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

Outline

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

[1] Paul Springer and Paolo Bientinesi. "Design of a high-performance GEMM-like Tensor-Tensor Multiplication". In: *TOMS, in review* ().

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)

[1] Paul Springer and Paolo Bientinesi. "Design of a high-performance GEMM-like Tensor-Tensor Multiplication". In: *TOMS, in review* ().

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**

[1] Paul Springer and Paolo Bientinesi. "Design of a high-performance GEMM-like Tensor-Tensor Multiplication". In: *TOMS, in review* ().

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

[1] Paul Springer and Paolo Bientinesi. "Design of a high-performance GEMM-like Tensor-Tensor Multiplication". In: *TOMS, in review* ().

Matrix-Matrix Multiplication

$A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$ 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

Matrix-Matrix Multiplication (Einstein notation)

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

$C_{m,n} \leftarrow A_{m,k} B_{k,n}$

Matrix-Matrix Multiplication (Einstein notation)

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

$C_{m,n} \leftarrow A_{m,k} B_{k,n}$

```
// 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} = α tmp + β C_{i,j}
```

Naive GEMM.

RWTH AACHEN UNIVERSITY

Matrix-Matrix Multiplication (Einstein notation)

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

```
// 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} = α tmp + β C_{i,j}
```

Naive GEMM.

```
// N-Loop
for n = 0 : nc : N − 1
   // K-Loop (contracted)
   for k = 0 : kc : K − 1
      B̂ = identify_submatrix(B, n, k)
      // pack B̂ into B̃
      B̃ = packB(B̂) // B̃ ∈ ℝ^{kc×nc}
      // M-Loop
      for m = 0 : mc : M − 1
         Â = identify_submatrix(A, m, k)
         // pack Â into Ã
         Ã = packA(Â) // Ã ∈ ℝ^{mc×kc}
         Ĉ = identify_submatrix(C, m, n)
         // matrix-matrix product: ÃB̃
         macroKernel(Ã, B̃, Ĉ, α, β)
```

High-performance GEMM.

AICES

- 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 Contractions

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

Tensor Contractions

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

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 Contractions

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

$\Rightarrow$ Quite similar to GEMM.

Tensor-Tensor Multiplication (Einstein notation)

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

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

Tensor-Tensor Multiplication (Einstein notation)

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

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

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

GETT

```
// N-Loops
for n_1 = 1 : S_{n_1}
   // ... remaining N-loops omitted ...
   for n_ζ = 1 : S_{n_ζ}
      // M-Loops
      for m_1 = 1 : S_{m_1}
         // ... remaining M-loops omitted ...
         for m_γ = 1 : S_{m_γ}
            tmp = 0
            // K-Loops (contracted)
            for k_1 = 1 : S_{k_1}
               // ... remaining K-loops omitted ...
               for k_ξ = 1 : S_{k_ξ}
                  tmp += A_{Π^A(m_1,...,m_γ,k_1,...,k_ξ)} B_{Π^B(k_1,...,k_ξ,n_1,...,n_ζ)}
                  // update C
            C_{Π^C(m_1,...,m_γ,n_1,...,n_ζ)} = α tmp + β C_{Π^C(m_1,...,m_γ,n_1,...,n_ζ)}
```

Naive GETT.

GETT

```
// 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 += $\mathcal{A}_{\Pi^{\mathcal{A}}(m_1,\dots,m_\gamma,k_1,\dots,k_\xi)} \mathcal{B}_{\Pi^{\mathcal{B}}(k_1,\dots,k_\xi,n_1,\dots,n_\zeta)}$
                  // update $\mathcal{C}$
            $\mathcal{C}_{\Pi^{\mathcal{C}}(m_1,\dots,m_\gamma,n_1,\dots,n_\zeta)} = \alpha$ tmp + $\beta \mathcal{C}_{\Pi^{\mathcal{C}}(m_1,\dots,m_\gamma,n_1,\dots,n_\zeta)}$
```

Naive GETT.

```
// N-Loops
for n_1 = 1 : S_{n_1}
   // ... remaining N-loops omitted ...
   for n_ζ = 1 : S_{n_ζ}
      // M-Loops
      for m_1 = 1 : S_{m_1}
         // ... remaining M-loops omitted ...
         for m_γ = 1 : S_{m_γ}
            tmp = 0
            // K-Loops (contracted)
            for k_1 = 1 : S_{k_1}
               // ... remaining K-loops omitted ...
               for k_ξ = 1 : S_{k_ξ}
                  tmp += A_{Π^A(m_1,...,m_γ,k_1,...,k_ξ)} B_{Π^B(k_1,...,k_ξ,n_1,...,n_ζ)}
                  // update C
            C_{Π^C(m_1,...,m_γ,n_1,...,n_ζ)} = α tmp + β C_{Π^C(m_1,...,m_γ,n_1,...,n_ζ)}
```

Naive GETT.

GETT

```
// N-Loops
for n_1 = 1 : S_{n_1}
   // ... remaining N-loops omitted ...
   for n_ζ = 1 : S_{n_ζ}
      // M-Loops
      for m_1 = 1 : S_{m_1}
         // ... remaining M-loops omitted ...
         for m_γ = 1 : S_{m_γ}
            tmp = 0
            // K-Loops (contracted)
            for k_1 = 1 : S_{k_1}
               // ... remaining K-loops omitted ...
               for k_ξ = 1 : S_{k_ξ}
                  tmp += A_{Π^A(m_1,...,m_γ,k_1,...,k_ξ)} B_{Π^B(k_1,...,k_ξ,n_1,...,n_ζ)}
                  // update C
            C_{Π^C(m_1,...,m_γ,n_1,...,n_ζ)} = α tmp + β C_{Π^C(m_1,...,m_γ,n_1,...,n_ζ)}
```

Naive GETT.

GETT

```
// N-Loop
for n = 1 : nc : S_{I_n}
   // K-Loop (contracted)
   for k = 1 : kc : S_{I_k}
      \widehat{\mathcal{B}} = identify_subtensor(B, n, k)
      // pack \widehat{\mathcal{B}} into \widetilde{\mathcal{B}}
      \widetilde{\mathcal{B}} = packB(\widehat{\mathcal{B}})
      // M-Loop
      for m = 1 : mc : S_{I_m}
         \widehat{\mathcal{A}} = identify_subtensor(A, m, k)
         // pack \widehat{\mathcal{A}} into \widetilde{\mathcal{A}}
         \widetilde{\mathcal{A}} = packA(\widehat{\mathcal{A}})
         \widehat{\mathcal{C}} = identify_subtensor(C, m, n)
         // compute matrix-matrix product of \widetilde{\mathcal{A}}\widetilde{\mathcal{B}}
         macroKernel(\widetilde{\mathcal{A}}, \widetilde{\mathcal{B}}, \widehat{\mathcal{C}}, \alpha, \beta)
```

High-performance GETT.

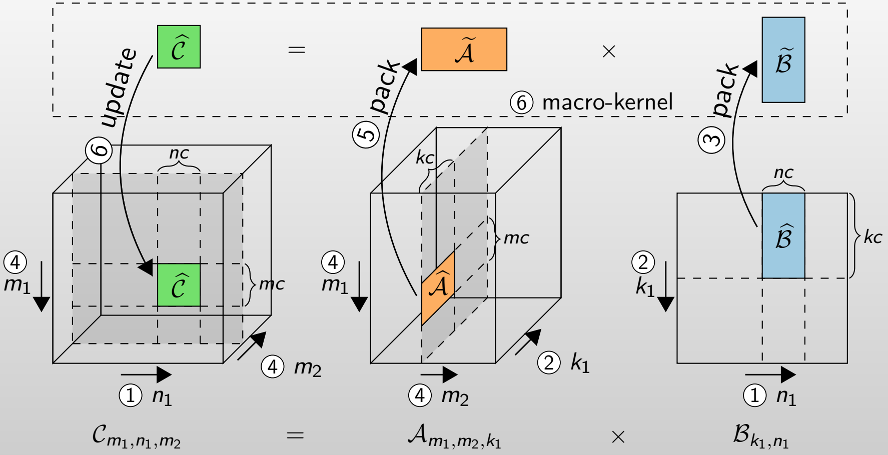
GETT

```
// N-Loop
for n = 1 : nc : S_{I_n}
   // K-Loop (contracted)
   for k = 1 : kc : S_{I_k}
      \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_{I_m}
         \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}, α, β)
```

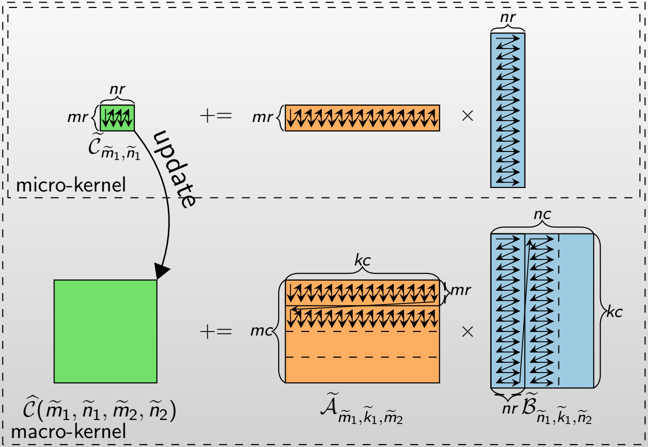
High-performance GETT.

Key Idea

Pack-and-transpose while moving data into the caches

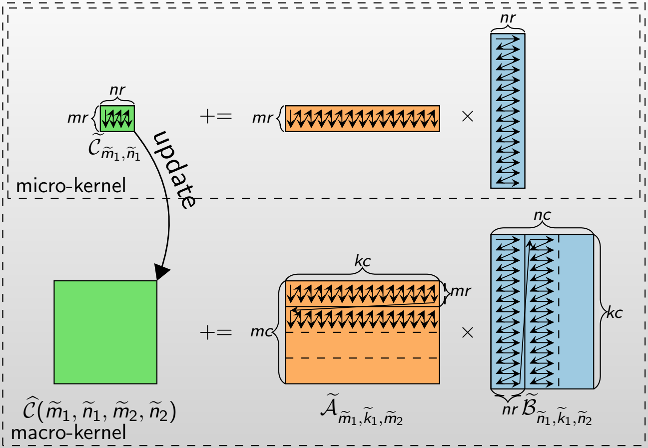

$\widehat{\mathcal{C}}$
=
$\widetilde{\mathcal{A}}$
×
$\widetilde{\mathcal{B}}$
update
pack
pack
6 macro-kernel
6
5
3
nc
kc
mc
mc
nc
kc
$\widehat{\mathcal{C}}$
$\widehat{\mathcal{A}}$
$\widehat{\mathcal{B}}$
4 m_1
4 m_1
2 k_1
1 n_1
4 m_2
4 m_2
2 k_1
1 n_1
$\mathcal{C}_{m_1,n_1,m_2}$
=
$\mathcal{A}_{m_1,m_2,k_1}$
×
$\mathcal{B}_{k_1,n_1}$

RWTH AACHEN UNIVERSITY

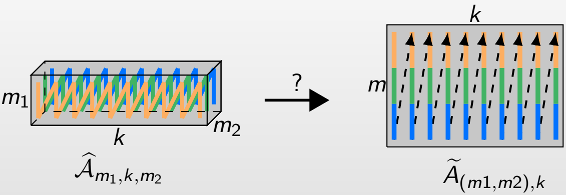


- Blocking for L3, L2, L1 cache as well as registers

GETT: Macro- /Micro-Kernel

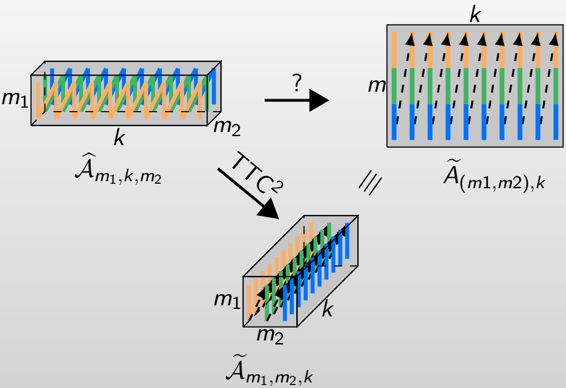


- Blocking for L3, L2, L1 cache as well as registers
- Written in AVX2 intrinsics



m_1
k
m_2
$\widehat{\mathcal{A}}_{m_1,k,m_2}$
?
k
m
$\widetilde{A}_{(m1,m2),k}$

Packing via Tensor Transpositions



- Preserve stride-1 index
  - $\Rightarrow$ Efficient packing routines

[2] Paul Springer, Jeff R. Hammond, and Paolo Bientinesi. "TTC: A high-performance Compiler for Tensor Transpositions". In: *TOMS, in review* ().

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

Tensor Contraction Code Generator (TCCG)

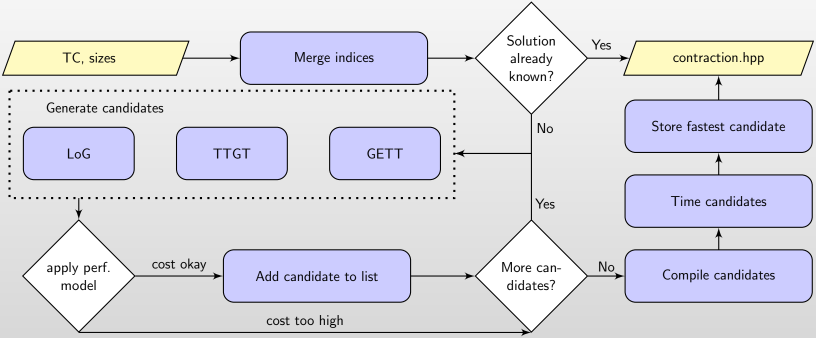
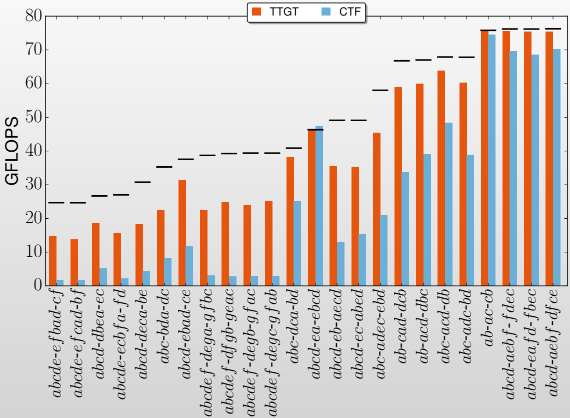


Figure: Schematic overview of TCCG.

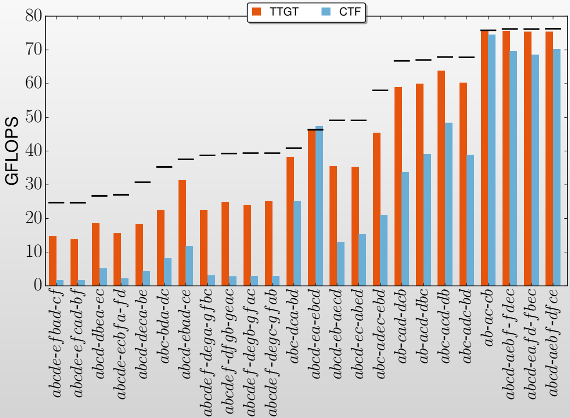
Performance

- 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

Performance



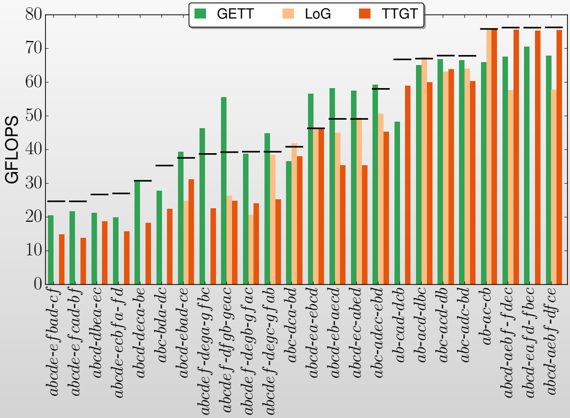
Performance



- TTGT good in compute-bound regime
- TTGT bad in bandwidth-bound regime
- TTGT faster than CTF everywhere.

Performance

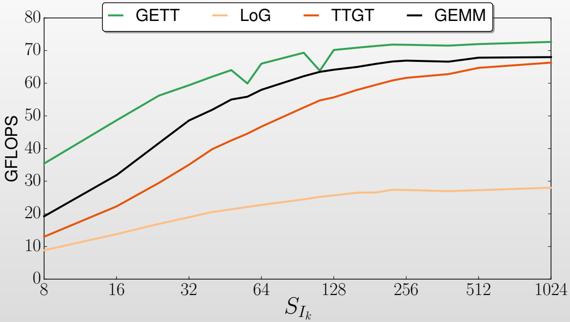
RWTH AACHEN UNIVERSITY



- GETT excels in bandwidth-bound regime.
- GETT slightly lags behind in compute-bound regime.

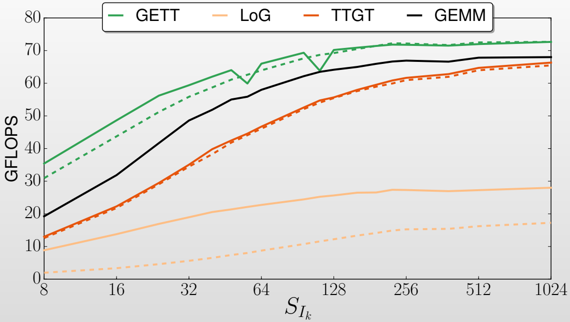
AICES

Performance: $i_1 j i_2$-$i_1 k i_2$-jk



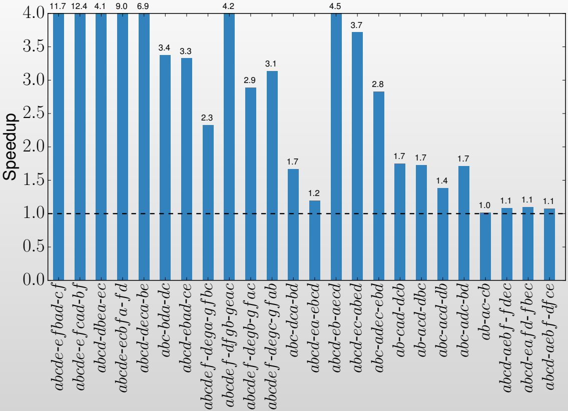
- GETT especially good in bandwidth-bound regime
  - GETT still attains up to 91.3% of peak floating-point performance
- TTGT poor in bandwidth-bound regime

RWTH AACHEN UNIVERSITY



- 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



- 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`

RWTH AACHEN UNIVERSITY

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`

Future Work

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

ces

Conclusion and Future Work

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

Thank you for your attention.

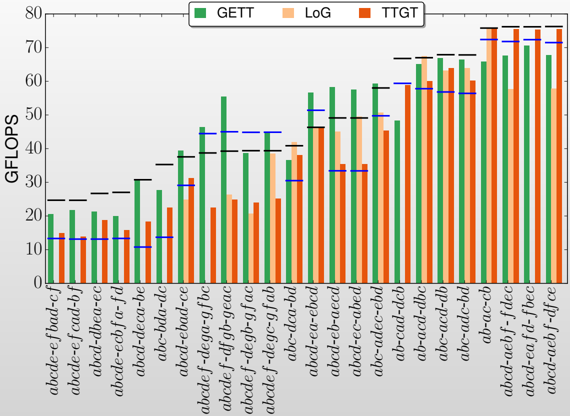
- Give it a try: `https://github.com/HPAC/tccg`

Future Work

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

Performance - SP

RWTH AACHEN UNIVERSITY

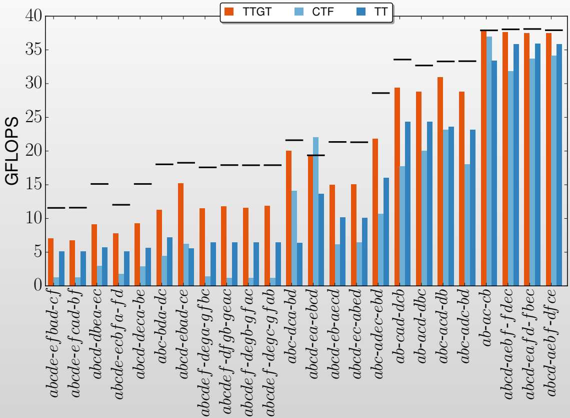


- GETT excels in bandwidth-bound regime.
- GETT slightly lags behind in compute-bound regime.
- GETT attains min/avg/max performance of GEMM:
  - SP: 72.4% / 98.1% / 141.4%
  - DP: 60.8% / 97.0% / 132.9%

AICES

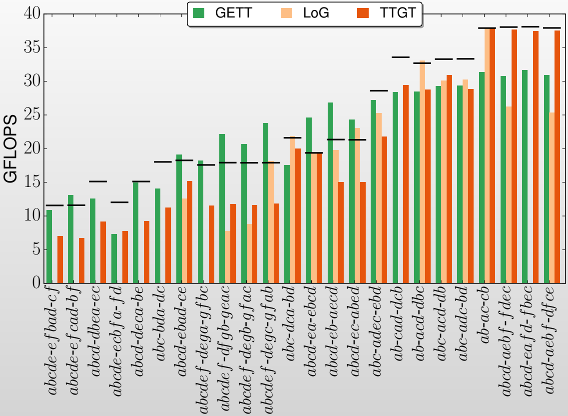
Performance - DP

RWTH AACHEN UNIVERSITY



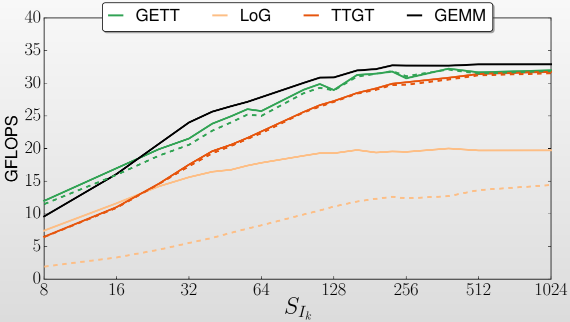
- TTGT faster than CTF everywhere.
- TTGT good in compute-bound regime
- TTGT bad in bandwidth-bound regime

AICES

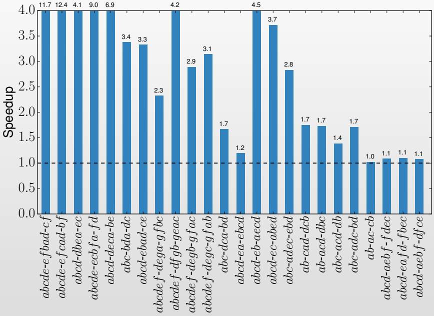


- GETT excels in bandwidth-bound regime.
- GETT slightly lags behind in compute-bound regime.
- GETT attains min/avg/max performance of GEMM:
  - SP: 72.4% / 98.1% / 141.4%
  - DP: 60.8% / 97.0% / 132.9%

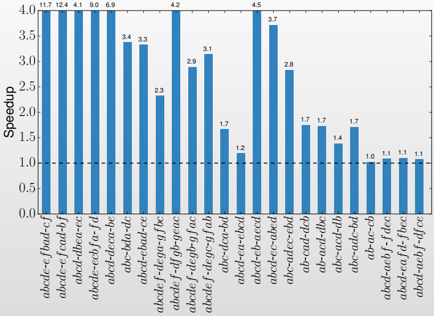
Performance: $i_1 j_1 i_2 j_2$-$i_1 k i_2$-$j_1 k j_2$ - DP



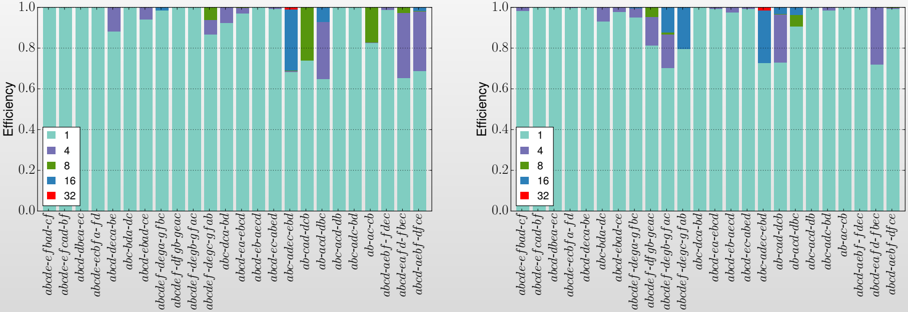
- GETT especially good in bandwidth-bound regime
  - GETT still attains up to 91.3% of peak floating-point performance
- TTGT poor in bandwidth-bound regime
- LoG performance can become arbitrarily bad
- GETT and TTGT barely affected by higher dimensions



(a) Single-Precision.



(b) Double-Precision.



(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%

A I ces

```
C[a,b,i,j] = A[i,m,a] * B[m,j,b]
a = 24
b = 24
i = 24
j = 24
m = 24
```

Figure: Exemplary input file for TCCG.

Argument	Description
`--floatType=[s,d]`	data type
`--maxWorkspace=<value>`	maximum auxiliary workspace in GB
`--maxImplementations=<value>`	maximum #implementations
`--arch=[hsw,knl,cuda]`	selected architecture
`--numThreads=<value>`	number of threads

Table: TCCG's command line arguments.

Transpose-Transpose-GEMM-Transpose

$A_{\Pi^m(I_m),\Pi^k(I_k)} \leftarrow \mathcal{A}_{\Pi^{\mathcal{A}}(I_m \cup I_k)}$ `// unfold` $\mathcal{A}$
$B_{\Pi^k(I_k),\Pi^n(I_n)} \leftarrow \mathcal{B}_{\Pi^{\mathcal{B}}(I_n \cup I_k)}$ `// unfold` $\mathcal{B}$
$X_{\Pi^m(I_m),\Pi^n(I_n)} \leftarrow \mathtt{op}(A)_{\Pi^m(I_m),\Pi^k(I_k)} \times \mathtt{op}(B)_{\Pi^k(I_k),\Pi^n(I_n)}$ `// contract` A `and` B `via a GEMM`
$\mathcal{C}_{\Pi^{\mathcal{C}}(I_m \cup I_n)} \leftarrow X_{\Pi^m(I_m),\Pi^n(I_n)}$ `// fold` X

TTGT pseudo code for a general tensor contraction
$\mathcal{C}_{\Pi^{\mathcal{C}}(I_m \cup I_n)} = \mathcal{A}_{\Pi^{\mathcal{A}}(I_m \cup I_k)} \mathcal{B}_{\Pi^{\mathcal{B}}(I_n \cup I_k)} + \mathcal{C}_{\Pi^{\mathcal{C}}(I_m \cup I_n)}$.

- $\Pi^m(I_m), \Pi^n(I_n)$ and $\Pi^k(I_k)$ 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)

- $C_{m_1,n_1,m_2,n_2} = A_{m_1,m_2,k_1} B_{k_1,n_1,n_2}$

```
for m2 = 0: M2
    for n2= 0: N2
        GEMM( &A[m2 * M1], &B[n2 * K1 * N1], &C[
               m2 * M1 * n1 + n2 * m1 * N1 * M2])
```