Linear Layouts: Robust Code Generation of Efficient Tensor Computation Using F_2

Keren Zhou*, Mario Lezcano*, Adam Goucher*, Akhmed Rakhmati, Jeff Niu, Justin Lebar, Pawel Szczerbuk, Peter Bell, Phil Tillet, Thomas Raoux, Zahi Moudallal

$$w = L \times v$$

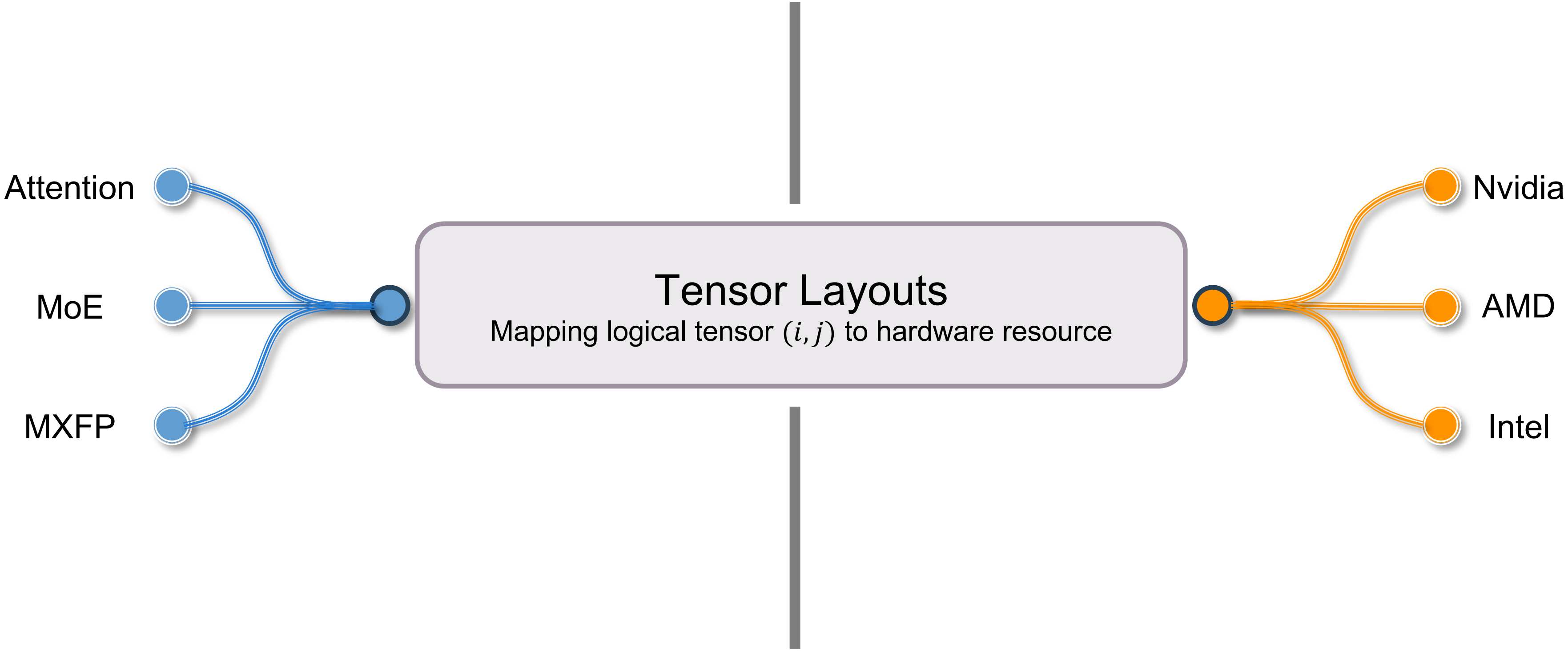
$$\underbrace{\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}}_{w} = \underbrace{\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}}_{L} \times \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}}_{v}$$

Outline

- Background
- Linear Layouts
- Code Generation
- Experiments

Background

Increased Model Complexity and Hardware Diversity



Tile-Based Programming Languages

- Users mainly focus on dividing operand tensors into *tiles*
  - Layout-related code generation are offloaded the compiler

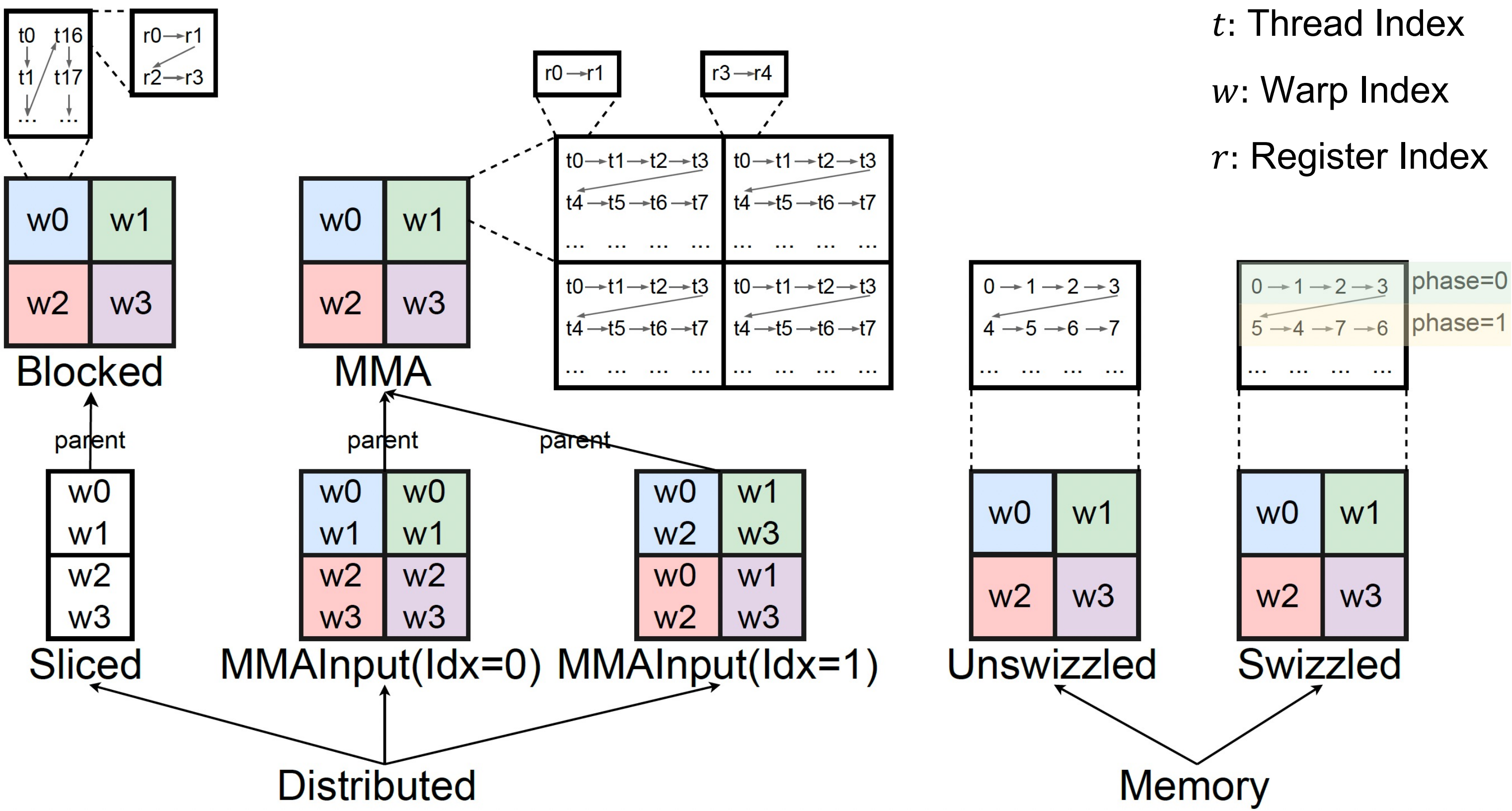
```
z: dim0 x dim1 = x: dim0 x dim1 + y: dim0 x dim1
```

- Kernel decorator
- Programming model
- Creation ops
- Memory ops

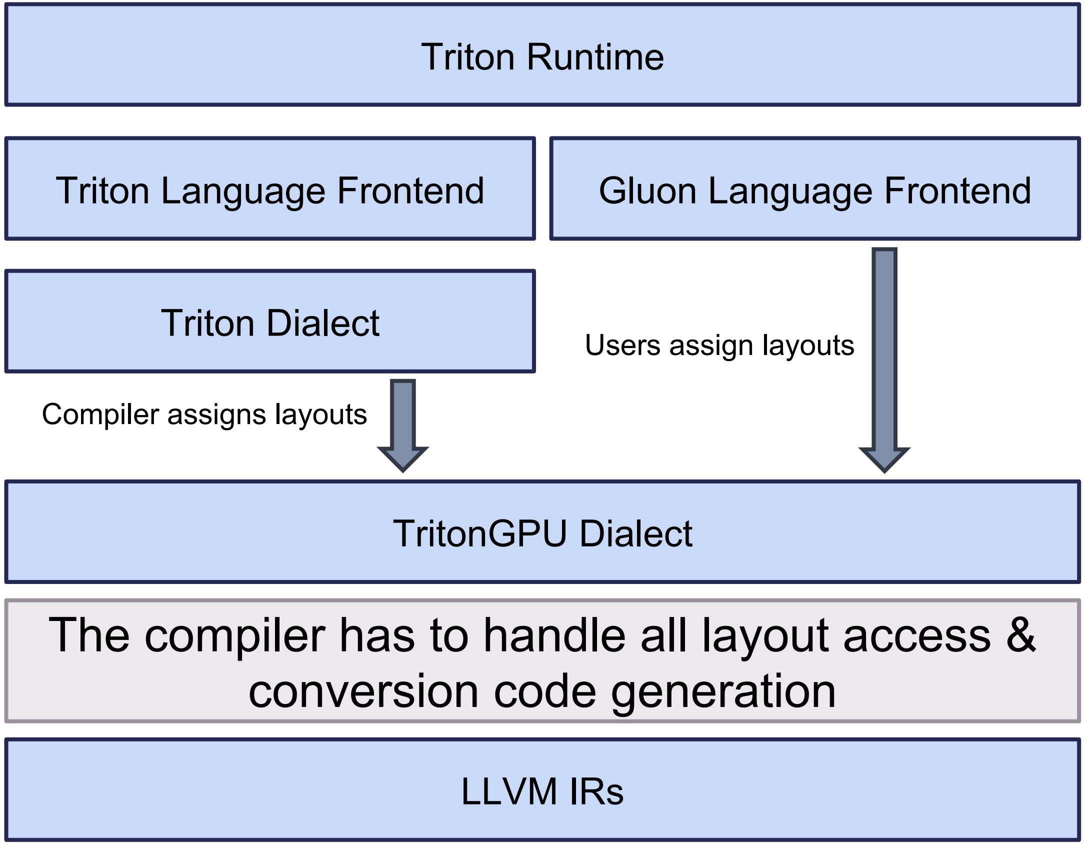
vecAdd

```python
@triton.jit
def add_kernel(x_ptr, y_ptr, z_ptr, dim0, dim1,
               BLOCK_DIM0: tl.constexpr, BLOCK_DIM1: tl.constexpr):
  pid_x = tl.program_id(axis=0)
  pid_y = tl.program_id(axis=1)
  block_start = pid_x * BLOCK_DIM0 * dim1 + pid_y * BLOCK_DIM1
  offsets_dim0 = tl.arange(0, BLOCK_DIM0)[:,None]
  offsets_dim1 = tl.arange(0, BLOCK_DIM1)[None, :]
  offsets = block_start + offsets_dim0 * dim1 + offsets_dim1
  masks = (offsets_dim0 < dim0) & (offsets_dim1 < dim1)
  x = tl.load(x_ptr + offsets, mask=masks)
  y = tl.load(y_ptr + offsets, mask=masks)
  output = x + y
  tl.store(z_ptr + offsets, output, mask=masks)
```

Tensor Layouts

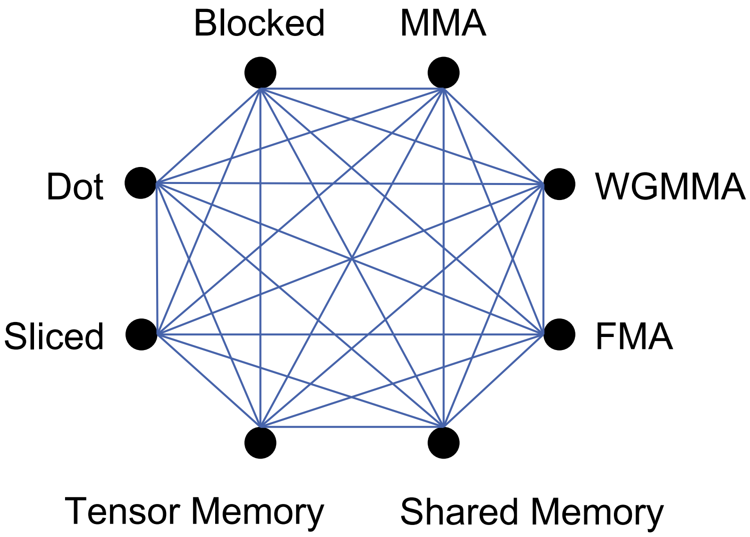


Triton’s Layout Systems



The Cost of Heuristics

- Case by case layout implementations are
  - Fragile
  - Inefficient
  - Fixed
- **~12% of Triton bugs** are layout bugs

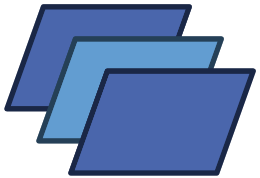


Legacy Triton Layouts

Add a new layout requires **O(N)** conversions

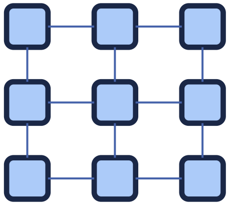
Linear Layouts

Motivation: Indices are Bits



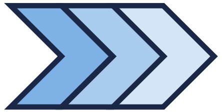
Registers

$r_3 \rightarrow 0b11$



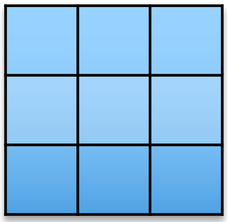
Threads

$t_{19} \rightarrow 0b10011$



Warps

$w_2 \rightarrow 0b10$

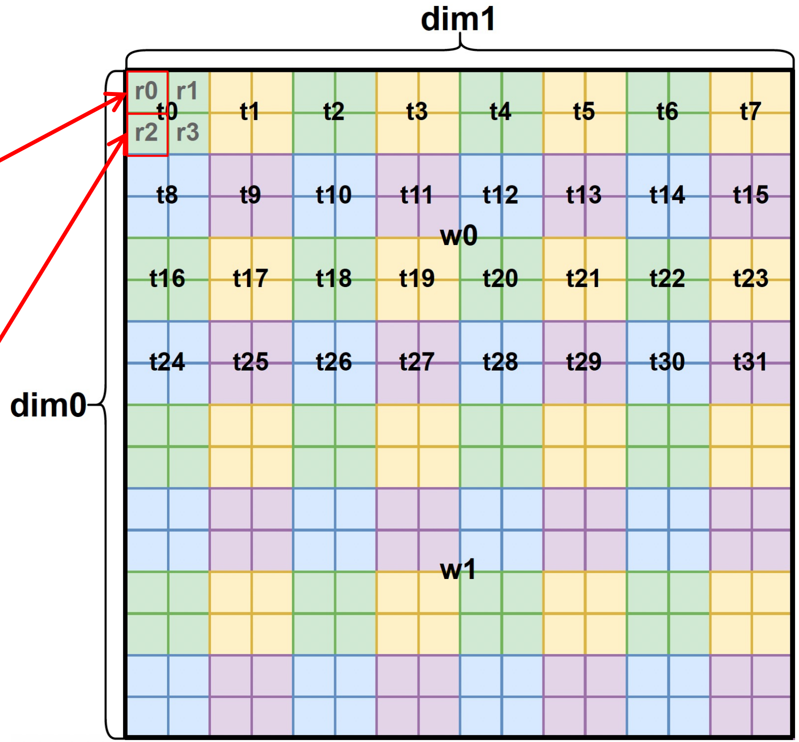


Logical Tensor Locations

$i_2 \rightarrow 0b10$
$j_1 \rightarrow 0b01$

Defining Linear Layouts

Location	Register	Thread	Warp
(0, 0) / (0b00, 0b00)	r_0 / 0b00	t_0 / 0b0000	w_0 / 0b00
(0, 1) / (0b00, 0b01)	r_1 / 0b01	t_0 / 0b0000	w_0 / 0b00
(0, 2) / (0b00, 0b10)	r_0 / 0b00	t_1 / 0b0001	w_0 / 0b00
(0, 3) / (0b00, 0b11)	r_1 / 0b01	t_1 / 0b0001	w_0 / 0b00
…	…	…	
(1, 0) / (0b01, 0b00)	r_2 / 0b10	t_0 / 0b0000	w_0 / 0b00
(1, 1) / (0b01, 0b01)	r_3 / 0b11	t_0 / 0b0000	w_0 / 0b00
…	…	…	
(2, 2) / (0b10, 0b10)	r_0 / 0b00	t_9 / 0b1001	w_0 / 0b00
(2, 3) / (0b10, 0b11)	r_1 / 0b01	t_9 / 0b1001	w_0 / 0b00
…	…	…	
(3, 2) / (0b11, 0b10)	r_2 / 0b10	t_9 / 0b1001	w_0 / 0b00
(3, 3) / (0b11, 0b11)	r_3 / 0b11	t_9 / 0b1001	w_0 / 0b00
…	…	…	



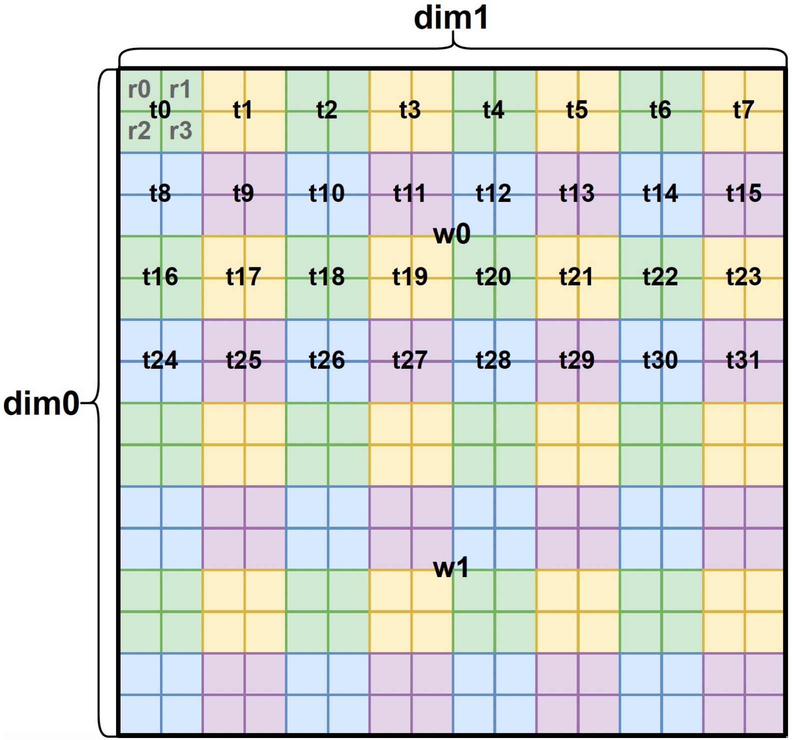
Defining Linear Layouts

$$W = L \times v$$

Hardware bits
(register/thread/warp)

Logical tensor
coordinates
(dim0, dim1)

The layout matrix
(binary transformation)



Defining Linear Layouts

$$W = L \times v$$

$$W = (0,1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$L = \left[\begin{array}{c|cc|ccccc|c} & \multicolumn{2}{c|}{\texttt{Reg}} & \multicolumn{5}{c|}{\texttt{Thr}} & \texttt{Wrp} \\ \hline & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathrm{j} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathrm{i} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

$$v = (2,0,0) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$L: Reg \times Thr \times Wrp \rightarrow F_2^n \times F_2^m$

is a linear map between labeled vector spaces over F_2

F_2 Mathematics

- The field of two elements {0, 1}
- Addition

$$a \oplus b = (a + b)\ mod\ 2 = a\ XOR\ b$$

- Multiplication

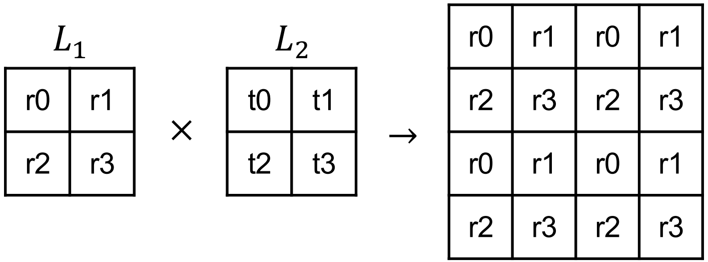
$$a \cdot b = (a \times b)\ mod\ 2 = a\ AND\ b$$

Basic Operators – Composition

- Given vector spaces U, V, and W over F_2 and $L_1: U \rightarrow V$ and $L_2: V \rightarrow W$, we define their composition as $L_2 \circ L_1:\ U \rightarrow W$
- Example
  - $L_1: Reg \rightarrow Memory\ Offset$
  - $L_2: Memory\ Offset \rightarrow Logical\ Coordinate$
  - $L_2 \circ L_1 = L_2(L_1) = Reg \rightarrow Logical\ Coordinate$

Basic Operators – Product

- Given two linear layouts $L_1 : U_1 \to V_1$ and $L_2 : U_2 \to V_2$, we define their product as $L_1 \times L_2 : U_1 \times U_2 \to V_1 \times V_2$, where
  - $U_1 \times U_2 = \{(u_1, u_2) \mid u_1 \in U_1, u_2 \in U_2\}$
  - $V_1 \times V_2 = \{(v_1, v_2) \mid v_1 \in V_1, u_2 \in V_2\}$
- Example
  - $L_1 \colon Reg \to Memory\ Offset$
  - $L_2 \colon Thread \to Memory\ Offset$
  - $L_1 \times L_2 \colon Reg \times Thread \to Memory\ Offset$



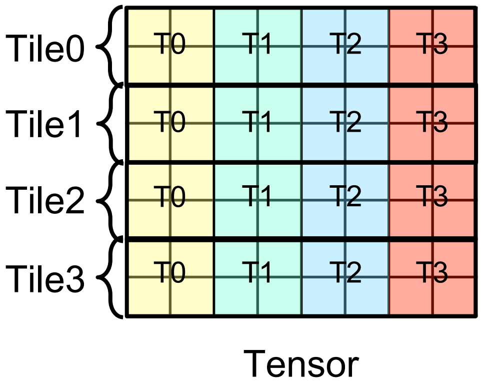
Intuitively, we can consider it as repeating the register/offsets mapping across multiple threads

Basic Operators – Right Inverse

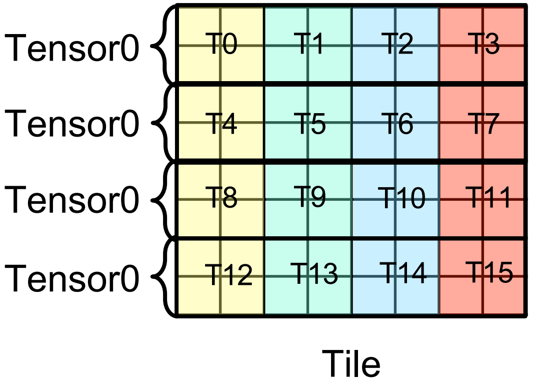
- A surjective linear layout $L: U \longrightarrow V$ over F_2 has a right inverse $L': V \rightarrow U$
  - Surjective: every element in V is the image of at least one element from U
  - Injective: No two elements in U map to the same element in V
- Let L be a $m \times n$ matrix M, we define M as the solution obtained by Gaussian elimination of $MX = I_m$, where I_m is an identity matrix
- Example
  - L_1: $Reg \rightarrow Memory\ Offset$
  - L_1^{-1}: $Memory\ Offset \rightarrow Reg$

Code Generation

Broadcasting – Background



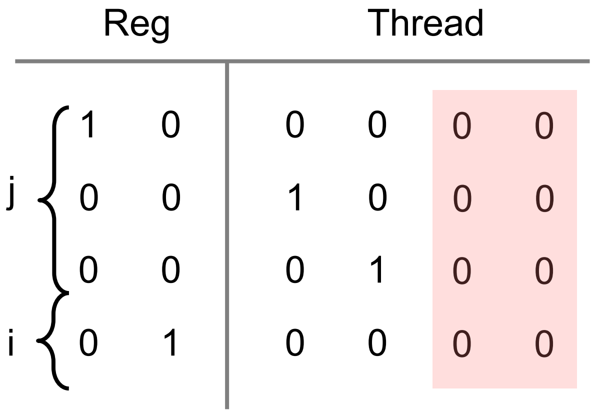
Tile < Tensor

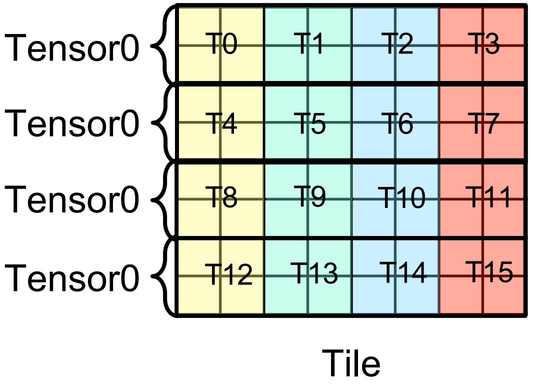


Tensor < Tile

Broadcasting – Linear Layouts

- The same values are broadcasted every four threads





Broadcasted threads, e.g., $L\times(r_1, t_5) = L\times(r_1, t_1)$

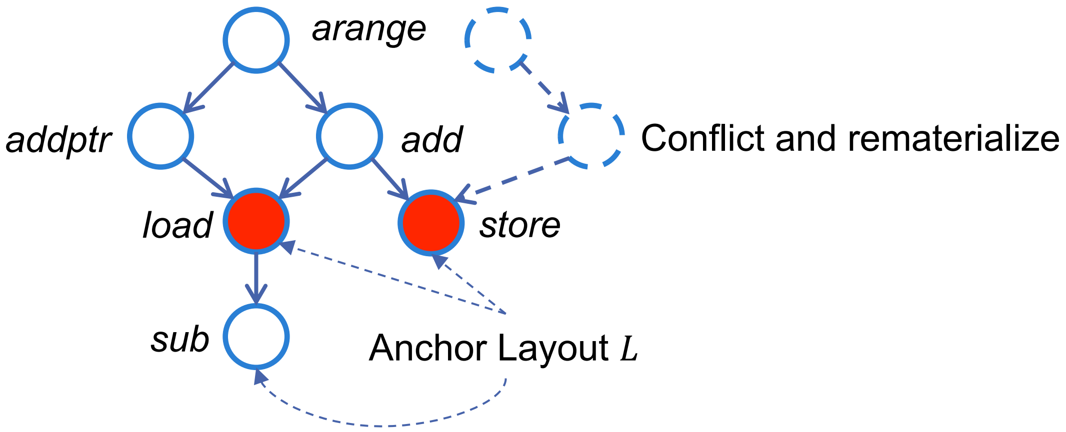
Tensor < Tile

Layout Conversions – Background

- Given distributed layouts L_A and L_B, we can convert the tensor/hardware resource mapping from L_A to L_B by $L_B^{-1} \circ L_A$
  - $L_A\colon Reg_A \times Thread_A \times Warp_A \rightarrow Coordinate$
  - $L_B\colon Reg_B \times Thread_B \times Warp_B \rightarrow Coordinate$
  - $L_B^{-1} \circ L_A = L_B^{-1}(L_A) = Reg_A \times Thread_A \times Warp_A \rightarrow Reg_B \times Thread_B \times Warp_B$

Layout Conversions – Background

- First determine an anchor: layout that is optimized by heuristics
- Forward propagate the layout to uses
- Backward propagate the layout to defs and rematerialize to resolve conflicts



Layout Conversions – Intra Thread

- If $L_A^{Thr} = L_B^{Thr}$ and $L_A^{Wrp} = L_B^{Wrp}$, $(L_B^{-1} \circ L_A)^{Thr}$ and $(L_B^{-1} \circ L_A)^{Wrp}$ are identity
- We can interchange data elements within each thread by register permutation

	Reg$_A$		Thread$_A$			
Reg$_B$	0	1	0	0	0	0
	1	0	0	0	0	0
Thread$_B$	0	0	1	0	0	0
	0	0	0	1	0	0
	0	0	0	0	1	0
	0	0	0	0	0	1
	4 registers		16 threads			

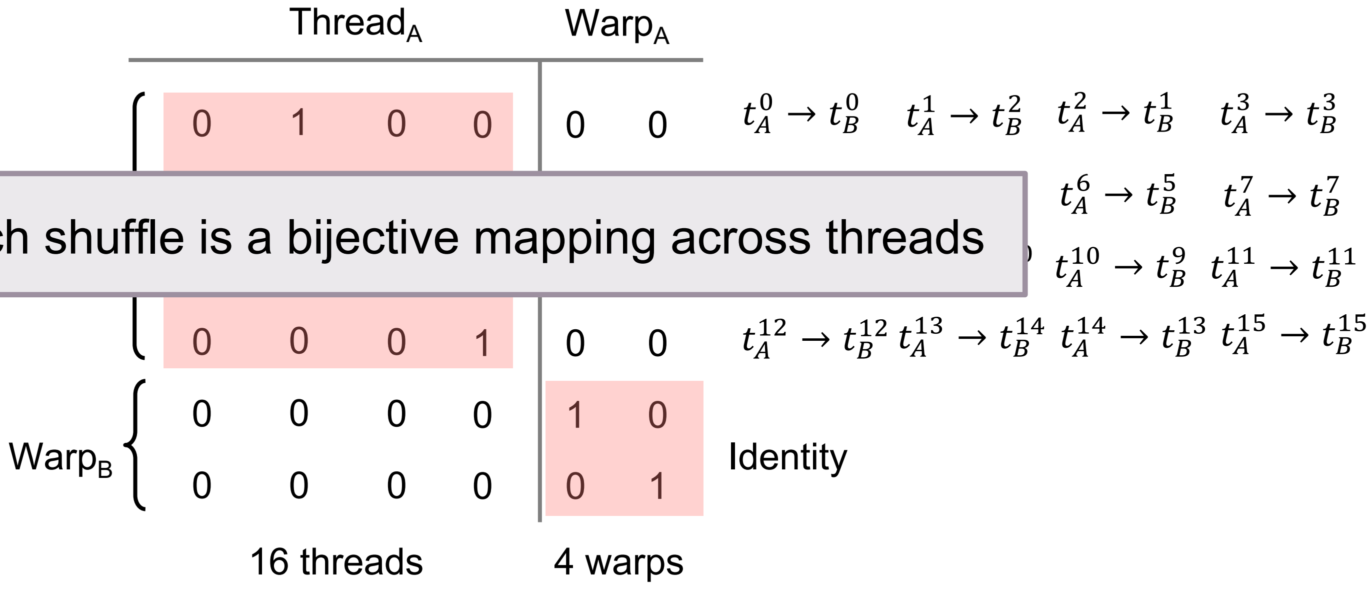
$r_A^0 \to r_B^0 \quad r_A^2 \to r_B^1$

$r_A^1 \to r_B^2 \quad r_A^3 \to r_B^3$

Identity

Layout Conversions – Intra Warp

- If $L_A^{Wrp} = L_B^{Wrp}$, $(L_B^{-1} \circ L_A)^{Wrp}$ is identity
- We can interchange data elements within each warp by shuffles

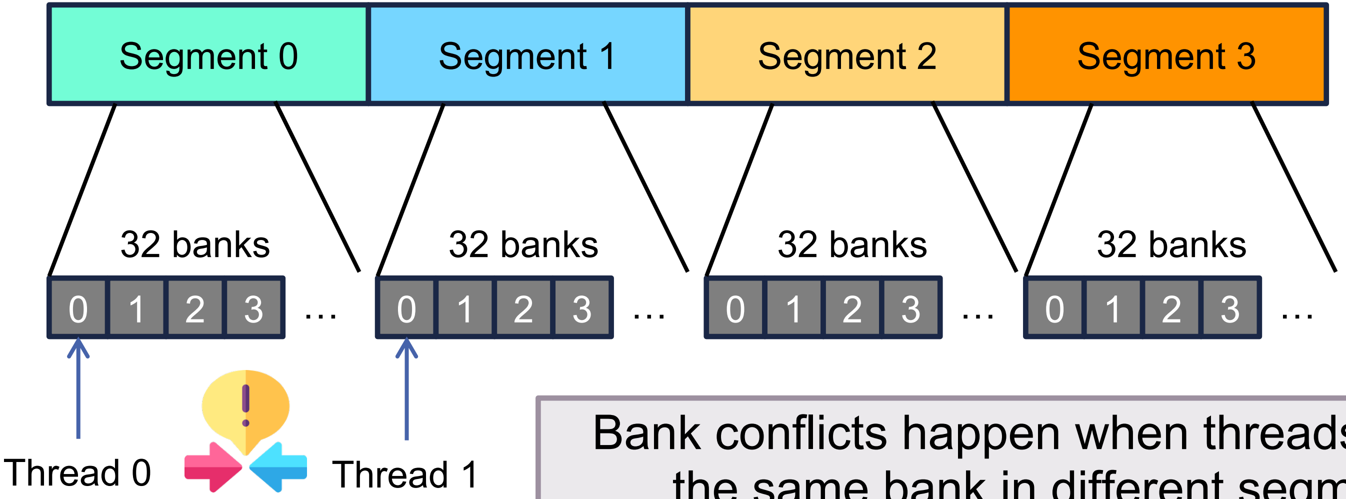


Layout Conversions – Intra CTA

- Both $(L_B^{-1} \circ L_A)^{Thr}$ and $(L_B^{-1} \circ L_A)^{Wrp}$ are not identity
- Construct a new memory layout $L_S: Coordiniate \rightarrow Memory\ Offset$
- Construct new layouts $L_{AS} = L_S \circ L_A$, $L_{BS} = L_B^{-1} \circ L_S^{-1}$
  - $L_{AS}: Reg_A \times Thread_A \times Warp_A \rightarrow Memory\ Offset$
  - $L_{BS}: Memory\ Offset \rightarrow Reg_B \times Thread_B \times Warp_B$
- Store the tensor with L_{AS} on the shared memory
- Load the tensor into L_{BS} from the shared memory
- The most general layout conversion mechanism

Layout Conversions – Bank Conflicts

- The most general layout conversion mechanism is using shared memory
- Shared memory is divided into segments
- Each segment consists of multiple banks

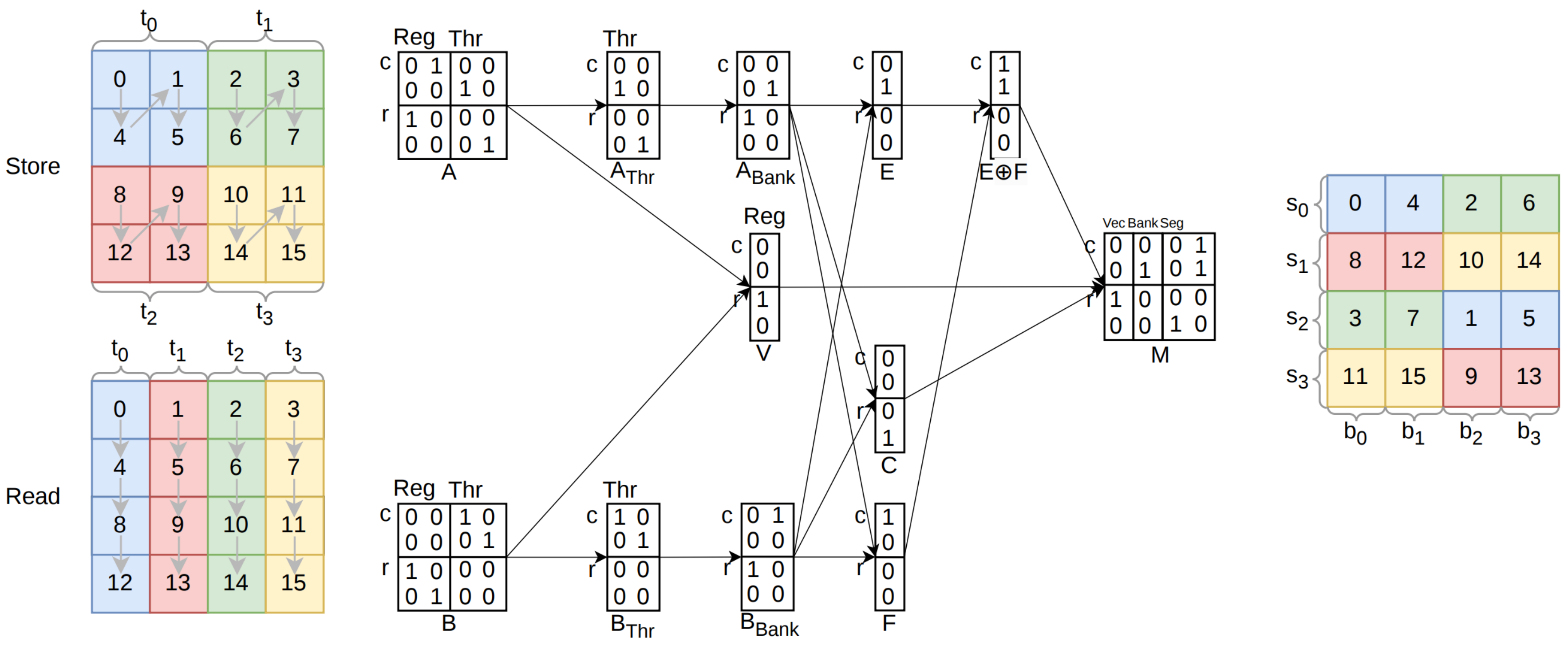


Bank conflicts happen when threads access the same bank in different segments

Layout Conversions – Avoid Bank Conflicts

- Model memory offsets as $Vec \times Bank \times Segment$
- Intuition1: threads access the same segment do not conflict
- Intuition2: we can find the largest vector and bank subspaces covering all banks used by L_A and L_B
- Intuition3: the bases in the segment subspace should not overlap with bases in the bank subspace, otherwise we may access the same bank but different segments

Layout Conversions - Example



Please refer to Figure 5 in the paper

Other Use Cases

- Software emulation of MXFP types
  - Linear layouts-based layout conversion
- Tile-based load/store (e.g., *ldmatrix*/*stmatrix*)
  - Linear layout division operations
- Contiguous elements
  - Largest u such that $L_{reg}^{-1}(i) = i$ for any $i \leq u$
- Generalized vectorization
  - Permute register values to get a larger number of contiguous elements

Experiments

Setup

- Compare legacy **Triton** *vs* **Triton-Linear**
- Microbenchmarks show benefits of linear layout in specific codegen path
- Real benchmarks measure the overall speedups

Platform	GPU Model	Memory	Notes
RTX4090	NVIDIA RTX4090	24GB GDDR6X	Consumer GPU
GH200	NVIDIA GH200	80GB HBM2e	Data center GPU
MI250	AMD MI250	64GB HBM2	Data center GPU

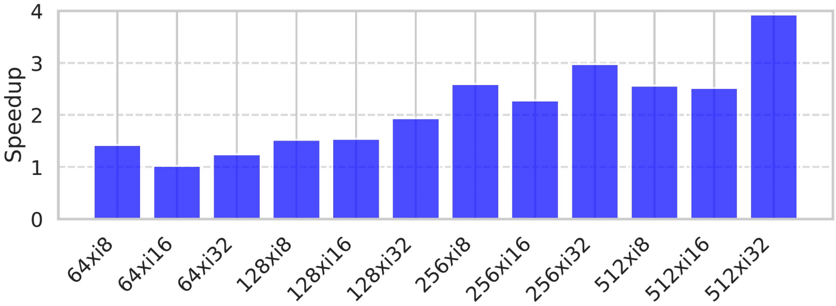
Case 1: Mixed Precision Pass Rate

- Using linear layout's automatic code generation instead of heuristic-based code conversions

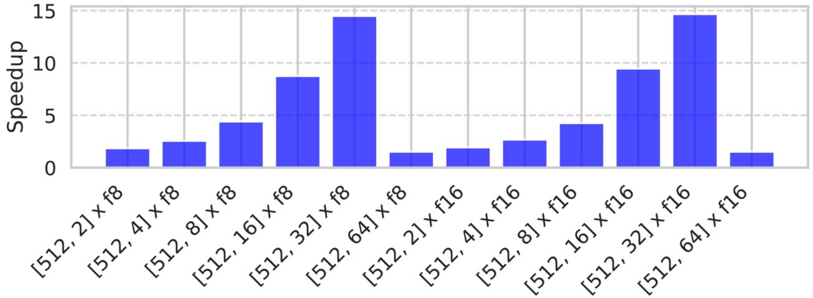
Data Type	Triton	Triton-Linear	Data Type	Triton	Triton-Linear
i16/f16	32/64	64/64	*i16/f32*	32/32	32/32
i16/f64	32/32	32/32	*i16/f8*	36/96	96/96
i32/f16	32/32	32/32	*i32/f64*	16/32	32/32
i32/f8	18/48	48/48	*i64/f16*	32/32	32/32
i64/f32	16/32	32/32	*i64/f8*	18/48	48/48
i8/f16	36/96	96/96	*i8/f32*	18/48	48/48
i8/f64	18/48	48/48	*i8/f8*	30/144	144/144

Case 2: Layout Conversion and Gather Speedups

- Using warp shuffle to communicate within warps without going through shared memory



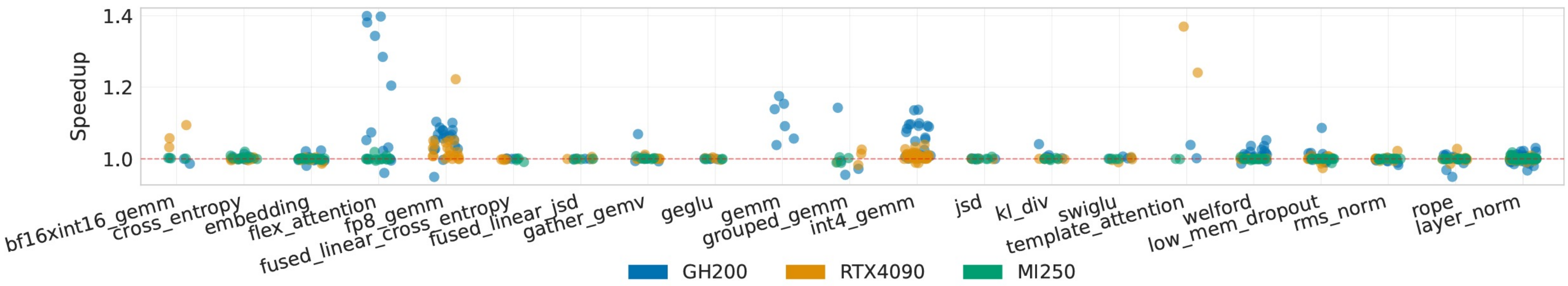
Warp Shuffle



Convert Layout

Real Benchmarks

- 21 benchmarks and 265 cases in TritonBench
  - GH200: up to 1.40x speedup
  - RTX4090: up to 1.37x speedup
  - MI250x: up to 1.03x speedup



Conclusions

More Readings

- Lei’s blog posts
  - https://www.lei.chat/posts/triton-linear-layout-concept/
  - https://www.lei.chat/posts/triton-bespoke-layouts/
- Justin’s blog post
  - https://jlebar.com
- Paper
  - https://arxiv.org/abs/2505.23819
  - Figure 4 and Figure 5 have been fixed in the Arxiv version
  - Unfortunately it’s not easy to update the paper in ACM DL

Thaihoa's Layout Visualizer

- https://deep-learning-profiling-tools.github.io/linear-layout-viz/

Code

- Triton bespoke layout definition
  - https://github.com/triton-lang/triton/blob/main/include/triton/Dialect/TritonGPU/IR/TritonGPUAttrDefs.td
- Linear layout operations
  - https://github.com/triton-lang/triton/blob/main/include/triton/Tools/LinearLayout.h
- Linear layout Python interface
  - https://github.com/triton-lang/triton/blob/main/python/src/linear_layout.cc

Comparison with CuTe

Aspect	Linear Layouts	CuTe
Goal	Integrated into a compiler framework	Manual layout description
Math model	Linear algebra over F2	Category theory
Layout conversion	Automatically generated	No general mechanism
Swizzling	Inherently defined layout formulation	A separate step
Dimension semantics	Labeled (e.g., Reg, Thr, Wrp)	Unlabeled

Takeaways

- Linear layouts bridges complex hardware components and logical tensors through theoretical foundation and implementation
- Family of linear layouts facilitates robust code generation and enables key optimizations
- The primary limitation of linear layouts is the restriction to power-of-two shapes
    - But can be mitigated by defining larger tensors and masking out-of-boundary elements

Related Work

- A. Edelman, S. Heller, and S. Lennart Johnsson. Index transformation algorithms in a linear algebra framework. IEEE Transactions on Parallel and Distributed Systems, 5(12):1302–1309, 1994. doi:10.1109/71.334903.
- T.H. Cormen. Fast permuting on disk arrays. Journal of Parallel and Distributed Computing, 17(1):41–57, 1993. doi:10.1006/jpdc.1993.1004.
- Mathis Bouverot-Dupuis and Mary Sheeran. Efficient gpu implementation of affine index permutations on arrays. In Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing, FHPNC 2023, page 15â˘ A¸S28, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3609024.3609411.