oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation

Jianhui Li, Zhennan Qin, Yijie Mei, Jingze Cui, Yunfei Song, Ciyong Chen, Yifei Zhang, Longsheng Du, Xianhang Cheng, Baihui Jin, Jason Ye, Eric Lin, Dan Lavery

Software and Advanced Technology Group, Intel

Abstraction

With the rapid development of deep learning models and hardware support for dense computing, the deep learning (DL) workload characteristics changed significantly from a few hot spots on compute-intensive operations to a broad range of operations scattered across the models. Accelerating a few compute-intensive operations using the expert-tuned implementation of primitives doesn't fully exploit the performance potential of AI hardware. Various efforts are made to compile a full deep neural network (DNN) graph. One of the biggest challenges is to achieve end-to-end compilation by generating expert-level performance code for the dense compute-intensive operations and applying compilation optimization at the scope of DNN computation graph across multiple compute-intensive operations.

We present oneDNN Graph Compiler, a tensor compiler that employs a hybrid approach of using techniques from both compiler optimization and expert-tuned kernels for high-performance code generation of the deep neural network graph. oneDNN Graph Compiler addresses unique optimization challenges in the deep learning domain, such as low-precision computation, aggressive fusion, optimization for static tensor shapes and memory layout, constant weight optimization, and memory buffer reuse. Experimental results demonstrate up to 2x performance gains over primitives-based optimization for performance-critical DNN computation graph patterns on Intel® Xeon® Scalable Processors.

Introduction

With the vigorous development of AI applications, deep learning software stacks and hardware are rapidly evolving. Data Scientist is continuously exploring new deep neural network (DNN) models to improve the accuracy of the models by increasing the model parameters, using larger training datasets, and exploring innovative DNN structures and operations. Deep learning frameworks, like TensorFlow[3] and PyTorch[4], are developed to support the development and deployment of deep learning models. While supporting Data Scientists to develop new models, DL frameworks also need to efficiently use hardware resources to meet the huge computing needs of deep learning. Meanwhile, various hardware supports for deep learning have been introduced, including adding matrix operation units on GPU and CPU, and hardware accelerators dedicated to deep learning computation.

Deep learning (DL) frameworks provide a rich set of deep neural network (DNN) operations for developers to describe a DNN model and use primitives libraries by default to offload the most performance-critical operations to CPU and GPU [1][2][20]. Most of the execution time of DL applications is spent on the DNN model. DL frameworks represent the DNN models internally as a computation graph of DNN operations. After performing high-level graph optimizations, the graph is executed operation by operation. On top of their own implementation of DNN ops, DL frameworks use third-party primitives libraries to offload the most performance-critical DNN operations.

Primitives library offers a simple and effective way to offload deep learning computation. However, its performance benefit doesn't scale to new AI workloads and hardware due to its limited optimization scope. With the fast evolution of AI software and hardware, the performance characteristics of deep learning workload have been shifted from a few hot spots of concentrated compute-intensive operations to many scattered DNN operations including memory-bound operations. There are multiple sources that contribute to the increasing time percentage on memory-bound operations. The deep neural network used for natural language processing [19] and recommendation systems [18] has smaller input data and overall lower compute intensity compared to computer vision models [22] [23]. Instead of supporting each innovative activation function with a complex DNN operation, DL Frameworks tend to compose multiple existing fine-grain operations, to maintain a balance of scalability and usability. Lastly, hardware acceleration usually focuses on accelerating the dense computation of

low-precision data types and relies on the software to optimize memory-bound operations.

The performance characteristic shift drives the development of low-level graph compilers in the deep learning domain also known as tensor compilers [5][6] [7] [10] [15] [17]. The tensor compilers focus on optimization techniques specific to the deep learning workload, aiming at generating highly efficient code for a DNN computation graph. Tensor compilers view DNN operations as tensor computation and internally represented as nested multi-level loops with the innermost loop body processing each tensor element. It uses compiler loop transformation techniques to parallelize, vectorize, reorder, and merge the nested loops.

It has been a hot research area on how to automatically generate code for compute-intensive primitives and achieve high performance comparable to expert-tuned implementation [9] [14] [16][21] [25] [26] [27] [28]. Due to the inherent complexity of nested loop transformation, some research uses the autotune method to search for a good solution in a large search space, and some use analytical models to determine optimal tuning parameters for generating high-performance primitives. The tensor compilation generated code is mainly used in specific use cases where the expert-tuned primitives library can't offer the required performance and the users are willing to spend extra resources to find a better solution very often via autotuning.

oneDNN Graph Compiler is an open-source tensor compiler that automates the code generation for compute-intensive DNN operations like matrix multiplication and achieves the same level of computing efficiency as primitives library implementation [1][11][12][13]. Based on that, it further explores more advanced optimization for a subgraph with multiple compute-intensive DNN operations, such as optimizing the whole Multilayer Perception (MLP) network construct containing multiple matrix multiplication operations.

oneDNN Graph Compiler applied domain-specific expert knowledge that was distilled from the expert-tuned kernel development process to an automated compilation process and achieved performance in parity. It combines compiler and kernel library techniques and focuses on domain-specific optimization problems. With expert-tuned microkernels and two levels of compiler IR, oneDNN Graph Compiler addresses domain-specific optimization challenges, such as generating efficient code for the compute-intensive kernels of static tensor shapes with blocked memory layout, constant weight optimization, aggressive fusion, and memory buffer reuse. Experimental results show that oneDNN Graph Compiler delivers significant performance gains over primitive-based optimization for performance-critical DNN computation graph patterns on CPU.

High-level Design

The principal of oneDNN Graph Compiler high-level design is performance, simplicity, and modularity. oneDNN Graph Compiler has two levels of intermediate representations: Graph IR, and Tensor IR. The input DNN computation graph is internally represented as Graph IR. The Graph IR optimization module performs a number of transformations that optimize and group the computation graph as a sequence of fused operations. Graph IR is further lowered to Tensor IR. The Tensor IR doesn't preserve DNN operation semantics and is close to the C program semantics. The data structure it operates on is multidimensional arrays, representing tensor buffers in physical memory. Tensor IR is then further lowered to LLVM IR and intrinsic calls to Microkernels.

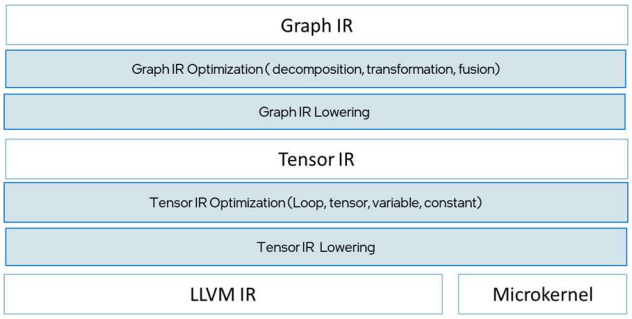


Figure 1. oneDNN Graph Compiler IR and Optimization

The Graph IR keeps the DNN OP semantics, so most domain-specific optimizations are done at this level. Instead of lowering the DNN OP to Tensor IR and performing sophisticated loop analysis to achieve the best loop schedule and fusion, oneDNN Graph Compiler uses expert-developed templates, microkernels, and heuristics to guide the code generation of compute-intensive operations and the fusion process. The decisions of parallel task decomposition, loop scheduling and tiling, tensor memory layout, and whether to fuse with neighbor operations are based on the Graph IR with DNN OP semantics. This simplifies Tensor IR design. The Tensor IR supports mechanically lowering Fused OP to nested loop, but there is no need to support sophisticated loop analysis and complex nested loop transformation. Tensor IR optimization mainly focuses on tensor buffer optimization and supports low-level code generation. The use of compilation techniques, including Graph IR and Tensor IR, helps to increase the optimization scope from individual primitives to a larger subgraph with multiple compute-intensive operations.

Graph IR uses graph, logical tensor, and OP to describe a computation graph. A graph contains a set of OPs and logical tensors. Each OP represents an operation in a computation graph. A logical tensor represents the tensor's metadata, like the element's data type, shape, and memory layout. OP has kind, category, attributes, and logical tensors for inputs and outputs.

Graph IR optimization module first decomposes complex OPs into basic DNN OPs. The complex DNN OPs are OPs with complex semantics which could be composed of simple fundamental operations like addition and multiplication. They are introduced by DL frameworks to support high-level DNN OP semantics for ease of programming, such as batchnorm, quantize, gelu, and many activation operations. The basic DNN OPs are categorized to be either Tunable OP or Fusible OP. Tunable OPs describe DNN operations that use tunable parameters to instantiate a pre-defined template to generate the best-performing code. The example includes compute-intensive operations like matmul. Fusible OP refers to operations that can be fused to Tunable OPs, such as element-wise operations, broadcast, reduction, and data movement operations.

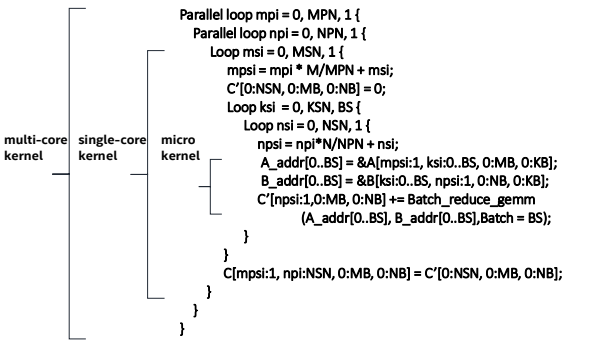
The decomposition of complex DNN operations simplifies the Graph IR optimization module so it only needs to handle basic DNN operations. Besides the general compiler optimizations like common subexpression elimination (CSE), dead code elimination, and constant folding, it includes domain-specific optimizations like low-precision conversion, tensor memory layout propagation, constant weight preprocessing, and fusion. The fusion optimization pass decides whether it is profitable to fuse two operations and keeps fusing OPs to form a subgraph, which is represented as a Fused OP. The Graph IR is transformed into a graph of Fused OPs.

The Graph IR is then lowered into Tensor IR. Just like the C program, Tensor IR supports function, statement, expression, and intrinsic functions. The Tensor IR module, lowered from a Graph IR graph, contains multiple functions, each of which represents a lowered Fused OP. The Tensor IR module has an entry function that contains a sequence of calls to other functions lowered from Fused OPs. A Tensor IR function contains multiple statements build on expressions, which operate on constants, variables, and tensors. Constants and variables represent individual data elements, used to represent scalar data like loop index, tensor shape, address and offset to tensor buffer. Tensors represent multi-dimension arrays backed by a data buffer. The intrinsic function is used to represent a microkernel, which is carefully hand-tuned and fulfills a subtask of a DNN OP with data in the fastest cache on a single CPU core.

This paper describes the most effective techniques used in the oneDNN Graph Compiler. To build a high-quality tensor compiler, there are many details to consider at the implementation level. We first explain the foundation, using templates to guide the lowering of Tunable OP and Fused OP. Then we describe the Graph IR optimization which maximizes the fusion opportunity to form a large and profitable Fused OP and low-level Tensor IR optimization for efficient code generation.

Microkernel-Based Template for Tunable OP Lowering

Automating the high-performance code generation for Tunable OPs is the foundation of a tensor compiler. oneDNN Graph Compiler took an approach inherited



	m	n	k
Index of single-core kernel within multi-core kernel	mpi	npi	kpi
Number of single-core kernels within multi-core kernel	MPN	NPN	KPN
Index of microkernel within single-core kernel	msi	nsi	ksi
Number of microkernel within single-core kernel	MSN	NSN	KSN/BS
Index of microkernel within multi-core kernel	mpsi	npsi	kpsi
Number of microkernel within multi-core kernel	MPSN	NPSN	KPSN/BS
Tensor size	M	N	K
Tensor block size	MB	NB	KB
Tensor slice size accessed by Microkernel size (batch size = BS)	MB	NB	KB * BS
Tensor slice size accessed by single-core kernel	MSBN = MB * MSN	NSBN = NB * NSN	KSBN = KB * KSN
Tensor slice size accessed by multi-core kernel	M = MB * MSN * MPN = MB * MPSN	N = NB * NSN * NPN = NB * NPSN	K = KB *KSN * KPN = KB * KPSN

Tensor is described with a Tensor name followed by index and size for each dimension. Tensor A[0:M, 0:K] refers to 2 dimensions tensor starting from the position [0,0] with size [M, K]. A[0:MB, 0:KB] refers to a tensor slice containing a subset of A tensor elements, starting from position 0 to MB-1 along the m dimension, and 0 to NB-1 along the n dimension. The pseudo-code uses a blocked layout for A, B, and C. C[0:MPSN, 0:NPSN, 0:MB, 0:NB] denotes the full C tensor C[0:M, 0:N] reordered with a blocked layout. C[mps:1, np:NSN, 0:MB, 0:NB] denotes a tensor slice which "slice" the C tensor in the first 2 dimensions starting from position "mps" and "np" with size "1" and "NSN". A_addr[0..BS] denotes an array with BS elements from A_addr[0] to A_addr[BS-1]. A[mps:1, ks:0..BS, 0:MB, 0:KB] denotes an array of BS tensor slices from A[mps:1, ks:0, 0:MB, 0:KB] to A[mps:1, ks:BS-1, 0:MB, 0:KB].

Figure 2. Microkernel based template for Tunable OP

from the performance library development, which first creates the code templates for a given Tunable OP and then instantiates it with parameters decided by a heuristic. The parameters are decided based on the input data tensor shape and hardware sizes of the microarchitecture.

The template shown above is for a matmul op that does matrix multiplication over A[M, K] and B[K, N] and produces C[M, N]. The template is applied to a common deep learning use case where the computation uses multiple cores, and the size of input and output tensor fits within the cache system. The outer parallel loops divide the kernel into multiple subtasks for multi-cores. Each subtask is assigned to one single core, named single-core kernel, which is represented by the inner loops which call a microkernel in the innermost loop body.

The microkernel and the single-core kernel operate on a tensor slice that represents a subset of tensor elements. For example, the original tensor is represented as A[0:M, 0:N], where the subscription represents starting offset and size for each dimension. The tensor slice is represented as A[0:MB, 0:NB], where MB and NB refer to the tile size of the tensor slice along m and n dimensions. A submatrix is a special case of a 2-dimension tensor slice. In the template above, the microkernel produces a small submatrix C[0:MB, 0:NB], and the single-core kernel outputs a larger submatrix C[0:MSN, 0:NSN].

The microkernel is an important element for the oneDNN Graph Compiler to achieve comparable performance to expert-tuned primitives. oneDNN Graph Compiler uses the microkernel named batch-reduce GEMM [8][24]. The microkernel has two inputs, both representing a batch of 2D matrices. It first applies matrix multiplication with each batch element to produce a batch of immediate 2D matrices and then sums them to a final 2D matrix output. This interface can be used for many variants of matmul op in both inference and training use cases and was adopted by both oneDNN primitives and oneDNN Graph Compiler.

The microkernel is fine-tuned to maximize the compute efficiency by fully utilizing the compute function unit and the high bandwidth provided by registers and the L1 cache. It abstracts the ISA difference so oneDNN Graph Compiler doesn't need to deal with different vector or matrix instructions provided by different CPUs. However, the oneDNN Graph Compiler needs to choose the input submatrix sizes for the microkernel so that they are usually multiples of register sizes used by the vector and matrix function units. Also, it needs to choose the batch size for the microkernel so that the whole input and output submatrices fit within the L1 cache. To further streamline the cache access, the input and output tensors are blocked. To simplify the implementation, the input and output tensors are blocked using the submatrix sizes [MB, NB, KB]. So, each microkernel accesses a contiguous memory buffer.

The parameters for lowering a matmul op refer to the variable values in the template above: MPN, NPN, MB, NB, KB, BS, and ordering of loops indexed by ms, ks, and ns. The other parameters can be derived from the parameters above. oneDNN Graph Compiler uses an expert-tuned heuristic to decide these parameters. For a given output matrix size, it first proposes single-core kernel size options, a set of [MPN, NPN], which can use all cores with good load balance. It further proposes microkernel size options, a set of [MB, NB, KB, BS], which ensure good microkernel performance. Then the heuristic picks a pair of these sizes, which has the best overall kernel performance for the entire system with all cores. It iteratively searches for the best parameters, based on a cost model which considers multi-core load balancing and single-core kernel efficiency. Heuristic also reports the loop ordering of the inner loops which it assumes when computing the single-core kernel efficiency during the search process.

oneDNN Graph Compiler developed multiple templates for different uses. One Tunable OP can have multiple templates depending on the use cases. For example, for training use cases, the input activation can be significantly larger than the L2 cache size so requires an additional loop level to block the data for the L2 cache. For inference cases, sometimes the use case only processes one data sample with multiple cores so that the template may have to apply “k-slicing” to extract additional parallelism from the reduction axis.

Very often some dimensions of the input tensors are not multiple of microkernel sizes. oneDNN Graph Compiler pads the input tensors. The padding and un-padding happen at the entry and exit points of the computation graph and are fused into the Tunable OP. The examples used in this paper assume the tensor sizes are aligned with hardware vector and matrix operand sizes to simplify the description.

Template with Anchors for Fused OP Lowering

Fusion is a very important technique to achieve high performance in the deep learning domain. oneDNN Graph Compiler supports two types of fusion: fine-grain fusion fuses one Tunable OP with multiple adjacent Fusible OPs to a Fused OP, and coarse-grain fusion fuses multiple Fused OPs. The fine-grain fusion is supported by the template of Tunable OP. The template contains

placeholders, as known as anchors, at the beginning and the end of each loop level for the input and output tensors. The Graph IR fusion optimization decides whether it is profitable to fuse a Fusible OP to a Tunable OP and which anchor point is assigned to the Fusible OP. The Fused OP lowering pass retrieves anchors for Fusible OPs and directly inserted its corresponding Tensor IR at the anchor.

The main benefit of fusion is that the operation being fused only needs to access tensor slices associated with the anchor. The anchors before the microkernel are called pre-op anchors, which work on the input tensors' tensor slices. The anchors after the microkernel are called post-op anchors, which work on the output tensor's tensor slice. The Fusible op is called pre-op if it is inserted into pre-op anchors, and the Fusible op for post-op anchors is called post-op.

The fusion optimization evaluates total memory access and computation cost for each anchor point and selects one. There are multiple choices of commit anchors to insert a pre-op or post-op fusion, depending on the different levels of the loop nest. The commit anchors inside the innermost loop work on the smallest tensor slice, which provides a low per-access cost as the data is in the fastest cache. However, the inner loop bodies also have more computations since these computations are performed redundantly along the orthogonal dimensions.

Meanwhile, the fusion process may require allocating additional temporary tensors, which the fusion optimization also needs to evaluate carefully and factor in the potential interference with the cache behavior of Tunable OP. For example, the pre-op introduces temporary tensors since they can't work on the original input tensors directly, which causes data conflicts with the read accesses from other parallel cores.

Figure 3 illustrates the pre-op anchors and post-op anchors within a template and the associated tensor slices for each anchor. The right table in Figure 3 shows the tensor slice working set size for each anchor point which describes the memory size accessed by the fused operation at the anchor point on a single core. It also shows the formula to compute how many times the fused op is invoked within a single-core kernel and how many total tensor element memory accesses are needed for each anchor point. The concrete number can be deduced when the template is instantiated with the parameters for a Tunable OP.

The fusion optimization uses a heuristic to decide which anchor to choose. The heuristic evaluates the cost of a single-cost kernel between all possible anchors and the option of not fusion, and then it chooses the one with the lowest estimated cost. Many choices are straightforward so there is no need for over-tuning. For the template illustrated in Figure 3, anchor #4 is a good option for pre-

```
Parallel loop mpi = 0, MPN, 1 {
  pre_op_anchor#1 : A[mpi*MSN:MSN, 0:KSN, 0:MB, 0:KB];
  pre_op_anchor#1 : B[0:KSN, 0:NPSN, 0:NB, 0:KB];
  Parallel loop npi = 0, NPN, 1 {
    pre_op_anchor#2 : A[mpi*MSN:MSN, 0:KSN, 0:MB, 0:KB];
    pre_op_anchor#2 : B[0:KSN,npi*NSN:NSN, 0:NB, 0:KB];
    Loop msi = 0, MSN, 1 {
      mpsi = mpi * M/MPN + msi;
      pre_op_anchor#3 : A[mpsi:1, 0:KSN, 0:MB, 0:KB];
      pre_op_anchor#3 : B[0:KSN,npi*NSN:NSN, 0:NB, 0:KB];
      C'[0:NSN, 0:MB, 0:NB] = 0;
      Loop ksi = 0, KSN, BS {
        pre_op_anchor#4 : A[mpsi:1, ksi:BS, 0:MB, 0:KB];
        pre_op_anchor#4 : B[ksi:BS, npi*NSN:NSN, 0:NB, 0:KB];
        Loop nsi = 0, NSN, 1 {
          npsi = npi*N/NPN + nsi;
          pre_op_anchor#5 : A[mpsi:1, ksi:BS, 0:MB, 0:KB];
          pre_op_anchor#5 : B[ksi:BS, npsi:1, 0:NB, 0:KB];
          A_addr[0..BS] = &A[mpsi:1, ksi: 0..BS, 0:MB, 0:KB];
          B_addr[0..BS] = &B[ksi:0..BS, npsi:1, 0:NB, 0:KB];
          C'[nsi:1,0:MB, 0:NB] += Batch_reduce_gemm
                (A_addr[0..BS], B_addr[0..BS],Batch = BS);
        }
      }
      C[mpsi:1, npi:NSN, 0:MB, 0:NB] = C'[0:NSN, 0:MB, 0:NB];
      post_op_anchor#1 : C[mpsi:1, npi:NSN, 0:MB, 0:NB];
    }
    post_op_anchor#2 : C[mpi*MSN:MSN, npi*NSN:NSN, 0:MB, 0:NB];
  }
  post_op_anchor#3 : C[mpi*MSN:MSN, 0:NPSN, 0:MB, 0:NB];
}
```

Anchor	Tensor slice's working set size per core	Access times per core	Total memory access per core
pre_op_anchor#1	A' [MSN, KSN, MB, KB] B' [KSN,NPSN, NB, KB]	1	MSN* MB * KSN * KB NPSN * NB * KSN * KB
pre_op_anchor#2	A' [MSN, KSN, MB, KB] B' [KSN,NSN, NB, KB]	1	MSN* MB * KSN * KB NSN * NB * KSN * KB
pre_op_anchor#3	A' [KSN, MB, KB] B' [KSN, NSN, NB, KB]	MSN	MSN * MB * KSN * KB MSN * NSN * NB * KSN * KB
pre_op_anchor#4	A' [BS, MB, KB] B' [BS, NSN, NB, KB]	MSN * KSN/BS	MSN * MB * KSN * KB MSN * NSN * NB * KSN * KB
pre_op_anchor#5	A' [BS, MB, KB] B' [BS, KB, NB]	MSN * NSN * KSN/ BS	MSN * MB * KSN * KB *NSN MSN * NSN * NB * KSN * KB
post_op_anchor#1	C[MB, NSBN]	MSN	MSBN*NSBN
post_op_anchor#2	C[MSBN, NSBN]	1	MSBN*NSBN
post_op_anchor#3	C[MSBN, N]	1	MSBN * N

The template has predefined anchors as placeholders to fuse pre-ops and post-ops. Each anchor point is associated with a tensor slice for the pre-ops and post-ops to work on. Once the blocking parameters are decided, the tensor slice size and access times can be deduced to support the fusion decision. The fusion optimization pass chooses anchor points for groups of pre-ops and post-ops according to the estimated computation cost.

Figure 3. Fused OP template with anchors and cost table

op processing input tensor A. Compared to anchor #5, the associated tensor slice is the same, but there is an additional "nsi" loop which causes redundant and same computation for the tensor slices under each iteration. However, anchor #5 is a good option for B, since the "nsi" loop reduces the associated tensor slice size to B[BS, NB, KB], compared to B[BS, NSN, NB, KB] from anchor #4. Although the total memory access for B is the same between anchor #4 and #5, the memory access cost for anchor #4 is smaller since the tensor slice is more likely located in the cache closer to the CPU core.

The post-op fusion must be done after k-dimension reduction is done, or else the post-op computation interferes with the reduction and produces incorrect results. So, the first post-op anchor in Figure 3 is not in the innermost loop until the "ksi" loop is completed. Among the three post-op anchors, anchor #1 is the most profitable choice, as it has the smallest tensor slice so likely the data is still "hot" in the cache. For reduction post-op, it may split into two anchor points. It first inserts the first half of reduction in anchor #1 , which reduces to a partial result into a temporary tensor, and then inserts the second half at either anchor #2 or #3 which reduces to the final result. If the reduction is along "n" dimension, it usually chooses anchor #3 since at this point there is no need to perform synchronization across multiple cores for the final reduction as the value for the "n" dimension is all computed.

```
Parallel loop mp i= 0, MPN, 1 {
  Parallel loop npi = 0, NPN, 1 {
    Loop msi = 0, MSN, 1 {
      mpsi = mpi * M/MPN + ms;
      C'[0:NSN, 0:MB, 0:NB] = 0;
       Loop ksi = 0, KSN, BS {
          Reorder(A, [1, 1], A'[mpsi:1, ksi:BS, 0:MB, 0:KB],
                                          [MB, KB], from=[mpsi, ksi]);
          Loop nsi = 0, NSN, 1 {
            npsi = npi * N/NPN + nsi;
            A'_addr[0:BS] = &A'[mpsi, ksi:BS, 0, 0];
            B_addr[0:BS] = &B[ksi:BS, npsi, 0, 0];
            C'[nsi:1,0:MB, 0:NB] += Batch_reduce_gemm
                  (A'_addr[0:BS], B_addr[0:BS],Batch = BS);
          }
      }
      C''[mpsi:1, npsi:NSN, 0:MB, 0:NB] = C'[0:NSN, 0:MB, 0:NB];
      C'''[mpsi:1, npi:NSN, 0:MB, 0:NB]) =
                                  Relu(C''[mpsi:1, npi:NSN, 0:MB, 0:NB]);
      Reorder( C'''[mpsi:1, npi:NSN, 0:MB, 0:NB], [MB, NB],
                                          C, [MB2, NB2], to=[mpsi, npi]);
    }
  }
}
```

Figure 4. Pseudo code for Fused OP

Figure 4 shows a pseudo-code for fusing reorder and ReLU (rectified linear unit) ops to an instantiated GEMM OP. The first reorder op is inserted as pre-op fusion at anchor #4, which converts from a plain layout tensor A to a blocked layout A' with blocking factors MB and KB. The fused reorder op works on the tensor slice of A', denoted as A'[mpsi:1, ksi:BS, 0:MB, 0:KB], which starts from the position A'[mpsi, ksi, 0, 0] and has a slice with the size of [BS, MB, KB]. By fusing the reorder at anchor #4, the single-core kernel handles the small amount of data in the cache and then immediately feeds the data to the micro-kernel as input.

The example code in Figure 4 fuses two post-ops, a ReLU op followed by a reorder op. Both operations are inserted at the post-op anchor #1. Instead of saving to output tensors, it saves to a temporary tensor, C'', and applies the ReLU computation. Similarly, the ReLU op outputs a temporary tensor C''', which is reordered to the result tensor C. The reorder op changes the memory layout of the C tensor from the blocking factor of MB and NB to MB2 and NB2. These temporary tensors introduced by the post-op fusion are non-essential and will be reduced to minimum size at the Tensor IR optimization stage.

Graph IR Optimization

The Tunable OP template provides the foundation for automating the process of building high-performance compute-intensive primitives and fusing neighbor memory-bound operations. It serves as the core functionality of oneDNN Graph Compiler. On top of that, oneDNN Graph Compiler also exploits optimization opportunities only available when the computation is offloaded as a computation graph instead of individual operations. The Graph IR is first decomposed into a graph of basic DNN operations, applied a number of optimizations, fused to a number of Fused OPs, and then lowered to Tensor IR using the templates for Tunable OPs and Fused OPs. This section introduces a few graph-level optimizations specific to the deep learning domain: low-precision conversion, constant weight preprocess, layout propagation, and fusion. Figure. 5 illustrates these important optimizations with a quantized multilayer perceptron (MLP) example.

The low-precision computation brings significant speedup as it reduces both the computation and memory bandwidth required to compute a deep learning model. The low-level precision computation graph preserves the compute-intensive operations in the FP32 data type with surrounding type conversion operations inserted by quantization tools. Low-precision conversion transforms the input DNN computation graph and converts the compute-intensive operation to low-precision.

Figure 5 starts with the input quantized DNN graph, which contains an FP32 matmul op surrounded by two dequantize ops and a quantize op, denoted by DQ and Q. The dequantize converts an Int8 data type tensor to FP32 and the quantize op does the reverse. The green box indicates a constant tensor and the orange box regular tensor. The fine line indicates scalar, the middle line indicates Int8 tensor, and the thick line indicates FP32 tensors. The example shows an asymmetric dynamic quantization case, so the first dequantize op scales A input tensor by a_s and then offset by a_z to adjust the zero point, and the other dequantize op just scales the B matrix with b_s. B matrix is the weight matrix. Low-precision conversion optimization first breakdown the quantize and dequantize op to be simple addition and multiply ops and transform the graph to be a mathematically equivalent form that uses Int8 matmul op. The transformation can be illustrated by the following mathematic equation. Matrix multiplication is denoted by "X", and elementwise multiply and addition are denoted by "*" and "+", the broadcast and type conversion needed for Uint8 or Int8 data type processing are omitted. The transformed equation looks more complex, but it lowers the matmul precision from FP32 to Int8, which is the main goal of the optimization.

$C[m,n] = Quantize\ (Dequantize\ (A[m, k], a\_s, a\_z)\ X_{fp32}\ Dequantize\ (B[k, n], b\_s), c\_s, c\_z)$

=>

$C[m,n] = (A[m, k]\ X_{int8}\ B[k, n]\ *_{fp32}\ (a\_s\ *_{fp32}\ b\_s)\ +_{fp32}\ (a\_z[m,k]\ X_{fp32}\ B[k, n]\ *\ b\_s)\ )\ *_{fp32} c\_s\ +_{int64}\ c\_z$

The const weight preprocessing optimization is to exploit the optimization opportunity that some of the input tensors are constant at the execution time. For the static quantization inference use case, the weight tensors and quantization parameters are constant, so a portion of the post-transformation equation contains computation over constant weight, scale, and zero point can be avoided completely at runtime. The challenge is that the weight data buffer might not be available during the compilation, so the compiled code needs to preprocess the constant weight at the execution time when it first arrives. As a_s, b_s, c_s, and c_z are constants passed as dequantize op's attribute, these constants can be folded in the compile-time. The equation above can be further transformed as

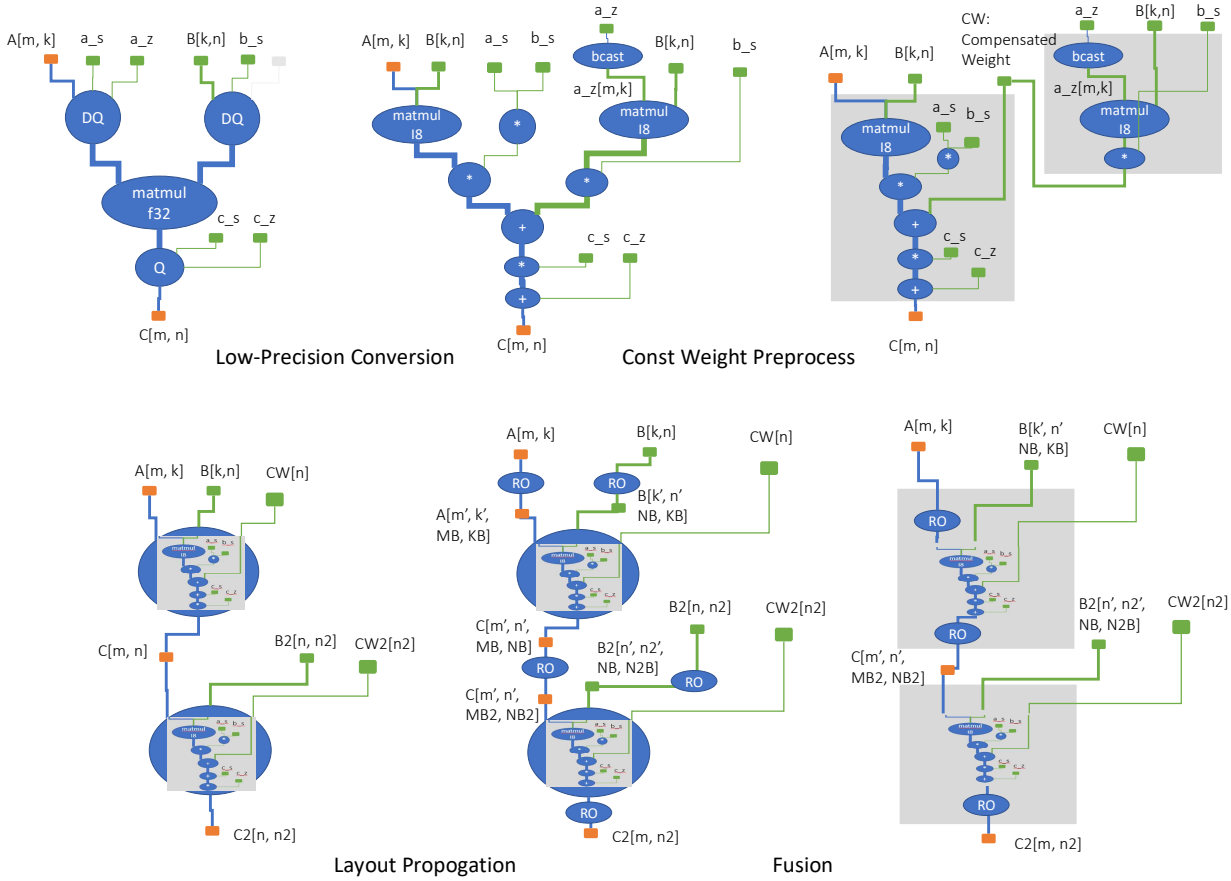


Figure 5. Graph IR optimization passes

below. Figure. 5 also shows the Graph IR after the constant weight preprocess optimization.

*if (init) const_weight_comp = a_z[m,k] X_{fp32} B[k, n] * $_{fp32}$ b_s;*

*C[m,n] = (A[m, k] X_{int8} B[k, n] * $_{fp32}$ a_s * $_{fp32}$ b_s + $_{fp32}$ const_weight_comp) * $_{fp32}$c_s + $_{int64}$ c_z;*

The constant weight preprocess optimization recognizes the constant tensor and builds a special initial function that preprocesses the constant weight and reuses the processed weight at the runtime. It recognizes the weight matrix B is passed as a constant logical tensor in the input graph, meaning that the weight buffer holds a constant value and won't change since the first execution of the compiled partition. This constant is named a runtime constant. If a DNN op's inputs are runtime constant or compile-time constant, the output tensor is runtime constant as well. The optimization propagates and marks all the runtime constants throughout the graph. Later the lowering generates special code for runtime constants, to make sure these runtime constants only be executed once in the first execution, and all future execution will reuse the processed result.

The layout propagation optimization exploits extra performance benefits with a sequence of Tunable OP by allowing Tunable OP to use the most desired block layout. As Tunable OP relies on the blocked layout to achieve the best performance on the CPU, very often the best-performed block layout might be different between two Tunable ops. It allows the Tunable ops within a subgraph to use a blocked layout but keep the graph input/output tensor as a plain layout. It first inserts reorder operations at the graph boundary to ensure the entry and exit points using the plain layout. Then it iterates the DNN computation graph and inserts reorder operation between two Tunable OPs if they use different blocked layouts. Before inserting a reorder OP before the Tunable OP, it first queries the Tunable OP for its desired blocked layouts, if none of the desired blocked layouts is consistent with the current layout, it inserts a reorder layout. The inserted reorder OPs are fused to the previous Tunable OP. The inserted layout reordering for the input weights produces runtime constants in the inference scenario, and they are handled by constant weight preprocessing like the compensated weight.

The Fusion optimization supports both fine-grain fusion and coarse-grain fusion. The fine-grain fusion inserts Fusible ops to anchor points of Tunable op and forms one fused op, and coarse-grain fusion tries to merge multiple fused ops together to generate optimized code for an even larger group of operations together.

Both pre-op fusion and post-op fusion help improve the memory locality since the Fusible OPs work on tensor slices instead of tensors after being fused into anchor points of Tunable OP, and tensor slices have a much higher possibility in the memory system closer to the compute function unit. For a Fusible op between two Tunable ops, it is more desirable to fuse it to the previous Tunable operation as post-op since the overall cost would be lower. The pre-op fusion only supports limited cases like reorder and transpose operations and only be used at the entry point of the graph. There are typically multiple Fusible operations that need to be fused as post-op. For example, matmul ops are usually followed by bias, activation, or normalization ops. These OPs are first broken down to a sequence of Fusible op, like elementwise, broadcast, reorder, and reduction ops, and then fused into Tunable OP as groups of post-ops.

The fine-grain fusion optimization grows a sequence of post-ops using a simple heuristic to decide whether the fusion is profitable. It first considers the immediate succeeding operations of the Tunable op as post-op candidates and keeps growing the sequence. The heuristic simply sets a limit of operations so stop growing the sequence when the limit is reached. For example, the post-op sequence can only have one reorder and one reduction op. As the post-op may involve accessing additional memory, like the second operand tensor for a binary op, the heuristic fusion optimization also monitors the total additional memory being accessed and limit the potential negative impact on the Tunable op execution. Then, the fine-grain fusion optimization sorts the post-ops in topological order and splits them into two groups if the post-ops contain one reduction op: the post-ops not dependent on the reduction op, and the reduction op and its dependent ops. The first group is inserted into the same post-op anchor. The reduction op may be split into two anchor points, and its dependent ops are inserted at the second anchor point where the final reduction result is collected.

The coarse-grain fusion optimization further merges fused ops. The larger scope of the DNN graph opens many possibilities. Multiple Fused ops could be lowered to one parallel loop, in order to improve data locality or better exploit the parallelism. For example, the outermost "mpi" loop of two fused ops may have the same blocking factor, so that they can be merged as one loop. When the heuristic chooses the parameters for each Tunable op, it

tries to choose the outermost loop blocking factor best aligned with core numbers, so each instantiated fused op has the same blocking factors as its neighbor. When the coarse-grain fusion optimization decides to merge two fused ops, it marks the two nested loops in Tensor IR as "mergeable" during the lowering process. Then Tensor IR merges two nested loops mechanically as guided by the Graph IR optimizations.

Tensor IR optimization

Tensor IR is the lowest intermediate representation in the oneDNN Graph Compiler. At the Tensor IR level, the DNN computation graph is lowered to a C-like program, which includes function, statement, expression, and intrinsic functions. The Fused OP is lowered as a function, which contains nested loops. A complex statement describes a structure like a loop, and a simple statement does computation. Var and Tensor represent scalar variables and multi-dimension arrays respectively.

Tensor IR supports Graph IR optimization by merging loops as instructed by Graph IR. Figure 6 shows the example of Tensor IR for the pseudo-code in figure 4. In the Tensor IR, the computation on the tensor slices is represented by either a nested loop or a function call to the microkernel. The inserted pre-op and post-op are lowered to nested loops. The two post-op, ReLU and reorder ops, are merged as one nested loop using the hint passed by Graph IR.

The main optimizations on Tensor IR are tensor size optimization and memory buffer optimization. Tensor size optimization tries to reduce the tensor size of each temporary tensor. The temporary tensors are introduced in the pre-op and post-op fusion process. The temporary tensor was initially introduced as a full-size tensor in the lowering process and then reduced by the tensor size optimization. In the example code of figure 6, the post-ops are fused into one loop nest in the Tensor IR. Since the accesses of the temporary tensors, C'' and C''', are local to the innermost loop body, the temporary tensor could be replaced by a scalar variable. Compared to accessing global and larger original tensors, the result code has a much smaller memory footprint and better cache locality. The temporary tensor introduced by pre-op fusion can be reduced similarly by analyzing the scope of the tensor usage. For example, A'[MSN, BS, MB, KB] could be reduced to A' [BS, NB, KB], since the producer of A' and consume are within the "msi" loop, so there is no need to save the result along the 2nd dimension of A'.

After tensor size optimization, the multiple-dimension tensor representation is flattened to a one-dimensional array to represent the memory buffer. The memory buffer

```
Var Const MPN, NPN, MSN, NSN, BS, KSN, MB, NB;
Tensor FP32[M, K] A;
Tensor FP32[M/MB, K/KB, MB, KB] A';
Tensor FP32[K/KB, N/NB, NB, KB] B;
Tensor FP32[NSN, MB, NB] C';
Tensor FP32[M/MB,N/NB, MB, NB] C'', C''';
Tensor FP32[M/MB2, N/NB2, MB2, NB2] C;
Var int* A_addr[BS], B_addr[BS];
Var Index mp, np, ms, ks, ns, nps, mps;
Parallel loop mp i= 0, MPN, 1 {
  Parallel loop npi = 0, NPN, 1 {
    Loop msi = 0, MSN, 1 {
      mpsi = mpi * M/MPN + msi;
      Loop nsi = 0, NSN, 1 {
        Loop mbi = 0, MB, 1 {
          Loop nbi = 0, NB, 1 {
            C'[0:NSN, 0:MB, 0:NB] = 0;
          }
        }
      }
      Loop ksi = 0, KSN, BS {
        Loop bsi = 0, BS, 1 {
          ksbi = ksi * BS + bsi;
          Loop mbi = 0, NB, 1 {
            Loop kbi = 0, KB, 1 {
              A'[mpsi, ksbi, mbi, kbi]
                  = A[mpsi*MB + mbi, ksbi*KB+kbi];
            }
          }
        }
        Loop nsi = 0, NSN, 1 {
          npsi = npi * N/NPN + nsi;
          Loop bsi = 0, BS, 1 {
            A'_addr[bsi] = &A'[mpsi, ksi, 0, 0];
            B_addr[bsi] = &B [ksi, npsi, 0, 0];
          }
          C'_addr = &C'[nsi,0, 0] ;
          Batch_reduce_gemm(A'_addr, B_addr,
              C'_addr, MB, NB, KB, Batch = BS);
        }
      }
      Loop nsi = 0, NSN, 1 {
        npsi = npi * N/NPN + nsi;
        Loop mbi = 0, MB, 1 {
          Loop nbi = 0, NB, 1 {
            C''[mpsi, npsi, mbi, nbi] = C'[nsi, mbi, nbi];
            C'''[mpsi, npsi, mbi, nbi]= max(C''[mpsi, npsi, mbi, nbi], 0);
            C[(mpsi*MB+mbi)/MB2, (npsi*NB+nbi)/NB2,
              (mpsi*MB+mbi)%MB2, (npsi* NB+nbi)%NB2]
                = C'''[mpsi, npsi, mbi, nbi];
          }
        }
      }
    }
  }
}
```

Figure 6. Lowering to Tensor IR

optimization tries to reuse the memory buffer of temporary tensors to have a minimum overall temporary buffer size for the compiled code and tries to improve the locality of the temporary buffer use.

Table 1. Workload parameters

Workload Category	data type	input batch size	sequence length	hidden size	head numbers
MLP_1	Int8, FP32	32, 64, 128, 256, 512	N/A	13x512x256x128	N/A
MLP_2	Int8, FP32	32, 64, 128, 256, 512	N/A	479x1024x1024x512x256x1	N/A
MHA_1	Int8, FP32	32, 64, 128	128	768	8
MHA_2	Int8, FP32	32, 64, 128	128	768	12
MHA_3	Int8, FP32	32, 64, 128	384	1024	8
MHA_4	Int8, FP32	32, 64, 128	512	1024	16

The main target of memory buffer optimization is to reuse the memory buffer created for the temporary tensors between fused op. In the inference use case, the output tensor is only consumed by the next fused op, and so the buffer could be reclaimed once the next fused op completed execution. Since the input tensor size is known to the compilation process, the memory buffer usage can be tracked at the compile time and optimized to improve efficiency.

Memory buffer optimization uses life span analysis like traditional compiler analysis for register allocation based on the def-use chain. Memory buffer optimization has extra consideration for buffer reuse since the memory buffer size used is not uniform as registers. It scans the Tensor IR, tracks all the memory buffers alive, and then computes the peak size of the total memory buffers needed for the entire compiled graph. The algorithm considers both reusing the hot memory and reducing the overall peak memory. At each point, when an intermediate buffer is needed, it tries to reuse the free intermediate buffers, which are already allocated but not used anymore. Among multiple choices of reusable memory buffers, it chooses the one that was used most recently, so likely the data is still in the cache system.

Experimental results

oneDNN Graph Compiler targets performance-critical DNN computation graph, which is usually a subgraph of the whole DNN model graph. We selected two DNN computation subgraphs as target workloads to evaluate the performance. The Multilayer Perceptron (MLP) workload contains multiple matmul ops intermixed with activation ops like ReLU. The MLP subgraph is the basic building block for many deep learning models, including recommendation systems and natural language processing. The Multi-Head Attention (MHA) subgraph is the key element to Transformer based deep learning models like Bert for natural language processing. The MHA workload focuses on the scaled dot-product attention portion of the MHA graph, which contains two batch matmul ops and a softmax as well as other binary ops between them. Depending on the use case, the MLP and MHA subgraphs tend to account for more than half of total model execution time, especially for DLRM or Bert Large models.

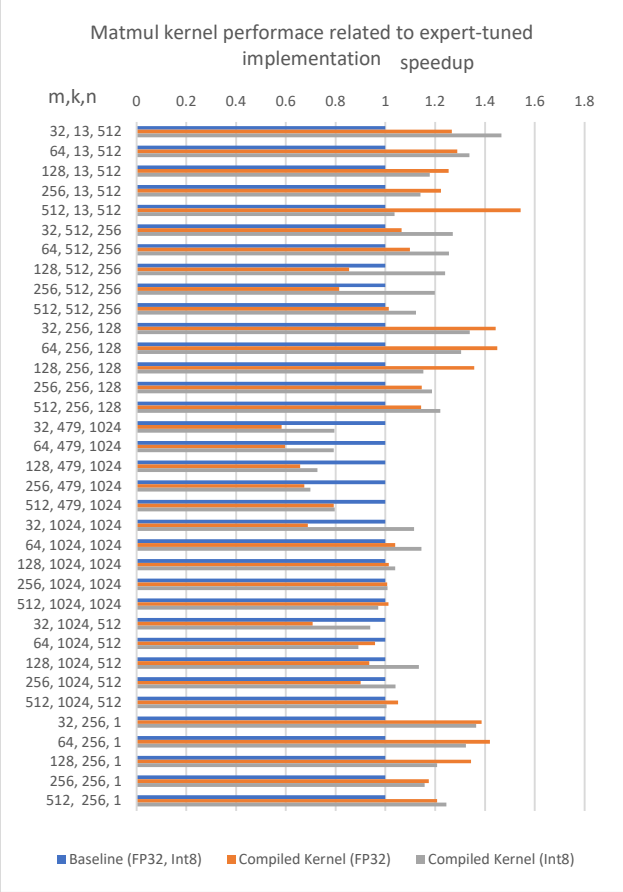


Figure 7. Performance comparison for individual Matmul op

We choose the inference use case for the evaluation and measure the performance for both FP32 and Int8 data types. We choose several representative data shapes for weights and input tensors and select a wide range of batch sizes. The weight sizes for MLP are from the MLPerf

DLRM model, and the sequence length and hidden size choices for MHA are from the Bert models. The performance data is collected on an Intel® Xeon® Platinum 8358 processor with 32 cores.

We first study the fine-grain fusion performance for individual layers since this is the foundation of the oneDNN Graph Compiler. The tests evaluate all the problem sizes used in the MLP tests. The baseline is heavily optimized and uses oneDNN primitives, which is the industry standard best-performant expert-tuned implementation. Both the baseline and oneDNN Graph Compiler assume weight being pre-packed, compensated, and preprocessed. The input and output matrixes are in plain layouts. Figure 7. shows that the performance of oneDNN Graph Compiler's automated kernel generated using the template approach is 6% better than the expert-tuned primitives for the given test cases. oneDNN Graph Compiler outperforms the expert-tuned primitives in many smaller problem sizes and falls behind on certain cases, particularly with k=479. The performance of individual layers heavily depends on the algorithm and heuristic, and we expect the performance gap on specific cases would be narrowed as oneDNN Graph Compiler continues to develop the algorithm and heuristic, and vice versa.

Figure 8 shows performance comparisons for MLP and MHA tests between oneDNN Graph Compiler and oneDNN Primitives. The left side shows performance data for FP32 data type and the right side is Int8. Each test is named with workload category, batch size, and data type as shown in Table 1. For each test, we measure the performance of the baseline, oneDNN Graph Compiler, and the middle setting which disables the coarse-grain fusion and evaluates the rest optimizations including the fine-grain fusion for oneDNN Graph Compiler. The baseline uses expert-tuned oneDNN primitive with fusion support and has been integrated into multiple DL frameworks to accelerate deep learning on the CPU. Specifically, it uses oneDNN primitives post-op fusion to fuse matmul op with ReLU for the MLP tests and with division and addition ops for the MHA tests. For Int8 tests, before calling the oneDNN primitives, the baseline applies similar low-precision graph transformation and maps the graph to low-precision matmul and post-ops. Besides that, the baseline also performs weight pre-packing, compensated weight preprocessing, and caches the result to avoid re-computation at runtime.

The performance results show oneDNN Graph Compiler significantly improves the performance of the target DNN computation subgraph. For MLP tests, oneDNN Graph

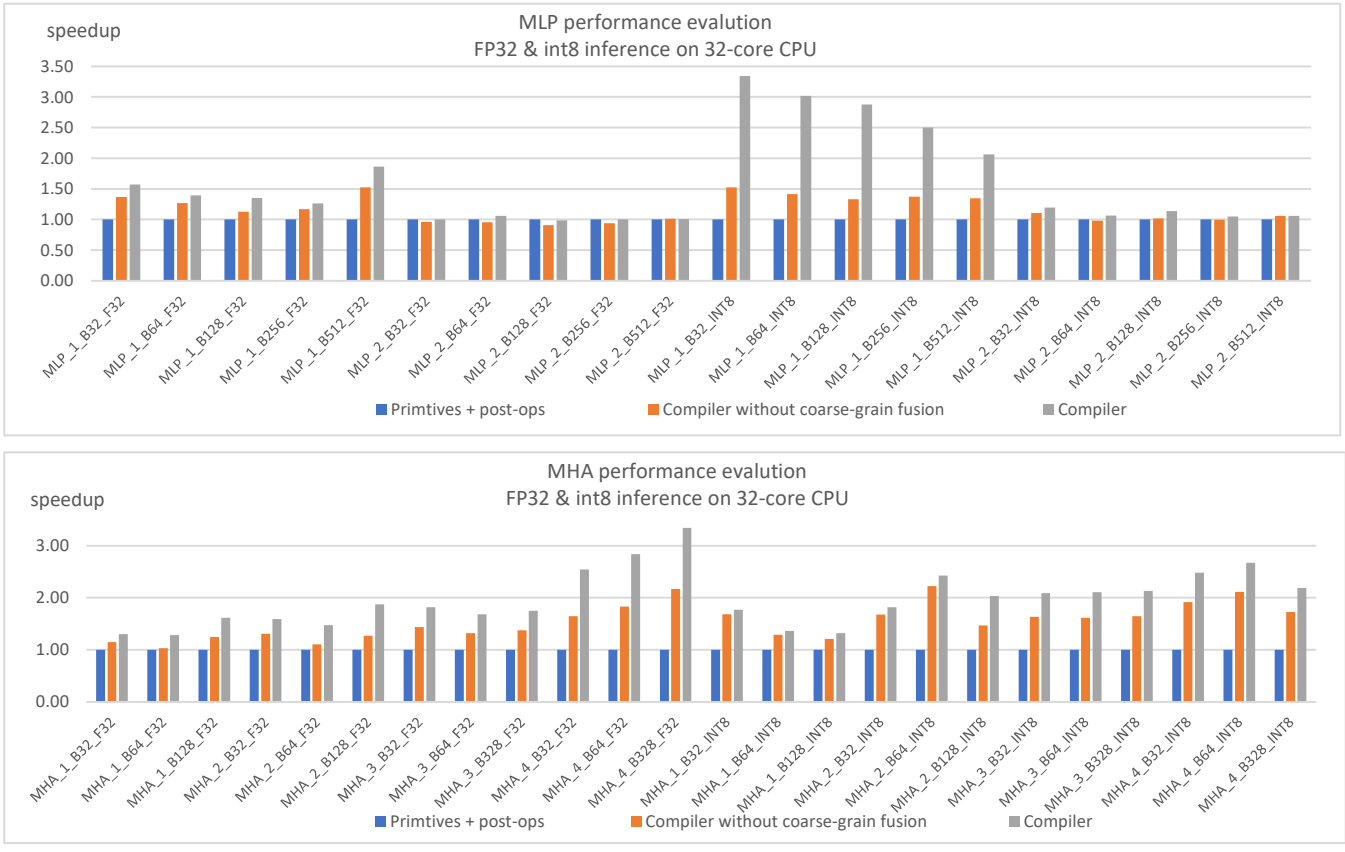


Figure 8. oneDNN Graph Compiler performance evaluation for MLP and MHA subgraph

Compiler demonstrates an average of 1.73x speed up on Int8 data type and 1.22x speed up on FP32. The five MLP_1 tests for Int8 data type show the highest speedup at an average of 2.72x. Among a total of 2.72x speed up, coarse-grain fusion is the main contributor and accounts for 1.95x. It merges 3 parallel loops, lowered from 3 matmul ops, into one parallel loop. The coarse-grain fusion greatly reduces the synchronization overhead and permits the activation data to be in the fastest cache for the next matmul op. For the MLP_1 Int8 tests, the entire activation and weight tensor fit in the L2 cache, so the coarse-grain fusion performs very well for these cases. When disabling the coarse-grain fusion, the remaining optimization accounts for about 1.4x. There are mainly three reasons. First, although the baseline implements the same fusion, oneDNN Graph Compiler has better performance for each individual matmul op in MLP_1, as shown in Figure 7. Second, the layout propagation allows all three matmul ops to run with the same blocked layout without extra reordering. Last, due to the MLP_1 tests being relatively short, the total API call overhead takes up to 10% of the execution time for the baselines, which is reduced by about 3 times since the compiled code needs only to be called one time. Compared to MLP_1 Int8 tests, the MLP_1 FP32 tests show an overall 1.47x performance gain, with 1.15x from coarse-grain fusion and 1.28x from rest optimizations.

For MLP_2 test cases, the oneDNN Graph Compiler shows an overall 10% better performance Int8 data type and 1% on FP32. When the coarse-grain fusion is disabled, oneDNN Graph Compiler is 1% slower compared to the baseline with FP32 and Int8 test cases combined. As we learned from the individual matmul op analysis, the initial layer of MLP_2 (k=479) has lower performance for oneDNN Graph Compiler and negatively impacts the overall performance. MLP_2 tests also get benefit from the coarse-grain fusion but to a lesser extent. The coarse-grain fusion is not able to merge all the loop nests due to the current heuristic limitation. We believe that the performance for MLP_2 test cases can be further improved with more heuristics tuning.

For the MHA subgraph, oneDNN Graph Compiler demonstrates an overall 1.91x performance gain over 24 MHA tests with 1.99x on Int8 data type and 1.84x on FP32. In contrast to the MLP performance, the performance benefit is more significant for the tests with larger problem sizes, and fine-grain fusion helps more than coarse-grain fusion. The baseline doesn't have the same fusion capability as the fine-grain fusion, as oneDNN post-op baseline doesn't fuse the softmax op with the preceding batch matmul op. oneDNN Graph Compiler decomposes softmax op to multiple basic operations, and its fine-grain fusion optimization fuses them to the preceding batch matmul ops. The basic operations are divided into two groups of post-ops: a group of element-wise ops and a group led by a reduction op. These two groups are inserted into the batch matmul op. This gives a significant boost to the performance by an average of 1.51x. The coarse-grain fusion adds another 27% performance gain on top of fine-grain fusion by merging the two nested loops translated from two batch matmul ops.

Conclusion

We proposed a hybrid approach to address the unique challenges of deep learning compilation. It distilled key ingredients of expert-tuned primitives for compute-intensive DNN operations like matrix multiplication and uses compiler techniques on the DNN computation graph to fully exploit the performance opportunity at the graph level. The template uses an expert-developed microkernel, algorithm, and heuristic, to ensure compiler-generated code achieves comparable performance to expert-tuned primitives. The compiler uses two-level intermediate representations at the level of both DNN op graph and C program to support domain-specific optimizations needed for deep learning computation, including low-precision, constant weight, tensor memory layout, fine-grain fusion, coarse-grain fusion, and tensor memory buffer reuse. Performance evaluation shows up to 2x performance gain for performance critical DNN computation graph in CPU inference usage.

Reference

[1] oneDNN. https://github.com/oneapi-src/oneDNN

[2] cuDNN. https://developer.nvidia.com/cudnn

[3] Tensorflow. https://www.tensorflow.org/

[4] Pytorch. https://pytorch.org/

[5] XLA. https://www.tensorflow.org/xla.

[6] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. CoRR abs/2002.11054 (2020)

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated endto-end optimizing compiler for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[8] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press.

[9] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. CoRR, abs/1802.04730, 2018.

[10] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, Gennady Pekhimenko. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. OSDI 2022: 233-248

[11] Kazushige Goto, Robert A. van de Geijn. Anatomy of High-Performance Matrix Multiplication, ACM Transactions on Mathematical Software Volume 34, Issue 3, May 2008, Article No.: 12. pp 1–25

[12] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, Enrique S. Quintana-Orti. Analytical Modeling Is Enough for High-Performance BLIS, ACM Transactions on Mathematical Software Volume 43, Issue 2, June 2017, Article No.: 12. pp 1–18

[13] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of High-Performance Many-Threaded Matrix Multiplication. IPDPS , page 1049-1059. IEEE Computer Society, (2014)

[14] Navdeep Katel, Vivek Khandelwal, Uday Bondhugula. MLIR-based code generation for GPU tensor cores. CC 2022: 117-128

[15] Philippe Tillet, H. T. Kung, David Cox. Triton: an intermediate language and compiler for tiled neural network computations, MAPL 2019: Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, June 2019, Pages 10–19

[16] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Bharat Kaul, Gagandeep Goyal, Ramakrishna Upadrasta, PolyDL: Polyhedral Optimizations for Creation of High Performance DL primitives, ACM Transactions on Architecture and Code Optimization, Volume 18, Issue 1, March 2021, Article No.: 11, pp 1–27

[17] AITemplate: Faster, more flexible inference on GPUs using AITemplate, a revolutionary new inference engine. https://ai.facebook.com/blog/gpu-inference-engine-nvidia-amd-open-source/

[18] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. CoRR, abs/1906.00091, 2019.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. NAACL-HLT (1) 2019: 4171-4186

[20] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali

Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasilii Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, Mayank Daga. MIOpen: An Open Source Library For Deep Learning Primitives. arXiv:1910.00078v1 [cs.LG]

[21] Nicolas Vasilache, Oleksandr Zinenko, Aart J.C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, Albert Cohen. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. arXiv:2202.03293 [cs.PL]

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Advances in neural information processing systems, 2012, pp. 1097–1105

[23] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1492–1500, 2017.

[24]. Evangelos Georganas, Dhiraj D. Kalamkar, Sasikanth Avancha, Menachem Adelman, Cristina Anderson, Alexander Breuer, Narendra Chaudhary, Abhisek Kundu, Vasimuddin Md, Sanchit Misra, Ramanarayan Mohanty, Hans Pabst, Barukh Ziv, and Alexander Heinecke. Tensor processing primitives: A programming abstraction for efficiency and portability in deep learning workloads. CoRR, abs/2104.05755, 2021.

[25] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. OSDI'20: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. November 2020 Article No.: 49, Pages 863–879

[26] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Depei Qian: The Deep Learning Compiler: A Comprehensive Survey. CoRR abs/2002.03794 (2020)

[27] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy. Learning to optimize tensor programs. NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing SystemsDecember 2018 Pages 3393–3404

[28] T. Zerrell and J. Bruestle, "Stripe: Tensor compilation via the nested polyhedral model," CoRR, vol. abs/1903.06498, 2019. [Online]. Available: http://arxiv.org/abs/1903.06498