Accelerators and Accelerated Systems

סמסטר אביב, תשע"ז
9.07.2017

מרצה: מרק וולברשטיין
מתורגן: אמיר והד

מבחן סופי - מועד א’, פתרון

מותר כל חומר כתוב או ממוחשב
אורים המבחנים: שלוש שעות
בדקו כי במחברת 12 שורות
囱ות 105 שורות
כל התרשויות ייכתבו במחברת המ pobliית
תשובה לשאלות יינתנו במחברת המ溥ית ל atol צל שאלות

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) {

    int tid = threadIdx.x;
    char p;
    __shared__ int p_arr[N_ROWS];

    p = (*ip & table[tid].mask) == table[tid].ip;
    p_arr[tid] = p;
    __syncthreads();

    inclusive_scan_sum(p_arr, N_ROWS); /* p_arr is in-out */
    __syncthreads();

    if ((p_arr[tid] == 1) && p) {
        *port = table[tid].port;
    }

    /* possible also to solve using exclusive scan or reduce */
}
```
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.

Struct of Arrays will make access to global memory coalesced (all IPs are next to each other in memory, same for Masks).

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.

Shared memory lives until kernel terminates. Loading the table to shared memory to access each item only once adds overhead with no benefit. (We still read the table from global memory, but now in addition we need to write it to and read it from shared memory.) In order to justify the overhead, we want to access shared memory many times after loading it from global memory, hence batching.

(An alternative to batching would be using persistent threadblocks, similar to homework 2, where the threadblock lives forever and gets its tasks via a CPU-GPU queue. But the engineer in this question didn’t take our course).
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.

   Multiple QPs/CQs:
   1. This allows us to allocate a different QP/CQ to each process, giving us isolation between processes without the need of the OS to enforce it. This way we can bypass the OS kernel, removing its performance overhead, while still having isolation between processes. (The terminology used, QP and CQ, is borrowed from Infiniband, and was a hint to think in this direction)
   2. Allows to have multiple QPs/CQs per process, so that the process can place independent commands in different QPs. (Similar to CUDA streams).
   3. Without atomics over PCIe, the CPU and GPU may use its own QP/CQ without the need to synchronize
   4. Similar to 3, but for multiple CPU processes

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?

   1. Data race: accessing the hash table while it is being modified by the FHU might result in the CPU seeing the hash table in an illegal state. In order to perform changes in a safe way the FHU would have needed to use PCIe atomics, which are not supported in this question.
   2. Different address space: Items in each bucket are chained as a linked list. The pointers in the linked list can be either: physical addresses or virtual addresses in the **memory space of the FHU**. In either of these cases, these pointers have no meaning in the context of a CPU process. Trying to access them will either cause a segfault or read from unrelated memory.
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.)

This question focuses on the latency hiding of FHU and GPU, but the lack of there of in CPU. Both FHU and GPU can handle thousands of independent updates concurrently, but FHU is bottlenecked on PCIe memory, whereas GPU’s bandwidth to GDDR is much higher. That’s why FHU<<GPU.

CPU cannot handle so many concurrent requests, and for every access to the hash it is hung on the *latency* of pointer chasing in the linked list. FHU managed to hide that latency completely. Therefore, CPU<<FHU

This question was sometimes misunderstood. The question was why CPU<<FHU<<GPU. Surprisingly, some students understood it exactly in the opposite way, and managed to come up with arguments, which certainly made no sense.
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?

   1. Deadlock if called by different threads in the same warp. One thread takes the lock, others are still busy-waiting. The thread which has the lock cannot proceed because threads in a warp execute in lock-step. The lock is never released and all threads are stuck.
   
   2. No memory barrier between counter increment and mutex release. These memory writes might be reordered, resulting in data races and therefore incorrect value of the counter.

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

   ```c
   __device__ counter_increment(int *mutex, int *counter) {
   char done = 0;
   while (! done) {
       done = (atomicCAS(mutex, 0, 1) == 0);
       if (done) {
           (*counter) ++;
           __threadfence();
           atomicCAS(mutex, 1, 0);
       }
   }
   }
   ```
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.

1. DMA. Good for large transfers, as it can perform large memory reads at once. It has an overhead of setup (preparing the gather entry and invocation of the engine), so it is not worth using for small transfers where the overhead dominates. (recall homework 1, where we saw that DMA performance is not proportional to data size, time per byte was orders of magnitude lower (better) for large transfers compared to small transfers. The reason is this overhead).

2. MMIO (Memory Mapped I/O). Allows the CPU to perform memory writes (stores) to the GPU’s memory via its “BAR”. CPU’s memory writes are limited in size (32bit / 64bit), and each write is a CPU instruction (store) which might take a few CPU cycles. It is worth using for small data size as it doesn’t have the DMA setup overhead, but not for large data, as it will need many CPU cycles and will generate many small PCIe write transactions, reducing its performance.

Notice also that for weak CPUs (e.g. Willow’s SPU’s) this approach becomes less and less appealing even for small transfers, since the clock rate of the CPU governs the rate in which these writes are executed. For this reason DMA was a critical design element in Willow.

This question was sometimes misunderstood. Some students answered “reads from GPU vs. writes to GPU”. This is the same mechanism, however.

Another incorrect answer was “shared memory via cudaHostRegister”. This was not correct because this command maps CPU memory into GPU address space, and therefore does not copy memory from CPU to GPU.
**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.

   If producer index (tail) is stored in the producer, consumer might never see it advance, and think that there are no new tasks. ⇒ No progress.
   If producer index is stored in the consumer, producer might read an old value of it over and over, overriding unconsumed tasks. (keeps writing tasks to the same location) ⇒ Correctness is broken.
   (Similar argument goes for the consumer index.)

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.”

   We need to avoid remote reads since remote reads are essentially broken:
   1. We store the buffer in the consumer’s memory. Producer writes go through the interconnect and consumer reads are local.
   2. Producer and consumer indexes are duplicated (producer has a copy of each and so is consumer):
      a. Producer updates its local producer index (locally) and updates the producer index in the consumer’s memory via remote writes.
      b. Consumer does the same with the consumer index.
      c. Each reads its local copy of these indexes.

   This way all reads are local, and only writes go through the interconnect. (Since writes work same as in PCIe, this means they arrive in order.)

   Note that this idea is advantageous even with normal PCIe because writes are posted, and reads are local, thus it allows achieving better performance.
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.

```
nice 19 RMP 1000 & # execute RMP 1000 with least favorable priority
RMP 1 # execute RMP 1 with normal priority
```

If RMP uses CPU, we expect the shorter job with more favorable priority to finish first. (It will have a higher pid because it was invoked last, so we’ll see the prints reporting “END RUN” of a higher pid then a lower pid).

If RMP uses GPU, priority will have no effect on the scheduling (GPU runs to completion and does not support preemption). We’ll see the lower pid (first job) finishing first.

In order to be certain of our conclusion and avoid corner cases (e.g. RMP uses GPU but we caught it just before it invoked the kernel) we perform this experiment a few times.
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.

Unfortunately, some students misunderstood the question as if we are comparing program A to program B. This is not the case. When we talk about speedup/slowdown we compare a program to itself (in different conditions / optimizations / implementations, etc..). We do not call the performance ratio of completely different programs a speedup/slowdown.

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

   Willow reads (Direct-IO) bypass the operating system, and thus do not incur system call (mode switch) overheads (these overheads include: CPU going to the interrupt descriptor table, switching to kernel stack, saving registers, switching segment selectors..., once when moving from user to kernel and again when moving back to user).

   Other explanations which were accepted:
   1. In order to reach the required block, it might be necessary to go through multiple blocks (indirect blocks). In Willow this happens in the SSD, while without Willow we’ll have to bring each of these blocks to the OS.
   2. With Willow the data is copied from SSD directly to user space. Without Willow data is copied first to kernel memory, before the kernel copies it again to user memory.

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

   While bypassing the OS kernel avoids the system call overhead, we also lose the benefits of the page cache. Page cache caches file pages in DRAM, and allows faster access to them by removing the need to go to the storage device. Page cache was not relevant in (1) because we read different pages (no reuse) and did so randomly (no prefetching). In this sub-question we read the same page, and page cache will save us going to the storage device every time (except first access).