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

TENTAMEN / *EXAM*

TDDD56

Multicore and GPU Programming

30 aug 2014, 14:00–18:00, TER2

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

Hjälpmedel / *Admitted material:*

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

General instructions

- This exam has 9 assignments and 6 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. Hardware multithreading
      iii. Bus snooping

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

   (c) Recent many-core processor architectures (such as Intel SCC, Kalray MPPA, Tilera TILE64) are increasingly organized in a way that differs from standard multicore (dual-core, quadcore...) processor architectures. What is the main difference, what is the technical reason for it, and what does it imply for the programmer? (2p)

2. (2 p.) **Parallel Programming with Tasks**

   How does a work-stealing scheduler work?

3. (5.5 p.) **Non-blocking Synchronization**

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

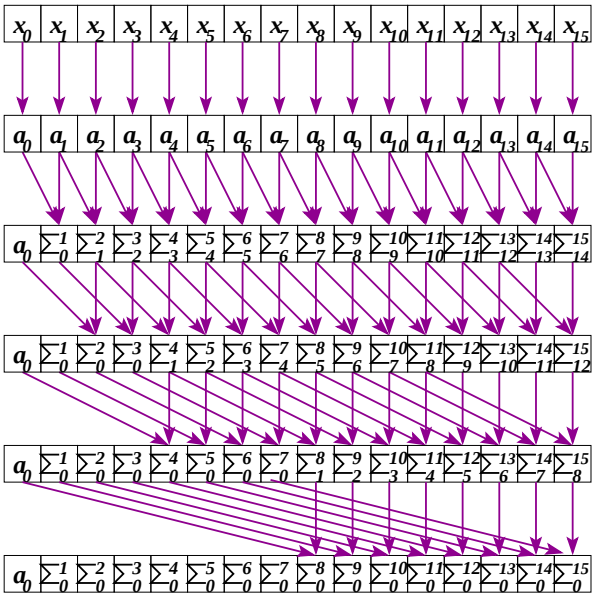
   (b) In the lecture, we considered a *fair lock* implementation using the atomic `FetchAndIncr` operation:

```
// 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.)

   You are 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)

   (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)



EREW parallel prefix sums, pseudocode:

```
float x : array[0..N − 1]; // input array
float a : array[0..N − 1]; // output array
int dist;
forall i : [0..N − 1] in parallel do
    // using N PRAM processors
    a[i] ← x[i]; // copy x to a
od
dist ← 1;
while dist < N do
    forall i : [0..N − 1] in parallel do
        // using N PRAM processors
        if i ≥dist then
            a[i] ← a[i−dist ] + a[i];
    od
    dist ← dist * 2;
od
```

Figure 1: EREW parallel prefix algorithm, with an illustration for $N = 16$ elements and processors.

4. (8 p.) **Design and Analysis of Parallel Algorithms**

   (a) Formulate Brent's Theorem (explained formula), give its interpretation, derive it (calculation), and describe its implications for parallel algorithm design and analysis. (3p)

   (b) The *parallel prefix sums* problem is defined as follows:
   Given a set S (e.g., the integers) and a sequence of N elements $x_0, \ldots, x_{N-1} \in S$, compute the sequence a of *prefix sums* defined by

$$a_i = \sum_{j=0}^{i} x_j \text{ for } 0 \leq i < N$$

   The massively parallel algorithm in Figure 1 has been proposed for the EREW (exclusive read, exclusive write) PRAM model; it uses one processor (hardware thread) per element in the input sequence x.
   (i) Explain why the algorithm is an EREW algorithm. (1p)
   (ii) Analyze the parallel execution time, parallel work and parallel cost (as formulas in N, using big-O notation) of this algorithm for a problem size N using N processors.
   (A solid derivation of the formulas is expected, just guessing the right answer gives no points.) (3p)
   (iii) Is this algorithm *work-optimal*? Justify your answer (formal argument). (1p)

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

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

   Write code (CUDA or OpenCL) or pseudocode for an algorithm that efficiently computes a 5×5 low-pass filter over a 2D array in parallel on a GPU. Use images to clarify how the computing is arranged geometrically. Motivate your arrangement from a performance point of view. Are there known optimizations beyond what you describe? (5p)

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

   (a) In many algorithms, one thread can produce values that affect other threads. Suggest two different ways to make sure that the results are produced without conflicts. The two approaches do not have to be relevant for the same situations. What is the performance impact of each approach? (Only dependencies within the same block are taken into account here.) (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? (2p)

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

   (a) Imagine a CUDA programmer who uses the practice to always use as big block size as possible. Why will this not always result in the highest possible performance? (1p)

   (b) What kind of algorithms benefit from using constant memory? (1p)

   (c) With shader-based GPGPU computing, suggest one limitation that prevents it from performing as well as CUDA and OpenCL. (This may apply to certain GPU generations, not necessarily the latest.) (1p)

   (d) Suggest one important feature in GPUs that was added for performing some specific graphics effect. Name the effect too. (1p)

   (e) List three different kinds of hardware that OpenCL runs on. (Similar systems by different vendors count as one.) (1p)

8. (2.5 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=1; i<N; i++)
     x[i] = a[i] + x[i-1];
}
```

   Assume that the arrays `x`, `a` and `b` do not overlap. Could the iterations of *this* loop be run in parallel? Justify your answer *(formal argument). (If you need to make additional assumptions, state them carefully.)* (1p)

   (b) What is the main motivation and the general principle (no details) of *dynamic* (i.e., runtime) parallelization of sequential loops? (1p)

   What is its main drawback compared to static (i.e., compile-time) parallelization? (0.5p)

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

   Give two main advantages and two main drawbacks of *skeleton-based parallel programming*.

Good luck!