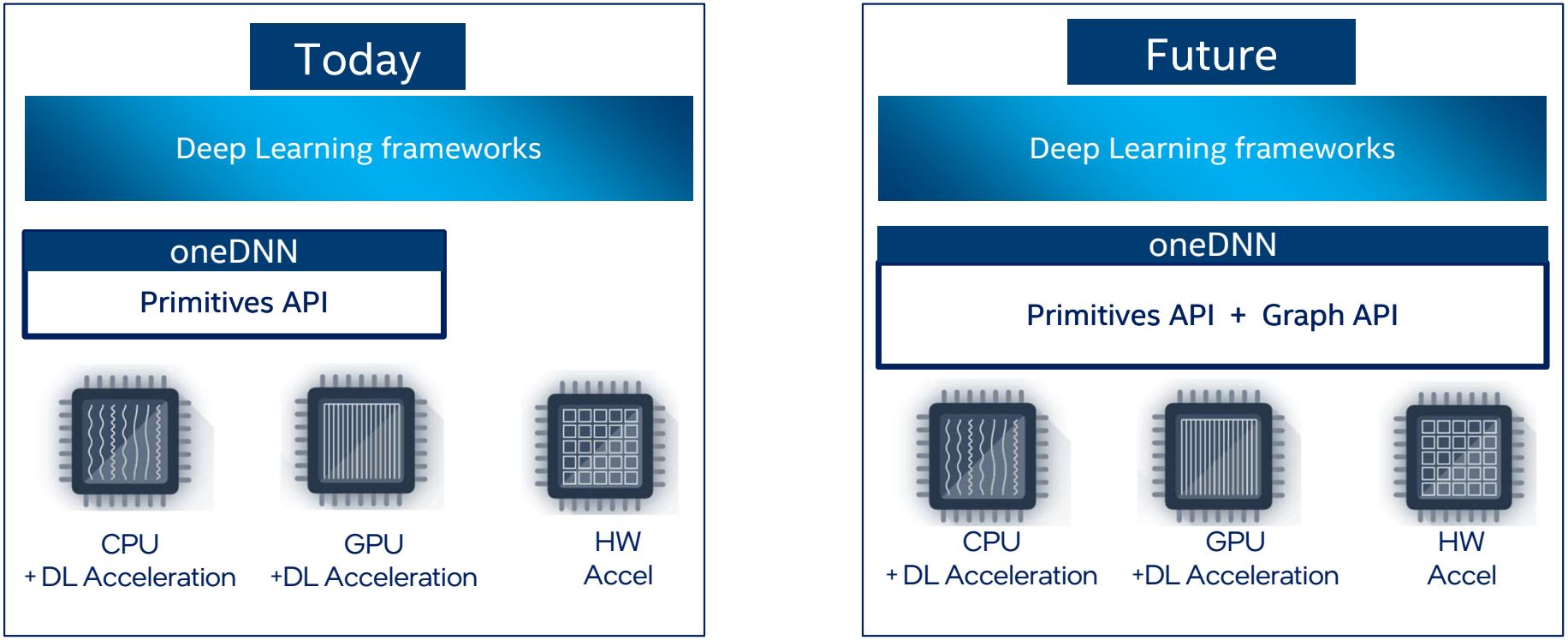
Generating Efficient Code for Deep Neural Network

Jianhui Li
Software and Advanced Technology Group

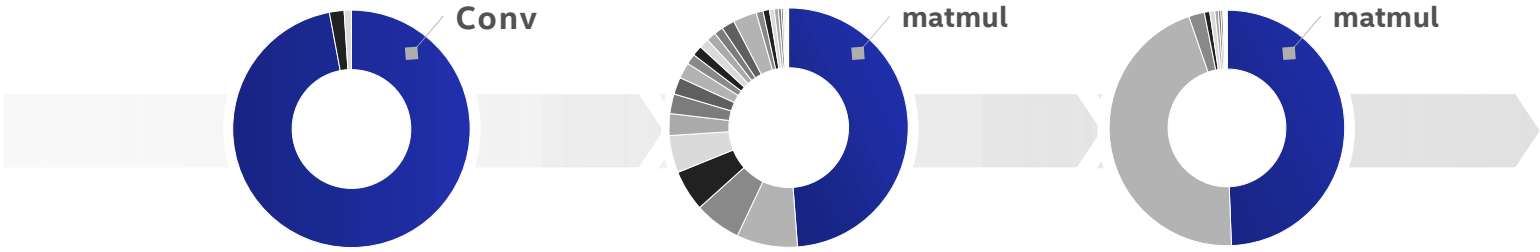
oneDNN is evolving...



- Graph API allows HW backend to achieve high efficiency
- Same integration for multiple AI HW: CPU, GPU, and accelerators

The driving forces of AI Optimization

Diversifying AI application

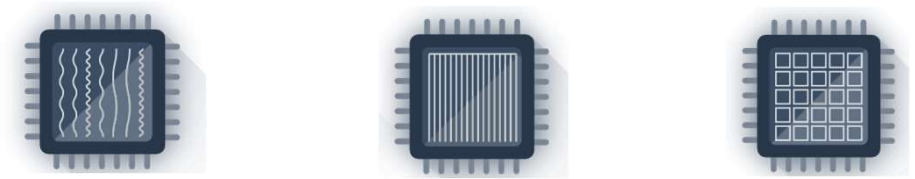


Computer Vision

Natural Language Processing

Recommendation Engine

Hardware Acceleration for AI

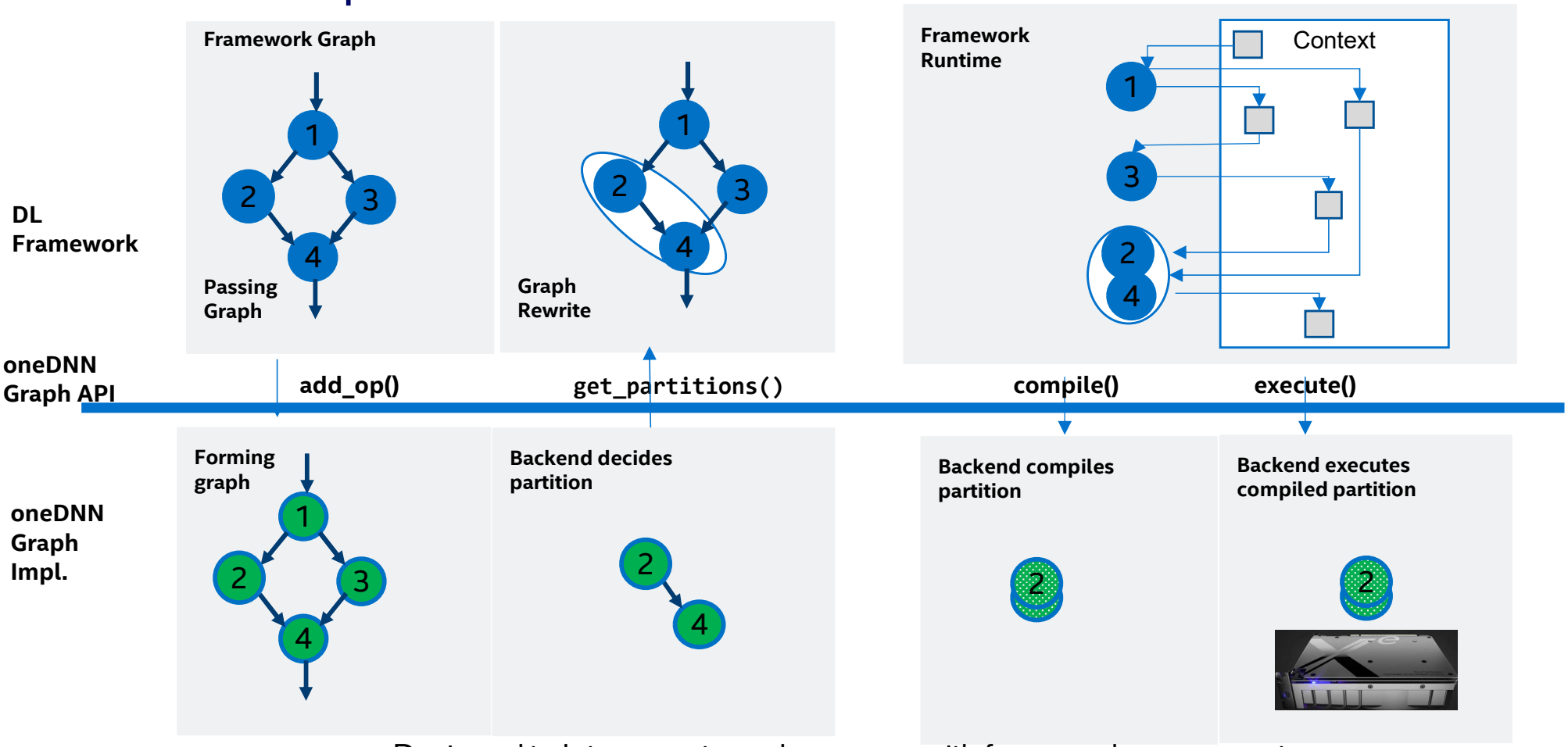


CPU
+ DL Acceleration

GPU
+DL Acceleration

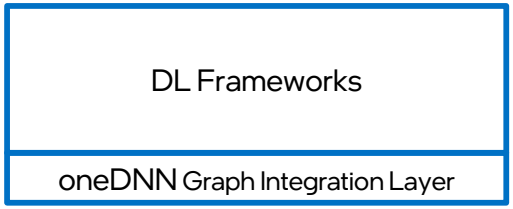
Accelerators

oneDNN Graph API



Designed to Interoperate and compose with framework components

Deep learning with oneDNN Graph



oneDNN Graph API

Partition | Compilation | Execution

Code Gen Algorithm selection

oneDNN Graph Implementation

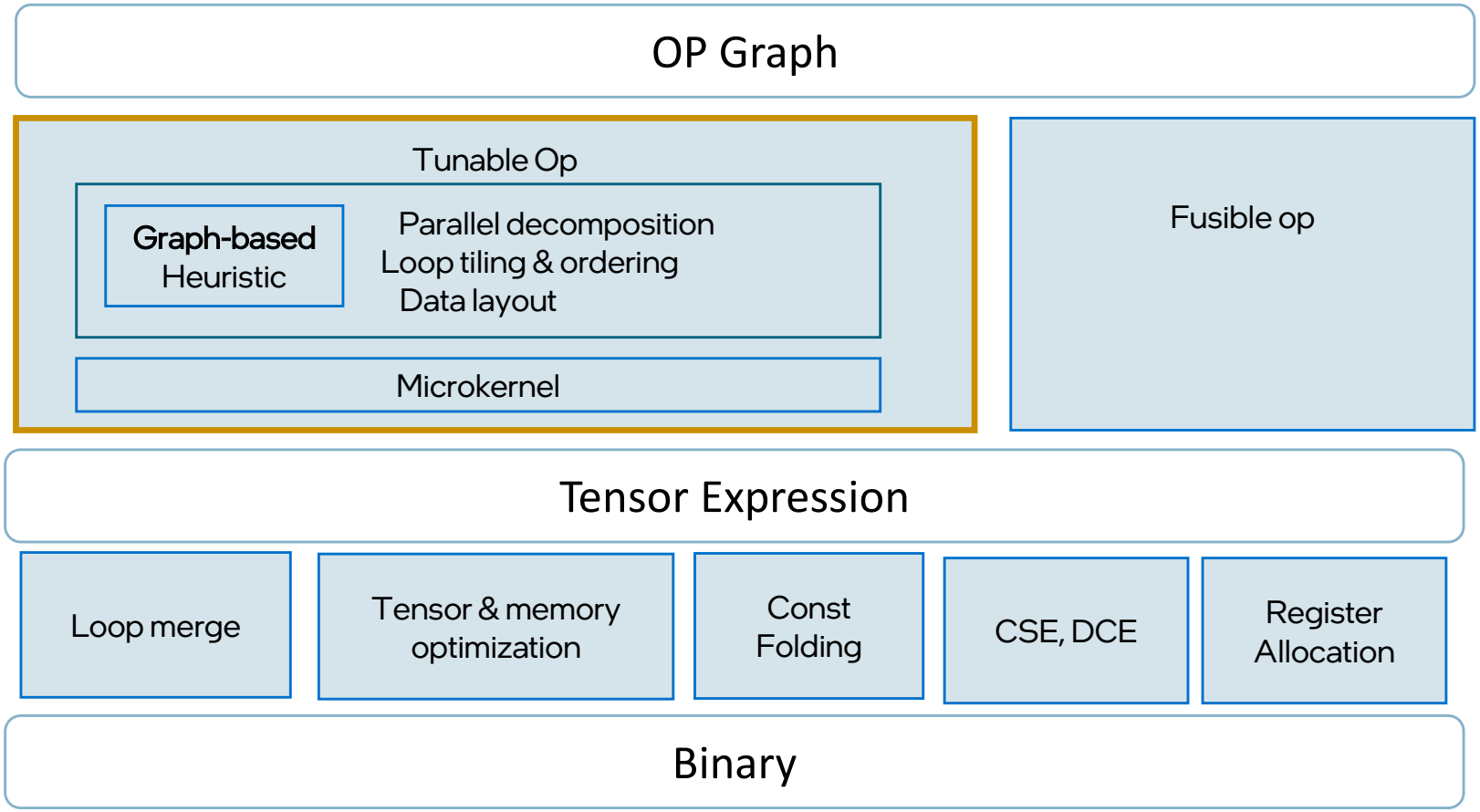
Primitives + Post-Op Fusions

oneDNN Graph Compiler

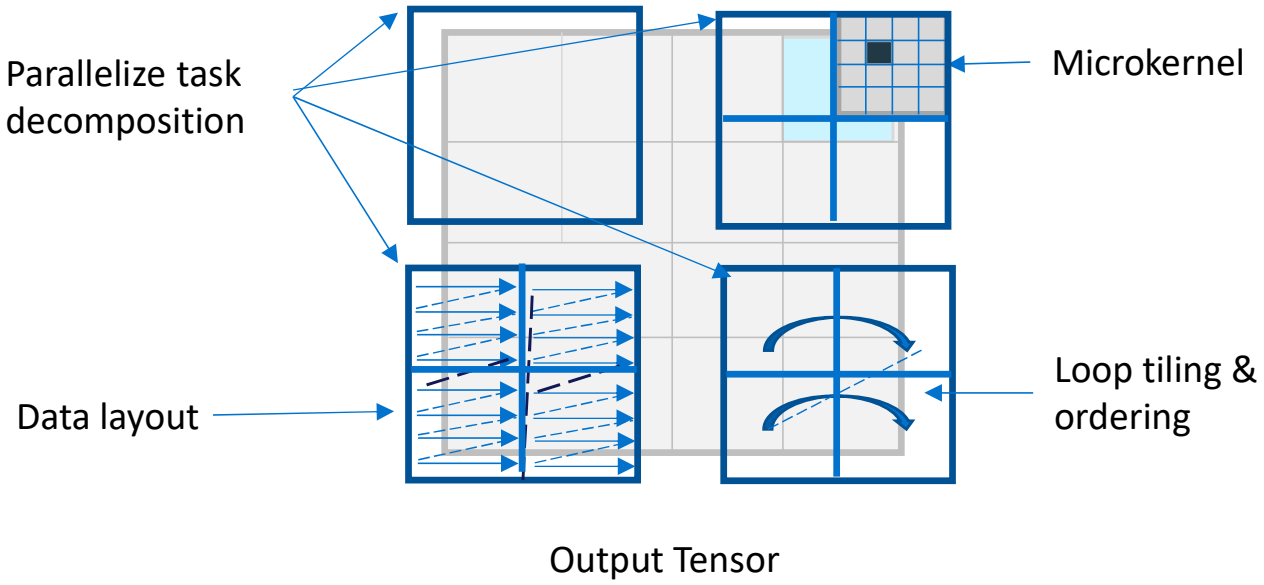
Micro kernel

low-level graph compiler focusing on code generation for performance-critical DNN graph partition

oneDNN Graph Compiler

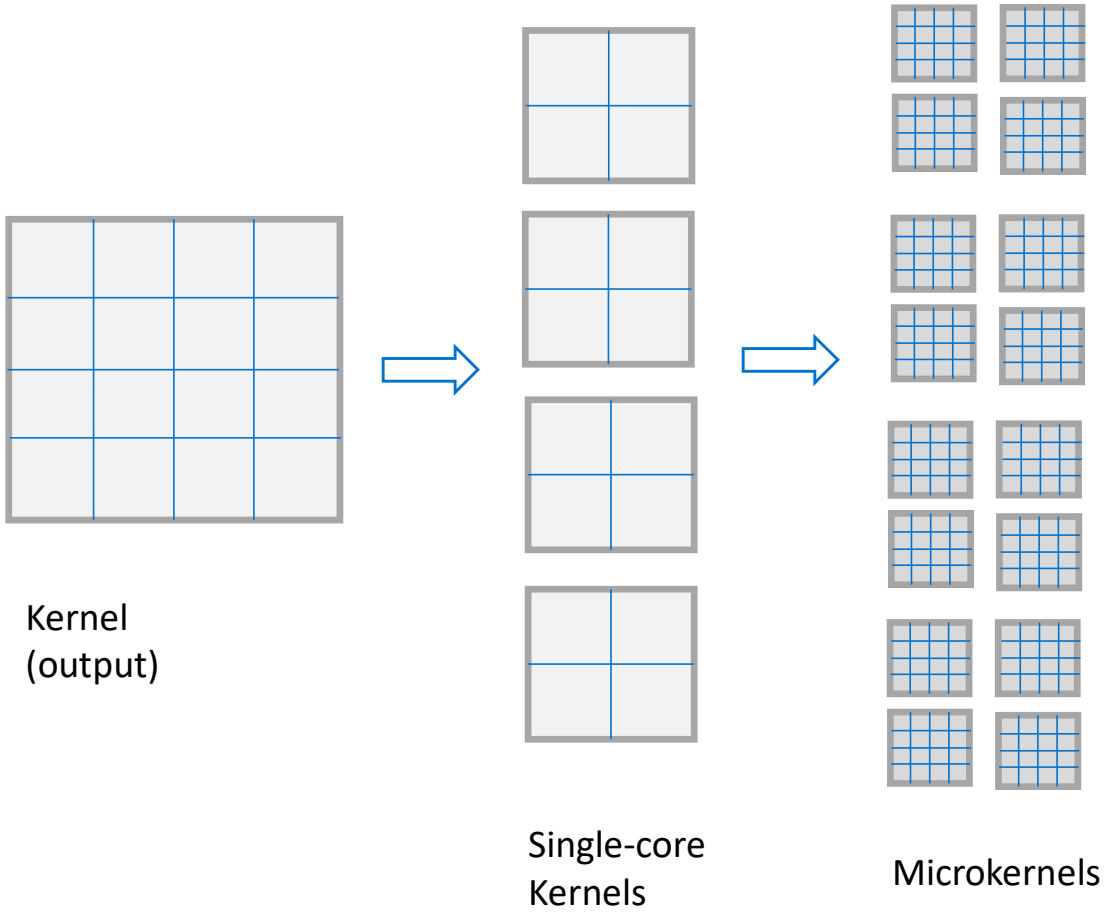


Recipe for efficient kernel



Tunable hyperparameters	Hardware efficiency
Single-core task size	Multi-Core
Microkernel size	Vector/Matrix
Loop tiling size Loop order	Cache/TLB
Data blocking factor	Cache/TLB

Parallel task decomposition

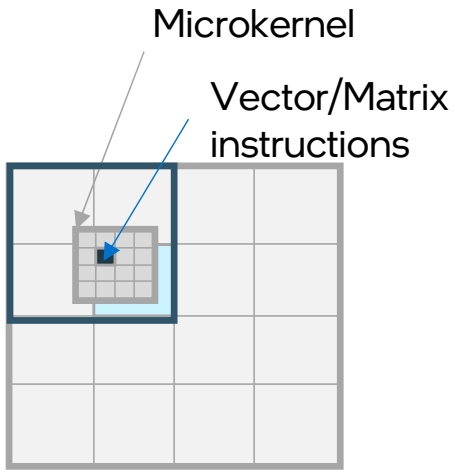


Propose single-core kernel size options, using all cores with balanced load

Propose options of microkernel sizes with high vector/matrix utilization

Search for a pair of single kernel size and microkernel size based on a cost model

Microkernel



$$C[M,N] = \sum_{i=0}^{batch} A_i[M,K] * B_i[K,N]$$

Batch-Reduce gemm interface

High-Performance Deep Learning via a Single Building Block, Evangelos Georganas, Alexander Heinecke

Microkernel code example

$$C[32,32] = \sum_{i=0}^{16} A_i[32,32] * B_i[32,32]$$

```
Batch_reduce_gemm(A[0:32, 0:32],
                  B[0:32, 0:32],
                  C[0:32, 0:32],
                  batch = 16)
{
  tileloaddt1(tmm0, C);
  tileloaddt1(tmm1, C + 64);
  tileloaddt1(tmm2, C + 2048);
  tileloaddt1(tmm3, C + 2112);
  for (int i = 0; i < batch; i++) {
    tileloaddt1(tmm4, A);
    tileloaddt1(tmm5, A + 1024);
    tileloaddt1(tmm6, B);
    tdbpf16ps(tmm0, tmm4, tmm6);
    tdbpf16ps(tmm2, tmm5, tmm6);
    tileloaddt1(tmm7, B + 64);
    tdbpf16ps(tmm1, tmm4, tmm7);
    tdbpf16ps(tmm3, tmm5, tmm7);
    A += 1024; B += 1024;
  }
  tilestored(C, tmm0);
  tilestored(C + 64, tmm1);
  tilestored(C + 2048, tmm2);
  tilestored(C + 2112, tmm3);
}
```

Specialized for specific tensor shape

Aligned with HW vector/matrix size

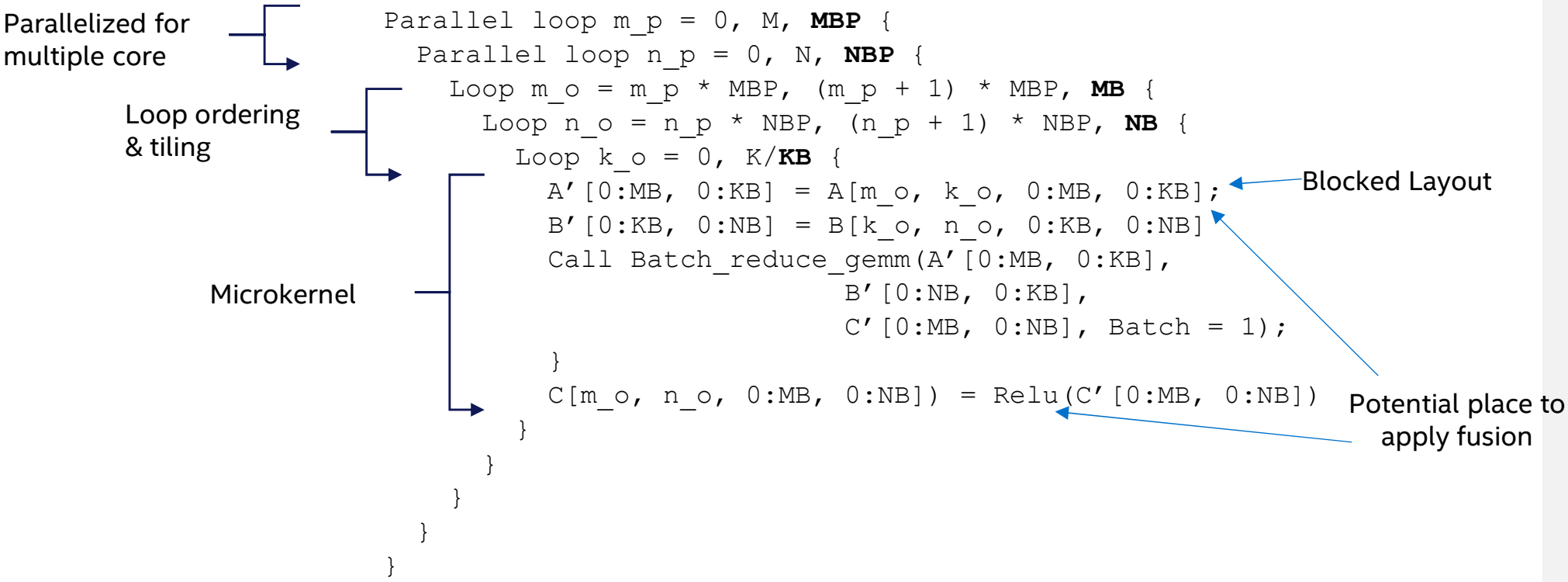
Unroll, software pipelining, prefetching

Vectorize/Tensorize

Careful and "manual" register blocking and allocation

Based on carefully hand-crafted code template

Synthesized kernel for fused op



Good heuristic required to decompose to single-core kernel and microkernel, loop order, and data layout

Compile subgraph by composing kernels

matmul
relu

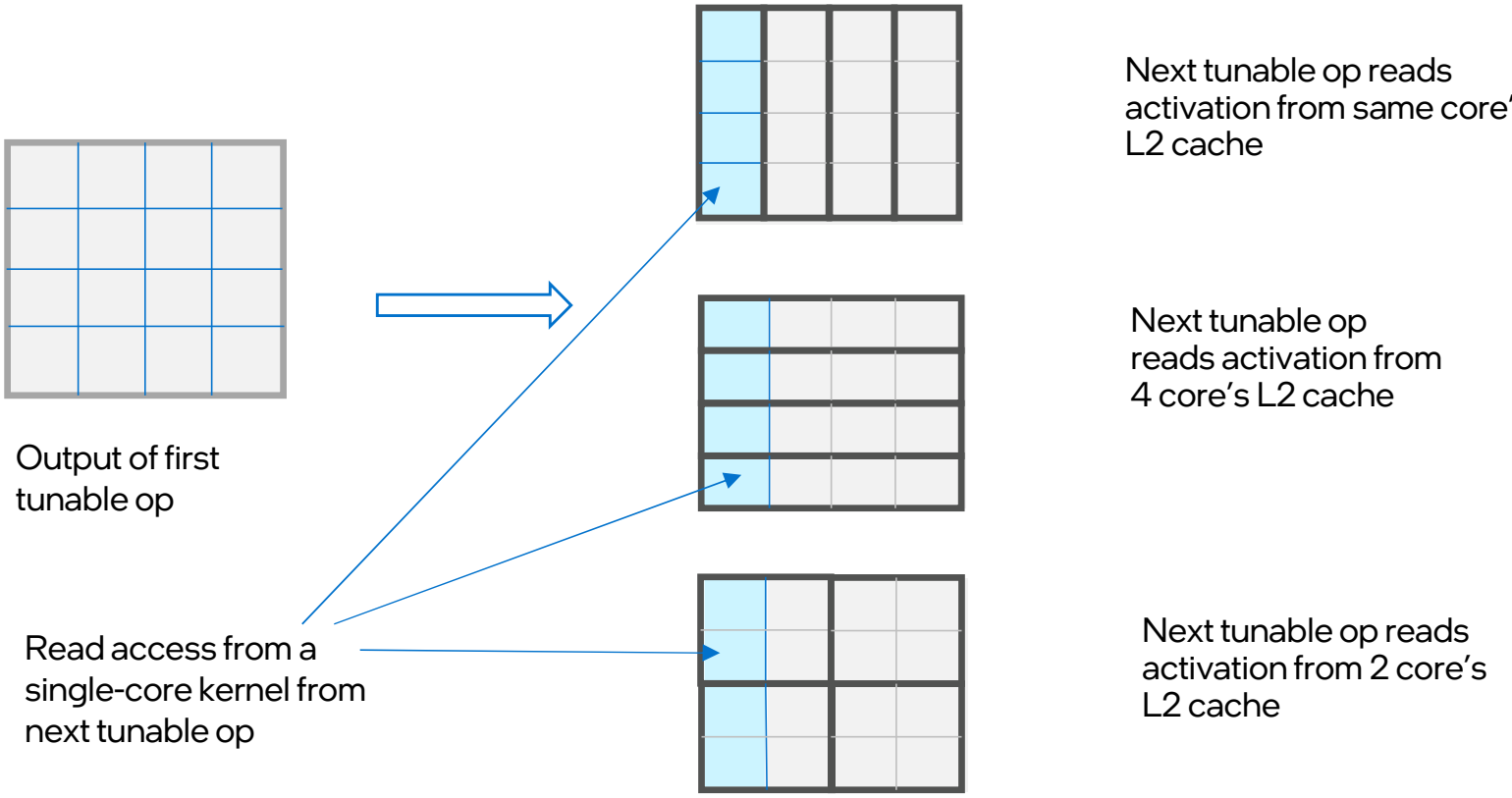
```
C – Output tensor, [M][N]
A – Input tensor (activations) [M][K]
B – Input tensor (weights) [K][N]

Parallel loop m_p = 0, M, MBP {
  Parallel loop n_p = 0, N, NBP {
    Loop m_o = m_p * MBP, (m_p + 1) * MBP, MB {
      Loop n_o = n_p * NBP, (n_p + 1) * NBP, NB {
        Loop k_o = 0, K/KB {
                A'[0:MB, 0:KB] = A[m_o, k_o, 0:MB, 0:KB];
          B'[0:KB, 0:NB] = B[k_o, n_o, 0:KB, 0:NB]
          Call Batch_reduce_gemm(A'[0:MB, 0:KB],
                                 B'[0:NB, 0:KB],
                                 C'[0:MB, 0:NB]);
          }
          C[m_o, n_o, 0:MB, 0:NB]) = relu ( C'[0:MB, 0:NB])
        }
      }
    }
  }
}
```

```
Parallel loop m_p = 0, M, MBP {
  Parallel loop n_p = 0, N, NBP {
    Loop m_o = m_p * MBP, (m_p + 1) * MBP, MB {
      Loop n_o = n_p * NBP, (n_p + 1) * NBP, NB {
        Loop k_o = 0, K/KB {
          A'[0:MB, 0:KB] = A[m_o, k_o, 0:MB, 0:KB];
          B'[0:KB, 0:NB] = B[k_o, n_o, 0:KB, 0:NB]
          Call Batch_reduce_gemm(A'[0:MB, 0:KB],
                                 B'[0:NB, 0:KB],
                                 C'[0:MB, 0:NB]);
          }
          C[m_o, n_o, 0:MB, 0:NB]) = relu ( C'[0:MB, 0:NB])
} } } } }
```

```
Parallel loop m_p = 0, M, MBP {
  Parallel loop n_p = 0, N, NBP {
    Loop m_o = m_p * MBP, (m_p + 1) * MBP, MB {
      Loop n_o = n_p * NBP, (n_p + 1) * NBP, NB {
        Loop k_o = 0, K/KB {
          A'[0:MB, 0:KB] = A[m_o, k_o, 0:MB, 0:KB];
          B'[0:KB, 0:NB] = B[k_o, n_o, 0:KB, 0:NB]
          Call Batch_reduce_gemm(A'[0:MB, 0:KB],
                                 B'[0:NB, 0:KB],
                                 C'[0:MB, 0:NB]);
          }
          C[m_o, n_o, 0:MB, 0:NB]) = relu ( C'[0:MB, 0:NB])
} } } } }
```

matmul
relu
↓
matmul
relu
↓
matmul
relu

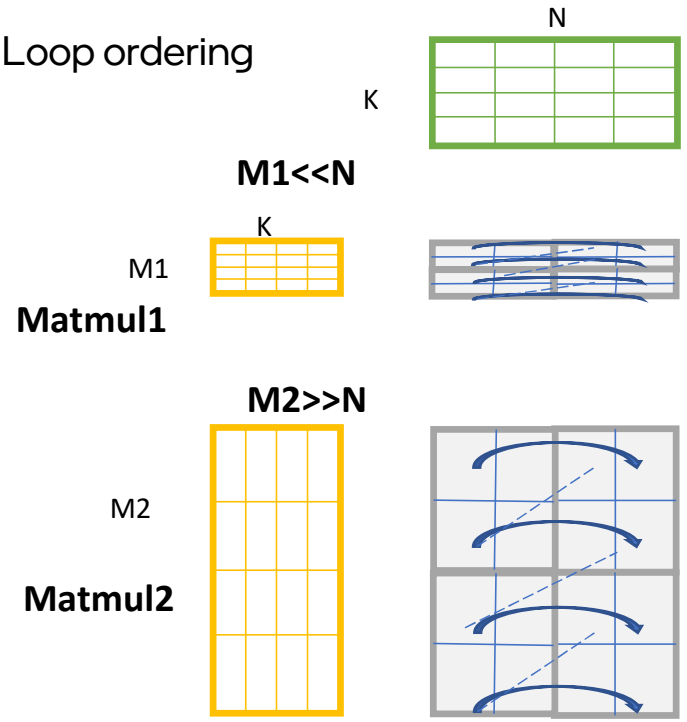
Parallel task decomposition within Graph context



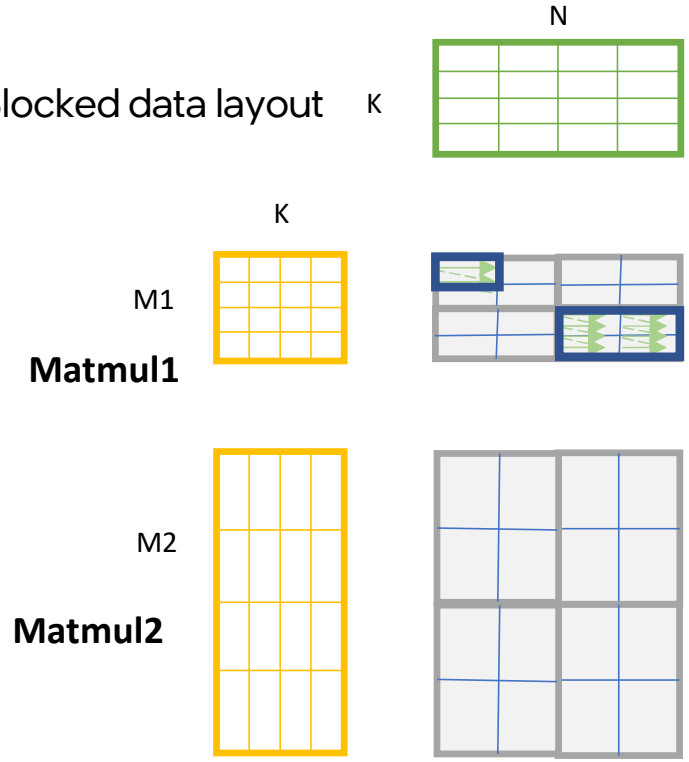
Graph-based heuristic considers the cost of reading activation data from last tunable op

Graph-based heuristic for Single-core Kernel

Loop ordering



Blocked data layout

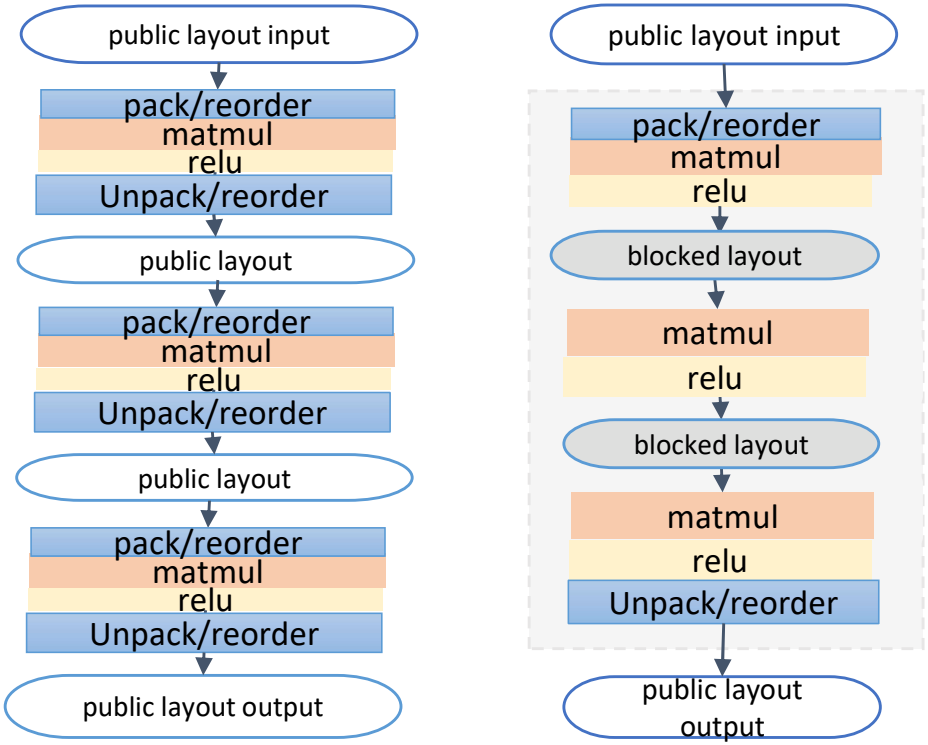


Graph knows input/weight is “hot” or “cold”

Input stationary preferred as weight is usually “cold”

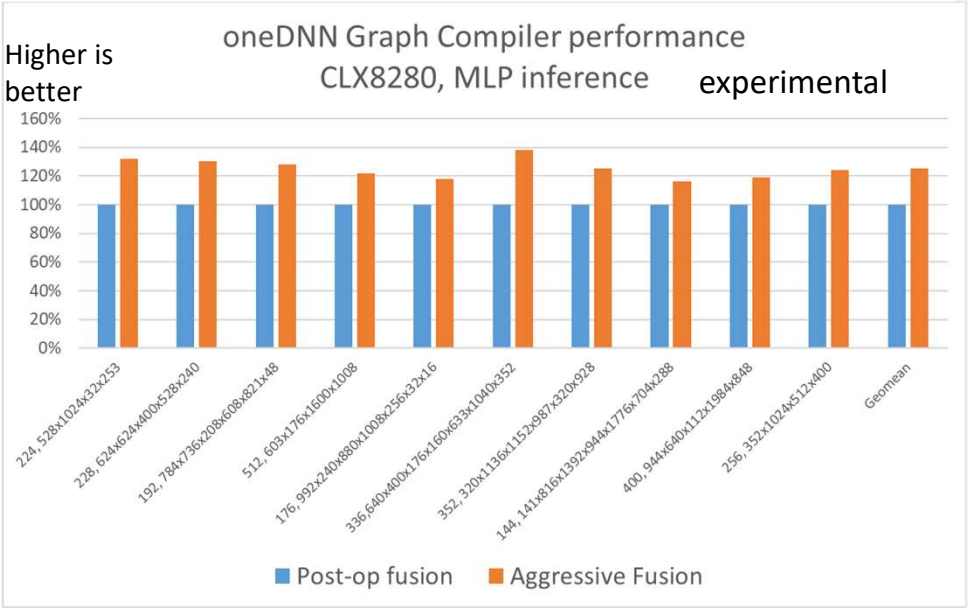
Choose blocked layout works best for overall performance

Aggressive Fusion



Post-op fusion

Aggressive fusion



Aggressive fusion covers MLP, MHA, Conv Block, both inference and training, fp32, int8, and bf16 on CPU

Substantial performance gains observed

Summary

oneDNN Graph API offers flexible API to accelerate DNN Graph with partitioning

oneDNN Graph compiler merges nested loops of tuable op and fusible op, compose a large kernel for aggressive fusion, and Automate kernel generation

Recipes from hand-tuned kernel, micro-kernel, and blocked layout

Graph-based heuristic further improves kernel performance within graph context

Notices and Disclaimers

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

No product or component can be absolutely secure.

Your costs and results may vary.