Accelerators and Accelerated Systems

 tudem, תשמ"ז
 9.07.2017

מרצה: מרק דר '| ס텟
מתרגל: אמיר וותד

מבחן סופי - מועד А'

מותר כל חומר כתוב או ממוחשב

א durée המבחן: שלוש שעות

בVerdana 12 ש欠缺

בנכתב 105 נקודות

כל התשובות יписанו במכתבת המוחיבה

תשובה לישאלות יינתנה במכתבת המוחיבהiano

046278
Question #1 (30 points)
CUDA Programming

We want to implement a simple IP router using the GPU:
Given a machine with multiple network interfaces (we will call them ports) which acts as a router: for every received packet it decides to which port to send the packet. The port is determined by the router by finding the destination IP address in the routing table. The routing table is defined as follows:

```c
struct routing_entry_t {
    unsigned int ip;
    unsigned int mask;
    int port;
};
struct routing_entry_t table[N_ROWS];
```

The lookup algorithm works as follows: for a given IP `ip` it tries to find the first matching entry in the routing table by scanning the table from the lowest index to the highest index. The table might contain multiple matching entries, but only the first one is used. The entry `i` is considered a match if `(ip & table[i].mask) == table[i].ip`. For the matching entry `i` the router returns `table[i].port` as its output. Last row in the table always has IP = 0, Mask = 0 (So it is guaranteed that at least this row will always match).

Below is the example table. For IP=0x2468AC23 there are 3 entries that match (2nd, 4th and 5th entry), but since the 2nd entry has lower index, port 3 is returned.

<table>
<thead>
<tr>
<th>IP</th>
<th>Mask</th>
<th>Output port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345678</td>
<td>0xFFFFFFFF</td>
<td>0</td>
</tr>
<tr>
<td>0x2468AC00</td>
<td>0xFFFFFFFF00</td>
<td>3</td>
</tr>
<tr>
<td>0x12340000</td>
<td>0xFFFF0000</td>
<td>1</td>
</tr>
<tr>
<td>0x24000000</td>
<td>0xFF000000</td>
<td>1</td>
</tr>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>2</td>
</tr>
</tbody>
</table>

In the GPU implementation, the router invokes a GPU kernel called `route()` which computes the port number. The routing table is in the GPU global memory.
1. (20 points) Write parallel GPU code that implements the `route()` kernel. Assume we invoke 1 kernel for each packet. Each kernel is invoked with 1 threadblock of N_ROWS threads. (N_ROWS <= 1024). The signature of the kernel is given below (ip points to the packet’s IP address, port points to the memory allocated for the kernel’s output). You must not use atomics in your implementation. Assume that reduce, exclusive scan, and inclusive scan (with any binary operator you need) is implemented and available for you to use (you don’t have to reimplement them).

```c
__global__
void route(struct routing_entry_t *table, int *ip, int *port) {
```

2. **(5 points)** An engineer suggested that for the sake of performance it is better to store the routing table as a struct of arrays (instead of an array of structs):

```c
struct routing_table_t {
    unsigned int ip[N_ROWS];
    unsigned int mask[N_ROWS];
    int port[N_ROWS];
};
```

Explain why.

3. **(5 points)** In addition to the optimization in (2), another engineer noticed that the routing table is small enough to fit in the shared memory of each threadblock. But she claimed that in order for the shared memory to improve performance it is necessary to batch multiple packet lookups together (i.e., invoke `route()` with an array of IPs). Explain why.
Question #2 (15 points)
Accelerators

We want to design a new special accelerator, called FHU (Fast Hash Unit) for fast lookup and insertion of values in a hash table. The hash table is a simple array, each entry is a bucket containing the pointer to the linked list of the hash table entries in the bucket.

The FHU does not have its own memory, and instead it operates on memory of another processor (i.e., GPU or CPU, or both) by mapping remote memory into its own address space. FHU performs thousands of parallel updates and properly handles hash table contentions among the requests. It connects to the rest of the system via PCIe. In our setup PCIe does not support atomic operations.

A processor may offload multiple independent updates/lookups to FHU. It is the responsibility of the programmer to impose ordering on dependent updates/lookups.

System parameters: PCIe bandwidth= 16GB/s; CPU DRAM bandwidth/core=16GB/s; GPU GDDR bandwidth/GPU=600GB/s.

1. (5 points): There are two alternatives for designing the software interface to interact with the FHU from all the processes in the system: (a) single task queue (b) multiple QP/CQ queues. Explain which approach you would suggest to the FHU designer and why.

2. (5 points) The programmer revealed that accessing the hash table from another processor while it is in use by the FHU may produce incorrect results or segfault. List at least two reasons why?
3. **(5 points)** The designer runs an application with thousands of independent hash table updates. She was surprised to observe a slowdown when comparing FHU to the optimized GPU kernel having the same functionality, but saw significant speedup over the similar workload running on the CPU. Why?

(Hint: recall that hash table updates and lookups may require multiple dependent memory accesses. Consider the main bottlenecks.)
Question #3 (10 points)
GPU Synchronization and Consistency

We want to implement a counter in CUDA. Unfortunately our GPU doesn’t support any Atomic operations except of atomicCAS (Compare And Swap) documented below:

```c
int atomicCAS(int *address, int compare, int val)
```

Atomically 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.

Below is a wrong implementation of the counter using atomicCAS in CUDA. The intention is for `counter_increment()` to be called by any thread which wants to increment the counter.

```c
/* memory pointed by mutex and counter is initialized to zero before any GPU code is running */
__device__ counter_increment(int *mutex, int *counter) {
    /* keep trying to acquire lock */
    while (atomicCAS(mutex, 0, 1) == 1) {}

    (*counter) ++;

    *mutex = 0;

    __threadfence();
}
```
1. **(5 points)** List all the problems in this implementation?

2. **(5 points)** Rewrite the code so it works. (Only atomicCAS can be used for atomic operations):
Question #4 (10 points)

PCIe

There are two different mechanisms to copy data from CPU memory to GPU memory. List both, explain the advantages and disadvantages of each, and provide use cases demonstrating the best use of each one.
Question #5 (20 points)
Producer-Consumer

Assume a new architecture in which the GPU and the CPU are connected via a new interconnect that works as follows: when reading a mapped memory from the remote processor, it immediately reads the value (possibly an old one). There is no guarantee that the most updated value will be ever read. When writing into mapped memory, it behaves the same as PCIe.

1. (10 points) Explain why the standard producer-consumer queue implementation as learned in the class would not work.

2. (10 points) Explain how you would implement it correctly. For example, the form of the answer could be: “the implementation is the same as with PCIe but with the tail update performed before writing the data.”
Question #6 (10 points)

GPU Scheduling

We are given an application, called RMP, which computes a matrix product of random square matrices. To run, one specifies matrix dimension as an integer up to 1000. For example, RMP 100 will compute a product of two matrices 100x100. RMP prints the results to stdout, and prints one line “END RUN” <process ID> right before termination. RMP can be implemented as a single GPU kernel or as a single CPU function.

The machine is equipped with a single GPU and a single CPU core. However, we do not know whether RMP uses the GPU, and our goal is to figure that out.

Unfortunately, the system administrator explicitly disabled invocation of binary besides RMP, and any utility that collects system performance statistics (i.e., top, nvidia-smi). The only operation you can do is to invoke RMP with different legal inputs as many times as you like.

Assumptions.
1. GPU memory is enough to store 3 matrices 1000x1000.
2. You cannot measure the time it takes to perform the run
3. RMP does not change the choice of the processor: it either runs on the GPU or on the CPU for all invocations.

Hint: you are allowed to use all the standard shell operations, i.e., redirection, background execution, signals, priority.

Please write the protocol comprised of invocations of RPM (possibly with different parameters) that will allow you to determine with 100% certainty that it uses a GPU.
Question #7 (10 points)
Programmable SSD

Willow allows to implement filesystem calls such as read() and write() on the SSD. We run 2 programs, A and B, and find that program A has a speedup when using Willow read calls, while program B has a slowdown when using Willow read calls. After some examination, we find that program A randomly reads data from a file of size 10K pages (reads each page once in random order), while program B repeatedly reads only the first page of this file.

Answer the following questions.

1. (5 points) A has a speedup when using Willow reads because:

2. (5 points) B has a slowdown when using Willow reads because: