Final GPUC Assignment: A* GPGPU Implementations for Path-Finding

Florian Klemme

Draft – July 1, 2017

1 Introduction

*A** is a popular search algorithm when it comes to finding the shortest path between to vertices in a graph. I will implement two different approaches of bringing the A* search algorithm to the GPU. The first approach is rather straight forward by using the parallel capabilities of the GPU to compute paths for multiple agents at the same time [1]. In contrast, the second approach focuses on parallelizing the algorithm itself to provide maximal hardware occupancy for a single agent. [3]

2 Implementation

Although both algorithms are inspired by the classic A*, their approach for parallelization is completely different. The algorithm described by [1] (hereinafter referred to as *mA**) "exploits the parallelism in performing the navigation of thousands of agents" at the same time. The implementation should end up being quite comparable to the CPU version. The memory layout is adapted to enable coalesced memory access. A *adjacency directory* is provided which allows the look-up of edges with node indices. As far as I understood this structure, depending on the graph, it might be required to have multiple copies of edges in the edge list for the look-up to work. According to the authors, the implementation of the priority queue has the biggest impact on how good this algorithm performs.

The algorithm described by [3] (*GA**, as the authors call it) tries to achieve maximum parallelization for a single agent by expanding multiple nodes at a time. For this to be fast an conflict-free, GA* uses multiple priority queues to keep track of open nodes. New open nodes are filtered for duplicates before they are distributed back into the priority queue. Pseudocode is listed in the publication that gives a good impression on how this algorithm could be implemented. Although it might be possible to compute everything in one kernel, I think it's reasonable to split the code into one kernel for each `for ... in parallel do`. For section 3.2, I named those kernel by the labels from figure 1. The downside of this approach is that for some information to be persistent it has to be written to global memory that – information that otherwise could have been held in local memory. I think this only affects the priority queues and it might be able to read/write them to global memory in a coalesced manner.

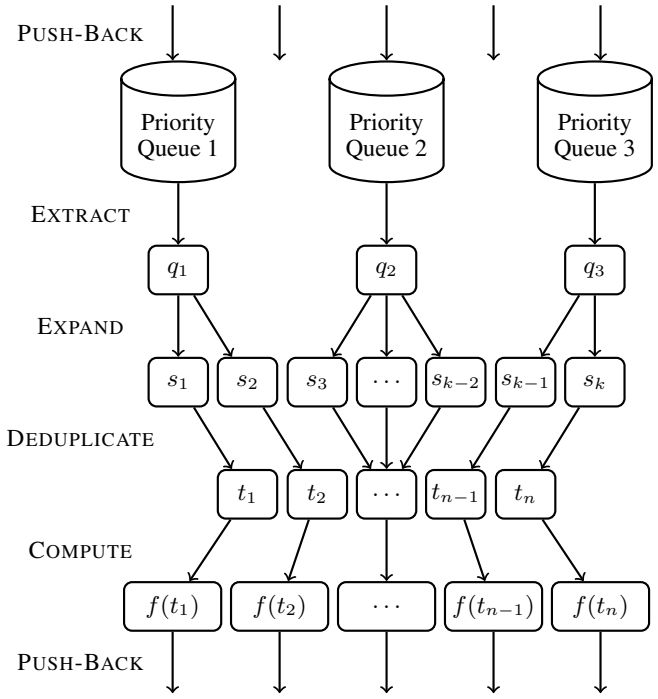

Figure 1: Data flow in GA*. Multiple priority queues allow the conflict-free parallel extraction and expansion of multiple nodes. Syncronization is needed when it comes to cultivating the set of *closed nodes*. (Image from [3])

For finding closed nodes, [3] discuses different hash

algorithms of which the *Parallel Hashing with Replacement* algorithm is chosen for GA*. The pseudocode for this algorithm is not included in the article but in the separately published appendix [2].

3 Evaluation and Results

Both algorithms (which means, the implementation by their authors) have already been compared to common A* CPU implementations so there are some expectations. First of all, A* on a CPU already performs quite well, so the test cases have to load the GPUs to their limits in order to shine. Both algorithms are bounded by their memory consumption so the work size will set the limit. The big difference is that mA* is supposed to perform well for a huge amount of agents (and consequently, a relative small graph) while GA* is supposed to perform well for huge graphs. Both algorithms were able to shine in their use case and I hope that I can recreate that performance in my implementation. It might be hard to compare these algorithms to one another as their strengths are so different.

In their original work, both algorithms have been evaluated using grid-like two-dimensional graphs. In [1], those graphs have been created automatically and paths have been chosen randomly. For [3] the origin of test cases is not specified but by the names ("zigzag", "random" and "empty") and a size of 10000×10000 automatic generation is to be expected. I'll go for an automatic generation of test cases as well, as it has some advantages: Both algorithms need different test cases to show their capabilities. Specific test cases could not be reused for the other algorithm. Also for GA* test graphs are really big which makes it hard to find one or create one by hand. Making them two-dimensional also brings some some benefits: While developing it is easy to visualize and debug the behavior of the algorithms. At the end, nice visualizations might even be usable for the project's presentation.

3.1 Overall points

2 points initial concept

14 points implementation in OpenCL

2 points short presentation

2 points evaluation and discussion (Q & A)

3.2 Implementation points

Priority queue Implement priority queue (binary heap). Required for both search algorithms.

- Implement *push* function (1 point)
- Implement *pop* function (1 point)

Multi-agent A* as proposed by [1].

- Set up data structures to copy required data to/from the GPU. (1 point)
- Implement A* kernel. One thread per agent. (2 points)

Parallel GA* as proposed by [3].

- Set up data structures to copy required data to/from the GPU. (1 point)
- Implement *Extract & Expand* kernel. (2 points)
- Implement kernel that checks if search is finished. (1 point)
- Implement *Duplicate detection* kernel, parallel hashing with replacement [2]. (3 points)
- Implement *Compute & Push-back* kernel. (2 points)

References

[1] Andre Silva, Fernando Rocha, Artur Santos, Geber Ramalho, and Veronica Teichrieb. Gpu pathfinding optimization. In *Games and Digital Entertainment (SBGAMES), 2011 Brazilian Symposium on*, pages 158–163. IEEE, 2011.

[2] Yichao Zhou and Jianyang Zeng. Massively parallel a* search on a gpu: Appendix, 2015. URL `http://iiis.tsinghua.edu.cn/%7Ecompbio/papers/aaai2015apx.pdf`.

[3] Yichao Zhou and Jianyang Zeng. Massively parallel a* search on a gpu. In *AAAI*, pages 1248–1255, 2015.