B37

# GAMPACK (GPU Accelerated Algebraic Multigrid Package)

K.P. Esler* (Stone Ridge Technology), V. Natoli (Stone Ridge Technology) & A. Samardzic (Stone Ridge Technology)

## SUMMARY

In reservoir simulation, the elliptic character of the pressure subsystem and the inhomogeneous permeability field result in extremely slow convergence for conventional iterative solvers. Algebraic multigrid (AMG) methods address this challenge by constructing a multilevel hierarchy of matrices that naturally adapts to the permeability channels of the underlying geology. Preconditioning with AMG allows difficult cases with millions of unknowns to be solved in just a few iterations.

In just a few years, graphical processing units (GPUs) have progressed from a research curiosity to a productivity workhorse by reducing time-to-solution and overall hardware cost. The highly irregular computation patterns of AMG, however, require new approaches to adapt to the many-core paradigm. The construction of the coarse matrix hierarchy and grid transfer operators poses a particular challenge for GPU acceleration.
We show that by carefully selecting algorithms with sufficient fine-grained parallelism, and implementing them with novel approaches, it is possible to substantially accelerate both the setup and solve stages. We present GAMPACK, a library for accelerated AMG, and show that on a single GPU it can typically reduce the total setup and solve time by a factor of over 5, when compared to a widely-used AMG solver running on 8 Xeon cores.

## Introduction

Throughout the long development of reservoir simulation, linear solver technology has played a critical role in its computational performance. Historically, methods developed within the reservoir simulation community, such as the alternating direction implicit (ADI) method (Peaceman and Rachford, Jr. (1955)), have had broad reach outside the field. Other technologies, such as algebraic multigrid (AMG) (Ruge and Stüben (1987)), have been imported to the field to great effect, particularly when combined with extant practices, such as the constrained pressure residual (CPR) method (Wallis (1983); Wallis et al. (1985)). The gradual improvement of these methods have resulted in solvers that are robust, efficient, and reasonably scalable.

Over the past few years, however, the computational paradigms associated with the underlying hardware have been undergoing significant changes. Around 2004, the threshold voltage of CMOS silicon technology ceased to scale as it had in the past, resulting in a plateau of clock speeds due to power and thermal dissipation constraints. Since the "free lunch" of steady clock-speed growth ended, gains in performance have been attained primarily through the increasing use of parallelism. In addition to the relatively straightforward multiplication of traditional CPU cores, new types of processors that focus on computational throughput through massive parallelism have become available. Perhaps the most successful of these are graphics processing units (GPUs), which provide significantly higher floating point performance and memory bandwith at the expense of requiring the programmer to expose an abundance of fine-grained parallelism. If enough parallelism can be exploited for the central computational kernels, a significant acceleration is often possible. As a result, GPUs have been rapidly adopted for many applications in geoscience, most notably in seismic processing. The lack of efficient implementations of the most robust linear solvers, however, has thus far limited the adoption of GPUs in reservoir simulation.

For many simple sparse linear solvers, exposing parallelism has been relatively straightforward. Solvers based on a simple Krylov subspace iteration combined with trival preconditioners, such as Jacobi, are available in many high-performance implementations (EM Photonics (2011); NVIDIA (2010); Bell et al. (2009); Rupp (2012)). However, the local nature of these preconditioners typically results in very slow convergence on the highly ill-conditioned matrices common to reservoir simulation. Additional work based on incomplete factorizations has shown promise (Heuveline et al. (2011)), including some application to reservoir simulation (Sudan et al. (2010); Appleyard and Appleyard (2011); Li and Saad (2010)).

It has been well established over the past decade that algebraic multigrid (AMG) provides a robust preconditioner for the pressure subsystem. Traditional AMG methods, however, require a costly setup phase, typically employing algorithms which are either explicitly sequential in nature or expose only coarse-grained parallelism. The highly irregular computation patterns of AMG require new approaches to adapt to the many-core paradigm of GPUs. In the paper, we discuss the approach we have taken to build the GPU Algebraic Multigrid PACKage (GAMPACK), a preconditioner and solver built on the NVIDIA CUDA platform and designed for use in reservoir simulation. We discuss how AMG can easily be combined with other preconditioners for the non-pressure degrees of freedom in the well-known two-stage CPR scheme. Finally, we give the results of the application of these methods to a variety of sample problems.

## Methodology

Darcy's equations model the flow of fluids through porous media, and, for insoluble fluids, can be given by

$$\frac{\partial}{\partial t}\left(\frac{\phi S_i}{B_i}\right) = \nabla \cdot [\mathbf{T}_i (\nabla p_i - \gamma_i \nabla z)] + \frac{q_i}{B_i}, \tag{1}$$

where $p_i$ and $S_i$ give, respectively, the pressure and saturation for phase $i$. In a black-oil model, three primary variables are typically selected to describe the state of the system in each cell. One pressure and two saturations are a common option, but other choices are possible. The equations governing the pressure degrees of freedom (DOFs), have an elliptic character, while those governing the saturations have a hyperbolic character. Stated in a more intuitive way, the pressure solution at a given point, $\mathbf{r}$ is sensitive to changes in the system at a distant point, $\mathbf{r}'$, and solutions to the pressure equation is spatially smooth. In contrast, the saturation system is "near-sighted", in that the solution at a given point depends strongly only on its immediate neighborhood. Simple preconditioners, such as Jacobi, ILU0, or Gauss-Seidel have a predominantly local effect, which can often be effective for the saturation system. However, the "far-sightedness" of the pressure system, which is characteristic of elliptic PDEs, has proven problematic for them, since they quickly eliminate the components of the error with high spatial frequency, but have little effect on the smooth modes which dominate the solution. This problem is manifested as a rapid increase in the number of iterations required to reach convergence as the system size grows.

*Algebraic multigrid*

Multigrid preconditioners are constructed explicitly to handle the low-frequency modes by mapping the full problem to a coarser mesh, in which these modes appear to have higher frequency. Residuals on the fine mesh are mapped to the coarse mesh by applying a linear *restriction* operator, $\mathbf{R}$. After solving on the coarse mesh, the resulting solution can be interpolated back to original fine-scale mesh with the interpolation operator, or *prolongator*, $\mathbf{P}$, to correct the solution for the low-frequency modes. By applying this coarse-grid correction recursively, a hierarchy of meshes can be constructed, which, when combined, can effectively eliminate errors on all length scales. The simplest and most common multigrid algorithm is the so-called *V-cycle*, shown in Alg. 1. In practice, the system is not solved exactly on each level, but rather some simple iterative method, called the *smoother*, $\mathbf{M}^{-1}$, is applied to reduce the residual.

The original multigrid methods apply these ideas geometrically for problems on a Cartesian mesh. In this case, the restriction and prolongation operators are based on simple geometric weighting factors, and the resulting solvers and preconditioners are often very effective for largely homogeneous and isotropic Poisson-type problems.

**Algorithm 1** The standard multigrid V-cycle.

$b_0 \leftarrow \text{RHS}$
**for** $k = 0$ to $N_{\text{level}} - 2$ **do** ▷ Downward sweep of the V-cycle
  $x_k \leftarrow \mathbf{M}_k^{-1} b_k$ ▷ Apply the pre-smoother
  $r_k \leftarrow b_k - \mathbf{A}_k x_k$ ▷ Compute the residual
  $b_{k+1} \leftarrow \mathbf{R}_k r_k$ ▷ Restrict the residual to the next level
**end for**
$k \leftarrow N_{\text{level}} - 1$
$x_k \leftarrow \mathbf{M}_k^{-1} b_k$ ▷ Apply pre-smoother to the coarsest level
$r_k \leftarrow b_k - \mathbf{A}_k x_k$ ▷ Compute residual on the coarsest level
$x_k \leftarrow \mathbf{M}_k^{-1} r_k$ ▷ Apply post-smoother to coarsest level
**for** $k = N_{\text{level}} - 2$ down to 0 **do** ▷ Upward sweep of the V-cycle
  $x_k \leftarrow x_k + \mathbf{P}_k x_{k+1}$ ▷ Apply the coarse-grid correction
  $r_k \leftarrow b_k - \mathbf{A}_k x_k$ ▷ Compute the new residual
  $x_k \leftarrow x_k + \mathbf{M}_k^{-1} r_k$ ▷ Apply the post-smoother
**end for**

The strong heterogeneity and anisotropy that typifies reservoir matrices, however, renders this geometric multigrid ineffective. However, a later generalization, known as *algebraic multigrid* (AMG), constructs

a grid hierachy based solely on the coefficient matrix itself, specifically on the strength of the coefficients connecting the unknowns. Two general classes of AMG algorithms have emerged in the literature: 1) classical AMG methods in which the unknowns in the system are partitioned into coarse variables (which are propagated to the next coarser level), and fine variables (which are not propagated); 2) aggregation-based methods in which the degrees of freedom are pooled into small collections, called aggregates, of strongly-connected variables. While it should not be stated without exception, it has been our general observation that aggregation-based AMG offers faster hierarchy construction in the setup phase, while classical AMG offers more robust convergence in the solve phase. Because of the strong heterogeneity of reservoir systems, we chose to implement classical AMG schemes.

*AMG on GPUs*

Implementing AMG on GPUs poses several difficulties related to the performance characteristics of the processors. Since GPUs are optimized for high-parallel throughput, they require that a large number (presently in the thousands) of independent threads be available for exectution in order to reach peak efficiency. Furthermore, the execution units for these threads are ganged into groups which share scheduling resources. While threads in the same group can take oposite branches in a conditional, this *branch divergence* incurs a signicant performance penalty. The dynamic nature of many AMG subalgorithms can result in significant load imbalance between execution units unless great care is taken to avoid it. Finally, the GPU memory subsystem achieve peak bandwidth only when load/store operations occur in contiguous blocks. The indirection involved with sparse matrix representations often makes these *coallesced* memory operations impossible.

Early work on GPU-accelerated AMG for computational fluid dynamics focused on efficient sparse matrix-vector products for use during the solution phase (Haase et al. (2010)), but the most challenging problems relate to the AMG hierarchy construction, which proceeds in four main steps. First, the unknowns in the level matrix $\mathbf{A}_k$ are partitioned by labeling each unknown as a coarse or fine variable. Next, the prolongation operator, $\mathbf{P}_k$, is constructed to interpolate solutions on the coarse level, $k+1$, to the fine level, $k$. $\mathbf{P}_k$ is then transposed to form the restriction operator, $\mathbf{R}_k$, which transfers residuals from the fine grid to the coarse grid. Finally, the coarse level matrix, $\mathbf{A}_{k+1}$ is constructed as a triple matrix product, $\mathbf{R}_k\mathbf{A}_k\mathbf{P}_k$. Because of the coarsening, the dimensions of $\mathbf{A}_{k+1}$ are smaller than $\mathbf{A}_k$, typically by a factor of two to four. This four-step process is then applied recursively to level $k+1$, until the level matrix is small enough to utilize an exact solver efficiently, or until no coarse points remain.

*Coarse/fine partitioning*

The most well-known algorithms for coarse/fine partitioning are attributed to Ruge and Stüben (1987). While effective, the algorithm is inherently serial in nature. Later modifications of these methods exposed coarse-grained parallelism by applying the serial algorithm on separate chunks of the matrix, then repairing the boundaries between domains in a second pass (Henson and Yang (2001)). While suitable for MPI or OpenMP-based parallelism in which each core is responsible for a substantial part of the matrix, the thousands of threads required for efficient operation on GPUs would result in virtually all of the variables being on the boundary.

Fortunately, more appropriate algorithms are available, such as the parallel maximal independent set (PMIS) method (De Sterck et al. (2004)). This algorithm first weights each unknown by the number of neighbors to which it is strongly connected. Each weight is augmented by a random value in the range $(0,1]$ to break ties. Each variable is then marked as a coarse point if its weight exceeds that of all its neighbors. The immediate neighbors of the coarse points are then marked as fine, and the process repeated until all points have been labeled. Each iteration of this partitioning procedure can be performed in parallel, and is thus amenable to GPU execution.

*Interpolator construction*

A downside of the use of PMIS is that, in contrast to many serial partitioning algorithms, some fine points will have no coarse points as strongly-connected neighbors. As a result, a *distance-two* interpolation procedure is typically required (De Sterck et al. (2007)). In particular, we chose to implement two such interpolation algorithms: *standard* and *extended*. In practice, we find these two methods provide similar convergence rates, but the former is slightly easier to implement efficiently on GPUs.

The interpolation operator requires determining the set of coarse points, and associated weights, from which each fine point will be interpolated. For distance-two methods, this requires a loop not only over each fine point's neighbors, but its neighbors' neighbors as well. While this is straightforward in principle, it can lead to a large dynamic load imbalance on a GPU if not handled properly. However, this nested loop can be roughly mapped to computing the product of two sparse matrices. Hence, with appropriate modifications, we can utilize similar techniques as for the Galerkin product discussed below.

*Galkerkin product*

After the interpolator is constructed, its tranpose is taken to form the restriction operator, which carries residuals from each level to the next coarser one. Finally, we need to construct the *level matrices*, $\mathbf{A}_k$, which are approximately solved at each level. The level matrix is constructed as the variational product

$$\mathbf{A}_{k+1} = \mathbf{R}_k \mathbf{A}_k \mathbf{P}_k, \tag{2}$$

which we compute in two distinct matrix-matrix product operations.

Consider, then, a single sparse matrix-matrix product, $\mathbf{C} = \mathbf{AB}$, with each matrix in the compressed sparse row (CSR) format. Computing this product on a GPU poses two significant challenges. First, the variance in the number of nonzeros in each row of $\mathbf{A}$ and $\mathbf{B}$ can lead to a significant load imbalance if a single GPU thread is assigned to a row of $\mathbf{A}$. Forming the $i^{\text{th}}$ row of $\mathbf{C}$ involves taking each nonzero $A_{ij}$ in the $i^{\text{th}}$ row of $\mathbf{A}$ and using it to scale the $j^{\text{th}}$ row of $\mathbf{B}$. These scaled rows are then summed to form row $i$ of $\mathbf{C}$. However, it is frequently the case that nonzeros occur in the same columns of these scaled rows, resulting in duplicate column entries if the rows are simply concatenated in the sparse representation of the product. The second challenge is to merge these duplicate column entries into single coefficients efficiently in parallel.

The first issue was addressed in an elegant and relatively efficient fashion by Bell et al. (2011). We employ a version of the method in that paper which has been modified for greater efficiency. In that work, the duplicate column elimination stage was handled through a two-stage global sorting procedure, followed by a stream compaction. While very efficient GPU sorts are employed, this is still a computational bottleneck. In contrast, we use a hash-based approach, which greatly reduced the total computation time.

*The solution stage*

The solution stage of the AMG preconditioner involves two primary operations: the sparse matrix-vector product (SpMv) and the application of the smoother. Efficient methods to perform the SpMv on GPU have been addressed extensively in the literature (Bell and Garland (2008)). During the solution stage, we utilize the hybrid (HYB) format, which is an optimized partitioning of the matrix between an ELL and COO formats.

Efficient smoothing can be challenging on a massively parallel system. The simplest smoothers, such as Jacobi, often do not provide sufficient strength to achieve convergence. Perhaps the most commonly used AMG smoother on CPUs is Gauss-Seidel, but this method is inherently serial. However, it is possible to expose paralellism by an appropriate reordering of the matrix, called multicoloring. In this

method, the unknowns are divided into independent subsets, labeled by colors. All of the unknowns of a given color can then be updated simultaneously with the equivalent of Jacobi smoothing, before synchronizing and proceeding to the next color.

Multicolored smoothing requires an additional setup step which partitions the unknowns into the independent subsets. Determining the optimal coloring (which minimizes the number of required colors) is known to be an NP-hard problem, but good heuristic methods exist which produce sub-optimal, but still very high-quality colorings. The most common is the greedy algorithm, which assigns the lowest available nonconflict color to each unknown in sequence. Clearly this sequential algorithm is not ideal for execution on GPUs, but parallel coloring schemes exist which are very efficient (Gebremedhin (1999)). We have also explored reordering the matrices based on the coarse-fine partitioning of the unknowns, and have found that doing so can improve the robustness of the smoothers.

More sophisticated smoothers, such as those based upon ILU factorization, have been shown to be very effective (Heuveline et al. (2011)). However, as we show below, the solution stage of the AMG preconditioner is significantly less expensive than the setup stage, even with the relatively simple smoothers mentioned above. The time required for the factorizations would thus likely outweigh the gains from a reduced number of iterations in the solve stage. In some applications, however, the same matrix is solved for many right-hand-sides. In this case, the amortization of the additional setup time may allow a lower overall time-to-solution with more expensive smoothers.

*Two-stage preconditioning for fully-implicit matrices*

The simultaneous solution, or fully-implicit, method for reservoir simulation couples the pressure and saturation degrees of freedom in a single Jacobian matrix. While some preconditioning methods treat all unknowns on the same footing, it has been shown that a two-stage approach (Wallis et al. (1985)) can converge very quickly by first preconditioning the pressure variables alone, followed by a global preconditioner. (R. P. Hammersley (2008)) showed the use of AMG as the first-stage preconditioner can yield a very robust solver. (Cao et al. (2005)) discussed various methods to weaken the coupling between the pressure and saturation degrees of freedom, thereby speeding convergence.

These two-stage methods transfer over well to GPUs. The decoupling process is naturally parallel and the second solution stage can be effective with relatively simple preconditioners. We have explored some decoupling strategies and second-stage preconditioners, which have proven quite effective, but have not yet determined with certainty the optimal combination. We find, however, that at least for two-phase problems, combinations are available that converge in roughly the same number of iterations as the pressure matrix alone.

*Running on multiple GPUs*

The memory capacity of present GPU cards limits the size of the matrices which can be handled by the solver. We find that for typical pressure matrices, approximately 1 GB of memory is required for every million cells, which limits the maximum system size to about six million cells for the largest GPU cards presently available. To overcome this limitation, and to further reduce solution time, we can utilize several GPU cards in tandem.

Extending a linear solver to execute over multiple processors can be approached in at least two ways. The first approach aims to reproduce exactly the same operations that would be applied on a single processor. In the context of AMG, a single, global multigrid hierarchy is constructed. This approach has the advantage that the rate of converge should be identical to the single-processor case, but it requires frequent communication between processors and the bookkeeping needed to orchestrate the data transfer can add overhead. In addition, the single-hierarchy approach requires substantial coding and debugging time.

In a loosely-coupled approach, the matrix is decomposed into (sometimes overlapping) chunks which are solved independently, at least in part. The results from the separate domains are then combined in some manner to construct the global preconditioner. In AMG, this would result in an independent hierarchy for each processor. In the simplest method, known as additive Schwarz (AS), the results from each domain are simply added. Related methods, such as restricted additive Schwarz (Cai and Sarkis (1999)), and optimized Schwarz methods (Gander (2006)) improve upon AS. The multiple-hierarchy approach has two advantages: the implementation is usually relatively straightforward, and little inter-processor communication is required. The major disadvantage is that the rate of convergence is often significantly degraded. In some cases, the increased number of required iterations can outweigh the speed advantage due to parallelism, resulting in a longer time-to-solution and possible convergence failure.

To preserve robustness, we have elected to implement the single-hierarchy approach. Whenever possible, we have endeavored to reproduce exactly the results of our single-GPU solver in our multi-GPU version. In particular, if care is taken to use the same random numbers in each version, the multi-GPU version produces the same AMG hierarchy with precisely the same convergence rate as the single-GPU solver.

We employ an abstract communication model that mirrors the primitives available in MPI. Our first implementation is on top of OpenMP, using NVIDIA's GPU Direct peer-to-peer transfers. However, the abstraction should make it trivial to drop in an MPI version of the communicator to enable a scale-out solution. Recent work on GPU-aware MPI implementations (Potluri et al. (2012)) should allow very efficient MPI communication between GPUs. We minimize the communication overhead by first determining which vector elements are required by each remote GPU. These are then transferred in an efficient gather-transfer-scatter operation that minimizes the number of messages.

The bookkeeping requirements for such an approach impose some performance penalty, so that the solution speed is not directly proportional to the number of GPUs. In terms of capacity, however, the solution is scalable, allowing very large matrices to be solved efficiently. Furthermore, we employ a methodology similar that that in the HYPRE BoomerAMG package (Henson and Yang (2001)), which has demonstrated scalability up to 100,000 CPU cores for very large problems. Although we have only tested scaling within multi-GPU nodes, the scalability of the method we have employed has thus been demonstrated. At present, we are also exploring alternative schemes with an intermediate degree of coupling that may reduce bookkeeping overhead while retaining efficient convergence.

**Numerical Experiments**

In order to determine the efficiency of the GPU-based solver, we compare setup and solve times with those from the BoomerAMG subpackage in version 2.8b of the HYPRE suite of linear solvers from Lawrence Livermore National Laboratory. We compare with HYPRE for two reasons: 1) the algorithms implemented in GAMPACK are modeled after those in HYPRE and therefore the comparison is useful; 2) HYPRE is a mature, robust, and widely-used open-source software package with excellent scaling and performance sufficient to justify its use on some of the largest supercomputing installations in the world.

In the following comparisons, we have attempted to select good parameters for each solver on a case-by-case basis based on experimentation and our experience using both solvers. HYPRE's AMG preconditioner provides a very large set of adjustable parameters, however, so that an exhaustive sampling was not possible. Therefore, it is possible that additional performance may be possible with HYPRE. For both GAMPACK and HYPRE, we utilized PMIS coarsening and either standard or extended interpolation in all cases. We used flexible GMRES as the outer solver in all cases. All timings were averaged over five runs.

The CPU-based tests were run on a server with two quad-core Intel Xeon X5560 processors running

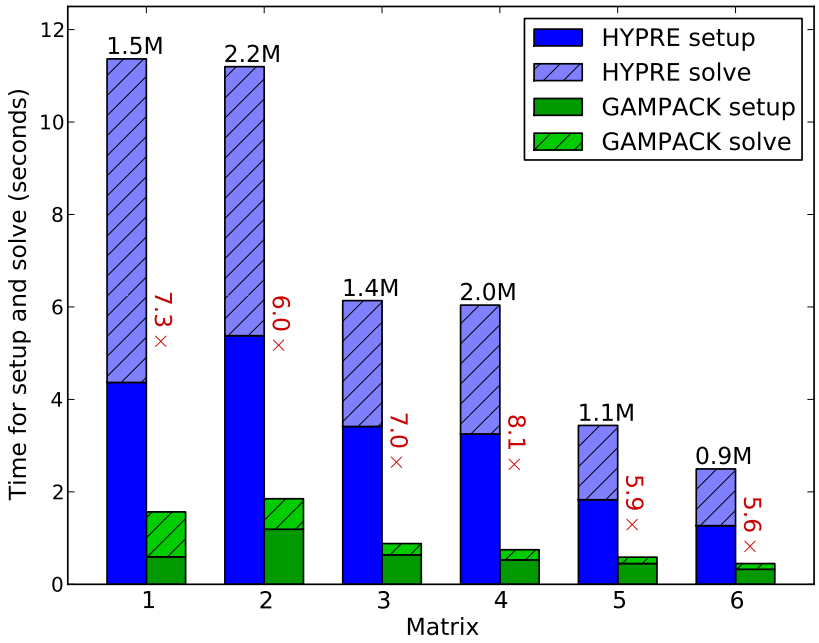


***Figure 1*** *Comparative timings for a GPU-based AMG solver (GAMPACK) and a CPU-based solver (HYPRE) on a range of contributed real and synthetic (e.g. SPE10) matrices from reservoir simulation and basin modeling programs. The number above each bar gives the number of cells and the numbers in red on the side gives the GPU speedup.*

at 2.8 GHz. The initial GAMPACK benchmarks in Figure 1 were performed on a single NVIDIA Tesla M2090 GPU card, housed in the same server. All tests were performed under Linux. Multi-GPU benchmarks shown later were also performed on a server with four Tesla M2050 GPUs.

*Single-GPU performance on pressure matrices from basin and reservoir simulation*

For purposes of our initial comparison, we have selected six representative pressure matrices from both reservoir simulation and basin modeling, with sizes in the range of about one to two millions cells. Larger matrices will be considered in the next section. Figure 1 shows a graph of the setup and solve times for these matrices. Most of these matrices are from production reservoirs, although one matrix from the synthetic SPE10 model has been included as Matrix 5.

Several observations can be made from this data. GAMPACK running on a single GPU provides a typical six-fold speed advantage over the CPU package running on two quad-core processors. When considering the setup and solve times separately, one observes that in most cases, the GPU solve stage has a larger performance advantage than the setup. The solve stage involves almost exclusively the application of SpMv and smoother operations. Since these operations are considerably more predictable and regular than the setup operations, better performance on a GPU is expected. The speedup also appears to be better for the larger matrices. This can be attributed to the greater amount of parallel work available to saturate the GPU's processing capacity, in addition to a better amortization of fixed latencies.

*Running large problems on multiple GPUs*

The solution time for a given matrix depends on both its dimensions and the intrinsic difficulty of the underlying problem. In order to disambiguate these effects, we consider a series of down-scaled problems.

| Factor | $N$ | 1×M2090 | 2×M2090 | 1×M2050 | 2×M2050 | 4×M2050 | HYPRE | Iters |
|---|---|---|---|---|---|---|---|---|
| 1 | 1.1M | 0.617 | 0.639 | 0.790 | 0.767 | 0.835 | 3.697 | 11 |
| 2 | 2.2M | 1.155 | 0.954 | 1.431 | 1.182 | 1.136 | 8.849 | 11 |
| 4 | 4.4M | 2.336 | 1.625 | | 2.012 | 1.626 | 16.641 | 12 |
| 8 | 8.9M | | 3.168 | | | 2.754 | 35.976 | 13 |
| 12 | 13.5M | | 4.686 | | | 3.981 | 56.973 | 13 |

***Table 1*** *Timings, in seconds, for GAMPACK setup and solve on single and multiple GPUs for the original SPE10 pressure matrix problem and a range of down-scaled versions. The first column gives the scaling factor, while the second gives the number of cells. All timings are in seconds and include both setup and solve to a relative tolerance of* $10^{-6}$*. HYPRE timings are given with the BoomerAMG preconditioner running with OpenMP on two quad-core Xeon X5560s using the same coarsening and interpolation algorithms as GAMPACK. Blank entries indicate insufficient GPU memory for a given problem size.*

Toward this end, we determine the time for setup and solve for a series of down-scalings of the original SPE10 problem (Christie and Blunt (2001)) generated by a streamline reservoir simulator. Since the largest systems do not fit in the 6GB DRAM of a single GPU, we include timings for runs on multiple GPUs.

As shown in Table 1, the overall GPU performance significantly exceeds the CPU performance in the widely-used HYPRE package (Henson and Yang (2001)) from Lawrence Livermore National Laboratory. The multi-GPU scaling on small problems of size one to two million unknowns is currently somewhat limited. For larger problems that require a multi-GPU solution, however, the scaling performance is quite reasonable. For the largest problem, GAMPACK running on two M2090s outperforms the two-CPU HYPRE solution (running the same coarsening and interpolation algorithms) by a factor of about 12. The 4-GPU performance is limited in part by bandwidth sharing between cards on our 1U test platform. The combination of increased lane count and the improved per-lane bandwidth in PCI-E 3.0 included in the most recent Intel processors should significantly reduce inter-GPU communication overhead.

The scaling of the SPE10 system illuminates the distinct performance characteristics of the two hardware platforms. Figure 2 is a plot of the rate of solution (including both setup and iteration, in unknowns per second) as a function of the system size. In the CPU platform, the rate of solution decreases mildly as the system size increases, which can likely be attributed to a decreasing cache hit rate. In contrast, the efficiency of the GPU implementation grows as the system size increases, for two main reasons: a higher degree of parallelism is exposed, and the ratio of inter-GPU communication to computation is decreased. For this problem, the GPU performance approaches its asymptotic maximum at about six million cells, which is near the limit imposed by the 6 GB of GPU memory. That present memory limit appears to be sufficient for the GPU to approach maximum efficiency.

**Conclusions**

We have shown that it is possible to construct a strong preconditioner with robust convergence properties that runs efficiently on GPUs. Achieving this efficiency requires a very careful selection of algorithms, combined with novel implementations suited to the performance characteristics of the massively multi-threaded GPU architecture. Although the degree of difficulty may be substantial, the potential benefits are also significant: an order-of-magnitude speedup over CPU solutions is attainable for large systems, when compared on a chip-to-chip basis.

It is clear that while accelerating only the linear solver can give a significant boost in performance, other parts of the simulator may require attention to realize the maximum benefit. Fortunately, other significant parts, such as the construction of the Jacobian, are naturally parallelizable, which should make porting

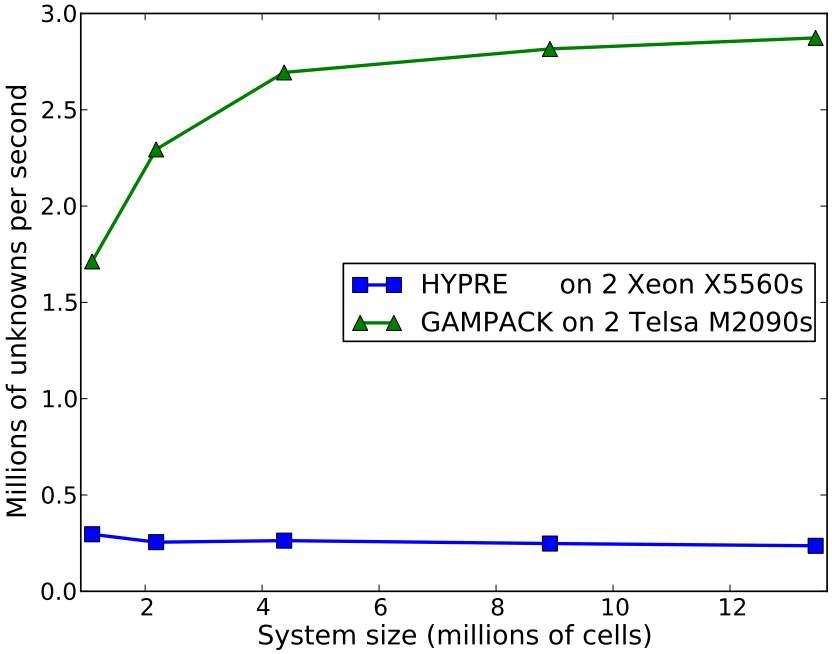


***Figure 2*** *Scaling of computation rate (in millions of unknowns per second) with system size. The rates include both setup (hierarchy construction) and iterative solution time.*

to GPUs relatively painless. File IO can be a performance bottleneck if handled sequentially, but it can be easily overlapped with GPU computation by utilizing the otherwise idle CPUs.

The practical benefits of a fully-accelerated simulator would be manyfold. For the reservoir engineer, the decrease in turn-around time for everyday simulations would increase productivity. Higher fidelity models, with little or no upscaling, could be utilized more readily. The practice of utilizing sector models to avoid excessive simulator runtime could be largely eliminated. Uncertainty analysis could become more throrough and frequent. Finally, next-generation closed-loop model optimization methods could become standard practice.

**Acknowledgements**

We would like to thank the Marathon Oil Corporation for its initial support of this work. We would also like to acknowledge Schlumberger, Permedia, and Arthur Yuldashev (Ufa State Aviation Technical Univerity, Russia), in collaboration with Ilshat Sayfullin (LTD RN-UfaNIPIneft, a project institute of Rosneft Oil Company) for contributing matrices for testing. Finally, we would like to thank NVIDIA for providing development hardware on which to test our solver.

**References**

Appleyard, J.R. and Appleyard, J.D. [2011] Accelerating reservoir simulators using GPU technology. *SPE Reservoir Simulation Symposium*, SPE 141402, Society of Petroleum Engineers.

Bell, N. et al. [2009] CUSP Library. `http://code.google.com/p/cusp-library/`.

Bell, N., Dalton, S. and Olson, L.N. [2011] Exposing fine-grained parallelism in algebraic multgrid methods. NVIDIA Technical Report NVR-2011-002, NVIDIA.

Bell, N. and Garland, M. [2008] Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004.

Cai, X.C. and Sarkis, M. [1999] A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, **21**(2), 792–797.

Cao, H., Tchelepi, H., Wallis, J. and Yadumian, H. [2005] Parallel scalable unstructured CPR-type linear solver for reservoir simulation. *SPE Annual Technical Conference and Exhibition*, SPE 96809, Society of Petroleum

Engineers.
Christie, M. and Blunt, M. [2001] Tenth SPE comparative solution project: A comparison of upscaling techniques. *SPE Reservoir Engineering and Evaluation*, (SPE 72469), 308–317.
De Sterck, H., Falgout, R., Nolting, J. and Yang, U.M. [2007] Distance-two interpolation for parallel algebraic multigrid. Tech. Rep. UCRL-JRNL-230844, Lawrence Livermore National Laboratory.
De Sterck, H., Yang, U.M. and Heys, J.J. [2004] Reducing complexity in parallel algebraic multigrid preconditioners. Tech. Rep. UCRL-JRNL-206780, Lawrence Livermore National Laboratory.
EM Photonics [2011] CULA Sparse Library. `http://www.culatools.com/sparse/`.
Gander, M.J. [2006] Optimized Schwarz methods. *SIAM Journal of Numerical Analysis*, **44**(2), 699–731.
Gebremedhin, A.H. [1999] *Parallel Graph Coloring*. Candidatus scientarum, University of Bergen.
Haase, G., Liebmann, M., Douglas, C. and Plank, G. [2010] A parallel algebraic multigrid solver on graphics processing units. In: Zhang, W., Chen, Z., Douglas, C. and Tong, W. (Eds.) *High Performance Computing and Applications*. Springer Berlin / Heidelberg, vol. 5938 of *Lecture Notes in Computer Science*, ISBN 978-3-642-11841-8, 38–47.
Henson, V. and Yang, U.M. [2001] BoomerAMG: a parallel algebraic muligrid solver and preconditioner. Technical Report UCRL-JC-141495, Lawrence Livermore National Laboratory.
Heuveline, V., Lukarski, D., Trost, N. and Weiss, J.P. [2011] Parallel smoothers for matrix-based multigrid methods on unstructured meshes using multicore CPUs and GPUs. *Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)*, 2011-09, EMCL.
Li, R. and Saad, Y. [2010] GPU-accelerated preconditioned iterative linear solvers. Tech. rep., University of Minnesota, http://www-users.cs.umn.edu/ saad/PDF/umsi-2010-112.pdf.
NVIDIA [2010] cuSPARSE library. `http://developer.nvidia.com/cusparse`.
Peaceman, D. and Rachford, Jr., H. [1955] The numerical solution of parabolic and elliptic differential equations. **3**(1), 28–41.
Potluri, S., Wang, H., Bureddy, D., Singh, A.K., Rosales, C. and Panda, D.K. [2012] Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication. *Int'l Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, Society of Petroleum Engineers.
R. P. Hammersley, D.K.P. [2008] Solving linear equations in reservoir simulation using multigrid methods. *SPE Russian Oil & Gas Technical Conference and Exhibition*, SPE 115017.
Ruge, J. and Stüben, K. [1987] Algebraic multigrid. In: McCormick, S. (Ed.) *Multigrid Methods*. SIAM, 73–130.
Rupp, K. [2012] ViennaCL library. `http://viennacl.sourceforge.net/`.
Sudan, H., Klie, H., Li, R. and Saad, Y. [2010] High performance manycore solvers for reservoir simulation. *ECMOR XII: 12th European Conference on the Mathematics of Oil Recovery*, A044.
Wallis, J. [1983] Incomplete gaussian elimination as a preconditioning for generalized conjugate gradient acceleration. *9th International Forum on Reservoir Simulation, Dallas, TX*, SPE 12265.
Wallis, J., Kendall, R. and Little, T. [1985] Constrained residual acceleration of conjugate residual methods. *SPE Reservoir Simulation Symposium, San Francisco, CA*, SPE 13536.