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 interface can be treated with "default arguments" (which is deviating from LAPACK/BLAS standard however without compromising the binary compatibility).

```
/** Call 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*/);
/** Call 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. In contrast, the C++ interface is supplying overloaded versions which allow to passing m, n, and k by-value (which makes 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 allowing NULL to be passed). For convenience, a similar 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. 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.

```
! Call automatically dispatched dense matrix multiplication (single/double-precision).
CALL libxsmm_?gemm(m=m, n=n, k=k, a=a, b=b, c=c)
! Call automatically dispatched dense matrix multiplication (generic interface).
CALL libxsmm_gemm(m=m, n=n, k=k, a=a, b=b, c=c)
```

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,
                                        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_dmmfunction libxsmm_dmmdispatch(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);
```

A variety of overloaded function signatures is provided allowing to omit arguments not deviating from the configured defaults. Moreover, 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

To generate the interface 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
```

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 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"
```

Installation

Installing LIBXSMM makes the most sense if the JIT backend has been enabled (default), because a statically specialized library is more application-specific as well as system-specific. Remember that statically specialized functions cannot be retargeted to a different instruction set extension! However, even a JIT-enabled library (in particular within a heterogeneous system environment) should be built using an applicable baseline code path: SSE=1, AVX=1|2|3. Remember, LIBXSMM is by default built using an "arch-native" approach where the system running the compiler is determining the baseline architecture. There are two main mechanisms to install LIBXSMM: (1) building the library in an out-of-tree fashion, and (2) installing the library into a certain location (both mechanisms can be combined). Building in an out-of-tree fashion looks like:

```
cd libxsmm-install
make -f /path/to/libxsmm/Makefile
make clean
```

Assuming the library is already built, one can install LIBXSMM into a certain location:

```
make install PREFIX=/path/to/libxsmm-install
make clean
```

Performing `make install-minimal` omits to install the documentation under (`PREFIX/share/libxsmm`).

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
```

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 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 enabling a default strategy ("AL2_BL2viaC" rather than "nopf"). 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 three arguments are specifying the locations of the operands of the next multiplication (the next a, b, and c).

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) 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. 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 both single-precision and double-precision, and hence to not save any space in the cache (PRECISION=0), whereas PRECISION=1 denotes to generate only single-precision code versions and PRECISION=2 denotes the preference for double precision.

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. 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. 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 in LIBXSMM can also be disabled (`make 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 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).

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_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_generator dense foo.c foo 16 16 16 32 32 32 1 1 1 1 hsw nopf DP
bin/libxsmm_generator dense_asm foo.c foo 16 16 16 32 32 32 1 1 1 1 knl AL2_BL2viaC DP
bin/libxsmm_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 are modeling a certain application which might be not representative in general. Instead, information on how to reproduce the results is given underneath of the presented results (figure 1-3).

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).

Implementation

Limitations

The statically generated code is depending on a single code path which is selected at build time of the library whereas the JITted code depends on the actual CPUID flags of the target system executing the library. For the statically generated code and without a specific flag (SSE=1, AVX=1|2|3), the 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).

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 substituted by LIBXSMM (see https://github.com/hfp/libxsmm/raw/master/documentation/cp2k.pdf).

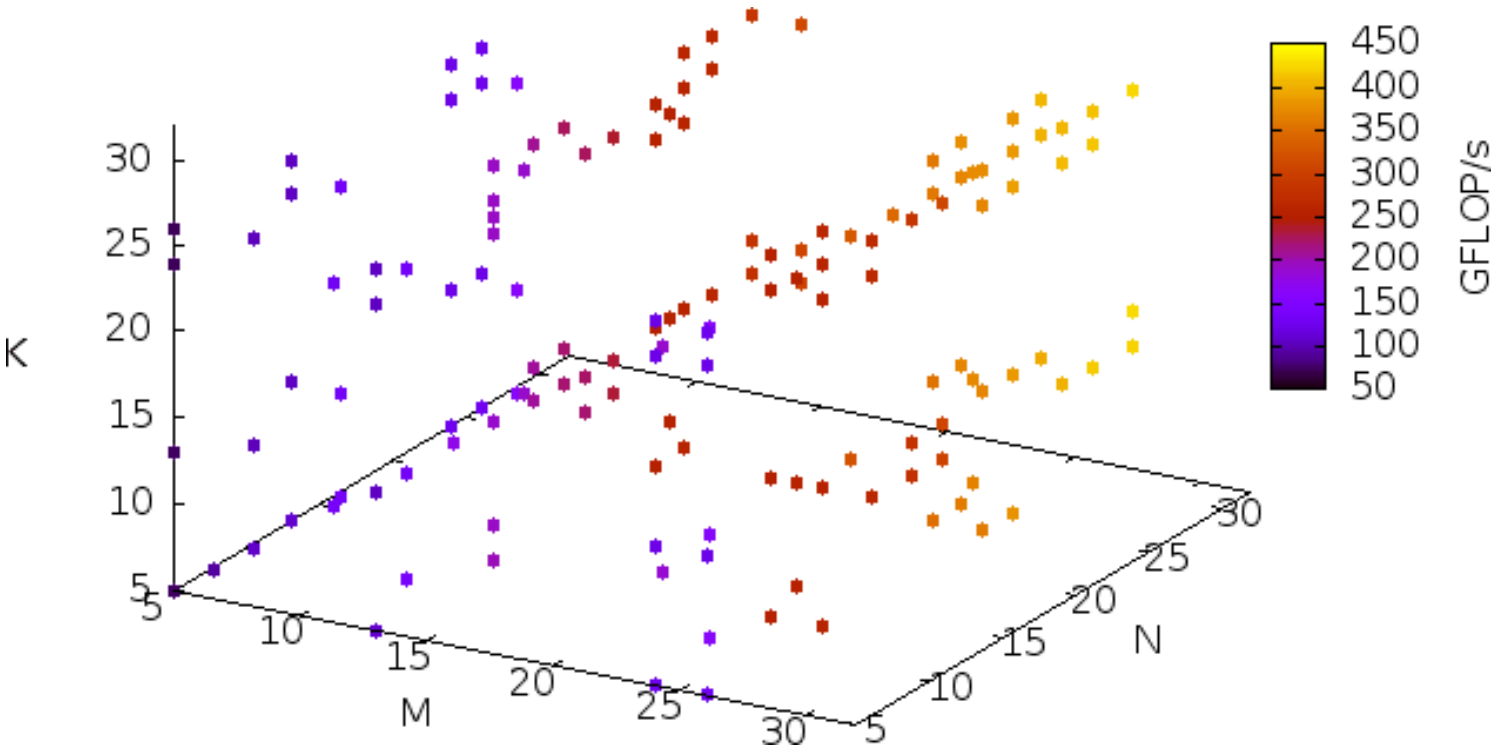


Figure 1: This plot shows the performance (based on LIBXSMM 1.0) for a dual-socket Intel Xeon E5-2699v3 ("Haswell") shows a "compact selection" (to make the plot visually more appealing) out of 386 specializations as useful for CP2K Open Source Molecular Dynamics [1]. The code has been generated and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

[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.

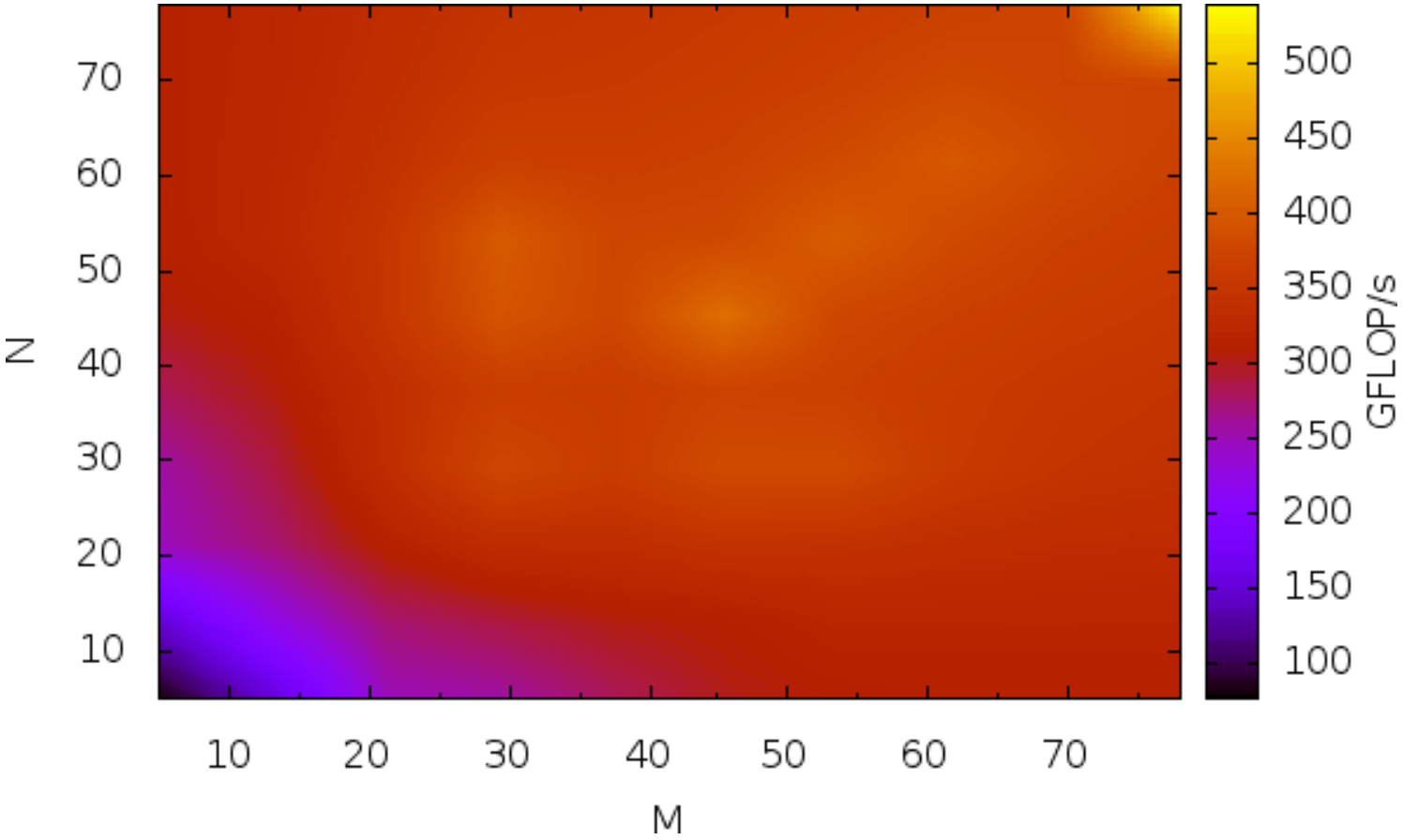


Figure 2: This plot summarizes the performance (based on LIBXSMM 1.0) of the generated kernels by averaging the results over K (and therefore the bar on the right hand side may not show the same maximum when compared to other plots). The performance is well-tuned across the parameter space with no "cold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

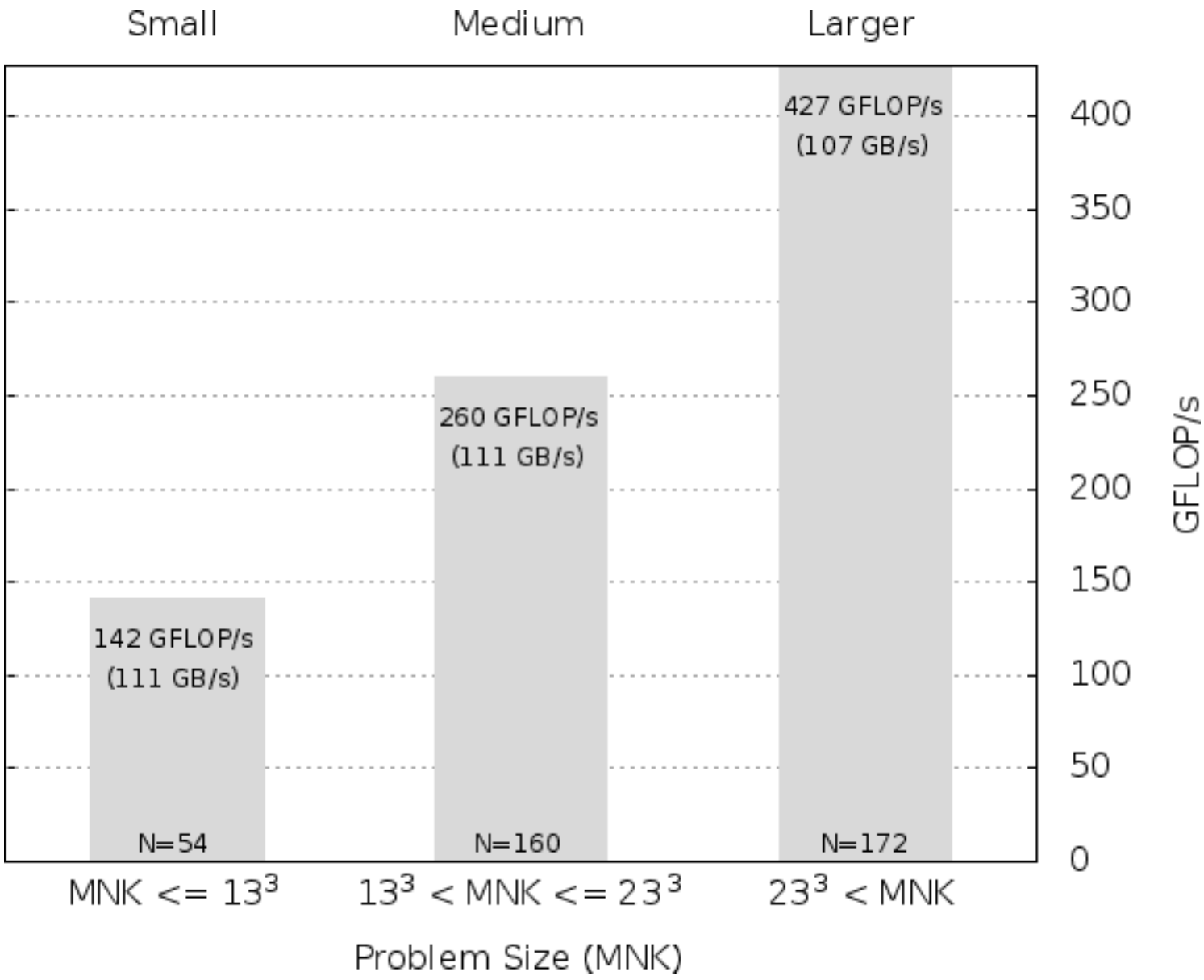
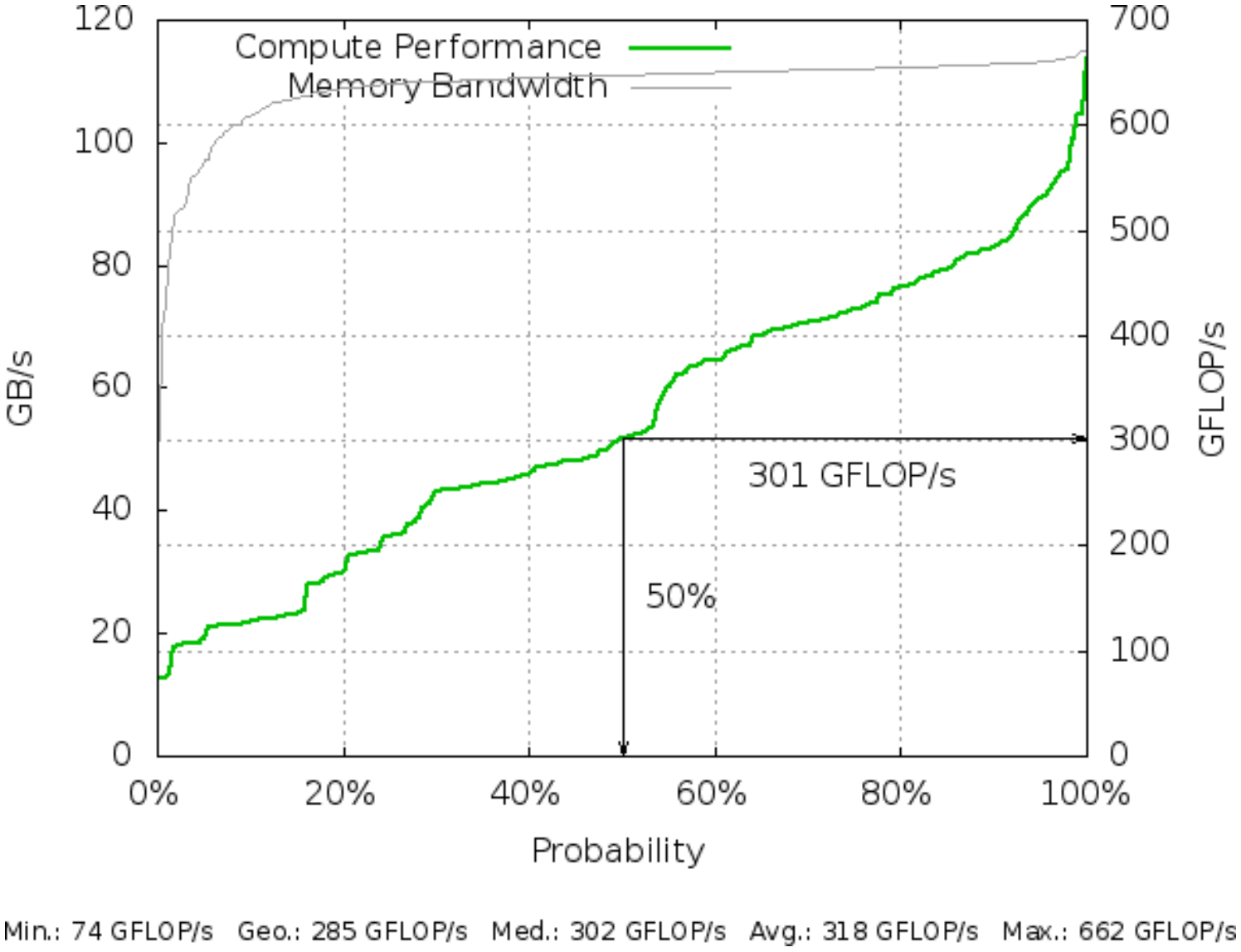


Figure 3: This plot shows the arithmetic average (non-sliding) of the performance (based on LIBXSMM 1.0) with respect to groups of problem sizes (MNK). The problem sizes are binned into three groups according to the shown intervals: "Small", "Medium", and "Larger" (notice that "larger" may still not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



Min.: 74 GFLOP/s Geo.: 285 GFLOP/s Med.: 302 GFLOP/s Avg.: 318 GFLOP/s Max.: 662 GFLOP/s

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.