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, meanwhile optimized assembly code is targeting all fore mentioned instruction set extensions.

What is a small matrix-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)ˆ(1/3) verbose.txt). The collected output is the starting point for evaluating the problem sizes as imposed by the workload (grep -a "MKL_VERBOSE DGEMM" verbose.txt | 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. To perform the matrix-matrix multiplication cm x n = cm x n + am x k * bk x n, 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++ and FORTRAN function overloading, the library allows to omit the 's' and 'd' prefixes denoting the numeric type in the above C interface. Further, in C++ a type 'libxsmm_mm_dispatch' can be used to instantiate a functor rather than making a distinction for the numeric type in 'libxsmm_?mm_dispatch'.

Build Instructions

To generate the interface inside of the 'include' directory and to build the library, run one of the following commands (by default OFFLOAD=1 implies MIC=1):

```
make
make MIC=1
make OFFLOAD=1
```

By default, only the non-coprocessor target is 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, all libraries are built statically (STATIC=1).

To remove intermediate files (`make install` is a shortcut for `make; make clean`) or to remove all generated files and folders (including the interface and the library archives), run one of the following commands:

```
make clean
make install
make realclean
```

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
```

By default, LIBXSMM is not optimized for particular matrix sizes (M, N, and K values). Specializing the library for certain matrix sizes (and therefore optimizing the performance) can be achieved in the following way:

```
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 (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 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)
```

Testing the generated cases means capturing the console output of the cp2k code sample:

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

The recorded output file can be further evaluated. For example:

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

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 generating all supported host code paths (and letting the compiler pick the one representing the host), specifying a particular code path will not only save some time when generating the code ("printing"), but also enable cross-compilation for a target that is different from the compiler's host: SSE=3 (in fact SSE!=0), AVX=1, AVX=2 (with FMA), and AVX=3 are supported. The latter is targeting 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 C/C++ code accordingly. The client code may be arranged accordingly by checking the build parameters of the library (at compile-time using the preprocessor). 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 exposed by the build system because: (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) today's Advanced Vector Extensions including the AVX-512 capable hardware supports unaligned load/store instructions.

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 LIBXSMM's problem domain.

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 is orchestrating three levels:

1. Specialized routine (implemented in assembly code),
2. Inlinable C/C++ code or optimized FORTRAN code, and
3. BLAS library call.

All three levels are accessible directly (see Interface) allowing to customize the 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 also call a specific kernel e.g., 'libxsmm_dmm_4_4_4' multiplying 4x4 matrices. For the latter, the code generator generates prototypes for all specialized functions (interface).

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 multiplication falls back to level (3) of calling the LAPACK/BLAS library alongside of LIBXSMM. 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. Furthermore, the size of the direct lookup table is limited to 512 KB (currently hardcoded). 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.

Directly invoking the generator backend

In some special, extremely performance-critical situations it might be useful to bypass LIBXSMM's frontend entirely and to directly call into the generated code. This is possible by either invoking the code generator after a regular build process (as described above) or by just building the backend and invoking the generator:

```
make generator
bin/generator
```

The generator application requires following inputs:

1. dense/dense_asm/sparse (dense create C file, dense_asm creates ASM)
2. Filename to append
3. Routine name to be created in 2.
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 (currently only 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. Prefetching mode (just dense & dense_asm, see next list)
15. SP/DP single or double precision
16. CSC file (just required when 1. is "sparse"). Matrix market format.

The prefetch specifier 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

Since this is a complicated, here come some examples:

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

Please note, in samples/generator and samples/seissol examples are shown that directly use LIBXSMM’s generator backend.

Results

The library does not claim to be “optimal” or “best-performing”, and the presented results are modeling certain applications which are not representative “in general”. Instead, information on how to reproduce the results are given for each of the reported cases.

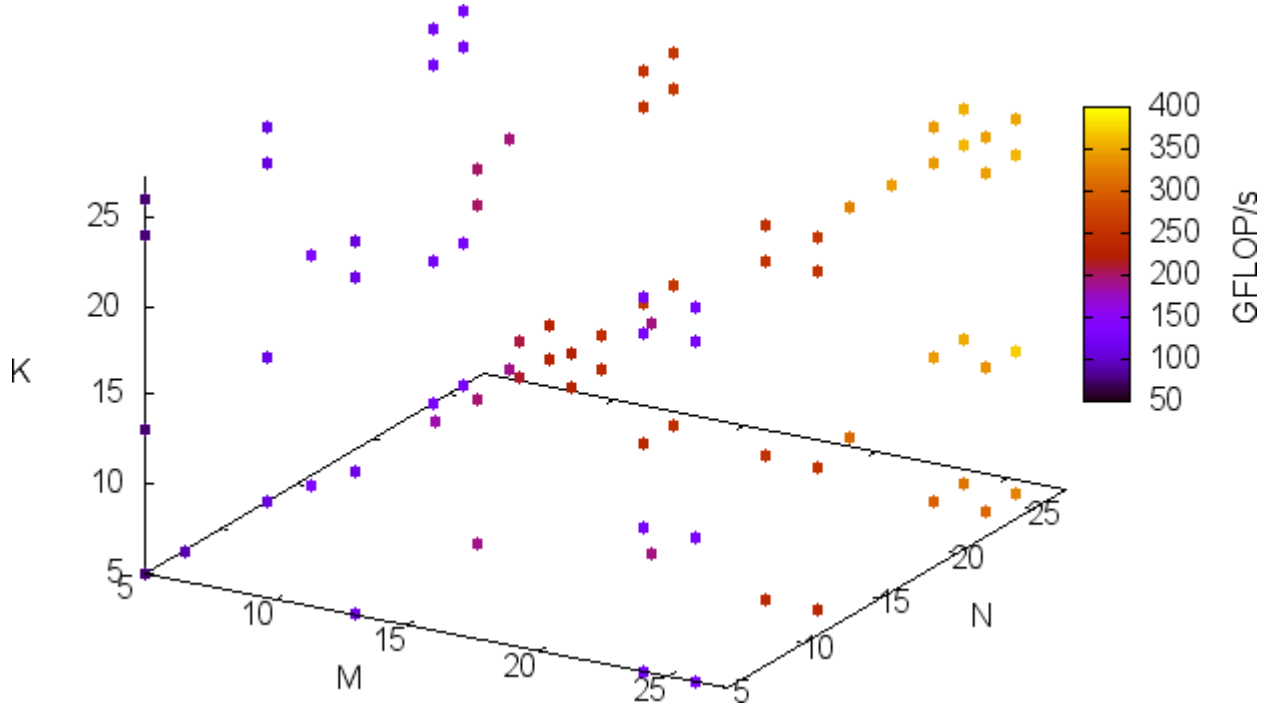

Figure 1: This performance plot for a dual-socket Intel Xeon E5-2699v3 (“Haswell”) shows a “compact selection” (to make the plot visually more appealing) out of 328 specializations as useful for CP2K Open Source Molecular Dynamics [1]. The code has been generated and built by running “./make.sh -cp2k AVX=2 -j”. This and below plots were generated by running “cd samples/cp2k ; ./cp2k-plot.sh specialized cp2k-specialized.png -1”. Please note that larger problem sizes (MNK) carry a higher arithmetic intensity which usually leads to higher performance (less bottlenecked by memory bandwidth).

Implementation

Limitations

Beside of the inlinable C or optimized FORTRAN 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. 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.

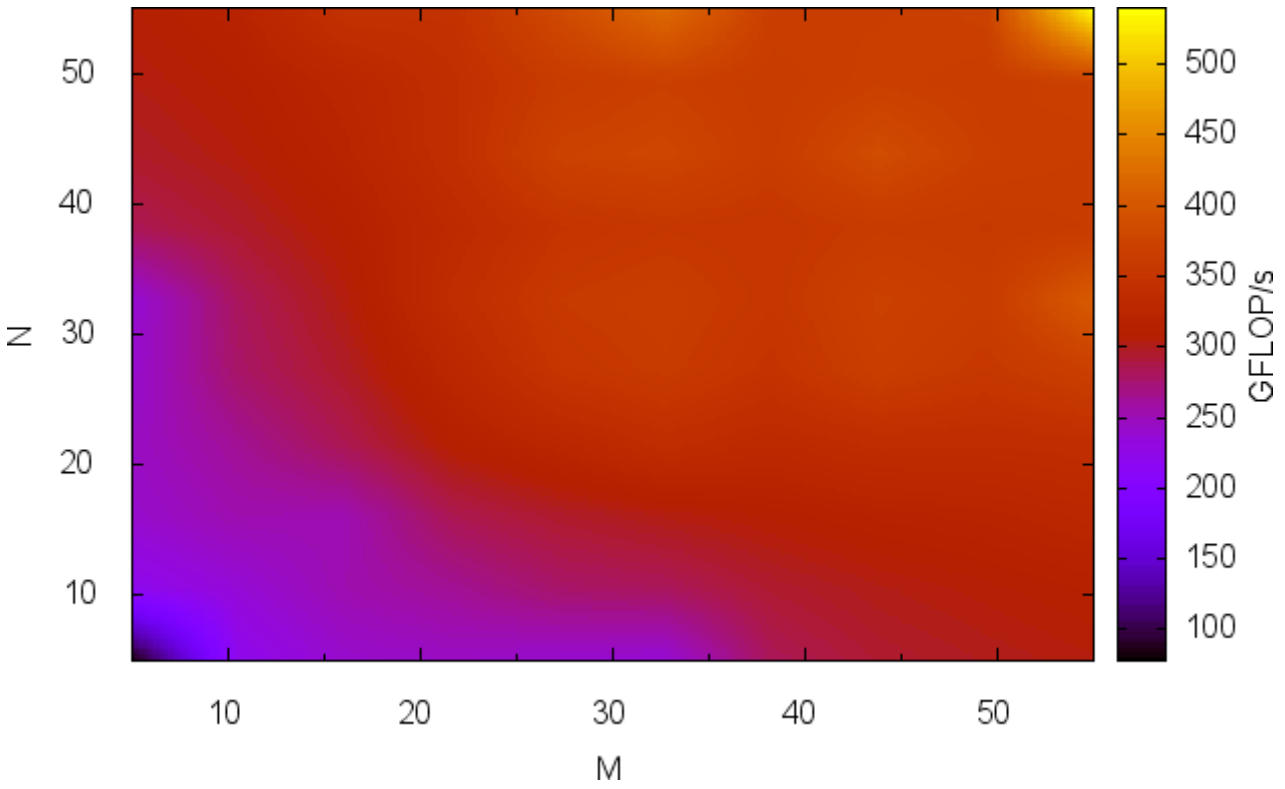

Figure 2: This plot summarizes the performance of the generated kernels by averaging the results over K (and therefore the bar on the right hand side may not show the same maximum when compared to other plots). The performance is well-tuned across the parameter space with no "cold islands", and the lower left "cold" corner is fairly limited. Please refer to the first figure on how to reproduce the results.

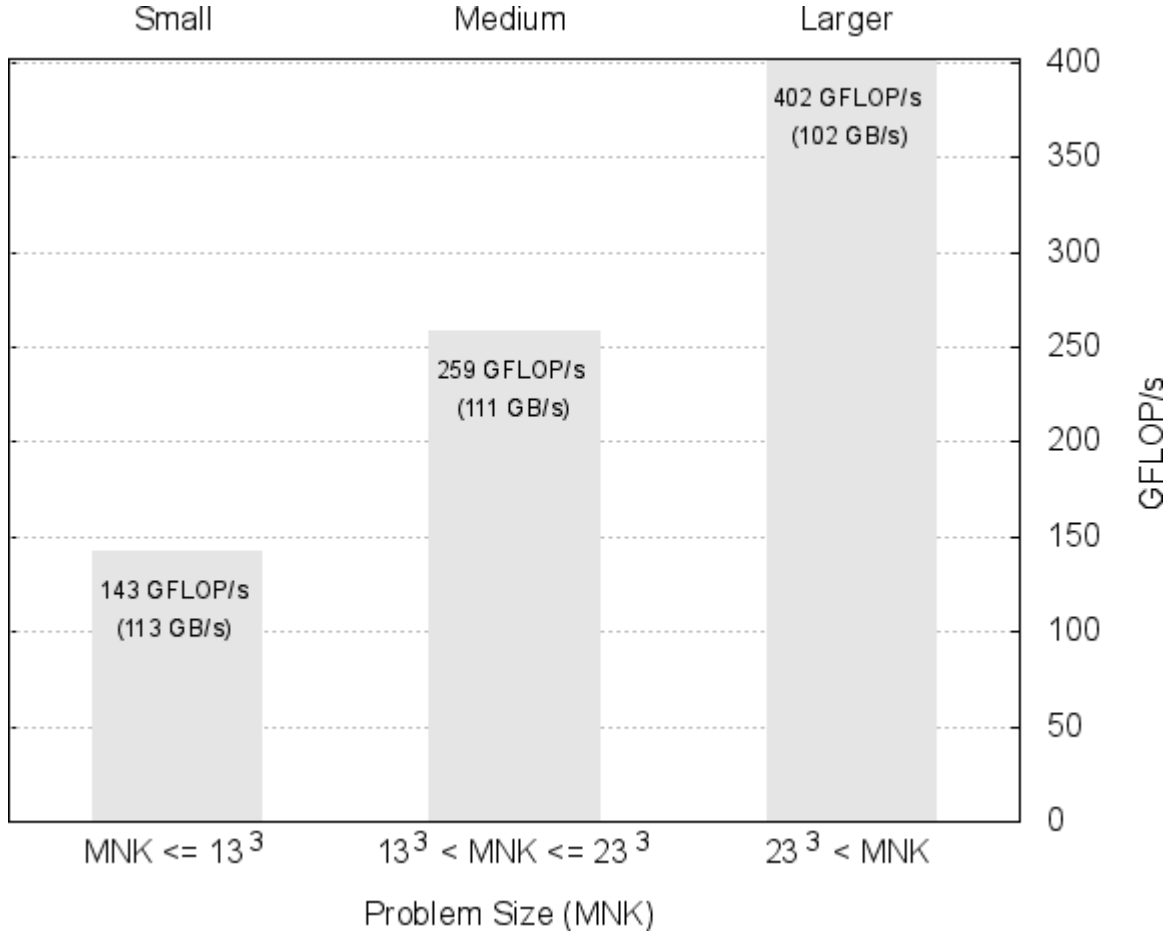

Figure 3: This plot shows the arithmetic average (non-sliding) of the performance with respect to groups of problem sizes (MNK). The problem sizes are binned into three groups according to the shown intervals: "Small", "Medium", and "Larger" (notice that "larger" may still not be a large problem size). Please refer to the first figure on how to reproduce the results.

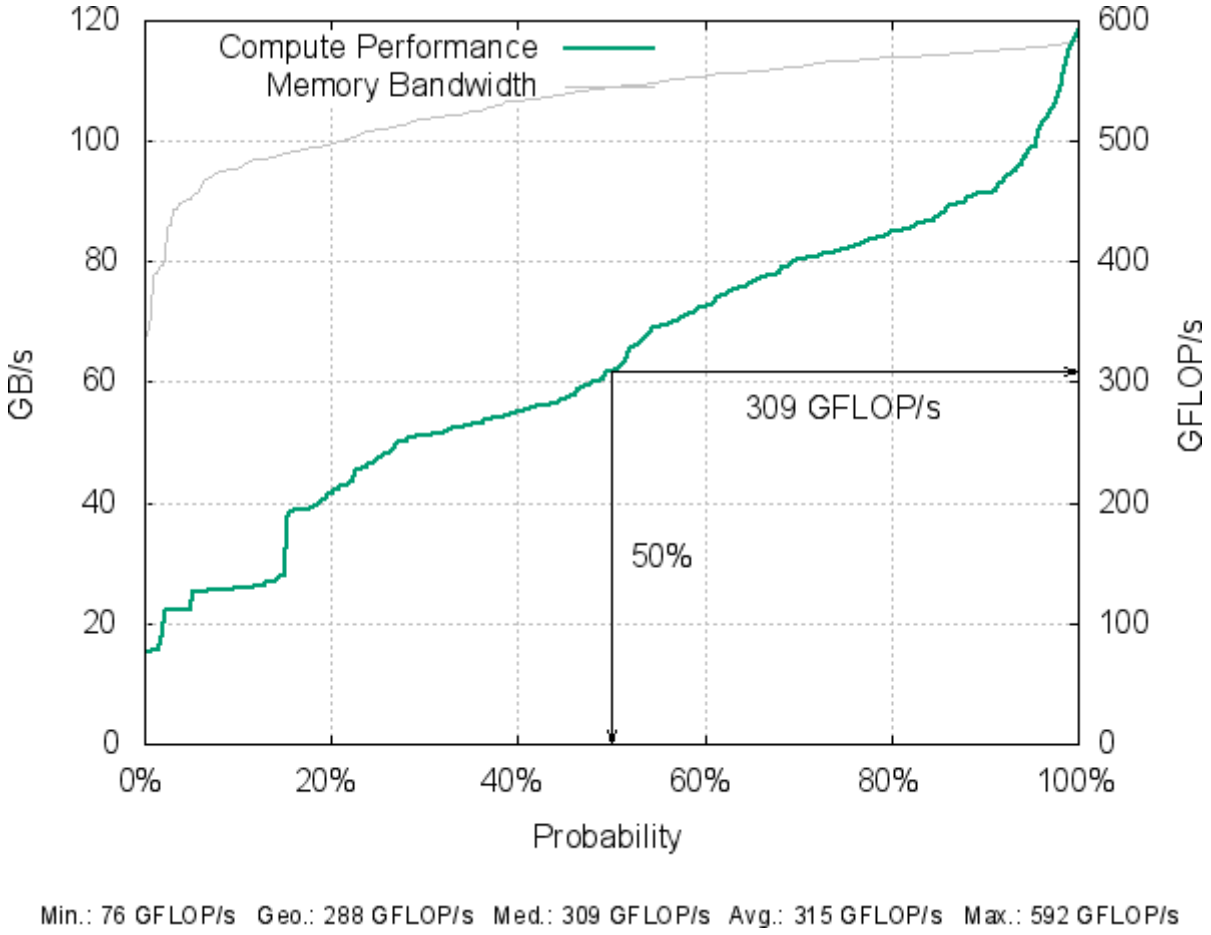

Figure 4: In order to further summarize the previous plots, this graph shows the cumulative distribution function (CDF) of the performance across all cases. Similar to the median value at 50%, one can read for example that 100% of the cases are yielding less or equal the largest discovered value. The value highlighted by the arrows is usually the median value, the plot script however attempts to highlight a single "fair performance value" representing all cases by linearly fitting the CDF, projecting onto the x-axis, and taking the midpoint of the projection (usually at 50%). Please note, that this diagram shows a statistical distribution and does not allow to identify any particular kernel. Moreover at any point of the x-axis ("Probability"), the "Compute Performance" and the "Memory Bandwidth" graph do not necessarily belong to the same kernel! Please refer to the first figure on how to reproduce the results.

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 or optimized FORTRAN 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). However, auto-tuning also imposes a reasonable burden (similar to profile guided optimization) and the current library is supposed to generate extremely efficient code for everything that does not rely on L2 cache blocking (below THRESHOLD), and except for very small problem sizes where a dedicated kernel may beat our implementation (e.g., MNK=4x4x4).

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:

- Full xGEMM interface, and extended code dispatcher
- Just-in-Time (JIT) runtime dynamic code generation
- API supporting sparse matrices and other cases

Applications and References

[1] http://cp2k.org/: Open Source Molecular Dynamics which (optionally) uses LIBXSMM. The application is generating batches of small matrix-matrix multiplications ("matrix stack") out of a problem-specific distributed block-sparse matrix (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] http://software.intel.com/xeonphicatalog: Intel Xeon Phi Applications and Solutions Catalog.

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