Lab 4: Parallelism

Hung-Wei Tseng

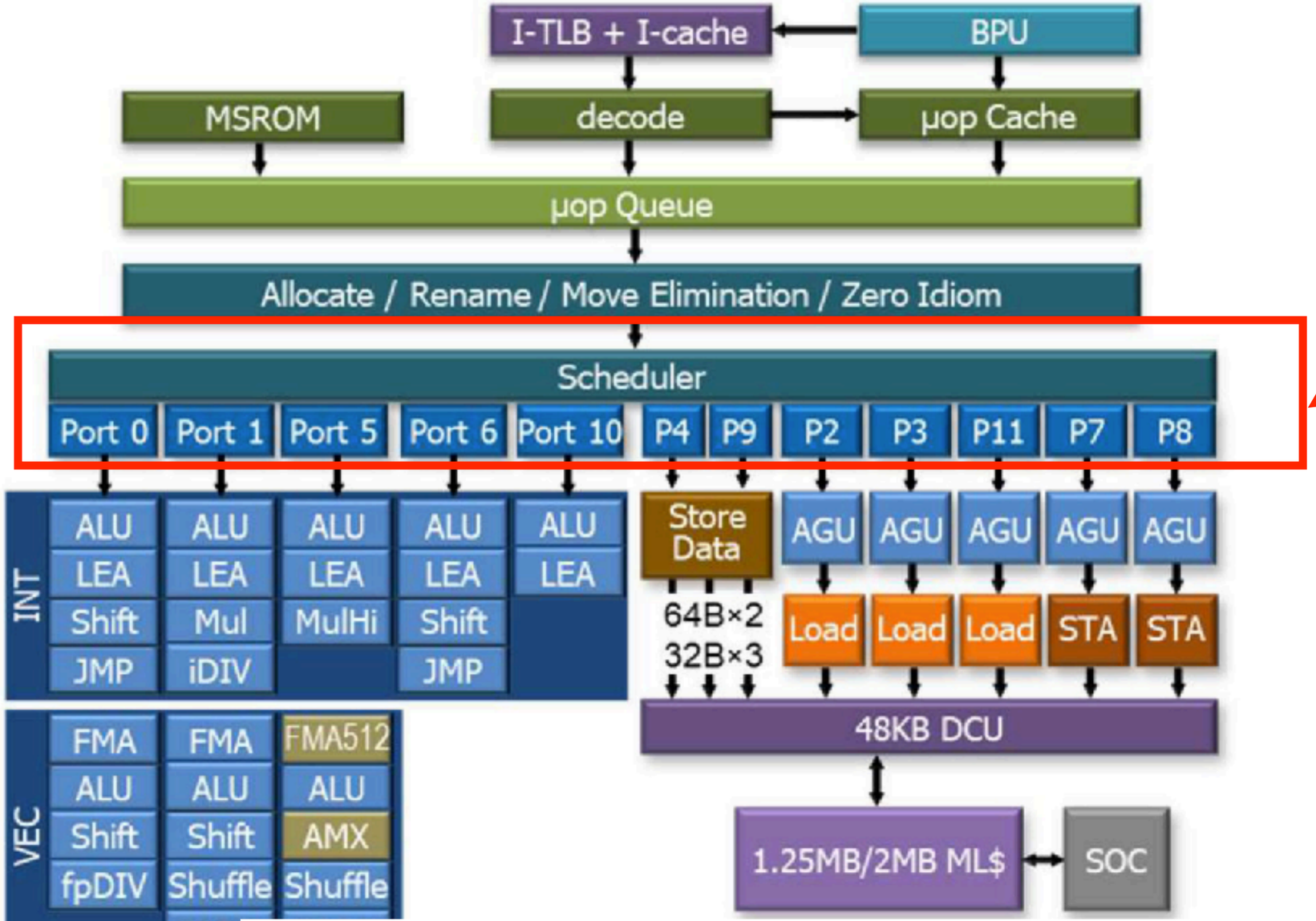
Parallelism in modern computers

- Instruction-level parallelism (ILP)
  - The ability to execute multiple instructions concurrently
- Thread-level parallelism (TLP)
  - The ability to execute multiple program instances concurrently
- Data-level parallelism (DLP)
  - The ability to process data concurrently

Parallelism in modern computers

- SISD — single instruction stream, single data
- SIMD — single instruction stream, multiple data (DLP)
- MIMD — multiple instruction stream (e.g. multiple threads, multiple processes), multiple data (TLP)

Intel Alder Lake (P-Core)

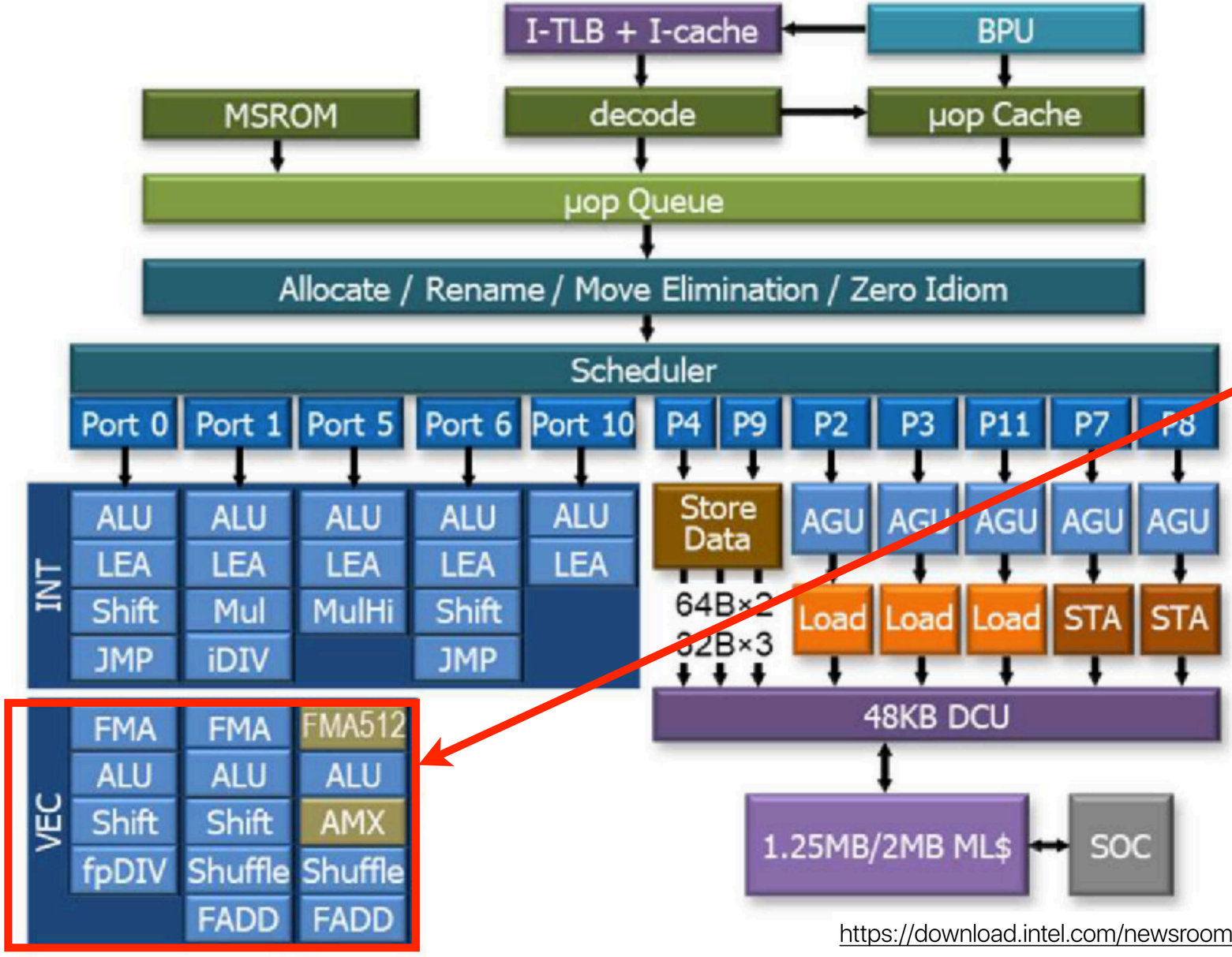


Instruction-Level Parallelism (ILP)

https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

CMP/SMT can help with TLP

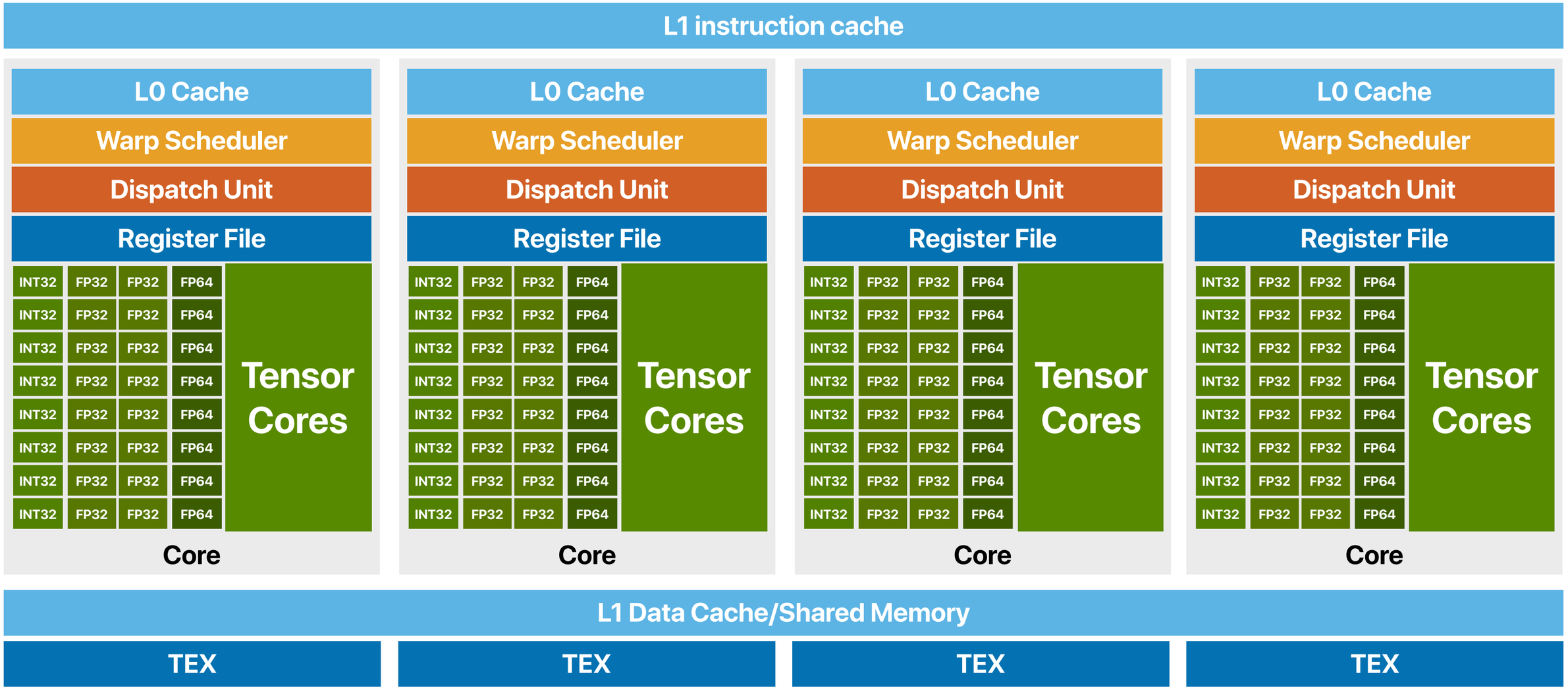
Intel Alder Lake (P-Core)



Data-Level Parallelism (DLP)

https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

DLP: NVIDIA GPU Architectures

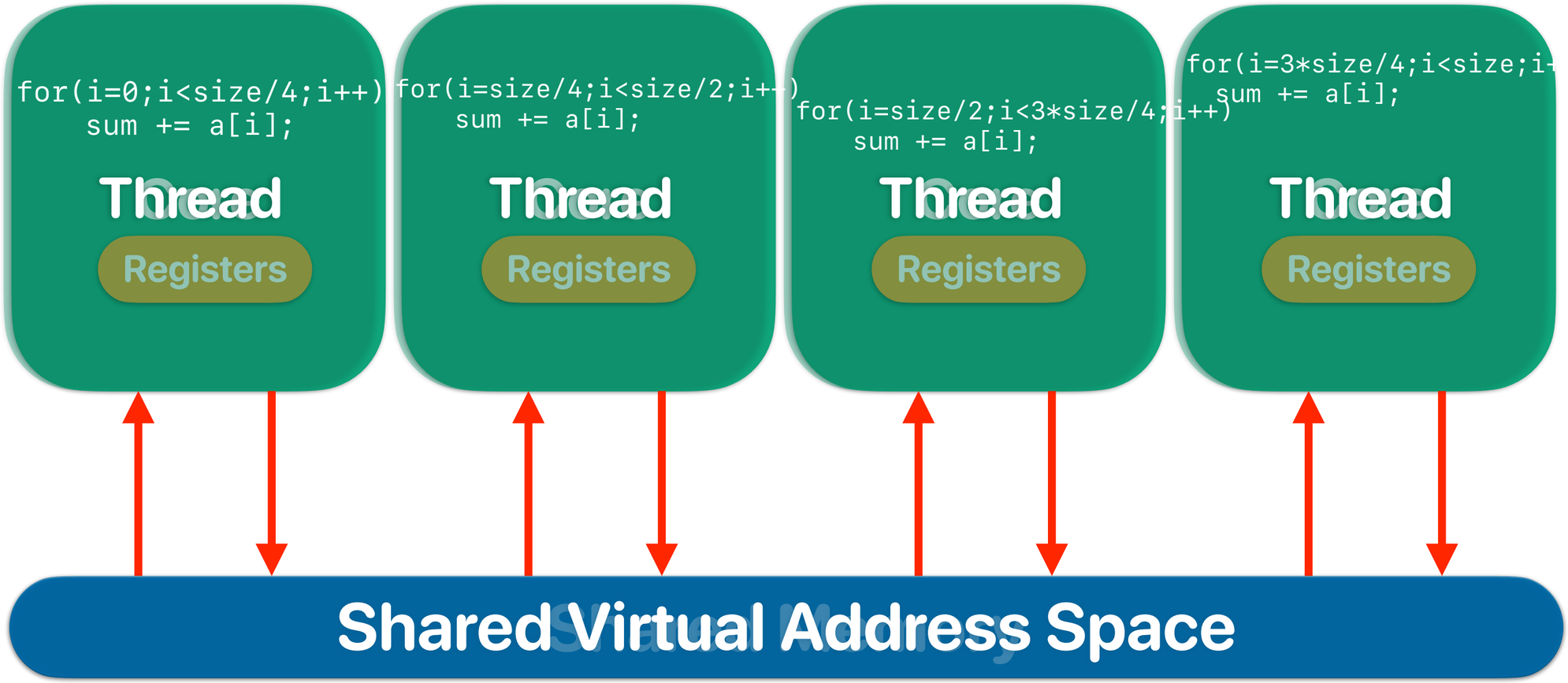


Parallelism in modern computers

- SISD — single instruction stream, single data (ILP)
  - Pipelining instructions within a single program
  - Superscalar
- SIMD — single instruction stream, multiple data (DLP)
  - Vector instructions
  - GPUs
- MIMD — multiple instruction stream (e.g. multiple threads, multiple processes), multiple data (TLP)
  - Multicore processors
  - Multiple processors
  - Simultaneous multithreading

Thread programming

The “abstracted” multithreading machine



Thread programming — POSIX threads

- pthread_t — thread descriptor
- pthread_init() — init a thread descriptor
- pthread_create() — create a thread running a “start_routine” function with “arg” as the argument

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

- pthread_join() — synchronize thread execution
- pthread_mutex_lock, pthread_mutex_unlock — managing a lock

C++ abstraction

```
t = fiddle("threads.cpp", function="threads", opt="-O1", cmdline=r"",
code=r"""
#include"function_map.hpp"
#include<cstdint>
#include<thread>

void go() {
    for(int i = 0; i < 15; i++)
        std::cerr << i << "\n";
}

extern "C"
uint64_t* threads(uint64_t threads, uint64_t * data, uint64_t size, uint64_t arg1, uint64_t arg2, uint64_t arg3) {
    std::thread t1(go);  // Create a thread to run go().  Pass no arguments.
    std::thread t2(go);
    std::thread t3(go);

    t1.join(); // wait for t1 to finish.
    t2.join();
    t3.join();
    std::cerr << "FINISHED EXECUTION\n";
    return data;
}
FUNCTION(one_array_2arg, threads);
""")
```

the header file

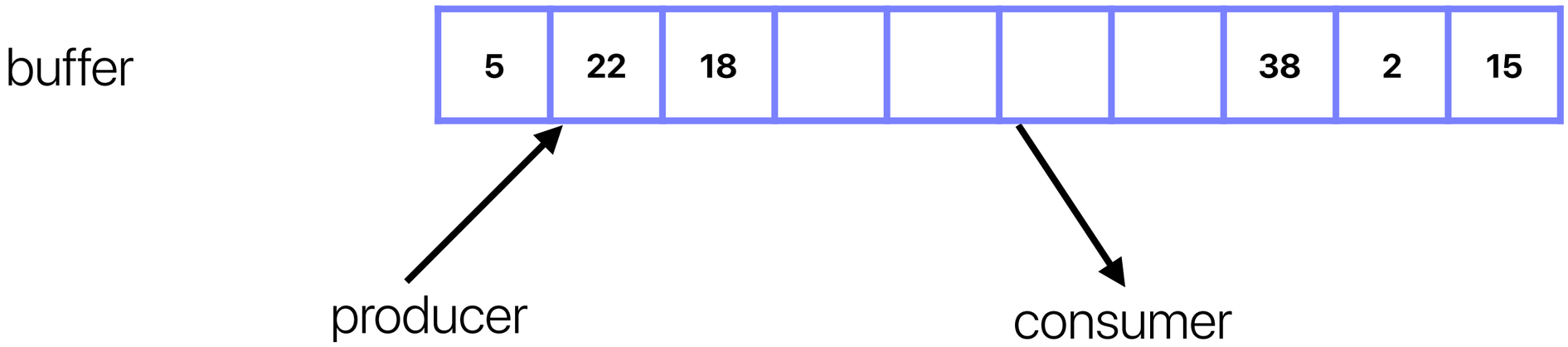
the thread function

pthread_init()/pthread_create()

pthread_join()

Bounded-Buffer Problem

- Also referred to as "producer-consumer" problem
- Producer places items in shared buffer
- Consumer removes items from shared buffer



Will the code function? (Related to Q2)

```
int buffer[BUFF_SIZE]; // shared global
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
       int item = …;

       buffer[in] = item;
       in = (in + 1) % BUFF_SIZE;

    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {

        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;

        // do something w/ item
    }
    return NULL;
}
```

Use locks

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if the thread crashes/halts here?

what if context switch happens here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

what if the thread crashes/halts here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;         // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

all threads can see lock as 0 at this point

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

what if the thread crashes/halts here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;         // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

all threads can see lock as 0 at this point

coherence cache misses? page fault?

We must use atomic instructions

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
```

```
static inline uint xchg(volatile unsigned int *addr,
unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}
```

exchange the content in %0 and %1

a prefix to xchgl that locks the whole cache line

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    // what code should go here?
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    // what code should go here?
}
```

We must use atomic instructions

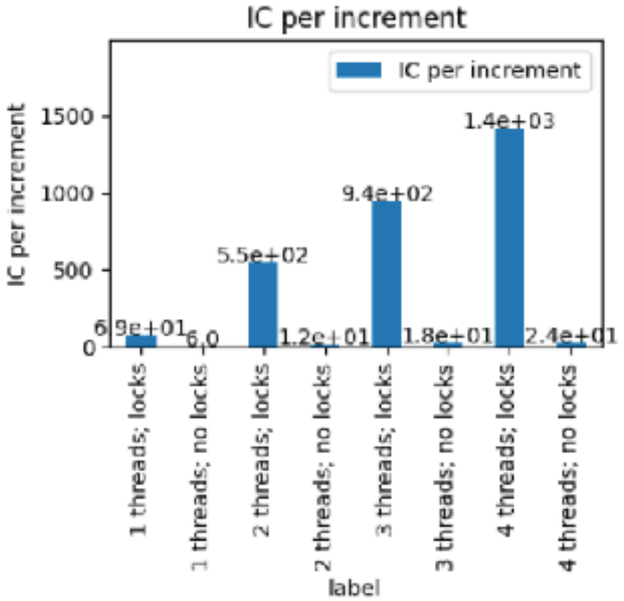
```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

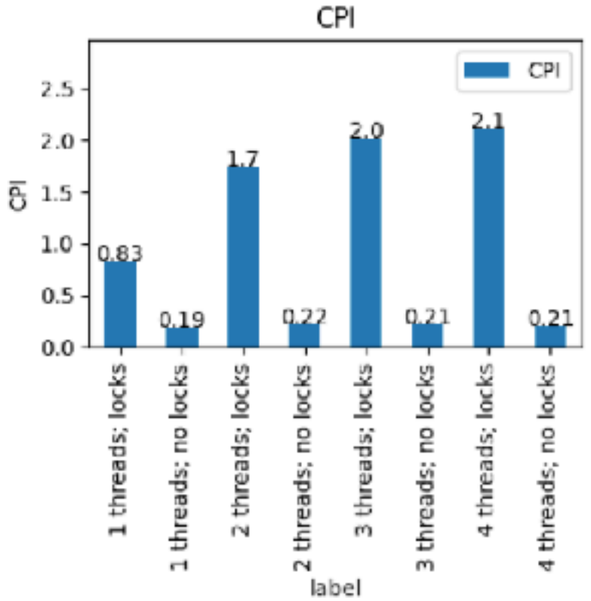
```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock
    }
    printf("parent: end\n");
    return 0;
}
```

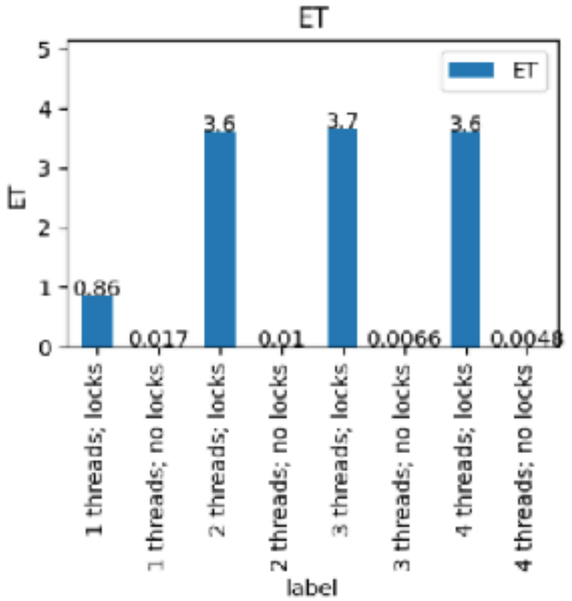
```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
```

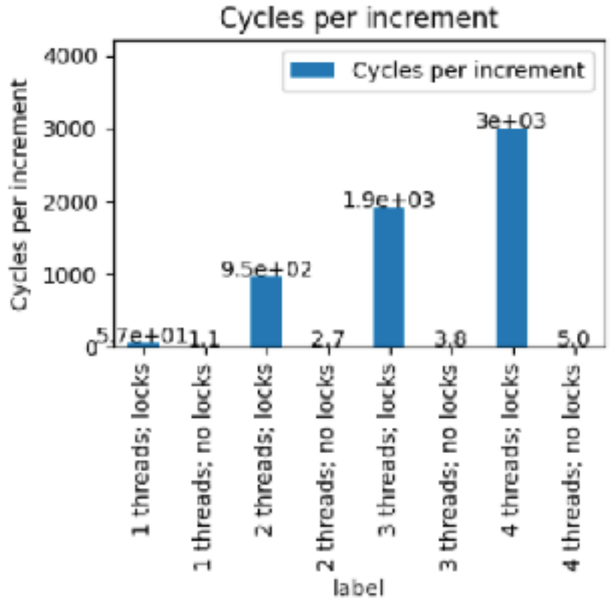
```
static inline uint xchg(volatile unsigned int *addr,
unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}

void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1);
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}
```

Q3-Q4, Q8—Q10: Locks are expensive...

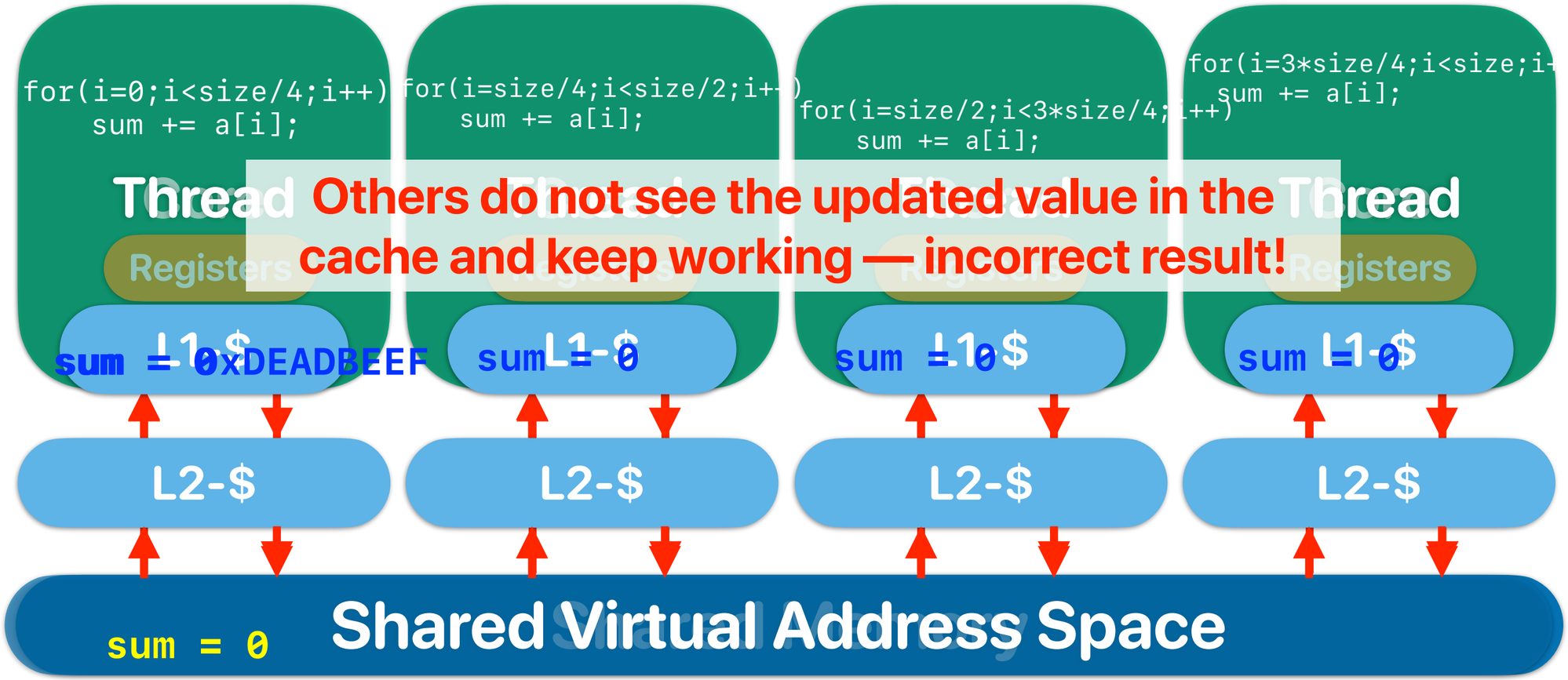






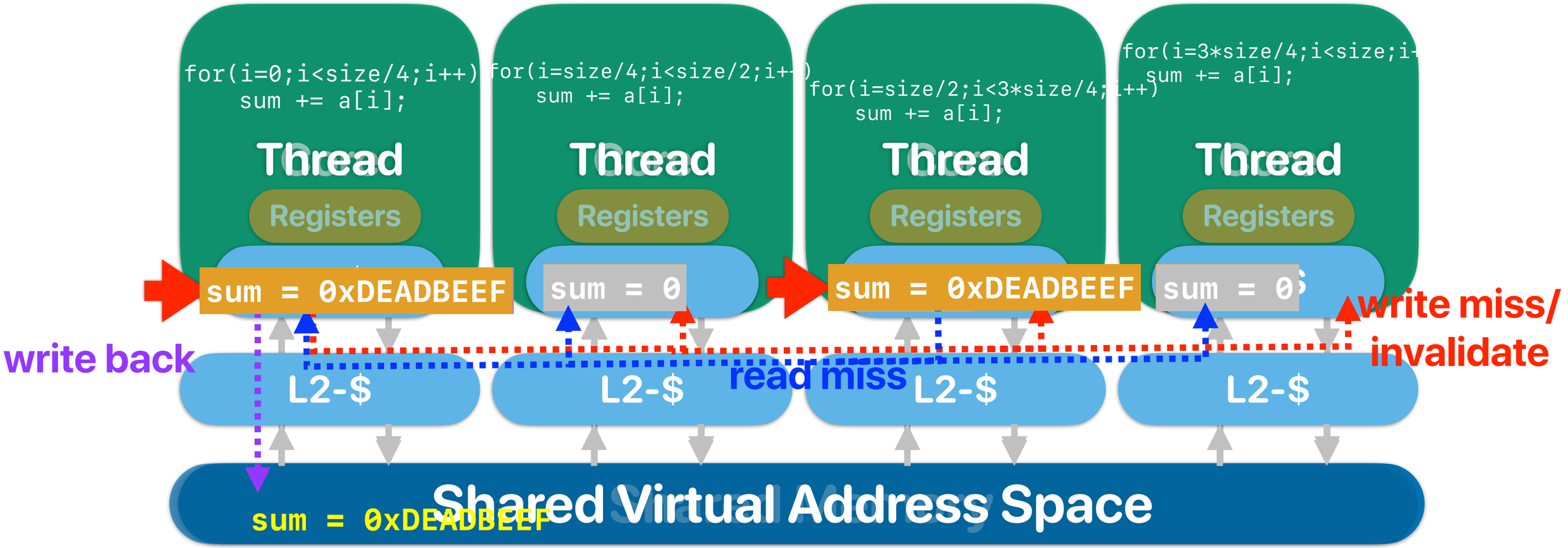


What software thinks about “multiprogramming” hardware

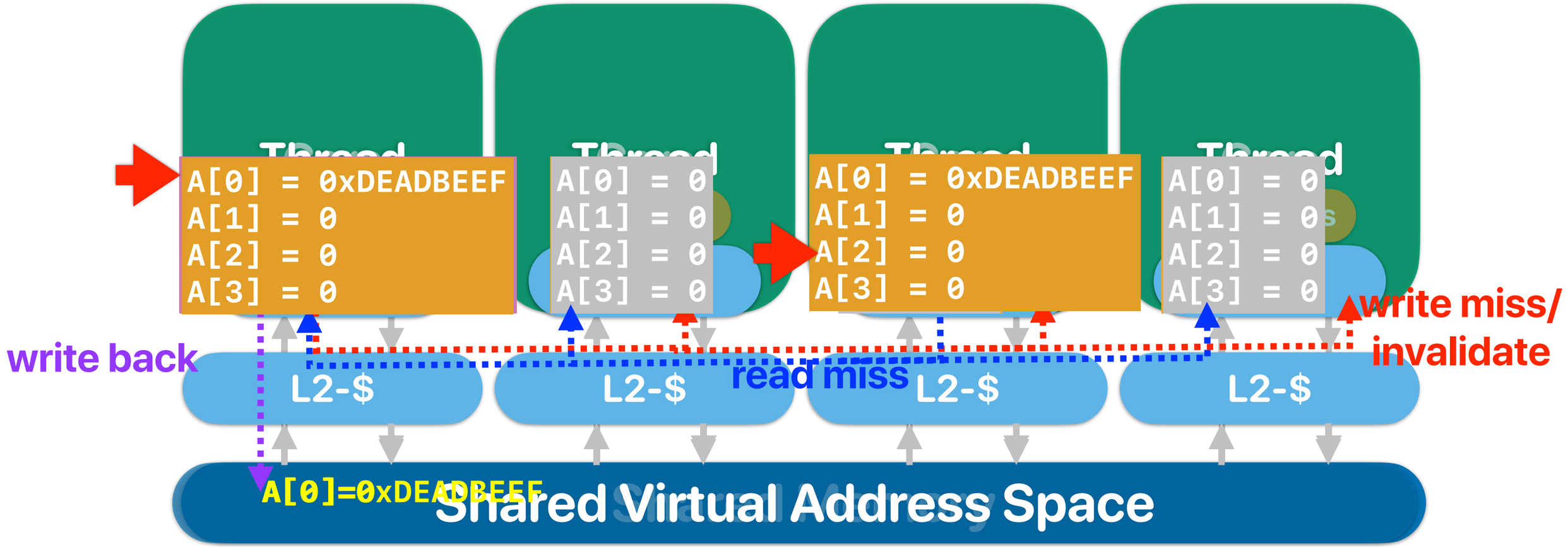


Others do not see the updated value in the cache and keep working — incorrect result!

What happens when we write in coherent caches?



Cache coherency



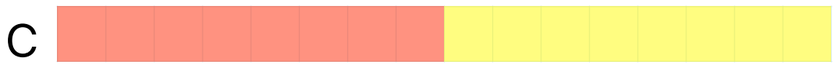
Q5—Q6: L v.s. R

Version L

```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
  {
        c[i] = a[i] + b[i];
  }
  return NULL;
}
```

Version R

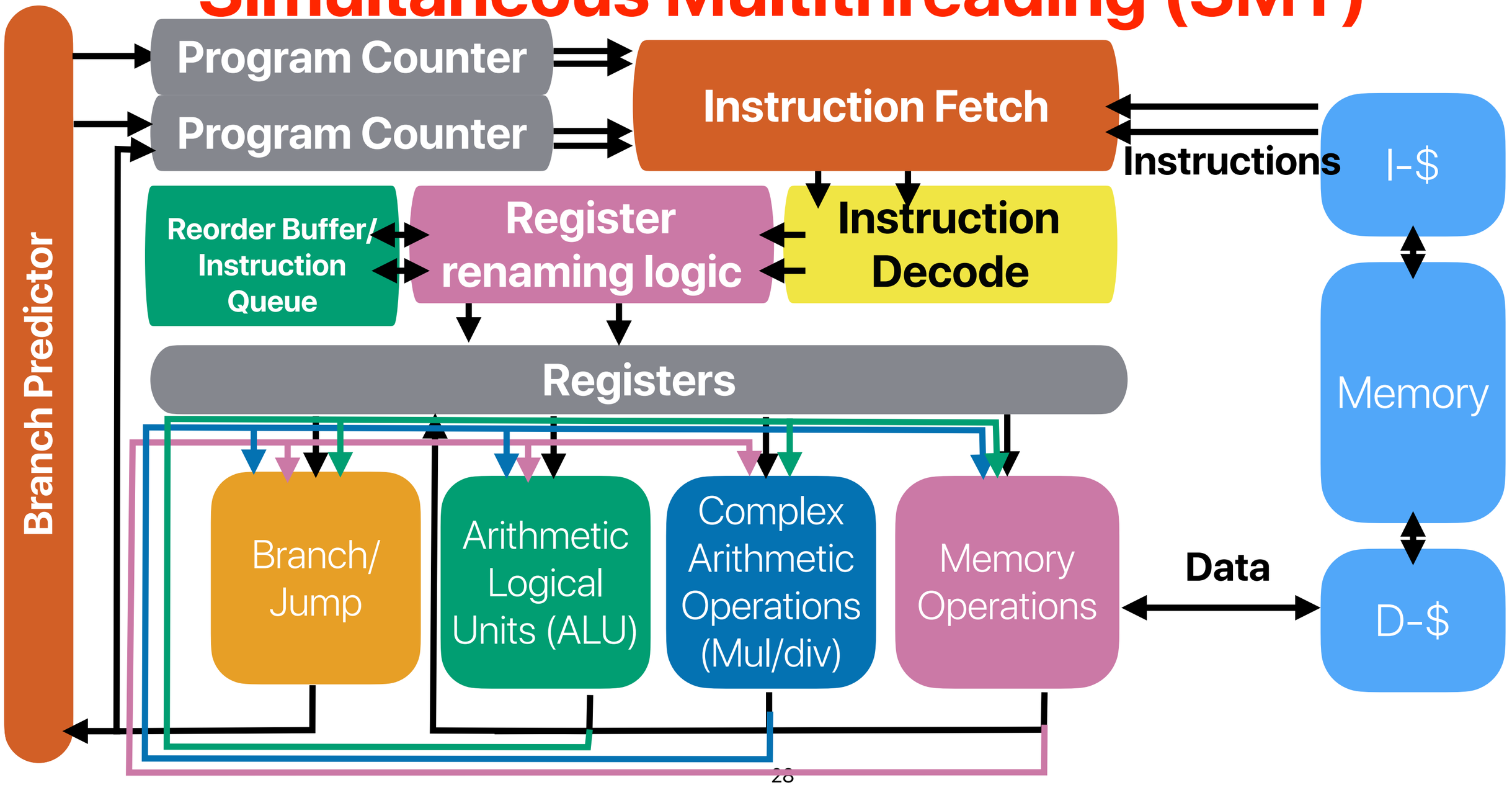
```
void *threaded_vadd(void *thread_id)
{
  int tid = *(int *)thread_id;
  int i;
  for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
  {
      c[i] = a[i] + b[i];
  }
  return NULL;
}
```



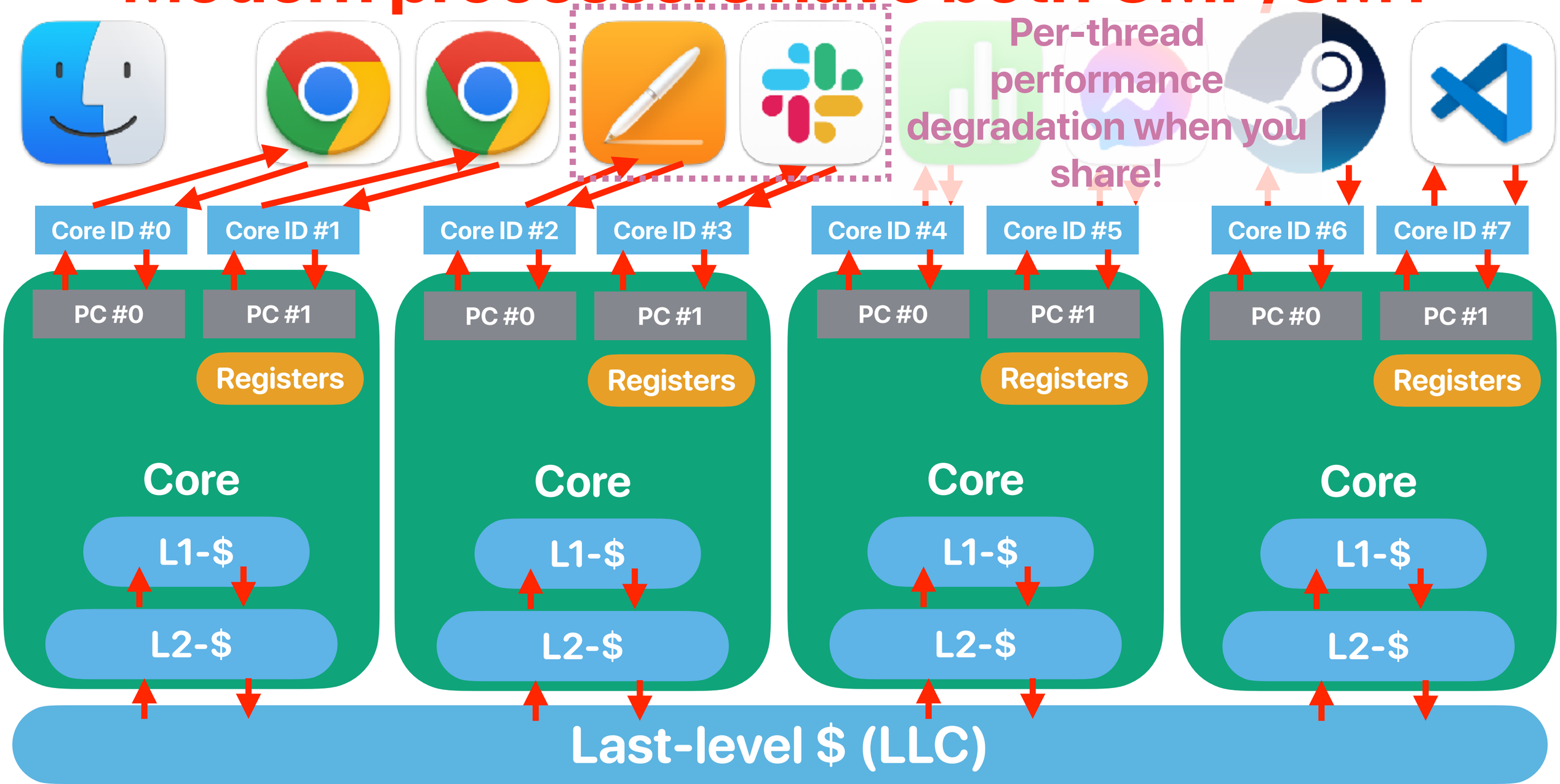
Tips of performance thread programming

- Reduce sharing: coherence misses
- Avoid fine-grained locks: serialization of execution
- Avoid short threads: thread spawning overhead

Simultaneous Multithreading (SMT)

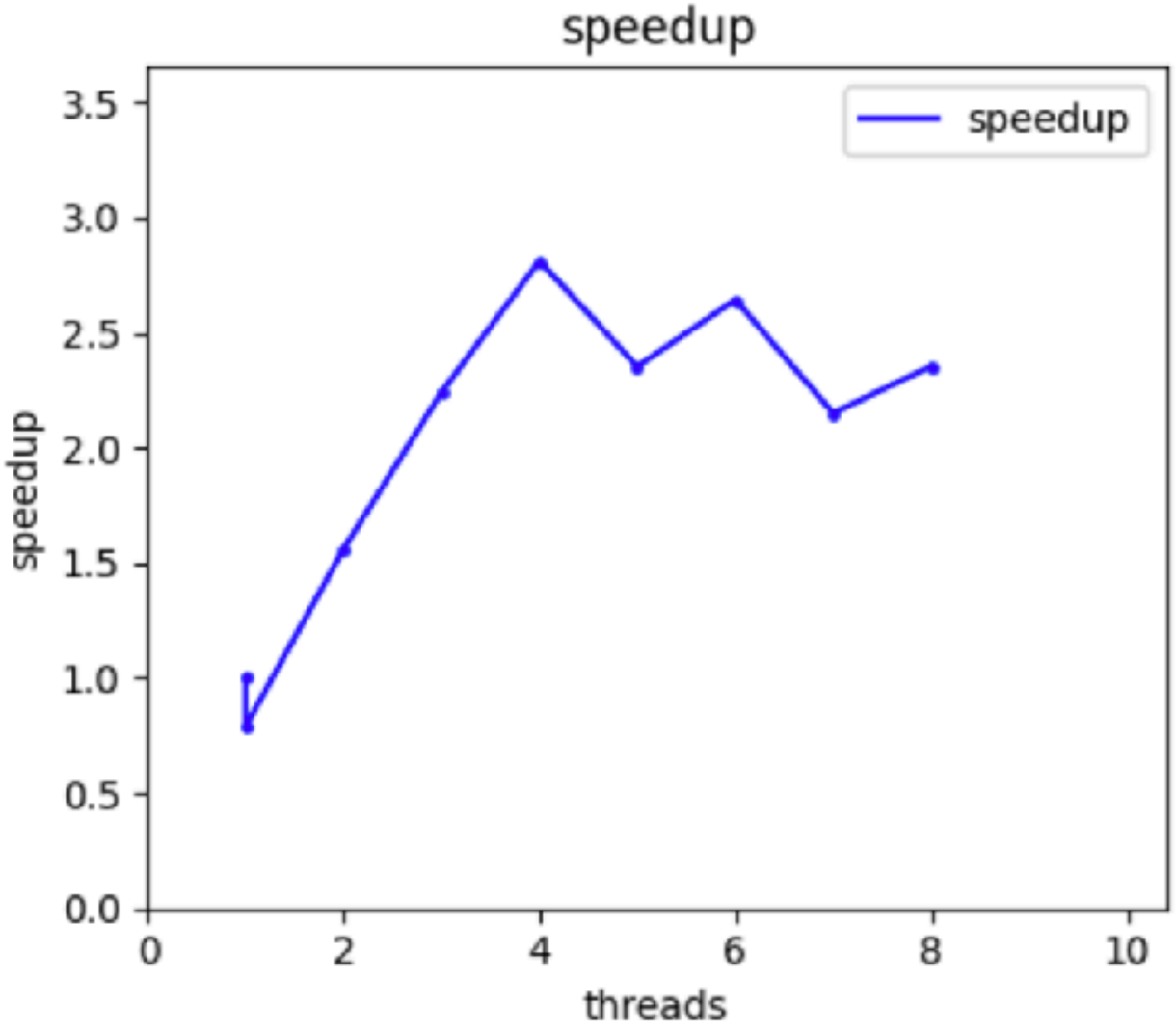


Modern processors have both CMP/SMT



We're using an SMT processor

```
]: ! cse142 job run "lscpu"

Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           151
Model name:                      12th Gen Intel(R) Core(TM) i3-12100F
Stepping:                        5
CPU MHz:                         3300.000
CPU max MHz:                     5500.0000
CPU min MHz:                     800.0000
BogoMIPS:                        6604.80
Virtualization:                  VT-x
L1d cache:                       192 KiB
L1i cache:                       128 KiB
```



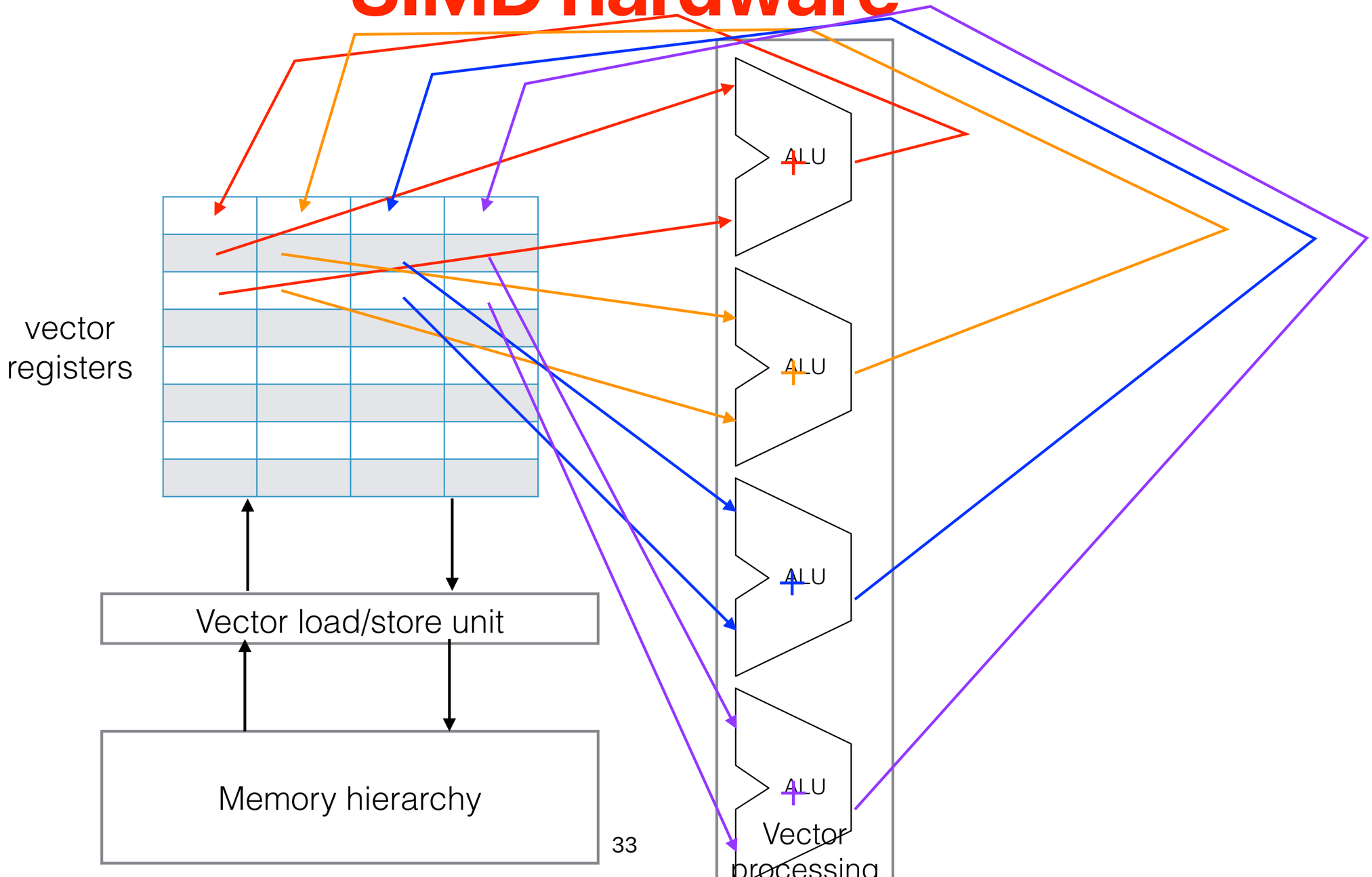
Data level parallelism

SIMD in processors

- SIMD: Single instruction, multiple data
  - Each instruction can perform operations on several datasets concurrently
- Streaming SIMD Extensions (SSE) that allows x86 processor architectures to support “SIMD” execution model
- ARM’s NEON

$$\begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix} + \begin{bmatrix} 5.0 \\ 6.0 \\ 7.0 \\ 8.0 \end{bmatrix} = \begin{bmatrix} 6.0 \\ 8.0 \\ 10.0 \\ 12.0 \end{bmatrix}$$

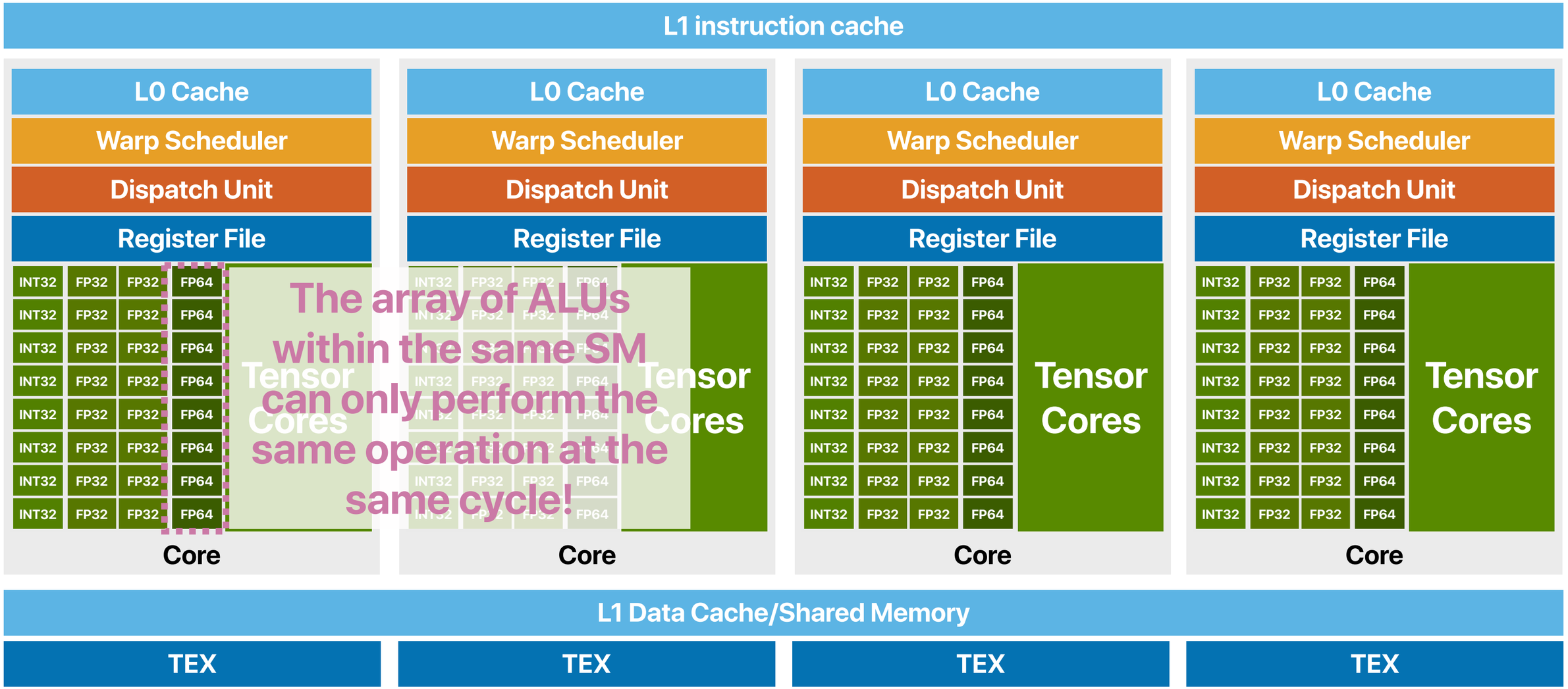
SIMD hardware



Streaming SIMD Extensions (SSE)

- SSE, introduced by Intel in 1999 with the Pentium III, creates eight new 128-bit registers
  - Added 8 128-bit registers — XMM0-XMM7
  - You may use each register to store
    - 2 double-precision floating point numbers
    - 4 single-precision floating point numbers
  - Extended since introduced — SSE2, SSE3, SSE4, SSE4.1, SSE4.2, SSE4a
- They are processor-dependent instructions
  - AMD RyZen supports SSE4a, SSE4.1, SSE4.2
  - intel Core i7 doesn't support SSE4a
  - VIA Nano only support SSE4.1

DLP: NVIDIA GPU Architectures



Matrix multiplication with SSE4

```
void vector_blockmm(double **a, double **b, double **c)
{
  int i,j,k, ii, jj, kk, x;
  __m256d va, vb, vc;  // compiler would allocate a register as long as these variables can fit
  for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
      for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
          for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
              for(ii = i; ii < i+(ARRAY_SIZE/n); ii++) {
                  for(jj = j; jj < j+(ARRAY_SIZE/n); jj+=VECTOR_WIDTH) {
                      vc = _mm256_load_pd(&c[ii][jj]);     // load values into a vector register

                      for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                      {
                          va = _mm256_broadcast_sd(&a[ii][kk]);     // load one value & fill the vector register
                          vb = _mm256_load_pd(&b[kk][jj]);            // load values into a vector register
                          vc = _mm256_add_pd(vc,_mm256_mul_pd(va,vb));      // vector multiplication
                      }
                      _mm256_store_pd(&c[ii][jj],vc);                  // store values into a vector register
                  }
              }
          }
      }
  }
}
```

Using OpenMP for both TLP/DLP

Three things you need to know about OpenMP

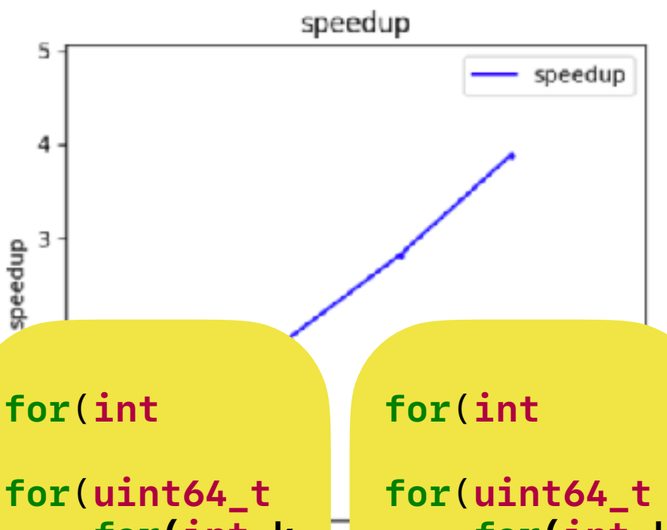
- A C/C++ extension with compiler supports
- They all start with "`#pragma omp`"
- **`#pragma omp parallel`** for
  - Run the following for loop with **multiple threads**.
  - The loop needs to be pretty simple.
  - Something like "for(int i = C; i < K; i+=B)"
    - **K and B need to fixed for the execution of the loop**
    - Otherwise, the compiler will ignore the #pragma
  - Apply to one loop
    - Nesting is not productive
    - Use an outer loop – Bigger chunks of work for the threads.

Three things you need to know about OpenMP

- `#pragma omp critical` for critical sections.
  - The next statement/code block will be considered as critical section
  - Automatically insert calls to thread locking primitives
- `#pragma omp simd` for vectorizing loops
  - Consider each loop iteration can be considered independent from each other
  - Will use SIMD/MMX/SSE/AVX instructions to vectorize the operations

Histogram in OpenMP

```
// histogram.cpp:277-310 (34 lines)
//START_OPENMP_PRIVATE

extern "C"
uint64_t* run_openmp_private_histogram(uint64_t thread_count, uint64_t * data,
uint64_t size, uint64_t arg1, uint64_t arg2, uint64_t arg3) {

    for(int i =0; i < 256;i++) {
        histogram[i] = 0;
    }

#pragma omp parallel for
    for(uint64_t ii = 0; ii < size; ii+=arg1) {
        uint64_t my_histogram[256];
        for(int i =0; i < 256;i++) {
            my_histogram[i] = 0;
        }
        for(uint64_t i = ii; i < size && i < ii + arg1; i++) {

            for(int k = 0; k < 64; k+=8) {
                uint8_t b = (data[i] >> k)& 0xff;
                my_histogram[b]++;
            }
        }
#pragma omp critical
        for(int i =0; i < 256;i++) {
            histogram[i] += my_histogram[i];
        }
    }
    return data;
}
//END_OPENMP_PRIVATE
```




```
for(int

for(uint64_t
    for(int k
```
Thread

```
for(int

for(uint64_t
    for(int k
```
Thread

```
for(int

for(uint64_t
    for(int k
```
Thread

```
for(int

for(uint64_t
    for(int k
```
Thread

```
for(int i =0; i < 256;i++) {
    histogram[i] += my_histogram[i];
}
```
critical section

```
for(int i =0; i < 256;i++) {
    histogram[i] += my_histogram[i];
}
```
critical section

```
for(int i =0; i < 256;i++) {
    histogram[i] += my_histogram[i];
}
```
critical section

Vector Sum in OpenMP

```
extern "C"
uint64_t vsum(uint64_t *a, uint64_t* b, uint64_t * c, uint64_t len)
{
    uint64_t s = 0;
    for(unsigned int i = 0; i < len; i++) {
        c[i]=a[i]+b[i];
    }
    return s;
}

extern "C"
uint64_t vsum_simd(uint64_t *a, uint64_t* b, uint64
{
    uint64_t s = 0;
#pragma omp simd
    for(unsigned int i = 0; i < len; i++) {
        c[i]=a[i]+b[i];
    }
    return s;
}


extern "C"
uint64_t* dp(uint64_t threads, uint64_t * list, uin
    if(arg1 == 0) {
        list[0] = vsum(list, &list[size/3], &list[s
    } else if(arg1 == 1){
        list[0] = vsum_simd(list, &list[size/3], &list[size*2/3], size/3);
    }
    return list;
}
//END_OPENMP_PRIVATE
```

```
vsum:
.LFB2984:
        .cfi_startproc
        endbr64
        testq   %rcx, %rcx
        je      .L13
        xorl    %r9d, %r9d
        xorl    %eax, %eax
        .p2align 4,,10
        .p2align 3
.L14:
        movq    (%rsi,%rax,8), %r8
        addq    (%rdi,%rax,8), %r8
        movq    %r8, (%rdx,%rax,8)
        leal    1(%r9), %eax
        movq    %rax, %r9
        cmpq    %rcx, %rax
        jb      .L14
.L13:
        xorl    %eax, %eax
        ret
        .cfi_endproc
.LFE2984:
```

```
vsum_simd:
.LFB2985:
        .cfi_startproc
        endbr64
        testl   %ecx, %ecx
        je      .L20
        cmpl    $1, %ecx
        je      .L24
        movl    %ecx, %r8d
        xorl    %eax, %eax
        shrl    %r8d
        salq    $4, %r8
        .p2align 4,,10
        .p2align 3
.L22:
        movdqu  (%rdi,%rax), %xmm0
        movdqu  (%rsi,%rax), %xmm1
        paddq   %xmm1, %xmm0
        movups  %xmm0, (%rdx,%rax)
        addq    $16, %rax
        cmpq    %rax, %r8
        jne     .L22
        movl    %ecx, %eax
        andl    $-2, %eax
        andl    $1, %ecx
        je      .L20
.L21:
        movq    (%rsi,%rax,8), %rcx
        addq    (%rdi,%rax,8), %rcx
        movq    %rcx, (%rdx,%rax,8)
.L20:
        xorl    %eax, %eax
        ret
```

Programming assignment

Matrix exp: $B = A^N$

```
template<typename T>
void __attribute__((noinline)) matexp_solution(tensor_t<T> & dst, const tensor_t<T> & A, uint32_t power,
            // parameters you can use for whatever purpose you want (e.g., tile sizes)
            int64_t p1=1,
            int64_t p2=1,
            int64_t p3=1,
            int64_t p4=1,
            int64_t p5=1) {
    // Tags for moneta

    TAG_START("dst", dst.start_address(), dst.end_address(), false);
    TAG_START("A", A.start_address(), A.end_address(), false);

    // In psuedo code this just
    //
    // dst = I
    // for(i = 0..p)
    //    dst = dst * A

    // Start off with the identity matrix, since M^0 == I
    // The result will end up in dst when we are done.
    for(int32_t x = 0; x < dst.size.x; x++) {
        for(int32_t y = 0; y < dst.size.y; y++) {
            if (x == y) {
                dst.get(x,y) = 1;
            } else {
                dst.get(x,y) = 0;
            }
        }
    }


    for(uint32_t p = 0; p < power; p++) {
        tensor_t<T> B(dst); // Copy dst, since we are going to modify it.
        TAG_START("B", B.start_address(), B.end_address(), false);
        mult_solution(dst,B,A, p1,p2,p3, p4,p5); // multiply!
        TAG_STOP("B");

    }

    TAG_STOP("dst");
    TAG_STOP("A");
```

```
template<typename T>
void  __attribute__((noinline)) mult_solution(tensor_t<T> &C, const tensor_t<T> &A,
                            int64_t p1=1,
                            int64_t p2=1,
                            int64_t p3=1,
                            int64_t p4=1,
                            int64_t p5=1) {

    // This is just textbook matrix multiplication.

    for(int i = 0; i < C.size.x; i++) {
        for(int j = 0; j < C.size.y; j++) {
            C.get(i,j) = 0;
            for(int k = 0; k < B.size.x; k++) {
                C.get(i,j) += A.get(i,k) * B.get(k,j);
            }
        }
    }
}
```

Continue from Lab 2

- Same problem as last time.
- Now with threads and vectors
- Speedup targets — Overall you need to achieve ~40x speedup
- Get your work done in matexp_solution.hpp
- You're supposed to use OpenMP
  - You may try other available tools on datahub, not supported.

Announcement

- No more lab lectures
- Two in-person office hours by Hung-Wei Tseng next week
  - TuTh 3:30p-5:30p @ CSE 3128

Computer
Science &
Engineering

142L

最終回