LIBXSMM: A High Performance Library for Small Matrix Multiplications

Alexander Heinecke*, Hans Pabst† and Greg Henry‡
*Intel Corporation, 2200 Mission College Blvd., Santa Clara, 95054, CA, USA
†Intel Semiconductor AG, Badenerstrasse 549, 8048 Zurich, Switzerland
‡Intel Corporation, 2111 NE 25th Avenue, Hillsboro, 97124, OR, USA

***Abstract*—In this work we present a library, LIBXSMM, that provides a high performance implementation of small sparse and dense matrix multiplications on latest Intel architectures. Such operations are important building blocks in modern scientific applications and general math libraries are normally tuned for all dimensions being large. LIBXSMM follows a matrix multiplication code generation approach specifically matching the applications' needs. By providing several interfaces, the replacement of BLAS calls is simple and straightforward. We show that depending on the application's characteristics, LIBXSMM can either leverage the entire DRAM bandwidth or reaches close to the processor's computational peak performance. Our performance results of CP2K and SeisSol therefore demonstrate that using LIBXSMM as a highly-efficient computational backend, leads to speed-ups of greater than two compared to compiler generated inlined code or calling highly-optimized vendor math libraries.**

I. Introduction and Overview

Small dense and sparse matrix multiplications play an important role in modern high performance scientific applications, such as high-order FEM codes, blocked compressed sparse row/column subroutines and sparse direct solvers using super-blocks, to mention just a few. It is well-known that for problem sizes of $\sqrt[3]{M \cdot N \cdot K} <\approx 80$ high-performance libraries such as the Intel® Math Kernel Library do not deliver optimal performance. Furthermore, the compiler generated code of the corresponding triple-nested loops suffers from performance problems as well if $\sqrt[3]{M \cdot N \cdot K} > 4$. Here, regular optimization routines are not optimal in case of xGEMM. To address this problem, several codes (CP2K, in particular DBCSR, http://dbcsr.cp2k.org/ [1], SeisSol https://github.com/SeisSol/SeisSol/ [2], MADNESS https://github.com/m-a-d-n-e-s-s/madness [3], [4]) have developed application-specific solutions in the past.

The goal of LIBXSMM is to replace these application-specific solutions with an optimized general purpose library for small matrix multiplications.

II. Implementation

LIBXSMM achieves its high application level performance by a modular design such that it can be used in a very flexible way: LIBXSMM separates into a frontend (routine selection) and backend (specific xGEMM code generation).

A. Frontend

The frontend handles the selection of the actual backend implementation. To serve as an optimal drop-in replacement LIBXSMM has three different backend options: (1) its own assembly code generator, (2) compiler generated and inlined code and (3) calling Intel's MKL (sequential) routines.

LIBXSMM offers two ways of selecting from the available backends: (a) through renamed BLAS and modified BLAS function calls and (b) an auto dispatcher which decides which backend should be executed for the given parameter set. The overhead of auto-dispatched multiplications becomes negligible with reasonable problem sizes (above $\approx 20 \times 20$ matrices), but may be significant for very small auto-dispatched matrix-matrix multiplication. In such cases, the user has the option to employ (a) and to directly call one of the three backend implementations. If an "unknown" size is called, LIBXSMM offers an experimental feature which calls the assembly code generator and builds the needed kernel just-in-time (JIT) by emitting the kernel's byte-sequence into an executable code buffer. As explained in the next paragraph, there is also the possibility to bypass the frontend entirely (targeting power-users) and to call the generated assembly code directly and to leverage features which are not part of the standard BLAS interface.

B. Backend

The compiler generated backend is fully inlined into the frontend to keep overheads as low as possible. The Intel MKL backend is just a redirected call. Our assembly code generation as this targets the critical problem sizes between 4 and 70 and is now discussed in more detail.

The generator's implementation follows ideas from [5] and applies them in a modified fashion to the small xGEMM problem. This means that due to the small sizes we cannot employ copy routines to build optimal A or B panels, but we derive 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. Here S stands for scalar execution and V for vector-register length which ranges in from 2 to 8 depending on instruction set and requested precision. Additionally, our assembly backend optionally inserts software-prefetch instruction into a kernel which can be used to prefetch matrix operands across xGEMM operations boundaries.

Furthermore, our generator backend is able to vectorize small sparse xGEMM operations, by hard-wiring their sparsity

TABLE I
CP2K PERFORMANCE USING LIBXSMM.

problem size	GFLOPS	GB/s
$\sqrt[3]{M \cdot N \cdot K} \leq 13$	142	111
$\sqrt[3]{M \cdot N \cdot K} \leq 23$	260	111
$\sqrt[3]{M \cdot N \cdot K} > 23$	427	107

pattern into LIBXSMM's code. This is desirable in case of high-order FEM codes to reduce time-to-solution even further where the small matrix operands can be (very) sparse.

Note that, the mentioned optimal sparse kernel routines and prefetching across xGEMM operations is currently not supported in our high-level interface discussed above as it would add significant overheads to the dispatching. Nevertheless, power-users can leverage these features by directly calling the generator's code and bypassing LIBXSMM's standard interface.

III. PERFORMANCE SUMMARY

We evaluate the performance of LIBXSMM in the context of CP2K and SeisSol. In case of CP2K we employed LIBXSMM's regular interface whereas SeisSol makes use of LIBXSMM's power-user interface as prefetches across xGEMMs and sparse routines are needed.

A. CP2K

CP2K is a program to perform atomistic and molecular simulations of solid state, liquid, molecular, and biological systems. It provides a general framework for different methods such as e.g., density functional theory (DFT). CP2K's DBCSR component requires very fast execution of small DGEMMs as these routines are used inside a library for parallel operations on a distributed block CSR (DBCSR) data-structure. We implemented a performance reproducer which is within a low single-digit percentage of the full CP2K application's performance and enabled it to leverage LIBXSMM. A performance summary is shown in Tab. I. As CP2K requires up to 386 different kernels, we bin them in to three groups and show the average performance per group. In case of the small and medium bin we are able to achieve stream triad bandwidth of the dual socket Intel® Xeon® E5-2699v3 test system (118 GB/s stream bandwidth and 1.1 TFLOPS large dgemm performance). The large bin, which is also partially memory-bound, reaches close to 45% peak floating point performance on average.

B. SeisSol

SeisSol is one of the leading codes for earthquake scenarios, in particular for simulating dynamic rupture processes and for problems that require discretization of very complex geometries and was a Gordon Bell Finalist in 2014. SeisSol is based on the discontinuous Galerkin (DG) method with spatial and Arbitrary high order DERivatives (ADER) for time discretization. Its cell-local routines boil down to small sparse and dense matrix multiplications. Table II lists achieved memory bandwidth and floating-point peak performance with respect to the employed convergence for the LOH.1 benchmark [6]. Depending on the order of convergence the DOF matrices are within $\{4, 10(12), 20, 35(36), 56, 84\} \times 9$. Keeping CP2K's results in mind, LIBXSMM is able to achieve close to stream triad bandwidth for low-order executions and more than 70% floating point peak performance for high-order runs on the same dual-socket Xeon E5-2699v3 server. Besides these results the LIBXSMM-enhanced SeisSol version recently sustained more than 1.4 PFLOPS on the 3 PFLOPS SuperMUC Phase system on more than 80,000 cores.

TABLE II
SEISSOL PERFORMANCE USING LIBXSMM.

convergence order	GFLOPS	GB/s
2	139	101
3	323	93
4	560	81
5	770	61
6	870	40
7	855	26

IV. CONCLUSION AND FUTURE WORK

The current status of the work demonstrates that it is possible to implement a highly-efficient small sparse and dense matrix multiplication for Intel processors. This is true for both scenarios: memory bandwidth bound and compute bound usage.

However, there are still design parameters which can be even further optimized. Along these lines auto-tuning of e.g. the composing of different micro-kernels (e.g. $M = 24$ can be built by 16+8 or 12+12) is a future research direction.

REFERENCES

[1] U. Borstnik *et al.*, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Computing*, vol. 40, no. 5-6, pp. 47–58, 2014.

[2] A. Heinecke *et al.*, "Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers," in *SC14*, Nov. 2014, pp. 3–14, gordon Bell Finalist.

[3] K. Stock *et al.*, "Model-driven simd code generation for a multi-resolution tensor kernel," in *IPDPS 2011*, May 2011, pp. 1058–1067.

[4] J. Shin *et al.*, "Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology." Springer New York, 2010, pp. 353–370.

[5] K. Goto *et al.*, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1356052.1356053

[6] A. Breuer *et al.*, "High-Order ADER-DG Minimizes Energy- and Time-to-Solution of SeisSol," in *ISC 2015*, Heidelberg, Jul. 2015.