LIBXSMM

LIBXSMM is a library for small dense and small sparse matrix-matrix multiplications targeting Intel Architecture (x86). The library is generating code 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 ("KNL") and future Intel Xeon processors. Historically the library was solely targeting the Intel Many Integrated Core Architecture "MIC") using intrinsic functions, meanwhile optimized assembly code is targeting all aforementioned instruction set extensions (static code generation), and Just-In-Time (JIT) code generation is targeting Intel AVX and beyond.

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.

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 $(M\ N\ K)^{1/3} <= 80$ (which illustrates that non-square matrices or even "tall and skinny" shapes are covered as well). However the code generator only generates code up to the specified threshold. Raising the threshold may not only generate excessive amounts of code (due to unrolling in M and K dimension), but also miss to implement a tiling scheme to effectively utilize the cache hierarchy. For problem sizes above the configurable threshold, LIBXSMM is falling back to BLAS.

What about "medium-sized" matrix multiplication? A more recent addition are GEMM routines which are parallelized using OpenMP (`libxsmm_omp_?gemm`). These routines are leveraging the same specialized kernel routines as the small matrix multiplications, in-memory code generation (JIT), and automatic code/parameter dispatch but implementing a tile-based multiplication scheme i.e., a scheme suitable for larger problem sizes.

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 link against Intel MKL 11.2 (or higher), set the environment variable MKL_VERBOSE=1, and run the application using a representative workload (`env MKL_VERBOSE=1 ./workload > verbose.txt`). The collected output is the starting point for evaluating the problem sizes as imposed by the workload (`grep -a "MKL_VERBOSE DGEMM(N,N" verbose.txt | cut -d'(' -f2 | cut -d, -f3-5`).

Interface

The interface of the library is *generated* according to the Build Instructions, and 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.

In order to initialize the dispatch-table or other internal resources, one may call an explicit initialization routine in order to avoid lazy initialization overhead when calling LIBXSMM for the first time. The library deallocates internal resources automatically, but also provides a companion to the aforementioned initialization (finalize).

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

To perform the dense matrix-matrix multiplication $C_{m\,x\,n} = alpha \cdot A_{m\,x\,k} \cdot B_{k\,x\,n} + beta \cdot C_{m\,x\,n}$, the full-blown GEMM interface can be treated with "default arguments" (which is deviating from the LAPACK/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 and in particular 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 are allowed to 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 LAPACK/BLAS interface) by allowing to omit arguments where the C/C++ interface is 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` instead of `libxsmm_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. The associated routines are by default opening a parallel region internally, however participating on an already opened parallel region (without relying on nested parallelism) is also possible by using the environment variable LIBXSMM_OMP=1 (1: sequential, and 2: parallelized/default). The actual parallelism is based on "classic" OpenMP by default (thread-based), but can be adjusted to OpenMP 3.0 task-based parallelism (environment variable LIBXSMM_TASKS=1). At least the latter parallelization is dynamically scheduled. Please note that these routines are hosted by the extension library (libxsmmext) keeping the main library agnostic with respect to a particular threading runtime. For the same reason, the FORTRAN interface does not allow optional arguments for this pair of routines.

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

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

```
/** 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);
```

A variety of overloaded function signatures is provided allowing to omit arguments not deviating from the configured defaults. In C++, a type `libxsmm_mmfunction<type>` can be used to instantiate a functor rather than making a distinction for the numeric type in `libxsmm_?mmdispatch`. Similarly in FORTRAN, when calling the generic interface (`libxsmm_mmdispatch`) the given `LIBXSMM_?MMFUNCTION` is dispatched such that `libxsmm_call` can be used to actually perform the function call using the PROCEDURE POINTER wrapped by `LIBXSMM_?MMFUNCTION`. Beside of dispatching code, one can also call a specific kernel (e.g., `libxsmm_dmm_4_4_4`) using the prototype functions included for statically generated kernels.

Build Instructions

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`). Of course both methods can be mixed, however the second method may requires to supply the `-e` flag. Please note that the CXX, CC, and FC keys are handled such that they are taken into account 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
```

By default, only the non-coprocessor targets are built (OFFLOAD=0 and MIC=0). In general, the subfolders of the 'lib' directory are separating the build targets where the 'mic' folder is containing the native library (MIC=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 MIC=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 for particular matrix sizes (M, N, and K values):

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

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

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

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

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

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

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

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

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

Of course, both mechanisms (M/N/K and MNK based) can be combined using the same command line (make). Static optimization and JIT can also be combined (no need to turn off the JIT backend).

Testing the generated cases can be accomplished by capturing the console output of the cp2k code sample:

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

The recorded output file can be further evaluated (see also cp2k-test.sh). For example:

```
grep "diff" samples/cp2k/cp2k-perf.txt | grep -v "diff=0.000"
```

NOTE: by default, a combination of 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=`). 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' and 'libxsmm' (this order),
- By including the interface 'libxsmm.f', and by linking solely against 'libxsmm', or by
- Optionally declaring `libxsmm_?gemm` (BLAS signature), and by linking against 'libxsmm'.

At the expense of a limited functionality (`libxsmm_?gemm`), the latter method also works with FORTRAN 77 (otherwise the FORTRAN 2003 standard is necessary).

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 SSE3 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 only real question remains for PREFETCH=0 (default) versus PREFETCH=1. 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
```

NOTE: the library is agnostic with respect to the threading-runtime, and enabling OpenMP (OMP=1) when building the library is a non-default option (untested). The libraries are also agnostic with respect to the selected LAPACK-/BLAS library (if supported by OS and link model), and therefore linking GEMM routines when building the library (by supplying BLAS=1|2) may prevent a user from deciding at the time of linking the actual application.

Running

Call Wrapper

Since the library is binary compatible with existing GEMM calls (LAPACK/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 LAPACK/BLAS. Currently this only works for the Linux OS (not validated under OS X), and it is also not sufficient to rely on a GNU tool chain under Microsoft Windows. Of course, using LIBXSMM's programming interface when performing the same multiplication multiple time in a consecutive fashion (batch-processing) allows to extract higher performance. However, using the call wrapper might motivate to make use of the LIBXSMM API.

There are two cases to consider: (1) an application which is linking statically against LAPACK/BLAS, and (2) an application which is dynamically linked against LAPACK/BLAS. The first case requires to wrap the `sgemm_` or `dgemm_` symbol by making `-Wl,--wrap=*gemm_ /path/to/libxsmm.a` (star to be replaced) part of the link-line and then relinking the application:

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

Relinking the application is often accomplished by copying, pasting, and modifying the linker command as shown when running "make" (or a similar build system), and then just re-invoking the modified link step. Please note that this first case is also working for an applications which is dynamically linked against LAPACK/BLAS. The static link-time wrapper technique in general may only work with a GCC-compatible tool chain (GNU Binutils: `ld`, or `ld` via compiler-driver), and it has been tested with GNU GCC, Intel Compiler, and Clang. The latter unfortunately does not include Compiler 6.1 or earlier under OS X, and it has not been tested with a later version of this tool chain.

If an application is dynamically linked against LAPACK/BLAS, the unmodified application allows for intercepting these calls at startup time (runtime) by using the LD_PRELOAD mechanism:

```
LD_PRELOAD=/path/to/libxsmmext.so ./myapplication
```

This case obviously requires to build a shared library of LIBXSMM:

```
make STATIC=0
```

The behavior of the intercepted GEMM routines (statically wrapped or via LD_PRELOAD) can be controlled using the environment variable LIBXSMM_GEMM i.e., 0: sequential below-threshold routine without OpenMP but with fallback to BLAS (default), 1: OpenMP-parallelized but without internal parallel region, and 2: OpenMP-parallelized with internal parallel region. However in case of the static wrapper, it is required to link against `libxsmmext` and `libxsmm` (in this order; see link-line above).

```
LIBXSMM_GEMM=2 ./myapplication
```

Please note that calling SGEMM is more sensitive to dispatch-overhead when compared to multiplying the same matrix sizes in double-precision. In case of single-precision, an approach of using the call wrapper is often not able to show an advantage if not regressing with respect to performance (therefore SGEMM is likely asking for making use of the API). In contrast, the double-precision case can show up to two times the performance of a typical LAPACK/BLAS performance (and more when using the API for processing batches).

Verbose Mode

The verbose mode allows for an insight into the code dispatch mechanism by receiving a small tabulated statistics 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       7      7      0      0
    14..23       0      0      0      0
    24..80       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 ($\mathrm{MNK}^{1/3} <= 13$), medium-sized kernels ($13 < \mathrm{MNK}^{1/3} <= 23$), and larger kernels ($23 < \mathrm{MNK}^{1/3} <= 80$; the actual upper bound depends on LIBXSMM_MAX_MNK as selected at compile-time). Keep in mind, that "larger" is supposedly still fairly 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 JIT and STA counters distinct the aforementioned event into dynamically (JIT) and statically (STA) generated code but also count only actually registered kernels. In case the capacity ($\mathrm{O}(n) = 10^5$) of the code registry is exhausted, no more kernels can be registered although further attempts are not prevented. Registering many kernels ($\mathrm{O}(n) = 10^3$) may ramp the number of hash key collisions (COL), which can degrade performance. The latter is prevented if the small thread-local cache is effectively utilized.

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 implies standard error (`stderr`) output in case of an error/warning condition inside of the library. Towards developing an application which is successfully using the library, being able to trace library calls might be useful as well (can be combined with DBG=1 or OPT=0):

```
make TRACE=1
```

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

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

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

higher levels of the call-tree with *i=1* being the default (level zero is the highest at the same level as the main function). The last argument is taking the number of inclusive call levels with *n=-1* being the default (signifying no filter).

Although the `ltrace` (Linux utility) provides similar insight, the trace facility might be useful due to the aforementioned 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. No facility of the library (other than DBG or TRACE/LIBXSMM_TRACE) is performing visible (console) or other non-private I/O (files).

Performance

Profiling

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 as supported by 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) triggers using VTune's JIT Profiling API:

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

The root directory is automatically determined from an environment variable (VTUNE_AMPLIFIER_*_DIR), which 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 actually not 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 might 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 the latter can be achieved by adding

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

to the `mpirun` command line. Please notice the `:4=exclusive` which is unrelated to VTune's command line syntax but related to mpirun's gtool arguments; these arguments 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 the aforementioned JIT Profiling API. For instance "libxsmm_hsw_dnn_23x23x23_23_23_23_a1_b1_p0::jit" encodes an Intel AVX2 ("hsw") double-precision kernel ("d") which is multiplying matrices without transposing them ("nn"). The rest of the name encodes M=N=K=LDA=LDB=LDC=23, Alpha=Beta=1.0 (all similar to GEMM), and no prefetch strategy ("p0").

Tuning

Specifying a particular 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 according to 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 kernel which is picking up more than three arguments and actually picking up garbage data is disabling the hardware prefetcher (due to software prefetches) followed by a misleading prefetch location plus an eventual page fault due to an out-of-bounds (garbage-)location.

Further, the generated interface of the library also encodes the parameters 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 in order 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. LAPACK/BLAS library call (fallback).

Both levels are accessible directly (see Interface section) allowing 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 calling into the LAPACK/BLAS library alongside of LIBXSMM. 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.

In order 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 calculating a CRC32 checksum, avoiding costly synchronization (needed for thread-safety) until it is ultimately known that the requested kernel is not yet JIT'ted, and also 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.

JIT Backend

There might be situations in which it is up-front not clear which problem sizes will be needed when running an application. In order 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 according to 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. In order 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|skx). Please note that LIBXSMM_TARGET cannot enable the JIT backend if it was disabled at build time (JIT=0).

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

Note: Modern Linux kernels are supporting transparent huge pages (THP). LIBXSMM is sanitizing this feature when setting the permissions for pages holding the executable code. However, we measured up to 30% slowdown when running JIT'ted code in cases where THP decided to deliver a huge page. For systems with Linux kernel 2.6.38 (or later) THP will be automatically disabled for the `mmap`'ed regions (using `madvise`).

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 or 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)
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. “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
8. “AL2jpst”: aggressive A’ prefetch of first rows without any structure
9. “AL2jpst_BL2viaC”: combines AL2jpst 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.

Results

The LIBXSMM repository provides an orphaned branch “results” which is collecting collateral material such as measured performance results along with explanatory figures. The results can be found at https://github.com/hfp/libxsmm/tree/results#libxsmm-results.

Please note that comparing performance results depends on whether or not streaming the operands of the matrix multiplication. For example, running a matrix multiplication code many time with all operands covered by the L1 cache may have an emphasis towards an implementation which actually performs worse for the real workload (if this real workload needs to stream some or all operands from the main memory).

Applications

[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 was integrating 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, in particular 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/Nek5000/NekBox**: NekBox is a version of the highly scalable and portable spectral element Nek5000 code which is specialized for box geometries, and intended for prototyping new methods as well as leveraging FORTRAN beyond the FORTRAN 77 standard. LIBXSMM provides optimized kernels aiming to conveniently substitute the MXM_STD code.

References

[1] **http://sc16.supercomputing.org**: LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation (accepted full paper, not yet published). 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 two-page extended abstract). SC’15: The International Conference for High Performance Computing, Networking, Storage and Analysis, Austin (Texas).