Full Name:

Andrew Id:

15-418/618 Spring 2020
Exercise 3
SOLUTION

Assigned:	Mon., Feb. 17
Due:	Fri., Feb. 21, 11:00 pm

Overview

This exercise is designed to help you better understand the lecture material and be prepared for the style of questions you will get on the exams. The questions are designed to have simple answers. Any explanation you provide can be brief—at most 3 sentences. You should work on this on your own, since that's how things will be when you take an exam.

You will submit an electronic version of this assignment to Gradescope as a PDF file. For those of you familiar with the LaTeX text formatter, you can download the template and configuation files at:

http://www.cs.cmu.edu/~418/exercises/config-ex3.tex
http://www.cs.cmu.edu/~418/exercises/ex3.tex

Instructions for how to use this template are included as comments in the file. Otherwise, you can use this PDF document as your starting point. You can either: 1) electronically modify the PDF, or 2) print it out, write your answers by hand, and scan it. In any case, we expect your solution to follow the formatting of this document.

Problem 1: GPU Programming

Your friends have been hearing that you now know how to program a GPU and take advantage of all its capabilities. Eager to bask in your knowledge, they come to you with different problems that they think might benefit from GPU parallelism.

One such friend heard that GPUs can launch thousands of parallel "threads." They want to identify the indices of a large array of n elements in an even larger array (of size m) using binary search. They suggest to you that every thread on the GPU will be searching for one element in the array, so that with say, 1024 threads, they'll be able to search get near $1024\times$ performance.

A. First off, how do you explain that threads on a CPU are not the same as threads on a GPU? (What's the difference between the implementation of p_threads and CUDA threads?) Give at least 3 important distinctions.

A CUDA thread is an abstract entity used to represent the execution of a kernel for one set of data values in an SPMD programming model. Threads are restricted to launching within blocks and are even collected into warps of 32 for SIMD execution. They are best purposed for massively parallel execution of independent operations. Unlike CPU threads/p_threads, they don't have independent control flow and cannot synchronize with each other.

B. Still convinced of their binary search's potential, they write this kernel function.

```
// Array S and R are of size n
// Array A is of size m, with its elements ordered.
__global__ void cudaBinarySearch(int n, int m, int *S, int *A, int *R) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i >= n) return;

    int search = S[i];
    int left = 0, right = m - 1, middle = 0;
    R[i] = -1;  // In case not found
    // Binary Search
    while (left <= right) {
      middle = left + (right - left) / 2;
      if (search < A[middle])
        right = middle - 1;
      else if (search > A[middle])
        left = middle + 1;
      else
        R[i] = middle;
        break;
    }
}
```

After testing it out, they find that they get very little performance gain relative to the number of threads that they are running. Sullenly, they come back to you and ask where they went wrong. Give at least two reasons for this code's poor performance.

1) Divergence because some threads will find their element sooner than others. 2) Many gather instructions called whenever many threads need to access `A[middle]` because each one's middle will not be the same. 3) Reading often from global memory.

Problem 2: Problem Scaling

This problem is a three-dimensional extension of the grid problem described in the lecture on workload-driven performance evaluation:

http://www.cs.cmu.edu/~418/lectures/10_perfeval.pdf.

Consider an iterative solver that operates on a three-dimensional grid of size $N \times N \times N$. It requires N iterations to reach convergence. We refer to the parameter N as the *problem size*. Increasing N by a factor of 2 increases the number of grid elements $8\times$ and the number of iterations $2\times$.

The state of the system is represented as a three-dimensional array of values `state`. The function `update` computes a new value of the state based on the current state. For each element `state[x][y][z]`, `update` computes the new value as a function of its current value and that of its six neighbors:

Self: `state[x][y][z]`

West: `state[x-1][y][z]`

East: `state[x+1][y][z]`

North: `state[x][y-1][z]`

South: `state[x][y+1][z]`

Down: `state[x][y][z-1]`

Up: `state[x][y][z+1]`

(You don't need to know the exact function.)

The overall operation can then be described in a pseudo-C notation as:

```
// For each iteration
for (iter = 0; iter < N; iter++) {
   // Main computation
   for all x, y, z in 0..N
       nstate[x][y][z] = update(state, x, y, z);
```

(Figure omitted)

Figure 1: Scaling a problem by doubling the grid size N (a), or by increasing the number of processors $8\times$ (b).

```
    for all x, y, z in 0..N
        state[x][y][z] = nstate[x][y][z]
}
```

In mapping the computation onto P processors, the state array is split into P cubic blocks, each containing N^3/P array elements. Each iteration (labeled "Main computation") involves having each processor 1) communicate the boundary values with its (up to six) neighbors, and 2) computing the update function for all N^3/P of its assigned elements. The program uses M gigabytes of memory per processor and runs in total time T.

1. Consider the following two scaling possibilities, yielding a problem of size N' running on P' processors, as illustrated in Figure 1.

   (a) Double the value of N, while holding P constant: $N' = 2N$, $P' = P$

   (b) Hold N constant, while increasing P by $8\times$: $N' = N$, $P' = 8P$

   Fill in the table below showing how the amounts of computation and communication for a single processor, performing a single iteration would scale, relative to the original problem. (For example, $2\times$ would indicate growth by a factor of two.)

	Computation	Communication
(a): $N' = 2N$, $P' = P$	$8\times$	$4\times$
(b): $N' = N$, $P' = 8P$	$1/8\times$	$1/4\times$

2. Now assume both N and P can vary. Based on these specific cases, give formulas for how the computation, communication, and arithmetic intensity would scale as functions of N and P. Again, quantify these values with respect to the computation and communication each processor would perform during a single iteration. (By way of reference, the formulas for the two-dimensional version were given in slide 7 of the lecture.)

Computation	Communication	Arithmetic Intensity
N^3/P	$N^2/P^{2/3}$	$N/P^{1/3}$

3. Suppose we have a machine with $8P$ processors, each identical to the original P processors. We consider three scaling possibilities, yielding a new problem of size N' requiring total time T' and M' gigabytes per processor:

Problem scaling: $N' = N$. Solve the same problem faster. Goal is to minimize T'

Memory scaling: $M' = M$. Make full use of the increased total memory. Goal is to maximize N'.

Time scaling: $T' = T$. Solve a bigger problem in the same amount of time. Goal is to maximize N'.

Fill in the following table with formulas indicating the problem size N', the per-processor memory requirement M', the ideal total time T', and the change in arithmetic intensity. (By way of reference, this information for the two-dimensional version was given in slides 17, 22, and 25 of the lecture.)

Scaling Type	N'	M'	T'	Arith. Intensity
Problem	N	$M/8$	$T/8$	$1/2\times$
Memory	$2N$	M	$2T$	$1\times$
Time	$2^{3/4}N \approx 1.68N$	$2^{-3/4}M \approx 0.59M$	T	$2^{-1/4}\times \approx 0.84\times$

Problem 3: Cache coherency

Your friend suggests modifying the MSI coherence protocol so that PrRd / BusRd behavior on the I-to-S transition is changed to PrRd / BusRdX, as is shown below:

(Figure omitted)

A. Is the memory system still coherent? What impact does this change have on the system?

The system is still coherent, although it will be greatly reduced in efficiency because all cache miss reads will result in invalidations of all other holders.

B. You are hired by Intel to design an invalidation based cache-coherent processor with a large number of cores. The main application for this processor will be to train a chess AI by playing games against itself for experiments. It is critical that it doesn't play too many games during training to maintain the validity of the experiment, so we store how many games are played in a shared counter:

```
int num_games = 100000;
int games_played = 0;

// This code is run on each core
while (true) {
  // Atomically get value of count prior to increment, and
  // write incremented value to games_played
  int val = atomic_add(&games_played, 1);

  if (val > num_games) break;
  play_game()
}
```

Unsure on how to design the cache coherence of this processor, you check with one of the senior designers. She suggests that you should use a bus-based, snooping coherence implementation, rather than a directory-based protocol, because the broadcasting it performs would be more efficient in this case. Do you agree or disagree? Why or why not?

Disagree, this program has frequent writes so only one core will have the shared counter in cache at a time. Thus a directory would be better as it would only have to invalidate one other cache line, rather than broadcast to all.

C. You are now tasked with finding the best performing chess AI. You plan to run the following code on the processor:

```
int num_experiments = 10000;
int best_score = 0;

#pragma omp parallel for
for (int i = 0; i < num_experiments; i++) {
  int score = perform_experiment(i);
  if (score > best_score) {
    // Atomically retrieve best score, compute max,
    // and write result to best_score
    atomic_max(&best_score, score);
  }
}
```

Trying to improve the performance of this code, you remove the `if` statement prior to the `atomic_max`, noticing that it is redundant. Suprisingly, the performance of the program is drastically reduced. Why did this happen?

Remember that `fetch_and_max` always causes a write. By removing the if statement, every iteration performs a write even if no write was needed. This causes each iteration to take a cache miss on `best_score`.

D. You are hired as a TA for 15-418/618. In an attempt to speed up exam grading, you decide to multithread the grading program on a cache coherent system as shown below

```
int scores[NUM_STUDENTS];

#pragma omp parallel for schedule(dynamic)
for (int student = 0; student < NUM_STUDENTS; student++) {
    for (int problem = 0; problem < NUM_EXAM_PROBLEMS; problem++) {
        // performs a write to scores[student]
        grade_and_update(student, problem, &scores[student]);
    }
}
```

Dismayed by the speed of this program, you analyze the runtime of this program and find that there is a large amount of bus communication during the execution.

What kind of communication is happening in this program? What performance problem is this a result of?

There is a large amount of artifactual communication happening in this program. As multiple student's scores occupy the same cache line, there is a lot of false sharing requiring large amounts of communication due to the coherence protocol.