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 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 aforementioned instruction set extensions (static code generation), and Just-In-Time (JIT) code generation is targeting Intel AVX and beyond.

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)^(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 L2 cache. For problem sizes above the configurable threshold, LIBXSMM is falling back to BLAS.

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" 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.

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();
/** Uninitialize the library and free internal memory (optional). */
void libxsmm_finalize();
```

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/BLAS interface can be treated with "default arguments":

```
/** Calling the automatically dispatched dense matrix multiplication (single-precision, C code). */
libxsmm_sgemm(0/*transa*/, 0/*transb*/, &m/*required*/, &n/*required*/, &k/*required*/,
  0/*alpha*/, a/*required*/, 0/*lda*/, b/*required*/, 0/*ldb*/,
  0/*beta*/, c/*required*/, 0/*ldc*/);
/** Calling the automatically dispatched dense matrix multiplication (C++ code). */
libxsmm_gemm(0/*transa*/, 0/*transb*/, m/*required*/, n/*required*/, k/*required*/,
  0/*alpha*/, a/*required*/, 0/*lda*/, b/*required*/, 0/*ldb*/,
  0/*beta*/, c/*required*/, 0/*ldc*/);
```

For the C interface (with type prefix 's' or 'd'), all required arguments and in particular m, n, and k are passed by pointer. This is needed for binary compatibility with the original GEMM/BLAS interface. In contrast, the C++ interface makes this requirement more clear by passing m, n, and k by-value.

```
! Calling the automatically dispatched dense matrix multiplication (single-precision).
CALL libxsmm_sgemm(m=m, n=n, k=k, a=a, b=b, c=c)
! Calling the 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 similar BLAS-based dense matrix multiplication is provided and simply re-exposed from the underlying GEMM/BLAS implementation (libxsmm_blas_gemm). However, the re-exposed functions perform argument twiddling to account for ROW_MAJOR storage order (if enabled). The BLAS-based GEMM might be useful for validation/benchmark purposes, and more important as a fallback implementation when building an application-specific dispatch mechanism. Successively calling a certain kernel (i.e., multiple times) allows for amortizing the cost of the 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_sfunction libxsmm_sdispatch(int m, int n, int k,
                                    int lda, int ldb, int ldc,
                                    /* supply NULL as a default for alpha or beta */
                                    const float* alpha, const float* beta);
/** If non-zero function pointer is returned, call (*function_ptr)(a, b, c). */
libxsmm_dfunction libxsmm_ddispatch(int m, int n, int k,
                                    int lda, int ldb, int ldc,
                                    /* supply NULL as a default for alpha or beta */
                                    const double* alpha, const double* beta);
```

Further, a variety of overloaded function signatures is provided allowing to omit arguments not deviating from the configured defaults. Moreover, in C++ a type 'libxsmm_function<*type*>' can be used to instantiate a functor rather than making a distinction for the numeric type in 'libxsmm_?dispatch'. Similarly in Fortran, when calling the generic interface (libxsmm_dispatch) the given LIBXSMM_?MM_FUNCTION is dispatched such that libxsmm_call can be used to actually perform the function call using the PROCEDURE POINTER wrapped by LIBXSMM_?MM_FUNCTION.

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 or column-major (default) order matrices. The row-major storage scheme is accomplished by setting ROW_MAJOR=1 (0 for column-major, and row-major otherwise):

```
make ROW_MAJOR=1
```

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 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)
```

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"
```

Performance

Tuning

By default all supported host code paths are generated (with the compiler picking the one according to the feature bits of the host). Specifying a particular code path will not only save some time when generating the static code ("printing"), but also enable cross-compilation for a target that is different from the compiler's host. The build system allows to conveniently select the target system when invoking 'make': 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 foundational Intel AVX-512 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 general alignment (ALIGNMENT=64) as well as an alignment specific for the store instructions (ALIGNED_STORES can be specified when invoking 'make' (by default ALIGNED_STORES inherits ALIGNMENT when "enabled" using ALIGNED_STORES=1). 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 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 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.

An extended interface can 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 Directly invoking the generator backend 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 enabling a default strategy ("AL2_BL2viaC" rather than "nopf"). The following example is requesting the "AL2jpst" strategy:

```
make PREFETCH=8
```

The interface which is supporting software prefetches extends the signature of all kernels by three arguments (pa, pb, and pc) allowing the call-side to specify where to prefetch the operands of the "next" multiplication from (a, b, and c). There are macros available (C/C++ only) allowing to call the matrix multiplication functions in a prefetch-agnostic fashion (see cp2k or smm code samples).

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_?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'. For statically generated kernels, the interface includes prototypes of the specialized functions.

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) 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. If a problem size is below the threshold, dispatching the code requires to figure out whether a specialized routine exists or not.

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 offers an experimental JIT (just-in-time) backend which generates the requested kernels on the fly. This is accomplished by emitting the corresponding byte-code directly into an executable buffer. As the JIT backend is still experimental, some limitations are in place:

1. There is no support for SSE3 (Intel Xeon 5500/5600 series) and IMCI (Intel Xeon Phi coprocessor code-named Knights Corner) instruction set extensions
2. LIBXSMM uses Pthread mutexes to guard updates of the JITted code cache (link line with -lpthread is required); building with OMP=1 employs an OpenMP critical section as an alternative locking mechanism.
3. There is no support for the Windows calling convention.

The JIT backend support in LIBXSMM can be enabled using:

```
make JIT=1
```

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_?dispatch) are not subject to a problem size threshold. In any case, JIT code generation can be used to 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 JITted 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).

Directly invoking the generator backend

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/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 (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. Prefetch strategy, see below enumeration (dense/dense_asm only)
15. single precision (SP), or double precision (DP)
16. 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/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, there are additional examples given in samples/generator and samples/seissol.

Results

The library does not claim to be "optimal" or "best-performing", and the presented results (figure 1-3) are modeling a certain application which might be not representative in general. Instead, information about how to reproduce the results is given.

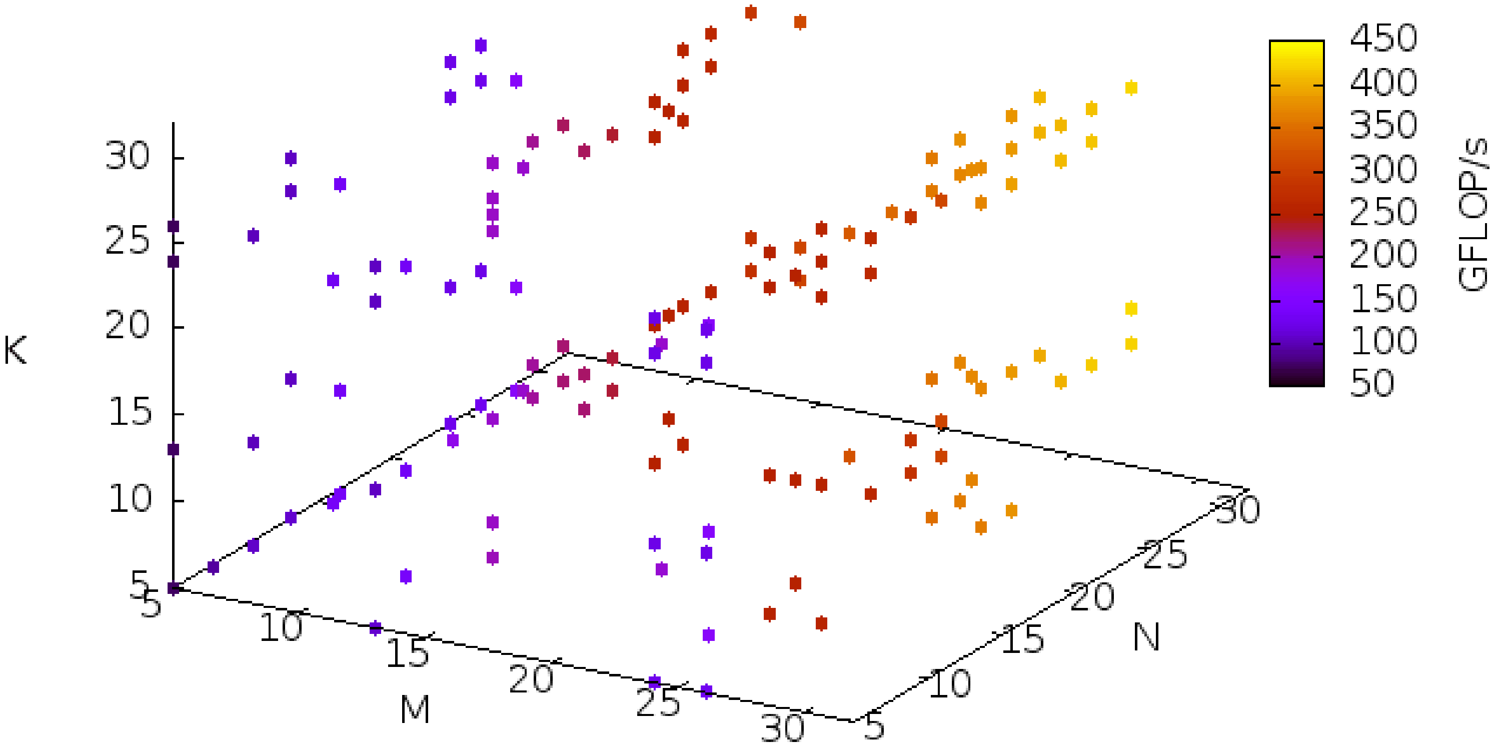

Figure 1: This plot shows the performance (based on LIBXSMM 1.0) for a dual-socket Intel Xeon E5-2699v3 ("Haswell") shows a "compact selection" (to make the plot visually more appealing) out of 386 specializations as useful for CP2K Open Source Molecular Dynamics [1]. The code has been generated and built by running "./make.sh -cp2k AVX=2 test -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).

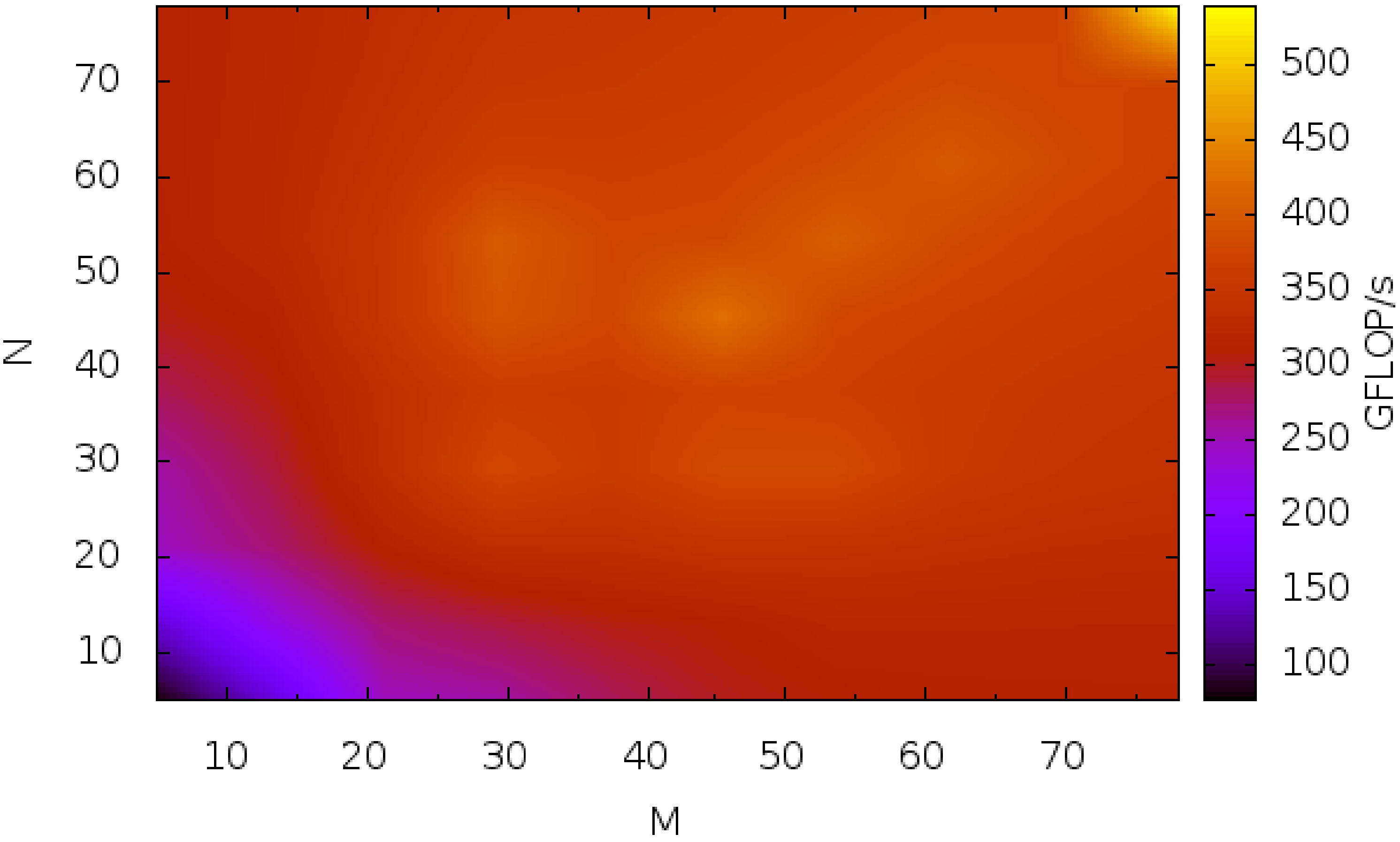

Figure 2: This plot summarizes the performance (based on LIBXSMM 1.0) of the generated kernels by averaging the results over K (and therefore the bar on the right hand side may not show the same maximum when compared to other plots). The performance is well-tuned across the parameter space with no "cold islands", and the lower left "cold" corner is fairly limited. Please refer to the first figure on how to reproduce the results.

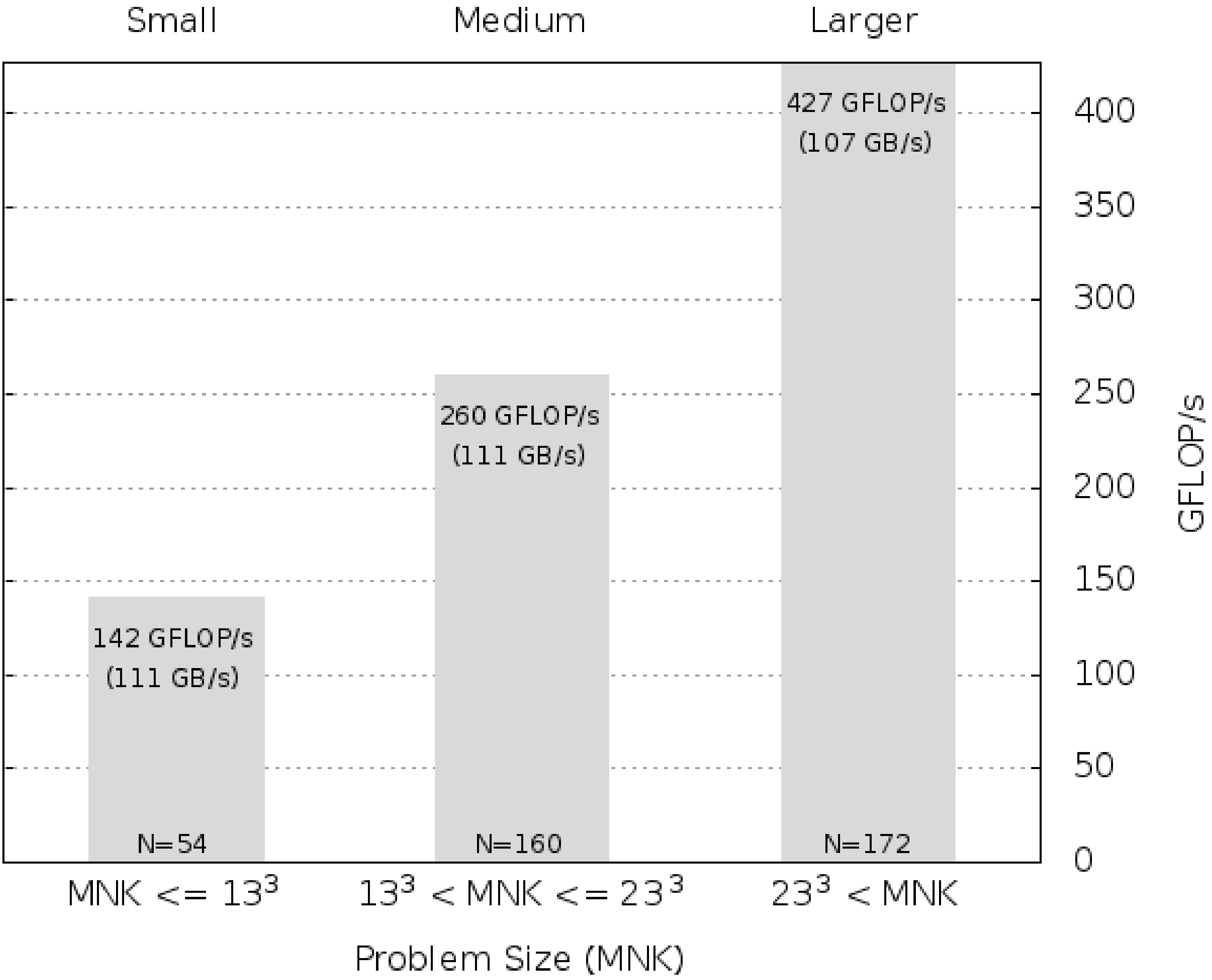

Figure 3: This plot shows the arithmetic average (non-sliding) of the performance (based on LIBXSMM 1.0) with respect to groups of problem sizes (MNK). The problem sizes are binned into three groups according to the shown intervals: "Small", "Medium", and "Larger" (notice that "larger" may still not be a large problem size). Please refer to the first figure on how to reproduce the results.

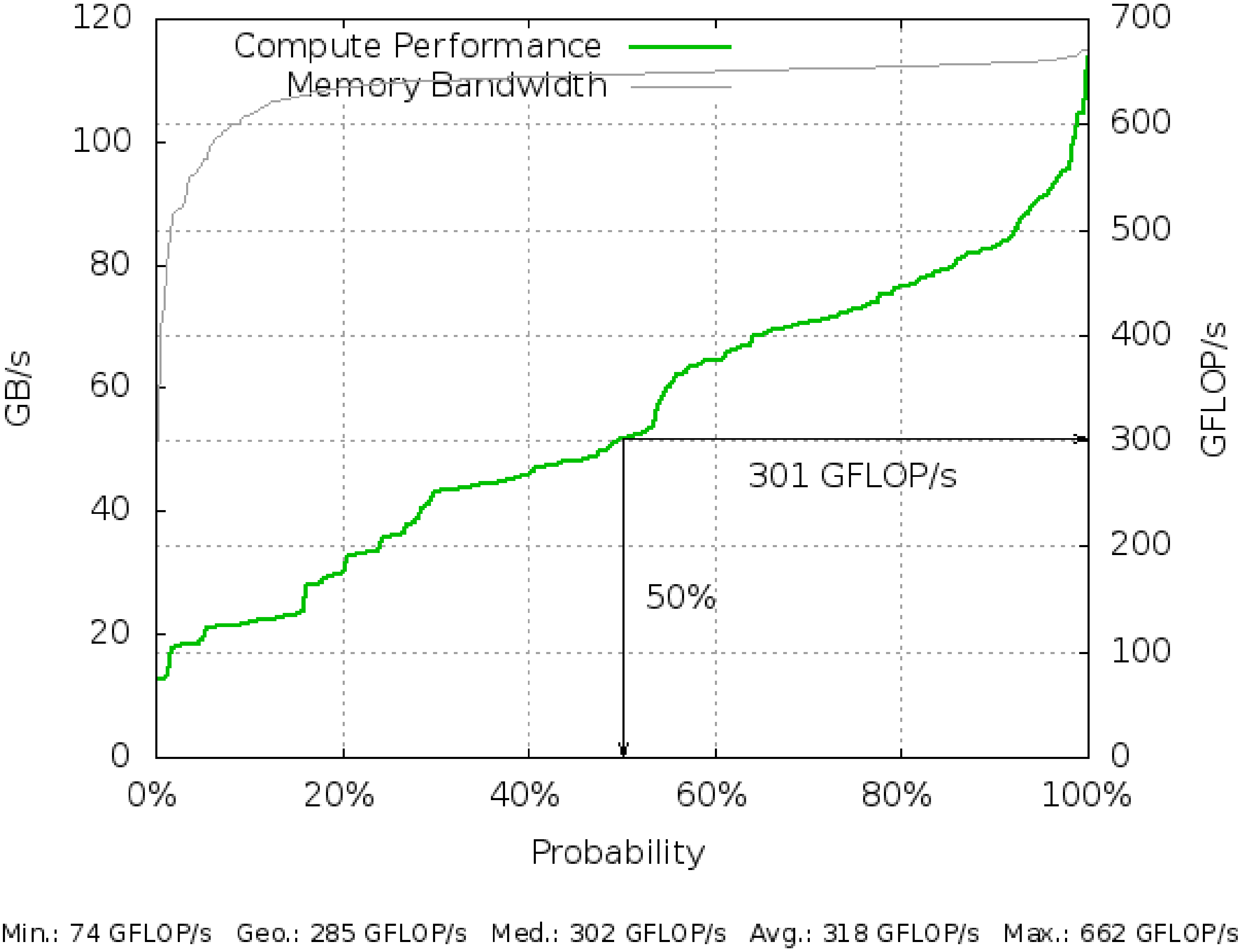

Figure 4: In order to further summarize the previous plots, this graph shows the cumulative distribution function (CDF) of the performance (based on LIBXSMM 1.0) across all cases. Similar to the median value at 50%, one can read for example that 100% of the cases are yielding less or equal the largest 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.

Implementation

Limitations

Beside of the inlinable C code or the optimized FORTRAN code path, the library is currently limited to a single code path which is selected at build time of the library (also true for JITted code). Without a specific flag (SSE=1, AVX=1|2|3), the static code generation 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.

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
- API supporting sparse matrices and other cases

Applications and References

[1] **http://cp2k.org/**: Open Source Molecular Dynamics with its DBCSR component generating batches of small matrix multiplications ("matrix stacks") out of a problem-specific distributed block-sparse matrix. The idea and the interface of LIBXSMM is sharing some origin with CP2K's "libsmm" library which can be optionally substituted by 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] **http://software.intel.com/xeonphicatalog**: Intel Xeon Phi Applications and Solutions Catalog.

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