Problem Collection

Programming and Architecture of Computing Systems

January 20, 2026

Contents

Preliminary Notes

This brief collection of problems is divided in three parts. The first part covers small questions, the second part includes test questions, and the third part refers to some longer exercises.

To report erratas, typos... please mail either alvabre@unizar.es, rgran@unizar.es or dario@unizar.es.

Small Questions

1. Please briefly respond to the following questions: ¿Is a concurrent application always parallel? ¿Is a parallel application always concurrent?
2. According to Amdahl's Law, for a program where the sequential part represents the 15% of the total, what would be the potential speed-up for a 16-core machine.
3. Can a processor execute instructions from two different instruction sets?
4. Enumerate what are the key design features of GPUs to allow a very fast context switching of wavefronts.
5. Explain what the conditional branching problem is on GPUs and how it is solved.
6. Make comparative analysis between a GPU and an ASIC.
7. The iron law of computer performance states that the execution time can be defined as: $Ex.\ \ Time = N \times CPI \times T_{cycle}$. To improve performance and save energy consumption, a new vector extension has been proposed. The extension reduces both the number of instructions and the frequency by half and 10%, respectively. Since the extra hardware complexity increases the cycles per instruction by 33%, could you please identify which alternative provides the lowest execution time.

Test Questions

1. The Local Data Share (LDS) cache on a GPU is used to:
   a) Amplify the regular cache bandwidth
   b) Execute atomic instructions
   c) Synchronization of wavefronts
   d) All of the above

   e) None of the above

2. In OpenCL, Local Memory is shared between:
   a) All the workitems of a global work domain
   b) Workitems in the same kernel launch
   c) Local Memory is an abstraction not present in OpenCL
   d) Workitems in the same workgroup
   e) None of the previous ones

3. In OpenCL, workitems that access global shared variable must explicitly assure memory order in order to avoid race conditions
   a) Always
   b) Just in case they do not belong to the same workgroup
   c) Just in case they do belong to the same workgroup
   d) Never

Exercises

1. The dot product algorithm takes two vectors of the same length and returns a single number. The number is the sum of the products of the corresponding entries in the input vectors.

   In C++, the algorithm can be coded as follows:

```
template<typename T>
T doc_product(const std::vector<T> &a, const std::vector<T> &b>
{

if(a.lenght() ≠ b.lenght()) {
    error( ... );
}

// initialize to 0 regardless the type
T dot_p{}; // Also T dot_p = T();

for(size_t i = 0; i < a.length(); ++i) {
  dot_p+=(a[i]*b[i]);
}
```

   Please answer the following questions:

   a. Implement the doc product using threads and static partitioning.
   b. Implement the doc product assuming you have the thread pool and the thread-safe queue from Laboratory 4.
   c. For the thread-pool version, would all tasks perform the same ammount of work?

2. Given an `std::vector<int>` array, could you please write a parallel algorithm that finds the minimum and maximum values of the array.

3. See Exercise 2 from the collection of exercises referring to metrics.

4. Write an OpenCL program that calculates the dot product of to integer arrays. Additionally to the kernel code, in the host side of the program, just focus on the buffer management, command-queue management and kernel launch.

   a. Please, analytically model the execution time of this work assuming the computational device has the following characteristics: 8 compute units, each compute unit has 128 parallel cores, each core has two floating-point arithmetic units and, frequency of the computational device is 1.5GHz. Assumption 1: just floating point instructions contribute to the execution time. Assumption 2: each FPU can process a floating point instruction per cycle.

5. Please write a parallel program that given an array of integer values, it finds those values that are prime and larger than a given element. The solution should follow a fork-join parallelism model in C++. To know whether an integer value is prime, you can assume that the function `bool is_prime(int n)` is available:

```
bool is_prime(int n) {
  if (n <= 3) {
    return n > 1;
  }

  if (((n % 2) == 0) || (( n % 3) == 0)) {
    return false;
  }

  for(int i = 5; i*i <= n; i+=6) {
    return false;
  }
  return true;
}
```

6. Sorting is one of the most important problem in computing. Its computational intensity makes sorting an ideal candidate for parallelization. One of the most common implementation is bucket sort where the input array is split between N buckets that are independently sorted and then concatenated.

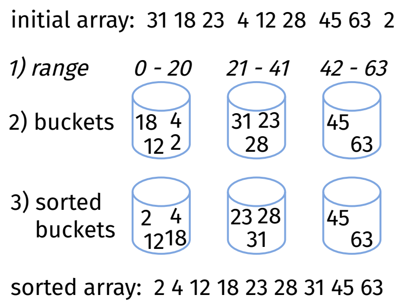

Figure 1: Bucket sort example

The upper figure shows bucket sort main steps: 1) Computation of the ranges. With the maximum value stored in the input array and the number of buckets, you compute the range, 63/3=21 in the example, and create the buckets. 2) Bucket insertion. Each value of the array goes to its bucket. 3) Sorting buckets. Each bucket is sortened independently, and 4) the sorted array is assembled by concatenating the arrays of each bucket.

a. Please implement a parallel version of the buckle sort algorithm. For sorting the buckets, you can use any standard sequential sorting algorithm as

```
template<typename T>
void insertion_sort(std::vector<T>& array)
{
  for(size_t i = 1; i < array.size(); ++i) {
    for(size_t j = i; (j > 0) && (array[j-1] > array[j]); --j) {
      std::swap(array[j], array[j-1]);
    }
  }
}
```

b. ¿Is there any phatological case where the parallel version could not be faster than the sequential version?

Notes: You can concatenate two `std::vector` arrays with the `insert` method; e.g., `dst.insert(dst.end(), src.begin(), src.end())`. The function `float std::floor(float arg)` computes the largest integer value not greater than `arg`.

7. Many Computer Vision applications require the computation of histograms, which help to understand the distributions of a set of numbers. To compute an histogram, you need to visit all elements of the set, compute their bucket, and then increase the corresponding counter of that bucket. For example, if the input array contains this set of numbers {0, 1, 1, 1, 2, 3}, the output histogram with 4 buckets will be {1, 3, 1, 1}.

a. Please write a parallel version of an histogram for integer values following a fork-join approximation in C++. The histogram result will be stored in an array of atomic variables and you have to minimize contention on this array.

You can assume the following initial skeleton:

```
int main() {

    const size_t N = 1024*8; // array size
    const size_t m_buckets = 32; // buckets
    const size_t n_threads = 8;

    std::vector<int> array; // please assume this array has been already initialized
    std::vector<std::atomic<init>> histogram(m_buckets);

    // ...
```

b. What could be the maximum speed-up of this implementation?

8. Matrix multiplication belongs to the most used algorithm list in robotics, graphics, and computer vision applications. Therefore, almost every application requires a fast parallel matrix multiplication algorithm.

Assuming a basic 3 nested loop serial implementation in C++:

```
using fmatrix = matrix<float>;

fmatrix matrix_multiply(const fmatrix &a, const fmatrix &b)
{
    fmatrix c{rows, cols};

    for(size_t i = 0; i < a.rows(); ++i) {
       for(size_t j = 0; j < b.cols(); ++j) {
           float val{0.0f};
           for(size_t k = 0; k < a.cols(); ++k) {
               val+=a(i, k) * b(k, j);
           }
           c(i, j)=val;
       }
    }
    return c;
}
```

a. Using `std::async`, write a parallel version `async_col_matrix_multiply` that when the number of concurrent threads supported by the system is 1, only uses 1 thread. Otherwise, the maximum number of concurrent `std::async` will be the number of columns of a matrix, `a.cols()`.

b. Imagine you have access to a 1024 multi-core machine, and `a.cols()` is always smaller than 128. What would be the maximum speed-up of the parallel version implemented in step a? Could you please write another version, `asyc_matrix_multiply` able to extract all the possible parallelism for this 1024 multi-core machine.

c. Assuming that every `std::async` creates a new thread on every invocation, could an implementation based on a thread-pool be faster than the version based on `std::async`? Why?

Note: You can assume that the `matrix<float>` *class provides all requirements for storing matrices. If you need extra trivial methods of the class besides rows and cols, please fell free to use them without writing their implementation.*

9. Alpha compositing is a computer graphics method that combines a foreground and a background images to simulate transparency. With 2D images, alpha compositing extends each pixel with an additional value representing transparency. This new alpha value ranges between 0, fully transparency, and 1 (fully opaque). For example, assuming two images named f and b, so that f is over b, in another words, f is the foreground, the over operator can be computed following these equations:

$$\alpha_o = \alpha_f + \alpha_b(1 - \alpha_f)$$

$$p_o = \frac{p_f \alpha_f + p_b \alpha_b (1 - \alpha_f)}{\alpha_o}$$

Where p_x represents the three color channels (red, green, blue) of each pixel and α_x represents the alpha value of the output (o), foreground (f), and background images (b).

Assuming a pixel and image classes as follows:

```
struct pixel {
  public:
    uint8_t red, green, blue;
    uint8_t alpha; // 255 corresponds to opaque
};

template <typename T, size_t N, size_t M>
class image {
  using storage_type = std::array<std::array<T, M>, N>;
  public:
    image(){};
    T& operator()(size_t i, size_t j) {return _array[i][j];};
    T operator()(size_t i, size_t j) const {return _array[i][j];};
  private:
    storage_type _array;
};
const size_t height = 128, width = 128;
using alpha_image = image<pixel, height, width>;
```

a. Please write a sequential version of a `alpha_image alpha_over_operator(const alpha_image& f, const alpha_image& b)` free function that returns the result of performing an `alpha_over_operator` on two input images. You can use `std::clamp(uint8_t v, uint8_t lo, uint8_t hi)` to clamp the resulting operations if required.

b. Please write a parallel version of `alpha_over_operator` that extracts parallelism and pick between data and task level parallelism depending on the regularity of the problem.

10. Please briefly describe and correct the concurrency programming errors, if any, in the following fragments of code:

1.
```
void f() { … }
int main() {
  std::thread t(f);
  t.deatch();
  t.join();
}
```

2.
```
int fib(int n) {
    std::mutex m;
    if(n < 2) { return n; }

    int result = 0;
    {
        std::lock_guard lk(m);
        auto fib_n_1 = std::async(std::launch::async, fib, n - 1).get();
        auto fib_n_2 = std::async(std::launch::deferred, fib, n - 2).get();
        result = fib_n_1 + fib_n_2;
    }
    return result;
}
```

3.
```
std::mutex m;
std::condition_variable cv;

void producer() { cv.notify_one(); }
void consumer() {
   std::unique_lock<std::mutex> lk(m);
```

```
        cv.wait(lk);
    }
    int main () {
        std::thread tp(producer);
        std::thread tc(consumer);
        tp.join();
        tc.join();
    }
```

Note: Please assume the inclusion of all required headers. They have been excluded to save space.

11. Event-based cameras produce a unidimensional stream of events that can include timestamps, pixel position, and polarity. The number of events, and consequently the stream size, varies depending on the scene being observed. One feasible parallelization approach for unknown-size streams is the divide-and-conquer pattern. In this pattern, when the work size exceeds a predetermined threshold, the work is divided into two smaller chunks that are executed in parallel. This process repeats recursively until the size of each chunk falls below the threshold.

Please implement a parallel `size_t count_polarity(const stream& s, const     size_t min_threshold = 128)` method that returns the number of events whose polarity is true. For the minimum threshold, you can assume 128 events.

The `event` and `stream` classes are as follows:

```
class event
{
   std::pair<size_t, size_t> _pos;
   bool _polarity;
   std::chrono::time_point<std::chrono::steady_clock> _timestamp;

 public:
   event() : _pos(), _polarity(), _timestamp() {}
   event(bool polarity) : _pos(), _polarity(polarity), _timestamp() {}
   bool polarity() const { return _polarity; }
};

using stream = std::vector<event>;
```

12. Max pooling is an important operation in convolutional neural networks (CNN) that downsamples feature maps, which helps reducing the spacial dimensions of the data with retaining the key information.

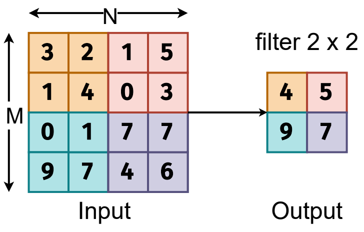

Figure 2: Max Pooling Example

The accompanying figure illustrates the input and resultant feature map of a max pooling operation, configured with a 2×2 filter. Max pooling systematically traverses the input feature map, applying a non-overlapping window (the filter) to distinct regions. Within each 2×2 region, the algorithm identifies and selects the pixel possessing the maximum value, which is then propagated to the corresponding position in the output feature map.

a. Write a serial version of a max pooling operating assuming a filter size of 2 by 2 and a input feature map size of 128 by 128 for `unsigned int` values. What will be the size of the output? Please assume the following definitions for completing the code:

```
template <typename T, size_t N, size_t M>
class feature_map
{
 using storage_type = std::array<std::array<T, M>, N>;
 public:
 image(){};
```

```
     T& operator()(size_t i, size_t j) {return _array[i][j];};
     T operator()(size_t i, size_t j) const {return _array[i][j];};

     private:
     storage_type _array;
    };

    template <typename T, size_t in_N, size_t in_M, size_t filter_N, size_t filter_M>
    void max_pooling_serial(const feature_map& in<T, in_N, in_M>, feature_map& out<T,
     ↪   ...>);
```

Where N, M, `filter_N`, and `filter_M` represents the rows and columns for both the input feature map and the filter, respectively.

b. If the max pooling operation is paralellizable, please write a parallel version justifying your choice between data and task-level parallelism. Also your solution should have the same arguments and return type of the serial version.

13. In Neural networks, sparsification is a technique aimed at reducing the amount of computation by avoid multiplication and addition operations in convolutions. Its operation is as follows. When a kernel weight is smaller than a threshold, the weight is replaced by a zero. As a result, the sparse kernel neither stores the value nor performs the multiplication and addition operations when applying the filter. For example assuming a 2x2 kernel with values `[[0.1, 0.2], [0.1, [0.1]]`, the equivalent sparse kernel will only store `[[0, 1, 0.2]]` for a threshold of 0.15. The 3-tuple correspond to row 0, column 1, and value 0.2, which is the single element of the kernel larger than 0.15. Assumming the following definitions:

```
template <typename T, size_t N, size_t M>
class matrix
{
    using storage_type = std::array<std::array<T, M>, N>;
    public:
    matrix(){};
    size_t rows() const { return _array.size(); }
    size_t cols() const { return _array[0].size(); }
    T& operator()(size_t i, size_t j) {return _array[i][j];};
    T operator()(size_t i, size_t j) const {return _array[i][j];};

    private:
    storage_type _array;
};

using value_type = float;
const size_t N = 480;
const size_t M = 640;
using image = matrix<value_type, N, M>;
using kernel = matrix<value_type, 3, 3>;
```

Please answer the following questions:

a. Write a `sparse_kernel` class definition where each sparse element corresponds to a struct made of three data members: `size_t row`, `size_t col`, and `value_type value`. The `sparse_kernel` class has to provide methods for returning the number of rows and columns of the kernel at least. The class should use the minimum required space to save all values larger than the threshold and could store kernels of any 2D width and height.

b. Write a free function `void apply_sparse_kernel(const image& in_image, const sparse_kernel& skernel, image& out_image, const size_t begin_row, const size_t end_row)` that applies the sparse kernel to the input image for the rows between `begin_row` and `end_row`. As a reference, please consider that he pseudo-code for an standard convolution on a dense (non sparse) kernel is as follows:

```
for row in in_image rows {
  for col in in_image cols {
    sum = 0.0f;
    for krow in kernel rows {
      for kcol in kernel cols {
        if is_valid_position(row, col, krow, kcol) {
          sum += kernel(krow, kcol) * in_image(row + krow, col + kcol);
        }
      }
    }
    out_image(row, col) = sum;
  }
}
```

Both input and output images have the same size, and for the edges, if the position is invalid, the computation can be avoided.

c. Implement a free function `void parallel_sparse_convolution(const image& in_image, const sparse_kernel& skernel, image& out_image)` that perform the operation on the whole image concurrently. Justify the choice between data or task level parallelism and use as many parallel threads as possible in the running machine.