Design of a High-Performance GEMM-like Tensor-Tensor Multiplication

Paul Springer and Paolo Bientinesi

Aachen Institute for Advanced Study in Computational Engineering Science

Austin, Sep. 20th 2016
1. Introduction

2. GEMM-like Tensor-Tensor Multiplication

3. Tensor Contraction Code Generator

4. Performance

5. Conclusion and Future Work
Introduction

- Tensors can be thought of as higher dimensional matrices
- Tensor contraction can be thought of as higher dimensional GEMMs

---

Introduction

- Tensors can be thought of as higher dimensional matrices
- Tensor contraction can be thought of as higher dimensional GEMMs
- Essentially three approaches:
  - Nested loops
  - Transpose-Transpose-GEMM-Transpose (TTGT)
  - Loops over GEMM (LoG)

Introduction

- Tensors can be thought of as higher dimensional matrices
- Tensor contraction can be thought of as higher dimensional GEMMs
- Essentially three approaches:
  - Nested loops
  - Transpose-Transpose-GEMM-Transpose (TTGT)
  - Loops over GEMM (LoG)
- We propose a novel approach: GETT\(^1\)
  - Akin to a high-performance GEMM implementation
  - Adopts the BLIS methodology: **Breaking through the BLAS layer**

---

Tensors can be thought of as higher dimensional matrices.

Tensor contraction can be thought of as higher dimensional GEMMs.

Essentially three approaches:
- Nested loops
- Transpose-Transpose-GEMM-Transpose (TTGT)
- Loops over GEMM (LoG)

We propose a novel approach: GETT\(^1\)
- Akin to a high-performance GEMM implementation
- Adopts the BLIS methodology: **Breaking through the BLAS layer**

Tensor Contraction Code Generator (TCCG)
- combine GETT, TTGT and LoG into a unified tool

---


Paul Springer (AICES)
Matrix-Matrix Multiplication

Let $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$ and $C \in \mathbb{R}^{M \times N}$ be 2D tensors:

$$C_{m,n} \leftarrow \sum_k A_{m,k} B_{k,n}$$
Matrix-Matrix Multiplication (Einstein notation)

\[ A \in \mathbb{R}^{M \times K}, \quad B \in \mathbb{R}^{K \times N} \quad \text{and} \quad C \in \mathbb{R}^{M \times N} \quad \text{be 2D tensors:} \]
\[ C_{m,n} \leftarrow A_{m,k} B_{k,n} \]
Matrix-Matrix Multiplication

Matrix-Matrix Multiplication (Einstein notation)

\[ A \in \mathbb{R}^{M \times K}, \quad B \in \mathbb{R}^{K \times N} \quad \text{and} \quad C \in \mathbb{R}^{M \times N} \quad \text{be 2D tensors:} \]

\[ C_{m,n} \leftarrow A_{m,k} B_{k,n} \]

Naive GEMM.

```c
// N-Loop
for j = 0 : N - 1
  // M-Loop
  for i = 0 : M - 1
    tmp = 0
    // K-Loop (contracted)
    for k = 0 : K - 1
      tmp += A_{i,k} B_{k,j}
    // update C
    C_{i,j} = \alpha \cdot \text{tmp} + \beta C_{i,j}
```

Paul Springer (AICES)  Tensor Contraction Code Generator  Sep. 20th 2016 4 / 19
Matrix-Matrix Multiplication (Einstein notation)

\[ A \in \mathbb{R}^{M \times K}, \ B \in \mathbb{R}^{K \times N} \text{ and } C \in \mathbb{R}^{M \times N} \] 
be 2D tensors:
\[
C_{m,n} \leftarrow A_{m,k}B_{k,n}
\]

Naive GEMM.

High-performance GEMM.

\[
// \text{ N-Loop} \\
\text{for } n = 0 : nc : N - 1 \\
// \text{ K-Loop (contracted)} \\
\text{for } k = 0 : kc : K - 1 \\
\hat{B} = \text{identify_submatrix}(B, n, k) \\
// \text{ pack } \hat{B} \text{ into } \hat{B} \\
\hat{B} = \text{packB}(\hat{B}) \ // \hat{B} \in \mathbb{R}^{kc \times nc} \\
// \text{ M-Loop} \\
\text{for } m = 0 : mc : M - 1 \\
\hat{A} = \text{identify_submatrix}(A, m, k) \\
// \text{ pack } \hat{A} \text{ into } \hat{A} \\
\hat{A} = \text{packA}(\hat{A}) \ // \hat{A} \in \mathbb{R}^{mc \times kc} \\
\hat{C} = \text{identify_submatrix}(C, m, n) \\
// \text{ matrix-matrix product: } \hat{A}\hat{B} \\
\text{macroKernel}(\hat{A}, \hat{B}, \hat{C}, \alpha, \beta)
\]
Tensor contraction examples:

\[ C_{m,n} \leftarrow A_{m,k} B_{k,n} \]
Tensor contraction examples:
- $C_{m,n} \leftarrow A_{m,k} B_{k,n}$
- $C_{m_1,m_2,n} \leftarrow A_{m_1,m_2,k} B_{k,n}$
Tensor contraction examples:

- $C_{m,n} \leftarrow A_{m,k} B_{k,n}$
- $C_{m_1,m_2,n} \leftarrow A_{m_1,m_2,k} B_{k,n}$
- $C_{m_1,n,m_2} \leftarrow A_{m_1,m_2,k} B_{k,n}$
Tensor Contraction Code Generator

Tensor contraction examples:

- \( C_{m,n} \leftarrow A_{m,k} B_{k,n} \)
- \( C_{m_1,m_2,n} \leftarrow A_{m_1,m_2,k} B_{k,n} \)
- \( C_{m_1,n,m_2} \leftarrow A_{m_1,m_2,k} B_{k,n} \)
- \( C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k} B_{n_2,k,n_1} \)
Tensor Contractions

- Tensor contraction examples:
  - $C_{m,n} \leftarrow A_{m,k} B_{k,n}$
  - $C_{m_1,m_2,n} \leftarrow A_{m_1,m_2,k} B_{k,n}$
  - $C_{m_1,n,m_2} \leftarrow A_{m_1,m_2,k} B_{k,n}$
  - $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k} B_{n_2,k,n_1}$
  - $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,k_1,m_2,k_2} B_{k_2,n_2,k_1,n_1}$
Tensor contraction examples:

- $C_{m,n} \leftarrow A_{m,k} B_{k,n}$
- $C_{m_1,m_2,n} \leftarrow A_{m_1,m_2,k} B_{k,n}$
- $C_{m_1,n,m_2} \leftarrow A_{m_1,m_2,k} B_{k,n}$
- $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k} B_{n_2,k,n_1}$
- $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,k_1,m_2,k_2} B_{k_2,n_2,k_1,n_1}$
- $C_{m_1,n_1,n_2,m_2,n_3} \leftarrow A_{m_1,k_1,m_2,k_2} B_{n_3,k_2,n_2,k_1,n_1}$
- ...
Tensor Contraction examples:

- $C_{m,n} \leftarrow A_{m,k} B_{k,n}$
- $C_{m_1,m_2,n} \leftarrow A_{m_1,m_2,k} B_{k,n}$
- $C_{m_1,n,m_2} \leftarrow A_{m_1,m_2,k} B_{k,n}$
- $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,m_2,k} B_{n_2,k,n_1}$
- $C_{m_1,n_1,n_2,m_2} \leftarrow A_{m_1,k_1,m_2,k_2} B_{k_2,n_2,k_1,n_1}$
- $C_{m_1,n_1,n_2,m_2,n_3} \leftarrow A_{m_1,k_1,m_2,k_2} B_{n_3,k_2,n_2,k_1,n_1}$
- ...

$\Rightarrow$ Quite similar to GEMM.
Tensor-Tensor Multiplication (Einstein notation)

Let the input tensors $\mathcal{A} \in \mathbb{R}^{S_1^A \times S_2^A \times \ldots \times S_{r_A}^A}$, and $\mathcal{B} \in \mathbb{R}^{S_1^B \times S_2^B \times \ldots \times S_{r_B}^B}$ update the output tensor $\mathcal{C} \in \mathbb{R}^{S_1^C \times S_2^C \times \ldots \times S_{r_C}^C}$:

$$\mathcal{C}_{\Pi^C(I_m \cup I_n)} \leftarrow \alpha \mathcal{A}_{\Pi^A(I_m \cup I_k)} \mathcal{B}_{\Pi^B(I_n \cup I_k)} + \beta \mathcal{C}_{\Pi^C(I_m \cup I_n)}.$$
Tensor-Tensor Multiplication (Einstein notation)

Let the input tensors $\mathcal{A} \in \mathbb{R}^{S_1^A \times S_2^A \times ... \times S_r^A}$, and $\mathcal{B} \in \mathbb{R}^{S_1^B \times S_2^B \times ... \times S_r^B}$, update the output tensor $\mathcal{C} \in \mathbb{R}^{S_1^C \times S_2^C \times ... \times S_r^C}$:

$$\mathcal{C}_{\Pi^C(I_m \cup I_n)} \leftarrow \alpha \mathcal{A}_{\Pi^A(I_m \cup I_k)} \mathcal{B}_{\Pi^B(I_n \cup I_k)} + \beta \mathcal{C}_{\Pi^C(I_m \cup I_n)}.$$

- These index sets $I_m$, $I_n$ and $I_k$ are critical
  - $I_m := \{m_1, m_2, ..., m_\gamma\}$: free indices of $\mathcal{A}$
  - $I_n := \{n_1, n_2, ..., n_\zeta\}$: free indices of $\mathcal{B}$
  - $I_k := \{k_1, k_2, ..., k_\xi\}$: contracted indices
Naive GETT.
Naive GETT.

```plaintext
// N-Loops
for n_1 = 1 : S_{n_1}
    // ... remaining N-loops omitted ...
for n_\zeta = 1 : S_{n_\zeta}

// M-Loops
for m_1 = 1 : S_{m_1}
    // ... remaining M-loops omitted ...
for m_\gamma = 1 : S_{m_\gamma}
    tmp = 0
    // K-Loops (contracted)
for k_1 = 1 : S_{k_1}
    // ... remaining K-loops omitted ...
for k_\xi = 1 : S_{k_\xi}
    tmp += A_\Pi A(m_1, \ldots, m_\gamma, k_1, \ldots, k_\xi) B_\Pi B(k_1, \ldots, k_\xi, n_1, \ldots, n_\zeta)
    // update C
    C_\Pi C(m_1, \ldots, m_\gamma, n_1, \ldots, n_\zeta) = \alpha \cdot tmp + \beta C_\Pi C(m_1, \ldots, m_\gamma, n_1, \ldots, n_\zeta)
```
Naive GETT.
Naive GETT.
// N-Loop
for n = 1 : nc : S_{ln}
// K-Loop (contracted)
for k = 1 : kc : S_{lk}
\hat{B} = identify_subtensor(B, n, k)
// pack \hat{B} into \tilde{B}
\tilde{B} = packB(\hat{B})
// M-Loop
for m = 1 : mc : S_{lm}
\hat{A} = identify_subtensor(A, m, k)
// pack \hat{A} into \tilde{A}
\tilde{A} = packA(\hat{A})
\hat{C} = identify_subtensor(C, m, n)
// compute matrix-matrix product of \tilde{A}\tilde{B}
macroKernel(\tilde{A}, \tilde{B}, \hat{C}, \alpha, \beta)

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

High-performance GETT.

Key Idea

Pack-and-transpose while moving data into the caches
\[ C_{m_1,n_1,m_2} = A_{m_1,m_2,k_1} \times B_{k_1,n_1} \]
Blocking for L3, L2, L1 cache as well as registers
GETT: Macro- /Micro-Kernel

- Blocking for L3, L2, L1 cache as well as registers
- Written in AVX2 intrinsics
Packing via Tensor Transpositions

\[ \tilde{A}_{m_1,k,m_2} \to ? \to \tilde{A}_{(m_1,m_2),k} \]

Preserve stride-1 index

Efficient packing routines
Packing via Tensor Transpositions

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

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

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

- 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
Figure: Schematic overview of TCCG.
Performance

- **System**: Intel Xeon E5-2680 v3 CPU (Haswell)
  - Single core
  - Turbo Boost: disabled

- **Compiler**: `icpc 16.0.1 20151021`

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

- Correctness checked against naive loop-based implementation
TTGT good in compute-bound regime
TTGT bad in bandwidth-bound regime
TTGT faster than CTF everywhere.

Paul Springer (AICES)
Sep. 20th 2016 15 / 19
- TTGT good in compute-bound regime
- TTGT bad in bandwidth-bound regime
- TTGT faster than CTF everywhere.
GETT excels in bandwidth-bound regime.

GETT slightly lags behind in compute-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
Performance: $i_1 j_1 i_2 j_2 - i_1 k_1 j_2 - j_1 k_j$

- 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 varies between $1.0 \times$ and $12.4 \times$
Conclusion

- GETT: a systematic way to reduce an arbitrary TC to a GEMM-like macro-kernel
- GETT exhibits high performance across a wide range of TCs
  - It especially excels in the bandwidth-bound regime
  - It attains up to 91.3% of peak floating-point performance
- A survey of different approaches to TCs has been presented
- Give it a try: https://github.com/HPAC/tccg
## Conclusion

- **GETT**: a systematic way to reduce an arbitrary TC to a GEMM-like macro-kernel
- GETT exhibits high performance across a wide range of TCs
  - It especially excels in the bandwidth-bound regime
  - It attains up to $91.3\%$ of peak floating-point performance
- A survey of different approaches to TCs has been presented
- Give it a try: [https://github.com/HPAC/tccg](https://github.com/HPAC/tccg)

## Future Work

- Assess TCCG’s performance on KNL
- Add parallelism
- Turn TCCG into a C library?
Conclusion

- GETT: a systematic way to reduce an arbitrary TC to a GEMM-like macro-kernel
- GETT exhibits high performance across a wide range of TCs
  - It especially excels in the bandwidth-bound regime
- It attains up to 91.3% of peak floating-point performance

Future Work

- Assess TCCG’s performance on KNL
- Add parallelism
- Turn TCCG into a C library?

Thank you for your attention.

Give it a try: https://github.com/HPAC/tccg
GETT excels in bandwidth-bound regime.
GETT slightly lags behind in compute-bound regime.
GETT attains min/avg/max performance of GEMM:
- SP: 72.4% / 98.1% / 141.4%
- DP: 60.8% / 97.0% / 132.9%
TTGT faster than CTF everywhere.
TTGT good in compute-bound regime
TTGT bad in bandwidth-bound regime
• GETT excels in bandwidth-bound regime.
• GETT slightly lags behind in compute-bound regime.
• GETT attains min/avg/max performance of GEMM:
  - SP: 72.4% / 98.1% / 141.4%
  - DP: 60.8% / 97.0% / 132.9%
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
### Single-Precision

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcde-efbad-cf</td>
<td>11.7</td>
</tr>
<tr>
<td>abcde-efcad-bf</td>
<td>12.4</td>
</tr>
<tr>
<td>abcd-dbea-ec</td>
<td>4.1</td>
</tr>
<tr>
<td>abcde-ecbfa-fd</td>
<td>9.0</td>
</tr>
<tr>
<td>abcd-deca-be</td>
<td>6.9</td>
</tr>
<tr>
<td>abc-bda-dc</td>
<td>3.4</td>
</tr>
<tr>
<td>abcd-ebad-ce</td>
<td>3.3</td>
</tr>
<tr>
<td>abcde-dega-gfbc</td>
<td>2.3</td>
</tr>
<tr>
<td>abcdef-dfgb-gabc</td>
<td>2.9</td>
</tr>
<tr>
<td>abcd-degb-gfac</td>
<td>1.7</td>
</tr>
<tr>
<td>abcd-degc-gfab</td>
<td>1.2</td>
</tr>
<tr>
<td>abcde-debf-fdec</td>
<td>1.4</td>
</tr>
<tr>
<td>abcdef-degb-gfac</td>
<td>1.7</td>
</tr>
<tr>
<td>abcde-abab-acab</td>
<td>1.7</td>
</tr>
<tr>
<td>abcdef-degb-gfac</td>
<td>1.1</td>
</tr>
<tr>
<td>abcdef-degb-gfac</td>
<td>1.1</td>
</tr>
<tr>
<td>abcdef-degb-gfac</td>
<td>1.1</td>
</tr>
</tbody>
</table>

### Double-Precision

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcde-efbad-cf</td>
<td>11.7</td>
</tr>
<tr>
<td>abcde-efcad-bf</td>
<td>12.4</td>
</tr>
<tr>
<td>abcd-dbea-ec</td>
<td>4.1</td>
</tr>
<tr>
<td>abcde-ecbfa-fd</td>
<td>9.0</td>
</tr>
<tr>
<td>abcd-deca-be</td>
<td>6.9</td>
</tr>
<tr>
<td>abc-bda-dc</td>
<td>3.4</td>
</tr>
<tr>
<td>abcd-ebad-ce</td>
<td>3.3</td>
</tr>
<tr>
<td>abcde-dega-gfbc</td>
<td>2.3</td>
</tr>
<tr>
<td>abcdef-dfgb-gabc</td>
<td>2.9</td>
</tr>
<tr>
<td>abcd-degb-gfac</td>
<td>1.7</td>
</tr>
<tr>
<td>abcd-degc-gfab</td>
<td>1.2</td>
</tr>
<tr>
<td>abcde-debf-fdec</td>
<td>1.4</td>
</tr>
<tr>
<td>abcdef-degb-gfac</td>
<td>1.7</td>
</tr>
<tr>
<td>abcdef-degb-gfac</td>
<td>1.1</td>
</tr>
<tr>
<td>abcdef-degb-gfac</td>
<td>1.1</td>
</tr>
<tr>
<td>abcdef-degb-gfac</td>
<td>1.1</td>
</tr>
</tbody>
</table>

(a) Single-Precision.

(b) Double-Precision.
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%
Tensor Contraction Code Generator (TCCG)

\[ C[a,b,i,j] = A[i,m,a] \times B[m,j,b] \]
\[ a = 24 \]
\[ b = 24 \]
\[ i = 24 \]
\[ j = 24 \]
\[ m = 24 \]

**Figure:** Exemplary input file for TCCG.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--floatType=[s,d]</td>
<td>data type</td>
</tr>
<tr>
<td>--maxWorkspace=&lt;value&gt;</td>
<td>maximum auxiliary workspace in GB</td>
</tr>
<tr>
<td>--maxImplementations=&lt;value&gt;</td>
<td>maximum #implementations</td>
</tr>
<tr>
<td>--arch=[hsw,knl,cuda]</td>
<td>selected architecture</td>
</tr>
<tr>
<td>--numThreads=&lt;value&gt;</td>
<td>number of threads</td>
</tr>
</tbody>
</table>

**Table:** TCCG’s command line arguments.
TTGT pseudo code for a general tensor contraction

\[ C_{\Pi^c(l_m \cup l_n)} = A_{\Pi^A(l_m \cup l_k)} B_{\Pi^B(l_n \cup l_k)} + C_{\Pi^c(l_m \cup l_n)}. \]

- \( \Pi^m(l_m) \), \( \Pi^n(l_n) \) and \( \Pi^k(l_k) \) represent arbitrary, but fixed, permutations
- Transpositions account for pure overhead
- Requires additional memory
- Good if GEMM dominates performance (i.e., compute-bound)
- Bad if transpositions dominate performance (i.e., bandwidth-bound)
Loop-over-GEMM (LoG)

- Loop over 2D slices of the tensors
- Contract these 2D slices via GEMM

**Advantages**

- Exploits GEMM’s high-performance
- No additional memory

**Disadvantages**

- Performance can become arbitrarily poor
- Sometimes not applicable (if stride-one accesses are required)

\[
C_{m_1,n_1,m_2,n_2} = A_{m_1,m_2,k_1}B_{k_1,n_1,n_2}
\]

```c
for m_2 = 0: M_2
  for n_2 = 0: N_2
    GEMM( &A[m_2 * M_1], &B[n_2 * K_1 * N_1], &C[ m_2 * M_1 * n_1 + n_2 * m_1 * N_1 * M_2 ])
```

Paul Springer (AICES)
Tensor Contraction Code Generator
Sep. 20th 2016 9 / 9