MCQ

1. For a vector addition, assume that the vector length is 2,000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?
   a. 2,000
   b. 2,024
   c. 2,048
   d. 2,096
2. You are working with an image of size 512x1024 pixels. You want to process each pixel with a CUDA thread. If you use a block size of 16x16 threads, how many blocks are needed in the x-direction and y-direction?
   A) 32 blocks in x-direction and 64 blocks in y-direction
   B) 64 blocks in x-direction and 32 blocks in y-direction
   C) 32 blocks in x-direction and 32 blocks in y-direction
   D) 64 blocks in x-direction and 64 blocks in y-direction
3. In a CUDA program, you have a 1D grid of 10 blocks with each block containing 32 threads. If a specific thread within block 7 (0-based index) has a thread index of 5, what is its global thread ID?
   A) 237
   B) 235
   C) 229
   D) 239
4. What is the main difference between `__global__` and `__device__` functions in CUDA?

   A) `__global__` functions can be called from the host and execute on the device, while `__device__` functions are called only from other device functions.

   B) `__global__` functions execute on the host, while `__device__` functions execute on the device.

   C) `__global__` functions are optimized for device memory access, while `__device__` functions are optimized for host memory access.

   D) `__global__` functions can only be called from the device, while `__device__` functions can be called from both host and device.

5. The function cudaMemcpy() is used to copy data
   a) From one location of device memory to another location of device memory
   b) Between different GPUs in multi-GPU systems
   c) From one location of device memory to another location of host memory
   d) From one location of host memory to another location of device memory
   **a, c and d only
   c and d only
   b, c and d only
   All of the above

4 marks

6. Write a program in MPI to get the following output.

Input string read by root: MIT TOP HUT

I am rank 0, my string is: TIM

I am rank 1, my string is: POT POT POT

I am rank 2, my stirng is: TUH TUH TUH TUH TUH TUH

Output string displayed by root: TIM POT POT POT TUH TUH TUH TUH TUH TUH

The root will distribute words in the string to the processes including itself and collects to print the final output. Each process is involved in reversing the word/partial string received and repeating the input word equal to (word length * rank) times. If its rank 0, at least 1 time the reversed string needs to be printed. Use collective communication to distribute the words from the input string and use point to point communication to send the repeated string to root process. Each process should print its computed string as well.

Solution:

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_STRING_LEN 100
// Function to reverse a string
void reverse_str(char *str) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        char temp = str[i];
        str[i] = str[len - 1 - i];
        str[len - 1 - i] = temp;
    }
}

int main(int argc, char **argv) {
    int rank, size;
    char input_str[MAX_STRING_LEN] = "MIT TOP HUT";  // Input string at the root
    char words[3][MAX_STRING_LEN] = {"MIT", "TOP", "HUT"};  // Manually split words
    char recv_word[MAX_STRING_LEN];  // Buffer for each process to receive a word
    MPI_Status status;

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Root process manually divides the input string into words (Scatter preparation)
    if (rank == 0) {
        printf("Input string read by root: %s\n", input_str);
    }

    // Scatter the words to each process using collective communication (MPI_Scatter)
    MPI_Scatter(words, MAX_STRING_LEN, MPI_CHAR, recv_word, MAX_STRING_LEN, MPI_CHAR, 0, MPI_COMM_WORLD);

    // Reverse the word received by each process
    reverse_str(recv_word);

    // Repeat the reversed word (rank * word length) times
    char final_str[MAX_STRING_LEN * 10] = "";
    int word_len = strlen(recv_word);
    int repeat_count = rank * word_len;

    // If rank == 0, at least one repetition is required
    if (rank == 0) {
        repeat_count = 1;
    }
    // Concatenate the reversed word "repeat_count" times
    for (int i = 0; i < repeat_count; i++) {
        strcat(final_str, recv_word);
        if (i < repeat_count - 1) {
            strcat(final_str, " ");
        }
    }

    // Each process prints its string
    printf("I am rank %d, my string is: %s\n", rank, final_str);

    // Now, each process sends its result back to the root process using point-to-point communication
    if (rank != 0) {
        MPI_Send(final_str, MAX_STRING_LEN * 10, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        // Root process collects the results
        char result[MAX_STRING_LEN * 10] = "";  // Store the final result
        strcpy(result, final_str);  // Root's own result

        for (int i = 1; i < size; i++) {
            char temp[MAX_STRING_LEN * 10];
            MPI_Recv(temp, MAX_STRING_LEN * 10, MPI_CHAR, i, 0, MPI_COMM_WORLD, &status);
            strcat(result, " ");
            strcat(result, temp);
        }
        // Root process prints the final combined result
        printf("Output string displayed by root: %s\n", result);
    }
    // Finalize MPI
    MPI_Finalize();
    return 0;
}
```

Explanation and Evaluation Scheme (4 Marks):

1. **Setting up MPI environment:** **1 mark**
2. **Collective Communication (Scatter)**:
   - The MPI_Scatter function is used to **distribute the words** to each process. The root process (rank 0) sends the words "MIT", "TOP", and "HUT" to different processes. **1 mark**
3. **String Manipulation**:
   - After receiving the word, each process **reverses the word** and repeats it according to its rank. The reversed string is then constructed by repeating it based on the (rank * word length). **1 mark**
4. **Point-to-Point Communication**:
   - **Each process sends its processed string** back to the root process using MPI_Send.
   - The root process **receives the strings** from all other processes using MPI_Recv and concatenates them into a single result. **0.5 mark**
5. **Root Process Output**:
   - After collecting all the strings, the root process displays the final output string, which consists of all the reversed and repeated words from the other processes. **0.5 mark**

7. Apply OpenMP to parallelize a program that sums the elements of an array without using the reduction clause. How would you ensure the correctness of the final result?

Solution:

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

int main() {
   int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int n = 10;
   int sum = 0;

   #pragma omp parallel
   {
      int partial_sum = 0;  // Each thread has its own partial sum
      #pragma omp for
      for (int i = 0; i < n; i++) {
         partial_sum += array[i];
      }
      // Use critical section to update the final sum
      #pragma omp critical
```

```
        {
          sum += partial_sum;
        }
      }

      printf("Sum of array elements: %d\n", sum);
      return 0;
    }
```

Explanation:

- Each thread calculates a partial sum of the array elements.
- The partial sums are accumulated into the global sum inside a critical section to prevent race conditions, ensuring that only one thread modifies the global sum at a time.

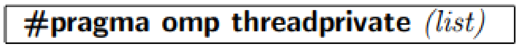
Evaluation Scheme (3 Marks):

- Correct application of OpenMP parallelism (1 Mark):
  - ➢ Parallel region is correctly created using #pragma omp parallel.
  - ➢ Each thread calculates its own partial sum inside a parallel region.
- Correct use of work-sharing constructs (1 Mark):
  - Use of #pragma omp for to distribute the loop iterations across threads.
- Ensuring correctness (1 Mark):
  - Correct use of the #pragma omp critical directive to safely accumulate the partial sums, preventing race conditions and ensuring the correctness of the final result.

8. Explain the significance of the threadprivate and copyprivate clauses in OpenMP. How do they affect the scope of variables in a parallel region?

Solution:

- By default, global data (static in c) is shared. This means that if one thread modifies a static variable, other threads will see the modified value.
- But in some situations we may need, or would prefer to have, private data that persists throughout the computation.
- This can be achieved using the **threadprivate** directive in OpenMP. Each thread gets a private or "local" copy of the specified global variables.



#pragma omp threadprivate *(list)*

- The **copyprivate** clause is used to broadcast the value of a private variable from one thread to all other threads in the team.
- It is supported on the single directive only. It is used in conjunction with the single directive to ensure that one thread initializes the data, which is then copied to other threads.
- After the single construct has ended, the values of variables specified in the copyprivate list are copied to other threads before they leave the associated barrier. This means all threads will synchronize at this point.

Evaluation Scheme (3 Marks):

- **Explanation of threadprivate (1 Mark):**
  - Correctly explaining that threadprivate makes global variables private to each thread and maintains the variable's value across parallel regions.

- **Explanation of copyprivate (1 Mark):**
  - Correctly explaining that copyprivate is used to broadcast a variable's value from one thread to all other threads and specifying that is supported in single directive only.
- **Effect on Variable Scope (1 Mark):**
  - Correct explanation of how threadprivate and copyprivate affect the scope of variables in parallel regions, with threadprivate maintaining thread-specific copies and copyprivate synchronizing values across threads.

9. Explain any three OpenMP work-sharing constructs. Provide an example of each and analyze the differences in how the work is shared among threads.

Solution:

#pragma omp for

This directive is used to **distribute loop iterations** among the threads. Each thread gets a portion of the loop iterations to work on. By default, OpenMP divides the iterations statically among the threads, but dynamic scheduling is also possible.

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

int main() {
   int i, n = 10;
   int a[10];

   #pragma omp parallel for
   for (i = 0; i < n; i++) {
      a[i] = i * i;
      printf("Thread %d is processing iteration %d\n", omp_get_thread_num(), i);
   }

   return 0;
}
```

Analysis:

In the above example, each thread gets a chunk of the iterations to execute.

OpenMP dynamically or statically distributes the loop iterations. The default scheduling is static, where the iterations are divided evenly, but dynamic scheduling can also be specified using schedule(dynamic) to give threads more flexibility.

#pragma omp sections

The sections directive is used to divide work into separate **independent sections**. Each section is executed by a different thread. This is useful when different tasks, not necessarily related, need to be performed in parallel.

Example:

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

int main() {
   #pragma omp parallel
```

```
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                printf("Thread %d is executing section 1\n", omp_get_thread_num());
            }
            #pragma omp section
            {
                printf("Thread %d is executing section 2\n", omp_get_thread_num());
            }
            #pragma omp section
            {
                printf("Thread %d is executing section 3\n", omp_get_thread_num());
            }
        }
    }
    return 0;
}
```

Analysis:

- Each section (#pragma omp section) is executed by a different thread.
- The work within sections does not need to be divided evenly because different sections may involve different tasks. Each section runs independently, and once a thread finishes a section, it doesn't participate in another unless dynamic scheduling is used.

#pragma omp single

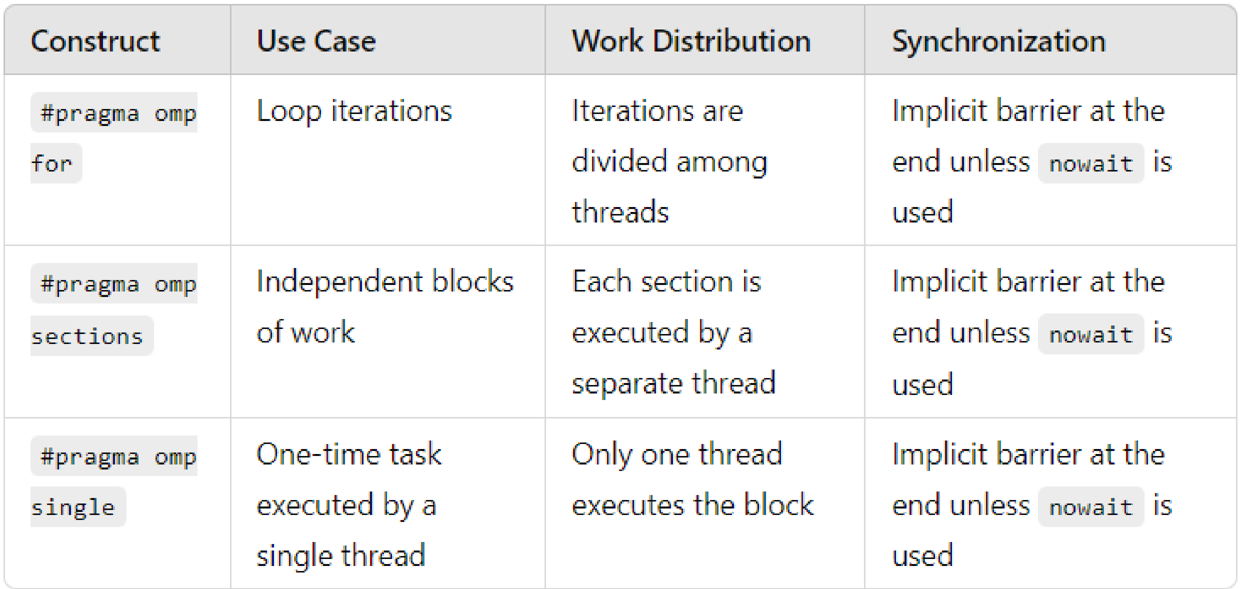
The single construct ensures that a block of code is executed by **only one thread**, while the rest of the threads in the team skip that block. This is useful when you have tasks that only need to be done once in a parallel region.

Example:

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

int main() {
    #pragma omp parallel
    {
        printf("Thread %d is doing parallel work\n", omp_get_thread_num());

        #pragma omp single
        {
            printf("Thread %d is executing the single section\n", omp_get_thread_num());
        }
    }

    return 0;
}
```

Analysis:

- Only one thread executes the block within the single directive.
- Other threads wait until this block is executed unless the nowait clause is used to allow them to proceed without waiting.



Construct	Use Case	Work Distribution	Synchronization
`#pragma omp for`	Loop iterations	Iterations are divided among threads	Implicit barrier at the end unless `nowait` is used
`#pragma omp sections`	Independent blocks of work	Each section is executed by a separate thread	Implicit barrier at the end unless `nowait` is used
`#pragma omp single`	One-time task executed by a single thread	Only one thread executes the block	Implicit barrier at the end unless `nowait` is used

Evaluation Scheme (3 Marks): Each work sharing construct is for 1 marks

10. Explain how the schedule clause in OpenMP controls the assignment of iterations in a loop. Compare the static and dynamic schedules, and explain which one would perform better in a load-imbalanced scenario.

Solution:

The schedule Clause in OpenMP:

- The schedule clause in OpenMP allows the programmer to control how loop iterations are divided among threads. It affects how the workload is distributed across threads in parallel loops.
- Syntax: #pragma omp for schedule(type[, chunk])
- The type can be static, dynamic, or guided, and chunk specifies the number of iterations a thread will take in one go.
- **Static Schedule:**
  - ➢ **Iterations are divided into equal sized chunks of size chunk size, and each chunk is assigned to a thread** before the loop starts. (iterations are evenly divided among threads.)
  - ➢ The **chunks are assigned to the threads statically in a round-robin manner, in the order of the thread number.**
  - ➢ This approach is most effective when the **workload is evenly distributed** and predictable.
  - ➢ **Best for loops where each iteration takes roughly the same amount of time**.
  - ➢ **When no chunk size is specified, the iteration space is divided into chunks that are approximately equal in size.**
  - ➢ In this approach, iterations are divided equally among the threads before they start working. Each thread knew exactly which iterations they had to complete.
- **Dynamic Schedule:**

- ➢ In the dynamic schedule, **iterations are assigned to threads as they finish their previous assignments.**
- ➢ The **thread executes the chunk of iterations** (controlled through the chunk size parameter), then requests another chunk until there are no more chunks to work on.
- ➢ The last chunk may have fewer iterations than chunk size. **When no chunk size is specified, it defaults to 1.**
- ➢ Dynamic scheduling is **beneficial in load-imbalanced scenarios** where iterations take varying amounts of time, as it helps balance the workload by dynamically reallocating work to threads that finish earlier.

- **Which Schedule Performs Better in a Load-Imbalanced Scenario?**
  - Dynamic scheduling performs better in load-imbalanced scenarios because it reallocates work as threads finish, ensuring that no thread remains idle for long periods of time. In contrast, static scheduling can lead to idle threads if some iterations take longer than others.

Evaluation Scheme (3 Marks):

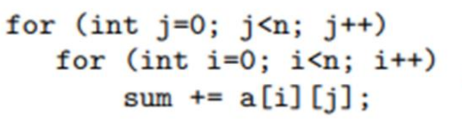
- **Explanation of the schedule clause (0.5 Mark):**
  - ➢ Correctly explaining that the schedule clause controls how loop iterations are divided among threads in a parallel region.
  - ➢ Mentioning the syntax and describing its parameters (static, dynamic, chunk).
- **Comparison of Static and Dynamic Schedules (1+1 Marks):**
  - ➢ Correctly explaining that in the static schedule, iterations are divided evenly among threads before the loop starts.
  - ➢ Correctly explaining that in the dynamic schedule, iterations are assigned to threads as they finish their previous work.
- **Performance in Load-Imbalanced Scenarios (0.5 Mark):**
  - ➢ Correctly identifying that dynamic scheduling performs better in load-imbalanced scenarios and explaining why (dynamic reallocation of work to threads that finish earlier to avoid idle time).

11. Apply loop interchange and unrolling to a nested loop in OpenMP. With an example explain how loop interchange optimizes memory access and improves cache locality.

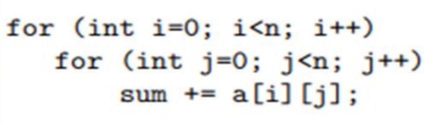
Solution:

- **Loop Interchange** is a technique where you **swap the order of nested loops in a program**. This can help improve the program's performance by making **better use of the memory hierarchy, particularly the cache.**
- The order in which loops are executed can greatly affect how data is accessed in memory. **By changing the loop order**, you can make the program access memory in a more efficient way, **reducing cache misses and improving overall performance**.
- This strategy is called **loop interchange (or loop exchange**).
- A programmer should consider transforming a loop
  If accesses to arrays in the loop nest do not occur in the order in which they are stored in memory.
  If a loop has a large body (many operations) and the references to an array element or its neighbors are far apart, this can lead to data being evicted from the cache before it's

reused, which is inefficient. (In a large loop where an array element is accessed, and then the loop does many other operations before returning to that element, the data might be evicted from the cache, forcing another slow memory access.)

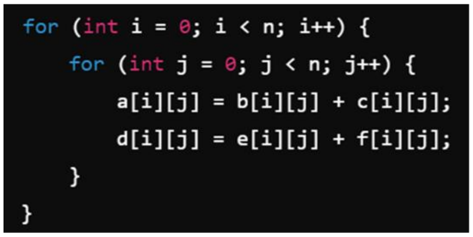


```
for (int j=0; j<n; j++)
   for (int i=0; i<n; i++)
      sum += a[i][j];
```

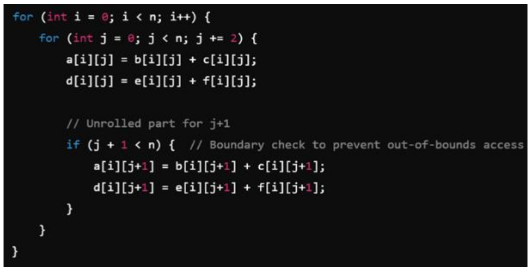


```
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      sum += a[i][j];
```

- **Loop unrolling** (Since the question specifically asks **for loop unrolling in a nested loop, the appropriate technique would be unroll and jam.)**
  - It is a technique where the **iterations of a loop are expanded or "unrolled" so that multiple iterations are performed within a single loop cycle**.
  - Unroll and jam is an extension of loop unrolling that is appropriate for some loop nests with multiple loops.
  - It is a loop transformation technique that **involves unrolling the iterations of an inner loop and then jamming or merging the resulting code into the outer loop.**
  - **This reduces the overhead associated with loop control** (like updating the loop counter and checking the exit condition) and can lead to more efficient execution.
  - Loop unrolling can help to improve cache line utilization by improving data reuse. Since multiple operations are performed on data loaded into the cache, the chances of a cache miss are reduced.
  - It can also help to increase the instruction-level parallelism: more instructions can be executed in parallel, which is especially beneficial on modern CPUs that support multiple execution units.



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



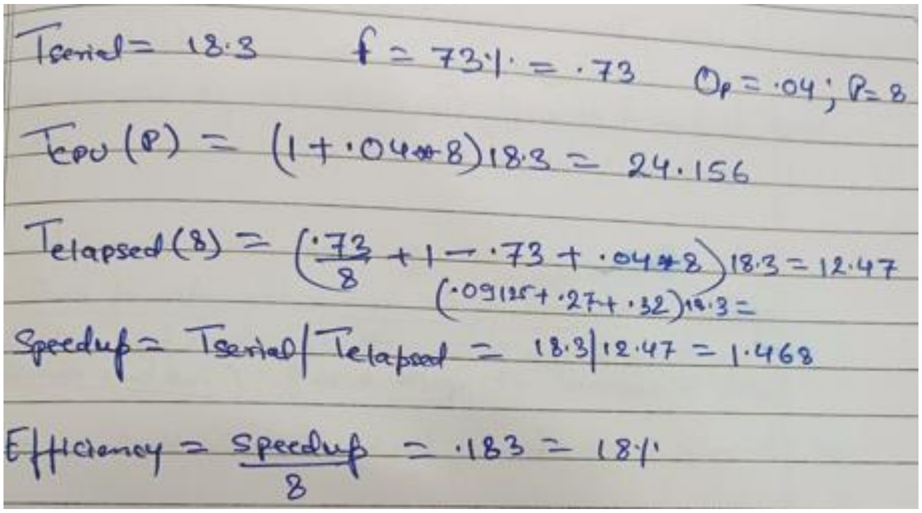
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j += 2) {
        a[i][j] = b[i][j] + c[i][j];
        d[i][j] = e[i][j] + f[i][j];

        // Unrolled part for j+1
        if (j + 1 < n) {  // Boundary check to prevent out-of-bounds access
            a[i][j+1] = b[i][j+1] + c[i][j+1];
            d[i][j+1] = e[i][j+1] + f[i][j+1];
        }
    }
}
```

Evaluation Scheme (3 Marks):

- **Explanation of Loop Interchange (1 Mark):**
  - **0.5 Marks** for defining loop interchange and explaining how it optimizes memory access.
  - **0.5 Marks** for describing how it improves cache locality by aligning memory access with storage format.
- **Explanation of Unroll and Jam (1 Mark):**
  - **0.5 Marks** for defining unroll and jam and explaining how it combines unrolling and jamming to reduce loop overhead.
  - **0.5 Marks** for describing how it increases data reuse and improves performance.
- **Example and Optimization Explanation (1 Mark):**
  - **0.5 Marks** for providing a valid example of loop interchange and unroll and jam applied to a nested loop.
  - **0.5 Marks** for explaining how these transformations improve memory access patterns, cache locality, and performance.

2 marks

12. If the original program takes T_{Serial} = 18.3 seconds to run and code corresponding to 73% of the execution time has been parallelized. Assume that each additional processor adds a 4% overhead to the total CPU time. Compute Speedup and Efficiency of the parallel program with 8 processors. Also estimate estimate the T_{CPU} and $T_{Elapsed}$ of the given program.



$T_{serial} = 18.3 \quad f = 73\% = .73 \quad O_p = .04; P = 8$

$T_{CPU}(P) = (1 + .04*8)18.3 = 24.156$

$T_{elapsed}(8) = \left(\frac{.73}{8} + 1 - .73 + .04*8\right)18.3 = 12.47$

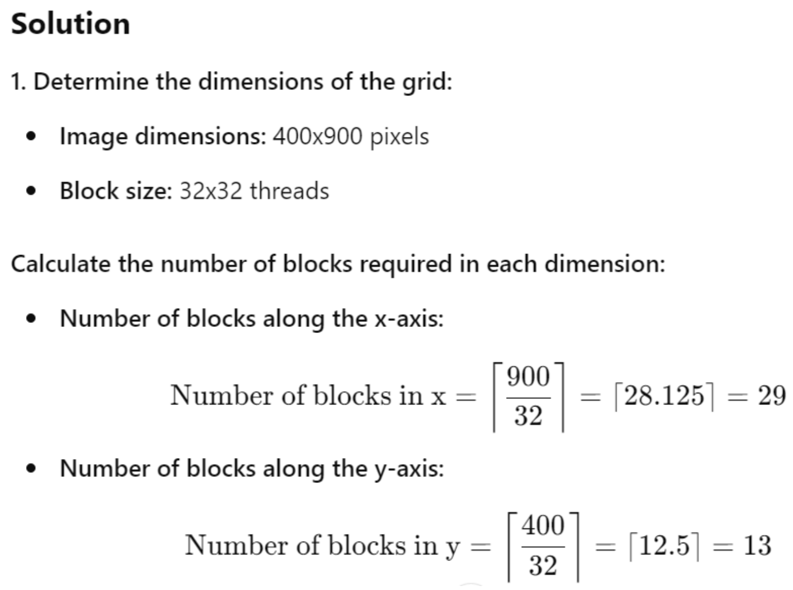
$(.09125 + .27 + .32)18.3 =$

$Speedup = T_{serial} / T_{elapsed} = 18.3 / 12.47 = 1.468$

$Efficiency = \frac{Speedup}{8} = .183 = 18\%$

Each parameter is for 0.5 marks

13. You are working with an image of size **400x900 pixels**, where each pixel is processed by a single CUDA thread. You would like your thread blocks to be square of size 32x32. How would you choose the grid dimensions to cover the entire image? In one case, consider the thread located in the second block along the x-direction and the third block along the y-direction. Within this block, the thread is positioned at the third location along the x-axis and the fourth location along the y-axis. Using this setup, calculate the global thread ID for the specified thread.



Solution

1. Determine the dimensions of the grid:

- **Image dimensions:** 400x900 pixels
- **Block size:** 32x32 threads

Calculate the number of blocks required in each dimension:

- Number of blocks along the x-axis:

$$\text{Number of blocks in x} = \left\lceil \frac{900}{32} \right\rceil = \lceil 28.125 \rceil = 29$$

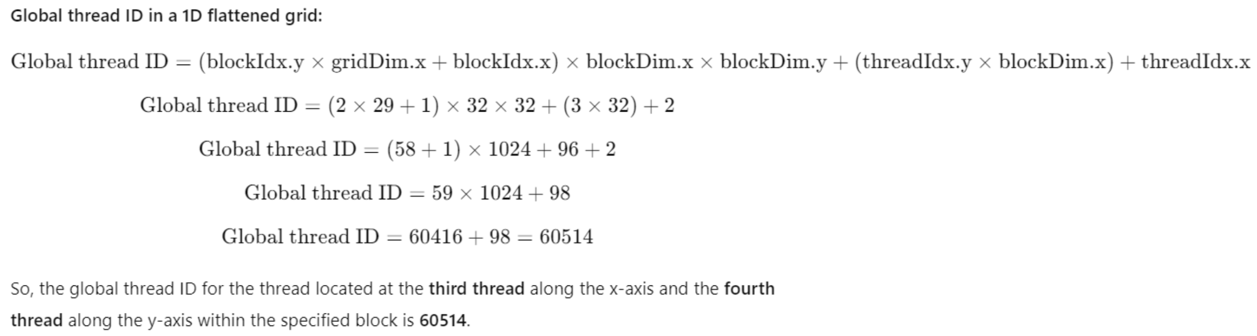
- Number of blocks along the y-axis:

$$\text{Number of blocks in y} = \left\lceil \frac{400}{32} \right\rceil = \lceil 12.5 \rceil = 13$$

So, the grid dimensions are **29 blocks in x-direction** and **13 blocks in y-direction**. 0.5 marks

Thread location:

- Block index: **second block** along the x-axis and **third block** along the y-axis, blockId=(2,1). 0.5 marks
- Thread index within the block: **third thread** along the x-axis and **fourth thread** along the y-axis, ThreadIdx=(3,2). 0.5 marks



Global thread ID in a 1D flattened grid:

$$\text{Global thread ID} = (\text{blockIdx.y} \times \text{gridDim.x} + \text{blockIdx.x}) \times \text{blockDim.x} \times \text{blockDim.y} + (\text{threadIdx.y} \times \text{blockDim.x}) + \text{threadIdx.x}$$

$$\text{Global thread ID} = (2 \times 29 + 1) \times 32 \times 32 + (3 \times 32) + 2$$

$$\text{Global thread ID} = (58 + 1) \times 1024 + 96 + 2$$

$$\text{Global thread ID} = 59 \times 1024 + 98$$

$$\text{Global thread ID} = 60416 + 98 = 60514$$

So, the global thread ID for the thread located at the **third thread** along the x-axis and the **fourth thread** along the y-axis within the specified block is **60514**.

0.5 marks

14. Compare and contrast blocking and non-blocking communication in MPI, and provide the functions used for each. Analyze the scenarios in which one might be preferred over the other.

Solution:

Feature	**Blocking Communication**	**Non-Blocking Communication**
Behavior	Process waits until operation completes	Process can continue without waiting
Performance Impact	May lead to idle time	Can improve performance through overlapping
Use Case	Small-scale applications requiring simplicity and strict synchronization	Large-scale applications where computation and communication overlap is important
Function	**MPI_Send, MPI_Recv**	**MPI_Isend, MPI_Irecv**

Each point is for 0.5 marks