SUMMER SCHOOL 2023

Computing einsum expressions using LIBXSMM

Max Koch

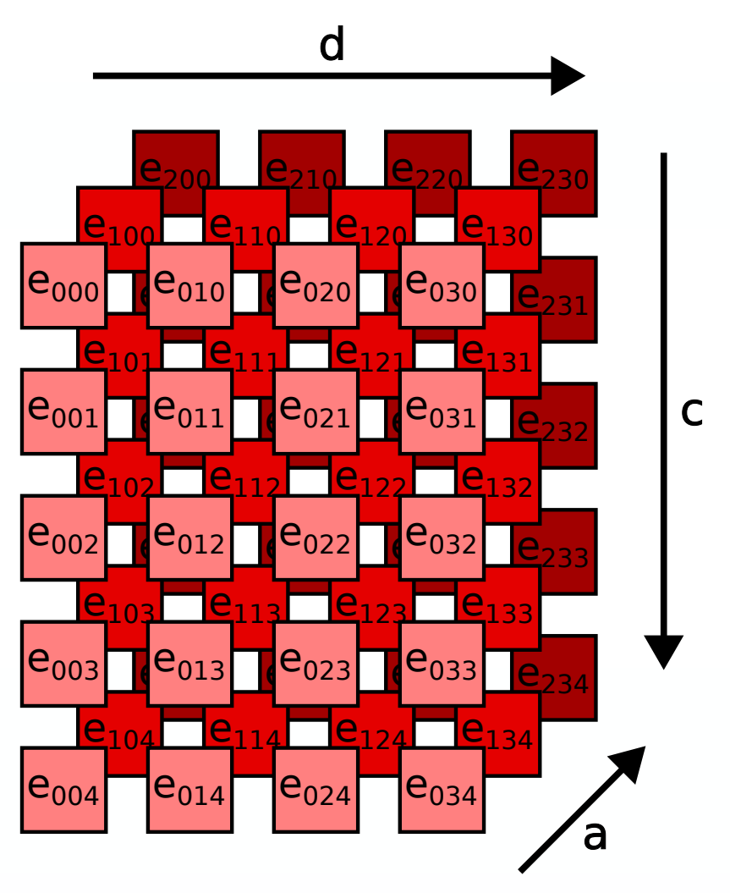
Friedrich Schiller University Jena

ai.uni-jena.de

INTRODUCTION TO EINSUM EXPRESSIONS

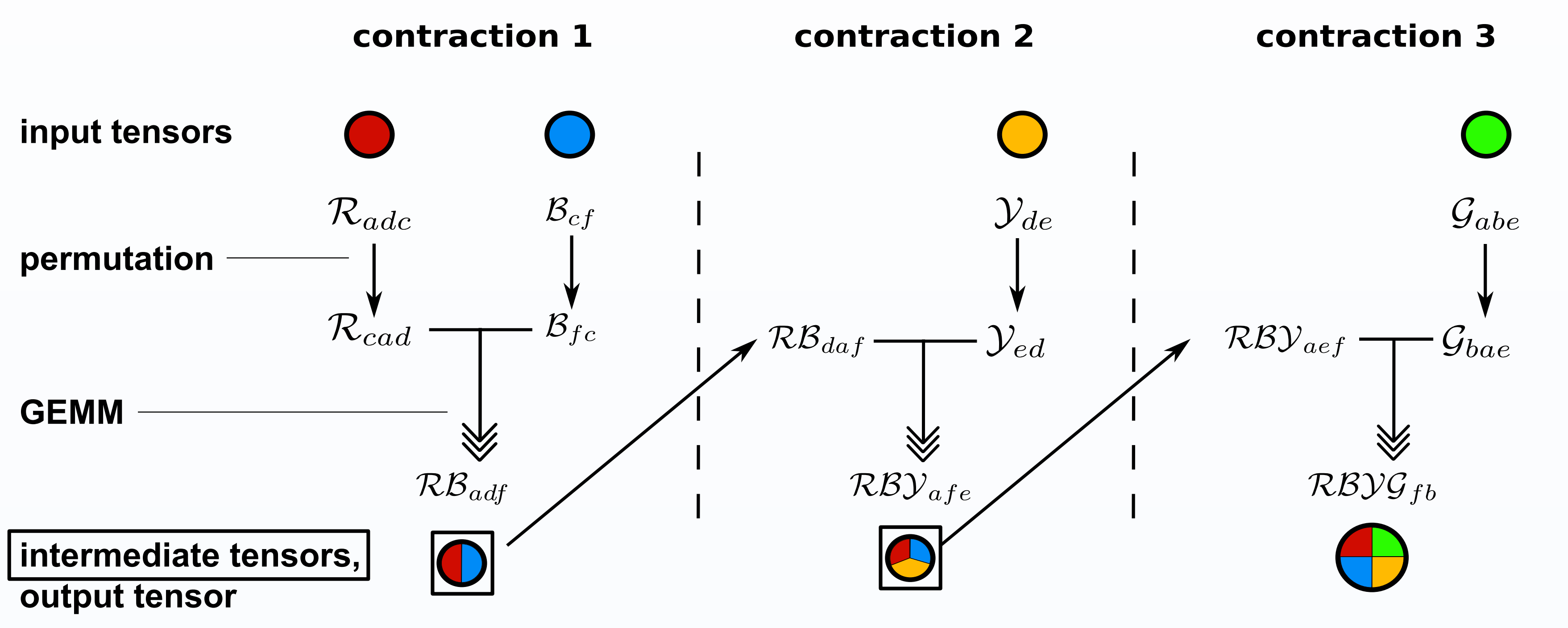
The Einstein summation convention (einsum) is a powerful notation to express operations on tensors. In my research, I tried to find an approach to compute einsum expressions in a way that reduces memory overhead. The following is an einsum encoding a tensor network contraction:

adc, cf, de, abe -> fb



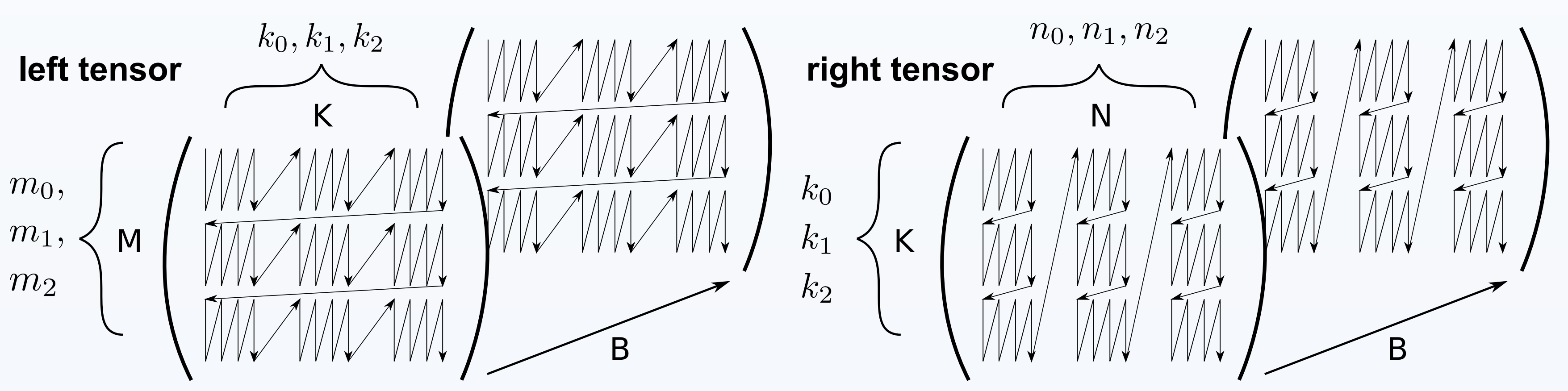
Multiple sub-strings divided by commas reference different tensors. The characters are dimension identifiers for a single tensor. The "->" divides the expression into input tensors (left) and the output tensor (right).

To compute an einsum, it is beneficial to contract input tensors in binary fashion until one final tensor is left. It is further possible to compute binary einsum contractions using general matrix multiplications (GEMMs), harnessing the performance of highly optimized subroutines. The backbone for matrix multiplications of my tests was LIBXSMM, a C++ library targeting small matrix multiplications. More traditional contraction approaches must ensure that tensors have a specific memory layout in order to be able to perform a GEMM. Normally, this memory layout is achieved by **permuting** the tensors.



MEMORY LAYOUT FOR INTERMEDIATE TENSORS

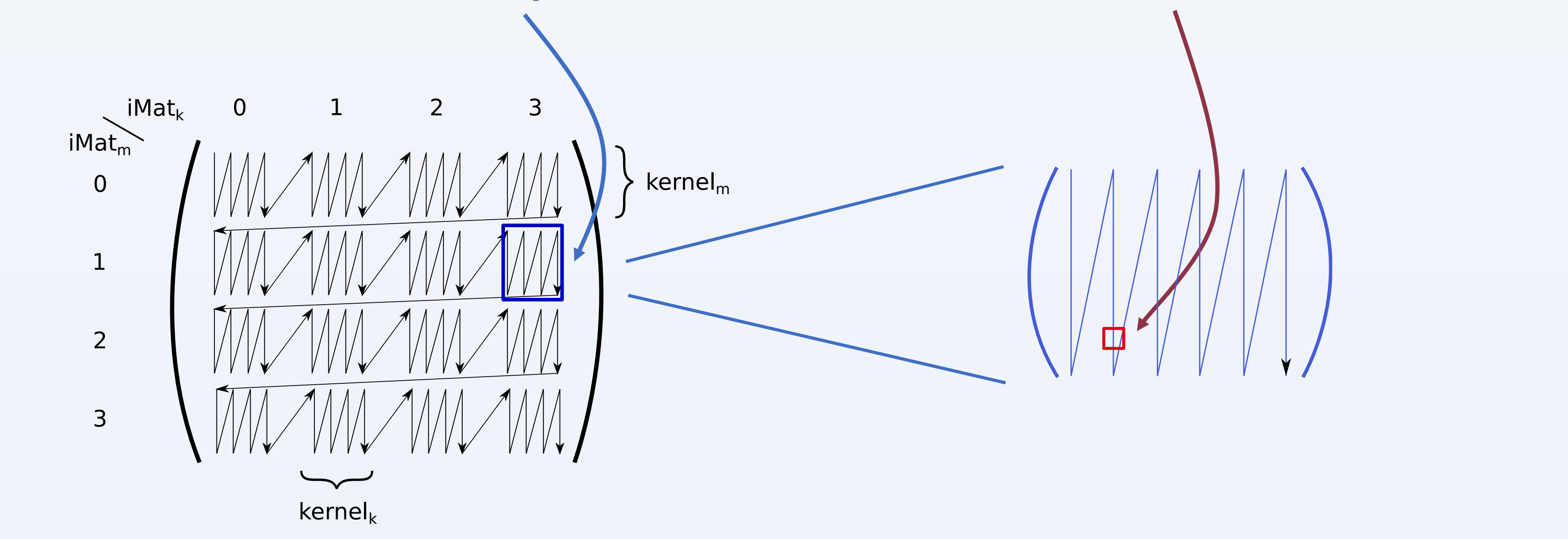
Permuting an intermediate tensor after its calculation results in a memory overhead. Multiple memory operations can be merged into one. By merging the matrix write-back and the permutation, one essentially gains a free permutation of the tensor. It is now possible to describe a memory layout for intermediate tensors that reduces the execution time:



Furthermore, an unpacking routine that writes respective elements to their correct position in memory can be built, so that the result elements in the registers are directly written to corresponding addresses. No additional permutation is needed.

Unpack element of kernel sub-matrix given indices for each dimension

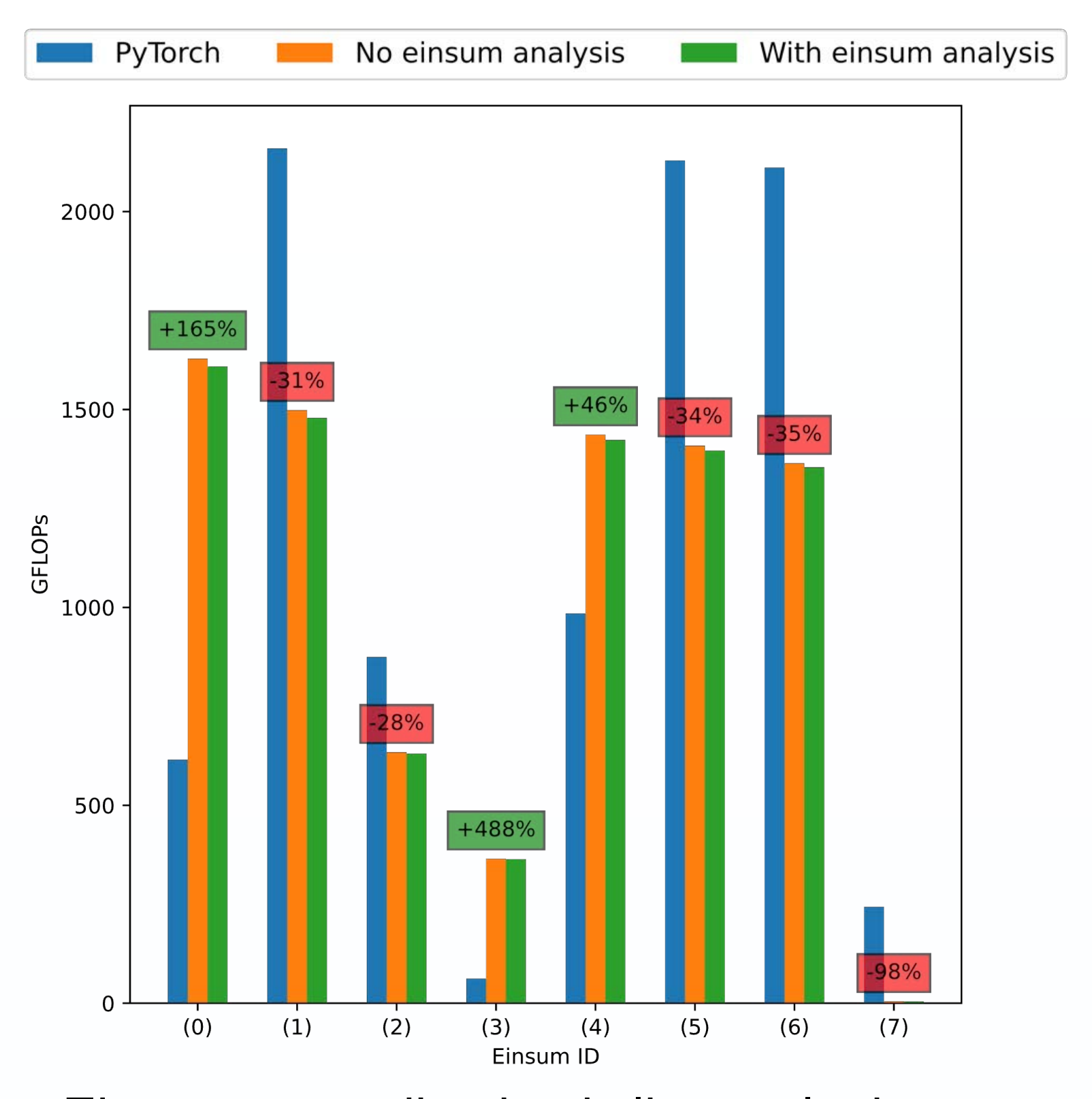
1. Calculate indices of output sub-matrix **2. Calculate offset inside the matrix**



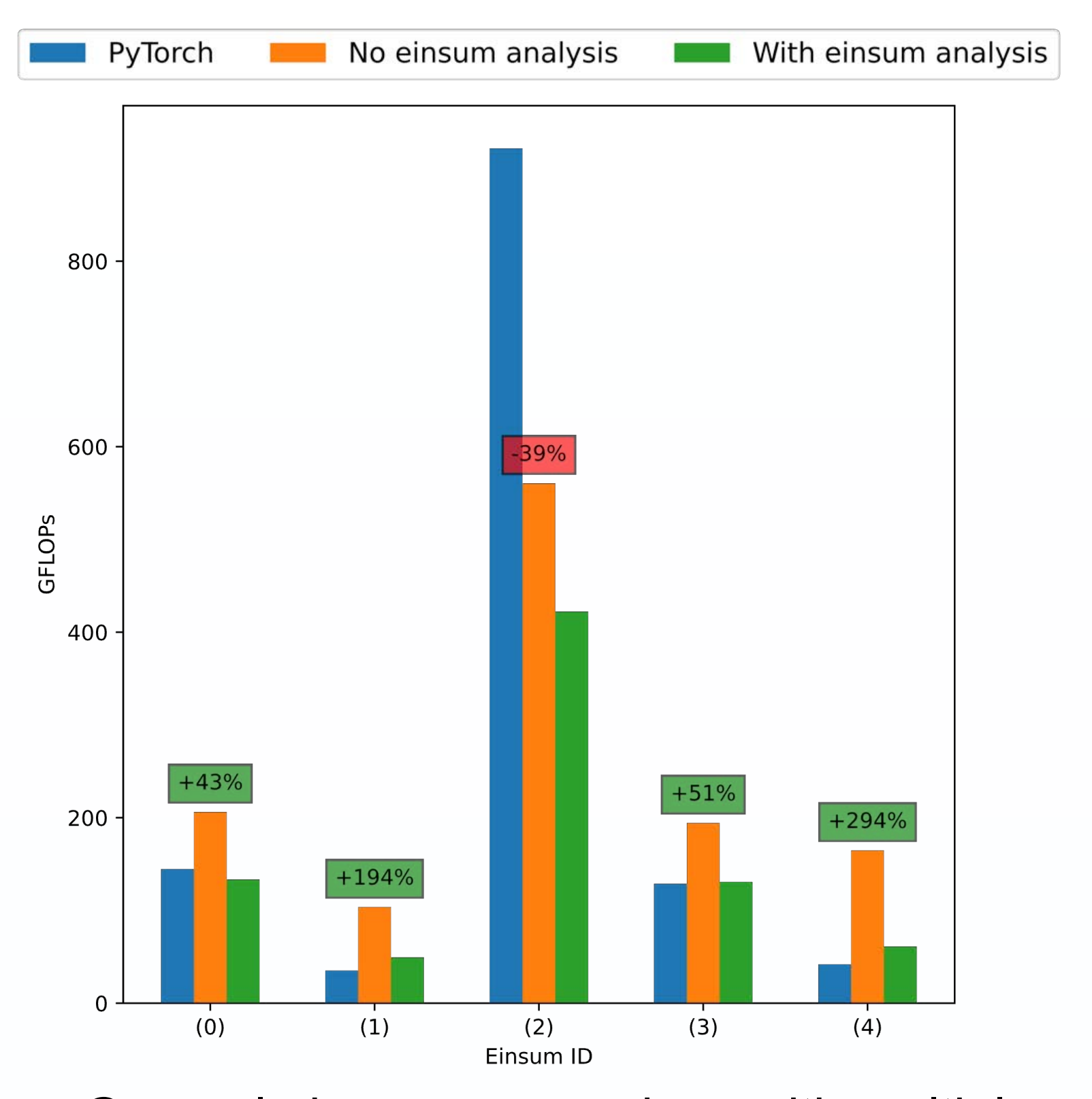
3. Write element to memory of the output tensor (possibly suboptimal memory access patterns)

RUNTIME PERFORMANCES

The runtimes of the approach with dedicated memory layouts for intermediate tensors were compared to the ones of PyTorch's einsum implementation. PyTorch analyzes the einsum string every time the subroutine is called. In contrast, my implementation allows for a preceding analysis with basic offset caching, which has to be computed only once. The same einsum expression can be processed with different input data without further analysis.

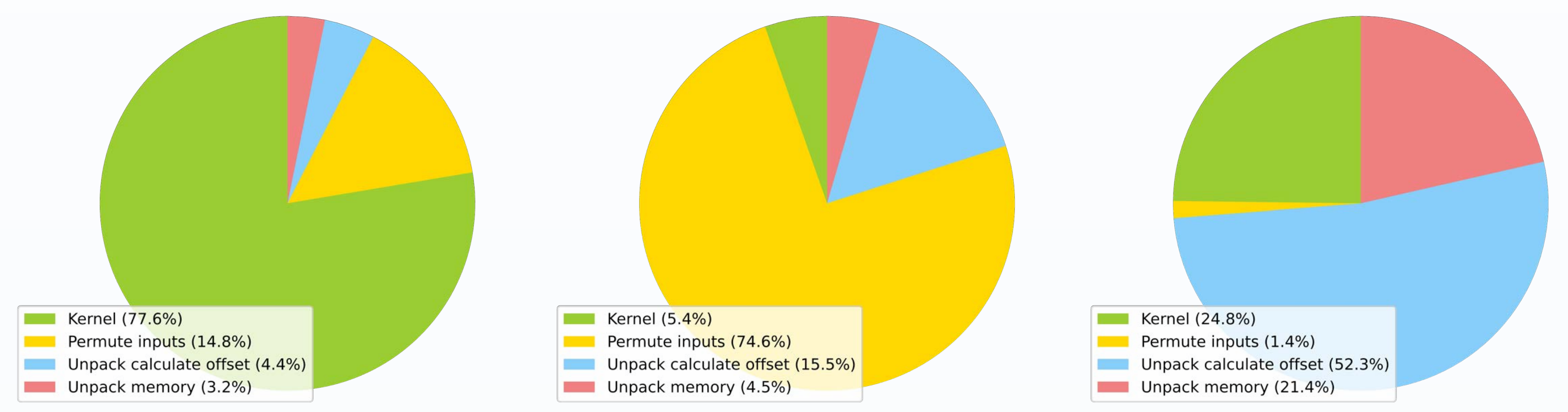


Einsums encoding basic linear algebra operations like (chain) matrix multiplications



General einsum expressions with multiple (5 – 20) input tensors

The piecharts display a more detailed runtime analysis. Wedges show relative runtimes of the time taken by the kernel, permuting the inputs (they are not intermediate tensors, so they have to be permuted initially), and the unpacking routine.



km, nk -> nm
(dimension sizes: 2048)

abcd, beft, cdjl, fhij, gikl -> aehk
(dimension sizes: left 2, right 8)

CONCLUSIONS

The calculation of offsets takes a large amount of time. The reason for that is the way elements are being chosen to be part of different sub-matrices. Elements are fetched by increasing indices a fixed number of times. This ensures a desired kernel shape but complicates offset calculations. In the future, it is possible to research more memory layouts for intermediate tensors that might perform better.

order of increasing the indices

$\longleftarrow$ $\longleftarrow$

	(k_2, k_0, k_1)	(k_0, k_1, k_2)
Matrix 1	$(0,0,0)$	$(0,0,0)$
	$(0,0,1)$	$(0,0,1)$
	$(0,1,0)$	$(0,0,2)$
	$(0,1,1)$	$(0,1,0)$
Matrix 2	$(1,0,0)$	$(0,1,1)$
	$(1,0,1)$	$(0,1,2)$
	$(1,1,0)$	$(1,0,0)$
	$(1,1,1)$	$(1,0,1)$
Matrix 3	$(2,0,0)$	$(1,0,2)$
	$(2,0,1)$	$(1,1,0)$
	$(2,1,0)$	$(1,1,1)$
	$(2,1,1)$	$(1,1,2)$

$$\boxed{\begin{array}{c} |k_0| = 2 \\ |k_1| = 2 \\ |k_2| = 3 \end{array}}$$

The right column is the current approach.

PRO:
- no search for dimension order
- guaranteed kernel shape

CON:
- complicated offset calculations

REFERENCES

Alexander Heinecke et al. "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation". In: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2016, pp. 981–991. doi: 10.1109/SC.2016.83.

Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. Red Hook, NY, USA: Curran Associates Inc., 2019.

Tensor Network. https://tensornetwork.org/.

CONTACT

Max Koch
max.koch.m@gmx.de