High Performance Computing

Unit 3 Exercises: GPUs

Thread Scheduling:

3.1.- The architecture of a GPU is characterized by:

- The maximum number of threads per block is 512.
- The warp size is 32.
- The number of registers per SM multiprocessor is 8192.
- The maximum number of blocks that can run simultaneously on an SM multiprocessor is 8.
- The maximum number of warps that can run simultaneously on an SM multiprocessor is 24.

For matrix multiplication, which distribution of threads per block (block size) of the following is the most appropriate?

a) 8x8
b) 16x16
c) 32x32

Justify the answer

3.2.- In the previous architecture:

What can be concluded in terms of thread allocation, if an application is simultaneously running 24 Warps on one of the SM multiprocessors?

3.3.- In the previous architecture suppose that:

- To execute (dispatch) all the threads of a warp needs 4 clock cycles.
- A kernel performs a global memory access that occurs every 4 instructions.
- Each access to global memory supposes a latency of 200 cycles.

Estimate the number of Warps the system must be running to hide the memory access penalties.

3.4.- Suppose that a SM (Stream Multiprocessor) has the following characteristics:

- Maximum number or warps per SM = 64
- Maximum number or blocks per SM =32
- Register usage = 256KB
- Available Shared memory = 64KB.

You want to run a kernel, with the number of blocks nBlk and each block with nThr Treads:

Kernel <<< nBlk,bThr >>> (…)

Considering that the kernel code uses shared memory as indicated in the following code:

```
__global__ Kernel (…) {
        __shared__ int A [1024];  // size of "int" is 4 bytes
        int x =4;
        index= blockIdx.x * blockDim.x + threadIdx.x
        for (a = 0, a <MAX, a++) {
           m[a] = 2* A[index+…] * C;
           …
        }
…
}
```

a) Under these conditions, what is the maximum number of Blocks that can be executed when launching this kernel in the SM?

b) When doubling the value of MAX (block index) it is detected when compiling the program that the number of registers used per block goes from 8K to 32K. How has the previous situation changed?

3.5.- A novice CUDA programmer is trying to optimize his first GPU kernel for performance. He/she wants to find the best execution configuration (that is, the grid size, the block size, and the number of threads per block). As he/she allocates one thread per input element, he/she calculates the grid size (i.e., the total number of blocks) as follows. For N input elements, the grid size is [N/block_size] , where block_size is the number of threads per block. The optimizing part will be to figure out which block size produces the best performance by trying 5 different possible block sizes for your GPU (64, 128, 256, 512 and 1024 Threads).

A general recommendation for kernel optimization is to maximize the occupancy of all GPU Stream Multiprocessors (SM). Occupancy is defined as the ratio of active threads to the maximum possible number of threads per SM.

To calculate occupancy, it is necessary to take into account the available resources. It is known that in each SM of his/her GPU:

- The total shared memory is 16KB.
- The total number of 4-byte records is 16384.

In an early version of the kernel code, each thread uses 2 4-byte elements in shared memory. Also, each block, regardless of its size, needs an additional 16 4-byte elements in shared memory for inter-thread communication.

To determine register usage, the number of registers needed by each Thread is investigated by using a compiler flag and it is found that each thread in the first kernel version uses 9 registers.

a) Assuming that the number of blocks is limited by shared memory, which of the above options (64, 128, 256, 512 and 1024 Threads/Block) is the most suitable.

Other GPU limitations may modify the above choice. Hardware restrictions of each SM.

- The maximum number of blocks per SM is 8.
- The maximum number of threads per SM is 2048.

b) With these new restrictions, which of the above options (64, 128, 256, 512 and 1024 Threads/Block) is the most appropriate?

Block_size= num Threads/block	Blocks /SM	Treads/SM	

c) Considering the restrictions of section b), how many registers are used in each of the options (64, 128, 256, 512 and 1024 Threads/Block) and which one is closer to reaching the limitation by registers?

Block_size=numThreads/block	Blocks /SM	Treads/SM	Register/block	Register/SM

d) The performance obtained by the first version of the kernel does not meet the necessary acceleration. Therefore, the programmer writes a second version of the kernel that reduces the number of instructions at the expense of using one more register per thread. What would be the highest occupancy for the second kernel? What block size do you get?

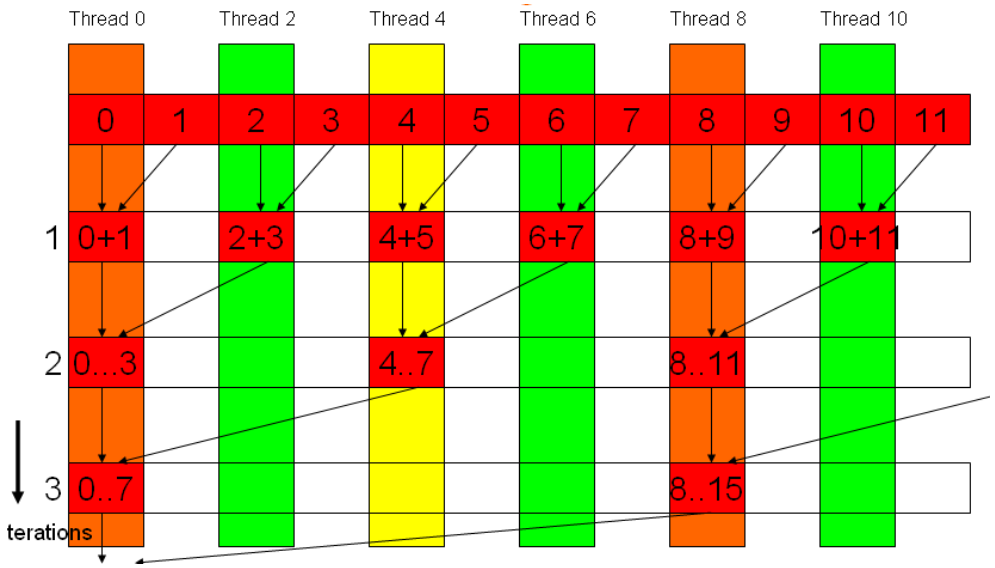
Block_size=numThreads/block	Blocks /SM	Treads/SM	Register/block	Register/SM

3.6.- What kind of incorrect behavior can occur if we forget to use the __syncthreads() instruction in the kernel shown below?

There are two calls to the function __syncthreads() justifies each of them.

```
__global__
void matrix_mul_kernel(float* Md, float* Nd, float* Pd, const int cWidth)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx_ = blockIdx.x;
  int by_ = blockIdx.y;
  int tx_ = threadIdx.x;
  int ty_ = threadIdx.y;

  int row_ = by_ * TILE_WIDTH + ty;
  int col_ = bx_ * TILE_WIDTH + tx;

  float p_value_ = 0.0f;

  for (int m = 0; m < cWidth / TILE_WIDTH; ++m)
  {
    Mds[ty_][tx_] = Md[row_ * cWidth + (m * TILE_WIDTH + tx_)];
    Nds[ty_][tx_] = Nd[(m * TILE_WIDTH + ty_) * cWidth + col_];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      p_value_ += Mds[ty_][k] * Nds[k][tx_];

    __syncthreads();
  }

  Pd[row_ * cWidth + col_] = p_value_;
}
```

3.7.- Reduction is needed on a system using a GPU to store the sum of all elements of a vector in element 0 of the same vector.



A programmer makes a first version of the program that performs the reduction based on the following conditions:

- The original vector is stored in the global memory of the GPU.
- Shared memory is used to store the partial sum.
- Each iteration performs a partial sum according to the figure.
- The final solution is stored in element 0.

The part of the code where the vector is loaded is the following:

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
       stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
       partialSum[t] += partialSum[t+stride];
}
```

For an execution on a very large vector (>10,000), it is parallelized with a block size of 512, and for this, if necessary, additional elements are added to the vector of zero value.

It is requested:

a) Explain how the code works indicating the improvement in performance compared to a non-parallelized version.
b) Detail how the system behaves in terms of efficiency.
c) Indicate, justifying your answer, how many warps are active per iteration.

d) Add any improvement to the above code that improves the performance of the application. Represent with a scheme similar to the figure the execution of the new code and justify the reason why it is expected to improve.

Detail how it changes, if it is the case, the execution of threads, warps and memory access.