

ECE 574 – Cluster Computing

Lecture 14

Vince Weaver

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

vincent.weaver@maine.edu

5 October 2026

Announcements

- HW#5 was posted, due Friday
- Updated midterm date, 19th (Monday)
- Will post project info soon

Homework #5 – Coarse Code

- Just make two threads, one for sobelx, one for sobely
- Join at end before calling combine
- This part of the homework is to make sure you understand the basics of pthread programming
- Hardest part is probably putting all the arguments in one struct pointer, then doing the casting to call into it and then uncasting back in the function. I provide code to help with that.

Homework #5 – Fine Code

- This is much harder
- You can do this any way you want, but below is a suggested way
- If parallelizing for N threads you need to split up the work N ways
- While it might be tempting to come up with a complex scheme to have x , y , and combine all going at once, it might be best to keep it simple (and that might actually be faster)

Homework #5 – Fine Code Borders

- Modify the parameter structure so it takes a start and end value
- Modify the convolution loops to go from ystart to yend rather than 0 to ysize
- Before launching the thread, set up the boundaries in the struct
 - thread0 run from $0..(ysize/N)$
 - thread1 from $(ysize/N)..((ysize/N)*2)$
 - etc
- Note that C loops when you use less than it stops 1 short, which is why the boundaries above might look like they are off by 1

Homework #5 – Edge Cases

- Be sure for the last end value that you handle workloads not divisible by N
- The simplest way to do that is to just have a special case for the last range, and have it end at y_{size} rather than $(y_{\text{size}}/N)*N$
- Also remember to avoid the edges (row 0 and last row). You can do this outside or inside the convolve, but just be careful it only happens once

Homework #5 – Allocating Threads

- You will need to have N copies of the parameter struct to use to pass to the N threads.
- You can't re-use a single one, as it's global state and you'll have a race condition
- See the `pthread_join.c` example from last class for example code of this being done

HW#3 – General Comments

- Comment your code!
- Don't ignore compiler warnings!
- You can compare your butterfinger results against the provided ones. `md5sum` can be used for that.

HW#3 – Issues

- You need to saturate to 255 in combine function too
 $\text{sqrt}(255*255+255*255)$ is greater than 255.
If you wrap around in 8-bits your results will be off.
- Be sure the borders are 1 to X-1 and 1 to Y-1
- You can't use the modulus operator to saturate

HW#3 – Butterfinger Results

Butterfinger was a pet guinea pig from long ago.
Note on benchmark images, most famous for image processing “Lena”

```
time ./sobel ./butterfinger.jpg
output_width=320, output_height=320, output_components=3
SOBELX  L3 CACHE MISSES: 1554    CYCLES 9436089
SOBELY  L3 CACHE MISSES: 0       CYCLES 9362614
COMBINE L3 CACHE MISSES: 3       CYCLES 6574264

real    0m0.048s user    0m0.024s sys     0m0.004s
```

HW#3 – More Butterfinger Results

- Why 0 cache misses for SOBELY?
Cache. $320*320*3=307k$
IN, SOBEL_X, SOBEL_Y, COMBINED, so $300k*4 = 1.2MB$ or so
- Spacestation is $4288*2929*3 = 37MB$ or so
- Haswell-EP has 20MB of L3 cache
- Reading causes misses to read input in, rest are writing out so while not necessarily hits, with write allocate cache do not seem to be accounted for as misses
- Multiple runs the cache misses are lower, probably due to operating system disk cache

HW#3 – Haswell-EP Brief Cache Overview

- Haswell-EP caches (multiple levels)
 - memory – 200+ cycles best case
 - 20MB of L3, 20MB, 64B/line (30-60 cycles?)
 - 256kB per-core L2, 64B/line, 8-way (12-cycles)
 - 32kB per-core L1, 64B/line, 8-way (4 cycles)
- Note the L1 and L2 are split if hardware threading enabled

HW#3 – Haswell-EP More Cache

- Chunks of fast memory close to CPU
- Memory broken up into cacheline sized chunks (64-byte on HSW-EP)
- When access an address, all 64-B brought in even if not need rest
- When cache full, something is kicked out to make room (usually oldest)
- Want to take advantage of spatial and temporal locality
- With butterfinger all fits in L3 cache

HW#3 – Earth, Straight implementation of pse

```
./sobel ./earth_06_03_2018.jpg  
output_width=2048, output_height=2048, output_components=3  
SOBELX L3 CACHE MISSES: 318,572 CYCLES 559,078,407  
SOBELY L3 CACHE MISSES: 285,851 CYCLES 556,456,869  
COMBINE L3 CACHE MISSES: 593,838 CYCLES 335,950,332  
  
real    0m0.759s user    0m0.688s sys     0m0.032s
```

12MB, fits in cache?

HW#3 – Space Station, Straight implementation

Some perf results, if curious.

```
./sobel ./space_station_hires.jpg  
output_width=4288, output_height=2929, output_components=3  
L3 CACHE MISSES: 1,135,130          CYCLES 1,670,349,917  
L3 CACHE MISSES: 1,125,314          CYCLES 1,638,624,347  
L3 CACHE MISSES: 1,751,949          CYCLES 967,758,034  
  
real      0m1.741s  user      0m1.647s  sys       0m0.048s
```

HW#3 – Space Station, (simple)

perf report

67.88%	sobel	sobel	[.] generic_convolve
20.27%	sobel	sobel	[.] main
0.74%	sobel	[unknown]	[k] 0xfffffffffa1e00a27
0.33%	sobel	libjpeg.so.62.2.0	[.] 0x00000000000037902
0.27%	sobel	[unknown]	[k] 0xfffffffffa1e0015f
0.26%	sobel	libjpeg.so.62.2.0	[.] 0x00000000000037912
0.24%	sobel	libjpeg.so.62.2.0	[.] jpeg_fill_bit_buffer

HW#3 – Space Station, (simple)

perf annotate (weird that conditional move is at the top)

0.39	add	%r11d,%ebx	
2.86	cmp	\$0xff,%ebx	
3.57	cmovg	%eax,%ebx	
			output_image->pixels[(y*output_ima
0.04	mov	(%r12),%eax	
0.78	imul	%r14d,%eax	
0.05	add	%esi,%eax	
0.42	imul	0x8(%r12),%eax	
0.06	mov	0x10(%r12),%rsi	
0.69	add	%ecx,%eax	
0.18	test	%ebx,%ebx	
7.08	cmovs	%edi,%ebx	
1.82	cltq		

HW#3 – Space Station, (simple)

perf annotate last time

```
                                sum += filter[0][2]*(input_image->p
0.61 |      movslq %r11d,%r11
0.66 |      movzbl (%rcx,%r11,1),%esi
    |      convert():
    |          return (y*xsize*depth)+(x*depth)+color;
42.22 |      lea      (%r9,%rbx,1),%r11d
    |      generic_convolve():
```

HW#3 – More

- Conditional move?
- Compiler does wacky things. All mixed up. In-lined the combine routine.
- $4288 * 2929 = 36\text{MB}$ (larger than L3)
- If you compile with `-march=haswell` you'll get much different results, gcc these days can vectorize some stuff with AVX

HW#3 – Loop Order Optimization

- How is an array laid out in memory?
Row-major (C) vs Column-major (Fortran)
- Default with loop x then y, are actually walking columns. Worst case.
- Switch order of loops, things get a lot better.

```
time ./sobel_improved ./IMG_1733.JPG
output_width=3888, output_height=2592, output_components=3
SOBELX  L3 CACHE MISSES: 21,246 CYCLES 882,000,608
SOBELY  L3 CACHE MISSES: 19,556 CYCLES 881,998,207
COMBINE L3 CACHE MISSES: 1,241,446          CYCLES 1,183,759,970

real      0m1.181s user      0m1.112s sys      0m0.052s
```

HW#3 – Loop Unrolling

- Loop unrolling. Unroll the color loop (explicitly do the three things 0, 1, 2 and put the values in.
- Can have benefits. Change all occurrences of “color” to be a constant, which can be optimized.
- Remove branches, which can be slow or mispredicted.
- More code for out-of-order processor to work with and try to do in parallel.
- Downsides: if gets too large: no longer fit in instruction cache or loop stream detector.

HW#3 – Other Optimizations

- Other optimizations, often are things the compiler does for you with -O2.
- Hoisting (move things out of loop that only need to be done once)
- Simplification. Lots of things.
- Try another compiler (clang?)
- Take a compiler class.

HW#3 – Convert to one single Loop

No need to iterate X and Y and Color, just walk through output linearly. Really you have three pointers of input (line above, current line, below).

```
time ./sobel_improved ./IMG_1733.JPG
output_width=3888, output_height=2592, output_components=3
SOBELX  L3 CACHE MISSES: 15,703 CYCLES 411,148,087
SOBELY  L3 CACHE MISSES: 15,334 CYCLES 411,284,853
COMBINE L3 CACHE MISSES: 1,245,842      CYCLES 1,186,204,125

real    0m0.924s user    0m0.848s sys      0m0.044s
```

HW#3 – Same for Combine

No need to offset, just start at beginning of x and y and write to output, doing the combine operation.

```
time ./sobel_improved ./IMG_1733.JPG output_width=3888, output_height=2592, out$  
L3 CACHE MISSES: 16,188 CYCLES 410,983,833  
L3 CACHE MISSES: 14,850 CYCLES 411,059,831  
L3 CACHE MISSES: 36,652 CYCLES 496,394,104  
  
real    0m0.690s  
user    0m0.628s  
sys     0m0.040s
```

HW#3 – Same for Combine

ISRA= interprocedural scalar replacement of aggregates,

39.71%	sobel_improved	sobel_improved	[.] generic_convolve.isra.0
24.51%	sobel_improved	sobel_improved	[.] main
2.41%	sobel_improved	[kernel.kallsyms]	[k] clear_page_c_e
1.23%	sobel_improved	libjpeg.so.62.2.0	[.] jpeg_fill_bit_buffer
1.02%	sobel_improved	libjpeg.so.62.2.0	[.] 0x000000000039356
0.83%	sobel_improved	[kernel.kallsyms]	[k] page_fault

OpenMP – Open MultiProcessing

A few good references:

- <https://hpc-tutorials.llnl.gov/openmp/>
- <http://bisqwit.iki.fi/story/howto/openmp/>
- http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf

OpenMP

- Goal: parallelize serial code by just adding a few compiler directives here and there
- No need to totally re-write code like you would with pthread or MPI

OpenMP Background

- Shared memory multi-processing interface
- Main thread with Fork/Join methodology
- C, C++ and FORTRAN
- Industry standard made by lots of companies
- OpenMP 1.0 came out in 1997 (FORTRAN) or 1998 (C), newest version 6.0 (2024)

OpenMP Compiler Support

- gcc support “recently” (2008?) donated, CLANG even newer
- gcc added support in 4.2 (OpenMP 2.5)
4.4 (OpenMP 3.0), 4.7 (OpenMP 3.1), 4.9 (OpenMP 4.0), 5.0 (Offloading)
- gcc-14.0 still reports OpenMP 4.5 as its version but it supports some 5.0 features

OpenMP Interface

- Compiler Directives
- Runtime Library Routines
- Environment Variables

Communicating with Compiler

- Command line option
 - On gcc, pass `-fopenmp`
- Commands in code to indicate OpenMP directives
 - C: `#pragma omp`
 - FORTRAN: `C$OMP` or `!$OMP`

Compiler Directives

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

Library routines

- Need to `#include <omp.h>`
- Getting and setting the number of threads
- Getting a thread's ID
- Getting and setting threads features
- Checking if in parallel region
- Checking nested parallelism
- Locking
- Wall clock measurements

Environment Variables

- Setting number of threads (`OMP_NUM_THREADS = 4`)
- Configuring loop iteration division
- Processor bindings
- Nested Parallelism settings
- Dynamic thread settings
- Stack size
- Wait policy