D036

# Accelerating Reservoir Simulation with GPUs

K.P. Esler* (Stone Ridge Technology), S. Atan (Marathon Oil Corp.), B. Ramirez (Marathon Oil Corp.) & V. Natoli (Stone Ridge Technology)

## SUMMARY

Over the past several decades, reservoir simulation has progressed in capability from coarse two-dimensional models to inhomogeneous and detailed three-dimensional models, providing a high degree of fidelity to empirical production rates. Part of this increase can be attributed to advances in solution algorithms and part to advances in computational hardware. In this presentation, we discuss examples of both types. In an IMPES formulation, computational effort is divided into an implicit solve of the discretized pressure equation and explicit updates of the saturation equations. By porting and optimizing saturation updates to run on multiple GPUs, we accelerate their execution speed by a factor of about 400. By replacing the multithreaded direct sparse solver for the pressure equation by an iterative solver preconditioned with algebraic multigrid, we reduce the solution time by a factor of 20. Combining these two enhancements, we improve the overall execution speed of Marathon Oil Corporation Multiscale Simulator (MMS) by a factor of over 100 for the SPE10 Model 2 benchmark problem.

## Introduction

Over the past several decades, reservoir simulation has progressed in capability from coarse two-dimensional models to inhomogeneous and detailed three-dimensional models, providing a high degree of fidelity to empirical production rates. Part of this increase can be attributed to advances in solution algorithms and part to advances in computational hardware. In this presentation, we discuss examples of both types. By porting and optimizing saturation updates to run on multiple GPUs, and by utilizing algebraic multigrid for the implicit pressure solve, we improve the overall execution speed of Marathon Oil Corporation Multiscale Simulator (MMS) by a factor of over 100.

The MMS code employs an implicit pressure, explicit saturation (IMPES) formulation of the black-oil model. It has several novel features, including adaptive saturation time stepping and a multiscale formulation of the pressure equation. For complex, highly inhomogeneous models such as SPE10 (M.A. Christie and M.J. Blunt (2001)), very small explicit saturation time steps are required to retain accuracy and stability. Adaptive time stepping improves this significantly by allowing larger time steps to be taken when possible, however thousands of saturation steps are still typically required between each pressure solve. Thus, the saturation part of the problem initially dominates the overall run time.

Taking advantage of the opportunities for parallelism, we reformulate the problem to work efficiently on multiple GPUs using single precision arithmetic and thereby increase the speed of the saturation update by a factor of over 400. With the accelerated saturation, the implicit pressure solve becomes the dominant computation bottleneck. Then, switching from direct sparse solvers and traditional iterative solvers to a GMRES solver preconditioned by algebraic multigrid, we then accelerate the pressure solve by a factor of nearly 20. By combining these two enhancements, we reduce the execution time for a benchmark problem from nearly three days to just over half an hour.

## Method

### Executing saturation updates on GPUs

In the IMPES formulation of reservoir simulation, the pressure equation is first solved implicitly, the result of which is used to determine the aggregate flow rates. The flow is then used to update the cell saturations over many explicit time steps. During each time step, a maximum step size is selected with an analog of the CFL stability criterion by considering the amount of time it will take to either completely saturate or deplete a cell. The minimum time is then taken over all the cells to determine the adaptive time step, allowing large steps to be taken when possible. Since the saturation steps dominate the total execution time, and each step has a high degree of inherent parallelism, this phase of the computation is a natural candidate for acceleration with graphical processing units (GPUs).

At present, a GPU can be described as a multicore vector processor with numerous floating point units (FPUs), a wide, high-throughput memory bus, and a memory hierarchy with relaxed coherency rules. It uses multithreading at a massive scale to effectively hide memory latency and achieve high FPU utilization. For efficient execution on GPUs, a code must expose thousands of loosely-coupled tasks which can be performed semi-independently. Since the saturations at each mesh point are updated independently for a given step, this can be readily achieved. The saturation update has relatively few arithmetic operations per data load/store, so performance is limited by memory bandwidth. Thus, it is imperative to make memory access as efficient as possible by laying out the data appropriately.

A central decision in this layout is how to best aggregate the data. For data stored on a mesh, with several values per point, it is typical to choose to store the data either as an array of structures (AoS), or as a structure of arrays (SoA). If all elements of the structure for a given mesh point are used simultaneously, the AoS format may be advantageous. If only a subset are accessed at a time, however, the AoS storage format will be inefficient, since all the data members will be implicitly loaded from DRAM a cache line at a time, and unused members of the structure will waste bandwidth. In contrast, the SoA format allows individual data members to be contiguous in memory. Such contiguous access is particularly important for good performance on GPUs. For this reason, each physical quantity (e.g. saturation, mobility, etc.) is stored in a separate array in GPU memory.

Since the PCI-E bus used for transferring data between CPU and GPU memory is relatively slow, effort was made to minimize this data transfer. In the main simulation loop, pressures are first computed implicitly and used to compute flow rates. These flow rates are transferred to the connections in GPU memory. After this data transfer, a sequence of GPU kernels is called to perform a saturation step. First, in kernel `compute_connection_flux_kernel` the flow rate for each connection is determined by weighting the total flow rate by the relative mobility for each phase and adding on the gravitational contribution. These rates are stored back to global GPU memory.

Next, the saturation update itself is executed in three kernels. In `update_saturation_kernel1`, the total flux into each cell is computed by summing over its connections. The maximum allowable time step for each cell is also computed from the CFL criterion. These computations are done in parallel on the GPU in blocks of typically 64 cells. Since these blocks cannot efficiently intercommunicate, each block determines the minimum of the CFL time steps and writes its result to GPU memory. A second kernel, `min_dt_kernel` then determines the minimum over all the block results, writing the minimum time step back to GPU memory. In `update_saturation_kernel2`, the saturations are updated with a simple Euler integration step, along with saturation-dependent quantities. Finally, upstream weighting is performed in kernel `copy_upstream_kernel`.

For models with a large number of cells, it is possible to further accelerate the saturation update by taking advantage of multiple GPUs, which requires careful design. In parallelizing over GPUs, we divide the work by distributing an equal number of cells to each GPU. We use OpenMP to assign one CPU thread to manage each GPU. In the computational mesh, each *connection* has terminals at two nodes. Some of these connections will then span the boundary between two GPUs. In order to properly handle upstream weighting, it is necessary to transfer data between GPUs at each step. At the beginning of the run, we identify which connections span the boundary between GPUs. For each GPU, we construct a list of connections which must be exported to every other GPU. A GPU kernel is used to gather the data from these nodes into a contiguous *export buffer*, whose contents are then copied to CPU memory. The data from each GPU is then copied to a contiguous *import buffer* for each GPU, which is in turn copied to GPU memory. Finally, a second GPU kernel is then called to scatter the data to their appropriate locations in the connection data arrays.

**AMG preconditioning for implicit pressure solves**

The pressure equation employed in IMPES reservoir simulation is an elliptic partial differential equation (PDE). Elliptic PDEs are characterized by the fact that a localized change to the source term results in delocalized changes to the solution. Once the PDE is discretized using, e.g., finite differences, this *long-range* character of the PDE makes the solution of resulting linear equations often very inefficient to solve by simple iterative methods because of their large condition number. Typical methods, such as GMRES or ORTHOMIN, rapidly reduce the high-frequency components of the error in the solution vector, but eliminating the low-frequency errors takes many more iterations.

Multigrid methods were developed to address precisely this difficulty presented by elliptic PDEs. The first multigrid methods, known as *geometric multrigrid*, achieve high convergence rates for problems on structured grids by constructing an analogous system of equations on a coarser mesh. On the coarse mesh, low-frequency components of the solution vector converge more rapidly, and this coarse solution can be interpolated to improve the solution on the original mesh. By applying this procedure recursively, multigrid solvers can be shown to efficiently solve a system with $N$ cells in $\mathscr{O}(N)$ time.

In reservoir simulation, however, unstructured grids are often employed, making the use of geometric

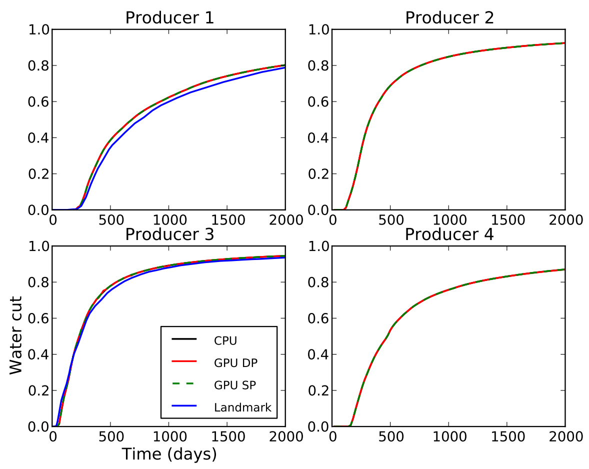

***Figure 1*** *A comparison of the watercuts for the SPE10 Model 2.*

multigrid problematic. Furthermore, the strong spatial inhomogeneity of the transmissibility prevents efficient solution by geometric multigrid even on regular Cartesian meshes. In recent years, however, multigrid concepts have been generalized to allow efficient solution of linear systems without any knowledge of the underlying geometry. This *algebraic multigrid* has been shown to be very effective across a broad range of problems, including reservoir simulation (Klaus Stüben et al. (2007)). When used as a preconditioner to an iterative method, such as GMRES, even greater efficiency is possible.

Several open-source packages with AMG functionality are available. Among the most mature and full-featured is the BoomerAMG component of the *hypre* package from Lawrence Livermore National Laboratory (R.D. Falgout et al. (2006)). We employ the Fortran bindings to this C library, utilizing the BoomerAMG solver as a preconditioner to FlexGMRES. Many distinct AMG algorithms are available in the package, each with adjustable parameters. We attempted to find the combination of methods which yielded a robust solution in the least time. We found that of the methods we sampled, PMIS coarsening (H. De Sterck et al. (2006)) with extended interpolation (H. De Sterck et al. (2007)) and a strength parameter of 0.5 often yielded the fastest solution. These methods are also among the most amenable to parallel setup and execution. For the SPE10 Model 2 we discuss below, the solver usually converged in less than 20 iterations and about 3.25 seconds, in comparison to 63 seconds for the direct, multithreaded solver in Intel's Math Kernel Library (MKL).

## Results

The Tenth SPE Comparative Solution Project has become a standard benchmark for reservoir simulation. As such, we select Model 2 from the project for testing and comparison. All runs were performed on the full fine mesh without upscaling. The CPU runs were performed on a workstation with 2$\times$ quad-core 3.2 GHz Intel Xeon W5580 processors. The machine was also equipped with two NVIDIA Tesla C1060 GPU cards. A similar machine was equipped with two hex-core 2.67 GHz Intel Xeon X5650 processors and four NVIDIA Tesla C2050 GPU cards. 2000 days of production were simulated with a pressure time step of 20 days and a minimum saturation time step of 0.001 days.

Since the last generation of GPUs, including the C1060, had poor double-precision performance relative to single precision, it is useful to determine whether single-precision is sufficient for a given application. Figure 1 shows plots of the watercuts from our SPE10 simulations. At this scale, the results from the original CPU code are indistiguishable from the GPU results, in both single and double precision, demonstrating that single-precision is sufficient for the explicit saturation updates. We note also that the results are in good agreement with the published fine-grid results from the Landmark code.

| | SP total (min.) | DP total (min.) | SP sat. step (ms) | DP sat. step (ms) |
|---|---|---|---|---|
| CPU only | N/A | 3977 | N/A | $\sim$ 1460.0 |
| 1 $\times$ C1060 GPU | 69.12 | 172.38 | 16.35 | 50.40 |
| 2 $\times$ C1060 GPUs | 47.68 | 96.78 | 8.75 | 25.60 |
| 1 $\times$ C2050 GPU | 48.20 | 69.65 | 9.10 | 16.35 |
| 2 $\times$ C2050 GPUs | 37.88 | 47.30 | 4.95 | 9.00 |
| 3 $\times$ C2050 GPUs | 35.80 | 45.32 | 3.85 | 6.80 |
| 4 $\times$ C2050 GPUs | 35.23 | 43.00 | 3.45 | 5.85 |

***Table 1*** *Run times for SPE10 problem with single and double-precision saturation updates on CPU only and on GPU. Both the total simulation time for 2000 days of production and the time for a single saturation update are given.*

Table 1 gives timings for the same simulation running in several hardware configurations. Running only on CPUs, the run requires almost three days to complete. Introducing one C1060 GPU reduces this dramatically to just over an hour in single-precision. Using faster C2050 GPUs reduces this further. As we increase the number of GPUs, each saturation step takes less time, although the required inter-GPU communication prevents perfect linear scaling of speed with GPU count. However, for more than two C2050 GPUs, the gains are marginal since the saturation update no longer dominates the run time.

## Conclusions

After decades of development, the field of reservoir simulation continues to be one of vigorous research with frequent advances. By taking advantage of modern heterogeneous computing platforms and advanced algorithms for linear solvers, we have seen that a dramatic performance enhancement is possible, yielding a run-time reduction of over two orders of magnitude. Nonetheless, room remains for further gains, through both improved algorithms and porting the remaining sections of code to run on GPUs. Advances in both hardware and algorithms will undoubtedly continue to enable faster simulations with higher fidelity.

## Acknowledgements

We thank the Marathon Oil Corporation for supporting this work and for permitting its publication.

## References

H. De Sterck, R.D. Falgout, J.W. Nolting and U.M. Yang [2007] Distance-two interpolation for parallel algebraic multigrid. *Journal of Physics: Conference Series*, **78**(1), 012017.

H. De Sterck, U.M. Yang and J.J. Heys [2006] Reducing Complexity in Parallel Algebraic Multigrid Preconditioners. *SIAM J. Matrix Anal. Appl.*, **27**(4), 1019–1039, ISSN 0895-4798, doi:10.1137/040615729.

Klaus Stüben, Tanja Clees, Hector Klie, Bo Lu and Mary F. Wheeler [2007] Algebraic Multigrid Methods (AMG) for the Efficient Solution of Fully Implicit Formulations in Reservoir Simulation. *SPE Reservoir Simulation Symposium*.

M.A. Christie and M.J. Blunt [2001] Tenth SPE Comparative Solution Project: A Comparison of Upscaling Techniques. *SPE Reservoir Evaluation & Engineering*, **4**(4), 308–317.

R.D. Falgout, J.E. Jones and U.M. Yang [2006] *Numerical Solution of Partial Differential Equations on Parallel Computers*, Springer-Verlag, chap. 8: The Design and Implementation of em hypre, a Library of Parallel High Performance Preconditioners. 267–294.