Linköpings universitet
IDA Department of Computer and Information Sciences
Prof. Dr. Christoph Kessler

TENTAMEN / *EXAM*

TDDD56

Multicore and GPU Programming

8 jan 2018, 14:00–18:00, U11/U14

Jour: Christoph Kessler (070-3666687, 013-282406), visiting ca. 16:00.
Ingemar Ragnemalm (070-6262628), visiting ca. 16:00.

Hjälpmedel / *Admitted material:*

– Engelsk ordbok / *Dictionary from English to your native language*

General instructions

- This exam has 9 assignments and 5 pages, including this one.
  Read all assignments carefully and completely before you begin.
- It is recommended that you use a new sheet of paper for each assignment, because they will be corrected by different persons.
  Sort the pages by assignment, number them consecutively and mark each one on top with your exam ID and the course code.
- You may answer in either English or Swedish. **English is preferred** because not all correcting assistants understand Swedish.
- Write clearly. Unreadable text will be ignored.
- Be precise in your statements. Unprecise formulations may lead to a reduction of points.
- Motivate clearly all statements and reasoning.
- Explain calculations and solution procedures.
- The assignments are *not* ordered according to difficulty.
- The exam is designed for 40 points. You may thus plan about 5 minutes per point.
- Grading: U, 3, 4, 5. The preliminary threshold for passing is 20 points.
- We expect to have the exam corrected by ca. *22 january*. An exam review session around end of january will be announced on the course homepage.

1. (6 p.) **Multicore Architecture Concepts**

   (a) Define and explain the following technical terms:

      i. Moore's Law
      ii. (Cache) capacity miss
      iii. False sharing
      iv. Heterogeneous multicore system

      (Remember that an example is not a definition. Be general and thorough.) (4p)

   (b) Did the transition from single-core to multi-*core* CPUs, in itself, help to overcome the so-called *memory wall* in computer systems? Explain why or why not. (1p)

   (c) Why does the MSI coherence protocol, as described in the lecture, guarantee sequential (memory) consistency? (1p)

2. (5 p.) **Design and Analysis of Parallel Algorithms**

   (a) How could knowing the *working set size* of a (parallel) algorithm possibly help to explain the reason of a *speedup anomaly* observed with a multicore program? (1p)

   (b) Define the following two properties:

      - Critical path length
      - Parallel work

      of a parallel algorithm, and explain (calculation) how they are related to its parallel execution time with p processors through Brent's Theorem. (1.5p)

      (Hint: Make sure to properly introduce all symbols used and explain their meaning.)

   (c) Recall that the *prefix sums* vector $y = (y_1, ..., y_n)$ of an input vector $(x_1, ..., x_n)$ is defined by

$$y_i = \sum_{j=1}^{i} x_j$$

      for $i = 1, 2, ..., n$ (here, for inclusive prefix sums).

      We learned about several *parallel* algorithms for calculating the *prefix sums* of an array of n elements using (up to) n processors.

      Name and describe (using pseudocode or a well explained annotated picture) one such parallel algorithm of your choice that only performs $O(\log n)$ time steps using n processors (for an arbitrary PRAM model), and derive its time and work complexity (calculation). (2.5p)

3. (4 p.) **Parallel Programming with Threads and Tasks**

   (a) What does *thread pinning* mean?

      Describe 2 different scenarios where thread pinning is useful, and shortly explain why. (2p)

   (b) How does a work-stealing task scheduler work? And for what purpose should it be used? (2p)

4. (6 p.) **Non-blocking Synchronization**

   (a) In the lecture, we considered a *fair lock* implementation using the atomic `FetchAndIncr` operation:

```
// two shared counters, statically initialized to 0:
shared int ticket = 0;  // next waiting ticket to grab
shared int active = 0;  // ticket now entitled to enter CS

void acquire()   // acquire fair lock:
{
  int myticket = FetchAndIncr( &ticket, 1 );
  while (myticket != active)
     ;  // busy waiting
}

void release()   // release fair lock:
{
  active ++;
}
```

   (This implementation assumes a sequentially consistent memory.)

   (i) You are now given a multicore processor that has no atomic *fetch_and_increment* instruction but that has a *compare_and_swap* (CAS) instruction instead. Rewrite the above fair lock implementation using CAS such that the behavior is the same. Explain your solution. (2.5p)

   (ii) What is the *ABA problem* (in general) that can occur with CAS operations? (1p)

   (iii) Can the ABA problem occur in your CAS-based implementation of the fair lock? Explain why or why not. If yes, how likely is it to occur? Motivate your answer. (1p)

   (b) Do you know another kind of hardware atomic operation that can be used as an (at least equally powerful) alternative to CAS and that does not suffer from the ABA problem? (technical term only, no details) (0.5p)

   (c) Obviously, heap memory allocation for multithreaded programs must be thread-safe. Why should lock-free concurrent dynamic data structures (allocated on the heap) preferably be used with a lock-free heap memory allocator instead of traditional lock-based `malloc`/`free` implementations? (1p)

[In case of questions about the following 3 assignments, ask I. Ragnemalm in the first hand.]

Note from Ingemar Ragnemalm: *In all GPU questions, you may use CUDA or OpenCL style code as you please, but CUDA style is recommended. Exact syntax is not important.*

5. (5 p.) **GPU algorithms and coding**

   (a) Write code or detailed pseudo code for calculating the maximum value of a dataset using reduction on a GPU. Both GPU code and the relevant CPU code should be included. The approach should be reasonably optimized. Point out the most important optimization considerations and mark where in the code this occurs. (3p)

   (b) A Mandelbrot algorithm is given as sequential code as follows:

```
for (int x = 0; x < SIZE; x++)
    for (int y = 0; y < SIZE; y++)
        data[x, y] = computeFractal(x, y);
```

   that is, the fractal computing code is already available (you do not have to write it). How can you port this to an efficient GPU implementation? Outline vital parts of the code, both CPU and GPU sides. (2p)

6. (5 p.) **GPU conceptual questions**

   (a) Shared memory is fast temporary storage, but its access times still depends on something. What do you need to do to get the fastest possible shared memory access? (2p)

   (b) Explain why is it not possible to synchronize between blocks/work groups. What can you do about it? Give a demonstration of the problem and its solution based on bitonic merge sort. (3p)

7. (5 p.) **GPU Quickies**

   (a) What do you have to do for achieving better performance by coalescing? (1p)

   (b) Constant memory is fast under a certain condition. Which condition is that? (1p)

   (c) Texture access provides two unique features that we otherwise do not have. Name one, and describe with a brief sentence. (1p)

   (d) In GPU Computing using the graphics pipeline, in what stage are the computations usually carried out? (1p)

   (e) Why is load balancing often not a (big) problem in GPU computing, e.g. when computing fractals? (1p)

8. (1 p.) **Optimization and Parallelization**

   (a) Give a sufficient condition (dependence based argument) for that a loop's iterations can be executed in parallel. (1p)

9. (3 p.) **Parallel algorithmic design patterns and High-level parallel programming**

   (a) What is/are the main advantage(s) of programming using skeletons compared to, e.g., multithreaded programming? (1p)

   (b) In Lab 3, you implemented one version of *dot product* with separate *Map* and *Reduce* skeleton calls and another one using the *MapReduce* skeleton of SkePU.
   Explain why the version using *MapReduce* is, in general, more efficient
   (a) if executing both versions on CPU, and
   (b) if executing both versions on GPU? (2p)

Good luck!