ECE-GY 9143

Introduction to High Performance Machine Learning

Lecture 2 02/05/22

Parijat Dube

ML Performance Optimization

Summary

- Problem definition
- System vs. Algorithmic view
- Performance Optimization Methodology:
  - Measurement
  - Analysis
  - Optimization

System vs. Algorithmic view

ML Performance Factors

20% of the course

Algorithm Performance
- Training vs. Inference Performance
- Training Algorithms Ex: Asynchronous vs. Synchronous Distributed SGD

Hyperparameters Performance
- ex: Learning rate, Network dimensions, Batch size, etc.

80% of the course

Implementation Performance
- Algorithms Implementation (vectorization, comm./comp. overlapping, parallelization, data access)

Framework Performance
- ML Frameworks: PyTorch, TensorFlow, Caffe, MXNET

Library Performance
- Math libraries (cuDNN), Communication Libraries (MPI, GLU)

Hardware Performance
- CPU, DRAM, GPU, HBM, Tensor Units, Disk/Filesystem, Network

System view: Bottom-Up (this course)

Algorithmic view: Top-down

A couple of examples

- Implementation Performance:
  - too many mallocs() in C (or *new* in C++): easily 10 – 100x slowdown

- Algorithmic Performance:
  - Search 1 element in 10 billion stored in an array
    - Linear search: O(n) – average: about 5 billions comparisons expected (*)
    - Binary search: O(log n) – average: about 32 comparisons expected (*)

(*) Assuming exactly one matching element exists and elements are uniformly distributed

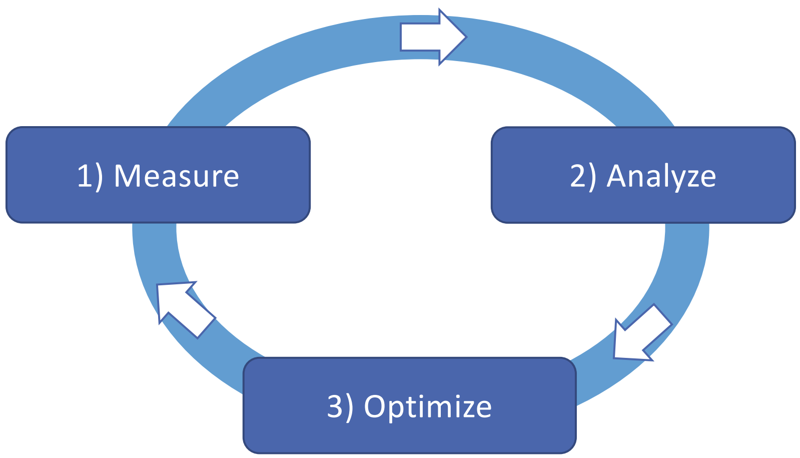
Software Performance Optimization

ML Performance Optimization Definition

- Software Performance Optimization for ML
  - Given:
    - A **system** (ex: NYU Compute node + PyTorch)
    - An **algorithm** (ex: Distributed SGD training) + **hyperparameters**
    - A **dataset** (ex: CIFAR100)
  - Obtain the **maximum** performance

Performance Optimization Methodology

- Execute workload
- Profiling
- Tracing
- Time measuring



- Understand hardware
- Understand software stack
- Understand data-movement
- Identify Critical Path
- Identify Bottleneck

- Implement code optimizations
- Change software configurations and parameters

Performance optimization methodology (1): Measurement

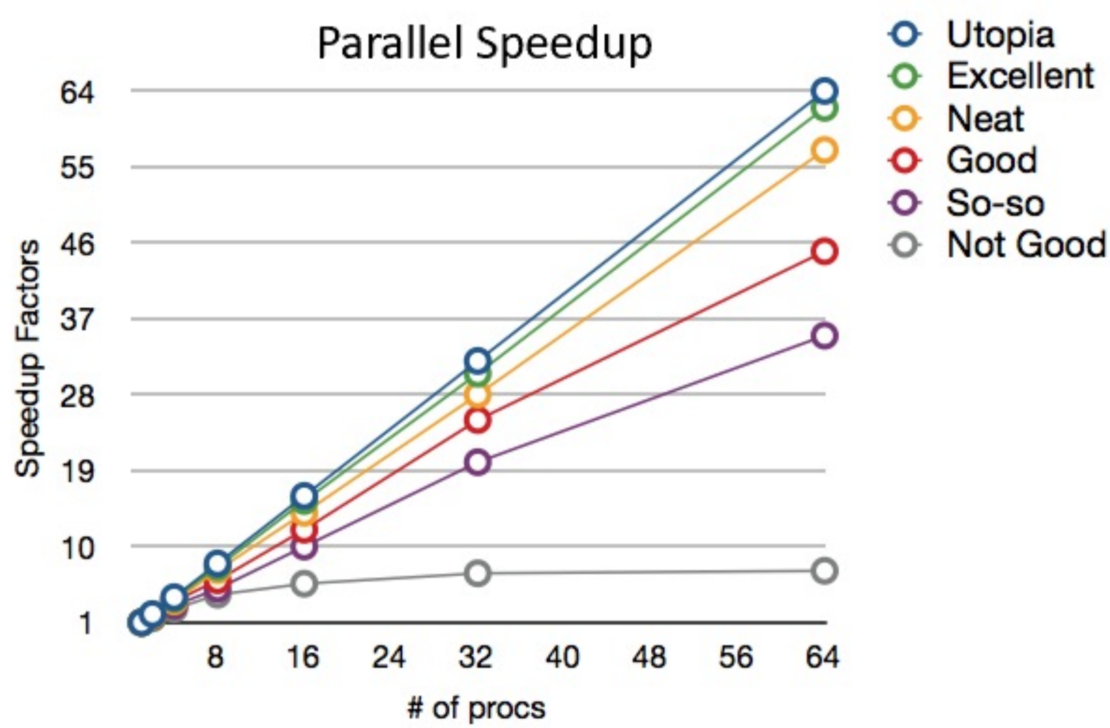
What is performance?

- Basic metrics:
  - Execution time: t (for a single operation is called **latency**)

- Derived metrics:
  - Throughput: $\frac{\#\, operations}{t}$ or $\frac{\#\, programs}{t}$
  - FLOPS: $\frac{\#\, floating\_point\_operations}{t}$ (https://en.wikipedia.org/wiki/FLOPS)

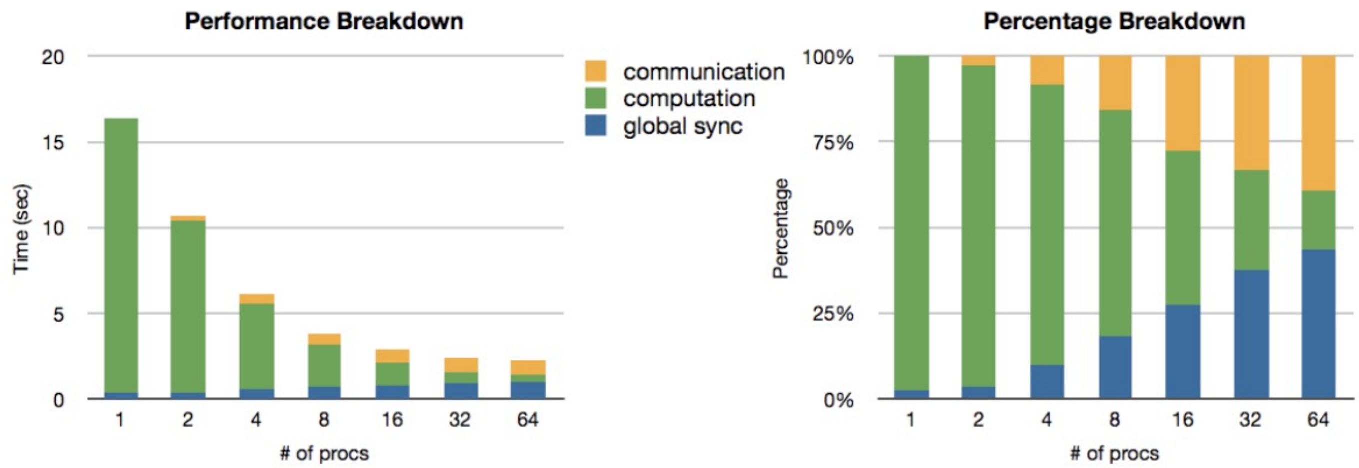
Speedup

- Speedup of B w.r.t. A: $\frac{t_A}{t_B}$
- Parallel Speedup: $\frac{t_{serial}}{t_{parallel}}$
- Slowdown is inverse of Speedup



from: http://web.eecs.utk.edu/~huangj/hpc/hpc_intro.php

“Not Good” speedup



Scalability

- **Scaling Efficiency**
  - E = $\frac{t_{serial}}{t_{parallel} * p}$ <=1

p is the number of processes/threads/…

- **Strong Scaling**: Constant problem size while increasing p
  - Increasing synchronization cost, but fixed amount of work

- **Weak Scaling**: Increasing problem size proportional to p
  - Work per process is constant
  - Increasing synchronization cost, increasing work

What Scaling?

- When my problem continues to increase in size, I can still solve the problem within the same amount of time by simply dedicating proportionally more resources at it.

- When my problem stays at the same size, I can solve the problem 10 times faster by dedicating 10 times more resources.

Computing Averages

- Average Execution Time
  - Arithmetic mean: $\frac{1}{n}\sum_{i=1}^{n} t_i$
- Average Performance or Throughput
  - If t is held constant => Arithmetic mean
  - If *#operations* is held constant => Harmonic mean: $\frac{n}{\sum_{i=1}^{n} \frac{t_i}{\#operations}}$
- Average Speedup, Slowdown or any Ratio
  - Geometric mean: $\sqrt[n]{\prod_{i=1}^{n} speedup_i}$

Benchmarking Workloads

- Benchmarks in ascending order of complexity:
  1. Micro-kernels: test a specific processor feature

     Examples: Floating point, L1 Cache, L2 Cache,
  2. Micro-benchmark: small program from a programming assignment

     Examples: Merge sort in isolation
  3. Kernels: a specific algorithm in a real program

     Examples: Quicksort, Binary Search, DGEMM, DAXPY with context
  4. Synthetic Benchmarks: try to reproduce the workload of a class of applications

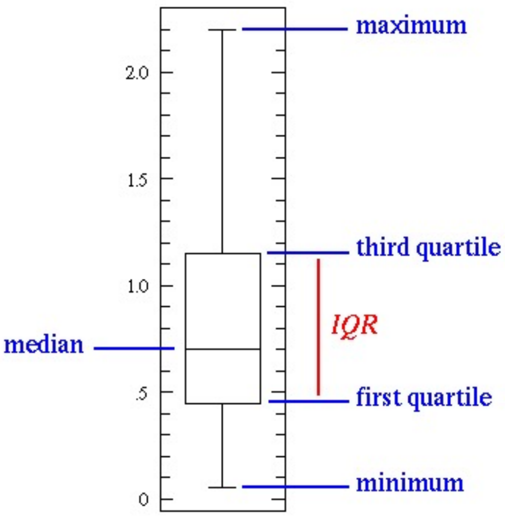
     Examples: Dhrystone, Linpack
  5. Real Applications: a real application used for a specific purpose

     Examples: Word, MySQL, NAMD (Molecular Dynamics)
  6. Real Workflows: a set of applications working together

     Example: CANDLE workflow

Measuring and Reporting Performance

- Reproducibility
  - Always include absolute execution time
  - Report relevant hardware and software info:
    - CPU, Memory, Network, Disk, etc.
    - Experiment configuration
    - Code, Pseudo code
    - Compiler ver., Compilation Flags, Libraries ver., OS ver.
- Accuracy
  - Repetitions: 5, 10, 100, … (depends on variability)
  - If high-variability results:
    - Try to understand why and reduce it
    - Include stddev, variance, max-min, inter-quartile range
    - Use box-plot for chart representation as shown in figure

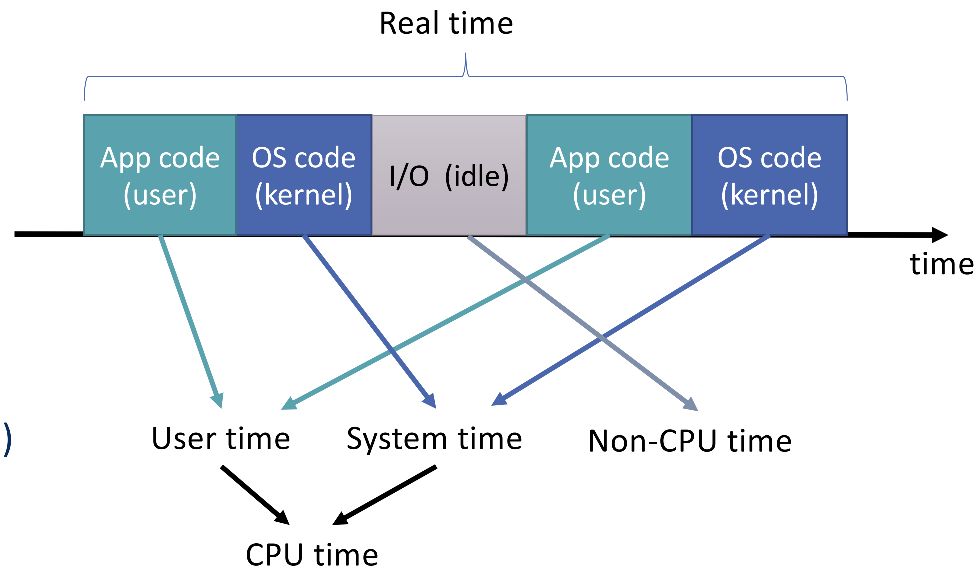


Basic and advanced measurement techniques

- Basic:
  - Time measurement
  - Application Throughput
  - Breakdown phases or iterations
- Advanced:
  - Profiling
  - Tracing

Time definitions

- **Real (or Wall Clock or Elapsed) Time** : actual elapsed time from a point in the past
- **CPU (or Process) Time**: time spent executing CPU instructions
  - **User Time** : time spent in user space
  - **System Time** : time spent in kernel space (OS)
- **Non-CPU Time**: time spent waiting (idle CPU) for: I/O, Virtualization, etc.



https://en.wikipedia.org/wiki/CPU_time

Time Measurement - Linux

- *time* command - Real, User and System times

```
$ time ./executable

real    0m1.057s
user    0m1.015s
sys     0m0.000s
```

- millisecond granularity, accuracy may vary between systems!
- real >= user + sys

Time measurement in C

- clock_gettime(CLOCK_MONOTONIC,..) - Real time
  - Nanosecond granularity - measuring in usec:

```
#include <time.h>
struct timespec start, end;

clock_gettime(CLOCK_MONOTONIC,&start);
<CODE TO MEASURE>
clock_gettime(CLOCK_MONOTONIC,&end);

double time_usec=(((double)end.tv_sec *1000000 + (double)end.tv_nsec/1000)
     - ((double)start.tv_sec *1000000 + (double)start.tv_nsec/1000));
printf("a=%d time: %.03lf\n", a, time_usec);
```

- http://btorpey.github.io/blog/2014/02/18/clock-sources-in-linux/

Execution Time measurement in Python

- Real Time:
    - granularity fractions of seconds – printing in seconds (Python 3.3)

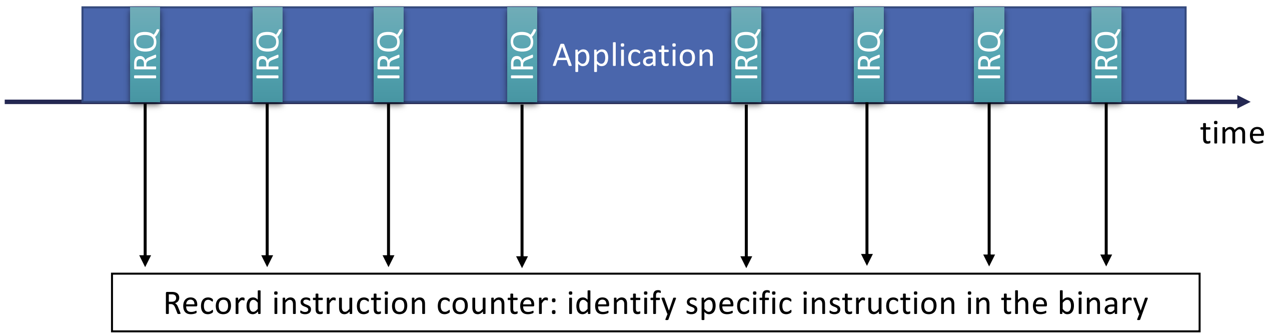
```
import time

start=time.monotonic()
<CODE TO MEASURE>
end=time.monotonic()

print("time: " + str(end-start))
```

- From Python 3.7: *time.monotonic_ns()* (granularity in nanoseconds)
- https://docs.python.org/3.7/library/time.html

Profiling

- Sampling:
  - Sample applications during execution to infer a statistical distribution: Example: approximate time spent in each instruction of the code
- Counting:
  - Count exact events
  - Software counters (implemented in kernel): count specific events
    - Example: count number of memory allocations (malloc())
  - **Hardware performance counters (aka Performance Counters)**
    - Counters maintained in registers
    - Examples: count number of L2 misses, Floating-point ops, Integer ops, number of branch mispredictions

Profiling - Sampling



- IRQ (interrupt request): interruption of the application to execute a different routine
- Profiling uses IRQs to register instruction counter and other metrics at regular intervals
- Relatively low-overhead, depending on IRQ frequency

Profiling - Sampling

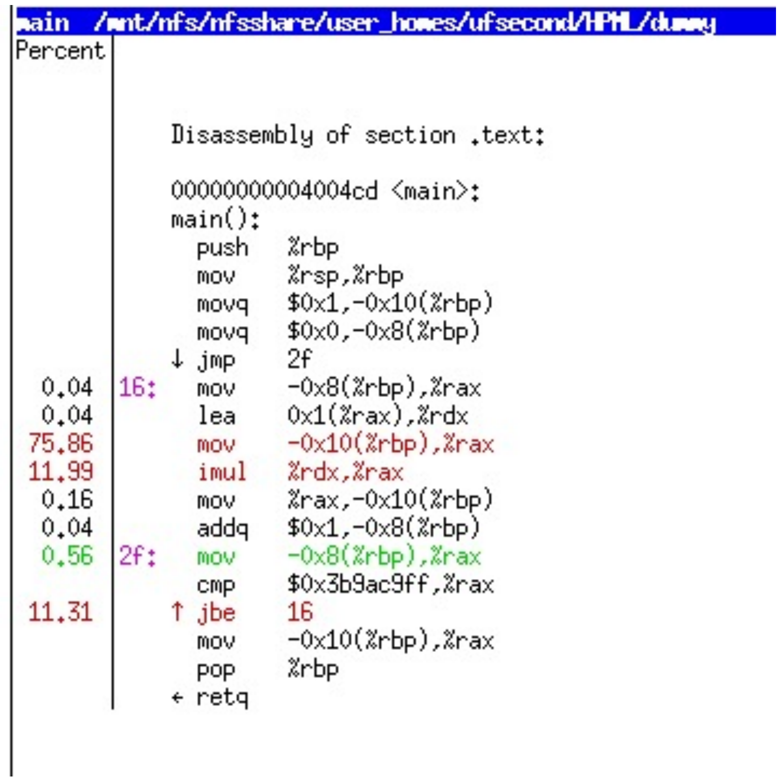
```
int
main() {
  long i,a=1;
  for ( i=0; i<1000000; i++)
    a += a*i;
  return a;
}
```

```
main  /mnt/nfs/nfsshare/user_homes/ufsecond/HPML/dummy
Percent│
       │
       │
       │      Disassembly of section .text:
       │
       │      00000000004004cd <main>:
       │      main():
       │        push   %rbp
       │        mov    %rsp,%rbp
       │        movq   $0x1,-0x10(%rbp)
       │        movq   $0x0,-0x8(%rbp)
       │      ↓ jmp    2f
       │16:     mov    -0x8(%rbp),%rax
       │        lea    0x1(%rax),%rdx
 20.00 │        mov    -0x10(%rbp),%rax
 80.00 │        imul   %rdx,%rax
       │        mov    %rax,-0x10(%rbp)
       │        addq   $0x1,-0x8(%rbp)
       │2f:     cmpq   $0xf423f,-0x8(%rbp)
       │      ↑ jle    16
       │        mov    -0x10(%rbp),%rax
       │        pop    %rbp
       │      ← retq
```

- Example of Linux `perf annotate`
  - Annotated code showing time percentage
  - All the time associated with only 2 instructions ?
  - https://perf.wiki.kernel.org/index.php/Tutorial

Profiling – Sampling 2

```
int
main() {
  long i,a=1;
  for ( i=0; i<1000000000UL; i++)
    a += a*i;
  return a;
}
```



- Linux `perf annotate`
  - Annotated code showing time percentage
  - More samples => more realistic time association
  - https://perf.wiki.kernel.org/index.php/Tutorial

Profiling Call Trees

```
extern int fa(unsigned size) {
    unsigned j,tmp=0;
    for (j=0;j<size;j++) {
       tmp+=j; tmp = tmp%5555555;
    }
}
extern int fsmall(unsigned size) {
    return fa(size);
}
extern int flarge(unsigned size) {
    return fa(size);
}
int main(void) {
    unsigned j, tmp;
    for (j=0;j<1000;j++) {
      tmp += fsmall(10);
      tmp += flarge(1000000);
    }
    return tmp;
}
```

Gprof, RHEL7.6

index	%time	self	children	called	name
[1]	100.0	0.00	65.56		main[1]
		0.00	32.78	1000/1000	fsmall[4]
		0.00	32.78	1000/1000	flarge[3]

- Gprof only samples the last stack entry
- Assembles call chains incrementally
- Assumes all calls to the same function F take the same time to derive call tree annotation!

https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_5.html

Tracing

```
int main(int argc, char **argv) {
    RECORD_TRACE_EVENT("after_main");
    struct timespec start, end;
    int i,a=1;
    clock_gettime(CLOCK_MONOTONIC,&start);
    RECORD_TRACE_EVENT("before_loop");
    for ( i=0; i < 1000000000; i++) {
       RECORD_TRACE_EVENT("in_loop");
       a += a*i;
    }
    RECORD_TRACE_EVENT("after_loop");
    clock_gettime(CLOCK_MONOTONIC,&end);
    RECORD_TRACE_EVENT("before_return");
    return 0;
}
```

TRACE Example

usec	**event**
[000012]	after_main
[000013]	before_loop
[000021]	in_loop
[000024]	in_loop
…	…
[012122]	in_loop
[012132]	after_loop
[012223]	before_return

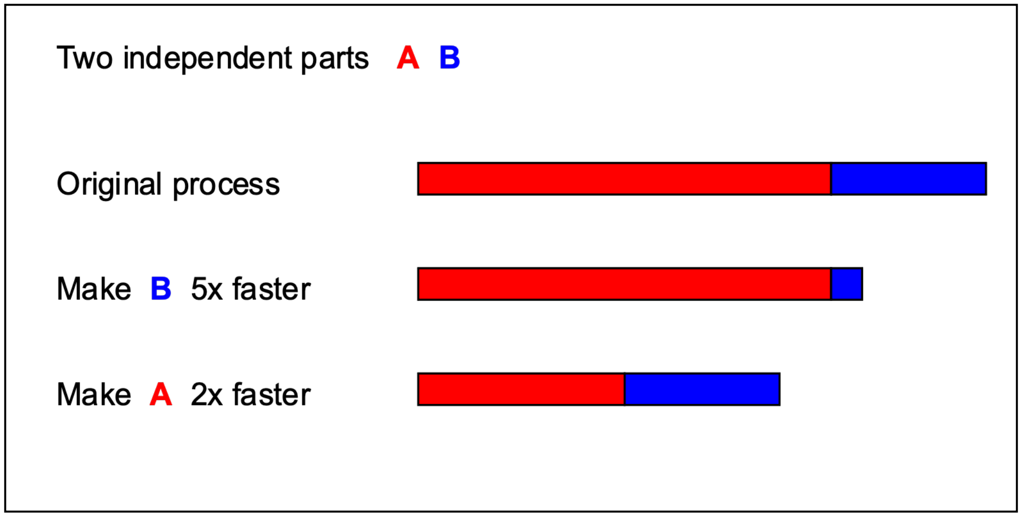
- Explicit code instrumentation with tracing primitives
- Higher overhead than profiling
- Linux perf tracing can be applied to any code: Applications, Runtime, Kernel, Etc.
- Tracing utilities: strace (trace system calls made by an application), ftrace (trace execution flow of kernel functions)

Performance optimization methodology (2): Analysis

Amdahl’s Law

- $S(p, s) = \frac{1}{(1-p)+p/s}$
  - *S*: speedup of the entire application (or runtime, OS, etc.)
  - *p*: portion of the execution time that is spent in the code section before improvement (if time for *p* is high the section is called **critical section**)
  - *s*: speedup of the improved code section
- Overall speedup is limited by how much time the improved code takes compared to the rest

Amdhal’s law effect example



From: https://en.wikipedia.org/wiki/Amdahl%27s_law

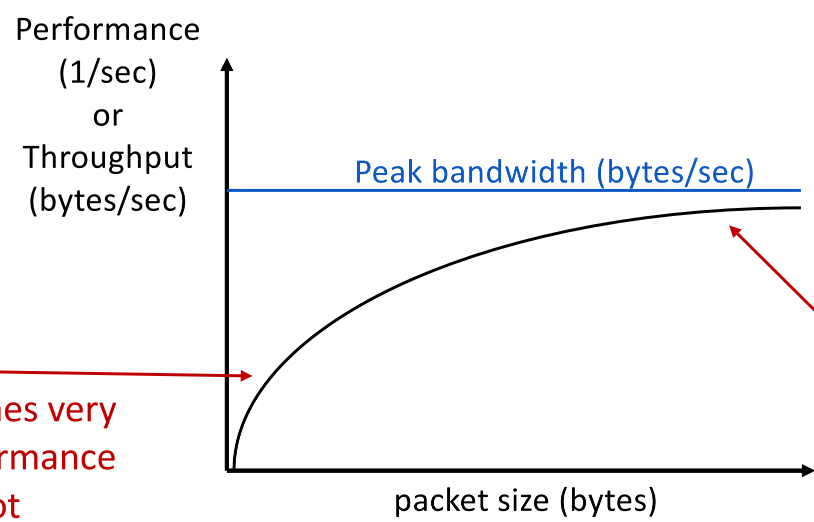
Performance Analysis Step

1. Identify **Critical Path**: the section of the program that accounts for most of the time (high value of Amdahl's *p*)
   - Critical Path characteristics: very slow to execute (high latency) and/or executed many times
   - Use output of performance measurement step (profiling, tracing, etc.)
   - Verify hypothesis of critical path: comment code and run again

2. Identify the **Bottleneck**: the **system resource** that affects the execution time of the critical path
   - Need to understand software/hardware architecture
   - Bottleneck type: **Data Movement** vs. **Computation**

Data Movement and Packet Size

- True for any **Data Movement**: Network, PCIe, DRAM, etc.

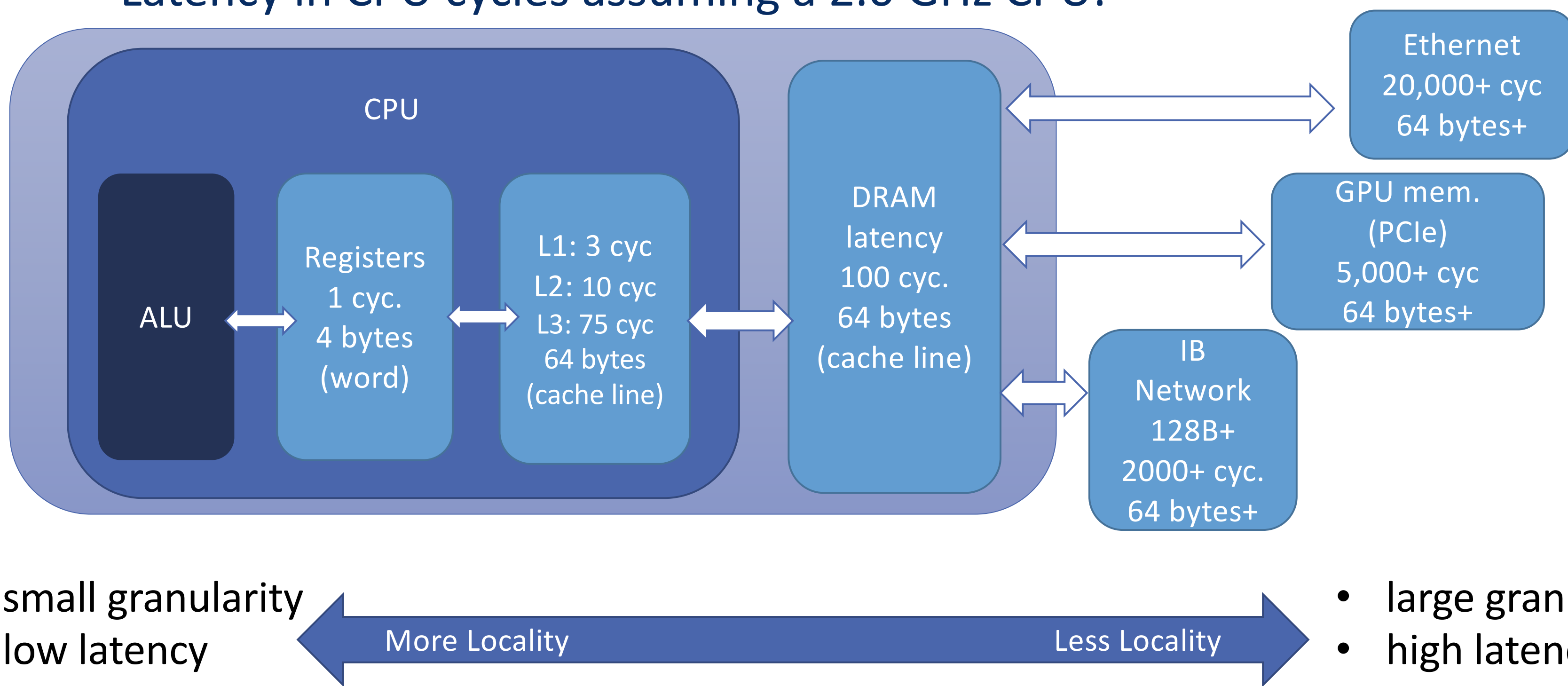


- SMALL Packets:
  - **Low Latency** becomes very important for performance
  - Full Bandwidth is not reached

- LARGE Packets:
  - **High Bandwidth** becomes very important for performance
  - Latency is not relevant

Data movement Locality Principle - Latency

- Latency in CPU cycles assuming a 2.0 GHz CPU:

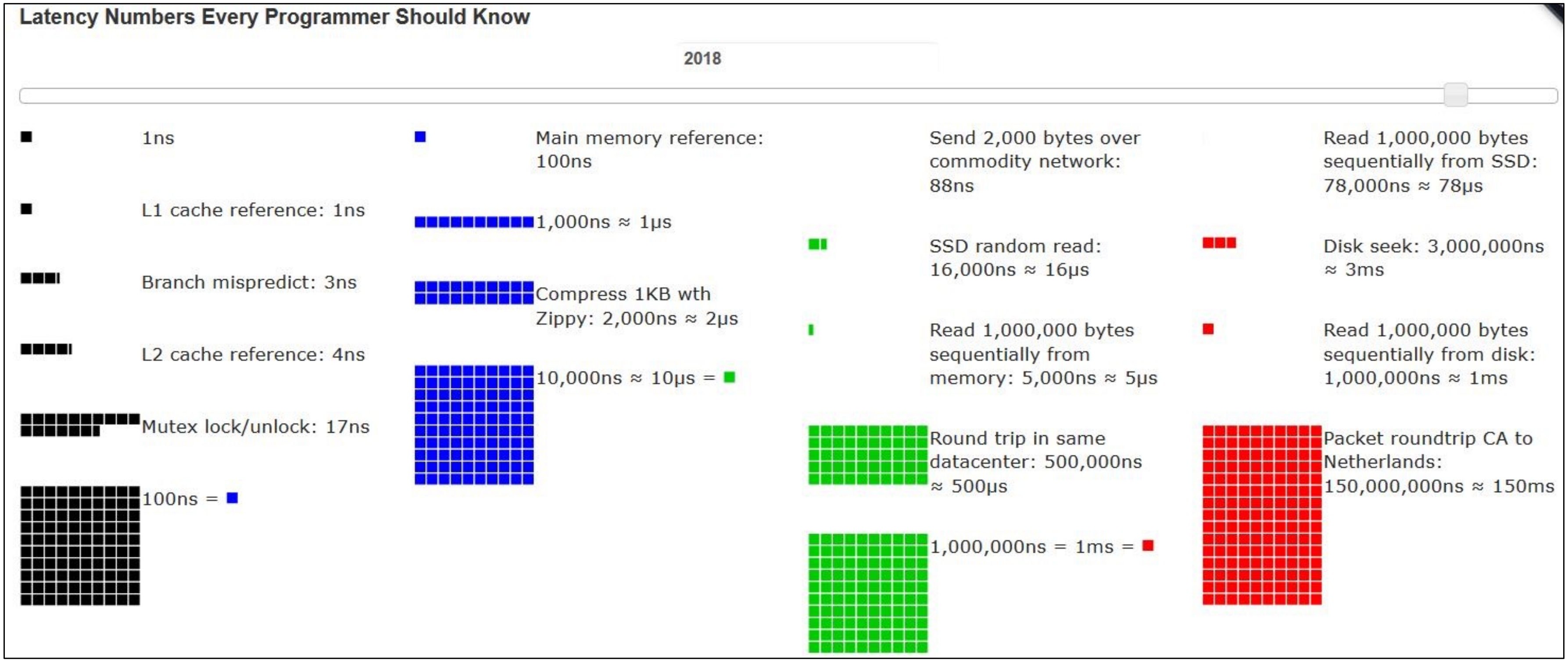


- small granularity
- low latency
- high bandwidth

- large granularity
- high latency
- low bandwidth

Latencies every programmer should know: link

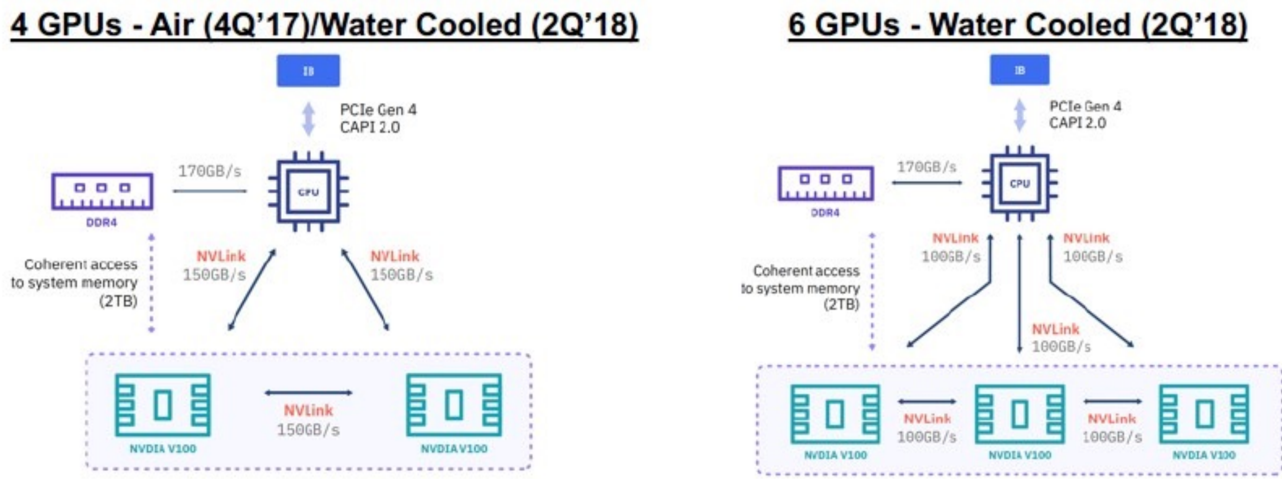
Latency values over the years – very cool tool!



- https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Data Movement Locality Principle - Bandwidth

- IBM POWER9 + NVIDIA Volta GPU



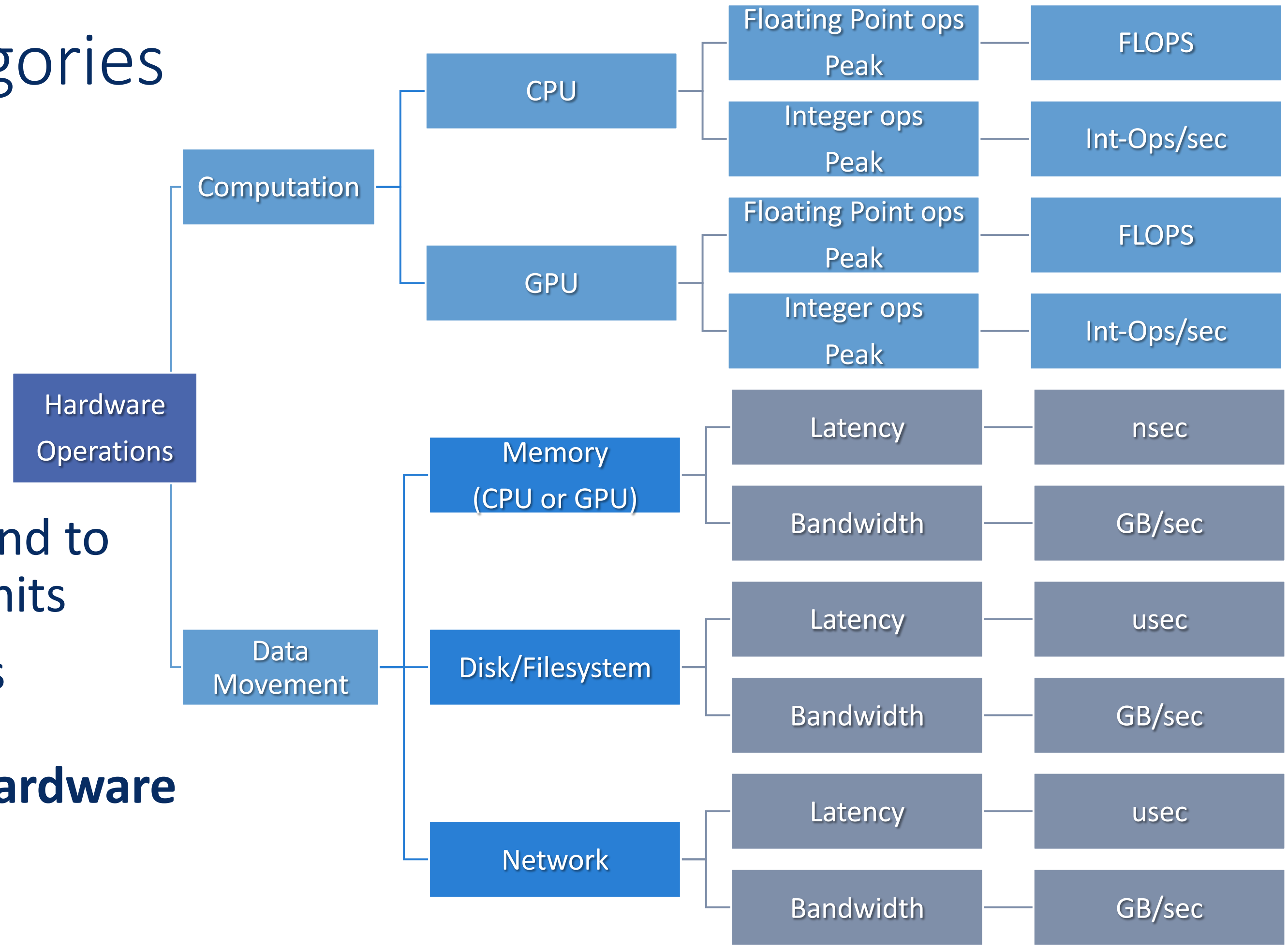
- Up to 4 GPUs, air/water cooled options
- 150GB/s of bandwidth from CPU-GPU

- Up to 6 GPUs, water cooled only
- 100 GB/s of bandwidth from CPU-GPU

- Coherent access to system memory
- PCIe Gen 4 and CAPI 2.0 to InfiniBand
- Water cooled options available in 2Q'18

- (Bi-directional bandwidth)

Bottleneck categories



- Bottlenecks correspond to specific Hardware Limits
- Performance Analysis Problem: **How far is performance from Hardware Limits?**

Performance Models Objectives

- Identify performance bottlenecks
- Determines Hardware Limits to Optimization
  - Determines how fare we are from hardware limits
  - Motivate algorithmic changes
- Project performance on future hardware or applications

Peak FLOPS

- Peak FLOPS depend on:
  - Compute unit architecture: CPU, GPU, TPU, FPGA etc.
  - #cores and #threads
  - Clock Frequency
  - Precision: DP (64 bit), SP (32 bit), HP (16 bit)
  - SIMD instructions in the cores: Intel AVX, IBM Altivec
- CPU formula for Peak FLOPS: $\#tot\_cores \cdot \frac{cycles}{seconds} \frac{FLOPs}{cycles}$
- see https://en.wikipedia.org/wiki/FLOPS

Performance Model – Constants (HW specs)

- Examples of HW Specs for the performance model
  - **CPU peak DP/SP FLOPS**: GFLOPS/s
  - **DRAM peak Bandwidth**: GB/s
  - **GPU peak DP FLOPS**: TFLOPS/s
  - **HBM peak Bandwidth**: TB/s
- How to obtain:
  - Vendor hardware specifications
  - Alternative: run micro-benchmarks for compute and memory (lower bounds than specs)

Performance Model - Variables

- Actual Experimental Measurements :
  - Computation (CPU/GPU) Performance: FLOPS
  - Memory throughput: GB/s or TB/s
- How to measure:
  - FLOPS: **hw performance counters** for FLOP divided by time
  - GB/s: **hw performance counters** for memory-ops divided by time

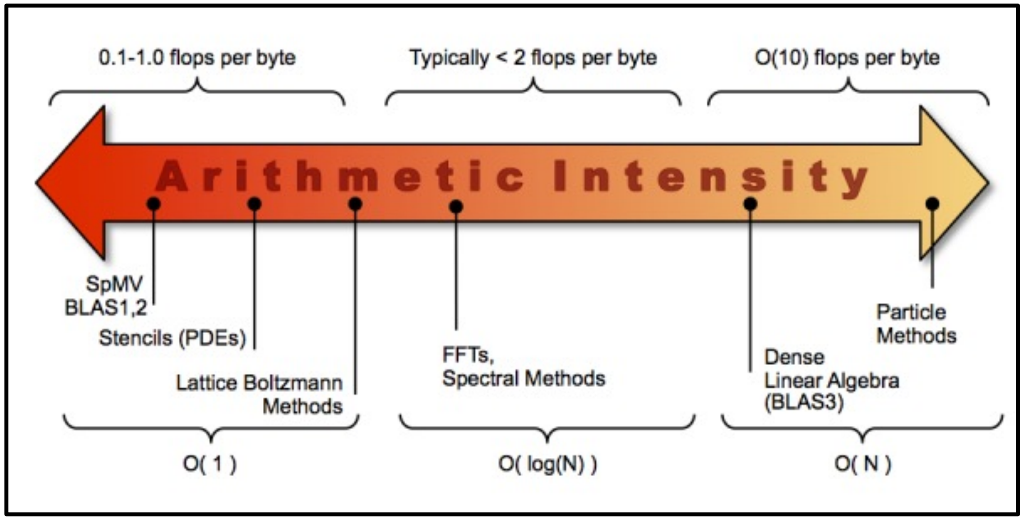
Roofline Performance Model (1)

- Throughput-based model
- Developed at
  DOE Lawrence Berkeley Labs
- Metrics:
  - Peak FLOPS
  - Memory Bandwidth: $\frac{data}{time}\left[\frac{bytes}{sec}\right]$
  - Arithmetic Intensity (program property):

$$\frac{\#arithmetic\ ops}{DRAM\ data}\left[\frac{FLOP}{bytes}\right]$$

(bytes as seen from DRAM)

note: FLOP ≠ FLOPS



https://crd.lbl.gov/departments/computer-science/PAR/research/roofline

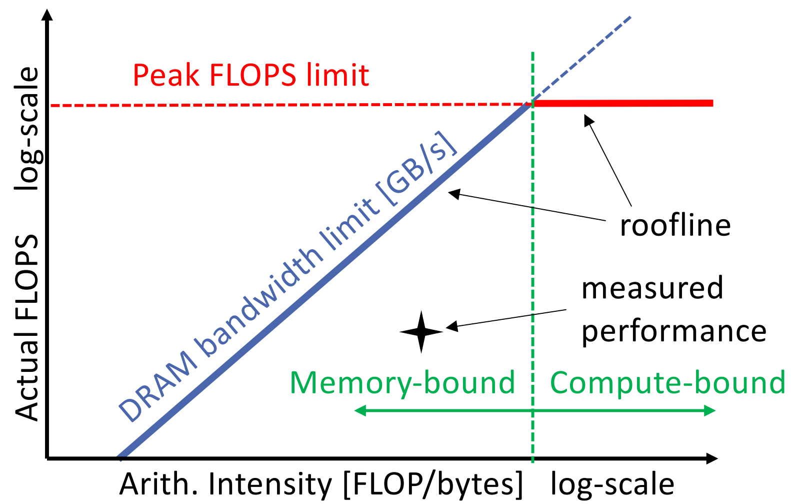
Arithmetic Intensity

- The ratio between the number of executed operations and the number of bytes transferred between the CPU and the memory is called arithmetic intensity.

- Smaller arithmetic intensity means a larger pressure on the memory subsystem, and conversely, larger arithmetic intensity means a larger pressure on the CPUs computational resources.

Roofline Performance Model

- Actual FLOPS are limited first by DRAM bandwidth (memory-bound) and then by CPU (or GPU) Peak FLOPS (compute-bound)
- Actual measured performance is below the roofline
- Depending on Arithmetic Intensity:
  - **memory-bound** code
  - **compute-bound** code



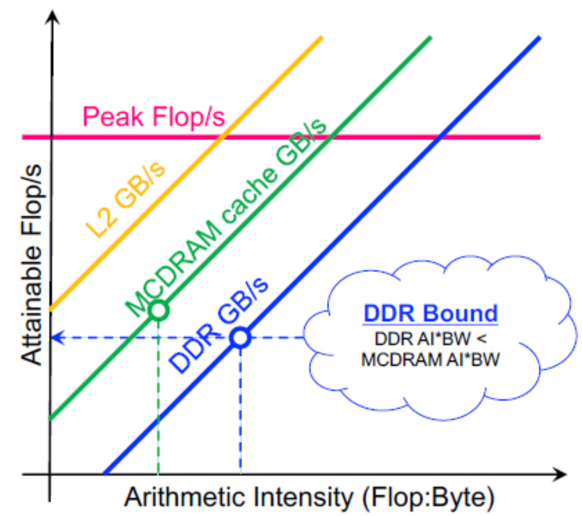
- **Log-log scale is used for clarity**

More complex Roofline Models

- We considered a basic **DRAM-only Roofline Model:**
  - Bytes as seen from DRAM access

- Not covered: Hierarchical Roof-line
  - for problems that fit in the cache we can add:
    - L1, L2, L3 bandwidth
  - Each Cache Level has its own A.I. (different bytes going through that level of the mem. Hierarchy)

- Also not covered: Cache-aware Roof-line
  - FLOP/bytes as seen from CORE
  - Different roof but same A.I.
  - Need to know from which level data is coming
  - http://www.inesc-id.pt/ficheiros/publicacoes/9068.pdf



Hierarchical Roofline
from: LBNL (SC17 Roofline Model Workshop slides)

crackle1.cims.nyu.edu compute node @NYU

- Intel Xeon E5630@2.53GHz performance:
  1. #cores: 4
  2. LLC (L3) size: 12MB
  3. Clock frequency: 2.53GHz
  4. **DRAM peak bandwidth**: 25.6 GB/s
  5. **CPU Peak FLOPS**: 81.3 DP GFLOPS – 162.56 SP GFLOPS

- DRAM peak bandwidth:
  - https://ark.intel.com/products/47924/Intel-Xeon-Processor-E5630-12M-Cache-2_53-GHz-5_86-GTs-Intel-QPI

- CPU peak FLOPS :
  - FLOPS = frequency * total_cores * FLOPS/cyc
  - https://en.wikipedia.org/wiki/FLOPS (architectures list - this is a *Sandy Bridge*)

```
$ ssh username@access.cims.nyu.edu
$ ssh crackle1.cims.nyu.edu
$ cat /proc/cpuinfo
```

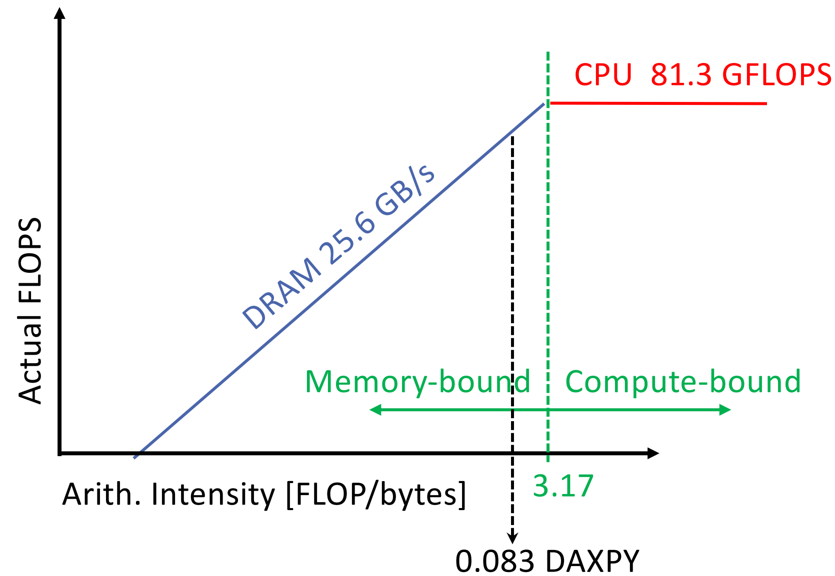
```
processor	: 0
vendor_id	: GenuineIntel
cpu family	: 6
model		: 44
model name	: Intel(R) Xeon(R) CPU E5630  @ 2.53GHz
stepping	: 2
microcode	: 0x15
cpu MHz		: 2527.014
cache size	: 12288 KB
physical id	: 0
siblings	: 8
core id		: 10
cpu cores	: 4
```

Roofline Model Example – crackle1

- *crackle1*:
  - CPU peak: 81.3 DP GFLOPS
  - DRAM peak BW: 25.6 GB/s
- DAXPY code:

```
for (i=0;i<N;i++) {
 Z[i]= A * (X[i] + Y[i])
}
```

- Y,A,X are 64 bit float (DP)
- DRAM and CPU cross at:
  - *81.3 GFLOPS /25.6 GB/s = 3.17 FLOP/byte*
  - CPU_peak/DRAM_BW
    - Where DP-bytes=8, SP-bytes=4, HP-bytes=2

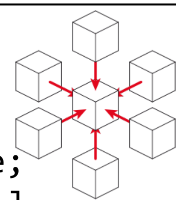


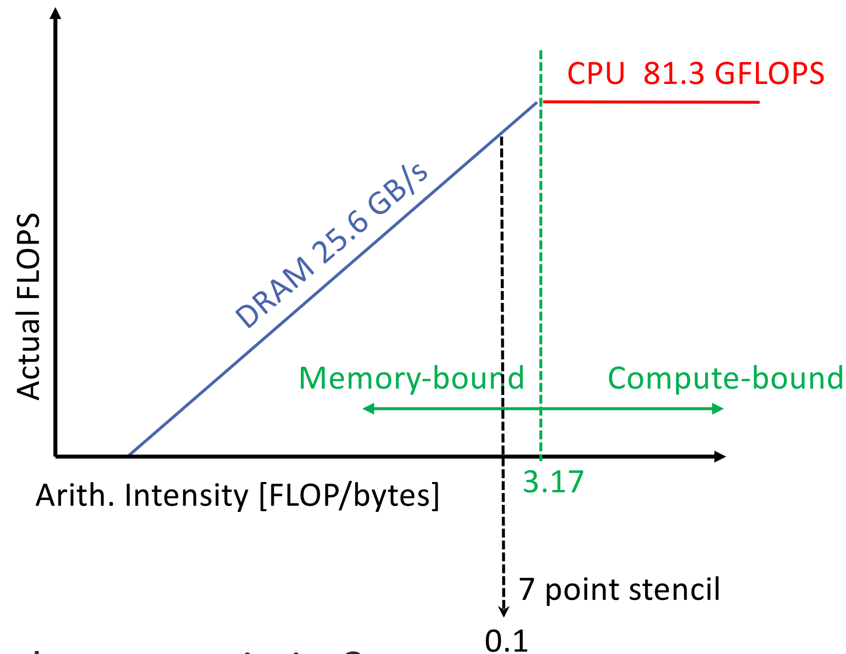
- A.I. = 2 FLOP/(3*8) bytes = 0.083 FLOP/byte
- Result: *0.083* < *3.17* => **Memory-bound** => how far for DRAM BW?

Roofline Model Example (2) – crackle1

- 7-Points Stencil code:

```
for(k=1;k<N;k++){
for(j=1;j<N;j++){
for(i=1;i<N;i++){
 int ijk = i+j*jStride+k*kStride;
 new[ijk] =-6.0*old[ijk         ]
               +old[ijk-1       ]
               +old[ijk+1       ]
               +old[ijk-jStride ]
               +old[ijk+jStride ]
               +old[ijk-kStride ]
               +old[ijk+kStride ];}}}
```

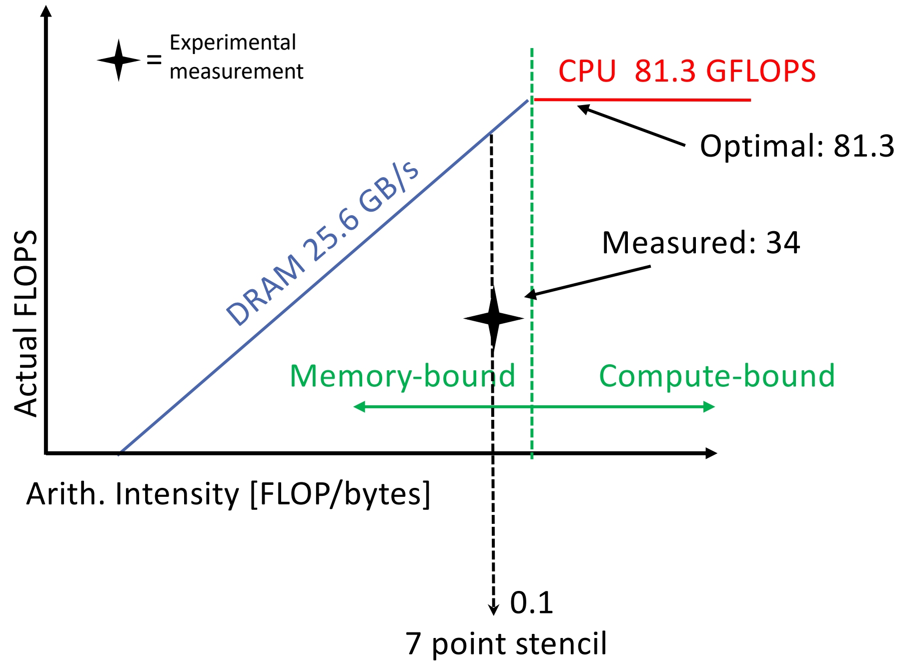




- 7 DP flops (new[] is 64bits)
- 8 memory references
- DRAM/CPU cross = 3.17 FLOP/byte
- AI = 7 FLOP/(8*8) bytes = 0.109 FLOP/byte
- Result: *0.109* < *3.17* => still **Memory Bound** => how to optimize?

Next steps - Optimization

1. Know the limitation from the model: CPU vs. DRAM
2. Measure actual performance:
   Example: 34 GFLOPS
3. Optimize to get close to max FLOPS! (81.3 GFLOPS)



Performance optimization methodology (3): Optimization

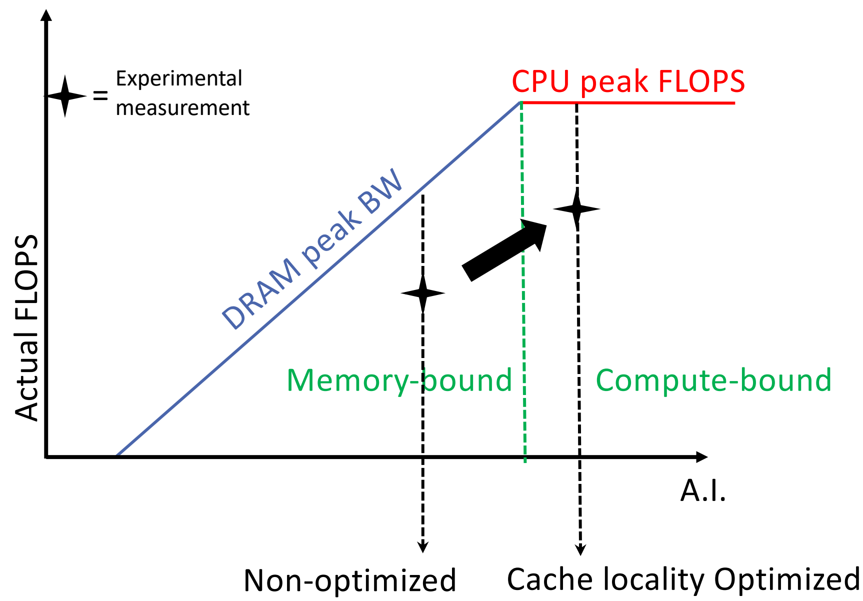
Two ways to Performance

- Reduce latency (do one operation faster):
  - Data access latency reduction

- Increase Parallelism (do more operations at the same time):
  - Vectorization
  - Instruction Level Parallelism
  - Thread Level Parallelism
  - Multi-core design
  - Computer Clusters

Optimization Example: Cache Blocking

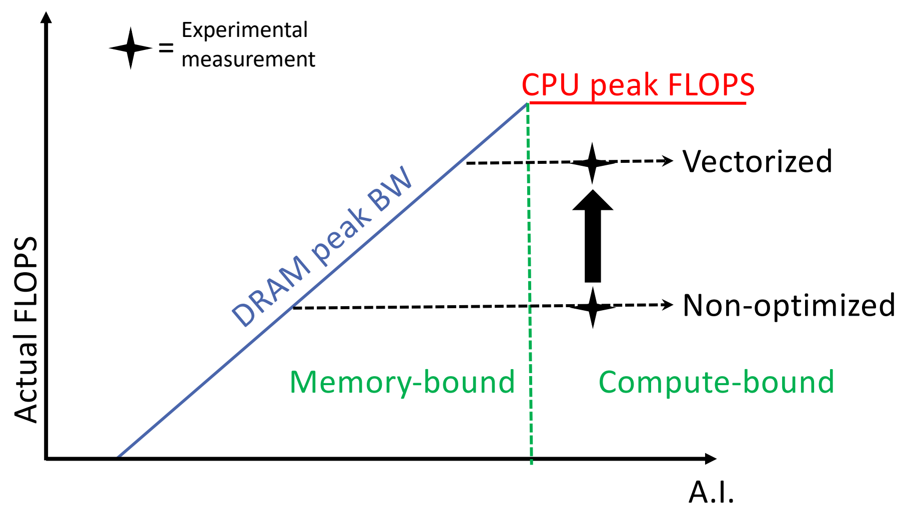
- Observation: Cache access latency is about 10x lower than DRAM and BW is much higher
- Optimization (cache blocking):
  - Divide program data structures in blocks of the **cache size**
  - Work on each block before switching to the next
  - Less DRAM bytes: **cache is filtering DRAM accesses**
  - A.I. [FLOPS/(DRAM bytes)] is higher
- Result:
  - Bottleneck moves: Code (may) become compute-bound with higher FLOPS!



https://www.intel.com/content/www/us/en/developer/articles/technical/cache-blocking-techniques.html

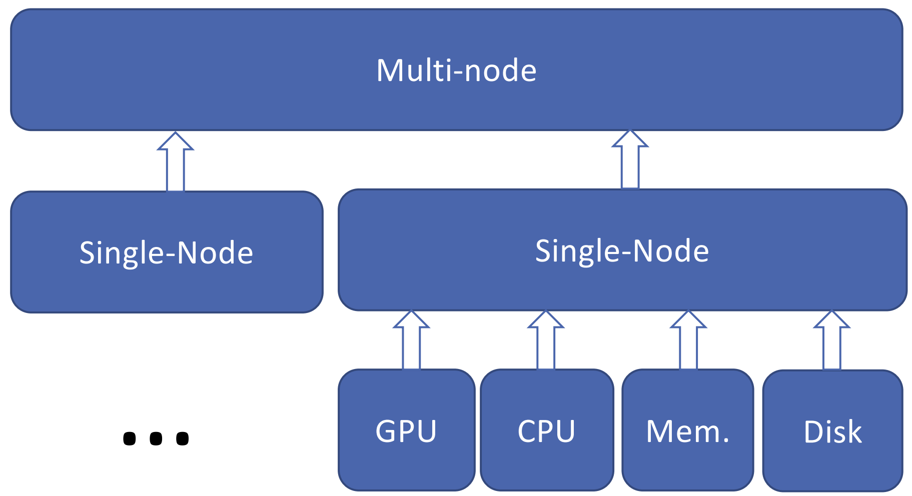
Optimization Example: Vectorization

- Observation: SIMD instructions execute multiple FLOP (2,4,8,16..) with 1 instruction => higher FLOPS
- Optimization (Vectorization):
  - Replace normal code with SIMD instructions
  - **Hint:** use math libraries like BLAS (CPU) or cuDNN (GPU) and they will do it for you!
- Result:
  - Code reaches higher FLOPS!



Hierarchical Perf. Optimization – (next lessons)

- Single Node optimizations:
  - CPU:
    - Vectorize/SIMD optimizations
    - CPU Cache/Memory optimizations
    - Multi-core scalability/parallelism
  - GPU:
    - SM Optimizations
    - SM Cache/Memory optimization
  - Disk and IO
- Multi-node optimizations:
  - Parallelism exposure
  - Domain decomposition
  - Load-balancing
  - Reduce synchronizations
  - Reduce collectives



Performance Optimization Methodology Recap

1) Measure
- 1.1) **Time** various sections of the code
- 1.2) Measure resource usage (**Perf Counters**)

2) Analyze
- 2.1) Identify **Critical Path** based on Time Measurement
- 2.2) Identify **Bottleneck** with **Roofline Model**

3) Optimize
- 3.1) Apply correct **Optimization Technique** based on bottleneck

Floating Point Errors

- Error: $E = |f(x)-F(x)|$
  - $F(x)$ is the correct result, $f(x)$ is the numerically computed result
- Relative error: $R = E/|F(x)|$
  - Floating point 'roundoff' relative error depends on number of bits in the mantissa!
- Cancellation
  - $C = A+B$ → may result in $C==A$ for $B << A$
- Catastrophic cancellation
  - $C = B+A-A$ → may result in 0 for $B<<A$, relative error is 1
  - $C = 1/(B+A-A)$!

Floating Point Error Example

- IEEE standard 754
  - FP32 1 bit sign + 8 bit exp + 23 bit mantissa, bias 127
  - FP64 1 bit sign + 11 bit exp + 52 bit msantissa, bias 1023
  - $(-1)^S * 1.M * 2^{\{E-bias\}}$

$$\frac{1}{3} \cong (-1)^0 * (1.3333333) * 2^{125-127} = 0.333333325$$

$$R = \left(\frac{1}{3} - 0.333333325\right) * 3 \cong 2.5 * 10^{\{-8)}$$

0111 1101b = 125

011 0010 1101 1100 1101 0101b = 3333333

Lesson Key Points

- ML Performance Factors
- Performance Optimization Methodology:
    1. Measurement: Metrics, Time/Resources and Techniques
    2. Analysis: Amdahl's Law, Bandwidth/Latency, Roofline Model
    3. Optimization (in relationship to Roofline model)