Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN)

May 2019

Nathan Greeneltch

Deep Learning Software Stack for Intel® Processors

Deep learning and AI ecosystem includes edge and datacenter applications.

- Open source frameworks (TensorFlow*, MXNet*, CNTK*, PaddlePaddle*)
- Intel deep learning products (Neon™ framework , BigDL, OpenVINO™ toolkit)
- In-house user applications

Intel MKL and Intel® MKL-DNN optimize deep learning applications for Intel processors :

- Through collaboration with framework maintainers to upstream changes (TensorFlow*, MXNet*, PaddlePaddle*, CNTK*)
- Through Intel optimized forks (Caffe*, Torch*, Theano*)
- By partnering to enable proprietary solutions

Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN) is an open source performance library for deep learning applications (available at https://github.com/intel/mkl-dnn)

- Fast open source implementations for wide range of DNN functions
- Early access to new and experimental functionality
- Open for community contributions

Intel® Math Kernel Library (Intel® MKL) is a proprietary performance library for wide range of math and science applications

- Distribution: Intel Registration Center, package repositories (apt, yum, conda, pip)

TensorFlow* with Intel MKL/Intel® MKL-DNN

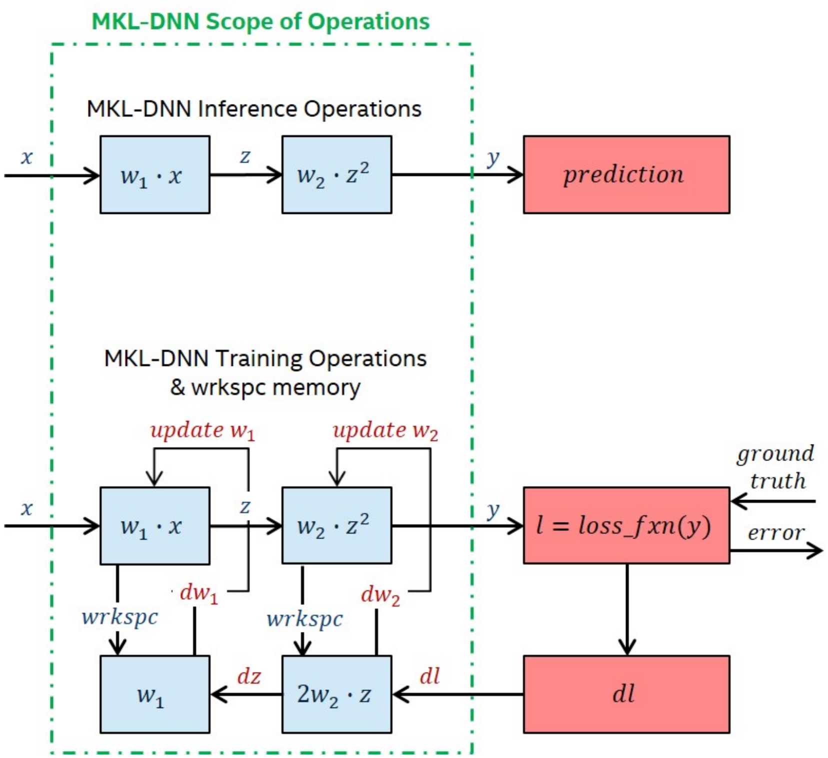
Use Intel Distribution for Python*

- Uses Intel MKL for many NumPy operations thus supports MKL_VERBOSE=1
- Available via Conda, or YUM and APT package managers

Use pre-built TensorFlow* wheels or build TensorFlow* with `bazel build --config=mkl`

- **Building from source required for integration with Intel Vtune™ Amplifier**
- Follow the CPU optimization advices including setting affinity and # of intra- and inter- ops threads
- More Intel® MKL-DNN-related optimizations are slated for the next version: Use the latest TensorFlow* master if possible

Intel® MKL-DNN scope



Primitives	Class
• (De-)Convolution • Inner Product • Vanilla RNN, LSTM, GRU	Compute intensive operations
• Pooling AVG/MAX • Batch Normalization • Local Response Normalization • Activations (ReLU, Tanh, Softmax, ...) • Sum	Memory bandwidth limited operations
• Reorder • Concatenation • Shuffle	Data movement

Intel® MKL-DNN overview

Features:

- Training (float32) and inference (float32, int8)
- CNNs (1D, 2D and 3D), RNNs (plain, LSTM, GRU)
- Optimized for Intel processors

Portability:

- Compilers: Intel C++ compiler/Clang/GCC/MSVC*
- OSes: Linux*, Windows*, Mac*
- Threading: OpenMP*, TBB

Frameworks that use Intel® MKL-DNN:

IntelCaffe, TensorFlow*, MxNet*, PaddlePaddle*

CNTK*, OpenVino, DeepBench*

Primitives	Class
• (De-)Convolution • Inner Product • Vanilla RNN, LSTM, GRU	Compute intensive operations
• Pooling AVG/MAX • Batch Normalization • Local Response Normalization • Activations (ReLU, Tanh, Softmax, ...) • Sum	Memory bandwidth limited operations
• Reorder • Concatenation • Shuffle	Data movement

Intel® MKL-DNN philosophy

Intel® MKL-DNN Overview

Descriptor: a structure describing memory and computation properties

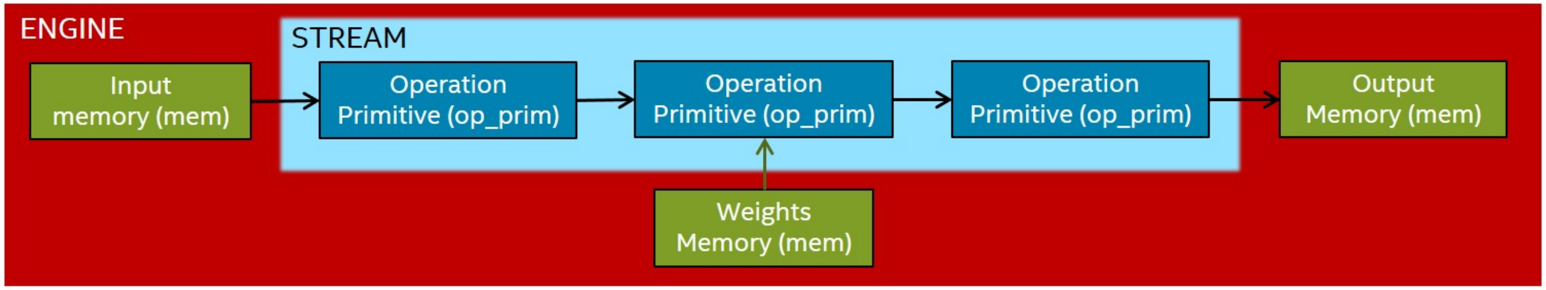
Primitive: a handle to a particular compute operation

- Examples: Convolution, ReLU, Batch Normalization, etc.
- Three key operations on primitives: **create**, **execute** and **destroy**
- Separate **create** and **destroy** steps help amortize setup costs (memory allocation, code generation, etc.) across multiple calls to **execute**

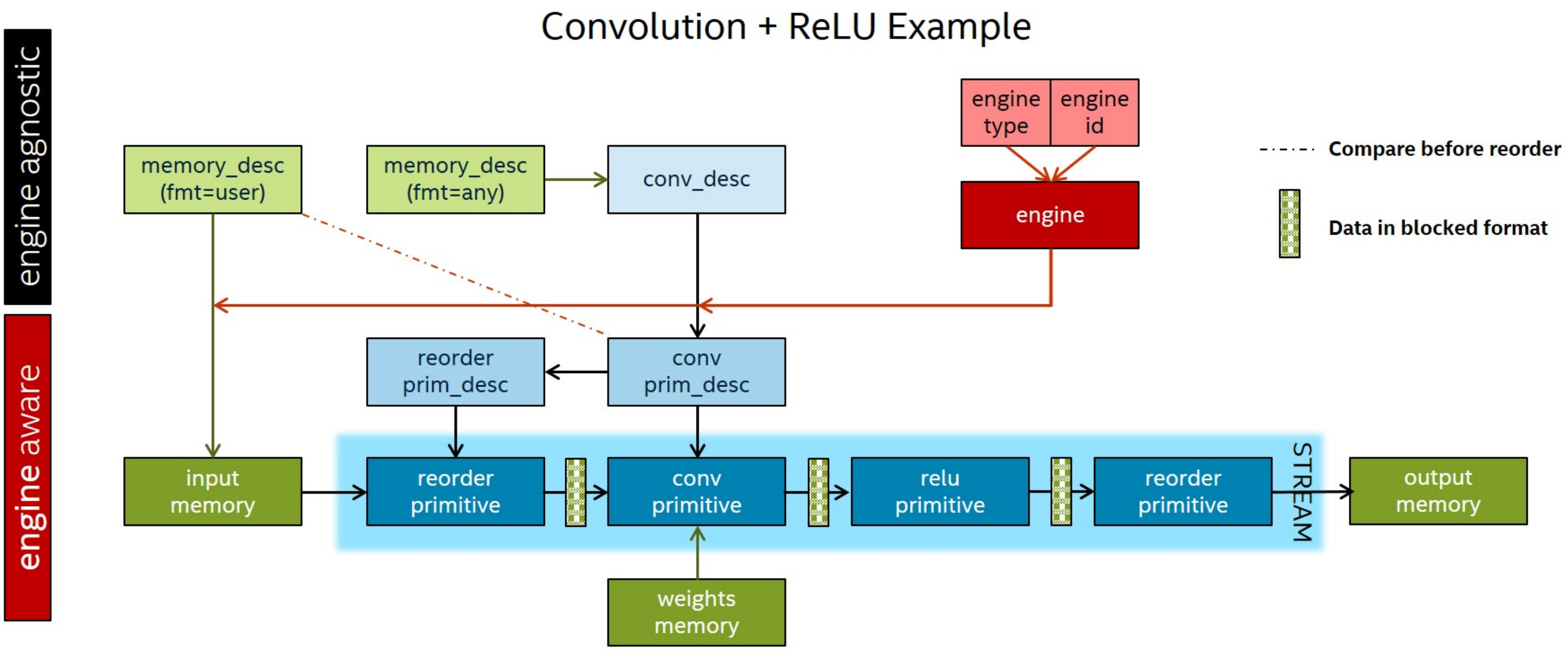
Memory: a handle to data

Stream: a handle to an execution context

Engine: a handle to an execution device

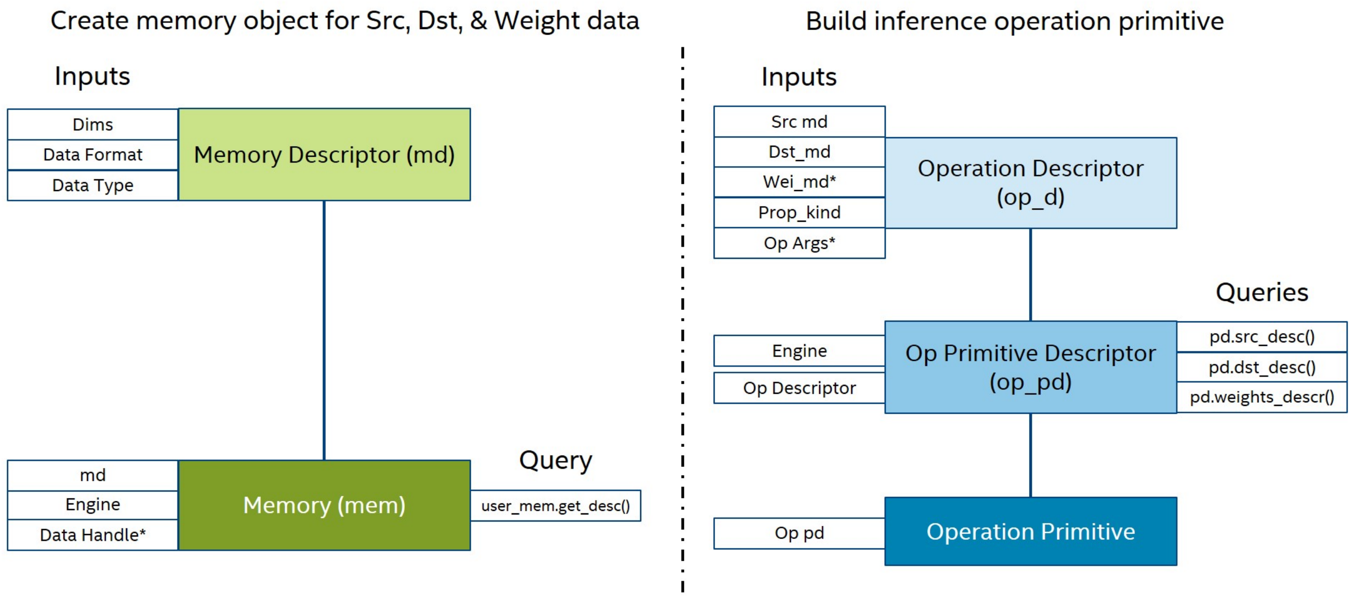


Intel® MKL-DNN Detailed Flow



Intel® MKL-DNN Object Snapshots

Create engine and associated stream



Query *mem* and *op_pd*, reorder if needed:
reorder(user_mem, op_mem).execute(stream, user_mem, op_mem)

Operation.execute (stream, {{MKLDNN_ARG_SRC, src_op_mem},{MKLDNN_ARG_DST, dst_op_mem}})

Key performance considerations on Intel processors

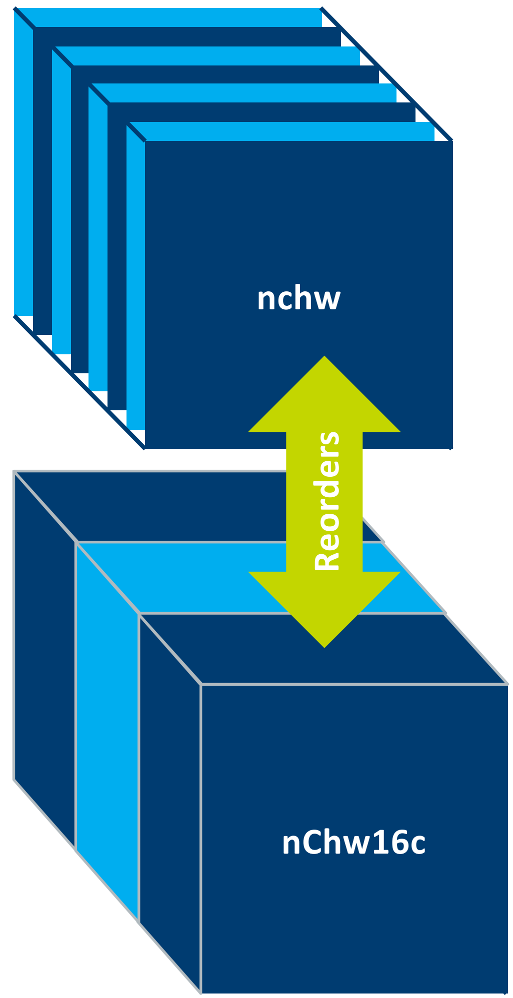
Memory layouts

Most popular memory layouts for image recognition are **nhwc** and **nchw**

- Challenging for Intel processors either for vectorization or for memory accesses (cache thrashing)

Intel® MKL-DNN convolutions use blocked layouts

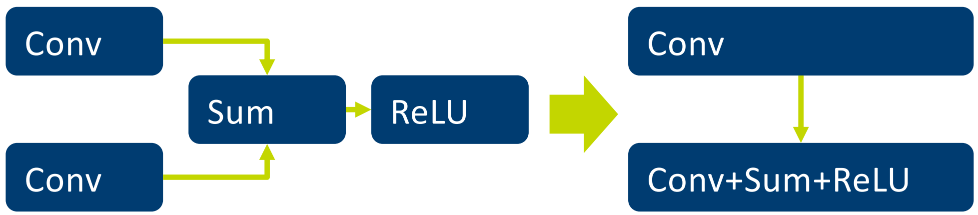
- Example: **nhwc** with channels blocked by 16 – **nChw16c**
- Convolutions define which layouts are to be used by other primitives
- Optimized frameworks track memory layouts and perform reorders **only** when necessary



Layout propagation: the steps to create a primitive

1. Create memory descriptors
   - These describe the shapes and memory layouts of the tensors the primitive will compute on
   - Use the **layout 'any'** as much as possible for every input/output/weights if supported (e.g. convolution or RNN). Otherwise, use the **same layout as the previous layer output**.
2. Create primitive descriptor and primitive
3. Create needed input reorders
   - Query the primitive for the input/output/weight layout it expects
   - Create the needed memory buffers and reorder primitives to accordingly reorder the data to the appropriate layout
4. Enqueue primitives and reorders in the stream queue for execution

Fusing computations



On Intel processors a high % of time is typically spent in BW-limited ops

- ~40% of ResNet-50, even higher for inference

The solution is to fuse BW-limited ops with convolutions or one with another to reduce the # of memory accesses

- Conv+ReLU+Sum, BatchNorm+ReLU, etc
- Done for inference, WIP for training

The FWKs are expected to be able to detect fusion opportunities

- IntelCaffe already supports this

Major impact on implementation

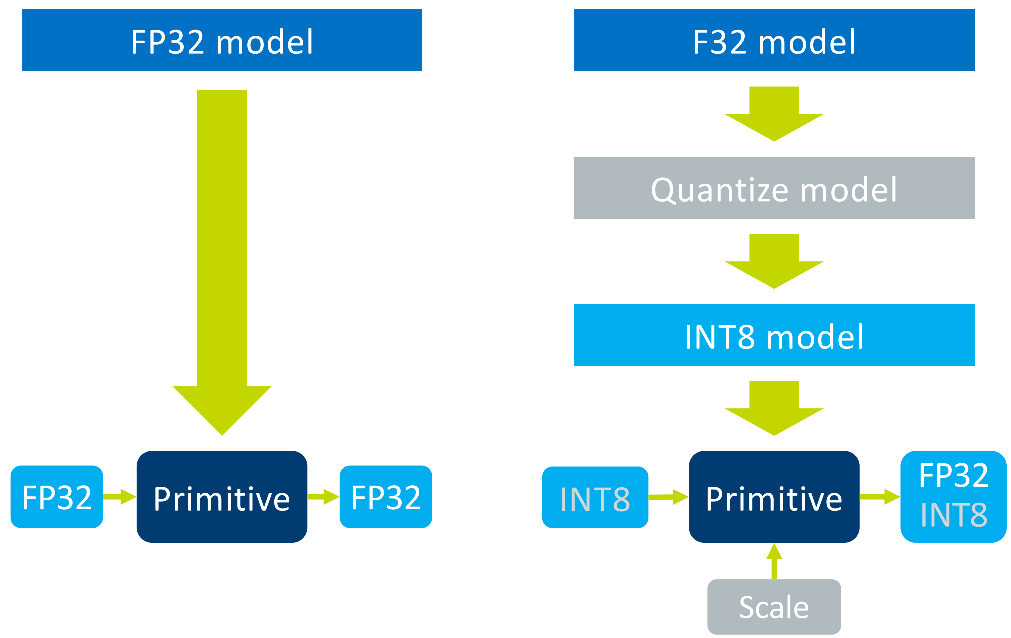
- All the impls. must be made aware of the fusion to get max performance
- Intel® MKL-DNN team is looking for scalable solutions to this problem

Low-precision inference

Proven only for certain CNNs by IntelCaffe at the moment

A trained float32 model quantized to int8

Some operations still run in float32 to preserve accuracy



Primitive attributes

Fusing layers through post-ops

1. Create a post_ops structure
2. Append the layers to the post-ops structure (currently supports sum and elementwise operations)
3. Pass the post-op structure to the primitive descriptor creation through attributes

Quantized models support through attributes (more details)

1. Set the scaling factors and rounding mode in an attribute structure
2. Pass the attribute structure to the primitive descriptor creation

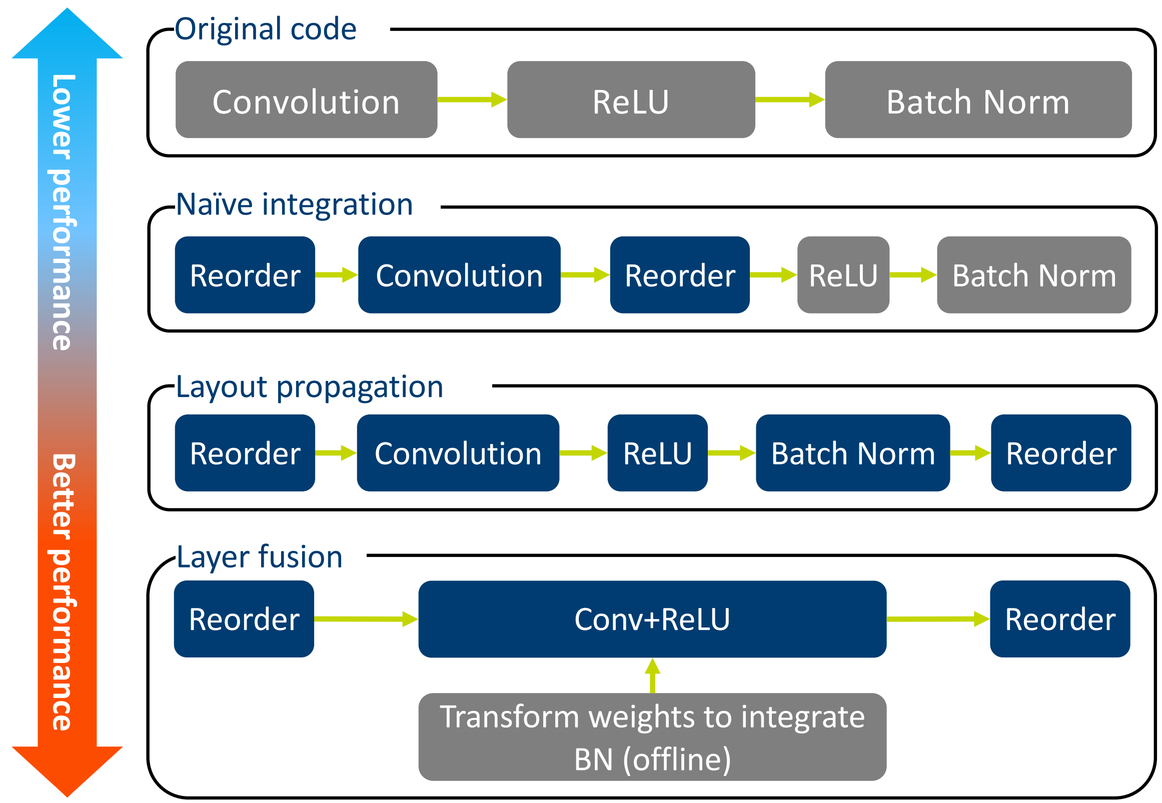
Intel® MKL-DNN integration levels

Intel® MKL-DNN is designed for best performance.

However, topology level performance will depend on Intel® MKL-DNN integration.

- Naïve integration will have reorder overheads.
- Better integration will propagate layouts to reduce reorders.
- Best integration will fuse memory bound layers with compute intensive ones or with each other.

Example: inference flow



Intel® MKL-DNN: How To Get?

Build from source using walkthrough → https://software.intel.com/en-us/articles/intel-mkl-dnn-part-1-library-overview-and-installation

Download and Build the Source Code

Clone the Intel MKL-DNN library from the GitHub repository by opening a terminal and typing the following command:

```
git clone https://github.com/01org/mkl-dnn.git
```

Validating the Build

To validate your build, execute the following command from the *mkl-dnn/build* directory:

```
make test
```

This step executes a series of unit tests to validate the build. All of these tests should indicate *Passed*, and the processing time as shown in Figure 3.

```
bbrown@bbrown-PC: ~/mkl-dnn/build
bbrown@bbrown-PC:~/mkl-dnn/build$ make test
Running tests...
Test project /home/bbrown/mkl-dnn/build
      Start  1: simple-net-c
 1/22 Test  #1: simple-net-c .........................   Passed    2.52 sec
      Start  2: simple-net-cpp
 2/22 Test  #2: simple-net-cpp .......................   Passed    0.65 sec
      Start  3: api-c
 3/22 Test  #3: api-c ................................   Passed    0.01 sec
      Start  4: test_c_symbols-c
 4/22 Test  #4: test_c_symbols-c .....................   Passed    0.01 sec
      Start  5: test_sum
 5/22 Test  #5: test_sum .............................   Passed    0.08 sec
      Start  6: test_reorder
```

Intel® MKL-DNN: How to know if you have it in framework? → MKLDNN_VERBOSE

```
export MKLDNN_VERBOSE=1

./program.exe
```

```
mkldnn_verbose,info,Intel(R) MKL-DNN v0.18.0 (Git Hash 4cfed5bf82f1339d7c8c7f622fda02dc00ec8ad8),
Intel(R) Advanced Vector Extensions 2 (Intel(R) AVX2)
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_nchw out:f32_nChw8c,num:1,2x16x7x7,0.529053
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_oihw out:f32_OIhw8i8o,num:1,16x16x5x5,0.98999
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_nchw out:f32_nChw8c,num:1,2x16x7x7,0.453125
mkldnn_verbose,exec,reorder,simple:any,undef,in:f32_x out:f32_x,num:1,16,0.388916
mkldnn_verbose,exec,convolution,jit:avx2,forward_training,fsrc:nChw8c fwei:OIhw8i8o fbia:x
fdst:nChw8c,alg:convolution_direct,mb2_ic16oc16_ih7oh7kh5sh1dh0ph2_iw7ow7kw5sw1dw0pw2,0.0241699
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_nChw8c out:f32_nchw,num:1,2x16x7x7,0.469971
```

Intel® MKL-DNN verbose mode overview

Simple yet powerful analysis tool:

- Similar to [Intel MKL verbose]
- Enabled via environment variable or function call
- Output is in CSV format

Output includes:

- The marker, state and primitive kind
- Implementation details (e.g. jit:avx2)
- Primitive parameters
- Creation or execution time (in ms)

Example below (details [here])

```
$ # MKLDNN_VERBOSE is unset
$ ./examples/simple-net-c
passed

$ export MKLDNN_VERBOSE=1 # report only execution parameters and runtime
$ ./examples/simple-net-c # | grep "mkldnn_verbose"
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_oihw out:f32_Ohwi8o,num:1,96x3x11x11,12.2249
mkldnn_verbose,exec,eltwise,jit:avx2,forward_training,fdata:nChw8c,alg:eltwise_relu,mb8ic96ih55iw55,0.437988
mkldnn_verbose,exec,lrn,jit:avx2,forward_training,fdata:nChw8c,alg:lrn_across_channels,mb8ic96ih55iw55,1.70093
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_nChw8c out:f32_nchw,num:1,8x96x27x27,0.924805
passed
```

Performance gaps causes

Functional gaps: your hotspot is a commonly/widely used primitive and is not enabled in Intel® MKL-DNN

Integration gaps: your hotspot uses Intel® MKL-DNN but runs much faster in a standalone benchmark (more details in the hands-on session)

Intel® MKL-DNN performance issue: your hotspot uses Intel® MKL-DNN but is very slow given its parameters

In any of these cases, feel free to contact the Intel® MKL-DNN team through the Github* page issues section.

TensorFlow* integration

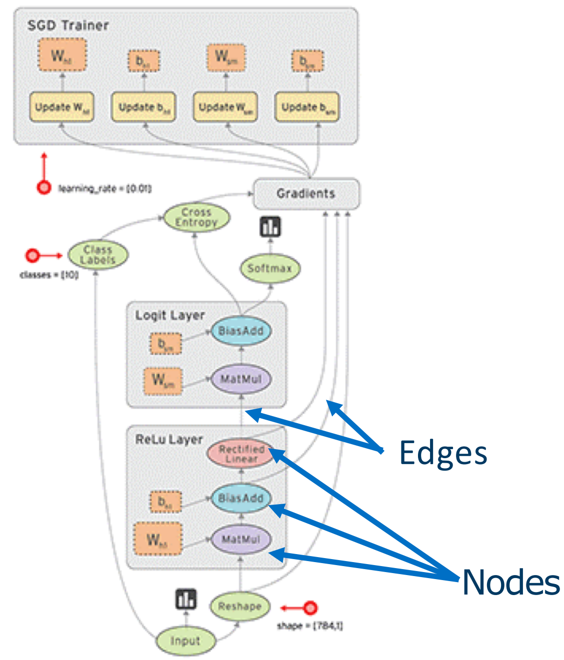
Prototyping a TensorFlow* model

TensorFlow* Core

1. Build a computational graph (a tf.Graph).

- Google's definition: "A computational graph is a series of TensorFlow* operations arranged into a graph. "
- Nodes(compute operations) & Edges (Tensors: ndarray)

2. Run the computational graph (using a tf.Session).



Data Flow graph Advantages

1. Parallelism
2. Distributed execution

Intel-optimized TensorFlow*

Intel® MKL-DNN

Primitives for DNN domain

Library is open-source (https://github.com/intel/mkl-dnn) and downloaded automatically when building TensorFlow*.

MKL-DNN accelerates AlexNet, VGG, GoogleNet, MXNet and ResNet neural networks

Optimizations introduced in TF:

Operator optimizations

Graph optimizations

System optimizations

Coming straight from MKL-DNN, out-of-the-box, no code changes required!!

Operator optimizations

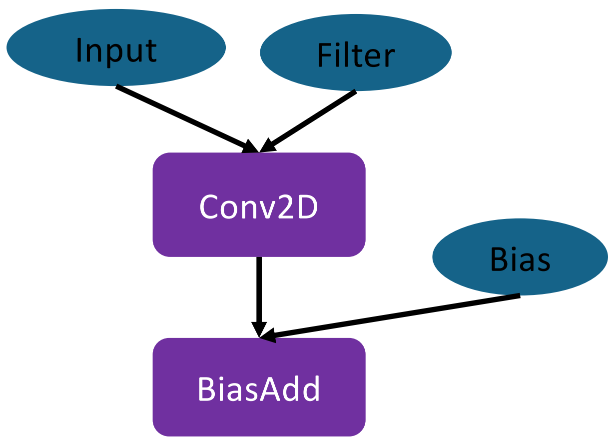
Replace default (Eigen) kernels by highly-optimized kernels (using Intel® MKL-DNN)

Key optimizations:

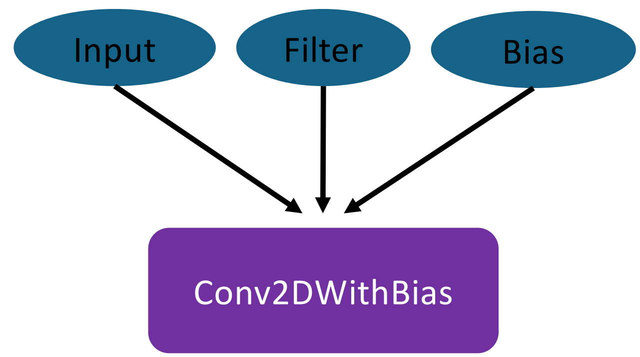
- Direct batched convolution
- Inner product
- Pooling: maximum, minimum, average
- Normalization: local response normalization across channels (LRN), batch normalization
- Activation: rectified linear unit (ReLU)
- Data manipulation: multi-dimensional transposition (conversion), split, concat, sum and scale.

Forward	Backward
Conv2D	Conv2DGrad
Relu, TanH, ELU	ReLUGrad, TanHGrad, ELUGrad
MaxPooling	MaxPoolingGrad
AvgPooling	AvgPoolingGrad
BatchNorm	BatchNormGrad
LRN	LRNGrad
MatMul, Concat	

Graph optimizations: fusion



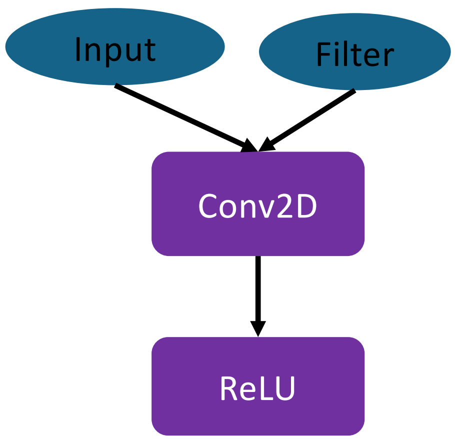
Before Merge



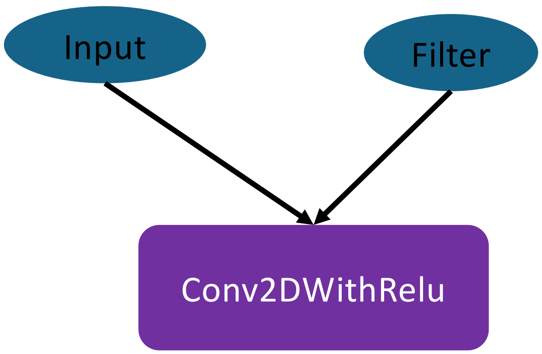
After Merge

Software

Graph optimizations: fusion



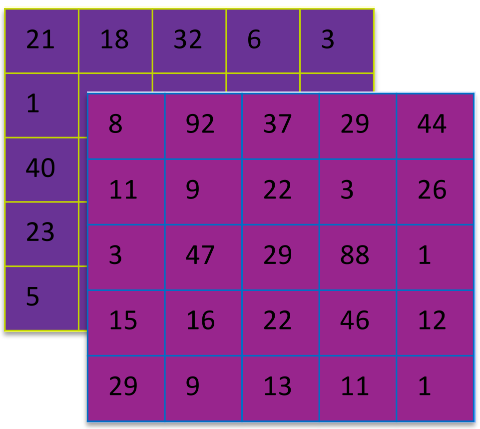
Before Merge



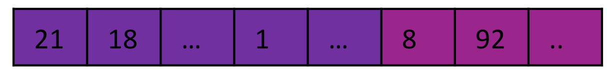
After Merge

Graph optimizations: layout propogation

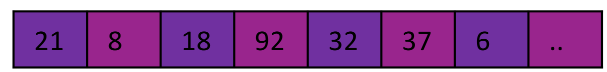
- What is layout?
  - How do we represent N-D tensor as a 1-D array.



{N:2, R:5, C:5}



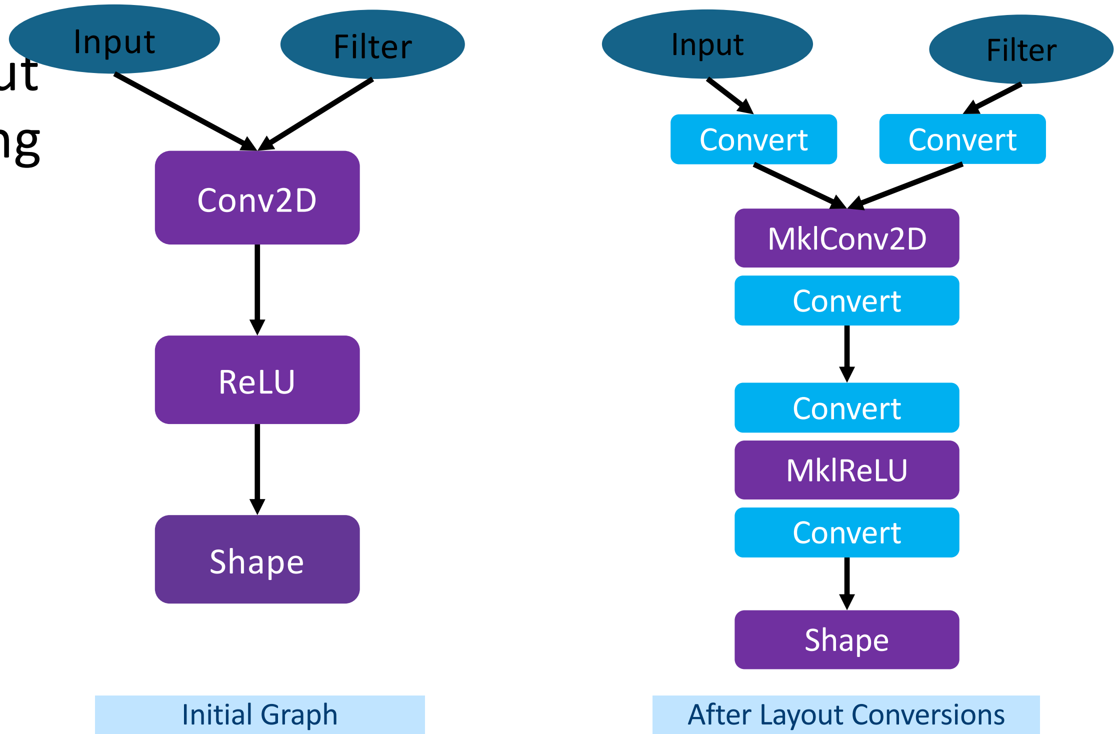
Better optimized for
some operations
vs.



Impacts performance during
weight updates

Graph optimizations: layout COnversion

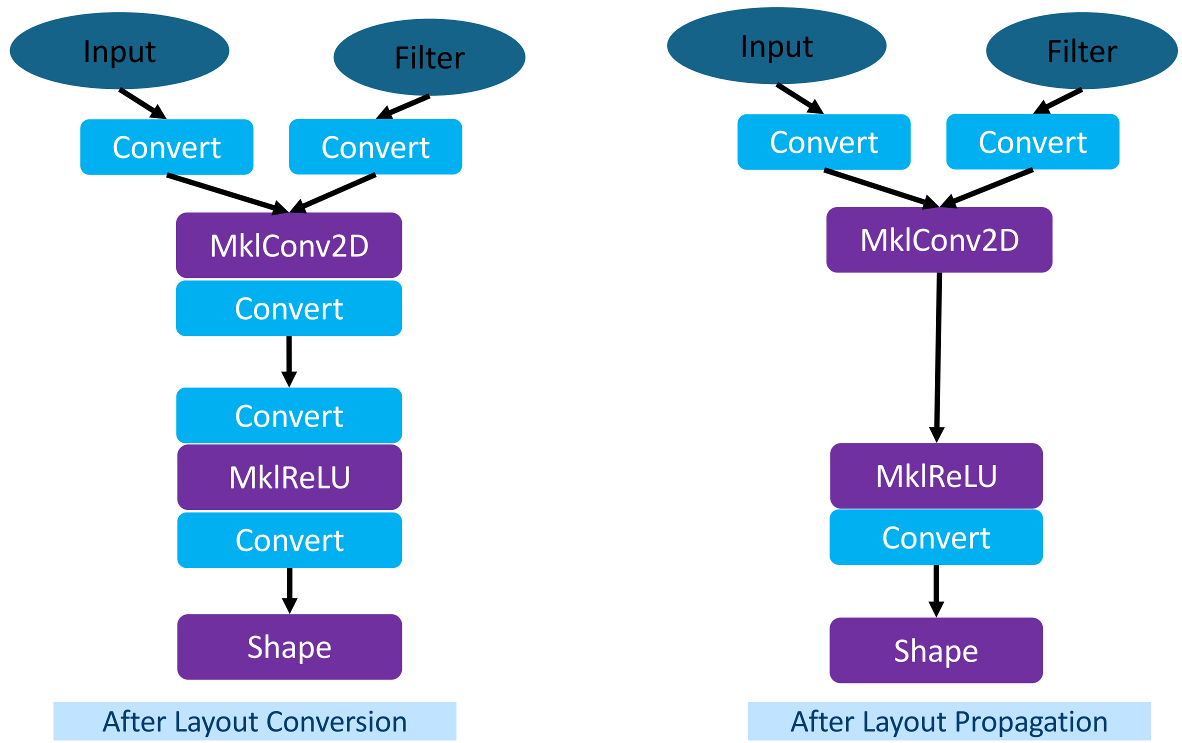
- Converting to/from optimized layout can be less expensive than operating on un-optimized layout.
- All MKL-DNN operators use highly-optimized layouts for TensorFlow* tensors.



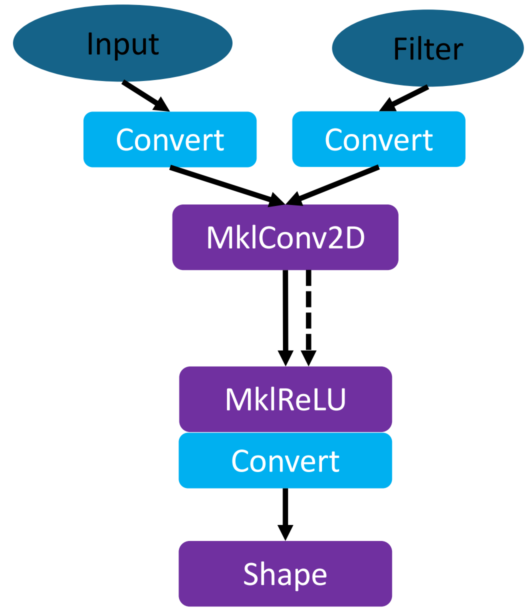
Graph optimizations: layout propagation

Did you notice anything wrong with previous graph?

Problem: redundant conversions



System optimizations: load balancing



TensorFlow* graphs offer opportunities for parallel execution.

Threading model, Tune you MKL w/

1. **`inter_op_parallelism_threads`** = max number of operators that can be executed in parallel
2. **`intra_op_parallelism_threads`** = max number of threads to use for executing an operator
3. **`OMP_NUM_THREADS`** = MKL-DNN equivalent of **`intra_op_parallelism_threads`**

More details: https://www.TensorFlow*.org/performance/performance_guide

```
>>> config = tf.ConfigProto()
>>> config.intra_op_parallelism_threads = 56
>>> config.inter_op_parallelism_threads = 2
>>> tf.Session(config=config)
```

```
os.environ["KMP_BLOCKTIME"] = "1"
os.environ["KMP_AFFINITY"] = "granularity=fine,compact,1,0"
os.environ["KMP_SETTINGS"] = "0"
os.environ["OMP_NUM_THREADS"] = "56"
```

System optimizations: load balancing

Incorrect setting of threading model parameters can lead to over- or under-subscription, leading to poor performance.

```
OMP: Error #34: System unable to
allocate necessary resources for
OMP thread:

OMP: System error #11: Resource
temporarily unavailable

OMP: Hint: Try decreasing the
value of OMP_NUM_THREADS.
```

Solution:

- Set these parameters for your model manually.
- Guidelines on TensorFlow* webpage

Key Takeaways

Key Takeaways

1. Application developers already benefit of Intel® MKL-DNN through integration in popular frameworks
2. Framework developers can get better performance on Intel processors by integrating Intel® MKL-DNN
3. There are different levels of integration, and depending on the level you will get different performance
4. Profiling can help you identify performance gaps due to
   - Integration not fully enabling Intel® MKL-DNN potential (more on that in the hands-on session).
   - Performance sensitive function not enabled with Intel® MKL-DNN (make requests on Github*)
   - Performance issue in Intel® MKL-DNN (raise the issue on Github*)

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

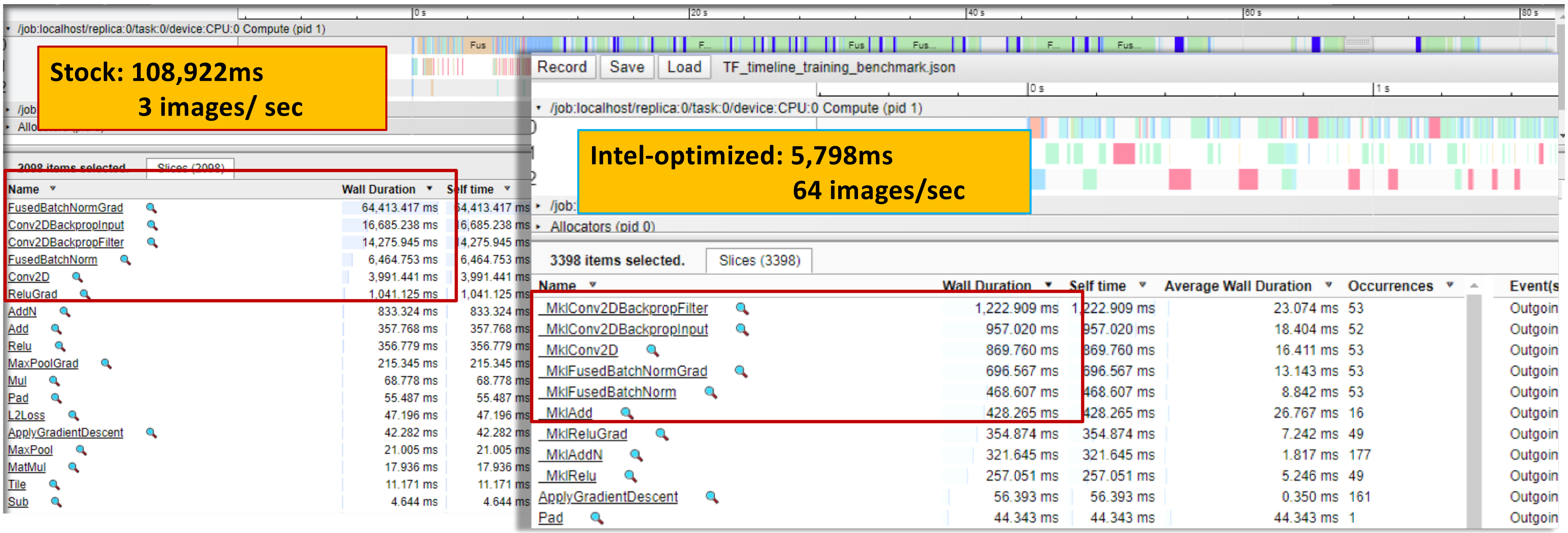
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

BACKUP

Profiling ResNet50 Training



Name	Wall Duration	Self time
FusedBatchNormGrad	64,413.417 ms	4,413.417 ms
Conv2DBackpropInput	16,685.238 ms	6,685.238 ms
Conv2DBackpropFilter	14,275.945 ms	4,275.945 ms
FusedBatchNorm	6,464.753 ms	6,464.753 ms
Conv2D	3,991.441 ms	3,991.441 ms
ReluGrad	1,041.125 ms	1,041.125 ms
AddN	833.324 ms	833.324 ms
Add	357.768 ms	357.768 ms
Relu	356.779 ms	356.779 ms
MaxPoolGrad	215.345 ms	215.345 ms
Mul	68.778 ms	68.778 ms
Pad	55.487 ms	55.487 ms
L2Loss	47.196 ms	47.196 ms
ApplyGradientDescent	42.282 ms	42.282 ms
MaxPool	21.005 ms	21.005 ms
MatMul	17.936 ms	17.936 ms
Tile	11.171 ms	11.171 ms
Sub	4.644 ms	4.644 ms

Name	Wall Duration	Self time	Average Wall Duration	Occurrences	Event(s
MklConv2DBackpropFilter	1,222.909 ms	1,222.909 ms	23.074 ms	53	Outgoin
MklConv2DBackpropInput	957.020 ms	957.020 ms	18.404 ms	52	Outgoin
MklConv2D	869.760 ms	869.760 ms	16.411 ms	53	Outgoin
MklFusedBatchNormGrad	696.567 ms	696.567 ms	13.143 ms	53	Outgoin
MklFusedBatchNorm	468.607 ms	468.607 ms	8.842 ms	53	Outgoin
MklAdd	428.265 ms	428.265 ms	26.767 ms	16	Outgoin
MklReluGrad	354.874 ms	354.874 ms	7.242 ms	49	Outgoin
MklAddN	321.645 ms	321.645 ms	1.817 ms	177	Outgoin
MklRelu	257.051 ms	257.051 ms	5.246 ms	49	Outgoin
ApplyGradientDescent	56.393 ms	56.393 ms	0.350 ms	161	Outgoin
Pad	44.343 ms	44.343 ms	44.343 ms	1	Outgoin

- Nearly **19X** faster & **20X** images processed/sec

```
$ python tf_cnn_benchmarks.py --device=cpu --mkl=True --data_format=NHWC \
   --kmp_affinity='granularity=fine,noverbose,compact,1,0' --kmp_blocktime=1 \
   --kmp_settings=1 --num_warmup_batches=20 --batch_size=256 --num_batches=50 \
   --model=resnet50 --num_intra_threads=56 --num_inter_threads=2 --forward_only=false \
   --trace_file='tf_timeline_training_benchmark_latest.json'
```

Benchmarking script:
https://github.com/TensorFlow*/benchmarks/tree/master/scripts
Open the json result with
chrome://tracing/ → load

Profiling

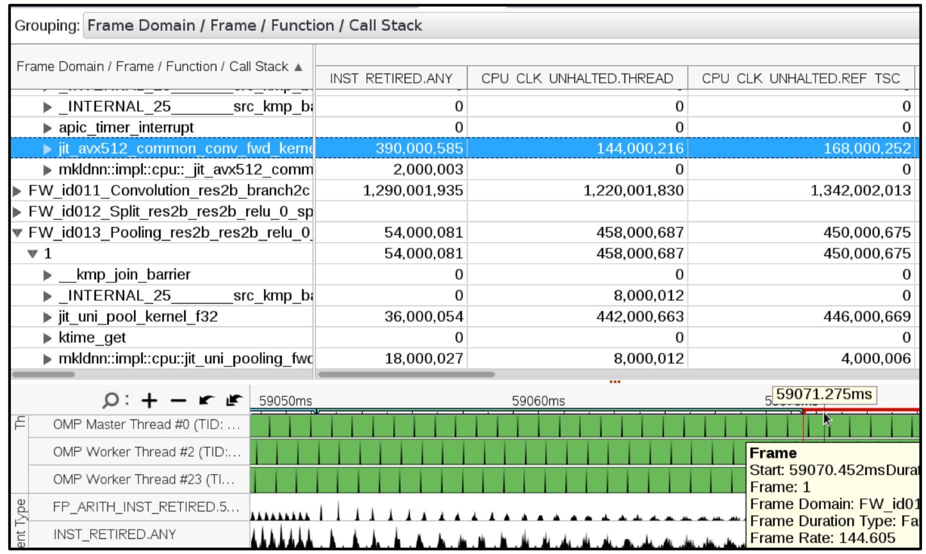
Integration with Intel VTune Amplifier

Full application analysis

Report types:

- CPU utilization
- Parallelization efficiency
- Memory traffic

Profiling of run-time generated code must be enabled at compile time




```
$ # building Intel® MKL-DNN using cmake
$ cmake -DVTUNEROOT=/opt/intel/vtune_amplifier_2018 .. && make install
$ # an alternative: building Intel® MKL-DNN using sources directly, e.g. in TensorFlow*
$ CFLAGS="-I$VTUNEROOT/include -DJIT_PROFILING_VTUNE" LDFLAGS="-L$VTUNEROOT/lib64 -ljitprofiling" bazel build
```

Intel® MKL-DNN verbose mode overview

Simple yet powerful analysis tool:

- Similar to Intel MKL verbose
- Enabled via environment variable or function call
- Output is in CSV format

Output includes:

- The marker, state and primitive kind
- Implementation details (e.g. jit:avx2)
- Primitive parameters
- Creation or execution time (in ms)

Example below (details here)

```
$ # MKLDNN_VERBOSE is unset
$ ./examples/simple-net-c
passed

$ export MKLDNN_VERBOSE=1 # report only execution parameters and runtime
$ ./examples/simple-net-c # | grep "mkldnn_verbose"
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_oihw out:f32_Ohwi8o,num:1,96x3x11x11,12.2249
mkldnn_verbose,exec,eltwise,jit:avx2,forward_training,fdata:nChw8c,alg:eltwise_relu,mb8ic96ih55iw55,0.437988
mkldnn_verbose,exec,lrn,jit:avx2,forward_training,fdata:nChw8c,alg:lrn_across_channels,mb8ic96ih55iw55,1.70093
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_nChw8c out:f32_nchw,num:1,8x96x27x27,0.924805
passed
```

Performance gaps causes

Functional gaps: your hotspot is a commonly/widely used primitive and is not enabled in Intel® MKL-DNN

Integration gaps: your hotspot uses Intel® MKL-DNN but runs much faster in a standalone benchmark (more details in the hands-on session)

Intel® MKL-DNN performance issue: your hotspot uses Intel® MKL-DNN but is very slow given its parameters

In any of these cases, feel free to contact the Intel® MKL-DNN team through the Github* page issues section.