Advanced Systems Lab

Spring 2020

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

Instructor: Markus Püschel, Ce Zhang

TA: Joao Rivera, Bojan Karlas, several more

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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
- **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)

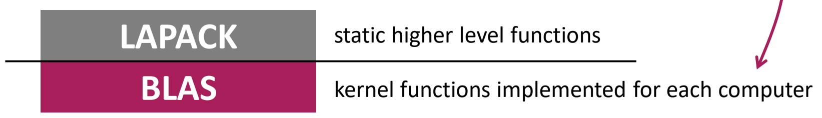
The Path to Fast Libraries

- **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)

The Path to Fast Libraries

Now there is implementation effort for each processor!

- **LAPACK (late 1980s, early 1990s)**
  - Redesign all algorithms to be "block-based" to increase locality
  - Jim Demmel, Jack Dongarra et al.
- **Two-layer architecture**



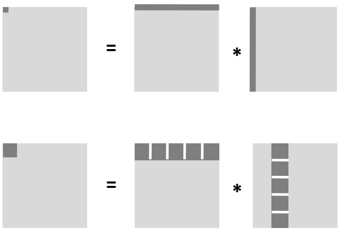
- **Basic Linear Algebra Subroutines (BLAS)**
  - BLAS 1: vector-vector operations (e.g., vector sum)
  - BLAS 2: matrix-vector operations (e.g., matrix-vector product)
  - BLAS 3: matrix-matrix operations (e.g., MMM)
- **LAPACK uses BLAS 3 as much as possible**

$I(n) =$
$O(1)$
$O(1)$
$O(\sqrt{C})$

cache
size

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):** $I(n) =$



$O(1)$

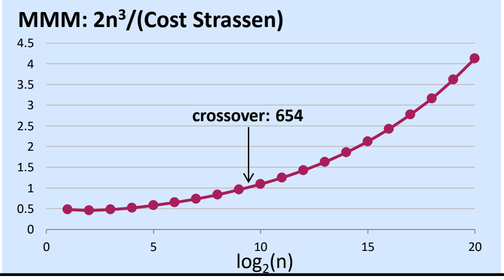
$O(\sqrt{C})$

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 → 15/4
- **Fewer ops from n = 654, but …**
  - Structure more complex → runtime crossover much later
  - Numerical stability inferior
- **Can we reduce more?**



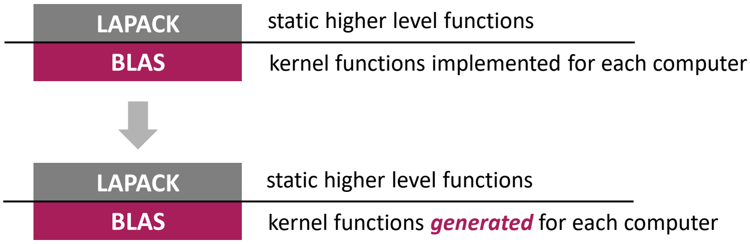
MMM Complexity: What is known

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



LAPACK — static higher level functions

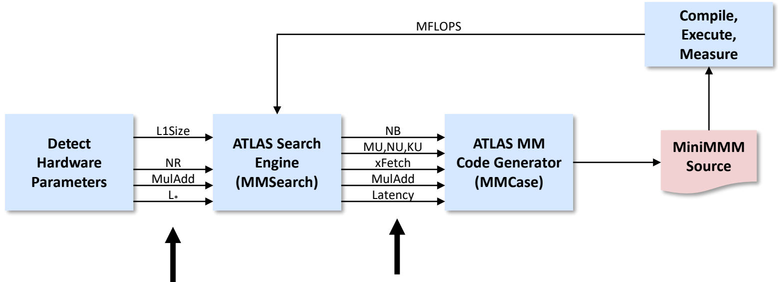
BLAS — kernel functions implemented for each computer

LAPACK — static higher level functions

BLAS — kernel functions ***generated*** 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



MFLOPS

Compile,
Execute,
Measure

Detect
Hardware
Parameters

L1Size
NR
MulAdd
L*

ATLAS Search
Engine
(MMSearch)

NB
MU,NU,KU
xFetch
MulAdd
Latency

ATLAS MM
Code Generator
(MMCase)

MiniMMM
Source

Search parameters:

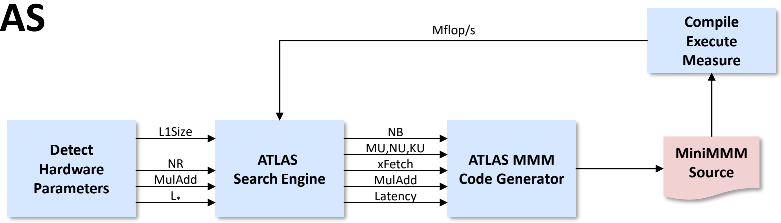
- for example blocking sizes
- span search space
- specify code
- found by orthogonal line search

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, et al., Cornell U.

ATLAS



Mflop/s

Compile
Execute
Measure

Detect
Hardware
Parameters

L1Size
NR
MulAdd
L*

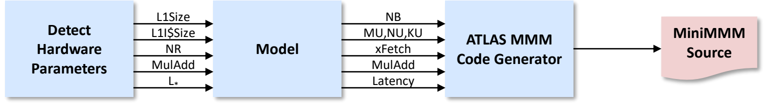
ATLAS
Search Engine

NB
MU,NU,KU
xFetch
MulAdd
Latency

ATLAS MMM
Code Generator

MiniMMM
Source

Model-Based ATLAS (2005)



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
- Much faster + provides understanding of what parameters are found

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

Optimizing MMM

- **References:**

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

  K. Goto and R. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Transactions on mathematical software (TOMS), 34(23), 2008

  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.

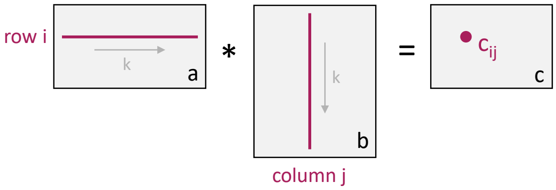
  Our presentation is based on this paper

0: Starting Point

Standard triple loop

```
// 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

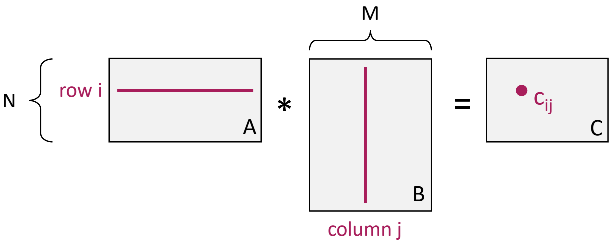


- **Most important in practice (based on usage in LAPACK)**
  - Two out of N, M, K are small
  - One out of N, M, K is small
  - None is small (e.g., square matrices)

1: Loop Order

```
// 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
```

i,j,k loops can be permuted in any order!

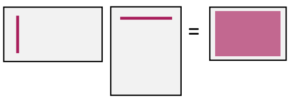


i-j-k: B is reused, good if M < N (B is smaller than A)

j-i-k: A is reused, good if N < M

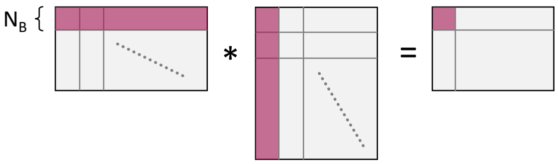
ATLAS does versioning (code for both variants)

Other options are inferior, e.g., k-i-j:



Poor temporal locality w.r.t. C

2: Blocking for Cache



Like multiplying matrices consisting of size N_B x N_B entries

Assume $N_B | M,N,K$

Results in six-fold loop

Formally obtained through loop-tiling and loop exchange

```
for i = 0:N_B:N-1
  for j = 0:N_B:M-1
    for k = 0:N_B:K-1
      for i’ = i:i+N_B-1
        for j’ = j:j+N_B-1
          for k’ = k:k+N_B-1
            c_i’j’ = c_i’j’ + a_i’k’*b_k’j’
```

mini-MMMs

How to find the best N_B?

ATLAS: uses search over all $N_B^2 \le C_1$ (cache size)

Model: explained next

2: Blocking for Cache

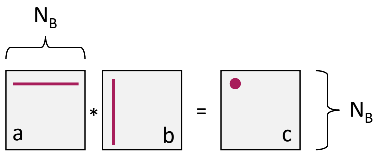
a) Idea: Working set has to fit into cache
Easy estimate: | working set | = 3 N_B^2
Model: 3 $N_B^2 \leq C_1$

b) Closer analysis of working set:

$$\underset{\substack{\uparrow \\ \text{all of b}}}{N_B^2} + \underset{\substack{\uparrow \\ \text{row of a}}}{N_B} + \underset{\substack{\uparrow \\ \text{element of c}}}{1} \leq C_1$$

c) Take into account cache block size B_1:

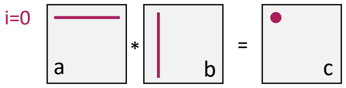
$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq \frac{C_1}{B_1}$$



a mini-MMM

2: Blocking for Cache

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



i=0: $a_{0,0}\, b_{0,0}\, a_{0,1} b_{1,0} \dots a_{0,N_B-1}\, b_{N_B-1,0}\, c_{0,0}$ (j=0)
$a_{0,0}\, b_{0,1}\, a_{0,1} b_{1,1} \dots a_{0,N_B-1}\, b_{N_B-1,1}\, c_{0,1}$ (j=1)
$\dots$
$a_{0,0}\, b_{0,N_B-1}\, a_{0,1} b_{1,N_B-1} \dots a_{0,N_B-1}\, b_{N_B-1,N_B-1}\, c_{0,N_B-1}$ (j=N_B-1)

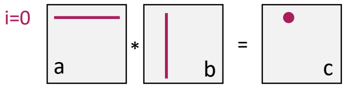
Corresponding history:

$b_{0,0}\, b_{1,0} \dots b_{N_B-1,0}\, c_{0,0}$
$b_{0,1}\, b_{1,1} \dots b_{N_B-1,1}\, c_{0,1}$
$\dots$
$a_{0,0}\, b_{0,N_B-1}\, a_{0,1} b_{1,N_B-1} \dots a_{0,N_B-1}\, b_{N_B-1,N_B-1}\, c_{0,N_B-1}$

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):

$b_{0,0}\, b_{1,0} \dots b_{N_B-1,0}\, c_{0,0}$
$b_{0,1}\, b_{1,1} \dots b_{N_B-1,1}\, c_{0,1}$
$\dots$
$a_{0,0}\, b_{0,N_B-1}\, a_{0,1} b_{1,N_B-1} \dots a_{0,N_B-1}\, b_{N_B-1,N_B-1}\, c_{0,N_B-1}$

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

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

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq C_1$$

2: Blocking for Cache

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

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B M_U}{B_1} \right\rceil + \left\lceil \frac{M_U N_U}{B_1} \right\rceil \leq \frac{C_1}{B_1}$$

3: Blocking for Registers

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

i-j-k:

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

k-i-j:

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

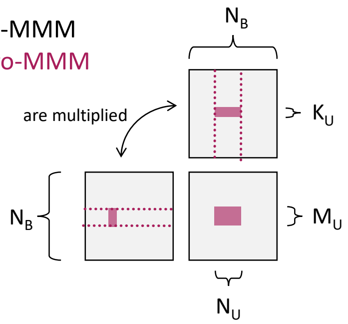
} Better ILP (but larger working set)

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

3: Blocking for Registers

```
for i = 0:N_B:N-1
  for j = 0:N_B:M-1
    for k = 0:N_B:K-1
      for i’ = i:M_U:i+N_B-1                                mini-MMM
        for j’ = j:N_U:j+N_B-1
          for k’ = k:K_U:k+N_B-1
            for k” = k’:k’+K_U-1                           micro-MMM
   X  {       for i” = i’:i’+M_U-1
                for j” = j’:j’+N_U-1
                  c_i”j” = c_i”j” + a_i”k”*b_k”j”
```

mini-MMM
micro-MMM



How to find the best M_U, N_U, K_U?

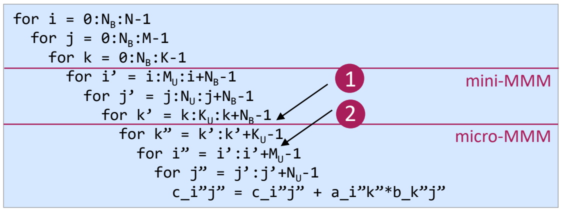
ATLAS: uses search with bound

$$\underbrace{M_U + N_U + M_U N_U}_{\text{size of working set in X}} \le N_R$$

number of registers

Model: Use largest M_U, N_U that satisfy this equation and $M_U \approx N_U$

4: Basic Block Optimizations



```
for i = 0:N_B:N-1
  for j = 0:N_B:M-1
    for k = 0:N_B:K-1
      for i' = i:M_U:i+N_B-1
        for j' = j:N_U:j+N_B-1
          for k' = k:K_U:k+N_B-1
            for k" = k':k'+K_U-1
              for i" = i':i'+M_U-1
                for j" = j':j'+N_U-1
                  c_i"j" = c_i"j" + a_i"k"*b_k"j"
```

Unroll micro-MMMs

Scalar replacement

Loads from c (M_UN_U many) at ①

Loads from a and b ($M_U + N_U$ many) at ②

Requires $M_U + N_U + M_UN_U$ scalar variables

Example of ATLAS-generated code

5: Other optimizations

- **Skewing: separate dependent add-mults for better ILP**
- **Software pipelining: move load from one iteration to previous iteration to high 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**

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 by Renaming

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

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

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>
```

Resolving WAR by Renaming

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

dependency only by name → rename

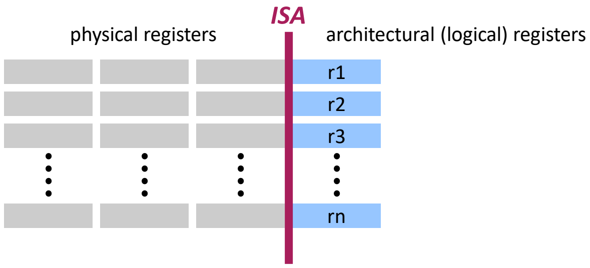
$r_1 = r_2 + r_3$
$r = r_4 + r_5$

now ILP

Renaming can be done at three levels:

- **C source code (= you rename)**
- **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



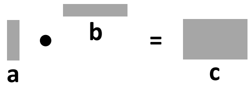
- **Hardware manages mapping architectural → physical registers**
- **Each logical register has several associated physical registers**
- **Hence: more instances of each r_i can be created**
- **Used in superscalar architectures (e.g., Intel Core) to increase ILP by dynamically resolving WAR/WAW dependencies**

Micro-MMM Standard Model

- MU*NU + MU + NU ≤ NR – ceil((Lx+1)/2)

this parameter I did not explain, see paper

- Core (NR = 16): 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)

- Set MU = 1, NU = NR – 2 = 14

reuse in c

- Code sketch (KU = 1)

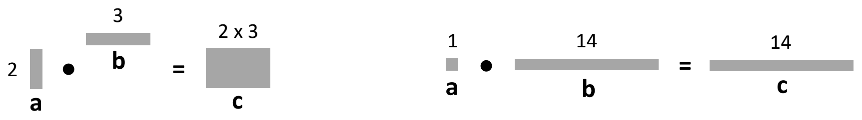
```
rc1 = c[0], …, rc14 = c[13] // 14 registers
loop over k {
  load a          // 1 register
  rb  = b[1]      // 1 register
  rb  = rb*a      // mult (two-operand)
  rc1 = rc1 + rb  // add  (two-operand)
  rb = b[2]       // reuse register (WAR: register renaming resolves it)
  rb = rb*a
  rc2 = rc2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

Summary:

- no reuse in a and b

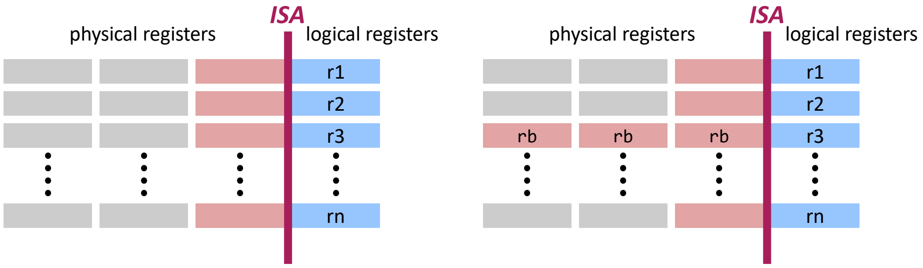
+ larger tile size available for c since for b only one register is used

Visualization of What Seems to Happen



reuse in a, b, c

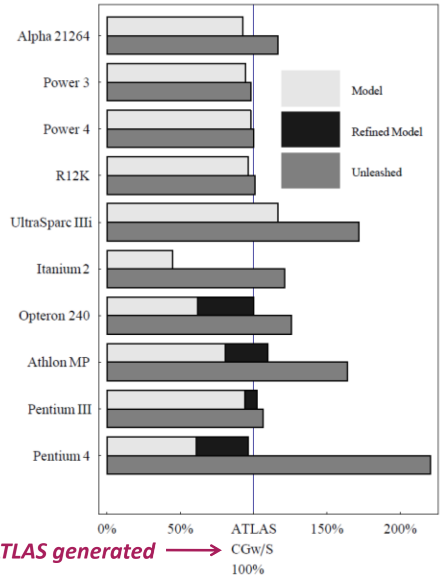
reuse in c



🟥 **registers 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

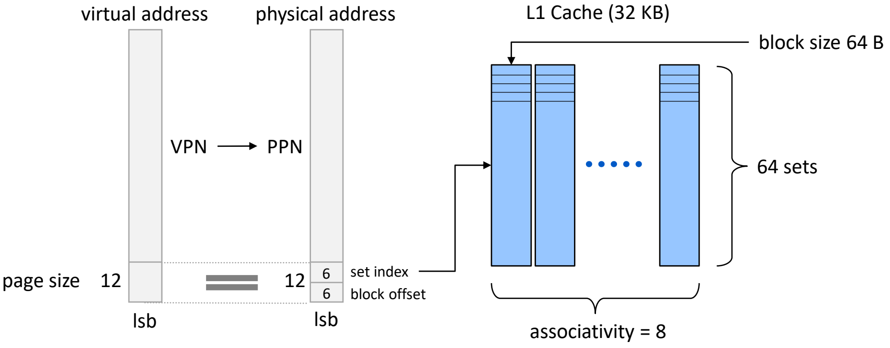
- Register renaming and the refined model for x86
- **TLB-related optimizations**

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



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)**
- **Haswell/Skylake:**

Level 1	ITLB (instructions): 128 entries DTLB (data): 64 entries
Level 2	Shared: 1024/1536 entries (Haswell/Skylake)

- **Miss Penalties:**
  - DTLB hit: no penalty
  - DTLB miss, STLB hit: few cycles penalty
  - STLB miss: can be very expensive

Impact on Performance

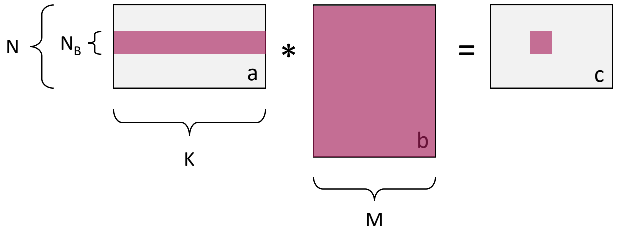
Repeatedly accessing a working set spread over too many pages yields TLB misses and can result in a significant slowdown.

Example Haswell:
STLB = 1024

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



Working set at highest level

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

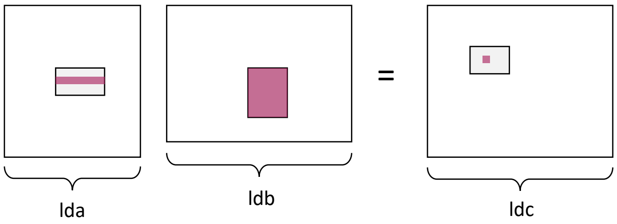
- Block row of a: contiguous
- All of b: contiguous
- Block of c: if M > 512, then spread over N_B pages

Typically, N_B is in the 10s, so no problem

Example MMM, contd.

Interface BLAS function: $\texttt{dgemm}(\underbrace{\texttt{a, b, c}}_{\text{matrices}}, \underbrace{\texttt{N, K, M}}_{\text{sizes}}, \underbrace{\texttt{lda, ldb, ldc}}_{\text{leading dimensions}})$

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.

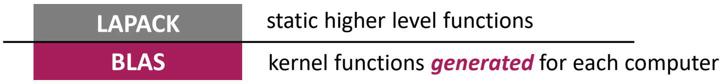
Resulting code (sketch):

```
// all of b reused: possible copy
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



LAPACK static higher level functions

BLAS kernel functions ***generated*** for each computer

- **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)**
- **Program generator for small linear algebra operations (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 studies as carefully as dense linear algebra