Lecture 9 - GP-GPU and High Performances Computing

Performance Evaluation

Previously

Understanding computational decomposition and how to structure parallel problems

Key concepts:

- Task decomposition
- Data decomposition
- Domain decomposition

Computing Performance

Performance is defined by **2 factors**:

1. **Computational requirements** (what needs to be done)
2. **Computing resources** (what it costs to do it)

Computational problems translate to requirements

Factors: hardware, time, energy, money

Why is Performance Important?

- Performance itself is a **measure of how well** the computational requirements can be satisfied
- We evaluate performance to understand the **relationships between requirements and resources**
  - → Decide how to change methodology to target objectives
- Performance measures reflect **decisions** about how and how well approaches are able to satisfy the computational requirements

Definitions of Parallelism

Performance issues when using a parallel computing environment

- **Performance is why we do parallelism**
- Parallel performance versus sequential performance
- If the "performance" is not better, parallelism is not necessary

Parallel processing includes

Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …

Parallelism must deliver performance: How? How well?

What Can We Expect?

- If each processor is rated at **k-GFlops** and there are **p** processors, should we see **k × p GFlops** performance?
- If it takes **100 seconds on 1 processor**, shouldn't it take **10 seconds on 10 processors**?

Several causes affect performance

- Each must be understood separately
- They interact with each other in complex ways
  - Solution to one problem may create another
  - One problem may mask another

Need to understand performance space

Analytical Measures

Embarrassingly Parallel Computation

An embarrassingly parallel computation is one that can be **obviously divided into completely independent parts** that can be executed simultaneously

Types

- **Truly embarrassingly parallel**: No interaction between separate processes
- **Nearly embarrassingly parallel**: Results must be distributed and collected/combined in some way

Potential

- Have potential to achieve **maximal speedup** on parallel platforms
- If it takes T time sequentially, potential to achieve T/P time with P processors
- **What would cause this not to be the case always?**

Scalability

- A program can scale up to use many processors: **What does that mean?**
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation: if double the number of processors, what to expect? Is scalability linear?
- Use **parallel efficiency measure**: is efficiency retained as problem size increases?
- Apply performance metrics

A Measure of Performance

Evaluation

- Sequential runtime T_{seq} is a function of **problem size** and **architecture**
- Parallel runtime T_{par} is a function of **problem size**, **parallel architecture** and **number of processors**
- Parallel performance affected by **algorithm** and **architecture**

> **Definition:** Scalability
>
> Ability of parallel algorithm to achieve performance gains proportional to the **number of processors** and the **size of the problem**

Performance Metrics and Formulas

Metrics

- T_1 = execution time on a single processor
- T_p = execution time on p processors

Cost:

$$Cost(p) = p \times T_p$$

Speedup:

$$S(p) = \frac{T_1}{T_p}$$

Efficiency:

$$E(p) = \frac{S(p)}{p}$$

Cost-optimal: Parallel time = sequential time
$(C(p) = T_1, E(p) = 100\%)$

Performance Metrics

Time Execution Challenges

Optimizing an application of fixed size is **difficult**

Challenges

1. **High parallel management costs** (synchronization and communication)
   - → Low parallel acceleration
2. **High complexity** of parallel approach vs best sequential algorithm
   - → Weak final gain

Time Representation

If parallel algorithm has:

- Decreasing execution time
- Limited overhead on one resource

→ Final gain might be important

But analysis should continue to consider

- Execution time vs Ideal execution time
- Speedup
- Efficiency
- Size-up and scalability

How to Represent Time Execution

It is best to choose a representation to **identify simple shapes**: $N^3, N^2, \log(N)$

For parallel execution, the ideal curve is:

$$T(P) = \frac{T(1)}{P}$$

But hard to identify simply, hence use:

$$\log(T(P)) = \log(T(1)) - \log(P)$$

Or use a log scale on the graph

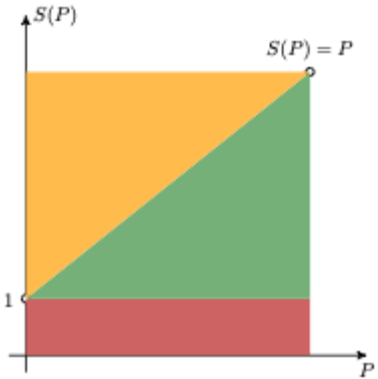
Benefits

- ✓ Easier to detect distance to theory
- ✓ Easier to detect distance to other laws

Speedup

$$S(P) = \frac{T(1)}{T(P)}$$

Typical Speedup Curve



Interpretation

- $S(P) < 1$: **parallelism is erroneous**
- $1 < S(P) < p$: **normal behavior**
- $S(P) > p$: **super-linear speedup**

Super-Linear Speedup

This observation is an anomaly:

- It should be analyzed to provide clear explanation
- A correction should be proposed or an optimization should be operated

Sample explanations

1. **The operations performed are not correct** → result is wrong
2. **Data fit in the total cache** of P processors
3. **The starting algorithm has been modified** → convergence is faster (e.g. optimized genetic algorithm)
4. **Computation is based on tree exploration** and the program is stopped early

Efficiency Metrics

Usage rate of available resources or percentage of usage from perfect speedup

$$e(P) = \frac{S(P)}{P}$$

Interpretation

- $e(P) \in [0, 1]$
- $e(P) > 100\%$: super-linear speedup

How to Choose Sequential Reference?

Algorithm

- Same program on one processor
- Same algorithm than in sequential
- Best sequential algorithm

Compilation

- Sequential compilation with same compiler?
- Best sequential compiler?

Optimizations

- Sequential optimization allowed by parallelism
- Best sequential optimizations

Execution

- On one core of parallel computer?
- On the best sequential computer?

All choices are valid - depends on your objectives and problematic

The choice should be clearly stated

Sequential Reference: Examples

All choices are valid. Each choice depends on:

- A different **point of view**
- A different **concern**
- A different **objective** in the analysis

Examples

- **Final user**: his sequential program on his sequential computer
- **Developer**: his parallel program on one processor of his parallel computer

Sequential Reference Options

The sequential reference may be obtained:

1. **Easy approach:**
   - Same algorithm, same language, same sequential optimization, same processor
   - → Good performances, **easy to get**
2. **Ideal approach:**
   - Best algorithm, best language, best sequential optimization, best processor
   - → Good performances, **very hard to get**

Losing Performances

Under usage of each core

- Sequential sub-optimization
- No possibility to vectorize loops

Under usage of nodes

- Sequential fractionning
- Overhead of synchronization
- Overhead of communication
- Unbalanced between tasks and loads

Weak platform

- Sequential IO
- Interconnection network undersized

Size-Up Metrics

Objective 1: Work on larger problem on more resources

⚠ An application that **replicates** most of the data on all compute units will always be **memory bound**. It won't be able to scale up.

✓ An application that **distributes** most of the data on all compute units will be able to store more data per compute unit. It **will be able to scale up**.

Size-Up Design Principles

> Design an application with an **initial distribution** of data and a **communication scheme** with minimal volume.

Requirements

1. Need to **move initial data** from compute unit to compute unit to allow continuation of computation on data different than its own

2. Need to have **intermediate data move** from compute unit to compute unit to allow continuation of computation from another compute unit, with its own data

Size-Up: Execution Time

Objective 2: Maintain constant execution time

$$T(1 \times n_1, p_1) = T(2 \times n_1, p_2) = T(k \times n_1, p_k) = C$$

Where:

- n_1 is the size of the problem
- k is size scale up
- p_k is the number of compute units needed to solve the problem at constant time

Size-Up: Time and Resources

Objective 3: Maintain constant execution time with the **minimal number of resources**

$$T(1 \times n_1, p_1) = T(2 \times n_1, p_2) = T(k \times n_1, p_k) = C$$

$$\mathcal{O}(p_k) = \mathcal{O}(k \times n_1)$$

Criteria for Scale-Up

Metric 1: Based on T

When the size of the problem increases such that we are **not able to perform sequential execution**:

Limitations

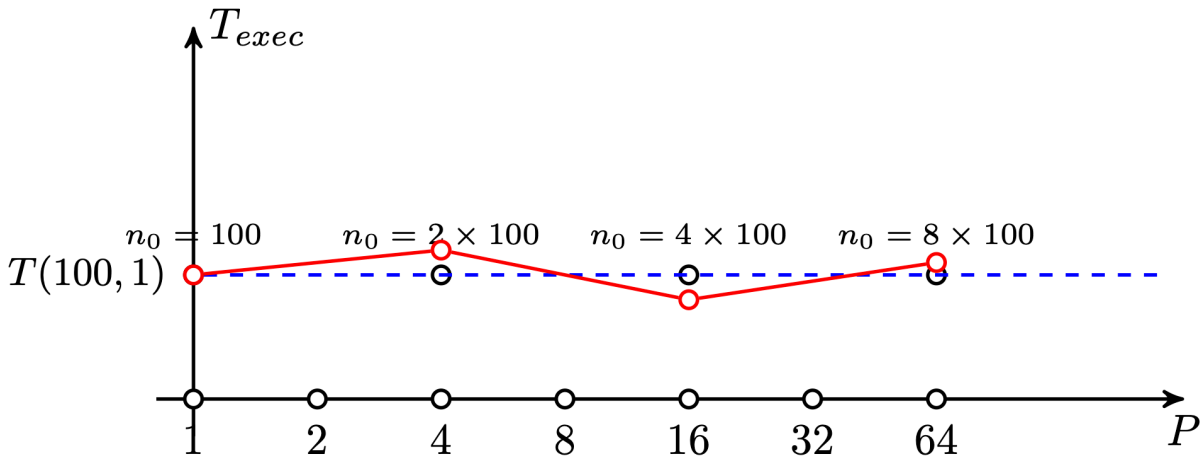
- **Not enough memory**, not enough storage
- **Execution time too high**, compute unit not available

Simple Size-Up

Definition and criteria

- Allow for a size up to deal with larger problem on more compute units
- Beware of energy consumption and other costs

Size up with constant time and minimal compute units for complexity $\mathcal{O}(n^2)$

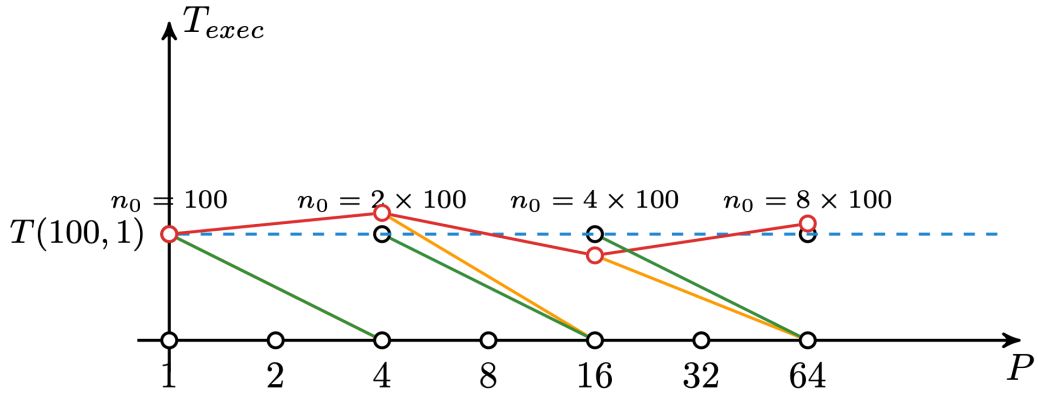


Complete Size-Up

Definition and criteria

- Allow for a size up to deal with larger problem on more compute units
- Beware of energy consumption and other costs
- **Allow for a speedup for all sizes of problem**
- **Keep the same profile of time evolution**

Size up with constant time and minimal compute units for complexity $\mathcal{O}(n^2)$



Plotting Scale-Up

How to build and use a graph for scale up

1. Measure $T(n, p)$ for different size of problem n and number of compute units p
2. Plot $T(n, p)$ using **log scale**

Ideal case: For each size of problem, we obtain a line **parallel to others**

Validation: Comparing measures to expected curves, slope and positions

Use as an abacus: For a given problem size, identify number of compute units to respect a maximum T_{exec}

Scale-Up Checklist

Requirements

- ✓ Meet the needs of computation
- ✓ Require the least possible number of resources
- ✓ Quantify the number of required resources
- ✓ Plan for associated expenses

Laws on Performance

Amdahl's Law

Let f be the fraction of a program that is **sequential**
$1 - f$ is the fraction that can be **parallelized**

Let T_1 be the execution time on 1 processor
Let T_p be the execution time on p processors

S_p is the speedup:

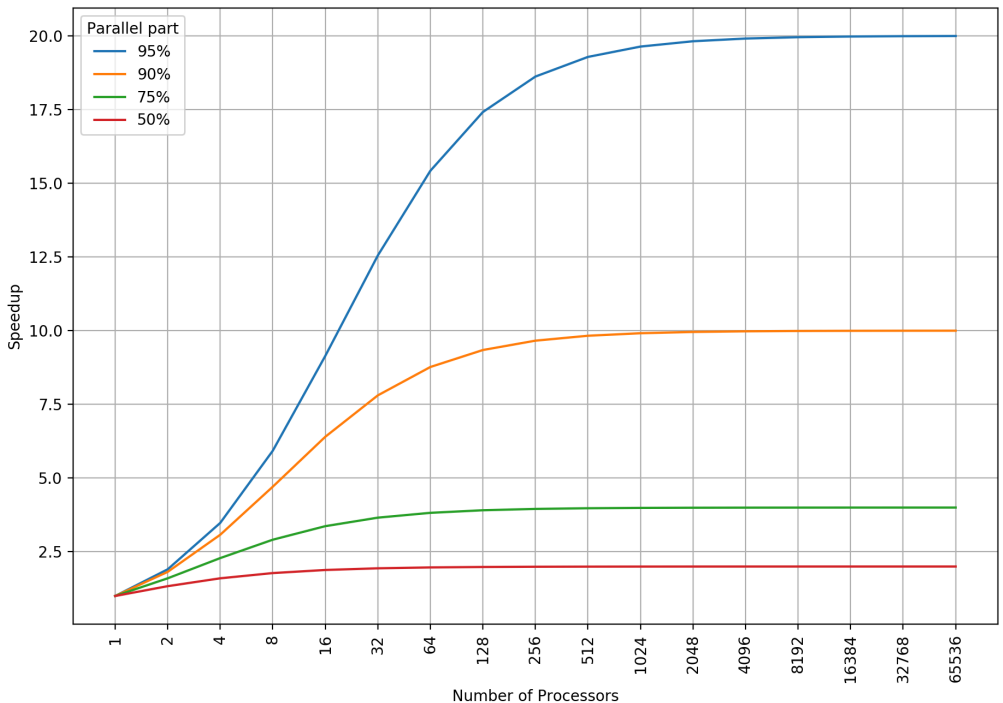
$$S_p = \frac{T_1}{T_p} = \frac{T_1}{fT_1 + (1-f)T_1/p} = \frac{1}{f + (1-f)/p}$$

As $p \to \infty$

$$S_p = \frac{1}{f}$$

The sequential fraction limits the maximum speedup!

Amdahl’s Law Visualization



Gustafson-Barsis' Law

> **Scalability:** Ability of parallel algorithm to achieve performance gains proportional to the **number of processors** and the **size of the problem**

When does Amdahl's Law apply?

- When the **problem size is fixed**
- **Strong scaling** ($p \to \infty, S_p = 1/f$)
- Speedup bound is determined by the degree of **sequential execution time**, not number of processors
- Perfect efficiency is **hard to achieve**

Gustafson-Barsis' Law: Derivation

Often interested in **larger problems** when scaling:

- How big of a problem can be run?
- Constrain problem size by **parallel time**

Assume parallel time is kept **constant**: $T(p) = C = (f + (1 - f)) \times C$

Let $C = 1$, then:

- $T_s = f_{seq} + p(1 - f_{seq}) = 1 + (p - 1) f_{par}$

Speedup:

$$S(p) = \frac{T_s}{T_p} = \frac{T_s}{1} = f_{seq} + p(1 - f_{seq}) = 1 + (p - 1) f_{par}$$

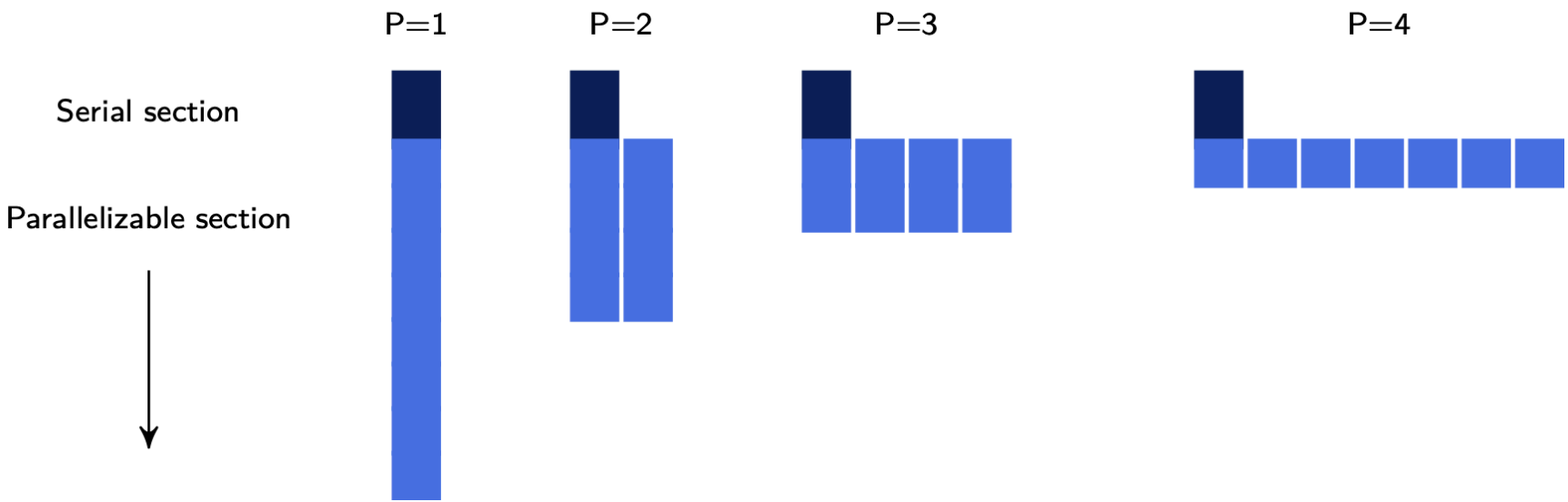
Gustafson's Law Application

When does Gustafson's Law apply

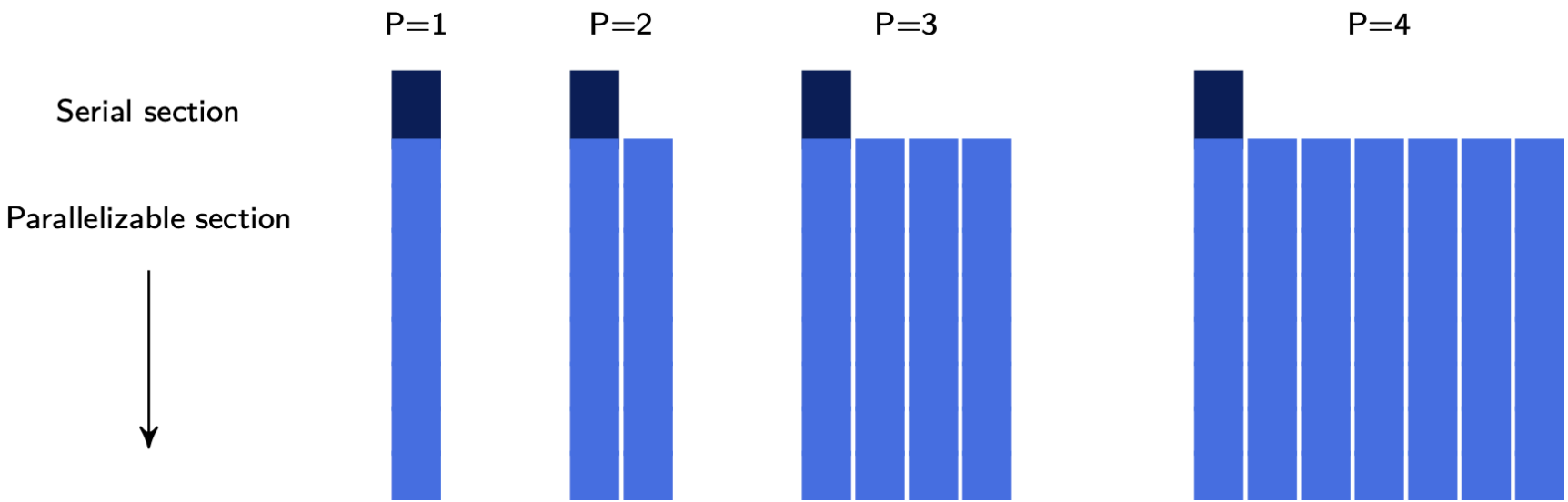
- When the **problem size can increase** as the number of processors increases
- **Weak scaling**: $S_p = 1 + (p - 1) f_{par}$
- Speedup function **includes the number of processors**
- Can **maintain or increase parallel efficiency** as the problem scales

✓ **More optimistic view:** As we add processors, we can solve proportionally larger problems while maintaining efficiency

Amdahl vs Gustafson



Amdahl vs Gustafson



Amdahl vs Gustafson

Amdahl's Law

Strong Scaling

- Fixed problem size
- Speedup bounded by $1/f$
- Focus: How fast can we solve **this** problem?
- Pessimistic view

Gustafson's Law

Weak Scaling

- Scalable problem size
- Speedup: $1 + (p - 1)f_{par}$
- Focus: How **big** a problem can we solve?
- Optimistic view

> **Key difference:** Amdahl fixes work, Gustafson fixes time

DAG Model of Computation

A program seen as **directed acyclic graph (DAG)** of tasks

Properties

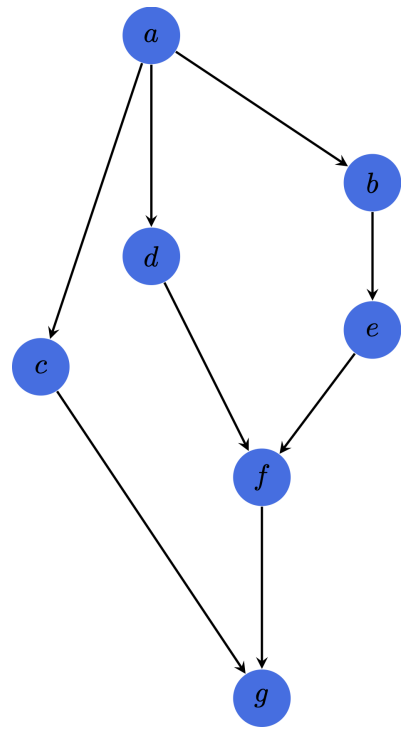
- A task cannot execute until all the **inputs** to the tasks are available
- These come from **outputs** of earlier executing tasks
- DAG shows explicitly the **task dependencies**

Execution

- Hardware consists of **workers** (processing units)
- We consider a **greedy scheduler** of the DAG tasks to workers
- → No worker is idle while there are tasks still to execute

DAG Example

Consider a simple DAG with task dependencies:

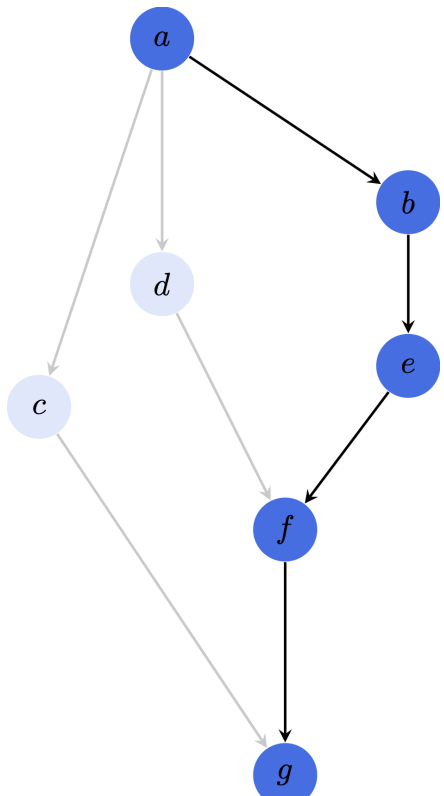


Execution Time in DAG Model

- T_P = time to execute with P workers
- T_1 = time for **serial execution** (sum of all tasks)
- T_∞ = time along the **critical path**
  - Sequence of task execution through DAG that takes the **longest time**
  - Assumes an **infinite number of workers** available

DAG Example: Execution Times

Let each task take **1 unit of time**



$T_1 = 7$: All tasks executed in serial order
(A → B → D → F and C → E counted)

$T_\infty = 5$: Critical path: A → C → E → F
or A → B → D → F
(longest dependency chain)

DAG Bounds

Suppose we only have P workers. **Work-span formula** to derive a lower bound on T_P:

$$\max\left(\frac{T_1}{P}, T_\infty\right) \leq T_P$$

T_∞ is the **best possible execution time**

Brent's Theorem

Captures the additional cost executing tasks **not on the critical path**

Assume we can do so without overhead:

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty$$

Application of Brent's Theorem

Given:

- $T_1 = 7$
- $T_\infty = 5$
- $P = 2$

Calculate:

$$T_2 \leq \frac{T_1 - T_\infty}{P} + T_\infty$$

$$T_2 \leq \frac{7 - 5}{2} + 5$$

$$T_2 \leq 6$$

Estimation of Running Time

Scalability requires that T_∞ be **dominated** by T_1:

$$T_P \simeq \frac{T_1}{P} + T_\infty \quad \text{if } T_\infty \ll T_1$$

Implications

- Increasing work hurts parallel execution **proportionately**
- The **span** impacts scalability, even for finite P
- Need sufficient parallelism for good scaling

Available Parallelism

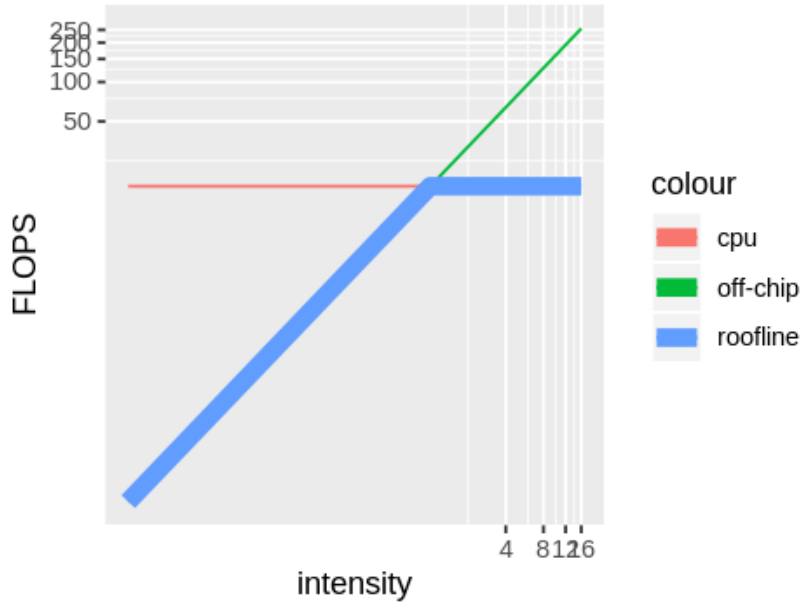
Sufficient parallelism implies **linear speedup**

$$T_p \sim \frac{T_1}{P} \quad \text{if } \frac{T_1}{T_\infty} \gg P$$

The ratio T_1/T_∞ measures the **parallelism** available
It must be much larger than P for good scaling

Roofline Model

Roofline: Concepts



The total number of **FLOPS/s** realized is bounded by either:

1. The amount of **data delivered** to the processor × the operational intensity
   (GB/s · FLOPS/GB)

2. The **peak processing throughput** of the processor

> **Operational intensity** is a measure of how many times you use each byte

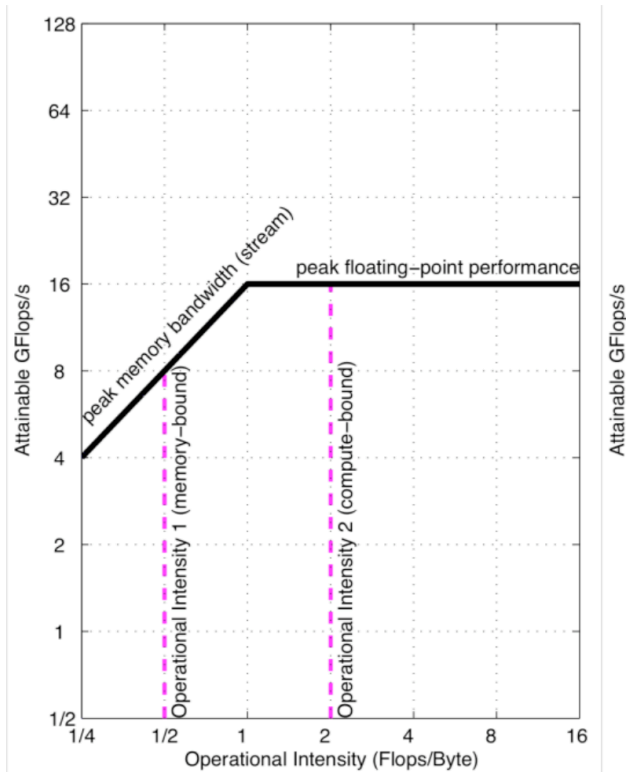
Roofline: Limits
Performance Boundaries

The roofline defines the **feasible region** for computation:

- **No possible program** can exceed the roofline
- **Inefficiencies** in programs may perform below the roofline

Two regimes

1. **Memory-bound**: Limited by data transfer rate
2. **Compute-bound**: Limited by processor throughput



Roofline: Kernels

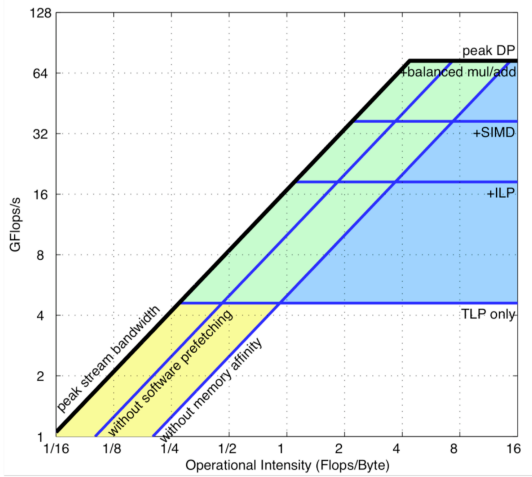
The idea is to measure a given program's **operational intensity**, which corresponds to a vertical line on the plot.

Memory-Bound Kernels

- Low operational intensity
- Limited by bandwidth
- **Optimize:** Data movement

CPU-Bound Kernels

- High operational intensity
- Limited by compute
- **Optimize:** Computation



> The peak of the roofline divides these two regions

Using Roofline to Optimize Code

Enhance CPU Performance

→ Raise peak

- Unroll loops (increase compute density)
- SIMD (vector instructions)
- Better algorithms

Improve Data Movement

→ Shift memory bound

- Sequential data access
- Coherent data access
- Software prefetching
- Cache optimization

✓ **Strategy:** Identify which regime your kernel is in, then apply appropriate optimizations

Conclusion

Conclusion

Key Takeaways

1. **Different metrics** for optimization
   - Speedup, efficiency, scalability
2. **Metrics are associated** with different objectives
   - Strong scaling vs weak scaling
   - Fixed time vs fixed problem size
3. **Comparing with the correct sequential approach** is critical
   - Choice of baseline affects interpretation
   - Must be clearly stated