FPGA-Based TPU

Austin York
york@ucdavis.edu

Bradley Manzo
brmanzo@ucdavis.edu

Gurkirat Dhillon
gddhillon@ucdavis.edu

Kent Cassidy
kdcassidy@ucdavis.edu

Electrical and Computer Engineering Department
University of California, Davis
Davis, California

Abstract—Emerging AI applications require high performance and energy efficient hardware for training large models. CPUs are inefficient for training large scale models, and GPUs are expensive and power hungry. Our FPGA-based Tensor Processing Unit (TPU) is configurable and practical for scalable computation with systolic data flow architecture that reduces memory bandwidth requirements while enabling massively parallel computation. Unlike typical TPUs, which are essentially fixed-function ASICs, our FPGA-based design is flexible to meet the requirements of different applications and customers. The customization makes it energy-efficient and cost-effective for a variety of applications. We specifically demonstrate over 4X improvement in performance and efficiency on key matrix multiplications that dominate the computational requirements of training deep learning models used in many applications.

I. Introduction and Background

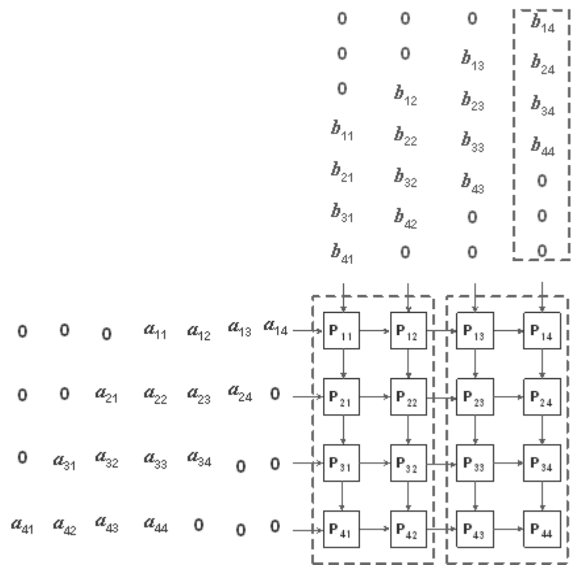
Moore's Law: the prediction that the number of transistors on computer processors will double every two years has held true for over fifty years, and is by far the most influential trend driving innovation in computer architecture. However, as transistors shrink and are packed more densely onto integrated circuits, power dissipation becomes an increasing concern as the resulting high temperatures threaten to damage circuit components. Modern general purpose processors elect to isolate transistor activity by switching off neighboring components, however this decreases overall performance. The marginal gains in performance from continuing to pack transistors more densely is known as the power wall and demands creative architectural solutions in order to surpass it.

Our TPU aims to circumvent these restrictions through the design and implementation of a Domain Specific Architecture (DSA). Matrix Multiplication (MM) is a key feature of modern processors, enabling complex graphical calculations such as ray tracing, decoding digital video and sound with Fourier transforms, and for calculating network connections within graph theory. Furthermore, MM is becoming increasingly important in recent years as it is an essential operation for machine learning and neural networks. MM for AI requires massive amounts of data and parameters to be multiplied and added together. By running a TPU as a coprocessor to the CPU, it allows for cases where the complexity is distributed between the two processors. This is preferable to overwhelming a single CPU, and if applicable, the entire set of operations may be efficiently outsourced to the TPU.

A typical CPU calculation of a 16×16 MM, each element must be read 31 times, that's a total of 7,936 sequential memory operations. To work around these limitations and increase performance beyond the capabilities of pure software running on general purpose hardware, Google developed their Tensor Processing Unit: a matrix processor specialized for neural networks. It loads parameters into the arithmetic-logic unit (ALU), then takes the data from memory. The systolic ALU array will multiply and sum the data and parameters, which require massive amounts of calculations and data throughput, without reliance on memory access, as data need only be loaded a single time. It is this design philosophy that inspired us to develop our own version of a systolic array as the core of our project alongside other techniques to reduce the memory bandwidth required for such an architecture.

II. Problem Definition

Due to the increased need for massive calculations commonly used in neural networks and machine learning, a domain specific processor is advantageous. Hence our motivation for creating a custom processor for matrix-matrix and matrix-vector computation using systolic architecture, Fig. 1. This design achieves higher computational throughput and significantly reduced memory accesses during computation due to the capability of omitting redundant reads by chaining and recycling inputs.



b_{11} b_{21} b_{31} b_{41}
b_{12} b_{22} b_{32} b_{42}
b_{13} b_{23} b_{33} b_{43}
b_{14} b_{24} b_{34} b_{44}
a_{11} a_{12} a_{13} a_{14}
a_{21} a_{22} a_{23} a_{24}
a_{31} a_{32} a_{33} a_{34}
a_{41} a_{42} a_{43} a_{44}
P_{11} P_{12} P_{13} P_{14}
P_{21} P_{22} P_{23} P_{24}
P_{31} P_{32} P_{33} P_{34}
P_{41} P_{42} P_{43} P_{44}

Figure 1: 4×4 Systolic Array [1]

Though CPUs are brilliantly flexible as general purpose machines, they are restricted when performing arithmetic instructions in parallel. MM becomes increasingly arduous with larger matrices when computing sequentially. CPUs become forced to perform $O(n^3)$ sequential operations for square matrices of width n. Advanced algorithms have been able to achieve the same in $O(n^{2.37188})$ time [2], but that seems to be the lower limit as of now. Another method is to use a Graphics Processing Unit (GPU) for computing hundreds or even thousands of arithmetic instructions in parallel, as opposed to the conventional CPU. However the main problem then becomes feeding several thousand ALUs inside the GPU with limited memory bandwidth. By directly arranging hardware in a systolic array structure, there emerges true parallel processing. Rather than wait for each single operation to consume an entire instruction cycle (or a large fraction with pipelining), the TPU can simultaneously process an immense amount of calculations despite limited memory bandwidth. Though this hardware is limited to the specific application of MM, our TPU drastically reduces the time spent on what is a predictable and standardized set of operations.

Matrices containing large amounts of zeros, or sparse matrices, are also a concern in machine learning as writing zeros as if it were data increases redundant computations while still requiring the same excessive bandwidth. Techniques to reformat the matrix, such as Nvidia's Blocked Ellpack format, can reduce such memory bandwidth and unnecessary computations by omitting zero-blocks.

ASICs (Application Specific Integrated Circuits) can serve as custom hardware and present the highest performance and lowest power consumption, however, fabricating new chips is time consuming and expensive. Only companies like Google and Nvidia can build these custom designs compared to a small research team due to the initial costs and development time. It may also be unnecessary for small applications to need a fully custom circuit board. Google uses an ASIC design for their TPUs but they also produce TPUs in bulk for their data centers. ASICs cannot be modified after manufacturing as their design is fixed in hardware. On the other hand, FPGAs are the preferred method for developing and prototyping custom processors as logic and memory blocks allow the chip to be endlessly reprogrammed for different applications.

III. Related Work

Google designed the first TPU in 2016. It is centered around a 256×256 8-bit ALU multiply unit which makes use of a systolic algorithm to reduce the expected amount of spent operation cycles, from 256^3 = 33,554,432 to only 256 x 2 = 512. This great reduction in cycles (with the same relative amount of operations) is thanks to the inherent parallelism of the systolic Matrix Multiplier Unit (MXU).

Google offers a Cloud TPU service which allows remote users to calculate MM for their desired ML needs. Their latest unit, the TPU v4, is a chip that contains two “Tensor Cores" – each with four MXUs, a vector unit for general computations, and a scalar unit for control and maintenance operations. Each MXU is capable of performing 16K multiply-accumulate (MAC) operations per clock cycle. The final accumulation is in FP32 format. Each TPU v4 pod contains 4096 TPU chips. By this given data, we can extrapolate that each TPU v4 pod offered by Google is capable of a stunning 524,288,000 operations per clock cycle.

Unlike the refined structure of Google’s MXU surrounding their systolic array, our project is limited due to the supporting resources available on our FPGA. While Google can arbitrarily change limiting factors such as bandwidth, buffer size, and area, the circuitry of the FPGA is fixed and ill suited to handle extremely fast and large amounts of data. As our design is set to synthesize on decade-old FPGA devices, ours would be more appropriate for dedicated small scale operations, and will work without an internet connection.

IV. Approach

A. Software

When designing the program to run on the Hard Processor System (HPS) as shown in Fig. 2. our team identified three core functions to provide to the TPU and to the user: To format and write the input matrices to the SRAM, to read the resulting matrix out of SRAM, and to provide a reliable interface to the user.

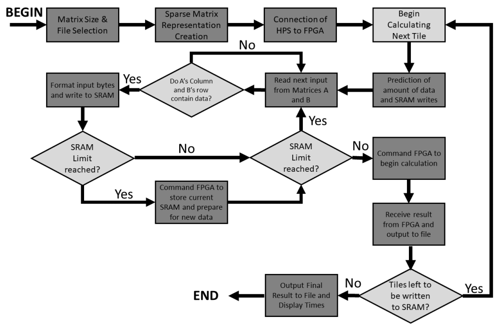

BEGIN
Matrix Size & File Selection
Sparse Matrix Representation Creation
Connection of HPS to FPGA
Begin Calculating Next Tile
No
Format input bytes and write to SRAM
Yes
Do A's Column and B's row contain data?
Read next input from Matrices A and B
Prediction of amount of data and SRAM writes
Yes
SRAM Limit reached?
No
SRAM Limit reached?
No
Command FPGA to begin calculation
Yes
Command FPGA to store current SRAM and prepare for new data
Receive result from FPGA and output to file
END
Output Final Result to File and Display Times
No
Tiles left to be written to SRAM?
Yes

Figure 2: Software Flowchart

When planning how our data should be formatted within SRAM, we decided to represent the 2D matrix in column major order in memory. However, since the FPGA control Finite State Machine (FSM) requires inputs to the systolic array in a specific order, we decided it’d be easier to configure in software. In order to reduce unnecessary memory complexity and improve access time for the software, we initially decided to memory map both input files directly into SRAM. Further formatting avoids the use of data structures by directly addressing said memory using pointers, and accessing specific matrix elements, by performing arithmetic to calculate the proper offset from the base address. However, as we began operating on exceptionally large matrices, initializing input arrays with millions of elements on the stack would cause segmentation faults within the program. Therefore, we resorted to initializing our input matrices in the software as array structures on the heap, and outputting directly to a file instead of a matrix structure. The output matrix file must then be reinterpreted by an external program capable of storing it. Indexing the arrays makes use of the same logic as with memory mapping simply without the base address offset, and slightly improved the overall write time of the program.

In order to perform the multiplication of matrices larger than 8x8 (the instantiated hardware on the FPGA), we implemented a tiling algorithm. By dividing larger matrices into submatrices of 8x8, we can effectively divide the task and sequentially feed the set of tiles to our TPU in column major order to produce the whole output. Because each output is independent of another, the only determining factor in calculating any size matrix is the input size, as each tile requires the corresponding row and column of input tiles. For example: tile C0,0 in Fig. 3 requires the first 8x16 row from matrix A, and the first 16x8 column of matrix B. While this algorithm allows us to theoretically break down infinitely sized matrices on finite hardware, the bounding factors become apparent further on.

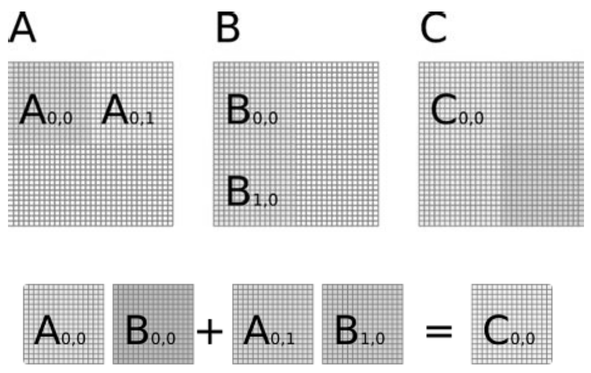


Figure 3: Tiling Method [3]

Because rearranging input matrices within the FPGA would be laborious and convoluted, we opted to format the data in software before sending the inputs to the FPGA. This is achieved through an FSM in our C program and writes data in the format shown in Fig. 4 after operating for 32 cycles. Because the lightweight bridge we are transmitting across is only 32 bits wide, we write a single byte each cycle, alternating between a double word of input matrix A and a double word of input matrix B. These addresses are relative to the inputs for each tile the offsets can be calculated using (1) where row and column are of the respective tile, and M is the overall matrix width.

$$tile\,offset = 8(column\cdot M + row) \quad (1)$$

If we were to write the inputs for the second tile in column major order, the matrix B inputs would have an offset of eight. Furthermore, if we were to write the tile directly to the right of the starting tile, the row offset would instead be 8·M. The program reads each byte from the input matrix using these offsets and relative addresses and is each accumulated onto a four byte word. To move down one within a tile, add one, and to move right add M. To move down one tile, add 8, and to move right a tile, add 8·M. When the fourth byte has been accumulated, it is written to the next available space in SRAM. Because the input data for matrices larger than 1024×1024 exceeds the maximum capacity for SRAM, the program will run an algorithm to detect when SRAM needs to be refreshed. Before any data is written to the SRAM, we predict the total amount of data we will write for the input. We then write to SRAM up to 1,024 times and if there is any data left to be sent, the HPS commands the FPGA to prematurely begin processing the input data. Once the FPGA has used all the data, we continue to write the remaining data. After a complete tile's worth of inputs is written to SRAM, the HPS commands the FPGA to complete the final calculation and write the complete outputs to SRAM. Because the result will never exceed one tile's worth of outputs, each four bytes wide, we do not have to refresh the SRAM in this instance.

A03	A02	A01	A00
A07	A06	A05	A04
B24	B16	B08	B00
B56	B48	B40	B32
A11	A10	A09	A08
A15	A14	A13	A12
B25	B17	B09	B01
B57	B49	B41	B33
A19	A18	A17	A16
A23	A22	A21	A20
B26	B18	B10	B02
B58	B50	B42	B34
A27	A26	A25	A24
A31	A30	A29	A28
B27	B19	B11	B03
B59	B51	B43	B35
A35	A34	A33	A32
A39	A38	A37	A36
B28	B20	B12	B04
B60	B52	B44	B36
A43	A42	A41	A40
A47	A46	A45	A44
B29	B21	B13	B05
B61	B53	B45	B37
A51	A50	A49	A48
A55	A54	A53	A52
B30	B22	B14	B06
B62	B54	B46	B38
A59	A58	A57	A56
A63	A62	A61	A60
B31	B23	B15	B07
B63	B55	B47	B39

Figure 4: Data Layout of 8x8 Input Matrices in SRAM

The reason we have to predict the amount of data to be written is because that amount of data depends on the sparsity of the input matrices. Because the number of writes is dependent on the variable sparsity, and the TPU requires exactly 1,024 inputs for each intermediate operation, we must have this additional check to prevent SRAM overflow and ensure accuracy of the result. Sparse matrix multiplication is a form of matrix multiplication accelerated by ignoring empty inputs. Since an input of zero results in the multiplication and addition also being zero, we can omit the value and avoid wasting cycles writing the zero to SRAM. Because the TPU accepts input data at the resolution of a single 1×8 row and 8×1 column, this is the maximum resolution of our sparse detection. Therefore, to avoid these negligible writes to SRAM, the software scans rows of matrix A and the columns of matrix B, recording for each if there is any data or if it is completely empty. These results are stored in separate matrices representing the rows of A in row major order and columns of B in column major order respectively. Before formatting and writing any pair of row and column from A and B, the representation matrices are referenced to see if both have data. If so, the inputs are formatted as normal, but if at least one is empty, the resulting MAC would also be zero, allowing us to disregard the write, and save not only formatting time, but also TPU time, as less inputs requires less TPU cycles.

The second function of the software is to read the output result matrix from the FPGA out of SRAM. The FPGA writes back the output matrix tile in column major order at the SRAM base address for the program to read from. Since it is stored in column major order and printed out to the terminal by row, we must increment by 8 values, or by 32 bytes each read, and after the eighth read, return to the base and increment by just four bytes. Due to the aforementioned memory allocation issues, we output each consecutive tile to an output file, with each tile in column major order to be interpreted by a separate program. We have written this program in MATLAB in order to verify the accuracy of our multiplier at every size.

We were able to time the operation of the "HPS format and write" and "FPGA read, calculate, and write" using the time.h C library. Furthermore, we surround the code segments of interest with the start and end timer functions and accumulate a total time after each tile to be displayed upon the completion of the final tile. Despite some additional complexity in our formatting and testing phases, these values serve as benchmarks for our TPU and validate the performance in hardware over software.

One final part of our program worth mentioning are the quality of life features implemented on the interface to assist in executing and testing the TPU. Not only does the interface allow you to select new files to input, but it also automatically remembers past file names to input if the user wishes too. Additionally, the interface allows the user to generate random input matrices at configurable sparsity levels, printing them to the terminal as well as outputting them to their own files similar to the output file.

HPS-to-FPGA Communication

The HPS-to-FPGA communication involves sending and receiving a 32 bit signal to the TPU's FSM. The first three bits of the sending signal, Control_to_FPGA, are one-hot encoded to reset, start, and matrix size as shown in Fig. 5. Bits 31 to 27 represent the number of times the matrix size is larger than 1024^2. Bits 17 to 4 is the matrix size that the TPU will compute. While the data memory maps to SRAM, the control bus sends a reset signal to the TPU followed by the start signal once the data is fully mapped to the SRAM and the size signal with the number of double words. The receiving signal, "Control_to_HPS," currently receives a 1 bit signal flagging done from the TPU similarly to how reset is the first bit of the sending control signal. This line is also used during the acknowledgment protocol when refreshing the SRAM with more data.

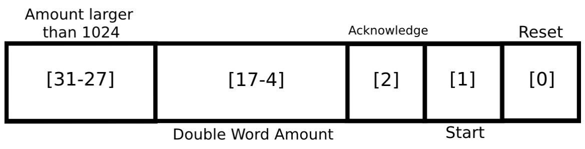


Figure 5: 32-bit Control Bus

Once the TPU hits the bottom of the SRAM, the HPS sends more in the same pattern to fully use the available space in SRAM in each send. The Tensor Core holds the partially computed tiles in its MACs when waiting for more data and in the buffer once computation is finished. The TPU writes the completed output matrix back to SRAM for the HPS to read.

B. Hardware

We used Intel® Quartus® Prime for hardware design and the DE1-SoC configurations, and ModelSim-Intel® for simulation verification. Our TPU uses a FSM, Fig. 7, to read data from the SRAM and put it into the Tensor Core. It also controls the logic on when to start computation and when to read from the buffer and write to the SRAM. The TPU is connected to a control bus that the FSM can interact with. Wait request, acknowledge, and done signals are asserted within the FSM for the protocols when stalling within reads, running out of data, and finishing computation. The Tensor Core is made of four parts, internal timers, systolic setup, 64 MACs, and a buffer for results, Fig. 6.

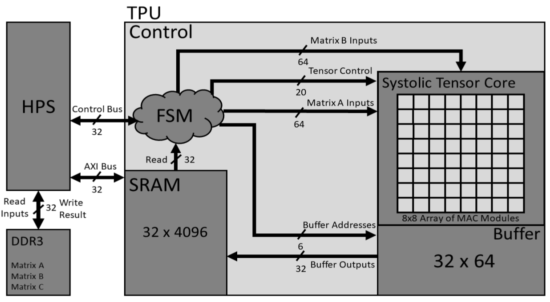


Figure 6: Embedded System Diagram

The start control will take the TPU out of its idle/reset state and begin reading the data and starting the Tensor Core's internal timer. The reset control stops computation, clears MACs by injecting zeros, and resets all values to their initial states. The size signal determines the logic and timers for the Tensor Core, which allows for larger size matrix compatibility. The wait request is used to halt the Tensor Core until more data can be inputted and save current results in the MAC. The done flag is asserted high when all values have been transferred from the buffer into the SRAM. During multiple operations, reset will need to be asserted after the done flag is received.

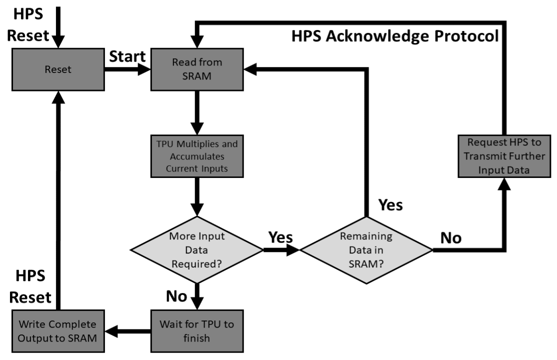

HPS Reset
Reset
Start
Read from SRAM
HPS Acknowledge Protocol
TPU Multiplies and Accumulates Current Inputs
Request HPS to Transmit Further Input Data
Yes
More Input Data Required?
Yes
Remaining Data in SRAM?
No
HPS Reset
No
Write Complete Output to SRAM
Wait for TPU to finish

Figure 7: Hardware FSM

The internal timer is based on the matrix size of the matrix you are computing. The design is of an 8×8 systolic nature, where each additional double word takes an additional cycle to compute. A typical 8×8 matrix would take 23 cycles for data to flow from the first MAC 0,0 to the last MAC 7,7. A 8×8 tile of a 16×16 matrix computed on our hardware would be 31 cycles. The results are stored one after another after 8·n + diagonal. MAC 0,0 completes at 8·n cycles and MAC 7,7 at 8·n + 15. After all the values are computed, they are stored in the buffer in column major order. Each MAC uses an eight bit multiplier and a 32-bit adder, Fig. 8. The input sizes are eight bits and due to the future uses of our design we had to accommodate large enough accumulated values up to 32 bits. The adders are our biggest clock constraint and limit our frequency to 100MHz.

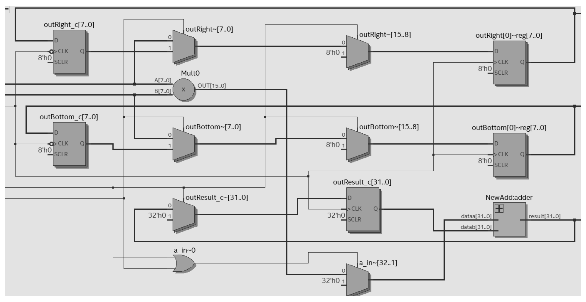


Figure 8: MAC

Due to limitations on the SRAM, it takes three cycles from when the address is changed to see the correct value on the read line. This can be staggered on top of the last read cycle making subsequent reads one less cycle, i.e., 3 + 2 + 2 + … . However, writing to the SRAM only takes one cycle after the first cycle changes the address. i.e., 2 + 1 + 1 + … . Total cycles are calculated using (2).

$$(65 + 8 + (x \cdot 8)) \cdot y^2/64 \qquad (2)$$

65 is the amount of cycles writing to SRAM. $16 \cdot y$ out of $16 \cdot y + 8$ cycles of the TPU's computation happens during the reads, so only 8 is additional. $x \cdot 8$ is the additional reads for matrix size x. The $y^2/64$ determines how many tiles need to be computed by multiplying matrix size divided by eight, y, and dividing by tile size, 8×8. A typical 8x8 matrix computation would take 137 cycles shown in Fig. 9.

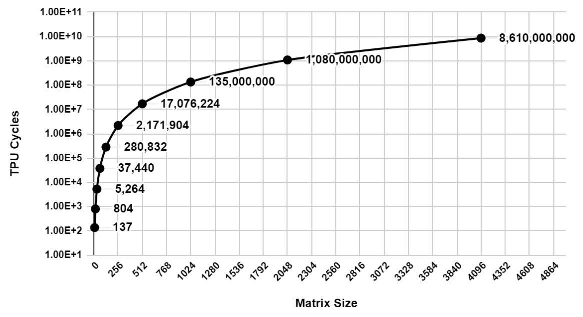


Figure 9: TPU Cycles vs. Matrix Size

The internal timers have a resolution of a double word instead of a tile, this gives an 8 times higher computation accuracy. It allows any size greater than 8×8 and is not limited to multiples of eight. There is a limit on timer complexity and clock line length because increasing the upper limit will reduce overall clock speed. Extra states were added in order to allow for multiple full SRAM reads. If more than 4096 words are necessary, and the TPU has read all values, a protocol is in place to handle up to 4096×4096, four full SRAM reads. This is done by using the control bus as an acknowledgment line between the HPS and FPGA. They will respond on that line once their actions have been completed, and the other will continue to their next state.

V. Results

A. Dense CPU vs. Dense CPU+TPU

We consider our product to be successful should our combined CPU and TPU system be able to calculate matrices faster than a typical CPU approach on the same hardware. We measured our CPU+TPU combination to be consistently faster at any matrix size above and including widths of 256. Measurements indicate an average difference of 0.07 seconds in which our system beats typical CPU operation (at size 256^2). In Fig. 10, our maximum measured matrix size of 4096^2 yielded 4,792 seconds CPU vs. 1,165 seconds CPU+TPU, a difference of 3,637s. This difference between methods improved by a factor of 50,000 as the input size grew. We can expect much greater time savings at larger matrices. At sizes lower than some threshold between 128^2 and 256^2, we've determined the CPU to be much more efficient than the combined system. At any reasonably demanding workload, we can fairly claim our combined system to be the better option compared to a standalone CPU.

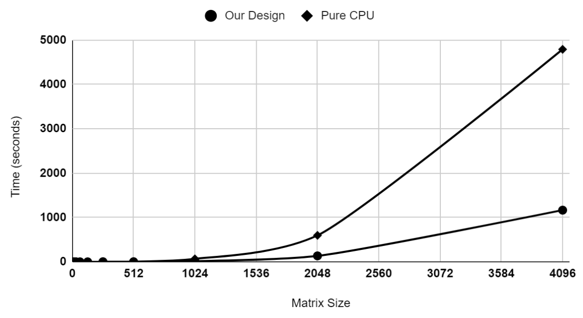


Figure 10: Our Design vs. Pure CPU over Matrix Size

B. CPU+TPU at Varying Sparsities

Building upon our existing improvements, we were able to format the CPU to omit empty columns and rows (containing all zero values). This drastically reduces computation time by levels of sparsity, starting from 50%, as seen in Fig. 11, 12. As the sparsity of a row or column depended on the complete sparsity of the structure (product of eight independent probabilities), there is evidently little ability to omit data until at least half of the entire matrix was empty. The probability of emptiness of the substructure increases exponentially, as the greater sparsity increases by percentage. At 99% sparsity, we were able to reduce memory accesses by 99.4% for sizes of 4096^2. As memory accesses caused our greatest bottleneck in time spent due to the limited bridge resources, we were consequently able to reduce the total processing time by a similar magnitude, seen in Fig. 13. At 99% sparsity, our HPS time was decreased to 3.5% the time of dense calculation, and TPU time was decreased to 1.2% the time of dense calculation at 4096^2.

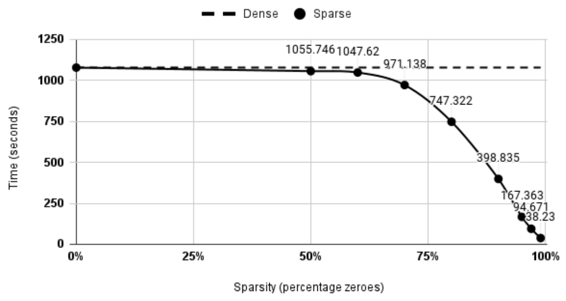


Figure 11: CPU Time vs. Sparsity

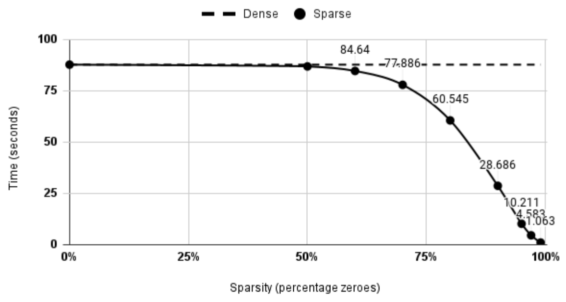


Figure 12: TPU Time vs. Sparsity

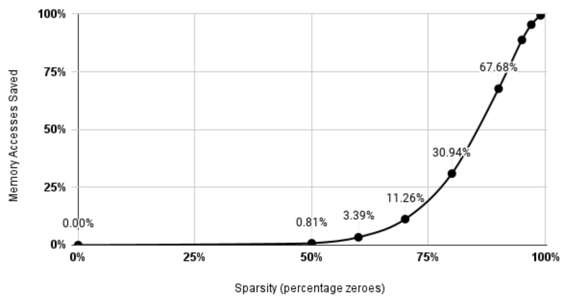


Figure 13: Memory Accesses Saved due to Sparsity

VI. Discussion

The results verify our expectations by producing a 4X improvement in computation time. Adding the sparsity algorithm to our architecture alleviates the memory bandwidth problem for large matrices with small amounts of useful data.

When developing the software, the greatest initial challenge was interfacing between the HPS and the FPGA, as we found very little documentation on the subject. Intel® had a separate outdated program called AMP but we wanted to create an embedded system that ran separately from a PC application. We referenced Cornell's method of memory mapping regions of memory in the HPS corresponding to the same regions of memory mapped in hardware using platform designer through Linux instead of AMP. After this setup was established, transmitting control signals and data through the bridge was as simple as reading and writing values to a specific base address.

Another ever present challenge facing the software was navigating each 2D matrix within a 1D memory representation. The data formatting, tiling algorithm, sparse prediction, input printing, output printing, and output testing all required different methods of navigating each matrix. Devising each new offset calculation to move between positions demanded our full attention and resources.

As mentioned before, the limited resources on our FPGA were also a challenge to work around, as the limited bridge width of four bytes and SRAM size of 16 KB greatly bottlenecked the overall design. While the latter was alleviated using the SRAM refresh algorithm, we were unable to algorithmically minimize the impact of the former outside of increasing the clock frequency.

The hardest challenges when designing the hardware were the timings of the TPU. There had to be multiple connections working together one after the other, and each result and input had to line up for each computation. Functionalities necessary to work with the HPS and SRAM required more conditional controls. A potential challenge will be the limited size of the SRAM, which is 16 KB. Doing a 1024×1024 matrix would fit, but anything greater requires extra communication protocols. Ideally, we could send some data, compute, then receive the results without stalling the system. Extensive FSM logic, C code, and testing was necessary to produce a reliable embedded system. We originally wanted to use a 16×16 systolic array, but resource limitations hindered our goal. We originally had an eight double word resolution in the Tensor Core timers, but when adding sparsity compatibility, the timers had to be expanded to compute within a single double word. When accommodating larger matrix sizes, we had to take into account the upper limit of storage size. Our answer was to use an efficient 32 bit adder design which is sufficient for large accumulations. Using efficient logic, we could keep our max clock frequency above 100 MHz, the effective double data rate from the base 50 MHz clock.

VII. Future Work

The initial design involved building the TPU around the SRAM to send and receive data from the HPS. For matrices larger than 1024×1024, the SRAM size (16 KB) became the bottleneck, since the design needs to refresh the SRAM with new data. Implementing FIFOs instead of using SRAM would lower time and memory complexity for larger matrices due to the simpler formatting requirements. However, FIFOs require different timing parameters compared to SRAM which leads to a fundamentally different FSM setup. The FPGA's 64 MB SDRAM is another alternative for data buffers. By manipulating the SDRAM controller in the FPGA, larger data sets can be stored in the SDRAM. This SDRAM is also accessible by the A9 processor using word, half word, or byte operations. A Direct Memory Access (DMA) controller would be needed to use the SDRAM instead of the SRAM for HPS to FPGA communication. Unfortunately, we couldn't find the appropriate resources to understand DMA operation in time for the project deadline. Eventually, the SDRAM would have introduced the same refresh issue as the SRAM at larger data sizes.

Optimizations for the hardware design would include pipelining the HPS with the TPU to concurrently compute while data is being written/read. The key component here would require figuring out the timing between the HPS and TPU. As the TPU is significantly speedier than the HPS, we would like to avoid stalling when the TPU is waiting for new data. FIFOs would mitigate this problem of timing as the TPU would continuously process data as long as the FIFO is not empty. Further optimizations would include reducing the overall FPGA logic and interconnects. With a bigger FPGA, more Tensor Cores can be added to compute data faster in parallel for increasing matrix sizes.

Further optimization of the software may include multi-threading the formatting process. As each write to SRAM was the sum of two independent processes, we could have made use of the dual cores of the ARM Cortex-A9 processor and used the pthread library to compute them in parallel, potentially halving the format time.

Lastly, we could improve the TPU configurability by wrapping the existing program inside a single function call with discrete parameters and a return value. This would allow users to effectively integrate our TPU within their existing codebases, saving development resources and reducing difficulties in importing our hardware.

Acknowledgment

We would like to acknowledge our EEC 181 Professor, Dr. Venkatesh Akella and Teaching Assistant Thomas Abbott for their full support over the six months of the course. They allowed us to meet our milestones and pushed us as far as possible. We would also like to acknowledge Bruce Land from Cornell University for his detailed documentation on the DE1-SoC [4]. His examples were crucial to get our design off the ground and past the embedded communication stage.

References

[1] Dr. S. Ziavras, New Jersey Institute of Technology, “ECE 459 - Advanced Computer System Design Laboratory,” http://ecelabs.njit.edu/ece459/, 2015.

[2] J. Alman and V. Vassilevska Williams, “A Refined Laser Method and Faster Matrix Multiplication,” Oct 2020, doi: arXiv:2010.05846

[3] L. Araujo-Santos, *Parallel* “Programming, Bigger Matrix Multiply problem,” https://leonardoaraujosantos.gitbook.io/opencl/, 2023.

[4] B. R. Land, Cornell University, “DE1-SoC: ARM HPS and FPGA Addresses and Communication, Cornell ece5760,” https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherials/FPGA_addr_index.html, 2019.