1.

a. Moore's law continues to allow increasing numbers of hardware features to be placed on a single chip, but computer designers need to be increasingly careful of power consumption. It has been argued that in the near future it will no longer be possible to power the whole of a processor chip at once. Instead different parts may be powered up at different times, depending on the programs being run at a particular time. This phenomenon, where much of the chip must be powered off at any time has been called *dark silicon*. Briefly describe some of the key problems that might arise with dark silicon, and how it might affect the number and types of cores and other features on future processors.

[10 marks]

Notes:

- CMOS technology scaling has enabled higher integration, faster switching and lower power consumption per transistor.
- However, recently, the power per transistor has not been scaling in proportion with transistor area, resulting in increasing power density. Nonetheless, current device packaging and cooling technology strongly constrain the peak power density beyond which chips become thermally unstable.
- These trends have resulted in the so-called "dark silicon" problem.
- The next generation of chips will have more transistors than can be simultaneously powered on. Parts of a chip might have to be kept powered off (or "dark") at any given instant. The problem, therefore, is how to best utilize the abundance of transistors in the dark silicon era to continue to reap the rewards of Moore's Law down to the sub-10 nm node (when transistors are <10nm in size).

Answer:

Dark silicon is a solution to the problem of heat dissipation & power consumption, but since we are changing how a CPU is designed, to handle a problem that will only increase with time, we also need to change the characteristics of that CPU - most notably how to most effectively manage dark silicon to continue to improve computation according to Moore's Law.

Chip and system designers will be force to adopt new asynchronous circuits that help them address the the power consumption issue. Dedicated co-processors, e.g. the Floating Point Unit (FPU), SIMD coprocessors, (ARMv8 even has instructions dedicated to AES encryption) have been introduced to reduce the pressure on the overall circuitry. If complex jobs can be delegated to specialized areas of computing it reduces the burden of power.

The problems that arise with dark silicon primarily revolve around management. There will be overhead introduced to manage these asynchronous circuit and different architectures will outperform others. There is an optimal dark silicon solution, though that has to be discovered first.

This is the trend that we've seen in response to the dark silicon. Future processors will have more cores/other features, that will be specialized to compute a certain limited range of tasks quickly, and using limited power.

i.e. The floating-point unit consumes very little power when you're not executing floating-point instructions; when you are, the FPU executes them a lot faster and more power-efficiently.

Overall the solution is to have dedicated asynchronous coprocessors. For example, the GPU and other collections of coprocessors for image, video and sound processing.

ARM's big.LITTLE designs are a simple case of heterogeneous multicore. They have a "LITTLE" processor for maximum power efficiency and while the "big" processors provide maximum compute performance. Both are coherent and share the same instruction set.

b. Modern architectures commonly provide one or more atomic machine instructions that can be used to implement locks. One of these is the atomic compare-and-swap instruction. The following pseudocode shows the high-level behaviour of the atomic compare-and-swap instruction:

```
int compare_and_swap(int * address, int testval, int newval)
    {
    int oldval;

    oldval = * address;
    if (oldval == testval) {
            *address = newval;
    }
    return old_val;
}
```

Explain how this instruction can be used to implement locks on shared-memory parallel computers.

[20 marks]

Compare and swap ensures atomicity by testing the variable we want to manipulate with an expected value for that variable.

In this example the variable that we want to manipulate is at the address pointed to by address, we load this value into oldVal and in one action compare it to an expected value - testval.

If the value is what we expect it to be, then we have access to the correct data, and can manipulate it accordingly. If the value is different than we expect then another thread must have manipulated it and we cannot securely operate on it. This is how the lock is implemented, if the data is not what we expect then the value is locked. We must then decide how to wait for it to be unlocked, whether the thread spins or dies.

The high level description shows the concept, but the key part is that this action happens atomically, that is to say as one indivisible block of code. This is provided at the machine instruction level.

However, in compare and swap, a value must be fetched, assessed and potentially changed atomically. What prevents another core from accessing the memory address after the first has fetched it but before it sets the new value?

The cache coherency protocol already manages access rights for cache lines. So if a core has (temporal) exclusive access rights to a cache line, no other core can access that cache line. To access that cache line the other core has to obtain access rights first, and the protocol to obtain those rights involves the current owner. In effect, the cache coherency protocol prevents other cores from accessing the cache line silently.

c. Examine each of the following pieces of code. State if each individual piece of code can be vectorized using SSE. If not, state clearly why. If the code can be vectorized, use SSE intrinsics to vectorize it. Write a short note explaining the parallelization strategy on any code you vectorize.

```
/* code segment 1 */
void add_scaled(float * a, float * b, float *c, float factor)
{
    for ( int i = 0; i < 1024; i++ ) {
            a[i] = b[i] + c[i] * factor;
    }
}
```

[10 marks]

```
/* code segment 2 */
float dot_product(float *restrict a, float *restrict b, int size)
{
    float sum = 0.0;
    for ( int i = 0; i < size; i++ ) {
            sum = sum + a[i] * b[i];
    }
    return sum;
}
```

[10 marks]

1. This code can be vectorized, the body of the for loop deals with a 1D array, or a vector, meaning it is easy to vectorize. My strategy was to load each of the value into an SSE vector variable and use the specialized command for multiplication and addition to perform the operation

```
#include xmmintrin.h
void add_vect(float * a, float * b, float * c, float factor)
{
  __m128 vfactor= _mm_set1_ps (factor); //load 4 copies of factor
  for(i = 0; i<1024; i+=4){
    __m128 va= _mm_load_ps_(&a[i]);
    __m128 vb= _mm_load_ps_(&b[i]);
    __m128 vc= _mm_load_ps_(&c[i]);
    vx = mm_mul_ps(vc, vfactor);
    vy = mm_add_ps(vb, vx)
    _mm_store_ps(&a[i],vy);
  }
}
```

2. This code is more difficult to vectorize because of the scalar sum value. However what we can do is create 4 partial sums and the sum them horizontally at the end.

```
void add_vect(float *restrict a, float *restrict b, int size)
{
float sum;

    __m128 vsum = _mm_set1_ps(0.0f);

    assert((n & 3) == 0);

    assert(((uintptr_t)a & 15) == 0);

    for (int i = 0; i < n; i += 4)

    {

       __m128 va= _mm_load_ps_(&a[i]);

       __m128 vb= _mm_load_ps_(&b[i]);

       __m128 vx = mm_mul_ps(va, vb);
       vsum = mm_add_ps(vsum, vx)
    }

    vsum = _mm_hadd_ps(vsum, vsum);

    vsum = _mm_hadd_ps(vsum, vsum);

    _mm_store_ss(&sum, vsum);

    return sum;

}
```

2. A common problem is to find cases where the same string appears more than once on huge lists of strings. A function is needed that takes an array of strings as a parameter, and writes to the screen all the strings that appear more than once in the array.

   Write a parallel function (or set of functions) that performs this task using C and either OpenMP or pthreads. Your function should be parallel, and should make efficient use of machine resources on a modern multi-core computer. The prototype of your function should look like:

   void print_duplicates(char ** strings, int num_strings)

   In the prototype *strings* is the array of C strings containing the strings among which we are searching for duplicates. The number of items in the strings array is stored in *num_strings*. The order in which the duplicates are written out is not important.

   [30 marks]

   Write a short commentary on your function explaining how it works, and why you believe it to be efficient. You should comment on the time and space complexity of your solution, and also explain why you think it is likely to run efficiently on a modern multi-core architecture.

   [20 marks]

```c
#include<stdio.h>
#include<stdlib.h>

void printRepeating(char ** strings, int num_strings)
{

  qsort(strings, num_strings, sizeof(char *), strcmp);
  #pragma omp parallel for
  for (int i = 0; i < num_strings; i++)
   { if (strcmp(strings[i], strings[i-1])==0)
        printf("%d\n", strings[i]);
   }
}

int main()
{
  printRepeating(strings, num_strings);
  return 0;
}
```

Serial time complexity is O(nlogn). This is more efficient than the brute force O(n^2) with a nested for loop that iterates over all elements. The speed would be further increase by the OMP multithreaded for loop. This speed up is obtained by sorting the array of string firsts, then comparing adjacent strings, if there are duplicates they will be adjacent due to the sorting.

The parallelism is implemented using an OpenMP parallel for loop.

3. Computers are always designed with typical applications in mind. The result is that computers to solve different problems have very different features. The following C co computes the standard deviation of the items in an array.

```
/* compute standard deviation of numbers in array */
double stdev(const double a[ ], const int size, double mean)
{
    double sum = 0.0;
    for ( int i = 0; i < size, i++ ) {
            double diff = a[i] - mean;
            sum = sum + diff*diff;
    }
    return sqrt(sum/((double) size));
}
```

Identify any potential parallelism in the above code. [10 ma

Discuss the suitability of various parallel computer architectures for executing this c assuming a large array. The parallel architectures you should discuss are:

I. out-of-order superscalar
II. very long instruction word (VLIW)
III. vector processor
IV. multithreaded/simultaneous multithreaded
V. shared memory multi-core processor
VI. multiple chip symmetric multiprocessor
VII. NUMA multiprocessor
VIII. distributed memory multicomputer.

You should explain which aspects of each architecture suit the code well and identify likely bottlenecks or problems in the running of the code. For each architecture should also describe any programmer intervention that may be required to parallelize code for the particular architecture.

The for loop could be parallelized using multiple threads, The assignment of sum however is a critical section that must be executed in isolation, so as to keep a consistent and accurate record of sum.

1. The only parallelism that I can identify takes place in the loop, as by the end the threads must converge to give a single return value. The loop however is parallelizable

I. OoO Superscalar

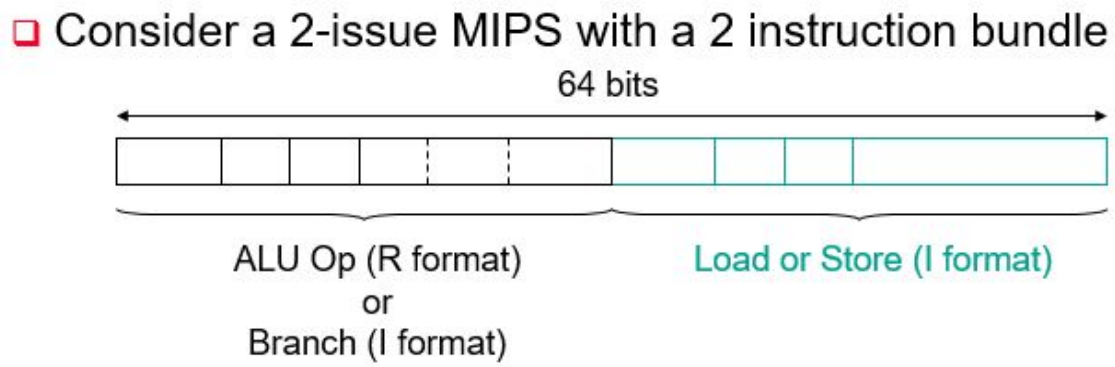
Also called Dynamic Instruction Scheduling, Out of Order Superscalars are processors that can begin execution of multiple instructions at once. A superscalar will typically have multiple processing units, (e.g. 2 ALUs) that can execute two instructions at once. In an OoO architecture, if an instruction takes a large amount of cycles to execute,, other instructions will be done before it, on condition that they are not dependent on the currently executing instruction.. Leading to out of order code execution.

The above code would see some improvement from an OoO superscalar architecture but that would mostly just coming from the superscalar aspect.

II. VLIW

Static multiple-issue processors (aka VLIW) use the compiler to decide which instructions to issue and execute simultaneously.

Instructions are always fetched, decoded and issued in pairs. If one instruction of the pair can not be used, it is replaced with a noop.



What this means is that ALU operations and Memory Access operations can happen at the same time. In the ase of the above code this means that loading or storing values for diff or sum will be done concurrently with other instructions.

We can imagine the code above in (pseudo) Assembly

LDR r2, &diff

MUL r3, r2, r2

STR r2, [&sum]

Under VLIW word some of these instructions could execute concurrently, otherwise they would execute as normal with noops.

III. Vector Processor

In computing, a vector processor or array processor is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors, compared to scalar processors, whose instructions operate on single data items. In order to be fully optimized by this type of processor the above code would have to be vectorized. However a vector processor could speed up the operation of the line -

Double diff = a[i]=mean

As this is a single operation on a set of data. This is exactly the type of thing a vector processor is for; notably array manipulation.

IV. Simultaneous Multithreading

In simultaneous multithreading, instructions from more than one thread can be executed in any given pipeline stage at a time. This is done without great changes to the basic processor architecture: the main additions needed are the ability to fetch instructions from multiple threads in a cycle, and a larger register file to hold data from multiple threads. The number of concurrent threads can be decided by the chip designers. Two concurrent threads per CPU core are common, but some processors support up to eight concurrent threads per core.

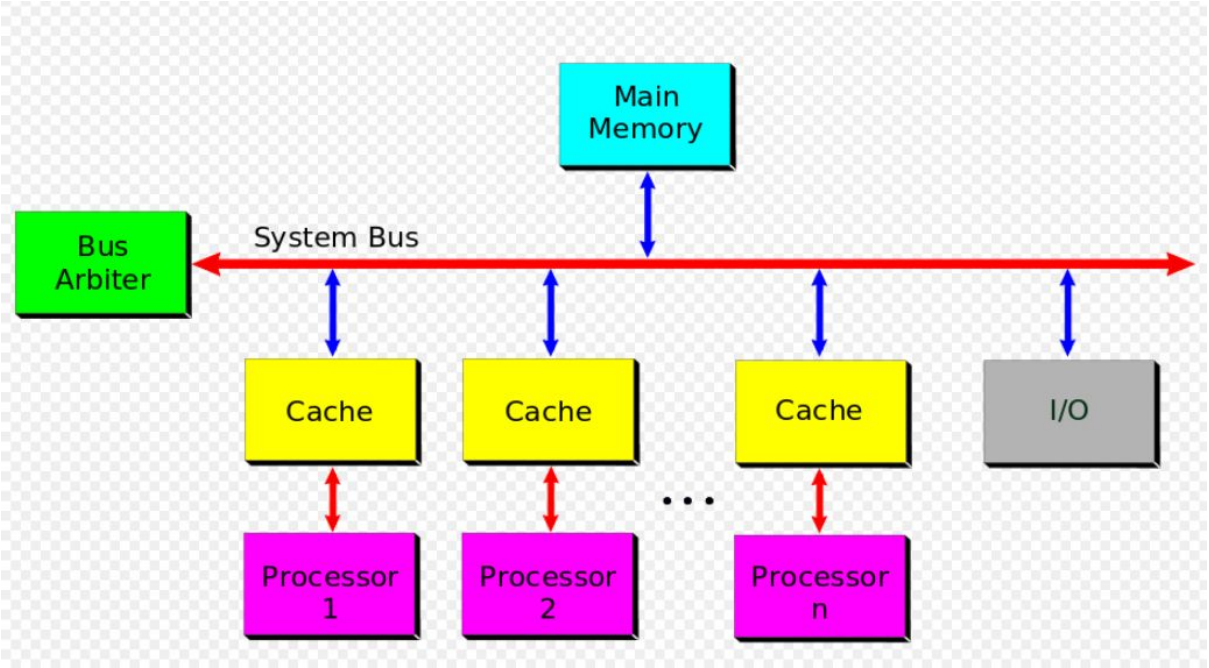
Simultaneous multithreading would have a positive impact on the above code. The ability to execute more pipeline stages at once would simply increase efficiency. More instructions could be executed in a given time interval. However assuming a large array, many of the instructions are sequential, such as the summation of diff *diff and rely on the answer from previous instructions and so would see negligible gains from multithreading.

V. Shared Memory Multicore Processor

A multi-core processor is a single computing component with two or more independent processing units called cores, which read and execute program instructions. Often these different cores can have different purposes/specializations. This naturally leads to speed up as multiple instructions are executed at the same time.Problems in this instance arise from the shard data aspect. When one core has access to a shared variable, the other core cannot access it, and may enter spin or die. Shared data means that cache coherency is a barrier to real speed improvement. For example in the above code, if sum is being written to in the for loop then the value of it cannot be changed by another core.

VI Multi chip symmetric processor

Symmetric multiprocessing (SMP) involves a multiprocessor computer hardware and software architecture where two or more identical processors are connected to a single, shared main memory, have full access to all input and output devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes.



The above code would see improvement from this architecture as the Operating system manages the synchronization of different processors. Threads would be scheduled in order to induce speed-up. In the above code multiple instructions for calculating the standard deviation would occur in parallel and then combined. The processor can communicate with each other via the bus, and the symmetric aspect means that we avoid data corruptions.

VII NUMA multi processor

Non-uniform memory access (NUMA) is a computer memory design where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors).

The above function is high in data locality the same (or sequential) areas of memory are repeatedly accessed which means that if a Numa architecture was used we would we some speed up

VIII Distributed Memory Multicomputer

In a distributed memory machine every cores has access to it's own, local, private version of memory. This can potentially introduce overhead as a memory coherence protocol must be implemented, however this excludes race conditions. Distributed memory put the onus on the programmer to think about data distribution, however if used properly it can be a secure way to implement concurrent programs. While race conditions are eliminated, and data hazards are avoid, data distribution brings its own set of problems.