Integrated Project

Profiling + CUDA

A finite volume case study from an industrial application (Extended Abstract)

Miguel Palhas — pg19808@alunos.uminho.pt
Pedro Costa — pg19830@alunos.uminho.pt

Braga, January 2012

1 Introduction

This document analyzes an application which computes the evolution of a pollutant in a given environment, studying possibilities of optimization and/or parallelization.

The application calculates the pollutant spread area at variable moments during a given time interval. This pollutant spreads according to a partial differential equation which is solved using the Finite Volume Method.

The algorithm used by the program loops until the specified time interval is reached. At each iteration of this main loop, the flux of pollution is calculated in each edge (performed by a function named `compute_flux`) and the mesh pollution levels are updated (performed by the `update` function).

Two attempts of parallelization are intended: one using shared memory on a multiprocessor system and one using a GPU.

The final aim is to have a qualitative and quantitive comparison of both versions against each other and the original sequential code, and an evaluation of possible bottlenecks and/or optimizable code parts.

The initial profiling of the application, including the methodology and conclusions, is presented in section 2. In section 3, the initial profiling information is used to identify the parts of the algorithm/code which are better suited for optimization and/or parallelization. The implementation of a shared memory parallel version and a GPU version are explained in section 4 and section 5, respectively.

2 Sequential

The analysis of the sequential code starts with the generation of it's callgraph, from which it is possible to find the function where the program execution spends most of it's time. According to Amdahl's Law, this function should be the first to study for optimizations and/or parallelization, as it's where one can profit the most from speedup.

For the purpose of generating the call graph, the `Callgrind` tool[1] was used. The information generated by this tool could then be visualized using `KCachegrind`.

Originally this reported 11% of the execution time in the `compute_flux` function and 8% in the `update`, while the remaining[2] 80% of the time was spent with I/O (to build an animation). Since the goal is optimize the computation, the I/O calls were deactivated and the profiling tool reported 63% of the time in the `compute_flux` and the remaining time in the `update`.

[1] From the Valgrind suite.

[2] A very small percentage of the time is spent on other functions, which can therefore be ignored

3 Optimization & Parallelization

As determined in section 2, the best targets for optimization and/or parallelization are the `compute_flux` and `update` functions, which will be the primary and secondary goals for this project, respectively (each parallelized version will be focusing on these functions by this order).

The first optimization possible in the `compute_flux` function is the removal of the `Parameter.getDouble()` call. This was meant to retrieve the Dirichlet condition value from the parameter file, but being this a constant throughout the program execution, the value can be loaded in an early preparation stage and sent as an argument to `compute_flux`. Additionally, moving this argument to a global variable can reduce the function call overhead.

Excessive dereferencing may be a common problem on both functions and is directly related to the FVLib library. Although the usage of pointers may greatly improve code readability, the increased memory accesses caused by deep levels of dereferencing tend to aggravate effects of any memory bottlenecks. With some adjustments to the library internal structures, the same or better readability might be achievable with fewer pointers and lower reference depth.

Lastly, to allow the parallelization in both intended approaches, the loops themselves have to be changed both in `compute_flux` and `update`. The original code uses a non-standard iterator-like approach to loop through the elements of the mesh. While this works in sequential code, it won't work with conventional parallelization mechanisms, as there would be no knowledge of the iterations other than the data itself (which is not enough). This mechanisms requires index based accesses, or at least random-access iterators[3]. Since the FVLib library implements index based accesses, the changes in the loops are trivial.

3.1 Dependencies

In the `compute_flux` function a data dependency can be found which prevents parallelization. This dependency is found in the calculation of Δt_i (the final value Δt_n is returned by the function) and causes each iteration to depend on the previous one (the first iteration uses a preset value Δt_0).

Using mathematical operations this calculation can be removed from the loop and the dependency substituted by a

[3] OpenMP works with both, CUDA works naturally with indexes.

calculation of the maximum computed velocity (v_{max}), which can be computed through a reduction.

The method of calculation through reduction is the best option to parallelize the calculation of iteration dependent associative binary operators ($\sum$,$\prod$,min,max). The goal of this method is to reduce the problem size as the computation advances: each thread picks two values, and in the next step only half the threads are needed, until only one is left with the final value

The `update` function also holds a problem when thinking of parallelization. This function iterates through the edges of the mesh, and updates the values associated with the cells adjacent to those edges. Since each cell has multiple edges and the final value associated with a cell is the sum of the contributions from all it's edges, then multiple iterations might be changing this value simultaneosly.

It's possible to remove this dependency by changing the loop itself to iterate over each cell calculating the final value from all it's edges, instead of iterating over the edges. Considering that no edge data is changed, this completely removes de dependency and allows the parallelization of the function.

4 OpenMP

4.1 Implementation

The main feature about OpenMP that makes it so interesting to create shared memory parallel code is that the program itself is almost equal to the sequential version, with the addition of the required directives and adaptation tweaks (mainly to solve concurrency problems).

Because there are no nested parallelizable loops in this code, the number of threads to be issued are set by default to the maximum number of threads supported by the hardware to be executing at any given time.

4.1.1 Flux Computation

To parallelize this function, a `parallel for` OpenMP directive is placed before the inner loop. Considering all the variables external to the parallel zone as shared, all the required variables must be explicitly made private (either with the `private(list)` clause or by declaring the variables only inside the parallel zone).

5 CUDA

5.1 Data Structures

For a GPU implementation, all the data structures had to be converted to a suitable format (using sequential memory positions instead of dereferencing). Since this was required from the start, it was also a chance to optimize memory for GPU computing, by using a structure of arrays instead of an array of structures, to take advantage of the GPU's coallesced memory accesses.

5.2 Implementation

5.2.1 Flux Computation

A total of 3 CUDA kernels were written for the implementation. The first one was to replace the `compute_flux` function. This is a very straight-forward implementation, with no major changes to the original algorithm, aside from the fact that each CUDA thread handles a single element, instead of looping through the mesh.

5.2.2 Mesh Update

The `update` function was also converted to CUDA. This was not as interesting to parallelize in the OpenMP version, since it doesn't occupy the majority of the time. However, using the original CPU implementation would mean that for every iteration of the main loop, flux and polution data had to be copied back from the GPU to execute the update, and then copied again to GPU for the next iteration. This would eliminate any advantage of GPU parallelization, so this function had to be re-written, taking into consideration the data races of the original one.

6 Conclusion

In this document the original application code was analyzed and studied. The two key functions in the program execution were identified using the program's callgraph. Also, a major bottleneck was identified in the animation generation, which invalidated any computation optimization effort when activated.

Several problems were easily detected in early stages, along with parallelization obstacles. These problems were mostly related with the program's memory usage, and some even prevented parallelization. The identified obstacles, namely data dependences, were solved by manipulating the original code, only possible due to an increasingly better understanding of the algorithm.

Two parallel versions of the application were implemented. The first version implemented the same code using the OpenMP directives to parallelize the most critical part of the program. The second version adapted the original code to perform all the required computations in a GPU using CUDA.

All versions still lack a formal profiling which would allow to identify bottlenecks and compare efficiency. Despite that, both of the implemented parallel versions showed a small but consistent speedup in multiple executions. While this shows that the application can be improved with parallelization, the differences classify the algorithm as non-scalable.

Further optimization efforts can be done to improve the obtained results. For the OpenMP implementation, these include extending the parallelization to the `update` function and changing the data structures. For the CUDA implementation, a revision of memory access efficency might be useful, as well as the usage of CUDA Streams to minimize idle time of the GPU. Also, for both implementations, changing how the FVLib library works, specifically how the data is accessed, could greatly improve the program efficiency specially with the animation generation.

References

[OMP] *OpenMP Application Program Interface*
OpenMP Architecture Review Board
Version 3.1, July 2011

[NVIDIA]
`http://developer.nvidia.com/category/zone/cuda-zone`