LIBXSMM

LIBXSMM is a library for small dense and small sparse matrix-matrix multiplications as well as for deep learning primitives such as small convolutions targeting Intel Architecture. Small matrix multiplication kernels are generated for the following instruction set extensions: Intel SSE, Intel AVX, Intel AVX2, IMCI (KNCni) for Intel Xeon Phi coprocessors ("KNC"), and Intel AVX-512 as found in the Intel Xeon Phi processor family (Knights Landing "KNL", Knights Mill "KNM") and Intel Xeon processors (Skylake-SP "SKX"). Historically small matrix multiplications were only optimized for the Intel Many Integrated Core Architecture "MIC") using intrinsic functions, meanwhile optimized assembly code is targeting all afore mentioned instruction set extensions (static code generation), and Just-In-Time (JIT) code generation is targeting Intel AVX and beyond. Optimized code for small convolutions is JIT-generated for Intel AVX2 and Intel AVX-512.

What is a small matrix multiplication? When characterizing the problem-size using the M, N, and K parameters, a problem-size suitable for LIBXSMM falls approximately within $(M\ N\ K)^{1/3} <= 128$ (which illustrates that non-square matrices or even "tall and skinny" shapes are covered as well). The library is typically used to generate code up to the specified threshold. Raising the threshold may not only generate excessive amounts of code (due to unrolling in M or K dimension), but also miss to implement a tiling scheme to effectively utilize the cache hierarchy. For auto-dispatched problem-sizes above the configurable threshold (explicitly JIT'ted code is **not** subject to the threshold), LIBXSMM is falling back to BLAS. In terms of GEMM, the supported kernels are limited to *Alpha := 1*, *Beta := { 1, 0 }*, *TransA := 'N'*, and *TransB = 'N'*.

What is a small convolution? In the last years, new workloads such as deep learning and more specifically convolutional neural networks (CNN) emerged, and are pushing the limits of today's hardware. One of the expensive kernels is a small convolution with certain kernel sizes (3, 5, or 7) such that calculations in the frequency space is not the most efficient method when compared with direct convolutions. LIBXSMM's current support for convolutions aims for an easy to use invocation of small (direct) convolutions, which are intended for CNN training and classification. The Interface is currently ramping up, and the functionality increases quickly towards a broader set of use cases.

For more questions and answers, please have a look at https://github.com/hfp/libxsmm/wiki/Q&A.

Documented functionality and available domains:

- MM: Matrix Multiplication
- DNN: Convolutional Deep Neural Networks
- AUX: Service Functions
- PERF: Performance
- BE: Backend

For additional functionality, please have a look at https://github.com/hfp/libxsmm/tree/master/include.

Build Instructions

Classic Library (ABI)

The build system relies on GNU Make (typically associated with the `make` command, but e.g. FreeBSD is calling it `gmake`). The build can be customized by using key-value pairs. Key-value pairs can be supplied in two ways: (1) after the "make" command, or (2) prior to the "make" command (`env`) which is effectively the same as exporting the key-value pair as an environment variable (`export`, or `setenv`). Both methods can be mixed, however the second method may require to supply the `-e` flag. Please note that the CXX, CC, and FC keys are considered in any case.

To generate the interface of the library inside of the 'include' directory and to build the static library (by default, STATIC=1 is activated), simply run the following command:

```
make
```

On CRAY systems, the CRAY Compiling Environment (CCE) should be used regardless of using the CRAY compiler, the Intel Compiler, or the GNU Compiler Collection (GCC). The latter is achieved by switching the programming environment to the desired compiler, but always relying on:

```
make CXX=CC CC=cc FC=ftn
```

If the build process is not successful, it may help to avoid more advanced GCC flags. This is useful with a tool chain, which pretends to be GCC-compatible (or is treated as such) but fails to consume the afore mentioned flags. In such a case one may raise the compatibility:

```
make COMPATIBLE=1
```

By default, only the non-coprocessor targets are built (OFFLOAD=0 and KNC=0). In general, the subfolders of the 'lib' directory are separating the build targets where the 'mic' folder is containing the native library (KNC=1) targeting the Intel Xeon Phi coprocessor ("KNC"), and the 'intel64' folder is storing either the hybrid archive made of CPU and coprocessor code (OFFLOAD=1), or an archive which is only containing the CPU code. By default, an OFFLOAD=1 implies KNC=1.

To remove intermediate files, or to remove all generated files and folders (including the interface and the library archives), run one of the following commands:

```
make clean
make realclean
```

By default, LIBXSMM uses the JIT backend which is automatically building optimized code. However, one can also statically specialize the matrix sizes (M, N, and K values), for convolutions the options below can be ignored:

```
make M="2 4" N="1" K="$(echo $(seq 2 5))"
```

The above example is generating the following set of (M,N,K) triplets:

```
(2,1,2), (2,1,3), (2,1,4), (2,1,5),
(4,1,2), (4,1,3), (4,1,4), (4,1,5)
```

The index sets are in a loop-nest relationship (M(N(K))) when generating the indices. Moreover, an empty index set resolves to the next non-empty outer index set of the loop nest (including to wrap around from the M to K set). An empty index set is not participating anymore in the loop-nest relationship. Here is an example of generating multiplication routines which are "squares" with respect to M and N (N inherits the current value of the "M loop"):

```
make M="$(echo $(seq 2 5))" K="$(echo $(seq 2 5))"
```

An even more flexible specialization is possible by using the MNK variable when building the library. It takes a list of indexes which are eventually grouped (using commas):

```
make MNK="2 3, 23"
```

Each group of the above indexes is combined into all possible triplets generating the following set of (M,N,K) values:

```
(2,2,2), (2,2,3), (2,3,2), (2,3,3),
(3,2,2), (3,2,3), (3,3,2), (3,3,3), (23,23,23)
```

Of course, both mechanisms (M/N/K and MNK based) can be combined using the same command line (make). Static optimization and JIT can also be combined (no need to turn off the JIT backend). Testing the library is supported by a variety of targets with "test" and "test-all" being the most prominent for this matter.

Functionality of LIBXSMM, which is unrelated to GEMM can be used without introducing a dependency to BLAS. This can be achieved in two ways: (1) building a special library with `make BLAS=0`, or (2) linking the application against the 'libxsmmnoblas' library. Some care must be taken with any matrix multiplication which does not appear to require BLAS for the given test arguments. However, it may fall back to BLAS (at runtime of the application), if an unforeseen input is given (problem-size, or unsupported GEMM arguments).

NOTE: By default, a C/C++ and a FORTRAN compiler is needed (some sample code is written in C++). Beside of specifying the compilers (`make CXX=g++ CC=gcc FC=gfortran` and maybe `AR=ar`), the need for a FORTRAN compiler can be relaxed (`make FC=` or `make FORTRAN=0`). The latter affects the availability of the MODule file and the corresponding 'libxsmmf' library (the interface 'libxsmm.f' is still generated). FORTRAN code can make use of LIBXSMM in three different ways:

- By relying on the module file, and by linking against 'libxsmmf', 'libxsmm', and (optionally) 'libxsmmext',
- By including the interface 'libxsmm.f' and linking against 'libxsmm', and (optionally) 'libxsmmext', or
- By declaring e.g., `libxsmm_?gemm` (BLAS signature) and linking 'libxsmm' (and 'libxsmmext' if needed).

At the expense of a limited set of functionality (`libxsmm_?gemm[_omp]`, `libxsmm_blas_?gemm`, and `libxsmm_[s|d]otrans[_omp]`), the latter method also works with FORTRAN 77 (otherwise the FORTRAN 2003 standard is necessary). For the "omp" functionality, the 'libxsmmext' library needs to be present at the link line. For no code change at all, the Call Wrapper might be of interest.

Link Instructions

The library is agnostic with respect to the threading-runtime, and therefore an application is free to use any threading runtime (e.g., OpenMP). The library is also thread-safe, and multiple application threads can call LIBXSMM's routines concurrently. Forcing OpenMP (OMP=1) for the entire build of LIBXSMM is not supported and untested ('libxsmmext' is automatically built with OpenMP enabled).

Similarly, an application is free to choose any BLAS or LAPACK library (if the link model available on the OS supports this), and therefore linking GEMM routines when linking LIBXSMM itself (by supplying BLAS=1|2) may prevent a user from making this decision at the time of linking the actual application.

NOTE: LIBXSMM does not support to dynamically link 'libxsmm' or 'libxsmmext' ("so"), when BLAS is linked statically ("a"). If BLAS is linked statically, the static version of LIBXSMM must be used!

Header-Only

Version 1.4.4 introduced support for "header-only" usage in C and C++. By only including 'libxsmm_source.h' allows to get around building the library. However, this gives up on a clearly defined application binary interface (ABI). An ABI may allow for hot-fixes after deploying an application (when relying on the shared library form), and ensures to only rely on the public interface of LIBXSMM. In contrast, the header-only form not only exposes the internal implementation of LIBXSMM but may also reduce the turnaround time during development of an application (due to longer compilation times). The header file is intentionally named "libxsmm_**source**.h" since it relies on the src folder (with the implications as noted earlier).

To use the header-only form, 'libxsmm_source.h' needs to be generated. The build target shown below ('header-only') has been introduced in LIBXSMM 1.6.2, but `make cheader` can be used alternatively (or instead with earlier versions). Generating the C interface is necessary since LIBXSMM should be configured (see configuration template).

```
make header-only
```

NOTE: Differences between C and C++ makes a header-only implementation (which is portable between both languages) considerably "techy". Mixing C and C++ translation units (which rely on the header-only form of the library) is not supported. Also, remember that building an application now shares the same build settings with LIBXSMM. The latter is important for instance with respect to debug code (`-DNDEBUG`).

Installation

Installing LIBXSMM makes possibly the most sense when combining the JIT backend (enabled by default) with a collection of statically generated SSE kernels (by specifying M, N, K, or MNK). If the JIT backend is not disabled, statically generated kernels are only registered for dispatch if the CPUID flags at runtime are not supporting a more specific instruction set extension (code path). Since the JIT backend does not support or generate SSE code by itself, the library is compiled by selecting SSE code generation if not specified otherwise (AVX=1|2|3, or with SSE=0 falling back to an "arch-native" approach). Limiting the static code path to SSE4.2 allows to practically target any deployed system, however using SSE=0 and AVX=0 together is falling back to generic code, and any static kernels are not specialized using the assembly code generator.

There are two main mechanisms to install LIBXSMM (both mechanisms can be combined): (1) building the library in an out-of-tree fashion, and (2) installing into a certain location. Building in an out-of-tree fashion looks like:

```
cd libxsmm-install
make -f /path/to/libxsmm/Makefile
```

For example, installing into a specific location (incl. a selection of statically generated Intel SSE kernels) looks like:

```
make MNK="1 2 3 4 5" PREFIX=/path/to/libxsmm-install install
```

Performing `make install-minimal` omits the documentation (default: 'PREFIX/share/libxsmm'). Moreover, PINCDIR, POUTDIR, PBINDIR, and PDOCDIR allow to customize the locations underneath of the PREFIX location. To build a general package for an unpredictable audience (Linux distribution, or similar), it is advised to not over-specify or customize the build step i.e., JIT, SSE, AVX, OMP, BLAS, etc. should not be used. The following is building and installing a complete set of libraries where the generated interface matches both the static and the shared libraries:

```
make PREFIX=/path/to/libxsmm-install STATIC=0 install
make PREFIX=/path/to/libxsmm-install install
```

Interface for Matrix Multiplication

The function domain for dense Matrix Multiplications (MM) provides special support for Small Matrix Multiplications (SMM) as well as the industry-standard interface for GEneral Matrix Matrix multiplication (GEMM). All details can be found in a separate document.

Interface for Convolutions

The function domain for Deep Neural Networks (DNN) is detailed by a separate document. Please also note the separate guide for Getting Started using TensorFlow™ and LIBXSMM.

Service Functions

For convenient operation of the library and to ease integration, a few service routines are available. They do not exactly belong to the core functionality of LIBXSMM (SMM or DNN domain), but users are encouraged to rely on these routines of the API. There are two categories: (1) routines which are available for C and Fortran, and (2) routines which are only available with the C interface.

The service function domain (AUX) contains routines for:

- Getting and setting the target architecture
- Getting and setting the verbosity
- Measuring time durations (timer)
- Loading and storing data (I/O)
- Allocating memory

The details can be found in a separate document.

Running

Verbose Mode

The verbose mode allows for an insight into the code dispatch mechanism by receiving a small tabulated statistic as soon as the library terminates. The design point for this functionality is to not impact the performance of any critical code path i.e., verbose mode is always enabled and does not require symbols (SYM=1) or debug code (DBG=1). The statistics appears (`stderr`) when the environment variable LIBXSMM_VERBOSE is set to a non-zero value. For example:

```
LIBXSMM_VERBOSE=1 ./myapplication
[... application output]

HSW/SP       TRY    JIT    STA    COL
   0..13       0      0      0      0
  14..23       0      0      0      0
 24..128       3      3      0      0
```

The tables are distinct between single-precision and double-precision, but either table is pruned if all counters are zero. If both tables are pruned, the library shows the code path which would have been used for JIT'ting the code: `LIBXSMM_TARGET=hsw` (otherwise the code path is shown in the table's header). The actual counters are collected for three buckets: small kernels ($\text{MNK}^{1/3} <= 13$), medium-sized kernels ($13 < \text{MNK}^{1/3} <= 23$), and larger kernels ($23 < \text{MNK}^{1/3} <= 128$; the actual upper bound depends on LIBXSMM_MAX_MNK as selected at compile-time). Keep in mind, that "larger" is supposedly still small in terms of arithmetic intensity (which grows linearly with the kernel size). Unfortunately, the arithmetic intensity depends on the way a kernel is used (which operands are loaded/stored into main memory) and it is not performance-neutral to collect this information.

The TRY counter represents all attempts to register statically generated kernels, and all attempts to dynamically generate and register kernels. The TRY counter includes rejected JIT requests due to unsupported GEMM arguments. The JIT and STA counters distinct the successful cases of the afore mentioned event (TRY) into dynamically (JIT) and statically (STA) generated code. In case the capacity ($\text{O}(n) = 10^5$) of the code registry is exhausted, no more kernels can be registered although further attempts are not prevented. Registering many kernels ($\text{O}(n) = 10^3$) may ramp the number of hash key collisions (COL), which can degrade performance. The latter is prevented if the small thread-local cache is utilized effectively.

Since explicitly JIT-generated code (`libxsmm_?mmdispatch`) does not fall under the THRESHOLD criterion, the above table is extended by one line if large kernels have been requested. This indicates a missing threshold-criterion (customized dispatch), or asks for cache-blocking the matrix multiplication. The latter is already implemented by

LIBXSMM's "medium-sized" GEMM routines (`libxsmm_?gemm_omp`), which perform a tiled multiplication. Setting a verbosity level of at least two summarizes the number of registered JIT-generated kernels, which includes the total size and counters for GEMM, MCOPY (matrix copy), and TCOPY (matrix transpose) kernels.

```
Registry: 20 MB (gemm=0 mcopy=14 tcopy=0)
```

NOTE: Setting LIBXSMM_VERBOSE to a negative value will binary-dump each generated JIT kernel to a file with each file being named like the function name shown in Intel VTune. Disassembly of the raw binary files can be accomplished by:

```
objdump -D -b binary -m i386 -M x86-64 [JIT-dump-file]
```

Call Trace

During the initial steps of employing the LIBXSMM API, one may rely on a debug version of the library (`make DBG=1`). The latter also implies console output (`stderr`) in case of an error/warning condition inside of the library. It is also possible to print the execution flow (call trace) inside of LIBXSMM (can be combined with DBG=1 or OPT=0):

```
make TRACE=1
```

Building an application which traces calls (inside of the library) requires the shared library of LIBXSMM, alternatively the application is required to link the static library of LIBXSMM in a dynamic fashion (GNU tool chain: `-rdynamic`). Tracing calls (without debugger) can be the accomplished by an environment variable called LIBXSMM_TRACE.

```
LIBXSMM_TRACE=1 ./myapplication
```

Syntactically up to three arguments separated by commas (which allows to omit arguments) are taken (*tid*,*i*,*n*): *tid* signifies the ID of the thread to be traced with 1…NTHREADS being valid and where LIBXSMM_TRACE=1 is filtering for the "main thread" (in fact the first thread running into the trace facility); grabbing all threads (no filter) can be achieved by supplying a negative id (which is also the default when omitted). The second argument is pruning higher levels of the call-tree with *i=1* being the default (level zero is the highest at the same level as the main function). The last argument is taking the number of inclusive call levels with *n=-1* being the default (signifying no filter).

Although the `ltrace` (Linux utility) provides similar insight, the trace facility might be useful due to the afore mentioned filtering expressions. Please note that the trace facility is severely impacting the performance (even with LIBXSMM_TRACE=0), and this is not just because of console output but rather since inlining (internal) functions might be prevented along with additional call overhead on each function entry and exit. Therefore, debug symbols can be also enabled separately (`make SYM=1`; implied by TRACE=1 or DBG=1) which might be useful when profiling an application.

Performance

Profiling an application using LIBXSMM is well-supported using Intel VTune Amplifier. Performance analysis using Linux perf is supported as well. Both references give details on how to include profiler support in LIBXSMM and how to run the application. At build time, a variety of options exist for tuning the library for specific needs. LIBXSMM is setup for a broad range of use cases with sophisticated defaults for general use. Details about customizing LIBXSMM can be found in a separate document, which also includes build settings that impact the auto-dispatch behavior.

To find performance results of applications or performance reproducers, the repository provides an orphaned branch "results" which collects collateral material such as measured performance results along with explanatory figures. The results can be found at https://github.com/hfp/libxsmm/tree/results#libxsmm-results. Please note that comparing performance results depends on whether the operands of the matrix multiplication are streamed or not. For example, running a matrix multiplication code many time with all operands covered by the L1 cache may have an emphasis towards an implementation which perhaps performs worse for the real workload (if this real workload needs to stream some or all operands from the main memory). Most of the code samples are aimed to reproduce performance results, and it is encouraged to model the exact case or to look at real applications.

JIT Backend

More information about the JIT backend and the code generator can be found in a separate document, which also includes information about LIBXSMM's stand-alone generator-driver programs.

Applications

High Performance Computing (HPC)

[1] **https://cp2k.org/**: Open Source Molecular Dynamics with its DBCSR component processing batches of small matrix multiplications ("matrix stacks") out of a problem-specific distributed block-sparse matrix. Starting with CP2K 3.0, LIBXSMM can be used to substitute CP2K's 'libsmm' library. Prior to CP2K 3.0, only the Intel-branch of CP2K integrated LIBXSMM (see https://github.com/hfp/libxsmm/raw/master/documentation/cp2k.pdf).

[2] **https://github.com/SeisSol/SeisSol/**: SeisSol is one of the leading codes for earthquake scenarios, for simulating dynamic rupture processes. LIBXSMM provides highly optimized assembly kernels which form the computational back-bone of SeisSol (see https://github.com/TUM-I5/seissol_kernels/).

[3] **https://github.com/NekBox/NekBox**: NekBox is a highly scalable and portable spectral element code, which is inspired by the Nek5000 code. NekBox is specialized for box geometries, and intended for prototyping new methods as well as leveraging FORTRAN beyond the FORTRAN 77 standard. LIBXSMM can be used to substitute the MXM_STD code. Please also note LIBXSMM's NekBox reproducer.

[4] **https://github.com/Nek5000/Nek5000**: Nek5000 is the open-source, highly-scalable, always-portable spectral element code from https://nek5000.mcs.anl.gov/. The development branch of the Nek5000 code incorporates LIBXSMM.

[5] **http://pyfr.org/**: PyFR is an open-source Python based framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach. PyFR 1.6.0 optionally incorporates LIBXSMM as a matrix multiplication provider for the OpenMP backend. Please also note LIBXSMM's PyFR-related code sample.

Machine Learning (ML)

[6] **https://github.com/baidu-research/DeepBench**: The primary purpose of DeepBench is to benchmark operations that are important to deep learning on different hardware platforms. LIBXSMM's DNN primitives have been incorporated into DeepBench to demonstrate an increased performance of deep learning on Intel hardware. In addition, LIBXSMM's DNN sample folder contains scripts to run convolutions extracted from popular benchmarks in a stand-alone fashion.

[7] **https://www.tensorflow.org/**: TensorFlow™ is an open source software library for numerical computation using data flow graphs. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team for the purposes of conducting machine learning and deep neural networks research. LIBXSMM can be used to increase the performance of TensorFlow on Intel hardware.

[8] **https://github.com/IntelLabs/SkimCaffe**: SkimCaffe from Intel Labs is a Caffe branch for training of sparse CNNs, which provide 80-95% sparsity in convolutions and fully-connected layers. LIBXSMM's SPMDM domain (SParseMatrix-DenseMatrix multiplication) evolved from SkimCaffe, and since then LIBXSMM implements the sparse operations in SkimCaffe.

References

[1] **http://sc16.supercomputing.org/presentation/?id=pap364&sess=sess153**: LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation (paper). SC'16: The International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City (Utah).

[2] **http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post137.html**: LIBXSMM: A High Performance Library for Small Matrix Multiplications (poster and abstract). SC'15: The International Conference for High Performance Computing, Networking, Storage and Analysis, Austin (Texas).

[3] **https://software.intel.com/en-us/articles/intel-xeon-phi-delivers-competitive-performance-for-deep-learning-and-getting-better-fast**: Intel Xeon Phi Delivers Competitive Performance For Deep Learning - And Getting Better Fast. Article mentioning LIBXSMM's performance of convolution kernels with DeepBench. Intel Corporation, 2016.