GEMM3: Constant-workspace high-performance multiplication of three matrices for matrix chaining

Krzysztof A. Drewniak

The University of Texas at Austin

April 13, 2018

Matrix chaining problem

- Problem: compute $A_1 A_2 \cdots A_n$ efficiently, A_i matrices
- Where do the parentheses go?
- $O(n \log n)$ algorithm, also $O(n^3)$ with dynamic programming
- Fewer flops $\rightarrow$ more performance?

Generalized matrix chaining

- In reality — transposes, inverses, properties
- Ex:
  Ensemble Kalman filter $X_i^b S_i (Y_i^b)^T R_i^{-1}$
  Tridiagonalization $\tau_u \tau_v v v^T A u u^T$
  Two-sided triangular solve $L^{-1} A L^{-H}$ (L lower triangular)
- Performance with BLAS/LAPACK – must be expert
- Less performance with Matlab, numpy, etc. (left-to-right)
- Linnea: expression $\rightarrow$ BLAS calls automagically

GEMM3 — Why bother?

- Examples again:
  - $\boldsymbol{X_i^b S_i (Y_i^b)^T} R_i^{-1}$
  - $\tau_u \tau_v \boldsymbol{vv^T A uu^T}$
  - $\boldsymbol{L^{-1} A (L^{-1})^H}$ (L lower triangular)
- All multiply three matrices as a subproblem
- (Notation: $G \mathrel{+}= DEF$ and GEMM3)

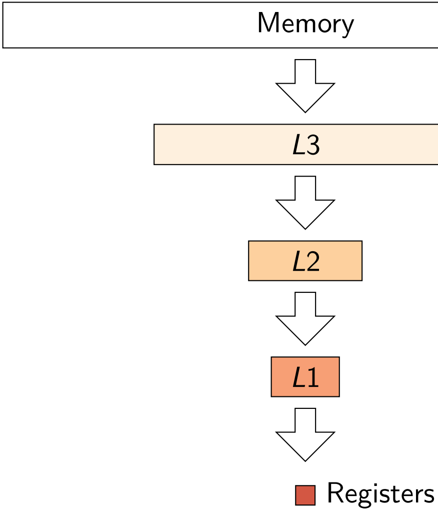
GEMM3 — Why a new algorithm?

- Current approach: parentheses, multiply twice, store temporary T
- T often eats memory
- Writing/reading T can hit your performance
- We can do better!
- Use how GEMM works to nest computations
- $O(1)$ extra memory, maybe more performance

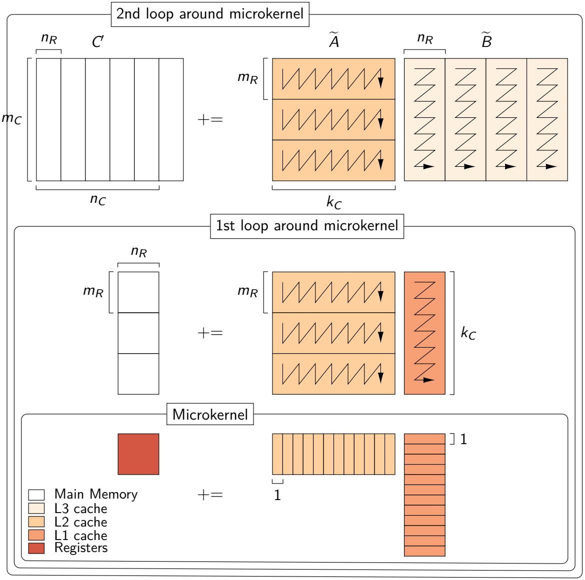
Section 2

High-Performance GEMM

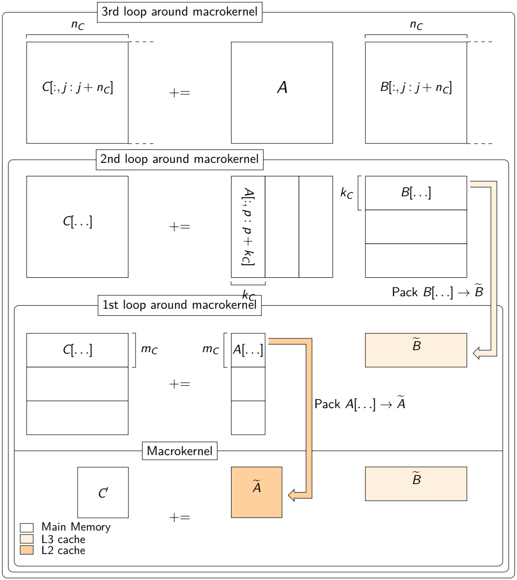
Memory hierarchy



GEMM: The kernels



GEMM: The algorithm



Data reuse

- Every loop reads *something* repeatedly
- Relevant things: packed blocks — making them takes time
- Packed block reuse problems:
  - m small — low time between remakes of $\widetilde{B}$
  - n small — same for $\widetilde{A}$
  - k tiny — microkernel doesn't do much, small caches

Key concept of the algorithm

- We want $G \mathrel{+}= DEF$, (dimensions: m, k, l, n in order)
- EF first needed in packing step
- Compute a block then
- Have GEMM algorithm, but

Deriving GEMM3: Partitionings

$G \mathrel{+}= D(EF)$ with BLIS, (EF) virtual.

1. Partition n dimension by n_C
   Limits rows of (EF), F, G
2. Partition k dimension by k_C
   Limits columns of D, (EF); rows of E

- Block of EF is $k_C \times n_C$.
- Now needed for $\tilde{E}F$

Deriving GEMM3: Inner algorithm

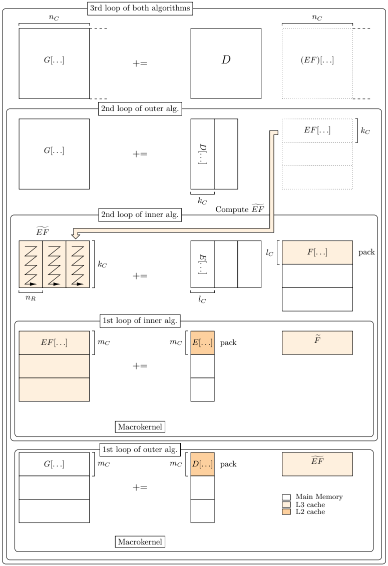
- Problem size: $k_C \times l \cdot l \times n_C$.
- Panel-matrix — has good performance with BLIS
- $k_C \times n_C$ output
- Only point to compute in constant memory
- GEMM algorithm needs tweaks

Deriving GEMM3: The tricky bits

Problem	Solution
Redundant loop over n ($n \leq n_C$)	Remove it
Packing output wastes space/time	Tweak microkernel params
$\widetilde{F}$ fights $\widetilde{EF}$ in $L3$	Halve n_C
Low $\widetilde{F}$ reuse	Low impact in practice
$m_R \nmid k_C$, leaving fringe	Shrink k_C slightly

Table: Tweaks needed to make GEMM fusion work

The algorithm



$G \mathrel{+}= (DE)F$

- Putting parentheses there sometimes better
- Deriving directly doesn't work — bad shape
- However, $G \mathrel{+}= (DE)F \Leftrightarrow G^T \mathrel{+}= F^T(E^T D^T)$

Section 4

Experiments and Results

Implementation details

- Multilevel Optimization of Matrix Multiply Sandbox (MOMMS)
- Extended to support three matrices
- Implement both GEMM3 and BLIS algorithm
- BLIS algorithm port performs like BLIS
- Experiments on Haswell machine from UT lab

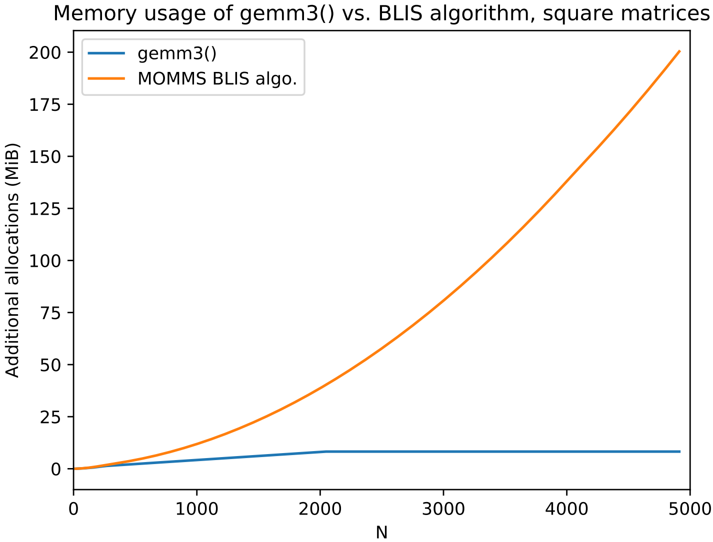
	GEMM3	BLIS algorithm
m_C	72	72
k_C	252	256
l_C	256	
n_C	2040	4080

Table: Constants for Haswell CPUs

Experiments

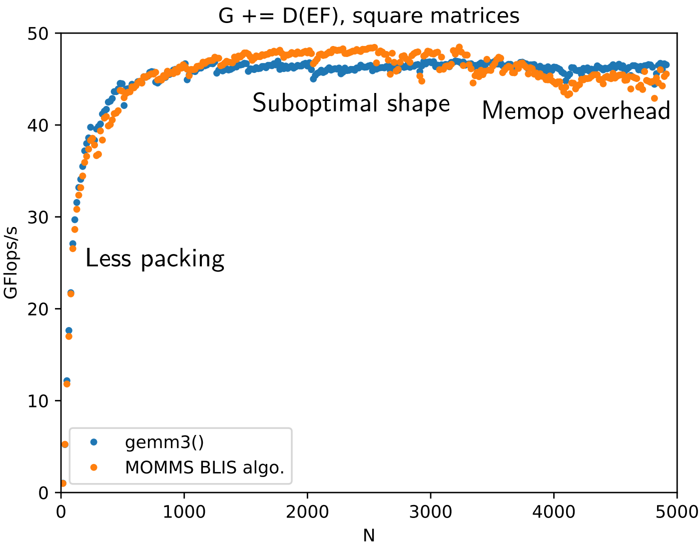
1. $G \mathrel{+}= D(EF)$, square matrices
   - Inputs column-major, outputs row-major for fairness
2. $G^T \mathrel{+}= F^T(E^T D^T)$, square matrices
   - After transpose, all row major
3. $G \mathrel{+}= D(EF)$, rectangles (one dimension small)

Workspace usage, square matrices

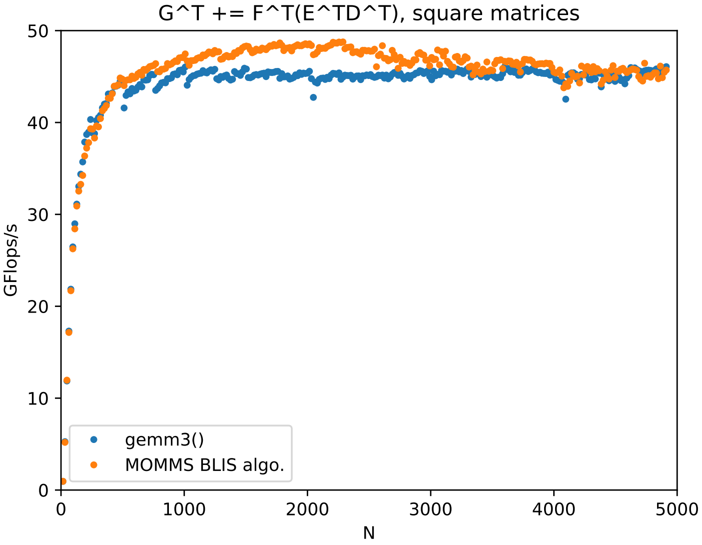


$G += D(EF)$, square matrices

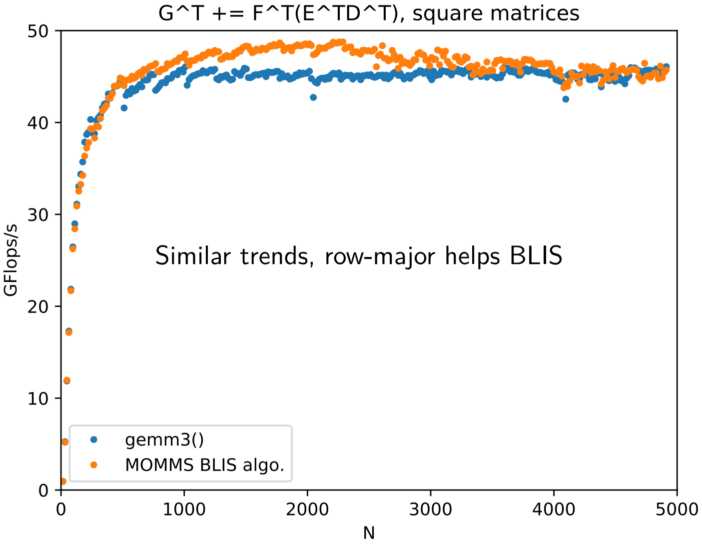
$G += D(EF)$, square matrices



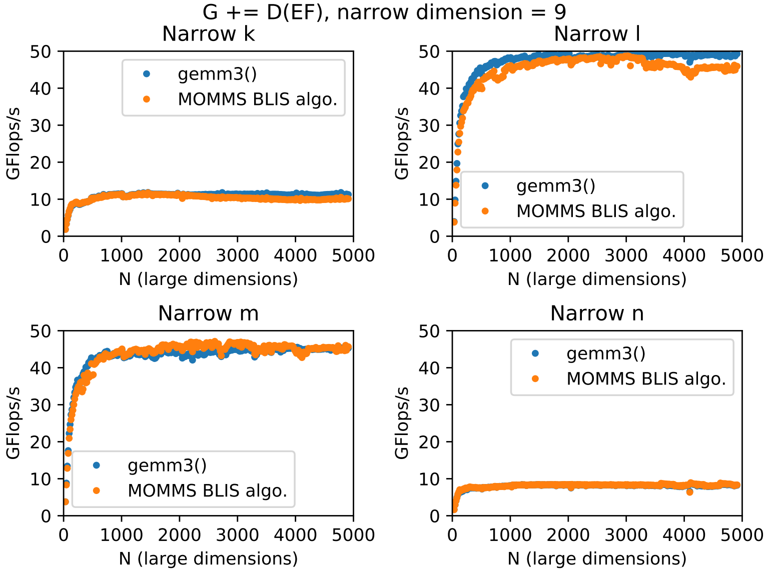
$G \mathrel{+}= (DE)F$, square matrices



$G \mathrel{+}= (DE)F$, square matrices



$G += D(EF)$, rectangular matrices



Acknowledgments

- Prof. Robert van de Geijn, for advising and providing the inspiration for this work
- Dr. Tyler Smith, for writing MOMMS and helping with algorithm design
- Prof. Tze Meng Low, for performance and paper-writing advice
- NSF grant **TODO NNNNNNN** for funding

Questions?

Picking constants: m_R, n_R

- Determine microkernel
- Based on microarchitecture — register width, FMA properties
- We're reusing BLIS's work
- Can swap m_R and n_R

Picking constants: k_C

Placing memory in cache: [tag][set #][offset in line]

$$m_r k_c S_{elem} = C_A C_{L1} N_{L1} \qquad n_r k_C S_{elem} = C_B C_{L1} N_{L1}$$

L1 Cache:

$$\begin{matrix} A & A & B & B \\ A & B & B & C \end{matrix} \quad \cdots$$

$$C_A + C_B + 1 \leq W_{L1}$$

Maximizing k_C improves performance

$$C_B = \left\lceil \frac{n_R k_C S_{elem}}{N_{L1} C_{L1}} \right\rceil$$

$$= \left\lceil \frac{n_R}{m_R} C_A \right\rceil$$

$$C_A \leq \left\lfloor \frac{W_{L1} - 1}{1 + \frac{n_R}{m_R}} \right\rfloor$$

Picking constants: m_C and n_C

- For m_C: reserve ways for B and C
- Then take all you can
- n_C, leave out what architecture requires, then divide
- L3 is very big, tuning is much less needed