How to Write Fast Numerical Code

Spring 2019

*Lecture:* Dense linear algebra, LAPACK/BLAS, ATLAS, fast MMM

**Instructor:** Markus Püschel
**TA:** Tyler Smith, Gagandeep Singh, Alen Stojanov

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, engineering)
- Efficient software is extremely relevant

The Path to Fast Libraries

- EISPACK and LINPACK (early 1970s)
  - Jack Dongarra, Jim Bunch, Cleve Moler, Gilbert Stewart
  - LINPACK still the name of the benchmark for the TOP500 (Wiki) list of most powerful supercomputers

- 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)
The Path to Fast Libraries

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

Now there is implementation effort for each processor!

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) =

\[
\begin{align*}
I(n) &= \\
O(1) &= \\
O(\sqrt{C}) &= 
\end{align*}
\]
```
The Path to Fast Libraries

- Before we continue a little detour

Matlab

- Invented in the late 1970s by Cleve Moler
- Commercialized (MathWorks) in 1984
- Motivation: Make LINPACK, EISPACK easy to use
- Matlab uses LAPACK and other 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)
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 $\rightarrow$ runtime crossover much later
  - Numerical stability inferior
- Can we reduce more?
MMM Complexity: What is known

- Makes MMM is $O(n^{2.376})$
- Current best: $O(n^{2.373})$
- But unpractical

- 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)

- **ATLAS** (late 1990s, inspired by **PhiPAC**): BLAS generator
  - Enumerates many implementation variants (blocking etc.) and picks the fastest (**example**)
  - Enables automatic performance porting
  - Most important: BLAS3 MMM generator
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, Cornell U.

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, Cornell U.
Optimizing MMM

- Blackboard

References:


*Our presentation is based on this paper*

Remaining Details

- Register renaming and the refined model for x86
- TLB effects
Dependencies

- Read-after-write (RAW) or true dependency
  \[ W \begin{align*}
  r_1 &= r_3 + r_4 \\
  r_2 &= 2r_1 
  \end{align*} \]
  nothing can be done
  no ILP

- Write after read (WAR) or antidependency
  \[ R \begin{align*}
  r_1 &= r_2 + r_3 \\
  r_2 &= r_4 + r_5 
  \end{align*} \]
  dependency only by name → rename
  \[ R \begin{align*}
  r_1 &= r_2 + r_3 \\
  r &= r_4 + r_5 
  \end{align*} \]
  now ILP

- Write after write (WAW) or output dependency
  \[ W \begin{align*}
  r_1 &= r_2 + r_3 \\
  \vdots \\
  r_1 &= r_4 + r_5 
  \end{align*} \]
  dependency only by name → rename
  \[ W \begin{align*}
  r_1 &= r_2 + r_3 \\
  \vdots \\
  r &= r_4 + r_5 
  \end{align*} \]
  now ILP

Resolving WAR by Renaming

\[ R \begin{align*}
  r_1 &= r_2 + r_3 \\
  r_2 &= r_4 + r_5 
  \end{align*} \]
  dependency only by name → rename
  \[ R \begin{align*}
  r_1 &= r_2 + r_3 \\
  r &= r_4 + r_5 
  \end{align*} \]
  now ILP

Renaming can be done at three levels:
- 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
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));
ts268 = ((9.0*s272) + (5.4*s273));
t289 = ((5.4*s272) + (12.6*s273));
t291 = ((1.8*s271) + (1.2*s274));
t292 = ((1.7*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 - a122) + (3.6*s267));
t296 = (a122 + a125 + (3.6*s269));
```

Resolving WAR by Renaming

\[ R \quad r_1 = r_2 + r_3 \quad \text{dependency only by name \rightarrow rename} \quad r_1 = r_2 + r_3 \quad \text{now ILP} \]
\[ W \quad r_2 = r_4 + r_5 \]

Renaming can be done at three levels:
- C source code (= you rename): use SSA style (next slide)
- Compiler: Uses a different register upon register allocation, \( r = r_6 \)
- Hardware (if supported): dynamic register renaming
  - Requires a separation of architectural and physical registers
  - Requires more physical than architectural registers
Register Renaming

- Hardware manages mapping architectural \(\rightarrow\) physical registers
- Each logical register has several associated 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-MMMM Standard Model

- \(\text{MU} \times \text{NU} + \text{MU} + \text{NU} \leq \text{NR} - \text{ceil}((\text{Lx}+1)/2)\)
- Core (\(\text{NR} = 16\)): \(\text{MU} = 2\), \(\text{NU} = 3\)
  \[
  \begin{array}{c}
  a \\
  b \\
  c
  \end{array}
  \quad \text{reuse in } a, b, c
  
  - Code sketch (\(\text{KU} = 1\))

  ```c
  rc1 = c[0,0], ..., rc6 = c[1,2] // 6 registers
  loop over k {
    load a // 2 registers
    load b // 3 registers
    compute // 6 indep. mults, 6 indep. adds, reuse a and b
  }
  c[0,0] = rc1, ..., c[1,2] = rc6
  ```
Extended Model (x86)

- Set MU = 1, NU = NR – 2 = 14

\[ a \bullet b = c \quad \text{reuse in } c \]

- Code sketch (KU = 1)

\[
\begin{align*}
rc1 &= c[0], \ldots, rc14 = c[13] \quad \text{// 14 registers} \\
\text{loop over } k \{ & \\
& \text{load } a \quad \text{// 1 register} \\
& rb = b[1] \quad \text{// 1 register} \\
& rb = rb \ast a \quad \text{// mult (two-operand)} \\
& rc1 = rc1 + rb \quad \text{// add (two-operand)} \\
& \text{rb} = \text{rb} \ast a \\
& \text{rc2} = \text{rc2} + \text{rb} \\
& \ldots \\
& c[0] = rc1, \ldots, c[13] = rc14
\end{align*}
\]

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

- reuse in a, b, c
- reuse in c
Experiments

- **Unleashed**: Not generated = hand-written contributed code
- **Refined model** for computing register tiles on x86
- Blocking is for L1 cache

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

---

Remaining Details

- Register renaming and the refined model for x86
- TLB effects
  - Blackboard

---

*graph: Pingali, Yotov, Cornell U.*
Path to Fast Libraries

- 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)
- 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)
- Program generator for small linear algebra operations (SLinGen/LGen)