Tai Duc Nguyen - ECEC 622 - 06/10/2020

Final Exam

Problem 1

```
int thr_id[5], array_1[1000000], array_2[5];

for(int i = 0; i < 5; i++) {
    pthread_create(&thr_id[i], NULL, add, NULL));
    pthread_join(thr_id[i], NULL);
}

/* Code to compute the final sum from the partial sums stored in array_2 */
...
```

```
/* The add function executed by each thread */
void add(void *arg) {
    /* Compute the partial sum and store in array_2 */
    ...
}
```

Answer

This code above uses the chunking method, with each thread handling 200,000 elements. Hence, the effect of false sharing should be negligible.

Therefore, the only reason for no speed up moving from the serial version to multithreaded version is that the overhead of creating threads dominates the execution time of the program. Also, since the processor only has 4 parallel cores, creating 5 threads means that 2 threads will be run in serial fashion on a core. This may mean that one thread will finish much later than other threads. Hence, in order to reduce thread management overheads, the number of threads used can be reduced from 5 to 4 or to 3.

```
int thr_id[3], array_1[1000000], array_2[3];

for(int i = 0; i < 3; i++) {
    pthread_create(&thr_id[i], NULL, add, NULL));
    pthread_join(thr_id[i], NULL);
}

/* Code to compute the final sum from the partial sums stored in array_2 */
...
```

The code for `void add(void *arg)` will be adjusted according to the number of threads.

Problem 2

```
semaphore resource_1 = 1;
semaphore resource_2 = 1;

void thread_A(void) {
    probe_semaphore(resource_1);
    probe_semaphore(resource_2);

    use_both_resources();

    signal_semaphore(resource_2);
    signal_semaphore(resource_1);
}

void thread_B(void) {
    probe_semaphore(resource_2);
    probe_semaphore(resource_1);

    use_both_resources();

    signal_semaphore(resource_1);
    signal_sempahore(resource_2);
}
```

Answer

The deadlock can be obtained in the following situation:

1. thread_A tries and obtains lock on resource_1
2. thread_B tries and obtains lock on resource_2
3. thread_A tries to obtain lock on resource_2 but thread_B is holding the lock
4. thread_B tries to obtain lock on resource_1 but thread_A is holding the lock

Hence, both threads are dead locked.

A simple fix would be to switch the sequence of obtaining locks in either thread:

```
void thread_B(void) {
    probe_semaphore(resource_1);
    probe_semaphore(resource_2);

    use_both_resources();

    signal_semaphore(resource_2);
    signal_sempahore(resource_1);
}
```

This way, if thread_A obtains lock on resource_1, then thread_B can't obtain lock on resource_1, it will wait for thread_A to release resource_1.

Problem 3

If a CUDA device's SM can take up to 1536 threads and up to four thread blocks, which of the following block configurations would result in the most number of threads in the SM: (a) 128 threads per block; (b) 256 threads per block; (c) 512 threads per block; or(d) 1024 threads per block? Explain your answer clearly.

Answer

If the maximum of thread blocks is 4 and there are 1536 threads total, then:

a. 128 threads/block would result in 12 blocks (VIOLATE MAX TB)
b. 256 threads/block would result in 6 blocks (VIOLATE MAX TB)
c. 512 threads/block would result in 3 blocks
d. 1024 threads/block would result in 2 blocks (VIOLATE MAX THR)

Hence, only configuration (c) make sense since it does not violate the maximum number of thread blocks and the maximum number of threads.

Problem 4

A CUDA programmer says that if they launch a kernel with only 32 threads per block, thay can leave out the __syncthreads() instruction whenever barrier synchronization is needed. Do you think this is a good Idea? Explain.

Answer

It is never a good idea to leave out a barrier sync if such barrier is necessary to maintain the correctness of the code. Race conditions can occur as one thread is writing to the exact memory location that is being read by another thread.

In addition, there is no guarantee that the block size will continue to be 32. The code may break if this number is higher.

Problem 5

A kernel performs 36 floating-point operations and seven 32-bit word global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute or memory bound: (a) Peak FLOPS = 200 GFLOPS, peak memory bandwidth = 100 GB/s; (b) Peak FLOPS = 300 GFLOPS, peak memory bandwidth = 250 GB/s.Explain your answer clearly.

Answer

7 32-bit word = 7*32/4 = 56 bytes
Hence, for each bytes access from memory, the number of FLOPS is: 36/56 = 0.6429

a. Peak FLOPS = 200 GFLOPS, peak memory bandwidth = 100 GB/s:

Hence, the number of FLOPS if each thread uses peak memory bandwidth is:
100 * (36/56) = 64.29 GFLOPS (< 200 GFLOPS)
Therefore, this kernel is memory bound.

b. Peak FLOPS = 300 GFLOPS, peak memory bandwidth = 250 GB/s:

Hence, the number of FLOPS if each thread uses peak memory bandwidth is:
250 * (36/56) = 160.71 GFLOPS (< 300 GFLOPS)
Therefore, this kernel is memory bound.

Problem 6

```
__shared float partialSum[];

unsigned int t = threadIdx.x;
int stride;

for(stride = blockDim.x >> 1; stride > 0; stride = stride >> 1) {
    if(t < stride)
        partialSum[t] += partialSum[t + stride];
        __syncthreads();
}
```

Answer

This code can reduce the accuracy of the result because when an arbitrary small number (i.e 0.00001) is added to large number (i.e 1000) in the case of a running sum, the result is often rounded (i.e: 1000 + 0.00001 = 1000). Hence, in order to obtain more accurate result, it is preferred to use a higher precision type like `double` (64 bit) instead.

Problem 7

Consider performing a 1D tiled convolution using the GPU kernel shown below on an array of size n with a mask of size m using a tile of size t. Answer the following questions: (a) how many blocks are needed?; (b) how many threads per block are needed? (c) how much shared memory is needed in total?

Answer

Array size = n
Mask size = m
Tile size = t

a. If one thread is calculating the conv result for 1 element, then, the number of blocks needed are:

`NUM_BLOCKS = (n + t - 1)/t`

b. Since the tile size is t, then:

`THREAD_BLOCK_SIZE = t`

c. Since the mask size is m, each thread block will maintain a shared float array of:
`SHARED_ARR_SIZE = THREAD_BLOCK_SIZE + MASK_SIZE - 1 = t + m - 1`
Hence, each thread block will maintain `SHARED_ARR_SIZE*8` bytes of memory. Therefore, the total amount of shared memory use is: `SHARED_ARR_SIZE*8*NUM_BLOCKS` bytes

Problem 8

Answer part A

```
dim3 thread_block_size = 512;
int num_thread_blocks = (num_elements + thread_block_size - 1)/thread_block_size;
dim3 execution_grid = (num_thread_blocks, 1)
```

Answer part B

```
__global__ void compute_squares_kernel (float *Ad, float *Rd, int num_elements) {
    int threadX = threadIdx.x;

        int blockX = blockIdx.x;

        int column = blockDim.x * blockX + threadX;

    if (column >= num_elements)
        return

    Rd[column] = Ad[column] * Ad[column];

    return;
}
```