ECE 574 – Cluster Computing
Lecture 10

Vince Weaver

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

vincent.weaver@maine.edu

16 February 2023

Announcements

- HW#5 will be posted, OpenMP
- I will send out HW#3 Grades

HW#3 – General Comments

- Please put results in the README file and submit using "make submit"
- Comment your code!
- Don't ignore compiler warnings!
- You can compare your butterfinger results against the provided ones. `md5sum` can be used for that.
- Issues I saw:
  - You need to saturate to 255 in combine function too 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
```

- 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
  - 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 L2, 64B/line, 8-way (4 cycles)
- Chunks of fast memory close to CPU
- Multiple levels
- 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 pseudo-code

```
./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 of pseudo-code

```
./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
```

```
perf report

  67.88%  sobel    sobel                [.] generic_convolve
  20.27%  sobel    sobel                [.] main
   0.74%  sobel    [unknown]            [k] 0xffffffffa1e00a27
   0.33%  sobel    libjpeg.so.62.2.0    [.] 0x0000000000037902
   0.27%  sobel    [unknown]            [k] 0xffffffffa1e0015f
   0.26%  sobel    libjpeg.so.62.2.0    [.] 0x0000000000037912
   0.24%  sobel    libjpeg.so.62.2.0    [.] jpeg_fill_bit_buffer

perf annotate

  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
```

```
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():
```

- Conditional move?
- Compiler crazy. All mixed up. In-lined the combine routine.
- 4288*2929=36MB (larger than L3)

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
```

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   [.] 0x0000000000039356
   0.83%  sobel_improved  [kernel.kallsyms]   [k] page_fault
```

HW#3 – SIMD (SSE/AVX)

- SIMD = Single Instruction, multiple data
  One instruction (say add) can add multiple values at once

- On intel chips SSE, SSE2, etc. Up to AVX/AVX2 on newer systems

- 256-bit wide registers. So sixteen 16-bit values (can do integer), Four 64-bit doubles, etc.

- Large number of these registers, xmm0 (128bit) ymm0 (256bit) zmm0 (512bit on newer machines)
- 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

HW#3 – Initial SIMD 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

The 18 ops (9mul/9add) turned into 4 ops
```

Problems

- Math is very fast, handfull of instructions
- Problem is getting memory from 3 pointers with 3-byte offsets into registers
- 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 is actually worse than regular code

- Challenge: if picture not multiple of 16-bytes

HW#3 – 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 R
   D * RGB RGB RGB RGB RGB RGB R
   E *     RGB RGB RGB RGB RGB R
   F *         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 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

Back to OpenMP

Reductions

- Used when a loop is used to combine a large number of results to one variable
- Common example: vector dot product

```
for(i=0;i<N;i++) {
    dot_product=dot_product+(a[i]*b[i]);
}
```

- normally this would be bad in parallel, as race on the dot_product value
- with special reduction command the work is split up in chunks before, but at the end these are automatically combined for the final result

Reduction Example

```
for (int i=0;i<10;++i) {
    a = a op expr
}
```

- expr is a scalar expression that does not read a
- limited set of operations, +,-,*
- variables in list have to be shared

```
#pragma omp parallel for reduction(+:sum) schedule(static,8)
        for(i = 0; i < N; i++) {
                /* Why does this need to be a reduction?*/
                sum = sum + i*a[i];
        }

        printf("sum=%lld\n",sum);
```

OMP Sections – Another way to make code parallel

```
#pragma omp parallel sections

#pragma omp section
// WORK 1
#pragma omp section
// WORK 2
```

- Will run the two sections in parallel at same time.
- Useful if you have multiple chunks of code that's not a loop but still can run at the same time
- You could implement this with for() and a case statement (gcc does it that way?)

Synchronization functions

- Can manually set up locks
- `omp_init_lock()`
- `omp_destroy_lock()`
- `omp_set_lock()`
- `omp_unset_lock()`
- `omp_test_lock()`

OMP Synchronization

- Instead of manually setting locks, can use synchronization directives and OMP will do the hard work for you

OMP Synchronization – Master

```
#pragma omp master
```

- OMP MASTER – only master executes instructions in this block

OMP Synchronization – Critical

```
#pragma omp critical
```

- OMP CRITICAL – only one thread is allowed to execute in this block
- OMP ATOMIC – like critical but for only one instruction, a memory access faster

OMP Synchronization – Barrier

- OMP BARRIER – force all threads to wait until all are done before continuing
- there's an implicit barrier at the end of for, section, and parallel blocks
- It is useful if using nowait in loops

OMP Flush directive

- `#pragma omp flush(a,b)`
- Compiler might cache variables, etc, so this forces a and b to be uptodate across threads

OMP – Calling Functions

- can call functions
- functions outside of directives can still have OpenMP directives in them (orphan directives)

Nested Parallelism

- can have nested for loops, but by default the number of threads comes from the outer loop so an inner parallel for is effectively ignored
- can collapse loops if perfectly nested
- perfectly nested means that all computation happens in inner-most loop
- `omp_set_nested(1);` can enable nesting, but then you end up with OUTER*INNER number of threads
- alternately, just put the #parallel for only on the inner

loop

OpenMP Versions

- 5.0
  - task reduction
  - not-equals can appear in loop comparisons
- 4.0
  - support for accelerators (offload to GPU, etc)
  - SIMD support (specify simd)
  - better error handling
  - CPU affinity
  - task grouping

- user-defined reductions
- sequential consistent atomics
- Fortran 2003

• 3.1

• 3.0

  - tasks
  - lots of other stuff

OpenMP Pros and Cons

- Pros
  - portable
  - simple
  - can gradually add parallelism to code; serial and parallel statements (at least for loops) are more or less the same.
- Cons
  - Race conditions?
  - Runs best on shared-memory systems

◦ Requires recent compiler

OpenMP Examples

See the course website for a link to a tarball with all the examples.

Simple

`openmp_simple.c`

- just creates a parallel region and prints thread number.
- By default, how many threads are set up on the Haswell-EP machine?
- Try with `OMP_NUM_THREADS=4`

Scope

TODO: private/shared variable example

for

`openmp_for.c`

- Parallelizes the memory init loop.
- Thread number set from command line and the `num_threads()` directive.
- What happens to performance as you add threads?

static schedule

`openmp_static_schedule.c`

- Creates 100 threads with chunksize of 1.
- Threads are assigned loop indices at statically at start of loop
- In example, thread 0 is fastest and 4 the slowest.
- You can see thread 0 runs through its assignment fast and then sits around doing nothing while the rest slowly finish.

dynamic schedule

`openmp_dynamic_schedule.c`

- Creates 100 threads with chunksize of 1.
- Threads are assigned loop indices dynamically.
- Each thread starts with one, but zero runs all the rest because it is so fast.

Changing Chunksize

`openmp_dynamic_chunk.c`

- Creates 100 threads with a prime number chunksize.
- Threads are assigned same amount of time to run.
- Spread mostly evenly but the last set of chunks, only two threads get assigned while the others have nothing to do.
- Switch to "guided" and the chunksize decreases over time and the ending is a bit more balanced.

critical

`openmp_critical.c`

- Has a parallel loop, but a shared global counter inside.
- What happens without a critical section? (race condition)
- Put in the critical section get right results.
- But slow!
- No need to manually add mutexes, OpenMP abstracts that away.

section

`openmp_section.c`

- For parallelism when you don't have a loop
- Have multiple functions that have no dependencies, want to run at same time?
- No matter how many threads you have, only can run up to the maximum number of sections at a time.

reduction

`openmp_reduction.c`

- What if you calculate something in each loop iteration, but want to sum them all in the end? Something like a vector dot product?
- You could put it in a for loop, $sum = sum + i * a[i]$ but race condition on shared sum.
- Could put in critical section but that's slow as we saw.
- Instead can use special reduction directive.

simd reduction

`openmp_simd_reduction.c`

https://software.intel.com/en-us/articles/enabling-simd-in-program-using-openmp40

- simd directive
- Supported by recent GCC (5.0 and later)
- Tries to map your code into SSE/AVX vector instructions if available on your processor.
- Our example turns out runs *slower*. Possibly our input set is not big enough.
- Can look at assembly code to verify it is making SIMD

code:

`objdump --disassemble-all openmp_simd_reduction`

- Also you can use `gcc -S` to generate assembly.

  look for pmul and xmm registers

offload

Can offload to GPU or MIC.

`https://gcc.gnu.org/wiki/Offloading`

- Need separate compiler for component.
- Support really isn't there yet(?) verify that

HW#5 Preview

- Will use OpenMP for sobel
- Coarse version – use OMP Sections to run sobelx and sobely at same time
- Fine version – use OMP for directive to do fine grained parallelism

Show off Lovebyte entry

- 1k entry got 1st place in old-school competition
- `https://youtu.be/T6iG3AiOURQ`