The Thesis Committee for Sangramsinh Kate
Certifies that this is the approved version of the following thesis:

A Tensor Processing Unit Design for FPGA Benchmarking

APPROVED BY
SUPERVISING COMMITTEE:

Prof. Dr. Lizy Kurian John, Supervisor

Prof. Dr. Earl Swartzlander

A Tensor Processing Unit Design for FPGA Benchmarking

by

Sangramsinh Kate

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

Spring 2021

Dedication

To my family and friends for their support.

Acknowledgements

I would like to thank Aman Arora for his constant support and help throughout the research activities. My thanks to professor Lizy John for the supervision and guidance. I would also like to thank professor Earl Swartzlander for being on my supervision committee and my gratitude to him and Professor John for their time and many suggestions during the submission of this master's thesis.

Abstract

A Tensor Processing Unit Design for FPGA Benchmarking

Sangramsinh Kate, MSE

The University of Texas at Austin, 2021

Supervisor: Lizy Kurian John

The recent exposure of use of FPGAs for deep learning applications have opened a wide range of use cases for FPGAs. The scalability and programmability of FPGAs are essential to update the hardware to encompass the state-of-the-art network architectures with special purpose units to accelerate the computation. However, these accelerator designs vary according to different design structures and properties. It is essential to understand the efficient FPGA architecture for a specific type of workload. This thesis provides an academic version of Google's tensor processing unit (TPU v2) design as a benchmark for FPGA architecture evaluation. The thesis provides a reference microarchitecture for TPU v2 core design. The thesis uses Verilog-to-Routing (VTR) tool which is a widely used open-source academic FPGA architecture analysis and research tool to perform the analysis of benchmark on different types of FPGA architecture.

Table of Contents

List of Tables

List of Figures

INTRODUCTION

Deep learning (DL) has enabled the shift in the usage of machine learning techniques in numerous applications. Contemporary deep learning frameworks such as Google TensorFlow, PyTorch, Caffe, Keras, Microsoft Cognitive toolkit have enabled various deep learning architectures suitable for different sets of applications. The availability of a massive amount of data has enabled this paradigm shift in learning algorithms, focusing on finding meaningful insights from the data to the problem of classification, recognition, and prediction. Deep learning has extensive use-cases in computer vision, Image processing, Speech recognition, and analysis. While deep learning applications are vast, they are computationally complex and require a unique set of hardware for higher performance. While using FPGAs for deep learning is prevalent, a recent trend is trying to optimize FPGA architecture for deep learning workloads [2].

The current design tools that follow a software-like approach to program an FPGA have made them a preferable option for deep learning hardware accelerators for their flexible hardware configuration and better performance per unit cost [2]. The researchers in deep learning can use these FPGA design tools to use high-level design descriptions to program an FPGA. While a traditional FPGA constitutes flip-flops for sequential logic, Block-RAM for memory requirements, and stores combinatorial logic in the form of lookup tables (LUTs), more recent FPGA architectures include special-purpose hardware such as DL optimized fabrics [3], DL accelerators [4][5].

To enhance the use-case of FPGA for DL applications, researchers are exploring different ways of optimizing FPGA architectures and CAD to achieve a better quality of result (QoR) for DL applications. The FPGA architectures need to be tested against a wide range of benchmark designs to understand and improve the QoR for an architecture. The FPGA benchmarks, such as the Microelectronics Centre of North Carolina (MCNC) benchmark, VTR benchmark, and Titan framework, are very well known in academia. These benchmarks include a range of designs that

contains small to large-size benchmarks. However, these benchmarks are not the most optimal, and there is a need to create more complex benchmarks [5]. [7] explains the lack of deep learning specific benchmark suites, which are essential for the FPGA analysis for deep learning workloads.

This thesis work addresses this problem by providing a deep learning specific benchmark for FPGA architecture research. The benchmark is a model of the Google's Tensor Processing Unit version 2 core. The thesis establishes the groundwork and provides the design details to create an academic DL accelerator benchmark.

BACKGROUND AND RELATED WORK

This section discusses the background work related to the DL accelerators, neural networks, and vector processor cores that the thesis uses as a basis of TPU v2 core design. The intention behind vector processor core design is that the open-source Verilog code can be utilized efficiently for overall design process.

Neural Networks

Neural networks are one of the promising types of a supervised learning algorithms. The ability of the neural network to learn the nonlinearity of the data efficiently helps neural networks to learn complex training data and predict with higher accuracy. A classical neural network comprises nodes, each of which has specific inputs and an output. The input to each neuron is called activation. A neuron in a neural network uses a weighted sum of the activations to generate an output. Each neuron uses a type of nonlinear function to generate the output. Neurons in neural networks are connected to form multiple layers such that the structure of each neuron resembles the structure of a neuron in a brain.

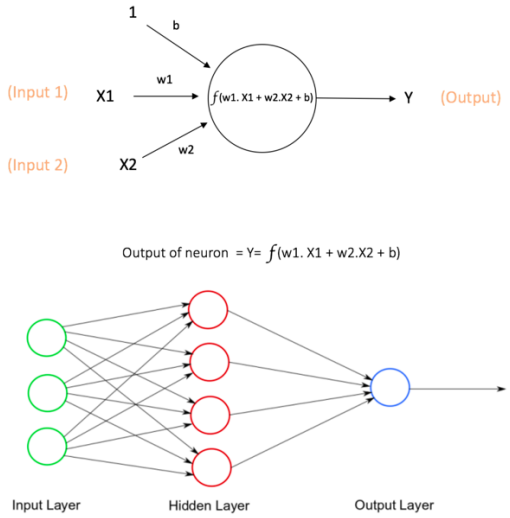

Figure 1: Neuron and Neural Network

Each layer can have multiple neurons connected parallel to the previous layer's neurons and generate activation for the next layer's neurons. The input layer is the first layer of any neural network. The last layer of the neural network is called the output layer, which generates the output. In between the input and output layers, a neural network can have multiple layers, which are called hidden layers.

Deep neural network extends this neural network class by adding more types of computational layers such as convolutional layers, pooling layers, activation layers, softmax layer, etc. Each of these layers plays a crucial role in terms of data compression and feature extractions. The sampled features of the image are forwarded to the fully connected layers for classification and identification problems. Figure 2 shows an example of a Deep neural network.

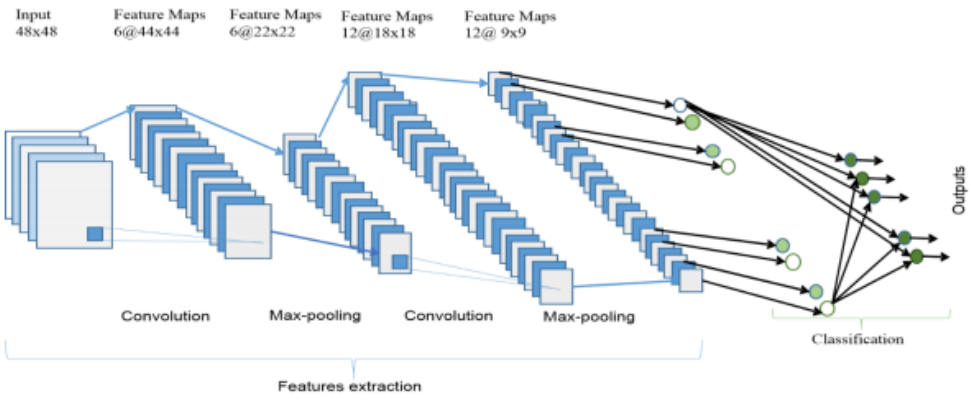

Figure 2: An Example of Deep Neural Network (Taken from [8])

Figure 2 shows different layers and their relationship in a DL program. The figure is taken from [8]. Here are some of the joint operations performed in the deep neural network (DNN):

CONVOLUTION

A convolution operation includes the dot product operation of a filter over the input with a fixed stride. The convolution operation can be thought of as a reduction operation as it reduces the size of the input matrix by the factor of filter size. This operation helps to reduce down the complexity of DNN as the reduction in size results in smaller, fully connected nodes, which

computationally very costly. Furthermore, the convolution operation is a multiplication operation, the rearrangement of inputs converts a convolution problem into a matrix multiplication problem.

MATRIX MULTIPLICATION

Matrix multiplication is the heart of most complex neural networks. It multiples the input activations with the weights to produce the output of a neuron in a fully connected layer. This operation is the most important part of DNN as the trained weights encode the complex functional relationship information between input and output

POOLING

The pooling operations is a simple reduction operation. The most commonly used pooling operations include MaxPool and AvgPool. A MaxPool operation reduces the input features to include only maximum value features by sliding the non-overlapping filter over the input matrix. Thus, pooling is the easiest way to reduce a matrix.

ACTIVATION

The activation layer adds nonlinearity to the functioning of the neural network. The most commonly used activation functions include rectilinear activation function (ReLU), sigmoid activation function (Sig), hyperbolic tan function (Tanh). The output of a neuron from a layer acts as an input to the nonlinear activation function. The activation layer provides the input for the next layer in the network.

SOFTMAX

The softmax layer is used to generate the final output of a DNN. This layer generates the probabilistic distribution of each output over a given network. This probabilistic answer is always in the range of 0 to 1.

DNNs use multiple structures of the combination of these layers. Table 1 compares a few typical examples of the most widely used neural network for their performance. It can be observed how the performance of a network can be different by changing the architecture of a DNN [8]. The table shows how different deep learning architectures vary in terms of number of layers, number

of nodes per layer, filter dimension, etc. It also discusses the total number of operations required in each type of network and their performance. The table is taken from [8].

Methods	**LeNet-5[48]**	**AlexNet [7]**	**OverFeat (fast)[8]**	**VGG-16[9]**	**GoogLeNet [10]**	**ResNet-50(v1)[11]**
Top-5 errors	n/a	16.4	14.2	7.4	6.7	5.3
Input size	28x28	227x227	231x231	224x224	224x224	224x224
Number of Conv Layers	2	5	5	16	21	50
Filter Size	5	3,5,11	3,7	3	1,3,5,7	1,3,7
Number of Feature Maps	1,6	3-256	3-1024	3-512	3-1024	3-1024
Stride	1	1,4	1,4	1	1,2	1,2
Number of Weights	26k	2.3M	16M	14.7M	6.0M	23.5M
Number of MACs	1.9M	666M	2.67G	15.3G	1.43G	3.86G
Number of FC layers	2	3	3	3	1	1
Number of Weights	406k	58.6M	130M	124M	1M	1M
Number of MACs	405k	58.6M	130M	124M	1M	1M
Total Weights	431k	61M	146M	138M	7M	25.5M
Total MACs	2.3M	724M	2.8G	15.5G	1.43G	3.9G

Table 1: A comparison of Different Deep learning Architectures and their performance.

Deep Neural Network Accelerators

The majority of the workload in a DL workload is involved with different types of operations on input matrices. Convolution and fully connected layers predominantly use matrix multiplication operation. A matrix multiplication application involves a lot of data reuse as each row gets multiplied with all columns. Moreover, the row-column pair multiplication is agnostic to each other and can be executed in parallel. In general-purpose hardware, due to the limited resources and sequential execution-style, such workload experiences a performance loss as they cannot extract enough parallelism. Due to the prevalence of deep learning applications, it is essential to improve the performance of these applications. Although traditionally, use of general-purpose GPU for DL applications is common, some special-purpose accelerator designs have been proposed and developed both in academia and industry. This thesis provides a brief overview of two industrial accelerators, Tensor processing unit version 1 (TPU v1) and Tensor processing unit version 2(TPU v2), to understand the design points behind these accelerators.

TENSOR PROCESSING UNIT V1

TPU v1 is Google's DL accelerator for inference. TPU v1 is an off-chip accelerator, which a host CPU handles to offload DL workload for inference. The capability of TPU to utilize the reuse of data and extract parallelism for matrix multiplication helps to achieve higher performance. The TPU v1 uses a systolic array for the matrix multiplication operation. The matrices multiplied in systolic arrays are tiled into small blocks of size optimally equal to or less than that of systolic array size. The systolic array outputs are stored in an accumulator for reuse. The TPU v1 loads the data from the DRAM and host interface to be stored in the local buffer. The weight metrics are directly pulled from the DRAM interface. The layer-wise computation starts by loading the data into the systolic arrays for the purpose of convolution or dense layer computation. The output of the systolic array is further used by the activation block to process the outputs by an activation function. TPU v1 also provides pooling and normalization layer function. The local buffer acts as temporary storage for passing the data from one convolutional/dense layer to another one. The overall flow is shown in Figure 3 below which is a block diagram of TPU v1 along with the data flow. The Figure 3 is taken from [9].

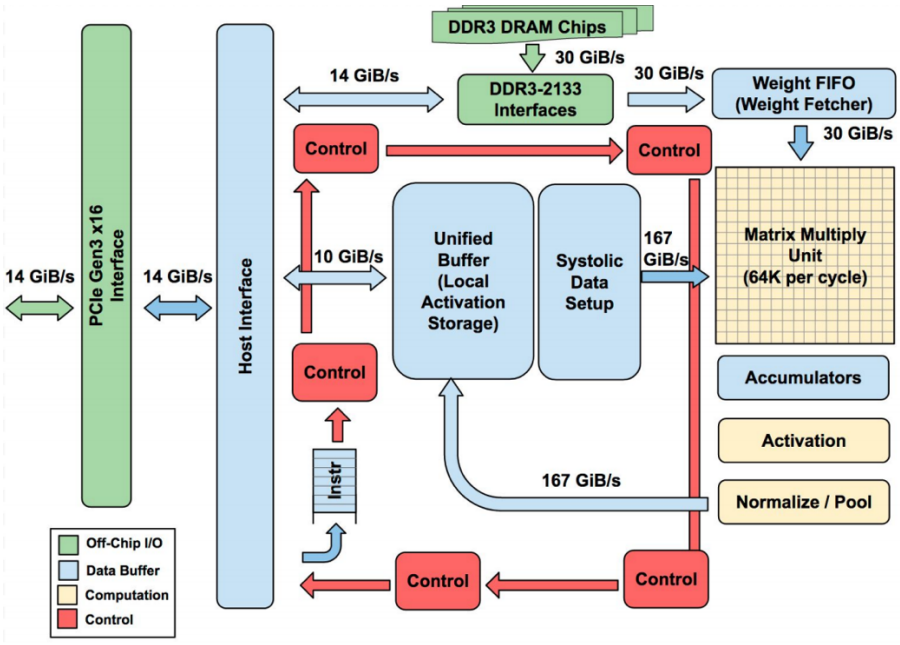

Figure 3: TPU v1 Microarchitecture details

Systolic Array

Systolic arrays are a 2-D mesh of many small processing elements connected to multiplying and accumulating operations. These processing elements pass the computed result or unmodified input to the adjacent processing element on each cycle [10]. This functionality of systolic arrays helps in utilizing more reuse as the data transfer happens only between adjacent processing elements [10],[11]. In the computational model for the systolic array pipeline, the data transfers between each PE from the top and left side of the systolic array. Thus, the output of the matrix multiplication is pushed down from the bottom of the systolic array over multiple clock cycles with one systolic row of each clock cycle.

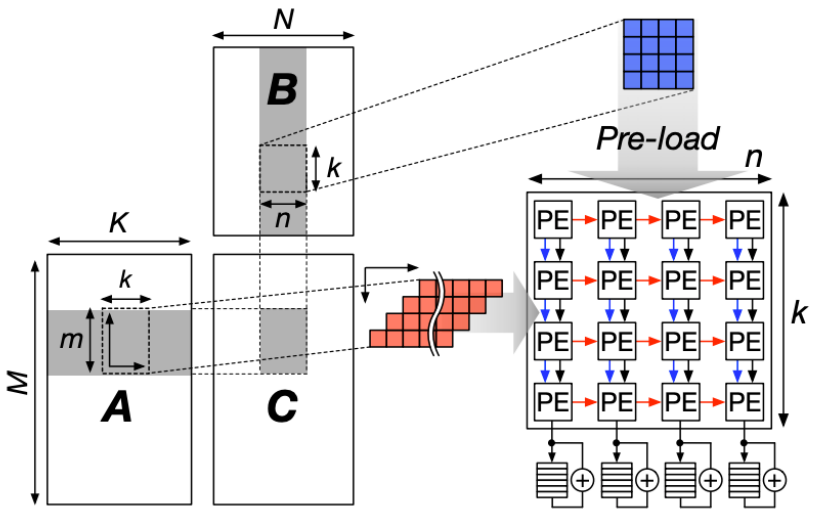

Figure 4: Matrix multiplication using Systolic array

Figure 4 is taken from [10]. Each PE in the systolic array computes the multiplication between input weight and activation and adds the result into a local accumulator. The accumulated results are passed down to the next PE.

TENSOR PROCESSING UNIT VERSION 2

Google improved their TPU v1 architecture to add more generalization into the architecture. TPU v2 contains a scalar unit call a core sequencer and a vector unit. In addition, each unit supports memories called scalar mem and vector mem. The overall architecture of TPU v2 is shown in figure 5.

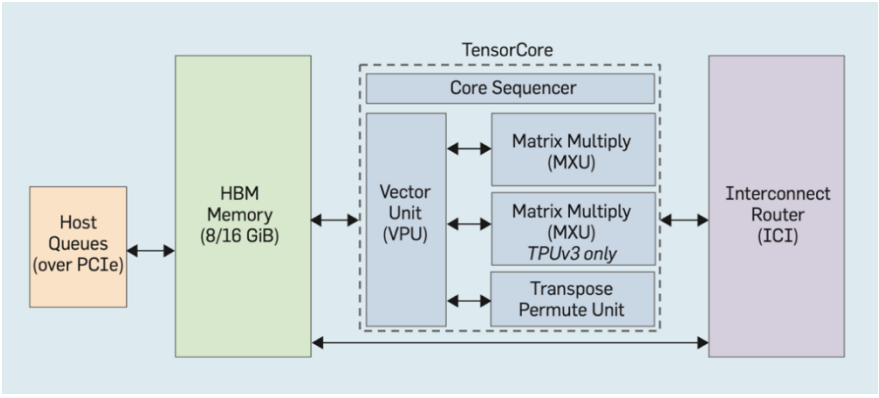

Figure 5: TPU v2 core block diagram (taken from [11])

The scalar unit is a simple processor without advancements like branch prediction, cache used for efficient single-threaded execution. The scalar unit uses scalar instructions for normal control operations to execute the program. The instructions for the scalar unit and the data associated with it are stored in scalar memory. The vector unit is the heart of the TPU v2, where most of the DL-related workload execution happens. Figure 7 describes the vector unit for a TPU v2 core. Figures 6 and 7 are taken from the videoblog on TPU v2 design [18].

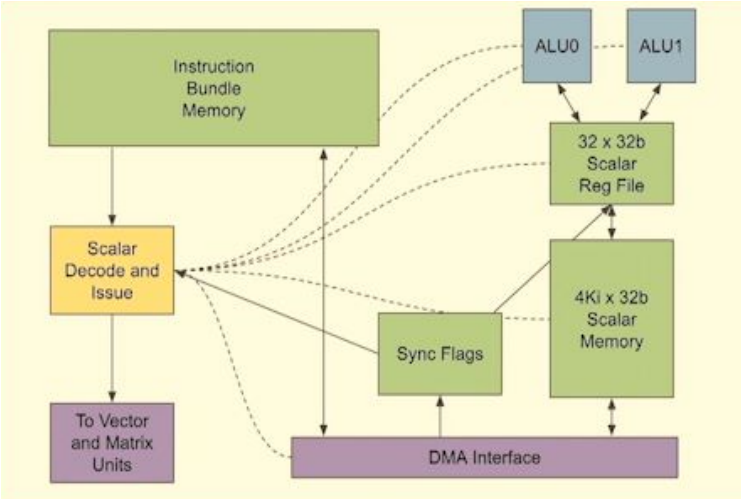

Figure 6: TPU v2 Scalar Unit block diagram

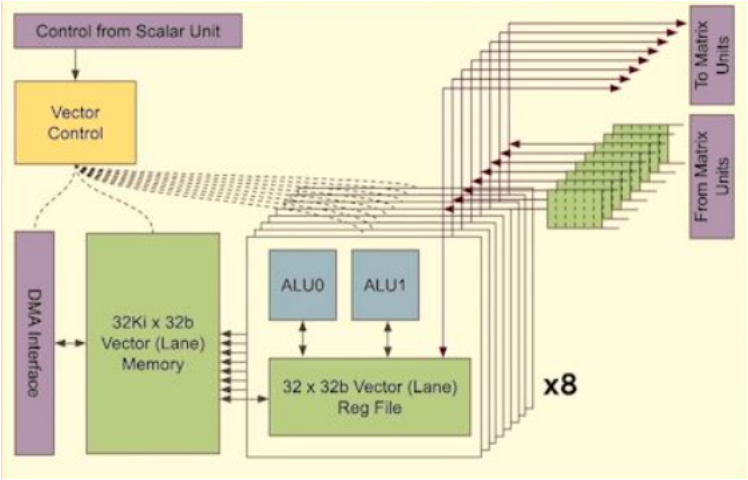

Figure 7: TPU v2 vector unit lane block diagram

The vector unit consists of vector ALUs, which are embedded into multiple lanes. Each of these lanes contains vector memory, DMA, multiple ALU blocks, and register files divided into sub-lanes. The vector unit is connected with a matrix multiplication unit. The matrix multiplication unit is a systolic array of size 128 x 128 elements. The systolic array takes input from the register files of all vector lanes and provides a matrix multiplication result shared among all lanes.

Along with the matrix multiply unit, TPU v2 core also have transpose, reduction, and permute (TRP) units. These functional elements are also shared among all vector lanes. These blocks are used for matrix transpose operation, matrix reduction operation, and matrix permutation operation, as their name suggests. The scalar unit, vector unit, and matrix multiply and TRP unit are together called a TPU core. In TPU v2, there are two cores in each node. These nodes are connected using an interconnect router, enabling the TPU v2 to share data among its different nodes. These TPU cores in each node are also connected with high bandwidth memory per core for faster data access.

The interconnect network allows each TPU to be connected in a mesh structure to enable data communication and breakdown of DL workload by sharing it into multiple nodes. TPU v2 supports VLIW architecture with software-managed memory. Each instruction in TPU v2 includes 322b format of 2 scalar instructions, four-vector instructions, and two matrix instructions. The overall architecture of TPU v2 as compared to TPU v1 is more generalized to accommodate the wide range of deep neural networks and tasks associated with them. TPU v2 also provides more local buffers and more accessibility and communication channel through the interconnects. This thesis focuses on TPU v2 core implementation. The thesis compares two different implementations of vector processors, VESPA, and VIPERS, to form the foundational design. The details of these processors have been mentioned in the thesis.

VESPA and VIPERS vector processor cores

VESPA [12] and VIPERS [13] are vector processor implementations that are used for the comparison to finalize basis design for TPU v2 core. These processors support a vector processing unit and a scalar processing unit along with a vector control pipeline. The VESPA uses MIPS based simple 3 stage pipeline scalar core, while the VIPERS use a non-pipelined 4 stage multithreaded scalar core. In addition, VESPA and VIPERS use an adapted version of the vector IRAM instruction set [14] for their vector cores. Both these implementations supported vector to scalar and scalar to vector data transfers. The overall architecture of VESPA implementation is described in Figures 8, 9 below. The figures are taken from [12]

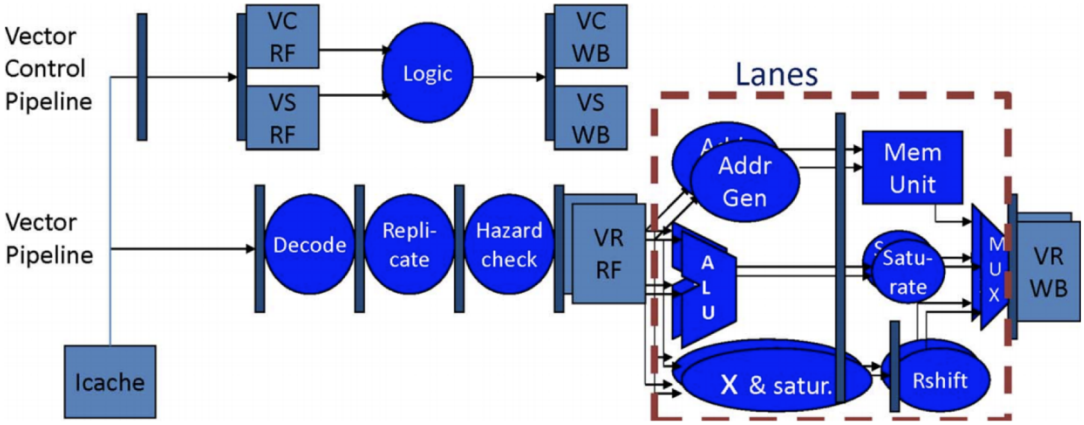

Figure 8: VESPA Architecture diagram

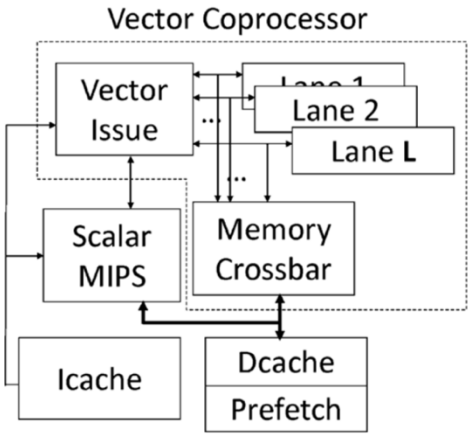

Figure 9: VESPA Implementation Block diagram

Here is the overview of comparison analysis:

VIPERS

a. VIPERS do not support vector chaining. This is because the loss of vector chaining potentially would hurt the performance for long vector lengths.
b. VIPERS implementation does not support external memory access. Instead, it uses an on-chip ram to store the data and program.
c. Each vector lane has its local memory, as expected in the implementation of TPU v2 core.
d. Vipers do not use caches.

VESPA

a. VESPA cores are simpler than vipers in terms of design complexity.
b. VESPA cores supported I-cache and D-cache instead of local memories. VESPA uses a multiplexer and demultiplexer logic with arbitration to access the D-cache between the scalar core and all lanes of the vector core.

c. VESPA core supported a DDR memory controller to access main memory.
d. VESPA cores support vector chaining.

With this brief analysis, the VESPA vector processor implementation was a reasonable basis for TPU v2 design. In order to implement over benchmark design, several modifications were carried out which are explained in detail in further section.

FPGA Benchmarking

The development of novel FPGA architectures uses different types of benchmarks. These benchmarks are very crucial to capture the market the application market targeted by the FPGA architecture. A lack of these benchmarks or non-representative benchmarks does not help to optimize the benchmarks for targeted segments. The commonly used benchmarks for FPGA benchmarking mentioned in introduction section includes very small design which does not utilize the complex blocks within FPGA and are not representatives of state-of-the-art design use in the target applications. The UMass RCG HDL Benchmarks [21] represents complex design which use digital signal processing applications and are not targeted for open-source FPGA benchmarking. The TPU v2 core provided in the thesis will be a part of larger set of frameworks for benchmarking of FPGA for DL specific workloads. The TPU v2 core is a commercial design of DL accelerator used by Google [10]. Therefore, it represents the commercial application of DL specific workload.

Verilog To Routing (VTR)

VTR tool is an open-source tool for FPGA architecture and CAD research. The VTR design flow takes a verilog RTL design file and an FPGA architecture description file of target architecture. The tool then performs elaboration and synthesis, logic optimization and technology mapping, and place and route for the design on target FPGA to produce the details of design implementation such as frequency, wirelength, component utilization, etc. [19]

The VTR design flow uses Odin II for synthesis and elaboration. Some of the complex design constructs, such as generate statements, multidimensional arrays, and integer variables, etc., are not supported by this open-source academic tool. Scripts were used to modify the verilog construct used in the benchmark design into supported constructs to work around this limitation. Any additional vendor specific design elements were replaced with the ones that are compatible with VTR.

DESIGN AND VERIFICATION

In order to create a TPU v2 benchmark design from the VESPA vector cores, microarchitectural updates and the addition of functional units to support some of the complex neural network functions were required. In this section of the thesis, the design implementation and microarchitecture of the benchmark TPU v2 have been explained.

SCALAR CORE OF VESPA

The MIPS-based scalar core of VESPA has a 3-stage pipeline architecture with data forwarding [12]. The processor is auto generated by SPREE RTL generator [15]. The scalar core can work in parallel with the vector processing unit without being stalled. The three pipeline stages include the fetch and decode stage, register read and execute state, and writeback state. The scalar core also executes memory access in execute state. Finally, the scalar core can communicate with the vector core through communication instructions. Due to the limited role of this core processor in TPU v2, the execution of VESPA's scalar core is not modified due to its limited use in DL applications for the implementation of the TPU v2 core.

VECTOR CORE OF VESPA

The vector unit is a 7-stage pipeline unit, as shown in figure 8. The seven stages design includes multiple lanes to support vector operations. In TPU v2 benchmark design, the number of vector lanes is set to be eight to process eight operations in parallel. Each vector to be processed by the vector processor can have a length greater than or equal to the size of vector lanes. In such a case, the vector to be processed utilizes multiple clock cycles to process. The replicate stage takes care of such replication operations. It includes a dispatcher unit that dispatches part of the vector instruction that the vector core can process at a time. The maximum vector length for a program is defined using a parameter and is fixed throughout the program.

DESIGN CHANGES

The VESPA vector processor is modified to resemble the TPU v2 core, as discussed in Section 2.3. The VESPA vector processor lacks functional units like the matrix multiplication unit and TRP unit. These are the heart of the TPU v2 core as they accelerate the matrix operations such as multiplication, transpose, reduce, permute, etc. The caches were not required as TPU v2 have simple scratchpad memories where instructions can move data into local memory. The programmer can address the data as the program manages data allocation within the memory by itself. This reduces the indeterministic nature of caches and provides simple hardware. The VESPA implementation of vector processor lanes and scalar processor uses the same D-cache with an arbiter and mux logic to prioritize requests as they reach the data cache. However, TPU v2 includes a local memory for processing per lane of its vector unit and the scalar unit, enabling each vector lane to operate separately. The TPU V2 includes two TPU v2 core connected through the interconnect network within a TPU v2 node. The overall VESPA implementation lacks the compatibility to be connected with another VESPA implementation. Google introduced a new custom floating-point format called "bfloat16" a less precise version of IEEE 754 single-precision floating-point number format [16]. The smaller size of bfloat16 reduces the overall data size while not a significant loss of precision for deep learning workloads [17]. The VESPA implementation does not support floating-point operations and bfloat16 data type.

INSTRUCTION SET

The vector processor is based on an instruction set called vector IRAM [14]. Table 2 provides a list of instructions supported in the VESPA implementation of the vector processor. For DL-specific workloads, The benchmark design added few more instructions, which are listed in Table 2 as custom instructions (instruction opcodes of some existing instructions were repurposed to reduce the work of changing the compiler/assembler and focusing on hardware changes). In addition, to convert the hardware-managed cache to software-managed memory.

Mnemonic	Operation
Integer Arithmetic Instructions	
vabs	Absolute Value
vadd	Add
vsub	Subtract
vmullo	Multiply Low
vmulhi	Multiply High
vmod	modulus

Mnemonic	Operation
Logical	
vand	And
vor	Or
vxor	Xor
vnor	Nor
vsll	Shift left logical
vsrl	Shift right logical

Mnemonic	Operation
Integer Arithmetic Instructions	
vsra	Shift Rigth Arithmetic
vcmp	Compare
vmin	Min element
vmax	Max element
Load/store	
vfld	Load flag
vld	Unit stride load
vlds	Variable stride load
vldx	Indexed Load
vfst	Flag store
vst	Unit stride store
vsts	Variable stride store
vstx	Index stride store
Custom Instructions	
vdiv	Matrix multiplication

Mnemonic	Operation
Flag logical	
vfand	Flag And
vfor	Flag Or
vfxor	Flag Xor
vfnor	Flag nor
vfclr	Flag clear
vfset	Flag set
Control Instructions	
vmcts	Move control to scalar
vmstc	Move scalar to control
vsatvl	Saturate vector length
cfc2	Control from cop2
Mtc2	Move from cop2
Ctc2	Control to cop2
Custom Instructions	
Vsts_w	AXI store

vld_w	Activation operation	Vsts_w	AXI load
vlds_w	Transpose operation	Vsqrt	Bfloat16 add
vldx_w	Reduce operation	vxlmul	Bfloat16 multiply
Vst_w	Permute operation		

Table 2: Instruction set for TPU v2 benchmark.

Most of the instructions are adapted from VIRAM as done in VESPA [19]. In addition, some custom instructions are added for the support required for DL work.

MATRIX MULTIPLICATION UNIT

Matrix multiplication is the most critical operation in DNN. It is used for both matrix multiplication as well as convolution. Although the support for vector multiplication of bfloat16 data format is available, a matrix multiplication operation requires a high throughput design to extract the parallelism and reuse available within the data [10]. A systolic array is integrated into the vector processor as a matrix multiplication unit to achieve the performance goals of the matrix multiplication operation. The data to the matrix multiplication unit is transferred from the registers. The size of the systolic array is restricted to 8x8 for this academic benchmark. The matrix multiplication unit takes 29 cycles to produce a matrix multiplication of two 8x8 matrices. The result of matrix multiplication is stored in the registers over eight cycles. Each cycle stores the result into the register file of each lane.

BFLOAT16 UNITS

TPU v1 includes an accumulator block to accumulate the result of matrix multiplication, as shown in figure 3. The vector addition operation replaces the accumulation in TPU v2. As the matrix data is stored in bfloat16 format, functional units supporting bfloat16 formats are necessary for the benchmark. Therefore, the TPU v2 core benchmark includes add-subtract unit and multiplies unit supporting bfloat16 data type. Each of these units is 3 stage pipeline design.

REPLACEMENT OF CACHE TO LOCAL MEMORY

The TPU v2 cores have local memories per lane used by each lane to store the results and load the data. Thus, each lane can optimally work in an independent way to do the operation. VESPA vector core uses multiplexer logic to share the D-cache accesses between the scalar core and each lane of the vector core. In order to resemble the TPU v2 approach, the D-cache access from each vector lane is replaced by a local scratchpad memory for each lane. The continuous address space is divided so that a sequence of 8 contiguous address spaces is divided among each lane.

A DMA is designed used to load/store data from the main memory into these local memories. The DMA access is done through programming of control register with the instructions `ctc2` and `cfc2`. The DMA takes the start address, length of transfers and source address, and the destination address to store the data as required. The DMA polls the control registers present in the vector control pipeline. The transfer is initiated by first loading the transfer-related information into the DMA and then setting the control register with value 1. The DMA data transfer writes value one into another control register after the completion of data transfer. Software polling is necessary to this control register to ensure the program data is available in local memories. An interrupt-based scheme can be developed in the future. The local memories used in each lane are two-port memories where one port is accessible by the DMA, and the other port is accessible by the vector unit. Figure 10 explains the data flow of local memory. There is one local memory per vector lane. The vector control registers control the DMA access.

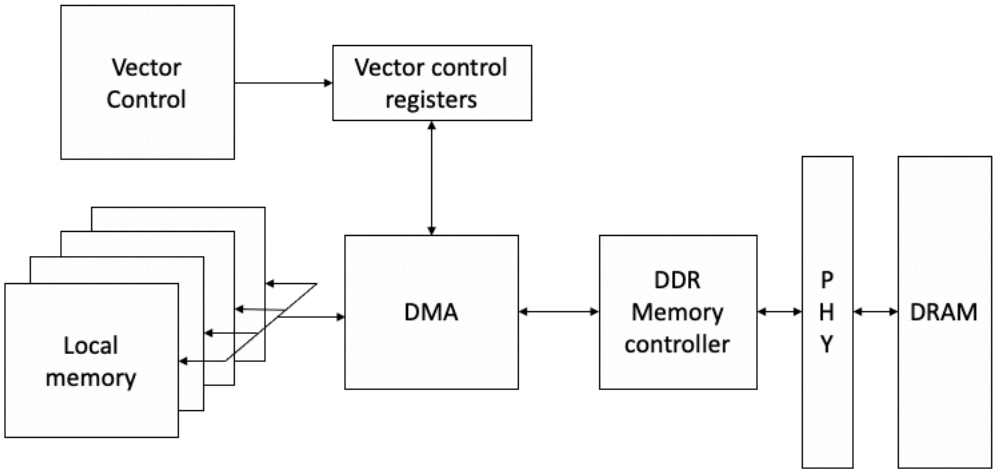

Figure 10: The block diagram for TPU v2 core benchmark memory system for vector unit.

TRANSPOSE REDUCE PERMUTE (TRP) UNIT

TRP unit helps to transpose, reduce or permute the matrix. This unit can load an 8x8 matrix from registers and provide the transpose of the matrix. The design uses a local 8x8 flop structure to store the matrix and does the transformation in 1 cycle. The output is stored into the register in 8 cycles with one write operation in each lane in each cycle. The reduce operation reduces all the elements of the matrix. The reduction operation can be selected using mode signals. The design supports three reduction operations: the addition of all elements, largest element reduction, and least element reduction. The reduction operation uses a tree-based structure to generate a single output which is stored in all lanes. The permute operation performs matrix permutations in a way that it can shuffle the matrix rows and columns. The permute operation loads an 8x8 matrix and accepts a transform vector for rows or columns. The transform vector provides the details for shuffling of rows or columns. The output of permute unit is a shuffled matrix.

CORE TO CORE INTERFACE

The TPU v2 core has the functionality to communicate with other TPU v2 core. An AXI master-slave interface is added to the TPU v2 core. The AXI master initiates a transfer when the benchmark design issues an instruction for AXI transfer. The AXI slave interface is connected to the vector memory using a multiplexer structure that shares the other port with DMA logic. The overall design is present in figure 11. The AXI master-slave interface enables two TPU v2 cores to be connected. The AXI interface initiates transactions through instruction at vector core. The slave interface shares the access to the local memory with DMA.

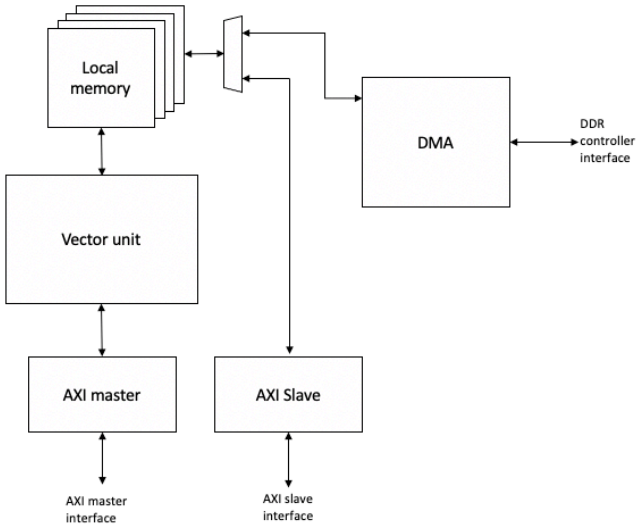

Figure 11: AXI interface for TPU v2 core benchmark

ACTIVATION UNIT

Most of the Deep learning workloads require an activation function after the convolution layer and dense layers. The TPU v1 used activation units to carry out the activation function of Neural Networks. The TPU v2 core has special execution units in the vector processor to carry out activation operations on data read from register. The benchmark design adds another functional unit called the activation unit to support activation, carrying out the activation function. The

activation unit in the benchmark uses ReLU activation function. The unit performs a check sign bit of input activation. For the numbers with negative sign, the output of activation unit is set to value zero. For all positive numbers, the output of activation unit is the activation itself.

OVERALL MICROARCHITECTURE

The overall design for TPU v2 after the modifications mentioned earlier is shown in Figure 12 below.

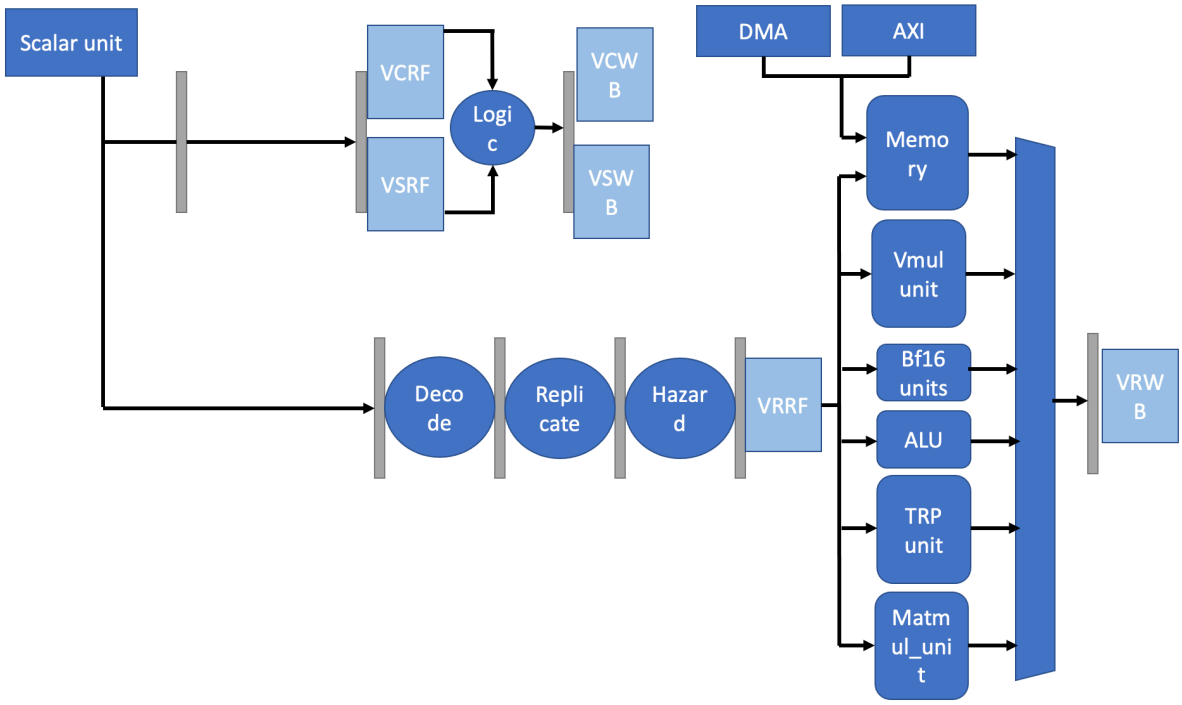

Figure 12: The overall microarchitecture of TPU v2 benchmark design

FILE STRUCTURE AND CODE INFORMATION

Verilog code that models a TPU v2 like processor is created in this work. The verilog code is open source and available on github: https://github.com/sangramkate/tpu_v2. The overall source code is present in folder `tpu`. The `tpu` folder contains 5 major directories: `apps, verif, design, doc ,vtr.`

The `apps` directory contains the programs to be run on TPU v2. It includes a `Makefile` to create new programs. It requires a reference to the `compiler-vector` which is a modified MIPS compiler. The compiler is available at: https://www.eecg.utoronto.ca/VESPA/. Along with the C program, an VIRAM assembler function is required for instructions related to vector co-processor. The assembler function and the main c program gets compiled by the `Makefile`.

The `design` folder contains the source code for the design. All vector processor code is stored in `vector` folder. The scalar core is stored in `scalar` folder. Some of the logic such as adder subtractor unit, fifo, etc. were inserted to replace the existing ones. These can be found in `local` folder. The `top` folder contains the top-level module is in file `de.v`. which instantiates `processor.v` which includes the scalar core as well as the vector core. The top level of scalar core is defined in `system.v` under `scalar` folder and the top level of vector core is defined in `vpu.v` under the `vector` folder. The `bfloat` folder contains the additional logic for bfloat16 adder, subtractor, multiplier. It also includes the transpose, reduction and permutation operation. The design for systolic array is present in the `top` directory.

The `doc` directory contains the update log and setup commands for the repository. The `verif` folder contains the test bench `de3_test_bench.v` file. The initialization files include `instr.dat` and `data.dat` which contains instructions and data to be run on the TPU v2 benchmark. These files are stored in the `verif` directory. The `verif` directory includes a `Makefile` to compile and simulate the program. This `Makefile` is accessed by the top level `Makefile` present at the root of repository

The `Vtr` directory contains the scripts that are used to convert the complex Verilog contructs into the `Vtr` acceptable constructs. The directory contains the scripts for scalar processor at the top. It also contains a folder called `vector scripts` which contains the scripts for vector processor. As the defines and parameters are propagated in vector processor from top level hierarchy to all leaf level modules, the toplevel script `vpu.py` runs all scripts to generate the design. There are separate `Makefiles` to generate both scalar and vector processor.

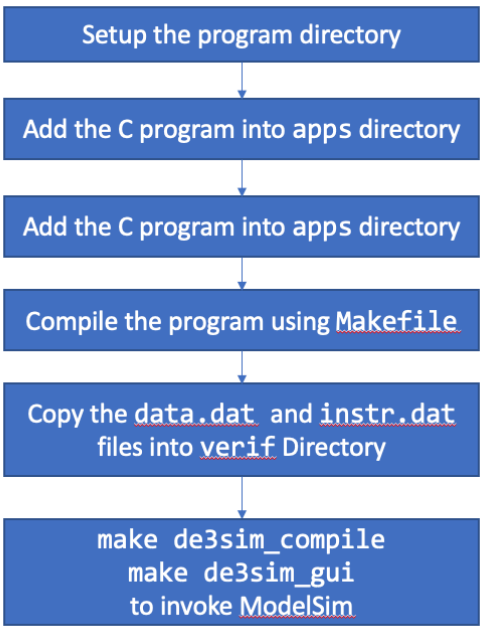

Figure 13: Overall execution and simulation flow for TPU v2.

VERIFICATION

To verify the modified design elements, a C program is used which performs the matrix multiplication operation of two 8x8 matrices. bfloat16 add and multiply operations are also performed using a similar program that loads 16 bfloat16 values and performs bfloat16 add, subtract and multiply for these operations and stores the result back into the main memory. The VESPA core uses a modified MIPS compiler which also supports the extended VIRAM ISA that is used in vector processor. The vector code is written in a `program.S` file which is added to the C program as a function. The required data for the processing of vector operations is passed as a function input to these codes. The vector code is an assembly code which directly use the instructions referred in table 2. The main C program is compiled into a MIPS binary. The VIRAM instructions for vector processor are embedded into this binary file.

The compiled data and instruction files are copied to the `verif` directory where the simulation runs. The Verification flow requires use of ModelSim. The program stores this data into the main memory. The instructions are stored into the local memory of scalar core. The

VESPA implementation supports DDR2 memory access as it uses an altera memory controller logic. This memory controller is used to read the data into the vector processor. The overall flow of program execution is shown in the figure below.

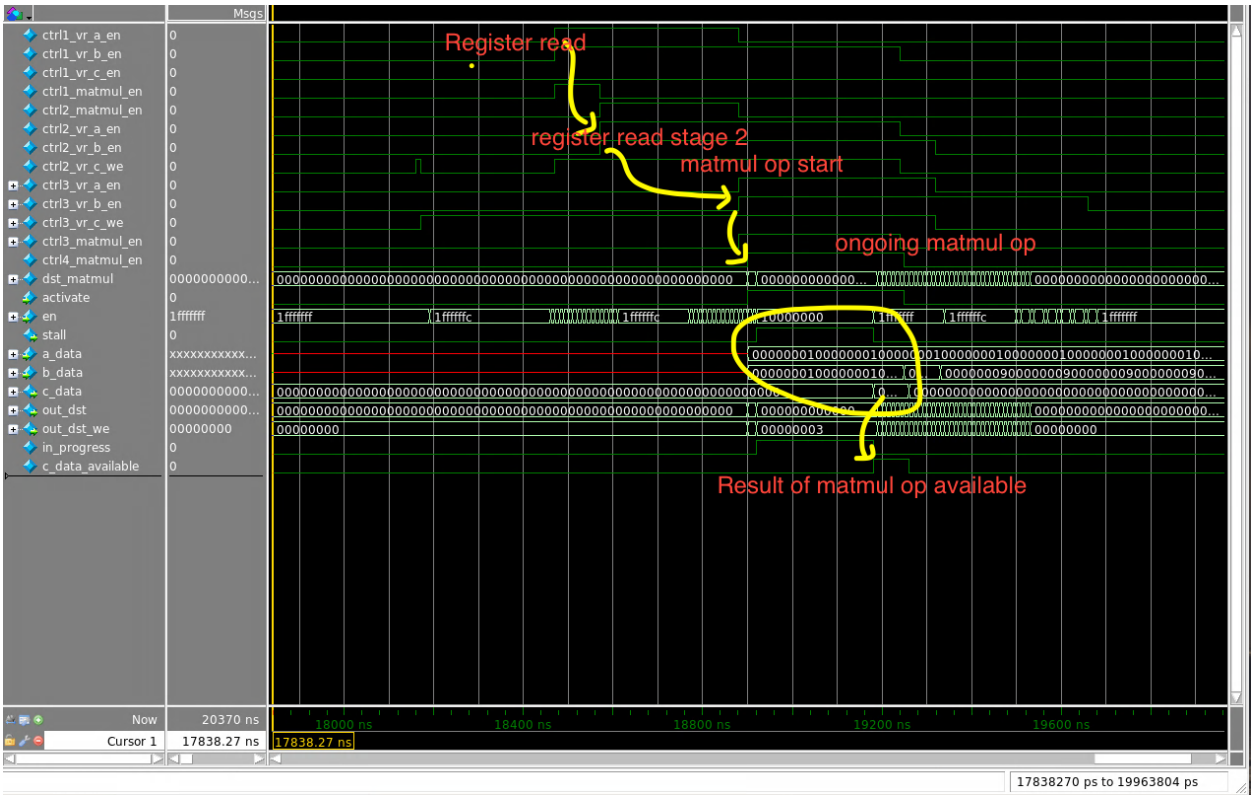

Figure 14: A simulation of matrix multiplication program

VTR ANALYSIS AND RESULTS

This section of the thesis explains the experiments carried out with the design of the TPU v2 core, to facilitate its use as a benchmark. As TPU v2 is a deep learning application-specific design, it is a benchmark for FPGA architecture evaluation for deep learning tasks. The analysis for the benchmark is done using the Verilog-to-Routing (VTR) tool, which is one of the most widely used FPGA architecture analysis frameworks [19]. The utilization of block RAMs (BRAM) and DSP slices of a specific FPGA architecture impact the performance of workload design. The following section analyzes the impact of different FPGA architectures on the performance of the TPU v2 benchmark.

EXPERIMENTAL SETUP

The thesis work uses VTR 8.0 for the experiments which is the latest version. First, an SDC (Synopsys Design Constraints) file is provided for VTR flow. The SDC file declares all IO-to-register paths as false paths for timing analysis, thereby keeping only register-to-register paths. Then, VTR flow is run to optimize the clock frequency for thedesign. The auto-layout feature is used for all experiments and a maximum of 150 routing optimization iterations with a channel width set to 300. Finally, an average from 3 runs is taken to demonstrate the final result.

A custom FPGA architecture description file is used for the experiments, which is modeled using COFFE [20] with a 22 nm technology node. This custom architecture file uses columns of logic blocks, DSPs, and block RAMs (BRAM). The DSP slices and BRAMs are interleaved between the columns of logic blocks. The density of these blocks is varied for different sets of

architectures. The architecture uses unidirectional wire segments of lengths 4 and 16. The block pins are accessible through a wire length of 4 only. Furthermore, the switches appear on every 4th column.

RESULTS

Tables 3, 4, and 5 show the main results of VTR analysis on TPU v2 design. First, the result shows the design of TPU v2 with netlist primitives of 25k. Next, Table 3 discusses the utilization of logic within an FPGA for the benchmark. Finally, the data in Table 3,4, and 5 helps us to understand the design complexity involved.

Parameters	Value
Netlist primitives	25655
Logic depth	8
Used IOs	489
Used LBs	702
Used DSPs	68

Parameter	Value
Used BRAMs	49
Single bit adders	2604
Flip-flops	5272
Max fanout	5753
LUTs	9192

Table 3: VTR Flow results for TPU-v2 core.

Max Frequency(in MHz)	Routed wire length(in length 1 wires)	Grid Size
100.9677761	104973	70x70

Table 4: VTR flow results for TPU-v2 core.

Parameters	Values
VTR flow elapsed time	1865.04
Odin time	121.38
ABC time	184.22
Pack time	49.81
Place time	41.23
Route time	6.7
Peak Memory usage	1231832

Table 5: VTR Flow runtime analysis for experiments on TPU-v2

Table 5 provides us VTR Flow related analysis useful for CAD research purposes. The experiments were carried over Intel Xeon CPU E5-2430 running at 2.5 GHz with 64 GB of Memory. The table provides the details of peak memory usage for using VTR flow on the TPU v2 benchmark with the design size and complexities mentioned in Tables 3 and 4. As for the baseline run, the achieved frequency of operation was 100 MHz with a grid size of 70x70.

Architecture Exploration

In this section of work, VTR flow is run on TPU v2 benchmark to analyze the impact of different types of FPGA architectures on the performance of benchmark design. The experiments use several different types of architecture configurations, as shown in Figure 12. These architectures are different from each other in terms of the placement of DSP slices and BRAMs and their densities. For example, in Figure 12, Each architecture style depicts three different versions ranging from lower densities of DSP slice and BRAM at the top to denser architectures in the following rows.

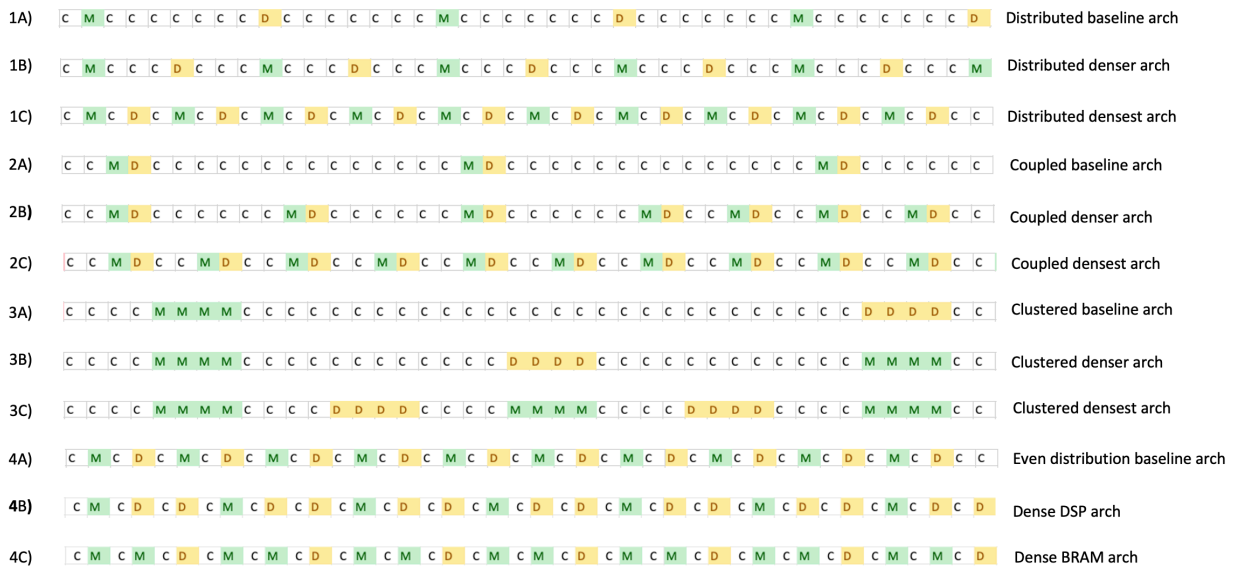

Figure 15: Different FPGA architectures for TPU v2 experiments

The analysis of these experiments is summarized in Table 6,7,8 and 9. Each architecture style impacts the achieved frequency, total wire length, and grid size. Therefore, four different comparisons of the design- architecture relation are provided in Tables 6,7,8 and 9. experiment 1 shows the tradeoffs between distributed, coupled, and clustered architecture with respect to their baseline. Experiment 2,3 further evaluates the impact of densities of BRAMs and DSP slices on benchmark performance for clustered and coupled architectures. Finally, experiment 4 compares architectures with higher densities of DSP slices vs. the architecture having higher densities of BRAMs.

Experiment 1	Frequency (MHz)	Wirelength*	Grid Size	Avg wire segments per net*
1A	99.87162691	107549.3333	70x70	8.47681
2A	106.8608573	105547.6667	70x70	8.42256
3A	106.8963744	96137	70x70	8.06376

Table 6: VTR Flow result comparisons for TPU-v2

Experiment 2	Frequency (MHz)	Wirelength*	Grid Size	Avg wire segments per net*
3A	106.8963744	96137	70x70	8.06376
3B	106.517103	99169	56x56	8.12544
3C	99.07488318	98556.33333	38x38	8.3923

Table 7: VTR Flow result comparisons for TPU-v2

Experiment 3	Frequency (MHz)	Wirelength*	Grid Size	Avg wire segments per net*
4A)	99.06077481	102137	39x39	8.74357
4B)	100.7686076	103311.3333	39x39	8.84129
4C)	96.3184402	103482.3333	44x44	8.72768

Table 8: VTR Flow result comparisons for TPU-v2

From experiment 1, It is can be observed that coupled and clustered FPGA architectures for DSP slices and BRAMs provide better performance for the TPU v2 core benchmark than that of distributed DSP slices BRAM architecture. The evaluation of the coupled and clustered architectures for higher densities is done in experiment 2. As the densities of these modules increase, there is a decrease in overall grid size. However, due to the smaller number of logic blocks between the clustered DSP slices and BRAMS, an overall gain in wirelength and average wire segment per net is observed, as shown in table 7. This increased wire length amounts to decreased frequency of operation. In the last experiment, the performance of the TPU v2 benchmark for uneven distribution of DSP slices and BRAMs is compared. The VTR flow utilized 68 DSP slices and 49 BRAM for the TPU v2 run, as shown in table 3. It is observed that an architecture that has fewer DSP slices suffers a performance loss compared to other architectures. This set of experiments provides good details about the dependency of benchmark design performance on different architectures.

*The unit of wirelength and avg wirelength per net are length 1 wire which is size of 1 logic block in an FPGA.

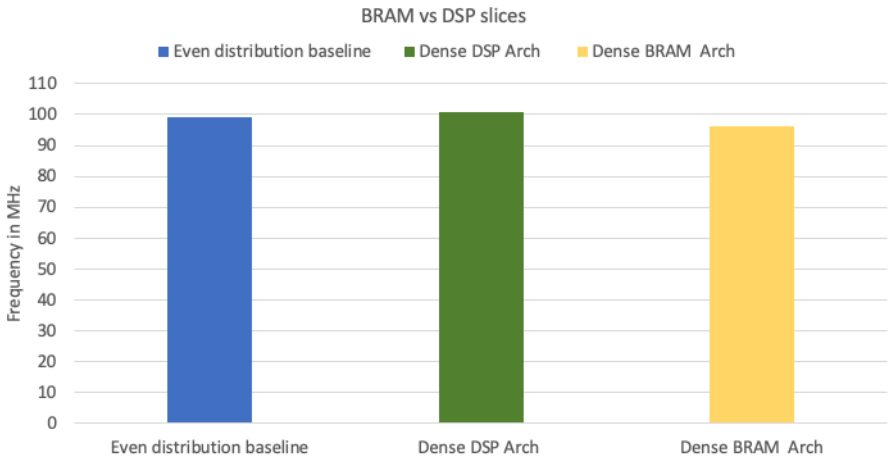

Figure 16: VTR flow analysis on TPU v2 Frequency for different architectures

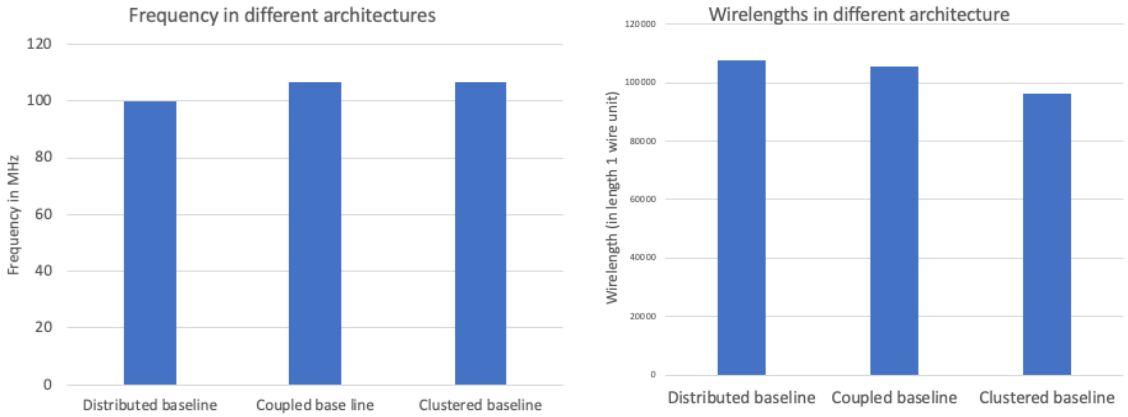

Figure 17: Frequency and wirelength analysis using VTR Flow for different DSP slice and BRAM densities

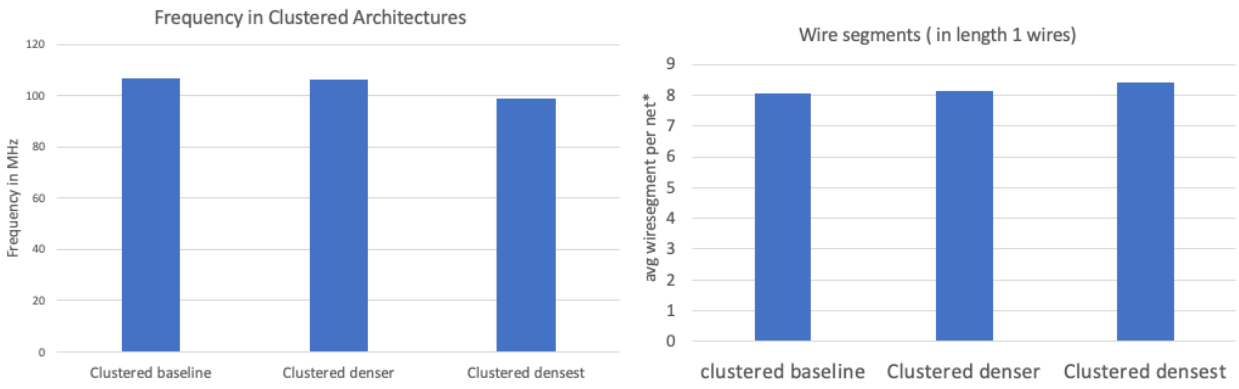

Figure 18: Frequency and avg wirelength per net analysis using VTR flow for different DSP slice and BRAM densities.

CONCLUSION

In this thesis, an academic TPU v2 deep learning benchmark design for FPGA architecture analysis is presented. This design is an academic version of the core logic of Google's tensor processing unit version 2. The microarchitectural details of the benchmark TPU v2 core and the result of running this design using VTR flow are presented. The thesis further demonstrates the TPU v2 benchmark as a reference for architecture analysis through the different set of the experiment using target FPGA architectures different from each other in terms of the densities of BRAMs and DSP slices and their distribution within the FPGA. This benchmark design can further help to understand the target FPGA architectures for DL-oriented workloads.

BIBLIOGRAPHY

[1] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner, MIT Lincoln Laboratory Supercomputing Center arxiv: https://arxiv.org/pdf/1908.11348.pdf

[2] G. Lacey, G. W. Taylor, and S. Areibi, "Deep Learning on FPGAs: Past, Present, and Future," *arXiv:1602.04283 [cs, stat]*, Feb. 2016, Accessed: Apr. 18, 2021. [Online]. Available: http://arxiv.org/abs/1602.04283.

[3]M. Langhammer et al., "Stratix 10 NX Architecture and Applications," in International Symposium on Field-Programmable Gate Arrays (FPGA), 2021.

[4] E. Nurvitadhi et al., "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019.

[5] S. Ahmad et al., "Xilinx First 7nm Device: Versal AI Core (VC1902)," in Hot Chips Symposium, 2019.

[6] E. Vansteenkiste, A. Kaviani and H. Fraisse, "Analyzing the divide between FPGA academic and commercial results," 2015 International Conference on Field Programmable Technology (FPT), Queenstown, New Zealand, 2015, pp. 96-103, doi: 10.1109/FPT.2015.7393137.

[7] N. P. Jouppi *et al*., "In-Datacenter Performance Analysis of a Tensor Processing Unit," *arXiv:1704.04760 [cs]*, Apr. 2017, Accessed: Apr. 18, 2021. [Online]. Available: http://arxiv.org/abs/1704.04760.

[8] M. Z. Alom *et al*., "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches," *arXiv:1803.01164 [cs]*, Sep. 2018, Accessed: Apr. 18, 2021. [Online]. Available: http://arxiv.org/abs/1803.01164.

[9]Sangkug Lym, and Mattan Erez. FlexSA: Flexible Systolic Array Architecture for Efficient Pruned DNN Model Training., 2020.

[10] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A domain-specific supercomputer for training deep neural networks. *Commun. ACM* 63, 7 (July 2020), 67–78. DOI:https://doi.org/10.1145/3360307

[11] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. 2008. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems* (*CASES' 08*). Association for Computing Machinery, New York, NY, USA, 61–70. DOI:https://doi.org/10.1145/1450095.1450107

[12] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. 2009. Vector Processing as a Soft Processor Accelerator. *ACM Trans. Reconfigurable Technol. Syst.* 2, 2, Article 12 (June 2009), 34 pages. DOI:https://doi.org/10.1145/1534916.1534922

[13] C. E. Kozyrakis and D. A. Patterson, "A new direction for computer architecture research," in *Computer*, vol. 31, no. 11, pp. 24-32, Nov. 1998, doi: 10.1109/2.730733

[14] P. Yiannacouras, J. Gregory Steffan, and J. Rose, **Application-Specific Customization of Soft Processor Microarchitecture**, to appear in ACM International Symposium on Field-Programmable Gate Arrays (FPGA 2006), February 2006, Monterey, CA

[15] Wang, Shibo; Kanwar, Pankaj (2019-08-23). "BFloat16: The secret to high performance on Cloud TPUs". *Google Cloud*. Retrieved 2020-08-11.

[16] D. Kalamkar *et al.*, "A Study of BFLOAT16 for Deep Learning Training," *arXiv:1905.12322 [cs, stat]*, Jun. 2019, Accessed: Apr. 18, 2021. [Online]. Available: http://arxiv.org/abs/1905.12322.

[17] Videoblog on TPU v2 design at HotChips 2020: Link: (https://www.anandtech.com/show/16005/hot-chips-2020-live-blog-google-tpuv2-and-tpuv3-230pm-pt)

[18] ViRAM instruction set for vector processor: Link;(http://iram.cs.berkeley.edu/isa.ps)

[19] K. E. Murray *et al.*, "VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling," *ACM Transactions on Reconfigurable Technology Systems (TRETS)*, vol. 13, no. 2, 2020.

[20] S. Yazdanshenas and V. Betz, "COFFE2: Automatic Modelling and Optimization of Complex and Heterogeneous FPGA Architectures," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, 2019.

[21] J. Allen. (2006) UMass RCG HDL Benchmark Collection. [Online]. Available:http://www.ecs.umass.edu/ece/tessier/rcg/benchmarks/