High performance framework for modelling of complex subsurface flow and transport applications

Khait, M.; Voskov, D.; Zaydullin, R.

DOI
10.3997/2214-4609.202035188

Publication date
2020

Document Version
Final published version

Published in
ECMOR 2020 - 17th European Conference on the Mathematics of Oil Recovery

High Performance Framework for Modelling of Complex Subsurface Flow and Transport Applications

M. Khait[1*], D. Voskov[1,2], R. Zaydullin[3]

[1] Delft University of Technology; [2] Stanford University; [3] Total E&P Research and Technology

Summary

Numerical modelling of multiphase multicomponent flow coupled with mass and energy transport in porous media is crucially important for many applications including oil recovery, carbon storage and geothermal. To deliver robust simulation results, a fully or adaptive implicit method is usually employed, creating a highly nonlinear system of equations. It is then solved with the Newton-Raphson method, which requires a linearization procedure to assemble a Jacobian matrix. Operator Based Linearization (OBL) approach allows detaching property computations from the linearization stage by using piece-wise multilinear approximations of state-dependent operators related to complex physics. The values of operators used for interpolation are computed adaptively in the parameter-space domain, which is uniformly discretized with the desired accuracy. As the result, the simulation performance does not depend on the cost of property computations, making it possible to use expensive equation-of-state formulations (e.g., fugacity-activity thermodynamic models) or even black-box chemical packages (e.g., PHREEQC) for an accurate representation of governing physics without penalizing runtime. On the other hand, the implementation of the simulation framework is significantly simplified, which allows improving the simulation performance further by executing the complete simulation loop on GPU architecture. The integrated open-source framework Delft Advanced Research Terra Simulator (DARTS) is built around the OBL concept and provides a flexible, modular and computationally efficient modelling package. In this work, we evaluate the computational performance of DARTS for various subsurface applications of practical interests on both CPU and GPU platforms. We provide a detailed performance comparison of particular workflow pieces composing Jacobian assembly and linear system solution, including both stages of Constrained Pressure Residual solver.

Introduction

Numerical simulations are essential for the modern development of subsurface reservoirs (Aziz and Settari, 1979; Dake, 1983; Peaceman, 2000). They are widely used for the evaluation of oil recovery efficiency, performance analysis, and various optimization problems. Due to the complexity of the underlying physical processes and considerable uncertainties in the geological representation of reservoirs, there is a persistent demand for more accurate models.

Fully implicit methods (FIM) are conventionally used in reservoir simulation because of their unconditional stability (Aziz and Settari, 1979). On the other hand, after discretization is applied to governing Partial Differential Equations (PDE) of a problem, the resulting nonlinear system represents different tightly coupled physical processes, which is difficult to solve. Usually, a Newton-based iterative method is applied, which demands an assembly of the Jacobian and the residual for the combined system of equations (i.e., linearization) at every iteration forming a linear system of an equal size (often ill-conditioned). Precisely the solution of such systems takes most of the simulation time in most practical applications.

Conventionally used in most practical applications Newton-based nonlinear solvers require linearization. Several conventional linearization approaches exist, though neither of them is robust, flexible, and computationally efficient all at once. Numerical derivatives provide flexibility in the nonlinear formulation (see (Xu et al., 2011), for example), but a simulation based on numerical derivatives may lack robustness and performance (Vanden and Orkwis, 1996). Straightforward hand-differentiation is the state-of-the-art strategy in modern commercial simulators (Schlumberger, 2011; Cao et al., 2009). However, this approach requires an introduction of a complicated framework for storing and evaluating derivatives for each physical property, which in turn reduces the flexibility of a simulator to incorporate new physical models and increases the probability for potential errors. The development of Automatic Differentiation (AD) technique allows preserving both flexibility and robustness in derivative computations. In reservoir simulation, the Automatically Differentiable Expression Templates Library (ADETL) was introduced by Younis (2011). Being attractive from the perspective of flexibility, the AD technique by design inherits computational overhead, which affects the performance of reservoir simulation (Khait and Voskov, 2017a).

Another linearization approach called Operator-Based Linearization (OBL) was proposed in Voskov (2017). It could be seen as an extension of the idea to abstract the representation of properties from the governing equations, suggested in Zaydullin et al. (2013) and Haugen and Beckner (2015). In the OBL approach, the parameterization is performed based on the conventional molar variables. The proposed approach was utilized for molar formulation. A similar approach can be designed for the natural formulation, but it requires dealing with several parameter spaces and switching between them.

In the OBL approach, all properties involved in the governing equations are lumped in a few operators, which are parameterized in the physical space of the simulation problem either in advance or adaptively during the simulation process. The control on the size of parameterization hyperrectangle helps to preserve the balance between the accuracy of the approximation and the performance of nonlinear solver (Khait and Voskov, 2017b). Note, that the OBL approach does not require the reduction in the number of unknowns, and only employs the fact that physical description (i.e., fluid properties) is approximated using piecewise linear interpolation.

Delft Advanced Research Terra Simulator (DARTS) was introduced and described in (Khait, 2019). It exploits the OBL approach to decouple the computations of physical properties from the main simulator core. Jacobian assembly in DARTS is therefore simplified and generalized increasing its portability to alternative computational architectures, such as GPU. In this work, we evaluate the computational performance of DARTS for various subsurface applications of practical interest on both CPU and GPU platforms. We provide a detailed performance com-

parison of particular workflow pieces composing Jacobian assembly and linear system solution, including both stages of Constraint Pressure Residual (CPR) preconditioner.

Method

Here, we briefly describe various ingredients of Delft Advanced Reservoir Terra Simulation (DARTS, 2019) framework used in this study.

Governing equations

First, we describe the conventional nonlinear formulation for a general purpose thermal compositional model. Mass and energy transport for a system with n_p phases and n_c components is considered. For this model, the n_c component mass and energy conservation equations can be written as

$$\frac{\partial m_c(\boldsymbol{\xi},\boldsymbol{\omega})}{\partial t} + \operatorname{div} f_c(\boldsymbol{\xi},\boldsymbol{\omega}) + q_c(\boldsymbol{\xi},\boldsymbol{\omega},\boldsymbol{u}) = 0, \quad c = 1,\dots,n_c+1. \tag{1}$$

Here, $\boldsymbol{\xi}$ are space-dependent parameters, $\boldsymbol{\omega}$ are state-dependent parameters and $\boldsymbol{u}$ are control variables and

$$m_c(\boldsymbol{\xi},\boldsymbol{\omega}) = \phi \sum_{j=1}^{n_p} x_{cj}\rho_p s_j, \quad c = 1,\dots,n_c, \tag{2}$$

and

$$m_c(\boldsymbol{\xi},\boldsymbol{\omega}) = \phi \sum_{j=1}^{n_p} \rho_p s_j u_j + (1-\phi)u_r, \quad c = n_c+1, \tag{3}$$

where t is time, ϕ is effective rock porosity, x_{cj} is component c concentration in phase j, ρ_j denotes phase j molar density, s_j is saturation of phase j and u_j is phase internal energy. Similarly,

$$f_c(\boldsymbol{\xi},\boldsymbol{\omega}) = \sum_{j=1}^{n_p} x_{cj}\rho_j \vec{v}_j, \quad c = 1,\dots,n_c, \tag{4}$$

and

$$f_c(\boldsymbol{\xi},\boldsymbol{\omega}) = \sum_{j=1}^{n_p} h_j\rho_j \vec{v}_j - \Big(\phi \sum_{j=1}^{n_p} \kappa_j s_j + (1-\phi)\kappa_r\Big)\nabla T, \quad c = n_c+1, \tag{5}$$

where h_j is the phase enthalpy, κ_j is phase thermal conduction and following Darcy's law

$$\vec{v}_j \;=\; -\Big(\boldsymbol{K}\frac{k_{rj}}{\mu_j}(\nabla p_j - \gamma_j \nabla D)\Big). \tag{6}$$

Here, $\boldsymbol{K}$ is the effective permeability tensor, k_{rj} is relative permeability, μ_j is phase viscosity, p_j is phase pressure, γ_j is hydrostatic gradient, and D is depth.

Operator form of governing equations

According to the Operator Based Linearization (OBL) method proposed in (Voskov, 2017), all terms in the Equation 1 are written as functions of a physical state $\boldsymbol{\omega}$ and a spatial coordinate $\boldsymbol{\xi}$. The physical state represents a unification of all state variables (i.e., nonlinear unknowns: pressure, temperature/enthalpy, saturations/compositions, etc.) of a single control volume. In the overall molar formulation, the nonlinear unknowns are pressure p, fluid enthalpy h and overall composition z_c, therefore the physical state $\boldsymbol{\omega}$ is completely defined by these variables. The spatial coordinate $\boldsymbol{\xi}$ defines the location of a given control volume which reflects the distribution of heterogeneous rock properties (e.g., porosity, thermal conduction) and elements of space discretization (e.g., transmissibility). Besides, well control variables $\boldsymbol{u}$ are introduced to represent various well management strategies.

Equation 1 is discretized in space using finite-volume two-point flux approximation and in time using backward Euler approximation. The applied Fully Implicit Method (FIM) yield that the convective flux term depends on the values of nonlinear unknowns at the current time step. Next, we rewrite Equation 1 neglecting for simplicity buoyancy and capillary forces (see more general treatment in Khait and Voskov, 2018a), and represent each term as a product space-dependent properties and of state-dependent operators (Khait and Voskov, 2018b). The resulting conservation equations read

$$\begin{aligned}\frac{V\phi_0}{\Delta t}\left(\alpha_c(\boldsymbol{\omega})-\alpha_c(\boldsymbol{\omega}^n)\right) &- \sum_l \beta_c^l(\boldsymbol{\omega})\Gamma^l(\boldsymbol{\xi})\Delta p^l \\ &- \beta_c^w(\boldsymbol{\xi},\boldsymbol{u})\Gamma^w(\boldsymbol{\xi})\Delta p^w = 0, \quad c=1,\dots,n_c, \end{aligned} \tag{7}$$

and

$$\begin{aligned}&\frac{V}{\Delta t}\Big[\phi_0\left(\alpha_e(\boldsymbol{\omega})-\alpha_e(\boldsymbol{\omega}^n)\right)+(1-\phi_0)u_r(\boldsymbol{\xi})\left(\alpha_r(\boldsymbol{\omega})-\alpha_r(\boldsymbol{\omega}^n)\right)\Big]-\sum_l \beta_h^l(\boldsymbol{\omega})\Gamma^l(\boldsymbol{\xi})\Delta p^l \\ &-\sum_l \Gamma_r^l(\boldsymbol{\xi})\Big[\phi_0\beta_e^l(\boldsymbol{\omega})+(1-\phi_0)\kappa_r\alpha_r(\boldsymbol{\omega})\Big]\Delta T^l-\beta_h^w(\boldsymbol{\omega},\boldsymbol{u})\Gamma^w(\boldsymbol{\xi})\Delta p^w=0,\end{aligned} \tag{8}$$

where V is the volume of mesh grid block, ϕ_0 is rock porosity at the reference pressure, Γ^l and Γ_r^l is the geometric part of convective and conductive transmissibility respectively, Δp^l and ΔT^l are pressure and temperature gradients at the interface l (including wells). A state-dependent operator is defined as a function of the physical state only. Therefore, it is independent of spatial position and represents physical properties of fluids and rock

$$\alpha_c(\boldsymbol{\omega}) = (1+c_r(p-p_0))\sum_{j=1}^{n_p} x_{cj}\rho_j s_j, \quad c=1,\dots,n_c, \tag{9}$$

$$\alpha_e(\boldsymbol{\omega}) = (1+c_r(p-p_0))\sum_{j=1}^{n_p} u_j\rho_j s_j, \tag{10}$$

$$\alpha_r(\boldsymbol{\omega}) = \frac{1}{1+c_r(p-p_0)}, \tag{11}$$

$$\beta_c(\boldsymbol{\omega}) = \sum_{j=1}^{n_p} x_{cj}\rho_j\frac{k_{rj}}{\mu_j}, \quad c=1,\dots,n_c, \tag{12}$$

$$\beta_h(\boldsymbol{\omega}) = \sum_{j=1}^{n_p} h_j\rho_j\frac{k_{rj}}{\mu_j}, \tag{13}$$

$$\beta_e(\boldsymbol{\omega}) = \Big(1+c_r(p-p_0)\Big)\sum_{j=1}^{n_p} s_j\kappa_j. \tag{14}$$

In the equations above, ϕ_0 - rock porosity at the reference pressure, c_r is rock compressibility, p_0 - reference pressure, while $\boldsymbol{\omega}$ and $\boldsymbol{\omega}_n$ represent the nonlinear state (unknowns of a single reservoir block) at the current and previous time step respectively.

The physical meaning of mass accumulation operator α_c is the molar mass of component c per unit pore volume of uncompressed rock under physical state $\boldsymbol{\omega}$. The physical meaning of the mass flux operator for component c is the total mobile molar mass of that component in all phases of the mixture under physical state $\boldsymbol{\omega}$ per unit time, pressure gradient, and constant geometrical part of transmissibility. This representation allows us to identify and distinguish the physical state-dependent operators in the governing conservation equations 7 and 8.

DARTS Structure

From the perspective of the simulation nonlinear loop, the operator interpolation replaces properties calculations in equations 9-14 during the Jacobian assembly step following the idea of operator-based linearization (Voskov, 2017). In addition, it also 'shadows' physical phenomena behind the operators, leaving out only the values of supporting points, which are rarely computed but utilized all the time during interpolation for Jacobian evaluation. This allows to detach fluid and rock properties calculations (now only performed during operator evaluation at supporting points) from the main nonlinear loop, as well as to relax the performance requirements for such calculations.

The Jacobian assembly depends on the choice of the nonlinear variables and the governing physical mechanisms which are taken into account. The former determine the dimensionality of parameter space, while the latter define the operators required for the assembly. Once the choice is made, the Jacobian assembly becomes simply a combination of interpolated operator values and their partial derivatives with spatial properties, encapsulated in a simulation *engine*. It is connected with an *interpolator* which is responsible for computing interpolated operator values and derivatives. This connection represents the major data workflow occurring during a simulation. Finally, the interpolator is connected to a specific set of properties (i.e., *operator set*) which are used for the simulation. Operator sets must be chosen in agreement with the selected engine. They are only invoked when a new supporting point is needed to perform the interpolation.

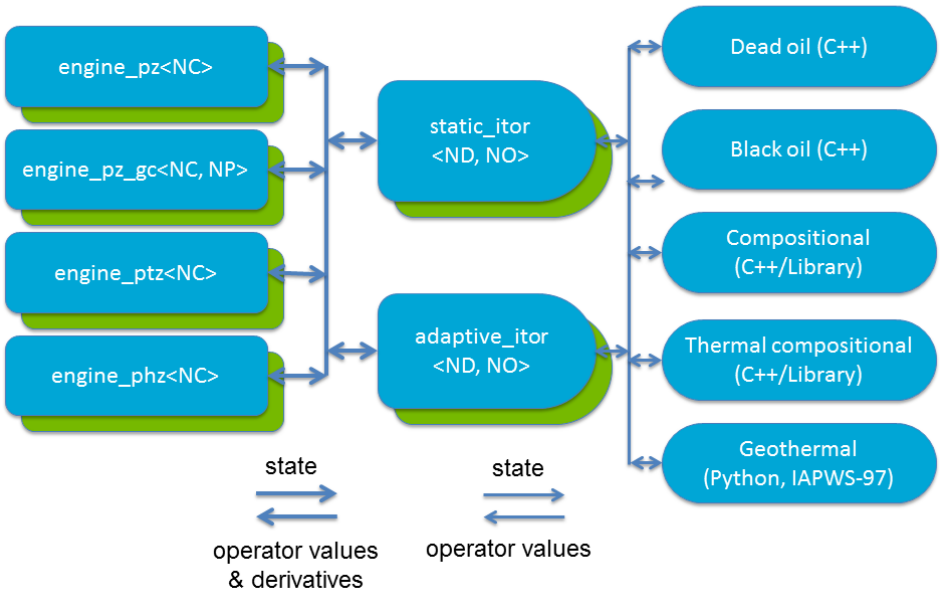


Figure 1 *Delft Advanced Research Terra Simulator (DARTS) modular structure*

The structure of DARTS is summarized in Figure 1. On the left, four simulation multiphase multi-component engines are shown:

- *engine_pz* – mass flow and transport, $\boldsymbol{\omega} = \{p, z_1, \ldots, z_{n_c-1}\}$;
- *engine_pz_gc* – mass flow and transport with gravity and capillarity, $\boldsymbol{\omega} = \{p, z_1, \ldots, z_{n_c-1}\}$;
- *engine_ptz* – mass and energy flow and transport, $\boldsymbol{\omega} = \{p, T, z_1, \ldots, z_{n_c-1}\}$;

- *engine_phz* – mass and energy flow and transport, $\boldsymbol{\omega} = \{p, h, z_1, ..., z_{n_c-1}\}$.

All engines are written in a general manner for n_c components (and n_p phases for engine_pz_gc). Notion $< NC >$ here indicates that the variable represents integer template parameter of the corresponding class, known at compile time. This approach allows to maximize various compiler optimizations (e.g., loop unrolling).

Next, two interpolators are available (Figure 1, middle):

- *static_itor* – pre-computes all supporting points in advance, and can be useful for coarse physical representation and low-dimensional parameter space;
- *adaptive_itor* – adaptively computes supporting points along with the simulation (see Khait and Voskov, 2018a, for details).

Both interpolators are written in a general way for n_d degrees of freedom and n_o operators. All operator values are stored together to benefit from faster search and interpolation. Moreover, operator values computed during simulation can be stored into a file and loaded before subsequent simulation, which can be extremely beneficial in case of running multiple models with the same physical properties common for inverse modelling or optimization.

Finally, several operator sets are present (see Figure 1, right):

- Dead-oil – water and oil components, water and oil phases, $\boldsymbol{\omega} = \{p, z_w\}$;
- Black-oil – water, oil, and gas components, water, oil, and gas phases, $\boldsymbol{\omega} = \{p, z_g, z_o\}$;
- Compositional – n_c components, liquid and vapor phases, $\boldsymbol{\omega} = \{p, z_1, ..., z_{n_c-1}\}$;
- Thermal-compositional – n_c components, liquid and vapor phases, $\boldsymbol{\omega} = \{p, T, z_1, ..., z_{n_c-1}\}$;
- Geothermal – water component, liquid and vapor phases, $\boldsymbol{\omega} = \{p, h\}$.

GPU implementation has been developed for *engine_pz*, *engine_phz* and *static_itor*. Therefore, dead-oil, black-oil, geothermal and even compositional models for a low number of components and OBL resolution can be simulated in DARTS entirely on GPU. The requirement of low parameter space resolution comes from the fact that only static interpolator has been ported to GPU. Since it computes all supporting points in advance, their amount is limited by available dynamic memory and initialization time. The rest of the engines (as well as the same engine with *adaptive_itor*) have a possibility to utilize GPU-based linear solver, which still occupies the overwhelming majority of total simulation time.

Compared to our previous work on this topic described in (Khait and Voskov, 2017a), several major improvements have been made:

- GPU implementation has been embedded into the DARTS framework;
- Jacobian assembly and static interpolation GPU kernels have been generalized for n_c components, n_d degrees of freedom and n_o operators;
- Linear solver on GPU has been written using a two-stage CPR preconditioning technique.

Jacobian Assembly on GPU

DARTS has the same initialization scheme for both GPU and CPU versions, which are developed as interchangeable parts. The GPU version first loads the required initial data to GPU memory and then performs all major computations on the device. In order to initialize the GPU version of static interpolator, all supporting points are first computed on CPU and then copied to device memory. Alternatively, they can be loaded from a file if one was created during a previous simulation and the fluid properties and physical space parameterization settings have not been changed since then. In particular, for Jacobian assembly, this data includes a connection list, pore volume and initial reservoir state arrays. The Jacobian structure is assumed to be fixed during simulation, so it is also initialized once and copied to the device memory prior to the run.

Interpolation of operator values and derivatives is performed as a preparatory step before Jacobian assembly. The kernel is implemented on a thread-per-cell basis, such that every GPU thread is responsible for computation of all operators and their derivatives for a given state $\boldsymbol{\omega}$. The data layout is analogous to the one used for CPU interpolator version, where values corresponding to a given cell are grouped together, so coalesced memory access does not take place. However, global memory accesses are minimized as the interpolation uses a workspace array placed in register memory (unless register spill occurs, which is possible for high-dimensional parameter space and a large number of operators).

The GPU version of DARTS includes only static interpolator. It is much easier for parallel execution: all operator values are generated in advance, during the initialization stage. Therefore, interpolation becomes an embarrassingly parallel procedure in this case. On the other hand, for an adaptive interpolator, there is a necessity to synchronize threads when new operator values are requested for the interpolation. Even for the CPU version, when there is no need to send generated operator values to device memory, there are different ways how to organize such synchronization. The implementation of adaptive interpolator for the GPU version is even more complicated, as the data exchange between GPU and CPU, required for the generation of new operator values, would preferably need to be overlapped with interpolation computations. In order to provide a direct comparison, we used static interpolator for both CPU and GPU versions of DARTS.

The Jacobian assembly GPU kernel, as well as all components of GPU-based linear solver (except AMG), is based on classical Block CSR matrix format. It is known that this format is not the best in terms of performance (Bell and Garland, 2009). However, it was chosen out of compatibility considerations: both with DARTS code base and available linear solver implementations on GPU. This kernel is also implemented on a thread-per-cell basis, and therefore global memory accesses are not coalesced here similarly to the interpolation kernel. In order to minimize those, the diagonal entry of Jacobian, as well as corresponding right-hand side block, are accumulated in register memory. Once the matrix row is completely processed, final values are written to corresponding global memory arrays. As opposed to diagonal entries of Jacobian, off-diagonal ones depend only on a single connection and their values are written directly to global memory.

The well part of Jacobian is first formed on a host system and then sent to a device after the assembly for the reservoir part is complete. Usually, the size of the well part is negligible compared to that of the full system, therefore the overhead is small even with synchronous memory operations. It is, however, possible to reduce the overhead to a minimum using asynchronous memory routines and CUDA streams or even recently introduced CUDA graphs. Those instruments allow overlapping of kernel execution and memory transfer. In that scenario, while the reservoir part is assembling on GPU during corresponding kernel invocation, the well part is computed on CPU and transferred to device memory. While the size of the well part is small, its computations and transfer can be hidden behind the assembly of the reservoir part almost

entirely.

Linear Solver on GPU

Khait and Voskov (2017a) used a simple configuration of GPU linear solver based on a Krylov subspace iteration method with ILU(0) preconditioner. It has limited applicability to highly heterogeneous reservoir simulation problems requiring many iterations for convergence. Significantly more robust and efficient preconditioning scheme is based on the CPR technique. Wallis (1983); Wallis et al. (1985) designed it for efficient treatment of linear systems with mixed elliptic-hyperbolic equations. Such systems arise, in particular, from FIM discretization scheme for reservoir simulation problem. They are comprised of a near-elliptic pressure equation, a near-hyperbolic composition (saturation) equation, while the temperature equation can be either type depending on whether the process is conduction- (thermal diffusion) or convection-dominated.

The CPR method is a two-stage preconditioner, where at the first stage, the pressure system is decoupled from the full system and solved separately, usually with AMG-based solver. Often a single V-cycle is enough for efficient preconditioning. At the second stage, the full system is processed by an ILU(0) preconditioner using the pressure solution from the first stage. This strategy has proved to be very robust and efficient even for highly heterogeneous reservoirs with strong coupling between elliptic and hyperbolic parts of the linear system. This results in stable convergence within a limited number of linear solver iterations even when simulation time steps are very large.

The linear system in DARTS is solved either on CPU or entirely on GPU using the Flexible Generalized Minimum Residual (FGMRES) iterative method (Saad, 1993) with CPR-based preconditioner. All matrix operations are performed in native BCSR format. The pressure system is decoupled from the FIM system using a True-IMPES reduction approach directly from the BCSR storage. Then, a single V-cycle of the AMG solver is used to obtain an approximation of the pressure solution. Finally, it is substituted in the full system and a block ILU(0) preconditioner is applied.

GPU implementation of FGMRES method is straightforward. All vectors with the linear system size, as well as the matrix itself, reside on GPU, while the plane rotation procedure is kept on CPU. Vector and matrix routines are taken from the cuBLAS (dot, scale, axpy) and the cuSPARSE (bsrmv) libraries.

CPR preconditioner is implemented in DARTS analogously to the true-IMPES reduction in AD-GPRS simulation framework described by Zhou (2012). Decoupling is performed by a single kernel which fills out the values of the pressure matrix. Its structure is equivalent to the Jacobian matrix with the only difference that the former is pointwise. Each GPU thread is again assigned to a single matrix row. For the solution phase, we developed three simple kernels (for right-hand side reduction, solution prolongation, and stage solutions summation) and employed the matrix linear combination routine from cuSPARSE (bsrmv). To ensure efficient multiprocessor occupancy, launch grid dimensions for all kernels are computed via *cudaOccupancyMaxPotentialBlockSize* routine.

For the first stage of CPR preconditioner in the CPU version of DARTS linear solver, we use proprietary AMG library. The GPU version is based on the open-source library AMGX (Naumov et al., 2015) (specifically, the commit ID 0e32e35 was used). The second stage of preconditioning uses block ILU(0) algorithm for both CPU and GPU. The GPU version of linear solver relies on the bsrilu02 routine of the cuSPARSE library.

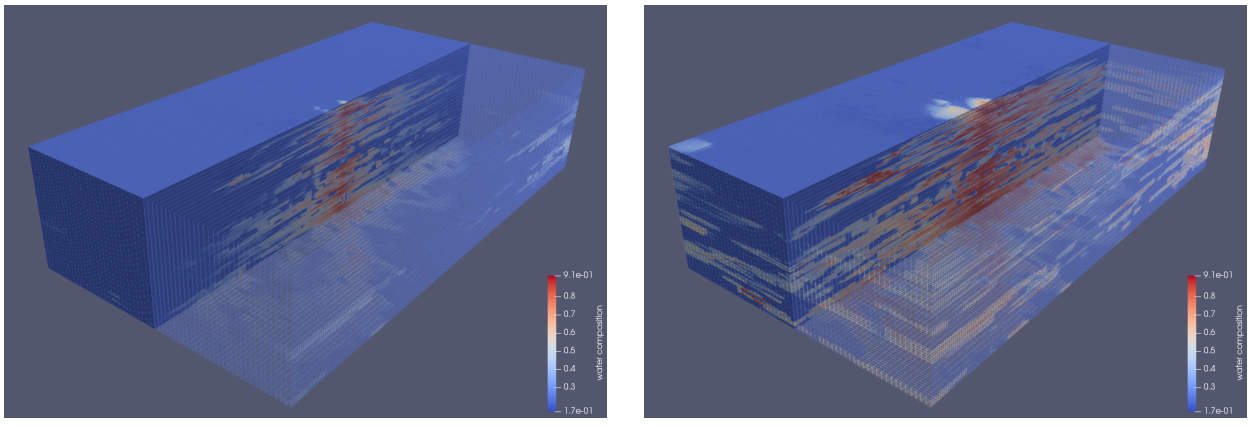
Results

The main idea behind this study was to figure out to what extent the numerical simulation can be improved by using relatively simple GPU code (maximizing the use of available libraries) and conventional hardware. For the latter, we performed our tests on a gaming workstation with Intel Core i7-8086K CPU clocked at 4.2 GHz with 32 GB DDR4 memory clocked at 2.4 GHz with a peak memory bandwidth of 41.6 GB/s and NVidia GeForce RTX2080 Ti graphics card with 11GB GDDR6 memory onboard with a peak memory bandwidth of 616 GB/s. For such generally memory-bound problems as FIM reservoir simulation, in case of parallel execution, peak memory bandwidth is a key performance indicator, not the clock speed. Therefore, the gaming card is expected to perform on the level of NVidia Tesla P100 GPU accelerator having only 15% smaller peak memory bandwidth and newer microarchitecture. At the same time, the gap between peak memory bandwidth for CPU and GPU hints the difference in the performance capacity of these platforms for reservoir simulation. The software configuration of the workstation included Ubuntu 18.04.3 operating system, GCC 7.5.0 compiler and CUDA Toolkit 10.2.

Realizing that the workstation described above does not fully reflect the efficiency of the multithread code in terms of relative performance because of both relatively low peak memory bandwidth and relatively high CPU clock frequency, we also performed additional tests on a two-socket cluster node. The detailed description and discussion are provided in the Appendix.

SPE10

The SPE10 test case (Christie and Blunt, 2001), initially created to compare upscaling techniques, is probably the most commonly used model to benchmark the performance of reservoir simulators. Due to its highly heterogeneous permeability distribution, achieving 10 orders of magnitude, and considerable size of 1.1 million cells, this model is quite challenging for both linear and nonlinear solvers. An extensive overview of the performance achievements by different simulators on this model is given by Esler et al. (2014).



200 days 2000 days

Figure 2 *Water composition distribution in SPE10 model at different timesteps.*

Figure 2 demonstrates the heterogeneous behaviour of waterflooding process of this artificial model scaled vertically by a factor of 3. Water displaces oil from the center, where the injection well is located, to corners with producers, following various flow paths. Water injection is performed at a high rate of $Q_{inj} = 794.93$ m^{3}d^{-1} causing very fast breakthrough to all four producers, as can be seen from Figure 3.

Various simulation parameters can significantly affect the total simulation time, while the difference in the final solution will be minimal. Moreover, by adjusting nonlinear and linear

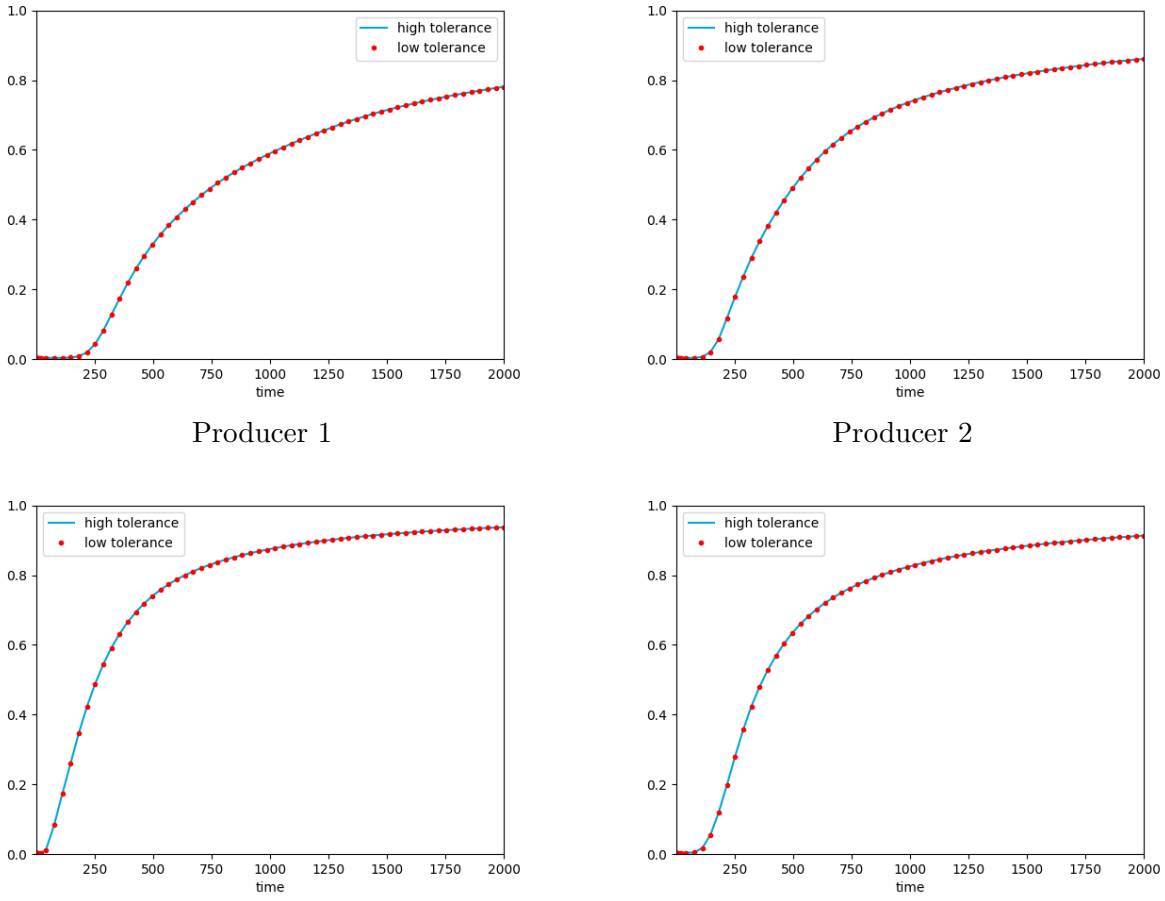


Figure 3 *Watercut for production wells in SPE10. Almost exact match between solutions obtained from simulation with high and low tolerance for both linear and nonlinear solvers.*

tolerances, one can shift the distribution of simulation time over setup and solve phases of linear solver, changing the number of linear iterations per nonlinear iterations. For the sake of direct comparison, we adjusted the nonlinear and linear tolerance trying to match the total amount of timesteps, nonlinear and linear iterations with the numbers specified by Esler et al. (2014) for their SPE10 run - 68, 418 and 1678 accordingly. With these parameters, the total runtime of their GPU simulator was reported to be 103 seconds.

Table 1 compares the overall performance of DARTS on SPE10 test for different platforms. The first line corresponds to the CPU platform on the workstation with disabled OpenMP clauses. Among all tests we have run, this configuration always leads to the fastest single-thread simulation: enabling OpenMP instructions and limiting the number of threads to one leads to a noticeable overhead of 10-15%. The second line corresponds to the multithread launch with 6 threads, which is the fastest option if total simulation time normalized by the number of linear iterations is taken into account. We tried to use 2, 4, 6, 8, 10 and 12 threads for the simulation, but starting from 6 threads, simulation time stopped to improve and only fluctuated around the same value. It is expected, since Intel Core i7-8086K processor has 6 hardware cores, while additional logical cores (or Threads, introduced by Hyper-Threading) do not help in case of memory-bound problems. The last line refers to the fully-offloaded GPU simulation on the same workstation. All runs were performed with the same nonlinear tolerance of $5 \cdot 10^{-2}$ and linear tolerance of 10^{-1}. The columns 2-4 show the number of time steps, nonlinear and linear iterations. The rest columns present the amount of time spent in initialization, Jacobian

assembly, setup and solve phases of linear solver and total simulation time.

Table 1 *Overall simulation performance of SPE10 on different platforms: CPU - Intel Core i7-8086K; GPU - NVidia GeForce RTX2080 Ti. Sequential run is denoted by (s), multithread run - with the amount of threads.*

Platform	TS	NI	LI	Init, s	Jacobian, s	Setup, s	Solve, s	Total, s
CPU (s)	68	383	1767	7.36	124.70	363.31	534.46	1039.32
CPU (6)	68	389	1794	7.16	39.36	120.77	342.27	519.35
GPU	68	417	1925	7.47	10.47	43.80	39.11	120.80

As can be seen from Table 1, the overall CPU performance on the workstation was improved twice using the multithread version with 6 threads, while some parts benefited from multithread execution more than others. For example, Jacobian assembly and linear solver setup have been improved thrice, while solving – only by a factor of 1.5.

The GPU version improves the total execution time by 8.6 times, compared to the fastest sequential run. Note that the initialization stage consumes around the same amount of time, compared to the CPU platform, but its contribution is much more significant for GPU since all the rest stages have been substantially improved. Jacobian assembly is 11.9 times faster, linear setup is 8.3 times faster, while linear solve – 13.7 times. Note that these speedups were achieved even for a larger amount of both nonlinear and linear iterations (by around 10%) compared to CPU version.

Compared to the result reported by (Esler et al., 2014), DARTS performance on GPU is only around 15% slower, while the number of timesteps and nonlinear iterations matched closely, and the number of linear iterations was around 15% higher. On the one hand, our result has been shown 6 years later on a more modern GPU device. On the other hand, we have not spent many efforts on the GPU version, using available linear solver pieces as much as possible and keeping simple (compatible with CPU) GPU data structure.

The choice of relatively low nonlinear and linear tolerances impacted simulation performance without significant influence on accuracy. For example, using the same nonlinear tolerance, with restricted to 10^{-3} linear tolerance reduces the total number of nonlinear iterations almost twice resulting in a total simulation time of just 102 seconds, 12 of which is spent on initialization. Given that the solve phase is improved more efficiently than setup, this improvement was expected since the restriction of linear tolerance shifts the focus from the setup phase to the solve phase.

To verify that the relaxed tolerances did not impact the final solution, we also performed a simulation with high tolerances of 10^{-5} for nonlinear and 10^{-6} for linear iterative processes. Figure 3 demonstrates that there is no noticeable difference between the final results, while the overall simulation time doubles.

Table 2 *Detailed simulation performance of SPE10 on different platforms: CPU - Intel Core i7-8086K; GPU - NVidia GeForce RTX2080 Ti. Sequential run is denoted by 's' in brackets, multithread run - with the amount of threads.*

Platform	setup, s			solve, s				
	CPR	AMG	ILU(0)	GMRES	SPMV	CPR	AMG	ILU(0)
CPU (s)	23.39	313.29	26.62	102.75	103.35	66.86	279.87	84.98
CPU (6)	23.45	70.29	27.04	71.22	58.45	43.20	144.26	83.60
GPU	2.13	28.67	12.99	6.57	4.39	2.62	5.55	24.37

From Table 2, one can see detailed information about the performance of all components of the

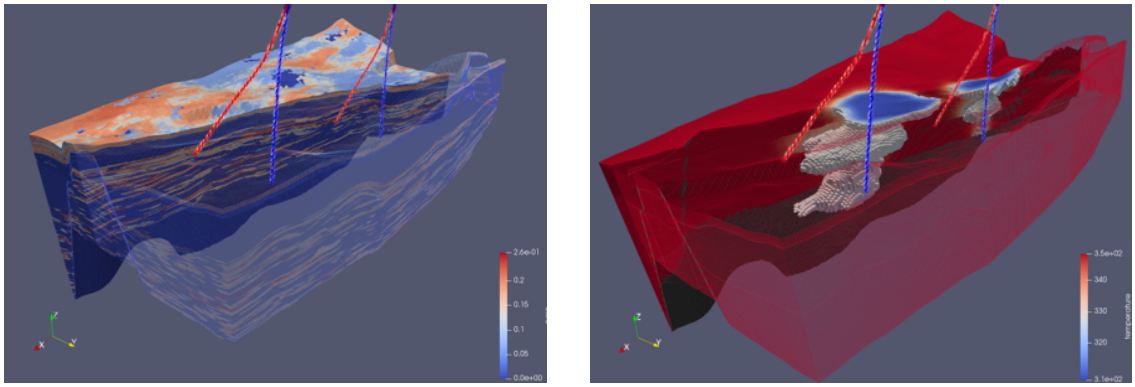
linear solver on different platforms, all times are exclusive. For example, CPR setup column shows only time spent on the construction of the pressure matrix, without taking into account time spent on subsequent calls of AMG and ILU(0) preconditioning steps. The only exception is GMRES column - it includes time spent on sparse matrix-vector multiplication (SPMV, which is also shown separately), but excludes CPR solve time.

First of all, it is easy to see that some parts of the linear solver do not speed up during the parallel run on CPU: CPR setup and both ILU(0) stages. These procedures were not made parallel, so they run sequentially in multithread execution. The acceleration of most of the paralleled code portions of linear solver lies in the range between 1.5 and 2 times with an exception for AMG setup, which is 4.4 times faster on 6 threads.

At the same time, various parts of the linear solver for GPU demonstrate different speedup ranging from 50x for AMG solve stage down to 2x for ILU(0) setup. Our CPR preconditioner setup phase shows only 10x improvement indicating that that kernel can be improved, while its solve phase shows solid 25x speedup. However, overall, the obvious weak link in the current GPU linear solver configuration is the implementation of ILU(0). It now occupies almost 50% of total linear solver time and is a first candidate on further improvement. The bsrilu02 routine of cuSPARSE library is based on the multilevel parallel approach providing an accurate solution but having a limited parallel resource. A different approach based on the multi-coloring strategy will work better here, which is one of the directions of our future research.

Realistic geothermal model

The reservoir under investigation is located in the West Netherlands Basin (WNB), which is an inverted rift basin in the Netherlands. The reservoir properties of Delft Sandstone have been extensively studied before by Willems et al. (2016, 2017). Figure 4,a shows the porosity distribution at the geological resolution of the target reservoir scaled vertically by a factor of 3. The model includes intersections of sandstone and shale facies. The facies distribution corresponds to circa 0.8 million grid blocks for the sandstone and 2.4 million blocks for shale facies.



a. Model initial porosity b.Temperature distribution after 100 years

Figure 4 *Geological model of aqueous reservoir with two geothermal doublets.*

Even though the water mainly flows through the sandstone formation, for thermal simulation it is crucially important to take shale facies into account too, as was shown in the recent benchmark study by Wang et al. (2020). The presence of the shale layers in the simulation allows the use of higher discharge rates that result in higher energy production for an equivalent system lifetime. The predicted lifetime of both doublets is significantly extended when the shale layers are included in the model. As the injected cold water transports through the sandstone layers, it is re-heated, extracting energy from the sandstone layers. As time evolves, a temperature

gradient is built up between the sandstone bodies and the neighbouring shale layers with the shales providing thermal recharge by heat conduction. The spatial intercalation of the sandstone and shale facies increases the contact area between them and amplifies the effect of the thermal recharge.

Using the full model with 3.2 mln cells, we computed the forecast for 100 years of two geothermal doublets production with the maximum time step of 365 days. The simulation results (cold water plums distribution) can be seen in Figure 4,b. The overall simulation performance of this model is shown in Table 3. It is structured the same way as Table 1. This time, the overall performance on the workstation CPU was improved by a factor of 1.7, which is lower than for SPE10. Again, the setup phase benefited more from the multithread execution than the solve phase.

The initialization stage takes significantly more time than for SPE10. It is explained by both larger model size and higher amount of rock properties required for a geothermal model. Also, the generation of operator values for the entire parameter space, which is included in initialization, takes more time because in this case, the properties are computed by an external Python library, which is noticeably less efficient than property calculations based on C++. Nevertheless, in the scenarios where the simulation runtime matters the most (e.g., inverse modeling, uncertainty quantification or optimization), the fluid properties most likely to be constant, allowing to store calculated values of OBL operators during the first run and load them in the subsequent ones.

The speedups for Jacobian assembly, setup and solve phases on CPU amounted to 2.8, 2.8 and 1.55 respectively. Lower speedup of the latter along with its larger contribution to the total simulation time explain decreased overall speedup. Analogously to the previous model, it took around 10% more linear iterations for the linear solver on GPU to converge to the same tolerance. Nevertheless, overall simulation time, compared to the fastest sequential run, was reduced by a factor of 10.3. At the same time, the sequential initialization stage contributes more than 15% of the total time. It can be entirely neglected in the computationally intensive scenarios mentioned above, so the overall simulation performance of this geothermal model can be improved on GPU by a factor of 12.

Table 3 *Overall simulation performance of geothermal model on different platforms: CPU - Intel Core i7-8086K; GPU - NVidia GeForce RTX2080 Ti. Sequential run is denoted by 's' in brackets, multithread run - with the amount of threads.*

Platform	TS	NI	LI	Init, s	Jacobian, s	Setup, s	Solve, s	Total, s
CPU (s)	107	287	4819	78.88	219.99	819.40	3885.12	5019.17
CPU (6)	107	288	4854	73.18	76.22	297.27	2494.92	2957.04
GPU	107	288	5161	78.72	18.03	78.51	276.98	486.60

The detailed simulation performance of the linear solver for this model is shown in Table 4. It is structured the same way as Table 2. Once again, the AMG setup phase benefits substantially more from multithread execution on the workstation than other phases. It runs 3.7 times faster, while other parallel stages improve only by a factor of 1.5-2 - possible because they are more memory-bound, while AMG setup is more compute-bound.

AMG solve phase, being the main contributor to the total time for sequential simulation, is boosted on GPU by a factor of 58. The other parts of linear solver expose smaller, wide-spread speedup factors, ranging from 2 to 26. Compared to the situation on CPU, this underlines the potential of further improvement of simulation on GPU.

Table 4 *Detailed simulation performance of geothermal model on different platforms: CPU - Intel Core i7-8086K; GPU - NVidia GeForce RTX2080 Ti. Sequential run is denoted by 's' in brackets, multithread run - with the amount of threads.*

Platform	setup, s			solve, s				
	CPR	AMG	ILU(0)	GMRES	SPMV	CPR	AMG	ILU(0)
CPU (s)	47.30	716.40	55.70	984.31	639.01	448.76	1885.87	566.18
CPU (6)	47.87	193.32	56.07	632.63	362.83	301.82	993.34	567.13
GPU	4.04	49.53	24.93	43.95	25.94	16.66	31.93	184.45

Conclusions

Delft Advanced Research Terra Simulator (DARTS) framework is built on top of the Operator-Based Linearization approach. It substantially simplifies Jacobian construction and reduces the time required for porting simulation code to different architectures, such as GPU. Proving this claim, we demonstrated two different examples of fully-offloaded to GPU simulations: the classical hydrocarbon production SPE10 case and a realistic model of geothermal energy production. To the best of our knowledge, this is the first simulation of a geothermal field fully offloaded to a GPU device. Compared to sequential CPU execution, the multithread version reduces simulation time almost twice on a workstation with a single socket. At the same time, the GPU version demonstrated overall improvement in the range of 8x-10x (10x-12x without sequential initialization stage). The GPU version of DARTS provides a forecast for 100 years with 3.2 million grid blocks geological model in only 7 minutes. The performance of DARTS on the full SPE10 problem is around 100 seconds which is comparable to industrial-grade simulators. All results were achieved on a regular workstation equipped with a gaming GPU graphics card.

The developed GPU linear solver uses available open-source codes as much as possible and is based on the standard BCSR matrix format. This minimized the development time and maximized the applicability of the linear solver. It can be immediately used for the whole variety of problems which can be solved in DARTS: hydrocarbon and geothermal energy production, subsurface storage and CO_2 sequestration, modeling of chemical reactions with dissolution and precipitation at reservoirs and lab scales, simulation of flow with geomechanics etc. Based on the speedup of professionally-tuned individual parts of the linear solver, its overall performance can be improved even more. The same stands for Jacobian assembly. Revision of underlying storage structures and access patterns is required to increase the simulation efficiency further and will be the focus of our future research.

Acknowledgements

We would like to acknowledge TOTAL S.A. for financial support of this research. We would like to thank Hui Cao and Arthur Yuldashev for their insightful comments and suggestions.

Appendix

The cluster node included two Intel Xeon E5-2640 v4 CPU processors clocked at 2.4 GHz with 256 Gb memory with a peak memory bandwidth of 136.6 GB/s for the two-socket system. The software configuration of the cluster node included CentOS Linux 7 operating system and GCC 4.8.5 compiler. We also used Intel C++ Compiler 16.0 as an alternative to seeing whether it provides better quality of parallel code compilation. As opposed to the workstation, the cluster node is a system with non-uniform memory access (NUMA). For such systems, it is important to prevent OpenMP threads from moving between processors to achieve higher memory bandwidth (provided that the implementation is also NUMA-aware). We performed our tests with *OMP_PLACES=cores* and *OMP_PROC_BIND=spread* environment variables achieving noticeable improvement in multithread performance in case of the cluster node. For

the workstation, thread affinity did not have a substantial effect on simulation time. Lastly, compilation flags can significantly affect the performance and there is a lot of possible options here. We limited ourselves here to a simple set *"-O3 -march=native"* for both compilers.

SPE10

Table 5 compares the overall performance of DARTS on SPE10 test for CPU platform on the workstation (denoted as CPU1) and on the cluster node (denoted as CPU2). We show the result for 6 threads to compare directly with the workstation, while 20 threads correspond to the total amount of hardware cores on the node.

Table 5 *Overall simulation performance of SPE10 on different platforms: CPU1 - Intel Core i7-8086K; CPU2 - 2 x Intel Xeon E5-2640 v4. Sequential run is denoted by 's' in brackets, multithread run - with the amount of threads. gcc represents GNU Compiler Collection 4.8.5, icc - Intel C++ Compiler 16.0*

Platform	TS	NI	LI	Init, s	Jacobian, s	Setup, s	Solve, s	Total, s
CPU1 (s)	68	383	1767	7.36	124.70	363.31	534.46	1039.32
CPU1 (6)	68	389	1794	7.16	39.36	120.77	342.27	519.35
CPU2 (s), gcc	68	383	1767	10.78	148.73	445.79	658.91	1276.76
CPU2 (6), gcc	68	389	1794	12.33	54.26	203.04	336.97	622.22
CPU2 (20), gcc	68	376	1751	14.82	22.00	128.16	215.74	394.88
CPU2 (6), icc	68	389	1794	10.87	33.67	175.45	275.22	509.93
CPU2 (20), icc	68	381	1773	11.11	15.85	137.35	221.83	401.64

As can be seen from Table 5, the sequential run on the cluster node is slower than that on the workstation by a factor of 1.2. Using only 6 threads, the overall simulation time was improved twice by GGC and 2.5 times by Intel C++ Compiler. Despite the cluster node has a slower processor (in terms of sequential code runtime), the total time with 6 threads is nevertheless faster here than on the workstation thanks to higher memory bandwidth. Using 20 threads, both compilers speed up the simulation thrice. In particular, Jacobian assembly time was improved 9.3 times on the cluster node with 20 threads. The performance improvement of both linear setup and linear solve phases on the cluster was around 3 times for both compilers.

Table 6 *Detailed simulation performance of SPE10 on different platforms: CPU1 - Intel Core i7-8086K; CPU2 - 2 x Intel Xeon E5-2640 v4. Sequential run is denoted by 's' in brackets, multithread run - with the amount of threads. gcc represents GNU Compiler Collection 4.8.5, icc - Intel C++ Compiler 16.0*

Platform	setup, s			solve, s				
	CPR	AMG	ILU(0)	GMRES	SPMV	CPR	AMG	ILU(0)
CPU1 (s)	23.39	313.29	26.62	102.75	103.35	66.86	279.87	84.98
CPU1 (6)	23.45	70.29	27.04	71.22	58.45	43.20	144.26	83.60
CPU2 (s), gcc	28.99	381.00	35.77	134.28	126.80	89.70	325.02	109.91
CPU2 (6), gcc	42.38	120.92	39.73	56.63	47.81	26.45	136.07	117.81
CPU2 (20), gcc	35.76	54.54	37.85	27.86	21.83	13.61	60.50	113.76
CPU2 (6), icc	31.62	96.06	47.77	38.03	32.26	22.21	92.16	122.82
CPU2 (20), icc	32.88	53.14	51.32	20.87	19.48	12.73	48.99	139.24

From Table 6, it is easy to spot sequential parts of linear solver - CPR setup and both ILU(0) stages. Interestingly, the NUMA memory design makes parallel code faster but interferes sequential code. This effect is more pronounced for Intel C++ compiler. These sequential sections are also the reason behind similar overall performance of multithread execution for both compilers on cluster node: GCC provided faster execution of sequential components while Intel C++ compiler - for parallel ones. The acceleration of paralleled code portions of the linear solver is in the range of 6 to 7 times on the cluster node for 20 threads with Intel C++ Compiler. Importantly, some of those sections were sometimes significantly faster with 6 threads on the cluster

node than on the workstation. For example, CPR solve took 43 seconds on the workstation and 22 seconds on the cluster node. Other stages, like AMG setup, were, on the opposite, slower. This confirms our assumption about the balance between compute-bound and memory-bound for specific linear solver parts.

Realistic geothermal model

The overall performance on the workstation was improved by a factor of 1.7, while on the cluster node with 20 threads the improvement factor reached 2.5 - which is lower than that for SPE10 for both systems (see Table 7). The cluster node again was faster than the workstation even with 6 threads (in case of Intel C++ Compiler).

On the contrary to SPE10, the best result overall on CPU for this model is provided by GCC compiler. The speedups achieved for Jacobian assembly, setup and solve phases amounted to 7.5, 3.5 and 3.4, while with Intel C++ Compiler - 9.6, 3.2 and 2.7. Despite better parallel performance during Jacobian assembly, the time loss on sequential parts of the linear solver for Intel C++ Compiler overweighted time gain on parallel parts.

Table 7 *Overall simulation performance of geothermal model on different platforms: CPU1 - Intel Core i7-8086K; CPU2 - 2 x Intel Xeon E5-2640 v4. Sequential run is denoted by 's' in brackets, multithread run - with the amount of threads. gcc represents GNU Compiler Collection 4.8.5, icc - Intel C++ Compiler 16.0*

Platform	TS	NI	LI	Init, s	Jacobian, s	Setup, s	Solve, s	Total, s
CPU1 (s)	107	287	4819	78.88	219.99	819.40	3885.12	5019.17
CPU1 (6)	107	288	4854	73.18	76.22	297.27	2494.92	2957.04
CPU2 (s), gcc	107	286	4803	99.46	266.95	1082.38	5073.58	6544.14
CPU2 (6), gcc	107	290	4870	98.98	89.91	513.81	2541.99	3272.76
CPU2 (20), gcc	107	291	4916	103.04	35.21	305.41	1506.01	1976.64
CPU2 (6), icc	107	291	4884	90.48	60.91	449.27	2075.36	2702.21
CPU2 (20), icc	107	294	4959	89.56	27.69	332.53	1892.83	2370.72

Table 8 yet again demonstrates different speedup behaviour for various linear solver components on different systems. On the workstation, AMG setup runs 3.7 times faster, while other parallel stages improve only by a factor of 1.5-2. On the cluster node, it is the opposite: AMG setup is improved there only by a factor of 3.3 (6 threads, Intel C++ Compiler), while other kernels - by factors ranging from 3.5 to 4.4. Surprisingly, for 20 threads Intel C++ Compiler does not demonstrate the better performance of parallel sections and is still slower in sequential ones. The best speedups of parallel phases for the geothermal model on the cluster node provided by GCC compiler are ranging from 5.9 to 6.7.

Table 8 *Detailed simulation performance of geothermal model on different platforms: CPU1 - Intel Core i7-8086K; CPU2 - 2 x Intel Xeon E5-2640 v4. Sequential run is denoted by 's' in brackets, multithread run - with the amount of threads. gcc represents GNU Compiler Collection 4.8.5, icc - Intel C++ Compiler 16.0*

Platform	setup, s			solve, s				
	CPR	AMG	ILU(0)	GMRES	SPMV	CPR	AMG	ILU(0)
CPU1 (s)	47.30	716.40	55.70	984.31	639.01	448.76	1885.87	566.18
CPU1 (6)	47.87	193.32	56.07	632.63	362.83	301.82	993.34	567.13
CPU2 (s), gcc	60.11	946.23	76.04	1386.92	796.25	617.85	2330.57	738.24
CPU2 (6), gcc	80.89	345.51	87.41	479.66	285.95	190.50	878.12	993.70
CPU2 (20), gcc	80.23	141.41	83.77	221.20	120.91	97.44	395.57	791.80
CPU2 (6), icc	67.12	282.99	99.15	328.43	180.24	165.78	665.32	915.82
CPU2 (20), icc	70.73	161.11	100.69	247.99	132.74	134.99	463.85	1046.00

References

Aziz, K. and Settari, T. [1979] *Petroleum Reservoir Simulation*. Applied Science Publishers.

Bell, N. and Garland, M. [2009] Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.

Cao, H., Crumpton, P. and Schrader, M. [2009] Efficient general formulation approach for modeling complex physics. **2**, 1075–1086.

Christie, M. and Blunt, M. [2001] Tenth SPE comparative solution project: A comparison of upscaling techniques. *SPE Reservoir Evaluation and Engineering*, **4**(4), 308–316.

Dake, L.P. [1983] *Fundamentals of reservoir engineering*, 8. Elsevier.

DARTS [2019] Delft Advanced Research Terra Simulator. `https://darts.citg.tudelft.nl`.

Esler, K. et al. [2014] Realizing the Potential of GPUs for Reservoir Simulation. In: *ECMOR XIV-14th European Conference on the Mathematics of Oil Recovery*.

Haugen, K. and Beckner, B. [2015] A general flow equation framework. *SPE Reservoir Simulation Symposium 2015*, **2**, 1310–1318.

Khait, M. [2019] *Delft Advanced Research Terra Simulator: General Purpose Reservoir Simulator with Operator-Based Linearization*. Ph.D. thesis, TU Delft.

Khait, M. and Voskov, D. [2017a] GPU-Offloaded General Purpose Simulator for Multiphase Flow in Porous Media. In: *SPE Reservoir Simulation Conference*. Society of Petroleum Engineers.

Khait, M. and Voskov, D. [2018a] Adaptive parameterization for solving of thermal/compositional nonlinear flow and transport with buoyancy. *SPE Journal*, **23**, 522–534.

Khait, M. and Voskov, D. [2018b] Operator-based linearization for efficient modeling of geothermal processes. *Geothermics*, **74**, 7–18.

Khait, M. and Voskov, D.V. [2017b] Operator-based linearization for general purpose reservoir simulation. *Journal of Petroleum Science and Engineering*, **157**, 990 – 998.

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N. et al. [2015] AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, **37**(5), S602–S626.

Peaceman, D.W. [2000] *Fundamentals of numerical reservoir simulation*, 6. Elsevier.

Saad, Y. [1993] A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing*, **14**(2), 461–469.

Schlumberger [2011] ECLIPSE Reference Manual 2011.2. Tech. rep., Schlumberger, Houston.

Vanden, K. and Orkwis, P. [1996] Comparison of numerical and analytical Jacobians. *AIAA Journal*, **34**(6), 1125–1129.

Voskov, D.V. [2017] Operator-based linearization approach for modeling of multiphase multicomponent flow in porous media. *Journal of Computational Physics*, **337**, 275–288.

Wallis, J. [1983] Incomplete Gaussian elimination as a preconditioning for generalized conjugate gradient acceleration. *Proc. of 7th SPE Symposium on Reservoir Simulation*.

Wallis, J., Kendall, R., Little, T. and Nolen, J. [1985] Constrained residual acceleration of conjugate residual methods. *Proc. of 8th SPE Symposium on Reservoir Simulation*.

Wang, Y., Voskov, D., Khait, M. and Bruhn, D. [2020] An efficient numerical simulator for geothermal simulation: A benchmark study. *Applied Energy*, **264**.

Willems, C., Nick, H., Weltje, G. and Bruhn, D. [2017] An evaluation of interferences in heat production from low enthalpy geothermal doublets systems. *Energy*, **135**, 500–512.

Willems, C.J.L., Goense, T., Nick, H.M. and Bruhn, D.F. [2016] The relation between well spacing and Net Present Value in fluvial Hot Sedimentary Aquifer geothermal doublets; a West Netherlands Basin case study. In: *Workshop on Geothermal Reservoir Engineering*.

Xu, T., Spycher, N., Sonnenthal, E., Zhang, G., Zheng, L. and Pruess, K. [2011] Toughreact version 2.0: A simulator for subsurface reactive transport under non-isothermal multiphase flow conditions. *Computers and Geosciences*, **37**(6), 763–774.

Younis, R. [2011] *Modern Advances in Software and Solution Algorithms for Reservoir Simulation*. Ph.D. thesis, Stanford University.

Zaydullin, R., Voskov, D. and Tchelepi, H. [2013] Nonlinear formulation based on an equation-of-state free method for compositional flow simulation. *SPE Journal*, **18**(2), 264–273.

Zhou, Y. [2012] *Parallel General-Purpose Reservoir Simulation with Coupled Reservoir Models and Multi-Segment Wells*. PhD Thesis, Stanford University.