A Novel Fault-Tolerant Architecture for Tiled Matrix Multiplication

Sandeep Bal*, Chandra Sekhar Mummidi*, Victor da Cruz Ferreira†, Sudarshan Srinivasan†, Sandip Kundu*
*University of Massachusetts, Amherst, USA
†Intel Corporation, Bengaluru, India

***Abstract*—General matrix multiplication (GEMM) is common to many scientific and machine-learning applications. Convolution, the dominant computation in Convolutional Neural Networks (CNNs), can be formulated as a GEMM problem. Due to its widespread use, a new generation of processors features GEMM acceleration in hardware. Intel recently announced an Advanced Matrix Multiplication (AMX®) instruction set for GEMM, which is supported by 1kB AMX registers and a Tile Multiplication unit (TMUL) for multiplying tiles (sub-matrices) in hardware. Silent Data Corruption (SDC) is a well-known problem that occurs when hardware generates corrupt output. Google and Meta recently reported findings of SDC in GEMM in their data centers. Algorithm-Based Fault Tolerance (ABFT) is an efficient mechanism for detecting and correcting errors in GEMM, but classic ABFT solutions are not optimized for hardware acceleration. In this paper, we present a novel ABFT implementation directly on hardware. Though the exact implementation of Intel TMUL is not known, we propose two different TMUL architectures representing two design points in the area-power-performance spectrum and illustrate how ABFT can be directly incorporated into the TMUL hardware. This approach has two advantages: (i) an error can be concurrently detected at the tile level, which is an improvement over finding such errors only after performing the full matrix multiplication; and (ii) we further demonstrate that performing ABFT at the hardware level has no performance impact and only a small area, latency, and power overhead.**

***Index Terms*—accelerator, matrix multiplication, abft, concurrent error detection, low power**

I. Introduction

General Matrix Multiplication (GEMM) operations are widespread in machine-learning and high performance computing applications [1]. With its growing usage in various applications and services machine-learning has become quite pervasive [2]. Data center workloads are often dominated by machine learning [3]. Though varied in computation models, ML applications ranging from Convolutional Neural Networks (CNNs), Recurrent Neural Networks, and Multi-Layer Perceptions (MLPs) all use GEMM at their core [4]. Thus GEMM operations constitute lion's share of computation in data centers [5].

Given the prevalence of GEMM operations in modern computation, CPU and GPU providers are workings towards hardware acceleration to increase performance. As a case in point, 4th generation Intel Xeon processors include accelerators for matrix operations enabled by the Advanced Matrix Extensions (AMX) instruction set, which speeds up AI applications in data centers. In AMX, a matrix is tiled into sun-matrices where the elements of a sub-matrix can be packed into a 1kB register. AMX architecture features 8 such registers. AMX architecture includes a tiled matrix multiplication unit that accepts data from two AMX registers (tiled matrices) producing a tiled matrix output aligned for storing in an AMX register. The tiled matrix multiplication unit (TMUL) can perform up to 1024 parallel multiply and add operations, a significant improvement over the usual sequential operations of a CPU. As a result, GEMM is accelerated significantly [6]. Tiling a matrix so that each tile's data matches the target architecture's register width is a popular solution [6]. Matrix of any size can be split into tiled matrices for faster computations [6].

Occurrences of silent data corruption (SDC) in data centers have recently been reported by Google and Meta [7], [8]. The majority of the applications in their data centers involve machine-learning workloads and recommendation models. Matrix multiplications constitute the dominant kernel in these workloads [4]. Detecting SDC during execution is crucial but challenging [7]. In safety-critical systems SDC poses a major threat to safety [9]. This motivates us to investigate the online detection of execution errors in TMUL architecture.

There have been numerous studies in the past that have tackled the problem of error detection in matrix multiplication. Typically, these solutions involve redundant execution using double or triple modular redundancy to detect or correct errors in hardware [10]. We do not consider these solutions due to their high overhead. In data centers with large fleet sizes and high power consumption, such error detection techniques are not ractical [11]. Algorithm-based fault tolerance (ABFT) has been proposed as a low-cost alternative to hardware redundancy for detecting matrix multiplications errors at run time [12].

Various ABFT solutions for matrix multiplications have been proposed in the literature [13] [14]. Recently, ABFT was studied by NVIDIA for error detection in matrix multiplication [15]. So far, ABFT has only been implemented as a software solution. Given the emergence of hardware accelerators in general and TMUL in particular, we investigate the cost/benefit of implementing ABFT directly onto the hardware accelerator.

Our work in this paper makes the following contributions to concurrent error detection in TMUL units:

- We propose two baseline designs for supporting AMX multiplication instructions in hardware.
- We demonstrate that ABFT can be incorporated in the hardware directly for fault-tolerant operation of tiled multiplication
- Based on experiments, we show that area, power and latency overhead of our approach is less than 1.5%, 1.3% and 7.7% for the first baseline design and less than 1.8%, 7.2% and 1% for the second baseline design, respectively.

978-3-9819263-7-8/DATE23/© 2023 EDAA

- Our proposed modifications to the baseline designs does not impact the frequency of the architecture.
- We implemented a simulation environment to demonstrate the algorithm's efficacy using fault injection experiments.

II. Background

A. TMUL

To accelerate AI workloads, Advanced Matrix Extensions (AMX) instructions were introduced [6]. AMX architecture incorporates a tile matrix multiply unit (TMUL) along with low-latency, high-bandwidth memory access to accelerate GEMM operation [6]. AMX features 1kB registers to hold tile data. TMUL takes its operands from two source registers and producing an output which is stored in a destination AMX register. TMUL comprises a 2D array of processing elements (PE), where each PE performs a multiply and add operation, also known as fused multiply and adds (FMA) units as shown in Figure 1.

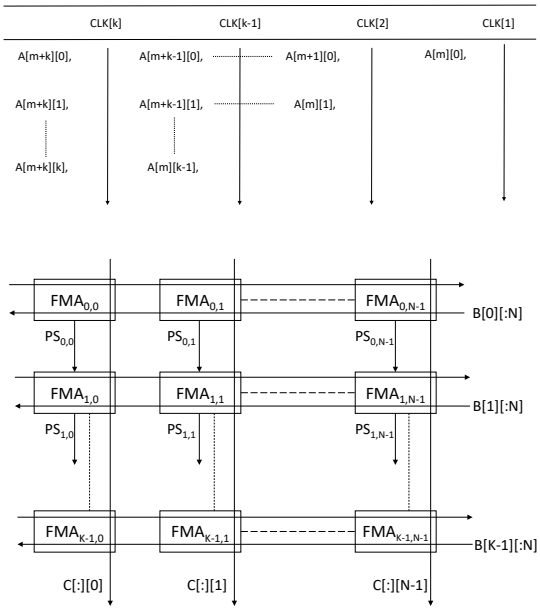

Fig. 1. TMUL unit

During TMUL execution, up to 1024 FMAs execute in parallel drawing large amounts of power. To address this power issue, the processor frequency is lowered during TMUL operation and every core in the architecture will be instilled with power controllers to facilitate this operation.

B. TMUL Architecture

The exact implementation of TMUL in 4th generation Xeon has not been published. However, based on the available details, in this section, we describe two baseline design choices. The first baseline TMUL design is shown in Figure 2 is inspired by the systolic array architecture published in [16]. We modify this design to be applicable to tile-based matrix multiplications. Further, although Intel's proprietary TMUL unit is not revealed, we approximate Intel's TMUL design based on available information and produce our second baseline design as shown in Figure 3 [6]. Based on the AMX register size and the supported data types, the size of the TMUL can be inferred. Below we illustrate basic TMUL operation between two matrices A and B which produces C based on these dimensions.

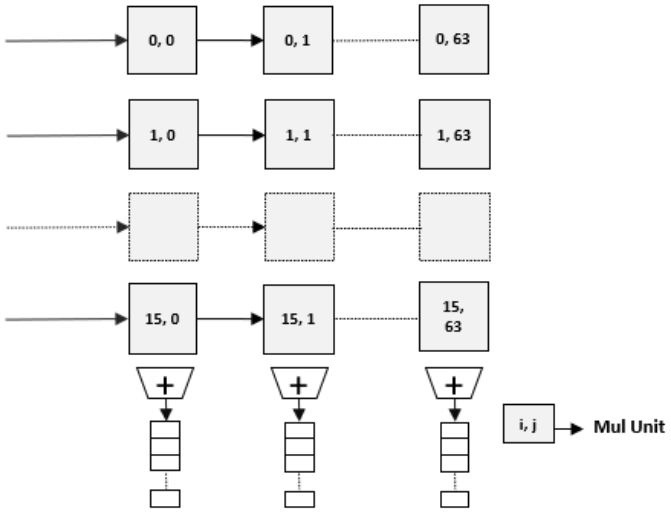

Fig. 2. Baseline Tiled Multiplication Unit Design 1

$$A_{64\times16} = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,15} \\ a_{1,0} & a_{1,1} & \dots & a_{1,15} \\ \vdots & \vdots & \ddots & \vdots \\ a_{63,0} & a_{63,1} & \dots & a_{63,15} \end{bmatrix} \quad (1)$$

$$B_{16\times64} = \begin{bmatrix} b_{0,0} & b_{0,1} & \dots & b_{0,63} \\ b_{1,0} & b_{1,1} & \dots & b_{1,63} \\ \vdots & \vdots & \ddots & \vdots \\ a_{15,0} & a_{15,1} & \dots & a_{15,63} \end{bmatrix} \quad (2)$$

$$C_{64\times64} = \begin{bmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,63} \\ c_{1,0} & c_{1,1} & \dots & c_{1,63} \\ \vdots & \vdots & \ddots & \vdots \\ c_{63,0} & c_{63,1} & \dots & c_{63,63} \end{bmatrix} \quad (3)$$

1) Baseline TMUL 1:

a) Design: The first baseline design is shown in Figure 2. It has 1024 multiplier units denoted by $[i, j]$. They are distributed across 16 rows and 64 columns. Each column has a dedicated tree adder to generate an output. The tree adders consist of 16 basic adders divided into four stages.

b) Operation: In the first 64 clock cycles, $B_{16\times64}$ is serially loaded column-wise into the multiplier units. $A_{64\times16}$ is transposed ($A^T_{64\times16}$) and shifted right row-wise. In the 65^{th} cycle, the first column of the $A^T_{64\times16}$ is loaded in the first column of the multiplier array units from $[0, 0] \rightarrow [63, 0]$ and multiplied with the first column of $B_{16\times64}$, element-wise. In the 65^{th} clock cycle, the first element of the output is also produced and offloaded by adding all the 16 partial products in the tree adder. In the subsequent cycles, the following columns of $A^T_{64\times16}$ are loaded in the multiplier array units on a one-column per cycle basis. It takes 64 clock cycles to load

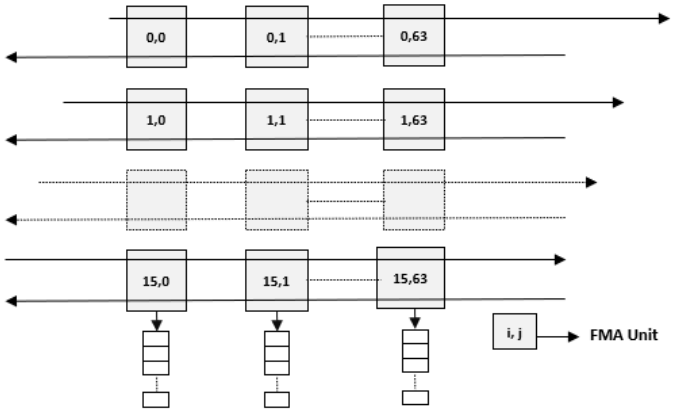

Fig. 3. Baseline Tiled Multiplication Unit Design 2

$A^T_{64\times16}$ completely into the array structure while multiplying with $B_{16\times64}$. From the 129^{th} cycle, offloading of the $A^T_{64\times16}$ matrix begins. The first column of the $A^T_{64\times16}$ leaves the last column $[0,63] \rightarrow [15,63]$ of the multiplier unit, leaving behind its output element. It takes 64 more cycles to drain $A^T_{64\times16}$ and generate the complete results by the 192^{nd} clock cycle.

2) Baseline TMUL 2:

a) Design: Our second baseline design is shown in Figure 3. It has 1024 FMA units denoted by $[i,j]$. They are distributed across 16 rows and 64 columns as well.

b) Operation: For this architecture, $B_{16\times64}$ is serially shifted into the multiplier units for the first 64 clock cycles. $A^T_{64\times16}$ is arranged in a systolic array data flow at the input. In the 65^{th} cycle, $a_{0,0}$ of $A^T_{64\times16}$ is loaded in the FMA unit $[0,0]$ and multiplied to $b_{0,0}$ of $B_{16\times64}$. In the 66^{th} cycle, the partial sum from FMA unit $[0,0]$ moves down to the FMA unit $[1,0]$ along with $a_{0,0}$ of $A^T_{64\times16}$ being forwarded to FMA unit $[0,1]$, where the next partial sum is generated. Meanwhile, $a_{1,0}$ is loaded in the FMA unit $[0,0]$ and $a_{0,1}$ from the second row of $A^T_{64\times16}$ is loaded in the FMA unit $[1,0]$. This data flow continues until the 129^{th} clock cycle, when $a_{0,0}$ of $A^T_{64\times16}$ is unloaded from the FMA unit $[0,63]$. The combined process of loading and unloading of $A^T_{64\times16}$ in systolic data flow continues till the 192^{nd} clock cycle. At the 193^{th} clock cycle, $c_{0,0}$ of the output tiled matrix $C_{64\times64}$ is generated downwards from the FMA unit $[15,0]$. This systolic data flow process of loading and unloading $A^T_{64\times16}$ and offloading $C_{64\times64}$ lasts until the 320^{th} clock cycle, when the final element $c_{63,63}$ of $C_{64\times64}$ is generated from the FMA unit $[15,63]$.

C. Traditional ABFT for GEMM

ABFT for GEMM checks the correctness of the operation by checking reduced output values. In GEMM ABFT technique, column checksum for input matrix *A* of dimension *nxm*, $RC_j = \sum_{i=0}^{n-1}(a_{ij})$ and row checksum for input matrix *B* of dimension *mxp*, $CC_i = \sum_{j=0}^{p-1}(b_{ij})$ are appended to them respectively, where i and j are the row and column indices respectively. These augmented matrices multiplied with GEMM generate an extended output matrix *C* of dimension $(n+1)\times(p+1)$. Next, the row checksum and column checksum of output matrix C is computed for the first n rows and the first p columns and compared against the last row and last column of the matrix C. In an error-free operation, the following identities must hold true:

$$C_{nj} = \sum_{i=0}^{n-1} C_{ij} \tag{4}$$

$$C_{ip} = \sum_{j=0}^{p-1} C_{ij} \tag{5}$$

$$C_{np} = \sum_{i=0}^{n-1} C_{ip} = \sum_{j=0}^{p-1} C_{nj} \tag{6}$$

Since GEMM basic operations such as partial products and summations are commutative and associative, any mismatch indicates an error, while the location of the erroneous element(s) can be found from the indices of the mismatched rows and columns.

D. Modified ABFT for GEMM

To optimize the above, a modified ABFT technique only computes the input checksum from the column of one of the matrices and checks against the column checksum of the output matrix. This modification reduces the number of summation operation that needs to be performed, overall. In modified GEMM ABFT technique, column checksum for input matrix *A* of dimension *nxm*, $RC_j = \sum_{i=0}^{n-1}(a_{ij})$ is appended to *A* matrix, where i and j are the rows and column indices respectively. This augmented matrix is then multiplied to unchanged *B* in GEMM which generates an extended output matrix *C* of dimension $(n+1)\times(p)$. Further, the column checksum of output matrix C is computed for the first n rows and compared against the last row of the output matrix C. In an error-free operation, the following identities must hold true:

$$C_{nj} = \sum_{i=0}^{n-1} C_{ij} \tag{7}$$

$$C_{np} = \sum_{i=0}^{p-1} C_{ip} \tag{8}$$

Again, since modified GEMM basic operations such as partial products and summations are commutative and associative as well, any mismatch will indicate an error, whereas in this case, the location of the erroneous element(s) will not be apparent. Since we are focusing on error detection and not a correction in the GEMM operation, this trade-off is justified. It also reduces the total number of computations, the power, and hardware overhead making it more suitable for hardware implementation.

III. Proposed tiled-ABFT architecture for GEMM

In this section, we extend the baseline architectures described earlier in Section II. The extensions will facilitate concurrent error detection by implementing ABFT at the hardware level of the previous architectures. We call these extensions Fault Tolerant tiled-Multiplication Unit or FTMU. In the following

subsections, we discuss the two FTMU designs proposed in this paper.

A. FTMU for baseline design 1

a) Design: Figure 4 represents the FTMU architecture for the first baseline design. In addition to the core architecture, we add two extra units. One is called the Input Checksum (IC) unit and the other is the Output Checksum (OC) unit. The IC unit consists of 16 basic adders to compute the input checksums, one for each of the 16 rows of the multiplier unit. Similarly, the OC consists of 64 basic adders to compute the output checksums, one for each of the 64 tree adder units. Additionally, we propose adding interconnects at the base of each multiplier unit to load $B_{16\times64}$ in 1 clock cycle. This would reduce the number of clock cycles by 63.

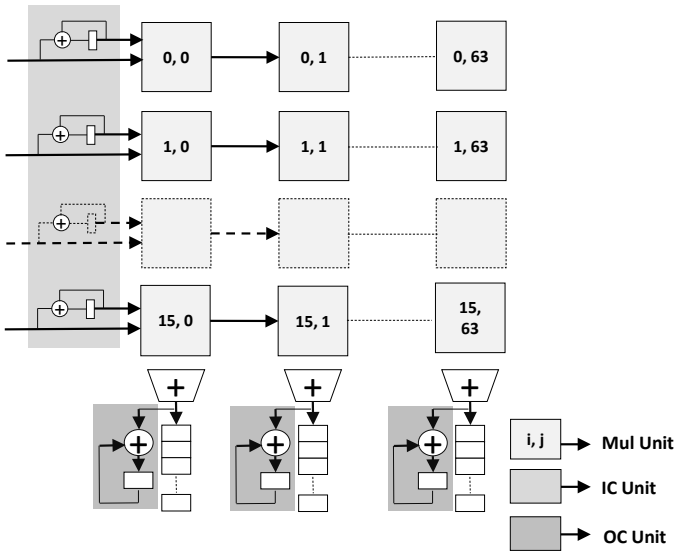

Fig. 4. Fault-Tolerant Tiled Multiplication Unit Design 1

b) Operation: To illustrate the operation, we refer to equations (1), (2), and (3). Apart from loading $B_{16\times64}$ in the multiplier units in 1 clock cycle through the interconnects, the core GEMM operation remains the same as previously discussed in Section II. In addition to that, when the columns of $A^T_{64\times16}$ are loaded in the baseline 1 architecture at the input, each element $a_{i,0} \rightarrow a_{i,15}$ are added and stored in the registers $0 \rightarrow 15$ of IC on a per-clock basis. These registers run from top to bottom in IC unit. This operation is performed in parallel to loading and adding $A^T_{64\times16}$ in the architecture and for all the columns of $A^T_{64\times16}$. Further, when the elements of $C_{64\times64}$ are offloaded, the OC unit adds and stores the elements of each column in the registers. This happens in parallel to the offloading of $C_{64\times64}$. For example, the elements $c_{0,i} \rightarrow c_{63,i}$ are added and stored on a per-clock basis in the i^{th} register of OC. Due to the interconnects, the final output $C_{64\times64}$ is computed in 129^{th} clock cycle instead of 192^{nd} as discussed earlier. Finally, from $128^{th} \rightarrow 130^{th}$ clock cycle, the final IC will be multiplied with each column of $B_{16\times64}$ on a per clock basis to generate the OC_{input}. In an error-free computation OC and OC_{input} are equal.

B. FTMU for baseline design 2

a) Design: Figure 5 shows the proposed FTMU design 2 for the second baseline design. Similar to the FTMU design 1, we add two extra units apart from the core architecture here as well. They are the Output Checksum (OC) unit and the Input Checksum (IC) unit. Once more, the IC unit uses 16 basic adders, one for each of the 16 rows of the multiplier unit—to compute the input checksums. A basic adder for each of the 64 columns for FMA units makes up the OC.

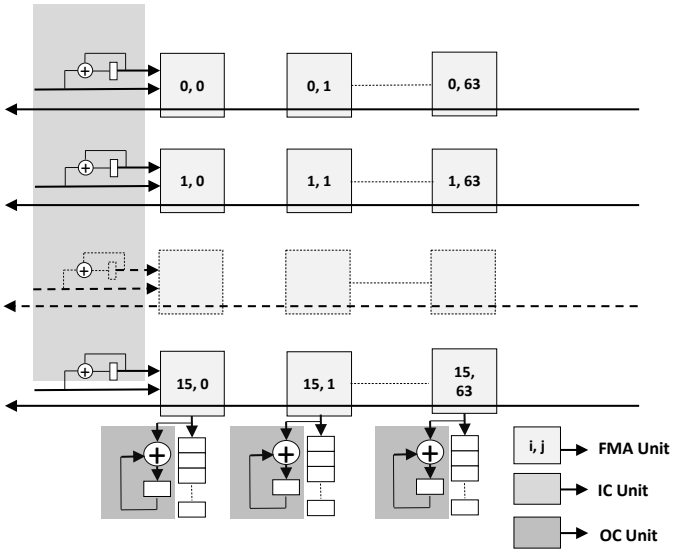

Fig. 5. Fault-Tolerant Tiled Multiplication Unit Design 2

b) Operation: The fundamental elements of the second FTMU design's GEMM operation are the same as those that were covered in Section II. Additionally, each value from $a_{i,0} \rightarrow a_{i,15}$ is added and stored in the registers $0 \rightarrow 15$ of the IC each clock cycle when the columns of $A^T_{64\times16}$ are loaded in the FTMU design 2 at the input in a systolic data flow. Once more, the IC's registers are arranged from top to bottom. This operation takes place concurrently with the loading of $A^T_{64\times16}$ in the architecture and for all of its columns. Additionally, the OC unit adds and saves the elements of each column in the registers after offloading the $C_{64\times64}$ elements. This occurs concurrently with the unloading of $C_{64\times64}$. For instance, the elements $c_{0,i}$ and $c_{63,i}$ are inserted and stored in the OC's i^{th} register on a per-clock basis. During the $319^{th} \rightarrow 321^{st}$ clock cycle, the OC_{input} will be produced by multiplying the final IC by each column of $B_{16\times64}$ on a per clock basis. OC and OC_{input} are identical in a computation that is error-free.

IV. Results

This section describes the results showing the effectiveness of the hardware-based ABFT presented in this paper and the overhead of the two proposed designs. We discuss the results from an area-power-performance standpoint for the two different designs. First, we describe the basic experimental setup. Next, describe the implementation and justification for our design choices. We then describe results from fault-injection

experiments to study the efficacy of FTMU designs. Finally, we present the power, performance, and area overhead of the proposed FTMU designs.

A. Experimental Setup and Implementation

The experiments were run on an Intel® Core™ i7-4790 processor with a clock speed of 3.60 GHz. The machine runs on an x86 architecture with 8 CPUs having 4 cores and 2 threads per core. It is equipped with 32K L1d cache, 32K L1i cache 256K L2 cache, and 8192K L3 cache.

TABLE I
FPGA EXPERIMENTATION SETUP

Tool	Vivado 2017.2
Hardware Description Language	Verilog
Board Vendor	Xilinx
Board Display Name	Artix-7 AC701 Evaluation Platform
Part	xc7a200tfbg676-2
Version	Latest

As shown in Table I, we ran an experiment on Xilinx Vivado 2017.2 for FPGA studies of the proposed architectures. We targeted the latest version of the Artix-7 AC701 Evaluation Platform to implement the hardware components in Verilog.

Table II presents the number of FPGA components used for each of the hardware components of our proposed architecture such as look-up tables, slices, input-output buffers, registers, flip flops, etc.

We implemented our hardware design components previously implemented in Xilinx in Cadence Virtuoso. We used the synthesis tool Synopsys Design Compiler Version E - 2010.12-SP5-2. We used the NCSU_Devices_FreePDK45 library which uses 45-nanometer technology to implement the designs. We wrote TCL scripts to automate our design programs (Table III).

Table IV shows the power and performance of the hardware components used in our proposed architecture implemented in FPGA. The basic adder is designed to add two 8-bit integer numbers. The baseline and the baseline FTMU architectures require adding 16 8-bit integers in one cycle. The basic adder adds 16 8-bit integers in 110 nanoseconds. To improve the performance, a tree adder performs the same operation in 24 nanoseconds. Further, each multiplication operation in an array multiplier takes 22.6 nanoseconds. Instead, a Wallace-Tree multiplier is used which performs in 12.8 nanoseconds. To make the proposed design more balanced, the tree adder is further pipelined into two stages. The pipelined tree adder performs 16 8-bit additions in 12 nanoseconds. The FTMU 2 and baseline 2 design's fused multiply-add unit, on the other hand, uses a basic adder and an array multiplier.

TABLE II
FPGA CELLS USED

	Cells Used
Basic Adder	8 LUTs - 2 Slices – 24 IOBs
Tree Adder	64 LUTs - 19 Slices – 136 IOBs
Pipelined Tree Adder	100 LUTs - 29 Slices – 137 IOBs – 40 Registers – 40 Flip Flops – 1 BUFGCTRL
Wallace Tree Multiplier	77 LUTs - 23 Slices – 32 IOBs
Array Multiplier	114 LUTs - 34 Slices – 32 IOBs

TABLE III
ASIC EXPERIMENTATION SETUP

Synthesis Tool	Synopsys Design Compiler
Version	E - 2010.12-SP5-2
Library	NCSU_Devices_FreePDK45
Technology	45 nanometer
Scripting Tool	TCL

TABLE IV
FPGA POWER AND DELAY

	Dynamic Power (*W*)	Leakage Power (*W*)	Delay (*ns*)
Basic Adder	5.3	0.2	7.3
Tree Adder	11.4	0.2	24.0
Pipelined Tree Adder	11.5	0.2	12.0
Wallace Tree Multiplier	13.6	0.2	12.8
Array Multiplier	14.8	0.2	22.6

TABLE V
ASIC POWER AND DELAY

	Area (μm^2)	Dynamic Power (μW)	Leakage Power (μW)	Delay (ns)
Basic Adder	34.1	16.1	0.4	2.6
Tree Adder	487.0	374.0	0.006	3.7
Pipelined Tree Adder	668.0	241.0	0.008	2.1
Wallace Tree Multiplier	342.0	253.5	0.005	5.0
Array Multiplier	323.5	269.0	0.005	24.8

Similarly, Table V describes the power-performance-area of the hardware components in 45-nanometer technology. The two-stage pipelined tree performs the addition of 16 8-bit integer operations in 2.1 nanoseconds. It outperforms a basic adder performing the same operation in 39 nanoseconds. On the other hand, the Wallace-Tree multiplier multiplies two 8-bit integers in 5 nanoseconds compared to 22.6 nanoseconds of an array multiplier. Hence, FTMU design 1 uses a Wallace-Tree multiplier and two-staged pipelined tree adder. On the other hand, there is no change in FTMU design 2 utilizing basic adder and array tree multiplier circuits for their FMA units.

Further, to establish FTMU's concurrent error detection capability, we run an experiment in a simulated environment in C++ Version 11. The details are discussed in the following subsection.

B. Fault Injection Experiment

We performed a fault injection experiment on two 64×16 and 16×64 matrix inputs with random 8-bit integer numbers. The errors are simulated by randomly selecting an element in the TMUL array and injecting an error in a randomly selected bit place. These experiments were run for 1000 inputs with and without errors. Our proposed tiled-ABFT approach was able to successfully detect errors in all the erroneous tiled matrix multiplication operations and did not have any false positives in the error-free runs. This is expected as ABFT guarantees 100% error detection for integers. The fault injection experiment also validates our overall implementation.

C. Analysis and Overhead

Table VI shows the power-performance-area-latency overhead of FTMU design 1 over the baseline design 1 implemented

TABLE VI
FTMU DESIGN 1 OVERHEAD

	Baseline 1	FTMU 1	Overhead (%)
Power (mW)	2500	2532	1.3
Performance (ns)	56	56	0
Area (mm^2)	385.4	391.6	1.5
Latency ($cycles$)	129	130	7.7

TABLE VII
FTMU DESIGN 2 OVERHEAD

	Baseline 2	FTMU 2	Overhead (%)
Power (mW)	2359	2528.8	7.2
Performance (ns)	219.2	219.2	0
Area (mm^2)	360	366.4	1.8
Latency ($cycles$)	320	321	1

in 45-nanometer technology. The table shows that the power overhead is only 1.3% for the FTMU design 1 along with an area overhead of 1.5%. It also shows that there is a negligible latency overhead of 7.7% due to one extra clock cycle for the output checksum whereas there was no performance overhead observed.

Similarly, Table VII shows that for baseline design 2, the power and area overhead to implement the FTMU design 2 in hardware is 7.2% and 1.8%, respectively. Again, there is an overhead in latency of 1% with no performance overhead.

D. Fault Diagnosis

FTMU 1 and 2 designs compare column checksum. Thus, we can diagnose the faulty column. However, since row checksum is not implemented in the hardware, the diagnostic resolution is limited to faulty column only. We cannot diagnose the faulty PE. For repeatable errors, this limitation can be addressed by choosing target inputs for multiplication as described below. We assume a single fault model as ABFT is not equipped to deal with all possible multiple faults.

The modified ABFT technique was described in Section II and Section III which helps locate the faulty column through input checksums (IC) and output checksums (OC). To diagnose the faulty PE, we iteratively replace every individual row one at a time with zeroes and run the FTMU operation until the IC and OC are equal. This will help identify the faulty row. Thus, with faulty column and row information, we can identify the faulty PE which lies at the intersection of faulty row and faulty column.

V. Conclusion

GEMM is the most common operation in machine learning and HPC applications. Consequently, CPU and GPU manufacturers are implementing hardware accelerators to accelerate GEMM using a tiled matrix multiplier. Recent publications from Meta and Google document the prevalence of hardware errors in data centers. This motivates us to study online error detection in TMUL hardware. We implement ABFT directly into TMUL hardware because it blends well with the data flow architecture. Our proposed FTMU design 1 implementation of ABFT is shown to have a low overhead of only 1.3%, 1.5%, and 7.7% for power, area, and latency, respectively. Additionally, it is demonstrated that our suggested FTMU design 2 implementation of ABFT has a low overhead of just 7.2% percent, 1.8% percent, and 1% percent for power, area, and latency, respectively. Also, using fault-injection experiments, they were shown to detect all errors in integer operations. There has been no compromise in terms of performance for either of the FTMU designs presented in this paper.

VI. Acknowledgments

This research was funded in part by a grant from the National Science Foundation and Intel Corporation.

References

[1] F. W. Note, J. Huang, and R. A. van de Geijn, "Blislab: A sandbox for optimizing gemm," 2016.

[2] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information*, vol. 11, no. 4, p. 193, 2020.

[3] M. Naumov, J. Kim, D. Mudigere, S. Sridharan, X. Wang, W. Zhao, S. Yilmaz, C. Kim, H. Yuen, M. Ozdal *et al.*, "Deep learning training in facebook data centers: Design of scale-up and scale-out systems," *arXiv preprint arXiv:2003.09518*, 2020.

[4] 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 *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[5] E. Georganas, K. Banerjee, D. Kalamkar, S. Avancha, A. Venkat, M. Anderson, G. Henry, H. Pabst, and A. Heinecke, "High-performance deep learning via a single building block," *arXiv preprint arXiv:1906.06440*, 2019.

[6] "Intel® architecture instruction set extensions and future features," 2021. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf

[7] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[8] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 9–16.

[9] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[10] M. Franklin, "A study of time redundant fault tolerance techniques for superscalar processors," in *Proceedings of International Workshop on Defect and Fault Tolerance in VLSI*. IEEE, 1995, pp. 207–215.

[11] J. G. Koomey, "Worldwide electricity used in data centers," *Environmental research letters*, vol. 3, no. 3, p. 034008, 2008.

[12] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[13] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410–416, 2009.

[14] A. Roy-Chowdhury and P. Banerjee, "Algorithm-based fault location and recovery for matrix computations," in *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*. IEEE, 1994, pp. 38–47.

[15] S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2546–2558, 2021.

[16] B. Asgari, R. Hadidi, and H. Kim, "Proposing a fast and scalable systolic array for matrix multiplication," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 204–204.