Hybrid Parallel Streamline Extraction Combining MPI and OpenCL

Michael Vetter*
Universität Hamburg

Stephan Olbrich†
Universität Hamburg

ABSTRACT

Recently scientific simulation application take advantage of modern accelerator technology more and more. For in-situ visualization techniques especially in this case scalability will become an issue. In this work we present a scalability evaluation for a hybrid parallelized streamline extraction algorithm.

Index Terms: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—Single-instruction-stream, multiple-data-stream processors (SIMD); C.1.4 [Processor Architectures]: Parallel Architectures—Distributed architectures; G.1.o [Numerical Analysis]: General—Parallel algorithms

1 INTRODUCTION

In typical postprocessing scenarios the simulated raw data are stored by the simulation running in batch mode and processed for visualization in a seperate task. This visualization can be done either interactively or in a batch mode, too. With increasing data volume this approach leads to a storage bottleneck in both tasks. One option to evade this bottleneck is to avoid storing the raw data and do the visualization in-situ with the simulation. Common in-situ approaches fulfill the full visualization task on the supercomputer running the simulation and store just the 2D pixel data within a movie. In contrast, in the DSVR Framework [3, 1] a distributed visualization approach has been implemented. In contrast to the simulations' raw data, which typically scales in volume with $O(n^3)$ where n is the grid size, data volume for storing 3d geometries scales with $O(g)$ where g is the amount of 3d geometric primitives.

The extraction of streamlines, streaklines, or pathlines is a widely used method for geometric flow field visualization. Algorithms typically used for the underlying particle tracing are Euler integration or Runge Kutta integration for example. An overview is given at [2].

2 PARALLEL ALGORITHMS

A critical challange for most in-situ visualization techniques is the load balancing. The simulations' large datasets are usually spread in domain decomposition over several compute nodes regarding to the simulations' best load balancing in massively parallel computing. In addition to the distributed memory parallelization with MPI for example, modern accelerator technology get common, demanding for hybrid parallelization schemes. For the visualization the domain decomposition should be used as given by the simulation, since data rearrangement is mostly a big communication overhead. The second issue with scalability for geometric flow field visualization is given by the typically low complexity of the algorithms, which can not benefit from more than a few cores. For example, the complexity of the presented algorithm is $O(n)$ where n is the number of seed points multiplied by the number of supporting points, which is limited to avoid visual cluttering. So for this kind of algorithms good scalability will mean no, or at least less slowing down at higher core counts. Speedup can be attained in conjunction with the simulation by reducing the simulations' I/O. We have recently done a comparison of parallel algorithms for in-situ pathline extraction [5]. Analog work for streamline extraction was presented in [4].

In this work we like to present the evaluation of our approach for in-situ streamline extraction utilizing OpenCL-kernels spread over multiple MPI nodes. This algorithm uses 4th order Runge Kutta integration. We asume a simulation using domain decomposition and will locate the streamlines on the MPI node which helds the needed raw data. So the lines may change over the processes by MPI communication. While within Euler integration a line may only pass over domain boundaries of processes after each integration step, with 4th order Runge Kutta the lines may also pass over during one integration step. This has to be handled by the algorithm. So for streamline extraction we used a modified version (see algorithm 1) of one of our pathline extraction algorithms described in [5]. This algorithm was originally executed in parallel on all MPI processes. For supporting OpenCL enabled devices the second level loop was complemented by a kernel with one work item for each streamline.

Algorithm 1 streamline algorithm

```
while in global scope some lines not finished do
    while in local scope some lines not finished do
        work on next line
        while line is not finished and stays in local scope do
            integrate line's next supporting point
        end while
    end while
    exchange not finished lines including runge kutta information
    with neighbour domains
end while
```

3 EVALUATION

We have implemented our approach in C and evaluated it on two different flow fields. Goal of this evaluation was getting information about scalability and performance of the presented algorithm.

For evaluation we used a small compute cluster with 8 nodes. Each node (HP SL 390) provides 12 cores (2 * Intel Xeon X5650), 48 GB of memory and 3 Nvidia Tesla M2070Q GPGPUs. The nodes are connected via QDR-Infiniband allowing a maximum bandwith of 32 Gbit/s. The cluster operates with SLES 11. We used the gcc comiler suite (in version 4.3.4), mvaphich MPI environment (in version 1.8) and CUDA runtime environment (in version 4.2).

Scalability

For testing the overall scalability we used a well-known routine by Crawfis[1] producing a tornado-like swirl on a 3D rectilinear grid. This was used since it represents an real world scenario. The tornado was calculated on a grid consisting of 400^3 grid points wich would not exceed the memory limitations of one Tesla card. As the algorithm does not scale with data set size but with amount of seeded lines and supporting points as (see sec. 2), the results can be easily adopted to larger data sets.

*e-mail: michael.vetter@rrz.uni-hamburg.de
†e-mail: stephan.olbrich@rrz.uni-hamburg.de

[1] http://www.cse.ohio-state.edu/~crawfis/Data/Tornado/tornadoSrc.c

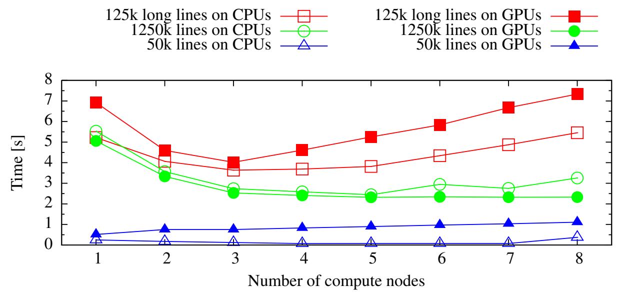

Figure 1: Overall runtimes for the extraction of streamlines on a dataset by Crawfis.

For this scalability test we used three configurations, seeding $125 * 10^4$ lines with 10^2 supporting points, $125 * 10^3$ lines with 10^3 supporting points, and $5 * 10^4$ lines with 10^3 supporting points randomly over the volume. The first two cases nearly need the same amount of flops is for round about $125 * 10^6$ integration steps.

First we like to note that the algorithm performs on one GPGPU up to 2.5 times faster than on one CPU core. For further discussion we have always utilized full compute nodes using either 3 GPGPUs or 12 CPU cores. Figure 1 shows the overall scaling of the algorithm, which is not that bad. It achieves maximum performance utilizing three nodes, but running on more nodes will not hurt the overall performance in most cases. Just calculating fewer, longer lines on the GPGPU will lead to a significant performance breakdown when using more than 4 nodes. This is caused by two facts:

1. As the lines get longer, they tend to leave the local dataset more often, leading to more MPI communication.
2. Since each streamline represents one work item the ratio between work items and data volume get worse.

On the other hand calculating just 50000 lines with 100 supporting points each, the performance using GPGPUs ist also acceptable even though it is not as good as on CPUs.

Kernel execution times

Since we have seen in the scalability tests that the OpenCL enabled version of our algorithm may perform bad under some circumstances we tried to get more details on this. Beside the algorithm's overall speedup we liked to get further information about the kernel behaviour. As GPGPUs are massively parallel SIMD architectures they are known to have performance issues with branching on the one hand and not fully utilized cores on the other hand. Therefore we used an artificial flow field combining a dipole with a static flow for calculating streamlines with 300 supporting points each. This measurement does not take into account any data transfers, execution times for distributed kernels are summed. First we measured the execution time of a single launched OpenCL kernel in comparison to one CPU core evaluating the influence of the workload by seeding 256, 512, and 1024 lines. Figure 2 shows, that the performance of the OpenCL kernel is hurt using smaller workloads. The OpenCL kernel could finish this task up to 6 times faster than a single CPU. For the same task the overall performance of the algorithm's GPGPU version over the CPU version was 3 times faster.

In our algorithm branching occurs every time a streamline could not be calculated completely on one MPI node. So using a disadvantageous domain decomposition will not only lead to an increased MPI communication overhead but also to decreased OpenCL kernel performance. Figure 3 shows the performance of the pure OpenCL kernel and the equivalent CPU function for different domain decompositions. The CPUs performace is not measurably impaired by branching. On the other hand the OpenCL

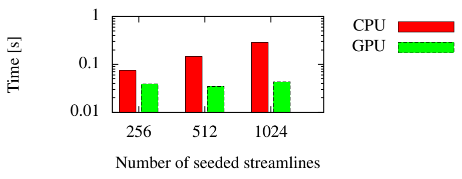

Figure 2: Kernel execution time for different workloads (256, 512, and 1024 streamlines with 300 supporting points each) on a single MPI process at logarithmic scale.

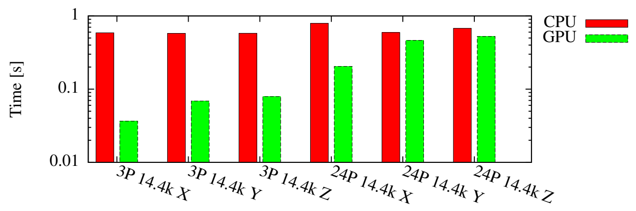

Figure 3: Kernel execution time for 3 and 24 MPI processes and different domain decompositions (partitioning in X, Y, or Z dimension) at logarithmic scale.

kernel is massively affected by the domain decomposition leading to a performance degradation of a factor of two.

4 Conclusion

In this work we have shown that acceptable scalability as specified in sec. 2 can be reached using GPGPUs for streamline extraction even though the algorithm may perform better on CPUs. Scalability of the OpenCL kernel mainly depends on the amount of seeded lines. In future work we like to evaluate a presorting of lines to avoid branching as well as combined utilization of one GPGPU by more than one process. We also like to compare costs and energy efficiency of simulation and visualization combined.

Acknowledgements

We like to thank Viktor Borbus for supporting our work with his bachelor thesis.

References

[1] N. Jensen, S. Olbrich, H. Pralle, and S. Raasch. An efficient system for collaboration in tele-immersive environments. In S. N. Spencer, editor, *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization (EGPGV-02)*, pages 123–132, New York, Sept. 9–10 2002. ACM Press.

[2] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen. Over two decades of integration-based, geometric flow visualization. *Computer Graphics Forum*, 29(6):1807–1829, 2010.

[3] S. Olbrich, H. Pralle, and S. Raasch. Using streaming and parallelization techniques for 3D visualization in a high-performance computing and networking environment. In L. O. Hertzberger, A. G. Hoekstra, and R. Williams, editors, *High-Performance Computing and Networking*, volume 2110 of *Lecture Notes in Computer Science*, pages 231–240. Springer, 2001.

[4] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 16:1–16:12, New York, NY, USA, 2009. ACM.

[5] M. Vetter and S. Olbrich. Scalability issues of in-situ visualization in parallel simulation of unsteady flows. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 177–190. Springer Berlin Heidelberg, 2012.