# CS 5220

## Performance Basics

David Bindel

2024-08-29

# Soap Box

# The Goal

The goal is right enough, fast enough — not flop/s.

# More than Speed

Performance is not all that matters.

- Portability, readability, ease of debugging, …
- Want to make intelligent tradeoffs

# Start at the Beginning

The road to good performance
starts with a single core.

- Even single-core performance is hard
- Helps to build well-engineered libraries

# Fair Comparisons

Parallel efficiency is hard!

- $p$ processors $\neq$ speedup of $p$
- Different algorithms parallelize differently
- Speed vs untuned serial code is cheating!

# Peak Performance

# Whence Rmax?

Top 500 benchmark reports:

- Rmax: Linpack flop/s
- Rpeak: Theoretical peak flop/s

Measure the first; how do we know the second?

# What is a float?

Start with what is floating point:

- (Binary) scientific notation
- Extras: inf, NaN, de-normalized numbers
- IEEE 754 standard: encodings, arithmetic rules

# Formats

- *64-bit double precision (DP)*
- *32-bit single precision (SP)*
- Extended precisions (often 80 bits)
- 128-bit quad precision
- 16-bit half precision (multiple)
- Decimal formats

Lots of interest in 16-bit formats for ML. Linpack results are double precision

# What is a flop?

- Basic floating point operations: $+, -, \times, /, \sqrt{\cdot}$
- FMA (fused multiply-add): $d = ab + c$
- Costs depend on precision and op
- Often focus on add, multiply, FMA (“flams”)

# Perlmutter specs

Consider Perlmutter

# Flops / cycle / core

Processor does more than one thing at a time. On one CPU core of Perlmutter (AMD EPYC 7763 (Milan)):

$$2\frac{\text{flops}}{\text{FMA}} \times 4\frac{\text{FMA}}{\text{vector FMA}} \times 2\frac{\text{vector FMA}}{\text{cycle}} = 16\frac{\text{flops}}{\text{cycle}}$$

## Flops / sec / core

At standard clock (2.45 GHz)

$$16\,\frac{\text{flops}}{\text{cycle}} \times 2.4 \times 10^9\,\frac{\text{cycle}}{\text{s}} = 39.2\,\frac{\text{Gflop}}{\text{s}}$$

At max boost clock (3.5 GHz)

$$16\,\frac{\text{flops}}{\text{cycle}} \times 3.5 \times 10^9\,\frac{\text{cycle}}{\text{s}} = 56\,\frac{\text{Gflop}}{\text{s}}$$

# Flops / sec / CPU

Each CPU has 64 cores, at standard clock

$$39.2\frac{\text{Gflop}}{\text{s}} = 2508.8\frac{\text{Gflop}}{\text{s}} \approx 2.5\frac{\text{Tflop}}{\text{s}}$$

Peak CPU flop/s by partition:

- GPU: $2.5808$ Tflop/s/CPU $\times 1536$ CPU $\approx$ 3.9 Pflop/s
- CPU: $2.5808$ Tflop/s/CPU $\times 2$ CPU/node $\times 3072$ nodes $\approx 15.4$ Pflop/s
    - NERSC docs inconsistent re 2 CPU/node?

# Flops / sec / GPU

- GPU partition nodes have 4 NVIDIA A100 each.
- Different peak performance depending on FP type (9.7 Tflop/s FP64)

# But...

Rpeak $>$ Rmax $>$ Gordon Bell $>$ Typical

- Performance is *application dependent*
- Hard to get more than a few percent on most

Consider HPCG - June 2024.

Problem: Data movement is expensive!

# Serial Costs

# Naive Matmul

```
void square_dgemm(int n, double* C, double* A, double* B)
{
    // Accumulate C += A*B for n-by-n matrices
    for (i = 0; i < n; ++i)
      for (j = 0; j < n; ++j)
        for (k = 0; k < n; ++k)
          C[i+j*n] += A[i+k*n] * B[k+j*n];
}
```

- Inner product formulation of matrix multiply
- Takes $2n^3$ flops
- Cost is much more than Rpeak suggests!
- Problem is communication cost / memory traffic

# Price to Fetch

Two pieces to cost of fetching data

**Latency** Time from operation start to first result (s)

**Bandwidth** Rate at which data arrives (bytes/s)

## Price to Fetch

- Usually latency $\gg$ bandwidth$^{-1}$ $\gg$ time per flop
- Latency to L3 cache is 10s of ns
- DRAM is $3-4\times$ slower
- Partial solution: caches (to discuss next time)

See: Latency numbers every programmer should know

# Price to Fetch

- Lose orders of magnitude if too many memory refs
- And getting full vectorization is also not easy!
- We'll talk more about (single-core) arch next time

# Takeaways

Start with a simple model

- But flop counting is *too* simple
- Counting every detail complicates life
- Want enough detail to predict something

# Watch for Hidden Costs

- Flops are not the only cost!
- Memory/communication costs are often killers
- Integer computation may play a role, too

# Parallelism?

Picture gets even more complicated!

# Parallel Costs

# Naive model

Too simple:

- Serial task takes time $T(n)$
- Deploy $p$ processors
- Parallel time is $T(n)/p$

# What’s Wrong?

Why is parallel time not $T(n)/p$?

- **Overheads:** Communication, synchronization, extra computation and memory overheads
- **Intrinsically serial** work
- **Idle time** due to synchronization
- **Contention** for resources

# Quantifying Performance

- Start with good *serial* performance
- (Strong) scaling study: compare parallel vs serial time as a function of $p$ for a fixed problem

$$\text{Speedup} = \frac{\text{Serial time}}{\text{Parallel time}}$$
$$\text{Efficiency} = \frac{\text{Speedup}}{p}$$

# Quantifying Performance

Perfect (linear) speedup is $p$. Barriers:

- Serial work (Amdahl's law)
- Parallel overheads (communication, synchronization)

If $s$ is the fraction that is serial:

$$\text{Speedup} < \frac{1}{s}$$

Looks bad for strong scaling!

**Strong scaling** Fix problem size, vary $p$

**Weak scaling** Fix work per processor, vary $p$

# Scaled Speedup

Scaled speedup

$$S(p) = \frac{T_{\text{serial}}(n(p))}{T_{\text{parallel}}(n(p), p)}$$

Gustafson:

$$S(p) \leq p - \alpha(p-1)$$

where $\alpha$ is fraction of serial work.

# Imperfect Parallelism

Problem is not just with purely serial work, but

- Work that offers limited parallelism
- Coordination overheads.

# Dependencies

Main pain point: *dependency* between computations

```
a = f(x)
b = g(x)
c = h(a,b)
```

Can compute $a$ and $b$ in parallel with each other.

But not with $c$!

True dependency (read-after-write). Can also have issues with false dependencies (write-after-read and write-after-write), deal with this later.

# Granularity

- Coordination is expensive
  - including parallel start/stop!
- Need to do enough work to amortize parallel costs
- Not enough to have parallel work, need big chunks!
- Chunk size depends on the machine.

# Patterns and Benchmarks

# Pleasing Parallelism

“Pleasingly parallel” (aka “embarrassingly parallel”) tasks require very little coordination, e.g.:

- Monte Carlo computations with independent trials
- Mapping many data items independently

Result is “high-throughput” computing – easy to get impressive speedups!

Says nothing about hard-to-parallelize tasks.

# Displeasing Parallelism

If your task is not pleasingly parallel, you ask:

- What is the best performance I reasonably expect?
- How do I get that performance?

# Partly Pleasing Parallelism?

Matrix-matrix multiply:

- Is not pleasingly parallel.
- Admits high-performance code.
- Is a prototype for much dense linear algebra.
- Is the key to the Linpack benchmark.

Look at examples somewhat like yours – a *parallel pattern* – and maybe seek an informative benchmark. Better yet: reduce to a previously well-solved problem (build on tuned *kernels*).

NB: Uninformative benchmarks will lead you astray.

# Recap

# Recap

Speed-of-light "Rpeak" is hard to reach

- Communication (even on one core!)
- Other overhead costs to parallelism
- Dependencies limiting parallelism

Want

- *Models* to understand real performance
- *Building blocks* for getting high performance