SIMD / vector instructions
Exceptions

April 18 2023

unvectorized add (original)

```
unsigned int A[512], B[512];
...
for (int i = 0; i < N; i += 1) {
    A[i] = A[i] + B[i];
}
```

unvectorized add (unrolled)

```
unsigned int A[512], B[512];
...
for (int i = 0; i < 512; i += 8) {
    A[i+0] = A[i+0] + B[i+0];
    A[i+1] = A[i+1] + B[i+1];
    A[i+2] = A[i+2] + B[i+2];
    A[i+3] = A[i+3] + B[i+3];
    A[i+4] = A[i+4] + B[i+4];
    A[i+5] = A[i+5] + B[i+5];
    A[i+6] = A[i+6] + B[i+6];
    A[i+7] = A[i+7] + B[i+7];
}
```

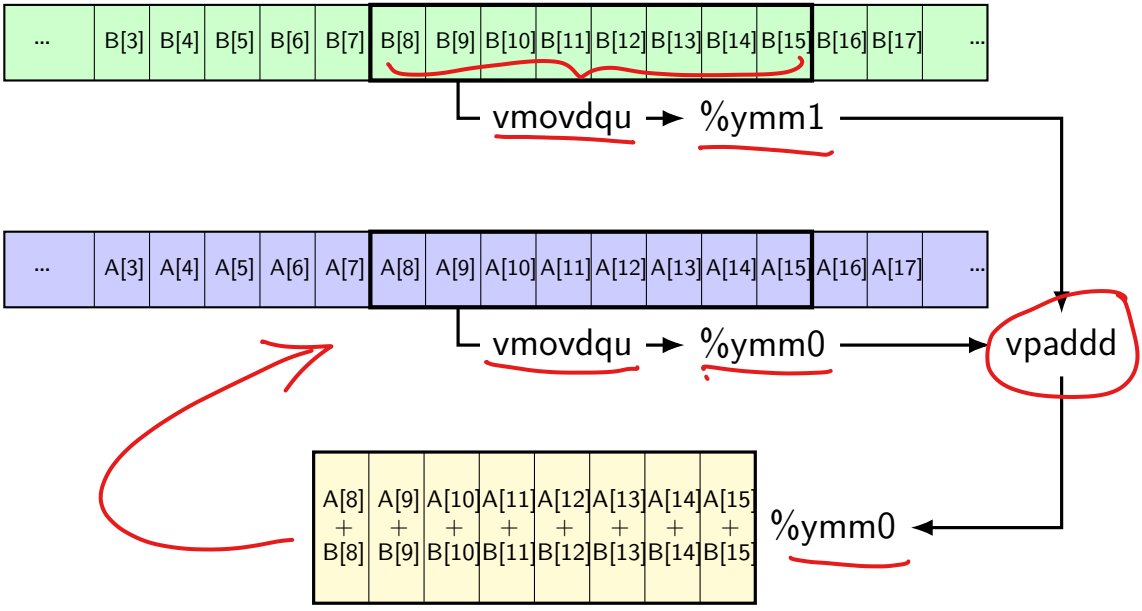
goal: use SIMD add instruction to do all 8 adds above

SIMD = single instruction, multiple data

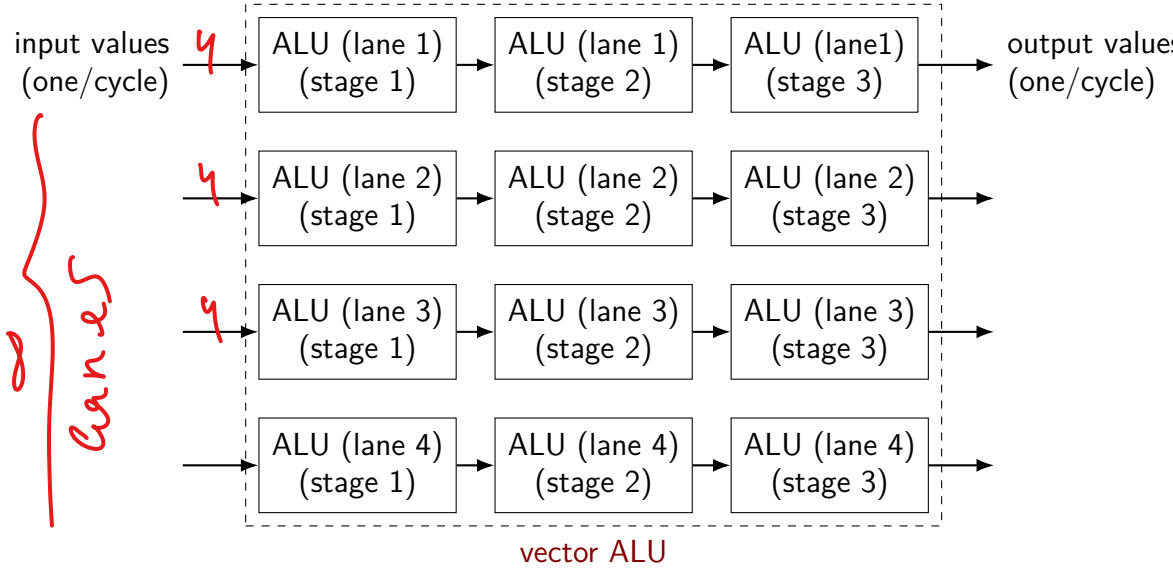
desired assembly

```
   xor %rax, %rax
the_loop:
   vmovdqu A(%rax), %ymm0      /* load 256 bits of A into ymm0 */
   vmovdqu B(%rax), %ymm1      /* load 256 bits of B into ymm1 */
   vpaddd %ymm1, %ymm0, %ymm0  /* ymm1 + ymm0 -> ymm0 */
   vmovdqu %ymm0, A(%rax)      /* store ymm0 into A */
   addq $32, %rax              /* increment index by 32 bytes */
   cmpq $2048, %rax            /* offset < 2048 (= 512 * 4) bytes */
   jne the_loop
```

vector add picture



one view of vector functional units



why vector instructions?

lots of logic not dedicated to computation

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

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, e.g. order of store/load

by conditionals, e.g. add some values only

by some operation with no vector instruction

...

very non-intuitive for me when compiler will/will not use vector instructions

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

```
int A[512], B[512];

for (int i = 0; i < 512; i += 8) {
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = 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)

```
int A[51
for (int i = 0; i < 512; i += 8) {
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = 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)

```
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector intrinsics: add example

```
int A[512], B[512];

for (int i = 0; i < 512; i += 8) {
  // "si256" --
  // a_values =                                    (8 x 32 bits)
  __m256i a_val                                  256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

function to add
epi32 means “8 32-bit integers”

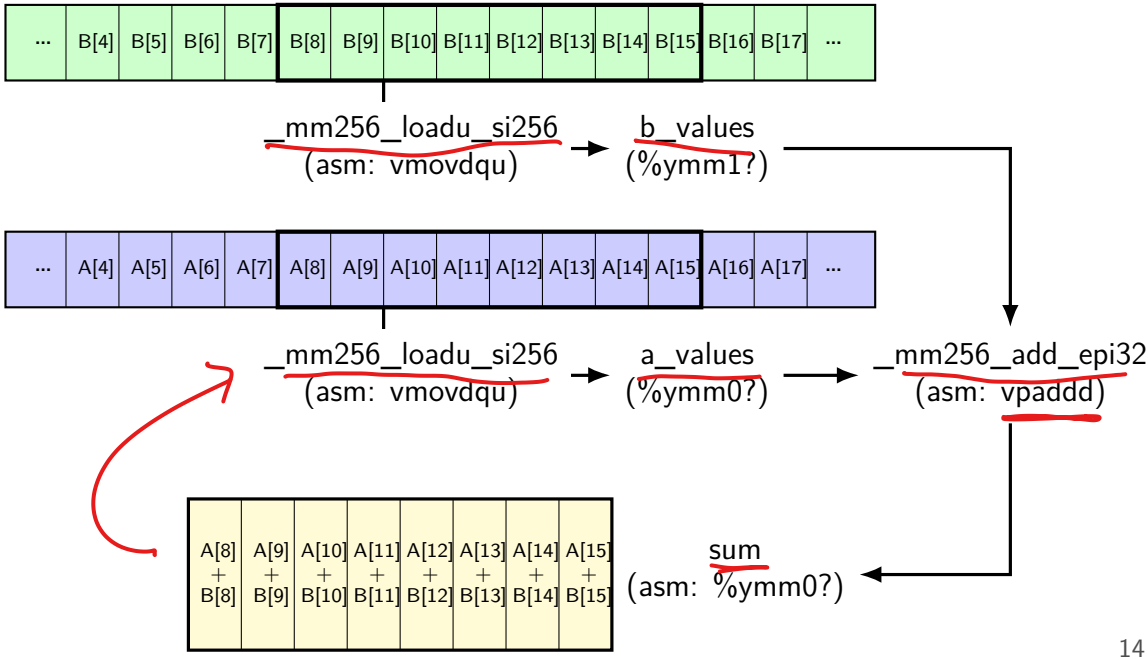
vector intrinsics: different size

```
long A[512], B[512]; /* instead of int */
...
for (int i = 0; i < 512; i += 4) {
  // a_values = {A[i], A[i+1], A[i+2], A[i+3]} (4 x 64 bits)
   __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1], B[i+2], B[i+3]} (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], A[i+2], A[i+3]} = sums
   _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

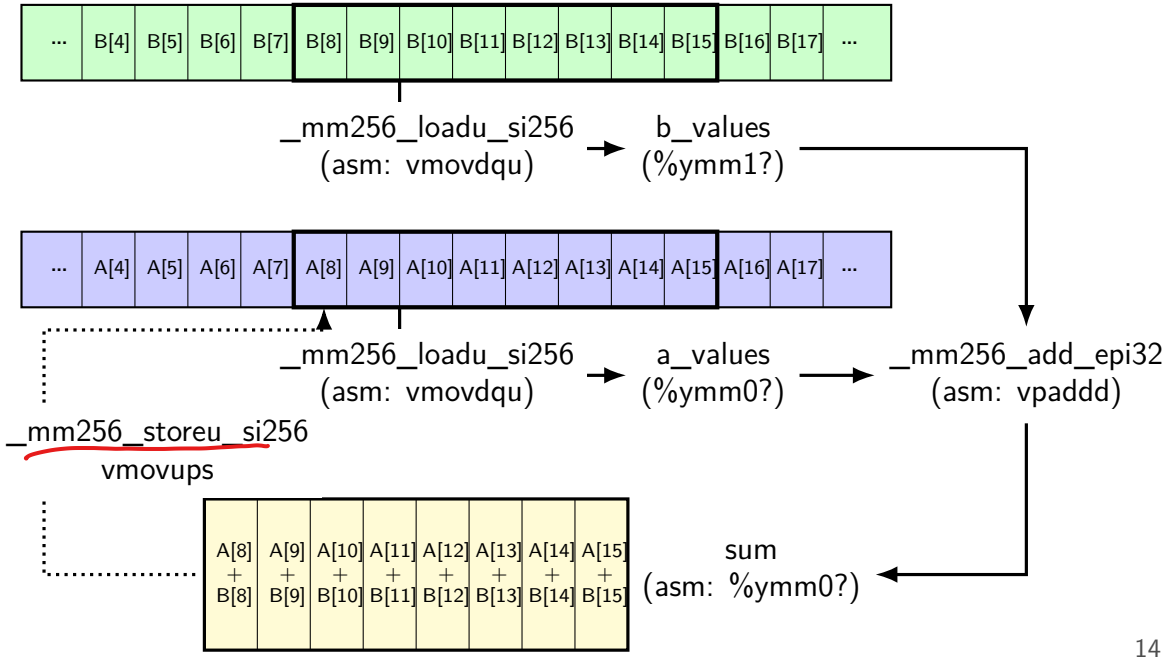
vector intrinsics: different size

```
long A[512], B[512]; /* instead of int */
...
for (int i = 0; i < 512; i += 4) {
  // a_values = {A[i], A[i+1], A[i+2], A[i+3]} (4 x 64 bits)
   __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1], B[i+2], B[i+3]} (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], A[i+2], A[i+3]} = sums
   _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector add picture (intrinsics)



vector add picture (intrinsics)



exercise

```
long foo[8] = {1,1,2,2,3,3,4,4};
long bar[8] = {2,2,2,3,3,3,4,4};
__mm256i foo0_as_vector = _mm256_loadu_si256((__m256i*)&foo[0])
__mm256i foo4_as_vector = _mm256_loadu_si256((__m256i*)&foo[4])
__mm256i bar0_as_vector = _mm256_loadu_si256((__m256i*)&bar[0])

__mm256i result = _mm256_add_epi64(foo0_as_vector, foo4_as_vector);
result = _mm256_mullo_epi64(result, bar0_as_vector);
_mm256_storeu_si256((__mm256i*) &bar[4], result);
```

Final value of bar array?

A. {2,2,2,3,12,12,24,24}
B. {2,2,2,3,15,15,28,28}
C. {2,2,2,3,8,8,12,18}
D. {12,12,24,24,3,3,4,4}
E. {14,14,26,27,3,3,4,4}
F. {14,14,26,27,12,12,24,24}
G. something else

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 256-bit value

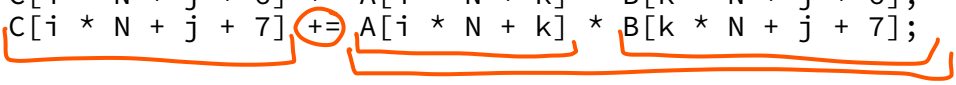
- instructions generated vary; one example: `vmovd` + `vpbroadcastd`

`_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_si256((__m256i*) &B[k * N + j + 0]);
// load eight 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, multiply_results;

// 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}}
multiply_results = _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, multiply_results);

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

vector exercise (2)

```
long A[1024], B[1024];
...
for (int i = 0; i < 1024; i += 1)
    for (int j = 0; j < 1024; j += 1)
        A[i] += B[i] * B[j];
```

(casts omitted below to reduce clutter:)

```
for (int i = 0; i < 1024; i += 4) {
    A_part = _mm256_loadu_si256(&A[i]);
    Bi_part = _mm256_loadu_si256(&B[i]);
    for (int j = 0; j < 1024; /* BLANK 1 */) {
        Bj_part = _mm256_/* BLANK 2 */;
        A_part = _mm256_add_epi64(A_part,
            _mm256_mullo_epi64(Bi_part, Bj_part));
    }
    _mm256_storeu_si256(&A[i], A_part);
}
```

What goes in BLANK 1 and BLANK 2?

A. `j += 1`, `loadu_si256(&B[j])`
B. `j += 4`, `loadu_si256(&B[j])`
C. `j += 1`, `set1_epi64(B[j])`
D. `j += 4`, `set1_epi64(B[j])`

vector exercise 2 explanation

```
for (int i = 0; i < 1024; i += 1)
    for (int j = 0; j < 1024; j += 1)
        A[i] += B[i] * B[j];
/* -- transformed into -- */
for (int i = 0; i < 1024; i += 4)
    for (int j = 0; j < 1024; j += 1) {
        A[i+0] += B[i+0] * B[j];
        A[i+1] += B[i+1] * B[j];
        A[i+2] += B[i+2] * B[j];
        A[i+3] += B[i+3] * B[j];
    }

/* not the much harder to vectorize: */
for (int i = 0; i < 1024; i += 1)
    for (int j = 0; j < 1024; j += 4) {
        A[i] += B[i] * B[j+0];
        A[i] += B[i] * B[j+1];
        A[i] += B[i] * B[j+2];
        A[i] += B[i] * B[j+3];
    }
```

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} */
```

other vector instructions

multiple extensions to the X86 instruction set for vector instructions

early versions (128-bit vectors): SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2

> 128-bit vectors

this class (256-bit): AVX, AVX2

not this class (512+-bit): AVX-512

> 512-bit vectors

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

GPUs are essentially vector-instruction-specialized CPUs

optimizing real programs

ask your compiler to try first

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?

Amdahl's law:

> the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used

profilers

first step — tool to determine where you spend time

tools exist to do this for programs

example on Linux: `perf`

example

```
Samples: 37K of event 'cycles', Event count (approx.): 37367555513
  Children      Self  Command          Shared Object          Symbol
+  100.00%     0.00%  hclrs-with-debu  hclrs-with-debuginfo   [.] _start
+  100.00%     0.00%  hclrs-with-debu  libc-2.31.so           [.] __libc_start_main
+  100.00%     0.00%  hclrs-with-debu  hclrs-with-debuginfo   [.] main
+  100.00%     0.00%  hclrs-with-debu  hclrs-with-debuginfo   [.] std::sys_common::backtrace::__rust_begin_short_backt
+  100.00%     0.00%  hclrs-with-debu  hclrs-with-debuginfo   [.] hclrs::main
+   99.99%     9.75%  hclrs-with-debu  hclrs-with-debuginfo   [.] hclrs::program::RunningProgram::run
+   60.37%    31.67%  hclrs-with-debu  hclrs-with-debuginfo   [.] hclrs::ast::SpannedExpr::evaluate
+   41.34%    23.29%  hclrs-with-debu  hclrs-with-debuginfo   [.] hashbrown::map::make_hash
+   18.08%    18.07%  hclrs-with-debu  hclrs-with-debuginfo   [.] <std::collections::hash::map::DefaultHasher as core:
+   16.33%     0.68%  hclrs-with-debu  hclrs-with-debuginfo   [.] hclrs::program::Program::process_register_banks
+    9.54%     3.15%  hclrs-with-debu  hclrs-with-debuginfo   [.] std::collections::hash::map::HashMap<K,V,S>::get
+    9.10%     9.09%  hclrs-with-debu  libc-2.31.so           [.] __memcmp_avx2_movbe
+    6.11%     2.10%  hclrs-with-debu  hclrs-with-debuginfo   [.] hashbrown::map::HashMap<K,V,S>::get_mut
+    2.32%     0.88%  hclrs-with-debu  hclrs-with-debuginfo   [.] std::collections::hash::map::HashMap<K,V,S>::get
+    1.45%     0.52%  hclrs-with-debu  hclrs-with-debuginfo   [.] hashbrown::map::HashMap<K,V,S>::insert
     0.37%     0.11%  hclrs-with-debu  hclrs-with-debuginfo   [.] <alloc::string::String as core::clone::Clone>::clone
     0.19%     0.19%  hclrs-with-debu  libc-2.31.so           [.] malloc
```

an infinite loop

main: jp main

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

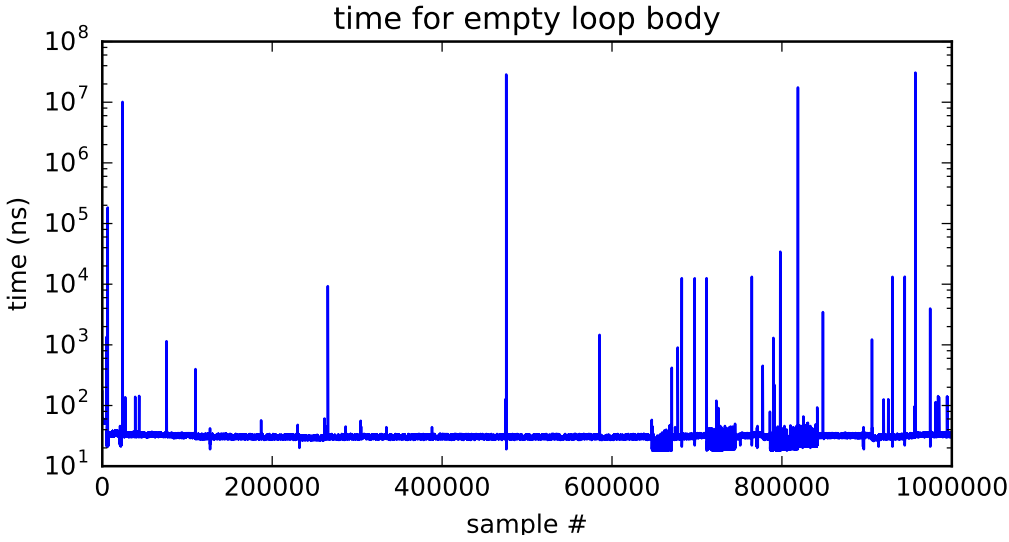
If I run this on a shared department machine, can you still use it?
...if the machine only has one core?

timing nothing

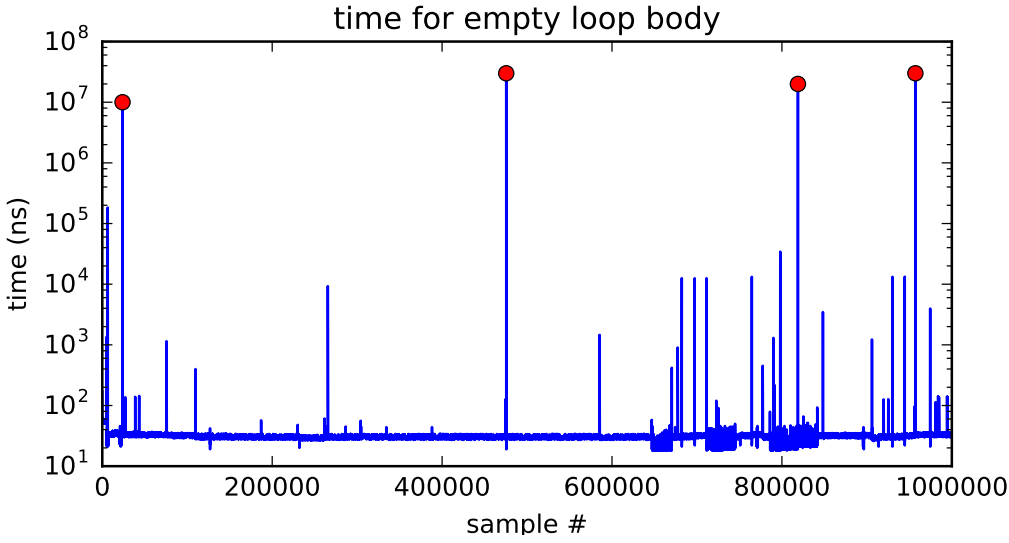
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system

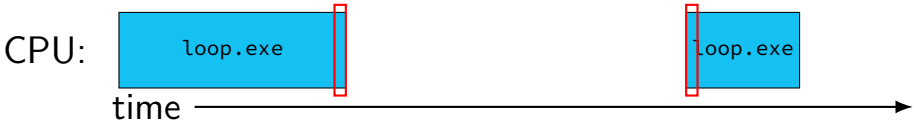


doing nothing on a busy system



time multiplexing

time multiplexing

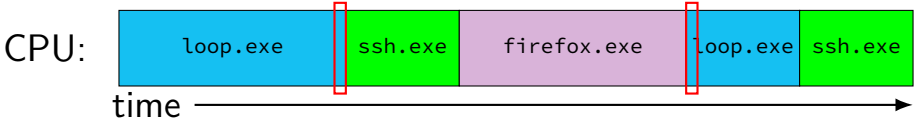



```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

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
```

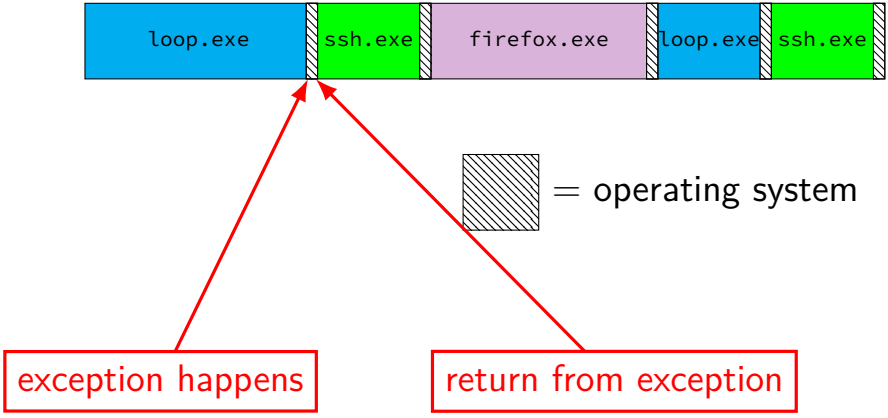
million cycle delay

```
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

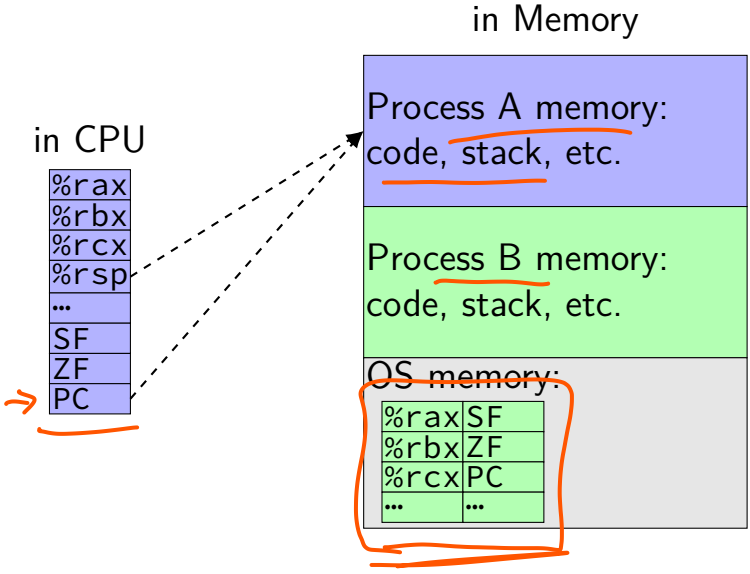
program counter

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

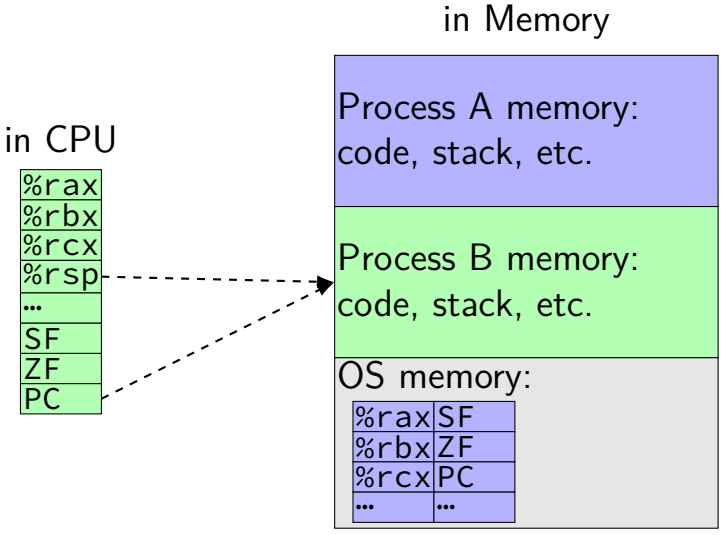
context switch pseudocode

```
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 ...

A. 42 B. 99 C. 0x10000

D. 42 or 99 (depending on timing/program layout/etc)

E. 42 or program might crash (depending on ...)

F. 99 or program might crash (depending on ...)

G. 42 or 99 or program might crash (depending on ...)

H. something else

memory protection

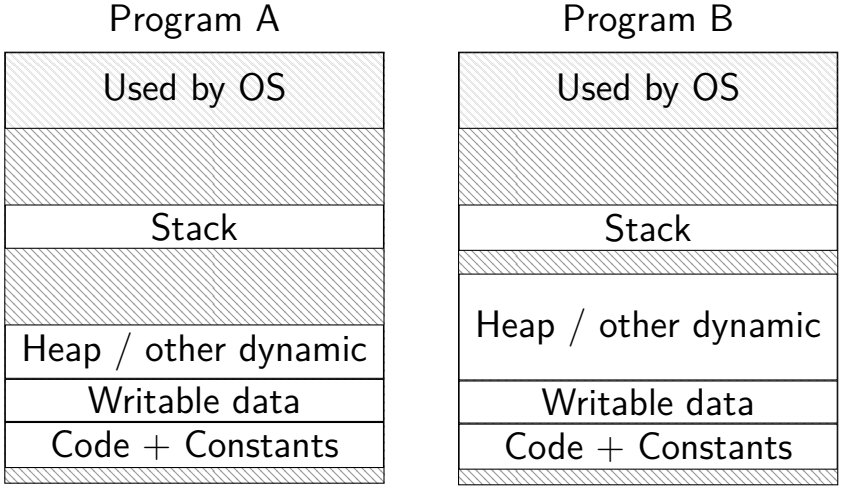
reading from another program's memory?

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

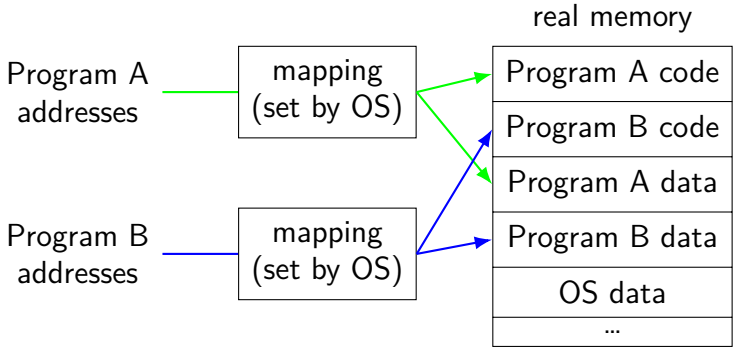
program memory (two programs)



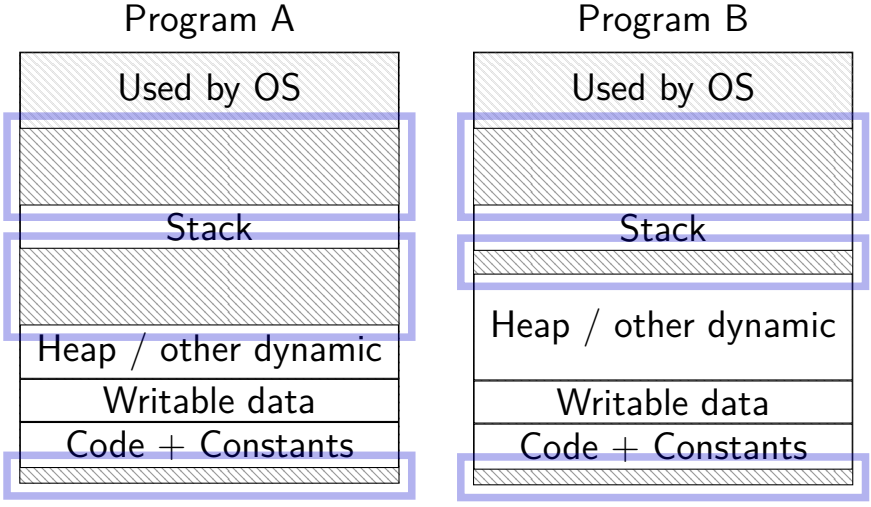
address space

programs have illusion of own memory

called a program's address space



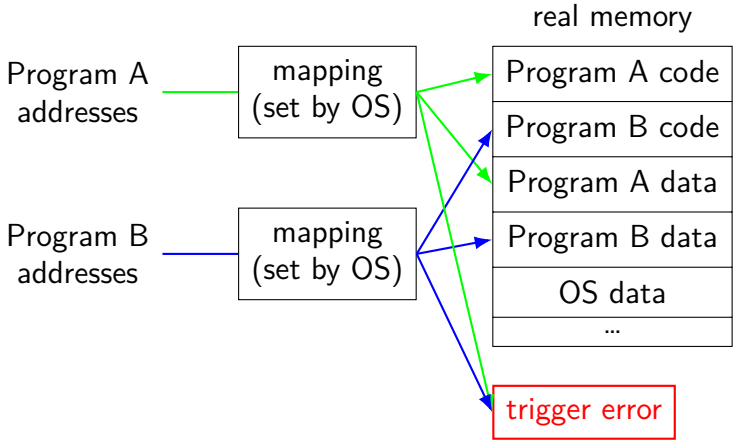
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



address space mechanisms

topic after exceptions

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

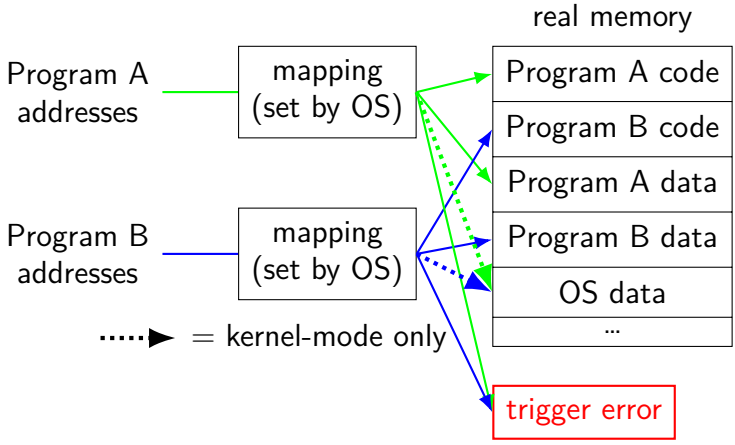
~~i.e. all visible state in your CPU except memory~~

address space: map from program to real addresses

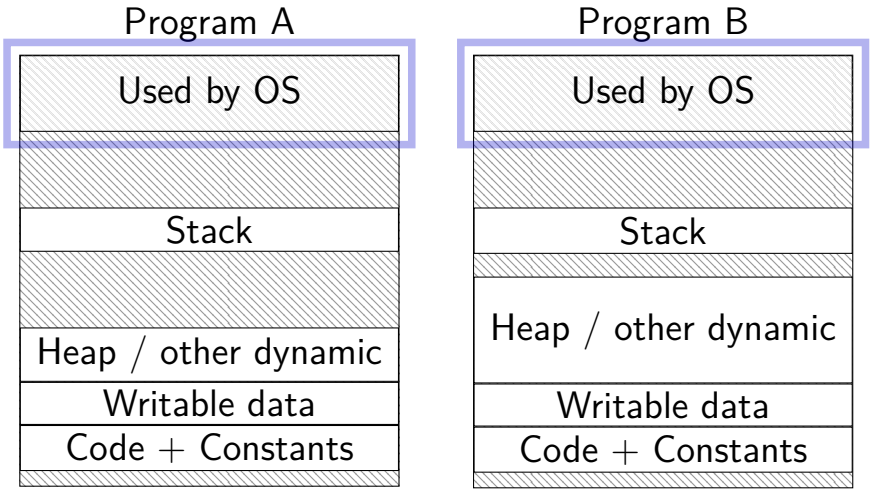
address space

programs have illusion of own memory

called a program's address space



program memory (two programs)



backup slides

exercise: miss estimating (3)

```
for (int kk = 0; kk < 1000; kk += 10)
    for (int jj = 0; jj < 1000; jj += 10)
        for (int i = 0; i < 1000; i += 1)
            for (int j = jj; j < jj+10; j += 1)
                for (int k = kk; k < kk + 10; k += 1)
                    A[k*N+j] += B[i*N+j];
```

assuming: 4 elements per block

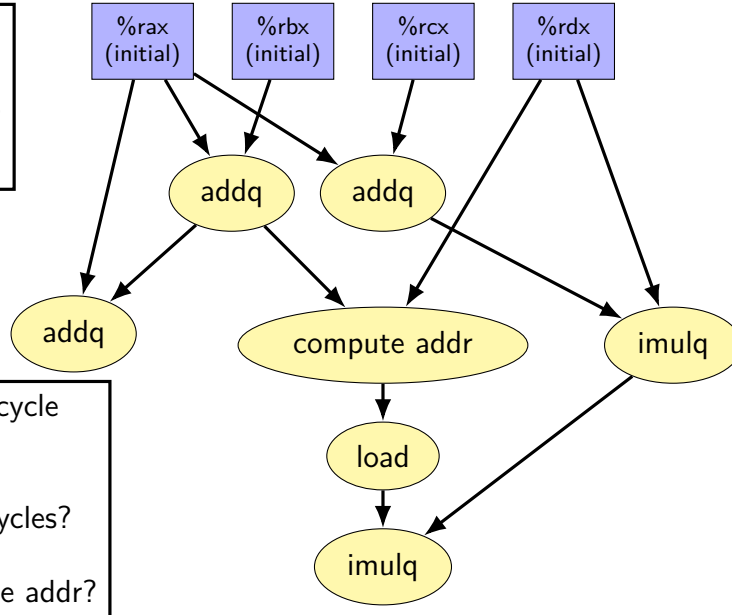
assuming: cache not close to big enough to hold 1K elements, but big enough to hold 500 or so

estimate: *approximately* how many misses for A, B?

hint 1: part of A, B loaded in two inner-most loops only needs to be loaded once

a data flow example

```
addq %rax, %rbx
addq %rax, %rcx
imulq %rdx, %rcx
movq (%rbx, %rdx), %r8
imulq %r8, %rcx
addq %rax, %rbx
```



addq, compute addr: 1 cycle
imulq: 3 cycle latency
load: 3 cycle latency
Q1: latency bound on cycles?
Q2: what can be done
at same time as compute addr?

recall: shifts

we mentioned that compilers compile $x/4$ into a shift instruction

they are really good at these types of of transformation...

> "strength reduction": replacing complicated op with simpler one

but can't do without seeing special case (e.g. divide by constant)

Intel Skylake OOO design

2015 Intel design — codename 'Skylake'

94-entry instruction queue-equivalent

168 physical integer registers

168 physical floating point registers

4 ALU functional units

- but some can handle more/different types of operations than others

2 load functional units

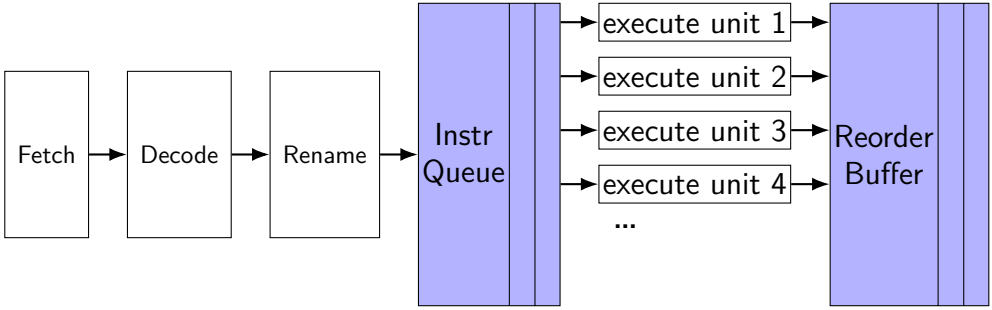
- but pipelined: supports multiple pending cache misses in parallel

1 store functional unit

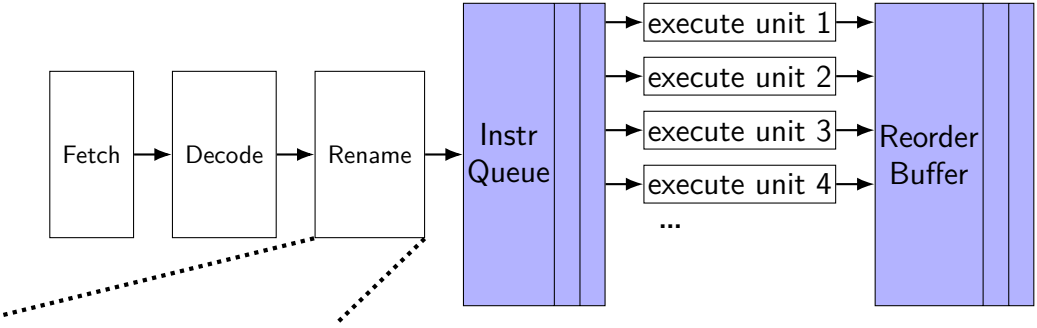
224-entry reorder buffer

- determines how far ahead branch mispredictions, etc. can happen

exceptions and OOO (one strategy)



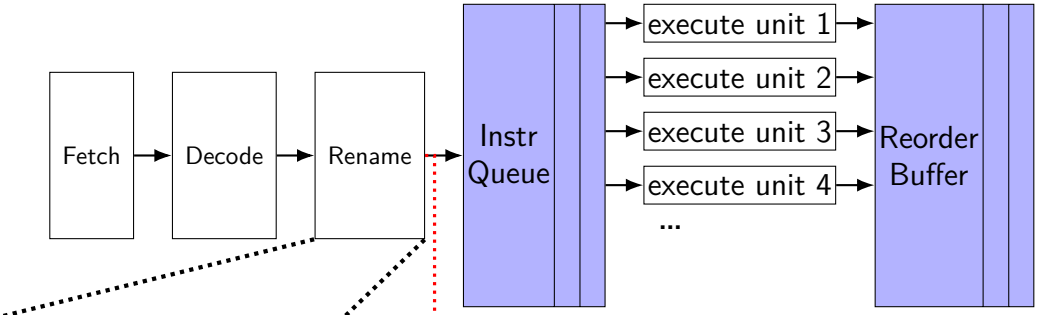
exceptions and OOO (one strategy)



X19
X23
...

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

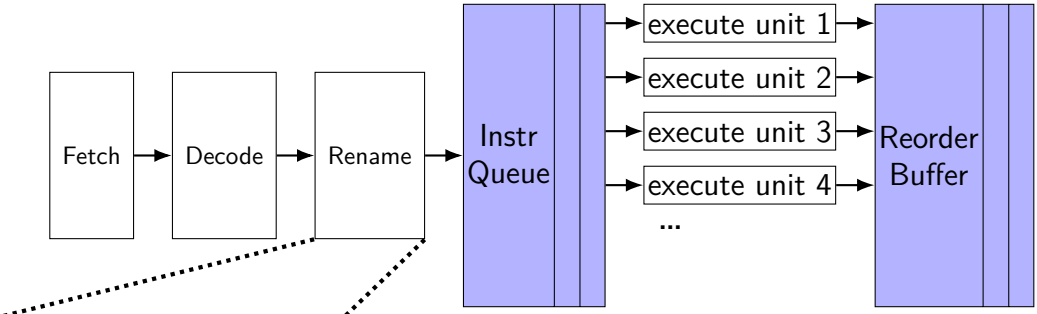
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

done instrs committed in order

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
17	0x1244	RCX / X32		
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
...	...	...	...	...

new instrs added

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

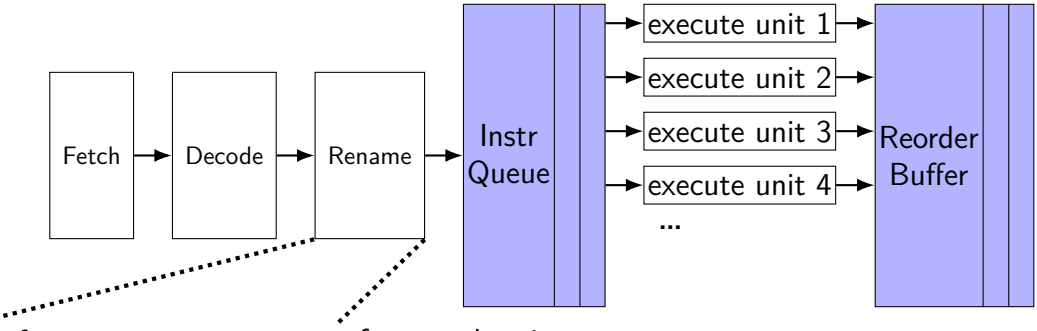
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
17	0x1244	RCX / X32	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

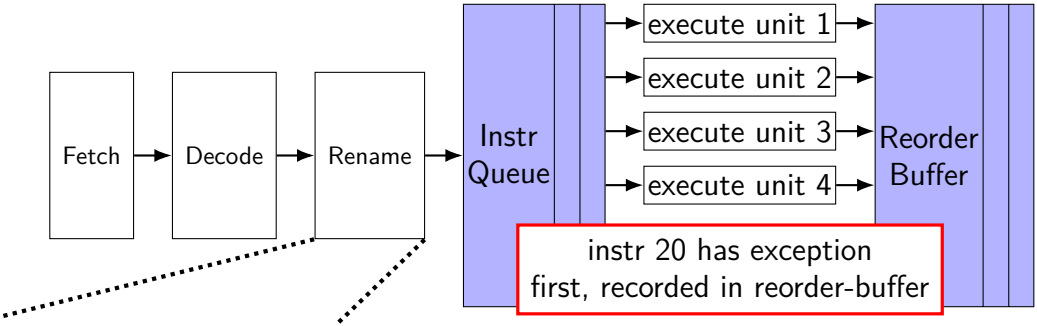
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05		
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

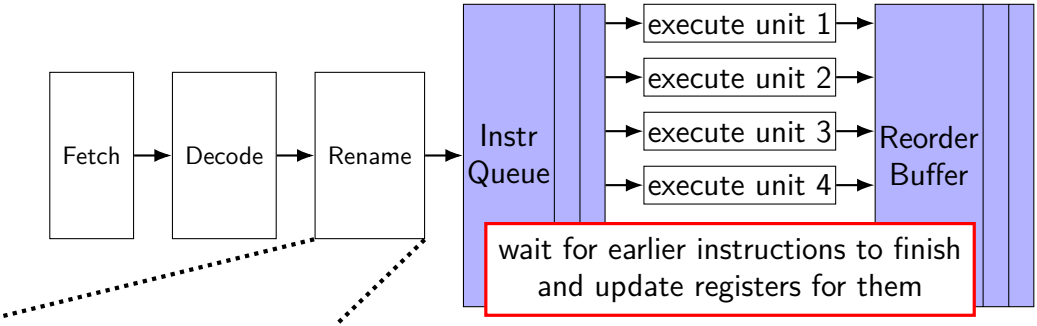
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	X21
RCX	~~X2~~ X32
RBX	X48
RDX	X37
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
18	0x1248	RDX / X34		
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



free regs

X19
X23
...

for new instrs

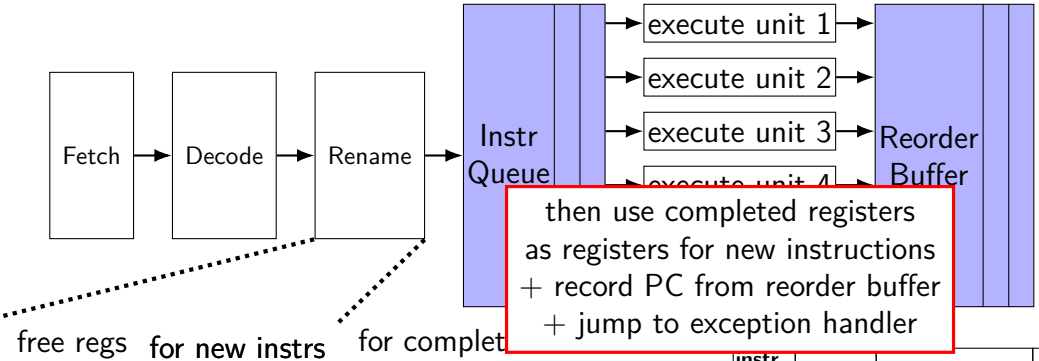
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
~~18~~	~~0x1248~~	~~RDX / X34~~	✓	
~~19~~	~~0x1249~~	~~RAX / X38~~	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



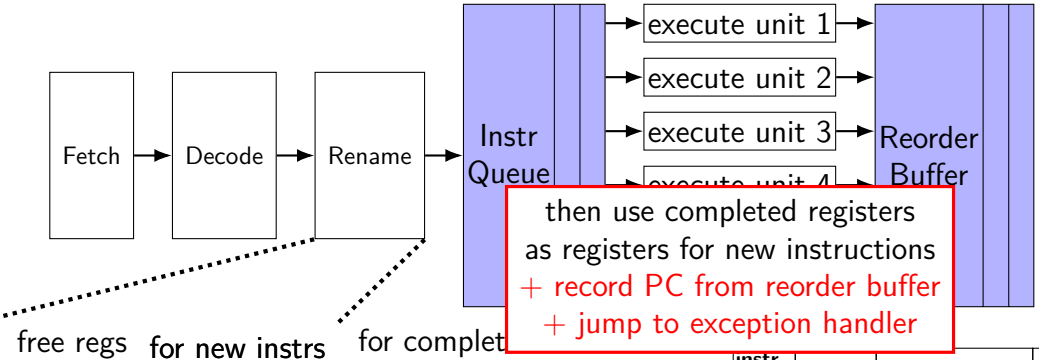
X19
X23
...

arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
...	...

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	✓	
~~18~~	~~0x1248~~	~~RDX / X34~~	✓	
~~19~~	~~0x1249~~	~~RAX / X38~~	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



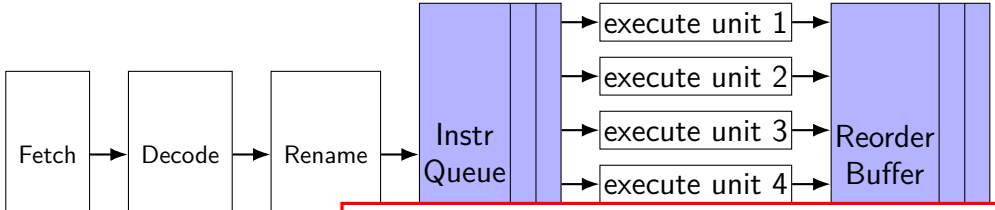
X19
X23
...

arch. reg	phys. reg
RAX	X38
RCX	X32
RBX	X48
RBX	X34
...	...

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
~~18~~	~~0x1248~~	~~RDX / X34~~	~~✓~~	
~~19~~	~~0x1249~~	~~RAX / X38~~	~~✓~~	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



variation: could store architectual reg. values instead of mapping for completed instrs.
(and copy values instead of mapping on exception)

free regs

X19
X23
...

for new instrs

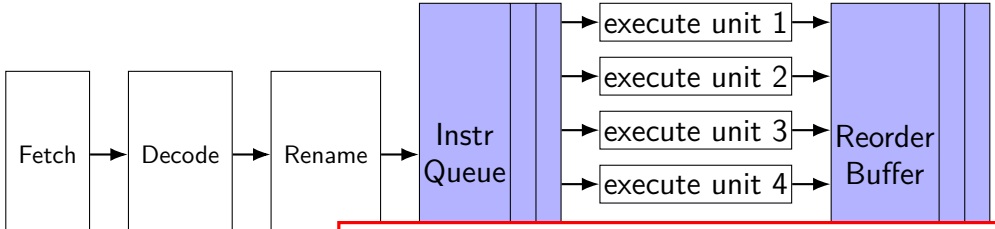
arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	value
RAX	0x12343
RCX	0x234543
RBX	0x56782
RDX	0xF83A4
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

exceptions and OOO (one strategy)



stopping instructions in progress for exception
similar to how 'squashing' mispredicted instructions

free regs

X19
X23
...

for new instrs

arch. reg	phys. reg
RAX	X15
RCX	X17
RBX	X13
RBX	X07
...	...

for complete instrs

arch. reg	phys. reg
RAX	~~X21~~ X38
RCX	~~X2~~ X32
RBX	X48
RDX	~~X37~~ X34
...	...

instr num.	PC	dest. reg	done?	except?
...	...	...	...	...
~~17~~	~~0x1244~~	~~RCX / X32~~	~~✓~~	
18	0x1248	RDX / X34	✓	
19	0x1249	RAX / X38	✓	
20	0x1254	R8 / X05	✓	✓
21	0x1260	R8 / X06		
...	...	...	...	...

addressing efficiency

```
for (int kk = 0; kk < N; kk += 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 often 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 often 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];
    // ...
```

```
int offset = 0;
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[offset] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[offset] * B[k * n + j];
    // ...
    offset += n;
```

compiler will sometimes 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];
    // ...
```

```
int offset = 0;
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[offset] * B[k * n + j];
    C[(i+1) * n + j] += Ai1_base[offset] * B[k * n + j];
    // ...
    offset += n;
```

compiler will sometimes 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];
    // ...
```

```
int offset = 0;
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];
    // ...
    offset += n;
```

storing 20 `AiX_base`? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

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];
    // ...
```

```
int offset = 0;
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];
    // ...
    offset += n;
```

storing 20 `AiX_base`? — need the stack

maybe faster (quicker address computation)

maybe slower (can't do enough loads)

alternative addressing transformation

instead of:

```
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];
    // ...
```

---

could do:

```
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;
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

avoids spilling on the stack, but more dependencies

alternative addressing transformation

instead of:

```
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];
    // ...
```

---

could do:

```
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;
    C[(i+1) * n + j] += *A_ptr * B[k * n + j];
    // ...
```

avoids spilling on the stack, but more dependencies

addressing efficiency generally

mostly: compiler does very good job itself

- eliminates multiplications, use pointer arithmetic
- often will do better job than if how typically programming would do it manually

sometimes compiler won't take the best option

- if spilling to the stack: can cause weird performance anomalies
- if indexing gets too complicated — might not remove multiply

if compiler doesn't, you can always make addressing simple yourself

- convert to pointer arith. without multiplies