Changelog

Changes made in this version not seen in first lecture:

- 15 November: vector addr picture: make order of result consistent with order of inputs
- 15 November: correct `square` to `matmul` on several vector slides
- 15 November: correct mixups of A and B, B and C on several matmul vector slides
- 15 November: correct some si128 instances to si256 in vectorization slides
- 15 November: addressing transformation: correct more A/B/C mixups

Vector Insts / Profilers / Exceptions intro

last time

loop unrolling/cache blocking

instruction queues and out-of-order

- list of available instructions
- multiple *execution units* (ALUs + other things that can run instr.)
- each cycle: ready instructions from queue to execution units

reassociation

- reorder operations to reduce data dependencies, expose more parallelism

multiple accumulators — reassociation for loops

shifting bottlenecks

- need to optimize what's slowest — determines longest latency
- e.g. loop unrolling helps until parallelism limit
- ..., but after improving parallelism, loop unrolling more helps again
- e.g. cache optimizations won't matter until loop overhead lowered or

aliasing problems with cache blocking

```
for (int k = 0; k < N; k++) {
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < N; j += 2) {
      C[(i+0)*N + j+0] += A[i*N+k] * B[k*N+j];
      C[(i+1)*N + j+0] += A[(i+1)*N+k] * B[k*N+j];
      C[(i+0)*N + j+1] += A[i*N+k] * B[k*N+j+1];
      C[(i+1)*N + j+1] += A[(i+1)*N+k] * B[k*N+j+1];
    }
  }
}
```

can compiler keep `A[i*N+k]` in a register?

“register blocking”

```
for (int k = 0; k < N; ++k) {
  for (int i = 0; i < N; i += 2) {
    float Ai0k = A[(i+0)*N + k];
    float Ai1k = A[(i+1)*N + k];
    for (int j = 0; j < N; j += 2) {
      float Bkj0 = A[k*N + j+0];
      float Bkj1 = A[k*N + j+1];
      C[(i+0)*N + j+0] += Ai0k * Bkj0;
      C[(i+1)*N + j+0] += Ai1k * Bkj0;
      C[(i+0)*N + j+1] += Ai0k * Bkj1;
      C[(i+1)*N + j+1] += Ai1k * Bkj1;
    }
  }
}
```

vector instructions

modern processors have registers that hold "vector" of values

example: current x86-64 processors have 256-bit registers

- 8 ints or 8 floats or 4 doubles or ...

256-bit registers named `%ymm0` through `%ymm15`

instructions that act on all values in register

- vector instructions or SIMD (single instruction, multiple data) instructions

extra copies of ALUs only accessed by vector instructions

(also 128-bit versions named `%xmm0` through `%xmm15`)

example vector instruction

`vpaddd %ymm0, %ymm1, %ymm2` (packed add dword (32-bit))

Suppose registers contain (interpreted as 4 ints)

%ymm0: [1, 2, 3, 4, 5, 6, 7, 8]
%ymm1: [9, 10, 11, 12, 13, 14, 15, 16]

Result will be:

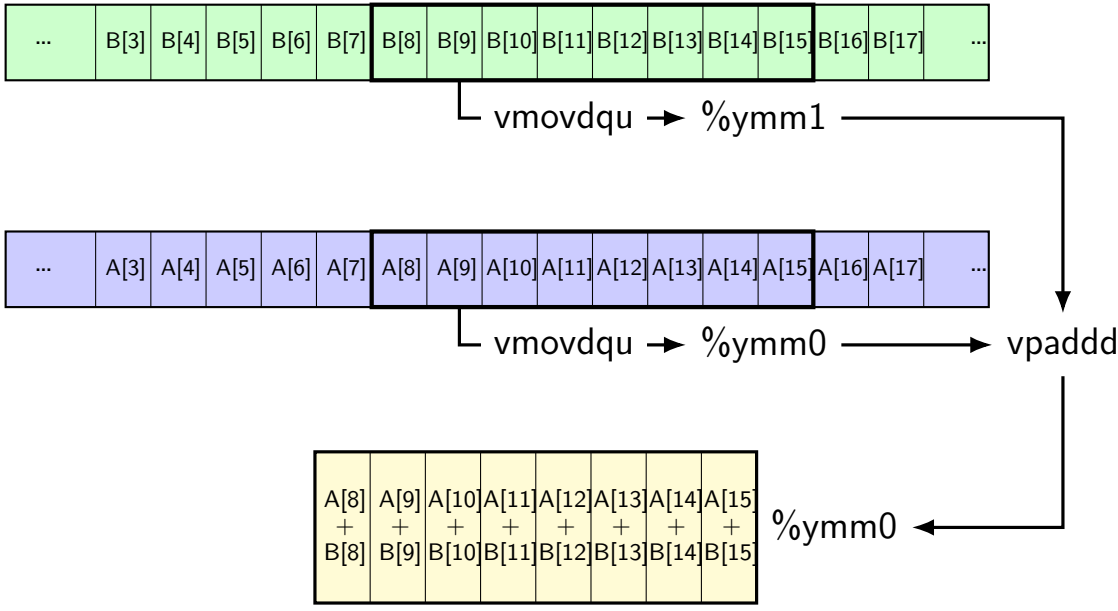
%ymm2: [10, 12, 14, 16, 18, 20, 22, 24]

vector instructions

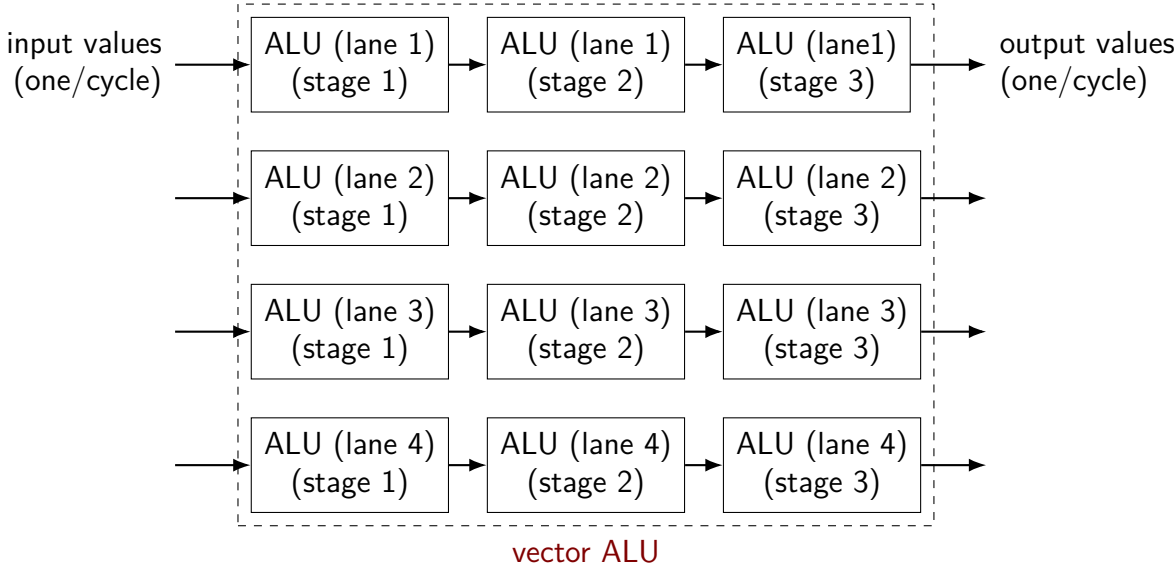
```
void add(int * restrict a, int * restrict b) {
    for (int i = 0; i < 512; ++i)
        a[i] += b[i];
}
```

```
add:
  xorl %eax, %eax
the_loop:
  vmovdqu (%rdi,%rax), %ymm0  /* load A into ymm0 */
  vmovdqu (%rsi,%rax), %ymm1  /* load B into ymm1 */
  vpaddd %ymm1, %ymm0, %ymm0  /* ymm1 + ymm0 -> ymm0 */
  vmovdqu %ymm0, (%rdi,%rax)  /* store ymm0 into A */
  addq $32, %rax              /* increment index by 32 bytes */
  cmpq $2048, %rax
  jne the_loop
  vzeroupper        /* <-- for calling convention reasons */
  ret
```

vector add picture



one view of vector functional units



why vector instructions?

lots of logic not dedicated to computation

- instruction queue
- reorder buffer
- instruction fetch
- branch prediction
- ...

adding vector instructions — little extra control logic

...but a lot more computational capacity

vector instructions and compilers

compilers can sometimes figure out how to use vector instructions

(and have gotten much, much better at it over the past decade)

but easily messsed up:

by aliasing

by conditionals

by some operation with no vector instruction

...

fickle compiler vectorization (1)

GCC 8.2 and Clang 7.0 generate vector instructions for this:

```
#define N 1024
void foo(unsigned int *A, unsigned int *B) {
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                B[i * N + j] += A[i * N + k] * A[k * N + j];
}
```

but not:

```
#define N 1024
void foo(unsigned int *A, unsigned int *B) {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                B[i * N + j] += A[i * N + k] * A[j * N + k];
}
```

fickle compiler vectorization (2)

Clang 5.0.0 generates vector instructions for this:

```
void foo(int N, unsigned int *A, unsigned int *B) {
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                B[i * N + j] += A[i * N + k] * A[k * N + j];
}
```

but not: (fixed in later versions)

```
void foo(long N, unsigned int *A, unsigned int *B) {
    for (long k = 0; k < N; ++k)
        for (long i = 0; i < N; ++i)
            for (long j = 0; j < N; ++j)
                B[i * N + j] += A[i * N + k] * A[k * N + j];
}
```

vector intrinsics

if compiler doesn't work...

could write vector instruction assembly by hand

second option: "intrinsic functions"

C functions that compile to particular instructions

vector intrinsics: add example

```
void vectorized_add(int *a, int *b) {
  for (int i = 0; i < 128; i += 8) {
     // "si256" --> 256 bit integer
     // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
     __m256i a_values = _mm256_loadu_si256((__m256i*) &a[i]);
     // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
     __m256i b_values = _mm256_loadu_si256((__m256i*) &b[i]);

     // add four 32-bit integers
     // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
     __m256i sums = _mm256_add_epi32(a_values, b_values);

     // {a[i], a[i+1], a[i+2], a[i+3]} = sums
     _mm256_storeu_si256((__m256i*) &a[i], sums);
  }
}
```

vector intrinsics: add example

special type `__m256i` — "256 bits of integers"
other types: `__m256` (floats), `__m128d` (doubles)

```
void vec
  for (int i = 0; i < 128; i += 8) {
    // "si256" --> 256 bit integer
    // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
    __m256i a_values = _mm256_loadu_si256((__m256i*) &a[i]);
    // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
    __m256i b_values = _mm256_loadu_si256((__m256i*) &b[i]);

    // add four 32-bit integers
    // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
    __m256i sums = _mm256_add_epi32(a_values, b_values);

    // {a[i], a[i+1], a[i+2], a[i+3]} = sums
    _mm256_storeu_si256((__m256i*) &a[i], sums);
  }
}
```

vector intrinsics: add example

functions to store/load

`si256` means "256-bit integer value"

u for "unaligned" (otherwise, pointer address must be multiple of 32)

```
v
        // "si256" --> 256 bit integer
        // a_values = {a[i], a[i+1], a[i+2], a[i+3]}
        __m256i a_values = _mm256_loadu_si256((__m256i*) &a[i]);
        // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
        __m256i b_values = _mm256_loadu_si256((__m256i*) &b[i]);

        // add four 32-bit integers
        // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
        __m256i sums = _mm256_add_epi32(a_values, b_values);

        // {a[i], a[i+1], a[i+2], a[i+3]} = sums
        _mm256_storeu_si256((__m256i*) &a[i], sums);
    }
}
```

vector intrinsics: add example

```
void vectorized_add(int *a, int *b) {
  for (int i = 0; i < 128; i += 8) {
    // "si256"
    // a_values
    __m256i a_v                                    _m256i*) &a[i]);
    // b_values = {b[i], b[i+1], b[i+2], b[i+3]}
    __m256i b_values = _mm256_loadu_si256((__m256i*) &b[i]);

    // add four 32-bit integers
    // sums = {a[i] + b[i], a[i+1] + b[i+1], ....}
    __m256i sums = _mm256_add_epi32(a_values, b_values);

    // {a[i], a[i+1], a[i+2], a[i+3]} = sums
    _mm256_storeu_si256((__m256i*) &a[i], sums);
  }
}
```

function to add

epi32 means “8 32-bit integers”

vector intrinsics: different size

```
void vectorized_add_64bit(long *a, long *b) {
  for (int i = 0; i < 128; i += 4) {
    // a_values = {a[i], a[i+1], ...} (4 x 64 bits)
     __m256i a_values = _mm256_loadu_si256((__m256i*) &a[i]);
    // b_values = {b[i], b[i+1], ...} (4 x 64 bits)
     __m256i b_values = _mm256_loadu_si256((__m256i*) &b[i]);
     // add four 64-bit integers: vpaddq %ymm0, %ymm1
     // sums = {a[i] + b[i], a[i+1] + b[i+1], ...}
     __m256i sums = _mm256_add_epi64(a_values, b_values);
     // {a[i], a[i+1]} = sums
     _mm256_storeu_si256((__m256i*) &a[i], sums);
  }
}
```

vector intrinsics: different size

```
void vectorized_add_64bit(long *a, long *b) {
  for (int i = 0; i < 128; i += 4) {
    // a_values = {a[i], a[i+1], ...} (4 x 64 bits)
     __m256i a_values = _mm256_loadu_si256((__m256i*) &a[i]);
    // b_values = {b[i], b[i+1], ...} (4 x 64 bits)
     __m256i b_values = _mm256_loadu_si256((__m256i*) &b[i]);
     // add four 64-bit integers: vpaddq %ymm0, %ymm1
     // sums = {a[i] + b[i], a[i+1] + b[i+1], ...}
     __m256i sums = _mm256_add_epi64(a_values, b_values);
     // {a[i], a[i+1]} = sums
     _mm256_storeu_si256((__m256i*) &a[i], sums);
  }
}
```

128-bit version, too

history: 256-bit vectors added in extension called AVX (c. 2011)

before: 128-bit vectors added in extension called SSE (c. 1999)

128-bit intrinsics exist, too:

- `__m256i` becomes `__m128i`
- `__mm256_add_epi32` becomes `__mm_add_epi32`
- `__mm256_loadu_si256` becomes `__mm_loadu_si128`

intrinsics in assignments

smooth assignment: you will use instriniscs

disabled compiler vectorization

goal: you understand how vectorization optimization works

goal: in case you needed to do more than compiler would do

missing "pattern" for how to use vectors, aliasing, code size tradeoffs, ...

matrix multiply

```
void matmul(unsigned int *A, unsigned int *B, unsigned int *C)
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                C[i * N + j] += A[i * N + k] * B[k * N + j];
}
```

(simple version, no cache blocking, no avoiding aliasing beteeen C, B, A,...)

matmul unrolled

```
void matmul(unsigned int *A, unsigned int *B, unsigned int *C) {
  for (int k = 0; k < N; ++k) {
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; j += 8) {
        /* goal: vectorize this */
        C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
        C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
        C[i * N + j + 2] += A[i * N + k] * B[k * N + j + 2];
        C[i * N + j + 3] += A[i * N + k] * B[k * N + j + 3];
        C[i * N + j + 4] += A[i * N + k] * B[k * N + j + 4];
        C[i * N + j + 5] += A[i * N + k] * B[k * N + j + 5];
        C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
        C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
      }
}
```

(NB: would probably also want to do cache blocking...)

handy intrinsic functions for matmul

`_mm256_set1_epi32` — load eight copies of a 32-bit value into a 128-bit value

- instructions generated vary; one example: `vmovd` + `vpbroadcastd`

`_mm256_mullo_epi32` — multiply eight pairs of 32-bit values, give lowest 32-bits of results

- generates `vpmulld`

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements from C
Cij = _mm256_loadu_si256((__m256i*) &C[i * N + j + 0]);
... // manipulate vector here
// store eight elements into C
_mm_storeu_si256((__m256i*) &C[i * N + j + 0], Cij);
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements from B
Bkj = _mm256_loadu_si256((__m256i*) &B[k * N + j + 0]);
... // multiply each by B[i * N + k] here
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements starting with B[k * n + j]
Bkj = _mm256_loadu_si128((__m256i*) &B[k * N + j + 0]);
// load four copies of A[i * N + k]
Aik = _mm256_set1_epi32(A[i * N + k]);
// multiply each pair
multiply_results = _mm256_mullo_epi32(Aik, Bkj);
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
Cij = _mm256_add_epi32(Cij, multiply_results);
// store back results
_mm256_storeu_si256(..., Cij);
```

matmul vectorized

```
__m256i Cij, Bkj, Aik, Aik_times_Bkj;

// Cij = {C_{i,j}, C_{i,j+1}, C_{i,j+2}, ..., C_{i,j+7}}
Cij = _mm256_loadu_si256((__m256i*) &C[i * N + j]);
// Bkj = {B_{k,j}, B_{k,j+1}, B_{k,j+2}, ..., B_{k,j+7}}
Bkj = _mm256_loadu_si256((__m256i*) &B[k * N + j]);

// Aik = {A_{i,k}, A_{i,k}, ..., A_{i,k}}
Aik = _mm256_set1_epi32(A[i * N + k]);

// Aik_times_Bkj = {A_{i,k} × B_{k,j}, A_{i,k} × B_{k,j+1}, A_{i,k} × B_{k,j+2}, ..., A_{i,k} × B_{k,j+7}}
Aik_times_Bkj = _mm256_mullo_epi32(Aij, Bkj);

// Cij= {C_{i,j} + A_{i,k} × B_{k,j}, C_{i,j+1} + A_{i,k} × B_{k,j+1}, ...}
Cij = _mm256_add_epi32(Cij, Aik_times_Bkj);

// store Cij into C
_mm256_storeu_si256((__m256i*) &C[i * N + j], Cij);
```

moving values in vectors?

sometimes values aren't in the right place in vector

example:

have: [1, 2, 3, 4]

want: [3, 4, 1, 2]

there are instructions/intrinsics for doing this

called shuffling/swizzling/permute/...

sometimes might need combination of them

worst-case: could rearrange on stack..., I guess

example shuffling operation (1)

goal: [1, 2, 3, 4] to [3, 4, 1, 2] (64-bit values)

```
/* x = {1, 2, 3, 4} */
__m256i x = _mm256_setr_epi64x(1, 2, 3, 4);
__m256i result = _mm256_permute4x64_epi64(
        x,
        /* index 2, then 3, then 0, then 1 */
        2 | (3 << 2) | (0 << 4) | (1 << 6)
        /* could also write _MM_SHUFFLE(1, 0, 3, 2) */
    );
/* result = {3, 4, 1, 2} */
```

256-bit with 128-bit?

Intel designed 256-bit vector instructions with 128-bit ones in mind

goal: possible to use 128-bit vector ALUs to implement 256-bit instructions

- split 256-bit instruction into two ALU operations

means less instructions move values from top to bottom half of vector

- in particular, complicated to move 16-bit value between halfs

aside on AVX and clock speeds

some processors ran slower when 256-bit ALUs are being used

- includes a lot of notable Intel CPUs

why? they give out heat — can't maintain higher clock speed

- for energy reasons, shut down when not used

still faster *assuming you're using vectors a lot*

alternate vector interfaces

intrinsics functions/assembly aren't the only way to write vector code

e.g. GCC vector extensions: more like normal C code

- types for each kind of vector
- write + instead of `_mm_add_epi32`

e.g. CUDA (GPUs): looks like writing multithreaded code, but each thread is vector "lane"

other vector instructions

multiple extensions to the X86 instruction set for vector instructions

first version: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2

- 128-bit vectors

this class: AVX, AVX2

- 256-bit vectors

not this class: AVX-512

- 512-bit vectors

also other ISAs have these: e.g. NEON on ARM, MSA on MIPS, AltiVec/VMX on POWER, ...

other vector instructions features

SSE pretty limiting

other vector instruction sets often more featureful:

(and require more sophisticated HW support)

better conditional handling

better variable-length vectors

ability to load/store non-contiguous values

some of these features in AVX512

optimizing real programs

spend effort where it matters

e.g. 90% of program time spent reading files, but optimize computation?

e.g. 90% of program time spent in routine A, but optimize B?

profilers

first step — tool to determine where you spend time

tools exist to do this for programs

example on Linux: `perf`

perf usage

sampling profiler

stops periodically, takes a look at what's running

`perf record OPTIONS program`

example OPTIONS:

`-F 200` — record 200/second

`--call-graph=lbr` — record stack traces (using method "lbr")

`perf report` or `perf annotate`

children/self

“children” — samples in function or things it called

“self” — samples in function alone

demo

other profiling techniques

count number of times each function is called

not sampling — exact counts, but higher overhead

- might give less insight into amount of time

tuning optimizations

biggest factor: how fast is it actually

setup a benchmark

- make sure it's realistic (right size? uses answer? etc.)

compare the alternatives

addressing efficiency

```
for (int k = 0; k < N; k += 2) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        Cij += A[i * N + k] * B[k * N + j];
      }
      C[i * N + j] = Cij;
    }
  }
}
```

tons of multiplies by N??

isn't that slow?

addressing transformation

```
for (int kk = 0; k < N; kk += 2)
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      float *Bkj_pointer = &B[kk * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        // Bij += A[i * N + k] * A[k * N + j~];
        Bij += A[i * N + k] * Bkj_pointer;
        Bkj_pointer += N;
      }
      C[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will usually do this!

increment/decrement by N ($\times$ sizeof(float))

addressing transformation

```
for (int kk = 0; k < N; kk += 2)
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Cij = C[i * N + j];
      float *Bkj_pointer = &B[kk * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        // Bij += A[i * N + k] * A[k * N + j~];
        Bij += A[i * N + k] * Bkj_pointer;
        Bkj_pointer += N;
      }
      C[i * N + j] = Bij;
    }
  }
```

transforms loop to iterate with pointer

compiler will usually do this!

increment/decrement by N ($\times$ sizeof(float))

addressing efficiency

compiler will usually eliminate slow multiplies

doing transformation yourself often slower if so

`i * N; ++i` into `i_times_N; i_times_N += N`

way to check: see if assembly uses lots multiplies in loop

if it doesn't — do it yourself

another addressing transformation

```
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
float *Ai0_base = &A[k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

compiler will do this, too

another addressing transformation

```
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
float *Ai0_base = &A[k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 4) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

compiler will do this, too

another addressing transformation

```
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

storing 20 `AiX_base`? — need the stack

another addressing transformation

```
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += A[(i+0) * n + k] * B[k * n + j];
    C[(i+1) * n + j] += A[(i+1) * n + k] * B[k * n + j];
    // ...
```

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
float *Ai2_base = Ai1_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

storing 20 `AiX_base`? — need the stack

alternative addressing transformation

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

```
float *Ai0_base = &A[k];
for (int i = 0; i < n; i += 20) {
    float *A_ptr = &Ai0_base[i*n];
    C[(i+0) * n + j] += *A_ptr * A[k * n + j];
    A_ptr += n;  // what about multiple accumulators???
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

more dependencies (latency bound?), more additions?, less registers

might need multiple accumulator transformation?

alternative addressing transformation

```
float *Ai0_base = &A[0*n+k];
float *Ai1_base = Ai0_base + n;
// ...
for (int i = 0; i < n; i += 20) {
    C[(i+0) * n + j] += Ai0_base[i*n] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[i*n] * B[k * n + j];
    // ...
```

```
float *Ai0_base = &A[k];
for (int i = 0; i < n; i += 20) {
    float *A_ptr = &Ai0_base[i*n];
    C[(i+0) * n + j] += *A_ptr * A[k * n + j];
    A_ptr += n;  // what about multiple accumulators???
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

more dependencies (latency bound?), more additions?, less registers

might need multiple accumulator transformation?

an infinite loop

```
int main(void) {
    while (1) {
        /* waste CPU time */
    }
}
```

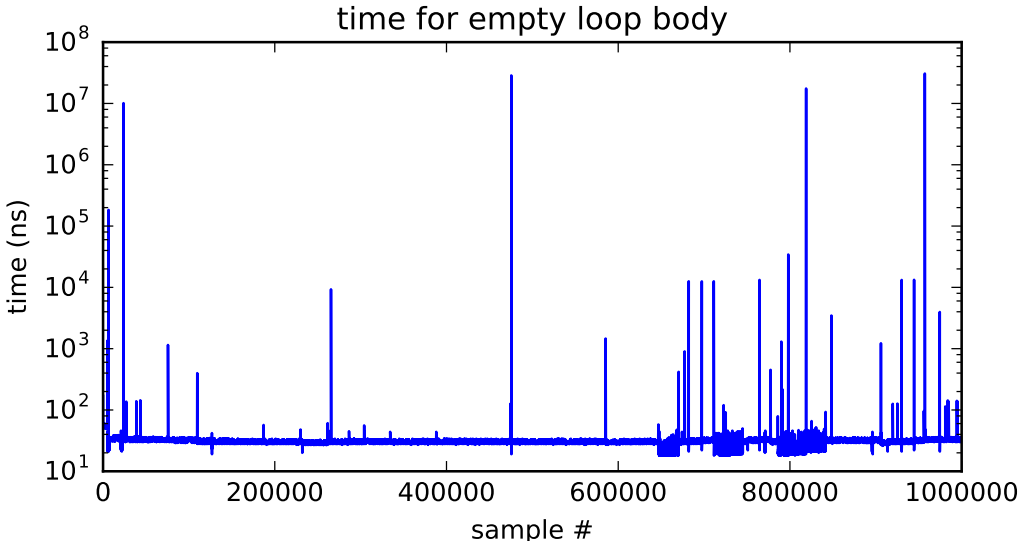
If I run this on a shared department machine, can you still use it?
...if the machine only has one core?

timing nothing

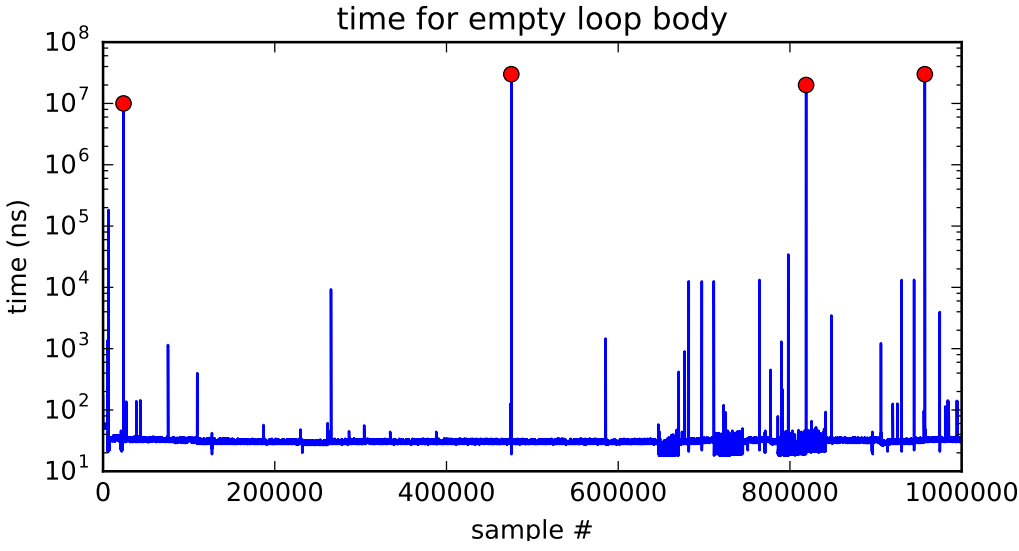
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



time multiplexing

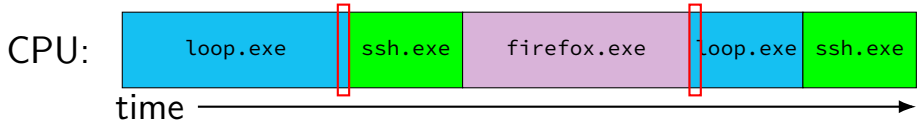
time multiplexing


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing




```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

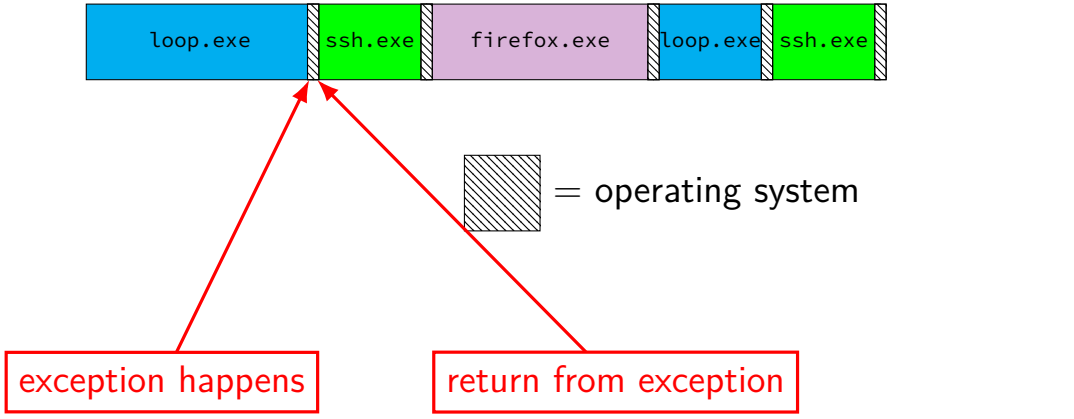
million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing really

 = operating system

time multiplexing really



OS and time multiplexing

starts running instead of normal program

- mechanism for this: exceptions (later)

saves old program counter, registers somewhere

sets new registers, jumps to new program counter

called context switch

- saved information called context

context

all registers values

`%rax` `%rbx`, …, `%rsp`, …

condition codes

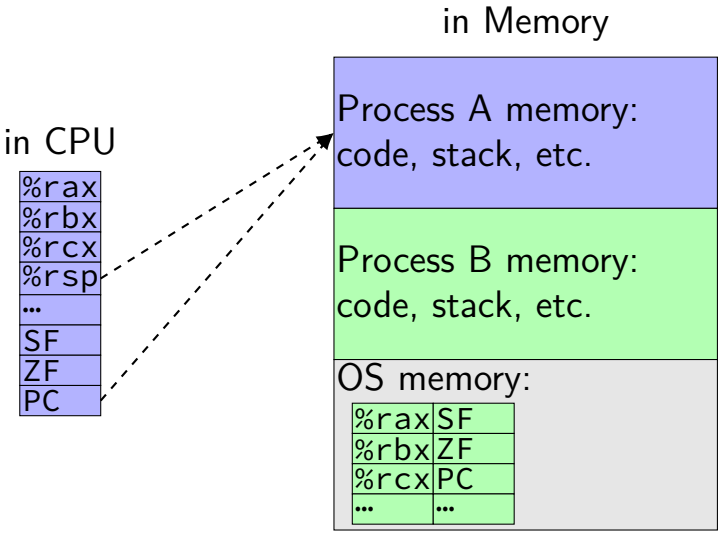
program counter

i.e. all visible state in your CPU except memory

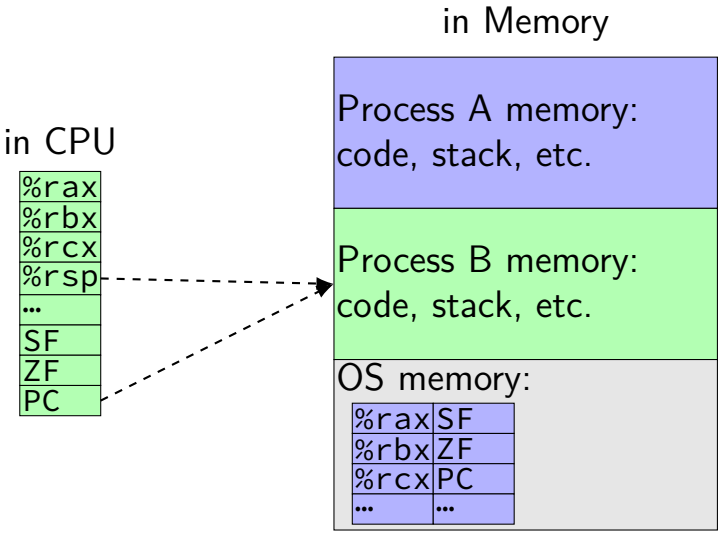
context switch pseudocode

```
context_switch(last, next):
  copy_preexception_pc last->pc
  mov rax,last->rax
  mov rcx, last->rcx
  mov rdx, last->rdx
  ...
  mov next->rdx, rdx
  mov next->rcx, rcx
  mov next->rax, rax
  jmp next->pc
```

contexts (A running)



contexts (B running)



memory protection

reading from another program's memory?

Program A	Program B

`0x10000: .word 42`
`// ...`
`// do work`
`// ...`
`movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

memory protection

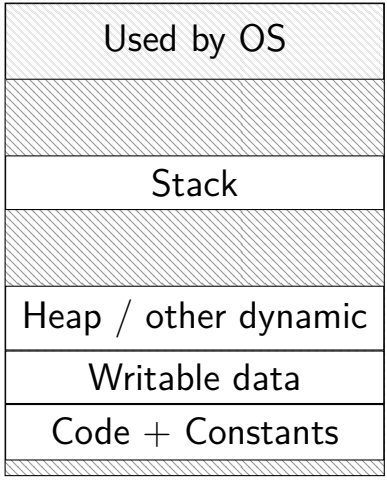
reading from another program's memory?

Program A	Program B

`0x10000: .word 42`
`// ...`
`// do work`
`// ...`
`movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

result: `%rax` is 42 (always)	result: might crash

program memory



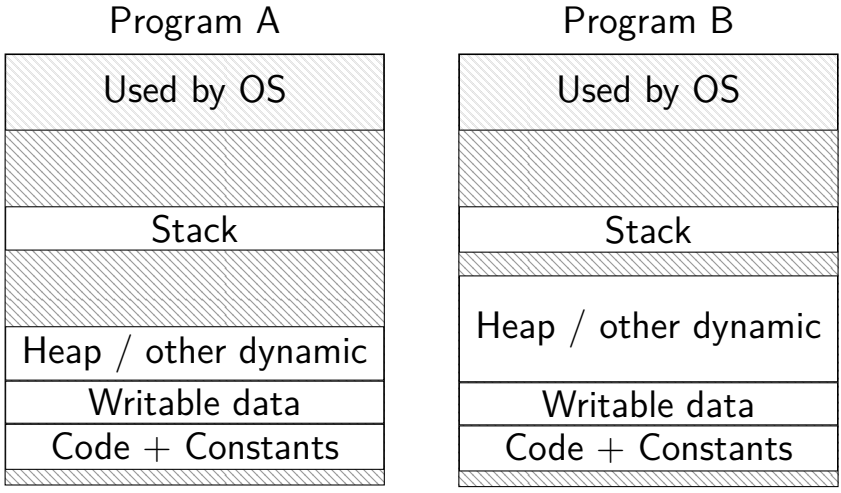
0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

0x0000 0000 0040 0000

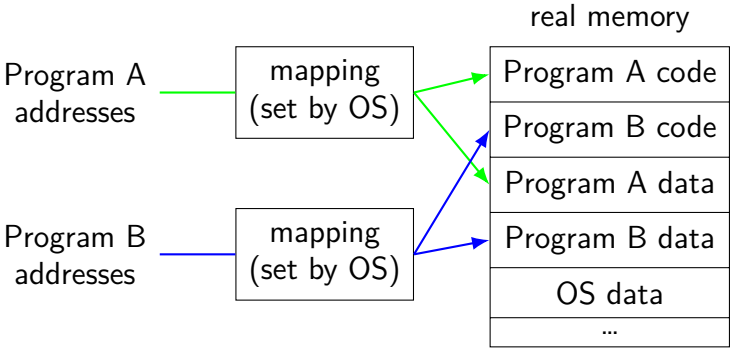
program memory (two programs)



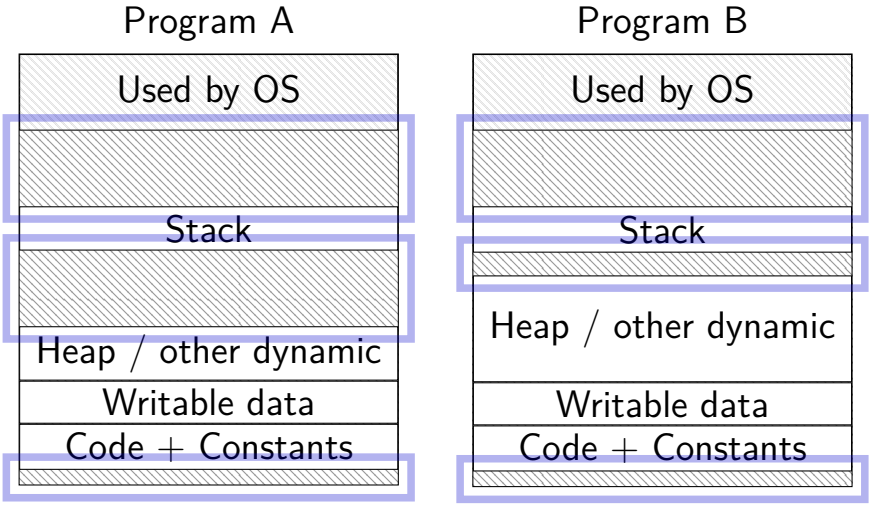
address space

programs have illusion of own memory

called a program's address space



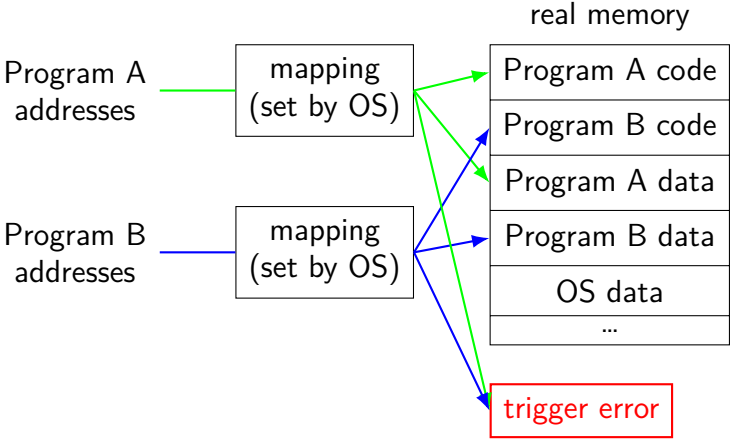
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



address space mechanisms

next topic

called virtual memory

mapping called page tables

mapping part of what is changed in context switch

context

all registers values

`%rax` `%rbx`, …, `%rsp`, …

condition codes

program counter

~~i.e. all visible state in your CPU except memory~~

address space: map from program to real addresses

The Process

process = thread(s) + address space

illusion of dedicated machine:

- thread = illusion of own CPU
- address space = illusion of own memory

synchronous versus asynchronous

synchronous — triggered by a particular instruction

- traps and faults

asynchronous — comes from outside the program

- interrupts and aborts
- timer event
- keypress, other input event

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- divide by zero
- invalid instruction

traps — intentionally triggered exceptions

- system calls — ask OS to do something

aborts

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

faults — errors/events in programs

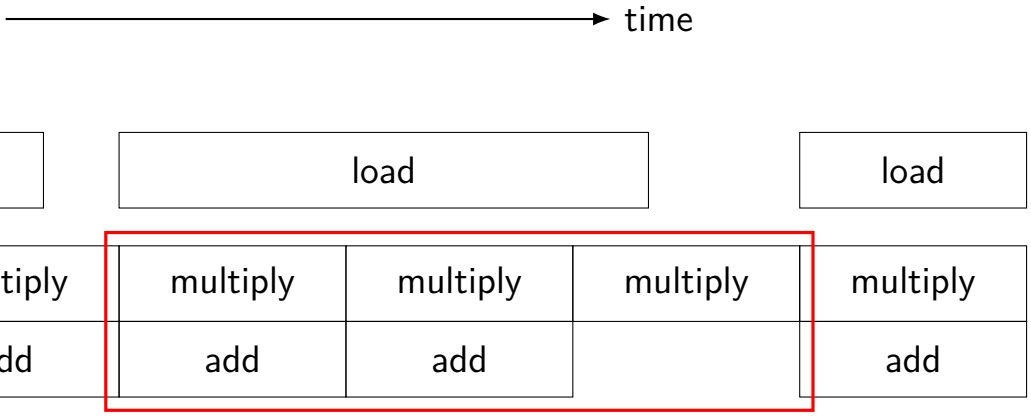
- memory not in address space ("Segmentation fault")
- divide by zero
- invalid instruction

traps — intentionally triggered exceptions

- system calls — ask OS to do something

aborts

overlapping loads and arithmetic



speed of load might not matter if these are slower

optimization and bottlenecks

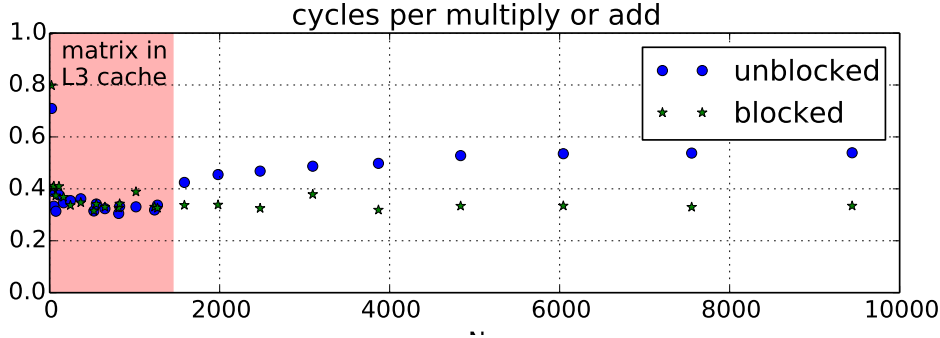
arithmetic/loop efficiency was the bottleneck

after fixing this, cache performance was the bottleneck

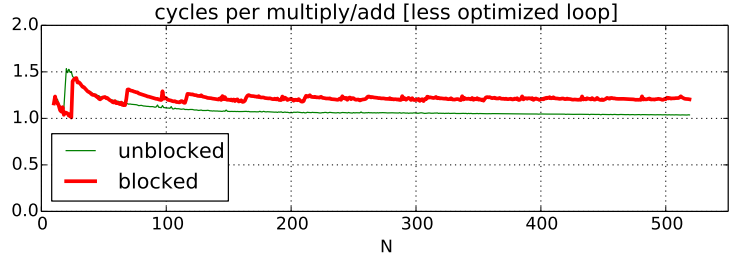
common theme when optimizing:

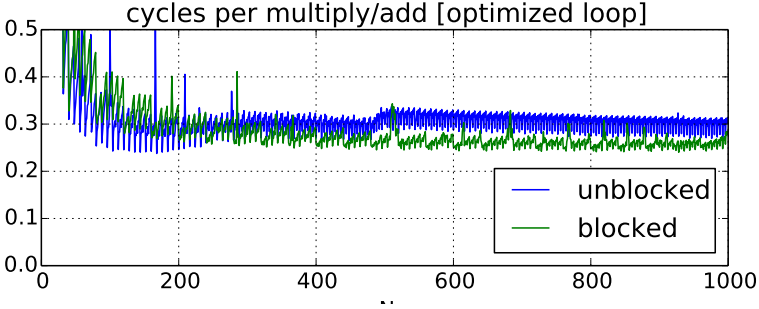
X may not matter until Y is optimized

cache blocking performance (big sizes)



cache blocking performance (small sizes)





constant multiplies/divides (1)

```
unsigned int fiveEights(unsigned int x) {
    return x * 5 / 8;
}
```

```
fiveEights:
        leal    (%rdi,%rdi,4), %eax
        shrl    $3, %eax
        ret
```

constant multiplies/divides (2)

```
int oneHundredth(int x) { return x / 100; }
```

```
oneHundredth:
        movl    %edi, %eax
        movl    $1374389535, %edx
        sarl    $31, %edi
        imull   %edx
        sarl    $5, %edx
        movl    %edx, %eax
        subl    %edi, %eax
        ret
```

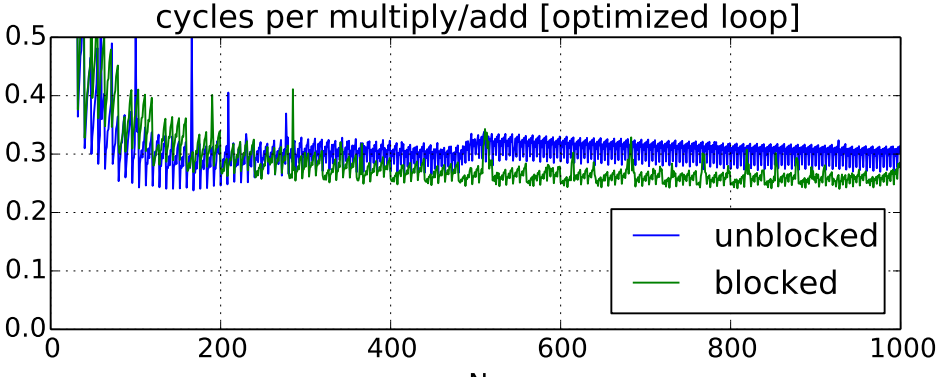
$$\frac{1374389535}{2^{37}} \approx \frac{1}{100}$$

constant multiplies/divides

compiler is very good at handling

...but need to actually use constants

wiggles on prior graphs



variance from this optimization

8 elements in vector, so multiples of 8 easier

aliasing

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
}
```

the compiler **cannot** generate this:

```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax ← *py
        addq    %rax, %rax   // rax ← 2 * *py
        addq    %rax, (%rsi) // *px ← 2 * *py
        ret
```

aliasing problem

```
void twiddle(long *px, long *py) {
    *px += *py;
    *px += *py;
    // NOT the same as *px += 2 * *py;
}
...
    long x = 1;
    twiddle(&x, &x);
    // result should be 4, not 3
```

```
twiddle: // BROKEN // %rsi = px, %rdi = py
        movq    (%rdi), %rax // rax ← *py
        addq    %rax, %rax   // rax ← 2 * *py
        addq    %rax, (%rsi) // *px ← 2 * *py
        ret
```

non-contrived aliasing

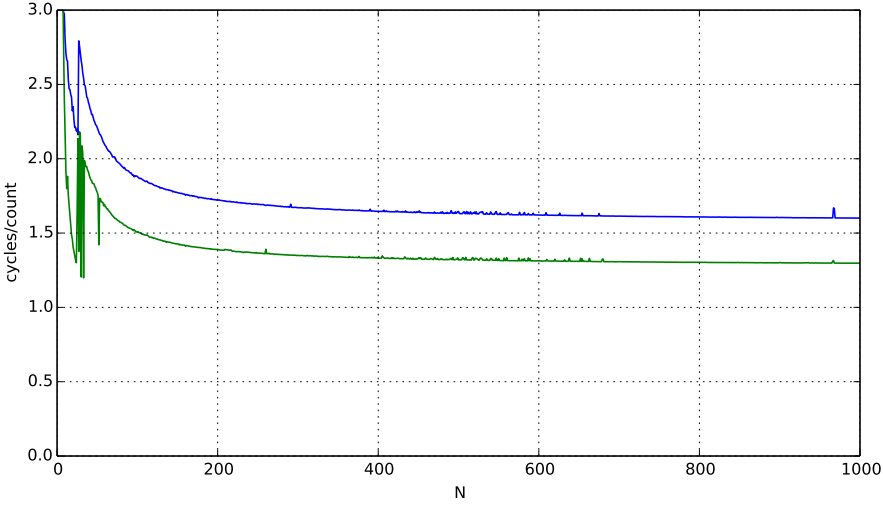
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

non-contrived aliasing

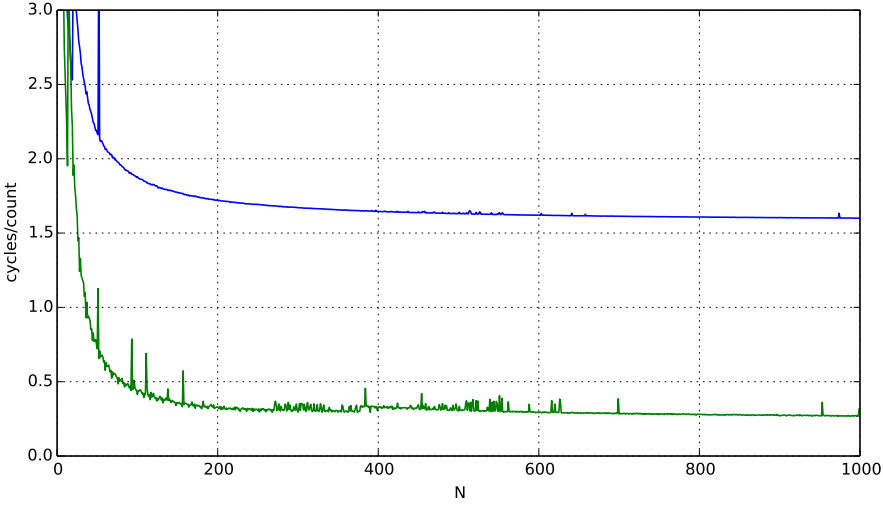
```
void sumRows1(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        result[row] = 0;
        for (int col = 0; col < N; ++col)
            result[row] += matrix[row * N + col];
    }
}
```

```
void sumRows2(int *result, int *matrix, int N) {
    for (int row = 0; row < N; ++row) {
        int sum = 0;
        for (int col = 0; col < N; ++col)
            sum += matrix[row * N + col];
        result[row] = sum;
    }
}
```

aliasing and performance (1) / GCC 5.4 -O2



aliasing and performance (2) / GCC 5.4 -O3



aliasing and cache optimizations

```
for (int k = 0; k < N; ++k)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```

```
for (int i = 0; i < N; ++i)
  for (int j = 0; k < N; ++j)
    for (int k = 0; k < N; ++k)
      B[i*N+j] += A[i * N + k] * A[k * N + j];
```

`B = A`? `B = &A[10]`?

compiler can't generate same code for both