# Lab Guide 2

# Impact of the Memory Hierarchy

**Objectives:**

- Review of base locality concepts: spatial and temporal locality
- Understand basic programming technics to improve data locality
- Quantify the impact of data locality on performance

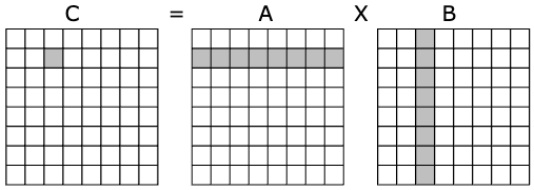
## Introduction

The gap between processor performance and memory performance makes the memory hierarchy a key element of computer performance. The efficient usage the memory hierarchy requires locality in memory accesses: spatial locality when memory is accessed through continuous addresses; temporal locality when the same memory address is accessed several times in a short period. The matrix multiplication case study is an algorithm where locality can be easily exploited, but there are many algorithms where locality is hard to explore or inexistent.

## Exercise 1 - Spatial Locality

```
foreach line of A
     foreach column of B
       C_line,column = DOT_line of A,column of B
```

$$C_{ij} = DOT_{linha\_A_i, coluna\_B_j} = \sum_{k=0}^{n-1}(A_{ik} * B_{kj})$$




a) Data locality analysis: Look at the traditional matrix multiplication version (the DOT version, in the algorithm/figure above). For each memory access ($C_{ij}$, $A_{ik}$ and $B_{kj}$) identify those that exhibit spatial locality (suggestion: start by looking at the DOT definition, eventually also look at the DOT assembly code in lab guide 1, exercise 2c).

b) Estimation of data locality impact**:** Estimate the number of level 1 cache misses for a NxN matrix, assuming that matrix size is greater than the L1 data cache size (suggestion: focus on the matrix accesses without spatial locality). Fill the table with the estimated values for this base version. How many L1 cache misses are expected per instruction executed (with `-O2` optimisation)?

c) Improvement of spatial locality: Design an optimised DOT implementation by transposing matrix A and/or B in order to get the best spatial locality. Estimate the number of level 1 cache misses of this "transp" version and fill the table with the estimated values (ignore the cost of transposing the matrix).

d) Measure and discuss the results: Measure the number of L1 data cache misses with `perf stat -e L1-dcache-load-misses -M cpi ./a.out`. Complete the table for matrix size of 512x512 (**always** using the `-O2` optimisation level) for the base and for the optimised (i.e., with the transpose) versions. What is the explanation for the performance differences between these two versions? Why is the gain not directly proportional to the improvement in cache misses?

| N | Version | Time | CPI | #I (inst_retired.any) | L1_DMiss (estimated) | L1_DMiss | Miss/#I |
|---|---|---|---|---|---|---|---|
| 512 | base() | | | | | | |
| | transp() | | | | | | |

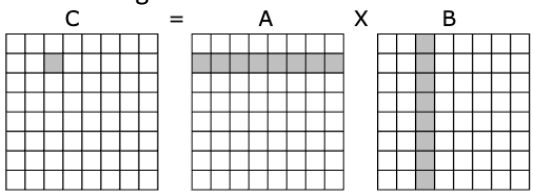
e) (*) MM data locality analysis**:** Matrix multiplication is implemented with three nested loops. The algorithm has the particularity of producing correct results for any order of these loops. How many variations are possible by changing the order of the loops? What is the order that minimises the number of cache misses (i.e., with the best spatial locality), assuming a matrix larger than the L1 cache?

f) (*) Impact of matrix representation: Compare the MM "transp" implementation against an implementation using an array of pointers (also with the transpose). Take some metrics that give insights of the performance difference between versions.

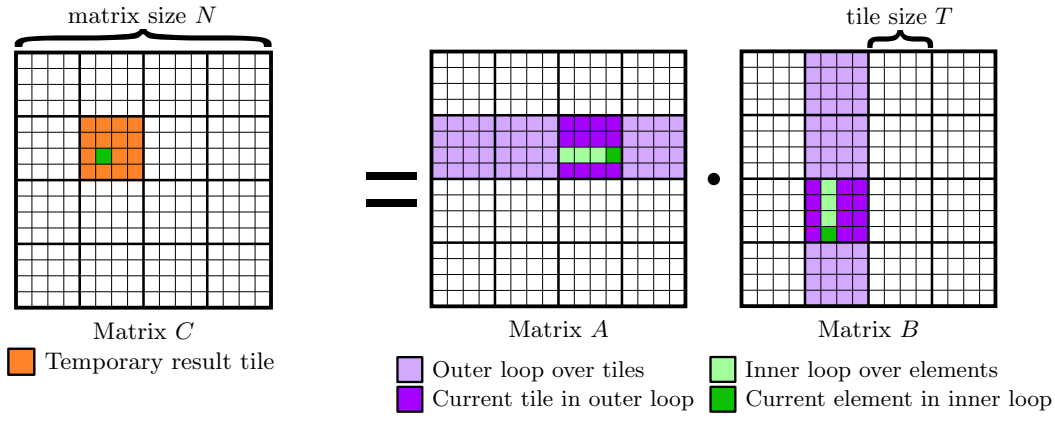
## Exercise 2 - Temporal locality

a) Look at the base matrix multiplication code and identify matrix positions (only for matrix A and B) that are **reloaded** from memory in outer loops (note: this can also be identified looking at the DOT figure). Change the inner loop to take advantage of this temporal locality (e.g., to compute 4 dot products in the inner loop) and estimate the improvement in L1 cache misses for a large matrix size.

```
/* Cij =0 */
for i=0 to N-1
     for j=0 to N-1
          for k=0 to N-1
              C[i][j] += A[i][k] * B[k][j]
```




b) (*) Develop a tiled-based implementation of matrix multiplication, based on the next figure. The generic idea in this exercise is to reorder the matrix computation to maximise the reuse of data in L1 cache. This implementation divides the computation of matrix C into small blocks (i.e., tiles of size TxT) that are computed by multiplying a small block of matrix A with a small block of matrix B (both A and B TxT blocks should fit into L1 cache). This requires 3 additional outer loops to step across blocks, as well as the rewrite of the original (now inner) loops to compute within blocks of size T. Fill the table with experimental data.




| | Block (T) | Time | CPI | #I (inst_retired.any) | L1_DMiss |
|---|---|---|---|---|---|
| base | --- | | | | |
| | | | | | |
| Tile | 8 | | | | |
| | 16 | | | | |
| | 32 | | | | |
| | 64 | | | | |