

---

# Workshop 1:

# GPU computing background

Jeremy Jacobson  
Lecturer



EMORY  
COLLEGE  
OF ARTS AND  
SCIENCES

Department of Quantitative  
Theory and Methods

---

# Overview

## Main points

- (5 minutes) QTM Honors student John Cox: GPU computing lightning talk from Honors thesis
- (20 minutes) **Computing trends**
  - Using a GPU is a big investment in time. We will explain why it is a wise long term investment even if you don't need one now.
- (20 minutes) **GPUs and matrices**
  - GPUs are fast for data science and ML. Why?

# Computing Trends

- What is a computer?
- 50 years of architecture with David Patterson

---

**What is a  
computer?**

---

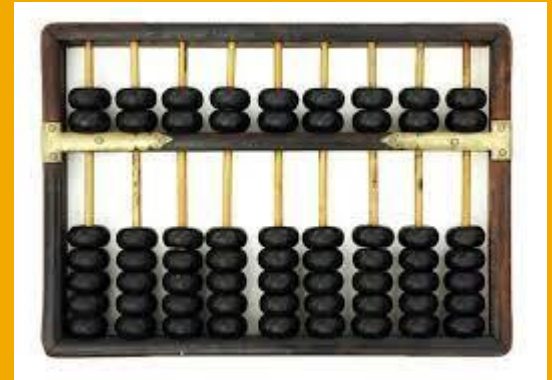
# What is a computer?

- For today's workshop it is:
  - Physical memory and a means of manipulating it

# What is a computer?

- For example

- 



# David Patterson

- Won ACM Turing Award in 2017 for innovative Reduced Instruction Set Computers (RISC)

# David Patterson

- Won ACM Turing Award in 2017 for innovative Reduced Instruction Set Computers (RISC)
  - 99 percent of the more than 16 billion microprocessors produced annually are RISC processors, and they are found in nearly all smartphones
-



# David Patterson

- Won ACM Turing Award in 2017 for innovative Reduced Instruction Set Computers (RISC)
  - 99 percent of the more than 16 billion microprocessors produced annually are RISC processors, and they are found in nearly all smartphones
  - Following slides are from his talk at the Google Faculty Institute 2018
-



# Outline

## Part I

History of Architecture

Mainframes, Minicomputers,  
Microprocessors, RISC vs  
CISC, VLIW

## Part II

Current Architecture Challenges

Ending of Dennard Scaling and  
Moore's Law, Security

## Part III

Future Architecture Opportu

Domain Specific Languages

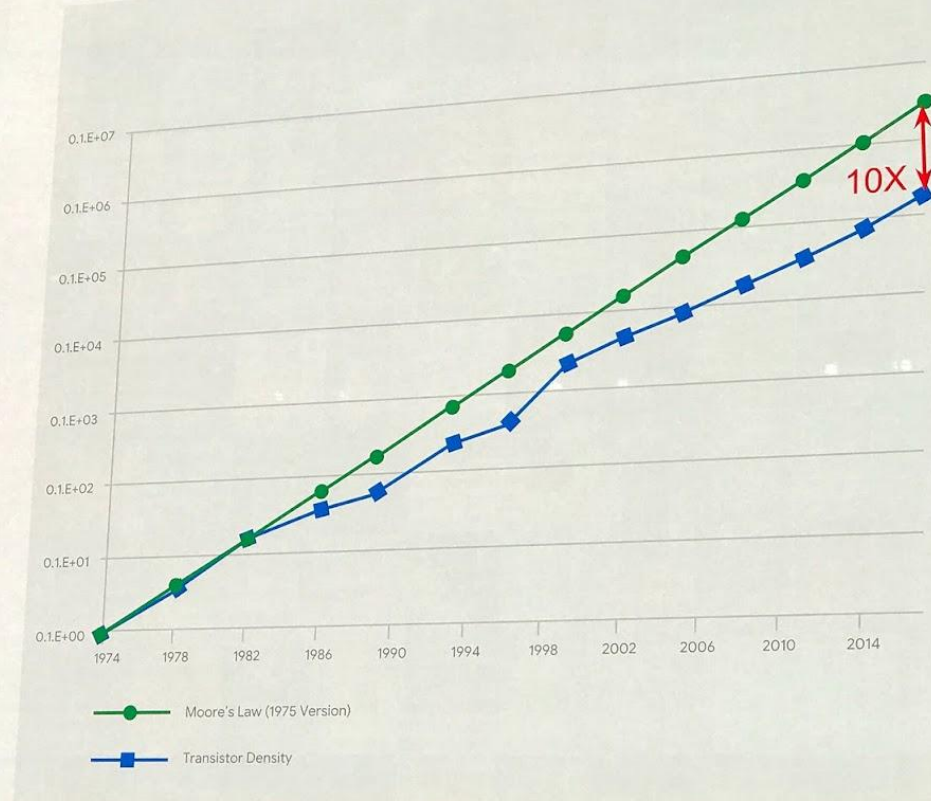
Architecture, Open Architec

Agile Hardware Developmen

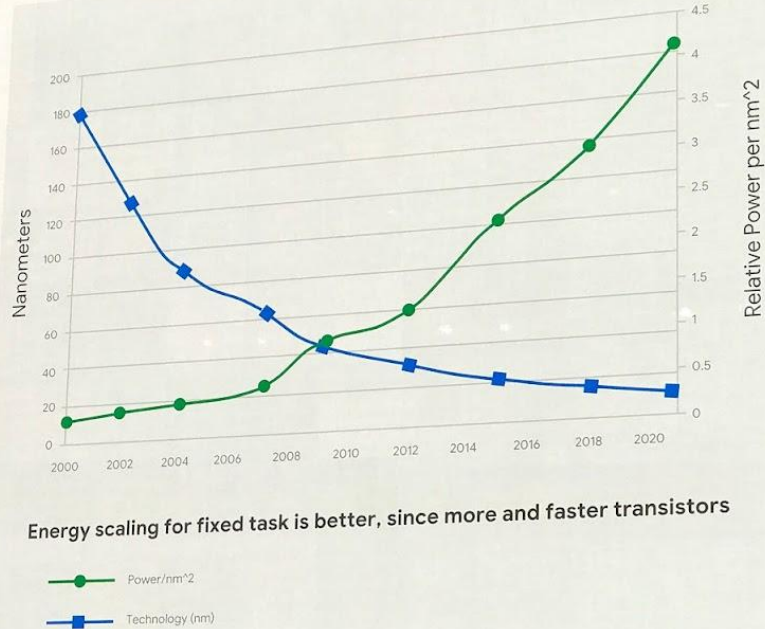


# Moore's Law slowdown in Intel Processors

transistor slowing down faster,  
fab costs.



## Technology & Power: Dennard Scaling



Power consumption based on models in "Dark Silicon and the End of Multicore Scaling," Hadi Esmaeilzadeh, ISCA, 2011



## What Opportunities Left?

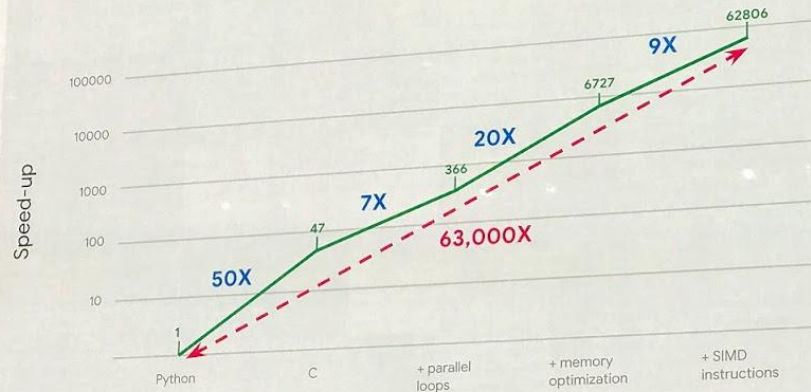
- SW-centric
  - Modern scripting languages are interpreted, dynamically-typed and encourage reuse
  - Efficient for programmers but not for execution
- HW-centric
  - Only path left is Domain Specific Architectures
  - Just do a few tasks, but extremely well
- Combination
  - Domain Specific Languages & Architectures




# What's the Opportunity?

Matrix Multiply: relative speedup  
to a Python version (18 core Intel)

Matrix Multiply Speedup Over Native Python



from: "There's Plenty of Room at the Top," Leiserson, et. al., to appear.



## Domain Specific Languages

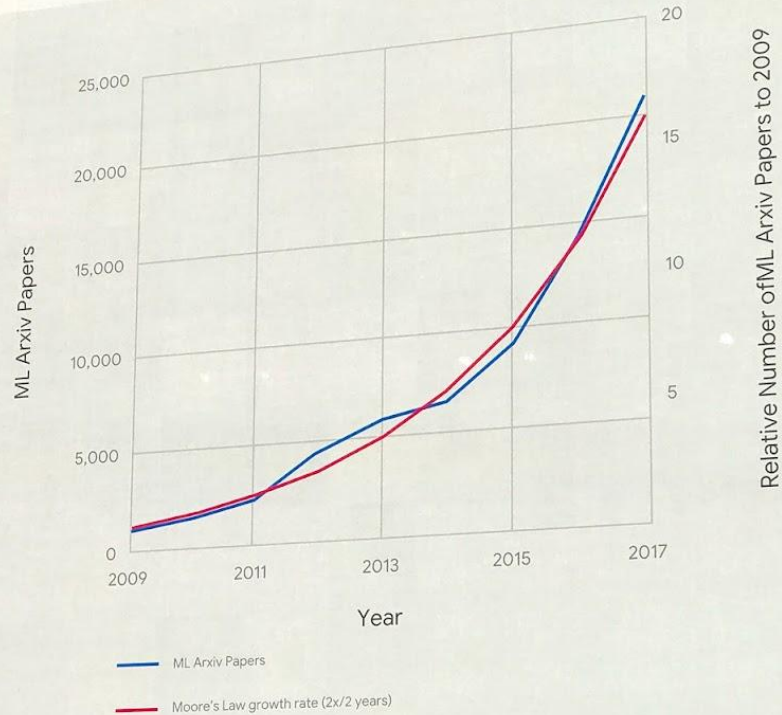
DSAs require targeting of high level operations to the architecture

- Hard to start with C or Python-like language and recover structure
- Need matrix, vector, or sparse matrix operations
- Domain Specific Languages specify these operations:
  - OpenGL, TensorFlow, P4
- If DSL programs retain architecture-independence, interesting compiler challenges will exist





Deep learning is  
causing a machine  
learning revolution



From "A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution."  
Dean, J., Patterson, D., & Young, C. (2018). IEEE Micro, 38(2), 21-29.

# GPUs and matrices

- A look at the QTM GPU
- Matrix operations and machine learning
- Matrix multiply on a GPU

# QTM GPU

- Exterior
- Interior
- Silicon die
- Cartoon illustration of die layout
- Table of specs

---

A6000 (our GPU) exterior



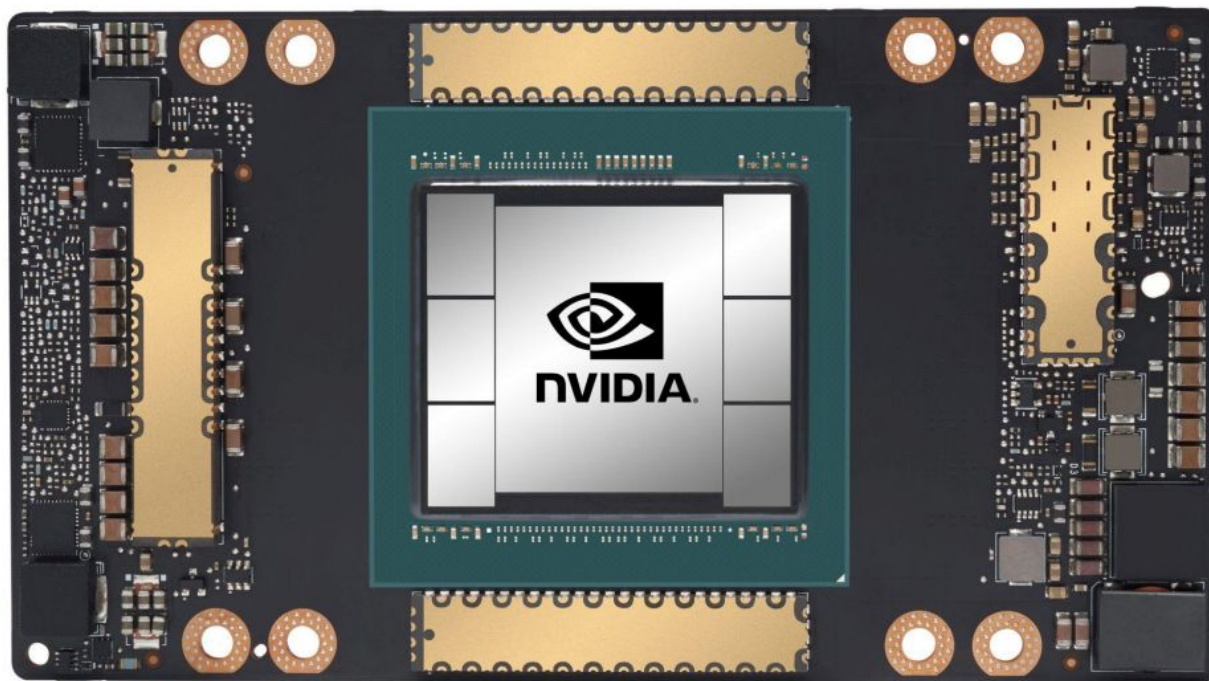
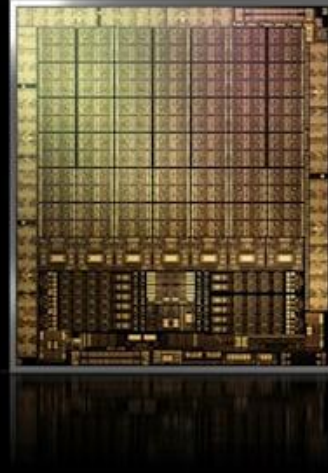
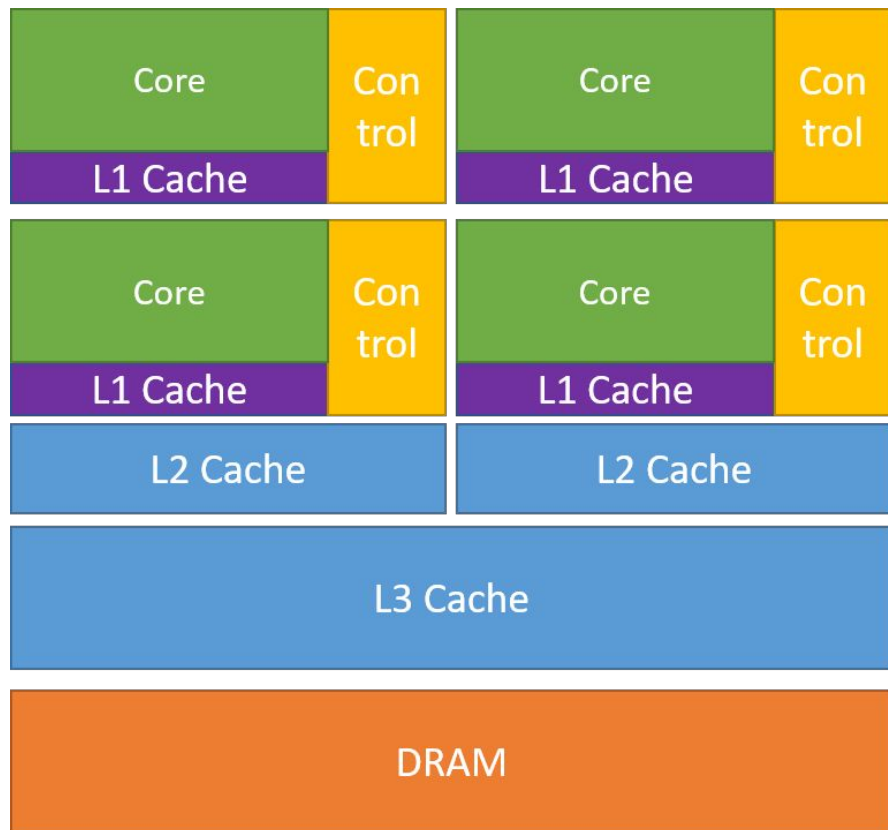


Figure 3. NVIDIA A100 GPU on new SXM4 Module

## A6000 (QTM GPU) Die shot

# NVIDIA AMPERE ARCHITECTURE





CPU



GPU

<b>L2 Cache Size</b>		6144 KB
<b>Register File Size</b>		21504 KB
<b>TGP (Total Graphics Power)</b>		300W
<b>Transistor Count</b>	28.3 Billion	28.3 Billion
<b>Die Size</b>	628.4 mm <sup>2</sup>	628.4 mm <sup>2</sup>
<b>Manufacturing Process</b>	Samsung 8 nm 8N NVIDIA Custom Process	Samsung 8 nm 8N NVIDIA Custom Process

Almost 10X  
a high end  
laptop chip

1. Peak rates are based on GPU Boost Clock.
2. Effective TOPS / TFLOPS using the new Sparsity Feature
3. TOPS = IMAD-based integer math

**NOTE: Refer to Appendix C for RTX A6000 performance data.**



# Matrix operations and machine learning

- Definition of neural network
- Chain rule and gradient descent
- Matrix multiply and gradient descent is data parallel

**Definition II.1.** Let  $L, N_0, N_1, \dots, N_L \in \mathbb{N}$ ,  $L \geq 2$ . A map  $\Phi : \mathbb{R}^{N_0} \rightarrow \mathbb{R}^{N_L}$  given by

$$\Phi(x) = \begin{cases} W_2(\rho(W_1(x))), & L = 2 \\ W_L(\rho(W_{L-1}(\rho(\dots\rho(W_1(x)))))), & L \geq 3 \end{cases}, \quad (1)$$

with affine linear maps  $W_\ell : \mathbb{R}^{N_{\ell-1}} \rightarrow \mathbb{R}^{N_\ell}$ ,  $\ell \in \{1, 2, \dots, L\}$ , and the ReLU activation function  $\rho(x) = \max(x, 0)$ ,  $x \in \mathbb{R}$ , acting component-wise (i.e.,  $\rho(x_1, \dots, x_N) := (\rho(x_1), \dots, \rho(x_N))$ ) is called a ReLU neural network. The map  $W_\ell$  corresponding to layer  $\ell$  is given by  $W_\ell(x) = A_\ell x + b_\ell$ , with  $A_\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$  and  $b_\ell \in \mathbb{R}^{N_\ell}$ .

trix/vector form. The basic building block of vectorized gradients is the *Jacobian Matrix*. Suppose we have a function  $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$  that maps a vector of length  $n$  to a vector of length  $m$ :  $\mathbf{f}(\mathbf{x}) = [f_1(x_1, \dots, x_n), f_2(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n)]$ . Then its Jacobian is the following  $m \times n$  matrix:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

That is,  $(\frac{\partial \mathbf{f}}{\partial \mathbf{x}})_{ij} = \frac{\partial f_i}{\partial x_j}$  (which is just a standard non-vector derivative). The Jacobian matrix will be useful for us because we can apply the chain rule to a vector-valued function just by multiplying Jacobians.

As a little illustration of this, suppose we have a function  $\mathbf{f}(x) = [f_1(x), f_2(x)]$  taking a scalar to a vector of size 2 and a function  $\mathbf{g}(\mathbf{y}) = [g_1(y_1, y_2), g_2(y_1, y_2)]$  taking a vector of size two to a vector of size two. Now let's compose them to get  $\mathbf{g}(x) = [g_1(f_1(x), f_2(x)), g_2(f_1(x), f_2(x))]$ . Using the regular chain rule, we can compute the derivative of  $\mathbf{g}$  as the Jacobian

$$\frac{\partial \mathbf{g}}{\partial x} = \begin{bmatrix} \frac{\partial}{\partial x} g_1(f_1(x), f_2(x)) \\ \frac{\partial}{\partial x} g_2(f_1(x), f_2(x)) \end{bmatrix} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial g_1}{\partial f_2} \frac{\partial f_2}{\partial x} \\ \frac{\partial g_2}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial g_2}{\partial f_2} \frac{\partial f_2}{\partial x} \end{bmatrix}$$

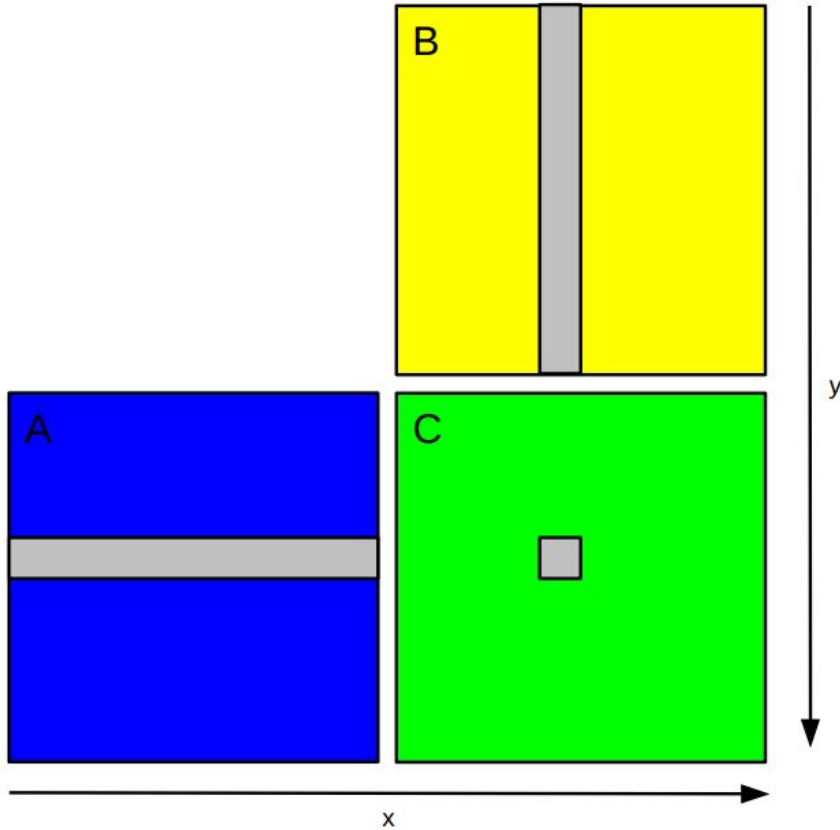
As a little illustration of this, suppose we have a function  $\mathbf{f}(x) = [f_1(x), f_2(x)]$  taking a scalar to a vector of size 2 and a function  $\mathbf{g}(\mathbf{y}) = [g_1(y_1, y_2), g_2(y_1, y_2)]$  taking a vector of size two to a vector of size two. Now let's compose them to get  $\mathbf{g}(x) = [g_1(f_1(x), f_2(x)), g_2(f_1(x), f_2(x))]$ . Using the regular chain rule, we can compute the derivative of  $\mathbf{g}$  as the Jacobian

$$\frac{\partial \mathbf{g}}{\partial x} = \begin{bmatrix} \frac{\partial}{\partial x} g_1(f_1(x), f_2(x)) \\ \frac{\partial}{\partial x} g_2(f_1(x), f_2(x)) \end{bmatrix} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial g_1}{\partial f_2} \frac{\partial f_2}{\partial x} \\ \frac{\partial g_2}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial g_2}{\partial f_2} \frac{\partial f_2}{\partial x} \end{bmatrix}$$

And we see this is the same as multiplying the two Jacobians:

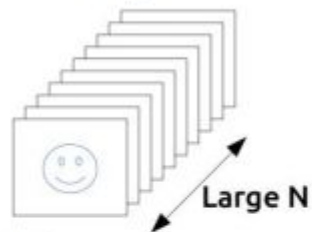
$$\frac{\partial \mathbf{g}}{\partial x} = \frac{\partial \mathbf{g}}{\partial \mathbf{f}} \frac{\partial \mathbf{f}}{\partial x} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1} & \frac{\partial g_1}{\partial f_2} \\ \frac{\partial g_2}{\partial f_1} & \frac{\partial g_2}{\partial f_2} \end{bmatrix} \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \end{bmatrix}$$

**Matrix multiply  $AB = C$**



important for inference as well.

## Training



## Inference

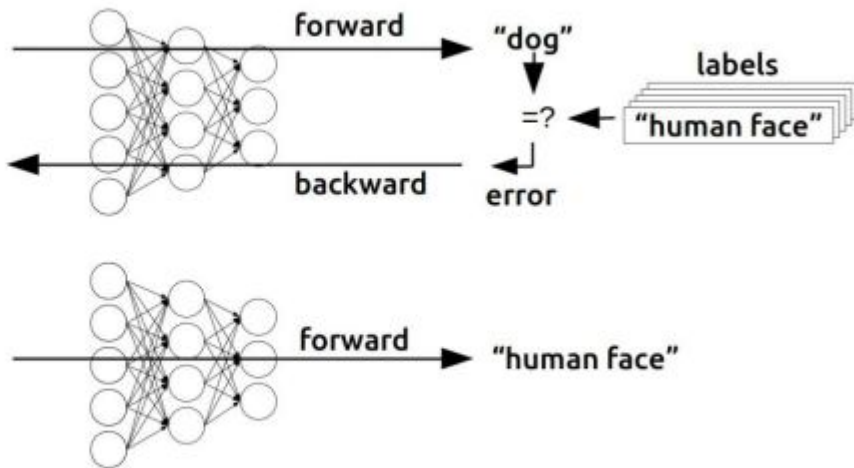
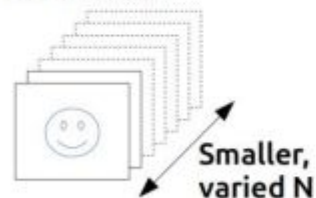


Figure 1: Deep learning training compared to inference. In training, many inputs, often in large batches, are used to train a deep neural network. In inference, the trained network is used to discover information within new inputs that are fed through the network in smaller batches.

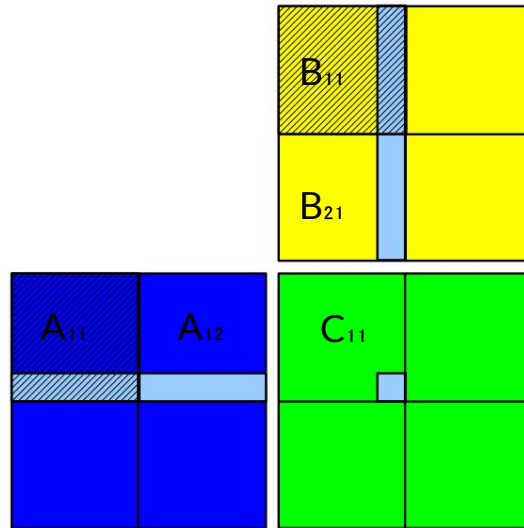
# Matrix multiply on a GPU

- Block matrices
- Different cores

---



# Block multiplication of matrices



$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}$$
$$\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}$$

$$D = AB + C$$

**D** =

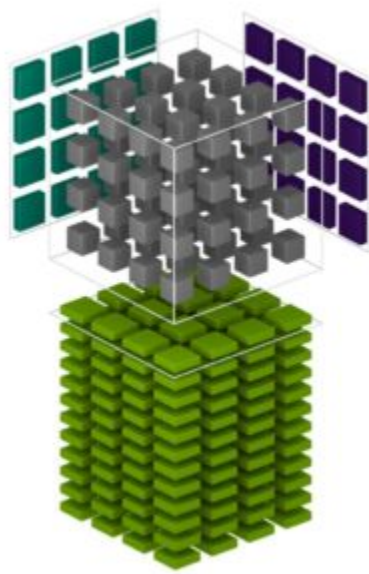
$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

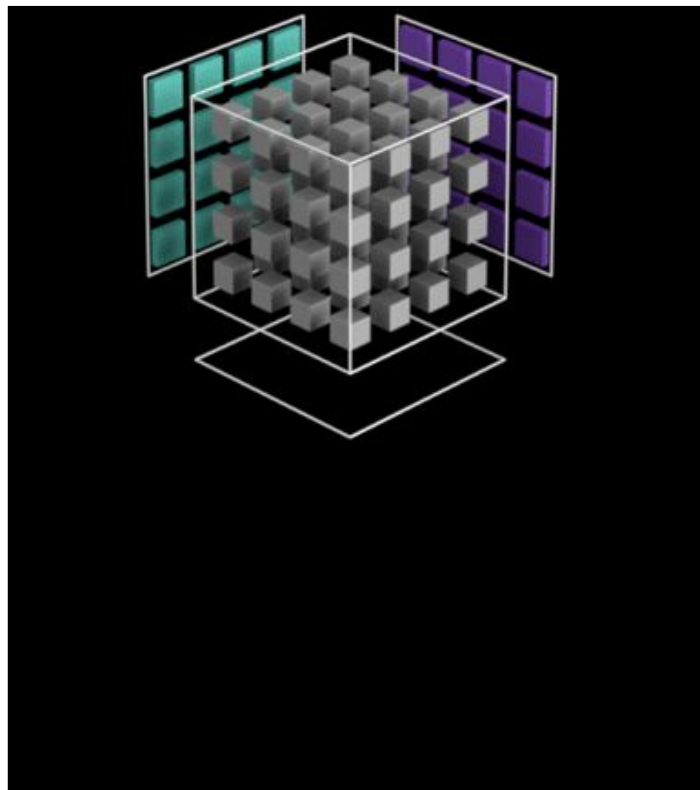
$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

+

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$C_{0,3}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$C_{1,3}$
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$
$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	$C_{3,3}$

Each Tensor Core provides a 4x4x4 matrix processing array which performs the operation  $\mathbf{D} = \mathbf{A} * \mathbf{B} + \mathbf{C}$





Graphics Card	NVIDIA RTX A6000	NVIDIA A40
GPU Codename	GA102	GA102
GPU Architecture	NVIDIA Ampere	NVIDIA Ampere
GPCs	7	7
TPCs	42	42
SMs	84	84
CUDA Cores / SM	128	128
CUDA Cores / GPU	10752	10752
Tensor Cores / SM	4 (3rd Gen)	4 (3rd Gen)
Tensor Cores / GPU	336 (3rd Gen)	336 (3rd Gen)
RT Cores	84 (2nd Gen)	84 (2nd Gen)
GPU Boost Clock (MHz)	1800	1740
Peak FP32 TFLOPS (non-Tensor) <sup>1</sup>	38.7	37.4
Peak FP16 TFLOPS (non-Tensor) <sup>1</sup>	38.7	37.4
Peak BF16 TFLOPS (non-Tensor) <sup>1</sup>	38.7	37.4
Peak INT32 TOPS (non-Tensor) <sup>1,3</sup>	19.4	18.7
Peak FP16 Tensor TFLOPS with FP16 Accumulate <sup>1</sup>	154.8/309.6 <sup>2</sup>	149.7/299.4 <sup>2</sup>
Peak FP16 Tensor TFLOPS with FP32 Accumulate <sup>1</sup>	154.8/309.6 <sup>2</sup>	149.7/299.4 <sup>2</sup>
Peak BF16 Tensor TFLOPS with FP32 Accumulate <sup>1</sup>	154.8/309.6 <sup>2</sup>	149.7/299.4 <sup>2</sup>
Peak TF32 Tensor TFLOPS <sup>1</sup>	77.4/154.8 <sup>2</sup>	74.8/149.6 <sup>2</sup>

# References

- [GPU programming with CUDA](#)
  - [Automatic differentiation](#)
  - [NVIDIA Ampere architecture whitepaper](#)
  - [CUDA programming guide](#)
  - [Chapter 1 of Programming massively parallel processors](#)
  - [Chapter 2 of Programming massively parallel processors](#)
  - [Benefits of GPU for data science](#)
-