18-643 Lecture 5: Performance!=Performance

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today: appreciate the many subtleties and dimensionalities of performance
  Digested from three 18-447 lectures

- Notices
  - Handout #3: lab 1, due noon, 9/23
  - Zedboard ready for pick up (see Handout #3a)
  - Recitation starts this week, Wed 4:30~5:20

- Readings (for background)
The Zedboard Kit

- ECE is loaning to you
  - 1x Avnet ZedBoard 7020 baseboard
  - 1x 12 V power supply
  - 1x 4-GB SD Card (don’t need to use it)
  - 2x Micro-USB cable (1 in box; 1 additional)
  - 1x USB Adapter (don’t need to use it)
  - 1x set of documentations
  - 1x voucher to Xilinx Vivado® Design Edition license for 7Z020-only (keep/use it or give it back)

- You will treat it like your grade depends on it
- You will return all of it in perfect condition, or else

Looking Ahead

- Lab 1 (wk3/4): get cozy with Vivado
  - most important: learn logic analyzer and eclipse debugger
- Lab 2 (wk5/6): meet Vivado HLS
  - most important: decide if you would use it
- Lab 3 (wk7/8): hands-on with AFU
  - most important: have confidence it can work
- Project . . .
Performance is about time

- To the first order, performance $\propto 1 / \text{time}$
- Two very different kinds of performance!!
  - latency = time between start and finish of a task
  - throughput = number of tasks finished in a given unit of time (a rate measure)
- Either way, shorter the time, higher the performance, but . . .

Throughput $\neq 1$/Latency

- If it takes $T$ sec to do $N$ tasks, throughput=$N/T$; latency=$T/N$?
- If it takes $t$ sec to do 1 task, latency=$t$; throughput=$1/t$?
- When there is concurrency, throughput$\neq 1$/latency
- Optimizations can tradeoff one for the other
  (think bus vs F1 race car)
Throughput $\neq$ Throughput

- Throughput becomes a function of $N$ when there is a non-recurring start-up cost (aka overhead).
- For start-up-time = $t_s$ and throughput$_{raw} = 1/t_1$
  - throughput$_{effective} = N / (t_s + N \cdot t_1)$
  - if $t_s >> N \cdot t_1$, throughput$_{effective} \approx N/t_s$
  - if $t_s << N \cdot t_1$, throughput$_{effective} \approx 1/t_1$
    we say $t_s$ is “amortized” in the latter case.
- E.g., DMA transfer on a bus
  - $10^{-6}$ sec to setup a DMA
  - bus throughput$_{raw} = 1$ Byte / $(10^{-3}$ sec)
  - throughput$_{effective}$ to send 1B, 1KB, 1MB, 1GB?

Latency $\neq$ Latency

- What are you doing during the latency period?
- Latency = hands-on time + hands-off time
- In the DMA example
  - CPU is busy for the $t_s$ to setup the DMA
  - CPU has to wait $N \cdot t_1$ for DMA to complete
  - CPU could be doing something else during $N \cdot t_1$ to “hide” that latency

```
          CPU   t_s  t_s   t_s
          |     |     |
        bus  N*t_1  N*t_1
```
Relative Performance

- Pop Quiz: if X is 50% slower than Y and latency_X=1.0s, what is latency_Y?
  
  - Case 1: L_Y = 0.5s since L_Y/L_X=0.5
  
  - Case 2: L_Y = 0.66666s since L_X/L_Y=1.5

  English language is imprecise

Fixing the language, a la H&P

- “X is n times faster than Y” means
  
  \[ \frac{\text{throughput}_X}{\text{throughput}_Y} = \frac{\text{latency}_Y}{\text{latency}_X} \]

- “X is m% faster than Y” means
  
  \[ 1 + \frac{m}{100} = \frac{\text{Performance}_X}{\text{Performance}_Y} \]

- Delete “slower” from your dictionary

- On last slide
  
  - for case 1 say, “Y is 100% faster than X”
  
  - for case 2 say, “Y is 50% faster than X”
Faster!=Faster

• Given two designs X and Y,
  – X may be $m\%$ faster than Y on input A
  – X may be $n\%$ (where $m\neq n$) faster than Y on input B
  – Y may be $k\%$ faster than X on input C

• Which is faster and by how much?
  – depends on which input(s) you care about
  – if multiple, also depend on relative importance

• Many ways to summarize performance into a scalar metric to simplify comparison
  – more wrong ways than right
  – when in doubt, present the complete story
  – go read the H&P chapter on performance

Multi-Dimensional Optimizations

• HW design has many optimization dimensions
  – by area, by resource type utilization
  – performance and latency
  – power and energy
  – complexity, risk, social factors . . .

• Cannot optimize individual metrics without considering tradeoff between them, e.g.,
  – reasonable to spend more power for performance
  – converse also true (lower perf. for less power)
  – but never more power for lower performance
Pareto Optimality (2D example)

All points on front are optimal (can’t do better)

How to select between them?

Application-Defined Composite Metrics

- Define scalar function to reflect desiderata---incorporate dimensions and their relationships
- E.g., energy-delay-(cost) product
  - smaller the better
  - can’t cheat by minimizing one ignoring others
  - not required to have physical meaning
- Floors and ceilings
  - real-life designs more often about good enough than optimal
  - e.g., meet a perf floor under a power(cost)-ceiling (minimize design time, i.e., stop when you get there)
Power != Energy
(do not confuse them)

Power = Energy / time

- Energy (Joule) dissipated as heat when “charges” flow from VCC to GND
  - takes a certain amount of energy per operation, e.g., addition, reg read/write, (dis)charge a node
  - to the first order, energy \( \propto \) work
- Power (Watt=Joule/s) is rate of energy dissipation
  - more op/sec then more Joules/sec
  - to the first order, power \( \propto \) performance

It is all very easy if performance \( \propto \) frequency

\[ \text{Power} = (0.5CV^2) \cdot f \]
Power and Performance not Separable

• Easy to minimize power if don’t care about performance
• Expect superlinear increase in power to increase performance
  – slower design is simpler
  – lower frequency needs lower voltage
• Corollary: Lower perf also use lower J/op (=slope from origin)

All in all, slower is more energy/power efficient

When slower takes more energy

• Devices leak charge even when doing no ops
  – so called leakage current ($I_{\text{leakage}}$)
  – $\text{energy}_{\text{total}} = (J/\text{op})@\text{perf} \cdot \text{work} + (VCC \cdot I_{\text{leakage}}) \cdot \text{time}$
  – $\text{power}_{\text{total}} = (J/\text{op})@\text{perf} \cdot \text{perf} + (VCC \cdot I_{\text{leakage}})$
    = switching power + static power
• Slower reduces power but could increase energy
Perf/Watt != Perf/Watt

- Perf/Watt is a normalized measure
  - hides the scale of problem and platform
  - recall, Watt \( \propto \text{perf}^k \) for some \( k>1 \)
- 10 GFLOPS/Watt at 1W is a very different design problem than at 1KW or 1MW or 1GW
  - say 10 GFLOPS/Watt on a <GPGPU,problem>
  - now take 1000 GPEGPUUs to the same problem
  - realized perf is \(< 1000x\) (less than perfect parallelism)
  - required power \( > 1000x\) (energy to move data & heat)

In general be careful with normalized metrics

Parallelism and Performance
Parallelism Defined

- \( T_1 \) (work measured in time):
  - time to do work with 1 PE
- \( T_\infty \) (critical path):
  - time to do work with infinite PEs
  - \( T_\infty \) bounded by dataflow dependence
- Average parallelism:
  \[ P_{\text{avg}} = \frac{T_1}{T_\infty} \]
- For a system with \( p \) PEs
  \[ T_p \geq \max\{ \frac{T_1}{p}, T_\infty \} \]
- When \( P_{\text{avg}} > > p \)
  \[ T_p \approx \frac{T_1}{p} \], aka “linear speedup”

Linear Parallel Speedup

- Ideally, parallel speedup is linear with \( p \)

\[ \text{speedup} = \frac{\text{runtime}_{\text{sequential}}}{\text{runtime}_{\text{parallel}}} \]

- runtime
  - \( \propto \frac{1}{p} \)

- speedup
  - \( p=1 \rightarrow 1 \)}
**Linear Speedup! = Linear Speedup**

How could this be?

**It could be worse . . . . . . .**

limited scalability, $P_{avg} < p$
**Parallelization Overhead**

- Best parallel and seq. algo. need not be the same
  - best parallel algo. often worse at \( p=1 \)
  - if \( \text{runtime}_{\text{parallel@p=1}} = K \cdot \text{runtime}_{\text{sequential}} \)
    then best-case speedup = \( \frac{p}{K} \)
- Communication between PEs not instantaneous
  - extra time for the act of sending or receiving data
    as if adding more work (\( T_1 \))
  - extra time waiting for data to travel between PEs
    as if adding critical path (\( T\_\infty \))
  If overhead grows with \( P \), speedup can even fall

**Arithmetic Intensity**

- An algorithm has a cost in terms of operation count
  - runtime = \# operations / FLOPS
  - parallelization increases FLOPS
- An algorithm also has a cost in terms of number of bytes read/written (or sent/received in general)
  - runtime = \# bytes / memory BW
  - parallelization does not always increase BW
- Average Arithmetic Intensity (AI)
  - how many ops performed per byte accessed
  - \# operations / \# bytes
Attained Performance of a system (op/sec) vs AI of application

- AI is a function of the algorithm and the problem size
- Higher AI means more work per communication and therefore easier to parallelize and to scale

[Figure 6.17, Computer Organization and Design]
Arithmetic Intensity Example: MMM

- Multiplying two NxN matrices of doubles
  ```c
  void mmm(Number* A, Number* B, Number* C, int N) {
      int i, j, k, temp=0;
      for (j = 0; j < N; j++)
          for (i = 0; i < N; i++)
              for (k = 0; k < N; k++)
  }
  ```
- Best-case “off-chip” memory access (fits in cache)
  - read 2xN²x8 bytes, two input matrices of doubles
  - write N²x8 bytes, output matrix
- Operation count
  - N³ dbl-mult and N³ dbl-add (by standard algorithm)
  - BTW, you can do O(n².376) but the “C” is huge
- Arithmetic Intensity = 2N³/(24N²)=N/12

Strong vs Weak Scaling

- Speedup is a normalized metric
  always be careful with normalized metrics!!
- Strong Scaling: T₁ (work) remains constant
  - i.e., improve same workload with more PEs
  - hard to maintain linearity as p grows toward p_avg
- Weak Scaling: T₁’=p·T₁
  - i.e., improve larger workload with more PEs
  - S_p=runtime_sequential(p·T₁)/runtime_parallel(p·T₁)
  - is this harder or easier?
  - ans: depends on how T∞ scales with T₁’ and how AI scales with T₁’
Amdahl’s Law: First and Last Word on Performance

- If only a fraction \( f \) is parallelizable by a factor of \( p \)

\[
\begin{array}{c}
\text{time}_{\text{sequential}} \\
\hline
(1 - f) \quad f \\
\hline
\text{time}_{\text{parallelized}} \\
(1 - f) \quad f/p
\end{array}
\]

\[
\text{time}_{\text{parallel}} = \text{time}_{\text{sequential}} \cdot ( (1-f) + f/p )
\]

\[
\text{speedup} = \frac{1}{(1-f) + f/p}
\]

- if \( f \) is small, \( p \) doesn’t matter
- even when \( f \) is large, diminishing return on \( p \); eventually “1-f” dominates

Parting Thoughts

- HW/FPGA design and optimization must address multiple dimensions (each one nuanced)
  - optimizations often involve tradeoff
  - over simplifying is dangerous and misleading
  - must understand application needs

  power and energy is first-class

- Pay attention to cost and bottleneck of the entire system
- Real-life designs have non-technical requirements