BYTTES MOT STANDARD SIDE

Beskjed til eksamensvakter:
ALLE SVAR MAA ANGIS AV KANDIDATENE PAA VEDLAGTE EKSAMENSARK SOM LEVERES INN SAMMEN MED OMSLAG OG NOTATARK. NOTATARK VIL GENERELT IKKE BLI RETTET.

Ingen hjelpemidler er tillatt.

Note to students:
ALL ANSWERS NEED TO BE WRITTEN ON THIS EXAM´s NUMBERED PAGES WHERE INDICATED AND THESE NUMBERED SHEETS NEED TO BE TURNED IN FOR GRADING.
YOU MAY ONLY USE THE EXTRA NUMBERED SHEETS ATTACHED FOR THE PROGRAMMING PROBLEMS.

Please fill in your candidate number on each sheet before turning them in.

YOU MAY NOT KEEP OR DISTRIBUTE ANY COPIES OF THIS EXAM.

Aids :
No aids, including calculators, are permitted.

Grades will be assigned by January 4, 2014.
It is NOT necessary to justify your answer on true/false questions, unless requested.

Written by: Anne C. Elster **Checked by: Thomas L. Falch and Ruben Spaans**

1. WARM-UPS – TRUE/ FALSE (10 %)

Circle your answers -- Note: You will get a -1% negative score for each wrong answer and 0 for not answering or circling both TRUE and FALSE.

a) **Programming in MPI forces you to think about data locality ….....TRUE/FALSE**

b) **Branching can resolve performance problems ….....TRUE/FALSE**

c) **Caches make use of spatial and temporal locality to increase performance TRUE/FALSE**

d) **A reduction is a race condition ..... TRUE/FALSE**

e) **Processes are like a thread except for communication …..... TRUE/FALSE**

f) **Threads of the same process share that process' memory …..... TRUE/FALSE**

g) **Data locality matter on NUMA shared memory systems …..... TRUE/FALSE**

h) **Caching is used to help overcome data locality issues …..... TRUE/FALSE**

i) **Caching may lead to superscalar performance .....TRUE/FALSE**

j) **Elster's Serial Algorithm for BitReversal is O(N log N)..... TRUE/FALSE**

k) **OpenMP is an API for multithreading …..... TRUE/FALSE**

l) **Recursion benefits from on-chip registers available for stack ….....….....TRUE/FALSE**

m) **Computations with image output is well suited for GPU computing ….....TRUE/FALSE**

n) **Constant memory in CUDA is read-only from host ….....TRUE/FALSE**

o) **Block and grid dimensions in CUDA can have up to 3 dimensions ….....TRUE/FALSE**

p) **CUDA threads may access any registers in a given warp ….....TRUE/FALSE**

q) **Team is the OpenCL-equivalent of a CUDA warp …..... TRUE/FALSE**

r) **CUDA is less verbose than OpenCL,.....TRUE/FALSE**

s) **Open Access publishing may be costly for the authors …..... TRUE/FALSE**

t) **Divergent branching affects CUDA performance …..... TRUE/FALSE**

2. PARALLEL COMPUTING BASICS (12%)

a) Which 3 components constitute the "BRICK WALL" described in Lecture 3 (Hint: the term was coined by Prof. David Patterson at UC Berkeley.)

i) Power wall

ii) ILP wall

iii) Memory wall

b) What are the Pros and Cons of Replicated vs. Distributed grids?

Replicated: Pros: Easier to program

Cons: Less scalable, more memory

Distributed: Pros: Scalable, memory efficient

Cons: Communication

c) Give an example of parallel programming where you would NOT use MPI:

Program with extensive communication between processors

d) Draw a 2D Torus and a 5D hypercube

x

3. More on MPI and Basic Concepts(12%)

a) When does the MPI standard send mode return? (Circle what applies)

i. **It depends on message size**
ii. **It depends on the platform**
iii. **It may complete before any reception has been posted**
iv. **All of the above**

b) For what and when is MPI_Buffer_attach used?

When Bsend is used

c) For what and when is MPI_ Request used?

When non-blocking is sent with request as status

d) Does MPI collectives use tags in MPI 1 and 2? Why or why not?

All processes must participate in collective operations and leaves it unnecessary to differentiate with tags

e) Which trick did Elster use to get the serial BitReversal from O(N*N) to O(N logN)?

The trick for going from an O(NlogN) algorithm to a linear algorithm is to view the computations of a bit-reversed index as a mapping from one sequence to another

f) Describe the trick used to get the above algorithm suitable for multi-core systems? (Hint: Similar method used in Atlas and FFTW)

Using a computation tree which may be approached in both depth-first or breadth-first manner, similar to FFTW

4. Optimizations and other issues (12%)

a) Which of the following types of branches would one generally optimize:

i) Conditional branches executed for the first time
ii) Conditional branches that have been executed more than once.
ii) Call and Return
iv) Indirect calls & jumps (function pointers & jump tables)
v) Unconditional direct branches/jumps

b) The SPMD model versus SIMD

i) Obtains parallelism through branching: SPMD

ii) CUDA warps use the SIMD **model.**

iii) MPI follows the SPMD **model.**

iv) Intel AVX instructions follow the SIMD **model**

c) What saves us from Amdahl´s law on modern 10 000 core+ supercomputers?

Gustafson's law, scaling the problem size accordingly

d) Explain the difference between speedup and efficiency, using T_serial and T_parallel on system with p processes:

Efficiency is the speedup divided by number of processors. Describes the speedup per thread, eliminating the speedup of adding more and more processors.

Speedup is the relative difference between T_serial and T_parallel. Giving the speedup achieved by parallelizing the code.

e) d) Fill in all that apply of: MPI, OpenMP, CUDA and/or OpenCL

i) Makes you think about memory locality: MPI,OpenMP,CUDA,OpenCL

ii) Most widely used on share memory multiprocessors: OpenMP

iii) Most widely used for scalable cluster computing: MPI

iv) Most widely used on AMD GPUs: OpenCL

v) Designed for use on heterogeneous clusters: OpenCL

vi) Uses ANSI C extension CUDA

5. MPI Programming (10%)

Consider the following code, where rank 0 distributes the contents of the array `buffer` to all the ranks.

```
if (rank == 0) {
    for (int i=0; i<size; i++)
        MPI_Send(&buffer[i*count], count, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
}

MPI_Recv(&local_buffer[0], count, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

a) Explain why the code may not work:

MPI_Send may or may not block, dependent on buffer-size

b) Explain why it may often work in practice:

Often the hardware will have enough memory to non-blocking

c) Rewrite the code so it always works using some of the functions provided below (not all functions may be required/are relevant). Do not use other MPI functions.

```
int MPI_Ssend( void *buf, int count, MPI_Datatype datatype,
                      int dest, int tag, MPI_Comm comm )

int MPI_Wait ( MPI_Request  *request, MPI_Status   *status)

int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag,
                      MPI_Comm comm, MPI_Request *request )

int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source,
                      int tag, MPI_Comm comm, MPI_Request *request )

int MPI_Issend( void *buf, int count, MPI_Datatype datatype, int dest,
                      int tag, MPI_Comm comm, MPI_Request *request )
```

Extra space for 5, if needed:

6. OpenMP Programming (5%)

Consider the following code.

```
for(int i = 0; i < 10000; i++){
     array[i] = myFunction(i);
}
```

Assume that `myFunction()` does not have any side effects, and does not access shared/global memory.

a) Show how this code can be parallelized using OpenMP.

b) Which schedule would be most appropriate to specify assuming that:

i. The execution time of `myFunction()` is always the same.
ii. The execution time of `myFunction()` is proportional to the value of the argument.
iii. The execution time of `myFunction()` varies randomly.

7. Vectorizing (5%)

Consider the following code, which multiplies the elements of two arrays of doubles, `a` and `b`.

```
for(int i = 0; i < N; i++){
    c[i] = a[i] * b[i];
}
```

The following code attempts to speed up the computation with SSE instructions.

```
__m128d x,y,z;

for(int i = 0; i < N; i++){
    x = _mm_loadu_pd(&a[i]);
    y = _mm_loadu_pd(&b[i]);

    z = _mm_mul_pd(x,y);

    _mm_storeu_pd(&c[i],z);
}
```

a) This code does not achieve any speedup over serial code. Explain why.

b) Modify the code above so that it achieves a speedup.

8. OpenCL (10%)

Consider the following OpenCL kernel, which performs image blurring:

```
__kernel void(__global float* inputImage, __global float* outputImage){

      int x = get_global_id(0);
      int y = get_global_id(1);

      int id = x * 512 + y;

      if(x > 0 && x <511 && y > 0 && y < 511){

            float r = 0;
            for(int i = -1; i < 2; i++){
                  for(int j = -1; j < 2; j++){
                        int temp_id = (x+i)*512 + (y+j);
                        r += inputImage[temp_id];
                  }
            }
            outputImage[id] = r/9;
      }
      else{
            outputImage[id] = 0;
      }
}
```

a) Modify the code so that it uses shared memory to speed up the computation. Indicate what work-group size you are using. You may assume that the image is always 512 by 512, and that the work-group size can be hardcoded.

Shared memory arrays, where the size is known at compile time is declared with the following syntax:

`__local float localBuffer[1024];` (for a buffer of 1024 floats).

In addition, the function `barrier()` synchronizes work-items in the same work-group.

Extra space for 8, if needed.

9. CUDA (24%)

Consider an image processing algorithm which applies one of two functions to each pixel based on whether the pixel value is above or below a threshold as described by the following pseudocode:

```
for each i in image:
        if(input[i] > threshold)
                output[i] = func1(input[i])
        else
                output[i] = func2(input[i]
```

a) Write a CUDA kernel which performs this algorithm, and show how it should be started/called. Make no assumptions about what the images will look like.

9 b) Consider the two images below. Which would achieve the best speedup with your algorithm? Assume that the threshold value lies between the light and dark colors in each image.

Image 1

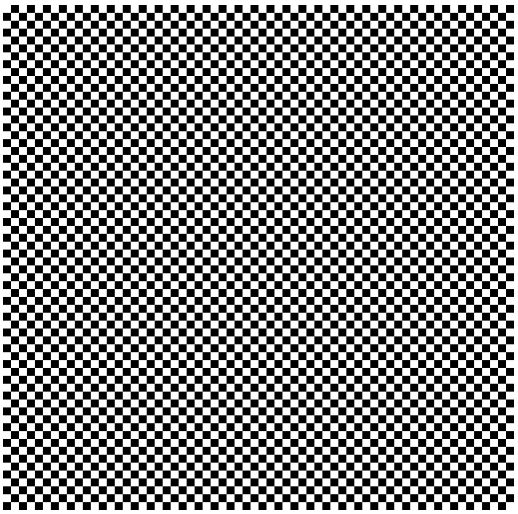

Image 2

9 c) Assume that it is known that the input images will always have a chessboard pattern (where alternating pixels have values above and below the iso value). Modify the kernel so that it is optimized for this kind of input.

EXTRA SHEET for grading