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

TENTAMEN / *EXAM*

TDDD56
Multicore and GPU Programming

5 apr 2018, 14:00–18:00, TER1

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. (7 p.) **Multicore Architecture Concepts**

   (a) How do SIMD (vector) instructions work (in hardware)? What kind of parallelism do they exploit, for what kind of computations can they boost processor performance, and what are the conditions for their proper usage on the programmer's (or compiler's) side? (2p)

   (b) Define and explain the following technical terms:

      i. ILP (instruction-level parallelism) wall (for single-threaded CPUs)
      ii. MIMD parallelism
      iii. Dennard Scaling
      iv. Hardware multithreading

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

   (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) 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.)

   (b) The *suffix sums* vector $y = (y_1, ..., y_n)$ of an input vector $(x_1, ..., x_n)$ of n elements is defined by

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

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

      Describe (using pseudocode or a well explained annotated picture) a parallel algorithm of your choice for the suffix sums problem that only performs $O(\log n)$ time steps using n processors (for an arbitrary PRAM model), and derive its *time*, *work* and *cost* complexity (calculation). Explain also which parallel algorithmic design pattern(s) you used in your algorithm. (3.5p)

3. (3 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) What is the purpose of back-off strategies in mutex-lock implementations? (1p)

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

   (a) An *ordered linked list* (here, for integers) uses list items of the following type:

   ```
   struct elem {
     int value;
     struct elem *next;
   }
   ```

   Pointer variable `head` points to the first list element. Insertion of a new element with value v is done by searching for v and inserting a new list element `e` for v after the element where the search ended. As a simplification we assume that the list always contains at least one element, namely an artificial dummy element with value $-\infty$ as the first element in the list. We also assume for simplicity that *no remove operations* occur.

   ```
   struct elem *e = (struct elem *)malloc(sizeof(struct elem));
   struct elem *p = head;
   // assume for simplicity that the list contains at least 1 element
   // (the first element is a dummy element with value -infinity)
   while (p->next!=NULL && p->next->value < v)
     p = p->next;
   e->value = v;
   // insert e into the list after p:
   e->next = p->next;
   p->next = e;
   ```

   Assume that the list elements and the `head` pointer are stored in shared memory.

   i. Show that, without proper synchronization, two concurrent insertions may lead to an incorrect result. (Give a simple scenario, start with a one-element list).
   In general, what is the condition (on concurrently inserted values and list contents) for such a conflict to occur?
   Suggest a simple mutex-lock-based solution to make insertion thread-safe. (1.5p)

   ii. Use appropriate CAS operations (i.e., no mutex locks) to provide a non-blocking *insert* operation (pseudocode). Explain your code.
   Explain how possible conflicts between concurrent *insert* operations are recognized and handled properly by your implementation.
   In particular, argue why even in the case of conflicts at least one of the conflicting operations will succeed. (3p)
   (Hint: If you use a different CAS function than the one used in labs, you should carefully explain the function that you use as compare and swap: what are its parameters, what does it return and to what pseudo-code it is equivalent, using an `atomic{}` keyword/construct or another equivalent construct that needs to be defined.)

   (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.]

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

   (a) 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. How can you port this to an efficient GPU implementation? Outline vital parts of the code (CPU and GPU). (2p)

   (b) Describe, in pseudo code and figures, how an optimized matrix multiplication can be performed on the GPU. Your answer should focus on structure and vital features rather than detailed code. (3p)

6. **GPU Conceptual Questions** (5p)

   (a) Describe the major architectural differences between a multi-core CPU and a GPU (apart from the GPU being tightly coupled with image output). Focus on the differences that are important for parallel computing. (3p)

   (b) Some image filters are separable. This has potential to improve performance. Why? (1p)

   (c) A typical case of separable filter is to split a filter into one horizontal and one vertical filter, often of the same size. However, the two parts may each run with significantly different performance, one much faster than the other. Suggest a likely reason why this could happen. (1p)

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

   (a) In CUDA, you can use the modifiers `__global__` and `__device__`. What is the difference between them? (1p)

   (b) What particular algorithm feature makes bitonic merge sort particularly suitable for parallel implementation? (1p)

   (c) In graphics, data is always input as geometrical shapes. What geometry is usually used for fragment shader based GPU computing?

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

   (e) What do you have to do for achieving better performance by coalescing? (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!