ECE 574 – Cluster Computing
Lecture 6

Vince Weaver

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

vincent.weaver@maine.edu

7 February 2019

Announcements

- HW#3 will be posted
- I found my code with SSE intrinsics, here is a brief excerpt of what it looks like to allocate a 256-bit variable and load a vector to it.

```
__m256i in1;

    /* vmovdqa (%rcx),%ymm1 */
    __m256i filter_avx = _mm256_load_si256( (__m256i *)filter);
```

Workload for future Homeworks

- Matrix multiply is typical, but boring
- What else can we use that's embarrassingly parallel, but interesting?

Convolution

- https://en.wikipedia.org/wiki/Kernel_%28image_processing%29
- Specifically 2-D convolution
- Widely used in image processing
- Walk over every pixel in an image, convolving a matrix over it. The new value is based on some combination of the surrounding pixels.
- Usually a 3x3 grid, but can be larger

Common Convolution Matrices

- Identity $= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
- Blur $= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ (need to normalize)
- Sharpen $= \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
- Emboss $= \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$
- Sobel (edge detection) $= \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$

What does a framebuffer look like

- Depends on many things
- Bits-per-pixel, 1bpp, 2bpp, 4bpp, 8bpp, 15bpp, 16bpp, 24bpp, 32bpp
- We will be using 24bpp, with RGB each being one byte
- 2D image is a 2D array, but that's hard to do in C, so we will just do a 1D array

One way to implement the convolution

There are many ways you can implement this, some will be faster than others. The one shown below is definitely not the fastest.

Below is *pseudo code*. It won't compile, as you won't be able to do the triple array access as pictured, you'll have to access the values as a 1-D array as discussed in class.

```c
for(x=1;x<width-1;x++) {
    for(y=1;y<height-1;y++) {
        for(color=0;color<3;color++) {

            sum=0;

            sum+=filter[0][0]*old[x-1][y-1][color];
            sum+=filter[1][0]*old[x][y-1][color];
            sum+=filter[2][0]*old[x+1][y-1][color];
            sum+=filter[0][1]*old[x-1][y][color];
            sum+=filter[1][1]*old[x][y][color];
            sum+=filter[2][1]*old[x+1][y][color];
            sum+=filter[0][2]*old[x-1][y+1][color];
            sum+=filter[1][2]*old[x][y+1][color];
            sum+=filter[2][2]*old[x+1][y+1][color];

            /* Normalize if necessary */
            /* (not necessary for Sobel) */

            /* Saturate if necessary */
            /* Make sure stays in 0 to 255 range */
            (your code here)
```

```
            /* Set the new value */
            new[x][y][color]=sum;
        }
    }
}
```

Hints:

- `a[x][y][color]` should be done as `a[(y*xsize*3)+(x*3)+color]` You might want to write a helper function that does this for you.
- Remember in C that array indexes begin at 0, not 1.

Sobel Convolution

- For Sobel we do not need to normalize the result, but we do need to saturate
  Meaning if the results is greater than 255, set to 255, or if less than zero, set to zero. Otherwise will wrap and give odd results.
- In the homework we will find the horizontal edge, the vertical edge, and then combine the two for the final result by for each element squaring the two results then taking the square root.

PAPI Usage Instructions

- Initialize with:
  `PAPI_library_init(PAPI_VER_CURRENT);`
  Check the result to see if it matches `PAPI_VER_CURRENT`

- All other functions should return `PAPI_OK` if successful.

- If using pthreads need to do:
  `PAPI_thread_init(pthread_self);`

- Eventsets are just integers
  `int eventset=PAPI_NULL;`

- Gathered results are typically 64-bit integers
  `long long values[NUM];`
  Where NUM is the number of events you are measuring at once.

- Create an eventset:
  `PAPI_create_eventset(&eventset);`

- Available events can be seen with the `papi_avail` and `papi_native_avail` commands.

- Add an event. You can run multiple times to add multiple events.

```
PAPI_add_named_event(eventset,"PAPI_TOT_INS");
```

- Before the code of interest do a
  ```
  PAPI_start(eventset);
  ```

- Afterward do a
  ```
  PAPI_stop(eventset,values);
  ```
  and you can print the value or save it for later.

- When printing, remember the results are 64 bits.
  ```
  printf("Result:  %lld",values[0]);
  ```

How to Optimize

- ROW vs Column Major? FORTRAN vs C? Comes down to using cache in an expected way.
- Loop order? Again, want to access in a way that keeps things in cache
- Loop unrolling? Avoids branch issues, etc.
- SIMD? Definitely a case where we could load all 4 channels and operate on them at once. Possibly multiple. A bit advanced for this class though.

Types of Clusters

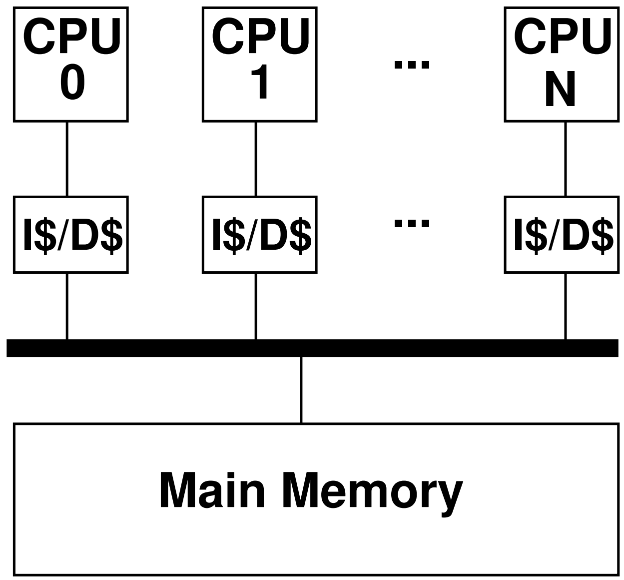
- Shared-memory: many CPUs, but one shared memory address space. Usually one copy of operating system. When write to memory, all CPUs can see it.

- Distributed: man systems spread across network. Each has own memory. For other CPUs to see data have to send message across network.

Multicore Systems

- Single Package: CMP (Chip-multiprocessor) or SMP (Symmetric-multiprocessor)
- Multi-package: Multiple CMP packages in system.

CMP Diagram



Hardware Multi-Threading

- Idea is to re-use a pipeline to execute multiple threads at once, *without* fully replicating the entire CPU (so less than multicore)
- You will have to replicate some things (program counter for each, etc)
- Usually they appear to the CPU as full separate processors even though they are not.
- Various ways to do this:

- ◦ Fine-grained – rotate threads every cycle
- ◦ Coarse-grained – rotate threads only if long latency event happens (cache miss)
- ◦ Simultaneous – issue from any combination of threads, to maximize use of pipeline (have to be superscalar)

- Why do this? Often on superscalar running only one thread will leave parts idle, try to make use of these.

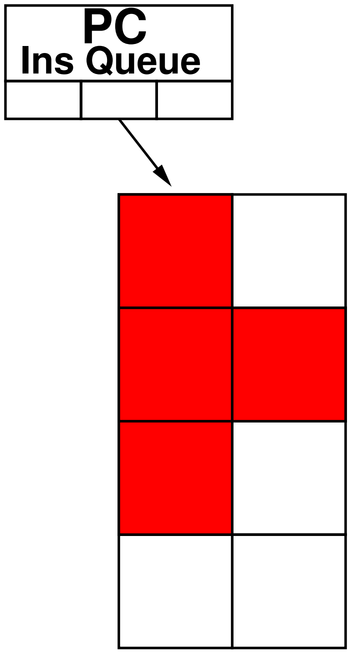
- Bad side effects?
  Can actually slow down code (especially if both threads

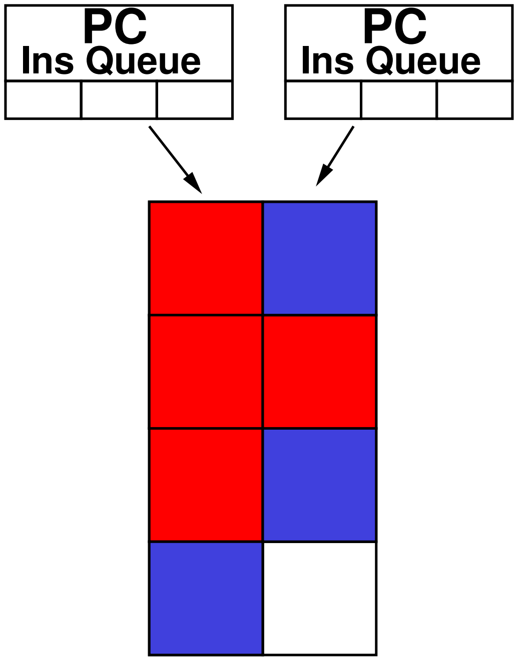
trying to use same functional units, also if both using memory heavily as cache is often shared)

- Sometimes see it talked about as SMT (Simultaneous Multithreading), Intel Hyperthreading is more or less the same thing

- Modern security issues, leak info between threads

SMT Diagram





Cache Coherency

- How do you handle data being worked on by multiple processors, each with own cache of main memory?
- Cache coherency protocols.
- Many and varied. MESI is a common one
- Directory vs Snoopy

MESI

- Modified, Exclusive, Shared, Invalid

Barriers and Ordering

- On modern out-of-order execution, memory accesses can happen out-of-order

- Sequential consistency – all happen in order

- Strong consistency – stores

- Weak consistency – can be arbitrarily reordered, only barriers protect you

- A memory barrier instruction makes sure all previous loads/stores finish before moving on

- Most important for things like locks, as well as memory-mapped I/O

Ordering Example

```
    y1=0
    y2=0
    y1=3
    y2=4
Another core
    x1=y1
    x2=y2
```

What values of x1 and x2 can you get?

Strong:

x1=0,x2=0

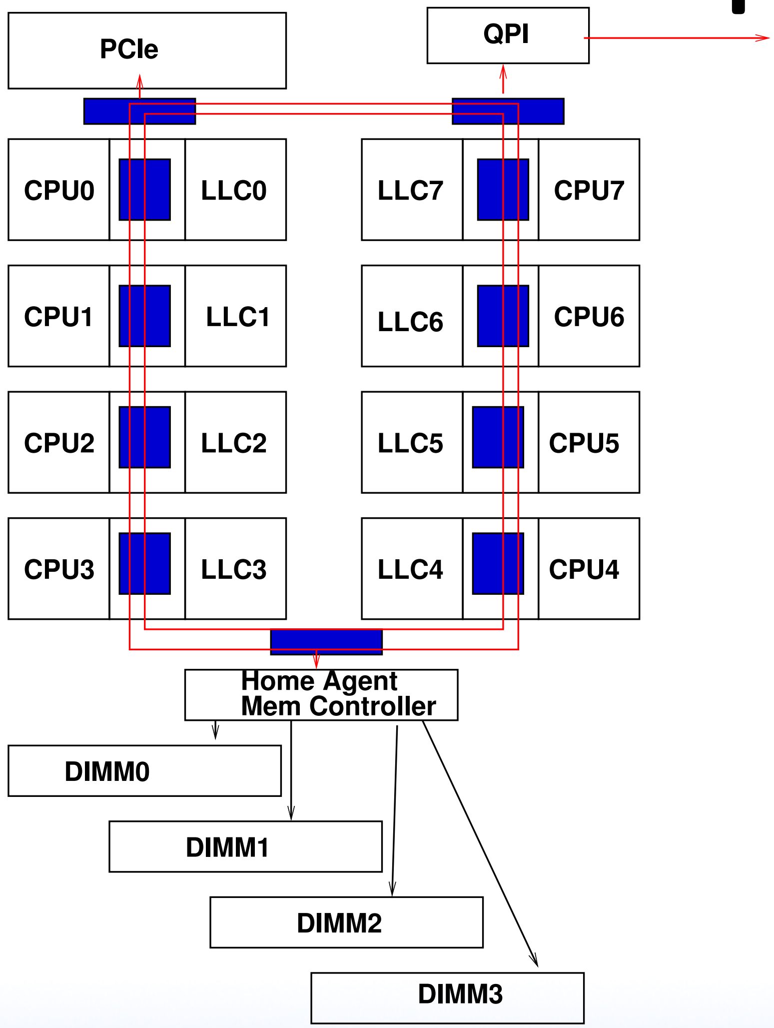
x1=3,x2=0

x1=3,x2=4

Weak:

$x1=0,x2=4$

Haswell EP Setup



NUMA

Non-uniform memory access – some accesses will have to cross to other processors, causing extra delay. How can you optimize this?

Traditional NUMA Layout