Week 2: Computer Architecture & Memory Hierarchy

Why Memory Matters More Than CPU Speed

19th of February 2026

Jimmy Jessen Nielsen
`jjn@es.aau.dk`

Dept. of Electronic Systems

Aalborg University

Welcome to Week 2!

Today's Goal: Achieve your first major speedup (5-50×) with NumPy vectorization
Key Question: Why is memory often the bottleneck, not CPU?
What You'll Do:

- Understand memory hierarchy and cache
- Learn vectorization with NumPy
- Transform your Mandelbrot code
- Measure dramatic speedup!

Today's Format:

- Fixed timing for lectures & activities
- **Flexible pacing** for studio work (milestones + extensions)
- Self-managed breaks after milestones
- Can leave after 15:25 if done

Week 1 Recap
Quick Check-In

Who completed the naive Mandelbrot implementation?
Typical baseline timings (1024×1024 grid):

- Naive Python: 20-60 seconds
- Some slower, some faster - that's fine!
- This is your baseline for comparison

Git Status:

- Repository created? ✓
- First commits pushed to GitHub? ✓
- Ready to add vectorized version today!

Issues with Week 1? Ask during studio time or after class

Today's Schedule
Flexible Studio Format

12:30-12:40 Welcome & Recap (10 min)
12:40-13:00 Memory Hierarchy Recap (20 min)
13:00-13:25 Draw-Pair-Share Activity (25 min)
13:25-13:45 Multi-Core & SIMD Recap (20 min)
13:45-13:55 NumPy Introduction (10 min)
13:55-15:00 Studio Session: Vectorize Mandelbrot (65 min)

- Milestone-based with self-paced breaks
- Share with neighbor after each milestone
- Extensions for fast students

15:00-15:10 Convergence Break (10 min)
15:10-16:00 Results Discussion & Performance Analysis (50 min)
Can leave after 15:25 if done!

Memory Hierarchy

Reading Recap: Cache & Memory
Eijkhout Ch. 1.3

The next 6 slides recap important points from your reading on memory hierarchies.

Pay attention! After the recap, you will be asked to *draw* as much as you can remember about these concepts.

Plot twist: You cannot use letters or numbers in your drawings.

Find your pen — distribute paper.

But don't use it before I say so!

Latency and Bandwidth

Two Concepts for Data Movement

Eijkhout 1.3.2: Two fundamental concepts describe memory performance:

Latency: The *delay* between requesting a data item and it arriving.

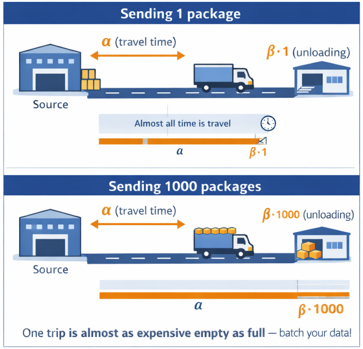
- Measured in nanoseconds or clock cycles
- If the CPU must wait for data, the execution *stalls* ("memory stall")
- Modern CPUs use out-of-order execution to hide latency

Bandwidth: The *rate* at which data arrives after the initial latency.

- Measured in GB/s or bytes/cycle
- Depends on: bus speed × bus width × channels × sockets

Combined model: The time to transfer n bytes is:

$$T(n) = \underbrace{\alpha}_{\text{latency}} + \underbrace{\beta \cdot n}_{1/\text{bandwidth} \times \text{size}}$$



Memory Hierarchy
The Speed Gap

Access Times (approximate):

Level	Latency	Bandwidth	Typical Size
Registers	1 cycle	–	∼1 KB
L1 Cache	4-5 cycles	384 GB/s	32-64 KB
L2 Cache	10-16 cycles	128 GB/s	256 KB-2 MB
L3 Cache	40-80 cycles	51 GB/s	8-36 MB (shared)
RAM	200+ cycles	6-30 GB/s	Gigabytes

(Representative numbers; Intel Sapphire Rapids. See Eijkhout Fig. 1.5)

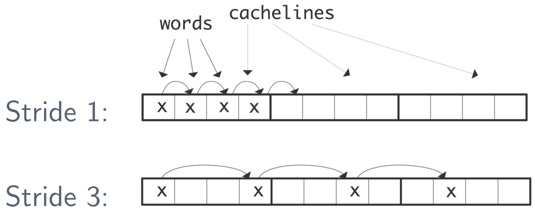
Takeaway:

- RAM is ∼200× higher latency than L1 cache
- Bandwidth drops by ∼60× from L1 to RAM
- CPU can compute much faster than memory supplies data
- Good code keeps data in cache — bad code constantly fetches from RAM

Cache Lines and Spatial Locality
Eijkhout 1.3.5.8

Cache line: The smallest unit of data transferred between RAM and cache. Typically **64 bytes** — enough to hold 8 double-precision (float64) values or 16 single-precision (float32) values.

Why cache lines? Exploit *spatial locality* — if you need one word, you probably need its neighbors soon.



- **Stride 1:** Sequential access loads entire cache line → elements 2–8 are "free"
- **Stride 3:** Only 2-3 elements per cache line used → 2/3 of bandwidth wasted

How RAM Addresses Map to Cache
Eijkhout 1.3.5.9–1.3.5.11

Cache is much smaller than RAM — multiple addresses share each cache location. A memory address is split into three fields:

Tag	**Index (set bits)**	**Offset**
upper bits	middle bits	lower bits (LSBs)

- **Offset** (LSBs): byte position *within* a cache line — 6 bits for a 64-byte line ($2^6 = 64$)
- **Index**: selects the *cache set* (which row the line can go into)
- **Tag**: stored with the data to identify which RAM address is cached

Direct-mapped ($k = 1$): one possible location per address — fast, but prone to *thrashing* when two hot addresses share a set.
k-**way set-associative** ($k = 2 \ldots 8$): k candidate slots per set — fewer conflict misses. Most modern L1/L2 caches are 4- or 8-way.

Note: Arrays sized as exact powers of 2 are especially prone to conflict misses in direct-mapped caches — one practical reason to pad arrays.

Cache Hits, Misses, and Replacement
Eijkhout 1.3.5.2

Cache hit: Data found in cache (fast!)
Cache miss: Data not in cache, must fetch from slower memory (stall!)

Types of cache misses:

- **Compulsory:** First access to data (unavoidable, but prefetch helps)
- **Capacity:** Working set larger than cache $\rightarrow$ data gets *flushed*
- **Conflict:** Multiple addresses map to same cache location
- **Invalidation:** Another core modified the data (multi-core only)

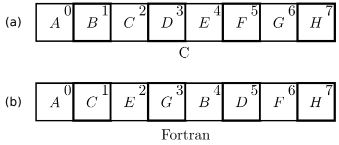
Replacement policy:

- When cache is full, which line to evict? Usually LRU (Least Recently Used)
- ***k*-way associative caches** ($k = 2 \ldots 8$): each address can map to k locations, reducing conflict misses vs. direct-mapped caches

Memory Layout
Row-Major vs Column-Major

How is a 2D array stored in 1D memory?

$$\mathbf{A} = \begin{bmatrix} A & B \\ C & D \\ E & F \\ G & H \end{bmatrix} \in \mathbb{R}^{4\times 2}$$


(a)
A 0 B 1 C 2 D 3 E 4 F 5 G 6 H 7
C
(b)
A 0 C 1 E 2 G 3 B 4 D 5 F 6 H 7
Fortran

- **C / Python (NumPy default):** Row-major $\rightarrow$ row elements are contiguous
- **Fortran / MATLAB:** Column-major $\rightarrow$ column elements are contiguous
- **Rule:** Iterate over the contiguous dimension in the innermost loop!

For NumPy Mandelbrot: Arrays are row-major by default. Accessing `Z[i,:]` (a row) is cache-friendly; `Z[:,j]` (a column) is stride-N.

Draw-Pair-Share

Instructions

Individual Drawing

4 minutes

Draw what you remember:

- Memory hierarchy (levels, latency, bandwidth)
- Cache lines & spatial locality (stride)
- Memory layout (row-major vs column-major)
- How a RAM address maps to a cache set (index/tag/offset)

No letters/numbers. No speaking!

Draw-Pair-Share
Pair Explanation

Pair Explanation

10 minutes total (5 min each)

Find a partner
Take turns explaining

When explaining:

- Walk through your drawing
- Explain the concepts

When listening:

- Ask questions
- Note differences

Draw-Pair-Share
Class Discussion

Class Discussion - 5 minutes

Let's discuss:

- What did people get right?
- Common gaps?
- Biggest "aha!" moments?
- Connection to Mandelbrot optimization?

Key Takeaway: Memory access patterns matter more than algorithm complexity!

Effective Bandwidth

Recap: Connecting the Concepts

Recall: $T(n) = \alpha + \beta \cdot n$, where $\beta = 1/\text{bandwidth}$

But β is not a hardware constant! It depends on your access pattern:

Access pattern	**Cache line use**	**Effective β**
Stride-1 (sequential)	8/8 doubles used per line	β_{peak} (hardware limit)
Stride-2	4/8 doubles used per line	$\sim 2 \times \beta_{\text{peak}}$
Stride-N (column in row-major)	1/8 doubles used per line	$\sim 8 \times \beta_{\text{peak}}$
Random access	often 1/8 per line + no prefetch	$\gg 8 \times \beta_{\text{peak}}$

Why? Hardware loads entire cache lines (64 bytes). If you only use 1 element per line, you waste 7/8 of the bandwidth.

The model still holds — but $\beta_{\text{effective}}$ can be 8–50$\times$ worse than β_{peak} depending on memory layout and access stride.

You'll measure this yourself in today's exercises.

Parallel Architectures

Multi-Core Architecture

How Cores Share Memory

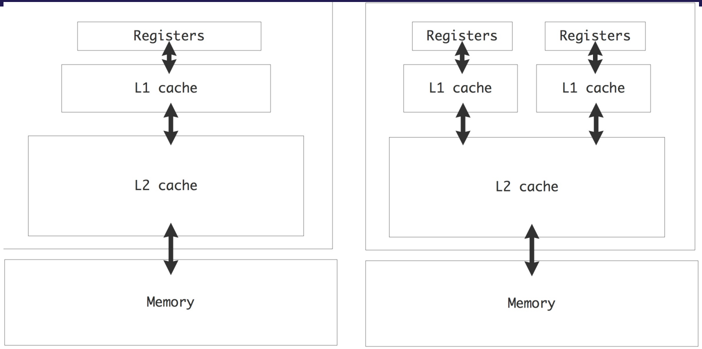


Figure 1.13: Cache hierarchy in a single-core and dual-core chip

Key points (Eijkhout 1.4):

- Each core has **private L1 cache** (fastest, smallest)
- Cores may share L2 or L3 cache (architecture-dependent)
- All cores share main memory

Cache Coherence
Keeping Caches Consistent (Eijkhout 1.4.1)

The Problem: Core 1 and Core 2 both cache data X. Core 1 modifies X → Core 2's copy is *stale*!

MSI Protocol — each cache line is in one of three states:

- **Modified:** Changed locally; must write back to shared cache/memory before eviction or sharing
- **Shared:** Present and unmodified; may exist in other caches
- **Invalid:** Not usable; absent or invalidated by another core's write

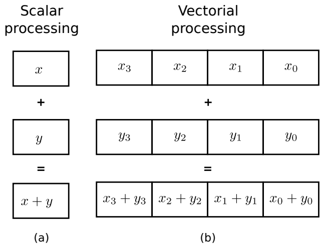
Two approaches for maintaining coherence:

1. **Snooping:** Each cache monitors the memory bus for writes by other cores. Simple, works for small core counts.
2. **Directory-based:** Central directory tracks which caches hold which lines. Scales to many cores but adds overhead.

Performance cost: Coherence traffic uses bandwidth that could be used for computation. Minimizing shared, writable data between cores is important!

SIMD: Single Instruction, Multiple Data
Eijkhout 2.3.1

Key idea: One instruction operates on *multiple* data elements simultaneously



- **AVX-256:** 4 doubles per instruction (most current CPUs)
- **AVX-512:** 8 doubles per instruction (Intel server chips)
- Requires *contiguous, aligned* data in memory $\to$ cache-friendly arrays!
- Compilers can auto-vectorize simple loops; NumPy/BLAS use SIMD internally

Connection: When NumPy computes `Z = Z*Z + C`, the underlying C code uses SIMD to process 4–8 values per instruction — one reason for 5–50$\times$ speedup.

Memory Consistency Models

How Operations Are Ordered (Eijkhout 1.4.2)

Beyond cache coherence: When one core writes data, *when* do other cores see the new value?

Sequential consistency (strongest, slowest)	**Relaxed consistency** (weaker, faster)
All cores see writes in the same order they were issued. Predictable but slow — CPU cannot reorder operations.	CPU/compiler may reorder operations for speed. Different cores may temporarily disagree on values. Needs explicit synchronization when ordering matters.
Analogy: a shared whiteboard where everyone sees updates instantly.	*Analogy: memos that arrive at different desks at different times.*

In practice: x86 (Intel/AMD) is mostly ordered. ARM (Apple M-series) is more relaxed.

For NSC: Python's multiprocessing and Dask handle synchronization for you. But this is *why* synchronization has a cost — it forces ordering that the hardware would otherwise skip.

NumPy Introduction

What is NumPy?
And Why Is It So Fast?

NumPy = Numerical Python

Why naive Python is slow:

- Interpreted (not compiled)
- Dynamic typing (checks types at runtime)
- Python loops call interpreter for each iteration
- Lists are not memory-contiguous

Why NumPy is fast:

- **Pre-compiled C/Fortran code** (no interpretation overhead)
- **Contiguous memory** (cache-friendly! Stride-1 access)
- **SIMD vectorization** (processes 4–8 values per instruction)
- **Optimized BLAS libraries:** Intel MKL, OpenBLAS, Apple Accelerate
- Operates on entire arrays at once

Result: 5-50$\times$ speedup over naive Python loops!

NumPy Basics
Arrays vs Lists

Python list (slow, flexible):

```
x = [1, 2, 3, 4]  # Can hold mixed types
y = [x[i]**2 for i in range(len(x))]  # Python loop
```

NumPy array (fast, typed):

```
import numpy as np
x = np.array([1, 2, 3, 4])  # All same type
y = x**2  # Vectorized! No Python loop
```

Key Differences:

- Homogeneous (all same type) $\rightarrow$ contiguous memory
- Operations apply to entire array (no Python loop)
- Executed in compiled C code with SIMD

Creating Arrays
For Mandelbrot

What we need: Grid of complex numbers $c = x + iy$

Step 1: Create 1D arrays

```
x = np.linspace(-2, 1, 1024)      # 1024 x-values
y = np.linspace(-1.5, 1.5, 1024) # 1024 y-values
```

Step 2: Create 2D grid (meshgrid)

```
X, Y = np.meshgrid(x, y)
# X, Y are now 1024x1024 arrays
```

Step 3: Create complex grid

```
C = X + 1j*Y  # Note: 1j is imaginary unit in Python
# C is now 1024x1024 complex array
```

Studio Session: Vectorize Mandelbrot

Studio Session Overview

Milestone-Based with Flexible Pacing

Time: 65 minutes - Work at your own pace!

Structure:

- ✓ **Milestone 1:** Basic arrays & meshgrid (Target: 15 min)
- ✓ **Milestone 2:** Full vectorized Mandelbrot (Target: 40 min)
- ✓ **Milestone 3:** Memory access pattern analysis (Target: 15 min)
- ✓ **Milestone 4:** Problem size scaling (Target: 15 min)
- ⋆ **Extensions:** Memory profiling, region exploration (no time limit)

After EACH milestone:

1. Share with neighbor (compare approaches)
2. Take a 10-minute break (self-managed)
3. Continue to next milestone

Getting stuck? Ask neighbor → Raise hand → Fast students can help others

Performance Tracker
Log Your Optimization Journey

New this semester: A shared Moodle database where you log your Mandelbrot runtimes as you progress through the course.

How it works:

1. After each successful optimization, click "Add entry" in the Performance Tracker on Moodle
2. Fill in your implementation type, resolution, and median runtime
3. Browse others' entries — are you in the right ballpark?

Benchmarking standard: Report the **median of ≥3 runs** using `time.perf_counter()`. Default resolution: 1024×1024.

Today's task: After Milestone 2, log your **naive baseline** and your **NumPy vectorized** result. Your naive entry is the most important one — it's your starting point!

How to Measure Runtime
Reproducible Benchmarking

Use `time.perf_counter()` — highest resolution, monotonic (immune to clock sync).

```python
import time, statistics

def benchmark(func, *args, n_runs=3):
    """Time func, return median of n_runs."""
    times = []
    for _ in range(n_runs):
        t0 = time.perf_counter()
        result = func(*args)
        times.append(time.perf_counter() - t0)
    median_t = statistics.median(times)
    print(f"Median: {median_t:.4f}s "
          f"(min={min(times):.4f}, max={max(times):.4f})")
    return median_t, result

t, M = benchmark(my_mandelbrot, -2, 1, -1.5, 1.5, 1024, 1024, 100)
```

Why median? First run is often slow (cold cache). Outliers from background processes inflate the mean. Median is robust to both.

Milestone 1: Basic Arrays

Target: 15 min

Your Task: Create the complex grid C for the Mandelbrot set

```python
import numpy as np

x = np.linspace(-2, 1, 1024)        # 1024 x-values
y = np.linspace(-1.5, 1.5, 1024)  # 1024 y-values
X, Y = np.meshgrid(x, y)            # 2D grids
C = X + 1j*Y                        # Complex grid

print(f"Shape: {C.shape}")  # (1024, 1024)
print(f"Type: {C.dtype}")   # complex128
```

Checklist:

- ☐ Create 1D arrays with `linspace`
- ☐ Create 2D grid with `meshgrid`
- ☐ Combine into complex array `C`
- ☐ Verify shape and dtype

✓ **Done?** Commit → share with neighbor → break → Milestone 2

Milestone 2: Vectorize Mandelbrot

Target: 40 min

Your Task: Replace Python loops with NumPy operations

Naive approach has 3 nested loops:

1. Loop over rows (`i`)
2. Loop over columns (`j`) → selects pixel/point c
3. Loop over iterations (`n`) → computes $z = z^2 + c$ until escape

NumPy approach — eliminate loops 1 & 2:

- Initialize `Z` and `M` arrays (same shape as `C`)
- Keep only loop 3 (iterations), operate on *all* pixels at once
- Boolean mask: `mask = np.abs(Z) <= 2`
- Update only unescaped points: `Z[mask] = Z[mask]**2 + C[mask]`
- Increment iteration count: `M[mask] += 1`

Hints: `np.zeros_like(C)`, `np.zeros(C.shape, dtype=int)`

✓ **Done?** Commit → log in Performance Tracker → share with neighbor

Validating Your Results

Comparing Naive vs NumPy

Problem: == doesn't work for floating-point comparison!

```
# WRONG - Don't do this:
if naive_result == numpy_result:  # Unreliable!

# CORRECT - Use np.allclose():
if np.allclose(naive_result, numpy_result):
    print("Results match!")
else:
    print("Results differ!")

# Check where they differ:
diff = np.abs(naive_result - numpy_result)
print(f"Max difference: {diff.max()}")
print(f"Different pixels: {(diff > 0).sum()}")
```

Why? Floating-point arithmetic is not exact. Different order of operations can give slightly different results.

Milestone 3: Memory Access Patterns

Performance Analysis

Your Task: Measure the effect of memory layout on performance

1. Create a large **square** array: `A = np.random.rand(N, N)` with $N = 10\,000$
2. Write a function that computes **row sums** by looping over rows:
   `for i in range(N): s = np.sum(A[i, :])`
3. Write a function that computes **column sums** by looping over columns:
   `for j in range(N): s = np.sum(A[:, j])`
4. Time both. Both loops run N times — which is faster and why?
5. Now try with `A_f = np.asfortranarray(A)` (column-major). What changes?

Connect to theory: Row traversal in C-order = stride-1 = full cache line use = β_{peak}. Column traversal = stride-N = 1 element per cache line = $\beta_{\text{effective}} \gg \beta_{\text{peak}}$.

✓ **Done?** Commit → share results with neighbor → Milestone 4

Milestone 4: Problem Size Scaling
Performance Analysis

Your Task: How does Mandelbrot runtime scale with grid size?

1. Run your **vectorized** Mandelbrot for grid sizes: 256, 512, 1024, 2048, 4096
2. Record runtime for each (use `timeit` or manual timing)
3. Plot: grid size vs. runtime
4. **Predict:** If 1024×1024 takes X seconds, what should 2048×2048 take? (Hint: $4\times$ the pixels — do you get $4\times$ the time?)

Questions to consider:

- Is the scaling linear in number of pixels?
- At what size does the working set exceed your L3 cache?
- Do you see a “knee” in the scaling curve where performance degrades?

✓ **Done?** Commit → share plot with neighbor → try extensions

Extension Activities

For Fast Students

Finished all milestones? Try these:

⋆ **Extension 1: Memory Profiling**

- Install: `mamba install memory_profiler`
- Compare peak memory usage: naive vs NumPy
- Why does the vectorized version use more memory?

⋆⋆ **Extension 2: Explore Different Mandelbrot Regions**

- Zoom into the boundary region (more iterations needed)
- Does runtime change? Why? (Hint: the mask)
- Try the famous named regions: **Seahorse Valley**, **Elephant Valley**, or a **Deep Seahorse Spiral**
- *Coordinates for these regions are on the last backup slide*

⋆⋆ **Extension 3: Help Others!**

- Teaching deepens your understanding

Troubleshooting Common Issues

Issue 1: "Shape mismatch errors"

- Check: `C.shape`, `Z.shape`, `M.shape`
- Use `np.zeros_like(C)` to match shapes

Issue 2: "Results look wrong"

- Test single point: Compare naive vs NumPy
- Verify mask-based updates

Issue 3: "Not much speedup"

- Still have Python loops inside NumPy operations?
- Grid too small? Try 1024×1024 or larger

Issue 4: "Complex number errors"

- Use `1j` for imaginary unit (not `i`)
- Use `np.abs()` not `abs()` for arrays

Commit Your Work!

Don't Forget Git

After finishing each milestone:

```
git add mandelbrot.py
git commit -m "Add NumPy vectorization - 47x speedup"
git push
```

Good commit messages:

- ✓ "Add NumPy vectorization - 47× speedup"
- ✓ "Optimize memory access pattern"
- × "Update code"
- × "Fix stuff"

Why commit often?

- Track your progress
- Can revert if you break something
- Shows work for MP1 grading

Convergence Break

Convergence Break
Bring Everyone Together

Convergence Break

10 minutes

Wherever you are, take a break now!

For everyone: Stretch, coffee, save your work.

Not done yet? That's okay! Take the break — you'll have time to continue after the discussion.

Results Discussion

Quick Share-Out
Who Got What Speedup?

Let's see the range of results!

Raise your hand if you got:

- $<5\times$ speedup?
- 5–10$\times$ speedup?
- $>10\times$ speedup?
- Still working on it? (That's fine!)

Interesting findings to share?

- Surprising results?
- Clever optimizations?
- Unexpected challenges?

Why Such Big Speedups?

Connecting Theory to Practice

Typical speedup: 5-50$\times$ (varies with hardware)

Map your result back to today's concepts:

1. **Compiled code instead of interpreted loops**
   - NumPy calls pre-compiled C — no per-element interpreter overhead
2. **SIMD vectorization**
   - One instruction processes 4–8 values simultaneously
   - Only works on contiguous memory (arrays, not lists)
3. **Cache-friendly memory access**
   - Contiguous arrays $\rightarrow$ stride-1 $\rightarrow$ full cache lines utilized
   - Python lists: scattered pointers $\rightarrow$ cache misses

These factors multiply: compiled $\times$ SIMD $\times$ cache $=$ 5-50$\times$

Key insight: All three require *contiguous, typed arrays* — this is why NumPy exists.

Continued Work Time
Final 35 minutes

If you're done with Milestones 1–2:

- Continue with Milestones 3–4 (performance analysis)
- These are **required** — not optional!

If you're done with all milestones:

- Try extension activities
- Help others (teaching deepens learning!)
- Commit your final version
- **You can leave if truly done**

If you're still working on Milestone 2:

- Continue at your own pace
- Goal: Working vectorized version by end of today
- Milestones 3–4 can be homework if needed

Instructor available — circulating to help, or come to front desk.

Wrap-Up

Before Next Week

Complete at Home

Must complete (if not done today):

1. Finish NumPy vectorization
2. Verify image matches naive version
3. Measure and document speedup
4. Complete memory access pattern exercise (Milestone 3)
5. Complete size scaling exercise (Milestone 4)
6. Commit to Git with good message and push

Optional experiments:

- Memory profiling (Extension 1)
- Different regions of Mandelbrot set (Extension 2)

Next week (Week 3): Profiling & Numba

- Algorithmic intensity: compute-bound vs. memory-bound
- Find bottlenecks with profilers
- JIT compilation with Numba $\rightarrow$ another 5-10$\times$ speedup!

Key Takeaways from Week 2

Memory hierarchy (from reading & lecture):

- Latency and bandwidth both degrade away from CPU
- Cache lines exploit spatial locality (stride-1 = good)
- Memory layout (row-major vs column-major) determines access patterns
- Cache coherence is needed when cores share data

NumPy vectorization:

- Operates on entire arrays at once
- Pre-compiled C code + SIMD + cache-friendly layout
- Result: 5-50$\times$ speedup!

Studio format insights:

- Milestone-based pacing works better than fixed timing
- Self-managed breaks respect different speeds
- Peer teaching benefits everyone

Questions?

Questions about today's material?

Struggling with implementation?

Want to discuss your results?

Stick around or email: jjn@es.aau.dk

CPU Architecture Comparison (Backup)

Apple M vs Intel vs AMD

Feature	Apple M-series	Intel Core	AMD Ryzen
Architecture	ARM (RISC)	x86-64 (CISC)	x86-64 (CISC)
Total cores	6-16	6-24	6-16
Core types	P + E cores	P + E cores (hybrid)	All equal
L1 cache/core	128-192 KB	80 KB	64 KB
L2 cache	4-36 MB (shared)	1.25-2 MB/core	1 MB/core
L3 cache	8-16 MB	12-36 MB	32-96 MB
Power efficiency	Excellent	Good	Good
Clock speed	3-4 GHz	4-5.5 GHz	4-5 GHz

Key Differences:

- **ARM (Apple):** Simple instructions (RISC), lower power, high efficiency
- **x86 (Intel/AMD):** Complex instructions (CISC), higher clock speeds, more legacy compatibility
- **Performance cores (P):** High-performance, high-power
- **Efficiency cores (E):** Lower performance, low-power (for background tasks)

Interesting Mandelbrot Regions (Backup)

Coordinates for Extension 2

Standard full view (your baseline):

Region	x_min	x_max	y_min	y_max	max_iter
Full set	−2.0	1.0	−1.5	1.5	100

Famous named regions (zoomed in, need more iterations!):

Region	x_min	x_max	y_min	y_max	max_iter
Seahorse Valley	−0.8	−0.7	0.05	0.15	500
Elephant Valley	0.175	0.375	−0.1	0.1	500
Deep Seahorse Spiral	−0.7487667139	−0.7487667078	0.1236408449	0.1236408510	2000

Why do these need more iterations?

- Near the boundary, points take longer to escape (or never escape)
- Low `max_iter` causes boundary points to be mis-classified as "in the set"
- More iterations → more compute → good test of your speedup!
- **Deep Seahorse Spiral:** zoom factor $\sim 10^9\times$ — enter all decimal places exactly, or the window lands in the wrong place