Lecture 4

GPUs

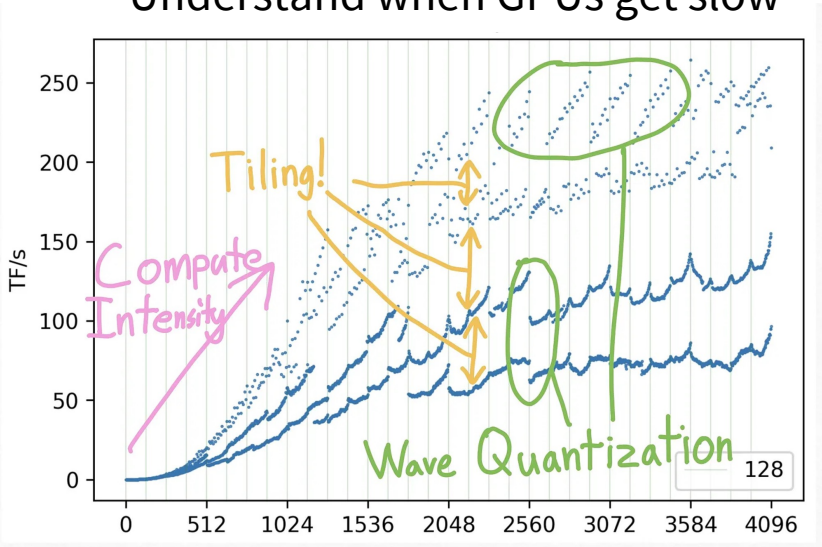
CS336

Tatsu H

Outline and goals

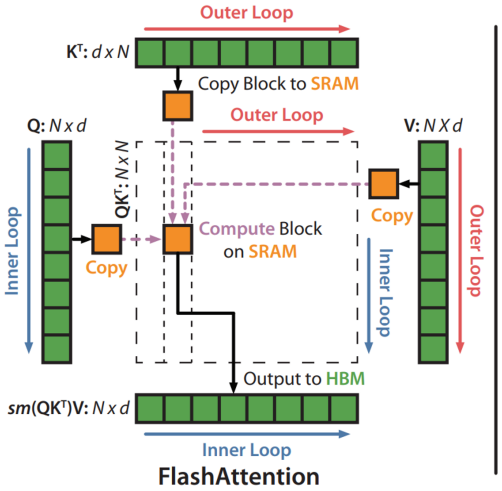
- Make CUDA and GPUs less magic

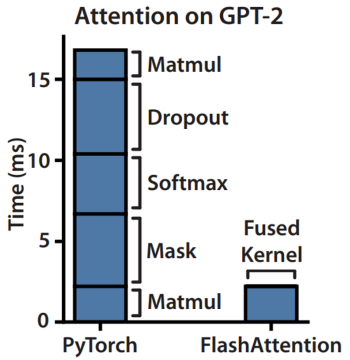
Understand when GPUs get slow



https://www.thonking.ai/p/what-shapes-do-matrix-multiplications

Understand how to make fast algorithms

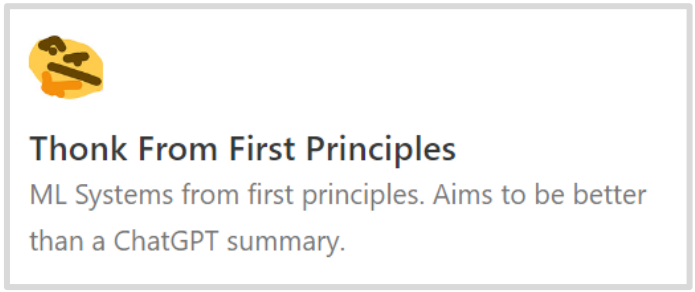




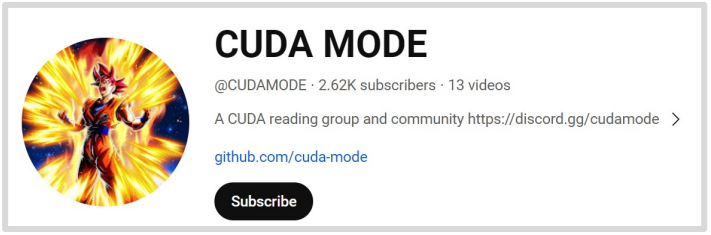
Dao et al, Flash Attention

Before we start..

Substantial credit goes to a few sources that I'd like to highlight..



Horace He's blog



CUDA Mode group

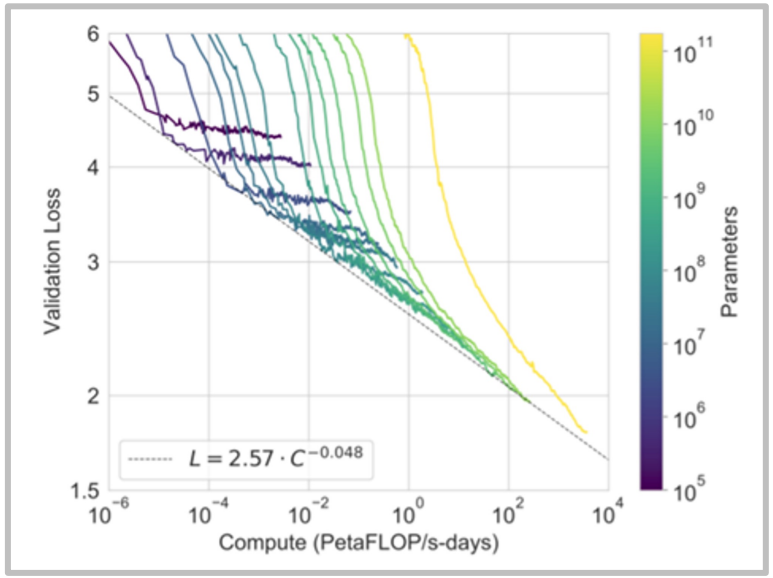
And other sources including https://nichijou.co/, https://jonathan-hui.medium.com/

Organization today:

- **Part 1**: GPUs in depth – how they work and important parts
- **Part 2**: Understanding GPU performance
- **Part 3**: Putting it together – unpacking FlashAttention

Setting the stage: compute leads to predictable perf

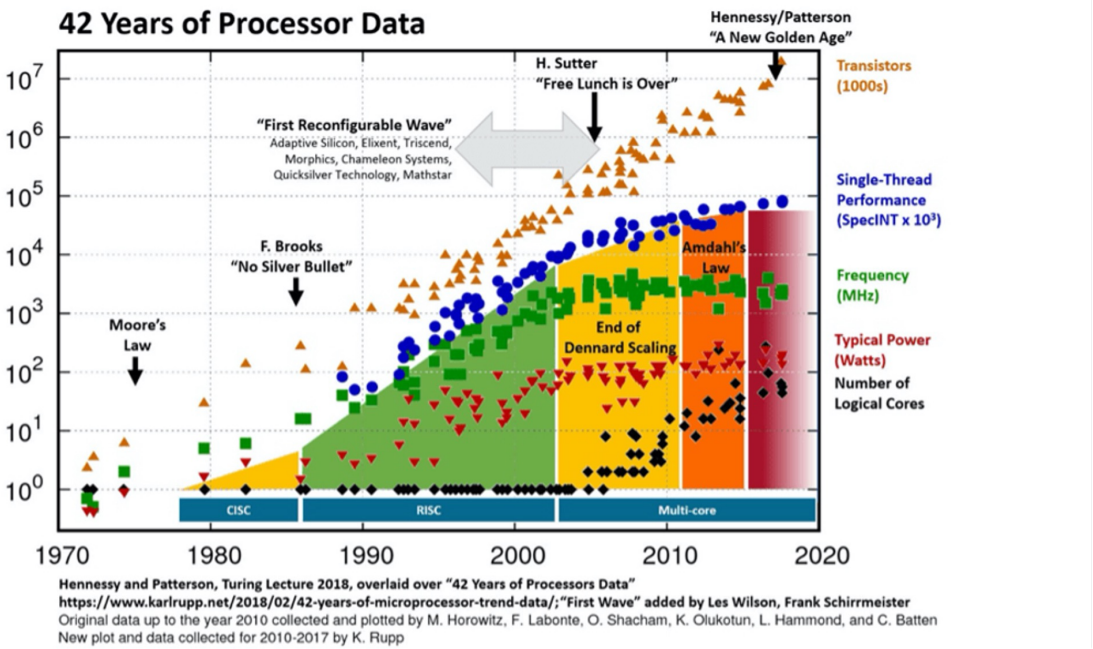
Often times, compute leads to predictable performance gains for language models



Kaplan et al, Neural Scaling Laws

Faster hardware, better utilization, improved parallelization alone can drive progress (for now..)

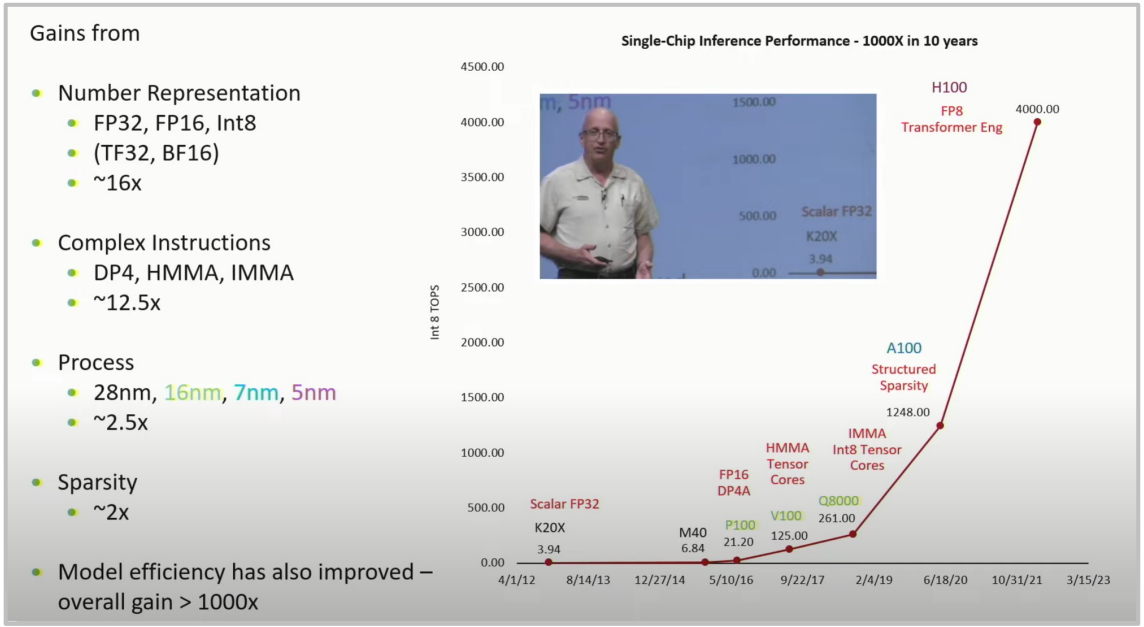
Dennard scaling for CPU performance… is dead.



But the traditional form of scaling (*Dennard scaling*) from 1980-2000s has tapped out.

.. How do we feed LLMs' insatiable appetite for compute?

Parallel scaling continues



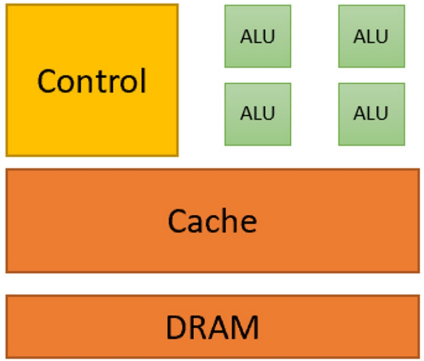
Bill dally, HotChips keynote

Parallel scaling with GPUs has scaled > 1000x in 10 years.

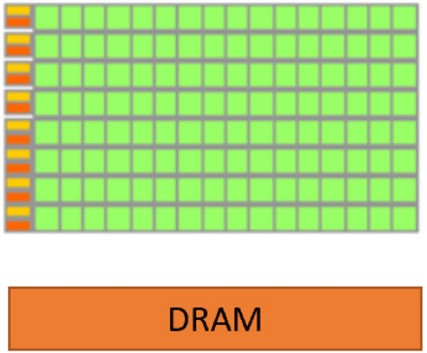
There is no LLM scaling without GPU scaling

How is a GPU different from a CPU?

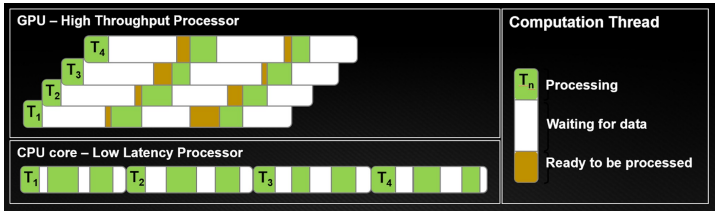
CPUs optimize for a few, fast threads while GPUs optimize for many many threads



CPU



GPU



Many tiny compute units (ALUs).
Much less support for branching (control, cache)

CPUs optimize for latency (each thread finishes quickly)
GPUs optimize for throughput (total processed data)

https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/

Anatomy of a GPU (execution units)

SM

GA100 Full GPU with 128 SMs

Each SM further contains many **SPs (streaming processor)** that can execute ‘threads’ in parallel

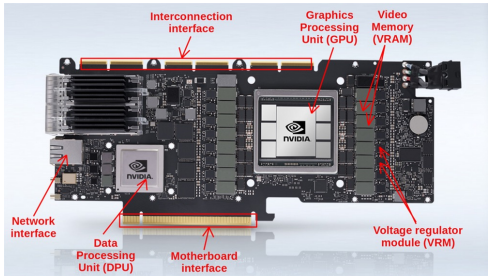
GPUs have *many* **SM (streaming multiprocessors)** that independently execute ‘blocks’ (jobs).

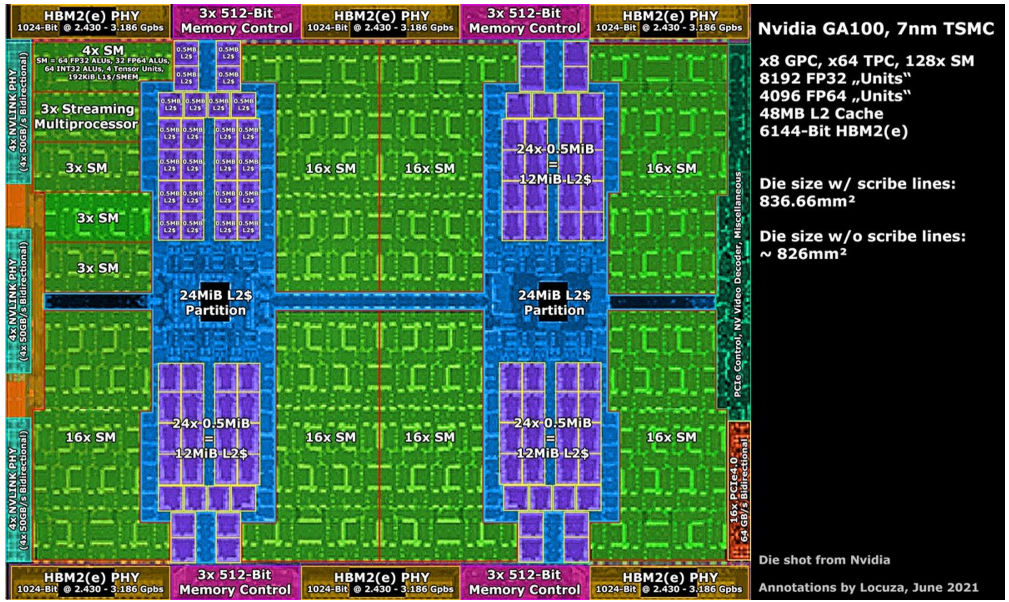
Anatomy of a GPU (memory)

The closer the memory to the SM, the faster it is – L1 and shared memory is *inside* the SM. L2 cache is on die, and global memory are the memory chips next to the GPU

TABLE IV
THE MEMORY ACCESSES LATENCIES

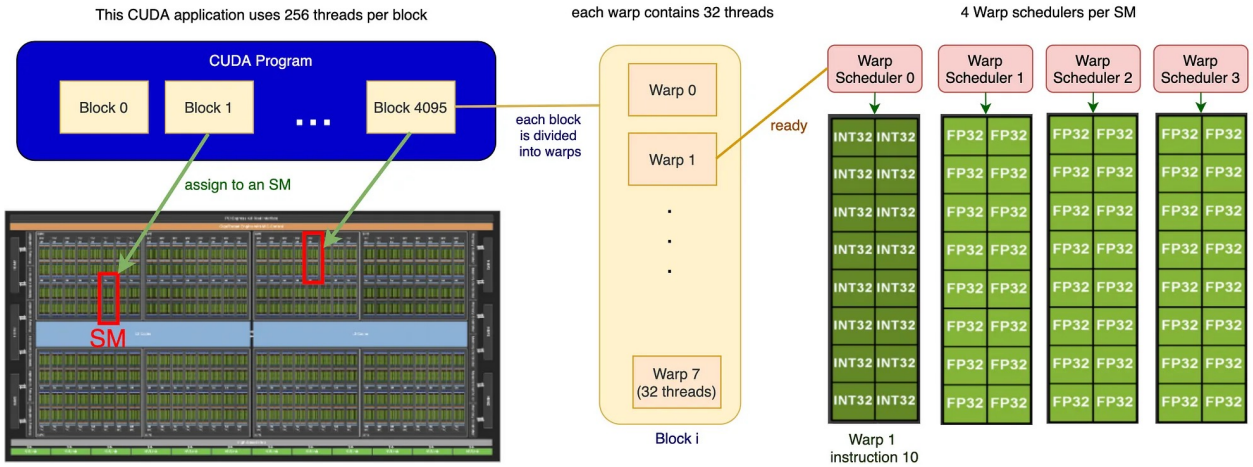
Memory type	*CPI (cycles)*
Global memory	290
L2 cache	200
L1 cache	33
Shared Memory (ld/st)	(23/19)





SRAM (shared/cache memory) is much more expensive (100x) but ~ 8x faster than DRAM (Global memory)

Execution model of a GPU



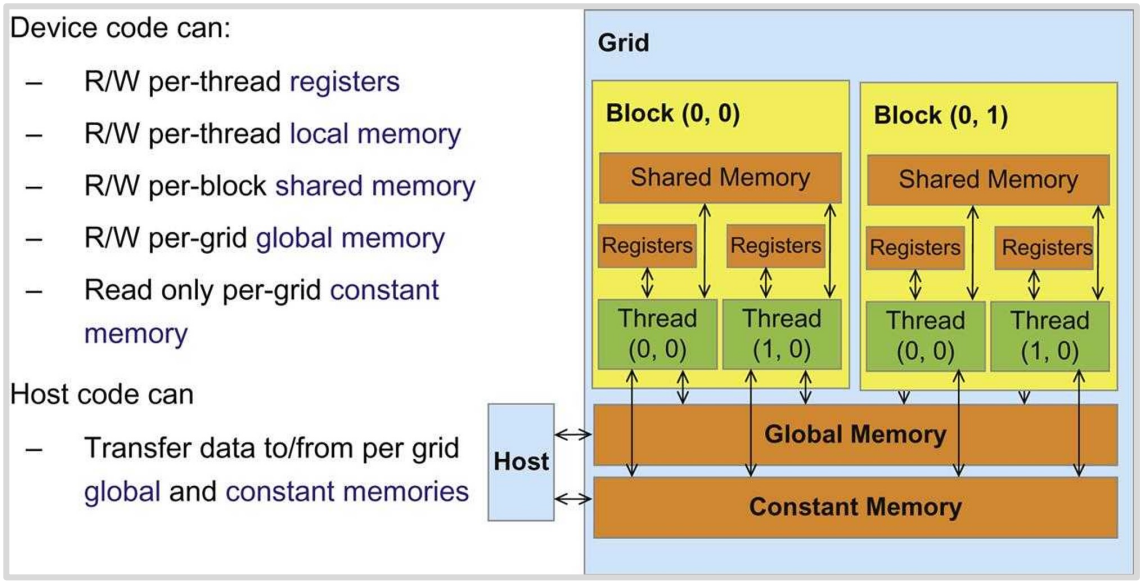
There are 3 important players in the execution model

Threads: Threads 'do the work' in parallel – all threads execute the same instructions but with different inputs (SIMT).

Blocks: Blocks are groups of threads. Each block runs on a SM w/ its own shared memory.

Warp: Threads always execute in a 'warp' of 32 consecutively numbered threads each.

Memory model of a GPU



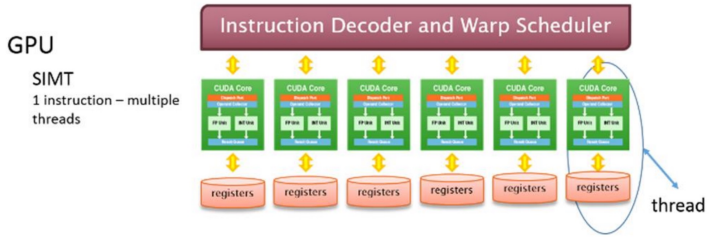
Each thread can access its own register, and shared memory within the block.

Information that goes across blocks need to be read/written to global memory (slow)

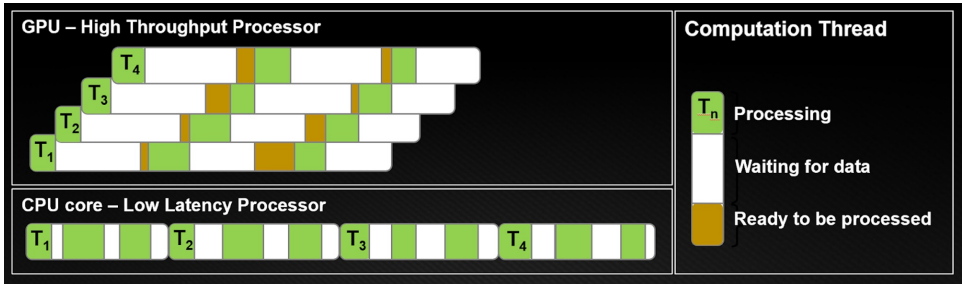
Strengths of the GPU model

- Easily scales up hard workloads (by adding more SMs)

- Easy (?) to program due to the SIMT model



- Threads are ‘lightweight’ and can be stopped and started



GPUs as fast matrix multipliers

Fast Matrix Multiplies using Graphics Hardware

E. Scott Larsen
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA
larsene@cs.unc.edu

David McAllister
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA
davemc@cs.unc.edu

Implementation

We mention here some observations we made during our implementation that may be of interest to those duplicating our results.

Refresh Rate We found that setting the refresh rate on the monitor as low as possible made marginal improvements (about 10%).

RGBA We found that 4 numbers can be packed into a single pixel, by setting the red, green, blue, and alpha channels to different values.

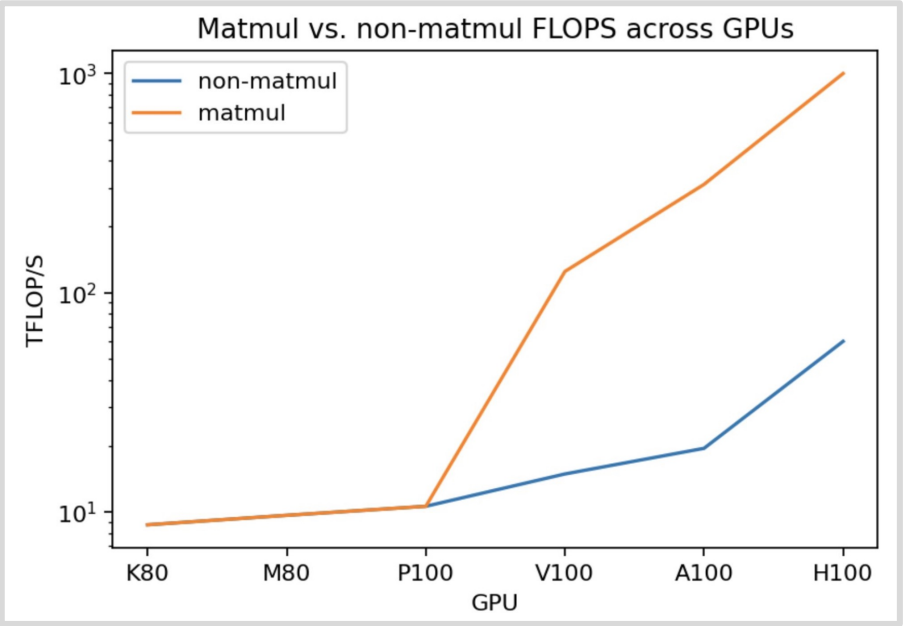
Texture Format Changing the format of the texture creation and read-back from RGBA to ABGR_EXT (in OpenGL) gave about 40% improvement on our hardware. This is because the hardware driver avoids reformatting the data from the application format to the card format. There is a number of options here, with near equal performance for each option except the one used natively on the specific hardware. The native format should give significant improvement.

Full Screen Running full screen instead of in a window provides improved performance.

Various other optimizations yielded minor (<1%) improvements.

Early days of NVIDIA GPUs – programmable shaders. Researchers hacked this to do matmuls

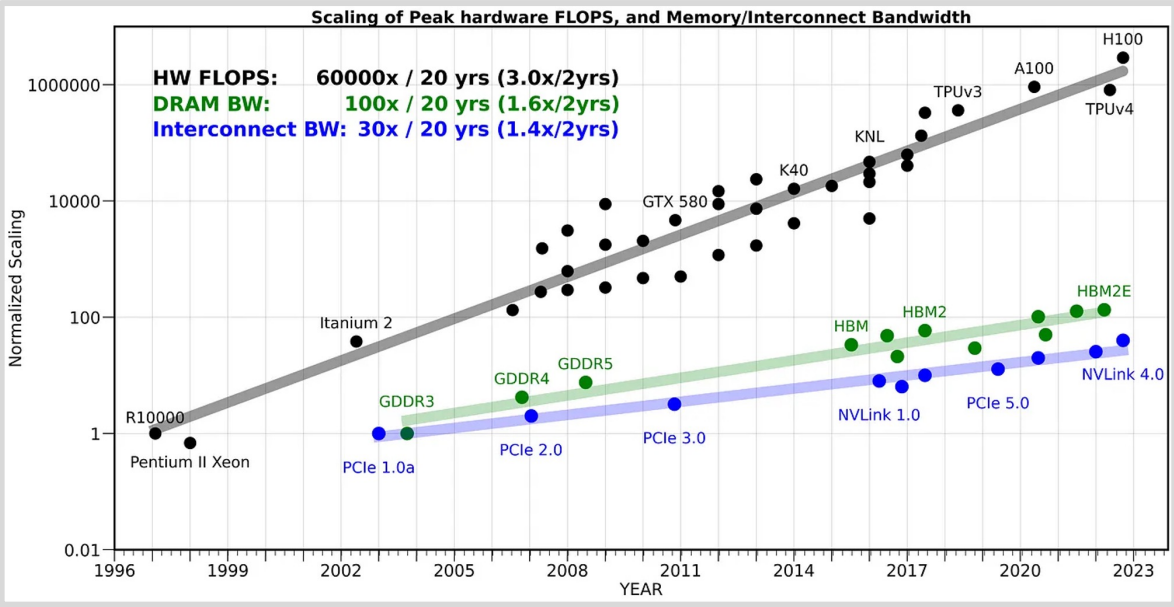
New matmul hardware means matmuls are fast and special



Tensor cores (introduced in V, T series) are specialized matrix multiplication circuits.

Matmuls are >10x faster than other floating point ops!

Compute scaling is faster than memory scaling



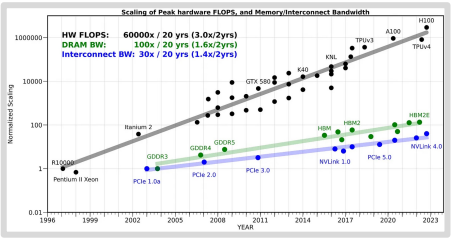
https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

FLOPs scale faster than memory – it's hard to keep our compute units fed with data!

Recap: GPUs – what are they and how do they work

- GPUs are massively parallel – same instructions applied across many workers

- Compute (and esp matmuls) have scaled faster than memory

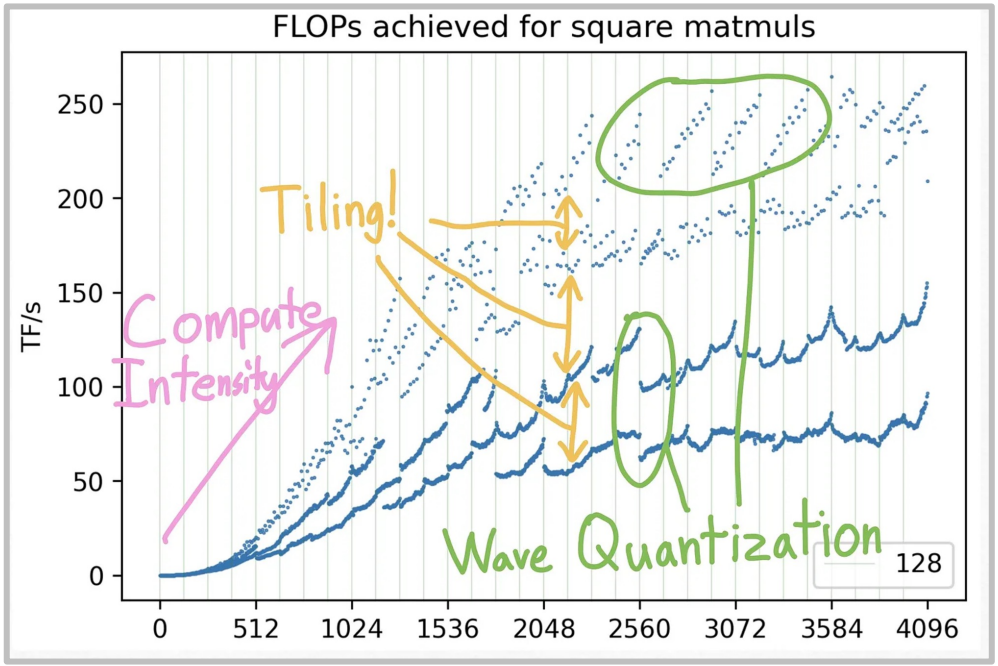


- We have to respect the memory hierarchy to make things go fast.

TABLE IV
THE MEMORY ACCESSES LATENCIES

Memory type	*CPI (cycles)*
Global memory	290
L2 cache	200
L1 cache	33
Shared Memory (ld/st)	(23/19)

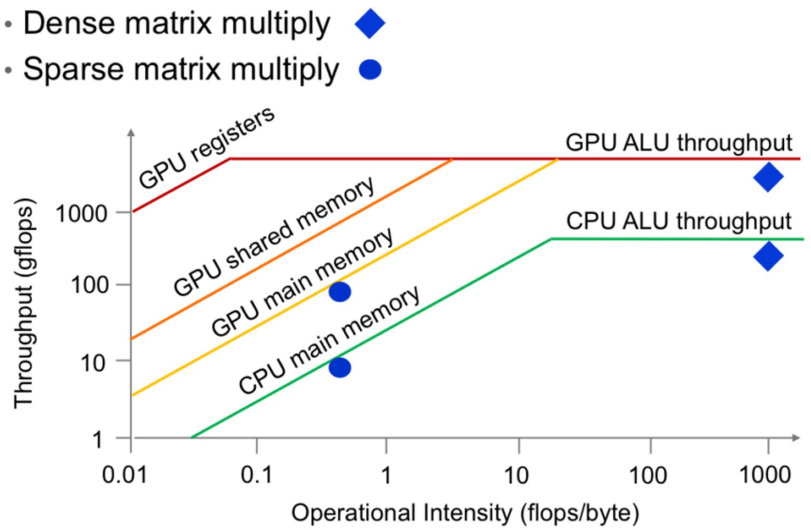
Part 2: Making ML workloads fast on a GPU



Performance on a GPU can be complex, even for something as simple as a square matmul

What makes ML workloads fast?

The roofline model



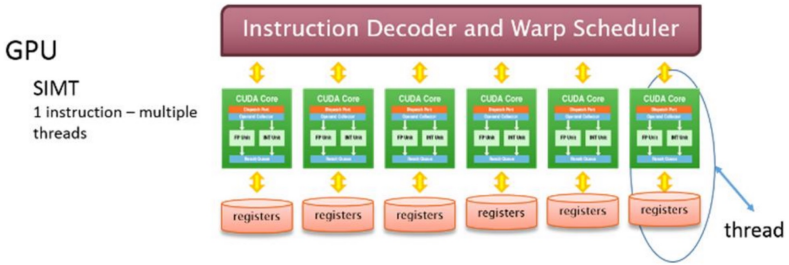
Key to this section: **how do we avoid being memory bound?**

How do we make GPUs go fast?

1. **Control divergence (not a memory bottleneck..)**
2. **Low precision computation**
3. **Operator fusion**
4. **Recomputation**
5. **Coalescing memory**
6. **Tiling**

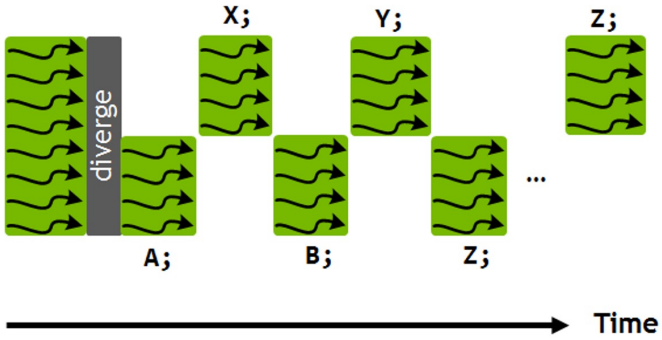
Control divergence (not a memory issue)

GPUs operate in a SIMT model – every thread in a warp is executing the same instruction

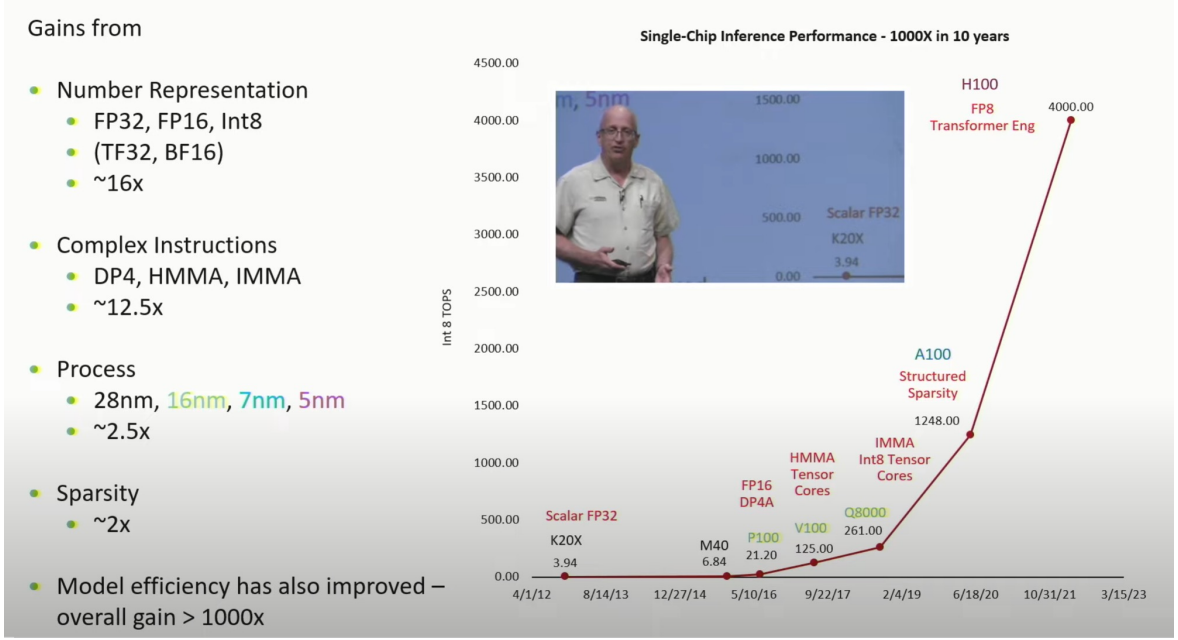


Conditionals are fine, but lead to significant overhead from the execution model

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Trick 1: Low precision computation



If you have fewer bits, you have fewer bits to move

Low precision improves arithmetic intensity

Example: elementwise ReLU ($x = \max(0, x)$) on a vector of size n**.**

(Float 32 case)

Memory access: 1 read (x), 1 write (if x < 0), float 32 = 8 bytes

Operations: 1 comparison op, 1 FLOP.

Intensity: 8 bytes / FLOP

(Float 16)

Memory access: 1 read (x), 1 write (if x < 0), float 16 = 4 bytes

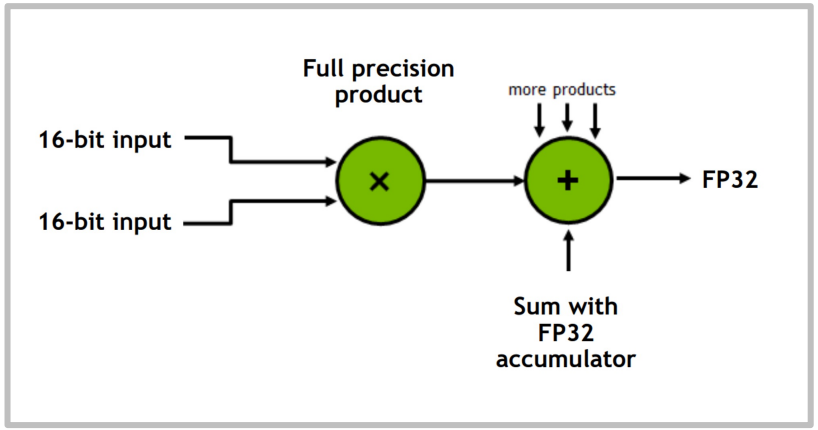
Operations: 1 comparison op, 1 FLOP.

Intensity: 4 bytes / FLOP

Low precision drives faster matrix multiplies

Lots of operations in modern GPUs are accelerated via low / mixed precision operations

Tensor cores



Operations that can use 16-bit storage (FP16/BF16)

- Matrix multiplications
- Most pointwise operations (e.g. relu, tanh, add, sub, mul)

Operations that need more precision (FP32/FP16)

- Adding small values to large sums can lead to rounding errors
- Reduction operations (e.g. sum, softmax, normalization)

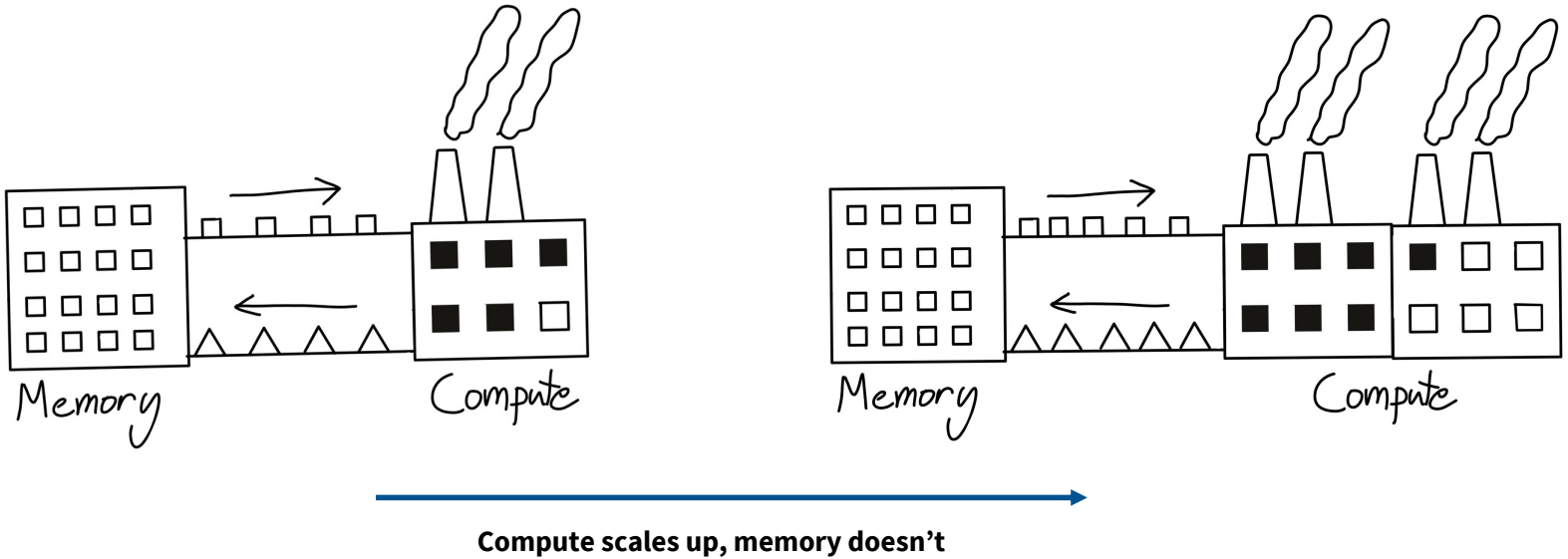
Operations that need more range (FP32/BF16)

- Pointwise operations where $|f(x)| \gg |x|$ (e.g. exp, log, pow)
- Loss functions

https://nvlabs.github.io/eccv2020-mixed-precision-tutorial/files/dusan_stosic-training-neural-networks-with-tensor-cores.pdf

Trick 2: Operator fusion

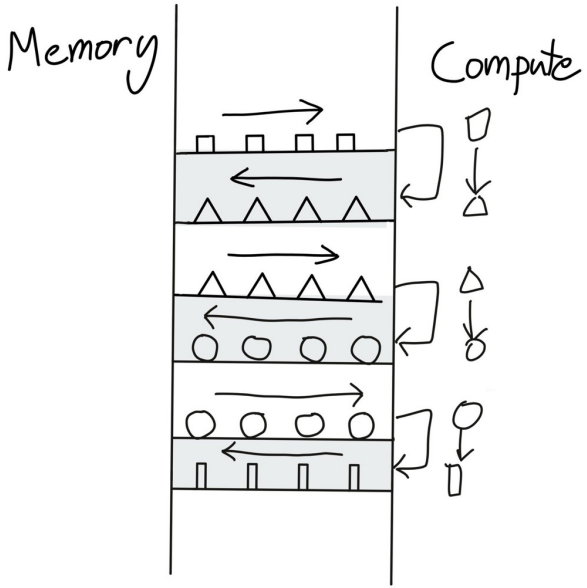
Think of a GPU like a factory – inputs come from a warehouse (memory) and is processed at a factory



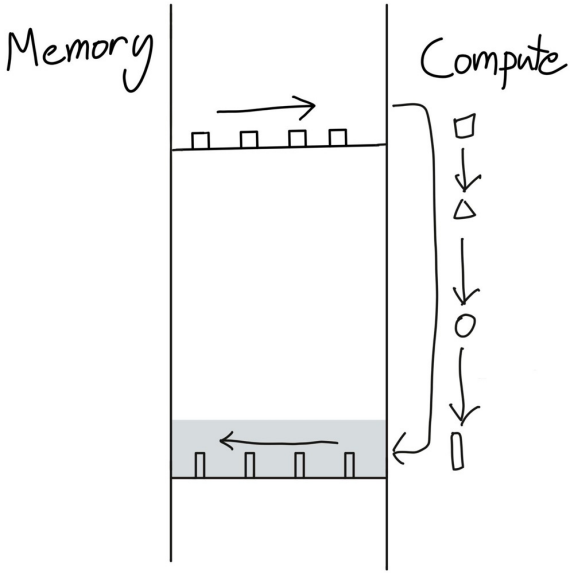
https://horace.io/brrr_intro.html

Operator fusion to minimize memory access

What if we have to do many operations? Shipping back and forth is somewhat silly



Naïve (non-fused)

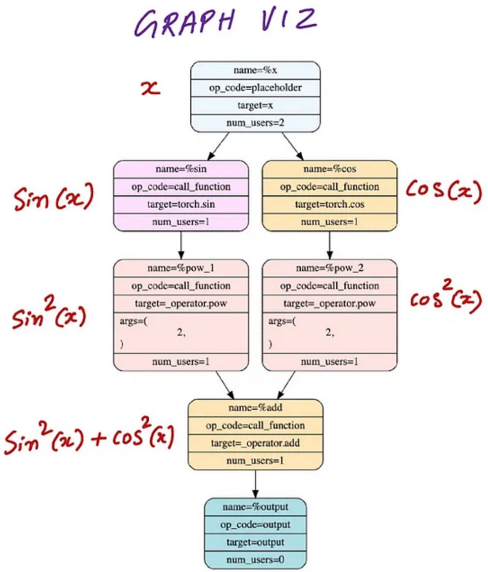


Fused kernel

Example – sines and cosines

FX GRAPH IR

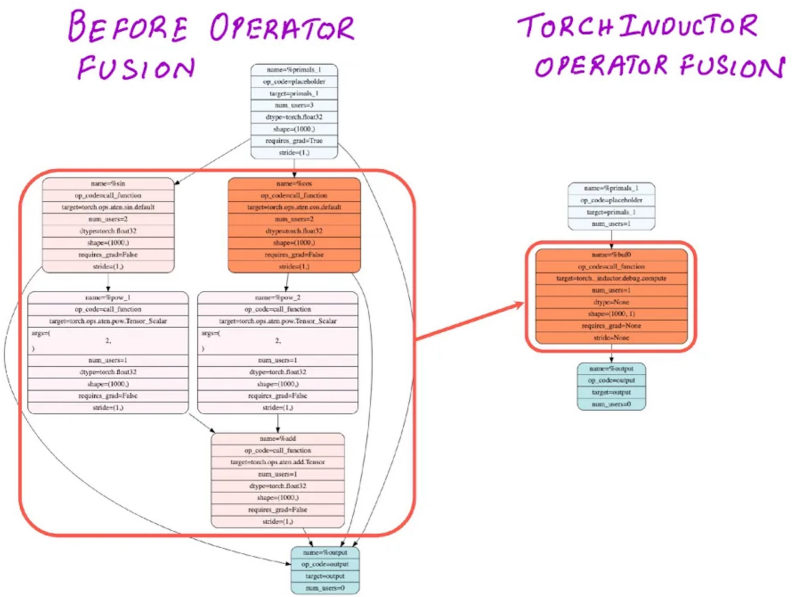
```
class GraphModule(torch.nn.Module):
    def forward(self, x : torch.Tensor):
        # File: /tmp/ipykernel_2583/1502985755.py:2, code:
        sin = torch.sin(x)
        pow_1 = sin ** 2;  sin = None
        cos = torch.cos(x);  x = None
        pow_2 = cos ** 2;  cos = None
        add = pow_1 + pow_2;  pow_1 = pow_2 = None
        return (add,)
```



Computing $\sin^2 x + \cos^2 x$ naively launches 5 CUDA kernels (back and forth)

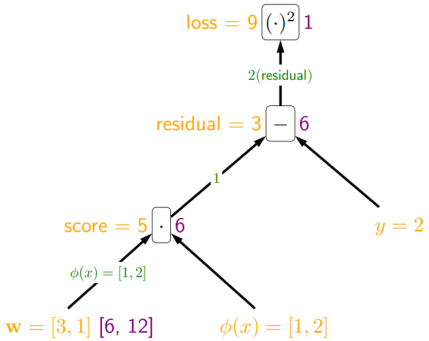
https://towardsdatascience.com/how-pytorch-2-0-accelerates-deep-learning-with-operator-fusion-and-cpu-gpu-code-generation-35132a85bd26

Fusion example



All 5 pointwise operations can be fused into a single CUDA kernel call.
‘Easy’ fusions like this can be done automatically by compilers (torch.compile)

Trick 3: recomputation



$$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

$\mathbf{w} = [3, 1], \phi(x) = [1, 2], y = 2$

backpropagation

$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = [6, 12]$

Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{loss}}{\partial f_i}$ is how f_i influences loss

Algorithm: backpropagation algorithm

Forward pass: compute each f_i (from leaves to root)

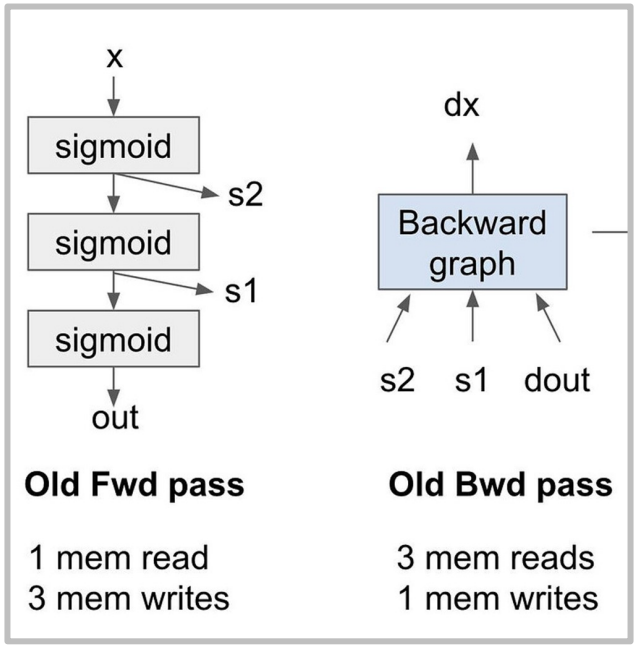
Backward pass: compute each g_i (from root to leaves)

[From cs221]

In backpropagation, we store the activations (yellow) and compute Jacobians (green)

Storing (and retrieving) activations can be expensive!

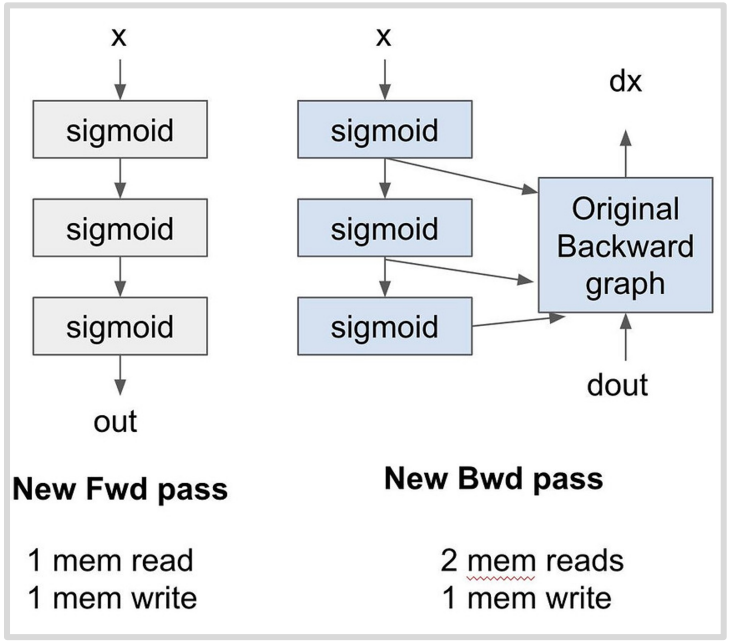
Let's say we stack 3 sigmoids on top of each other.



This is really terrible for perf – 8 mem read/writes, very low arithmetic intensity.

https://dev-discuss.pytorch.org/t/min-cut-optimal-recomputation-i-e-activation-checkpointing-with-aotautograd/467

Throw away the activations, re-compute them!

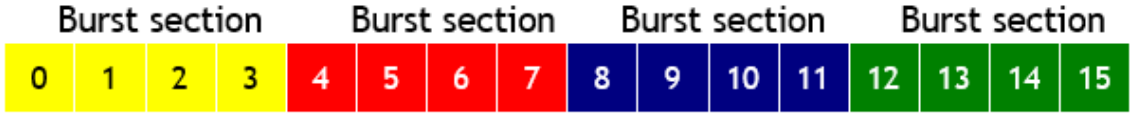


Throwing away computation can actually be optimal, w/ 5/8th the memory accesses!

https://dev-discuss.pytorch.org/t/min-cut-optimal-recomputation-i-e-activation-checkpointing-with-aotautograd/467

Trick (?) 4: Memory coalescing and DRAM

DRAM (global memory) is read in ‘burst mode’ – each read gives you many bytes!

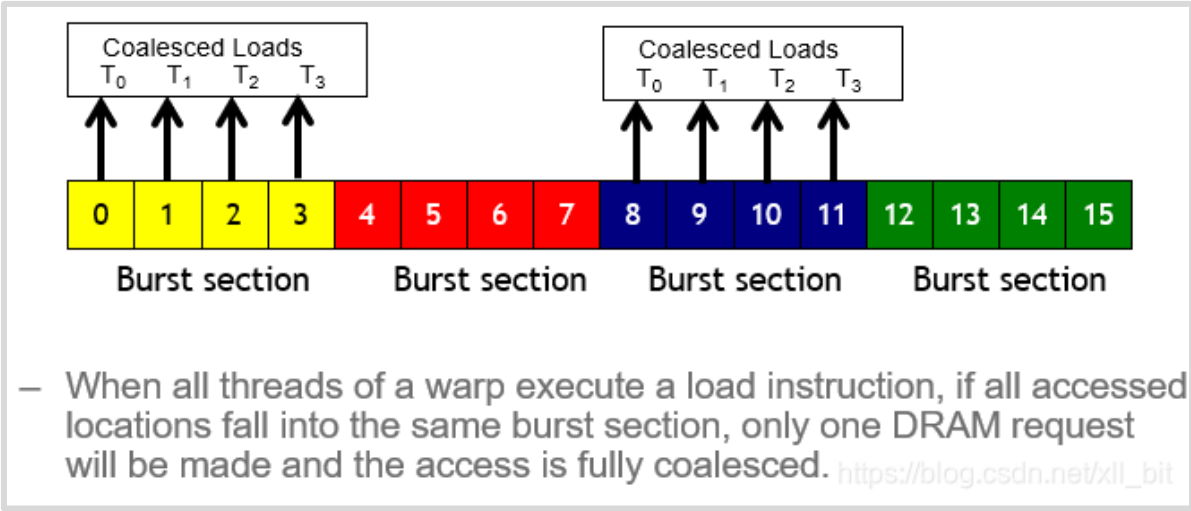


- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

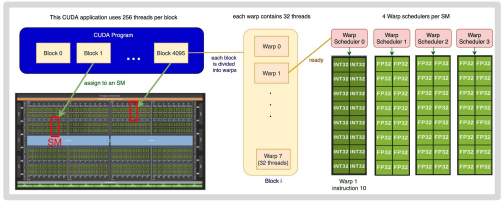
[https://blog.csdn.net/xll_bit/article/details/117702476]

Memory coalescing

Memory accesses are *coalesced* if all the threads (in a warp) fall within the same burst

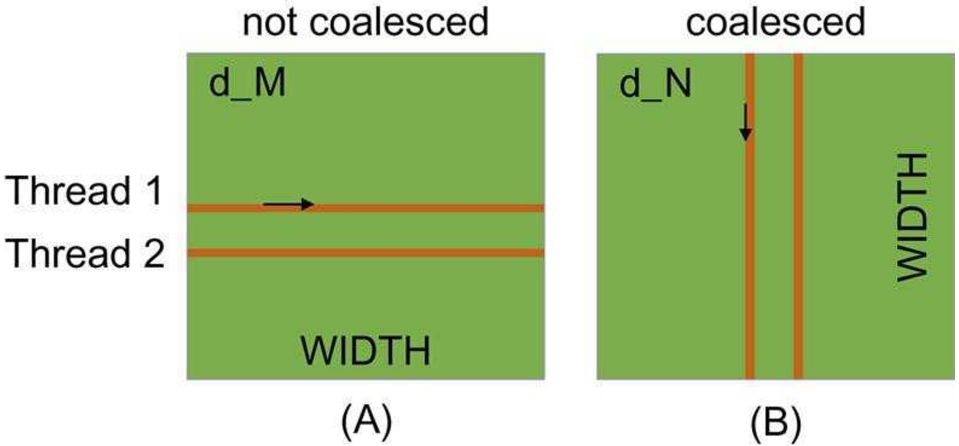


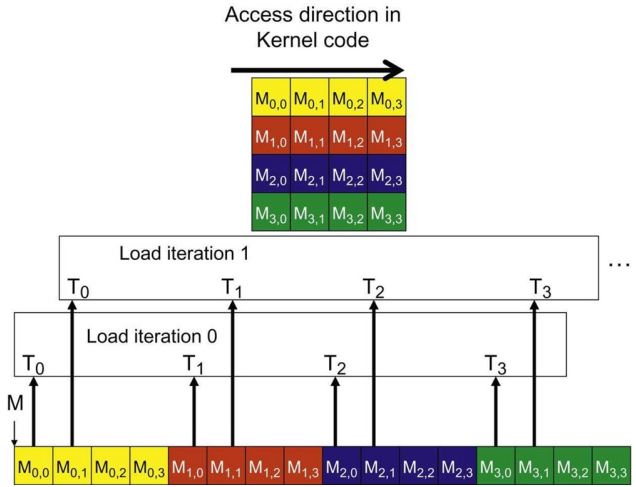
– When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.



Reminder: a warp is a set of 32 consecutively numbered threads that execute together in a block. Memory accesses happen together

Coalescing for matrix multiplication





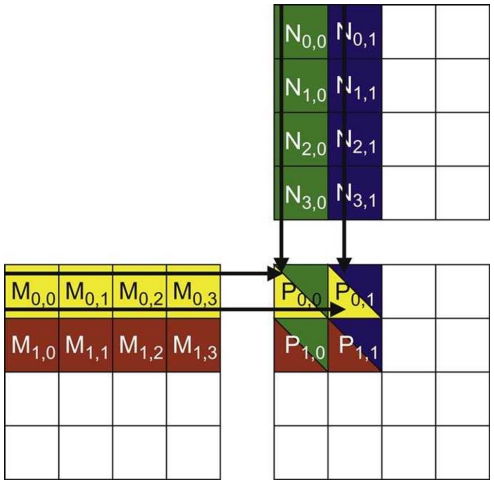
For row-major matrices – **threads that move along rows are not coalesced**

Note how the second diagram reads the entire vector at each step!

Trick 5 (the big one): tiling

Tiling is the idea of grouping and ordering threads to minimize global memory access.

Let’s go back to matrix multiplication..



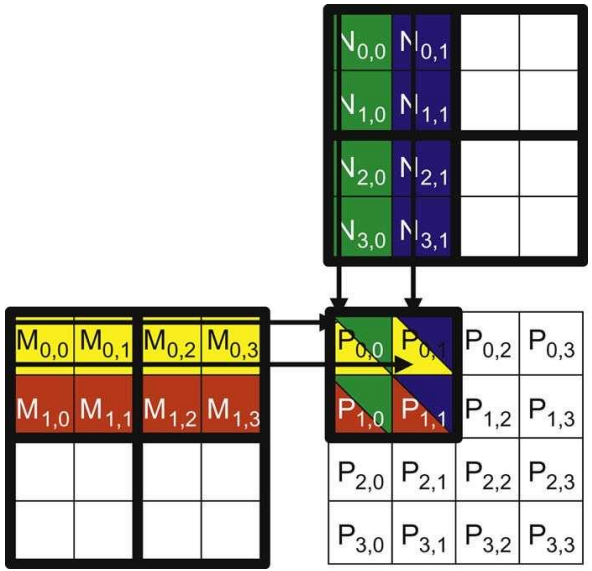
Access order

$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

Note that memory access is not coalesced, and repeated (M0,0 and N1,0)

Tiling – store and reuse information in shared memory

Cut up the matrix into smaller ‘tiles’, and load this into shared memory



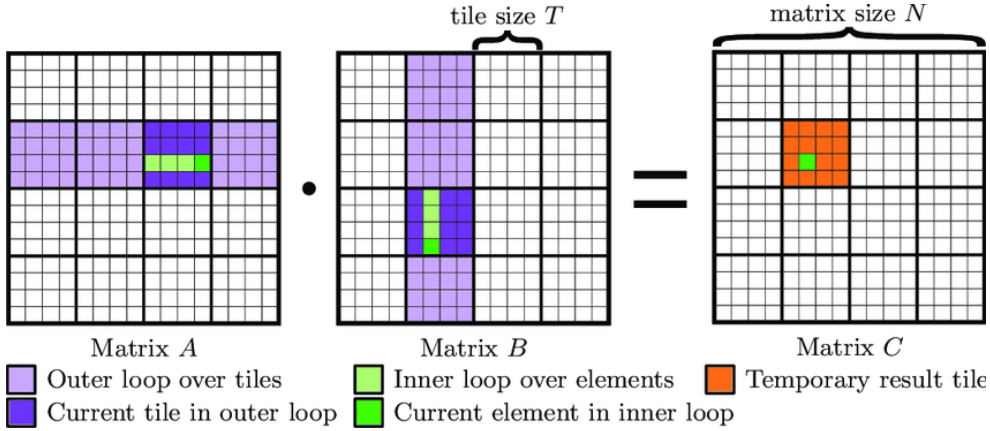
Compute the matrix multiply in ‘phases’

1. Load $M_{0,0}$ and $N_{0,0}$ tiles into SHM
2. Compute partial sums for P

   (Done with one tile)
3. Load the $M_{0,0}$ and $N_{2,0}$ tile into SHM
4. …

Advantages: repeated reads now access shared, not global memory and memory access can be coalesced

Tiling math



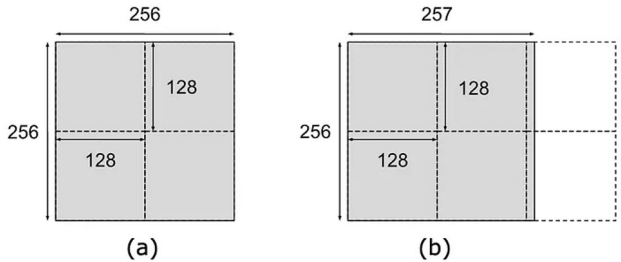
Non-tiled matrix multiply: each input is read N times from global memory

Tiled matrix multiply: each input is read $\frac{N}{T}$ times from global memory, and T times within each tile. This is a factor of T reduction in global memory access

Complexities with tiling

Tile sizes may not divide the matrix size and lead to low utilization

Figure 6. Example of tiling with 128x128 thread block tiles. (a) Best case - matrix dimensions are divisible by tile dimensions (b) Worse case - tile quantization results in six thread blocks being launched, two of which waste most of their work.



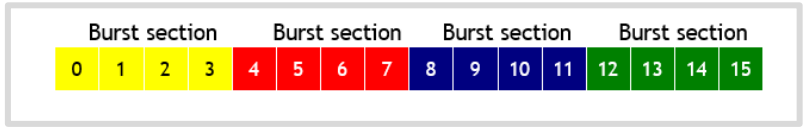
Factors affecting tile sizes

- Coalesced memory access
- Shared memory size
- Divisibility of the matrix dim

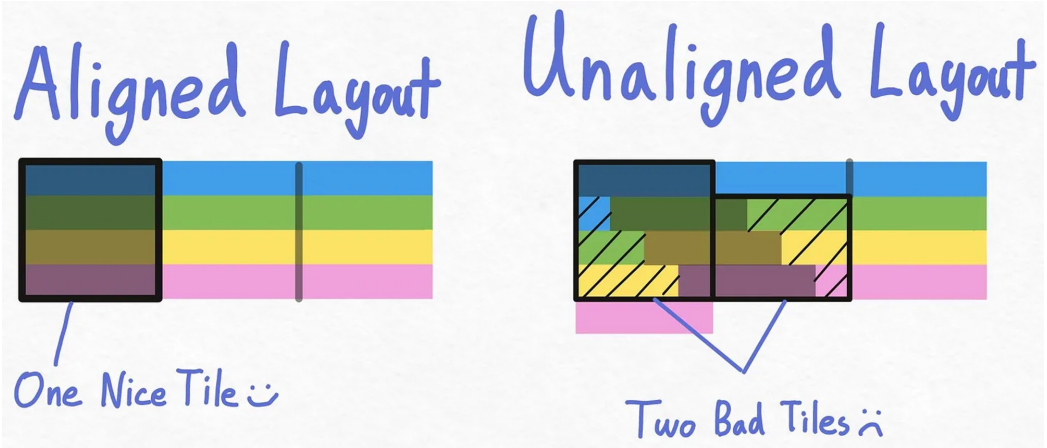
https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#tile-quant

Complexities with tiling 2 – memory alignment

Memory comes in bursts



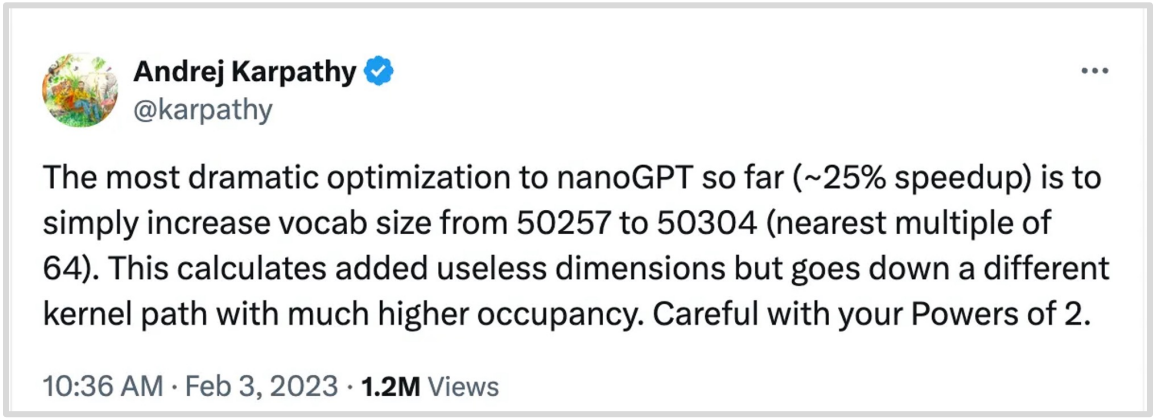
Loading tiles are fast if bursts align with the matrix



https://www.thonking.ai/p/what-shapes-do-matrix-multiplications

Coalesced accesses may be impossible depending on the dimension of the matrix..
(have to do padding)

Putting it together: understanding a matrix mystery



Andrej Karpathy ✓
@karpathy

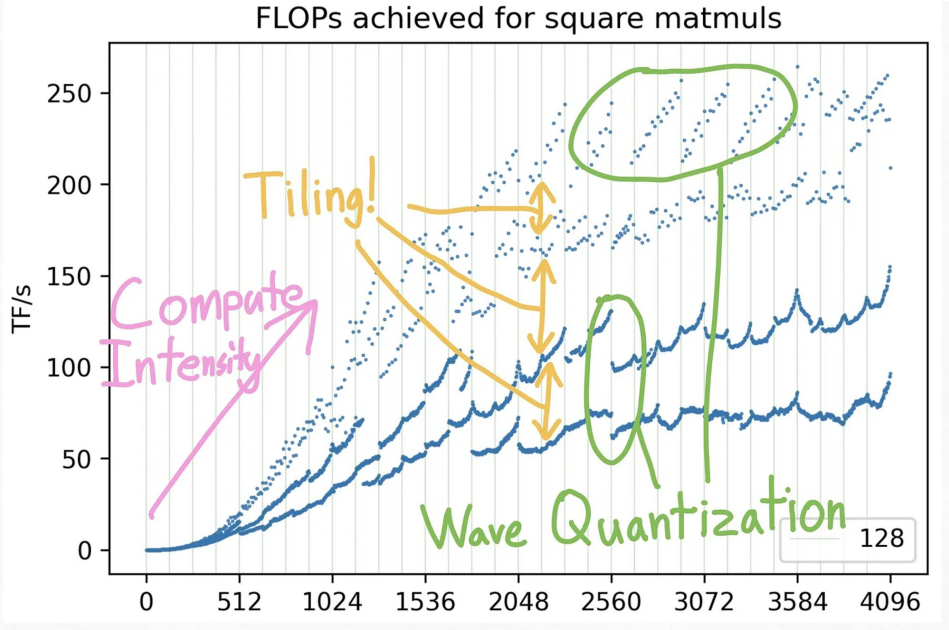
The most dramatic optimization to nanoGPT so far (~25% speedup) is to simply increase vocab size from 50257 to 50304 (nearest multiple of 64). This calculates added useless dimensions but goes down a different kernel path with much higher occupancy. Careful with your Powers of 2.

10:36 AM · Feb 3, 2023 · **1.2M** Views

Why is it *faster* to have bigger matrices?

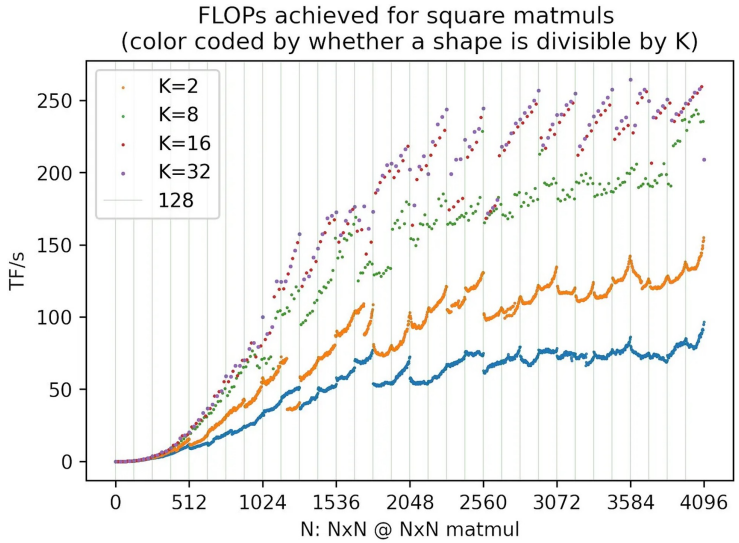
This section is from https://www.thonking.ai/p/what-shapes-do-matrix-multiplications

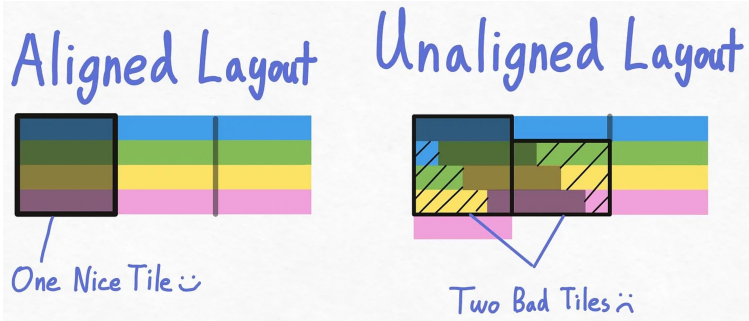
Matrix mystery



We understand some of this (compute intensity, tiling). Let’s take a closer look..

Part 1: tiling

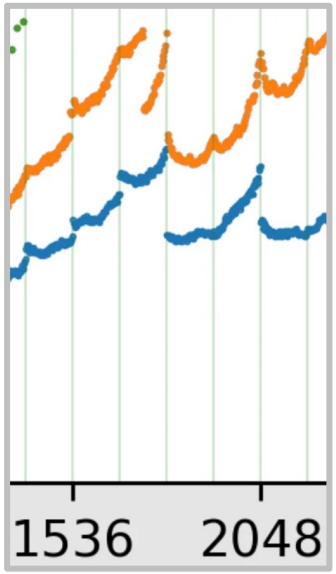




Tiling has a major impact through alignment.

Part 2: wave quantization

What's with the periodic behavior?



This happens at 1792 to 1793 size.

Why? Using a tile size of 256×128, there are

$$\frac{1792}{246}\times\frac{1792}{128}=7\times14=98$$

tiles. If we increase this to 1793, we have

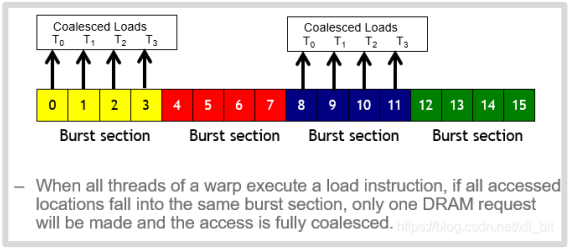
$$8\times15=120$$

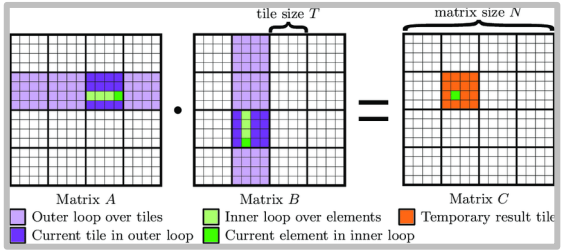
tiles.

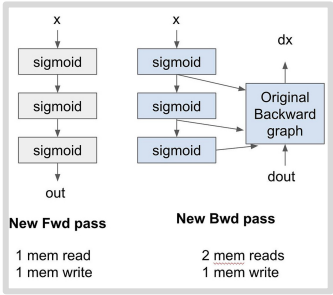
An A100 has 108 SMs, so it cannot execute all 120

Recap of part 2: making ML workloads go fast

- Reduce memory accesses
  - Coalescing
  - Fusion
- Move memory to shared memory
  - Tiling
- Trade memory for compute/accuracy
  - Quantization
  - Recomputation

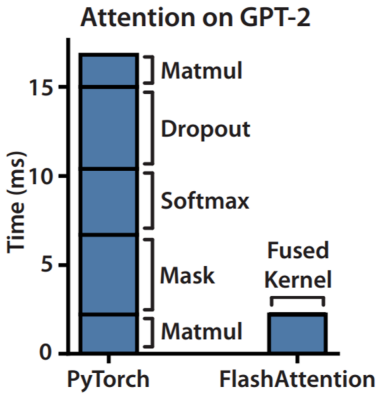




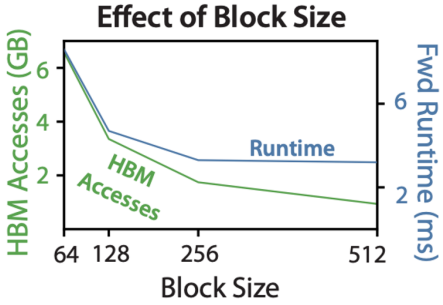


Part 3: Using what we know to understand Flash Attention

Flash attention [Dao et al] dramatically accelerates attention.. But how?



Attention	Standard	FLASHATTENTION
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

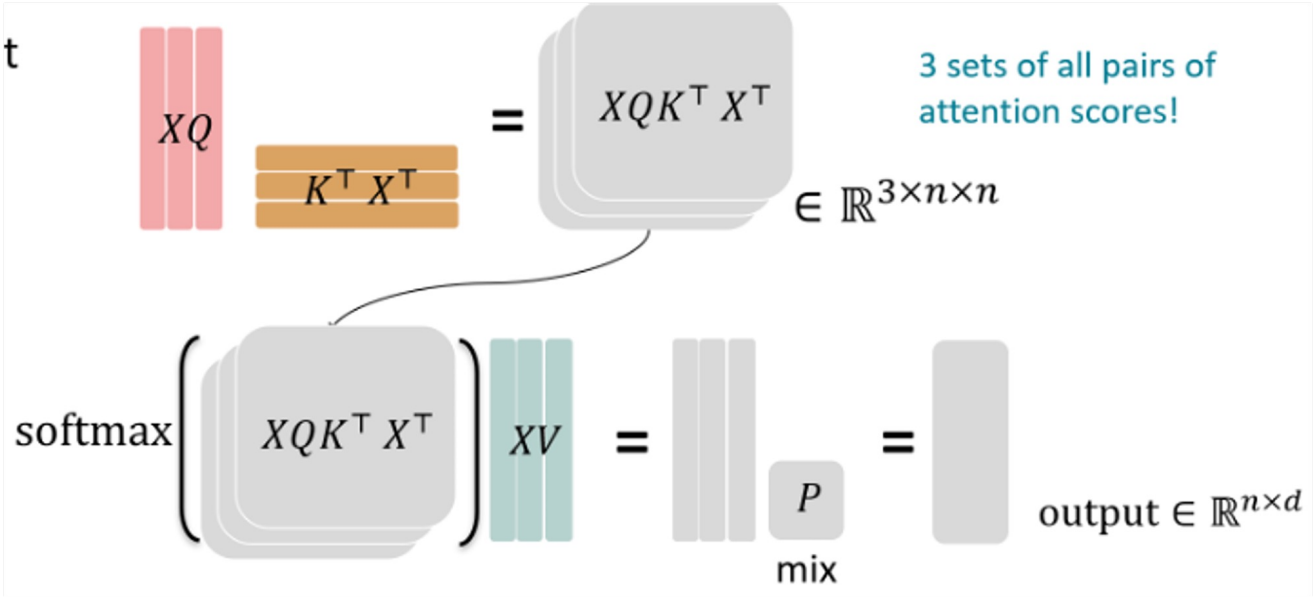


Technique from paper:

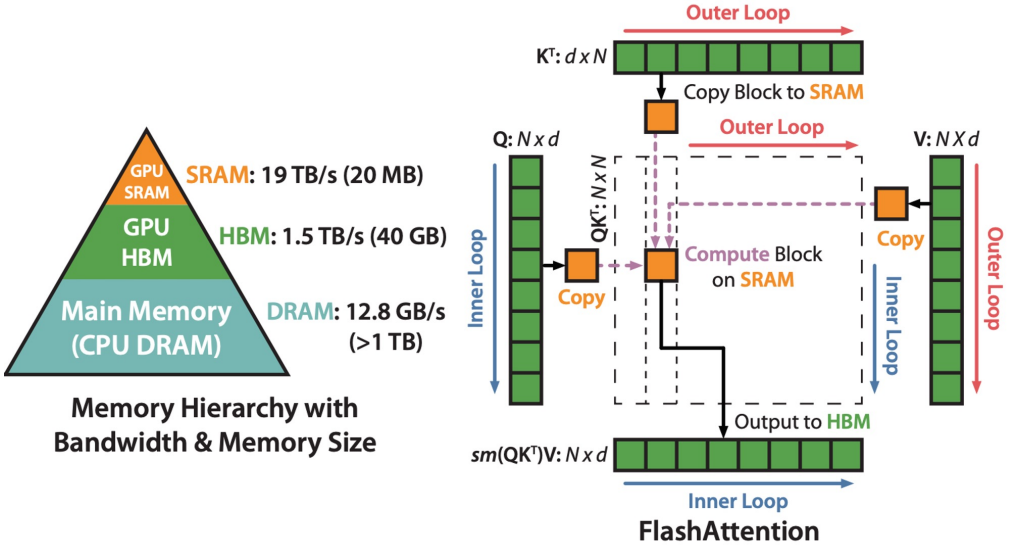
We apply two established techniques (tiling, recomputation) to overcome the technical challenge of computing exact attention in sub-quadratic HBM accesses. We describe this in Algorithm 1. The main idea

Recap of attention computation

Attention computation: 3 matrix multiplies (K, Q, V) with a softmax in between



Tiling part 1: tiling for the KQV matrix multiply



This figure 1 from the paper is literally just tiling for a KQV matrix multiply..

But what do we do about the softmax?

Tiling part 2: incremental computation of the softmax

From Mikailov and Gimelshein 2018,

Normal softmax

$$y_i = \frac{e^{x_i - \max_{k=1}^{V} x_k}}{\sum_{j=1}^{V} e^{x_j - \max_{k=1}^{V} x_k}} \tag{2}$$

All major DL frameworks are using this safe version for the Softmax computation: TensorFlow

Algorithm 2 Safe softmax

$m_0 \leftarrow -\infty$
for $k \leftarrow 1, V$ **do**
    $m_k \leftarrow \max(m_{k-1}, x_k)$
end for
$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
    $d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
end for
for $i \leftarrow 1, V$ **do**
    $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
end for

Online softmax

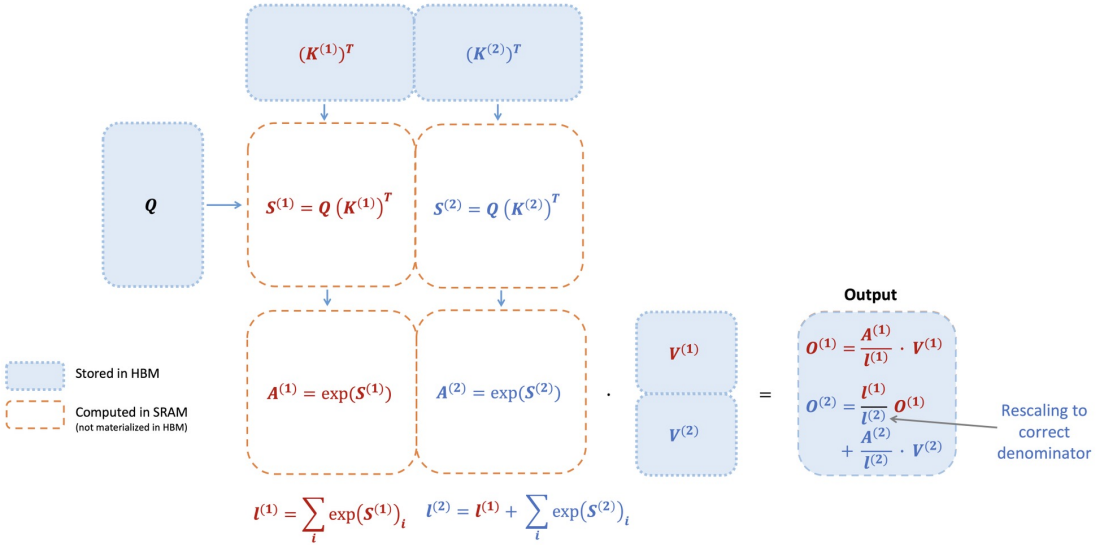
Algorithm 3 Safe softmax with online normalizer calculation

$m_0 \leftarrow -\infty$
$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
    $m_j \leftarrow \max(m_{j-1}, x_j)$
    $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
end for
for $i \leftarrow 1, V$ **do**
    $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
end for

To keep track of the max, incrementally update the max, and set up a telescoping sum

This lets you compute the softmax tile-by-tile

Putting it all together – the forward pass of flash attention



From Dao 2023, we see

- Tile-wise computation of the inner products, (S)
- Fusion of the exponential operator
- Tile-wise computation of the softmax via the online, telescoping sum trick

(We won't cover the backward pass – but they recompute tile-by-tile..)

Recap for the whole lecture

- Hardware powers scale, and low-level details determine what scales or doesnt
- Curent GPU based compute strongly encourages thinking about matmul + data movement
- Thinking carefully about the GPU (coalescing, tiling, fusion) leads us to good performance