EECS 570
Lecture 16
GPU Optimizations

Winter 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/co4urses/eecs570/

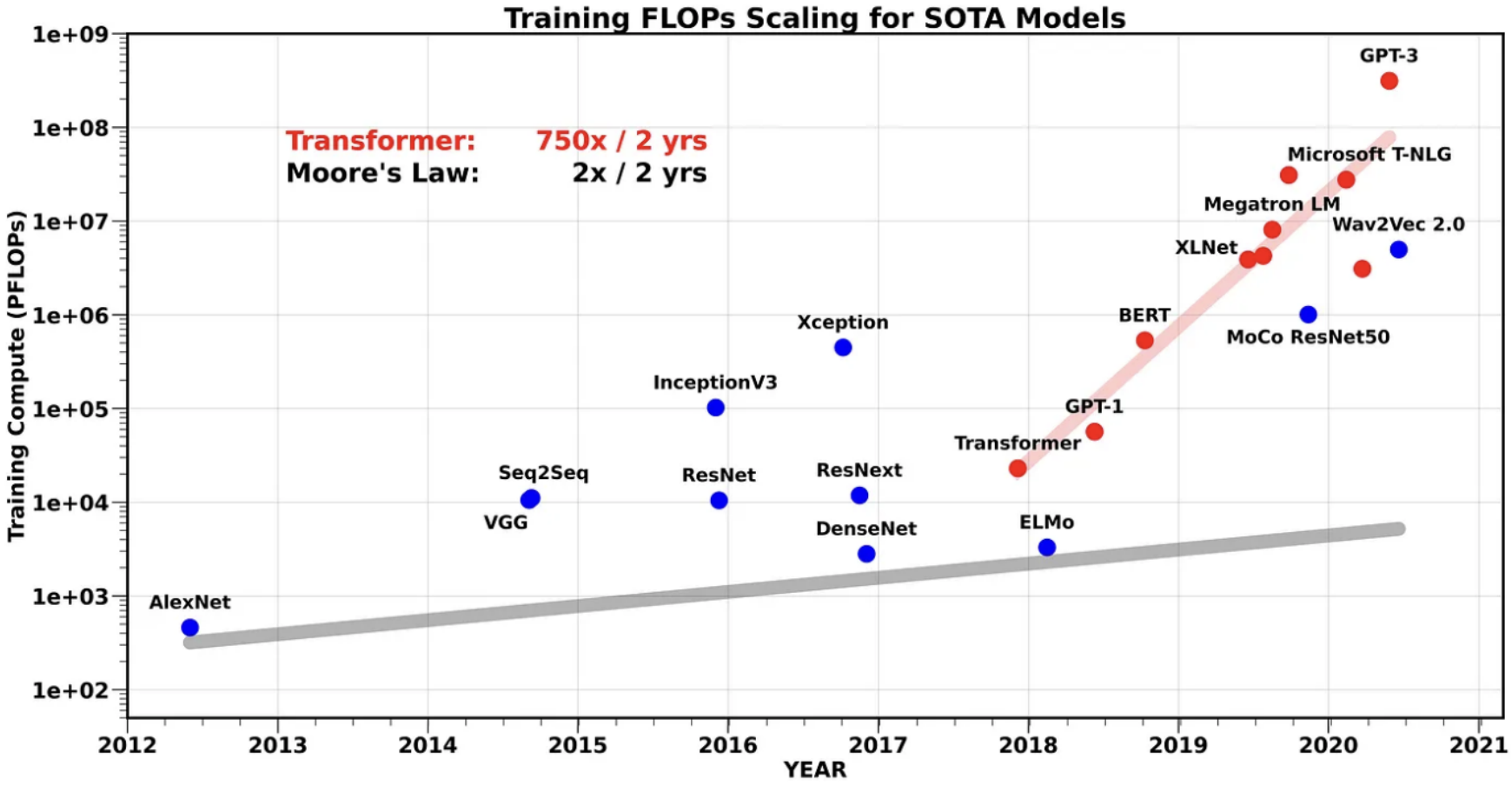
Slides adapted from instructional material from Simran Arora and Azalia Mirhoseini (Stanford)

Today’s Lecture

- Understanding performance -- Compute Vs Memory Bound
- A Few Principles For Improving Performance

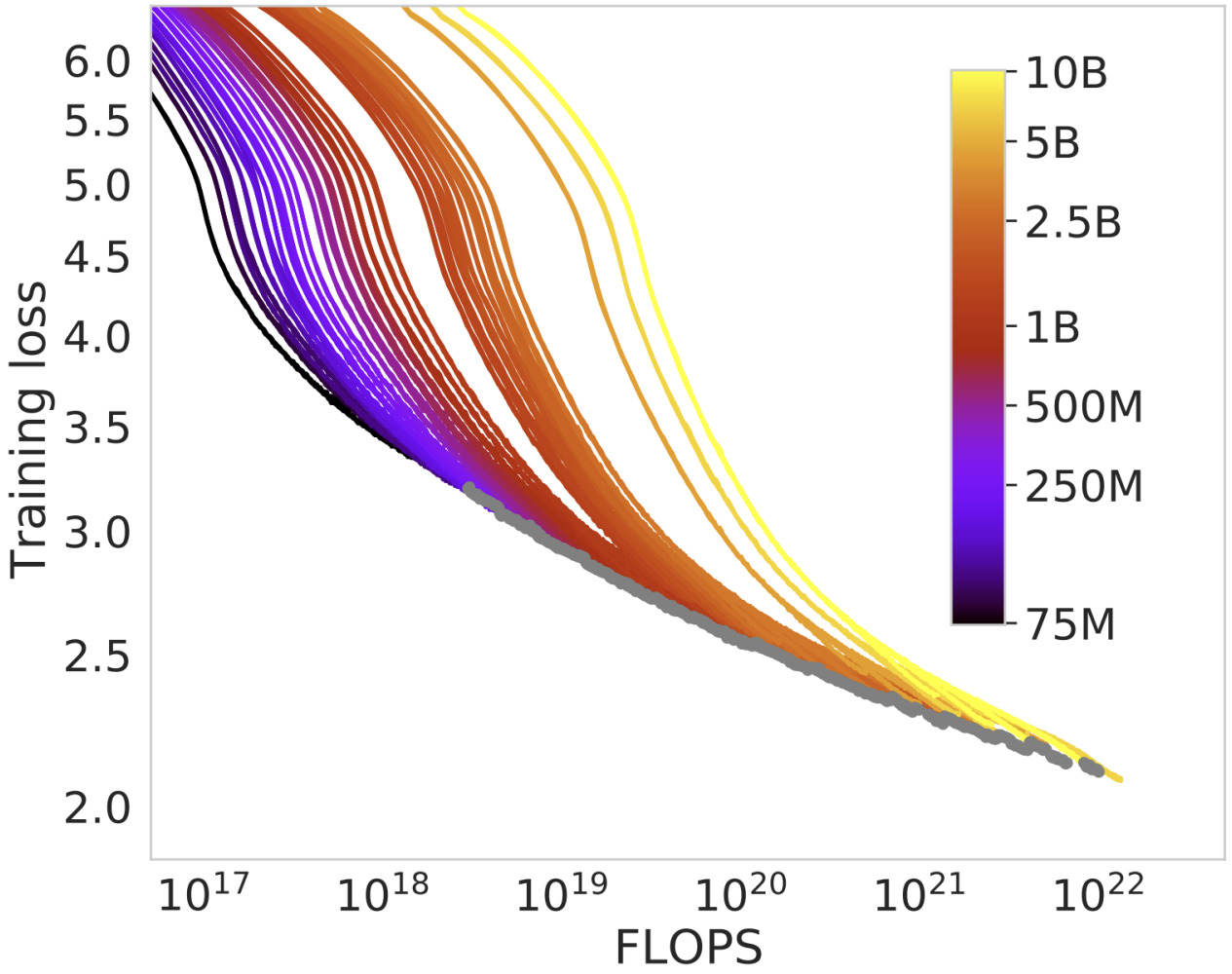
Hardware Overview

Rising Demand for Compute!



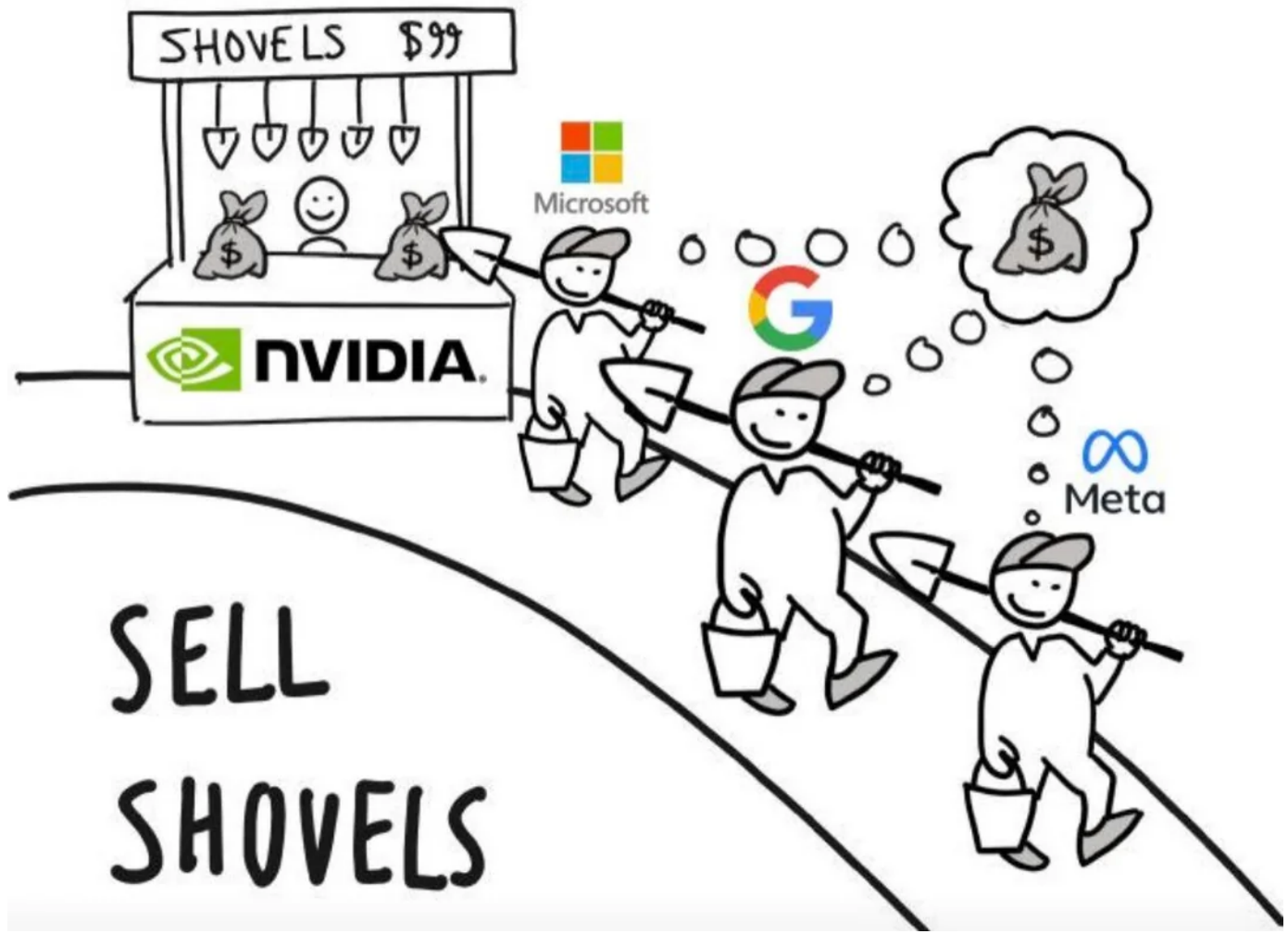
https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

The More FLOPS The Better!

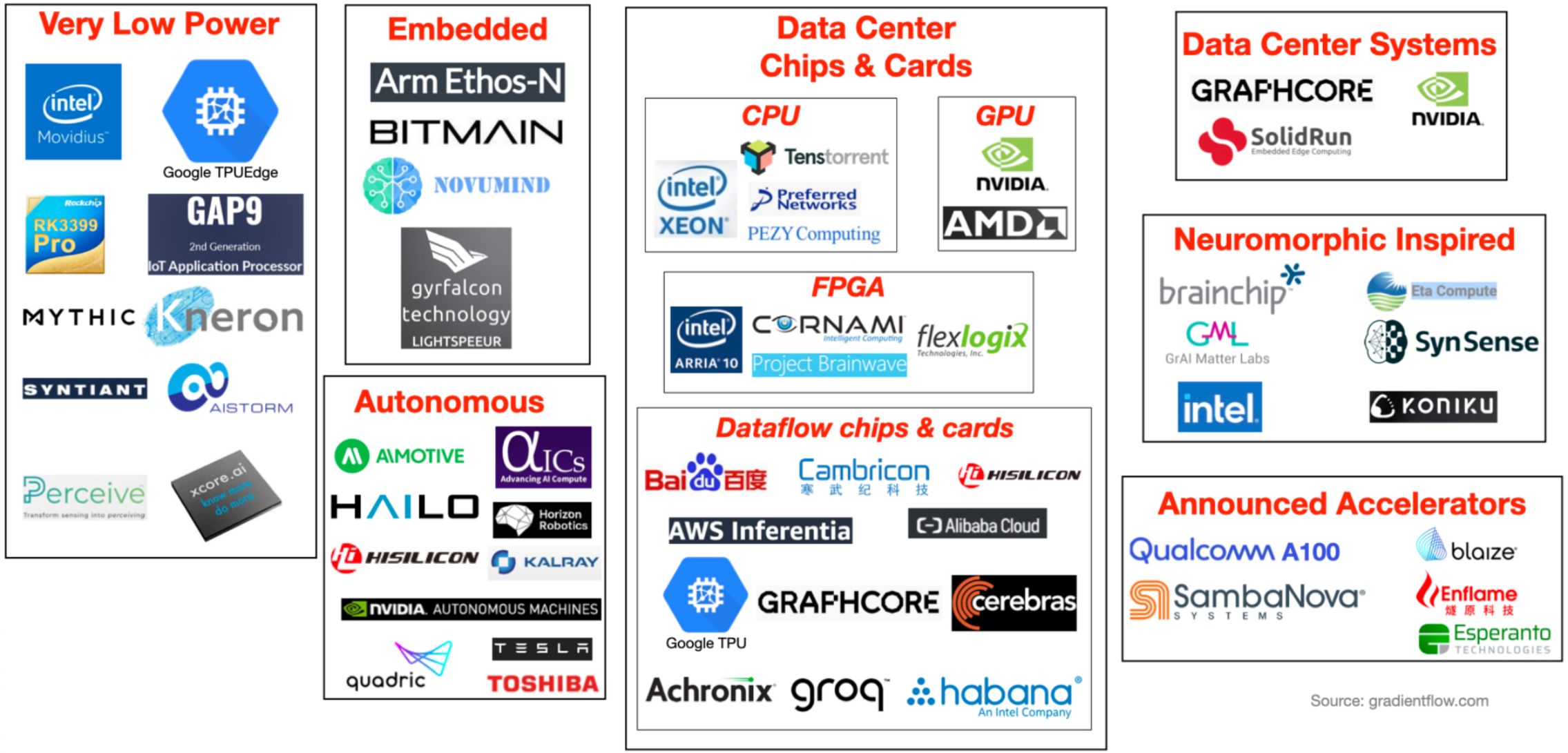


Chinchilla scaling laws - https://arxiv.org/pdf/2203.15556.pdf

WHEN EVERYONE DIGS FOR GOLD



More Companies Offering Deep Learning Accelerators



Source: gradientflow.com

https://gradientflow.com/tag/hardware/

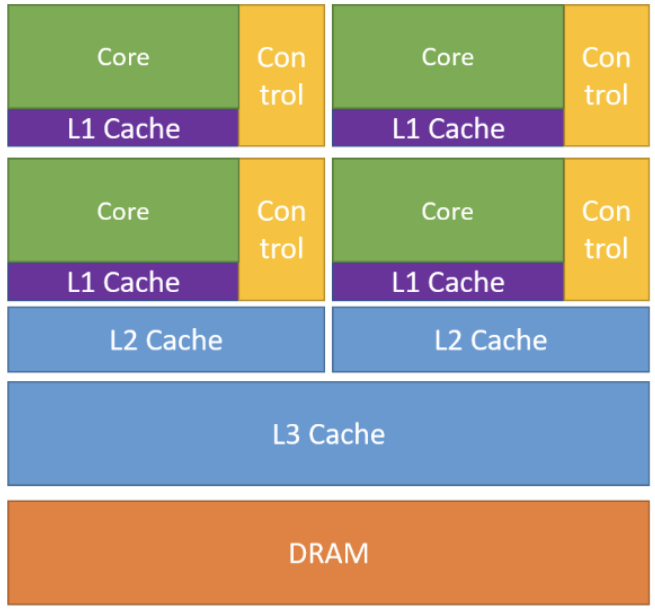
CPU vs. GPU

The core building block of deep learning models are vector and tensor multiplications and additions.

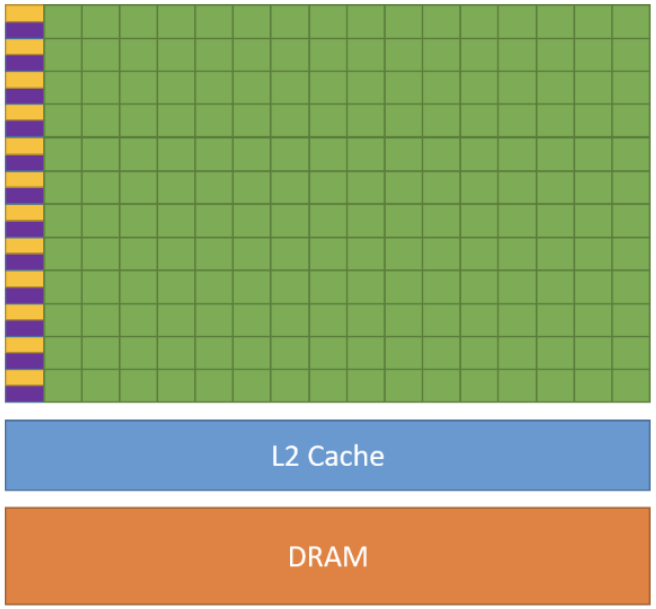
GPUs are optimized for parallel execution of these operations.

In CPUs, the majority of the transistors are used for data caching and flow control.

In GPUs the vast majority of transistors are used for data processing.



CPU



GPU

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#

CPU vs. GPU

CPUs are designed to execute a sequence of operations, called a thread, with minimum latency. They can execute a few tens of threads in parallel.

GPUs are designed to excel at executing thousands of threads in parallel. They have a slower single-thread performance, but achieve significantly higher throughput.

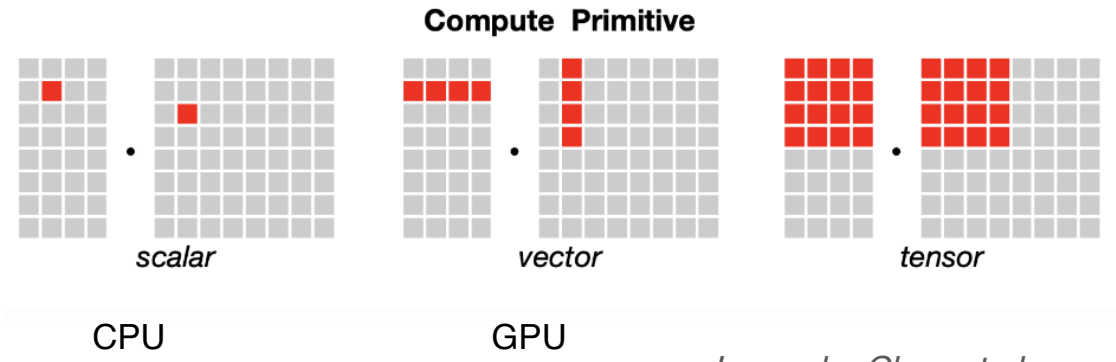
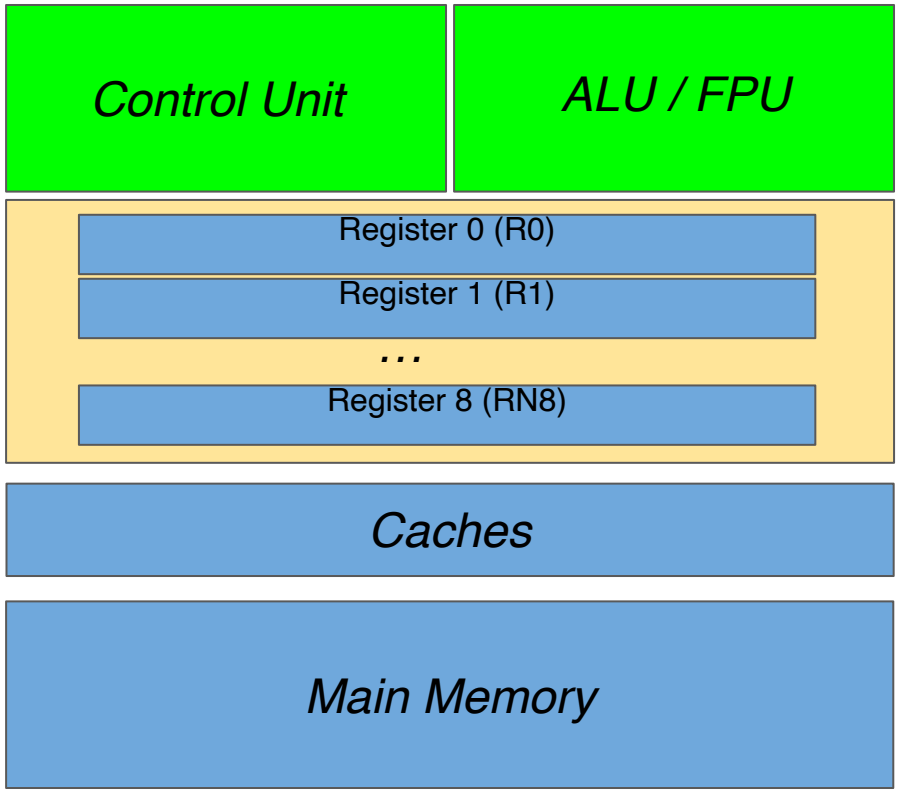


Image by Chen et al., 2018

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#

Central Processing Unit (CPU)
For general purpose computation



Control Unit: fetches the next instruction from memory and directs the arithmetic logic and floating point units (ALU and FPU)

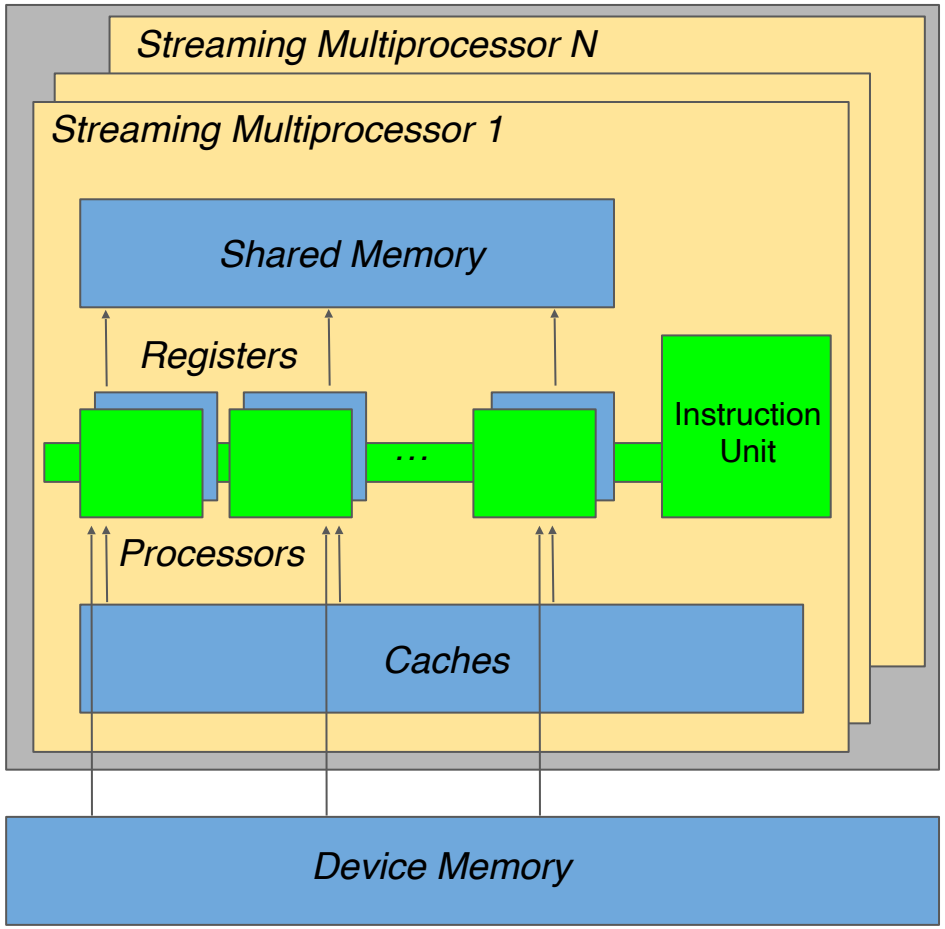
ALU & FPU: perform the bitwise instructions on integer and floating point numbers

Registers: for the next instruction, we can **read** our input values from the registers and **write** the intermediate values back out

Caches, main memory: much larger capacity than registers, used for storing instructions and data to run programs

Graphics Processing Unit (GPU)

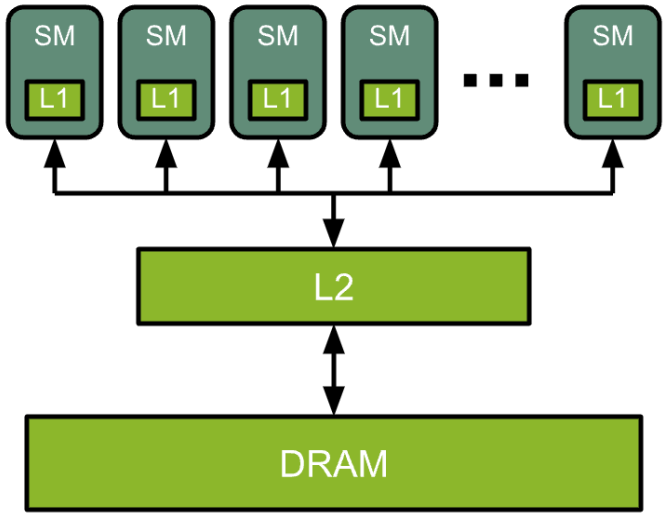
For parallelizable problems



Streaming Multiprocessors (SMs): each has one instruction unit that controls many processors that run the instruction in parallel

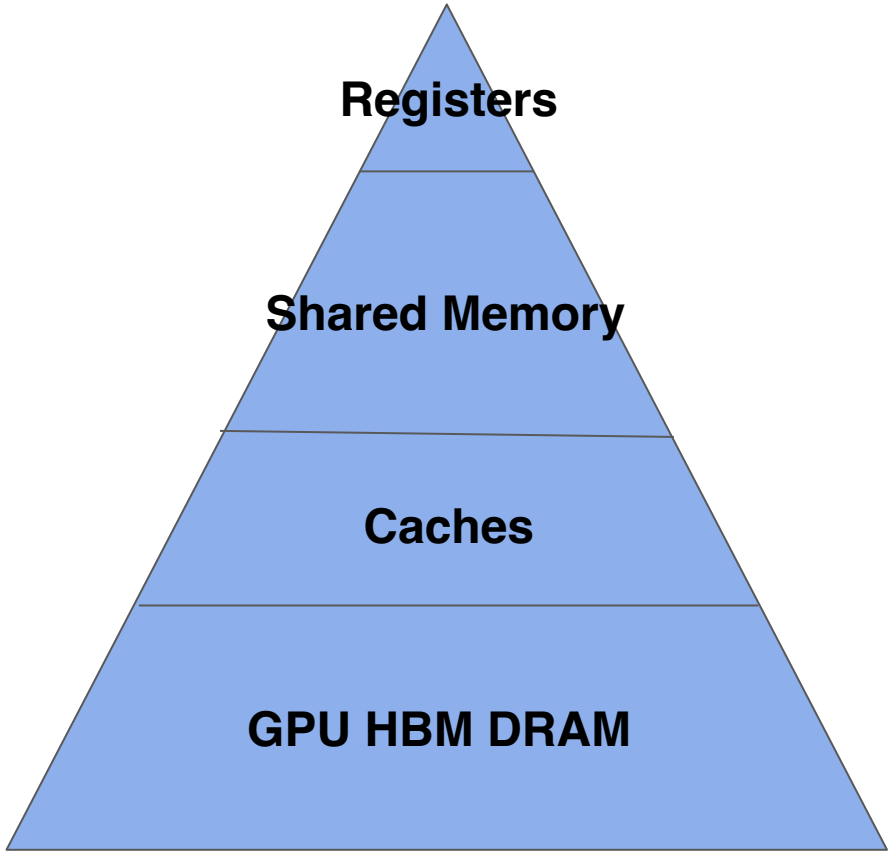
Registers, shared memory, caches, device memory: for reading and writing intermediate values

GPU Memory Hierarchy



https://docs.nvidia.com/deeplearning/

On a 40GB A100



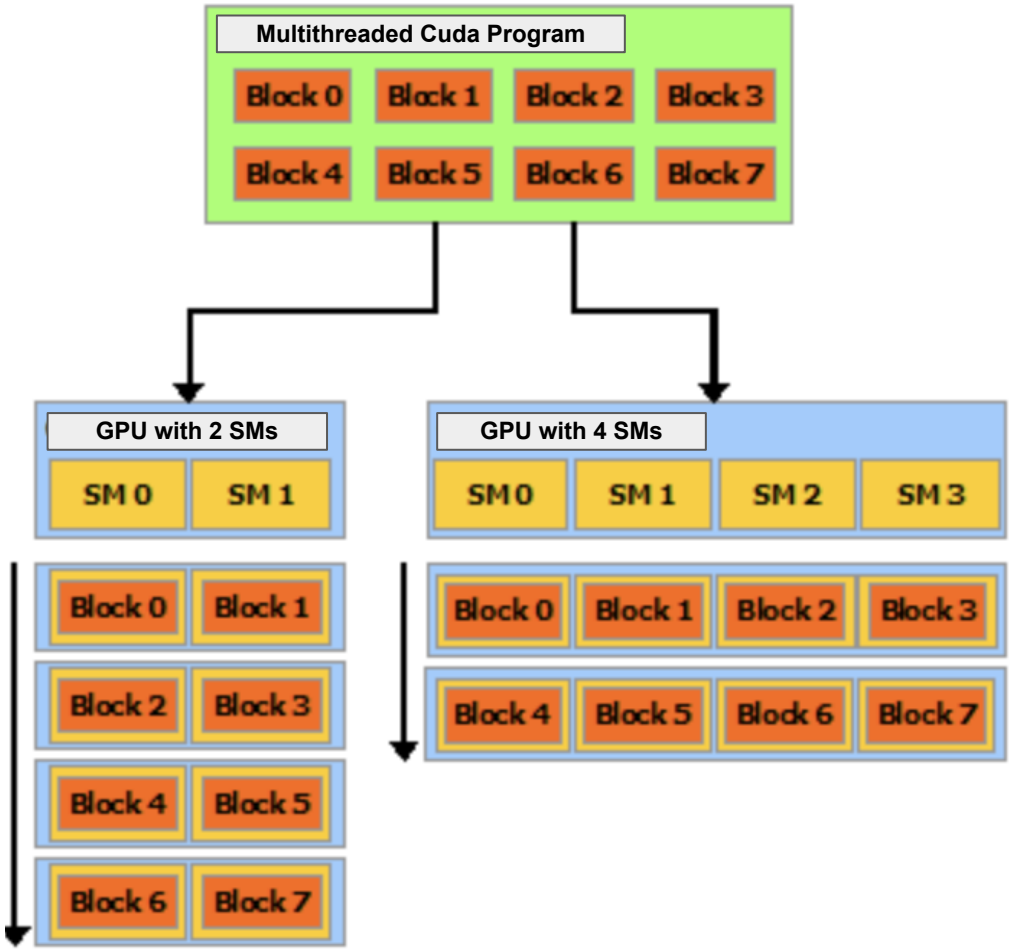
256 KB / SM

192 KB / SM

40 MB

40 GB

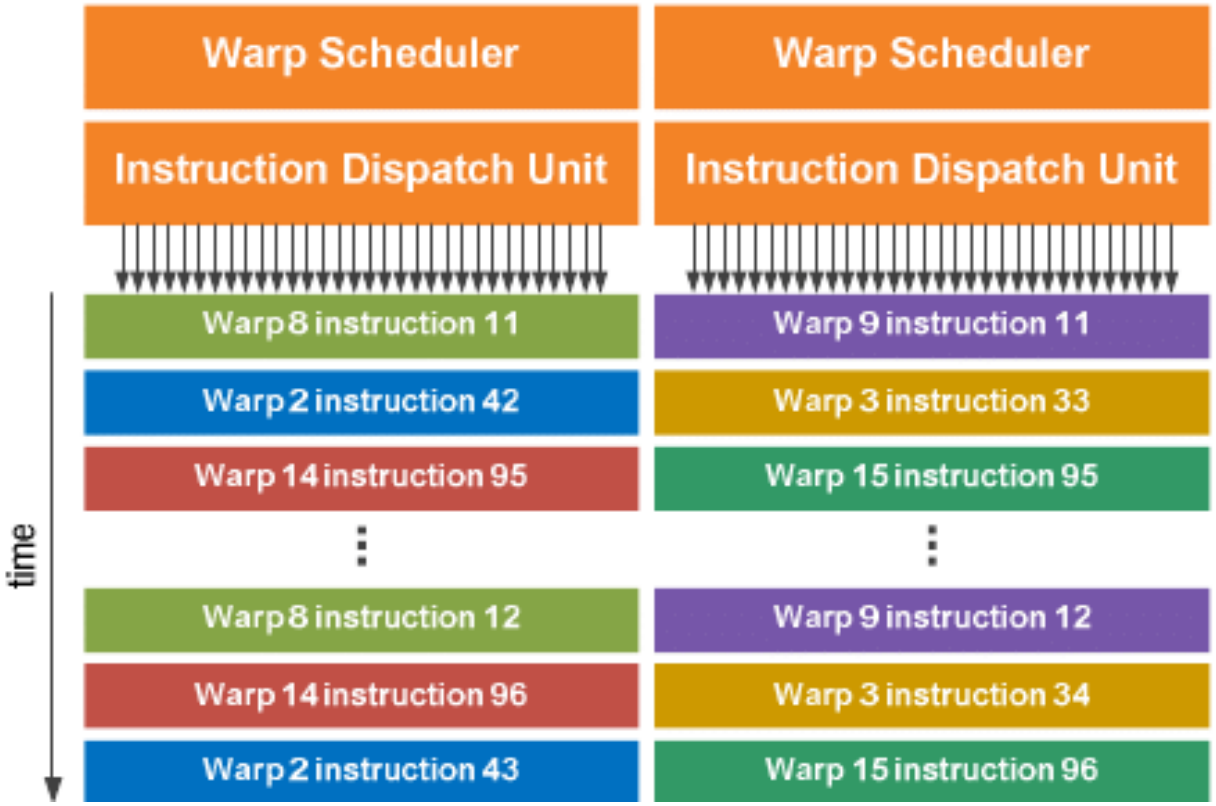
Programming Nvidia GPUs



- We make use of the Streaming Multiprocessors (SMs) on the GPU by organizing our computation into blocks.
- The blocks are organized into a grid.
- The blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity.
- Multiple thread blocks can execute concurrently on one multiprocessor.
- The threads of a block also execute concurrently on one multiprocessor.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#

SIMT (Single Instruction, Multiple Thread)



- A multiprocessor is designed to execute hundreds of threads concurrently.
- To manage such a large number of threads, it employs a unique architecture called *SIMT* (*Single Instruction, Multiple Thread*).
- The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#

Understanding Performance

How to think about performance

- We can think of algorithms in terms of asymptotics
  e.g., Strassen's Matrix Multiplication $O(N^{2.8074})$ vs. naive Matrix Multiplication $O(N^3)$, for NxN matrices

- However, the big O notation does not consider:
  - Constants which could make a large difference for different problem sizes
  - Implementation specific details (which matter a lot in real life)

- An algorithm may be asymptotically better but have worse performance if it is not well suited for specific hardware.

Implementation Matters: Algorithmic Scaling != Real-world Performance

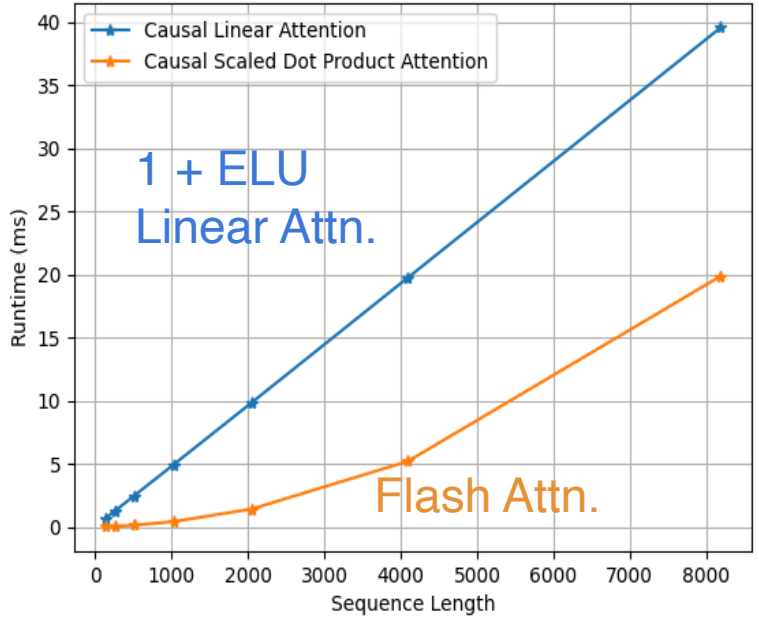
Consider 1 + ELU *: Fancy new linear attn algorithm.

$$V_i' = \frac{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j)} \qquad \phi(x) = \text{elu}(x) + 1$$

The complexity scales linearly O(N) w.r.t. context length N, vs. standard $O(N^2)$ attention.

Method	Complexity
Softmax	$\mathcal{O}(n^2 d)$
1 + ELU	$\boldsymbol{\mathcal{O}(nd^2)}$

Flash Attn, a *hardware aware* attention implementation, runs faster in wall-clock time**



*Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. Katharopoulos et al., ICML 2020. https://linear-transformers.com/
**Scaling without linear attention cuda kernel. Custom cuda kernel makes linear attention go faster.

Credit: Michael Zhang
https://michaelzhang.xyz/

How to think about performance

An algorithm may be asymptotically better but not well suited for specific hardware.

Hardware Utilization

- We know the theoretical peak computing performance and memory bandwidth of hardware.
- We can find the fraction of the peak theoretical performance attained by an algorithm and its implementation.
- We can use this information to decide how to improve the performance.

Theoretical Peak Performance

What happens inside your CPU/GPU is conceptually simple:

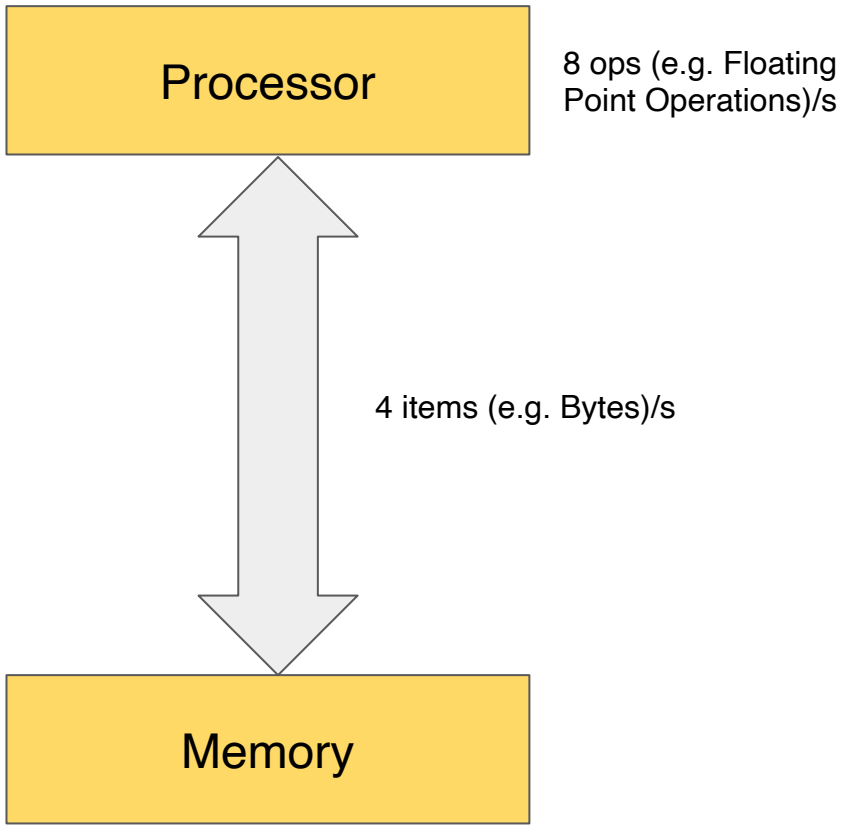
- Move items from memory to a processing unit
- Perform some computation
- Move items back to memory

Two key specifications:

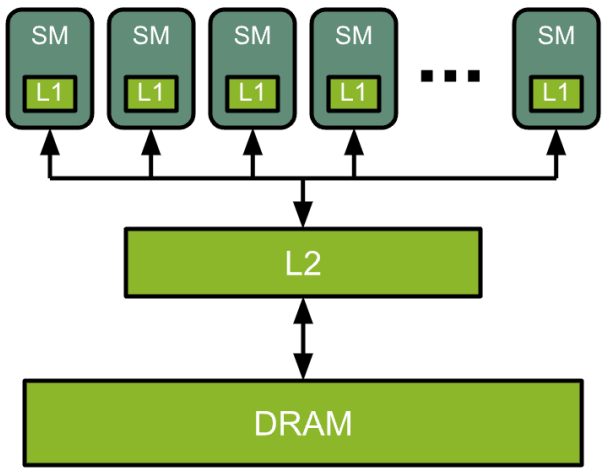
- **Memory Bandwidth**: The maximum number of items than can be moved from/to memory to/from the processor per second (4 items/s in the toy example)
- **Compute Bandwidth:** The maximum number of operations that can be done on the processor per second (8 items / second in the toy example)

These are properties of the hardware (not the algorithm).

Toy Example



Theoretical Peak Performance for Nvidia A100 Processors



https://docs.nvidia.com/deeplearning/

	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	
GPU Memory	80GB HBM2e	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s	2,039 GB/s

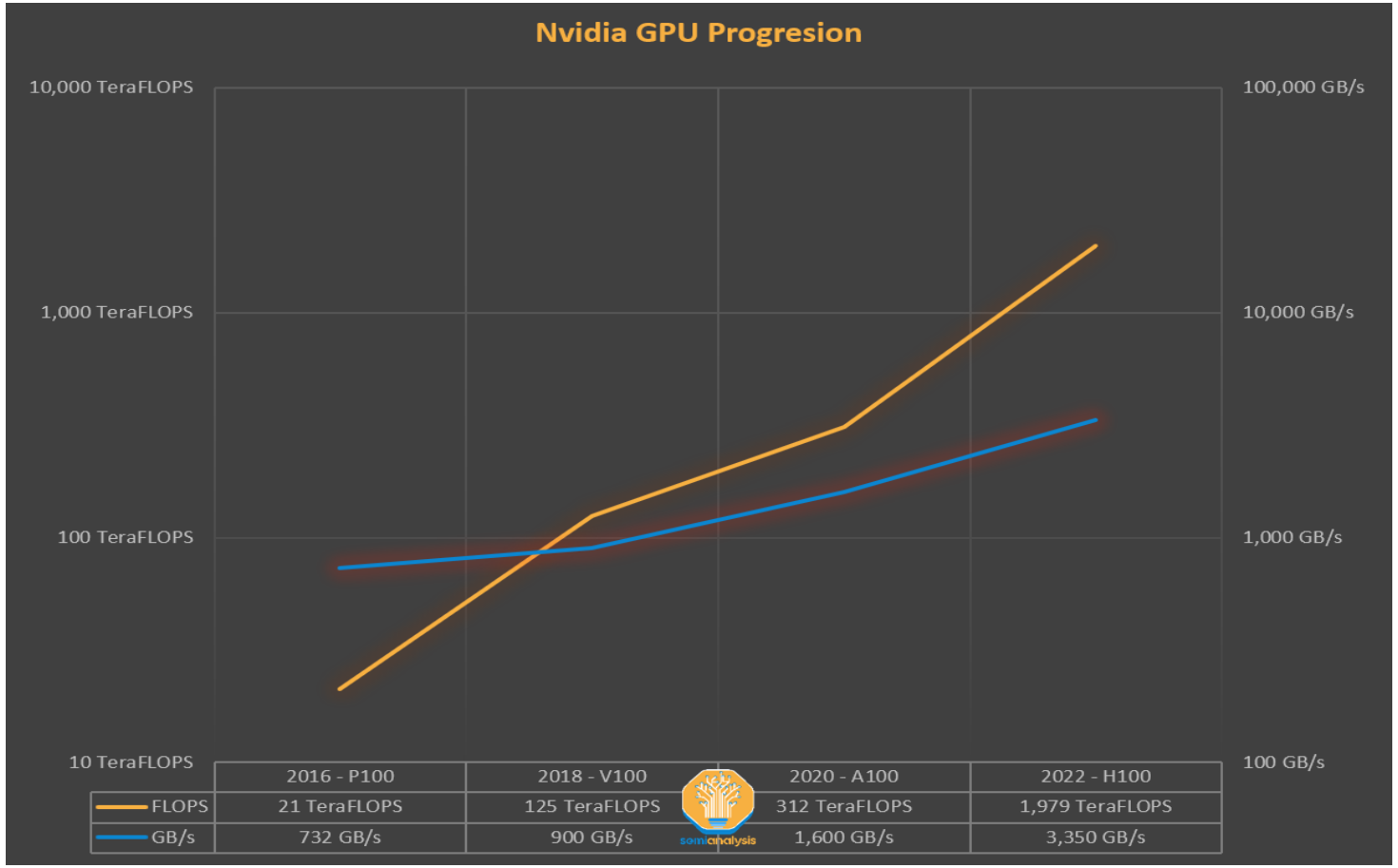
https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Peak Compute Vs Peak Bandwidth Mismatch

Two properties:

- We can do computation a lot faster than moving data to/from the processors.
- Memory is expensive, we currently cannot improve it as fast as compute!

What does this mean for performance?



https://www.semianalysis.com/p/nvidiaopenaitritonpytorch

Compute vs. Memory Bound

- Let T_{mem} be the time spent accessing memory, and T_{math} time is spent performing math operations.
- If memory and I/O are perfectly overlapped, then the total time of execution is:

$$max(T_{mem}, T_{math})$$

- The longer of the two times demonstrates what limits performance: If math time is longer, we say that the program is *math limited (compute bound)*, if memory time is longer then it is *memory limited (memory bound)*.

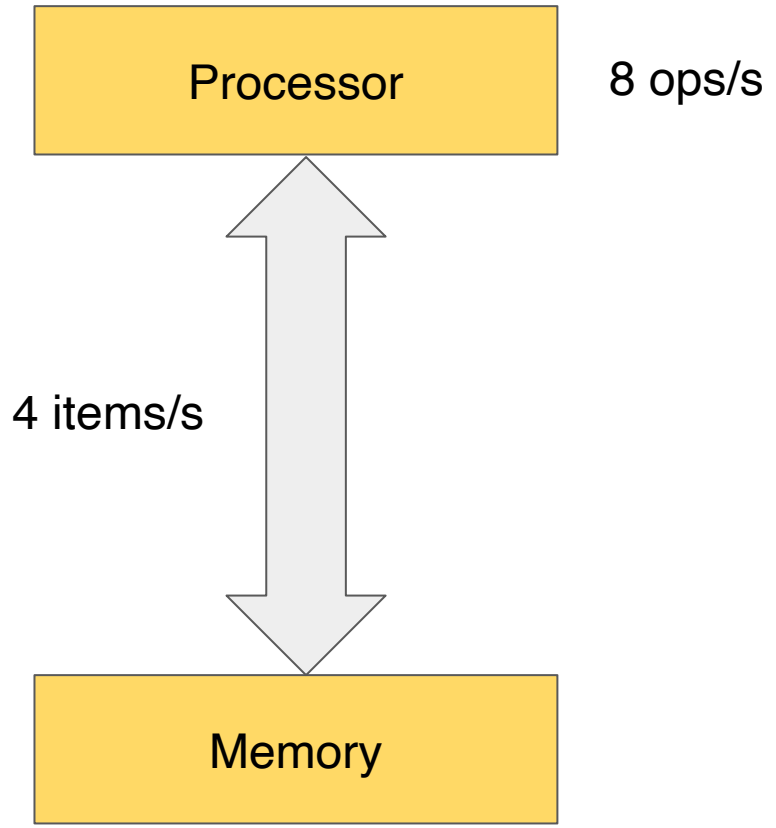
https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#gpu-execution

Peak Compute vs. Peak Bandwidth Mismatch

Suppose we implement an algorithm to run on this processor that does the following:

- Load 8 items from memory to the processor
- Compute 1 operation on each item

Assume compute overlaps with I/O. How much time does the implementation take to run?



Peak Compute vs. Peak Bandwidth Mismatch

Suppose we implement an algorithm to run on this processor that does the following:

- Load 8 items from memory to the processor.
- Compute 1 operation on each item

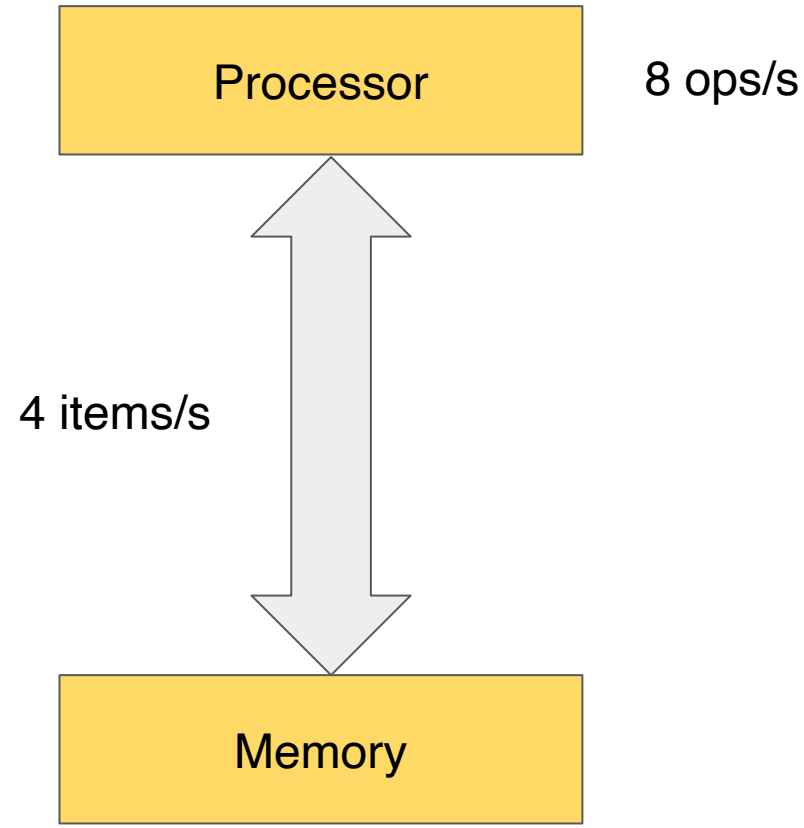
Assume compute overlaps with I/O. How much time does the implementation take to run?

Memory_time = 2s; Compute_time = 1s

Total Time = max (2s, ~~1s~~)

For sufficiently large number of items:

- 4 items arrive at the processor per second at steady state
- The processor can do 8 ops every second
- The processor only does 4 ops (1 per item) every second
- 50% compute utilization and 100% bandwidth utilization!



Peak Compute Vs Peak Bandwidth Mismatch

Increasing Compute Speed to 16 ops/s

What is the new run time?

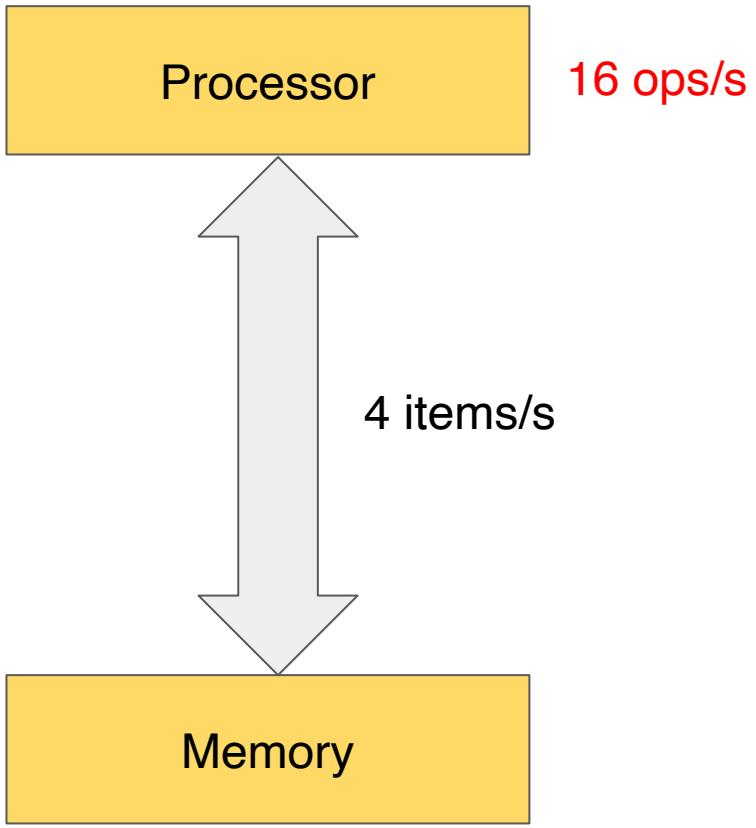
Memory_time = 2s; Compute_time = 0.5s

Total Time = max (2s, ~~0.5s~~)

- The total run time remains unchanged

For sufficiently large number of items:

- 4 items arrive at the processor per second at steady state
- The processor can do 16 ops every second
- The processor only does 4 ops (1 per item) every second
- 25% compute Utilization and 100% bandwidth utilization!
- Increasing compute speed does not improve performance.



Peak Compute vs. Peak Bandwidth Mismatch

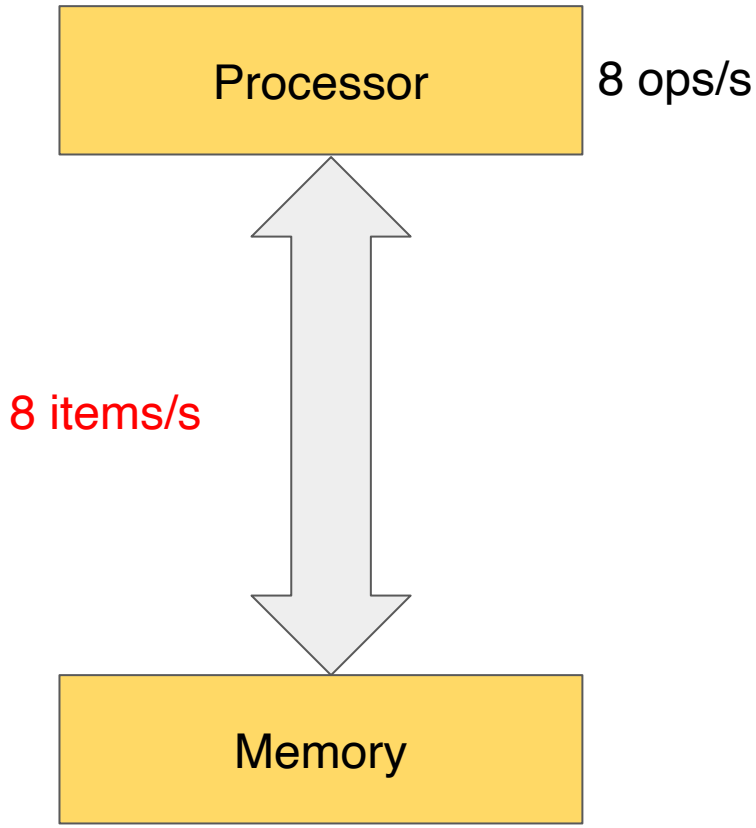
Increasing Memory Bandwidth to 8 items/s

What is the new run time?

Memory_time = 1s ; Compute_time = 1s

Total Time = max (1s, 1s) = 1s

The total run time reduces by 50%!



For sufficiently large number of items:

- 8 items arrive at the processor per second at steady state
- The processor can do 8 ops every second
- The processor only does 8 ops (1 per item) every second
- 100% compute and bandwidth utilization!

Lesson: Memory bandwidth is the main bottleneck and not compute speed (Memory Bound).

Peak Compute vs. Peak Bandwidth Mismatch

Suppose we now implement a new algorithm to run on this processor that does the following:

- Load 4 items from memory to the processor.
- Compute *4 operations* on each item
- Write output of all 4 items back to Memory

How much time does the implementation take to run?

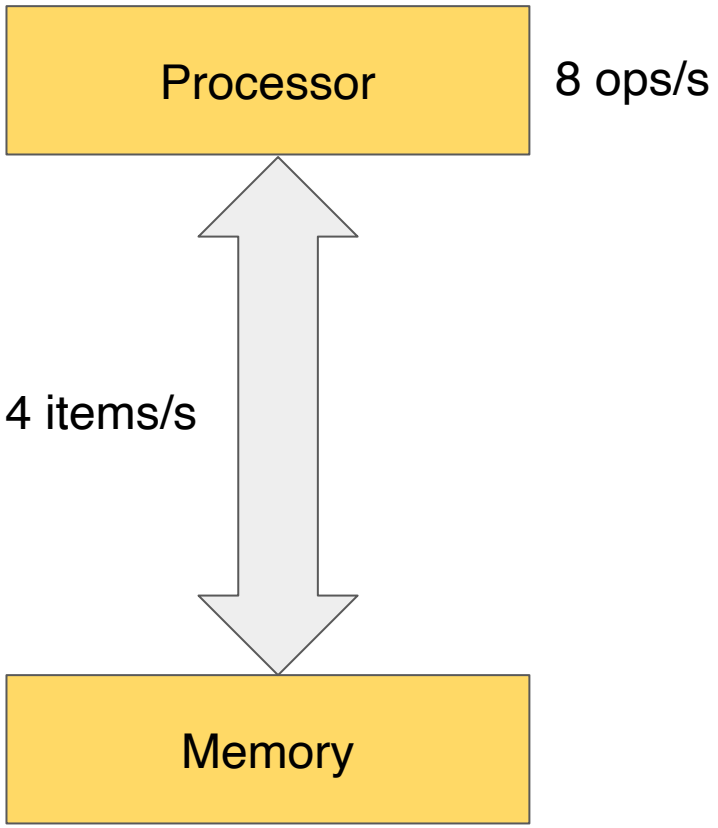
Memory_time = 1s ; Compute_time = 2s

Total Time = max (1s, 2s) = 2s

For sufficiently large number of items:

- 4 items arrive at the processor per second at steady state
- Processor needs to do 16 ops (4 per item) per second
- Processor can only do 8 ops per second
- 100% compute; But only 50% bandwidth utilization!

Lesson: Compute is the main bottleneck and not memory bandwidth (Compute Bound).



Compute Bound vs. Bandwidth Bound

- Compute bound if $T_{math} > T_{mem}$
- Memory bound if $T_{mem} > T_{math}$
- $T_{math} = \frac{\text{\# ops}}{\text{Compute Bandwidth}}$ $T_{mem} = \frac{\text{\# bytes accessed}}{\text{Memory Bandwidth}}$

https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#gpu-execution

Compute Bound vs. Bandwidth Bound

- Compute Bound: $\frac{\#\ \text{ops}}{\#\ \text{bytes accessed}} > \frac{\text{Compute Bandwidth}}{\text{Memory Bandwidth}}$
- Memory Bound: $\frac{\#\ \text{ops}}{\#\ \text{bytes accessed}} < \frac{\text{Compute Bandwidth}}{\text{Memory Bandwidth}}$

https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#gpu-execution

Arithmetic (Operational) Intensity

- Sometimes referred to as operational intensity

- How many operations we perform for each byte moved from memory

$$\text{Arithmetic Intensity} = \frac{\text{Total Number Of Operations}}{\text{Number Of Bytes To and From Memory (Write + Read)}}$$

Compute Bandwidth to Memory Bandwidth Ratio

The A100, a currently-popular GPU from NVidia supports:

- 1,935 GB/s memory bandwidth
- 312 TFLOPS for FP16 (compute bandwidth)

Peak FLOPS (compute bandwidth) to memory bandwidth:

= (312 TFLOPS)/(1,935 GB/s)

= 161 (FLOPs/Byte)

	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	
GPU Memory	80GB HBM2e	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s	2,039 GB/s

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

We can compute the arithmetic intensity of common algorithms…

Example: Matrix Multiplication

Compute

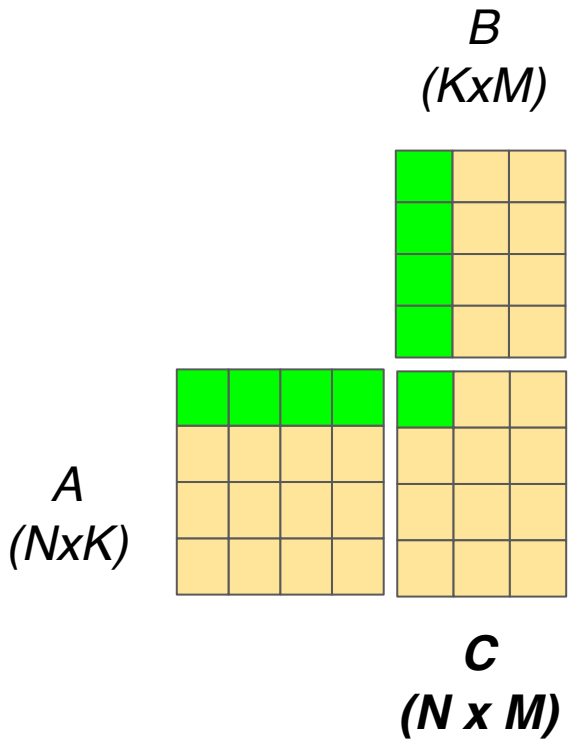
The matrix **C** has N x M entries, and each is obtained by taking the dot product of vectors (A_i, B_j) of dimension K

Dot product is K additions and K multiplications, totalling 2K FLOPs:

$$C_{ij} = A_{i1}B_{1j} + \ldots + A_{ik}B_{kj}$$

For every element in **C**, we perform:
k multiplications (one for each term in the sum)
k−1 additions (to sum up the products)
Roughly, 2K FLOPS

The total number of FLOPs for matmul is: 2MNK



Example: Matrix Multiplication

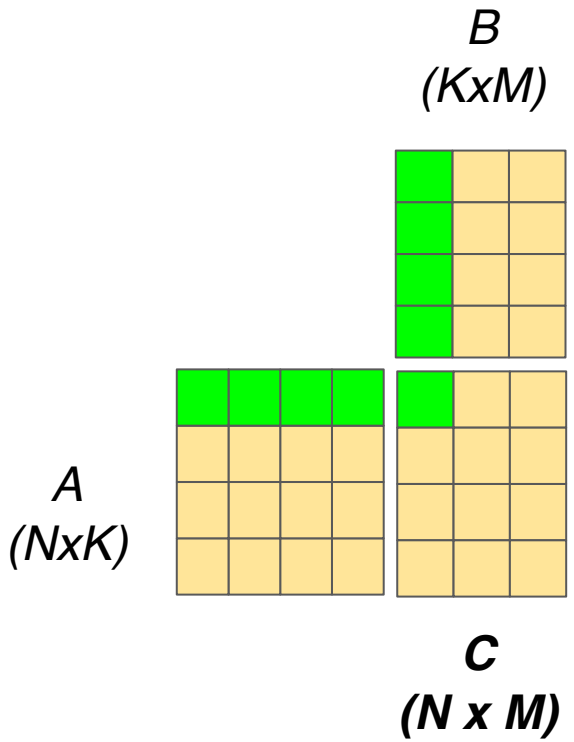
Memory

I/O includes:

- Read A: N x K
- Read B: K x M
- Write to C: N x M

Suppose each value is in FP16, i.e. 2 bytes.

Total byte accesses: 2 bytes * (K x M + N x K + N x M)



Example: Matrix Multiplication

The total number of FLOPs for matmul is: 2MNK

The total number of byte accesses is: 2 (K x M + N x K + N x M)

So:

$$\text{Arithmetic Intensity of MatMul} = (2\,MNK)/(2\,(KM + NK + NM))$$

Example: Matrix Multiplication

We saw the A100's arithmetic ratio is 161 FLOPs/Byte

If M = K = 8192, N = 128: (2 MNK) / (2 (KM + NK + NM)) = 124.1

Here 124.1 < 161 so we are memory bound.

Example: Matrix Multiplication

We saw the A100's arithmetic ratio is 161 FLOPs/Byte

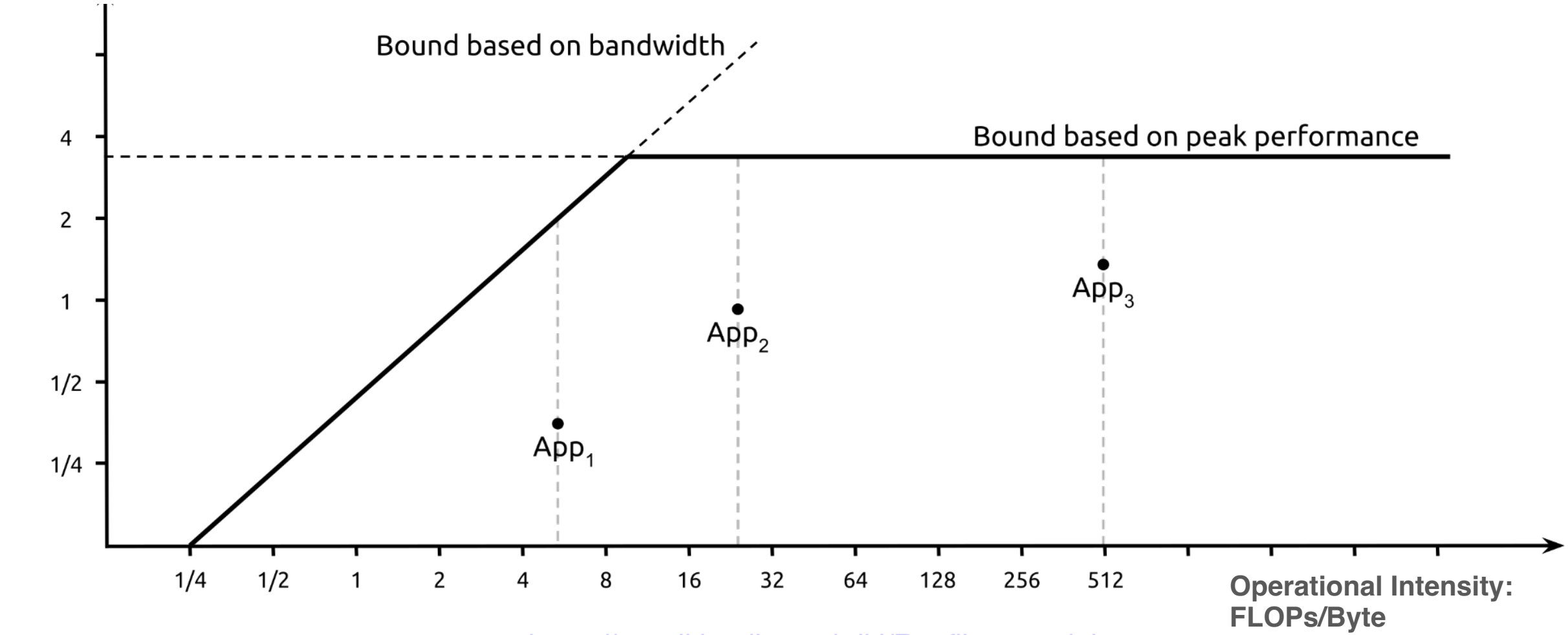
If M = K = 8192, N = 128: (2 MNK) / (2 (KM + NK + NM)) = 124.1

Here 124.1 < 161 so we are memory bound.

If M = K = N = 8192: (2 MNK) / (2 (KM + NK + NM)) = 2730.6

Here 2730.6 > 161 so we are compute bound.

Roofline Diagrams



https://en.wikipedia.org/wiki/Roofline_model

Simple Principles for High Performance

Simple Principles For Achieving High Performance

We want to make maximum use of compute and bandwidth capability of our device.

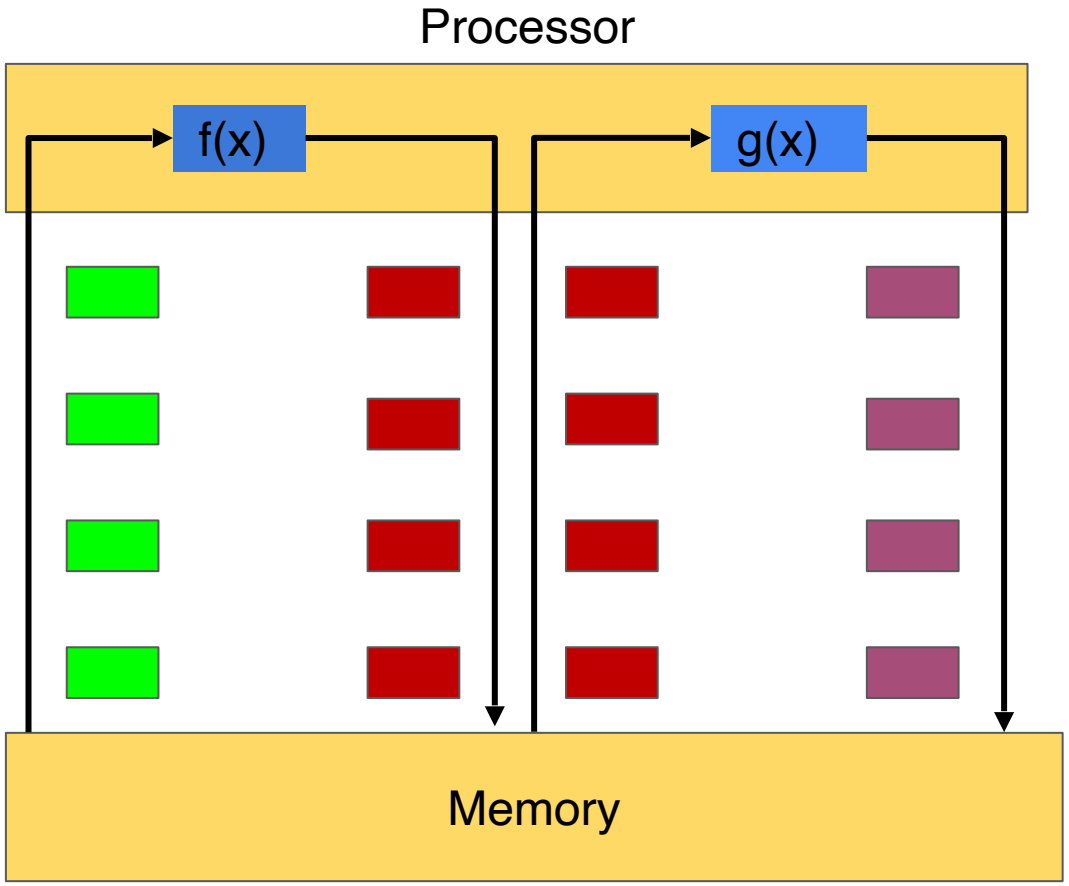
- Fusion
  - Save trips to/from memory by performing composite operations on data on processing units.
- Parallelization
  - Expose as much work as possible to saturate all parallel processing units.
- Blocking/Tiling
  - Assign tiles/blocks of work to parallel processing units to exploit as much locality as possible.
- Caching Vs Recomputation
  - For compute bound workloads, it makes sense to cache, for memory bound workloads it makes sense to recompute.
- Pipelining
  - Overlap computation with memory reads/writes to avoid stalls.
- Hardware Specific Optimizations
  - Intrinsics, New instructions, etc.

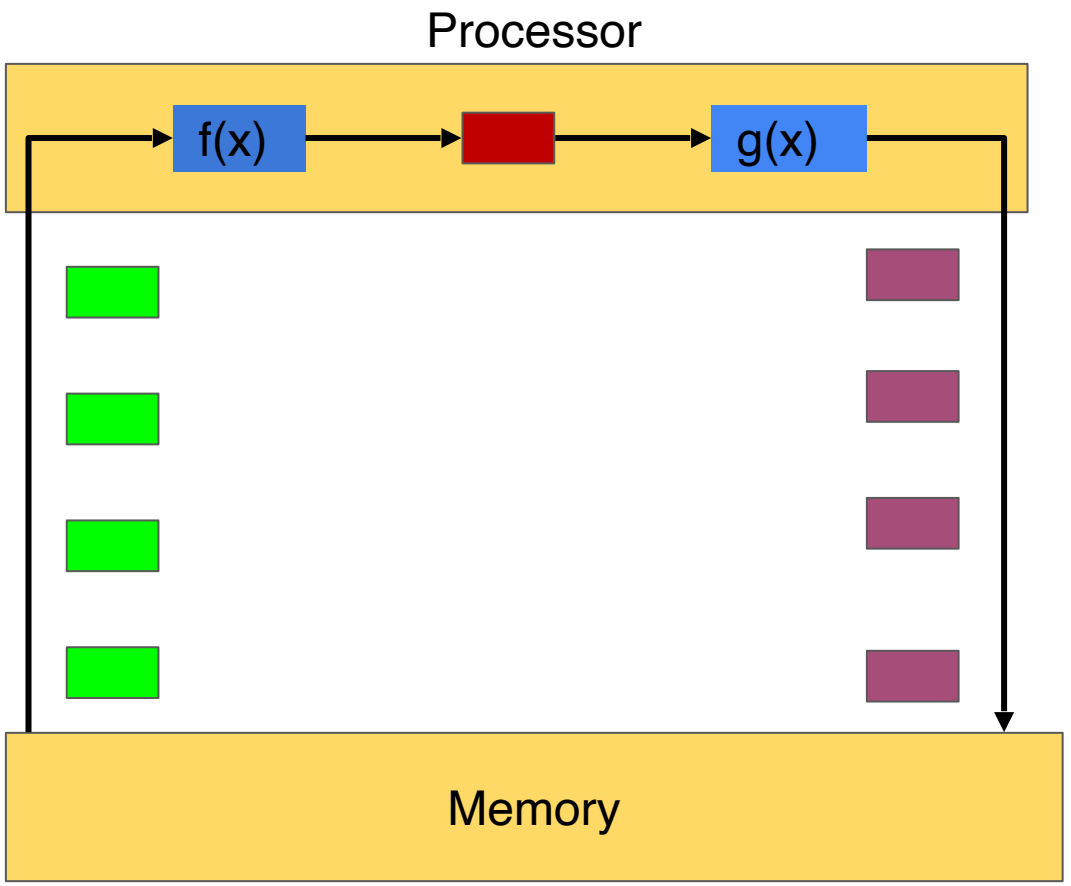
Simple Principles For Achieving High Performance

We want to make maximum use of compute and bandwidth capability of our device.

- **Fusion**
  - Save trips to/from memory by performing composite operations on data on processing units.
- Parallelization
  - Expose as much work as possible to saturate all parallel processing units.
- Blocking/Tiling
  - Assign tiles/blocks of work to parallel processing units to exploit as much locality as possible.
- Caching Vs Recomputation
  - For compute bound workloads, it makes sense to cache, for memory bound workloads it makes sense to recompute.
- Pipelining
  - Overlap computation with memory reads/writes to avoid stalls.
- Hardware Specific Optimizations
  - Intrinsics, New instructions, etc.

Fusion





Input

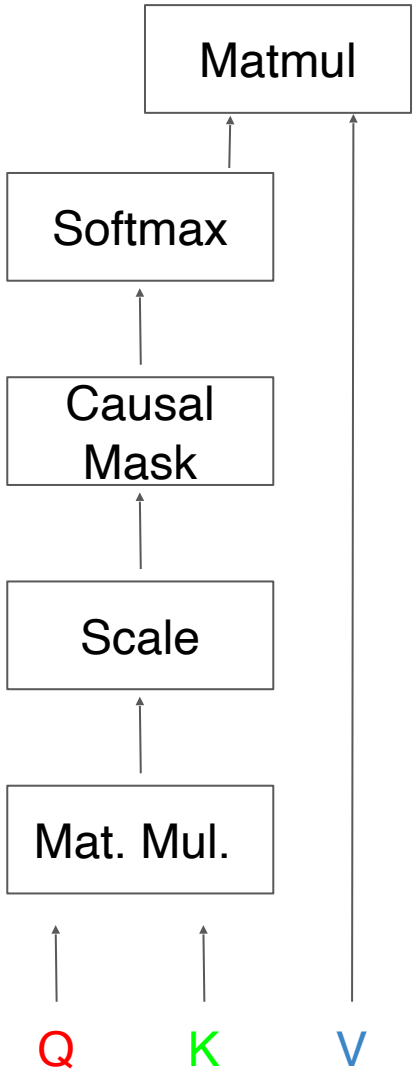
Intermediate output

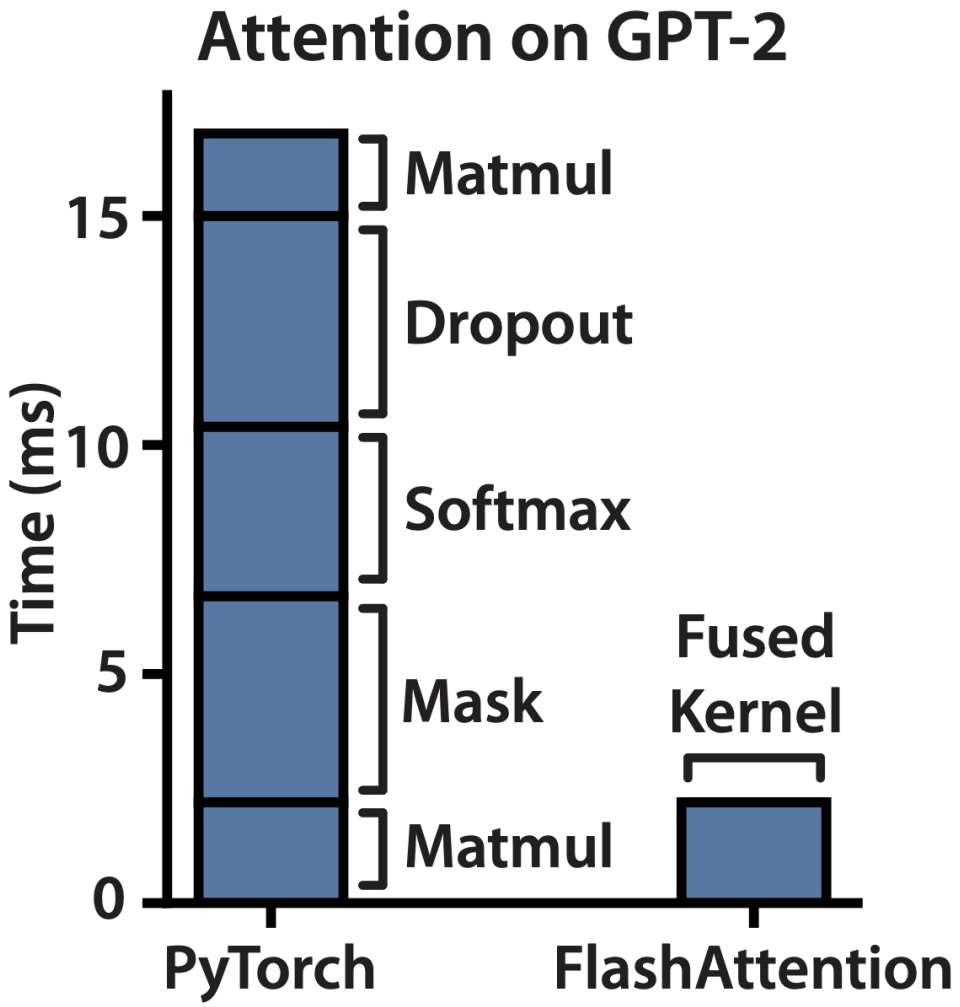
Output

- Fusion prevents unnecessary trips to memory

- This is critically for memory bound applications

Fusion for attention





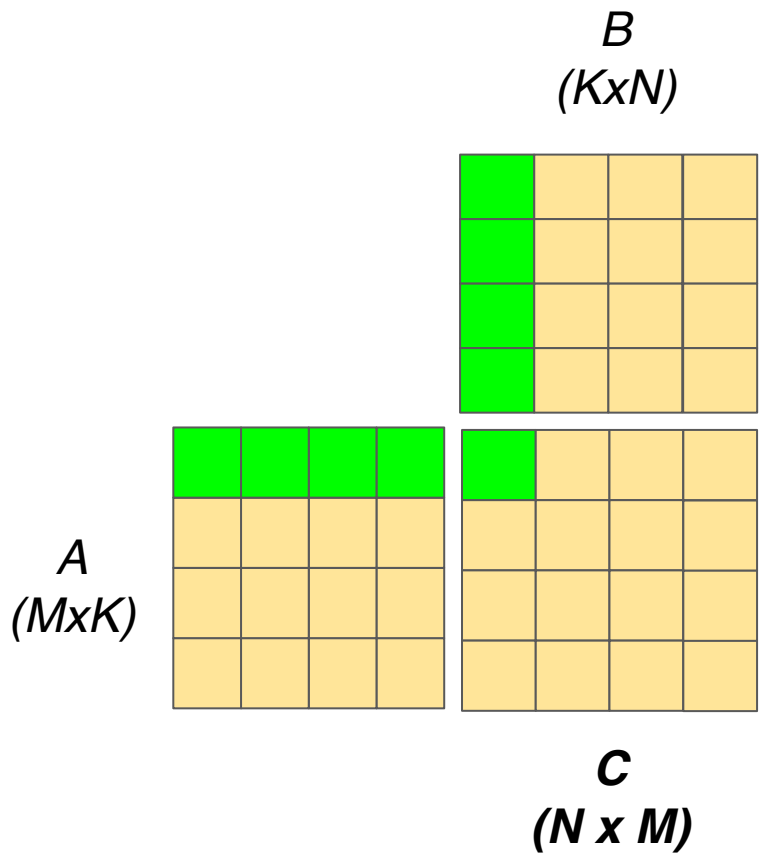
FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

Simple Principles For Achieving High Performance

We want to make maximum use of compute and bandwidth capability of our device. Solve useful techniques to achieve this:

- Fusion
  - Save trips to/from memory by performing composite operations on data on processing units.
- **Parallelization**
  - Expose as much work as possible to saturate all parallel processing units.
- **Blocking/Tiling**
  - Assign tiles/blocks of work to parallel processing units to exploit as much locality as possible.
- Caching Vs Recomputation
  - For compute bound workloads, it makes sense to cache, for memory bound workloads it makes sense to recompute.
- Pipelining
  - Overlap computation with memory reads/writes to avoid stalls.
- Hardware Specific Optimizations
  - Intrinsics, New instructions, etc.

Parallelization



- An A100 Tensor core GPU has 108 Streaming Multiprocessors (https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/)
- We would like to make sure that each SM has as much work as possible!
- We can parallelize across the output.
- We need at least 108 output items for all Streaming Multiprocessors to be in use on the A100!

Blocking/Tiling: Matrix Multiplication (Naive)

Consider a naive implementation where each thread computes a single value in the output **C.**

Compute:

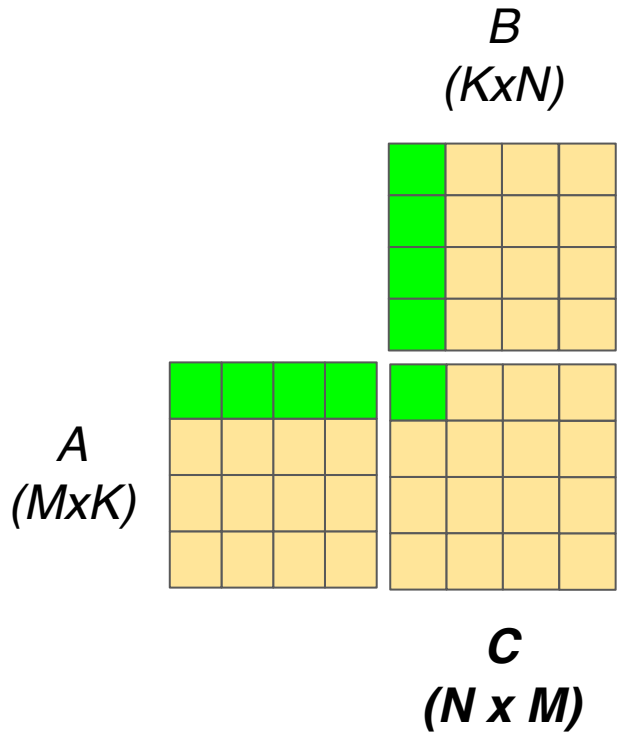
Dot product of a row of A and a column of B (2K total FLOPS)

Memory:

- Read a row of A and column of B
- Write a single element of C
- Suppose each value is in FP16, i.e. 2 bytes.

Total byte accesses: 2(2K + 1)

Arithmetic intensity = 2k / (2(2k + 1))!!



Blocking/Tiling: Matrix Multiplication (1D-Tiling)

In a 1D tiling version, each thread computes a tile two items in the output **C**.

Compute:

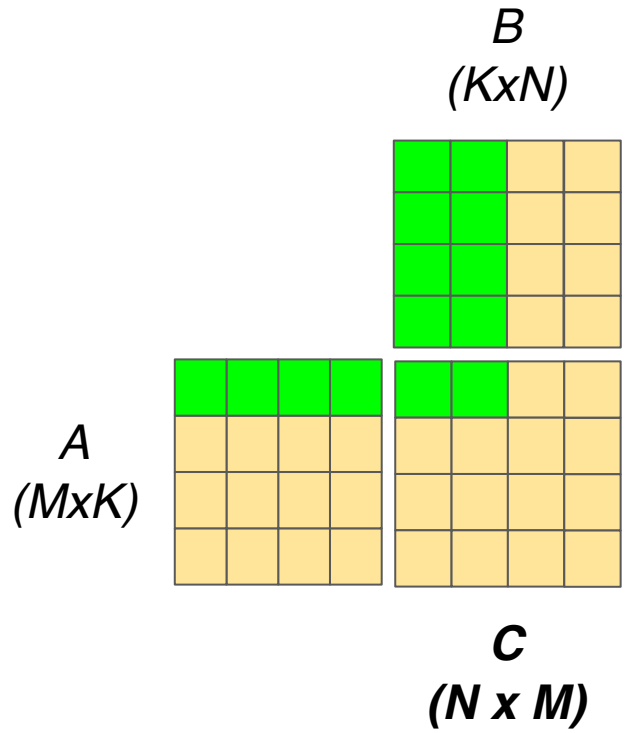
Dot product of a row of A and 2 columns of B (4K total FLOPS)

Memory:

- Read a row of A and 2 columns of B
- Write 2 elements of C
- Suppose each value is in FP16, i.e. 2 bytes.

Total byte accesses: 2(3K + 2)

Arithmetic intensity = 4k / ((3k + 2))!!



Blocking/Tiling: Matrix Multiplication (2D-Tiling)

In a 2D tiling version, each thread computes a 2D tile of the output **C**.

Compute:

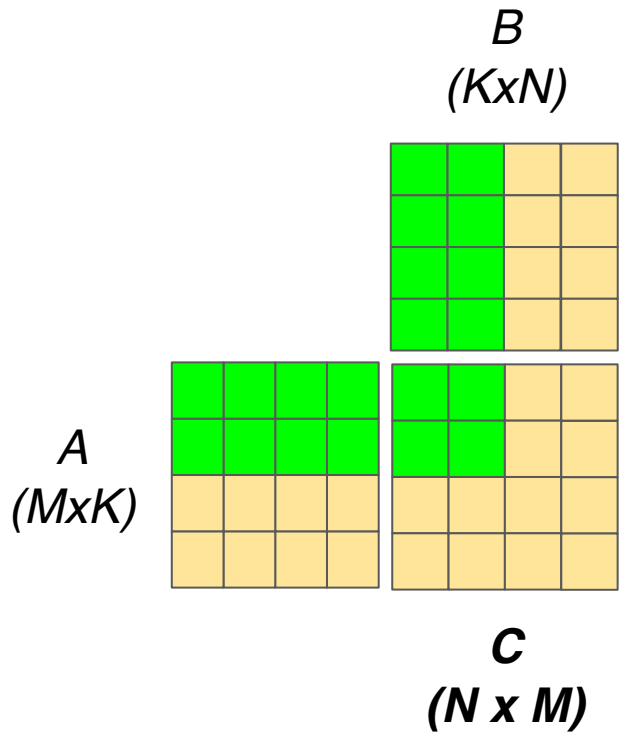
Dot product of 2 rows of A and 2 columns of B (8K total FLOPS)

Memory:

- Read 2 rows of A and 2 columns of B
- Write a 2x2 tile of elements of C
- Suppose each value is in FP16, i.e. 2 bytes.

Total byte accesses: 2(4K + 2*2)

Arithmetic intensity = 8k / (2(4k + 2*2))!!



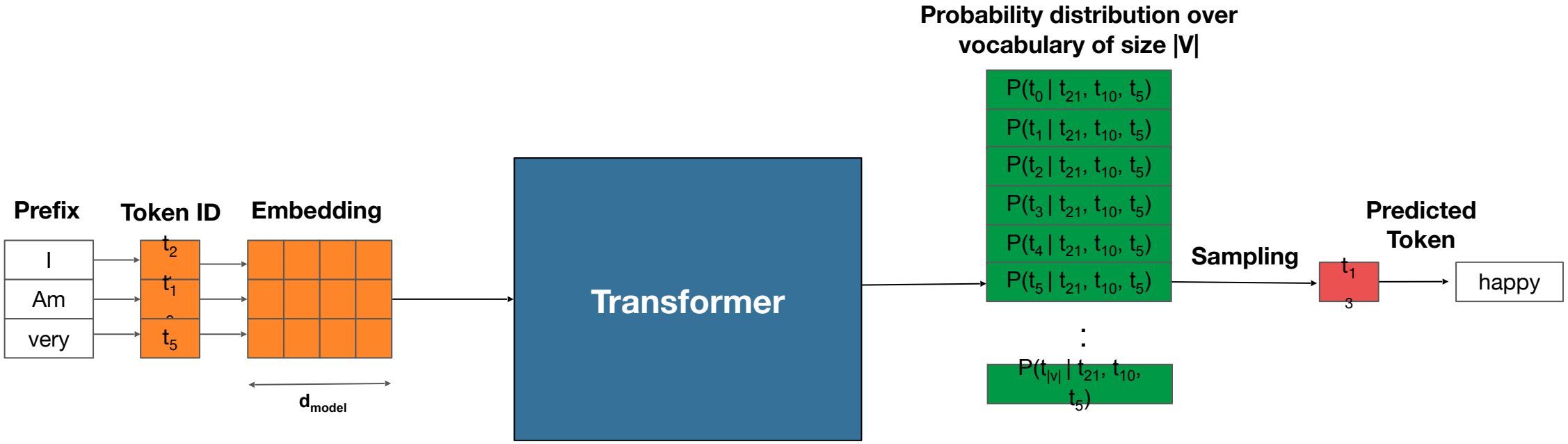
Simple Principles For Achieving High Performance

We want to make maximum use of compute and bandwidth capability of our device.

- Fusion
  - Save trips to/from memory by performing composite operations on data on processing units.
- Parallelization
  - Expose as much work as possible to saturate all parallel processing units.
- Blocking/Tiling
  - Assign tiles/blocks of work to parallel processing units to exploit as much locality as possible.
- **Caching Vs Recomputation**
  - For compute bound workloads, it makes sense to cache, for memory bound workloads it makes sense to recompute.
- Pipelining
  - Overlap computation with memory reads/writes to avoid stalls.
- Hardware Specific Optimizations
  - Intrinsics, New instructions, etc.

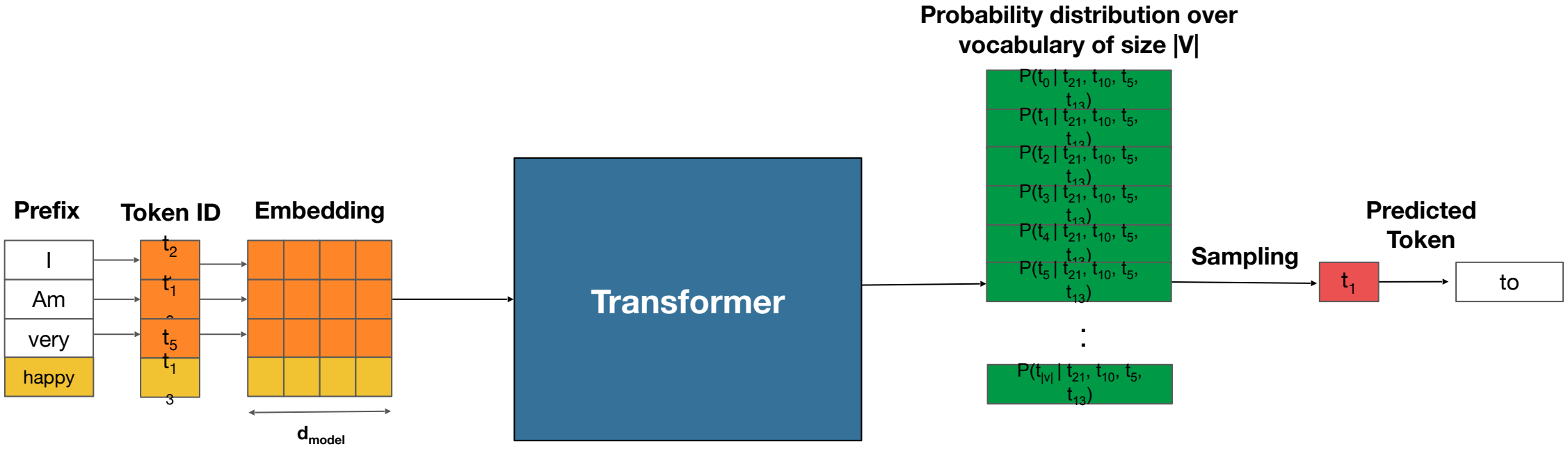
Caching vs Recomputaton: Autoregressive Generation

Given the current prompt sequence, the model predicts the next word:



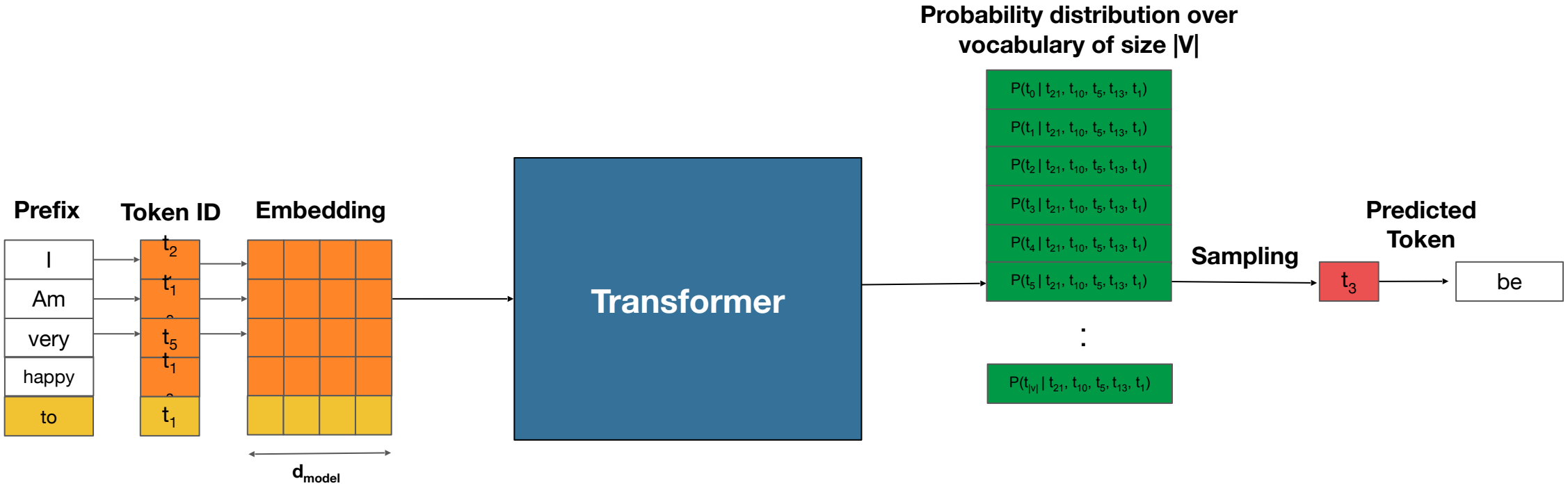
Caching vs Recomputaton: Autoregressive Generation

Given the current prompt sequence, the model predicts the next word:



Caching vs Recomputaton: Autoregressive Generation

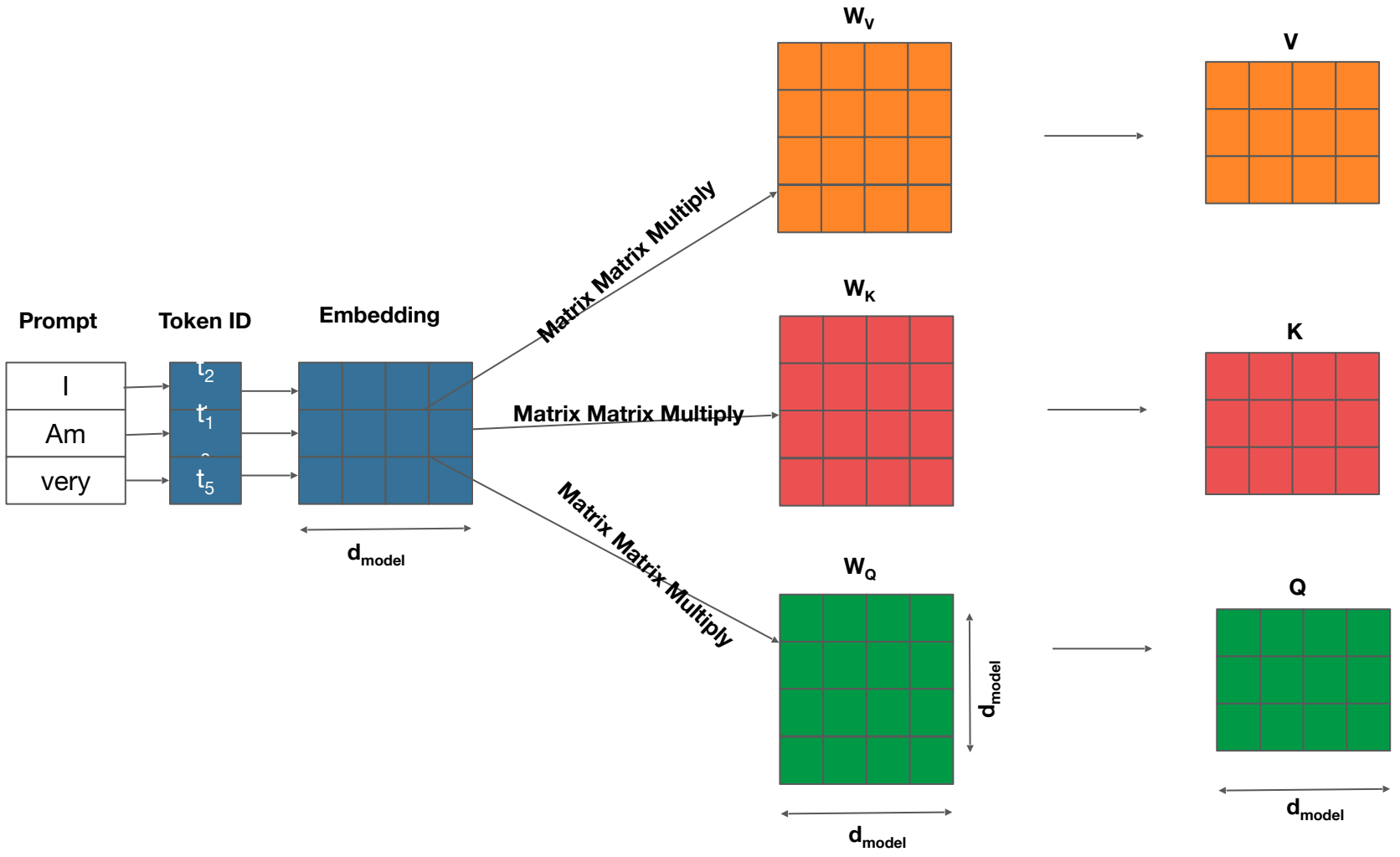
Given the current prompt sequence, the model predicts the next word:



Attention Layer During Autoregressive Generation

First we compute attention scores using the input prompt tokens.

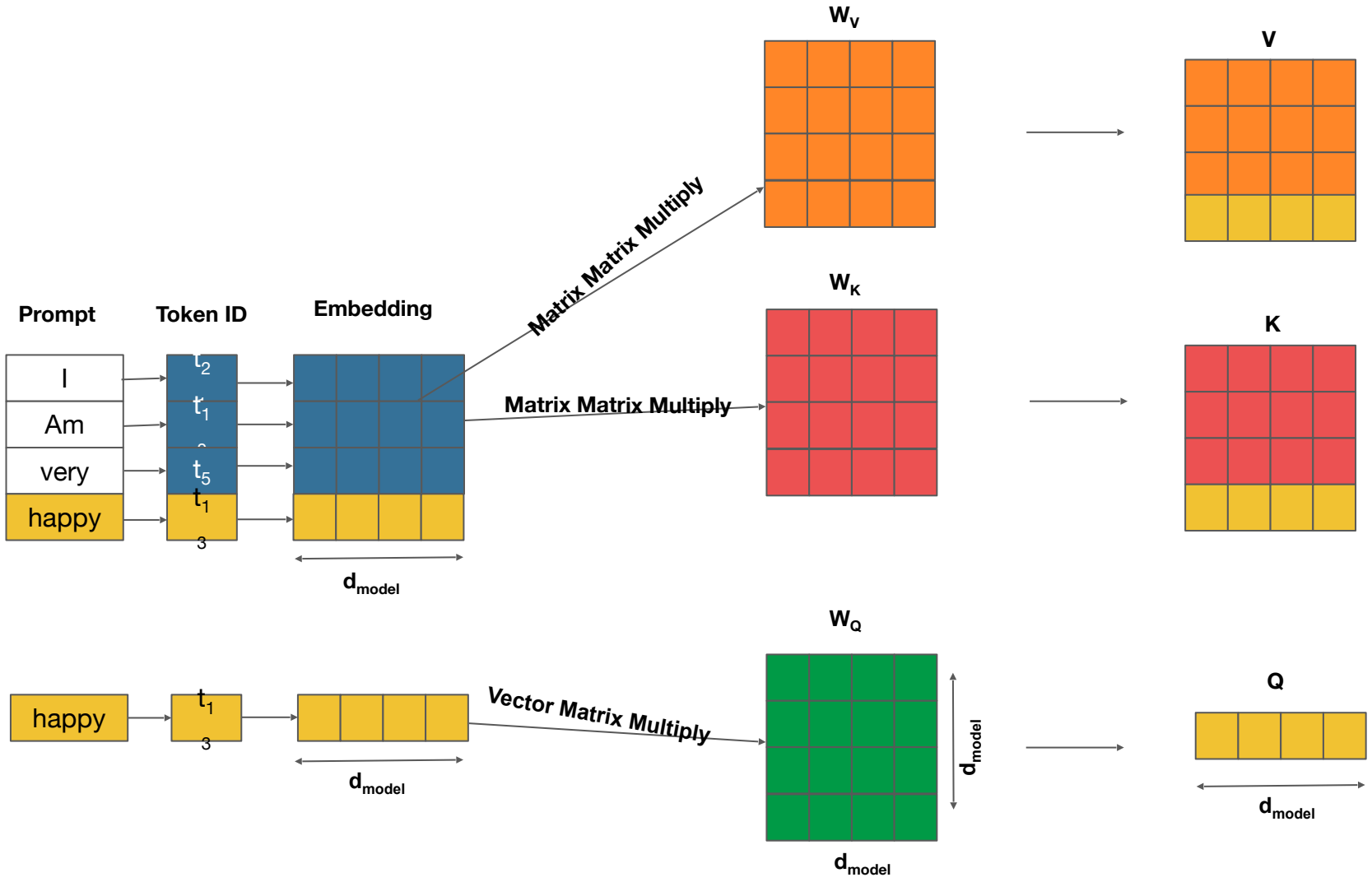
We obtain our Q, K, V vectors for each token embedding.



Attention Layer During Autoregressive Generation

Now we want to compute the next set of probabilities, conditioned on the new sequence:

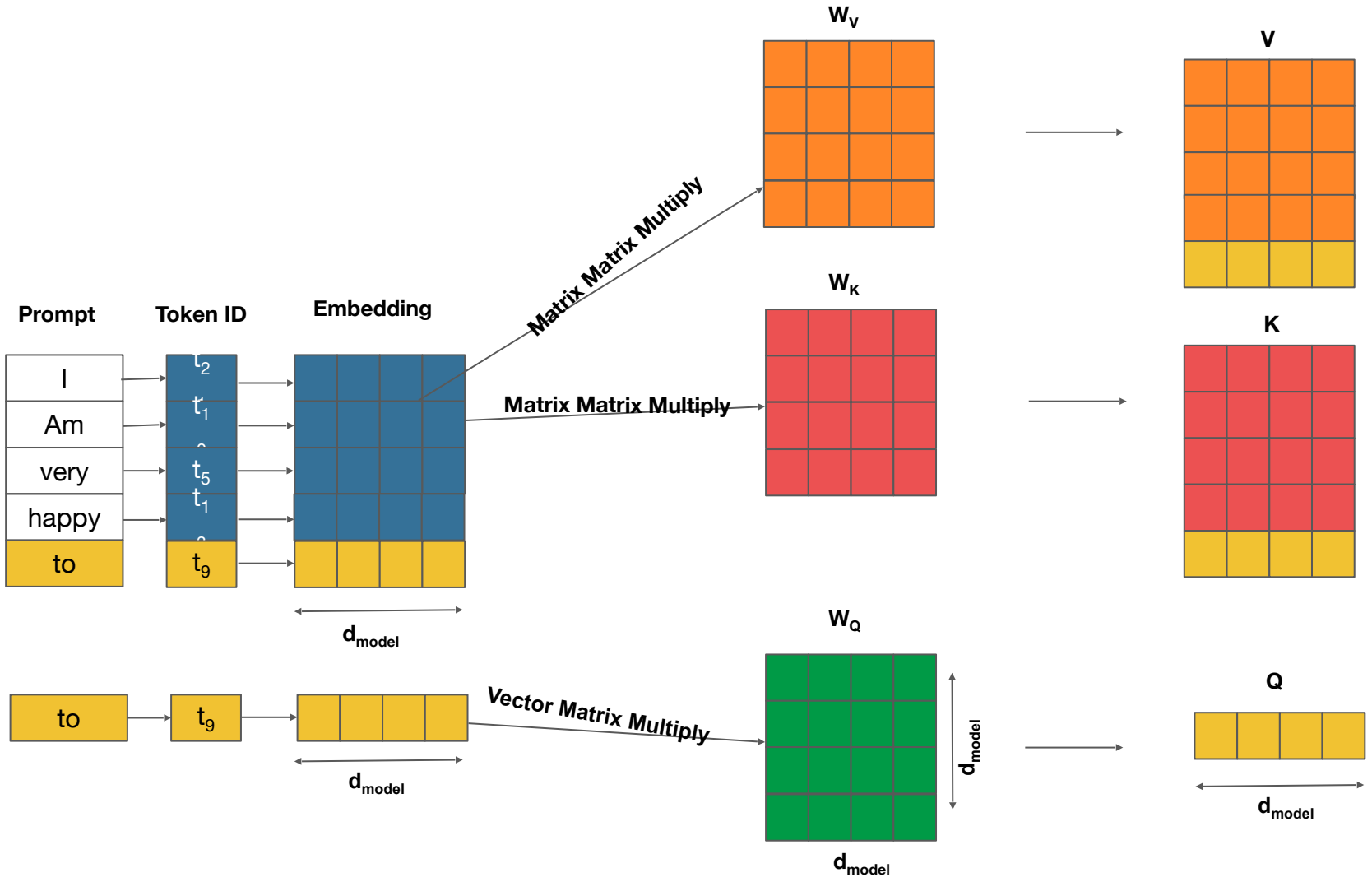
$P(t_i \mid t_{21}, t_{10}, t_5, \mathbf{t_{13}}) = ?$



Attention Layer During Autoregressive Generation

Now we want to compute the next set of probabilities, conditioned on the new sequence:

$P(t_i \mid t_{21}, t_{10}, t_5, t_{13}, \mathbf{t_9}) = ?$



Attention FLOPS

For each token we generate (N = sequence length so far):

1. Computing K, V for the full sequence: $2x(2Nd^2)$
2. Computing Q for our current token: $(2d^2)$
3. Multiplying $S = QK^T$: $(2Nd)$
4. Letting A = Softmax(S) , multiplying AV: $(2Nd)$

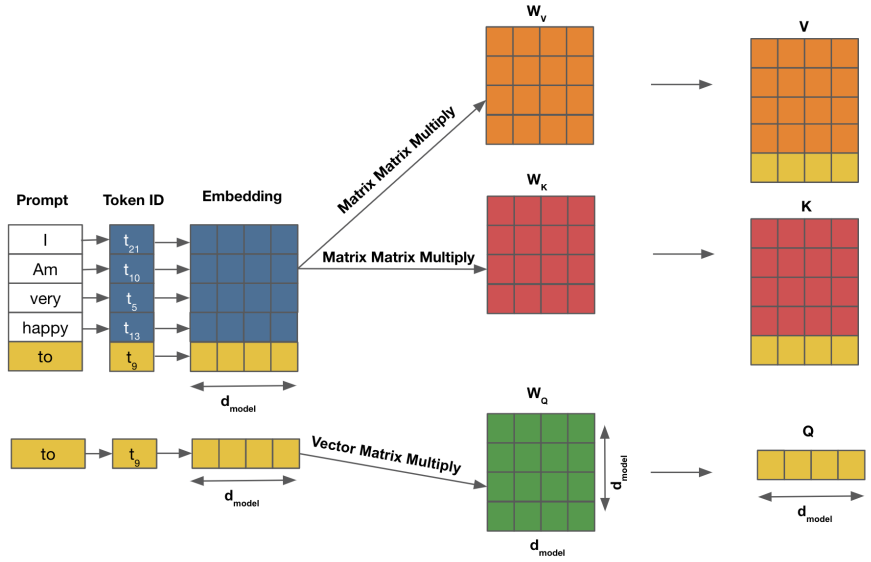
But there's a ton of repeated processing if we generate one token at a time…

KV Caching for Autoregressive Models

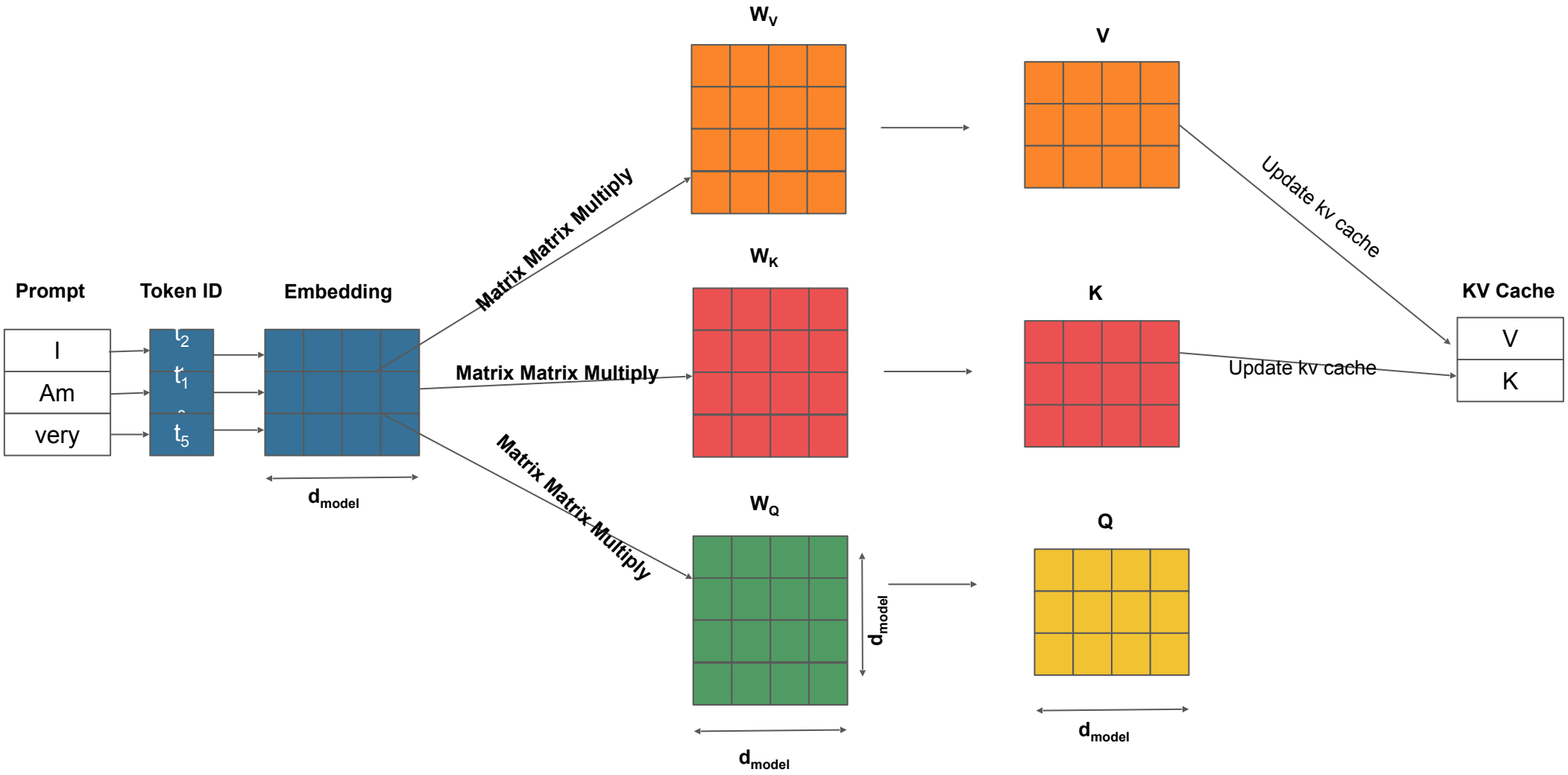
Key idea: the keys (K) and values (V) for the pre-existing tokens in the sequence are not changing – their values are only impacted by the tokens that come before them — so we can **reuse** them every time we sample a new token.

For each token we generate, letting N be the length of the current sequence:

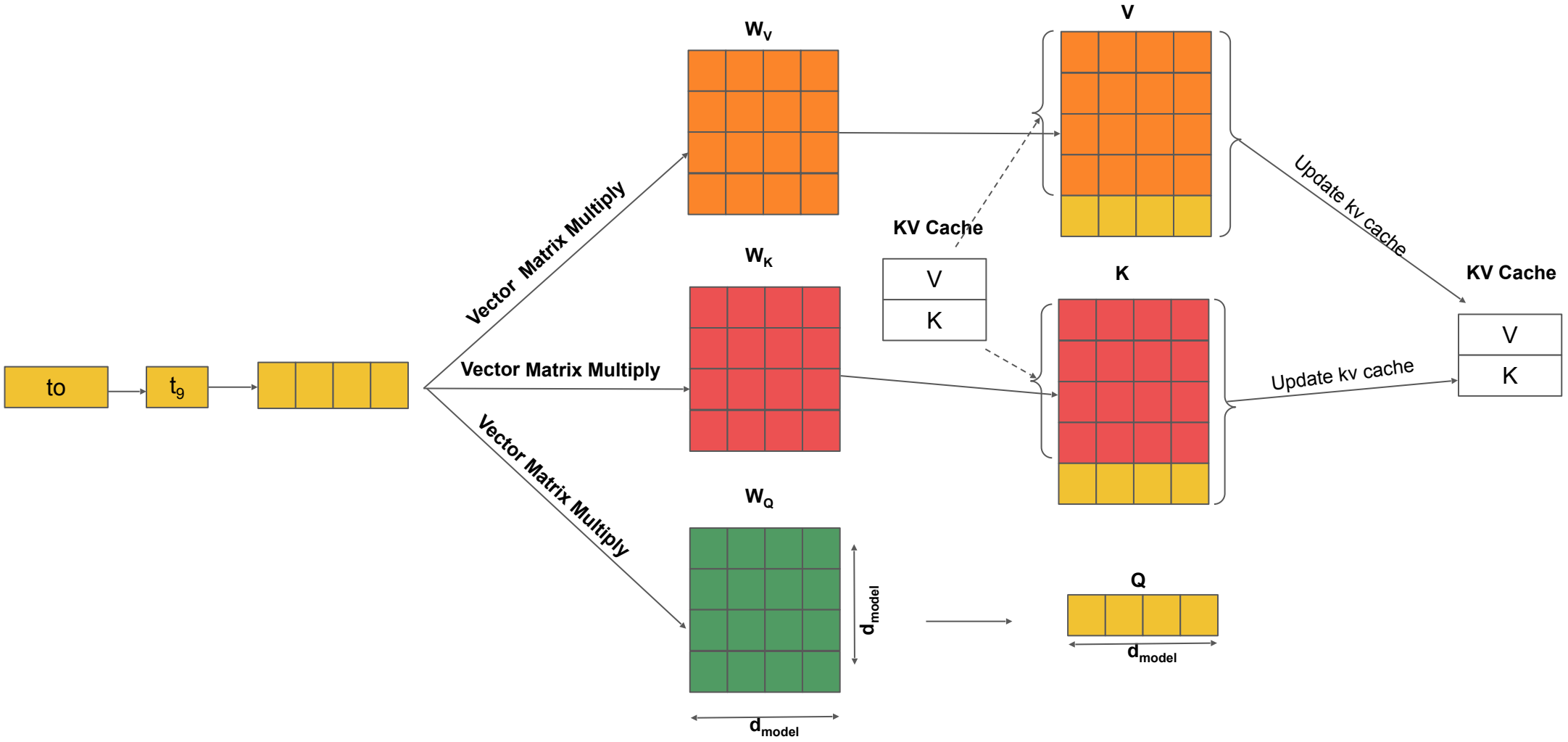
1. ~~Computing K, V for the full sequence: $2x(2Nd^2)$~~
2. Computing K, Q, V for our current token: $3x(2d^2)$
3. Multiplying $S = QK^T$: (2Nd)
4. Letting A = Softmax(S) , multiplying AV: (2Nd)



Attention Layer During Autoregressive Generation



Attention Layer During Autoregressive Generation



Simple Principles For Achieving High Performance

We want to make maximum use of compute and bandwidth capability of our device.

- Fusion
  - Save trips to/from memory by performing composite operations on data on processing units.
- Parallelization
  - Expose as much work as possible to saturate all parallel processing units.
- Blocking/Tiling
  - Assign tiles/blocks of work to parallel processing units to exploit as much locality as possible.
- Caching Vs Recomputation
  - For compute bound workloads, it makes sense to cache, for memory bound workloads it makes sense to recompute.
- Pipelining
  - Overlap computation with memory reads/writes to avoid stalls.
- **Hardware Specific Optimizations**
  - **Intrinsics, New instructions, etc.**

DPX

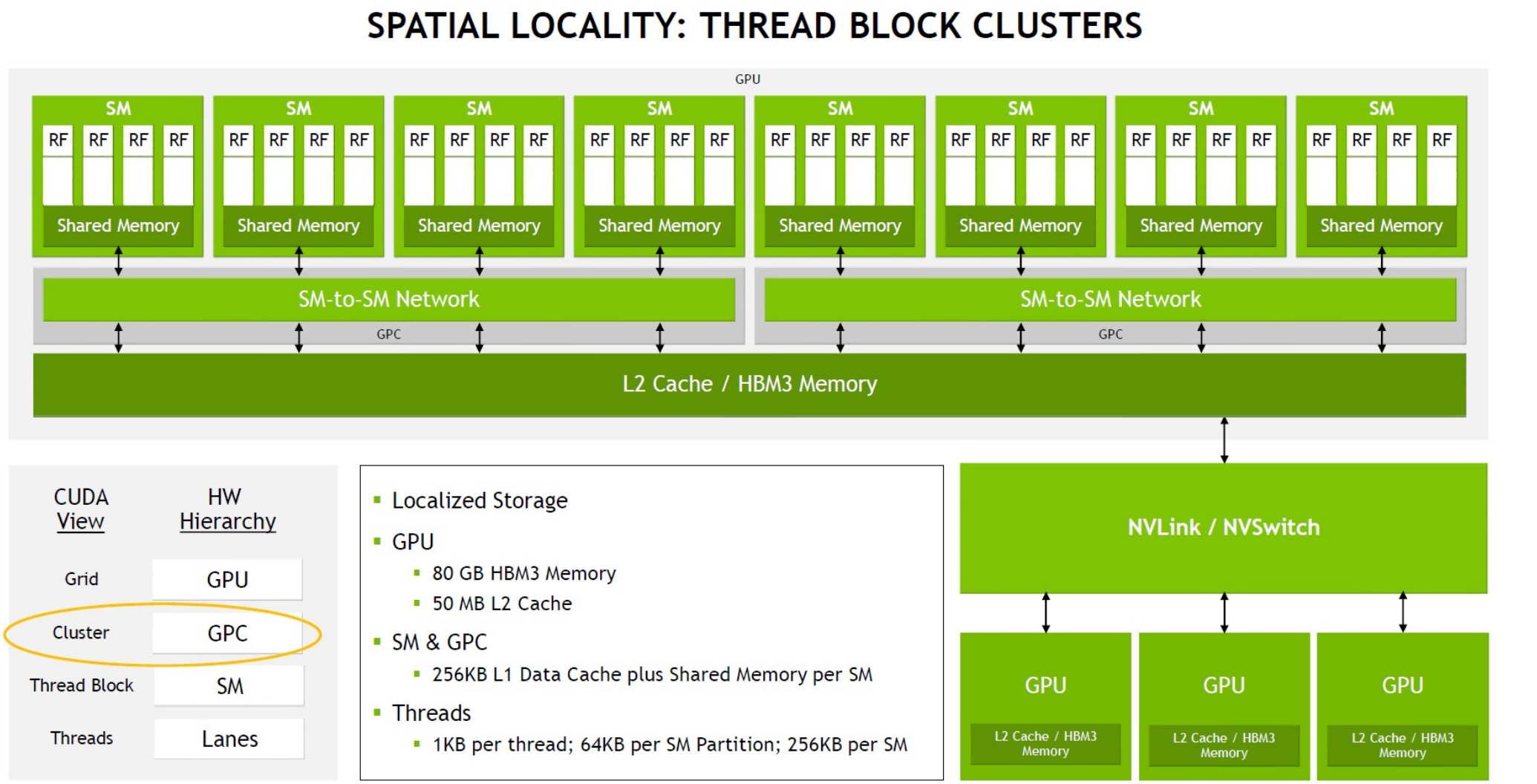
DPX Instructions accelerate dynamic programming algorithms by up to 7x over the A100 GPU.

Examples:

Smith-Waterman algorithm for genomics processing

Floyd-Warshall algorithm used to find optimal routes
for a fleet of robots through a dynamic warehouse environment.

DSM



NVIDIA H100 Spacial Locality Thread Block Clusters - ServeTheHome