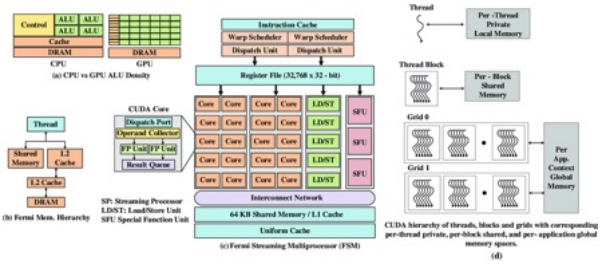
Christof Teuscher

ECE 410/510: Hardware for AI and ML

TPU + CNN/DNN + Transformers

Portland State University
Department of Electrical and Computer Engineering (ECE)
www.teuscher-lab.com
teuscher@pdx.edu

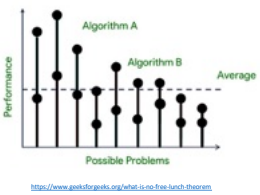
NVIDIA Fermi Microarchitecture



10.1109/ACCESS.2020.3008250

No Free Lunch theorem (NFL)

- The "No Free Lunch" (NFL) theorem in optimization and machine learning states that, when averaged across all possible problems, all optimization algorithms perform equally well. I.e., **no algorithm consistently outperforms random search or any other algorithm.**
- The NFL theorem was formalized by David Wolpert and William Macready in 1997.
- **This principle has profound implications for ML and HW optimization:**
  - It explains why specialized algorithms/architectures work well for specific domains but fail in others.
  - It highlights the importance of incorporating domain knowledge when selecting algorithms/architectures.
  - It reminds us that claims about algorithm/architecture superiority must specify the context/problem class.



https://www.geeksforgeeks.org/what-is-no-free-lunch-theorem

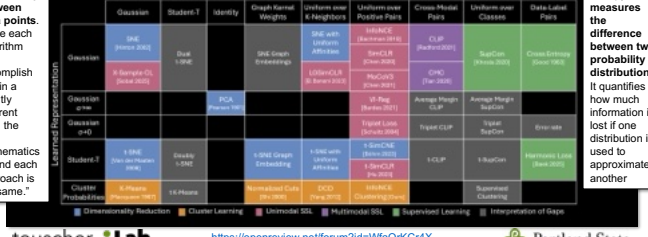
https://ieeexplore.ieee.org/document/585893

Discovering a Periodic Table for Machine Learning

"All these algorithms learn a **specific kind of relationship between data points**. While each algorithm may accomplish that in a slightly different way, the core mathematics behind each approach is the same."

Generalized loss function for all algorithms

$$\mathcal{L} = \int KL(p(j|i) \,||\, q(j|i))\, di$$



Kullback-Leibler (KL) divergence, also known as relative entropy, **measures the difference between two probability distributions**. It quantifies how much information is lost if one distribution is used to approximate another

https://openreview.net/forum?id=WfaQrKCr4X

Tensor cores (1)

Each Tensor Core operates on a 4x4 matrix and performs the following operation:

D = A×B + C

where **A**, **B**, **C**, and **D** are 4x4 matrices (Figure 8). The matrix multiply inputs **A** and **B** are FP16 matrices, while the accumulation matrices **C** and **D** may be FP16 or FP32 matrices (see Figure 8).

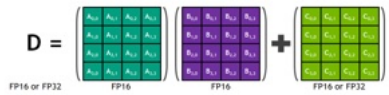


Figure 8. Tensor Core 4x4 Matrix Multiply and Accumulate

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Tensor cores (2)

Tensor Cores operate on FP16 input data with FP32 accumulation. The FP16 multiply results in a full precision product that is then accumulated using FP32 addition with the other intermediate products for a 4x4x4 matrix multiply (see Figure 9). In practice, Tensor Cores are used to perform much larger 2D or higher dimensional matrix operations, built up from these smaller elements.

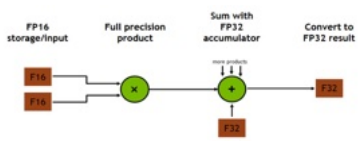


Figure 9. Mixed Precision Multiply and Accumulate in Tensor Core

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Another example

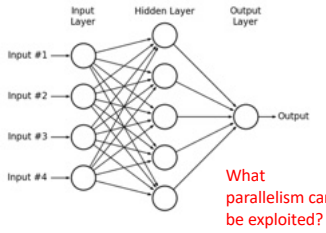
SAXPY stands for "Single-precision A*X Plus Y"

```
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);

  cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
}
```

https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c

How to map a neural network on a GPU?

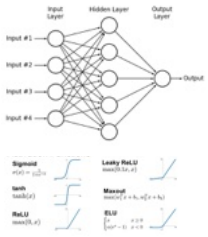
Multi-layer perceptron (MLP)



What parallelism can be exploited?

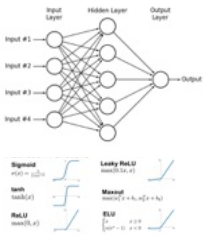
Multi-layer perceptron (MLP)

- **Data parallelism**: You can process multiple training examples simultaneously. Since each example is processed independently through the forward pass, you can distribute batches across different computing units.
- **Model parallelism**: The network itself can be partitioned, with different parts of the model assigned to different processors. This works particularly well for very large networks.
- **Layer-wise parallelism**: Within each layer, the computations can be parallelized since neurons in the same layer operate independently.



Multi-layer perceptron (MLP)

- **Matrix multiplications**: The core operation in MLPs (Wx + b) is highly parallelizable. Each output neuron's computation is independent.
- **Activation functions**: These are applied element-wise and can be computed in parallel across all neurons.
- **Backpropagation**: During training, gradient computations can be parallelized both within and across layers.

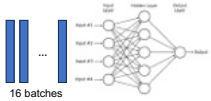


CUDA MLP (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <time.h>

// Dimensions of our network
#define INPUT_SIZE 4
#define HIDDEN_SIZE 5
#define OUTPUT_SIZE 1
#define BATCH_SIZE 16  // Process 16 samples at once

// Activation function (ReLU)
__device__ float relu(float x) {
    return x > 0 ? x : 0;
}
```

ReLU

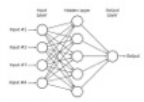
$\max(0, x)$



CUDA MLP (2)

```
// Forward pass kernel for first layer (input -> hidden)
__global__ void layer1_forward(float *input, float *weights, float *bias, float *output, int batch_size) {
    int batch_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int neuron_idx = blockIdx.y * blockDim.y + threadIdx.y;

    if (batch_idx < batch_size && neuron_idx < HIDDEN_SIZE) {
        float sum = bias[neuron_idx];
        for (int i = 0; i < INPUT_SIZE; i++) {
            sum += input[batch_idx * INPUT_SIZE + i] * weights[i * HIDDEN_SIZE + neuron_idx];
        }
        output[batch_idx * HIDDEN_SIZE + neuron_idx] = relu(sum);
    }
}

// Forward pass kernel for second layer (hidden -> output)
__global__ void layer2_forward(float *input, float *weights, float *bias, float *output, int batch_size) {
    int batch_idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (batch_idx < batch_size) {
        float sum = bias[0];
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            sum += input[batch_idx * HIDDEN_SIZE + i] * weights[i];
        }
        output[batch_idx] = sum;  // No activation on output layer in this example
    }
}
```

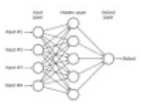
Global thread ID (x coordinate)
Global y coordinate



CUDA MLP (3)

Why separate kernels for each layer?

- **Memory optimization**: Each layer has different memory access patterns and working set sizes. Separate kernels allow you to optimize memory usage specifically for each layer's requirements rather than trying to find a one-size-fits-all approach.
- **Parallelism control**: Different layers may benefit from different thread configurations. For example, the first layer in our implementation handles 4 inputs to 5 neurons, while the output layer handles 5 inputs to 1 neuron - these naturally map to different parallelization strategies.
- **Resource utilization**: GPU resources like shared memory, registers, and thread counts can be tailored specifically to each layer's computational needs, which improves overall efficiency.
- **Kernel fusion limitations**: While fusing operations into a single kernel can reduce launch overhead, there are practical limits to how much computation can be packed into one kernel before it becomes inefficient. Separate kernels help manage this complexity.
- **Synchronization points**: Having distinct kernels creates natural synchronization points between layers, ensuring all computations from one layer are complete before the next layer begins processing.
- **Debugging and profiling**: With separate kernels, it's easier to identify performance bottlenecks or errors in specific parts of the network.
- **Specialized optimizations**: Different mathematical operations may benefit from specialized implementations - convolutions, fully-connected layers, and activation functions all have different optimal implementations on GPUs.



CUDA MLP (4)

How much should be in a kernel?

The optimal kernel size involves balancing several trade-offs:

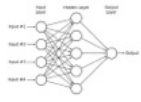
- **Kernel launch overhead**: Each kernel launch incurs some overhead. Too many small kernels can result in performance degradation due to this overhead.
- **Register pressure**: Larger kernels with more operations typically require more registers per thread, which can reduce occupancy (the number of threads that can run simultaneously on the GPU).
- **Instruction cache**: GPUs have limited instruction cache. Very large kernels might not fit well in the cache, causing additional latency.
- **Memory bandwidth utilization**: Kernels that load data, perform a small operation, and then store results can be memory-bound. Combining multiple operations in one kernel can improve arithmetic intensity and make better use of compute resources.
- **Thread divergence**: If your kernel contains many conditional branches that cause different threads to follow different execution paths, performance will suffer.

As a general guideline:

- **Good candidates for fusion**: Operations that work on the same data and have similar parallelization patterns (like a matrix multiplication followed by an element-wise activation function).
- **Better kept separate**: Operations with fundamentally different access patterns or parallelization strategies.

CUDA MLP (5)

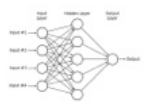
```
// Initialize weights with random values between -0.5 and 0.5
void initialize_weights(float *h_weights, int size) {
    for (int i = 0; i < size; i++) {
        h_weights[i] = ((float)rand() / RAND_MAX) - 0.5f;
    }
}

int main() {
    // Seed random number generator
    srand(time(NULL));

    // Host memory allocation
    float h_input[BATCH_SIZE * INPUT_SIZE];
    float h_hidden[BATCH_SIZE * HIDDEN_SIZE];
    float h_output[BATCH_SIZE * OUTPUT_SIZE];
    float h_weights1[INPUT_SIZE * HIDDEN_SIZE];
    float h_bias1[HIDDEN_SIZE];
    float h_weights2[HIDDEN_SIZE * OUTPUT_SIZE];
    float h_bias2[OUTPUT_SIZE];

    // Initialize weights and biases
    initialize_weights(h_weights1, INPUT_SIZE * HIDDEN_SIZE);
    initialize_weights(h_bias1, HIDDEN_SIZE);
    initialize_weights(h_weights2, HIDDEN_SIZE * OUTPUT_SIZE);
    initialize_weights(h_bias2, OUTPUT_SIZE);

    // Initialize some example input data
    for (int i = 0; i < BATCH_SIZE * INPUT_SIZE; i++) {
        h_input[i] = ((float)rand() / RAND_MAX);
    }
```



CUDA MLP (6)

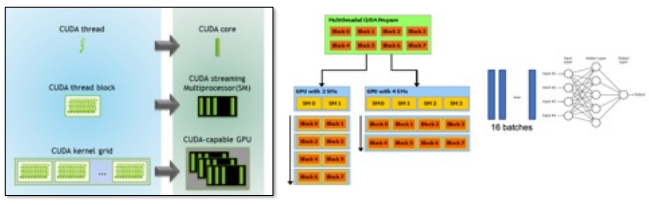
```
// Host memory allocation
float h_input[BATCH_SIZE * INPUT_SIZE];
float h_hidden[BATCH_SIZE * HIDDEN_SIZE];
float h_output[BATCH_SIZE * OUTPUT_SIZE];
float h_weights1[INPUT_SIZE * HIDDEN_SIZE];
float h_bias1[HIDDEN_SIZE];
float h_weights2[HIDDEN_SIZE * OUTPUT_SIZE];
float h_bias2[OUTPUT_SIZE];
```

```
// Device memory allocation
float *d_input, *d_hidden, *d_output;
float *d_weights1, *d_bias1, *d_weights2, *d_bias2;

cudaMalloc(&d_input, BATCH_SIZE * INPUT_SIZE * sizeof(float));
cudaMalloc(&d_hidden, BATCH_SIZE * HIDDEN_SIZE * sizeof(float));
cudaMalloc(&d_output, BATCH_SIZE * OUTPUT_SIZE * sizeof(float));
cudaMalloc(&d_weights1, INPUT_SIZE * HIDDEN_SIZE * sizeof(float));
cudaMalloc(&d_bias1, HIDDEN_SIZE * sizeof(float));
cudaMalloc(&d_weights2, HIDDEN_SIZE * OUTPUT_SIZE * sizeof(float));
cudaMalloc(&d_bias2, OUTPUT_SIZE * sizeof(float));

// Copy data from host to device
cudaMemcpy(d_input, h_input, BATCH_SIZE * INPUT_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_weights1, h_weights1, INPUT_SIZE * HIDDEN_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_bias1, h_bias1, HIDDEN_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_weights2, h_weights2, HIDDEN_SIZE * OUTPUT_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_bias2, h_bias2, OUTPUT_SIZE * sizeof(float), cudaMemcpyHostToDevice);
```



CUDA MLP (7)

How will we use threads for this MLP?



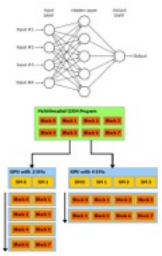
A GPU is built around an **array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other**.

CUDA MLP (8)

```
// Set up execution configuration for first layer
dim3 threadsPerBlock1(8, 5);  // 8 batches, all 5 hidden neurons
dim3 blocksPerGrid1((BATCH_SIZE + threadsPerBlock1.x - 1) / threadsPerBlock1.x,
                    (HIDDEN_SIZE + threadsPerBlock1.y - 1) / threadsPerBlock1.y);

// Set up execution configuration for second layer
dim3 threadsPerBlock2(16, 1);  // 16 batches
dim3 blocksPerGrid2((BATCH_SIZE + threadsPerBlock2.x - 1) / threadsPerBlock2.x, 1);

// Launch kernels
layer1_forward<<<blocksPerGrid1, threadsPerBlock1>>>(d_input, d_weights1, d_bias1, d_hidden, BATCH_SIZE);
layer2_forward<<<blocksPerGrid2, threadsPerBlock2>>>(d_hidden, d_weights2, d_bias2, d_output, BATCH_SIZE);

// Check for errors
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    printf("CUDA Error: %s\n", cudaGetErrorString(err));
    return -1;
}

// Copy results back to host
cudaMemcpy(h_output, d_output, BATCH_SIZE * OUTPUT_SIZE * sizeof(float), cudaMemcpyDeviceToHost);

// Print some outputs
printf("Sample outputs from MLP:\n");
for (int i = 0; i < 5; i++) {  // Print first 5 results
    printf("Sample %d: %.6f\n", i, h_output[i]);
}
```

Why?

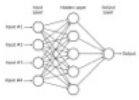
kernel<<M,T>>
The kernel launches with a grid of M thread blocks.
Each thread block has T parallel threads.



A GPU is built around an **array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other**.

CUDA MLP (9)

```
// Free GPU memory
cudaFree(d_input);
cudaFree(d_hidden);
cudaFree(d_output);
cudaFree(d_weights1);
cudaFree(d_bias1);
cudaFree(d_weights2);
cudaFree(d_bias2);
```



Deep learning frameworks

1. Deep learning frameworks provide interfaces to implement AI functionalities regardless of hardware or computing platforms.
2. Most deep learning frameworks also optimize the network in a **hardware-friendly manner** and utilize the hardware resources and acceleration functions as much as possible.
3. Because GPUs are commonly used in deep learning, deep learning frameworks **widely support constructing and deploying neural networks on GPU** and actively reflect the acceleration support from GPU vendors.
4. In addition, they provide optimizations on standalone accelerators such as **TPU** and specific environments such as cloud servers.

TensorFlow

Mishra, Ashutosh; Cha, Jaekwang; Park, Hyunbin; Kim, Shiho. Artificial Intelligence and Hardware Accelerators, 2025

PyTorch & CNNs

For CNNs, the mapping process involves:

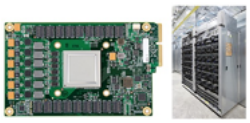
1. **Tensor operations**: PyTorch represents all data (images, weights, activations) as tensors that can be moved between CPU and GPU memory.
2. **Layer-by-layer optimization**: Each CNN layer (convolution, pooling, etc.) is mapped to specific GPU kernels that efficiently perform the required operations in parallel.
3. **Memory management**: PyTorch handles the complex memory allocations and transfers between host (CPU) and device (GPU) memory.
4. **Computation graphs**: PyTorch builds a dynamic computation graph that tracks operations and enables automatic differentiation for training.
5. **Batching**: Multiple inputs are processed simultaneously to maximize GPU utilization.
6. For the **convolution operations** specifically, PyTorch typically uses highly optimized libraries like cuDNN that implement various algorithms (direct convolution, FFT-based, Winograd, etc.) and automatically select the most efficient one based on input size and available GPU resources.

PyTorch

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define the network architecture
class SimpleMLPNet(nn.Module):
    def __init__(self):
        super(SimpleMLPNet, self).__init__()
        # First layer: 4 input neurons -> 5 hidden neurons
        self.fc1 = nn.Linear(4, 5)
        # Second layer: 5 hidden neurons -> 1 output neuron
        self.fc2 = nn.Linear(5, 1)
        # ReLU activation for hidden layer
        self.relu = nn.ReLU()

    def forward(self, x):
        # Forward pass through first layer and apply ReLU
        x = self.relu(self.fc1(x))
        # Forward pass through output layer (no activation)
        x = self.fc2(x)
        return x
```

```
# Create a batch of random input data (10 samples, 4 features each)
batch_size = 16
input_size = 4
x = torch.rand(batch_size, input_size)

# Create the network
model = SimpleMLPNet()

# Move everything to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
x = x.to(device)

# Forward pass
with torch.no_grad():  # No need to track gradients for inference
    output = model(x)

# Print some sample outputs
print("Sample outputs from MLP:")
for i in range(5):  # Print first 5 results
    print(f"Sample {i}: {output[i].item():.6f}")
```


GPU vs Tensor Processing Unit (TPU)



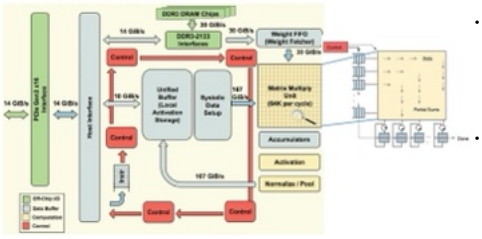
GPU vs TPU

	GPU	TPU
Origin and design purpose	Graphics	Created by Google for ML
Architecture	Thousands of smaller cores designed for parallel processing.	More specialized architecture with matrix processing units (MXUs) specifically optimized for tensor operations.
Performance focus	More flexibility, broader applicability	Optimized for matrix operations and ML workloads.
Development and availability	NVIDIA, AMD, etc.	Proprietary, only through Google cloud
Software compatibility	Various frameworks, e.g., CUDA	TensorFlow
Power efficiency	Less power-efficient, but more flexibility	More power-efficient

TPU key features

- **Matrix Multiplication Unit** (MXU): The central component designed to perform matrix operations quickly. It loads parameters and data from high-bandwidth memory (HBM), executes multiplications, and passes results to the next multiply-accumulator.
- **TensorCores**: Each TPU chip contains one or more TensorCores, with the number depending on the version. Each TensorCore consists of one or more matrix-multiply units, a vector unit, and a scalar unit.
- **Activation Unit** (AU): Provides hardwired activation functions commonly used in neural networks

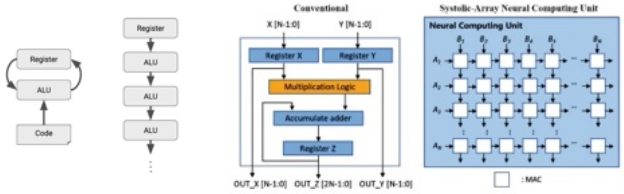
Architecture of a TPU accelerator



- The main component of TPU is the MXU that consists of 256 by 256 (i.e., 65,536) MACs to perform 8-bit mathematical operations.
- The MXU gets its input from the weighted FIFO and unified buffer (UB).

https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

Systolic array (1)

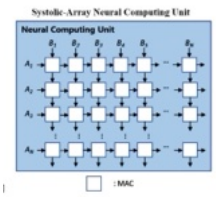


- The name "systolic" comes from an analogy to the human heart, where data pulses through the array of processors in a rhythmic, synchronized fashion.
- CPUs and GPUs often spend energy to access multiple registers per operation. A systolic array chains multiple ALUs together, reusing the result of reading a single register.

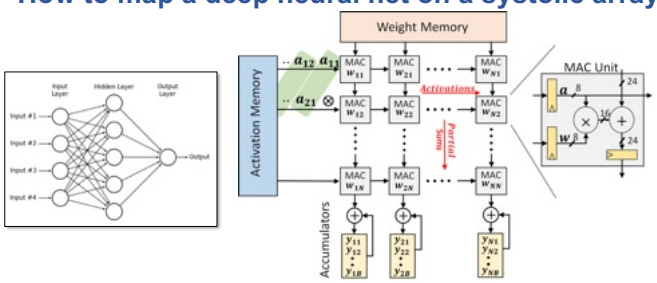
Mishra et al., 2023

Systolic array (2)

- Systolic arrays, are characterized by:
  - Pre-defined computational flow graph that connects their nodes.
  - Regular arrangement of (simple) processing elements (PEs) in a grid pattern.
  - Data flows rhythmically through the array of processors.
  - Each PE performs the same operation (typically multiply-accumulate).
  - Processing elements operate in lockstep, controlled by a global clock signal. [Wavefront computers, on the other hand, are asynchronous and data-driven.]
- In a systolic array, all processing elements execute the same instruction (or fixed function) at any given time, just on different data values.
- MISD or SIMD or ? Controversial…

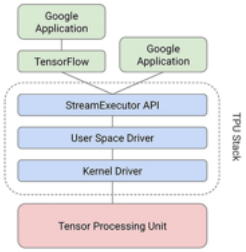


How to map a deep neural net on a systolic array



https://doi.org/10.1145/3195970.3196129

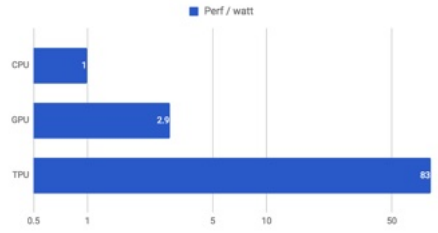
Google's TPU



	Operations per cycle
CPU	a few
CPU (vector extension)	tens
GPU	tens of thousands
TPU	hundreds of thousands, up to 128K

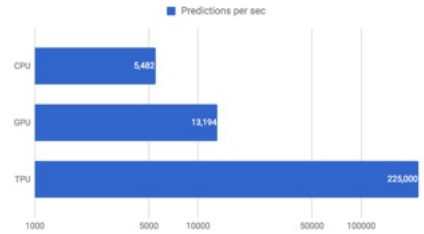
https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

Google's TPU



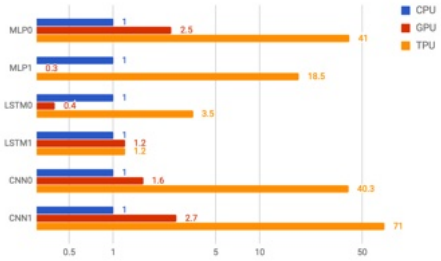
https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

Google's TPU



https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

Google's TPU



https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

Transformers

2017 landmark paper

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

https://arxiv.org/pdf/1706.03762

2017 landmark paper

Attention is all you need
A Vaswani, N Shazeer, N Parmar ... - Advances in neural ..., 2017 - proceedings.neurips.cc
... to attend to **all** positions in the decoder up to and including that position. **We need** to prevent ... **We** implement this inside of scaled dot-product **attention** by masking out (setting to $-\infty$) ...
☆ Save 99 Cite Cited by 177664 Related articles All 73 versions

A mathematical theory of communication
CE Shannon - The Bell system technical journal, 1948 - ieeexplore.ieee.org
... By a **communication** system we will mean a system of the type ... as **mathematical** entities, suitably idealized from their physical counterparts. We may roughly classify **communication** ...
☆ Save 99 Cite Cited by 109860 Related articles All 85 versions Web of Science: 19192

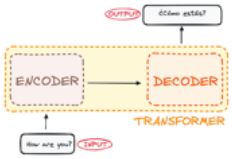
https://arxiv.org/pdf/1706.03762

What is a transformer?

A transformer is designed to comprehend context and meaning by analyzing the relationship between different elements.

The model is primarily composed of two blocks:

- **Encoder** (left): The encoder receives an input and builds a representation of it (its features). This means that the model is optimized to **acquire understanding** from the input.
- **Decoder** (right): The decoder uses the encoder's representation (features) along with other inputs to generate a target sequence. This means that the model is optimized for **generating outputs**.

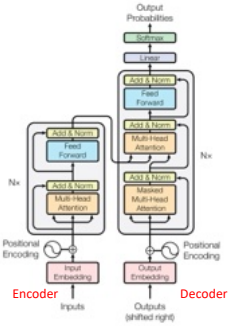


https://huggingface.co/learn/llm-course/en/chapter1/4
https://www.datacamp.com/tutorial/how-transformers-work

What is a transformer?

"A transformer model is a neural network that learns context and thus meaning by **tracking relationships in sequential data** like the words in this sentence." **Transformers made self-supervised learning possible.**

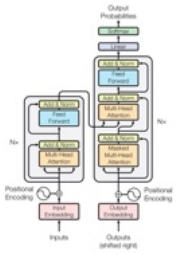
https://arxiv.org/pdf/1706.03762



"Transformer models apply an evolving set of mathematical techniques, called **attention** or **self-attention**, to detect subtle ways even distant data elements in a series influence and depend on each other."

What is a transformer?

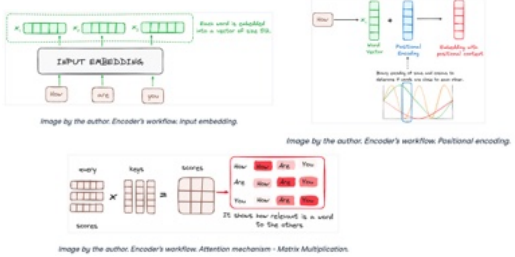
- Addresses all drawbacks of recurrent neural nets (RNN)
- No labels (costly and time-consuming)
- Very parallelizable
- Higher performance
- Like most neural networks, transformer models are basically large **encoder/decoder (compression/decompression)** blocks that process data.
- Transformers use **positional encoders** to tag data elements coming in and out of the network.
- **Attention units** follow these tags, calculating a kind of algebraic map of how each element relates to the others.
- **Attention queries** are typically executed in parallel by calculating a matrix of equations in what's called **multi-headed attention**.



https://arxiv.org/pdf/1706.03762
https://blogs.nvidia.com/blog/what-is-a-transformer-model

What is a transformer?

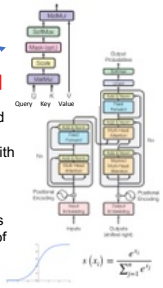
A transformer is designed to comprehend context and meaning by analyzing the relationship between different elements.



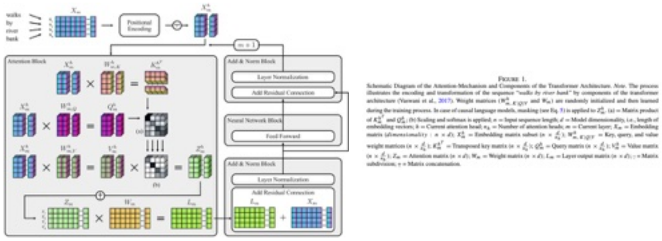
https://www.datacamp.com/tutorial/how-transformers-work

Main mathematical operations of a transformer

- **Matrix Multiplication** - Used throughout the architecture, particularly in the attention mechanism and feed-forward networks.
- **Attention Mechanism** - The core operation involving:
  - Query, Key, Value transformations (matrix multiplications)
  - Scaled dot-product attention: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ [GPU SFU]
  - Multi-head attention which runs multiple attention operations in parallel
- **Layer Normalization** - Normalizes the outputs of sub-layers using mean and variance statistics
- **Residual Connections** (Skip Connections) - Addition operations that help with gradient flow
- **Position-wise Feed-Forward Networks** - Two linear transformations with a ReLU activation in between
- **Softmax** - Used in the attention mechanism to convert scores to probabilities
- **Positional Encoding** - Often implemented using sine and cosine functions of different frequencies [GPU SFU]
- **Embedding** - Converting tokens to continuous vector representations



How to map transformers on a GPU



Hommel, B.E., Wollang, FJ.M., Kotova, V. et al. **Transformer-Based Deep Neural Language Modeling for Construct-Specific Automatic Item Generation**. Psychometrika 87, 749–772 (2022). https://doi.org/10.1007/s11336-021-09823-9

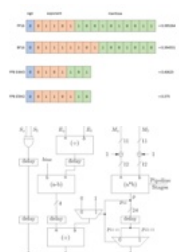
Reducing precision

- FP8 is an 8-bit floating point standard which was jointly developed by NVIDIA, ARM, and Intel to speed up AI development by improving memory efficiency during AI training and inference processes.
- **The ongoing industry evolution is evident in the shift from 32-bit to 16-bit, and currently, to 8-bit precision formats**. This change is especially advantageous for transformer networks, crucial AI breakthroughs, as they perform better with the detailed precision provided by 8-bit floating point formats.
- The **E5M2** format adapts the IEEE FP16 format, allocating five bits to the exponent and two bits to the mantissa. The **E4M3** format assigns four bits for the exponent and three bits for the mantissa with slight adjustments. These two eight-bit configurations are designed to optimize both training and inference phases, promising to cut down computational loads in comparison to their more precise counterparts.
- **The FP8 standard preserves accuracy comparable to 16-bit formats across a wide range of applications, architectures, and networks.**

https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html
https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html
https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture

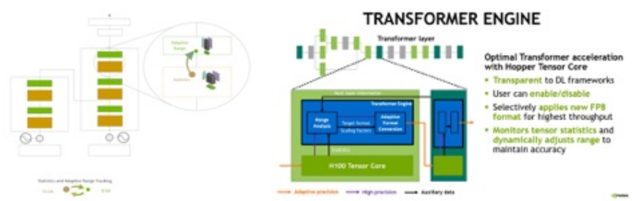
Why is FP8 faster than FP16?

- **Data Transfer Efficiency**: Since FP8 values are half the size of FP16 values, one can move twice as much data in the same memory bandwidth. Memory bandwidth is often a bottleneck in computations, so this alone can provide significant speedups.
- **Cache Efficiency**: Smaller data types mean more values can fit in cache, reducing expensive off-chip memory accesses.
- **Compute Density**: One can fit more FP8 arithmetic units in the same silicon area compared to FP16 units. This means more parallel operations can be performed simultaneously.
- **Power Efficiency**: Lower precision operations consume less power, allowing for higher clock speeds or more operations within the same power envelope.



NVIDIA Transformer Engine (1st gen)

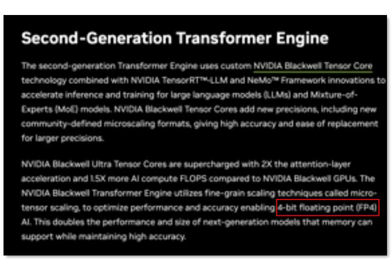
Essentially a library that uses the the GPU tensor cores.



Transformer Engine uses per-layer statistical analysis to **dynamically determine the optimal precision (FP16 or FP8) for each layer of a model**, achieving the best performance while preserving model accuracy.

https://www.hardwarezone.com.sg/tech-news-nvidia-h100-gpu-hopper-architecture-building-block-ai-infrastructure-dgx-h100-supercomputer
https://blogs.nvidia.com/blog/h100-transformer-engine
https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html
https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture

NVIDIA Blackwell Transformer Engine



Second-Generation Transformer Engine

The second-generation Transformer Engine uses custom NVIDIA Blackwell Tensor Core technology combined with NVIDIA TensorRT™-LLM and NeMo™ Framework innovations to accelerate inference and training for large language models (LLMs) and Mixture-of-Experts (MoE) models. NVIDIA Blackwell Tensor Cores add new precisions, including new community-defined microscaling formats, giving high accuracy and ease of replacement for larger precisions.

NVIDIA Blackwell Ultra Tensor Cores are supercharged with 2X the attention-layer acceleration and 1.5X more AI compute FLOPS compared to NVIDIA Blackwell GPUs. The NVIDIA Blackwell Transformer Engine utilizes fine-grain scaling techniques called micro-tensor scaling, to optimize performance and accuracy enabling 4-bit floating point (FP4) AI. This doubles the performance and size of next-generation models that memory can support while maintaining high accuracy.

https://blogs.nvidia.com/blog/h100-transformer-engine
https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html
https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture

Journals & Magazines > IEEE Transactions on Circuits... > Early Access

A Neuromorphic Transformer Architecture Enabling Hardware-Friendly Edge Computing

Publisher: IEEE Cite This PDF

P. J. Zhou ; R. C. Ma ; Y. C. Chen ; Z. T. Liu ; C. Y. Liu ; L. W. Meng All Authors

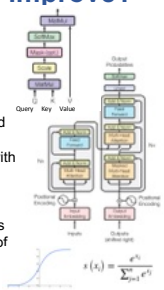
Abstract
Authors
Keywords

Abstract:
The transformer model has demonstrated significant capabilities in various intelligent tasks, attracting widespread attention in recent years. However, it involves numerous complex operations, including large-bit-width multiplication, division, matrix transposition, and exponentiation. These require substantial storage and computational resources, making it challenging to deploy on edge devices.

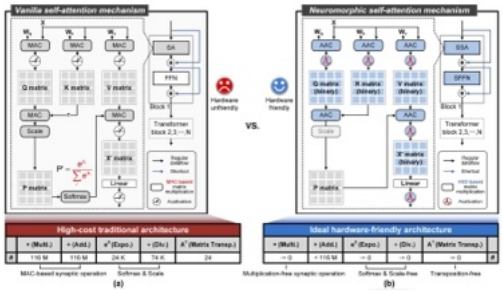
https://ieeexplore.ieee.org/abstract/document/10962199

What transformer operations could we improve?

- **Matrix Multiplication** - Used throughout the architecture, particularly in the attention mechanism and feed-forward networks.
- **Attention Mechanism** - The core operation involving:
  - Query, Key, Value transformations (matrix multiplications)
  - Scaled dot-product attention: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
  - Multi-head attention which runs multiple attention operations in parallel
- **Layer Normalization** - Normalizes the outputs of sub-layers using mean and variance statistics
- **Residual Connections** (Skip Connections) - Addition operations that help with gradient flow
- **Position-wise Feed-Forward Networks** - Two linear transformations with a ReLU activation in between
- **Softmax** - Used in the attention mechanism to convert scores to probabilities
- **Positional Encoding** - Often implemented using sine and cosine functions of different frequencies
- **Embedding** - Converting tokens to continuous vector representations



Neuromorphic self-attention mechanism



https://ieeexplore.ieee.org/abstract/document/10962199

To spike or not to spike

Spiking neural networks are more energy efficient than traditional artificial neural networks primarily because:

- **Event-driven processing**: SNNs only process information when neurons fire, unlike traditional ANNs which compute at every time step regardless of input changes. This significantly reduces power consumption.
- **Binary communication**: SNNs communicate using discrete spikes (binary events) rather than continuous values, requiring less data transfer and simpler hardware.
- **Temporal encoding**: Information in SNNs can be encoded in the timing of spikes, allowing for more efficient data representation compared to the amplitude-based encoding in ANNs.
- **Reduced precision requirements**: SNNs can work with lower precision hardware since they use binary spikes, while ANNs typically need higher precision for gradient calculations.
- **Closer to biological efficiency**: The human brain, which uses spiking neurons, consumes only about 20 watts while performing complex tasks that would require orders of magnitude more power on traditional computing hardware.