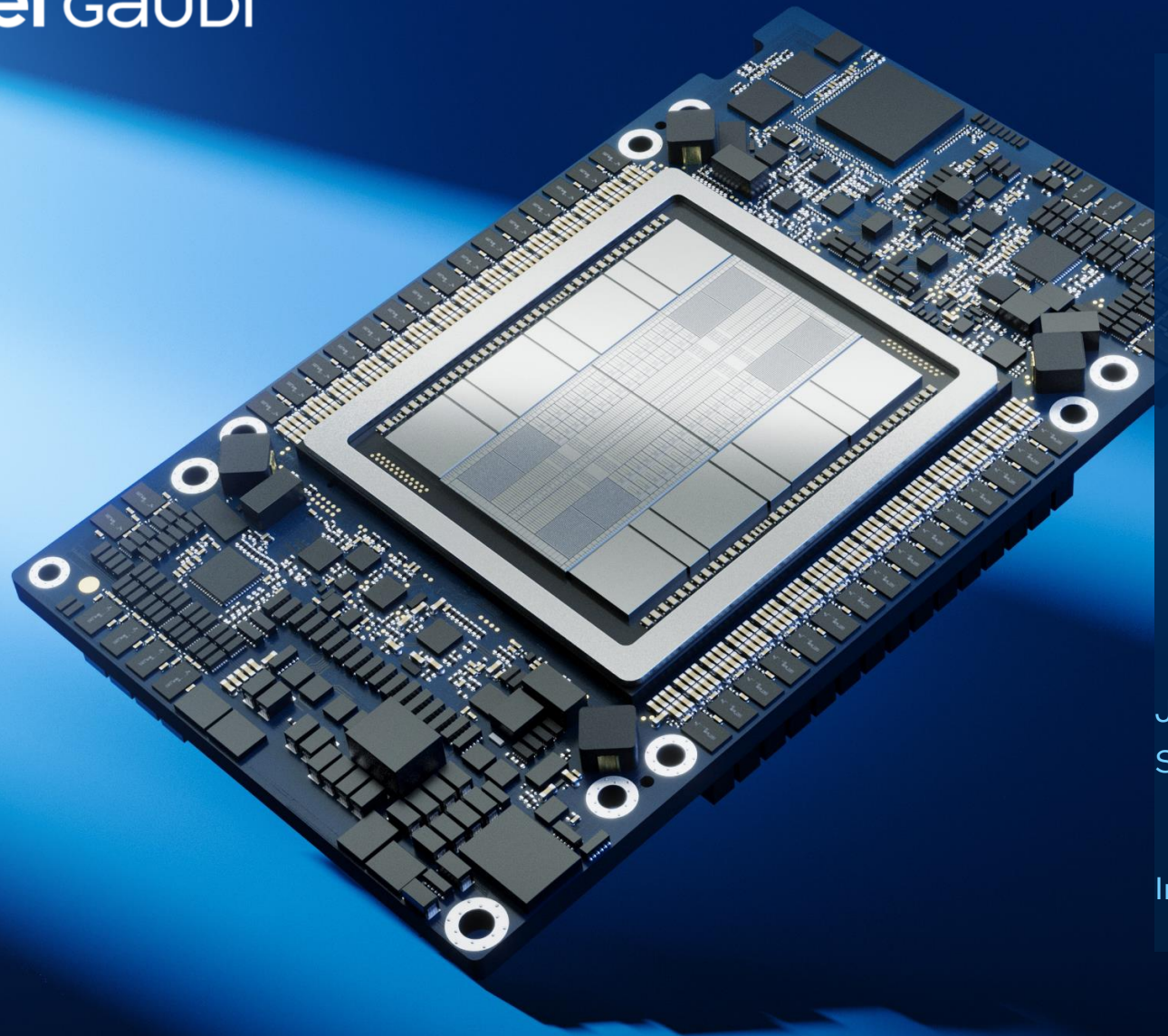
intel GAUDI



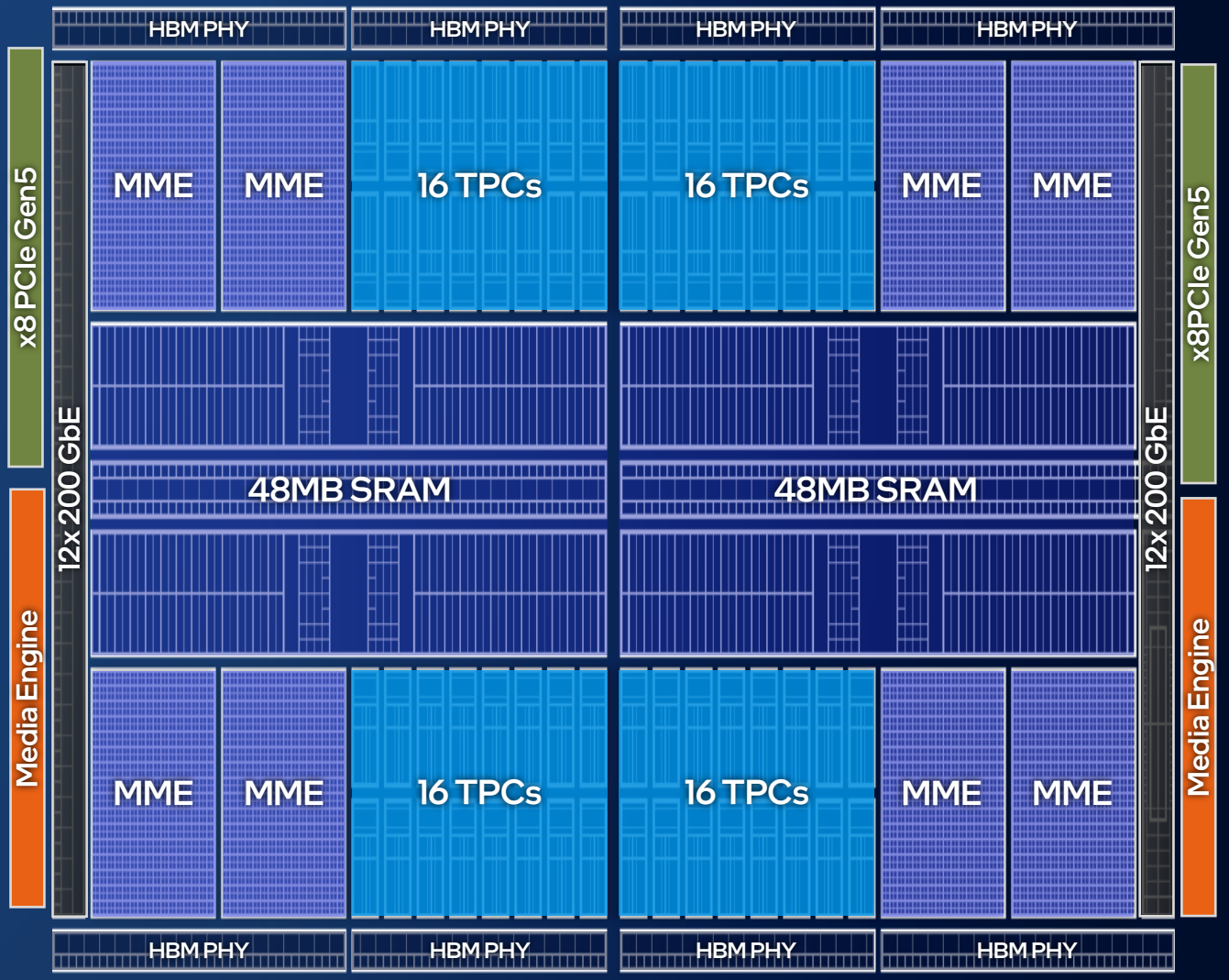
Speeding up Intel® Gaudi® deep-learning accelerators using an MLIR-based compiler

Jayaram Bobba, Tzachi Cohen, Dibyendu Das, Sergei Grechanik, Dafna Mordechai

Intel/Habana Labs

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)



Matrix Multiplication and Vector Engines

Matrix Multiplication Engine (MME): designed for AI efficiency

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 Processing Core (TPC):

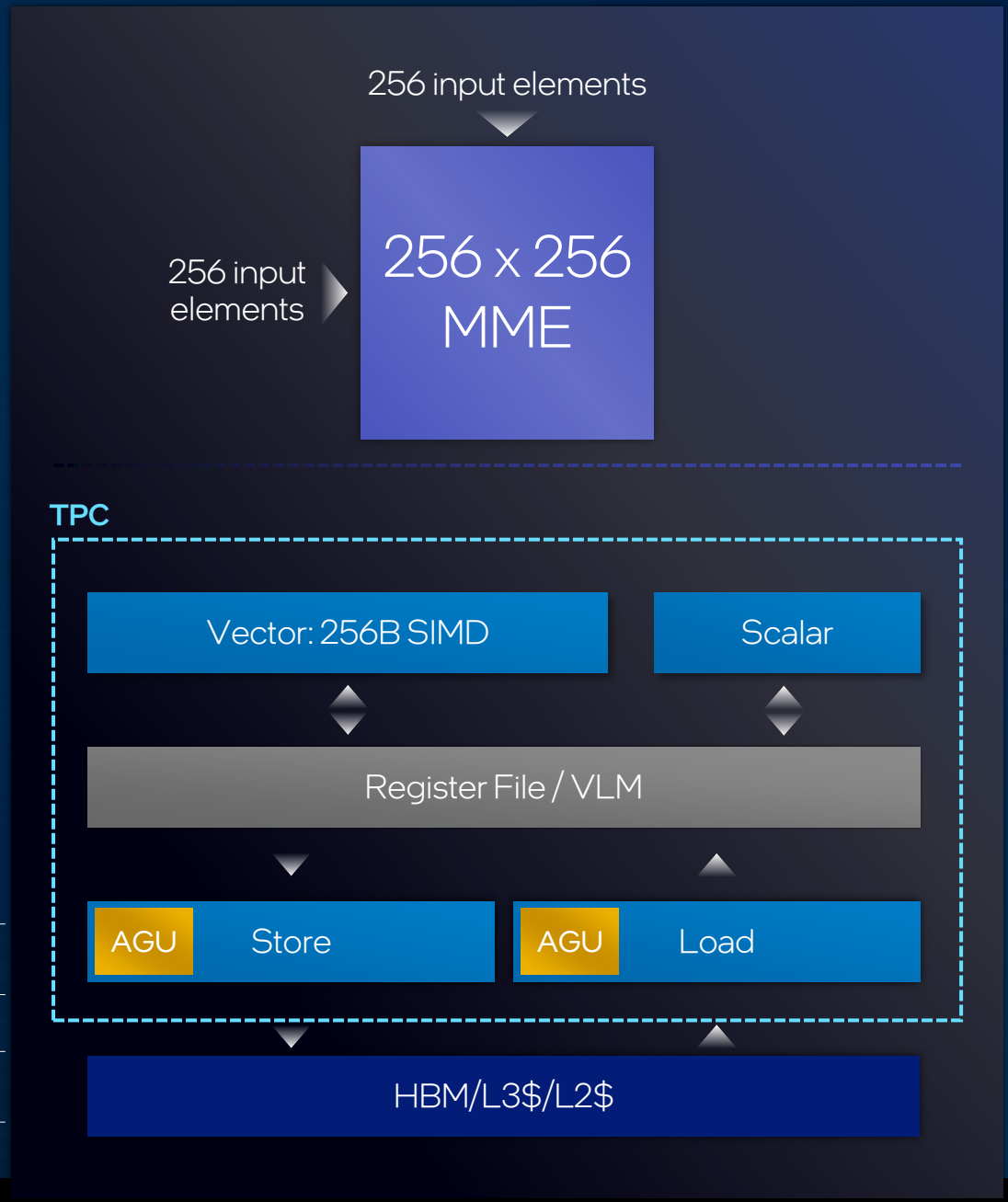
256B-wide SIMD Vector Processor

Programmable: 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

Integrates the main Gen AI frameworks used today

Supports FP16/BF16 → FP8 quantization

Main proprietary SW layers

Graph Compiler: Handles all engine dependency and scheduling logic

Matrix operations: Configuring the MME

TPC kernels: All non-Matrix operations

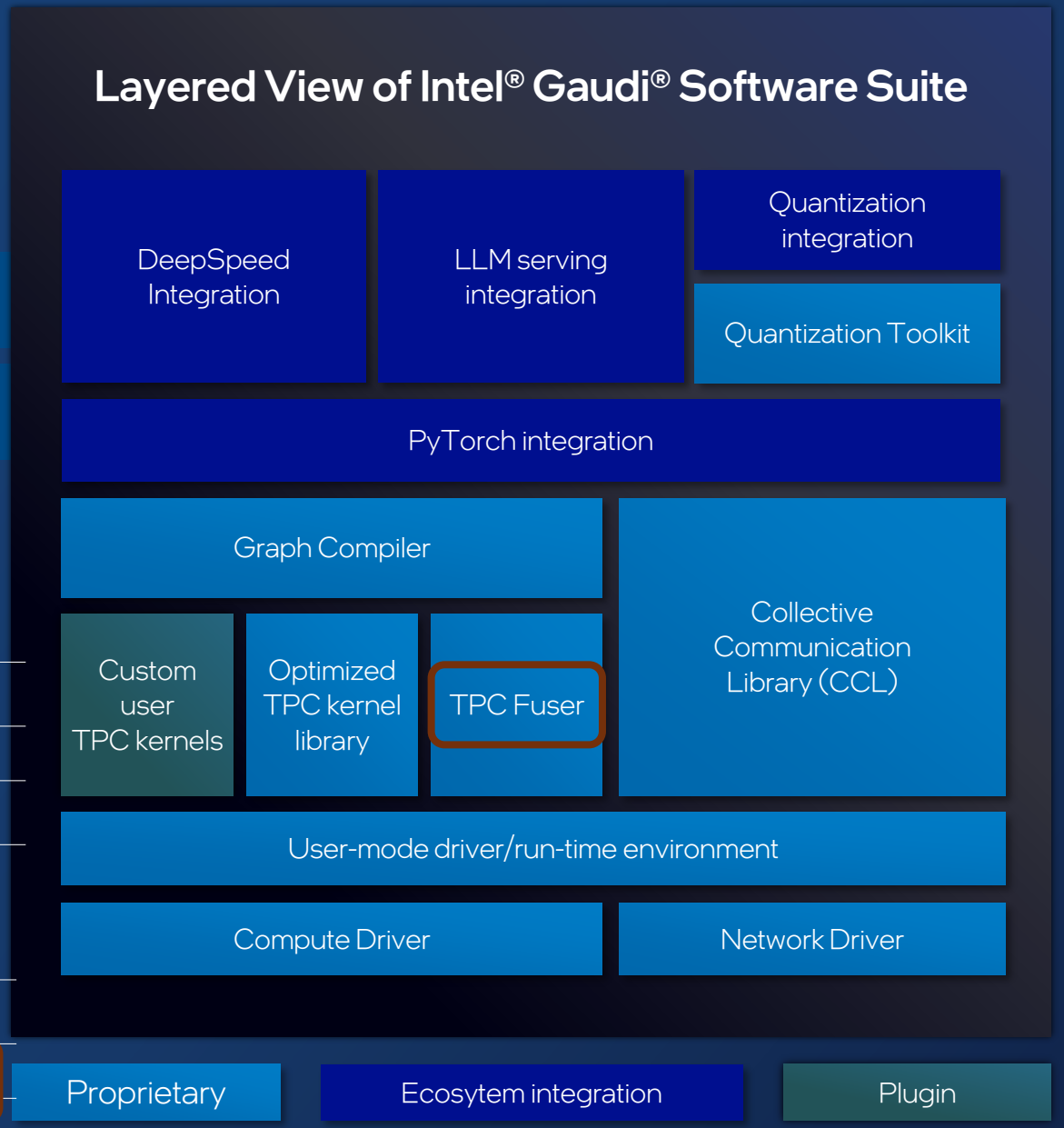
Collective Communication Library (CCL)

Several sources for TPC Kernels

Gaudi optimized TPC kernel library

Custom user kernels

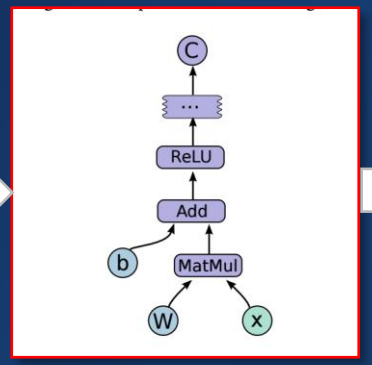
MLIR-based fused kernels: generated during graph compilation



Graph Compilation Flow

PyT code

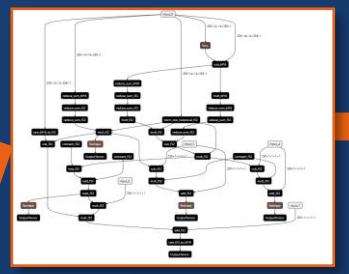
Pytorch-Synapse Bridge



Synapse Dataflow Graph

Synapse Graph Compiler (GC)

TPC Sub-graph



MLIR Fuser

tpc-clang

fused_kernel.o (JIT)

Pre-Compiled binary

Custom/TPC ops

custom-kernel.o

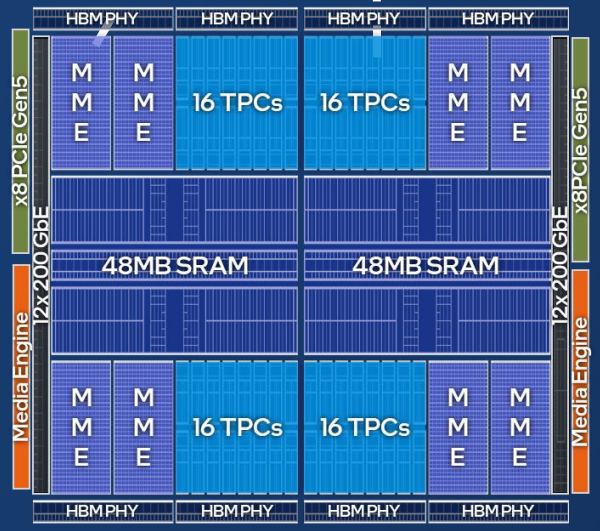
Perflib

tpc-clang

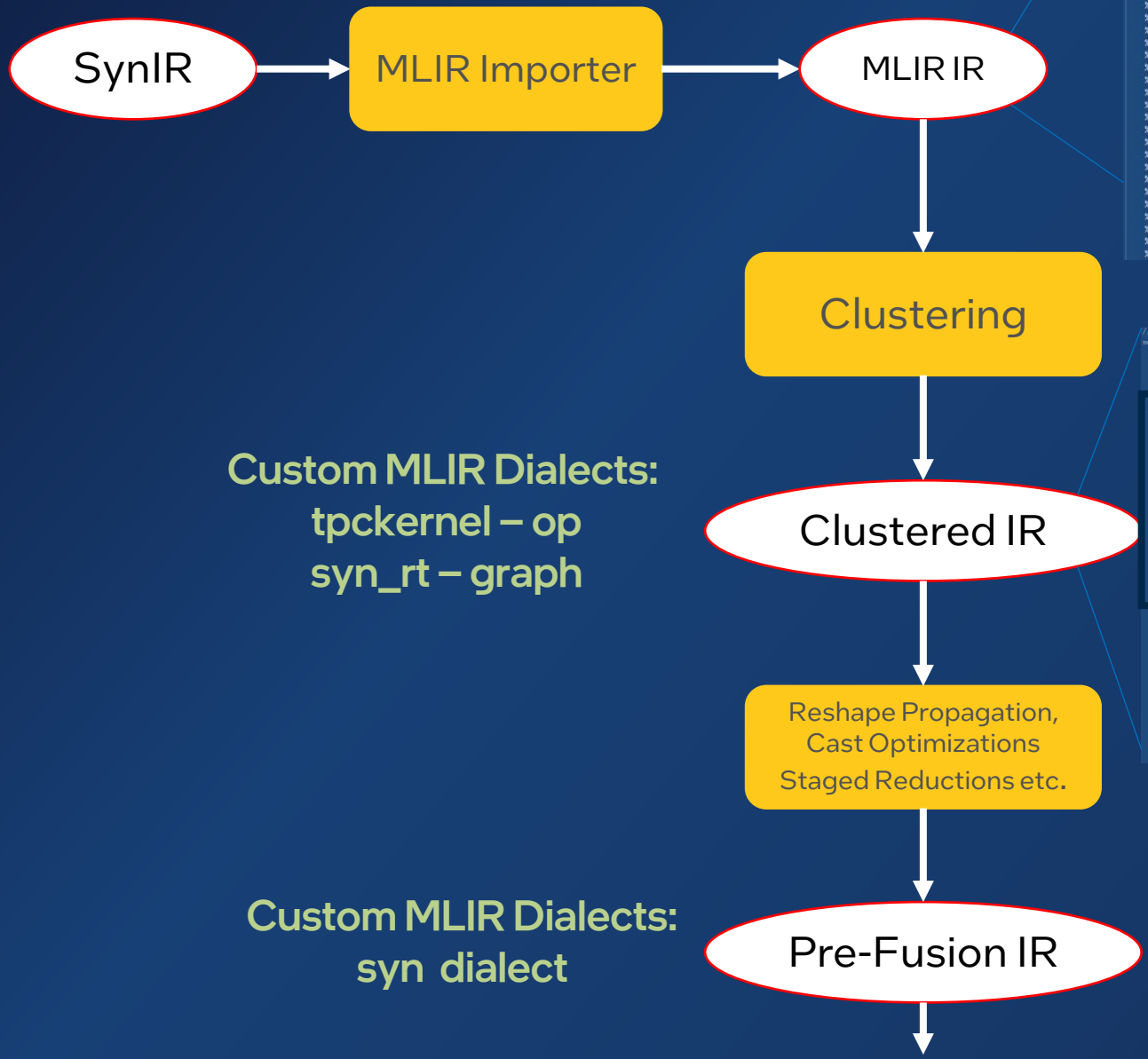
Custom Kernel in TPC-C language

Schedule ***MatMul*** on MME

Schedule ***kernels*** on TPC

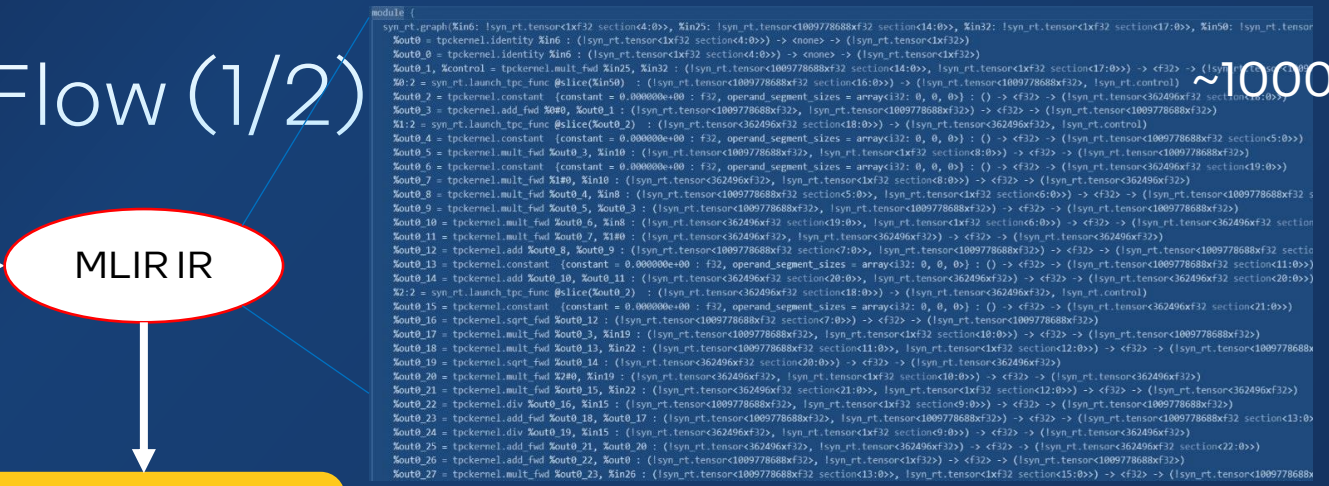


Fuser Compilation Flow (1/2)



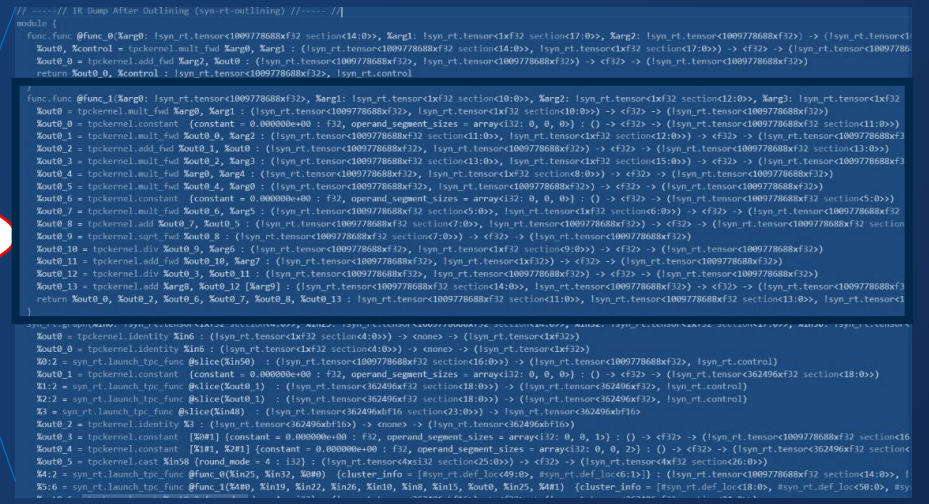
Custom MLIR Dialects:
tpckernel – op
syn_rt – graph

Custom MLIR Dialects:
syn dialect



~1000s of ops

10s of ops per cluster



1 cluster ~ 1 kernel

Primitive ops – elementwise, broadcast, reduction.
Complex ops like norms, softmax, dropout broken down into primitive ops.

Operator Fusion

```
func.func @softmax(%arg0: tensor<10x256xf32>, %arg1: tensor<10x256xf32>)
                   -> (tensor<10x256xf32>) {
  %c0 = arith.constant 0 : index
  %0 = syn.add %arg0, %arg1 : tensor<10x256xf32>
  %m = syn.reduce max %0 {dim = 1} : tensor<10x256xf32> -> tensor<10x1xf32>
  %mb = syn.broadcast %m : tensor<10x1xf32> to tensor<10x256xf32>
  %sub = syn.sub %0, %mb : tensor<10x256xf32>
  %e = syn.exp %sub : tensor<10x256xf32>
  %se = syn.reduce add %e {dim = 1} : tensor<10x256xf32> -> tensor<10x1xf32>
  %seb = syn.broadcast %se : tensor<10x1xf32> to tensor<10x256xf32>
  %div = syn.div %e, %seb : tensor<10x256xf32>
  return %div : tensor<10x256xf32>
}
```

Create ‘Loop’ Clusters subject to dimension ordering constraints

```
cluster 0 [10]  {
  cluster 3 [256] {
    %0 = syn.add %arg0, %arg1 : tensor<10x256xf32>
    %1 = syn.reduce max %0 {dim = 1 : i64} : tensor<10x256xf32> -> tensor<10x1xf32>
  }
  cluster 2 [256] {
    %2 = syn.broadcast %1 : tensor<10x1xf32> to tensor<10x256xf32>
    %3 = syn.sub %0, %2 : tensor<10x256xf32>
    %4 = syn.exp %3 : tensor<10x256xf32>
    %5 = syn.reduce add %4 {dim = 1 : i64} : tensor<10x256xf32> -> tensor<10x1xf32>
  }
  cluster 1 [256] {
    %6 = syn.broadcast %5 : tensor<10x1xf32> to tensor<10x256xf32>
    %7 = syn.div %4, %6 : tensor<10x256xf32>
  }
}
```

Generate loops that operate on scalar values

```
func.func @softmax(%arg0: tensor<10x256xf32>, %arg1: tensor<10x256xf32>) -> tensor<10x256xf32> {
  %c0 = arith.constant 0 : index
  %0 = syn.generate %arg2 : [10] outs(assign #map0 : tensor<10x256xf32>) {
    %1 = tensor.extract_slice %arg0[%arg2, 0] [1, 256] [1, 1] : tensor<10x256xf32> to tensor<256xf32>
    %2 = tensor.extract_slice %arg1[%arg2, 0] [1, 256] [1, 1] : tensor<10x256xf32> to tensor<256xf32>
    %3:2 = syn.generate %arg3 : [256] outs(assign #map1 : tensor<256xf32>, max #map2 : tensor<1xf32>) {
      %6 = tensor.extract_slice %1[%arg3] [1] [1] : tensor<256xf32> to tensor<f32>
      %7 = tensor.extract_slice %2[%arg3] [1] [1] : tensor<256xf32> to tensor<f32>
      %8 = syn.add %6, %7 : tensor<f32>
      syn.yield %8, %8 : tensor<f32>, tensor<f32>
    }
    %4:2 = syn.generate %arg3 : [256] outs(assign #map1 : tensor<256xf32>, add #map2 : tensor<1xf32>) {
      %6 = tensor.extract_slice %3#1[0] [1] [1] : tensor<1xf32> to tensor<f32>
      %7 = tensor.extract_slice %3#0[%arg3] [1] [1] : tensor<256xf32> to tensor<f32>
      %8 = syn.sub %7, %6 : tensor<f32>
      %9 = syn.exp %8 : tensor<f32>
      syn.yield %9, %9 : tensor<f32>, tensor<f32>
    }
    %5 = syn.generate %arg3 : [256] outs(assign #map1 : tensor<256xf32>) {
      %6 = tensor.extract_slice %4#1[0] [1] [1] : tensor<1xf32> to tensor<f32>
      %7 = tensor.extract_slice %4#0[%arg3] [1] [1] : tensor<256xf32> to tensor<f32>
      %8 = syn.div %7, %6 : tensor<f32>
      syn.yield %8 : tensor<f32>
    }
    syn.yield %5 : tensor<256xf32>
  }
  return %0 : tensor<10x256xf32>
}
```

Fusion Search Space Exploration

Beam Search

```
%0 = syn.add %x, %y : tensor<10x256xf32>
```

```
cluster 0 [10] {
  cluster 1 [256] {
    %1 = syn.exp %0 : tensor<10x256xf32> [0, 1]
  }
}
```

```
// No fusion
cluster 2 [10] {
  cluster 3 [256] {
    %0 = syn.add %x, %y : tensor<10x256xf32>
  }
}
cluster 0 [10] {
  cluster 1 [256] {
    %1 = syn.exp %0 : tensor<10x256xf32>
  }
}
```

```
// Only the outer dim is fused
cluster 0 [10] {
  cluster 3 [256] {
    %0 = syn.add %x, %y : tensor<10x256xf32> [0, 3]
  }
  cluster 1 [256] {
    %1 = syn.exp %0 : tensor<10x256xf32> [0, 1]
  }
}
```

```
// Only the inner dim is fused
cluster 1 [256] {
  cluster 2 [10] {
    %0 = syn.add %x, %y : tensor<10x256xf32>
  }
  cluster 0 [10] {
    %1 = syn.exp %0 : tensor<10x256xf32>
  }
}
```

```
// All dimensions are fused
cluster 0 [10] {
  cluster 1 [256] {
    %0 = syn.add %x, %y : tensor<10x256xf32>
    %1 = syn.exp %0 : tensor<10x256xf32>
  }
}
```

- Beam Creation: Variants dictated by scratch-pad allocations, register pressure, compute vs bandwidth tradeoffs etc.
- Beam Pruning: Cost model and heuristics to increase beam diversity

Fuser Compilation Flow (2/2)

```
func.func @fused_kernel_1A_bf16(%arg0: tensor<2x64x128xbf16>, %arg1: tensor<2x64x128xbf16>) ->
  %cst = arith.constant 5.000000e-01 : bf16
  %0 = syn.broadcast %cst : bf16 to tensor<2x64x128xbf16>
  %1 = syn.mult %arg0, %0 : tensor<2x64x128xbf16>
  %cst_0 = arith.constant 7.968750e-01 : bf16
  %2 = syn.broadcast %cst_0 : bf16 to tensor<2x64x128xbf16>
  %cst_1 = arith.constant 3.564450e-02 : bf16
  %3 = syn.broadcast %cst_1 : bf16 to tensor<2x64x128xbf16>
  %4 = syn.mult %arg0, %arg0 : tensor<2x64x128xbf16>
  %5 = syn.fma %4, %3, %2 {isNeg = false} : tensor<2x64x128xbf16>, tensor<2x64x128xbf16>
  %6 = syn.mult %arg0, %5 : tensor<2x64x128xbf16>
  %7 = syn.tanh %6 : tensor<2x64x128xbf16>
  %8 = syn.fma %1, %7, %1 {isNeg = false} : tensor<2x64x128xbf16>, tensor<2x64x128xbf16>
  %9 = syn.mult %arg1, %8 : tensor<2x64x128xbf16>
  return %7, %9 : tensor<2x64x128xbf16>, tensor<2x64x128xbf16>
}
```

```
func.func @fused_kernel_1A_bf16(%arg0: memref<2x64x128xbf16, 1>, %arg1: memref<2x64x12
  %cst = arith.constant 5.000000e-01 : bf16
  %cst_0 = arith.constant 7.968750e-01 : bf16
  %cst_1 = arith.constant 3.564450e-02 : bf16
  affine.for %arg4 = 0 to 128 {
    affine.for %arg5 = 0 to 2 {
      affine.for %arg6 = 0 to 64 {
        %0 = affine.load %arg0[%arg5, %arg6, %arg4] : memref<2x64x128xbf16, 1>
        %1 = arith.mulf %0, %cst : bf16
        %2 = arith.mulf %0, %0 : bf16
        %3 = tpc.macf %2, %cst_1, %cst_0 {isNeg = false} : bf16, bf16
        %4 = arith.mulf %0, %3 : bf16
        %5 = math.tanh %4 : bf16
        affine.store %5, %arg2[%arg5, %arg6, %arg4] : memref<2x64x128xbf16, 1>
        %6 = tpc.macf %1, %5, %1 {isNeg = false} : bf16, bf16
        %7 = affine.load %arg1[%arg5, %arg6, %arg4] : memref<2x64x128xbf16, 1>
        %8 = arith.mulf %7, %6 : bf16
        affine.store %8, %arg3[%arg5, %arg6, %arg4] : memref<2x64x128xbf16, 1>
      }
    }
  }
  return
}
```

Mostly upstream MLIR Dialects/Types: Affine, memref, arith, math

Op and Loop Fusion

Loop-based IR

Loop Optimizations Auto-Vectorization, Unroll, Interchange, Auto-parallelization, LICM etc.

Parallelized and Vectorized IR

TPC ISA Optimizations

Optimized LLVM IR

TPC Clang LLVM Backend

TPC ASM

```
func.func @fused_kernel_1A_bf16(%arg0: memref<2x64x128xbf16, 1>, %arg1: memref<2x64x128xbf16, 1>, %arg2: m
  %cst = arith.constant dense<7.968750e-01> : vector<128xbf16>
  %cst_0 = arith.constant dense<3.564450e-02> : vector<128xbf16>
  %cst_1 = arith.constant dense<5.000000e-01> : vector<128xbf16>
  %0 = tpc.get_index_space_start 2
  %1 = tpc.get_index_space_end 2
  %2 = tpc.get_index_space_start 1
  %3 = tpc.get_index_space_end 1
  affine.for %arg4 = %0 to %1 {
    affine.for %arg5 = %2 to %3 {
      %4 = affine.vector_load %arg0[%arg4, %arg5 * 4, 0] : memref<2x64x128xbf16, 1>, vector<128xbf16>
      %5 = affine.vector_load %arg0[%arg4, %arg5 * 4 + 1, 0] : memref<2x64x128xbf16, 1>, vector<128xbf16>
      %6 = affine.vector_load %arg0[%arg4, %arg5 * 4 + 2, 0] : memref<2x64x128xbf16, 1>, vector<128xbf16>
      %7 = affine.vector_load %arg0[%arg4, %arg5 * 4 + 3, 0] : memref<2x64x128xbf16, 1>, vector<128xbf16>
      %8 = arith.mulf %4, %cst_1 : vector<128xbf16>
      %9 = arith.mulf %5, %cst_1 : vector<128xbf16>
      %10 = arith.mulf %6, %cst_1 : vector<128xbf16>
      %11 = arith.mulf %7, %cst_1 : vector<128xbf16>
      %12 = arith.mulf %4, %4 : vector<128xbf16>
      %13 = arith.mulf %5, %5 : vector<128xbf16>
      %14 = arith.mulf %6, %6 : vector<128xbf16>
      %15 = arith.mulf %7, %7 : vector<128xbf16>
```

Parallelized

Vectorized

Loop Optimizations

Leverages many upstream affine optimizations and utilities

- Loop Fusion: **mlir::affine::fuseLoops**

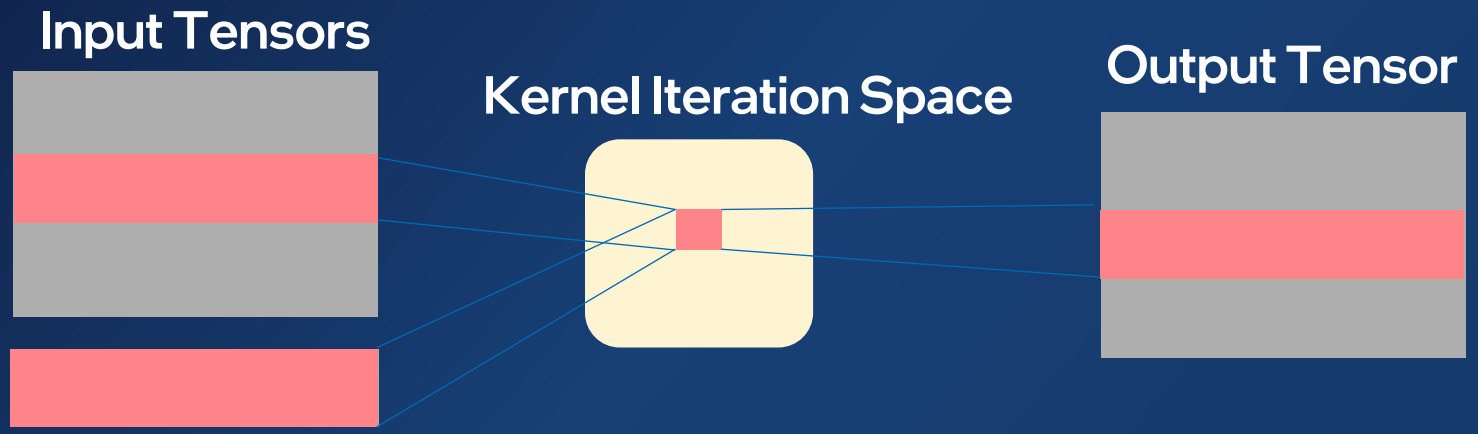
- Vectorization: SuperVectorize.cpp

- Unroll and Jam: loopUnrollJamByFactor

- Affine **Parallelization** to distribute iterations over TPCs

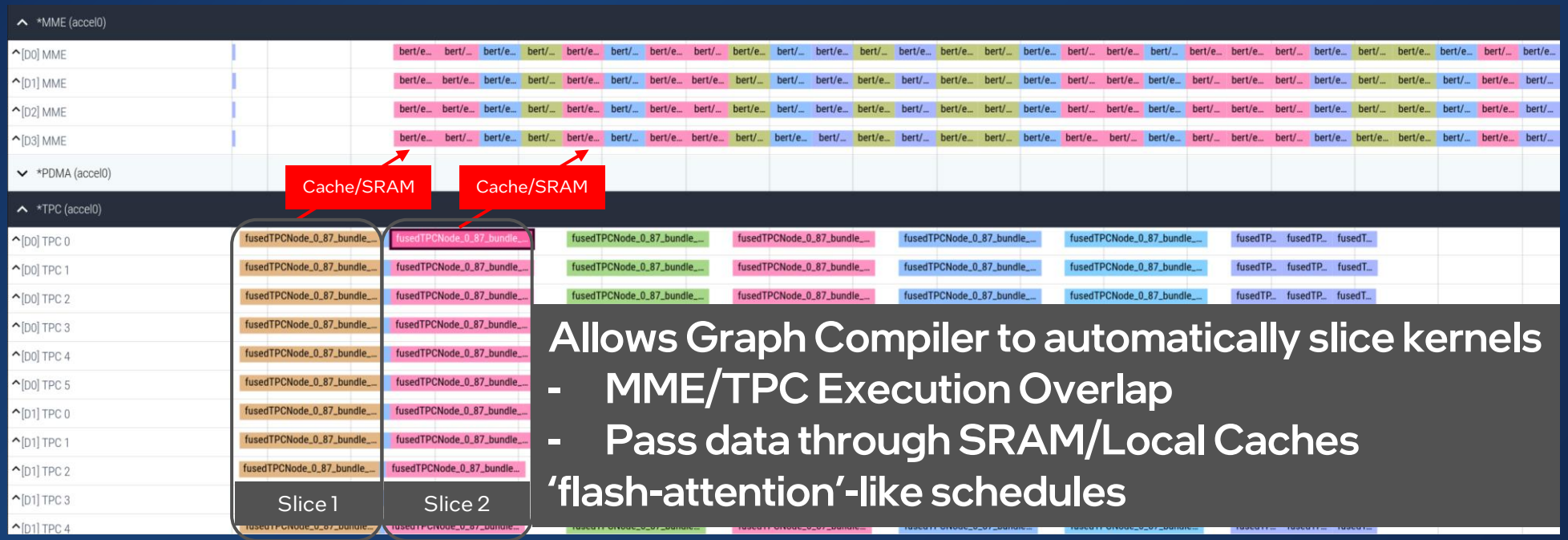
Upstreamed enhancements whenever feasible.
https://github.com/llvm/llvm-project/commit/14d0735d3453fb6403da916d7aee6a9f25af4147
https://github.com/llvm/llvm-project/commit/d80b04ab0015b218b613f8fe59506d45739817b8
https://github.com/llvm/llvm-project/commit/7ab14b8886d9ddaca1f8fc8a34ef8f03af208f26 etc.

Extracting Memory Access Information




```
func.func @fused_kernel_1A_bf16(%arg0: memref<2x64x128xbf16, 1
  %cst = arith.constant dense<7.968750e-01> : vector<128xbf16>
  %cst_0 = arith.constant dense<3.564450e-02> : vector<128xbf1
  %cst_1 = arith.constant dense<5.000000e-01> : vector<128xbf1
  %0 = tpc.get_index_space_start 2
  %1 = tpc.get_index_space_end 2
  %2 = tpc.get_index_space_start 1
  %3 = tpc.get_index_space_end 1
  affine.for %arg4 = %0 to %1 {
    affine.for %arg5 = %2 to %3 {
      %4 = affine.vector_load %arg0[%arg4, %arg5 * 4, 0] : mem
      %5 = affine.vector_load %arg0[%arg4, %arg5 * 4 + 1, 0]
      %6 = affine.vector_load %arg0[%arg4, %arg5 * 4 + 2, 0]
      %7 = affine.vector_load %arg0[%arg4, %arg5 * 4 + 3, 0]
```

Affine maps extracted through analysis of affine memory operations

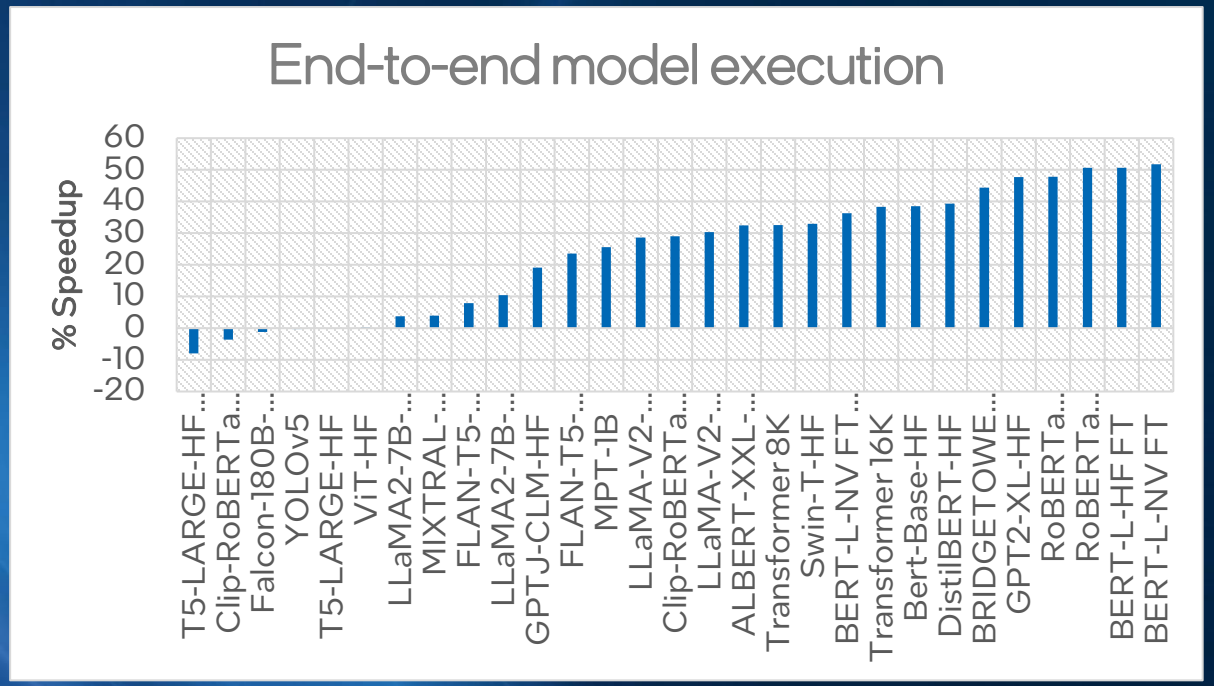


Allows Graph Compiler to automatically slice kernels

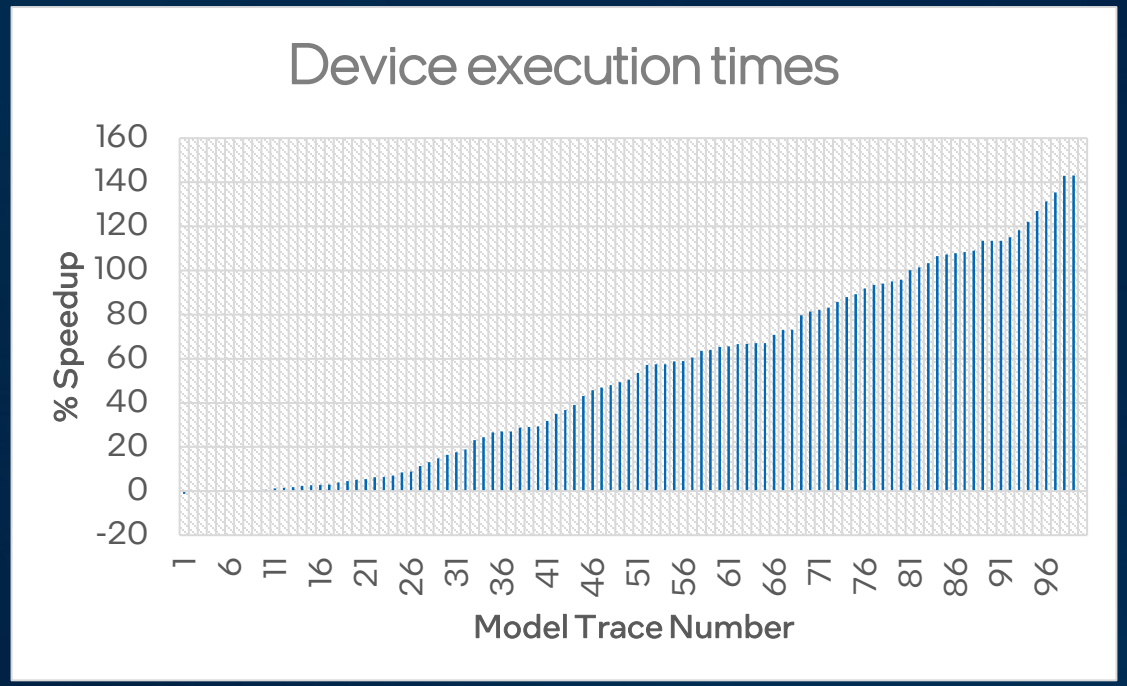
- MME/TPC Execution Overlap
- Pass data through SRAM/Local Caches

‘flash-attention’-like schedules

Fuser Performance Improvements



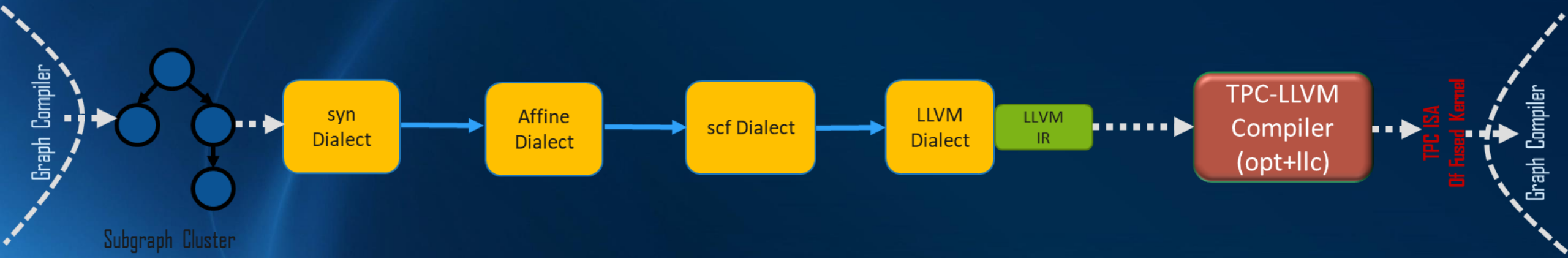
1.3X Avg Perf Improvement at model level



1.5X Avg Perf Improvement in device execution time

Measured on Intel Gaudi2

Conclusion



- 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
- Leverages upstream MLIR dialects like Affine, SCF, Arith, Math along with in-house dialects


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.