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

TENTAMEN / *EXAM*

TDDD56
Multicore and GPU Programming

25 apr 2019, 14:00–18:00, TER2

Jour: Christoph Kessler (070-3666687, 013-282406), visiting ca. 16:00 if possible.
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 8 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 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. (6 p.) **Multicore Architecture Concepts**

   (a) Define and explain the following technical terms:

      i. SIMD parallelism
      ii. Dennard Scaling
      iii. (Cache) capacity miss
      iv. Hardware multithreading
      v. Last-level cache

      (Remember that an example is not a definition. Be general and thorough.) (5p)

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

2. (9 p.) **Design and Analysis of Parallel Algorithms**

   (a) 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)

   (b) 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.)

   (c) (6 p.) **Finding the roots of a parallel forest**
      A *parallel binary forest* is a large shared array F of N elements of the type

```
struct treeelem {
  struct treeelem *leftchild, *rightchild; // given as input
  struct treeelem *root; // to be computed
  struct treeelem *parent; // auxiliary pointer, not given
  // ... further entries
} F[N];
```

      As known from sequential computing, a forest contains (usually, several) trees. Each node has at most one parent node; the nodes without parent are called the *roots* of the forest. The forest nodes are stored in arbitrary order in the array `F`. The child pointers are given as input; for leaf nodes the child pointers are NULL. Each node belongs to one tree, which is uniquely determined by (the address of) its root node. The task is to calculate, for each node, a pointer `root` to the root of the tree that it belongs to (see Figure 1, shown for node v).

      i. Develop a parallel algorithm for finding the roots of the N forest nodes in parallel time $O(\log N)$ on a CREW (Concurrent Read, Exclusive Write) PRAM with N processors.

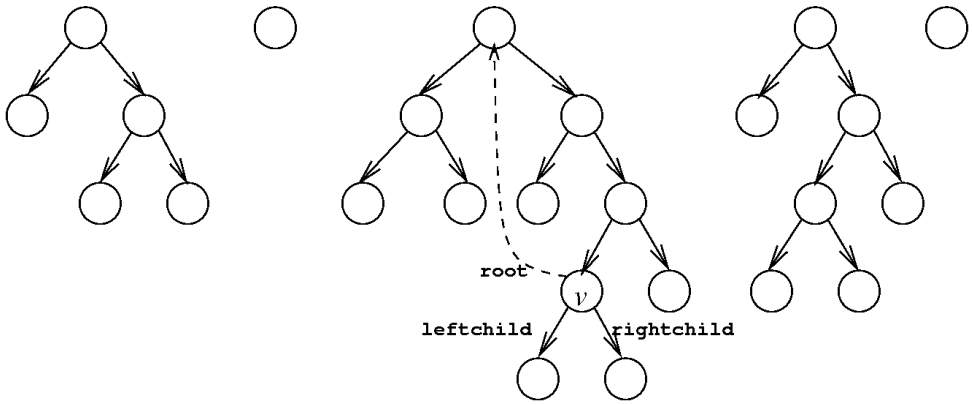

Figure 1: Forest example. The given pointers to the children are shown (solid arrows). The pointer `root` is to be computed for each node; it is shown here for node v only (dashed arrow).

(Hint: Work in two steps:
1. Use the auxiliary pointers `parent` and show how to calculate them quickly in parallel for all elements as a first step, so that for each non-root node its `parent` pointer now points to its direct parent node in the forest, and for each root node its `parent` pointer should be NULL.
2. Now develop an efficient method to calculate the `root` pointers with the help of the `parent` pointers. You might use a technique known from the lecture.)
Show and explain the resulting pseudocode. (3.5p)

ii. Analyze the algorithm for its asymptotic parallel time, parallel work and parallel cost (each as a formula in N, using $\Theta()$ notation). Explain. (1.5p)

iii. Explain why your algorithm matches the CREW constraint for the shared memory accesses, i.e., where concurrent read is used in the algorithm and why concurrent write access does not occur for any memory location. (1p)

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

   (a) What does *thread pinning* mean? (1p)

   (b) Spin-locks can, as we know, be implemented using the atomic *test-and-set* instruction in the *mutex_lock* operation. What is the purpose of instead using a *test-test-and-set* (TTAS) strategy for acquiring a spin-lock? Explain how it works. (1p)

   (c) How does a work-stealing task scheduler work? (1p)

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

   (a) The operation $y = \mathit{fetch\_and\_add}(p, a)$ atomically adds to the memory location pointed to by p the integer value a and returns the (integer) value y that the location had immediately before the update.

   You are given a multicore processor that has no atomic *fetch_and_add* instruction but that provides an atomic *compare_and_swap* (CAS) instruction instead. Write a software implementation of atomic *fetch_and_add* using CAS. Explain your solution. (2p)

(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)

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

Note from Ingemar Ragnemalm: In all GPU questions, you may use CUDA or OpenCL style code as you please, but CUDA style is recommended. Exact syntax is not important.

5. **GPU Algorithms and Coding**

(a) Describe, using figures and pseudo code in sufficient detail (full CUDA/OpenCL code is not needed), how matrix multiplication of large matrices can be implemented on the GPU. (GPU kernel code only.) Emphasize the most vital considerations for good performance. (3p)

(b) The following code implements bitonic merge sort sequentially on CPU.

```
void bitonic_cpu(unsigned int *data, int N)
{
  unsigned int i,j,k;

  printf("CPU sorting.\n");

  for (k=2;k<=N;k=2*k) // Outer loop, double size for each step
  {
    for (j=k>>1;j>0;j=j>>1) // Inner loop, half size for each step
    {
      for (i=0;i<N;i++) // Loop over data
      {
        int ixj=i^j; // Calculate indexing!
        if ((ixj)>i)
        {
          if ((i&k)==0 && data[i]>data[ixj])
             exchange(&data[i],&data[ixj]);
          if ((i&k)!=0 && data[i]<data[ixj])
             exchange(&data[i],&data[ixj]);
        }
      }
    }
  }
}
```

Describe how this can be ported to GPU code. (2p)

6. **GPU Architecture concepts**

   (a) List three different kinds of GPU memory and describe for each their characteristics in terms of performance, usage and accessibility. CUDA terminology is assumed, please note if you use OpenCL terminology. (3p)

   (b) Why can coalescing improve performance? How can you take advantage of coalescing for an algorithm with a non-coalesced memory access pattern? (2p)

7. **GPU Quickies**

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

   (b) Explain (briefly) why the G80 architecture had significantly higher performance than earlier GPUs. (1p)

   (c) In graphics, data is always input as geometrical shapes. What geometry is usually used for fragment shader based GPU computing? (1p)

   (d) Compare GPU computing using OpenGL Compute Shaders or Vulcan. (1p)

   (e) Why can an image filter be accelerated using separable filters? (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];
```

   Would it be correct to apply *loop interchange* to this loop nest? Justify your answer (dependence-based argument). (1p)

   (b) Give a sufficient condition (dependence based argument) for that a loop's iterations can be executed in parallel. (1p)

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

Good luck!