ECE 574 – Cluster Computing
Lecture 8

Vince Weaver

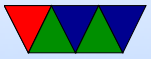
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

23 February 2021

Announcements

- HW#4 was posted

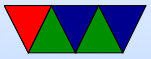


Homework #3 – Copying Files

- For Linux/OSX easiest way to copy files to local machine something like:

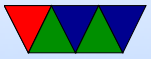
```
scp -P2131 ece574-0@weaver-lab.eece.maine.edu:hw03_submit.tar.gz .
```

- On windows `WinSCP` is widely used
- There are various GUI SCP options for Linux/OSX too but I've never used them so can't particularly recommend any.



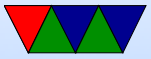
HW#3 – General Comments

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



General C array Comment

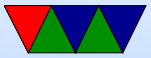
- Why a linear array and not multidimensional?
  C doesn't do dynamically sized multi-dimensional well
- It's all a fiction anyway. You always get a linear array. The multiply/add we are doing is what the compiler does behind the scenes.
- You might think the whole multiply/add stuff would kill performance, but the compiler can often reduces it to just one CPU instruction on x86 (the magical load effective address `lea` instruction



Butterfinger Results

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

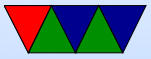
```
time ./sobel ./butterfinger.jpg
output_width=320, output_height=320, output_components=3
SOBELX  L3 CACHE MISSES: 1554   CYCLES 9436089
SOBELY  L3 CACHE MISSES: 0      CYCLES 9362614
COMBINE L3 CACHE MISSES: 3      CYCLES 6574264

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

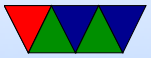
- Haswell-EP has 20MB of L3 cache

- 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



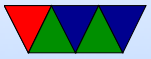
Brief Cache Overview

- 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)

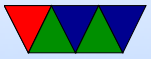
- When access an address, all 64-B brought in even if not need rest
- When cache full, something is kicked out to make room (usually oldest)
- Want to take advantage of spatial and temporal locality
- With butterfinger all fits in L3 cache



Earth, Straight implementation of pseudo-code

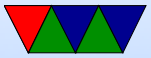
```
./sobel ./earth_06_03_2018.jpg
output_width=2048, output_height=2048, output_components=3
SOBELX L3 CACHE MISSES: 318,572 CYCLES 559,078,407
SOBELY L3 CACHE MISSES: 285,851 CYCLES 556,456,869
COMBINE L3 CACHE MISSES: 593,838 CYCLES 335,950,332

real    0m0.759s user    0m0.688s sys     0m0.032s
```

12MB, fits in cache?

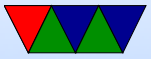


Space Station, Straight implementation of pseudo-code

```
./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
```

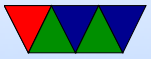


```
perf report

  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
```

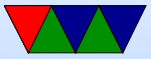


```
  0.42 |       imul   0x8(%r12),%eax
  0.06 |       mov    0x10(%r12),%rsi
  0.69 |       add    %ecx,%eax
  0.18 |       test   %ebx,%ebx
  7.08 |       cmovs  %edi,%ebx
  1.82 |       cltq
```

```
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():
```



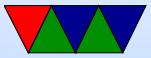
- Conditional move?
- Compiler crazy. All mixed up. In-lined the combine routine.
- 4288*2929=36MB (larger than L3)



Loop Order Optimization

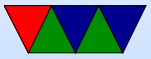
- How is an array laid out in memory?
  Row-major (C) vs Column-major (Fortran)
- Default with loop x then y, are actually walking columns.
  Worst case.
- Switch order of loops, things get a lot better.

```
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
```



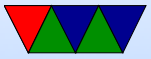
Loop Unrolling

- 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.



Other Optimizations

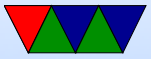
- Other optimizations, often are things the compiler does for you with -O2.
- Hoisting (move things out of loop that only need to be done once)
- Simplification. Lots of things.
- Try another compiler (clang?)
- Take a compiler class.



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).

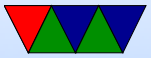
```
time ./sobel_improved ./IMG_1733.JPG
output_width=3888, output_height=2592, output_components=3
SOBELX  L3 CACHE MISSES: 15,703 CYCLES 411,148,087
SOBELY  L3 CACHE MISSES: 15,334 CYCLES 411,284,853
COMBINE L3 CACHE MISSES: 1,245,842      CYCLES 1,186,204,125

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



Same for Combine

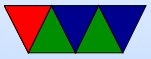
No need to offset, just start at beginning of x and y and write to output, doing the combine operation.

```
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
```

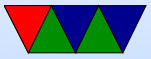


SIMD (SSE/AVX)

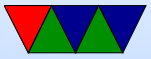
- SIMD = Single Instruction, multiple data
  One instruction (say add) can add multiple values at once

- 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.

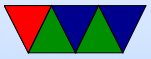


- Large number of these registers, xmm0 (128bit) ymm0 (256bit) zmm0 (512bit on newer machines)
- One way is to program in assembly language with some obscure opcodes: an example `PMADDWD` 16-bit integer parallel 128-bit multiply and add
- On recent gcc and other compilers there are "intrinsics" to use in C, for example you can use `_mm_madd_epi16()` to do a `PMADDWD` instruction



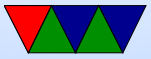
Initial SIMD try

```
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
```

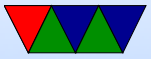


Problems

- 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



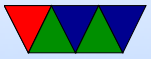
- Challenge: if picture not multiple of 16-bytes



Improved SIMD – Can we do better?

With many problems: re-think outside the serial box

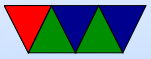
```
Load full 16 bytes of pixel info from the three pointers,
multiply by the 9 values in sobel filter, shifting right by 3
   A * RGB RGB RGB RGB RGB RGB R
   B *     RGB RGB RGB RGB RGB R
   C *         RGB RGB RGB RGB R
   D * RGB RGB RGB RGB RGB RGB R
   E *     RGB RGB RGB RGB RGB R
   F *         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
```



OpenMP

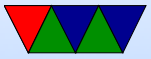
A few good references:

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



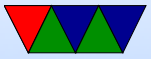
OpenMP

- 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



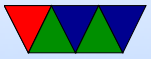
OpenMP Background

- Shared memory multi-processing interface
- C, C++ and FORTRAN
- Industry standard made by lots of companies
- OpenMP 1.0 came out in 1997 (FORTRAN) or 1998 (C), now version 5.1 (2020)
- gcc support “recently” 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)



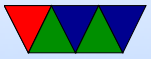
OpenMP

- Master thread with Fork/Join methodology
- Can possibly have nested threads (implementation dependent).
- Can possibly have dynamic num of threads (implementation dependent)
- Relaxed consistency, threads can cache local variables, so if you need memory to be consistent might need to flush it.



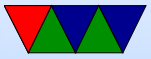
OpenMP Interface

- Compiler Directives
- Runtime Library Routines
- Environment Variables



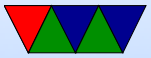
Compiler Support

- On gcc, pass `-fopenmp`
- C: `#pragma omp`
- FORTRAN: `C$OMP` or `!$OMP`



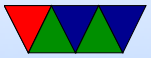
Compiler Directives

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



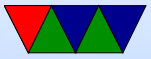
Library routines

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



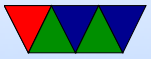
Environment Variables

- Setting number of threads
- Configuring loop iteration division
- Processor bindings
- Nested Parallelism settings
- Dynamic thread settings
- Stack size
- Wait policy

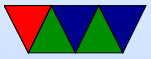


Simple Example

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>

int main (int argc, char **argv) {

        int nthreads,tid;

/* Fork a parallel region, each thread having private copy of tid */
#pragma omp parallel private(tid)
        {
        tid=omp_get_thread_num();
        printf("\tInside of thread %d\n",tid);

        if (tid==0) {
                nthreads=omp_get_num_threads();
                printf("This is the master thread, there are %d threads\n",
                        nthreads);
        }
```

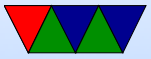


```
        }

        /* End of block, waits and joins automatically */

        return 0;
}
```



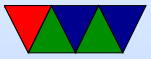
Notes on the example

- PARALLEL directive creates a set of threads and leaves the original thread the master, with tid 0.
- All threads will execute the code in parallel region
- There's an implied barrier/join at end of parallel region. Only the master continues after it.
- If any thread terminates in a parallel region, then all threads will also terminate.
- You can't `goto` into a parallel region.
- In C++ special rules on throw/catching



parallel directive

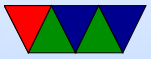
```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)

   structured_block
```



if

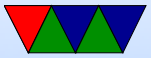
- if – you can do a check `if (i==0)`
  If true parallel threads are created, otherwise serially

- Why do this? Maybe it's only worth parallelizing if N greater than 16 due to overhead, can put `if (N>16)` then



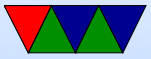
Variable Scope

- When you enter a parallel section, which variables are thread-local and which ones are globally visible?
- By default shared, but there are times you want per-thread data and not globally visible (loop indices for one)
- You specify in the parallel block how you want all of the variables to behave
  - private – variables that are private. The value is undefined at start and discarded at end

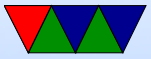


- shared – variables seen by all threads, all can be written to. Value at end is whatever the last thread wrote to it
- firstprivate – a variable inside a parallel section ends up with the value it had before the parallel section
- lastprivate – a variable after the parallel section gets the value from the last loop iteration
- copyin – you can declare special "Threadprivate" values that hold their value across parallel sections. Use this to copy the value in from the master thread.

• default – you can set to shared or none (more on C),

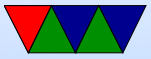


none means you have to explicitly share or private each var (makes it easier to catch bugs but more tedious)



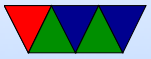
Setting number of threads

- Evaluation of the IF clause
- Setting of the NUM_THREADS clause
- Use of the `omp_set_num_threads()` library function
- Setting of the `OMP_NUM_THREADS` environment variable
- Implementation default – usually the number of CPUs on a node, though it could be dynamic (see next bullet).
- Threads are numbered from 0 (master thread) to N-1



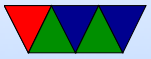
How do you actually share work?

Could do work with this, split things up manually by having a lock/critical section and divide up work per-thread. But easier way?



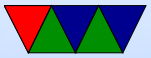
Work-sharing Constructs

- Must be inside of a parallel directive
  - do/for (do is Fortran, for is C)
  - sections
  - single – only executed by one thread
  - workshare – iterates over F90 array (Fortran90 only)



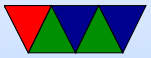
For Example

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>

static char *memory;

int main (int argc, char **argv) {

    int num_threads=1;
    int mem_size=256*1024*1024; /* 256 MB */
    int i,tid,nthreads;

    /* Set number of threads from the command line */
    if (argc>1) {
        num_threads=atoi(argv[1]);
    }

    /* allocate memory */
    memory=malloc(mem_size);
    if (memory==NULL) perror("allocating memory");
```



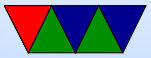
```
#pragma omp parallel shared(mem_size,memory) private(i,tid)
    {

    tid=omp_get_thread_num();
    if (tid==0) {
        nthreads=omp_get_num_threads();
        printf("Initializing %d MB of memory using %d threads\n",
            mem_size/(1024*1024),nthreads);
    }


    #pragma omp for schedule(static) nowait
    for (i=0; i < mem_size; i++)
        memory[i]=0xa5;
    }

    printf("Master thread exiting\n");

}
```

Note: loop must be simple. Integer expressions (nothing



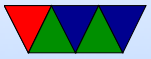
super fancy). Comparison must be only regular equals or greater/less. Iterator must be simple increment/decrement or add/subtract.

Loop iterator should be private. Why? What happens if all threads could update a global iterator?



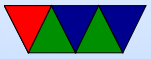
Do/For

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

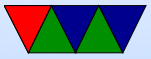


Scheduling

- By default, splits to N size/p threads chunks statically.
- schedule (static,n) chunksize n
  for example, if 10, and 100 size problem, 0-9 CPU 1, 10-19 CPU 2, 20-29 CPU3, 30-39 CPU4, 40-49 CPU1.
- But what if some finish faster than others?
- dynamic allocates chunks as threads become free. Can have much higher overhead though.
  - static – divided into size chunk, statically assigned to threads



- dynamic – divided into chunks, dynamically assigned threads as they finish
- guided – like dynamic but shrinking blocksize
  why do this? When problem first starts lots of big chunks left. But near end probably not even, could end up with one thread getting large chunk and rest none. Better load balancing.
- runtime – from OMP_SCHEDULE environment variable
- auto – compiler picks for you



Other Options

- nowait – threads do not wait at end of loop
- ordered – loops must execute in order they would in serial code
- collapse – nested loops can be collapsed
  if "perfectly nested" meaning nested with nothing inside the nests. Compiler can turn this into one big loop