How to Write Fast Numerical Code

Spring 2013

Lecture: Dense linear algebra, LAPACK, MMM optimizations in ATLAS

Instructor: Markus Püschel

TA: Georg Ofenbeck & Daniele Spampinato

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Today

- **Linear algebra software: history, LAPACK and BLAS**
- **Blocking (BLAS 3): key to performance**
- **How to make MMM fast: 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 most of the numerical computation across disciplines (sciences, computer science, engineering)**
- **Efficient software is extremely relevant**

The Path to LAPACK

- **EISPACK and LINPACK (early 70s)**
  - Libraries for linear algebra algorithms
  - 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 (in the 80s)
- **Solution: LAPACK**
  - Reimplement the algorithms "block-based," i.e., with locality
  - Developed late 1980s, early 1990s
  - Jim Demmel, Jack Dongarra et al.

Matlab

- **Invented in the late 70s by Cleve Moler**
- **Commercialized (MathWorks) in 84**
- **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)

LAPACK and BLAS

- **Basic Idea:**

LAPACK — **static**

BLAS — **reimplemented for each platform**

- **Basic Linear Algebra Subroutines (BLAS, list)**
  - 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 implemented on top of BLAS**
  - Using BLAS 3 as much as possible

$I(n) =$

$O(1)$

$O(1)$

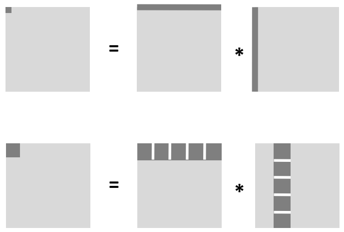
$O(\sqrt{C})$

cache size

Why is BLAS3 so important?

- **Using BLAS3 (instead of BLAS 1 or 2) in LAPACK**
  = blocking
  = high operational intensity I
  = high performance
- **Remember (blocking MMM):**

$I(n) =$



$O(1)$

$O(\sqrt{C})$

Today

- Linear algebra software: history, LAPACK and BLAS
- Blocking (BLAS 3): key to performance
- **How to make MMM fast: ATLAS, model-based ATLAS**

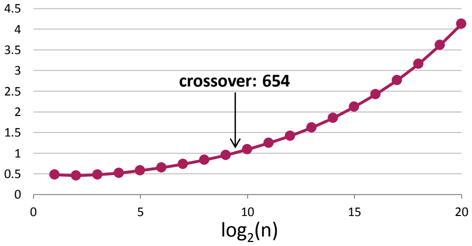
MMM: Complexity?

- **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 (see previous example)
  - 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:** $T(n) = 7T(n/2) + O(n^2) = O(n^{\log_2(7)}) \approx O(n^{2.808})$
- **Fewer ops from n=654, but …**
  - Structure more complex → performance crossover much later
  - Numerical stability inferior
- **Can we reduce more?**

MMM: Cost by definition/Cost Strassen



MMM Complexity: What is known

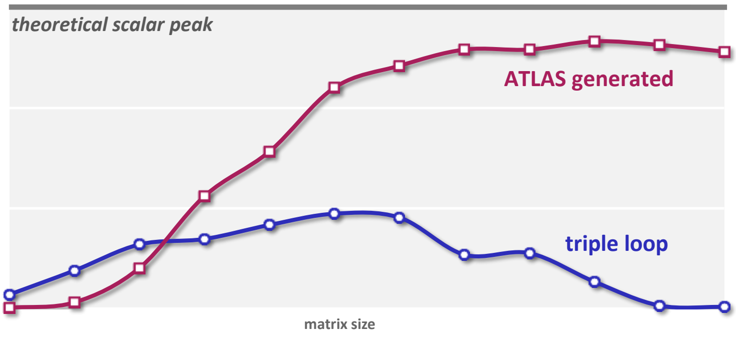
- Coppersmith, D. and Winograd, S.: **"Matrix Multiplication via Arithmetic Programming,"** *J. Symb. Comput.* 9, 251-280, 1990
- **MMM is $O(n^{2.376})$**

- **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

MMM: Memory Hierarchy Optimization

MMM (square real double) Core 2 Duo 3Ghz



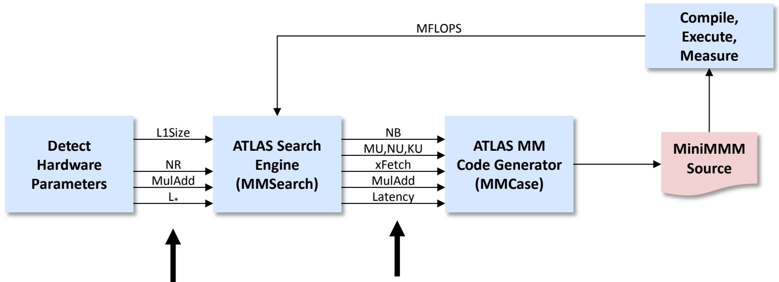
- **Huge performance difference for large sizes**
- **Great case study to learn memory hierarchy optimization**

ATLAS

- **BLAS program generator and library (web, successor of PhiPAC)**
- **Idea: automatic porting**

- **People can also contribute handwritten code**
- **The generator uses empirical search over implementation alternatives to find the fastest implementation**
  no vectorization or parallelization: so not really used anymore
- **We focus on BLAS 3 MMM**
- **Search only over cost $2n^3$ algorithms**
  (cost equal to triple loop)

ATLAS Architecture

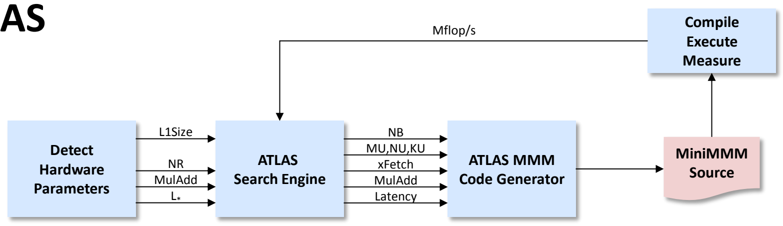


Search parameters:
- for example blocking sizes
- span search space
- specify code
- found by orthogonal line search

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

source: Pingali, Yotov, Cornell U.

ATLAS


Detect Hardware Parameters

L1Size

NR

MulAdd

L_*

ATLAS Search Engine

NB

MU,NU,KU

xFetch

MulAdd

Latency

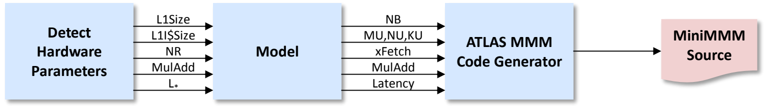
ATLAS MMM Code Generator

MiniMMM Source

Compile Execute Measure

Mflop/s

Model-Based ATLAS


Detect Hardware Parameters

L1Size

L1I$Size

NR

MulAdd

L_*

Model

NB

MU,NU,KU

xFetch

MulAdd

Latency

ATLAS MMM Code Generator

MiniMMM Source

- Search for parameters replaced by model to compute them
- More hardware parameters needed

source: Pingali, Yotov, Cornell U.

Optimizing MMM

- **Blackboard**
- **References:**

  "Automated Empirical Optimization of Software and the ATLAS project" by R. Clint Whaley, Antoine Petitet and Jack Dongarra. *Parallel Computing*, 27(1-2):3-35, 2001

  K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, Is Search Really Necessary to Generate High-Performance BLAS?, Proceedings of the IEEE, 93(2), pp. 358–386, 2005.

  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 $r_1 = r_3 + r_4$
R $r_2 = 2r_1$

nothing can be done
no ILP

- **Write after read (WAR) or antidependency**

R $r_1 = r_2 + r_3$
W $r_2 = r_4 + r_5$

dependency only by name → rename

$r_1 = r_2 + r_3$
$r = r_4 + r_5$

now ILP

- **Write after write (WAW) or output dependency**

W $r_1 = r_2 + r_3$
…
W $r_1 = r_4 + r_5$

dependency only by name → rename

$r_1 = r_2 + r_3$
…
$r = r_4 + r_5$

now ILP

Resolving WAR

R $r_1 = r_2 + r_3$
W $r_2 = r_4 + r_5$

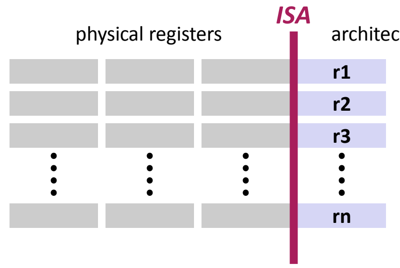
dependency only by name → rename

$r_1 = r_2 + r_3$
$r = r_4 + r_5$

now ILP

- **Compiler: Use a different register, r = r_6**
- **Hardware (if supported): register renaming**
  - Requires a separation of architectural and physical registers
  - Requires more physical than architectural registers

Register Renaming



- **Hardware manages mapping architectural → physical registers**
- **More physical than logical registers**
- **Hence: more instances of each r_i can be created**
- **Used in superscalar architectures (e.g., Intel Core) to increase ILP by resolving WAR dependencies**

Scalar Replacement Again

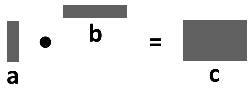
- **How to avoid WAR and WAW in your basic block source code**
- **Solution: Single static assignment (SSA) code:**
  - Each variable is assigned exactly once

no duplicates

```
<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 - a122) + (3.6*s267));
t296 = (a122 + a125 + (3.6*s269));
<more>
```

Micro-MMM Standard Model

- **MU*NU + MU + NU ≤ NR – ceil((Lx+1)/2)**
- **Core: MU = 2, NU = 3**



reuse in a, b, c

- **Code sketch (KU = 1)**

```
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)

- **MU = 1, NU = NR – 2 = 14**

reuse in c

- **Code sketch (KU = 1)**

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

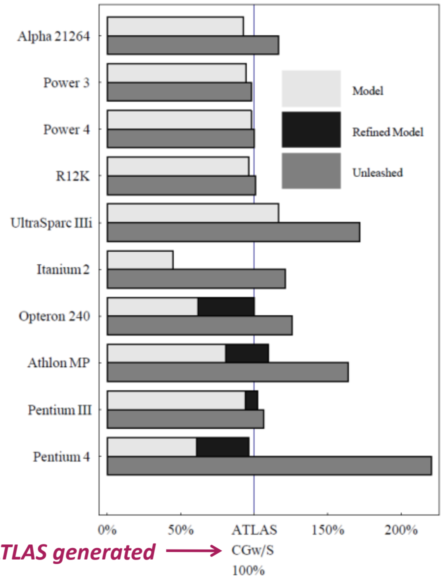
Summary:
- **no reuse in a and b**

+ larger tile size for c since for b only one register is used

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)**



ATLAS generated →

graph: Pingali, Yotov, Cornell U.

Remaining Details

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