MEISSA: Multiplying Matrices Efficiently in a Scalable Systolic Architecture

Bahar Asgari, Ramyad Hadidi, and Hyesoon Kim
Georgia Institute of Technology
{bahar.asgari, rhadidi, hyesoon.kim}@gatech.edu

***Abstract*—The fundamental building block of many algorithms such as data analytics and neural networks is matrix multiplication. Besides its popularity, matrix multiplication is one of the rare algebraic computations that demand high data reuse rate. During the past decades, systolic arrays have been proposed as a low-cost solution for implementing high data reuse, and they have seen a resurgence of interest recently. Particularly, two categories of systolic arrays have been proposed, both of which are made of connected multiply-and-accumulate (MAC) units: non-stationary and stationary architectures. While in the non-stationary architecture both operands of the matrix multiplication flow through the MAC units, in the stationary architecture, only one of them flows. Regardless of their advantages, their common challenges are that they have high latency and are not scalable. In other words, latency increases linearly when the input size grows. Particularly, these are crucial challenges for applications of large matrix multiplication (e.g., deep neural networks (DNNs)) in the edge, in which *latency* must be optimized not throughput. To resolve this challenge, we propose multiplying matrices efficiently in a scalable systolic architecture (Meissa). Meissa is a novel stationary systolic array that, unlike prior work, separates multipliers from the adders rather than combining them in a unified array of MACs. Such an interconnection enables Meissa to sustain a sublinear growing rate in latency with scaling problem size. Our experimental results on a ZYNQ XC7Z020 FPGA show that Meissa executes the single-batch inference of DNNs 1.99× and 1.83× as fast as the prior non-stationary and stationary systolic arrays, respectively.**

I. Introduction

Matrix multiplication is a fundamental computation in linear algebra, with numerous applications in mathematics, statistics, physics, economics, computer science, and artificial intelligence [1]. More recently, the advancements of computer vision and deep neural networks (DNN) [2] have increased the demand of matrix multiplication, particularly on large operands. The key computations of DNNs are convolution and fully-connected layers [3]–[6] that can be implemented as matrix multiplication [7], [8]. The increasing demand for matrix multiplication and the need to execute it quickly have motivated several proposals that go beyond software optimizations to design specialized hardware, among which *systolic arrays* have been a successful attempt [9]–[15]. The key reason for such a success is the unique requirement of data reuse in matrix multiplication, which is efficiently satisfied by the unique structure of systolic arrays.

Since 1979 [16], several studies have explored different implementations of systolic arrays for various applications. Many of these studies and industry products target matrix multiplication [9]–[15], [17], [18] to accelerate machine learning algorithms such as computer vision using neural networks. Regardless of the differences in systolic-based matrix multiplications, their skeleton can be categorized into two groups: non-stationary and stationary systolic arrays. Both groups share the principle of *flowing data* through an array of computation logic. Unlike non-stationary systolic arrays, in which both inputs flow, in stationary systolic arrays, one of the inputs stays in the array during the execution time. The stationary approach is most efficient for architectures with high-capacity memory elements [19].

The two mentioned categories of systolic arrays are beneficial for high *throughput* [13], [20]–[22], which is crucial in data centers where multiple inputs are available at once and processed together. However, for edge implementation of matrix multiplication, *latency* is more important than throughput [23] for two reasons. The first reason is strong real-time constrains of in-the-edge systems. In many edge applications such as computer vision in an autonomous drone (e.g., autel x-star [24], DJI Mavic Air 2 [25] with a 240fps camera) order of milliseconds improvements in single-batch inference matters. Similarly, a self-driving car must detect objects quickly and act promptly to prevent accidents. In such cases, the latency of prior throughput-oriented systolic arrays is not sufficient. The second reason and a motivation for not optimizing throughput in edge is that unlike in data centers, only one input is available at a time. Thus, we must process individual inputs as soon as they arrive and aspire to reduce single-batch latency.

In aforementioned systolic arrays, latency grows linearly with the size of inputs. Therefore, although for small problems slower execution of the prior systolic arrays could be negligible, it creates a crucial performance bottleneck for larger problem size. Given the growing size of the on-demand applications of matrix multiplication (e.g., the size of DNNs), the size of a problem (i.e., the size of matrices) must not negatively impact latency. Additionally, *scalability* is more challenging in the edge applications, in which the resources to achieve desired performance are limited. The mentioned challenges motivated us to optimize *latency*, which is inherent to the dataflow architecture, whereas throughput that can be improved artificially by adding more hardware units.

To utilize a given hardware budget to achieve lower latency, our main observation is the following: since matrix multiplication consists of multiplications and additions, and as additions can be done in the order of $log(n)$ rather than $O(n)$ (for n numbers), the overall latency *can* increase sublinearly with the input size and thus the linear growth of latency in prior systolic arrays is not optimal. To improve latency, we propose multiplying matrices efficiently in a scalable systolic architecture (Meissa[1]). The key insight of Meissa is to use an array of multipliers interconnected separately from the adders, connected in a tree topology, which differs from prior systolic architectures consisting of arrays of multiply-and-accumulate (MAC). In fact, the insight of Meissa is neither just using the well-established adder trees nor solely its systolic architecture. It is indeed optimizing single-batch inference latency, by integrating adder trees in a systolic dataflow architecture. In summary, Meissa makes the following key contributions:

- It is the first *scalable* systolic-based matrix multiplier.
- It consists of a multiplier array connected to adder trees for multiplying matrices quickly and energy efficiently.
- It uses a new interconnection and mechanism of flowing data to reduce transferred data within the array.
- It streams the operands through the systolic array, as they are in their original shapes; hence, unlike prior work, it prevents additional steps and resources for preprocessing.

We implement Meissa on a ZYNQ XC7Z020 FPGA, using a high-level synthesis (HLS) tool as a solution to prototype the harmonic structures of systolic arrays. Our results show that Meissa executes DNNs 1.99× and 1.83× as fast as the prior non-stationary and stationary systolic arrays, respectively.

[1]Meissa is a multiple star in the constellation of Orion.

II. Systolic Architectures & Prior Work

Since 1979 [16] various architectures have been introduced for systolic architectures [26], [27]. More recently, the advantages of artificial intelligence and the need for massive parallel matrix multiplication have motivated academia and industry to rethink the systolic arrays [9]–[13], [15], [17], [18], [28]. Systolic arrays for matrix multiplication have also been implemented by industry [13] in large data-center scales. Regardless of the different implementations, the systolic-based matrix multipliers used in prior studies can be categorized as non-stationary and stationary, based on the way the operands of the matrix multiplication are being handled during execution. In the following, we explore both categories for performing:

$$\mathbf{A}_{n\times m} \times \mathbf{B}_{m\times p} = \mathbf{C}_{n\times p}. \tag{1}$$

Non-stationary Systolic Array (NSA): The processing elements (PEs) of this systolic architecture (e.g., [26], [27], [29]) are multiply-and-accumulate (MAC) units. As its name indicates, none of the inputs stay in the PEs during the execution, and they pass through the PEs in two different directions. Upon the arrival of new inputs, each PE multiplies the two inputs coming from its neighbors and adds them to the prior accumulated results. At the end of the execution, each PE contains one element of the output matrix. Thus, the appropriate size of the systolic array for implementing an NSA for multiplying $\mathbf{A}_{n\times m}$ and $\mathbf{B}_{m\times p}$ is $n \times p$ (i.e., the size of output matrix).

To guarantee the correctness of computations, the inputs must arrive at each PE at the right time. To do this, the two inputs are inserted into the array as shown in Figure 1a (a simple example is presented in Figure 2 Section III). Since both inputs are non-stationary, the multiplication starts as soon as the first elements of the inputs arrive at a PE. Therefore, no additional time needs to be spent on loading. To finish the multiplication, all elements of both inputs must pass through the PEs completely. If we define a *time step* as the time for an input element to pass through a PE, the number of time steps for multiplication using the NSA is

$$T^{nsa}_{process} = n + m + p - 2, \tag{2}$$

which is equal to the number of time steps to move a window of size $n\times p$ over either one of the inputs of size $n\times(n+m-1)$ or $(m + p - 1) \times p$, horizontally in $\mathbf{A}$ and vertically in $\mathbf{B}$.

Once the multiplication is done, the outputs generated in the PEs must be sent out. To do so, if the PEs use the same output ports used for passing through $\mathbf{B}$, the number of time steps for offloading the outputs from the systolic array is

$$T^{nsa}_{out} = n + p - 1. \tag{3}$$

$p-1$ steps of reading the ready elements of the output is overlapped with the multiplication steps. Therefore, as Figure 1a illustrates, the total time steps for multiplying $\mathbf{A}_{n\times m} \times \mathbf{B}_{m\times p}$ using the NSAs is calculated as:

$$T^{nsa}_{total} = 2n + m + p - 2. \tag{4}$$

TPU-style Stationary Systolic Array (TSSA): A more popular type of systolic array for matrix multiplication is TSSA, which is the architecture of the systolic array in TPU [13]. TSSA is also called weight stationary [30] or static systolic arrays [31] and has been implemented for neural networks. The PEs of a TSSA are MAC units, too. However, unlike NSAs, the PEs keep one of the inputs in their registers and instead pass through their outputs (see Figure 1b). As a result, before starting the multiplications, one of the inputs (e.g., $\mathbf{B}_{m\times p}$) must be loaded to the registers of each PE. Besides, the appropriate size of the systolic array for implementing a TSSA, in which $\mathbf{B}_{m\times p}$ stays in the PEs, would be $m \times p$ (i.e., the size of the matrix $\mathbf{B}$). As Figure 1b shows, the number of time steps to load $\mathbf{B}$ into the systolic array is

$$T^{tssa}_{load} = m. \tag{5}$$

Moreover, the number of steps for multiplying the matrices is

$$T^{tssa}_{process} = n + m + p - 2, \tag{6}$$

which is the number of steps to move matrix $\mathbf{A}$ through the systolic array horizontally. Similar to the NSA, all elements of the output must be carried out even though they are being created and passed through the PEs. Since the size of the output matrix is $n \times p$, the number of steps for offloading the outputs from the systolic array is

$$T^{tssa}_{out} = n + p - 1. \tag{7}$$

However, only one step of the offloading and the multiplication is non-overlapping. Additionally, we can start the multiplication at the last step of loading. Thus, as Figure 1b shows, the number of steps for $\mathbf{A}_{n\times m} \times \mathbf{B}_{m\times p}$ with TSSA is

$$T^{tssa}_{total} = n + 2m + p - 2. \tag{8}$$

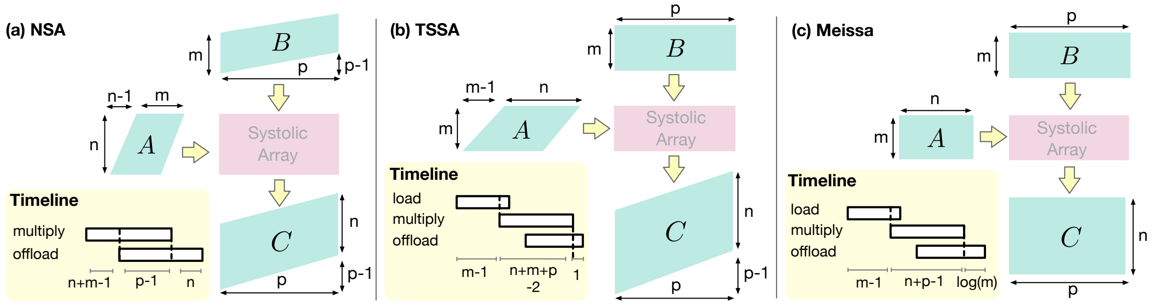


Figure 1. **Systolic Architectures for Multiplication:** Comparing the overall scheme and the timeline of two popular systolic arrays and Meissa: **(a)** NSA, **(b)** the TSSA, and **(c)** Meissa, our proposed scheme, introduced in Section III.

Key Challenge: Besides the success of NSA and TSSA in providing massive parallelism for matrix multiplications, they are not fast enough (in terms of latency) and, more importantly, they both suffer from the common challenge of *scalability*. To be specific, as Equations 4 and 8 illustrate, the total time for multiplying two matrices directly depends on (i.e., linearly changes with) the dimensions of the input sizes. The main reason for this stems from the *interconnection* of the PEs and the use of *MAC*s. The similar connectivity in both prior systolic arrays forces the reduction operation (i.e., adding the results of multiplication) to be done as one operation per cycle, which is not the fastest possible implementation. Scalability is particularly important in DNNs, the sizes of which have been growing. Besides the matrix multiplication itself, a key operation in DNNs is convolution. To efficiently use systolic arrays for performing convolution, converting convolution to matrix multiplication is a common practice [7], [8]. Convolving K filters of size $F \times F \times C$ on an input size of $W \times H \times C$ results in an output of size $W \times H \times K$ (for simplicity, we assumed the same padding):

$$\mathbf{O}_{K\times WH} = \mathbf{W}_{K\times FFC} \times \mathbf{I}_{FFC\times WH}. \tag{9}$$

III. Meissa

Key Insight: This paper proposes Meissa, a *scalable* systolic architecture for matrix multiplication. To provide scalability, Meissa multiplies matrices such that the total processing time has a sublinear relation with at least one dimension of the input matrices. To do so, the key insight of Meissa is to separate the multipliers from the adders and connect the adders in a *tree* topology. As a result, the time to add the results of multiplication is faster than that provided by the ordinary MAC-based systolic arrays. Meissa is particularly beneficial for the computation in DNNs because, while the total processing time is linearly dependent on the *non-growing* and smaller dimensions of the input matrices (i.e., W and H in Equation 9), it sublinearly depends on their *growing* and larger dimensions (i.e., FFC in Equation 9). After introducing the microarchitecture of Meissa, we analyze time to load, process, and offload, and use a simple example to clarify the mechanism of Meissa and other systolic arrays.

Processing Mechanism: Similar to the TSSA, Meissa includes three phases of processing: load, process, and offload. The load phase is similar to TSSA, as shown in Figure 1c. Before the computation starts, matrix $\mathbf{B}_{m\times p}$ is inserted into the systolic array. Therefore, the number of steps to load depends on the dimension of $\mathbf{B}$, or

$$T_{load}^{meissa} = m. \tag{10}$$

Once matrix $\mathbf{B}$ is loaded, multiplying $\mathbf{A} \times \mathbf{B}$ starts by passing $\mathbf{A}$ through the PEs. Later, the output of the multiplication will be added in adder trees rather than in the MAC units, for which matrix $\mathbf{A}$ is not inserted diagonally (unlike the TSSA). Since $\mathbf{A}$ is passing matrix $\mathbf{B}$ through their common dimension (i.e., m), the number of steps to process the inputs depends on the other two dimensions and is calculated as

$$T_{process}^{meissa} = n + p - 1. \tag{11}$$

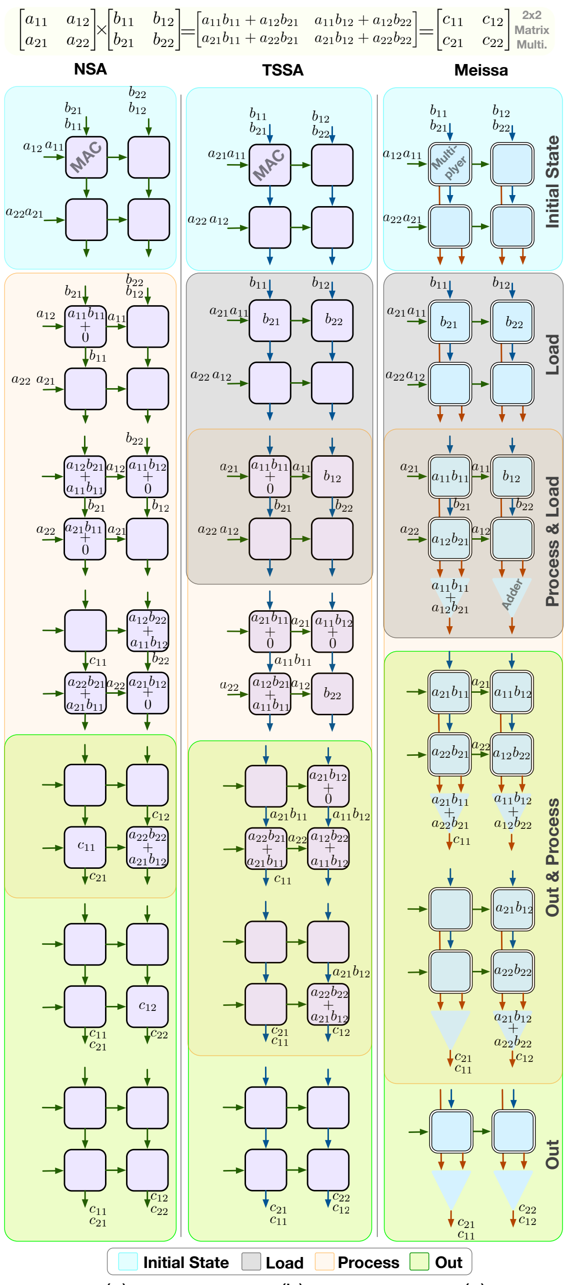
To simplify the comparison, we assume that, for Meissa, the *process* phase includes only the multiplication, and the add operations are calculated in the offloading phase. Since the adders are connected in a balanced tree topology, the time to read the output matrix from the systolic array is

$$T_{out}^{meissa} = n + log(m) + p - 1. \tag{12}$$

Similar to the other systolic arrays, these phases (i.e., load, process, and offload) can be overlapped. The overlap between the load and process phases is one step. Between the process and offload phases, all the steps are overlapped except for the last $log(m)$ steps for draining the adder tree (see Figure 1c). As a result, the total number of steps for $\mathbf{A}_{n\times m} \times \mathbf{B}_{m\times p}$ when using Meissa systolic arrays is

$$T_{total}^{meissa} = n + m + log(m) + p - 2. \tag{13}$$

Figure 2 provides an example to clarify Equations 4, 8, and 13. Here, since $n = m = p = 2$, the size of the MAC arrays, in NSA (Figure 2a), TSSA (Figure 2b), and the multiplier array of Meissa (Figure 2c), are all the same as 2×2. As Figure 2a shows, at each step, each MAC unit of the NSA multiplies its inputs together and accumulates the result to its stored value. The TSSA (Figure 2b), on the other hand, first loads $\mathbf{B}$ and then does a similar process as NSA. However, each MAC of TSSA passes its outcome to one of its neighbors (i.e., downstream). Therefore, in this example, both NSA and TSSA take the same time steps. As Figure 2c shows, after loading $\mathbf{B}$, Meissa starts inserting $\mathbf{A}$, and the adder trees start producing the output matrix as soon as their inputs arrive.



(a) (b) (c)

Figure 2. **An example of process:** The time steps required for multiplying two 2×2 matrices using **(a)** NSA, **(b)** TSSA, and **(c)** Meissa. The time steps include load (if any), process, and out (i.e., offload).

IV. Microarchitecture

Figure 3 shows PEs and the connections between the PEs for Meissa, and the two MAC-based systolic arrays, NSA and TSSA. The PEs of all three designs include two input registers (i.e., R1 and R2), and an output register (i.e., R3). Their differences are in where the adders are located, what data pass through, and in which direction data flows. Figure 3a illustrates the microarchitecture of Meissa consisting of an array of multipliers each row of which is connected to an adder tree. In Meissa (Figure 3a) and TSSA (Figure 3b right) each multiplier has one stationary (R2) and one streaming

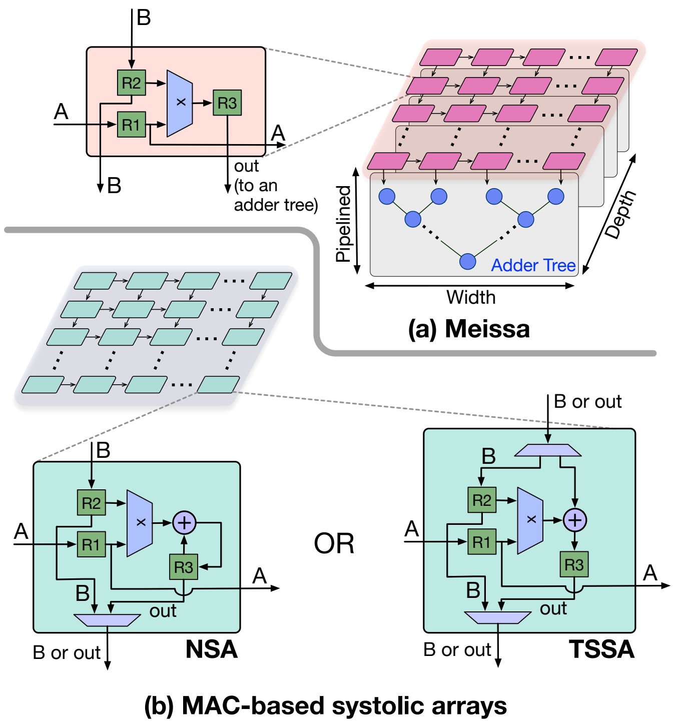


Figure 3. **The Microarchitecture of Systolic Arrays: (a)** Meissa, made of an array of multipliers, connected to adder trees, and **(b)** the popular MAC-based schemes, which are the common microarchitecture in NSA and TSSA.

(R1) input. Therefore, during the process phase, an operand stays in R2s and the other operand passes through the R1s, whereas in NSA (Figure 3b left) both inputs are streaming. Both TSSA and NSA accumulate the partial sums in their PEs. Their difference is that in TSSA, each PE adds the output of MAC to the partial results coming from the upstream, whereas in NSA, each PE adds its output of MAC to its own previously accumulated partial results. In NSA, the up-down stream is used for either streaming **B** or for offloading the final results. In TSSA, the up-down stream is used to either load the stationary operand (**B**), or to pass the partial results for accumulation.

At each time step, all multipliers of Meissa are active and process their inputs. R1s with streaming data are connected in a row within the array such that at each cycle their contents shift one column to right. To reduce the connections, only the first row/column of Meissa and TSSA is connected to the memory. Moreover, to further reduce the connections, each PE of the first column of Meissa and TSSA can only connect to one data stream line so that operands **A** and **B** use a shared link and based on the phase (i.e., load or process) the streaming data could be chosen to be loaded in R2s or be used in multiplication through R1. During the load phase, stationary operands are poured into connected registers in a column to fill them by using the connection among them. In such a case, however, the load and process phases cannot overlap.

V. Scalability Analysis

This section analyzes the performance of NSA, TSSA, and Meissa and explores their scalability. First, we summarize the characteristics of the three systolic architectures in Table I. The table lists the appropriate *shape* for the systolic arrays required

Table I
A SUMMARY OF THE SYSTOLIC-BASED MATRIX MULTIPLICATIONS. LEFT-TO-RIGHT AND UP-TO-DOWN INDICATE DATA FLOWING DURING PROCESSING.

Systolic Type	Shape	PE type	Left to Right	Up to Down	Stored	T_{load}	$T_{process}$	T_{out}	T_{total}
NSA	$n \times p$	MAC	matrix $A_{n\times m}$	matrix $B_{m\times p}$	partial output	0	$n+m+p-2$	$n+p-1$	$2n+m+p-2$
TSSA	$m \times p$	MAC	matrix $A_{n\times m}$	partial output	matrix $B_{m\times p}$	m	$n+m+p-2$	$n+p-1$	$n+2m+p-2$
Meissa	$m \times p$	multipliers & adder trees	matrix $A_{n\times m}$	nothing	matrix $B_{m\times p}$	m	$n+p-1$	$n+log(m)$ $+p-1$	$n+m+log(m)$ $+p-2$

to multiply matrices of size $n \times m$ and $m \times p$ *without splitting* neither their inputs nor their outputs. Since each PE of the two common systolic arrays includes one multiplier and one adder, in total, the NSA and TSSA consist of np and mp multipliers and adders, respectively. The *shape* for Meissa indicates the size of the *multiplier* array. The $m \times p$ multiplier array of Meissa is connected to p adder trees with m leaves. Therefore, Meissa consists of mp multipliers and $p(m-1)$ adders. Unlike the MAC units in NSA and TSSA, the multipliers of Meissa pass through the intermediate data in only one direction. As a result, only during the load phase does data pass through from up to down. The last four columns of Table I combine Equations 2 to 13. To analyze the impact of scaling the input size on performance, we vary one dimension of the inputs at a time and study the total time steps of multiplication together with the number of required PEs, which counts multipliers and adders separately.

As n increases (Figure 4a), time steps increase for all three systolic arrays. While for $n < 128$ the NSA works faster than the TSSA, it is the opposite for $n > 128$. However, in both cases, Meissa is the fastest. Thus, for large matrices, the NSA is not suggested. Besides, its number of required PEs keeps increasing as the input size scales up (purple line in Figure 4a). As Figure 4b illustrates, increasing m has a similar impact on the total time as increasing n does, whereas when $m > 128$, the TSSA works more slowly than the NSA, and Meissa is still the fastest for all ms. However, other factors must also be considered. First, although compared to Meissa, the NSA requires fewer PEs to deliver the same performance, the NSA is not flexible for splitting – which is necessary for $m > 128$ where PEs may not fit into an FPGA. This

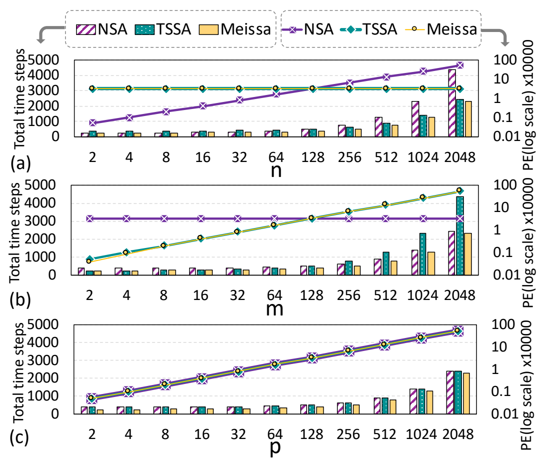


Figure 4. **Total Time Steps for Matrix Multiplication:** Analyzing the performance (T_{total}) of three systolic arrays when **(a)** n varies from 2 to 2048, $m = 128$, and $p = 128$, **(b)** m varies from 2 to 2048, $n = 128$, and $p = 128$, and **(c)** p varies from 2 to 2048, $n = 128$, $m = 128$.

is because the outputs of NSA are stored within the cells and must be drained after each partial multiplication, which limits pipelining. Moreover, we must consider the trend of scaling m together with either n or p. The reason is that in a DNN, C and K (see Equation 9) scale accordingly. While C corresponds to m, K may correspond to n or p. Either way, our choice will lead to a decision that Meissa is faster and requires a decent number of PEs. Finally, increasing p (Figure 4c) has a similar impact on all three systolic arrays. Therefore, for any p size, choosing Meissa is beneficial in terms of performance and the required PEs to deliver a given performance. In summary, in Figure 4b, the latency of Meissa grows sublinearly, in Figure 4a and 4c, the latency of all designs grow linearly and, in all cases, Meissa is the fastest (has the lowest latency).

VI. Implementation

We implement NSA, TSSA, and Meissa using Xilinx Vivado HLS and relevant `#pragmas` as hints to describe desired microarchitectures. We validate systolic-array generation by using the `Analysis` tool of Vivado HLS. The top function of all three systolic arrays is similar as shown in Figure 5a. The top function sequentially streams the operands of the matrix multiplications (`A` and `B`) and iteratively calls the specific systolic-based multipliers, the implementations of which are explained in the following. We partition the buffers that include the operands to enable parallel accesses to BRAM. For streaming the operands, partitioning through only one dimension is sufficient.

For NSA, `A` and `B` are parallelograms, which can either be streamed in this shape or padded in the top function. Seeking fair comparison, we choose the former to eliminate the extra steps for reshaping in FPGA, which negatively impacts the performance of our peers. Since in the NSA the elements of the output are created and stored in the MAC units, we simply generate the array using a nested loop of `MACs` (line 5 Figure 5b), the output of which is fed back to the inputs (`temp` register). We implement flowing matrices `A` and `B` through the array using a sliding window that moves along with the iterations of the outermost loop (line 1 Figure 5b).

Similar to NSA, the TSSA (Figure 5c) is made of `MACs`. However, its interconnection among the MACs differs. In this case, we implement the MAC array by connecting `MacCols` (defined in bottom of Figure 5c). A `MacCol` consists of m `MACs`. As line 5 in Figure 5c shows, we generate p `MacCol` by using the `#unroll` pragma. Within a `MacCol`, the outputs flow from up to down. Such a flow is implemented out of the `MacCol`, in lines 7 to 10 in Figure 5c. The flow of matrix `A`, on the other hand, is implemented similarly to that of `A`

```
function top (A[n][m], B[p][m], out[n][p], length):

1   #pragma HLS INTERFACE axis port=A
2   #pragma HLS INTERFACE axis port=B
3   #pragma HLS INTERFACE axis port=out
4   buff_A[n][m]
5   #pragma HLS ARRAY_PARTITION
          variable=buff_A complete dim=2
6   buff_B[p][m]
7   #pragma HLS ARRAY_PARTITION
          variable=buff_B complete dim=2
8   #pragma HLS dataflow
    ...
    Populate buffers
9   Multiply_XXX(buff_A, buff_B, out, length)
```

(a) Generic Top Function

```
function Multiply_NSA (A[n][m], B[p][m], out[n][p], length):

1   for i=0 to (n+m-1)+(p-1): #pragma HLS pipeline
2     for j=0 to p:           #pragma HLS unroll
3       for k=0 to n:         #pragma HLS unroll
4         if i-j>0:
5           temp[i][j] = MAC(A[k][i], B [j][i], temp[i][j])
6   out=temp
```

(b) NSA

```
function Multiply_TSSA (A[n][m], B[p][m], out[n+p-1][p],
                        length):
    MAC unit registers for passing through data
1   temp_in[p][m]
2   temp_out[p][m]

3   for i=0 to (n+m-1)+(p-1):   #pragma HLS pipeline
4     if i<length:
5       for j=0 to p:            #pragma HLS unroll
6       if i-j>0:
7         MACcol(A[i-j], B[j], temp_in[j], temp_out[j])
8         out[i][j] = temp_out[j][m-1]
9         for k=1 to m:
10          temp_in[j][k] = temp_out[j][k-1]

function MACcol (A[m], B[m], C[m], out[m]):

1   for i=0 to m:  #pragma HLS unroll
2     out[i] = MAC(A[i], B[i], C[i])
```

(c) TSSA

```
function Multiply_Meissa(A[n][m], B[p][m], out[n][p], length):
1   temp[p][m]
2   #pragma HLS ARRAY_PARTITION variable=temp complete
3   for i=0 to (n+p-1):   #pragma HLS pipeline
4     if i<length:
5       for j=0 to p:     #pragma HLS unroll
6         if i-j>0
7           HadamardProduct(A[i-j], B[j], temp[j])
8           out[i-j][j] = AdderTree(temp[j])

function HadamardProduct (A[m], B[m], out[m]):

1   for i=0 to m:  #pragma HLS unroll
2     out[i] = A[i] * B[i]

function AdderTree (in[m]):

1   out = 0
2   for i=0 to m:  #pragma HLS unroll
3     out = out + in[i]
4   return out
```

(d) Meissa

Figure 5. **HLS Pseudo Codes**: An overview of the main functions and the `pragmas` to generate the systolic arrays: **(a)** top function, and the multiply function for **(b)** NSA, **(c)** TSSA, and **(d)** Meissa.

and `B` in the NSA, in the outermost loop (line 3 Figure 5c). More specifically, at each time step, which corresponds to an iteration of the outer loop, a window of size $m \times p$ slides over matrix `A`. Note that here, only `A` is a parallelogram.

Meissa (Figure 5d) uses the same `#unroll` pragma as the peer architectures to generate the systolic arrays, whereas the main nested loop of Meissa (lines 3 to 8 of Figure 5d) generates `HadamardProducts`, each of which consists of m multipliers. The instances of multipliers are also generated by `#unroll` pragma. The output of a `HadamardProduct` is then added by an `AdderTree` (line 8 Figure 5d). The integer expressions are balanced by default Viviado HLS [32]. To evaluate this, for generating our desired parallel and *balanced* adder trees, we use `#expression_balance` pragma to evaluate the difference in the iteration latency by enabling/disabling this feature. Similar to the TSSA, flowing `A` through the multipliers is implemented in the outermost loop, whereas here, neither `A` nor `B` is a parallelogram.

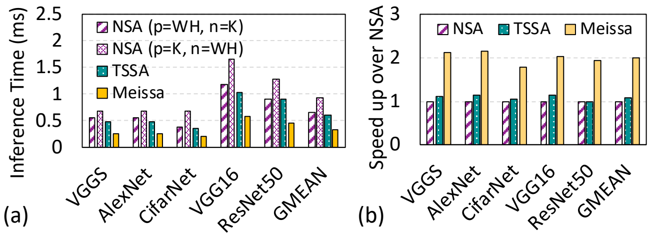


Figure 6. **Performance of DNNs: (a)** inference time and **(b)** speedup of Meissa and the TSSA over the NSA.

VII. Experiment Setup

We target the SoC system of a PYNQ-z1 board and hence synthesize and implement Meissa and the baselines on its FPGA, a ZYNQ XC7Z020. We verify the functionality of our HLS implementations using regression tests. We choose the largest possible array (i.e., 32×32, which implies $n = m = p = 32$) that fits in our target FPGA – this is defined by NSA and TSSA. We review the post-implementation latency, resource utilization, and power consumption reported by Vivado. All implemented architectures use similar memory stream interfaces to communicate with an external DDR3 memory. The inputs and outputs of the systolic architectures are transferred through the AXI stream interface. The clock frequency is set to 100 MHz. Since unlike throughput, the minimum steps (maxixmum performance) do not depend on clock frequency, we chose a moderate clock frequency. Increasing the clock frequency equally impacts NSA, TSSA, and Meissa by either removing positive slacks or increases the number of cycles. All computations are 32-bit integers. We execute the single-batch inference of five DNNs, including VGGS, AlexNet, CifarNet, VGG16, and ResNet50, consisting of various-size matrix multiplications (dimensions between 16 to 50176). Since we aim to improve and evaluate the latency of single-batch inference, which is the case in the edge, we do not overlap different runs although multiplications of a single run overlap (for TSSA and Meissa). A CPU host (e.g., the ARM A9 of PYNQ-z1 board) coordinates the relation between the layers of DNNs and applies the activation functions.

VIII. Evaluation Results

Our success metric to achieve a faster systolic array is *latency* (i.e., single-batch inference time for DNNs). Besides, to show that Meissa achieves higher performance more *efficiently*, we evaluate recourse utilization as well as power and

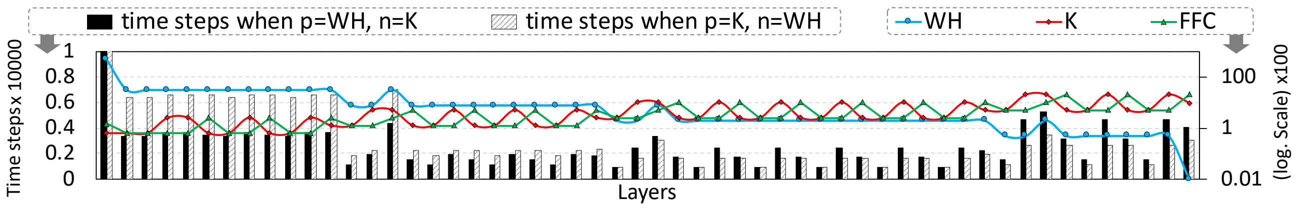


Figure 7. **Layer-by-Layer Analysis:** comparing multiplication time steps using NSA for the layers of ResNet50.

energy consumption. As throughput is defined by FLOP/Byte ratio (defined by the shape of systolic array) and memory bandwidth (i.e., FLOPS=FLOP/Byte×Bytes/Sec), our implemented NSA, TSSA, and Meissa with the same shapes have the same throughput.

A. Neural Network Performance

Since the 32×32 systolic arrays are smaller than many matrices in the target DNNs, we need to split the original matrix multiplications into sub-multiplications and call the systolic array, implemented on FPGA, several times to perform the matrix multiplications of a single layer. For TSSA and Meissa, the sub-multiplications can be arranged such that consecutive ones reuse a common *stationary* operand to save energy and the time of loading the operand (i.e., m time steps as listed in Table I). For the NSA, however, as the partial outputs are stored in the PEs, for every sub-multiplication, both operands must be restreamed into the array. This is one of the reasons that the NSA has been less frequently used.

Figure 6a compares the inference time (excluding CPU time) of three systolic arrays. For the NSA, choosing the direction of offloading the output impacts the total latency. In our analysis, while m is always FFC, the direction of offloading is defined by how we assign K and WH to n and p (i.e., the non-common dimension of two operands of matrix multiplication). To clarify this matter, Figure 6a shows the latency for both cases. As the inference times of the five DNNs suggest, the latency of the NSA in which $p = WH$ and $n = K$ is similar to the TSSA. The reason, as mentioned earlier, is that the stationary nature of TSSA helps eliminate the unnecessary time steps for reloading. However, if we had not split the original matrix, the TSSA would have performed slightly faster but not necessarily with fewer PEs (see Figure 4). Figure 6b illustrates the speedup of Meissa and the TSSA against the NSA. As the figure suggests, on average, Meissa works 1.99× and 1.83× faster than NSA and TSSA, respectively.

To investigate the performance of NSA, Figure 7 shows the time steps required to perform the matrix multiplication of each layer of ResNet50 along with the matrix dimensions. As the figure suggests, when $WH > K$, assigning $p = WH$ is better, which indicates offloading the output through its shorter dimension. When $K > WH$, the opposite assignment works better. While for ResNet50 and the other evaluated DNNs in this paper, choosing $p = WH$ and $n = K$ results in better overall performance, this might not be a general rule for all DNNs and should be carefully chosen for an NSA. Such a dependency of performance on a design choice is another downside of NSA and a reason for its lower popularity.

Table II
RESOURCE UTILIZATION AND POWER CONSUMPTION.

	NSA	TSSA	Meissa	Available
BRAM(18Kb)	70	51	32	140
LUT	3427	3970	2093	53200
FF	5423	5152	4365	106400
DSP	96	96	96	220
Dynamic Power(W)	0.067	0.073	0.037	N/A
Static Power(W)	0.118	0.131	0.123	N/A

B. Resource Utilization & Power Consumption

The resource utilization and the average power consumption of 32×32 NSA, TSSA, and Meissa, as well as the available resources of the target FPGA, are listed in Table II. Regardless of their interconnections, all implemented systolic architectures are similar in the total number of multipliers, adders, and registers (as they all store a value in their PEs, either an operand or a partial output). As a result, even though Meissa uses slightly fewer FFs and LUTs because of its multiplier-plus-adder-tree architecture, we do not see significant differences in resource utilization. Note that the smaller BRAM for Meissa stems from the non-parallelogram inputs streamed and buffered from external memory. Alternatively, for the other two systolic arrays, BRAM can be traded for logic if we choose to reshape the operands in the FPGA.

As listed in Table II, while the static power consumption of three designs is similar, their dynamic power consumption (including clock activity) differs. The main reason for this is the difference in the interconnections among the PEs and the amount of transferred data. To better explore the rationale behind the different power consumption, we compare the breakdown power consumption in Figure 8. As the figure suggests, the lower power consumption of Meissa stems from less signal transmission for two reasons: (i) during the multiplication steps, the multipliers transfer data only in one direction (the other connections are used only for loading the stationary operand), and (ii) the topology of the adder tree reduces the transmission of data across the PEs.

C. Energy Consumption

As a metric to evaluate *efficiency*, we compare energy per inference in Figure 9. Delivering higher performance at lower

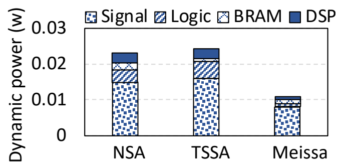


Figure 8. **Dynamic Power Consumption:** the break-down of power consumed by signal transmissions, logic, BRAM, and DSP for three systolic architectures. This diagram does not include the power consumed by clock.

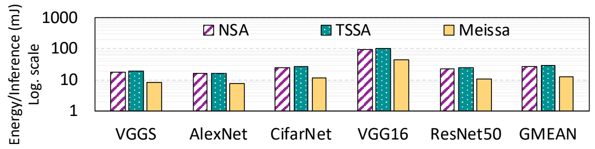


Figure 9. **Energy Consumption:** comparing the energy consumed by three systolic architectures to perform an inference.

energy is particularly important in applications with limited power resources, such as in embedded systems. As reported in Figure 9, on average, Meissa consumes 2.12× and 2.27× less energy compared to TSSA and NSA, respectively. Regardless of the implementations, the absolute number of multiplication and addition operations is similar. Thus, the key parameter, which leads to the lower energy of Meissa, is the way we perform a given number of operations (i.e., the PEs and their interconnections) that together build the systolic array.

IX. Conclusions

Efficiently utilizing the limited resources of small FPGAs to execute matrix multiplication quickly is important in real-time applications (e.g., using DNNs to detect objects in self-driving cars). To do so, we proposed Meissa, a fast and scalable systolic architecture for matrix multiplication, in which, unlike prior work, the latency grows *sub-linearly* with the input size.

Acknowledgment

We gratefully acknowledge the support of NSF CSR 1815047.

References

[1] J. M. Landsberg, "Tensors: geometry and applications," *Representation theory*, vol. 381, no. 402, p. 3, 2012.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[3] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[4] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[5] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[6] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[7] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[8] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré, "Caffe con troll: Shallow ideas to speed up deep learning," in *Proceedings of the Fourth Workshop on Data analytics in the Cloud*. ACM, 2015, p. 2.

[9] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: Analog convnet image sensor architecture for continuous mobile vision," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2016, pp. 255–266.

[10] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on FPGA*. ACM, 2017, pp. 75–84.

[12] S. Wang, D. Zhou, X. Han, and T. Yoshimura, "Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1032–1037.

[11] Y.-H. 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.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[14] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Eridanus: Efficiently running inference of dnns using systolic arrays," *IEEE Micro*, vol. 39, no. 5, pp. 46–54, 2019.

[15] H. Kung, B. McDanel, and S. Q. Zhang, "Adaptive tiling: Applying fixed-size systolic arrays to sparse convolutional neural networks," in *2018 24th International Conference on Pattern Recognition (ICPR)*. IEEE, 2018, pp. 1006–1011.

[16] H. Kung and C. E. Leiserson, "Systolic arrays (for vlsi)," in *Sparse Matrix Proceedings 1978*, vol. 1. Society for Industrial and Applied Mathematics, 1979, pp. 256–282.

[17] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 29.

[18] H. Kung, B. McDanel, S. Q. Zhang, X. Dong, and C. C. Chen, "Maestro: A memory-on-logic architecture for coordinated parallel use of many systolic arrays," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 42–50.

[19] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "Fulcrum: a simplified control and access mechanism toward flexible and practical in-situ accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 556–569.

[20] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 2004, pp. 133–137.

[21] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 2007, pp. 1–12.

[22] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus," *IBM Research Report RC24704*, no. W0812-047, 2009.

[23] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Towards collaborative inferencing of deep neural networks on internet of things devices," *IEEE Internet of Things Journal*, 2020.

[24] Autel, *Autel X-Star Quadcopter*, 2020 (accessed June 7, 2020). [Online]. Available: https://www.autelrobotics.com/x-star-camera-drone/

[25] DJI, *DJI Mavic Air*, 2020 (accessed June 7, 2020). [Online]. Available: https://www.dji.com/mavic-air

[26] S. Y. Kung, "Vlsi array processors," *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*, 1988.

[27] H.-T. Kung, "Why systolic architectures?" *IEEE computer*, vol. 15, no. 1, pp. 37–46, 1982.

[28] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[29] H. Lim, V. Piuri, and E. E. Swartzlander, "A serial-parallel architecture for two-dimensional discrete cosine and inverse discrete cosine transforms," *IEEE Transactions on Computers*, vol. 49, no. 12, pp. 1297–1309, 2000.

[30] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the 24 International Conference on ASPLOS*. ACM, 2019, pp. 821–834.

[31] R. Urquhart and D. Wood, "Systolic matrix and vector multiplication methods for signal processing," in *IEE Proceedings oF Communications, Radar and Signal Processing*, vol. 131, no. 6. IET, 1984, pp. 623–631.

[32] V.-H. Xilinx, "Vivado design suite user guide-high-level synthesis," 2014.