NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

CS3210 – Parallel Computing
(Semester 1: AY2022/23)

Final Assessment Paper

Time Allowed: 2 hours

INSTRUCTIONS TO STUDENTS

1. Write your Student Number only. Do not write your name.
2. This assessment paper contains **FIX** questions and comprises **SIXTEEN** printed pages.
3. Students are required to answer **ALL** questions within the space in this booklet.
4. This is an OPEN BOOK assessment. Calculators are allowed.
5. Feel free to use pencil for answering the questions.
6. Write your Student Number below.

STUDENT NO: ___________________________________

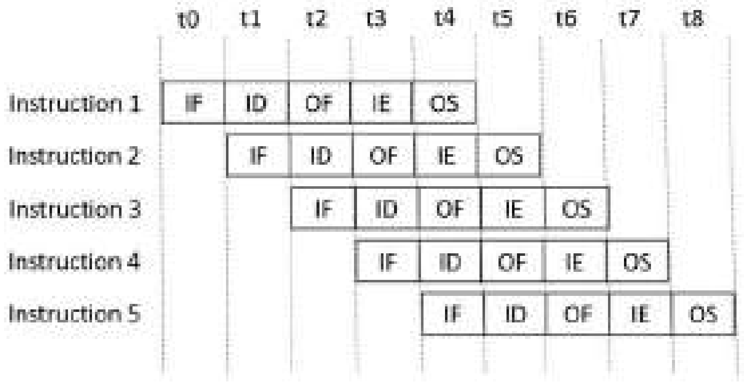
This portion is for examiner's use only

Q1. [14 marks] Parallel Performance

Consider a single-core, single-threaded processor that executes instructions using a simple five-stage pipeline. Each stage in the pipeline takes exactly 1 clock cycle to process each instruction. To keep things simple, assume this is the case for all instructions in the program, including loads and stores (memory is infinitely fast).

The figure shows the execution of a program with five independent instructions on this processor. The pipeline has 5 stages (Instruction Fetch, Instruction Decode, Operand Fetch, Instruction Execute, and Operand Store).

Note: if instruction 2 depends on the results of instruction 1, instruction 2 will not begin the IF (instruction fetch) stage of execution until the clock cycle after the OS (operand-store) stage completes for instruction 1.



Consider the following C program snippet (assume the snippet successfully compiles and runs):

```
Line#  C-like Pseudo-code
1      float a[500000];
2      float b[500000];
3      // assume a is initialized here
4      for (int i=0; i<500000; i++) {
5          float x1 = a[i];
6          float x2 = 9 * x1;
7          float x3 = 4 + x2;
8          b[i] = x3;
9      }
10
```

(Assume the code compiles and runs successfully.)

a. [6 marks] Consider *only* the four lines in the loop body (lines 5 – 8) (for simplicity, disregard instructions for managing the loop or calculating load/store addresses). Assume each of the 4 lines is represented by a single machine instruction. What is the average instruction throughput in instructions per cycle of this program? Justify your answer.
In your answer, show at least two loop iterations worth of work.
Assume there is no pipeline forwarding (ignore this term if you don't know what it means).
Mention any assumptions you have made.

Answer:

Grading Scheme

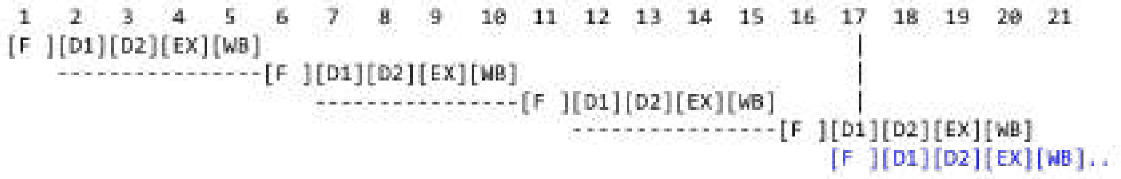
- [1m] Identification of instruction dependencies and/or execution behaviour of instructions across *at least* two loop iterations
  - No credit if execution behaviour only considers instructions from one loop iteration
- [1m] Correct calculation of average instruction throughput
  - No credit if calculation disagrees with the stated dependencies/execution behaviour
  - No credit if calculation does not extend the execution behaviour from the analysed loop iterations to *all 500000* iterations

Common Solution

- Instructions from different loop iterations are entirely independent → the last instruction of an iteration can be pipelined with the first instruction of the next iteration
- All four instructions in a loop iteration are dependent on the previous instruction
- Total number of instructions, $N_i = 4 \times 500000 = 2000000$

Solution

- Since the instruction-fetch stage cannot run unless there is no dependency between the current and next instruction – for all 4 instructions in the loop, they cannot be pipelined.
  - E.g., line 5 will run for 5 cycles (t0 … t4) and Line 6 will not run until t5.
  - A single loop iteration in *isolation* will take 4 ins x 5 cycles = 20 cycles to run
  - However, except the first instruction cycle, we can consider that every other loop takes 4 less cycles (20 – 4 = 16) since it's pipelined with the previous loop iteration (see diagram below)
  - Therefore, total number of cycles = cycles for all other loop iterations + cycles for first loop iteration = (500,000 x 16) + 4 = 8,000,004
  - Total number of instructions = 4 x 500,000 = 2,000,000
  - 2,000,000 / 8,000,004 = 0.249.. = 0.250 IPC
- Instruction throughput has units of instructions per unit time. Thus, answers are accepted in units of instructions/cycle or instructions/second (given some assumed clock frequency). Leniency was given for answers in CPI (clocks per instruction).



```
 1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21
[F ][D1][D2][EX][WB]                                            |
    ----------------[F ][D1][D2][EX][WB]                        |
                        ----------------[F ][D1][D2][EX][WB]    |
                                            ----------------[F ][D1][D2][EX][WB]
                                                                [F ][D1][D2][EX][WB]..
```

- The average instruction throughput is thus

$$R = \frac{N_I}{t_{cyc}} = \frac{2000000}{16 \times 500000 + 4} = \frac{500000}{2000001} = \mathbf{0.250\ (3\ dp)}$$

Common Mistakes

- Forgetting to multiply the number of loop instructions by the number of instructions in an iteration, giving $N_I = 500000$ instead
- Only computing the average instruction throughput for *exactly* 2 iterations, instead of extending the execution behaviour of 2 iterations to all 500000 iterations
- Not pipelining the first instruction of the next iteration with the last instruction of the first iteration – all instructions run sequentially (not how pipelined execution works!)
- Allowing multiple instructions to co-exist in the same pipeline stage in a given cycle (this is a structural hazard!)
- Calculating *degree of parallelism*, i.e. total sequential cycles/total cycles (dimensionless)
- Calculating average *instruction latency* instead, i.e. total cycles/total instructions (units of cycles/instruction)
- Calculating any metric that includes the unit of loop iterations
- Calculating any metric that includes the unit of pipeline stages
- Plotting execution behaviour for an incorrect number of instructions per iteration, i.e. 3/5
- Concluding that the throughput is 5 since the pipeline has 5 stages – note this is in fact (incorrectly) computing the *parallel speedup* assuming there are no dependencies!
- Assume the loop body compiles to include an additional load and store for each line, then failing to account for the additional RAW dependencies

Forgetting that a dependency between two instructions propagates stalls downstream to subsequent instructions

b. [2 marks] Modify the program to achieve peak instruction throughput on the processor. Please give your answer in C-like (or C++) pseudo-code. The program should continue to achieve the same tasks. Explain your approach.

Answer:

The accepted solution for this question depends on the (implicit or explicit) assumptions made in Q2(a), as it determines the average instruction throughput of the provided C program.

To obtain full credit for this part, your solution must:

- Correctly compute the whole array b
- Either:
  - **[R1/R2a/R2b/R3b]** Differ from the provided C program snippet **AND** exhibit the *maximum* possible instruction throughput
    - This includes additional instructions introduced, if any
  - **[R3a]** Make no change to the provided C program snippet **AND** explain why it already achieves the *maximum* instruction throughput

Grading Scheme

- [1m] **CORRECTNESS**: semantically correct C pseudo-code that computes the array b
  - No penalty for syntax errors
  - No penalty if there are a constant number of incorrect elements in b
- [1m] **OPTIMISATION**: Sound explanation (calculation optional) why your solution improves upon/maximises the average instruction throughput of the provided C program snippet
  - No credit if no explanation provided or the average instruction throughput is unchanged
- [1m] **OPTIMALITY**: Solution achieves close to (within 1% of) the *maximum* possible average instruction throughput
  - No credit for sub-optimal solutions

Common Solution

Observe that instructions from different loop iterations are independent. This provide us an insight to achieve the *maximum possible* instruction throughput: **partial unrolling** of the for-loop.

In addition, we accept another solution that attains *close to* (but not exactly) the *maximum possible* instruction throughput: **strided execution**.

Approach 1: partial unrolling

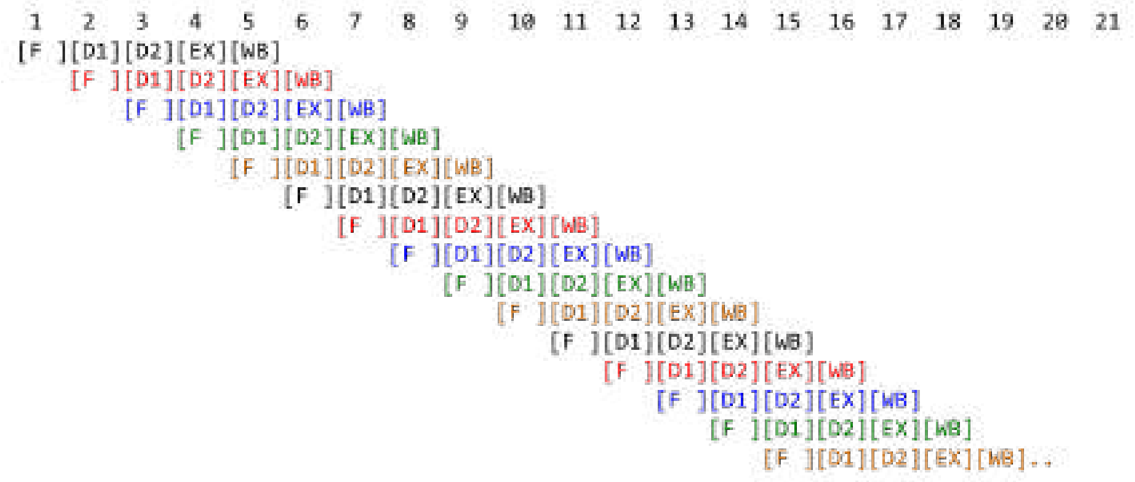
Depending on the assumption made in Q2(a), an instruction stalls the next dependent instruction for X cycles before it is allowed to execute → we can perform that same instruction from the next X iterations, eliminating the need to stall the dependent instruction.

Thus, each loop iteration can be re-written to *concurrently* perform the work of $(X + 1)$ loop iterations. Shown below is an example for assumption **[R1]**, where the $X = 4$ cycle stall for dependent instructions are hidden by weaving in 4 other instructions from other iterations.

```
for (int i = 0; i < 500000; i += 5) {
    float x1_1 = a[i];
    float x1_2 = a[i + 1];
    float x1_3 = a[i + 2];
    float x1_4 = a[i + 3];
    float x1_5 = a[i + 4];
    float x2_1 = 9 * x1_1;
    float x2_2 = 9 * x1_2;
    float x2_3 = 9 * x1_3;
    float x2_4 = 9 * x1_4;
    float x2_5 = 9 * x1_5;
    float x3_1 = 4 + x2_1;
    float x3_2 = 4 + x2_2;
    float x3_3 = 4 + x2_3;
    float x3_4 = 4 + x2_4;
    float x3_5 = 4 + x2_5;
    b[i] = x3_1;
    b[i + 1] = x3_2;
    b[i + 2] = x3_3;
    b[i + 3] = x3_4;
    b[i + 4] = x3_5;
}
```

[Note that you can reduce the number of temporary variables by re-using x1_B where B is the relative iteration number in the loop body.]

This modified for-loop has a different execution behaviour on the processor (in effect interleaving the execution of $X + 1$ loop iterations together):



```
 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
[F ][D1][D2][EX][WB]
    [F ][D1][D2][EX][WB]
        [F ][D1][D2][EX][WB]
            [F ][D1][D2][EX][WB]
                [F ][D1][D2][EX][WB]
                    [F ][D1][D2][EX][WB]
                        [F ][D1][D2][EX][WB]
                            [F ][D1][D2][EX][WB]
                                [F ][D1][D2][EX][WB]
                                    [F ][D1][D2][EX][WB]
                                        [F ][D1][D2][EX][WB]
                                            [F ][D1][D2][EX][WB]
                                                [F ][D1][D2][EX][WB]
                                                    [F ][D1][D2][EX][WB]
                                                        [F ][D1][D2][EX][WB]..
```

Approach 2: strided execution

Here, we rewrite the for-loop such that it executes one instruction from four successive iterations in a sliding window fashion, in effect "performing pipelining in software".

However, this does not *strictly* attain the maximum possible average instruction throughput, as some dependencies remain during the warm-up and spin-down of the "software pipeline".

Additionally, since there are only 3 other instructions between two dependent instructions from the same initial loop iteration, this solution is sub-optimal for scenario R1.

This solution also requires setup for mutable variables (to carry results across iterations), and requires the operations in the loop body to be *reversed* – otherwise the computation is incorrect.

```
// Warm-up software pipeline
float x1 = a[0];
float x2 = 9 * x1;
x1 = a[1];
float x3 = 4 + x2;
x2 = 9 * x1;
x1 = a[2];
```

```
for (int i = 3; i < 500000; i++) {
    // All four instructions here are completely independent
    b[i - 3] = x3;
    x3 = 4 + x2;
    x2 = 9 * x1;
    x1 = a[i];
}

// Spin-down software pipeline
b[499997] = x3;
x3 = 4 + x2;
x2 = 9 * x1;
b[499998] = x3;
x3 = 4 + x2;
b[499999] = x3;
```

Common Solution

We see from the execution behaviour that in each cycle, a new instruction will be inserted into the pipeline, and an existing instruction in the pipeline is retired.

Thus, the pipeline is always full (except during the initial warm-up and ending spin-down phases), and thus the modified program achieves the maximum *possible* average instruction throughput.

Common Mistakes

- (Not a solution) Incorrectly computing the output array b
- (Not a solution) Combining lines 5 to 8 into fewer lines of C code: b[i] = (a[i] * 9) + 4 or b[i] = (x1 * 9) + 4 does not change the number of assembly instructions!
    - This is accepted as a partial solution only if it is assumed the processor contains a MAC (multiplier-accumulator) unit and can execute multiply-accumulate instructions
- (Not a solution) Re-writing the outer loop to execute 5 iterations of the original loop *sequentially* → there is no change to the execution order of instructions!
- (Partial solution) Complete unrolling/strided execution that operates on partial results directly in memory, e.g. b[i] = x1 * 9; this necessitates an extra load and store for each line
- (Partial solution) Dividing the original loop body into two or more separate for-loops, but each new for-loop still exhibits dependencies between instructions from the same iteration
- (Partial solution) Re-writing and re-arranging the operations in the loop body → does not fully remove all pipeline stalls
- Using OpenMP/SIMD when not explicitly allowed to

c. Now assume the program is reverted to the original code, but the *for-loop* is parallelized using OpenMP, as follows (assume the snippet successfully compiles and runs).

Line# C-like Pseudo-code

```
/* assume iterations of this FOR LOOP are parallelized
across multiple worker threads in a thread pool. */
#pragma omp parallel for
for (int i=0; i<100000; i++) {
    float x1 = a[i];
    float x2 = 2*x1;
    float x3 = 3 + x2;
    b[i] = x3;
}

```

[6 marks] Given this program, imagine you wanted to choose a single-core processor with sufficient hardware threads (known as simultaneous multi-threading or hyperthreading) to obtain peak instruction throughput (100% utilization of execution resources). What is the smallest number of threads your chosen processor should support to achieve this goal? You may not change the program. Justify your answer.

Answer:

<u>**Grading Scheme**</u>

- [3m] Stating the correct *smallest* number of threads the processor should support that aligns with the assumption in Q2(a)
- [3m] Sound justification on why this is the *minimum* number of threads required for *peak* instruction throughput
  - No credit awarded if no justification is provided, or justification contradicts the question
  - Partial credit awarded if justification is incomplete, or the stated number of threads is not the minimum

<u>**Common Solution**</u>

glibc's OpenMP implementation defaults to static scheduling, where N loop iterations are divided into equal-sized chunks and statically assigned to OpenMP threads. Each OpenMP thread then executes iterations in its chunk sequentially.

Depending on the assumption made in Q2(a), dependent instructions stall for X cycles until a certain stage of the previous instruction finishes.

We can <u>hide</u> this X-cycle stall by interleaving the corresponding instruction from X other threads (which are **independent**) → processor needs to support $(X + 1)$ threads in total.

Remark: this is similar to how GPUs manage to hide the high latency of executing a single instruction from a warp, by maintaining and scheduling a large number of warps.

<u>**Common Mistakes**</u>

- Giving an excessive number of threads, e.g. 100000 – physically infeasible to support such a large number of hardware threads
- Specifying too few threads $Y < X$ to hide a X-cycle stall of dependent instructions, which does not fully eliminate stalls in all Y threads
- Counting the number of stalled cycles X between dependent instructions but forgetting to include 1 extra thread for the original loop iteration
- Incorrect or incomplete explanation: "able to run X threads concurently from the diagram" or "we need X threads to fill all X stages of the pipeline" or "4 threads as there are 4 instructions in the body of the for-loop"
- Referencing the number of physical or logical cores on the processor, despite being told the processor only has a single physical core *without* multi-threading
- Mentioning the overheads of context-switching between software threads (we are discussing adding *multi-threading* to the **PROCESSOR**, not the program!)
- Mentioning the degree of hardware multi-threading is irrelevant since it does not help with parallelism/ILP (not true – instructions from multiple hardware threads can be interleaved to fill the stalls incurred by RAW dependencies!)
- Mentioning the degree of parallelism of the original C code/number of independent "tasks" and using it to infer the number of threads required (note that all the hardware threads **share** execution resources!)
- Mentioning Amdahl's or Gustafson's law
- Mentioning X-way SIMD (this processor only has one set of execution resources and thus *cannot* support SIMD!)

Mentioning it is impossible to achieve peak instruction throughput

Q3. [14 marks] SortN in Parallel

In answering points a.-c. use the problem description of **SortN from Appendix Q3&Q4.**

Suppose you are working on parallelizing SortN problem from Figure 1. Figure 2 shows a visual representation of the MergeSort algorithm.

a. [2 marks] What part(s) of the code would you choose to parallelize from Figure 1? Briefly justify your choice and explain any assumptions you might have made.

Answer:

[2 marks] The calls to sequential_mergesort can be done in parallel: lines 39 and 40. The temp array has to change to use different arrays or different parts of the same array.

The call to merge is difficult to parallelize, but possible.

[1 mark] Only the call to merge is done in parallel.

b. [6 mark] What are two problems that you observe when you try to parallelize this sequential implementation for SortN (Figure 1)? Explain these problems with examples (Note that there is no need to give a solution for these problems).

Answer:

Any 2 of the following points bring full marks. 1 mark is deducted is explanation is unclear or incomplete.

- Limited parallelism: only two tasks are created at each step, and synchronization is needed before the merge is called in line 41.
- Total task number varies at each stage – very few tasks in the beginning and at the end
- Task granularity varies and can be too small
- Merge is not parallelized (is done sequentially as part of each task)
- Temp array might be shared producing race conditions (just mentioning race conditions does not count).

c. [6 marks] Next you are required to parallelize the sequential implementation shown in Figure 1 using OpenMP. How would you implement this parallelization (you can use the line numbers to refer to the lines of pseudo-code in Figure 1)? You should not try to optimize this approach.

Answer:

Using OMP sections or tasks is accepted.

[0/1 marks] – OpenMP pseudo-code is not attempted (only explained)

```
void parallel_mergesort_omp (int a[], int n, int temp[])
{
    if (n < size_threshold) {
        serial_mergesort(a, n, temp);
        return;
    }
    #pragma omp parallel sections  {
     #pragma omp section
     parallel_mergesort_omp (a, n/2, temp);
     #pragma omp section
     parallel_mergesort_omp (a + n/2, n - n/2, temp +
n/2);
    }
    merge(a, n, temp);
}
```

Q4. [14 marks] SortN in MPI

In answering points a.-c. use the problem description of **SortN from Appendix Q3&Q4.**

Suppose you are working on parallelizing SortN problem using MPI. Figure 2 shows a visual representation of the MergeSort algorithm. You may alter this algorithm when working on your parallel implementation.

a. [6 marks] Assume your approach in MPI is to divide the array into chunks, sort them and merge the results together. Sketch this implementation in MPI-like pseudo-code.
   You should not try to optimize this approach.

Answer:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

void merge(int *, int *, int, int, int);
void mergeSort(int *, int *, int, int);

int main(int argc, char** argv) {

        /********** Create and populate the array **********/
        int n = atoi(argv[1]);
        int *original_array = malloc(n * sizeof(int));

        int c;
        srand(time(NULL));
        printf("This is the unsorted array: ");
        for(c = 0; c < n; c++) {

                original_array[c] = rand() % n;
                printf("%d ", original_array[c]);

                }

        printf("\n");
        printf("\n");

        /********** Initialize MPI **********/
        int world_rank;
        int world_size;

        MPI_INIT(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        /********** Divide the array in equal-sized chunks **********/
        int size = n/world_size;

        /********** Send each subarray to each process **********/
        int *sub_array = malloc(size * sizeof(int));
        MPI_Scatter(original_array, size, MPI_INT, sub_array, size,
MPI_INT, 0, MPI_COMM_WORLD);

        /********** Perform the mergesort on each process **********/
        int *tmp_array = malloc(size * sizeof(int));
        mergeSort(sub_array, tmp_array, 0, (size - 1));

        /********** Gather the sorted subarrays into one **********/
        int *sorted = NULL;
        if(world_rank == 0) {

                sorted = malloc(n * sizeof(int));

                }

        MPI_Gather(sub_array, size, MPI_INT, sorted, size, MPI_INT, 0,
MPI_COMM_WORLD);

        /********** Make the final mergeSort call **********/
        if(world_rank == 0) {

                int *other_array = malloc(n * sizeof(int));
                mergeSort(sorted, other_array, 0, (n - 1));

                /********** Display the sorted array **********/
                printf("This is the sorted array: ");
                for(c = 0; c < n; c++) {

                        printf("%d ", sorted[c]);

                        }

                printf("\n");
                printf("\n");

                /********** Clean up root **********/
                free(sorted);
                free(other_array);

                }

        /********** Clean up rest **********/
        free(original_array);
        free(sub_array);
        free(tmp_array);

        /********** Finalize MPI **********/
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();

        }

/********** Merge Function **********/
void merge(int *a, int *b, int l, int m, int r) {

        int h, i, j, k;
        h = l;
        i = l;
        j = m + 1;

        while((h <= m) && (j <= r)) {

                if(a[h] <= a[j]) {

                        b[i] = a[h];
                        h++;

                        }
                                else {
```

```
                        b[i] = a[j];
                        j++;

                        }
                i++;

                }

        if(m < h) {

                for(k = j; k <= r; k++) {

                        b[i] = a[k];
                        i++;

                        }

                }

        else {

                for(k = h; k <= m; k++) {

                        b[i] = a[k];
                        i++;

                        }

                }

        for(k = l; k <= r; k++) {

                a[k] = b[k];

                }

        }

/********** Recursive Merge Function **********/
void mergeSort(int *a, int *b, int l, int r) {

        int m;

        if(l < r) {

                m = (l + r)/2;

                mergeSort(a, b, l, m);
                mergeSort(a, b, (m + 1), r);
                merge(a, b, l, m, r);

                }

        }
```

b. [2 marks] What parallel pattern did you use in your implementation from point a. (No justification needed.)

Answer:

Master-worker

c. [6 marks] What is one of the main performance issues particular to your MPI implementation from point a.? Explain this issue using theoretical concepts and laws discussed in CS3210.

Answer:

One of the following with theoretical explanation:

- Amdahl's law, or other issue.
- Communication overhead

Q5. [12 marks] Bitonic Sort in CUDA

In answering points a.-d. use the problem scenario of **SortN** from **Appendix Q3&Q4** and the problem scenario for **Bitonic Sort** from **Appendix Q5**.

In this question, we look at a different algorithm to solve the SortN problem called **Bitonic Sort**. This algorithm is explained in **Appendix Q5**.

a. [2 marks] The complexity of Merge sort to solve the sequential **SortN** problem (Figure 1) is `O(N log (N))`. However, the complexity of solving the sequential SortN problem using **Bitonic sort** (Figure 3) is $O(N \log^2 (N))$.
Explain why and when you would choose to use Bitonic sort. Be specific in your explanation. If needed, use a convincing example to illustrate your explanation.

Answer:

MergeSort is difficult to parallelize and has many downsides when parallelized (Q2c). BitonicSort takes advantage of parallelism, can be easily parallelized, and the work is independent of the input data. The number of tasks can be easily scaled.

BitonicSort is done in-place, while MergeSort needs additional memory space for a temp array.

b. [2 mark] Consider the MergeSort algorithm (Figure 1) and the Bitonic sort algorithm (Figure 3) to solve SortN problem. Which of the two algorithms will perform *better* when implemented in CUDA (running on an Nvidia GPU)? Justify your choice and mention why the implementation is *better*.
Note: In your explanation, you should not refer to the type of memory used to implement the two algorithms. You may assume that the memory is used in an equivalent way for each algorithm.

Answer:

Bitonic sort because the work is oblivious (independent) to the input data. Each thread in each step will have the same amount of work.

Explanations related to the recursive implementation of MergeSort (that cannot run on CUDA) were accepted.

Divergent control flows are possible for both BitonicSort and MergeSort. Hence that is not a reason for choosing BitonicSort over MergeSort for CUDA.

Explanations referring to global and shared memory are not accepted.

c. [6 marks] Briefly describe the jobs (tasks) done in parallel by each CUDA thread when Bitonic sort algorithm (Figure 3) is implemented to solve SortN problem. You should minimally show:
   - Kernel code in pseudo-code (or explanation)
   - When and how many times is the kernel called
   - Number of blocks, number of threads in a block, virtual arrangement of the threads (1D/2D/3D)
   - Type of memory that the kernel is using

Answer:

For an array of length N = 2^k, we need 1+2+ 3+..+k rounds of sorting (total k(k + 1)/2 rounds, divided into k stages, as shown in Figure 3). The kernel is called k(k+1)/2.

Each round calls a kernel function, which completes N/2 compare-exchange operations. Threads are grouped in an 1D block, number of threads = N/2.

Use global memory; it is not clear that shared memory would bring any advantage in this approach (as the kernel is called multiple times)

You can find some implementations and further hint about how to parallelize on GPU here:

- https://gist.github.com/mre/1392067
- https://developer.nvidia.com/gpugems/gpugems2/part-vi-simulation-and-numerical-algorithms/chapter-46-improved-gpu-sorting

Alternative approaches where the kernel does the work of a stage are accepted.

d. [2 marks] Assume N is very large (N>= 4096). Give one reason that makes your CUDA implementation of Bitonic sort inefficient. Briefly explain how you would attempt to fix the issue.
   If you feel that your implementation from point c. is efficient enough, explain why.

Answer:

You need multiple blocks.

Assume shared memory is used somehow and maximum block size is 1024 threads. There would not be enough shared memory if N is too big.

The approach would be to split the data into chunks. It might be better to try an approach where sections of the array are sorted separately, followed by a parallel merge.

Other explanations referring to the number of threads, starting the kernel multiple times, etc. were accepted.

END OF PAPER

Do not write your answers on this sheet. This sheet should not be submitted with your paper.

Appendix Q3&Q4. Problem scenario for Q3 and Q4

SortN: Sorting an array of N integer numbers using MergeSort

We refer to this problem using the name **SortN**. Please find below the implementation of the sequential **MergeSort** algorithm that sorts an array a of size n.

```
Line#  Pseudo-code
1      void merge(int a[], int size, int temp[]) {
2           int i1 = 0;
3           int i2 = n / 2;
4           int it = 0;
5           while(i1 < n/2 && i2 < n) {
6                if (a[i1] <= a[i2]) {
7                     temp[it] = a[i1];
8                     i1 += 1;
9                }
10               else {
11                    temp[it] = a[i2];
12                    i2 += 1;
13               }
14               it += 1;
15          }
16          while (i1 < n/2) {
17              temp[it] = a[i1];
18              i1++;
19              it++;
20          }
21          while (i2 < n) {
22              temp[it] = a[i2];
23              i2++;
24              it++;
25          }
26          memcpy(a, temp, n*sizeof(int));
27     }
28     void serial_mergesort(int a[], int n, int temp[])
29     {
30          int i;
31          if (n == 2) {
32               if (a[0] <= a[1])
33                    return;
34               else {
35                    // Swaps values of args
36                    SWAP(a[0], a[1]);
37                    return;
38               }
39          }
40          serial_mergesort(a, n/2, temp);
41          serial_mergesort(a + n/2, n - n/2, temp);
42          merge(a, n, temp);
       }
```

Figure 1: Sequential Implementation of MergeSort

Do not write your answers on this sheet. This sheet should not be submitted with your paper.

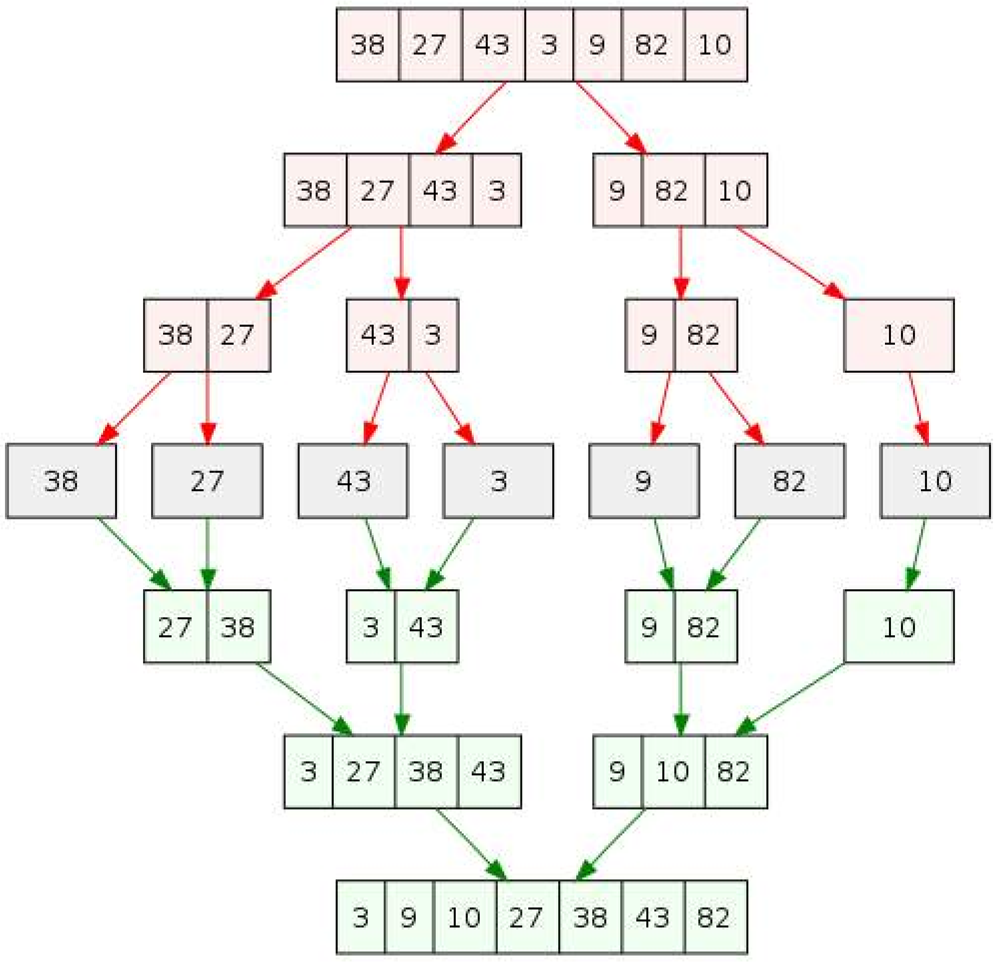


Figure 2: MergeSort Visual Representation

Do not write your answers on this sheet. This sheet should not be submitted with your paper.

Appendix Q5. Problem scenario for Q5: Bitonic Sort

Figure 3 shows how the sorting of `n` numbers can be done by comparing and swapping each of the array elements. This algorithm is called **Bitonic Sort**.

For example, the following sequence: `1,5,9,10,12,8,7,2` is a bitonic sequence because the first half of the sequence is monotonically increasing, while the second half of this sequence is monotonically decreasing.

Let A be an arbitrary input sequence to sort and let $n = 2^k$ be the length of `A.` The process of sorting `A` then consists of `k` stages. The subsequences of length 2 `(A[0],A[1]),(A[2],A[3]),...,A[n-2],A[n-1])` are bitonic sequences by definition.

- In the first stage, these subsequences are sorted using bitonic merge alternating descending and ascending, which makes the subsequences of length 4 bitonic sequences: `(A[0],A[1],A[2],A[3]),...,(A[n-4],A[n-3],A[n-2],A[n-1])`.
- In the second stage, these subsequences of length 4 are sorted alternating descending and ascending, resulting in subsequences of length 8 being bitonic sequences.
- In the `r`th stage of bitonic sort the total number of subsequences being sorted is 2^{k-r} and the length of each of these subsequences is 2^r. Sorting a sequence of length 2^r using bitonic merge consists of `r` steps.

Figure 3 shows the bitonic sorting network for input sequences of size `n = 8`. At each stage, we obtain bitonic sequences of increasing sizes by swapping two values in the array (as indicated by the arrows). The sorting network consists of `3 = log(8)` stages. For example, in stage 2, we observe 2 steps. The first step of stage 2 sorts pairs (`A[0]` and `A[2]`) and (`A[1]` and `A[3]`) in ascending order, and pairs (`A[4]` and `A[6]`) and (`A[5]` and `A[7]`) in descending order. The second step of stage 2 sorts pairs (`A[0]` and `A[1]`) and (`A[2]` and `A[3]`) in ascending order, and pairs (`A[4]` and `A[5]`) and (`A[6]` and `A[7]`) in descending order. At the end of stage 2, we obtain a bitonic sequence of length 8: (`A[0], A[1], …, A[7]`).

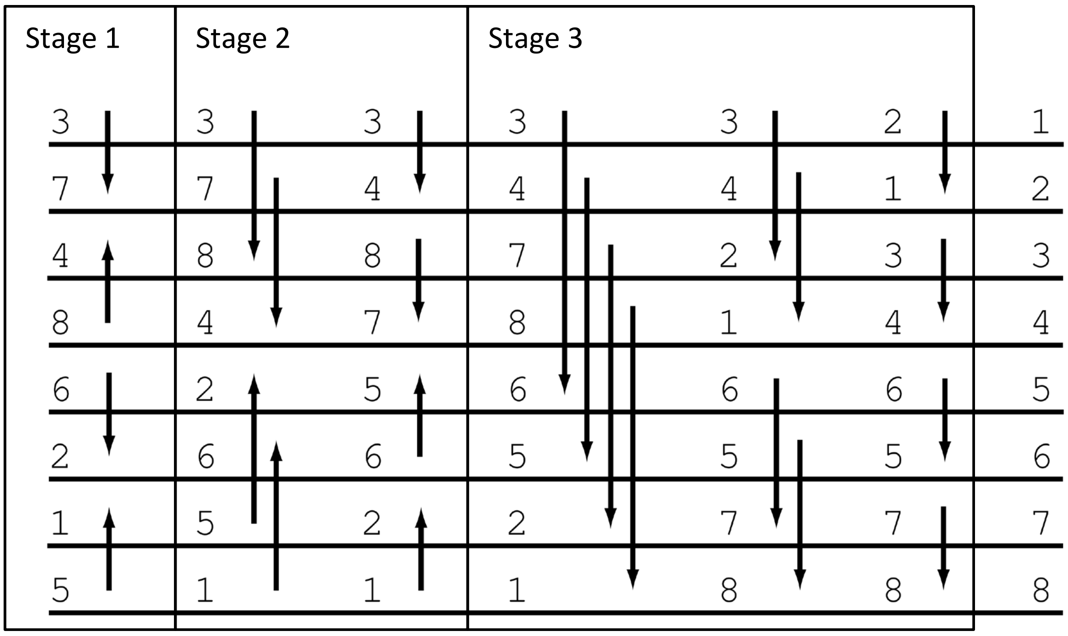


Figure 3: Applying Bitonic Sort for an array with 8 values