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× improvement in the utilization of a matrix engine on small matrices through hardware-managed static scheduling, and up to a 1.15× 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 Cholesky, 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.

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
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.

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
Unlike the DMA, the Gemmini execution command queue supports memory transactions and handles their ordering upon their return. The Gemmini DMA keeps track of these system, and the responses to these requests can return in variable latency and out-of-order. The Gemmini DMA can generate many load and store requests to the SoC memory. In our Chipyard SoC evaluation system [1], the main shared memory hierarchy, and as the roofline model dictates, the operation arithmetic intensity, there is less data re-use within the accelerator. Hence, for example, for 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.

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.
are related to the inner most loop in the nested loops. Similarly, when the \( k \) iterator associated with 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 \( 8 \times 8 \) 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.

![Figure 5: Gemmini (8 x 8) utilization using a hardware-managed static scheduling policy in comparison to different hardware controller operand matrix arbitration parameter values](image-url)
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 $\text{DIM} \times \text{DIM}$ elements, while a cache line contains $\text{CL}$ elements, we can see that if an operand matrix is represented in a row-major layout, a long-latency arrival of data from a single cache line could delay the arrival of approximately $\frac{\text{CL}}{\text{DIM}}$ 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 $\frac{\text{CL}}{\text{DIM}}$ 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 $\frac{\text{CL}}{\text{DIM}} \times \frac{\text{CL}}{\text{DIM}}$ 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-thrading 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 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.
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 $(K \times DIM)/CL$ 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 $32 \times 1000$ by $1000 \times 32$ 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× improvement in utilization of the Gemmini matrix engine on small matrices through hardware-managed static scheduling, and up to a 1.15× improvement in utilization on small matrices through dynamic scheduling and hardware-managed commutative micro-threading.