

Advanced Systems Lab - Part1

26395 characters in 4243 words on 766 lines

Florian Moser

June 10, 2020

1 introduction

to improve performance need techniques throughout the stack

1.1 motivation

computing needs unlimited performance
many different algorithms
only small set of critical components (100s)

applications

scientific computing (climate/financial/molecular modeling)
consumer computing (compression, security)
embedded computing (sensors, robotics)
research (biomedics, imaging, computer vision)

critical functions

transforms (fourier, etc.)
matrix multiplication
filters/correlations/convolutions
equation solving
dense/sparse linear algebra
coders/decodes
graph algorithms

1.2 archive speedups

with same compiler, flags, operation count
archive 10x - 100x speedup

levels

algorithms (lower complexity)
software (manually optimize low level)
compiler (automatically improve low level)
microarchitecture (optimize for specific processor)

low level optimizations

adapt to memory layout
use vector instructions
use parallelism

achievable

20x through memory hierarchy (caches)
4x through vector instructions
4x through parallelization

optimized code in practice

heavy LoC count
needs manual intervention to build
reduces cache misses (potentially more operations)
uses vector instructions and parallelization
often not portable to other microarchitecture
may violate software engineering practices
some parts maybe generated automatically

1.3 naive matrix multiplication (NMM)

for n times n matrixes
 $O(n^3)$ operations
 $O(n^2)$ space

for optimized algorithms

vector instructions reduce number of instructions
but memory access stays asymptotic same
because just constants improved

for parallelized algorithms

may take p processors into account
like $O(n^3/p)$

limitations

asymptotic runtime only describes eventual trend
constants matter; unclear when "eventual" starts

2 basics

2.1 o-definitions

$O(f(n))$ is set of functions; f *element of* $O(g)$
 $O(g)$ (nicht wesentlich schnelleres wachstum)
 $\theta(g)$ (genauso schnelles wachstum)
 $\omega(g)$ (nicht wesentlich langsames wachstum)

order

$1 < \log \log(n) < \log(n) < \sqrt{n} < n$
 $n < n \log(n) < n^{1+e} < n^2 < n^m < 2^n < n!$

simplifications

$\log_2(n) \rightarrow \log_3(n)$ (base of logarithm irrelevant)
 $O(n + \log(n)) \rightarrow O(n)$ (take highest factor)
 $O(100n) \rightarrow O(n)$ (remove constants)
 $O(n^{1.1} + n \log(n)) \rightarrow O(n^{1.1})$
 $2n + O(\log(n))$ (keep, bc. outside bigger)
 $O(n) + \log(n) \rightarrow O(n)$ (only $O(n)$ relevant)
 $O(n^2 + m)$ (keep, bc. different numbers)

2.2 cost analysis

count number of relevant operations

cost measure $C(n)$

define what cost is composed of
for example $C(n) = (\text{adds}(n), \text{mults}(n))$
only account for mathematical operations
ignore computer mapping operations
like array access, index calculations

index operations

ignore if simple or dominated by expensive op
not the bottleneck in good processor (bc very common)
for example integer & floating point units separate
 $c[i*m + k] * c[i*n + f]$ (ok; dominated by mul)
 $c[i + 5k + 4m*m]$ (not ok; complex)
 $c[i/m \% k] + c[n]$ (not ok; + does not dominate / or %)

2.3 performance

equals cost/runtime

machine peak performance

max flops possible on that machine
useful to compare to archived flops
90% peak percentage possible for matrix multiplication
48 Gflops = 4 cores * 2 SSE * 1 add/mul * 3 GHz

performance NMM

$C(n) = (\text{fadds}(n), \text{fmults}(n)) = (n^3, n^3)$
for $n = 10$, cost = 2000, runtime = 1000, peak performance = 2
gives performance = 0.5 hence 25% peak performance

use performance counters

count cache misses, floating point operations
heavily OS/processor dependent
may not be available / faulty
like Intel PCM, Intel Vtune, Perfmon

instrument code manually

measure time at a specific place in code

2.4 operational intensity

$I(n) = W(n) / Q(n)$
for flops (floating point operations) $W(n)$
for bytes transferred (assuming empty cache) $Q(n)$

intensity factors

computer (memory bandwidth, peak performance)
implementation (degree of locality, SIMD)

measurement (warm cache, cache source, ...)

compute-bound ($I > 1$)

archives 80-90% of peak performance
computation uses same value often; work package in cache
cache miss time neglectable compared to computation
improve performance by keeping floating point units busy
reduce cache misses & register spills, use ILP & vectorization

memory bound ($I < 1$)

archives 40-50% of peak performance
computation needs many values; work package too big for cache
cache miss time bigger factor than computation time
improve performance by reducing memory traffic
reduce cache misses & register spills, use compression

2.5 operational intensity examples

$y = y + x \quad O(n) / O(n)$
 $y = Ax \quad O(n^2) / O(n^2)$
FFT $O(n \log n) / O(n)$
 $C = AB + C \quad O(n^3) / O(n^2)$

fma two array sum

$W(n) = 2n$ flops needed (add + mul per row)
 $Q(n) \geq 16n$ (need 2 n-arrays; hence 16n byte loads)
 $I \leq 1/8$; hence memory bound
 $n/2$ without vector ops (use 2 fma ports)
 $n/8$ with vector ops (do 4 at same time)
 $n/4$ L1 & L2 (do 8 loads, need 2 per op)
 $n/2$ L3 (do 4 loads, need 2 per op)

double matrix multiplication

$W(n) = 2(n^3)$ flops needed (add + mul per row/column combination)
 $Q(n) \geq 3 \cdot 8 \cdot (n^2)$ (need 3-n*n matrixes; as doubles)
 $I(n) \leq n/12$; hence compute bound
 $(n^3)/2$ without vector ops (use 2 fma ports)
 $(n^3)/8$ vector ops (do 4 at same time)
 $(n^2) \cdot 3/8$ L1 & L2 (do 8 loads, 3 matrixes of size n*n)
 $(n^2) \cdot 3/4$ L3 (do 4 loads, 3 matrixes of size n*n)

2.6 locality

algorithm use data near or equal to those used recently
more locality leads to better, more stable performance
for example by ensuring data traversed linearly (like $a[i][j]$)

temporal

recently referenced items referenced again
cycles through the same instructions

spatial

nearby addresses accessed consecutively
instructions referenced in sequence

3 processor

3.1 history

memory bandwidth increased over time
CPU speedup until 2003 (more GHz)
since 2003 cores & vector units added

3.2 intel history

fully backwards compatible (hence instructions never removed)
end of moores law means no longer free speedup for legacy code

x86-16

8086

x86-32

386

486

Pentium

Pentium MXX (adds MXX)

Pentium III (adds SSE; 4-way single)

Pentium 4 (adds SSE2; 2-way double)

Pentium 4E (adds SSE3)

x86-64

Pentium 4F

Penryn (adds SSE4)

Sandy Bridge (adds AVX; 8-way single, 4-way double) 32nm

Haswell (adds AVX2) 22nm

Skylake-V (adds AVX-512; 16-way single, 8-way double) 14nm

tick-tock-opt

shrink of same generation process technology as tick
like sandy bridge to ivy bridge
new generation microarchitecture as tock
like sandy bridge to haswell
optimization of same generation as opt
like skylake to kaby lake (since 2016)

vectorization generations

MMX (multimedia extension)

SSE (streaming SIMD extension)

AVX (advanced vector extensions)

super-scalar processors

issue/execute multiple instructions in one cycle
instructions taken from sequential instruction stream

3.3 ports

multiple ports available with different execution units
can activate one execution unit per port per cycle

instruction level parallelism (ILP)

requires super-scalar processor
intructions are parallelized automatically
by using different ports for different instructions
example $t_2 = t_0 + t_1$; t_0 , t_1 calculation can be parallelized

example haswell

for address generation as AGU, sub/add/bitwise as alu
0 with int alu/div, int vector alu/mul, fp fma/mul/div, branch
1 with int alu/mul, int vector alu, fp add/fma/mul, bit scan
2 AGU, loads
3 AGU, loads
4 stores
5 with int alu, vector shuffle, int vector alu, AES
6 int alu, branch
7 AGU

application haswell

n adds, n mults in $c \rightarrow n/2$ (use fma)
 n adds, n mults in assembly $\rightarrow n$ (cause only n adds)
 n adds in $c \rightarrow n/2$ (could map $n/2$ adds to fma)
 n adds in assembly $\rightarrow n$ (cause only n adds)
 n adds, $n/2$ mults in $c \rightarrow n/2$ ($n/2$ fma, $n/2$ adds)

3.4 memory

bottleneck

processor performance doubles every 18 months
but bus bandwidth only every 36 months
for haswell, peak performance at 2 ops/cycle FMA
needs 192 bytes/cycle, but bus bandwidth 16 bytes cycle

hierarchy

L0 registers (words)

L1 on-chip L1 cache SRAM (lines)

L2 on-chip L2 cache SRAM (lines)

L3 main memory DRAM (blocks)

L4 local secondary storage disks (files)

L5 remote secondary storage disks (tapes, servers)

from smaller, faster, costly

to larger, slower, cheaper

example haswell

16 fp registers

L1 with 4 latency, 12 throughput (8loads, 4stores)

L2 with 11 latency, 8 throughput

L3 with 34 latency, 4 throughput (non-uniform)

RAM with 125 latency, 2 throughput

3.5 virtual memory system

processor works on virtual addresses
but caches work with physical addresses
page size can be 4KB, 2MB, 1GB (page size step = 2^9)

x86 L1 cache

64 (2^6) block size

64 (2^6) sets

8 associativity

32 KB

x86 L1 loopup

can do concurrently with address translation
because location inside cache given by lower 12 bits
which are not part of page identifier (as page size $\geq 2^{12}$)

address translation

uses translation lookaside buffer (TLB)
first level 128 ITLB (instructions) and 64 DTLB (data) entries
second level (STLB) with 1536 entries
small penalty for ITLB/DTLB miss, (very) big one for STLB miss

vs L2 cache

assume working set of 2^{11} doubles with each on different page
fits into L2 cache fully, but TLB misses

MMM example

block-column A * whole B = block C
if full MMM & row-major order
only block C not contiguous (NB pages)
if leading dimensions added
then block B & C not contiguous (NB + K + NB pages)
copying to continuous memory may pay off

4 optimizations

4.1 compiler optimizations

optimizations compiler is likely to do

mapping program to machine

register allocation
code selection & ordering
dead code elimination
eliminating minor inefficiencies

inline procedure calls

reduce overhead by call, bounds checking, ...
need to compile source code together

code motion (precomputation)

reduce computation frequency of pure computation
like moving code out of loop (loop-invariant code motion)
for example $[i*n + j]$; precompute $i*n$

strength reduction

replace costly operation with cheaper one
like shift/add instead of multiply by 2
for example $[i*n + j]$
transform $i*n$ to addition in outer loop

share common subexpression (if recognised)

reuse portion of expressions
for example $[(i+1*n) + j] [(i*n) + j+1]$
replace with $[c + n]$ and $[c + 1]$ for $c = i*n + j$

4.2 compiler optimization blockers

even unrealistic scenarios must not change
analysis constrained (only static procedure-wide info)
can't test different choices (for ILP, locality)
hence algorithmic restructuring hard

procedure calls

treated as black box with potential side effects
hence can not simply reorder, precompute
compiler may detect pure (built-in) functions

memory aliasing

overlapping memory prevents reuse optimizations
like reducing loads/stores, precomputation
compiler may add case distinction for aliasing ($a+n < b \parallel b+n < a$)
and then use optimized code when no memory alias

4.3 manual optimizations

algorithm

restructure for ILP, locality
use simple data representations; best are arrays
watch inner-most loop closely for overhead
unroll loops to enable other optimizations
try out different algorithms

optimize procedure calls

reduce overhead when calling procedures
precompute manually (taking out of loop & similar)
provide source code instead of linking (to enable inlining)

scalar replacements (avoid aliasing)

copy reused elements in local variables
structure algorithm in load, compute, store

unroll loops

unroll loop by some factor
do scalar replacement to save on loads
measure outcome and autotune unrolling factor

generate code

capture algorithm with domain specific language (DSL)
decide on choices with brute force/machine learning
generate easily transformable code to profit from compiler

tell compiler no aliasing

use compiler flag `-fno-alias`
use `#pragma ivdep`
pass argument with restrict keyword

4.4 loop unrolling

more work inside single loop iteration
faster iterator incrementation
but just unrolling usually no advantage
need to transform (reassociation, dependency reduction)

example

create two accumulators `i0, i1`
sum up `i` and `i+1` into respective accumulator
sum up `i0 + i1` in the end

compilers

transformations may break associativity
hence compiler not allowed to do so
many different forms of how to unroll
hence compiler search space too big

generalization

use K accumulators and L unrolls (for K divides L ; $L \geq K$)
choose K/L combination with best performance
but processor-specific hence often unportable
called autotuning, processor-tuning

approaches

try out K, L then measure and choose best
but large overhead for small inputs
pick best K, L using latency/throughput model
but models may not work in practice

4.5 loop unrolling examples

based on sum of array

trivial

for (entry in array) `sum += entry`;
each computation pays full latency
because always dependency on operation before

loop unrolling

for (entry, next_entry in array) `sum = (sum + entry) + next_entry`;

loop unroll, separate accumulator

for (entry, next_entry in array) `sum1 += entry`; `sum2 += next_entry`
now dependencies reduced by half
but for floats behaviour changes
for integer fine bc. overflow preserves associativity

loop unroll for mul

$\text{ceil}(5 \text{ lat}/0.5 \text{ gap})$ operations need independence
hence choose $L=K=10$

loop unroll for add

$\text{ceil}(3 \text{ lat}/1 \text{ gap})$ operations need independence
hence choose $L=K=3$
but best performance measured at $L=K=8$

loop unroll for add using fma

$\text{ceil}(5 \text{ lat}/0.5 \text{ gap})$ operations need independence
another 2x speedup bc one additional port used

5 instruction set architecture (ISA)

defines instruction set, registers
implemented with microarchitecture
which then defines caches, frequency, virtual memory

5.1 instructions

executes logic upon values in registers

flynns taxonomy

single/multiple instructions(S/MI)

per single/multiple data (S/MD)

SISD for mainframes, old PCs with single simple core

SIMD for GPUs, CPUs with vector instructions

MISD for pipelines (like space shuttle)

MIMD for scientific purposes

single instruction multiple data (SIMD)

for parallel computation on short int/float vectors

allow to compute on up to 8 numbers at the same time

useful (many applications have fine-grained parallism)

doable (easy to design, simply replicate units)

SIMD history

MMX for integers / 64bit (pentium MMX)

SSE for 4 singles / 128bit (pentium III)

introduces xmm0-xmm15

SSE2 for 2 doubles / 128bit (pentium 4)

AVX for 4 doubles / 256bit (sandy bridge)

introduces ymm0-ymm15; three-operand instructions

AVX2 for FMAs (haswell)

AVX-512 for 8 doubles / 512bit (skylake-x)

introduces zmm0-zmm31

registers

all instructions (including scalar) use same registers

hence xmm0 points to the same physical address as ymm0

instruction names

of the form operation-size-type

operation like add, mov, mul

size either single (s) or packed (p)

hence operate on single element or all

type either single (s) or double (d)

hence operate on 4 or 8 bytes

like addps, movss, mulpd

fused multiply add (FMA)

add & multiply in same instruction ($x + y*z$)

better accuracy bc only one rounding step

natural pattern in many algorithms

needs three operand instructions (used to be hard to build)

performance

latency is time to wait until result ready

throughput is #operations at the same time

gap is 1/throughput (intel calls this "throughput")

example haswell

fma has latency 5, throughput 2 (gap 1/2)

mul has latency 5, throughput 2 (gap 1/2)

add has latency 3, throughput 1 (gap 1/1)

same for vector instructions (except div)

each register has space for 256 bits (4 doubles, 8 ints)

6 vectorization

use SIMD instructions to parallelize computations

6.1 implementation options

vectorized library (but seldom available)

compiler flags (but seldom works)

intrinsics (but limited optimizations possible)

assembly (but no help anymore from compiler)

6.2 compiler vectorization

options

ensure data always aligned and use aligned loads

use unaligned loads

execute first entries unvectorized until aligned ("peel")

decision tree

if a,b aliased then unvectorized code

else if a,b aligned then vectorized

else peeling & aligned loads or unaligned loads

but impractical on large code bases

detect vectorization

annotation flags of compiler

generate vectorization report

improve

prevent aliasing with flags, pragmas or restrict keyword

avoid branches like switch, goto, return (straight-line code)

terminate loop by constant, linear function, loop invariant

inner loop with consecutive, dependencyfree access

access elements directly (with loop variable, index)

use arrays, not structs / pointers

use #pragma vector aligned or __assume_aligned(vec, 32)

6.3 intrinsics

C functions translating to specific assembly

intel intrinsics guide for detailed specification

same instruction in parallel easy & cheap (like add)

crossing lines difficult & expensive (like shuffle)

as an abstraction layer

need to explicitly load/store operated variables

need to have data aligned to 256bit (32bytes)

parameter variables not overwritten by operation

(not even if assembly would, like fma)

flavors

SSE with 4-way floats, 2-way doubles

%xmm0-%xmm15 128bits registers

AVX with 8-way floats, 4-way doubles

%ymm0-%ymm15 256bits registers

AVX-512 with 16-way floats, 8-way doubles

%zmm0-%zmm31 512bits registers

mixing flavors sets costly

bc need to save/restore upper bits

function naming

.<prefix>.<operation>_.<suffix>(<data>)

prefix like _mm, _mm256, *mm512*

operation like load, add, store

suffix like size-type of instruction

data like _m128, _m256, _m512 or naive types

like _mm256_load_pd(4.0,3.0,2.0,1.0)

types

native (one-to-one to assembly) like _mm256_load_pd()

multi (maps to several) like _mm256_set_pd()

macros (simplify code) like _MM_SHUFFLE()

data types

_m256, for float[4]

_m256d for double[2]

_m256i for char[16], short int[8], int[4] or uint64_t[2]

loads

load_pd(pointer) → pointers must always be 32bit aligned

loadu_pd(unaligned pointer; used to be more expensive)

loadh_pd(_m128d t, unaligned pointer p) → [t1, p]

maskload_pd(pointer, _m256i mask with 0x0 and 0xFFFF)

broadcast_sd(one double repeated to fill line)

boradcast_pd(two doubles repeated; for complex numbers)

i64gather_pd(pointer, *m256i* offsets, scale; 8 for doubles)

sets

set1_sd(3.0) → [3.0, 3.0, 3.0, 3.0]

set_pd(4.0, 3.0, 2.0, 1.0) → [4.0, 3.0, 2.0, 1.0]

setr_pd(4.0, 3.0, 2.0, 1.0) → [1.0, 2.0, 3.0, 4.0]

set_sd(4.0) → [0.0, 0.0, 0.0, 4.0]

setzero_pd() → [0.0, 0.0, 0.0, 0.0]

set_m128d(128bit low, 128bit high) → [low, high]

arithmetic

add, subb, mul, div, sqrt

max, min, ceil, floor, round

addsub(a, b) → [a1+b1, a2-b2, a3+b3, a4-b4]

hadd(a, b) → [a1+a2, b1+b2, a3+a4, b3+b4] (also hsub)

fmadd(a, b, c) → [a1*b1 + c1, ...] (also fmsub, fmaddsub)

cmp(a, b) → [if a1=b1 then 0xFF else 0x00, ...]

dp(a, b, int *mask_{sum}* |mask_result) for dotproduct

comparison

cmp_pd(a, b, mode) for mode = <operation>_.<ordering>

operation is EQ (=), GE (\geq), GT ($>$), LE (\leq), LT ($<$)

to negate, prefix operation with N

ordering is OQ (ordered; includes NaN), UQ (unordered; ignores NaN)

like cmp(a,b, _CMP_EQ_OQ), cmp(a,b, _CMP_NEQ_UQ)

special operations are _CMP_TRUE_UQ, _CMP_ORD_Q (and vice-versa)

result is 0xFFFF when fulfilled, and 0x0 otherwise

conversions

epi32 for 32bit integers

cvtepi32_pd(_m128i a) → _m256d (4 ints to doubles)

cvtepi32_ps(_m256i a) → _m256 (8 ints to floats)

cvtpd_epi32(_m256d a) → _m128i (4 doubles to ints)

cvtpd_epi32(_m256 a) → _m256i (8 floats to ints)

cvtps_pd(_m128 a) → _m256d (4 floats to doubles)

cvtpd_ps(_m256d a) → _m128 (4 doubles to floats)

cvttpd_epi32(_m256d a) → _m128i (4 doubles to ints)

cvtsd_f64(_m256d a) → double (first double extract)

cvtss_f32(_m256 a) → float (first float extract)

shuffles

unpackhi_pd(a, b) → [a2, b2, a4, b4]

unpacklo_pd(a, b) → [a1, b1, a3, b3]

movemask_pd(a) → 4-bit int; bit[i] = MSB of a[i]

movedup_pd(a) → [a1, a1, a3, a3]

blend_pd(a, b, mask) → [a1 or b1, a2 or b2, ...]

blendv_pd(a, b, (variable) mask) → [a1 or b1, a2 or b2, ...]

insertf128_pd(a, b, 0 or 1) → [b1, b2, a3, a4] or [a1, a2, b1, b2]

extractf128_pd(a, 0 or 1) → [a1, a2, 0, 0] or [a3, a4, 0, 0]

shuffle_pd(a, b, mask) → [a1 or a2, b1 or b2, a3 or a4, b3 or b4]

permute_pd(a, mask) → [a1 or a2, a1 or a2, a3 or a4, a3 or a4]

permute4x64_pd(a, mask) → [any a*, any a*, ...] (but 128bit cross)

permute2f128_pd(a, b, mask) → [a1 or a2 or b1 or b2 or 0, ...]

stores

store_pd(pointer); pointer must be 32bit aligned

storeu_pd(unaligned pointer); used to be more expensive

maskstore_pd(pointer, _m256i mask with 0x0 and 0xFFFF)

storeu2_m128d(low bits pointer, high bits pointer)

stream_pd(pointer) (no caching)

6.4 intrinsic examples

add own index to array x

(for i < x.size(); i+=4)

vec = load_pd(x+i)

index = set_pd(i+3, i+2, i+1, i)

vec = add_pd(vec, index)

store_pd(x+i, vec)

add own index to array x

index = set_pd(3, 2, 1, 0)

increment = set1_pd(4)

(for i < x.size(); i+=4)

vec = load_pd(x+i)

vec = add_pd(vec, index)

store_pd(x+i, vec)

index = add_pd(index, increment)

add 1 if negative, else add -1

ones = set1_pd(1)

negatives = set1_pd(-1)

comparator = set1_pd(0)

(for i < size; i+=4)

values = load_pd(x+i)

comparison = cmp_pd(values, comparator, _CMP_GE_OQ)

addition = blendv_pd(ones, negatives, comparison)

result = add_pd(values, addition)

store_pd(x+i, result)

load 4 from arbitrary memory

v1 = loadu_pd(p1), v2 = loadu_pd(p2)

v3 = loadu_pd(p1-2), v2 = loadu_pd(p2-2)

v1v2 = unpacklo_pd(v1, v2)

v3v4 = unpacklo_pd(v3, v4)

v = blend_pd(v1v2, v3v4, 0b0011)

load 4 from arbitrary memory with *set_pd*

v = set_pd(p4, p3, p2, p1)

reverse engineered the same as

v1 = _mm_load_sd(p1), v3 = _mm_load_sd(p3)

v1v2 = _mm_loadh_pd(v1, p2), v3v4 = _mm_loadh_pd(v3, p4)

v1v2d = _mm256_castpd128_pd256(v1v2)

v = _mm256_insertf128_pd(v1v2d, v3v4, 1)

7 memory

7.1 cache

memory with short access time

for frequently or recently used data/instructions

supports temporal locality (by definition)

may supports spatial locality (with lines, blocks)

general mechanics

data is copied/replaced in block units (intel core 64bytes)

faster cache has subset of blocks of larger memory

placement policy determines where block is placed

replacement policy determine which block is replaced

type of cache misses

compulsory, cold (on first access)

capacity (working set larger than cache)

conflict (many blocks map to same slot)

cache types

direct mapped (E=1); each block has unique location

associative (E>1, S>1); each block has many possible location

fully associative (S=1); each block to any location

replacement strategy

for associative caches (multiple possible locations)

defines replacement if all possible locations full

for example least recently used (LRU)

cache organisation

E as lines per set (associativity)

B for #blocks

S for #sets

cache block is the saved data

cache line = valid bit, tag + cache block

performance metrics

miss rate (references not in cache; misses/access)

hit rate (references in cache; 1-miss rate)

hit time (deliver block from cache to CPU; 3 cycles for L1)

miss penalty (time required for a miss; 100 cycles for L2)

writes to memory

on cache hit, defer until line replaced or write immediately

called write-back or write-through

first one used by intel; useful for many algorithms

on cache miss, first load cache or write directly

called write-allocate or no-write-allocate

first one used by intel; whole cache line loaded not just single element

7.2 examples

simple

for E=1, B=4 double, S=8

lower 3 bits 0 because doubles are 8 bytes

3 bit determine sets, 2 bits for block

data identified by 56 bit tag in single possible location

conflict

for E=1, B=2 double, S=8

loop with a[j][i] for n = 16

hence a[2][0] conflicts with a[0][0]

examples

for E=1, B=2 double, S=2

access 0123456701234567

8 miss, 8 hits; spatial yes, temporal no

access 0246135702461357

16 miss; spatial no, temporal no

access 0123012301230123

4 miss, 12 hits; spatial yes, temporal yes; optimal

7.3 traffic examples

vector addition ($z = x + y$)

memory traffic 4n doubles = 32n (bc measured in bytes)

operational intensity 1/4 (constant; this is bad)

naive matrix multiplication

for cache size $y \ll n$, block size 8

gives $n^2 * 9n/8$ cache misses

blocked matrix multiplication

for block size b such that $3b^2 \leq$ cache size y

gives $(n/b)^2 * nb/4$ cache misses

$3b^2$ bc 3 little matrix need to fit into cache

strided working sets problem

common in FFT, 2d transformations

assume two loops for ($i < n$; $i += t$) for t stride

see how locality changes with different strides

if stride = 1 then spatial full, temporal if $W \leq C$

if stride = 2 then spatial $B/2$, temporal if $W \leq 2C$

if stride = 4 then spatial $B/4$, temporal if $W \leq 4C$

(if stride not power of two not that much of an issue)

need to separate computation in chunks

need to copy data to better memory layout

7.4 memory dependencies

read after write (RAW)

also called true dependency

$r1 = r3 + r4$; $r2 = 2 * r1$

write after read (WAR)

also called antidependency

$r1 = r2 + r3$; $r2 = r4 + r5$

dependency on $r2$ only by name

write after write (WAW)

also called output dependency

$r1 = r2 + r3$; $r2 = r4 + r5$

dependency on $r2$ only by name

resolve by renaming

with single static assignment (SSA) code style

automatically by compiler (but black box)

hardware can do dynamic register renaming

needs separation of architectural (assembly) / physical layer

needs more physical than architectural registers

8 roofline plot

way to plot performance vs operational intensity

visualizes how well the algorithm is optimized to machine

x log axis is $I(n) = W(n) / Q(n)$

y log axis is $P(n) = W(n) / T(n)$

8.1 data

work $W(n)$ in [flops]

data movement $Q(n)$ in [bytes]

runtime $T(n)$ in [cycles]

8.2 roofs

peak performance (pi)

in [flops/cycle], per core

can never be faster than peak performance

draw horizontal line at $y = \text{peak performance}$

memory bandwidth (beta)

in [bytes/cycle], per memory layer

use stream benchmark to measure (could be conservative)

measured values up to 60% of theoretical possible from manual

$B \geq Q / T = P / I \Rightarrow \log(P) \leq \log(I) + \log(B)$

$y=x$ line with offset to cut peak performance root at π/B

interpretation

left intersection β/π memory bound, right compute bound

8.3 observations

introducing SIMD

increases π roof \Rightarrow more memory bound

bc can execute many operations in parallel

different algorithmic operation mix

lowers π roof \Rightarrow more compute bound

bc different available units/ports

introducing parallelism

increases performance, I ideally stays the same

because same work/data movement should stay same

warmup cache

I gets huge for small problem sizes

for large no effect bc problem does fit in cache

8.4 examples

get lower bound of Q with cold misses

$y = ax + y$ (in doubles)

$W = 2n$

$Q = 16n$ (read) + $8n$ (writes) = $24n$

$I = 1/12$

in plot, hits capacity line

for all inputs sizes at the same point

for parallel, hits increased capacity line

$y = Ax + y$

$W = 2n^2$

$Q \geq 8n^2$ (+ $24n$)

$I = 1/4$

in plot, nearly hits capacity line

for larger inputs sizes gets closer to memory line (vertical)

for parallel, hits increased capacity line

$C = AB + C$

$W = 2n^3$

$Q \geq 32n^2$ (\geq bc capacity misses for large matrixes)

$I \leq n/16$

in plot, nearly hits peak performance

for small inputs I increases (as long as cache available)

for large inputs, I only slightly smaller (bc of blocked implementation)

for parallel, nearly hits increased peak performance

for parallel, gets closer to hit capacity line

good algorithms are compute bound

trivial triple loop is memory bound

9 compilers

9.1 gcc

no optimizations on as default

flags

-O2, -O3 (group of optimization flags)

-march=native (architecture)

-mAVX (use avx instructions)

-m64 (64 bit architecture)

matrix multiplication example

1300 cycles at -O0

150 cycles at -O3 -m64 -march