PARALLEL COMPUTING exam questions PT2

Claudio Tessa

January 29, 2026

Contents

1 Exercises 1

1 Please describe how the Pack parallel pattern works. How is it implemented (pseudo code and PRAM complexity required)?

Used to eliminate unused elements from a collection:

1. Convert input array of Booleans into integer 0s and 1s;
2. Perform an exclusive scan of this array with the SUM operation;
3. Write values in the output array based on the offset

```
if(InitialMask[i])
    out[ExScanOf1[i]] = input[i];
```

If executed on N execution units, step 1 and 2 have a complexity of $O(1)$, while step 2 has a time complexity of $O(\log N)$, therefore the total time complexity is $O(\log N)$. Work is $O(N)$.

2 Consider the following nested loop:

```
for (j = 0; j < 999; j++) {
    for (i = 2; i < 999; i++) {
        a[i][j] = f(a[i-2][j]);
    }
}
```

Evaluate the cache-friendliness of this code and consider its relationship to parallelizability. How does the memory layout of the array affect performance, and what changes could improve cache utilization vs parallelizability? Justify your answer. Assume that `a` stores integers.

Original code is fully parallelizable with respect to loop `j` if function `f` is safe and it is partially parallelizable with respect to loop `i`.

The original code is poor with respect to cache friendliness because the variable `a` is accessed column-wise. Swapping the two loops fixes cache friendliness. Tiling could be another approach to improve cache friendliness and parallel computation

3 Please describe the parallel gather pattern, including assumptions, complexity, and examples.

Given a collection of ordered indices, read data from the source collection at each index, and write data to the output collection in index order. It can be seen as the inverse of scatter.

Typically assumes a CREW PRAM (Concurrent Read, Exclusive Write) model to allow multiple threads to read from the same source location simultaneously.

Time complexity is $O(1)$ since all reads happen in parallel. Work is $O(N)$, one operation per element.

An **example** is rotating an image: the new target pixel need to *gather* the color from the source pixel.

4 Please describe how the Split parallel pattern works. How is it implemented (pseudo code required)?

Split rearranges data so that all elements of the input are moved to either the lower or upper part of the output collection, given on a boolean collection that tells in which of the two portions an element will go. Relative ordering withing the lower/upper half is kept.

```
LastOf0 = ExScanOf1[last] + 1;
if(InitialMask[i])
    out[ExScanOf1[i] + LastOf0] = DataInput[i];
else
    out[ExScanOf0[i]] = DataInput[i];
```

2 Exercises 2

5 What is the impact of memory coalescing on DRAM access efficiency in CUDA?

When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section (coalesced access), only one DRAM request will be made and the access is fully coalesced.

6 What is the purpose of using shared memory in tiled matrix multiplication? Is the corner turning technique relevant for this purpose?

Reading A and B in a matrix multiplication results in A read in non-coalesced way and B read in a coalesced way. To improve performance, we copy the data into shared memory, allowing both input matrices to be accessed coalesced in both cases (corner turning).

7 Please describe how priority scatter works.

Given a collection of input data, and a collection of location, **scatter** copies the input data in the output data, at the location specified by the locations collection.

The location array could map two or more input elements to the same location. In this case, priority scatter solves this write conflict by allowing only the processor with highest priority to write.

8 How can a CUDA programmer judge if an access is coalesced?

By checking if consecutive threads in a warp are accessing consecutive memory addresses. Mathematically, the access is coalesced if the memory accesses are in the form of `Mem[Base + ThreadIdx.x]`. This creates a sequential pattern that allows the memory controller to merge the individual requests into a single transaction.

If the threads access memory e.g. randomly, the hardware must split the operation into multiple separate transactions, signaling uncoalesced access.

9 Describe the corner turning technique. How the size of the shared memory is defined in this technique?

See corner turning technique definition in a previous question

The allocated size is primarily defined by the tile dimensions (e.g., $TILE\_WIDTH \times TILE\_WIDTH$). However, in **Corner Turning**, the size is frequently defined as

$$TILE\_WIDTH \times (TILE\_WIDTH + 1)$$

This extra column of padding shifts the data layout in shared memory to ensure that when threads read down a column, they access different memory banks, preventing Bank Conflicts.

3 Exercises 3

10 Efficient parallel programming with CUDA requires knowledge about GPU architecture. Discuss this claim by providing concrete examples.

Yes because performance gains depend on aligning code with physical hardware constraints rather than just logical correctness.

For example, a developer must understand the SIMT (Single Instruction, Multiple Threads) execution model to avoid warp divergence, where conditional branches (`if-else`) force the hardware to serialize execution paths, leaving ALUs idle.

Additionally, maximizing memory throughput requires knowledge of memory coalescing, structuring data access so that threads read contiguous addresses, allowing the memory controller to merge individual requests into a single transaction.

11 What is Unified Memory in CUDA? When is it used?

Unified Memory is an abstraction that lets the programmer treat the host and device memories as a single address space. Data transfers between host and device happen at runtime. It is used to simplify the programming (so that data does not need to be moved explicitly between host and device) and it might be useful for sparse/irregular workloads or in multi-GPU systems.

12 Given the following CUDA kernel, discuss the impact of control divergence for different `N` values.

```
__global__ void myKernel(int *data, int N) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        if (idx % 2 == 0) {
            data[idx] *= 2;
        } else {
            data[idx] += 1;
        }
    }
}
```

The impact of control divergence in this kernel is severe regardless of `N`. Since CUDA threads are grouped into warps of 32 consecutive indices, the condition `idx % 2 == 0` creates an alternating pattern. Consequently, the GPU hardware must serialize the execution, halving the throughput for the entire kernel execution.

The value of `N` only affects the boundary check (`idx < N`). Ideally, `N` should be greater than the warp size, but the divergence explained above dominates the performance penalty.

4 Exercises 4

13 Provide an example (in pseudocode, with explanations) of a batch processing pattern where OpenMP and CUDA are used together to distribute work among multiple GPUs.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
std::vector<cudaStream_t> streams(deviceCount);

#pragma omp parallel for num_threads(deviceCount)
for(int dev = 0; dev < deviceCount; ++dev) {
    // for each GPU in the system:
    cudaSetDevice(dev);
    cudaStreamCreate(&streams[i]); // use a stream for data transfers
    // Allocate, initialize and transfer memory
}

#pragma omp parallel for num_threads(deviceCount)
for(int dev = 0; dev < deviceCount; ++dev) {
    cudaSetDevice(dev);
    kernel<<<gridDim, blockDim, streams[i]>>>(...);
}
```

14 Provide an example (in pseudocode, with explanations) of an application that uses OpenMP instead of CUDA to offload computation on a GPU.

```
// Offload the following region to a GPU,
// transfer contents of a,b,c,d
#pragma omp target map(a, b, c, d) {
    // use multiple GPU threads to parallelize this loop
    #pragma parallel for
    for (i = 0; i < N; i++) {
        a[i] = b[i] * c + d;
    }
}
```

15 Describe the trade-offs that can be explored in the design space created by the Halide scheduling directives.

The Halide scheduling directives navigates the trade-off between **locality**, **parallelism**, and **redundant computation** without altering the algorithm's correctness.

Halide solves these trade-offs by decoupling the algorithm (what to compute) from the schedule (how to execute it). Allows you to write the mathematical definition once and then use scheduling directives to manipulate the execution strategy instantly:

- Re-compute values on the fly to save memory bandwidth; or

- compute them once and store them, which avoids redundant math but increases memory traffic; or
- The "sweet spot" where you compute small tiles of data just before they are needed by the consumer loop, fitting them into the cache.

16 Provide at least two examples of situations where a combination of different parallel programming languages is used.

A common approach is MPI + OpenMP, used in supercomputing clusters where MPI manages communication between distinct computing nodes while OpenMP parallelizes the workload within each node.

Another example is OpenMP + CUDA, used in heterogeneous systems where the CPU uses OpenMP to manage threads and data distribution, while offloading parallel compute-intensive kernels to the GPU via CUDA.