# FlashAttention

wooosh!

# Erwartung: PyLEGO + Magie

- Layer oder ganze Modelle wie Bausteine behandeln
- ...und magisch funktioniert alles und ist schnell!

→model.to(“cuda”) nutzt hocheffiziente CUDA Libraries wie cuBLAS, die MatMul durchgespielt haben

# Erwartung: PyLEGO + Magie

- Layer oder ganze Modelle wie Bausteine behandeln
- ...und magisch funktioniert alles und ist schnell!

→model.to(“cuda”) nutzt hocheffiziente CUDA Libraries wie cuBLAS, die MatMul durchgespielt haben

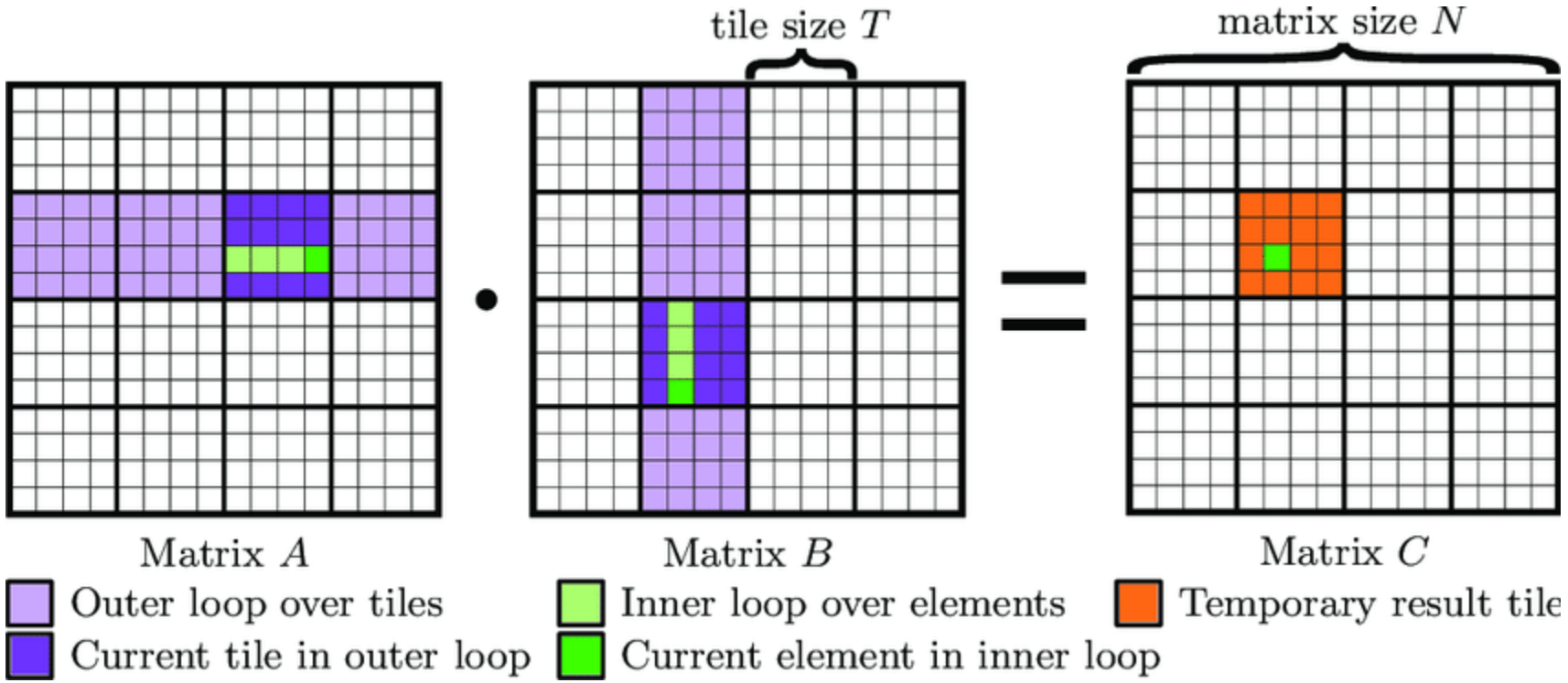
**FALSCH!** (so’n bisschen)

# Hintergrund: Naives MatMul

- Input: matrices $A$ and $B$
- Let $C$ be a new matrix of the appropriate size
- For $i$ from 1 to $n$:
  - For $j$ from 1 to $p$:
    - Let $\text{sum} = 0$
    - For $k$ from 1 to $m$:
      - Set $\text{sum} \leftarrow \text{sum} + A_{ik} \times B_{kj}$
    - Set $C_{ij} \leftarrow \text{sum}$
- Return $C$

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

# Hintergrund: MatMul mit Tiling




https://doi.org/10.48550/arXiv.1706.10086

# Tiling: The most ancient joke in the book

- Kennt man, seit die Memory Wall wichtig ist
- Wird auch in ETI gelehrt (Einführung in die technische Informatik)

- Tut cuBLAS doch sicher auch, oder?

# Tiling: The most ancient joke in the book

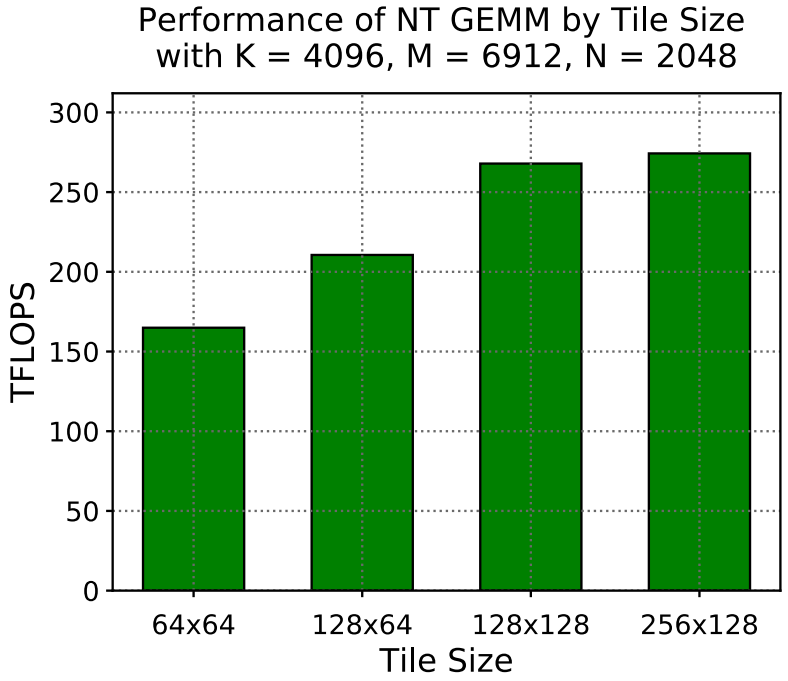
- Kennt man, seit die Memory Wall wichtig ist
- Wird auch in ETI gelehrt (Einführung in die technische Informatik)

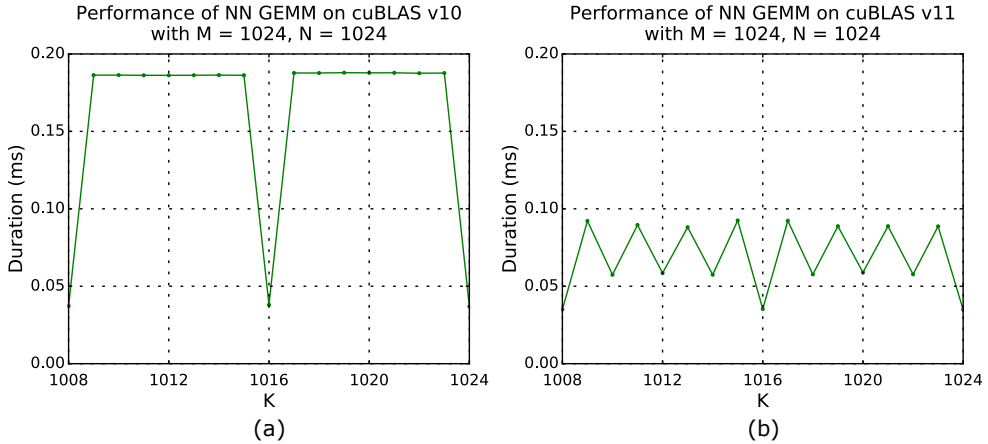
- Tut cuBLAS doch sicher auch, oder?

**ODER???**

# Tiling in cuBLAS

- Klar macht cuBLAS Tiling ☺
- Blockgröße aber unklar
  - Große Blöcke: Weniger Speicherzugriffe, mehr Rechnen
  - Zu groß: Passt nicht in L1
  - Heuristik zur Auswahl in cuBLAS

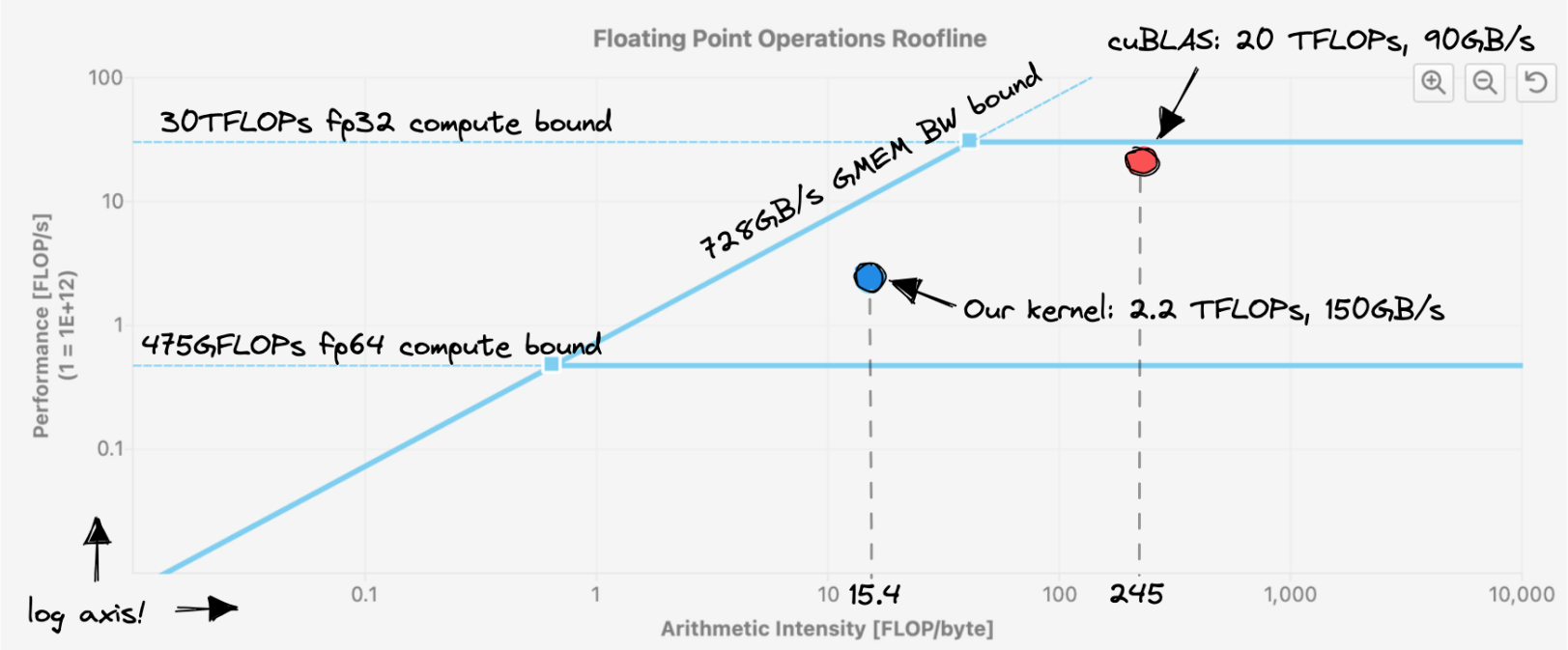


# Roofline Model und Arithmetische Intensität

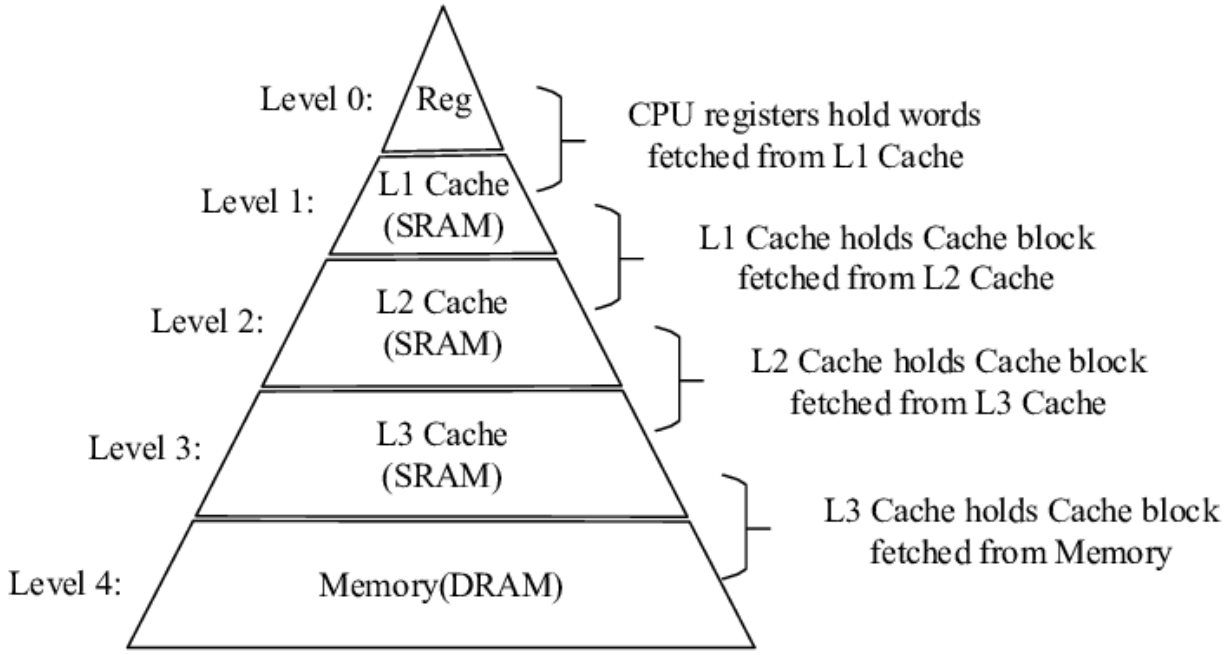
$$\text{Arithmetic Intensity} = \frac{\text{number of FLOPS}}{\text{number of byte accesses}} = \frac{2 \cdot (M \cdot N \cdot K)}{2 \cdot (M \cdot K + N \cdot K + M \cdot N)} = \frac{M \cdot N \cdot K}{M \cdot K + N \cdot K + M \cdot N}$$

Mit gegebener **Mem BW** und gegebener **Arith. Int.**, wie viele FLOPS können wir erreichen?




https://siboehm.com/articles/22/CUDA-MMM

# Der SRAM/Cache/... Refresher für Anja et al.

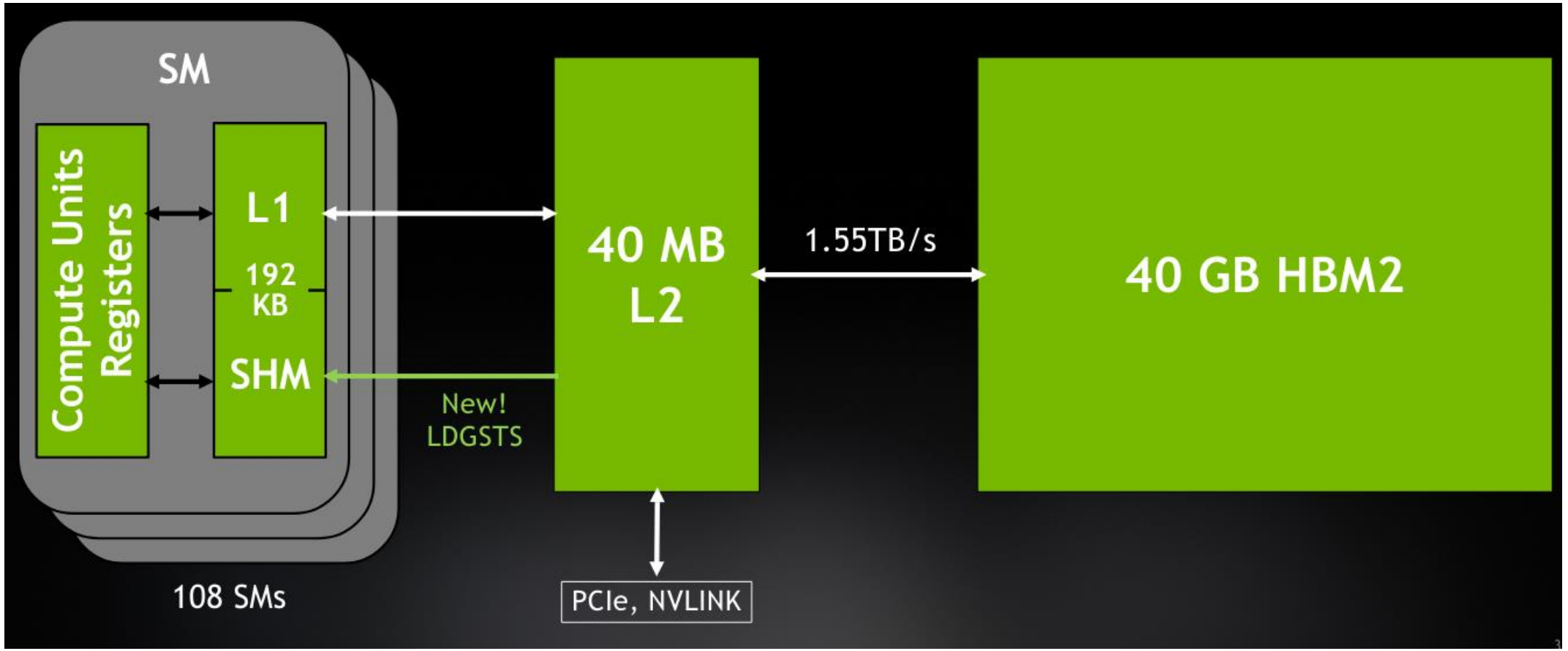



https://doi.org/10.1109/ICCCS55155.2022.9846079

# Der SRAM/Cache/… Refresher für Anja et al.


https://resources.nvidia.com/en-us-tensor-core

# Der SRAM/Cache/... Refresher für Anja et al.




https://blog.paperspace.com/a-complete-anatomy-of-a-graphics-card-case-study-of-the-nvidia-a100/

# FlashAttention

# Kernel Fusion

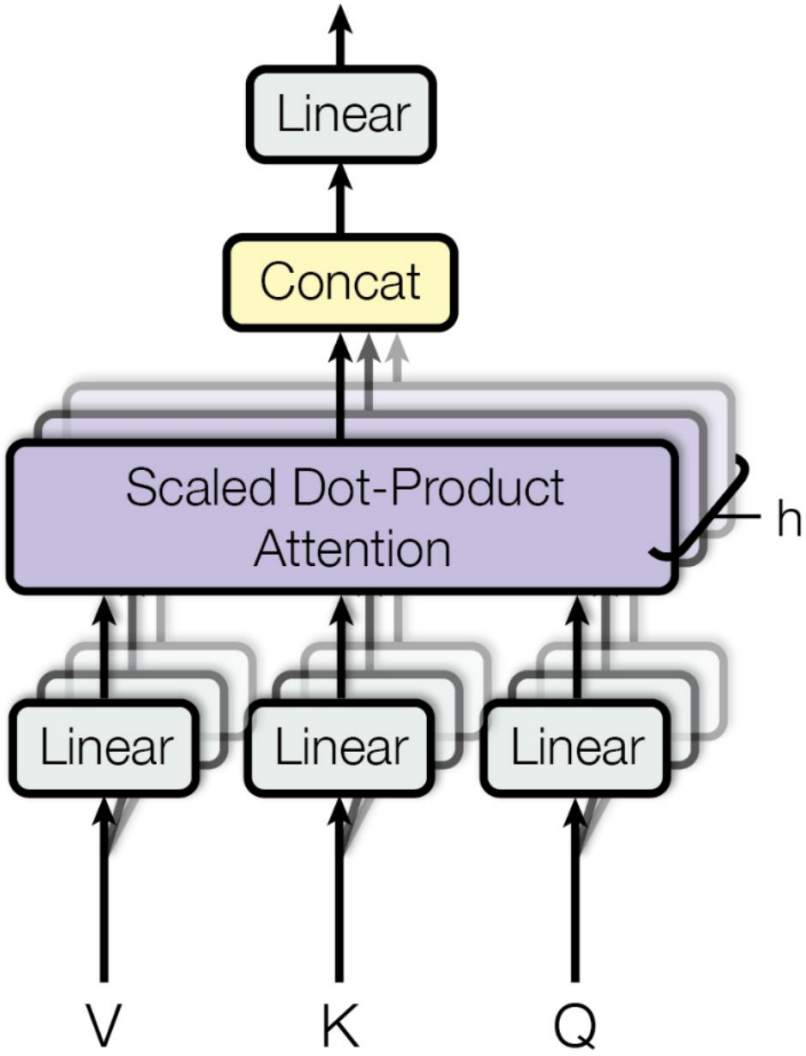
- Vertikale Zerlegung des Problems
  - ...also muss Softmax auch zerlegt werden
- Soweit ersichtlich, Attention teilweise berechnen

$$m(x) := \max_i \; x_i, \quad f(x) := \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix}, \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

...wird zu

$$m(x) = m(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \begin{bmatrix} e^{m(x^{(1)}) - m(x)} f(x^{(1)}) & e^{m(x^{(2)}) - m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$



# Tiling in Flash Attention

**Algorithm 1** FlashAttention

**Require:** Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size $M$.

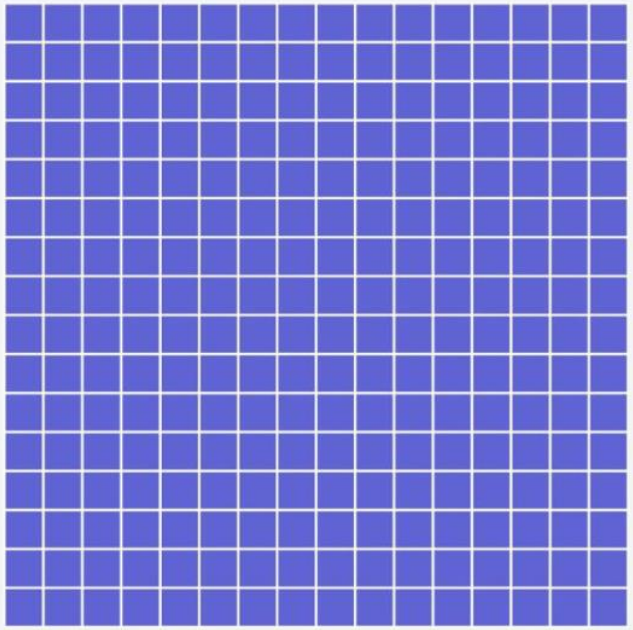
1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\ell$ into $T_r$ blocks $\ell_i, \ldots, \ell_{T_r}$ of size $B_r$ each, divide $m$ into $T_r$ blocks $m_1, \ldots, m_{T_r}$ of size $B_r$ each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

SRAM-Bedarf:

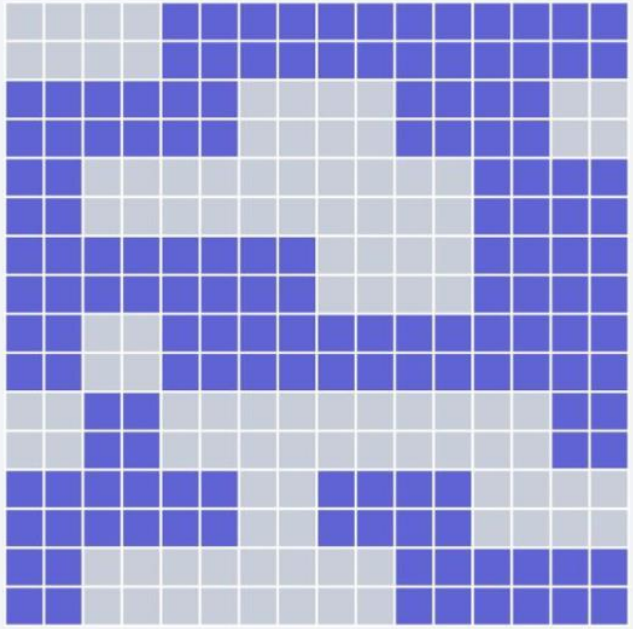
$$B_r \cdot d + B_c \cdot d \cdot 2 + B_r \cdot 2$$
$$= B_r(d + 2) + B_c(2d)$$
$$= \cdots = \frac{M(3d + 2)}{4d}$$

# Sparsity

Dense Weights



2×2 Block-Sparse Weights

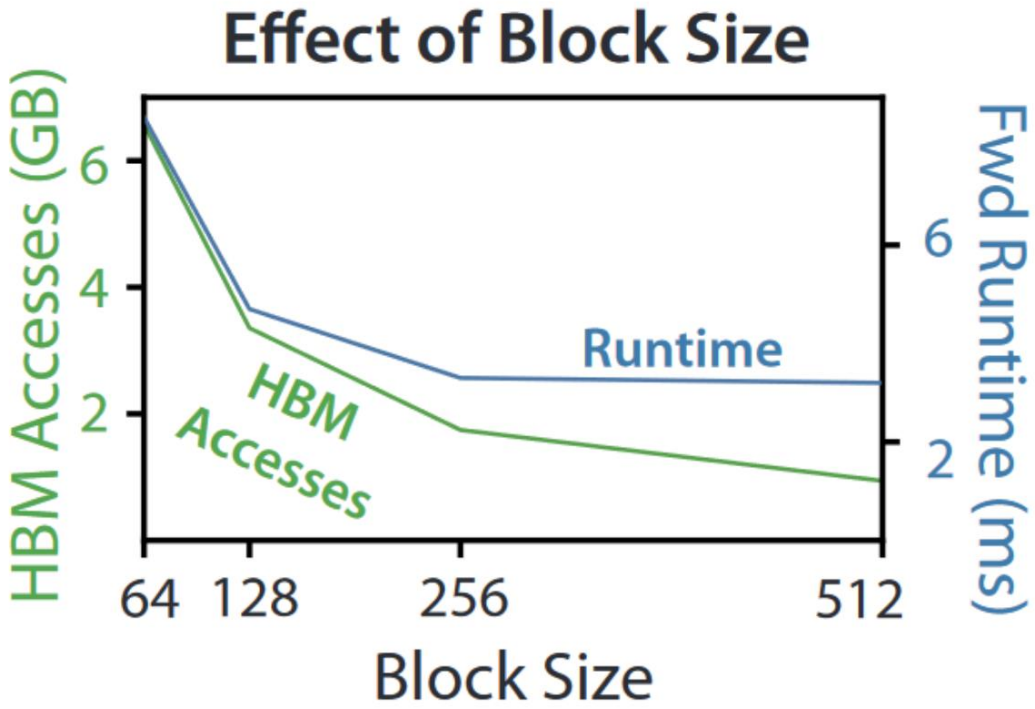


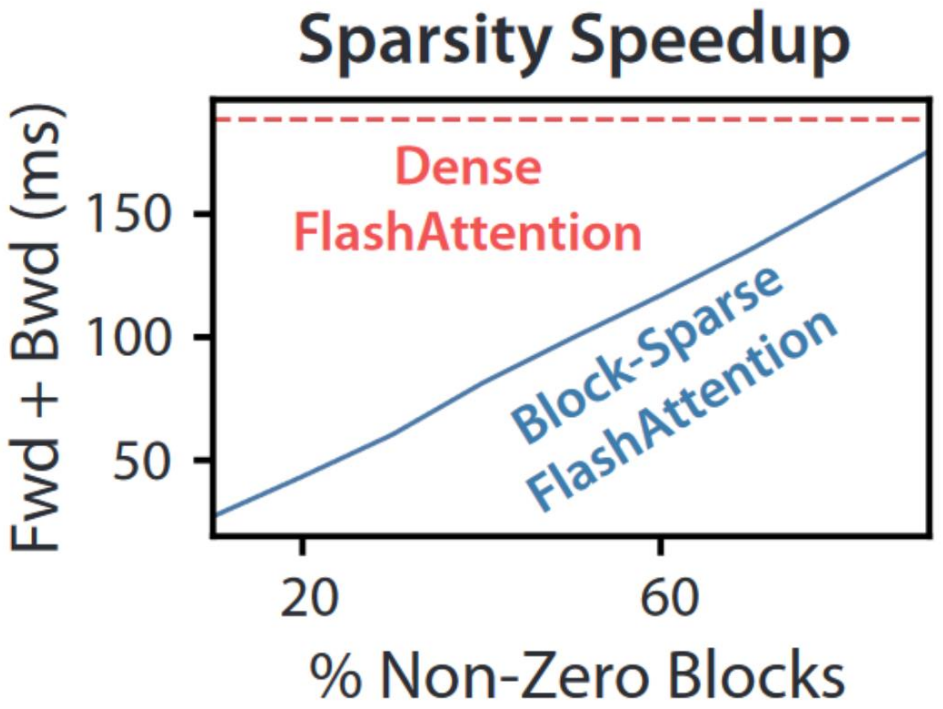
Corresponding Sparsity Pattern

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

http://www.jfrankle.com/lth-block-sparsity.pdf

# Ergebnisse


Effect of Block Size
HBM Accesses (GB)
6
4
2
Fwd Runtime (ms)
6
2
Runtime
HBM
Accesses
64
128
256
512
Block Size


Sparsity Speedup
Fwd + Bwd (ms)
150
100
50
Dense
FlashAttention
Block-Sparse
FlashAttention
20
60
% Non-Zero Blocks

# Benchmarks

- Natürlich alles schneller und besser!
  - BERT Training 1.15x besser
  - GPT-2 3x schneller als HuggingFace oder 1.8x schneller als Megatron
  - LRA 2.4x schneller

  - Trainiert GPT-2 mit 4K Context Schneller als vorher mit 1K
  - Erstmals besser-als-random in Path-X
  - Block Sparsity erstmals besser-als-random in Path-256

  - Außerdem speichereffizient

# Zu LRA...

| Models | ListOps | Text | Retrieval | Image | Pathfinder | Avg | Speedup |
|---|---|---|---|---|---|---|---|
| Transformer | 36.0 | 63.6 | 81.6 | 42.3 | 72.7 | 59.3 | - |
| FlashAttention | 37.6 | 63.9 | 81.4 | 43.5 | 72.7 | 59.8 | 2.4× |
| Block-sparse FlashAttention | +1.4 37.0 -1.8 | +7 63.0 -1.1 | 81.3 -0.9 | 43.6 -0 | 73.3 -0 | 59.6 -2 | **2.8×** |
| Linformer [84] | 35.6 | 55.9 | 77.7 | 37.8 | 67.6 | 54.9 | 2.5× |
| Linear Attention [50] | 38.8 | 63.2 | 80.7 | 42.6 | 72.5 | 59.6 | 2.3× |
| Performer [12] | 36.8 | 63.6 | 82.2 | 42.1 | 69.9 | 58.9 | 1.8× |
| Local Attention [80] | 36.1 | 60.2 | 76.7 | 40.6 | 66.6 | 56.0 | 1.7× |
| Reformer [51] | 36.5 | 63.8 | 78.5 | 39.6 | 69.4 | 57.6 | 1.3× |
| Smyrf [19] | 36.1 | 64.1 | 79.0 | 39.6 | 70.5 | 57.9 | 1.7× |