XBLAS AND MORE

XBLAS: Jim Demmel, *Greg Henry*, Jason Riedy, Peter Tang, Xiaoye S. Li

MORE: Alexander Heinecke, *Greg Henry*

Agenda

- Extended Precision BLAS (XBLAS)
- Batched BLAS Experiments

Current XBLAS (1.0.248) Definition

- Extension to the BLAS with extra precision and mixed precision
- Covers 17 common BLAS functionality with lots of parameter variants
- Primarily relies on double-double computations
- Implemented via M4 code generation
- Comes with a performance and testing suite. Used in LAPACK.
- 2008 XBLAS Developers: Xiaoye Li, Jim Demmel, David Bailey, Yozo Hida, Jimmy Iskandar, Anil Kapur, Michael Martin, Brandon Thompson, Teresa Tung, Daniel Yoo
- “Design, Implementation, and Testing of the Extended and Mixed Precision BLAS” from 2000: Li, Demmel, Bailey, Henry, Hida, Iskandar, Kahan, Kapur, Martin, Tung, Yoo
- http://www.netlib.org/xblas/ , 3 clause BSD, Last Release Nov. 2008
- Level 1 : 4 cases : Dot (inner product), Sum, AXPBY, WAXPBY
- Level 2 : 10 cases: GEMV, GBMV, SYMV, SBMV, SPMV, HEMV, HBMV, HPMV, GE_SUM_MV, TRSV
- Level 3 : 3 cases : GEMM, SYMM, HEMM

What is the current XBLAS (1.0.248)

Extended precision

- Can be used internally; input/output remain the same
- Example: DOT (…… , PREC)
  - PREC is a runtime variable = Single, Double, or Extra (we are suggest removing this)

Mixed precision

- Can mix real and complex, single and double for input/output
- This can be relevant for those interested in Half Precision and Single mixes
- We are trying to simplify the XBLAS as much as possible

Even with only 17 base functionalities, there are more final routines than in BLAS!

- Example: DOT now has 32 = 4 (old) + 28 (new)

XBLAS Goal: Enhance existing precisions for LAPACK-level libraries

- Extended precision (double-double or quad or otherwise) exists in LAPACK
- But performance is lacking
- Usage outside LAPACK might be buggy or contain dead code
- Too many cases/routines and ironically not the right cases
- Our goal: Enhance this existing work.
- To do this: we need fast vectorized routines with the appropriate API

Today’s look at extended precision…

- What is extra precision and how does it work
- A bad rep for extra precision
- Applications that require extended precision
- The XBLAS in Intel® MKL and LAPACK
- New Proposal/standards going forward
- Conclusion

What do we mean by extra precision?

- Quadruple precision is one example (note: lacking HW support in IA)
  - Can mean a binary128 standard (113 bits significand bits)
- Double-double (or more) might be faster in software
- Double precision (binary64) commonly follows IEEE 754 standard
  - 64 bits typically are: 1 sign bit, 11 exponent bits, 53 significand bits
- Suppose we use 2 real*8 doubles to store our numbers (DD from here out)
  - We can have over 106+ bits dedicated to each calculation (still keeping the 11 bits for the exponent)
  - Can try triple-double (DDD), quad-double (DDDD), or even double-quad (QQ)

XBLAS DOT Error Bound

|r_comp - r_acc | <= (n+2) * (🎁_int + 🎁_acc) * S + U + 🎁_out * |r_acc|

r_comp = r computed by the routine being tested

r_acc = r computed by most accurate routine

🎁_int = claimed internal precision

🎁_acc = our most accurate precision (106 bits)

🎁_out = output precision

u_int = underflow threshold in claimed internal precision

u_acc = underflow threshold in most accurate precision

u_out = underflow threshold in output precision

S = |alpha| * $Y_{i=1,n}$ |x_i * y_i| + |beta * r|

U = (|alpha| * n + 2) * (u_int + u_acc) + u_out

DDADD (Bailey) : DDC(*) = DDA(*) + DDB(*)

```
subroutine ddadd (dda, ddb, ddc)

real*8 dda(2), ddb(2), ddc(2), e, t1, t2

t1 = dda(1) + ddb(1)

e = t1 - dda(1)

t2 = ((ddb(1) - e) + (dda(1) - (t1 - e))) + dda(2) + ddb(2)

ddc(1) = t1 + t2

ddc(2) = t2 - (ddc(1) - t1)

end

// See also the QD library
```

DDMUL (Bailey) : DDC(*) = DDA(*) * DDB(*), FMA ver.

```
subroutine ddmul (dda, ddb, ddc)
real*8 dda(2), ddb(2), ddc(2), c11, c21, c2, e, t1, t2
c11 = dda(1) * ddb(1)
c21 = dda(1) * ddb(1) - c11
c2 = dda(1) * ddb(2) + dda(2) * ddb(1)
t1 = c11 + c2
e = t1 - c11
t2 = ((c2 - e) + (c11 - (t1 - e))) + c21 + dda(2) * ddb(2)
ddc(1) = t1 + t2
ddc(2) = t2 - (ddc(1) - t1)
end
```

Latest Efforts

- We consider other algorithms besides last two slides
- Note that FMA vs. non-FMA is an important consideration
    - We have considered AVX separately from AVX2, AVX-512
- Ogita, Rump, Oishi have a SIAM SISC 2005 paper: “Accurate Sum and Dot Product”
- We have implemented algorithms like ORO’s
- We require vectorized, tuned solutions
    - We have created optimized AVX/AVX2 kernels discussed here

Extra Precision: The Bad Reputation

- Extra Precision is not frequently used in linear algebra
- Software Quad is way too slow: 50x-100x slower than double-precision
- Using optimizations with the Intel Compiler on double-double can lead to incorrect results without the perfect compiler flags
- Most SW implementations of DD are very slow
- One mistake can lead to only double precision accuracy

Extra Precision Level-3 BLAS can suffer where regular BLAS won’t

- BLAS is broken up by Level:
  - Level-1 : O(n) computation, O(n) data (dot products, scalar*vector)
  - Level-2 : O(n^2) computation, O(n^2) data (matrix vector product)
  - Level-3 : O(n^3) computation, O(n^2) data (matrix-matrix product)
- Level-1 and Level-2 BLAS **can** be memory-bound
- Level-3 BLAS **can** be computation-bound (which is good)
- x86 algorithms **can** be register-starved
- DD computations **can** be FP latency-bound (which is bad)

Use Case 1: Iterative Refinement

- Solve Ax_0 = b in real*8 (perhaps via LU, O(n^3) work)
- "Error Bounds from Extra Precise Iterative Refinement" (LAWN165)
  - J. Demmel, Y. Hida, W. Kahan, X.S. Li, S. Mukherjee, E. J. Riedy
- "Extra-precise Iterative Refinement for Overdetermined Least Squares Problems" (LAWN 188)
  - J. Demmel, Y. Hida, X.S. Li, E.J. Riedy
- Form the residual error $r_0 = b - Ax_0$ in extended precision, O(n^2) work
- Solve $Ay_1 = r_0$ in real*8, O(n^2) work if we kept the LU factors
- Set $x_1 = x_0 + y_1$ (O(n) work), and iterate until the residuals are "small enough"
- Note that one can combine a few steps with extra precision as well
- If the condition of A is below 1/eps, this results in high accuracy
- The majority of the work (O(n^3)) is in double. Only some is in extended precision (O(n^2))
  - However, SW Quad is so slow that the O(n^2) work can dominate
  - DD libraries tend to be written in high-level code and aren't vectorized

Use Case 2: Least Squares, min $||Ax-b||_2$

- One can either do “iterative refinement” on least squares… Or…
- Consider the Cholesky QR algorithm:
  - Mixed-Precision Cholesky QR Factorization and Its Case Studies on Multicore CPU with Multiple GPUs
    - I. Yamazaki, S. Tomov, J. Dongarra
  - Form $A^TA = B$ (note squaring the condition number!)
  - Do a Cholesky decomposition of $B = R^*R'$ (R upper triangular)
  - (Optional if Q is desired) Do a triangular solve $Q = A^*R^{-1}$ for Q
  - Now we have a Tall-Skinny (TS) QR algorithm (less stable, what if a column of A is zero)
  - Use QR to finish solving the least squares problem
- Might need extra precision to complete Cholesky
- This also gives us a TSQR approach
- (One can also draw similar results for SVD)

Other use cases?

- Previous Use Cases (Using numbers from the XBLAS ref., omitting the previous 2 cases)

  2. Avoiding Pivoting in Sparse GE

  3. Accelerating Iterative Methods for Ax=b like GMRES

  4. Support of Applications needing mix of real and complex precisions

  6. Solving Ill-conditioned triangular systems

  7. Eigenvalues and Eigenvectors of SEP

  8. Cheap Error Bounds

- Climate MOM ocean models…
- Jon Wilkening from UCB uses DD/QD/MPFR/etc..
- Unsymmetric Inverse Iteration

Combinatorial explosions behind mixed precision

- BLAS come in 4 precisions (c,s,d,z) and have up to 3 inputs (A,B,C)
  - Doesn't count scalars like alpha, beta
- Adding DD and ZZ to this list makes it 6 precisions.
- If you consider each input could be in any of the precisions, and internally…
  - One routine could have 6*6*6*6=1296 possibilities instead of 4!
  - There are ~40 BLAS routines, so we're looking at 10^5 cases
  - Granted, many combinations are obviously useless…
- Do we need all the mixed cases code explosion (see next two slides)?
- We'd like to do without mixed precision for simplicity

12 “Limited” XBLAS GEMM Cases : alpha*A*B + beta*C

Alpha/Beta = highest precision	A	B	C
D	S	S	D
D	S	D	D
D	D	S	D
Z	C	C	Z
Z	C	Z	Z
Z	Z	C	Z
C	S	S	C
C	S	C	C
C	C	S	C
Z	D	D	Z
Z	D	Z	Z
Z	Z	D	Z

3 cases of DGEMM

First 3 cases of ZGEMM

3 cases of CGEMM

Second 3 cases of ZGEMM

The XBLAS has 980 routines… where from?

- There are 56 cases of XBLAS GEMM
- Half of them are F2C wrappers. The other 28?
- The “12” from the previous slide are just the mixed precision cases, not *_x() routines: DGEMM has 3, SGEMM has 0, CGEMM has 3, ZGEMM has 6
- DGEMM has 7 (C is D): *_s_d(), *_d_s(), *_x, *_s_s_x(), *_s_s(), *_s_d_x(), *_d_s_x()
  - A is S & B is D, A is D & B is S, A&B are D but internally use X, A&B are S but internally use X, A&B are S, A is S & B is D but internally use X, A is D & B is S but internally use X
- SGEMM has 1
- CGEMM has 7
- ZGEMM has 13

What portions of the 980 routines are used by LAPACK?

- Mostly just 28 cases of matrix-vector multiply:
  - Banded Complex: CGBMV_X, CGBMV2_X, ZGBMV_X, ZGBMV2_X
  - General Complex: CGEMV_X, CGEMV2_X, ZGEMV_X, ZGEMV2_X
  - Hermitian Complex: CHEMV_X, CHEMV2_X, ZHEMV_X, ZHEMV2_X
  - Symmetric Complex: CSYMV_X, CSYMV2_X, ZSYMV_X, ZSYMV2_X
  - Banded Real: DGBMV_X, DGBMV2_X, SGBMV_X, SGBMV2_X
  - General Real: DGEMV_X, DGEMV2_X, SGEMV_X, SGEMV2_X
  - Symmetric Real: DSYMV_X, DSYMV2_X, SSYMV_X, SSYMV2_X

What if we want OUTPUT to be in extended precision?

- The current XBLAS doesn’t suffice
- Then how would one do distributed memory computations?
- Or any case where we need to do part of computation then continue it later

Current XBLAS Implementation large…

- What about double inputs and DD outputs?
- XBLAS was released with LAPACK at first (then separated)
- XBLAS is still a separate entity primarily used for faster mixed precision iterative refinement in LAPACK
- XBLAS is (internal-only) in Intel® Math Kernel Library
  - 980 routines! 7 architecture types!
  - It’s not the number of routines, but the number of cases to optimize
  - Because internal-only, it’s not helpful for developers
  - Performance better than Quad, but still very slow
  - Not really exploiting FMAs!

Present Research…

- How to implement (ideally) a performant version of XBLAS
- How to implement a smaller subset that's easier to maintain
- So far: Implemented DOT and GEMV fast for iterative refinement
- So far: D, DD, Q precisions tuned
- Proposing a new **simplified** standard (externally)
  - No one wants to hand tune 980 kernels… (or commercially maintain them)
  - Perhaps use BLIS methodology of having only a few "micro-kernels"
  - Auto-tuning is useful
  - Auto-test generation might be an issue

Ivy Bridge (Intel® Xeon®CPU E5-2690) @ 2.9 GHz

Problem Size	Speedup of MKL’s XBLAS DDOT over using Intel® Compiler’s real*16	Speedup of a naïve DDOT over using real*16	Speedup of new one DDOT over using real*16 (Speed-up over MKL’s XBLAS in parenthesis)	Speedup of new 4 DDOTs over using real*16 (speed-up over MKL in parenthesis)
120	5.43	19.84	19.04 (3.51)	41.97 (7.73)
1200	5.91	21.9	39.74 (6.72)	44.7 (7.56)
12000	6.21	22.9	46.1 (7.42)	46.8 (7.54)
120000	6.17	22.8	46 (7.45)	46.4 (7.52)

Optimization Notice

Our Hierarchical Proposal

Auxiliary Routines:

- XBLAS_normalize()
- XBLAS_fma(), XBLAS_add(), XBLAS_mult(), etc..

Kernel Routines: (Development work goes here! Small as possible!)

- Assembly-coded and vectorized. **Small Group**

Higher Level Routines (the external XBLAS interface)

- In C, don't require the same tuning, build on top of Kernel Routines
- Most of them like their BLAS counterparts with a couple extra parameters

Kernel Proposal (Level-1)

XBLAS_[prec type]dot[1 or m]_[x input precision w or e][y input precision w or e]_()

“1 or m” is literal. If the unroll factor is four, I’m not suggesting a 4 here. See the GEMV example to understand why

prec_type is usually d (double precision) or z (complex*16) – let’s focus first on these rather than s and c

w (working precision) or e (extended precision)

Kernel Routine Name Level-1 Examples

XBLAS_?dot1_ww_ (AVX performance shown in previous table)

Ex: XBLAS_ddot1_ww_(n, x, incx, dy, incy)

XBLAS_?dotm_ww_ (pseudo-GEMV, AVX performance in previous table)

XBLAS_?dot1_we_

XBLAS_?dotm_we_ (pseudo-GEMV)

dxunroll_factor_for_multiple_dot_products() for the ?dotm routines

Kernel Routine Level-1 Syntax Examples

```
XBLAS_?dot1_we_( [optional char *conj ], // Only if complex*8 and complex*16 versions
        int *n,         // Number of elements in the inner product
        double *x,   // Single vector of double inputs
        int *incx,      // Tuned for incx=1 (suffices to build on top of)
        double *yhi,   // Single vector of double-double inputs stored with just the high parts (separated from low parts)
        double *ylo, // the low parts corresponding
        int *incy,     // Spacing between two values of yhi and ylo- should we have incyhi, incylo???
        double *reshi, // High order result from DD accumulation, only 1 element
        double *reslo )
```

Why separate the low and high parts of DD?

- Note that with incx/incy we can simulate overlapped versions as well
- Example:
  - Suppose our vector Y is (yhi_1, ylo_1,space,yhi_2,ylo_2,space,…)
  - Just set yhi=&Y[0], ylo=&Y[1], incy at 3
- One can go one way, but not the other.

What if we only want D output (like current XBLAS)?

- The results can be "normalized" so that:
    - (double) (high+low) = high
- To get the double-only result, just ignore the "low" part
- We need DD output for any algorithm that requires continuation (parallel)

Kernel Routine Level-1 Syntax Examples cont..

XBLAS_?dotm_we_([optional char *conj], // Only if complex*8 and complex*16 versions

int *n, // Do an inner-product of size n with m vectors against a single DD vector

double *x, // Multiple double input vectors interlaced: x(1,1) = 1st element of vector 1, x(2,1)= 1st element of vector 2, x(1,2) = 2nd element of vector 1, where the leading dimension is incx. Do we need a row-major equivalent?

int *incx, // The “leading dimension”

double *yhi, // Single vector of double-double inputs stored with just the high parts (separated from low parts)

double *ylo, // the low parts corresponding

int *incy, // Spacing between two values of yhi and ylo

double *reshi, // High order result from DD accumulation, n elements

double *reslo,

[int BlockSize]) // We debated if this parameter should be passed in

GEMV on top of these kernels (pseudo-code)

```
 C    Let unb  be the unroll factor for the multiple dot products
       do i = 1, m, unb*
           call XBLAS_ddotm_ww ( n, A(I,1), lda, x, incx, reshi, reslo, [unb] )
           do j = i , i + unb – 1
C                Do the following in DD with aux routines: y ( j ) = alpha * res(j-i+1) + beta * y(j)
C                Could use the ab + cd construct if it is available… Or use the mult and fma constructs:
                call XBLAS_mult ( [beta_hi,beta_lo], [y(j)_hi,y(j)_lo], [res2_hi,res2_lo])
                call XBLAS_fma ( [alpha_hi,alpha_lo], [reshi,reslo], [res2_hi,res2_lo], [res3_hi,res3_lo])
                yhi(j) = res3_hi, ylo(j) = res3_lo
           enddo
      enddo
```

* = Assuming m >= unb and m%unb==0 (can generalize these conditions away easily)

How to generalize separated HI/LOW into a matrix?

- In our pseudo-GEMV kernels, “Y” is one vector isolated into two parts
- In DDOT we have $x^T y$, in DGEMM we have op(A)*op(B). So x=A, y=B
- We need a y=B generalization for a pseudo-GEMM kernel: multiple vectors
- Generalize the separation by considering row_stride as well as column_stride
  - So Bhi and Blo are two matrix inputs for Y
  - Assume the same ldb works for Bhi and Blo?
- We don’t normally have dual-strided arrays, but this enables a full generalization
- XBLAS_DGEMM_WWW (transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc, Ctail, ldct)
- XBLAS_DGEMM_EEE (transa, transb, m, n, k, alpha, A, lda, Atail, ldat, B, ldb, Btail, ldbt, beta, C, ldc, Ctail, ldct) // Should alpha/beta also be DD?

XBLAS Conclusions

- We are working on a new external standard
- We are approaching this from a performance perspective
- The goal is to have as few internal kernels as possible (simplify development)
- Presently, DD and Q kernels have been optimized for AVX/AVX-2
- Please share thoughts/feedback

Batch BLAS Summary Notes

- General Wisdom: On super small problems, a simple OpenMP loop works well and on large problems, doing nothing suffices…
  - However, on medium problems, something more advanced is useful (and present in Intel® MKL)
- Grouping is ideal. The more GEMMs batched, potentially the better perf
  - Also allows faster preprocessing and easier dependency tracking
- The best thing for overall performance is tuning the small serial cases (as in LIBXSMM)
  - LIBXSMM is not currently in Intel® MKL DGEMM_BATCH (nor DIRECT_CALL)

Batched BLAS – Does it always work where we have many small BLAS calls

- DG-FEM/SEM methods of require multiple small BLAS routine within an element
- In this case batched BLAS calls would result into turning a computed bound workload into a bandwidth bound workload as we stream multiple times over the grid
- A higher level interface is possible, but would require an “execution plan” such as available in FFTW
- Our current solution at Intel:
  - Small GEMM library: LIBXSMM, available on github, for vectorization within an element
  - Vectorization across elements might be an alternative
  - small GEMM vs. batched: allows to apply non-BLAS data modifiers while data is still in cache: good for machine learning

Abstract and Motivation
"Improving Performance for Small GEMM Size Problems."

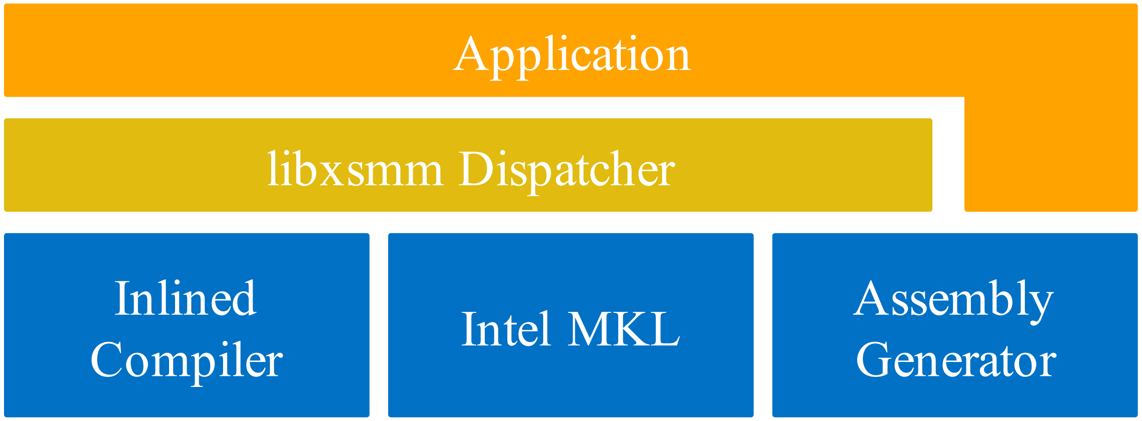
- Problem size is characterized by the M, N, and K parameters
  - Common building block for high order methods
  - Common building block for blocked Sparse Linear Algebra
- A suitable problem size may fall within (M N K)^(1/3) <= 60
  - Intel® Math Kernel Library (Intel® MKL) uses MKL_DIRECT_CALL
  - These sizes are smaller than regular S/DGEMM blocked macro-kernels, therefore MKL_DIRECT_CALL helps, but is only the tip of the iceberg- a lot more performance is necessary/possible

LIBXSMM

Interface (C/C++ and FORTRAN API)

Simplified interface for matrix-matrix multiplications

- $c_{m \times n} = c_{m \times n} + a_{m \times k} * b_{k \times n}$ (also full xGEMM)



License

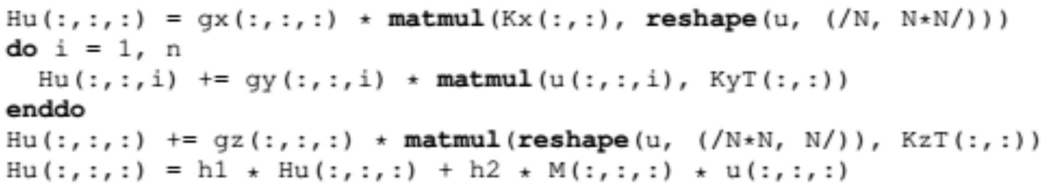
- Open Source Software (BSD 3-clause license)*

*** https://github.com/hfp/libxsmm**

NekBox/5000’s main compute routines (SEM)

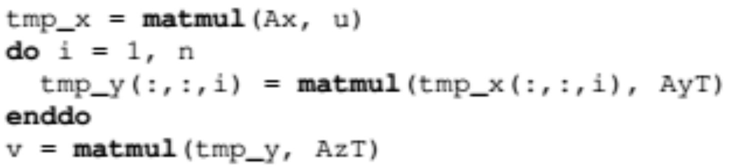
A typical NekBox run spends <1% in sparse computations & communications, ~40% in vector-vector or matrix-vector operations, ~60% matrix-matrix operations. -> Lot’s of reuse when doing small BLAS, but no when using batched BLAS.

Helmholtz operator:

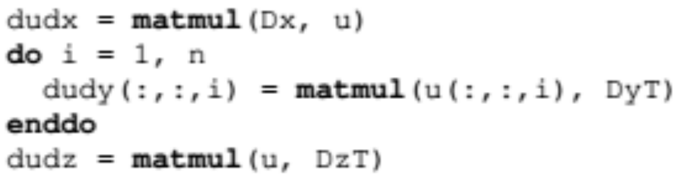


```
Hu(:,:,:) = gx(:,:,:) * matmul(Kx(:,:), reshape(u, (/N, N*N/)))
do i = 1, n
  Hu(:,:,i) += gy(:,:,i) * matmul(u(:,:,i), KyT(:,:))
enddo
Hu(:,:,:) += gz(:,:,:) * matmul(reshape(u, (/N*N, N/)), KzT(:,:))
Hu(:,:,:) = h1 * Hu(:,:,:) + h2 * M(:,:,:) * u(:,:,:)
```

Basis transformation:



```
tmp_x = matmul(Ax, u)
do i = 1, n
  tmp_y(:,:,i) = matmul(tmp_x(:,:,i), AyT)
enddo
v = matmul(tmp_y, AzT)
```

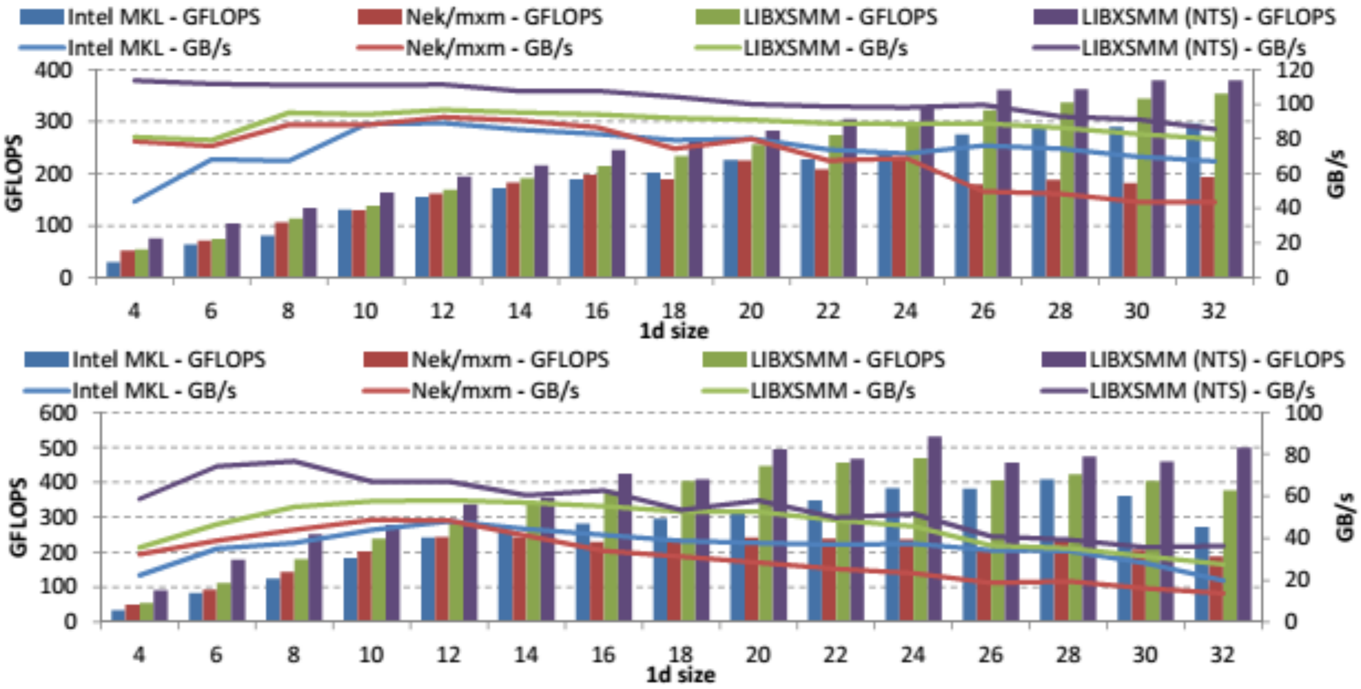


```
dudx = matmul(Dx, u)
do i = 1, n
  dudy(:,:,i) = matmul(u(:,:,i), DyT)
enddo
dudz = matmul(u, DzT)
```

Gradient calculation:

Optimization Notice

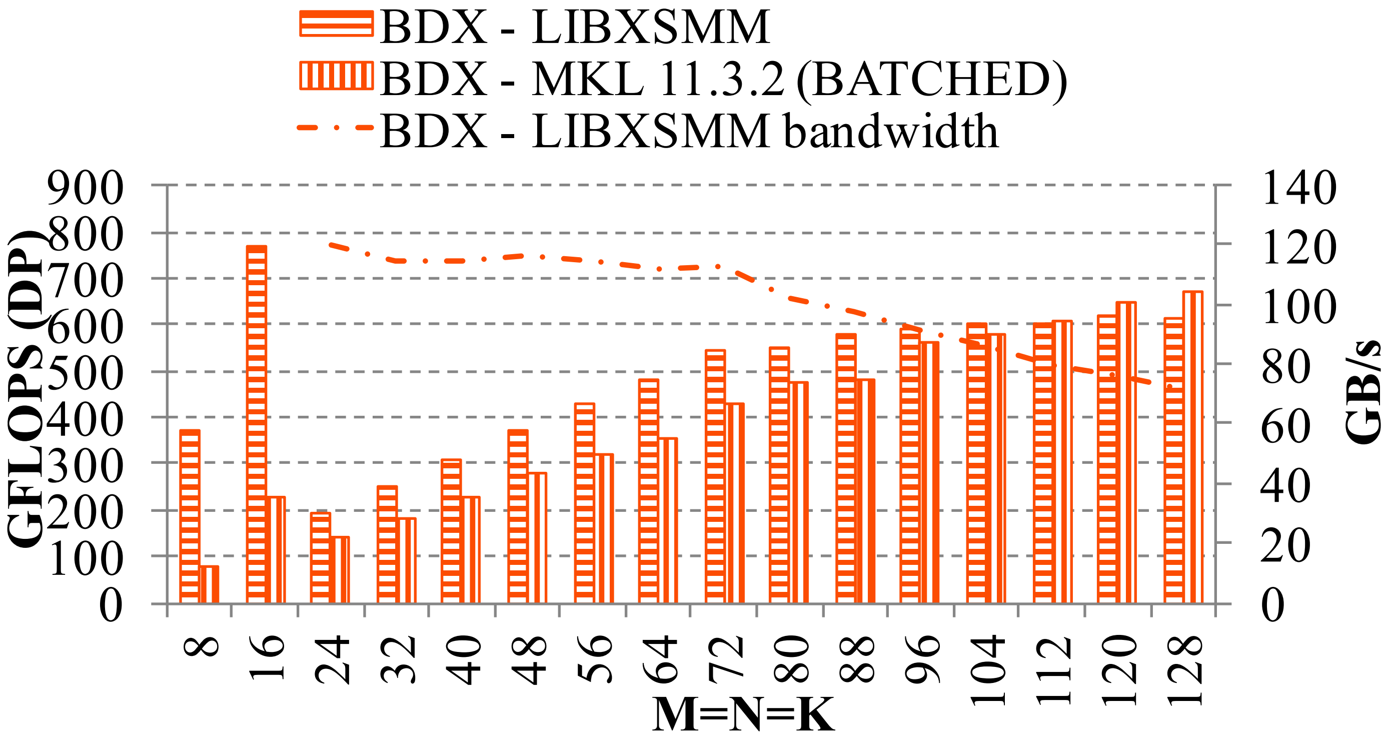
Helmholtz Operator / Basis Transformation



Performance of the Helmholtz operator reproducer (up) and Basis Transformation (bottom) using different implementation for the small matrix multiplications. NTS denotes the usage of non-temporal stores. Measured on Shaheen (32 cores of HSW-EP, 2.3 GHz)

Optimization Notice

LIBXSMM vs. MKL DGEMM_BATCH
2x Xeon E5-2697v4 (BDX) – SMALL BLAS is essential for batched anyway



Batch BLAS- More GEMMs the better

Helmholtz Solution	Time on Broadwell
Soln 1: DGEMM calls time total	0.0406733
Soln 2: K+2 Batches Setup	0.0560226000000
Soln 2: K+2 Batches Total Time	0.0566837
Soln 3: 3 Batches Setup	0.000284754
Soln 3: 3 Batches Kernel Time	0.0144893
Soln 3: 3 Batches Total Time	0.014774
Soln 4: 1 Batch Setup	0.00028225
Soln 4: 1 Batch Kernel Time	0.00515395
Soln 4: 1 Batch Total Time	0.0054362

Batch BLAS Summary Notes – Again…

- General Wisdom: On super small problems, a simple OpenMP loop works well and on large problems, doing nothing suffices…
  - However, on medium problems, something more advanced is useful (and present in Intel® MKL)
- Grouping is ideal. The more GEMMs batched, potentially the better perf
  - Also allows faster preprocessing and easier dependency tracking
- The best thing for overall performance is tuning the small serial cases (as in LIBXSMM)
  - LIBXSMM is not currently in Intel® MKL DGEMM_BATCH (nor DIRECT_CALL)

intel®

experience
what's inside™

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

LIBXSMM Implementation

Three Critical Parts of Technology:

- Highly efficient Frontend (Hans Pabst)
  - BLAS compatible (DGEMM, SGEMM) (even LD_PRELOAD)
  - Support for F77, C89, F2003, C++
  - 2-level code caching
  - Zero-overhead calls into assembly
- Code Generator (Alex Heinecke)
  - Supports all Intel Architectures since 2005, special focus on AVX-512
  - Prefetching across small GEMMs
  - Can generate *.s, inline assembly into *.h/*.c or feeds the JIT encoder
- JIT (Just-In-Time) Encoder (Greg Henry)
  - Encodes an instruction based on basic blocks
  - Very fast as no compilation is involved