# Parallel Matrix Multiplication

Group 20

蔡哲平　　陳玟璇
111062625　　111062588

# Outline

- Introduction
- Problem Formulation
- Implementation
  - Sequential
  - OpenMP
  - Pthread
  - MPI
  - Single-GPU
  - IO Improvement
- Experimental Results
- Future Works

# Introduction

# Introduction

Matrix multiplication is widely used in various fields, including mathematics, physics, engineering, etc.. Moreover, it’s one of the most basic and intensive operations in machine learning and deep learning which are popular nowadays. Therefore, if the speed of matrix multiplication can be accelerated, it can bring significant improvement in multiple fields.There have been many studies on how to speed up matrix multiplication from the past and now. In this project, we will try to use the parallelization method taught in the course to accelerate matrix multiplication.

# Problem Formulation

# Problem Formulation

- Given two nxn matrices, A and B respectively, output the matrix C representing the result of A*B.
  - 1 <= n <= 10000
  - 1 <= Aij, Bij <= 100

$$\underset{A}{\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}} * \underset{B}{\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}} = \underset{C}{\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}}$$

# Implementation

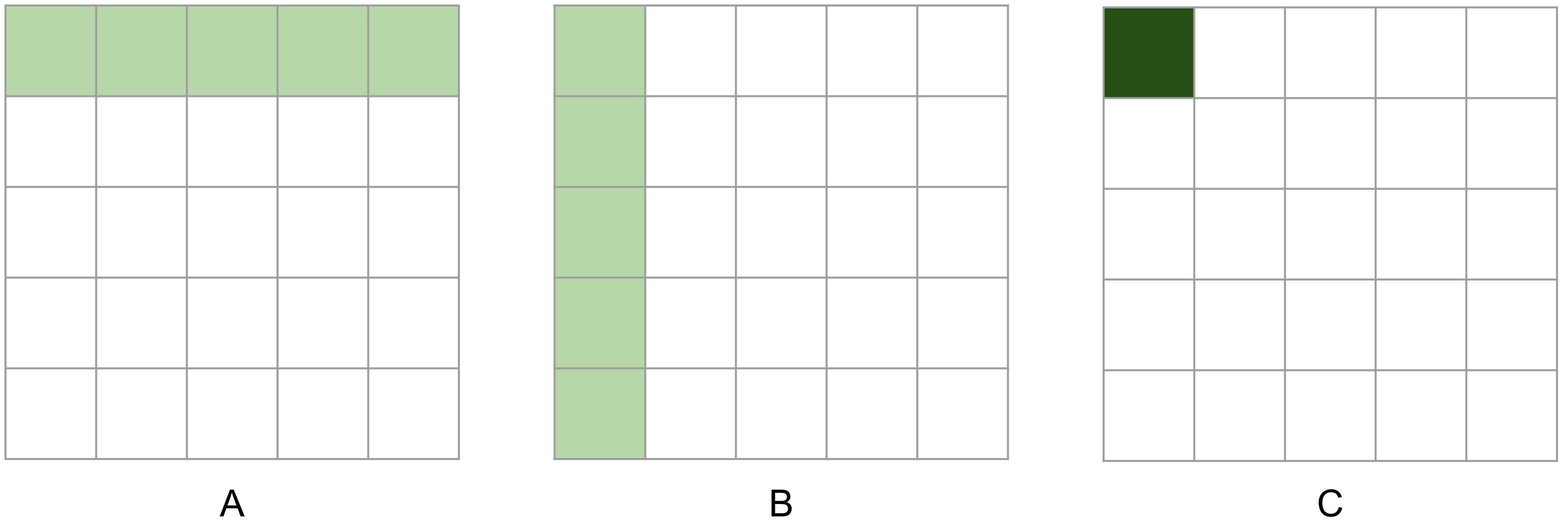
# Implementation: Sequential

- Naive
  - For each element in C, suppose its index is (i, j), it can be obtained by multiplying term-by-term the entries of the ith row of A and the jth column of B, and summing these n products.
  - Time complexity: O(n^3)

```c
void multiply_naive(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int sum = 0;
            for (int k = 0; k < n; k++)
                sum += a[i * n + k] * b[k * n + j];
            c[i * n + j] = sum;
        }
    }
}
```

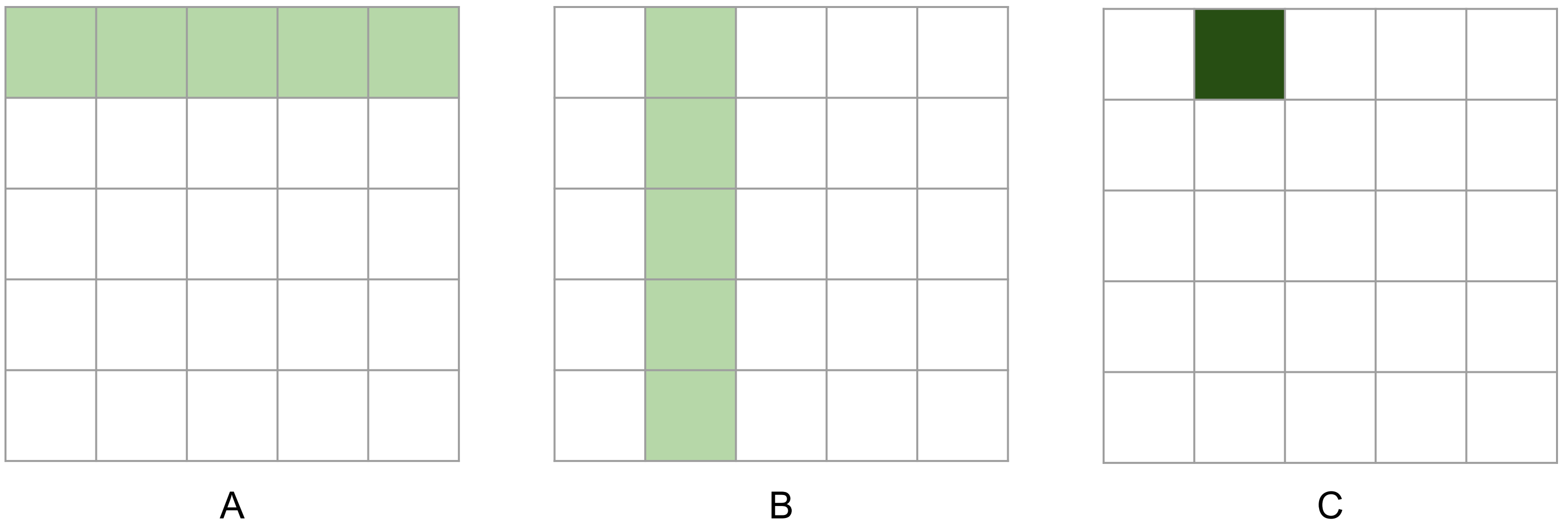
# Implementation: Sequential

- Naive: Calculating C(0, 0)



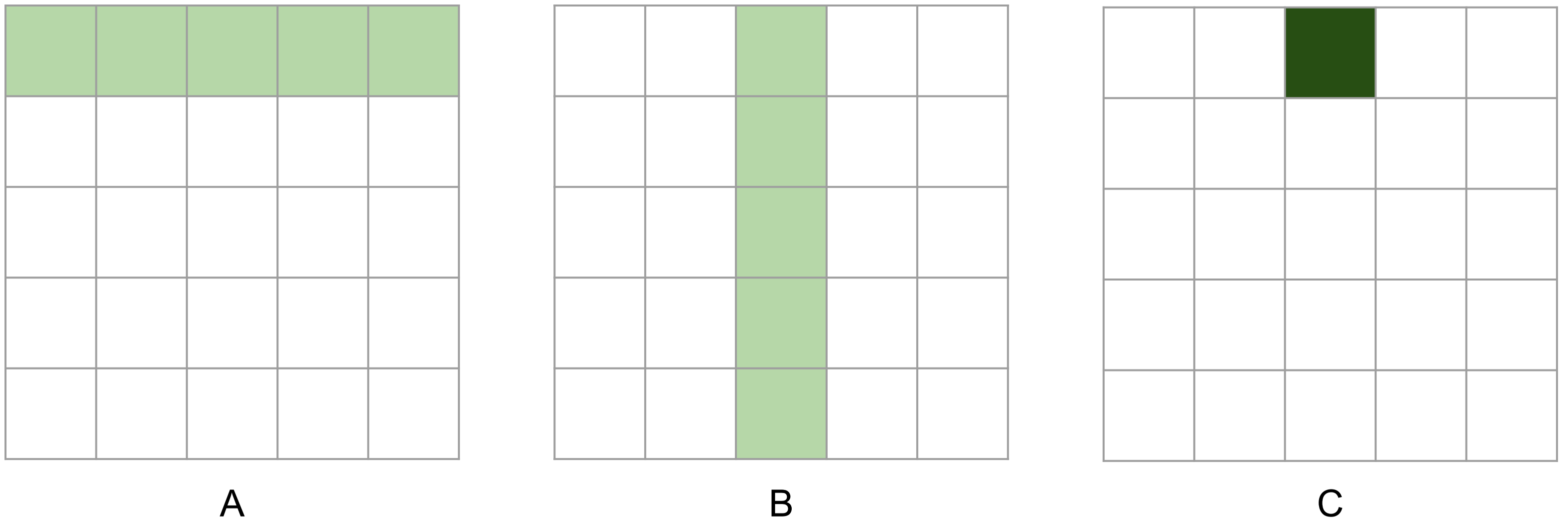
# Implementation: Sequential

- Naive: Calculating C(0, 1)



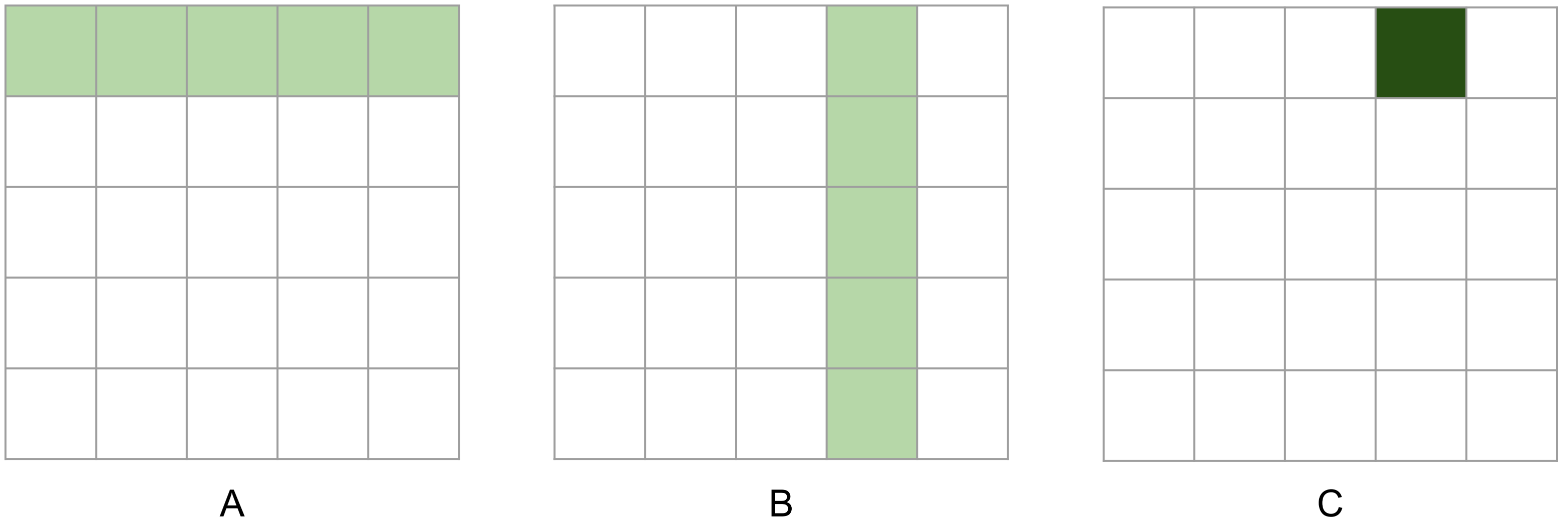
# Implementation: Sequential

- Naive: Calculating C(0, 2)



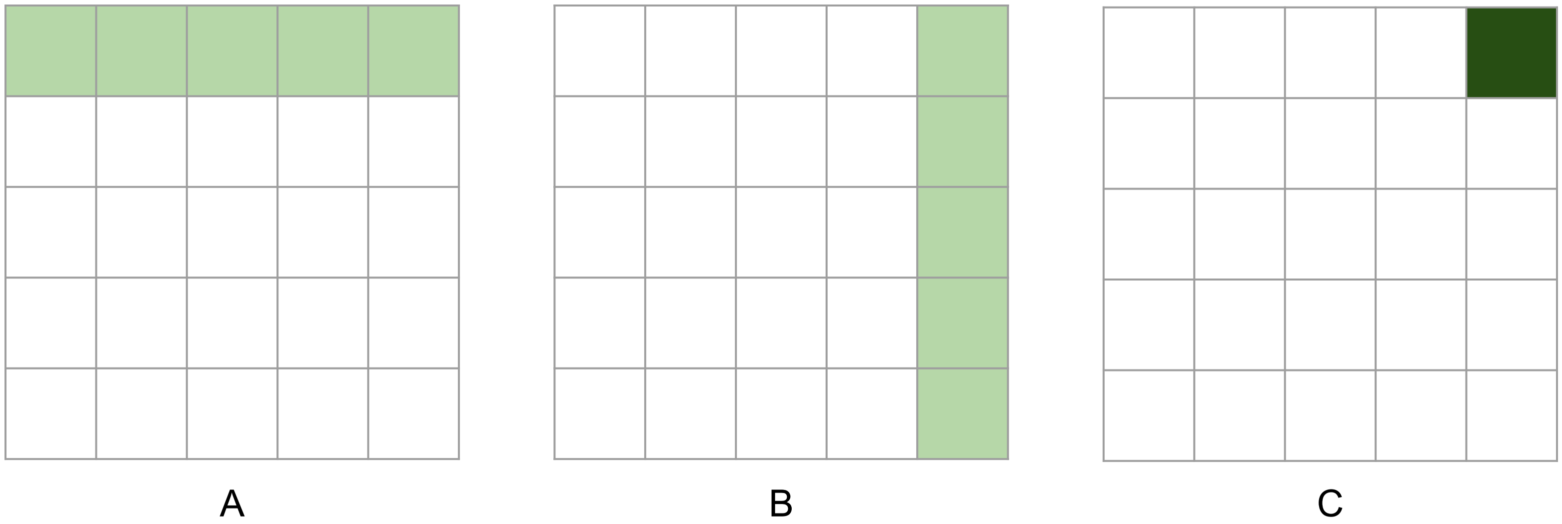
# Implementation: Sequential

- Naive: Calculating C(0, 3)



# Implementation: Sequential

- Naive: Calculating C(0, 4)



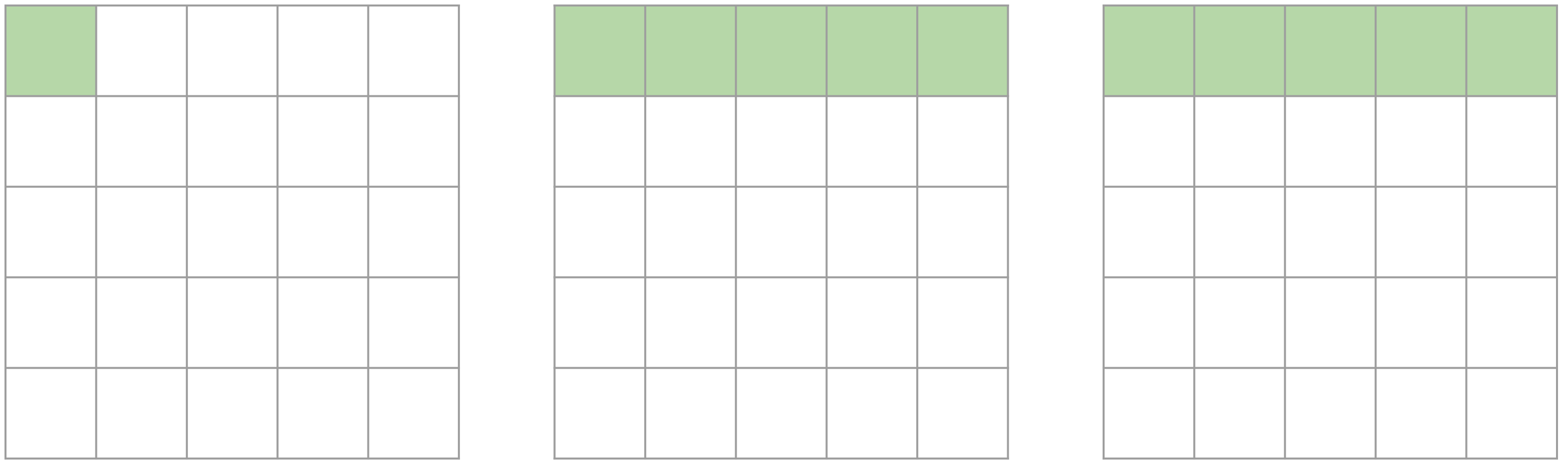
# Implementation: Sequential

- Cache-friendly
    - Temporal locality: Cached data that is recently used
    - Spatial locality: Cached nearby data of the recently used data
    - Array stored in memory with row-major.
    - In the naive version, the way we access B violates spatial locaity since we access B with column major.

```c
void multiply_cache_friendly(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            int r = a[i * n + k];
            for (int j = 0; j < n; j++)
                c[i * n + j] += r * b[k * n + j];
        }
    }
}
```

# Implementation: Sequential

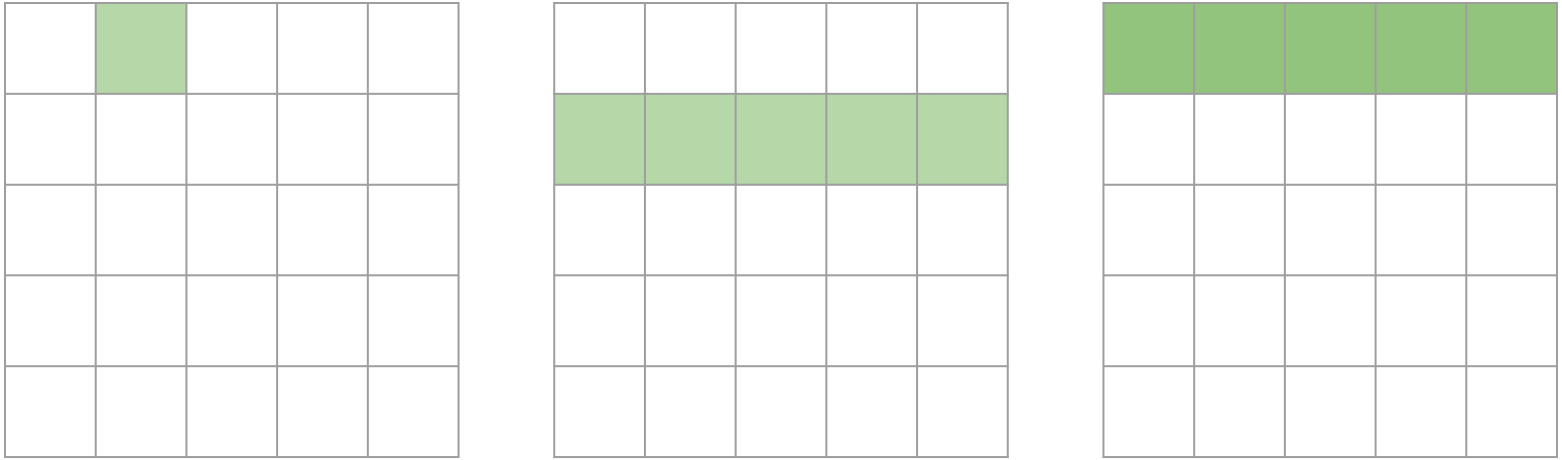
- Cache-friendly



A B C

# Implementation: Sequential

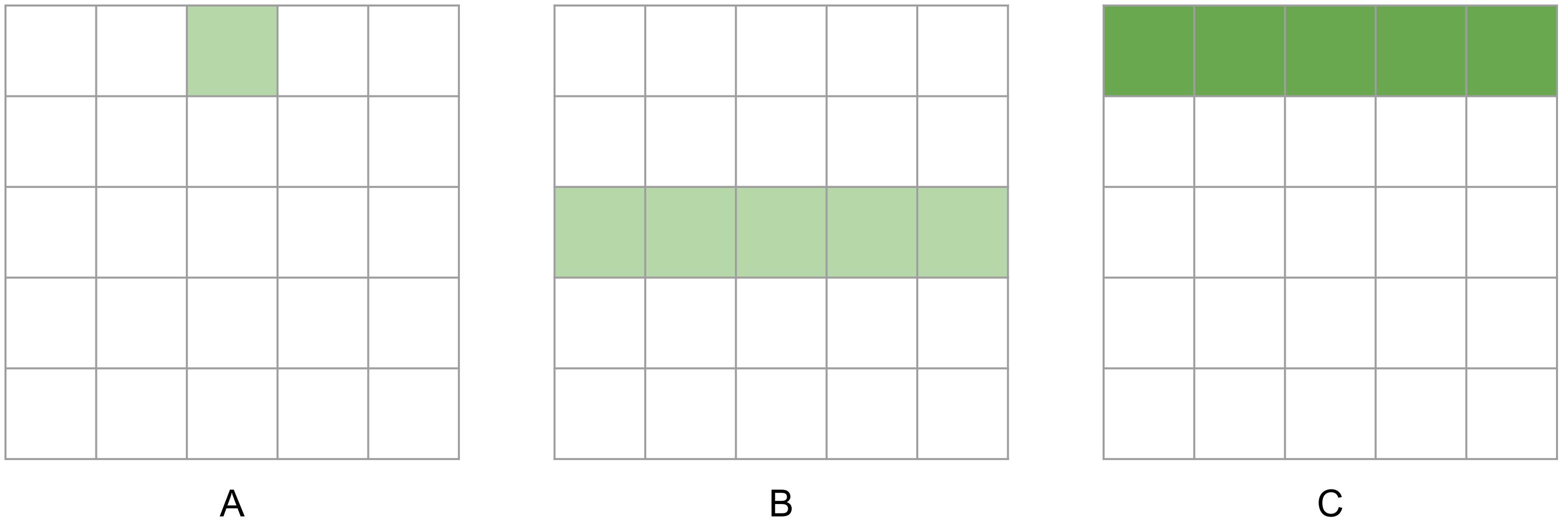
- Cache-friendly



A B C

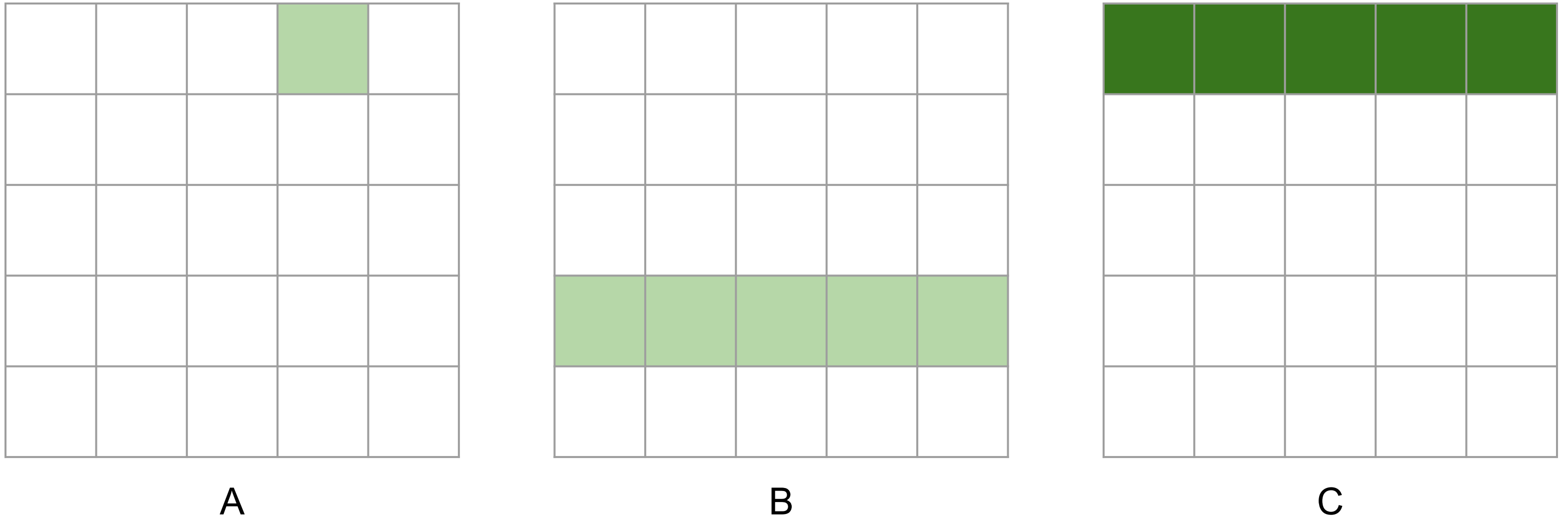
# Implementation: Sequential

- Cache-friendly



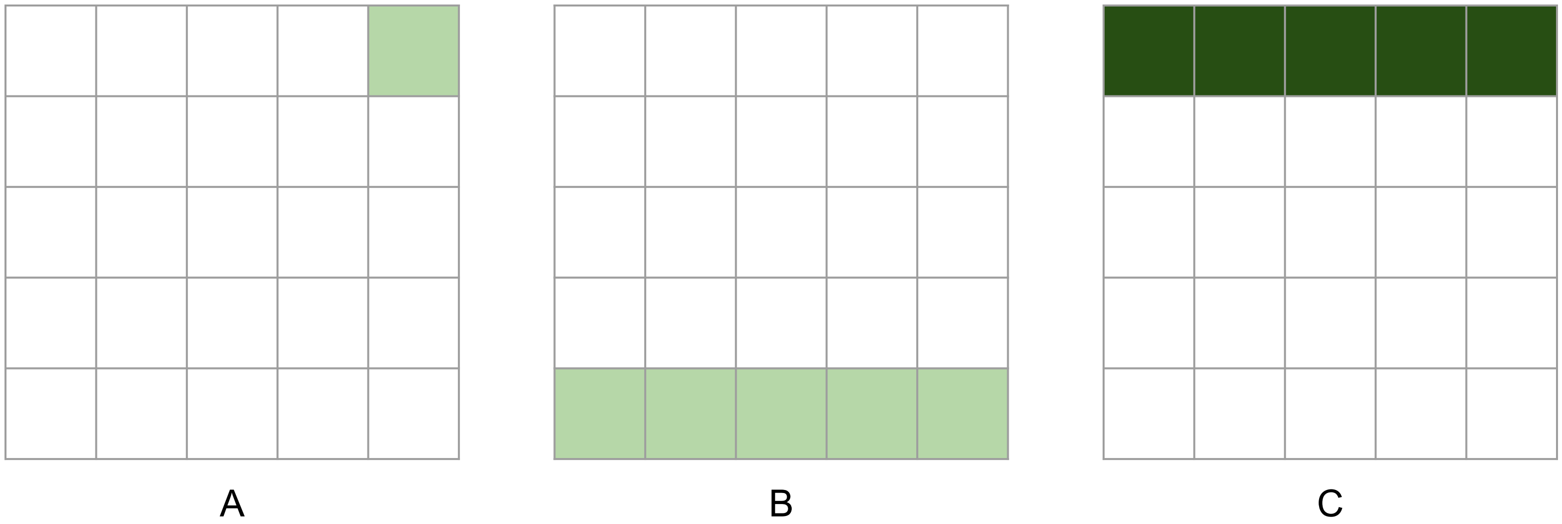
# Implementation: Sequential

- Cache-friendly



# Implementation: Sequential

- Cache-friendly



# Implementation: Sequential

- Cache-friendly using SIMD-v1
    - 128 bits-registers in Intel Intrinsics
    - 128 / 32 = 4, 4 integers can be computed simultaneously
    - Integrate it into cache-friendly version

| | | | |
|---|---|---|---|
| C(i, j) | C(i, j+1) | C(i, j+2) | C(i, j+3) |

+=

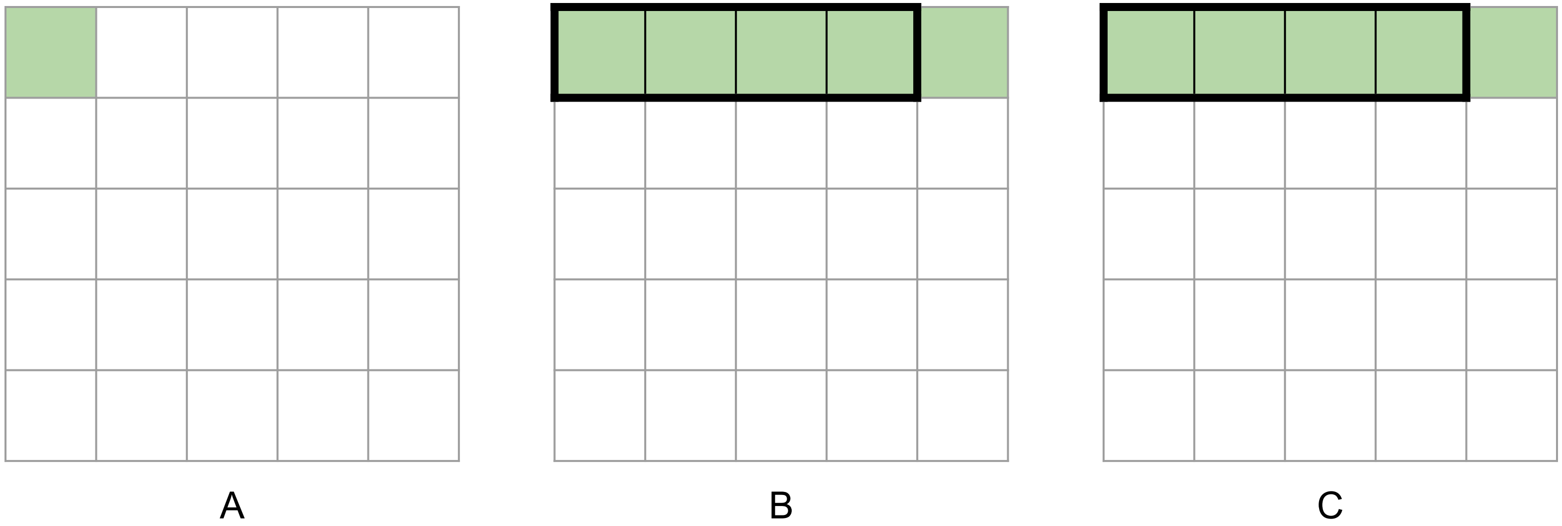
| | | | |
|---|---|---|---|
| A(i, k) | A(i, k) | A(i, k) | A(i, k) |

*

| | | | |
|---|---|---|---|
| B(k, j) | B(k, j+1) | B(k, j+2) | B(k, j+3) |

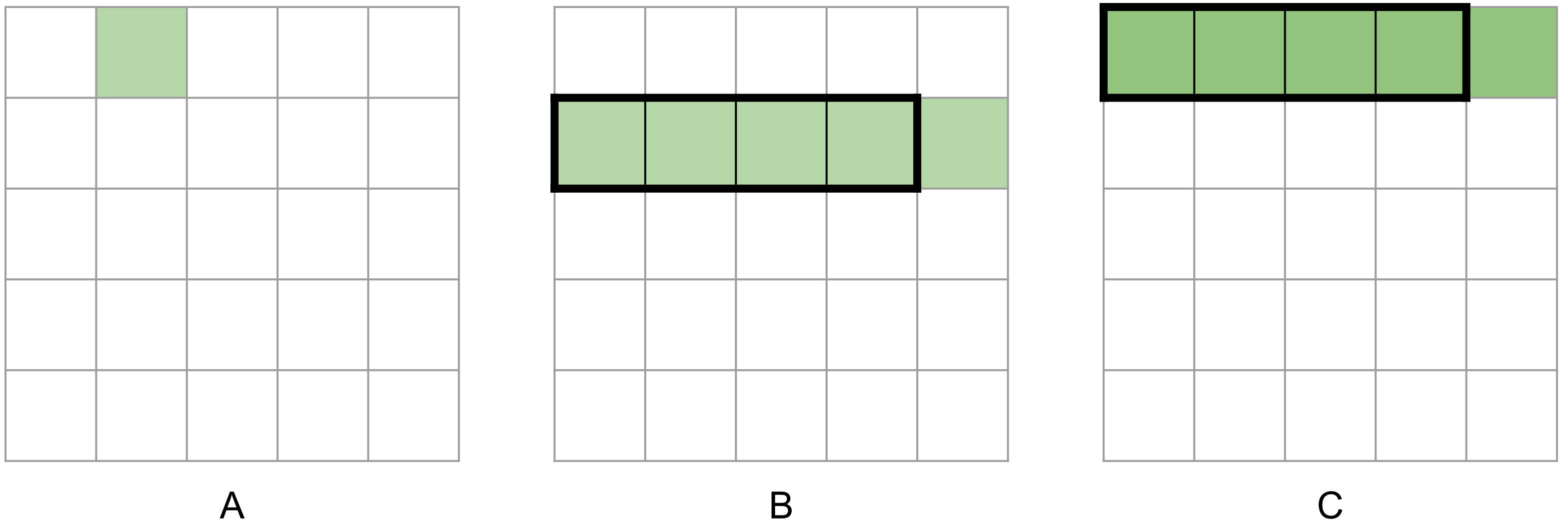
# Implementation: Sequential

- Cache-friendly using SIMD-v1



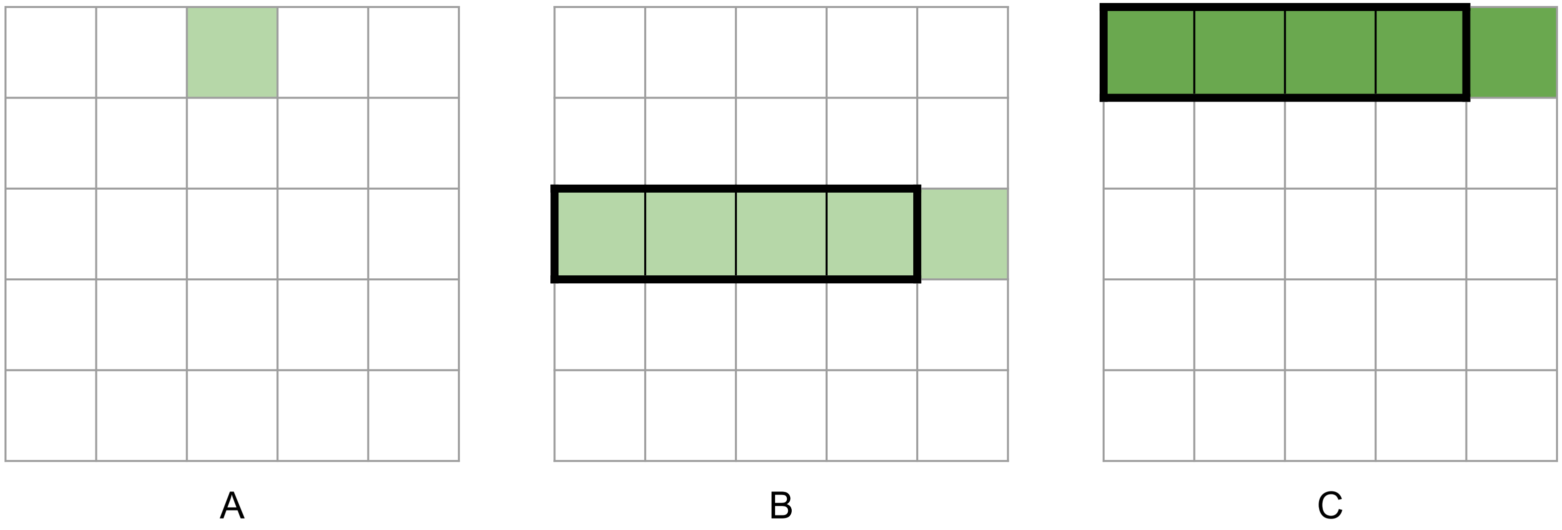
# Implementation: Sequential

- Cache-friendly using SIMD-v1



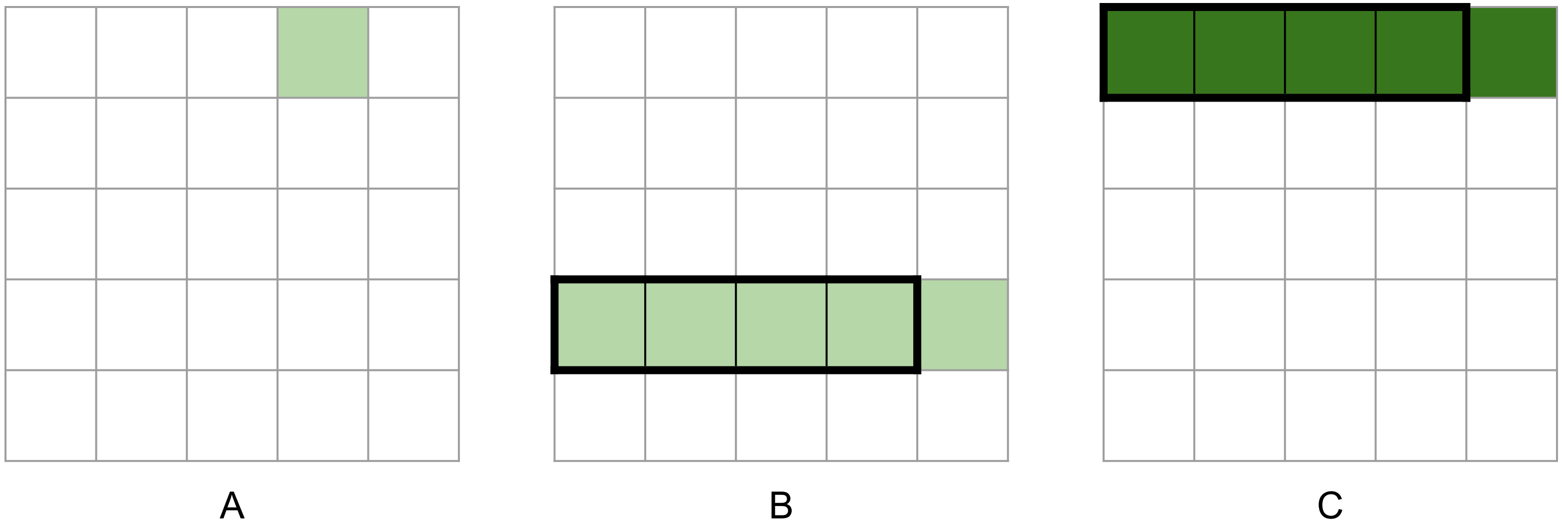
# Implementation: Sequential

- Cache-friendly using SIMD-v1



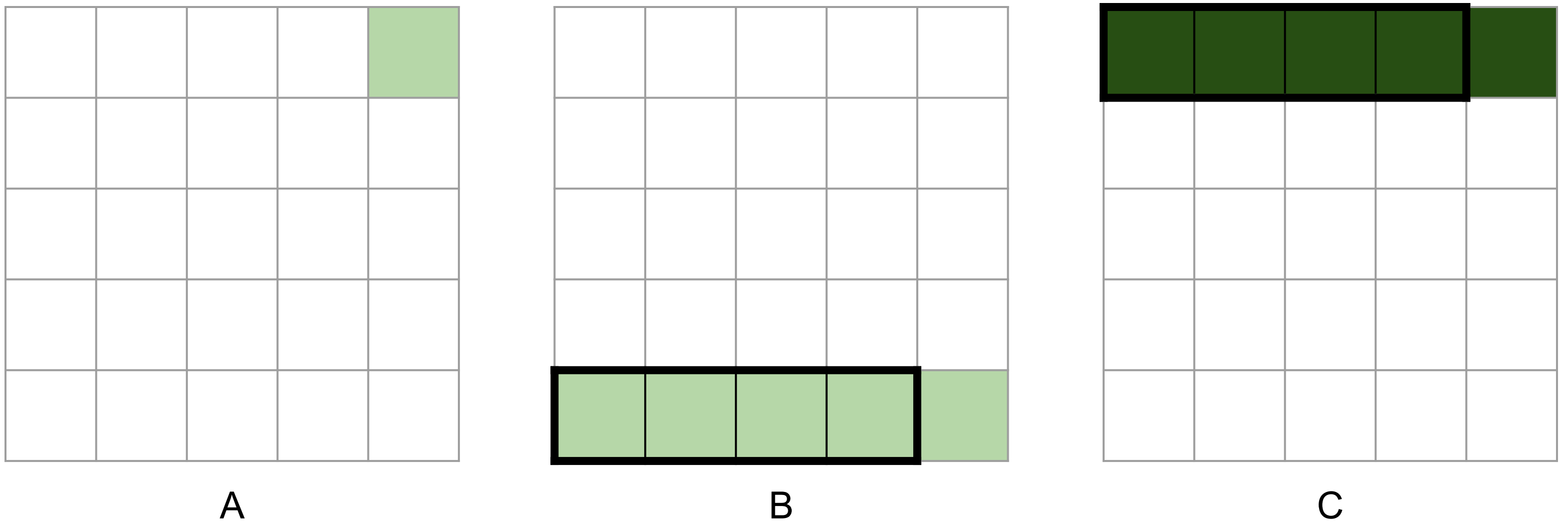
# Implementation: Sequential

- Cache-friendly using SIMD-v1



# Implementation: Sequential

- Cache-friendly using SIMD-v1

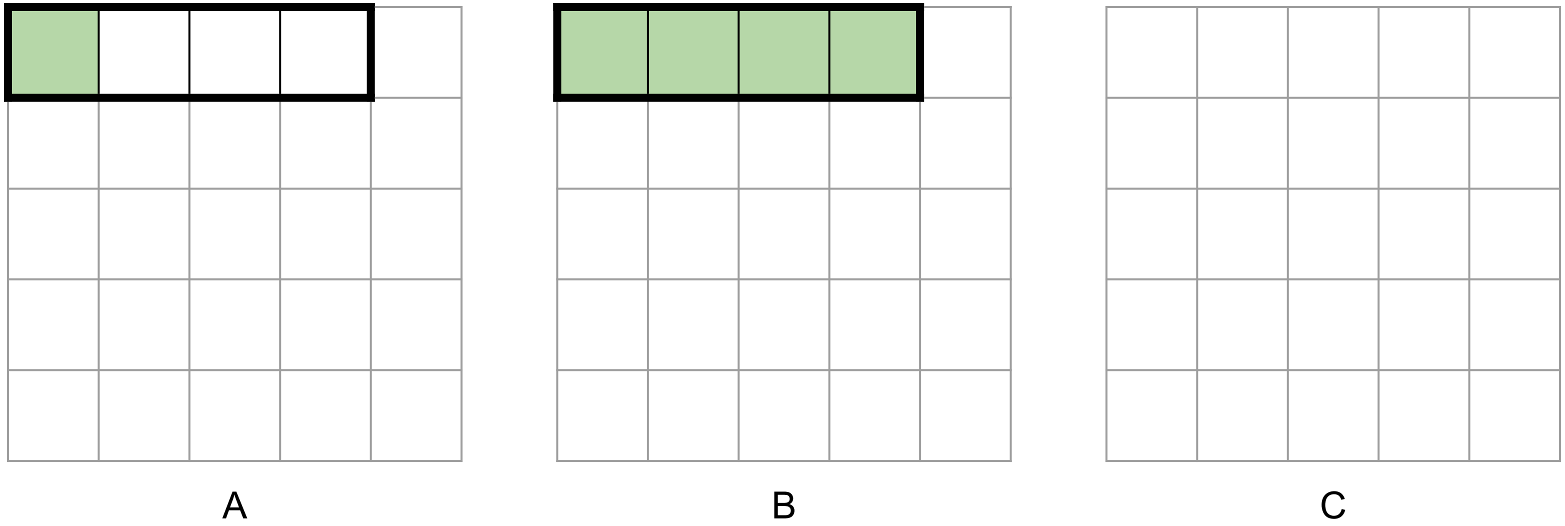


# Implementation: Sequential

- Cache-friendly using SIMD-v2
  - Computation of matrix multiplication between memory load store is less than HW2 (Mandelbrot Set). Therefore, the memory access overhead hides the computation improvement.
  - The way to speedup is try to do as much computation as possible between memory load store.

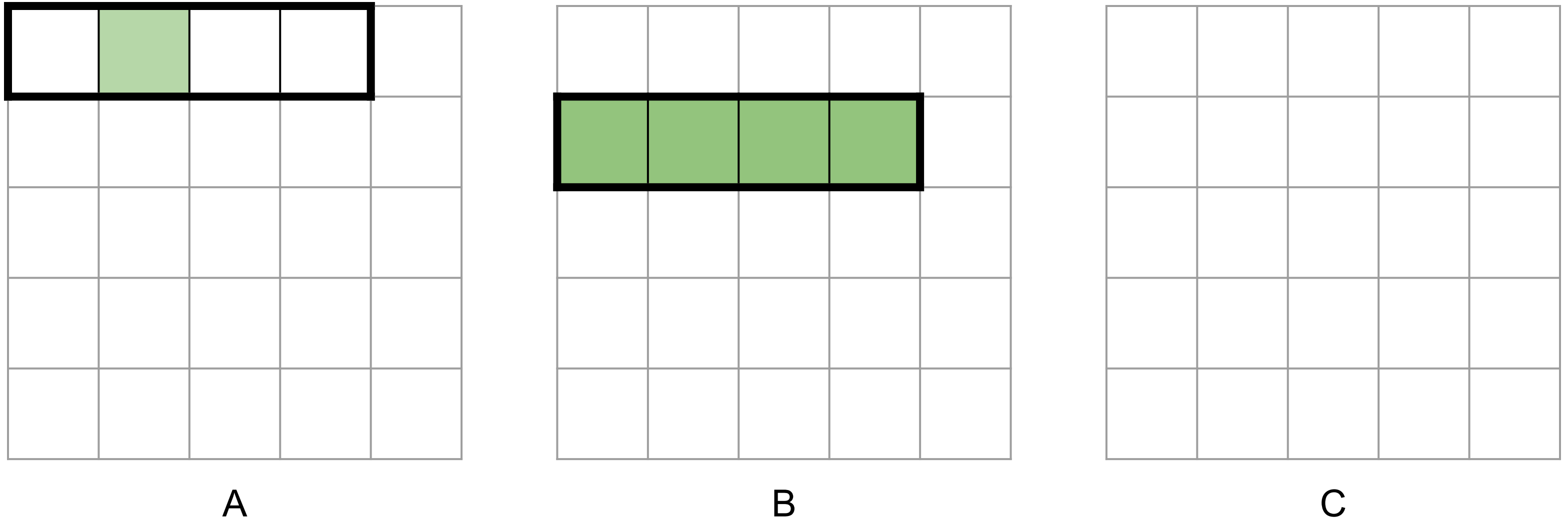
# Implementation: Sequential

- Cache-friendly using SIMD-v2



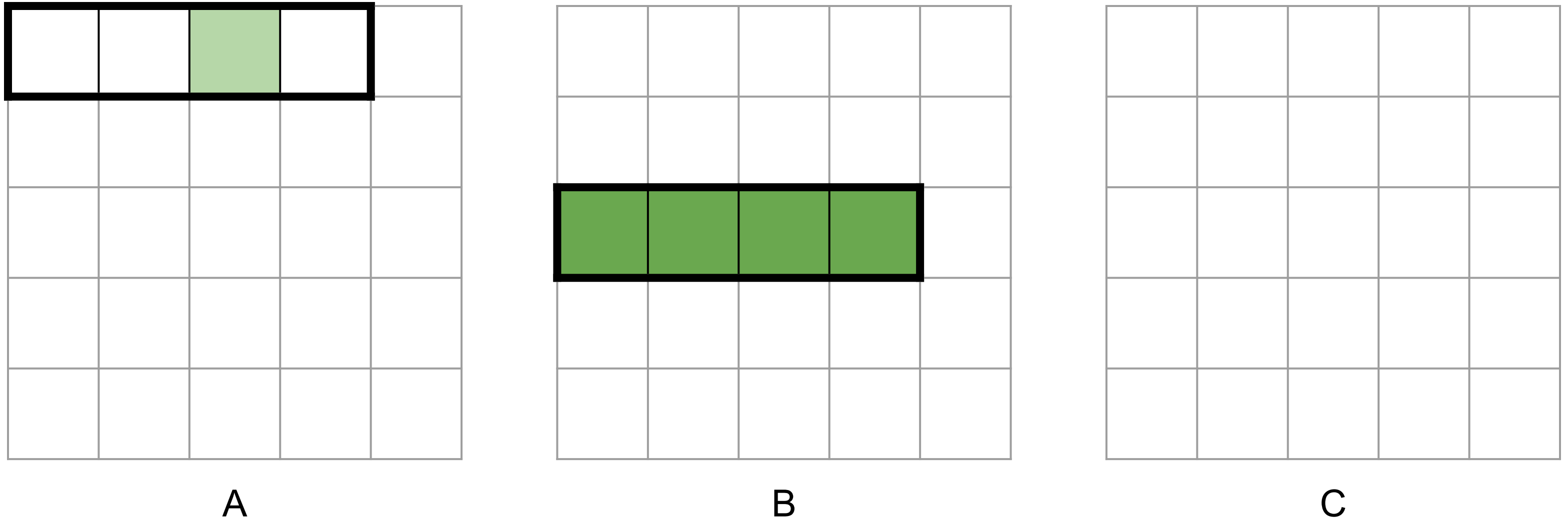
# Implementation: Sequential

- Cache-friendly using SIMD-v2



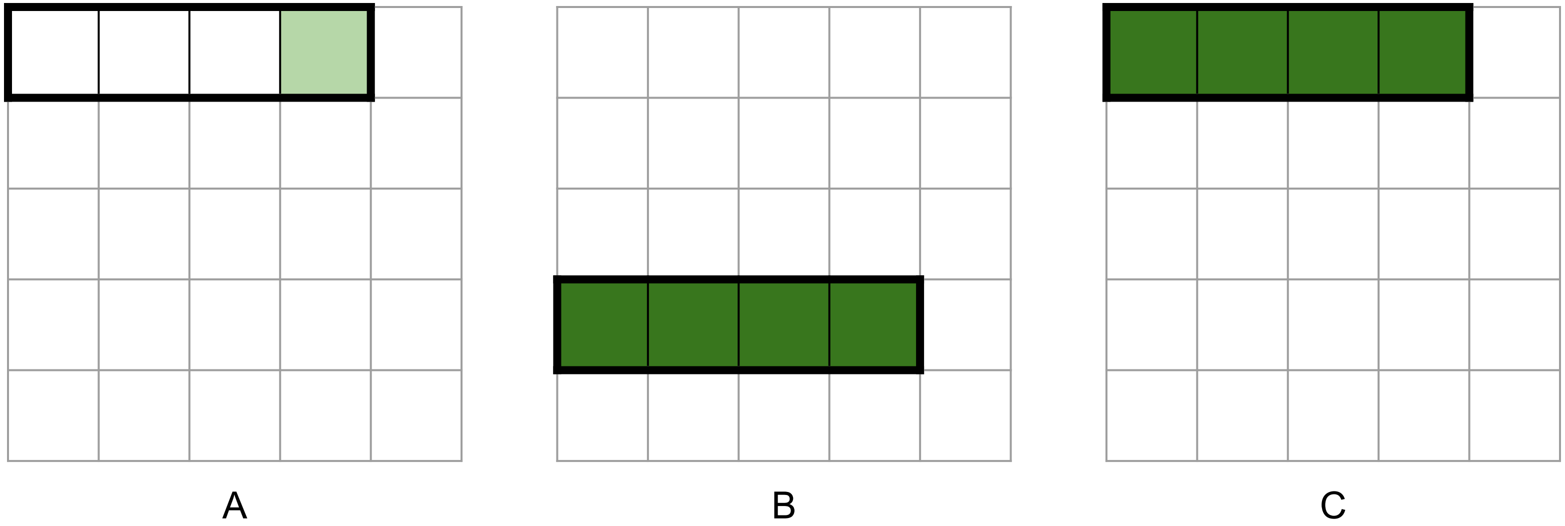
# Implementation: Sequential

- Cache-friendly using SIMD-v2



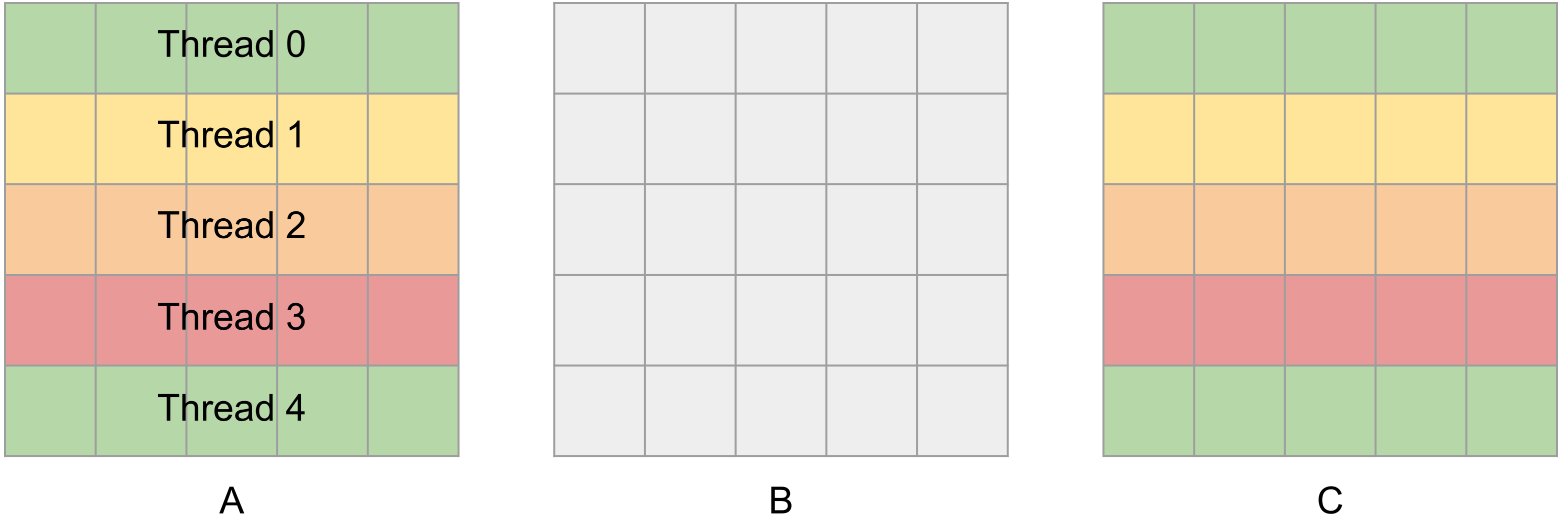
# Implementation: Sequential

- Cache-friendly using SIMD-v2



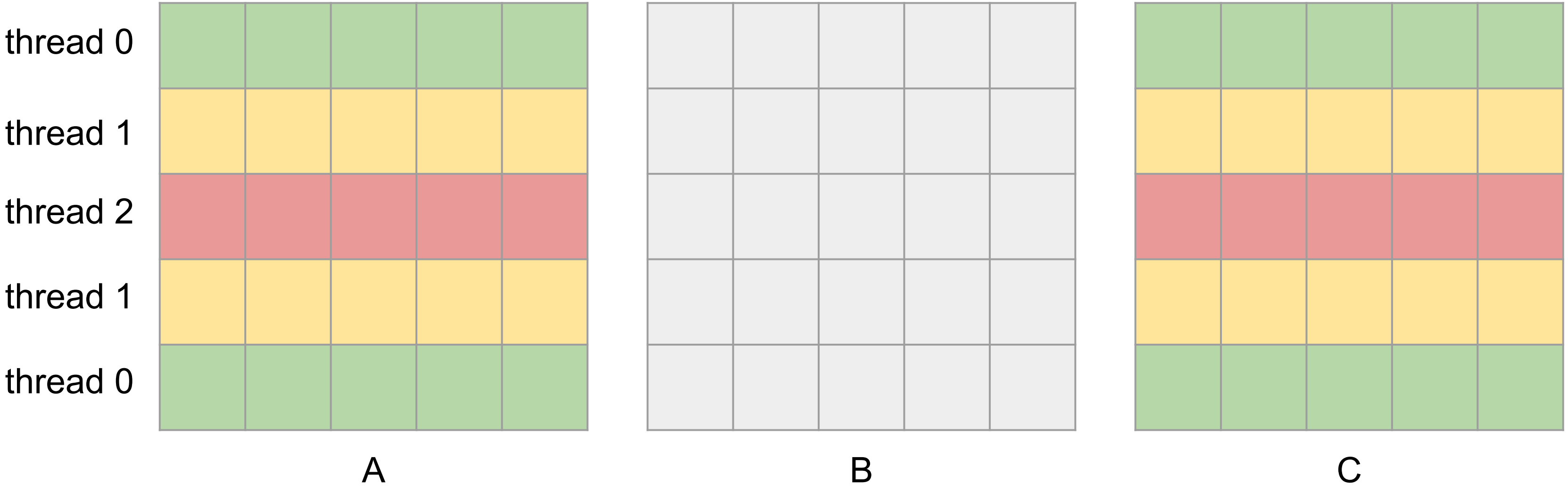
# Implementation: OpenMP

- Task Partition: row
  - Dynamic scheduling: # *pragma omp parallel for schedule(**dynamic**, 1)*
  - Static scheduling: # *pragma omp parallel for schedule(**static**, 1)*



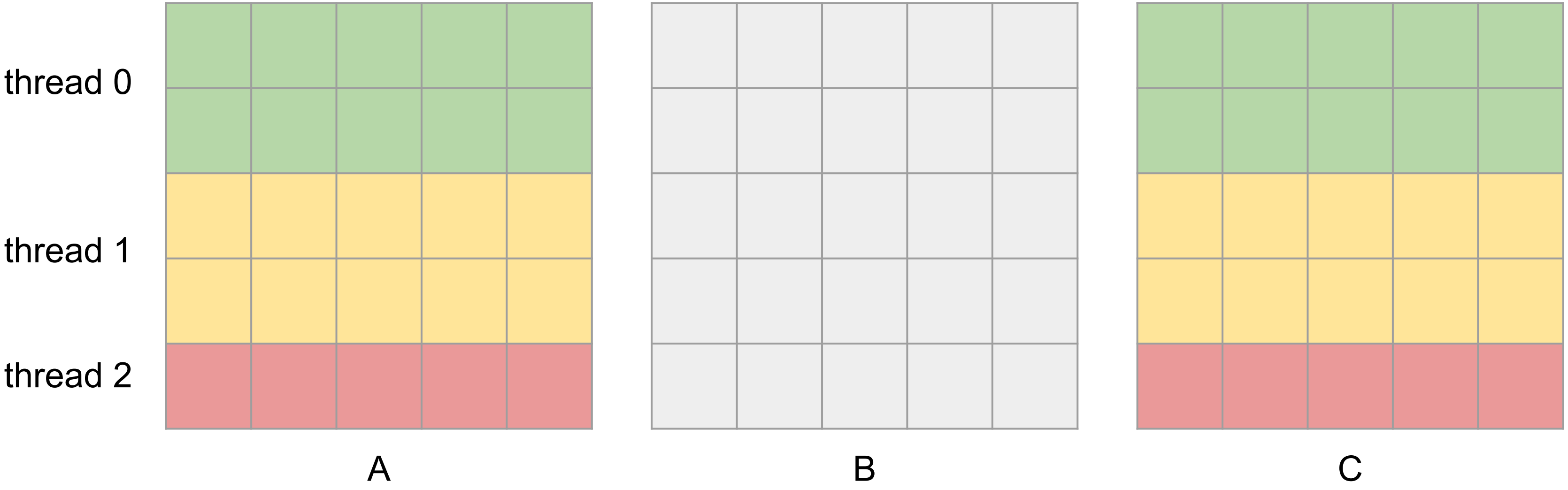
# Implementation: Pthread

- Dynamic Scheduling
  - Task partition: each thread will compute one row at a time
  - Mutex lock



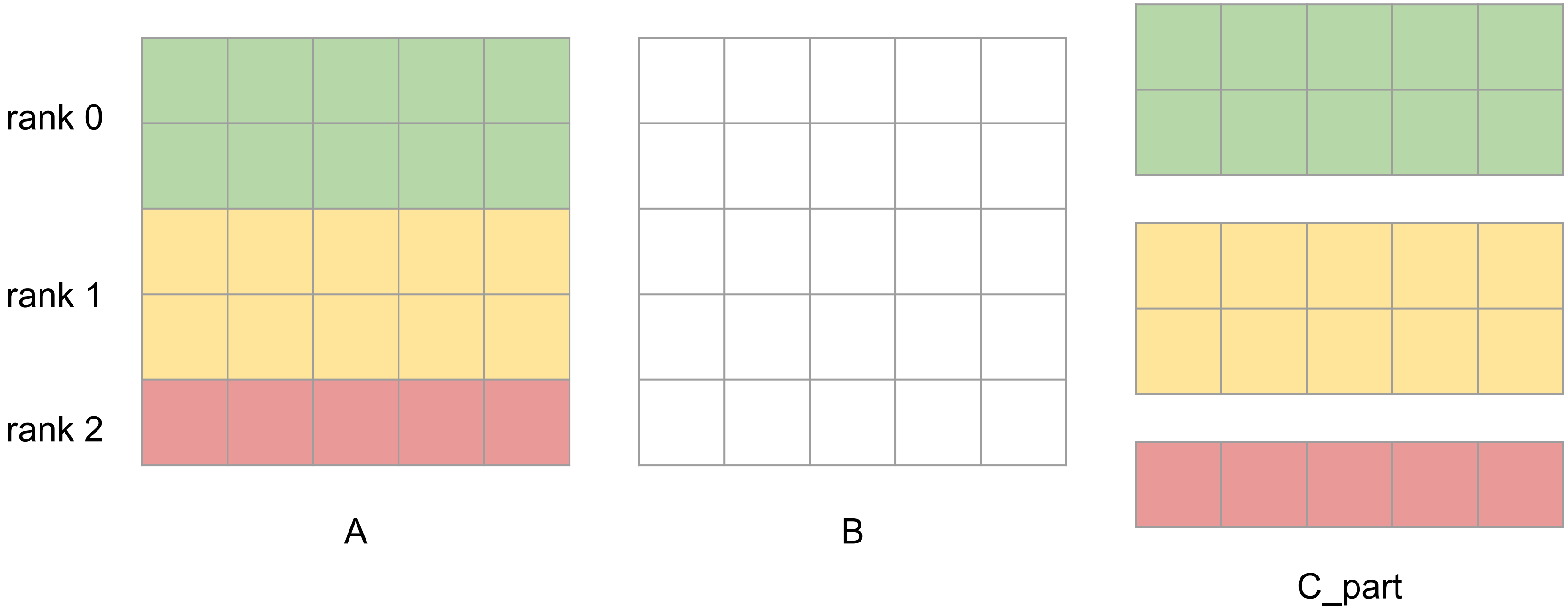
# Implementation: Pthread

- Static Scheduling
  - Task Partition: evenly distribute rows to each thread according to the total number of rows



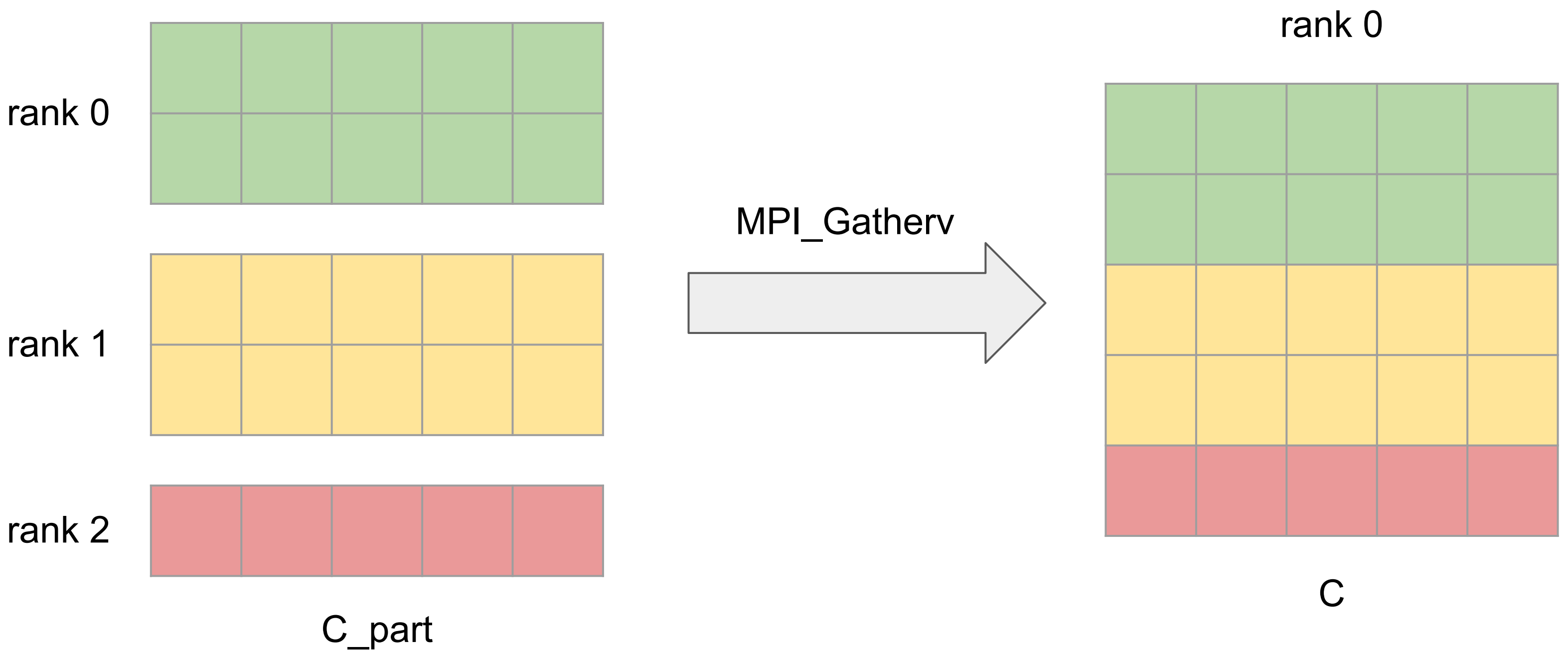
# Implementation: MPI

- How to assign data to each rank?



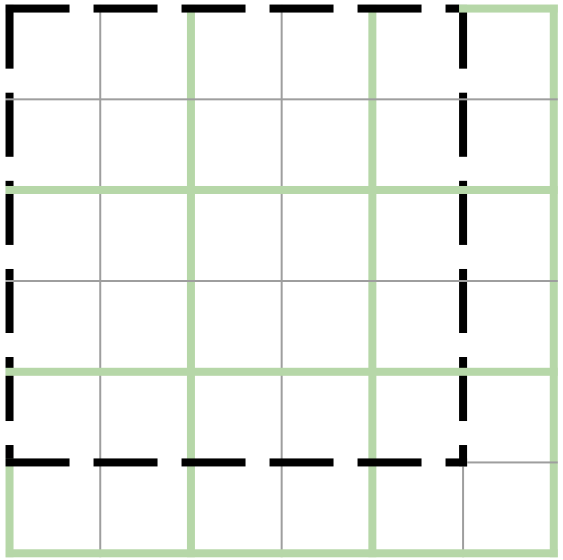
# Implementation: MPI

- How to gather data?

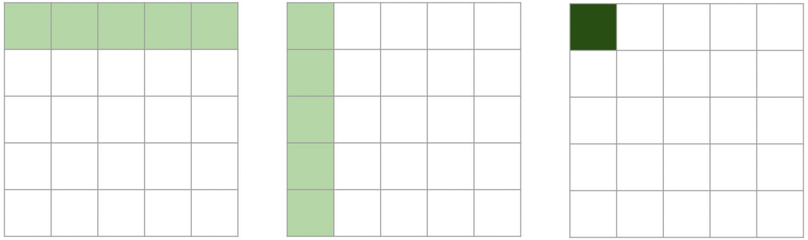


# Implementation: Single-GPU

- Accelerate naive sequential version
- Blk dim: dim3(B, B)
- Grid dim: dim3(n / B + 1, n / B + 1)
- e.g.
    - Given
        - n=5
        - B=2
    - Then
        - Grid dim: dim3(3, 3)
        - Blk dim: dim3(2, 2)



# Implementation: Single-GPU



- Code

```
__global__ void multiply_naive(int *d_a, int *d_b, int *d_c, int n) {
    int row_idx = blockIdx.y * blockDim.y + threadIdx.y;
    int col_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(row_idx >= n || col_idx >= n) return;
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += d_a[row_idx * n + i] * d_b[i * n + col_idx];
    d_c[row_idx * n + col_idx] = sum;
}
```

# Implementation: IO Improvement

- mmap-io
  - The memory mapping function Linux provides.
  - Map the file content to a segment of virtual memory.
  - Read and modify files by reading and modifying this segment of memory.

- Problem
  - This method is only suitable for files with small size in our implementation.
  - It seems that it will get segmentation fault error when the file size > 4MB.

# Experimental Results

# Experimental Results

- Environment of GPU version (NCHC)

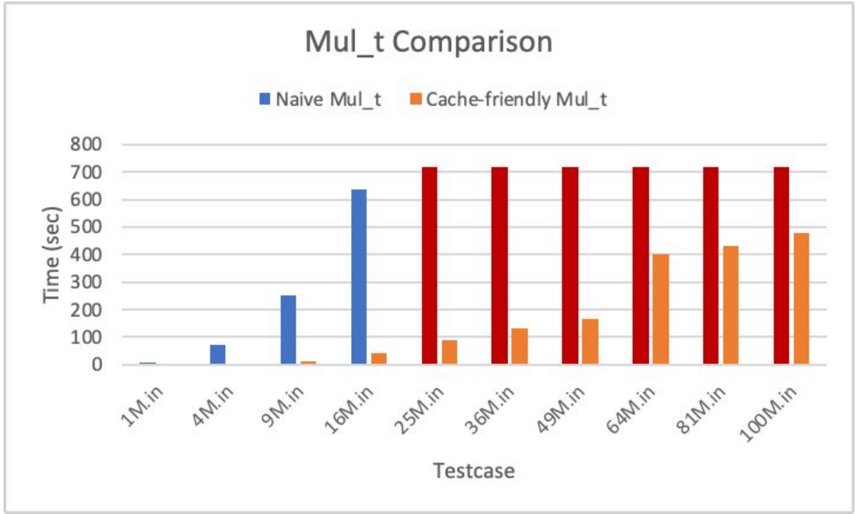
```
root@PP22:~/Parallel-Matrix-Multiplication# nvidia-smi -L
GPU 0: NVIDIA GeForce RTX 3070 (UUID: GPU-2e2f5d97-6554-4adb-6a0c-db488de6230e)
root@PP22:~/Parallel-Matrix-Multiplication# nvidia-smi
Sun Jan  8 08:32:48 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.57.02    Driver Version: 470.57.02    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:01:00.0 Off |                  N/A |
|  0%   31C    P8    13W / 220W |      1MiB /  7982MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

- Environment of Others (Apollo)

```
[pp22s69@apollo31 Parallel-Matrix-Multiplication]$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   40 bits physical, 48 bits virtual
CPU(s):                          12
On-line CPU(s) list:             0-11
Thread(s) per core:              1
Core(s) per socket:              6
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           44
Model name:                      Intel(R) Xeon(R) CPU           X5670  @ 2.93GHz
Stepping:                        2
Frequency boost:                 enabled
CPU MHz:                         2933.479
CPU max MHz:                     2933.0000
CPU min MHz:                     1600.0000
BogoMIPS:                        5866.98
Virtualization:                  VT-x
L1d cache:                       384 KiB
L1i cache:                       384 KiB
L2 cache:                        3 MiB
L3 cache:                        24 MiB
NUMA node0 CPU(s):               0-5
NUMA node1 CPU(s):               6-11
```

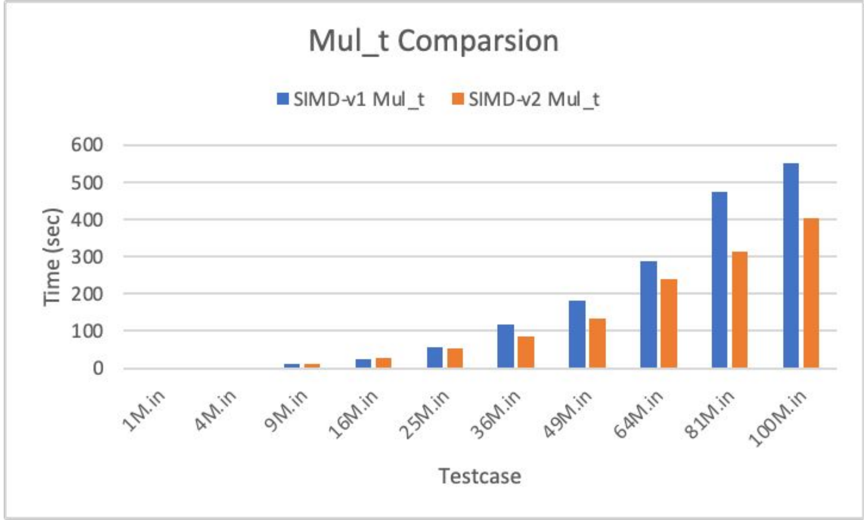
# Experimental Results

- Sequential version: Naive vs. Cache-friendly
  - Red bars represent TLE on apollo server
  - Almost 16 times faster on 16M.in



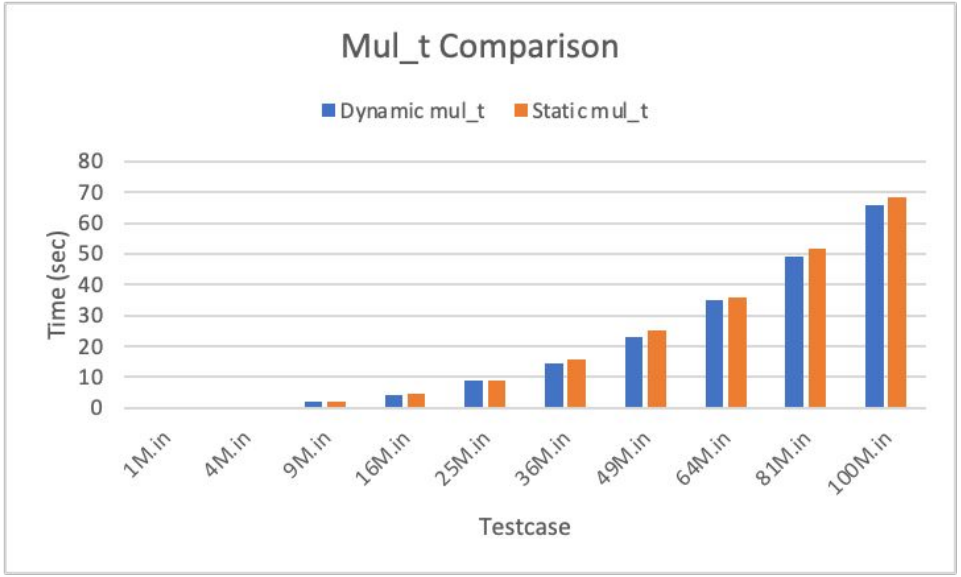
# Experimental Results

- Sequential version: Cache-friendly using SIMD-v1 vs. SIMD-v2
  - Average speedup is 1.23
  - Maximum speedup is 1.51 on 81M.in



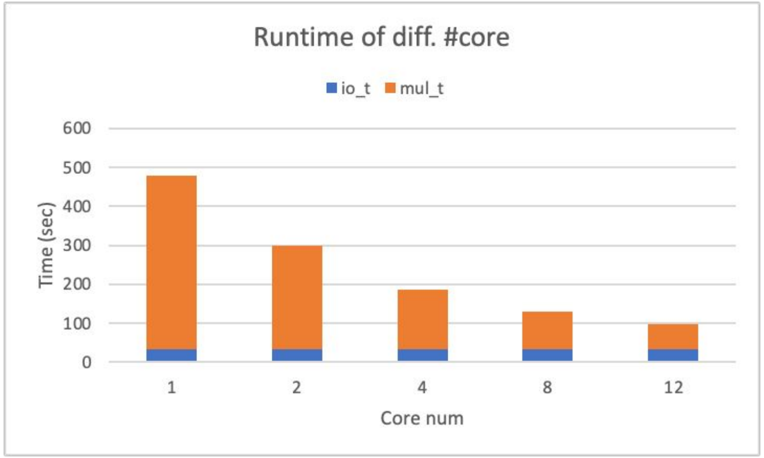
# Experimental Results

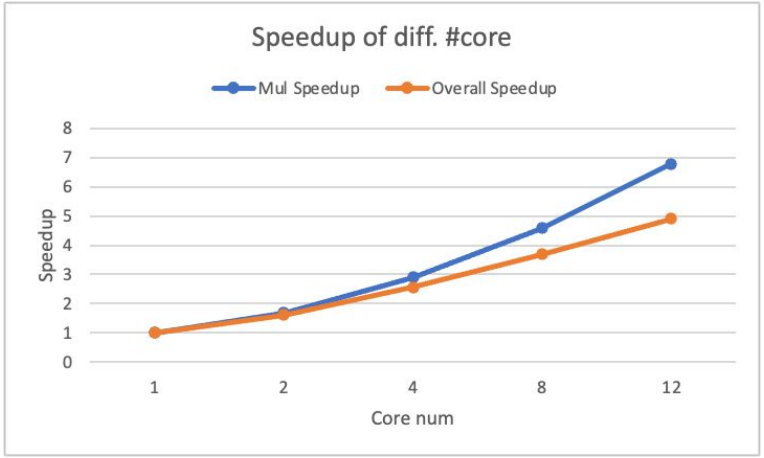
● OpenMP version: Dynamic Scehduling vs. Static Scheduling (12 cores)



# Experimental Results

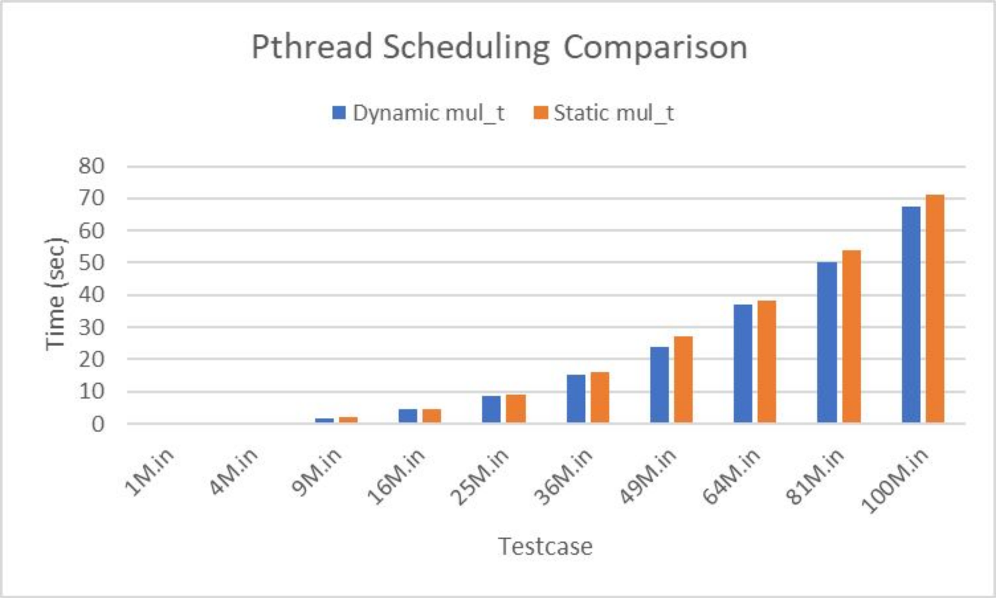
- OpenMP version: Runtime of diff. # Cores = {1, 2, 4, 8, 12}
    - Dynamic Scheduling, Testcase: 100M.in



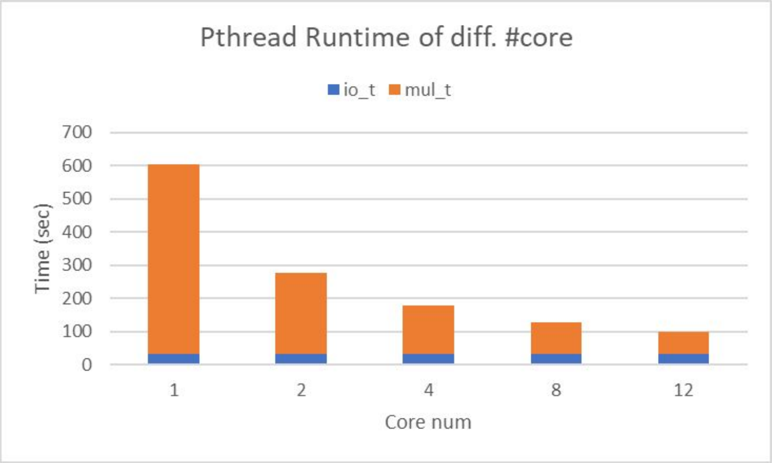
# Experimental Results

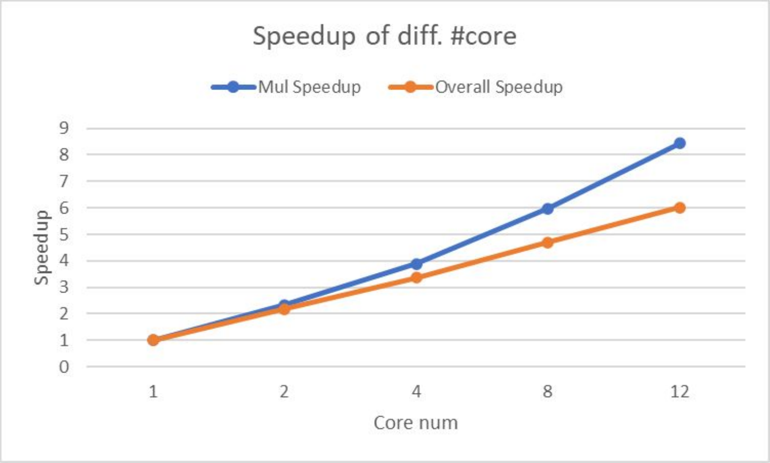
- Pthread version: Dynamic Scehduling vs. Static Scheduling (12 cores)



# Experimental Results

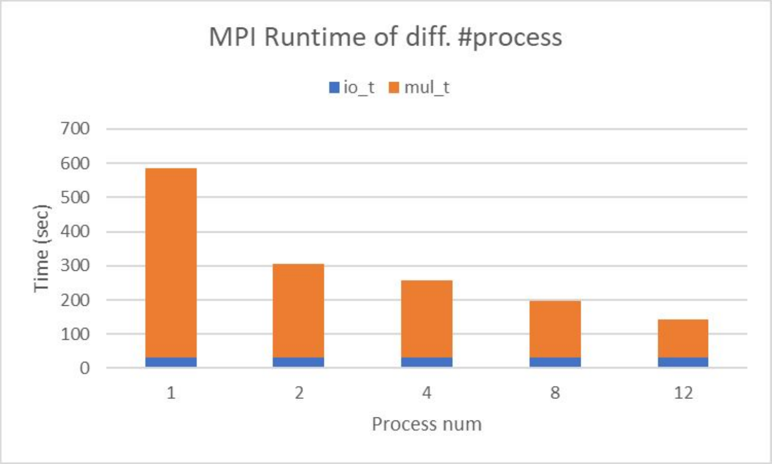
- Pthread version: Runtime of diff. # Cores = {1, 2, 4, 8, 12}
    - Dynamic Scheduling, Testcase: 100M.in

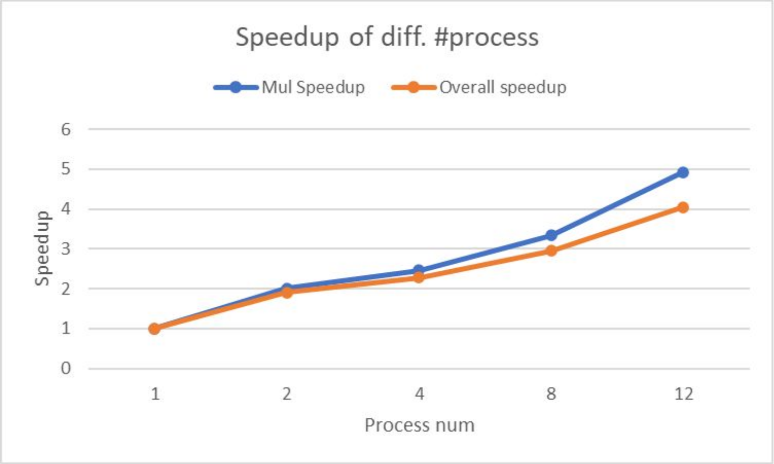


# Experimental Results

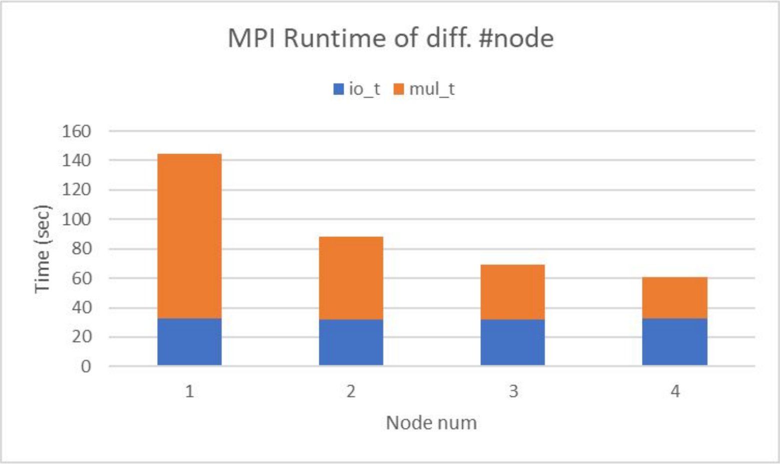
- MPI version: Runtime of diff. # proc = {1, 2, 4, 8, 12} under single-node
  - Testcase: 100M.in

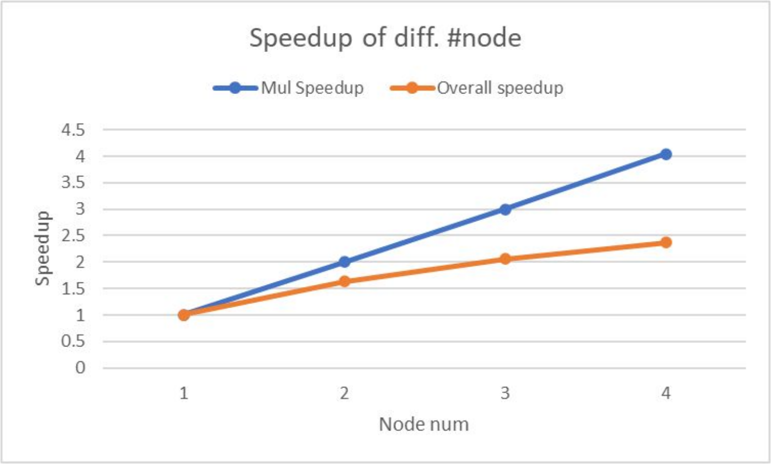


# Experimental Results

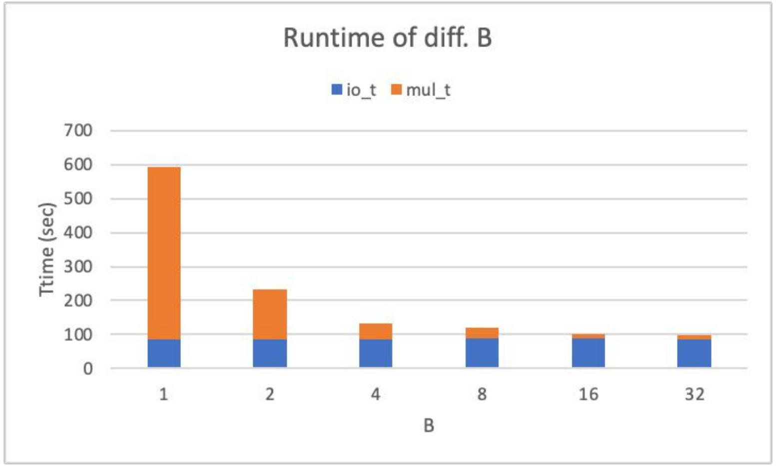
- MPI version: Runtime of diff. # nodes = {1, 2, 3, 4} with ppn=12
  - Testcase: 100M.in

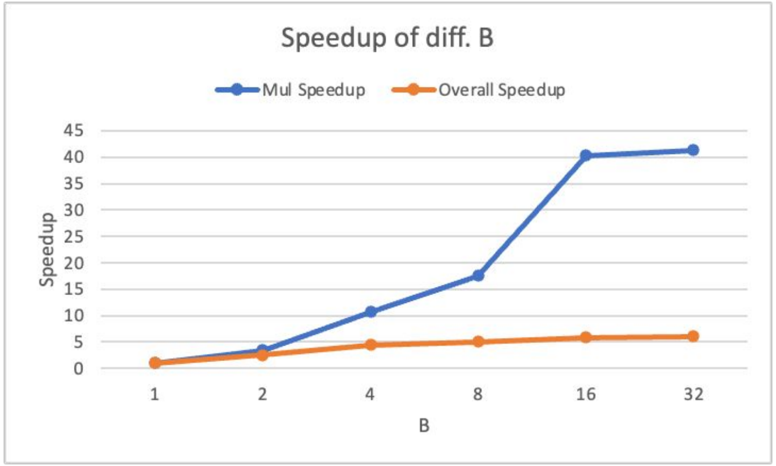


# Experimental Results

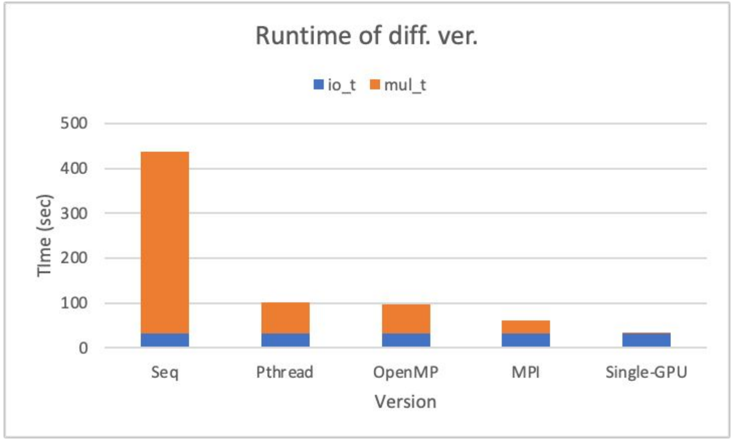
- Single-GPU version: Comparison of diff. B = {1, 2, 4, 8, 16, 32}
  - Testcase: 400M.in

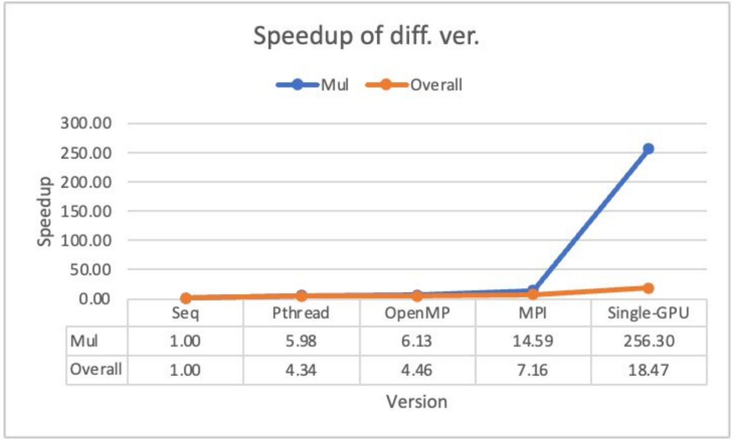


# Experimental Results

- Comparison of all implementation
  - Testcase: 100M.in
  - Seq: Cache + SIMD-v2
  - Pthread: Cache + SIMD-v2 + 12 cores
  - OpenMP: Cache + SIMD-v2 + 12 cores
  - MPI: Cache + SIMD-v2 + 4 nodes + (ppn=12)
  - GPU: Naive + (B=32)



# Future Works

# Future Works

- Strassen algorithm, time complexity = O(n^2.807).
- Since IO is the bottleneck of GPU version, it's necessary to accelerate IO.
- Combine MPI & OpenMP to achieve higher performance.
- GPU optimization
    - Memory coalescing
    - Submatrix multiplication
    - Shared memory

Thanks for Listening