CpSc 418 # Final Exam – Solution April 20 2016

1. The zero-one principle (**20 points** + **5 points Extra Credit**)
   Let $X[0..N-1]$ be a *bitonic* array of N elements, where N is even, and every element of X is either 0 or 1.

   (a) (**5 points**) Explain what it means that an array is bitonic. Use at most three sentences.

Solution

Any of the solutions below (or similar) are acceptable:

- A bitonic sequence is either a monotonically increasing sequence followed by a decreasing sequence or the other way around.
- (Because I said X is all 0s and 1s,) A bitonic sequence is of the form $0^*1^*0^*$ or $1^*0^*1^*$.
- If a solution just says "increasing followed by decreasing" or " a sequence of 0s followed by a sequence of 1s followed by a sequence of 0s", that will get full credit, even though the more general definition is needed for the proof in question 1c.

   (b) (**10 points**) Let Y be an array with N elements such that:

$$\begin{array}{rcll} Y[i] & = & \min(X[i], X[i+(N/2)]), & \text{if } 0 \leq i < N/2; \\ & = & \max(X[i-(N/2)], X[i]), & \text{if } N/2 \leq i < N. \end{array}$$

Show that either

- $Y[i] = 0$ for $0 \leq i < N/2$, or
- $Y[i] = 1$ for $N/2 \leq i < N$.

Both may hold for particular choices of X, but you just need to show that at least one of these conditions holds.

Solution

- If X is all 0s, then Y is all 0s, and $Y[i] = 0$ for $0 \leq i < N/2$ which satisfies the claim.
- Otherwise, let i_{lo} and i_{hi} be defined as suggested in the hint for the problem. Note, that this assumes that X is increasing-then-decreasing bitonic. The argument if X is decreasing-then-increasing is equivalent. Consider the three cases below:
  - case $N/2 \leq i_{\text{lo}}$. In this case, $X[i] = 0$ for $0 \leq i < N/2 \leq i_{\text{lo}}$; $Y[i] = 0$ for $0 \leq i < N/2 \leq i_{\text{lo}}$; and the claim is satisfied.
  - case $i_{\text{hi}} < N/2$, this is analogous to the previous case. In particular, $X[i] = 1$ for $i_{\text{hi}} < N/2 \leq i < N \leq i_{\text{lo}}$; $Y[i] = 1$ for $i_{\text{hi}} < N/2 \leq i < N \leq i_{\text{lo}}$; and the claim is satisfied.
  - case $i_{\text{lo}} < N/2 \leq i_{\text{hi}}$. In this case, $Y[0..(N/2)-1]$ is a sequence of 0s followed by a sequence of 1s, and $Y[(N/2)..(N-1)]$ is a sequence of 1s followed by a sequence of 0s. This produces two cases depending on how their "breaks" line up:
    * If $i_{\text{hi}} \leq i_{\text{lo}} + (N/2)$, then for all $0 \leq i < (N/2)$ either $X[i] = 0$ (i.e. for $0 \leq i < i_{\text{lo}}$) or $X[i+(N/2)] = 0$ (i.e. for $i_{\text{hi}} < i+(N/2) < N$). Therefore $Y[i] = 0$ for $0 \leq i < N/2$, and the claim is satisfied.
    * Otherwise $i_{\text{lo}} + (N/2) < N$, and a similar argument shows that $Y[i] = 1$ for $N/2 \leq i < N$. In a bit more detail, for all $N/2 \leq i < N$, either either $X[i] = 1$ (i.e. for $N/2 \leq i \leq i_{\text{hi}}$) or $X[i-(N/2)] = 1$ (i.e. for $i_{\text{lo}} \leq i-(N/2) < N/2$). Therefore $Y[i] = 1$ for $N/2 \leq i < N$, and the claim is satisfied.

   (c) (**5 points**) Let Y be defined as in question 1b. Show that $Y[0..(N/2)-1]$ is bitonic and that $Y[(N/2)..(N-1)]$ is bitonic.

Solution

- If $(N/2) \leq i_{\text{lo}}$ then $X[0..(N/2)-1]$ is all 0s; $Y[0..(N/2)-1]$ is all 0s; and the claim holds.
- If $i_{\text{hi}} < (N/2)$ then $X[(N/2)..(N-1)]$ is all 0s; $Y[0..(N/2)-1]$ is all 0s; and the claim holds.
- The final case is $0 \leq i_{\text{lo}} < (N/2) \leq i_{\text{hi}} < N$. If $i_{\text{lo}} + (N/2) \leq i_{\text{hi}}$, then the 1s in the lower half of X overlap the 1s in the upper half of X, and $Y[0..(N/2)-1]$ is $0^*1^*0^*$ bitonic. Otherwise, $i_{\text{lo}} + (N/2) > i_{\text{hi}}$; the 0s in the lower half of X overlap the 0s in the upper half of X, $Y[0..(N/2)-1]$ is all 0s (and the top half of Y is $1^*0^*1^*$ bitonic).

(d) (**5 points, Extra Credit**) Let A be a $N \times M$ array, where N is even, and $A[i,\, j]$ denotes the element of A in row i and column j. Let $A[i,\, 0..M-1]$ denote a row of A. Assume that every element of A is either 0 or 1, and that for $0 \leq i < N$, row $A[i,\, 0..M-1]$ is sorted into ascending order of i is even and into descending order if i is odd.

Let B be the $N \times M$ array where for $0 \leq j < M$, column $B[0..N-1,\, j]$ is $\mathsf{sort}(A[0..N-1,\, j])$, where sort sorts the elements of the column into ascending order.

Prove that at least $N/2$ rows of B are either all 0s are all 1s. For example, if $N = 16$, and B has 6 rows that are all 0s and three rows that are all 1s, then the claim is satisfied.

Solution

Note that we could sort the columns of A by first doing a compare and swap of $A[2i, j]$ and $A[2i+1, j]$ for $0 \leq i < (N/2)$, and then sorting the result. Doing this for $0 \leq j < M$ is the same as doing a compare-and-swap of rows $2i$ and $2i+1$. From questions 1b and 1c this produces (at least) one row that is all 0s or all 1s. The sort will take that row to the bottom of B if the row is all 0s or to the top of B if it is all 1s. Thus, each pair of rows of A produces at least one row that is all 0s or all 1s in B. Because A has N rows, B must have at least $N/2$ rows that are all 0s or all 1s.

2. CUDA (**20 points**) Let's say that I have an array of

$$n \;=\; 2^{27} \;=\; 2^{17} \cdot 2^{10} \;=\; 131{,}072 \cdot 1{,}024 \;=\; 134{,}217{,}728$$

`float`s, and I want to compute their sum using my blazing-fast GPU.

(a) (**10 points**) I've written three approaches to compute the sum below. Each is either incorrect or inefficient. Identify the problem with each, giving a short (at most three sentences) explanation for each one describing why it is wrong or why it is slow.

i. Create one block with 134,217,728 threads executing the kernel:

```
 0 __global__ void sum1(float *x, int n) {
 1    // assume n is a power of 2
 2    int myId = blockDim.x * blockIdx.x + threadIdx.x;
 3    if(myId < n) {
 4       for(int stride = 2; stride < n; stride *= 2) {
 5          __syncthreads();
 6          if((myId % stride) == 0)
 7             x[myId] = x[myId] + x[myId + stride]
 8       }
 9    }
10    // the result is in x[0]
11 }
```

The kernel launch is:

```
sum1<<<1, 134217728>>(x, 134217728)
```

ii. The same kernel as for version [i], but this time the kernel launch is:

```
sum1<<<131072, 1024>>(x, 134217728)
```

iii. Create one block with 1024 threads executing the kernel where each thread computes the sum of `n/1024` elements before the threads perform a reduce:

```
__global__ void sum3(float *x, int n, int m) {
    // assume n and m are powers of 2
    int myId = blockDim.x * blockIdx.x + threadIdx.x;
    if(myId < n) {
        // compute the sum for my part of the array
        float sum = 0.0;
        for(int i = 0; i < m; i++)
            sum += x[myId + i];

        // reduce
        for(int stride = 2; stride < blockDim.x; stride *= 2) {
            __syncthreads();
            if((myId % stride) == 0)
                x[myId] = x[myId] + x[myId + stride]
        }
    }
    // the result is in x[0]
}
```

The kernel launch is:

```
sum3<<<1, 1024>>(x, 134217728, 131072)
```

Solution

- Version (i) is incorrect because a block can have at most 1024 threads.
- Version (ii) is incorrect `__syncthreads()` only acts as a barrier for threads in the same block. It does not provide synchronization across blocks.
- Version (iii) is incorrect because
  - the threads compute their initial sums over overlapping parts of the array and miss most of the array.
  - the initial sum is not copied back to memory; so, it's not available for the reduce.

  Note, these two errors were unintentional – oops. But, the problem lets you point out any error or efficiency issue; so, this just made the problem a little bit easier. No bug-bounty for these.

  Version (iii) is also inefficient for many reasons (any one is fine):
  - Thread divergence. In the reduce loop, half of the threads in a warp are active in the first iteration, one fourth in the second, and so on.
  - Global memory accesses in the reduce loop. The global accesses in the initial sums are unavoidable, but we could use local memory here.
  - Lots of synchronization: we could use "warp synchronous" execution once all of the active threads are part of the same warp.

(b) (**10 points**) There are many ways to improve the performance of the slow kernel. Pick one such as coalescing global memory references, or using shared memory, or avoiding bank conflicts, or reducing thread divergence, or . . . .

- (**2 points**) State what optimization you are performing.
- (**4 points**) Write a short (no more than three sentence) description of what the optimization does.

- (**2 points**) Write the revised kernel. You can just write the lines that you change compared with the kernels I wrote above, and write
  lines 05-08 of sum2
  or similar to refer to lines from the kernels above.

Solution

I'll fix each of the problems listed above, but a real solution only needs to address one.

Fix sum3:

Change line 7 to:

```
sum += x[m*myId + i];
```

Just before the reduce (i.e. between lines 7 and 8) add

```
x[myId] = sum;
```

But now it will be slow due to uncoalesced memory references.

Global Memory Coalesscing:

Change line 6 to:

```
int stride = gridDim.x * blockDim.x;
for(int i = myId; i < n; i += stride) {
```

Global memory accesses in the reduce:

Declare a shared array, for example between lines 1 and 2 add

```
__shared__ float y[1024];
```

Then, change the line I added between lines 7 and 8 to:

```
y[myId] = sum;
```

and change line 12 to

```
y[myId] = y[myId] + y[myId+stride];
```

For an even faster solution, each thread would continue to use `sum` until the last iteration before it becomes idle. It would copy `sum` to `y[myId]` in that last iteration.

Reducing thread divergence:

As described in lecture and the text, start stride at `blockDim.x/2` and divide it by two until it is 0. This changes line 9-12 to:

```
 9       for(int stride = blockDim.x/2; stride > 0; stride /= 2) {
10          __syncthreads();
11          if((myId < stride) == 0)
12             y[myId] = y[myId] + y[myId + stride]
```

Using warp-synchronous execution:

Change the revised line 9 to:

```
for(int stride = blockDim.x/2; stride >= warpsize; stride /= 2) {
```

Then add another copy of the loop where `stride` goes from `warpsize/2` down to 0. Omit the `__syncthreads()` in this second loop.

(c) (**10 points**) Give two reasons that a GPU needs thousands of active threads to fully utilize the processors. For each, you can give its name (e.g. the nVidia™ jargon word or phrase) and a short description (at most four or five sentences for each of your two reasons).

Solution

Here are more than two, but a real solution should just have two.

- **Mitigating data hazards:** Using lots of threads allows the SPs to have deep pipelines (easier to implement, lower power, higher clock speeds) without the complications of bypasses.

- **Mitigating control hazards:** Even though a branch instruction takes 10s of cycles (or more) on a GPU, the SM can fetch from other warps until the branch is resolved.
- **Hiding global memory latency:** Accesses to global memory are slow. Other warps can execute while a warp is waiting for a load to finish.
- **Warps have lots of threads:** This amortizes the hardware, and especially the power consumption for instruction fetch, decode, and other pipeline control issues across many SPs.
- **A GPU has lots of SMs:** This is how it gets lots of parallelism. But many SMs times many warps per SM times many threads per warp results in needing thousands of threads to get high performance from GPU.

3. GPUs (**20 points**)
Earlier this month, nVidia™ announced the GP100 GPU. It has 3584 single-precision floating point cores with a peak 10.6 teraflops of single-precision floating point computation (1 teraflop = 1000 gigaflops = 10^{12} floating point operations per second). The interface to off-chip, global memory has a bandwidth of 720 GBytes/sec.

(a) (**10 points**) How many floating point operations must the GP100 perform on each `float` loaded from global memory if it is to achieve its peak floating point operation rate (and not by limited by memory bandwidth)?

Solution

A `float` is 4 bytes. Thus, the GP100 can load

$$\begin{aligned} & 720\,\frac{\text{Gbytes}}{\text{sec}} \cdot \frac{\text{1float}}{\text{4bytes}} \\ = \; & 180\,\frac{\text{Gfloats}}{\text{sec}} \end{aligned}$$

As stated in the problem, The GP100 can perform 10.6 teraflops. The number of floating point operations we need to do per global memory access to allow for peak floating-point rate is:

$$\begin{aligned} & 10.6 \cdot 10^{12}\,\frac{\text{float}-\text{ops}}{\text{sec}} \cdot \left(180 \cdot 10^{9}\,\frac{\text{floats}-\text{from}-\text{gmem}}{\text{sec}}\right)^{-1} \\ = \; & 58.\overline{8}\,\frac{\text{float}-\text{ops}}{\text{float}-\text{from}-\text{gmem}} \end{aligned}$$

This is more than twice the CGMA needed to fully utilize the SPs on the GTX 550 Ti on the `linXX` boxes that we used.

(b) (**10 points**) Give two reasons that a GPU needs thousands of active threads to fully utilize the processors. For each, you can give its name (e.g. the nVidia™ jargon word or phrase) and a short description (at most four or five sentences for each of your two reasons).

Solution

Here are more than two, but a real solution should just have two.

- **Mitigating data hazards:** Using lots of threads allows the SPs to have deep pipelines (easier to implement, lower power, higher clock speeds) without the complications of bypasses.
- **Mitigating control hazards:** Even though a branch instruction takes 10s of cycles (or more) on a GPU, the SM can fetch from other warps until the branch is resolved.
- **Hiding global memory latency:** Accesses to global memory are slow. Other warps can execute while a warp is waiting for a load to finish.
- **Warps have lots of threads:** This amortizes the hardware, and especially the power consumption for instruction fetch, decode, and other pipeline control issues across many SPs.
- **A GPU has lots of SMs:** This is how it gets lots of parallelism. But many SMs times many warps per SM times many threads per warp results in needing thousands of threads to get high performance from GPU.

4. **Speed-up** Let's say that I spend all day running four programs in sequence on my x86 linux machine:

   - Program-A runs for 1 minute on the x86.
   - Program-B runs for 1 minute on the x86.
   - Program-C runs for 1 minute on the x86.
   - Program-D runs for 1 minute on the x86.

   Each program depends on the results of the one that came before it; so, I can't run them in parallel ☹. After Program-D finishes, I go back and start the sequence with Program-A again.

   (a) (**5 points**) Assuming that I work 8 hours per day and get paid $1 each time I complete all four tasks, how much do I earn per day?

   ## Solution

   Completing all four tasks takes 4 minutes. That means 15 tasks in an hour, and 120 tasks in an eight-hour workday. I get paid $120/day.

   (b) (**5 points**) An nVidia™ sales person drops by my office and tells me that with a new, GP100 GPU, I can get the following speedups:

   - Program-A: speed-up = 100.0;
   - Program-B: speed-up = 10.0;
   - Program-C: speed-up = 1.0;
   - Program-D: speed-up = 0.1.

   If I sell my linux machine and move all four of my computing tasks to the GPU, long will it take me to complete all four tasks? What is the overall speed-up?

   ## Solution

   The total time for all four tasks is

   $$\begin{aligned} & \left(\tfrac{1}{100} + \tfrac{1}{10} + \tfrac{1}{1} + \tfrac{1}{0.1}\right) \text{ minutes} \\ = \; & 11.11 \text{ minutes} \end{aligned}$$

   Running on the x86, the time for all four is 4 minutes. The speedup is

   $$\begin{aligned} SpeedUp &= \tfrac{4\text{min}}{11.11\text{min}} \\ &= 0.36 \end{aligned}$$

   Yikes! I've been swindled, and now I'll only get $43/day. Ouch!

   (c) (**5 points**) Maybe selling my linux box wasn't such a good idea. Let's say I run Program-D on my linux machine, and I run the other three programs on the GPU. For simplicity, we'll ignore the time for communication when switching between running one task on the GPU and the next on the CPU (or the other way around). Because the tasks have sequential dependencies, I can't use the CPU and the GPU at the same time. With this combined CPU + GPU approach what is the speed-up compared with running everything on the CPU?

   ## Solution

   By calculation like the previous one, the total time is now for all four tasks is 2.11 minutes, and the speed-up is slightly less than 1.9.

   On a happier note, I'll now earn $211/day. Will the extra $91/day cover the cost of the GP100? It should.

(d) (**5 points**) What is Amdahl's law? Write the mathematical formula, and write two or three sentences to explain the intuition it gives for the speed-up results in the previous two sections.

Solution

The formula is:

$$SpeedUp \quad = \quad \frac{1}{f/s+(1-f)}$$

where $0 \leq f \leq 1$ is the fraction of an application that can be improved, and s is the speed-up for that fraction. In the limit that $s \rightarrow \infty$, the speed-up goes to $1/(1-f)$. This is what we are seeing in the example above. Half of the tasks (programs C and D) don't benefit from the speed up. If the other two had really large speed-ups (and 100 isn't bad), then we would still only get a speed-up of 2. The speed-up of 1.896 is getting close to the value of 2. Note that if the "improvement" results in a slow-down for some portion (Program-D in the part a), then that can result in a very large performance degradation.

Simple version: it's tempting to just try to average the speed-ups. For the problem above the average is 27.775. That sounds great. In reality, we need to account for the slowest part. If we can adjust our workload to have most of it get the large speed-up, then we can get a huge pay-off. In practice, this means:

- Parallel computing doesn't solve all of your problems.
- We adjust our uses of computers to line up with the problems that do get large speed-ups, such as graphics, animation, simulations, data analytics, and machine learning. Applications that don't benefit from parallel computing may still be important, but they will evolve much more slowly.

5. **Other stuff (20 points)**
Answer each question below with 1-3 sentences. Points may be taken off for long answers.

(a) (**4 points**) What is a pipeline bypass?

Solution

A mechanism that allows a result that is in the pipeline from an earlier issued instruction to be made available to a later issued instruction, even if the first instruction is still in the pipeline.

(b) (**4 points**) What is a fused multiply-add?

Solution

A common feature in hardware for floating-point arithmetic where a multiply followed by an add, e.g. `a*x + y`, can be computed as single operation. This can be very useful for common numerical algorithms (e.g. matrix multiplication), and it lets the marketing people claim higher GFlop or TFlop numbers by counting the fused multiply-add as two floating point operations.

(c) (**4 points**) What is the cross-section bandwidth of a message-passing multiprocessor?

Solution

Consider all ways to partition the N processors of the multiprocessor into two groups of size $N/2$. For each such partition, let B be the bandwidth of all the links between the two parts of the partition. The cross-section bandwidth is the smallest value of B for all possible partitions.

(d) (**4 points**) What is a "straggler" in a map-reduce computation?

Solution

A straggler is a task that takes much longer than the others.

(e) (**4 points**) How do “back-up tasks” mitigate the problems caused by stragglers?

Solution

We can use “work stealing”. Near the end of a map-reduce (or other parallel) computation, assign incomplete tasks to idle processors. Often, the stragglers are due to processor being slow (other jobs running on it, configuration problems, etc.), failed, or network problems. By replicating the task, the computation can finish when either the original task or the back-up task completes.