a_{ik}b_{kj} \quad (1)$$

$$= \sum_{k=0}^{(K/2)-1}\sum_{i=0}^{M}\sum_{j=0}^{N} a_{ik}b_{kj} + \sum_{k=K/2}^{K}\sum_{i=0}^{M}\sum_{j=0}^{N} a_{ik}b_{kj} \quad (2)$$

The controller simply manages multiple individual matrix multiplication execution sub-flows that accumulate into the same accumulator SRAM. By performing this hardware threading only for the execution queues rather than the memory queues, the controller does not need to maintain any additional state other than the indices tracking the state of the FSM generating addresses for execution commands on the systolic array. We use a fine-grained micro-thread interleaving scheme, in which the k dimension is partitioned into $(K \cdot DIM)/CL$ partitions, where each partition consists of CL/DIM blocks of size $DIM \times DIM$. Each partition is assigned to a different micro-thread in a periodic pattern, as illustrated in Figure 8. Each thread is responsible for processing CL/DIM consecutive commands before switching to the next partition it is assigned to. In this scheme, each micro-thread is responsible for handling CL/DIM commands, since we know all of those commands will depend on the same cache line, and therefore will not benefit from further internal micro-threading.

We evaluate our micro-threading implementation by comparing the utilization of the series of experiments on a 32×1000 by 1000×32 matrix multiplication, in order to evaluate its benefit for small matrices which cannot be double-buffered by the controller. We use dirty cache lines as a method of inducing variable tail latencies while maintaining complete system integrity (as opposed to isolated trace-driven testing of the accelerator). We vary the number of micro-threads and compare the utilization results to in-order and non-threaded out-of-order execution in Gemmini, as illustrated in Figure 9, and we observe that for micro-thread counts greater than 4 we see consistent benefits in accelerator utilization when using the out-of-order execution together with commutative hardware-managed micro-threading. Eight micro-threads appear to provide the optimal increase in utilization, with sixteen threads exhibiting diminishing returns with respect to the number of threads. Using eight micro-threads, we observe up to a 15% improvement in utilization compared to only in-order execution in Gemmini.

We further evaluate this technique on a wider spectrum of matrix shapes and sizes, derived from the collection of matrix shapes identified in Figure 1. We repeat the series of experiments using dirty cache lines as a method of inducing variable tail latencies, this time expanding our range of dirty cache lines to 1-100. Figure 10 illustrates the speedup distributions observed for each matrix shape across the series of experiments using 8 commutative micro-threads, compared to the baseline in-order configuration. In order to evaluate the cost-effectiveness of this method, we synthesize both configurations using Global Foundries 12nm FinFET process technology. We observe that the total Gemmini area for the baseline in-order configuration is 682,938 $(\mu m)^2$, while the total area for the configuration with our improvements is 685,555 $(\mu m)^2$, demonstrating an area addition of only 0.38%. We therefore conclude that compared to the net speedup of these techniques, ranging between 1%-25%, their area cost is very low, making these an effective choice for matrix engine controllers.

V. Conclusion

In this work, we characterized several key differences in the utilization of matrix engines for DNN inference vs. the broader numerical data analysis workloads category. We observed an increased importance for processing of matrices with a higher variety of shapes and sizes, including small and rectangular matrices. We demonstrated how accelerator utilization can be impacted by static scheduling within the matrix engines controller, as well as system-level effects generating variable memory-latency behavior observed by the accelerator at small matrix size regimes. Finally, we propose the implementation of several micro-architectural techniques in matrix engine controllers within DNN accelerators to better support both static scheduling and dynamic scheduling of operations within the accelerator, requiring only minor modifications to the current Gemmini DNN accelerator micro-architecture. We demonstrate up to a $1.25\times$ improvement in utilization of the Gemmini matrix engine on small matrices through hardware-managed static scheduling, and up to a $1.15\times$ improvement in utilization on small matrices through dynamic scheduling and hardware-managed commutative micro-threading.

REFERENCES

[1] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[2] E. Anderson, Z. Bai, J. J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. H. Bischof, and D. C. Sorensen, "LAPACK: a portable linear algebra library for high-performance computers," in *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*, J. L. Martin, D. V. Pryor, and G. Montry, Eds. IEEE Computer Society, 1990, pp. 2–11. [Online]. Available: https://doi.org/10.1109/SUPERC.1990.129995

[3] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones," in *21st European Symposium on Artificial Neural Networks, ESANN 2013, Bruges, Belgium, April 24-26, 2013*, 2013. [Online]. Available: http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2013-84.pdf

[4] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–284. [Online]. Available: https://doi.org/10.1145/2541940.2541967

[5] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[6] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, "Wide & deep learning for recommender systems," in *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, ser. DLRS 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 7–10. [Online]. Available: https://doi.org/10.1145/2988450.2988454

[7] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, "Accelerating reduction and scan using tensor core units," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 46–57. [Online]. Available: https://doi.org/10.1145/3330345.3331057

[8] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanović, B. Nikolić, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '21. IEEE Press, 2021.

[9] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, May 2008. [Online]. Available: https://doi.org/10.1145/1356052.1356053

[10] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 603–613.

[11] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. Stockholmsmässan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 1919–1928. [Online]. Available: http://proceedings.mlr.press/v80/hashemi18a.html

[12] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. New York, NY, USA: Association for Computing Machinery, 2021.

[13] G. Jeong, E. Qin, A. Samajdar, C. J. Hughes, S. Subramoney, H. Kim, and T. Krishna, "Rasa: Efficient register-aware systolic array matrix engine for cpu," in *Proceedings of the 58th Annual Design Automation Conference, DAC 2021*. ACM, 2021.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[15] D. Richins, D. Doshi, M. Blackmore, A. Thulaseedharan Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long, D. Zimmerman, and V. Janapa Reddi, "Missing the forest for the trees: End-to-end ai application performance in edge data centers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 515–528.

[16] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 6655–6659.

[17] Y. S. Shao, "Design and modeling of specialized architectures," Ph.D. dissertation, Harvard University, 2016.

[18] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "Dnpu: An 8.1tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 240–241.

[19] G. Shomron and U. C. Weiser, "Non-blocking simultaneous multithreading: Embracing the resiliency of deep neural networks," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 2020, pp. 256–269. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00032

[20] F. Sijstermans, "The nvidia deep learning accelerator," in *Hot Chips 30: The Flint Center for the Performing Arts, Cupertino, California, August 19–21, 2018*, 2018. [Online]. Available: https://www.hotchips.org/hc30/2conf/2.08_NVidia_DLA_Nvidia_DLA_HotChips_10Aug18.pdf

[21] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[22] B. Yang, X. Fu, N. D. Sidiropoulos, and M. Hong, "Towards k-means-friendly spaces: Simultaneous deep learning and clustering," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 3861–3870. [Online]. Available: http://proceedings.mlr.press/v70/yang17b.html

[23] K. Yang, Y.-F. Chen, G. Roumpos, C. Colby, and J. Anderson, "High performance monte carlo simulation of ising model on tpu clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356149

[24] S. Zhang, E. Baharlouei, and P. Wu, "High accuracy matrix computations on neural engines: A study of qr factorization and its applications," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 17–28. [Online]. Available: https://doi.org/10.1145/3369583.3392685

[25] S. Zhang, R. Shah, and P. Wu, "Tensorsvm: Accelerating kernel machines with tensor engine," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392770