# CUDA Performance Checklist

Mark Saroufim

## Follow along

Locally or remotely https://lightning.ai/

git clone https://github.com/cuda-mode/lectures

cd lecture8

nvcc -o benchmark *.cu

ncu benchmark

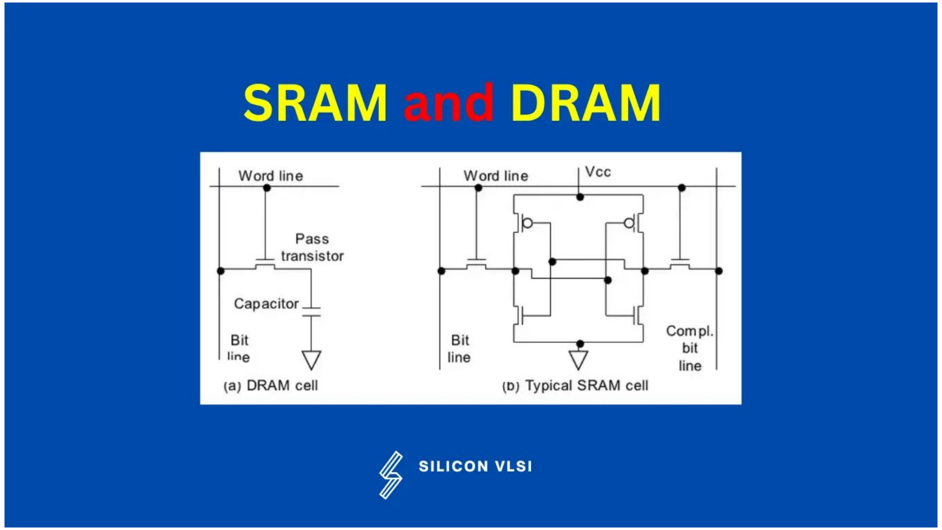
Approach follows style of https://arxiv.org/pdf/1804.06826.pdf

# DRAM vs SRAM

DRAM: 1 transistor, 1 capacitor

SRAM: 6 transistors

So SRAM is faster but more expensive, takes up more space and gets hotter



Listen to Bill Dally talks on YouTube: https://www.youtube.com/watch?v=kLiwvnr4L80

https://siliconvlsi.com/why-sram-is-faster-than-dram/

## Performance checklist

Coalesced Global Memory Access

Maximize occupancy

*Understand if memory or compute bound*

Minimize control divergence

Tiling of reused data

Privatization

Thread Coarsening

*Rewrite your algorithm using better math*

# Memory latencies

https://arxiv.org/abs/2208.11174

Microbenchmarks using PTX

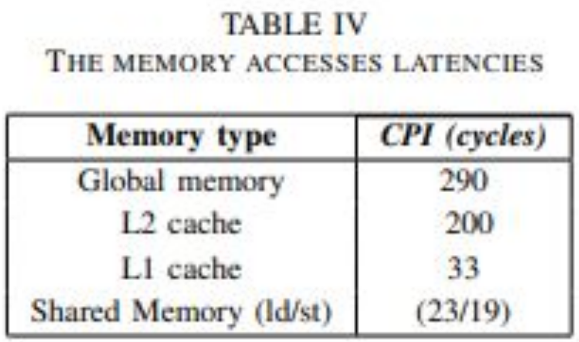
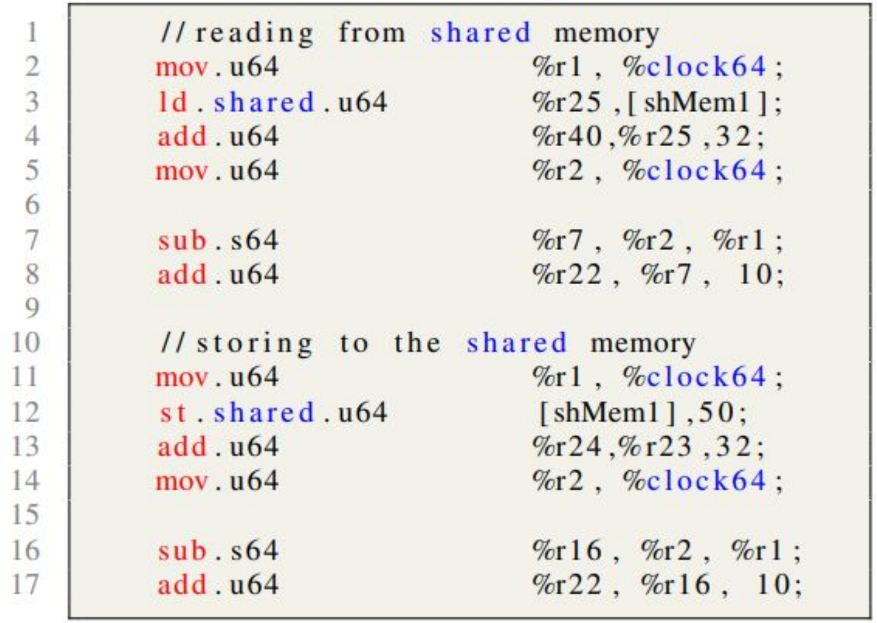


TABLE IV
THE MEMORY ACCESSES LATENCIES

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



```
//reading from shared memory
mov.u64              %r1, %clock64;
ld.shared.u64        %r25,[shMem1];
add.u64              %r40,%r25,32;
mov.u64              %r2, %clock64;

sub.s64              %r7, %r2, %r1;
add.u64              %r22, %r7, 10;

//storing to the shared memory
mov.u64              %r1, %clock64;
st.shared.u64        [shMem1],50;
add.u64              %r24,%r23,32;
mov.u64              %r2, %clock64;

sub.s64              %r16, %r2, %r1;
add.u64              %r22, %r16, 10;
```

Fig. 3. Computing device shared memory access latency.

## It’s the latency stupid

Throughput is easy, latency is not: “You can get 80 phone lines in parallel, and send one single bit over each phone line, but that 100ms latency is still there.”

Quantization: “For example, to reduce packet size, wherever possible Bolo uses bytes instead of 16-bit or 32-bit words.”

http://www.stuartcheshire.org/rants/latency.html

## Memory Coalescing

You can’t reduce latency but you can hide it by reading contiguous elements of memory

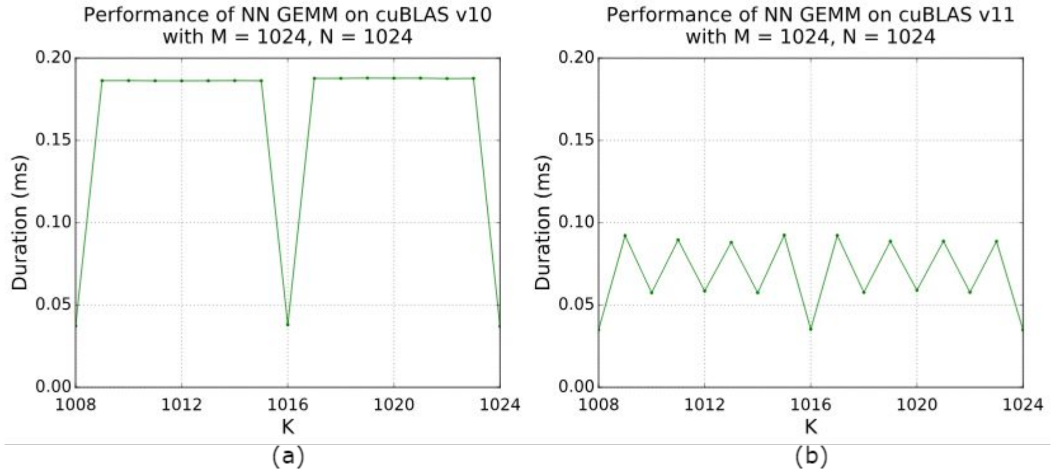
Case study - pay attention to

- DRAM Throughput
- Duration
- L1 cache throughput

## Occupancy

Tile quantization: matrix dimensions are not divisible by the thread block tile size.

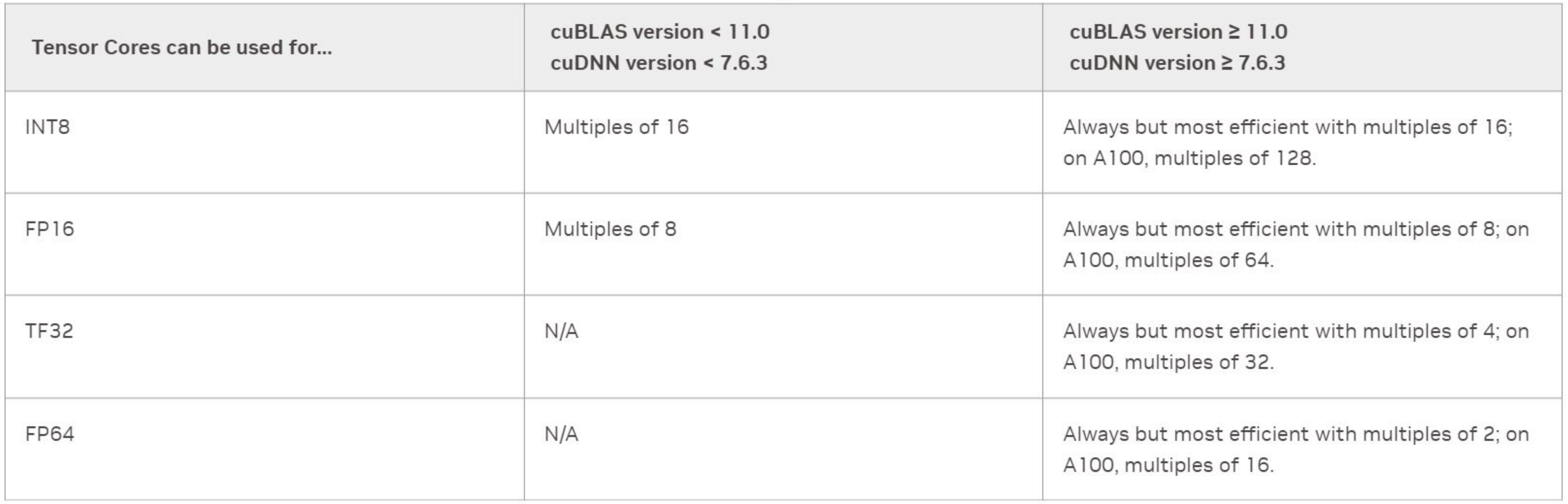
Wave quantization: total number of tiles is not divisible by number of SM on GPU



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

## Solution in PyTorch Land: Padding

**Table 1. Tensor Core requirements by cuBLAS or cuDNN version for some common data precisions. These requirements apply to matrix dimensions M, N, and K.**



| Tensor Cores can be used for... | cuBLAS version < 11.0<br>cuDNN version < 7.6.3 | cuBLAS version ≥ 11.0<br>cuDNN version ≥ 7.6.3 |
|---|---|---|
| INT8 | Multiples of 16 | Always but most efficient with multiples of 16; on A100, multiples of 128. |
| FP16 | Multiples of 8 | Always but most efficient with multiples of 8; on A100, multiples of 64. |
| TF32 | N/A | Always but most efficient with multiples of 4; on A100, multiples of 32. |
| FP64 | N/A | Always but most efficient with multiples of 2; on A100, multiples of 16. |

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

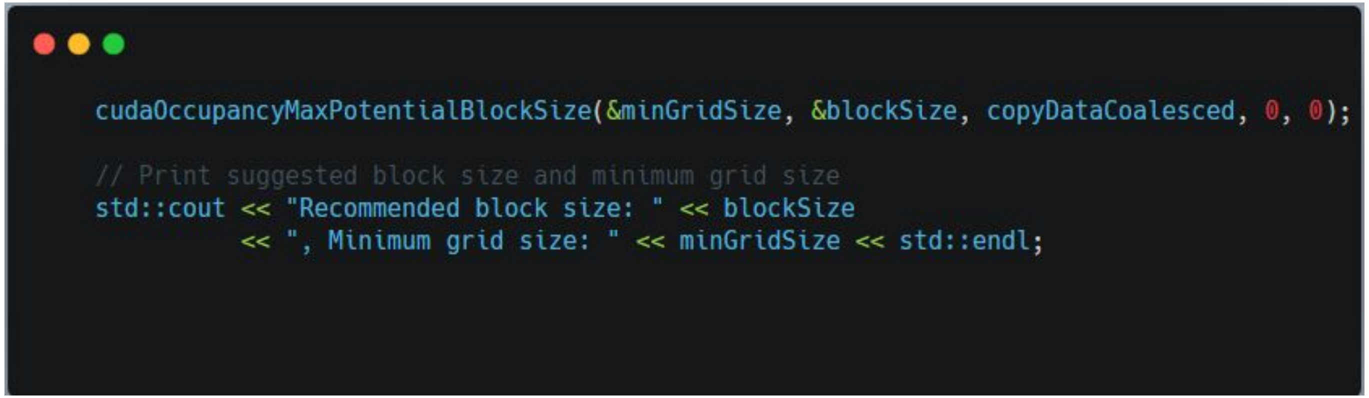
## Solution in CUDA Land: Change the kernel

Back to our case study let’s change the kernel launch params

## CUDA Occupancy calculator

This used to be an excel spreadsheet!

Turns out optimal for T4 is grid: 40 and blocks 1024

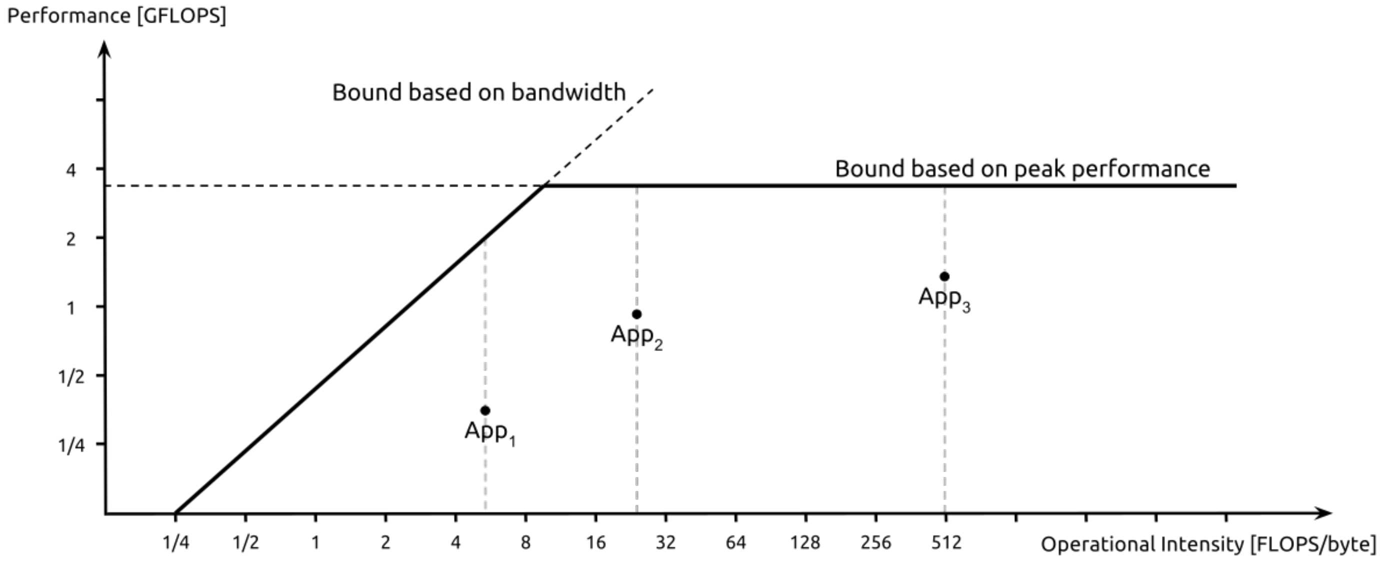


```
cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, copyDataCoalesced, 0, 0);

// Print suggested block size and minimum grid size
std::cout << "Recommended block size: " << blockSize
          << ", Minimum grid size: " << minGridSize << std::endl;
```

## Still has 1 more problem

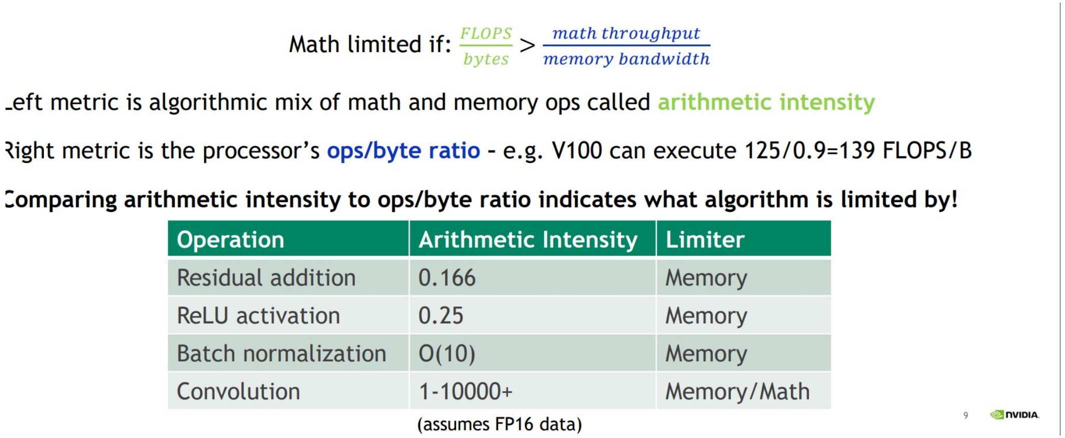
```
    WRN   Memory is more heavily utilized than Compute: Look at the Memory
Workload Analysis section to identify the

          DRAM bottleneck. Check memory replay (coalescing) metrics to make sure
you're efficiently utilizing the

          bytes transferred. Also consider whether it is possible to do more work per
memory access (kernel fusion) or

          whether there are values you can (re)compute.
```

# Roofline model



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

# Arithmetic intensity



Math limited if: $\frac{FLOPS}{bytes} > \frac{math\ throughput}{memory\ bandwidth}$

eft metric is algorithmic mix of math and memory ops called arithmetic intensity

ight metric is the processor's ops/byte ratio - e.g. V100 can execute 125/0.9=139 FLOPS/B

**omparing arithmetic intensity to ops/byte ratio indicates what algorithm is limited by!**

| Operation | Arithmetic Intensity | Limiter |
|---|---|---|
| Residual addition | 0.166 | Memory |
| ReLU activation | 0.25 | Memory |
| Batch normalization | O(10) | Memory |
| Convolution | 1-10000+ | Memory/Math |

(assumes FP16 data)

9 NVIDIA

https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9926-tensor-core-performance-the-ultimate-guide.pdf

## Back of the envelope

Apply ReLU function elementwise on a vector: f(x) = max(0, x)

For each element 1 read, 1 comparison op and maybe 1 write

Assume float32 -> 4 bytes (32 / 8)

Ops: 1

Byte: 2 * 4 = 8

So arithmetic intensity in the worst case ⅛

Best case 1/4

1/4 < 1: Memory bound!

## Let’s do this again with float16

Apply ReLU function elementwise on a vector: f(x) = max(0, x)

For each element 1 read, 1 comparison op and maybe 1 write

Assume float16 -> 2 bytes (16 / 8)

Ops: 1

Byte: 2 * 2 = 4

So arithmetic intensity in the worst case 1/4

## Ok last one: Matmul

A = [M, N]

B = [N, K]

C = A * B

FLOPS: For each output in C need dot product between row of A and column of B -> N multiplications and N additions -> M * K * 2N

Bytes: MN + NK (load each matrix) and then write the output (MK): MN + NK + MK

AI = 2MNK / (MN + NK + MK)

Compute bound for large matrices otherwise bandwidth bound

## TL;DR

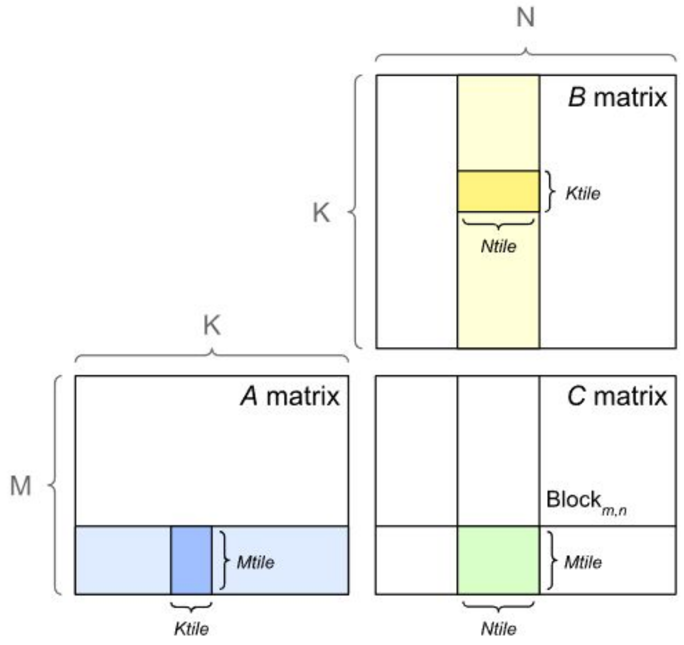
Bandwidth Bound Kernels: Fuse, quantize, compile

Compute Bound Kernels: Write a better algorithm

## Tiling of reused data

Already covered in past CUDA MODE lectures
https://www.youtube.com/@CUDAMODE

Less global memory traffic



## Minimize control divergence

Related to occupancy but if conditions can be a source of a lot of threads not doing anything, this is bad!

Drastic difference

processArrayWithDivergence took 0.074272 milliseconds

processArrayWithoutDivergence took 0.024704 milliseconds

ncu --set full divergence

(Not sure why ncu shows avg divergence as 0)

## Thread Coarsening

So far threads do as little as possible

But threads can do more and be faster

```
main ~/lecturex ./benchmark
VecAdd execution time: 0.245600 ms
VecAddCoarsened execution time: 0.015264 ms
```

Half as many threads launched

Why is this faster? Hint: AI

# Privatization

Apply partial updates to private copy of data before writing back to global or shared memory.

Example: Sliding window algorithm

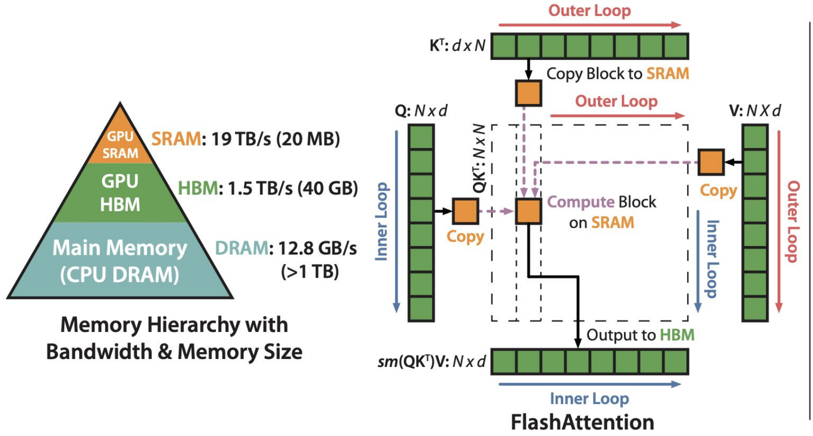
```
1   2  [3] [4] [5]  6   7
```

Higher occupancy

Higher compute SM throughput

Lower DRAM throughput

# softmax(QK^T) V



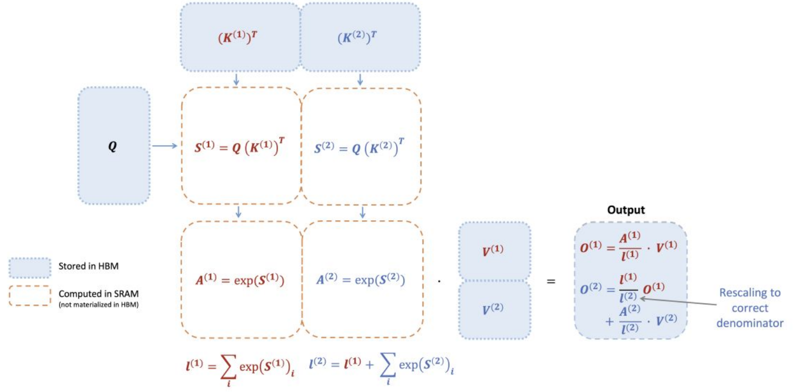

$(K^{(1)})^T$
$(K^{(2)})^T$
$Q$
$S^{(1)} = Q\,(K^{(1)})^T$
$S^{(2)} = Q\,(K^{(2)})^T$
Stored in HBM
Computed in SRAM (not materialized in HBM)
$A^{(1)} = \exp(S^{(1)})$
$A^{(2)} = \exp(S^{(2)})$
$V^{(1)}$
$V^{(2)}$
Output
$O^{(1)} = \frac{A^{(1)}}{l^{(1)}} \cdot V^{(1)}$
$O^{(2)} = \frac{l^{(1)}}{l^{(2)}} O^{(1)} + \frac{A^{(2)}}{l^{(2)}} \cdot V^{(2)}$
Rescaling to correct denominator
$l^{(1)} = \sum_i \exp(S^{(1)})_i$
$l^{(2)} = l^{(1)} + \sum_i \exp(S^{(2)})_i$

Figure 1: Diagram of how FlashAttention forward pass is performed, when the key **K** is partitioned into two blocks and the value **V** is also partitioned into two blocks. By computing attention with respect to each block and rescaling the output, we get the right answer at the end, while avoiding expensive memory reads/writes of the intermediate matrices **S** and **P**. We simplify the diagram, omitting the step in softmax that subtracts each element by the row-wise max.

https://arxiv.org/abs/2205.14135

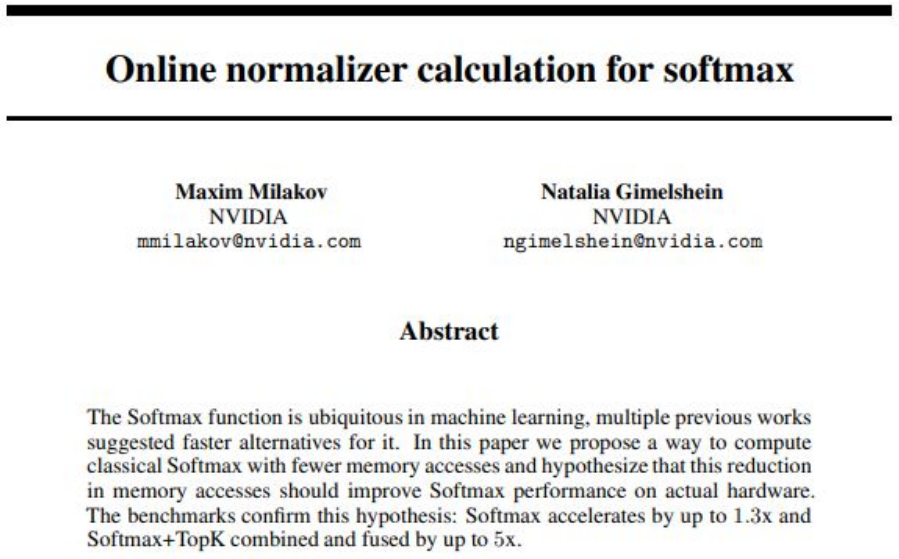
https://arxiv.org/abs/2307.08691

## Be better at math

Flash Attention v1 has 2 tricks

softmax(QK^T) V

- Tile based shared memory attention computation
- Online softmax



**Online normalizer calculation for softmax**

**Maxim Milakov**
NVIDIA
mmilakov@nvidia.com

**Natalia Gimelshein**
NVIDIA
ngimelshein@nvidia.com

**Abstract**

The Softmax function is ubiquitous in machine learning, multiple previous works suggested faster alternatives for it. In this paper we propose a way to compute classical Softmax with fewer memory accesses and hypothesize that this reduction in memory accesses should improve Softmax performance on actual hardware. The benchmarks confirm this hypothesis: Softmax accelerates by up to 1.3x and Softmax+TopK combined and fused by up to 5x.

https://arxiv.org/abs/1805.02867

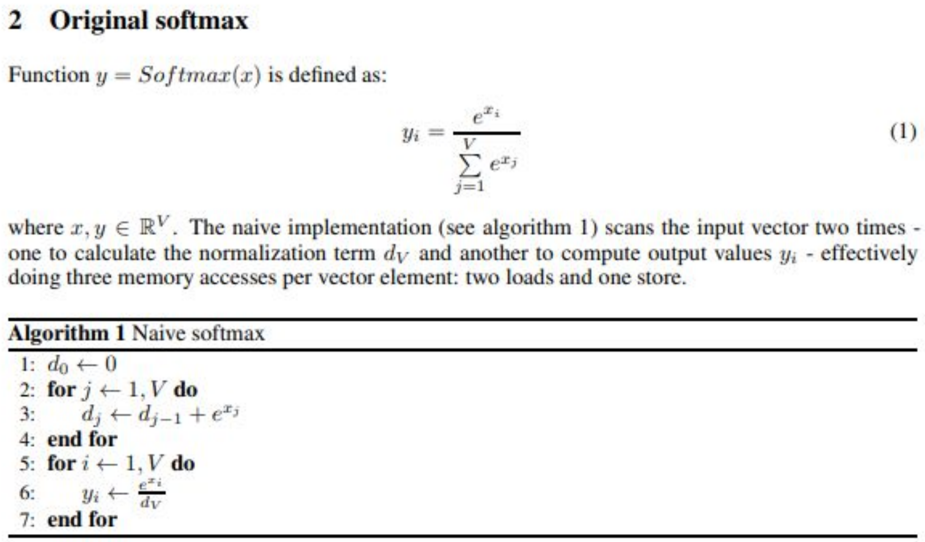
## Softmax

3 memory accesses per element

- 2 reads
- 1 store

Also can overflow!

1 solution is to store result of exponent in float64 but not great



**2 Original softmax**

Function $y = Softmax(x)$ is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}} \quad (1)$$

where $x, y \in \mathbb{R}^V$. The naive implementation (see algorithm 1) scans the input vector two times - one to calculate the normalization term $d_V$ and another to compute output values $y_i$ - effectively doing three memory accesses per vector element: two loads and one store.

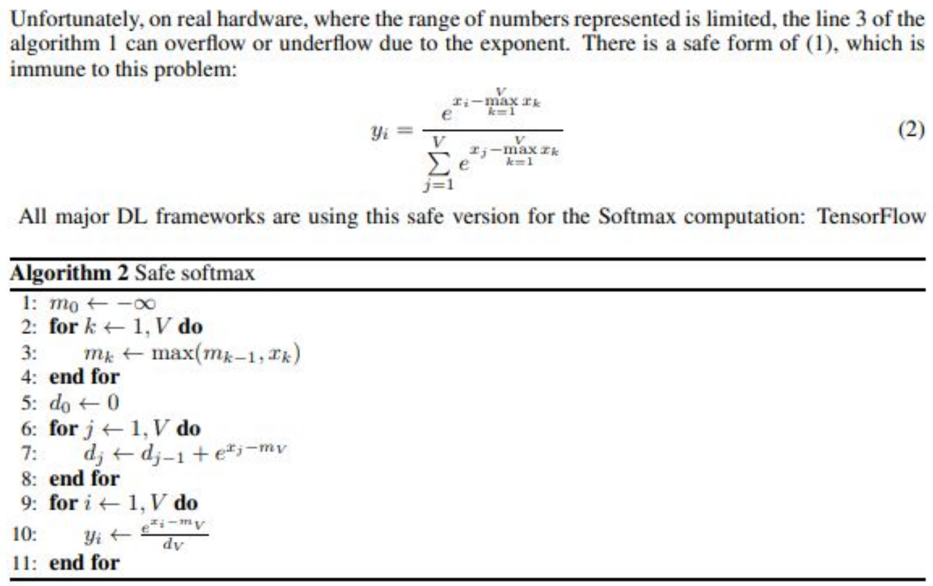
**Algorithm 1** Naive softmax

```
1: $d_0 \leftarrow 0$
2: for $j \leftarrow 1, V$ do
3:     $d_j \leftarrow d_{j-1} + e^{x_j}$
4: end for
5: for $i \leftarrow 1, V$ do
6:     $y_i \leftarrow \frac{e^{x_i}}{d_V}$
7: end for
```

# Safe softmax

4 memory access per element

- 3 read
- 1 store



Unfortunately, on real hardware, where the range of numbers represented is limited, the line 3 of the algorithm 1 can overflow or underflow due to the exponent. There is a safe form of (1), which is immune to this problem:

$$y_i = \frac{e^{x_i - \max\limits_{k=1}^{V} x_k}}{\sum\limits_{j=1}^{V} e^{x_j - \max\limits_{k=1}^{V} x_k}} \tag{2}$$

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

**Algorithm 2** Safe softmax

```
1: $m_0 \leftarrow -\infty$
2: for $k \leftarrow 1, V$ do
3:     $m_k \leftarrow \max(m_{k-1}, x_k)$
4: end for
5: $d_0 \leftarrow 0$
6: for $j \leftarrow 1, V$ do
7:     $d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: end for
9: for $i \leftarrow 1, V$ do
10:    $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
11: end for
```

## Online Softmax

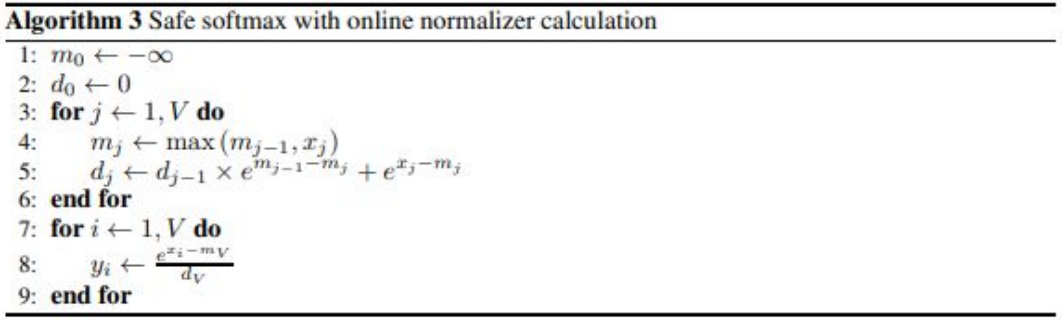
Key insight is line 5 where we can do localwise corrections

e^{m-n} = e^m e^-n

Cancel out the fake max and use the true max

3 memory access per element

- 2 read
- 1 store



**Algorithm 3** Safe softmax with online normalizer calculation

```
1: $m_0 \leftarrow -\infty$
2: $d_0 \leftarrow 0$
3: for $j \leftarrow 1, V$ do
4:     $m_j \leftarrow \max(m_{j-1}, x_j)$
5:     $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
6: end for
7: for $i \leftarrow 1, V$ do
8:     $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
9: end for
```

# TL;DR

Table from the Book

+ Compute vs Memory Bound workflows

+ Better Math

CUDA MODE: Wield math and computers to reach brrr nirvana

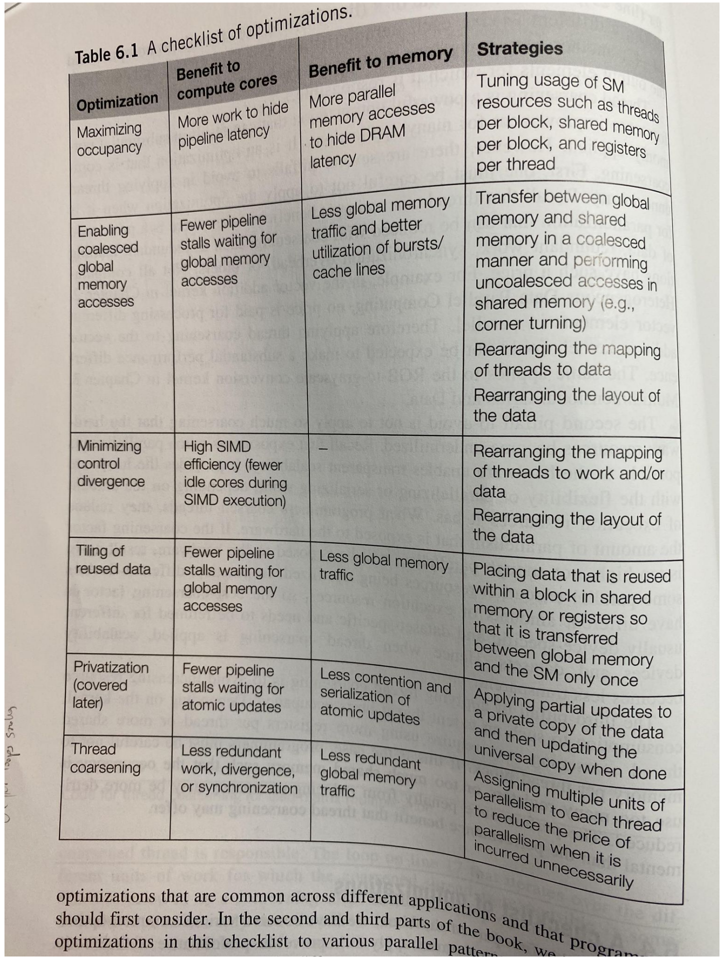


Table 6.1 A checklist of optimizations.

| Optimization | Benefit to compute cores | Benefit to memory | Strategies |
|---|---|---|---|
| Maximizing occupancy | More work to hide pipeline latency | More parallel memory accesses to hide DRAM latency | Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread |
| Enabling coalesced global memory accesses | Fewer pipeline stalls waiting for global memory accesses | Less global memory traffic and better utilization of bursts/cache lines | Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning)<br>Rearranging the mapping of threads to data<br>Rearranging the layout of the data |
| Minimizing control divergence | High SIMD efficiency (fewer idle cores during SIMD execution) | – | Rearranging the mapping of threads to work and/or data<br>Rearranging the layout of the data |
| Tiling of reused data | Fewer pipeline stalls waiting for global memory accesses | Less global memory traffic | Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once |
| Privatization (covered later) | Fewer pipeline stalls waiting for atomic updates | Less contention and serialization of atomic updates | Applying partial updates to a private copy of the data and then updating the universal copy when done |
| Thread coarsening | Less redundant work, divergence, or synchronization | Less redundant global memory traffic | Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily |

[illegible]

optimizations that are common across different applications and that program
should first consider. In the second and third parts of the book, we
optimizations in this checklist to various parallel pattern