

ECE 574 – Cluster Computing

Lecture 12

Vince Weaver

<https://web.eece.maine.edu/~vweaver>

vincent.weaver@maine.edu

30 September 2026

Announcements

- Homework #4 was finally posted
- It's a relatively short homework unless you attempt the extra credit coding

SIMD Redux

SSE example (From Wikipedia)

Doing a 4 element single-precision vector add would take 4 separate floating point adds:

```
vec_res.x = v1.x + v2.x;  
vec_res.y = v1.y + v2.y;  
vec_res.z = v1.z + v2.z;  
vec_res.w = v1.w + v2.w;
```

With SSE you only need one add instruction:

```
; move aligned packed single precision  
movaps xmm0, [v1]           ;xmm0 = v1.w / v1.z / v1.y / v1.x  
addps  xmm0, [v2]           ;xmm0 = v1.w+v2.w / v1.z+v2.z /  
                             ;      v1.y+v2.y / v1.x+v2.x  
  
movaps [vec_res], xmm0
```

Intrinsics

TODO: make this match the previous example

```
__m256i in1;
```

```
/* vmovdqa (%rcx),%ymm1 */
```

```
__m256i filter_avx = _mm256_load_si256( (__m256i *)filter);
```

Enabling with Compiler

- To enable any vectorization on gcc need `-O3`
- To get AVX need to specify something like `-mavx`, this can be set if you set the proper architecture `-march=haswell` or `-march=native`
- To see if any vectors worked `-fopt-info-vec`
- To see why not vectorizing: `-fopt-info-vec-missed`

Intrinsics

- One way is to program in assembly language with some obscure opcodes: an example PMADDWD 16-bit integer parallel 128-bit multiply and add
- On recent gcc and other compilers there are “intrinsics” to use in C, for example you can use `_mm_madd_epi16()` to do a PMADDWD instruction
- Before using, need to include headers

```
#include <emmintrin.h>
#include <smmmintrin.h>
#include <immintrin.h>
```
- <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>

Initial SIMD Sobel try

9 values from the three input pointers (16-bit)

A B C X D E F X G H I X X X X X

The sobel filter values (16-bit)

1 2 3 0 4 5 6 0 7 8 9 0 0 0 0 0

Multiply and add all in parallel

A1+B2 C3+0 D4+E5 F6+0 G7+H8 I9+00 0+0 0+0

Rearrange and then do a "horizontal add"

A1+B2+G7+H8 C3+I9 D4+E5 F6+0

Another Horizontal Add

0 0 A1+B2+G7+H8+C3+I9 D4+E5+F6

Another Horizontal Add

0 0 0 A1+B2+G7+H8+C3+I9+D4+E5+F6

Convert to 16-bit result, saturate, and be done

Problem: AVX2 the two "lanes" (top/bottom 128)

are separate and so need extra work to combine

Problem: horizontal add is slow

Problems

- Math is very fast, handfull of instructions
- Problem is getting memory from 3 pointers with 3-byte offsets into registers (can we make jpeg load as 4-bytes?)
- This is a “scatter/gather” problem found often with SIMD (and GPU)
- There are instructions to try to gather the values together, but not really suited for this
- Once you do it manually performance might be worse than regular code
- Challenge: if picture not multiple of 16-bytes

Improved SIMD – Can we do better?

With many problems: re-think outside the serial box

Load full 16 bytes of pixel info from the three pointers,
multiply by the 9 values in sobel filter, shifting right by 3

```
A * RGB RGB RGB RGB RGB RGB R
B *      RGB RGB RGB RGB RGB R
C *      RGB RGB RGB RGB RGB R
D * RGB RGB RGB RGB RGB RGB R
E *      RGB RGB RGB RGB RGB R
F *      RGB RGB RGB RGB RGB R
G * RGB RGB RGB RGB RGB RGB R
H *      RGB RGB RGB RGB RGB R
+ I *      RGB RGB RGB RGB RGB R
=====
```

RGB RGB RGB RGB R 13 values of result

Use compare instruction to saturate in parallel

Store out the 13 bytes at once

So (18*13) operations reduced to (~20) I think. Still haven't tried this yet

Combine Goes better

Handle 8 at a time
Load in 8 bytes from sobel_x and y
Expand to 32-bit floating point numbers
Multiply together to square
Add two multiply results
Take squareroot

Saturate in floating point with min/max

Have the result in 32-bit float, trouble
converting back to 8-bits
Convert to 32-bit int
Shuffle to grab raw bytes from the ints
256-bit is two 128-bit lanes to some
work to get the top values
Permute down to 64-bit result
Write out

Handling Concurrency

- In multi-threaded programs if two threads try to write to the same location at once, this can cause issues

Race Conditions

```
x=0; // global variable
```

A	B
x=x+1;	x=x+1;
ldr r0,x	ldr r0,x
add r0,#1	add r0,#1
str r0,x	str r0,x

- Shared counter address
- RMW on ARM
- Thread A reads value into reg
- Thread B reads value into reg, increments, writes out
- Thread A increments value, writes out
- What happened?
- What should the value be?

Critical Sections

- Want mutual exclusion, only one thread can access protected area at a time
 - no two processes can be inside critical section at once
 - no assumption can be made about speed of CPU
 - processes not in critical section may not block other processes
 - no process should wait forever

How to avoid

- Single Core machines (can still hit this issue)
 - Interrupts can switch to another process/thread
 - Disable interrupts. Heavy handed. Probably not allowed if operating system
- All Systems
 - Locks/mutex/semaphore

Locks

- Allow locking a chunk of code (critical section)
 - Only one thread is allowed into the critical section
 - Other threads must wait
 - Note: protects code, not memory. You might need to use the same lock in various locations if the same memory is accessed in different places
- There are many ways to implement this

Mutex (Mutual Exclusion) Lock

- A lock is declared, possibly just an integer
- Code calls `mutex_lock(&lock)`
 - If unlocked (0), then it sets lock and returns
 - If locked, returns 1, does not enter
- What do we do if locked?
 - Busy wait? (spinlock)
 - re-schedule? (yield)
- `mutex_unlock()`: sets variable to zero

Semaphore

- Counting vs Binary
- Allows locking with priority (mutex it's possibly luck which waiting thread gets to run)
- Up/Down (V increments, P decrements)
- Wait in queue
- Blocking
- As lock frees, the job waiting is woken up

Can you write your own lock in plain C?

```
int lock=0;

try_again:
if (lock==0) {
    lock=1;  // take lock
    ....
    lock=0;  // release lock
} else {
    goto try_again;
}
```

- Problem happens between checking if lock is free and setting it taken. If another task can also pass the check before set, things break. need atomic test-and-set or similar

Need for Memory Barriers

- Another issue is w/o barriers, loads and stores can be re-arranged and accesses can leak into/out-of critical section
- Modern out-of-order processors can execute loads or stores out-of-order
- What happens a load or store bypasses a lock instruction?
- Barriers can enforce this doesn't happen
- Processor Memory Ordering Models, as discussed earlier
- Special assembly language instructions

Volatile Variables (aside)

- When accessing shared variables is it ever a problem that the compiler or CPU might have the value in a register and not notice memory accesses from other threads?
- In this case should you force memory accesses while using the C `volatile` keyword?
- In theory the answer is no. When using pthreads and using locking the software stack should make sure any atomic accesses or barriers happen when they should
- You will find a lot of people will debate this though
- You may run into problems if you try to open-code locks or condition variables yourself in plain C

Locking Primitives

- To do proper locking need low-level CPU support
- Generally need assembly language help as not possible to express in C
- Often need some sort of “atomic” instruction (atomic as in “indivisible”)

Some Locking Primitives

- fetch and add (bus lock for multiple cores), xadd (x86)
- test and set (atomically test value and set to 1)
- test and test and set

More Locking Primitives – Compare and Swap

- Atomic swap instruction SWP (ARM before v6, deprecated)
- x86 CMPXCHG
- Does both load and store in one instruction!
- Why bad? Longer interrupt latency (can't interrupt atomic op)
- Especially bad in multi-core

More Locking Primitives – LL/SC

- load-link/store conditional
- Load a value from memory
- Later store instruction to same memory address.
- Only succeeds if no other stores to that memory location in interim.
- ldrex/strex (ARMv6 and later)

Transactional Memory

- (mostly abandoned)
- Tell your processor there's a critical section
- Processor notices if any writes happen from other threads
- Have to restart and try again
- Intel had support, had seen that fine-grained locking hard in Linux kernel. By the time the hardware support came out they had it mostly sorted

Locking Primitives – Equivalency

- can be shown to be equivalent
- how swap works:
lock is 0 (free). `r1=1; swap r1,lock`
now `r1=0` (was free), `lock=1` (in use)
lock is 1 (not-free). `r1=1, swap r1,lock`
now `r1=1` (not-free), `lock` still==1 (in use)

Deadlock

- Two processes both waiting for the other to finish, get stuck
- One possibility is a bad combination of locks, program gets stuck
- P1 takes Lock A. P2 takes Lock B. P1 then tries to take lock B and P2 tries to take Lock A.

Livelock

- Processes change state, but still no forward progress.
- Two people trying to avoid each other in a hall.
- Can be harder to detect

Starvation

- Not really a deadlock, but if there's a minor amount of unfairness in the locking mechanism one process might get “starved” (i.e. never get a chance to run) even though the other processes are properly taking and freeing the locks.

How to avoid Deadlock

- Don't write buggy code
- Pre-emption (let one of the stuck processes run anyway)
- Rollback (checkpoint occasionally)
- What to do if it happens?
 - Reboot the system
 - Kill off stuck processes