LIBXSMM

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

Where to go for documentation?

- **ReadtheDocs**: main and sample documentation with full text search.
- **PDF**: main documentation file, and separate sample documentation.

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

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

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

Documented functionality and available domains:

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

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

Build Instructions

Overview

The main interface file is *generated*, and it is therefore **not** stored in the code repository. Instead, one may have a look at the code generation template files for C/C++ and FORTRAN. The main interface consists of the general interface as well as the matrix multiplication interface.

There are two ways to incorporate LIBXSMM into an application:

- Classic Library (ABI) and Link Instructions
- Header-Only

Classic Library (ABI)

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

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

```
make
```

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

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

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

```
make COMPATIBLE=1
```

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

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

```
make clean
make realclean
```

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

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

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

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

The index sets are in a loop-nest relationship (M(N(K))) when generating the indices. Moreover, an empty index set resolves to the next non-empty outer index set of the loop nest (including to wrap around from the M to K set). An empty index set 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 using the same command line (make). Static optimization and JIT can also be combined (no need to turn off the JIT backend). Testing the library is supported by a variety of targets with "test" and "test-all" being the most prominent for this matter.

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

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

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

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

Link Instructions

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

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

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

Header-Only

Version 1.4.4 introduced support for "header-only" usage in C and C++. By only including 'libxsmm_source.h' allows to get around building the library. However, this gives up on a clearly defined application binary interface (ABI). An ABI may allow for hot-fixes after deploying an application (when relying on the shared library form), and 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 it can also reduce the turnaround time during development of an application (due to longer compilation times). The header file is intentionally named "libxsmm_**source**.h" since this header file relies on the src directory (with the implications as noted earlier).

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

```
make header-only
```

NOTE: Differences between C and C++ makes a header-only implementation (which is portable between both languages) considerably "techy". Mixing C and C++ translation units (which rely on the header-only form of the library) is **not** supported. Also remember: to build an application now shares the same build settings with LIBXSMM! This is important not only to omit debug code inside of LIBXSMM (use `-DNDEBUG`).

Installation

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

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

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

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

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

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

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

Interfaces

General Interface

To initialize the dispatch-table or other 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 to 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 details are covered in a separate document.

Deep Neural Networks

This domain (DNN) is detailed by a separate document. Please also note on how to Get Started with TensorFlow™ using LIBXSMM.

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)
- Loading and storing data (I/O)
- Allocating memory

Runtime Control

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 ($23 < MNK^{1/3} <= 128$; the actual upper bound depends on LIBXSMM_MAX_MNK as selected at compile-time). Keep in mind, that "larger" is supposedly still small in terms of arithmetic intensity (which grows linearly with the kernel size). Unfortunately, the arithmetic intensity depends on the way a kernel is used (which operands are loaded/stored into main memory) and it is not performance-neutral to collect this information.

The TRY counter represents all attempts to register statically generated kernels, and all attempts to dynamically generate and register kernels. The TRY counter includes rejected JIT requests due to unsupported GEMM arguments. The JIT and STA counters distinct the successful cases of the afore mentioned event (TRY) into dynamically (JIT) and statically (STA) generated code. In case the capacity ($O(n) = 10^5$) 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. The latter is already implemented by LIBXSMM's "medium-sized" GEMM routines (`libxsmm_?gemm_omp`), which perform a tiled multiplication. Setting a verbosity level of at least two summarizes the number of registered JIT-generated kernels, which includes the total size and counters for GEMM, MCOPY (matrix copy), and TCOPY (matrix transpose) kernels.

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

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

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

Call Trace

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

```
make TRACE=1
```

Building an application which traces calls (inside of the library) requires the shared library of LIBXSMM, alternatively the application is required to link the static library of LIBXSMM in a dynamic fashion (GNU tool chain: `-rdynamic`). Tracing calls (without debugger) can be 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 general 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/hfp/libxsmm/tree/results#libxsmm-results, or the results can be cloned as shown below.

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

Backend

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

Applications

High Performance Computing (HPC)

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

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

[3] **https://github.com/NekBox/NekBox**: NekBox is a highly scalable and portable spectral element code, which is inspired by the Nek5000 code. NekBox is specialized for box geometries, and intended 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 using the flux reconstruction approach. PyFR 1.6.0 optionally incorporates LIBXSMM as a matrix multiplication provider for the OpenMP backend. Please also note LIBXSMM's PyFR-related code sample.

Machine Learning (ML)

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

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

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

References

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

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

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

LIBXSMM Domains

Matrix Multiplication

Small Matrix Multiplication (SMM)

To perform the dense matrix-matrix multiplication $C_{mxn} = alpha \cdot A_{mxk} \cdot B_{kxn} + beta \cdot C_{mxn}$, the full-blown GEMM interface can be treated with "default arguments" (which is deviating from the BLAS standard, however without compromising the binary compatibility).

```
/** Automatically dispatched dense matrix multiplication (single/double-precision, C code). */
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*/);
/** Automatically dispatched dense matrix multiplication (C++ code). */
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. 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).

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.

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

For convenience, a BLAS-based dense matrix multiplication (`libxsmm_blas_gemm`) is provided for all supported languages which is simply re-exposing the underlying GEMM/BLAS implementation. The BLAS-based GEMM might be useful for validation/benchmark purposes, and more important as a fallback when building an application-specific dispatch mechanism.

```
/** Automatically dispatched 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*/);
```

A more recently added variant of matrix multiplication is parallelized based on the OpenMP standard. These routines will open an internal parallel region and rely on "classic" thread-based OpenMP. If these routines are called from inside of a parallel region, the parallelism will be based on tasks (OpenMP 3.0). Please note that all OpenMP-based routines are hosted by the extension library (libxsmmext), which keeps the main library agnostic with respect to a threading runtime.

```
/** OpenMP parallelized dense matrix multiplication (single/double-precision). */
libxsmm_?gemm_omp(&transa, &transb, &m, &n, &k, &alpha, a, &lda, b, &ldb, &beta, c, &ldc);
```

Manual Code Dispatch and Batched Matrix Multiplication

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. Overloaded function signatures are provided and allow to omit arguments (C++ and FORTRAN), which are then derived from the configurable defaults.

```
/** If non-zero function pointer is returned, call (*function_ptr)(a, b, c). */
libxsmm_smmfunction libxsmm_smmdispatch(int m, int n, int k,
  const int* lda, const int* ldb, const int* ldc,
  const float* alpha, const float* beta,
  const int* flags, const int* prefetch);
/** If non-zero function pointer is returned, call (*function_ptr)(a, b, c). */
libxsmm_dmmfunction libxsmm_dmmdispatch(int m, int n, int k,
  const int* lda, const int* ldb, const int* ldc,
  const double* alpha, const double* beta,
  const int* flags, const int* prefetch);
```

In C++, `libxsmm_mmfunction<type>` can be used to instantiate a functor rather than making a distinction between numeric types per type-prefix (see samples/smm/specialized.cpp).

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

Similarly in FORTRAN (see samples/smm/smm.f), a generic interface (`libxsmm_mmdispatch`) can be used to dispatch a `LIBXSMM_?MMFUNCTION`, and the encapsulated PROCEDURE POINTER can be called via `libxsmm_call`. Beside of dispatching code, one can also call any statically generated kernels (e.g., `libxsmm_dmm_4_4_4`) using the prototype functions included with the FORTRAN and C/C++ interface.

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

In case of batched SMMs, it can be beneficial to supply "next locations" such that the upcoming operands are prefetched ahead of time. 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`).

Call Wrapper

Since the library is binary compatible with existing GEMM calls (BLAS), these 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. An application which is linked statically against BLAS requires to wrap the 'sgemm_' and the 'dgemm_' symbol (an alternative is to wrap only 'dgemm_'), and a special build of the libxsmm(ext) library is required (`make WRAP=1` to wrap SGEMM and DGEMM, or `make WRAP=2` to wrap only DGEMM):
   ```
   gcc [...] -Wl,--wrap=sgemm_,--wrap=dgemm_ /path/to/libxsmmext.a /path/to/libxsmm.a /path/to/your_regular_blas.a
   ```
   - Relinking the application as shown above can often be accomplished by copying, pasting, modifying the linker command, and then re-invoking the modified link step. This linker command may appear as console output of the application's "make" command (or a similar build system).
   - 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), and it may not work under OS X (Compiler 6.1 or earlier, later versions have not been tested).
2. 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 LIBXSMM (`make STATIC=0`) can be used to intercept GEMM calls:
   ```
   LD_PRELOAD=/path/to/libxsmmext.so ./myapplication
   ```

NOTE: Using the same multiplication kernel in a consecutive fashion (batch-processing) allows to extract higher performance, when using LIBXSMM's native programming interface.

Deep Neural Networks

To achieve best performance with small convolutions for CNN on SIMD architectures, a specific data layout must be used. As this layout depends on several architectural parameters, the goal of LIBXSMM's interface is to hide this complexity from the user by providing copy-in and copy-out routines. This happens using opaque data types, which themselves are later bound to a convolution operation.

The interface is available for C. There is a collection of benchmark-style code samples (samples/dnn) with focus on Convolutional Deep Neural Networks (DNNs). Further, an example performing a single image convolution is provided as well. The general concept of the interface is circled around a few types: `libxsmm_dnn_layer`, `libxsmm_dnn_buffer`, `libxsmm_dnn_bias`, and `libxsmm_dnn_filter`. A handle of such a type is always setup by calling a create-function.

```c
/** Simplified LIBXSMM types which are needed to create a handle. */

/** Structure which describes the input and output of data (DNN). */
typedef struct libxsmm_dnn_conv_desc {
  int N;                                    /* number of images in mini-batch */
  int C;                                    /* number of input feature maps */
  int H;                                    /* height of input image */
  int W;                                    /* width of input image */
  int K;                                    /* number of output feature maps */
  int R;                                    /* height of filter kernel */
  int S;                                    /* width of filter kernel */
  int u;                                    /* vertical stride */
  int v;                                    /* horizontal stride */
  int pad_h;                                /* height of logical rim padding to input
                                               for adjusting output height */
  int pad_w;                                /* width of logical rim padding to input
                                               for adjusting output width */
  int pad_h_in;                             /* height of zero-padding in input buffer,
                                               must equal to pad_h for direct conv */
  int pad_w_in;                             /* width of zero-padding in input buffer,
                                               must equal to pad_w for direct conv */
  int pad_h_out;                            /* height of zero-padding in output buffer */
  int pad_w_out;                            /* width of zero-padding in output buffer */
  int threads;                              /* number of threads to use when running
                                               convolution */
  libxsmm_dnn_datatype datatype;            /* datatypes use for all input and outputs */
  libxsmm_dnn_tensor_format buffer_format;  /* format which is for buffer buffers */
  libxsmm_dnn_tensor_format filter_format;  /* format which is for filter buffers */
  libxsmm_dnn_conv_algo algo;               /* convolution algorithm used */
  libxsmm_dnn_conv_option options;          /* additional options */
  libxsmm_dnn_conv_fuse_op fuse_ops;        /* used ops into convolutions */
} libxsmm_dnn_conv_desc;

/** Type of algorithm used for convolutions. */
typedef enum libxsmm_dnn_conv_algo {
  /** let the library decide */
  LIBXSMM_DNN_CONV_ALGO_AUTO,   /* ignored for now */
```

```
  /** direct convolution. */
  LIBXSMM_DNN_CONV_ALGO_DIRECT
} libxsmm_dnn_conv_algo;

/** Denotes the element/pixel type of an image/channel. */
typedef enum libxsmm_dnn_datatype {
  LIBXSMM_DNN_DATATYPE_F32,
  LIBXSMM_DNN_DATATYPE_I32,
  LIBXSMM_DNN_DATATYPE_I16,
  LIBXSMM_DNN_DATATYPE_I8
} libxsmm_dnn_datatype;

libxsmm_dnn_layer* libxsmm_dnn_create_conv_layer(
  libxsmm_dnn_conv_desc conv_desc, libxsmm_dnn_err_t* status);
libxsmm_dnn_err_t libxsmm_dnn_destroy_conv_layer(
  const libxsmm_dnn_layer* handle);
```

A sample call looks like (without error checks):

```
/* declare LIBXSMM variables */
libxsmm_dnn_conv_desc conv_desc;
libxsmm_dnn_err_t status;
libxsmm_dnn_layer* handle;
/* setting conv_desc values.... */
conv_desc.N = ...
/* create handle */
handle = libxsmm_dnn_create_conv_layer(conv_desc, &status);
```

Next activation and filter buffers need to be linked, initialized and bound to the handle. Afterwards the convolution can be executed in a threading environment of choice (error checks are omitted for brevity):

```
float *input, *output, *filter;
libxsmm_dnn_buffer* libxsmm_reg_input;
libxsmm_dnn_buffer* libxsmm_reg_output;
libxsmm_dnn_filter* libxsmm_reg_filter;

/* allocate data */
input = (float*)libxsmm_aligned_malloc(...);
output = ...;

/* link data to buffers */
libxsmm_reg_input = libxsmm_dnn_link_buffer(  libxsmm_handle, LIBXSMM_DNN_INPUT, input,
                                              LIBXSMM_DNN_TENSOR_FORMAT_LIBXSMM_PTR, &status);
libxsmm_reg_output = libxsmm_dnn_link_buffer( libxsmm_handle, LIBXSMM_DNN_OUTPUT, output,
                                              LIBXSMM_DNN_TENSOR_FORMAT_LIBXSMM_PTR, &status);
libxsmm_reg_filter = libxsmm_dnn_link_filter( libxsmm_handle, LIBXSMM_DNN_FILTER, filter,
                                              LIBXSMM_DNN_TENSOR_FORMAT_LIBXSMM_PTR, &status);

/* copy in data to LIBXSMM format: naive format is: */
/* (mini-batch)(number-featuremaps)(featuremap-height)(featuremap-width) for layers, */
/* and the naive format for filters is: */
/* (number-output-featuremaps)(number-input-featuremaps)(kernel-height)(kernel-width) */
libxsmm_dnn_copyin_buffer(libxsmm_reg_input, (void*)naive_input, LIBXSMM_DNN_TENSOR_FORMAT_NCHW);
libxsmm_dnn_zero_buffer(libxsmm_reg_output);
libxsmm_dnn_copyin_filter(libxsmm_reg_filter, (void*)naive_filter, LIBXSMM_DNN_TENSOR_FORMAT_KCRS);

/* bind layer to handle */
libxsmm_dnn_bind_input_buffer(libxsmm_handle, libxsmm_reg_input, LIBXSMM_DNN_REGULAR_INPUT);
libxsmm_dnn_bind_output_buffer(libxsmm_handle, libxsmm_reg_output, LIBXSMM_DNN_REGULAR_OUTPUT);
libxsmm_dnn_bind_filter(libxsmm_handle, libxsmm_reg_filter, LIBXSMM_DNN_REGULAR_FILTER);

/* allocate and bind scratch */
scratch = libxsmm_aligned_scratch(libxsmm_dnn_get_scratch_size(
  libxsmm_handle, LIBXSMM_DNN_COMPUTE_KIND_FWD, &status), 2097152);
libxsmm_dnn_bind_scratch(libxsmm_handle, LIBXSMM_DNN_COMPUTE_KIND_FWD, scratch);

/* run the convolution */
#pragma omp parallel
{
  libxsmm_dnn_convolve_st(libxsmm_handle, LIBXSMM_DNN_CONV_KIND_FWD, 0,
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
/* copy out data */
libxsmm_dnn_copyout_buffer(libxsmm_output, (void*)naive_libxsmm_output,
  LIBXSMM_DNN_TENSOR_FORMAT_NCHW);

/* clean up */
libxsmm_dnn_release_scratch(...);
libxsmm_dnn_release_buffer(...);
...
libxsmm_dnn_destroy_buffer(...);
...
libxsmm_dnn_destroy_conv_layer(...);
```

Service Functions

Getting and Setting the 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**", "sse2"
- LIBXSMM_X86_SSE3: "**sse3**", "sse"
- LIBXSMM_X86_SSE4: "**wsm**", "nhm", "sse4", "sse4_2", "sse4.2"
- LIBXSMM_X86_AVX: "**snb**", "avx"
- LIBXSMM_X86_AVX2: "**hsw**", "avx2"
- LIBXSMM_X86_AVX512: "**avx3**", "avx512"
- LIBXSMM_X86_AVX512_MIC: "**knl**", "mic"
- LIBXSMM_X86_AVX512_KNM: "**knm**"
- LIBXSMM_X86_AVX512_CORE: "**skx**", "skl"

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).

Getting and Setting the Verbosity

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 various code regions. The timer is based on monotonic clock tick, which uses a platform-specific resolution. The counter may rely on the time stamp counter instruction (RDTSC), but this is not necessarily counting CPU cycles due to varying CPU clock speed (Turbo Boost), different clock domains (e.g., depending on the instructions executed), and other reasons (which are out of scope in this context).

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

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).

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: Only `libxsmm_free` is supported to deallocate the memory.

```
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* memory, libxsmm_malloc_info* info);
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);
```

There are currently no claims on the properties of the default memory allocation (except when tuning the thread scalability). In contrast, the scratch memory allocation is very effective and delivers a decent speedup over subsequent regular memory allocations. In contrast to the default allocation technique, the scratch memory establishes a watermark for repeatedly allocated and deallocated buffers. The scratch memory domain is (arbitrarily) limited to 2 GB of memory, but it is possible to set a different Byte-limit (available per libxsmm_malloc.h, and also per environment variable LIBXSMM_SCRATCH_LIMIT with optional "k|K", "m|M", and "g|G" units).

```
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, and supports only a limited number of pools for buffers of different life-time. 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)
```

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!

Performance Analysis

Intel VTune Amplifier

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 Amplifier. 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 /path/to/vtune_amplifier/amplxe-vars.sh
make SYM=1
```

Above, the root directory is automatically determined from the environment (VTUNE_AMPLIFIER_*_DIR). This variable is present after source'ing the Intel VTune environment, 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:

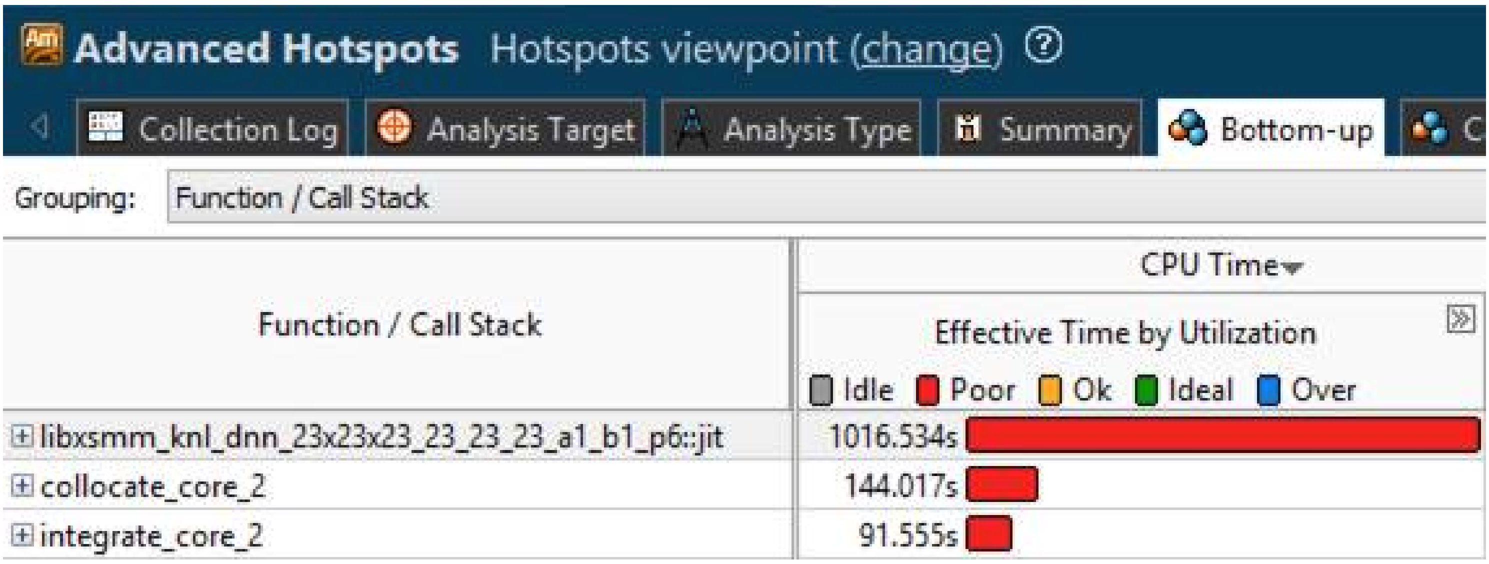
```
amplxe-cl -r result-directory -data-limit 0 -collect advanced-hotspots \
          -knob collection-detail=stack-sampling -- ./myapplication
```

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 'amplxe-cl -r result-directory \
                     -data-limit 0 -collect advanced-hotspots \
                     -knob collection-detail=stack-sampling:4=exclusive'
```

The `:4=exclusive` is related to mpirun's gtool arguments and unrelated to VTune's command line syntax; such argument(s) need to appear at the end of the gtool-string. For instance, the shown command line selects the 4th rank (otherwise all ranks are sampled) along with "exclusive" usage of the performance monitoring unit (PMU) such that only one event-collector runs for all ranks.

Intel VTune Amplifier 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`. For example, TensorFlow with LIBXSMM and Intel VTune Amplifier may use this way to gain insight into LIBXSMM's JIT-code (see here).

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

Tuning for Specific Targets

Specifying a code path is not really 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, the 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.

To improve thread-scalability of the default memory allocation domain, the library can rely on Intel TBB, which is discovered per TBBROOT environment variable: `make TBB=1`.

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.

To minimize the probability of key collisions (code cache), the preferred precision of the statically generated code can be selected:

```
make PRECISION=2
```

The default preference is to generate and register both single and double-precision code, and therefore no space in the dispatch table is saved (PRECISION=0). Specifying PRECISION=1|2 is only generating and registering either single-precision or double-precision code.

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 using various techniques such as using 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 might 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 no support for the Intel SSE (Intel Xeon 5500/5600 series) and IMCI (Intel Xeon Phi coprocessor code-named Knights Corner) instruction set extensions. However, statically generated SSE-kernels can be leveraged without disabling support for JIT'ting AVX kernels.
3. There is no support for the Windows calling convention (only kernels with PREFETCH=0 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). 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). The driver program is built as part of LIBXSMM's build process (when requesting static code generation), but also available via a separate build target:

```
make generator
bin/libxsmm_gemm_generator
```

The code generator driver program accepts the following arguments:

1. dense/dense_asm/sparse (dense creates C code, dense_asm creates ASM)
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 when 1. is "sparse" indicates A is sparse)
8. LDB (0 when 1. is "sparse" indicates B is sparse)
9. LDC parameter
10. alpha (1)
11. beta (0 or 1)
12. Alignment override for A (1 auto, 0 no alignment)
13. Alignment override for C (1 auto, 0 no alignment)
14. Architecture (noarch, wsm, snb, hsw, knc, knl, knm, skx)
15. Prefetch strategy, see below enumeration (dense/dense_asm only)
16. single precision (SP), or double precision (DP)
17. CSC file (just required when 1. is "sparse"). Matrix market format.

The prefetch strategy can be:

1. "nopf": no prefetching at all, just 3 inputs (A, B, C)
2. "pfsigonly": just prefetching signature, 6 inputs (A, B, C, A', B', C')
3. "BL2viaC": uses accesses to C to prefetch B'
4. "curAL2": prefetches current A ahead in the kernel
5. "curAL2_BL2viaC": combines curAL2 and BL2viaC
6. "AL2": uses accesses to A to prefetch A'
7. "AL2_BL2viaC": combines AL2 and BL2viaC
8. "AL2jpst": aggressive A' prefetch of first rows without any structure
9. "AL2jpst_BL2viaC": combines AL2jpst and BL2viaC
10. "AL1": prefetch A' into L1 via accesses to A
11. "BL1": prefetch B' into L1 via accesses to B
12. "CL1": prefetch C' into L1 via accesses to C
13. "AL1_BL1": prefetch A' and B' into L1
14. "BL1_CL1": prefetch B' and C' into L1
15. "AL1_CL1": prefetch A' and C' into L1
16. "AL1_BL1_CL1": prefetch A', B', and C' into L1

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.