High-Performance Computing 3.
Master CSMI - Strasbourg University

Bérenger Bramas
Inria Nancy, France
ICube Laboratory, France
University of Strasbourg, France

Year 2025/2026

Contents

Chapter 1

Document Organization

High-performance computing (HPC) is a subfield of computer science dedicated to the development and application of parallel computing techniques for efficiently solving complex problems. This document serves as the primary reference for the course *High-Performance Computing 3*, which is part of the CSMI master's program at the University of Strasbourg. This course covers various aspects of HPC, including shared-memory architectures, distributed-memory systems, CPU and GPU optimizations, heterogeneous computing, and C++ programming for parallelism.

The target audience for this course includes master's students in applied mathematics with a focus on computer science. The course assumes a solid foundation in programming, algorithms, and data structures, as well as familiarity with parallel computing concepts.

Effective programming for high-performance computing requires a deep understanding of the underlying hardware architectures and the ability to write efficient, parallel code. The aim of the course is to equip students with the knowledge and skills necessary to design and implement high-performance algorithms and applications.

The document is organized into the following chapters:

- **Execution Model:** Introduction to the fundamental concepts of parallel computing, including execution models, processes, threads, and their interactions.
- **Shared Memory:** Architecture, algorithms, and implementation techniques for shared-memory systems, in which multiple processors access a common memory space.
- **Distributed Memory:** Principles of distributed-memory systems, where each processor has local memory and communicates with others via message passing.
- **CPU Optimization:** Techniques for optimizing code for CPU architectures, such as using vectorization and improving cache utilization.
- **GPU Optimization:** Programming and optimization strategies to exploit the massive parallelism of GPUs.
- **Heterogeneous Computing:** Integration of diverse computing architectures (CPU and GPU) to maximize overall performance.
- **Algorithms:** Design and analysis of parallel algorithms, including sorting, searching, and numerical methods.
- **C++ for Parallel Computing:** Best practices and language features for writing high-performance C++ code with a focus on parallelism.
- **Performance Measurement:** Methods for profiling and benchmarking parallel applications to analyze and improve performance.

Whenever possible, we introduce simplified models to illustrate core concepts and mechanisms. These models are not exhaustive representations of real systems but serve to clarify key ideas and algorithms before delving into implementation details. The implementations will also be studied deeply during the practical sessions.

Chapter 2

Execution Model

This chapter reviews how operating systems manage program execution and hardware resources. Understanding these principles is essential for writing efficient parallel programs.

2.1 From Program to Execution

A program is a sequence of operations that we ask the computer to perform. Its execution consists of a series of steps, each executing a machine instruction. The definition of an operation depends on the programming language, while the resulting instructions depend on the hardware.

Translating code from a high-level language into machine instructions is called *compilation*. The compilation pipeline typically consists of lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. Each stage transforms the program into an intermediate representation or directly into machine code.

Computers execute *machine language*, which consists of binary instructions that the processor understands directly. Since machine code is not human-readable, we use high-level languages (e.g., C++, Python) to write programs, and then translate them into machine language via the compilation process.

Compilation: The process of transforming source code from one language into another, typically from a high-level language to machine language, while preserving the program's semantics.

Instruction: A low-level operation that the processor can execute, such as loads, stores, arithmetic operations, and control-flow operations.

Abstraction: The concept of hiding lower-level details to simplify interaction with a system.

Programming Language: A formal language defined by a grammar, used to express computations as a sequence of operations.

Binary (Executable): The machine-level code that can be loaded and executed directly by the computer.

2.2 Execution Model

When a binary is launched, the operating system creates a *process*, an instance of the program in execution. The process begins at a defined entry point (usually the `main` function) and executes instructions in sequence, except where control-flow constructs (loops, conditionals) alter the order.

Each execution of the same binary spawns a separate process, with its own memory space and resources, allowing independent execution.

The operating system includes a *scheduler* that allocates CPU time to processes, ensuring fair execution. By rapidly switching the CPU between processes (context switching), the scheduler creates the illusion of simultaneous execution on a single core.

Process: An isolated execution environment consisting of code, data, and system resources.

Thread: A lightweight execution unit within a process that shares the process's memory and resources.

Concurrency: The property of systems in which multiple tasks are in progress at overlapping time periods, possibly interleaved or in parallel.

Parallelism: The simultaneous execution of multiple tasks on separate processors or cores.

Scheduler: The OS component responsible for deciding which process or thread runs on the CPU at any given time.

CPU Core: An independent processing unit within a CPU capable of executing instructions.

Context Switch: The operation of saving the state of a running process or thread and loading the state of another.

Entry Point: The address or symbol where the OS transfers control to start program execution.

Control Flow: The order in which instructions are executed, determined by the program's structure and flow constructs.

2.3 Interaction Between Processes and Threads

2.3.1 Processes

Concurrent processes often need to communicate and share data. Operating systems provide inter-process communication (IPC) mechanisms (e.g., pipes, shared memory segments), but in HPC we typically rely on the Message Passing Interface (MPI), which standardizes communication across processes on single or multiple nodes. MPI supports message passing, synchronization, and collective operations.

2.3.2 Threads

Threads within the same process share memory, enabling efficient communication but requiring careful synchronization. Common synchronization primitives include mutexes, semaphores, and condition variables. Additionally, atomic operations allow indivisible updates to shared data without locks.

2.4 Key Concepts

After this chapter, you should be familiar with:

- The definitions of programs, processes, threads, and related terminology.
- The compilation process from high-level code to machine instructions.

- How operating systems manage execution via scheduling and context switching.
- The distinction between concurrency and parallelism.
- Basic IPC mechanisms and threading synchronization methods.

2.5 Exercices

Draw a diagram illustrating

- the compilation process, showing the stages from source code to machine code. Include examples of transformations at each stage.
- the execution model of a program, including processes, threads, OS scheduler, and their interactions. Label the components and explain their roles.
- the hardware architecture, showing the CPU and cores, and how they interact with processes and threads.

Chapter 3

Shared Memory Parallelization

Overview

This chapter focuses on shared-memory parallelization, a programming model that allows multiple threads to access a common memory space. This model is commonly used in multi-core processors, where each core can execute threads concurrently. Shared-memory parallelization is particularly useful for applications that require fast communication and synchronization between threads. In this chapter, we discuss the architecture of shared-memory systems, the algorithms used for parallelization, and the implementation details.

3.1 Architecture

A central processing unit (CPU) is the hardware component that performs the basic arithmetic, logic, control, and input/output operations specified by a program's instructions. The CPU executes operations at a speed determined by its clock frequency, measured in cycles per second (hertz). Clock frequency is limited by the CPU's physical properties and manufacturing technology; since frequency scaling has plateaued, other techniques such as vectorization and parallelization are used to improve performance.

Parallelization consists of splitting work into several tasks that can execute concurrently, while ensuring correct program execution.

CPU: The central processing unit, consisting of an arithmetic logic unit (ALU), control unit, and registers.

Register: A small, fast storage location within the CPU used to hold temporary data and instructions during execution.

Clock frequency: The speed at which the CPU executes instructions, measured in hertz (Hz). Higher frequencies yield faster execution, in most cases.

Vectorization: The process of converting scalar operations into vector operations that execute simultaneously on multiple data elements.

Parallelization: The process of dividing a task into smaller subtasks that can be executed concurrently on multiple processors or cores.

Different approaches can be used to create threads, but in the end they behave similarly. The operating system creates a thread and makes it start execution at a designated entry point. Conceptually, a thread is like a process but shares the same memory space as its parent process.

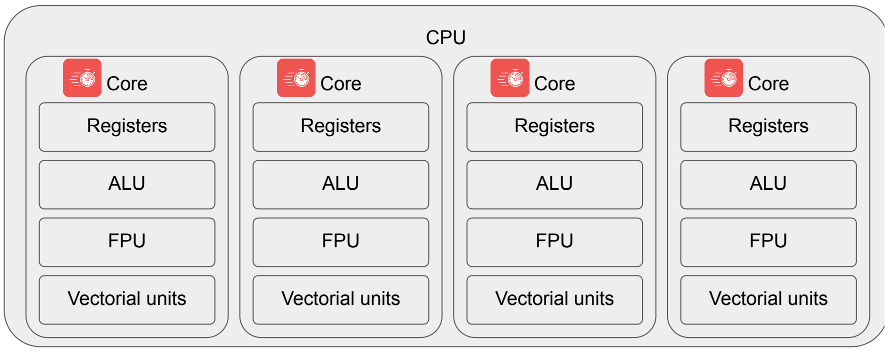


Figure 3.1: CPU architecture.

3.2 Algorithms

In this course, we first use a simplified algorithmic model to describe parallelization. We will not delve into hardware architecture details; instead, we focus on the algorithms and their implementation, and finally see how to implement them in C++.

There are two main parallelization strategy in shared memory:

- **Fork-Join:** A single thread starts the work and, when the problem size exceeds a threshold, divides it into two subtasks. Those subtasks are executed in parallel by other threads, and once both complete, their results are combined.
- **Data Parallelism:** Each thread processes a disjoint chunk of the input independently. After all threads finish, they synchronize and merge their partial results.

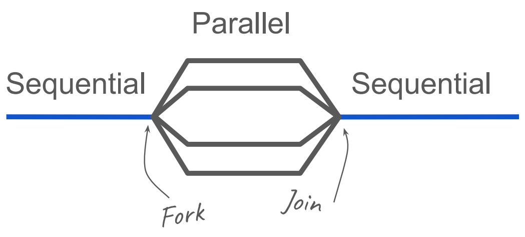


Figure 3.2: Fork-join model.

3.2.1 Shared-Memory Algorithmic Model

- `@start_thread_and_run(n){scope}`: Creates $n - 1$ new threads that execute the code in `scope`. The original thread also executes `scope`, so a total of n threads run it. Variables declared inside `scope` are private to each thread, while variables declared outside are shared (unless specified otherwise).
- `@one_at_a_time{scope}`: Ensures that only one thread at a time can execute the code in `scope`.
- `@in_order_execution{scope}`: Ensures that threads execute the code in `scope` in a specific order (based on their ids), one at a time.
- `@wait_for_all`: Blocks until all threads have reached the join point before proceeding.
- `@create_task(l)[copies]{scope}`: Creates a new task and inserts it into task list `l`. Variables listed in `copies` are passed by value, making them private to each task.

- `@wait_for_task(l)`: Blocks until all tasks in list `l` have completed (i.e., the list is empty).
- `@consume_task_until_finished(l)`: Repeatedly consumes tasks from list `l` until it is marked finished, then returns.
- `@consume_task_until_empty(l)`: Repeatedly consumes tasks from list `l` and returns when it becomes empty.
- `@marked_as_finished(l)`: Marks task list `l` as finished (no more tasks will be inserted), allowing `@consume_task_until_finished(l)` to return.
- `@make_private{declarations}`: Makes the declared variables private to each thread, preventing interference.
- `@thread_id`: Returns the ID of the current thread.
- `@thread_count`: Returns the number of active threads in the current parallel region (i.e., the n threads created by the most recent call to `@start_thread_and_run(n-1)`).
- `[lstart, lend] = @get_chunk(start, end, step, split)`: Divides the iteration space $[start, end)$ (with stride `step`) into `split` chunks and returns the start and end indices of the chunk assigned to the current thread.

3.2.2 Examples

To have each thread print its ID at the beginning and end of the scope:

```
@start_thread_and_run(n) {
    id = @thread_id;
    print("Thread %d started\n", id);
    print("Thread %d finished\n", id);
}
```

This produces n "started" and n "finished" messages, but their order is unpredictable.
To ensure all "started" messages appear before any "finished" messages, we can insert a barrier:

```
@start_thread_and_run(n) {
    id = @thread_id;
    print("Thread %d started\n", id);
    @wait_for_all;
    print("Thread %d finished\n", id);
}
```

To avoid interleaved output when multiple threads print simultaneously, we can use mutual exclusion:

```
@start_thread_and_run(n) {
    id = @thread_id;
    @one_at_a_time {
        print("Thread %d started\n", id);
    }
    @wait_for_all;
    @one_at_a_time {
        print("Thread %d finished\n", id);
    }
}
```

To enforce ordered execution of the prints:

```
@start_thread_and_run(n) {
    id = @thread_id;
    @in_order_execution {
        print("Thread %d started\n", id);
    }
    @wait_for_all;
    @in_order_execution {
        print("Thread %d finished\n", id);
    }
}
```

If only the total number of prints matters, tasks can be used:

```
task_list l;
@start_thread_and_run(n) {
    if (@thread_id == 0) {
        for (i = 1; i < n; ++i) {
            @create_task(l) {
                print("Thread %d started\n", @thread_id);
            }
        }
        @consume_task_until_empty(l);
        for (i = 1; i < n; ++i) {
            @create_task(l) {
                print("Thread %d finished\n", @thread_id);
            }
        }
        @consume_task_until_empty(l);
        @marked_as_finished(l);
    } else {
        @consume_task_until_finished(l);
    }
}
```

Using `@make_private` to localize variables:

```
task_list l;
@make_private {
    id = @thread_id;
}
@start_thread_and_run(n) {
    if (id == 0) {
        for (i = 1; i < n; ++i) {
            @create_task(l) {
                print("Thread %d started\n", id);
            }
        }
        @consume_task_until_empty(l);
        for (i = 1; i < n; ++i) {
            @create_task(l) {
                print("Thread %d finished\n", id);
            }
        }
        @consume_task_until_empty(l);
        @marked_as_finished(l);
    } else {
        @consume_task_until_finished(l);
    }
}
```

Parallelizing a loop with `@get_chunk`:

```
for (i = start; i < end; ++i) {
    foo(i);
}
// Parallel version:
@start_thread_and_run(n) {
    [lstart, lend] = @get_chunk(start, end, step, @thread_count);
    for (i = lstart; i < lend; i += step) {
        foo(i);
    }
}
```

An alternative (less efficient) approach using a shared index:

```
// Possible infinite loop!
global_index = start;
@start_thread_and_run(n) {
    local_index = start;
    while (local_index < end) {
        @one_at_a_time {
            local_index = global_index;
            global_index++;
        }
        if (local_index < end) {
            foo(local_index);
        }
    }
}
```

Note that when a thread updates a memory location, other threads may not observe the update immediately due to caching and compiler optimizations. Explicit synchronization mechanisms (locks, barriers) are required to ensure memory visibility across threads.

```
// Possible infinite loop due to reordering and caching
```

```
global_var = 0;
@start_thread_and_run(n) {
    if (@thread_id == 0) {
        global_var = 1;
    } else {
        while (global_var == 0) {
            // Waiting for global_var to be updated
        }
    }
}
```

The reasons for such behavior are complex, involving both compiler and hardware memory models. The key takeaway is that explicit synchronization is necessary to guarantee that threads see consistent memory updates.

3.2.3 Exercises

Provide a possible text output of the examples in Section 3.2.2.

3.3 Parallel Programming Patterns in Shared Memory

3.3.1 Divide and Conquer

A single thread begins the work and, when the problem size exceeds a threshold, divides it into two subtasks. Those subtasks are executed in parallel by other threads, and once both complete, their results are combined.

```
task_list l;

function divide_and_conquer(data) {
    if (size(data) <= threshold) {
        // Base case: solve directly
        solve_problem(data);
    } else {
        // Recursive case: split data
        @create_task(l)[data1] {
            divide_and_conquer(data1);
        }
        @create_task(l)[data2] {
            divide_and_conquer(data2);
        }
        @consume_task_until_empty(l);  // wait for both subtasks
        combine(data1, data2);         // merge results
    }
}

@start_thread_and_run(n) {
    if (@thread_id == 0) {
        divide_and_conquer(input_data);
        @marked_as_finished(l);
    }
    @consume_task_until_finished(l);
}
```

3.3.2 Split and Merge / Reduction

Each thread processes a disjoint chunk of the input independently. After all threads finish, they synchronize and merge their partial results.

```
@start_thread_and_run(n) {
    [lstart, lend] = @get_chunk(start, end, 1, @thread_count);

    for (i = lstart; i < lend; ++i) {
        local_res = merge(local_res, process_data(i));
    }

    @wait_for_all;    // ensure all threads have finished processing

    @one_at_a_time {
        merge_results(global_res, local_res);  // merge each thread s output into a global ...
     result
    }
}
```

A tree-based (pairwise) reduction halves the number of participants each round:

```
@start_thread_and_run(n) {
    local_val = compute_local_result();
    shared_array[@thread_id] = local_val;
    step = @thread_count / 2;

    while (step > 0) {
        if (@thread_id < step) {
            shared_array[@thread_id] += shared_array[@thread_id + step];
        }
        @wait_for_all;
        step /= 2;
    }
    // Thread 0 holds the total in shared_array[0]
}
```

3.3.3 Producer/Consumer

One thread (the producer) enqueues work items as tasks; all threads - including the producer - then dequeue and process them until none remain.

```
task_list l;

@start_thread_and_run(2) {
    if (@thread_id == 0) {
        // Producer thread
        for (i = 0; i < N; ++i) {
            item = produce_item(i);
            @create_task(l)[item] {
                // Task carries the item for the consumer
                consume_item(item);
            }
        }
        @marked_as_finished(l);
    } else {
        // Consumer thread
        @consume_task_until_finished(l);
    }
}
```

3.3.4 Coloring

When adjacent data elements must not be processed simultaneously (e.g. because they share state), data coloring partitions the work so no two threads touch neighboring elements at once.

```
@start_thread_and_run(n) {
    // Process even-indexed pairs
    [lstart, lend] = @get_chunk(0, N-1, 2, @thread_count);
    for (i = lstart; i < lend; i += 2) {
        foo(tab[i], tab[i+1]);
    }
    @wait_for_all;

    // Process odd-indexed pairs
    [lstart, lend] = @get_chunk(1, N-1, 2, @thread_count);
    for (i = lstart; i < lend; i += 2) {
        foo(tab[i], tab[i+1]);
    }
}
```

3.3.5 Exercises

Provide a parallel algorithm for each of the following sequential versions:

Parallel loop

```
for i = start to end {
    for j = start to end {
        foo(i, j);
    }
}
```

Parallel loop V2

```
for i = start to end {
    for j = start to end {
        tab[j] = foo(i, j);
    }
}
```

Fibonacci

```
// Fibonacci (recursive)
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Find minimum

```
// Find minimum in an array
int array[N];
int min = array[0];
for (i = 1; i < N; ++i) {
    if (array[i] < min) {
        min = array[i];
    }
}
```

GEMM

```
// General matrix-matrix multiplication (GEMM)
for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
        C[i][j] = 0;
        for (k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

get_chunk Implementation

Provide an implementation of the function `get_chunk(start, end, step, split)` that, given the iteration space $\{start, start + step, \ldots, end - 1\}$ and a number of partitions `split`, returns the $(l_{\text{start}}, l_{\text{end}})$ pair for the current thread so that the entire space is divided into `split` contiguous chunks of (nearly) equal size.

3.4 Implementation with OpenMP

The model we defined in the previous section is simplified and omits hardware details. In practice, a very similar tool - OpenMP - is the most widely used library for shared-memory parallelization in the HPC community. OpenMP provides compiler directives, runtime library routines, and environment variables to specify parallel regions in a program. It is designed to be easy to use and can be added to existing code with minimal changes.

Below is the correspondence between our abstract model and OpenMP:

- **#pragma omp parallel num_threads(n) {...}** Spawns n threads (one master + $n - 1$ workers), each executing the enclosed region. Variables declared inside are private by default.
- **#pragma omp critical {...}** Ensures that only one thread at a time executes the enclosed region.
- **#pragma omp ordered** Within an ordered region of a parallel loop, enforces that iterations execute the ordered block in loop-index order.

- **#pragma omp barrier** A synchronization point that blocks each thread until all threads in the parallel region reach it.
- **#pragma omp task firstprivate(...){...}** Creates a new task. Variables listed in `firstprivate` are captured by copy, making them private to the task.
- **#pragma omp taskwait** Blocks the invoking thread until all child tasks it generated have completed.
- **#pragma omp for schedule(dynamic)** Distributes loop iterations among threads dynamically - threads ”grab” new iterations until the loop is finished.
- **#pragma omp for schedule(dynamic,1)** Same as above, but hands out one iteration at a time for finer granularity.
- **#pragma omp taskgroup** Defines a task group; at its end there is an implicit `taskwait`, marking the group as finished.
- **#pragma omp parallel private(...)** Starts a parallel region in which the listed variables are private to each thread.
- **omp_get_thread_num()** Returns the calling thread's ID ($0 \ldots$ `omp_get_num_threads()` -1).
- **omp_get_num_threads()** Returns the total number of threads in the current parallel region.
- **#pragma omp for** Splits the following loop into chunks and executes them in parallel.
- **#pragma omp parallel for** Combines the `parallel` and `for` directives into a single parallel loop.

OpenMP threads are managed by the operating system scheduler, which maps them onto available CPU cores. Creating more threads than physical cores can cause oversubscription and degrade performance due to increased context switching.

OpenMP also supports thread affinity, which can be controlled via environment variables to bind threads to specific cores. This can reduce cache misses and improve data locality.

For portability, OpenMP pragmas are designed to be ignored if the compiler does not support OpenMP, so that the code remains valid even without OpenMP.

You can tune OpenMP at runtime using environment variables rather than changing source code. Common ones include:

- **OMP_NUM_THREADS**: Default number of threads in parallel regions (overridden by the `num_threads` clause).
- **OMP_SCHEDULE**: Default loop scheduling policy (`static`, `dynamic`, `guided`).
- **OMP_PLACES**: Defines thread placement on cores (controls affinity).
- **OMP_PROC_BIND**: Thread-binding policy (`close`, `spread`, `master`).
- **OMP_WAIT_POLICY**: Wait policy for waiting constructs (`passive`, `active`).

```c
int main() {
    int n = 10;
    int a[n];

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        a[i] = i;
    }

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Thread %d started\n", id);

        #pragma omp barrier

        printf("Thread %d finished\n", id);
    }

    return 0;
}
```

Code 3.1: Example of OpenMP code.

3.4.1 Exercises

Implement the examples from the previous section using OpenMP.

3.4.2 Advanced Topics

To optimize an OpenMP (and shared-memory) application, consider the following:

- **Load balancing:** Ensure all threads have approximately the same amount of work.
- **Task granularity:** Choose task sizes large enough to amortize overhead.
- **Critical sections:** Minimize their use to avoid contention.
- **Sequential vs. parallel workload:** Identify code regions that can be parallelized and those that must remain sequential.
- **Deadlocks:** Avoid cyclic dependencies that can cause threads to wait indefinitely.
- **Shared/private variables:** Make sure you understand which variables are shared and which are private to each thread.

Chapter 4

Distributed Memory Parallelization

Overview

This chapter focuses on distributed-memory parallelization, a programming model in which multiple processes communicate and share data over a network. This model is commonly used in high-performance computing (HPC) applications, where several nodes (computers) work together to solve large problems. The same principles also apply on a single machine with multiple CPU cores or on hybrid architectures combining shared and distributed memory.

4.1 Architecture

Computing nodes are connected through a network, and each node has its own local memory. Networks may use Ethernet, InfiniBand, or other high-speed interconnects. Rather than invoking low-level OS functions (such as sockets) directly, we rely on a library that provides a higher-level abstraction. The most widely used library for distributed-memory parallelization is MPI (Message Passing Interface), which offers functions for point-to-point messaging, synchronization, and collective communication.

The nodes are arranged according to a network topology - common examples include star, ring, mesh, and torus - and the MPI implementation is typically optimized for the chosen topology. Most of these optimizations benefit collective operations (such as broadcast or reduction), but for point-to-point communication the developer must carefully choose which processes communicate, especially at large scale.

Distributed memory: A parallel-computing model where each process has its own local memory and communicates with other processes via message passing.

Network topology: The arrangement of nodes and connections in a network. Common topologies include star, ring, mesh, and torus.

Collective communication: Operations involving all processes in a communicator, such as broadcast, scatter, gather, and reduction.

Point-to-point communication: Operations between two processes, such as send and receive.

Network interface: A hardware component that connects a node to the network, handling data transmission and reception.

Network protocol: A set of rules for communication over a network. Common protocols include TCP/IP, UDP, and MPI.

Network bandwidth: The maximum data rate of a network, typically measured in bits per second (bps).

Network latency: The time for a data packet to travel from source to destination, usually measured in milliseconds (ms).

Network congestion: When demand for network resources exceeds capacity, causing delays and packet loss.

Send: An operation that sends data to another process.

Receive: An operation that receives data from another process.

Non-blocking (asynchronous) communication: Operations that allow a process to continue executing while waiting for a message operation to complete, improving performance by overlapping communication and computation.

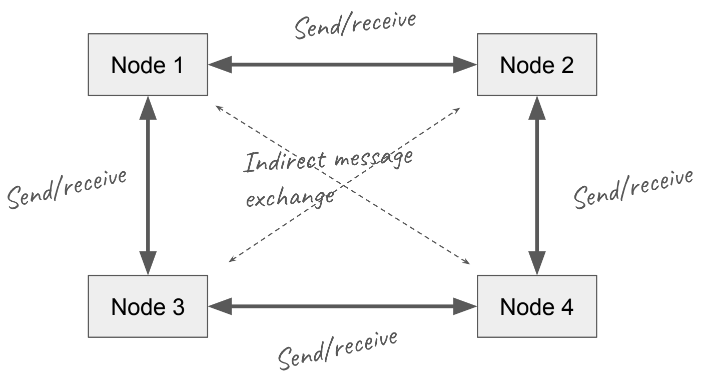


Figure 4.1: Distributed memory model.

4.2 Distributed Parallelization Algorithmic Model

Before using MPI, we introduce a simplified model analogous to our shared-memory model. The key differences are:

- There is *no* shared memory; all variables are private to each process.
- There is *no* implicit synchronization; all communication is via message passing.

In this model, we define the following *blocking* operations:

- **@send(dest, var):** Sends the variable `var` to process `dest`.
- **@receive(src, var):** Receives the variable `var` from process `src`.
- **@broadcast(src, dests, var):** Broadcasts the variable `var` from process `src` to all processes in `dests`.

- **@scatter(src, dests, var):** Scatters elements of the variable `var` from process `src` across the processes in `dests`.
- **@gather(srcs, dest, var):** Gathers the variable `var` from all processes in `srcs` to process `dest`.
- **@reduce(srcs, dest, var, OP):** Applies the reduction operation `OP` to `var` from all processes in `srcs`, delivering the result to process `dest`.
- **@barrier():** Blocks all processes until every process in the communicator reaches this barrier.
- **@proc_id():** Returns the rank (ID) of the calling process.
- **@proc_count():** Returns the total number of processes in the communicator.

Because blocking operations force processes to wait, they can degrade performance when processes stall on communication. To overlap communication and computation, we introduce these *non-blocking* variants, which return a request handle:

- **@async_send(dest, var):** Initiates sending `var` to process `dest` and returns a handle.
- **@async_receive(src, var):** Initiates receiving `var` from process `src` and returns a handle.
- **@async_broadcast(src, dests, var):** Initiates broadcasting `var` from process `src` to `dests` and returns a handle.
- **@async_scatter(src, dests, var):** Initiates scattering `var` from `src` to `dests` and returns a handle.
- **@async_gather(srcs, dest, var):** Initiates gathering `var` from `srcs` to `dest` and returns a handle.
- **@async_reduce(srcs, dest, var, OP):** Initiates reduction `OP` on `var` from `srcs` to `dest` and returns a handle.
- **@async_barrier():** Initiates a barrier and returns a handle.
- **@wait_all(handles):** Blocks until all operations in `handles` complete.
- **@wait_any(handles):** Blocks until any one operation in `handles` completes and returns the corresponding handle.

Note that each non-blocking operation returns a handle; you must ensure not to overwrite or free the associated variables until the operation completes (as indicated by `@wait_all` or `@wait_any`).

4.3 Parallel programming patterns for distributed memory

4.3.1 Hide communication with computation

```
// Overlap communication with computation
handle_send = @async_send(dest, my_data);
handle_recv = @async_receive(src, other_data);
compute();  // Perform computation while waiting
@wait_all([handle_send, handle_recv]);
```

4.3.2 Neighboring communications

```
id   = @proc_id();
np   = @proc_count();
left  = (id - 1 + np) % np;
right = (id + 1)      % np;

// Exchange data with neighbors
handle_left  = @async_send(left,   my_data);
handle_right = @async_receive(right, other_data);
compute();   // Perform computation while waiting
@wait_all([handle_left, handle_right]);
```

4.3.3 Broadcast implementation

```
// Naive broadcast
id      = @proc_id();
handles = [];

if (id == src) {
    // Root sends to all other processes
    for dest in dests {
        h = @async_send(dest, my_data);
        handles.push(h);
    }
} else {
    // Others receive from root
    h = @async_receive(src, my_data);
    handles.push(h);
}

@wait_all(handles);
```

```
// Tree-based broadcast
id      = @proc_id();
np      = @proc_count();
step    = 1;

while (step < np) {
    handles = [];

    if (id % (2 * step) == 0) {
        // Parent sends to its right child
        target = id + step;
        if (target < np) {
            handles.push(@async_send(target, my_data));
        }
    } else if (id % (2 * step) == step) {
        // Child receives from its left parent
        source = id - step;
        handles.push(@async_receive(source, my_data));
    }

    @wait_all(handles);
    step *= 2;
}
```

4.3.4 Synchronization

```
// Naive barrier using two-phase handshake
id      = @proc_id();
np      = @proc_count();
token   = 0;
handles = [];

if (id == 0) {
    // Phase 1: root receives token from all others
    for (i = 1; i < np; i++) {
        handles.push(@async_receive(i, token));
    }
    // Phase 2: root sends release token back
    for (i = 1; i < np; i++) {
        handles.push(@async_send(i, token));
    }
} else {
    // Non-root processes send token to root then wait for release
    handles.push(@async_send(0, token));
    handles.push(@async_receive(0, token));
}

@wait_all(handles);
```

4.3.5 Exercises

Implement the following operations using the distributed-memory model defined above:

Find the minimum

```
int min = array[0];
for (i = 1; i < N; ++i) {
    if (array[i] < min) {
        min = array[i];
    }
}
```

Reduction

```
sum = 0;
for (i = start; i <= end; ++i) {
    sum += array[i];
}
```

Stencil

```
for (i = start; i <= end; ++i) {
    res[i] = foo(array[i-1], array[i], array[i+1]);
}
```

Stencil 2D

```
for (i = start; i <= end; ++i) {
    for (j = start; j <= end; ++j) {
        res[i][j] = foo(array[i-1][j], array[i][j], array[i+1][j],
                        array[i][j-1], array[i][j+1]);
    }
}
```

4.4 Implementation with MPI

In HPC, the MPI library implements distributed-memory parallelization. MPI is a standardized, portable message-passing system that enables processes to communicate in a parallel computing environment. It provides functions for point-to-point communication, collective communication, and synchronization:

- **MPI_Init**: Initializes the MPI environment.
- **MPI_Finalize**: Cleans up the MPI environment.
- **MPI_Comm_rank**: Returns the rank (ID) of the calling process.
- **MPI_Comm_size**: Returns the total number of processes.
- **MPI_Send**: Blocking send.
- **MPI_Recv**: Blocking receive.
- **MPI_Isend**: Non-blocking send.
- **MPI_Irecv**: Non-blocking receive.
- **MPI_Wait**: Blocks until a non-blocking operation completes.
- **MPI_Test**: Checks if a non-blocking operation has completed.
- **MPI_Bcast**: Broadcasts data from one process to all others.
- **MPI_Scatter**: Distributes distinct data from one process to all others.
- **MPI_Gather**: Collects data from all processes to one process.
- **MPI_Reduce**: Combines data from all processes (e.g., sum, min, max).
- **MPI_Barrier**: Synchronizes all processes in a communicator.

- **MPI_Status**: Stores metadata about a received message.
- **MPI_Request**: Handle for non-blocking operations.
- **MPI_Comm_world**: Predefined communicator including all processes.

Writing MPI by hand can be cumbersome, so higher-level libraries or frameworks are often used in real applications.

4.4.1 Exercises

Re-implement the patterns and examples above using MPI calls.

Chapter 5

Optimization on CPU

Overview

In this chapter, we discuss optimization techniques for execution on the CPU, focusing on three areas: vectorization, memory access, and instruction pipelining/scheduling.

5.1 Architecture

CPU architectures are designed to execute instructions in a predetermined order. However, modern CPUs can execute multiple instructions simultaneously via pipelining, out-of-order execution, and the presence of multiple computational units.

Modern CPUs boost performance by exploiting both instruction-level and data-level parallelism through a suite of complementary techniques. Pipelining breaks instruction execution into discrete stages, allowing multiple instructions to overlap in flight, while superscalar designs add parallel execution units to issue several instructions per cycle, harnessing instruction-level parallelism (ILP). Out-of-order execution further maximizes resource utilization by dynamically reordering instructions around data dependencies, and branch prediction combined with speculative execution enables the processor to guess the outcome of conditional jumps, executing along predicted paths and rolling back if the guess is incorrect. To keep data close at hand, a hierarchy of caches (L1, L2, L3) stores memory in units called cache lines - cache hits deliver data almost instantly, whereas misses force slower main-memory fetches, and coherence protocols ensure consistency across cores. Prefetching also helps by loading likely-needed lines in advance. Finally, SIMD vectorization uses wide vector registers - each with a fixed vector length - to perform the same operation on multiple data elements in parallel, further increasing throughput.

Pipelining: A technique used in CPU architecture to improve instruction throughput by overlapping the execution of multiple instructions. The CPU is divided into stages, and each stage processes a different instruction simultaneously.

Out-of-order execution: A technique that allows the CPU to execute instructions in a different order than they appear in the program, based on data dependencies and resource availability. This can improve performance by reducing idle time and increasing instruction-level parallelism.

Instruction-level parallelism (ILP): The ability of a CPU to execute multiple instructions simultaneously by exploiting parallelism inherent in the instruction stream.

Superscalar architecture: A CPU design that can issue multiple instructions per clock cycle by using multiple execution units, allowing the CPU to exploit ILP and improve performance.

Speculative execution: A technique used in modern CPUs to improve performance by executing instructions before it is known whether they will be needed. If the speculation is correct, the results are committed; otherwise, they are discarded.

Branch prediction: A technique used to predict the outcome of conditional branches. The CPU speculatively executes instructions based on the prediction, rolling back if the prediction proves incorrect.

Cache: A small, fast memory located close to the CPU that stores frequently accessed data and instructions to reduce access latency.

Cache hierarchy: The organization of multiple cache levels (L1, L2, L3) in a CPU. Each level has different sizes and speeds, with L1 being the smallest and fastest, and L3 being the largest and slowest.

Cache line: The smallest unit of data that can be transferred between the cache and main memory, typically ranging from 32 to 128 bytes.

Cache hit: Occurs when the data requested by the CPU is found in the cache, yielding low access latency.

Cache miss: Occurs when the data requested by the CPU is not in the cache, forcing a slower access to main memory.

Cache coherence: A mechanism in multiprocessor systems that ensures all caches maintain a consistent view of memory, preventing the use of stale data.

Prefetching: A technique that loads data into the cache before it is requested, reducing cache misses and improving performance.

Vectorization: The process of converting scalar operations into vector operations that can be executed simultaneously on multiple data elements, typically using SIMD instructions.

SIMD (Single Instruction, Multiple Data): A parallel computing model that allows a single instruction to operate on multiple data elements at once, often used in vectorization.

Vector register: A wide register used in SIMD architectures to hold multiple data elements.

Vector length: The number of data elements that a SIMD instruction can process in parallel, determined by the width of the vector registers and the SIMD instruction set.

5.2 Vectorization

Vectorization consists of writing code that can be executed by the vector computation units. To this end, we must ensure that data are loaded into vector registers and that vector instructions are used.

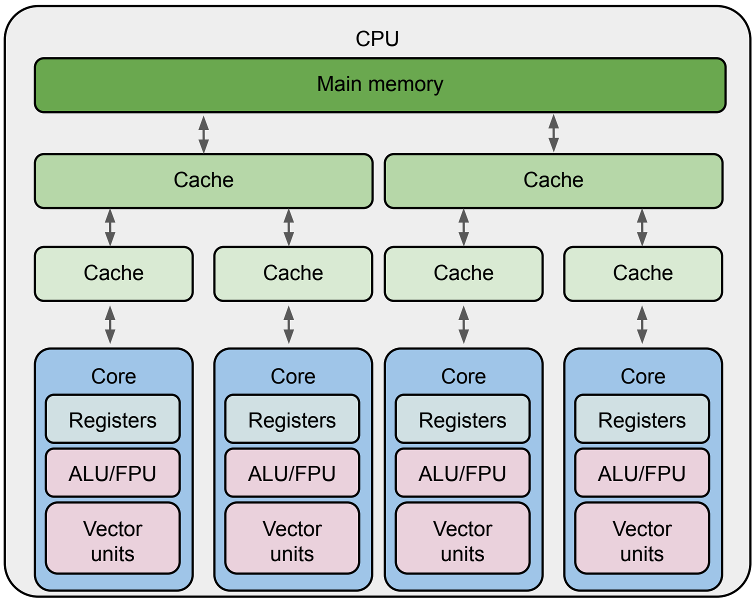


Figure 5.1: CPU architecture overview.

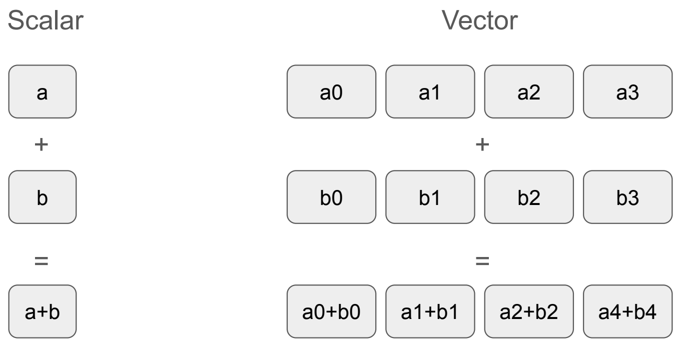


Figure 5.2: Vectorization principle.

5.2.1 Vectorization programming model

We use a simplified model to describe vectorization. The definitions are:

- **vec = @vector_load(ptr):** Load data from the memory location pointed to by `ptr` into the vector register `vec`.
- **@vector_store(vec, ptr):** Store data from the vector register `vec` into the memory location pointed to by `ptr`.
- **vec = @vector_op(vec1, vec2, op):** Apply operation `op` (e.g., addition, multiplication, logical AND/OR) to vector registers `vec1` and `vec2`, storing the result in `vec`.
- **vec = @vector_set(value):** Initialize all elements of the vector register `vec` to `value`.
- **scalar = @vector_reduce(vec, op):** Reduce the elements of vector register `vec` using operation `op` (e.g., sum, min, max, logical AND/OR) and store the result in a scalar register.
- **vec = @vector_gather(ptr, indices):** Load elements from memory at `ptr` indexed by `indices` into the vector register `vec`.
- **@vector_scatter(vec, ptr, indices):** Store elements from vector register `vec` into memory at `ptr` using `indices`.
- **vec = @vector_permute(vec, indices):** Rearrange the elements of vector register `vec` according to `indices`, storing the result in `vec`.

- **vec = @vector_select(vec1, vec2, mask):** Select elements from `vec1` or `vec2` based on the boolean mask `mask`, storing the result in `vec`.
- **mask = @vector_compare(vec1, vec2, op):** Compare elements of `vec1` and `vec2` using comparison `op` (e.g., equal, less-than, greater-than), producing a boolean mask.

5.2.2 Vectorization algorithms

Here are some examples of vectorized algorithms:

Scalar product

```c
// Scalar product (not vectorized)
float scalar_product(float *a, float *b, int n) {
    float result = 0;
    for (int i = 0; i < n; ++i) {
        result += a[i] * b[i];
    }
    return result;
}

// Scalar product (vectorized)
float scalar_product(float *a, float *b, int n) {
    float result = 0;
    int i = 0;
    for (; i + VEC_SIZE <= n; i += VEC_SIZE) {
        vec_t vec_a = @vector_load(&a[i]);
        vec_t vec_b = @vector_load(&b[i]);
        vec_t vec_res = @vector_op(vec_a, vec_b, '*');
        result += @vector_reduce(vec_res, '+');
    }
    for (; i < n; ++i) {
        result += a[i] * b[i];
    }
    return result;
}
```

Code 5.1: Example of vectorized code.

Find minimum

```c
// Find minimum (not vectorized)
float find_min(float *a, int n) {
    float min = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] < min) {
            min = a[i];
        }
    }
    return min;
}

// Find minimum (vectorized)
float find_min(float *a, int n) {
    vec_t vec_min = @vector_set(a[0]);
    int i = 1;
    for (; i + VEC_SIZE <= n; i += VEC_SIZE) {
        vec_t vec_a = @vector_load(&a[i]);
        vec_min = @vector_op(vec_min, vec_a, '<');
    }
    float min = @vector_reduce(vec_min, '<');
    for (; i < n; ++i) {
        if (a[i] < min) {
            min = a[i];
        }
    }
    return min;
}
```

Code 5.2: Example of vectorized code.

Particle interaction

```c
// Particle interaction (not vectorized)
void particle_interaction(float *x, float *y, float *z, float *res, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            float dx = x[i] - x[j];
            float dy = y[i] - y[j];
            float dz = z[i] - z[j];
            float dist = sqrtf(dx*dx + dy*dy + dz*dz);
            res[i] += f(dist);
        }
    }
}

// Particle interaction (vectorized)
void particle_interaction(float *x, float *y, float *z, float *res, int n) {
    for (int i = 0; i < n; i += VEC_SIZE) {
        vec_t vec_x = @vector_load(&x[i]);
        vec_t vec_y = @vector_load(&y[i]);
        vec_t vec_z = @vector_load(&z[i]);
        for (int j = i + 1; j < n; j += VEC_SIZE) {
            vec_t dx = @vector_load(&x[j]) - vec_x;
            vec_t dy = @vector_load(&y[j]) - vec_y;
            vec_t dz = @vector_load(&z[j]) - vec_z;
            vec_t dist = sqrt(vec_dx*vec_dx + vec_dy*vec_dy + vec_dz*vec_dz);
            res[i] += f(dist);
        }
        for (int k = (n/VEC_SIZE)*VEC_SIZE; k < n; ++k) {
            float dx = x[i] - x[k];
            float dy = y[i] - y[k];
            float dz = z[i] - z[k];
            float dist = sqrtf(dx*dx + dy*dy + dz*dz);
            res[i] += f(dist);
        }
    }
}
```

Code 5.3: Example of vectorized code.

Particle interaction with cutoff

```c
// Particle interaction with cutoff (not vectorized)
void particle_interaction(float *x, float *y, float *z, float *res, int n, float cutoff) {
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            float dx = x[i] - x[j];
            float dy = y[i] - y[j];
            float dz = z[i] - z[j];
            float dist = sqrtf(dx*dx + dy*dy + dz*dz);
            if (dist < cutoff) {
                res[i] += f(dist);
            }
        }
    }
}

// Particle interaction with cutoff (vectorized)
void particle_interaction(float *x, float *y, float *z, float *res, int n, float cutoff) {
    for (int i = 0; i < n; i += VEC_SIZE) {
        vec_t vec_x = @vector_load(&x[i]);
        vec_t vec_y = @vector_load(&y[i]);
        vec_t vec_z = @vector_load(&z[i]);
        for (int j = i + 1; j < n; j += VEC_SIZE) {
            vec_t dx = @vector_load(&x[j]) - vec_x;
            vec_t dy = @vector_load(&y[j]) - vec_y;
            vec_t dz = @vector_load(&z[j]) - vec_z;
            vec_t dist = sqrt(dx*dx + dy*dy + dz*dz);
            mask_t m = @vector_compare(dist, cutoff, '<');
            res[i] += f(dist, m);
        }
        for (int k = (n/VEC_SIZE)*VEC_SIZE; k < n; ++k) {
            float dx = x[i] - x[k];
            float dy = y[i] - y[k];
            float dz = z[i] - z[k];
            float dist = sqrtf(dx*dx + dy*dy + dz*dz);
            if (dist < cutoff) {
                res[i] += f(dist);
            }
        }
    }
}
```

Code 5.4: Example of vectorized code.

Exercices

Implement the following algorithms using the vectorization model defined above:

Matrix multiplication Implement a vectorized version of matrix multiplication using the vectorization model. Consider that the second matrix is stored in a transposed form to optimize memory access.

```c
void matrix_multiply(float *a, float *b, float *c, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j){
            c[i*n + j] = 0.0f;
            for (int k = 0; k < n; ++k) {
                c[i*n + j] += a[i*n + k] * b[j*n + k];
            }
        }
    }
}
```

Code 5.5: Matrix multiplication implementation (scalar).

Convolution Implement a vectorized version of a 1D convolution operation using the vectorization model. As a reminder, the convolution operation is defined as:

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(n-m)$$

where f is the input signal and g is the kernel.

```c
void convolution(float *input, float *kernel, float *output, int input_size, int ...
     kernel_size) {
    int half_kernel = kernel_size / 2;
    for (int i = 0; i < input_size; ++i) {
        sum = 0.0f;
        for (int j = -half_kernel; j <= half_kernel; ++j) {
            if (i + j >= 0 && i + j < input_size) {
                sum += input[i + j] * kernel[j + half_kernel];
            }
        }
        output[i] = sum;
    }
}
```

Code 5.6: Convolution implementation (scalar).

Histogram computation Implement a vectorized version of histogram computation for a given array of integers. The histogram should count the occurrences of each integer in a specified range.

```c
void histogram(int *data, int *hist, int n, int range) {
    for (int i = 0; i < range; ++i) {
        hist[i] = 0;
    }
    for (int i = 0; i < n; ++i) {
        hist[data[j]]++;
    }
}
```

Code 5.7: Histogram computation implementation (scalar).

5.2.3 Implementation

When implementing vectorized kernels, one must select the target ISA (Instruction Set Architecture). The most commonly used ISAs for x86 platforms are AVX, AVX2, and AVX-512. They differ primarily in the width of their vector registers, the number of registers, and the available instructions; for example, certain operations exist in AVX-512 but not in AVX2. There are also non-x86 ISAs - such as ARM's NEON, IBM's POWER, and SPARC - which likewise vary in register width, count, and instruction support.

Here are some examples of vectorized algorithms:

Scalar product

```
// Scalar product (not vectorized)
float scalar_product(float *a, float *b, int n) {
    float result = 0.0f;
    for (int i = 0; i < n; ++i) {
        result += a[i] * b[i];
    }
    return result;
}

// Scalar product (vectorized)
#include <immintrin.h>

float scalar_product(float *a, float *b, int n) {
    __m512 vec_sum = _mm512_setzero_ps();
    int i = 0;
    for (; i <= n - 16; i += 16) {
        __m512 vec_a   = _mm512_loadu_ps(a + i);
        __m512 vec_b   = _mm512_loadu_ps(b + i);
        vec_sum        = _mm512_fmadd_ps(vec_a, vec_b, vec_sum);
    }
    float result = _mm512_reduce_add_ps(vec_sum);
    for (; i < n; ++i) {
        result += a[i] * b[i];
    }
    return result;
}

```

Code 5.8: Scalar product implementation using AVX-512 intrinsics.

Find minimum

```
// Find minimum (not vectorized)
float find_min(float *a, int n) {
    float min_val = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] < min_val) {
            min_val = a[i];
        }
    }
    return min_val;
}

// Find minimum (vectorized)
#include <immintrin.h>

float find_min(float *a, int n) {
    __m512 vec_min = _mm512_set1_ps(a[0]);
    int i = 1;
    for (; i <= n - 16; i += 16) {
        __m512 vec_a = _mm512_loadu_ps(a + i);
        vec_min      = _mm512_min_ps(vec_min, vec_a);
    }
    float min_val = _mm512_reduce_min_ps(vec_min);
    for (; i < n; ++i) {
        if (a[i] < min_val) {
            min_val = a[i];
        }
    }
    return min_val;
}

```

Code 5.9: Find minimum implementation using AVX-512 intrinsics.

Particle interaction

```
// Particle interaction (not vectorized)
void particle_interaction(float *x, float *y, float *z, float *res, int n) {
    for (int i = 0; i < n; ++i) {
```

```c
        for (int j = i + 1; j < n; ++j) {
            float dx   = x[i] - x[j];
            float dy   = y[i] - y[j];
            float dz   = z[i] - z[j];
            float dist = sqrtf(dx*dx + dy*dy + dz*dz);
            res[i]    += f(dist);
        }
    }
}

// Particle interaction (vectorized)
#include <immintrin.h>

void particle_interaction(float *x, float *y, float *z, float *res, int n) {
    for (int i = 0; i < n; ++i) {
        __m512 vec_xi = _mm512_set1_ps(x[i]);
        __m512 vec_yi = _mm512_set1_ps(y[i]);
        __m512 vec_zi = _mm512_set1_ps(z[i]);
        for (int j = i + 1; j <= n - 16; j += 16) {
            __m512 vec_xj = _mm512_loadu_ps(x + j);
            __m512 vec_yj = _mm512_loadu_ps(y + j);
            __m512 vec_zj = _mm512_loadu_ps(z + j);

            __m512 dx    = _mm512_sub_ps(vec_xi, vec_xj);
            __m512 dy    = _mm512_sub_ps(vec_yi, vec_yj);
            __m512 dz    = _mm512_sub_ps(vec_zi, vec_zj);
            __m512 dist2 = _mm512_add_ps(
                              _mm512_add_ps(_mm512_mul_ps(dx, dx),
                                            _mm512_mul_ps(dy, dy)),
                              _mm512_mul_ps(dz, dz));
            __m512 dist = _mm512_sqrt_ps(dist2);

            __m512 fval  = f(dist); /* vectorized f */
            __m512 res_i = _mm512_loadu_ps(res + i);
            res_i        = _mm512_add_ps(res_i, fval);
            _mm512_storeu_ps(res + i, res_i);
        }
    }
}

```

Code 5.10: Particle interaction implementation using AVX-512 intrinsics.

Particle interaction with cutoff

```c
// Particle interaction with cutoff (not vectorized)
void particle_interaction(float *x, float *y, float *z, float *res, int n, float cutoff) {
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            float dx   = x[i] - x[j];
            float dy   = y[i] - y[j];
            float dz   = z[i] - z[j];
            float dist = sqrtf(dx*dx + dy*dy + dz*dz);
            if (dist < cutoff) {
                res[i] += f(dist);
            }
        }
    }
}

// Particle interaction with cutoff (vectorized)
#include <immintrin.h>

void particle_interaction(float *x, float *y, float *z, float *res, int n, float cutoff) {
    __m512 vec_cutoff = _mm512_set1_ps(cutoff);
    for (int i = 0; i < n; ++i) {
        __m512 vec_xi = _mm512_set1_ps(x[i]);
        __m512 vec_yi = _mm512_set1_ps(y[i]);
        __m512 vec_zi = _mm512_set1_ps(z[i]);
        for (int j = i + 1; j <= n - 16; j += 16) {
            __m512 vec_xj = _mm512_loadu_ps(x + j);
            __m512 vec_yj = _mm512_loadu_ps(y + j);
            __m512 vec_zj = _mm512_loadu_ps(z + j);

            __m512 dx    = _mm512_sub_ps(vec_xi, vec_xj);
            __m512 dy    = _mm512_sub_ps(vec_yi, vec_yj);
            __m512 dz    = _mm512_sub_ps(vec_zi, vec_zj);
            __m512 dist2 = _mm512_add_ps(
                              _mm512_add_ps(_mm512_mul_ps(dx, dx),
                                            _mm512_mul_ps(dy, dy)),
                              _mm512_mul_ps(dz, dz));
            __m512 dist    = _mm512_sqrt_ps(dist2);
            __mmask16 mask = _mm512_cmplt_ps_mask(dist, vec_cutoff);
            __m512 fval    = f(dist); /* vectorized f */
            fval            = _mm512_maskz_mov_ps(mask, fval);

            __m512 res_i = _mm512_loadu_ps(res + i);
            res_i        = _mm512_add_ps(res_i, fval);
            _mm512_storeu_ps(res + i, res_i);
        }
    }
}

```

Code 5.11: Particle interaction with cutoff using AVX-512 intrinsics.

Exercises

Implement the following algorithms using AVX-512 intrinsics.

5.3 Instruction scheduling

CPU architectures are designed to execute instructions in program order; however, modern processors can execute multiple instructions concurrently via pipelining, superscalar issue, and out-of-order execution. These techniques improve resource utilization and overall throughput.

Many of these capabilities stem from the exponential increase in transistor counts - Moore's Law - which has enabled designers to integrate more execution units, deeper pipelines, larger caches, and more sophisticated scheduling hardware.

Instruction scheduling: Reordering instructions to minimize stalls and maximize utilization of execution units; can be performed statically (at compile time) or dynamically (at runtime).

Out-of-order execution: Executing ready instructions even if earlier instructions are still pending, based on data dependencies and resource availability, to reduce idle cycles.

Instruction-level parallelism (ILP): The potential to execute multiple instructions simultaneously by exploiting parallelism inherent in the instruction stream.

Superscalar architecture: A design that issues multiple instructions per cycle by providing several parallel execution units.

Speculative execution: Executing instructions before it is known whether they will be needed (e.g., after a branch), committing results only if predictions are correct.

Branch prediction: Guessing the outcome of conditional branches to keep the pipeline full, with rollback if the prediction is incorrect.

Computational units: The CPU components (e.g., ALU, FPU, vector units) that perform arithmetic and logical operations.

Data hazards: Situations where an instruction depends on the result of a prior instruction, potentially causing pipeline stalls.

Pipeline stalls: Bubbles in the instruction pipeline caused by unresolved dependencies or resource conflicts, reducing throughput.

Register renaming: Dynamically mapping architectural registers to physical registers to eliminate false (name) dependencies and increase parallelism.

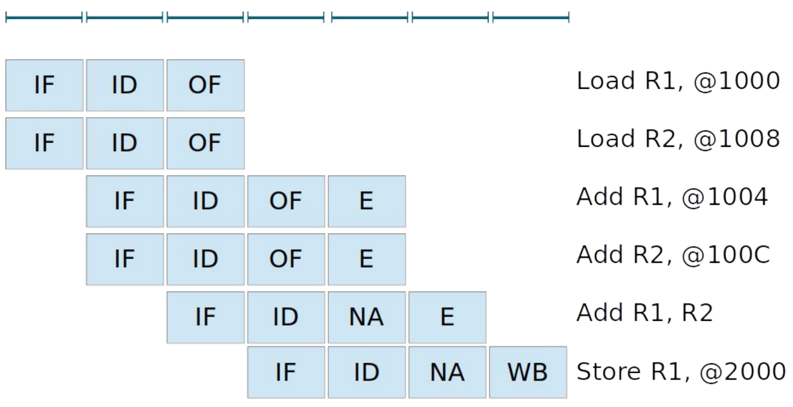


Figure 5.3: Pipelining.

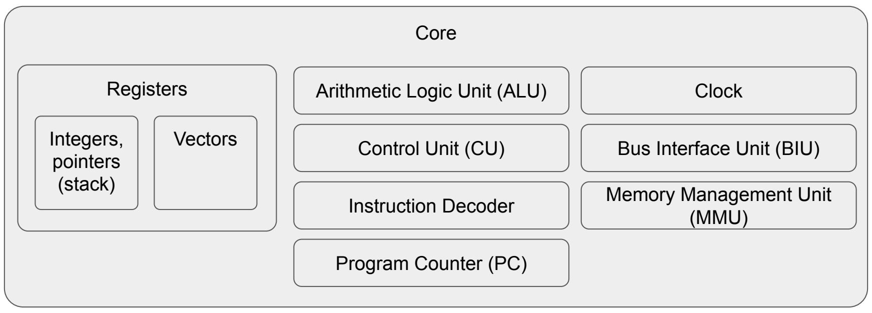


Figure 5.4: CPU architecture.

5.3.1 Programming model

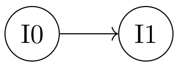
We use a simplified model for instruction scheduling:

- Construct a **dependency graph** in which **nodes** are **instructions** and **edges** denote **data** or **control dependencies**.
- Analyze the graph to determine:
  - the **critical path length** (longest dependent sequence),
  - the total **parallelism** (maximum number of instructions executable simultaneously),
  - and the **register pressure** (number of registers required).
- Treat each control structure (e.g., `if`, `for`, `while`) as delimiting a scheduling **block**, analyzed independently.

Examples

1. Simple Dependent Instructions

```
a = b + c;   // I0
d = a + e;   // I1
```



Execution Timeline:

- Time 0: I0
- Time 1: I1

Duration: 2 cycles
Registers:

- b: R0
- a: R0
- d: R0

2. Independent Instructions

```
a = b + c;   // I0
d = b + e;   // I1
```

Execution Timeline:

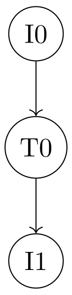
- Time 0: I0, I1 (executed in parallel)

Duration: 1 cycle
Registers:

- b: R0
- a: R1
- d: R0

3. Branching (`if` condition)

```
a = b + c;      // I0
if (a > 0) {    // T0 (test)
    d = b + e; // I1
}
```



Execution Timeline:

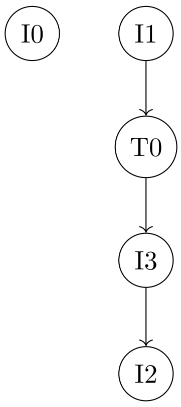
- Time 0: I0
- Time 1: T0
- Time 2: I1

Duration: 3 cycles
Registers:

- b: R0
- a: R1
- d: R1

4. Loop (`for` loop)

```
a = 0;                      // I0
for (i = 0;                  // I1
     i < N;                  // T0 (test)
     i++) {                  // I2
    a += t[i];               // I3
}
```



Execution Timeline:

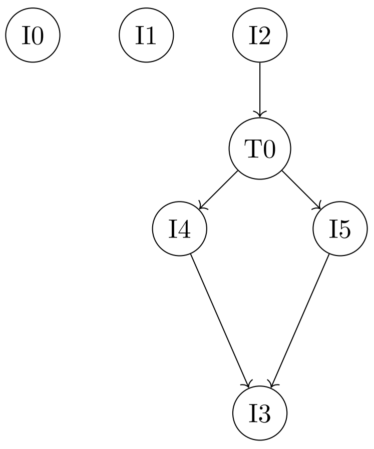
- Time 0: I0, I1
- For each iteration x from 0 to $N-1$:
  - Time $x+1$: T0
  - Time $x+2$: I3
  - Time $x+3$: I2

Duration: $1 + N \times 3$ cycles
Registers:

- a: R0
- i: R1
- N: R2 (optional)

5. Parallel Accumulation in a Loop

```
a0 = 0;                     // I0
a1 = 0;                     // I1
for (i = 0;                  // I2
     i < N/2;                // T0 (test)
     i++) {                  // I3
    a0 += t[i];              // I4
    a1 += t[i+1];            // I5
}
```



Execution Timeline:

- Time 0: I0, I1, I2
- For each iteration x from 0 to $(N/2)-1$:
  - Time $x+1$: T0
  - Time $x+2$: I4, I5 (executed in parallel)
  - Time $x+3$: I3

Duration: $1+\frac{N}{2}\times 3$ cycles
Registers:

- a0: R0
- a1: R1
- i: R2
- N: R3 (optional)

Exercices

Estimate the duration of the following code snippets, assuming that each instruction takes one cycle to execute and that there are no stalls or hazards. Also, indicate the registers used for each variable.

1.

```
a = b + c;   // I0
d = a + e;   // I1
a = b + c;   // I2
f = d + g;   // I3
```

2.

```
a = b + c;   // I0
d = a + e;   // I1
f = b + a;   // I2
g = d + f;   // I3
```

3.

```
a0 = b + c;   // I0
a1 = b + d;   // I1
a0 += e;   // I2
a1 += f;   // I3
a2 = a0 + a1;   // I4
```

4.

```
a = b + d;                  // I0
for (i = 0;              // I1
     i < a;              // T0 (test)
     i++) {              // I2
    t[i] += i;           // I3
}
```

5.3.2 Implementation

Improving pipelining and units usage

```
// Original code
float sum(float *b, float *c, int n) {
    float sum = 0;
    for (int i = 0; i < n; i++) {
        sum += b[i] * c[i];
    }
    return sum;
}
// Improved code
float sum(float *b, float *c, int n) {
    float sum1 = 0;
    float sum2 = 0;
    for (int i = 0; i < n-n%1; i += 2) {
        sum1 += b[i] * c[i];
        sum2 += b[i+1] * c[i+1];
    }
    if(n%1){
        sum2 += b[n-1] * c[n-1];
    }
    return sum1 + sum2;
}
```

Code 5.12: Improving pipeling.

Improving pipelining

```
// Original Code
float sum(float *b, float *c, int n) {
    float sum = 0;
    for (int i = 0; i < n; i++) {
        if(i%2 == 0) {
            sum += b[i] * c[i];
        } else {
            sum -= b[i] * c[i];
        }
    }
    return sum;
}
// Improved Code
float sum(float *b, float *c, int n) {
    float sum1 = 0;
    float sum2 = 0;
    for (int i = 0; i < n-n%2; i += 2) {
        sum1 += b[i] * c[i];
        sum2 -= b[i+1] * c[i+1];
    }
    if(n%2){
        sum2 -= b[n-1] * c[n-1];
    }
    return sum1 + sum2;
}
// Or
float sum(float *b, float *c, int n) {
    float sum = 0;
    for (int i = 0; i < n; i += 2) {
        sum +=  b[i] * c[i];
    }
    for (int i = 1; i < n; i += 2) {
        sum -= b[i] * c[i];
    }
    return sum;
}
```

Code 5.13: Improving branch.

Exercices

Improving the execution of the following code snippets by reordering instructions, reducing dependencies, and minimizing stalls.

1.

```
    for (int i = 0; i < n; i++) {
        sum += b[i] * c[i];
    }
```

Code 5.14: Improving branch.

2.

```
    for (int i = 0; i < n; i++) {
        sum += b[i] * c[i];
    }
    for (int i = 0; i < n; i++) {
        b[i] += 1;
    }
    for (int i = 0; i < n; i++) {
        c[i] += 1;
    }
```

Code 5.15: Improving memory access.

5.4 Memory accesses on CPU

The CPU architecture includes several levels of memory - registers, caches (L1, L2, L3), and main memory (RAM) - each with distinct speed, size, and access characteristics. A cache hierarchy exploits spatial and temporal locality to keep frequently used data and instructions close to the CPU, reducing latency to main memory.

Cache: A small, fast memory close to the CPU that stores frequently accessed data and instructions to reduce main memory latency.

Cache hierarchy: Multiple cache levels (L1, L2, L3) organized by increasing size and latency; L1 is the smallest and fastest, L3 is the largest and slowest.

Cache line: The smallest block of data transferred between cache and memory, typically 32-128 bytes.

Cache hit: When the requested data is found in the cache, yielding low-latency access.

Cache miss: When the requested data is not in the cache, forcing a slower fetch from a lower-level cache or main memory.

Cache coherence: A protocol ensuring consistency of shared data across multiple caches in a multiprocessor system.

Prefetching: Proactively loading data into the cache before it is requested, to reduce miss penalties.

Spatial locality: The tendency to access data addresses close to those accessed previously, improving cache line utilization.

Temporal locality: The tendency to reuse recently accessed data, increasing the likelihood of cache hits.

False sharing: When threads access different data in the same cache line, causing unnecessary coherence traffic and performance loss.

Cache associativity: The mapping of cache lines to sets; higher associativity reduces conflict misses at the cost of increased complexity.

Cache replacement policy: The strategy for evicting lines when the cache is full (e.g., LRU, FIFO, random).

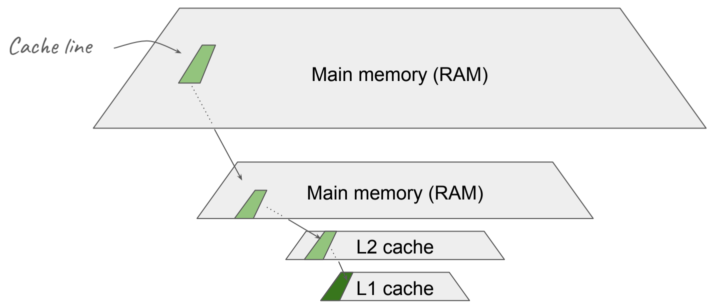


Figure 5.5: Memory organization.

5.4.1 Algorithms

Optimizing memory accesses is crucial for performance. The primary objective is to maximize data locality: spatial locality by accessing adjacent addresses consecutively, and temporal locality by reusing data soon after its first access. Algorithms should be structured to reuse data promptly and to group accesses to contiguous memory regions.

Registers

The CPU provides a limited set of registers, the fastest storage available, to hold intermediate values and frequently used variables. Register operations complete in a single cycle, but operands must reside in registers for arithmetic or logical operations, requiring explicit loads (e.g., `mov`). Excessive register usage leads to spilling to the stack or heap, incurring additional load/store overhead.

Cache

Modern CPUs feature a multi-level cache (L1, L2, L3) to buffer data and instructions. On an access, the CPU probes L1, then L2, then L3, before accessing main memory on a miss. Cache misses can be reduced by using contiguous data layouts, traversing arrays in cache-friendly order (e.g., row-major for matrices), and keeping working sets within cache capacity.

Cache line

A cache line is the smallest unit transferred between cache and memory, typically 32-128 bytes. Fetching an entire line on a miss leverages spatial locality, so accessing elements within the same line reduces misses. Misaligned or strided accesses across lines increase miss rates. Aligning data structures to cache-line boundaries and accessing data sequentially minimizes misses and bandwidth.

5.4.2 Examples

Here are some examples of memory-access patterns and their impact on performance:

Contiguous memory access

```c
for (int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}
```

This pattern accesses `a`, `b`, and `c` contiguously, exploiting spatial locality. The CPU loads full cache lines sequentially, reducing misses and improving throughput.

Strided memory access

```c
for (int i = 0; i < N; i++) {
    a[i] = b[2*i] + c[2*i];
}
```

Here, `b` and `c` are accessed with a stride of 2. This breaks contiguity, causing more cache lines to be fetched and increasing miss rates, which degrades performance.

Random memory access

```c
for (int i = 0; i < N; i++) {
    a[i] = b[rand() % N] + c[rand() % N];
}
```

Random indexing prevents any useful locality. Each access may fetch a different cache line, leading to frequent misses and very poor performance.

Loop unrolling

```c
for (int i = 0; i < N; i += 4) {
    a[i]   = b[i]   + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
}
```

Unrolling reduces loop-control overhead and can increase instruction-level parallelism. It also ensures that each iteration touches multiple contiguous elements, improving cache-line utilization.

Blocking (tiling)

```c
for (int i = 0; i < N; i += B) {
    for (int j = 0; j < N; j += B) {
        for (int k = 0; k < B; k++) {
            a[i+k][j] = b[i+k][j] + c[i+k][j];
        }
    }
}
```

Blocking divides the iteration space into tiles of size B. Each tile fits in cache, so inner loops reuse data in fast memory and reduce capacity misses.

Software prefetching

```c
for (int i = 0; i < N; i++) {
    __builtin_prefetch(&b[i+1], 0, 1);
    a[i] = b[i] + c[i];
}
```

Explicit prefetch hints the hardware to load `b[i+1]` into cache early, hiding memory latency and reducing miss penalties.

Data reuse (two operations)

```
for (int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i] + e[i];
}
```

Here, `a[i]` is produced and immediately consumed in the same loop, improving temporal locality and reducing reloads.

Data reuse (fusion vs. two passes)

```
// Two separate loops:
for (int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}
for (int i = 0; i < N; i++) {
    a[i] *= v;
}

// Fused loop (better if data do not persist in cache between passes):
for (int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
    a[i] *= v;
}
```

If N is large and $\{a, b, c\}$ do not all fit in cache simultaneously, fusing the two operations in one loop preserves `a[i]` in cache and avoids a second pass through memory.

Exercices

1.

```
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
            sum += tab[i][j];
        }
    }
```

Code 5.16: Improving memory access 2.

2.

```
    list lst = foo();
    iterator it = lst.begin();
    while (it != lst.end()) {
        sum += *it;
        it++;
    }
```

Code 5.17: Bad data structure.

Chapter 6

Optimization on GPU

Overview

Graphics Processing Units (GPUs) are specialized processors engineered for massive parallelism and high-throughput computation. Unlike CPUs, which optimize for low-latency, serial execution, GPUs contain thousands of lightweight cores that execute many threads concurrently. To harness this parallelism effectively, one must understand both the hardware architecture and the programming model.

At a high level, GPU resources are organized into a hierarchy:

- **Compute units (SMs/CUs):** Groups of cores that execute threads in lock-step (warps/wavefronts).
- **Thread hierarchy:** Threads are grouped into teams (blocks/work-groups) and higher-level grids.
- **Memory hierarchy:** Registers (per-thread), shared/local memory (per-team), and global/device memory, each with different capacity, latency, and bandwidth characteristics.

To program GPUs, one uses frameworks such as NVIDIA's CUDA or the vendor-agnostic OpenCL. These expose abstractions for:

- *Kernels*: Functions compiled for and launched on the GPU.
- *Memory management*: Explicit transfers or unified address spaces.
- *Synchronization*: Barriers at various levels and atomic operations.

Achieving peak performance requires careful management of data placement, thread synchronization, and workload distribution. The following definitions clarify key terms used throughout this chapter.

GPU: A Graphics Processing Unit, a massively parallel processor originally designed for rendering but now widely used for general-purpose computing.

CUDA: Compute Unified Device Architecture, NVIDIA's proprietary programming model and API for writing GPU code in C/C++ (and other languages).

OpenCL: Open Computing Language, an open-standard API for heterogeneous parallel computing across CPUs, GPUs, and other processors.

Kernel: A function annotated to run on the GPU, typically executed by many threads in parallel.

Device memory: The physical DRAM on the GPU, accessible by all threads but with high latency.

Global memory: Synonymous with device memory; large capacity, high-latency.

Shared (local) memory: Low-latency memory shared by all threads within a team (CUDA block or OpenCL work-group).

Registers: Fastest, private storage per thread.

Unified memory: CUDA feature allowing CPU and GPU to share a single virtual address space.

Atomic operations: Hardware-supported read-modify-write operations (e.g., `atomic_add`), used to synchronize updates without race conditions.

Occupancy: The ratio of active warps (or wavefronts) per compute unit to the maximum possible; determines how well the GPU's execution units are utilized.

6.1 GPU Architecture

6.1.1 Compute Units and Threading

A modern GPU comprises several streaming multiprocessors (SMs) or compute units (CUs). Each SM schedules and executes threads in groups called *warps* (CUDA) or *wavefronts* (OpenCL), typically 32 or 64 threads that run the same instruction in SIMD fashion. To hide memory latency, each SM maintains multiple active warps - when one warp stalls on a memory access, another can execute.

6.1.2 Memory Hierarchy

- **Registers:** Fastest, per-thread, limited number.
- **Shared (local) memory:** On-chip SRAM shared within a team, very low latency if bank conflicts are avoided.
- **Global (device) memory:** Off-chip DRAM, large but high latency. Coalesced accesses (adjacent threads accessing adjacent addresses) maximize bandwidth.
- **Constant and texture memory:** Read-only caches optimized for broadcast and 2D spatial locality.

6.2 GPU Programming Model

In both CUDA and OpenCL, the basic execution paradigm is:

1. Allocate and initialize host (CPU) and device (GPU) memory.
2. Copy inputs to device memory (unless using unified memory).
3. Launch one or more *kernels* with a specified grid/block (team/thread) configuration.
4. Within each kernel:
    - Compute thread- and team-local identifiers.

  - Perform the desired parallel computation.
  - Synchronize at team or grid level as needed.

5. Copy results back to host memory.

6.2.1 Thread Hierarchy and Built-ins

We adopt a simplified, multi-level model:

- `thread_id(level, dim)`: thread's index within its team at `level`, in dimension `dim`.
- `team_id(level, dim)`: team's index at `level`, in dimension `dim`.
- `number_of_teams(level, dim)`: total teams at `level`, in `dim`.
- `number_of_levels()`: total hierarchy depth.
- `team_barrier(level)`: barrier synchronization among threads in the same team at `level`.
- `atomic_add(var, val)`, `atomic_mul(var, val)`: atomic operations on shared or global variables.
- `__gpu_code__`: a qualifier indicating that the function is a GPU kernel.

6.2.2 Kernel Launch Configuration

- `config`: a list $\big[(T_0, N_0), (T_1, N_1), \dots\big]$, where at level i there are T_i teams each with N_i threads (in 1D; extendable to multi-D).
- `execute_kernel<config>(kernel, args)`: compiles and launches `kernel` with `config`.
- Qualifiers: `__device__` marks GPU-side functions; `__local(level)__` designates storage in level-specific shared memory.

6.2.3 Index computing

To compute the global thread index, we use:

```
int tid = thread_id(0,0) + thread_id(1,0) * number_of_teams(0,0);
```

This computes a unique identifier for each thread based on its position in the team hierarchy.

6.2.4 Performance Optimizations

- **Memory coalescing:** Align and group global memory accesses so threads in a warp access contiguous addresses.
- **Use of shared memory:** Stage frequently reused data in on-chip shared memory to reduce global loads.
- **Occupancy tuning:** Balance register and shared memory usage to maximize active warps per SM.
- **Minimize divergence:** Ensure threads within a warp follow the same execution path to avoid serialization.
- **Asynchronous operations:** Overlap data transfers and kernel execution using streams/queues.

6.2.5 Example

Hierarchical Reduction

```
// Configuration: level 0 -> 4 x 8 teams
//                level 1 -> 8 x 16 threads
// Total threads = 4x8 x 8x16 = 8192
config = [(4,8), (8,16)];

__gpu_code__ float axpy_reduction(const float* x,
                                const float* y,
                                int n)
{
  int L0 = number_of_teams(0,0),  L1 = number_of_teams(1,0);
  int t0 = thread_id(0,0),        t1 = thread_id(1,0);
  int total_threads = L0 * L1;
  int tid = t0 + t1 * L0;

  // Each thread computes a partial dot-product
  float partial = 0.0f;
  for (int i = tid; i < n; i += total_threads)
      partial += x[i] * y[i];

  // Level-1 (innermost) reduction in shared memory
  __local(1)__ float shared_sum;
  if (t0 == 0) shared_sum = 0.0f;
  team_barrier(1);
  atomic_add(shared_sum, partial);
  team_barrier(1);

  // Level-0 (global) reduction across teams
  __local(0)__ float global_sum;
  if (t1 == 0 && t0 == 0) global_sum = 0.0f;
  team_barrier(0);
  if (t1 == 0) atomic_add(global_sum, shared_sum);
  team_barrier(0);

  return global_sum;
}
```

Code 6.1: Hierarchical Reduction.

AXPY

```
// Configuration: level 0 -> 4 x 8 teams
//                level 1 -> 8 x 16 threads
// Total threads = 4x8 x 8x16 = 8192
config = [(4,8), (8,16)];
__gpu_code__ void axpy(const float* x,
                      const float* y,
                      float* z,
                      int n)
{
  int L0 = number_of_teams(0,0),  L1 = number_of_teams(1,0);
  int t0 = thread_id(0,0),        t1 = thread_id(1,0);
  int total_threads = L0 * L1;
  int tid = t0 + t1 * L0;

  // Each thread computes a partial axpy
  for (int i = tid; i < n; i += total_threads)
      z[i] = x[i] + y[i];
}
```

Code 6.2: axpy.

GEMM

```
// Configuration: level 0 -> 4 x 8 teams
//                level 1 -> 8 x 16 threads
// Total threads = 4x8 x 8x16 = 8192
config = [(4,8), (8,16)];
__gpu_code__ void gemm(const float* A,
                      const float* B,
                      float* C,
                      int M, int N, int K)
{
  int L0 = number_of_teams(0,0),  L1 = number_of_teams(1,0);
  int t0 = thread_id(0,0),        t1 = thread_id(1,0);
```

```
    int total_threads = L0 * L1;
    int tid = t0 + t1 * L0;

    // Each thread computes a partial C element
    for (int i = tid; i < M * N; i += total_threads) {
        int row = i / N;
        int col = i % N;
        float sum = 0.0f;
        for (int k = 0; k < K; ++k)
            sum += A[row * K + k] * B[k * N + col];
        C[i] = sum;
    }
}
```

Code 6.3: GEMM.

6.2.6 Exercices

Find a minimum value

```
float find_min(const float* data, int n) {
    float min_val = data[0];
    for (int i = 1; i < n; ++i) {
        if (data[i] < min_val) {
            min_val = data[i];
        }
    }
    return min_val;
}
```

Code 6.4: GPU find a minimum (sequential code).

Increase all values

```
void increase_values(float* data, int n, float increment) {
    for (int i = 0; i < n; ++i) {
        data[i] += increment;
    }
}
```

Code 6.5: GPU increase all values (sequential code).

6.3 Implementation

CUDA can be viewed as a concrete instance of our previously defined programming model, organized into three hierarchical levels:

- **Grid:** A 1D, 2D, or 3D array of *blocks*.
- **Block:** A 1D, 2D, or 3D array of *threads*.
- **Warp (hidden):** A group of 32 threads that execute the same instruction in lockstep. Although not directly exposed to the programmer, warp-level execution (including scheduling and divergence behavior) has a critical impact on performance.

Several intrinsics and synchronization primitives (e.g. `__syncwarp()`, warp-level shuffle functions, and block barriers) operate only within a warp or within a block.

CUDA kernel: A function compiled for and executed on the GPU by many threads in parallel.

CUDA grid: The top-level execution domain in CUDA, defined as a 1D, 2D, or 3D array of blocks.

CUDA block: A group of threads that execute the same kernel and can synchronize via shared memory.

CUDA thread: The smallest unit of execution in CUDA, identified by `threadIdx`.

CUDA warp: A set of 32 threads within a block that execute instructions in lockstep.

CUDA shared memory: Low-latency, on-chip memory shared by all threads in a block.

CUDA global memory: The GPU's main DRAM, large in capacity but with higher latency.

CUDA constant memory: A read-only cached memory space optimized for uniform broadcasts.

CUDA texture memory: A read-only memory with specialized caching for 2D spatial locality.

CUDA atomic operations: Hardware-supported operations (e.g. `atomicAdd()`) that prevent race conditions.

CUDA synchronization: Primitives such as `__syncthreads()` or `__syncwarp()` to coordinate thread execution.

CUDA memory coalescing: Aligning global memory accesses by consecutive threads to maximize bandwidth.

CUDA occupancy: The ratio of active warps per Streaming Multiprocessor (SM) to the maximum supported.

CUDA stream: An ordered sequence of operations (kernels, memory copies) that can overlap with other streams.

CUDA event: A marker for synchronizing and timing operations within or across streams.

CUDA profiler: Tools (e.g. `nvprof`, Nsight) for measuring kernel execution time, memory usage, and other metrics.

CUDA error handling: Checking and reporting runtime errors via functions like `cudaGetLastError()`.

CUDA unified memory: A feature providing a single virtual address space for CPU and GPU.

CUDA device query: APIs such as `cudaGetDeviceProperties()` to inspect GPU capabilities.

CUDA kernel launch: Syntax `kernel<<<gridDim,blockDim>>>(...)` to dispatch a kernel.

CUDA execution configuration: The `gridDim` and `blockDim` parameters specifying block and grid sizes.

blockIdx: A built-in 3D index identifying a block's position in the grid.

blockDim: A built-in 3D vector specifying the number of threads per block.

threadIdx: A built-in 3D index specifying a thread's position within its block.

gridDim: A built-in 3D vector specifying the number of blocks in the grid.

cudaMemcpy: A function to transfer data between host (CPU) and device (GPU) memory.

cudaMalloc: A function to allocate memory on the GPU device.

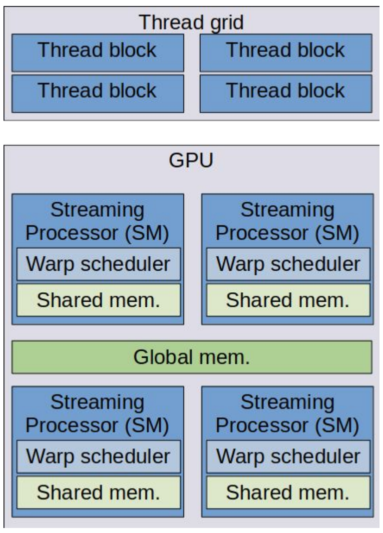


Figure 6.1: CUDA/GPU.

6.3.1 Principles

Effective CUDA implementations should:

- *Expose massive parallelism:* Decompose the problem into many independent tasks.
- *Maximize occupancy:* Launch enough warps per SM to hide memory and execution latencies.
- *Ensure memory coalescing:* Align global memory accesses across threads in a warp.
- *Leverage shared memory:* Stage frequently used data on-chip to reduce global memory loads.
- *Minimize divergence:* Keep threads in a warp following the same execution path.
- *Overlap compute and transfer:* Use streams and events to perform data transfers concurrently with kernel execution.

A typical GPU development workflow involves:

1. Decomposing the algorithm into fine-grained parallel tasks.
2. Selecting grid and block dimensions based on problem size and hardware constraints.
3. Optimizing memory access patterns and resource utilization to maximize throughput.

6.3.2 Examples

Here are some examples of GPU programming using CUDA:

Vector Addition

```
__global__ void add(const int* a,
                    const int* b,
                    int*       c,
                    int        n)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
```

Code 6.6: Simple element-wise vector addition.

Grid-stride Loop

```
__global__ void add_strided(const int* a,
                            const int* b,
                            int*       c,
                            int        n)
{
    int idx    = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = idx; i < n; i += stride) {
        c[i] = a[i] + b[i];
    }
}
```

Code 6.7: Grid-stride loop version of vector addition.

Reduction

```
__global__ void reduce(const int* input,
                       int*       output,
                       int        n)
{
    __shared__ int sdata[256];
    int tid = threadIdx.x;
    int idx = blockIdx.x * blockDim.x + tid;

    // Load input or zero
    sdata[tid] = (idx < n) ? input[idx] : 0;
    __syncthreads();

    // In-place tree-based reduction
    for (int stride = blockDim.x/2; stride > 0; stride >>= 1) {
        if (tid < stride) {
            sdata[tid] += sdata[tid + stride];
        }
        __syncthreads();
    }

    // Write per-block result
    if (tid == 0) {
        output[blockIdx.x] = sdata[0];
    }
}
```

Code 6.8: Block-wise reduction of an integer array into partial sums.

Matrix Multiplication (GEMM)

```
__global__ void gemm(const float* A,
                     const float* B,
                     float*       C,
                     int          N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; ++k) {
            sum += A[row*N + k] * B[k*N + col];
        }
        C[row*N + col] = sum;
    }
}
```

Code 6.9: Simple matrix-matrix multiplication (GEMM), one thread per output element.

Particle Interaction in 2D

```
__global__ void particle_interaction(const float2* pos,
                                     float2*       vel,
                                     int           N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        float2 pi = pos[idx];
        float2 vi = vel[idx];
        for (int j = 0; j < N; ++j) {
            if (j == idx) continue;
            float2 pj = pos[j];
            float  dx = pj.x - pi.x;
            float  dy = pj.y - pi.y;
            float  dist = sqrtf(dx*dx + dy*dy);
            vi.x += dx / dist;
            vi.y += dy / dist;
        }
        vel[idx] = vi;
    }
}
```

Code 6.10: Pairwise particle interaction in 2D using `float2`.

6.3.3 Exercices

Hierarchical Reduction

```
float hierarchical_reduction(const float* data, int n) {
    float sum = 0.0f;
    for (int i = 0; i < n; ++i) {
        sum += data[i];
    }
    return sum;
}
```

Code 6.11: Hierarchical Reduction sequential code.

AXPY

```
void axpy(const float* x,
                     const float* y,
                     float* z,
                     int n)
{
    for (int i = 0; i < n; ++i) {
        z[i] = x[i] + y[i];
    }
}
```

Code 6.12: axpy sequential code.

Find a minimum value

```
float find_min(const float* data, int n) {
    float min_val = data[0];
    for (int i = 1; i < n; ++i) {
        if (data[i] < min_val) {
            min_val = data[i];
        }
    }
    return min_val;
}
```

Code 6.13: GPU find a minimum (sequential code).

Increase all values

```
void increase_values(float* data, int n, float increment) {
    for (int i = 0; i < n; ++i) {
        data[i] += increment;
    }
}
```

Code 6.14: GPU increase all values (sequential code).

Chapter 7

Heterogeneous computing

Overview

Heterogeneous computing refers to systems that combine different types of processors or cores, classically CPUs and GPUs, to leverage their respective strengths. This approach allows for more efficient execution of diverse workloads by assigning tasks to the most suitable processor type.

The challenges of heterogeneous computing include:

- **Data transfer:** Efficiently moving data between different memory spaces (e.g., CPU RAM and GPU VRAM).
- **Task scheduling:** Deciding which tasks run on which processor, balancing load and minimizing latency.
- **Synchronization:** Coordinating execution across processors, especially when tasks depend on shared data.
- **Programming efficient kernels:** Writing code that effectively utilizes multiple processor types, often requiring different programming models (e.g., CUDA for GPUs, OpenMP for CPUs).

Using a task-based runtime system can help manage these challenges by abstracting the details of task scheduling, data transfer, and synchronization. This allows developers to focus on defining tasks and their dependencies rather than low-level implementation details.

The developer's role is then reduced to writing efficient computational tasks (what the tasks actually do) and expressing the tasks/dependencies (taking care of the degree of parallelism and granularity).

Note that this approach is well suited when the task graph does not change during the execution of the program, i.e., when the tasks and their dependencies does not depend on the results of previous tasks.

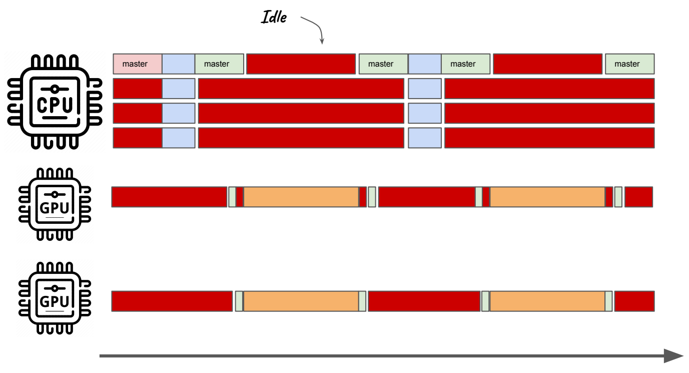


Figure 7.1: Classica GPU usage.

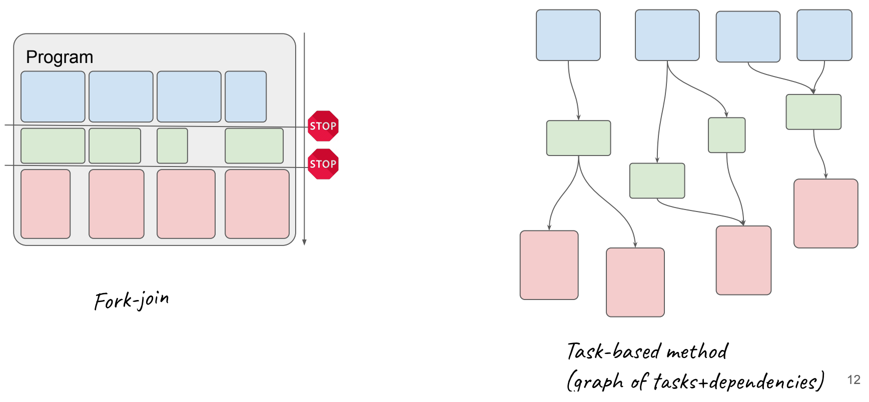


Figure 7.2: Fork-join vs task-based.

7.1 A simplified task-based programming model

We propose a model that abstracts the heterogeneous computing environment into a set of tasks, each with its own data dependencies. The model consists of:

- **task_descr = build_task_descr([arch, func]...)**: Creates a task description for a specific set of architectures and a function to execute. A description can include multiple architectures (e.g., CPU, GPU) and functions (e.g., different implementations for different hardware).
- **data_descr = build_data_descr([ptr, size])**: Defines a data description with a pointer to the data and its size. This can be used to specify how the data is accessed by tasks.
- **insert_task(task_descr, [acces, data_descr] ...)**: Defines a task with a description and its arguments, specifying how the data is accessed (read/write).
- **wait_all_tasks()**: Blocks until all previously defined tasks are completed.

7.1.1 Examples

Single Matrix multiplication

```
task_descr matmul_descr = build_task_descr(
    {CPU, matmul_cpu},
    {GPU, matmul_gpu}
);
function matmul(A, B, C, N) {
    data_descr A_descr = build_data_descr(A, N * N);
    data_descr B_descr = build_data_descr(B, N * N);
    data_descr C_descr = build_data_descr(C, N * N);

    // Insert CPU or GPU task
    insert_task(matmul_descr, {READ, A_descr}, {READ, B_descr}, {WRITE, C_descr});

    // Wait for all tasks to complete
    wait_all_tasks();
}
```

Panel Matrix multiplication

```
task_descr matmul_descr = build_task_descr(
    {CPU, matmul_cpu},
    {GPU, matmul_gpu}
);
function matmul(A, B, C, N, P) {
    vector<data_descr> A_descr;
    vector<data_descr> B_descr;
    vector<data_descr> C_descr;

    // Loop over panels
    for (int i = 0; i < N; i += P) {
        for (int j = 0; j < N; j += P) {
            // Create data descriptors for the current panel
            A_descr.push_back(build_data_descr(A + i * N + j, P * P));
            B_descr.push_back(build_data_descr(B + i * N + j, P * P));
            C_descr.push_back(build_data_descr(C + i * N + j, P * P));
        }
    }

    // Insert tasks for each panel multiplication
    for (int i = 0; i < A_descr.size(); ++i) {
        insert_task(matmul_descr, {READ, A_descr[i]}, {READ, B_descr[i]}, {WRITE, C_descr[i]});
    }

    // Wait for all tasks to complete
    wait_all_tasks();
}
```

Panel AXPY

```
task_descr axpy_descr = build_task_descr(
    {CPU, axpy_cpu},
    {GPU, axpy_gpu}
);
function axpy(a, b, c, N, P) {
    vector<data_descr> A_descr;
    vector<data_descr> B_descr;
    vector<data_descr> C_descr;

    // Loop over panels
    for (int i = 0; i < N; i += P) {
        A_descr.push_back(build_data_descr(a + i, P));
        B_descr.push_back(build_data_descr(b + i, P));
        C_descr.push_back(build_data_descr(c + i, P));
    }

    // Insert tasks for each panel multiplication
    for (int i = 0; i < A_descr.size(); ++i) {
        insert_task(axpy_descr, {READ, A_descr[i]}, {READ, B_descr[i]}, {WRITE, C_descr[i]});
    }

    // Wait for all tasks to complete
    wait_all_tasks();
}
```

7.1.2 Exercice

Implement a panel-based particle interaction algorithm using the task-based programming model described above. The algorithm should include two loops to iterate over the panels of the particles, and each panel interaction should be defined as a separate task.

7.2 Implementation with StarPU

StarPU is a runtime system that provides a task-based programming model for heterogeneous computing. It allows developers to define tasks and their dependencies, and it automatically manages data transfers and scheduling across different processing units (CPUs, GPUs, etc.).

The StarPU programming model consists of:

- **Codelets:** Functions that define the computation to be performed. They can be executed on different architectures (CPU, GPU).
- **Data descriptors:** Structures that describe the data used by tasks, including memory location and size.
- **Tasks:** Units of work that can be scheduled and executed. They are defined by codelets and data descriptors.
- **Scheduler:** Manages the execution of tasks, taking care of data dependencies and resource allocation.
- **Data transfer:** StarPU automatically handles data transfers between different memory spaces (e.g., CPU RAM and GPU VRAM) as needed.

StarPU provides a flexible API for defining tasks, data descriptors, and scheduling policies. It supports various architectures, including multi-core CPUs and GPUs, and allows for fine-grained control over task execution.

- **starpu_init():** Initializes the StarPU runtime.
- **starpu_data_register():** Registers data descriptors for tasks.
- **starpu_task_insert():** Inserts tasks into the scheduler with specified data dependencies.
- **starpu_task_wait_for_all():** Blocks until all previously defined tasks are completed.
- **starpu_shutdown():** Shuts down the StarPU runtime and cleans up resources.
- **starpu_data_register/starpu_vector_data_register():** Registers scalar and vector data descriptors for tasks, allowing for efficient handling of variable or large arrays.
- **starpu_codelet:** Defines a codelet with CPU and/or GPU functions, number of buffers, and access modes (read/write).

7.2.1 Matrix multiplication example with StarPU

```c
#include <starpu.h>
void matmul_cpu(void *args) {
    // CPU implementation of matrix multiplication
    // args contains pointers to matrices A, B, C and their size N
    // STARPU_VECTOR_GET_PTR allow getting the pointer to the data in the vector descriptor
    // STARPU_VECTOR_GET_NX gives the size of the vector
    float *A = (float *)STARPU_VECTOR_GET_PTR(args[0]);
    float *B = (float *)STARPU_VECTOR_GET_PTR(args[1]);
    float *C = (float *)STARPU_VECTOR_GET_PTR(args[2]);
    int N = STARPU_VECTOR_GET_NX(args[0]); // Assuming square matrices
    ...
}
void matmul_gpu(void *args) {
    // GPU implementation of matrix multiplication
    // args contains pointers to matrices A, B, C and their size N
    float *A = (float *)STARPU_VECTOR_GET_PTR(args[0]);
    float *B = (float *)STARPU_VECTOR_GET_PTR(args[1]);
    float *C = (float *)STARPU_VECTOR_GET_PTR(args[2]);
    int N = STARPU_VECTOR_GET_NX(args[0]); // Assuming square matrices
    ...
}
int main() {
    starpu_init(NULL);

    // Define task descriptors for CPU and GPU implementations
    struct starpu_codelet cl_cpu = {
        .cpu_funcs = {matmul_cpu},
        .nbuffers = 3,
        .modes = {STARPU_R, STARPU_R, STARPU_W}
    };
    struct starpu_codelet cl_gpu = {
        .gpu_funcs = {matmul_gpu},
        .nbuffers = 3,
        .modes = {STARPU_R, STARPU_R, STARPU_W}
    };

    double N = 1024; // Size of the matrices
    float *A = (float *)malloc(N * N * sizeof(float));
    float *B = (float *)malloc(N * N * sizeof(float));
    float *C = (float *)malloc(N * N * sizeof(float));
    // Initialize matrices A and B
    for (int i = 0; i < N * N; i++) {
        A[i] = rand() % 100;
        B[i] = rand() % 100;
        C[i] = 0.0f; // Initialize C to zero
    }

    // Create data descriptors for matrices A, B, C
    starpu_data_handle_t A_handle, B_handle, C_handle;
    starpu_vector_data_register(&A_handle, STARPU_MAIN_RAM, (uintptr_t)A, N * N, ...
     sizeof(float));
    starpu_vector_data_register(&B_handle, STARPU_MAIN_RAM, (uintptr_t)B, N * N, ...
     sizeof(float));
    starpu_vector_data_register(&C_handle, STARPU_MAIN_RAM, (uintptr_t)C, N * N, ...
     sizeof(float));

    // Insert tasks into the StarPU scheduler
    starpu_task_insert(&cl_cpu, STARPU_RW, A_handle, STARPU_RW, B_handle, STARPU_W, C_handle);
    starpu_task_insert(&cl_gpu, STARPU_RW, A_handle, STARPU_RW, B_handle, STARPU_W, C_handle);

    // Wait for all tasks to complete
    starpu_task_wait_for_all();

    // Cleanup
    starpu_data_unregister(A_handle);
    starpu_data_unregister(B_handle);
    starpu_data_unregister(C_handle);

    starpu_shutdown();

    free(A);
    free(B);
    free(C);
}
```

7.2.2 Exercice

Implement a panel-based matrix multiplication algorithm using the task-based programming model described above. The algorithm should include two loops to iterate over the panels of the matrices, and each panel multiplication should be defined as a separate task.

7.3 Implementation with Specx

Specx is a task-based runtime system that provides a programming model for heterogeneous computing. It allows developers to define tasks and their dependencies, and it automatically manages data transfers and scheduling across different processing units (CPUs, GPUs, etc.).

The Specx programming model consists of:

- **Task Graphs:** A directed acyclic graph (DAG) where nodes represent tasks and edges represent data dependencies.
- **Tasks:** Units of work that can be scheduled and executed. They are defined by their input and output data dependencies.
- **Data Views:** Abstractions for accessing data in different memory spaces (e.g., CPU RAM, GPU VRAM).
- **Schedulers:** Manage the execution of tasks, taking care of data dependencies and resource allocation.
- **Compute Engines:** Provide the execution context for tasks, allowing them to run on different architectures (CPU, GPU).

Specx provides a flexible API for defining task graphs, tasks, and data views. It supports various architectures, including multi-core CPUs and GPUs, and allows for fine-grained control over task execution.

- **SpTaskGraph:** Represents a task graph where tasks can be added and dependencies defined.
- **SpComputeEngine:** Manages the execution of tasks on different architectures (CPU, GPU).
- **SpDeviceDataView:** Represents data that can be accessed by tasks, allowing for efficient data transfers.
- **SpWrite/SpRead:** Macros to specify read/write access to data in tasks.
- **SpCpu/SpCuda:** Macros to define CPU and GPU implementations of tasks.
- **SpWorkerTeamBuilder:** Utility to create teams of workers (CPU and GPU) for task execution.
- **SpAbstractScheduler:** An abstract class for schedulers that manage task execution and data transfers.

- **SpMultiPrioScheduler:** A specific scheduler that supports multiple priorities and locality preferences for task execution.

7.3.1 Matrix multiplication example with Specx

```
int main() {
    std::unique_ptr<SpAbstractScheduler> scheduler = ...
     std::unique_ptr<SpAbstractScheduler>(new ...
     SpMultiPrioScheduler<MaxNbDevices,FavorLocality>(nbGpu*SpCudaUtils::GetDefaultNbStreams()));
    SpComputeEngine ce(SpWorkerTeamBuilder::TeamOfCpuGpuWorkers(nbCpus, nbGpu), ...
     std::move(scheduler));

    SpTaskGraph<SpSpeculativeModel::SP_NO_SPEC> tg;
    tg.computeOn(ce);

    int N = 1024; // Size of the matrices
    SpBlas::Block A(N,N);
    SpBlas::Block B(N,N);
    SpBlas::Block C(N,N);
    // Initialize matrices A and B
    for (int i = 0; i < N * N; i++) {
        A[i] = rand() % 100;
        B[i] = rand() % 100;
        C[i] = 0.0f; // Initialize C to zero
    }

    tg.task(SpWrite(C),
            SpRead(A), SpRead(B),
        SpCpu([inBlockDim](SpBlas::Block& blockC, const SpBlas::Block& blockA, const ...
     SpBlas::Block& blockB){
        })
    , SpCuda([inBlockDim](SpDeviceDataView<SpBlas::Block> paramC, const ...
     SpDeviceDataView<const SpBlas::Block> paramA,
                        const SpDeviceDataView<const SpBlas::Block> paramB) {
            // paramA.getRawPtr(), paramA.getRawSize()
        })
    );

    tg.waitAllTasks();

    return 0;
}
```

7.3.2 Exercice

Implement a panel-based matrix multiplication algorithm using the task-based programming model described above. The algorithm should include two loops to iterate over the panels of the matrices, and each panel multiplication should be defined as a separate task.

Chapter 8

Algorithms

Performance optimization in heterogeneous computing often involves choosing the right algorithms and data structures that can efficiently utilize the available hardware resources. Therefore, it is important to understand the complexity of the algorithms and how they can be adapted to run on different architectures. In addition, it is crucial to master the data structures.

It is important to understand that the algorithm with the best theoretical complexity is not always the best in practice. The actual performance depends on many factors, such as the hardware architecture, the data access patterns, and the implementation details. When comparing two algorithms, there is a input size for which the algorithm with the best theoritical complexity will be faster than the other one, but this size may be larger than the size of the data we are working with (or even impossible to reach on our existing hardware).

8.1 Data Structures

Data structures are essential for organizing and managing data efficiently. The choice of data structure can significantly impact the performance of algorithms, especially in heterogeneous computing environments where different architectures may have varying strengths and weaknesses.

Data structure: A specialized format for organizing, processing, and storing data. Examples include arrays, linked lists, trees, graphs, and hash tables.

Complexity: A measure of the computational resources required by an algorithm, typically expressed in terms of time (how long it takes to run) and space (how much memory it uses).

Algorithm: A step-by-step procedure or formula for solving a problem or performing a computation.

Big O notation: A mathematical notation used to describe the upper bound of an algorithm's complexity, focusing on the worst-case scenario as the input size grows.

Array: A collection of elements identified by index or key, stored in contiguous memory locations. Arrays provide fast access to elements but have fixed sizes. Complexity: O(1) for access, O(n) for insertion/deletion (unless at the end).

Linked list: A collection of nodes where each node contains data and a reference to the next node. Linked lists allow dynamic memory allocation but have slower access times. Complexity: O(n) for access, O(1) for insertion/deletion (at head).

Tree: A hierarchical data structure with nodes connected by edges, where each node has a parent and zero or more children. Trees are used for hierarchical data representation. Complexity: O(log n) for balanced trees (e.g., AVL, Red-Black), O(n) for unbalanced trees.

Graph: A collection of nodes (vertices) connected by edges. Graphs can be directed or undirected and are used to represent relationships between entities. Complexity: O(V + E) for traversal algorithms like BFS/DFS, where V is the number of vertices and E is the number of edges.

Hash table: A data structure that maps keys to values using a hash function. Hash tables provide fast access and insertion but can suffer from collisions. Complexity: O(1) on average for access, insertion, and deletion; O(n) in the worst case due to collisions.

Priority queue: A data structure that stores elements with associated priorities, allowing for efficient retrieval of the highest (or lowest) priority element. Complexity: O(log n) for insertion and deletion, O(1) for access to the highest priority element.

Stack: A linear data structure that follows the Last In First Out (LIFO) principle, allowing insertion and deletion at one end only. Complexity: O(1) for push and pop operations.

Queue: A linear data structure that follows the First In First Out (FIFO) principle, allowing insertion at one end and deletion at the other. Complexity: O(1) for enqueue and dequeue operations.

Set: A collection of unique elements, typically implemented using hash tables or trees. Sets allow for efficient membership testing and operations like union and intersection. Complexity: O(1) for membership testing in hash sets, O(log n) for tree-based sets.

8.2 Algorithm Complexity

Time complexity: A measure of the amount of time an algorithm takes to complete as a function of the input size, often expressed using Big O notation.

Space complexity: A measure of the amount of memory an algorithm uses as a function of the input size, also expressed using Big O notation.

Worst-case complexity: The maximum time or space required by an algorithm for any input of a given size.

Average-case complexity: The expected time or space required by an algorithm for a random input of a given size.

Best-case complexity: The minimum time or space required by an algorithm for any input of a given size.

Sorting algorithms: Algorithms that arrange elements in a specific order (e.g., ascending or descending). Common sorting algorithms include:

- **Bubble sort:** O(n^2) time complexity, simple but inefficient for large datasets.
- **Insertion sort:** O(n^2) time complexity, efficient for small or nearly sorted datasets.
- **Selection sort:** O(n^2) time complexity, inefficient for large datasets.
- **Merge sort:** O(n log n) time complexity, stable and efficient for large datasets.
- **Quick sort:** O(n log n) average-case time complexity, O(n^2) worst-case, efficient for large datasets.
- **Heap sort:** O(n log n) time complexity, uses a binary heap data structure.
- **Radix sort:** O(nk) time complexity, where k is the number of digits in the largest number, efficient for fixed-length integers.

Search algorithms: Algorithms that find specific elements in a data structure. Common search algorithms include:

- **Linear search:** O(n) time complexity, checks each element sequentially.
- **Binary search:** O(log n) time complexity, requires sorted data, divides the search space in half at each step.
- **Hashing:** O(1) average-case time complexity for lookups, uses a hash table to map keys to values.
- **Depth-first search (DFS):** O(V + E) time complexity for graphs, explores as far as possible along each branch before backtracking.
- **Breadth-first search (BFS):** O(V + E) time complexity for graphs, explores all neighbors at the present depth prior to moving on to nodes at the next depth level.

Graph algorithms: Algorithms that operate on graphs, such as:

- **Dijkstra's algorithm:** O((V + E) log V) time complexity, finds the shortest path from a source vertex to all other vertices in a weighted graph.
- **Floyd-Warshall algorithm:** O(V^3) time complexity, finds shortest paths between all pairs of vertices in a weighted graph.
- **Kruskal's algorithm:** O(E log E) time complexity, finds a minimum spanning tree for a connected, undirected graph.
- **Prim's algorithm:** O(E log V) time complexity, also finds a minimum spanning tree for a connected, undirected graph.

Dynamic programming: A method for solving complex problems by breaking them down into simpler subproblems, storing the results of subproblems to avoid redundant calculations. Examples include:

- **Fibonacci sequence:** O(n) time complexity using memoization or tabulation.
- **Knapsack problem:** O(nW) time complexity, where n is the number of items and W is the maximum weight capacity.
- **Longest common subsequence:** O(mn) time complexity, where m and n are the lengths of the two sequences.

8.3 Algorithm Design Techniques

Algorithm design techniques are systematic approaches to solving computational problems. They provide frameworks for developing efficient algorithms that can be applied across various domains. Here are some common algorithm design techniques:

Divide and conquer: A technique that breaks a problem into smaller subproblems, solves each subproblem independently, and combines their solutions to solve the original problem. Examples include merge sort and quick sort.

Greedy algorithms: A technique that builds a solution incrementally, always choosing the next piece that offers the most immediate benefit. Greedy algorithms are often used for optimization problems, such as finding the minimum spanning tree or the shortest path in a graph.

Backtracking: A technique that explores all possible solutions to a problem by incrementally building candidates and abandoning those that fail to satisfy the constraints. Backtracking is commonly used in combinatorial problems, such as the N-Queens problem or solving Sudoku puzzles.

Dynamic programming: A technique that solves problems by breaking them down into simpler subproblems, storing the results of subproblems to avoid redundant calculations. Dynamic programming is often used for optimization problems, such as the knapsack problem or finding the longest common subsequence.

Branch and bound: A technique that systematically explores the solution space by dividing it into smaller subproblems and using bounds to eliminate subproblems that cannot yield better solutions than the best found so far. Branch and bound is often used for combinatorial optimization problems, such as the traveling salesman problem.

Randomized algorithms: Algorithms that use random numbers to make decisions during execution. Randomized algorithms can provide efficient solutions to problems that are difficult to solve deterministically, such as randomized quick sort or Monte Carlo methods.

Heuristic algorithms: Techniques that find approximate solutions to complex problems by using rules of thumb or educated guesses. Heuristic algorithms are often used when exact solutions are impractical, such as in optimization problems like the traveling salesman problem or job scheduling.

Iterative improvement: A technique that starts with an initial solution and iteratively refines it to improve its quality. This approach is often used in optimization problems, such as hill climbing or simulated annealing.

8.4 Algorithm Puzzles

To practice algorithm design and implementation, it is higly beneficial to solve algorithm puzzles. These puzzles often require creative thinking and a deep understanding of algorithms and data structures. Here are some classic algorithm puzzles to consider:

- **Two Sum:** Given an array of integers and a target sum, find two numbers in the array that add up to the target. This puzzle can be solved using a hash table for efficient lookups.

- **Longest Substring Without Repeating Characters:** Given a string, find the length of the longest substring without repeating characters. This puzzle can be solved using a sliding window technique.
- **Valid Parentheses:** Given a string containing just the characters '(', ')', '', '', '[' and ']', determine if the input string is valid. This puzzle can be solved using a stack to track opening parentheses.
- **Kth Largest Element in an Array:** Find the kth largest element in an unsorted array. This puzzle can be solved using quick select or a min-heap.
- **Coin Change:** Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the target amount. This puzzle can be solved using dynamic programming.
- **Implement a stack with min function:** Design a stack that supports push, pop, top, and retrieving the minimum element in constant time. This puzzle can be solved by maintaining an auxiliary stack to track the minimum elements.
- **Implement a queue using stacks:** Design a queue that supports enqueue and dequeue operations using two stacks. This puzzle can be solved by reversing the order of elements between the two stacks.

It is important to understand that solving these puzzles requires not only knowledge of algorithms and data structures but also practice. Practice. And Practice. The more you practice, the better you will become at solving algorithm puzzles and designing efficient algorithms.

Chapter 9

C++

Overview

C++ is a general-purpose programming language that combines low-level memory control with high-level abstractions. It offers features such as classes, templates, operator overloading, and RAII (Resource Acquisition Is Initialization), enabling developers to write both high-performance and maintainable code.

In performance-critical domains (game engines, high-frequency trading, scientific computing), understanding how C++ manages memory, inlines functions, handles temporaries, and dispatches calls is essential. This chapter explores:

- Memory management: raw pointers, RAII, smart pointers
- Function inlining and its costs/benefits
- Temporary objects and move semantics
- Virtual functions vs. static polymorphism
- Container choice: complexity and memory layout
- STL algorithms for efficiency
- Compiler optimizations and PGO
- High-performance libraries (BLAS, LAPACK, FFTW, etc.)

9.1 Memory Management

9.1.1 Stack vs. Heap Allocation

Local (stack) allocation is extremely fast and automatically freed:

```cpp
// Stack allocation: no explicit delete needed
void foo() {
    int a[100];         // contiguous block on the stack
    std::vector<int> v; // v's internal buffer still on the heap
}
```

Heap allocation gives control at the cost of manual cleanup:

```cpp
void bar() {
    int* p = new int[100]; // allocate 100 ints on the heap
    // ...
    delete[] p;            // must remember to free
}
```

As a result, prefer stack allocation when possible and ideally when frequently used.

9.1.2 RAII and Smart Pointers

RAII binds lifetime to scope. The STL provides smart pointers:

```cpp
#include <memory>

void baz() {
    std::unique_ptr<MyClass> uPtr(new MyClass(42));
    // automatically deleted when uPtr goes out of scope

    auto sPtr = std::make_shared<MyClass>(7);
    // shared ownership, reference-counted
}
```

9.2 Inlining

Marking a function `inline` suggests replacing calls with body code:

```cpp
inline int add(int x, int y) {
    return x + y;
}

int main() {
    int z = add(3, 4); // expanded to z = 3 + 4;
}
```

Trade-off:

- *Benefit*: removes call/return overhead
- *Cost*: increases code size (potentially hurting I-cache)

9.3 Temporary Objects and Move Semantics

C++ often creates temporaries which can hurt performance:

```cpp
std::string make_name() {
    return "temporary"; // constructs a temporary std::string
}

void qux() {
    std::string name = make_name(); // copy or move
}
```

```cpp
std::vector<int>::iterator it = v.begin();
for(; it != v.end(); ++it) { // v.end() creates a temporary iterator
    // do something with *it
}
```

Use move semantics to avoid copies:

```cpp
std::string s1 = "hello";
std::string s2 = std::move(s1); // s1 is now empty; s2 takes ownership
```

9.4 Polymorphism and Function Pointers

9.4.1 Virtual Functions

Virtual dispatch uses a vtable lookup at runtime:

```cpp
struct Base {
    virtual void f() { /*...*/ }
};

struct Derived : Base {
    void f() override { /*...*/ }
};

void call(Base* b) {
    b->f(); // dynamic dispatch via vtable
}
```

Overhead: one extra indirection per call.

9.4.2 Static Polymorphism (CRTP)

Templates achieve compile-time ”polymorphism” without vtables:

```cpp
template <typename T>
struct CRTP {
    void f() { static_cast<T*>(this)->f_impl(); }
};

struct Impl : CRTP<Impl> {
    void f_impl() { /*...*/ }
};

void demo() {
    Impl i;
    i.f(); // resolves to Impl::f_impl() directly
}
```

9.5 Containers

9.5.1 Complexity Analysis

Container	Amortized Complexity
`std::vector<T>`	push_back: O(1), random access: O(1)
`std::list<T>`	insert/erase: O(1) (given iterator)
`std::deque<T>`	push/pop both ends: O(1)
`std::map<Key,T>`	lookup/insert: O(log n)
`std::unordered_map<Key,T>`	average lookup/insert: O(1)

9.5.2 Data Layout

- **Vector**: contiguous memory ⇒ excellent cache locality.
- **List**: nodes scattered in memory ⇒ poor locality but cheap splicing.
- **Deque**: blocks of contiguous memory.

9.6 Algorithms

The STL algorithms are heavily optimized:

```cpp
#include <algorithm>
#include <numeric>
#include <vector>

void algos() {
    std::vector<int> v = {5, 2, 8, 1, 4};
    std::sort(v.begin(), v.end());                   // O(n log n)
    int sum = std::accumulate(v.begin(), v.end(), 0); // O(n)
    std::transform(v.begin(), v.end(), v.begin(),
                   [](int x){ return x * x; });
}
```

9.7 Compiler Optimizations

9.7.1 Optimization Flags

- `-O2` / `-O3`: general speed tuning
- `-march=native`: enable CPU-specific instructions
- `-flto`: link-time optimization

9.7.2 Profile-Guided Optimization (PGO)

1. Compile with instrumentation (`-fprofile-generate`)
2. Run typical workloads to gather data
3. Recompile with feedback (`-fprofile-use`)

9.8 High-Performance Libraries

When possible, leverage battle-tested libraries:

- **BLAS** (Basic Linear Algebra Subprograms): Standardized API for vector/matrix ops. E.g., DGEMM for matrix multiply:

```c
#include <cblas.h>

void matmul(const double* A, const double* B, double* C,
            int N) {
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                N, N, N,
                1.0, A, N,
                     B, N,
                0.0, C, N);
}

```

- **LAPACK**: builds on BLAS for solving linear systems, eigenproblems.
- **FFTW** (Fastest Fourier Transform in the West): Highly optimized FFTs on many platforms.
- **Others**: Intel MKL, Eigen, cuBLAS/cuFFT (GPU).

9.9 Exercice

Optimize the following C++ code by either reducing the complexity of the algorithm or improving overall execution (avoid unnecessary copies, use move semantics, etc.).

```cpp
    #include <iostream>
#include <vector>

std::vector<int> doubleAndSort(const std::vector<int> input) {
    std::vector<int> temp;

    for (size_t i = 0; i < input.size(); ++i) {
        int doubled = input[i] * 2;
        temp.push_back(doubled);
    }

    for (size_t i = 0; i < temp.size(); ++i) {
        for (size_t j = 0; j < temp.size() - 1; ++j) {
            if (temp[j] > temp[j + 1]) {
                int t = temp[j];
                temp[j] = temp[j + 1];
                temp[j + 1] = t;
            }
        }
    }

    return temp;
}

int main() {
    std::vector<int> data = {5, 3, 8, 1};
    std::vector<int> result = doubleAndSort(data);

    for (int x : result) {
        std::cout << x << " ";
    }
    std::cout << std::endl;
}
```

Chapter 10

Measuring Performance

Overview

Measuring performance is essential for optimizing algorithms and understanding code efficiency. In this chapter, we introduce methods and tools for performance measurement, including profiling, benchmarking, and analysis of memory access patterns. Every optimization effort should be motivated and validated by performance data; this is the only way to determine whether an optimization is effective and worth the added complexity.

For example, you might notice a particular kernel appearing slow during execution and feel tempted to optimize it. However, if that kernel accounts for only 1% of the total runtime, optimizing it yields negligible benefit and can unnecessarily complicate the code.[1]

Performance is typically evaluated using one or more target metrics - such as execution time or energy consumption - that quantify an application's efficiency. By measuring these metrics at fine granularity (functions, loops, or even lines of code), we can locate *hotspots*: the code regions responsible for the bulk of resource usage and thus prime candidates for optimization. But raw timing or energy results alone are often insufficient to understand bottlenecks. Therefore, we also collect hardware and software counters - instruction counts, cache-miss rates, memory-access counts, branch-misprediction rates, etc. - to shed light on the underlying causes of performance limitations and guide our optimization strategy.

10.1 Profiling

Profiling analyzes a program's runtime behavior to pinpoint bottlenecks and guide optimizations. Profilers report metrics such as function-level execution time, memory usage, and hardware event counts. Common profiling tools include:

- **gprof**: GNU profiler that reports per-function execution times and call graphs.
- **perf**: Linux tool for CPU profiling, hardware event counting (e.g., cache misses), and tracing.
- **Valgrind**: Framework for memory debugging, leak detection, and profiling (e.g., Callgrind).
- **Intel VTune**: Detailed CPU and memory-access profiling on Intel architectures, including threading analysis.
- **NVIDIA Nsight**: GPU profiling and debugging suite for NVIDIA platforms; reports kernel times, memory throughput, and occupancy.

Key profiling metrics include:

Execution time: Total elapsed time for a program or a specific function.

CPU usage: Fraction of CPU cycles consumed by the program, highlighting CPU-bound behavior.

[1]Exceptions exist in high-frequency trading or real-time systems, where every microsecond matters.

Memory usage: Amount of RAM allocated and accessed, indicating memory-bound operations.

Cache misses: Number of accesses that miss the cache and go to main memory, revealing cache-bound hotspots.

Branch mispredictions: Count of incorrect branch-prediction events, highlighting control-flow inefficiencies.

Instruction count: Total number of CPU instructions executed, useful for identifying instruction-bound sections.

Memory accesses: Total load/store operations, showing data-movement intensity.

Thread contention: Occurrences of threads waiting on shared resources, indicating synchronization bottlenecks.

Energy consumption: Total energy used, important for power- or battery-constrained systems.

Power consumption: Instantaneous or average power draw, relevant to thermal and energy-efficiency studies.

Profiling granularity can range from coarse (whole-program) to fine (per-line). Choose the level that best isolates the performance issue at hand and is supported by your tools.

10.2 Parallel Efficiency Metrics

When scaling to multiple processors or cores, additional metrics help assess parallel performance:

Speedup

Speedup quantifies how much faster a parallel implementation runs compared to sequential:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$$

where $T_{\text{sequential}}$ is the execution time on one core and T_{parallel} is the time on P cores.

Efficiency

Efficiency measures resource utilization in parallel runs:

$$\text{Efficiency} = \frac{\text{Speedup}}{P}$$

with P the number of processors. Ideal efficiency is 1 (or 100%).

Load Balancing

A balanced workload ensures all processors finish tasks at similar times. We can quantify imbalance as the variance of per-processor runtimes:

$$\text{Imbalance} = \frac{1}{P}\sum_{i=1}^{P}\left(T_i - \bar{T}\right)^2$$

where T_i is the runtime on processor i and $\bar{T}$ is the mean.

Bandwidth

Bandwidth measures data transfer capacity between memory and CPU or among processors:

$$\text{Bandwidth} = \frac{\text{Total data transferred}}{\text{Total transfer time}}$$

typically in bytes per second (B/s).

Roofline Model

The roofline model graphically contrasts an algorithm's arithmetic intensity (operations per byte of memory transfer) against hardware ceilings of peak compute and memory bandwidth. It helps identify whether performance is limited by computation or data movement and guides where optimizations (e.g., increasing locality or parallelism) will have the greatest impact.

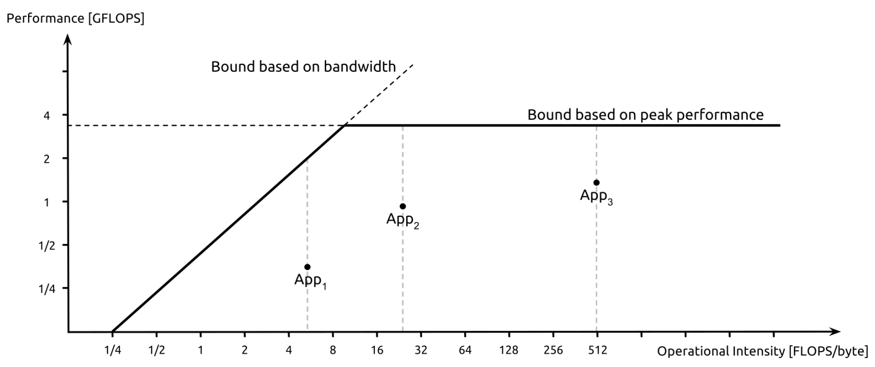


Figure 10.1: Example Roofline Model (from wikipedia)

Example roofline model parameters:

- Peak compute performance: 100 GFLOPS (or 10^{11} FLOP/s)
- Memory bandwidth: 20 GB/s (or 2×10^{10} B/s)
- Arithmetic intensity for matrix multiplication: 2 FLOP/byte

Example of code to compute the arithmetic intensity:

```cpp
for(int idx = 0 ; idx < N; ++idx) {
    // Perform some computation
    double result = 0.0;
    for(int j = 0; j < M; ++j) {
        result += A[idx][j] * B[j];
    }
    C[idx] = result;
}
// Calculate arithmetic intensity
int loads = N * M; // Assuming A is accessed M times per row
int stores = N; // One store per row in C
double arithmetic_intensity = (2.0 * N * M) / (loads + stores);
std::cout << "Arithmetic Intensity: " << arithmetic_intensity << " FLOP/byte" << std::endl;
```

10.3 Exercice

Provide the speedup and parallel efficiency of the following parallel implementation of matrix multiplication, given the following execution times:

- Sequential execution time: 100 seconds

- Parallel execution time on 4 cores: 30 seconds

Compute the load balancing if the execution times on each core are:

- Core 1: 30 seconds
- Core 2: 32 seconds
- Core 3: 28 seconds
- Core 4: 30 seconds

Compute the bandwidth if the total data transferred during the parallel execution is 1 GB and the total transfer time is 30 seconds.

Draw a roofline model for a hypothetical hardware with a peak compute performance of 100 GFLOPS and a memory bandwidth of 20 GB/s. Assume the arithmetic intensity of the matrix multiplication is 2 FLOP/byte.