ECE 574 – Cluster Computing
Lecture 5

Vince Weaver

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

vincent.weaver@maine.edu

2 February 2017

Announcements

- HW#4 will be posted Friday
- Supercomputer used to win at Poker

  `https://www.theregister.co.uk/2017/02/01/machines_can_now_conquer_poker/`

Workload for future Homeworks

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

Convolution

- 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) $= \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$

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.

```
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.

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]);`

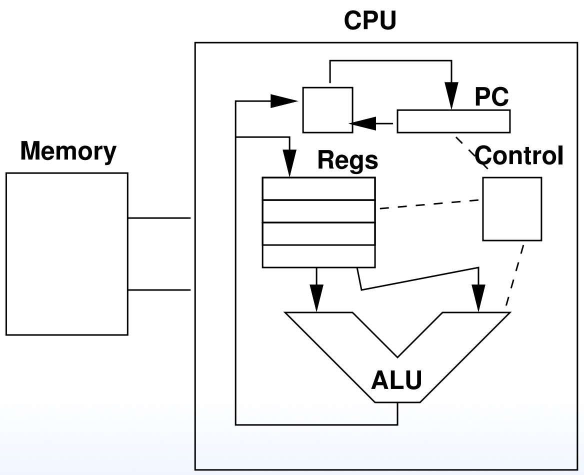
How to Optimize

- 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.

Parallel Computing – Single Core

Simple CPUs

- Ran one instruction at a time.
- Could take one or multiple cycles (Instructions per Cycle (IPC) 1.0 or less)
- Example – single instruction take 1-5 cycles?



Pipelined CPUs

- 5-stage MIPS pipeline
- From 2-stage to Pentium 4 31-stage
- Example – single instruction always take 5 cycles? But what about on average? (Theoretical max IPC 1.0)

IF	**ID**	**EX**	**MEM**	**WB**

Pipelined CPUs

- IF = Instruction Fetch.
  Fetch 32-bit instruction from L1-cache
- ID = Decode
- EX = execute (ALU, maybe shifter, multiplier, divide)
  Memory address calculated
- MEM = Memory – if memory had to be accessed, happens now.
- WB = register values written back to the register file

Data Hazards

Happen because instructions might depend on results from instructions ahead of them in the pipeline that haven't been written back yet.

- RAW – "true" dependency – problem. Bypassing?
- WAR – "anti" dependency – not a problem if commit in order
- WAW – "output" dependency – not a problem as long as ordered
- RAR – not a problem

Structural Hazards

- CPU can't just provide. Not enough multipliers for example

Control Hazards

- How quickly can we know outcome of a branch
- Branch prediction? Branch delay slot?

Branch Prediction

- Predict (guess) if a branch is taken or not.
- What do we do if guess wrong? (have to have some way to cancel and start over)
- Modern predictors can be very good, greater than 99%
- Designs are complex and could fill an entire class

Memory Delay

- Memory/cache is slow
- Need to bubble / Memory Delay Slot

The Memory Wall

- Wulf and McKee
- Processors getting faster more quickly than memory
- Processors can spend large amounts of time waiting for memory to be available
- How do we hide this?

Caches

- Basic idea is that you have small, faster memories that are closer to the CPU and much faster
- Data from main memory is cached in these caches
- Data is automatically brought in as needed.
  Also can be pre-fetched, either explicitly by program or by the hardware guessing.
- What are the downsides of pre-fetching?
- Modern systems often have multiple levels of cache. Usually a small (32k or so each) L1 instruction and data,

a larger (128k?) shared L2, then L3 and even L4.

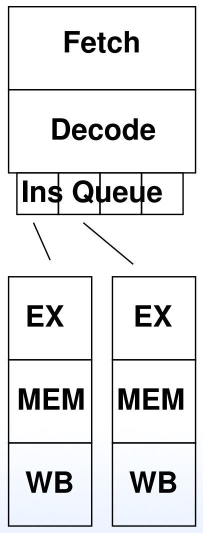
- Modern systems also might share caches between processors, more on that later
- Again, could teach a whole class on caches

Exploiting Parallelism

- How can we take advantage of parallelism in the control stream?
- Can we execute more than one instruction at a time?

Multi-Issue (Super-Scalar)

- Decode up to X instructions at a time, and if no dependencies issue at same time.
- Dual issue example. Can have theoretical IPC of 2.0
- Can have unequal pipelines.



Out-of-Order

- Tries to exploit instruction-level parallelism
- Instead of being stuck waiting for a resource to become available for an instruction (cache, multiplier, etc) keep executing instructions beyond as long as there are no dependencies
- Need to insure that instructions commit in order
- What happens on exception? (interrupt, branch mispredict, etc)

- Register Renaming
- Re-order buffer
- Speculative execution / Branch Prediction?

SIMD / Vector Instructions

- SISD – single instruction, single data, your normal serial processor
- SIMD – single instruction, multiple data – one instruction can act on many values in parallel
- MISD – multiple instruction, single data – wavefront or pipeline? some debate about if this really exists
- MIMD – sort of like a cluster

SIMD / Vector Instructions

- x86: MMX/SSE/SSE2/AVX/AVX2
  semi-related FMA
- MMX (mostly deprecated), AMD's 3DNow! (deprecated)
- PowerPC Altivec
- ARM: Neon

SSE / x86

- SSE (streaming SIMD): 128-bit registers XMM0 - XMM7, can be used as 4 32-bit floats
- SSE2 : 2*64bit int or float, 4 * 32-bit int or float, 8x16 bit int, 16x8-bit int
- SSE3 : minor update, add dsp and others
- SSSE3 (the s is for supplemental): shuffle, horizontal add
- SSE4 : popcnt, dot product

AVX / x86

- AVX (advanced vector extensions) – now 256 bits, YMM0-YMM15 low bits are the XMM registers. Now twice as many.
  Also adds three operand instructions a=b+c
- AVX2 – 3 operand Fused-Multiply Add, more 256 instructions
- AVX-512 – version used on Xeon Phis (knights landing) and Skylake – now 512 bits, ZMM0-XMM31

SSE example (From Wikipedia)

Doing a 4 element single-prevision 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:

```
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
```

ARM NEON

- Cortex A8, optional on Cortex A9
- 64 or 128bit, but some procs break 128-bit into two operations
- 8, 16, 32-bit ints, single-precision floating point

SIMD Benefits

- Can be faster (2, 4, 8, 16, etc. things at once)

SIMD Drawbacks

- Harder to code (assembly or clever compiler)
- Puts more pressure on memory.
- More registers to save at context switch