Improving serial performance

Prabhu Ramachandran

Department of Aerospace Engineering

IIT Bombay

High performance computing (HPC)

- Serial
- Parallel

Measuring performance

- FLOPS
- Megaflops: 10^6
- Gigaflops: 10^9
- Teraflops: 10^{12}
- Petaflops: 10^{15}
- Exaflops: 10^{18}

Count the flops

In [ ]:
```
import numpy as np
# Some made-up data.
N = 10000
b = np.linspace(0, 10, N)
c = np.random.random(N)
d = np.random.random(N)
a = np.empty_like(b)
```

In [ ]:
```
# Some computation
iters = 1000
for k in range(iters):
    for i in range(N):
        a[i] = b[i] + c[i]*d[i]
```

FLOPS and benchmarks

- Count the basic arithmetic operations
- Also count the LOAD and STORE operations
- Low-level benchmark
- Application benchmarks

Improving serial performance

- Important to understand architecture

Cache-based microprocessor

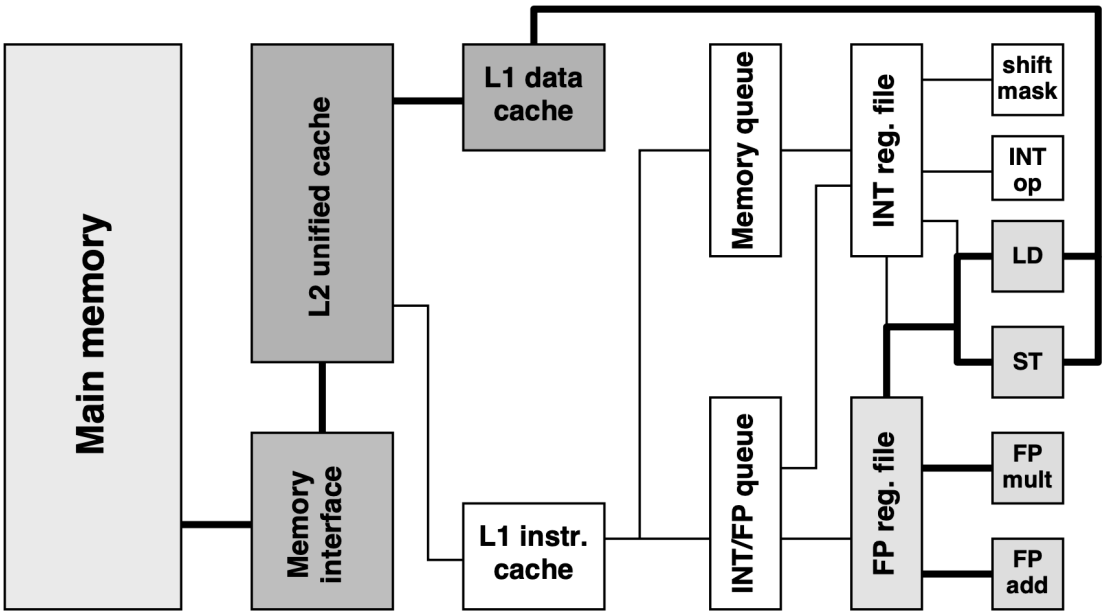


Image source: Introduction to HPC ..., Hager and Wellein 2010

Architecture basics

- Instructions and Data: both are cached
- Steps (Machine code):
  1. Fetch instruction
  2. Decode
  3. Operands
  4. Execute operation
  5. Store results
  6. Next instruction

Memory hierarchy

1. Limited CPU registers
2. Cache L1 $<$ L2 $<$ L3
3. RAM
4. Storage

Instruction Set Architecture (ISA)

- An abstraction in terms of programming instructions (at the machine level)
- Different implementations may exist for a given ISA
- x86-64 is an ISA
- Intel Core i7, AMD XXYY is an implementation
- ARMv7-A is an ISA

Virtual memory vs physical memory

- OS provides virtual memory
- Virtual memory mapped to physical memory
- Memory stored in "pages"
- This is also managed and costs

More architectural considerations

- Pipelining: multiple functional units, **assembly line**
- Superscalar: multiple assembly lines
- Out of order execution
- Branch prediction and speculative execution
- SIMD: SSE, AVX etc.: Vector operations
- Caches
- Instruction set differences: CISC vs. RISC

Illustrations

Beans

Assembly



Superscalar or Pipelining or SIMD?



Caches

- Faster memory but limited amount
- L1, L2, L3
- Cache lines: **64 bytes** typically (512 bits)
- Memory fetched per cache line
  - Takes time to get data from memory
- Writing to memory will also flush a cache line
- Prefetching (hardware and software)

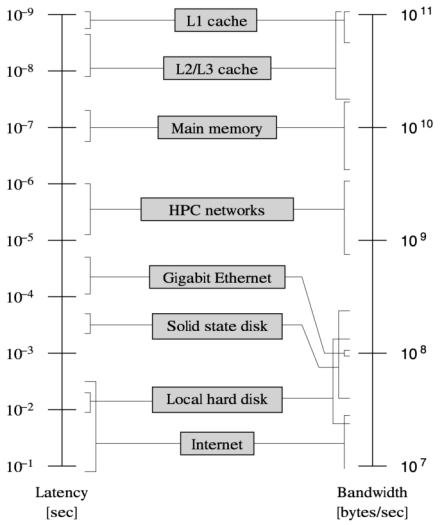
- Memory access patterns are important

Unexpected effects

In [ ]:
```
for i in range(N):
    a[i] = b[i] + c[i]*d[i]
```

In [ ]:
```
# Compare with
for i in range(1, N, 3):
    a[i] = b[i] + c[i]*d[i]
```

Memory latency and bandwidth



Slightly dated.

Image source: Introduction to HPC ..., Hager and Wellein 2010

Another resource (2012)

```
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns
14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns
20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns
Send 1K bytes over 1 Gbps network       10,000   ns
Read 4K randomly from SSD*             150,000   ns    ~1GB/
sec SSD
Read 1 MB sequentially from memory     250,000   ns
Round trip within same datacenter      500,000   ns
Read 1 MB sequentially from SSD*     1,000,000   ns    ~1GB/
sec SSD, 4X memory
Disk seek                           10,000,000   ns    20x d
atacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns    80x m
emory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns
```

Source: https://stackoverflow.com/questions/4087280/approximate-cost-to-access-various-caches-and-main-memory

Source: https://gist.github.com/jboner/2841832

More recent (2020)

```
L1 CACHE hit, ~4 cycles(2.1 - 1.2 ns)
L2 CACHE hit, ~10 cycles(5.3 - 3.0 ns)
L3 CACHE hit, line unshared              ~40 cycles(21.4 -
12.0 ns)
L3 CACHE hit, shared line in another core ~65 cycles(34.8 -
19.5 ns)
L3 CACHE hit, modified in another core    ~75 cycles(40.2 -
22.5 ns)
local  RAM
~60 ns
```

Source: https://www.forrestthewoods.com/blog/memory-bandwidth-napkin-math/

Simple steps for better performance

1. Profile your code! Focus on the *hot spots*:
    - Function and line profiling
    - Low-level tools (PAPI, OProfile, VTune)
2. Common sense:
    A. Do less work
    B. Avoid expensive operations
    C. Shrink the working set
3. Some simple measures
    A. Eliminate common subexpressions
    B. Avoid branches
    C. Avoid creating too many temporaries
4. Compiler:
    A. Use proper optimization flags on your compiler (-O2, -O3)
    B. Use inlining
    C. Avoid pointer aliasing

Other Profilers

- Scalene is Python specific
- PAPI: Linux only, Python wrappers exist
- OProfile: Linux only
- LIKWID: also pylikwid (Linux only)
- valgrind/cachegrind
- Non-Open Source:
    - Intel VTune: non-Open Source
    - AMD μProf: AMD specific

Example: less work

In [ ]:
```
flag = False
for i in range(N):
    if expensive_call(A[i]) < value:
        flag = True
```

Versus:

In [ ]:
```
flag = False
for i in range(N):
    if expensive_call(A[i]) < value:
        flag = True
        break
```

Example: avoid expensive ops, simplify expressions

- Avoid useless expressions
- Simplify code -- do you really need a square root?
- Precompute values

In [ ]:
```
for i in range(N):
   a[i] = sin(i%5)
   b[i] = pow(x, float(i))
```

In [ ]:
```
for i in range(N):
   a[i] = (s + r*sin(x))*i
```

versus

In [ ]:
```
tmp = s + r*sin(x)
for i in range(N):
   a[i] = tmp*i
```

Example: avoid branches

In [ ]:
```
for i in range(N):
    for j in range(N):
        if i >= j:
            sign = 1.0
        elif i < j:
            sign = -1.0
        else:
            sign = 0.0
        c[j] += sign*a[i, j]*b[i]
```

Versus

In [ ]:
```
for i in range(N):
    for j in range(i+1):
        c[j] = c[j] + a[i,j]*b[i]

for i in range(N):
    for j in range(i+1, N):
        c[j] = c[j] - a[i,j]*b[i]
```

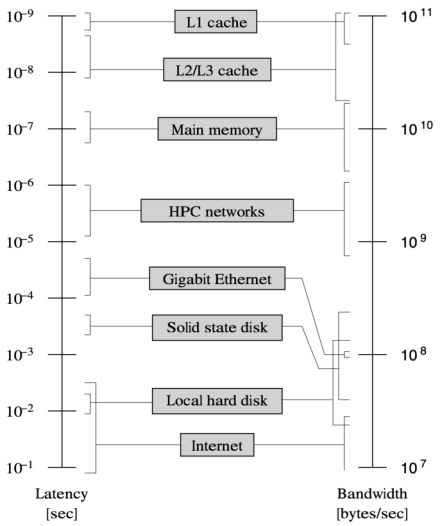
More involved approaches

- Use SIMD (involves work and hardware specific)
  - Compilers can use SIMD automatically
  - Do not use conditionals
  - Don't introduce loop-dependencies
- Think about:
  - "Compute bound"
  - "Memory bound"
- **Data access optimization**
- Memory access patterns

Memory hierarchies/Cache

- Makes a significant difference
- Cache is important
- How is cache used?

Memory latency and bandwidth



Slightly dated.

Image source: Introduction to HPC ..., Hager and Wellein 2010

Structured approach

- Load/Store vs Compute
- Measuring code balance
- Counting operations

Balance analysis

- Machine balance: B_m = Memory bandwidth/Peak performance
- $B_m = \frac{b_{max}}{P_{max}}$ = GWords/sec / GFlops/sec
- b_{max} = 30 GBps, and 3 GHz with 4 flops per cycle
- i7-7820HQ: 4 DP FLOPS per cycles = 0.055 W/F
- Assuming a 16 DP FLOPS per cycle = 0.01375 W/F

Code Balance

- B_c = data traffic [DP Words] / FLOPS
- Keep this small!
- Computational intensity = $1/B_c$
- Lightspeed = $\min(1, \frac{B_m}{B_c})$
- $P = lP_{max}$

Example

In [ ]:
```
s = 3.0 # ignore this!
for i in range(N):
   a[i] = b[i] + c[i]*d[i]
```

- 2 FLOPS
- 3 LOADs, 1 STORE
- $B_c = 4/2 = 2.0$
- Usually stores count for 2
- $B_c = 5/2 = 2.5$
- If $B_m = 0.1$
- Lightspeed = 0.04

A simple benchmark

In [ ]:
```
import numpy as np
from numba import njit
```

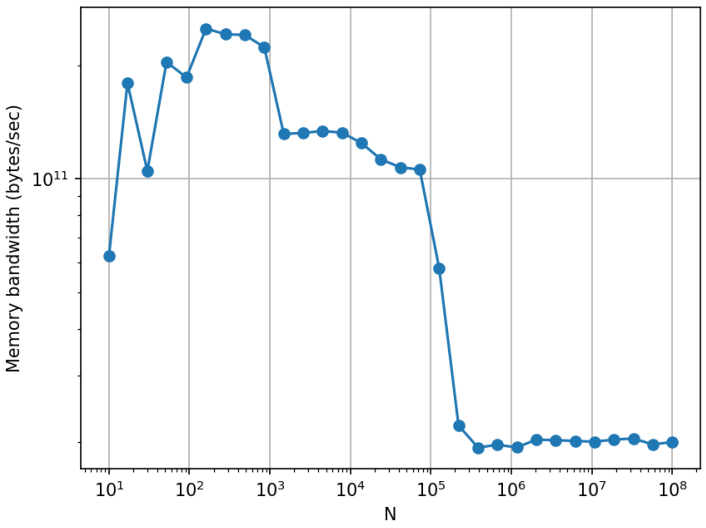
In [ ]:
```
def make_data(n):
    x = np.linspace(0, 2*np.pi, n)
    a, b = np.random.random((2, n))
    y = np.zeros_like(x)
    return y, x, a, b
```

In [ ]:
```
@njit
def axpb(y, x, a, b):
    for i in range(y.shape[0]):
        y[i] = a[i]*x[i] + b[i]
```

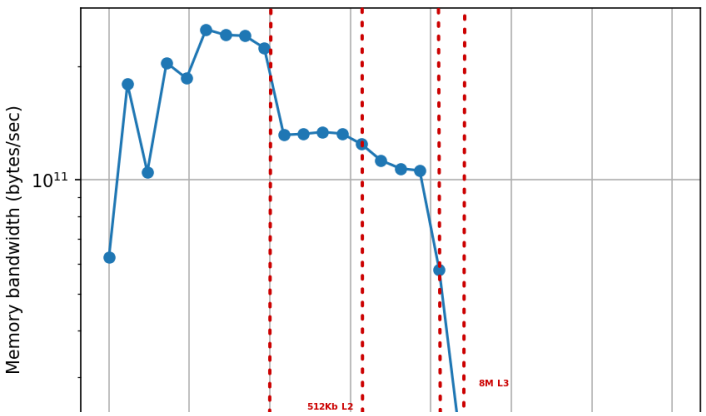
Measuring time accurately

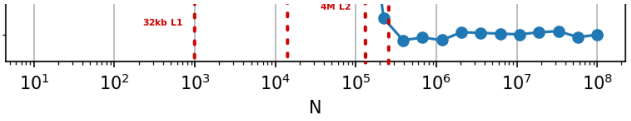
- Can use the `timeit` module but a bit tricky
- Can use `time.perf_counter`
- Invoke once to warm the jit
- Measure the time for a single loop
- Run as many loops to run for some small time, say 0.1
- Do this a few times, say 5 or 7
- Find the minimum of these and report that
- Repeat this as N changes, plot it.

Some results: Memory bandwidth for axpb

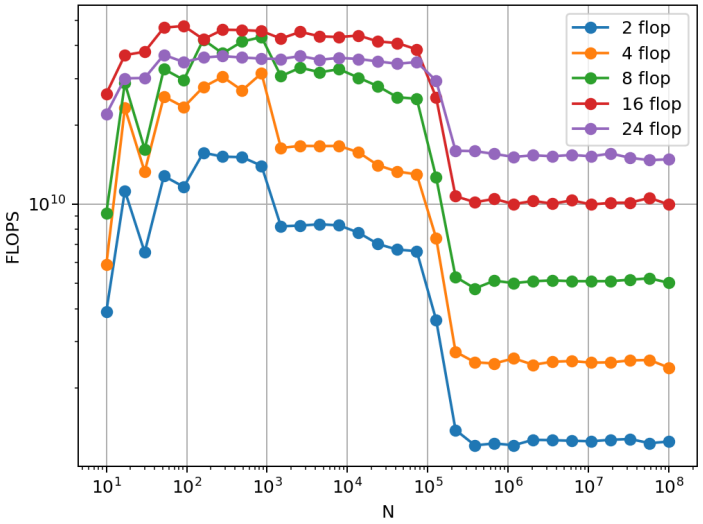


Some results: Memory bandwidth for axpb

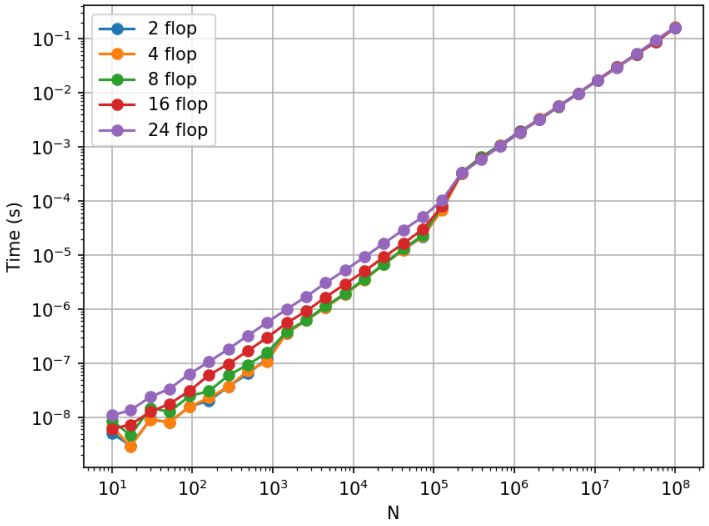




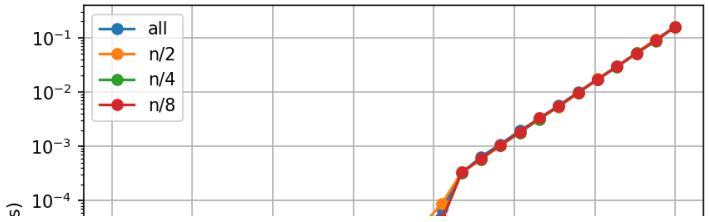
FLOPS vs N

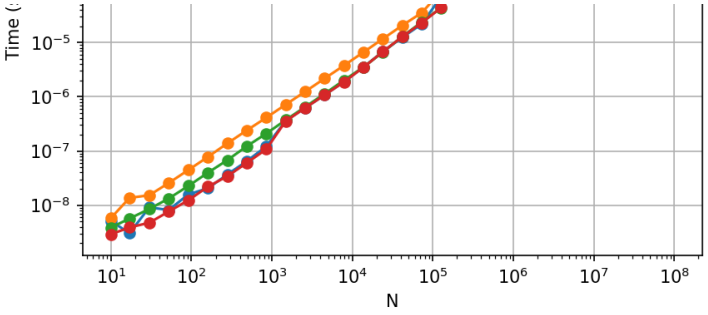


Time taken vs N

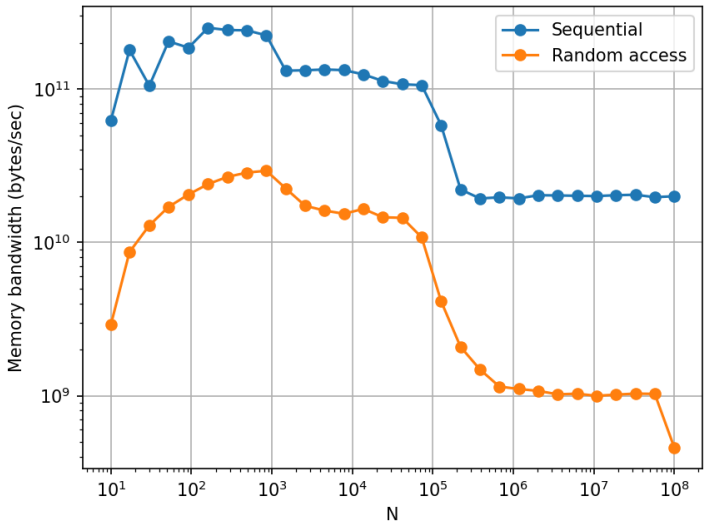


Time taken vs N with a stride





Memory bandwidth for axpb but random access



Matrix storage/access order

- C-style storage: Row-major
- FORTRAN-style storage: Column-major

```
N = 256
A = np.zeros((N, N))
```

- Beware of doing this right

In [ ]:
```
for i in range(N):
    for j in range(N):
        a[i, j] = i*j
```

Matrix-vector multiplication

- Simple example

In [ ]:
```
@njit
def matvec_col(A, x):
        N = A.shape[0]
        res = np.zeros_like(x)
        for i in range(N):
                for j in range(N):
                        res[i] += A[i, j]*x[j]
        return res
```

In [ ]:
```
@njit
def matvec_row(A, x):
        N = A.shape[0]
        res = np.zeros_like(x)
        for j in range(N):
                for i in range(N):
                        res[i] += A[i, j]*x[j]
        return res
```

In [ ]:
```
def make_data(n):
    A = np.random.random((n, n))
    x = np.random.random(n)
    return A, x
```

In [ ]:
```
n = 1000
A, x = make_data(n)
```

In [ ]:
```
%timeit matvec_row(A, x)
```

In [ ]:
```
%timeit matvec_col(A, x)
```

Loop fusion, unroll, jam

- Fusing loops helps, instead of two loops, fuse them into one.
- Consider matrix vector multiplication
- Nested loops
  - Unroll loops
  - Fusing operations (jam)

Roofline model

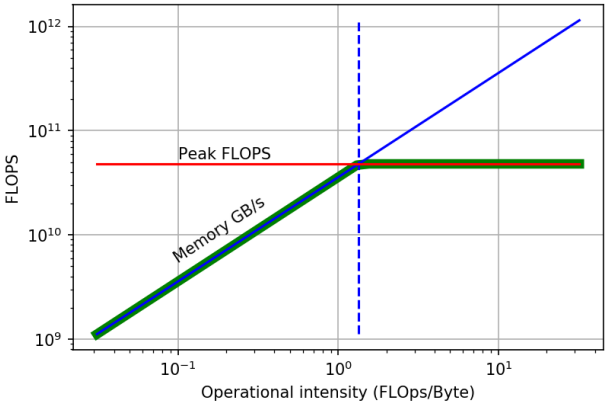
- Model to visualize/analyze performance
- Memory bandwidth limitation vs computation limitation
- Modern hardware tends to be throughput limited
- Latency is hidden due to prefetching, parallelism etc.

- What do you need to focus on for optimization?
- Compare different architectures?

Roofline definition

- $T = \max((\text{FP ops}/\text{Peak GFLOPS}), \text{Bytes}/\text{GBps})$
- $\text{FP ops}/T = \min(\text{Peak GFLOPS}, \text{FP ops} * \text{Peak GBps}/\text{Bytes})$

Roofline example



Some rough data

- Simple hand written benchmarks
- Very hardware specific
- These are for: Intel i3-6100 CPU
- Instrumented code using PAPI (Linux only)

Rough FLOPS for math operations

- Int add sub mul < Float add sub mul
- Integer division is generally very slow
- Float division is 3x slower than multiplication
- sqrt: O(5) flops!?
- sine: 28
- cosine: 28
- tan: 28
- asin: 22
- exp: 31
- log: 34
- tanh: 35
- pow: ≈ 85 flops!!

Summary

- Follow the simple steps first
- Be aware of how you access data
- Optimize based on this
- Beware of math operations
- Compute bound vs. memory bound

References

1. Introduction to High Performance Computing for Scientists and Engineers, Georg Hager and Gerhard Wellein, CRC Press, 2010.
2. Multicore and GPU Programming An Integrated Approach, Gerassimos Barlas, Morgan Kaufmann, 2015.
3. Roofline: an insightful visual performance model for multicore architectures, Samuel Williams and Andrew Waterman and David Patterson, Communications of the ACM, 4, pp 65--76 2009. https://doi.org/10.1145/1498765.1498785 (https://doi.org/10.1145/1498765.1498785)