ZJU-UIUC Institute
Second Exam, ECE 408

Thursday 23 May 2019

Name (pinyin and Hanzi):
Student ID:

- **Be sure that your exam booklet has NINE pages.**
- **Write your name and Student ID on the first page.**
- **Do not tear the exam apart.**
- **This is a closed book exam. You may <u>not</u> use a calculator.**
- **You are allowed one handwritten A4 sheet of notes (both sides).**
- **Absolutely no interaction between students is allowed.**
- **Show all work, and clearly indicate any assumptions that you make.**
- **Don't panic, and good luck!**

Problem 1	15 points	_____
Problem 2	26 points	_____
Problem 3	29 points	_____
Problem 4	30 points	_____
Total	100 points	_____

Problem 1 (15 points): Short Answer Questions

1. **(5 points)** Recall the notion of pinned memory, which is guaranteed to be physically resident and can never be swapped out to disk. Just as GPUs benefit from use of pinned memory to ensure that DMA engines are able to make progress without the need to wait for page faults, MPI communication benefits from use of pinned memory to ensure that DMA engines (this time on the Network Interface Card, or NIC) are able to make progress without the need to wait for page faults.

   So why is the supercomputing community interested in having NICs read/write MPI communication data directly from/into GPU memory?

2. **(5 points)** A GPU with 200 GB/s memory bandwidth is used to compute a sparse matrix-vector multiplication using data packed in the ELL format (the padded, transposed form of the compressed sparse row/CSR format). Assuming that the original matrix has 5 million non-zero elements (single-precision floating-point values) with an average of K non-zero elements per row and a max of N non-zero elements per row, how much time does the GPU require to read the matrix in for computation?

3. **(5 points)** Prof. Butala has a 1200 MB array that he wants to process with a GPU. After parallelizing the code, he finds that computation time alone, with data already in GPU memory, is 0.05 seconds, a factor of 70× faster than the sequential code. However, he must also account for moving the array from host memory to GPU memory and back over his system's PCIe Gen3 ×8 interconnect. What is the speedup when this extra time is included? Show your work.

Problem 2 (26 points): Scans and Reductions

1. **(5 points)** Explain how the use of double-buffering aids performance in the Kogge-Stone scan algorithm.

2. **(5 points)** The Brent-Kung scan algorithm performs many fewer operations than does the Kogge-Stone algorithm. Why might someone nevertheless choose to use the Kogge-Stone algorithm rather than the Brent-Kung algorithm?

3. Let's review the code for a reduction using MAX on floating-point values. On a CPU, we have something like

```
float data[N_ELTS];

int   idx;
float max;

for (idx = 1, max = data[0]; N_ELTS > idx; idx++) {
    if (max < data[idx]) {
        max = data[idx];
    }
}
// max now holds the largest value in the array data
```

(Continued on the next page.)

Problem 2, continued:

On a GPU, we have something similar to the code from the class slides…

```
__global__ void
reductionKernel (float* input, float* result)
{
    __shared__ float sum[2 * BLOCK_SIZE];
    int t      = threadIdx.x;
    int start  = 2 * blockIdx.x * blockDim.x;
    int stride;

    sum[t]              = input[start + t];

    sum[blockDim.x + t] = input[start + blockDim.x + t];

    for (stride = blockDim.x ; 1 <= stride ; stride >>= 1) {

        if (sum[t] < sum[t + stride] && t < stride) {

            sum[t] = sum[t + stride];

        }

    }
    if (0 == t) {

        result[blockIdx.x] = sum[0];

    }
}
```

a. **(4 points)** Oops! Prof. Lumetta left out the **`__syncthreads`** calls. Add them to the kernel above where necessary.

b. **(4 points)** If we ignore the **`for`** loop part of the sequential code, all but one small part of the GPU kernel is parallel overhead—code that is unnecessary. What is that one part?

c. **(4 points)** The sequential work (your answer to **part (b)**) should be done reasonably efficiently in a correctly implemented reduction kernel, but Prof. Lumetta introduced a bug that made it inefficient in the kernel above. Explain what needs to be done to make it efficient.

d. **(4 points)** Assuming your correction from **part (c)** above, and assuming that BLOCK_SIZE is 512, how many warps experience control divergence during the execution of the kernel above (assume that warp size is 32)?

Problem 3 (29 points): Parallelism with CUDA Streams

1. A friend in ECE408 implemented histogram code using privatization to allow each thread block to work first within shared memory. After learning about CUDA streams, the friend got extremely excited and decided to rewrite the histogram code to break the dataset into many segments and to use multiple streams to overlap computation of each segment with PCIe transmission of data for the next segment.

But now the friend is worried. The latest GPUs allow multiple kernels to execute in parallel, so multiple segments from the input data may be computed **at the same time!** Your friend is struggling to figure out how to express dependence from segment N to segment N+1 so that CUDA delays execution of segment N+1 until segment N's kernel finishes.

a. **(5 points)** Is your friend's concern valid? Explain your answer, and, if appropriate, suggest a solution.

b. **(5 points)** The same friend has also implemented a version of sparse matrix-vector multiply using the hybrid ELL-COO approach. As you should recall, in this approach, some number K of non-zero elements per row are padded and then transposed (the ELL format). For those rows with more than K non-zero elements, your friend creates an array indexing those rows along with a padded, transposed copy of the next K non-zero elements, then launches a second kernel in which each thread reads one element of the first kernel's output from global memory, adds the dot product of the new non-zero elements for that row, and writes the result back to global memory.

Is your friend's parallel kernel concern valid for this application? Explain your answer, and, if appropriate, suggest a solution.

2. **(5 points)** Another friend is also excited about streams and is eager to convince NVIDIA to make changes. This friend points out that the remaining overhead with streams is due to boundary effects and overhead. Consider the following: if one breaks an input set into many segments, transmission of the first segment overlaps with nothing: no kernel is executing, and nothing is crossing PCI in the other direction. And when the last segment finishes computation and the output data must be copied back to the host, the same situation arises.

To address this problem, the friend wants to make segments as small as possible. But data transfers and kernel launches are not instantaneous, so the friend argues that by making these overheads as small as possible, one enables much shorter segment lengths and thus dramatically improves GPU performance.

Is your friend correct? Or is there another issue that comes up when using shorter segments? Explain your answer.

Problem 3, continued:

3. It's a big day for streams. A third friend wants to convince NVIDIA to implement fair sharing for all resources. The friend has implemented a simulator to show the advantages, but is confused by some results and wants your help to understand them.

In particular, consider a computation based on four segments of data, each using a separate CUDA stream. When executed using all resources on the GPU, each of the segments requires 1 second of data transfer from host to device, 1 second of computation, and 1 second of data transfer from device to host.

a. **(5 points)** Draw a picture showing how the various pieces overlap in time with a GPU that allows only one operation at a time on any resource (transfer to device, transfer from device, and kernel computation). Time should go left to right. Name the pieces for segment N as follows: INPUT.N, COMP.N, and OUTPUT.N.

b. **(5 points)** Now draw a picture showing how the various pieces overlap in time with a GPU that allows fair sharing of all resources when more than one stream requests a resource. Assume that the time required to initiate operations is negligible.

c. **(4 points)** Which is better? Explain why.

Problem 4 (30 points): Writing Parallel Code to Do Stuff™

The sequential code below builds a hash table on **`thing_t`**'s based on each **`thing_t`**'s **`identity`**, an ASCII string. Assume for simplicity that the hash function **`HASH`** is implemented as preprocessor code, and that it returns a number between 0 and **`MAX_HASH - 1`** based on the entire **`identity`** string.

```
typedef struct thing_t thing_t;
struct thing_t {
    char         identity[64];
    unsigned int unique_id;
    // lots of other fields
    thing_t*     hash_next;
};

static thing_t data[N_THINGS];
static thing_t* bins[MAX_HASH]; // initialized to all NULLs

void
build_hash ()
{
    for (int idx = 0; N_THINGS > idx; idx++) {
        int h = HASH (&data[idx]);
        data[idx].hash_next = bins[h];
        bins[h] = &data[idx];
    }
}
```

A programmer wants to build the hash table in parallel on a GPU, then use the table on the CPU. The programmer has written the necessary device memory allocation calls and CUDA memory copies to copy **`data`** to GPU memory and to copy **`bins`** back to host memory, and is now trying to decide between alternatives for parallelization.

(Parallel versions start on the next page.)

Problem 4, continued:

In the GPU kernel implementation below, the programmer has changed the type of **`thing_t`**'s **`hash_next`** field and the hash table **`bins`**. Assume that **`bins`** (and **`d_bins`**) are now initialized with all -1 values.

1. **(5 points)** Explain why these changes are necessary.

2. **(5 points)** Will the changes affect the performance of CPU code that uses the resulting hash table? How?

3. **(5 points)** What is the main performance issue for the kernel below?

```
typedef struct thing_t thing_t;
struct thing_t {
    char         identity[64];
    unsigned int unique_id;
    // lots of other fields
    int          hash_next;    // CHANGED HERE
};

__global__ buildHashKernel (thing_t* d_data, int* d_bins) // in device memory
{
    // MAX_HASH threads
    int t = threadIdx.x;
    for (int idx = 0; N_THINGS > idx; idx++) {
        int h = HASH (&d_data[idx]);
        if (t == h) {
            d_data[idx].hash_next = d_bins[h];
            d_bins[h] = idx;
        }
    }
}
```

Problem 4, continued:

In the GPU kernel implementation below, the programmer has parallelized the computation of the **`HASH`** function for different **`thing_t`**'s. The **`atomicCAS`** (compare and swap) call compares the value at an address (the first argument) with the expected value (the second argument). If the two match, the value at the address is replaced with a new value (the third argument). Regardless, the value currently at the address is returned.

4. **(5 points)** Other than the use of global atomic operations, what is the main performance issue for the kernel below?

5. **(5 points)** What changes are necessary to address the problem that you mention in **part (4)** above? Will those changes necessitate changes to the sequential code that uses the **`thing_t`**'s?

6. **(5 points)** The programmer wants to make use of shared memory, but isn't sure as to how to privatize the hash table. Assume that **`MAX_HASH`** is small enough to allow **`d_bins`** to fit easily into shared memory. What's the hard part? Can it be solved? How?

```
typedef struct thing_t thing_t;
struct thing_t {
    char         identity[64];
    unsigned int unique_id;
    // lots of other fields
    int          hash_next;
};
__global__ buildHashKernel (thing_t* d_data, int* d_bins) // in device memory
{
    int total = gridDim.x * blockDim.x;
    int t = threadIdx.x + blockIdx.x * blockDim.x;
    for (int idx = t; N_THINGS > idx; idx += total) {
        int h = HASH (&d_data[idx]);
        do {
            int link = d_bins[h];
            d_data[idx].hash_next = link;
        } while (link != atomicCAS (&d_bins[h], link, idx));
    }
}
```