GPUs and TPUs: an analysis

[Anirudh Arputham, Ashutosh Kumar Verma, Mayank Singhal, Sriram Balasubramanian, Yash Gupta]

Abstract

We describe and compare in this report the architecture, working, implementation and several optimizations which are used in graphics processing units (GPUs) and tensor processing units (TPUs).
This report describes in detail the architecture of a GPU, its memory subsystem, various implementations of parallelism in the GPU and the different stages that a GPU follows. It also describes CUDA - a parallel computing environment by nVidia.
The instruction set architecture and block diagram of TPUs are described in detail. The TPU software stack is described. Optimizations employed in TPUs are discussed. Some future optimizations are pointed out.

Graphics Processing Unit (GPU)

Introduction

GPU (Graphics Processing Unit) is a specialized electronic chip which is used to manipulate and alter memory rapidly. In the early days, they were mounted on the graphics card (or video card) to convert 3D scenes to 2D images which finally has to be displayed on the user's screen. Nowadays, Modern GPUs are used for general purpose tasks which are highly parallelizable, and they act as a supplement to CPU. They can be mounted on a video card or embedded on the motherboard. In certain CPUs, they are integrated on the CPU die itself.

GPUs prioritize throughput over latency. In real time applications, computational requirements are enormous. So, GPUs have to be highly parallelizable to meet user requirements. GPUs have applications in several multimedia, scientific and research fields like Graphics rendering, Image Processing, Machine Learning, and Weather Forecasting.

The first GPU (GeForce 256) was launched by Nvidia in 1999. There are two main types of GPUs available in the market:

1. **Dedicated Graphics Card:** These GPUs have a dedicated memory which has much higher bandwidth as compared to the usual RAM. They are attached to the motherboard using PCIe (Peripheral Component Interface Express) or AGP (Accelerated Graphics Port) slot. These are the most powerful type of GPUs as compared to others.

2. **Integrated Graphics Card:** Integrated Graphics Card utilizes a portion of the system's RAM rather than having a dedicated memory. They are also called UMA (Unified Memory Architecture) systems. They are mostly integrated on the CPU die (e.g., Intel HD Graphics). Since the memory is shared between CPU and GPU and the bandwidth is lesser as compared to the dedicated graphics card, they have to compete with CPU for memory.

Graphics Pipeline

The task of a graphics system is to render a 3D scene onto a 2D screen. It is a pipelined process, and all the stages can easily work in parallel. It involves roughly five stages:

1. **Vertex Processing:** Each vertex is transformed into a 2D screen position (screen space) and shaded according to its interaction with light. This stage is well suited for parallel hardware because each vertex can be transformed independently.

2. **Primitive Processing:** In this stage, vertices are clipped and assembled into triangular primitives.

3. **Rasterization:** Primitives are rasterized into pixel fragments. Rasterization involves the computation of determining which screen-space pixel locations are covered by each triangle.

4. **Fragment Processing:** Using color information from the vertices and the texture information, each fragment is shaded to determine its final color. Each fragment can be processed independently.

5. **Pixel Operations (Composition):** Fragments are assembled into one final image. And one color is chosen for each pixel using z-index.

Parallelism in GPUs

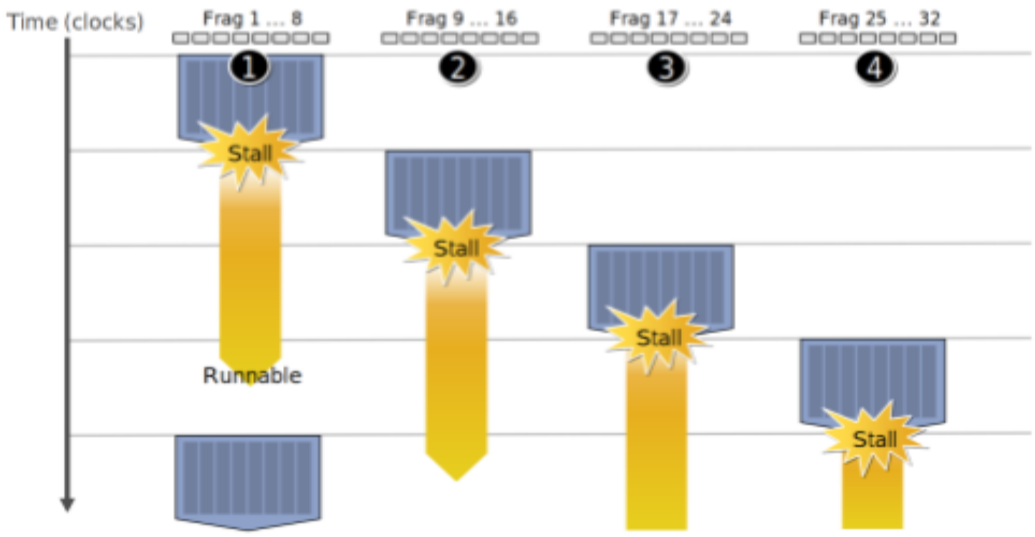
GPUs are originally designed to accelerate computer graphics, which involves updating millions of pixels on a screen 60 times every second. On a modern system with a Full-HD screen resolution, this can mean that the GPU has to execute more than a million million instructions per second!
Clearly, to maintain such a high throughput while maintaining latency within tolerable levels, a GPU must exploit a high degree of parallelism, and in several forms. Below is a brief overview of the various ways in which a GPU exploits parallelism to cope with its massive workloads, some details about which will become clearer in the upcoming sections.

Instruction-Level Parallelism

Pipelining: Similar to CPUs, all modern GPUs employ ‘pipelining’ at various levels in their architecture. Instructions can be executed out-of-order, different instructions can be simultaneously executing on different units, or even on the same unit.

Superscalar Processing: A GPU can start multiple instructions in the same cycle. For example, in nVidia’s Kepler architecture, upto 2 independent instructions can be started per clock, if they belong to the same warp.

Thread-Level Parallelism:



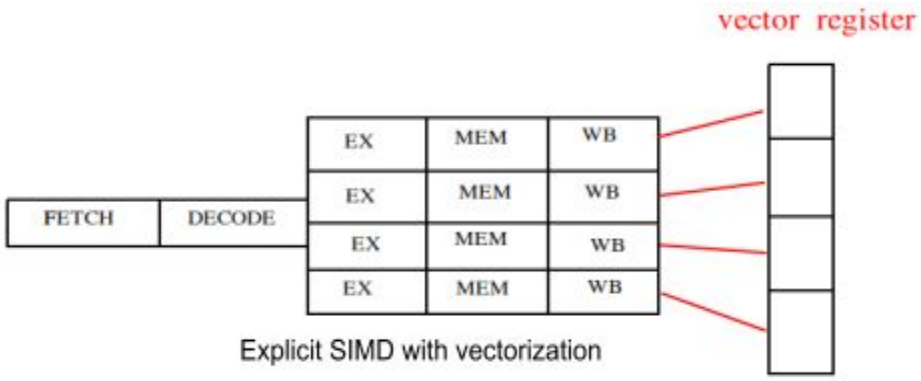
Hiding latency stalls by interleaving independent threads

A modern GPU is a multiprocessor made up of multiprocessors, which in turn consist of highly multithreaded processing cores. The end result is an extremely scalable, highly threaded processing device. The massive degree of multithreading, combined with the ability to switch execution contexts without any significant overhead(thanks to thousands of registers) allow GPUs to very effectively hide the latency of memory loads, execution stalls etc.

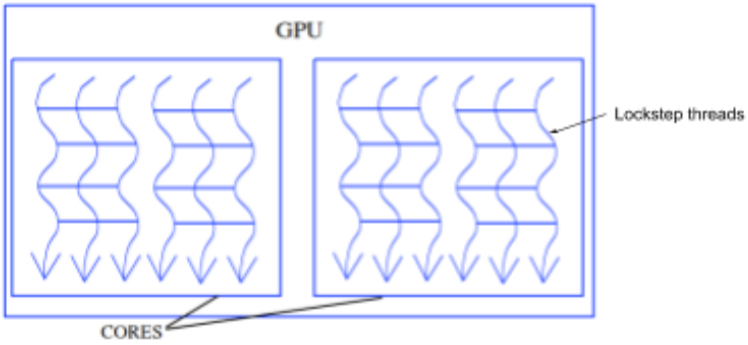
Data-Level Parallelism

Exactly how a GPU extracts data parallelism is closely coupled with its underlying architecture, and varies from one GPU to another. But broadly speaking, two overarching models emerge in modern GPUs:

Single Instruction Multiple Data (SIMD): In SIMD, vector instructions (and vector registers) are used and elements of short vectors are processed in parallel. It is a 'horizontal' kind of parallelism because a single instruction is broadcasted to multiple execution units. This way we split our workload over several execution units and at the same time eliminate redundant fetch/decodes.



Single Instruction Multiple Threads (SIMT): Developed by nVidia, SIMT is sort of a hybrid of SIMD and hardware multithreading, because it splits identical, independent work over multiple *lockstep* threads. Lockstep threads basically means that some group of threads execute in a synchronized fashion (they all have the same program counter) - and this is all implicitly managed by the hardware - the hardware determines how to setup stream sharing across multiple ALUs.



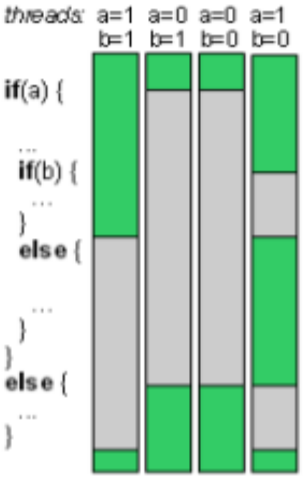
A GPU 'core' is a stream multiprocessor, which can handle multiple such warps simultaneously. While the obvious costs to SIMT are that many more registers are used to store so many execution contexts, but the advantages of doing so are numerous:

1. a core can effortlessly interleave many thread groups to hide latency stalls.
2. batching together groups of synchronized threads can reduce incoherent random memory accesses.

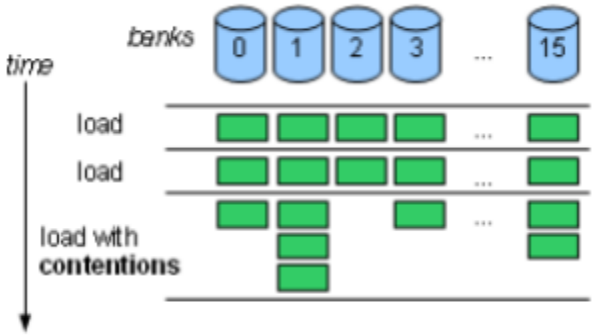
3. with several threads executing, throughput is greatly increased (at least when *divergence* is low)
4. user only needs to write code like a single thread program
5. frequent switching means that average latency for any one thread to finish is high, and high latencies permit the use of relatively slower, cheaper registers, caches etc. That is why GPUs can have hundreds of registers per core.

We have seen above that with favourable workloads, SIMT sees large performance gains, but at the same time it has its share of downsides too:

1. Divergence can severely limit performance in the case of conditional code. Locksteps threads always execute both sides of a branch, which means that not all threads in a warp are doing useful work.
2. Divergence can also slow down memory accesses, because different threads might access locations in the same memory bank, leading to contention. High contention will effectively serialize memory accesses, severely affecting throughput.
3. SIMT is targeted at massively parallel workloads. If the workload does not require a lot of threads to run, performance gains from SIMT are underwhelming because it cannot hide latency without a lot of switching.
4. SIMT threads do not support any other synchronization primitives apart from the already inherent ‘lockstep’ synchronization.



Flow Divergence in SIMT

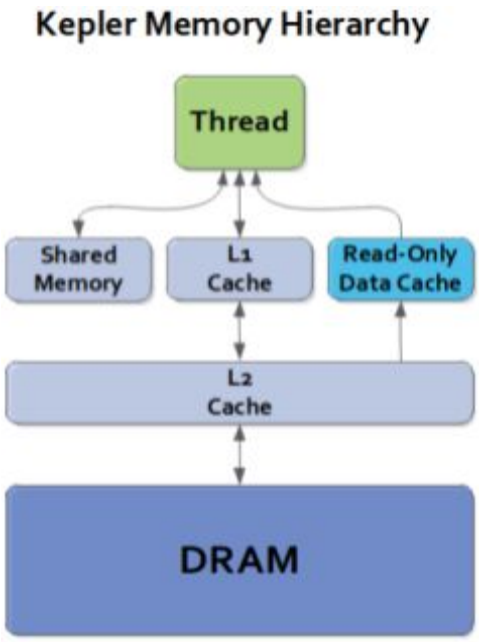


Memory Contention in SIMT

Memory Subsystem

We have seen in detail how a GPU makes heavy use of various forms of parallelism to scale up its computational power. However, no amount of parallelism can compute anything if the data does not reach the processors fast enough. Graphics workloads typically are highly memory intensive, requiring millions of accesses to and from memory per second. The memory subsystem is the second most importance determiner of performance in a GPU apart from the GPU itself.
So modern GPUs have very wide memory buses (around 200 GB/s) and have a special memory hierarchy to reduce their bandwidth requirements, which is explained below:

Registers: A GPU has thousands of on-chip registers, and hundreds of registers are local to a single thread. Registers offer the fastest latency for data access to a thread, and are usually used to store local variables. Do note that while GPU have many more registers, they are made of simple hardware and are not as fast as CPU registers. (otherwise GPU costs would skyrocket). The kepler architecture allows upto 255 registers per thread, for a total of 65,536 registers per multiprocessor!



Shared Memory: Each block of threads has an exclusive shared on-chip memory that it uses for communication between threads, and it has almost similar latency to registers. The purpose is mainly to offload accesses from the off-chip main memory which is high latency. The kepler architecture has 64kB of on-chip, which can be split between shared memory and L1-cache. Shared memory is managed by the software.

Global Memory: Global memory is the main DRAM memory, is visible to entire grids, and can be arbitrarily written by the GPU or even the CPU. It is slow to access (hundreds of cycles latency) and not cached, but coalescing allows slightly faster performance.

Caches: The primary purpose of caches in GPUs, unlike in CPUs, is not to reduce latency but to increase throughput. Latency reduction exploits temporal/spatial locality which is already addressed by use of shared memory between threads.
The L1 cache in a GPU is used to coalesce requests for the same block by different threads. It keeps a block in cache just long enough for each thread to hit it once, hence reducing bandwidth to main memory.
Modern GPUs also have a high-bandwidth L2 cache which serves as the single point of high-speed data unification across different SMX units in the GPU and acts as a staging buffer

for DRAM. Applications that have incoherent access patterns, or those which execute same data on multiple SMX units can greatly benefit because of the L2 cache.
Additionally, GPUs have a read-only on-chip cache which is used to store frequently-accessed constants,kernel arguments etc and can be read by any block.

GPU Architecture

Now let's have a closer look at the actual architecture of a GPU device. nVidia's GeForce 600 and GeForce 700 series were based on the nVidia Kepler Streaming Multiprocessor (SMX). Each SMX unit features 192 single-precision CUDA cores, 64 double-precision units, 32 special function units(SFUs) and 32 load/store units. It also some small on-chip memories and caches. Additionally, an SMX can also directly access off-chip DRAM memory. A high-end GPU can have many such SMX units on a single device.

Hardware multithreading: In terms of GPU computing, a kernel is a parallel section of application code that is run on the GPU by a thread. A thread block is a batch of threads that cooperate with each other. Threads within a block share memory and execute the same kernel. A block is basically a collection of 64 warps(32 lockstep threads each) which can be interleaved.A grid is a collection of blocks that can run in parallel. Number of concurrent grids on a GPU is limited by the exact implementation and use case.
Parallel workloads are partitioned into blocks across many multiprocessors. Each block is assigned a unique ID, and within each block, thread IDs are unique. Looking at raw figures, a single multiprocessor can hold upto 2048 threads at a time, but the hardware scheduler imposes limits on how many threads can run simultaneously. An SMX unit has 4 warp schedulers, so at each clock cycle, it can select upto 4 warps, and upto 2 independent instructions per warp can be issued and executed.
To support rapid switching, the Kepler architecture allows each thread to have 255 registers of its own, which prevents 'spilling' of registers to local memory. To reduce incoherent memory accesses, multiple small memory accesses within a block are *coalesced* into a single large memory request. Irregular access patterns between threads can(eg. in case of nested branches) still lead to highly random access times, though.

Double Precision Units: It should be noted that while CUDA cores support fully pipelined single-precision operations, and are highly compliant with the IEEE 754 floating-point spec, with having support for even advanced operations like *multiply-add,atomic maximum* etc, double-precision arithmetic in GPUs is still catching on. The DP units are slower, and the number of DP units on chip is much lesser compared to FP units, leading to generally lower performance with DP-based applications. (DP can be in the worst case be 4x slower, although newer architectures have improved this a lot)

Special Function Units: The SFUs are specialised computation units present in GPUs which can compute results of complex mathematical operations like reciprocal, sin, cos, log, exp, etc

on 32-bit floating point numbers in a single cycle. With a heavy mathematical workload, presence of SFUs can mean upto a 50% performance increase.

GPGPUs and CUDA

GPGPU (**G**eneral-**P**urpose **GPU**) is the use of a GPU to do general purpose computing. The basic model for GPU computing is for a CPU to at as a host and issue parallel workloads to the GPUs, thus forming a powerful heterogeneous co-processing computing model. The sequential part of the code is run by the CPU and the computationally parts are accelerated by the GPUs.

One realization that this model draws upon is that even though GPUs have theoretically much higher teraflops than a CPU, they are still bad with low-latency sequential processing, and the GPU driver which is a complex program in itself, invariably needs a CPU to run it. In recent years, heterogeneous architectures have been gaining traction. These chips have multiple CPU and GPU cores on the same die and can aggressively optimize for latency,throughput and power usage by adapting to the workload. (eg. AMD APUs, mobile SoCs)

CUDA(**C**ompute **U**nified **D**evice **A**rchitecture) is a scalable parallel programming model and software environment for parallel computing developed by nVidia. Naturally, it only works with nVidia GPUs. It is the first serious serious attempt to bring GPUs into the field of general-purpose computing, and is one of the most successful parallel computing platforms.

CUDA C/C++ consists of extensions to the familiar C/C++ environment, and allows programmers to write computing-intensive C/C++ programs that can effectively make use of the heterogeneous CPU+GPU architecture. A typical CUDA program does the following:

1. Declare and allocate host and device memory.
2. Initialize host data and transfer data from the host to the device.
3. Execute one or more kernels on the device.
4. Transfer results from the device to the host.

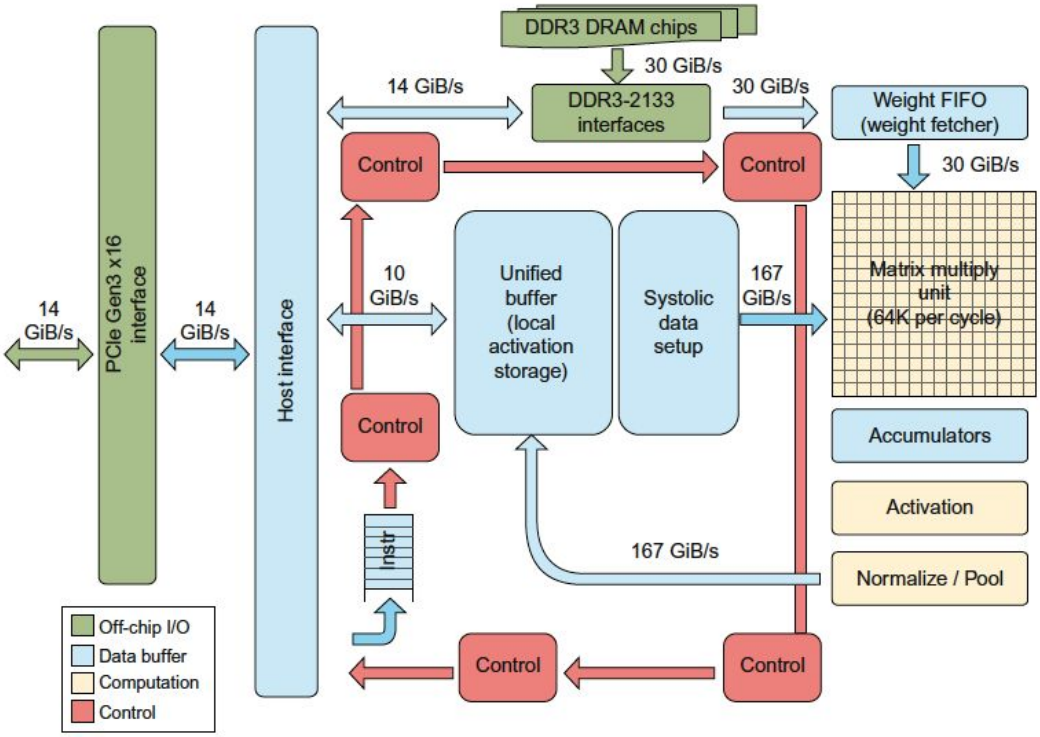
A main challenge to the programmer in writing CUDA programs is how to cleverly split the data across a large number of threads.
A CUDA program is unified source code for both host and the device. CUDA programs are compiled using the proprietary NVCC compiler. It separates the device functions from the host code, compiles the device functions using the proprietary NVIDIA compilers and assembler, compiles the host code using a standard compiler like GCC, and afterwards embeds the compiled GPU functions as binary images in the host object file. In the linking stage, specific CUDA runtime libraries are added for supporting remote SPMD procedure calling and for providing explicit GPU manipulation such as allocation of GPU memory buffers and host-GPU data transfer.

Tensor Processing Units (TPUs)

Introduction

A TPU (Tensor Processing Unit) is an Application Specific Integrated Circuit (ASIC) designed by Google in 2015 specifically for Neural Networks. They are being used at several Google applications like Google Photos and Page Rank. So far, three generations of TPUs have been announced by Google with performance increasing from 92 teraops to more than 100 petaflops. The main reason for their high performance is its Matrix Multiply Unit and a large high bandwidth memory. TPUs generally use bfloat16 floating-point format for floating point operations rather than using IEEE standard because neural networks do not require high precision but involve a large number of parallel operations.

TPU (v1) Architecture



1. **UB - Local Unified Buffer for activation:** A 24 MiB on-chip memory that holds the activations. It was sized to try to avoid spilling activations to DRAM when running a DNN(Deep Neural Network) (MiB = 2^20 bytes). The host can easily access this to write

new inputs. It is directly connected to the MXU. It holds upto 384 256 X 256 activation matrices.

2. **MXU (Matrix Multiplication Unit):** A systolic array of 256 X 256 8-bit arithmetic units that perform multiply-add operations. Its inputs are the Weight Memory and the Unified Buffer, and its output is the Accumulators.

3. **Accumulators:** These are 4096 25632-bit registers (4 MiB). The MXU writes back to the Unified Buffer through Accumulators.

4. **DDR3 DRAM:** This stores the weights that are required for the computations of activations. The weights are read into a Weight FIFO at the start of computations. This is a cache that is off the chip. The weights are stored in this because they are required to be loaded once per batch and hence can be put in this slow buffer.

5. **Activation Pipeline:** This pipeline applies standard neural network functions, like ReLu and Maxpool, that are not very computationally expensive. It applies these on the data collected from the MXU by the Accumulators and outputs them in the UB. However in the second version of the TPU, this was replaced by full vector and scalar units to give it a much wider range of available functions.

6. **Control:** This takes instructions from the host. This tells the MXU when to multiply, which weights to load and what operations the Activation Pipeline will perform.

7. **Host Interface:** This interface acts as a passage for three forms of data - weights(to the DDR3), activations(to the Unified Buffer) and control instructions(to the control). This interface connects to the controlling machine via the PCIe.

8. **PCIe (Peripheral Connect Interface Express):** The TPU is connected to its host via PCIe interface that can provide an effective bandwidth of 12.5GBps. Instructions are sent over this. This interface implements the PCIe bus standard (a high speed serial expansion bus standard) which is a point to point topology. Compared to other older bus standards, this has a higher maximum system bus throughput, better error detection, better performance scaling .etc.

TPU instruction set architecture

TPU instructions follow the CISC format, including a repeat field because the instructions are sent over the slow PCIe bus. TPUs don’t have a program counter or branch instructions. CPI of these instructions are typically 10-20.

These are the five main instructions:

1. **RHM - Read_Host_Memory:**
   RHM *src, dst*, N - reads N vectors from host memory at *src* and stores them at Unified Buffer starting at *dst*.

2. **RW - Read_Weights:** This instruction reads weights from the DDR3 DRAM into the Weight FIFO as input to the Matrix Unit. Syntax: RW *address*

3. **MMC - MatrixMultiply/Convolve:** Performs multiplication of matrices with vectors, matrices with matrices, element wise multiplication and convolutions from unified buffer to accumulator . A matrix operation takes an input(256*N), multiplies it by a 256 by 256 constant input, and produces a 256*N output, taking N pipelined cycles to complete comparatively. This instruction has the UB and Accumulator addresses.
   Syntax: MMC.{OS} *src, dst*, N - Here O is the overwrite flag and S is the switch flag(switch to using the next tile of weights, which must have already been pre-loaded). The first MMC instruction in a program should always have the S flag.

4. **ACT - Activate:** Applies the nonlinear activation functions of neural nodes, with options for ReLU, Sigmoid, tanh, etc. Takes in input from the Accumulators, and writes its output in the Unified Buffer.
   Syntax: ACT.{RQ} *src, dst*, N - Here R is for the ReLu operation and Q is for the Sigmoid operation for the activation. If nothing is specified, no activation is done.

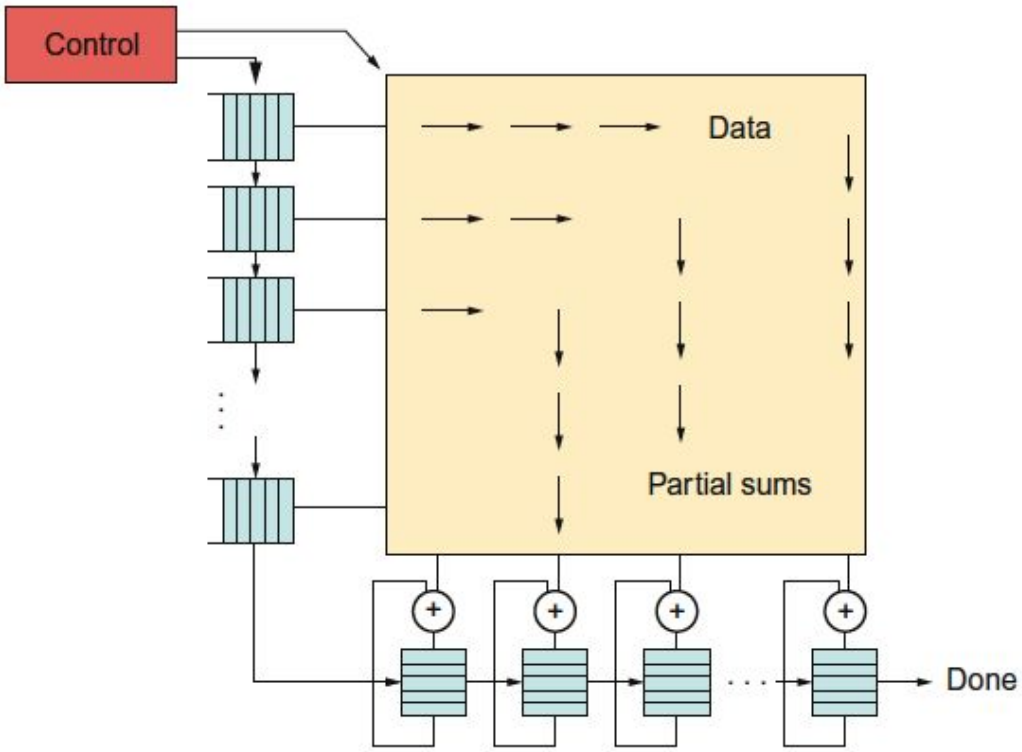
5. **WHM - Write_Host_Memory:** WHM *src, dst*, N - writes N vectors from the Unified Buffer at *src* to host memory at *dst.*

Other instructions include NOP (No operation for a few clock cycles), HLT - halt (stops the simulation), debug-tap, interrupt host, set configuration, host memory read/write (alternatives to the main instructions) and synchronization instructions.

TPU microarchitecture philosophy

The general plan is to mask other instructions behind the weighty Matrix Multiply instructions. Each of the above four classes of instructions has separate execution hardware, with Read_Host_memory and Write_Host_memory combined into one. This increases instruction parallelism.
Reading from and writing to the Unified Buffer(SRAM) is more expensive than arithmetic, so we use systolic arrays to reduce the number of reads and writes. **A systolic array is a two dimensional collection of arithmetic units that each independently compute a partial result as a function of inputs from other arithmetic units that are considered upstream to**

each unit. It relies on data from different directions arriving at cells in an array at regular intervals where they are combined.



The systolic data flow is described in the above diagram. A 256 element vector propagates from the left as a diagonal wavefront. At each systolic node, the weight is multiplied and it propagates to the right and bottom. In the end, all results are obtained at the bottom array.
Control and data are pipelined to give the illusion that the 256 inputs are read at once, and after a feed delay, they update one location of each of 256 accumulator memories. The software is unaware of the systolic design of the MMU.

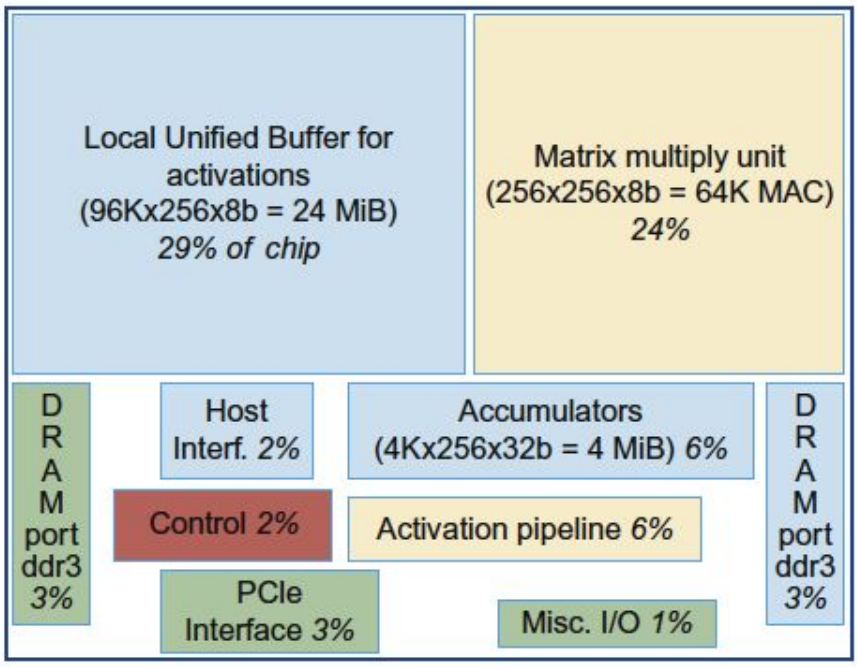
TPU software

The TPU software stack had to be compatible with that developed for CPUs and GPUs so that applications could be ported quickly. The part of the application run on TPU is written using TensorFlow developed by Google and compiled into an API.

The TPU stack is split into User Space Driver and Kernel Driver. The lightweight kernel driver manages memory and interrupts. It is designed for long term stability. The user space driver sets up and controls TPU execution, reformats data into TPU order, and translates API calls into TPU instructions and turns them into a binary. It compiles a model the first time it is evaluated, caches the model and writes the weights into the TPU weight memory.
The TPU runs most models completely from inputs to outputs, maximizing the ratio of TPU

compute time to I/O time.

TPU implementation



This figure illustrates the floor plan of the TPU die.

The 24 MiB Unified Buffer is about one third of the die, while the Matrix Multiply Unit is about a quarter. Thus datapath is almost ⅔ of the die. Control is only 2% of the die.The unused white space is a consequence of the emphasis on time to tape-out for the TPU.

Optimizations of TPUs

1. **Quantization:** Quantization is a method to handle floating point operations via integer operations. This significantly reduces the computational cost of multiplication.Machine learning applications typically don't require high precision, so 8 bit integers are used instead of the normal 32 bit numbers. Quantization considerably reduces the hardware and energy consumption of the TPU.

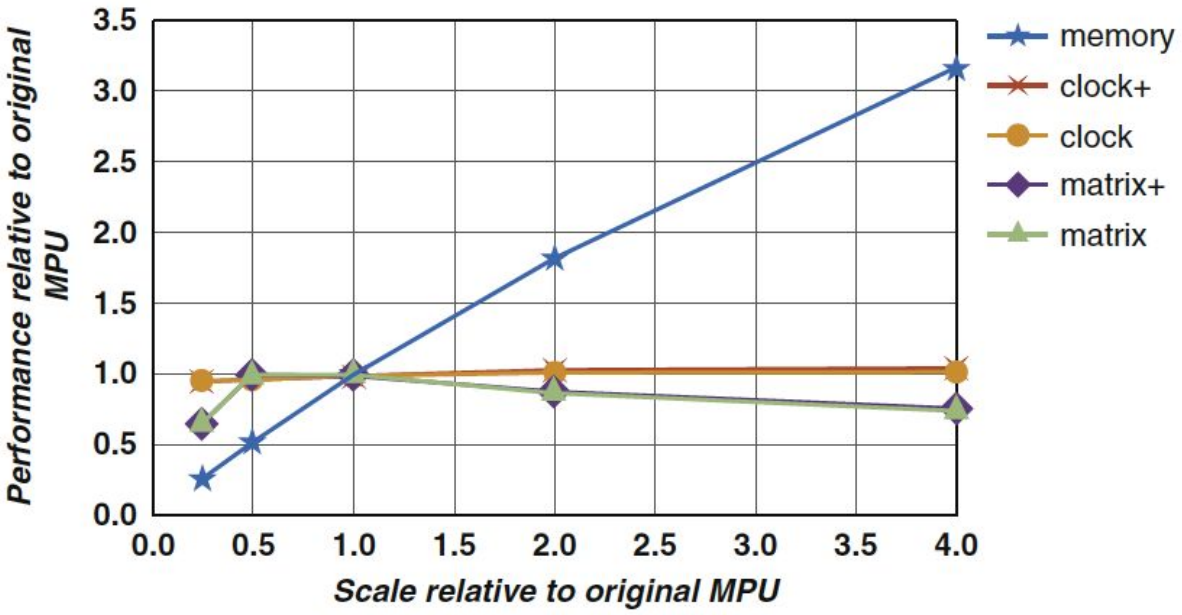
2. **Parallel Matrix Processing:** MXU of TPU can perform hundreds of thousands of operations (matrix operation) in a single clock cycle, as compared to a scalar operation of a CPU or a vector operation of a GPU.To implement this, TPUs use **systolic arrays,** much different from CPU/GPU architecture.Systolic arrays chains multiple ALUs

together, saving on the storing and retrieving from registers that CPUs/GPUs do.They are thus less general than CPUs, but better at what they are optimized to do.

3. **Dedicated memories to minimize access time:** The 24 MiB unified buffer is used to store intermediate matrices, vectors and feature maps. It is optimized for accessing 256 bytes at a time. It has 4MiB accumulators that collect the output of the matrix unit and act as input to nonlinear functions. The 8-bit weights are stored in a separate off-chip DRAM and are accessed via a weight FIFO.

4. **Minimalistic design:** TPUs do not have sophisticated features like caches, branch prediction, out of order execution, etc.This allows us to allocate much more space to memory and ALU and less to control.

Further improvements in the TPUs



TPU architects created a performance model and tested various parameters:

1. It is observed that increasing clock rate and accumulators do not significantly increase performance.
2. Increasing memory bandwidth (memory) significantly increases performance: it improves 3x on average when memory bandwidth increases 4x , because it reduces the waiting time for weight memory.
3. Performance degrades slightly when the matrix unit is expanded from 256 x 256 to 512 x 512, irrespective of accumulators.

The focus is thus on increasing memory bandwidth. GDDR5 can be used for this.

Comparison and Summary

- While GPUs are optimized for vector processing, TPUs go a step higher and perform matrix operations with ease.
- GPUs and TPUs both heavily rely on parallel processing.
- TPUs are much more narrowly optimized than GPUs, thus TPUs work much better at the tasks which they were designed to perform (application specific). However, GPUs perform a wider variety of tasks satisfactorily.
- Both TPUs and GPUs devote more space to computational resources and less to control logic.
- Both use domain specific languages, like TensorFlow for TPUs and CUDA and OpenCL for GPUs.

  To implement parallelization, GPUs and TPUs use different methods. TPUs use systolic arrays in the MMU for matrix multiplication, while GPUs use parallelism at different stages to get the maximum time benefits in addition to superscalar/vector processing. In this way, both TPUs and GPUs exploit parallelism and gain significant advantages over CPUs.

References:

1. Computer Graphics Slides [https://www.cs.cmu.edu/afs/cs/academic/class/15462-f11/www/lec_slides/lec19.pdf]
2. Introduction to CUDA [https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/]
3. nVidia Kepler Architecture Whitepaper [https://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf]
4. nVidia CUDA toolkit Docs [https://docs.nvidia.com/cuda/]
5. GPGPU Processing in CUDA Architecture [https://arxiv.org/pdf/1202.4347.pdf]
6. An Introduction to Modern GPU Architecture [http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf]
7. From Shader Code to a TeraFlop [http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf]
8. GPU Architectures - A CPU Perspective [https://courses.cs.washington.edu/courses/cse471/13sp/lectures/GPUsStudents.pdf]
9. Parallel programming: Introduction to GPU architecture [http://www.irisa.fr/alf/downloads/collange/cours/gpuprog_ufmg/gpuprog_1.pdf]
10. Patterson and Hennessy: Computer Architecture, A Quantitative Approach.
11. In-Datacenter Performance Analysis of a Tensor Processing Unit [https://arxiv.org/pdf/1704.04760.pdf]

12. An in-depth look at Google's first Tensor Processing Unit (TPU) [https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu]
13. TPU Unit Reference Manual

    [https://www.nxp.com/docs/en/reference-manual/TPURM.pdf]

Contributions:

For report

1. **Yash Gupta:** GPUs (from Parallelism in GPUs to GPGPUs and CUDA)
2. **Mayank Singhal:** GPUs (Introduction, Graphics Pipeline) and TPUs (Introduction)
3. **Sriram Balasubramanian :** TPUs (from microarchitecture philosophy to Comparison)
4. **Anirudh Arputham:** TPUs (TPU Architecture and TPU instruction set architecture)
5. **Ashutosh Kumar Verma:** TPUs (TPU instruction set architecture)

For presentation

1. **Sriram Balasubramanian** (TPUs and summary)
2. **Mayank Singhal** and **Anirudh Arputham** (GPUs and PPT style)