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

TENTAMEN / *EXAM*

TDDD56

Multicore and GPU Programming

1 sep 2018, 14:00–18:00, TER1

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

Hjälpmedel / *Admitted material:*

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

General instructions

- This exam has 9 assignments and 4 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 may 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. (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. Hardware multithreading
      ii. Write-back cache
      iii. Weak memory consistency (in a shared memory system)
      iv. False sharing

      (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. (6 p.) **Design and Analysis of Parallel Algorithms**

   (a) Define the following two properties:

      - Critical path length
      - Parallel work

      of a parallel algorithm, and explain (commented formula) how they are related to its parallel execution time with p processors through Brent's Theorem. (2p)

      (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 drawing) 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. (4p)

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

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

   (b) Show (pseudocode) how the recursive parallel sum algorithm of the theory lecture can be implemented using `futures`. Explain your code and the constructs used. (2p)

4. (5 p.) **Non-blocking Synchronization**

   (a) 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)

   (b) Explain the operations Load-Linked (LL) and Store-Conditional (SC), and explain how the behavior of LL+SC differs from that of CAS if used for atomic updating of a shared memory location in the context of lock-free shared data structures. If you had the choice between LL+SC and CAS, which one would you prefer, and why? (2p)

   (c) What is the difference between a *lock-free* and a *wait-free* concurrent data structure? Which of them is the stronger property, and where could that difference matter? (2p)

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

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

   (a) Describe, with code or pseudo code, how reduction can be used to calculate the maximum value of a large array of scalar values on a GPU. Be clear about how any intermediate data storage is arranged. (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. However, the algorithm can be significantly accelerated.

```
__kernel void sort( __global unsigned int *data,
                    __global unsigned int *out,
                    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)];
  out[pos]=val;
}
```

   Describe a way to accelerate the code using the same algorithm. (2p)

6. **GPU Architecture concepts** (5p)

   (a) Compare shared memory, global memory, constant memory and register memory in terms of performance, usage and accessibility. (CUDA terminology is assumed, please note if you use OpenCL terminology.) (3p)

(b) You are given the task of implementing an algorithm that you decide needs to be implemented in a number of blocks, but there are dependencies between the blocks. How can you handle dependencies between different blocks? (CUDA terminology is assumed unless stated otherwise.) (2p)

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

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

   (b) Where does GPU computing fit in Flynn's taxonomy? What name(s) does the architecture type have according to Flynn's taxonomy? (1p)

   (c) Can you rely on any threads/work items in a GPU to be executed simultaneously? Which ones? (1p)

   (d) When performing in-place neighborhood operations like filtering on a GPU, you can use either gather or scatter operations. Which one would you recommend, and why? (1p)

   (e) For what kind of problems are fragment shader-based GPU computing most suitable? Give one specific example. (1p)

8. (3 p.) **Optimization and Parallelization**

   (a) Consider the following loop nest:

```
for i = 1, ..., M
   for j = 1, ..., N-1
      A[i][j] = x*A[i-1][j-1] + y*A[i-1][j] + z*A[i-1][j+1];
```

   (i) Would *tiling* (if applicable) of this loop nest be beneficial for performance if `N` is large? Justify your answer. (1p)

   (ii) Is *tiling* of this loop nest (e.g. with 2×2 tiles) applicable here, or would it change the semantics of the code? Justify your answer. (1p)

   (b) Why is it, in general, so hard for C/C++ compilers to statically analyze a given sequential legacy program and parallelize it automatically? (1p)

9. (1 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)

Good luck!