Multi-core programming homework 1

César Leblic

Langage version and libraries used

C++11

I used C++11 to write the program. This choice was motivated by mutliple reasons : - Standard library include a multi-platform thread library - Lambda functions - I'll explain in more detail in a further part, but I use a small library of my creation which require usage of lambda functions.

FreeGlut

I used Freeglut instead of Glut : - Glut is not maintained since a lot of time - Glut do not permit to exit the game loop properly. In fact you cannot dealocate memory or destroy objects at the end of the program, it just autorize you to `exit()` the program. Usage of threads need to `join` them before exiting the main thread.

Imgui

I used Imgui (https://github.com/ocornut/imgui) which permit you to display easily a GUI.

TMQ

I used TMQ (Threaded Message Queue : https://github.com/cesarl/ThreadsMessagePassing). That's a small helper that I wrote one month ago to implement an execution pipeline in a personnal project (https://github.com/Another-Game-Engine/AGE).

TMQ was designed to handle commands of any type between threads. I tried to design it so it limit synchronisations and cache misses. We'll see further in this document that using TMQ was not the simpler option but that it does not impact performance at all.

Architecture

Class descrition

Architecture is relatively simple, it count a small number of class : - `Display` which is nothing more than a array of `char`. Pixel colors are writed in it and data are send to the GPU with `glDrawPixels` for the rendering.

- `GridBuffer` is one of the more important class of the program. It contain two buffers of `enum Celltype` (each `CellType` have the same size that an `unsigned char`), one for the reading, one for the writing. So that threads, when computing new state will read in the `read` buffer and save the solution in the `write` buffer. Then for the next step, buffer will be swapped so that `read` buffer will be used as `write` buffer and vice versa. In addition `GridBuffer` class contain all methods and rules which permit to compute the new state. Threads will call them at execution time.

- `ThreadQueue` are the combination between a thread and a `TMQ::Queue`. At it's creation a thread is launched and will wait for task to be pushed in the queue. I'll explain in more details how does TMQ works further.
- `WorkerThread` inherit from `ThreadQueue` and implement callback function when a task is received in the queue.

Workflow explaination

Each thread will treat a part of the grid (to limit memory fragmentation and cache missed, grid of cells is a one dimensionnal array and not a double). So, for example, if you create 4 `WorkerThread`, work will be divided by 4.

Threads are created at the beginning of the program, or when the number of threads is modified at run time, appart from this case they are not destroyed and wait for tasks during all the program lifetime.

TMQ, as explained before was designed to implement an execution pipeline in a game engine and reduce cache missed and synchronisation time. A TMQ queue can handle any type of message (`struct` with any number of attribute) and store them in a memory buffer. There is 2 buffers (double buffering) per queue so that when a thread write to a queue and another read from it there is no need of a mutex. Mutex are used only at synchonisation, when read buffer is swapped with the write buffer. After a buffer swap, the worker thread is notified with a condition variable that their is work to do, so it begin to pop commands and apply them, during this time, other threads can continue to push commands for the next step. In addition to that, TMQ allow you to pass command to a thread and recover a `std::future` in exchange, so that when the thread'll have finish his work the command thread can recover the return value of the command.

Here, in our program, we make a very simple usage of TMQ, in fact the usage we make of it is more comparable to a simple threadsafe queue that a double buffered command queue. At each frame, the main thread push a command `Execute` to each `WorkerThread` giving them the range of cell they have to treat, and in exchange recover a `std::future` which will be the counting of cells type in the giving range. After that, the main thread iterate throught the futures and add up result to have the total counting of cell type. We will see in the next section why that type of practice is not the better one and in which way it permit us to avoid bigger performance issues.

Difficulties

I encounter some difficulties doing this project :

- The first one is inherent to the subject himself. I didn't find a way to avoid cache miss when checking neighbours types. In fact, checking for neighbours in a one dimensional or 2d array imply necessary, I think, jump in memory and so, cache miss. That's why I think performances are not mutliplied by the number of threads, because adding thread divide the compute work, but I think multiply in an other hand cache missed in shared cache. See becnhmark result below (on an Intel I7 with 8 thread capacity)

    - 1 Worker : 21ms
    - 2 Workers : 13ms
    - 3 Workers : 10ms
    - 4 Workers : 9.2ms
    - 5 Workers : 8.8ms
    - 6 Workers : 8.3ms
    - 7 Workers : 7.9ms
- The second problem I had to face was to count cells. In fact, if each thread count cell type in their section, we have after to add all this chunks to display the total. My first approach was to use atomic types (`std::atomic_unsigned_int`). But after performance analysis, I realized that my program loose 20% of performance incrementing atomic ints. So I decide to use a new approach. Each worker have a local cell type counter. When a task is push to them, they return to the main thread a `std::future` and store a `std::promise` to set the counter value at the end. Then, when the main thread has pushed all the tasks to workers, it iterate throught futures and add their result to compute the final counting. That approach permit to my program to gain in performance, but it insert a new problem :
- By using future and promise, I can get the worker counting result, but I'm not able to know which thread has finish his work before another. However, when I `get()` the value of the future, if the thread don't have finish to compute it, the main thread will wait for the worker to finish the job. That's why if I get the value to soon I can loose some millisecond (about 2 to 4 millisecond on an I7 in Release built) in the main thread.

Conclusion

The bottleneck part of this program is in the algorythme to compute number of neighbours and not directly in the thread architecture.

I think that it'll be difficult to improve performance with threads.

I admit that TMQ was not design at the origin for that type of application, but using it permit me to test it in more depth and to fix bug, like deadlock.