LIBXSMM

Library for small matrix-matrix multiplications targeting Intel Architecture (x86). The initial version of the library was targeting the Intel Xeon Phi coprocessor (an instance of the Intel Many Integrated Core Architecture "MIC") particularly by using KNC intrinsic functions (called KNCni or IMCI). Today, the library reaches the Many Integrated Core Architecture as well as other hardware which is capable of executing Intel Advanced Vector Extensions 512 (Intel AVX-512). Please also have a look at the collection of upcoming enhancements.

The library provides a sophisticated dispatch mechanism (see More Details) which is also targeting other instruction sets (beside of the Intrinsic code path). The library can be also compiled to "MIC native code" which is able to run self-hosted as well as in an offloaded code region (via a FORTRAN directive or via C/C++ preprocessor pragma). The prerequisite for offloading the code is to compile it to position-independent (PIC) code even when building a static library.

Performance: the presented code is by no means "optimal" or "best-performing" - it just uses Intrinsics. In fact, a well-optimizing compiler may arrange better code compared to what is laid out via the library's Python scripts. The latter can be exploited by just relying on the "inlinable code" and by not generating specialized functions.

Interface

The interface of the library is *generated* according to the Build Instructions (therefore the header file 'include/libxsmm.h' is **not** stored in the code repository). The generated interface also defines certain preprocessor symbols to store the properties the library was built for.

To perform the matrix-matrix multiplication cm x $n = cm$ x $n + am$ x k * bk x n, one of the following interfaces can be used:

```
/** If non-zero function pointer is returned, call (*function)(M, N, K). */
libxsmm_smm_function libxsmm_smm_dispatch(int m, int n, int k);
libxsmm_dmm_function libxsmm_dmm_dispatch(int m, int n, int k);
/** Automatically dispatched matrix-matrix multiplication. */
void libxsmm_smm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm_dmm(int m, int n, int k, const double* a, const double* b, double* c);
/** Non-dispatched matrix-matrix multiplication using inline code. */
void libxsmm_simm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm_dimm(int m, int n, int k, const double* a, const double* b, double* c);
/** Matrix-matrix multiplication using BLAS. */
void libxsmm_sblasmm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm_dblasmm(int m, int n, int k, const double* a, const double* b, double* c);
```

With C++ function overloading, the library allows to omit the 's' and 'd' denoting the numeric type in the above C interface. Further, a type 'libxsmm_mm_dispatch<*type*>' can be used to instantiate a functor rather than making a distinction for the numeric type in 'libxsmm_?mm_dispatch'.

Build Instructions

To compile the library run:

```
make
```

The interface is produced inside of the 'include' directory. The library archives are produced inside of the 'lib' directory with the 'mic' subdirectory containing the native library and the 'intel64' folder storing the hybrid archive containing host and MIC code.

To remove intermediate files use:

```
make clean
```

or to remove all generated files including the interface and library archive files:

```
make realclean
```

The usual `make install` is simply a shortcut for `make; make clean`.

The library can be configured to accept row-major (default) or column-major order matrices. Change the variable ROW_MAJOR inside of the Makefile (0 for column-major, and row-major order otherwise), or build the library in the following way to configure the column-major format:

```
make ROW_MAJOR=0
```

To specialize LIBXSMM for certain matrix sizes (M, N, and K values), one can adjust the variables inside of the Makefile or for example build in the following way:

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

The above example generates the following set of (M,N,K) values:

```
(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 when generating the indices. Moreover, an empty index set resolves to the next non-empty "upper" index set while not participating anymore in the loop-nest relationship. Here is an example of generating multiplication routines for small square matrices:

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

Performance

Tuning

The build system allows to conveniently select the target system using an AVX flag when invoking "make". The default is to generate code according to the feature bits of the host system running the compiler. The device-side defaults to "MIC" targeting the Intel Xeon Phi family of coprocessors ("KNC"). However beside of AVX=1 and AVX=2 (with FMA), the following switch targets the Intel Knights Landing processor family ("KNL") and future Intel Xeon processors using Intel AVX-512 foundational instructions (AVX-512F):

```
make AVX=3
```

A future version of the library may support an auto-tuning stage when generating the code (to find M,N,K-combinations for the Intrinsic code path which are beneficial compared to inlined compiler-generated code). Auto-tuning the compiler code generation using a profile-guided optimization may be another option to be incorporated into the build system (Makefile).

The library supports generating code using an "implicitly aligned leading dimension" for the destination matrix of a multiplication. The latter is enabling unaligned store instructions, and also hints the inlined code accordingly. The client code may be arranged at compile-time (preprocessor) by checking the build parameters of the library. Aligned store instructions imply a leading dimension which is a multiple of the default alignment:

```
make ALIGNED_STORES=1
```

The default alignment (ALIGNMENT=64) as well as a non-default alignment for the store instructions (ALIGNED_STORES=n) can be specified when invoking "make". The "implicitly aligned leading dimension" optimization is not expected to have a big impact due to the relatively low amount of store instructions in the instruction mix. In contrast, supporting an "implicitly aligned leading dimension" for loading the input matrices is supposed to make a bigger impact, however this is not yet implemented. There are two reasons: (1) aligning a batch of input matrices implies usually larger code changes for the client code whereas accumulating into a local temporary destination matrix is a relatively minor change, and (2) the AVX-512 capable hardware supports unaligned load/store instructions. For further details, one may have a look at the sample code.

Auto-dispatch

The function 'libxsmm_?mm_dispatch' helps amortizing the cost of the dispatch when multiple calls with the same M, N, and K are needed. In contrast, the automatic code dispatch uses three levels:

1. Specialized routine,
2. Inlined code, and
3. BLAS library call.

All three levels are accessible directly (see Interface) in order to allow a customized code dispatch. The level 2 and 3 may be supplied by the Intel Math Kernel Library (Intel MKL) 11.2 DIRECT CALL feature. Beside of the generic interface, one can call a specific kernel e.g., 'libxsmm_dmm_4_4_4' multiplying 4x4 matrices.

Further, a preprocessor symbol denotes the largest problem size (M x N x K) that belongs to level (1) and (2), and therefore determines if a matrix-matrix multiplication falls back to level (3) of calling the BLAS library linked with the library. This threshold can be configured using for example:

```
make THRESHOLD=$((24 * 24 * 24))
```

The maximum of the given threshold and the largest requested specialization refines the value of the threshold. If a problem size falls below the threshold, dispatching the code requires to figure out whether a specialized routine exists or not. This is implemented searching a table of the specialized functions in a binary manner. At the expense of storing function pointers for the entire problem space below the threshold, a direct lookup can be implemented as well. This can be configured using for example:

```
make SPARSITY=2
```

A sparsity is calculated at construction time of the library determining whether to use binary search (value above given SPARSITY or in case of SPARSITY <= 1) or direct lookup (raising the given SPARSITY allows to prevent the binary search). However, the overhead of the binary search is negligible which still holds (~2%) when using a relatively slower clocked single core of an Intel Xeon Phi coprocessor.

Applications and References

[**1**] **http://cp2k.org/**: Open Source Molecular Dynamics application. Beside of CP2K's own SMM module, LIBXSMM aims to provide highly optimized assembly kernels.

[**2**] **https://github.com/TUM-I5/GemmCodeGenerator**: Code generator for matrix-matrix multiplications. Due to LIBXSMM's roadmap, this is a related project.

[**3**] **http://software.intel.com/xeonphicatalog**: Intel Xeon Phi Applications and Solutions Catalog.

[**4**] **http://goo.gl/qsnOOf**: Intel 3rd Party Tools and Libraries.