LIBXSMM

Library for small matrix-matrix multiplications targeting Intel Architecture (x86). The library generates code for the following instruction set extensions: Intel SSE3, 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, however meanwhile optimized assembly code is generated for the fore mentioned instruction set extensions.

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. This is accomplished using the variable ROW_MAJOR (0 for column-major, and row-major order otherwise). The following configures the column-major format:

```
make ROW_MAJOR=0
```

To specialize LIBXSMM for certain matrix sizes (M, N, and K values), the build process needs to be adjusted. For example:

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

The above example is generating 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))"
```

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

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

Each group of 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)
```

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), an AVX=3 targets the Intel Knights Landing processor family ("KNL") and future Intel Xeon processors using Intel AVX-512 foundational instructions (AVX-512F):

```
make AVX=3
```

The library supports generating code using an "implicitly aligned leading dimension" for the destination matrix of a multiplication. The latter is enabling aligned store instructions, and also hints the inlinable 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. Inlinable C 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 can be implemented by bisecting a table of all specialized functions (binary search). At the expense of storing function pointers for the entire problem space below the threshold, a direct lookup can be used instead. The actual behavior can be configured using for example:

```
make SPARSITY=2
```

A binary search is implemented when a sparsity (calculated at construction time of the library) is above the given SPARSITY value. Raising the given value prevents generating a binary search (and generates a direct lookup) whereas a value below or equal one is generating the binary search. The overhead of auto-dispatched multiplications based on the binary search becomes negligible with reasonable problem sizes (above ~20x20 matrices), but may be significant for very small auto-dispatched matrix-matrix multiplication.

Results

The generated code does not claim to be "optimal" or "best-performing" - it is just generating source code using Intrinsics or assembly code. Therefore a well-optimizing compiler may arrange better code based on the inlinable C code path when compared to what is laid out by the library's low level code generator.

Implementation

Limitations

Beside of the inlinable code path, the library is currently limited to a single code path which is selected at build time of the library. Without a specific flag (SSE=1, AVX=1|2|3), the assembly code generator emits code for all supported instruction set extensions whereas the Intrinsic code generator (GENASM=0) is actually covering only IMCI (KNCni) and Intel AVX-512F. However, the compiler is picking only one of the generated code paths according to its code generation flags (or according to what is native with respect to the compiler-host). A future version of the library may be including all code paths at build time and allow for runtime-dynamic dispatch of the most suitable code path.

The assembly code generator is currently limited to an M, N, and K combination where N is a multiple of three (Intel SSE3, AVX, and AVX2). For the assembly code targeting Intel Xeon Phi coprocessors however N needs to be less or equal than 30. These limitations will be relaxed in the future, and meanwhile the code generator is backed up in the fore mentioned cases by precompiling the inlinable C code path into the library.

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

Roadmap

Although the library is under development, the published interface is rather stable and may only be extended in future revisions. The following issues are being addressed in upcoming revisions:

- Extended Application Programming Interface (API) supporting sparse matrices and other cases
- Runtime dynamic selection of the most suitable Instruction Set Architecture (ISA dispatch)
- Just-in-Time (JIT) code generation as part of an application (JIT compiler)
- Native FORTRAN interface

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] http://www.seissol.org/: 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. The current usage of LIBXSMM in the context of SeisSol can be found .

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

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

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