Understanding `simhwrt` Output

November 22, 2011

Simulator Updates

- You may or may not want to grab the latest update…
- If you have changed the simulator code for your project, it might conflict
- Updates include small improvements to the way output is printed
  - And texture support, which most don't need

Performance Analysis

- Part of your final project will be analyzing HW/SW performance

- This will include:
  - Where is your program spending time?
  - What is causing stalls?
  - What HW changes (if any) might help?
  - Just find something "interesting"

- Document describing your findings/analysis

Performance Analysis

- We will send out an “assignment” pdf with more details of the analysis we want
- Step 1: Make sure your code runs in simhwrt (as you develop)
  - If not, we will help you fix it
- Step 2: Gather and interpret data

`simhwrt` output

- When running with a full chip, simhwrt spits out a ton of numbers

- You will want to redirect stdout to a file

```
./simhwrt …. > output.txt
```

- Most of the output is formatted to be inserted in to a spreadsheet
  - Keeping it as a text file may be sufficient though

`simhwrt` output

- All example output in these slides was generated with the following chip:

  `--num-thread-procs 4 --num-cores 10 --num-l2s 2`

- I’m using a smaller chip mostly so that numbers will fit on slides

- You will want to use the full chip, or something like it

Header Info

- The first part of the output is all data describing the simulation/scene
  - You can pretty much ignore this

- Useful data starts here:

  <=== Core 0 ===>

Thread Status, CPI

<=== Core 0 ===>

---- Thread 00 ----

PC 5: Stalled ----- 696358 in-flight CPI 1.2999 -- Total Cycles 906000

---- Thread 01 ----

PC 5: Stalled ----- 693510 in-flight CPI 1.3053 -- Total Cycles 906000

---- Thread 02 ----

PC 5: Stalled ----- 694825 in-flight CPI 1.3028 -- Total Cycles 906000

---- Thread 03 ----

PC 5: Stalled ----- 691985 in-flight CPI 1.3081 -- Total Cycles 906000

Total CPI 0.3260 , IPC 3.0675 -- Total Cycles 906000

Thread Status, CPI

- Current status of the thread

  "PC 5: Stalled"

  – Program counter 5 = HALT instruction
- Total instructions issued

  "696358 in-flight"
- Cycles per instruction (per thread)

  "CPI 1.2999"
- Total CPI / IPC is TM-wide
- Total cycles

  "Total Cycles 906000"

Total Cycles

- Keep in mind, all threads' clocks cycle simultaneously

- All threads in the TM have to run as long as the longest running thread

Profile Data

kernel	thread(called, cycles)			
	0	1	2	3
0	205, 615	204, 612	201, 603	205, 615
1	205, 15580	204, 15506	201, 15283	205, 15593
2	205, 398303	204, 402263	201, 406379	205, 398070

- My code has 3 profile kernels
  - 0: computing i, j pixel coordinates from atomicinc
  - 1: generating camera ray
  - 2: complete call to shading
- Use the `profile(int)` intrinsic

Profile Parallelism

- For a sufficiently large amount of work, any given thread's profile numbers will be close to average

  205, 398303 204, 402263 201, 406379 205, 398070

- On average the machine spent ~400K cycles on shading
- This program took 906958 total parallel clock cycles
  - Shading is about 44% of the work
  - Don't necessarily need to look at every core

Data Dependence Stalls

Data dependence stalls (caused by):

FPADD	11205	(2.462 %)
FPMUL	39165	(8.605 %)
LOAD	12	(0.003 %)
FPINVSQRT	62168	(13.659 %)
FPDIV	325411	(71.497 %)
DIV	15542	(3.415 %)
FPRSUB	1636	(0.359 %)

- Stalls are counted per-thread (“thread-cycles”)

Data Dependence Stalls

FPADD 11205 (2.462 %)

- Total cycles that instruction’s input data was not ready
  - (and percentage of data dependence cycles)

- Could be due to waiting on the result of a slow instruction (divide, etc)

- Could be waiting on a cache-missed load

Resource Contention

Number of thread-cycles contention found when issuing:

FPMUL	418	(0.196 %)
FPMIN	44927	(21.109 %)
LOAD	27476	(12.910 %)
FPINVSQRT	12	(0.006 %)
STORE	2172	(1.021 %)
ADDI	3619	(1.700 %)
ANDI	22	(0.010 %)

- "thread-cycles" means it counts *all* stalls, even if they happened in parallel
  - This number can be greater than total clock cycles

Resource Contention

FPDIV 131 (0.062 %)

- Total thread-cycles that instruction was unable to issue because the unit was already in use
- Only have 1 shared divide unit, if more than 1 tries to issue on same cycle, stall
- Also caused by cache bank conflicts

Resource Contention

- This number also includes write hazards
  - "pipeline hazards"
- The register file only has 1 write port
- If a slow instruction finishes at the same time as a fast one issued later
  - Can only write 1 register at a time
- This is typically a small percent of FU stalls

Instruction Count

Dynamic Instruction Mix: (2948634 total)

ADD	64744	(2.196 %)
MUL	822	(0.028 %)
BITOR	59768	(2.027 %)
FPADD	64633	(2.192 %)
FPMUL	297742	(10.098 %)
FPMIN	112330	(3.810 %)

…

- Also counted per-thread
  - More instructions than total cycles

Issue Breakdown

--Average #threads Issuing each cycle: 3.0691

--Total thread-cycles: 3624156

--total thread-cycles issued: 2780726 (76.727547%)

--iCache conflicts: 719 (0.019839%)

--thread*cycles of FU dependence: 213057 (5.878803%)

--thread*cycles of data dependence: 455533 (12.569354%)

--thread*cycles halted: 4395 (0.121270%)

Issue breakdown:

--thread*cycles of issue worked: 2780726 (76.727547%)

--thread*cycles of issue failed: 673704 (18.589266%)

--thread*cycles of issue NOP/other: 169726 (4.683187%)

Work Allocation

ATOMIC_INC called by threads:

0: 203

1: 204

2: 207

3: 208

- Ideally the difference between the highest and lowest is a small percent
- Otherwise you have a lot of idle threads
  - i.e. not enough work for the machine to do

Module Utilization

Core 0

Module Utilization

FP AddSub:	3.72
FP MinMax:	0.77
FP Compare:	0.51
Int AddSub:	2.26
FP Mul:	4.11
Int Mul:	0.14
FP InvSqrt:	0.77
FP Div:	2.20
Conversion Unit:	0.01

- Percentage of total issue capacity used

Module Utilization

- If these numbers are high (100%), you will likely reduce stalls by adding more units
  - At the cost of more area
  - More FU downtime
- You may want to minimize OR maximize this number
- For good performance per area, utilization is around 50%

L1 Cache Performance

L1 accesses:	4450236
L1 hits:	4400564
L1 misses:	49672
L1 bank conflicts:	58095
L1 stores:	49152
L1 near hit:	0 (ignore this)
L1 hit rate:	0.988838

- These are averages for each TM
- Only reported chip-wide

L2 Cache Performance

-= L2 #0 =-

L2 accesses:	24848
L2 hits:	234
L2 misses:	24614
L2 stores:	24588
L2 bank conflicts:	190
L2 hit rate:	0.009417
L2 memory faults:	0

L2 bandwidth limited stalls: 21156

Bandwidth

Bandwidth numbers for 1000MHz clock:

register to L1 bandwidth:	19627087872
L1 to L2 bandwidth:	7009841664
L2 to memory bandwidth:	6944216064

- L2 to memory is capped at 32GB/s
  - LOAD will stall if BW exceeded

Local vs. Global

- Keep in mind the only cached memory ops are the global ones
  - LOAD, STORE
  - Only these instructions can affect bandwidth

- Scratchpad memory is separate
  - SW, LW, SWI, LWI, etc…
- These always return in 1 cycle and are in a separate memory space (not cached)
- If the scratchpad overflows, crash

Size and Speed

Core size: 0.4367

L2 size: 0.0000

2-L2 size: 0.0000

20-core chip size: 8.7332

FPS Statistics:

Total clock cycles: 906958

FPS assuming 1000MHz clock: 1102.5869

Size and Speed

- L2-size is deprecated (ignore)
- Core size is TM size (FUs and registers)
- Cache sizes come from a separate table, generated by Cacti

"Total clock cycles" is the longest running thread out of all cores

Analysis

- Primary goal:
  - Find something interesting about the machine running your code based on the simulations
  - Definition of “interesting” will depend on the project
  - Provide insight in to behavior, suggest potential changes to the system

Analysis

- Where is your program spending time?
  - (profile the biggest 3-5 phases of your code)
- What is causing stalls?
  - Do you need more of a particular FU?
  - Do you need bigger caches, more banks?
  - Is your code inefficient? (avoidable divides, etc...)
  - Maybe it doesn’t need anything (a successful issue rate of 75% is extremely good, 40% is pretty bad)
- Does your project have specific HW needs that plain path tracing does not?
- What, if anything, might you change about the architecture?
- How is/isn’t the architecture suited to your code in general?

Analysis

- Turn in a small (1-3 pages or so) pdf with your analysis
- Graphs/charts will help
- Make sure to run on a large enough problem (at least 128x128) to avoid the machine being too big for the work
- Start with the 32x20x4 thread machine, using default.config

Analysis

- We ran 1000's of simulations to come up with the configuration we have
- Obviously we don't expect that level of analysis
  - Just as long as you show you're thinking about it and have an idea of why it performs as it does