Do we need more efficiency?

- Some software is fast/small enough
- Some isn’t
- More frequent invocations, different work flow
- Bigger inputs
- Better functionality
- Energy savings
Types of efficiency

Run time
- CPU
- hard disk/SSD
- network
- other I/O

Memory
- RAM
- ROM
- persistent storage
- removable storage
Costs of inefficiency

- Loss of user time
- Different work flow
- Misses real time requirements
- More expensive hardware
- Energy
How much efficiency is sensible?

- Command line: 300ms to response
- Music: 20ms latency
- Animated software: screen refresh rate (7-16ms).
- A different component dominates
- Commercial considerations
Other goals

- Correctness
- Simplicity
- Development effort
- Maintenance effort
- Time-to-market
- Security
Extreme positions

• No efficiency considerations

• Optimize everything!
Observations

• 80-20 Rule

• Programmers are bad at predicting hot spots

General approach

• Start simple, flexible, maintainable

• Measure

• Optimize critical parts

Problem: Bad efficiency due to specification and design
Method

unoptimized program
Tests
succeeded
Measurement
too inefficient
Profiling
sufficiently efficient
program transformation
Is this not a job for the compiler?

Compilers use program transformations, too, but

- use the input program as specification
- avoids potential pessimizations
- only performs optimizations that use little time and space during compilation.
- only performs optimizations useful for many applications (or for benchmarks)
- optimizations depend on each other

\[ *s1 == *s2 \&\& *s1! = 0 \&\& *s2! = 0 \]
Optimization: Compiler vs. Programmer

gcc-5.2.0 -O0 [-fno...]
clang-3.5 -O0 [-fno...]
gcc-2.7.2.3 -O3
cpp-3.5 -O3 [-fno...]
egcs-1.1.2 -O3
gcc-5.2.0 -O3
gcc-5.2.0 -O3 -fno...

speed
tsp1
tsp2
tsp3
tsp4
tsp5
tsp6
tsp8
tsp9

1
0.5
0.2
0.1
0.05
Example: Stumbling blocks for compilers

```c
for (i=0, best=0; i<n; i++)
    if (a[i]<a[best])
        best=i;
return best;

for (p=a, bestp=a, endp=a+n; p<endp; p++)
    if (*p < *bestp)
        bestp = p;
return bestp-a;

for (i=0, bestp=a; a+i<a+n; i++)
    if (a[i]<*bestp)
        bestp=a+i;
return bestp-a;
```
Common stumbling blocks for compilers

• Aliasing
  
  ```c
  *p = ...  
  ... = *q;  
  ```

  ```c
  for (i=0; i<n; i++)  
  a[i] = a[i]*b[j];  
  ```

• side effects, exceptions

  ```c
  if (flag)  
  printf(...)
  ```

  ```c
  for (i=0; i<n; i++)  
  a[i] = a[i]+1/b[j];  
  ```
Hardware properties

1c  2–8 independent instructions
1c  latency of an ALU instruction
3–5c latency of a load (L1-hit)
14c latency of a load (L1-miss, L2-hit)
50c latency of a load (L2-miss, L3-hit)
50–ns latency of a load (L3-miss, main memory access)
  3ns Transmission of a cache line (64B) from/to DDR4-2666, DDR5-5200
0–1c correctly predicted branch
20c mispredicted branch
4c latency integer multiply
4c latency FP addition/multiplication
30–90c latency division
>100us IP-Ping in local ethernet Ethernet
  10us 1KB transmission across GB Ethernet
  10ms latency hard disk access (seek+rotational delay)
  10ms 2500KB sequential hard disk access (without delay)
Hardware properties: latency

```c
while (i<n) {
    r+=a[i];
    i++;
}
```

```assembly
add   (%rdi),%rax
add   $0x8,%rdi
cmp   %rdx,%rdi
jne   top1
```

Skylake: 1.29c/Iteration

```c
while (a!=0) {
    r += a->val;
    a = a->next;
}
```

```assembly
add    0x8(%rdi),%rax
mov    (%rdi),%rdi
test   %rdi,%rdi
jne    top2
```

Skylake: 4c/Iteration
Hardware properties: latency

while (i<n) {
    r+=a[i];
    i++;
}

Skylake: 1.29c/Iteration

add    (%rdi),%rax
add    $0x8,%rdi
cmp    %rdx,%rdi
jne    top1

5
1
cycles
iterations

while (a!=0) {
    r += a->val;
    a = a->next;
}

Skylake: 4c/iteration

add 0x8(%rdi),%rax
mov    (%rdi),%rdi
test   %rdi,%rdi
jne    top2

4
8
iterations
cycles

4
iterations
15
Program properties: latency vs. throughput

// double a[], r;
while (i<n) {
    r+=a[i];
    i++;
}
Skylake: 4c/Iteration

// double a[], f;
while (i<n) {
    a[i]=a[i]+f;
    i++;
}
Skylake: 1.37c/iteration

with vectorization:
gcc -O3 -mavx:
Skylake: 0.45c/iteration
Program properties

Latency dominated
- dependent operations on the same data
- data often is in the cache
- most code (by lines)
- helpful: OoO, branch prediction, caches
- sometimes independent instances e.g., compilers, on-line-systems
- helpful: multi-core CPUs

Throughput dominated
- same operations on lots of data e.g., pictures, audio, graphics, matrices, tensors, neural nets
- often needs (main) memory bandwidth
- little code (by lines)
- much run time
- helpful: SIMD, multi-core CPUs, GPUs
- helpful: memory bandwidth
Hardware properties: memory/cache
Hardware properties: memory/cache

- temporal locality (program property)
- spatial locality (program property)
- compulsory misses (program property)
- capacity misses
- conflict misses

Intel Skylake (Core ix-6xxx):
- data cache (L1): 32KB, 64B/line, 8-way, 4c
- instruction cache (L1): 32KB, 64B/line, 8-way
- L2 cache: 256KB, 64B/line, 4-way, 12c
- L3 cache: 2-8MB, 64B/line, 4-16-way, ≥42c
- RAM: ≈50ns
- DTLB L1: 64 entries (4KB), 4-way
- DTLB L1: 32 entries (2MB), 4-way
- DTLB L2: 1536 e. (4KB, 2MB), 12-way, 9c
Data structures and algorithms

- Efficient implementation of an inefficient algorithm? Waste of time
- Efficient algorithm, never mind implementation efficiency?
- Efficient implementation of an efficient algorithm
- Efficient algorithm/data structure may conflict with simplicity
- Data structure may affect much of the code
- Abstract data type
  - Inefficiency due to abstraction:
    - interface overhead
    - lack of cost awareness
Algorithmic complexity (O(...))

- Helpful, but be aware of its limitations
- Often looks at the worst case
- Counts certain operations, not always relevant for run time
- Ignores constant factors
- Logarithmic factors

- E.g.: Search substring (length $m$) in string (length $n$)
  - Simple algorithm: $O(mn)$ (worst), $O(n)$ (best)
  - KMP: $O(n)$, but usually slower than the simple algorithm
  - BM: $O(n)$ (worst), $O(n/m)$ (best)

- Quicksort: $O(n^2)$ (worst), $O(n \ln n)$ (usual), spatial and temporal locality
  - Heapsort: $O(n \ln n)$, bad locality
  - Mergesort: $O(n \ln n)$, good locality
Parallel processing

- Problems: find parallelism, express parallelism, synchronization overhead
- Between CPU cores: multithreading, parallel computing
- Between CPU and mass storage: prefetching, write buffering
- Between graphics card and screen: triple buffering
- Between CPU and main memory: prefetching
- Between instructions: instruction scheduling
- SIMD
Triple buffering

- Double buffering without vertical sync: Tearing
- Double buffering with vertical sync: Wait for vsync
- Triple buffering: no tearing and no waiting
Exploit Word Parallelism/SIMD

for (count=0; x > 0; x >>= 1)
    count += x&1;

/* 64-bit-spezifisch */
x = (x & 0x5555555555555555L) + ((x>>1) & 0x5555555555555555L);
x = (x & 0x3333333333333333L) + ((x>>2) & 0x3333333333333333L);
x = (x+(x>>4)) &0x0f0f0f0f0f0f0f0fL;
x = (x+(x>>8)) /*&0x001f001f001f001fL*/;
x = (x+(x>>16))/*&0x0000003f0000003fL*/;
x = (x+(x>>32)) &0x7fL;
count = x;

0|0|0|1|1|0|1|1
  0|  1|  1|  2
     1|  3
        4
### Efficiency in specification: Copy a memory block

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memcpy()</code> (C)</td>
<td><code>move</code> (Forth)</td>
</tr>
<tr>
<td><code>memmove()</code> (C)</td>
<td><code>cmove</code> (Forth)</td>
</tr>
<tr>
<td><code>rep movsb</code> (AMD64)</td>
<td><code>memcpy()</code> (C)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition</th>
<th><code>memcpy()</code> (C)</th>
<th><code>move</code> (Forth)</th>
<th><code>cmove</code> (Forth)</th>
<th><code>rep movsb</code> (AMD64)</th>
<th><code>memcpy()</code> (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no overlap</td>
<td>source $\rightarrow$ dest.</td>
<td>source $\rightarrow$ dest.</td>
<td>source $\rightarrow$ dest.</td>
<td>source $\rightarrow$ dest.</td>
<td>source $\rightarrow$ dest.</td>
</tr>
<tr>
<td>start of dest. in source</td>
<td>start of source in dest.</td>
<td>pattern replication</td>
<td>source $\rightarrow$ dest.</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>implementation</td>
<td>decision</td>
<td>byte by byte decision</td>
<td>bigger units</td>
<td></td>
<td></td>
</tr>
<tr>
<td>efficient implementation</td>
<td>well specified</td>
<td>overspecified</td>
<td>underspecified</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### What’s wrong with “undefined behaviour”?

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.  

Hyrum’s law
Programming languages

- inherent inefficiency
- idiomatic inefficiency
- compiler efficiency
- (potential) efficiency due to development speed
- assembly language?
Programming languages: Examples

- Aliasing: C vs. Fortran (inherent)

```c
void f(double a[], double b[], double c[], long n) {
    for (long i=0; i<n; i++)
        c[i] = a[i] + b[i];
}
```
Programming languages: Examples

• Nested data: Java vs. C(++) (inherent)

  struct mystruct { int a; float b; double c; }
  struct mystruct a[10000];
  struct mystruct *b[10000];

• Scaling in address arithmetics: C vs. Forth (inherent/idiomatic)

  mystruct *p; ... constant p
  mystruct *q; ... constant q
  ...
  long d = q-p; q p - constant d1
  mystruct *r = p+d; p d1 + constant r
Programming languages: examples

- 0-terminated strings in C (inherent/idiomatic)
  
  ```c
  l=strlen(s);
  strcat(strcat(strcat(s,s1),s2),s3);
  ```

- “C++ is slow” (idiomatic)

- Microbenchmarks (compiler)

- programming contests (development speed)

- Riad air port
Code motion out of loops

for (...) {
    .... computation ...
}

computation has no side effects
computation does not need values computed in the loop

temp = computation;
for (...) {
    .... temp ...
}

Combining Tests

E.g., sentinel in search loops

```
for (i=0; i<n && a[i]!=key; i++)
```

`a[n]` is writable

```
a[n] = key;
for (i=0; a[i]!=key; i++)
  ;
```

lowers maintainability, reentrancy
Loop Unrolling

for (i=0; i<n; i++)
    body(i);

for (i=0; i<n-1; i+=2) {
    body(i);
    body(i+1);
}
for (; i<n; i++)
    body(i);
Transfer-Driven Unrolling/Modulo Variable Renaming

new_a = ...
...
= ... a ...
a = new_a

Unrolling by 2

a2 = ...
...
= ... a1 ...
a1 = ...
...
= ... a2 ...
Software Pipelining

for (...) {
    a = ...;
    ... = ... a ...;
}

Computing \( a \) has no side effects

a = ...;
for (...) {
    ... = ... a ...;
    a = ...;
}

\texttt{new}_a = ...;
for (...) {
    a = \texttt{new}_a;
    \texttt{new}_a = ...;
    ... = ... a ...;
}

Unconditional Branch Removal

while (test)
  code;

if (test)
  do
do
code;
while (test);
while (test) {
    code;
}

if (test) {
    code;
    while (test) {
        code;
    }
}
Loop Fusion

for (i=0; i<n; i++)
    code1;
for (i=0; i<n; i++)
    code2;

Iteration $k$ in code2 does not depend on Iteration $j > k$ in code1.
Code2 does not overwrite data that is read by code1.

for (i=0; i<n; i++) {
    code1;
    code2;
}
Exploit Algebraic Identities

\(~a\&\sim b\)

\(~(a\mid b)\)

Computer “integers” are not \(\mathbb{Z}\).
FP numbers are not \(\mathbb{R}\).

**Integer:** Overflow: \(a > b \not\iff a + n > b + n\)

**FP:** round-off errors: \(a + (b + c) \neq (a + b) + c\)
for (i=0, sum=0; i<n; i++)
    sum += x[i];
flag = sum > cutoff;

All $x[i] \geq 0$, sum and i are not used later.

for (i=0, sum=0; i<n && sum <= cutoff; i++)
    sum += x[i];
flag = sum > cutoff;

Unrolling for fewer comparisons and branches.
Arithmetics with flags

```c
if (flag)
    x++;

x += (flag != 0);
```
Different representation of flags

(a<0) != (b<0)

(a^b) < 0
Long-circuiting

A && B

A and B compute flags, B has no side effects

A & B

When to use: If B is cheap and A is hard to predict
Reordering Tests

A && B

A and B have no side effects

B && A

Which order?

- Cheaper first
- More predictable first
- higher probability of short-circuiting first
Reordering Tests

if (A)
   ...
else if (B)
   ...

A and B have no side effects, \( \neg (A \land B) \).

if (B)
   ...
else if (A)
   ...
Boolean/State Variable Elimination

flag = ...;
S1;
if (flag)
    S2;
else
    S3;

flag is not used later.

if (...) {
    S1;
    S2;
} else {
    S1;
    S3;
}
Collapsing Procedure Hierarchies

- Inlining
- Specialization

```c
foo(int i, int j)
{
...
}
... foo(1, a);

foo_1(int j)
{
...
}
```
int foo(char c)
{
    ...
}

foo() has no side effects.

int foo_table[] = {...};

int foo(char c)
{
    return foo_table[c];
}
Exploit Common Cases

Handle all cases correctly and common cases efficiently.

- Memoization: Remember results of earlier evaluations of expensive function
- Pre-computed tables or special code sequences for frequent parameters
Coroutines

Instead of multi-pass processing:

coroutine producer {
    for (...
        ... consumer(x); ...
}

coroutine consumer {
    for (...
        ... x = producer(); ...
}

Related: Pipelines, Iterators, etc.
Transformation on Recursive Procedures

- Tail call optimization
- Inlining
- Replace one recursive call by counter
- General case: Use explicit stack
- Use different method for small problems
- Use recursion instead of iteration for automatic cache blocking
Tail Call Optimization

void traverse_simple(PNODE p)
{
    if (p!=0)
    {
        traverse_simple(p->l);
        ...
        traverse_simple(p->r);
    }
}

void traverse_simple(PNODE p)
{
    start:
    if (p!=0)
    {
        traverse_simple(p->l);
        ...
        p = p->r; goto start;
    }
}
Replace one recursive call by counter

```c
foo()
{
    if (...) {
        code1;
        foo();
        code2;
    }
}
```

```c
while (...) {
    count++;
    code1;
}
```

```c
for (i=0; i<count; i++)
    code2;
```
Compile-Time Initialization

- Initialize tables at compile-time instead of at run-time
- CPU time vs. load time from disk
Strength Reduction/Incremental Algorithms/Differentiation

\[
y = x \times x; \\
x += 1; \\
y = x \times x; \\
y = x \times x; \\
x += 1; \\
y += 2x - 1;
\]
Common subexpression elimination/Partial Redundancy Elimination

a = Exp;
b = Exp;

Exp has no side effects

a = Exp;
b = a;
Pairing Computation

- Additional result for small effort
- E.g., division and remainder (C: div)
  sin and cos (glibc: sincos)
Data Structure Augmentation

- Redundant data for accelerating certain operations
- Redundancy: possibility of inconsistency
- Caching
- Memoization
- Hints that can be correct, or not (e.g., branch prediction)
- Example: dictionary in Gforth: linked list augmented with hash table
Automata

- state represents something more complex
- finite state machine for scanning
- pushdown automaton for parsing
- tree automaton for instruction selection

iburg (not an automaton) → burg
Lazy Evaluation

- Example: automaton for regular expressions
- Example: tree-parsing automaton
Memory efficiency: Packing

- No unused Bytes/Bits (bitfields in C, packed in Pascal)
- Data compression
- Code size
- cache behaviour
Interpreters, Factoring

- Turn similar code fragments into procedures and call them
- Implement schematic programs through an interpreter
Energy efficiency

• Fewer Cycles → less power consumption
  What do you do if the job is done?

• Dynamic Voltage and Frequency Scaling (DVFS)
  \[ P = C U^2 f \]

• Tools
  turbostat -show PkgWatt,CorWatt,GFXWatt,RAMWatt
  powerstat

• Race to idle?

• How can you as user influence that?
  set frequency limit
  set power limit
Source: https://chipsandcheese.com/2022/01/28/alder-lakes-power-efficiency-a-complicated-picture/
Source: https://chipsandcheese.com/2022/01/28/alder-lakes-power-efficiency-a-complicated-picture/
Program example: Traveling Salesman Problem

• Visit a set of cities, each city once
  Minimize total distance traveled

• Optimal solution: NP-complete

• Example by Jon Bentley: suboptimal algorithm
  Travel from each city to the nearest one (greedy)
  $O(n^2)$, $\approx 25\%$ worse than optimal
Tools

• gprof: profiling at function level
  
gcc -pg -O tsp1.c -lm -o tsp1
tsp1 10000 >/dev/null
gprof tsp1

• gcov: Profiling at line level
  
gcc -O --coverage tsp1.c -lm -o tsp1
tsp1 10000 >/dev/null
gcov tsp1
cat tsp1.c.gcov
Tools

- **perf stat**: Performance monitoring counters

  ```
  gcc -O tsp1.c -lm -o tsp1
  perf list
  ```

- **perf-based profiling**

  ```
  perf record -e cycles:u tsp1 10000 >/dev/null
  perf annotate -s tsp
  perf report
  ```
for (i=1; i<ncities; i++) {
    CloseDist = DBL_MAX;
    for (j=0; j<ncities-1; j++) {
        if (!visited[j]) {
            if (dist(cities, ThisPt, j) < CloseDist) {
                CloseDist = dist(cities, ThisPt, j);
                ClosePt = j;
            }
        }
    }
    tour[endtour++] = ClosePt;
    visited[ClosePt] = 1;
    ThisPt = ClosePt;
}
tsp1 → tsp2: Common subexpression elimination

```java
double ThisDist = dist(cities, ThisPt, j);
if (dist(cities, ThisPt, j) < CloseDist) {
    if (ThisDist < CloseDist) {
        CloseDist = ThisDist;
    } else {
        CloseDist = dist(cities, ThisPt, j);
    }
} else {
    CloseDist = ThisDist;
}
```
tsp2 → tsp3: Eliminate sqrt

double dist(point cities[], int i, int j) {
    return sqrt(
        sqr(cities[i].x-cities[j].x)+
        sqr(cities[i].y-cities[j].y));
}
double ThisDist =
    dist(cities, ThisPt, j);

double DistSqrdd(point cities[], int i, int j) {
    return (sqr(cities[i].x-cities[j].x)+
            sqr(cities[i].y-cities[j].y));
}
double ThisDist =
    DistSqrdd(cities, ThisPt, j);
tsp3 → tsp4: Eliminate visited

for (i=0; i<ncities; i++)
    visited[i]=0;
...
for (j=0; j<ncities-1; j++) {
    if (!visited[j]) {
        double ThisDist =
            DistSqrd(cities, ThisPt, j);
        ...
    }
}
ThisPt = ClosePt;
tour[endtour++] = ClosePt;
visited[ClosePt] = 1;

for (i=1; i<ncities; i++)
    tour[i]=i-1;
...
for (j=i; j<ncities; j++) {
    double ThisDist =
        DistSqrd(cities, ThisPt, tour[j]);
    ...
}
ThisPt = tour[ClosePt];
swap(&tour[i],&tour[ClosePt]);
tsp4 → tsp5: Inline DistSqrd

```
for (j=i; j<ncities; j++) {
    double ThisDist =
        DistSqrd(cities, ThisPt, tour[j]);
    double ThisX = cities[ThisPt].x;
    double ThisY = cities[ThisPt].y;
    for (j=i; j<ncities; j++) {
        double ThisDist =
            sqr(cities[tour[j]].x-ThisX)+
            sqr(cities[tour[j]].y-ThisY);
    }
}
```
tsp5 $\rightarrow$ tsp6: lazy computation of $y$-Distance

double ThisDist =
    sqr(cities[tour[j]].x-ThisX) +
    sqr(cities[tour[j]].y-ThisY);
if (ThisDist < CloseDist) {
    CloseDist = ThisDist;
    ClosePt = j;
}
Skipped: Integers instead of FP numbers
void tsp(point cities[], int tour[], int ncities)
...

double ThisX = cities[ThisPt].x;
double ThisY = cities[ThisPt].y;
CloseDist = DBL_MAX;
for (j=i; j<ncities; j++) {
    double ThisDist =
        sqr(cities[tour[j]].x-ThisX);
    if (ThisDist < CloseDist) {
        ThisDist +=
            sqr(cities[tour[j]].y-ThisY);
        ...
    }
}
ThisPt = tour[ClosePt];

void tsp(point cities[], point tour[], int ncities)
...

double ThisX = tour[i-1].x;
double ThisY = tour[i-1].y;
CloseDist = DBL_MAX;
for (j=i; j<ncities; j++) {
    double ThisDist =
        sqr(tour[j].x-ThisX);
    if (ThisDist < CloseDist) {
        ThisDist +=
            sqr(tour[j].y-ThisY);
        ...
    }
}
for (j=i; j<ncities; j++) {
    double ThisDist = sqr(tour[j].x-ThisX);
    if (ThisDist < CloseDist) {
        ThisDist += sqr(tour[j].y-ThisY);
        if (ThisDist < CloseDist) {
            CloseDist = ThisDist;
            ClosePt = j;
        }
    }
}

for (j=ncities-1; ;j--) {
    double ThisDist = sqr(tour[j].x-ThisX);
    if (ThisDist <= CloseDist) {
        ThisDist += sqr(tour[j].y-ThisY);
        if (ThisDist <= CloseDist) {
            if (j < i)
                break;
            CloseDist = ThisDist;
            ClosePt = j;
        }
    }
}
Example: Matrix multiply

\[ C = AB \]

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \]
Example: Matrix multiply

```
for (i=0; i<n; i++)
  for (j=0; j<p; j++) {
    for (k=0, r=0.0; k<m; k++)
      r += a[i*m+k]*b[k*p+j];
    c[i*p+j]=r;
  }
```

$n, p, m = 1000$: 4.6c/Iteration  
$n, p, m = 1000$: 4.1c/Iteration  THP

```
for (i=0; i<n; i++)
  for (j=0; j<p; j++)
    c[i*p+j] = 0.0;
for (i=0; i<n; i++)
  for (j=0; j<p; j++)
    for (k=0; k<m; k++)
      c[i*p+j] += a[i*m+k]*b[k*p+j];
```

$n, p, m = 1000$: 5.0c/Iteration  
$n, p, m = 1000$: 4.5c/Iteration  THP
Which nesting? \( n, p, m = 1000 \)

\[
\begin{align*}
\text{for (i=0; i<n; i++)} & \quad \text{for (i=0; i<n; i++)} & \quad \text{for (j=0; j<p; j++)} \\
\text{for (j=0; j<p; j++)} & \quad \text{for (k=0; k<m; k++)} & \quad \text{for (k=0; k<m; k++)} \\
\text{for (k=0; k<m; k++)} & \quad \text{for (j=0; j<p; j++)} & \quad \text{for (i=0; i<n; i++)} \\
\text{c[i*p+j]} & = \text{a[i*m+k]} \times \text{b[k*p+j]} & \text{c[i*p+j]} & = \text{a[i*m+k]} \times \text{b[k*p+j]} & \text{c[i*p+j]} & = \text{a[i*m+k]} \times \text{b[k*p+j]} \\
\end{align*}
\]

\[
\begin{align*}
\text{for (j=0; j<p; j++)} & \quad \text{for (k=0; k<m; k++)} & \quad \text{for (k=0; k<m; k++)} \\
\text{for (i=0; i<n; i++)} & \quad \text{for (i=0; i<n; i++)} & \quad \text{for (j=0; j<p; j++)} \\
\text{for (k=0; k<m; k++)} & \quad \text{for (j=0; j<p; j++)} & \quad \text{for (i=0; i<n; i++)} \\
\text{c[i*p+j]} & = \text{a[i*m+k]} \times \text{b[k*p+j]} & \text{c[i*p+j]} & = \text{a[i*m+k]} \times \text{b[k*p+j]} & \text{c[i*p+j]} & = \text{a[i*m+k]} \times \text{b[k*p+j]} \\
\end{align*}
\]
Which nesting?  \( n, p, m = 1000 \)

\[
\begin{align*}
\text{for (i=0; i<n; i++)} & \quad \text{for (j=0; j<p; j++)} & \quad \text{for (k=0; k<m; k++)} \\
\text{for (j=0; j<p; j++)} & \quad \text{for (k=0; k<m; k++)} & \quad \text{for (j=0; j<p; j++)} \\
\quad \text{for (k=0; k<m; k++)} & \quad \text{for (i=0; i<n; i++)} & \quad \text{for (k=0; k<m; k++)} \\
\text{c[i*p+j]=a[i*m+k]*b[k*p+j];} & \quad \text{c[i*p+j]=a[i*m+k]*b[k*p+j];} & \quad \text{c[i*p+j]=a[i*m+k]*b[k*p+j];}
\end{align*}
\]

\(-02: 5.0c/It \quad -02: 2.3c/It \quad -02: 17.5c/It\)

\(-02: 4.5c/It \quad -02: 2.2c/It \quad -02: 5.3c/It\)

\(-03: 4.5c/It \quad -03: 0.84c/It \quad -03: 5.3c/It\)

\(-02: 4.4c/It \quad -02: 2.5c/It \quad -02: 17.9c/It\)

\(-02: 4.2c/It \quad -02: 2.3c/It \quad -02: 5.1c/It\)

\(-03: 4.2c/It \quad -03: 0.99c/It \quad -03: 5.0c/It\)
Reasons

• spatial locality
  TLB misses
  cache misses
  \( j \) as inner loop
  \( j \) allows using SIMD instructions (auto-vectorization: \(-03\))

• Recurrences (Dependences between iterations)
  not \( k \) als innermost loop

• temporal locality
  \( k \) als middle loop: reuse \( c[i*p+j] \) line
void matmul(
    double a[], double b[], double c[],
    size_t m, size_t n, size_t p)
{

0.85Z/It

typedef double v8d
    __attribute__((vector_size(64)));

void matmul(
    double a[], v8d b[], v8d c[],
    size_t m, size_t n, size_t p)
{
    p=p/8;

0.72Z/It
mm3 → mm4: Loop-invariant code motion

for (j=0; j<p; j++)
c[i*p+j] += a[i*m+k]*b[k*p+j];

0.72Z/It

double aik = a[i*m+k];

for (j=0; j<p; j++)
c[i*p+j] += aik*b[k*p+j];

0.70Z/It
for (k=0; k<m; k++) {
    double aik = a[i*m+k];
}

for (j=0; j<p; j++)
    c[i*p+j] += aik*b[k*p+j];

for (k=0; k<m; k+=4) {
    double aik0 = a[i*m+k+0];
    double aik1 = a[i*m+k+1];
    double aik2 = a[i*m+k+2];
    double aik3 = a[i*m+k+3];
    for (j=0; j<p; j++) {
        v8d r;
        r = aik0*b[(k+0)*p+j];
        r += aik1*b[(k+1)*p+j];
        r += aik2*b[(k+2)*p+j];
        r += aik3*b[(k+3)*p+j];
        c[i*p+j] += r;
    }
}

0.70Z/It

0.66Z/It
for (i=0; i<n; i++)
    for (k=0; k<m; k+=4)

static void matmul1(
    double a[], v8d b[], v8d c[],
    size_t m, size_t n, size_t p,
    size_t m1, size_t n1)
{
    if (m1>=8) {
        size_t m2 = (m1/2)&~3;
        size_t m3 = m1-m2;
        matmul2(a ,b ,c,m,n,p,m2,n1);
        matmul2(a+m2,b+m2*p,c,m,n,p,m3,n1);
    } else {
        matmul2(a,b,c,m,n,p,m1,n1);
    }
}

0.66Z/It

0.28Z/It
for (i=0; i<n1; i++) {
    double aik0 = a[i*m+0];
    double aik1 = a[i*m+1];
    double aik2 = a[i*m+2];
    double aik3 = a[i*m+3];
    for (j=0; j<p; j++) {
        v8d r;
        r = aik0*b[0*p+j];
        r += aik1*b[1*p+j];
        r += aik2*b[2*p+j];
        r += aik3*b[3*p+j];
        c[i*p+j] += r;
    }
}

for (i=0; i<n1; i+=2) {
    double ai0k0 = a[(i+0)*m+0];
    double ai1k0 = a[(i+1)*m+0];
    double ai0k1 = a[(i+0)*m+1];
    double ai1k1 = a[(i+1)*m+1];
    double ai0k2 = a[(i+0)*m+2];
    double ai1k2 = a[(i+1)*m+2];
    double ai0k3 = a[(i+0)*m+3];
    double ai1k3 = a[(i+1)*m+3];
    for (j=0; j<p; j++) {
        v8d bk0j = b[0*p+j];
        v8d bk2j = b[2*p+j];
        v8d bk1j = b[1*p+j];
        v8d bk3j = b[3*p+j];
        v8d ci0j = ai0k0*bk0j+ai0k1*bk1j+ai0k2*bk2j+ai0k3*bk3j;
        v8d ci1j = ai1k0*bk0j+ai1k1*bk1j+ai1k2*bk2j+ai1k3*bk3j;
        c[(i+0)*p+j] += ci0j;
        c[(i+1)*p+j] += ci1j;
    }
}

0.28Z/It

0.25Z/It
ATLAS, OpenBLAS

- ATLAS: 0.54Z/It
- OpenBLAS (1 thread): 0.16Z/It