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
- $O(N \log N)$ algorithm[1], also $O(N^3)$ with dynamic programming[2]
- Fewer flops $\rightarrow$ more performance?
- Ex: $WXYZ \Rightarrow ((WX)Y)Z$, $(WX)(YZ)$, $W(X(YZ))$, $\ldots$

[1] Hu and Shing, 1984
[2] Barthels 2018

Generalized matrix chaining

- In reality — transposes, inverses, properties
- Ensemble Kalman filter[3] $X_i^b S_i (Y_i^b)^T R_i^{-1}$
  Tridiagonalization[4] $\tau_u \tau_v v v^T A u u^T$
  Two-sided triangular solve[5] $L^{-1} A L^{-H}$ (L lower triangular)
- Performance with BLAS/LAPACK[6] – must be expert
- Less performance with Matlab, numpy, etc. (left-to-right)
- Linnea[7]: expression $\rightarrow$ BLAS calls automagically

[3] Rao 2017
[4] Choi 1995
[5] Poulson 2011
[6] Dongarra 1990, Anderson 1999
[7] Barthels 2018

GEMM3 — Why bother?

-
  - $\boldsymbol{X_i^b S_i (Y_i^b)^T} R_i^{-1}$
  - $\tau_u \tau_v \boldsymbol{v v^T A u u^T}$
  - $\boldsymbol{L^{-1} A (L^{-1})^H}$ (L lower triangular)
- All multiply three matrices as a subproblem
- Not all problems subdivide like this

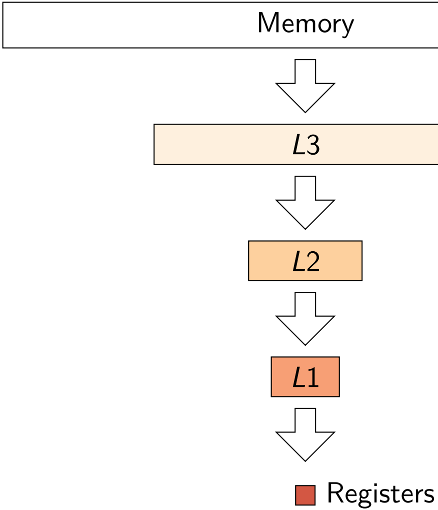
GEMM3 — Why a new algorithm?

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

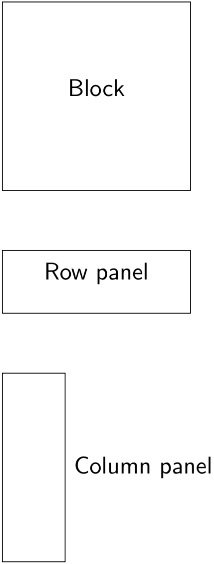
Section 2

High-Performance GEMM

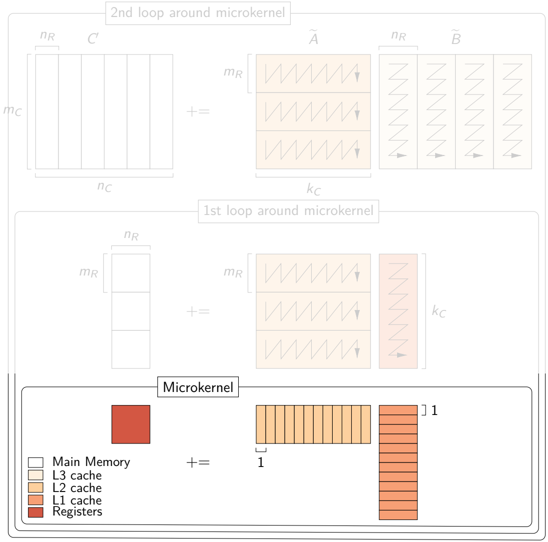
Memory hierarchy



Important matrix shapes

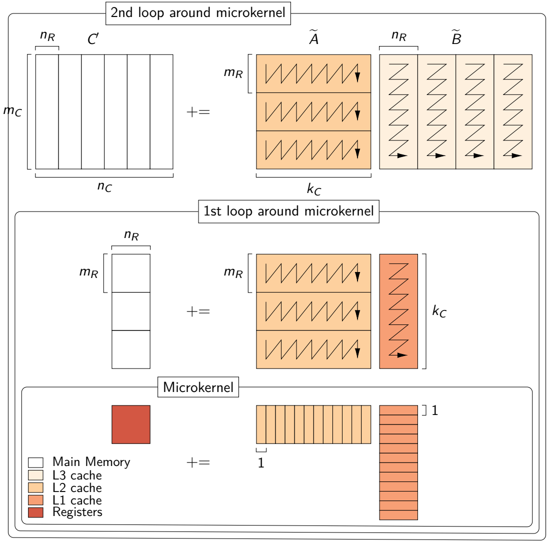


GEMM: The kernels



GEMM: The kernels

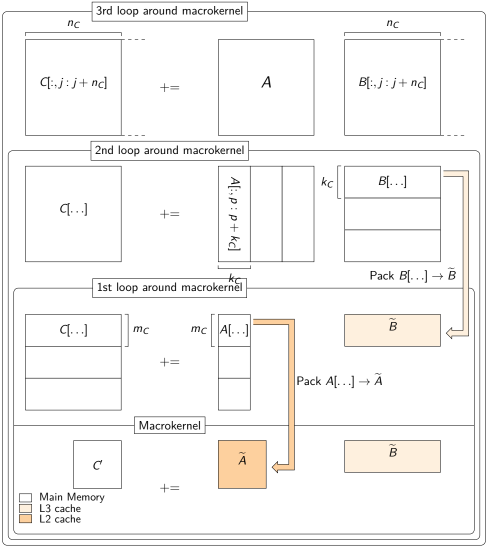
Macrokernel



Packing very important[8]

[8] Henry 1992

GEMM: The algorithm[9]

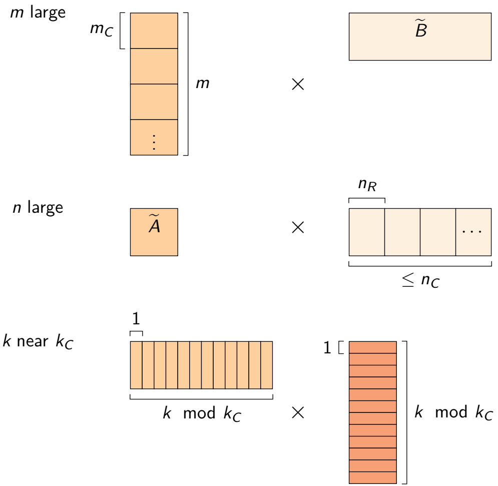


[9]Goto 2008

Data reuse

- Every loop reads *something* repeatedly

Want:

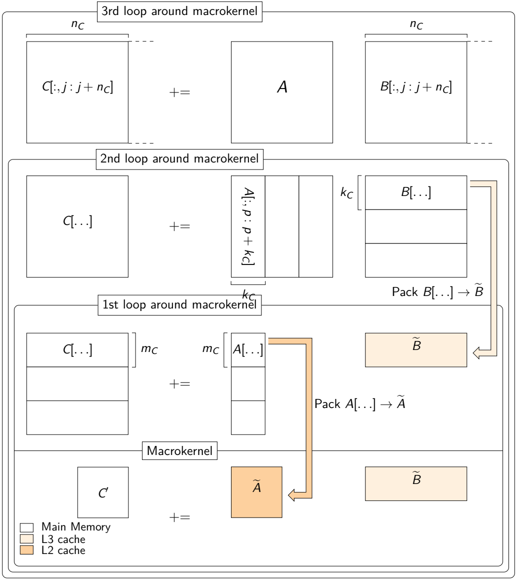


Section 3

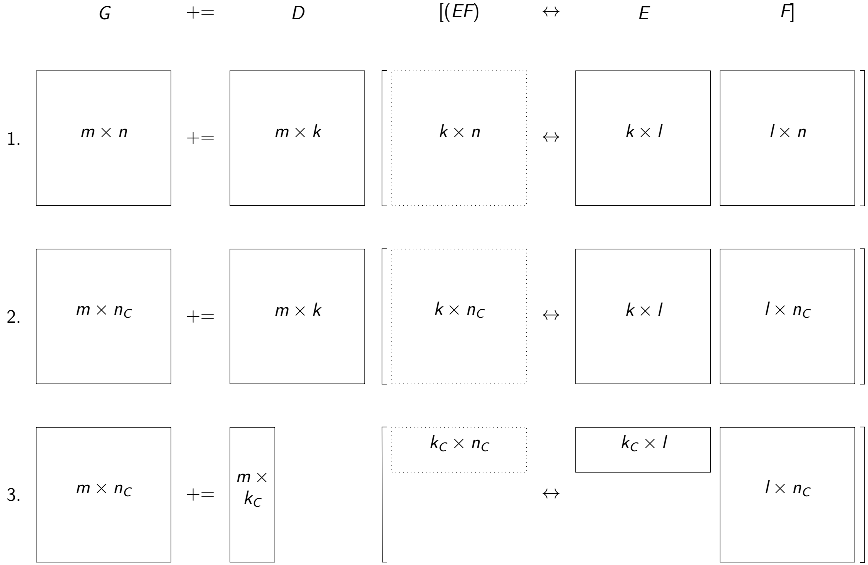
The GEMM3 algorithm

Key concept of the algorithm

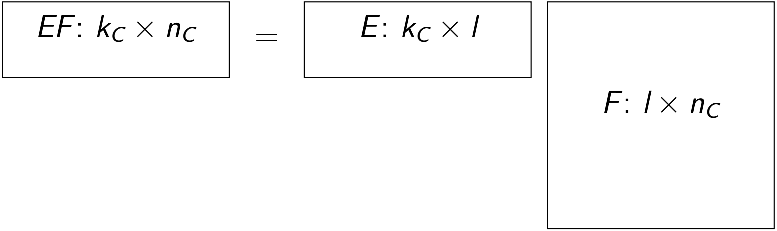
- We want $G \mathrel{+}= D(EF)$, (dimensions: m, k, l, n in order)
- EF first needed in packing step
- Don't do computation until then


3rd loop around macrokernel
n_C
$C[:, j : j + n_C]$
$+=$
A
n_C
$B[:, j : j + n_C]$
2nd loop around macrokernel
$C[\ldots]$
$+=$
$A[:, p : p + k_C]$
k_C
k_C
$B[\ldots]$
Pack $B[\ldots] \to \widetilde{B}$
1st loop around macrokernel
$C[\ldots]$
m_C
$+=$
m_C
$A[\ldots]$
$\widetilde{B}$
Pack $A[\ldots] \to \widetilde{A}$
Macrokernel
C'
$+=$
$\widetilde{A}$
$\widetilde{B}$
Main Memory
L3 cache
L2 cache

Deriving GEMM3: Partitionings

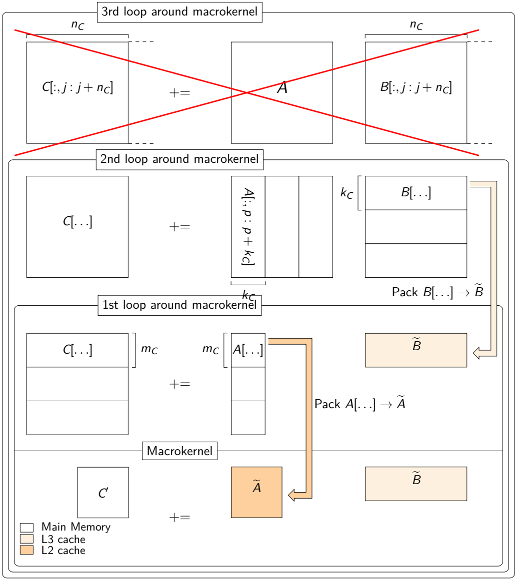


Deriving GEMM3: Inner algorithm

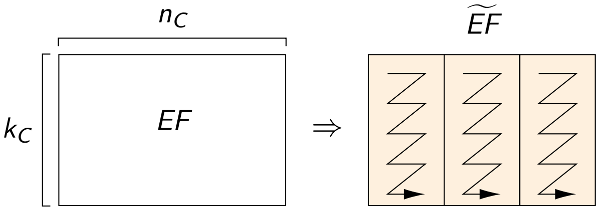


- Only point to compute EF in constant memory
- GEMM algorithm needs tweaks

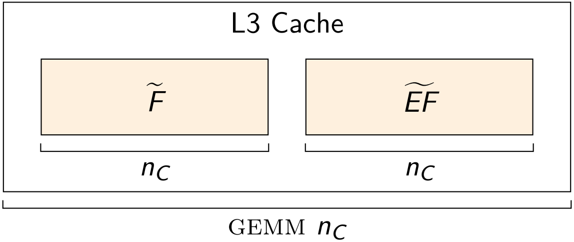
Inner algorithm tweaks: Removing the outer loop



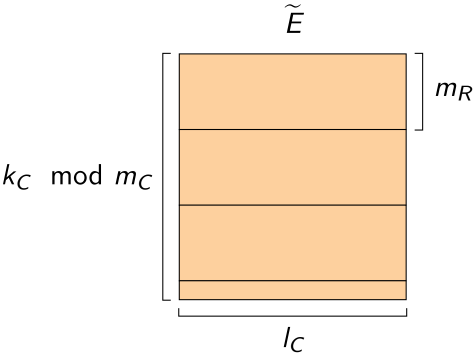
Inner algorithm tweaks: Microkernel packed writes



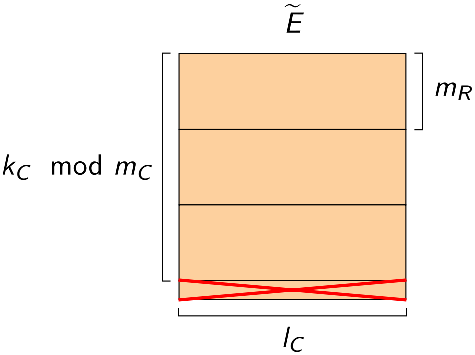
Inner algorithm tweaks: Halving n_C



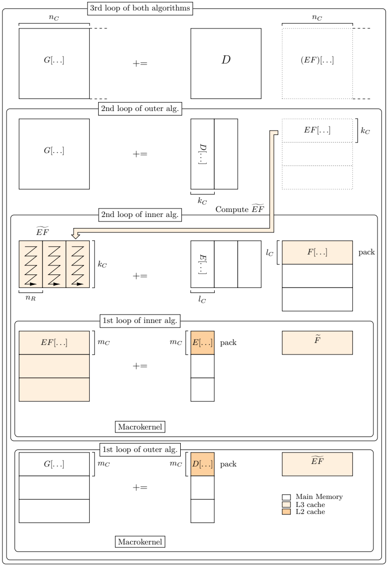
Inner algorithm tweaks: Small k_C reduction



Inner algorithm tweaks: Small k_C reduction

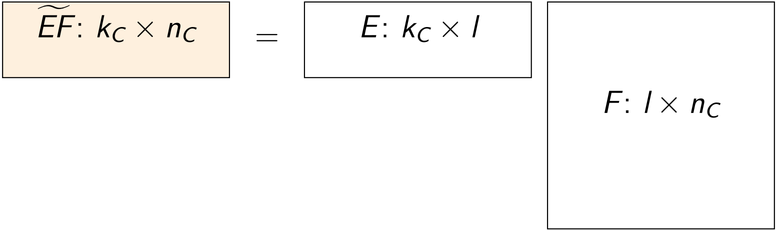


The algorithm

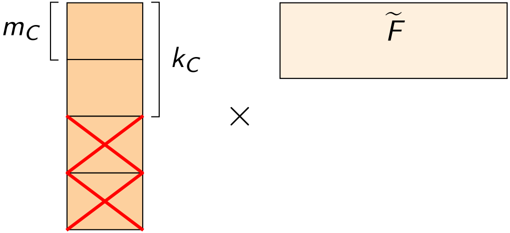


The small drawback

Problem shape:



Reuse problem: m small



Section 4

Experiments and Results

Implementation details

- Multilevel Optimization of Matrix Multiply Sandbox (MOMMS)[10]
- Extended to support three matrices
- Implement both GEMM3 and pair of GEMM algorithms
- GEMM (from BLIS[11]) port performs like BLIS
- Machine: 3.5 GHz (one core used), 15 GB RAM, 32 KB $L1$ cache, 256 KB $L2$, 8 MB $L3$. Peak perf 56 GFLOPS/s.

	GEMM3	GEMM algorithm
m_R	6	6
n_R	8	8
m_C	72	72
k_C	252	256
l_C	256	
n_C	2040	4080

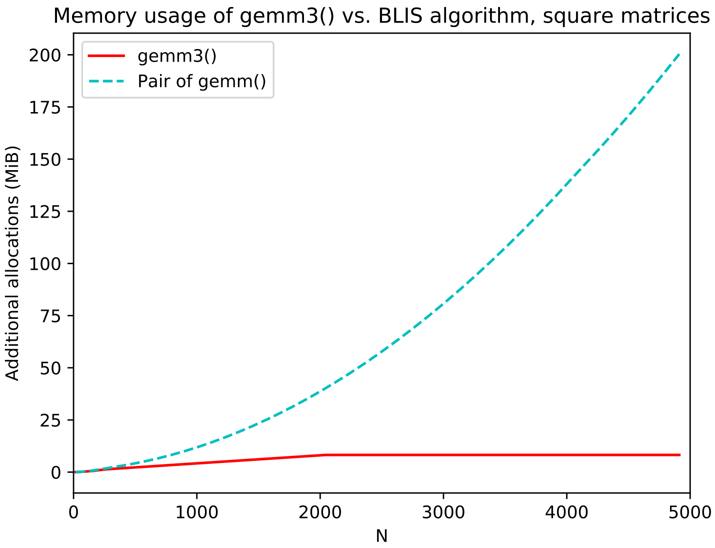
Table: Parameters for Haswell CPUs

[10] Smith 2018
[11] van Zee 2016

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



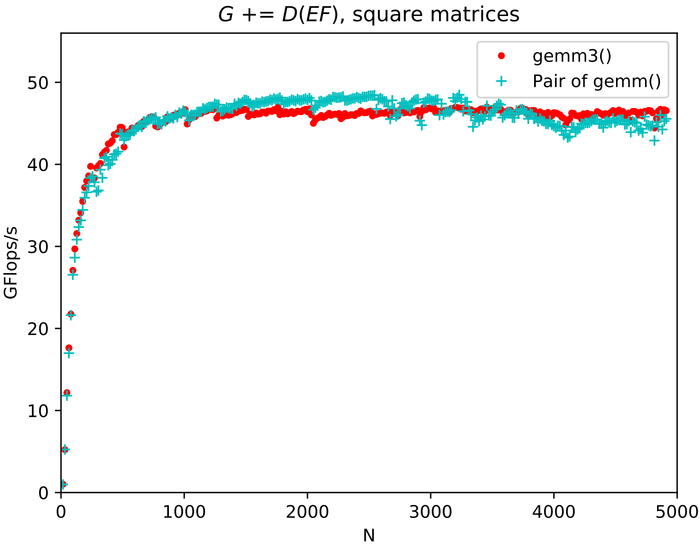
API simplicity

```
double *T = malloc(k * n * sizeof(double));
dgemm("N", "N", k, l, n,
      1, E, lde, F, ldf,
      0, T, k);
dgemm("N", "N", m, k, n,
      alpha, D, ldd, T, k,
      beta, G, ldg);
free(T);
```

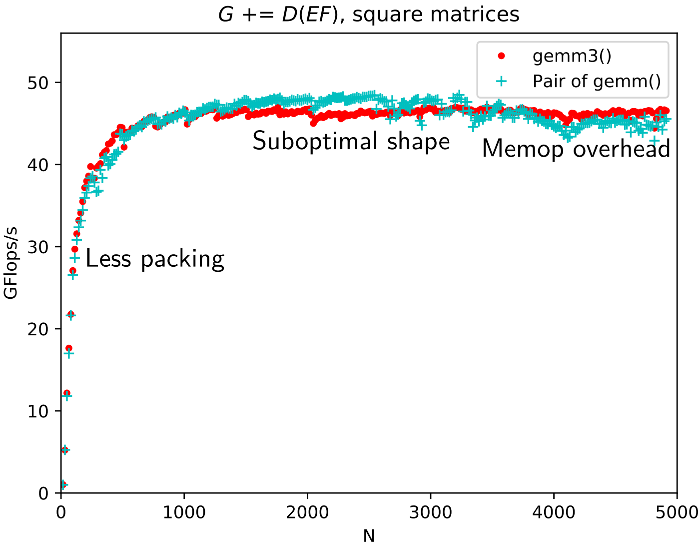
vs.

```
dgemm3("R", "N", "N", "N", m, k, l, n,
       alpha, D, ldd, E, lde, F, ldf,
       beta, G, ldg);
```

$G \mathrel{+}= D(EF)$, square matrices

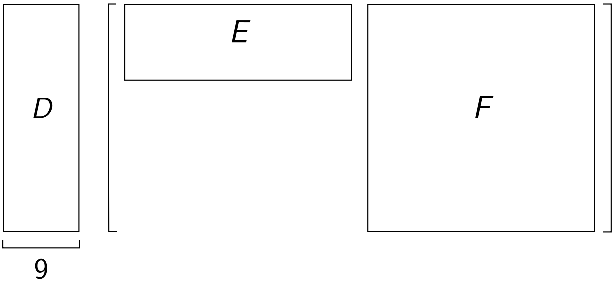


$G += D(EF)$, square matrices



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

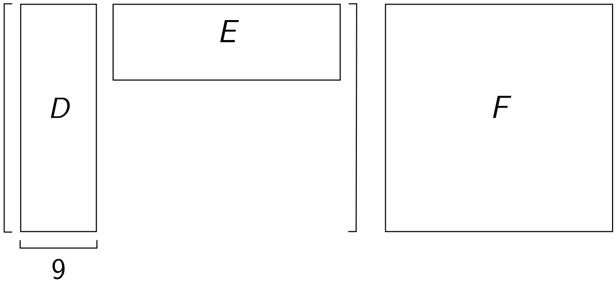
Ex: **Don't:**



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

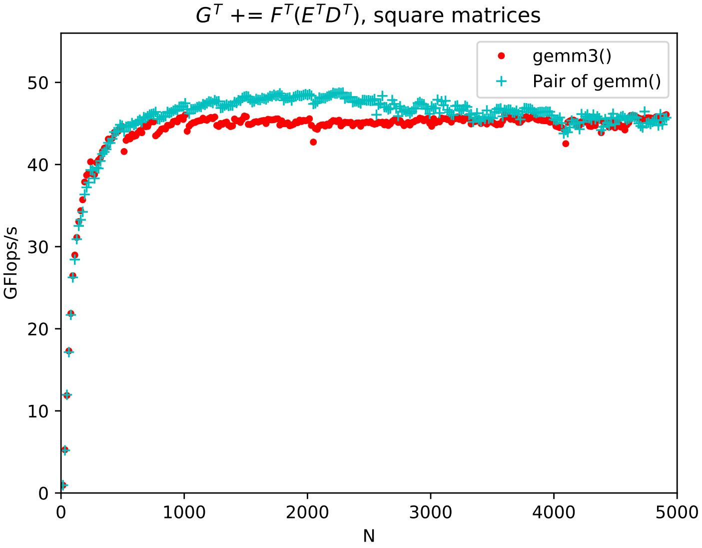
$G \mathrel{+}= (DE)F$

Ex: **Do:**

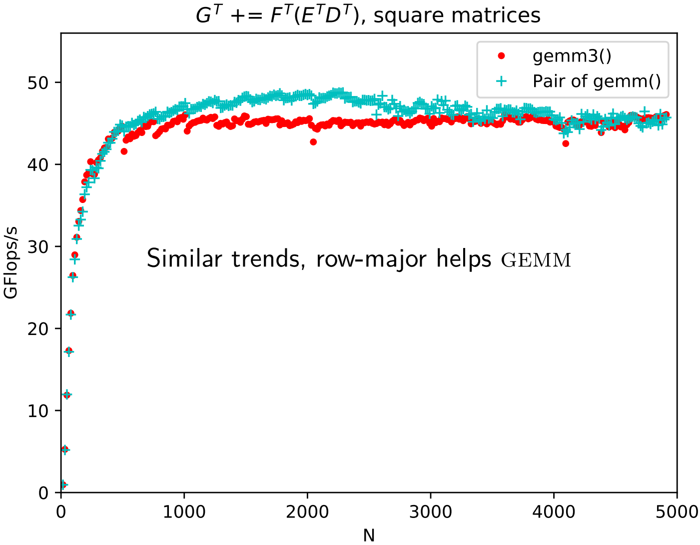


- Putting parentheses there sometimes better
- Deriving directly doesn't work — bad shape
- However, $G \mathrel{+}= (DE)F \Leftrightarrow G^T \mathrel{+}= F^T(E^T D^T)$

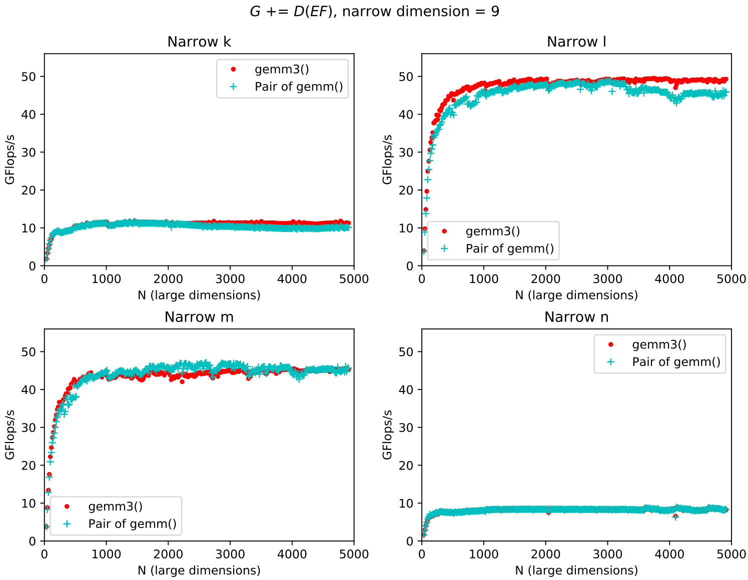
$G += (DE)F$, square matrices



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



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



Conclusions

- GEMM structure lets us make GEMM3
- Constant memory
- Cleaner API
- Comparable performance

Future Work

- Parallel case
- More architectures
- Variants (matrices with properties), autogeneration

Acknowledgments

- Prof. Robert van de Geijn — advising and providing inspiration
- Dr. Tyler Smith — writing MOMMS and algorithm design
- Prof. Tze Meng Low — performance fixes
- NSF awards CCF-1714091 and ACI-1550493 — funding

Questions?

Picking parameters: 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 parameters: 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

$$\begin{aligned} C_B &= \left\lceil \frac{n_R k_C S_{elem}}{N_{L1} C_{L1}} \right\rceil \\ &= \left\lceil \frac{n_R}{m_R} C_A \right\rceil \\ C_A &\leq \left\lfloor \frac{W_{L1} - 1}{1 + \frac{n_R}{m_R}} \right\rfloor \end{aligned}$$

Picking parameters: 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