PARALLEL COMPUTING exam questions PT1

Claudio Tessa

January 29, 2026

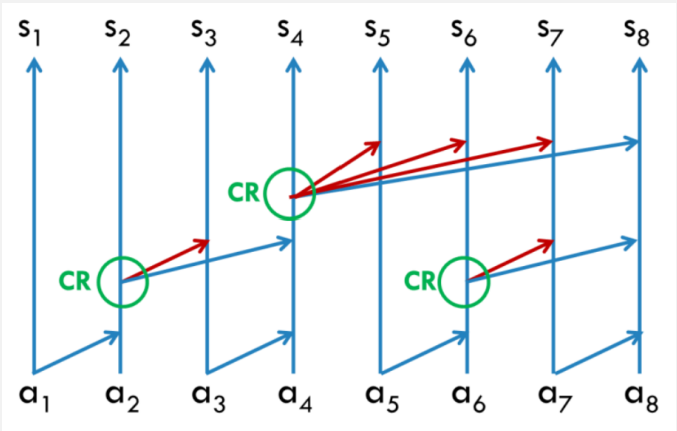
Contents

1 Exercises 1

1 What is the difference between EREW, CREW, and CRCW models? Provide an example for each of them.

- EREW (Exclusive Read Exclusive Write):
  - No two processors can read from or write to the same memory cell at the same time.
  - Example: Copying an array
- CREW (Concurrent Read Exclusive Write):
  - Multiple processors can read the same memory cell simultaneously, but only one can write to a given cell at a time.
  - Example: Matrix-vector multiplication
- CRCW (Concurrent Read Concurrent Write):
  - Multiple processors can read from and write to the same memory cell simultaneously. Write conflicts are resolved by a rule (e.g., common, priority, arbitrary).
  - Example: DNF computation where the variables read can be any of the input variables.

2 How can the prefix sum of an array be computed on a CREW PRAM using idle processors from the sum algorithm?



3 Given an algorithm with $T_1(n) = n^2$ and $T_p(n) = \frac{n^2}{p}$, analyze the speedup and efficiency. What can you conclude?

- SPEEDUP: $SU_p = \dfrac{T_1(n)}{T_p(n)} = p$
- EFFICIENCY: $E_p = \dfrac{T_1(n)}{pT_p(n)} = 1$

Conclusion: the algorithm achieves linear speedup in the best case and perfect efficiency.

4 What is the main conceptual difference between Amdahl's Law and Gustafson's Law?

Amdahl assumes a fixed problem size and shows the limits to the speedup as we increase the number of processors.

Gustafson assumes a fixed execution time and scales the problem size, demonstrating that the speedup can grow linearly with the number of processors.

5 On a PRAM model, the time complexity can be equal to the number of instructions executed. True/False? Why?

TRUE.

When $P = 1$, the PRAM model behaves like a standard sequential RAM model. In this case, since no operations can be performed in parallel, the time complexity is equal to the number of instructions executed.

6 Please describe how the matrix vector multiplication can be parallelized on a PRAM model. Please report also speedup, efficiency, work and cost.

We want to calculate $Y = AX$. On a PRAM model, Matrix-Vector Multiplication is parallelized by decomposing the matrix A into blocks $A_i, i = 1, ..., p$, where p is the number of processors:

1. processors concurrently read X
2. processors simultaneously read their section of A (A_i)
3. compute n^2/p operations per processor to get Y_i
4. each processor writes its Y_i (n/p elements per processor) simultaneously without conflicts

The algorithm achieves:

- $T_1 = O(N^2)$ $\qquad$ $T_p(N^2) = O(N^2/p)$

- SPEEDUP: $SU = \dfrac{T_1}{T_p} = \dfrac{O(N^2)}{O(N^2/p)} = p \implies$ linear speedup
- EFFICIENCY: $E_p = \dfrac{T_1}{pT_p} = \dfrac{O(N^2)}{O\left(p\frac{N^2}{p}\right)} = 1 \implies$ perfect efficiency
- COST: $O\left(p\frac{N^2}{p}\right) = O\left(N^2\right)$
- WORK: N^2

7 Among the laws we have seen during the course, which one is classified to address the weak scaling? Please also describe the assumptions that come with the identified law.

Gustafson's Law addresses weak scaling. It assumes that the problem size scales with the number of processors, maintaining a constant execution time. The formula is

$$SU = serial + parallel \cdot P$$

8 Please describe the policies adopted in a PRAM model when a CW is present.

To solve write conflicts:

- **Priority CW**: processors have priorities. The highest priority is allowed to complete write.
- **Common CW**: all processors are allowed to complete write if and only if all the values to be written are equal.
- **Arbitrary/Random CW**: one randomly chosen processor is allowed to complete write.

9 What can be said about the time complexity of a PRAM-based application when the PRAM model has a bounded number of shared memory cells?

The time complexity typically increases because the limited memory restricts the communication between processors. Since processors must use shared memory to exchange data, having fewer cells than the problem requires forces the algorithm to serialize communication.

10 In which way does the following assumption impact the PRAM analysis: "Each memory cell can hold an integer of unbounded size"?

Since PRAM is a theoretical mode in which communication occurs exclusively via shared memory, an unbounded cell size theoretically allows a processor to **transmit an infinite amount of data** in a single time step.

This also abuses **Bit Level Parallelism**, as it technically allows a processor to pack multiple data elements into one integer and process them all simultaneously with a single arithmetic operation in $O(1)$ time.

These are obviously not realistic on physical machines.

11 Could you please classify the READ/WRITE abilities of a PRAM system and how realistic and useful they are?

- **Exclusive Read (ER)**: Processors can simultaneously read only from distinct memory locations;
- **Exclusive Write (RW)**: Processors can simultaneously write only to distinct memory locations;
- **Concurrent Read (CR)**: Processors can simultaneously read from any memory location (also the same one);
- **Concurrent Write (CW)**: processors can simultaneously write to any memory location (also the same one).

And combinations of both (CREW, EREW, ...).

Especially CRCW is not very realistic, as it is a theoretical model that assumes no overhead and ignores physical limitations of the hardware (latency, circuit complexity, etc.).

Still they are very useful as they serve as a baseline for algorithm design (if a problem has no solution on PRAM, then it has no solution on any parallel machine).

12 Assuming $T^*(n) = T_1(n)$, what kind of relation do you have between the efficiency and the speedup of a PRAM-based application (direct, inverse,...)?

We have

- SPEEDUP $SU_p(n) = \dfrac{T_1(n)}{T_p(n)}$
- EFFICIENCY $E_p(n) = \dfrac{T_1(n)}{pT_p(n)}$

$\implies E_p(n) = \dfrac{SU_p(n)}{p} \implies$ efficiency is directly proportional to speedup.

13 Could you please define the speedup in a fixed-time model?

The speedup in a fixed-time model is defined by Gustafson's law:

$$SU = s + P(1 - s)$$

where:

- s is the time spent on the serial portion of the problem;
- $(1 - s)$ is the time spent on the parallel portion;
- P is the number of processors.

14 Please discuss the lemma deriving the slow-down factor when we move from an unbounded memory PRAM model to a bounded one.

Assume a problem can be solved on an unbounded memory PRAM needing M cells, and we move to a M'-cell PRAM, with $M' < M$:

Any problem that can be solved by a P-processor and M-cell PRAM in T steps, can be solved on a $\max(P, M')$-processor and M'-cell PRAM in $O\left(\frac{TM}{M'}\right)$ steps.

15 Complete the following sentence and explain what it means: "In a PRAM model, a read conflict occurs when..."

...when multiple processors attempt to read from the same memory location at the same time.

Whether or not this is allowed depends on the specific PRAM variant:

- Exclusive Read (ER): conflicts are illegal and the algorithm fails.
- Concurrent Read (CR): conflicts are allowed and the value is read by both processors.

16 Give a definition of the speedup when a PRAM model is considered.

For a PRAM model, the speedup on p proceessors, SU_p, is defined as:

$$SU_p = \frac{T^*}{T_p}$$

where:

- T^* is the execution time of the best known sequential algorithm;
- T_p is the execution time of the parallel algorithm using P processors.

17 Please outline the key aspects of Amdahl's Law.

Amdahl's law is used to calculate the theoretical maximum speedup of a task when only a portion of it can be parallelized:

$$SU = \frac{1}{1 - f + \frac{f}{p}}$$

where:

- f is the parallelizable fraction of the problem;
- p is the number of processors;

Amdahl's law addresses **strong scaling**. It gives us the speedup obtained by increasing the number of processors, but keeping the prpoblem size constant.

18 In a PRAM model, each processor has an unbounded number of registers. What purposes could such registers be used for? Is the access time different from the one for the global shared memory?

The unbounded local registers are a private storage for each processor to hold intermediate computational results, local variables, and instruction data that do not need to be visible to other processors.

In the PRAM model, the access time for accessing local registers is the same to that of accessing the global shared memory: $O(1)$.

19 Please define the Cost in a PRAM model.

The cost of a parallel algorithm for input size n running on p processors is:

$$C(n) = p \cdot T_p(n)$$

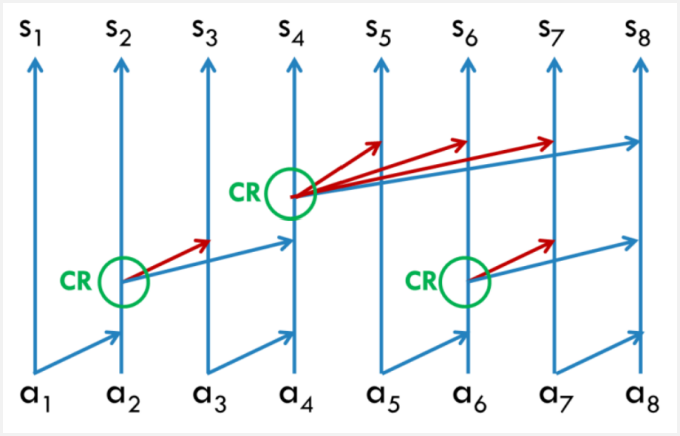
20 Could you please discuss the following sentence? "Gustafson's law presupposes that the computing requirements will stay the same, given increased processing power. In other words, an analysis of the same data will take less time given more computing power". Is it true or false? What does it mean?

FALSE.

This sentence describes Amdahl's law. Instead, Gustafson's law presupposes that the problem size (computing requirements) will scale up together with the number of processors (processing power).

In other words, if we get more powerful hardware (i.e., more processors), we use it to solve a larger and more complex problem in the **same amount of time**.

21 Please list the concurrent read the prefix sum algorithm has when a PRAM model is used.



22 Please show an example of the PRAM model solving CW with a common CW approach.

A common CW example is the **boolean DNF**. Here, CW are solved using the **arbitrary CW** rule: one randomply chosen processor is allowed to write. It works because writes are consistent (both processors would have written the same value).

23 Is the absolute serial time fixed with the Gustafson model? What about the parallel part?

Both the absolute serial and parallel time are fixed. The model assumes a **fixed-time** constraint: as you add more processors, you scale up the amount of work (problem size) proportionally. So the larger parallel workload takes the exact same amount of time, while the amount of work done increases.

24 Please list the characteristics of the PRAM model that simplify the space complexity estimations.

- **Unified Shared Memory**: assumes a single global address space accessible by all processors (no caches, registers, etc.), so space complexity is simply

calculated as the number of shared memory cells used.

- **Implicit Communication**: since communication occurs via writing to shared variables, there is no need to allocate space for message-passing buffers, etc.
- **Unbounded Capacity**: the model assumes infinite available memory. This allows the analysis to focus only on the algorithmic requirements.

25 Could you please list the properties/features PRAM has in common with CUDA?

- **Global Address Space**: just as PRAM assumes a single shared memory accessible by all processors, CUDA provides Global Memory visible to every thread in the grid, allowing implicit communication.
- **Unlimited Processors**: both models assume the availability of virtually unlimited threads (processors), abstracting away the physical core count.
- **SIMD execution**: CUDA uses uses the SIMT architecture, where multiple threads inside a warp execute the same instruction simultaneously (just like SIMD units).
- **Concurrent Write Support**: CUDA's Atomic Operations provide a practical implementation of the CRCW (Concurrent Write) PRAM model, allowing multiple threads to update the same memory location safely.

26 Please discuss the following sentence: "In a PRAM model, processors have private local memories". True/False?

FALSE.

The PRAM model is defined by a single global shared memory accessible by all processors with uniform O(1) access time. It intentionally abstracts away the concept of private local memories to simplify the analysis of parallel algorithms.

27 Please discuss PRAM prefix sum algorithm according to the Read/Write accesses.

The PRAM Prefix Sum algorithm relies on the CREW (Concurrent Read, Exclusive Write) model.

- **Concurrent Reads** because multiple processors frequently read the same source value at the same time. *Example*: at step 1, Processor i needs to read index i and $i-1$, while Processor $i+1$ also needs to read index i (and $i+1$), creating a read collision at index i.
- **Exclusive Writes** because each processor updates a unique index (Processor

i only writes to index i).

28 Please discuss the following sentence: "Amdahl's model is classified as a strong scaling model while Gustafson as a weak scaling model". True/False? Why?

TRUE.

- **Amdahl's Law** represents Strong Scaling because it assumes a fixed total problem size, measuring the speedup as you add processors.
- **Gustafson's Law** represents Weak Scaling because it assumes that the problem size scales up proportionally with the number of processors (while keeping execution time fixed).

29 Describe the steps followed by a PRAM machine.

Each processor synchronously perform:

1. Read input from an input cell
2. Read shared memory from a shared memory cell
3. Perform internal computation
4. Write to an output cell if needed
5. Write to the shared memory cell if needed

30 Assuming $P' < P$, is it possible to derive the slowdown factor when P' processors are used? How?

Using the following lemma valid when $P' < P$:

Any problem that can be solved by a P-processor PRAM in T steps, can be solved by a P'-processor PRAM in $O\left(T\frac{P}{P'}\right)$ steps.

Therefore, the slowdown factor of the new time is P/P'.

2 Exercises 2

31 Explain the difference between superscalar execution and SIMD execution.

Superscalar execution allows a processor to execute multiple independent instructions from the same instruction stream in parallel.

SIMD (Single Instruction Multiple Data) executes the same instruction on multiple data elements simultaneously.

Instruction stream coherence is crucial for SIMD because all ALUs execute the same instruction; divergence (branches) create inefficiencies. Meanwhile, superscalar processors can handle different instructions independently.

32 Why does multi-threading help hide memory latency, and how many threads are needed to achieve full utilization if each thread performs 3 arithmetic operations with a latency of one cycle, followed by a memory load with 12-cycle latency? Please motivate the answer.

Multi-threading hides memory latency by allowing the processor to switch to another thread when one is stalled.

To hide a 12-cycle latency with 3 arithmetic instructions per thread, 5 threads are needed: (3 instructions $\times$ 5 threads = 15 cycles, covering the 12-cycle stall).

33 What does it mean for a computation to be bandwidth-bound, and why is vector multiplication of large arrays an example of this?

A computation is bandwidth-bound when its performance is limited by the rate at which data can be transferred from memory, not by the processor's compute capability.

Vector multiplication involves multiple memory accesses per arithmetic operation, saturating memory bandwidth before compute units are fully utilized.

34 Write a short pseudocode that would result in worst-case SIMD efficiency on an 8-wide SIMD processor due to divergent execution.

```
forall (int i from 0 to N) {
    float t = x[i];
    if (i % 8 == 0) {
        t = t * 2.0;
    } else if (i % 8 == 1) {
        t = t + 3.0;
    } else if (i % 8 == 2) {
        t = t / 4.0;
    } else if (i % 8 == 3) {
```

```
        t = t - 5.0;
    } else if (i % 8 == 4) {
        t = sqrt(t);
    } else if (i % 8 == 5) {
        t = log(t);
    } else if (i % 8 == 6) {
        t = exp(t);
    } else {
        t = sin(t);
    }
    y[i] = t;
}
```

35 Briefly describe what "Divergent" execution" means. Provide also an example to support the description.

Divergent execution occurs in SIMT architectures (like GPUs) when threads in the same group are forced to follow different paths due to a conditional branch. Since the hardware issues the same instruction to all threads simultaneously, it cannot run different paths and it must serialize the execution.

Example:

```
if (threadIdx % 2 == 0) {
    A();
} else {
    B();
}
```

36 Describe the difference between the Message Passing and Shared Address Space programming model.

- **Shared Address Space**: all processors have access to a single global memory and communicate implicitly by reading and writing shared variables (using load/store instructions), which requires explicit synchronization (like locks) to manage access.
- **Message Passing**: each processor has its own private local memory. Communication is explicit, requiring processors to actively send and receive data packets to exchange information.

37 Can a program with many arithmetic operations and a small number of memory accesses be a problem for a multi-threaded processor? True/False. Why?

TRUE.

Multi-threaded processors hide memory latency by switching to another thread whenever the current one stalls on memory access. If a program has a small number of memory accesses, threads rarely stall, causing them to compete for the same execution units (ALUs). This prevents efficient interleaving and can degrade performance.

38 Compare Coarse-grain multithreading with Simultaneous multithreading.

- **Coarse-grained multithreading** executes instructions from a single thread at a time, switching contexts only when a long stall occurs.
- **Simultaneous Multithreading** mixes instructions from multiple threads in every clock cycle, filling almost every available execution unit.

39 Coherent execution is not necessary for efficient parallelization. True/False? When?

TRUE for MIMD (like CPUs) architectures, because each processor core has its own independent fetch and decode unit, allowing them to execute different instruction streams (divergent path) simultaneously.

FALSE for SIMD/SIMT architectures (like GPUs) where multiple execution units share a single control unit. Here divergence forces the hardware to serialize execution.

40 Three different forms of parallel execution: Superscalar, SIMD, and Multicore. Please describe the differences between these three forms of execution.

- **Superscalar**: exploits Instruction-Level Parallelism (ILP) within a single core by dynamically issuing multiple instructions from a single thread to different execution units for every clock cycle (e.g., pipelining).
- **SIMD**: exploits Data-Level Parallelism by applying a single operation to multiple data elements simultaneously.
- **Multicore**: exploits Thread-Level Parallelism (TLP) by using multiple physical processing units (cores) to execute completely independent instruction streams at the same time.

41 Please describe the tradeoffs between the silicon area dedicated to the cache and the processor contexts.

Allocating silicon area to Cache reduces the effective memory access time for individual threads (data locality).

Conversely, allocating area to Processor Contexts (e.g., large register files) allows the hardware to maintain many active threads simultaneously. This allows the processor to hide memory latency by context-switching to ready threads when active ones stall, maximizing the total throughput rather than reducing the latency of any single operation.

42 Show an example with a few lines of code where a data-parallel program allows SIMD/vectorize instructions extraction.

```
for (int i = 0; i < N; i++) {
    A[i] = B[i] + C[i];
}
```

Each iteration is completely **independent**, so the compiler can map these operations to SIMD units, executing multiple additions in a single clock cycle.

43 Please describe what stream coherence means and why it is related to SIMD processing resources.

In SIMD, **stream coherence** refers to the uniformity of operations between data elements being processed in parallel. If all SIMD units are doing the exact same thing at the same time and accessing neighbors data, then the stream is coherent. Otherwise, we have *divergence* (e.g., branches).

44 Write a short description of "Hardware-supported interleaved multi-threading."

Interleaved multi-threading is a tecnique where the CPU is designed to switch between different active threads in every cycle. *Hardware-supported* means that the hardware has physical elements on the chip to maintain multiple contexts (e.g., separate program counters and registers).

In this way we hide execution latencies (such as stalls) by executing work from other threads instead of waiting for the resource to become available. Furthermore, hardware-support allows to reduce the context-switching overhead.

45 List what a programmer can do to reduce the slow-down due to the memory stalls.

To reduce the slow-down, a programmer can:

- **avoid dependencies** where possible (loop-carried dependencies, reorder/remove instruction for flow-dependencies, etc.);
- improve **data locality**: access memory sequentially (e.g., iterate through an array linearly), use Structures Of Arrays (SoA) instead of Arrays Of Structures (AoS) to avoid loading unwanted data;
- use **multithreading** to hide latencies;
- maximize **data reuse** to avoid unnecessary reloading of data, and can even recompute the value instead of loading it (as computing is faster and does not stall).

46 "In a superscalar processor, the parallelism is automatically discovered." True/False? Why?

TRUE

The parallelism is automatically discovered by the hardware at **runtime**. The processor has logic that scans a sequential stream of instructions and identifies which ones are independent. It then dispatches multiple instructions to different execution units simultaneously within a single clock cycle.

47 Please describe the key characteristics of data-parallel program execution with SIMD instructions in the presence of conditional instructions.

In SIMD, all units execute the same instruction in parallel. In case of branching, SIMD processors use a techinique called **masking** to execude the code linearly. The processor checks the condition for all elements simultaneously, then creates a mask of true/false values (e.g., [T T F T F F F F]).

When the processor broadcasts the instructions to everyone, the active ALUs (True) perform the calculations, while the inactive ALUs (False) receive the instruction but discard the output of the calculations. If there is an else block, the processor flips the mask and broadcasts the instructions again.

48 Prefetching reduces stalls. True/False? Explain what is meant by prefetching and stall.

TRUE.

- A **stall** occurs when the processor must pause execution because the data (or instruction) required for the next step is not yet available (usually because

waiting for memory latency).

- **Prefetching** is a technique where the system predicts which data will be needed soon and loads it from the slow memory into the fast cache. This hides latency.

49 CUDA kernels may create dependencies between threads in a block. True/False?

TRUE.

Threads within the same CUDA block can interact through Shared Memory and synchronization barriers like `__syncthreads()`. This creates explicit dependencies.

For example, Thread B might wait at a barrier until Thread A has finished writing a specific value to shared memory, ensuring that Thread B reads the updated data rather than the old one.

50 Please discuss the following sentence: "The latency of the memory operation is changed by multi-threading."

False. Multi-threading does not change the physical latency of a single memory operation. that latency is determined by the hardware physics.

Instead, multi-threading is a technique used to hide latency. When one thread stalls waiting for memory, the hardware switches to another ready thread to keep the core busy, but the actual time that the memory request takes to complete generally remains the same.

51 Explain when the following sentence is not true: "A processor with multiple hardware threads has the ability to avoid stalls."

Multi-threading hides latency by switching to a "ready" thread when the current one stalls. However, if every single thread is waiting for a slow resource or if they are all waiting at a synchronization barrier, there is no valid work to switch to.

Additionally, if there are too few threads launched to mathematically cover the long wait times, the processor runs out of work before the memory fetch completes, forcing the hardware to stall despite having multi-threading capabilities.

52 What does it mean conditional execution in a data-parallel program?

Conditional execution (divergence) occurs when different execution units must take different paths through the code (e.g., some have to take the `if` branch while others the `else`).

Since SIMD hardware executes the same instruction on all units simultaneously, it cannot run both paths at once. instead, it serializes the execution (masking). The hardware first executes the `if` block (disabling threads that didn't take it), then executes the else block (disabling the others). This temporarily reduces the effective throughput of the processor.

3 Exercises 3

53 Please describe shared address space model in terms of communication abstraction and required hardware support.

Threads communicate by reading/writing to shared variables in a shared address space. This abstraction necessitates synchronization primitives (e.g., locks to ensure mutual exclusion).

Hardware implementation: there is an interconnect that allows any processor to directly reference the contents of any memory location. Interconnects can be of various types (e.g., shared bus, crossbar, etc.)

54 Why must CUDA allocate execution contexts for all threads in a block before execution begins?

Since CUDA threads in a block may synchronize (e.g., with `__syncthreads()`), all threads must be live concurrently to avoid deadlocks and ensure correct execution. This requires allocating resources (registers, shared memory) for all threads upfront.

55 Describe the distinct types of address spaces visible to kernels in a CUDA/GPU based environment. How many? How much is shared? How fast are they?

- **Per-thread private memory**: readable and writable only by the single thread that owns it (*fastest*);
- **Per-block shared memory**: readable and writable by all threads within the same block (*variable speed because they are cached, generally fast*;
- **Device global memory**: readable and writable by all threads in the entire grid (*slowest*).

56 Briefly describe how CUDA threads-block are assigned to hardware considering the V100 SM processor architecture.

The scheduler assigns each block to a streaming multiprocessor (SM). Threads within a block are divided into warps (groups of 32). Warps are the fundamental unit of execution and are assigned to one of the SM's four sub-cores. Instead of managing individual threads, the hardware schedules these warps, (SIMT).

57 Please describe what a stall is and how its effect can be reduced.

A stall occurs when a processor must pause execution because a necessary resource is not yet available.

Its effects can be reduced with multithreading, which hides the latency by switching to another thread while the stalled thread waits.

58 Please describe what a performant parallel program will do to overcome the bandwidth limits.

Performant parallel programs will organize computation to fetch data from memory less often, by

- Reusing data previously loaded by the same thread;
- Sharing data across threads;
- Performing additional arithmetic instead of reloading values (doing math is faster than the memory).

59 Please describe the thread hierarchy in the CUDA abstraction. Is a "warp" part of the CUDA language?

Individual threads are grouped into **thread blocks**, and thread blocks are grouped into a **grid**.

Threads in the same block can communicate via shared memory (needing synchronization, like barriers), while threads from different blocks operate independently and generally should not communicate directly. All threads, however, can read-/write the device global memory.

Warps are not part of CUDA as they are an hardware-specific execution unit (typically 32 threads per warp).

60 Please describe the main characteristics of a message-passing programming model.

Processes are independent, each with its own private local memory (they cannot directly read or write to each other's memory).

All data exchange and synchronization must occur through explicit communication, where one process executes a **send** operation and another executes a **receive** operation

This gives the programmer control over data distribution, making it highly scalable for distributed clusters.

61 Multi-threading is mainly meant for throughput-oriented systems. True/False. Why?

TRUE.

It hides latency by keeping the execution units busy: when one thread stalls, the hardware switches to another ready thread, which improves overall system utilization (throughput) while degrading the performance of individual threads due to resource sharing and context-switching overheads.

62 Please describe the different hardware architectures supporting multi-threading.

- **Fine-grain multithreading**: the processor switches threads after every cycle. If one thread stalls, others are executed immediately. Requires complex hardware to switch extremely fast.
- **Coarse-grain multithreading**: the processor only switches when a long stall occurs. This simplifies the hardware but fails to hide shorter stalls.
- **Simultaneous multithreading**: instructions from multiple threads issue to the execution units in the exact same clock cycle, using all the available functional units at once. This is useful in multiple-issue dynamically scheduled processors.

63 Please describe how a programmer can reduce the penalties due to the memory bandwidth bound.

Programs must access memory infrequently to utilize modern processors efficiently:

- Organize computation to fetch data from memory less often
  - Reuse data previously loaded by the same thread (temporal locality optimizations)
  - Share data across threads (inter-thread cooperation)

- Favor performing additional arithmetic to storing/reloading values (the math is "free")

64 What is meant for the Block/Tiling CUDA code optimization technique?

Tiling improves performance by restructuring code execution to improve data locality. It involves dividing the large input data into smaller blocks (tiles) fit entirely within the processor's high-speed memory (cache or shared memory).

This minimizes expensive accesses to main memory, since each thread has its needed tile of data very "close"

65 Please describe what is meant by GPU "SIMT".

SIMT is an execution model introduced by NVIDIA. The programmer writes code for a single logical thread (scalar code), the GPU hardware then dynamically groups these independent threads into Warps (typically groups of 32 threads) and executes the same instruction on every thread (similar to how SIMD does with the different SIMD units).

It gives the programmer the illusion that each thread is executing independently with its own register state and instruction address.

66 Please describe the key characteristics of the CUDA synchronization constructs.

We have:

- `__syncthreads()`: is a barrier. It forces all threads in a single block to wait until every other thread has reaached the synchronization point.
- Atomic operations (e.g., `atomicAdd`): available for both gloabal memory addresses and per-block shared memory addresses.
- Host/device synchronization: there is an implicit barrier at the return of the kernel, meaning that all threads in the grid have completed their execution before the kernel finishes.

67 Please describe how each sub-core runs the next instruction for the CUDA threads in the warp. What is meant by divergence in this context?

Each sub-core does not fetch its own instruction. Instead, **a single instruction unit maintains one Program Counter** for the entire warp, and each thread

executes the same instruction.

Divergence occurs when threads within this single warp evaluate a conditional branch differently (e.g., half the threads need to run the if block and half the else block). Because the hardware cannot fetch two different instructions at the same time, it serializes the execution: it runs the if path for the relevant threads while masking the others, and then runs the else path with the masks reversed.

68 Why CUDA kernels cannot exploit context switching?

Because of the massive size of the thread state. A GPU manages thousands of simultaneous threads, each holding private variables in a huge, on-chip Register File (often megabytes in size). Saving and restoring this enormous amount of data to memory would saturate the memory bandwidth, giving a prohibitive performance penalty.

Therefore, instead of swapping tasks out, GPUs keep all active contexts resident on the chip and use zero-overhead hardware switching between them to hide latency.

69 Is there a relation between the number of CUDA threads in a warp and the number of registers in a streaming multi-processor (SM) unit? What else influences the number of registers?

Yes, there is a direct relationship. The Streaming Multiprocessor (SM) has a fixed physical pool of registers. Since registers are private to each thread, the cost to host a single warp is $(32 \times RegistersPerThread)$. If this number is too high, the SM cannot fit the maximum number of active warps, forcing it to leave execution slots empty and reduce parallelism.

The number of registers allocated can also be influenced by the compiler's optimization strategy, and explicit limits set by the programmer.

70 On NVIDIA V100 a CUDA grid could be executed on multiple SM units. True/False?

TRUE.

A CUDA Grid is composed of many Thread Blocks. When a grid is launched, the GPU's hardware scheduler distributes these blocks across the many available Streaming Multiprocessors (SMs) on the chip.

71 How are scalar registers for CUDA threads organized in an NVIDIA V100SM "sub-core"?

The Streaming Multiprocessor (SM) is partitioned into four independent processing blocks (sub-cores). Each of these sub-cores contains its own dedicated Register File. This register file consists of scalar registers.

Registers are distributed among active warps. Warps assigned to a specific sub-core can only use the registers present in that sub-core's local file. This partitioning reduces hardware complexity and increases bandwidth for register accesses.

72 Please describe how multi-threading can be implemented at the core level.

Multi-threading is implemented at the core level (Simultaneous Multi-Threading or SMT) by duplicating the architectural state for each thread: Program Counter and Register File. The execution resources (ALUs, caches, and Floating Point Units) remain shared.

If Thread A stalls, the scheduler instantly fills the idle execution slots with instructions from Thread B, maximizing the utilization of the hardware without needing a full operating system context switch.

73 Please explain how the CUDA threads are executed.

CUDA threads are executed using a SIMT (Single Instruction, Multiple Threads) model.

1. Threads are grouped into **Warps** (typically 32 threads).
2. The SM's warp **scheduler** selects a warp that is ready to execute (not stalled) and issues the same instruction to all 32 threads simultaneously.
3. Each thread **executes** this instruction on its own private data. If threads diverge (e.g., due to an `if-else`), the hardware serializes the execution paths.

4 Exercises 4

74 What is the difference between joining and waiting on a barrier in Pthreads?

Joining synchronizes the execution of two threads, the joined thread is destroyed.

A **barrier** synchronizes the execution of a group of threads, all of them continue working after the last one reaches the barrier.

75 Give two examples of OpenMP environment variables and how they influence pro-

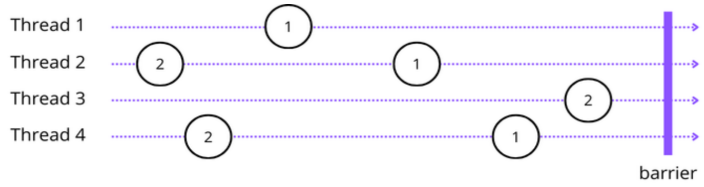
gram execution.

Examples:

- `OMP_NUM_THREADS` to set the number of threads if not specified in the program
- `OMP_NESTED` to enable/disable nested parallelism
- `OMP_CANCELLATION` to enable/disable cancellation

Environment variables influence the execution of any OpenMP program launched in that environment.

76 Describe how each thread in the following diagram behaves at runtime as it encounters points marked with 1, representing pragma omp cancel, and with 2, representing pragma omp cancellation point.



In order:

1. Thread 2 reaches a cancellation point but no thread has requested cancellation, so it continues.
2. Thread 4 reaches a cancellation point but no thread has requested cancellation, so it continues.
3. Thread 1 reaches a cancel directive, sets the cancellation flag, stops executing, waits at the barrier.
4. Thread 2 reaches a cancel directive, sees that the cancellation flag has been already set, stops executing, waits at the barrier.
5. Thread 4 reaches a cancel directive, sees that the cancellation flag has been already set, stops executing, waits at the barrier.
6. Thread 3 reaches a cancellation point, sees that the cancellation has been set, stops executing, waits at the barrier

77 What happens when a variable is declared as firstprivate in an OpenMP pragma?

Each thread creates its own private copy of that variable, which is initialized with the value that the original variable had immediately before the parallel region began.

78 What is the difference between a static and a dynamic schedule in the OpenMP `for` construct?

- `static`: loop iterations are divided into blocks of size `chunk` and then statically assigned to threads. If `chunk` is not specified, the iterations are evenly divided.
- `dynamic`: loop iterations are divided into blocks of size `chunk`, and dynamically scheduled among the threads. When a thread finishes one chunk, it is dynamically assigned to another. The default chunk size is 1.

79 Write a small example of nested parallelism in OpenMP and clearly indicate how many threads execute each region.

```
#pragma omp parallel num_threads(2)
{
    // Outer parallel region
    // executed by 2 threads: master and 1 slave

    // Each outer thread creates a new team of 3 threads
    #pragma omp parallel num_threads(3)
    {
        // Inner parallel region
        // executed by 2*3 = 6 threads
        // each of the 2 outer threads becomes the master of
        // its new team of 3 threads
    }
}
```

80 What happens if a `main()` function that launched multiple Pthreads threads returns after launching `pthread_exit()`.

If the `main` thread calls `pthread_exit()` as its final action (rather than returning or callling `exit()`), the main thread terminates but the process remains alive, allowing the other created threads to "survive" and continue running.

81 Can a programmer request more OpenMP threads than there are processors in a system? How is the program executed in that case?

Yes. In this case, instead of running truly in parallel, the threads take turns executing on the cores with context switching. This degrades performance due to the overhead of switching contexts.

82 Describe the effects of the following pragma:

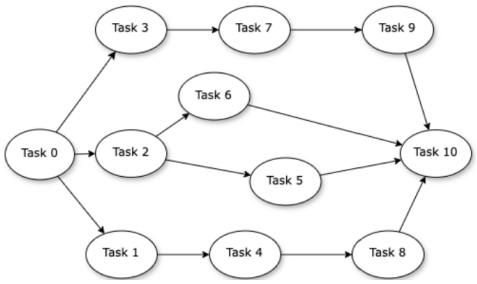
```
#pragma omp task depend(in: status[i])
```

It specifies a unit of work (task) that will be executed with `status[i]` as an input (flow dependency). This means that it will wait for `status[i]` to be written by a previous task before starting execution.

83 Which methods are available in OpenMP to protect access to a shared variable?

- `atomic` directive: useful for single-statement operations (e.g., counters). Means that the operation cannot be interrupted before its completed (e.g., the involved memory address cannot be touched).
- `critical` directive: protects a block of code so that only one thread at a time can execute it.
- **Locks**: OpenMP provides functions to lock a specific section of code and then to unlock it. It is done at runtime (in contrast to `critical`).

84 Consider the following task graph:



W(Task 0) = 1
W(Task 1) = 100
W(Task 2) = 100
W(Task 3) = 100
W(Task 4) = 100
W(Task 5) = 100
W(Task 6) = 100
W(Task 7) = 100
W(Task 8) = 100
W(Task 9) = 100
W(Task 10) = 1

Is it a good idea to implement the program with 4 threads? What is the theoretical improvement in execution time with respect to a sequential version?

Yes. The maximum number of tasks that can be executed simultaneously is exactly 4 (int the phase containing 4, 5, 6, 7).

To calculate the speedup we compare:

- $T_1 = 1 + 900 + 1 = 902$
- $T_4 = \underbrace{1}_{task0} + \underbrace{100}_{\text{tasks 1, 2, 3}} + \underbrace{100}_{\text{tasks 4, 5, 6, 7}} + \underbrace{100}_{\text{tasks 8, 9}} + \underbrace{1}_{\text{task 10}} = 302$

$$\implies SU = \frac{T_1}{T_4} = \frac{902}{302} \approx 3$$

85 Consider the previous task graph. Write a corresponding OpenMP implementation.

```
omp_set_num_threads(4);

#pragma omp parallel
{
    #pragma omp single // only one thread generates the graph
    {
        #pragma omp task depend(out: dep[0])
        task0();

        #pragma omp task depend(in: dep[0]) depend(out: dep[1])
        task1();

        #pragma omp task depend(in: dep[0]) depend(out: dep[2])
        task2();

        #pragma omp task depend(in: dep[0]) depend(out: dep[3])
        task3();

        #pragma omp task depend(in: dep[1]) depend(out: dep[4])
        task4();

        #pragma omp task depend(in: dep[2]) depend(out: dep[5])
        task5();

        #pragma omp task depend(in: dep[2]) depend(out: dep[6])
        task6();

        #pragma omp task depend(in: dep[3]) depend(out: dep[7])
        task7();

        #pragma omp task depend(in: dep[4]) depend(out: dep[8])
        task8();

        #pragma omp task depend(in: dep[7]) depend(out: dep[9])
        task9();

        #pragma omp task depend(in: dep[5], dep[6], dep[8], dep[9])
        task10();
    }
}
```

86 What is a mutex in Pthreads, and how does it work?

Mutex (mutual exclusion) variables are a method to protect shared data when multiple writes occur.

Mutexes serialize data accesses. Only one thread can lock a mutex variable at any given time. If multiple threads try to lock a mutex, only one will be successful. Threads that could not acquire the mutex are blocked.

Example:

```
pthread_mutex_lock(&my_lock);
/* critical section */
pthread_mutex_unlock(&my_lock);
```

87 In OpenMP, tasks are better than sections for irregular algorithms. Is this statement true or false? Why?

TRUE.

Tasks allow for dynamic run-time generation of work units, while **sections** require the parallel structure to be fixed at compile time.

Since irregular problems generate unpredictable workloads that cannot be evenly mapped to static sections, tasks can help to keep all threads active even for these kinds of problems.

88 The following code snippet is supposed to always run Task 3 after Tasks 1 and 2. Why does it sometimes not happen?

```
#pragma omp parallel {
    #pragma omp single {
        #pragma omp task { Task1(); }
        #pragma omp task { Task2(); }
        Task3();
    }
}
```

When the thread executing the `single` block encounters the first `task`, it creates the task and places it in a queue to be executed (potentially later by a different thread). It does not wait for the task to finish. Same with the second `task`.

Finally, it moves to `Task3()`, which is a standard function call that runs immediatly (meanwhile Tasks 1 and 2 **might** still be in the queue).

The fix is to put `#pragma omp taskwait` *(synchronization barrier) before* `Task3()`*. Alternatively, use task dependencies with the* `depend` *clause, or put* `Task3()` *outside of the* `parallel` *block (implicit barrier).*

89 Please briefly describe when coalesced memory access happened.

Coalesced memory access happens when threads within a Warp simultaneously execute a memory instruction that requests memory addresses from the same section. In this case, the GPU memory controller combines those accesses and makes only one DRAM request, maximizing global memory bandwidth.

90 Please explain the conditions under which a memory system is defined as "consistent". Which of the conditions a **sequentially consistent memory system** is relaxing?

There are 4 types of conditions that a consistent memory system has to respect:

- $W_{\texttt{X}} \rightarrow R_{\texttt{Y}}$: write to `X` must commit before subsequent read from `Y`.
- $R_{\texttt{X}} \rightarrow R_{\texttt{Y}}$: read from `X` must commit before subsequent read from `Y`.
- $R_{\texttt{X}} \rightarrow W_{\texttt{Y}}$: read from `X` must commit before subsequent write to `Y`.
- $W_{\texttt{X}} \rightarrow W_{\texttt{Y}}$: write to `X` must commit before subsequent write to `Y`.

A sequentially consistent memory system **maintains all 4** memory operations ordering (not relaxing any condition).

91 Please briefly describe what DRAM Banking is meant for.

In DRAM, each bit of data is not accessed individually. Instead, you have to access an entire row of data from the core array, and then select the column (the specific chunk of data). This process is slow: **banking** is used to improve DRAM access performance. We have multiple core arrays (banks), each with its own row decoder and sense amps.

92 What do the terms "custom computing" and "heterogeneous systems" mean?

- Custom computing is the practice of designing hardware specifically to match the requirements of a particular application to achieve better performance and energy efficiency compared to general-purpose processors;
- Heterogeneous systems integrate multiple types of processing units into a single system or chip, allowing different workloads to be handled to the execution unit best suited for them.

93 Please briefly describe when uncoalesced memory access happen.

Uncoalesced memory access happens when the parallel threads within a single warp attempt to simultaneously access (load or store) misaligned (e.g., non-contiguous) memory addresses that cannot be grouped into a single hardware transaction. Instead of fetching one efficient chunk for the whole warp, the hardware is forced to issue multiple transactions, wasting bandwidth.

94 Please briefly describe how DRAM Bursting Timing is.

Bursting is a technique where a single read or write command initiates a rapid sequence of multiple data transfers (burst) from adjacent memory locations (so the data bus remains fully utilized).

Let's say the SDRAM cores run at just $1/N$ speed of the interface. We load ($N \times$ InterfaceWidth) bits from the same row at once to an internal buffer. This buffered data is then transferred sequentially in N steps at the higher interface speed (e.g., DDR3/GDDR4 use a buffer width = $8\times$ interface width).

95 Please briefly describe with an example how the "Corner turning" technique works. Why is improving the kernel performance and in which case?

An example is **matrix multiplication** where the algorithm logically requires reading the second matrix column-by-column (vertically), while the matrix in global memory is stored "by rows" (horizontally). The hardware must fetch a huge chuck of row data just to read a single value. **Corner turning** loads a "tile" of data into the fast Shared Memory, the threads can then read that data vertically much faster and perform matrix multiplication with values from shared memory.

96 DRAM Banking could reduce the dead time. True/False? Why?

TRUE.

While one bank is busy performing a slow operations like row activation, the controller can immediately switch to accessing an active row in a different bank. This keeps the high-speed data bus continuously saturated.

97 We say that coalesced access happens if the index in an array is in the form of . . . (fill the dots)

`ThreadIdx.x` (the thread identifier within the warp).

This means that Thread 0 reads address 0, Thread 1 reads 1, and so on. This

pattern allows the hardware to fetch all required data in a single continuous memory transaction.

98 Please define what is meant by "Memory consistency".

Memory Consistency refers to the defined rules of the ordering of memory updates between parallel threads. It answers the question: "When does a value written by Thread A become visible to Thread B?".

99 Provide an example of CUDA coalesced accesses.

Sequential read: `float value = input[ThreadIdx.x]`

Thread 0 reads address 0, thread 1 reads address 1, and so on. Since the accesses are sequential, the hardware is able to merge all contiguous request into one memory transaction.

100 Please explain what a Total Store Ordering consistency model does.

Total Store Ordering (TSO) relaxes the rule $W_X \rightarrow R_Y$. A processor P can read Y before its write to X is seen by all processors (a processor can move its own reads in front of its own writes).

Obviously, reads by other processors cannot return the new value of X until the write to X is observed by all processors

101 In Heterogeneous processing moving less data may give better power consumption. True/False? When?

TRUE.

Moving data across the physical wires of a chip consumes significantly energy than the actual mathematical computation. In heterogeneous systems, data transfers are avoided by chaining multiple kernels together on the device so that intermediate data stays in the local memory, or by using techniques like Near-Memory Processing, where the calculation happens right next to where the data lives.

102 Is the time required to access the memory different from the time taken to access the main memory in a PRAM model?

No. PRAM is a theoretical abstraction that assumes Unit Time Access $O(1)$ for accessing any memory location in the shared global memory.

103 Is it tiling related to the GPUs' shared memory? Please explain why and when.

Yes. Since the global memory is slow, we use **tiling**, a technique where you break a large dataset into small blocks ("tiles") that fit perfectly into the fast shared memory. You use tiling when an algorithm has high data reuse (like Matrix Multiplication).

Instead of every thread repeatedly fetching the same data from slow Global Memory, the threads collaborate to load a "tile" of data into Shared Memory once, and then everyone reuses that data multiple times from this high-speed cache.

104 Please define what memory coherency means.

Memory **Coherency** ensures that all processors see the same data for a specific memory address, even when that data is cached in multiple local caches. In other words, Coherency guarantees that all caches eventually agree on what the value is.

105 Why custom computing and heterogeneous systems are relevant today?

Since we can no longer simply increase CPU clock speeds without overheating, the industry has shifted to specialized hardware. By combining a CPU with e.g. GPUs, heterogeneous systems perform specific heavy tasks more efficiently than a general-purpose processor could. This allows performance to continue growing by optimizing performance per watt rather than raw frequency.