LIBXSMM: A High Performance Library for Small Matrix Multiplications

Alexander Heinecke[1], Hans Pabst[2], Greg Henry[3]

[1] Intel Corporation, Intel Labs, Mission College Boulevard 2200, Santa Clara 95054, CA, USA
[2] Intel Semiconductor AG, Software and Services Group, Badenerstrasse 549, 8048 Zurich, Switzerland
[3] Intel Corporation, Software and Services Group, 2111 NE 25th Avenue, Hillsboro 97124, OR, USA

Introduction

Small dense and small sparse matrix multiplications play an important role in modern high performance scientific applications, such as high-order FEM codes, block-sparse compressed row/column subroutines and sparse direct solvers using super-blocks, to mention just a few. Our proposed solution: LIBXSMM. 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 future Intel Xeon Phi processor family and future Intel Xeon processors. Hereby a small problem is characterized by the M, N and K parameter of the corresponding matrix-matrix multiplication. LIBXSMM is best suitable for problem sizes where $\sqrt{M \times N \times K} < 80$. LIBXSMM is available as free software at https://github.com/hfp/libxsmm.

General Design and Interface

LIBXSMM achieves its high application-level performance by a modular design providing a frontend (high level language interface, and routine selection), and a backend part (application specific xGEMM code generation).

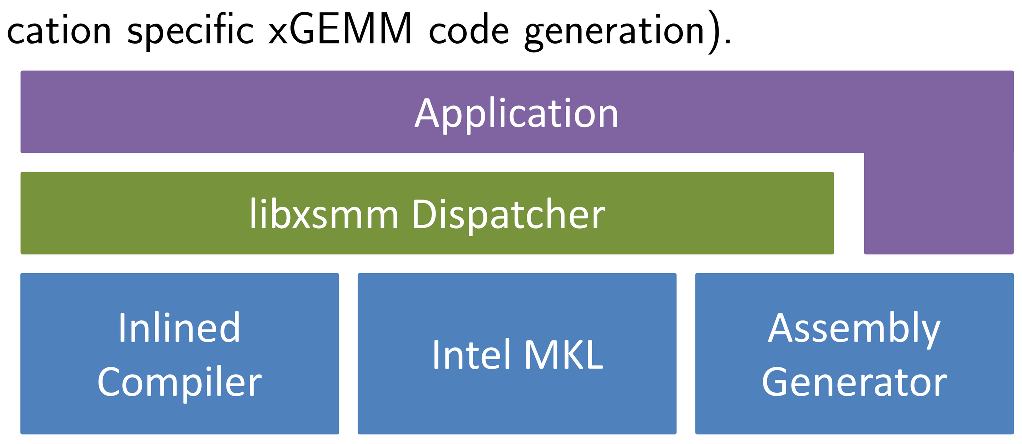

Figure 1: High-level software architecture of LIBXSMM.

Currently, LIBXSMM supports three high-level frontends:

- C, functions for single and double precision (Fig. 2)
- C++ using polymorphism (overloaded fn., templates)
- FORTRAN using polymorphism (overloaded routines)

```
/** Automatically dispatched 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 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 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);
/** If non-zero function pointer is returned,
call LIBXSMM's assembly routine.  */
libxsmm_smm_function libxsmm_smm_dispatch(int
m, int n, int k);
libxsmm_dmm_function libxsmm_dmm_dispatch(int
m, int n, int k);
```

Figure 2: Simple C language S/DGEMM interface of LIBXSMM.

In addition to this simple interface LIBXSMM also offers calls with full S/DGEMM interface to ensure a very simple integration. This interfaces also supports various values for α and β as well as leading dimensions which differ from M,N,K.

Performance

We have evaluated the performance in the context of CP2K, and in particular DBCSR, http://dbcsr.cp2k.org/ [1], and SeisSol https://github.com/SeisSol/SeisSol/ [2].
Our test platform is a dual-socket Intel Xeon E5-2699v3 ("Haswell") machine with 36 cores reaching 118 GB/s memory bandwidth in STREAM Triad, and 1.1 TFLOPS compute performance with large DGEMMs. All measurements are based on Version 1.0 of LIBXSMM, https://github.com/hfp/libxsmm/releases.

CP2K

The performance depends on the workload, and therefore our results consist of 386 important code specializations which are binning into three groups of different problem sizes.

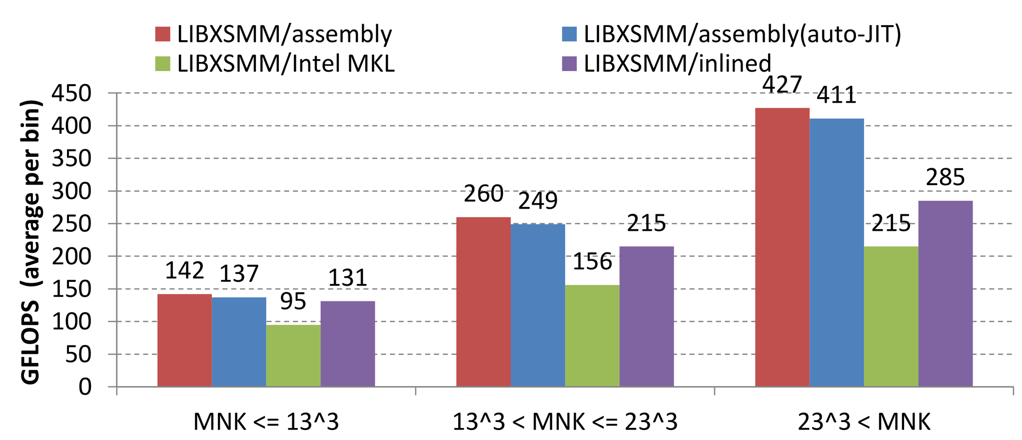

Figure 3: This plot shows the average of the performance with respect to groups of problem sizes (M,N,K). The bins correspond to certain arithmetic intensities: [0.4, 1.1], (1.1, 1.9], and (1.9, 4.5] in DP-FLOPS/Byte.

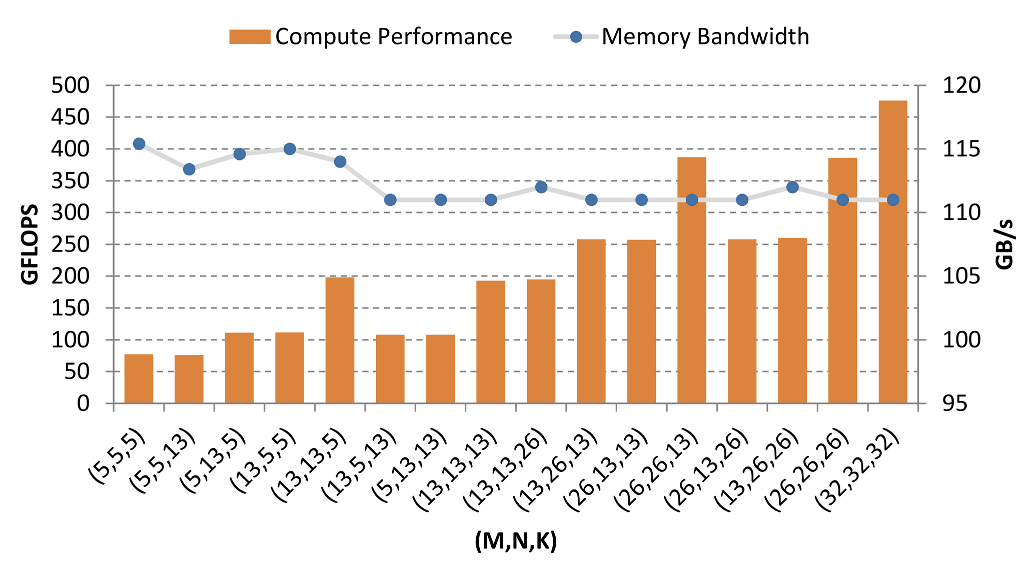

Figure 4: This plot shows selected problem instances where kernels with lower arithmetic intensity are bottlenecked by memory bandwidth.

SeisSol

The earthquake code SeisSol is based on a high-order discontinuous Galerkin (DG) method. Its cell-local routines boil down to small sparse and dense matrix multiplications. The performance plots show one of the most important routines (multiplication with stiffness and flux matrices) and full application performance for several convergence orders.

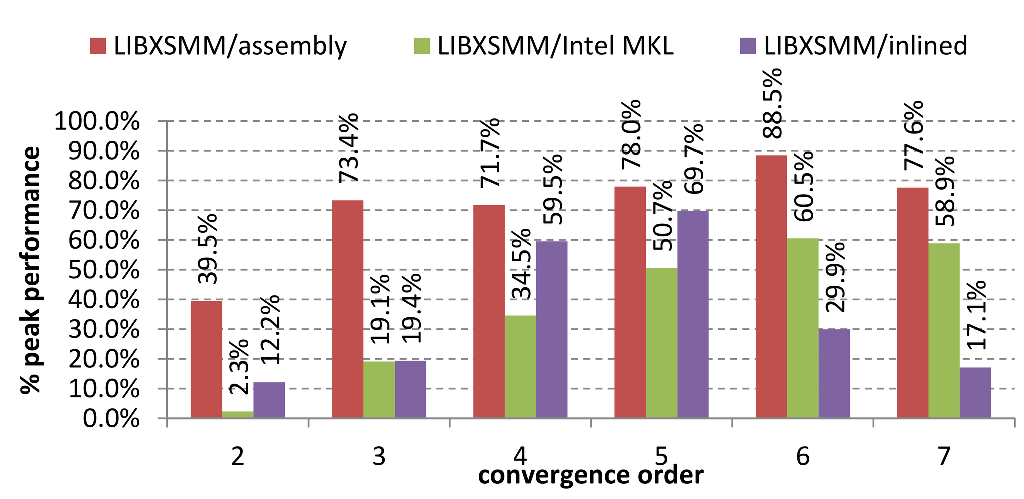

Figure 5: SeisSol's kernel performance for $M = K = \{4, 12(10), 20, 36(35), 56, 84\}$ and $N = 9$.

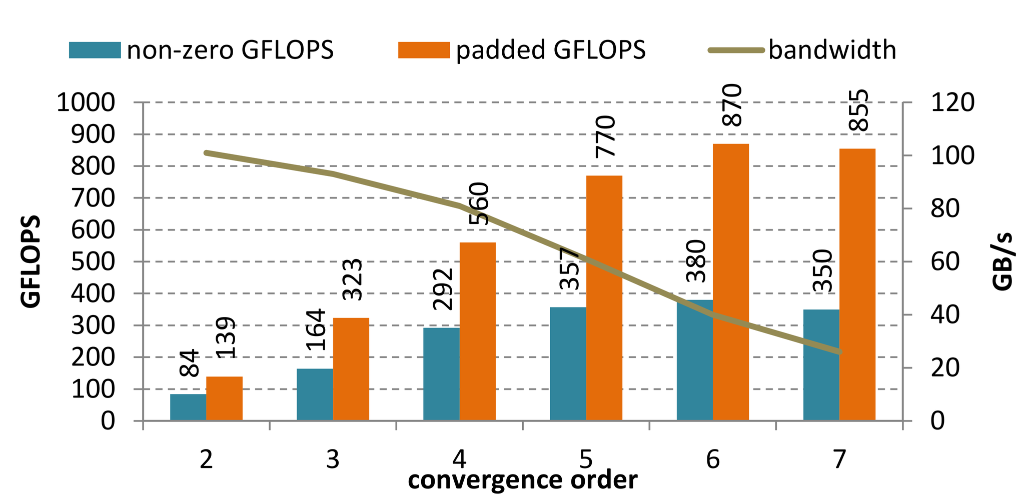

Figure 6: SeisSol performance of the LOH.1 benchmark (all FEM matrices dense, 1.1% slower than [3]).

Implementation

Frontend

LIBXSMM implements a three-level dispatch mechanism which helps executing the potentially best-performing implementation by finding specialized code for a particular problem instance (M,N,K):

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

All three levels are directly accessible allowing to customize the mechanism (Fig. 2). The library also allows to amortize the cost of the dispatch when multiple calls with the same M, N, and K are needed. Moreover, the threshold determining when to fallback into the BLAS implementation can be adjusted.
The assembly-code selection is based on a hashtable using CRC32 checksums (whose calculation can be accelerated by SSE4.2 instructions). Furthermore, the frontend features an experimental auto-just-in-time compilation option, which allows to build needed kernels on the fly. This includes an implementation of a JIT-code cache to keep overheads as low as possible.

Assembly Backend

The assembly generator's implementation follows ideas well-known in large xGEMMs, but due to the small sizes we cannot employ copy routines to build optimal A or B panels. Therefore it derives several highly performing micro-kernels which are orchestrated to form a small xGEMM operation. Specifically, our backend relies and on set of $M \times N = \{1S, 0.5V, 1V, 2V, 3V, 4V\} \times \{1, 2, 3, 4\}$ micro-kernels in case of SSE,AVX and AVX2. S stands for scalar execution and V for vector-register length. This ranges from 2 to 8 depending on instruction set and precision.

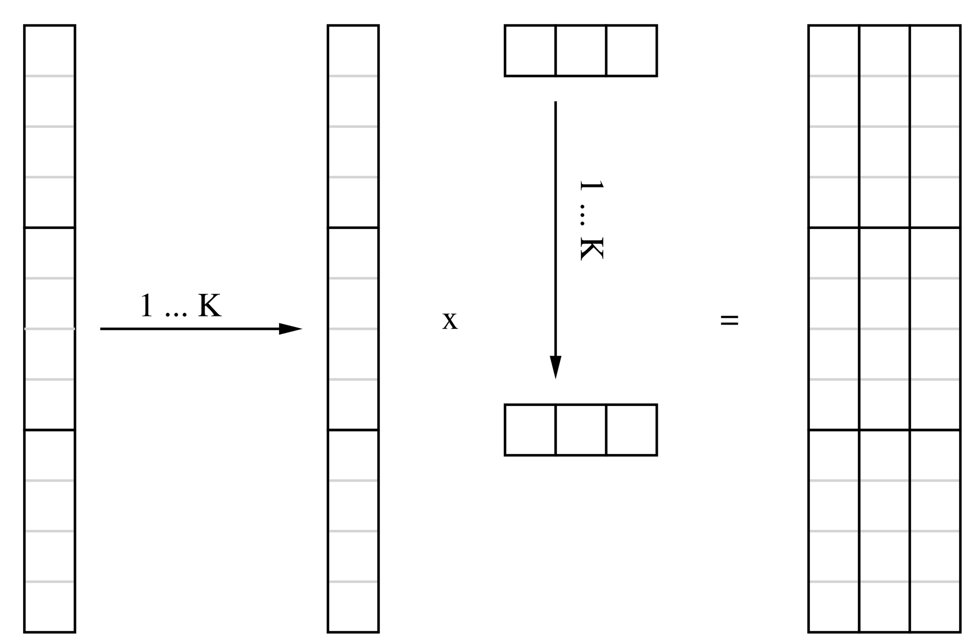

Figure 7: schematic $M \times N = 12 \times 3$ micro-kernel.

Summary/Outlook

LIBXSMM ...

- ...achieves optimal performance for wide range of small matrix multiplications on latest Intel Xeon E5v3 processors.
- ...achieves much better performance with respect to DRAM bandwidth and peak FLOPs than vendor libraries.
- ...is freely available and includes examples for usage.

Current Research:

- adding a runtime auto-tuning component for both dispatching and micro-kernel composition

References

[1] U. Borštnik *et al.*, *Sparse matrix multiplication: The distributed block-compressed sparse row library*, Parallel Computing **2014**, 40, 47-58.
[2] A. Heinecke *et al.*, *Petascale High Order Dynamic Rupture Earthquake Simulations on Heterogeneous Supercomputers*, SC14, Gordon Bell Finalist **2014**, 3-14.
[3] A. Breuer *et al.*, *High-Order ADER-DG Minimizes Energy- and Time-to-Solution of SeisSol*, ISC15, **2015**.