Solutions for the Exam in DAT400 (Chalmers) and DIT431 (GU) High Performance Parallel Programming, Friday, October 27th, 2023, 14:00h - 18:00h

Problem 1 (10 points)

In this problem we ask you to reason about how user level threads and kernel level threads impact performance, how threading potentially impacts MPI processing, and how CUDA threads relate to these concepts.

Tasks:

(a) Assume a multicore system on which we want to run a parallel application. The runtime library manages user level threads (ULT) with a context switch overhead of 1 time unit. Threads managed by the operating system (kernel level threads, KLT) have a context switch overhead of 5 units, which can be triggered either by I/O (disk, network, etc.) or because all ULTs assigned to a running KLT have completed. The total work of the application is 1000 tasks, such that each task takes 10 units. The application is to be executed on 10 cores. Each ULT executes a single task. The developers consider the following three models: (1) `N:1` (i.e. all ULTs mapped to 1 KLT), (2) `1:1` (each ULT mapped to a different KLT), and `N:M`, with a 10:1 ratio (i.e. 10 ULTs mapped to 1 KLT). What are the execution times for all these three cases?

(b) The program is modified so that each task now includes a checkpointing stage at the end of the computation. The time it takes to perform the checkpoint is 10 time units. Checkpointing consists in writing the tasks' data to disk. Reason about the suitability of the three threading models `N:1`, `1:1` and `N:M` (with both 100:1 and 10:1 ratios).

(c) Instead of checkpointing, the developers now intend to use MPI in order to delegate checkpointing to a different MPI process running on a different node. Assume an MPI library that has been written with only one kernel level thread (KLT) in mind. Discuss how the various threading models (`1:1`, `N:M` and `N:1`) potentially impact the program. Depending on the case, will it be necessary to modify the MPI library or the application?

(d) Now let's consider the CUDA threading model. CUDA supports very fine-grained threads. How do CUDA threads compare to ULTs/KLTs? How does CUDA effectively support the fine thread granularity? Explain also in 1-2 lines how the CUDA threading model relates to SIMD.

Answers

(a) Consider the execution times on a single-core

Case N:1 (all ults on 1 klt)

1000 x (10 +1) = 11000

```
ULTs_per_core x (time_task + ULT_context_switch)
```

Case 1:1

1000 / 10 x (10 + 5) = 1500

```
KLTs_per_core x (time_task + KLT_context_swtich)
```

Case N:M (10:1)

10 x (10 x (10 + 1) + 5) = 10 x 115 = 1150

```
KLTs_per_core x (ULTs_per_KLT x (task_time + ULT_context_switch) + KLT_context_switc
```

(b) **N:1** is clearly suboptimal since each write to disk will suspend the execution. So, despite ample parallelism the execution time will duplicate

1:1 This is a good option, every time the thread checkpoints, a different KLT can be invoked to continue processing the program. This will effectively hide the latency of the checkpointing operation.

N:M with 1000:100 Similar to the previous case, as we still have 10 KLTs per core, we can make use of I/O and computation overlap, thus hiding the latency of the checkpointing.

N:M with 1000:10 In this case we have 100 ULTs for each KLT, so 10 KLTs will execute the whole application. As there are 10 cores, we execute 1 KLT for each core. This means that upon a context switch, there will be no ready KLTs to execute, and the execution time of the program will increase.

(c) **Case N:1** It will work out of the box since the MPI library is invoked always by the same KLT. However, if MPI calls are synchronous the execution time will be impacted proportionally. To avoid this serialization the MPI library could collaborate with the user space threading library to switch to a new ULT while the MPI library tracks the communication in the background

Case 1:1 Since the MPI library does not expect more than one KLT (i.e., it is not thread-safe), this will likely result in MPI errors. Two ways to avoid this are (1) having a single thread call all MPI routines on behalf of the other threads, or (2) extend the MPI library to be (KLT) thread-aware

Case N:M Case 1:1 applies since the MPI library needs to be thread safe. If the number of KLTs is lower than the number of cores, then case N:1 also applies, since communication-computation overlap will not automatically happen

(d) CUDA threads are more closely related to ULTs as they are fully managed by the CUDA library. Unlike KLTs, CUDA threads cannot be preempted, and context switching is only possible when the threads have run to completion. CUDA supports fine grained threading via specific hardware support. Fine-grained threading is used in CUDA to hide the latency of memory access. Consecutive threads within a thread block are dynamically assembled and executed as a single warp (a bunch of threads tracked as a single instruction). Hence a warp is basically a SIMD instruction.

Problem 2 (10p)

Consider the following example of a nested loop with depth 2.

```
#define N 64
for(int j=0;j<N;j++)
{
    for(int k=0;k<N;k++)
    {
        a[j*N+k] = (j+1)*N+k;
    }
}
```

Tasks:

(a) Consider the following schedulers: (a) static scheduling, dynamic scheduling with (b) "self-scheduling", and (c) "chunk-scheduling with a chunk size of 6", and (d) "guided scheduling with a minimum chunk size of 4". The above loop is to be executed on a given system with 8 processors. Considering the above schedulers applied to a parallelization of the outer loop. How many time units will it take to complete the given code? Assume that each iteration takes 2.0 time units to complete.

(b) Next assume the programmers have bought a system with 100 processors. Propose a loop transformation for the above code to offer enough parallelism to this new system with 100 processors. Show the transformed code.

(c) Now assume that a fixed overhead is paid to create the parallel loop (the transformed loop from part (b)). This overhead includes the generation of all the threads needed for the parallel execution, and the destruction of those threads after the execution. The overhead depends on the number of processors (P), and is P $\times$ 0.5 time units. Considering the case of static scheduling: after which value of P does adding new processors result in a slowdown instead of a speed-up?

Answers

(a) Each outer iteration takes 128 time units. Following will be tasks, iterations and time unit

1. 8 tasks, each 8 outer iterations = 16.0 x 64 = 1024 time units
2. 64 tasks, each 1 iteration = 16.0 x 64 = 1024 time units
3. total 11 tasks: 10 tasks with 6 iterations each, and one task with 4 iterations = 24.0 x 64 = 1536 time units
4. 13 tasks with iterations: 8,7,7,6,5,4,4,4,4,4,4,4,3 = 18.0 x 64 = 1152 time units.

(b) Loop collapsing

```
for(int j=0;j<N*N;j++)
{
    a[j] = j+N;
}
```

Since N x N = 4096, the loop now provides enough parallelism to 100 core processors machine.

(c) static scheduling = (4096/p) * 2 ns + (P * 0.5ns)
derivative with respect to p = 8192*pow(p, -2) + 0.5
0 = (-1)*8192*pow(p, -2) + 0.5
8192 = 0.5*(pow(p, 2))
(pow(p, 2)) = 8192/0.5 = 16384
p = $\sim$128 processors

Problem 3 (10p)

Now consider the following matrix-matrix multiplication code:

```
#define M 100
#define N 100
#define K 100
void gemm( float *A, float *B, float *C)
{
    int i,j,k;
    for(i = 0; i < M; ++i){
        for(k = 0; k < K; ++k){
            for(j = 0; j < N; ++j){
                C[i*N+j] +=  A[i*K+k]*B[k*N+j];
            }
        }
    }
}
```

Tasks:

(a) What is the Arithmetic Intensity of this code? Consider a multicore chip in which each core has a performance of 4 GFLOPS (single precision) and a shared memory bus of 16GB/s. Using the simplified DRAM roofline model, what will be the performance (i.e. the execution time) if the code is run on a single core or on 8 cores? Note: a single `float` consists of 4 bytes.

(b) By the use of SIMD, the performance of each core is multiplied by four. In this new regime, what is the next bottleneck that should be addressed: FLOPS or Memory (GB/s)? Explain your reasoning. Further, assume that the developers have bought a 32-core machine, with each core having the same SIMD unit. Will that change the bottleneck that should be addressed?

(c) Let's assume the above code is included in a program that has a serial component that takes 0.5 ms. Consider the multi-core chip from part (a). What is the upper limit to the speed-up (compared to a single core) according to Amdahl's law? For the 8 core system, compute the speed-up over a single core. Discuss, individually, the suitability of (i) executing the kernel on even more cores, (ii) optimizing the kernel to achieve a higher AI, or (iii) optimizing the serial part of the computation.

Answers

(a) FLOPS=2*M*N*K, Memory access = 4*(MN+NK+KN) = 12e4

AI = 2e6/12e4 = 16.7 FLOPS/BYTES

GEMM's AI depends on the input size.

Performance on the system with 1 core

min(4 GFLOPS, 16.7 $\times$ 16GB/s) = min(4 GFLOPS, 265 GFLOPS) = 4 GFLOPS

Performance on the system with 8 cores min(32 GFLOPS, 16.7 $\times$ 16GB/s) = min(32 GFLOPS, 265 GFLOPS) = 32 GFLOPS

The number of flops of the algorithm is 2 x M x N x K = 2000000. Hence execution times are:
serial: 2e6 / 4e9 = 0.5 ms

parallel: 2e6 / 32e9 = 0.06ms

(b) SIMD unit will increase the performance of each core by 4 times. So new values of GFLOPS per core = 16
Performance on the system with 1 core:
min(16 GFLOPS, 16.7 $\times$ 16GB/s) = min(16 GFLOPS, 265 GFLOPS) = 16 GFLOPS

Performance on the system with 8 cores = min(128 GFLOPS, 16.7 $\times$ 16GB/s) = min(128 GFLOPS, 265 GFLOPS) = 128 GFLOPS

The number of flops of the algorithm is 2 x M x N x K = 2000000. Hence execution times are:
serial: 2e6 / 16e9 = 0.125 ms
parallel: 2e6 / 128e9 = 0.016ms

Adding more performance to the system improves the overall speedup. Still, the algorithm is compute-bound, so adding more optimizations will improve the performance.

Increase the number of cores to 32. Performance with the system with 32 cores:
Performance on the system with 16 cores=
min(512 GFLOPS, 16.7 $\times$ 16GB/s) = min(512 GFLOPS, 265 GFLOPS) = 265 GFLOPS

With 32 cores, memory (GB/s) becomes a bottleneck.

(c) Computational time with 1 core from part[(a)] is 0.5ms.
the serial component that is now added in the program = 0.5ms
Total execution time on 1 core= 1ms
Serial fraction that can not be parallelized= 50%
Maximum speed-up limit = 1/0.5 = 2x
Achieved speedup with 8 cores = T(s) / T(p) = 1/0.56 = 1.8x

Now let's say we increase the cores: that will only increase the performance for the parallel part and that will not be beyond 2x

Optimizing the kernel also can help a bit but again the performance is limited by the serial fraction.
Optimizing the serial fraction should be considered and it will provide scalability and speedup with more cores.

Problem 4 (10p)

The following two code listings show two different loops.

Loop 1:

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

Loop 2:

```
int quad = 0

int N = 1000;
int range = 1;
float dx = (float)range/(float) N;
float* x = (float *) malloc(sizeof(float)*N);
x[0] = 0;
float temp = 0;

for(int i = 1; i < N; i++)
{
    x[i] = i* dx;
    temp = dx/(1+x[i]);

    quad += temp;
}
```

Tasks:

(a) Can Loop 1 be vectorized? why? why not?

(b) Based on your answer for task (a), apply the relevant changes to the code and add the relevant OpenMP constructs to vectorize the loop.

(c) Loop 2 presents the numerical integration $I = \int_0^1 \frac{1}{1+x} dx = \log(2)$. List three methods of parallelizing the given code using OpenMP. Write the pragmas you would use for these methods, along with the data sharing attributes for the different variables.

(d) How do you expect the performance of the three methods listed in task (c) to differ? For each of the methods, briefly describe one way in which the OpenMP compiler could implement these methods.

Answers

(a) No the code can't be vectorized, because of the dependencies between the different iterations.

(b) Solution for task b is as follows.

Listing 1: solution for task b

```
1     #pragma omp simd
2     for (int i = 0 ; i < N/2 ; i++)
3         A[i] += A[N-1-i];
4
5     #pragma omp simd
6     for (int i = N/2+1 ; i <= N-1 ; i++)
7         A[i] += A[N-1-i];
```

(c) 1. First method `pragma omp parallel for private(temp) reduction(+:quad)`

2. Second method `pragma omp parallel for private(temp) and pragma omp critical`

3. Third Method `pragma omp parallel for private(temp) and pragma omp atomic`

Other methods are also possible.

(d) Assuming the previous three answers have been provided, the expected performance is as follows. Using reduction is expected to perform best (method 1), followed by omp atomic (method 3) and finally omp critical (method 2). These methods may be implemented in the following way:

reduction Private copies of the reduction variable are generated. A local reduction is performed on these copies. Outside of the parallel loop the final values are then globally reduced

critical This can be implemented by using a global lock that protects the reduction.

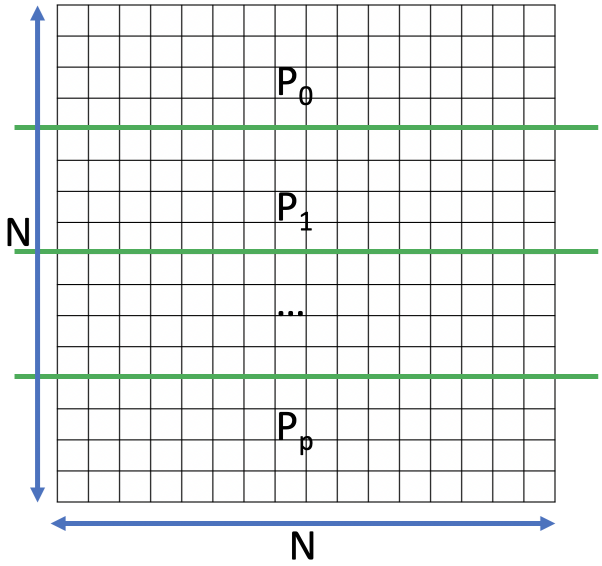
atomic In this case the compiler will avoid using a global lock and instead use atomic instructions at the ISA level, which enables more fine grained atomicity and thus offers better performance than a global lock

Problem 5 (10p)

For this problem consider a set of P processors, each on a separate node, and without shared memory (i.e. with distributed memory).

Tasks:

(a) Consider the matrix shown below, and its partitioning over the set of P processors. The matrix is initially stored in the DRAM memory of one processor. Give at least two different ways to distribute the different blocks of the matrix to the P different processors. The answer should be in the form of a pseudo-code.



(b) Next consider an array of M = 10000 integers (type `int`). Write an MPI code, to be run on P processors, which counts the number of zeros (0) in the given array. While doing so, you should minimize the data communication between the P processes.

(c) Finally, consider an MPI program which uses the MPI communication primitives *SSend* and *SSrecv*. We know that this program does not deadlock. Later on these are changed to non-blocking *Isend* and *Irecv* with *Wait* calls added immediately after each *Isend* and *Irecv*. Will the new program deadlock from communication? Explain your reasoning.

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,
24          MPI_Comm comm)
```

Answers

(a) First method:

Listing 2: solution 1 for task a

```
1  float a[N*N];
2  MPI_init(&argc, &argv);
3  int mpi_rank; MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
4  int mpi_size; MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
5
6  MPI_Scatter(a, size(a)/mpi_size, MPI_FLOAT, b, size(a)/mpi_size,
7  MPI_FLOAT, 0, MPI_COMM_WORLD);
8
9  MPI_finalise();
10 return 0;
```

Second Method:

Listing 3: solution 2 for task a

```
1  float a[N*N];
2  int mpi_rank;
3  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
4  int mpi_size;
5  MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
6
7  int tag = 12;
8
9  count = (N*N)/(mpi_size-1)
10
11 int *localArray = malloc(count * sizeof(float));
12
13 if (mpi_rank == 0) {
14
15     for(int dest = 1; dest < mpi_size; ++dest)
16     {
17         MPI_Send(&a[(dest-1)*count], count, MPI_FLOAT, dest, tag,
18         MPI_COMM_WORLD);
19     }
20 }
21 else
22 {
23     MPI_Recv(localArray, count, MPI_FLOAT, 0, tag, MPI_COMM_WORLD,
24     MPI_STATUS_IGNORE);
25 }
26
27  MPI_Finalize();
```

```
28   return 0;
```

(b) Here is one potential solution for this task. In our solution, we use scatter instead of bcast in order to reduce communication.

Listing 4: solution for task b

```
1
2  #include <mpi.h>
3  #define N 10000
4  int main (int argc, char *argv[])
5  {
6   int A[N], B[N];
7   int i, rank, answer, P = 10, totalZero = 0;
8   int chunksize = (int) ceil( ((double)N) / P);
9   MPI_Status status;
10  MPI_Init (&argc, &argv);
11  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
12  MPI_Comm_size( MPI_COMM_WORLD, &P);
13  MPI_Scatter(A, chunksize, MPI_INT, B, chunksize, 0, MPI_COMM_WORLD);
14  totalZero = 0;
15  for (i = 0; i < chunksize && rank * chunksize + i < N; i++) {
16     totalZero += (B[i] == 0);
17  }
18  MPI_Reduce(&totalZero, &answer, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
19  // Now we have the answer
20  std::cout<<'Number of zeros is: '<< answer<< std::endl;
21  MPI_Finalize()
22
23  return 0;
24 }
```

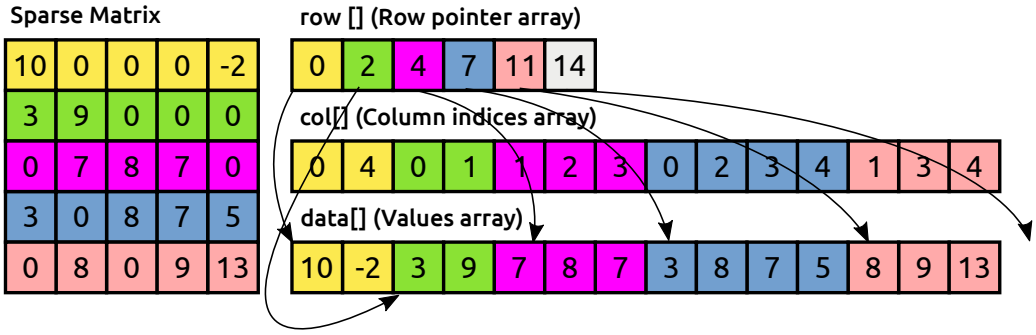
(c) No, the program would not deadlock. This is because *Wait* call after non-blocking communication(*Isend* and *Irecv* in this case) corresponds to blocking communication. Since if synchronous communication doesn't deadlock then blocking communication also does not deadlock, this implies that this program will also not deadlock.

Problem 6 (10p)

The following CUDA code implements a Sparse Matrix Vector product, i.e. the product of a sparse matrix (a matrix that contains many zero elements) and a dense vector. To effectively store the matrix, all non-zero values are held in a vector `data[]`, sorted by rows and columns. The vector `row[]` contains indices into the vector `data[]` such that `row[x]` points to the first non-zero element of row 'x'. For example, `row[1]` points to the first non-zero value of row 1, while `row[2]-1` points to the last non-zero value. If row 1 contains no non-zero values, then `row[1] == row[2]`. Similarly, `col[]` holds the column indices corresponding to each value of `data[]`. See also the figure on the next page. In the below kernel code, each thread computes one row of the output vector `y[]`.

```
1  __global__ void spmv(int *row, int *col, float *data, float *x, float *y, int N) {
2      int tid = blockIdx.x * blockDim.x + threadIdx.x; // row ID
3      if (tid < N) {
4          float dot = 0.0;
5          for (int i = row[tid]; i < row[tid + 1]; i++) {
6              dot += data[i] * x[col[i]];
7          }
8          y[tid] = dot;
9      }
10 }
11 // Sparse Matrix-Vector Product: y[] = A[][] * x[]; A[][] is sparse, x[],y[] are dense
12 int main() {
13     int N;  // number of rows
14     int M;  // number of columns
15     int row[] = {...}; // Row pointer array
16     int col[] = {...}; // Column indices
17     float data[] = {...}; // Non-zero values
18     float x[M] = {...}; // Input vector
19     float y[N] = {...}; // Output vector
20
21     int *d_row, *d_col;
22     float *d_data, *d_x, *d_y;
23
24     // Allocate memory on the device
25     cudaMalloc(A, (N + 1) * sizeof(int));
26     cudaMalloc(B, sizeof(col));
27     cudaMalloc(C, sizeof(data));
28     cudaMalloc(D, M * sizeof(float));
29     cudaMalloc(E, N * sizeof(float));
30
31     // Copy data to device
32     cudaMemcpy(F, (N + 1) * sizeof(int), cudaMemcpyHostToDevice);
33     cudaMemcpy(G, sizeof(col), cudaMemcpyHostToDevice);
34     cudaMemcpy(H, sizeof(data), cudaMemcpyHostToDevice);
35     cudaMemcpy(I, N * sizeof(float), cudaMemcpyHostToDevice);
36
37     // Perform SpMV
38     spmv<<<J>>>(K);
39
40     // Copy the result back to the host
41     cudaMemcpy(L, N * sizeof(float), cudaMemcpyDeviceToHost);
42
43     // Free memory on the device
44     cudaFree(d_row);     cudaFree(d_col);
45     cudaFree(d_data);    cudaFree(d_x);
46     cudaFree(d_y);
47     return 0;
48 }
```

The clarify the storage format, the following figure shows a sample sparse matrix and its representation as the three arrays `data[]`, `row[]` and `col[]`.



Tasks:

(a) Complete the code of the `main()` function by filling the values of A through L.

(b) Next we want to use shared memory to improve the kernel. Identify which of the vectors offers temporal locality. Show how to modify the above CUDA kernel code to make use of shared memory. For this task you should assume that the length of the vector `x[]` (same as the number of columns of the matrix) is no more than 1024 .

(c) How would you change the code if the length of vector `x[]` is above 1024 but still fitting into shared memory? Provide the CUDA code the implements this modification

(d) How should you change the code if vector `x[]` is so large that it does not fit into the shared memory? Explain either in words or pseudocode.

(e) Is it possible to use CUDA streams to overlap computation and communication in the previous code? Explain how the code would need to be changed.

Answers

(a)
```
A = &d_row
B = &d_col
C = &d_data
D = &d_x
E = &d_y
F = d_row, row
G = d_col, col
H = d_data, data
I = d_x, x
J = 1, N   // as long as A x B >= N, any pair of parameters should work
K = d_row, d_col, d_data, d_x, d_y, N
L = y, d_y
```

Note: there was a typo in line 28, it should have said 'M' instead of 'N'. This has been corrected in the question, mistakes in this line will not be taken into account.

(b) In a matrix-vector product, the vector is reused several times (depending on the number of rows and the sparsity). Hence one option is to load this vector into the shared memory. Since we know that the vector has length less than 1024 entries, we can use all the vectors in a threadblock once to load the elements into the shared memory.

```
    #include <stdio.h>

#define VECTOR_SIZE 1024

__global__ void spmv(int *row, int *col, float *data, float *x, float *y, int N) {
    __shared__ float shared_x[VECTOR_SIZE];

    int tid = blockIdx.x * blockDim.x + threadIdx.x; // the row index, used later

    // Load x into shared memory
    if (threadIdx.x < VECTOR_SIZE) {
        shared_x[threadIdx.x] = x[threadIdx.x];
    }
    __syncthreads();

    if (tid < N) {
        float dot = 0.0;
        for (int i = row[tid]; i < row[tid + 1]; i++) {
            dot += data[i] * shared_x[col[i]];
        }
        y[tid] = dot;
    }
}
```

(c) One solution is to loop several times until all the elements have been copied

```
for(int i; i < ceil(BLOCK_SIZE / blockDim.x); i++)
    if (threadIdx.x + i*blockDim.x < VECTOR_SIZE)
        shared_x[i*blockDim.x + threadIdx.x ] = x[threadIdx.x + i*blockDim.x];
```

(d) If the vector is so long that it does not fit, then the vector can be partitioned across several CUDA thread blocks. Since the code already uses several thread blocks to write all the rows, this will lead to having a 2D grid of thread blocks, each of which will lead to a partial update of the corresponding output vector rows

(e) Yes, this is possible. Each stream could be responsible for a subset of the rows in the matrix. In the case of variant (d), there would be several thread blocks per stream to process the various chunks of vector `x[]`.