The Landscape of High-Performance Tensor Contractions

Paul Springer and Paolo Bientinesi

Aachen Institute for Advanced Study in Computational Engineering Science

Atlanta, Feb. 24th 2017
Introduction

A tensor is a multidimensional array:
- 0-order tensor: scalar $\alpha$

Tensor contractions can be thought of as generalized GEMMs.

Three approaches to tensor contractions:
- Nested loops
- Loops over GEMM (LoG)
- Transpose-Transpose-GEMM-Transpose (TTGT)

We propose a novel approach: GETT.

Akin to a high-performance GEMM implementation.
Introduction

- A tensors is a multidimensional array:
  - 0-order tensor: scalar $\alpha$
  - 1-order tensor: vector $\mathcal{A}_{i_1}$

Tensor contractions can be thought of as generalized GEMMs.

Three approaches to tensor contractions:
- Nested loops
- Loops over GEMM (LoG)
- Transpose-Transpose-GEMM-Transpose (TTGT)

We propose a novel approach: GETT

Akin to a high-performance GEMM implementation
A tensors is a multidimensional array:
- 0-order tensor: scalar $\alpha$
- 1-order tensor: vector $A_{i_1}$
- 2-order tensor: matrix $A_{i_1,i_2}$
A tensors is a multidimensional array:

- 0-order tensor: scalar $\alpha$
- 1-order tensor: vector $\mathcal{A}_{i_1}$
- 2-order tensor: matrix $\mathcal{A}_{i_1,i_2}$
- $n$-order tensor: $\mathcal{A}_{i_1,i_2,...,i_n}$
A tensor is a multidimensional array:

- 0-order tensor: scalar $\alpha$
- 1-order tensor: vector $A_{i_1}$
- 2-order tensor: matrix $A_{i_1,i_2}$
- $n$-order tensor: $A_{i_1,i_2,\ldots,i_n}$

Tensor contractions can be thought of as generalized GEMMs

---

A tensor is a multidimensional array:
- 0-order tensor: scalar $\alpha$
- 1-order tensor: vector $A_{i_1}$
- 2-order tensor: matrix $A_{i_1,i_2}$
- $n$-order tensor: $A_{i_1,i_2,...,i_n}$

Tensor contractions can be thought of as generalized GEMMs

Three approaches to tensor contractions:
- Nested loops
- Loops over GEMM (LoG)
- Transpose-Transpose-GEMM-Transpose (TTGT)

A tensors is a multidimensional array:
- 0-order tensor: scalar $\alpha$
- 1-order tensor: vector $\mathcal{A}_{i_1}$
- 2-order tensor: matrix $\mathcal{A}_{i_1,i_2}$
- $n$-order tensor: $\mathcal{A}_{i_1,i_2,...,i_n}$

Tensor contractions can be thought of as generalized GEMMs

Three approaches to tensor contractions:
- Nested loops
- Loops over GEMM (LoG)
- Transpose-Transpose-GEMM-Transpose (TTGT)

We propose a novel approach: GETT$^1$
- Akin to a high-performance GEMM implementation

---

Outline

- Approaches to Tensor Contractions:
  - Loops over GEMM (LoG)
  - Transpose-Transpose-GEMM-Transpose (TTGT)
  - GEMM-like Tensor-Tensor Multiply (GETT)

- Tensor Contraction Code Generator

- Performance Evaluation

---

\(^2\)Source code available at: https://github.com/HPAC/tccg
Loop over GEMM (LoG)

Conceptual Idea

Identify 2D subtensors and contract them via GEMM

\[ C_{m_1,n_1} \leftarrow \sum_{k_1} A_{m_1,k_1} B_{k_1,n_1} \]
Conceptual Idea

Identify 2D subtensors and contract them via GEMM

\[ C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1} \]
Conceptual Idea

Identify 2D subtensors and contract them via GEMM

\[ C_{m_1, n_1} \leftarrow A_{m_1, k_1} B_{k_1, n_1} \]

\[
\text{gemm}(M_1, N_1, K_1, A[:,:), B[:,:), C[:,:])
\]
Conceptual Idea

Identify 2D subtensors and contract them via GEMM

- $C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1}$
- $C_{m_1,m_2,n_1} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_1}$
Loop over GEMM (LoG)

Conceptual Idea
Identify 2D subtensors and contract them via GEMM

\[ C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1} \]
\[ C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1} \]

\[
gemm(M_1 \times M_2, N_1, K_1, A[:,:,:], B[:,:,:], C[:,:,:])
\]
Loop over GEMM (LoG)

Conceptual Idea
Identify 2D subtensors and contract them via GEMM

- $C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1}$
- $C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1}$
- $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_2,n_1}$

```
for ( m_2 = 0; m_2 < M_2; m_2++ )
    for ( n_1 = 0; n_1 < N_1; n_1++ )
        gemm( M_1, N_2, K_1, A[:,:,:], B[:,:,:], C[:,:,:] )
```
Loop over GEMM (LoG)

Conceptual Idea

Identify 2D subtensors and contract them via GEMM

\[ C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1} \]
\[ C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1} \]
\[ C_{m_1,n_1,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_2,n_1} \]

for ( \( m_2 = 0; \ m_2 < M_2; \ m_2++ \) )
    for ( \( n_2 = 0; \ n_2 < N_2; \ n_2++ \) )
        gemm (\( M_1 \), \( N_1 \), \( K_1 \), \( A[:,m_2,:] \), \( B[:,n_2,:) \), \( C[:,n_2,m_2] \))
Loop over GEMM (LoG)

Conceptual Idea

Identify 2D subtensors and contract them via GEMM

- \( C_{m_1, n_1} \leftarrow A_{m_1, k_1} B_{k_1, n_1} \)
- \( C_{(m_1, m_2), n_1} \leftarrow A_{(m_1, m_2), k_1} B_{k_1, n_1} \)
- \( C_{m_1, n_1, n_2, m_2} \leftarrow A_{m_1, m_2, k_1} B_{k_1, [n_2], n_1} \)

```c
for ( m2 = 0; m2 < M2; m2++ )
    gemm_batch (M1, N1, K1, A[: , m2 , :], B[:, n2 , :], C[:, n2 , m2], N2)
```
Loop over GEMM (LoG)

Conceptual Idea

Identify 2D subtensors and contract them via GEMM

- $C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1}$
- $C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1}$
- $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_2,n_1}$

```c
for ( n_2 = 0; n_2 < N_2; n_2++ )
    gemm_batch( M_1, N_1, K_1, A[::,m_2,:], B[::,n_2,:], C[::,n_2,m_2], M_2 )
```
Loop over GEMM (LoG)

Conceptual Idea

Identify 2D subtensors and contract them via GEMM

- $C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1}$
- $C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1}$
- $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_2,n_1}$
- $C_{m_1,n_1} \leftarrow A_{k_1,m_1,k_2} B_{k_2,n_1,k_1}$

for ($k_1 = 0; k_1 < K_1; k_1++$)
gemm($M_1$, $N_1$, $K_2$, $A[k_1,:,]$, $B[:,k_1]$, $C[:,:]$)
Loop over GEMM (LoG)

Conceptual Idea
Identify 2D subtensors and contract them via GEMM

- \( C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1} \)
- \( C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1} \)
- \( C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_2,n_1} \)
- \( C_{m_1,n_1} \leftarrow A_{k_1,m_1,k_2} B_{k_2,n_1,k_1} \)

\[
\text{for} \ (k_1 = 0; \ k_1 < K_1; \ k_1++) \\
gemm(M_1, \ N_1, \ K_2, \ A\{k_1, \ldots\}, \ B\{\ldots, k_1\}, \ C\{\ldots\})
\]
Loop over GEMM (LoG)

Conceptual Idea

Identify 2D subtensors and contract them via GEMM

\[ C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1} \]
\[ C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1} \]
\[ C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_2,n_1} \]
\[ C_{m_1,n_1} \leftarrow A_{k_1,m_1,k_2} B_{k_2,n_1,k_1} \]

\[
\text{for } (k_2 = 0; k_2 < K_2; k_2++)
\]
\[
gemm(M_1, N_1, K_1, A[:, :, k_2]^T, B[k_2, :, :]^T, C[:, :])
\]
Loop over GEMM (LoG)

Conceptual Idea

Identify 2D subtensors and contract them via GEMM

- \( C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1} \)
- \( C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1} \)
- \( C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_2,n_1} \)
- \( C_{m_1,n_1} \leftarrow A_{k_1,m_1,k_2} B_{k_2,n_1,k_1} \)

\[
\text{for } ( k_2 = 0; k_2 < K_2; k_2++ ) \\
gemm ( M_1, N_1, K_1, A[::, k_2]^T, B[k_2::]^T, C[::] )
\]
Loop over GEMM (LoG)

Conceptual Idea

Identify 2D subtensors and contract them via GEMM

\[
C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1}
\]
\[
C_{(m_1,m_2),n_1} \leftarrow A_{(m_1,m_2),k_1} B_{k_1,n_1}
\]
\[
C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_2,n_1}
\]
\[
C_{m_1,n_1} \leftarrow A_{k_1,m_1,k_2} B_{k_2,n_1,k_1}
\]

```
for ( n = 0; n < N_1; n_1++ )
    for ( k_2 = 0; k_2 < K_2; k_2++ )
        gemv ( M_1, K_1, A[:,:,:], B[k_2,n_1,:], C[:], n )
```
Loop Over GEMM (LoG)

- Search space:

  - GEMM indices: $m$, $n$, $k$

  - Loop order

  - Advantages:
    - Easy to implement
    - Exploits existing BLAS libraries
    - No additional memory required

  - Disadvantages:
    - Some contractions cannot be implemented via straight LoG
    - GEMM's arithmetic intensity can be suboptimal
Loop Over GEMM (LoG)

- Search space:
  - GEMM indices: $m, n, k$

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
- No additional memory required

Disadvantages:
- Some contractions cannot be implemented via straight LoG
- GEMM's arithmetic intensity can be suboptimal

Paul Springer (AICES)

High-Performance Tensor Contractions

Feb. 24th 2017
Loop Over GEMM (LoG)

- Search space:
  - GEMM indices: $m, n, k$
  - Loop order

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
- No additional memory required

Disadvantages:
- Some contractions cannot be implemented via straight LoG
- GEMM's arithmetic intensity can be suboptimal
Search space:
- GEMM indices: $m, n, k$
- Loop order
- Batched index

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
- No additional memory required

Disadvantages:
- Some contractions cannot be implemented via straight LoG
- GEMM's arithmetic intensity can be suboptimal
Loop Over GEMM (LoG)

- Search space:
  - GEMM indices: $m, n, k$
  - Loop order
  - Batched index

- Advantages:
Loop Over GEMM (LoG)

- Search space:
  - GEMM indices: $m$, $n$, $k$
  - Loop order
  - Batched index

- Advantages:
  - Easy to implement

Advantages:
- Exploits existing BLAS libraries
- No additional memory required
Loop Over GEMM (LoG)

- Search space:
  - GEMM indices: $m, n, k$
  - Loop order
  - Batched index

- Advantages:
  - Easy to implement
  - Exploits existing BLAS libraries
Search space:
- GEMM indices: \(m, n, k\)
- Loop order
- Batched index

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
- No additional memory required
Loop Over GEMM (LoG)

- Search space:
  - GEMM indices: $m, n, k$
  - Loop order
  - Batched index

- Advantages:
  - Easy to implement
  - Exploits existing BLAS libraries
  - No additional memory required

- Disadvantages:
Loop Over GEMM (LoG)

- **Search space:**
  - GEMM indices: $m$, $n$, $k$
  - Loop order
  - Batched index

- **Advantages:**
  - Easy to implement
  - Exploits existing BLAS libraries
  - No additional memory required

- **Disadvantages:**
  - Some contractions cannot be implemented via straight LoG
Loop Over GEMM (LoG)

- **Search space:**
  - GEMM indices: $m, n, k$
  - Loop order
  - Batched index

- **Advantages:**
  - Easy to implement
  - Exploits existing BLAS libraries
  - No additional memory required

- **Disadvantages:**
  - Some contractions cannot be implemented via straight LoG
  - GEMM’s arithmetic intensity can be suboptimal
Map Tensor Contractions to BLAS

- Free indices of $A$
  - $l_m := \{m_1, m_2, \ldots, m_\gamma\} = l_A \cap l_C$

---

Map Tensor Contractions to BLAS

- Free indices of $A$
  - $l_m := \{m_1, m_2, \ldots, m_\gamma\} = l_A \cap l_C$

- Free indices of $B$
  - $l_n := \{n_1, n_2, \ldots, n_\zeta\} = l_B \cap l_C$

---

Map Tensor Contractions to BLAS

- Free indices of $A$
  - $l_m := \{m_1, m_2, ..., m_\gamma\} = l_A \cap l_C$

- Free indices of $B$
  - $l_n := \{n_1, n_2, ..., n_\zeta\} = l_B \cap l_C$

- Contracted indices
  - $l_k := \{k_1, k_2, ..., k_\xi\} = l_A \cap l_B$

---

3 Di Napoli et al. “Towards an Efficient Use of the BLAS Library for Multilinear Tensor Contractions”

4 Yang Shi et al. “Tensor Contractions with Extended BLAS Kernels on CPU and GPU”
Map Tensor Contractions to BLAS

- **Free indices of** $A$
  - $l_m := \{m_1, m_2, ..., m_\gamma\} = l_A \cap l_C$

- **Free indices of** $B$
  - $l_n := \{n_1, n_2, ..., n_\zeta\} = l_B \cap l_C$

- **Contracted indices**
  - $l_k := \{k_1, k_2, ..., k_\xi\} = l_A \cap l_B$

- **Tensor contractions can be mapped to BLAS routines**$^3,^4$:
  - **GEMM:** $l_m \neq \emptyset$ and $l_n \neq \emptyset$ and $l_k \neq \emptyset$.

---

$^3$Di Napoli et al. “Towards an Efficient Use of the BLAS Library for Multilinear Tensor Contractions”

$^4$Yang Shi et al. “Tensor Contractions with Extended BLAS Kernels on CPU and GPU”
Map Tensor Contractions to BLAS

- Free indices of $A$
  - $l_m := \{m_1, m_2, ..., m_\gamma\} = l_A \cap l_C$

- Free indices of $B$
  - $l_n := \{n_1, n_2, ..., n_\zeta\} = l_B \cap l_C$

- Contracted indices
  - $l_k := \{k_1, k_2, ..., k_\xi\} = l_A \cap l_B$

- Tensor contractions can be mapped to BLAS routines\(^3\),\(^4\):
  - **GEMM**: $l_m \neq \emptyset$ and $l_n \neq \emptyset$ and $l_k \neq \emptyset$.
  - **GEMV**: $(l_m = \emptyset$ or $l_n = \emptyset$) and $l_k \neq \emptyset$

---

\(^3\) Di Napoli et al. “Towards an Efficient Use of the BLAS Library for Multilinear Tensor Contractions”

\(^4\) Yang Shi et al. “Tensor Contractions with Extended BLAS Kernels on CPU and GPU”
Map Tensor Contractions to BLAS

- Free indices of $A$
  - $l_m := \{m_1, m_2, ..., m_\gamma\} = l_A \cap l_C$

- Free indices of $B$
  - $l_n := \{n_1, n_2, ..., n_\zeta\} = l_B \cap l_C$

- Contracted indices
  - $l_k := \{k_1, k_2, ..., k_\xi\} = l_A \cap l_B$

Tensor contractions can be mapped to BLAS routines\textsuperscript{3,4}:

- **GEMM**: $l_m \neq \emptyset$ and $l_n \neq \emptyset$ and $l_k \neq \emptyset$.
- **GEMV**: $(l_m = \emptyset$ or $l_n = \emptyset)$ and $l_k \neq \emptyset$
- **GER**: $l_m \neq \emptyset$ and $l_n \neq \emptyset$ and $l_k = \emptyset$

\textsuperscript{3}Di Napoli et al. “Towards an Efficient Use of the BLAS Library for Multilinear Tensor Contractions”

\textsuperscript{4}Yang Shi et al. “Tensor Contractions with Extended BLAS Kernels on CPU and GPU”
Free indices of $A$
- $I_m := \{m_1, m_2, ..., m_\gamma\} = I_A \cap I_C$

Free indices of $B$
- $I_n := \{n_1, n_2, ..., n_\zeta\} = I_B \cap I_C$

Contracted indices
- $I_k := \{k_1, k_2, ..., k_\xi\} = I_A \cap I_B$

Tensor contractions can be mapped to BLAS routines\(^3\),\(^4\):
- **GEMM**: $I_m \neq \emptyset$ and $I_n \neq \emptyset$ and $I_k \neq \emptyset$.
- **GEMV**: ($I_m = \emptyset$ or $I_n = \emptyset$) and $I_k \neq \emptyset$
- **GER**: $I_m \neq \emptyset$ and $I_n \neq \emptyset$ and $I_k = \emptyset$
- **AXPY**: ($I_m = \emptyset$ or $I_n = \emptyset$) and $I_k = \emptyset$

\(^3\)Di Napoli et al. “Towards an Efficient Use of the BLAS Library for Multilinear Tensor Contractions”

\(^4\)Yang Shi et al. “Tensor Contractions with Extended BLAS Kernels on CPU and GPU”
Map Tensor Contractions to BLAS

- Free indices of $A$
  - $l_m := \{m_1, m_2, ..., m_\gamma\} = l_A \cap l_C$

- Free indices of $B$
  - $l_n := \{n_1, n_2, ..., n_\zeta\} = l_B \cap l_C$

- Contracted indices
  - $l_k := \{k_1, k_2, ..., k_\xi\} = l_A \cap l_B$

- Tensor contractions can be mapped to BLAS routines$^3,^4$:
  - GEMM: $l_m \neq \emptyset$ and $l_n \neq \emptyset$ and $l_k \neq \emptyset$
  - GEMV: $(l_m = \emptyset$ or $l_n = \emptyset$) and $l_k \neq \emptyset$
  - GER: $l_m \neq \emptyset$ and $l_n \neq \emptyset$ and $l_k = \emptyset$
  - AXPY: $(l_m = \emptyset$ or $l_n = \emptyset$) and $l_k = \emptyset$
  - DOT: else.

---

$^3$Di Napoli et al. “Towards an Efficient Use of the BLAS Library for Multilinear Tensor Contractions”

$^4$Yang Shi et al. “Tensor Contractions with Extended BLAS Kernels on CPU and GPU”
Conceptual Idea

1. “Flatten” the tensors to matrices
2. Use GEMM for contraction
3. “Unflatten” output matrix to tensor
Conceptual Idea

1. “Flatten” the tensors to matrices
2. Use GEMM for contraction
3. “Unflatten” output matrix to tensor

\[ C_{m_1,n_1} \leftarrow \mathbf{A}_{k_1,m_1,k_2} \mathbf{B}_{k_2,n_1,k_1} \]
Conceptual Idea

1. “Flatten” the tensors to matrices
2. Use GEMM for contraction
3. “Unflatten” output matrix to tensor

\[ C_{m_1,n_1} \leftarrow A_{k_1,m_1,k_2} B_{k_2,n_1,k_1} \]

- \( \tilde{A}_{m_1,(k_1,k_2)} \leftarrow A_{k_1,m_1,k_2} \)
- \( \tilde{B}_{(k_1,k_2),n_1} \leftarrow B_{k_2,n_1,k_1} \)
- \( \text{gemm}(M_1, N_1, K_1 \times K_2, \tilde{A}, \tilde{B}, C) \)
Conceptual Idea

1. "Flatten" the tensors to matrices
2. Use GEMM for contraction
3. "Unflatten" output matrix to tensor

\[ C_{m_1,n_1} \leftarrow A_{k_1,m_1,k_2} B_{k_2,n_1,k_1} \]

\[ \tilde{A}_{m_1,(k_1,k_2)} \leftarrow A_{k_1,m_1,k_2} \]
\[ \tilde{B}_{(k_1,k_2),n_1} \leftarrow B_{k_2,n_1,k_1} \]
\[ \text{gemm}(M_1, N_1, K_1 \times K_2, \tilde{A}, \tilde{B}, C) \]

\[ \tilde{A}_{(k_1,k_2),m_1} \leftarrow A_{k_1,m_1,k_2} \]
\[ \tilde{B}_{(k_1,k_2),n_1} \leftarrow B_{k_2,n_1,k_1} \]
\[ \text{gemm}(M_1, N_1, K_1 \times K_2, \tilde{A}^T, \tilde{B}, C) \]
**Conceptual Idea**

1. "Flatten" the tensors to matrices
2. Use GEMM for contraction
3. "Unflatten" output matrix to tensor

\[ C_{m_1,n_1} \leftarrow A_{k_1,m_1,k_2} B_{k_2,n_1,k_1} \]

\[
\begin{align*}
\tilde{A}_{m_1,(k_1,k_2)} & \leftarrow A_{k_1,m_1,k_2} \\
\tilde{B}_{(k_1,k_2),n_1} & \leftarrow B_{k_2,n_1,k_1} \\
gemm(M_1, N_1, K_1 \times K_2, \tilde{A}, \tilde{B}, \tilde{C})
\end{align*}
\]

\[
\begin{align*}
\tilde{A}_{(k_1,k_2),m_1} & \leftarrow A_{k_1,m_1,k_2} \\
\tilde{B}_{(k_1,k_2),n_1} & \leftarrow B_{k_2,n_1,k_1} \\
gemm(M_1, N_1, K_1 \times K_2, \tilde{A}^T, \tilde{B}, \tilde{C})
\end{align*}
\]

\[
\begin{align*}
\tilde{A}_{(k_1,k_2),m_1} & \leftarrow A_{k_1,m_1,k_2} \\
\tilde{B}_{(k_1,k_2),n_1} & \leftarrow B_{k_2,n_1,k_1} \\
gemm(M_1, N_1, K_1 \times K_2, \tilde{B}^T, \tilde{A}, \tilde{C})
\end{align*}
\]

\[
C_{m_1,n_1} \leftarrow \tilde{C}_{n_1,m_1}
\]
Conceptual Idea

1. “Flatten” the tensors to matrices
2. Use GEMM for contraction
3. “Unflatten” output matrix to tensor

\[ C_{m_1, n_1} \leftarrow A_{k_1, m_1, k_2} B_{k_2, n_1, k_1} \]

\[ \tilde{A}_{m_1, (k_1, k_2)} \leftarrow A_{k_1, m_1, k_2} \]
\[ \tilde{B}_{(k_1, k_2), n_1} \leftarrow B_{k_2, n_1, k_1} \]
\[ \text{gemm}(M_1, N_1, K_1 \times K_2, \tilde{A}, \tilde{B}, C) \]

\[ \tilde{A}_{(k_1, k_2), m_1} \leftarrow A_{k_1, m_1, k_2} \]
\[ \tilde{B}_{(k_1, k_2), n_1} \leftarrow B_{k_2, n_1, k_1} \]
\[ \text{gemm}(M_1, N_1, K_1 \times K_2, \tilde{A}^T, \tilde{B}, C) \]

\[ \tilde{A}_{(k_2, k_1), m_1} \leftarrow A_{k_1, m_1, k_2} \]
\[ \tilde{B}_{(k_2, k_1), n_1} \leftarrow B_{k_2, n_1, k_1} \]
\[ \text{gemm}(M_1, N_1, K_1 \times K_2, \tilde{B}^T, \tilde{A}, \tilde{C}) \]
\[ C_{m_1, n_1} \leftarrow \tilde{C}_{n_1, m_1} \]
Conceptual Idea

1. “Flatten” the tensors to matrices
2. Use GEMM for contraction
3. “Unflatten” output matrix to tensor

\[ C_{m_1, n_1} \leftarrow A_{k_1, m_1, k_2} B_{k_2, n_1, k_1} \]

\[ \tilde{A}_{m_1, (k_1, k_2)} \leftarrow A_{k_1, m_1, k_2} \]
\[ \tilde{B}_{(k_1, k_2), n_1} \leftarrow B_{k_2, n_1, k_1} \]
\[ \text{gemm}(M_1, N_1, K_1 \times K_2, \tilde{A}, \tilde{B}, C) \]

\[ \tilde{A}_{(k_1, k_2), m_1} \leftarrow A_{k_1, m_1, k_2} \]
\[ \tilde{B}_{(k_1, k_2), n_1} \leftarrow B_{k_2, n_1, k_1} \]
\[ \text{gemm}(M_1, N_1, K_1 \times K_2, \tilde{A}^T, \tilde{B}, C) \]

... and more.
Transpose-Transpose-GEMM-Transpose (TTGT)

- Search space:

---

Search space:
- Any permutation of $l_m, l_n, l_k$
Search space:
- Any permutation of $l_m, l_n, l_k$
- Transposed $\mathcal{A}$

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
- All TCs can be implemented via TTGT
- Large GEMM $\Rightarrow$ good performance?

Disadvantages:
- Transpositions account for pure overhead
- Additional memory required

---

Search space:
- Any permutation of $I_m, I_n, I_k$
- Transposed $A$
- Transposed $B$
Search space:
- Any permutation of $l_m, l_n, l_k$
- Transposed $A$
- Transposed $B$
- Interchange $A$ and $B$ within GEMM
Search space:
- Any permutation of $I_m, I_n, I_k$
- Transposed $A$
- Transposed $B$
- Interchange $A$ and $B$ within GEMM

Advantages:

---

Search space:
- Any permutation of $I_m, I_n, I_k$
- Transposed $A$
- Transposed $B$
- Interchange $A$ and $B$ within GEMM

Advantages:
- Easy to implement

---

Search space:
- Any permutation of $I_m, I_n, I_k$
- Transposed $A$
- Transposed $B$
- Interchange $A$ and $B$ within GEMM

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
Search space:
- Any permutation of $l_m, l_n, l_k$
- Transposed $A$
- Transposed $B$
- Interchange $A$ and $B$ within GEMM

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
- All TCs can be implemented via TTGT

---

Search space:
- Any permutation of $I_m, I_n, I_k$
- Transposed $A$
- Transposed $B$
- Interchange $A$ and $B$ within GEMM

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
- All TCs can be implemented via TTGT
- Large GEMM $\Rightarrow$ good performance?
Search space:
- Any permutation of $I_m, I_n, I_k$
- Transposed $A$
- Transposed $B$
- Interchange $A$ and $B$ within GEMM

Advantages:
- Easy to implement
- Exploits existing BLAS libraries
- All TCs can be implemented via TTGT
- Large GEMM $\Rightarrow$ good performance?

Disadvantages:
Transpose-Transpose-GEMM-Transpose (TTGT)

- Search space:
  - Any permutation of $l_m, l_n, l_k$
  - Transposed $A$
  - Transposed $B$
  - Interchange $A$ and $B$ within GEMM

- Advantages:
  - Easy to implement
  - Exploits existing BLAS libraries
  - All TCs can be implemented via TTGT
  - Large GEMM $\Rightarrow$ good performance?

- Disadvantages:
  - Transpositions$^5$ account for pure overhead

---

Transpose-Transpose-GEMM-Transpose (TTGT)

- Search space:
  - Any permutation of $l_m, l_n, l_k$
  - Transposed $A$
  - Transposed $B$
  - Interchange $A$ and $B$ within GEMM

- Advantages:
  - Easy to implement
  - Exploits existing BLAS libraries
  - All TCs can be implemented via TTGT
  - Large GEMM ⇒ good performance?

- Disadvantages:
  - Transpositions$^5$ account for pure overhead
  - Additional memory required

---

Key Idea

- Eliminate explicit transpositions
- Pack-and-transpose while moving data into the caches\(^5\)
  \[\Rightarrow\] Complexity offloaded into packing routines

**GEMM-like Tensor-Tensor Multiplication (GETT)**

**Key Idea**

- Eliminate explicit transpositions
- Pack-and-transpose while moving data into the caches\(^5\)

⇒ Complexity offloaded into packing routines

```plaintext
// N-Loop
for n = 1 : nc : S_{ln}
  // K-Loop (contracted)
  for k = 1 : kc : S_{lk}
    \( \hat{B} = \text{identify_subtensor}(B, n, k) \)
    // pack \( \hat{B} \) into \( \tilde{B} \) (L3 cache)
    \( \tilde{B} = \text{packB}(\hat{B}) \)
  // M-Loop
  for m = 1 : mc : S_{lm}
    \( \hat{A} = \text{identify_subtensor}(A, m, k) \)
    // pack \( \hat{A} \) into \( \tilde{A} \) (L2 cache)
    \( \tilde{A} = \text{packA}(\hat{A}) \)
    \( \hat{C} = \text{identify_subtensor}(C, m, n) \)
    // compute matrix-matrix product of \( \tilde{A}\tilde{B} \)
    \( \text{macroKernel}(\tilde{A}, \tilde{B}, \hat{C}, \alpha, \beta) \)
```

GEMM-like Tensor-Tensor Multiplication (GETT)

Key Idea

- Eliminate explicit transpositions
- Pack-and-transpose while moving data into the caches\(^5\)

\[ \Rightarrow \text{Complexity offloaded into packing routines} \]

```
1 // N-Loop
2 for n = 1 : nc : S_n
3 // pack \( \hat{B} \) into \( \tilde{B} \) (L3 cache)
4 \( \tilde{B} = \text{packB}(\hat{B}) \)
5 // M-Loop
6 for m = 1 : mc : S_m
7 \( \hat{A} = \text{identify_subtensor}(A, m, k) \)
8 // pack \( \hat{A} \) into \( \tilde{A} \) (L2 cache)
9 \( \tilde{A} = \text{packA}(\hat{A}) \)
10 \( \hat{C} = \text{identify_subtensor}(C, m, n) \)
11 // compute matrix-matrix product of \( \tilde{A}\tilde{B} \)
12 \( \text{macroKernel}(\tilde{A}, \tilde{B}, \hat{C}, \alpha, \beta) \)
```


Monday, February 27 10:15 - 10:35

TCCG: Tensor Contraction Code Generator
GEMM-like Tensor-Tensor Multiplication (GETT)

- Search space:

  - Blocking parameters: $mc$, $nc$, $kc$

  - Subtensors $\hat{A}$, $\hat{B}$, $\hat{C}$

Advantages:
- Same arithmetic intensity as GEMM
- No memory overhead

Disadvantages:
- Complex to implement
GEMM-like Tensor-Tensor Multiplication (GETT)

- Search space:
  - Blocking parameters: \( mc, nc, kc \)

Advantages:
- Same arithmetic intensity as GEMM
- No memory overhead

Disadvantages:
- Complex to implement
Search space:
- Blocking parameters: $mc$, $nc$, $kc$
- Subtensors $\hat{A}$, $\hat{B}$, $\hat{C}$
GEMM-like Tensor-Tensor Multiplication (GETT)

- Search space:
  - Blocking parameters: $mc$, $nc$, $kc$
  - Subtensors $\hat{A}$, $\hat{B}$, $\hat{C}$

- Advantages:
GEMM-like Tensor-Tensor Multiplication (GETT)

- Search space:
  - Blocking parameters: $mc, nc, kc$
  - Subtensors $\hat{A}, \hat{B}, \hat{C}$

- Advantages:
  - Same arithmetic intensity as GEMM
GEMM-like Tensor-Tensor Multiplication (GETT)

- **Search space:**
  - Blocking parameters: $mc, nc, kc$
  - Subtensors $\hat{A}, \hat{B}, \hat{C}$

- **Advantages:**
  - Same arithmetic intensity as GEMM
  - No memory overhead
GEMM-like Tensor-Tensor Multiplication (GETT)

- **Search space:**
  - Blocking parameters: $mc$, $nc$, $kc$
  - Subtensors $\hat{A}$, $\hat{B}$, $\hat{C}$

- **Advantages:**
  - Same arithmetic intensity as GEMM
  - No memory overhead

- **Disadvantages:**
GEMM-like Tensor-Tensor Multiplication (GETT)

- Search space:
  - Blocking parameters: $mc, nc, kc$
  - Subtensors $\hat{A}, \hat{B}, \hat{C}$

- Advantages:
  - Same arithmetic intensity as GEMM
  - No memory overhead

- Disadvantages:
  - Complex to implement
Tensor Contraction Code Generator (TCCG)

- **Input**: Mathematical description of TC
  - e.g., $C[a,b,i,j] = A[i,k,a] \ast B[k,j,b]$;
- **Output**: High-Performance C++ code
Tensor Contraction Code Generator (TCCG)

- **Input**: Mathematical description of TC
  - e.g., $C[a,b,i,j] = A[i,k,a] \times B[k,j,b]$;
- **Output**: High-Performance C++ code

---

**Figure**: Schematic overview of TCCG.
Performance — Haswell (single core)

Not all TCs can be implemented via LoG

Mixed performance

Paul Springer (AICES)
High-Performance Tensor Contractions
Feb. 24th 2017
Not all TCs can be implemented via LoG
• Not all TCs can be implemented via LoG
• Mixed performance
Performance — Haswell (single core)

**Diagram Description:**
- **GFLOPS** is plotted on the y-axis.
- **TTGT** and **LoG** are compared on the x-axis.
- TTGT is good for compute-bound TCs and bad for bandwidth-bound TCs.

**Legend:**
- Orange bars represent **LoG**.
- Red bars represent **TTGT**.

**Data Points:**
- Various sequences of characters (e.g., `abcdef`, `ab-cd-ef`, etc.) represent different tensor contraction types.

**Notes:**
- Paul Springer (AICES)
- High-Performance Tensor Contractions
- Feb. 24th 2017
- Slide 13/17
**TTGT:** good for compute-bound TCs
• TTGT: good for compute-bound TCs
• TTGT: bad for bandwidth-bound TCs
Performance — Haswell (single core)

GETT: excels for bandwidth-bound TCs
GETT: good for compute-bound TCs

Paul Springer (AICES)
● GETT: excels for bandwidth-bound TCs

Paul Springer (AICES)
GETT: excels for bandwidth-bound TCs
GETT: good for compute-bound TCs
(a) 2×Intel Xeon E5-2680 v3

- Performance gap increases for bandwidth-bound TCs
Performance — Multi-threaded

- Performance gap increases for bandwidth-bound TCs

(a) 2×Intel Xeon E5-2680 v3
(b) NVIDIA Tesla P100
Performance for equally-sized GEMMs varies greatly for different settings:

- opA, opB, interchanged

**Performance Model for TTGT and LoG:**

Account for varying GEMM perf

---


---

(a) 2×Intel Xeon E5-2680 v3

(b) NVIDIA Tesla P100
Performance for equally-sized GEMMs varies greatly
- For different settings: opA, opB, interchanged A and B

---


Paul Springer (AICES) High-Performance Tensor Contractions Feb. 24th 2017 16 / 17
GEMM Performance — Multi-threaded

Performance for equally-sized GEMMs varies greatly
- For different settings: opA, opB, interchanged A and B
- Performance Model for TTGT and LoG:
  - Account for varying GEMM perf

---


(a) $2 \times$ Intel Xeon E5-2680 v3

(b) NVIDIA Tesla P100
Conclusion

- A survey of different approaches to TCs has been presented
- GETT exhibits high performance across a wide range of TCs
- TCCG is available at https://github.com/HPAC/tccg
Conclusion

- A survey of different approaches to TCs has been presented
- GETT exhibits high performance across a wide range of TCs
- TCCG is available at https://github.com/HPAC/tccg

Future Work

- Implement TC library based on GETT
- Parallelize GETT
Conclusion

- A survey of different approaches to TCs has been presented
- GETT exhibits high performance across a wide range of TCs
- TCCG is available at https://github.com/HPAC/tccg

Future Work

- Implement TC library based on GETT
- Parallelize GETT

Thank you for your attention.
Systems:
- Intel Xeon E5-2680 v3 CPU (Haswell)
- NVIDIA Tesla P100 GPU (Pascal)

Compilers:
- icpc 16.0.1 20151021
- nvcc v8.0.44

Benchmark
- Collection of 48 TCs
- Compiled from four publications
- Each TC is at least 200 MiB

Correctness checked against naive loop-based implementation
- TTGT faster than CTF everywhere.
- TTGT good in compute-bound regime
- TTGT bad in bandwidth-bound regime
Performance: $m_1 n_1 m_2 - m_1 k_1 m_2 - n_1 k_1$

- GETT especially good in bandwidth-bound regime
  - GETT still attains up to 91.3% of peak floating-point performance
- TTGT poor in bandwidth-bound regime
GETT especially good in bandwidth-bound regime
  - GETT still attains up to 91.3% of peak floating-point performance

TTGT poor in bandwidth-bound regime

LoG performance can become arbitrarily bad

GETT and TTGT barely affected by higher dimensions
Speedup

(a) Single-Precision.

(b) Double-Precision.
\[ \hat{C}_{m_1,n_1,m_2} = \hat{A}_{m_1,m_2,k_1} \times \hat{B}_{k_1,n_1} \]
GETT: Macro- /Micro-Kernel

- Blocking for L3, L2, L1 cache as well as registers
Blocking for L3, L2, L1 cache as well as registers

Written in AVX2 intrinsics
Packing via Tensor Transpositions

\[ \tilde{A}_{m_1,k,m_2}, k \]

\[ \tilde{A}(m_1,m_2), k \]
Packing via Tensor Transpositions

Preserve stride-1 index

⇒ Efficient packing routines

GETT: Summary

- Blocking for caches
- Blocking for registers
- Explicitly vectorized
- Use TTC to generate high-performance packing routines
  - Exploits full cache line (avoids non-stride-one memory accesses)
- Explore large search-space:
  - Different GEMM-variants (e.g., panel-matrix, matrix-panel)
  - Different permutations
  - Different values for $mc$, $nc$ and $kc$
- Prune the search space via a performance model
TTGT good in compute-bound regime
TTGT bad in bandwidth-bound regime
TTGT faster than CTF everywhere.
- TTGT good in compute-bound regime
- TTGT bad in bandwidth-bound regime
- TTGT faster than CTF everywhere.
Figure: Limit the GETT candidates to 1, 4, 8, 16 or 32, respectively.

- Average performance without search: 90.7% / 92.3%
- Average performance of the four best candidates: 98.3% / 97.2%