Solutions for the Exam in DAT400 (Chalmers) and DIT431 (GU) High Performance Parallel Programming, Thursday, October 31st, 2024, 8:30h - 12:30h

Problem 1 (16 points)

This problem covers theoretical concepts related to parallel programming models.

(a) The three programming models introduced in DAT400 (OpenMP, MPI, CUDA) support some form of data parallelism. Briefly describe what is data parallelism and what is its main advantage. Next, list one challenge to exploit (i) task parallelism, (ii) pipeline parallelism, and (iii) instruction level parallelism (list one challenge for each type of parallelism).

(b) SIMT (Single Instruction Multiple Threads) is a variation of SPMD (Single Program Multiple Data) used in CUDA GPUs. In up to two sentences each, describe (1) how does CUDA SIMT differ from the SPMD implementations in OpenMP and MPI, and (2) how is SIMT related to SIMD (Single Instruction Multiple Data).

(c) An application is given that is composed of several parallel loops, as shown in the code snippet below. The application is to be run on a multicore system with only private per-core caches. The iterations in the loops have different execution time. Across loops, corresponding iterations process the same data. Initially a static schedule was used to parallelize, but the programmer discovered that it led to load imbalance. To address the problem, a dynamic schedule (`#pragma omp parallel for schedule(dynamic)`) has been used instead. To their surprise, they observe that the total execution time actually degrades. Propose two potential causes for the performance degradation resulting from the dynamic schedule.

```
int i;
DATA_t data[N]; // N is very large
#pragma omp parallel for schedule(dynamic)
for(i = 0; i < N; i++) // loop 1
    func1(&data[i]); // Exec time depends on data[i]

#pragma omp parallel for schedule(dynamic)
for(i = 0; i < N; i++) // loop 2
    func2(&data[i]); // Exec time depends on data[i]

#pragma omp parallel for schedule(dynamic)
for(i = 0; i < N; i++) // loop 3
    func3(&data[i]); // Exec time depends on data[i]

....
```

(d) An HPC company has decided to implement the CUDA programming model using User-Level Threads (ULTs) so that they can run CUDA applications efficiently on a shared memory multiprocessor. List one general challenge of working with ULTs. Next, list two specific challenges of implementing CUDA using ULTs. Assume the N:M model of thread execution.

(e) Assume an application that adds three arrays `A[]`, `B[]`, `C[]` and stores the result in a fourth array `D[]`, i.e. `D[] = A[] + B[] + C[]`. The input data is initially located in disk, in two possible formats described by the two C structs shown in Figure 1 and Figure 2.

The application is to be run on a distributed memory multiprocessor system with four multicore nodes. Each multicore node has its own local memory, and the four nodes are interconnected by a message passing network. To execute the addition, the data

```
struct{
    double A[SIZE];
    double B[SIZE];
    double C[SIZE];
    double D[SIZE];
} data;
```

Figure 1: Structure of Arrays (SoA)

```
struct {
    double A;
    double B;
    double C;
    double D;
    } data[SIZE];
```

Figure 2: Array of Structures (AoS)

needs to be loaded into each node's local memory. For each of the two data arrangements (AoS and SoA), assess the suitability of the following data distribution methods: blockwise, cyclic, or block-cyclic. Consider that data can only be distributed at a granularity of a full C `struct`, both across nodes and within nodes (i.e. across the cores of the multicore node).

Answers:

(a) Data parallelism means applying the same operation on different data. Data parallelism is easy to reason about. Task parallelism requires the programmer to identify independent program components and to worry about dependencies and synchronization. Pipeline parallelism requires expressing computations as sequences of operations applied to different data, and to worry about the latency of each stage. Exploiting Instruction level parallelism requires expensive hardware support, including register renaming, and tracking register dependencies.

(b) SIMT refers to the front-end of CUDA GPU Streaming Multiprocessors (SM) which, given the thread-centric CUDA programming model (CUDA kernels), converts bunches of 32 consecutive threads (called *warp*) into a single instruction that is executed on different data. This back-end organization is similar to the SIMD units featured in vector processors. Compared to SPMD, SIMT enforces a stronger synchronization across consecutive threads, while SPMD threads in OpenMP/MPI execute asynchronously.

(c) Two potential reasons could be (1) lack of data locality resulting from the dynamic schedule, and (2) high overheads due to the cost of dynamic scheduling (for example, using a shared queue) which by default uses a chunk size of (1)

(d) Two general challenges: 1) Without KLTs, ULTs alone cannot offer parallelism. 2) Progress guarantees are low, threads need to voluntarily context switch, and in the event of a blocking event (e.g. file system read), all ULTs are blocked. To achieve parallelism, a combination of ULTs and KLTs is necessary. This is called the N:M model.

CUDA challenges, assuming N:M model: (1) CUDA threads are supported by a hardware scheduler, it will be necessary to develop a software level CUDA scheduler. (2) Very fine-grained threads are supported by CUDA (thanks to the hardware scheduler). To achieve performance, it will be necessary to increase the number of tasks executed by each ULT. One potential way to achieve this is to execute a full thread block on each ULT, instead of executing individual CUDA threads.

(e) Structure of arrays (SoA): Since the whole data structure is defined a single C struct that cannot be subdivided, there is no data distribution that would work well in this case. For all distributions, the full data would end up on the first memory, and the whole addition would be done by a single core, which is clearly suboptimal.

Array of structures (AoS): The array of structures organization implements a cyclic arrangement which will lead to good load balancing and only local operations for all three of the data distributions. Within a node, the data distribution can potentially lead to false sharing. Since each struct element consists of 32 bytes (= 4x doubles), two elements can fit a 64-byte cache line. Hence a blockwise distribution, or a block-cyclic distribution with granularity of two elements should be chosen to avoid false sharing. A cyclic distribution will lead to different cores updating different parts of the same cache line, leading to unnecessary traffic.

Problem 2 (7 points)

Consider the following code that computes the Floyd-Warshall algorithm, which finds the shortest paths between all pairs of vertices in a weighted graph represented by an adjacency matrix `dist`.

```
void floyd_warshall(int dist[N][N]) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            for (int i = 0; i < N; i++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}
```

Tasks:

(a) Discuss cache performance issues associated with the current implementation of the Floyd-Warshall algorithm, focusing on memory access patterns and their impact on execution time.

(b) One optimization to consider is loop unrolling, unroll the innermost loop two times and write down the pseudo-code.

(c) List at least two loop optimizations that can be performed on this code, other than loop unrolling, and specify their advantages in a single sentence. Apply the optimizations in the simplest possible way and write down the resulting pseudo-code.

Answers

(a) Since `dist` is a 2D array stored in row-major order, iterating over `i` and then `j` while `k` is fixed results in the potentially poor locality of reference for cache usage. Accessing dist[i][k] and dist[k][j] may cause the cache lines to be less effectively utilized. The execution time of the program may decrease due to frequent cache misses.

(b) The following solution assumes N is even:

```
void floyd_warshall(int dist[N][N]) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            for (int i = 0; i < N; i+=2) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    }
                if (dist[i+1][k] + dist[k][j] < dist[i+1][j]) {
                    dist[i+1][j] = dist[i+1][k] + dist[k][j];
  }
            }
        }
    }
}
```

(c) Loop interchange:

```
void floyd_warshall(int dist[N][N]) {
for (int k = 0; k < N; k++) {
    for (int i = 0; i < N; i++) { // Interchanged i and j loops
        for (int j = 0; j < N; j++) {
            if (dist[i][k] + dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
}
```

Loop tiling:

```
#define TILE_SIZE 32
void floyd_warshall(int dist[N][N]) {
for (int k = 0; k < N; k += TILE_SIZE) { // Loop tiling for k
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int kk = k; kk < k + TILE_SIZE && kk < N; kk++) {
                if (dist[i][kk] + dist[kk][j] < dist[i][j]) {
                    dist[i][j] = dist[i][kk] + dist[kk][j];
                }
            }
        }
    }
}
}
```

Problem 3 (11 points)

Consider a scientific computing application that processes a large dataset and consists of both parallelizable and non-parallelizable components.

Tasks:

(a) The application spends 80% of its time performing an algorithm shown below (which is parallelizable) and 20% on sequential data preparation and result analysis. Using Amdahl's law, calculate the theoretical maximum speedup achievable for this program. If using an upgraded hardware improves the algorithm speed by a factor of 6x, what would be the overall speedup according to Amdahl's law?

(b) Cost and efficiency are two key metrics used to assess the performance of parallel programs. The serial execution time of the application is given as Z seconds. A programmer runs the application on an 8-core machine, achieving an execution time of K seconds. Calculate the cost and efficiency based on these values.

Consider the algorithm implementation (Note: a `float` is 4 bytes):

```
1  #define M 10000
2  #define N 10000
3
4  void algorithm(float *A, float *B, float *x, float *y) {
5      for (int i = 0; i < M; i++) {
6          for (int j = 0; j < N; j++) {
7              y[i * N + j] += A[i * N + j] * x[j] + B[j];
8          }
9      }
10 }
```

(c) Calculate the arithmetic intensity of this algorithm implementation.

(d) Assume a multicore processor where each core has a performance of 8 GFLOPS (single precision) and a shared memory bus of 32 GB/s. Using the simplified DRAM roofline model, estimate the execution time if this algorithm runs on 1 core and on 8 cores. According to your results, what are the primary bottlenecks: FLOPS or Memory bandwidth of running on 1 core and 8 cores respectively? Regarding the execution time calculation, you can simply provide the formula.

(e) If the shared memory bus bandwidth is doubled to 64 GB/s, how would this affect the performance of the algorithm on 1 core and 8 cores? Recalculate the execution times and discuss the implications. What strategies can be used to improve the performance based on your analysis of the results?

Answers

(a) Theoretical maximum speedup: Amdahl's law formula: S = 1 / ((1 - p) + (p / s)), where: S = maximum speedup, p = proportion of parallelizable part (80% or 0.8), s = number of processors (approaching infinity for theoretical maximum)

As s approaches infinity, p/s approaches 0, so we get:
S = 1 / (1 - p) = 1 / (1 - 0.8) = 1 / 0.2 = 5

After the hardware upgrade: S = 1 / ((1 - 0.8) + (0.8 / 6)) = 3
Therefore, the overall speedup according to Amdahl's law after the hardware upgrade is 3x.

(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. In this case: Cost = K $\times$ 8.

Efficiency describes how efficiently a program is parallelized. It is calculated as $E_p(n) = \frac{T_1(n)}{C_p(n)}$. In this case: Efficiency = J / (8K).

(c) FLOPS = (1 multiplication + 2 additions) $\times$ M $\times$ N = 3 $\times$ M $\times$ N
Memory access = 2 $\times$ M $\times$ N $\times$ 4 + N $\times$ 2 $\times$ 4 bytes = 8 $\times$ N $\times$ (M + 1) $\simeq$ 8 $\times$ M $\times$ N
Arithmetic intensity = 0.375 FLOPS/byte

(d) Execution time on 1 core:
min(0.375 $\times$ 32GB/s, 8GFLOPS) = 8 GFLOPS. The primary bottleneck is compute bandwidth.
Time = (M $\times$ N $\times$ 3 FLOPS) / 8 GFLOPS = 0.0375 sec

Execution time on 8 core:
min(0.375 $\times$ 32GB/s, 8 $\times$ 8 GFLOPS) = 12 GFLOPS. The primary bottleneck is now memory bandwidth. With 8 cores, the performance is now limited by the shared memory bus bandwidth of 32 GB/s.

(e) Execution time on 1 core:
min(0.375 $\times$ 64GB/s, 8GFLOPS) = 8 GFLOPS. The primary bottleneck is the FLOPS (compute-bound).
Time = (M $\times$ N $\times$ 3 FLOPS) / 8 GFLOPS = 0.0375 sec

Execution time on 8 core:
min(0.375 $\times$ 64GB/s, 8 $\times$ 8 GFLOPS) = 24 GFLOPS. The primary bottleneck becomes memory bandwidth. The operation is memory-bound.
Time = (M $\times$ N $\times$ 3 FLOPS) / 24 GFLOPS = 0.0125 sec

To improve performance, it is better to exploit a hardware with higher memory bandwidth and / or focus on optimizing memory access patterns or reducing data movement, rather than adding more computational resources.

Problem 4 (20 points)

This problem deals with parallelization of the "power iteration", an algorithm to find the eigenvalues and eigenvector of a diagonalizable square (NxN) matrix A. A sequential C code is shown below. The code first multiplies `matrix[N][N]` with `vector[N]` (matrix-vector product). The product `result[N]` is then normalized and copied back into `vector[N]`, to be used for the next iteration, unless the convergence check is satisfied or a maximum number of iterations is reached. The results are returned in `eigenvector[N]` and `eigenvalue`.

The goal of the problem is to devise MPI and CUDA versions of the program. The MPI and CUDA parts are independent of each other, and can be solved independently. To simplify the parallelization across MPI nodes and CUDA device, we will accelerate only on the matrix-vector product. All other operations are to be performed by one processor (either the *rank 0* in MPI, or the *host* in CUDA).

```
1  void power_iteration(double matrix[N][N], double vector[N], double *eigenvalue, double
       eigenvector[N]) {
2      double result[N], norm;
3      int iter, i, j;
4
5      for (i = 0; i < N; i++) {    // Initialize the vector with arbitrary values
6          vector[i] = 1.0;
7      }
8
9      for (iter = 0; iter < MAX_ITER; iter++) {
10         // Matrix-vector multiplication
11         for (i = 0; i < N; i++) {
12             result[i] = 0.0;
13             for (j = 0; j < N; j++) {
14                 result[i] += matrix[i][j] * vector[j];
15             }
16         }
17
18         // Calculate the norm of the result vector
19         norm = 0.0;
20         for (i = 0; i < N; i++) {
21             norm += result[i] * result[i];
22         }
23         norm = sqrt(norm);
24
25         // Normalize the result vector and copy it back to the input vector
26         for (i = 0; i < N; i++) {
27             vector[i] = result[i] / norm;
28         }
29
30         // Check for convergence
31         if (fabs(norm - *eigenvalue) < TOL) {
32             break;
33         }
34
35         *eigenvalue = norm;
36     }
37
38     // Copy the final vector to the eigenvector
39     for (i = 0; i < N; i++) {
40         eigenvector[i] = vector[i];
41     }
42 }
```

When providing your answers, you do not need to rewrite the full code again, but we ask you to clearly specify where in the sequential code your changes should be inserted (i.e. some number between 1 and 42) and what the changes are. For this problem you may provide your answers in pseudocode, but the call names and parameters should be accurate. A list of useful MPI and CUDA calls is shown below the questions.

The first set of questions deals with MPI:

(a) Modify the code with MPI calls to (1) distribute the matrix and vector to all ranks, (2) perform the matrix-vector product on each MPI rank, and (3) return the result. Use blocking point-to-point communication to solve this problem.

(b) Improve your solution to part (a) by using MPI collectives wherever applicable.

(c) An alternative way to achieve performance improvement is to target communication-computation overlap. Propose a code using non-blocking P2P communication to achieve this.

(d) Out of the two code variants developed in parts (b) and (c), which one do you expect to perform best? Briefly motivate why.

The second set of questions deals with CUDA:

(e) Write a CUDA host code that sends the matrix and vector to the CUDA device, executes the kernel and collects the resulting vector. You may assume that N is a multiple of 1024.

(f) Write a CUDA kernel that performs the matrix vector product. Do not use shared memory for this task.

(g) Improve performance by making use of shared memory. To simplify, you may assume that N <= 1024.

(h) Next, we want to improve the CUDA code with the goal of achieving computation-communication overlap. Briefly answer each of the following two questions: (1) What is a CUDA stream? (2) Draw a diagram showing a possible timeline for the overlapped execution, in which the different copies and kernel executions are visible.

(i) Finally, identify two parts of the code developed in part (e) that need to be changed and describe in words what needs to be changed.

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

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Send (const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm)
int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
    comm, MPI_Status *status)
int MPI_Isend (void *buffer, int count, MPI_Datatype type, int dest, int tag, MPI Comm
    comm, MPI_Request *request)
int MPI_Irecv (void *buffer, int count, MPI_Datatype type, int source, int tag, MPI Comm
    comm, MPI_Request *request)
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request,  MPI_Status *status)
int MPI_Barrier( MPI_Comm comm )
int MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm
    communicator)
```

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *
    recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf
    , int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

A subset of CUDA calls that can be useful for this problem are shown below

```
cudaMalloc(void** devPtr, size_t size)
cudaFree(void* devPtr)
cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)
Kernel<<<dimGrid, dimBlock, shared_mem, stream>>>(parameters)
__syncthreads()
cudaStreamCreate(cudaStream_t *)
cudaStreamDestroy(cudaStream_t)
cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind,
    cudaStream_t streamx)
cudaMallocHost(void** devPtr, size_t size)
cudaFreeHost(void* devPtr)
```

Answers

(a)

```
// Change #1 Insert in line 8 (to be performed once before iteration)
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
// rank 0 distributes all the pieces of the matrix
if(rank == 0)
    for(int i = 1; i < comm_size; i++)
        MPI_Send(&matrix[i][0], N * N/comm_size, MPI_DOUBLE, i, 0,
            MPI_COMM_WORLD);
else // if not rank 0
        MPI_Recv(&matrix[0][0], N * N/comm_size, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD, &status);
        // the block of rows can be stored at the starting address.
        // But we need to be aware that these belong to different rows in
            the original matrix
// End of change #1

// Change #2
// Insert in line 11. Needs to be distributed in each iteration
if(rank == 0)
    for(i = 1; i < comm_size; i++)
        MPI_Send(vector, N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
else
    MPI_Recv(vector, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

// For the Matrix-vector multiplication, the only change is in the iteration
     space since we only work on a subset of rows N/comm_size
for (i = 0; i < N/comm_size; i++) {
    result[i] = 0.0;
    for (j = 0; j < N; j++) {
        result[i] += matrix[i][j] * vector[j];
    }
}

// now the result is to be sent back, but be aware that each process only
    has a subset of the result vector
if(rank == 0)
```

```
        for(i = 1; i < comm_size; i++) // start at 1 since the first part is
            already locally available
            MPI_Recv(&result[i/commsize-1], N/commsize, MPI_DOUBLE, i, 0,
                MPI_COMM_WORLD, &status);
    else
        MPI_Send(result, N/comm_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

    // after this, normalization and convergence testing can proceed in the host
    // End of Change #2
```

(b)

```
    // we show only the places where collectives can be used

    // Change #1: Line 8
    // Distribution of the matrix can be done using colletives
    MPI_Scatter(matrix, N, MPI_DOUBLE, r_matrix, N, MPI_DOUBLE, 0,
        MPI_COMM_WORLD);
    // here we make use of a different variable in the receiving processes to
        avoid the problem of sending and receiving the same buffer at the same
        time (in process 0). Using matrix both in the sender and receiver can
        work in MPI, but you need to use a special flag MPI_IN_PLACE which has
        not been described in class. Hence, we will this a correct answer (even
        if it is wrong in practice)
    // End of Change #1

    // Change #2: Line 11 vector needs to be distributed in each iteration
    // this can be done with a broadcast
    MPI_Bcast(vector, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // now we can perform the Matrix-vector multiplication, only for the subset
        of rows
    // a small modification compared to (a): the result is temporarilly placed
        in a new vector r_result[]
    // necessary to avoid an in place Gather below
    for (i = 0; i < N/comm_size; i++) {
        r_result[i] = 0.0;
        for (j = 0; j < N; j++) {
            r_result[i] += matrix[i][j] * vector[j];
        }
    }
    // sending the result back can be achieved with a gather operation from
        r_result[] into result[]
    MPI_Gather(r_result, N/comm_size, MPI_DOUBLE, result, N/comm_size,
        MPI_DOUBLE, 0, MPI_COMM_WORLD);
    // End of Change #2
```

(c)

```
    // Change #1 at line 11
    // communication overlap can be achieved in the distribution of the vector
        and computation of the result
    if(rank == 0)
        for(i = 1; i < comm_size; i++)
            MPI_ISend(vector, N, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &request[i]);
    else
        MPI_Recv(vector, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status); // use
            blocking Recv, since there is nothing to overlap from the receivers
            perspective

    // now we can perform the Matrix-vector multiplication, only for the subset
        of rows
    for (i = 0; i < N/comm_size; i++) {
        result[i] = 0.0;
        for (j = 0; j < N; j++) {
            result[i] += matrix[i][j] * vector[j];
```

```
            }
        }
        // End of Change #1
        // note there is no need to MPI_Wait, because after this operation root (
            rank 0) needs to wait for the result via MPI_Recv (a), and the completion
             of this operation implies that the MPI_ISend() operations have also
            completed
```

(d) While the previous code can start the matrix-vector product on the root note (rank = 0) early, it is unlikely to lead to speed-up, since sending the vector and computing the partial matrix-vector on the worker processes (rank != 0) is likely to be the slowest component, making the code in (c) perform similar to (a). The code using collectives is likely to perform much better.

(e) Host code

```
    // Outside of the iteration

    // Create device pointers
    double *dev_matrix, *dev_vector, *dev_result;

    /* Allocate Memory on GPU */
    cudaMalloc((void**)&dev_matrix, sizeof(float)*N*N);
    cudaMalloc((void**)&dev_vector, sizeof(float)*N);
    cudaMalloc((void**)&dev_result, sizeof(float)*N);

    /* Matrix A does not change, hence it only needs to be copied once */
    cudaMemcpy(dev_matrix, matrix, sizeof(double)*N*N, cudaMemcpyHostToDevice);

    // Inside of the iteration:

    /* Vector needs to be copied every iteration */
    cudaMemcpy(dev_vector, vector, sizeof(double)*N, cudaMemcpyHostToDevice);

    /* Call the kernel */
    MatVecKernel<<<N/1024,1024>>>(dev_matrix, dev_vector, dev_result, N);

    /* Copy the vector b back to CPU */
    cudaMemcpy(result, dev_result, sizeof(double)*N, cudaMemcpyDeviceToHost);

    // normalization and update proceed
}
```

(f) The simplest implementation of the GPU kernel is to consider one CUDA thread per row.

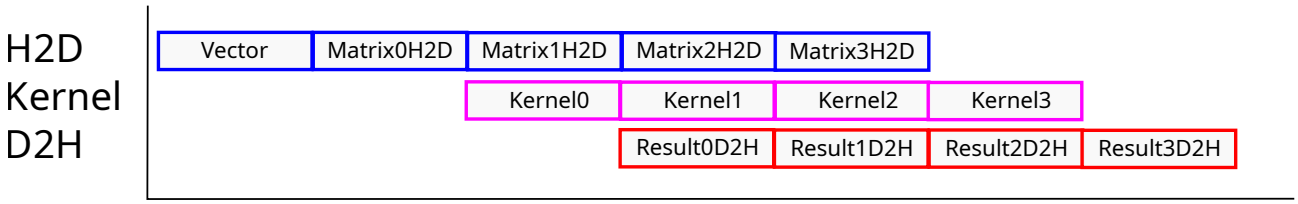
```
__global__ void MatVecKernel(double * matrix, double * vector, double * result, int N){
    /* Compute the "global" thread id, which is the same as the matrix row */
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    double sum = 0;
    /* A check for row < N is not needed, as long as 1024 % blockDim.x == 0 */
    for(int col = 0; i < N; i++) {
      sum += matrix[row * N + col] * vector[col];
    }
    result[row] = sum;
}
```

(g) The kernel can be improved by exploiting the temporal locality of the vector. For this, we can store the vector in the shared memory. We are given the constraint N < 1024. This allows to simplify the solution by considering that each thread in the thread block can

copy one element and compute one row of the product. At the same time, this means that there will only be a single thread block, so calculation of the row is also simplified.

```
__global__ void MatVecKernel(double *matrix, double *vector, double *result, int N) {
    // Allocate shared memory for the vector
    __shared__ double sharedVector[1024]; // 1024 is the max size

    // Calculate the row index of the matrix
    int row = threadIdx.x; // since there is a single thread block

    // Load the vector into shared memory
    sharedVector[threadIdx.x] = vector[threadIdx.x];

    // synchronize, to guarantee that all the threads have finished copying the vector
    __syncthreads();

    // Perform the matrix-vector multiplication
    double sum = 0.0;
    for (int col = 0; col < N; ++col) {
        sum += matrix[row * N + col] * sharedVector[col];
    }
    result[row] = sum;
}
```

(h) 1. CUDA streams are sequences of operations that execute on the CUDA device, asynchronously from the host. 2. CUDA streams can be used to overlap the sending, computation, and receiving of the matrix and the vectors. An idealized diagram is depicted below:



(i) First change: The matrix needs to be partitioned into blocks, such that each CUDA stream processes one block. Second change: The host needs to pin the matrix to memory using cudaMallocHost().

Problem 5 (6 points)

Consider the following prefix sum code.

Listing 1: prefix sum

```
int N = 1000;
int A[N], prefix_sum[N];

// Initialize array
for (int i = 0; i < N; i++) {
    A[i] = i + 1;
}

prefix_sum[0] = A[0];

for (int i = 1; i < N; i++) {
    prefix_sum[i] = prefix_sum[i - 1] + A[i];
}
```

Listing 2: prefix sum algorithm rewritten

```
int N = 1000;
int A[N], prefix_sum[N];

for (int i = 0; i < N; i++) {
    A[i] = i + 1;
}

prefix_sum[0] = A[0];

for (int d = 1; d < N; d *= 2) {
    int * read = (int*)malloc(sizeof(int)*N);

    memcpy(read, prefix_sum, sizeof(int)*N);

    for (int i = d; i < N; i += 1) {
        prefix_sum[i] += read[i - d];
    }
}
```

Tasks:

(a) Is the code in Listing 1 parallelizable using OpenMP. Why? Why not?

(b) To parallelize the code we have to rewrite the prefix sum algorithm, presented in Listing 2. Which `for` loops can be parallelized using OpenMP constructs? And why? Write down pseudo code.

(c) Will your code be faster than the serial version? Justify your answer.

(d) What are the main disadvantages of using synchronization constructs such as 'critical', 'barrier' and 'atomics' for this code?

Answers

(a) There is a dependency across the iterations, which makes it harder to parallelize by simply using OpenMP constructs.

(b) Below is one of the methods that solve the problem with OpenMP constructs.

```
int N = 1000;
int A[N], prefix_sum[N];

for (int i = 0; i < N; i++) {
    A[i] = i + 1;
}

prefix_sum[0] = A[0];

for (int d = 1; d < N; d *= 2) {
    int * read = (int*)malloc(sizeof(int)*N);

    memcpy(read, prefix_sum, sizeof(int)*N);

    #pragma omp parallel for
    for (int i = d; i < N; i += 1) {
        prefix_sum[i] += read[i - d];
    }
}
```

(c) The code given above is of order $O(n \log n)$. which should make this slower than serial code.

(d) The parallel code should be faster than using synchronization constructs such as barriers, which will introduce high overheads.