ESIGELEC
INGÉNIEUR.E.S GÉNÉRALISTES
SYSTÈMES INTELLIGENTS ET CONNECTÉS

Parallel Programming

Name:
Paul BLIGNY
Group Members:
Paul BLIGNY
Nathan PIVON

October 2024

1. Exercise 1

The objective of this exercise is to understand key elements in measuring code performance on a server. Specifically, the program calculates the execution time of functions for different input data sizes, with results saved to a CSV file. After execution, the recorded times will be plotted as a function of data size to identify each function's performance. This enables a comparison of initialization and execution runtimes across different functions.

a. Code

To reach the goal of this exercise, we modified the function "run_once". We measured the time with 3 other functions (line 33).

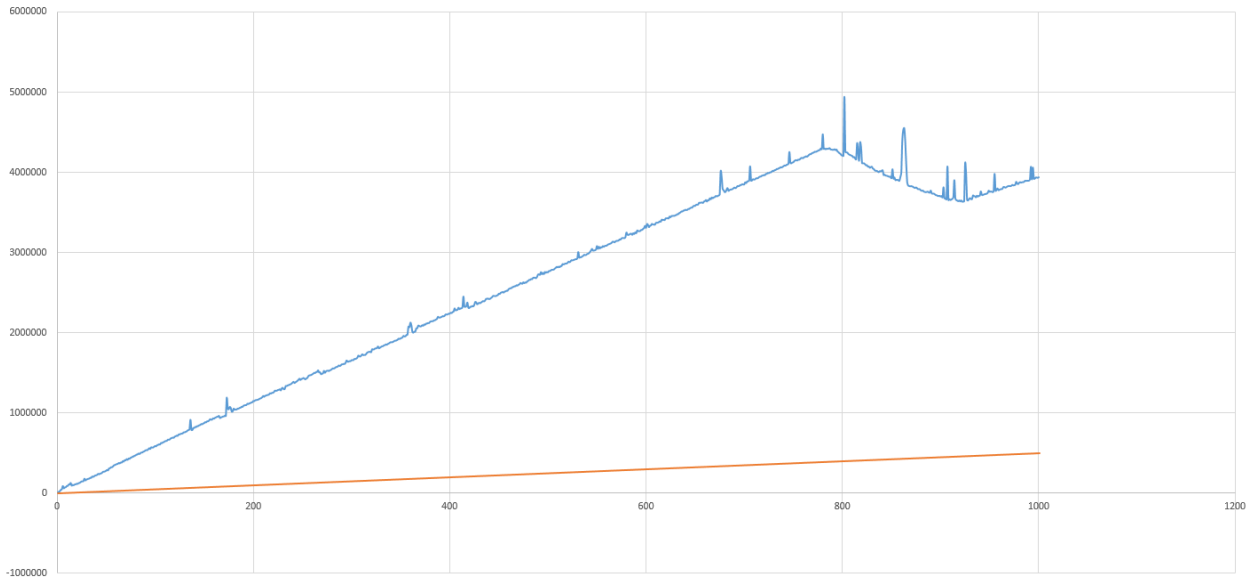
```
16  //run the full benchmark once for a given datasize
17  static void run_once(int32_t *data, ssize_t data_size, std::ostream &s)
18  {
19      perftimer t;
20
21      //initialization
22      t.start();
23          init_rand(data,(int32_t)data_size);
24      t.stop();
25
26      //print init time
27      s << data_size << "," << t.duration_ns();
28
29      //run the benchmarked function
30      t.start();
31          ////////////////////////////////////////////////////////
32          //Call benchmark functions here : sum_array, nbdiv_array, frequencies_array
33          frequencies_array(data,(int32_t)data_size);
34          ////////////////////////////////////////////////////////
35          //std::cerr << "Datasize " << data_size << " / Max : " << max << std::endl;
36          std::cerr << "Datasize " << data_size << std::endl;
37      t.stop();
38
39      //print run time
40      s << "," << t.duration_ns() << std::endl;
41  }
```

b. Results

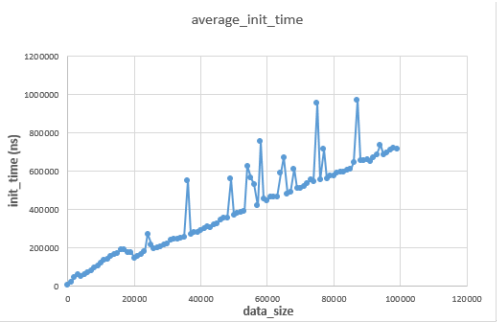
Basic Function:

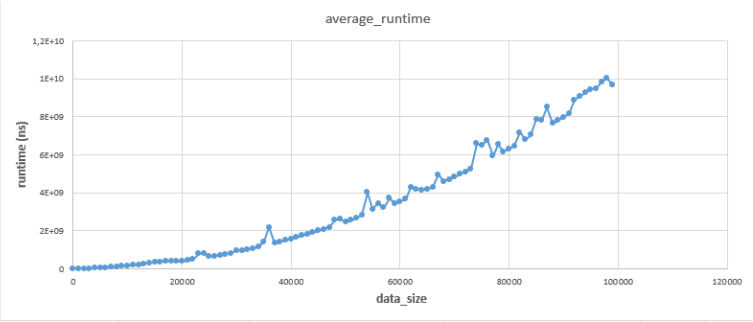
This graph shows the results of the code using very high parameters. In this case (figure 1), the max number of data cells (data_size_max) is set to 500k and the step between benchmarks (data_size_step) is set to 1000. The blue line corresponds to the execution runtime and the orange line corresponds to the initialization runtime. As we can see, they are very soft, this is due to the high number of data cells. The slop after 800 steps can come from a maximum cache reached on the server.

```
const int32_t data_size_max = 500000;                    //max number of data cells
const int32_t data_size_step = data_size_max / 1000;     //step between benchmarks (e.g. 1000 steps)
```



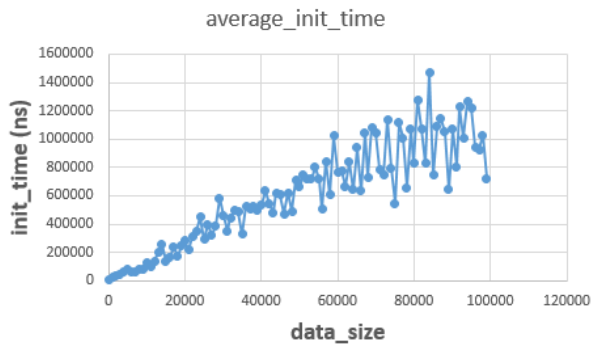
Frequencies Array:

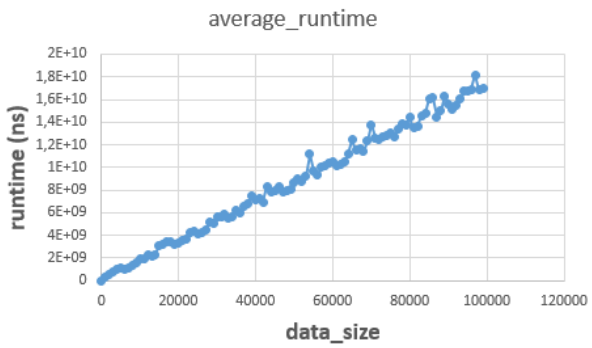




Both graphs show an increase of the init_time and execution time because of the data size (predictable). We can also see peaks that can be due to memory overflow or server overflow.

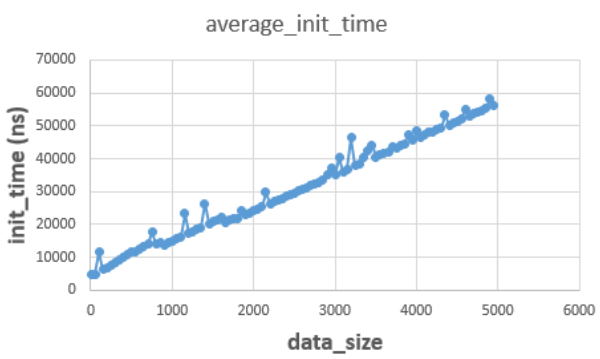
Nb_div Array:

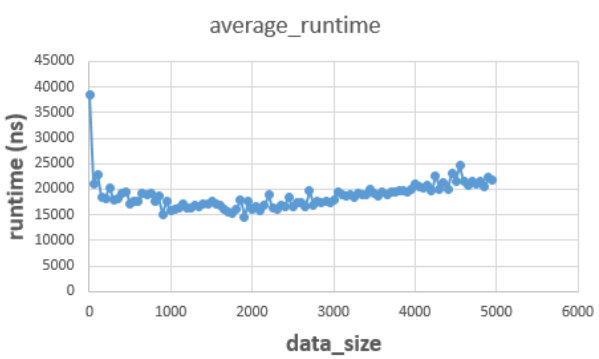




The curves are similar to the function 'frequencies array' but they are more linear. Compared to the previous graphs, this set shows more pronounced fluctuations in initialization time, indicating greater instability as data size grows, while the execution time trend remains similarly linear.

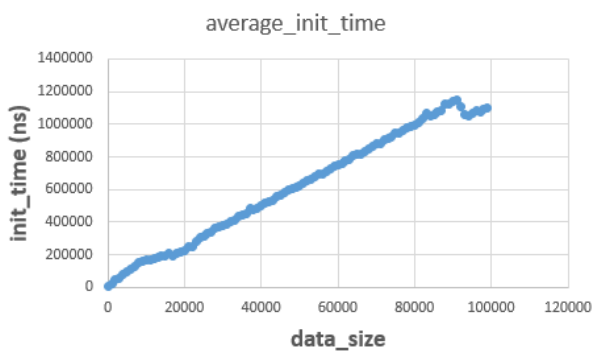
Find Max Array:

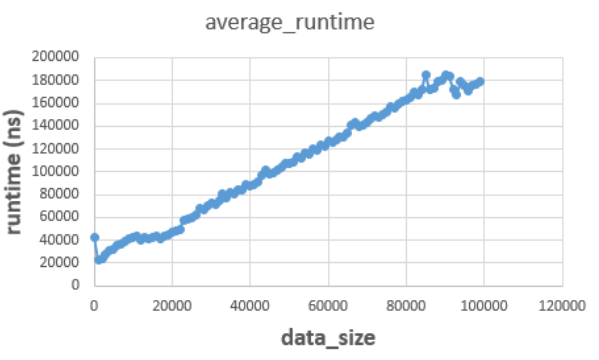




In this graph, the initialization time grows up as the data size increases, following a straight line with small fluctuations. The execution time behaves differently: it starts high, then quickly drops and stays stable as data size increases. This graph is more stable, especially in execution time. Here, the initialization time grows in a predictable way, while the execution time stays flat, showing better performance stability with larger data sizes.

Sum Array:





The sum array graphs is very linear at the beginning and increases slowly. As soon as a maximum amount of steps is reached, the init_time decreases a little bit. None of the curves has any major drops or irregularities. This set is more stable than the earlier graphs, especially in execution time. Unlike previous data where there were large fluctuations or a plateau effect, here both initialization and execution times rise in a predictable and linear way.

c. Comments

This exercise explores using CUDA for parallel programming on a GPU, focusing on how GPU threads and blocks handle tasks. We ran a program that printed messages from each GPU thread to understand the basics of thread and block structures. We also generated several graphs to see the impact of data size on initialisation and runtime. The graphs showed that, generally, initialisation time increased steadily with data size, while runtime had some variability, sometimes increasing steadily, sometimes fluctuating or decreasing due to cache memory. These methods offer some advantages: they allow faster processing of large data by running many tasks in parallel, as we will see in the next exercise.

2. Exercise 2

5. Define how the task will be divided in threads;

To divide the task into threads for the Game of Life, we can divide the grid by the number of threads. For example, if we have 4 threads, the grid will be divided in 4 zones. Each thread will be assigned to a specific zone. The repartition can be like a grid, in rows or in columns:

Thread 1	Thread 2
Thread 3	Thread 4

Thread 1
Thread 2
Thread 3
Thread 4

Thread 1	Thread 2	Thread 3	Thread 4

6. Decide what information is necessary to be given to each thread;

Each thread will know the information of the threads next to it. So that it will be able to calculate the boxes to modify (or to not modify). The thread will also give the information of its zone to each thread next to it.

3. Exercise 3

8. Run the code and observe and the output;

As Predicted, the output, shows how each thread and block is launched independently on the GPU. This is the point of parallel programming using CUDA.

```
Thread 1 in block 7
Thread 0 in block 6
Thread 1 in block 6
Thread 0 in block 13
Thread 1 in block 13
Thread 0 in block 14
Thread 1 in block 14
End
```

To answer this question, we modified the function “hello”:

```
5   __global__ void hello() {
6       printf("Grid Dim: (%d, %d, %d) - Block Dim: (%d, %d, %d)\n",
7              gridDim.x, gridDim.y, gridDim.z,
8              blockDim.x, blockDim.y, blockDim.z);
9
10      printf("Block %d - Thread %d: (ThreadIdx: %d, %d, %d) in BlockIdx: (%d, %d, %d)\n",
11             blockIdx.x, threadIdx.x,
12             threadIdx.x, threadIdx.y, threadIdx.z,
13             blockIdx.x, blockIdx.y, blockIdx.z);
14  }
```

As we can see, the function can now display information about the blocks and the grid. Then we can see how CUDA organizes threads and blocks when the program runs. These changes make it easier to understand how parallel programming is set up on the GPU.

6. Create a simple kernel and use the pre-defined variables to define the limits of the game's world segment processed by the kernel;

We upgraded the struct in the world.h to add the grid pointer:

```
22  // Game of life world data structure and type definition
23  typedef struct
24  {
25      size_t generation;
26      size_t size;
27      cell_t data[];
28      char *grid;
```

Then we created the function evolve_kernel :

```
17  __global__ void evolve_kernel(world_t *world_in, world_t *world_out, int W) {
18      // CCalculate index
19      int x = blockIdx.x * blockDim.x + threadIdx.x;
20      int y = blockIdx.y * blockDim.y + threadIdx.y;
21
22      // Verify the the index are in the world
23      if (x < W && y < W) {
24          // Count neighbours
25          int neighbors = count_alive_neighbours(world_in, x, y);
26
27          // Calculate the new state
28          unsigned char current_state = get_cell(world_in, x, y);
29          unsigned char new_state = (current_state == 1 && (neighbors == 2 || neighbors == 3)) ||
30                                    (current_state == 0 && neighbors == 3) ? 1 : 0;
31
32          // set cell
33          set_cell(world_out, x, y, new_state);
34      }
35  }
```

We allocate memory to the GPU then we copied the data from CPU to GPU. We also set the grid on the worlds.

```
// Copy worlds on the GPU
world_t *d_world_a, *d_world_b;
cudaMalloc(&d_world_a, sizeof(world_t));
cudaMalloc(&d_world_b, sizeof(world_t));

cudaMemcpy(d_world_a, world_a, sizeof(world_t), cudaMemcpyHostToDevice);
cudaMemcpy(d_world_b, world_b, sizeof(world_t), cudaMemcpyHostToDevice);

// Give memory to the grid
cudaMalloc(&world_a, sizeof(world_t) + (W * W * sizeof(cell_t)));
cudaMalloc(&world_b, sizeof(world_t) + (W * W * sizeof(cell_t)));
cudaMemcpy(d_world_a->grid, world_a->grid, sizeof(unsigned char) * W * W, cudaMemcpyHostToDevice);
cudaMemcpy(d_world_b->grid, world_b->grid, sizeof(unsigned char) * W * W, cudaMemcpyHostToDevice);
```

7. Run the game with different world sizes, grid sizes and block sizes, measuring the performance similarly to Laboratory 1;

The code shows me the following error :

```
world.h(27): error: incomplete type is not allowed
```

4. Conclusion

In this exercise about parallel programming, we explored fundamental concepts like measuring code performance, parallelizing tasks, and using threads on GPU with CUDA. We started by analyzing the speed of different functions. We also saw the impact of memory. This helped us understand performance and advantages of using parallel programming to manage a lot of data.

We then applied these principles to the Game of Life, dividing the map into zones and assigning tasks to threads. This exercise demonstrated the power of CUDA in breaking down tasks across threads and blocks, allowing for faster computations and greater scalability.

We also learned how CUDA organizes threads and blocks. This involved modifying data structures, allocating memory on the GPU, transferring data between CPU and GPU, and writing kernels to evolve the map.

In conclusion, parallel programming can be very useful for big datasets. This helps dividing tasks into threads and only calculate a part of the dataset, instead of calculating everything.