Solutions for the Exam in DAT400 (Chalmers) and DIT431 (GU) High Performance Parallel Programming, Thursday, October 30th, 2025, 8:30h - 12:30h

Problem 1 (12 points)

Complete the following **tasks** concerning the theory of parallel programming models:

(a) Identify if the following statements are true or false:

(1) The single goal of scheduling is to achieve load balancing.

(2) In scenarios of ample parallelism (many independent tasks), greedy schedulers achieve good load balancing.

(3) Since fine-grained tasking exposes more parallelism, one should aim to define the smallest possible tasks as scheduling units.

(4) Static scheduling minimizes overheads and is therefore always optimal in all scenarios.

(5) Data parallelism is the same as loop parallelism.

(6) SPMD is an embodiment of data parallelism.

(7) Implicit parallelism (e.g. auto-parallelization, functional programming) simplifies the programmer's task.

(8) OpenMP, MPI and CUDA are all examples of implicitly parallel programming models.

(b) Given the following vectorizable loop:

```
#define N 64
for(int i = 0; i < N; i++)
    b[i] = a[i] + b[i];  // a[] and b[] are non-overlapping
```

Consider the three implementations of vector architectures described in class: (i) pipelined "vector" architecture, (2) parallel SIMD architecture, and (3) vector lanes architecture, with the following characteristics:

- The latency of one addition is 5 cycles.
- For the pipelined architecture, the Initiation Interval is 2 cycles, and the vector length is 32.
- For the SIMD architecture, the width of the SIMD unit is 4 (i.e., each vector instruction can process four words in parallel), and the ALUs are not pipelined.
- For the Vector lanes architecture, the number of vector lanes is 4, and the vector length is 32. The Initiation Interval is 2.

Calculate the execution time of this loop for each of the three architectures.

(c) Next, consider the following variation of this loop

```
#define N 64
for(int i = 0; i < N; i++)
    b[i] = a[i] + b[N-i-1];  // a[] and b[] are non-overlapping
```

Describe why this loop cannot be simply vectorized, and propose a transformation to enable its vectorization.

(d) A team of developers wants to implement a novel implementation of OpenMP in which each call to `#pragma omp parallel` creates a team of ULTs (user level threads), instead of kernel level threads (KLT). Their goal is to support massive parallelism by enabling cases of `#pragma omp parallel num_threads(N)` with values of N orders of magnitude larger than the total number of cores in the system. They hope that by dynamically scheduling many OpenMP threads, load imbalance will be reduced and nested parallelism (i.e. calling `#pragma omp parallel` within `#pragma omp parallel`) will be better supported.

The developers are considering the three ULT to KLT mapping schemes: N:1, 1:1 and N:M. What method should they choose and why?

(e) Consider an application that processes a large array in a loop. The value of `array[i]` is updated during the execution of `process(int *)`.

```
int array[SIZE];
for(auto i=0; i<SIZE; i++)
    process(array+i);
```

The execution time of iteration i (`process(array+i)`) is proportional to the value of i: $\text{Exec Time}_i = \text{c} \times \text{i}$, for some constant c. The developers want to run this code on a system consisting of multiple shared memory multiprocessors.

Discuss individually the suitability of the following data distributions: Blockwise, Cyclic or Block-Cyclic. Based on your analysis, which is the best one? Considering that the hardware has a cache line of 64 bytes, and that the size of an `int` is 4 bytes, propose optimal values for the selected partitioning.

Answers:

(a)

1. False. Scheduling can have other goals such as energy consumption, or meeting a deadline. In such cases, load balancing is a secondary goal.

2. True. When there is high parallel slackness, greedy scheduling will lead to good load balancing and good execution time bounds

3. False. It does expose more parallelism which may be good for a greedy scheduler, but the scheduling overheads can degrade performance if tasks are too small

4. False. Static scheduling can only be optimal of the execution is independent of dynamic conditions. In the case of data dependent load imbalance, or external interference on the cores, a dynamic scheduler is likely to be better.

5. False. There are forms of data parallelism which do not involve loops (e.g. SIMD, SPMD)

6. True. SPMD (Single Program, Multiple Data) is a parallel programming model where a single program runs simultaneously on multiple processors, with each processor operating on a different portion of the data. This is a fundamental characteristic of data parallelism

7. True. Implicit parallelism means that there are no code modifications solely for expressing parallelism. This allows programmers to focus on the functionality, rather than parallelization

8. False. All these programming models have syntax and APIs specific for parallelism, hence they are explicitly parallel

(b) Addition of two vectors of 64 elements

Pipelined architecture: Two vector instructions will be required (since width is 32) 2 *(Lat + II * (32 - 1)) = 2 (5 + 2 * (31)) = 2 * 67 = 134 cycles

SIMD architecture: has width 4. Need 16 instructions 5 * 16 = 80 cycles

Vector lanes: Again it will be two vector instructions. Each of them distributed across four lanes 2 * (Lat + II * (8 -1)) = 2 (5 + 2 * 7) = 2 * 19 = 38 cycles

(c) This loop cannot be simply vectorized, because the second half of the iterations depend on modifications that are done during the first half. But it is possible to vectorize it by splitting the loop into two parts, each of which is individually vectorizable

```
#define N 64

#pragma omp simd
for(int i = 0; i < N/2; i++)
    b[i] = a[i] + b[N-i-1];

#pragma omp simd
for(int i = N/2; i < N; i++)
    b[i] = a[i] + b[N-i-1];
```

(d)

N:1 is clearly suboptimal. All the OpenMP ULTs will be mapped to a single KLT, which means only one core can be used. In this scenario, supporting many tasks is meaningless, since load balancing is never an issue

1:1 is also problematic, since each ULT will require a separate KLT. The number of ULTs is "orders of magnitude" larger than the number of cores, so it will be in the order of thousands or more. This is likely to overwhelm the OS and lead to low performance.

N:M is the only reasonable option. The number of KLT should be similar to the number of cores. N:M means that the runtime library will select a subset of OpenMP ULTs and map them to the KLTs dynamically, as ULTs complete.

(e)

Blockwise decomposition will not be a good option here since it will lead to load imbalance (the final set of elements having much longer execution times).

Cyclic will lead to good load balance, but it will incur false sharing since each element is just an integer of four bytes.

The best trade-off will be block cyclic. We want each processor to access a single block. There are 64/4 elements in a cacheline, hence a cyclic distribution with a chunk size of 16 elements will be a good choice.

Problem 2 (9 points)

Assume we have a loop with 12 iterations with execution times (ms): [1,2,1,2,1,2,10,10,10,1,2,1] The system consists of three cores (threads), and we can choose one of the following scheduling methods:

(i) Static

(ii) dynamic (chunk=1)

(iii) guided self-scheduling (min chunk=1)

Consider the following overheads:

- For static scheduling, there are no per-iteration overheads
- For dynamic scheduling, we consider an overhead of 1 ms for each scheduling request

Tasks:

(a) Compute the total execution time and idleness for each scheduling method, assuming that there are no dynamic scheduling overheads.

(b) Evaluate which scheduling method is more efficient when the overheads of dynamic scheduling are considered and explain why.

(c) Next, compute the execution time for a case in which the very first chunk for guided self-scheduling is changed from 4 to 2, but the algorithm performs as expected for the rest of iterations. How does it affect execution time?

(d) Describe one scenario in which guided scheduling should perform best.

Answers:

(a)

Table 1: Static Scheduling

Thread	Iterations	Execution Time	Idle time
T1	0, 1, 2, 3	$1+2+1+2=6$	17
T2	4, 5, 6, 7	$1+2+10+10=23$	0
T3	8, 9, 10, 11	$10+1+2+1=14$	9

Table 2: Dynamic Scheduling

Step	Thread	Iteration	Exec Time (ms)	Finish Time (ms)
1	T1	0	1	1
1	T2	1	2	2
1	T3	2	1	1
2	T1	3	2	$1+2=3$
2	T3	4	1	$1+1=2$
2	T2	5	2	$2+2=4$
3	T3	6	10	$2+10=12$
3	T1	7	10	$3+10=13$
3	T2	8	10	$4+10=14$
4	T3	9	1	$12+1=13$
4	T1	10	2	$13+2=15$
4	T2	11	1	$14+1=15$

Table 3: Dynamic Scheduling (Finish time & idle time)

Thread	Iterations (Exec Time)	Total Finish Times	Idle time
T1	0(1), 3(2), 7(10), 10(2)	15	0
T2	1(2), 5(2), 8(10), 11(1)	15	0
T3	2(1), 4(1), 6(10), 9(1)	13	2

Table 4: Guided Self-Scheduling (GSS) allocation steps

Step	Remaining Workload	Chunk Size C	Thread	Allocated	Remaining
1	12	4	T1	[1,2,1,2]	8
2	8	3	T2	[1,2, 10]	5
3	5	2	T3	[10, 10]	3
4	3	1	T1	[1]	2
5	2	1	T2	[2]	1
6	1	1	T3	[1]	0

Table 5: GSS: Total execution and idle times per thread

Thread	Total Execution (ms)	Idle Time (ms)
T1	18	0
T2	14	4
T3	11	7

(b) Execution time: Static - 23 ms, Dynamic - 15+12(overhead) = 27ms, Guided Self-scheduling - 18 + 6(overhead) = 24ms
Static scheduling is fast
Idle time is more in static scheduling

(c)

Table 6: Guided Self-Scheduling (GSS) allocations with chunk size 2

Allocation	**Chunk Size**	**Thread**	**Work Assigned**	**Workload Remaining**
1	2	T1	[1,2] $\rightarrow$ 3	10
2	3	T2	[1,2, 1] $\rightarrow$ 4	7
3	2	T3	[2, 10] $\rightarrow$ 12	5
4	2	T1	[10, 10] $\rightarrow$ 20	3
7	1	T1	[1] $\rightarrow$ 1	2
8	1	T2	[2] $\rightarrow$ 2	1
9	1	T3	[1] $\rightarrow$ 1	0

Total - 20 + 7 (overhead) = 27 ms

This is even worse than initial chunk size as 4.

(d) Guided Self-Scheduling (GSS) is the better choice when loop iterations have large variability and the scheduling overheads are high compared to the execution time of each iteration.

Problem 3 (11 points)

Complete the following **tasks** concerning performance analysis and loop optimizations:

(a) Explain the difference between strong scaling and weak scaling in parallel computing.

(b) Consider a parallel program running on P processors. Let:

- T_P = execution time on P processors
- T_1 = execution time on a single processor

1. Define parallel efficiency in terms of speedup and the number of processors.
2. Explain the relationship between cost and efficiency, and discuss how both change as the number of processors increases.

(c) Loop unrolling is a common loop optimization technique. Explain how loop unrolling increases ILP (Instruction Level Parallelism), and briefly describe its role in enabling vectorization (SIMD).

Consider the following matrix-matrix multiplication kernel operating on two NxN matrices:

```
void gemm( float *A, float *B, float *C)
{
    int i,j,k;
    for(i = 0; i < N; ++i){
        for(k = 0; k < N; ++k){
            for(j = 0; j < N; ++j){
                C[i*N+j] +=  A[i*N+k]*B[k*N+j];
            }
        }
    }
}
// Where A, B and C are N X N matrices of single-precision elements (4 bytes each).
```

The system characteristics are:

- Peak compute performance: 100 GFLOP/s
- Memory bandwidth: 25 GB/s

(d) Derive the expression for Arithmetic Intensity (AI) and compute its numerical value for N = 100. Using the simplified roofline model, determine the theoretically maximum achievable performance. State whether the kernel is memory-bound or compute-bound.

(e) Consider a scientific application whose total runtime consists of two main parts:

- **A serial setup phase**, which constitutes 10% of the total execution time.
- **Matrix Multiplication kernel**, which constitutes 90% of the total execution time and is 100% parallelizable.

The total measured performance for the full application running on a single processor ($P = 1$) is 80 GFLOP/s

Compute the theoretical speedup achievable when running the full application on 4 processors using Amdahl's Law. Also, estimate the expected performance (in GFLOP/s) for the full application on 4 processors.

Answers:

(a) Strong Scaling: Measures how the execution time decreases as the number of processors P increases for a fixed total problem size.
Weak scaling: Measures how the execution time changes as both the number of processors P and the total problem size increase proportionally, keeping the work per processor constant.

(b)

$$\mathrm{ParallelEfficiency} = \frac{\text{Speedup}}{\mathrm{P}} = \frac{\mathrm{T_1/T_P}}{\mathrm{P}} = \frac{\mathrm{T_1}}{\mathrm{P}}$$

$\times$ T$_P = \frac{T_1}{C_P}$
Higher efficiency means better utilization of processors.
As P increases, T_P may decrease, but due to overheads (communication, synchronization), the total cost $C_P = P \cdot T_P$ can increase, which reduces efficiency.

(c) Loop unrolling: Loop unrolling directly increases ILP by replicating the loop body multiple times, which exposes more independent operations within a single iteration. This reduces loop overhead (branch instructions and counter updates) and increases the instruction window, allowing the processor to identify and execute multiple independent operations simultaneously. In SIMD, hides the pipeline latency by maximizing the vector register utilization and increasing the parallelism in the algorithm.

(d) FLOPs=2*N*N*N FLOPs = 2,000,000 FLOPS
Bytes = 3 * N * N * 8 = 240000 bytes
Arithmetic Intensity = FLOPS / Bytes = 8.33 FLOPS / bytes
Roofline: $\mathrm{Pmax} = \min(\mathrm{Peak}, \mathrm{Bandwidth} \times \mathrm{AI}) = \min(100, 25 \times 8.33) = 100\mathrm{GFLOPS/s}$
That means kernel is compute-bound.

(e) $speedup = 1/((1 - 0.9) + (0.9/4)) = 3.08$
Expected perf = 80 (on 1 processor) * 3.08 = 246FLOP/s
Possible reasons - Synchronization, overhead

Problem 4 (9 points)

(a) Answer whether the following statements are True or False

1. `#pragma omp parallel` forms a team; all threads execute the same structured block.
2. Variables defined outside a `parallel` region are *private* by default.
3. `#pragma omp for nowait` removes the implicit barrier at the end of that loop, but a following `single` region still introduces a barrier at its end unless `nowait` is specified on `single`.
4. `schedule(static)` will always keep the same iteration→thread mapping across loops with the same iteration space (helpful for cache reuse).

(b) Complete `/* A */` and `/* B */` so that the following OpenMP parallel dot product is correct.

```
#include <omp.h>
double dot(const double *a, const double *b, int N) {
    double sum = 0.0;
    #pragma omp parallel
    {
        #pragma omp for /* A */
        for (int i = 0; i < N; ++i) {
            /* B */
        }
    }
    return sum;
}
```

(c) Answer briefly (one–two sentences each):

(1) Define *implicit* vs. *explicit* `omp` tasks.

(2) Contrast an **OpenMP barrier** with **taskwait**: what does each one wait for?

(3) What does `final(expr)` do when `expr` evaluates to true for a `omp task`, and why might you use it?

Answers

(a)

(1) T — A team is formed and all threads enter the region (work may diverge by control/-data).

(2) F — Such variables are *shared* by default unless overridden (e.g., `default(none)+private`).

(3) T — `nowait` removes the loop-end barrier (other region barriers still apply).

(4) T — Static scheduling is deterministic and can preserve mapping, aiding locality across similar loops.

(b)

```
#include <omp.h>
double dot(const double *a, const double *b, int N) {
    double sum = 0.0;
    #pragma omp parallel
    {
        #pragma omp for schedule(static) reduction(+:sum)
        for (int i = 0; i < N; ++i) {
            sum += a[i] * b[i];
        }
    }
    return sum;
}
```

(c)

(a) **Implicit vs. explicit tasks:** An *implicit* task is created by entering a parallel region (each team thread runs one); an *explicit* task is created with `#pragma omp task [clause]` inside a structured block. Example: implicit—work executed by a thread after `#pragma omp parallel`; explicit—work created by `#pragma omp task` inside that region.

(b) **Barrier vs. taskwait:** An **OpenMP barrier** synchronizes all threads in the team and ensures completion of all tasks they created up to the barrier; **taskwait** suspends only the *encountering task* until its *direct child* tasks complete.

(c) `final(expr)`**:** If `expr` is true, the task is *included* (undeferred) and executed immediately by the encountering thread (and its descendants can be treated as final), which reduces scheduling overhead—e.g., forcing very small or deep tasks to run in place.

Problem 5 (10 points)

Complete the following **tasks** concerning the Message Passing Interface (MPI):

(a) Identify if the following statements are true (T) or false (F).

(1) In MPI, standard communication mode is guaranteed to not lead to deadlocks.

(2) In MPI, calls to blocking sends may return before the receiving process has started to receive the message.

(3) The optimal way to implement an MPI collective is to use a binary tree.

(4) In MPI, there can be at most one communicator active at a time.

The following code implements matrix-vector multiplication (Ax = b) using only P2P MPI calls. The matrix `A[]` and the vector `x[]` are initially available only in the root node (rank 0). The computation is parallelized such that each MPI rank computes a contiguous chunk of the result vector `b[]`. At the end of the execution, `b[]` must be present in all ranks.

```
float A[M*N], x[N], b[M];
float local_A[M*N/2], local_b[M/2];
// local_A and local_b will contain only a subset of the matrix and result vector.
// here we have allocated enough space for the case of only two ranks
MPI_status status;
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// MPI root: send vector x[] to all ranks
if(rank == 0) //
  for(int i = 1; i< size; i++) MPI_Send(A);
else MPI_Recv(B);

// fill in the chunk size (= number of elements) to be sent from matrix A
chunk = C;

// MPI root: send chunks of A[] (the local part is copied directly to localA[] -- ignore)
if (rank == 0)
  for(int i = 1; i< size; i++) MPI_Send(D);
else MPI_Recv(E);

// fill the number of rows and columns to be used for the local matrix multiplication
local_rows = F;
local_cols = G;

// the following routine calculates the matrix vector product
// the partial product is stored in local_b[]
mat_vec_mult(local_A, x, local_b, local_rows, local_cols);

// MPI root: collect chunks of vector b[] (the local part is copied directly -- ignore)
if(rank == 0)
  for(int i = 1; i< size; i++) MPI_Recv(H);
else MPI_Send(I);

// next distribute the result to all nodes
if(rank == 0)
  for(i = 1; i< size; i++) MPI_Send(J);
else MPI_Recv(K);
// Execution is completed here
```

(b) Fill in the arguments to the MPI send and receive calls, and the sizes of `local_rows`, `local_cols` and `chunk`. To answer this subproblem show the contents of A, B, C... K.

(c) Identify parts of the code that can be replaced by MPI collectives. What collectives should be used? Try to find the minimum number of collectives required to execute the matrix-vector multiplication.

(d) Communication-computation overlap is a common optimization in MPI programs. Briefly explain how communication-computation overlap can be achieved in MPI and identify one point in the MPI matrix-vector multiplication where this optimization can be applied.

A subset of MPI calls that are useful for this problem are shown below:

```
1  int MPI_Send (const void *buf, int count, MPI_Datatype datatype,
2      int dest, int tag, MPI_Comm comm)
3  int MPI_Recv (void *buf, int count, MPI_Datatype datatype,
4      int dest, int tag, MPI_Comm comm, MPI_Status *status)
5  int MPI_Comm_rank(MPI_Comm comm, int *rank)
6  int MPI_Comm_size(MPI_Comm comm, int *size)
7  int MPI_Barrier( MPI_Comm comm )
8  int MPI_Bcast(void* data, int count, MPI_Datatype datatype,
9          int root, MPI_Comm communicator)
10 int MPI_Gather(const void *sendbuf, int sendcount,
11         MPI_Datatype sendtype, void *recvbuf, int recvcount,
12         MPI_Datatype recvtype, int root, MPI_Comm comm)
13 int MPI_Allgather(const void *sendbuf, int sendcount,
14         MPI_Datatype sendtype, void *recvbuf, int recvcount,
15         MPI_Datatype recvtype, MPI_Comm comm)
16 int MPI_Alltoall(const void *sendbuf, int sendcount,
17         MPI_Datatype sendtype, void *recvbuf, int recvcount,
18         MPI_Datatype recvtype, MPI_Comm comm)
19 int MPI_Scatter(const void *sendbuf, int sendcount,
20         MPI_Datatype sendtype, void *recvbuf, int recvcount,
21         MPI_Datatype recvtype, int root, MPI_Comm comm)
22 int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
23         MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
24 int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
25         MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Answers

(a) True / False questions

1. This is false. Standard communication means that the runtime decides, and the behavior can be synchronous or asynchronous, for example depending on whether buffers are being used. This means that an application that has been tested to work currently with a runtime using buffers, may lead to deadlock if the runtime decides to operate in synchronous mode

2. This is correct. A blocking send just waits until the buffer can be reused. It does not require for the receiver to have started receiving messages

3. This is false. The best way to implement a reduction will depend on many factors, such as the network topology, or the number of ranks. A binary tree will be better than sequentially sending the data, but it does not need to be optimal.

4. This is false. Applications start with one communicator `MPI_COMM_WORLD`, but they can create more communicators afterwards and they all remain active.

(b)

A = `MPI_Send(x, N, MPI_FLOAT, i, 0, MPI_COMM_WORLD)`
B = `MPI_Recv(x, N, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status)`
C = `local_rows * N`
D = `MPI_Send(A + chunk*i, chunk, MPI_FLOAT, i, 0, MPI_COMM_WORLD)`
E = `MPI_Recv(local_A, chunk, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status)`
F = `M / ranks;`
G = `N;`
H = `MPI_Recv (b + local_rows*i, local_rows, MPI_FLOAT, i, 0, MPI_COMM_WORLD, &status)`
I = `MPI_Send (local_b, local_rows, MPI_FLOAT, 0, 0, MPI_COMM_WORLD)`
J = `MPI_Send (b, M, MPI_FLOAT, i, 0, MPI_COMM_WORLD)`
K = `MPI_Recv (b, M, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status)`

(c) All communication can be replaced by collectives

A + B can be replaced by mpi bcast
D + E can be replaced by mpi scatter
H + I can be replaced by mpi gather
J + K can be replaced by mpi bcast

It is possible to reduce this to only three collectives by using mpi allgather for the operations H, I, J and K

(d) to achieve this we can use non blocking mpi send followed by mpi wait

For example,

```
// MPI root: send vector x[] to all ranks
if(rank == 0) //
  for(int i = 1; i< ranks; i++) MPI_Isend(A);
else MPI_Irecv(B);

foo();
MPI_Wait();
```

Note that the sending of the chunks of A and of vector x are independent. Hence the first MPI Wait could be skipped, and we could issue MPI wait only just before the matrix multiplication. However, the solution with lower overlapping already fulfills the goals of this problem.

Problem 6 (9 points)

Complete the following **tasks** concerning CUDA:

(a) Answer whether the following statements are True or False.

(1) Operations on CUDA shared memory are as fast as operations on registers.

(2) A `__syncthreads()` is needed after threads write to `__shared__` arrays to ensure all threads see updated values.

(3) Shared memory is allocated per SM and persists for the lifetime of the application.

(4) Using more shared memory per block can reduce occupancy by limiting blocks per SM.

(b) The next task is to implement a 3-point 1-D stencil. For interior points $1 \leq i \leq n-2$:

$$\texttt{out[i]} = c_0 \cdot \texttt{in[i-1]} + c_1 \cdot \texttt{in[i]} + c_2 \cdot \texttt{in[i+1]}$$

For boundaries, use a simple copy: `out[0]=in[0]`, `out[n-1]=in[n-1]`.

Complete the following two tasks:

1. Complete `/* A, B, C, D, E, F, G, H, I, and J */` in the following partial CUDA code.
2. Modify the code to use the shared memory.

```
// Kernel: one element per thread
__global__ void stencil1D(const float* in, float* out, int n,
                          float c0, float c1, float c2) {
    int i = /*A*/;
    if (/*B*/) {
        if (/*C*/) {
            out[i] = c0 * in[i-1] + c1 * in[i] + c2 * in[i+1];
        } else {
            /*D*/
        }
    }
}

// Host: allocate, copy, launch, copy back
void runStencil(const float* hIn, float* hOut, int n,
                float c0, float c1, float c2) {
    size_t bytes = n * sizeof(float);
    float *dIn = nullptr, *dOut = nullptr;
    /* E: cudaMalloc  */
    /* F: cudaMemcpy */

    int block = 256;
    int grid  = /* G */;
    stencil1D<<<grid, block>>>(/* H */);

    /* I: cudaMemcpy */
    /* J: cudaFree */
}
```

(c) We want to process an array in chunks using CUDA. For each chunk, the code issues async H2D $\rightarrow$ kernel $\rightarrow$ async D2H in a stream `s[i]` so that operations for different chunks can overlap. Find and correct the errors in the following code.

```
const int nStreams = 4;
cudaStream_t s[nStreams];
for (int i = 0; i < nStreams; ++i) cudaStreamCreate(&s[i]);

size_t streamBytes = streamSize * sizeof(float);
for (int i = 0; i < nStreams; ++i) {
    int off = i * streamSize;

    cudaMemcpyAsync(&dA[off], &hA[off], streamBytes,
                    cudaMemcpyHostToDevice);

    kernel<<<streamSize/blockSize, blockSize, 0>>>(dA, off);

    cudaMemcpyAsync(&hA[off], &dA[off], streamBytes,
                    cudaMemcpyDeviceToHost);
}

std::cout<<"First element of the solution:  "<< hA[0]<< std::endl;

for (int i = 0; i < nStreams; ++i) cudaStreamDestroy(s[i]);
```

Answers

(a)

(1) F — Registers are still faster; shared < global but > registers in latency.

(2) T — A block-wide barrier ensures writes are visible before dependent reads.

(3) F — Shared memory is *per block* and its lifetime is that block's execution.

(4) T — High shared memory usage can throttle concurrent blocks per SM, cutting occupancy.

(b)

```
__global__ void stencil1D(const float* in, float* out, int n,
                          float c0, float c1, float c2) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) {
        if (i >= 1 && i < n - 1) {
            out[i] = c0 * in[i - 1] + c1 * in[i] + c2 * in[i + 1];
        } else {
            // boundaries: copy-through
            out[i] = in[i];
        }
    }
}

void runStencil(const float* hIn, float* hOut, int n,
                float c0, float c1, float c2) {
    size_t bytes = n * sizeof(float);
    float *dIn = nullptr, *dOut = nullptr;

    cudaMalloc(&dIn,  bytes);
    cudaMalloc(&dOut, bytes);

    cudaMemcpy(dIn, hIn, bytes, cudaMemcpyHostToDevice);
```

```
    int block = 256;
    int grid  = (n + block - 1) / block;
    stencil1D<<<grid, block>>>(dIn, dOut, n, c0, c1, c2);

    cudaMemcpy(hOut, dOut, bytes, cudaMemcpyDeviceToHost);

    cudaFree(dIn);
    cudaFree(dOut);
}
```

Shared memory implementation in two different ways: 1)

```
__global__ void stencil1D_shared(const float* __restrict__ in,
                                 float* __restrict__ out,
                                 int n, float c0, float c1, float c2) {
    extern __shared__ float s[]; // size = blockDim.x + 2
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    int tx  = threadIdx.x + 1; // +1 for left halo

    if (gid < n) {
        // center element
        s[tx] = in[gid];

        // left halo (thread 0 loads)
        if (threadIdx.x == 0) {
            int gl = (gid > 0) ? gid - 1 : gid; // clamp at 0
            s[0] = in[gl];
        }
        // right halo (last thread or if gid is last element)
        if (threadIdx.x == blockDim.x - 1 || gid == n - 1) {
            int gr = (gid < n - 1) ? gid + 1 : gid; // clamp at n-1
            s[tx + 1] = in[gr];
        }

        __syncthreads();

        // stencil / boundary
        if (gid == 0 || gid == n - 1) {
            out[gid] = in[gid]; // copy-through on boundaries
        } else {
            out[gid] = c0 * s[tx - 1] + c1 * s[tx] + c2 * s[tx + 1];
        }
    }
}

void runStencilShared(const float* hIn, float* hOut, int n,
                      float c0, float c1, float c2) {
    size_t bytes = n * sizeof(float);
    float *dIn = nullptr, *dOut = nullptr;
    cudaMalloc(&dIn,  bytes);
    cudaMalloc(&dOut, bytes);
    cudaMemcpy(dIn, hIn, bytes, cudaMemcpyHostToDevice);

    int block = 256;
    int grid  = (n + block - 1) / block;
    size_t shmemBytes = (block + 2) * sizeof(float); // tile + 2 halos

    stencil1D_shared<<<grid, block, shmemBytes>>>(dIn, dOut, n, c0, c1, c2);

    cudaMemcpy(hOut, dOut, bytes, cudaMemcpyDeviceToHost);
    cudaFree(dIn);
    cudaFree(dOut);
```

```
}
```

2)

```
#ifndef MAX_BLOCK
#define MAX_BLOCK 1024
#endif

__global__ void stencil1D_shared_static(const float* __restrict__ in,
                                        float* __restrict__ out,
                                        int n, float c0, float c1, float c2) {
    __shared__ float s[MAX_BLOCK];  // static shared memory
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    int tx  = threadIdx.x;

    if (blockDim.x > MAX_BLOCK) return;           // safety guard
    if (gid < n) s[tx] = in[gid];
    __syncthreads();

    if (gid >= n) return;
    if (gid == 0 || gid == n - 1) { out[gid] = in[gid]; return; }

    float left  = (tx > 0) ? s[tx - 1] : in[gid - 1];
    float right = (tx + 1 < blockDim.x && gid + 1 < n) ? s[tx + 1] : in[gid + 1];

    out[gid] = c0 * left + c1 * s[tx] + c2 * right;
}
```

(c)

```
const int nStreams = 4;
cudaStream_t s[nStreams];
for (int i = 0; i < nStreams; ++i) cudaStreamCreate(&s[i]);


size_t streamBytes = streamSize * sizeof(float);
for (int i = 0; i < nStreams; ++i) {
    int off = i * streamSize;

    cudaMemcpyAsync(&dA[off], &hA[off], streamBytes,
                    cudaMemcpyHostToDevice, s[i]);

    kernel<<<(streamSize + blockSize - 1)/blockSize, blockSize, 0, s[i]>>>(dA, off);

    cudaMemcpyAsync(&hA[off], &dA[off], streamBytes,
                    cudaMemcpyDeviceToHost, s[i]);
}

// synchronize
cudaDeviceSynchronize();
std::cout<<"first element of the solutions"<< hA[0]<< std::endl;
for (int i = 0; i < nStreams; ++i) cudaStreamDestroy(s[i]);
```