

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 SSE, 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 the background of the name “LIBXSMM”? The “MM” stands for Matrix Multiplication, and the “S” clarifies the working domain i.e., Small Matrix Multiplication. The latter also means the name is neither a variation of “MXM” nor an eXtreme Small Matrix Multiplication but rather about Intel Architecture (x86) - and no, the library is 64-bit (only). The spelling of the name might follow the syllables of `libx\smm`, `libx’smm`, or `libx-smm`.

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 $(M \ N \ K)^{1/3} \leq 80$ (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(void);
/** Uninitialize the library and free internal memory (optional). */
void libxsmm_finalize(void);
```

To perform the dense matrix-matrix multiplication $C_{m \times n} = \alpha \cdot A_{m \times k} \cdot B_{k \times n} + \beta \cdot C_{m \times n}$, the full-blown GEMM interface can be treated with “default arguments” (which is deviating from LAPACK/BLAS standard however without compromising the binary compatibility).

```
/** 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*/);
/** 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. The C++ interface is also supplying overloaded versions where m, n, and k are allowed to be passed by-value (making 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.

```
! Automatically dispatched dense matrix multiplication (single/double-precision).
CALL libxsmm_?gemm(m=m, n=n, k=k, a=a, b=b, c=c)
! 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,
    const int* lda, const int* ldb, const int* ldc,
    const float* alpha, const float* beta,
    const int* flags, const int* prefetch);
/** If non-zero function pointer is returned, call (*function_ptr)(a, b, c). */
libxsmm_dmmfunction libxsmm_dmmdispatch(int m, int n, int k,
    const int* lda, const int* ldb, const int* ldc,
    const double* alpha, const double* beta,
    const int* flags, const int* prefetch);
```

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

The build system relies on GNU Make (typically associated with the `make` command, but e.g. FreeBSD is calling it `gmake`). The build can be customized by using key-value pairs. Key-value pairs can be supplied in two ways: (1) after the “make” command, or (2) prior to the “make” command (`env`) which is effectively the same as exporting the key-value pair as an environment variable (`export`, or `setenv`). Of course both methods can be mixed, however the second method may requires to supply the `-e` flag. Please note that the CXX, CC, and FC keys are handled such that they are taken into account in any case.

To generate the interface of the library 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"
```

NOTE: by default, a combination of a C/C++ and a Fortran compiler is needed (some sample code is written in C++). Beside of specifying the compilers (`make CXX=g++ CC=gcc FC=gfortran` and maybe `AR=ar`), the need for a Fortran compiler can be relaxed (`make FC=`). The latter affects the availability of the `MOD` file and the corresponding ‘libxsmmf’ library (the interface ‘libxsmm.f’ is still generated). Fortran code can make use of LIBXSMM in three different ways:

- By relying on the module file, and by linking against ‘libxsmmf’ and ‘libxsmm’ (this order),
- By including the interface ‘libxsmm.f’, and by linking solely against ‘libxsmm’, or by
- Optionally declaring `libxsmm_?gemm` (BLAS signature), and by linking against ‘libxsmm’.

At the expense of a limited functionality (`libxsmm_?gemm`), the latter method also works with Fortran 77 (otherwise the Fortran 2003 standard is necessary).

Installation

Installing LIBXSMM makes possibly the most sense when combining the JIT backend (enabled by default) with a collection of statically generated SSE kernels (by specifying M, N, K, or MNK). If the JIT backend is not disabled, statically generated kernels are only registered for dispatch if the CPUID flags at runtime are not supporting a more specific instruction set extension (code path). Since the JIT backend does not support or generate SSE code by itself, the library is compiled by selecting SSE code generation if not specified otherwise (`AVX=1|2|3`, or with `SSE=0` falling back to an “arch-native” approach). Limiting the static code path to SSE3 allows to practically target any deployed system, however using `SSE=0` and `AVX=0` together is falling back to generic code, and any static kernels are not specialized using the assembly code generator.

There are two main mechanisms to install LIBXSMM (both mechanisms can be combined): (1) building the library in an out-of-tree fashion, and (2) installing into a certain location. Building in an out-of-tree fashion looks like:

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

For example, installing into a specific location (incl. a selection of statically generated Intel SSE kernels) looks like:

```
make MNK="1 2 3 4 5" PREFIX=/path/to/libxsmm-install install
```

Performing `make install-minimal` omits the documentation (default: ‘PREFIX/share/libxsmm’). Moreover, `PINCDIR`, `POUTDIR`, `PBINDIR`, and `PDOCDIR` allow to customize the locations underneath of the PREFIX location. To build a general package for an unpredictable audience (Linux distribution, or similar), it is advised to not over-specify

or customize the build step i.e., JIT, SSE, AVX, OMP, BLAS, etc. should not be used. The only real question remains for PREFETCH=0 (default) versus PREFETCH=1. The following is building and installing a complete set of libraries where the generated interface matches both the static and the shared libraries:

```
make PREFIX=/path/to/libxsmm-install STATIC=0 install
make PREFIX=/path/to/libxsmm-install install
```

NOTE: the library is agnostic with respect to the threading-runtime, and enabling OpenMP (OMP=1) when building the library is a non-default option (untested). The libraries are also agnostic with respect to the selected LAPACK-/BLAS library (if supported by OS and link model), and therefore linking GEMM routines when building the library (by supplying BLAS=1|2) may prevent a user from deciding at the time of linking the actual application.

Running

Call Wrapper

Since the library is binary compatible with existing GEMM calls (LAPACK/BLAS), these calls can be replaced at link-time or intercepted at runtime of an application such that LIBXSMM is used instead of the original LAPACK/BLAS. Currently this only works for the Linux OS (not validated under OS X), and it is also not sufficient to rely on a GNU tool chain under Microsoft Windows. Of course, using LIBXSMM's programming interface when performing the same multiplication multiple time in a consecutive fashion (batch-processing) allows to extract higher performance. However, using the call wrapper might motivate to make use of the LIBXSMM API.

There are two cases to consider: (1) an application which is linking statically against LAPACK/BLAS, and (2) an application which is dynamically linked against LAPACK/BLAS. The first case requires making `-Wl,--wrap=sgemm_ /path/to/libxsmm.a` (star to be replaced) part of the link-line and then relinking the application:

```
gcc [...] -Wl,--wrap=sgemm_ /path/to/libxsmm.a
gcc [...] -Wl,--wrap=dgemm_ /path/to/libxsmm.a
gcc [...] -Wl,--wrap=sgemm_,-wrap=dgemm_ /path/to/libxsmm.a
```

Relinking the application is often accomplished by copying, pasting, and modifying the linker command as shown when running “make” (or a similar build system), and then just re-invoking the modified link step. Please note that this first case is also working for an applications which is dynamically linked against LAPACK/BLAS. The static link-time wrapper technique in general may only work with a GCC compatible tool chain (GNU Binutils: 1d, or 1d via compiler-driver), and it has been tested with GNU GCC, Intel Compiler, and Clang. The latter unfortunately does not include Compiler 6.1 or earlier under OS X, and it has not been tested with a later version of this tool chain. The symbol names to be wrapped unfortunately depend on the BLAS library, and the following two cases are supported: `sgemm_` and `dgemm_` are generally the generic choice and they have been tested with reference LAPACK/BLAS and OpenBLAS. For Intel MKL, `mk1_sgemm_` and `mk1_dgemm_` must be specified:

```
ld [...] -Wl,--wrap=mk1_sgemm_ /path/to/libxsmm.a
ld [...] -Wl,--wrap=mk1_dgemm_ /path/to/libxsmm.a
ld [...] -Wl,--wrap=mk1_sgemm_,-wrap=mk1_dgemm_ /path/to/libxsmm.a
```

If an application is dynamically linked against LAPACK/BLAS, the unmodified application allows for intercepting these calls at startup time (runtime) by using the LD_PRELOAD mechanism:

```
LD_PRELOAD=/path/to/libxsmmext.so ./myapplication
```

This case obviously requires to build a shared library of LIBXSMM:

```
make STATIC=0
```

Please note that calling SGEMM is more sensitive to dispatch-overhead when compared to multiplying the same matrix sizes in double-precision. In case of single-precision, an approach of using the call wrapper is often not able to show an advantage if not regressing with respect to performance (therefore SGEMM is likely asking for making use of the API). In contrast, the double-precision case can show up to two times the performance of a typical LAPACK/BLAS performance (and more when using the API for processing batches).

Call Trace

During the initial steps of employing the LIBXSMM API, one may rely on a debug version of the library (`make DBG=1`). The latter implies standard error (`stderr`) output in case of an error/warning condition inside of the library. Towards developing an application which is successfully using the library, being able to trace library calls might be useful as well (can be combined with `DBG=1` or `OPT=0`):

```
make TRACE=1
```

Building an application which is able to trace calls (inside of the library) requires the shared library of LIBXSMM, alternatively the application is required to link the static library in a dynamic fashion (GNU tool chain: `-rdynamic`). Actually tracing calls (without debugger) can be accomplished by an environment variable called LIBXSMM_TRACE.

```
LIBXSMM_TRACE=1 ./myapplication
```

Syntactically up to three arguments separated by commas (which allows to omit arguments) are taken (*tid,i,n*): *tid* signifies the ID of the thread to be traced with 1...NTHREADS being valid and where LIBXSMM_TRACE=1 is filtering for the “main thread” (in fact the first thread running into the trace facility); grabbing all threads (no filter) can be achieved by supplying a negative id (which is also the default when omitted). The second argument is pruning higher levels of the call-tree with *i=1* being the default (level zero is the highest at the same level as the main function). The last argument is taking the number of inclusive call levels with *n=-1* being the default (signifying no filter).

Please note that the trace facility is severely impacting the performance (even with LIBXSMM_TRACE=0), and this is not just because of console output but rather since inlining (internal) function might be prevented along with additional call overhead on each function entry and exit. Therefore debug symbols can be also enabled separately (`make SYM=1`; implied by TRACE=1 or DBG=1) which might be useful when profiling an application. No facility of the library (other than DBG or TRACE/LIBXSMM_TRACE) is performing visible (console) or other non-private I/O (files).

Performance

Tuning

Specifying a particular code path is not really necessary if the JIT backend is not disabled. However, disabling JIT compilation, statically generating a collection of kernels, and targeting a specific instruction set extension for the entire library looks like:

```
make JIT=0 AVX=3 MNK="1 2 3 4 5"
```

The above example builds a library which cannot be deployed to anything else but the Intel Knights Landing processor family (“KNL”) or future Intel Xeon processors supporting foundational Intel AVX-512 instructions (AVX-512F). The latter might be even more adjusted by supplying MIC=1 (along with AVX=3), however this does not matter since critical code is in inline assembly (and not affected). Similarly, SSE=0 (or JIT=0 without SSE or AVX build flag) employs an “arch-native” approach whereas AVX=1, AVX=2 (with FMA), and AVX=3 are specifically selecting the kind of Intel AVX code. Moreover, controlling the target flags manually or adjusting the code optimizations is also possible. The following example is GCC-specific and corresponds to OPT=3, AVX=3, and MIC=1:

```
make OPT=3 TARGET="-mavx512f -mavx512cd -mavx512er -mavx512pf"
```

An extended interface can be 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 “AL2jpt” strategy:

```
make PREFETCH=8
```

The prefetch interface is extending the signature of all kernels by three arguments (pa, pb, and pc). These additional arguments are specifying the locations of the operands of the next multiplication (the next a, b, and c matrices).

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 section) 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 ($M \times N \times K$) 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. Please note that explicitly JIT'ing and executing a kernel is possible and independent 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 and register both single and double-precision code, and therefore no space in the dispatch table is saved (PRECISION=0). Specifying PRECISION=1|2 is only generating and registering either single-precision or double-precision code.

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 implements a JIT (Just-In-Time) code generation backend which generates the requested kernels on the fly (in-memory). 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. In the current implementation, some limitations apply to the JIT backend specifically:

1. In order to stay agnostic to any threading model used, Pthread mutexes are guarding the updates of the JIT'ed code cache (link line with `-lpthread` is required); building with `OMP=1` employs an OpenMP critical section as an alternative locking mechanism.
2. There is no support for the Intel SSE (Intel Xeon 5500/5600 series) and IMCI (Intel Xeon Phi coprocessor code-named Knights Corner) instruction set extensions. However, statically generated SSE-kernels can be leveraged without disabling support for JIT'ing AVX kernels.
3. There is no support for the Windows calling convention (only kernels with `PREFETCH=0` signature).

The JIT backend can also be disabled at build time (`make JIT=0`) as well as at runtime (`LIBXSMM_JIT=0`). The latter is an environment variable which is also allowing to set a code path independent of the CPUID (`LIBXSMM_JIT=0|1|snb|hsw|knl`). Please note that `LIBXSMM_JIT=1` is only supported for symmetry, and this environment setup cannot enable the JIT backend if it was disabled at build time (`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 JIT'ed 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_gemm_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_gemm_generator dense foo.c foo 16 16 16 32 32 32 1 1 1 1 hsw nopf DP
bin/libxsmm_gemm_generator dense_asm foo.c foo 16 16 16 32 32 32 1 1 1 1 knl AL2_BL2viaC DP
bin/libxsmm_gemm_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).

Applications

[1] <https://cp2k.org/>: Open Source Molecular Dynamics with its DBCSR component processing batches of small matrix multiplications (“matrix stacks”) out of a problem-specific distributed block-sparse matrix. Starting with CP2K 3.0, LIBXSMM can be used to substitute CP2K’s ‘libsmm’ library. Prior to CP2K 3.0, only the Intel-branch of CP2K was integrating 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/).

References

[1] http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post137.html: LIBXSMM: A High Performance Library for Small Matrix Multiplications (poster and two-page extended abstract). SC’15: The International Conference for High Performance Computing, Networking, Storage and Analysis, Austin (Texas).

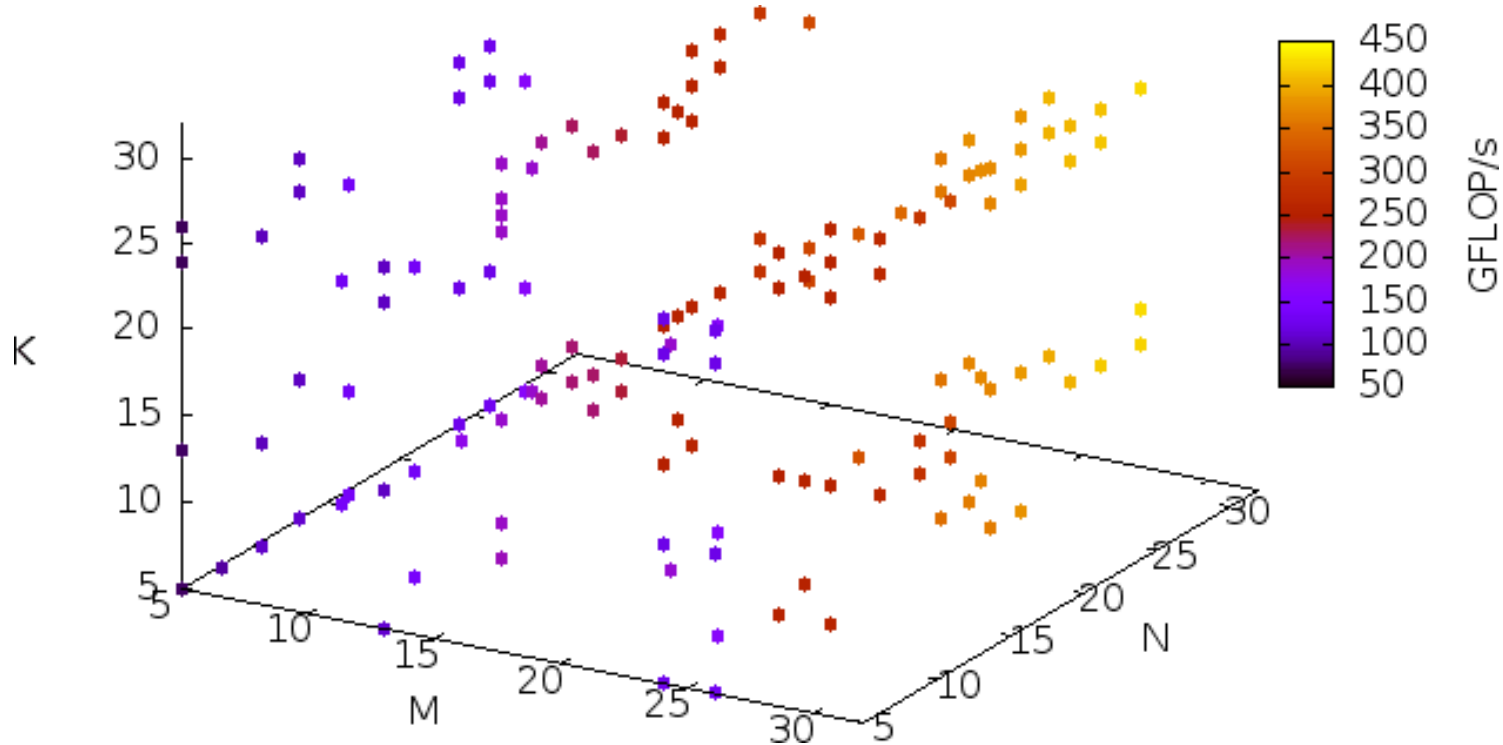


Figure 1: This plot shows the performance (based on LIBXSMM 1.0) for a dual-socket Intel Xeon E5-2699v3 (“Haswell”) of a “compact selection” (to make the plot visually more appealing) out of 386 specializations as useful for CP2K Open Source Molecular Dynamics [1]. The code has been generated and built 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] <http://software.intel.com/xeonphicatalog>: Intel Xeon Phi Applications and Solutions Catalog.

[3] <http://goo.gl/qsnOOof>: Intel 3rd Party Tools and Libraries.

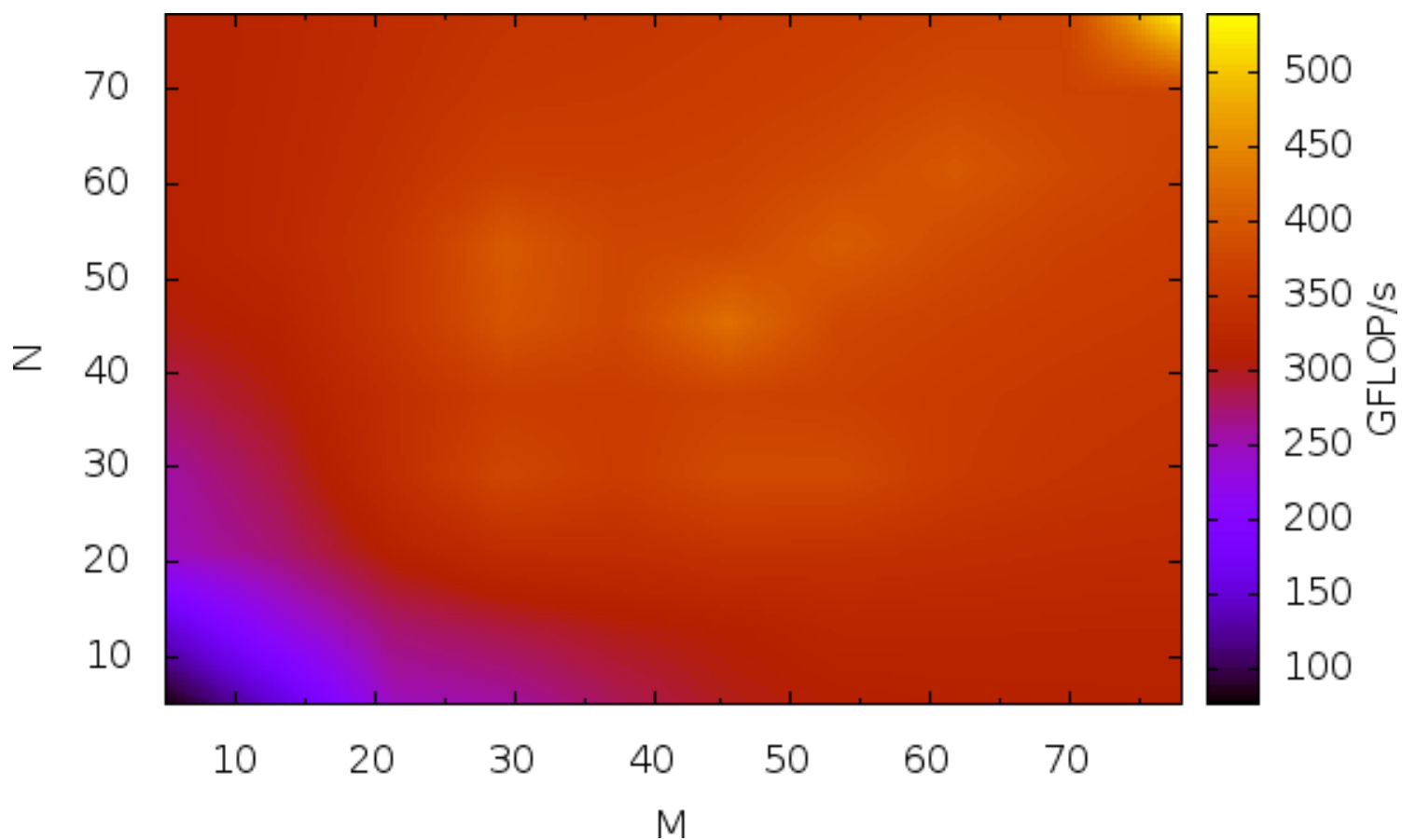


Figure 2: This plot summarizes the performance (based on LIBXSMM 1.0) of the generated kernels by averaging the results over K (and therefore the bar on the right hand side may not show the same maximum when compared to other plots). The performance is well-tuned across the parameter space with no “cold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

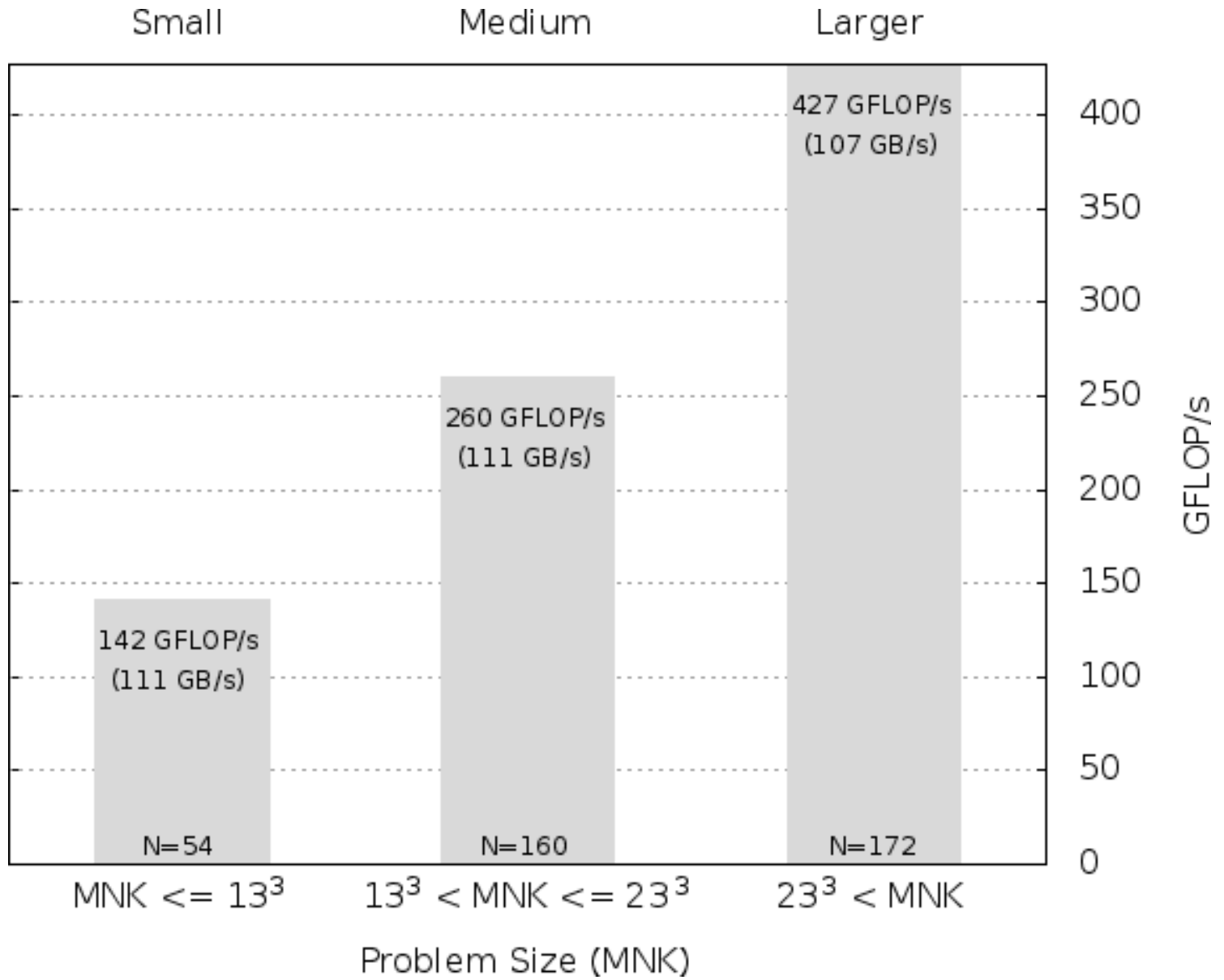
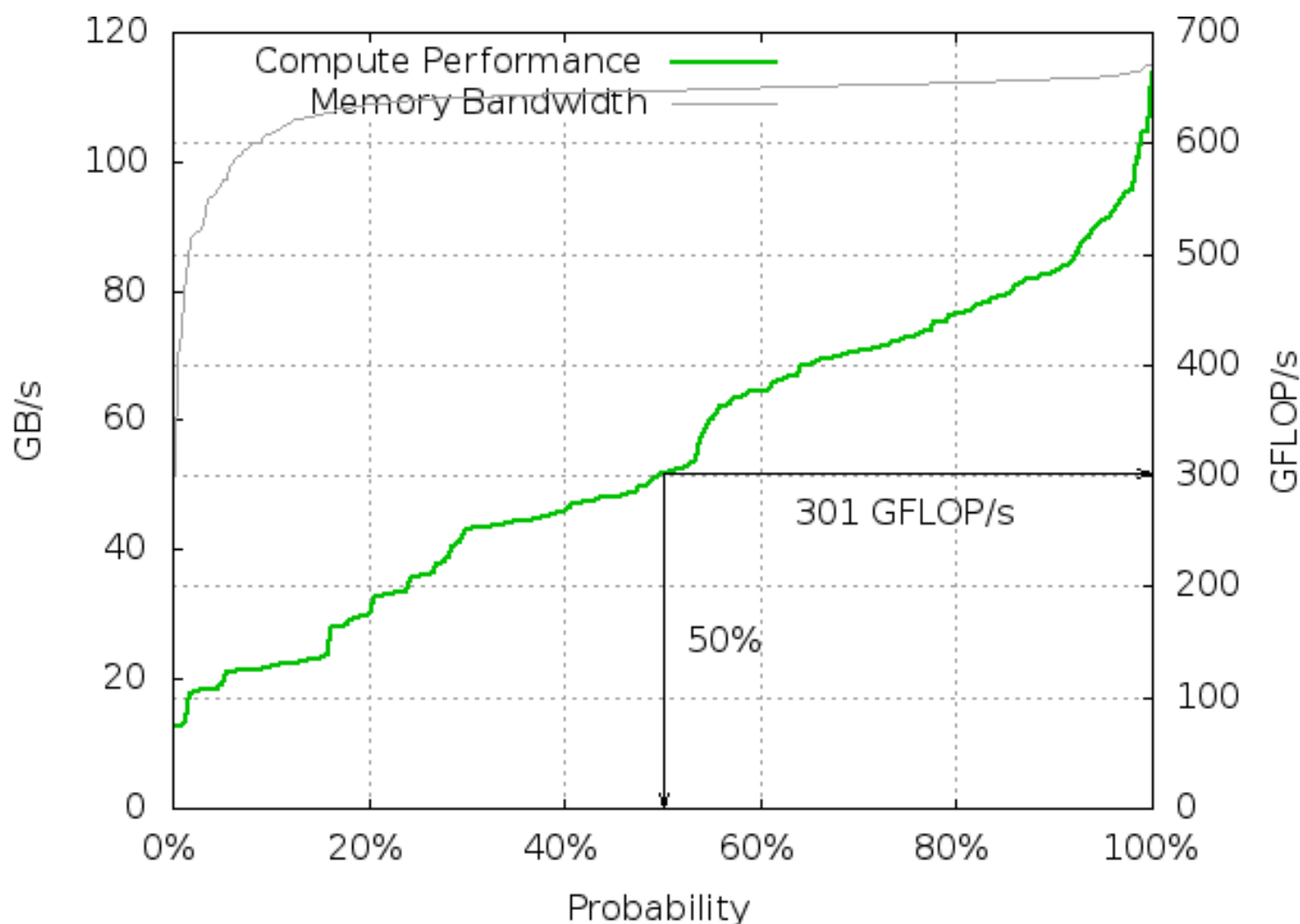


Figure 3: This plot shows the arithmetic average (non-sliding) of the performance (based on LIBXSMM 1.0) with respect to groups of problem sizes (MNK). The problem sizes are binned into three groups according to the shown intervals: “Small”, “Medium”, and “Larger” (notice that “larger” may still not 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.