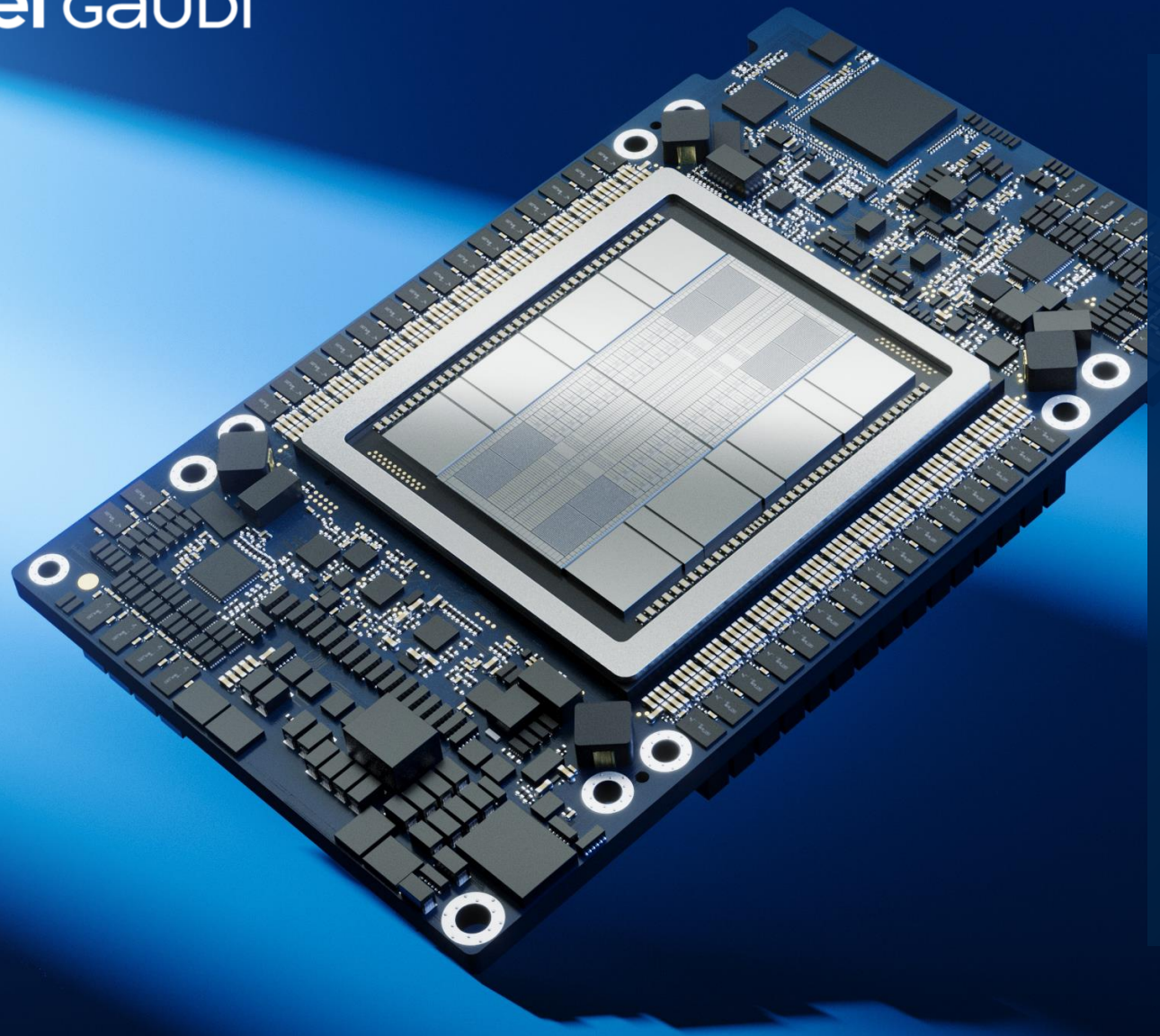
Core C++ 2024

Speeding Up Intel Gaudi Deep-Learning Accelerators Using an MLIR-Based Compiler

Dafna Mordechai, Omer Paparo Bivas

intel GAUDI



Speeding up Intel® Gaudi® deep-learning accelerators using an MLIR-based compiler

Dafna Mordechai, Omer Paparo Bivas, Jayaram Bobba, Sergei Grechanik, Tzachi Cohen, Dibyendu Das

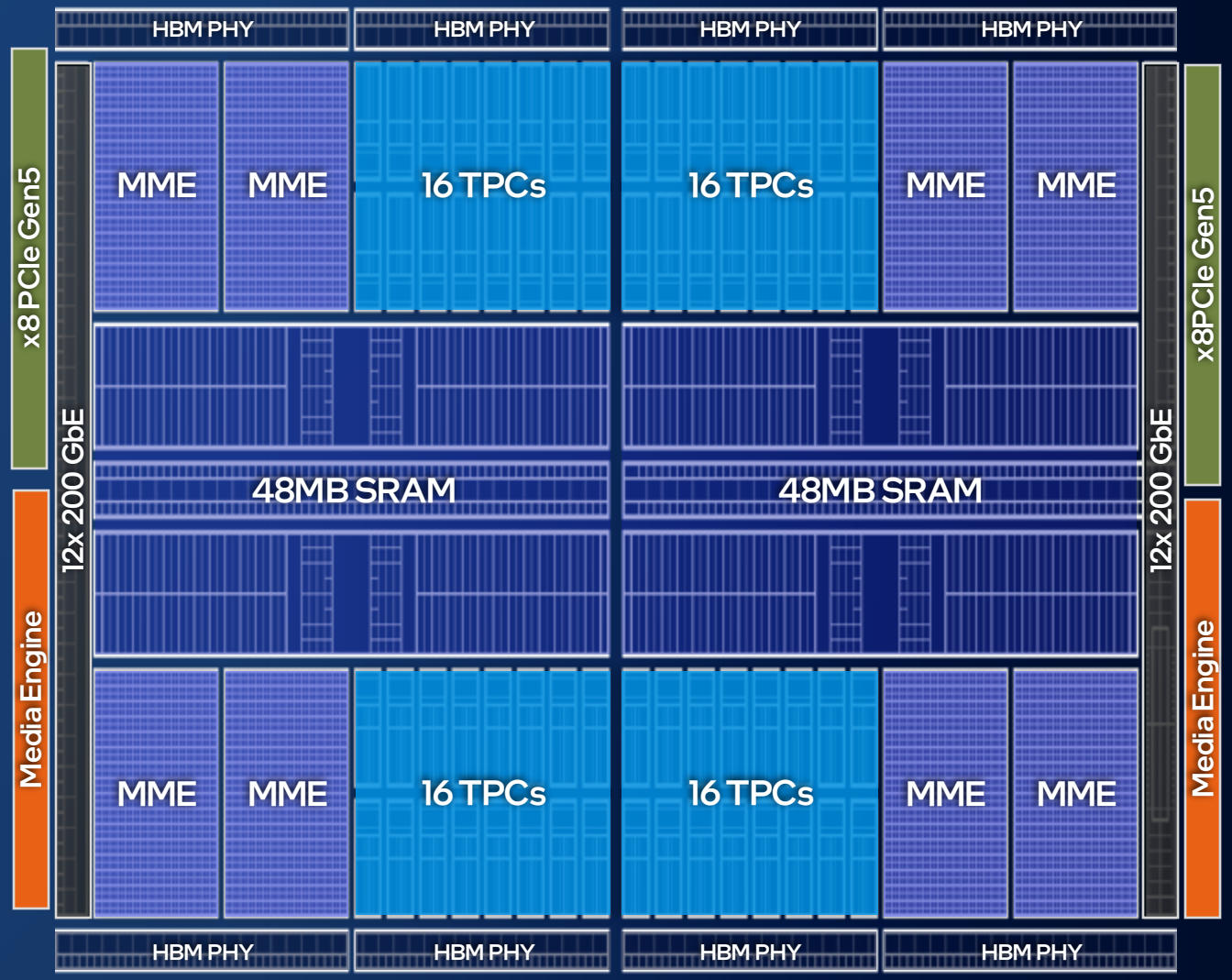
Intel/Habana Labs

Agenda

- Deep Learning Compilers: Transforming a Large Computational Graph into an optimized Execution Plan
- The TPC Fuser: A JIT compiler for deep learning kernels that delivers significant performance improvements
- Case Study: Adjusting the TPC-Fuser to LLMs Recent Challenges

Intel Gaudi 3 AI accelerator Spec and Block Diagram

Feature/Product	Intel® Gaudi® 3 Accelerator
BF16 Matrix TFLOPs	1835
FP8 Matrix TFLOPs	1835
BF16 Vector TFLOPs	28.7
MME Units	8
TPC Units	64
HBM Capacity	128 GB
HBM Bandwidth	3.67 TB/s
On-die SRAM Capacity	96 MB
On-die SRAM Bandwidth RD+WR (L2 Cache)	19.2 TB/s
Networking	1200 GB/s bidirectional
Host Interface	PCIe Gen5 x16
Host Interface Peak BW	128 GB/s bidirectional
Media Engine	Rotator + 14 Decoders (HEVC, H.264, JPEG, VP9)



MME - Matrix Multiplication Engine

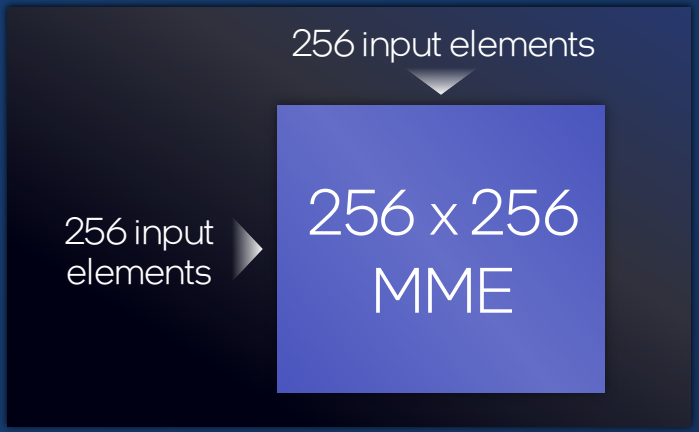
Configurable, not programmable

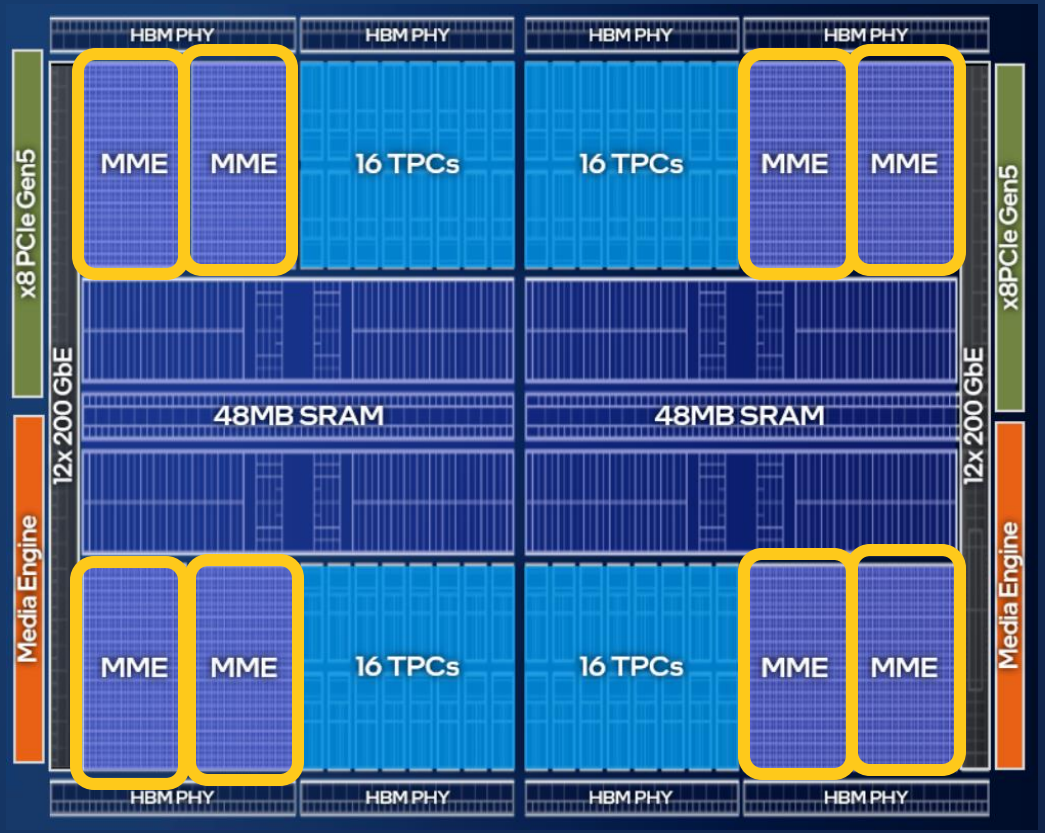
Each MME is a large output stationary systolic array

- 256x256 MAC structure w/ FP32 accumulators
- 64k MACs/cycle for BF16 and FP8

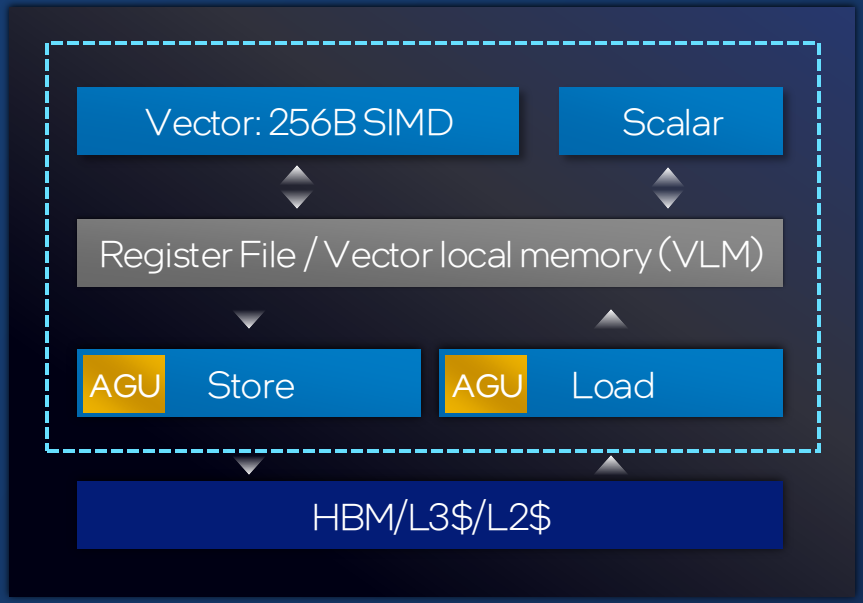
Large systolic array reduces intra-chip data movement, increasing efficiency

Internal pipeline to maximize compute throughput





Tensor Processor Core (TPC): 256B-wide SIMD Vector Processor



Fully Programmable using the TPC-C
C enhanced with TPC intrinsics

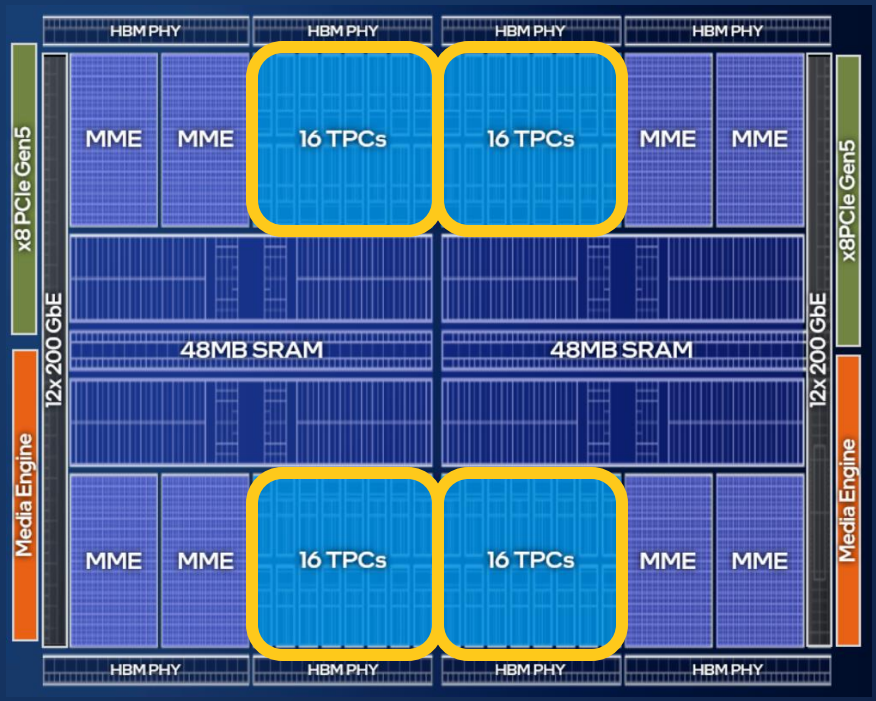
VLIW with 4 separate pipeline slots:
Vector, Scalar, Load & Store

Integrated Address Generation Unit for HW-accelerated address generation

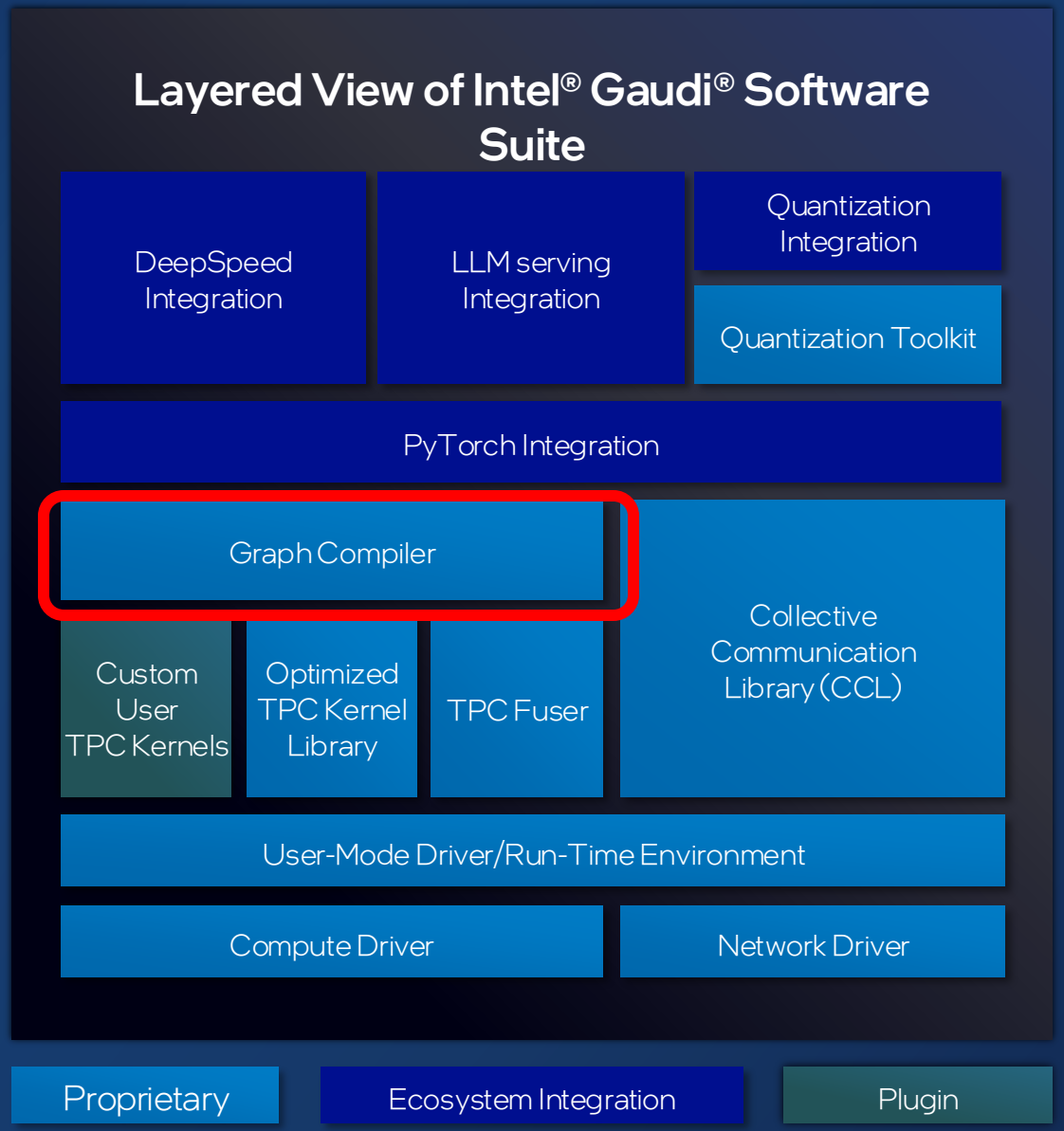
---

Supports main 1/2/4-Byte datatypes: Floating Point and Integer

---



Intel Gaudi Software Suite

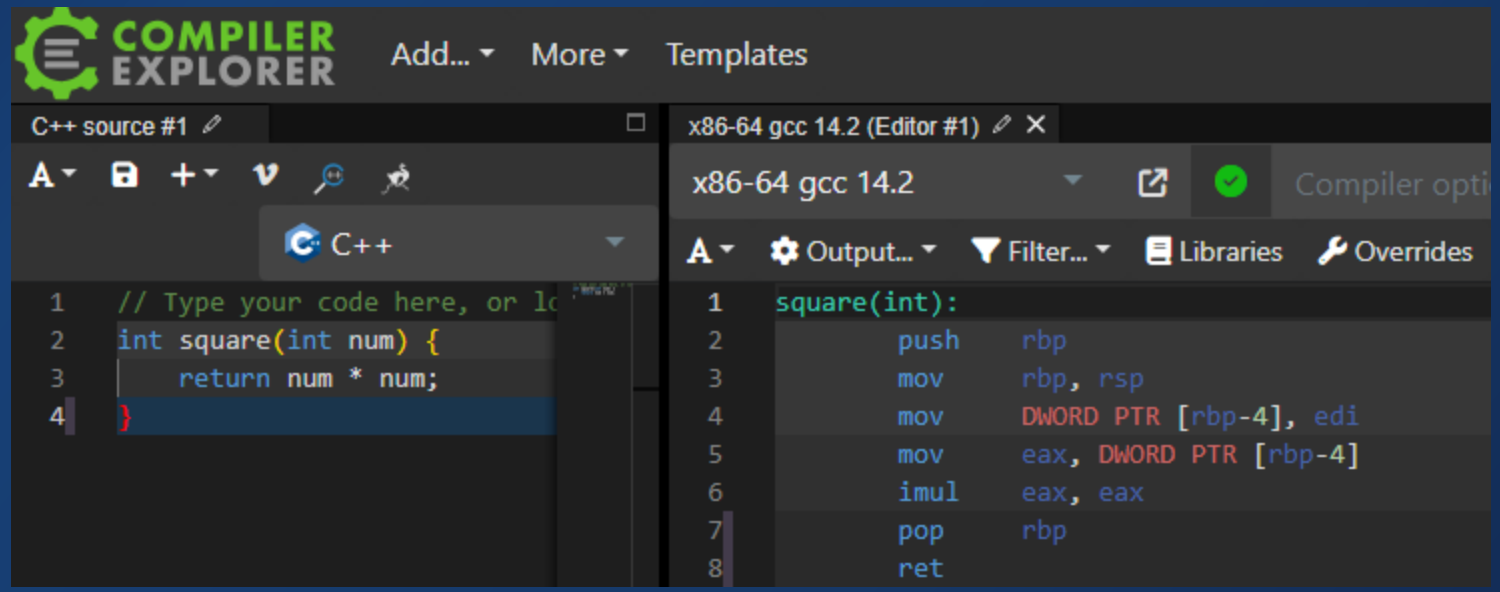


Compilers 101

Compiler(Source Code) --> Machine Code

[semantic preserving]

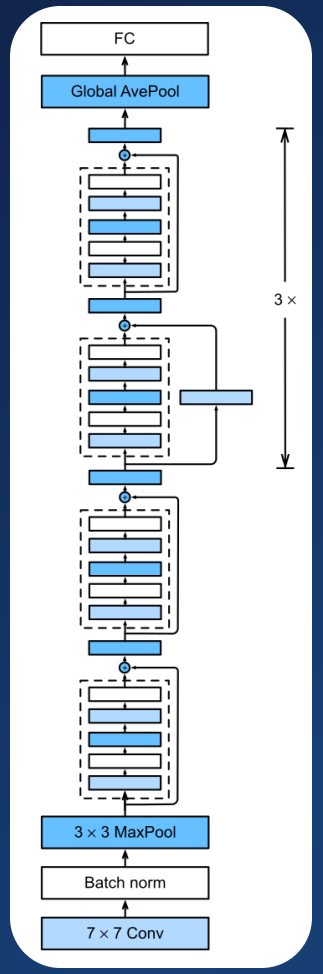
Compiler == (Translations + Transformations)



LLVM - Low Level Virtual Machine

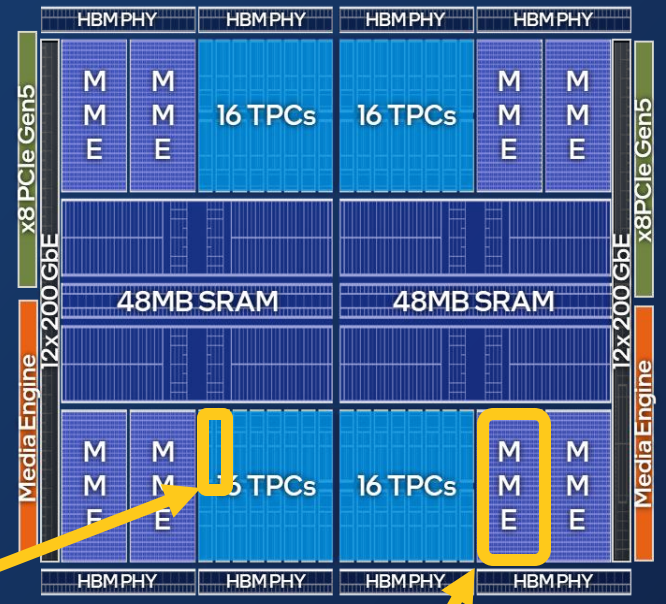
Clang - Front-end for languages in the C language family (C, C++, Objective C/C++, OpenCL, and CUDA)

Deep Learning Compilation 101



The original Resnet-18 architecture. Up to 152 layers were trained in the original publication (as "ResNet-152")
- Wikipedia

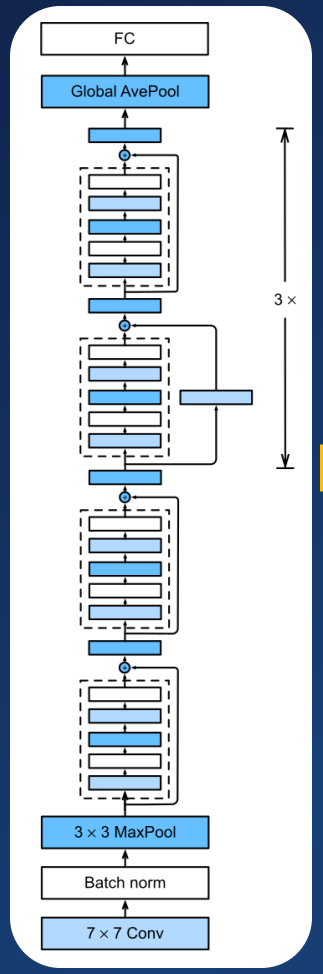
Graph Compiler

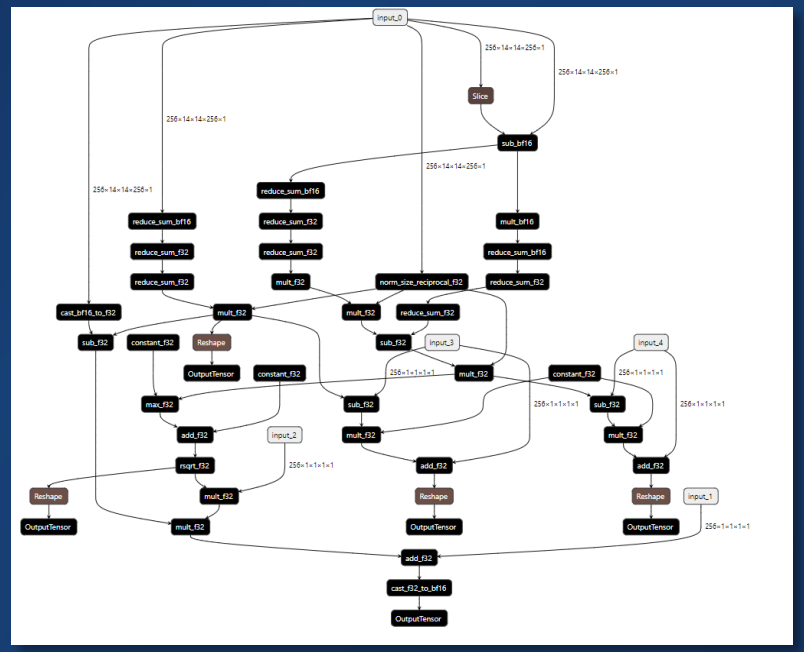


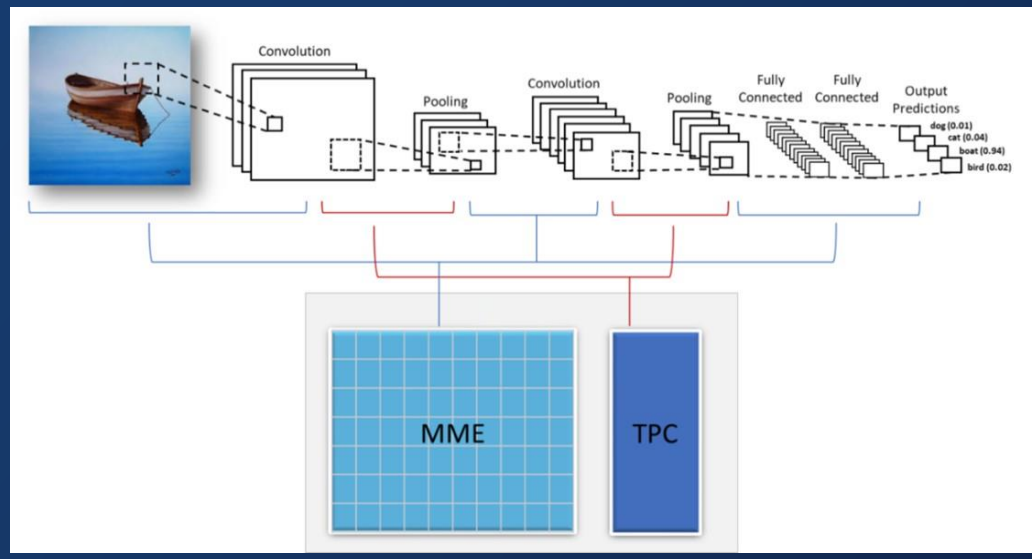
TPC Kernels
For example: tanh_f32

Configuration

Graph Compilation Flow: Transforming a Deep Learning Computational Graph into an Intel-Gaudi Execution Plan







Neural Network Hardware Mapping – Use of MME and TPC

The original Resnet-18 architecture. Up to 152 layers were trained in the original publication (as "ResNet-152")
- Wikipedia

TPC Kernel Library Providers

Pre-Compiled Library (TPC-C)

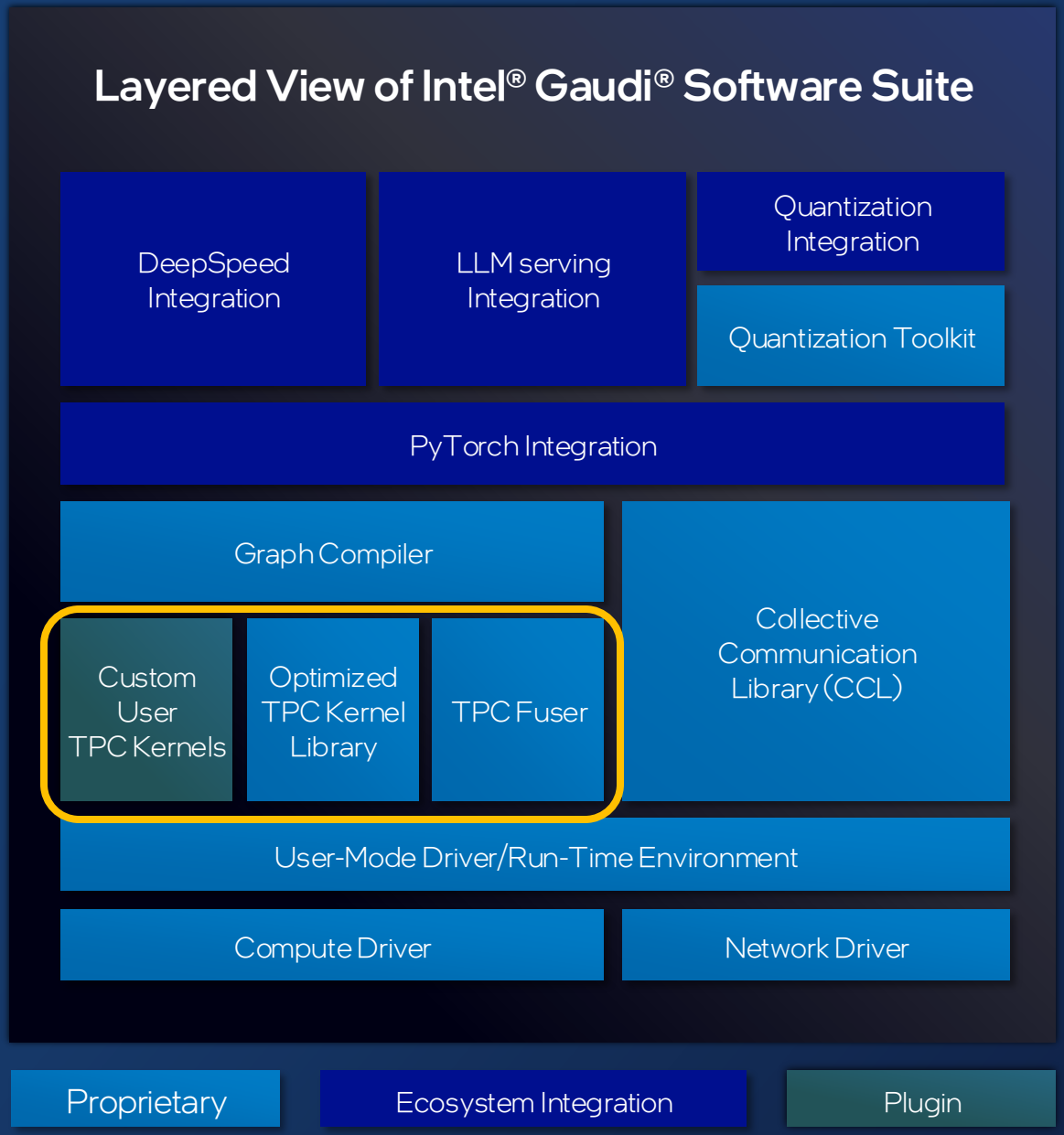
Intel-Gaudi optimized TPC kernel library

Custom user kernels

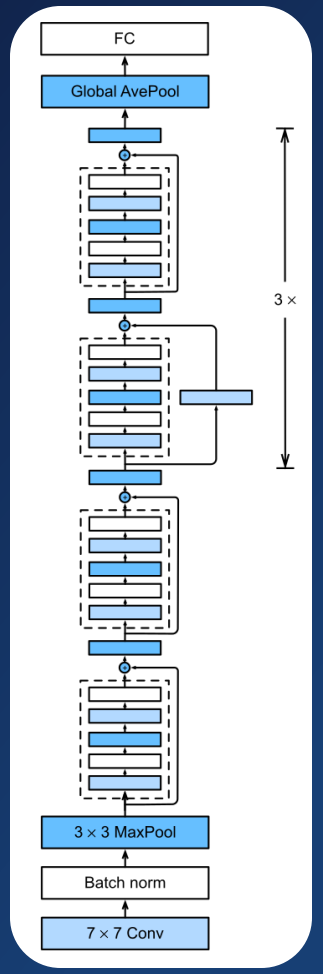
JIT Library

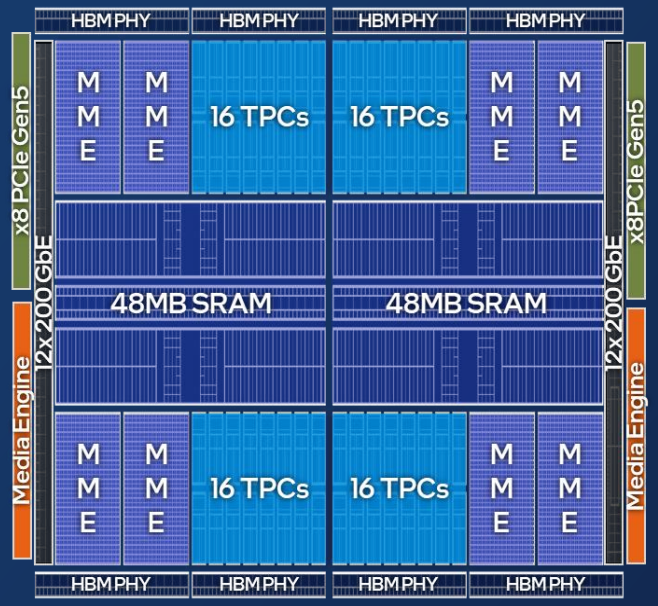
Auto-generated fused kernels, compiled during graph compilation, using the MLIR-based JIT compiler

All the kernels are compiled using the Clang-based TPC Compiler



Graph Compilation Flow

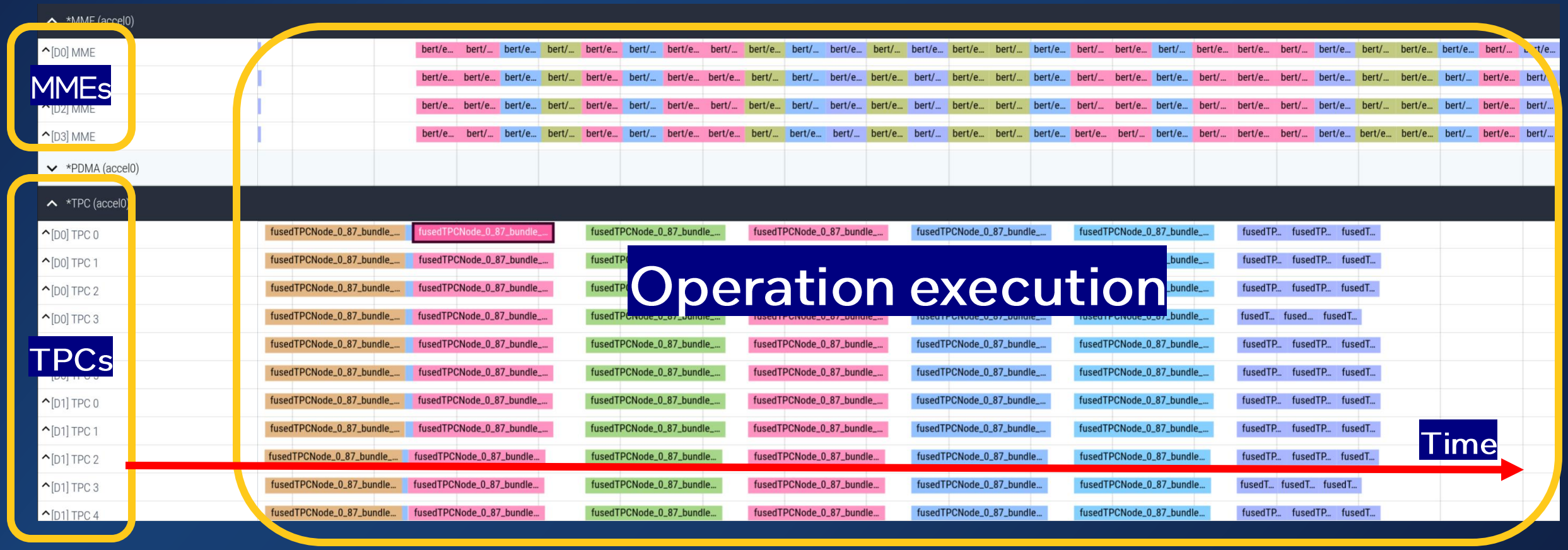




- Processes the deep learning topology to allocate operations across MME, TPC, and DMA engines
  - Generate MME Configurations
  - Select kernels from the different kernel library providers
- Optimizes the computational graph using techniques similar to traditional compilers
- Schedules operations while accounting for memory constraints and dependencies
- Configures hardware registers and system settings based on the execution plane (recipe)

The original Resnet-18 architecture. Up to 152 layers were trained in the original publication (as "ResNet-152")
- Wikipedia

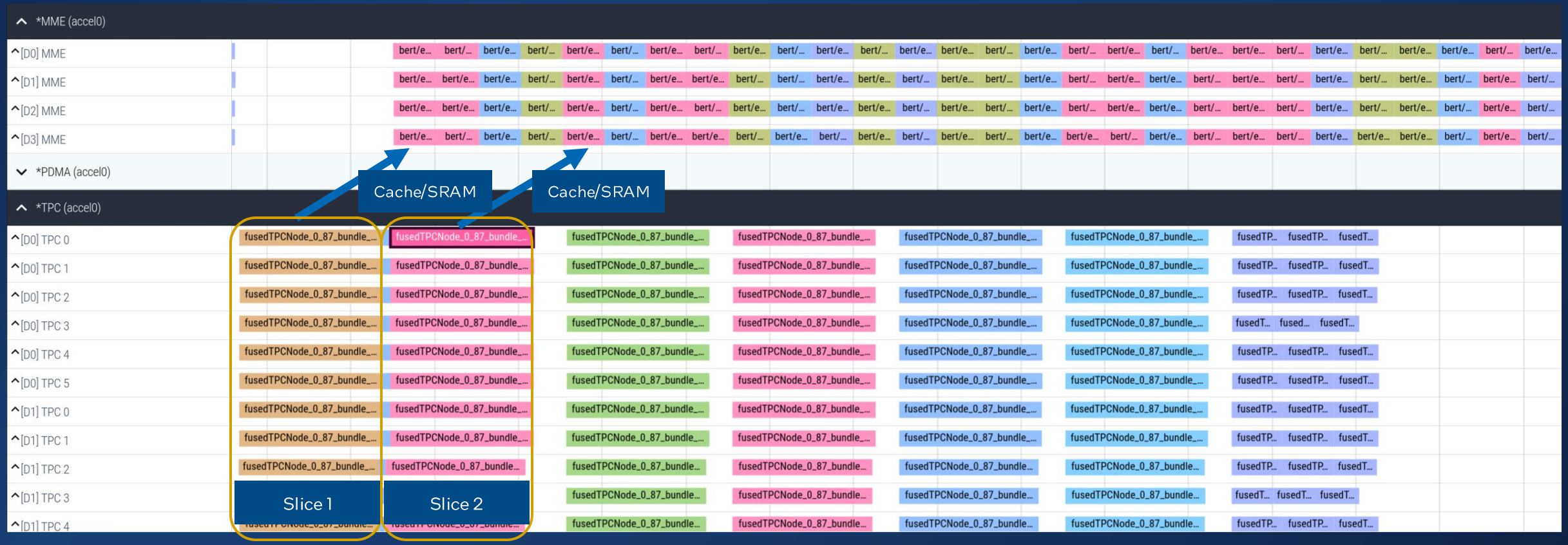
What's an Ideal Execution?



Graph Compiler: Slicing + Bundling

The Graph Compiler is designed to partition data and group operations efficiently to achieve overlapping execution between the MME and TPC units.

This optimization maximizes the use of SRAM and local caches, enhancing data transfer efficiency.

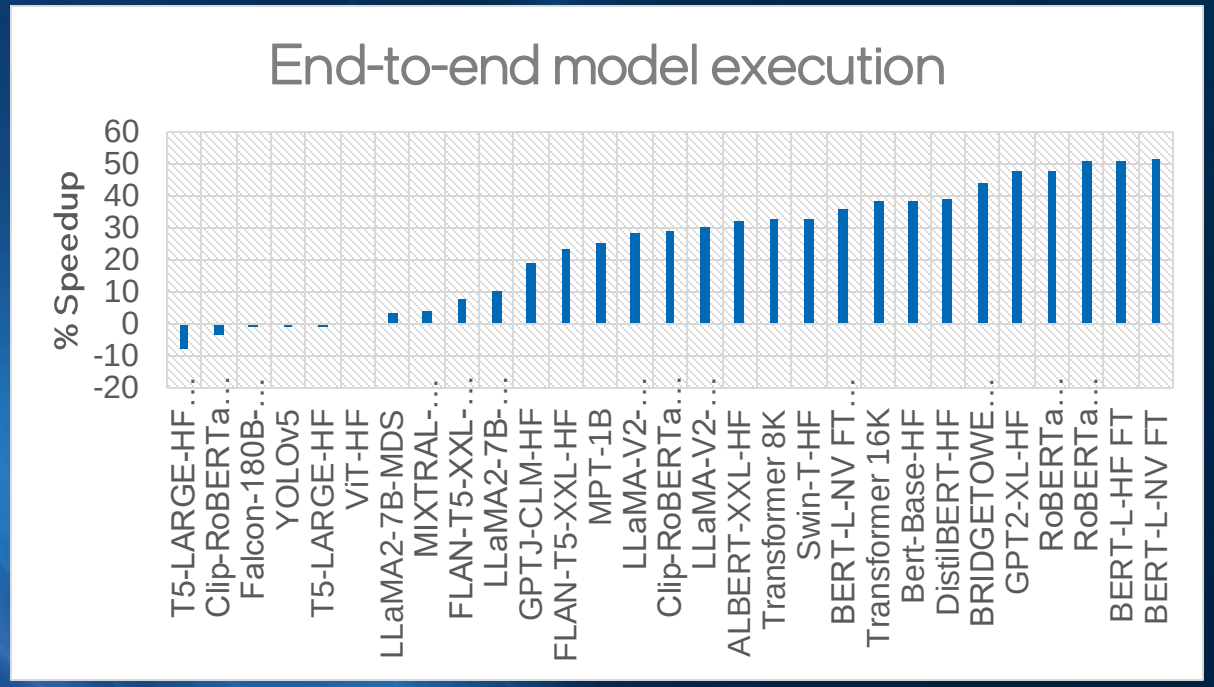


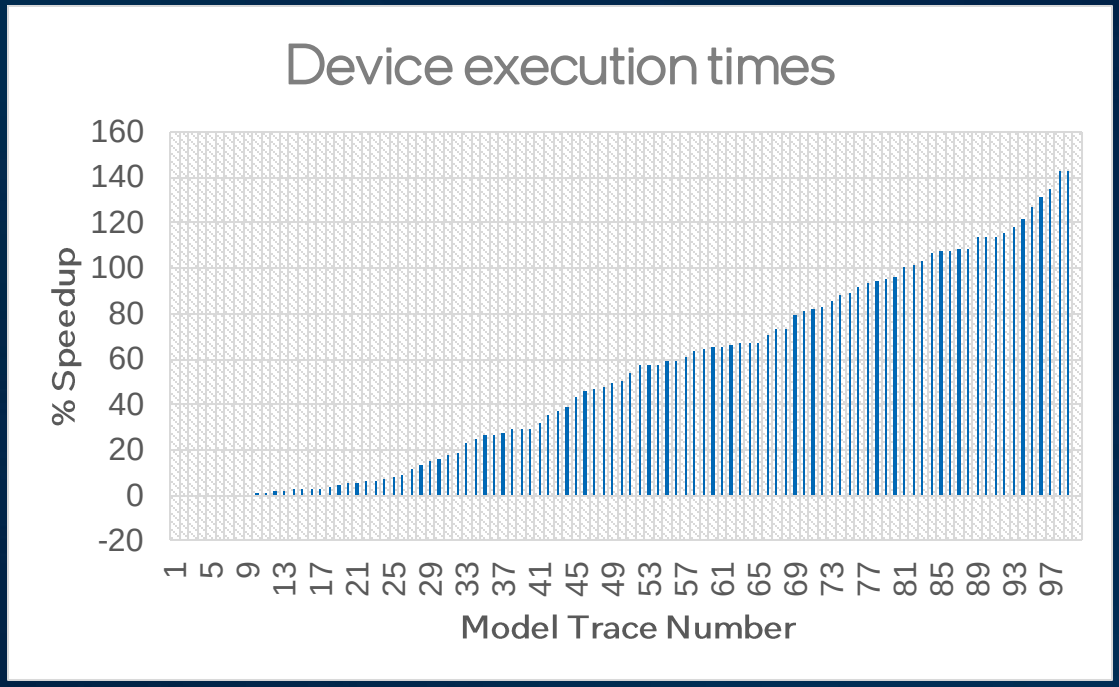
JIT Compiler: Key Benefits and Advantages

- Spare HBM bandwidth
- Enhances caches and local memory efficiency
- Spare kernel-to-kernel invocation latency
- Applies shape-based optimizations

The TPC Fuser Supports **element-wise operations**, **reductions** and **normalizations**

TPC-Fuser Performance Improvements





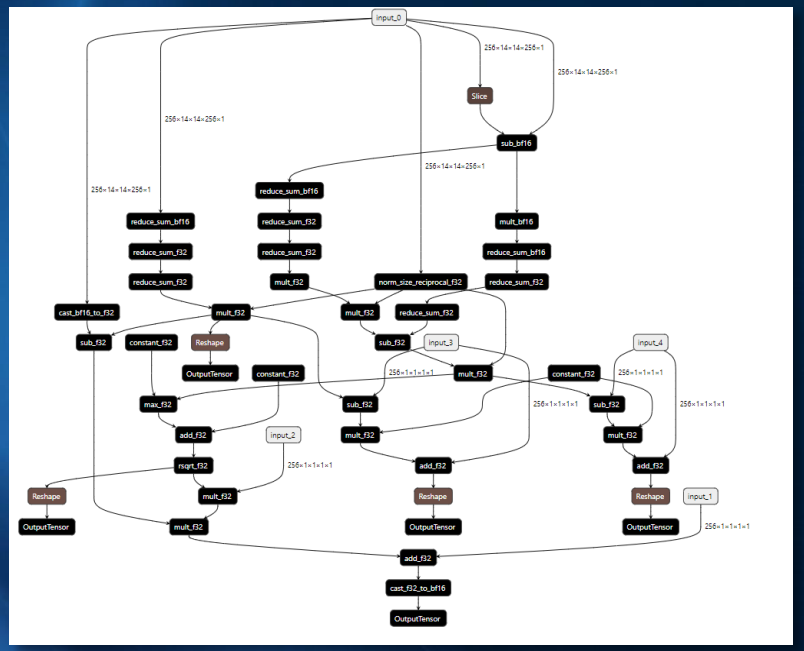
1.3X Avg Perf Improvement at model level

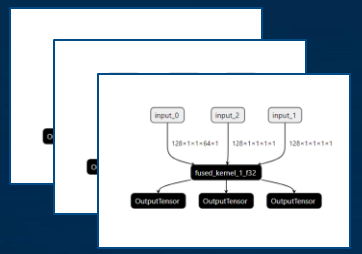
1.5X Avg Perf Improvement in device execution time

Measured on Intel Gaudi2

How Is It Done?

Clustering + Codegen





Let's Take an Example:

```
module {
  syn_rt.graph(%in0: !syn_rt.tensor<4x2x128xf32>, %in1: !syn_rt.tensor<4x2x128xf32>) -> !syn_rt.tensor<4x2x128xf32>{
    %out0 = tpckernel.add %in0, %in1 : (!syn_rt.tensor<4x2x128xf32>, !syn_rt.tensor<4x2x128xf32>) -> <f32> -> (!syn_rt.tensor<4x2x128xf32>)
    %out0_0 = tpckernel.sin %out0 : (!syn_rt.tensor<4x2x128xf32>) -> <f32> -> (!syn_rt.tensor<4x2x128xf32>)
    syn_rt.exit %out0_0 : !syn_rt.tensor<4x2x128xf32>
  }
}
```

```
module {
  func.func @fused_kernel_0xBEF27837_1_f32(%arg0: tensor<4x2x128xf32>, %arg1: tensor<4x2x128xf32>) -> tensor<4x2x128xf32> attributes {syn.expected_arch = "
    %0 = syn.add %arg0, %arg1 : tensor<4x2x128xf32>
    %1 = syn.sin %0 : tensor<4x2x128xf32>
    return %1 : tensor<4x2x128xf32>
  }
  syn_rt.graph(%in0: !syn_rt.tensor<4x2x128xf32>, %in1: !syn_rt.tensor<4x2x128xf32>) -> !syn_rt.tensor<4x2x128xf32>{
    %0 = syn_rt.launch_tpc_func @fused_kernel_0xBEF27837_1_f32(%in0, %in1)  : (!syn_rt.tensor<4x2x128xf32>, !syn_rt.tensor<4x2x128xf32>) -> !syn_rt.tensor<
    syn_rt.exit %0 : !syn_rt.tensor<4x2x128xf32>
  }
}
```

Loop Fusion

The compiler merges multiple loops with the same iteration range into a single loop

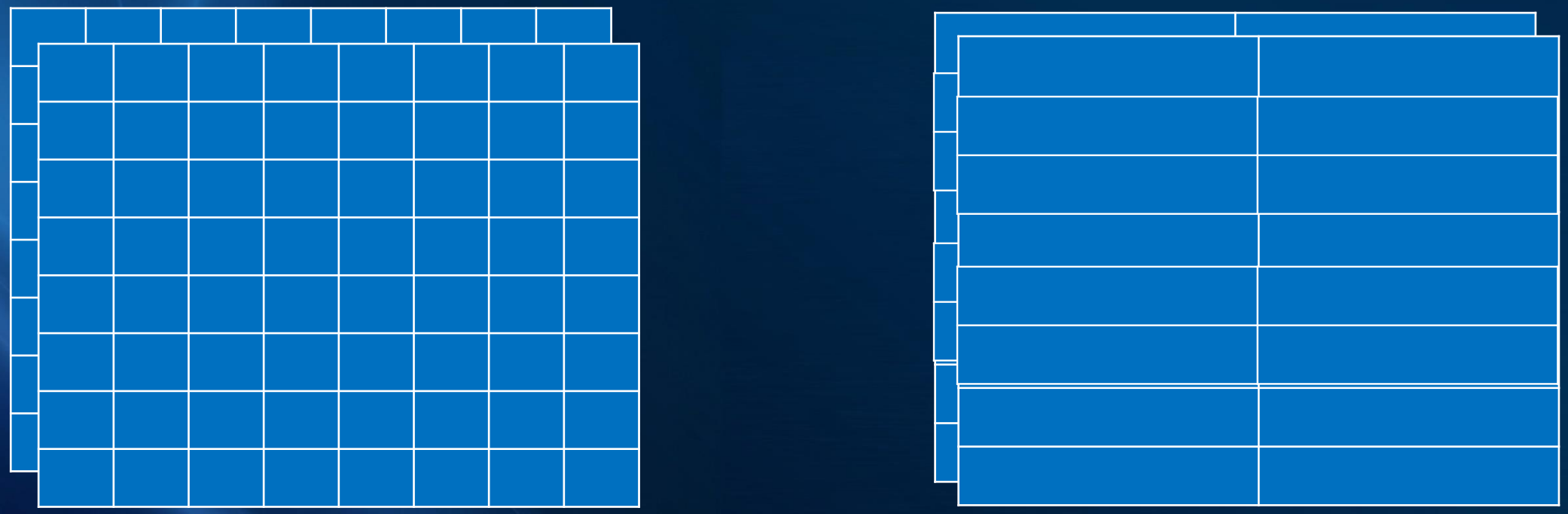
Loop Fusion

```
func.func @fused_kernel_0xBEF27837_1_f32(%arg0: memref<4x2x128xf32, 1>, %arg1: memr
  affine.for %arg3 = 0 to 128 {
    affine.for %arg4 = 0 to 2 {
      affine.for %arg5 = 0 to 4 {
        %0 = affine.load %arg0[%arg5, %arg4, %arg3] : memref<4x2x128xf32, 1>
        %1 = affine.load %arg1[%arg5, %arg4, %arg3] : memref<4x2x128xf32, 1>
        %2 = arith.addf %0, %1 : f32
        affine.store %2, %add_res[%arg5, %arg4, %arg3] : memref<4x2x128xf32, 1>
      }
    }
  }

  affine.for %arg6 = 0 to 128 {
    affine.for %arg7 = 0 to 2 {
      affine.for %arg8 = 0 to 4 {
        %3 = affine.load %add_res[%arg8, %arg7, %arg6] : memref<4x2x128xf32, 1>
        %4 = math.sin %3 : f32
        affine.store %4, %arg2[%arg8, %arg7, %arg6] : memref<4x2x128xf32, 1>
      }
    }
  }

  return
}
```

```
func.func @fused_kernel_0xBEF27837_1_f32(%arg0: memr
  affine.for %arg3 = 0 to 4 {
    affine.for %arg4 = 0 to 2 {
      affine.for %arg5 = 0 to 128 {
        %0 = affine.load %arg0[%arg3, %arg4, %arg5]
        %1 = affine.load %arg1[%arg3, %arg4, %arg5]
        %2 = arith.addf %0, %1 : f32
        %3 = math.sin %2 : f32
        affine.store %3, %arg2[%arg3, %arg4, %arg5]
      }
    }
  }
  return
}
```

Vectorization

The compiler converts independent loop iterations into SIMD instructions, based on the specific hardware vector width



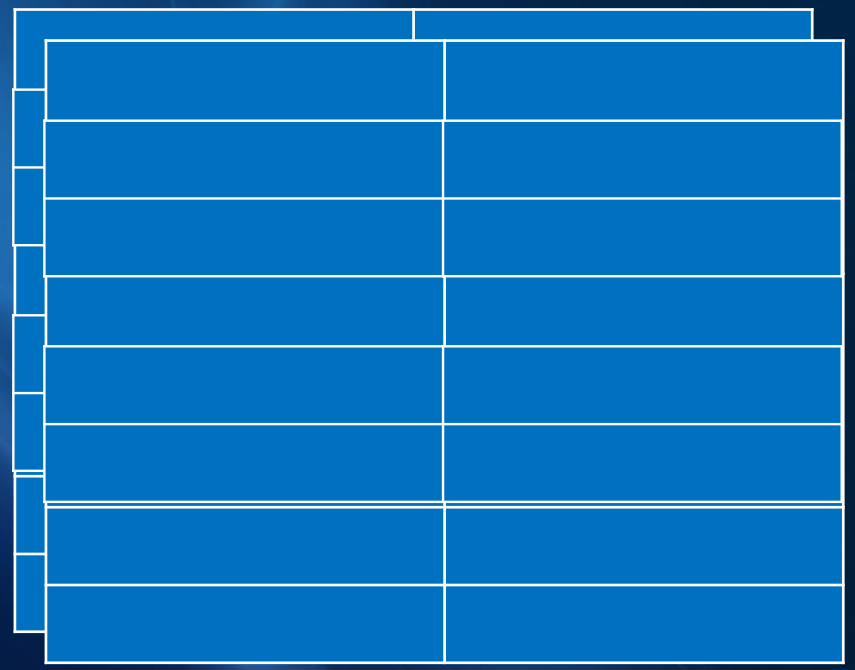
Vectorization

```
func.func @fused_kernel_0xBEF27837_1_f32(%arg0: memr
  affine.for %arg3 = 0 to 128 {
    affine.for %arg4 = 0 to 2 {
      affine.for %arg5 = 0 to 4 {
        %0 = affine.load %arg0[%arg5, %arg4, %arg3]
        %1 = affine.load %arg1[%arg5, %arg4, %arg3]
        %2 = arith.addf %0, %1 : f32
        %3 = math.sin %2 : f32
        affine.store %3, %arg2[%arg5, %arg4, %arg3]
      }
    }
  }
  return
}
```

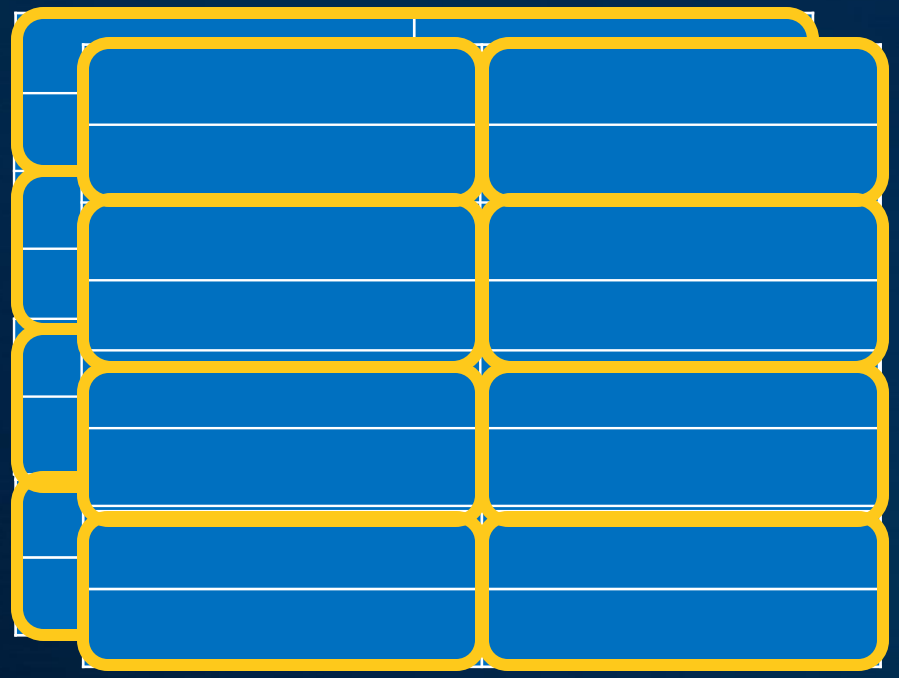
```
func.func @fused_kernel_0xBEF27837_1_f32(%arg0: memref<4x2x128
  affine.for %arg3 = 0 to 128 step 64 {
    affine.for %arg4 = 0 to 2 {
      affine.for %arg5 = 0 to 4 {
        %cst = arith.constant 0.000000e+00 : f32
        %0 = vector.transfer_read %arg0[%arg5, %arg4, %arg3],
        %cst_0 = arith.constant 0.000000e+00 : f32
        %1 = vector.transfer_read %arg1[%arg5, %arg4, %arg3],
        %2 = arith.addf %0, %1 : vector<64xf32>
        %3 = math.sin %2 : vector<64xf32>
        vector.transfer_write %3, %arg2[%arg5, %arg4, %arg3] {
      }
    }
  }
  return
}
```

Loop Unroll

The compiler combines operations from consecutive iterations and merges them into a single iteration



unroll_factor= 2



Loop Unroll

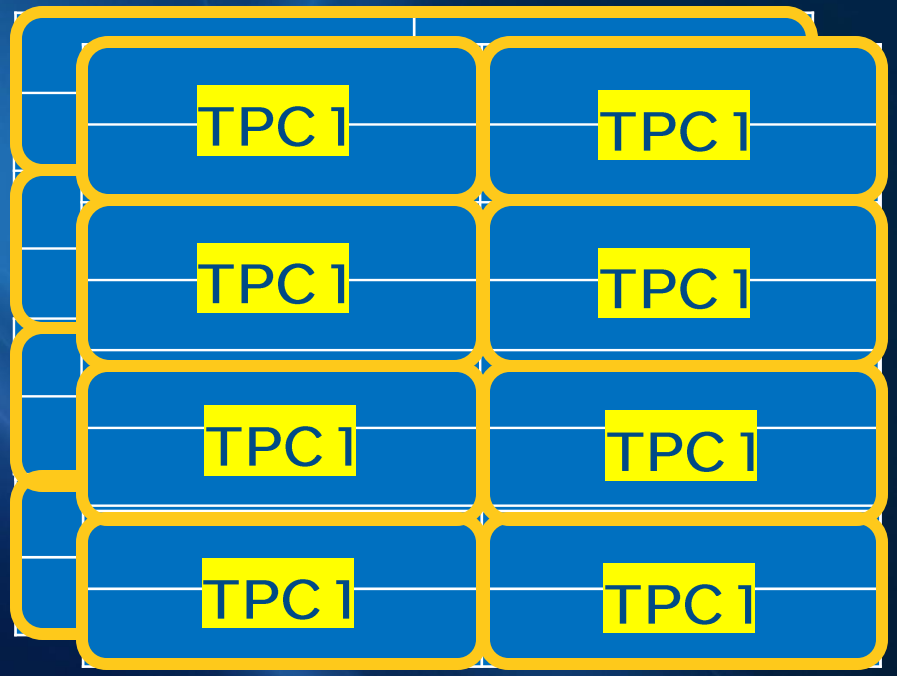
```
func.func @fused_kernel_0xBEF27837_1_f32(%arg0: memref
  affine.for %arg3 = 0 to 4 {
    affine.for %arg4 = 0 to 2 {
      affine.for %arg5 = 0 to 128 {
        %0 = affine.load %arg0[%arg3, %arg4, %arg5] : 
        %1 = affine.load %arg1[%arg3, %arg4, %arg5] : 
        %2 = arith.addf %0, %1 : f32
        %3 = math.sin %2 : f32
        affine.store %3, %arg2[%arg3, %arg4, %arg5] : 
      }
    }
  }
  return
}
```

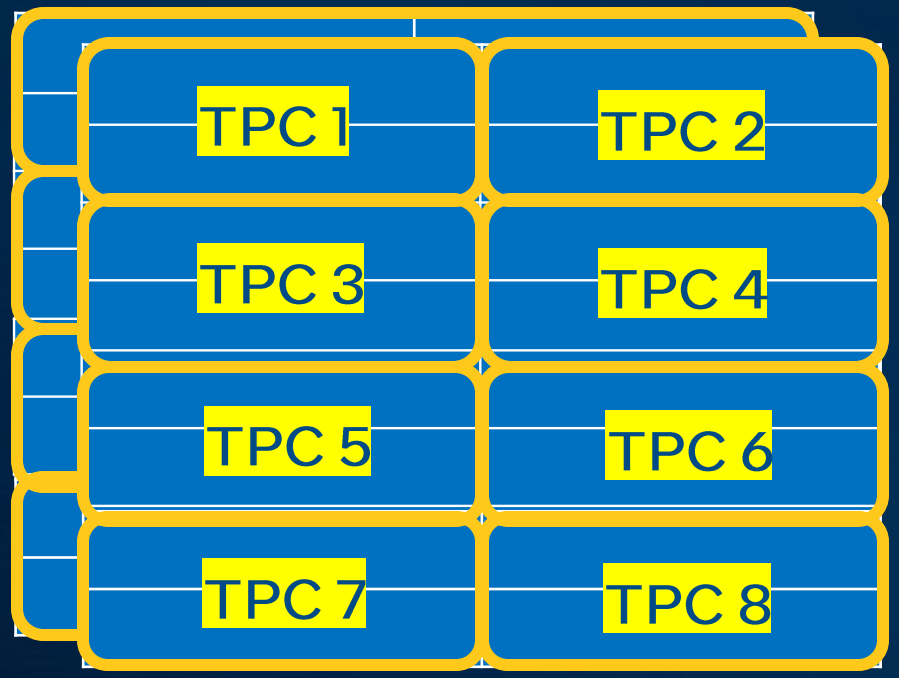
```
func.func @fused_kernel_0xBEF27837_1_f32(%arg0: memref<4x2x
  %c0 = arith.constant 0 : index
  affine.for %arg3 = 0 to 128 step 64 {
    affine.for %arg4 = 0 to 2 {
      %0 = affine.apply affine_map<(d0) -> (d0 + 3)>(%c0)
      %1 = affine.apply affine_map<(d0) -> (d0 + 2)>(%c0)
      %2 = affine.apply affine_map<(d0) -> (d0 + 1)>(%c0)
      %3 = affine.vector_load %arg0[%c0, %arg4, %arg3] : mer
      %4 = affine.vector_load %arg0[%2, %arg4, %arg3] : memr
      %5 = affine.vector_load %arg0[%1, %arg4, %arg3] : memr
      %6 = affine.vector_load %arg0[%0, %arg4, %arg3] : memr
      %7 = affine.vector_load %arg1[%c0, %arg4, %arg3] : mer
      %8 = affine.vector_load %arg1[%2, %arg4, %arg3] : memr
      %9 = affine.vector_load %arg1[%1, %arg4, %arg3] : memr
      %10 = affine.vector_load %arg1[%0, %arg4, %arg3] : mer
      %11 = arith.addf %3, %7 : vector<64xf32>
      %12 = arith.addf %4, %8 : vector<64xf32>
      %13 = arith.addf %5, %9 : vector<64xf32>
      %14 = arith.addf %6, %10 : vector<64xf32>
      %15 = math.sin %11 : vector<64xf32>
      %16 = math.sin %12 : vector<64xf32>
      %17 = math.sin %13 : vector<64xf32>
      %18 = math.sin %14 : vector<64xf32>
      affine.vector_store %15, %arg2[%c0, %arg4, %arg3] : me
      affine.vector_store %16, %arg2[%2, %arg4, %arg3] : mer
      affine.vector_store %17, %arg2[%1, %arg4, %arg3] : mer
      affine.vector_store %18, %arg2[%0, %arg4, %arg3] : mer
    }
  }
  return
}
```

unroll_factor= 4
"fully unroll" one of the loops

Parallelization

The compiler transforms the constant bounds of a loop into variables, enabling scalable parallel execution across multiple processing units





Parallelization

```
func.func @fused_kernel_0xBEF27837_1_f32(%arg0: memref<4x2x1
  %c0 = arith.constant 0 : index
  affine.for %arg3 = 0 to 128 step 64 {
    affine.for %arg4 = 0 to 2 {
      %0 = affine.apply affine_map<(d0) -> (d0 + 3)>(%c0)
      %1 = affine.apply affine_map<(d0) -> (d0 + 2)>(%c0)
      %2 = affine.apply affine_map<(d0) -> (d0 + 1)>(%c0)
```

```
func.func @fused_kernel_0xBEF27837_1_f32(%arg0: memref<4x2x128x
  %0 = tpc.get_index_space_start 0
  %1 = tpc.get_index_space_end 0
  %2 = tpc.get_index_space_start 1
  %3 = tpc.get_index_space_end 1
  %c0 = arith.constant 0 : index
  %4 = affine.apply affine_map<(d0) -> (d0 + 3)>(%c0)
  %5 = affine.apply affine_map<(d0) -> (d0 + 2)>(%c0)
  %6 = affine.apply affine_map<(d0) -> (d0 + 1)>(%c0)
  affine.for %arg3 = %0 to %1 {
    %7 = affine.apply affine_map<(d0) -> (d0 * 64)>(%arg3)
    affine.for %arg4 = %2 to %3 {
      %8 = affine.vector_load %arg0[%c0, %arg4, %7] : memref<4x
      %9 = affine.vector_load %arg0[%6, %arg4, %7] : memref<4x2
      %10 = affine.vector_load %arg0[%5, %arg4, %7] : memref<4x
```

Case Study:
Adjusting the TPC-Fuser to LLMs Recent Challenges

Large Language Models and Triangular Softmax

- Make predictions about the next word, for each prefix of the sentence.

The									
The	goal								
The	goal	of							
The	goal	of	the						
The	goal	of	the	model					
The	goal	of	the	model	is				
The	goal	of	the	model	is	to			
The	goal	of	the	model	is	to	predict		
The	goal	of	the	model	is	to	predict	the	
The	goal	of	the	model	is	to	predict	the	next

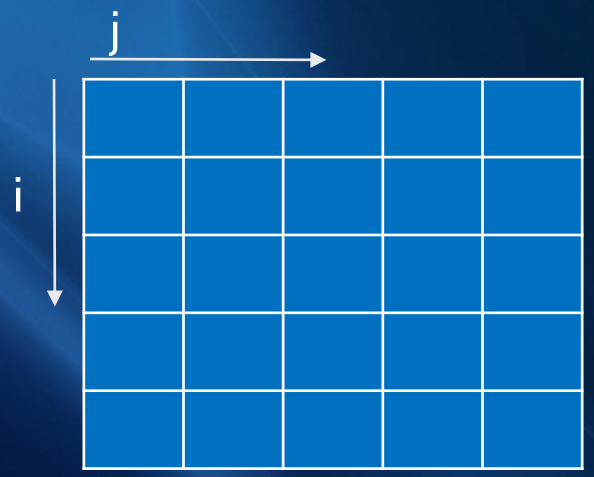
Token	Probability
word	0.73
sentence	0.12
level	0.07
time	0.05
step	0.0229
banana	0.0001

Traditional vs. Triangular Access to the Data

- Supporting a new type of data access created challenges.

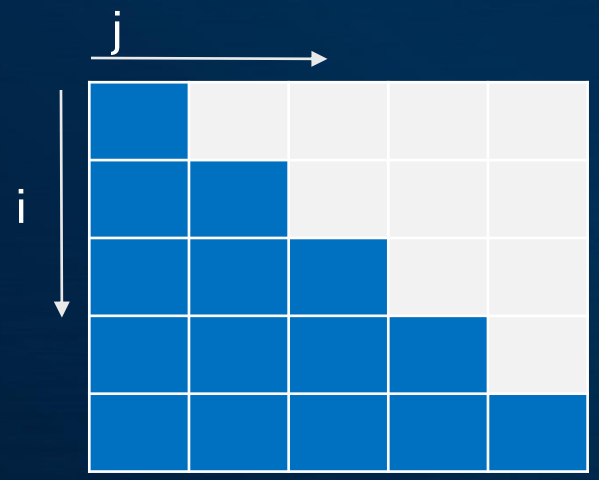
Traditional access

```
for (i = 0; i < N; ++i) {
  for (j = 0; j < M; ++j) {
    // Access A[i][j]
  }
}
```



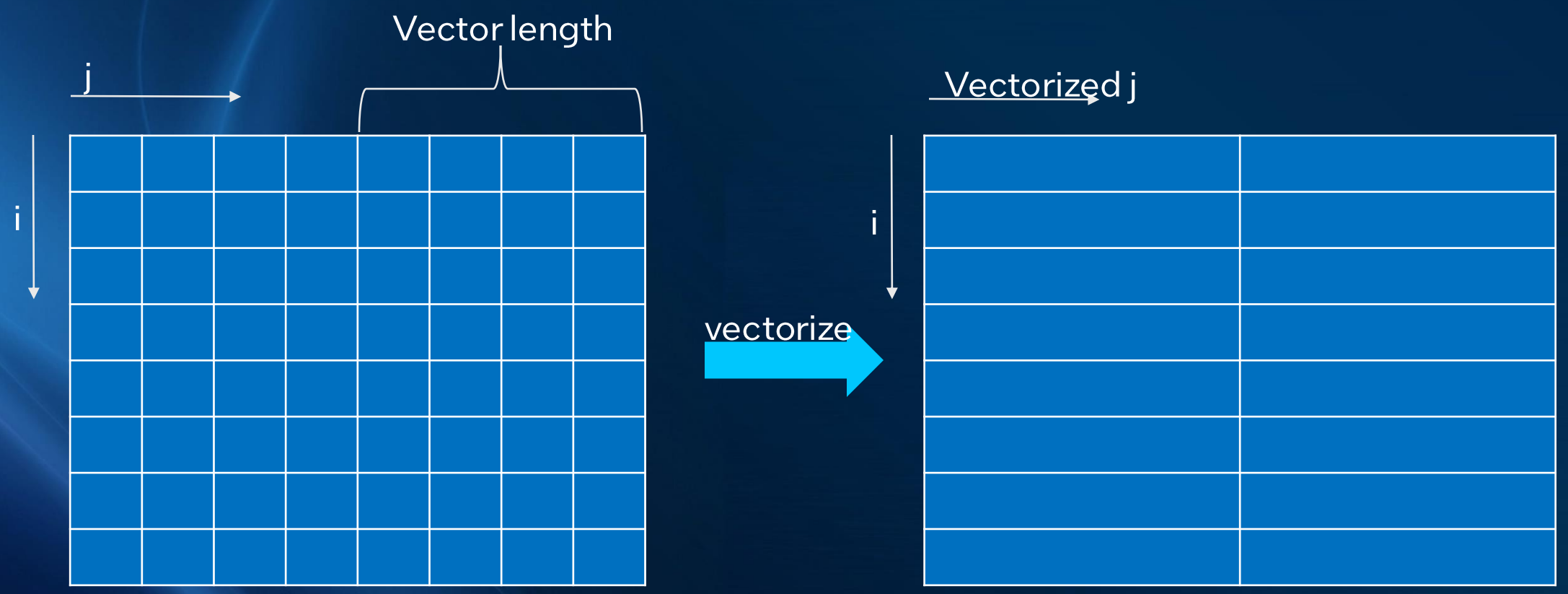
Triangular access

```
for (i = 0; i < N; ++i) {
  for (j = 0; j <= i; ++j) {
    // Access A[i][j]
  }
}
```



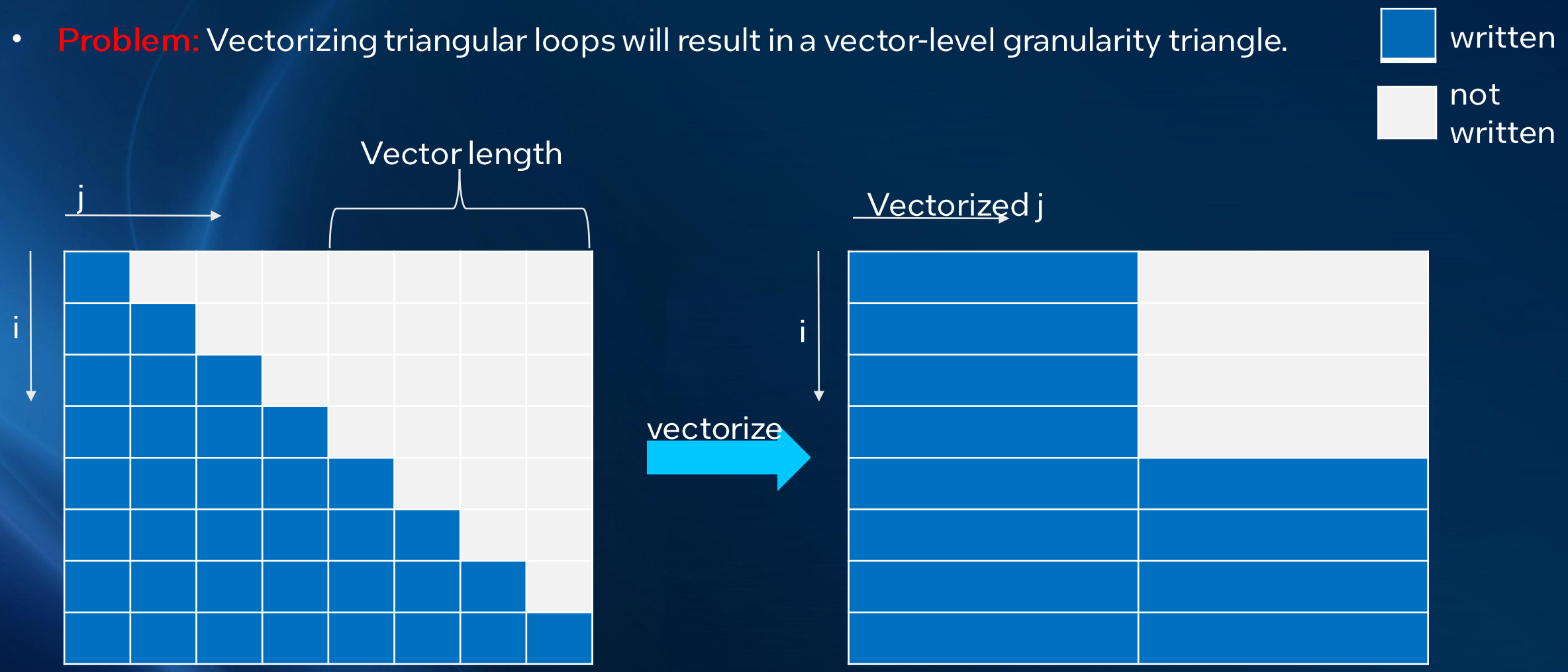
Vectorizing triangular loops

- Traditional vectorization, for rectangular (regular) tensor access:



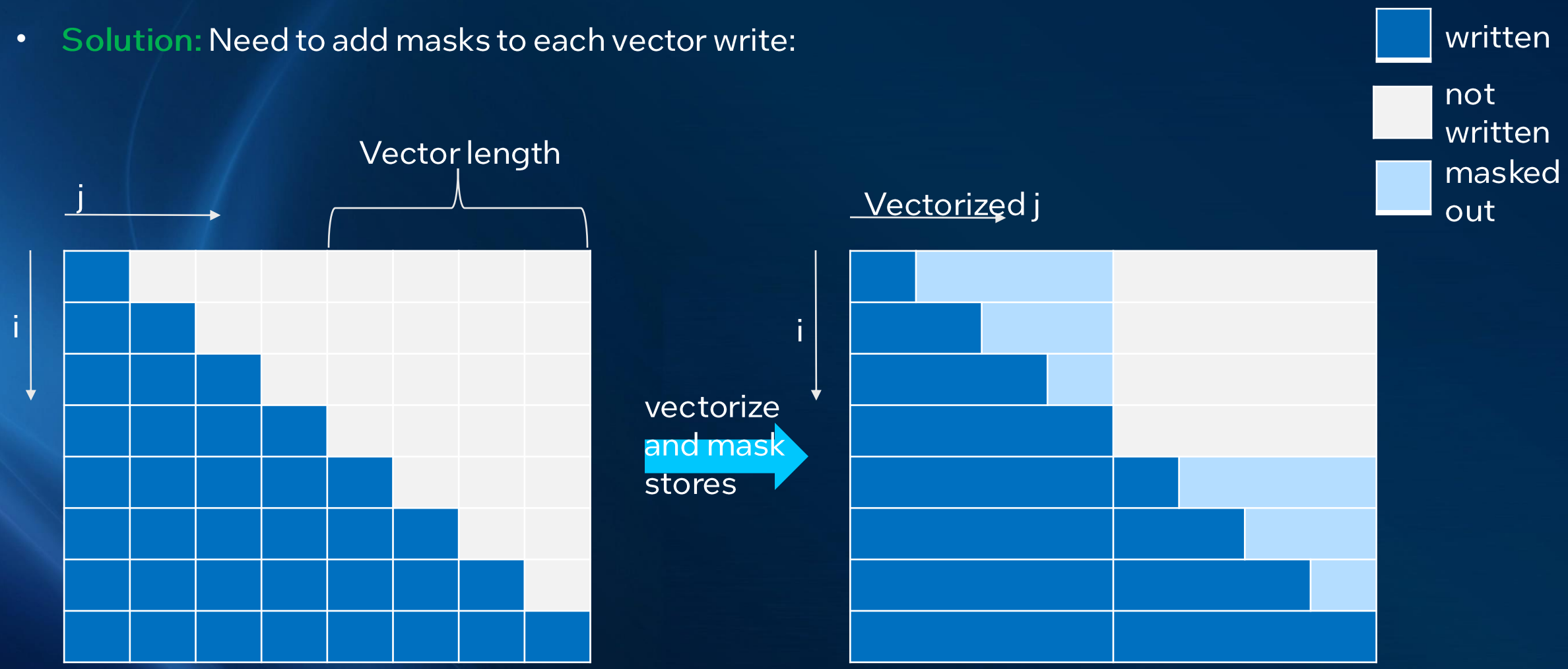
Vectorizing triangular loops

- **Problem:** Vectorizing triangular loops will result in a vector-level granularity triangle.



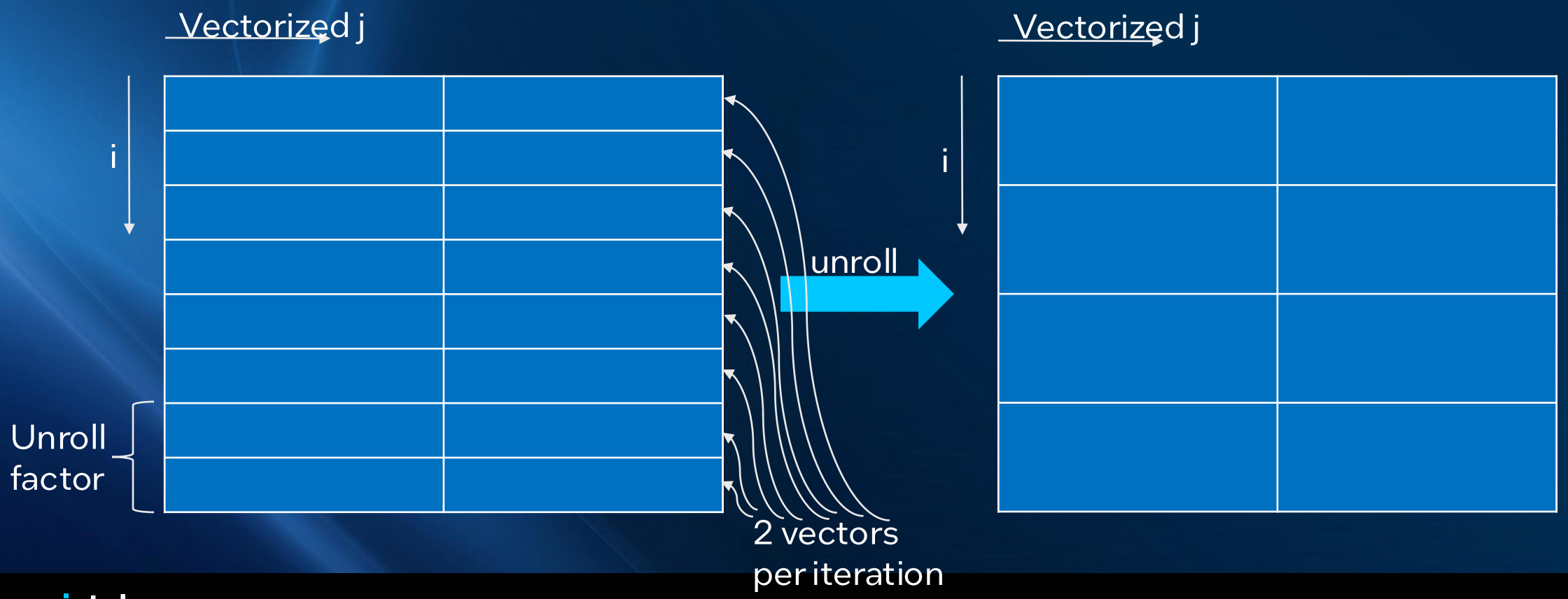
Vectorizing triangular loops

- **Solution:** Need to add masks to each vector write:



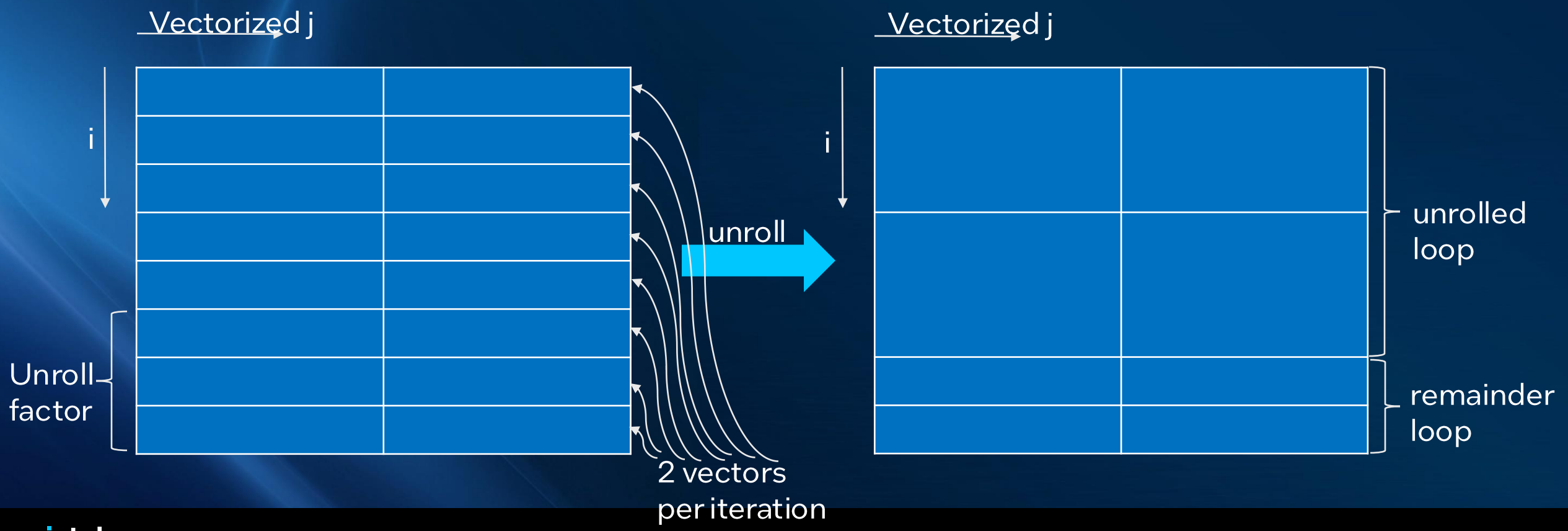
Unrolling triangular loops

- Traditional unrolling, for rectangular (regular) tensor access with unroll factor = 2:



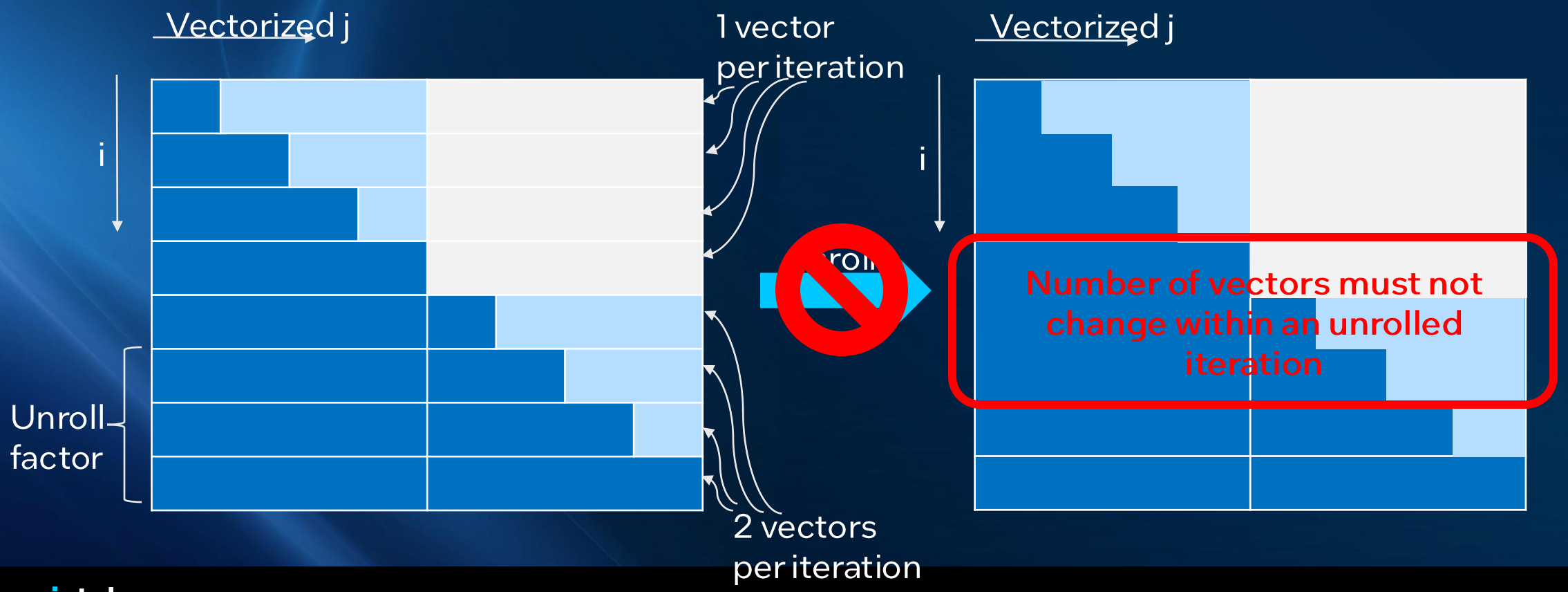
Unrolling triangular loops

- Traditional unrolling, for rectangular (regular) tensor access with unroll factor = 3:



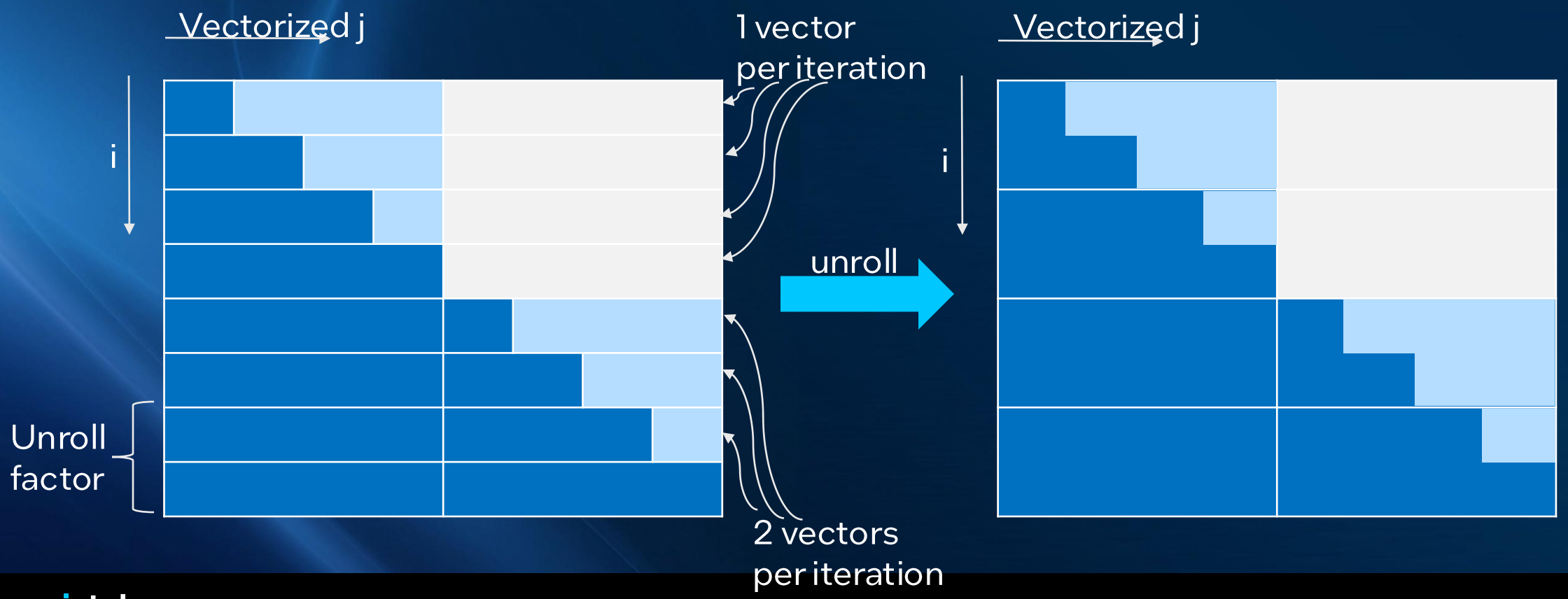
Unrolling triangular loops

- **Problem:** Some unroll factors may result in operating on different number of vectors
- Unrolling triangular tensor access with unroll factor = 3 is forbidden:

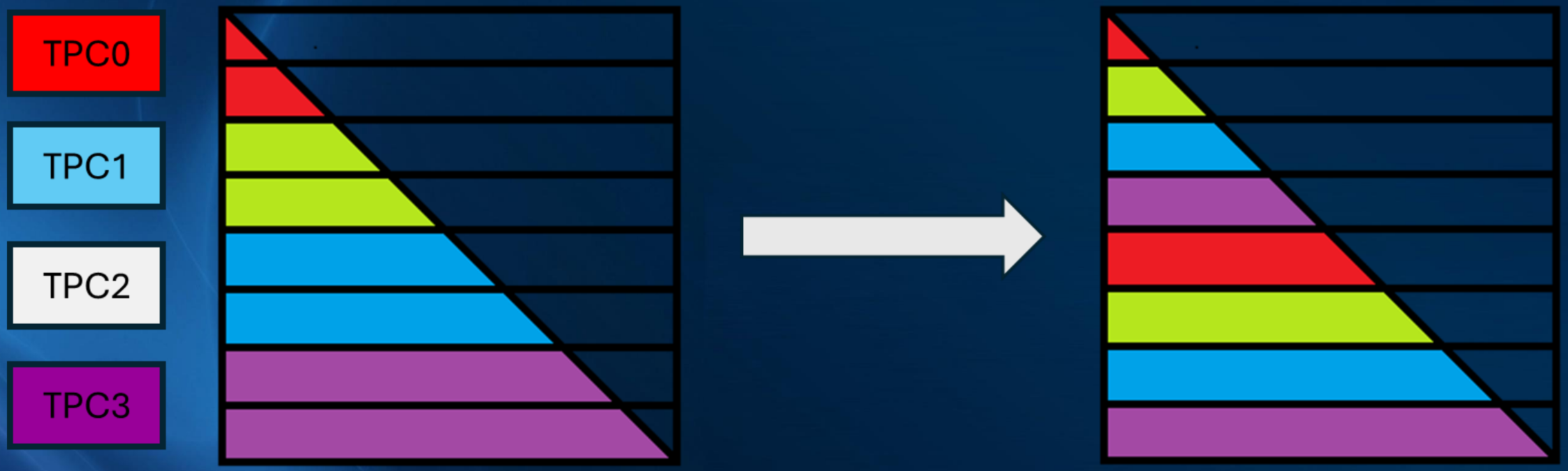


Unrolling triangular loops

- Solution: Allow only legal unroll factors:
  - vector_size % unroll_factor == 0
- Unrolling triangular tensor access with unroll factor = 2 is allowed:



Parallelizing triangular loops



Parallelizing Triangular Loops

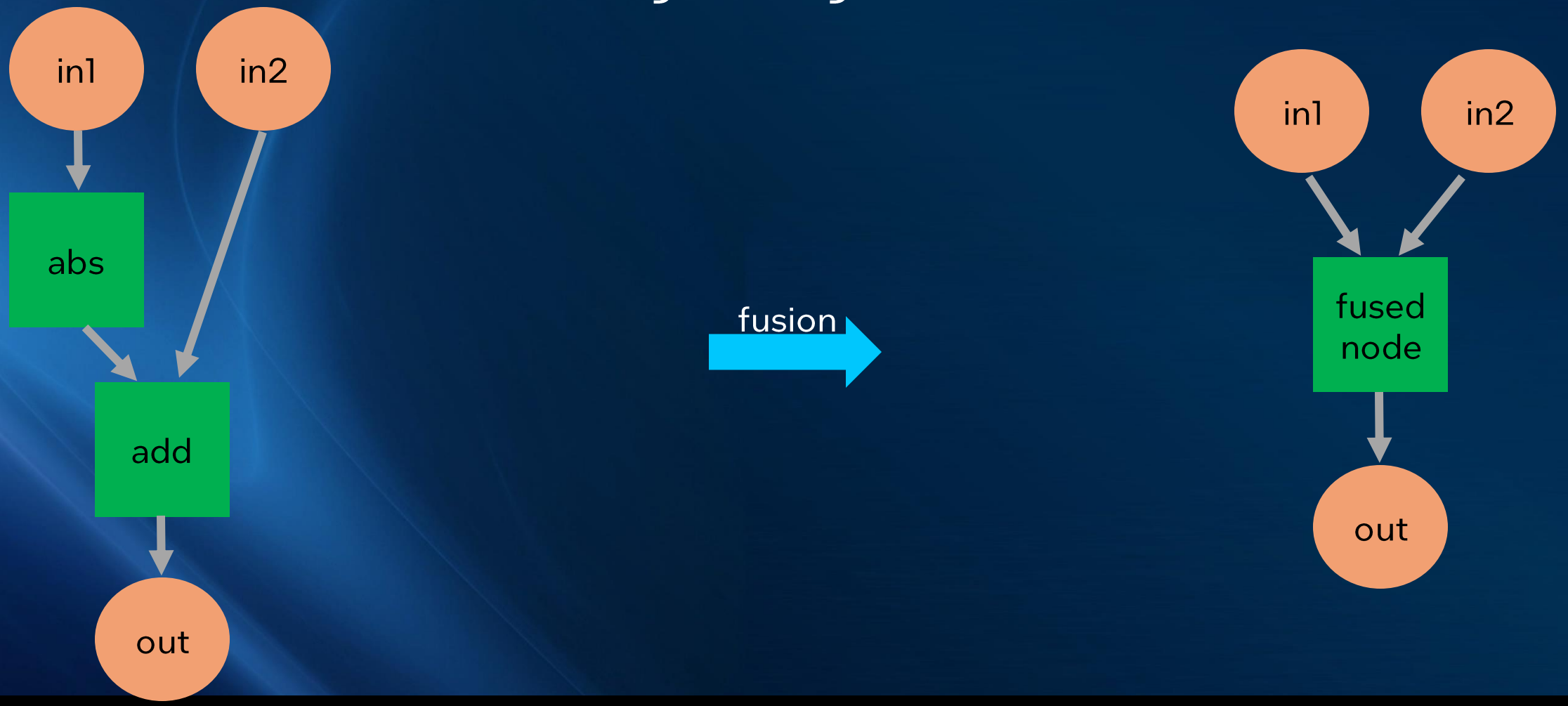
Performance with naïve parallelization – 219.6µs

Parallelizing Triangular Loops

Performance with triangular-adapted parallelization – 170.8µs. 22% speedup!

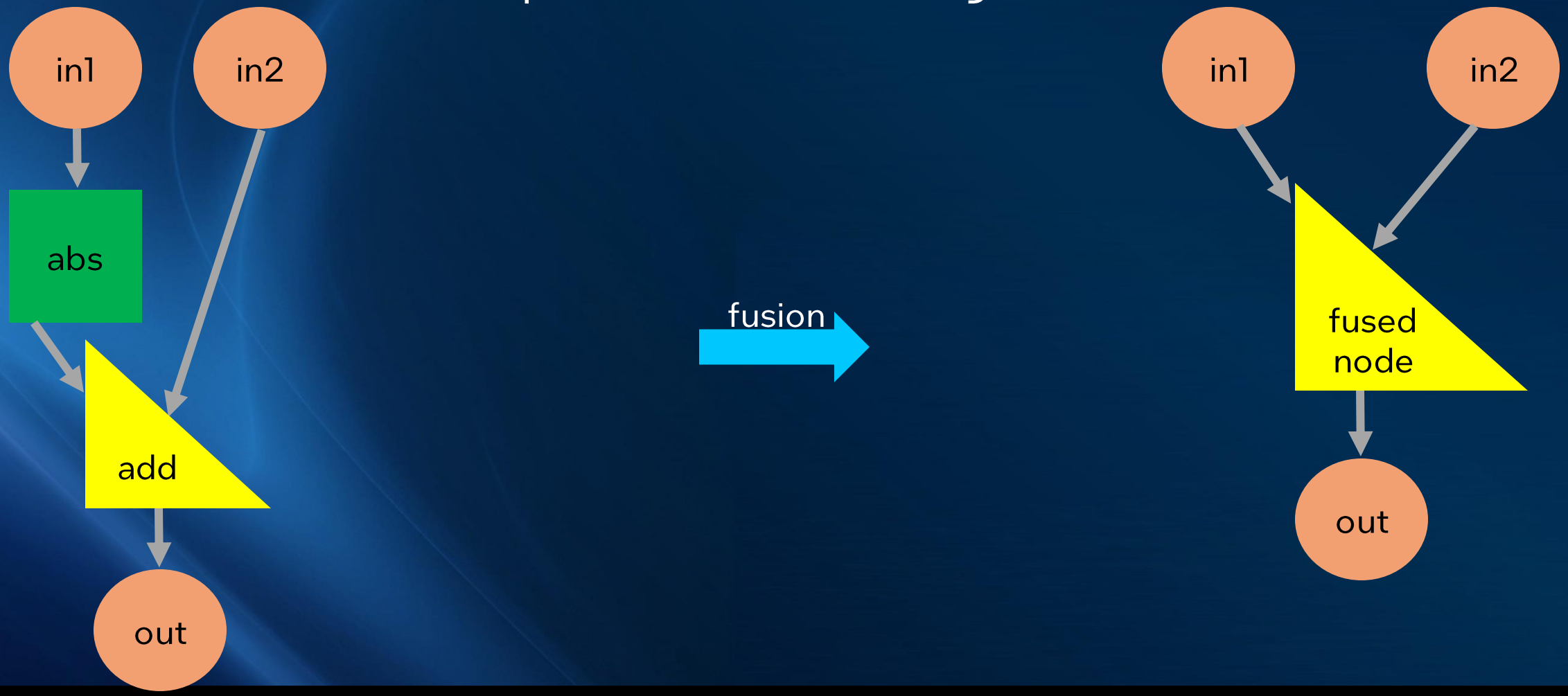
Fusing Triangular and Non-triangular Loops

Traditional fusion, for rectangular (regular) tensor access:



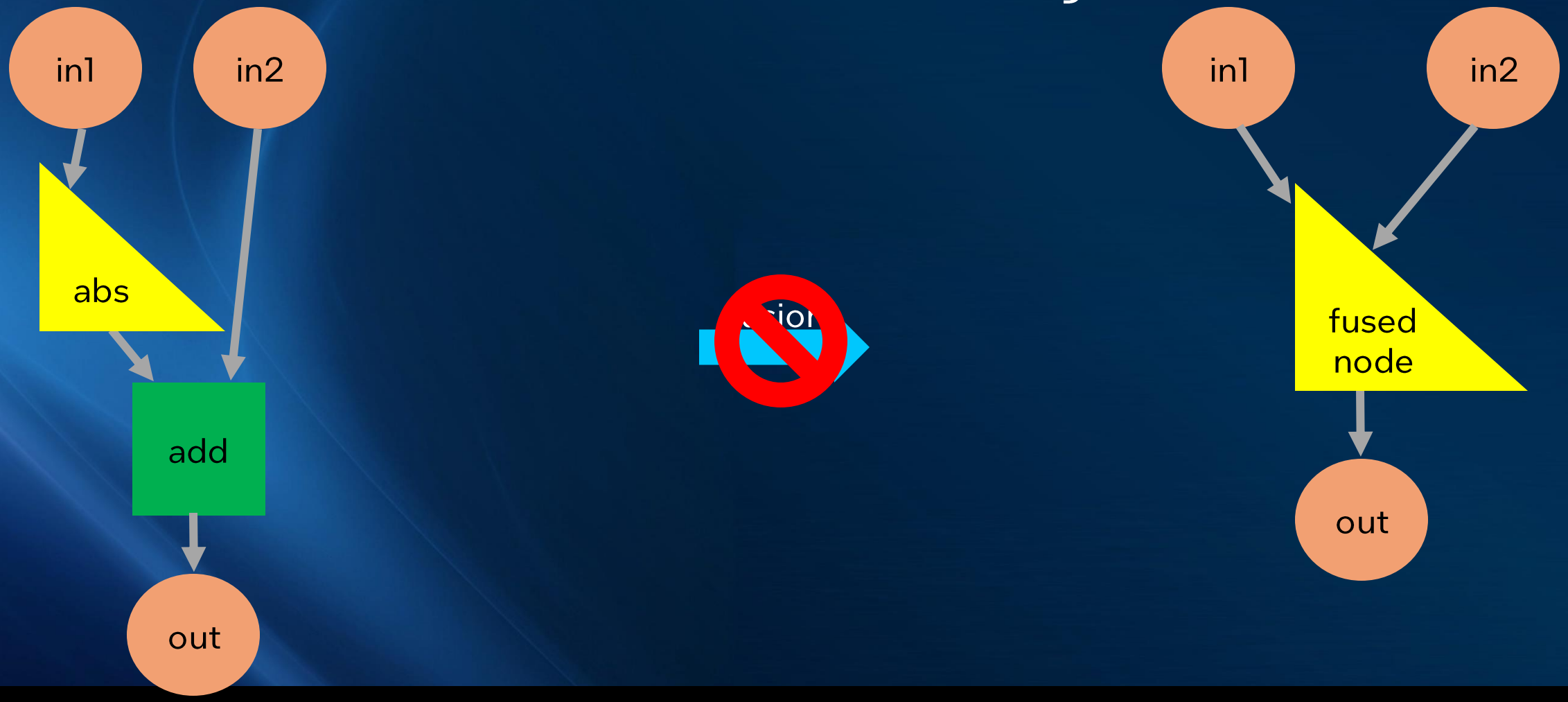
Fusing Triangular and Non-triangular Loops

Can fuse elementwise producers into a triangular cluster:



Fusing Triangular and Non-triangular Loops

Cannot fuse elementwise consumers into a triangular cluster:



Triangular Data Access

The introduction of a new data access pattern introduced new challenges:

- Correctness challenges
- Performance challenges
- Operation fusion challenges

Key Takeaways

- The TPC Fuser is a JIT compiler for deep learning kernels
- It is deployed as part of Gaudi Synapse SW stack
- Delivers significant performance improvements
- Works in-tandem with a Graph Compiler to optimize execution across the entire accelerator

intel®

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Intel technologies may require enabled hardware, software or service activation.

Availability of accelerators varies depending on SKU. Please contact your Intel sales representative for more information.

Your costs and results may vary.