LIBXSMM

LIBXSMM is a library for specialized dense and sparse matrix operations as well as for deep learning primitives such as small convolutions. The library is targeting Intel Architecture with Intel SSE, Intel AVX, Intel AVX2, Intel AVX-512 (with VNNI and Bfloat16), and Intel AMX (Advanced Matrix Extensions) supported by future Intel processor code-named Sapphire Rapids. Code generation is mainly based on Just-In-Time (JIT) code specialization for compiler-independent performance (matrix multiplications, matrix transpose/copy, sparse functionality, and deep learning). LIBXSMM is suitable for "build once and deploy everywhere", i.e., no special target flags are needed to exploit the available performance. Supported GEMM datatypes are: `FP64`, `FP32`, `bfloat16`, `int16`, and `int8`.

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

Where to go for documentation?

- **ReadtheDocs**: main and sample documentation with full text search.
- **PDF**: main documentation file, and separate sample documentation.
- **Articles**: magazine article incl. sample code (full list of Articles).

Getting Started: The following C++ code is focused on a specific functionality but may be considered as Hello LIBXSMM. Build the example with `cd /path/to/libxsmm; make STATIC=0` (shared library), save the code under `hello.cpp` (below) and compile with `g++ -I/path/to/libxsmm/include hello.cpp -L/path/to/libxsmm/lib -lxsmm -lblas -o hello` (GNU CCC), and finally execute with `LD_LIBRARY_PATH=/path/to/libxsmm/lib LIBXSMM_VERBOSE=2 ./hello`.

```
#include <libxsmm.h>
#include <vector>
int main(int argc, char* argv[]) {
  typedef double T;
  int batchsize = 1000, m = 13, n = 5, k = 7;
  std::vector<T> a(batchsize * m * k), b(batchsize * k * n), c(m * n, 0);
  /* C/C++ and Fortran interfaces are available */
  typedef libxsmm_mmfunction<T> kernel_type;
  /* generates and dispatches a matrix multiplication kernel (C++ functor) */
  kernel_type kernel(LIBXSMM_GEMM_FLAG_NONE, m, n, k, 1.0 /*alpha*/, 1.0 /*beta*/);
  assert(kernel);
  for (int i = 0; i < batchsize; ++i) { /* initialize input */
    for (int ki = 0; ki < k; ++ki) {
      for (int j = 0; j < m; ++j) a[i * j * ki] = static_cast<T>(1) / ((i + j + ki) % 25);
      for (int j = 0; j < n; ++j) b[i * j * ki] = static_cast<T>(7) / ((i + j + ki) % 75);
    }
  }
  /* kernel multiplies and accumulates matrices: C += Ai * Bi */
  for (int i = 0; i < batchsize; ++i) kernel(&a[i * m * k], &b[i * k * n], &c[0]);
}
```

Plain C code as well as Fortran code resemble the same example.

What is a small matrix multiplication? When characterizing the problem-size by using the M, N, and K parameters, a problem-size suitable for LIBXSMM falls approximately within $(M\ N\ K)^{1/3} <= 64$ (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 }*, and *TransA := '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 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.

Interfaces and Domains

Overview

Please have a look at https://github.com/libxsmm/libxsmm/tree/main/include for all published functions. Get started with the following list of available domains and documented functionality:

- MM: Matrix Multiplication
- TPP: Tensor Processing Primitives
- DNN: Deep Neural Networks
- AUX: Service Functions
- PERF: Performance
- BE: Backend

To initialize library internal resources, an explicit initialization routine helps to avoid lazy initialization overhead when calling LIBXSMM for the first time. The library deallocates internal resources at program exit, but also provides a companion of the afore mentioned initialization (finalize).

```
/** Initialize the library; pay for setup cost at a specific point. */
void libxsmm_init(void);
/** De-initialize the library and free internal memory (optional). */
void libxsmm_finalize(void);
```

Matrix Multiplication

This domain (MM) supports Small Matrix Multiplications (SMM), batches of multiple multiplications as well as the industry-standard interface for GEneral Matrix Matrix multiplication (GEMM).

The Matrix Multiplication domain (MM) contains routines for:

- Small, tiled, and parallelized matrix multiplications
- Manual code dispatch (customized matrix batches)
- Batched multiplication (explicit interface)
- Call wrapper (static and dynamic linkage)

Deep Learning

The Deep Learning domain is detailed by the following sample codes. Here we demonstrate how common operators in deep learning applications (GEMM with activation function fusion, Convolutions with activation function fusion, various norming and pooling operators, etc.) can be implemented using the Tensor Processing Primitive provided by LIBXSMM. Example drivers for performance evaluation are provided as part of LIBXSMM_DNN.

Service Functions

For convenient operation of the library and to ease integration, some service routines are available. These routines may not belong to the core functionality of LIBXSMM (SMM or DNN domain), but users are encouraged to use this domain (AUX). There are two categories: (1) routines which are available for C and FORTRAN, and (2) routines that are only available per 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)
- Dispatching user-data and multiple kernels
- Loading and storing data (I/O)
- Allocating memory

Backend

More information about the JIT-backend and the code generator can be found in a separate document. The encoder sample collection can help to get started writing a kernel using LIBXSMM. Please note, LIBXSMM's stand-alone generator-driver is considered legacy (deprecated).

Build Instructions

Overview

The main interface file is *generated*, and it is therefore **not** stored in the code repository. To inspect the interface for C/C++ and FORTRAN, one can take a look at the template files used to generate the actual interface. There are two general ways to build and use LIBXSMM:

- Classic Library (ABI) and Link Instructions (C/C++ and FORTRAN)
- Header-Only (C and C++)

Note: LIBXSMM is available as prebuilt package for Fedora/RedHat/CentOS, Debian/Ubuntu, FreeBSD, and others. Further, LIBXSMM can be installed with the Spack Package Manager or per EasyBuild+EasyConfig.

Classic Library (ABI)

There are two ways to rely on prebuilt code for a given project: (1) using LIBXSMM's Makefile based build system, (2) or using another build system and writing own rules for building LIBXSMM. The Makefile based 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 (the second method may require make's `-e` flag).

In contrast to header-only which does not require configuration by default, 3rd-party build systems can compile and link LIBXSMM's sources but still avoid configuring the library (per `libxsmm_config.py`). The prerequisite to omit configuration is to opt-in by defining LIBXSMM_DEFAULT_CONFIG (`-D`). The zero-config feature is not available for LIBXSMM's Fortran interface.

Note: By default, C/C++ and FORTRAN compilers are 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 `libxsmm.f` library (the interface `libxsmm.f` is still generated).

The build system considers a set of given key-value pairs as a single unique build and triggers a rebuild for a distinct set of flags. For more advanced builds or additional background, please consult the section about Customization. To generate the interface of the library inside of the `include` directory and to build the static library (by default, STATIC=1 is activated). Run any (or both) of the following command(s):

```
make STATIC=0
make
```

On CRAY systems, the CRAY Compiling Environment (CCE) should be used regardless of utilizing the CRAY compiler, the Intel Compiler, or the GNU Compiler Collection (GCC). The CCE is eventually suppressing to build shared libraries (STATIC=0). In any case, (1) switch to the desired compiler (module load/switch), and (2) rely on:

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

A variety of build environments is out-of-the-box compatible, see https://github.com/libxsmm/libxsmm/wiki/Compatibility. If the build process is not successful, it may help to avoid advanced GCC flags. This is useful with a tool chain, which pretends to be GCC-compatible (and is treated as such) but fails to consume the afore mentioned flags:

```
make COMPATIBLE=1
```

In case of outdated Binutils, compilation can fail to assemble code when building the library (this has nothing to do with JIT-generated code and it does not affect how JIT-code is targeting the system). LIBXSMM implements some functionality using compiler-intrinsics and multiple code-paths which are scheduled according to CPUID. In contrast to `INTRINSICS=2` (default), `INTRINSICS=1` enables a fully static code path according to the desired target. If no target is given (e.g., `AVX=3`, or `AVX=2`), instruction set extensions cannot be leveraged for such code-paths. Try to fix failing compilation by building the latest GNU Binutils (and `export PATH=/path/to/binutils/bin:${PATH}`). Binutils are versioned independently of GNU GCC and other compilers. If one cannot update Binutils, work around with a CPUID-value as tabulated in libxsmm_cpuid.h: start at the upper end (less than 1999) and decrement until compilation passes (make INTRINSICS=*CPUID*, e.g., `make INTRINSICS=1021`). As a last resort, rely on a fully static code path:

```
make INTRINSICS=1
```

To test and validate a build, please consult https://github.com/libxsmm/libxsmm/wiki/Validation. To run some basic sanity checks, remember that each set of given key-value pairs represents a different build (and test):

```
make STATIC=0 tests
```

To remove intermediate files, or to remove all generated files and folders (including the interface and the library archives), run one of the make-targets below. An additional distclean-target recursively cleans the entire tree (after version 1.9).

```
make clean
make realclean
```

FORTRAN code can make use of LIBXSMM:

- By using the module and linking with `libxsmmf`, `libxsmm`, and `libxsmmext`,
- By including `libxsmm.f` and linking with `libxsmm`, and `libxsmmext`, or
- By (implicitly) calling a SUBROUTINE and linking with `libxsmm`, and `libxsmmext`.

Note: `libxsmmf` requires `libxsmmext` (starting with LIBXSMM 2.0), and thereby requires to link with the OpenMP runtime as well.

Using the Fortran module (or including the interface), requires at least a Fortran 2003 compiler (F2K3). FORTRAN 77 compatibility is only implicitly available (no interface), and the available subset of routines is documented in `libxsmm.f` and marked with comments (part of the implementation).

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 it may also ensure to only rely on the public interface of LIBXSMM. In contrast, the header-only form not only exposes the internal implementation of LIBXSMM but can also increase the turnaround time during development of an application (due to longer compilation times). The header file is intentionally named "libxsmm_**source**.h" since this header file relies on the src directory (with the implications as noted earlier).

The header-only form depends on `libxsmm_source.h` which is *generated* according to the content of the source folder (`src`). LIBXSMM 1.16 (and later) provides header-only support without invoking a make-target (zero configuration) for any given checkout of LIBXSMM. To use configured header-only (non-default), LIBXSMM_CONFIGURED must be defined (`-D`). Previously, it was necessary to invoke `make header-only` (v1.6.2 or later), `make cheader` (prior to v1.6.2), or any target building the library (`make`). The zero-config feature allows 3rd-party build systems an easier integration of LIBXSMM, which also holds true if the system builds LIBXSMM from source (see classic ABI). Fortran code may include `libxsmm.f` but still requires that interface to be generated.

Note: building an application applies the same build settings to LIBXSMM! For instance, to omit debug code inside of LIBXSMM `NDEBUG` must be defined (`-DNDEBUG`).

Rules for building LIBXSMM

LIBXSMM can be used as header-only library, i.e., no source code must be (pre-)built. However, it can be desirable to build LIBXSMM as an intermediate library using a custom setup or build system. The latter can still implement custom build rules to configure LIBXSMM's interface before building the code. More likely, building LIBXSMM from source in a custom fashion can still be omitting to configure the interface and rely on "(zero-config)[#zero-config-abi]", i.e., defining LIBXSMM_DEFAULT_CONFIG (`-DLIBXSMM_DEFAULT_CONFIG`). For example, a CMake module for LIBXSMM can look like:

```
include(FetchContent)
FetchContent_Declare(
  xsmm
  URL https://github.com/chelini/libxsmm/archive/<your-preferred-revision>.tar.gz
  URL_HASH SHA256=<sha256sum-corresponding-to-above-revision>
)
FetchContent_GetProperties(xsmm)
if(NOT xsmm_POPULATED)
  FetchContent_Populate(xsmm)
endif()

set(LIBXSMMROOT ${xsmm_SOURCE_DIR})
file(GLOB _GLOB_XSMM_SRCS LIST_DIRECTORIES false CONFIGURE_DEPENDS ${LIBXSMMROOT}/src/*.c)
list(REMOVE_ITEM _GLOB_XSMM_SRCS ${LIBXSMMROOT}/src/libxsmm_generator_gemm_driver.c)
set(XSMM_INCLUDE_DIRS ${LIBXSMMROOT}/include)

add_library(xsmm STATIC ${_GLOB_XSMM_SRCS})
target_include_directories(xsmm PUBLIC ${XSMM_INCLUDE_DIRS})
target_compile_definitions(xsmm PUBLIC
  LIBXSMM_DEFAULT_CONFIG
)
target_compile_definitions(xsmm PRIVATE
  __BLAS=0
)
```

Above, LIBXSMM_DEFAULT_CONFIG is propagated to dependent code (`PUBLIC`) and further, LIBXSMM is configured to not require a LAPACK/BLAS library/fallback (`-D__BLAS=0`).

Link Instructions

Using the classic ABI (including Fortran code), requires linking LIBXSMM against the application. 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. Enabling OpenMP for LIBXSMM's main library is supported as well (OMP=1), and mostly affects the synchronization primitives used inside of the library. All of the "omp" functionality (function postfix) is served by the `libxsmmext` library, which is automatically built with OpenMP enabled. When using this "omp" functionality, `libxsmmext` needs to be present at the link line.

Library | Purpose :| libxsmm | Thread-safe core functions (same routine can be called concurrently). Contains routines that can take a thread-ID and the number of library-external threads. libxsmmf | Necessary when using the Fortran MODule but not when including `libxsmm.f` or relying on implicit interfaces (Fortran 77). libxsmmext | Provides library-internal OpenMP-threaded functions carrying the `omp` postfix when compared to function name names of the core library. libxsmmnoblas | Supplies faked symbols for `dgemm` (and others) and thereby removes the need to link against a LAPACK/BLAS library.

To ease linking with LIBXSMM, `pkg-config` can be used. For example:

```
export PKG_CONFIG_PATH=/path/to/libxsmm/lib
pkg-config libxsmm --libs
```

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. To use LIBXSMM without GEMM-related functionality, any BLAS-dependency can be removed in two ways: (1) building a special library with `make BLAS=0`, or (2) linking the application against the `libxsmmnoblas` library. If an application however uses BLAS already, the Call Wrapper can be used to intercept existing BLAS calls (and to rely on LIBXSMM instead).

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!

Installation

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
```

Installation into a specific location looks like (`PREFIX` or `DESTDIR`):

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

Both `PREFIX` and `DESTDIR` are equivalent and can be relative or absolute paths. An installation can be repeated for different locations without triggering a rebuild. The prefix directory *inside* of each of the package configuration files is set to where LIBXSMM is built (staging folder) unless `PREFIX` or `DESTDIR` is specified. The effect of `PREFIX` (or `DESTDIR`) with respect to the pkg-config files is independent of whether the install-target is invoked or not (make).

Further, 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
```

Runtime Control

Handling Errors

The library handles errors with mechanisms available to the C programming language (no exceptions). The backend uses result codes passed by an argument rather than an actual return value. Such an argument is often a descriptor (struct) guiding and covering the state of the code generation. The frontend however may not hand-out any error state, which can be a big relief on the call-side. Instead, the frontend implements a verbose mode to inform about unexpected input or an error captured from the backend. Guiding principles of LIBXSMM are muted operation by default (non-verbose) and no unexpected exit from execution.

Verbose Mode

The verbose mode (level of verbosity) 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 ($MNK^{1/3} <= 13$), medium-sized kernels ($13 < MNK^{1/3} <= 23$), and larger kernels ($23 < MNK^{1/3} <= 64$; 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 ($O(n) = 10^5$) of the code registry is exhausted, no more kernels can be registered although further attempts are not prevented. Registering many kernels ($O(n) = 10^3$) 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. 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)
```

If the call-wrapper is used, an additional runtime statistic becomes available (see Call Wrapper).

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 then 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, which uses LIBXSMM's JIT-code is well-supported. The library supports Intel VTune Amplifier and Linux perf. Details are given on how to include profiler support, and how to run the application.

- Profiling using Intel VTune Amplifier
- Profiling using Linux perf

At build time, a variety of options exist to customize LIBXSMM. The library is setup for a broad range of use cases, which include sophisticated defaults for typical use.

- Customizing performance
- Tuning auto-dispatch

To find performance results of applications or performance reproducers, the repository provides an orphaned branch called "results" which collects collateral material such as measured performance results along with explanatory figures. The results can be found at https://github.com/libxsmm/libxsmm/tree/results#libxsmm-results, or the results can be cloned as shown below.

```
git clone --branch results \
  https://github.com/libxsmm/libxsmm.git \
  libxsmm-results
```

Please note that comparing performance results depends on whether the operands of the matrix multiplication are streamed or not. For example, multiplying with all matrices 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 matrices 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.

Applications

High Performance Computing (HPC)

[1] https://cp2k.org/: Open Source Molecular Dynamics and the DBCSR library, which processes batches of small matrix multiplications. The batches originate from a distributed block-sparse matrix with problem-specific small matrices. Starting with CP2K 3.0, LIBXSMM can substitute CP2K's `libsmm` library.

[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 to prototype new methods as well as to leverage 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 by 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.

[6] http://dial3343.org/about/: The Extreme-scale Discontinuous Galerkin Environment (EDGE) is a solver for hyperbolic partial differential equations with emphasis on seismic simulations. The EDGE source code optionally relies on LIBXSMM, but for high performance LIBXSMM's kernels are highly recommended.

[7] https://sxs-collaboration.github.io/spectre/: SpECTRE is an open-source code for multi-scale, multi-physics problems in astrophysics and gravitational physics which runs at Petascale and is designed for Exascale computers. In the future, SpECTRE may be applied to problems across discipline boundaries in fluid dynamics, geoscience, plasma physics, nuclear physics, and engineering.

[8] https://ceed.exascaleproject.org/ceed-code/: The Center for Efficient Exascale Discretizations (CEED) is building on the efforts of the Nek5000, MFEM, MAGMA, OCCA and PETSc projects to develop application program interfaces (APIs), both at high-level and at low-level to enable applications to take advantage of high-order methods. The CEED low-level API, libCEED uses LIBXSMM as a backend for high performance on CPUs.

[9] https://github.com/romeric/Fastor: Fastor is a lightweight high performance tensor algebra framework for modern C++ and can optionally use LIBXSMM as JIT-backend.

Machine Learning (ML)

[10] https://github.com/plaidml/plaidml: PlaidML is an open source tensor compiler aiming for performance portability across a wide range of CPUs, GPUs and other accelerators. Combined with Intel's nGraph compiler, PlaidML is targeting popular deep learning frameworks such as PyTorch, Keras (TensorFlow), and OpenVino. PlaidML/v1 (development branch) adopted MLIR, an extensible compiler infrastructure gaining industry-wide adoption. PlaidML/v1 started using LIBXSMM as backend for targeting CPUs.

[11] https://github.com/intel/intel-extension-for-pytorch: Intel Extension for PyTorch aims for a smooth user experience of PyTorch on CPUs by the means of good performance. The extension pack started to rely on LIBXSMM for achieving high performance on CPUs.

[12] 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 was once used to increase the performance of TensorFlow on Intel hardware.

[13] 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.

[14] 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.

Automated Driving (AD)

[15] https://software.seek.intel.com/accelerating-eigen-math-library: Accelerating The Eigen Math Library for Automated Driving Workloads: The Need for Speed in Kalman Filtering. An article in Issue 31 of The Parallel Universe magazine (pdf).

References

[1] https://sc19.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost244.html: High-Performance Deep Learning via a Single Building Block (poster and abstract), SC'19: The International Conference for High Performance Computing, Networking, Storage, and Analysis, Denver (Colorado).

[2] https://dl.acm.org/doi/10.1109/SC.2018.00069: Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures (paper). SC'18: The International Conference for High Performance Computing, Networking, Storage, and Analysis, Dallas (Texas).

[3] https://pasc17.pasc-conference.org/fileadmin/user_upload/pasc17/program/post116s2.pdf: DBCSR: A Sparse Matrix Multiplication Library for Electronic Structure Codes (poster), PASC'17: The PASC17 Conference, Lugano (Switzerland).

[4] https://sc17.supercomputing.org/SC17%20Archive/tech_poster/tech_poster_pages/post190.html: Understanding the Performance of Small Convolution Operations for CNN on Intel Architecture (poster and abstract), SC'17: The International Conference for High Performance Computing, Networking, Storage, and Analysis, Denver (Colorado).

[5] https://www.computer.org/csdl/proceedings-article/sc/2016/8815a981/12OmNCeaQ1D: LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. SC'16: The International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City (Utah).

[6] 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).

[7] Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning & HPC Workloads SC'21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St Louis.

Articles

[1] https://www.nextplatform.com/2019/10/09/cloudy-supercomputers-join-the-hpc-petascale-club/: Cloudy Supercomputers Join the HPC Petascale Club. An article written by Rob Farber, 2019. The article covers LIBXSMM in a separate section.

[2] https://www.nextplatform.com/2019/06/26/counting-the-cost-of-scaling-hpc-applications/: Counting The Cost Of Scaling HPC Applications. An article written by Timothy Prickett Morgan, 2019. This article is about CP2K Open Source Molecular Dynamics and not about LIBXSMM. However, LIBXSMM was key for application performance.

[3] https://www.nextplatform.com/2019/06/26/counting-the-cost-of-scaling-hpc-applications/: Azure Benchmarks HC-series Across Twenty-thousand Cores for HPC. An article written by John Russell, 2019. This article is about CP2K Open Source Molecular Dynamics and not about LIBXSMM. However, LIBXSMM was key for application performance.

[4] https://software.intel.com/sites/default/files/parallel-universe-issue-34.pdf: LIBXSMM: An Open Source-Based Inspiration for Hardware and Software Development at Intel (pdf). An article written by Hans Pabst, Greg Henry, and Alexander Heinecke, 2018.

[5] https://medium.com/@rmfarber/libxsmm-brings-deep-learning-lessons-learned-to-many-hpc-applications-9143c6c93125: LIBXSMM Brings Deep-learning "Lessons Learned" to Many HPC Applications. An article written by Rob Farber, 2018.

[6] https://www.rdworldonline.com/largest-supercomputer-simulation-of-sumatra-andaman-earthquake/: Largest Supercomputer Simulation of Sumatra-Andaman Earthquake. An article written by Linda Barney, 2018.

LIBXSMM Domains

Matrix Multiplication

Overview

To perform the dense matrix-matrix multiplication C_{mxn} = alpha · A_{mxk} · B_{kxn} + beta · C_{mxn}, the full-blown GEMM interface can be treated with "default arguments" (which is deviating from the BLAS standard, however without compromising the binary compatibility). Default arguments are derived from compile-time constants (configurable) for historic reasons (LIBXSMM's "pre-JIT era").

```
libxsmm_?gemm(NULL/*transa*/, NULL/*transb*/,
  &m/*required*/, &n/*required*/, &k/*required*/,
  NULL/*alpha*/, a/*required*/, NULL/*lda*/,
                 b/*required*/, NULL/*ldb*/,
   NULL/*beta*/, c/*required*/, NULL/*ldc*/);
```

For the C interface (with type prefix `s` or `d`), all arguments including m, n, and k are passed by pointer. This is needed for binary compatibility with the original GEMM/BLAS interface.

```
libxsmm_gemm(NULL/*transa*/, NULL/*transb*/,
  m/*required*/, n/*required*/, k/*required*/,
  NULL/*alpha*/, a/*required*/, NULL/*lda*/,
                 b/*required*/, NULL/*ldb*/,
   NULL/*beta*/, c/*required*/, NULL/*ldc*/);
```

The C++ interface is also supplying overloaded versions where m, n, and k can be passed by-value (making it clearer that m, n, and k are non-optional arguments).

```
! Dense matrix multiplication (single/double-precision).
CALL libxsmm_?gemm(m=m, n=n, k=k, a=a, b=b, c=c)
! Dense matrix multiplication (generic interface).
CALL libxsmm_gemm(m=m, n=n, k=k, a=a, b=b, c=c)
```

The FORTRAN interface supports optional arguments (without affecting the binary compatibility with the original BLAS interface) by allowing to omit arguments where the C/C++ interface allows for NULL to be passed.

```
/** Dense matrix multiplication (single/double-precision). */
libxsmm_blas_?gemm(NULL/*transa*/, NULL/*transb*/,
  &m/*required*/, &n/*required*/, &k/*required*/,
  NULL/*alpha*/, a/*required*/, NULL/*lda*/,
                 b/*required*/, NULL/*ldb*/,
   NULL/*beta*/, c/*required*/, NULL/*ldc*/);
```

For convenience, a BLAS-based dense matrix multiplication (`libxsmm_blas_gemm`) is provided for all supported languages. This only re-exposes the underlying GEMM/BLAS implementation, but the interface accepts optional arguments (or NULL pointers in C) where the regular GEMM expects a value. To remove any BLAS-dependency, please follow the Link Instructions. A BLAS-based GEMM can be useful for validation/benchmark purposes, and more important as a fallback when building an application-specific dispatch mechanism.

Manual Code Dispatch

Successively calling a kernel (i.e., multiple times) allows for amortizing the cost of the code dispatch. Moreover, to customize the dispatch mechanism, one can rely on the following interface.

```
/** Call dispatched (*function_ptr)(a, b, c [, pa, pb, pc]). */
libxsmm_[s|d]mmfunction libxsmm_[type-prefix]mmdispatch(
  libxsmm_blasint m, libxsmm_blasint n, libxsmm_blasint k,
  /** NULL: tight fit (m) */ const libxsmm_blasint* lda,
  /** NULL: tight fit (k) */ const libxsmm_blasint* ldb,
  /** NULL: tight fit (m) */ const libxsmm_blasint* ldc,
  /** NULL: LIBXSMM_ALPHA */ const type* alpha,
  /** NULL: LIBXSMM_BETA  */ const type* beta,
  /** NULL: LIBXSMM_FLAGS */ const int* flags,
  /** NULL: LIBXSMM_PREFETCH_NONE (not LIBXSMM_PREFETCH!) */
  const int* prefetch);
```

Overloaded function signatures are provided and allow to omit arguments (C++ and FORTRAN), which are then derived from the configurable defaults. In C++, `libxsmm_mmfunction<type>` can be used to instantiate a functor rather than making a distinction between numeric types per type-prefix. For lower precision GEMMs, `libxsmm_mmfunction<itype,otype=itype>` optionally takes a second type (output type).

```
/* generates or dispatches the code specialization */
libxsmm_mmfunction<T> xmm(m, n, k);
if (xmm) { /* JIT'ted code */
  /* can be parallelized per, e.g., OpenMP */
  for (int i = 0; i < n; ++i) {
    xmm(a+i*asize, b+i*bsize, c+i*csize);
  }
}
```

Similarly in FORTRAN (see samples/utilities/smmbench/smm.f), a generic interface (`libxsmm_mmdispatch`) can be used to dispatch a `LIBXSMM_?MMFUNCTION`. The handle encapsulated by such a `LIBXSMM_?MMFUNCTION` can be called per `libxsmm_call`. Beside of dispatching code, one can also call statically generated kernels (e.g., `libxsmm_dmm_4_4_4`) by using the prototype functions included with the FORTRAN and C/C++ interface. Prototypes are present whenever static code was requested at compile-time of the library (e.g. per `make MNK="1 2 3 4 5"`).

```
TYPE(LIBXSMM_DMMFUNCTION) :: xmm
CALL libxsmm_dispatch(xmm, m, n, k)
IF (libxsmm_available(xmm)) THEN
  DO i = LBOUND(c, 3), UBOUND(c, 3) ! consider OpenMP
    CALL libxsmm_dmmcall(xmm, a(:,:,i), b(:,:,i), c(:,:,i))
  END DO
END IF
```

Batched Multiplication

In case of batched SMMs, it can be beneficial to supply "next locations" such that the upcoming operands are prefetched ahead of time. Such a location would be the address of the next matrix to be multiplied (and not any of the floating-point elements within the "current" matrix-operand). The "prefetch strategy" is requested at dispatch-time of a kernel. A strategy other than `LIBXSMM_PREFETCH_NONE` turns the signature of a JIT'ted kernel into a function with six

arguments (`a,b,c, pa,pb,pc` instead of `a,b,c`). To defer the decision about the strategy to a CPUID-based mechanism, one can choose `LIBXSMM_PREFETCH_AUTO`.

```
int prefetch = LIBXSMM_PREFETCH_AUTO;
int flags = 0; /* LIBXSMM_FLAGS */
libxsmm_dmmfunction xmm = NULL;
double alpha = 1, beta = 0;
xmm = libxsmm_dmmdispatch(23/*m*/, 23/*n*/, 23/*k*/,
  NULL/*lda*/, NULL/*ldb*/, NULL/*ldc*/,
  &alpha, &beta, &flags, &prefetch);
```

Above, pointer-arguments of `libxsmm_dmmdispatch` can be NULL (or OPTIONAL in FORTRAN): for LDx this means a "tight" leading dimension, alpha, beta, and flags are given by a default value (which is selected at compile-time), and for the prefetch strategy a NULL-argument refers to "no prefetch" (which is equivalent to an explicit `LIBXSMM_PREFETCH_NONE`). By design, the prefetch strategy can be changed at runtime (as soon as valid next-locations are used) without changing the call-site (kernel-signature with six arguments).

```
if (0 < n) { /* check that n is at least 1 */
# pragma parallel omp private(i)
  for (i = 0; i < (n - 1); ++i) {
    const double *const ai = a + i * asize;
    const double *const bi = b + i * bsize;
    double *const ci = c + i * csize;
    xmm(ai, bi, ci, ai + asize, bi + bsize, ci + csize);
  }
  xmm(a + (n - 1) * asize, b + (n - 1) * bsize, c + (n - 1) * csize,
  /* pseudo prefetch for last element of batch (avoids page fault) */
      a + (n - 1) * asize, b + (n - 1) * bsize, c + (n - 1) * csize);
}
```

To process a batch of matrix multiplications and to prefetch the operands of the next multiplication ahead of time, the code presented in the Overview section may be modified as shown above. The last multiplication is peeled from the main batch to avoid prefetching out-of-bounds (OOB). Prefetching from an invalid address does not trap an exception, but an (unnecessary) page fault can be avoided.

```
/** Batched matrix multiplications (explicit data representation). */
int libxsmm_gemm_batch_task(libxsmm_datatype iprec, libxsmm_datatype oprec,
  const char* transa, const char* transb,
  libxsmm_blasint m, libxsmm_blasint n, libxsmm_blasint k,
  const void* alpha, const void* a, const libxsmm_blasint* lda,
                     const void* b, const libxsmm_blasint* ldb,
   const void* beta,       void* c, const libxsmm_blasint* ldc,
  libxsmm_blasint index_base, libxsmm_blasint index_stride,
  const libxsmm_blasint stride_a[],
  const libxsmm_blasint stride_b[],
  const libxsmm_blasint stride_c[],
  libxsmm_blasint batchsize,
  int tid, int ntasks);
```

To further simplify the multiplication of matrices in a batch, LIBXSMM's batch interface can help to extract the necessary input from a variety of existing structures (integer indexes, array of pointers both with Byte sized strides). An expert interface (see above) can employ a user-defined threading runtime (`tid` and `ntasks`). In case of OpenMP, `libxsmm_gemm_batch_omp` is ready-to-use and hosted by the extension library (libxsmmext). Of course, `libxsmm_gemm_batch_omp` does not take `tid` and `ntasks` since both arguments are given by OpenMP. Similarly, a sequential version (shown below) is available per `libxsmm_gemm_batch` (libxsmm).

Please note that an explicit data representation should exist and reused rather than created only to call the explicit batch-interface. Creating such a data structure only for this matter can introduce an overhead which is hard to amortize (speedup). If no explicit data structure exists, a "chain" of multiplications can be often algorithmically described (see self-hosted batch loop).

```
void libxsmm_gemm_batch(libxsmm_datatype iprec, libxsmm_datatype oprec,
  const char* transa, const char* transb,
  libxsmm_blasint m, libxsmm_blasint n, libxsmm_blasint k,
  const void* alpha, const void* a, const libxsmm_blasint* lda,
                     const void* b, const libxsmm_blasint* ldb,
   const void* beta,       void* c, const libxsmm_blasint* ldc,
  libxsmm_blasint index_base, libxsmm_blasint index_stride,
  const libxsmm_blasint stride_a[],
  const libxsmm_blasint stride_b[],
```

```
  const libxsmm_blasint stride_c[],
  libxsmm_blasint batchsize);
```

In recent BLAS library implementations, `dgemm_batch` and `sgemm_batch` have been introduced. This BLAS(-like) interface allows for groups of homogeneous batches, which is like an additional loop around the interface as introduced above. On the other hand, the BLAS(-like) interface only supports arrays of pointers for the matrices. In contrast, above interface supports arrays of pointers as well as arrays of indexes plus a flexible way to extract data from arrays of structures (AoS). LIBXSMM also supports this (new) BLAS(-like) interface with `libxsmm_?gemm_batch` and `libxsmm_?gemm_batch_omp` (the latter of which relies on LIBXSMM/ext). Further, existing calls to `dgemm_batch` and `sgemm_batch` can be intercepted and replaced with LIBXSMM's call wrapper. The signatures of `libxsmm_dgemm_batch` and `libxsmm_sgemm_batch` are equal except for the element type (`double` and `float` respectively).

```
void libxsmm_dgemm_batch(const char transa_array[], const char transb_array[],
  const libxsmm_blasint m_array[], const libxsmm_blasint n_array[], const libxsmm_blasint k_array[],
  const double alpha_array[], const double* a_array[], const libxsmm_blasint lda_array[],
                              const double* b_array[], const libxsmm_blasint ldb_array[],
  const double  beta_array[],       double* c_array[], const libxsmm_blasint ldc_array[],
  const libxsmm_blasint* group_count, const libxsmm_blasint group_size[]);
```

Note: the multi-threaded implementation (`ntasks > 1` or "omp" form of the functions) avoids data races if indexes or pointers for the destination (C-)matrix are duplicated. This synchronization occurs automatically (`beta != 0`), but can be avoided by passing a negative `batchsize`, `group_size` and/or a negative `group_count`.

User-Data Dispatch

It can be desired to dispatch user-defined data, i.e., to query a value based on a key. This functionality can be used to, e.g., dispatch multiple kernels in one step if a code location relies on multiple kernels. This way, one can pay the cost of dispatch one time per task rather than according to the number of JIT-kernels used by this task. This functionality is detailed in the section about Service Functions.

Call Wrapper

Overview Since the library is binary compatible with existing GEMM calls (BLAS), such calls can be replaced at link-time or intercepted at runtime of an application such that LIBXSMM is used instead of the original BLAS library. There are two cases to consider: (1) static linkage, and (2) dynamic linkage of the application against the original BLAS library. When calls are intercepted, one can select a sequential (default) or an OpenMP-parallelized implementation (`make WRAP=2`).

Note: Intercepting GEMM calls is low effort but implies overhead, which can be relatively high for small-sized problems. LIBXSMM's native programming interface has lower overhead and can amortize overhead when using the same multiplication kernel in a consecutive fashion (and employ sophisticated data prefetch on top).

Static Linkage An application which is linked statically against BLAS requires to wrap the `sgemm_` and the `dgemm_` symbol (an alternative is to wrap only `dgemm_`). To relink the application (without editing the build system) can often be accomplished by copying and pasting the linker command as it appeared in the console output of the build system, and then re-invoking a modified link step (please also consider `-Wl,--export-dynamic`).

```
gcc [...] -Wl,--wrap=dgemm_,--wrap=sgemm_ \
          /path/to/libxsmmext.a /path/to/libxsmm.a \
          /path/to/your_regular_blas.a
```

In addition, existing BLAS(-like) batch-calls can be intercepted as well:

```
gcc [...] -Wl,--wrap=dgemm_batch_,--wrap=sgemm_batch_ \
          -Wl,--wrap=dgemm_batch,--wrap=sgemm_batch \
          -Wl,--wrap=dgemm_,--wrap=sgemm_ \
          /path/to/libxsmmext.a /path/to/libxsmm.a \
          /path/to/your_regular_blas.a
```

Above, GEMM and GEMM_BATCH are intercepted both, however this can be chosen independently. For GEMM_BATCH the Fortran and C-form of the symbol may be intercepted both (regular GEMM can always be intercepted per `?gemm_` even when `?gemm` is used in C-code).

Note: The static link-time wrapper technique may only work with a GCC tool chain (GNU Binutils: `ld`, or `ld` via compiler-driver), and it has been tested with GNU GCC, Intel Compiler, and Clang. However, this does not work under Microsoft Windows (even when using the GNU tool chain or Cygwin).

Dynamic Linkage An application that is dynamically linked against BLAS allows to intercept the GEMM calls at startup time (runtime) of the unmodified executable by using the LD_PRELOAD mechanism. The shared library of LIBXSMMext (`make STATIC=0`) can be used to intercept GEMM calls:

```
LD_LIBRARY_PATH=/path/to/libxsmm/lib:${LD_LIBRARY_PATH} \
LD_PRELOAD=libxsmmext.so \
   ./myapplication
```

Service Functions

Target Architecture

This functionality is available for the C and Fortran interface. There are ID based (same for C and Fortran) and string based functions to query the code path (as determined by the CPUID), or to set the code path regardless of the presented CPUID features. The latter may degrade performance if a lower set of instruction set extensions is requested, which can be still useful for studying the performance impact of different instruction set extensions.
Note: There is no additional check performed if an unsupported instruction set extension is requested, and incompatible JIT-generated code may be executed (unknown instruction signaled).

```
int libxsmm_get_target_archid(void);
void libxsmm_set_target_archid(int id);

const char* libxsmm_get_target_arch(void);
void libxsmm_set_target_arch(const char* arch);
```

Available code paths (IDs and corresponding strings):

- LIBXSMM_TARGET_ARCH_GENERIC: "**generic**", "none", "0"
- LIBXSMM_X86_GENERIC: "**x86**", "x64", "sse2"
- LIBXSMM_X86_SSE3: "**sse3**"
- LIBXSMM_X86_SSE42: "**wsm**", "nhm", "sse4", "sse4_2", "sse4.2"
- LIBXSMM_X86_AVX: "**snb**", "avx"
- LIBXSMM_X86_AVX2: "**hsw**", "avx2"
- LIBXSMM_X86_AVX512_MIC: "**knl**", "mic"
- LIBXSMM_X86_AVX512_KNM: "**knm**"
- LIBXSMM_X86_AVX512_CORE: "**skx**", "skl", "avx3", "avx512"
- LIBXSMM_X86_AVX512_CLX: "**clx**"
- LIBXSMM_X86_AVX512_CPX: "**cpx**"
- LIBXSMM_X86_AVX512_SPR: "**spr**"

The **bold** names are returned by `libxsmm_get_target_arch` whereas `libxsmm_set_target_arch` accepts all of the above strings (similar to the environment variable LIBXSMM_TARGET).

Verbosity Level

The verbose mode (level of verbosity) can be controlled using the C or Fortran API, and there is an environment variable which corresponds to `libxsmm_set_verbosity` (LIBXSMM_VERBOSE).

```
int libxsmm_get_verbosity(void);
void libxsmm_set_verbosity(int level);
```

Timer Facility

Due to the performance oriented nature of LIBXSMM, timer-related functionality is available for the C and Fortran interface (libxsmm_timer.h and libxsmm.f). The timer is used in many of the code samples to measure the duration of executing a region of the code. The timer is based on a monotonic clock tick, which uses a platform-specific resolution. The counter may rely on the time stamp counter instruction (RDTSC), which is not necessarily counting CPU cycles (reasons are out of scope in this context). However, `libxsmm_timer_ncycles` delivers raw clock ticks (RDTSC).

```
typedef unsigned long long libxsmm_timer_tickint;
libxsmm_timer_tickint libxsmm_timer_tick(void);
double libxsmm_timer_duration(
  libxsmm_timer_tickint tick0,
  libxsmm_timer_tickint tick1);
libxsmm_timer_tickint libxsmm_timer_ncycles(
  libxsmm_timer_tickint tick0,
  libxsmm_timer_tickint tick1);
```

User-Data Dispatch

To register a user-defined key-value pair with LIBXSMM's fast key-value store, the key must be binary reproducible. Structured key-data (`struct` or `class` type which can be padded in a compiler-specific fashion) must be completely cleared, i.e., all gaps may be zero-filled before initializing data members (`memset(&mykey, 0, sizeof(mykey))`). This is because some compilers can leave padded data uninitialized, which breaks binary reproducible keys, hence the flow is: claring heterogeneous keys (struct), initialization (members), and registration. The size of the key is arbitrary but limited to LIBXSMM_DESCRIPTOR_MAXSIZE (96 Byte), and the size of the value can be of an arbitrary size. The given value is copied and may be initialized at registration-time or when dispatched. Registered data is released at program termination but can be manually unregistered and released (`libxsmm_xrelease`), e.g., to register a larger value for an existing key.

```
void* libxsmm_xregister(const void* key, size_t key_size, size_t value_size, const void* value_init);
void* libxsmm_xdispatch(const void* key, size_t key_size);
```

The Fortran interface is designed to follow the same flow as the C language: (1) `libxsmm_xdispatch` is used to query the value, and (2) if the value is a NULL-pointer, it is registered per `libxsmm_xregister`. Similar to C (`memset`), structured key-data must be zero-filled (`libxsmm_xclear`) even when followed by an element-wise initialization. A key based on a contiguous array has no gaps by definition and it is enough to initialize the array elements. A Fortran example is given as part of the Dispatch Microbenchmark.

```
FUNCTION libxsmm_xregister(key, keysize, valsize, valinit)
  TYPE(C_PTR), INTENT(IN), VALUE :: key
  TYPE(C_PTR), INTENT(IN), VALUE, OPTIONAL :: valinit
  INTEGER(C_INT), INTENT(IN) :: keysize, valsize
  TYPE(C_PTR) :: libxsmm_xregister
END FUNCTION

FUNCTION libxsmm_xdispatch(key, keysize)
  TYPE(C_PTR), INTENT(IN), VALUE :: key
  INTEGER(C_INT), INTENT(IN) :: keysize
  TYPE(C_PTR) :: libxsmm_xdispatch
END FUNCTION
```

Note: This functionality can be used to, e.g., dispatch multiple kernels in one step if a code location relies on multiple kernels. This way, one can pay the cost of dispatch one time per task rather than according to the number of JIT-kernels used by this task. However, the functionality is not limited to multiple kernels but any data can be registered and queried. User-data dispatch uses the same implementation as regular code-dispatch.

Memory Allocation

The C interface (libxsmm_malloc.h) provides functions for aligned memory one of which allows to specify the alignment (or to request an automatically selected alignment). The automatic alignment is also available with a `malloc` compatible signature. The size of the automatic alignment depends on a heuristic, which uses the size of the requested buffer.
Note: The function `libxsmm_free` must be used to deallocate buffers allocated by LIBXSMM's allocation functions.

```
void* libxsmm_malloc(size_t size);
void* libxsmm_aligned_malloc(size_t size, size_t alignment);
void* libxsmm_aligned_scratch(size_t size, size_t alignment);
void libxsmm_free(const volatile void* memory);
int libxsmm_get_malloc_info(const void* m, libxsmm_malloc_info* i);
int libxsmm_get_scratch_info(libxsmm_scratch_info* info);
```

The library exposes two memory allocation domains: (1) default memory allocation, and (2) scratch memory allocation. There are similar service functions for both domains that allow to customize the allocation and deallocation function. The "context form" even supports a user-defined "object", which may represent an allocator or any other external facility. To set the allocator of the default domain is analogous to setting the allocator of the scratch memory domain (shown below).

```
int libxsmm_set_scratch_allocator(void* context,
  libxsmm_malloc_function malloc_fn, libxsmm_free_function free_fn);
int libxsmm_get_scratch_allocator(void** context,
  libxsmm_malloc_function* malloc_fn, libxsmm_free_function* free_fn);
```

The scratch memory allocation is very effective and delivers a decent speedup over subsequent regular memory allocations. In contrast to the default allocator, a watermark for repeatedly allocated and deallocated buffers is established. The scratch memory domain is (arbitrarily) limited to 4 GB of memory which can be adjusted to a different number of Bytes (available per libxsmm_malloc.h, and also per environment variable LIBXSMM_SCRATCH_LIMIT with optional "k|K", "m|M", "g|G" units, unlimited per "-1").

```
void libxsmm_set_scratch_limit(size_t nbytes);
size_t libxsmm_get_scratch_limit(void);
```

By establishing a pool of "temporary" memory, the cost of repeated allocation and deallocation cycles is avoided when the watermark is reached. The scratch memory is scope-oriented with a limited number of pools for buffers of different lifetime or held for different threads. The verbose mode with a verbosity level of at least two (LIBXSMM_VERBOSE=2) shows some statistics about the populated scratch memory.

```
Scratch: 173 MB (mallocs=5, pools=1)
```

To improve thread-scalability and to avoid frequent memory allocation/deallocation, the scratch memory allocator can be leveraged by intercepting existing malloc/free calls.

Note: be careful with scratch memory as it only grows during execution (in between `libxsmm_init` and `libxsmm_finalize` unless `libxsmm_release_scratch` is called). This is true even when `libxsmm_free` is (and should be) used!

Meta Image File I/O

Loading and storing data (I/O) is normally out of LIBXSMM's scope. However, comparing results (correctness) or writing files for visual inspection is clearly desired. This is particularly useful for the DNN domain. The MHD library domain provides support for the Meta Image File format (MHD). Tools such as ITK-SNAP or ParaView can be used to inspect, compare, and modify images (even beyond two-dimensional images).

Writing an image is per `libxsmm_mhd_write`, and loading an image is split in two stages: (1) `libxsmm_mhd_read_header`, and (2) `libxsmm_mhd_read`. The first step allows to allocate a properly sized buffer, which is then used to obtain the data per `libxsmm_mhd_read`. When reading data, an on-the-fly type conversion is supported. Further, data that is already in memory can be compared against file-data without allocating memory or reading this file into memory.

To load an image from a familiar format (JPG, PNG, etc.), one may save the raw data using for instance IrfanView and rely on a "header-only" MHD-file (plain text). This may look like:

```
NDims = 2
DimSize = 202 134
ElementType = MET_UCHAR
ElementNumberOfChannels = 1
ElementDataFile = mhd_image.raw
```

In the above case, a single channel (gray-scale) 202x134-image is described with pixel data stored separately (`mhd_image.raw`). Multi-channel images are expected to interleave the pixel data. The pixel type is per `libxsmm_mhd_elemtype` (libxsmm_mhd.h).

Thread Synchronization

LIBXSMM comes with a number of light-weight abstraction layers (macro and API-based), which are distinct from the internal API (include files in src directory) and that are exposed for general use (and hence part of the include directory).

The synchronization layer is mainly based on macros: LIBXSMM_LOCK_* provide spin-locks, mutexes, and reader-writer locks (LIBXSMM_LOCK_SPINLOCK, LIBXSMM_LOCK_MUTEX, and LIBXSMM_LOCK_RWLOCK respectively). Usually the spin-lock is also named LIBXSMM_LOCK_DEFAULT. The implementation is intentionally based on OS-native primitives unless LIBXSMM is reconfigured (per LIBXSMM_LOCK_SYSTEM) or built using `make OMP=1` (using OpenMP inside of the library is not recommended). The life-cycle of a lock looks like:

```
/* attribute variable and lock variable */
LIBXSMM_LOCK_ATTR_TYPE(LIBXSMM_LOCK_DEFAULT) attr;
LIBXSMM_LOCK_TYPE(LIBXSMM_LOCK_DEFAULT) lock;
/* attribute initialization */
LIBXSMM_LOCK_ATTR_INIT(LIBXSMM_LOCK_DEFAULT, &attr);
/* lock initialization per initialized attribute */
LIBXSMM_LOCK_INIT(LIBXSMM_LOCK_DEFAULT, &lock, &attr);
/* the attribute can be destroyed */
LIBXSMM_LOCK_ATTR_DESTROY(LIBXSMM_LOCK_DEFAULT, &attr);
/* lock destruction (usage: see below/next code block) */
LIBXSMM_LOCK_DESTROY(LIBXSMM_LOCK_DEFAULT, &lock);
```

Once the lock is initialized (or an array of locks), it can be exclusively locked or try-locked, and released at the end of the locked section (LIBXSMM_LOCK_ACQUIRE, LIBXSMM_LOCK_TRYLOCK, and LIBXSMM_LOCK_RELEASE respectively):

```
LIBXSMM_LOCK_ACQUIRE(LIBXSMM_LOCK_DEFAULT, &lock);
/* locked code section */
LIBXSMM_LOCK_RELEASE(LIBXSMM_LOCK_DEFAULT, &lock);
```

If the lock-kind is LIBXSMM_LOCK_RWLOCK, non-exclusive a.k.a. shared locking allows to permit multiple readers (LIBXSMM_LOCK_ACQREAD, LIBXSMM_LOCK_TRYREAD, and LIBXSMM_LOCK_RELREAD) if the lock is not acquired exclusively (see above). An attempt to only read-lock anything else but an RW-lock is an exclusive lock (see above).

```
if (LIBXSMM_LOCK_ACQUIRED(LIBXSMM_LOCK_RWLOCK) ==
    LIBXSMM_LOCK_TRYREAD(LIBXSMM_LOCK_RWLOCK, &rwlock))
{ /* locked code section */
  LIBXSMM_LOCK_RELREAD(LIBXSMM_LOCK_RWLOCK, &rwlock);
}
```

Locking different sections for read (LIBXSMM_LOCK_ACQREAD, LIBXSMM_LOCK_RELREAD) and write (LIBXSMM_LOCK_ACQUIRE, LIBXSMM_LOCK_RELEASE) may look like:

```
LIBXSMM_LOCK_ACQREAD(LIBXSMM_LOCK_RWLOCK, &rwlock);
/* locked code section: only reads are performed */
LIBXSMM_LOCK_RELREAD(LIBXSMM_LOCK_RWLOCK, &rwlock);

LIBXSMM_LOCK_ACQUIRE(LIBXSMM_LOCK_RWLOCK, &rwlock);
/* locked code section: exclusive write (no R/W) */
LIBXSMM_LOCK_RELEASE(LIBXSMM_LOCK_RWLOCK, &rwlock);
```

For a lock not backed by an OS level primitive (fully featured lock), the synchronization layer also a simple lock based on atomic operations:

```
static union { char pad[LIBXSMM_CACHELINE]; volatile LIBXSMM_ATOMIC_LOCKTYPE state; } lock;
LIBXSMM_ATOMIC_ACQUIRE(&lock.state, LIBXSMM_SYNC_NPAUSE, LIBXSMM_ATOMIC_RELAXED);
/* locked code section */
LIBXSMM_ATOMIC_RELEASE(&lock.state, LIBXSMM_ATOMIC_RELAXED);
```

Performance Analysis

Intel VTune Profiler

To analyze which kind of kernels have been called, and from where these kernels have been invoked (call stack), the library allows profiling its JIT code using Intel VTune Profiler. To enable this support, VTune's root directory needs to be set at build-time of the library. Enabling symbols (SYM=1 or DBG=1) incorporates VTune's JIT Profiling API:

```
source /opt/intel/vtune_profiler/vtune-vars.sh
make SYM=1
```

Above, the root directory is automatically determined from the environment (VTUNE_PROFILER_*_DIR or VTUNE_AMPLIFIER*_DIR with older versions). This variable is present after source'ing the Intel VTune environment (`source /path/to/vtune_amplifier/amplxe-vars.sh` with older version), but it can be manually provided as well (`make VTUNEROOT=/path/to/vtune_amplifier`). Symbols are not really required to display kernel names for the dynamically generated code, however enabling symbols makes the analysis much more useful for the rest of the (static) code, and hence it has been made a prerequisite. For example, when "call stacks" are collected it is possible to find out where the JIT code has been invoked by the application:

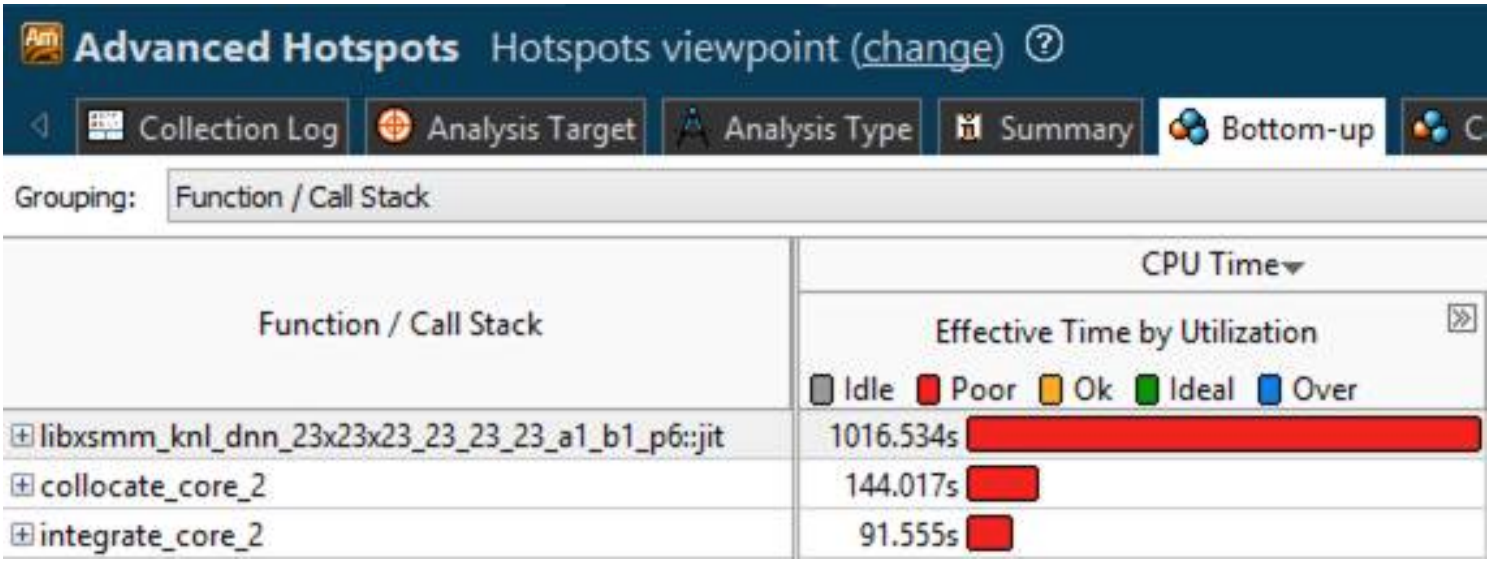
```
vtune -r resultdir -data-limit 0 -collect hotspots \
      -knob enable-stack-collection=true \
      -knob sampling-mode=hw \
      -knob stack-size=0 \
      -- ./myapplication
```

In case of an MPI-parallelized application, it can be useful to only collect results from a "representative" rank, and to also avoid running the event collector in every rank of the application. With Intel MPI both of which can be achieved by:

```
mpirun -gtool 'vtune -r resultdir -data-limit 0 -collect hotspots \
               -knob sampling-mode=hw -knob enable-stack-collection=true \
               -knob stack-size=0:4=exclusive' \
  [...] ./myapplication
```

The `:4=exclusive` is related to Intel MPI or mpirun's gtool arguments and unrelated to VTune's command line syntax (see `vtune --help` or `amplxe-cl --help` with older versions); such argument(s) need to appear at the end of the gtool-string. For instance, the shown command line selects the 5th rank (zero-based) along with exclusive usage of the performance monitoring unit (PMU) such that only one event-collector runs for all ranks (without rank-number, all ranks are sampled).

Intel VTune Profiler presents invoked JIT code like functions, which belong to a module named "libxsmm.jit". The function name as well as the module name are supplied by LIBXSMM using VTune's JIT-Profiling API. Below, the shown "function name" (`libxsmm_knl_dnn_23x23x23_23_23_23_a1_b1_p6::mxm`) encodes an AVX-512 ("knl") double-precision kernel ("d") for small dense matrix multiplication, which performs no transposes ("nn"). The name further encodes M=N=K=LDA=LDB=LDC=23, Alpha=Beta=1.0, and a prefetch strategy ("p6").



An application that cannot rely on LIBXSMM's build system can apply `-DLIBXSMM_VTUNE=2` during compilation, and link against `${VTUNE_AMPLIFIER_XE_2017_DIR}/lib64/libjitprofiling.a`.

Linux perf

With LIBXSMM, there is both basic (`perf map`) and extended support (`jitdump`) when profiling an application. To enable perf support at runtime, the environment LIBXSMM_VERBOSE needs to be set to a negative value.

- The basic support can be enabled at compile-time with PERF=1 (implies SYM=1) using `make PERF=1`. At runtime of the application, a map-file ('jit-*pid*.map') is generated ('/tmp' directory). This file is automatically read by Linux perf, and enriches the information about unknown code such as JIT'ted kernels.
- The support for "jitdump" can be enabled by supplying JITDUMP=1 (implies PERF=1) or PERF=2 (implies JITDUMP=1) when making the library: `make JITDUMP=1` or `make PERF=2`. At runtime of the application, a dump-file ('jit-*pid*.dump') is generated (in perf's debug directory, usually `$HOME/.debug/jit/`) which includes information about JIT'ted kernels (such as addresses, symbol names, code size, and the code itself). The dump file can be injected into `perf.data` (using `perf inject -j`), and it enables an annotated view of the assembly in perf's report (requires a reasonably recent version of Linux perf).

Customization

Intercepted Allocations

To improve thread-scalability and to avoid frequent memory allocation/deallocation, the scratch memory allocator can be leveraged by intercepting existing malloc/free calls. This facility is built into LIBXSMM's main library, but disabled at compile-time (by default); build with `make MALLOC=1` to permanently enable, or build with `make MALLOC=-1` to even require an environment variable `LIBXSMM_MALLOC=1` or an API-call (`libxsmm_set_malloc`). Both runtime settings allow an optional lower and/or an upper bound to select malloc-calls based on the size of the allocation. For the environment option, an extra variable is introduced, e.g., use `LIBXSMM_MALLOC=1 LIBXSMM_MALLOC_LIMIT=4m:1g`.

```
void libxsmm_set_malloc(int enabled, const size_t* lo, const size_t* hi);
int libxsmm_get_malloc(size_t* lo, size_t* hi);
```

Querying the status may return zero even if there was an attempt to enable this facility (limitation/experimental implementation). Please note, the regular Scratch Memory API (e.g., `libxsmm_[get|set]_scratch_limit`) and the related environment variables can apply as well (`LIBXSMM_SCRATCH_LIMIT`, `LIBXSMM_SCRATCH_POOLS`, `LIBXSMM_SCRATCH_SCALE`). If intercepted memory allocations are enabled, the scratch limit is adjusted by default to allow unlimited growth of the scratch domain. Further, an increased verbosity level can help to gain some insight (`LIBXSMM_VERBOSE=3`).

Intercepting malloc/free is supported by linking LIBXSMM's static or shared main library. The latter of which can be used to intercept calls of an existing and unchanged binary (LD_PRELOAD mechanism). To statically link with LIBXSMM and to intercept existing malloc/free calls, the following changes to the application's link stage are recommended:

```
gcc [...] -Wl,--export-dynamic \
  -Wl,--wrap=malloc,--wrap=calloc,--wrap=realloc \
  -Wl,--wrap=memalign,--wrap=free \
  /path/to/libxsmm.a
```

The main library causes a BLAS-dependency which may be already fulfilled for the application in question. However, if this is not the case (unresolved symbols), `libxsmmnoblas.a` must be linked in addition. Depending on the dependencies of the application, the link order may also need to be adjusted. Other i.e. a GNU-compatible compiler (as shown above), can induce additional requirements (compiler runtime libraries).

Note: The Intel Compiler may need "libirc", i.e., `-lirc` in front of `libxsmm.a`. Linking LIBXSMM's static library may require above mentioned linker flags (`--wrap`) in particular when using Intel Fortran (IFORT) as a linker driver unless `CALL libxsmm_init()` is issued (or at least one symbol of LIBXSMM's main library is referenced; check with `nm application | grep libxsmm`). Linking the static library by using the GNU compiler does not strictly need special flags when linking the application.

Linking the shared library form of LIBXSMM (`make STATIC=0`) has similar requirements with respect to the application but does not require `-Wl,--wrap` although `-Wl,--export-dynamic` is necessary if the application is statically linked (beside of LIBXSMM linked in a shared fashion). The LD_PRELOAD based mechanism does not need any changes to the link step of an application. However, `libxsmmnoblas` may be required if the application does not already link against BLAS.

```
LD_PRELOAD="libxsmm.so libxsmmnoblas.so"
LD_LIBRARY_PATH=/path/to/libxsmm/lib:${LD_LIBRARY_PATH}
LIBXSMM_MALLOC=1
```

Note: If the application already uses BLAS, of course `libxsmmnoblas` must not be used!

The following code can be compiled and linked with `gfortran example.f -o example`:

```
      PROGRAM allocate_test
        DOUBLE PRECISION, ALLOCATABLE :: a(:), b(:), c(:)
        INTEGER :: i, repeat = 100000
        DOUBLE PRECISION :: t0, t1, d

        ALLOCATE(b(16*1024))
        ALLOCATE(c(16*1024))
        CALL CPU_TIME(t0)
        DO i = 1, repeat
          ALLOCATE(a(16*1024*1024))
          DEALLOCATE(a)
        END DO
        CALL CPU_TIME(t1)
        DEALLOCATE(b)
        DEALLOCATE(c)
        d = t1 - t0

        WRITE(*, "(A,F10.1,A)") "duration:", (1D3 * d), " ms"
      END PROGRAM
```

Running with `LIBXSMM_VERBOSE=3 LIBXSMM_MALLOC=1 LD_PRELOAD=... LD_LIBRARY_PATH=... ./example` displays: `Scratch: 132 MB (mal`
which shows the innermost allocation/deallocation was served by the scratch memory allocator.

Static Specialization

By default, LIBXSMM uses the JIT backend which is automatically building optimized code (JIT=1). Matrix multiplication kernels can be also statically specialized at compile-time of the library (M, N, and K values). This mechanism also extends the interface of the library because function prototypes are included into both the C and FORTRAN interface.

```
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 indexes. 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 does not participate 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 by using the same command line (make). Static optimization and JIT can also be combined (no need to turn off the JIT backend).

User-Data Dispatch

It can be desired to dispatch user-defined data, i.e., to query a value based on a key. This functionality can be used to, e.g., dispatch multiple kernels in one step if a code location relies on multiple kernels. This way, one can pay the cost of dispatch one time per task rather than according to the number of JIT-kernels used by this task. This functionality is detailed in the section about Service Functions.

Targeted Compilation

Specifying a code path is not necessary if the JIT backend is not disabled. However, disabling JIT compilation, statically generating a collection of kernels, and targeting a specific instruction set extension for the entire library looks like:

```
make JIT=0 AVX=3 MNK="1 2 3 4 5"
```

The above example builds a library which cannot be deployed to anything else but the Intel Knights Landing processor family ("KNL") or future Intel Xeon processors supporting foundational Intel AVX-512 instructions (AVX-512F). The latter might be even more adjusted by supplying MIC=1 (along with AVX=3), however this does not matter since critical code is in inline assembly (and not affected). Similarly, SSE=0 (or JIT=0 without SSE or AVX build flag) employs an "arch-native" approach whereas AVX=1, AVX=2 (with FMA), and AVX=3 are specifically selecting the kind of Intel AVX code. Moreover, controlling the target flags manually or adjusting the code optimizations is also possible. The following example is GCC-specific and corresponds to OPT=3, AVX=3, and MIC=1:

```
make OPT=3 TARGET="-mavx512f -mavx512cd -mavx512er -mavx512pf"
```

An extended interface can be generated which allows to perform software prefetches. Prefetching data might be helpful when processing batches of matrix multiplications where the next operands are farther away or otherwise unpredictable in their memory location. The prefetch strategy can be specified similar as shown in the section Generator Driver, i.e., by either using the number of the shown enumeration, or by exactly using the name of the prefetch strategy. The only exception is PREFETCH=1 which is automatically selecting a strategy per an internal table (navigated by CPUID flags). The following example is requesting the "AL2jpst" strategy:

```
make PREFETCH=8
```

The prefetch interface is extending the signature of all kernels by three arguments (pa, pb, and pc). These additional arguments are specifying the locations of the operands of the next multiplication (the next a, b, and c matrices). Providing unnecessary arguments in case of the three-argument kernels is not big a problem (beside of some additional call-overhead), however running a 3-argument kernel with more than three arguments and thereby picking up garbage data is misleading or disabling the hardware prefetcher (due to software prefetches). In this case, a misleading prefetch location is given plus an eventual page fault due to an out-of-bounds (garbage-)location.

Further, a generated configuration (template) of the library encodes the parameters for which the library was built for (static information). This helps optimizing client code related to the library's functionality. For example, the LIBXSMM_MAX_* and LIBXSMM_AVG_* information can be used with the LIBXSMM_PRAGMA_LOOP_COUNT macro to hint loop trip counts when handling matrices related to the problem domain of LIBXSMM.

Auto-dispatch

The function `libxsmm_?mmdispatch` helps amortizing the cost of the dispatch when multiple calls with the same M, N, and K are needed. The automatic code dispatch is orchestrating two levels:

1. Specialized routine (implemented in assembly code),
2. BLAS library call (fallback).

Both levels are accessible directly, which allows to customize the code dispatch. The fallback level may be supplied by the Intel Math Kernel Library (Intel MKL) 11.2 DIRECT CALL feature.

Further, a preprocessor symbol denotes the largest problem-size (M x N x K) that belongs to the first level, and therefore determines if a matrix multiplication falls back to BLAS. The problem-size threshold can be configured by using for example:

```
make THRESHOLD=$((60 * 60 * 60))
```

The maximum of the given threshold and the largest requested specialization refines the value of the threshold. Please note that explicitly JIT'ting and executing a kernel is possible and independent of the threshold. If a problem-size is below the threshold, dispatching the code requires to figure out whether a specialized routine exists or not.

For statically generated code, the precision can be selected:

```
make PRECISION=2
```

The default preference is to generate and register both single and double-precision code (PRECISION=0). Specifying PRECISION=1|2 is generating and registering single-precision or double-precision code respectively.

The automatic dispatch is highly convenient because existing GEMM calls can serve specialized kernels (even in a binary compatible fashion), however there is (and always will be) an overhead associated with looking up the code-registry and checking whether the code determined by the GEMM call is already JIT'ted or not. This lookup has been optimized with various techniques such as specialized CPU instructions to calculate CRC32 checksums, to avoid costly synchronization (needed for thread-safety) until it is ultimately known that the requested kernel is not yet JIT'ted, and by implementing a small thread-local cache of recently dispatched kernels. The latter of which can be adjusted in size (only power-of-two sizes) but also disabled:

```
make CACHE=0
```

Please note that measuring the relative cost of automatically dispatching a requested kernel depends on the kernel size (obviously smaller matrices are multiplied faster on an absolute basis), however smaller matrix multiplications are bottlenecked by memory bandwidth rather than arithmetic intensity. The latter implies the highest relative overhead when (artificially) benchmarking the very same multiplication out of the CPU-cache.

Backend

Code Generator (JIT)

There can be situations in which it is up-front not clear which problem-sizes will be needed when running an application. To leverage LIBXSMM's high-performance kernels, the library implements a JIT (Just-In-Time) code generation backend which generates the requested kernels on the fly (in-memory). This is accomplished by emitting the corresponding byte-code directly into an executable buffer. The actual JIT code is generated per the CPUID flags, and therefore does not rely on the code path selected when building the library. In the current implementation, some limitations apply to the JIT backend specifically:

1. To stay agnostic to any threading model used, Pthread mutexes are guarding the updates of the JIT'ted code cache (link line with `-lpthread` is required); building with OMP=1 employs an OpenMP critical section as an alternative locking mechanism.
2. There is limited support for the Windows calling convention (only kernels without prefetch signature).

The JIT backend can also be disabled at build time (`make JIT=0`) as well as at runtime (`LIBXSMM_TARGET=0`, or anything prior to Intel AVX). The latter is an environment variable which allows to set a code path independent of the CPUID (LIBXSMM_TARGET=0|1|sse|snb|hsw|knl|knm|skx|clx|cpx|spr). Please note that LIBXSMM_TARGET cannot enable the JIT backend if it was disabled at build time (JIT=0).

One can use the afore mentioned THRESHOLD parameter to control the matrix sizes for which the JIT compilation will be automatically performed. However, explicitly requested kernels (by calling `libxsmm_?mmdispatch`) fall not under a threshold for the problem-size. In any case, JIT code generation can be used for accompanying statically generated code.

Generator Driver

In rare situations, it might be useful to directly incorporate generated C code (with inline assembly regions). This is accomplished by invoking a driver program (with certain command line arguments).

Note: The stand-alone generator-driver is considered legacy (deprecated). Associated functionality may be removed and future instruction set extensions may not be addressed with printed assembly code. The cost of dispatching JIT-code for every code region of an application, and for every visit of such region, can be amortized in several ways and without dispensing JIT-generated code. Dispatching multiple kernels at once or (most effectively) tabulating JIT'ted function pointers manually, can elleviate or remove first-time code generation and (more important) the cost of subsequently dispatching kernels (when code was already JIT-generated).

The generator driver program is usually built as part of LIBXSMM's build process, but also available as a separate build target:

```
make generator
bin/libxsmm_gemm_generator
```

The code generator driver program accepts the following arguments:

1. Select: dense, dense_asm, sparse, sparse_csr, or sparse_csr_reg
2. Filename of a file to append to
3. Routine name to be created
4. M parameter
5. N parameter
6. K parameter
7. LDA (0 indicates A is sparse if 1st arg. is "sparse*")
8. LDB (0 indicates B is sparse if 1st arg. is "sparse*")
9. LDC parameter
10. Alpha (1)
11. Beta: (0 or 1)
12. Alignment override for A (1 auto, 0 unalignment)
13. Alignment override for C (1 auto, 0 unalignment)
14. Architecture (noarch, wsm, snb, hsw, knl, knm, skx, clx, cpx)
15. Prefetch strategy, see below (only nopf or pfsigonly for "sparse*")
16. SP (single-precision), DP (double-recision), or I16 (only "dense*")
17. CSC file in Matrix market format (only if 1st arg. is "sparse*").

The prefetch strategy can be:

1. "nopf": data is not prefetched, just three arguments: A, B, and C
2. "pfsigonly": no prefetches, kernel signature: A, B, C, A', B', and C'
3. "BL2viaC": uses accesses to C to prefetch B'
4. "AL2": uses accesses to A to prefetch A
5. "curAL2": prefetches current A ahead in the kernel
6. "AL2_BL2viaC": combines AL2 and BL2viaC
7. "curAL2_BL2viaC": combines curAL2 and BL2viaC

Here are some examples of invoking the driver program:

```
bin/libxsmm_gemm_generator dense foo.c foo 16 16 16 32 32 32 1 1 1 1 hsw nopf DP
bin/libxsmm_gemm_generator dense_asm foo.c foo 16 16 16 32 32 32 1 1 1 1 knl AL2_BL2viaC DP
bin/libxsmm_gemm_generator sparse foo.c foo 16 16 16 32 0 32 1 1 1 1 hsw nopf DP bar.csc
```

Please note, there are additional examples given in samples/generator and samples/seissol.

Development Concepts

The low-level code generator is hosted by a single translation unit (src/generator_x86_instructions.c). The code generator emits instructions as enumerated in src/generator_common.h. A kernel then is a buffered stream of instructions in either binary/encoded or textual form. The latter is leveraged by stand-alone generator drivers that can print C functions with an assembly section (inline). A generator driver may exists for some of LIBXSMM's

function domains. Please note that emitting the textual form is not needed to inspect the emitted code since the binary encoded form can be easily disassembled (objdump).

The binary encoded form is directly suitable for execution by casting the code-buffer into a function-pointer of the corresponding signature. It is advised to rely on LIBXSMM's internal memory allocation routines to acquire an executable buffer (see libxsmm_malloc_flags, libxsmm_xmalloc, and libxsmm_malloc_attrib in src/libxsmm_main.h). This ensures correct behavior in security-hardened environments. As a bonus, profiler support for the emitted code is enabled transparently.

To debug the JIT'ted code, GNU GDB can be used to disassemble a given memory address (`disas address,+length`). Having the code disassembled side-by-side (while debugging) helps to look ahead and to have some orientation. For the latter, objdump can be used to acquire the source code (assembly) along with hexadecimal line numbers (length). The offset position (for GDB's disas) directly corresponds to objectdump's line numbers.

The kernel development is much like assembly programming, except that an API is used to emit instructions. For further reference, some existing source code for building kernels can be inspected (e.g., matcopy). This may help to capture the concept of mapping registers (basically a table to avoid hard-coding register names).

Appendix

Compatibility

Linux

All Linux distributions are meant to be fully supported (please report any compatibility issue). A shared library (`STATIC=0`) necessarily implies some performance hit when accessing thread-local memory (contended multicore execution). The GNU Compiler Collection prior to v5.1 may imply performance hits in some CPUID-dispatched code paths (non-JIT).

> In case of outdated Binutils, compilation can fail to assemble code that originates from code sections using Intrinsics (see issue #170 and #212). To resolve the problem, please use `INTRINSICS=1` along with the desired target e.g., `AVX=3 MIC=0`, or `AVX=2`.

CRAY

In addition to the regular Linux support, The CRAY Compiling Environment (CCE) is supported: Intel Compiler as well as the GNU Compiler Collection are detected even when invoked per CCE, and the CRAY compiler is likely configured to build for the architecture of the compute nodes and hence the compiler is sufficiently treated without specific build flags (`COMPATIBLE=1` is implicitly set). The CCE may suppress to build a shared library (`STATIC=0`), which also affects the TRACE facility (requires dynamic linkage even for static archives).

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

The compatibility settings imply minor issues when using the CRAY compiler: full control and customization is not implemented, enabling symbols (`SYM=1`) appears to imply an unoptimized debug-build (due to the `-g` flag being present). Some sample codes/benchmarks enable symbols but are meant to not enable debug-code. The LIBXSMM library however is built without symbols by default.

Windows

Microsoft Windows Microsoft Windows is supported using the Microsoft Visual Studio environment (no `make`). It is advised to review the build settings. However, the following configurations are available: `debug`, `release`, and release mode with `symbols`. JIT-code generation is enabled but limited to the MM domain (GEMM kernels and matcopy kernels; no transpose kernels). GEMM kernels with prefetch signature remain as non-prefetch kernels i.e., prefetch locations are ignored due to the effort of fully supporting the Windows calling convention. As a workaround and to properly preserve caller-state, each JIT-kernel call may be wrapped by an own function.

Cygwin Cygwin (non-MinGW) is fully supported. Please note, that all limitations of Microsoft Windows apply.

```
make
```

LIBXSMM can be built as a static library as well as a dynamic link library (STATIC=0).

MinGW/Cygwin This is about the Cygwin-hosted bits of MinGW. The `-fno-asynchronous-unwind-tables` compiler flag is automatically applied. Please note, that all limitations of Microsoft Windows apply.

```
make \
  CXX=x86_64-w64-mingw32-g++ \
  CC=x86_64-w64-mingw32-gcc \
  FC=x86_64-w64-mingw32-gfortran
```

To run tests, `BLAS=0` may be supplied (since Cygwin does not seem to provide BLAS-bits for the MinGW part). However, this may be different for "native" MinGW, or can be fixed by supplying a BLAS library somehow else.

MinGW This is about the "native" MinGW environment. Please note, there is the original MinGW as well as a fork (made in 2007). Both of which can target Windows 64-bit. Here, the MSYS2 installer (scroll down on that page to see the full installation instructions) has been used (see the details on how to install missing packages).

```
pacman -S msys/make msys/python msys/diffutils \
  mingw64/mingw-w64-x86_64-gcc mingw64/mingw-w64-x86_64-gcc-fortran \
  mingw64/mingw-w64-x86_64-openblas
```

Similar to Cygwin/MinGW, the `-fno-asynchronous-unwind-tables` flag is automatically applied.

```
make
```

LIBXSMM can be built as a static library as well as a dynamic link library (`STATIC=0`).

Apple macOS

LIBXSMM for macOS (OSX) is fully supported (i.e., it qualifies a release). The default is to rely on Apple's Clang based (platform-)compiler ("gcc"). However, the actual GCC as well as the Intel Compiler for macOS can be used.

FreeBSD

LIBXSMM is occasionally tested under FreeBSD. For libxsmmext, it is necessary to install OpenMP (`sudo pkg install openmp`).

```
bash
gmake
```

An attempt to run the tests may ask for a LAPACK/BLAS installation (unless `BLAS=0` is given). Both, Netlib BLAS (reference) and OpenBLAS are available (in case of linker error due to the GNU Fortran runtime library, one can try `gmake CXX=g++7 CC=gcc7 FC=gfortran7` i.e., select a consistent tool chain and adjust `LD_LIBRARY_PATH` accordingly e.g., `/usr/local/lib/gcc7`).

PGI Compiler

The PGI Compiler 2019 (and later) is supported. Earlier versions were only occasionally tested and automatically enabled the `COMPATIBLE=1` and `INTRINSIC=0` settings. Still, atomic builtins seem incomplete (at least with `pgcc`) hence LIBXSMM built with PGI Compiler is not fully thread-safe (tests/threadsafety can fail). Support for GNU's libatomic has been incorporated mainly for PGI but is also missing built-in compiler support hence supposedly atomic operations are mapped to normal (non-atomic) code sequences (`LIBXSMM_SYNC_SYSTEM`).

```
make CXX=pgc++ CC=pgcc FC=pgfortran
```

ARM AArch64 LIBXSMM 2.0 is the initial version supporting AArch64 (baseline is v8.1), which practically covers ARM 64-bit architecture from embedded and mobile to supercomputers. The build and installation process of LIBXSMM is the same as for Intel Architecture (IA) and the library can be natively compiled or cross-compiled. The latter for instance looks like:

```
make PLATFORM=1 AR=aarch64-linux-gnu-ar \
  FC=aarch64-linux-gnu-gfortran \
  CXX=aarch64-linux-gnu-g++ \
  CC=aarch64-linux-gnu-gcc
```

Validation

Basic Tests

To run basic tests:

```
make tests
```

Remember: a set of key-value pairs represents a single unique (re-)build (and test):

```
make STATIC=0 tests
```

Test Suites

It is possible to run whole test suites or collections of tests like for LIBXSMM's Continuous Integration (CI). The script `tool_test.sh` is included in archives and releases, i.e., it also works for non-repository folders. To run an entire collection (aka `scripts/tool_test.sh 0`).

```
scripts/tool_test.sh
```

It is also possible to select a single test (out of the whole collection):

```
scripts/tool_test.sh 1
```

In general, key-value pairs which are valid for LIBXSMM's `make` can be specified:

```
AVX=3 DBG=1 scripts/tool_test.sh
```

There are several collections of tests covering specific domains:

- `samples/utilities/wrap/wrap-test.sh`: test substituting standard symbols at link/run-time (gemm, gemv, etc).
- `samples/xgemm/kernel_test.sh`: test SMM kernels in an almost exhaustive fashion (brute-force).
- `samples/eltwise/run_test.sh`: test all kinds of element-wise kernels and variants.
- `samples/pyfr/test.sh`: test Sparse Matrix times Dense Matrix (FsSpMDM).

Reproduce Failures

LIBXSMM's verbose mode can print the invocation arguments when launching a test driver (`LIBXSMM_VERBOSE=4` and beyond). For example (`LIBXSMM_VERBOSE=4 ./run_test_avx2.sh`), the termination message of a failing test may look like:

```
[...]
LIBXSMM_TARGET: hsw
Registry and code: 13 MB + 8 KB (meltw=1)
Command: ./eltwise_binary_simple 1 0 F32 F32 F32 F32 10 10 10 10
[...]
```

Note: scripts such `scripts/tool_pexec.sh` suppress error output (console) by default and capture error output in individual files, i.e., verbose output may not be immediately visible.

Portability

It is desirable to exercise portability and reliability of LIBXSMM's source code even on Non-Intel Architecture by the means of compilation, linkage, and generic tests. This section is *not* about Intel Architecture (or compatible). Successful compilation (or even running some of the tests successfully) does not mean LIBXSMM is valuable on that platform.

Make sure to rely on `PLATFORM=1`, otherwise a compilation error should occur *Intel Architecture or compatible CPU required!* This error avoids (automated) attempts to upstream LIBXSMM to an unsupported platform. LIBXSMM is upstreamed for Intel Architecture on all major Linux distributions, FreeBSD, and others. If compilation fails with *LIBXSMM is only supported on a 64-bit platform!*, `make PLATFORM=1 DBG=1` can be used to exercise compilation.

If platform support is forced (`PLATFORM=1`), runtime code generation is disabled at compile-time (`JIT=0`). Runtime code generation can be also enabled (`PLATFORM=1 JIT=1`) but code-dispatch will still return NULL-kernels. However, some tests will start failing as missing JIT-support it is not signaled at compile-time as with `JIT=0`.

Note: JIT-support normally guarantees a non-NULL code pointer ("kernel") if the request is according to the limitations (user-code is not asked to check for a NULL-kernel), which does not hold true if JIT is enabled on a platform that does not implement it.

TinyCC The Tiny C Compiler (TinyCC) supports Intel Architecture, but lacks at least support for thread-local storage (TLS).

```
make CC=tcc THREADS=0 INTRINSICS=0 VLA=0 ASNEEDED=0 BLAS=0 FORCE_CXX=0
```

IBM XL Compiler for Linux (POWER) The POWER platform requires aforementioned `PLATFORM=1` to unlock compilation.

```
make PLATFORM=1 CC=xlc CXX=xlc++ FC=xlf
```

Cross-compilation for ARM ARM AArch64 is regularly supported. However, 32-bit ARM requires aforementioned `PLATFORM=1` to unlock compilation (similar to 32-bit Intel Architecture). Unlocking compilation for 32-bit ARM is not be confused with supporting 32-bit ARM architectures.

```
make PLATFORM=1 AR=arm-linux-gnueabi-ar \
  FC=arm-linux-gnueabi-gfortran \
  CXX=arm-linux-gnueabi-g++ \
  CC=arm-linux-gnueabi-gcc
```

Q&A

What is the background of the name "LIBXSMM"?

The "MM" stands for Matrix Multiplication, and the "S" clarifies the working domain i.e., Small Matrix Multiplication. The latter also means the name is neither a variation of "MXM" nor an eXtreme Small Matrix Multiplication but rather about Intel Architecture (x86) - and no, the library is 64-bit only. The spelling of the name might follow the syllables of libx\/smm, libx'smm, or libx-smm.

> **NOTE**: the library does not support 32-bit architecture (64-bit only)

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 }*, and *TransA := 'N'*.

> **NOTE**: *Alpha*, *Beta*, and *TransA* are limited to `1`, `{ 1, 0 }`, and `'N'` respectively.

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.

What about "medium-sized" and big(ger) matrix multiplications?

For cache-tiled or parallelized routines, please rely for example on OpenBLAS or Intel Math Kernel Library (Intel MKL). It is possible to reuse LIBXSMM's kernels for big(ger) matrix multiplications however, an implementation is out of scope for LIBXSMM's core functionality.

How to determine whether an application can benefit from using LIBXSMM or not?

Given the application uses BLAS to carry out matrix multiplications, one may use the Call Wrapper, and measure the application performance e.g., time to solution. However, the latter can significantly improve when using LIBXSMM's API directly. To check whether there are applicable GEMM-calls, the Verbose Mode can help to collect an insight. Further, when an application uses Intel MKL 11.2 (or higher), then running the application with the environment variable MKL_VERBOSE=1 (`env MKL_VERBOSE=1 ./workload > verbose.txt`) can collect a similar insight (`grep -a "MKL_VERBOSE DGEMM(N,N" verbose.txt | cut -d'(' -f2 | cut -d, -f3-5"`).

Is LIBXSMM compatible from version-to-version, or what is the ABI commitment?

One may have a look at issue #120 or #282, but in summary:

- Binary compatibility is not continuously tested (only manually for a subset of the API namely SMM domain).
- Major versions are likely breaking binary compatibility with existing integrations (that is typical).
- Minor versions may break binary compatibility of recently introduced features (may not be typical).
- Update and patch versions are binary compatible but may only be released on request (issue).

LIBXSMM's API for Small Matrix Multiplications (SMMs) is considered stable, and all major known applications (e.g., CP2K, EDGE, NEK5K, and SeisSol) either rely on SMMs or are able (and want) to benefit from an improved API of any of the other domains (e.g., DL). Until at least v2.0, LIBXSMM is not able to track or even maintain binary compatibility and hence the SONAME also goes with the semantic version. A list of public functions is maintained (but there is no distinction for a small subset of them that are only meant for communication between LIBXSMM and LIBXSMM/ext).

I am relying on a prebuilt version of CP2K (or another application), is LIBXSMM incorporated and which version is it?

This can be determined using the environment variable `LIBXSMM_VERBOSE=2` (or higher verbosity). It is not even required to use an input or workload since the information in question is presented when the program terminates. For example:

```
LIBXSMM_VERBOSE=1 exe/Linux-x86-64-intelx/cp2k.psmp
[...]
LIBXSMM_VERSION: release-1.11
LIBXSMM_TARGET: clx
```

I am relying on a prebuilt version of an application, and I am concerned about optimal compiler flags.

LIBXSMM uses JIT-generated code according to the CPUID of the system. This is independent of the compiler flags used to build the library. If LIBXSMM was incorporated per classic ABI, `LIBXSMM_DUMP_BUILD=1` environment variable allows to print build flags used for LIBXSMM at termination of the application. This output of `LIBXSMM_DUMP_BUILD=1` can yield hints about the flags used to build the application (if similar).

For concerns regarding the code of an application that cannot benefit from LIBXSMM, one may have a look at the build recipes of the XCONFIGURE project.

What Operating Systems are covered by LIBXSMM, and what about Microsoft Windows?

The answer here focuses on the actual runtime support rather than the supported compiler tool chains used to build the library. All flavors of Linux are supported (if the library was successfully built), which includes installations running a security-hardened Linux kernel (SELinux). The Apple OS (OSX) is supported, which also includes more recent SIP-enabled versions (System Integrity Protection). The BSD OS is likely supported, but building the library is only occasionally validated. Microsoft Windows is supported for non-JIT operation, and for most (e.g., GEMM and MATCOPY) of the JIT-kernels (prefetch signature is not supported). There is currently no support for JIT in the DNN domain (no further check is performed i.e., crash at runtime). See also issue #71.

Does LIBXSMM has some support for GEMV?

The library generates acceptable code when using `M=1` or `N=1`. For example, building with `make M=16 N=1 K=16 AVX=2` and inspecting the assembly (build directory) or dumping/disassembling the JIT code (see reference documentation) shows the minimum number of load/store instructions. Given that GEMV is a memory bound operation, this suggests reasonable code quality. LIBXSMM selects from multiple microkernels (specific for each ISA extension) by using a fixed scheme/heuristic, which should be acceptable for GEMV. The sample code under samples/smm provides ready-to-use benchmark drivers that can help to compare the performance with LAPACK/BLAS. Afore mentioned benchmarks exercise streaming all possible combinations of operands.

What about complex and mixed types?

This question refers to the following kind of element type of the GEMM interface of LIBXSMM:

- Complex types: complex numbers in single and double-precision,
- Mixed types: e.g. real double-precision and complex double-precision There are no (immediate) plans to support more types for the GEMM part. Please note, that LIBXSMM indeed supports lower precision GEMM (wgemm).

What about voting for features?

All feedback and issue reports are handled openly, are welcome and considered (answered, and collected). However, we do not seek for "feature votes" since the development of the library is not a democratic process.

<DEPRECATED> What is the purpose of ROW_MAJOR vs. COL_MAJOR?

This build configuration is deprecated (issue 85), otherwise there is nothing one cannot achieve with row-major as opposed to column-major storage order. In particular the choice is not about whether a program is written in C/C++ or in FORTRAN. The ROW_MAJOR setting is just offered for existing code, which calls into function(s) that assume row-major storage order and where these calls are to be replaced by LIBXSMM in a "1:1 fashion". It is encouraged to avoid the ROW_MAJOR setting since BLAS implies COL_MAJOR (and LIBXSMM is supposed to be compatible with BLAS). More...