LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation

Alexander Heinecke*, Greg Henry‡, Maxwell Hutchinson§, and Hans Pabst†
*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
§Department of Physics, University of Chicago, 5720 S. Ellis Ave, Chicago IL, 60637, USA

***Abstract*—Many modern highly scalable scientific simulations packages rely on small matrix multiplications as their main computational engine. Math libraries or compilers are unlikely to provide the best possible kernel performance. To address this issue, we present a library which provides high performance small matrix multiplications targeting all recent x86 vector instruction set extensions up to Intel AVX-512. Our evaluation proves that speed-ups of more than 10 × are possible depending on the CPU and application. These speed-ups are achieved by a combination of several novel technologies. We use a code generator which has a built-in architectural model to create code which runs well without requiring an auto-tuning phase. Since such code is very specialized we leverage just-in-time compilation to only build the required kernel variant at runtime. To keep ease-of-use, overhead, and kernel management under control we accompany our library with a BLAS-compliant frontend which features a multi-level code-cache hierarchy.**

***Index Terms*—Small GEMM, JIT compilation, code generation, Block CSR, FEM, SEM**

I. Introduction

Highly-efficient execution of linear algebra problems is one of the most important needs in modern extreme-scale HPC. This is because nearly all discretization techniques for partial differential equations (PDEs) end up solving a linear system of equations, so fast matrix operations are key for shortening the time-to-solution. Traditionally, these systems are of sparse nature, and solving them is not a scalable endeavor on hundreds of thousands of cores as found in today's fastest machines. In the last two decades, new solver strategies, especially for hyperbolic equations such as wave equations and in the field of computational fluid dynamics (CFD) have been developed. Here, two of the most widely used ones are the discontinuous Galerkin finite element (DG-FEM) and the spectral element (SEM) method. These methods support and aim at a high(er) orders of convergence to make the best use of one grid point invested with respect to accuracy and time-to-solution. Furthermore, they reduce global communication to an absolute minimum which allows efficient scaling to the largest of today's and tomorrow's machines. In contrast to large sparse systems whose solution is memory bandwidth bound, DG-FEM and SEM rely on computational heavy kernels as we will discuss. Depending of the chosen order of convergence, the resulting simulation run is only slightly memory bandwidth bound, at the kink of the roof-line model or (partially) compute-bound. Even with emerging high-bandwidth memory technologies, the gap between compute-power and memory performance is still widening. The new strategies we present offer an opportunity to leverage the increasing computational power of a single chip.

Because these solvers scale well, single node/core performance is a key success factor for a high performing higher-order DG-FEM or SEM package. At higher order, these applications use either BLAS (Basic Linear Algebra Subroutines) or hand-coded matrix routines as their main compute engine. More precisely, they rely on one of the most compute-intense BLAS functionality: general matrix multiplications (GEMM). However, these GEMM calls are of very small size (matrix dimensions in the order of tens) compared to commonly used GEMM benchmarks (matrix orders in the order of thousands). Therefore, well-known and established BLAS libraries, such as the Intel® Math Kernel Library (Intel® MKL), OpenBLAS, ATLAS, BLIS or similar do not deliver the best possible performance, as they are tuned for the aforementioned larger problem sizes, by leveraging tiling and cache blocking. These techniques aim to reduce memory subsystem pressure and optimize computational efficiency [1]–[4]. For small GEMMs, some of these concepts can be reused, but they need to be adjusted. For many small independent GEMMs, the batched GEMM routines have been recently proposed [5]–[8]. However, often they are hard to use, as the scientific application has to be rewritten. But more importantly, a batched interface normally only provides more parallelism at a library level (to make use of multi or many-core systems) and is perhaps factoring out argument checks, but it typically does not improve the actual multiplication within a batch.

DG-FEM and SEM are not the only operations relying on small GEMM operations for best performance. Block-sparse matrix formats such as Block Compressed Sparse Row Format (BSR) [9] can also substantially benefit from fast small GEMMs. These BSR matrices can appear in material science applications denoting electron states [10]. In this case, scientifically relevant matrix shapes may not align with the length of the vector-register-lane.

Beside of these classic HPC fields, the emerging workloads such as deep learning and machine learning are also built up-on small or medium sized GEMMs [11]–[13]. However,

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

evaluating machine learning applications is beyond the scope of this paper. Nevertheless, it will become clear that there is no limitation in LIBXSMM which would block the usage in these scenarios.

Therefore, this article makes the following contributions:

- to the best of our knowledge, it introduces for the first-time a library, LIBXSMM, which seamlessly integrates into existing BLAS libraries and targets high-performance executions of small GEMMs on Intel Architecture systems with AVX2 (Intel Xeon, *Haswell, Broadwell*) and AVX-512 (Intel Xeon Phi *Knights Landing*) extensions.
- it describes a hardware-aware code generator which is able to create optimal code without a lengthy auto-tuning phase since just-in-time (JIT) compilation is leveraged. This allows it to exploit problem dimensions which are normally not available.
- it provides a rich performance evaluation section of LIBXSMM. This includes synthetic benchmarks (single core and GEMM BATCHED) as well as performance results of three former Gordon Bell finalist codes (material-science, DG-FEM and SEM) which already use LIBXSMM today.

The paper is organized as follows: in the next section, the three main components of LIBXSMM are described: (a) frontend, (b) code generation, and (c) JIT compilation. Afterwards in Sect. III, we analyze LIBXSMM's performance whereas Sect. IV relates LIBXSMM to batched GEMM routines, and finally in Sect. V the paper is concluded.

Furthermore, we will use two different hardware platforms for our performance evaluations:

BDX a dual-socket Intel® Xeon® E5-2697v4 processor (*previously code-named Broadwell-EP*) system with 2×18 cores, 2.0 GHz (running at AVX-base frequency), 128 GB of DDR4-2400 memory.

KNL a single-socket Intel® Xeon Phi™ 7250 processor (*previously code-named Knights Landing*) with 68 cores, 1.2 GHz core-clock (running at AVX-base frequency), 1.7 GHz mesh-clock, 16 GB MCDRAM@7.2 GT, 96 GB DDR4-2400, FLAT/QUADRANT memory mode. For details, please refer to [14].

II. Implementation

A. Frontend and User Interface

The terms "frontend" and "backend" are usually used with compilers when referring to a language parser, which builds an intermediate representation (frontend), and the code generator, which translates the intermediate representation into code that can be consumed by hardware (backend). However, with LIBXSMM, the frontend is neither a text parser nor a general programming language but rather an API, and it is also questionable whether the definition fits with a Domain Specific Language (DSL). In fact, the library frontend is binary compatible with a well-known industry standard interface known from LAPACK's BLAS Level-3 GEMM routines. The library backend does not take an intermediate representation but rather assembles the machine code directly, driven by an internal API. A high-level breakdown of the LIBXSMM library is shown in Fig. 1 with the frontend presented in this section, and the backend detailed in the next two sections (code generation and Just-In-Time compilation).

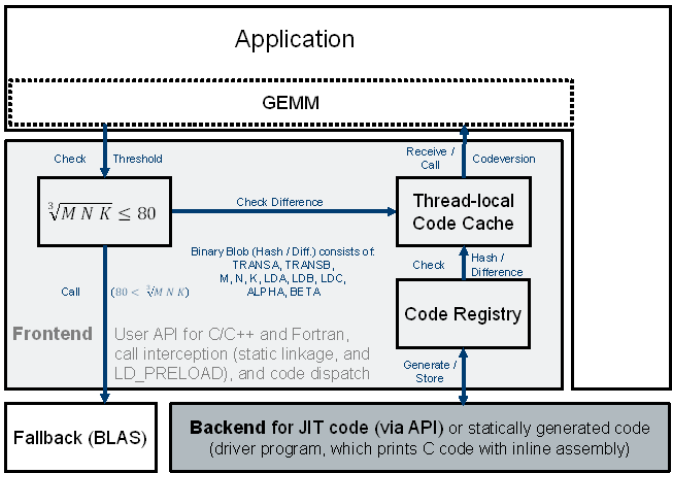

Fig. 1. LIBXSMM frontend (and backend): the user-facing frontend not only provides routines which are binary compatible with LAPACK's BLAS Level-3 GEMM, but also allows queries to a particular kernel f_S for a given set of GEMM arguments S. A specialization S consists of all arguments but the a, b, and c matrices i. e., such a kernel is fully specialized with respect to $S := m, n, k, \ldots$ and can be called with $f_S(a, b, c)$. If called in a compatible fashion, the frontend checks against a configurable threshold (problem size) and eventually queries the code registry for the requested kernel. This is sped up by a high-quality hash (CRC32) based on S, but may also use an exact difference in case of hash key collisions. To account for repeated calls of a limited set of kernels, a very small thread-local cache is maintained based on exact differences. In any case, overhead is not only lowered due to an optimized critical path, but also by avoiding costly synchronization until actually necessary (registry update).

Fig. 2. Kernels which are requested at build time of the library are referred as "statically generated kernels" (as opposed to code which is generated Just-In-Time at runtime of an application). They participate (similar to JIT code) in automatic code dispatch (e. g., during GEMM compatible calls), and explicit code dispatch (querying a code version at runtime). In addition to these services, prototype declarations are added to the generated library interface. The first command gives an example where the M, N, and K parameters are treated like an index sets originating from a triple-nested loop. This relationship ($M(N(K))$) may also resolve to the next non-empty outer index of the loop nest (wraps around from M to K). The example generates eight routines which are "squares" with respect to M and N (N inherits the value of M). The second example follows the MNK-scheme by taking a list of indexes which are eventually grouped (commas). Each group is combined into all possible triplets, and the given example generates nine routines.

```
[1]$ make M="2␣4" N="1" K="$(echo␣$(seq␣2␣5))"
```

Generated index set:

```
(2,1,2), (2,1,3), (2,1,4), (2,1,5),
(4,1,2), (4,1,3), (4,1,4), (4,1,5)
```

```
[2]$ make MNK="2␣3,␣23"
```

Generated index set:

```
(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)
```

Listing 1. Due to supporting C/C++ and FORTRAN, code listings may alternate between C and FORTRAN code (C++ code may be shown as well). For binary compatibility with existing applications (e. g., via `LD_PRELOAD`), the single-precision form of the GEMM interface is shown. Underneath, the subroutine calls present two aspects of the native "look and feel" of the API: (1) subroutines with a type prefix in front of the name ('s' and 'd' for single- and double-precision respectively) as well as a generic interface (FORTRAN 90), and (2) optional arguments to omit values which are not deviating from the (configured) defaults of the library. The latter works without affecting the binary compatibility as claimed earlier (compatible symbols are present in addition). Optional arguments are available with FORTRAN and C++, and they are making it easier to adopt the library beyond the compatible usage model (small matrix multiplications often do not need all aspects of the full GEMM interface).

```
SUBROUTINE libxsmm_sgemm(transa, transb, m, n, k, &
alpha, a, lda, b, ldb, beta, c, ldc)
   CHARACTER, INTENT(IN), OPTIONAL :: transa, transb
   INTEGER(BLASINT_KIND), INTENT(IN) :: m, n, k
   INTEGER(BLASINT_KIND), INTENT(IN), OPTIONAL :: lda
   INTEGER(BLASINT_KIND), INTENT(IN), OPTIONAL :: ldb
   INTEGER(BLASINT_KIND), INTENT(IN), OPTIONAL :: ldc
   REAL(C_FLOAT), INTENT(IN), OPTIONAL :: alpha, beta
   REAL(C_FLOAT), INTENT(IN) :: a(:,:), b(:,:)
   REAL(C_FLOAT), INTENT(INOUT) :: c(:,:)
END SUBROUTINE

CALL libxsmm_sgemm(m=m, n=n, k=k, a=a, b=b, c=c)
CALL libxsmm_dgemm(m=m, n=n, k=k, a=a, b=b, c=c)
CALL libxsmm_gemm(m=m, n=n, k=k, a=a, b=b, c=c)
```

Listing 2. Beyond the compatible usage model, the explicit dispatch functionality allows the caller to check for and to generate a particular matrix multiplication kernel i. e., `NULL` (or similar) may be returned. There are two reasons why a kernel might be unavailable: (1) JIT support has been turned off at build-time of the library, or (2 a) the CPUID flags determine an unsupported instruction set extension, and (2 b) a statically generated code version is not available i. e., this kernel was not requested at build-time of the library. The shown set of functions attempts to be as convenient and as type-safe as possible: the type-agnostic descriptor based form (`libxsmm_xmmdispatch`) enables more advanced use cases while the other form (`libxsmm_smmdispatch`, etc.) expects a detailed and type-safe function signature. The latter form also allows for default arguments (or placeholders in case of the C language i. e., allowing to pass `NULL` where it makes sense). This family of functions not only enables explicit code dispatch but also to successively call (i. e., multiple times) or to keep a particular kernel at hand, and therefore amortizing the cost of the code dispatch.

```
/** Query or JIT-generate a function (descriptor form). */
libxsmm_xmmfunction libxsmm_xmmdispatch(
   const gemm_descriptor* descriptor);
/** Similar to libxsmm_xmmdispatch (single-precision). */
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);
/** Similar to libxsmm_xmmdispatch (double-precision). */
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);
```

The frontend (and user-interface) of the library supports C/C++ and FORTRAN by presenting a consistent API, which is however customized for each of the programming languages. The latter is to provide a native "look and feel", and to adjust for the different language features (see Listing 1). The actual interface files are generated according to build instructions, and are available as code generation template files prior to actually building the library. These files are almost C/C++ or FORTRAN source code but also contain a few tags to be replaced at build time. Moreover, additional subroutine declarations are added for each of the statically specialized kernels. Another aspect of the frontend is the build system since it is not only building the library but allows to optionally define a collection of small matrix multiplications to be included at build time (independent of JIT runtime code generation). Fig. 2 shows two implemented schemes which can be used to specify the M, N, and K parameters of matrix kernels to be statically generated.

The compatible usage model requires (automatic) dispatching of the requested matrix multiplication for each and every call. The actual dispatch mechanism (detailed later) has been highly tuned, however an additional function allows the caller to query or generate a particular kernel, and thereby amortizes the cost of the code dispatch (see Listing 2). Beyond amortizing the cost, this explicit mechanism also allows building a customized code dispatch e. g., to decide when and where to call a fall back function. This way, larger matrix multiplications may fall back to a vendor-optimized BLAS library. The LIBXSMM library already does this for problem sizes beyond a compile-time configurable THRESHOLD (80^3) where this problem size (MNK) is characterized by the M, N, and K parameters of the multiplication according to the GEMM interface. The range of problems that falls under the term "small" is approximately given by the L2 cache size of a processor, but may reach a bit further until a regular BLAS library should take over in terms of performance.

The code dispatch mechanism is a frontend service which is distinct from purely assembling ("compiling") machine code in the computer's memory. This mechanism provides: (1) to quickly determine (in a thread-safe fashion) whether the code version is already generated or not, (2) to enter a locked region if the code version is not yet assembled, (3) to allocate executable buffers to hold the machine instructions, calling the backend to actually generate the code version, and (4) to keep the code version on record for a future code dispatch. Each of the stages has been optimized to make an automatic code dispatch not only possible but worth using it (e. g., for GEMM-compatible function calls). For example, the first stage has been tuned to only rely on atomic instructions. The second uses an advanced technique or lock-grading (mapping a checksum of the GEMM descriptor onto a set of locks) to not disable concurrent code generation. The third manages low-level memory allocation of executable buffers across the supported operating systems. Last but not least, the fourth maintains a code registry by calculating checksums in an accelerated fashion (CRC32 instruction based as available since Intel SSE 4.2), and resolves hash key collisions efficiently.

To further optimize dispatching a code version, the library maintains a small cache of the most recently used kernels. This cache size (power-of-two) is meant to only cover a few code versions because the GEMM descriptors acting as a key for the code versions (function pointers) need to be compared against the incoming descriptor i. e., in case of no (cache-)hit the entire set of cached descriptors has to be checked. This comparison is

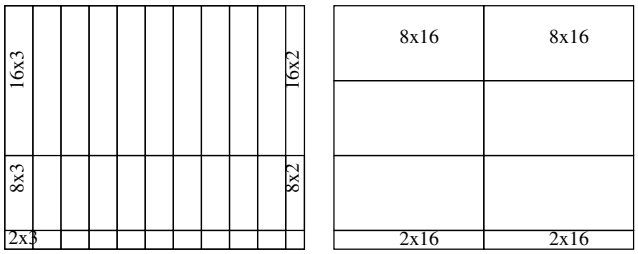

Fig. 3. Schematic sketch of micro tiles used in the AVX2 kernel (left) and AVX512 (right) for a $M \times N = 26 \times 32$ output matrix.

relatively costly due to the size of the descriptor, which needs to cover all non-actual GEMM arguments or "parameters". The cache mechanism also maintains an index of the last hit which is only updated whenever a hit occurs, therefore calling the very same kernel in a consecutive fashion will immediately hit the correct code version by only checking against a single descriptor. This index also provides an informed strategy to evict an entry (because the farthest entry can be replaced). In order to avoid any synchronization primitives (including atomic memory operations), the cache is independently available for each thread (thread-local storage).

While JIT code generation is done according to the CPUID flags, statically generated kernels are only registered if JIT has been disabled, or if there is no better instruction set extension available than what was used at (static) build-time. To further support deploying the library into an unknown environment, all performance critical but optimized code paths (using extensions beyond the level used at build time) are dispatched according to the CPUID flags. Overall, there are various details which add to the performance of the frontend e. g., by exploiting the property of Mersenne prime numbers being next to power-of-two numbers (POT), integer divisions (due to the modulo operator when mapping into the registry) are lowered to bitwise operators.

B. Code Generation

We have described how we manage different GEMM kernels under the regular BLAS API to allow a simple integration into existing software. The focus of the following paragraphs is to describe how we generate the respective kernels' instruction stream. We cover the two latest x86 vector instruction set extensions, AVX2 and AVX-512. The code generation component supports three different output formats: inline assembly (*.h, *.c, *.cpp), pure assembly (*.s-file) and encoding at runtime (discussed separately in the next section). The code generator supports arbitrary M, K, N, lda, ldb and ldc. This flexibility is needed so that we can mix-and-match the various different (sub)-matrix shapes which occur in HPC applications. To address odd-shaped sizes, we support masking instructions. This avoids unwanted application-side padding. Please note, the following description is done with double precision in mind. The generator works analogously for single precision by doubling the number of elements per vector-register.

AVX2 targeting the Intel Xeon processor: The AVX2 instruction set extension is the most recent extension of Intel Xeon processors. It features full support for integer operations for 256-bit vector types and fused-multiply-add (FMA) instructions for floating point numbers. The FMA is of huge interest for matrix operations such as GEMM. Whereas the dimensions of the GEMM microkernel for register blocking do not play an important role in case of large GEMM (we simply use the fastest one), for the small sizes the microkernel's shape is essential to reach high performance. However, it is possible to apply the design concepts of large GEMMs to small GEMMs when selecting a suitable microkernel. These microkernels are based on an outer-product formulation which targets the following result blocks ($M_g \times N_g$) in matrix C: $\{16, 12, 8, 4, 2, 1\} \times \{1, 2, 3\}$. The size of 16×3 is determined by the AVX2 instruction set extensions which offer 16 four-element-wide vector registers called `ymm0-15`, assuming double precision. To store this C-result buffer, 12 ($4 \cdot 3$) out of the 16 registers are needed. Additional three registers hold broadcasts of the kth row of three columns of B. The remaining last register is a ring buffer, containing 4 entries, for 16 rows of column k of A. This leads to an optimal usage of all 16 available registers. As the vector register length in AVX2 is four, best efficiencies are expected for $M \mod 4 = 0$. Therefore for the cases $\{12, 1\} \times \{1, 2, 3\}$ which are used to pad the last rows, a performance hit is expected due to unaligned loads, cf. Fig. 3. The load instruction performance inside the GEMM microkernel is lowered by unaligned loads as they suffer from cacheline splits (vector loads and stores of data spanning across cacheline boundaries). This issue is hardware-related and cannot be worked-around in LIBXSMM. In order to avoid performance degradations stemming from unaligned loads, applications can use zero-padding of their data structures to enforce aligned kernels. This padding is a general application performance optimization trick.

AVX512 targeting the Intel Xeon Phi processor: The ideas discussed in the last section can be easily extended to the AVX512 instruction set extensions. These new instructions offer several possibilities (e. g. fusing a memory broadcast as one operand into an arithmetic instruction) such that better performance can be achieved, especially in case of small GEMMs. There are twice as many and twice as long registers in AVX512, so our AVX2 kernel can increase to 48×3 (at maximum). This size can no longer be efficiently blocked due to the skewed tile-size. Furthermore, the employed 2D-blocking is not optimal on the Xeon Phi x200 family. Due to its relationship to Intel's Silvermont architecture, the out-of-order pipeline is only two-issue wide with two FPUs in the backend. Therefore, any instruction that is not of arithmetic nature decreases the computational efficiency of the core. It's desirable to use a tall-and-skinny or short-and-wide blocking in the DGEMM kernel to only have one load instruction to A or B and then a series of Fused-Multiply-Adds which have a fused memory operand. As we have 32 vector-registers available, we work in all cases on max. $N_g < 30$ columns of B and C simultaneously. We use a "short-and-wide" micro-

kernel: we load 8 rows of A of column k into a register and then perform up-to N_g FMA instructions which broadcast the kth row of all N_g columns on the fly. After having processed all columns of A/rows of B we hold a $8 \times N_g$ sub-matrices of C in N_g accumulator registers where stored back to all N_g columns of C. If the N parameter of the request GEMM exceeds N_g, we evenly divide N and handle any remainder separately, c.f. Fig. 3. If the M parameter is not a multiple of 8, we leverage AVX512 masking options to only compute the desired rows of C. That means all loads from A and all loads and stores referencing C are simply annotated by the masking register `k1` which can be initialized cheaply at the beginning of a GEMM kernel. The mask variants of instructions do not run substantially slower than their unmasked counterpart, so only performance disadvantages because of unaligned memory accesses remain.

In order to obtain good performance on Xeon Phi and its AVX512 implementation, we apply two enhancements to the microkernel of our small GEMM. They are required because the enhanced Silvermont core still has limitations which need to be taken into consideration: a) only up to 16 bytes can be fetched per cycle from the instruction cache, b) the memory pipeline of KNL is in-order.

Both need be addressed by implementing the kernel thoughtfully. Issue a) is problematic since we cannot afford to restructure our data in an optimal way as it is normally done for large GEMMs. We therefore have strided accesses (stride ldb) when reading B in broadcast fashion. Even for mid-order runs ldb is bigger than 16, resulting in an offset larger than $128 \cdot 8$ byte between columns (AVX512 offers a compressed displacement encoding which multiplies the offset by the datatype size). In such a case the length of the FMA instruction increases from 7 to 10 bytes and instructions of the increased length cannot be fetched on a sustained basis from the instruction cache. However, the instruction size can be reduced to 8 bytes per FMA if the x86 SIB (scale index base) addressing mode is utilized. Since we have spare general purpose registers and we need to express 9 streams, we therefore can load ldb to a register and use different base registers to assemble all nine column streams. Due to the scaling in the SIB mode, $\{1,2,4,8\}\times$ ldb can be added for free. The different base registers hold pointers to the first, fourth and seventh column of B. Every 128th k we need to increase these pointers by 16 to stick in the one byte offset range.

Issue b) is addressed by pipelining the loads of rows per column k of A. This is easily doable as we left 4 register intentionally free ($N_g \leq 30$). To even hide the L2 latency of 17 cycles we use a 2-register ring-buffer of A column-vectors. Depending on the size of N_g this ring buffer can be increased up to six entries as we have more spare registers to hide L2 latencies for the smaller cases.

C. Just-In-Time compilation

Every x86 assembly instruction is encoded with a series of hexadecimal machine opcodes (usually two to eight bytes in length not counting memory displacements). Our objective in an effective JIT approach is mapping the assembly instructions into these opcodes. However, we need a quick implementation. One does not want to do file I/O or invoke a compiler. After all, one could compile the assembly instructions and disassemble the object code to find the opcodes for each instruction, but that would be way too slow.

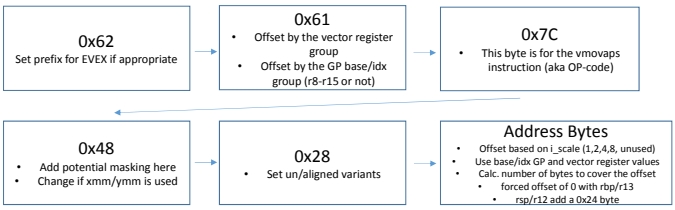

Fig. 4. Byte manipulations of the `vmovaps` instruction needed to encode an arbitrary variant.

We know, based on the theoretical memory and floating point bandwidth, how fast a GEMM should perform. We do not know how many cycles one should spend compiling an assembly instruction into machine opcodes for a JIT approach. But if an instruction uses two general purposes registers and one vector register, it should take some time proportional to the time required to query the values of those various registers and not a lot more.

We consider encoding an assembly instruction like "vmovaps 256(%rax,%rcx,2), %ymm16" (AVX2). For this assembly example, the opcode byte sequence is eight bytes long at: 0x62,0xE1,0x7C,0x28,0x28,0x44,0x48,0x08. Fig. 4 shows the AVX512 case, when zmm registers are used, or at least one of the xmm and ymm registers used are in the range 16-31. The second byte starts as 0x61, but gets adjusted to 0xE1 because %ymm16 is in the 3rd group of eight registers. The third byte gets reduced by 0x20 because the ymm registers are used. The sixth byte and beyond become 0x44, 0x48, and 0x08. The 0x44 is used because the displacement 256 can be expressed with a single byte. The 0x48 partially reflects the i_scale parameter of 2 as well as which GP registers are used. The 0x08 is explained below.

We call a routine that takes properties such as the displacement factor 256, or the scaling factor 2, and tells us which integer registers are used and which floating point registers are used. Any fast routine must examine each parameter at least once and adjust each byte that this parameter impacts. And in this example, %rax impacts the seventh byte. Changing %rax into %rcx changes the seventh byte from 0x48 to 0x49.

An alternate, but faster way to encode opcodes, would be to block multiple assembly instructions at once. This is faster because adjusting one byte at a time is slower than storing four-byte integers or eight-byte doubles. But encoding multiple instructions at once might be harder to maintain. If each instruction is mapped to a series of opcodes, then when creating new code, one has to only consider the individual assembly instructions and not create entirely new block operations.

So we sacrifice a little performance on the opcode creation by considering our problem at an instruction level instead

of a block operation level. But we hope the resulting code is easier to understand, use, and expand upon. Beyond that small performance sacrifice, we wanted to minimize any other negative performance impact.

Most instructions are put together in groups of eight. Every register, integer or floating point, uses these groups. There are sixteen integer registers, so there are two groups of eight. We need to know which group of eight the register is in and where it is in that group in order to find the impact of that register on the final opcode sequence. If i_gp_reg_base is the integer register parameter, we need to query if i_gp_reg_base is in the first group of eight or the second and where it is in that group. We need to consider then (i_gp_reg_base % 8 and i_gp_reg_base less than 8). Sometimes a change can be drastic. For instance, in our example, changing %ymm16 into %ymm15 changes every byte in the sequence of opcodes. The integer registers can be mapped as follows: (%rax = 0, %rcx = 1, %rdx = 2, %rbx= 3, %rsp=4, %rbp=5, %rsi = 6, %rdi=7, %r8 = 8, %r9 = 9, %r10 = 10, %r11 = 11, %r12 = 12, %r13 = 13, %r14 = 14, %r15 = 15.) The relationship between %rax and %rcx is always one, but it might be scaled by eight. If it is scaled by eight, then the opcode difference between %rax and %rdx will always be sixteen.

We had to pay special attention to %rsp and %rbp as well as their eight-byte mirrors %r12 and %r13, as there are some offsets that are handled differently when these registers are used. We also had to pay attention to which group of eight is used. For instance, when %ymm16 is used in our example, the displacement 256 is really considered as eight units of sixty-four bytes and the only number stored is 0x08 (the last byte in the sequence.) Whereas when %ymm15 is used instead, then 256 is taken literally, and four bytes are used to store it.

By examining which group of eight we are in, and where we are, we know the final opcode sequence. An implementation that just touches these parameters and asks only a couple ifs per parameter must be ideal. At least, ideal when just considering a single assembly instruction at a time, as shown in Fig. 5.

In Fig. 5, the duration to create the JIT code is shown in the number of DGEMM calls possible when using Intel MKL. If we compare against the time it takes to run LIBXSMM's DGEMM kernel, the results would be even better and the corresponding crossover points is reached earlier.

III. Application-based Performance Evaluation

After the detailed implementation description of the last section, we analyze LIBXSMM's performance under several conditions. We start with a simple synthetic analysis of the code quality in the next sub-section. This is followed by application-driven evaluations, based on CP2K [15], [16], SeisSol [17] and Nek5000/NekBox [18]. We close the performance analysis by demonstrating that the target applications still exhibit excellent scaling after lifting their single node performance. Please note, we are restricting ourselves to double precision in this evaluation, however LIBXSMM's single precision implementation offers similar speed-ups if the number of rows is divisible by SIMD length for both, single and double, single precision doubles the throughput.

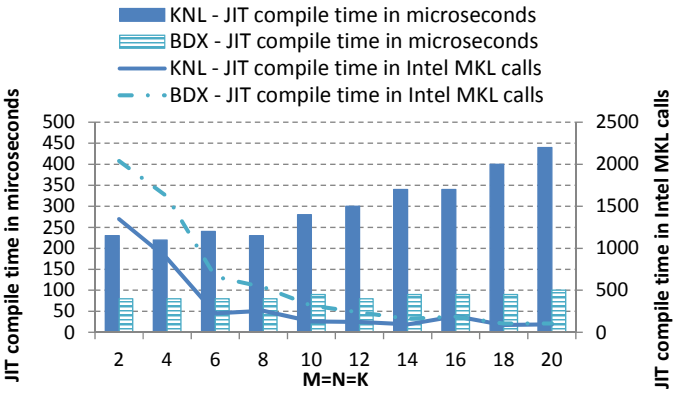

Fig. 5. JIT compile overhead of LIBXSMM in microseconds and in Intel MKL DGEMM calls on BDX and KNL.

A. Single-Core Efficiency Benchmarks

We start our performance evaluation by focusing on single core performance out of L1 cache for small GEMMs. We compare LIBXSMM against Intel MKL using direct call mode and two C++ expression template based libraries, Eigen [19] and BLAZE [20]. Both offer intrinsic based implementations of GEMM and also feature compile-time static support for GEMM sizes aiming at high performance. Fig. 6 summarizes the performance measurements for BDX in the upper plot and for KNL in the lower one. LIBXSMM achieves the best performance across all tests with only one exception. It is the clear winner when the static code generation feature of Eigen and BLAZE is not used. In this case even Intel MKL is able to outperform Eigen and BLAZE. Eigen is clearly the slowest library in our comparison. We also want to highlight the fact that Eigen and BLAZE suffer from performance differences which are caused by using different compilers. As LIBXSMM generates directly assembly, it doesn't show these effects and is independent of the compilation type and version. Furthermore, Eigen and BLAZE do not offer a (high-performing) BLAS interface and are hard to incorporate into large scientific packages written in FORTRAN. We therefore limit ourselves to Intel MKL as comparison point for the rest of the paper. In summary, we can conclude these artificial tests by seeing a 4–25 × speed-up on KNL and 0.9–12 × speed-up on BDX from using LIBXSMM.

B. CP2K

CP2K is a powerful and scalable program for atomistic simulations of a wide range of systems, including condensed phase, molecular systems and complex interfaces. CP2K features a wide range of atomistic interaction models including classical potentials, semi-empirical schemes, Density Functional Theory (DFT), Hartree-Fock (HF) and post-HF correlation methods such as MP2 and RPA. With CP2K, computing the electronic structure of up to 1 million atoms has been demonstrated [21]. It was a 2015 Gordon Bell Finalist [22].

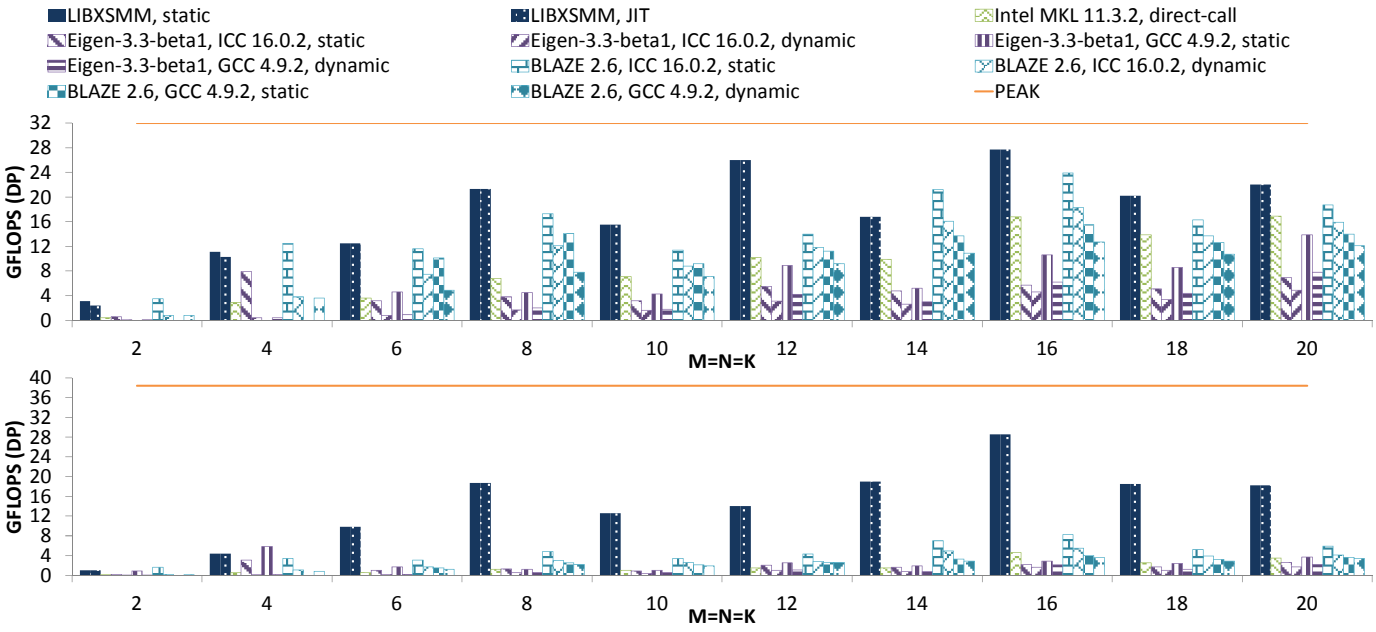

Fig. 6. Performance of LIBXSMM for static and JIT compilation for square matrices of order 2 until 20 on a single core of the Intel Xeon E5-2697v4 processor clocked at 2.0 GHz, its AVX-base frequency (top) and a single core of the Intel Xeon Phi 7250 processor clocked at 1.2 GHz, its AVX-base frequency (bottom). LIBXSMM's performance is compared against various other libraries: Intel MKL 11.3.2, Eigen-3.3-beta1 and BLAZE 2.6. We want to note that a source scan of Eigen and BLAZE creates the impression that there are no special optimizations for AVX-512F instructions set extensions.

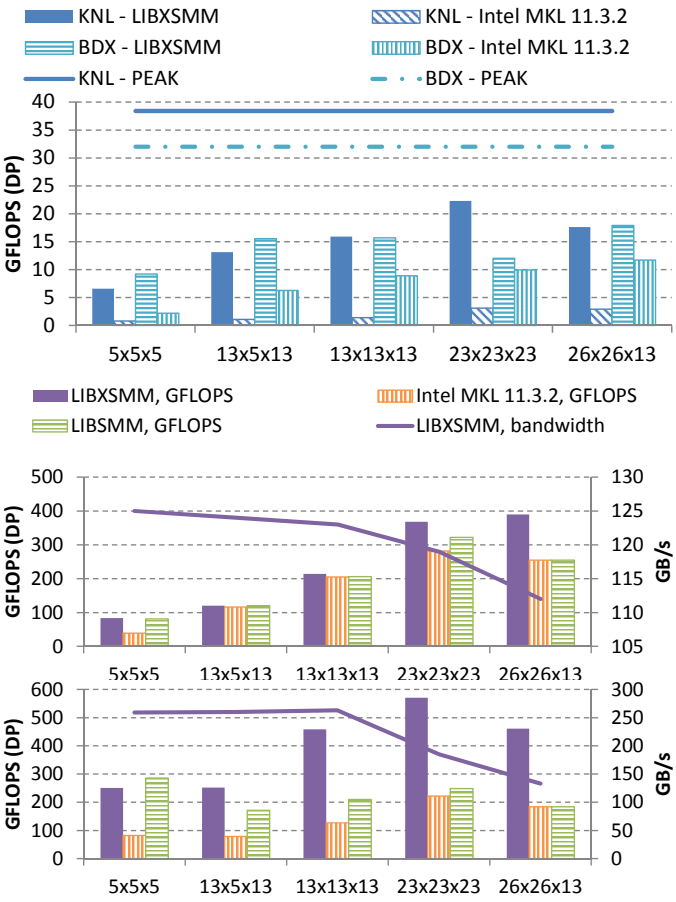

Fig. 7. CP2K kernel performance on BDX and KNL (top plot) and DBCSR reproducer performance for BDX (middle plot) and KNL (bottom plot).

The CP2K linear scaling DFT ([23]) implementation exploits the fact that operators in a localized atomic basis are sparse. CP2K's sparse matrix library DBCSR (Distributed Block Compressed Sparse Row) [10] is designed to take full advantage of the block-sparse nature of the matrices and is well parallelized with inter-process communication based on Cannon's algorithm [24]. At the node level, the computation consists of two main steps: (1) book-keeping work to decide which matrix blocks need to be computed, and (2) processing batches of block-wise small matrix multiplications. Important block sizes typically fall within the range of 4–32 e. g., 5 (Hydrogen), 13 (Oxygen, Carbon), or 23 (Water) are popular examples of ending up with uneven SIMD-sizes. However, efficiently batching small matrix multiplications is key to achieve high performance in large scale DFT calculations.

The upper plot of Fig. 7 depicts a narrow selection of important DBCSR kernels. As the matrix shapes are based on the physical setting, zero-padding is not an option and we notice the aforementioned performance degradation due to unaligned loads. However, LIBXSMM is still able to outperform Intel MKL, often even by 1.2–5 ×.

The lower two plots compare a reproducer of DBCSR's stack processing for the same kernel sizes, the upper plot shows dual-socket BDX, the lower KNL. Additionally, we also compare the performance to CP2K's own LIBSMM library which performs an extensive auto-tuning stage to generate code variants for small matrix multiplications and alternative BLAS implementations to achieve significant speedups over vendor-tuned math libraries (10×). On both platforms, LIBXSMM matches (extremely memory bandwidth bound

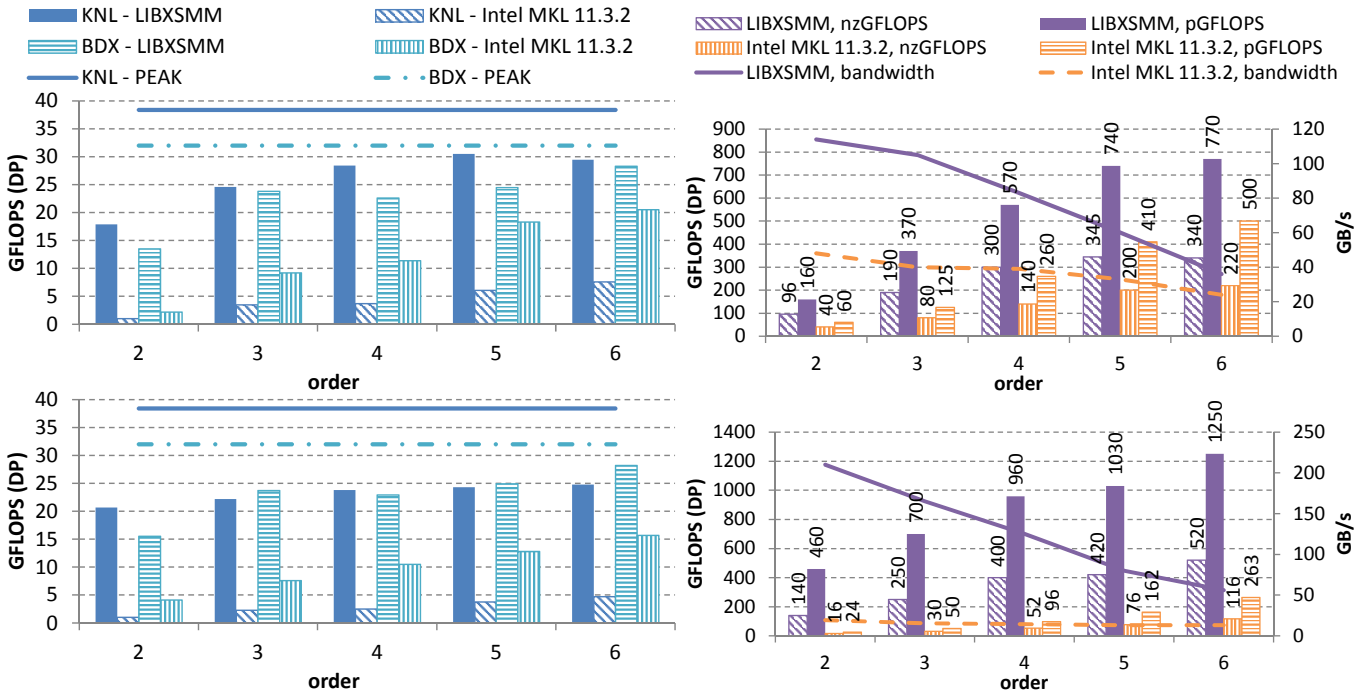

Fig. 8. SeisSol kernel performance on BDX and KNL (left plots, upper plot: $\{4, 12, 20, 36, 56\} \times 9 \times \{4, 12, 20, 36, 56\}$, lower plot: $\{4, 12, 20, 36, 56\} \times 9 \times 9$) and SeisSol reproducer performance execution the LOH.1 scenario with 380K elements (right plots, upper plot: dual-socket BDX, lower plot: KNL).

executions) or exceeds (slightly bigger matrices) LIBSMM's or Intel MKL's performance. Due to LIBXSMM, KNL can achieve a solid speed-up over dual-socket BDX.

C. 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. SeisSol 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. Depending on the order of convergence the DOF matrices are within $\{4, 10, 20, 35, 56\} \times 9$ and stiffness and flux matrices are of order $\{4, 10, 20, 35, 56\}$. SeisSol employs application-based zero padding to assemble dense matrices which allow for aligned memory accesses. Therefore a typical element-local operation involves GEMMs of size $\{4, 12, 20, 36, 56\} \times 9 \times \{4, 12, 20, 36, 56\}$ (Fig. 8, left, upper) and $\{4, 12, 20, 36, 56\} \times 9 \times 9$ (Fig. 8, left, bottom) on BDX. On KNL these numbers (except 9) need to be rounded-up to multiples of 8. Even at petascale, SeisSol spends more than 98% of its time in these routines.

The plots in the left column of Fig. 8 depict the single core performance of running SeisSol zero-padded matrix shapes on BDX and KNL. LIBXSMM allows us to run these kernels with identical GFLOPS rates on KNL as on BDX although KNL features a lower frequency. This shows the high code quality of LIBXSMM once again, as the wider vector instruction set can be fully leveraged. For the larger cases (upper-left plot) Intel MKL is able to catch up but not close the gap. As higher order runs also require the lower order kernels in the time integration phase of SeisSol, an Intel MKL enabled SeisSol is still expected to run significantly slower than when using LIBXSMM. On KNL LIBXSMM is in general 3–5 $\times$ faster than Intel MKL and on BDX these numbers change to 1.2–4 $\times$.

The right column of Fig. 8 covers numbers measured by SeisSol performance proxy which models SeisSol single node performance at 98% accuracy. We also provide padded (p) and non-zero (nz) GFLOPS rated. The upper plot corresponds to an execution on BDX whereas the lower one is gathered on KNL. Especially for higher order (here SeisSol is compute bound) they reproduce the kernel performance of the left plots. As the lower order runs are memory bandwidth bound, the gap between LIBXSMM and Intel MKL becomes smaller. From a platform comparison perspective, KNL achieves more than 2 $\times$ the performance of dual-socket BDX for low order runs and is roughly 1.6 $\times$ faster for order 6 executions.

D. NekBox

NekBox is a stripped-down version of the Nek5000 (a high order CFD) code. As a descendant of Nek5000, it implements SEM [25] with tunable order. NekBox takes advantage of static, uniform meshes to solve the coarse part of the preconditioner with FFTs or DCTs. This improves its overall efficiency and scalability. SEM is implemented via a two-level discretization constructed from tensor products of Gauss-Lobatto-Legendre (GLL) quadrature points within elements and continuity across elements, forming a mesh. Fields are represented

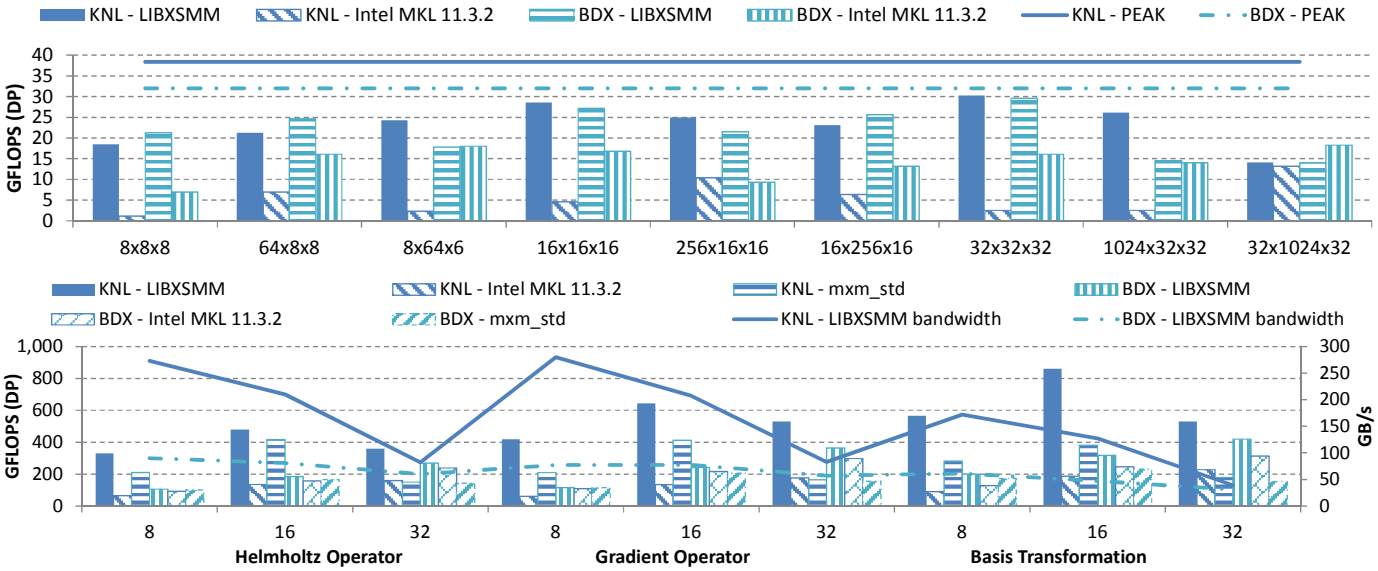

Fig. 9. NekBox kernel performance on BDX and KNL (upper plot) NekBox reproducer performance (lower plot) for Helmholtz operator, tensor product gradient and basis transformation of different polynomial order.

as $u(x,y,z) = \sum_{i=0}^{p}\sum_{j=0}^{p}\sum_{k=0}^{p}\tilde{u}_{i,j,k,e}h_i(x)h_j(y)h_k(z)$, where p is the polynomial order of the method, $e(x,y,z)$ is the index of the element in the mesh, and $h_i(x)$ is the ith Lagrange polynomial through the GLL points of element e. In SEM, operators are formulated as the product of a local operator and direct stiffness summation, which enforces continuity at the elements' boundaries. The arithmetic intensity of an operator evaluation in SEM is $O(p)$ as linear operators from $R^{N\times N\times N} \rightarrow R^{N\times N\times N}$ can be evaluated in $O(N^4)$ operations instead of $O(N^6)$ [26]. Implementing the tensor product operations requires several small GEMM operations: one of each shape $p^2 \times p \times p$, $p \times p^2 \times p$ and p of shape $p \times p \times p$. A typical NekBox execution spends around 40-50% of its time in these tensor product routines. This number can vary with the scientific problem being solved.

Fig. 9 depicts our performance tests performed with respect to NekBox. The performance of running single core and out of hot L1 and L2 caches is given in the upper plot of Fig. 9 for element sizes 8, 16, 32 which span the region of orders used in production runs. In general, these results confirm our synthetic single core results from Fig. 6: on KNL LIBXSMM is outperforming Intel MKL by a large fraction except the case with large N, $32 \times 1024 \times 32$ where blocking techniques of large GEMMs are beneficial. In all other cases, LIBXSMM is between 3 to 10 times faster than Intel MKL. On BDX the difference between LIBXSMM and Intel MKL is much smaller including parity of both libraries. The speed-up of LIBXSMM ranges between 0.85 (for $32 \times 1024 \times 32$) to 3 times.

The bottom part of Fig. 9 shows the performance of NekBox's key kernels (Helmholtz operator, gradient operator and basis transformation) for three different polynomial orders and using all cores available on the systems. It also adds NekBox's own small GEMM implementation *mxm* to the mix which is implemented using compiler-tuned FORTRAN primitives.

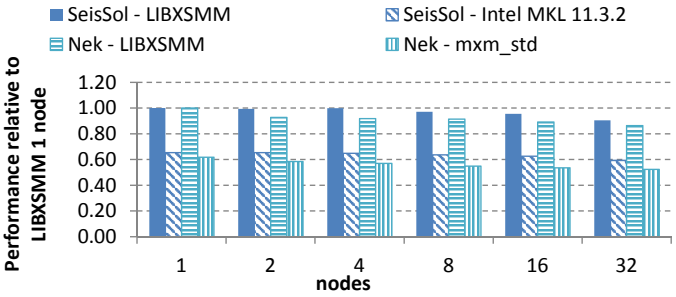

Fig. 10. Parallel efficiency with respect to a LIBXSMM normalized strong-scaling of SeisSol and weak-scaling of NekBox on up to 32 BDX nodes (1156 total cores).

As these operators are bandwidth limited, especially for the lower order cases, the differences between LIBXSMM, mxm and Intel MKL are rather low on BDX but high on KNL. This is because KNL offers up to 360 GB/s read bandwidth which can be reached when using a highly efficient kernel to create enough memory pressure. Even on BDX LIBXSMM can outperform Intel MKL by 10-20% for higher orders where it materializes gains of over two times over mxm. When comparing BDX to KNL we can see that for the most often used element size 8, KNL is able to speed-up NekBox's kernels by close to three times comparing to the dual-socket BDX platform.

E. Cluster-Level Results

We have demonstrated during the last three paragraphs that LIBXSMM can significantly speed-up the execution of several important scientific codes. We therefore want to close our application-based performance evaluation by full-application scaling experiments on up to 32 nodes of BDX (1156 total cores) connected by a 100GiB Intel Omni-Path network for

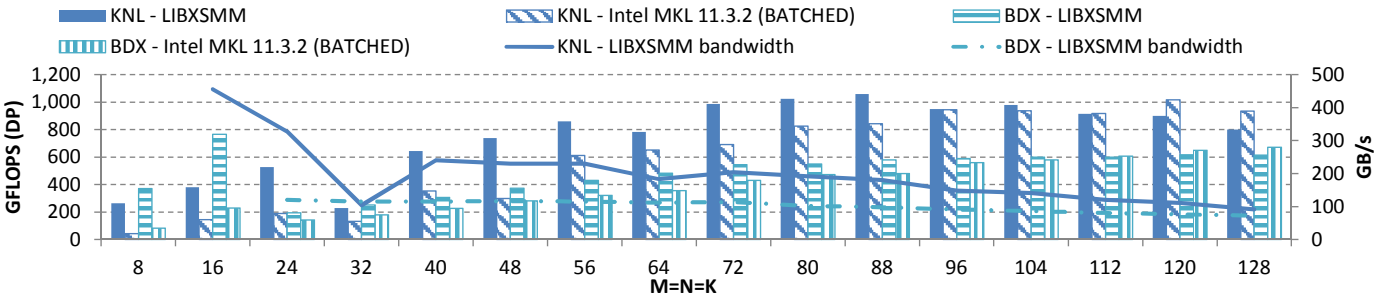

Fig. 11. LIBXSMM vs. Intel MKL DGEMM_BATCHED for a batch size of 11,200 small and independent DGEMMs.

SeisSol and NekBox. Fig. 10 depicts a SeisSol strong-scaling of the LOH.1 benchmark with 380K elements at order 6 and a NekBox weak-scaling of the Rayleigh-Taylor instability with eight elements per process at element size 32. We see that we can nicely reproduce the aforementioned performance gains at application level and at scale. Both SeisSol and NekBox see a substantially boost (roughly 50%) from using LIBXSMM with respect to time-to-solution and maintain their very good scaling.

IV. Batched Matrix Multiplication

The library supports a novel kind of function signature which allows to multiply batches of matrices. The signature of the kernel function $f_S(a, b, c)$ is extended by three arguments a', b', and c' (f_S becomes $f_S(a, b, c, a', b', c')$) while the specialization S stays the same i. e., $S := m, n, k, \ldots$. The additional arguments represent the locations of the next operands to be prefetched. An ongoing multiplication becomes the driver for issuing software prefetches according to a prefetch strategy. The "prefetch signature" can be requested at build time of the library in order as a default (which also applies to statically generated kernels). JIT code generation allows a kernel with prefetch signature to be to dispatched independent of the (static) build-time preference.

LIBXSMM's "batch interface" is different to CUDA and MKL, where multiple kernel calls are enqueued into several execution streams (perhaps according to the number of elements in the batch) while still referring to the same kernel [6]. In contrast to CUDA, LIBXSMM still separates the code dispatch ("launch overhead") from chaining calls into a batch while the parallelization technique is also not internalized but user-controlled. For example, [5], [7], [8] are two libraries providing a batch interface which not only requires an explicit data structure (as opposed to an implicit prefetch signature), but are also built on top of a parallel programming model. So for LIBXSMM batching is calling a series of single GEMMs.

Fig. 11 compares LIBXSMM to Intel MKL's batched solution for a sweep of small sizes on dual-socket BDX and KNL. For cases with matrix order smaller than 96 the simple combination of LIBXSMM with auto-prefetching and OpenMP is able to gain performance improvements of 20-100% over Intel MKL depending on architecture and size. These results nicely demonstrate that even a batch GEMM implementation which can rely on much matrix level parallelism is sensitive to small GEMM performance.

V. Conclusions

We ample demonstrated the importance of fast small GEMM routines for various scientific applications from different application fields. Due to being compatible with regular BLAS libraries as well as due to the JIT compilation phase, LIBXSMM can be regarded as agnostic with respect to Intel's vector instructions sets. Using LIBXSMM improves the ease of use for CP2K and Nek significantly, and reduces the development effort for large scientific codes. Although CP2K and Nek offer their custom solutions (required to be tuned for every new architecture), LIBXSMM is able to match their performance in memory bandwidth bound scenarios, and even significantly outperforms these highly specialized kernels towards compute-bound cases. For the latter, LIBXSMM's hardware-aware and precise code generation becomes most effective and the library is able to outperform current state-of-the-art vendor libraries by more than $10\times$. Additionally, we outlined the importance of an efficient small GEMM when implementing a batched GEMM interface. For batches with matrix dimensions below $N = 100$, an OpenMP parallelized for-loop plus LIBXSMM clearly outperforms Intel MKL by 20-50% depending on the batch size, matrix shape, and the processor used.

There are two future research possibilities which we are currently pursuing: (1) extending the technology to other important LAPACK functions such as LU, Cholesky or SVD and EVD (HPC-centric), and (2) using LIBXSMM in machine learning, and deep learning applications in particular. Matrices needed in this field are normally odd-shaped with classic GEMM implementations not delivering the best performance. As demonstrated for CP2K, LIBXSMM can master challenging combinations of dimensions. Further using JIT compilation, different blocking strategies for medium-sized GEMMs can be selected at runtime to optimally adjust for the requested shape.

References

[1] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the atlas project," *PARALLEL COMPUTING*, vol. 27, p. 2001, 2000.

[2] K. Goto and R. A. v. d. Geijn, "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

[3] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 25:1–25:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503219

[4] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 14:1–14:33, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454

[5] NVIDIA Corporation, "CUDA Toolkit Documentation," this function performs the matrix-matrix multiplications of an array of matrices. [Online]. Available: http://docs.nvidia.com/cuda/cublas/#cublas-lt-t-gt-gemmbatched

[6] ——, "CUDA Toolkit Documentation," for instance, [...][batching kernels] to make many small independent matrix-matrix multiplications with dense matrices. [Online]. Available: http://docs.nvidia.com/cuda/cublas/#batching-kernels

[7] Intel Corporation, "Intel MKL 11.3 Release Notes," 2015, introduced ?GEMM_BATCH and (C/Z)GEMM3M_BATCH functions to perform multiple independent matrix-matrix multiply operations. [Online]. Available: https://software.intel.com/en-us/articles/intel-mkl-113-release-notes

[8] ——, "Intel MKL Documentation," this function performs the matrix-matrix multiplications of an array of matrices. [Online]. Available: https://software.intel.com/en-us/articles/introducing-batch-gemm-operations

[9] B. Bani-Ismail and G. Kanaan, "Comparing different sparse matrix storage structures as index structure for arabic text collection," *Int. J. Inf. Retr. Res.*, vol. 2, no. 2, pp. 52–67, Apr. 2012. [Online]. Available: http://dx.doi.org/10.4018/ijirr.2012040105

[10] U. Borstnik, J. VandeVondele, V. Weber, and J. Hutter, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Computing*, vol. 40, pp. 47–58, 2014.

[11] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. C. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *CoRR*, vol. abs/1512.02595, 2015. [Online]. Available: http://arxiv.org/abs/1512.02595

[12] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015. [Online]. Available: http://arxiv.org/abs/1509.09308

[13] ——, "maxdnn: An efficient convolution kernel for deep learning with maxwell gpus," *CoRR*, vol. abs/1501.06633, 2015. [Online]. Available: http://arxiv.org/abs/1501.06633

[14] A. Sodani *et al.*, "Knights Landing (KNL): 2nd Generation Intel(R) Xeon Phi(TM) Processor," *IEEE Micro, Hot Chips Special Issue, to appear*, March 2016.

[15] J. Hutter, M. Iannuzzi, F. Schiffmann, and J. VandeVondele, "CP2K: Atomistic Simulations of Condensed Matter Systems," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 4, no. 1, pp. 15–25, 2014.

[16] The CP2K Developers Group, "CP2K: Open Source Molecular Dynamics." [Online]. Available: http://www.cp2k.org

[17] The SeisSol Developers Group, "SeisSol is a software package for simulating wave propagation and dynamic rupture based on the arbitrary high-order accurate derivative discontinuous Galerkin method (ADER-DG)." [Online]. Available: http://www.seissol.org/

[18] P. Fischer, J. Lottes, S. Kerkemeier, O. Marin, K. Heisey, E. Obabko, A. annd Merzari, and Y. Peet, *Nek5000 User Documentation*. Argonne National Laboratory, 2016. [Online]. Available: https://nek5000.mcs.anl.gov/documentation/

[19] The Eigen Developers Group, "Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms." [Online]. Available: http://eigen.tuxfamily.org

[20] K. Iglberger, G. Hager, J. Treibig, and U. Rde, "High performance smart expression template math libraries," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, July 2012, pp. 367–373.

[21] J. VandeVondele, U. Borstnik, and J. Hutter, "Linear scaling self-consistent field calculations for millions of atoms in the condensed phase," *The Journal of Chemical Theory and Computation*, vol. 8, no. 10, pp. 3565–3573, 2012.

[22] M. Calderara, S. Brück, A. Pedersen, M. H. Bani-Hashemian, J. VandeVondele, and M. Luisier, "Pushing back the limit of ab-initio quantum transport simulations on hybrid supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 3:1–3:12.

[23] W. Kohn and L. J. Sham, "Self–Consistent Equations Including Exchange and Correlation Effects," *Physical Review*, vol. 140, pp. A1133–A1138, 1965.

[24] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University, 1969.

[25] A. T. Patera, "A spectral element method for fluid dynamics: laminar flow in a channel expansion," *Journal of computational Physics*, vol. 54, no. 3, pp. 468–488, 1984.

[26] H. M. Tufo *et al.*, "Terascale spectral element algorithms and implementations," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999, p. 68.