ECE 571 – Advanced Microprocessor-Based Design Lecture 20

Vince Weaver

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

vincent.weaver@maine.edu

12 April 2016

Project/HW Reminder

- Homework #9 was posted

Raspberry Pi Cache Hierarchy Digression

Raspberry Pi A+/B BCM2835 700MHz LPDDR2 RAM 1MBx16 memset()

Hardware	Software	cycles	time	MB/s
No Cache	C 1-byte	936754552	1.338s	12.0 MB/s
L1-I$	C 1-byte	355098645	0.507s	31.5 MB/s
L1-I$+brpred	C 1-byte	271038891	0.387s	41.3 MB/s
L1-I$+brpred+D$	C 1-byte	116346597	0.166s	96.3 MB/s
No Cache	C 4-byte	205749402	0.294s	54.4 MB/s
L1-I$	C 4-byte	67745267	0.097s	165 MB/s
L1-I$+brpred	C 4-byte	63533353	0.091s	176 MB/s
L1-I$+brpred+D$	C 4-byte	28633484	0.041s	391 MB/s
No Cache	ASM 64B	23437080	0.0335s	478 MB/s
L1-I$	ASM 64B	17749501	0.0253s	631 MB/s
L1-I$+brpred	ASM 64B	18006681	0.0257s	622 MB/s
L1-I$+brpred+D$	ASM 64B	8829849	0.0126s	1268 MB/s

Theoretical Maximum speed of LPDDR2@400MHZ = 8GB/s
Linux glibc memset() maxes out around 1400 MB/s

Interesting issue

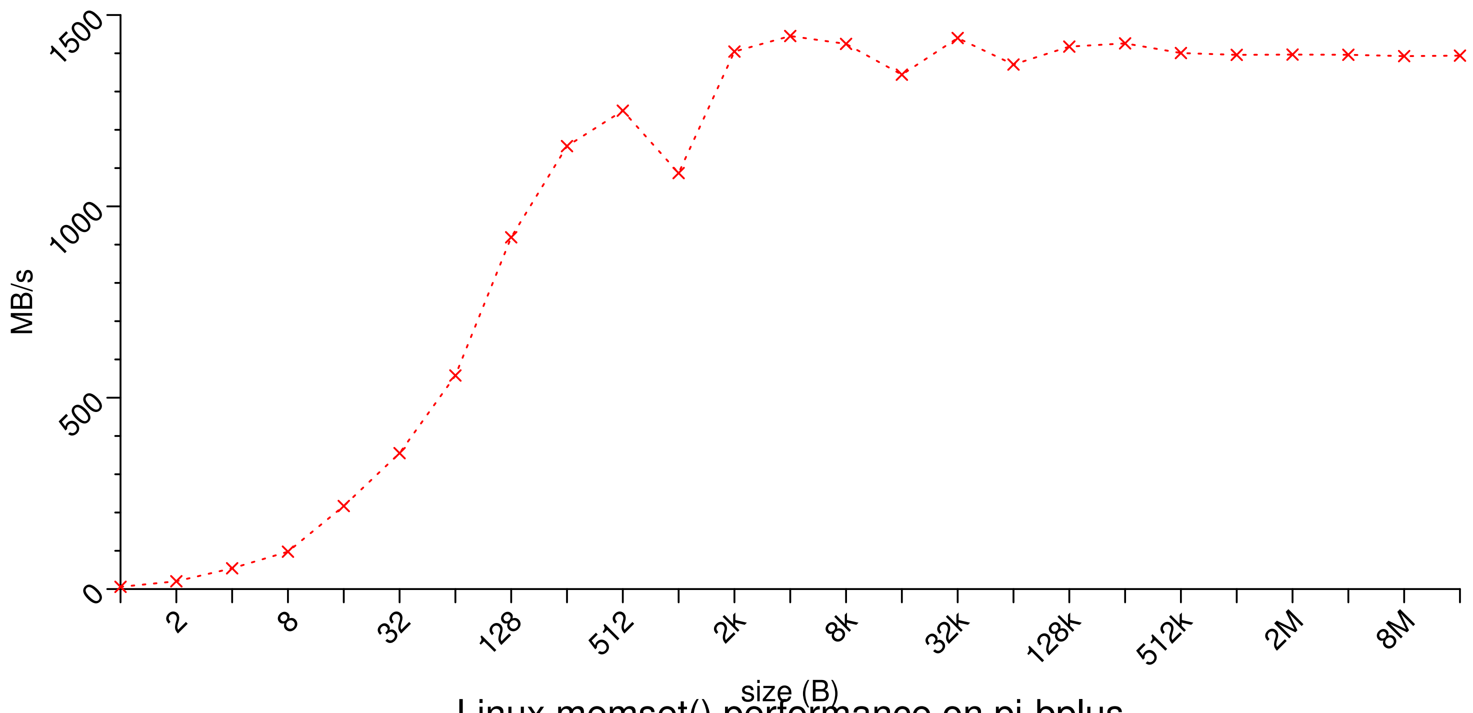
- Sometimes the byte-by-byte gets 7MB/s, sometimes 11MB/s. Not cache (turned off). Not branch predictor (turned off).
- Is based on memory offset, if you add printk to debug, can start or stop
- Careful poking around and adding nops revealed if the inner store loop crosses a 64-byte boundary (i.e. branched from 0x44 to 0x3c) performance dropped off

by 40%.

- Why the cause? No icache so the loop must be crossing some 64-byte barrier

- In LPDDR2 a 64-byte (512 bit) rowsize is common. So maybe we are directly seeing the impact of not having code stay in the same open row.

Linux memset performance



Linux memset() performance on pi-bplus

Graphics and Video Cards

Old CRT Days

- Electron gun
- Horizontal Blank, Vertical Blank
- Atari 2600 – only enough RAM to do one scanline at a time
- Apple II – video on alternate cycles, refresh RAM for free
- Bandwidth key issue. SNES / NES, tiles. Double

buffering vs only updating during refresh

Old 2D Video Cards

- Framebuffer (possibly multi-plane), Palette
- Dual-ported RAM, RAMDAC (Digital-Analog Converter)
- Interface (on PC) various io ports and a 64kB RAM window
- Mode 13h
- Acceleration – often commands for drawing lines, rectangles, blitting sprites, mouse cursors, video overlay

Modern Graphics Cards

- Can draw a lot of power
- 2D (optional these days)
- 3D
- Video decoders

Interface

- Integrated or stand alone
- Integrated traditionally less capable, but changing. Share Memory bandwidth, take memory.

GPUs

- Display memory often broken up into tiles (improves cache locality)
- Massively parallel matrix-processing CPUs that write to the frame buffer (or can be used for calculation)
- Texture control, 3d state, vectors
- Front-buffer (written out), Back Buffer (being rendered) Z-buffer (depth)

- Originally just did lighting and triangle calculations. Now shader languages and fully generic processing

Video RAM

- VRAM – dual ported. Could read out full 1024Bit line and latch for drawing, previously most would be discarded (cache line read)
- GDDR3/4/5 – traditional one-port RAM. More overhead, but things are fast enough these days it is worth it.
- Confusing naming, GDDR3 is equivalent of DDR2 but with some speed optimization and lower voltage (so

higher frequency)

Busses

- DDC – i2c bus connection to monitor, giving screen size, timing info, etc.
- PCIe (PCI-Express) – most common bus used in x86 systems
  Original PCI and PCI-X was 32/64-bit parallel bus
  PCIe is a serial bus, sends packets
  Can power 25W, additional power connectors to supply can have 75W, 150@ and more
  Can transfer 8GT/s (giga-transfers) a second

In general PCIe is the main limiting factor to getting data to GPU.

Connectors

CRTC (CRT Controller) Can point to same part of memory (mirror) or different.

- RCA – composite/analog TV
- VGA – 15 pin, analog
- DVI – digital and/or analog. DVI-D, DVD-I, DVD-A
- HDMI – compatible with DVI (though content restrictions). Also audio. HDMI 1.0 – 165MHz, 1080p

or 1920x1200 at 60Hz. TMDS differential signalling. Packets. Audio sent during blanking.

- Display Port – similar but not the same as HDMI
- Thunderbolt – combines PCIe and DisplayPort. Intel/Apple. Originally optical, but also Copper. Can send 10W of power.
- LVDS – Low Voltage Differential Signaling – used to connect laptop LCD

LCD Displays

- Crystals twist in presence of electric field
- Asymmetric on/off times
- Passive (crossing wires) vs Active (Transistor at each pixel)
- Passive have to be refreshed constantly
- Use only 10% of power of equivalent CRT

- Circuitry inside to scale image and other post-processing
- Need to be refreshed periodically to keep their image
- New “bistable” display under development, requires not power to hold state

Interfaces

- OpenGL – SGI
- DirectX – Microsoft
- For consumer grade, driven by gaming

GPGPUS

- Interfaces needed, as GPU companies do not like to reveal what their chips due at the assembly level.
  - CUDA (Nvidia)
  - OpenCL (Everyone else) – can in theory take parallel code and map to CPU, GPU, FPGA, DSP, etc

Why GPUs?

- Old example:
  - 3GHz Pentium 4, 6 GFLOPS, 6GB/sec peak
  - GeForceFX 6800: 53GFLOPS, 34GB/sec peak
- Newer example
  - Raspberry Pi, 700MHz, 0.177 GFLOPS
  - On-board GPU: Video Core IV: 24 GFLOPS

Key Idea

- using many slimmed down cores
- have single instruction stream operate across many cores (SIMD)
- avoid latency (slow textures, etc) by working on another group when one stalls

GPU Benefits

- Specialized hardware, concentrating on arithmetic. Transistors for ALUs not cache.
- Fast 32-bit floating point (16-bit?)
- Driven by commodity gaming, so much faster than would be if only HPC people using them.
- Accuracy? 64-bit floating point? 32-bit floating point? 16-bit floating point? Doesn't matter as much if color slightly off for a frame in your video game.

- highly parallel

GPU Problems

- optimized for 3d-graphics, not always ideal for other things
- Need to port code, usually can't just recompile cpu code.
- Companies secretive.
- serial code
- a lot of control flow

- lot of off-chip memory transfers

GPU Performance

- Like stream processors, need parallel. Only can operate on independent things, but can do many many at once. Stream processors are records that all need similar operations done to them. Kernels are the code applied in each processor. Vertices and fragments have shaders run on them.

Traditional GPU Setup

- CPU send list of vertices to GPU.
- Transform (vertex processor) (convert from world space to image space). 3d translation to 2d, calculate lighting. Operate on 4-wide vectors (x,y,z,w in projected space, r,g,b,a color space)
- Rasterizer – transform vertexes/vectors into a grid. Fragments. break up to pixels and anti-alias
- Shade (Fragment processor) compute color for each

pixel. Use textures if necessary (texture memory, mostly read)

- Write out to framebuffer (mostly write)

GPGPU Key Ideas

- Using many slimmed down cores
- Have single instruction stream operate across many cores (SIMD)
- A void latency (slow textures, etc) by working on another group when one stalls

GPU Benefits

- Specialized hardware, concentrating on arithmetic. Transistors for ALUs not cache.
- Fast 32-bit floating point (16-bit?)
- Driven by commodity gaming, so much faster than would be if only HPC people using them.
- Accuracy? 64-bit floating point? 32-bit floating point? 16-bit floating point? Doesn't matter as much if color slightly off for a frame in your video game.

- highly parallel

GPU Problems

- Optimized for 3d-graphics, not always ideal for other things
- Need to port code, usually can't just recompile cpu code.
- Companies secretive.
- Serial code with a lot of control flow runs poorly
- Off-chip memory transfers can be slow

Latency vs Throughput

- CPUs = Low latency, low throughput
- GPUs = high latency, high throughput
- CPUs optimized to try to get lowest latency (caches); with no parallelism have to get memory back as soon as possible
- GPUs optimized for throughput. Best throughput for all better than low-latency for one

Older / Traditional GPU Pipeline

- In old days, fixed pipeline.
- Modern chips much more flexible, but the old pipeline can still be implemented in software via the fancier interface.

Older / Traditional GPU Pipeline

- CPU send list of vertices to GPU.
- Transform (vertex processor) (convert from world space to image space). 3d translation to 2d, calculate lighting. Operate on 4-wide vectors (x,y,z,w in projected space, r,g,b,a color space)
- Rasterizer – transform vertexes/vectors into a grid. Fragments. break up to pixels and anti-alias

- Shader (Fragment processor) compute color for each pixel. Use textures if necessary (texture memory, mostly read)

- Write out to framebuffer (mostly write)

GPGPUs

- Started when the vertex and fragment processors became generically programmable (originally to allow more advanced shading and lighting calculations)

- By having generic use can adapt to different workloads, some having more vertex operations and some more fragment

Graphics vs Programmable Use

Vertex	Vertex Processing	Data	MIMD processing
Polygon	Polygon Setup	Lists	SIMD Rasterization
Fragment	Per-pixel math	Data	Programmable SIMD
Texture	Data fetch, Blending	Data	Data Fetch
Image	Z-buffer, anti-alias	Data	Predicated Write

Example for Shader 3.0, came out DirectX9

They are up to Pixel Shader 5.0 now

Shader 3.0 Programming – Vertex Processor

- 512 static / 65536 dynamic instructions
- Up to 32 temporary registers
- Simple flow control
- Texturing – texture data can be fetched during vertex operations

- Can do a four-wide SIMD MAD (multiply ADD) and a scalar op per cycle:
  - EXP, EXPP, LIT, LOGP (exponential)
  - RCP, RSQ (reciprocal, r-square-root)
  - SIN, COS (trig)

Shader 3.0 Programming – Fragment Processor

- 65536 static / 65536 dynamic instructions (but can time out if takes too long)
- Supports conditional branches and loops
- fp32 and fp16 internal precision
- Can do 4-wide MAD and 4-wide DP4 (dot product)

GPGPUs

- Interfaces needed, as GPU companies do not like to reveal what their chips due at the assembly level.
  - CUDA (Nvidia)
  - OpenCL (Everyone else) – can in theory take parallel code and map to CPU, GPU, FPGA, DSP, etc
  - OpenACC?

Program

- Typically textures read-only. Some can render to texture, only way GPU can share RAM w/o going through CPU. In general data not written back until entire chunk is done. Fragment processor can read memory as often as it wants, but not write back until done.
- Only handle fixed-point or floating point values
- Analogies:
  - Textures $==$ arrays

- Kernels $==$ inner loops
- Render-to-texture $==$ feedback
- Geometry-rasterization $==$ computation. Usually done as a simple grid (quadrilateral)
- Texture-coordinates $=$ Domain
- Vertex-coordinates $=$ Range

Flow Control, Branches

- only recently added to GPUs, but at a performance penalty.
- Often a lot like ARM conditional execution

Terminology (CUDA)

- Thread: chunk of code running on GPU.
- Warp: group of thread running at same time in parallel simultaneously
- Block: group of threads that need to run
- Grid: a group of thread blocks that need to finish before next can be started

Terminology (cores)

- Confusing. Nvidia would say GTX285 had 240 stream processors; what they mean is 30 cores, 8 SIMD units per core.

CUDA Programming

- Since 2007
- Use `nvcc` to compile
- *host* vs *device*
  host code runs on CPU
  device code runs on GPU
- Host code compiled by host compiler (gcc), device code by custom NVidia compiler

- `__global__` parameters to function – means pass to CUDA compiler
- cudaMalloc() to allocate memory and pointers that can be passed in
- call global function like this `add<<<1,1>>>(args)` where first inside brackets is number of blocks, second is threads per block
- cudaFree() at the end
- Can get block number with blockIdx.x and thread index

with threadIdx.x

- Can have 65536 blocks and 512 threads (At least in 2010)

- Why threads vs blocks?
  Shared memory, block specific
  `__shared__` to specify

- `__syncthreads()` is a barrier to make sure all threads finish before continuing

Code Example

```
#include <stdio.h>

#define N 10


 __global__ void add (int *a, int *b, int *c
        int tid=blockIdx.x;

        if (tid<N) {
                c[tid]=a[tid]+b[tid];
        }
}


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

        int a[N],b[N],c[N];
        int *dev_a,*dev_b,*dev_c;
        int i;

        /* Allocate memory on GPU */
```

```
cudaMalloc((void **)&dev_a,N*sizeof
cudaMalloc((void **)&dev_b,N*sizeof
cudaMalloc((void **)&dev_c,N*sizeof

/* Fill the host arrays with values
for(i=0;i<N;i++) {
        a[i]=-i;
        b[i]=i*i;
}

cudaMemcpy(dev_a,a,N*sizeof(int),cu
cudaMemcpy(dev_b,b,N*sizeof(int),cu

add<<<N,1>>>(dev_a,dev_b,dev_c);

cudaMemcpy(c,dev_c,N*sizeof(int),cu

/* results */
for(i=0;i<N;i++) {
        printf("%d+%d=%d\n",a[i],b
}
```

```
        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);

        return 0;
}
```

OpenCL

similar to Cuda at least conceptually

Other Accelerator Options

- XeonPhi – came out of the larabee design (effort to do a GPU powered by x86 chips). Large array of x86 chips(p5 class on older models, atom on newer) on PCIe card. Sort of like a plug-in mini cluster. Runs Linux, can ssh into the boards over PCIe. Benefit: can use existing x86 programming tools and knowledge.
- FPGA – can have FPGA accelerator. Only worthwhile if you don't plan to reprogram it much as time delay in reprogramming. Also requires special compiler support

(OpenMP?)

- ASIC – can have hard-coded custom hardware for acceleration. Expensive. Found in BitCoin mining?
- DSPs – can be used as accelerators