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

TENTAMEN / *EXAM*

TDDD56
Multicore and GPU Programming

26 aug 2017, 14:00–18:00, TER1

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

Hjä lpmedel / *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.
- There is no exam review session for re-exams. After grading, the exams will be archived and available for inspection in the IDA student expedition (E-house, upper floor).

1. (4 p.) **Multicore Architecture Concepts**

   (a) Define and explain the following technical terms:

      i. SIMD parallelism

      ii. Hardware multithreading

      iii. False sharing

      (Remember that an example is not a definition. Be general and thorough.) (3p)

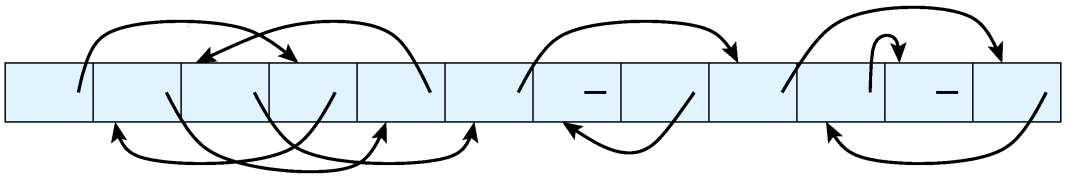
   (b) Why does the MSI coherence protocol, as described in the lecture, guarantee sequential (memory) consistency? (1p)

2. (7 p.) **Design and Analysis of Parallel Algorithms**

   (a) Why does it make sense to start the design of a new parallel program for a modern multicore processor or GPU with looking for suitable algorithms based on the PRAM model? (1p)

   (b) 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)

   (c) (5p) In the lecture, we introduced *list-ranking* as a fundamental computational problem on a shared array containing N list items that belong to one or several linked lists. The simplest scenario considered was calculating, for each list item, a pointer to the end of its list.



   (i) For this problem scenario, describe (pseudocode) and explain the *recursive pointer doubling* based algorithm using N parallel threads, as introduced in the lecture. Make sure to point out where (and what kind of) synchronization between threads takes place to guarantee correct execution of the algorithm. (2p)

   (ii) Derive its asymptotic parallel time, work and cost complexity (in terms of N). (1.5p)

   (iii) Adapt the algorithm to use a given fixed number $P \ll N$ of threads. Explain and motivate the correctness of your solution. How does the parallel time now depend on P? (1.5p)

   (iv) **Bonus question (+1p):** If P is chosen as $P(N) = o(N)$ (i.e., as a function that is growing properly slower than N, such as $N/\log N$), will this reduce the asymptotic parallel *cost* of the algorithm? Explain your answer (calculation).

   (Hint: This requires that you have solved the previous questions (i-iii) correctly.)

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

   (a) What is the purpose of back-off strategies in mutex-lock implementations? (1p)

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

4. (7 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) The busy-waiting `while` loop above contains a potential performance issue on standard (cache-based, sequentially consistent) multicore CPUs. Explain why, and describe one possible workaround. (2p)

   (ii) 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)

   (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. (2p)

   (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)

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

5. (6p.) **GPU Algorithms and Coding**

   (a) Outline how matrix multiplication of two large matrices is performed on a GPU. A clarifying figure and pseudo code are expected. (3p)

   (b) The following algorithm (given as OpenCL code) performs rank sort on the GPU, a simple but not very efficient sorting algorithm for data with unique keys.

```
__kernel void sort( __global unsigned int *data,
                    __global unsigned int *outdata,
                    const unsigned int length )
{
  unsigned int pos = 0;
  unsigned int i;
  unsigned int val;

  //find out how many values are smaller
  for (i = 0; i < get_global_size(0); i++)
    if (data[get_global_id(0)] > data[i])
      pos++;

  val = data[get_global_id(0)];
  outdata[pos]=val;
}
```

   This code can be significantly accelerated. Describe *two different* ways to accelerate the code. This *must* be made based on the same algorithm! Replacing the code with a totally different algorithm is *not* accepted! (3p)

6. (4p.) **GPU Conceptual Questions**

   (a) Outline how *reduction* is implemented in an efficient way, using text and figures. You may assume that the reduction problem in question deals with finding the maximum of a large dataset. Assume that the dataset can be of highly varying size, including very large. (2p)

   (b) Three important kinds of GPU memory include *shared* (local), *global* and *texture* memory. Describe these in terms of performance, usage and accessibility. CUDA terminology is assumed, please note if you use OpenCL terminology. (2p)

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

   (a) Some operations, like image filters, can be implemented using scatter or gather algorithms. If you use scatter, what specific operation must be used to make it work correctly? (1p)

   (b) Suggest a feature that texture units have that other memory access paths do not have. (1p)

(c) If you want to process a large array in fragment shader based computing, how will that data typically be represented? (1p)

(d) In CUDA, you can use the modifier `__device__`. What does this signify? (1p)

(e) Can you rely on any threads/work groups in a GPU computation to be literally executed in parallel? If so, which ones? (1p)

8. (2 p.) **Optimization and Parallelization**

(a) What is a *loop-carried data dependence*? (0.5p)

(b) Name and shortly describe a loop transformation that can improve the cache hit rate of a loop, and explain why. (1.5p)

9. (2 p.) **Parallel algorithmic design patterns and High-level parallel programming**

(a) The *dot product* of two n-element vectors x, y is defined as $\sum_{i=0}^{n-1} x_i y_i$.

Write a high-level parallel program for this computation using suitable *algorithmic skeletons* (pseudocode; explain the skeletons that you use). (1p)

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

Good luck!