Solutions for the Exam in DAT400 (Chalmers) and DIT430 (GU) High Performance Parallel Programming, Wednesday, October 30th, 2019, 14:00h - 18:00h

Problem 1 (8 points)

The parallelization of a sequential program consists of three steps: task decomposition, scheduling and mapping. In this problem we ask you to describe, and to reason about the scheduling and mapping steps.

Tasks:

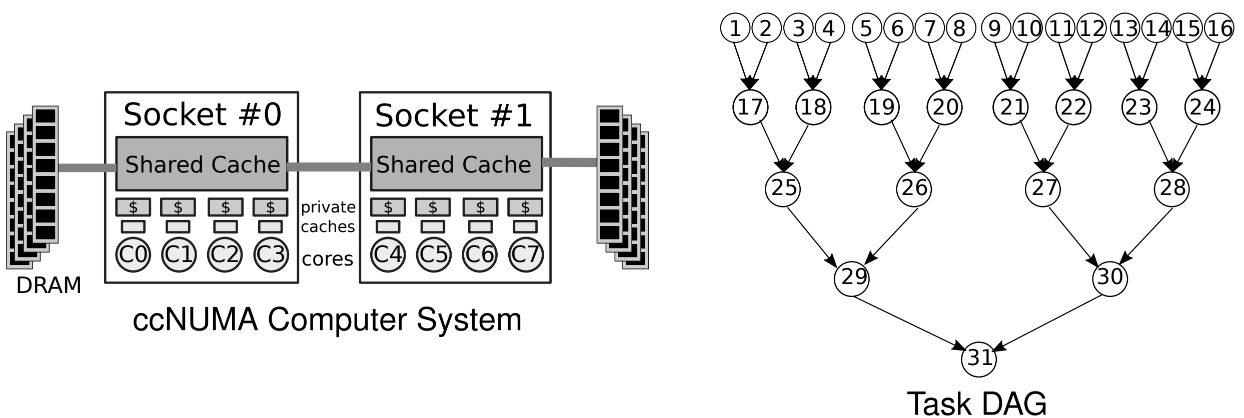
(a) Describe in 3-5 sentences the differences between static and dynamic scheduling.

(b) Write a loop (pseudo-code is acceptable) in which dynamic scheduling should be used instead of static scheduling. Justify your answer.

(c) Describe in 3-5 sentences the main goals of the scheduling and mapping steps.

(d) Consider the (ccNUMA) computer system shown below consisting of two sockets, each with four cores per socket. Consider also the below DAG (Directed Acyclic Graph) which describes the tasks of the application and shows how tasks communicate data. Your task is to come up with a good schedule and a good mapping of this DAG on the aforementioned computer system. Which tasks will be executed by each core? Briefly motivate your choice. Consider the following assumptions:

- The initial location of the data is in the DRAM of Socket #0.
- All tasks perform the same operation on the same amount of data.
- Task execution time is dominated by data movement.
- Data communicated between sockets is 10× slower than data communicated between cores on the same socket.



ccNUMA Computer System

Task DAG

Answer to Problem 1:

Part (a) *Scheduling* refers to the assignment of tasks to processes or threads. There are two types of categories:

- **Static scheduling** : the assignment is done in the initialization phase at program start. This usually means that the programmer or compiler determines the schedule before execution
- **Dynamic scheduling** : the assignment is done during program execution. This usually means that the runtime system performs scheduling decisions depending on the state of execution.

Part (b) Dynamic scheduling is the preferred option for loops in which each iteration has a different duration. A simple example would be a double nested loop in which the inner loop has a variable number of iterations.

```
for(int i = 0; i < N; i++)   // executes N iterations
    for(int j = 0; j < i; j++) // executes i iterations
        c[j] = a[j] + b[j];
```

Part (c) As mentioned above, *scheduling* refers to the assignment of tasks to processes or threads. The main goal of scheduling is to achieve good load balancing, i.e. each process or thread should execute approx. the same number of computations.
Mapping, on the other hand, refers to the process of assigning threads (or processes) to actual physical processors. The main goal of mapping is to get equal utilization of processors or cores and keep communication between these processors as small as possible.

Part (d) Consider the following schedule and mapping of tasks to cores

```
C0 : Tasks |  1|  2|  3|  4|17|18|25|29|31|
C1 : Tasks |  5|  6|  7|  8|19|20|26|--|--|
C2 : Tasks |  9|10|11|12|21|22|27|30|--|
C3 : Tasks |13|14|15|16|23|24|28|--|--|
C4 : Tasks |--|--|--|--|--|--|--|--|--|
C5 : Tasks |--|--|--|--|--|--|--|--|--|
C6 : Tasks |--|--|--|--|--|--|--|--|--|
C7 : Tasks |--|--|--|--|--|--|--|--|--|
```

Rationale: The goal of scheduling is to minimize the number of steps. Considering only load balancing constraints, the best schedule would be one that makes use of all the cores in the system. However, because of the high communication cost between sockets and the fact that the execution time is fully determined by data movement, only a single socket should be used in this example.

The above schedule minimizes the number of steps done by any processor on a four-core socket. As long as there are enough ready tasks, they are distributed to processors in a balanced way. Out of the last three tasks (29, 30, 31), two of them (29,30) can be executed in parallel, hence they should be mapped to two different cores, e.g. Core 0 and Core 2. Note that any combination of cores could be used. It is also not relevant which core executes task 31.
The resulting execution time would be 9 time units. Since data communicated between sockets is $10\times$ slower, it is easy to see that no schedule making use of both sockets could be faster than this.

Problem 2 (6p)

Assume we want to parallelize the following loop on a system with four (4) processors.

```
for(int i = 0; i < 32; i++)
  c[i] = a[i] + b[i];
```

Tasks:

(a) Describe how many tasks the following scheduling methods will generate. Describe also what is the number of iterations performed by each task:

1. Static schedule
2. Dynamic schedule: "self-scheduling"
3. Dynamic schedule: "chunk scheduling", with chunk size 5
4. Guided schedule with minimum chunk size of 1: "GSS(1)"

(b) Assume that each iteration takes 1.0 time units to complete. On the given system with 4 processors, how many time units will it take to complete each loop considering the four scheduling schemes listed above?

(c) Now consider only the dynamic schedulers 2 and 3. Assuming that the dispatch overhead for one iteration is 0.2 time units, and the dispatch overhead for scheduling a chunk of size 5 is 0.8 time units, which of the two scheduling policies will execute faster? Justify your answer.

Answer to Problem 2:

Part (a)

1. 4 tasks, each 8 iterations
2. 32 tasks, each 1 iteration
3. total 7 tasks: 6 tasks with 5 iterations each, and one task with 2 iterations
4. 10 tasks with iteration counts: 8, 6, 5, 4, 3, 2, 1, 1, 1, 1

Part (b)

1. 8.0 time units
2. 8.0 time units
3. 10.0 time units
4. 8.0 time units

Part (c)

2. $T_{\text{exec}} = 8 \times (1.0 + 0.2) = 9.6$ time units
3. $T_{\text{exec}} = 2 \times (5.0 + 0.8) = 11.6$ time units

Although executing a chunk of five iterations has comparatively lower overhead than executing individual iterations (20% lower overhead), the load imbalance introduced by Scheduler 3. still leads to overall worse execution time.

Problem 3 (6p)

Libraries that support parallel programming implement a particular execution model, with support from the operating system. In this problem, we ask you to describe differences between different options.

Tasks:

(a) Describe in 3-5 sentences the main differences between processes and threads. According to the type of information exchange, when is each of these choices most suitable?

(b) There are two types of thread implementations: kernel threads and user-level threads. Describe in 3-5 sentences the main pros and cons of each type.

(c) The Linux pthreads implementation (NPTL) follows an execution model known as 1:1. How is this model implemented? How are threads scheduled in this case?

Answer to Problem 3:

Part (a) A process comprises an executable program and the necessary information for execution. Each process has its own address space, hence exchanging data between processes needs explicit communication such as sockets. Therefore, processes are the preferred way to implement information exchange in distributed address spaces.
Each process may consist of multiple independent control flows called threads. The threads of one process share the address space of the process, hence exchanging data between threads can be done quickly via via memory. Therefore, threads are required to implement information exchanged in shared address spaces.

Part (b)

User-level threads are provided and managed by a *thread library* without specific support of the operating system:

- Pro: (a) simpler OS (does not require specific thread support), (b) thread switching is very fast
- Con: (a) since OS has no knowledge of thread all user-level threads are mapped to same processor, (b) full process is context switched on IO blocking operation (e.g. `read()`)

Kernel threads are provided and scheduled by the operating system:

- Pro: (a) Allows efficient use of cores in multicore system (shared cache + parallelism), (b) lower overheads compared to processes
- Con: (a) Higher overhead compared to user-level threads, (b) requires OS support

Part (c) In the 1:1 model, user-level threads are mapped to a kernel-level threads. As a consequence, it is the scheduler of the operating system that selects which (user/kernel) threads are executed and how they are mapped to the processors. This scheme enables parallel processing within a process.

Problem 4 (8p)

The following code is used to approximate the solution of the Poisson problem $\nabla^2 u = f$ on a square matrix:

```
void compute(float** anew, float** aold, int j_start, int j_end,
             int i_start, int i_end){
    for(int j=j_start; j<j_end; ++j) {
      for(int i=i_start; i<i_end; ++i)
        anew[i][j] = (aold[i][j]+
                      aold[i-1][j] + aold[i+1][j] +
                      aold[i][j-1] + aold[i][j+1]) / 5.0;
    }
}
int main() {
  ...
  for (int time = 0; time < MAX_TIME; time++){
    compute(pnew, pold, j_start, j_end, i_start, i_end);
    pold = pnew; // update pold for next iteration
  }
  ...
}
```

This is a 2D 5-point stencil computation. The size of the matrix is N^2 and it is divided among *P* processes as shown in the figure below.

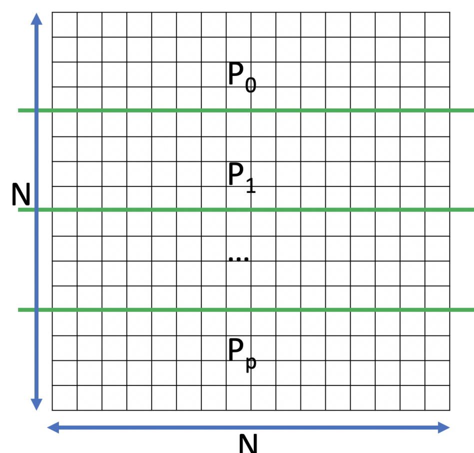
Tasks:

(a) What is the size of local (per-process) matrix as a function of *N* and *P*?

(b) How many matrix elements are exchanged in each iteration? Provide the solution as a function of *N* and *P*.

(c) Add the necessary MPI calls to the above `main()` function (pseudo-code is ok) to implement the information exchange in a distributed memory environment.

(d) Is it possible to leverage non-blocking communication to improve execution time? Justify your answer in 3-5 sentences.



Domain decomposition among *P* processes

Answer to Problem 4:

(a) Size of local matrix = $[\frac{N}{P}+2, N]$, since upper and lower boundary need to be exchanged. If we do not consider boundary rows then Size of local matrix = $[\frac{N}{P}, N]$ but then implementation should take care of extra boundary rows.

(b) 2N elements need to be exchanged per process for exchanging first and last row of their local matrix. However, first and last process will exchange only one row.

(c) The code below shows the solution using MPI. For the purpose of exam correction, explaining a high-level pseudo-code will be considered enough as long as the main calls to MPI routines and the exchange buffers are correctly identified.

(d) Each process can use non-blocking calls for the exchange and wait before starting computation on receiving data. Thus overlapping computation over communication can improve execution time.

```
main(){
    // Read local data for each process
    // start computation
    for (int time = 0; time < MAX_TIME; time++){
        j_start = 0; i_star = N;
        j_end = N; i_end = N/p;
        compute(pnew, local_A, j_start, j_end, i_start, i_end);
        // Assign p_new to local_A starting from local_A[1][0]
        // because first row is boundary-row
        local_A = pnew;
        // Exchange
        int upper, lower;
        if(rank==0) upper = p-1;
        else upper = rank-1;
        if (rank == p-1) lower = 0;
        else lower = rank +1;

        //send up then recieve from below
        MPI_Send(&local_A[0], N*sizeof(float),
            MPI_FLOAT, upper, 1, MPI_COMM_WORLD);
        MPI_Recv(&local_A[(N/p)+N], N*sizeof(float),
            MPI_FLOAT, lower, 1, MPI_COMM_WORLD, status);

        //send down then recieve from above
        MPI_Send(&local_A[(N/p)+N], N*sizeof(float),
            MPI_FLOAT, lower, 2, MPI_COMM_WORLD);
        MPI_Recv(&local_A[0], N*sizeof(float),
            MPI_FLOAT, upper, 2, MPI_COMM_WORLD, status);
  }
}
```

Problem 5 (6p)

The main goal of parallelization is to improve the performance of an application. Unlike single-processes execution, a parallel computation (such as BSP) consists of (1) local computation, (2) waiting time, (3) data exchange, and (4) synchronization time.

Tasks:

(a) Explain what are these four components. Assume an SPMD computation to justify your answer.

(b) Two common metrics to characterize parallel execution are cost and efficiency. What do these two metrics describe?

(c) Briefly describe in 3-5 sentences how Amdahl's law is related to cost and efficiency.

Answer to Problem 5:

Part (a)

- **Local computation**: this is the time of each process while executing application code
- **Data exchange**: this is time spent by each process in communication calls. This includes point-to-point calls such as as send/recv and collective calls such broadcast or accumulation.
- **Synchronization time**: this is time spent by each process while waiting in barriers between computation stages or other synchronization such as critical sections or condition variables.
- **Waiting time**: This is extra time in which processors need to wait due to load imbalance, i.e. different processes being occupied with work for different periods of time.

Part (b) The *Cost* metric describes the total amount of work performed by all processes. It is calculated as $C_p(n) = p \times T_p(n)$, where p = number of processors, n = is the problem size and $T_p(n)$ = parallel runtime.

Efficiency describes how efficiently a program is parallelized. It is calculated as $E_p(n) = \frac{T^*(n)}{C_p(n)}$. Note that for a fair comparison $T^*(n)$ (time on best sequential algorithm) should be considered instead of $T_1(n)$.

Part (c) Amdahl's law describes the potential speed-up that can be achieved by a program as a function of the program part that cannot be improved (e.g. parallelized). A program that is completely sequential cannot be parallelized, hence the efficiency will be $E_p(n) \leq \frac{1}{p}$. Similarly such a program will have very high cost: $C_p(n) = p \times T_1(n)$. On the other hand, a program that can be completely parallelized (i.e. linear speed-up) will have an efficiency of $\frac{T^*(n)}{T_1(n)}$, which should hopefully be close to 1.0.

Problem 6 (8p)

Below is a minimal 2D N-Body code that calculates the force and potentional in free space due to all particles using the equation (for potential) $P_i = \sum_{j=1}^{N} \frac{m_i m_j \hat{r_{ij}}}{\sqrt{\Delta r^2}}$ and assumes a unit mass. The outer loop is called the "target" loop, whereas the inner loop is called the "source" loop.

```
typedef struct {
float x, y;   // position
float vx, vy; // velocity
} Body;
void n_body(Body *p, float dt, int N) {
  for (int i = 0; i < N; i++) {   // target loop
    float Fx = 0.0f;
    float Fy = 0.0f;
    for (int j = 0; j < N; j++) { // source loop
      float dx = p[j].x - p[i].x;
      float dy = p[j].y - p[i].y;
      float invDist3 = get_invDist3(dx, dy);
      Fx += dx * invDist3;
      Fy += dy * invDist3;
    }
    // update velocities
    p[i].vx += dt*Fx; p[i].vy += dt*Fy;
  }
}
```

Tasks:

(a) How many floating point operations will this code make as a factor of N (the problem size)? What kind/level of parallelism can this code exploit?

(b) Using OpenMP, provide a short snippet of a parallel implementation of the source (inner) loop. Explain - in one line per each - the semantics of the used OpenMP directives/constructs.

(c) Parallelize the target loop. If nested parallelism is **not** to be used, which do you think will be the best loop to parallelize in terms of performance? Explain why.

(d) Assume you run on a dual-socket, 8 core/socket, 4-way multithreaded (SMT) totaling 16 cores and 64 threads. Modify your code in (c) such that it binds an OpenMP thread to a hardware thread and uses 32 threads (2 threads per core).

Answer to Problem 6:

Part (a) The floating point operation count: An answer can be:
OpCount = $N^2 \times (1\text{subtract} + 1\text{subtract} + \text{getInvDist3} + 2\text{multiply} + 2\text{add}) + N \times (2\text{add} + 2\text{multiply}) = (6 + \text{getInvDist3})N^2 + 4N$
The answer is also valid if the MultiplyAdd is considered as 1 operation (e.g. Fx += dx * invDist3), which yields: OpCount = $N^2 \times (1\text{subtract} + 1\text{subtract} + \text{getInvDist3} + 2\text{multiplyAdd}) + N \times (2\text{multiplyAdd}) = (4 + \text{getInvDist3})N^2 + 2N$
The level of parallelism: Data or Loop parallelism

Part (b) Code:

```
#pragma omp parallel for reduction(+:Fx) reduction(+:Fy)
    for (int j = 0; j < N; j++)
```

OR

```
#pragma omp parallel for reduction(+:Fx,Fy)
    for (int j = 0; j < N; j++)
```

Semantics:

- `parallel`: create/fork the threads that compose a parallel region.
- `for`: partition and distribute the subsequent `for` loop over the threads in the parallel region.
- `reduction`: apply parallel sum reduction on the variable

Another answer:
Code:

```
#pragma omp parallel for
for (int j = 0; j < N; j++)
{
    ...
#pragma omp critical
//OR
#pragma omp atomic
{
  Fx += dx * invDist3;
  Fy += dy * invDist3;
}
}
```

Semantics:

- `parallel`: create/fork the threads that compose a parallel region.
- `for`: partition and distribute the subsequent for loop over the threads in the parallel region.
- `critical`: create a critical section in the enclosed region.
- `atomic`: atomic update for shared memory locations `Fx` and `Fy`.

Part (c) Parallelize the target loop:

```
#pragma omp parallel for
    for (int i = 0; i < N; i++)
```

Best loop to parallelize: It is the outer/target loop. The reasons include the following:

- If the inner is parallelized, threads will be created and joined N times, as opposed to 1 time in the outer loop parallelism. This entails a big overhead.
- As noted in Part (b), the inner loop updates a shared variable in parallel, which requires a level of synchronization (parallel reduction, critical section or atomic update).

Part (d)

```
OMP_PLACES=threads OR OMP_PLACES={0:1}:32:2 OR OMP_PLACES={0}:32:2
#pragma omp parallel for proc_bind(spread) num_threads(32)
    for (int i = 0; i < N; i++)
```

OR

```
OMP_PLACES=threads OR OMP_PLACES={0:1}:32:2 OR OMP_PLACES={0}:32:2
OMP_NUM_THREADS = 32
#pragma omp parallel for proc_bind(spread)
    for (int i = 0; i < N; i++)
```

Problem 7 (10p)

The MPI interface defines several communication modes for point-to-point communication operations (send/receive). The two main modes are the synchronous mode and the buffered (asynchronous) mode. In additon, MPI defines blocking and non-blocking communication. These concepts have important implications on synchronization and performance.

Assume the following execution of two processes in which blocking routines for send/receive are identified under the generic name `Send(send_buffer,destination_rank)` and `Recv(receive_buffer,source_rank)`. The downwards timeline describes the order in which processes perform calls in a particular execution that is to be considered in this problem. '0' and '1' identify the MPI ranks of the two processes, and DA, DB, DC are addresses of identical data structures.

Timestep	Process 0	Process 1
1	`Send(&DA,1)`	
2	`do_work1(&DB)`	
3	`do_work2(&DA)`	
4	`MPI_Barrier(MPI_COMM_WORLD)`	`Recv(&DC,0)`
5		`MPI_Barrier(MPI_COMM_WORLD)`

Tasks:

(a) Describe in 3-5 sentences the difference between synchronous and asynchronous communication.

(b) How would the usage of synchronous and buffered mode impact the above execution? What possible timelines (i.e. orders among calls) will result?

(c) Describe in 5-7 sentences: (1) the difference between blocking and non-blocking communication, and (2) the purpose of non-blocking communication.

(d) Modify the example above by adding MPI calls to achieve better performance via the usage of non-blocking primitives.

Answer to Problem 7:

Part (a) The concepts of synchronous and asynchronous communication define how communication behaves from a global (multi-process) view.

- Communication is called **synchronous** when the communication operation between sender and receiver does not complete until both parties have respectively started their send and receive operation.
- Communication is called **asynchronous** when the sender can execute its communication operation without any coordination with the receiving process.

Part (b)

Synchronous mode: In synchronous mode, the send and receive calls will need to perform a handshake before any of the two can proceed.

Timestep	Process 0	Process 1
1	`Send(&DA,1)`	
2		
3		
4	(synchronization happens)	`Recv(&DC,0)`
5	`do_work1(&DB)`	`MPI_Barrier(MPI_COMM_WORLD)`
6	`do_work2(&DA)`	
7	`MPI_Barrier(MPI_COMM_WORLD)`	(barrier synchronization)

Buffered mode: In buffered mode, however, no handshake is necessary. The same execution as in the example will result.

Timestep	Process 0	Process 1
1	`Send(&DA,1)`	
2	`do_work1(&DB)`	
3	`do_work2(&DA)`	
4	`MPI_Barrier(MPI_COMM_WORLD)`	`Recv(&DC,0)`
5		`MPI_Barrier(MPI_COMM_WORLD)`

Part (c)

Part (1): Blocking and non-blocking are concepts that describe how local per-process resources are managed in MPI. When an operation is blocking, control returns to the calling process only after all resources (e.g. buffers) can be safely reused. When an operation is nonblocking, control returns to the calling process immediately, even if the operation has not completed yet. In this case it is not safe to reuse resources.
Part (2): The purpose of non-blocking communication is to overlap communication with independent computations. This will achieve higher utilization of processors, lead to better efficiency and hopefully decrease execution time.

Part (d) Non-blocking communication can be used to allow Process 0 perform the call to `do_work1(&DB)` in parallel with the first communication operation. However, the call to `do_work2(&DA)` must wait, since it makes use of the same buffer as the send. Hence it is necessary to insert a `MPI_Wait(request)` call to ensure that the buffer can be safely reused. In this code sequence there is way to improve the performance of the receiver via nonblocking receive calls (there is no independent work that can be overlapped).

Timestep	Process 0	Process 1
1	`MPI_Isend(&DA,1,request)`	
2	`do_work1(&DB)`	
3	`MPI_Wait(request)`	
4	`do_work2(&DA)`	
5	`MPI_Barrier(MPI_COMM_WORLD)`	`Recv(&DC,0)`
6		`MPI_Barrier(MPI_COMM_WORLD)`

Problem 8 (8p)

The code below is an outline of a device (GPU) function that implements a block-tiled N-Body kernel. It basically processes the inner loop (as of the code in Problem 6) in a tiled fashion. Note that `float2` is a convenience CUDA aligned data type which is essentially a 2 dimensional float.

```
__global__
void n_body(float2 *p, float2 *v, float dt, int n) {
  int i = /*MISSING 1*/
  if (i < n) {
    float Fx = 0.0f; float Fy = 0.0f;
    for (int tile = 0; tile < /*MISSING 2*/; tile++) {
      __shared__ float2 spos[BLOCK_SIZE];
      float2 tpos = p[/*MISSING 3*/];
      spos[/*MISSING 4*/] = make_float2(tpos.x, tpos.y);
      /*MISSING 5*/
      for (int j = 0; j < BLOCK_SIZE; j++) {
        float dx = spos[j].x - p[i].x;
        float dy = spos[j].y - p[i].y;
        // force calculation same as before
      }
      /*MISSING 6*/
    }
    v[i].x += dt*Fx; v[i].y += dt*Fy;
  }
}
int main() {
  const int nBodies = 10000;
  ...
  int nBlocks = (nBodies + BLOCK_SIZE - 1) / BLOCK_SIZE;
  ...
  n_body<<<nBlocks, BLOCK_SIZE>>>(d_p.pos, d_p.vel, dt, nBodies);
  ...
}
```

Tasks:

(a) Fill in the 6 missing code snippets which do the following, respectively 1) Get the index of the element to be calculated; 2) Number of tiles/blocks; 3) Position of element in global array; 4) Position in block shared memory; 5) Ensure all data is loaded in shared memory; 6) Ensure block data has been calculated.

(b) Assume $N = 10000$. How many bytes will be transferred from host to device and back (just write the equation). If your GPU architecture can run 1024 threads per SM, what would be an efficient choice of shared memory block length for the implementation above (`BLOCK_SIZE` in this case), given that the available shared memory size is 32kB?

(c) In no more than 3 lines, list the performance considerations that you would study before deciding to use the CPU version of this code or offload to the GPU.

Answer to Problem 8:

Part (a)

```
blockDim.x * blockIdx.x + threadIdx.x; // MISSING 1
gridDim.x                              // MISSING 2
tile * blockDim.x + threadIdx.x        // MISSING 3
threadIdx.x                            // MISSING 4
__syncthreads();                       // MISSING 5
__syncthreads();                       // MISSING 6
```

Part (b) Global data transfer size: 2 (considering the HOST to DEVICE and DEVICE to HOST trips) x 4 (float size) x 2 (the velocity/potential vector width) x 10000 (N)

Rationale behind BLOCK_SIZE choice: `BLOCK_SIZE` can take up to 32kB according to the described architecture, however, since each thread handles one element in the block, we test the limit, which is 1024 (the maximum threads per SM) x 4 (bytes in a float) x 2 (vector width) = 8kB. This is still much less than 32kB. However, the most efficient choice is 1024, even though the shared memory can accommodate more data due to the 1024 threads per SM limit.

Part (c) The reasons include, but are not limited to:

- The host-device memory transfer bandwidth.
- The ratio of kernel computations to its global memory access (CGMA).
- The data access pattern (regular or irregular).
- The absolute performance of the kernel on CPU/GPU.
- Whether the kernel is potentially data parallel (loose dependencies across parallel units).