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

TENTAMEN / *EXAM*

TDDD56
Multicore and GPU Programming

9 apr 2015, 14:00–18:00, TER2

Jour: Nicolas Melot (073-7033860), visiting ca. 16:00.
Christoph Kessler (070-3666687), on travel, by phone only.
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.
- An exam review session will be announced on the course homepage.

1. (5 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. Dennard Scaling
      ii. Sequential (memory) consistency
      iii. Heterogeneous multicore system

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

2. (4 p.) **Parallel Programming with Threads and Tasks**

   (a) What does *thread pinning* mean, and what is its purpose? (2p)

   (b) How does programming with `futures` work? (2p)

3. (7.5 p.) **Non-blocking Synchronization**

(a) Name 2 problems of lock-based synchronization that are removed by non-blocking synchronization. (1p)

(b) 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)

(c) 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)

4. (5.5 p.) **Design and Analysis of Parallel Algorithms**

   Parallel sorting algorithms

   In the lecture on parallel sorting we took a closer look at, among others, the three algorithms *fully parallel quicksort*, *fully parallel mergesort* and *bitonic sort*. Choose your favorite one among the three algorithms (state which one), describe it with its main subroutines by well-explained pseudocode and a flowchart diagram, and analyze (i.e., derive by calculation, results given in big-O notation) its *parallel time*, *parallel work*, *maximum number of processors used at any time* and *parallel cost* for a problem size of n elements on an ideal parallel computer (CRCW PRAM) with an unlimited number of processors. If you need to make further assumptions, state them carefully. (5.5p)

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

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

   Describe in code/pseudocode and figures how an optimized matrix multiplication for large matrices can be efficiently implemented on the GPU.

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

   (a) Motivate why GPUs can give significantly better computing performance than ordinary CPUs. Is there any reason to believe that this advantage will be reduced over time? (2p)

   (b) 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)

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

   (a) GPUs have evolved around the needs of graphics applications. Give an example of a feature, apart from multiple threads, that was added for the needs of graphics which is valuable for GPU computing. (1p)

   (b) In what way(s) is a texturing unit more than just another memory? (1p)

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

   (d) Some operations can be implemented either as scatter or gather operations. Which is most suitable for parallel implementation (on GPUs in particular)? Why? (1p)

   (e) Compare OpenCL and Compute Shaders in terms of portability. You should know at least one strong point of each. (1p)

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

   (a) Consider the following sequential loop:

```
void foo( float *x, float *a, int N )
{
  int i;
  x[0] = a[0];
  for (i=0; i<N-1; i++)
     x[i] = a[i] + x[i+1];
}
```

   Assume that the arrays `x` and `a` do not overlap.

   Could the iterations of *this* loop (i.e., without further code restructuring), be run in parallel? Justify your answer *(formal argument)*.

   If yes, suggest a suitable mapping of the iterations to $p > 1$ parallel threads, and show why the behavior is the same as with sequential execution.

   If not, suggest a semantics-preserving code transformation of this loop that enables parallelization, and show the resulting source code.

   (If you need to make additional assumptions, state them carefully.) (2p)

9. (1 p.) **Parallel algorithmic design patterns and High-level parallel programming**

   Give two main advantages and two main drawbacks of *skeleton-based parallel programming*. (1p)

Good luck!