Do we need to support alternative data formats for Batched BLAS?

Jonathan Hogg

STFC Rutherford Appleton Laboratory

May 18, 2016
WARNING

- I’ve not implemented this stuff personally
- Slightly outside my area of expertise
- Please shout up if you think I’m wrong!
What is the point of B-BLAS if I can do this?

```c
#pragma omp parallel do
for(int i=0; i<n; ++i)
    dgemm(...);
```
What is the point of B-BLAS if I can do this?

```
#pragma omp parallel do
for (int i = 0; i < n; ++i)
    dgemm(...);
```

Get rid of overheads?
- Only important for small matrices!
Starting Point #2

Sparse Cholesky

- Not uncommon to have 100s-10,000s small matrices
- Can be treated independently
- Sizes often in range $2 \times 1$ to $250 \times 8$.
- Perform partial Cholesky or similar
Small matrices

At these sizes keeping FMA port fed is the hard part.

Four enemies:

- Special op throughput (sqrt, div)
  - Haswell VDIVPD, VSQRTPD 16+ clocks/op
  - Steamroller VDIVPD, VSQRTPD 9+ clocks/op

- Instruction latency / number of avx registers
  - Haswell VFMADDPD 5 clocks; 16 registers
  - Steamroller VFMADDPD 5-6 clocks; 16 registers

- Memory latency
  - Haswell L1=4 cycles, L2=12 cycles, L3=36 cycles, RAM=L3+57ns \( \approx \) 170 cycles

- Memory bandwidth
  - Haswell per cache line L1=0.5 cycles, L2=2.2 cycles, L3=4.7 cycles, RAM \( \approx \) 45 cycles
Focus on: memory

Only 16 AVX registers

- Can’t have that many operations in flight
- Need most of these registers just to hide instruction latency?
- But L1 cache can (almost) keep up
  - Bandwidth $\Rightarrow$ 2 loads/cycle
  - 2 FMA ports/cycle
- L2, L3 and main memory can’t keep up.
- Need to work with L1-size batches for implement batched Cholesky etc. on top?

Avoid wasted memory loads
Padding is a poor solution

We could just pad with zeroes to multiple of vector size.

AVX vector length 8 SP or 4 DP
AVX512 vector length 16 SP or 8 DP
Cuda warpSize 32 SP or 32 DP (but loads more flexible?)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>129</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX</td>
<td>0.25</td>
<td>0.50</td>
<td>0.75</td>
<td>1.00</td>
<td>0.63</td>
<td>0.75</td>
<td>0.88</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
</tr>
<tr>
<td>AVX512</td>
<td>0.13</td>
<td>0.25</td>
<td>0.38</td>
<td>0.50</td>
<td>0.63</td>
<td>0.75</td>
<td>0.88</td>
<td>1.00</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>CUDA</td>
<td>0.03</td>
<td>0.06</td>
<td>0.09</td>
<td>0.13</td>
<td>0.16</td>
<td>0.19</td>
<td>0.22</td>
<td>0.25</td>
<td>0.81</td>
<td>0.89</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>129</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX</td>
<td>0.13</td>
<td>0.25</td>
<td>0.38</td>
<td>0.50</td>
<td>0.63</td>
<td>0.75</td>
<td>0.88</td>
<td>1.00</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>AVX512</td>
<td>0.07</td>
<td>0.13</td>
<td>0.19</td>
<td>0.25</td>
<td>0.31</td>
<td>0.38</td>
<td>0.44</td>
<td>0.50</td>
<td>0.90</td>
<td>0.94</td>
</tr>
<tr>
<td>CUDA</td>
<td>0.03</td>
<td>0.06</td>
<td>0.09</td>
<td>0.13</td>
<td>0.16</td>
<td>0.19</td>
<td>0.22</td>
<td>0.25</td>
<td>0.81</td>
<td>0.89</td>
</tr>
</tbody>
</table>

FP16 even worse!
Padding is a poor solution

We could just pad with zeroes to multiple of vector size.

AVX vector length 8 SP or 4 DP
AVX512 vector length 16 SP or 8 DP
Cuda warpSize 32 SP or 32 DP (but loads more flexible?)

DP (useful loads) / (total loads):

<table>
<thead>
<tr>
<th>m</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>129</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX</td>
<td>0.25</td>
<td>0.50</td>
<td>0.75</td>
<td>1.00</td>
<td>0.63</td>
<td>0.75</td>
<td>0.88</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
</tr>
<tr>
<td>AVX512</td>
<td>0.13</td>
<td>0.25</td>
<td>0.38</td>
<td>0.50</td>
<td>0.63</td>
<td>0.75</td>
<td>0.88</td>
<td>1.00</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>CUDA</td>
<td>0.03</td>
<td>0.06</td>
<td>0.09</td>
<td>0.13</td>
<td>0.16</td>
<td>0.19</td>
<td>0.22</td>
<td>0.25</td>
<td>0.81</td>
<td>0.89</td>
</tr>
</tbody>
</table>

SP (useful loads) / (total loads):

<table>
<thead>
<tr>
<th>m</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>129</th>
<th>257</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX</td>
<td>0.13</td>
<td>0.25</td>
<td>0.38</td>
<td>0.50</td>
<td>0.63</td>
<td>0.75</td>
<td>0.88</td>
<td>1.00</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>AVX512</td>
<td>0.07</td>
<td>0.13</td>
<td>0.19</td>
<td>0.25</td>
<td>0.31</td>
<td>0.38</td>
<td>0.44</td>
<td>0.50</td>
<td>0.90</td>
<td>0.94</td>
</tr>
<tr>
<td>CUDA</td>
<td>0.03</td>
<td>0.06</td>
<td>0.09</td>
<td>0.13</td>
<td>0.16</td>
<td>0.19</td>
<td>0.22</td>
<td>0.25</td>
<td>0.81</td>
<td>0.89</td>
</tr>
</tbody>
</table>

FP16 even worse!

Generally want \( m > 2.5 \times \text{vector length} \)
The interleaved solution

- If we have vector length operations of same type/size

4 matrices of size $2 \times 2$:

\[
\begin{array}{cccc}
  a_{00}^{(0)} & a_{00}^{(1)} & a_{00}^{(2)} & a_{00}^{(3)} \\
  a_{10}^{(0)} & a_{10}^{(1)} & a_{10}^{(2)} & a_{10}^{(3)} \\
  a_{01}^{(0)} & a_{01}^{(1)} & a_{01}^{(2)} & a_{01}^{(3)} \\
  a_{11}^{(0)} & a_{11}^{(1)} & a_{11}^{(2)} & a_{11}^{(3)}
\end{array}
\]
The interleaved solution

- If we have vector length operations of same type/size

4 matrices of size $2 \times 2$:

$$
\begin{align*}
& a_{00}^{(0)} a_{00}^{(1)} a_{00}^{(2)} a_{00}^{(3)} a_{10}^{(1)} a_{10}^{(2)} a_{10}^{(3)} a_{01}^{(0)} a_{01}^{(1)} a_{01}^{(2)} a_{01}^{(3)} a_{11}^{(1)} a_{11}^{(2)} a_{11}^{(3)} \\
& a_{01}^{(0)} a_{01}^{(1)} a_{01}^{(2)} a_{01}^{(3)} a_{11}^{(1)} a_{11}^{(2)} a_{11}^{(3)}
\end{align*}
$$

- Easy to load the $a_{ij}$-th entry of vec-len matrices in one go
- All operations with different vector operations are independent
- No need for horizontal reductions
- No wasted loads (if multiple of vector size in batch)
Sparse Cholesky: can we use it?

% small nodes vectorized

Problem

fully
partial

Jonathan Hogg, STFC Rutherford Appleton Laboratory
Transform to interleaved?

Just use a more complicated load...

<table>
<thead>
<tr>
<th>Data set size</th>
<th>VMOVAP</th>
<th>_mm256_set_pd()</th>
<th>VGATHER</th>
</tr>
</thead>
<tbody>
<tr>
<td>234 KB</td>
<td>[2.874, 3.271]</td>
<td>[4.143, 4.869]</td>
<td>[10.333, 11.663]</td>
</tr>
<tr>
<td>7.64 MB</td>
<td>[3.732, 3.954]</td>
<td>[4.243, 4.934]</td>
<td>[10.326, 10.995]</td>
</tr>
</tbody>
</table>

- Overhead >40% on L2, >10% on L3
- New VGATHER only worthwhile from Main memory?
Special function units?

Do we need to consolidate DIV, SQRT to be vectorized?

Single TRSM solve $XA = B$, $A$ is $n \times n$ and $X, B$ are $m \times n$:

- At least $n$ DIVs
- At least $mn$ FMAs

$\Rightarrow$ DTRSM bound on scalar DIV throughput if $m \leq 64$.

$(64 = 16$ clocks/DIV $\times$ veclen 4 for FMA)$

$\Rightarrow$ Reduced to $m \leq 16$ if we can use vector DIV.
Special function units?

Do we need to consolidate DIV, SQRT to be vectorized?

Single TRSM solve $XA = B$, $A$ is $n \times n$ and $X, B$ are $m \times n$:

- At least $n$ DIVs
- At least $mn$ FMAs

$\Rightarrow$ DTRSM bound on scalar DIV throughput if $m \leq 64$.

$(64 = 16 \text{ clocks/DIV} \times \text{veclen 4 for FMA})$

$\Rightarrow$ Reduced to $m \leq 16$ if we can use vector DIV.

- For TRSM can get this within single matrix - but not for Cholesky (RSQRT is on critical path + also need a DIV per column).
- Can do this without interleaving - but fiddly!
- Interleaved data makes this trivial.
- But we might be bound on memory loads anyway?
Why not a separate "microblas"?

- This might be a good idea implementation-wise
- But we’re discussing an API specification
- Balance: extra implementation cost vs benefit

If we don’t support very small matrices what’s the point?

- Clearly batched BLAS are useful on GPUs:
  - Can’t write own fast code - need access to physical register allocation that isn’t available without going below PTX.
  - Kernel launch overheads.
- Less useful on CPUs if we do the trivial:
  - Running multiple matrices with OpenMP not hard.
  - DIY is better with existing parallel schemes (eg tasks).
- On CPUs need to deliver a benefit on a single core level
  - Only run argument checking overhead once (or not at all!)
  - Use multiple matrices to hide memory and instruction latencies
  - Above advantages only really significant for very small matrices?
Support for memory alignment

Without memory alignment:
- Need “top” and “tail” loops (handle unaligned parts)
- These can be large overhead on small matrices

With memory alignment:
- Users would need to promise aligned vectors
- Leading dimension multiple of vector size
- To avoid tail loop, need to zero out unwanted part
- Users can often meet these conditions cheaply!
- Probably faster even without explicit exploitation

Notes:
- No native support for memory alignment in Fortran :(
Suggested Changes

Change: support memory alignment

- Add “aligned” flag. User promises:
  1. First element of any matrix is aligned
  2. lda is multiple of vector length
- Easily ignored by vendors if no desire to implement
- Easy win?
Suggested Changes

Change: support memory alignment

- Add “aligned” flag. User promises:
  1. First element of any matrix is aligned
  2. lda is multiple of vector length
- Easily ignored by vendors if no desire to implement
- Easy win?

Change: support interleaved data format

- Either “interleaved” flag; or
- Add an “lda-like” variable for the interleaving.
  - Easy to detect when this is 1 and run traditional code.
- Extra work for implementors
- Lack may drive people to implement own code instead
- Need to assess cost/benefit?
One more thing...

Can we tack a 2d sparse scatter on the end of _gemm?

- i.e. provide arrays rlist, clist and do
  
  \[ c(rlist(i), clist(j)) + = \sum_k a(i, k) \times b(k, j) \]

- Avoid \( C \) falling out of cache
- Hide indirection latency behind arithmetic
- Essential to sparse direct solvers
Discuss.
## Aligned vs Unaligned Loads

<table>
<thead>
<tr>
<th>Data set size</th>
<th>VMOVAP aligned</th>
<th>VMOVUP unaligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>234 KB</td>
<td>[2.897, 3.462]</td>
<td>[2.886, 3.427]</td>
</tr>
<tr>
<td>7.64 MB</td>
<td>[3.707, 3.968]</td>
<td>[3.659, 3.946]</td>
</tr>
<tr>
<td>764 MB</td>
<td>[7.967, 7.979]</td>
<td>[7.956, 7.969]</td>
</tr>
</tbody>
</table>

\[ \text{mean-1sd, mean+1sd} \times 10^{-10} \text{ per load} \]

⇒ Alignment of data important, not instruction?
## Aligned vs Unaligned Loads

<table>
<thead>
<tr>
<th>Data set size</th>
<th>VMOVAP aligned</th>
<th>VMOVUP unaligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>234 KB</td>
<td>[2.897, 3.462]</td>
<td>[2.886, 3.427]</td>
</tr>
<tr>
<td>7.64 MB</td>
<td>[3.707, 3.968]</td>
<td>[3.659, 3.946]</td>
</tr>
<tr>
<td>764 MB</td>
<td>[7.967, 7.979]</td>
<td>[7.956, 7.969]</td>
</tr>
</tbody>
</table>

[mean-1sd, mean+1sd]×10^{-10} per load

⇒ Alignment of data important, not instruction?

**BUT:** Alignment allows direct load in eg VFMADDPD.

<table>
<thead>
<tr>
<th>VADDPPD w mem operand</th>
<th>234 KB</th>
<th>7.64 MB</th>
<th>764 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[6.607, 7.788]</td>
<td>[6.849, 7.349]</td>
<td>[7.019, 7.153]</td>
</tr>
</tbody>
</table>
Summary of results

- Aligned vs unaligned load instructions does not matter too much.
- Alignment of memory does: 5–15% penalty.
- Transforming to interleaved in registers: 40% overhead.
- Padding bad solution for small matrices: Need $m \geq 2.5v_{len}$ to be at least 75% efficient. Length 32 vector $\Rightarrow m \geq 72$