Lattice theory - Project 1

Giorgio Daneri[1]

University of Luxembourg.

Contributing authors: giorgio.daneri.001@student.uni.lu;

1 Introduction

This project consists of two exercises that concern the application of lattice theory to parallel programming. The first exercise consists in implementing an algorithm to find the minimum of an array using different approaches. The core concept is the comparison between a more traditional approach, which uses a parallel reduction pattern, and a lattice-based implementation based on a fixed point iteration. The former makes use of the pessimistic approach, which introduces locks, atomic operations and critical sections to avoid data races, no matter how rare they are. The latter employs an optimistic approach that is free of data synchronization constructs and opts for additional work in order to reach a coherent and stable result, hence the term fixed point iteration. The second exercise consists in implementing an algorithm for a combinatorial optimization problem, which consists in finding all possible solutions given n variables, the relative domains and a set of constraints. Again we compare a standard implementation with one based on lattice theory and the fixed point model of computation.

2 Parallel minimum

The implementation is carried out in a single file, which contains both the main program, the serial algorithm, the function that handles CUDA memory allocation, data movement, kernel launch and result gathering, as well as the two CUDA kernels. The main program simply creates a vector of random numbers with arbitrary dimensions, and calls all the other function of the program. These include the `kernel_wrapper` and the function to compute the minimum on the CPU, of which it measures the execution time.

2.1 Kernel_wrapper

This function is tasked with allocating memory for all necessary data structures on the device, copy the corresponding data, and launching both kernels to compare their performance. For all the `cudaMalloc` and `cudaMemcpy` primitives I added a standard error handling methods to pinpoint eventual bugs in the code. Then comes the implementation of the fixed point iteration. It consists in a loop that iterates until the solution at the previous step is not equal to the one at the current step. Inside the loop, it launches the CUDA kernel with the desired number of threads, specified in terms of `GRID_SIZE` and `BLOCK_SIZE`. It measures the execution time of the whole fixed point iteration and prints the found minimum to ensure the correctness of the algorithm.

2.2 Fixed Point Kernel

The kernel called inside the fixed point iteration exploits shared memory to reduce the memory access time and increase the efficiency of the computations. As usual, I compute the thread global thread id, as well as that inside the corresponding block. Then I declare a block of shared memory of dimension `BLOCK_SIZE`, which is populated with the corresponding array portion, while checking threads do not go out of bounds.

Then each thread eventually updates the global minimum value without any synchronization. This does not avert race conditions, which occur since the write operations are done to a single memory location. Though the fixed point iteration, which consists in a monotone function, does not require any kind of synchronization between the CUDA threads. This is because a monotonic function can only retain or decrease its value, so the iteration eventually converges to the correct result, at the cost of additional work.

2.3 Parallel Reduction Kernel

The traditional approach consists in a parallel reduction pattern that once again uses shared memory to achieve better performance. The initial steps of allocating and populating the shared memory blocks are the same. After the shared memory has been prepared, the threads are synchronized. The reduction pattern reduced the complexity of associative operations, i.e. computing the sum or the minimum, from linear to logarithmic. The computation is arranged in a tree-like structure, where the height of the tree is logarithmic in the number of nodes. At each step the number of active threads if halved, as well as the stride. This amount to making the memory accesses more coalesced, i.e. threads are accessing values that are stored contiguously in memory (stored in the same logical blocks), requiring less overall memory accesses. After the computation of the local minima for each block of share memory, the global minimum is computed by applying the built-in `atomicMin` function. This amount to using the first thread of each block to perform an update on the global variable, if the local minimum is lower than the current value of the global minimum.

2.4 Performance results

Since it was prescribed to use a grid size and a block size of 1024, the dimension of the input should be at least equal to the number of total threads, i.e. 1048576. Though this way we are not fully exploiting the power of GPUs, hence I chose an input size of 10^9. The results are the following:

```
Time taken by fixed point iteration: 6.5e-05
Time taken by parallel reduction: 2.8e-05
Time taken by CPU: 2.01417
```

The performance of the parallel reduction algorithm is comparable to that of the fixed point implementation, even though the former seems to be slightly faster.

3 Combinatorial Optimization

The problem consists in finding all the possible assignments of n variables that satisfy a set of constraints, where the assigned values are within the domains of each variable. I personally wrote the algorithm from scratch since I came to the conclusion that `nqueens` is too specific with respect to its set of constraints. The algorithm to solve this combinatorial problem needs to be more generic with respect to the applicable constraints.

3.1 Core Algorithm

I use a DFS (depth-first search) algorithm to iteratively build a tree of all assignments compatible with the given constraints, where each leaf node is a solution. The height of the tree is equal to the number of variables, since it corresponds to a valid assignment for each one of them. The basic component of the tree is a `Node struct`, which holds the currently assigned values at its depth, as well as the last assigned variable. This information is redundant for the CPU version, but necessary when computing on the device, since the size of a vector cannot be known when dealing with pointers. At first I stored a copy of the domains for each node in the tree, but then I introduced an optimization that only requires a global domain variable, that is reset accordingly to the backtracking. This saves a significant amount of memory and computational time. Note that the maximum number of nodes in the stack in equal to the number of variables, since I create one node for each variable that is assigned and then remove it or modify it with the next valid value in its domain. The algorithm proceeds as follows:

1. create the root node of the tree, where the variable x_1 is assigned the first valid value in its domains, i.e. 0

2. apply the constraints to remove all invalid values from the domains of the variables $x_j, \forall j > i$, since they have not been assigned yet. This is done in a fixed point loop, to make sure that if the domain of a variable is reduced to a singleton, this is treated as another assignment, so the constraints are applied again until the current node configurations does not change. All the variables which domain is reduced to a singleton are stored in an ad hoc vector. When a new singleton is created, we need to insert the value assigned to the relative variable in the `node.assignedVals` vector, extending its size and filling it with -1 values for all the variables that have not yet been assigned or which domain has not been reduced to a singleton (i.e. for all variables greater than the last assigned one, but smaller than the one which domain has been reduced to a singleton)
3. check if the current node is a leaf node, i.e. the last assigned variable is the *n-th* one. There are two possible behaviors:
   - the domain of x_n is empty, so we need to perform backtracking and find the variable with the biggest index, whose domain is not empty, and assign it the next valid value. All the domains with a greater index are reset and step (2) is performed again
   - the domain of x_n is not empty, so we add a solution for each non-zero value in its domain. We need not apply again the constraints since we already reached a fixed point and the domains would not change. After expiring all valid values in the domain, perform backtracking as in the previous item
4. if the current node is not a leaf, we need to check whether singletons domains exist or not. If this is true, we need have two possibilities for the next variable in order:
   - its domain is empty, so the current configuration does not admit a solution. We just need to modify the current node by setting the `branchedVar` to n-1, so that the algorithm can perform backtracking on it
   - its domain is not empty and its domain has been reduced to a singleton, so we need not find the next value to be assigned since we already have it. If the current variable is the last one, increase the number of solutions

   In all other cases, we need to assign the next variable in increasing order as well as the next valid value in its domain, since the iteration over the constraints has reached a fixed point but we did not obtain a solution. After this (2) is repeated.

3.2 Domain Restriction

For what concerns the removal of values from the domains, the rationale is the following.

1. iterate over all the constraints and extract the left and right hand side variables for each one.
2. check if both variables are already assigned, i.e. their index is smaller than that of the last assigned one. If it is, move to the next constraints, since the domain of both has already been reduced to a singleton
3. if one of the two variables is assigned, remove its value from the domain of the other one, since they are not compatible
4. if none is assigned, do nothing

The function that performs domain restriction in called `restrict_domains` and is called inside a fixed point loop until the domains do not change from one iteration to the following one. A side note: when working with a limited number of variables, with relatively big domains, it is unlikely that the fixed point loop runs more than one iteration. It is best applied when we have a large number of constraints and variables, which have a small domain. This makes it more likely that the domain of a variable is reduced to a singleton following the domain restriction, making it necessary to execute it again.

3.3 GPU implementation

The GPU implementation features a CUDA kernel to perform the domain restriction in parallel, without the need of synchronization constructs and atomic access to the domain structure, which is again global for all the nodes. Since I used a `std::vector<std::vector<bool>>` to store the domains in the serial version, it is necessary to flatten the domains into a single array of `uint8_t`. This is done before each call of the kernel, while afterwards we need to reconstruct the domains from the flattened version. Due to time constraints, I was not able to convert the domains structure into a `uint8_t**`, which would have averted the need of these operations. The structure of the algorithm is exactly the same as the serial

one. The only changes concern the fixed point iteration, where the kernel is called. Before executing it, the domains, assigned values and last assigned variable of the nodes are copied to the device. After the kernel has finished, the domains are copied back to the host. Since this has to be done for each iteration of the fixed point loop, the performance of the implementation proves to be very inefficient. This is due to the fact that the program is memory bound and the computation to be done on the device is very limited, especially for a small number of variables with large domains, which neglect the purpose of the fixed point concept.
As for the acceleration on the GPU, I implemented two kernels, one using global memory and another one that leverages shared memory to store the constraints. Unfortunately, the latter did not influence the overall execution time of the program, since the time taken by the kernel is negligible when compared to that required by `cudaMemcpyAsync` functions mentioned earlier. Moreover, I did not register a performance improvement with the introduction of the shared memory, probably due to the limited number of constraints and the time taken to allocate the memory portion and move the data from the global memory to the shared buffer. The structure of the kernel mimics the CPU version of `restrict_domains`, with the additional need of making sure that threads do not go out of bounds when selecting the data to be processed.

3.4 Kernel Performance Comparison

The comparison is conducted with respect to the `pco_3.txt` and `pco_5.txt` tests, which consists of 3 and 5 variables respectively, with randomly generated upper bounds for the domains comprised between 0 and 100. The results of averaging 10 runs are the following:

```
CPU execution time to solve pco_3: 0.000176 s
GPU execution time to solve pco_3: 0.125 s
CPU execution time to solve pco_5: 0.773 s
GPU execution time to solve pco_5: 25.3 s
```