PCCP assignment

Daniele Ferrario

December 5, 2024

1 Warmup

1.1 Sequential

The sequential algorithm can be found in *sequential.cpp*.

1.2 Fixedpoint

The algorithm can be found in *fixedpoint.cu*. The relative kernel is this one:

```
__global__ void fp_min(int *d_arr, int *d_min, int N){
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if(idx < N){
        if(d_arr[idx] < *d_min){
            *d_min = d_arr[idx];
        }
    }
}
```

We are exploiting the fact that collisions are relatively rare, so we are following this optimistic approach by avoiding atomics.

```
    // MAX_DATA_RACES+2 iteration at its worst
    for(int i=0; i<MAX_DATA_RACES+2; i++){
        if(min == old_min){
            break;
        }
        old_min = min;
        fp_min<<<grid_size,  block_size>>>(d_arr, d_min, N);
        cudaMemcpy(&min, d_min, sizeof(int), cudaMemcpyDeviceToHost);
    }
```

We run the kernel at max MAX_DATA_RACES, which is $N-1$, but in practice we stop after a few iterations, since collisions are relatively rare.

1.3 Parallel reduction

The algorithm can be found in *reduction.cu*. There are two versions, with the second one using shared memory in the kernel.

```
__global__ void reduce_min_2_kernel(int *input, int *output, int N){
    extern __shared__ int sdata[];
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int tid = threadIdx.x;

    // Copy all elements to shared memory of block
    sdata[tid] = input[i];
    __syncthreads();

    for(unsigned int s = blockDim.x/2; s > 0; s >>= 1){
        if(tid < s){
            if(sdata[tid + s] < sdata[tid]){
                sdata[tid] = sdata[tid + s];
            }
        }
        __syncthreads();
    }

    // Write the result for this block to global memory
    if(tid == 0){
        output[blockIdx.x] = sdata[0];
    }

}
```

Then we can find the minimum sequentially, iterating on each block result.

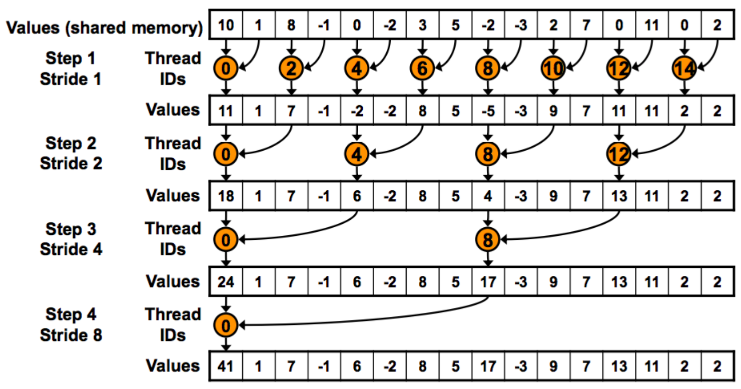

Figure 1: Example of reduction with stride (Sum of elements, in this case.)

1.4 Atomics

The algorithm can be found in *atomic_min.cu*. The kernel is very simple:

```
__global__ void atomic_min_naive_kernel(int *d_arr, int *d_min, int N){
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if(idx < N){
        atomicMin(d_min, d_arr[idx]);
    }

}
```

Each thread cannot access *d_min* until the lock has been released by another thread, making this kernel essentially sequential.

2 Combinatorial optimization

To solve this problem, I implemented a similar approach used in the *n_queens* file as requested, creating a tree to span all possible solutions, which correspond to the leaves node. We use *Node* objects containing the *domains* of possible values during the fixed point iteration. **The fixed point iteration in this context means applying the constraints that reduce the domains of the variables, while it is possible.**

2.1 Sequential

The logic behind the algorithm is the following:

1. Pop a node from the stack, containing the nodes of the tree.
2. If the node is a leaf, it means we found a solution, being the domains of the variables all singletons. Otherwise we proceed.
3. We apply the fixed point iteration. It means that we update the domains of the variables following the constraints, until the method *update_domains()* no more modifies the *domains*.
4. Given $i =$ "*depth of node*", we branch on the variable i, creating as many children nodes as there are values in the domain of variable i. Each of the children will have the same domain of the parent, but the domain of $variable_i$ will be a singleton.

The domain of the variables are implemented as a boolean mask (*bool_mask.hpp*).

2.1.1 Domain object in detail

Variable	Domain
x_0	1, 1, 0, 1, 1
x_1	1, 1, 0
x_2	1, 0, 1, 1
$\vdots$	$\vdots$

This means that the first variable domain is 0,1,3,4.

2.1.2 Update domains in detail

When applying the constraint, we can restrict the domain only if one of the two involved variables is a singleton. So we find all the variables which can assume only one value and apply the constraint to the other ones.

```
// For each singleton
for(auto& p : unique_true_values){
    int i = p.first;
    int j = p.second; // Column where the unique true value is found

    // If there is a constraint between i (singleton variable) and k,
    // then I remove the value at column j from the domain
    // of the variables k that have a constraint
    for(int k = 0; k < domains.rows; k++){
        if(C[i][k] == 1){
            if(domains[k][j]){
                domains[k][j] = false;
                updated = true;
            }
        }
    }

}
```

2.2 Parallel

The parallel implementation focuses on parallelizing the fixed point iteration.

Firstly, the algorithm searches for the singletons variables in parallel spawning n threads. In particular this kernel searches for the index of the last "1" value, and notes in *d_row_is_singleton* if it is the only one in the array.

```
__global__ void find_true_index_kernel(BoolMatrixGPU domain,
size_t *d_true_indices, bool *d_row_is_singleton, size_t n){

    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if(i >= n)
        return;

    assert(domain.data != nullptr);
    assert(domain.cols != nullptr);

    size_t row_len = domain.cols[i];
    size_t *d_offsets = domain.d_offsets;

    int count = 0;
    for(int j = 0; j < row_len; j++){
        if(domain.data[d_offsets[i] + j] == true){
            d_true_indices[i] = j;
            count ++;
        }
    }

    d_row_is_singleton[i] = count == 1;

}
```

Secondly, spawning n^2 threads to target each of the constraint values, we reduce the domains, by checking each pair of variables.

```
__global__ void update_domains_kernel(bool *data, size_t rows, size_t *d_offsets,
int *d_C, bool *d_updated, size_t *d_last_true_row_indices,
bool *d_row_is_singleton){

    int i = blockIdx.x * blockDim.x + threadIdx.x;

    size_t n = rows;
    if(i >= n*n)
        return;

    int row_a = i / n;
    int row_b = i % n;

    // If there is a constraint between the variables
    if(d_C[row_a*n + row_b] == 1){

        // Proceed only if it is a singleton
        if(!d_row_is_singleton[row_a])
            return;

        size_t c = d_last_true_row_indices[row_a];
        // Set that column in row_b to false
        if(data[d_offsets[row_b] + c] == true){
            data[d_offsets[row_b] + c] = false;
            *d_updated = true;
        }
    }

}
```

2.3 Performances

Unfortunally, the performances of the parallel implementation do not reflect the theoretical speedup that the current implementation provides. After profiling the code, it is evident that run-time is greatly affected by the allocation and de-allocation of data in the device memory when creating the children, even if the transfers are ”device-to-device”.

This suggest that a different implementation with less data copies would be beneficial for the given problem. However, if we just consider the update of the domains using the fixed-point approach, the performances increase, and would be more evident when applied to a much bigger number of variables.

3 Pccp considerations

The implementation proposed indeed follows the model of the parallel concurrent constraint programming model. In particular:

PCCP Concept	
Domains	The sets $D(x_0), D(x_1), D(x_2)$ represent the possible values each variable can take.
Constraints	Constraints like $x_0 \neq x_1$ ecc. define relationships between variable domains.
Constraint Store	The current state of all variable domains acts as the **constraint store** shared by all processes. Each domain can be seen as a Lattice L of integer numbers within a range, and the overall store $S = L_1 \times L_2 \times ...L_n$
Tell Operation	**Reducing a domain** (e.g., removing a value) corresponds to a **tell** operation, updating the store.
Ask Operation	**Checking if a constraint is satisfied** corresponds to an **ask** operation.
Parallel Execution	Multiple constraints are evaluated **in parallel**, each modifying the domain independently.
Fixed Point	The process halts when no further domain reductions are possible (i.e., there are no singletons and constraints to apply).

Table 1: PCCP Concepts and Their Correspondence in the Domain Reduction Algorithm

The theorem of Equivalence Between Sequential and Parallel Operators guarantees that the fixed point of our reductive PCCP processes is the same when ran in parallel and sequentially.