Solutions for the Exam in DAT400 (Chalmers) and DIT430 (GU) High Performance Parallel Programming, Wednesday, October 28th, 202020, 14:00h - 18:00h

Problem 1 (8p)

OpenMP and MPI are two different libraries used to parallelize an application on a given system.

(a) Motivate a scenario where one must consider using MPI (or similar libraries), either with or without OpenMP, to parallelize their application.

(b) Programming in MPI is known to be prone to deadlocks. While in OpenMP, there is a high risk of race conditions if not carefully considered. In no more than 2 lines, what is the reason of the risk associated with each programming model?

(c) A code snippet is provided below. Suggest an OpenMP and an MPI implementation of this code. You can consider mix of the two as well. Which version do you think is better and why?

```
1  for(int i=0; i<N; i++)
2    for(int j=0; j<M; j++)
3      for(int k=0; k<M; k++)
4        B[i][j][k] = A[i-1][j][k] * B[i][j][k-1];
```

Answer to Problem 1:

(a) When the parallel application's data can no longer fit in a single node's memory, i.e. there's a growing need to solve a larger problem, MPI or similar libraries should be used to leverage distributed memory from multiple nodes. This resembles the notion of Gustafson's law.

(b) In OpenMP, variables are shared among the threads, if defined outside a parallel region. This causes a race condition if those variables are read and written in a non-deterministic way. Additionally, MPI Processes contain a private copy of data but the communication is strict to the order of MPI calls. Hence, if Sends and Recvs are not carefully ordered it may cause a deadlock situation.

(c)
```
1   // OpenMP - 1
2   #pragma omp parallel for
3   for(int i=0; i<N; i++)
4     for(int j=0; j<M; j++)
5       for(int k=0; k<M; k++)
6         B[i][j][k] = A[i-1][j][k] * B[i][j][k-1];
7   // OpenMP - 2
8   for(int i=0; i<N; i++)
9   #pragma omp parallel for
10    for(int j=0; j<M; j++)
11      for(int k=0; k<M; k++)
12        B[i][j][k] = A[i-1][j][k] * B[i][j][k-1];
13  // Third loop cannot be parallelized because of loop carried dependency.
14  //MPI
15  //code for sending A and B from master to all processes
16  int start = rank * N/COMM_SIZE;
17  int end = start + N/COMM_SIZE;
18  for(int i=start; i<end; i++)
19    for(int j=0; j<M; j++)
20      for(int k=0; k<M; k++)
21        B[i][j][k] = A[i-1][j][k] * B[i][j][k-1];
22  MPI_Barrier(MPI_COMM_WORLD)
23  //Gather B from all processes
24
25  //MPI + OMP
26  //code for sending A and B from master to all processes
27  int start = rank * N/COMM_SIZE;
28  int end = start + N/COMM_SIZE;
29  for(int i=start; i<end; i++)
30  #pragma omp parallel for
31    for(int j=0; j<M; j++)
32      for(int k=0; k<M; k++)
33        B[i][j][k] = A[i-1][j][k] * B[i][j][k-1];
34  MPI_Barrier(MPI_COMM_WORLD)
35  //Gather B from all processes
```

Problem 2 (10p)

A research computing company is assigned a task, that is to run a large application on their hardware infrastructure. The application includes four programs. Each program depends on the result returned from the one before it. Hence, Program i cannot start before Program $i-1$ has returned the result ($1 < i \leq 4$). The team analyzed and tested the **serial** execution time of each program on an x86 CPU. The results came as follows:

- Program 1 - 10 minutes (can be parallelized)
- Program 2 - 10 minutes (can be parallelized)
- Program 3 - 10 minutes (cannot be parallelized)
- Program 4 - 10 minutes (cannot be parallelized)

(a) Assume the company works 8 hours per day and gets $10 each time they complete running the whole application. How much do they earn per day?

(b) They get some advice from a GPU sales person that running entirely on a GPU will help a lot, and they could obtain the following speedups for each program:

- Program 1 - Speedup = 100
- Program 2 - Speedup = 10
- Program 3 - Speedup = 1
- Program 4 - Speedup = 0.2

By averaging the speedups = (100+10+1+0.2)/4 = 27.8, they think it is profitable to do so, and decide to buy the GPU product for this application. What do you think?

(c) They also think they could run program 4 on the CPU and run the other three on the GPU. For simplicity, ignore the time for communication time sending data from CPU to GPU and vice versa. How about this combined method, comparing with running on CPU totally?

(d) Think about Amdahl's law. What is the upper limit speedup for this application? Give explanations about the speedups you get from parts (b) and (c).

Answer to Problem 2:

Part (a) Completing the whole application takes 40 mins, which means (8×60)/40 = 12 times per day. Then they can earn 12 × 10 = $120 per day.

Part (b) The total execution time for all four programs when running on GPU:

$$10/100 + 10/10 + 10/1 + 10/0.2 = 61.1mins. \tag{1}$$

Running on GPU causes a slowdown compared to running on x86 entirely. So it is not a good idea to run the whole application on the GPU.

Part (c) Execution time when using CPU + GPU:

$$10/100 + 10/10 + 10/1 + 10 = 21.1mins. \tag{2}$$

Speedup = 40 / 21.1 = 1.896.

Part (d) Amdahl's law:

$$Speedup = \frac{1}{\frac{f}{s} + (1 - f)} \tag{3}$$

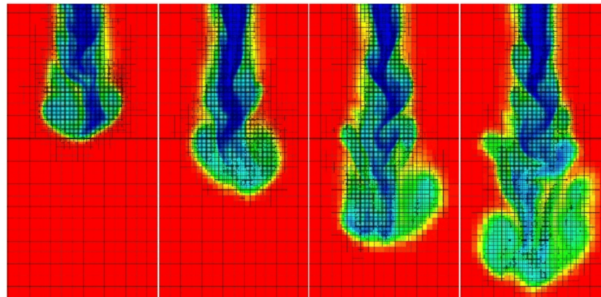
where f is the fraction of an application that can be improved. In the application, half of programs (program 1 and 2) can be parallelized, which means f = 0.5. The upper limit of speedup is 1/(1-f) = 2. The speed-up of 1.896 we get from problem (c) is getting close to the value of 2.

Note that if the "improvement" results in a slow-down for some portion (Program 4), then that can result in a very large performance degradation. It's tempting to just try to average the speedups. However in reality, we need to account for the slowest part.

Problem 3 (8p)

Consider that there are 4 regions of computations as visualized by the figure below. You would like to use MPI to parallelize these computations on a small cluster that contains 4 nodes, where each node has 8 processors.

(a) Each region contains computations that require broadcasting the data among processes within each node. How will you divide the work between communicators? Write a pseudo code to demonstrate that.



Domain decomposition for AMR

(b) Assume you would like to block until all processes in the communicator have reached a certain point in the code. How would you achieve such a behavior within a given node and also across all nodes?

```
1 float a = rank;
2 float b;
3 if(rank != 0)
4 {
5   MPI_Recv(&b, 1, MPI_FLOAT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status)
6   printf("Value of b = %f from rank %d", b, rank);
7 }
8 else
9   MPI_Send(&a, 1, MPI_FLOAT, 2, 1, MPI_COMM_WORLD)
```

(c) You encounter the code above. What could be an issue with this code? Correct the issue using only MPI_Send and MPI_Recv calls.

(d) Can MPI_Send and MPI_Recv calls be replaced by any collective in the given scenario? What would be such a collective?

Answer to Problem 3:

(a) Task 1: Create 4 communicators, 8 processes per communicator. Task 2;

```
//Solution for a
// Assuming there are 32 processes
int color = rank/8; // 4 different region
MPI_Comm region_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &region_comm);
int region_rank, region_size;
MPI_Comm_rank(region_comm, &region_rank);
MPI_Comm_size(region_comm, &region_size);
printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
world_rank, world_size, region_rank, region_size);
MPI_Bcast(&color, 1, MPI_INT, 0, region_comm);
MPI_Barrier(region_comm);
if(region_rank == 0)
    printf("Color is %d ", color);
MPI_Comm_free(&region_comm);
```

(b) Change the communicator based on what level you would like to block processes.

```
//Solution for b
//Barrier across region
MPI_Barrier(region_comm);
//Barrier across all the nodes
MPI_Barrier(MPI_COMM_WORLD);
```

(c) A deadlock happens in this case because only a single receive is fulfilled. A possible fix is as follows:

```
//Solution for c
float b;
if(rank != 0){
    MPI_Recv(&b, 1, MPI_FLOAT,0,1,MPI_COMM_WORLD, &status);
    printf("Value of b = %f from rank %d", b, rank));
}
else {
    float a = rank; //which is 0 here
    for(int i=1; i<COMM_SIZE; i++)
        MPI_Send(&a, 1 , MPI_FLOAT, i, 1, MPI_COMM_WORLD);
}
```

(d) The previous code in (c) appears to be a broadcast call with rank 0 being root.

```
//Solution for d
  MPI_Bcast(&a, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  printf("Value of b = %f from rank %d", b, rank);
```

Problem 4 (8p)

The code below approximates a mathematical constant.

```
1  const int steps = 100000;
2  double step = 1.0/(double) steps;
3  double ax;
4  int i;
5  double x;
6  double sum = 0.0;
7  #ifdef PARALLEL
8  #pragma omp parallel private(...) shared(...)
9  #endif
10 {
11   for (i=0; i<steps; ++i)
12   {
13     x=step*(i+0.5);
14     x*=x;
15     ax=4.0/(1.0+x);
16     sum = sum + ax;  // accumulate
17   }
18 }
19 double result=step*sum;
```

(a) You turn on parallelism (i.e. you `#define PARALLEL` preprocessor flag) and you discover that even though multiple threads are created and the program runs to completion, the `result` does not match that of the serial version. **Assume that the missing** `private,shared` **fields are already set correctly**. In no more than two lines, explain why the parallel code is broken the way it is.

(b) For the 6 variables defined before the parallel region. Write the appropriate scoping levels from the list `private,shared` for a parallel version to work correctly.

(c) Add the necessary OpenMP clause(s) for the code to work correctly. For each added clause, briefly motivate the use of such clause.

(d) If you decide to use any synchronization/coordination in (c), is it possible to avoid such usage? Show the code that avoids it, and mention 1 advantage and 1 disadvantage of each approach (i.e. an approach that uses synchronization/coordinate versus an approach that does not).

Answer to Problem 4:

(a)
- The work is not partitioned between the threads. All threads executed the code, which is lots of unnecessary work.
- `sum` is a shared variable that needs to capture the results. Updates need to be coordinated or synchronized.
- Even if updates are synchronized, `sum` will have the accumulated result as per NTHRADS executions of the code

(b) `private(i,x,ax) shared(steps,step,sum)`

(c) One way to fix the code:

```
1  #pragma omp parallel for reduction(+:sum)
2                  private(i,x,ax) shared(steps,step,sum)
3  for (i=0; i<steps; ++i)
4  {
5    x=step*(i+0.5);
6    x*=x;
7    ax=4.0/(1.0+x);
8    sum += ax;  // accumulate
9  }
10 double result=step*sum;
```

Atomics can also be used:

```
1  #pragma omp parallel for
2                  private(i,x,ax) shared(steps,step,sum)
3  for (i=0; i<steps; ++i)
4  {
5    x=step*(i+0.5);
6    x*=x;
7    ax=4.0/(1.0+x);
8  #pragma omp atomic
9    sum += ax;  // accumulate
10 }
11 double result=step*sum;
```

(d) Yes, it is possible with privatization, i.e. accumulate the sum in thread local variables, then sum up the results.

```
1  #pragma omp parallel
2  {
3  int tid = omp_get_thread_num();
4  #pragma omp for private(i,x,ax) shared(steps,step,sum)
5  for (i=0; i<steps; ++i)
6  {
7    x=step*(i+0.5);
8    x*=x;
9    ax=4.0/(1.0+x);
10   sum_t[tid] += ax;  // accumulate
11 }
12 }
13 for(int i=0; i<num_threads; ++i) sum+=sum_t[i];
14 double result=step*sum;
```

An advantage of privatization is avoiding the overheads of synchronization especially with too many workers.
A disadvantage is vulnerability to false-sharing.
An advantage of coordination is enhanced code readability such that there's no specific code (except directives) that needs to be added for the parallel case.
A disadvantage could be the fact that not all operators or data types support atomic operations or reductions. For reductions, a special function need to be added for operators outside of the list. Atomic operations may impose an extra overhead for non-integer types.

Problem 5 (8p)

You would like to run the 3D N-Body code below on System X. For that, you would have to understand the system's performance and the expected behavior and limitations of your code on such system. So you decide to use DRAM Roofline model to simplify the task. System X hosts a CPU that is clocked at 2.5 GHz and is capable of 1 single precision operation per cycle (i.e. 1 FLOP/cycle). It has a DRAM bandwidth of 5GB/s.

```
1  for (i=0; i<N; i++) {
2    float pi = 0;
3    float axi = 0;
4    float ayi = 0;
5    float azi = 0;
6    float xi = x[i];
7    float yi = y[i];
8    float zi = z[i];
9    for (j=0; j<N; j++) {
10         float dx = x[j] - xi;
11         float dy = y[j] - yi;
12         float dz = z[j] - zi;
13         float R2 = dx * dx + dy * dy + dz * dz + EPS2;
14         float invR = rsqrtf(R2);
15         float mj   = m[j];
16         float invR3 = mj * invR * invR * invR;
17         pi += mj * invR;
18         axi += dx * invR3;
19         ayi += dy * invR3;
20         azi += dz * invR3;
21   }
22   p[i] = pi;
23   ax[i] = axi;
24   ay[i] = ayi;
25   az[i] = azi;
26 }
```

The CPU has 2 generations: **Gen.1)** single core CPU. **Gen.2)** 4-core CPU.

(a) What is the arithmetic intensity of the code above? Note that `rsqrtf` is equivalent to 6 floating point operations on the underlying instruction set. Show your steps. Remember that Arithmetic Intensity (AI) is the ratio of total floating-point operations (FLOPs) performed by a given code or code section, to the total data movement (Bytes) required to support those FLOPs.

(b) For the 2 generations, specify if the code is bounded by compute or memory. Show your steps.

(c) On **Gen.2**, you run the code for some large N, and you discover that for the specific arithmetic intensity that you calculated in (a), the value of the FLOP/s falls below the roof. State 1-2 reasons why this could be the case.

(d) There are several algorithms to implement the N-Body problem (e.g. Barnes-Hut, Fast Multipole, the code above, etc.). Your task is to experimentally decide on the best algorithm on System X for some N. You choose the FLOP/s as your metric, such that the algorithm that scores the highest FLOP/s is the most optimal. Is this a good or a bad choice? Why or why not? If not, what is the alternative?

Answer to Problem 5:

(a) The number of floating point operations is Note the format is $(count : line)$. First part is the count and the second part is the corresponding line of code.

$$\begin{aligned} N^2 \times ((1:10)+(1:11)+(1:12) \\ +(6:13)+(6:14)+(3:16)+(2:17)+(2:18)+(2:19)+(2:20)) = \\ 26N^2 \end{aligned} \quad (4)$$

The number of memory load/stores bytes

$$\begin{aligned} 4 \times (N \times ((1:6)+(1:7)+(1:8)+(1:22)+(1:23)+(1:24)+(1:25))+ \\ N^2 \times ((1:10)+(1:11)+(1:12)+(1:15))) \\ = 4 \times (7N + 4N^2) \\ = 28N + 16N^2 \end{aligned} \quad (5)$$

The arithmetic intensity is:

$$\begin{aligned} &\frac{26N^2}{(28N+16N^2)} \\ &= \frac{26\frac{N^2}{N^2}}{(28\frac{N}{N^2}+16\frac{N^2}{N^2})} \\ &= \frac{26}{\frac{28}{N}+16}, \frac{28}{N} \approx 0, N \gg 28 \\ &= \frac{26}{16} = 1.625(FLOP/Byte) \end{aligned} \quad (6)$$

(b) Theoretical N-Body FLOP/s on System X = $Intensity \times BW = 1.625 \times 5 = 8.125$ GFLOP/s
Peak FLOP/s for Gen.1 = $2.5 \times 1 = 2.5$ GFLOP/s
Peak FLOP/s for Gen.2 = $2.5 \times 1 \times 4(cores) = 10$ GFLOP/s
Classification on Gen.1: 8.125 > 2.5 => **compute-bound**
Classification on Gen.2: 8.125 < 10 => **memory-bound**

(c) Theoretically speaking, intensity is a ratio that should not change for a fixed code and instruction set. However, the attained memory BW can be reduced in experimental runs. Possible reasons include:

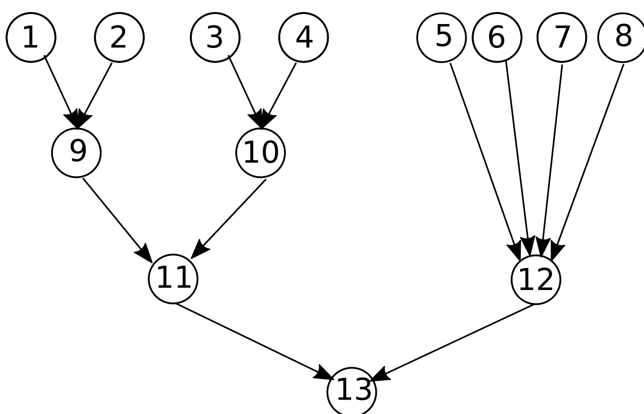
- The memory BW is not fully utilized. One reason could be the lack of parallelism that may result in the lack of using the available memory channels.
- Competition over memory BW from other applications.

(d) FLOP/s is a bad metric to figure out the most optimal across different implementations. A slow implementation such as the N^2 code above have many unnecessary FLOPs and will possibly score higher than optimal algorithms. An alternative method is to compare the execution time for the whole application for a given N.

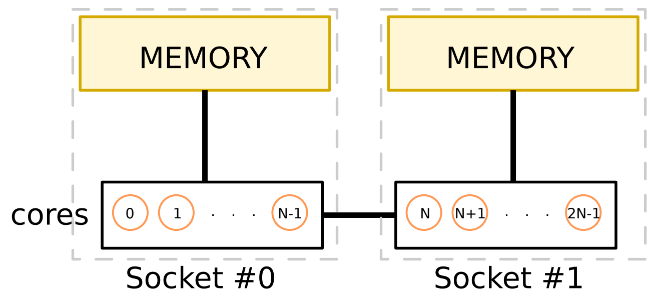
Problem 6 (8p)

You have written a program that includes an asymmetric reduction pattern shown below as a task DAG, where each task is a single-threaded computation executing on one core and taking the same time (1 time-unit) and arrows indicate precedence constraints. Consider the following assumptions:

- The DAG is executed on a machine with a number of cores that are equally distributed among two chips, each with its local memory (i.e. dual-socket), as shown on the right-side figure.
- Assume initially that communication costs between tasks do not add to the execution time.
- The best sequential algorithm has an execution time of 8 time-units.



Task DAG



System Architecture

Tasks:

(a) For systems with 2, 4 and 8 cores, come up with a schedule and mapping that minimizes execution time. For each machine and your proposed optimal schedule, compute the Cost and the Efficiency.

(b) Repeat the problem considering now that each task execution incurs a scheduling overhead that is proportional to the number of incoming edges (num_edges) such that $T_{overhead} = \mathrm{num_edges}$ time-units?

(c) What is the largest number of cores that can accelerate the computation? Why? Explain in 1-3 sentences.

(d) Assume now that communication costs add one time-unit for each edge that crosses across the two sockets. Repeat task (a) with this new assumption.

Answer to Problem 6:

(a) Case 2 cores (execute left branch first)

Time	1	2	3	4	5	6	7	8	9	10	11	12	13
Core 0	1	3	9	5	7	11	13						
Core 1	2	4	10	6	8	12							

Parallel cost = $7 \times 2 = 14$
Efficiency = $\frac{8}{14} = 0.57$

Case 4 cores:

Time	1	2	3	4	5	6	7	8	9	10	11	12	13
Core 0	5	1	9										
Core 1	6	2	10										
Core 2	7	3	12										
Core 3	8	4		11	13								

Parallel cost = $5 \times 4 = 20$
Efficiency = $\frac{8}{20} = 0.4$

Case 8 cores

Time	1	2	3	4	5	6	7	8	9	10	11	12	13
Core 0	1	9	11	13									
Core 1	2	10											
Core 2	3	12											
Core 3	4												
Core 4	5												
Core 5	6												
Core 6	7												
Core 7	8												

Parallel cost = $4 \times 8 = 32$
Efficiency = $\frac{8}{32} = 0.25$

(b) Each task incurs overhead proportional to number of incoming edges. In the timeline this is shown as T.1, T.2, T for the case of $T_{overhead} = num_edges = 2$

2 cores:
Option 1 (left branch first)

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Core 0	1	2	9.1	9.2	9	11.1	11.2	11									
Core 1	3	4	10.1	10.2	10	5	6	7	8	12.1	12.2	12.3	12.4	12	13.1	13.2	13

Option 2 (right branch first)

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Core 0	5	7	1	2	3	4	9.1	9.2	9		11.1	11.2	11			
Core 1	6	8	12.1	12.2	12.3	12.4	12	10.1	10.2	10				13.1	13.2	13

Parallel cost = $16 \times 2 = 32$
Efficiency = $\frac{8}{32} = 0.25$

The general observation is that one should attempt to execute task 12 early in order to amortize its scheduling overhead. Hence we accept as correct all solutions that execute the right branch before the left branch. The same applies for cases of 4 cores. In the case of 8 cores there is no difference between both approaches as there are enough cores to execute all ready tasks.

(c) 8 cores. The maximum possible number of ready tasks at any point in time in the DAG is 8. Any increase in the number of cores results in more idleness and lost efficiency.

(d) We add an overhead unit 'C' each time a dependency edge crosses between two sockets

Case 2 cores:

Cost	1	2	3	4	5	6	7	8	9	10	11	12	13
Core 0	1	3	C	9	5	7	C	11	C	13			
Core 1	2	4	C	10	6	8	C	12					

Parallel cost = $10 \times 2 = 20$
Efficiency = $\frac{8}{20} = 0.4$

It is possible to eliminate communication overheads by scheduling all tasks on the same core, but this would lead to a longer execution time of 13 time units.

Case 4 cores:
7 timeunits would be achieved by executing all tasks on the two cores of the first socket (see question (a)). By executing the left branch on the two cores of the first socket and the right branch on the second socket we achieve an execution time of 6 timeunits.

Case 8 cores:
Again, by executing all the tasks on the first socket we can achieve 5 timeunits (see question (a)). By executing the left branch on the four cores of the first socket and the right branch on the four cores of the second socket we again achieve an execution time of 5 timeunits. This is because critical path is 4, and one extra timeunit is required for communication.

Problem 7 (10p)

Assume a vector x is sized of n floats and n = 67,108,864. The programmer wants to compute the sum of all vector elements on a GPU platform. In order to do the experiments, the programmer writes two versions of CUDA kernel code and launches them with different configurations. However, they are either incorrect and/or inefficient. Your task is to identify the issue(s) in each kernel version.

(a) In the following scenarios, you are expected to list the errors arising from the indicated usages. Errors can be related to hitting a limitation when launching the kernel, and/or incorrect implementation of the highlighted vector sum kernel (program correctness).

[i] Launching one block with 67,108,864 threads executing the kernel, i.e., VectorSum1 ⋘1,67108864⋙ (x,67108864):

```
1  __global__ void VectorSum1(float *x, int n) {
2      int tid = blockDim.x * blockIdx.x + threadIdx.x;
3      if(tid < n) {
4          for(int stride = 2; stride < n; stride *= 2){
5              __syncthreads();
6              if((tid % stride) == 0)
7                  x[tid] = x[tid] + x[tid + stride];
8          }
9      }
10     // The result is in x[0]
11 }
```

[ii] Launching the same code as [i] but with VectorSum1⋘65536,1024⋙ (x, 67108864).

[iii] Launching only one block with 1024 threads executing the kernel where each thread computes the sum of n/1024 elements before the threads perform a reduce, i.e., VectorSum2 ⋘1,1024⋙ (x, 67108864, 65536).

```
1  __global__ void VectorSum2(float *x, int n, int m) {
2      int tid = blockDim.x * blockIdx.x + threadIdx.x;
3      if(tid < n) {
4          float sum = 0.0;
5          // Compute the sum for my part of the array
6          for(int i = 0; i < m; i++)
7              sum += x[tid + i];
8          // Reduce
9          for(int stride = 2; stride < n; stride *= 2){
10             __syncthreads();
11             if((tid % stride) == 0)
12                 x[tid] = x[tid] + x[tid + stride];
13         }
14     }
15     // The result is in x[0]
16 }
```

(b) Firstly, correct your listed errors in VectorSum2 kernel. There are many ways to improve the performance of the slow kernel, such as coalescing global memory references, using shared memory, and so on.

[i] State one optimization you would like to exploit in this kernel (i.e. VectorSum2) and explain why you chose it?

[ii] Write the correct version of VectorSum2 that contains the optimization that you chose in (b.i).

Answer to Problem 7:

Part (a) Version (i) is incorrect because a block can have at most 1024 threads.

Version (ii) is incorrect:

- __syncthreads() only acts as a barrier for threads in the same block. It does not provide synchronization across blocks.
- The calculation is not correct.
  For example, when stride=2, x[0] = x[0]+x[2]; x[2] = x[2] + x[4]; ...
  Those sums are overlapping. It should be: x[0] = x[0]+x[1]; x[2] = x[2] + x[3]; ...

Version (iii) is incorrect:

- The threads compute their initial sums over overlapping parts of the array and miss most of the array.
- The initial sum is not copied back to memory; so it's not available for the reduce.
- The reduce calculation is not correct as version (ii).

Version (iii) is also inefficient. For example:

- Global memory accesses in the reduce loop. The global accesses in the initial sums are unavoidable, but we could use local memory here.

Part (b) Firstly, fix errors of version (iii):

(1) Line 7: sum += x[m * tid + i];

(2) Before reduce add: x[tid] = sum;

Problems and Optimizations:

(1) **Global Memory Coalesscing.** Solution:

```
1  __global__ void VectorSum2(float *x, int n, int m) {
2      int tid = blockDim.x * blockIdx.x + threadIdx.x;
3      if(tid < n) {
4          float sum = 0.0;
5          int stride = gridDim.x * blockDim.x;
6          for(int i = tid; i < n; i += stride)
7              sum += x[m * tid + i];
8          x[tid] = sum;
9          __syncthreads();
10         for(stride = 2; stride < n; stride *= 2){
11             __syncthreads();
12             if((tid % stride) == 0)
13                 x[tid] = x[tid] + x[tid + stride/2];
14         }
15     }// The result is in x[0]
16 }
```

(2) **Global memory accesses in the reduce.** Solution: using shared memory.

```
1  __global__ void VectorSum2(float *x, int n, int m) {
2      int tid = blockDim.x * blockIdx.x + threadIdx.x;
3      if(tid < n) {
4          __shared__ float y[1024];
5          int stride = gridDim.x * blockDim.x;
6          for(int i = tid; i < n; i += stride)
7              y[tid] += x[m * tid + i];
8          __syncthreads();
9          for(stride = 2; stride < n; stride *= 2){
10             __syncthreads();
11             if((tid % stride) == 0)
12                 y[tid] = y[tid] + y[tid + stride/2];
13         }
14     }// The result is in y[0]
15 }
```