RASA: Efficient Register-Aware Systolic Array Matrix Engine for CPU

Geonhwa Jeong*, Eric Qin*, Ananda Samajdar*, Christopher J. Hughes†,
Sreenivas Subramoney†, Hyesoon Kim* and Tushar Krishna*
*Georgia Institute of Technology, Atlanta, USA
{geonhwa.jeong, ecqin, anandsamajdar}@gatech.edu, hyesoon@cc.gatech.edu, tushar@ece.gatech.edu
†Intel Labs, {christopher.j.hughes, sreenivas.subramoney}@intel.com

***Abstract*—As AI-based applications become pervasive, CPU vendors are starting to incorporate matrix engines within the datapath to boost efficiency. Systolic arrays have been the premier architectural choice as matrix engines in offload accelerators. However, we demonstrate that incorporating them inside CPUs can introduce under-utilization and stalls due to limited register storage to amortize the fill and drain times of the array. To address this, we propose RASA, Register-Aware Systolic Array. We develop techniques to divide an execution stage into several sub-stages and overlap instructions to hide overheads and run them concurrently. RASA-based designs improve performance significantly with negligible area and power overhead.**

***Index Terms*—deep learning, systolic array, CPU, MLP, CNN**

I. Introduction

In diverse areas including, but not limited to, computer vision, natural language processing, and personal recommendation, Deep Learning (DL) models have shown dramatic performance, even exceeding that of humans for some tasks. DL workloads are both notoriously compute hungry and commonplace in applications, motivating enhancements to hardware and software platforms to improve performance and energy efficiency [1]–[3]. General Matrix-Matrix Multiplication (GEMM), a staple of high-performance computing, is also a critical building block for many DL applications including Transformers for natural language processing, Multi-Layer Perceptrons (MLPs) for recommendation models, and Convolutional Neural Networks (CNNs) for computer vision tasks. For DL workloads, GEMM performance and energy-efficiency are sufficiently important that special-purpose hardware support has become common. A systolic array is one of the most efficient structures for dense GEMMs given its simple construction, high concurrency, and ability to efficiently exploit the inherent data reuse in the computation. To that end, most DL accelerators include systolic arrays for GEMM, including Google's TPU, Xilinx's XDNN, and Habana's Goya. Further, even less specialized platforms, such as Nvidia GPUs, also use (small) systolic arrays [4].

In this work, we explore adding a systolic array for GEMMs to a CPU for DL workloads. The decision to go with systolic arrays is motivated by simple construction and small control overhead while simultaneously achieving high throughput compared to other accelerators [1]. While custom DL accelerators' and GPUs' are highly popular for running compute-intensive DL workloads in the datacenter and edge due to their high compute density and memory bandwidth, studies have shown that most DL workloads are actually running on CPUs today [5], given their pervasiveness. CPUs trade off reduced compute density and energy efficiency for improved generality and programmability. To meet the huge demand for running DNNs efficiently, CPU vendors like IBM, ARM, and Intel have begun introducing DNN-specific optimizations. Recently, Intel announced the inclusion of Advanced Matrix Extensions (AMX) [6] to their ISA, to allow software for CPUs to express deep learning GEMMs with a greatly reduced instruction count. Despite this intense commercial activity, to the best of our knowledge, no previous work explores the design trade-offs of integrating systolic arrays in CPUs for GEMMs. Unlike standalone accelerator hardwares, a systolic array housed within the CPU core is subject to constraints of the existing conventional memory system. We identify that the limited size of registers inside CPUs to feed a systolic array can lead to increased under-utilization of the array during its fill and drain times, and even during some of output generation times. To address this, we propose a systolic array based functional unit called *Register-Aware Systolic Array Matrix Engine* (RASA) and develop techniques to divide an execution stage into several sub-stages for properly overlapping instructions.

In summary, the key contributions of our work are as follows. **(i)** *We first introduce an efficient register-aware systolic array based matrix engine for CPU and show how it can be driven by simple GEMM instructions.* **(ii)** *We identify challenges for integrating a systolic array as a matrix engine in a CPU and propose a set of* RASA-*Control and* RASA-*Data optimizations.* RASA-*Control extracts higher performance by introducing novel pipelining and bypassing schemes on the conventional systolic array, whereas* RASA-*Data optimizations include microarchitectural changes in processing elements.* **(iii)** *We implement* RASA *based designs on RTL with Nangate-15nm library and evaluate with various real DL workloads.* Our results show that RASA-Control optimizations reduce runtime by 30.9% with control hardware changes only, and using both RASA-Control and RASA-Data optimizations improves 79.2% in runtime while consuming a total $0.847mm^2$ in area.

This work was supported by an award from Intel. We thank the LIBXSMM team (Alexander Heinecke and Hans Pabst) for their help with our methodology.

II. Background

A. Deep Neural Networks (DNNs) and GEMM

DNNs are comprised of a series of layers, where each layer represents a particular computation. The most common computationally intensive layers are fully connected (FC) and convolutional layers. Multi-Layer Perceptrons (MLP) are composed of a number of FC layers, and are at the heart of many modern DNNs, including recommendation models such as DLRM [7] and natural language processing models like BERT [8]. If we group multiple inputs (i.e., *batch*) through an FC layer, the computation becomes (primarily) a GEMM. Convolutional Neural Networks (CNNs) are dominated by convolutional layers and are extremely popular in image and video processing. Many implementations "lower" the convolution computation to GEMMs [9].

B. SIMD/Matrix Extensions in CPUs for running GEMM

Popular high-performance GEMM implementations heavily leverage Single Instruction Multiple Data (SIMD), especially fused multiply-add (FMA) operations. Due to the performance and efficiency demands of DL workloads, CPU vendors have begun introducing enhancements to their SIMD hardware for GEMMs, including Intel's AVX512_VNNI instructions [10], IBM Power's outer product instructions, and support for smaller data types such as BF16.

Recently, CPU vendors have started introducing support in the ISA for running GEMM. For the purpose of illustration, we use Intel's Advanced Matrix Extensions (AMX) [6] as a reference design, though our analysis and conclusions are more broadly applicable. AMX includes eight 1KB 2D registers (called "tile registers") and instructions to operate on them [6]. For software operating on a 2D array, i.e., a matrix, each tile register may hold an entire small matrix, or, for larger matrices, a sub-matrix (tile or block). Tile register can be moved from/to memory with the *tileload* and *tilestore* instructions, respectively—a tile in memory is a set of up to 16 chunks of data up to 64B each, separated by a fixed stride. Given this ISA support for GEMM, the goal of this work is to study microarchitectural implementations for GEMM engines to natively run GEMMs in CPUs.

C. Baseline Matrix Engine: Systolic Array

Systolic arrays are two-dimensional arrays of processing elements (PEs), connected with peer-to-peer links. They provide high concurrency and high compute to memory ratios due to their regular structure and simple construction. This enables attaining high performance while achieving favorable energy efficiencies. Therefore, several industrial and academic accelerators use them [1], [11], [12].

A GEMM kernel multiplies a $M \times K$ matrix $\boldsymbol{A}$ with a $K \times N$ matrix $\boldsymbol{B}$ to generate the output $M \times N$ matrix $\boldsymbol{C}$. A mapping determines how the operands are fed into and stored in the array and leads to different *dataflows*. As described in a recent work [12], there are three popular classes of dataflows for GEMM, Input Stationary (IS), Weight Stationary (WS), and Output Stationary (OS). Each dataflow maps one

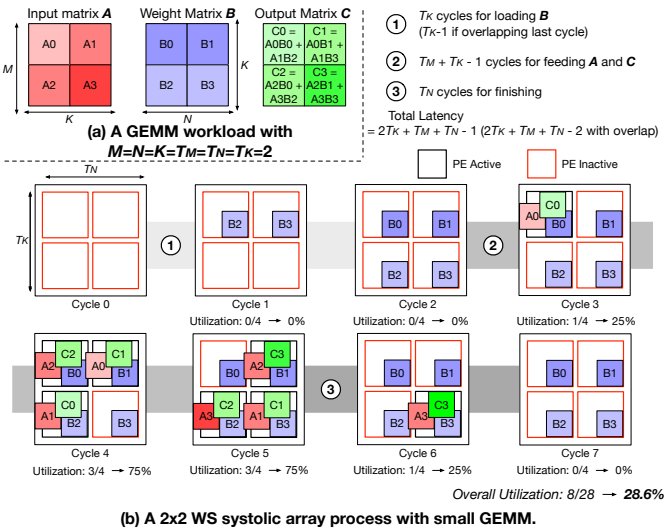

Fig. 1. A 2×2 WS systolic array for processing 2×2 input matrix and weight matrix.

of $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$, to the 2D array of PEs[1], holding a single element of that matrix in place at a PE throughout execution; elements of the other matrices flow through PEs. The choice of dataflow affects performance and energy efficiency, with the best option depending on the dimensions of the operands and the parameters of the systolic array [12]. In modern accelerators, WS is generally preferred since it exploits high spatio-temporal reuse of weights [1]. As mentioned earlier, when mapping into the array the entire computation will often not be mapped at once but will be mapped in portions or "tiles". We use the T_M, T_N and T_K for the dimensions of the tiles when mapped onto the array. Each tile will then be mapped onto the in various passes (which we call a "fold") until all outputs are generated. In a traditional systolic array, each fold requires several steps such as filling the stationary operands, streaming the non-stationary operands, reducing across MAC units, and draining the generated outputs. As shown in Fig. 1, for a tiled GEMM on a WS systolic array with $T_K \times T_N$ PEs, the bottom right PE is in the critical path. It takes T_K cycles to load the stationary weight elements. Then, it takes $T_N - 1$ cycles to fill the pipeline and the first input operand to arrive at this PE, and the next T_M cycles are spent on MAC computations for all the incoming input elements. Finally, additional $T_K - 1$ cycles are needed for reduction and ejection of the last output element. The total latency can thus be calculated as [11], [12]:

$$Latency_{tot} = 2T_K + T_N + T_M - 2 \quad (1)$$

III. Challenges for Systolic Arrays in CPUs

Systolic arrays, given their two-dimensional construction with neighbor-to-neighbor links, are simple to construct, control, and are capable of high throughput and operand reuse when fully utilized. However, simple construction leads to challenges with utilization.

[1]We assume DNN weights are in $\boldsymbol{B}$, but software could place them in A.

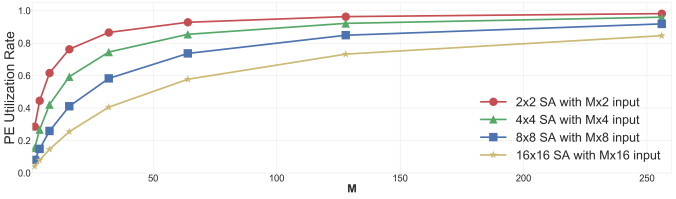

Fig. 2. PE utilization ratio with different input size and SA (Systolic Array) dimensions.

Under-utilization in Systolic Arrays. In Fig. 1, we show a toy example of a 2×2 weight stationary systolic array processing 2×2 input matrices and observe that the average PE utilization is only 28.6%. Though this is a toy example, the challenge of under-utilization in systolic arrays is quite fundamental [12]. There are three cases in which the entire array may not be utilized for all cycles. (i) *Mapping inefficiency*. Due to the mismatch between the workload and the array dimensions, some compute units are left idle. (ii) *Memory Stalls*. This occurs when the memory system cannot supply data fast enough to keep up the throughput. (iii) *Pipeline fill and drain delay*. Recall from subsection II-C, each PE computes for T_M cycles, the rest of the time goes in fill and drain. Thus, even if we have 100% mapping and sufficient memory bandwidth, each MAC is still inactive for the following cycles,

$$Time_{inactive} = Latency_{tot} - T_M \quad (2)$$

Note that all the PEs are idle at the different cycles due to the pipelined movement of operands through the array as shown in Fig. 1. For cases when T_M is comparatively small, this leads to severe performance degradation. In Fig. 1(b), each PE is active for $T_M = 2$ cycles and inactive for the remaining 5 cycles (71% performance degradation).

Combating Under-utilization. The focus of this work is on combating under-utilization due to pipeline fill and drain. Fundamentally, there can be two ways of addressing this. The first is to speed up fill and drain itself via richer interconnects like buses [2] for single-cycle operand broadcast and trees for reduction in logarithmic steps [3]. Unfortunately, this comes at the high area and power overhead [3] which is not acceptable within a CPU since it has tight area and power constraints. The second is to use a large tile size for M. As can be seen from Equation 2, a large T_M can make the inactive time percentage of each PE $(1 - \frac{T_M}{Latency_{tot}})$ converge to 0. The basic idea is once the array is filled up with $\boldsymbol{B}$, we keep streaming elements of $\boldsymbol{A}$ and $\boldsymbol{C}$ through the array, keeping it at peak utilization for a longer time. Fig. 2 quantifies this effect, showing that under-utilization of the array is alleviated with a large T_M. Keeping in mind that the peak throughput of the array is $T_K \times T_N$ operations per cycle, we would like to grow these; Fig. 2 shows that a large T_M helps even as T_K and T_N grow.

Recent DNN accelerators [1] have followed the simple strategy of deploying an array with many PEs, i.e., large T_K and T_N. To achieve good efficiency, they leverage a very large T_M. For example, the first layer of ResNet50 [13], can be transformed to a GEMM with M=47524, N=64, K=147, allowing even a large systolic array to have reasonable efficiency by employing large tile sizes for M.

This cannot be applied on a CPU systolic array, however, since the tile sizes are limited by the size of the tile registers determined by the ISA (e.g., 1KB in Intel AMX). Increasing the size of the tile registers comes with overhead in area and power. This is a challenge, especially for CPUs due to its nature of general-purpose. Area and power devoted to specialized features under the same budget mean lower performance and efficiency for workloads that cannot use them. Another solution is to feed the array directly from memory bypassing the tile registers completely. But this introduces many challenges for a CPU, including that a CPU must be responsive to interrupts and exceptions.

The limitations on a large T_M for CPUs motivate our solution that enables higher utilization despite limitations in register size.

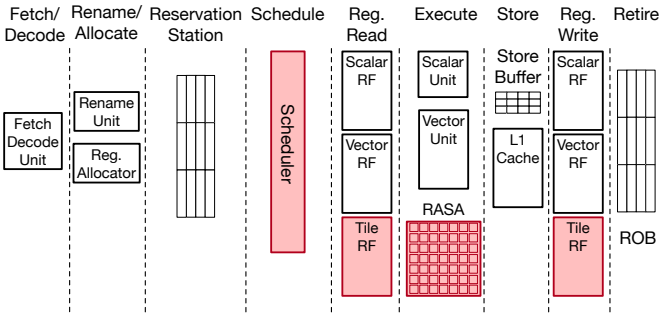

Fig. 3. System overview. Blocks in red include our contributions.

Algorithm 1: A code example with RASA instructions

Input: Tile addresses for 32×32 matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$, ATile0-1, BTile0-1, CTile0-3 for $\boldsymbol{C}$ += $\boldsymbol{A} \times \boldsymbol{B}$ with BF16 and FP32 for input and output data types.

```
// Step 1. Load C Tiles to tile registers
rasa_tl treg0, ptr[CTile0]
rasa_tl treg1, ptr[CTile1]
rasa_tl treg2, ptr[CTile2]
rasa_tl treg3, ptr[CTile3]
// Step 2. Compute partial sums
rasa_tl treg4, ptr[BTile0]
rasa_tl treg6, ptr[ATile0]
rasa_mm treg0, treg6, treg4
rasa_tl treg7, ptr[ATile1]
rasa_mm treg1, treg7, treg4
rasa_tl treg5, ptr[BTile1]
rasa_mm treg2, treg6, treg5
rasa_mm treg3, treg7, treg5
// Step 3. Store C Tiles back to the memory
rasa_ts ptr[CTile0], treg0
rasa_ts ptr[CTile1], treg1
rasa_ts ptr[CTile2], treg2
rasa_ts ptr[CTile3], treg3
```

IV. RASA

A. System Overview

Inspired by Intel AMX, we assume a system with eight architectural tile registers treg0-7, each composed of 16 rows

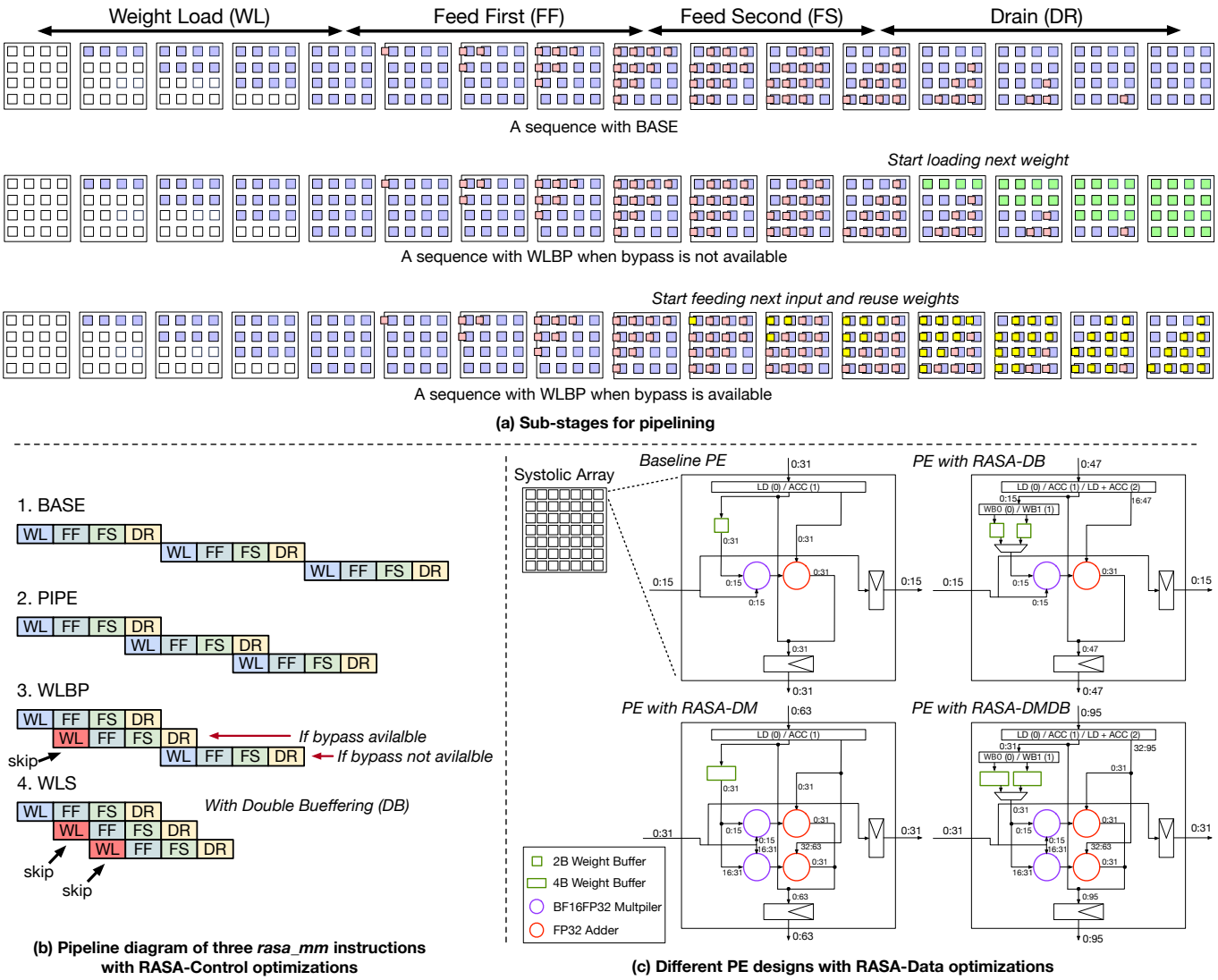

Fig. 4. RASA overview. (a) shows sub-stages used in RASA for pipelining. (b) shows three RASA-Control optimizations. We add one-cycle bubble at the end of FS stage to make all stages have same latency for this example for clear explanation. (c) shows three RASA-Data optimizations.

of 64B. We use *rasa_tl*, *rasa_ts*, *rasa_mm* as the interface with the matrix engine. *rasa_tl* loads data from memory to the specified register while *rasa_ts* stores it back to the memory. A matrix multiplication and accumulation can be done by using the *rasa_mm* instruction on tile registers. A high-level system overview is shown in Fig. 3. A simple example code using RASA instructions is shown in Algorithm 1. For the baseline, we use a WS systolic array with 32×16 PEs and each PE have a single multiply-accumulate (MAC) unit. Fig. 1 shows how a WS systolic array works. All of the operands are read/written from/to tile registers. The hardware first reads $\boldsymbol{B}$ a row at a time from the bottom up, inserting data into the north port of the array. Next, it inserts values from $\boldsymbol{A}$ and $\boldsymbol{C}$ into the west and north ports of the systolic array, respectively, in a skewed manner. Each cycle, every PE forwards the current $\boldsymbol{A}$ element to the east and the previous cycle's result, a partial sum, to the south. The PEs at the bottom of the systolic array produce the final outputs which are written back to the output register.

B. Design

Fig. 4(a) illustrates which inputs and weights are mapped on PEs in each time step when executing two *rasa_mm* instructions in a 4×4 systolic array. PEs become purple when the weights of the first *rasa_mm* instruction is loaded while they change to green when the weights for the second instruction are loaded. Red and yellow boxes correspond to the inputs of the first and second instructions respectively. We first divide the execution stage of the WS dataflow on a systolic array into four sub-stages and explain how to overlap sub-stages, including which resources should be added.

① Weight Load (WL): WS dataflow needs weight values ($\boldsymbol{B}$) to be loaded in PEs. Without dedicated links for this, it takes T_K cycles from the top edge of the systolic array to the bottom edge. (The last cycle of WL can be overlapped with the first cycle of FF. In that case, WL effectively takes $T_K - 1$ cycles.). ② Feed First (FF): During this stage, $\boldsymbol{A}$

and C elements are fed into the systolic array from the left and top. FF ends when we finish feeding the first row of the systolic array. We split the Feed stage here for pipelining. This stage takes T_M cycles. ③ Feed Second (FS): In FS stage, we finish feeding inputs for the remaining rows of the array. Note that some PEs are now idle, and we will leverage those to start executing the next instruction. This stage takes $T_K - 1$ cycles. ④ Drain (DR): In this stage, we let the data in the systolic array finish propagating east and south. Draining the remaining outputs takes T_N cycles. Without any optimizations or pipelining, *rasa_mm* instructions are completely serialized (as shown in Fig. 4(b) BASE), and throughput will be one instruction every $Latency_{tot}$ cycles (Eq. 1). We call this as the BASE design.

RASA-Control Optimizations. We propose three control optimizations, called RASA-control, for pipelining multiple *rasa_mm* instructions within a systolic array concurrently as shown in Fig. 4(a) and (b).

(i) *Basic Pipelining* (PIPE): We observe that the DR stage of previous *rasa_mm* instruction can be overlapped with the WL stage of the next *rasa_mm* instruction. This does not require new hardware in the PEs.

(ii) *Weight Load Bypass* (WLBP): We observe that sometimes consecutive *rasa_mm* instructions reuse the B (weight) register. For example, in Algorithm 1, lines 9 and 11 reuse treg4 and lines 13 and 14 reuse treg5 as the weight register. If we identify this, we can skip the WL stage in RASA unless the content in the reused register has been changed in the meantime. We add a dirty bit to each tile register to track whether it has been changed after the previous *rasa_mm*. With WLBP, if B is reused and has a clear dirty bit, we can start the FF stage during the DR stage of the previous instruction. We further observe that the FS stage does not fully utilize the left-top PEs, which are needed by the FF stage of the next instruction; thus, when we skip WL, we also allow these stages to be overlapped.

(iii) *Weight Load Skip* (WLS): WLBP aggressively pipelines the systolic array, but only when weights are reused. To achieve this same throughput in other cases, we can do WL for an instruction during the previous instruction's FF; however, this requires extra buffers and links, which will be discussed later. Thus, WLS hides WL latency by prefetching.

RASA-Data Optimization. In Fig. 4(c), we show the baseline PE and the PE designs with RASA-Data optimizations. The PEs perform mixed-precision matrix multiplication (BF16 in, FP32 out). The baseline PE design has one multiplier, one adder, and buffers a single weight. RASA-DB uses PEs with Double Buffering (DB). These use an extra weight buffer and links to enable one of the aforementioned RASA-Control optimizations, RASA-WLS. We also propose RASA-DM, a PE design with a Double Multiplier (DM), and an extra adder. This PE updates two partial sums in parallel; thus, we place a row of adders at the bottom of the systolic array to merge the two partial sums. DM doubles the size of an individual PE, but we compensate by reducing half of the total number of PEs in the DM systolic array implementation. Finally, RASA-DMDB includes both DB and DM features. We show in Section V that the RASA-Data optimizations cost negligible area overhead over the baseline.

V. Evaluation

To evaluate the proposed design and optimizations, we use Intel AMX instructions as the interface with the proposed design. We use optimized convolutions and MLPs from the LIBXSMM library [9], using AVX and AMX. We collected traces with Intel SDE. We implement the RASA optimizations in MacSim [14], a trace-driven cycle-level simulator with the following configuration: CPU (and NoC) at 2GHz, 16 pipeline stages, ROB size of 97, fetch/issue/retire width of 4, similar to Intel's Skylake. To focus on the systolic array tradeoffs, we assume that the core is not stalled by memory.

Workloads. We choose three workloads from MLPerf to represent different tasks, ResNet50 [13] for computer vision, DLRM [7] for recommendation, and BERT [8] for natural language processing. We choose three layers from each workload. Table I summarizes the dimensions of the convolution layers in ResNet50 and FC layers in DLRM and BERT that we use. We use the following notation: N for batch size, K for the number of filters, C for the number of input channels, X and Y for input dimensions, R and S for filter dimensions, NIN for the number of input neurons, NON for the number of output neurons. We perform experiments on inference due to the extremely long simulation time for training; however, our proposed concept is not limited to inference since GEMM is also a key building block for training [9].

TABLE I
Layer Dimensions Used in Evaluation

Layer	Dimensions
ResNet50-1	N=32 K=C=64 X=Y=56 R=S=1
ResNet50-2	N=32 K=C=64 X=Y=56 R=S=3
ResNet50-3	N=32 K=512 C=1024 X=Y=14 R=S=1
DLRM-1	N=512 NIN=1024 NON=1024
DLRM-2	N=512 NIN=1024 NON=64
DLRM-3	N=512 NIN=2048 NON=2048
BERT-1	N=256 NIN=768 NON=768
BERT-2	N=256 NIN=3072 NON=768
BERT-3	N=256 NIN=768 NON=3072

We evaluate the baseline design (no pipelining with baseline PEs) and seven RASA-based designs, whose names indicate the applied optimizations. For example, RASA-DM-PIPE uses DM and PIPE. For fair comparisons, we use the same number of multipliers in all systolic arrays. We use a 32×16 array of PEs (16×16 if DM is applied) to match the tile register dimensions. We run all systolic arrays at 500 MHz.

Area Overhead and Energy-Efficiency. We implemented and synthesized the RASA-data optimizations with the Synopsis DC compiler on Nangate 15nm. Then, we used Cadence Innovus for place-and-route to understand area and power costs. The area of the baseline systolic array (32×16 baseline PEs) is 0.7% of the total die size of an Intel Skylake GT2 4C CPU. RASA-DB, RASA-DM, and RASA-DMDB have 3.1%, 2.6%, and 5.5% area overhead over the baseline systolic

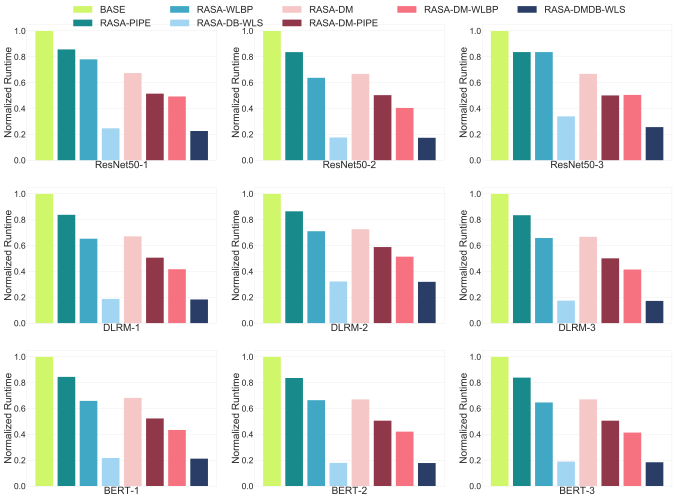

Fig. 5. Runtime of different RASA optimizations normalized to runtime of baseline. The relative performances of various configurations are independent of workloads.

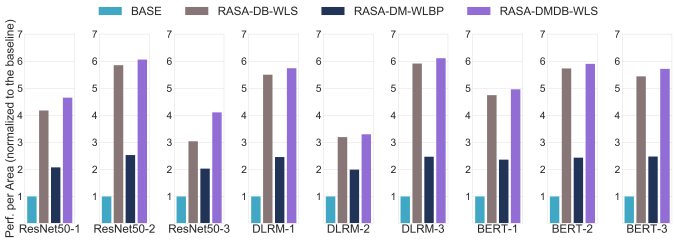

Fig. 6. Performance Per Area (PPA) for different RASA-Data optimizations.

array, respectively. RASA-DB, RASA-DM, and RASA-DM-DB achieve average energy efficiency improvements vs. the baseline of 4.38×, 2.19×, and 4.59×.

Runtime and PPA. Fig. 5 compares the runtime of each design normalized to the baseline. RASA-PIPE and RASA-WLBP reduce runtime by an average of 15.7% and 30.9%, respectively. These designs only require control hardware changes (and eight dirty bits for tile registers with RASA-WLBP) over the baseline. RASA-DB-WLS adds buffers to enable the most aggressive pipelining, achieving a 78.1% average reduction in runtime. The remaining designs use DM to merge pairs of PEs. RASA-DM-WLBP gives a 55.5% average improvement in runtime (24.6% benefit over RASA-WLBP). Combining DM and DB, RASA-DMDB-WLS gives similar performance to RASA-DB-WLS, a 79.2% average runtime reduction.

In Fig. 6, we compare three different RASA-Data optimizations: RASA-DB, RASA-DM, and RASA-DMDB in terms of performance per area (normalized to the baseline). We apply the best-performing RASA-Control optimization to each RASA-Data optimization, i.e., RASA-DB-WLS, RASA-DM-WLBP, RASA-DMDB-WLS. Since the area overhead of RASA-Data optimizations are small, performance per area shows the similar trend with runtime.

Sensitivity on Batch Size. In Fig. 7, we explore how RASA works with different batch sizes. In this graph, we use RASA-DMDB-WLS to understand the largest performance gain that can be achieved with RASA optimizations for different batch sizes. First, we observe that the FC layers with very small batches (1, 2, 4, 8, and 16), have very similar normalized runtimes due to the systolic array being 16×16 (or 16×32). Thus, these runs all use the same number of *rasa_mm* instructions since 16 is the smallest granularity of work. Second, as batch size increases, runtime approaches an asymptote. For large batches, *rasa_mm* instructions dominate the workload. $L_{baseline} = 95$ cycles for the configuration in our evaluation (Eq. 1). If we perfectly pipeline all *rasa_mm*, we complete a *rasa_mm* every 16 cycles. Thus, RASA-DMDB-WLS can at best bring the normalized runtime down to $\frac{16}{95} = 0.168$.

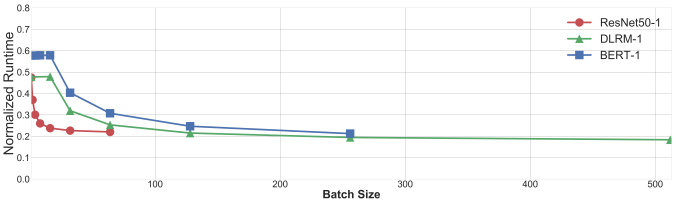

Fig. 7. Normalized runtime of RASA-DMDB-WLS with different batch sizes.

VI. Conclusions

This work is the first to study the implications of integrating a matrix engine inside the CPU pipeline. Specifically, we identify challenges with under-utilization of a systolic array when used as a matrix engine due to frequent fills and drains because of limited sized registers inside CPUs. We propose an efficient pipelined register-aware systolic array called RASA that provides 79.2% runtime improvement and 4.59× energy efficiency over a baseline systolic array on various DL workloads with negligible area overhead.

References

[1] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.
[2] Y.-H. Chen *et al.*, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.
[3] H. Kwon *et al.*, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *ASPLOS*, 2018.
[4] "Nvidia tesla v100 gpu architecture." 2017, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.
[5] C.-J. Wu *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *HPCA*, 2019.
[6] Intel, "Intel architecture instruction set extensions programming reference," 2020.
[7] M. Naumov *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv*, 2019.
[8] J. Devlin *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv [cs.CL]*, 2019.
[9] E. Georganas *et al.*, "Harnessing deep learning via a single building block," in *IPDPS*, 2020.
[10] A. Rodrigues *et al.*, "Lower numerical precision deep learning inference and training," *Whitepaper*, 2018.
[11] B. Asgari *et al.*, "Meissa: Multiplying matrices efficiently in a scalable systolic architecture," in *ICCD*, 2020.
[12] A. Samajdar *et al.*, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," in *ISPASS*, 2020.
[13] K. He *et al.*, "Deep residual learning for image rec." *CVPR-2016*.
[14] H. Kim *et al.*, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," 2012.