LIBXSMM

Accelerating Small Matrix Multiplications by Runtime Code Generation

Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst

Presented by Isuru Fernando.

November 2, 2018

Motivation

Small matrix-multiplication is used in,

- Discontinuous Galerkin methods
- Spectral element methods
- Information Retrieval (Blocked Compressed Sparse Row matrices)

What's Small?

Motivation

Small matrix-multiplication is used in,

- Discontinuous Galerkin methods
- Spectral element methods
- Information Retrieval (Blocked Compressed Sparse Row matrices)

What's Small?

For $A \in \mathbb{R}^{M \times K}$ and $B \in \mathbb{R}^{K \times N}$, let $C = AB$

$$MNK \leq 80^3$$

Why specialized library?

- General purpose code is not optimal for all scenarios.
- Lack of specialization is not good for small matrices.
- Building specialized code at compile time $\implies$ library is too large.

Overview

For $AVX2$, micro-kernels of size $\{16, 12, 8, 4, 2, 1\} \times \{1, 2, 3\}$ for C

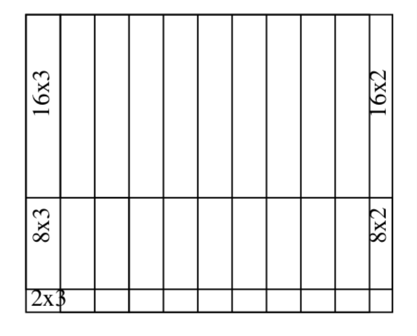

Figure 1: Partitioning of C matrix of size, 26×32

Overview

Micro-kernel of size 16×3 uses,

- 12 AVX2 registers to store the result C
- 3 AVX2 registers for storing the l^{th} row of B broadcasted. (eg: b_{21}, b_{22}, b_{23})
- 1 AVX2 register containing 4 entries of column l of A. (Loads $[a_{12}, a_{22}, a_{32}, a_{42}]$ and then $[a_{52}, a_{62}, a_{72}, a_{82}]$)

$$\begin{bmatrix} a_{12} & & \\ a_{22} & & \\ a_{32} & & \\ a_{42} & & \\ \vdots & & \\ a_{16_3} & & \end{bmatrix} \begin{bmatrix} b_{21} & b_{22} & b_{23} \\ & & \\ & \vdots & \\ & & \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \\ c_{41} & c_{42} & c_{43} \\ \vdots & \vdots & \vdots \\ c_{16,1} & c_{16,2} & c_{16,3} \end{bmatrix}$$

Overview

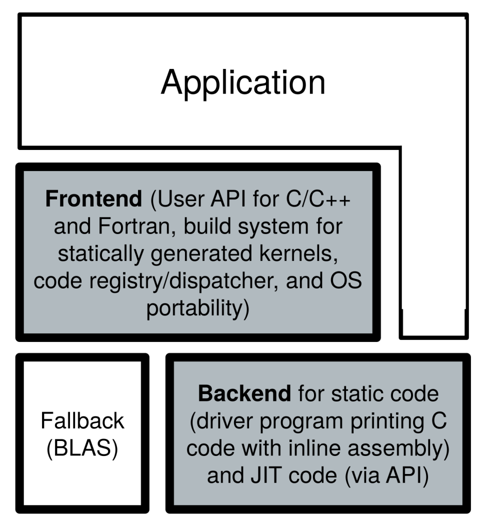

Figure 2: LIBXSMM application overview

Code generation

Generate C code with inline assembly at library build time.

$$C = \alpha AB + \beta C$$

Configurable parameters

- set of M, N, K tuples
- Architecture (noarch, wsm, snb, hsw, knc, knl, knm, skx)
- single precision or double precision or both
- prefetch strategy
- LDA, LDB, LDC (leading dimensions of A, B, C)

Limitations

- $\alpha = 1$
- $\beta = 0, 1$
- Column major only
- No dynamic architecture selection if statically compiled.

JIT compilation

Evaluation criteria for a JIT

- Fast
- Supports AVX512
- Actively maintained
- Open Source

Authors looked at,

- LLVM - Full blown (with IR, phases, etc.), "slow" JIT, complex ([2])
- Xbyak - No AVX512 support (in 2015)
- XED - closed source (in 2015)

JIT compilation

- Generate code if no static kernel exists
- Generate machine code in memory.
  - No bulky compiler is used. Internal implementation
  - vmovaps 256(%rax,%rcx,2), %ymm16 $\Longrightarrow$ 0x62,0xE1,0x7C,0x28,0x28,0x44,0x48,0x08
  - Cast executable buffer to a function pointer
  - Faster than compiler backends like LLVM
  - AVX2 Kernel source Codegen source
- Keep a thread-local cache of already built kernels
  - Use CRC32 hash of $M, N, K, LDA, LDB, LDC, transA, transB$ and prefetch strategy.
  - Check the last hit first.

Prefetching

- "nopf": no prefetching at all, just 3 inputs (A, B, C)
- "pfsigonly": just prefetching signature, 6 inputs (A, B, C, A', B', C')
- "BL2viaC": uses accesses to C to prefetch B'
- "curAL2": prefetches current A ahead in the kernel
- "curAL2-BL2viaC": combines curAL2 and BL2viaC
- "AL2": uses accesses to A to prefetch A'
- "AL2-BL2viaC": combines AL2 and BL2viaC
- "AL2jpst": aggressive A' prefetch of first rows without any structure
- "AL2jpst-BL2viaC": combines AL2jpst and BL2viaC
- "AL1": prefetch A' into L1 via accesses to A
- "AL1-BL1": prefetch A' and B' into L1
- "AL1-BL1-CL1": prefetch A', B', and C' into L1

Results

- **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 MC-DRAM@7.2 GT, 96 GB DDR4-2400, FLAT/QUADRANT memory mode.

Results

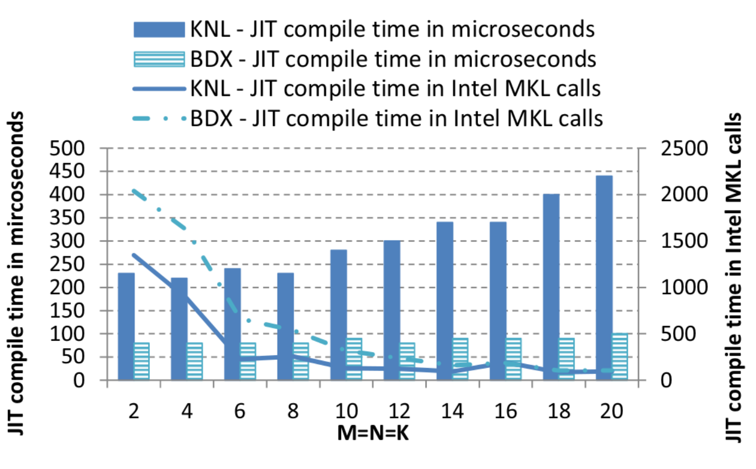

Figure 3: JIT compile overhead of LIBXSMM in microseconds and in Intel MKL DGEMM calls on BDX and KNL. Source: [1]

Results

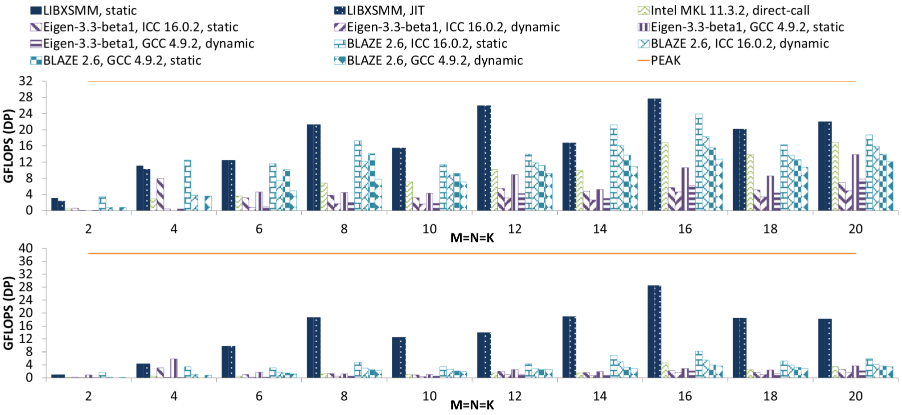

Figure 4: *Performance of LIBXSMM for static and JIT compilation for square matrices of order 2 until 20 on a single core of the Intel Xeon E5-2697v4 processor clocked at 2.0 GHz, its AVX-base frequency (top) and a single core of the Intel Xeon Phi 7250 processor clocked at 1.2 GHz, its AVX-base frequency (bottom). LIBXSMMs 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. Source: [1]*

Results

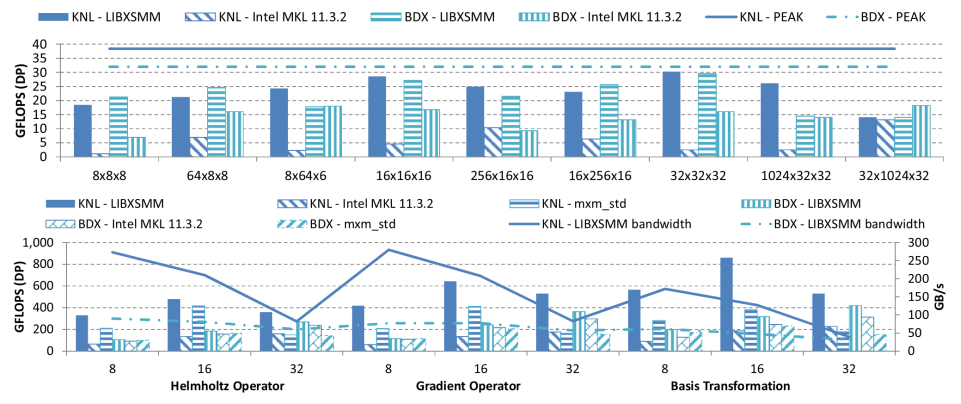

Figure 5: NekBox kernel performance on BDX and KNL (upper plot) NekBox reproducer performance (lower plot) for Helmholtz operator, tensor product gradient and basis transformation of different polynomial order. Source: [1]

Future work

- Dynamic dispatch of statically generated kernels
- Row major
- Mixed types
- Complex, half and other precision formats

Summary

- Specialized kernels give good performance for small matrix multiplications.
- Maximize the use of AVX2/AVX512 registers.
- Using a JIT compilation approach to avoid building large number of configurations.
- Use a cache to amortize the cost of compilation.

References

Alexander Heinecke et al. "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Salt Lake City, Utah: IEEE Press, 2016, 84:1–84:11. ISBN: 978-1-4673-8815-3. URL: http://dl.acm.org/citation.cfm?id=3014904.3015017.

Hans Pabst. *Intel and ML*. https://indico.cern.ch/event/595059/contributions/2499304/attachments/1430242/2196659/Intel_and_ML_Talk_HansPabst.pdf. [Online; accessed 30-Oct-2018]. 2017.