Overview

Linear algebra software: the path to fast libraries, LAPACK and BLAS

Blocking (BLAS 3): key to performance

Fast MMM
- Algorithms
- ATLAS
- Model-based ATLAS
Linear Algebra Algorithms: Examples

Solving systems of linear equations
Eigenvalue problems
Singular value decomposition
LU/Cholesky/QR/... decompositions
... and many others

Make up much of the numerical computation across disciplines (sciences, computer science, data science and machine learning, engineering)
Efficient software is extremely relevant

The Path to Fast Libraries

**EISPACK** and **LINPACK** (early 1970s)
- Focus on dense matrices
- Jack Dongarra, Jim Bunch, Cleve Moler, Gilbert Stewart
- LINPACK still the name of the benchmark for the [TOP500 Wiki](https://en.wikipedia.org/wiki/TOP500) list of most powerful supercomputers

Matlab: Invented in the late 1970s by Cleve Moler
Commercialized (MathWorks) in 1984
Motivation: Make LINPACK, EISPACK easy to use
Matlab uses linear algebra libraries but can only call it if you operate with matrices and vectors and do not write your own loops
- A*B (calls MMM routine)
- A\b (calls linear system solver)
The Path to Fast Libraries

EISPACK/LINPACK Problem:
- Implementation vector-based = low operational intensity (e.g., MMM as double loop over scalar products of vectors)
- Low performance on computers with deep memory hierarchy (became apparent in the 80s)

LAPACK (late 1980s, early 1990s)
- Redesign all algorithms to be “block-based” to increase locality
- Jack Dongarra (Turing award 2021), Jim Demmel et al.

Two-layer architecture

- **LAPACK**
  - static higher-level functions
- **BLAS**
  - kernel functions implemented for each computer

Basic Linear Algebra Subroutines (BLAS)
- BLAS 1: vector-vector operations (e.g., vector sum)
- BLAS 2: matrix-vector operations (e.g., matrix-vector product)
- BLAS 3: matrix-matrix operations (e.g., MMM)

LAPACK uses BLAS 3 as much as possible

Fast MMM → fast BLAS3 (since all MMM related)
Reminder: Why is BLAS3 so important?

Using BLAS 3 (instead of BLAS 1 or 2) in LAPACK
= blocking
= high operational intensity $I$
= high performance

Remember (blocking MMM):

$I(n) =$

\[
\begin{bmatrix}
1 & 1 \\
1 & 1
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
1
\end{bmatrix}
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\]

$O(1)$

\[
\begin{bmatrix}
1 & 1 \\
1 & 1
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\]

$O(\sqrt{C})$
Small Detour: MMM Complexity?

Usually computed as $C = AB + C$

Cost as computed before
- $n^3$ multiplications + $n^3$ additions = $2n^3$ floating point operations
- = $O(n^3)$ runtime

Blocking
- Increases locality
- Does not decrease cost

Can we reduce the op count?

Strassen’s Algorithm

Strassen, V. "Gaussian Elimination is Not Optimal," *Numerische Mathematik* 13, 354-356, 1969

*Until then, MMM was thought to be $\Theta(n^3)$*

Recurrence for flops:
- $T(n) = 7T(n/2) + 9/2 n^2 = 7n\log_2(7) - 6n^2 = O(n^{2.808})$
- Later improved: $9/2 \rightarrow 15/4$

Fewer ops from $n = 654$, but ...
- Structure more complex → runtime crossover much later
- Numerical stability inferior

Can we reduce more?
MMM Complexity: What is known


Makes MMM $O(n^{2.376...})$

Current best (Oct. 2022): $O(n^{2.37188...})$ Previous best: $O(n^{2.3728596...})$

But unpractical (a galactic algorithm)

MMM is obviously $\Omega(n^2)$

It could well be close to $\Theta(n^2)$

Practically all code out there uses $2n^3$ flops

Compare this to matrix-vector multiplication:
- Known to be $\Theta(n^2)$ (Winograd), i.e., boring

The Path to Fast Libraries (continued)

LAPACK \rightarrow BLAS \rightarrow static higher-level functions

LAPACK \rightarrow BLAS \rightarrow kernel functions implemented for each computer

ATLAS (late 1990s, inspired by PhiPAC): BLAS generator

Enumerates many implementation variants (blocking etc.) and picks the fastest (example): advent of so-called autotuning

Enables automatic performance porting

Most important: BLAS3 MMM generator
ATLAS Architecture

Hardware parameters:
- L1Size: size of L1 data cache
- NR: number of registers
- MulAdd: fused multiply-add available?
- L*: latency of FP multiplication

Search parameters:
- for example blocking sizes
- span search space
- specify code
- found by orthogonal line search

source: Pingali, Yotov, et al., Cornell U.

ATLAS

Model-Based ATLAS (2005)

- Search for parameters replaced by model to compute them
- Much faster + provides understanding of what parameters are found

source: Pingali, Yotov, et al., Cornell U.
Optimizing MMM

References:


Our presentation is based on this paper

0: Starting Point

Standard triple loop

```matlab
// Computes c = c + ab
for i = 0:N-1
    for j = 0:M-1
        for k = 0:K-1
            c_ij = c_ij + a_ik*b_kj
```

Matlab-style code notation

Most important in practice (based on usage in LAPACK)

- Two out of N, M, K are small
- One out of N, M, K is small
- None is small (e.g., square matrices)
1: Loop Order

```c
// Computes C = C + AB
for i = 0:N-1
  for j = 0:M-1
    for k = 0:K-1
      c_ij = c_ij + a_ik*b_kj
```

```latex
\begin{equation}
\begin{array}{c}
\text{ij,k loops can be permuted in any order!}
\end{array}
\end{equation}
```

- $i$-$j$-$k$: $B$ is reused, good if $M < N$ ($B$ is smaller than $A$)
- $j$-$i$-$k$: $A$ is reused, good if $N < M$
- Other options are inferior, e.g., $k$-$i$-$j$:

```latex
\begin{equation}
\begin{array}{c}
\text{Poor temporal locality w.r.t. } C
\end{array}
\end{equation}
```

2: Blocking for Cache

```latex
\begin{equation}
\begin{array}{c}
\text{Results in six-fold loop}
\end{array}
\end{equation}
```

Formally obtained through loop-tiling and loop exchange

```c
\begin{tabular}{l}
\textbf{How to find the best }N_B^2\textbf{?} \\
\textbf{ATLAS: uses search over all }N_B^2 \leq \min(C, 80^2)\quad (C = \text{measured cache size})
\end{tabular}
```

*Model: explained next, uses $C_1 = \text{measured L1 cache size}$*
2: Blocking for Cache

a) Idea: Working set has to fit into cache
   Easy estimate: \(|\text{working set}| = 3 N_b^2\)
   Model: \(3 N_b^2 \leq C_1\)

b) Closer analysis of working set:
   \[ N_b^2 + N_B + 1 \leq C_1 \]
   all of b \( \uparrow \)
   row of a \( \uparrow \)
   element of c

c) Take into account cache block size \(B_1\):
   \[ \left\lfloor \frac{N_b^2}{B_1} \right\rfloor + \left\lfloor \frac{N_B}{B_1} \right\rfloor + 1 \leq \frac{C_1}{B_1} \]

2: Blocking for Cache

d) Take into account LRU replacement
   Build a history of accessed elements

   \(i=0:\)
   \[ b_{0,0} b_{0,1} a_{0,0} b_{0,1,1} \ldots a_{0,N_B-1} b_{N_B-1,0} c_{0,0} \]  (j=0)
   \[ b_{0,0} b_{0,1} a_{0,0} b_{0,1,1} \ldots a_{0,N_B-1} b_{N_B-1,1} c_{0,1} \]  (j=1)
   \[ \vdots \]
   \[ b_{0,0} b_{0,N_B-1} a_{0,0} b_{1,1,1} \ldots a_{0,N_B-1} b_{N_B-1,N_B-1} c_{0,N_B-1} \]  (j=N_b−1)

   Corresponding history:
   \[ b_{0,0} b_{1,0} \ldots b_{N_B-1,0} c_{0,0} \]
   \[ b_{0,1} b_{1,1} \ldots b_{N_B-1,1} c_{0,1} \]
   \[ \vdots \]
   \[ a_{0,0} b_{0,N_B-1} b_{1,1} b_{1,1} \ldots a_{0,N_B-1} b_{N_B-1,N_B-1} b_{N_B-1,N_B-1} c_{0,N_B-1} \]

   Observations:
   - All of b has to fit for next iteration \((i = 1)\)
   - When \(i = 1\), row 1 of a will not cleanly replace row 0 of a
   - When \(i = 1\), elements of c will not cleanly replace previous elements of c
2: Blocking for Cache

d) Take into account LRU replacement

History (i = 0):

\[
\begin{align*}
&b_{0,0} b_{1,0} \ldots b_{N_B-1,0} c_{0,0} \\
&b_{0,1} b_{1,1} \ldots b_{N_B-1,1} c_{0,1} \\
&\vdots \\
&a_{0,0} b_{0,N_B-1} a_{0,1} b_{1,N_B-1} \ldots a_{0,N_B-1} b_{N_B-1,N_B-1} c_{0,N_B-1}
\end{align*}
\]

Observations:
- All of \( b \) has to fit for next iteration (\( i = 1 \))
- When \( i = 1 \), row 1 of \( a \) will not cleanly replace row 0 of \( a \)
- When \( i = 1 \), elements of \( c \) will not cleanly replace previous elements of \( c \)

This has to fit:
- Entire \( b \)
- 2 rows of \( a \)
- 1 row of \( c \)
- 1 element of \( c \)

\[
\begin{align*}
\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq \frac{C_1}{B_1}
\end{align*}
\]

2: Blocking for Cache

e) Take into account blocking for registers (next optimization)

\[
\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B M_{L_C}}{B_1} \right\rceil + \left\lceil \frac{M_{L_C} N_{L_C}}{B_1} \right\rceil \leq \frac{C_1}{B_1}
\]
3: Blocking for Registers

Blocking mini-MMMs into micro-MMMs for registers revisits the question of loop order:

For fixed i, j: 2n operations
- n independent mults
- n dependent adds

For fixed k: 2n^3 operations
- n^2 independent mults
- n^2 independent adds

\[ \star = i-j-k: \]

\[ \star = k-i-j: \]

Better ILP
(but larger working set)

Result: k-i-j loop order for micro-MMMs

---

3: Blocking for Registers

For i = 0:N
for j = 0:M
for k = 0:K
for i' = i:M
for j' = j:N
for k' = k:K
for k'' = k':k'+K
for i'' = i':i'+M
for j'' = j':j'+N
\[ c_{i''j''} = c_{i''j''} + a_i^k b_k^l \]

mini-MMM

\[ M_U \]

micro-MMM

\[ N_U \]

\[ N_R \]

\[ K_U \]

\[ \text{How to find the best } M_U, N_U, K_U? \]

ATLAS: uses search with bound

\[ M_U + N_U + M_U N_U \leq N_R \]

number of registers

size of working set in \( \text{(all the red parts on the left)} \)

Model: Use largest \( M_U, N_U \) that satisfy this equation and \( M_U = N_U \)
4: Basic Block Optimizations

Unroll micro-MMMs
Scalar replacement
Loads from \( c \) (\( M_U N_U \) many) at 1
Loads from \( a \) and \( b \) (\( M_U + N_U \) many) at 2
Requires \( M_U + N_U + M_U N_U \) scalar variables (all the red parts on the right)

Example of ATLAS-generated code

5: Other optimizations (see paper)

Skewing: separate dependent add-mults for better ILP

Software pipelining: move load from one iteration to previous iteration to hide load latency (a form of prefetching)

Buffering to avoid TLB misses (later)
Remaining Details

Register renaming and the refined model for x86

TLB-related optimizations

Dependencies

Read-after-write (RAW) or true dependency

\[
\begin{align*}
W & \quad r_1 = r_3 + r_4 \\
R & \quad r_2 = 2r_1
\end{align*}
\]

nothing can be done
no ILP

Write after read (WAR) or antidependency

\[
\begin{align*}
R & \quad r_1 = r_2 + r_3 \\
W & \quad r_2 = r_4 + r_5
\end{align*}
\]

dependency only by name → rename

\[
\begin{align*}
\begin{array}{c}
W \quad r_1 = r_2 + r_3 \\
& \ldots \\
W \quad r_1 = r_4 + r_5
\end{array}
\end{align*}
\]

dependency only by name → rename

\[
\begin{align*}
\begin{array}{c}
W \quad r_1 = r_2 + r_3 \\
& \ldots \\
W \quad r_1 = r_4 + r_5
\end{array}
\end{align*}
\]

dependency only by name → rename

\[
\begin{align*}
W & \quad r_1 = r_2 + r_3 \\
& \ldots \\
W & \quad r_1 = r_4 + r_5
\end{align*}
\]

now ILP
Resolving WAR by Renaming

Renaming can be done at three levels:

1. C source code (= you rename): use SSA style (next slide)

Scalar Replacement + SSA

How to avoid WAR and WAW in your basic block source code

Solution: Single static assignment (SSA) code:

- Each variable is assigned exactly once

```plaintext
<more>
s266 = (t287 - t285);
s267 = (t282 + t286);
s268 = (t282 - t286);
s269 = (t284 + t288);
s270 = (t284 - t288);
s271 = (0.5*(t271 + t280));
s272 = (0.5*(t271 - t280));
s273 = (0.5*((t281 + t283) - (t285 + t287)));
s274 = (0.5*(s265 - s266));
t289 = ((9.0*s272) + (5.4*s273));
t290 = ((5.4*s272) + (12.6*s273));
t291 = ((1.8*s271) + (1.2*s274));
t292 = ((1.2*s271) + (2.4*s274));
a122 = (1.8*(t269 - t278));
a123 = (1.8*s267);
a124 = (1.8*s269);
t293 = ((a122 - a123) + a124);
a125 = (1.8*(t267 - t276));
t294 = (a125 + a123 + a124);
t295 = ((a125 + a123) + (3.6*s267));
t296 = (a122 + a125 + (3.6*s269));
<more>
```
Resolving WAR by Renaming

Renaming can be done at three levels:

1. C source code (= you rename)
2. Compiler: Uses a different register upon register allocation, \( r = r_6 \)
3. Hardware (if supported): dynamic register renaming
   - Requires a separation of architectural and physical registers
   - Requires more physical than architectural registers

Register Renaming

Each logical register has several associated physical registers

Hardware manages mapping architectural \( \rightarrow \) physical registers

Hence: more instances of each \( r_i \) can be created

Used in superscalar architectures (e.g., Intel Core) to increase ILP by dynamically resolving WAR/WAW dependencies
Micro-MMM Standard Model

\[ M_U N_U + M_U + N_U \leq N_R - \text{ceil}(L_x+1)/2 \]

Core \((N_R = 16): M_U = 2, N_U = 3\)

```
  a     b     c
  \bullet
```

\text{reuse in } a, b, c

Code sketch \((K_U = 1)\)

```
rc1 = c[0,0], ... , rc6 = c[1,2] // 6 registers
loop over k {
    load a // 2 registers
    load b // 3 registers
    compute // 6 independent mults, 6 independent adds, reuse a and b
    c[0,0] = rc1, ... , c[1,2] = rc6
}
```

But on x86 that's not what ATLAS' search found

Extended Model (x86)

Set \(M_U = 1, N_U = N_R - 2 = 14\)

```
  a     b     c
  \bullet
```

\text{reuse in } c only

Code sketch \((K_U = 1)\)

```
rc1 = c[0], ... , rc14 = c[13] // 14 registers
loop over k {
    load a // 1 register
    rb = b[1] // 1 register
    rc1 = rc1 + rb // add (two-operand)
    rb = b[2] // reuse register (WAR: register renaming resolves it)
    rb = rb*a
    rc2 = rc2 + rb
    ...
    c[0] = rc1, ... , c[13] = rc14
}
```

\text{Summary:}
- no reuse in a and b
+ larger tile size available for c since for b only one register is used
Visualization of What Seems to Happen

\[ \begin{align*}
2 & \quad \bullet \quad 3 = 2 \times 3 \\
\text{reuse in } a, b, c
\end{align*} \]

\[ \begin{align*}
1 & \quad \bullet \quad 14 = 14 \\
\text{reuse in } c \text{ only}
\end{align*} \]

Experiments

**Unleashed:** Not generated = handwritten contributed mini-MMM code

**Refined model** for computing register tiles on x86

Blocking by model is for L1 cache

**Result:** Model-based is comparable to search-based (except Itanium)

Search optimized for L2 cache (without knowing), the model blocks for L1 cache which here cannot store floats

*ATLAS generated (Code generation with search) is the baseline*  
*graph: Pingali, Yotov, Cornell U.*
Remaining Details

Register renaming and the refined model for x86

TLB-related optimizations

Virtual Memory System (Core Family)

The processor works with *virtual addresses*

All caches work with *physical addresses*

Both address spaces are organized in pages

Page size: 4 KB (can be changed to 2 MB and even 1 GB in OS settings)

Address translation: virtual address → physical address
Virtual/Physical Addresses

Processor: virtual addresses  
Caches: physical addresses  
Page size = 4 KB

VPN: virtual page number  
PPN: physical page number

L1 cache lookup can start concurrently with address translation!

How would Intel (likely) increase the L1 cache size?

Address Translation

Uses a cache called translation lookaside buffer (TLB)

Skylake:

- **Level 1**
  - ITLB (instructions): 128 entries
  - DTLB (data): 64 entries

- **Level 2**
  - Shared (STLB): 1536 entries

Miss Penalties:

- *DTLB hit*: no penalty
- *DTLB miss, STLB hit*: few cycles penalty
- *STLB miss*: can be very expensive
Impact on Performance

Repeatedly accessing a working set spread over too many pages yields TLB misses and can result in a significant slowdown.

Example Skylake: STLB = 1536

A computation that repeatedly accesses a working set of 2048 doubles spread over 2048 pages will cause STLB misses.

How much space will this working set occupy in cache (assume no conflicts)?

2048 * 64 B = 128 KB (fits into L2 cache)

Example MMM

We are looking for parts in the working set that are spread out in memory:

- Block row of a: contiguous
- All of b: contiguous
- Block of c: if M > 512, then spread over $N_B$ pages

Typically, $N_B$ is in the 10s, so no problem
Example MMM, contd.

Interface BLAS function: `dgemm(a, b, c, N, K, M, lda, ldb, ldc)`

Leading dimensions: Enable use on matrices inside matrices

Assume `lda, ldb, ldc > 512`:
- Block row of `a`: spread over ≥ `N_B` pages
- All of `b`: spread over ≥ `K` pages
- Block of `c`: spread over ≥ `N_B` pages

So copying to contiguous memory may pay off

Example MMM, contd.

Resulting code (sketch):

```c
// all of b reused: possible copy to contiguous memory
for i = 0:N_B:N-1
  // block row of a reused: possibly copy
  for j = 0:N_B:M-1
    // block of c reused: possibly copy
    for k = 0:N_B*K-1
      ....
```
Fast MMM: Principles

Optimization for memory hierarchy
- Blocking for cache
- Blocking for registers

Basic block optimizations
- Loop order for ILP
- Unrolling + scalar replacement
- Scheduling & software pipelining

Optimizations for virtual memory
- Buffering (copying spread-out data into contiguous memory)

Autotuning
- Search over parameters (ATLAS)
- Model to estimate parameters (Model-based ATLAS)

All high performance MMM libraries do some of these (but possibly in slightly different ways)

Path to Fast Libraries

<table>
<thead>
<tr>
<th>LAPACK</th>
<th>static higher level functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>kernel functions generated for each computer</td>
</tr>
</tbody>
</table>

The advent of SIMD vector instructions (SSE, 1999) made ATLAS obsolete

The advent of multicore systems (ca. 2005) required a redesign of LAPACK (just parallelizing BLAS is suboptimal)

Recently, BLAS interface needs to be extended to handle higher-order tensor operations (used in machine learning)

Automatic generation of blocked algorithms, alternatives to LAPACK (FLAME)

Small scale linear algebra requires quite different optimizations (see program generator SLinGen/LGen)
Lessons Learned

Implementing even a relatively simple function with optimal performance can be highly nontrivial

Autotuning can find solutions that a human would not think of implementing

Understanding which choices lead to the fastest code can be very difficult

MMM is a great case study, touches on many performance-relevant issues

Most domains are not studied as carefully as dense linear algebra