On the High Performance Implementation of Quaternionic Matrix Operations

wavefunction91.github.io

SIAM-CSE 2019
David Williams-Young
Scalable Solvers Group
Computational Research Division
Lawrence Berkeley National Lab

February 28, 2019
Problem Motivation

- Moving towards the end of Moore’s law: Simply applying existing algorithms and data structures not sufficient.
- Quaternion symmetry is very common in many scientific and engineering disciplines, especially those whose target involves physical space.
- Much research has been afforded to real / complex linear algebra algorithms to exploit this symmetry (Dongerra, et al, 1984; Shiozaki, 2018)

*The quaternion algebra is [...] somewhat complicated, and its computation cannot be easily mapped to highly optimized linear algebra libraries such as BLAS and LAPACK.*
How can we leverage techniques such as auto-tuning and microarchitecture optimization to provide optimized implementations of quaternion linear algebra software?
This talk will attempt to answer (discuss) three questions:

- What are quaternions and why do we care?
This talk will attempt to answer (discuss) three questions:

• What are quaternions and why do we care?
• What possible use could I have for matrices of quaternions?
This talk will attempt to answer (discuss) three questions:

- What are quaternions and why do we care?
- What possible use could I have for matrices of quaternions?
- What does all of this have to do with auto-tuning?
Quaternions: Formally

Quaternions are defined as the set \( \mathbb{H} \) of all \( q \) such that

\[
q = q^0 e_0 + q^1 e_1 + q^2 e_2 + q^3 e_3, \quad q^0, q^1, q^2, q^3 \in \mathbb{R}
\]

with

\[
e_0 e_j = e_j e_0 = e_j, \quad j \in \{0, 1, 2, 3\},
\]

\[
e_i e_j = -\delta_{ij} e_0 + \sum_{k=1}^{3} \varepsilon_{ij}^k e_k, \quad i, j \in \{1, 2, 3\},
\]
Quaternions: Formally

\[ a, b, c \in \mathbb{R}, \quad c = ab, \quad [a, b] = 0 \]

\[ w, v, z \in \mathbb{C}, \quad z = wv = w^0 v^0 - w^1 v^1 + (w^0 v^1 + w^1 v^0)i \quad [w, v] = 0 \]
Quaternions: Formally

\[ a, b, c \in \mathbb{R}, \quad c = ab, \quad [a, b] = 0 \]

\[ w, v, z \in \mathbb{C}, \quad z = wv = w^0v^0 - w^1v^1 + (w^0v^1 + w^1v^0)i \quad [w, v] = 0 \]

\[ p, q, r \in \mathbb{H} \]

\[ r = pq = \left( p^0q^0 - \sum_{i=1}^{3} p^i q^i \right) e_0 + \sum_{k=1}^{3} \left( p^0q^k + p^k q^0 + \sum_{i,j=1}^{3} \varepsilon_{ij}^k p^i q^j \right) e_k, \]

\[ [p, q] = \sum_{i,j,k=1}^{3} \varepsilon_{ij}^k \left( p^i q^j - p^j q^i \right) e_k \neq 0. \]
Quaternion Applications: Spacial Rotations

Topologically, the set of unit quaternions (versors)
\[ \mathbb{V} = \{ v \in \mathbb{H} \text{ s.t. } ||v|| = 1 \} \]
is \( S^3 \), and thus isomorphic to \( SU(2) \) which provides a double cover of \( SO(3) \) (rotations in \( \mathbb{R}^3 \)).

We may describe spatial rotations in \( \mathbb{R}^3 \) via

\[
\begin{align*}
    r &\in \mathbb{R}^3 \mapsto r^H = r^1 e_1 + r^2 e_2 + r^3 e_3 \\
    R(\hat{e}, \theta) &\in SO(3) \mapsto \pm v = \pm \exp \left( \frac{\theta}{2} (\hat{e}^1 e_1 + \hat{e}^2 e_2 + \hat{e}^3 e_3) \right)
\end{align*}
\]
such that

\[
    r' = R(\hat{e}, \theta) r \quad \mapsto \quad r'^H = vr^H v^{-1}
\]
Quaternion Applications: Spacial Rotations

The SO(3) cover has found extensive exploitation in computer graphics / vision

- \((v^0, v^1, v^2, v^3)\) (4 real numbers) vs. \([a \ b \ c]
  
  \begin{bmatrix}
    d & e & f \\
    g & h & i
  \end{bmatrix}
\) (9 real numbers)

- \(v_1, v_2 \in \mathbb{H}, v_1 v_2\) (16 FLOPs) vs \(R_1, R_2 \in SO(3), R_1 R_2\) (27 FLOPs)

- SLERP (Spherical Linear Interpolation)
Matrices of Quaternions

The algebra generated by \( \{ e_0, e_1, e_2, e_3 \} \) is identical to the algebra generated by the Pauli matrices, thus \( \mathbb{H} \cong \langle SU(2) \rangle \subset M_2(\mathbb{C}) \),

\[
e_0 \leftrightarrow \sigma_0, \quad e_1 \leftrightarrow i\sigma_3, \quad e_2 \leftrightarrow i\sigma_2, \quad e_3 \leftrightarrow i\sigma_1,
\]

with

\[
\sigma_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.
\]

Such that

\[
q \leftrightarrow q_C = \begin{bmatrix} q^0 + q^1i & q^2 + q^3i \\ -q^2 + q^3i & q^0 - q^1i \end{bmatrix} = \begin{bmatrix} \frac{q^0}{q\bar{q}} & \frac{q^1}{q\bar{q}} \\ -\frac{q^1}{q\bar{q}} & \frac{q^0}{q\bar{q}} \end{bmatrix} \in M_2(\mathbb{C})
\]
Matrices of Quaternions

The set of quaternion matrices, $\mathbb{M}_N(\mathbb{H})$, is defined by

$$Q = Q^0 e_0 + Q^1 e_1 + Q^2 e_2 + Q^3 e_3, \quad Q^0, Q^1, Q^2, Q^3 \in \mathbb{M}_N(\mathbb{R}).$$

Examining the $\mathbb{M}_2(\mathbb{C})$ representation of a particular element $(Q_{\mu\nu})_C = Q^0_{\mu\nu}\sigma_0 + iQ^1_{\mu\nu}\sigma_3 + iQ^2_{\mu\nu}\sigma_2 + iQ^3_{\mu\nu}\sigma_1.$

which yields the Kronecker structure

$$Q_C = Q^0 \otimes \sigma_0 + Q^1 \otimes i\sigma_3 + Q^2 \otimes i\sigma_2 + Q^3 \otimes i\sigma_1$$

$$= \begin{bmatrix} Q^0 & Q^1 \\ -Q^1 & Q^0 \end{bmatrix} \in \mathbb{M}_{2N}(\mathbb{C}),$$
Matrices of Quaternions

\[ Q \in \mathbb{M}_N(\mathbb{H}) \leftrightarrow \begin{bmatrix} Q^0 & Q^1 \\ -Q^1 & Q^0 \end{bmatrix} \in \mathbb{M}_{2N}(\mathbb{C}), \]

- Ubiquitous in quantum chemistry / nuclear physics (time-reversal symmetry).
- Applications in image processing and machine learning (quaternion PCA, etc).

Formal theory for quaternion linear algebra has been developed

- QR Algorithm
- Diagonalization, SVD
- LU, Cholesky, LDLH Factorizations
Performance Considerations

Table: Real floating point operations (FLOPs) comparison for elementary arithmetic operations using $\mathbb{H}$ and $\mathbb{M}_2(\mathbb{C})$ data structures.

<table>
<thead>
<tr>
<th>Operation</th>
<th>FLOPs in $\mathbb{H}$</th>
<th>FLOPs in $\mathbb{M}_2(\mathbb{C})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Multiplication</td>
<td>16</td>
<td>32</td>
</tr>
</tbody>
</table>

$p + q \leftrightarrow p_C + q_C,$

$pq \leftrightarrow p_C q_C,$
### Performance Considerations

**Table:** Real floating point operations (FLOPs) comparison for common linear algebra operations using $\mathbb{M}_N(\mathbb{H})$ and $\mathbb{M}_{2N}(\mathbb{C})$ data structures.

<table>
<thead>
<tr>
<th>Operation</th>
<th>FLOPs in $\mathbb{M}_N(\mathbb{H})$</th>
<th>FLOPs in $\mathbb{M}_{2N}(\mathbb{C})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>$4N^2$</td>
<td>$8N^2$</td>
</tr>
<tr>
<td>Multiplication</td>
<td>$16N^3$</td>
<td>$32N^3$</td>
</tr>
</tbody>
</table>

\[ P + Q \leftrightarrow P_\mathbb{C} + Q_\mathbb{C}, \]

\[ PQ \leftrightarrow P_\mathbb{C} Q_\mathbb{C}, \]
Performance Considerations

Quaternion arithmetic offers:

- 0.5x required FLOPs
- 0.5x memory footprint (4x / 8x floats)
- 2x arithmetic intensity (FLOPs / byte)

We should be using quaternion arithmetic!
• gh/wavefunction91/HAXX
• Optimized C++14 library for quaternion arithmetic
• HBLAS: Optimized quaternionic BLAS functionality
• H LAPACK: Optimized quaternionic LAPACK functionality (in progress)
• Intrinsics + Assembly kernels
The most fundamental linear algebra operation is the general matrix multiply (GEMM).
Quaternion Matrix Multiplication

The most fundamental linear algebra operation is the general matrix multiply (GEMM).

Great! Quaternion GEMM $\implies$ HP Quaternion Linear Algebra, Right?
Quaternion Matrix Multiplication

The most fundamental linear algebra operation is the general matrix multiply (GEMM).

Great! Quaternion GEMM $\Rightarrow$ HP Quaternion Linear Algebra, Right?

![Graphs showing the relationship between quaternion problem dimension and time/s.](image-url)
Quaternion Matrix Multiplication

The most fundamental linear algebra operation is the general matrix multiply (GEMM).

Great! Quaternion GEMM \implies HP Quaternion Linear Algebra, Right?
Reference GEMM

Algorithm 0: Reference GEMM

Input : $A \in \mathbb{M}_{m,k}(\mathbb{F})$, $B \in \mathbb{M}_{k,n}(\mathbb{F})$, 
        $C \in \mathbb{M}_{m,n}(\mathbb{F})$, 
        Scalars $\alpha, \beta \in \mathbb{F}$

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

for $j = 1 : n$ do

1. Load $C_j = C(:,j)$
2. $C_j = \beta C_j$
   for $l = 1 : k$ do
3. Load $A_l = A(:,l)$
4. $C_j = C_j + \alpha A_l B_{lj}$
   end
5. Store $C_j$
end

Pros:

✓ Able to implement in an afternoon
✓ Architecture agnostic

Cons:

✗ No caching of $B$
✗ Reloads all of $A$ for each $C_j$
✗ For large $m, k$, $A$ load boots $C_j$ from cache
✗ Relies on optimizing compiler for SIMD, FMA, etc
✗ (Scalable) parallelism is non-trivial
✗ Not tunable
High-Performance Matrix-Matrix Multiplication

A layered (Goto-style) algorithm significantly improves performance

**Pros:**

✓ Caches parts of $A, B$ for maximum resuability
✓ Factors architecture specific μ-ops into single micro-kernel
✓ Obvious avenue for SMP
✓ **Tunable!**

**Cons:**

✗ Significantly more complicated than naive algorithm
✗ Requires allocation of auxiliary memory
✗ Micro-kernel must be written for each architecture

High-Performance Quaternionic GEMM (HGEMM)

The optimized implementation of GEMM in $\mathbb{H}$BLAS utilizes the Goto algorithm. In essence, Goto’s original algorithm may be extended to $\mathbb{H}$ by specialization of two sets routines:

- Micro-kernels which perform the $\mathbb{H}$ rank-1 update (assembly / intrinsics)
- Efficient matrix packing routines (intrinsics)

and optimization of 3 caching parameters, $m_c$, $n_c$ and $k_c$ for the architecture of interest.
High-Performance Quaternionic GEMM (HGEMM)

The optimized implementation of GEMM in \( \mathbb{H} \)BLAS utilizes the Goto algorithm. In essence, Goto’s original algorithm may be extended to \( \mathbb{H} \) by specialization of two sets routines:

- **Micro-kernels which perform the \( \mathbb{H} \) rank-1 update (assembly / intrinsics)**
- **Efficient matrix packing routines (intrinsics)**

and optimization of 3 caching parameters, \( m_c, n_c \) and \( k_c \) for the architecture of interest.
High-Performance Quaternionic GEMM (HGEMM)

The optimized implementation of GEMM in HBLAS utilizes the Goto algorithm. In essence, Goto’s original algorithm may be extended to H by specialization of two sets routines:

- Micro-kernels which perform the H rank-1 update (assembly / intrinsics)
- Efficient matrix packing routines (intrinsics)

and optimization of 3 caching parameters, $m_c$, $n_c$ and $k_c$ for the architecture of interest.
Optimization of Cache Parameters

- OpenTuner: An open source Python framework for auto-tuning
- Register blocks fixed (on AVX / AVX2), \( n_r = m_r = 2 \).
- Integer discretize \( m_c, k_c \in \{2^n\}_{n=3}^{12}, n_c \in \{2^n\}_{n=5}^{16} \)
- Find \( \{m_c, n_c, k_c\} \) which minimizes run time (maximizes GFLOP/s)
  - Average over 5 cold (cache invalidated) runs on select matrix sizes (500,1k,2k,4k)

Possible to brute force optimize, but not convenient!
Optimization of Cache Parameters

![Graph showing optimization of cache parameters with log2Mc and log2Kc on the axes and GFLOPs Nc = 128 as the color scale.](image-url)
Optimization of Cache Parameters

GFLOPs $N_c = 256$

- $\log_2 M_c$
- $\log_2 K_c$
- GFLOPs

- $N_c = 256$
- $20.7$ to $21.5$
Optimization of Cache Parameters

![Graph showing the relationship between $\log_2 M_c$ and $\log_2 K_c$ with GFLOPs $N_c = 512$. The y-axis represents $\log_2 K_c$ ranging from 6.0 to 10.0, and the x-axis represents $\log_2 M_c$ ranging from 6.0 to 10.0. The color scale ranges from 20.7 to 21.5.]
Optimization of Cache Parameters

![Heatmap showing GFLOPs for different combinations of $\log_2 M_c$ and $\log_2 K_c$ with $N_c = 1024$.]
Optimization of Cache Parameters

GFLOPs $N_c = 2048$
Optimization of Cache Parameters

The figure shows a heatmap with the $\log_2 N_c = 4096$ GFLOPs. The x-axis represents $\log_2 M_c$, and the y-axis represents $\log_2 K_c$. The color scale indicates the performance in GFLOPs.
AVX Optimized HGEMM Implementation

Intel Sandy Bridge (L1d: 32k, L1i: 32k, L2: 256k, L3: 20480k)

OpenTuner results:

• 10 tests x 10 runs (∼1 hour vs 10 hours brute force)
• $m_c = k_c = 64$
• $n_c = 1024$

![Graph showing performance comparison between HGEMM-Ref, HGEMM-OptAVX, and ZGEMM-MKL](image1)

![Graph showing GFLOP/s comparison](image2)
Conclusions

- With instruction sets newer than AVX, high-performance quaternionic linear algebra is possible and a viable alternative to complex linear algebra for appropriate problems.

- Goto’s algorithm + auto-tuning drastically improves performance
  - Impractical with reference implementations.
Future Work

• Fill out \texttt{HBLAS} and \texttt{HLAPACK} coverage of the BLAS and LAPACK standards.

• Package autotuner and tuning methodology to automate optimization of caching parameters (\ldots) on new architectures.

• Address parallelism (SMP + MPI)
Acknowledgments

- Xiaosong Li (UW)
- $$$ ACI-1547580 (NSF), OAC-1663636 (NSF to XL)
- $$$ LAB 17-1775 (DOE-BES)
- Benjamin Pritchard (MolSSI)
- Edward Valeev (VT)
- Wissam Sid-Lakhdar (LBNL)
- Organizers
- Audience