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

Paul Springer and Paolo Bientinesi

Aachen Institute for Advanced Study in
Computational Engineering Science

Atlanta, Feb. 27th 2017

- Tensors are multi-dimensional arrays
  - 0-order tensor: scalar α
  - 1-order tensor: vector $\mathcal{A}_{i_1}$
  - 2-order tensor: matrix $\mathcal{A}_{i_1,i_2}$
  - n-order tensor: $\mathcal{A}_{i_1,i_2,\ldots,i_n}$

Introduction

- Tensors are multi-dimensional arrays
  - 0-order tensor: scalar α
  - 1-order tensor: vector $\mathcal{A}_{i_1}$
  - 2-order tensor: matrix $\mathcal{A}_{i_1,i_2}$
  - n-order tensor: $\mathcal{A}_{i_1,i_2,\ldots,i_n}$
- Tensor contraction can be thought of as higher dimensional GEMMs

- Tensors are multi-dimensional arrays
  - 0-order tensor: scalar α
  - 1-order tensor: vector $\mathcal{A}_{i_1}$
  - 2-order tensor: matrix $\mathcal{A}_{i_1,i_2}$
  - n-order tensor: $\mathcal{A}_{i_1,i_2,\ldots,i_n}$
- Tensor contraction can be thought of as higher dimensional GEMMs
- Essentially three approaches:
  - 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
  - Adopts the BLIS methodology: **Breaking through the BLAS layer**
  - Conceptually similar to direct convolutions[2]

[1] Paul Springer et al. "Design of a high-performance GEMM-like Tensor-Tensor Multiplication"

[2] Sharan Chetlur et al. "CUDNN: Efficient Primitives for Deep Learning"

- We propose a novel approach: GETT[1]
  - Akin to a high-performance GEMM implementation
  - Adopts the BLIS methodology: **Breaking through the BLAS layer**
  - Conceptually similar to direct convolutions[2]
- Tensor Contraction Code Generator (TCCG)
  - combine GETT, TTGT and LoG into a unified tool

[1] Paul Springer et al. "Design of a high-performance GEMM-like Tensor-Tensor Multiplication"

[2] Sharan Chetlur et al. "CUDNN: Efficient Primitives for Deep Learning"

RWTH AACHEN UNIVERSITY

Matrix-Matrix Multiplication

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

RWTH AACHEN UNIVERSITY

Matrix-Matrix Multiplication (Einstein notation)

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

Matrix-Matrix Multiplication (Einstein notation)

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

```
// 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)

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

```
// 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.

Tensor Contractions

- Tensor contraction examples:
  - $C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1}$

- Tensor contraction examples:
  - $C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1}$
  - $C_{m_1,m_2,n_1} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_1}$

Tensor Contractions

- Tensor contraction examples:
  - $C_{m_1,n_1} \leftarrow A_{m_1,k_1} B_{k_1,n_1}$
  - $C_{m_1,m_2,n_1} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_1}$
  - $C_{m_1,n_1,m_2} \leftarrow A_{m_1,m_2,k_1} B_{k_1,n_1}$

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

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

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

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

$\Rightarrow$ Quite similar to GEMM.

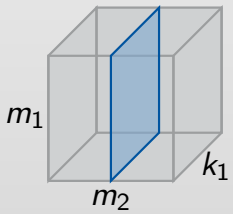
RWTH AACHEN UNIVERSITY

Key Idea

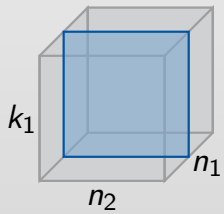
Identify 2D subtensors and contract them via GEMM

- $\mathcal{C}_{m_1,n_1,n_2,m_2} \leftarrow \mathcal{A}_{m_1,m_2,k_1}\mathcal{B}_{k_1,n_2,n_1}$

```
for( m2 = 0; m2 < M2; m2++ )
  for( n1 = 0; n1 < N1; n1++ )
    gemm(M1, N2, K1, A[:, m2, :], B[:, :, n1], C[:, n1, :, m2])
```



$\mathcal{A}_{m_1,m_2,k_1}$



$\mathcal{B}_{k_1,n_2,n_1}$

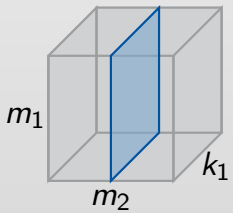
RWTH AACHEN UNIVERSITY

Key Idea

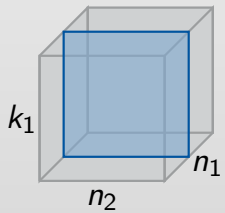
Identify 2D subtensors and contract them via GEMM

- $\mathcal{C}_{m_1,n_1,n_2,m_2} \leftarrow \mathcal{A}_{m_1,m_2,k_1}\mathcal{B}_{k_1,n_2,n_1}$

```
for( m2 = 0; m2 < M2; m2++ )
  for( n1 = 0; n1 < N1; n1++ )
    gemm(M1, N2, K1, A[:, m2, :], B[:, :, n1], C[:, n1, :, m2])
```

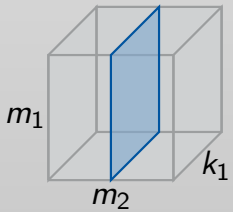


$\mathcal{A}_{m_1,m_2,k_1}$

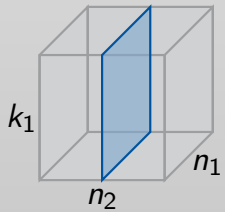


$\mathcal{B}_{k_1,n_2,n_1}$

```
for( m2 = 0; m2 < M2; m2++ )
  for( n2 = 0; n2 < N2; n2++ )
    gemm(M1, N1, K1, A[:, m2, :], B[:, n2, :], C[:, :, n2, m2])
```



$\mathcal{A}_{m_1,m_2,k_1}$



$\mathcal{B}_{k_1,n_2,n_1}$

RWTH AACHEN UNIVERSITY

Key Idea

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

- $\mathcal{C}_{m_1,n_1,n_2,m_2} \leftarrow \mathcal{A}_{m_1,m_2,k_1}\mathcal{B}_{k_1,n_2,n_1}$

AICES

Key Idea

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

- $\mathcal{C}_{m_1,n_1,n_2,m_2} \leftarrow \mathcal{A}_{m_1,m_2,k_1}\mathcal{B}_{k_1,n_2,n_1}$

$$\widetilde{\mathcal{C}}_{(m_1,m_2),(n_2,n_1)} \leftarrow \mathcal{A}_{(m_1,m_2),k_1} \times \mathcal{B}_{k_1,(n_2,n_1)}$$
$$\mathcal{C}_{m_1,n_1,n_2,m_2} \leftarrow \widetilde{\mathcal{C}}_{m_1,m_2,n_2,n_1}$$

Key Idea

Pack-and-transpose while moving data into the caches

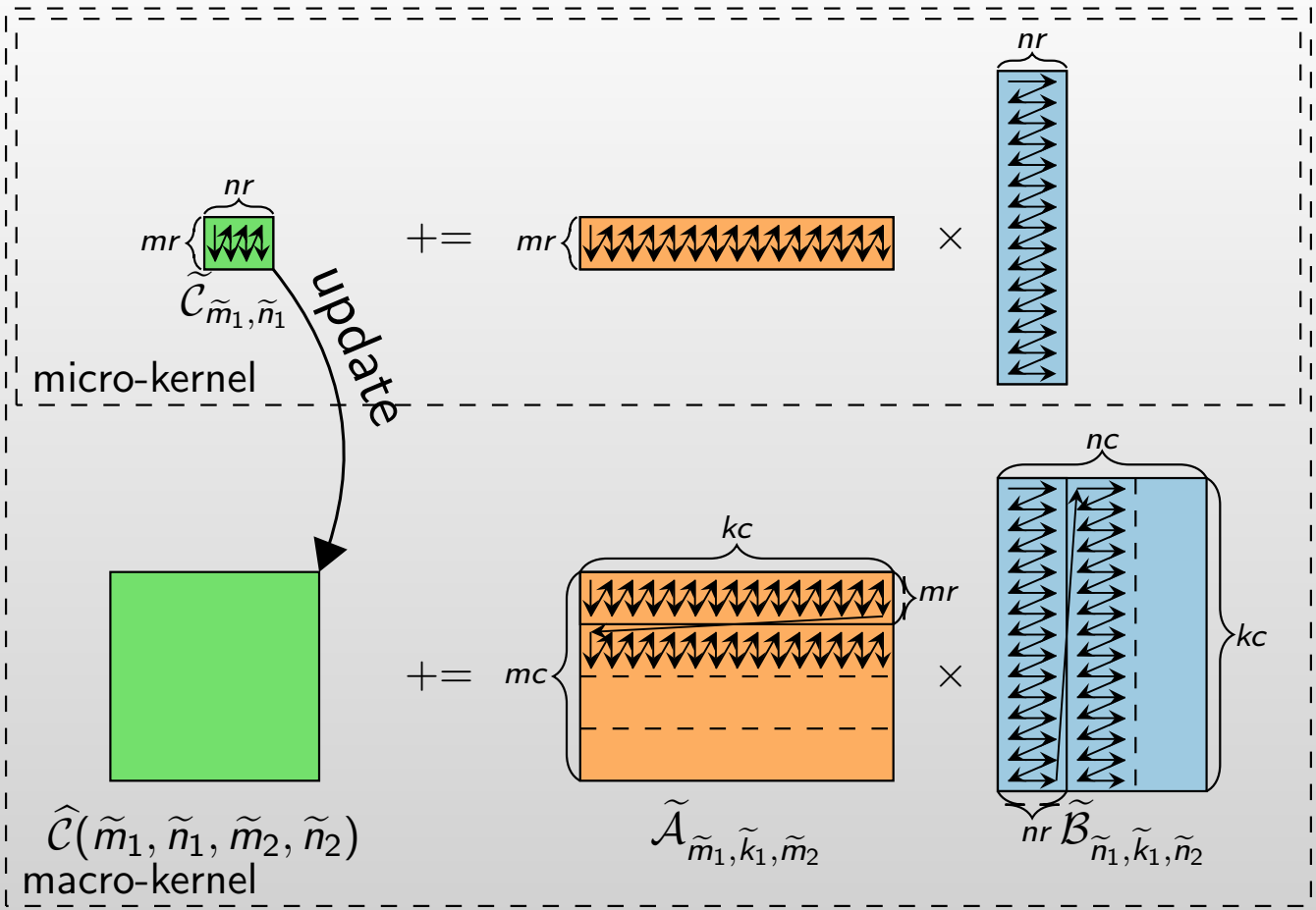
Key Idea

Pack-and-transpose while moving data into the caches

```
1  // N-Loop
2  for n = 1 : nc : S_{l_n}
3     // K-Loop (contracted)
4     for k = 1 : kc : S_{l_k}
5        B̂ = identify_subtensor(B, n, k)
6        // pack B̂ into B̃ (L3 cache)
7        B̃ = packB(B̂)
8        // M-Loop
9        for m = 1 : mc : S_{l_m}
10          Â = identify_subtensor(A, m, k)
11          // pack Â into Ã (L2 cache)
12          Ã = packA(Â)
13          Ĉ = identify_subtensor(C, m, n)
14          // compute matrix-matrix product of ÃB̃
15          macroKernel(Ã, B̃, Ĉ, α, β)
```

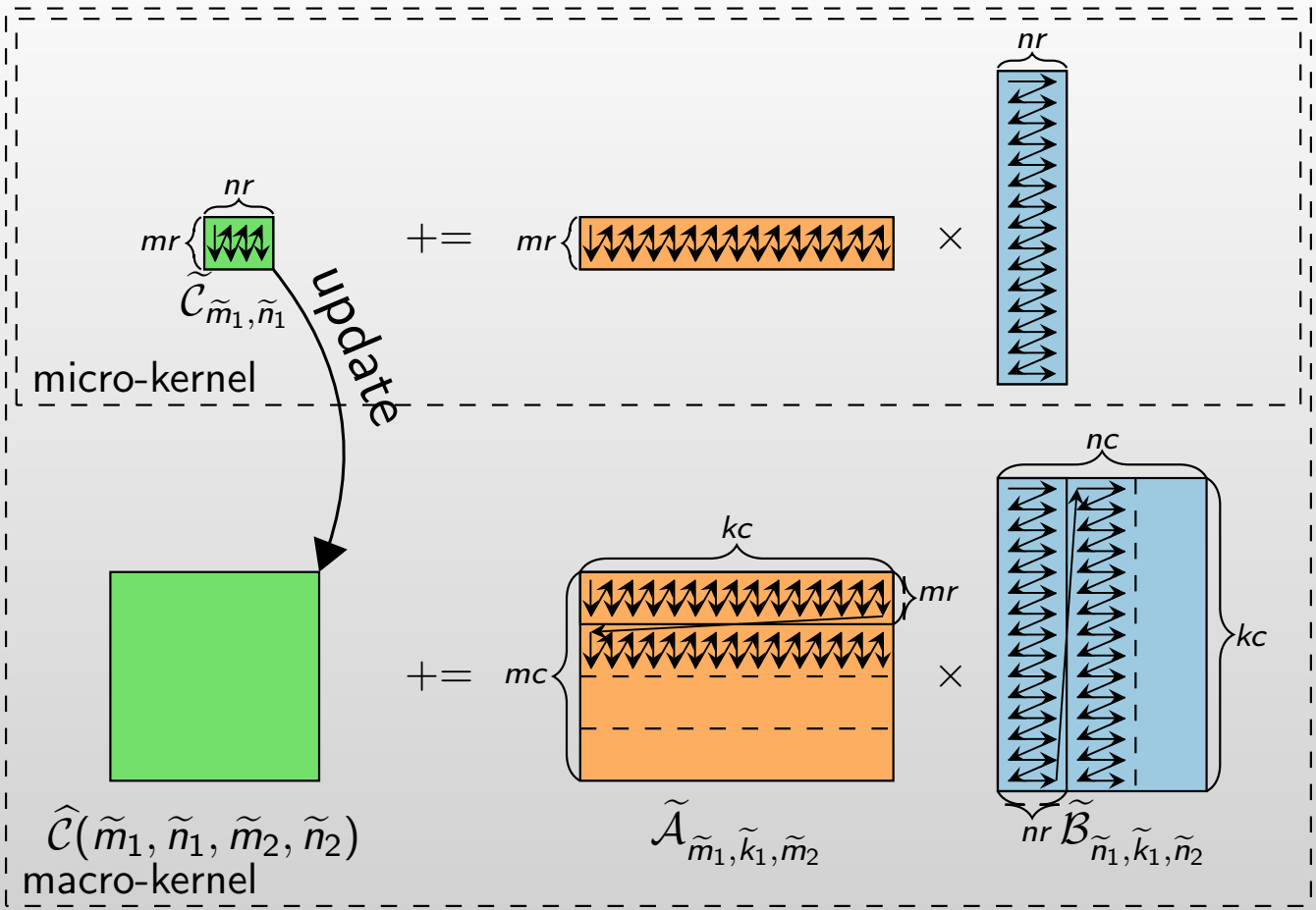
High-performance GETT.

GETT: Macro-/ Micro-Kernel



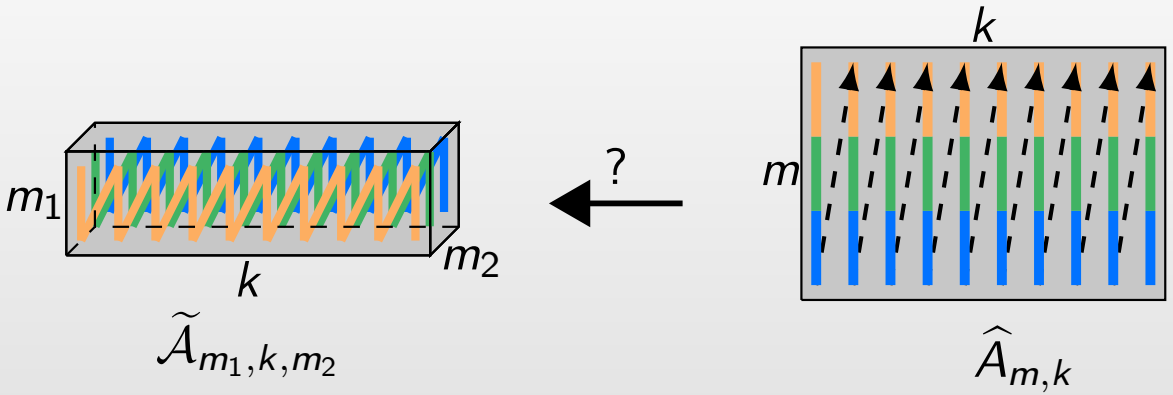
- Blocking for L3, L2, L1 cache as well as registers

GETT: Macro-/ Micro-Kernel

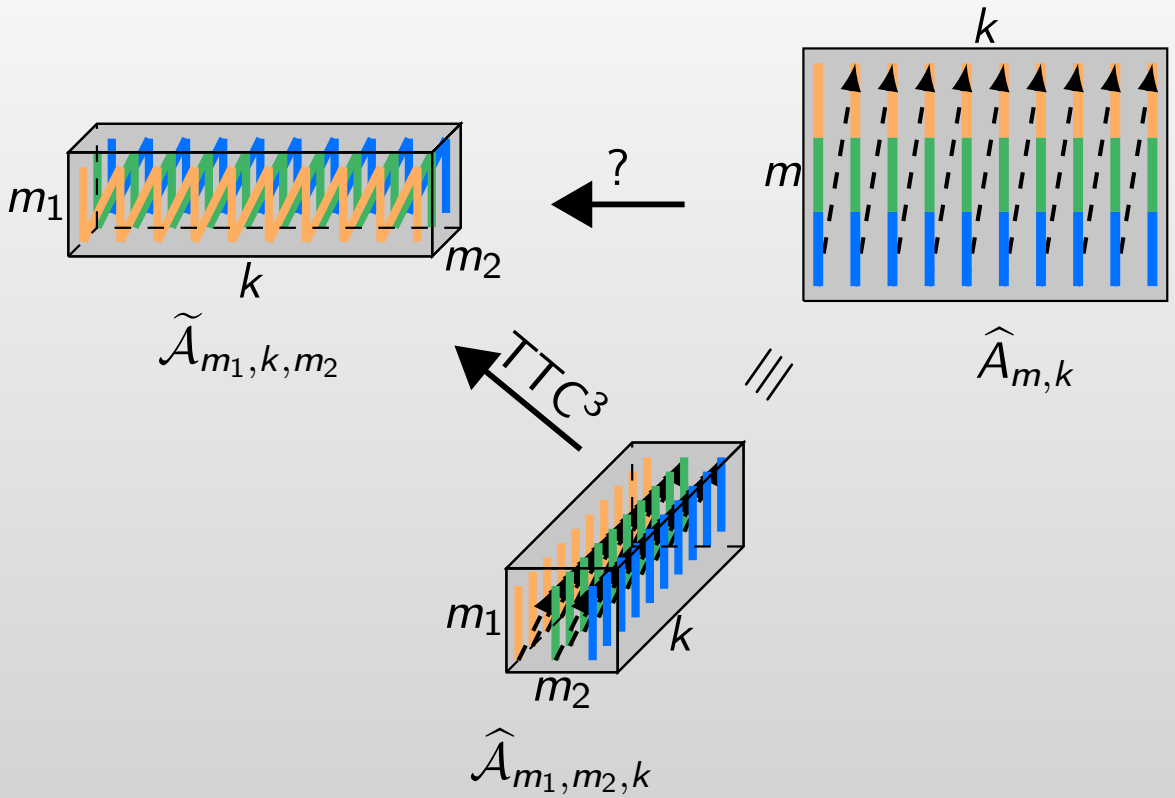


- Blocking for L3, L2, L1 cache as well as registers
- Written in AVX2 intrinsics

Packing via Tensor Transpositions



Packing via Tensor Transpositions



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

[3] Paul Springer et al. "TTC: A high-performance Compiler for Tensor Transpositions"

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 values for mc, nc and kc
- Prune the search space via a performance model

Tensor Contraction Code Generator (TCCG)

RWTH AACHEN UNIVERSITY

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

AICES

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

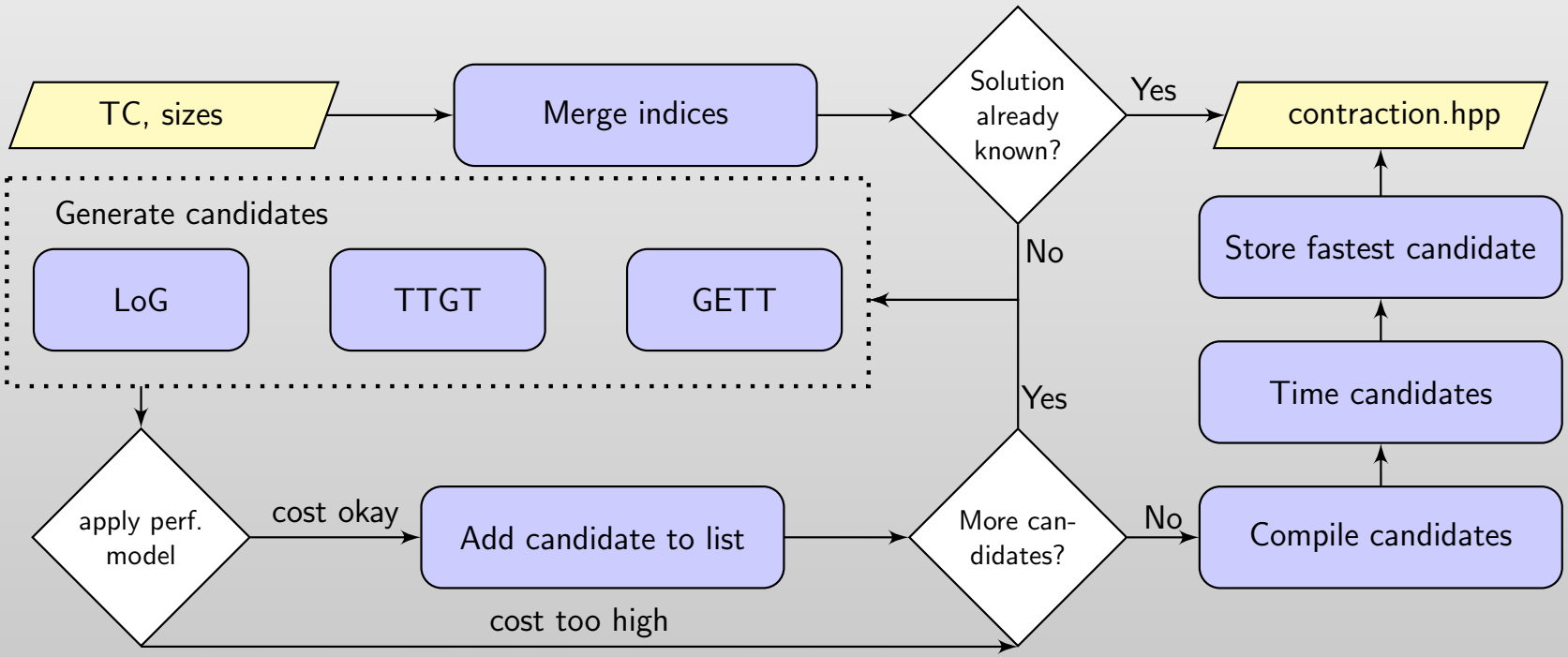
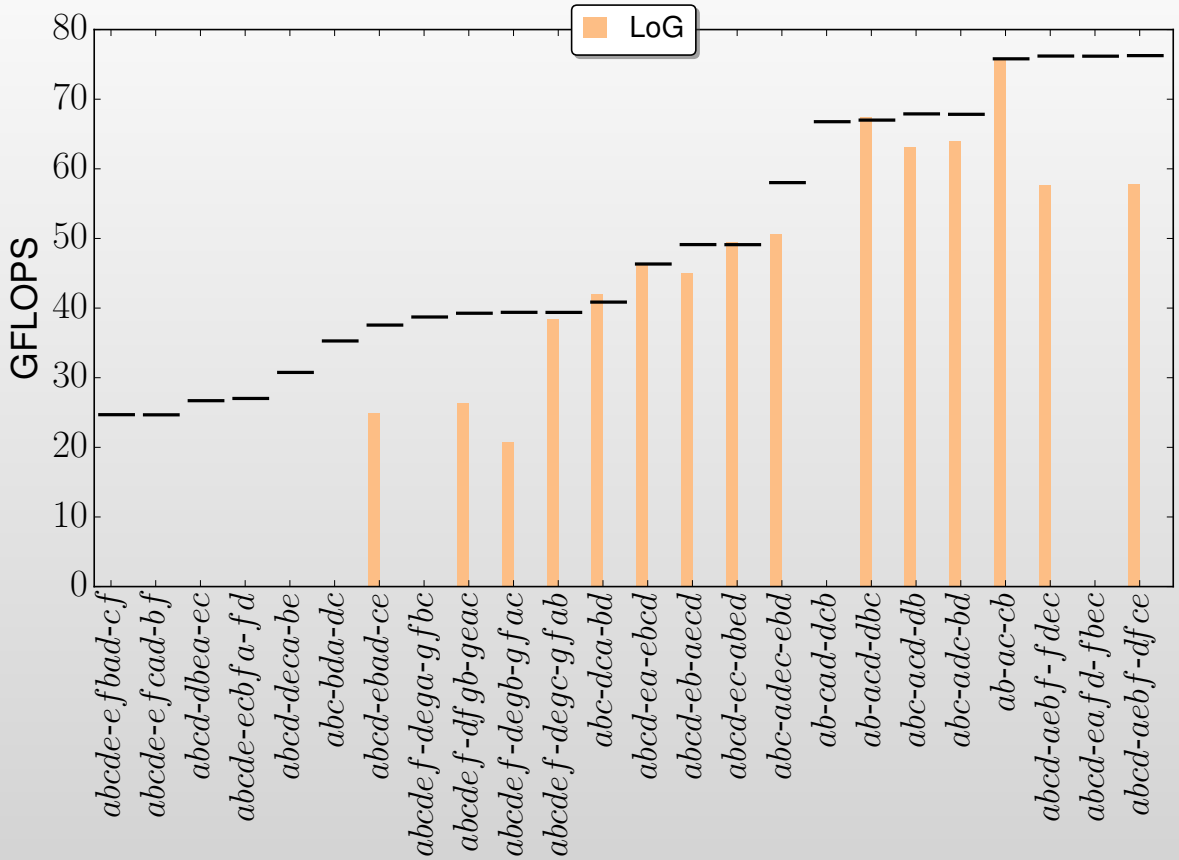
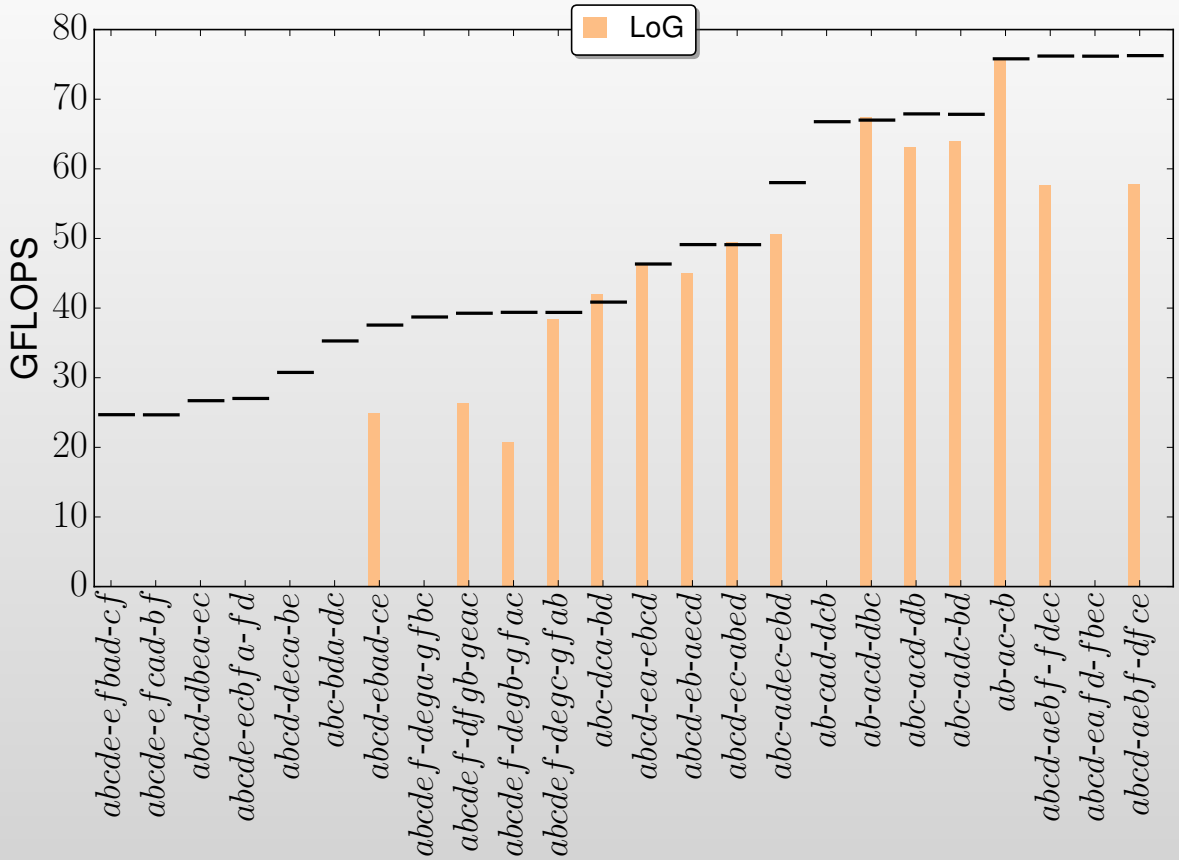


Figure: Schematic overview of TCCG.

Performance — Haswell (single core)

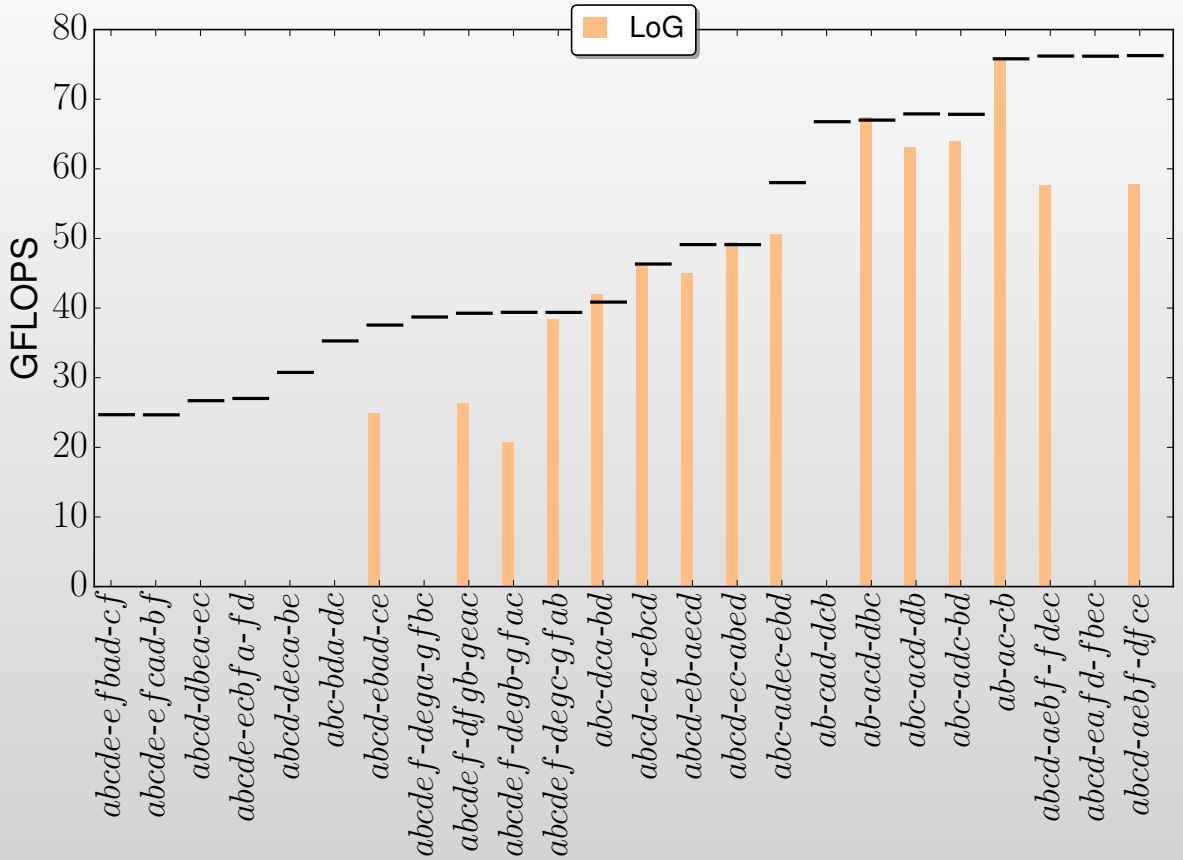


Performance — Haswell (single core)



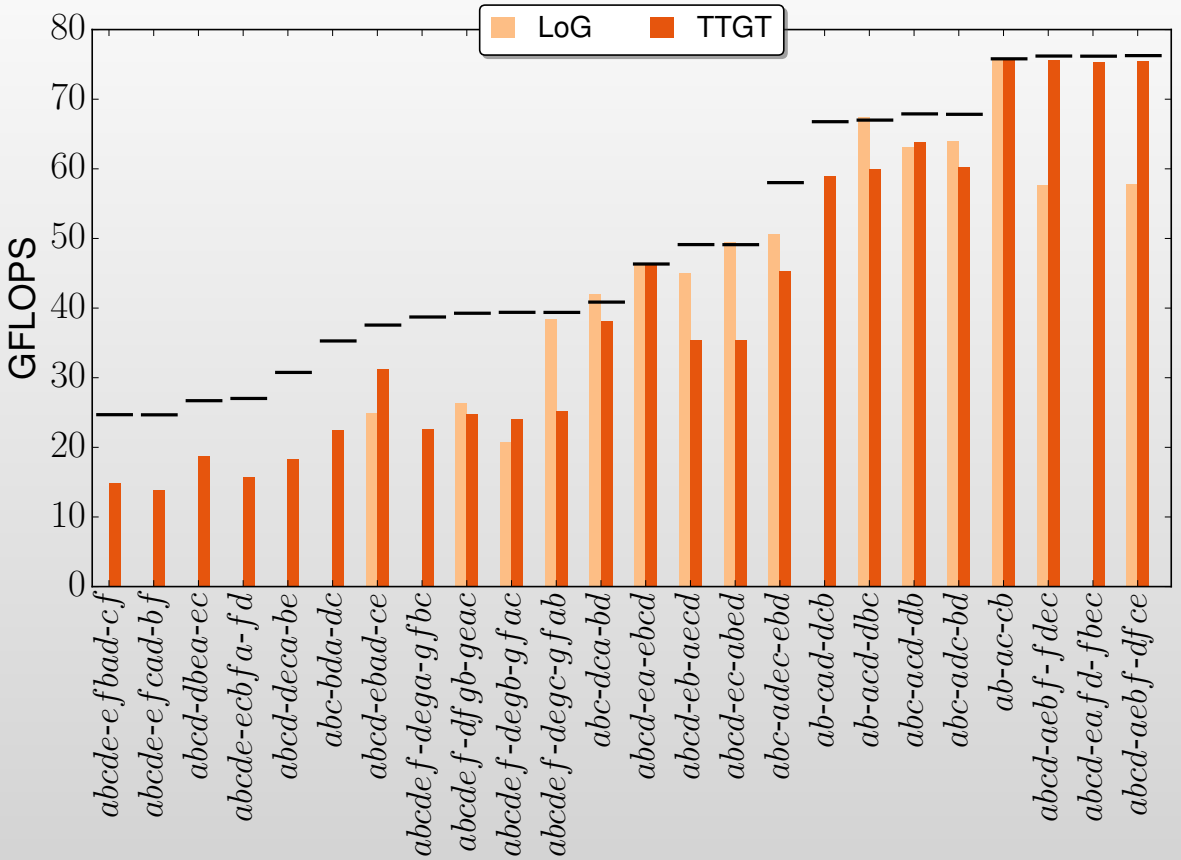
- Not all TCs can be implemented via LoG

Performance — Haswell (single core)

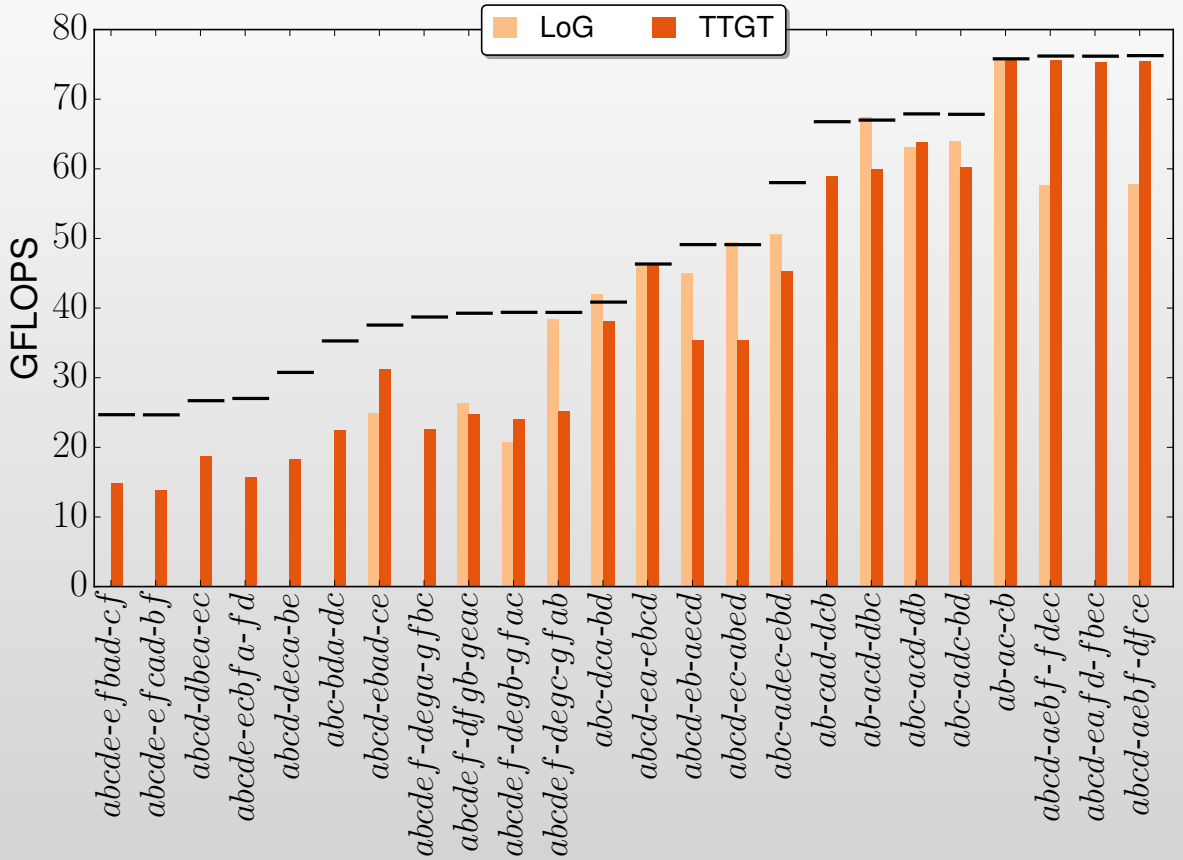


- Not all TCs can be implemented via LoG
- Mixed performance

Performance — Haswell (single core)

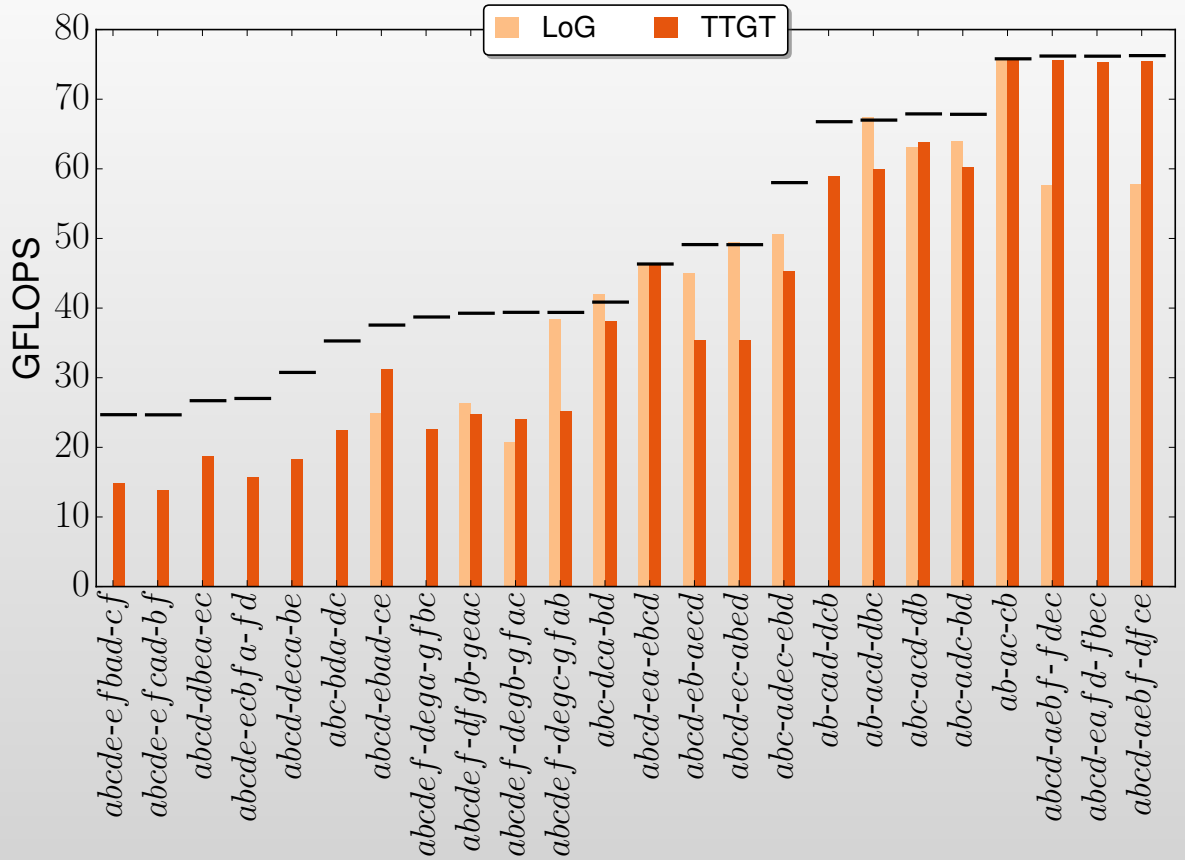


Performance — Haswell (single core)



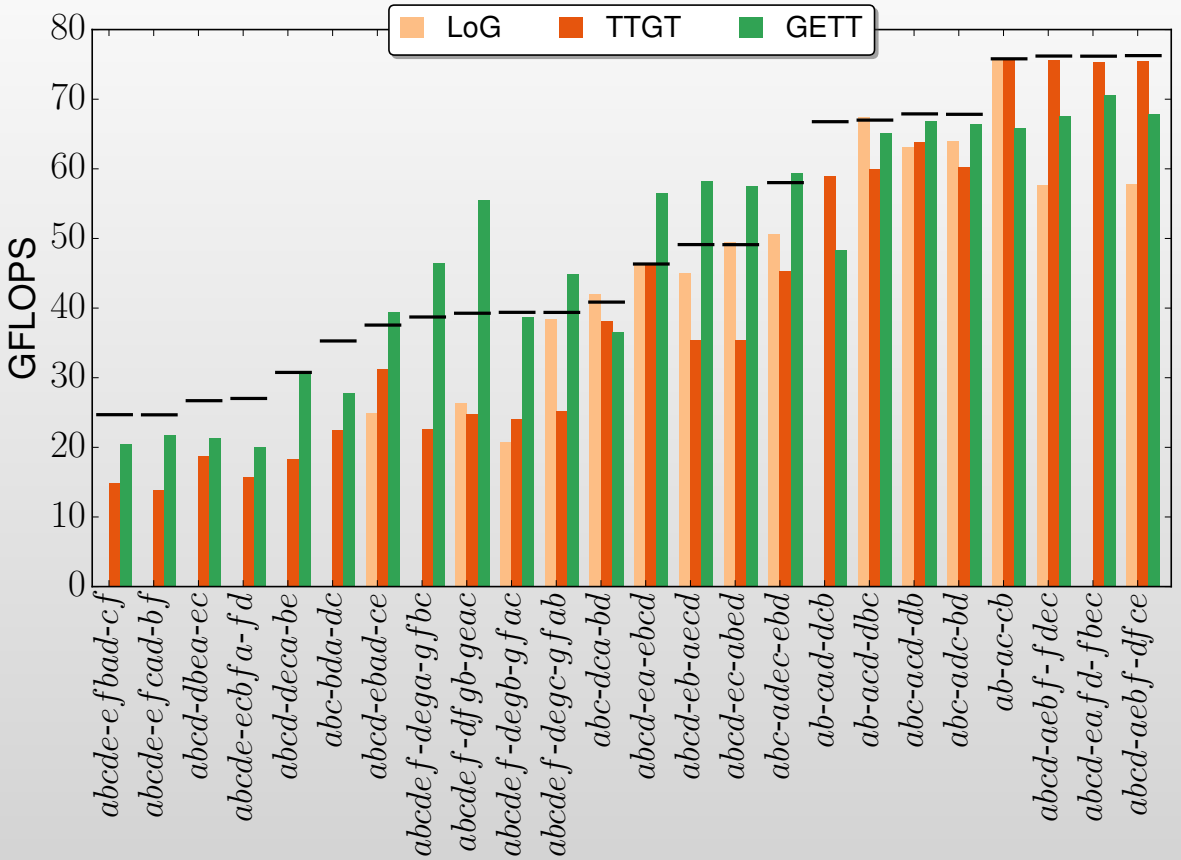
- TTGT: good for compute-bound TCs

Performance — Haswell (single core)

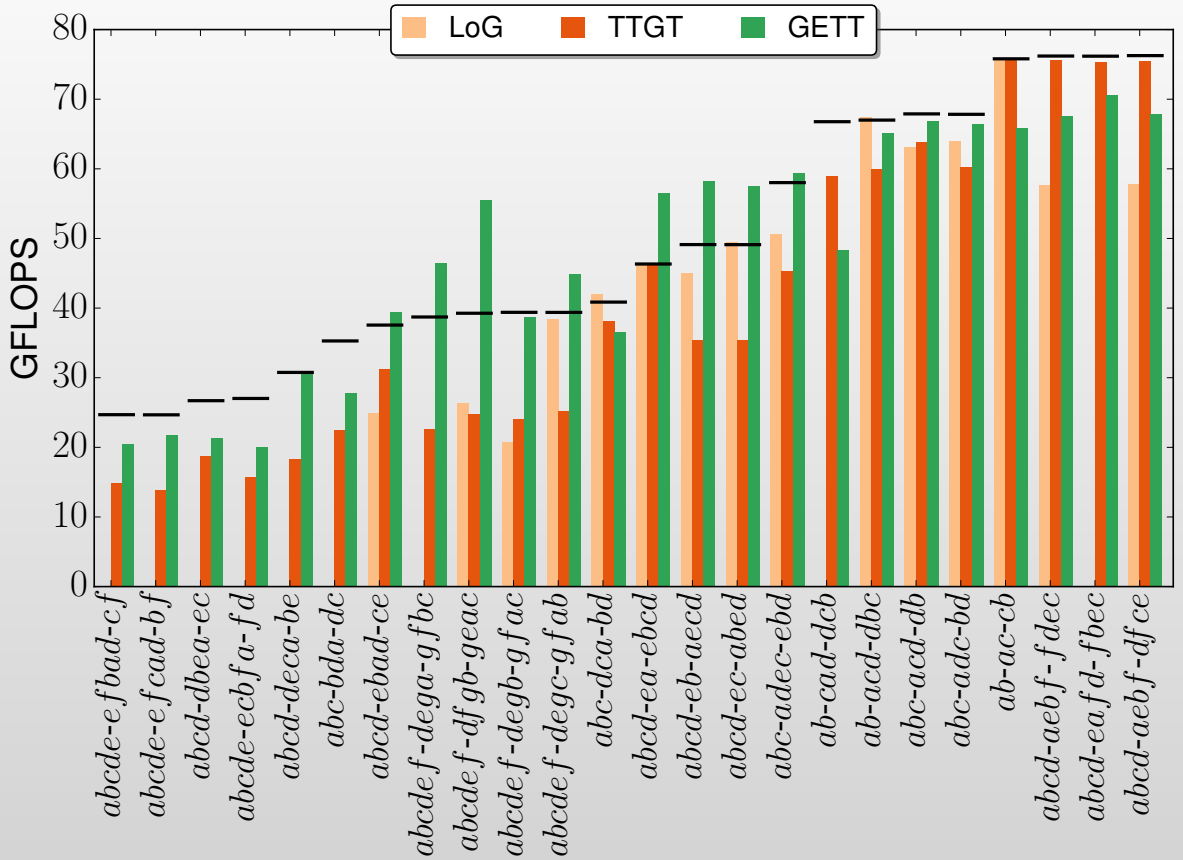


- TTGT: good for compute-bound TCs
- TTGT: bad for bandwidth-bound TCs

Performance — Haswell (single core)

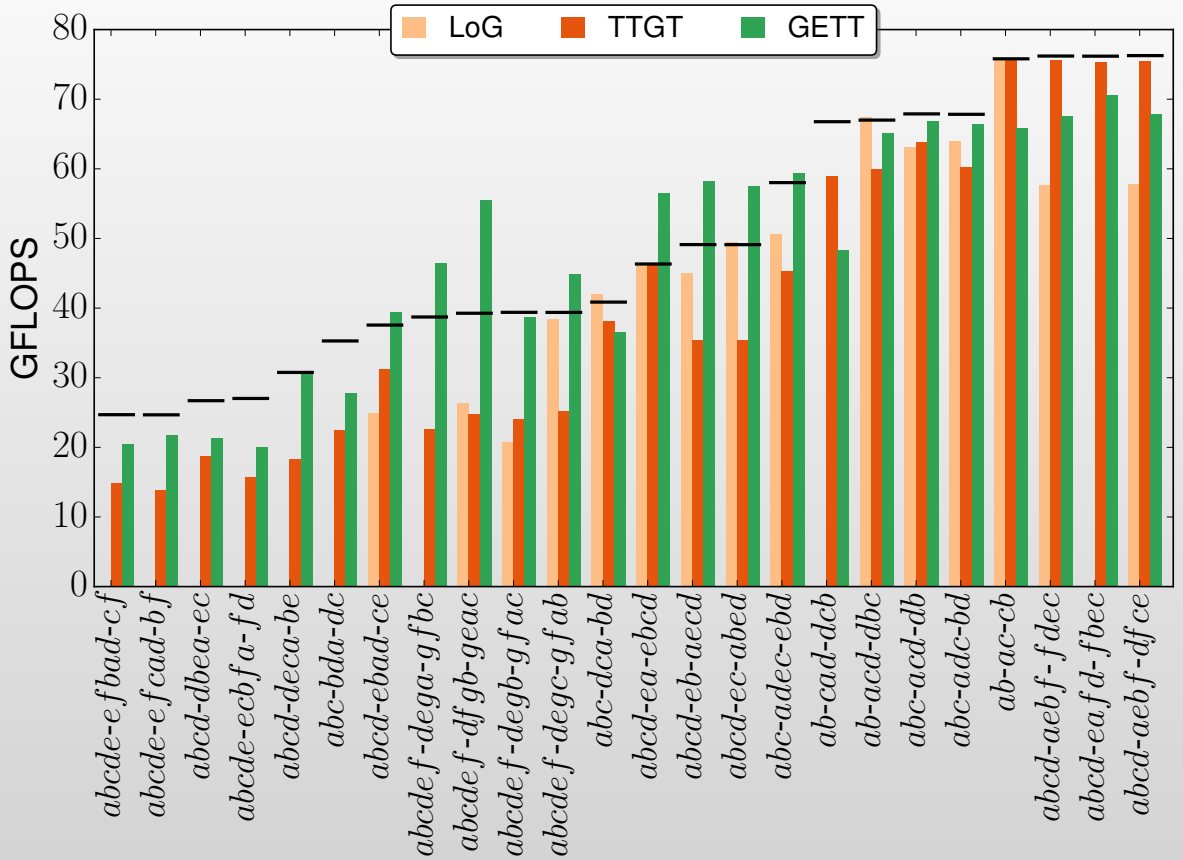


Performance — Haswell (single core)



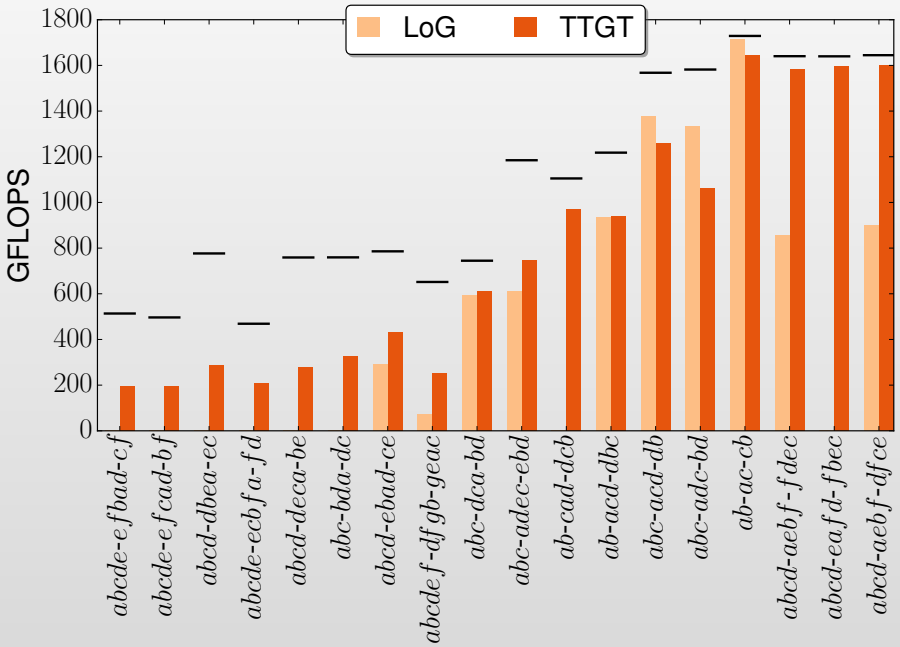
- GETT: excels for bandwidth-bound TCs

Performance — Haswell (single core)



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

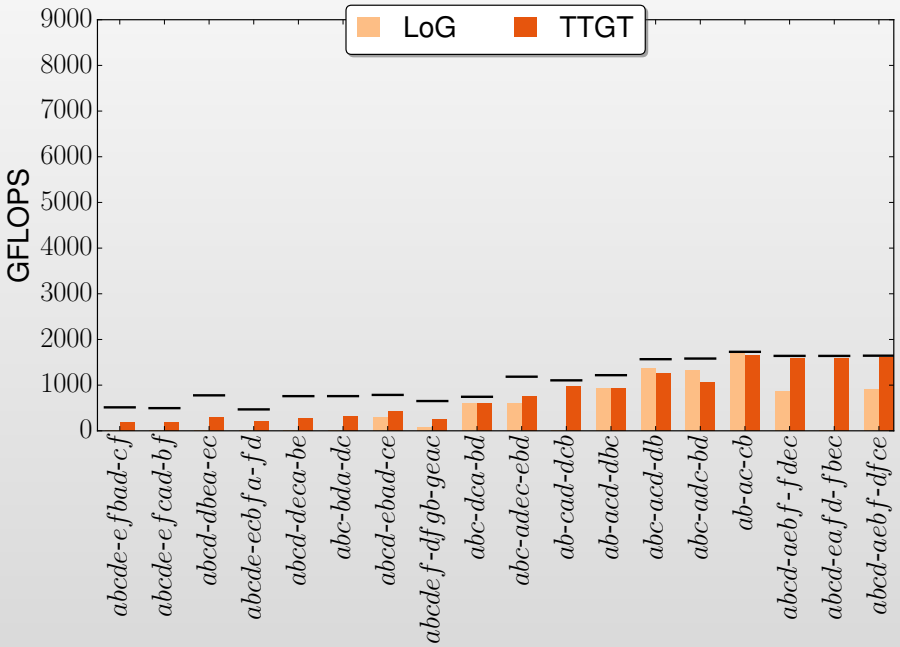
Performance — Multi-threaded



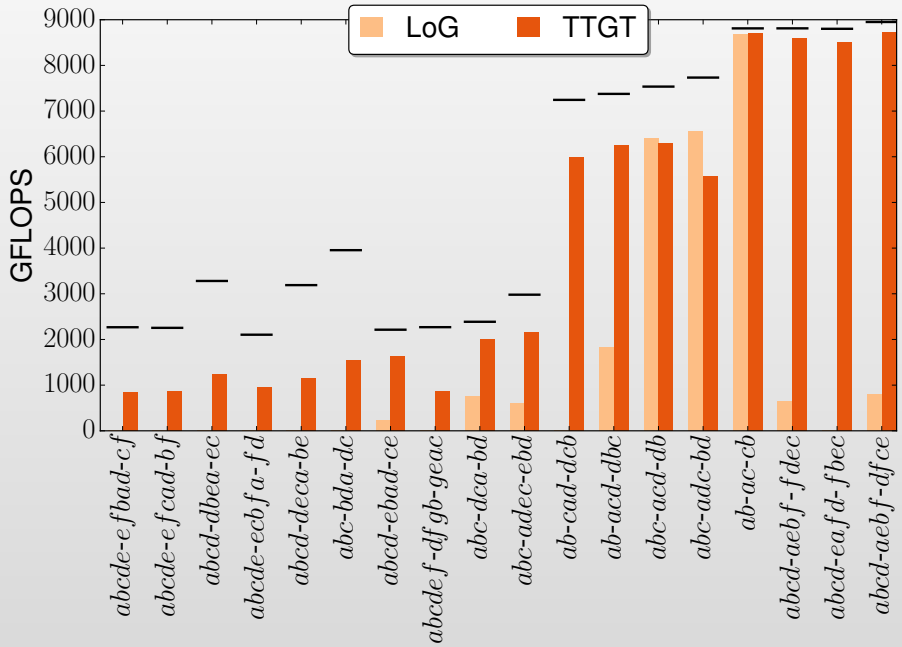
(a) 2×Intel Xeon E5-2680 v3

- Performance gap increases for bandwidth-bound TCs

Performance — Multi-threaded



(a) 2×Intel Xeon E5-2680 v3



(b) NVIDIA Tesla P100

- Performance gap increases for bandwidth-bound TCs

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
- TCCG is available at `https://github.com/HPAC/tccg`

AICES

Conclusion and Future Work

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
- TCCG is available at `https://github.com/HPAC/tccg`

Future Work

- Implement TC library based on GETT
- Parallelize GETT

AICES