1. OPENMP

OpenMP (Open Multi-Processing) is an API for parallel programming on shared-memory systems, widely used in C/C++, and Fortran. It provides a set of compiler directives (pragmas), library routines, and environment variables that make it easier to exploit multi-core CPUs without rewriting an entire program from scratch. OpenMP automatically manages thread creation, workload distribution, and synchronization. C++ is compulsory and Linux is recommended in this course. The following scripts are for installing OpenMP in Linux and MacOS environments.

```
# Linux installation
sudo apt install libomp-dev

# MacOS installation
brew install libomp llvm
```

To enable the compiler to recognize OpenMP directives, we must include the "-fopenmp" flag when compiling our programs. The remaining procedure for compiling and running a C++ program remains unchanged.

```
# Compile with g++
g++ -fopenmp <PROGRAM>.cpp -o <PROGRAM_NAME>

# Compile with clang
clang -fopenmp <PROGRAM>.cpp -o <PROGRAM_NAME>
```

A typical OpenMP program consists of two parts, one is sequential regions and other is parallel regions. Parallel regions in OpenMP are defined using the "#pragma omp parallel" syntax. This indicates that the following block of code (enclosed in braces) is executed by multiple threads in parallel. Each thread runs the same code within the region, but may work on different portions of the data depending on additional clauses which will be mentioned later.

```
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        // parallel region
        int tid = omp_get_thread_num();
        int n   = omp_get_num_threads();
        printf("Hello from thread %d of %d\n", tid, n);
    }
    return 0;
}
```

By default, each parallel region in OpenMP is executed using all of the available hardware threads on the machine. However, the number of threads can be explicitly controlled in two ways, which are mainly used.

- **Environment variable (OMP_NUM_THREADS)**. sets the number of threads globally, affecting all parallel regions in the program.

```
# Execute with <NUM_THREAD> threads
OMP_NUM_THREADS=<NUM_THREAD> ./<PROGRAM_NAME>
```

- **Clause (num_threads(n))**. applied directly in the directive, which overrides the global setting but only for that specific parallel region.

```
#pragma omp parallel num_threads(<NUM_THREAD>)
{
    // parallel region
}
```

Variables declared outside of a parallel region are shared among all threads, in contrast, variables declared inside a parallel region are private to each thread, so every thread has its own copy that is not visible to the others. However, we can change this default behaviors by utilizing OpenMP clause such as *shared, private, firstprivate, etc*. When working with shared variables, race conditions must always be considered. To ensure correct results, OpenMP provides synchronization mechanisms such as the *atomic* (for simple operations) , *critical* (for protecting larger code sections) , *etc*.

```
#include <stdio.h>
#include <omp.h>

int main() {
    int share_var = 0; // share variable across all threads

    #pragma omp parallel
    {
        int private_var; // private variable for each thread
        int tid = omp_get_thread_num();

        private_var = tid;
        #pragma omp atomic
        share_var++;

        printf("private=%d from thread %d\n", private_var, tid);
    }
    printf("share=%d\n", share_var);

    return 0;
}
```

For data parallelism, OpenMP provides the “for” clause, which automatically partitions loop iterations among the available threads. This allows each thread to work on a different subset in parallel, which greatly improves performance on large iterative tasks.

OpenMP also supports the thread-safe “reduction” clause to efficiently handle operations that combine results across threads (e.g., summation, product, minimum, or maximum). With reduction, each thread maintains a private copy of the variable during the loop execution, and finally combines them into a single final result.

```
#include <stdio.h>
#include <iostream>
#include <omp.h>

int main() {
    const long N = 1000000000L;
    int *arr = new int[N];
    for (long i = 0; i < N; ++i) arr[i] = 1.0;

    double s0 = omp_get_wtime();
    int seq_sum = 0;
    for (long i = 0; i < N; ++i) {
        seq_sum += arr[i];
    }
    double s1 = omp_get_wtime();

    // parallel bottle neck
    double p0 = omp_get_wtime();
    int par_sum = 0.0;
    #pragma omp parallel for
    for (long i = 0; i < N; ++i) {
        #pragma omp atomic
        par_sum += arr[i];
    }
```

```
    double p1 = omp_get_wtime();

    // efficient parallel sum
    double r0 = omp_get_wtime();
    int reduce_sum = 0.0;
    #pragma omp parallel for reduction(+:reduce_sum)
    for (long i = 0; i < N; ++i) {
        reduce_sum += arr[i];
    }
    double r1 = omp_get_wtime();

    printf("sequential sum = %d in %.3fs\n", seq_sum, s1 - s0);
    printf("parallel sum = %d in %.3fs\n", par_sum, p1 - p0);
    printf("reduce sum = %d in %.3fs\n", reduce_sum, r1 - r0);

    delete[] arr;
    return 0;
}
```

OpenMP allows programmers to explicitly control how loop iterations (jobs) are distributed among threads through the schedule clause. While OpenMP supports several scheduling modes [1], the most commonly used are:

- **Static:** Iterations are divided into equal-sized chunks and assigned to threads in advance, before the loop begins. Achieves lowest scheduling overhead but may lead to workload imbalance among threads.
- **Dynamic:** Iterations are divided into chunks, but threads request new chunks as soon as they finish their current ones. Costs higher scheduling overhead and potentially worse locality than other modes.
- **Guided:** Similar to dynamic scheduling, but chunk sizes start large and decrease over time. This mode is a compromise solution between Static and Dynamic.

The following code demonstrates how to implement matrix multiplication in OpenMP using the most naïve approach. The "collapse" clause is applied to effectively flatten the nested iteration space into a single loop before dividing it among threads. This allows OpenMP to schedule the combined iterations more evenly across threads.

```
#include <stdio.h>
#include <iostream>
#include <omp.h>

#define N 1000

int compute_cell(int** A, int row, int col) {
    int sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += A[row][k] * A[k][col];

    return sum;
}

double sequential(int** matrix, int** result) {
    double base = omp_get_wtime();
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            result[i][j] = compute_cell(matrix, i, j);

    return omp_get_wtime() - base;
}

double parallel(int** matrix, int** result) {
    double base = omp_get_wtime();
    #pragma omp parallel for schedule(static, 1000) collapse(2)
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
```

[1] For more details: https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP/.

```
            result[i][j] = compute_cell(matrix, i, j);

    return omp_get_wtime() - base;
}

int main(void) {
    int **matrix = new int*[N];
    int **result = new int*[N];
    for (int i = 0; i < N; ++i) {
        matrix[i] = new int[N];
        result[i] = new int[N];
    }

    std::cout << "sequential time: " << sequential(matrix, result) << "\n";
    std::cout << "parallel time: " << parallel(matrix, result) << "\n";

    for (int i = 0; i < N; ++i) {
        delete[] matrix[i];
        delete[] result[i];
    }

    return 0;
}
```

2. CONVOLUTION

Convolution plays an essential role across signal and image processing, computational science, and modern machine learning. At its core, it's a local, translation-invariant operation that mixes each data point with its neighbors using a small kernel (filter), letting us emphasize or suppress specific spatial or temporal patterns.

Discrete convolution of a 2D matrix (e.g., an image) with a kernel produces another matrix where each output pixel is a weighted sum of a local neighborhood in the input. Given an input $I \in R^{H \times W}$ and a kernel $K \in R^{M \times N}$, the 2D convolution at location (i, j) can be calculated as the formula below:

$$(I * K)[i, j] = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} I[i - u, j - v] K[u, v]$$

In another word, 2D convolution is just a sequence of local matrix multiplications between the kernel and small patches of the original input. As shown in Figure 1 [2], at each position, a sub-matrix is extracted from the input, element-wise multiplied with the kernel, and then summed up to produce a single output value. Repeating this process across all possible positions generates the full output feature map.

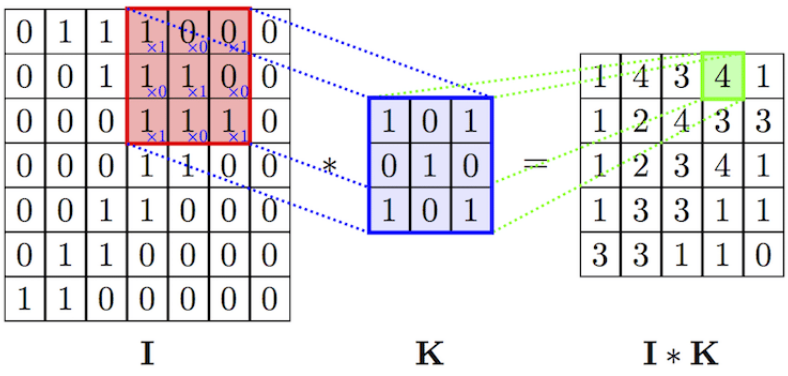

Figure 1: Illustration of a 2D convolution.

[2] Source: Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques.

When the kernel overlaps the boundary of an image during convolution or correlation, some of its elements fall outside the valid range of indices, resulting in a smaller output. A common approach is zero padding, where the image is surrounded with zeros so that the kernel can still be applied at the edges, preserving the original size, as illustrated in Figure 2 [3]. Another alternative is replicate padding, where the nearest edge value is copied outward, which will be used later in the exercises.

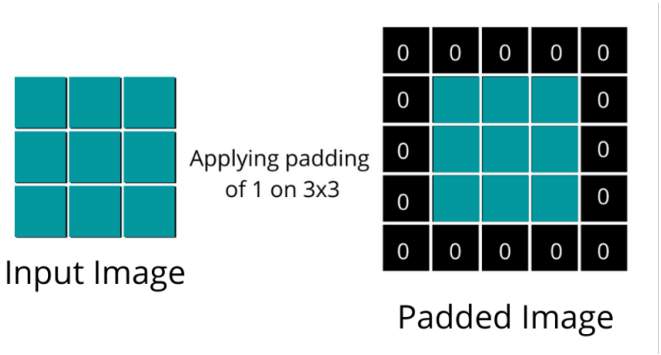

Figure 2: Padding in a 2D convolution.

The stride determines the step size by which the kernel's sliding window moves across the input during convolution. With a stride of $s = 1$ (usually is the default behavior), the kernel shifts one element at a time, increasing the stride to $s > 1$ means the kernel skips intermediate positions, effectively reducing the number of computations and producing a smaller output map, as depicted in Figure 3 [4].

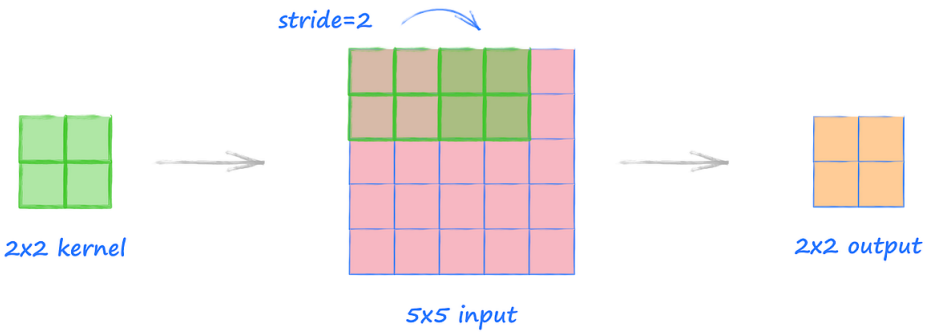

Figure 3: Stride=2 in a 2D convolution.

Convolution and correlation are actually "two sides of the same coin" and often used interchangeably because they both involve sliding a kernel across an input and computing weighted sums at each position; the only difference is that convolution first flips the kernel horizontally and vertically (a 180° rotation) before applying, while correlation uses the kernel as it is. For symmetric kernels such as Gaussian blur or Laplacian, this flip makes no difference, so the two operations yield identical results.

[3] Source: https://www.almabetter.com/bytes/articles/what-is-padding-in-convolutional-neural-network
[4] Source: https://medium.com/@akp83540/stride-of-a-convolution-operation-05bc7365d91c

3. EXERCISES

On 3 September 2025, China staged a Victory Day parade marking the 80th anniversary of its victory over Japan and the end of World War II. Notably, China unveiled for the first time the DF-61 intercontinental ballistic missile (ICBM) – described as one of its most advanced systems – with an estimated range of 12,000 to 15,000 kilometers. Rumors suggest that the DF-61 is equipped with seven nuclear warheads, each with an estimated yield of around 650 kilotons – approximately 300 times more powerful than the "Little Boy" bomb dropped on Hiroshima.

In this exercise, students are required to estimate the heat diffusion process created from a nuclear explosion. Heat diffusion is no more than just a 2D convolution with a kernel $K = \begin{pmatrix} 0.05 & 0.1 & 0.05 \\ 0.1 & 0.4 & 0.1 \\ 0.05 & 0.1 & 0.05 \end{pmatrix}$ that spreads heat to nearby cells.

Students are provided with a matrix 4000×4000, representing the heat intensity of a city (around 1600 km^2) where each cell corresponds to a 100 m^2 area with 10 meters of length and width. The matrix models the aftermath of a nuclear disaster, with peak temperatures at the center that gradually decrease toward the surface, while surrounding areas maintain a baseline temperature of 30°C (use 30 for padding instead of zero). In particular, students are required to:

1. Perform sequential convolution for at least 100 iterations.
2. Implement at least two different parallel convolution algorithms using OpenMP, each running for at least 100 iterations.
3. Conduct a time comparison between the sequential and parallel implementations.

** **Submission:** Works must be done individually.

- Compress all necessary files as **"Lab_1_<Student ID>.zip"**.
- Students must submit and demonstrate their results in a detailed report.