ECE 571 – Advanced Microprocessor-Based Design Lecture 21

Vince Weaver

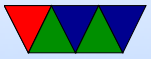
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

14 April 2016

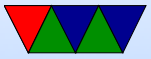
Announcements

- HW#10 will be another reading



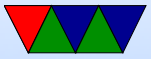
Project Stuff

- Status result/Literature search due on Tuesday
- Would like a brief update on how things are going.
- Do you need to borrow any hardware (only 1 spare wattsup-pro)
- Willing to volunteer for Tuesday rather than Thursday
- Literature search: 5 items for alone, 8 in group
  Prefer if academic, but some things those might not



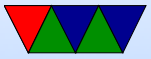
exist so books, web-pages, blogs acceptable too.
If academic and can't find paper, if IEEE, ACM, etc you can log in on UMaine library website and download for free if on campus.

- Cite your references.
- You can fold this into your larger project writeup.
- It's OK if you find out someone else has done your exact project before. It's good to validate results.

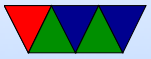


Reading

A Validation of DRAM RAPL Power Measurements
by Desrochers, Paradis and Weaver

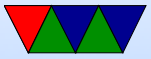


Digression on Academic Papers



Page 1

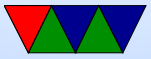
- Work I've been doing with some students.
- MEMSYS conference
- Haswell-EP server with 80GB RAM is 13W of power that's not even with all slots full
  428GFLOPS incidentally (2.1 GFLOPS/w)



Page 2

- Notes on the documentation. Intel tries, but their documentation can be a real pain sometimes, often conflicting and out of date. Also their teminology an be really confusing.

- Instrumenting the hardware
  P4 power connector
  ATX power measurement and previous students
  Hall effect sensors vs sense resistors

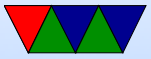


Page 3

- DIMM extender card
  PCIe extender cards
  small resistance. Instrumentation amplifier
  Data acquisition board.

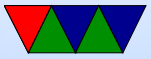
- RAPL measured using perf tool

- Synchronizing the measurements. Hard at high frequencies.
  Other ways to do it? Use serial port and data acquisition



board
On green500 list/wattsup just use NTP to make sure within a second.

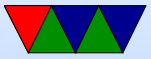
- RAPL overhead, only measure at 10Hz.
  Overhead of too many interrupts, writing to disk. Also power overhead.



Page 4

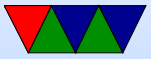
- Benchmark choice.
  idle
  stream
  BLAS: ATLAS, OpenBLAS, MLK

- GPU: OpenCL ray-tracer
  KSP

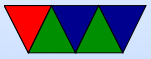


Page 5

- Results
- Figure 2: Idle. Is a system truly idle? It is measuring the perf counters and such.
- Figure 3: Stream benchmark. Package power reads a bit low. DRAM very close when busy, low when idle.
- Figure 4: HPL Atlas Bursty. Note that when LLC miss happens, CPU power goes down (CPI gets worse) but memory power goes up.

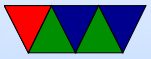


- Figure 5: HPL OpenBLAS
- Figure 6: HPL MKL
- Figure 7: Raytracer
  DRAM behavior not well captured
- Figure 8: KSP: more CPU as at least one CPU is running the physics engine. Again DRAM power not captured well.
- Table results. Why no FLOPS/w for stream? Do ratios hold up?

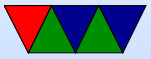


Easy Future Experiments

- Conduct same measurements on other Haswell machines (we have at least two others)
- See if the memory extender causes any slowdown/different RAPL results
- Get another memory extender and see how it works with two DIMMs
- Measure RAPL overhead, can we run at 1kHz if we read

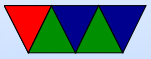


MSR directly too a buffer w/o any other overhead? Still need a timer of some sort.



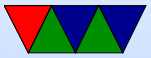
More Difficult Future Experiments

- Measure on server system (our SNB-EP is dead, and HSW-EP is DDR4 which requires two voltage planes and DDR4 extender)



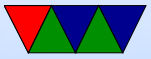
More GPU stuff?

- What is meant by a FLOP? 64/32/16-bit



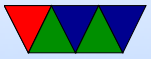
Latency vs Throughput

- 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



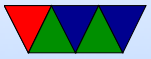
GPGPUs

- 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



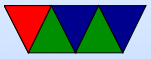
Graphics vs Programmable Use

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



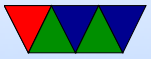
Example for Shader 3.0, came out DirectX9

They are up to Pixel Shader 5.0 now

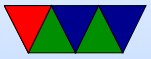


Shader 3.0 Programming – Vertex Processor

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

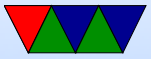


- Can do a four-wide SIMD MAD (multiply ADD) and a scalar op per cycle:
  - EXP, EXPP, LIT, LOGP (exponential)
  - RCP, RSQ (reciprocal, r-square-root)
  - SIN, COS (trig)



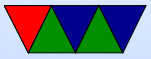
Shader 3.0 Programming – Fragment Processor

- 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)



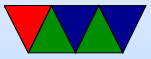
GPGPUs

- 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?

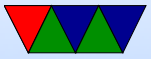


Program

- 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

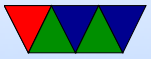


- Kernels == inner loops
- Render-to-texture == feedback
- Geometry-rasterization == computation. Usually done as a simple grid (quadrilateral)
- Texture-coordinates = Domain
- Vertex-coordinates = Range



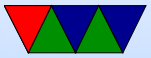
Flow Control, Branches

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



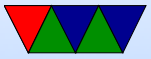
Terminology (CUDA)

- 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



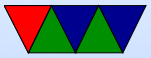
Terminology (cores)

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

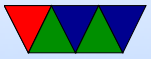


CUDA Programming

- 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

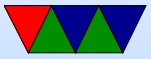


- `__global__` parameters to function – means pass to CUDA compiler
- cudaMalloc() to allocate memory and pointers that can be passed in
- call global function like this `add<<<1,1>>>(args)` where first inside brackets is number of blocks, second is threads per block
- cudaFree() at the end
- Can get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



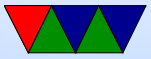
with threadIdx.x

- 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

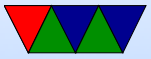


Code Example

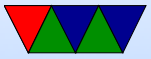
```
#include <stdio.h>

#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];
```



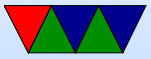
```
        }
}


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 */
```



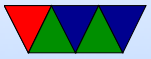
```
cudaMalloc((void **)&dev_a,N*sizeof
cudaMalloc((void **)&dev_b,N*sizeof
cudaMalloc((void **)&dev_c,N*sizeof

/* 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
}
```

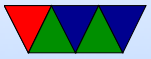


```
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```



OpenCL

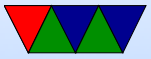
similar to Cuda at least conceptually



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