LIBXSMM Samples

CP2K Artificial Benchmark

The first code sample given for LIBXSMM was a performance reproducer exercising the same set of kernels usually generated for CP2K's SMM library. The code sample attempted to model the way "matrix stacks" are processed in CP2K, however there are two different code paths in CP2K: (1) the "main" code path used when processing stacks on the host-side, and (2) a code path targeting offload devices. Beside of the host-sided parallelization via MPI (and perhaps OpenMP), the secondly mentioned code path relies on an additional level of parallelization (which is obviously necessary to drive a potentially highly parallel offload device). Also, the additional level of parallelism is not exactly "nested" in the sense that it participates on sharing the same resources as the host-side. In fact, this "artificial benchmark" (cp2k code sample) is modeling a code path as utilized in the secondly mentioned case (offload device).

Hello LIBXSMM

This example is focused on a specific functionality but may be considered as "Hello LIBXSMM". Copy and paste the example code and build it either manually and as described in our main documentation (see underneath the source code), or use GNU Make:

```
cd /path/to/libxsmm
make

cd /path/to/libxsmm/samples/hello
make

./hello
```

Alternatively, one can use the Bazel build system. To further simplify, Bazelisk is used to boot-strap Bazel:

```
cd /path/to/libxsmm/samples/hello
bazelisk build //...

./bazel-bin/hello
```

The C/C++ code given here uses LIBXSMM in header-only form (`#include <libxsmm_source.h>`), which is in contrast to the code shown in the main documentation. The Fortran code (`hello.f`) can be manually compiled like `gfortran -I/path/to/libxsmm/include hello.f -L/path/to/libxsmm/lib -libxsmmf -lxsmm -lxsmmnoblas -o hello` or as part of the above described invocation of GNU Make.

Magazine

Overview

This collection of code samples accompany an article written for issue #34 of the magazine The Parallel Universe, an Intel publication. The articles focuses on Blaze-, Eigen-, and LIBXSMM-variants of Small Matrix Multiplications (SMMs). The set of sample codes now also includes a variant relying on BLAS and a variant that showcases LIBXSMM's explicit batch-interface.

The baseline requirements are libraries that can operate on column-major storage order, "zero copy" when using existing memory buffers, and an API that is powerful enough to describe leading dimensions. Typically a library-internal parallelization of matrix multiplication is desired. However, for the magazine sample collection there is no performance gain expected since the matrices are small, and nested parallelism may only add overhead. Hence library-internal parallelism is disabled (BLAZE_USE_SHARED_MEMORY_PARALLELIZATION=0, EIGEN_DONT_PARALLELIZE). LIBXSMM provides parallelization on a per-functions basis and no global toggle is needed.

The sample codes rely on the minimum programming language supported by the library in question (API): C++ in case of Blaze and Eigen, and C in case of LIBXSMM (both C++ and Fortran interfaces are available as well). For Blaze and Eigen, the build-system ensures to not map implementation into a BLAS library (normally desired but this would not test the library-native implementation).

Results

To reproduce or repeat the performance measurements on a system of choice, all matrix operands are streamed by default. The file magazine.h can be edited to reproduce the desired combination (STREAM_A, STREAM_B, and STREAM_C). Whether or not matrix operands are streamed is motivated in publication. To reduce dependency on the compiler's OpenMP implementation, the benchmarks run single-threaded by default (`make OMP=1` can parallelize the batch of matrix multiplications). The outer/batch-level parallelization is also disabled to avoid accounting for proper first-touch memory population on multi-socket systems (NUMA). For the latter, the init-function (located in magazine.h) is not parallelized for simplicity.

```
cd libxsmm; make
cd samples/magazine; make
```

To run the benchmark kernels presented by the article:

```
./benchmark.sh
```

Please note that if multiple threads are enabled and used, an appropriate pin-strategy should be used (OMP_PLACES=threads, OMP_PROC_BIND=TRUE). To finally produce the benchmark charts:

```
./benchmark-plot.sh blaze
./benchmark-plot.sh eigen
./benchmark-plot.sh xsmm
```

The plot script relies at least on Gnuplot. ImageMagick (mogrify) can be also useful if PNGs are created, e.g., `./benchmark-plot.sh xsmm png 0` (the last argument disables single-file charts in contrast to multi-page PDFs created by default, the option also disables chart titles).

The set of kernels executed during the benchmark can be larger than the kernels presented by the plots: benchmark.set selects the kernels independent of the kernels executed (union).

NEK Sample Collection

This directory contains kernels taken from Nek{Box,5000}. They aim to represent most of the matrix-matrix workloads.

Please note that the mxm_std.f source code is protected by an (US) GOVERNMENT LICENSE, and under the copyright of the University of Chicago.

stpm

Small tensor-product multiple (stpm) replicates the axhelm kernel, which computes the Laplacian with spectral elements. Usage:

```
./stpm m n k size1 size
```

The elements are m-by-n-by-k, mode picks the LIBXSMM interface used, and size scales the number of spectral elements.

rstr

Restriction operator transforms elements from one size to another. This occurs in multi-grid, the convection operator, and, when the sizes are the same, the local Schwarz solves. Usage:

```
./rstr m n k mm nn kk size1 size
```

The input elements are m-by-n-by-k and the output elements are mm-by-nn-by-kk. When m=mm, n=nn, k=kk, this half of a Schwarz solve.

SMM Sample Collection

This collection of code samples exercises different memory streaming cases when performing the matrix multiplication $C_{mxn} = alpha \cdot A_{mxk} \cdot B_{kxn} + beta \cdot C_{mxn}$: (1) streaming the matrices A, B, and C which is usually referred as batched matrix multiplication, (2) streaming the inputs A and B but accumulating C within cache, (3) streaming the A and C matrices while B is kept in cache, (4) streaming the B and C matrices while A is kept in cache, and (4) not streaming any of the operands but repeating the very same multiplication until the requested number of matrix multiplications has been completed.

Beside of measuring the duration of a test case, the performance is presented in GFLOPS/s. As an alternative metric, the memory bandwidth is given (the artificial "cached" case omits to present the cache-memory bandwidth). The "pseudo-performance" given in FLOPS/cycle is an artificial scoring, it not only uses a non-standard formula for calculating the FLOPS ($2 * M * N * K - M * N$ rather than $2 * M * N * K$) but also relies on (pseudo-)clock cycles:

```
$ ./specialized.sh 0
m=32 n=32 k=32 size=87381 memory=2048.0 MB (DP)

Batched (A,B,C)...
        pseudo-perf.: 10.7 FLOPS/cycle
        performance: 23.9 GFLOPS/s
        bandwidth: 11.1 GB/s
        duration: 239 ms
Finished
```

There are two sub collections of samples codes: (1) a collection of C++ code samples showing either BLAS, Compiler-generated code (inlined code), LIBXSMM/dispatched, LIBXSMM/specialized functions to carry out the multiplication, and (2) a Fortran sample code showing BLAS versus LIBXSMM including some result validation.

C/C++ Code Samples: Command Line Interface (CLI)

- Takes an optional number (1st arg.) to select the streaming-case (0...8)
- Optionally takes the M, N, and K parameter of the GEMM in this order
- If only M is supplied, the N and K "inherit" the M-value
- Example I (A,B,C): ./specialized.sh 0 16 8 9
- Example II (A,B): ./specialized.sh 6 16

Fortran Code Sample: Command Line Interface (CLI)

- Optionally takes the M, N, and K parameter of the GEMM in this order
- Optional problem size (in MB) of the workload; M/N/K must have been supplied
- Optional total problem size (in MB) implying the number of repeated run
- If only M is supplied, the N and K are "inheriting" the M-value
- Shows the performance of each of the streaming cases
- Example I: ./smm.sh 16 8 9 1024 16384
- Example II: ./smm.sh 16

SPECFEM Sample

This sample contains a dummy example from a spectral-element stiffness kernel taken from SPECFEM3D_GLOBE.

It is based on a 4th-order, spectral-element stiffness kernel for simulations of elastic wave propagation through the Earth. Matrix sizes used are (25,5), (5,25) and (5,5) determined by different cut-planes through a three dimensional (5,5,5)-element with a total of 125 GLL points.

Usage Step-by-Step

This example needs the LIBXSMM library to be built with static kernels, using MNK="5 25" (for matrix size (5,25), (25,5) and (5,5)).

Build LIBXSMM

General Default Compilation In LIBXSMM root directory, compile the library with:

```
make MNK="5 25" ALPHA=1 BETA=0
```

Additional Compilation Examples Compilation using only single precision version and aggressive optimization:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3
```

For Sandy Bridge CPUs:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1
```

For Haswell CPUs:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=2
```

For Knights Corner (KNC) (and thereby creating a Sandy Bridge version):

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1 \
OFFLOAD=1 KNC=1
```

Installing libraries into a sub-directory workstation/:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1 \
OFFLOAD=1 KNC=1 \
PREFIX=workstation/ install-minimal
```

Build SpecFEM example code For default CPU host:

```
cd sample/specfem
make
```

For Knights Corner (KNC):

```
cd sample/specfem
make KNC=1
```

Additionally, adding some specific Fortran compiler flags, for example:

```
cd sample/specfem
make FCFLAGS="-O3 -fopenmp" [...]
```

Note that steps 1 and 2 could be shortened by specifying a "specfem" make target in the LIBXSMM root directory:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1 specfem
```

For Knights Corner, this would need two steps:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1 OFFLOAD=1 KNC=1
make OPT=3 specfem_mic
```

Run the Performance Test

For default CPU host:

```
./specfem.sh
```

For Knights Corner (KNC):

```
./specfem.sh -mic
```

Results

Using Intel Compiler suite: icpc 15.0.2, icc 15.0.2, and ifort 15.0.2.

Sandy Bridge - Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz Library compilation by (root directory):

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1
```

Single threaded example run:

```
cd sample/specfem
make; OMP_NUM_THREADS=1 ./specfem.sh
```

Output:

```
============================================================
average over           15 repetitions
 timing with Deville loops    =   0.1269
 timing with unrolled loops   =   0.1737 / speedup =   -36.87 %
 timing with LIBXSMM dispatch =   0.1697 / speedup =   -33.77 %
 timing with LIBXSMM prefetch =   0.1611 / speedup =   -26.98 %
 timing with LIBXSMM static   =   0.1392 / speedup =    -9.70 %
============================================================
```

Haswell - Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz Library compilation by (root directory):

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=2
```

Single threaded example run:

```
cd sample/specfem
make; OMP_NUM_THREADS=1 ./specfem.sh
```

Output:

```
============================================================
average over           15 repetitions
 timing with Deville loops    =   0.1028
 timing with unrolled loops   =   0.1385 / speedup =   -34.73 %
 timing with LIBXSMM dispatch =   0.1408 / speedup =   -37.02 %
 timing with LIBXSMM prefetch =   0.1327 / speedup =   -29.07 %
 timing with LIBXSMM static   =   0.1151 / speedup =   -11.93 %
============================================================
```

Multi-threaded example run:

```
cd sample/specfem
make OPT=3; OMP_NUM_THREADS=24 ./specfem.sh
```

Output:

```
OpenMP information:
  number of threads =           24

[...]

============================================================
average over           15 repetitions
 timing with Deville loops    =   0.0064
 timing with unrolled loops   =   0.0349 / speedup =  -446.71 %
 timing with LIBXSMM dispatch =   0.0082 / speedup =   -28.34 %
 timing with LIBXSMM prefetch =   0.0076 / speedup =   -19.59 %
 timing with LIBXSMM static   =   0.0068 / speedup =    -5.78 %
============================================================
```

Knights Corner - Intel Xeon Phi B1PRQ-5110P/5120D Library compilation by (root directory):

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 OFFLOAD=1 KNC=1
```

Multi-threaded example run:

```
cd sample/specfem
make FCFLAGS="-O3 -fopenmp -warn" OPT=3 KNC=1; ./specfem.sh -mic
```

Output:

```
OpenMP information:
  number of threads =          236

[...]

================================================================
average over           15 repetitions
 timing with Deville loops    =   0.0164
 timing with unrolled loops   =   0.6982 / speedup = -4162.10 %
 timing with LIBXSMM dispatch =   0.0170 / speedup =    -3.89 %
 timing with LIBXSMM static   =   0.0149 / speedup =     9.22 %
================================================================
```

Matrix Transpose (TCOPY)

Overview

This code sample aims to benchmark the performance of matrix transposes. The C/C++ and FORTRAN sample code differ slightly with the C/C++ code sample offering a richer set of command line options as well as build settings available inside of the translation unit.

The available command line options of the sample code may be reviewed by looking into the source code. Generally, the idea is to support the following:

transpose [<kind> [<m> [<n> [<ldi> [<ldo>]]]]]
transposef [<m> [<n> [<ldi> [<ldo>]]]]

Above, `m` and `n` specify the matrix shape, and `ldi` the leading dimension of the matrix. The argument `ldo` allows to specify an output dimension, which may differ from `ldi`. The transpose kind shall be either out-of-place (`o`) or in-place (`i`).

Running the C sample code may look like:

```
$ ./transpose.sh o 20000
m=20000 n=20000 ldi=20000 ldo=20000 size=3052MB (double, out-of-place)
        bandwidth: 18.8 GB/s
        duration: 159 ms
```

Instead of executing a wrapper script, one may affinitize the multi-threaded execution manually (OpenMP runtime). In case of an executable built using the Intel Compiler this may look like:

```
LIBXSMM_VERBOSE=2 KMP_AFFINITY=balanced,granularity=fine,1 \
./transpose o 20000
m=20000 n=20000 ldi=20000 ldo=20000 size=3052MB (double, out-of-place)
        bandwidth: 21.1 GB/s
        duration: 141 ms

Registry: 20 MB (gemm=0 mcopy=0 tcopy=1)
```

In the above case one can see from the verbose output (`LIBXSMM_VERBOSE=2`) that one kernel (tcopy) served transposing the entire matrix. To avoid duplicating JIT-kernels under contention (code registry), one may also consider `LIBXSMM_TRYLOCK=1`, which is available per API-call as well.

OpenTuner

To tune the tile sizes ("block sizes") internal to LIBXSMM's transpose routine, the OpenTuner extensible framework for program autotuning can be used. In case of issues during the tuning phase ("no value has been set for this column"), please install the latest 1.2.x revision of SQLAlchemy (`pip install sqlalchemy==1.2.19`). A tuning script (`transpose_opentuner.py`) is provided, which accepts a range of matrix sizes as command line arguments.

transpose_opentuner.py <begin> <end> [*nexperiments-per-epoch*] [*tile-size-m*] [*tile-size-n*]

To start a tuning experiment for a new set of arguments, it is highly recommended to start from scratch. Otherwise the population of previously generated tuning results is fetched from a database and used to tune an eventually unrelated range of matrix shapes. To get reliable timings, the total time for all experiments per epoch is minimized (hence a different number of experiments per epoch also asks for an own database). Optionally, the initial block size can be seeded (`tile-size-m` and `tile-size-n`).

```
rm -rf opentuner.db
```

The script tunes matrices with randomized shape according to the specified range. The leading dimension is chosen tightly for the experiments. The optimizer not only maximizes the performance but also minimizes the value of $M * N$ (which also helps to prune duplicated results due to an additional preference).

```
rm -rf opentuner.db
./transpose_opentuner.py --no-dups 1 1024 1000

rm -rf opentuner.db
./transpose_opentuner.py --no-dups 1024 2048 100

rm -rf opentuner.db
./transpose_opentuner.py --no-dups 2048 3072 20

rm -rf opentuner.db
./transpose_opentuner.py --no-dups 3072 4096 20

rm -rf opentuner.db
./transpose_opentuner.py --no-dups 4096 5120 16

rm -rf opentuner.db
./transpose_opentuner.py --no-dups 5120 6144 12

rm -rf opentuner.db
./transpose_opentuner.py --no-dups 6144 7168 8

rm -rf opentuner.db
./transpose_opentuner.py --no-dups 7168 8192 6
```

The tuning script uses the environment variables `LIBXSMM_TCOPY_M` and `LIBXSMM_TCOPY_N`, which are internal to LIBXSMM. These variables are used to adjust certain thresholds in `libxsmm_otrans` or to request a specific tiling-scheme inside of the `libxsmm_otrans_omp` routine.

XGEMM: Tiled GEMM Routines

Overview

This sample code calls the `libxsmm_?gemm_omp` routines provided by the LIBXSMM extension library (`libxsmmext`). These routines are meant for big(ger) xGEMM routines, and thereby provide an OpenMP-based parallelization.

The driver program (`xgemm.c`) currently accepts all typical GEMM arguments (except for the transposition specifier): `m`, `n`, `k`, `lda`, `ldb`, `ldc`, `alpha`, and `beta`. All arguments are optional (or will inherit defaults from previously specified arguments). Matrix transposition as part of the `libxsmm_?gemm_omp` routines will become available in an upcoming release of LIBXSMM. Please also note that unsupported Alpha or Beta values will cause a fall back to the related BLAS routine. The single-precision matrix multiplications require to change the `ITYPE` in `xgemm.c`.

```
./xgemm.sh 2000
```

OpenTuner

To tune the tile sizes ("block sizes") internal to LIBXSMM, the OpenTuner extensible framework for program autotuning can be used. In case of issues during the tuning phase ("no value has been set for this column"), please install the latest 1.2.x revision of SQLAlchemy (`pip install sqlalchemy==1.2.19`). A tuning script (`xgemm_opentuner.py`) is provided, which optionally accepts a list of grouped parameters as command line arguments. The syntax of the arguments is per LIBXSMM's `MNK` build-option, and expands to "triplets" specifying the matrix shapes. For instance, four matrix multiplications of square-matrices can be benchmarked and tuned using the following command.

```
./xgemm_opentuner.py 1024,1280,1536,1792
```

To start a tuning experiment for a new set of arguments, it is highly recommended to start from scratch. Otherwise the population of previously generated tuning results is fetched from a database and used to tune an unrelated range of matrix shapes. Optionally, the initial block size can be seeded (`tile-size-m`, `tile-size-n`, and `tile-size-k`).

```
rm -rf opentuner.db
```

The script tunes the geometric mean of the performance for each of the requested triplets. However, the optimizer not only maximizes the performance but also minimizes the value of $M * N * K$ (which also helps to prune duplicated results due to an additional preference). As a limitation of the current implementation, the multiplication kernels are not accompanied by copy-kernels (and not accompanied by transpose kernels). This negatively impacts performance on power-of-two matrix shapes (POT) due to trashing the LLC. However, it has been found, that tuning for POT shapes likely achieves superior performance when compared to tuning for non-POT shapes of the same range.

```
rm -rf opentuner.db
./xgemm_opentuner.py --no-dups 192,256,320,512,768

rm -rf opentuner.db
./xgemm_opentuner.py --no-dups 1024,1280,1536,1792

rm -rf opentuner.db
./xgemm_opentuner.py --no-dups 2048,2304,2560,2816

rm -rf opentuner.db
./xgemm_opentuner.py --no-dups 3072,3328,3584,3840

rm -rf opentuner.db
./xgemm_opentuner.py --no-dups 4096,4416,4736

rm -rf opentuner.db
./xgemm_opentuner.py --no-dups 5120,5440,5760

rm -rf opentuner.db
./xgemm_opentuner.py --no-dups 6144,6464,6784

rm -rf opentuner.db
./xgemm_opentuner.py --no-dups 7168,7488,7808
```

Above, the series of matrix multiplications from 192-8K is separately tuned in eight ranges. The tuning script uses the environment variables `LIBXSMM_TGEMM_M`, `LIBXSMM_TGEMM_N`, and `LIBXSMM_TGEMM_K` which are internal to LIBXSMM. These variables are used to request a specific tiling-scheme within LIBXSMM's `libxsmm_?gemm_omp` routines.

1D Dilated Convolutional Layer

This package contains the optimized kernels for the 1D dilated convolutional layer. The C++ implementation has code for both FP32 and BF16 formats. You can run this code on AVX-512 enabled CPUs. Ex. - Cascade Lake or Cooper lake.

Install instructions

Install PyTorch in an anaconda or virtual environment before installing the package. Use GCC version 8.3.0 or higher.

```
conda activate environment              # Activate anaconda or virtual environment containing PyTorch

cd Conv1dOpti-extension/
python setup.py install                 # Install package
cd ..
```

A user can either use run.sh script to run the torch_example.py code, or he/she can follow the following commands.

```
export LD_LIBRARY_PATH={LIBXSMM_ROOT/lib}           # Set LD_LIBRARY_PATH
export OMP_NUM_THREADS=28                           # Set number of threads
export KMP_AFFINITY=compact,1,0,granularity=fine    # Set KMP affinity

python torch_example.py                             # Run the pytorch example
```

In the previous example, we compare "nn.Conv1d" layer with our optimized "Conv1dOpti" layer. The example shows how "nn.Conv1d" can be replaced with "Conv1dOpti" layer in a neural network without requiring any other change. The optimized python layer can be imported using "from Conv1dOpti_ext import Conv1dOpti" in python. The example checks the accuracy of the results and calculates the computation time of both layers.

Limitations of the current version

- Keep padding=0 in the options. The current layer doesn't do padding. Explicit padding is needed for the optimized convolutional layer. You can use the example for reference.
- Optimized convolutional layer code can only run with stride = 1.
- Similarly, apply the nonlinearity (Ex. ReLU) separately.

To run code in BFloat16, set enable_BF16 flag to True. BFloat16 code runs only when the parameters of Input width, number of filters and input channels to the layer are even number. Ex. - Filters = 16, Channels = 16, Input width = 60000, enable_BF16 = True BF16 run If any of the previous parameter is odd number then code runs in FP32 format.

Keep batch size as multiple of ununtilized cores (Ex. - 28, 56, 84, 128 .... on a 28 core cascade lake) for optimal performance with the Conv1dOpti layer. Each batch will run on a seperate thread thus performance may go down if some core are not free, or batch size is not equal to the number of free cores. Keep the batch size as power of 2 with the MKLDNN backend (Conv1d) for optimal performance.

Deep Learning with GxM

Compiling and Building GxM

1. Install Pre-requisite Libraries: Google logging module (glog), gflags, Google's data interchange format (Protobuf), OpenCV, LMDB
2. In Makefile.config, set GXM_LIBRARY_PATH variable to the path containing above libraries
3. In Makefile.config, set LIBXSMM_PATH variable to the path containing LIBXSMM library
4. Set/clear other flags in Makefile.config as required (see associated comments in Makefile.config)
5. source setup_env.sh
6. make clean; make

Running GxM

The network topology definitions directory is "model_zoo". Currently, it contains definitions for AlexNet (without LRN), ResNet-50, Inception v3 along with CIFAR10 and MNIST as simple test definitions. Each topology definition is in a .prototxt file. ResNet-50 can run with "dummy data", raw JPEG image data or with LMDB. Filenames indicate the data source along with the minibatch size. Inception v3 runs only with compressed LMDB data.

The hyperparameter definitions for each topology are also in the corresponding directory under "model_zoo" in a .prototxt file with the suffix "solver". For a single-node, this file is called solver.prototxt. For multi-node the filename also contains the global minibatch size (=single node minibatch size x number of nodes);, e.g., solver_896.prototxt contains hyperparameters for MB=56 per node and 16 nodes. The "solver*" file also contains a flag that specifies whether to start execution from a checkpoint (and thus read load weights from the "./weights" directory) or from scratch; by default execution starts from scratch.

Optimal parallelization of Convolutional layers in LIBXSMM happens when the number of OpenMP threads = MiniBatch. Therefore, on Xeon

```
export OMP_NUM_THREADS=<MiniBatch>
export KMP_AFFINITY=compact,granularity=fine,1,0
```

The command line for a training run is:

```
./build/bin/gxm train <topology filename> <hyperparameter filename>
```

For example:

```
./build/bin/gxm train model_zoo/resnet/1_resnet50_dummy_56.prototxt model_zoo/resnet/solver.prototxt
```

Preping on RHEL 8.0 / CentOS 8.0

```
dnf install protobuf
wget http://mirror.centos.org/centos/8/PowerTools/x86_64/os/Packages/protobuf-compiler-3.5.0-7.el8.x86
dnf install protobuf-compiler-3.5.0-7.el8.x86_64.rpm
wget http://mirror.centos.org/centos/8/PowerTools/x86_64/os/Packages/protobuf-devel-3.5.0-7.el8.x86_64
dnf install protobuf-devel-3.5.0-7.el8.x86_64.rpm
dnf install lmdb
dnf install lmdb-devel
wget http://repo.okay.com.mx/centos/8/x86_64/release/opencv-devel-3.4.1-9.el8.x86_64.rpm
wget http://repo.okay.com.mx/centos/8/x86_64/release/opencv-3.4.1-9.el8.x86_64.rpm
dnf install opencv-3.4.1-9.el8.x86_64.rpm
dnf install opencv-devel-3.4.1-9.el8.x86_64.rpm
wget http://mirror.centos.org/centos/8/PowerTools/x86_64/os/Packages/gflags-2.1.2-6.el8.x86_64.rpm
wget http://mirror.centos.org/centos/8/PowerTools/x86_64/os/Packages/gflags-devel-2.1.2-6.el8.x86_64.r
dnf install gflags-2.1.2-6.el8.x86_64.rpm
dnf install gflags-devel-2.1.2-6.el8.x86_64.rpm
wget http://mirror.centos.org/centos/8/PowerTools/x86_64/os/Packages/glog-devel-0.3.5-3.el8.x86_64.rpm
wget http://mirror.centos.org/centos/8/PowerTools/x86_64/os/Packages/glog-0.3.5-3.el8.x86_64.rpm
dnf install glog-0.3.5-3.el8.x86_64.rpm
dnf install glog-devel-0.3.5-3.el8.x86_64.rpm
```

Make sure that the makefile follows the OpenCV Ver 3 path!

DNN Training with Incremental Sparsification + Sparse JIT Kernels

This project contains code for the following DNN models

1. Resnet - ported from link
2. Transformer - ported from link
3. DLRM - ported from link
4. PCL_MLP - A python extension of the `torch.nn.Linear` module that uses efficient sparse JIT kernels for matrix multiplication (supports forward, backward and update pass) - ported from link

Features

1. Training scripts for all three models located at the root of each directory in a form of a shell file
2. By specifying each of the four parameters, the pruning criteria (magnitude-based or random-based), the pruning start time and end time and target sparsity you can apply incremental sparsity to model weights for training
3. Additionally, by specifying a tensorboard log directory, one can examine training logs and metrics using tensorboard.

Data preparation

Each model requires an extensive amount of data to be properly stress-tested against incremental sparsity. According to The State of Sparsity and by extensive experimentation, using a relatively small dataset or an overparameterized model may lead to false performance implications. For instance, when a ResNet-50 model is trained with the CIFAR-10 dataset or if the base Transformer is trained with a limited sentence pair dataset (i.e., EN-VI) it may seem as if the model isn't impacted even with extremely high sparsity since the model was overdetermined to begin with.

- For Resnet
- For Resnet training, a smaller subset of ImageNet was used, called ImageNette due to its massiveness in size. Download from here.
- For Transformer
- As a neural machine translation task, the transformer model requires the WMT2014 EN_DE dataset. Preprocessing steps are described here
- For DLRM
- Training the DLRM requires the terabyte dataset link

Running scripts

Each project consists of two scripts: a script that launches `sbatch` scripts for experimenting various target sparsities (usually named as `launch_pruning_runs.sh`)and a script that runs a single experiment. Use accordingly.

1. ResNet model `./launch_pruning_jobs.sh ${TARGET_SPARSITY}` or `python train.py ${TARGET_SPARSITY}`
2. Transformer(FAIRSEQ) model `./launch_pruning_runs.sh` or `./prune_en_de.sh ${TARGET_SPARSITY} ${PRUNE_TYPE} ${EMB}` where PRUNE_TYPE is either `magnitude` or `random` and EMB indicates whether the embedding portion is pruned alongside the weights
3. DLRM model `./launch_pruning_runs.sh` or `./run_terabyte.sh ${TARGET_SPARSITY} ${PRUNE_TYPE}` where PRUNE_TYPE is either `magnitude` or `random`

Xsmm LSTM

This code may be integrated with Tensorflow to make use of LIBXSMM's LSTM. Support for creating a Python wheel and a pip package can be found in the directory as well.

Dispatch

Microbenchmark

This code sample benchmarks the performance of (1) the dispatch mechanism, and (2) the time needed to JIT-generate code for the first time. Both mechanisms are relevant when replacing GEMM calls (see Call Wrapper section of the reference documentation), or in any case of calling LIBXSMM's native GEMM functionality.

Command Line Interface (CLI)

- Optionally takes the number of dispatches/code-generations (default: 10000).
- Optionally takes the number of threads (default: 1).

Measurements (Benchmark)

- Duration of an empty function call (serves as a reference timing).
- Duration to find an already generated kernel (cached/non-cached).
- Duration to JIT-generate a GEMM kernel.

In case of a multi-threaded benchmark, the timings represent a highly contended request (worst case). For thread-scaling, it can be observed that read-only accesses (code dispatch) stay roughly with a constant duration whereas write-accesses (code generation) are serialized and hence the duration scales linearly with the number of threads.

The Fortran example (`dispatch.f`) could use `libxsmm_dmmdispatch` (or similar) like the C code (`dispatch.c`) but intentionally shows the lower-level dispatch interface `libxsmm_xmmdispatch` and also omits using the LIBXSMM module. Not using the module confirms: the same task can be achieved by relying only on FORTRAN 77 language level.

User-Data Dispatch

Further, another Fortran example about user-data dispatch is not exactly a benchmark. Dispatching user-data containing multiple kernels can obviously save multiple singular dispatches. The C interface for dispatching user-data is designed to follow the same flow as the Fortran interface.

MHD Image I/O

This code sample aims to provide a simple piece of code, which takes an image and produces a visual result using LIBXSMM's MHD image file I/O. Performing a single convolution is *not* a showcase of LIBXSMM's Deeplearning as the code only runs over a single image with one channel. LIBXSMM's CNNs are vectorized over image channels (multiple images) according to the native vector-width of the processor and otherwise fall back to a high-level implementation.

Note: For high-performance deep learning, please refer to the collection of CNN layer samples.

The executable can run with the following arguments (all arguments are optional):

mhd [<filename-in> [<nrepeat> [<kw> [<kh>] [<filename-out>]]]]

For stable timing (benchmark), the key operation (convolution) may be repeated (`nrepeat`). Further, `kw` and `kh` can specify the kernel-size of the convolution. The `filename-in` and `filename-out` name MHD-files used as input and output respectively. The `filename-in` may be a pseudo-file (that does not exist) but specify the image resolution of generated input (`w`[`xh`] where the file `wxh.mhd` stores the generated image data). To load an image from a familiar format (JPG, PNG, etc.), please have a look at Meta Image File I/O.

Scratch Memory Allocation (Microbenchmark)

This code sample aims to benchmark the performance of the scratch memory allocation. This facility is a viable option to satisfy the need for temporary memory when using the DNN domain of LIBXSMM (small convolutions). Although any kind of readable/writable buffer can be bound to a convolution handle, LIBXSMM's `libxsmm_aligned_scratch` features a thread-safe linear allocator mechanism which can help to lower allocation overhead.

Wrapped DGEMM

This code sample is calling DGEMM and there is no dependency on the LIBXSMM API as it only relies on LAPACK/BLAS interface. Two variants are linked when building the source code: (1) code which is dynamically linked against LAPACK/BLAS, (2) code which is linked using `--wrap=`*symbol* as possible when using a GNU GCC compatible tool chain. For more information, see the Call Wrapper section of the reference documentation.

The same (source-)code will execute in three flavors when running `dgemm-test.sh`: (1) code variant which is dynamically linked against the originally supplied LAPACK/BLAS library, (2) code variant which is linked using the wrapper mechanism of the GNU GCC tool chain, and (3) the first code but using the LD_PRELOAD mechanism (available under Linux).

Command Line Interface (CLI)

- Optionally takes the number of repeated DGEMM calls
- Shows the performance of the workload (wall time)