



Locality and Blocking

Loop interchange and cache-line reuse | Practical 3

Ömer Gölcük — Sabancı University

Course lead: Prof. Dr. Özcan Öztürk

Same operation, different traversal

$$B[j, i] = 2 \times A[i, j] + 1$$

The result stays the same. We change when each element is visited.

Input A

0	1	2
3	4	5
6	7	8



Output B

1	7	13
3	9	15
5	11	17

Separate row-major arrays; one read and one output update per element.

Original teaching example, continued from Practical 1. Reordering is valid for these separate input and output arrays.

Loop interchange swaps the access pattern

Baseline: i outside, j inside

```
for (std::size_t i = 0; i < n; ++i)
    for (std::size_t j = 0; j < n; ++j)
        b[j * n + i] = 2.0 * a[i * n + j] + 1.0;
```

Read A: +8 bytes

Write B: +8n bytes

Interchanged: j outside, i inside

```
for (std::size_t j = 0; j < n; ++j)
    for (std::size_t i = 0; i < n; ++i)
        b[j * n + i] = 2.0 * a[i * n + j] + 1.0;
```

Read A: +8n bytes

Write B: +8 bytes

[1] Inner-loop address increments for 8-byte doubles. Neither order makes both arrays contiguous; measure the outcome.

Blocking finishes a small region first

Bring nearby updates closer together in the traversal.

Read offsets in A

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

2 → 3 → 6 → 7



Write offsets in B

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

8 → 12 → 9 → 13

Reuse is at cache-line granularity—not repeated computation on one value.

Blocking concept after [1]; diagram original. $n = 4$, $\text{tile} = 2$, $ii = 0$, $jj = 2$. Highlighted regions are not packed buffers.

The blocked kernel

```
for (std::size_t ii = 0; ii < n; ii += tile) {  
    for (std::size_t jj = 0; jj < n; jj += tile) {  
        const std::size_t i_end = std::min(ii + tile, n);  
        const std::size_t j_end = std::min(jj + tile, n);  
        for (std::size_t i = ii; i < i_end; ++i)  
            for (std::size_t j = jj; j < j_end; ++j)  
                b[j * n + i] = 2.0 * a[i * n + j] + 1.0;  
    }  
}
```

The outer loops choose a tile; the inner loops visit its elements.

Exact loop-body excerpt from `scaled_transpose_blocked` in `matrix.cpp`. `tile` is a runtime parameter; no extra tile buffers.

Choosing a tile size

Logical data in one input tile and one output tile:

$$W \approx 2 \times \text{tile}^2 \times 8 \text{ bytes}$$

tile = 16

4 KiB

tile = 32

16 KiB

tile = 64

64 KiB

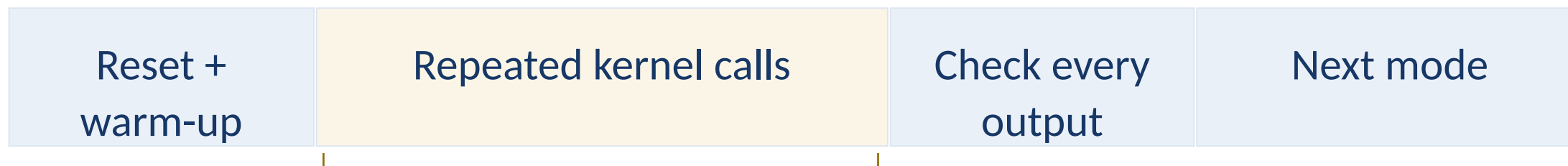
Small tiles add loop overhead; large tiles can lose useful line reuse.

Capacity alone is not enough: alignment and set mapping also matter.

[1] Capacity estimate for two tile × tile regions of 8-byte values. Cache-line rounding, mapping and other active data are not included.

A fair three-way comparison

Same arrays, input, work, build and CPU; randomized mode order.



Re-measure the baseline here; do not reuse an earlier video's timing.

For perf, profile one mode per command—not all three together.

[2–4] Compiler barriers preserve repeated work, not a fixed cache state. External perf collection covers the entire command.

Interpreting the live comparison

Timing plots	Compare kernel time per pass and ns per output element. Show individual runs and medians, not just the fastest run.
Paired speedup	Baseline time $\div$ candidate time in the same program run. A ratio above 1 means that the candidate finished faster.
Source + assembly	Use perf report / annotate to locate each named kernel. Check loop order and vectorization in the actual binary.
Counter evidence	Use perf stat on one mode with matching event scopes. Whole-command counts include setup, warm-up and verification.

[2-4] Plot actual unprofiled runs. General cache events are not direct DRAM-traffic measurements; event support is host-dependent.

References

[\[1\] Cornell CS 3410 — Cache Optimization](#)

Loop interchange and blocked matrix traversal.

[\[2\] Linux perf tutorial](#)

Counting and sampling; event scope and collection support.

[\[3\] perf-annotate\(1\) manual page](#)

Source and assembly view; see also perf-stat(1), perf-record(1) and perf-report(1).

[\[4\] GCC documentation](#)

Optimization options; compiler barriers are not processor fences.

Teaching example: Ömer Gölcük. Original implementation and diagrams; full commands and links accompany the lecture.

Thank you!



**Co-funded by
the European Union**



EuroHPC
Joint Undertaking



EuroCC 3 has received funding from the [European High-Performance Computing Joint Undertaking \(JU\)](#) under Grant Agreement No. 101306701. The JU receives support from the [European Union](#)'s Digital Europe Programme and Germany, Albania, Austria, Belgium, Bosnia and Herzegovina, Bulgaria, Croatia, Cyprus, Czechia, Denmark, Estonia, Finland, France, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Montenegro, the Netherlands, North Macedonia, Norway, Poland, Portugal, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Türkiye, and Kosovo.

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or EuroHPC Joint Undertaking. Neither the European Union nor the EuroHPC Joint Undertaking can be held responsible for them.