046274

GPU- accelerated systems

מטסڕ אביכ, ﺪشئ"ו
22.09.2016

מרצה: מרק ויילברשטיין

מבחין סופי - מועד ב',

נזור כ- חורר חתוב או ממושב
כ- 110 מקודקוח לא ייודע פסוקר.

אדר הままה: שלוש שעות

מקדם כ- 11 דפים

cל התשובות ייכבדו בתפריט המ炻ינה

תשובה לשאלות פונות להנות ממתקים המפורעים לפני כל שאלה

תשובה לשאלות אסכנאואית יパー融资租赁ה ב-שון תשובות על ידי עלוי

046274 - מועד ב', תשע"ו.
Question #1 (30 points)

Given an array of 4096 items of the type unsigned char. Each item is in the range [0,255], we want to create a histogram of the values in the array. A histogram is an array of integers of size 256. Cell number \(i\) contains the number of elements in the input array which have the value \(i\).

For example, if 10 cells in the input array have the value 16, then histogram[16] = 10.

The histogram should be created in two stages.

Stage 1 - First kernel - compute_partial_histograms:
The input array is divided between 4 threadblocks of 1024 threads each. Each threadblock should create a histogram for only its part of the array.

Stage 2 - Second kernel - merge_histograms:
A second kernel should merge the 4 histograms into one final histogram.

1. The CPU code is given below. Implement the kernels: compute_partial_histograms and merge_histograms

```c
int main() {
    unsigned char *gpu_input_array = alloc_and_populate();
    int *partial_histograms;
    int *final_histogram;
    cudaMemcpy(partial_histograms, 256 * 4 * sizeof(int));
    cudaMemcpy(final_histogram, 256 * sizeof(int));
    compute_partial_histograms<<<4, 1024>>>(gpu_input_array, partial_histograms);
    merge_histograms<<<1, 256>>>(partial_histograms, final_histogram);
    cudaDeviceSynchronize();
    return 0;
}
```
__global__ void compute_partial_histograms (unsigned char *input_array,
                int *partial_histograms) {

    int my_idx = blockIdx.x * 1024 + threadIdx.x;
    int hist_offset = blockIdx.x * 256;
    __shared__ int histogram_local[256];

    if (threadIdx.x < 256) histogram_local[threadIdx.x] = 0;
    __syncthreads();

    unsigned char my_item = input_array[my_idx];
    atomicAdd(&histogram_local[my_item], 1); /* potential bank conflicts here */
    __syncthreads();

    if (threadIdx.x < 256) {
        partial_histograms[hist_offset + threadIdx.x] = histogram_local[threadIdx.x];
    }
}

__global__ void
merge_histograms(int *partial_histograms, int* final_histogram)
{
    int tid = threadIdx.x;
    for (int i = 0; i < 4; i++) {
        final_histogram[tid] += partial_histograms[i * 256 + tid];
    }
}
2. Does your code have **divergence**? If yes, mark that part of the code.

**Answer:** No

3. Does your code perform **uncoalesced** memory accesses? If yes, mark that part of the code.

**Answer:** No. (But if shared memory is not used then first kernel will have uncoalesced accesses).

4. Does your code have **bank conflicts**? If yes, mark that part of the code.

**Answer:** Potentially yes. Code marked with a comment.
Question #2 (10 points)
Consistency

Consider the following code executed by two threads in the same threadblock

```c
__global__ int x;
__shared__ int y; // initialized to zero
int z;

if (tid == T2){ y = 1; }
if (y == 1) { z = 1; } else { z = 2; }
__syncthreads();
if (tid == T1) x = z;
```

Choose all the correct answers

**Answer:**

a) The kernel will deadlock
b) if T1 and T2 are in the same warp, x=1
c) if T1 and T2 are in the same warp, x=2
d) if T1 and T2 are in different warps, x=1
e) if T1 and T2 are in different warps, x=2
f) None of the above answers is correct
An engineer wrote the following CUDA code.

```c
/* memory pointed by x is initialized to zero before the kernel is called */
__global__ my_kernel(int *x) {
    /* keep trying to acquire lock */
    while (atomicCAS(x, 0, 1) == 1) {}  
      
    do_work();
    
    *x = 0;

    __theardfence();
}
```

The documentation of atomicCAS (Atomic Compare & Swap) is given:

```c
int atomicCAS(int *address, int compare, int val)
Atomeically perform the following operations:
Reads the integer pointed by address, if it is equal to compare it sets its new value to val, otherwise it doesn’t change its value.
In either cases, the function returns the old value.
```

The code was invoked on two different computers, on the first computer it was invoked with 1 warp (32 threads), and on the other with 2 warps (64 threads). Which of the following is correct?

1) The code will work just fine in both cases.
2) The code will deadlock in both cases.
3) The code will deadlock with 2 warps but will not deadlock with 1 warp.
4) The behavior will be random: The code will deadlock only sometimes in both cases.
Question #4 (10 points)
ActivePointers

Assuming no other threads were/are running in the system, write for each of the marked lines in the following code whether it will have: No Page Fault / Minor Page Fault / Major Page Fault

/* assume fd is a valid file descriptor */
Aptr<float> ptr = gvmmap(PAGE_SIZE * 100, O_RDONLY, fd, 0);  /*line 1*/
ptr = ptr + PAGE_SIZE;  /*line 2*/
float x = *ptr;  /*line 3*/
ptr = ptr + PAGE_SIZE;
ptr = ptr - PAGE_SIZE;  /*line 4*/
float y = *ptr;  /*line 5*/

Line 1: No Page Fault
Line 2: No Page Fault
Line 3: Major Page Fault
Line 4: No Page Fault
Line 5: Minor Page Fault

Question #5 (10 points)
Occupancy

A kernel uses 20 registers per thread, 4KB shared memory per threadblock, 1024 threads per threadblock.
An SM can run a maximum number of 2048 threads, a maximum number of 8 threadblocks, has 8 KB of shared memory, and 32K registers.

The kernel is invoked on the GPU with a very large number of threadblocks.

1. What is the occupancy of the GPU?
   50%. SM capacity is 2048 threads, but since 1024 threads already use 20K registers out of 32K it is not possible to run more than 1 threadblock per SM. Hence each SM will run only 1024 threads (1 threadblock), i.e. 50% of its capacity.
Question #6 (10 points)
Compute and DMA

Given a GPU with a single SM which can run only 1 threadblock at a time, and a single DMA engine.
We want to run the following computations and data movements on two streams:

Stream1: Memcpy1, Kernel1, Memcpy2
Stream2: Kernel2, Memcpy3

Memcpy1 takes 300 usec.
Memcpy2 takes 200 usec.
Memcpy3 takes 400 usec.
Kernel1 takes 300 usec.
Kernel2 takes 100 usec.

kernel1 and Kernel2 are both single threadblock kernels.

1. What is the best possible runtime of the whole program (i.e. Lower bound on run time)?

900 usec

2. Describe the optimal running order which gives the above time? (Which operation runs at which time).

Timeline:
Question #7 (10 points)

GPUfs

A programmer writes a GPU program which reads data from many files using GPUfs. Each read() accesses a different page. Prefetching is disabled. The programmer then runs the program once until termination, and then runs it again (The two runs are functionally identical). The first run is in Process 1 and the second run is in Process 2.

Choose all the correct answers:
1) All reads of process 1 will miss in GPUfs buffer cache and all reads of process 2 will hit in GPUfs buffer cache.
2) All reads of both processes will miss in GPUfs buffer cache.
3) Whether reads of process 1 will miss or hit depends on the size of GPUfs buffer cache.
4) Whether reads of process 2 will miss or hit depends on the size of GPUfs buffer cache.

Question #8 (10 points)

GPUnet

An engineer wanted to create a unified CPU-GPU socket, so that both CPU and GPU can perform send() and recv() operation on the same socket, such that:

- When a recv() is called on this socket from the CPU, the data is received from the network directly to CPU memory, and if recv() is called from GPU, the data is received from the network directly to GPU memory.
- When send() is called on this socket from the CPU, the data is sent to the network directly from CPU memory, and when send() is called from GPU, the data will be sent directly from GPU memory to the network.

Choose the correct answer and provide a short explanation:

1) It is possible to implement both send() and recv() as described.
2) It is not possible to implement neither send() nor recv() as described.
3) It is possible to implement send() but not recv() as described.
4) It is possible to implement recv() but not send() as described.

Explanation:

A packet is sent when send() is called, hence it's possible to tell the network card where to take the packet from.

On the other hand, we have no control on when a new packet will come from the network. recv() merely brings the packet from memory to the user, but the network card has decided where to place the packet long before recv() was called. When the network card receives a new packet it does not know whether the next recv() call will be from CPU or GPU.
Question #9 (10 points)
MapD

MapD uses Column-store (rather than the more traditional Row-store).

1. Explain the difference between column-store and row-store
   In Row-store, each row resided is a contiguous memory area. In column-store each column resided in a contiguous memory area.

2. Why does MapD use column-store?
   Two reasons:
   1. Column-store makes it easier to move whole columns to the GPU memory. This allows MapD to move only columns which are accessed by the GPU to GPU memory instead of moving the whole row.
   2. Allows coalesced access. If different threads need to access same field of different records, it is better that these fields be close in memory. Similar to “Struct of Arrays” representation (see next question)

3. How is column-store related to “Struct of Arrays” representation?
   A row is like a struct: A set of fields. Row-store is like an “Array of Structs”, rows (structs) are stored one after the other as in an array.
   Column-store stores arrays of fields. First it stores first field of all structs (creating an array of first field), then second field of all structs and so on, which is exactly what “Struct of Arrays” means.