EE4-63
EE9AO14

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2012

MSc and EEE/ISE PART III/IV: MEng, BEng and ACGI

HIGH PERFORMANCE COMPUTING FOR ENGINEERS

Thursday, 3 May 10:00 am

Time allowed: 2:00 hours

There are FOUR questions on this paper.

Answer THREE questions.

All questions carry equal marks.

Please use a SEPARATE ANSWER BOOK for each question.

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible First Marker(s) : D.B. Thomas
Second Marker(s) : M.M. Draief

1. A high-performance signal processing system is being designed which relies on matrix-matrix multiplication. There are n input channels, each of which is sampled k times per second to produce a matrix $\mathbf{S}$ containing n columns and k rows, which is then multiplied by a constant $m \times k$ matrix $\mathbf{A}$ to produce the $m \times n$ element output $\mathbf{D}$:

$$\mathbf{D} = \mathbf{A} \times \mathbf{S}$$

The system will be implemented using a platform which provides a fused multiply-add, which multiplies any two memory locations, and adds it to another memory location. For example, given a memory pointer `p`, each operation can perform `p[dst] = p[dst] + p[a]*p[b]` for arbitrary indices `dst`, `a`, and `b`.

a) Describe the differences between a VLIW or SIMD processor, and explain which is the most appropriate architecture for this problem. [4]

b) i) Give pseudo-code for iterative matrix-matrix multiplication. [4]

ii) How many multiply-adds must the system perform per second? [2]

iii) Given hard requirements that $n = 50$ and $m = k/16$, suggest upper bounds of feasibility for k. Explain and justify any assumptions you make. [4]

c) The n columns of $\mathbf{D}$ will be sent to n output channels corresponding to the n input channels used to construct $\mathbf{S}$.

i) Using a diagram, show how the matrices and calculation could be split between two processors, each handling $n/2$ channels. [3]

ii) Assuming that the two processors implement memory consistency through co-operating caches, would you expect to see cache contention between the two processors? [3]

2. Cilk is used to schedule the following code on a system containing two CPUs:

```
cilk int Worker(int x)
{
  if(x>5){
    return F(x);
  }else{
    int a=Worker(x*2);
    int b=spawn Worker(x*2+1);
    sync;
    return a+b;
  }
}
```

The function `F(x)` contains no parallelism and always takes exactly one second to execute.

a) Identify the Cilk keywords in this sample and explain what they mean. [4]

b) The above program is designed to employ recursive parallelism, but contains a bug.

i) What is the bug, and what are the implications for parallelism in the program? [2]

ii) What is the run-time of the statement `Worker(1)` with the code as written. [2]

iii) Modify the program to fix the bug, but without increasing the number of spawn statements. [3]

iv) Show the execution schedule of the modified programme on a machine with two CPUs, showing the executions of `F(x)` including the value of the function argument, and the processor on which each one executes, starting from the statement `Worker(1)`. [5]

c) Someone suggests adding a new feature to Cilk called `idle`, such that the statement `idle Function()` would cause the given function to be spawned as a task whenever a CPU was unable to find a task ready to execute. For example:

```
cilk void IdleWorker()
{
... // Do small amount of work
}

void main()
{
  idle IdleWorker(); // Execute when CPUs are available
  MainWorker();  // Starts immediately, may spawn sub-tasks
}
```

would start `MainWorker`, but also allow `IdleWorker` to make use of any spare CPU cycles.

Explain how this breaks one of the key design principles of Cilk, and give a concrete example where this principle is violated. [4]

3. a) Describe the Map-Reduce design pattern. [3]

b) Explain whether Map-Reduce is valid in the following calculations, and whether it should be applied:

i) Multiplication of matrices: $\mathbf{R} = \mathbf{A}_1 \times \mathbf{A}_2 \times .. \times \mathbf{A}_{n-1} \times \mathbf{A}_n$ [2]

ii) Finding an element in a sorted array. [2]

iii) Calculating a continued fraction: $r = a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$ [2]

c) Why does `tbb::parallel_reduce` pass a range to function objects, rather than just an index? [3]

d) The same functionality as `tbb:parallel_reduce` be achieved using `tbb::parallel_for` and `tbb::atomic`.

i) Give pseudo-code for a TBB solution that uses this approach to find the minimum of an unsorted array of integers. [5]

ii) What would be the expected performance of this approach compared to `tbb:parallel_reduce` and why? [3]

4. A particular GPU architecture has 8 threads per warp, uses pipelined ALUs requiring 5 cycles per operation at a frequency of 800MHz, and has 10 processors each with 640 register words.

a) i) What is the peak performance of the GPU in ALU operations per second? [2]

ii) What is the minimum number of threads per block required to fully utilise the ALUs? [2]

iii) A given kernel requires 7 registers – what is the maximum block size that could be supported? [2]

iv) A different kernel requires 13 registers and 10^8 ALU operations per thread, contains no branches, and is launched using 32 threads per block, and 27 blocks per grid. Provide a lower bound on the execution time of the grid on this GPU. [4]

b) The following kernel has been developed:

```
__global__ void MyKernel(float *inout)
{
  int id=threadIdx.x;
  float value=inout[id];
  if((id%4)==0){
    value=F1(value);
  }else{
    value=F2(value);
  }
  inout[id]=value;
}
```

On average the function F1 requires 10000 ALU instructions, and F2 requires 15000 ALU instructions.

i) Identify a performance problem with this kernel. [2]

ii) Estimate the maximum number of executions per second for the above kernel on the above GPU. [2]

iii) Rewrite the kernel to eliminate the problem, making any reasonable assumptions necessary. [6]

Solutions 2012 Master Copy 11/4/12

ICE4-67
EE9 A-014

1. A high-performance signal processing system is being designed which relies on matrix-matrix multiplication. There are n input channels, each of which is sampled k times per second to produce a matrix $\mathbf{S}$ containing n columns and k rows, which is then multiplied by a constant $m \times k$ matrix $\mathbf{A}$ to produce the $m \times n$ element output $\mathbf{D}$:

$$\mathbf{D} = \mathbf{A} \times \mathbf{S}$$

The system will be implemented using a platform which provides a fused multiply-add, which multiplies any two memory locations, and adds it to another memory location. For example, given a memory pointer `p`, each operation can perform `p[dst] = p[dst] + p[a]*p[b]` for arbitrary indices `dst`, `a`, and `b`.

a) Describe the differences between a VLIW or SIMD processor, and explain which is the most appropriate architecture for this problem.
Answer : A Very Long Instruction Word (VLIW) processor allows each instruction to specific two or more different types of operations to be applied, with the ability to select inputs from amongst the register file. A Single-Instruction Multiple-Data processor only supports one operation per cycle, but this is applied to all registers within the SIMD lane.

In this instance a SIMD processor would be more appropriate, as there is clear vector parallelism within the matrix-matrix multiplication. [4]

b) i) Give pseudo-code for iterative matrix-matrix multiplication.
Answer :

```
for row=1:m
  for col=1:n
    D[row*n+col]=0;
    for z=1:k
      D[row*n+col] += A[row*n+z] * S[z*k+col]
    end
  end
end
```

[4]

ii) How many multiply-adds must the system perform per second?
*Answer : n*m*k - we don't need to know the batch size because the work scales linearly with k.* [2]

iii) Given hard requirements that $n = 50$ and $m = k/16$, suggest upper bounds of feasibility for k. Explain and justify any assumptions you make.
*Answer : As this is a signal processing system there are likely to be requirements on latency, so we can only use compute that can be placed local to the inputs and outputs, suggesting only a few compute devices can work on the problem. Assuming a GPU-like device has 32 ALUs * 30 processors * 1GHz we have 960 billion multiply-adds per second. The total work we have is* $50 \times (k/16) \times k = 3.125 \times k^2$*, so:*

$$960 \times 10^9 = 3.125 \times k^2 \quad (1.1)$$

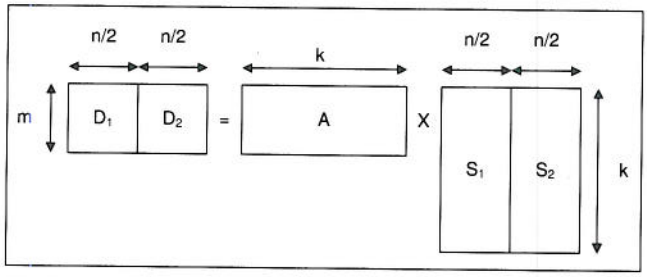
$$k = \sqrt{960 \times 10^9 / 3.125} \quad (1.2)$$

$$k = 554256.3 \quad (1.3)$$

So a feasible upper bound for k is probably around $k \approx 500000$. [4]

c) The n columns of **D** will be sent to n output channels corresponding to the n input channels used to construct **S**.

i) Using a diagram, show how the matrices and calculation could be split between two processors, each handling $n/2$ channels.
Answer : The input and output matrices can be split into two sub-matrices, grouping together original columns (i.e. streams of input and output samples).



Processor 1 can read from S_1 and safely write to D_1, as processor 2 will only write to D_2. Essentially each column is completely independent, so the problem is actually a large number of matrix-vector multiplies.
[3]

ii) Assuming that the two processors implement memory consistency through co-operating caches, would you expect to see cache contention between the two processors?
Answer : Cache contention will only occur if the two processors read and write to the same memory. The only read-write memory is matrix D, and if it is partitioned as above, then there should be no true contention over different words. However, if the matrix is in row-major order (i.e. store the matrix as rows) then at the boundary point a single cache line will contain addresses for both processors, which could cause significant contention for small n. A solution is to use column major order (store the matrix as columns), or to pad the rows of sub-matrices with zeros so that no cache line is shared between processors.
[3]

2. Cilk is used to schedule the following code on a system containing two CPUs:

```
cilk int Worker(int x)
{
  if(x>5){
    return F(x);
  }else{
    int a=Worker(x*2);
    int b=spawn Worker(x*2+1);
    sync;
    return a+b;
  }
}
```

The function `F(x)` contains no parallelism and always takes exactly one second to execute.

a) Identify the Cilk keywords in this sample and explain what they mean.
Answer : The three cilk keywords are "cilk", "spawn", and "sync". "cilk" is used to distinguish spawnable functions from standard C functions; "spawn" is used to create a sub-task which can be executed asynchronously; and "sync" is used to block the task until all spawned sub-tasks have completed. [4]

b) The above program is designed to employ recursive parallelism, but contains a bug.

i) What is the bug, and what are the implications for parallelism in the program?
Answer : The first recursive call to Worker executes synchronously within the current task, while the second call is spawned but then immediately synced. The effect is that though tasks are spawned there is no actual parallelism, and only one CPU will be used. [2]

ii) What is the run-time of the statement `Worker(1)` with the code as written.
Answer : The expansion is: Worker(1):
Worker(2)+Worker(3)
(Worker(4)+Worker(5)) + (Worker(6)+Worker(7))
([Worker(8)+Worker(9)]+[Worker(10)+Worker(11)]) + (F(6)+F(7))
([F(8)+F(9)]+[F(10)+F(11)]) + (F(6)+F(7))
So in total the function F is called 6 times, and the execution time will be 6 seconds. [2]

iii) Modify the program to fix the bug, but without increasing the number of spawn statements.
Answer : If the spawn is moved to the first recursive call, then it can proceed in parallel with the original thread. This allows full parallelism, without two spawn statements per function.

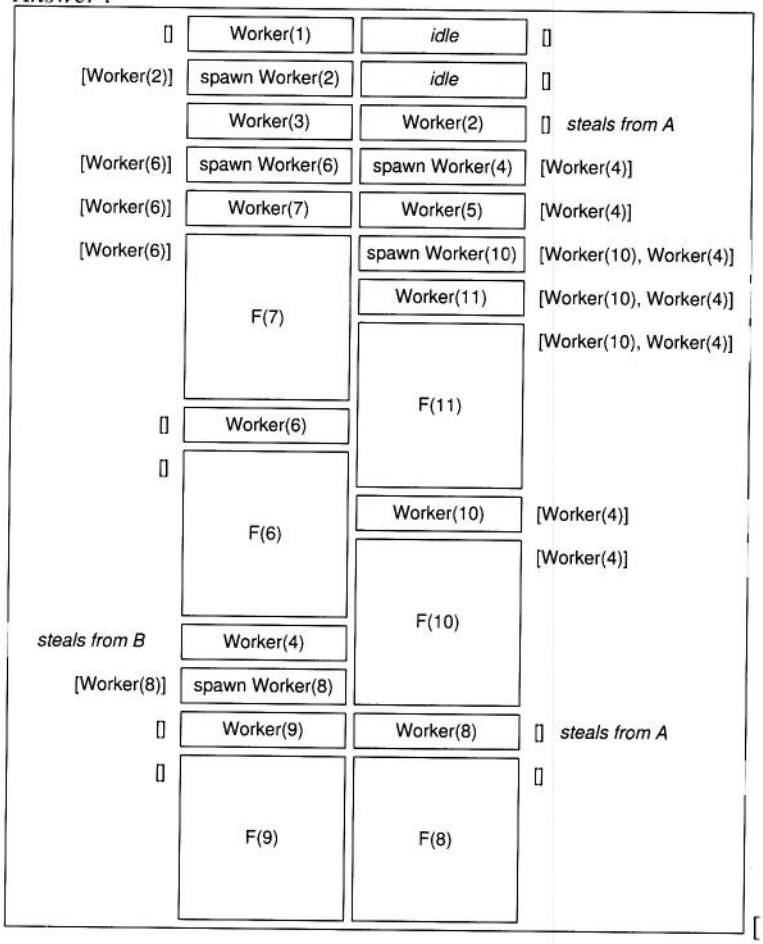
```
cilk int Worker(int x)
{
  if(x>5){
    return F(x);
  }else{
    int a=spawn Worker(x*2);
    int b=Worker(x*2+1);
    sync;
```

```
    return a+b;
  }
}
```

[3]

iv) Show the execution schedule of the modified programme on a machine with two CPUs, showing the executions of `F(x)` including the value of the function argument, and the processor on which each one executes, starting from the statement `Worker(1)`.

Answer :



[5]

c) Someone suggests adding a new feature to Cilk called `idle`, such that the statement `idle Function()` would cause the given function to be spawned as a task whenever a CPU was unable to find a task ready to execute. For example:

```
cilk void IdleWorker()
{
... // Do small amount of work
}

void main()
{
  idle IdleWorker(); // Execute when CPUs are available
  MainWorker();  // Starts immediately, may spawn sub-tasks
}
```

would start `MainWorker`, but also allow `IdleWorker` to make use of any spare CPU cycles.

Explain how this breaks one of the key design principles of Cilk, and give a concrete example where this principle is violated.

Answer : This breaks the serial elision principle - if the cilk keywords (including "idle") are removed, then the programme should still perform the same task. For example, the following code:

```
cilk void IdleWorker()
{
printf("Idle\n");
}

void main()
{
  idle IdleWorker();
  printf("Main");
}
```

would print something like "IdleIdleIdleMain" on a four-core machine, while if the keywords were removed it would print "IdleMain" independent of the number of CPUs. [4]

3. a) Describe the Map-Reduce design pattern.
Answer : The map reduce pattern applies when a function needs to be applied to many independent pieces of data (map), and then the many results are recursively combined to produce a single result (reduce). All executions of the map function should be independent, and the reduce function should be associative, i.e. r(a,r(b,c))=r(r(a,b),c). [3]

b) Explain whether Map-Reduce is valid in the following calculations, and whether it should be applied:

i) Multiplication of matrices: $\mathbf{R} = \mathbf{A}_1 \times \mathbf{A}_2 \times .. \times \mathbf{A}_{n-1} \times \mathbf{A}_n$
Answer : Matrix multiplications are associative, as $A_1 \times (A_2 \times A_3) = (A_1 \times A_2) \times A_3$ so this could use map reduce with an identity map function. It should be applied because the expensive matrix multiplications can be performed in parallel during the reduction. [2]

ii) Finding an element in a sorted array.
Answer : One could use map-reduce, with a map function that returns a pointer to the array entry if it contains the element, and a reduction that returns the first non-null pointer. However, this would require $O(n)$ work and a $O(\log n)$ critical path versus the $O(\log n)$ work and critical path of a sequential bisection search, so while map-reduce could be used, it should not be used here. [2]

iii) Calculating a continued fraction: $r = a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$
Answer : Given the reduction function $r(a,b) = a + 1/b$, we find that:

$$r(a,r(b,c)) = r(a,b+1/c) = a+1/(b+1/c) \quad (3.1)$$
$$r(r(a,b),c) = r(a+1/b,c) = a+1/b+1/c \quad (3.2)$$

So the reduction is not associative, and parallel reduce does not apply, and cannot be used. [2]

c) Why does `tbb::parallel_reduce` pass a range to function objects, rather than just an index?
Answer : The range represents the grain size which should be processed sequentially, and is a form of agglomeration - by switching to sequential processing at a certain granularity the cost of work is reduced. [3]

d) The same functionality as `tbb:parallel_reduce` be achieved using `tbb::parallel_for` and `tbb::atomic`.

i) Give pseudo-code for a TBB solution that uses this approach to find the minimum of an unsorted array of integers.
Answer : A sketch of a C++ implementation is:

```
class Worker
{
  atomic<int> *pMin;
  int *data;

  Worker(atomic<int> *_min, int *_data)
  {
```

```
    pMin=_min;
    data=_data;
  }

  template<class R>
  void operator(const R &r) const
  {
    int localMin=data[r.begin()];
    for(int i=r.begin()+1;i<r.end();i++){
     localMin=min(localMin,data[i]);
    }

    while(true){
      int curr=pMin->fetch_and_store(localMin);
      if(curr >= localMin)
       break;
      localMin=curr;
    }
  }
};

int find_min(int n, int *data)
{
  atomic<int> minv;
  minv=0;
  parallel_for(blocked_range(0,n), Worker(&minv,data));
  return minv;
}
```

[5]

ii) What would be the expected performance of this approach compared to `tbb:parallel_reduce` and why?

Answer : The atomic operations may be slightly slower than the non-atomic maths used in the parallel_reduce join. However, for a sufficiently large grain size the different should be minimal, as the ratio of sequential code to atomic operations would be very large. [3]

4. A particular GPU architecture has 8 threads per warp, uses pipelined ALUs requiring 5 cycles per operation at a frequency of 800MHz, and has 10 processors each with 640 register words.

a) i) What is the peak performance of the GPU in ALU operations per second?
*Answer : 8 ALUs/processor * 10 processors/GPU = 80 ALUs/GPU.*

*80 ALUs/GPU * 800 MHz = 64 billion ALU operations/second.* [2]

ii) What is the minimum number of threads per block required to fully utilise the ALUs?
*Answer : Five warps are needed to fill the ALU pipeline: 5*8=40 threads/block.* [2]

iii) A given kernel requires 7 registers – what is the maximum block size that could be supported?
Answer : There are 640/8=80 warp-size registers, so the maximum warps per processor is $\lfloor 80/7 \rfloor = 11$, *and the maximum threads per block is* $11 \times 8 = 88$. [2]

iv) A different kernel requires 13 registers and 10^8 ALU operations per thread, contains no branches, and is launched using 32 threads per block, and 27 blocks per grid. Provide a lower bound on the execution time of the grid on this GPU.
Answer : Each processor can support $\lfloor 640/8/13 \rfloor = 6$ *warps, but each block contains* $32/8 = 4$ *warps, so only one block can be active on a processor at once. Each of the four warps must execute* 10^8 *ALU instructions, but because there are only 4 warps one in five ALU cycles will be wasted, so the execution time per block is* $10^8 \times 5$ */ 800MHz* $=$ *0.625 seconds. There are a total of 27 blocks to execute on 10 processors, so the total time is at least* $\lceil 3 \rceil \times 0.625 = 1.875$ *seconds.*

This is a lower bound because any branching or scheduling can only increase the execution time. [4]

b) The following kernel has been developed:

```
__global__ void MyKernel(float *inout)
{
  int id=threadIdx.x;
  float value=inout[id];
  if((id%4)==0){
    value=F1(value);
  }else{
    value=F2(value);
  }
  inout[id]=value;
}
```

On average the function F1 requires 10000 ALU instructions, and F2 requires 15000 ALU instructions.

i) Identify a performance problem with this kernel.
Answer : The branch will cause thread divergence, as each warp will have to execute both F1 and F2. [2]

ii) Estimate the maximum number of executions per second for the above kernel on the above GPU.

Answer : Must assume that the block and grid size are chosen to hide ALU latency, that there are enough blocks to occupy processors, and that the memory reads and writes at the beginning and end have negligible cost compared to F1 and F2.

The total time for each thread is $10000+15000=25000$ *ALU instructions, so (64 billion ops/sec/GPU) / (25000 ops/kernel) = 2.56 million kernels/sec/GPU* [2]

iii) Rewrite the kernel to eliminate the problem, making any reasonable assumptions necessary.

Answer : We'll make the assumption that the block size is always a multiple of 32. This means that within 32 threads there are 8 executions of F1, which can be packed into one warp, and the following three warps will all execute F2.

```
__global__ void MyKernel(float *inout)
{
  int id=threadIdx.x;

  // Split into blocks of 32
  int local=id%32;
  int global=id-local;

  // Remap within each block
  local=(local%4)*8 + (local/4);

  // turn back into an id
  id=global+local;

  float value=inout[id];
  if((id%4)==0){
    value=F1(value);
  }else{
    value=F2(value);
  }
  inout[id]=value;
}
```

[6]