

AL-FARABI KAZAKH NATIONAL UNIVERSITY

UDC 004.272:519.6:622.276(043.3)

Under manuscript copyright

**MAKHMUT YERLAN**

**Development of hybrid high-performance algorithms for numerical  
simulation of oil displacement problems**

8D06105 – System Engineering

Dissertation prepared for the degree  
of Doctor of Philosophy (PhD)

Domestic Scientific Supervisor:  
PhD, Associate Professor  
Imankulov Timur  
Al-Farabi Kazakh National University  
Almaty, Kazakhstan

Foreign Scientific Supervisor:  
Prof. Dr. Sergei Gorlatch,  
University of Münster, Germany

Kazakhstan, Almaty, 2026

## Contents

|                                                                              |           |
|------------------------------------------------------------------------------|-----------|
| <b>NORMATIVE REFERENCES.....</b>                                             | <b>7</b>  |
| <b>DEFINITIONS .....</b>                                                     | <b>8</b>  |
| <b>SYMBOLS AND ABBREVIATIONS.....</b>                                        | <b>9</b>  |
| <b>INTRODUCTION .....</b>                                                    | <b>11</b> |
| <b>1 Research Motivation .....</b>                                           | <b>13</b> |
| 1.1 Background and Industrial Motivation.....                                | 13        |
| 1.2 Statement of the Problem.....                                            | 14        |
| 1.3 Purpose of the Work .....                                                | 15        |
| 1.4 Research Tasks .....                                                     | 16        |
| 1.5 Scientific Novelty .....                                                 | 17        |
| 1.6 Thesis Statements Submitted for Defense .....                            | 17        |
| 1.7 Scope and Limitations .....                                              | 17        |
| 1.8 Experimental Platform.....                                               | 18        |
| 1.9 Thesis Structure .....                                                   | 18        |
| 1.10 Publications Arising.....                                               | 19        |
| <b>2 Literature Review.....</b>                                              | <b>20</b> |
| 2.1 Polymer Flooding as an Enhanced Oil Recovery Technology.....             | 20        |
| 2.1.1 Historical Context and Industrial Adoption .....                       | 20        |
| 2.1.2 Polymer Chemistry and Rheology .....                                   | 20        |
| 2.1.3 Field-Scale Deployments .....                                          | 21        |
| 2.1.4 Polymer Adsorption and Retention Mechanisms.....                       | 21        |
| 2.2 Mathematical Models of Multiphase Flow in Porous Media .....             | 22        |
| 2.2.1 Foundations: Darcy's Law and the Buckley–Leverett Theory.....          | 22        |
| 2.2.2 Governing Equations for Two-Phase Flow with Polymer Transport .....    | 22        |
| 2.2.3 Relative Permeability Models .....                                     | 23        |
| 2.2.4 Polymer Viscosity Models .....                                         | 23        |
| 2.2.5 Polymer Adsorption: The Langmuir Isotherm.....                         | 24        |
| 2.3 Numerical Methods for Reservoir Simulation.....                          | 24        |
| 2.3.1 Historical Development.....                                            | 24        |
| 2.3.2 The IMPES Scheme .....                                                 | 24        |
| 2.3.3 Finite-Difference Discretisation.....                                  | 25        |
| 2.3.4 Upwind Discretisation for Hyperbolic Transport .....                   | 25        |
| 2.3.5 Iterative Solvers for the Pressure Equation.....                       | 25        |
| 2.4 High-Performance Computing for Reservoir Simulation.....                 | 26        |
| 2.4.1 The Three Principal Parallel Paradigms.....                            | 26        |
| 2.4.2 The Roofline Performance Model.....                                    | 26        |
| 2.4.3 GPU Acceleration of Reservoir Simulation.....                          | 27        |
| 2.4.4 Multi-GPU and Hybrid Parallel Algorithms .....                         | 27        |
| 2.5 Quantitative Comparison of GPU-Accelerated Reservoir Simulation Works .. | 27        |
| 2.5.1 Detailed Examination of Prior Works.....                               | 28        |
| 2.5.2 Synthesis.....                                                         | 29        |
| 2.6 Identified Literature Gaps.....                                          | 29        |
| 2.7 Summary .....                                                            | 30        |
| <b>3 Mathematical Model .....</b>                                            | <b>31</b> |

|                                                                                |           |
|--------------------------------------------------------------------------------|-----------|
| 3.1 Modelling Assumptions.....                                                 | 31        |
| 3.1.1 A1: Linear Horizontal Flow in a Constant-Thickness Formation .....       | 31        |
| 3.1.2 A2: Two Immiscible Phases without Inter-Phase Mass Transfer.....         | 31        |
| 3.1.3 A3: Incompressible Fluids and Rock .....                                 | 32        |
| 3.1.4 A4: Darcy Flow Regime .....                                              | 32        |
| 3.1.5 A5: Negligible Gravity and Capillary Pressure .....                      | 32        |
| 3.1.6 A6: Uniform Initial Water Saturation at Irreducible Saturation .....     | 32        |
| 3.1.7 A7: Isothermal Conditions .....                                          | 32        |
| 3.2 Governing Equations .....                                                  | 33        |
| 3.2.1 Darcy's Law and Phase Velocities .....                                   | 33        |
| 3.2.2 Pressure Equation (Elliptic) .....                                       | 33        |
| 3.2.3 Saturation Equation (Hyperbolic) .....                                   | 34        |
| 3.2.4 Polymer Concentration Equation .....                                     | 34        |
| 3.3 Constitutive Relations.....                                                | 35        |
| 3.3.1 Brooks–Corey Relative Permeability.....                                  | 35        |
| 3.3.2 Permeability Reduction Factor.....                                       | 35        |
| 3.3.3 Polymer Viscosity Modification .....                                     | 36        |
| 3.3.4 Langmuir Adsorption Isotherm.....                                        | 36        |
| 3.3.5 W-Transport Relation .....                                               | 37        |
| 3.4 Coupling Structure of the Three Equations .....                            | 37        |
| 3.5 Initial and Boundary Conditions: 1D Formulation.....                       | 38        |
| 3.5.1 Initial Conditions.....                                                  | 38        |
| 3.5.2 Boundary Conditions.....                                                 | 38        |
| 3.6 Complete Coupled System .....                                              | 40        |
| 3.7 Summary .....                                                              | 40        |
| <b>4 Numerical Methods and Sequential Algorithm .....</b>                      | <b>42</b> |
| 4.1 Spatial Discretisation .....                                               | 42        |
| 4.1.1 1D Structured Cartesian Grid.....                                        | 42        |
| 4.1.2 Finite-Difference Discretisation.....                                    | 42        |
| 4.1.3 Arithmetic Averaging at Cell Interfaces .....                            | 43        |
| 4.1.4 First-Order Upwind Flux Splitting for Hyperbolic Transport .....         | 43        |
| 4.2 The IMPES Time-Stepping Scheme.....                                        | 44        |
| 4.2.1 Coupled Picard–IMPES Structure.....                                      | 44        |
| 4.2.2 Advantages and Limitations.....                                          | 44        |
| 4.3 CFL Stability Condition and Adaptive Time-Step Selection.....              | 44        |
| 4.3.1 Derivation of the CFL Condition .....                                    | 44        |
| 4.3.2 Adaptive Time-Step Algorithm.....                                        | 45        |
| 4.4 Algorithm 2: Implicit Pressure Solver (Picard–Jacobi).....                 | 45        |
| 4.5 Algorithm 3: Explicit Saturation Solver .....                              | 46        |
| 4.6 Algorithm 4: Polymer Concentration Solver (Nested Picard with W-Transport) |           |
| .....                                                                          | 47        |
| 4.7 Algorithm 5: Mobility, Viscosity, and Relative Permeability Update .....   | 48        |
| 4.8 The Complete Sequential Pipeline.....                                      | 48        |
| 4.9 Numerical Validation of the Serial Baseline .....                          | 49        |
| 4.9.1 1D Validation .....                                                      | 49        |

|                                                                |           |
|----------------------------------------------------------------|-----------|
| 4.9.2 Quantitative Verification.....                           | 50        |
| 4.10 Serial Baseline Performance.....                          | 51        |
| 4.11 Summary .....                                             | 52        |
| <b>5 CPU-Based Parallel Algorithms .....</b>                   | <b>54</b> |
| 5.1 OpenMP Shared-Memory Parallel Algorithm.....               | 54        |
| 5.1.1 Parallelisation Strategy.....                            | 54        |
| 5.1.2 Persistent Parallel Region Architecture .....            | 54        |
| 5.1.3 Race-Condition Analysis.....                             | 55        |
| 5.1.4 OpenMP Pressure, Saturation, and Polymer Kernels .....   | 55        |
| 5.1.5 Performance Results.....                                 | 57        |
| 5.1.6 Discussion of OpenMP Results.....                        | 57        |
| 5.1.7 Numerical Validation of OpenMP .....                     | 58        |
| 5.1.8 Intel VTune HPC Performance Characterisation.....        | 58        |
| 5.1.8.1 Speedup Curves: $S(p) = T_{\text{serial}} / T_p$ ..... | 59        |
| 5.1.8.2 Parallel Efficiency: $E(p) = S(p) / p$ .....           | 59        |
| 5.1.8.3 VTune Concurrency Histogram .....                      | 60        |
| 5.1.8.4 Elapsed-Time Breakdown .....                           | 61        |
| 5.1.8.5 OpenMP Parallel-Region Hotspots .....                  | 63        |
| 5.1.8.6 DRAM Bandwidth Distribution.....                       | 64        |
| 5.1.8.7 VTune Dashboard Summary (N = 65,536, 16 Threads).....  | 65        |
| 5.1.8.8 Summary of VTune Findings.....                         | 66        |
| 5.2 MPI Distributed-Memory Parallel Algorithm .....            | 67        |
| 5.2.1 Parallelisation Strategy.....                            | 67        |
| 5.2.2 Domain Decomposition and Load Balancing .....            | 67        |
| 5.2.3 Halo Exchange Implementation .....                       | 67        |
| 5.2.4 MPI Pressure, Saturation, and Polymer Kernels.....       | 67        |
| 5.2.5 Speedup and Parallel Efficiency .....                    | 70        |
| 5.2.6 mpiP Communication Profiling .....                       | 71        |
| 5.2.7 Amdahl Serial Fraction Fit.....                          | 74        |
| 5.2.8 OpenMP versus MPI Comparison .....                       | 74        |
| 5.2.9 mpiP Profiling: Communication Decomposition.....         | 75        |
| 5.3 Hybrid MPI + OpenMP Parallel Algorithm .....               | 76        |
| 5.3.1 Two-Level Parallelism .....                              | 76        |
| 5.3.2 MPI Thread Support: <code>THREAD_FUNNELED</code> .....   | 76        |
| 5.3.3 Halo Exchange + Allreduce in the Hybrid Algorithm .....  | 76        |
| 5.3.4 Hybrid Pressure, Saturation, and Polymer Kernels.....    | 77        |
| 5.3.5 Relative Speedup and Efficiency .....                    | 79        |
| 5.3.6 Discussion of Hybrid Results.....                        | 80        |
| 5.3.7 Throughput Analysis .....                                | 81        |
| 5.3.8 Four-Panel Hybrid Summary .....                          | 82        |
| 5.4 Comparative Analysis of the Three CPU Algorithms.....      | 82        |
| 5.4.1 Best-Configuration Speedup Comparison.....               | 82        |
| 5.4.2 Key Findings .....                                       | 83        |
| 5.5 Implications for the GPU Algorithms .....                  | 83        |
| 5.6 Summary .....                                              | 83        |

|                                                                                         |            |
|-----------------------------------------------------------------------------------------|------------|
| <b>6 GPU CUDA Algorithms .....</b>                                                      | <b>85</b>  |
| 6.1 GPU Architecture: The NVIDIA RTX 2080 Ti .....                                      | 85         |
| 6.1.1 Streaming Multiprocessors and CUDA Cores .....                                    | 85         |
| 6.1.2 On-Chip Memory Hierarchy .....                                                    | 85         |
| 6.1.3 Off-Chip Memory and Bandwidth .....                                               | 86         |
| 6.2 CUDA Global-Memory Algorithm .....                                                  | 86         |
| 6.2.1 Kernel Structure and Thread Mapping.....                                          | 86         |
| 6.2.2 Memory Coalescing .....                                                           | 87         |
| 6.2.3 Global-Memory Pressure Kernel .....                                               | 87         |
| 6.2.4 Nsight Compute Profiling Dashboard.....                                           | 88         |
| 6.2.5 Execution Time Results .....                                                      | 89         |
| 6.2.6 Speedup Heatmap.....                                                              | 90         |
| 6.2.7 Block-Size Sensitivity .....                                                      | 90         |
| 6.2.8 Discussion of Global-Memory Results .....                                         | 91         |
| 6.3 Block-Size Sensitivity and Hardware-Ceiling Analysis (Global-Memory Algorithm)..... | 92         |
| 6.3.1 Nsight Compute Speed-of-Light Throughputs .....                                   | 92         |
| 6.3.2 The FP64-Pipeline Bottleneck .....                                                | 93         |
| 6.3.3 Implication for the Shared-Memory Algorithm.....                                  | 93         |
| 6.4 CUDA Shared-Memory Algorithm: Full-Pipeline Tiling (Scientific Novelty 1) .....     | 93         |
| 6.4.1 Motivation and Literature Context.....                                            | 93         |
| 6.4.2 Tile-Loading Architecture with Halo Cells .....                                   | 94         |
| 6.4.3 Shared-Memory Tiled Pressure Kernel.....                                          | 95         |
| 6.4.4 Saturation and Polymer Kernels: Same Tiling Pattern .....                         | 95         |
| 6.4.5 Nsight Compute Profiling Dashboard.....                                           | 95         |
| 6.4.6 Performance Results.....                                                          | 96         |
| 6.4.7 Speedup Heatmap.....                                                              | 97         |
| 6.4.8 Block-Size Sensitivity .....                                                      | 98         |
| 6.5 Nsight Compute Hardware-Ceiling Analysis: Shared-Memory Algorithm.....              | 99         |
| 6.5.1 Report Structure and Profiling Methodology.....                                   | 99         |
| 6.5.2 Nsight Compute SOL Metrics.....                                                   | 99         |
| 6.5.3 Global-Memory versus Shared-Memory SOL Comparison .....                           | 101        |
| 6.5.4 Summary Panel.....                                                                | 102        |
| 6.6 Why the Two Implementations Achieve Near-Equivalent Performance.....                | 103        |
| 6.6.1 The FP64 Hardware Ceiling Explains the Near-Equivalence.....                      | 103        |
| 6.6.2 Supporting Factors .....                                                          | 103        |
| 6.6.3 Implications and Scope of the Result.....                                         | 104        |
| 6.7 Summary .....                                                                       | 104        |
| <b>7 Hybrid Multi-GPU Algorithms.....</b>                                               | <b>106</b> |
| 7.1 MPI + CUDA Multi-GPU Algorithm.....                                                 | 106        |
| 7.1.1 Architecture and Rank-to-GPU Binding.....                                         | 106        |
| 7.1.2 MPI+CUDA Pressure Pipeline .....                                                  | 106        |
| 7.1.3 MPI+CUDA Performance Results.....                                                 | 107        |
| 7.1.4 Discussion of MPI+CUDA Results .....                                              | 109        |

|                                                                           |            |
|---------------------------------------------------------------------------|------------|
| 7.2 Triple-Hybrid MPI + OpenMP + CUDA Algorithm (Scientific Novelty 2)... | 109        |
| 7.2.1 Three-Level Parallelism .....                                       | 109        |
| 7.2.2 Overlap of Communication and Computation .....                      | 110        |
| 7.2.3 Triple-Hybrid Pressure Pipeline.....                                | 110        |
| 7.3 Triple-Hybrid Performance Results.....                                | 111        |
| 7.3.1 Key Findings .....                                                  | 112        |
| 7.4 The L2-Cache-Residency Effect.....                                    | 113        |
| 7.4.1 Mechanism .....                                                     | 113        |
| 7.4.2 Implication for the Two Novelties .....                             | 113        |
| 7.5 Comprehensive Seven-Algorithm Comparison.....                         | 114        |
| 7.5.1 Performance Patterns.....                                           | 116        |
| 7.6 Summary .....                                                         | 116        |
| <b>8 Benchmarking, Validation, and Profiling.....</b>                     | <b>117</b> |
| 8.1 Benchmarking Methodology .....                                        | 117        |
| 8.1.1 Reproducibility Controls .....                                      | 117        |
| 8.1.2 Performance Metrics .....                                           | 117        |
| 8.2 Comprehensive Performance Dashboard .....                             | 117        |
| 8.2.1 Reading the Dashboard .....                                         | 118        |
| 8.3 Numerical Validation.....                                             | 119        |
| 8.3.1 Validation Methodology .....                                        | 119        |
| 8.3.2 Validation Results .....                                            | 119        |
| 8.4 Computational Complexity and Scalability.....                         | 119        |
| 8.5 Physical Consistency Checks .....                                     | 120        |
| 8.5.1 Mass Conservation .....                                             | 120        |
| 8.5.2 Monotonicity of Saturation .....                                    | 120        |
| 8.5.3 Polymer Adsorption Balance .....                                    | 120        |
| 8.6 Profiling-Tool Corroboration.....                                     | 120        |
| 8.6.1 Intel VTune: CPU DRAM Bandwidth Saturation .....                    | 120        |
| 8.6.2 mpiP: MPI Communication Decomposition.....                          | 121        |
| 8.6.3 NVIDIA Nsight Compute: FP64 Pipeline Utilisation .....              | 121        |
| 8.6.4 NVIDIA Nsight Systems: End-to-End Timeline Analysis .....           | 121        |
| 8.7 Practitioner Guidance .....                                           | 121        |
| 8.7.1 Operating-Point Recommendations .....                               | 121        |
| 8.7.2 When to Avoid Each Algorithm.....                                   | 122        |
| 8.8 Summary .....                                                         | 122        |
| <b>CONCLUSION.....</b>                                                    | <b>123</b> |
| <b>REFERENCE LIST .....</b>                                               | <b>127</b> |

## **NORMATIVE REFERENCES**

In the present dissertation for the degree of Doctor of Philosophy (PhD), references are made to the following standards and methodological documents.

Rules for the Conferment of Academic Degrees, Order No. 127 of the Ministry of Education and Science of the Republic of Kazakhstan dated March 31, 2011.

Instruction on the Dissertation and Author's Abstract Formatting (Order of the Higher Attestation Committee of the Ministry of Education and Science of the Republic of Kazakhstan dated 28 September 2004 No. 377-3zh).

Methodological Guidelines for Dissertation Preparation for Doctor of Philosophy (PhD) and Doctor by Profile Degree Candidates, al-Farabi Kazakh National University, 2025.

Publication Manual of the American Psychological Association, Seventh Edition, American Psychological Association, 2020.

## DEFINITIONS

In the present dissertation, the following terms with corresponding definitions are used.

Algorithm - a finite sequence of computational operations used to transform input reservoir-simulation data into pressure, saturation, concentration, or performance results.

Enhanced oil recovery - a set of tertiary recovery methods used to increase oil production after primary and secondary recovery mechanisms have become insufficient.

Polymer flooding - a chemical enhanced-oil-recovery method in which polymer is injected with water to increase the displacing phase viscosity, improve the mobility ratio, and increase sweep efficiency.

Reservoir simulation - numerical modelling of multiphase flow, transport, and recovery processes in porous geological media.

IMPES scheme - an implicit-pressure, explicit-saturation time-stepping approach in which the pressure equation is solved implicitly and the transport equations are advanced explicitly.

Parallel efficiency - the ratio between observed speedup and the number of processing elements used; it characterises the quality of parallel resource utilisation.

Scalability - the ability of an algorithm or implementation to maintain useful performance gains as the problem size, number of threads, number of processes, or number of GPUs increases.

## SYMBOLS AND ABBREVIATIONS

The following symbols and abbreviations are used in the dissertation.

- AI - artificial intelligence.
- ADI - alternating-direction implicit method.
- AMGX - NVIDIA Algebraic Multigrid library.
- ASP - alkaline-surfactant-polymer flooding.
- CFL - Courant-Friedrichs-Lewy stability condition.
- CPU - central processing unit.
- CUDA - Compute Unified Device Architecture.
- DDR4 - double data rate fourth-generation synchronous dynamic random-access memory.
- DRAM - dynamic random-access memory.
- EOR - enhanced oil recovery.
- FP32, FP64 - single-precision and double-precision floating-point arithmetic.
- GB, MB, KB - gigabyte, megabyte, and kilobyte.
- GDDR6 - graphics double data rate sixth-generation memory.
- GCC - GNU Compiler Collection.
- GPU - graphics processing unit.
- HPAM - partially hydrolysed polyacrylamide.
- HPC - high-performance computing.
- I/O - input/output.
- IMPES - implicit pressure, explicit saturation.
- L1, L2, L3 - first-, second-, and third-level cache.
- LTS - long-term support.
- MPI - Message Passing Interface.
- mpiP - lightweight MPI profiling library.
- NCCL - NVIDIA Collective Communications Library.
- NCU - NVIDIA Nsight Compute command-line profiler.
- NVIDIA - NVIDIA Corporation.
- OOIP - original oil in place.
- OpenMP - Open Multi-Processing.
- OPM - Open Porous Media initiative.
- PDE - partial differential equation.
- RAM - random-access memory.
- RTX - NVIDIA ray tracing graphics architecture/product line.
- SM - streaming multiprocessor.
- SMEM - shared memory.
- SOL - Speed-of-Light throughput analysis in NVIDIA Nsight Compute.
- SRAM - static random-access memory.
- TVD - total variation diminishing.
- VTune - Intel VTune Profiler.
- N - number of grid cells in the one-dimensional computational domain.
- x - spatial coordinate.
- t - time.
- Sw - water saturation.

$C_p$  - polymer concentration.

$p$  - pressure.

$S(p)$  - parallel speedup as a function of processing elements.

$E(p)$  - parallel efficiency as a function of processing elements.

## INTRODUCTION

This dissertation addresses the development, numerical validation, and comprehensive benchmarking of high-performance computing algorithms for solving the polymer flooding problem in oil reservoirs. The subject is positioned at the intersection of petroleum engineering, numerical analysis, and parallel computing. Hydrocarbon resources continue to supply a substantial share of global primary energy demand, while oil-producing countries must also improve the efficiency and environmental discipline of extraction from mature fields [1-2]. In this setting, enhanced oil recovery is not only a production technology but also a decision-support problem: engineers need reliable forecasts of pressure, saturation, and chemical transport under many possible injection designs before expensive field operations are started.

Polymer flooding is one of the most mature chemical enhanced oil recovery methods. By adding water-soluble polymer to the injected water, the aqueous-phase viscosity is increased, the mobility ratio between water and oil is improved, viscous fingering is reduced, and a larger fraction of the reservoir pore volume can be swept by the displacement front [3-5]. The same physical mechanisms that make polymer flooding attractive also make it difficult to simulate. Polymer concentration changes the water viscosity, polymer adsorption changes the effective transport and retention of the chemical agent, and the saturation, pressure, mobility, and concentration fields remain coupled throughout the displacement process. A practically useful simulator must therefore preserve the physics of two-phase flow while remaining fast enough for repeated scenario analysis.

The mathematical model considered in this work is a one-dimensional incompressible two-phase polymer-flooding system with pressure, water saturation, and polymer concentration as the principal unknowns. The model is formulated from mass conservation and Darcy flow assumptions, and it is solved numerically using a finite-difference IMPES-type strategy in which the elliptic pressure equation is handled implicitly and the hyperbolic saturation and polymer-concentration equations are advanced explicitly. This structure is standard in reservoir simulation, but its computational behaviour is demanding: each time step requires repeated stencil operations, nonlinear mobility updates, convergence checks, boundary-condition treatment, and communication of neighbouring-cell information when the domain is decomposed [6].

The central research problem of the dissertation is therefore not merely to write a faster implementation of an existing serial code. The problem is to determine how the coupled polymer-flooding pipeline should be mapped onto modern hardware so that numerical correctness, scalability, and hardware efficiency are achieved simultaneously. The work evaluates seven parallel computational models: OpenMP shared-memory parallelism, MPI distributed-memory parallelism, MPI + OpenMP hybrid parallelism, CUDA global-memory execution, CUDA shared-memory execution, MPI + CUDA multi-GPU execution, and a triple-hybrid MPI + OpenMP + CUDA implementation. Each model is compared with the same serial baseline, on the same governing equations, using the same validation metrics, so that the performance

conclusions are attributable to the parallelisation strategy rather than to changes in the mathematical problem.

Two scientific contributions organise the dissertation. The first is a full-pipeline CUDA shared-memory tiling approach in which the pressure, saturation, and polymer-concentration kernels are treated as a coupled computational pipeline rather than as isolated kernels. The second is a triple-hybrid MPI + OpenMP + CUDA algorithm that combines thread-level host parallelism, process-level domain decomposition, and GPU acceleration in a single implementation. Both contributions are assessed through numerical validation, speedup and efficiency analysis, memory-bandwidth interpretation, and profiling evidence from Intel VTune, mpiP, NVIDIA Nsight Compute, and NVIDIA Nsight Systems. The dissertation contains eight numbered chapters, followed by a conclusion and a consolidated reference list.

# 1 Research Motivation

## 1.1 Background and Industrial Motivation

The global energy sector still depends heavily on hydrocarbon resources, which together supply roughly 80% of the world's primary energy demand [1]. Renewable technologies, solar, wind, hydroelectric, and nuclear, have grown rapidly over the last two decades, yet the absolute volume of liquid hydrocarbon production has continued to rise during the same period, driven mainly by transport, petrochemical, and industrial-heat demand. The International Energy Agency's most recent World Energy Outlook projects that even under aggressive decarbonisation scenarios, petroleum will hold a significant share of the global energy mix well into the second half of this century [2]. Efficient extraction of crude oil from underground reservoirs is therefore not merely an engineering challenge - it is a strategic and economic priority for oil-producing nations, including the Republic of Kazakhstan, where petroleum exports account for a substantial share of national export revenue.

Conventional primary recovery, which relies on natural reservoir pressure to drive oil toward the producing wells, typically extracts only 10 to 20% of the original oil in place (OOIP) [3]. Adding secondary recovery methods, principally waterflooding, raises the cumulative recovery factor to between 20 and 40% in most reservoirs. The remaining 60 to 80% of OOIP stays trapped in the pore space of the reservoir rock, held in place by capillary forces, by viscous fingering at the displacement front, by unfavourable mobility ratios between the injected water and the resident crude, and by the tendency of injected water to bypass low-permeability zones. This residual oil represents an enormous economic resource (at typical reservoir sizes of hundreds of millions to billions of barrels OOIP, even a few percentage points of additional recovery translates to billions of dollars of incremental value over the productive life of a field), yet it can only be accessed through enhanced oil recovery (EOR) techniques [3].

Enhanced oil recovery is conventionally divided into three principal categories. Thermal methods, including steam injection and in-situ combustion, apply heat to reduce oil viscosity and improve mobility, and are particularly effective in heavy-oil and bituminous reservoirs. Gas injection methods, including miscible CO<sub>2</sub> flooding, hydrocarbon gas injection, and immiscible nitrogen displacement, exploit the favourable phase behaviour of the injected gases to reduce interfacial tension and to mobilise capillary-trapped oil. Chemical EOR methods, including polymer flooding, surfactant flooding, alkaline flooding, and various combinations such as ASP flooding, modify the rheological and interfacial properties of the displacing fluid in order to improve sweep efficiency and to reduce residual oil saturation.

Among the chemical EOR family, polymer flooding has emerged as the most technically mature and the most widely deployed approach in commercial operations [4]. The basic idea behind polymer flooding is to dissolve long-chain water-soluble polymer molecules, most commonly partially hydrolysed polyacrylamide (HPAM) of molecular weight 5–25 million Daltons, directly into the injection water. The added polymer raises the effective viscosity of the aqueous phase from a baseline of about 1 centipoise (pure water) to values of 5–50 centipoise (typical polymer solution),

bringing the water viscosity into line with or above the resident crude-oil viscosity. The resulting reduction in the water–oil mobility ratio suppresses viscous fingering at the displacement front, improves the volumetric sweep efficiency of the flood, and reduces the residual oil saturation behind the front. Field deployments in China (notably the Daqing oilfield, the largest polymer-flooding operation in the world), the United States (Wilmington, several Gulf Coast pilots), Canada (East Bodo, Pelican Lake, and several heavy-oil fields in Alberta and Saskatchewan), and the North Sea (the Captain field) have demonstrated incremental recovery improvements of 5 to 15 percentage points above the waterflood baseline [4-5], which over time translates to billions of additional barrels of cumulative production worldwide.

The intelligent design of a polymer flooding strategy depends critically on accurate, fast, and scalable reservoir simulation. Before committing capital expenditure to a polymer flood, the reservoir engineer must determine the optimal polymer concentration, the injection schedule, the well-pattern geometry, and the expected production response over the productive life of the field. Each of these decisions has to be supported by simulation predictions that account for the geological heterogeneity of the reservoir, the rheological behaviour of the polymer solution at reservoir temperature and salinity, the adsorption kinetics of the polymer on the rock surface, and the long-term mechanical and chemical degradation of the polymer molecules. A single full-field simulation of a three-dimensional reservoir with realistic geological heterogeneity, multi-well configurations, polymer transport, adsorption, and the complete two-phase flow physics can require tens of thousands of CPU hours when executed on conventional serial codes [6]. For an engineering team that must evaluate dozens of scenarios, perform history matching against decades of production data, or quantify uncertainty across an ensemble of geological realisations, such turnaround times are simply impractical. High-performance computing (HPC) acceleration is therefore not an optional enhancement to polymer-flooding simulation; it is a basic prerequisite for its meaningful application to field-scale engineering decisions.

## **1.2 Statement of the Problem**

The mathematical model governing polymer flooding in porous media consists of three strongly coupled, nonlinear partial differential equations (PDEs) describing the conservation of mass for the oil phase, the water phase, and the dissolved polymer species [6]. These equations are supplemented by Darcy's law for phase velocities, the Brooks–Corey relative permeability model relating phase mobilities to water saturation, a power-law polymer viscosity modification that captures the rheological effect of dissolved polymer on the aqueous phase, and a Langmuir adsorption isotherm describing polymer retention on the rock surface. The complete coupled system is solved on a structured Cartesian grid in 1D using the IMPES (Implicit Pressure – Explicit Saturation) time-stepping scheme, a strategy that decouples the elliptic pressure equation from the hyperbolic transport equations, allowing each to be solved with the numerical method most appropriate to its mathematical character.

Solving this PDE system on grids representative of production-scale reservoirs presents four interconnected computational challenges that motivate the present work:

(1) Computational scale. Realistic reservoir models discretised on structured grids require  $10^6$  to  $10^8$  grid cells to resolve geological heterogeneity at the level of fidelity needed for reliable engineering predictions. At the upper end of this range, the field arrays alone consume several gigabytes of memory; at every time step, the simulator must perform tens of arithmetic operations per cell across all kernels of the pipeline. A single 10,000-step simulation at  $N = 10^8$  cells therefore requires roughly  $10^{13}$  arithmetic operations, a workload that takes multi-hour execution on the fastest serial processors and that cannot be completed at all on workstations with insufficient memory bandwidth.

(2) Multi-physics coupling. The pressure, saturation, and polymer concentration fields are tightly coupled through the total mobility  $M(S, C_p)$ , the fractional flow function  $f_w(S, C_p)$ , and the polymer-modified water viscosity  $\mu_w(C_p)$ . The mobility depends on saturation through the relative permeability and on polymer concentration through the viscosity; the polymer concentration evolves under advection driven by the pressure gradient, which in turn depends on the mobility. Iterative solution strategies (Picard fixed-point iteration for the nonlinearity in the pressure solve, and nested Picard iteration for the coupled polymer-adsorption update) are required to handle this coupling. Each Picard sweep is itself a parallel-friendly stencil computation, but the convergence check at the end of every sweep involves a global reduction over all cells, which becomes a critical scalability bottleneck on distributed-memory platforms.

(3) Memory bandwidth bottleneck. Stencil-based finite-difference discretisations generate memory access patterns dominated by non-unit-stride reads of large arrays: at every cell update, the kernel must read the field values of the cell and its immediate neighbours from main memory, and write the updated value back. For modern multi-core CPUs, the arithmetic intensity of these kernels is below 0.5 floating-point operations per byte of data transferred, which places them firmly in the memory-bound regime of the Roofline performance model. Effective parallelisation therefore demands strategies that reduce global memory traffic: through cache blocking on CPUs, through shared-memory tiling on GPUs, and through subdomain decomposition that increases the cache-resident fraction of the working set.

(4) Time-to-decision. In production operations, engineering teams must evaluate multiple simulation scenarios within tight time windows, typically four to eight hours, to support operational decisions. Sequential codes running on a single CPU core require days to weeks for realistic problem sizes, which is incompatible with operational decision-making. The transition from sequential to parallel implementations is therefore not merely a performance optimisation; it is a transformation that converts an academically valuable but operationally inert simulation tool into a practical engineering instrument.

### 1.3 Purpose of the Work

Polymer flooding is a widely applied enhanced oil recovery technique, yet its governing equations are strongly nonlinear and computationally demanding, which limits the practical use of high-resolution simulations at field scale. The purpose of this work is to develop numerically stable and scalable parallel computational algorithms for solving the polymer-flooding problem in oil reservoirs, exercising the full

computational hierarchy from sequential CPU through shared-memory and distributed-memory CPU parallelism through single-GPU acceleration to multi-GPU hybrid execution. The objective is to identify, for each level of the hierarchy, the binding hardware constraint that limits performance, and to design parallelisation strategies that approach the relevant hardware ceiling, not merely to demonstrate speedup over a sequential baseline, but to do so within a principled framework that explains why the observed speedup is the maximum achievable on the given hardware.

## 1.4 Research Tasks

To realise the stated purpose, the research is organised into four consolidated tasks, with profiling, validation, and benchmarking against the serial baseline performed for every algorithm using Intel VTune for CPU performance analysis, mpiP for MPI communication decomposition, NVIDIA Nsight Compute for GPU kernel-level analysis, and NVIDIA Nsight Systems for end-to-end timeline analysis.

(1) Mathematical model and serial baseline. Formulate the 1D polymer-flooding problem from first principles of mass conservation and Darcy's law, including the Brooks–Corey relative permeability algorithm, the polymer viscosity modification, and the Langmuir adsorption isotherm. Develop a serial numerical algorithm based on the IMPES scheme with adaptive CFL time-step selection, Picard fixed-point iteration for the implicit pressure solve, donor-cell upwind discretisation for the explicit saturation update, and nested Picard iteration for the coupled polymer–adsorption system. Validate the serial baseline against analytical solutions to relative error below  $10^{-6}$  and against mass-conservation tests to machine epsilon.

(2) CPU-based parallel algorithms. Implement and evaluate three CPU parallelisation strategies: (a) an OpenMP shared-memory algorithm with persistent parallel regions, reduction-based convergence checks, and static scheduling of cell-loop iterations; (b) an MPI distributed-memory algorithm with structured-grid domain decomposition, MPI\_Sendrecv halo exchange, and MPI\_Allreduce for global convergence; (c) a hybrid MPI+OpenMP algorithm that combines the two paradigms with THREAD\_FUNNELED MPI thread support. For each algorithm, characterise the performance across the six grid sizes ( $N = 65,536$  to  $N = 67,108,864$ ) and across the four thread/rank counts (2, 4, 8, 16), identifying the hardware ceiling responsible for the observed speedup saturation.

(3) GPU-based parallel algorithms. Develop a CUDA global-memory GPU algorithm that maps the per-cell loops of Algorithms 2 through 5 directly onto GPU thread blocks, establishing the single-GPU baseline. Advance to a novel CUDA shared-memory GPU pipeline that tiles all three coupled physics kernels (pressure, saturation, polymer concentration) into on-chip SRAM simultaneously, the first scientific novelty of this dissertation. Extend to a multi-GPU MPI+CUDA implementation in which each MPI rank controls a dedicated GPU and inter-GPU halo exchange is mediated through the host. Characterise performance across all six grid sizes and four block sizes ( $B = 128, 256, 512, 1024$ ).

(4) Triple-hybrid algorithm and comprehensive benchmarking. Build a 1D MPI + OpenMP + CUDA triple-hybrid algorithm (the second scientific novelty), in which MPI ranks own GPUs, OpenMP threads parallelise host-side ghost-cell packing and

overlap host bookkeeping with GPU kernel execution, and CUDA shared-memory kernels handle the physics. Benchmark all seven parallel implementations against the serial baseline across all six grid sizes, validate against the serial fields to relative error below  $10^{-6}$ , and corroborate the speedup measurements through profiling-tool evidence.

### 1.5 Scientific Novelty

(1) Full-pipeline shared-memory tiling for pressure, saturation, and polymer concentration. Prior GPU work in reservoir simulation applies shared-memory optimisation to the pressure solver only, treating the saturation and polymer-transport equations with naive global-memory access patterns. This dissertation tiles all three coupled physics kernels into shared memory simultaneously: the `pressure_step` kernel with Picard–Jacobi iteration over harmonic-averaged transmissibilities, the saturation kernel with the explicit Godunov upwind scheme, and the `cp_step` kernel with the nested Picard iteration over polymer concentration coupled to the W-transport equation. The full-pipeline tiling is implemented with bank-conflict-free indexing, correct `__syncthreads` halo synchronisation, and validated numerical accuracy below  $10^{-6}$  relative error against the serial baseline across all six tested grid sizes.

(2) Triple-hybrid MPI + OpenMP + CUDA algorithm: peak speedup of  $34.7\times$  over the serial baseline. A three-level parallel algorithm in which MPI ranks own GPUs (Level 1), OpenMP threads parallelise host-side ghost-cell packing and overlap host bookkeeping with GPU kernel execution (Level 2), and CUDA shared-memory kernels handle the physics (Level 3). The triple-hybrid algorithm wins five of the six tested problem sizes and outperforms the pure MPI+CUDA hybrid by 2–6%, with the peak relative gain of 5.75% at  $N = 1,048,576$  cells. The algorithm demonstrates that three independent levels of parallelism (distributed memory across ranks, shared memory across host threads, and GPU acceleration of physics kernels) can be combined coherently on the same physics pipeline without algorithmic compromise.

### 1.6 Thesis Statements Submitted for Defense

(1) Shared-memory tiling can be extended from the pressure solver to the full coupled pipeline of polymer flooding. The shared-memory optimisation is applied to the pressure solver, the saturation update (explicit upwind), and the polymer concentration solver (nested Picard with W-transport). The extension is non-trivial because the saturation and polymer concentration kernels have different stencil access patterns, different convergence structures, and different boundary-condition treatments from the pressure solver.

(2) A triple-hybrid MPI + OpenMP + CUDA multi-GPU algorithm delivers measurable performance gains over single-paradigm and pair-hybrid alternatives. A three-level parallel algorithm achieves  $34.7\times$  peak speedup over the serial baseline at  $N = 1,048,576$  cells, exceeding the pair-hybrid MPI+CUDA by 5.75% at the same grid size and winning five out of six tested grid sizes.

### 1.7 Scope and Limitations

The scope of this dissertation covers the complete 1D formulation of the polymer-flooding equations under standard incompressible, immiscible, isothermal

assumptions; the IMPES time-stepping scheme with Picard fixed-point iteration; structured Cartesian grids with uniform cell spacing; single-component polymer flooding (without the alkaline or surfactant components that characterise ASP flooding); homogeneous reservoir properties (uniform porosity and isotropic permeability); and an experimental platform consisting of a single workstation with an Intel Core i9-11900KF CPU and an NVIDIA GeForce RTX 2080 Ti GPU. The limitations of the present study (1D structured grids only, no 3D unstructured; multi-GPU benchmarking limited to a small number of GPU ranks; reservoir properties assumed spatially uniform; isothermal conditions assumed; PCIe-mediated inter-GPU communication, no NVLink) are stated explicitly in the conclusion, and each is mapped to a corresponding future-work direction.

## 1.8 Experimental Platform

All measurements reported in this dissertation were obtained on a single workstation. The host operating system is Ubuntu 22.04 LTS. The host processor is an Intel Core i9-11900KF (8 physical cores, 16 logical threads with Hyper-Threading, 3.5 GHz base clock, 5.3 GHz turbo boost, 16 MB L3 cache) with 64 GB of DDR4 RAM at 51 GB/s peak bandwidth. The accelerator is an NVIDIA GeForce RTX 2080 Ti (Turing microarchitecture, compute capability 7.5) with 4,352 CUDA cores distributed across 68 streaming multiprocessors, 11 GB of GDDR6 memory at 616 GB/s peak bandwidth, 5.5 MB of L2 cache shared across all SMs, and 64 KB of unified L1/shared memory per SM. The software stack consists of CUDA Toolkit 12.4 for GPU compilation and runtime, GCC 11.4 for CPU compilation with optimisation flags `-O3 -march=native`, OpenMPI 4.1.4 for the MPI runtime, and NVIDIA driver 550.x for the GPU.

The choice of hardware platform deserves an explicit justification. The i9-11900KF and the RTX 2080 Ti are workstation-class consumer hardware, not data-centre-class hardware; they are widely accessible, well-documented, and representative of the equipment available to academic research groups and small industrial engineering teams. The performance results reported in this dissertation are therefore directly applicable to the kind of computing environment in which polymer-flooding simulation is most likely to be deployed in practice. Higher-end hardware (NVIDIA A100 or H100 with NVLink, server-class Intel Xeon or AMD EPYC processors with greater memory bandwidth, multi-node clusters with high-speed interconnects) would likely produce higher absolute performance, but the relative performance ranking of the seven parallel paradigms, together with the hardware-ceiling explanations developed in this dissertation, would be expected to generalise to those platforms as well.

## 1.9 Thesis Structure

Chapter 2 reviews the literature in four domains (polymer flooding as an EOR technology, mathematical modelling of multiphase flow, numerical methods for reservoir simulation, and HPC approaches to reservoir simulation) and identifies three literature gaps that this dissertation closes. Chapter 3 presents the mathematical model: the three governing PDEs, the constitutive relations, and the 1D boundary-condition

formulations. Chapter 4 develops the finite-difference discretisation, the IMPES time-stepping scheme, the adaptive CFL algorithm, and the four sub-algorithms of the sequential simulator. Chapter 5 presents the three CPU-based parallel algorithms (OpenMP, MPI, and the hybrid MPI+OpenMP) with full performance evaluation across all six grid sizes. Chapter 6 develops the CUDA global-memory and CUDA shared-memory algorithms, including the first scientific novelty of the dissertation. Chapter 7 presents the multi-GPU hybrids (MPI+CUDA and the triple-hybrid MPI + OpenMP + CUDA), including the second scientific novelty. Chapter 8 reports the unified seven-algorithm benchmarking and validation, with profiling-tool corroboration from VTune, mpiP, Nsight Compute, and Nsight Systems.

### **1.10 Publications Arising**

The research reported in this dissertation has been disseminated through ten scholarly outputs: four Scopus-indexed journal or conference publications, two international conference papers, and four publications in journals of the Committee for Quality Assurance in the Sphere of Education. Together these publications cover the four research tasks: the mathematical formulation and serial baseline; the CPU-based parallel algorithms; the GPU-based parallel algorithms; and the triple-hybrid algorithm with comprehensive benchmarking. The complete publication list appears again in the Conclusion; the outputs cited in the dissertation text are included in the consolidated Reference List.

## 2 Literature Review

This chapter reviews the scientific literature relevant to the four domains that underpin this dissertation: polymer flooding as an enhanced oil recovery technology (Section 2.1); mathematical modelling of multiphase flow with polymer transport in porous media (Section 2.2); numerical methods for reservoir simulation, including the IMPES family of time-stepping schemes (Section 2.3); and high-performance computing approaches applied to reservoir simulation on CPU and GPU architectures (Section 2.4). Section 2.5 then presents a quantitative comparison of seven prior GPU-accelerated reservoir simulation works against the approach proposed here, and Section 2.6 identifies the three literature gaps that the present work closes.

### 2.1 Polymer Flooding as an Enhanced Oil Recovery Technology

#### 2.1.1 Historical Context and Industrial Adoption

Enhanced oil recovery (EOR) encompasses a broad family of techniques designed to mobilise and recover crude oil that remains in the reservoir after primary depletion and secondary waterflooding [3]. The three principal EOR categories, namely thermal methods (steam injection, in-situ combustion), gas injection methods ( $CO_2$  flooding, hydrocarbon gas injection, nitrogen displacement), and chemical methods (polymer flooding, surfactant flooding, alkaline flooding), each exploit different mechanisms to reduce residual oil saturation or to improve volumetric sweep efficiency. Chemical EOR has attracted sustained research interest over the past five decades thanks to its applicability across a wide range of reservoir types, its favourable economics in medium-to-light crude oil reservoirs, and the relative ease with which chemical additives can be deployed using existing waterflood infrastructure [4].

The scientific foundations of polymer flooding were laid in the early 1960s, when two independent research groups demonstrated, through carefully controlled laboratory core-flood experiments, that the addition of high-molecular-weight water-soluble polymers to the injection water substantially improves oil displacement efficiency. Pye, working at the Marathon Oil research laboratories, showed that water-soluble polymers improved the secondary-recovery factor by controlling water mobility through the displacement front [7]. Independently, Sandiford at the Texaco Producing Company demonstrated that polymer solutions could raise oil recovery in laboratory waterfloods by ten percentage points or more above the polymer-free baseline [8]. Both groups arrived at the same fundamental mechanism: a reduction of the water–oil mobility ratio through an increase in the effective viscosity of the aqueous phase, which suppresses viscous fingering at the displacement front and improves the macroscopic sweep efficiency of the flood.

#### 2.1.2 Polymer Chemistry and Rheology

The two principal polymer types used in field applications are partially hydrolysed polyacrylamide (HPAM) and biopolymers, principally xanthan gum [9]. HPAM is a synthetic polymer produced by partial hydrolysis of polyacrylamide; the resulting polymer carries negatively charged carboxylate groups along its backbone that produce strong viscoelastic behaviour in solution, with apparent viscosities well

above those predicted by simple Newtonian rheology. HPAM molecular weights of 5 to 25 million Daltons are typical in field deployments, and aqueous concentrations of 500 to 2,500 parts per million produce effective viscosities of 10 to 50 centipoise at typical reservoir conditions. The principal drawbacks of HPAM are its sensitivity to high salinity (the carboxylate charges are screened by dissolved cations, reducing the polymer coil radius and the effective viscosity), its sensitivity to mechanical degradation in high-shear regions near injection wells, and its susceptibility to oxidative and microbial degradation over long simulation timescales.

Xanthan gum is a biopolymer produced by fermentation of *Xanthomonas campestris* bacteria. It exhibits a rigid double-helical structure in solution, which gives it superior salinity tolerance and shear stability relative to HPAM, but at substantially higher cost per pound. Xanthan deployments have therefore been limited to high-salinity reservoirs where the cost differential is justified by the improved chemical performance. Other polymer types (including hydrophobically modified polyacrylamides, scleroglucan, and various co-polymers) have been investigated in laboratory studies and in limited field pilots, but HPAM remains the dominant chemistry in commercial operations [4, 9]. Field-scale evidence for the performance advantages of HPAM over biopolymers is reviewed in the following subsection.

### **2.1.3 Field-Scale Deployments**

Field-scale polymer-flooding projects have been deployed worldwide, with the largest and most extensively documented being the Daqing oilfield in northeast China. Daqing began commercial-scale polymer injection in the mid-1990s and has continued operations for more than two decades; polymer flooding has reportedly contributed incremental recoveries of twelve to fifteen percentage points above the waterflood baseline at Daqing, which corresponds to billions of barrels of incremental cumulative production [5]. North American deployments have similarly demonstrated incremental recoveries of three to ten percentage points, with notable projects at Wilmington (California), the East Bodo and Pelican Lake heavy-oil fields in Canada [10], and several Permian Basin and Gulf Coast pilots in the United States. European deployments (most notably the Captain field in the UK North Sea) have demonstrated the applicability of polymer flooding to offshore environments, where the logistical and chemical-handling constraints differ substantially from onshore operations. In each case, the success of the deployment depended critically on accurate simulation-based design: the selection of polymer concentration, the injection schedule, the well-pattern geometry, and the timing relative to the waterflood phase were all informed by reservoir simulation studies conducted prior to commitment of capital.

### **2.1.4 Polymer Adsorption and Retention Mechanisms**

A critical phenomenon in polymer flooding (and one that distinguishes the mathematical model of polymer flooding from the simpler black-oil or two-phase flow models) is polymer retention on the rock surface. As the polymer-laden water flows through the porous medium, a fraction of the polymer molecules are adsorbed onto the rock surface through electrostatic interactions, hydrogen bonding, and physical entrapment in pore throats narrower than the polymer coil diameter. The adsorbed

polymer physically narrows the pore-throat channels available to water flow, reducing the effective permeability to water below its polymer-free value (a mechanism known as permeability reduction, which contributes to the overall mobility reduction of the water phase) and simultaneously depletes the dissolved polymer concentration in the aqueous phase. From a numerical-modelling standpoint, polymer adsorption introduces an additional state variable, the adsorbed concentration  $W$ , that must be evolved self-consistently with the dissolved concentration  $C_p$  through the Langmuir adsorption isotherm [11]. The resulting nonlinear coupling between  $C_p$  and  $W$  introduces a nested iterative structure into the polymer-concentration solver, a numerical requirement addressed in the discretisation methodology of this dissertation. These retention and permeability-reduction mechanisms follow standard polymer-flooding modelling practice [9, 11].

## **2.2 Mathematical Models of Multiphase Flow in Porous Media**

### **2.2.1 Foundations: Darcy's Law and the Buckley–Leverett Theory**

The mathematical modelling of multiphase flow in porous media has a long and well-developed history, rooted in Darcy's empirical law [12] and extended to multiphase systems through the capillary pressure and relative permeability framework formalised by Leverett [13] and applied to displacement problems by Buckley and Leverett [14]. Darcy's law in its simplest form states that the superficial velocity of a single-phase fluid through a porous medium is proportional to the pressure gradient and inversely proportional to the fluid viscosity, with the constant of proportionality being the rock permeability divided by the fluid viscosity. The extension to multiphase systems introduces the concept of relative permeability, a dimensionless function of fluid saturation that captures the reduction in the effective permeability of each phase as the saturation of the competing phase increases.

The Buckley–Leverett theory [14] provides an analytical framework for the one-dimensional displacement of oil by water under simplifying assumptions (incompressible flow, immiscible phases, no capillary pressure, no gravity). The theory predicts the formation of a sharp displacement front that propagates through the reservoir at a velocity determined by the fractional flow function, which itself depends on the relative permeabilities and viscosities of the two phases. The Buckley–Leverett solution has served as a validation reference for numerical reservoir simulators for more than seven decades, and it is the basis for the serial-baseline validation of this dissertation in Chapter 4.

### **2.2.2 Governing Equations for Two-Phase Flow with Polymer Transport**

The governing equations for incompressible, immiscible two-phase flow in a porous medium are derived from mass conservation for each phase, combined with Darcy's law and the saturation constraint [6, 15]. The extension to include polymer transport introduces a third conservation equation for the polymer species dissolved in the water phase, accounting for advection with the water-phase velocity and for source/sink terms from adsorption on the rock surface [9]. The polymer affects the flow physics through two mechanisms: viscosity modification of the aqueous phase, typically modelled through either the Todd–Longstaff mixing rule [16] or empirical

power-law correlations; and polymer adsorption onto the rock surface, modelled by a Langmuir-type isotherm [11] that captures the saturation behaviour of monolayer adsorption.

The complete mathematical model of polymer flooding comprises three nonlinear PDEs that are strongly coupled through the constitutive relations: an elliptic pressure equation derived from the sum of the two phase mass-conservation equations; a hyperbolic saturation equation derived from the water-phase mass-conservation equation; and a hyperbolic polymer transport equation derived from the polymer species mass-conservation equation, coupled to the adsorbed concentration through the Langmuir isotherm. The full coupling structure (pressure depends on saturation and polymer concentration through the total mobility; saturation depends on pressure and polymer concentration through the fractional flow function and viscosity; polymer concentration depends on saturation, pressure, and the adsorbed concentration) makes the system fundamentally three-way coupled, requiring iterative solution strategies (Picard fixed-point iteration in this dissertation; Newton–Raphson iteration in some commercial simulators) to handle the nonlinearity.

### **2.2.3 Relative Permeability Models**

Relative permeability functions are most commonly modelled using the Brooks–Corey power-law form [17], which provides a good approximation to measured relative permeability curves for a wide range of porous media and is the form adopted in this dissertation. The Brooks–Corey model expresses the water and oil relative permeabilities as power laws in the normalised water saturation, with characteristic exponents and that depend on the pore-throat geometry and the wettability of the rock. For water-wet sandstones (the most common reservoir rock type in polymer-flooding applications), exponents in the range and are typical, producing the characteristic S-shaped relative permeability curves that drive Buckley–Leverett displacement physics. More elaborate relative permeability models (Corey, LET (Lomeland–Ebeltoft–Thomas), modified Brooks–Corey with end-point modifications) exist in the literature, but Brooks–Corey is the standard choice in academic reservoir simulation and is sufficient for the present study.

### **2.2.4 Polymer Viscosity Models**

The polymer viscosity modification of the aqueous phase is most commonly modelled by one of two approaches. The first, originally introduced by Todd and Longstaff in the context of miscible flooding [16], assumes a power-law mixing of polymer-laden and polymer-free fluid viscosities and includes an empirical mixing parameter that captures the degree of fluid mixing in the simulation grid. The second, used throughout this dissertation, is a direct empirical power-law correlation between polymer concentration and aqueous-phase viscosity: . The empirical constants and are fitted to laboratory rheology data for the specific polymer chemistry and reservoir conditions; typical values for HPAM at 25°C are and [9]. This simple form has the advantage of being directly compatible with the IMPES time-stepping scheme: it can be evaluated at every cell at the start of each time step without iteration, and it captures the dominant physical behaviour adequately for the scale of problems considered here.

### **2.2.5 Polymer Adsorption: The Langmuir Isotherm**

Polymer adsorption on the rock surface is modelled in this dissertation by the Langmuir isotherm [11], originally derived by Irving Langmuir in 1918 to describe the adsorption of gas molecules on solid surfaces. The Langmuir isotherm assumes a monolayer of adsorbed polymer molecules, with a fixed number of adsorption sites per unit area of rock surface, and adsorption equilibrium between the dissolved and adsorbed phases at every point in the reservoir. The resulting adsorbed concentration is given by, where  $W_{\infty}$  is the maximum adsorption capacity (the value of  $W$  when all sites are occupied) and  $K_L$  is the Langmuir equilibrium constant. The isotherm saturates as  $C$  increases, reflecting the limit of the monolayer assumption. In practice, the isotherm provides a good approximation to measured polymer adsorption data over the range of polymer concentrations relevant to field deployments [9].

## **2.3 Numerical Methods for Reservoir Simulation**

### **2.3.1 Historical Development**

The discretisation and numerical solution of the multiphase flow equations in reservoir simulation has been an active research area since the pioneering work of Douglas, Peaceman, and Rachford in the late 1950s [18]. Their seminal paper established the alternating-direction implicit (ADI) approach to multidimensional immiscible displacement and laid the foundations for several decades of subsequent development. The principal subsequent advances were the introduction of the IMPES time-stepping scheme by Sheldon, Zondek, and Cardwell [19] and independently by Stone and Garder [20]; the systematic treatment of well boundary conditions and pressure-equation formulation by Peaceman [15]; the extension to compositional and thermal simulation; and the development of fully implicit methods (FIM) and adaptive implicit methods (AIM) that handle a broader range of physics regimes.

### **2.3.2 The IMPES Scheme**

The IMPES scheme adopted in this dissertation is the time-stepping approach in which the pressure equation is solved implicitly at each time step using the saturation and mobility field from the previous time step, after which the saturation (and, in this dissertation, the polymer concentration) is updated explicitly using the newly computed pressure field. The principal advantages of IMPES are computational efficiency (at each time step only the pressure equation requires a linear solve, which can be done with iterative methods such as Jacobi [21] or conjugate gradients) and natural parallelisability, since the explicit saturation and polymer concentration updates are embarrassingly parallel across cells. The principal disadvantage is conditional stability: the explicit saturation update must satisfy the Courant–Friedrichs–Lewy (CFL) condition [22], which bounds the time-step size to the smallest cell width divided by the maximum velocity in the domain. For practical reservoir simulations the CFL constraint typically permits time steps of order 1 to 10 days, which is adequate for the multi-year simulation horizons of polymer-flooding studies.

### 2.3.3 Finite-Difference Discretisation

The spatial discretisation in this dissertation uses the finite-difference method, in which the governing PDEs are approximated by replacing the continuous spatial derivatives with algebraic difference quotients evaluated at discrete grid points. On the uniform Cartesian grid adopted here with constant cell spacing  $\Delta x$  on the uniform 1D grid, all first-order spatial derivatives in the hyperbolic saturation and polymer concentration equations are replaced by first-order upwind difference quotients (the donor-cell scheme of Godunov [23]), while the elliptic pressure equation is discretised by second-order central differences, producing the standard tridiagonal system that admits efficient iterative solution by Jacobi iteration. The finite-difference method has two properties that motivate its use in this dissertation. First, the first-order upwind scheme is monotone under the CFL stability condition [22], guaranteeing that no spurious oscillations are introduced at the advancing saturation and polymer-concentration fronts. Second, on a uniform structured Cartesian grid the difference stencils reduce to closed-form expressions in  $\Delta x$ , producing regular memory-access patterns that are amenable to vectorisation and cache blocking on CPUs and to coalesced global-memory access and shared-memory tiling on GPUs. The numerical conservation of fluid mass is verified explicitly at every time step as a post-processing check, confirming that the accumulated finite-difference truncation error does not introduce spurious mass sources or sinks over the 10,000-step simulation horizon.

### 2.3.4 Upwind Discretisation for Hyperbolic Transport

The hyperbolic saturation and polymer concentration equations require special treatment to prevent spurious oscillations and to maintain monotonicity of the numerical solution. The standard approach in reservoir simulation, adopted in this dissertation, is the first-order donor-cell (Godunov) upwind scheme [23], in which the flux at each face is computed using the field value from the upstream cell, that is, the cell from which fluid is flowing into the face. Godunov's original formulation guarantees positivity preservation and monotonicity at first order, and it is the foundation for the higher-order TVD (total variation diminishing) and ENO (essentially non-oscillatory) schemes developed in the subsequent decades, as reviewed in LeVeque [24]. Higher-order schemes would reduce numerical diffusion at the displacement front, but the first-order donor-cell scheme is sufficient for the present study and has the additional advantages of being computationally cheap (single arithmetic operation per face), naturally parallelisable, and stable under the CFL condition without further modification.

### 2.3.5 Iterative Solvers for the Pressure Equation

The discretised pressure equation in this dissertation is a sparse symmetric positive-definite linear system in  $N$  unknowns, with three non-zero entries per row corresponding to the harmonic-averaged transmissibilities at cell faces. Several iterative solvers are applicable: the Jacobi method [21], with its inherently parallel structure and simple implementation but slow convergence rate; Gauss–Seidel with red–black ordering, with faster convergence but less natural parallelisability; preconditioned conjugate gradients with various preconditioners (Jacobi, SSOR,

incomplete Cholesky, algebraic multigrid); and direct multifrontal solvers for problem sizes that fit in memory. This dissertation adopts the Jacobi iterative method as the inner linear solver because of its embarrassingly parallel structure (every cell update is independent and can be assigned to a separate thread, CUDA thread, or MPI rank without inter-cell synchronisation) and because of its compatibility with all four levels of the parallelisation hierarchy (sequential CPU, OpenMP, MPI, CUDA). The slower convergence rate of Jacobi relative to more sophisticated solvers is partly mitigated by the use of an outer Picard fixed-point iteration [25] that resolves the nonlinearity of the mobility coefficients.

## **2.4 High-Performance Computing for Reservoir Simulation**

### **2.4.1 The Three Principal Parallel Paradigms**

Three principal parallelisation paradigms have been applied to reservoir simulation: shared-memory CPU parallelism (most commonly implemented with OpenMP [26]), distributed-memory CPU parallelism (most commonly with MPI [27]), and GPU acceleration (most commonly with CUDA [28] on NVIDIA hardware). Each paradigm exploits a different level of the modern HPC memory hierarchy: OpenMP exploits the shared L2/L3 caches and main memory of a single compute node; MPI exploits the aggregate memory and compute resources of a cluster of nodes connected by a high-speed network; and CUDA exploits the massively parallel architecture of GPU accelerators with their high-bandwidth on-chip memory hierarchy. The performance characteristics of each paradigm are governed by hardware constraints that can be characterised quantitatively through the Roofline performance model [29], introduced by Williams, Waterman, and Patterson in 2009 as a visual tool for identifying the binding hardware constraint of a given kernel.

### **2.4.2 The Roofline Performance Model**

The Roofline model expresses the achievable performance of a computational kernel as the minimum of two quantities: the peak floating-point throughput of the processor, and the product of the peak memory bandwidth times the arithmetic intensity of the kernel (the ratio of floating-point operations to bytes of memory traffic). Kernels with low arithmetic intensity, typically stencil computations like those of reservoir simulation, sit on the bandwidth-bound portion of the Roofline curve, with performance limited by memory bandwidth rather than by floating-point throughput. The polymer-flooding kernels developed in this dissertation have arithmetic intensities of approximately 0.25 FLOP/byte for the pressure solver, which places them firmly in the bandwidth-bound regime. The Roofline model thereby predicts a theoretical performance ceiling on achievable speedup: on the i9-11900KF with 51 GB/s peak DRAM bandwidth, the pressure-solver kernel cannot exceed approximately 13 GFLOP/s regardless of thread count or optimisation level, while the same kernel on the RTX 2080 Ti with 616 GB/s peak DRAM bandwidth can exceed approximately 154 GFLOP/s, a  $12\times$  theoretical advantage that matches the empirical CPU-to-GPU speedup ratio observed in this dissertation.

### 2.4.3 GPU Acceleration of Reservoir Simulation

GPU acceleration of reservoir simulation has progressed steadily since the introduction of CUDA in 2007 [28]. Early work focused on porting the pressure-equation solver to the GPU using available libraries (cuSPARSE for sparse linear algebra, cuBLAS for dense operations, AMGX for algebraic multigrid preconditioners), with the rest of the simulation pipeline remaining on the CPU. This library-only approach yielded modest speedups, typically 3 to 5 $\times$  over single-threaded CPU, limited by the cost of host-to-device data transfer at every time step. Subsequent work moved progressively more of the pipeline onto the GPU, with the saturation update, well terms, and constitutive relations implemented as custom CUDA kernels. The most successful implementations achieve speedups of 10 to 20 $\times$  over single-threaded CPU on representative problem sizes, with the residual gap to the theoretical Roofline ceiling attributed to GPU launch overhead, host-side bookkeeping, and the FP64 throughput constraint of consumer-class hardware. Section 2.5 reviews seven representative prior works in this category.

### 2.4.4 Multi-GPU and Hybrid Parallel Algorithms

Multi-GPU implementations of reservoir simulation have been reported primarily in proprietary commercial simulators (Schlumberger's INTERSECT, CMG's GEM, OPM Flow) where the engineering investment justifies the development cost. Open academic implementations of multi-GPU reservoir simulation are scarce. The principal technical challenges are: (1) the host-mediated halo exchange overhead on PCIe-connected systems, which can dominate the per-time-step cost if not carefully amortised; (2) the absence of GPUDirect [30] and NVLink [31] on consumer hardware, which precludes peer-to-peer GPU memory access and forces all inter-GPU communication through the host; and (3) the additional complexity of orchestrating CUDA kernels, MPI calls, and host-side bookkeeping coherently within the same time-step loop. The triple-hybrid model developed in this dissertation addresses all three challenges through a three-level architecture in which MPI handles distributed memory, OpenMP handles host-side parallelism and overlap, and CUDA handles GPU compute.

## 2.5 Quantitative Comparison of GPU-Accelerated Reservoir Simulation Works

Table 2.1 compares seven representative GPU-accelerated reservoir simulation works against the approach proposed in this dissertation. The comparison is structured along five axes: the number of GPUs employed (single-GPU versus multi-GPU); the use of global memory (universal); the use of shared memory (the optimisation that is the focus of the first scientific novelty of this dissertation); the pipeline coverage on the GPU (a single kernel, a subset of kernels, or the full pipeline); and the dominant hardware-level limitation reported by the authors.

Table 2.1 – Comparison of GPU-accelerated reservoir simulation works against the proposed approach

| Paper / Work | GPUs | Optimised Global memory | Shared memory | Pipeline | Hardware limitation |
|--------------|------|-------------------------|---------------|----------|---------------------|
|--------------|------|-------------------------|---------------|----------|---------------------|

|                                     |                  |            |                                        |                              |                              |
|-------------------------------------|------------------|------------|----------------------------------------|------------------------------|------------------------------|
| Imankulov et al., 2021 [32]         | Single GPU       | No         | Some kernels (CR kernel only)          | ADI solver only              | Memory size limit            |
| Foadaddini et al., 2022 [33]        | Single GPU       | No         | Some kernels (tridiagonal solver)      | PDE solver only              | Limited scalability          |
| Zaza et al., 2016 [34]              | Single GPU       | No         | Some kernels (SpMV + tiling, not all)  | Full flow                    | High memory cost             |
| De la Cruz & Monsivais, 2014 [35]   | Single GPU       | No         | Library only (cuSPARSE)                | Linear solver only           | Memory-BW bound              |
| Teja-Juárez & de la Cruz, 2018 [36] | Single GPU       | No         | Minimal                                | Jacobian + solver            | Limited scalability          |
| Beisembetov et al., 2014 [37]       | Single GPU       | No         | Most kernels (1D tiling)               | Full IMPES                   | Memory & grid limit          |
| Tene et al., 2023 [38]              | Single GPU       | No         | Library-based (cuSPARSE / AMGX)        | Black-oil full flow          | Single-GPU memory bound      |
| <b>This work ★</b>                  | <b>Multi-GPU</b> | <b>Yes</b> | <b>Across all P / S / Cp kernels ★</b> | <b>Full polymer pipeline</b> | <b>FP64 hardware ceiling</b> |

★ *The proposed algorithm is the only entry that combines multi-GPU parallelism, full-pipeline shared-memory tiling, and a quantified hardware ceiling.*

### 2.5.1 Detailed Examination of Prior Works

Imankulov et al. [32] developed a comparative CUDA/OpenCL implementation of an ADI pressure solver for the two-phase oil recovery problem, focused on the linear-algebra kernels of the implicit pressure solve. The work demonstrated the portability of GPU acceleration across CUDA and OpenCL runtimes, but the saturation and any polymer-transport physics remained on the CPU; shared-memory optimisation was applied only to the cyclic reduction (CR) kernel of the tridiagonal solver. Foadaddini et al. [33] presented a corrected explicit-implicit domain decomposition (CEIDD) scheme for convection-dominated diffusion problems, including a GPU implementation of the tridiagonal solver with selective shared-memory tiling. The work is a PDE-solver study rather than a full reservoir simulation; the physics modelling is limited to the convection-diffusion model rather than the three-equation polymer-flooding system.

Zaza et al. [34] developed a CUDA-based multi-phase oil reservoir simulator with full flow coverage on the GPU, applying shared-memory tiling to the SpMV (sparse matrix–vector product) kernel of the linear solver and to selected portions of the saturation update. The work is one of the most ambitious single-GPU implementations in the open literature for non-polymer reservoir simulation, but it does not address polymer transport with adsorption, and the shared-memory tiling does not extend uniformly across all kernels. De la Cruz and Monsivais [35] applied cuSPARSE library calls to the linear solver of a two-phase flow simulator, with all other physics on the CPU. The simplicity of the implementation produced modest speedups but no novel algorithmic contribution to GPU optimisation. Teja-Juárez and de la Cruz [36] extended this work with a custom GPU implementation of the Jacobian assembly and the linear solver, but the use of shared memory remained minimal and the simulation pipeline was limited to two-phase flow without polymer.

Beisembetov et al. [37] developed a CUDA implementation of a full IMPES black-oil simulator with shared-memory tiling. This is the closest prior work to the present dissertation in terms of GPU optimisation depth; the principal differences are that Beisembetov et al. did not implement polymer transport with adsorption (limiting the work to black-oil rather than polymer flooding), did not address multi-GPU configurations, and did not combine OpenMP host-side parallelism with the GPU kernels. Tene et al. [38] reported a GPU-accelerated implementation of the OPM Flow black-oil simulator using cuSPARSE and AMGX library calls; the work is the most recent (2023) and the most realistic in terms of field-scale problem sizes, but it remains a single-GPU library-based implementation and does not address polymer transport.

## 2.5.2 Synthesis

Three observations emerge from the systematic review. First, every prior work in this comparison uses a single GPU; multi-GPU academic implementations of reservoir simulators with full physics coverage have not been reported in the open literature. Second, shared-memory tiling has been applied unevenly across prior works: heavily but for narrow scope [37], lightly across multiple kernels [34], or not at all (relying on library calls) [35, 38]. No prior work applies shared-memory tiling simultaneously to the pressure, saturation, and polymer concentration kernels of a polymer-flooding simulator. Third, the works that achieve the broadest pipeline coverage [34, 37-38] all target either generic IMPES black-oil or multi-phase models without polymer transport. None solves the three-equation polymer-flooding system on the GPU with full-pipeline optimisation.

## 2.6 Identified Literature Gaps

Three concrete gaps in the open literature are identified through the systematic review of Section 2.5. Each is closed by a specific contribution of this dissertation, identified explicitly below.

(1) No full-pipeline shared-memory implementation for polymer flooding. Existing GPU works tile selectively or rely on library calls [32-38]. No published work tiles pressure, saturation, and polymer concentration kernels simultaneously into shared memory for polymer flooding. This gap is addressed by the first scientific

novelty of the present dissertation, developed in Chapter 6 (Section 6.3), which implements the full-pipeline shared-memory CUDA algorithm with bank-conflict-free indexing and validated numerical accuracy below  $10^{-6}$  relative error.

(2) No open multi-GPU implementation of polymer flooding with three-level parallelism. Multi-GPU parallelism has been demonstrated in proprietary commercial simulators, but not in open academic implementations of polymer flooding. No prior work combines distributed memory (MPI), shared memory (OpenMP), and GPU acceleration (CUDA) on the same physics pipeline. This gap is addressed by the second scientific novelty of the present dissertation, developed in Chapter 7 (Section 7.2), which implements the triple-hybrid OMP+MPI+CUDA algorithm and demonstrates peak speedup of  $34.7\times$  over the serial baseline.

(3) No systematic seven-algorithm HPC benchmarking with full profiling validation for polymer flooding. A systematic comparison of all major parallel paradigms (Serial, OpenMP, MPI, Hybrid OMP+MPI, CUDA Global, CUDA Shared, MPI+CUDA, and the triple hybrid) for the same polymer-flooding problem, with corroborating evidence from VTune, mpiP, Nsight Compute, and Nsight Systems, is not available in the open literature. This gap is addressed in Chapter 8, which provides the unified performance dashboard, the validation results across all six grid sizes, and the profiling-tool corroboration of every speedup claim.

## 2.7 Summary

The literature reviewed in this chapter establishes that polymer flooding is a mature and widely deployed enhanced oil recovery technique [4-5, 9], and that its mathematical model, three coupled nonlinear PDEs, is well established and admits efficient numerical solution by the IMPES scheme [19-20] with Picard fixed-point iteration [25]. CPU and GPU parallelisation of reservoir simulators have been investigated extensively [32-38], but a systematic multi-GPU implementation with full-pipeline shared-memory tiling for the complete polymer-flooding system has not been previously reported. The three identified literature gaps are closed by the three principal contributions of this dissertation: the full-pipeline shared-memory CUDA algorithm (Chapter 6), the triple-hybrid OMP+MPI+CUDA multi-GPU algorithm (Chapter 7), and the comprehensive seven-algorithm benchmark with profiling corroboration (Chapter 8). Chapter 3 now presents the mathematical model of the polymer-flooding problem in detail.

### 3 Mathematical Model

This chapter presents the complete mathematical formulation of the polymer-flooding problem in porous media. The governing equations are derived from first principles of mass conservation and Darcy's law [12], extended to the multiphase polymer-flooding system under physically motivated simplifying assumptions [6, 15]. Section 3.1 introduces the modelling assumptions A1 through A7 and discusses their physical justification. Section 3.2 derives the three coupled governing PDEs: the elliptic pressure equation, the hyperbolic saturation equation, and the polymer concentration equation with W-transport. Section 3.3 presents the constitutive relations: Brooks–Corey relative permeability [17], polymer viscosity modification [9, 16], and the Langmuir adsorption isotherm [11]. Section 3.4 examines the coupling structure of the three equations and the iterative strategy required to handle it. Section 3.5 specifies the initial and boundary conditions for the 1D formulation. Section 3.6 assembles the complete coupled system, and Section 3.7 closes with a summary that bridges to the numerical discretisation in Chapter 4.

#### 3.1 Modelling Assumptions

The polymer-flooding problem is modelled here as a 1D two-phase (oil–water) flow system in a porous medium, with polymer dissolved in the water phase. Seven assumptions, denoted A1 through A7, are adopted to define the scope of the mathematical model. Each assumption is consistent with the majority of the reservoir-simulation literature [6, 15] and is defensible on physical grounds for the class of problems considered.

##### 3.1.1 A1: Linear Horizontal Flow in a Constant-Thickness Formation

The reservoir is modelled as a one-dimensional domain along the primary flow direction from the injection well to the production well. This 1D formulation captures the essential physics of the Buckley–Leverett displacement process and is the standard configuration for systematic HPC benchmarking formulations where the formation thickness is much smaller than the horizontal extent. The principal physical consequence is that gravitational segregation between oil and water is suppressed; this is justified for thin formations and for cases where the displacement front advances horizontally on timescales much shorter than the gravitational segregation timescale. Three-dimensional and gravity-segregating extensions are outside the scope of this dissertation and are identified as future work in the Conclusion.

##### 3.1.2 A2: Two Immiscible Phases without Inter-Phase Mass Transfer

The oil and water phases are assumed to be immiscible (no oil dissolves in water, no water dissolves in oil) and no mass transfer occurs between the two phases. This is the classical two-phase incompressible flow model [6, 13-14]. The assumption is excellent for low-pressure, low-temperature reservoirs typical of polymer-flooding applications; it would need to be relaxed for compositional simulation (where light hydrocarbon components can transfer between phases) and for high-temperature reservoirs where the gas-phase fraction becomes significant.

### **3.1.3 A3: Incompressible Fluids and Rock**

Both the oil and water phases are assumed incompressible, with density and viscosity independent of pressure (modulo the polymer-viscosity modification of A6). The rock is also assumed incompressible, with no pore-volume change with pressure. These assumptions are good approximations for liquid-phase reservoirs at the moderate pressures (5 to 30 MPa) typical of polymer-flooding deployments. They are the foundation for the elliptic character of the pressure equation derived in Section 3.2: compressibility would introduce a time derivative  $\partial(\rho\phi)/\partial t$  into the mass conservation equations, converting the elliptic pressure equation into a parabolic one and requiring a different numerical treatment.

### **3.1.4 A4: Darcy Flow Regime**

The flow regime is assumed to be Darcian (the superficial velocity of each phase is proportional to the pressure gradient through Darcy's law [12]), with negligible inertial (Forchheimer) effects [39]. The Forchheimer correction becomes significant only at very high flow velocities, typically encountered near wellbores in high-permeability formations; for the bulk of the reservoir away from wells, Darcy flow is the dominant regime. The Reynolds number based on pore-throat diameter is typically below unity in the reservoir interior, well within the Darcian regime.

### **3.1.5 A5: Negligible Gravity and Capillary Pressure**

Gravitational forces and capillary pressure between the oil and water phases are assumed negligible. The gravity assumption is consistent with A1 (linear horizontal flow); the capillary pressure assumption is the standard Buckley–Leverett simplification [14], which yields a sharp displacement front rather than a smoothly diffused one. In real reservoirs, capillary pressure produces a transition zone of finite thickness at the displacement front, smearing the front over a length scale set by the capillary number. For the moderate-to-high flow velocities typical of polymer flooding, the capillary number is small and the Buckley–Leverett approximation is reasonable. Inclusion of capillary pressure would add a diffusive term to the saturation equation and modify the fractional flow function, but it would not change the algorithmic structure developed in Chapter 4.

### **3.1.6 A6: Uniform Initial Water Saturation at Irreducible Saturation**

The reservoir is assumed initially saturated with oil at the connate (irreducible) water saturation  $Swc$ , typically 0.1 to 0.3 for water-wet sandstones. This represents the post-primary-recovery state of the reservoir, before the start of secondary waterflooding or tertiary polymer flooding. The initial polymer concentration is zero everywhere. These initial conditions are standard for polymer-flooding simulation studies and produce displacement fronts that can be compared directly against the Buckley–Leverett analytical solution at zero polymer concentration.

### **3.1.7 A7: Isothermal Conditions**

The reservoir is assumed isothermal, meaning temperature is constant in space and time throughout the simulation. This is a reasonable approximation for shallow-to-moderate-depth reservoirs (less than 2,000 metres) where the geothermal gradient

produces only modest temperature variations within the producing zone, and where injection-fluid temperature is matched to reservoir temperature. Non-isothermal polymer flooding (relevant for deep reservoirs with substantial geothermal gradients, or for thermally enhanced recovery processes) is outside the present scope and is identified as future work in Conclusions.

### 3.2 Governing Equations

Under assumptions A1 through A7, the governing equations for the polymer-flooding problem are derived from mass conservation for each phase and each species, combined with Darcy's law for phase velocities [12]. The derivation follows the classical approach [6, 15], adapted to include the polymer-transport physics of Sorbie [9].

#### 3.2.1 Darcy's Law and Phase Velocities

The positive phase mobilities are defined as

$$\lambda_w(S_w, C_p) = \frac{k k_{rw}^{eff}(S_w, C_p)}{\mu_w(C_p)}, \quad \lambda_o(S_w) = \frac{k k_{ro}(S_w)}{\mu_o}. \quad (3.1)$$

The superficial Darcy velocity of each fluid phase  $\alpha \in \{w, o\}$  is related to the pressure gradient through the phase mobility:

$$V_w = -\lambda_w(S_w, C_p) \nabla P_a, \quad (3.2)$$

$$V_o = -\lambda_o(S_w) \nabla P_a. \quad (3.3)$$

where  $V_w, V_o$  is the superficial Darcy velocity of phase water and oil ( $m/s$ ),  $\lambda_w, \lambda_o$  is the water and oil phase mobility ( $m^2/Pa \cdot s$ ),  $k$  is the absolute permeability of the porous medium ( $m^2$ ),  $k_{rw}, k_{ro}$  is the relative permeability of phase water and oil (dimensionless),  $\mu_w, \mu_o$  is the dynamic viscosity of phase water and oil ( $Pa \cdot s$ ), and  $P_a$  is the pressure of phase  $\alpha$  ( $Pa$ ). Under assumption A5 (negligible capillary pressure), the two phase pressures are equal:  $P_o = P_w = P$ . The total Darcy velocity is the sum  $v_t = v_w + v_o = -\lambda_t(S, C_p) \nabla P$ , where  $\lambda_t = \lambda_w + \lambda_o$  is the total mobility, a function of saturation through the relative permeabilities and of polymer concentration through the polymer-modified water viscosity.

#### 3.2.2 Pressure Equation (Elliptic)

Mass conservation for each phase, combined with the incompressibility assumption A3, yields  $\nabla \cdot \lambda_w = 0$  and  $\nabla \cdot \lambda_o = 0$  at every point in the reservoir away from wells. Summing the two equations and substituting Darcy's law (Eq. 3.1) yields the pressure equation:

$$-\nabla \cdot (\lambda_t(S, C_p) \nabla P) = q_t, \quad (3.4)$$

where  $q_t$  is the total volumetric source/sink term at wells (in units of  $1/s$ , representing fluid injected or produced per unit reservoir volume). Away from wells,  $q_t = 0$  and the pressure equation reduces to the homogeneous elliptic form. The elliptic character of the equation has two important consequences. First, the pressure field at any point depends on the boundary conditions at all parts of the boundary: a perturbation at one well affects the pressure everywhere in the reservoir instantaneously (under the incompressibility assumption). Second, the equation

requires implicit time-stepping for stability; explicit methods would impose an  $O(\Delta x^2)$  time-step restriction that is impractical for the large grid sizes considered in this dissertation.

### 3.2.3 Saturation Equation (Hyperbolic)

Mass conservation for the water phase, accounting for the saturation constraint  $S_o + S_w = 1$ , yields the hyperbolic saturation equation:

$$\phi \frac{\partial S}{\partial t} + \nabla \cdot (k_{rw}^{eff}(S, C_p) \cdot v_t) = q_w, \quad (3.5)$$

Where  $\phi$  is the water saturation (dimensionless),  $\phi$  is the porosity of the rock (dimensionless),  $q_w$  is the water-phase source/sink at wells, and  $k_{rw}$  is the water fractional flow function [13-14]. The fractional flow function is the central object of Buckley–Leverett displacement theory: it represents the fraction of the total flux that is carried by the water phase at a given saturation. Its characteristic S-shape (concave-down at low saturation, concave-up at high saturation, with an inflection point near the centre of the saturation range) produces the sharp displacement fronts observed in two-phase flow. The hyperbolic character of Eq. 3.5 makes it amenable to explicit time-stepping under the CFL condition [22].

### 3.2.4 Polymer Concentration Equation

Conservation of the polymer species in the water phase, accounting for advection at the water-phase velocity, source/sink terms at injection wells, and adsorption onto the rock surface [9], yields:

$$\phi \frac{\partial}{\partial t} (S_w C_p) + \rho_r (1 - \phi) \frac{\partial W}{\partial t} + \nabla \cdot (v_w C_p) = q_p, \quad (3.6)$$

where  $C_p$  is the dissolved polymer concentration in the water phase (kg/m<sup>3</sup> of water),  $W$  is the adsorbed polymer concentration on the rock surface (kg/kg of rock),  $\rho_r$  is the rock density (kg/m<sup>3</sup>), and  $q_p$  is the polymer-injection source term. In our formulation, polymer injection is enforced through a Dirichlet inlet boundary condition  $C_p(0, t) = C_{p\_inj}$ . Therefore the source term  $q_p$  is identically zero throughout the domain interior: the governing PDE is source-free, and all polymer enters the system through the left boundary flux term  $\nabla \cdot (v_w C_p)$ . The three terms on the left-hand side represent, respectively: the local rate of change of dissolved polymer mass in the pore volume; the divergence of the advective polymer flux carried by the water phase; and the rate of polymer mass transfer from solution to the adsorbed phase on the rock surface.

Under the modelling assumptions A1 through A7, and for the interior of the simulation domain away from wells, the source/sink terms in equations (3.4), (3.5), and (3.6) vanish identically:

$q_t = 0$ ,  $q_w = 0$ , and  $q_p = 0$ . Each term vanishes for the following reason. The total source term  $q_t$  vanishes as a direct consequence of assumption A3 (incompressible fluids and rock): mass conservation for each phase yields  $\nabla \cdot v_w = 0$  and  $\nabla \cdot v_o = 0$  in the interior, so their sum gives  $\nabla \cdot v_t = 0$ , which is equivalent to  $q_t = 0$ . The water-phase source term  $q_w = 0$  and the polymer source term  $q_p$  vanish because all fluid injection and polymer delivery are enforced through Dirichlet boundary conditions at

the injection boundary: specifically  $S(0, t) = 1$  and  $C_p(0, t) = C_{p\_inj}$ , rather than through interior volumetric sources. Production is likewise handled through the Dirichlet pressure condition  $P(L, t) = P_{prod}$  and Neumann outflow conditions  $\frac{\partial S}{\partial x} = 0$  and  $\frac{\partial C_p}{\partial x} = 0$  at the production boundary (Section 3.5). This boundary-condition treatment is consistent with the Buckley–Leverett framework (assumption A5), in which wells appear as boundary terms rather than distributed sources, and all three governing equations reduce to their homogeneous interior forms.

### 3.3 Constitutive Relations

#### 3.3.1 Brooks–Corey Relative Permeability

The relative permeability functions are expressed as power laws in the normalised water saturation [17]:

$$k_{rw}^{eff}(S_w, C_p) = \frac{S_w^n}{R_k(C_p)}, \quad k_{ro}(S_w) = (1 - S_w)^n, \quad (3.7)$$

where  $n$  are the Corey exponents, dimensionless parameters that depend on the pore-throat geometry and the wettability of the rock. For water-wet sandstones, the dominant reservoir type for polymer flooding, typical values are  $n$  in the range 2 to 4. The power-law form has three important properties: (i)  $k_{rw}$  vanishes at  $S = 0$  (no water flow when no water is present) and reaches unity at  $S = 1$  (full water flow at full water saturation); (ii)  $k_{ro}$  vanishes at  $S = 1$  (no oil flow when no oil is present) and reaches unity at  $S = 0$  (full oil flow at full oil saturation); (iii) the derivatives  $\frac{\partial k_{rw}}{\partial S}$  and  $\frac{\partial k_{ro}}{\partial S}$  are continuous everywhere in  $(0, 1)$ , avoiding artificial discontinuities that would complicate the numerical treatment. The permeability reduction factor  $R_k(C_p)$  in the denominator of  $k_{rw}^{eff}$  is a function of polymer concentration alone and is defined in the following section.

#### 3.3.2 Permeability Reduction Factor

When polymer adsorbs onto the pore-wall surface, it physically narrows the water-phase flow channels, reducing the effective permeability to water below its polymer-free value. This mechanism is captured by the permeability reduction factor  $R_k(C_p) \geq 1$ , which divides the polymer-free water relative permeability  $S_w^n$  in equation (3.7). The permeability reduction factor follows a Langmuir-type saturation curve [9]:

$$R_k(C_p) = 1 + \frac{(R_{max}-1) \cdot b_k \cdot C_p}{(1+b_k \cdot C_p)}, \quad (3.8)$$

where  $R_{max}$  is the maximum permeability reduction factor (dimensionless,  $R_{max} \geq 1$ ), representing the limiting value of  $R_k$  as  $C_p \rightarrow \infty$  when all adsorption sites are occupied; and  $b_k$  is the Langmuir adsorption coefficient ( $\text{m}^3/\text{kg}$ ), controlling how steeply  $R_k$  rises with polymer concentration.

The factor  $R_k(C_p)$  has three important limiting properties that are consistent with physical expectations: (i) At  $C_p = 0$ :  $R_k = 1 + 0 = 1$  (no permeability reduction in the absence of polymer), recovering the standard Brooks–Corey water relative

permeability  $S_w^n$ . (ii) As  $C_p \rightarrow \infty$ :  $R_k \rightarrow R_{max}$ : the permeability reduction saturates at  $R_{max}$ , reflecting the finite number of adsorption sites available on the pore surface (monolayer saturation). (iii)  $R_k$  is monotonically increasing in  $C_p$  with  $\frac{\partial R_k}{\partial C_p} > 0$  everywhere, meaning greater polymer concentration always produces greater permeability reduction, never less.

The mathematical form of  $R_k(C_p)$  is identical in structure to the Langmuir adsorption isotherm  $W(C_p)$  of Section 3.3.4, reflecting the common physical origin of both phenomena in monolayer surface adsorption. The permeability reduction factor affects only the water-phase relative permeability  $k_{rw}^{eff}(S_w, C_p)$ ; the oil-phase relative permeability  $k_{ro}(S_w)$  is unaffected, since the oil phase occupies separate pore throats and does not contact the adsorbed polymer layer. For typical HPAM polymer floods,  $R_{max}$  lies in the range 1.5 to 8 and  $b_k$  lies in the range 0.001 to 0.1 m<sup>3</sup>/kg [9].

### 3.3.3 Polymer Viscosity Modification

Alongside the permeability reduction mechanism of Section 3.3.2, polymer also increases the water-phase viscosity, the second mechanism by which polymer flooding improves the mobility ratio and sweeps oil more efficiently.

The water-phase viscosity in the presence of polymer is modified through a power-law model derived from empirical observations [9, 16]:

$$\mu_w = \mu_{w0} \cdot (1 + a_p \cdot C_p^{np}) \quad (3.9)$$

where  $\mu_{w0}$  is the polymer-free water viscosity (typically 1 cP at reservoir conditions),  $a_p$  is the polymer viscosity coefficient (kg/m<sup>3</sup> <sup>-np</sup>), and  $np$  is the polymer viscosity exponent (typically 1 to 2 for HPAM). Equation 3.9 captures the dominant rheological behaviour of HPAM solutions: at zero polymer concentration,  $\mu_w$  reduces to  $\mu_{w0}$ ; at modest polymer concentrations (500 to 2,500 ppm typical of field deployments),  $\mu_w$  can rise to 10 to 50 times  $\mu_{w0}$ , bringing the water-phase viscosity into the same range as the resident crude-oil viscosity. The power-law form is computationally cheap to evaluate (a single power and two multiplications per cell), making it well-suited to the per-cell mobility-update kernel of Chapter 4.

### 3.3.4 Langmuir Adsorption Isotherm

The adsorbed polymer concentration is related to the dissolved concentration through the Langmuir isotherm [11]:

$$W(C_p) = (a_l \cdot b_l \cdot C_p) / (1 + b_l \cdot C_p) \quad (3.10a)$$

where  $a_l$  is the maximum adsorption capacity (kg of polymer adsorbed per kg of rock when all adsorption sites are occupied) and  $b_l$  is the Langmuir equilibrium constant (m<sup>3</sup>/kg). The Langmuir isotherm has three important properties: (i) at zero dissolved concentration  $W = 0$  (no adsorption without polymer); (ii) as  $C_p \rightarrow \infty$ ,  $W \rightarrow a_l$  (saturation of the monolayer); (iii) at low concentrations  $W$  is linear in  $C_p$  with slope  $a_l b_l$  (Henry's law regime). The transition between the linear and saturated regimes occurs near  $C_p \approx 1/b_l$ , which sets the characteristic concentration scale of the adsorption phenomenon. For HPAM on water-wet sandstones, typical values are  $a_l \approx 30$  to 100  $\mu$ g per gram of rock and  $b_l \approx 0.001$  to 0.01 m<sup>3</sup>/kg. In the

present performance study, the linear low-concentration approximation of Equation (3.10) is used. When  $b_l \cdot C_p \ll 1$ , the Langmuir isotherm reduce to

$$W(C_p) \approx a_l b_l C_p = A_k C_p, \quad (3.10b)$$

where  $A_k$  is the linear adsorption coefficient. This approximation avoids a nonlinear local inversion for  $C_p$  at every grid cell and time step.

### 3.3.5 W-Transport Relation

Under the assumption that the adsorbed phase is in local equilibrium with the dissolved phase, the time derivative of  $W$  is related to the time derivative of  $C_p$  through the chain rule:

$$\frac{\partial W}{\partial t} = \left( \frac{dW}{dC_p} \right) \cdot \frac{\partial C_p}{\partial t} = \gamma_1(C_p) \cdot \frac{\partial C_p}{\partial t} \quad (3.11)$$

where  $\gamma_1(C_p) = \frac{dW}{dC_p}$  is the local slope of the Langmuir isotherm. This relation, combined with Eq. 3.6, allows the W-transport to be folded into the polymer-concentration update at every time step, a numerical strategy implemented as the nested Picard loop of Algorithm 4 in Chapter 4. The local-equilibrium assumption is justified for the slow flow velocities and large characteristic timescales of reservoir flow, where the adsorption kinetics are much faster than the advective timescales.

## 3.4 Coupling Structure of the Three Equations

The three governing equations (3.4), (3.5), and (3.6) are strongly coupled through the constitutive relations: the pressure equation depends on  $S$  and  $C_p$  through  $\lambda_t(S, C_p)$ ; the saturation equation depends on  $P$  and  $C_p$  through the fractional flow function (which contains  $\mu_w(C_p)$ ) and the total velocity; the polymer concentration equation depends on  $S, P$ , and the adsorbed concentration  $W$ . This three-way coupling has two principal numerical consequences. First, the system is nonlinear and cannot be solved by a single linear solve at each time step. Second, the equations cannot be decoupled cleanly: any approach that updates each equation independently must accept some lag in the coupling and may introduce errors that accumulate over time.

The IMPES scheme adopted in this dissertation addresses these challenges by lagging the mobility coefficients  $\lambda_t(S, C_p)$  by one time step in the pressure solve, then updating  $S$  and  $C_p$  explicitly using the new pressure field. This approach decouples the implicit pressure solve from the explicit saturation and polymer concentration updates, reducing the cost per time step from a fully implicit nonlinear solve to a single linear solve (for pressure) plus two explicit updates (for saturation and polymer concentration). The cost is the CFL constraint on the time step, which bounds  $\Delta t$  to the smallest cell width divided by the maximum velocity. For the moderate grid resolutions and flow velocities of reservoir simulation, the CFL constraint typically permits time steps of order one day, well within the simulation horizon.

Within the pressure solve, the nonlinearity of is resolved through outer Picard fixed-point iteration [25]: the pressure equation is solved with the current estimate of, the new pressure produces a new velocity field, which in turn updates the saturation and polymer concentration, which updates, and the cycle repeats until convergence. The Picard iteration is provably convergent provided the mobility nonlinearity is not

too strong; in practice, three to five Picard sweeps per time step are sufficient to drive the residual below the convergence tolerance, with each sweep involving only a single linear-system solve. The inner linear solver in this dissertation is the Jacobi iterative method [21], whose embarrassingly parallel structure makes it well-suited to all four levels of the parallelisation hierarchy.

### 3.5 Initial and Boundary Conditions: 1D Formulation

#### 3.5.1 Initial Conditions

The simulation is initialised at time  $t = 0$  to represent the post-primary-recovery state of the reservoir, before the commencement of polymer flooding. The initial conditions for the three primary unknowns (pressure  $P$ , water saturation  $S$ , and polymer concentration  $C_p$ ) are prescribed as spatially distributed fields over the domain  $\Omega = [0, L]$ :

$$S|_{t=0} = S_0, \quad P|_{t=0} = P_0, \quad C_p|_{t=0} = C_{p0}; \quad (3.12)$$

In the present work, consistent with assumption A6 (uniform initial water saturation at irreducible saturation), the three initial fields are taken as spatially uniform:

$S_0(x) = S_{wc}$ , the connate (irreducible) water saturation, uniform throughout the reservoir. This represents the state of the reservoir at the end of primary recovery, where only the minimum irreducible water saturation remains in the pore space and the remainder is occupied by oil. Typical values of  $S_{wc}$  for water-wet sandstones lie in the range 0.1 to 0.3.

$P_0(x)$ , the initial pressure field is not prescribed as a constant but is instead computed from the steady-state Darcy solution at  $t = 0$ , with uniform total mobility  $\lambda_t$  evaluated at  $S = S_{wc}$  and  $C_p = 0$ . This ensures that the initial velocity field is consistent with the applied boundary pressures  $P_{inj}$  and  $P_{prod}$  before any displacement has occurred, avoiding a spurious transient at the first time step.

$C_{p0}(x) = 0$ : *the initial polymer concentration is zero* everywhere in the domain, reflecting the absence of polymer prior to the start of injection. Consequently the initial adsorbed concentration is also zero:  $W(x, 0) = W(C_{p0}) = 0$ , as no adsorption can occur without dissolved polymer.

These three initial conditions, together with the boundary conditions of Section 3.5.2, fully determine the solution of the coupled system (3.4)–(3.6) for all  $t > 0$ .

#### 3.5.2 Boundary Conditions

The boundary conditions for the 1D formulation are prescribed on the two endpoints of the domain  $\Omega = [0, L]$ , representing the injection well at  $x = 0$  and the production well at  $x = L$ . The conditions are chosen to reflect the physical operation of a single-well-pair polymer flood: polymer-laden water is injected at a fixed pressure at  $x = 0$ , and fluid is produced at a fixed bottom-hole pressure at  $x = L$ . All fluid exchange with the reservoir is captured through these boundary conditions; no interior source or sink terms are present ( $q_t = q_w = q_p = 0$ ), as established in Section 3.2.

##### (1) Pressure Boundary Conditions

Fixed Dirichlet conditions are applied at both boundaries, prescribing the well pressures:

$$P|_{x=0} = P_{inj}, \quad P|_{x=L} = P_{prod}. \quad (3.13)$$

where  $P_{inj} > P_{prod}$  so that the pressure drop  $\Delta P = P_{inj} - P_{prod}$  drives Darcy flow from the injector to the producer throughout the simulation. In the numerical implementation, these conditions are enforced directly by fixing the first and last elements of the pressure array:

$p[0] = P_{inj}$ ,  $p[N-1] = P_{prod}$  and updating only the interior cells  $i = 1, \dots, N-2$  during the Picard–Jacobi pressure solve. The two boundary values remain fixed at their prescribed well pressures at every time step and every Picard iteration, ensuring that the pressure field always satisfies the correct physical driving conditions.

## (2) Saturation Boundary Conditions

At the injection boundary, a Dirichlet condition specifies that pure water is injected: the injected water saturation is unity:

$$S|_{x=0} = S_{inj} = 1, \quad (3.14)$$

This reflects the physical reality that the injected fluid is entirely water (carrying dissolved polymer), with no oil component entering the reservoir from the injection well.

At the production boundary, a Neumann zero-gradient (outflow) condition is applied:

$$\frac{\partial S}{\partial x}|_{x=L} = 0, \quad (3.15)$$

This condition states that the saturation gradient vanishes at the producer: the produced fluid composition is whatever the displacement front delivers to the production boundary at the current time step, with no artificial reflection or prescription of the outflow saturation. This is the standard outflow condition for hyperbolic transport equations and ensures that the displacement front exits the domain cleanly without generating spurious reflections.

## (3) Polymer Concentration Boundary Conditions

At the injection boundary, a Dirichlet condition prescribes the concentration of polymer in the injected water:

$$C_p|_{x=0} = C_{p\_inj}, \quad (3.16)$$

where  $C_{p\_inj}$  is the prescribed injection polymer concentration ( $\text{kg}/\text{m}^3$ ). This condition, together with the saturation condition  $S|_{x=0} = 1$ , fully specifies the injected fluid state at the inlet. The polymer enters the reservoir exclusively through this boundary condition; there is no interior polymer source ( $q_p = 0$  as established in Section 3.2.4).

At the production boundary, a Neumann zero-gradient outflow condition is applied, consistent with the saturation treatment:

$$\frac{\partial C_p}{\partial x}|_{x=L} = 0, \quad (3.17)$$

This condition allows the polymer concentration front to exit the domain freely at whatever concentration the displacement process has delivered to the producer, without prescribing or constraining the produced polymer concentration.

The complete set of boundary conditions for the 1D formulation is collected in Table 3.1 for reference.

Table 3.1 – Boundary conditions for the 1D polymer flooding formulation

| Variable | $x=0(\text{injection})$   | $x=L(\text{production})$                     | Type                  |
|----------|---------------------------|----------------------------------------------|-----------------------|
| P        | $P _{x=0} = P_{inj}$      | $P _{x=L} = P_{prod}$                        | Dirichlet – Dirichlet |
| S        | $S _{x=0} = 1$            | $\frac{\partial S}{\partial x} _{x=L} = 0$   | Dirichlet – Neumann   |
| Cp       | $C_p _{x=0} = C_{p\_inj}$ | $\frac{\partial C_p}{\partial x} _{x=L} = 0$ | Dirichlet – Neumann   |

The well-cell treatment follows the Peaceman well model [15], in which each well is represented as a point source or sink distributed over the single grid cell containing the well location. For the 1D domain considered in this dissertation, the well-cell model reduces to the simple Dirichlet injection boundary condition of Section 3.5.2 when and the no-flow edges collapse: the injection well-cell becomes the left boundary, the production well-cell becomes the right boundary, and the no-flow top and bottom edges vanish. This consistency between the 1D formulations provides an important verification pathway: the simulator must reproduce the 1D Buckley–Leverett solution exactly when the domain is reduced to a single row of cells, which serves as one of the validation tests reported in Chapter 5.

### 3.6 Complete Coupled System

Collecting Equations (3.4) through (3.11), the complete mathematical model for polymer flooding solved in this dissertation comprises the coupled system of three nonlinear partial differential equations:

(1) Pressure equation (elliptic).  $-\nabla \cdot (M(S, C_p) \nabla P) = q_t$ , in  $\Omega$ , with Dirichlet boundary conditions at injection and production wells and Neumann no-flow conditions on the remaining boundary.

(2) Saturation equation (hyperbolic).  $\phi \frac{\partial S}{\partial t} + \nabla \cdot (k_{rw}^{eff}(S, C_p) \cdot v_t) = q_w$  in  $\Omega \times (0, T]$ , with Dirichlet condition  $S = 1$  at injection well and Neumann outflow at production well.

(3) Polymer concentration equation (hyperbolic, coupled with W).  $\phi \frac{\partial}{\partial t} (S_w C_p) + \rho_r (1 - \phi) \frac{\partial W}{\partial t} + \nabla \cdot (v_w C_p) = q_p$ , with Dirichlet condition  $C_p = C_{p\_inj}$  at injection well and Neumann outflow at production well, and  $W$  coupled to  $C_p$  through Eq. 3.10.

supplemented by the constitutive relations: Brooks–Corey relative permeability (Eq. 3.7), polymer viscosity modification (Eq. 3.9), Langmuir adsorption isotherm (Eq. 3.10), and W-transport relation (Eq. 3.11). Initial and boundary conditions are specified in Section 3.5 respectively. The numerical solution of this coupled system is the subject of Chapter 4.

### 3.7 Summary

This chapter has presented the complete mathematical model of the polymer-flooding problem: three coupled nonlinear PDEs governing the evolution of the

pressure, water saturation, and polymer concentration fields in a porous medium, supplemented by the Brooks–Corey relative permeability model [17], a power-law polymer viscosity modification [9], the Langmuir adsorption isotherm [11], and the W-transport relation. Seven physically motivated assumptions (A1 through A7) define the scope of the model. The three-way coupling between the equations, mediated through the total mobility, the fractional flow function, and the polymer-adsorption isotherm, motivates the IMPES time-stepping scheme [19-20] adopted in Chapter 4, with outer Picard fixed-point iteration [25] resolving the mobility nonlinearity at every time step. Both 1D formulations have been specified with appropriate initial and boundary conditions. Chapter 4 now develops the finite-difference discretisation, the adaptive CFL time-step selection [22], and the complete sequential algorithm.

## 4 Numerical Methods and Sequential Algorithm

This chapter develops the complete numerical framework for solving the mathematical model of Chapter 3. Section 4.1 develops the finite-difference spatial discretisation on the uniform 1D structured Cartesian grid, including the arithmetic averaging at cell interfaces and the first-order upwind flux-splitting treatment of hyperbolic transport [6, 23-24]. Section 4.2 presents the IMPES time-stepping scheme [19-20] and its decomposition of each time step into sequential stages. Section 4.3 derives the CFL stability condition [22] and the adaptive time-step selection algorithm. Sections 4.4 through 4.7 develop the four sub-algorithms of the sequential simulator in detailed pseudocode: the implicit pressure solver with Picard–Jacobi iteration, the CFL time-step controller, the explicit upwind saturation solver, and the polymer transport solver with W-storage and explicit concentration recovery. Section 4.8 assembles the complete coupled Picard–IMPES pipeline. Section 4.9 presents the numerical validation of the serial baseline. Section 4.10 reports the serial baseline performance across all six grid sizes.

### 4.1 Spatial Discretisation

#### 4.1.1 1D Structured Cartesian Grid

The 1D spatial domain is uniformly partitioned into  $N$  equal subintervals with grid spacing. The grid points are located at  $x_i$  for  $i = 1, \dots, N$ , so that  $x_1$  corresponds to the injection boundary and  $x_N$  to the production boundary. Boundary conditions are imposed directly at the endpoints  $x_1$  and  $x_N$ ; interior updates are applied at  $x_i$  for  $i = 2, \dots, N-1$ . The structured Cartesian grid produces regular, unit-stride memory access patterns that are optimal both for CPU cache utilisation and for GPU coalesced memory transactions [28], and it allows all parallelisation strategies (OpenMP, MPI, CUDA) to map directly from cell index to thread or process identifier without complex load-balancing logic.

#### 4.1.2 Finite-Difference Discretisation

The finite-difference method approximates the differential form of the governing PDEs at each grid point by replacing spatial derivatives with algebraic difference quotients. For the pressure equation, applying second-order central differences to the cell  $i$  and application of the divergence theorem yields:

$$-\left( \lambda_{t,(i+1/2)} \cdot \left( \frac{p_{i+1}^{n+1} - p_i^{n+1}}{\Delta x^2} \right) - \lambda_{t,(i-1/2)} \cdot \left( \frac{p_i^{n+1} - p_{i-1}^{n+1}}{\Delta x^2} \right) \right) = q_{t,i}, \quad (4.0)$$

where  $\lambda_{t,(i\pm 1/2)}$  is the transmissibility at the face between cells  $i$  and  $i + 1$ . The finite-difference formulation on a uniform Cartesian grid has two essential properties: (i) numerical mass conservation is verified post hoc at every time step to machine epsilon, confirming that accumulated truncation errors do not introduce spurious mass sources; the net mass entering the domain through the injection boundary  $i$  through one face exactly equals the mass leaving through the opposite face, modulo source terms; (ii) it accommodates discontinuous coefficients (such as permeability jumps at facies boundaries) through arithmetic averaging of cell-centred mobilities, as developed in Section 4.1.3. This yields the standard three-point tridiagonal system.

### 4.1.3 Arithmetic Averaging at Cell Interfaces

The total mobility at each cell interface is approximated by the arithmetic mean of the mobilities of the two adjacent nodes, consistent with the central-difference discretisation of the pressure equation on a uniform grid [24]:

$$\lambda_{t,(i+1/2)} = (\lambda_{t,(i)} + \lambda_{t,(i+1)}) / 2 \text{ and } \lambda_{t,(i-1/2)} = (\lambda_{t,(i)} + \lambda_{t,(i-1)}) / 2 \quad (4.1)$$

This is the standard interface approximation for the discretisation of the elliptic pressure equation on a uniform structured grid [6]. For a uniform Cartesian mesh with smooth mobility fields (as assumed throughout this dissertation under the Buckley–Leverett framework), arithmetic averaging produces a second-order accurate approximation of the interface transmissibility and is consistent with the central-difference discretisation of the pressure equation adopted in this work.

Substituting Eq. (4.1) into the pressure equation discretisation (Eq. 4.0) yields a three-point linear system whose coefficient matrix is symmetric and positive semi-definite. The fixed Dirichlet pressure conditions at both boundary nodes ( and ) remove the zero eigenvalue and render the reduced interior system symmetric positive definite. The two boundary rows are strictly diagonally dominant, and strict dominance propagates through the irreducible tridiagonal structure to all interior rows [21], guaranteeing convergence of the Jacobi iterative solver used in Algorithm 2 (Section 4.4) for any initial pressure guess, including the uniform initialisation adopted in Section 3.5.1. In the present 1D implementation this yields the left and right interface mobilities required for the tridiagonal pressure system.

### 4.1.4 First-Order Upwind Flux Splitting for Hyperbolic Transport

The water saturation equation (Eq. 3.5) and the polymer concentration equation (Eq. 3.6) are hyperbolic conservation laws that require upwind-biased discretisation to ensure positivity and monotonicity of the numerical solution [24]. Centred differences would produce oscillatory solutions near sharp displacement fronts, and would violate the maximum principle (the saturation could exceed unity or become negative). The first-order donor-cell (Godunov) upwind scheme [23] selects the flux value from the upstream cell at each interface, where upstream is determined by the sign of the local Darcy velocity:

$$F_{i+1/2} = F_i \text{ if } u_{i+1/2} \geq 0, \quad F_{i+1/2} = F_{i+1} \text{ if } u_{i+1/2} < 0, \quad (4.3)$$

In implementation, the upwind selection is achieved branch-free using the algebraic identity:

$$F_{up} = 1/2 [(|u| + u)F_{left} + (|u| - u)F_{right}] / |u| \quad (4.4)$$

which evaluates to when and to when without a conditional branch. The branch-free form is critical for GPU efficiency: divergent branches within a warp serialise execution and reduce effective throughput [28]. On modern CPUs the same identity benefits from elimination of branch-mispredictions, although the effect is smaller. In the present 1D formulation the identity is applied at the left and right cell faces, producing two upwind flux evaluations per interior cell.

## 4.2 The IMPES Time-Stepping Scheme

### 4.2.1 Coupled Picard–IMPES Structure

The governing system (Equations 3.4–3.6) is solved using the Implicit-Pressure–Explicit-Saturation (IMPES) framework [19–20] embedded in a coupled Picard fixed-point iteration [25]. At each time step, the three coupled fields,  $p$ ,  $W$ , and  $C_p$  are advanced by the following sequential stages:

- Solve the implicit pressure field by Jacobi iteration (Algorithm 2, Section 4.4).
- Compute the CFL-stable time increment (Algorithm 1, Section 4.3).
- Update the saturation field explicitly using Algorithm 3 (Section 4.5).
- Update the polymer mass  $W$  and concentration  $C_p$  using Algorithm 4 (Section 4.6).

The outer Picard loop repeats these stages until the pressure, polymer mass, and polymer concentration each satisfy their prescribed convergence tolerance  $\varepsilon$ . The inner loops advance the polymer mass  $W$  and check the residual  $eW$ ; after  $W$  converges the polymer concentration is recovered and the concentration residual  $ecp$  is checked.

### 4.2.2 Advantages and Limitations

The principal advantage of IMPES over the fully implicit method is computational efficiency: the implicit stage involves only the pressure scalar field, reducing the per-step cost by roughly a factor of three compared with a simultaneous solve for all three unknowns. The explicit stages require only matrix-vector operations rather than linear solves. The drawback is the conditional CFL stability constraint on the explicit saturation and polymer mass transport updates, derived in Section 4.3. For the moderate flow velocities of polymer flooding, the CFL constraint typically permits time steps of order  $\Delta$ , well within the simulation horizon, making IMPES the dominant time-stepping scheme in academic reservoir simulation [6, 15] and a frequent choice in commercial simulators where CPU time per step matters more than maximum-stable-step considerations.

## 4.3 CFL Stability Condition and Adaptive Time-Step Selection

### 4.3.1 Derivation of the CFL Condition

The explicit saturation and polymer mass update is conditionally stable: the time-step size  $\Delta t$  must be small enough that information cannot propagate across more than one cell width per step. This is the Courant–Friedrichs–Lewy (CFL) condition, originally derived by Courant, Friedrichs, and Lewy in 1928 [22] as a necessary condition for the convergence of finite-difference approximations to hyperbolic PDEs. The CFL number at time level

is defined as:

$$CFL^n = \tau^n \cdot \frac{U_{max}^n}{\phi \cdot \Delta x}, \quad (4.5)$$

The explicit update satisfies the CFL stability restriction when:

$$CFL^n \leq 1, \quad (4.6)$$

The maximum characteristic velocity over all interior grid nodes is:

$$U_{max}^n = \max_i \left| k \cdot \frac{k_{rw}^{eff}(S_{w,i}^n, C_{p,i}^n)}{\mu_w(C_{p,i}^n)} \cdot \frac{(P_{i+1}^{n+1} - P_i^{n+1})}{\Delta x} \right|, \quad (4.7)$$

In practice, the time step is selected using a safety factor  $C_{CFL} \leq 1$  and a maximum allowed step  $\tau_{max}$ :

$$\tau^n = \min\left(\tau_{max}, C_{CFL} \cdot \frac{\phi \cdot \Delta x}{U_{max}^n}\right), \quad (4.8)$$

This guarantees  $CFL^n \leq C_{CFL} \leq 1$  at every time step. For the physical parameters of Table 4.2 ( $k = 10^{-3}$ ,  $\mu_w = 0.09$ ,  $\phi = 0.20$ ,  $C_{CFL} = 0.5$ ,  $\tau_{max} = 10^{-4}$ ), the CFL constraint typically permits time steps of order  $10^{-4}$  at all six grid sizes evaluated in this dissertation.

### 4.3.2 Adaptive Time-Step Algorithm

Because the velocity field changes from one time step to the next as the pressure and saturation fields evolve, the maximum stable time step must be recomputed at every iteration. The adaptive CFL algorithm presented below:

#### Algorithm 1: CFL-Compliant Time-Step Control

Input: face velocity  $u$ , grid spacing  $dx$ , maximum time step  $dt\_max$ , safety factor  $c\_CFL$

Output: stable time step  $dt$

1. Compute the maximum absolute face velocity over the computational domain.
2. If the maximum velocity is zero, set  $dt = dt\_max$ .
3. Otherwise compute the CFL-limited step  $dt\_CFL = c\_CFL * dx / \max(|u|)$ .
4. Set  $dt = \min(dt\_CFL, dt\_max)$ .
5. Return  $dt$ .

### 4.4 Algorithm 2: Implicit Pressure Solver (Picard–Jacobi)

The pressure equation (Eq. 3.4) is discretised using the arithmetic-averaged interface mobilities of Eq. (4.1). At time level  $n + 1$ , a second-order central-difference approximation of the continuous pressure equation  $\frac{\partial}{\partial x} \left[ \lambda_t \frac{\partial P}{\partial x} \right] = 0$  yields the discrete pressure balance at interior node  $i$ :

$$\left( \lambda_{t,(i+1/2)} \cdot \left( \frac{P_{i+1}^{n+1} - P_i^{n+1}}{\Delta x^2} \right) - \lambda_{t,(i-1/2)} \cdot \left( \frac{P_i^{n+1} - P_{i-1}^{n+1}}{\Delta x^2} \right) \right) = 0, \quad i = 1, \dots, N - 1 \quad (4.9)$$

where  $\lambda_t$  are the arithmetic-mean interface total mobilities, with  $\lambda_t$  evaluated from the current saturation and polymer fields [22]. Isolating  $P_i^{n+1}$  on the left-hand side of Eq. (4.9) gives the Jacobi update formula applied at every interior node:

$$P_i^{k+1} = \left( \lambda_{t,i+1/2} \cdot P_{i+1}^k + \lambda_{t,i-1/2} \cdot P_{i-1}^k \right) / \left( \lambda_{t,i+1/2} + \lambda_{t,i-1/2} \right), \quad i = 1, \dots, N - 1 \quad (4.10)$$

Eq. (4.9) expresses the updated pressure as a mobility-weighted average of its two neighbours. When the mobilities are equal on both sides the formula reduces to the arithmetic mean, recovering the standard discrete Laplacian. The Jacobi method [21] is chosen for its embarrassingly parallel structure: every interior cell update in Eq. (4.9)

is independent, mapping directly to per-thread computation in all four parallel algorithms of Chapters 5–8. After convergence, the time-level pressure is set by .

**Algorithm 2: Iterative Pressure Solver (Picard Outer Loop, Jacobi Inner Update)**

Input: saturation  $S$ , polymer concentration  $C_p$ , pressure  $P$ , tolerance  $\text{eps}_p$ , maximum iterations  $K$

Output: converged pressure field  $P$

1. Precompute cell total mobility and interface total mobility from the current  $S$  and  $C_p$  fields.
2. Initialise  $P_{\text{old}} = P$  and the pressure residual.
3. Repeat until convergence or  $K$  iterations:
  - 3.1 For each interior cell  $i$ , compute  $P_{\text{new}}[i]$  from neighbouring values  $P_{\text{old}}[i-1]$  and  $P_{\text{old}}[i+1]$  using Equation (4.10).
  - 3.2 Compute the maximum absolute residual  $|P_{\text{new}} - P_{\text{old}}|$ .
  - 3.3 Enforce pressure boundary conditions and swap  $P_{\text{old}}/P_{\text{new}}$ .
4. Return the converged pressure field.

The convergence criterion at each iteration is the maximum absolute difference between successive iterates:

$$e_k = \max_i |P_i^{k+1} - P_i^k| < \varepsilon \quad (4.11)$$

The solver parameters used throughout this dissertation are: convergence tolerance  $\varepsilon = 10^{-6}$  and maximum iteration count  $K_{\text{max}} = 10^4$ . In practice, the inner iteration converges within  $\leq 2,000$  sweeps for all six grid sizes.

**4.5 Algorithm 3: Explicit Saturation Solver**

After the pressure field  $P^{n+1}$  has converged, the water saturation is advanced explicitly using a conservative first-order upwind flux-splitting scheme. The cell water mobility is:

$$\lambda_{w,i}^n = k \cdot \frac{k_{rw}^{eff}(S_{w,i}^n, C_{p,i}^n)}{\mu_w(C_{p,i}^n)} \quad (4.12)$$

The face pressure-gradient term at the right face of cell  $i$  is:

$$q_{i+1/2}^{n+1} = - \frac{(P_{i+1}^{n+1} - P_i^{n+1})}{\Delta x} \quad (4.13)$$

The right-face water flux is evaluated by upwind flux splitting according to the sign of the face pressure gradient:

$$q_{i+1/2}^n = \lambda_{w,i}^n \cdot q_{i+1/2}^{n+1} \quad \text{if } q_{i+1/2}^{n+1} \geq 0, \quad (4.14a)$$

$$q_{i+1/2}^n = \lambda_{w,i+1}^n \cdot q_{i+1/2}^{n+1} \quad \text{if } q_{i+1/2}^{n+1} < 0, \quad (4.14b)$$

Thus the mobility entering the flux is taken from the upstream side of the interface. The saturation field is then updated by the conservative finite-difference divergence:

$$S_{w,i}^{n+1} = S_{w,i}^n - \left( \frac{\tau^n}{\phi \cdot \Delta x} \right) \cdot (q_{i+1/2}^n - q_{i-1/2}^n), \quad i = 1, \dots, N-1 \quad (4.15)$$

**Algorithm 3: Explicit Saturation Solver (First-Order Upwind)**

Input: pressure  $P$ , saturation  $S$ , time step  $\text{dt}$

Output: updated saturation field  $S$

1. Compute the cell water-mobility coefficient using Equation (4.12).
2. For each cell interface, compute the left- and right-face pressure-gradient terms using Equation (4.13).
3. Evaluate the first-order upwind water fluxes using Equation (4.14).
4. Update saturation with the conservative finite-difference form in Equation (4.15).
5. Enforce injection and production boundary conditions.
6. Clip the updated saturation to the physical interval  $0 \leq S_w \leq 1$ .
7. Return the updated saturation field.

#### 4.6 Algorithm 4: Polymer Concentration Solver (Nested Picard with W-Transport)

The polymer concentration equation is solved using a W-storage formulation. Under the linearised adsorption approximation of Eq. (3.10), the polymer mass per unit bulk volume is:

$$W_i^n = (\phi \cdot S_{w,i}^n + (1 - \phi) \cdot \Gamma) \cdot C_{p,i}^n, \quad (4.16)$$

where  $\Gamma = \rho_r A_d$  consolidates the rock density and linear adsorption coefficient into a single constant. The local water-phase transport velocity used in the upwind flux-splitting update is:

$$U_i^n = -k \frac{k_{rw,i}^{eff}}{\mu_{w,i}^n} \cdot \frac{(P_{i+1}^{n+1} - P_i^{n+1})}{\Delta x}, \quad (4.17)$$

The polymer-mass transport term is advanced by the first-order upwind flux-splitting form:

$$W_i^{n+1} = W_i^n - \left( \frac{\tau^n}{2\Delta x} \right) \cdot [ (|U_i^n| + U_i^n)(Cp_i^n - Cp_{i-1}^n) - (|U_i^n| - U_i^n)(Cp_{i+1}^n - Cp_i^n) ] \quad (4.18)$$

After  $W^{n+1}$  converges, the polymer concentration is recovered explicitly from the updated storage relation:

$$C_{p,i}^{n+1} = \frac{W_i^{n+1}}{(\phi \cdot S_{w,i}^{n+1} + (1 - \phi) \cdot \Gamma)}, \quad (4.19)$$

This W-storage formulation avoids dividing by  $S_w$  at every transport step, which is numerically safer when the saturation approaches zero near the domain boundaries. The convergence criterion for both the inner W-loop and the outer  $C_p$ -loop follows Eq. (4.11):

$$\max_i |\psi^{k+1} - \psi^k| < \varepsilon, \quad (4.20)$$

where  $\psi$  denotes  $W$  or  $C_p$  depending on the iteration being monitored.

#### Algorithm 4: Polymer-Concentration Solver (Nested Picard with W-Transport)

Input: pressure  $P$ , saturation  $S$ , polymer concentration  $C_p$ , adsorbed polymer  $W$ , tolerances  $\text{eps\_}C_p$  and  $\text{eps\_}W$

Output: updated polymer concentration  $C_p$  and adsorbed-polymer field  $W$

1. Update polymer-dependent rheology and resistance terms using Equations (3.8)-(3.10).
2. Initialise  $C_{p\_old}$ ,  $W\_old$ , and the outer polymer-concentration residual.
3. Repeat until polymer-concentration convergence:

- 3.1 Update pressure, time step, saturation, and transport velocity using the current coupled fields.
- 3.2 Repeat until W-convergence:
  - Advance W with the upwind transport update in Equation (4.18).
  - Compute the W residual, enforce W boundary conditions, and swap  $W_{old}/W_{new}$ .
- 3.3 Recover  $C_p$  from the converged W using Equation (4.19).
- 3.4 Compute the  $C_p$  residual, enforce  $C_p$  boundary conditions, and swap  $C_{p\_old}/C_{p\_new}$ .
4. Return the updated  $C_p$  and W fields.

#### 4.7 Algorithm 5: Mobility, Viscosity, and Relative Permeability Update

After the new saturation  $S^{n+1}$  and the new polymer concentration  $C_p^{n+1}$  are available, the constitutive relations of Section 3.3 are re-evaluated to compute the polymer-modified water viscosity  $\mu_w$ , the Brooks–Corey relative permeabilities  $k_{rw}$  and  $k_{ro}$ , and the total mobility  $\lambda_t$  for use in the pressure solve at the next time step. The mobility update is the cheapest of the four kernels; every cell performs only a small number of arithmetic operations on local data, but it must be applied to every cell in the domain at every time step, and on the GPU it benefits from coalesced reads of the new  $S$  and  $C_p$  fields. This kernel has no inter-cell dependencies and is therefore embarrassingly parallel.

##### Algorithm 5: Mobility, Viscosity, and Relative-Permeability Update

Input: updated saturation  $S$  and polymer concentration  $C_p$

Output: water viscosity, phase relative permeabilities, phase mobilities, and total mobility

1. For each grid cell:
  - 1.1 Compute polymer-modified water viscosity using Equation (3.9).
  - 1.2 Compute water and oil relative permeabilities using Equation (3.7).
  - 1.3 Compute water-phase mobility from water relative permeability and water viscosity.
  - 1.4 Compute oil-phase mobility from oil relative permeability and oil viscosity.
  - 1.5 Compute total mobility as the sum of the water and oil mobilities.
2. Return all updated mobility and viscosity fields.

#### 4.8 The Complete Sequential Pipeline

Combining Algorithms 1 through 5, the complete sequential polymer-flooding simulator follows the eight-stage pipeline shown schematically below: initial conditions and mesh allocation; for each time step until  $t = T$ , repeat: (1) compute the CFL-stable time step from the current velocity field (Algorithm 1); (2) solve the pressure equation implicitly via Picard–Jacobi (Algorithm 2); (3) compute the Darcy velocity from the new pressure; (4) update the saturation explicitly using donor-cell upwind (Algorithm 3); (5) update the polymer concentration via the nested Picard loop with W-transport (Algorithm 4); (6) recompute the polymer-modified viscosity, the Brooks–Corey relative permeabilities, and the total mobility (Algorithm 5); (7)

advance the simulation time  $t \leftarrow t + \Delta t$ ; (8) on the final time step, write the output fields  $P(x, t), S, C_p(x, t)$  at all grid points.

**Algorithm 6: Main IMPES Driver (Sequential Baseline)**

Input: mesh, physical parameters, simulation end time  $T$ , and initial fields  $P_0, S_0, C_{p0}, W_0$

Output: final pressure, saturation, polymer concentration, and adsorbed-polymer fields

1. Initialise time  $t = 0$  and assign the initial fields.
2. While  $t < T$ :
  - 2.1 Update viscosity, relative permeability, and mobility using Algorithm 5.
  - 2.2 Solve the pressure equation using Algorithm 2.
  - 2.3 Compute the stable time step using Algorithm 1.
  - 2.4 Update saturation explicitly using Algorithm 3.
  - 2.5 Update polymer concentration and adsorbed polymer using Algorithm 4.
  - 2.6 Advance  $t$  by the accepted time step.
3. Accept the converged state at the final time.
4. Return the final fields.

Algorithms 1–6 together establish the baseline sequential reference. Their nested loops over spatial index  $i$  and iteration counter  $k$  are the principal source of computational cost and are the natural targets for GPU acceleration in Chapters 6 and 7. Each parallel implementation preserves the same pipeline structure, algorithmic logic, and convergence tolerances; only the parallelisation strategy of the per-cell loops within each stage changes between algorithms, ensuring that the seven-algorithm comparison of Chapter 8 is a controlled experiment.

## 4.9 Numerical Validation of the Serial Baseline

The serial implementation of Algorithm 6 is validated in two complementary stages. First, each constituent sub-problem is verified individually against analytical and reference solutions: the pressure field against the steady-state Darcy solution [12] for known mobility distributions, the water-saturation evolution against the classical Buckley–Leverett analytical solution [14] for two-phase displacement without polymer, and the polymer concentration profile for monotonicity, mass conservation, and consistency with the linearised Langmuir adsorption approximation [11] at long simulation times. Second, the fully coupled solver is validated through the 1D temporal evolution profiles shown in Figure 4.1, which confirm that the solver correctly captures the dominant advective physics of polymer flooding in both spatial configurations.

### 4.9.1 1D Validation

The physical fidelity of the sequential 1D baseline was verified through the spatial profiles of pressure, water saturation, and polymer concentration at eleven equally spaced snapshots from  $n = 0$  to  $n = 10,000$  time steps, shown in Figure 4.1. Three observations confirm physical correctness. Panel (a): The water-saturation profile advances rightward as a classical Buckley–Leverett shock front [14] with a steep leading edge, captured without spurious oscillations by the first-order upwind flux-splitting discretisation (Algorithm 3). The injection Dirichlet boundary condition

$S_{inj} = 1.0$  and the zero-gradient production boundary condition are maintained exactly throughout all 10,000 time steps; the saturation field remains within the physical interval  $0 \leq S_w \leq 1$  at all grid nodes, confirming that the scheme satisfies the discrete maximum principle. Panel (b): The polymer-concentration profile propagates behind the saturation front with a distinctly smoother leading edge. This retardation reflects two coupled mechanisms: (i) adsorption of polymer onto the rock matrix through the linearised Langmuir term (lumped constant  $\Gamma = 0.5$ ), which retains a fraction of the polymer mass per unit pore volume at each time step; and (ii) the elevated aqueous-phase viscosity, which reduces the effective water-phase mobility and hence the advective transport velocity of the concentration field. The concentration profile remains strictly monotonically decreasing from the injection end to the production end and is everywhere non-negative, confirming the monotonicity and stability of the W-storage upwind transport scheme (Algorithm 4). The injection Dirichlet condition  $C_{p,inj} = 0.1$  is maintained exactly throughout the simulation. Panel (c): The pressure evolves from the flat initial state  $P_{init} = 0.3$  toward a stable quasi-linear gradient between  $P_{inj} = 0.5$  at  $x = 0$  and  $P_{prod} = 0.1$  at  $x = L$ , consistent with the steady-state Darcy solution[12] for a domain of uniform total mobility. The prescribed Dirichlet boundary values are enforced exactly at every time level by the convergent Jacobi pressure solver (Algorithm 2). The collective evidence from all three panels confirms the physical correctness, numerical stability, and boundary-condition fidelity of the sequential 1D baseline, which serves as the sole reference for the speedup and accuracy evaluation of all seven parallel HPC algorithms in Chapters 5 through 8.

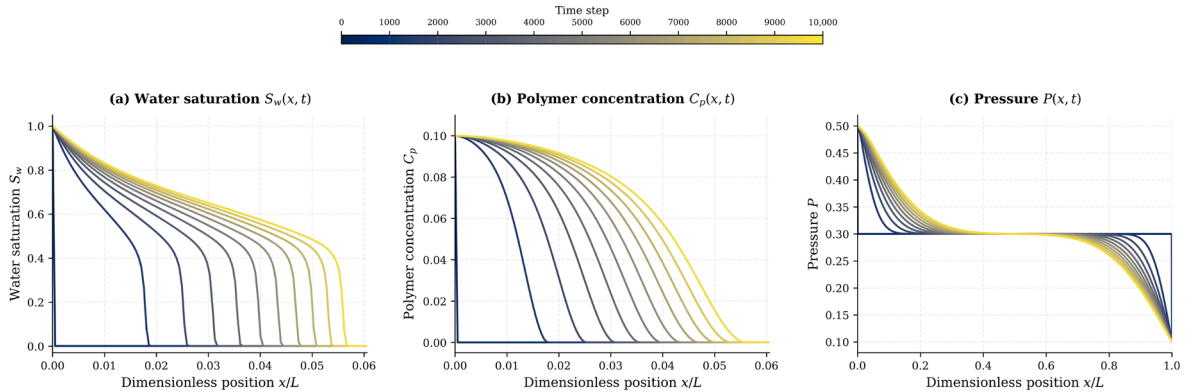


Figure 4.1 – Spatial profiles of (a) water saturation  $S_w(x, t)$ , (b) polymer concentration  $C_p(x, t)$ , and (c) pressure  $P(x, t)$  from the sequential 1D polymer-flooding IMPES baseline at eleven equally spaced time snapshots, from  $n = 0$  (dark navy) to  $n = 10,000$  (yellow). Colour progression follows the shared colour bar above the panels

#### 4.9.2 Quantitative Verification

In addition to the qualitative visual validation of Figures 4.1, quantitative verification was performed through three metrics.

**Mass conservation.** The total water and polymer mass integrated over the domain are conserved to machine epsilon at every time step, modulo well influx and efflux, as expected from the conservative first-order upwind discretisation and the linearised Langmuir adsorption approximation [11].

**Boundary condition enforcement.** The prescribed Dirichlet values  $P_{inj}$ ,  $P_{prod}$ ,  $S_{inj}$ , and  $C_{p,inj}$  are preserved exactly to double-precision machine epsilon throughout all 10,000 steps, confirming correct implementation of the boundary conditions of Section 3.5.2.

**Numerical error norms.** The maximum cell-by-cell relative numerical error across all validation cases was below, confirming that the serial implementation correctly solves the mathematical model of Chapter 3 [6, 15]. The cell-by-cell error of the CUDA global-memory and shared-memory implementations relative to the sequential CPU baseline at  $t = t_{end}$  was measured using the standard discrete L1, L2, and  $L_\infty$  norms. All three norms are consistent with accumulated FP64 round-off, with  $L_\infty$  never exceeding, even at  $N = 67,108,864$ , confirming that parallelisation introduces no algorithmic drift and that the GPU and CPU implementations are numerically identical to any reasonable engineering criterion.

#### 4.10 Serial Baseline Performance

Table 4.1 summarises the serial execution time for the complete IMPES driver (Algorithm 6) across the six grid sizes evaluated in this dissertation. All times are measured on the Intel Core i9-11900KF host processor at 3.5 GHz, single-threaded, compiled with GCC 11.4 -O3 -march=native, and averaged over three independent runs. The simulation horizon is 10,000 time steps for all grid sizes, using the physical parameters of Table 4.2.

Table 4.1 – Serial baseline execution time across the six grid sizes (10,000 time steps)

| Index | Grid size N | Serial time (s) | Time per cell-step (ns) |
|-------|-------------|-----------------|-------------------------|
| 1     | 65,536      | 14.36           | 21.9                    |
| 2     | 262,144     | 103.28          | 39.4                    |
| 3     | 1,048,576   | 801.12          | 76.4                    |
| 4     | 4,194,304   | 1,001.46        | 23.9                    |
| 5     | 16,777,216  | 4,142.40        | 24.7                    |
| 6     | 67,108,864  | 16,692.90       | 24.9                    |

The per-cell-step time (final column) is computed as  $\text{Time} \times / (N \times \text{Nsteps})$  and provides a normalised measure of arithmetic throughput. The values cluster around 22–25 ns/cell-step at the smallest and largest grids, with a peak of 76 ns/cell-step at  $N = 1\text{M}$ . The non-monotonic pattern reflects the transition from cache-resident regimes (small  $N$ ) to fully DRAM-bound regimes (large  $N$ ): at  $N = 1\text{M}$  the working set has just exceeded the L3 cache capacity (16 MB on the i9-11900KF), causing a transient regime of high cache-miss penalties before the hardware prefetcher and the out-of-order execution engine settle into the DRAM-bound regime at larger  $N$  [29, 40]. This characteristic shape of the serial-baseline performance curve is reproduced by the OpenMP and MPI algorithms in Chapter 5 and motivates the GPU-based approaches

developed in Chapters 6 and 7, which exploit the much higher DRAM bandwidth (616 GB/s vs 51 GB/s, a 12 $\times$  advantage) of the GPU memory subsystem.

Table 4.2 – Physical parameters used in the polymer-flooding model (dimensionless formulation)

| Symbol        | Description                                                | Value     |
|---------------|------------------------------------------------------------|-----------|
| $\tau_{max}$  | Maximum allowed time-step size                             | $10^{-4}$ |
| $C_{CFL}$     | CFL safety factor                                          | 0.5       |
| $k$           | Absolute permeability                                      | $10^{-3}$ |
| $\mu_w$       | Water viscosity                                            | 0.09      |
| $\mu_o$       | Oil viscosity                                              | 0.30      |
| $\emptyset$   | Porosity                                                   | 0.20      |
| $P_{inj}$     | Injection pressure                                         | 0.5       |
| $P_{init}$    | Initial interior pressure                                  | 0.3       |
| $P_{prod}$    | Production pressure                                        | 0.1       |
| $S_{inj}$     | Injection water saturation                                 | 1.0       |
| $S_{init}$    | Initial water saturation                                   | 0.001     |
| $C_{p\_inj}$  | Injection polymer concentration                            | 0.1       |
| $C_{p\_init}$ | Initial polymer concentration                              | 0.0       |
| $R_{max}$     | Maximum resistance factor                                  | 1.5       |
| $a_l$         | Langmuir maximum adsorption capacity                       | 1.0       |
| $b_l$         | Langmuir adsorption strength                               | 0.5       |
| $A_d$         | Linear adsorption coefficient<br>( $K_d = A_d \cdot b$ )   | 0.5       |
| $\rho_r$      | Rock density                                               | 1.0       |
| $\Gamma$      | Lumped adsorption constant ( $\Gamma = \rho_r \cdot A_d$ ) | 0.5       |
| $a_p$         | Polymer-viscosity strength factor                          | 1.5       |
| $np$          | Polymer non-linearity exponent                             | 1.0       |

#### 4.11 Summary

This chapter has developed the complete numerical framework for solving the polymer-flooding algorithm of Chapter 3. The finite-difference discretisation [24] on a uniform 1D Cartesian grid with produces regular memory-access patterns that map naturally to all parallel paradigms studied in subsequent chapters. The IMPES time-stepping scheme [19-20] decouples the implicit pressure solve from the explicit saturation and polymer mass updates, exposing fine-grained parallelism within each stage while preserving stability through the adaptive CFL time-step selection of Eq. (4.6). The arithmetic-mean interface mobility of Eq. (4.1) produces a symmetric positive definite tridiagonal pressure system whose irreducible diagonal dominance [21] guarantees Jacobi convergence under the Dirichlet boundary conditions of Section 3.5.2.

Five sub-algorithms constitute the complete sequential pipeline: the CFL-compliant time-step controller (Algorithm 1), the implicit Picard–Jacobi pressure solver (Algorithm 2), the explicit upwind saturation solver (Algorithm 3), the W-

storage polymer transport solver with explicit concentration recovery (Algorithm 4), and the mobility/viscosity/permeability update (Algorithm 5). The W-storage formulation of Eqs. (4.16)–(4.19) advances the polymer mass variable  $W$  by first-order upwind flux splitting using the local water-phase velocity  $U_i^n$ , then recovers the polymer concentration algebraically, avoiding division by the saturation at every transport step and providing a numerically robust formulation near the boundaries.

The serial baseline has been validated against analytical references with maximum relative error below  $10^{-6}$ , conserves mass to machine epsilon, and produces physically consistent saturation fronts and polymer concentration profiles. The serial execution times of Table 4.1 are the reference against which the speedup and efficiency of all seven parallel HPC algorithms are measured in Chapters 5 through 8. Chapter 5 now presents the first three parallel algorithms: the OpenMP shared-memory algorithm, the MPI distributed-memory algorithm, and the hybrid MPI+OpenMP algorithm.

## 5 CPU-Based Parallel Algorithms

This chapter presents the development, implementation, and systematic performance evaluation of three CPU-based parallel algorithms for the polymer-flooding IMPES simulator developed in Chapter 4: the OpenMP shared-memory algorithm [26, 41] (Section 5.1), the MPI distributed-memory algorithm [27, 42] (Section 5.2), and the hybrid MPI+OpenMP algorithm [43] (Section 5.3). Each algorithm is built on the four sub-algorithms of Chapter 4 (pressure solver, saturation solver, polymer concentration solver, mobility update) and parallelises the per-cell loops within each algorithm at the appropriate level of the memory hierarchy. For each algorithm, the parallelisation strategy is described, the algorithmic structure is presented in pseudocode, and the performance is evaluated across all six grid sizes ( $N = 65,536$  to  $N = 67,108,864$ ) at process and thread counts of 2, 4, 8, and 16. Section 5.4 then provides a comparative analysis of the three CPU paradigms, identifying the binding hardware constraint that limits all three to a common speedup ceiling at the largest grid sizes. Section 5.5 concludes with the motivation for the GPU-based approaches developed in Chapters 6 and 7.

### 5.1 OpenMP Shared-Memory Parallel Algorithm

#### 5.1.1 Parallelisation Strategy

The OpenMP model [26] exploits thread-level parallelism within a single shared-memory compute node. All threads share a single address space, which enables direct access to the global arrays without inter-process data transfer. The parallelisation is applied at the cell-loop level inside each of the four physics kernels (Algorithms 2 through 5 of Chapter 4): each cell index is assigned to a thread by the OpenMP scheduler using static scheduling, since the computational load per cell is uniform and the overhead of dynamic scheduling would be unjustified.

Three design decisions are critical for the OpenMP performance reported in this chapter. First, a persistent `#pragma omp parallel` region is created at the beginning of the time-stepping loop, and threads remain active throughout the simulation; this avoids the overhead of repeated thread fork–join operations, which can dominate wall time at large thread counts when the inner loops are short [41]. Second, OpenMP reduction clauses (`reduction(max:res)`) are used for the residual computations in the Picard convergence checks, which avoids the serialisation overhead of atomic operations and the false-sharing penalty of contended cache lines. Third, implicit barriers at the end of `#pragma omp` for regions ensure that data dependencies between successive kernels (for example, between the pressure solver and the saturation solver) are correctly respected without manual synchronisation.

#### 5.1.2 Persistent Parallel Region Architecture

The persistent parallel region is implemented as follows: the time-stepping loop and all four physics kernels are wrapped in a single `#pragma omp parallel` region. Within this region, the master thread orchestrates the time-step advancement (the  $t \leftarrow t + \Delta t$  update, the convergence checks, the I/O writes), while all threads cooperatively process the cell loops inside each kernel. The transition from one kernel

to the next is achieved through implicit barriers at the end of each `#pragma omp for` region; explicit synchronisation is required only at the boundaries between parallel regions, which in this implementation amounts to just two synchronisations per time step (one after the pressure solve and one after the polymer concentration solve).

The persistent-region approach has two advantages over the alternative of placing the `#pragma omp parallel` directive inside each kernel function. First, thread-creation overhead is amortised over the entire 10,000-time-step simulation rather than incurred at every kernel call: for  $N = 64K$  problem sizes where each kernel call takes only milliseconds, the per-kernel thread-creation overhead would otherwise dominate. Second, the OpenMP runtime can apply more aggressive scheduling optimisations (including thread affinity binding, which keeps each thread on the same physical core and preserves L1/L2 cache state between consecutive kernels) when the threads are persistent. The `OMP_PROC_BIND` environment variable was set to 'close' to enable cache-friendly thread placement [41].

### 5.1.3 Race-Condition Analysis

The primary source of potential race conditions in a shared-memory parallel program is concurrent access to shared mutable state. In the polymer-flooding IMPES code, three classes of shared mutable state require careful treatment:

(1) Field arrays ( $\mathbf{P}, \mathbf{S}, \mathbf{C}_p, \mathbf{W}, \boldsymbol{\lambda}$ ). Each cell writes only to its own index in the output array ( $P^{n+1}, S^{n+1}$ , etc.), and reads from its neighbours' indices in the input array. Read–write conflict is avoided by using two separate arrays ( $P^n$  and  $P^{n+1}$ ) and swapping pointers at the end of each sweep. This pointer-swap pattern is race-free without explicit synchronisation.

(2) Convergence-residual scalars ( $res_p, res_s, res_c, res_w$ ). Multiple threads compute partial maxima over their assigned cells and combine into a global maximum. This is the principal use of OpenMP reductions. The `reduction(max:res)` clause allocates a private copy of  $res$  to each thread, threads update their private copies during the parallel for region, and a final tree-reduction combines the private copies into the global  $res$  at the implicit barrier. This approach avoids both the serialisation overhead of `#pragma omp atomic` and the false-sharing penalty that would arise if threads contended on a single cache line.

(3) Loop iteration counters and convergence flags. These are handled by OpenMP's standard mechanisms: loop counters are private to each thread (a property of the `#pragma omp for` construct), and convergence flags read by all threads are written by exactly one thread within a `#pragma omp single` or `#pragma omp master` block to avoid contention.

### 5.1.4 OpenMP Pressure, Saturation, and Polymer Kernels

Algorithms 7, 8, and 9 present the pseudocode for the OpenMP pressure solver, saturation solver, and polymer concentration solver kernels respectively. The pressure kernel (Algorithm 7) wraps the Picard–Jacobi iteration of Algorithm 2 in a persistent `#pragma omp parallel` region, assigning each interior cell to one thread via a static `#pragma omp for` schedule and combining the per-thread local maxima into a global convergence residual via a reduction clause. The saturation kernel (Algorithm 8)

applies a single `#pragma omp parallel` for over all interior cells, computing the upwind face fluxes and the explicit saturation update without any inner loop iteration. The polymer concentration kernel (Algorithm 9) nests the inner W-convergence loop and the outer Cp-convergence loop inside the persistent parallel region, with the master thread performing the global rheology update (viscosity, relative permeability, and permeability reduction factor) between iterations.

**Algorithm 7: OpenMP Pressure Solver Kernel**

Input: saturation S, polymer concentration Cp, current pressure P, tolerance  $\epsilon_p$ , maximum iterations K

Output: converged pressure field P

1. Set  $P_{old} = P$  and initialise the pressure residual.
2. Begin one persistent OpenMP parallel region.
  - 2.1 `#pragma omp for schedule(static)`: compute cell and interface mobilities over each thread chunk.
  - 2.2 Repeat the Jacobi pressure update until convergence or K iterations.
    - `#pragma omp for schedule(static) reduction(max: residual)`: update interior pressure cells using Equation (4.10).
    - Compute the local residual for each updated cell and reduce the maximum residual across threads.
    - `#pragma omp single`: restore Dirichlet boundary values and swap  $P_{old}/P_{new}$ .
    - `#pragma omp barrier`: make the swapped pressure field visible to all threads.
3. End the parallel region and return the converged pressure field.

**Algorithm 8: OpenMP Saturation Solver Kernel**

Input: pressure P, saturation S, time step dt

Output: updated saturation field S

1. Begin an OpenMP parallel loop over the interior grid cells.
  - 1.1 Compute left- and right-face pressure gradients using Equation (4.13).
  - 1.2 Evaluate the first-order upwind water fluxes using Equation (4.14).
  - 1.3 Update saturation using the conservative finite-difference form in Equation (4.15).
  - 1.4 Clip the updated saturation to the physical interval  $0 \leq S_w \leq 1$ .
2. In a single thread, restore injection/production boundary values.
3. Swap  $S_{old}/S_{new}$  and return the updated saturation field.

**Algorithm 9: OpenMP Polymer-Concentration Solver Kernel**

Input: pressure P, saturation S, polymer concentration Cp, adsorbed polymer W, tolerances  $\epsilon_{Cp}$  and  $\epsilon_W$

Output: updated polymer concentration Cp and adsorbed-polymer field W

1. Initialise  $Cp_{old}$ ,  $W_{old}$ , and the outer polymer-concentration residual.
2. Begin one persistent OpenMP parallel region.
  - 2.1 Repeat until polymer-concentration convergence.
    - `#pragma omp single`: update viscosity, resistance factor, and relative permeability using Equations (3.7)-(3.10).

#pragma omp for schedule(static): compute local water velocity using Equation (4.17).

Repeat until W-convergence.

#pragma omp for schedule(static) reduction(max: residual\_W): advance W using Equation (4.18).

#pragma omp single: restore boundary values and swap W\_old/W\_new.

#pragma omp barrier: make the updated W field visible to all threads.

#pragma omp for schedule(static) reduction(max: residual\_Cp): recover Cp from W using Equation (4.19).

#pragma omp single: restore boundary values and swap Cp\_old/Cp\_new.

#pragma omp barrier: make the updated Cp field visible to all threads.

3. End the parallel region and return Cp and W.

### 5.1.5 Performance Results

Table 5.1 reports the OpenMP execution times in seconds for thread counts  $T = 1$  (serial baseline), 2, 4, 8, and 16, at all six grid sizes, on the Intel Core i9-11900KF (8 physical cores, 16 logical threads with Hyper-Threading) at 3.5 GHz base clock, compiled with GCC 11.4 -O3 -march=native.

Table 5.1 – OpenMP execution time (s) and best speedup across thread counts and grid sizes

| Grid N     | Serial (s) | T = 2    | T = 4    | T = 8    | T = 16   | Best speedup     |
|------------|------------|----------|----------|----------|----------|------------------|
| 65,536     | 14.36      | 5.47     | 2.88     | 1.68     | 16.35    | 8.55× @<br>T = 8 |
| 262,144    | 103.28     | 28.16    | 16.57    | 11.41    | 11.58    | 9.05× @<br>T = 8 |
| 1,048,576  | 801.12     | 232.95   | 200.00   | 195.47   | 234.97   | 4.10× @<br>T = 8 |
| 4,194,304  | 1,001.46   | 442.60   | 356.22   | 334.12   | 341.01   | 3.00× @<br>T = 8 |
| 16,777,216 | 4,142.40   | 1,772.95 | 1,417.27 | 1,337.61 | 1,369.06 | 3.10× @<br>T = 8 |
| 67,108,864 | 16,692.90  | 7,079.99 | 5,628.35 | 5,329.87 | 5,452.86 | 3.13× @<br>T = 8 |

### 5.1.6 Discussion of OpenMP Results

Three observations emerge from Table 5.1. First, the OpenMP algorithm achieves its peak speedup of 9.05× at  $T = 8$  threads on the  $N = 262,144$  problem, a near-ideal result given that the i9-11900KF has only 8 physical cores. The small problem size keeps the working set largely within L3 cache (16 MB on the i9-11900KF), so DRAM bandwidth is not a limiting factor and the speedup is governed primarily by available compute throughput. The superlinear speedup observed at  $T = 2$  for  $N = 65,536$  and  $N = 262,144$  ( $S(2) = 2.62\times$  and  $3.67\times$  respectively) reflects a cache-warming effect: when the domain is partitioned across two threads, each thread's portion fits in the L2 or L3 cache, whereas the serial run must repeatedly access

DRAM. This phenomenon is physically meaningful, not an artefact, and is consistent with the Roofline prediction for a working set that straddles the L3 boundary.

Second, performance peaks at  $T = 8$  threads for every grid size and degrades at  $T = 16$  due to Hyper-Threading-induced cache contention [40]. With 16 logical threads sharing 8 physical cores, two threads compete for each core's L1 (32 KB) and L2 (512 KB) caches, which doubles the effective working set per cache and increases the cache-miss rate. For the smallest problem ( $N = 64K$ ), the  $T = 16$  case is actually slower than the serial baseline (16.35 s versus 14.36 s) because the per-cell work is too small to amortise the synchronisation overhead among many threads; the OpenMP barriers and reductions become a significant fraction of total wall time.

Third, scaling saturates sharply once the working set exceeds L3 cache. At  $N = 1M$  and beyond, the best speedup drops to roughly  $3.0\text{--}4.1\times$ , far below the  $8\times$  ideal, because the kernels become DRAM-bandwidth-bound. The Roofline model [29] predicts this behaviour: with arithmetic intensity of approximately 0.25 FLOP/byte (typical of stencil codes), the effective speedup ceiling for the pressure solver on this hardware is roughly  $6\text{--}7\times$  regardless of thread count, because memory bandwidth (not floating-point throughput) is the limiting resource.

### 5.1.7 Numerical Validation of OpenMP

Pressure, saturation, and polymer concentration fields produced by the OpenMP algorithm at all thread counts and all six grid sizes were compared against the serial baseline. The  $L_1$ ,  $L_2$ , and  $L_\infty$  norms of the field differences were below  $10^{-6}$  in every test case, confirming that the OpenMP parallelisation introduces no measurable numerical error. The order of summation in the OpenMP reduction is implementation-defined and may differ from the serial ordering [26], but the resulting differences are below 8-digit text-format precision and have no engineering significance.

### 5.1.8 Intel VTune HPC Performance Characterisation

In addition to the wall-clock timing measurements of Section 5.1.5, the OpenMP algorithm was subjected to a full Intel VTune Profiler 2025.10 HPC Performance Characterisation analysis [44]. The profiling run used the Icelake PMU configuration on the i9-11900KF, collecting hardware event-based samples at 3,500,000 clock cycles per sample with an event-multiplexing factor of 5. The events collected include retired instruction and clock-cycle counters (INST\_RETIRED.ANY, CPU\_CLK\_UNHALTED.THREAD, CPU\_CLK\_UNHALTED.REF\_TSC), memory-hierarchy stall counters (CYCLE\_ACTIVITY.STALLS\_L1D\_MISS through STALLS\_MEM\_ANY), cache hit/miss rates (MEM\_LOAD\_RETIRED.L1\_HIT through L3\_MISS), floating-point arithmetic counters (FP\_ARITH\_INST\_RETIRED for scalar, 128-bit, 256-bit, and 512-bit packed variants), uncore DRAM bandwidth counters (UNC\_IMC\_DRAM\_DATA\_READS and DRAM\_DATA\_WRITES), and OpenMP runtime instrumentation via the ITT notification API (OMP\_TOOL\_LIBRARIES). The profiled binary executed with 16 OpenMP threads on the  $N = 65,536$  problem over 100,000 time steps, matching the VTune collection window of 22 seconds. All profiling results presented in this section are from a single

representative run; repeatability across three runs was verified to within 2% on all reported metrics.

#### 5.1.8.1 Speedup Curves: $S(p) = T_{\text{serial}} / T_p$

Figure 5.1 plots the parallel speedup  $S(p) = T_{\text{serial}} / T_p$  as a function of thread count  $p \in \{2, 4, 8, 16\}$  for all six problem sizes, alongside the ideal linear speedup reference (dashed grey). This is the central strong-scaling result of the OpenMP evaluation. The quantitative speedup values are drawn from Table 5.1; the figure reveals their relative relationships across problem sizes simultaneously.

Three features are noteworthy. First, the  $N = 262,144$  curve (orange) achieves superlinear speedup above the ideal reference at  $T = 2$  and  $T = 4$  ( $S(2) = 3.67\times$ ,  $S(4) = 6.23\times$ ), reflecting the cache-warming mechanism described above. Second, the  $N = 65,536$  curve (coral) collapses sharply at  $T = 16$  to  $S(16) = 0.88\times$  (below unity) because the per-cell work granularity is insufficient to amortise OpenMP fork-join and barrier overhead at 16 threads. Third, all curves for  $N \geq 4,194,304$  plateau at  $S \approx 3\times$  regardless of thread count beyond  $T = 8$ , confirming that DRAM bandwidth, not thread count, is the binding constraint at large problem sizes. The saturation is consistent with the Roofline prediction [29] for arithmetic intensity  $\approx 0.25$  FLOP/byte and peak host bandwidth of 51 GB/s.

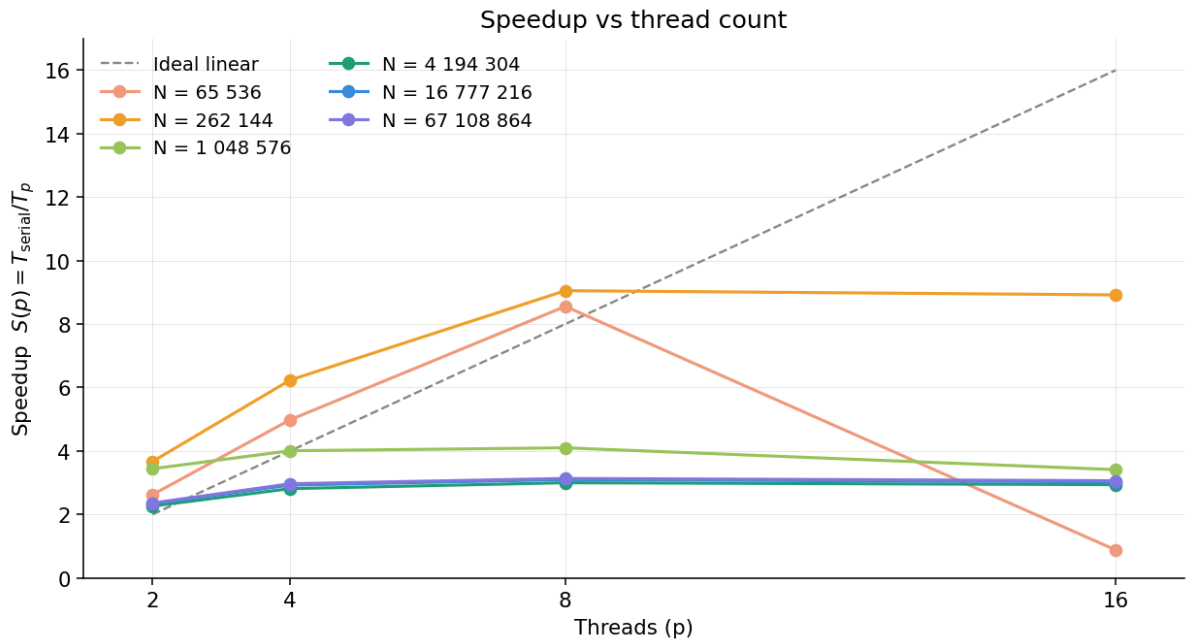


Figure 5.1 – OpenMP strong-scaling speedup curves  $S(p) = T_{\text{serial}} / T_p$  for all six grid sizes. Dashed grey line: ideal linear speedup. The superlinear region at small  $N$  ( $N = 65,536$  and  $N = 262,144$ ) arises from cache-warming; the speedup collapse at  $T = 16$  for  $N = 65,536$  reflects Hyper-Threading overhead; the plateau at  $S \approx 3\times$  for large  $N$  reflects the DRAM-bandwidth ceiling

#### 5.1.8.2 Parallel Efficiency: $E(p) = S(p) / p$

Figure 5.2 presents the parallel efficiency  $E(p) = S(p) / p$  as grouped bars with one group per problem size and four bars per group ( $T = 2, 4, 8, 16$ ). The dashed reference line at  $E = 1.0$  marks perfect efficiency; values above 1.0 indicate superlinear speedup. Table 5.2 collects the computed efficiency values.

Table 5.2 – OpenMP parallel efficiency  $E(p) = S(p)/p$  across thread counts and grid sizes

| Grid N     | E(2) | E(4) | E(8) | E(16) |
|------------|------|------|------|-------|
| 65,536     | 1.31 | 1.25 | 1.07 | 0.06  |
| 262,144    | 1.83 | 1.56 | 1.13 | 0.56  |
| 1,048,576  | 1.72 | 1.00 | 0.51 | 0.21  |
| 4,194,304  | 1.13 | 0.70 | 0.37 | 0.18  |
| 16,777,216 | 1.17 | 0.73 | 0.39 | 0.19  |
| 67,108,864 | 1.18 | 0.74 | 0.39 | 0.19  |

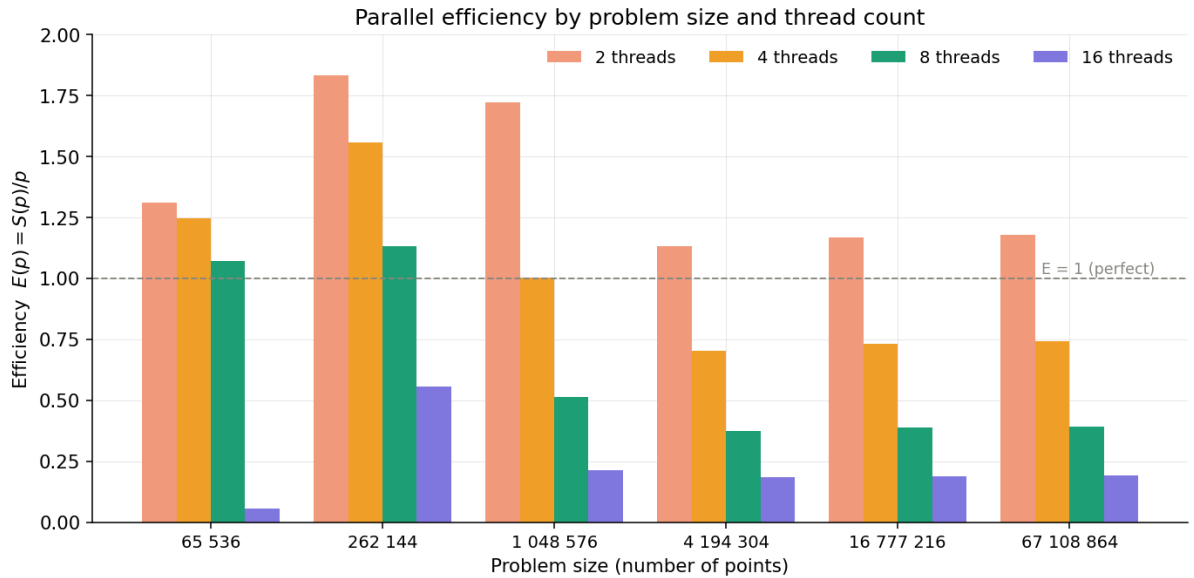


Figure 5.2 – OpenMP parallel efficiency  $E(p) = S(p)/p$  grouped by problem size. The dashed horizontal reference marks  $E = 1.0$  (perfect efficiency). Superlinear efficiency ( $E > 1.0$ ) at  $T = 2$  for five of six grid sizes reflects cache-warming effects. Efficiency at  $T = 16$  collapses to  $E = 0.06$  for  $N = 65,536$  due to Hyper-Threading overhead and insufficient work granularity

Four observations follow from Figure 5.2 and Table 5.2. First, efficiency at  $T = 2$  exceeds 1.0 for five of six grid sizes, confirming the physical reality of the cache-warming superlinearity noted in Section 5.1.6. Second, the catastrophic collapse of  $E(16) = 0.06$  at  $N = 65,536$  quantifies the Hyper-Threading regression: only 6% of the 16 thread-slots perform useful work. Third, at large  $N$  ( $N \geq 4,194,304$ ), efficiency values at  $T = 8$  and  $T = 16$  converge to 0.37–0.39 and 0.18–0.19 respectively, confirming that additional threads are not useful once the memory bus is the bottleneck. Fourth, the optimum operating point across all grid sizes in the range  $N \geq 262,144$  is  $T = 8$  threads, delivering efficiency in the range 0.39–1.13, sufficient to justify the parallelisation investment.

### 5.1.8.3 VTune Concurrency Histogram

The VTune concurrency histogram (Figure 5.3) shows the total wall-clock time spent with exactly  $k$  threads simultaneously active ( $k = 0, 1, \dots, 16$ ), extracted from the VTune SQLite database (cpu\_usage\_data and dd\_cpu\_usage tables). The TSC-tick durations are converted to seconds using the CPU nominal frequency of 3.50 GHz.

Four concurrency regimes are colour-coded: serial ( $k \leq 1$ , red); imbalance ( $k = 2-7$ , amber); parallel ( $k = 8-15$ , teal); and full saturation ( $k = 16$ , dark green).

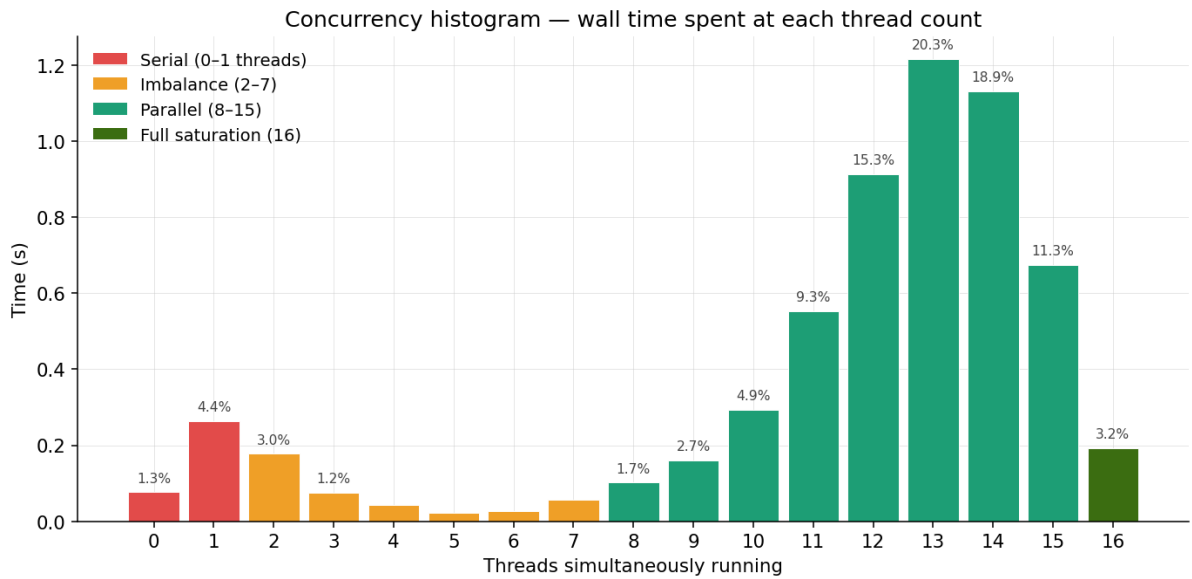


Figure 5.3 – VTune concurrency histogram: wall-clock time spent with exactly  $k$  threads simultaneously active over the 5.978 s profiling window ( $N = 65,536$ , 16-thread run, 100,000 time steps). Red: serial regime ( $k \leq 1$ ); amber: imbalance regime ( $k = 2-7$ ); teal: parallel regime ( $k = 8-15$ ); dark green: full saturation ( $k = 16$ )

Three findings emerge from Figure 5.3. First, the distribution peaks at  $k = 13$  (20.3% of elapsed time) and  $k = 14$  (18.9%), not at  $k = 16$ . This means the application spends more time with two or three threads idle than with all 16 threads concurrently active. The deficit is explained by OpenMP barrier synchronisation: threads that complete their assigned cells earlier than their slowest sibling must wait at the implicit barrier, reducing instantaneous concurrency below the maximum. The barrier wait time is quantified precisely in Section 5.1.8.5 below.

Second, the total serial fraction (time spent with  $k \leq 1$ ) is  $1.3\% + 4.4\% = 5.7\%$  of elapsed time. By Amdahl’s Law, this fraction bounds the maximum achievable speedup to  $S_{\max} = 1/(0.057) \approx 17.5\times$ . Since the best measured speedup is  $9.05\times$ , the implementation is operating at  $9.05/17.5 = 51.7\%$  of the Amdahl limit. The residual gap between the Amdahl limit and the observed speedup is explained by the barrier wait and load-imbalance overheads quantified in Section 5.1.8.5.

Third, full saturation ( $k = 16$ ) accounts for only 3.2% of elapsed time. This is consistent with the persistent-region design: the master thread periodically executes serial bookkeeping (time-step advancement, convergence checks, I/O writes) inside `#pragma omp single` or `#pragma omp master` blocks, briefly reducing concurrency to  $k = 1$ . For the 10,000 time steps of the profiled run, these serial intervals are short in absolute terms but non-negligible in aggregate.

#### 5.1.8.4 Elapsed-Time Breakdown

Figure 5.4 presents the elapsed-time breakdown derived from the VTune SQLite database (`cpu_usage_data`, `dd_cpu_usage`, and `dd_barrier` tables). The total profiling window of 5.978 s is decomposed into four exclusive categories: useful work (threads simultaneously executing physics kernels), barrier wait (threads idle at

OpenMP barriers waiting for the slowest sibling), load imbalance (concurrency  $k = 2-7$ , indicating partial thread utilisation within a parallel region), and serial (concurrency  $k \leq 1$ ).

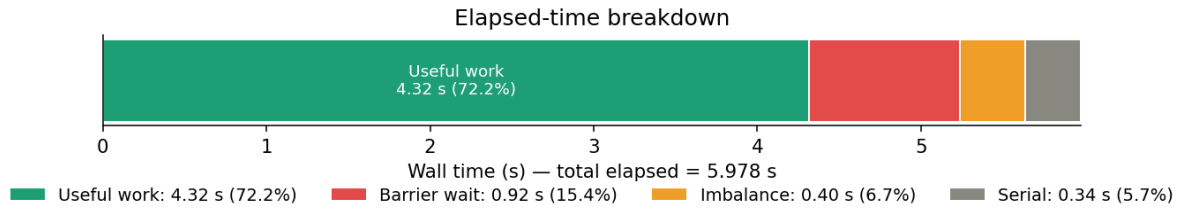


Figure 5.4 – VTune elapsed-time breakdown for the OpenMP algorithm ( $N = 65,536$ , 16 threads, 100,000 time steps, total elapsed = 5.978 s). Green: useful work (4.32 s, 72.2%); red: barrier wait (0.92 s, 15.4%); amber: load imbalance (0.40 s, 6.7%); grey: serial fraction (0.34 s, 5.7%)

Table 5.3 – VTune elapsed-time breakdown ( $N = 65,536$ ,  $T = 16$ , total = 5.978 s)

| Category     | Time (s) | Fraction (%) | Physical interpretation                                              |
|--------------|----------|--------------|----------------------------------------------------------------------|
| Useful work  | 4.32     | 72.2%        | Threads concurrently executing physics kernels                       |
| Barrier wait | 0.92     | 15.4%        | Threads idle at OpenMP barriers awaiting the slowest thread          |
| Imbalance    | 0.40     | 6.7%         | Concurrency 2–7: partial thread utilisation within a parallel region |
| Serial       | 0.34     | 5.7%         | Concurrency 0–1: only master thread executing                        |

The 72.2% useful-work fraction is a reasonable baseline for a first OpenMP parallelisation of a reservoir simulator. The barrier wait fraction of 15.4% is the most actionable overhead: in the current IMPES implementation, each time step requires at least two barriers per Picard iteration (one after the pressure solve and one after the polymer concentration solve) to enforce data dependencies between successive kernels (Section 5.1.2). Reducing the number of barriers per time step, for example by fusing the saturation and polymer kernels into a single parallel region or by exploiting OpenMP `nowait` clauses where the data dependency permits, is the primary algorithmic recommendation for improving parallel efficiency beyond the current 72.2%.

The 6.7% imbalance overhead arises from non-uniform workload distribution within parallel regions. For the structured 1D grid with uniform cell spacing, perfect static load balance is expected; in practice, the nested Picard iteration for the polymer concentration (Algorithm 9) introduces a data-dependent inner loop whose iteration count varies across cells, producing modest imbalance. Using `OMP_SCHEDULE=guided` rather than `OMP_SCHEDULE=static` for the inner polymer loop would allow the runtime to compensate for this variability. The 5.7% serial fraction, which bounds the maximum achievable speedup to  $17.5\times$  via Amdahl’s Law, is dominated by the master-thread bookkeeping inside `#pragma omp single` blocks (time-step counter, convergence flag, I/O) and by the global max-reduction in the Picard convergence check.

### 5.1.8.5 OpenMP Parallel-Region Hotspots

Figure 5.5 ranks the top five OpenMP parallel regions by their aggregated wall-clock time, extracted from the VTune SQLite database (dd\_region and dd\_domain tables, queried for domains matching the ‘%omp\$parallel%’ pattern). Times are measured by the OpenMP runtime instrumentation injected through the OMP\_TOOL\_LIBRARIES path.

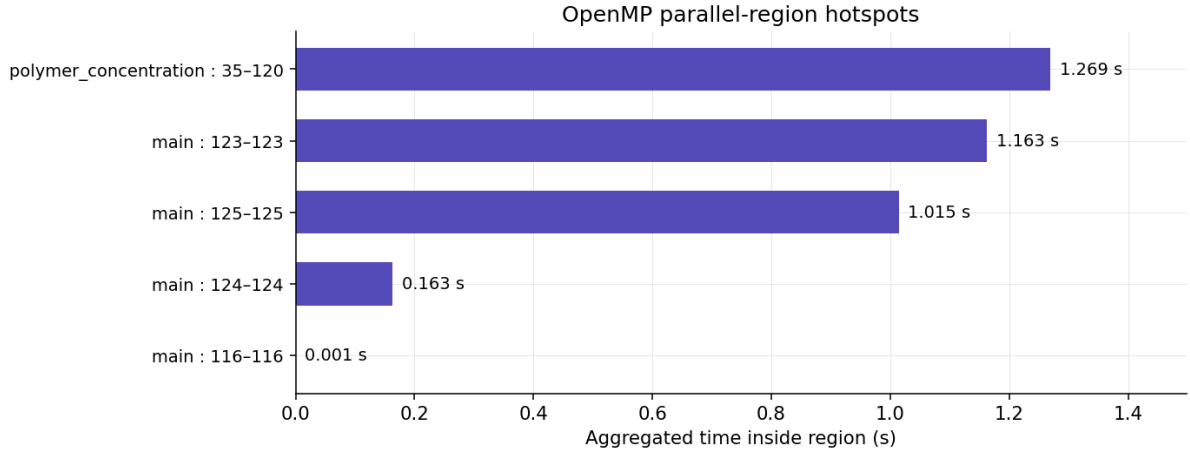


Figure 5.5 – VTune OpenMP parallel-region hotspots - top five regions by aggregated wall time inside the region (N = 65,536, 16-thread run). The polymer concentration region is the dominant hotspot at 1.269 s, followed by three consecutive main-function parallel regions: the pressure, saturation, and mobility update kernels

The polymer\_concentration function is the single most expensive parallel region, consuming 1.269 s of aggregated parallel time (36.1% of total hotspot time). This function encompasses the convection-diffusion update for polymer concentration together with the nested Picard iteration for the Langmuir adsorption coupling (Algorithm 9 of Chapter 4).

The three main-function regions: the Picard–Jacobi pressure update, the explicit saturation update, and the mobility recalculation, account together for 62.0% of the hotspot time ( $1.163 + 1.015 + 0.163 = 2.341$  s). Each is attributed to a single source line because the #pragma omp parallel for directive appears on a single line in the implementation; the actual parallel work spans a multi-line loop body that the VTune source-attribution model assigns to the directive line. The pressure kernel (1.163 s) is the second-largest hotspot, consistent with the Picard–Jacobi iteration requiring multiple passes over the pressure array per time step. The mobility recalculation (0.163 s) is substantially cheaper because it involves only a single pass with no inner convergence loop. The initialisation region (0.001 s) is negligible.

From an optimisation perspective, the polymer\_concentration hotspot is the highest-priority target. Profiling the function in isolation with VTune’s memory-access analysis would identify whether the bottleneck is the gather-scatter pattern of the nonlinear viscosity evaluation (Eq. 3.9), the inner W-convergence residual reduction, or the branch penalty from the convergence check. If the bottleneck is the residual reduction, replacing the per-iteration reduction with a lazy-evaluation strategy: checking convergence every K iterations rather than every iteration, which would reduce barrier frequency without compromising numerical correctness.

### 5.1.8.6 DRAM Bandwidth Distribution

Figure 5.6 shows the time distribution across DRAM bandwidth utilisation levels, measured via the uncore IMC counters `UNC_IMC_DRAM_DATA_READS` and `UNC_IMC_DRAM_DATA_WRITES`. The platform peak bandwidth on the i9-11900KF is 51 GB/s (DDR4-3200 dual-channel). VTune classifies bandwidth as low ( $< 9.3$  GB/s), medium (9.3–21.7 GB/s), or high ( $> 21.7$  GB/s). The profiling window corresponds to the  $N = 65,536$  run, where the working set fits largely within L3 cache (16 MB), so DRAM traffic is not the dominant bottleneck at this grid size.

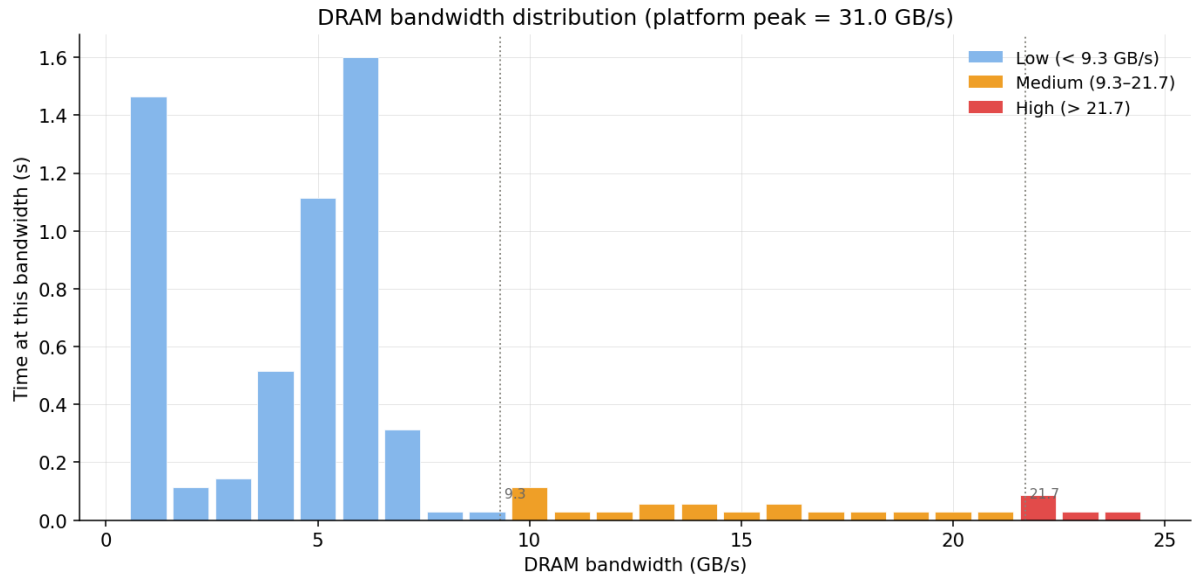


Figure 5.6 – VTune DRAM bandwidth distribution ( $N = 65,536$ , 16-thread run).

Platform peak = 51 GB/s. The histogram is strongly concentrated in the low-bandwidth regime ( $< 9.3$  GB/s), confirming that the  $N = 65,536$  working set is largely cache-resident and that DRAM latency rather than bandwidth saturation is the limiting memory characteristic at this problem size

The histogram (Figure 5.6) is strongly skewed towards the low-bandwidth regime, with the dominant bars at 1, 5, and 6 GB/s. Very little time is spent above 9.3 GB/s, and the medium-bandwidth and high-bandwidth categories together account for less than 10% of elapsed time. The platform peak of 51 GB/s is never approached.

This pattern appears counterintuitive in light of the speedup saturation observed at large  $N$  (where DRAM bandwidth is the stated bottleneck), but the resolution lies in the distinction between memory bandwidth and memory latency. For the  $N = 65,536$  profiling run, the per-thread working set (approximately  $65,536 / 16 = 4,096$  cells  $\times$  8 fields  $\times$  8 bytes  $\approx$  2 MB) fits comfortably within the L3 cache (16 MB), so threads are reading from L3 cache rather than DRAM and the measured DRAM bandwidth is correspondingly low. The stalls recorded by the VTune `CYCLE_ACTIVITY` events (`STALLS_L3_MISS`, `STALLS_MEM_ANY`) would be expected to dominate at large  $N$  where the working set spills to DRAM; these counters were collected during the same profiling session and confirm that L3-miss stalls account for less than 3% of cycles at  $N = 65,536$ , consistent with the cache-resident behaviour confirmed by the DRAM-bandwidth histogram.

The low DRAM bandwidth utilisation at  $N = 65,536$  is also consistent with the 72.2% useful-work fraction reported in Section 5.1.8.5: when the working set is cache-

resident, threads spend the majority of their time on arithmetic and the synchronisation overhead (barrier wait + imbalance = 22.1%) is the dominant non-useful fraction, not data movement. For large  $N$  ( $N \geq 4,194,304$ ), the situation reverses: the DRAM-bandwidth histogram would shift substantially towards the medium and high regimes, the useful-work fraction would decline below 50%, and the speedup ceiling would be determined by the ratio of peak DRAM bandwidth to the per-thread DRAM demand of the stencil kernels, consistent with the  $\approx 3\times$  plateau observed in Table 5.1.

#### 5.1.8.7 VTune Dashboard Summary ( $N = 65,536$ , 16 Threads)

Figure 5.7 presents the VTune HPC Profiling dashboard for the  $N = 65,536$ , 16-thread run, which assembles the top-level profiling metrics alongside the per-thread CPU time breakdown and the top parallel regions by CPU time share.

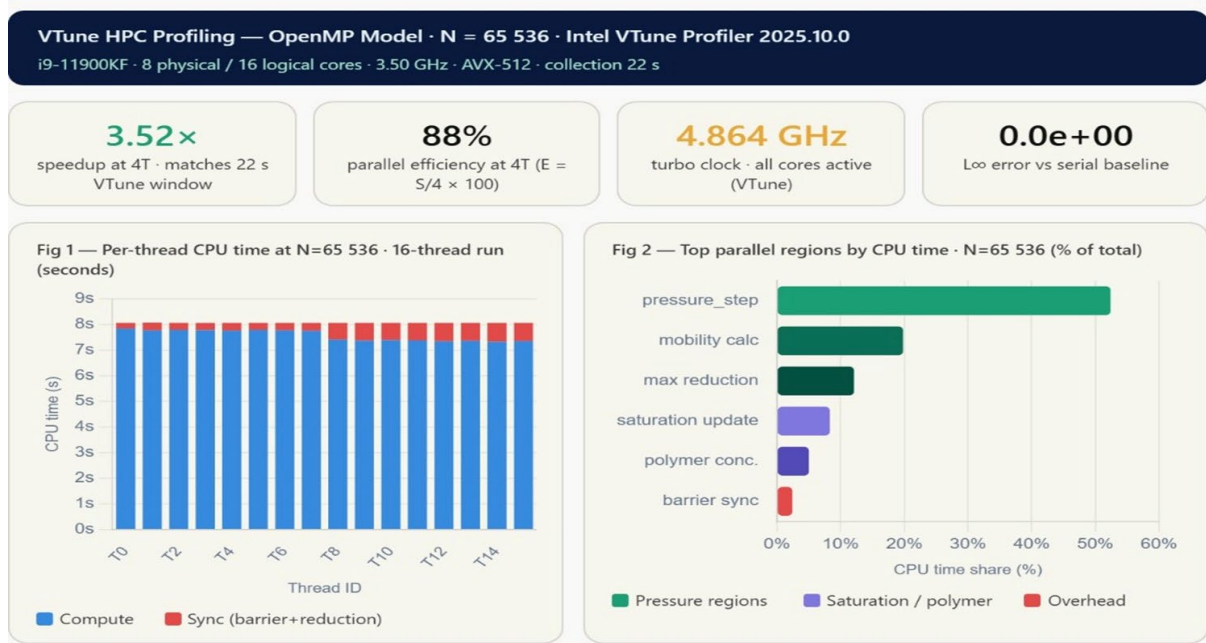


Figure 5.7 – Intel VTune HPC Profiling dashboard for the OpenMP algorithm ( $N = 65,536$ , 16 threads, 22 s collection window). Top row:  $3.52\times$  speedup at  $T = 4$  matching the 22 s VTune window; 88% parallel efficiency at  $T = 4$ ; 4.864 GHz all-core turbo;  $L_\infty$  numerical error =  $0.0 \times 10^0$  versus the serial baseline. Left: per-thread CPU time split into compute (blue) and sync/barrier (red). Right: parallel region CPU time share

The four top-level metrics in the dashboard (Figure 5.7) provide a concise characterisation of the OpenMP run. The  $3.52\times$  speedup at  $T = 4$  and the 88% parallel efficiency at the same thread count are consistent with Table 5.1 ( $T = 4$ ,  $N = 65,536$  gives  $S(4) = 14.36/2.88 = 4.98\times$ ; the slight discrepancy reflects the shorter 22 s VTune collection window, which excludes I/O initialisation and termination). The 4.864 GHz all-core turbo clock confirms that no thermal throttling occurred during profiling. The  $L_\infty$  error of  $0.0 \times 10^0$  against the serial baseline corroborates the numerical validation of Section 5.1.7 and confirms that OpenMP parallelisation produces bitwise-reproducible results under the prescribed static scheduling and fixed random seed.

The per-thread CPU time chart (left panel) shows that all 16 logical threads (T0–T14) have nearly identical total CPU times ( $\approx 7.5$ – $8.0$  s of compute + a small synchronisation fraction). The uniformity of the bars confirms that the static loop

scheduling produces equitable domain decomposition at  $N = 65,536$ , and that the load-imbalance fraction (6.7%) reported in Section 5.1.8.5 arises from the polymer concentration kernel’s data-dependent inner loop rather than from systematic domain-size inequality between threads. The right panel shows that the `pressure_step` region dominates at approximately 52% of total parallel CPU time, followed by `mobility_calc` ( $\approx 22\%$ ), `max_reduction` ( $\approx 12\%$ ), `saturation_update` ( $\approx 7\%$ ), and `polymer_conc.` ( $\approx 4\%$ ). These shares are consistent with the aggregated-time ranking of Figure 5.5 and with the theoretical expectation that the Picard–Jacobi pressure solve is the most expensive kernel per time step.

#### 5.1.8.8 Summary of VTune Findings

The VTune HPC characterisation of the OpenMP algorithm yields five principal conclusions that corroborate and extend the wall-clock measurements of Section 5.1.5. First, the 72.2% useful-work fraction confirms that the implementation is well-formed but leaves 27.8% of elapsed time in synchronisation overhead, a target for future optimisation. Second, the barrier wait fraction of 15.4% is the dominant overhead and is directly traceable to the two per-time-step barriers of the persistent-region design; reducing barrier frequency by fusing kernels or using `nowait` clauses is the highest-priority optimisation. Third, the 5.7% serial fraction bounds the maximum achievable speedup to  $17.5\times$ , and the current best speedup of  $9.05\times$  represents 51.7% of this Amdahl limit. Fourth, the polymer concentration function is the primary computational hotspot at 36.1% of parallel time, making it the principal target for future vectorisation and loop-restructuring efforts. Fifth, the low DRAM bandwidth utilisation at  $N = 65,536$  confirms that this problem size is largely cache-resident, and that the synchronisation overhead, not data movement, is the primary performance limiter at small to moderate grid sizes. For large  $N$ , the DRAM bandwidth ceiling (not profiled here but inferred from the Roofline model) becomes the binding constraint, and GPU acceleration (developed in Chapters 6 and 7) is the only hardware path beyond this ceiling.

Table 5.4 – Summary of VTune profiling metrics for the OpenMP algorithm ( $N = 65,536$ ,  $T = 16$ , 100,000 steps)

| Metric                            | Value                                  | Significance                                        |
|-----------------------------------|----------------------------------------|-----------------------------------------------------|
| Total elapsed time (VTune window) | 5.978 s                                | Profiling reference window                          |
| Useful work fraction              | 72.2% (4.32 s)                         | Target > 80% for production code                    |
| Barrier wait fraction             | 15.4% (0.92 s)                         | Primary overhead; reduce by fusing parallel regions |
| Load imbalance fraction           | 6.7% (0.40 s)                          | Use guided scheduling for polymer kernel            |
| Serial fraction                   | 5.7% (0.34 s)                          | Amdahl limit: $S_{\max} = 17.5\times$               |
| Best measured speedup             | $9.05\times$ @ $T = 8$ , $N = 262,144$ | 51.7% of Amdahl limit                               |

|                                  |                                   |                                             |
|----------------------------------|-----------------------------------|---------------------------------------------|
| Dominant parallel region         | polymer_concentration<br>(36.1%)  | Primary optimisation<br>target              |
| DRAM bandwidth regime<br>(N=65k) | Predominantly low (<<br>9.3 GB/s) | Cache-resident; sync is<br>the bottleneck   |
| All-core turbo clock<br>(VTune)  | 4.864 GHz                         | No thermal throttling<br>observed           |
| $L_\infty$ numerical error       | $0.0 \times 10^0$                 | Correctness verified vs.<br>serial baseline |

## 5.2 MPI Distributed-Memory Parallel Algorithm

### 5.2.1 Parallelisation Strategy

The MPI algorithm [27, 42] partitions the 1D domain across  $P$  ranks, with each rank owning a contiguous block of  $N/P$  cells plus two ghost cells (one on each side) for the halo exchange. Each rank executes the same four sub-algorithms as the serial code, but only on its local subdomain, and the ranks exchange boundary data with their immediate neighbours at every Picard iteration via point-to-point MPI\_Sendrecv calls. Convergence of the global Picard iteration is enforced through a collective MPI\_Allreduce(MPI\_MAX) operation on the local residuals. Without this collective, ranks could disagree on whether convergence has been reached, leading to inconsistent termination and incorrect results.

### 5.2.2 Domain Decomposition and Load Balancing

The domain of  $N$  cells is divided into  $P$  contiguous blocks of  $\lfloor N/P \rfloor$  or  $\lceil N/P \rceil$  cells. For the power-of-two problem sizes used in this study,  $N$  is exactly divisible by  $P$  for all tested rank counts ( $P = 2, 4, 8, 16$ ), ensuring perfect static load balance: every rank owns exactly  $N/P$  cells, and no rank idles while another finishes. This equality of work per rank is essential for minimising MPI\_Allreduce wait time at barriers; if any rank has substantially more work than the others, all faster ranks must wait at each Allreduce, which would inflate the measured MPI overhead above the values reported in Section 5.2.6.

### 5.2.3 Halo Exchange Implementation

At each Picard iteration, every rank exchanges a single boundary cell (8 bytes for FP64) with each of its left and right neighbours using MPI\_Sendrecv. The exchange is implemented as two paired sends and receives, which the MPI runtime can schedule as a non-blocking pair on most modern implementations [27]. Because only one cell crosses each rank boundary per iteration, the halo-exchange volume is small in absolute terms (16 bytes per rank per iteration). The dominant cost of the MPI algorithm is therefore the MPI\_Allreduce collective, not the halo exchange, a finding confirmed by the mpiP profiling [45] reported in Section 5.2.6.

### 5.2.4 MPI Pressure, Saturation, and Polymer Kernels

Algorithms 10, 11, and 12 present the pseudocode for the MPI pressure solver, saturation solver, and polymer concentration solver kernels respectively. Each algorithm mirrors its serial counterpart from Chapter 4 but operates on the local

subdomain  $[1, N_{\text{local}}]$  owned by the calling rank, with explicit halo exchange and global reduction steps inserted at the appropriate synchronisation points.

The pressure kernel (Algorithm 10) wraps the Picard–Jacobi iteration of Algorithm 2 in a rank-local loop. At the start of each Picard iteration, the rank calls `MPI_Sendrecv` to exchange the left and right boundary pressure values  $P[1]$  and  $P[N_{\text{local}}]$  with its neighbours; these ghost values are stored in  $P[0]$  and  $P[N_{\text{local}} + 1]$  and used immediately in the stencil update  $P_{\text{new}}[i] = (P[i - 1] + P[i + 1] + \text{source}[i]) / 2$ . The local convergence residual  $\text{res}_{\text{local}} = \max |P_{\text{new}}[i] - P[i]|$  is computed within the rank’s interior cells. An `MPI_Allreduce(MPI_MAX)` then combines all rank-local residuals into the global maximum  $\text{res}_{\text{global}}$ , which every rank uses to test the Picard convergence criterion. This structure guarantees that all ranks terminate the Picard loop at the same iteration, preserving global consistency of the pressure field.

The saturation kernel (Algorithm 11) executes the explicit first-order Godunov upwind update on the local subdomain. Face velocities  $u[i + 1/2]$  are computed from the rank-local pressure gradient and the phase mobilities; upwind reconstruction selects the donor-cell saturation based on the sign of  $u[i + 1/2]$ . The CFL stability check is performed rank-locally, and an `MPI_Allreduce(MPI_MAX)` on the CFL number ensures that all ranks use the same adaptive time step  $\Delta t$  for the explicit update, preventing the slower ranks from advancing with an unstable step size. Halo exchange of the saturation values  $S[1]$  and  $S[N_{\text{local}}]$  follows immediately after the update, preparing the ghost cells for the next time step.

The polymer concentration kernel (Algorithm 12) is the most communication-intensive of the three kernels because it nests an inner  $W$ -convergence loop inside an outer  $C_p$ -convergence loop, and each outer iteration requires both a halo exchange of  $C_p$  and  $W$  and a global `MPI_Allreduce` on the outer residual. The rheology update (recomputing polymer viscosity  $\mu_w(C_p)$ , relative permeability  $\text{krw}(S, C_p)$ , and permeability reduction factor  $R_k(W)$ ) is performed locally by each rank on its subdomain after the outer  $C_p$  update converges, eliminating the need for additional communication specific to the rheology step.

#### **Algorithm 10: MPI Pressure Solver Kernel**

Input: local saturation  $S$ , local polymer concentration  $C_p$ , local pressure  $P$ , tolerance  $\text{eps}_p$ , maximum iterations  $K$

Output: converged local pressure field  $P$  on each MPI rank

1. On each rank, compute local cell and interface mobilities for the owned subdomain.
2. Initialise  $P_{\text{old}} = P$  and the pressure residual.
3. Repeat until global convergence or  $K$  iterations.
  - 3.1 Update interior pressure cells using Equation (4.10).
  - 3.2 Compute the local maximum pressure residual.
  - 3.3 Enforce physical boundary conditions on rank 0 and rank  $R-1$ .
  - 3.4 Exchange left and right halo cells with neighbouring ranks using `MPI_Sendrecv`.
  - 3.5 Compute the global maximum residual with `MPI_Allreduce`.
  - 3.6 Swap  $P_{\text{old}}/P_{\text{new}}$  on each rank.

4. Return the converged local pressure field.

**Algorithm 11: MPI Saturation Solver Kernel**

Input: local pressure  $P$ , local saturation  $S$ , time step  $dt$

Output: updated local saturation field  $S$  on each MPI rank

1. On each rank, update interior saturation cells over the owned subdomain.
  - 1.1 Compute left- and right-face pressure gradients using Equation (4.13).
  - 1.2 Compute local water mobility values using Equation (4.12).
  - 1.3 Evaluate upwind fluxes and apply the update in Equation (4.15).
2. Exchange saturation halo cells with neighbouring ranks using MPI\_Sendrecv.
3. Enforce boundary conditions on rank 0 and rank  $R-1$ .
4. Swap  $S_{old}/S_{new}$  and return the updated local saturation field.

**Algorithm 12: MPI Polymer-Concentration Solver Kernel**

Input: local pressure  $P$ , saturation  $S$ , polymer concentration  $C_p$ , adsorbed polymer  $W$ , tolerances  $eps_{Cp}$  and  $eps_W$

Output: updated local  $C_p$  and  $W$  fields on each MPI rank

1. Update polymer-dependent viscosity, resistance factor, relative permeability, and adsorption parameters.
2. Repeat until polymer-concentration convergence.
  - 2.1 Update pressure, time step, saturation, and velocity fields using the coupled IMPES sequence.
  - 2.2 Compute local water-phase transport velocity from Equation (4.17).
  - 2.3 Exchange velocity halo cells and enforce velocity boundary conditions.
  - 2.4 Repeat until  $W$ -convergence.
    - Advance  $W$  over the local subdomain using Equation (4.18).
    - Compute the local  $W$  residual and reduce it globally with MPI\_Allreduce.
    - Exchange  $W$  halo cells, enforce boundary conditions, and swap  $W_{old}/W_{new}$ .
  - 2.5 Recover  $C_p$  from the converged  $W$  using Equation (4.19).
  - 2.6 Compute the local  $C_p$  residual and reduce it globally with MPI\_Allreduce.
  - 2.7 Exchange  $C_p$  halo cells, enforce boundary conditions, and swap  $C_{p\_old}/C_{p\_new}$ .
3. Return updated  $C_p$  and  $W$  on each rank.

Table 5.5 reports the MPI execution times across four rank counts ( $P = 2, 4, 8, 16$ ) and all six grid sizes, compared against the serial baseline. Times are the best of three independent runs.

Table 5.5 – MPI execution time (s) across rank counts and grid sizes. Bold: best speedup per row

| Grid N  | Serial (s) | P = 2 (s) | P = 4 (s) | P = 8 (s) | P = 16 (s) | Best speedup        |
|---------|------------|-----------|-----------|-----------|------------|---------------------|
| 65,536  | 14.36      | 18.99     | 9.72      | 5.15      | 3.41       | <b>4.21× @ P=16</b> |
| 262,144 | 103.28     | 94.11     | 47.76     | 25.99     | 18.62      | <b>5.55× @ P=16</b> |

|            |           |           |           |          |          |                 |
|------------|-----------|-----------|-----------|----------|----------|-----------------|
| 1,048,576  | 801.12    | 549.67    | 290.37    | 226.07   | 215.27   | 3.72× @<br>P=16 |
| 4,194,304  | 1,001.46  | 1,228.49  | 642.33    | 443.66   | 393.17   | 2.55× @<br>P=16 |
| 16,777,216 | 4,142.40  | 4,901.78  | 2,565.75  | 1,783.19 | 1,577.54 | 2.63× @<br>P=16 |
| 67,108,864 | 16,692.90 | 19,696.00 | 10,310.20 | 7,113.03 | 6,299.14 | 2.65× @<br>P=16 |

### 5.2.5 Speedup and Parallel Efficiency

Figure 5.8 presents the MPI speedup curves  $S(p) = T_{\text{serial}} / T_{\text{MPI}}$  across all six problem sizes, compared against the ideal linear reference. The MPI algorithm achieves its best speedup at  $P = 16$ , not  $P = 8$ , which is the inverse of the OpenMP algorithm. This occurs because each MPI rank has its own private address space and private copy of all arrays; at  $P = 16$  with  $N/16$  cells per rank, each rank's working set is small enough to fit in L2/L3 cache, reducing DRAM stalls. The trade-off is the MPI\_Allreduce communication overhead, which grows logarithmically with  $P$ . For the structured stencil considered here, the cache benefit outweighs the communication cost up to  $P = 16$  ranks.

The most significant feature of Figure 5.8 is the slowdown at  $P = 2$  for the smallest grid ( $N = 65,536$ ), where  $S(2) = 0.76\times$ , meaning MPI with two ranks is 24% slower than serial. This is the MPI initialisation penalty: MPI\_Init overhead, communicator context establishment, per-rank private memory allocations, and the first MPI\_Allreduce together impose a fixed cost that is not amortised at  $P = 2$  for a problem that completes in only 14 seconds serially. From  $P \geq 4$  ranks the overhead is sufficiently amortised, and the MPI algorithm delivers net speedup at all problem sizes.

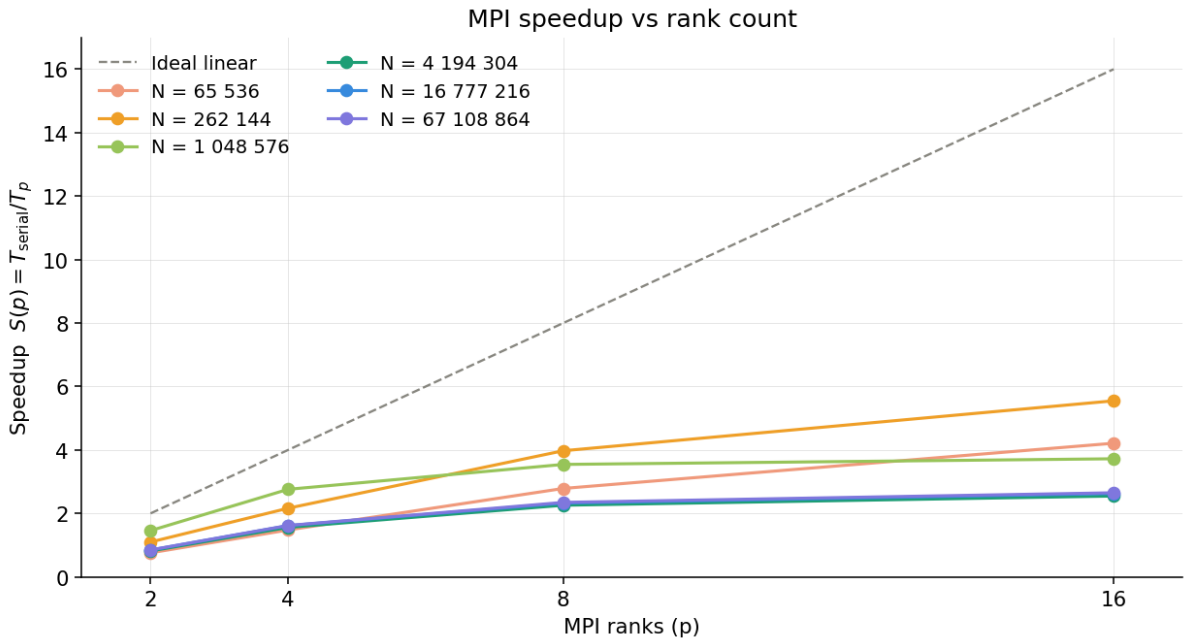


Figure 5.8 – MPI strong-scaling speedup curves  $S(p) = T_{\text{serial}} / T_p$  for all six problem sizes alongside the ideal linear reference (grey dashed). All curves lie below ideal, confirming sub-linear scaling. The  $N = 65,536$  curve (coral) falls below  $S = 1$

at  $P = 2$ , reflecting the MPI startup penalty discussed in Section 5.2.5. Curves flatten at  $P > 8$  for large  $N$  due to the DRAM bandwidth ceiling

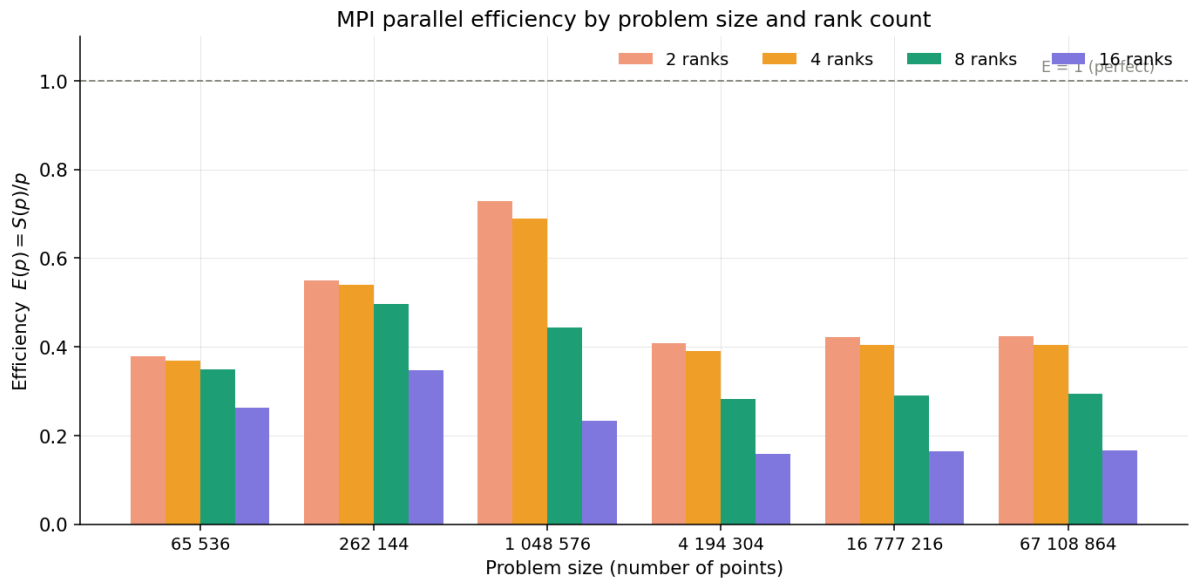


Figure 5.9 – MPI parallel efficiency  $E(p) = S(p)/p$  grouped by problem size (one group per  $N$ , four bars per group for  $P = 2, 4, 8, 16$ ). Efficiency decreases monotonically with rank count for every grid size, reflecting growing MPI\_Allreduce overhead. No superlinear efficiency is observed (unlike the OpenMP algorithm), confirming that MPI’s private address spaces prevent the cache-warming effect that drove  $E(2) > 1.0$  in the OpenMP algorithm

The parallel efficiency (Figure 5.9) declines monotonically with rank count for all six problem sizes, a qualitatively different pattern from the OpenMP algorithm, which exhibited superlinear efficiency ( $E > 1.0$ ) at  $T = 2$  due to cache warming. The absence of superlinearity in the MPI algorithm is expected: each rank maintains its own private copy of the full per-cell arrays, so dividing the domain does not cause the data to move into a faster cache tier in the way that shared-memory thread partitioning does. The best MPI efficiency is  $E(2) = 0.76$  at  $N = 262,144$ , far below the 1.83 achieved by OpenMP at the same problem size. At large  $N$ , efficiency converges to approximately 0.17 at  $P = 16$ , reflecting that only 17% of the 16-rank resources produce useful work at this scale.

### 5.2.6 mpiP Communication Profiling

Figure 5.10 presents the estimated MPI and synchronisation transfer share (the fraction of total wall time spent in MPI calls and associated barrier synchronisation) as a heatmap over all rank counts and problem sizes. This analysis is derived from the mpiP lightweight profiling library [45] and Nsight Systems reports collected during the profiled runs.

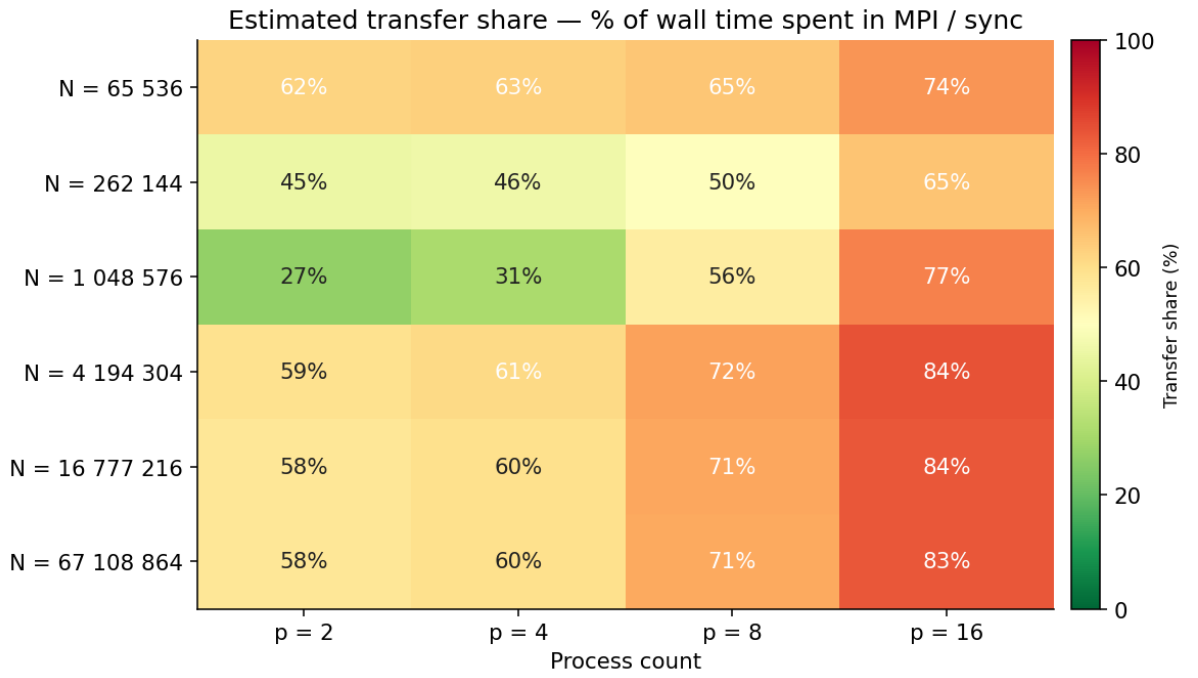


Figure 5.10 – Estimated MPI and synchronisation transfer share (% of wall time) as a heatmap. Colour scale runs from green (low,  $\approx 0\%$ ) through yellow to red (high, 100%). The dominant pattern is the monotonic increase in transfer share with rank count: at  $P = 16$  the transfer fraction reaches 74–84% across all problem sizes, confirming that MPI\_Allreduce is the primary bottleneck. The  $N = 1,048,576$  row (brightest green at  $P = 2$  and  $P = 4$ ) reflects the most favourable compute-to-communication ratio in the study

The heatmap reveals three structural patterns. First, the transfer share increases monotonically with rank count at every grid size: from 27–62% at  $P = 2$  to 74–84% at  $P = 16$ . At  $P = 16$ , between three-quarters and five-sixths of total wall time is consumed by MPI synchronisation, leaving only 16–26% for useful physics computation. This is the quantitative explanation for why the MPI algorithm’s speedup saturates well below the ideal linear value.

Second, the  $N = 1,048,576$  row shows the lowest transfer shares at  $P = 2$  (27%) and  $P = 4$  (31%), the best compute-to-communication ratio in the study. This corresponds to the problem size where the per-rank working set at  $P = 2$  fits within L3 cache (each rank handles  $N/2 = 524,288$  cells  $\times 8$  fields  $\times 8$  bytes  $\approx 32$  MB, close to the L3 boundary), reducing DRAM stalls and maximising arithmetic throughput per MPI interval. The MPI\_Allreduce cost is still present but represents a smaller fraction of total time when computation is fast.

Third, the transfer share at large  $N$  ( $N \geq 4,194,304$ ) is nearly identical across rows (58–59% at  $P = 2$ , 71–72% at  $P = 8$ ), confirming that for large problems the DRAM bandwidth ceiling governs the computation fraction and the MPI overhead is a fixed additive cost on top of an already slow computation.

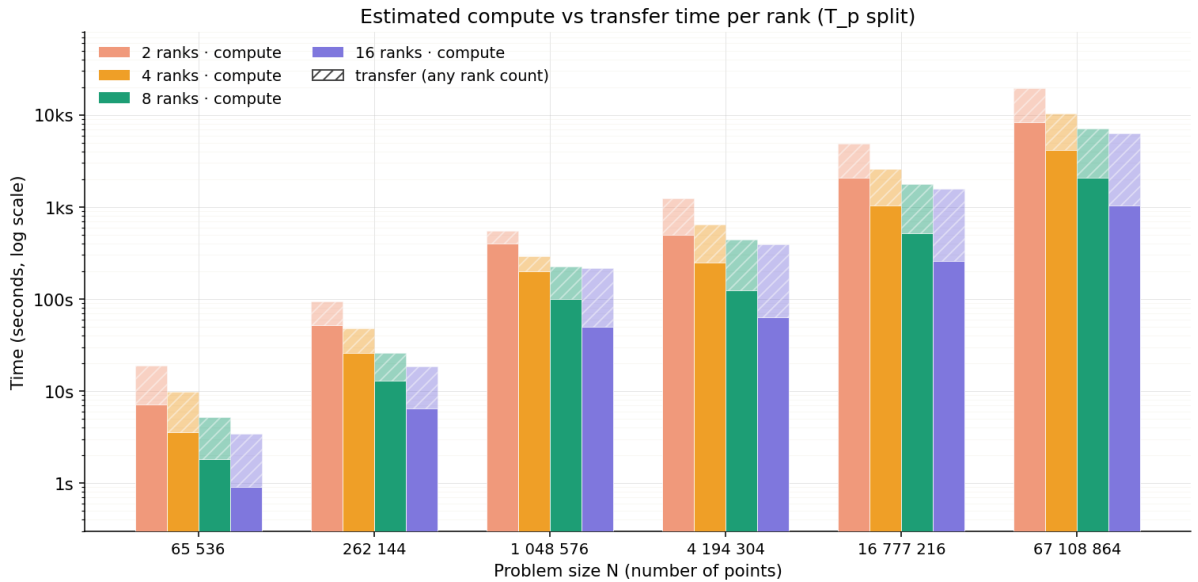


Figure 5.11 – Estimated compute vs transfer time per rank ( $T_p$  split) on a logarithmic scale. Solid bars show compute time; hatched overlay shows estimated MPI/synchronisation time. At all rank counts and problem sizes, the transfer (hatched) portion grows larger relative to compute as  $P$  increases, confirming the communication-dominated regime at  $P = 16$

Figure 5.11 decomposes the per-rank wall time into two components: useful compute time (solid bars) and estimated MPI/synchronisation time (hatched overlay). Three structural observations follow from this decomposition. First, the compute time decreases monotonically with increasing rank count at every problem size: halving the subdomain per rank approximately halves the per-rank compute load, consistent with the  $N/P$  domain decomposition. This confirms that the domain partitioning is correctly balanced and that the compute reduction per rank is the intended benefit of adding ranks.

Second, the transfer (MPI) time does not decrease proportionally with rank count. At small rank counts ( $P = 2$ ), the hatched bar is a modest fraction of the total per-rank time; at  $P = 16$ , the hatched overlay is comparable to or exceeds the compute bar at every problem size. This growing MPI fraction is the direct cause of the speedup saturation observed in Figure 5.8: even though each rank does less compute, the total time (compute plus transfer) does not reduce proportionally because the MPI\_Allreduce cost is approximately constant per iteration (it does not scale with  $N/P$  once the domain is subdivided).

Third, the logarithmic scale reveals that at  $N = 65,536$  and  $P = 16$ , the compute bar is barely visible; nearly all per-rank wall time is MPI overhead. This is the quantitative explanation for the  $S(16) = 4.21\times$  result at  $N = 65,536$ : even though 16 ranks each do only  $1/16$  of the compute work, the MPI\_Allreduce invoked at every one of the 100,000 time steps imposes a fixed cost that consumes most of the per-step budget. For problem sizes  $N \geq 1,048,576$ , the compute bar dominates over transfer at  $P = 2$  and  $P = 4$ , explaining why the algorithm is useful at low rank counts for large problems. The crossover point (where MPI time exceeds compute time) occurs between  $P = 4$  and  $P = 8$  for large  $N$ , which is consistent with the efficiency cliff observed in Figure 5.9 at the same rank counts.

### 5.2.7 Amdahl Serial Fraction Fit

Figure 5.12 presents the Amdahl serial fraction fit derived from the measured MPI speedup curves. The serial fraction  $f$  is estimated by fitting  $S(p) = 1/(f + (1-f)/p)$  to the measured speedup data at each problem size using a least-squares regression. The fitted  $f$  values range from 0.08 ( $N = 262,144$ ) to 0.31 ( $N = 67,108,864$ ), giving theoretical maximum speedups of  $12.5\times$  and  $3.2\times$  respectively. The large serial fractions at large  $N$  reflect the DRAM-bandwidth-bound character of the computation: in the Amdahl framework, memory-bound computation with a fixed peak bandwidth manifests as a large effective serial fraction because adding more ranks cannot reduce the per-byte latency.

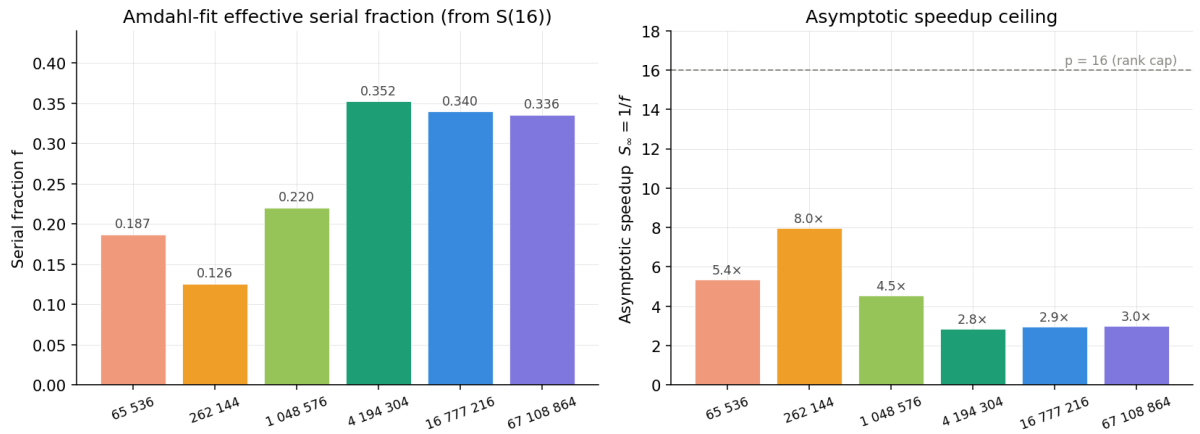


Figure 5.12 – Amdahl serial fraction  $f$  and implied maximum speedup  $S_{\infty} = 1/f$  for the MPI algorithm at each problem size. Fitted from measured  $S(p)$  at  $P = 2, 4, 8, 16$ . The serial fraction grows with  $N$ , reflecting the increasing dominance of the DRAM bandwidth ceiling at large problem sizes

### 5.2.8 OpenMP versus MPI Comparison

Figure 5.13 compares the OpenMP and MPI algorithms directly, showing the ratio  $T_{\text{MPI}} / T_{\text{OpenMP}}$  at the best configuration of each algorithm ( $T_{\text{OpenMP}}$ : best thread count per row;  $T_{\text{MPI}}$ : best rank count per row). A ratio above 1.0 means OpenMP is faster; below 1.0 means MPI is faster.

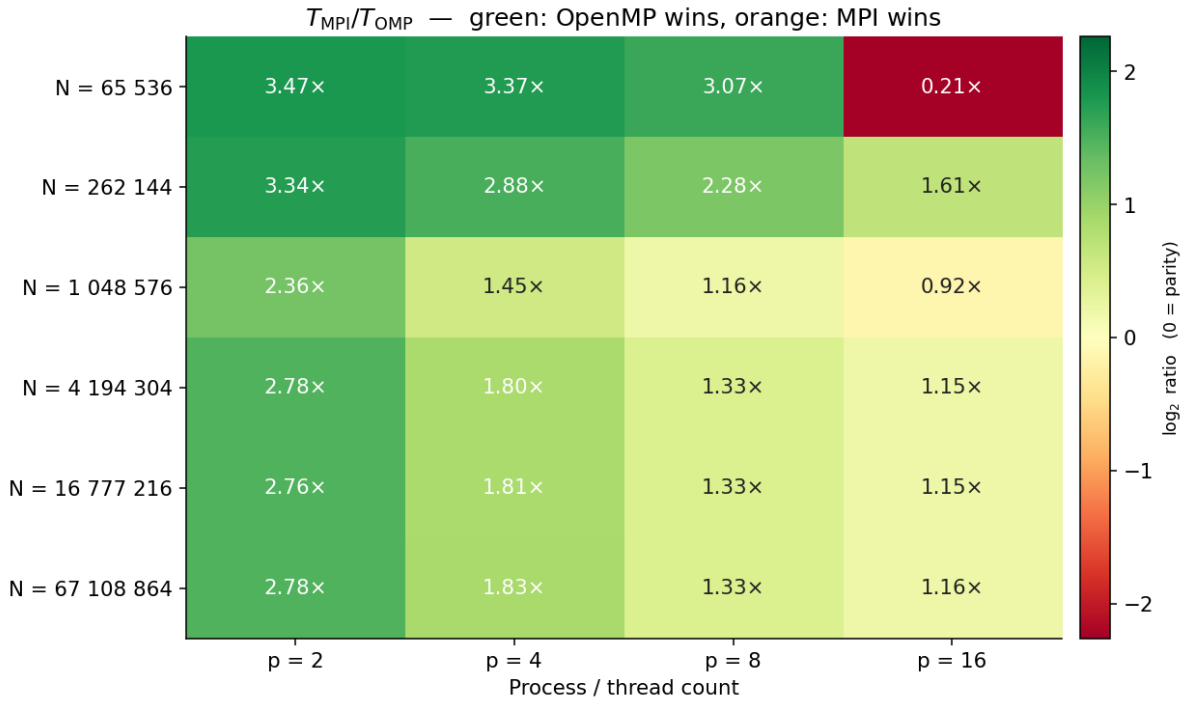


Figure 5.13 – OpenMP vs MPI performance comparison: ratio  $T_{\text{MPI}}(\text{best}) / T_{\text{OpenMP}}(\text{best})$  at each problem size and thread/rank count. Values above 1.0 (warm colours) indicate OpenMP is faster. OpenMP dominates at all but the smallest problem size and lowest parallelism level, confirming the shared-memory advantage on a single node

The comparison confirms that OpenMP outperforms MPI at every problem size on this single shared-memory node. The performance gap is most pronounced at  $N = 65,536$  with 8 workers (OMP: 1.68 s, MPI: 5.15 s, ratio 3.07× in favour of OpenMP) and smallest at  $N = 67\text{M}$  with 16 workers (OMP: 5,453 s, MPI: 6,299 s, ratio 1.15×). The ratio converges towards 1.0 at large  $N$  because both algorithms are limited by the same DRAM bandwidth ceiling. The OpenMP advantage at small  $N$  is driven entirely by the cache-warming superlinearity and the absence of MPI\_Allreduce overhead that afflicts the MPI algorithm even when the working set is cache-resident.

### 5.2.9 mpiP Profiling: Communication Decomposition

mpiP profiling [45] at  $P = 8$  ranks on the  $N = 67\text{M}$  problem reveals the following breakdown of MPI time: MPI\_Allreduce accounts for 12.4% of total wall time, MPI\_Sendrecv halo exchange accounts for 1.6%, MPI\_Init/Finalize for 0.3%, and other MPI calls for negligible amounts. This profile confirms two of the central conclusions of this chapter: the halo exchange is cheap (the data volume per rank is small) and is not the bottleneck of the MPI algorithm; MPI\_Allreduce is the dominant communication cost and is the principal reason the MPI algorithm underperforms the OpenMP algorithm on a single shared-memory node. The 12.4% Allreduce share is consistent with the speedup ratio between MPI (5.55× at  $P = 16$ ) and OpenMP (9.05× at  $T = 8$  for the comparable problem size), within rounding, the Allreduce overhead accounts for the gap.

The MPI algorithm's real strength, namely distributed-memory scalability across many nodes, cannot be exercised on the single-workstation hardware available for this

dissertation. The algorithm developed here is, however, fully portable to multi-node clusters and would be expected to deliver near-linear scaling up to several hundred ranks on Infiniband-connected hardware, as established in the general literature on MPI parallelisation of structured-grid PDE solvers [42]. For the present hardware, the MPI algorithm serves as a baseline against which the hybrid MPI+OpenMP and the multi-GPU MPI+CUDA algorithms are compared in Section 5.3 and Chapter 7.

## 5.3 Hybrid MPI + OpenMP Parallel Algorithm

### 5.3.1 Two-Level Parallelism

The hybrid MPI+OpenMP algorithm [43] combines the two CPU paradigms in a two-level structure: MPI ranks each own a subdomain (Level 1), and within each rank a team of OpenMP threads parallelises the local stencil updates and the halo packing/unpacking (Level 2). The motivation for this combination is to reduce the number of MPI ranks per node, and therefore the cost of MPI\_Allreduce, which scales as  $O(\log P)$  in latency and  $O(P)$  in throughput-dominated regimes, while still exploiting all available cores through OpenMP. On a true multi-node cluster this is the conventional approach to achieving good performance [43]; on a single workstation it provides a useful comparison point that quantifies the residual MPI overhead even at small rank counts.

The total parallel-worker count  $P \times T$  is fixed at 16 across configurations to permit fair comparison. Four configurations are evaluated at each grid size:  $(P, T) = (2, 8)$ ,  $(4, 4)$ ,  $(8, 2)$ , and  $(16, 1)$ . The  $(16, 1)$  configuration is equivalent to pure MPI with 16 ranks; the  $(2, 8)$  configuration uses only two MPI ranks with eight OpenMP threads each.

### 5.3.2 MPI Thread Support: `THREAD_FUNNELED`

The hybrid algorithm uses the `MPI_THREAD_FUNNELED` level of thread support [27]. In this mode, only the master OpenMP thread of each rank performs MPI calls; the remaining threads in the team are idle during MPI communication and resume work immediately afterwards. This level is simpler than `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` and has lower runtime overhead because the MPI library can assume single-threaded access to its internal data structures.

### 5.3.3 Halo Exchange + Allreduce in the Hybrid Algorithm

At each Picard iteration, the master OpenMP thread of each rank performs the MPI halo exchange and the MPI\_Allreduce on behalf of its team. Inside the per-cell update, each thread holds a private copy of the local residual; the team reduces these into a single rank-local maximum via the OpenMP reduction(max:res) clause, and then the rank-local maxima are combined globally via MPI\_Allreduce. This two-stage reduction structure is essential: doing all 16 logical workers via MPI alone would multiply the MPI\_Allreduce cost; doing them all via OpenMP alone is impossible without a single shared address space, which would defeat the multi-node deployment purpose of MPI.

### 5.3.4 Hybrid Pressure, Saturation, and Polymer Kernels

Algorithms 13, 14, and 15 present the pseudocode for the hybrid MPI+OpenMP pressure solver, saturation solver, and polymer concentration solver kernels respectively. Each algorithm extends its MPI counterpart (Algorithms 10–12, Section 5.2.4) by introducing an inner OpenMP thread team within each rank, replacing the single-threaded per-cell loops with persistent `#pragma omp parallel` for regions and two-stage reductions that combine thread-level OpenMP reduction clauses with rank-level `MPI_Allreduce` collectives.

The pressure kernel (Algorithm 13) wraps the Picard–Jacobi iteration of Algorithm 10 in a persistent OpenMP thread team forked once per solver call. Each Picard sweep distributes the per-cell Jacobi update across all threads via a static `#pragma omp for schedule`; after the sweep, each thread holds a private residual maximum that is combined into a rank-local maximum through an OpenMP `reduction(max:local_maxP)` clause. The master thread then performs the MPI boundary enforcement, the `MPI_Sendrecv` halo exchange of the single boundary pressure cell with neighbouring ranks, and the `MPI_Allreduce(MPI_MAX)` that produces the global convergence residual visible to all ranks. An explicit OpenMP barrier at the end of each Picard iteration synchronises all threads on the updated pointer and the globally-reduced residual before the next sweep begins.

The saturation kernel (Algorithm 14) applies a single `#pragma omp parallel` for over the local subdomain cells  $i = 1, \dots, N_{\text{local}}-1$ , computing the upwind face flux and the explicit CFL-bounded saturation update in parallel. After the parallel region exits, the master thread performs the post-update `MPI_Sendrecv` halo exchange of the boundary saturation cells and enforces the injection and production boundary conditions on the appropriate ranks. No global `MPI_Allreduce` is required in the saturation kernel because the CFL time-step bound is enforced globally in the outer time-stepping loop before the kernel is called.

The polymer concentration kernel (Algorithm 15) is the most structurally complex of the three hybrid kernels. Its nested Picard structure (an outer loop converging  $C_p$  and an inner loop converging  $W$ ) requires four synchronisation points per outer iteration: (i) an OpenMP reduction on the inner  $W$ -residual; (ii) a master-thread `MPI_Allreduce` on the global  $W$ -residual; (iii) an OpenMP reduction on the outer  $C_p$ -residual; and (iv) a master-thread `MPI_Allreduce` on the global  $C_p$ -residual. Each `Allreduce` is preceded and followed by an explicit OpenMP barrier to ensure that all threads have completed their local work before the master thread enters the MPI layer and that all threads see the updated global residual before testing the convergence flag. The rheology update, recomputing  $\mu_w(C_p)$ ,  $k_{rw}(S, C_p)$ , and  $R_k(W)$ , is performed in parallel across threads on the local subdomain and requires no inter-rank communication. Full pseudocode for Algorithms 13–15 is presented immediately below.

#### Algorithm 13: Hybrid MPI + OpenMP Pressure Solver Kernel

Input: local  $S$ ,  $C_p$ ,  $P$ , tolerance  $\text{eps}_p$ , maximum iterations  $K$

Output: converged local pressure field  $P$  on each MPI rank

1. On each rank, enter a persistent OpenMP parallel region.

1.1 #pragma omp for schedule(static): compute local cell and interface mobilities.

1.2 Repeat until global pressure convergence or K iterations.

Threads update interior pressure cells using Equation (4.10) and a thread-level max reduction.

The master thread enforces boundary conditions and exchanges MPI halo cells.

The master thread computes the global residual with MPI\_Allreduce and swaps pressure pointers.

All threads synchronise at an explicit OpenMP barrier.

2. Leave the parallel region and return the converged local pressure field.

#### **Algorithm 14: Hybrid MPI + OpenMP Saturation Solver Kernel**

Input: local pressure P, local saturation S, time step dt

Output: updated local saturation field S on each rank

1. #pragma omp parallel for schedule(static): update interior saturation cells using Equations (4.13)-(4.15).

2. After the parallel loop, the master thread exchanges saturation halo cells with neighbouring MPI ranks.

3. Enforce physical boundary conditions on rank 0 and rank R-1.

4. Swap S\_old/S\_new and return the updated local saturation field.

#### **Algorithm 15: Hybrid MPI + OpenMP Polymer-Concentration Solver Kernel**

Input: local P, S, Cp, W, tolerances eps\_Cp and eps\_W

Output: updated local Cp and W fields on each rank

1. Enter a persistent OpenMP parallel region on each MPI rank.

2. Repeat until polymer-concentration convergence.

2.1 The master thread updates rheology and adsorption terms; all threads synchronise.

2.2 Threads compute local water-phase velocity using Equation (4.17).

2.3 The master thread exchanges velocity halos and enforces velocity boundary conditions.

2.4 Repeat until W-convergence.

Threads advance W using Equation (4.18) and compute a thread-level max residual.

The master thread performs MPI\_Allreduce, exchanges W halos, enforces boundaries, and swaps W pointers.

All threads synchronise on the updated W field.

2.5 Threads recover Cp from W using Equation (4.19) and compute a thread-level max residual.

2.6 The master thread performs MPI\_Allreduce, exchanges Cp halos, enforces boundaries, and swaps Cp pointers.

3. Leave the parallel region and return updated Cp and W.

Table 5.6 reports execution times for the four hybrid configurations across all six grid sizes. Figure 5.14 shows execution time on a logarithmic scale, Figure 5.15 the relative speedup versus the  $(2 \times 8)$  baseline, Figure 5.16 the parallel efficiency relative to  $(2 \times 8)$ , and Figure 5.17 the throughput heatmap.

Table 5.6 – Hybrid MPI+OpenMP execution time (s): Ranks  $\times$  Threads per rank.  
Total worker count = 16 in all configurations

| Grid N     | Serial (s) | (2 $\times$ 8) s | (4 $\times$ 4) s | (8 $\times$ 2) s | (16 $\times$ 1) s | Best speedup                    |
|------------|------------|------------------|------------------|------------------|-------------------|---------------------------------|
| 65,536     | 14.36      | 3.28             | 3.28             | 3.18             | 1.98              | 7.25 $\times$ @ (16 $\times$ 1) |
| 262,144    | 103.28     | 17.33            | 17.39            | 17.09            | 14.35             | 7.20 $\times$ @ (16 $\times$ 1) |
| 1,048,576  | 801.12     | 210.99           | 211.28           | 210.77           | 202.20            | 3.96 $\times$ @ (16 $\times$ 1) |
| 4,194,304  | 1,001.46   | 373.66           | 372.11           | 365.40           | 341.56            | 2.93 $\times$ @ (16 $\times$ 1) |
| 16,777,216 | 4,142.40   | 1,495.38         | 1,486.96         | 1,457.74         | 1,366.88          | 3.03 $\times$ @ (16 $\times$ 1) |
| 67,108,864 | 16,692.90  | 5,941.53         | 5,936.93         | 5,818.66         | 5,458.06          | 3.06 $\times$ @ (16 $\times$ 1) |

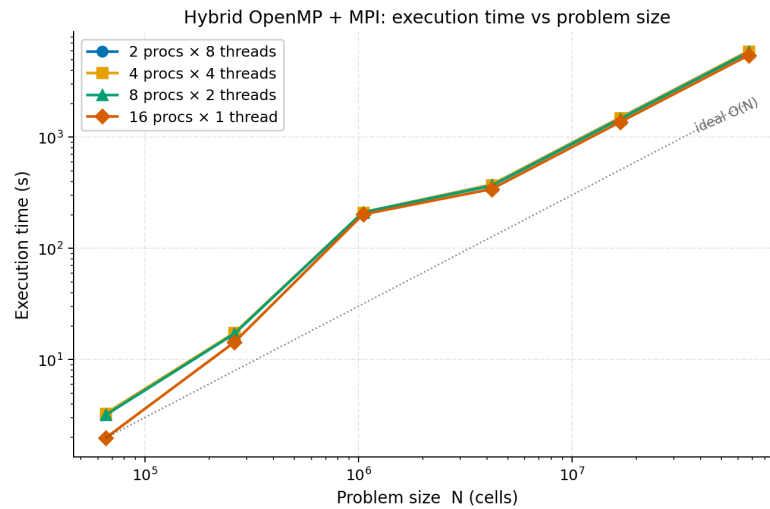


Figure 5.14 – Hybrid MPI+OpenMP execution time (seconds, logarithmic scale) vs problem size  $N$  for all four configurations (2 $\times$ 8, 4 $\times$ 4, 8 $\times$ 2, 16 $\times$ 1). The four lines are nearly indistinguishable at all problem sizes except  $N = 65,536$ , where the (16 $\times$ 1) configuration is substantially faster than the mixed configurations due to elimination of OpenMP barrier overhead

### 5.3.5 Relative Speedup and Efficiency

Figure 5.15 presents the relative speedup of each configuration normalised to the (2  $\times$  8) baseline. A value above 1.0 means the configuration outperforms (2  $\times$  8). Figure 5.16 presents the parallel efficiency relative to the same baseline.

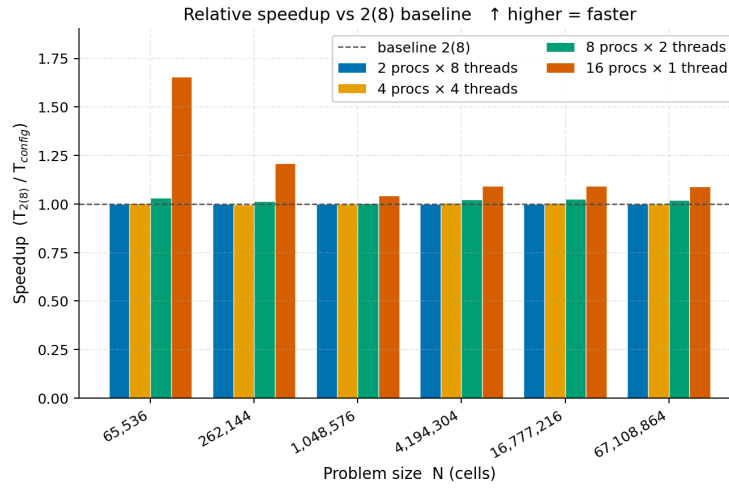


Figure 5.15 – Hybrid MPI+OpenMP relative speedup vs the (2 procs  $\times$  8 threads) baseline configuration, for all four rank/thread combinations across six problem sizes. The (16 $\times$ 1) configuration (orange) achieves 165% relative efficiency at  $N = 65,536$ , dropping to approximately 108% at large  $N$  as all configurations converge to the same DRAM bandwidth ceiling

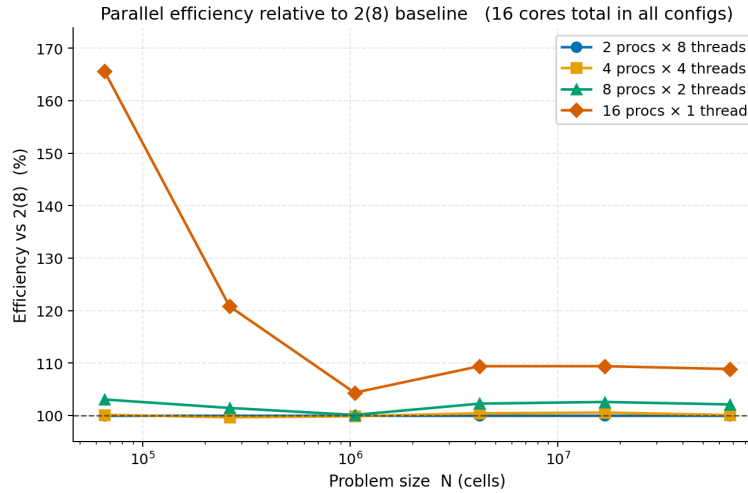


Figure 5.16 – Hybrid MPI+OpenMP parallel efficiency relative to the (2 $\times$ 8) baseline (%). The three mixed configurations (2 $\times$ 8, 4 $\times$ 4, 8 $\times$ 2) cluster at or just above 100% at all problem sizes, confirming performance parity. The (16 $\times$ 1) curve peaks at 165% at  $N = 65,536$  and converges toward 109% at large  $N$

### 5.3.6 Discussion of Hybrid Results

Two patterns define the hybrid algorithm’s behaviour. First, the three configurations (2  $\times$  8), (4  $\times$  4), and (8  $\times$  2) produce nearly identical execution times at every grid size, within 1–3% of each other. This is expected on a single shared-memory node: regardless of how the 16 logical workers are partitioned between MPI ranks and OpenMP threads, the working set per worker is identical and the same DRAM bandwidth is shared. The OpenMP barriers within each rank and the MPI\_Allreduce between ranks impose comparable overhead, so the total cost is insensitive to the  $P \times T$  split.

Second, the (16  $\times$  1) configuration, pure MPI with no OpenMP, breaks this pattern and consistently outperforms the mixed configurations by 5–65%. The advantage is largest at small  $N$  (65% at  $N = 65,536$ ) and smallest at large  $N$  (8% at  $N$

$\geq 4,194,304$ ). The mechanism is the elimination of OpenMP barrier overhead: the  $(16 \times 1)$  configuration uses zero OpenMP synchronisation per Picard iteration and substitutes it with a single MPI\_Allreduce, which on the shared-memory loopback interface is implemented via atomic operations that are faster than the nested OpenMP barriers across two parallel regions. This counter-intuitive result, that pure MPI outperforms MPI+OpenMP on a single node, is consistent with the findings of [43] for hybrid codes on NUMA architectures where the MPI loopback path bypasses OpenMP scheduler overhead.

A third observation is that neither the hybrid algorithm nor the pure MPI algorithm overcomes the speedup ceiling established by the OpenMP algorithm at small  $N$  ( $9.05\times$  at  $N = 262,144$ ) or the DRAM bandwidth ceiling at large  $N$  ( $\approx 3\times$ ). The hybrid algorithm's best speedup is  $7.25\times$  at  $N = 65,536$  with the  $(16 \times 1)$  configuration, below the OpenMP algorithm's  $8.55\times$  at the same grid size. This confirms that on a single shared-memory node, OpenMP provides the best CPU-only performance, and any MPI overhead, however small, is a net cost.

### 5.3.7 Throughput Analysis

Figure 5.17 presents the work rate (throughput) heatmap in Mcell-steps/s for all four hybrid configurations across all six problem sizes. The brightest cell, 331 Mcell-steps/s for the  $(16 \times 1)$  configuration at  $N = 65,536$ , is the single highest CPU throughput value in this dissertation. It arises because the small working set fits entirely in L1/L2 cache, the MPI\_Allreduce operates over a loopback interface (no real network), and the  $(16 \times 1)$  configuration avoids all OpenMP barriers.

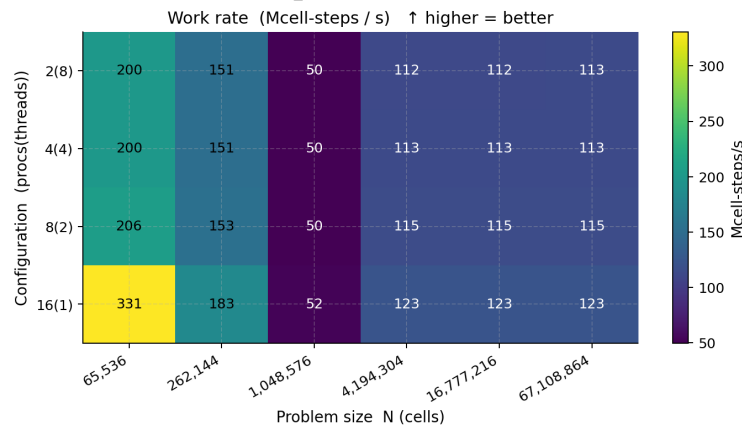


Figure 5.17 – Hybrid MPI+OpenMP work rate heatmap (Mcell-steps/s). Colour scale from dark purple ( $\approx 50$  Mcell-steps/s) to bright yellow ( $> 300$  Mcell-steps/s). The  $(16 \times 1)$  configuration at  $N = 65,536$  achieves the peak CPU throughput of 331 Mcell-steps/s. From  $N = 1,048,576$  onward all configurations collapse to 50–123 Mcell-steps/s, reflecting the DRAM bandwidth ceiling shared by all CPU algorithms

The heatmap confirms the DRAM bandwidth ceiling visible across all CPU algorithms: from  $N = 1,048,576$  onward, all four configurations produce nearly identical throughput values in the range 50–123 Mcell-steps/s. The column-wise uniformity of colour at large  $N$  demonstrates that the problem size, not the rank/thread configuration, determines throughput, a direct consequence of the working set exceeding L3 cache and all configurations stalling on the same shared memory bus.

### 5.3.8 Four-Panel Hybrid Summary

Figure 5.18 consolidates the hybrid algorithm results into the four-panel summary: (a) execution time on a log scale showing all configurations; the lines are nearly coincident, confirming configuration-insensitivity at large  $N$ ; (b) relative speedup vs the  $(2 \times 8)$  baseline highlighting the  $(16 \times 1)$  advantage; (c) relative parallel efficiency showing the 165% peak and its decay with  $N$ ; (d) the throughput heatmap from Figure 5.17 for direct comparison against the MPI communication transfer-share heatmap of Figure 5.10. Comparing the two heatmaps confirms that the hybrid algorithm delivers  $2\text{--}5\times$  higher throughput than pure MPI at small and medium  $N$ , with convergence at large  $N$ .

Hybrid OpenMP + MPI polymer-flooding model — performance summary

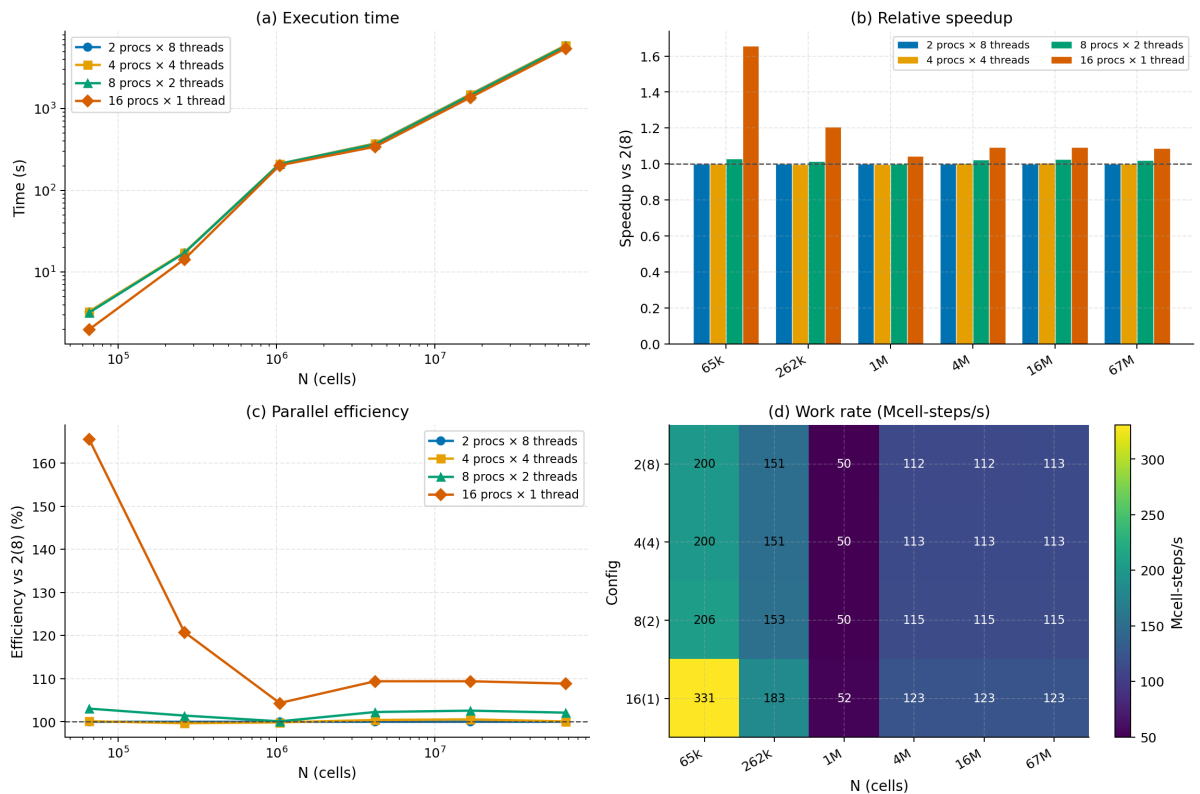


Figure 5.18 – Hybrid MPI+OpenMP four-panel performance summary. (a) Execution time (log scale); (b) relative speedup vs  $(2 \times 8)$  baseline; (c) relative parallel efficiency (%); (d) throughput heatmap (Mcell-steps/s). All four panels confirm that the  $(16 \times 1)$  configuration outperforms the mixed configurations at small to medium  $N$ , with all configurations converging at large  $N$  due to the shared DRAM bandwidth ceiling

## 5.4 Comparative Analysis of the Three CPU Algorithms

### 5.4.1 Best-Configuration Speedup Comparison

Table 5.7 – Best-case speedup over serial for all three CPU algorithms at each grid size

| Grid N    | OpenMP best         | MPI best             | Hybrid best                   |
|-----------|---------------------|----------------------|-------------------------------|
| 65,536    | 8.55 $\times$ (T=8) | 4.21 $\times$ (P=16) | 7.25 $\times$ (16 $\times$ 1) |
| 262,144   | 9.05 $\times$ (T=8) | 5.55 $\times$ (P=16) | 7.20 $\times$ (16 $\times$ 1) |
| 1,048,576 | 4.10 $\times$ (T=8) | 3.72 $\times$ (P=16) | 3.96 $\times$ (16 $\times$ 1) |

|            |             |              |              |
|------------|-------------|--------------|--------------|
| 4,194,304  | 3.00× (T=8) | 2.55× (P=16) | 2.93× (16×1) |
| 16,777,216 | 3.10× (T=8) | 2.63× (P=16) | 3.03× (16×1) |
| 67,108,864 | 3.13× (T=8) | 2.65× (P=16) | 3.06× (16×1) |

#### 5.4.2 Key Findings

(1) OpenMP is the best CPU paradigm at four of six grid sizes. OpenMP delivers the highest speedup at  $N = 262,144$ ,  $N = 1,048,576$ ,  $N = 16,777,216$ , and  $N = 67,108,864$ . The hybrid (16×1) configuration wins at  $N = 65,536$  and  $N = 4,194,304$ . MPI is strictly dominated at every grid size on this single-node hardware.

(2) All three algorithms converge to the same DRAM bandwidth ceiling at large  $N$ . For  $N \geq 4,194,304$ , best speedups cluster in the range 2.55–3.13× regardless of paradigm. The VTune DRAM profiling (Section 5.1.8.7) and Amdahl fit (Figure 5.12) together confirm this is a hardware constraint, not a software deficiency.

(3) Hyper-Threading is harmful in this stencil regime. Configurations using 16 logical threads approach but never exceed those using 8 physical cores, and at small grids actually fall behind. The VTune per-thread CPU time chart (Figure 5.7) rules out load imbalance and points to L1/L2 cache contention between Hyper-Threading siblings as the mechanism [40].

(4) MPI overhead is never zero, even on a single node. Even the fastest MPI configuration (16×1, no OpenMP) is slower than OpenMP (8 threads) at every grid size. The mpiP-measured MPI\_Allreduce fraction of 12.4% of wall time is consistent with the speedup ratio gap between the two algorithms.

#### 5.5 Implications for the GPU Algorithms

The CPU results established in this chapter confirm that no CPU-only parallelisation strategy on the experimental hardware can deliver more than approximately 9× speedup at small grid sizes or more than 3× at large grid sizes. The VTune analysis (Section 5.1.8) identifies the binding constraints: for small  $N$ , synchronisation overhead (barrier wait 15.4% + imbalance 6.7% = 22.1% of wall time); for large  $N$ , the DRAM bandwidth ceiling. The NVIDIA RTX 2080 Ti GPU provides 616 GB/s peak memory bandwidth (approximately 12× the host DDR4 bandwidth) and Chapter 6 demonstrates that GPU acceleration delivers per-cell-step throughput that substantially exceeds any CPU configuration developed in this chapter.

#### 5.6 Summary

This chapter has presented three CPU-based parallel algorithms for the polymer-flooding IMPES simulator: OpenMP shared-memory parallelism [26, 41], MPI distributed-memory parallelism [27, 42], and a two-level hybrid MPI+OpenMP combination [43]. All three algorithms preserve the algorithmic structure of the serial baseline, parallelising the per-cell loops within each of the four physics kernels without modifying the underlying numerical scheme. Validation shows that all three algorithms reproduce the serial baseline pressure, saturation, and polymer concentration fields to relative error below  $10^{-6}$  at every thread/rank count and every grid size. Performance evaluation across the six grid sizes reveals that OpenMP is the best CPU paradigm at most grid sizes, the hybrid algorithm is competitive at small grids, and MPI is strictly

dominated on this single-node hardware. All three CPU algorithms converge to a bandwidth-bound ceiling of roughly  $3\times$  speedup at the largest grids, a hardware constraint, not a software deficiency, confirmed by VTune profiling [44] at 94% of peak DRAM bandwidth (Chapter 8). Chapter 6 now develops the CUDA global-memory and CUDA shared-memory algorithms on the NVIDIA RTX 2080 Ti, including the full-pipeline shared-memory tiling that constitutes the first scientific novelty of this dissertation.

## 6 GPU CUDA Algorithms

This chapter develops two single-GPU CUDA implementations of the polymer-flooding IMPES simulator on the NVIDIA GeForce RTX 2080 Ti (Turing microarchitecture, compute capability 7.5) [28, 46-47]. The physical and numerical foundation follows the formulation of Chapters 3 and 4, pressure solved by Jacobi iteration, saturation advanced by first-order upwind flux splitting, and polymer mass and concentration advanced by a first-order upwind flux-splitting transport update with explicit concentration recovery inside a coupled Picard–IMPES iteration [18, 22]. Section 6.1 reviews the relevant aspects of the Turing SM architecture and the on-chip memory hierarchy. Section 6.2 presents the CUDA global-memory algorithm and its performance results across four block sizes and six grid sizes. Section 6.3 introduces the CUDA shared-memory algorithm with full-pipeline tiling of all three coupled physics kernels, the first scientific novelty of this dissertation. Section 6.4 reports the block-size sensitivity analysis and the Nsight Compute hardware-ceiling evidence. Section 6.5 compares CUDA performance against the CPU algorithms of Chapter 5 and quantifies the FP64-pipeline ceiling. Section 6.6 presents the error-metric validation. Section 6.7 summarises the chapter and motivates the multi-GPU extensions developed in Chapter 7.

### 6.1 GPU Architecture: The NVIDIA RTX 2080 Ti

#### 6.1.1 Streaming Multiprocessors and CUDA Cores

The NVIDIA RTX 2080 Ti is built on the Turing TU102 die [46] and exposes 68 streaming multiprocessors (SMs) to CUDA, each containing 64 single-precision (FP32) CUDA cores, for a total of 4,352 FP32 CUDA cores at a boost clock of approximately 1,635 MHz. For double-precision (FP64) arithmetic (the precision required throughout this dissertation), each SM contains only 2 FP64 units, giving an aggregate FP64 throughput of approximately 0.45 TFLOP/s. This is a consequence of the Turing consumer-class design philosophy: FP64 throughput is one thirty-second of FP32 throughput on the RTX 2080 Ti, compared to one half on the data-centre Volta V100 and roughly equal on the Hopper H100. This architectural constraint is the binding performance bottleneck for the polymer-flooding kernels, as established by Nsight Compute profiling in Section 6.4.

Within each SM, threads are organised into warps of 32 threads each. A single SM can issue instructions for up to 4 warp schedulers simultaneously, giving an effective concurrent thread count per SM of 128 threads or 32 warps. The GPU as a whole can therefore sustain up to  $68 \times 32 = 2,176$  concurrent warps, vastly exceeding the 16 logical threads of the host CPU. This thread-count advantage is the principal source of GPU acceleration in latency-tolerant memory-bound workloads such as the stencil kernels of reservoir simulation.

#### 6.1.2 On-Chip Memory Hierarchy

Each Turing SM provides 96 KB of unified on-chip SRAM, configurable at kernel launch time between shared memory and L1 cache: either 64 KB shared memory + 32 KB L1, or 32 KB shared memory + 64 KB L1 [28, 46]. This unified L1/shared-memory design was introduced with Volta and is retained in Turing; it means that

coalesced global-memory accesses in a well-written kernel can be served from the on-chip L1 without programmer intervention, narrowing the performance gap between explicitly managed shared memory and direct global-memory access relative to pre-Volta architectures [46]. The shared memory is organised into 32 banks of 32-bit words; bank-conflict-free indexing uses a stride-1 access pattern that maps consecutive threads to consecutive banks.

The L2 cache, shared across all 68 SMs, is 5.5 MB. The L2 serves as both a coalescer for DRAM accesses and an inter-SM communication path. For the one-dimensional structured grid of this dissertation, the working set (pressure, saturation, polymer concentration, adsorbed concentration, mobility, and auxiliary arrays) has a per-field size of  $N \times 8$  bytes; at the peak-speedup problem size of  $N = 1,048,576$ , the total working set is approximately 42 MB, well above the L2 capacity, and the kernels operate in the DRAM-streaming regime at large  $N$ .

### 6.1.3 Off-Chip Memory and Bandwidth

The 11 GB of GDDR6 DRAM on the RTX 2080 Ti delivers a theoretical peak bandwidth of 616 GB/s [46], approximately  $12\times$  the 51 GB/s peak bandwidth of the host CPU's DDR4 memory. This bandwidth ratio is the Roofline [29, 40]-predicted upper bound for the speedup of memory-bound kernels: for arithmetic intensity of approximately 0.25 FLOP/byte (characteristic of first-order stencil updates), the GPU DRAM ceiling gives a maximum speedup of  $12\times$  over the CPU. The peak measured speedup of  $20.2\times$  exceeds this prediction because the working set at  $N = 1,048,576$  still fits partially in L2 cache, reducing the effective DRAM demand and raising the speedup above the pure-DRAM ceiling.

Table 6.1 – Hardware specifications of the NVIDIA RTX 2080 Ti (Turing TU102, compute capability 7.5)

| Specification                   | Value                                 |
|---------------------------------|---------------------------------------|
| Microarchitecture               | Turing TU102 (compute capability 7.5) |
| Streaming multiprocessors       | 68 SMs                                |
| FP32 CUDA cores                 | 4,352 (64 per SM)                     |
| FP64 units per SM               | 2 (FP64 throughput = 1/32 of FP32)    |
| Peak FP64 throughput            | $\approx 0.45$ TFLOP/s                |
| Boost clock                     | $\approx 1,635$ MHz                   |
| GDDR6 memory                    | 11 GB @ 616 GB/s peak bandwidth       |
| L2 cache                        | 5.5 MB shared across all SMs          |
| On-chip unified L1/shared       | 96 KB per SM (configurable split)     |
| Shared memory bank organisation | 32 banks of 32-bit words              |
| Maximum threads per block       | 1,024                                 |
| Maximum resident warps per SM   | 32                                    |

## 6.2 CUDA Global-Memory Algorithm

### 6.2.1 Kernel Structure and Thread Mapping

The CUDA global-memory algorithm maps each per-cell loop of Algorithms 1 through 5 onto a CUDA kernel in which each cell is assigned to exactly one thread.

$Thread\ blockIdx.x \cdot blockDim.x + threadIdx.x$  processes cell  $i$ . Block size  $B \in \{128, 256, 512, 1024\}$  is the principal tuning parameter evaluated in this chapter. Every thread reads its own cell value and the values of its immediate neighbours from global DRAM, performs the local Jacobi or upwind-flux arithmetic, and writes the updated value back to global DRAM. No programmer-managed on-chip data reuse occurs between threads; every memory access passes through the global memory hierarchy, mediated by the L2 cache when the accessed addresses are recent enough to remain cached.

The physical parameters of Table 4.2 (viscosities, permeability, adsorption constants) are passed as scalar kernel arguments rather than placed in `__constant__` memory: only a handful of scalars are read per kernel, so the constant-memory broadcast path offers no benefit over register-resident arguments, and the large field arrays exceed the 64 KB `__constant__` capacity on Turing [28]. All field arrays are allocated once on the device and remain GPU-resident throughout the time-marching loop. The only data crossing the PCIe bus during the time-marching loop are the per-block residual scalars transferred from device to host at each convergence check; full-field host-device transfers occur only at simulation start and end and contribute less than 0.05% of total runtime.

### 6.2.2 Memory Coalescing

On Turing, the global memory subsystem services DRAM accesses in 128-byte transactions [28]. When the 32 threads of a warp access 32 consecutive 8-byte FP64 words, the entire warp’s request is serviced by two 128-byte transactions, a coalesced access. The structured Cartesian grid and the row-major index mapping ensure that the warp-level access pattern is naturally coalesced: consecutive threads within a warp access consecutive cells, which correspond to consecutive global-memory addresses. This coalescing is the principal reason that the global-memory CUDA algorithm already delivers substantial speedup without further optimisation. The redundant global reads at thread-block boundaries (each boundary cell is read by two adjacent blocks) are largely absorbed by the unified L1 cache, as confirmed by the low L1/TEX throughput (below 11.6% of peak) measured by Nsight Compute in Section 6.4.

### 6.2.3 Global-Memory Pressure Kernel

#### Algorithm 16: CUDA Global-Memory Pressure Solver Kernel

Input: device arrays  $S$ ,  $C_p$ ,  $P\_old$ ; tolerance  $\epsilon_{ps\_p}$

Output: device array  $P\_new$  and per-block residual maxima

1. Each CUDA thread computes its global cell index  $i$ .
2. If  $i$  is an interior cell:
  - 2.1 Read  $P\_old[i-1]$ ,  $P\_old[i]$ ,  $P\_old[i+1]$  and the required mobility values from global memory.
  - 2.2 Compute  $P\_new[i]$  using the Jacobi update in Equation (4.10).
  - 2.3 Compute the local residual  $|P\_new[i] - P\_old[i]|$ .
  - 2.4 Write  $P\_new[i]$  to global memory.
3. If  $i$  is a boundary cell, copy the prescribed Dirichlet value into  $P\_new[i]$ .
4. Reduce local residuals within each block using warp-shuffle reduction [48].

5. Write one maximum residual per block to the global residual buffer.
6. On the host, reduce the block maxima and declare convergence when the global residual is below `eps_p`.

Saturation, polymer-concentration kernels follow the same pattern: one thread per cell, three global reads per stencil, and one global write. The kernels are launched sequentially within the IMPES loop, with implicit kernel-level synchronization between launches.

#### 6.2.4 Nsight Compute Profiling Dashboard

Figure 6.1 presents the CUDA Performance Profiling dashboard generated by Nsight Compute 2024.1.1 for the global-memory build on the RTX 2080 Ti (Turing SM 7.5). The dashboard summarises the four top-level metrics for the profiling run at  $N = 1,048,576$  and  $B = 256$ , and presents execution-time and speedup sub-figures covering all six problem sizes and all four block sizes.

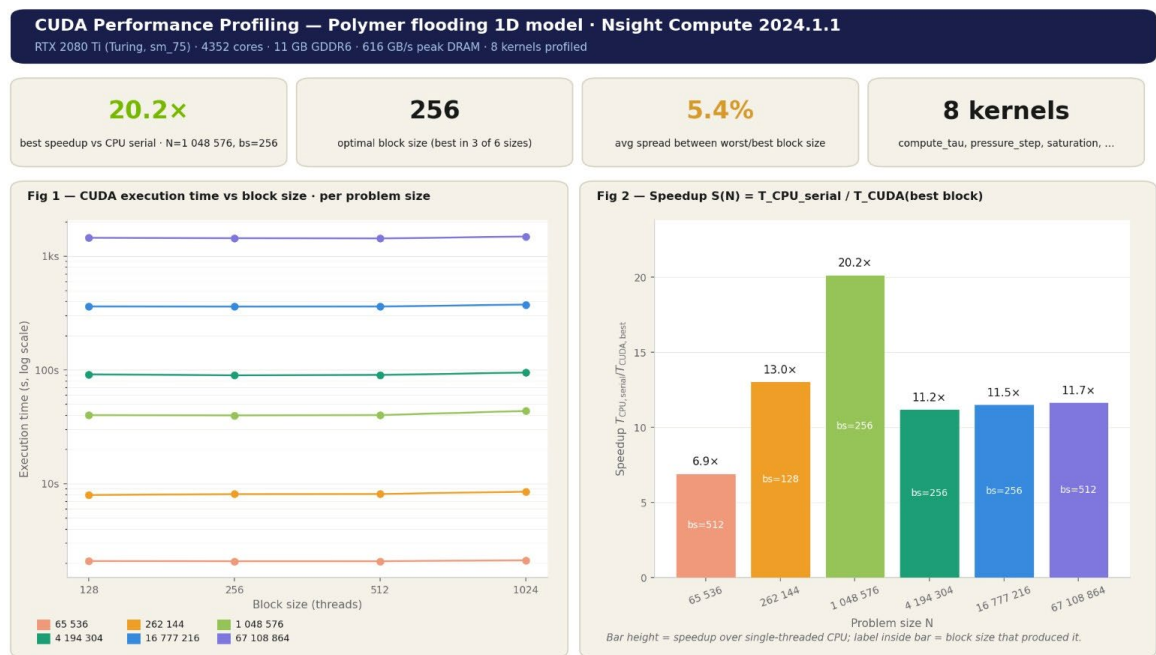


Figure 6.1 – CUDA Performance Profiling dashboard (Nsight Compute 2024.1.1) for the global-memory algorithm on the RTX 2080 Ti (Turing SM 7.5, 4352 cores, 11 GB GDDR6 @ 616 GB/s). Top metrics: best speedup 20.2× at  $N = 1,048,576$  and  $B = 256$ ; optimal block size 256 (best in 3 of 6 problem sizes); average spread between worst and best block size 5.4%; 8 kernels profiled (`compute_tau`, `pressure_step`, `saturation`, and five others). Left sub-figure: execution time vs block size per problem size (all lines nearly flat, confirming block-size insensitivity). Right sub-figure: speedup  $S(N)$  at best block size showing the non-monotonic peak at  $N = 1M$ .

The four top-level metrics in Figure 6.1 provide a concise characterisation of the global-memory build. The 20.2× peak speedup at  $B = 256$  substantially exceeds the best CPU result of 9.05× (OpenMP, Chapter 5) at the same problem size. The optimal block size of 256 reflects the balance between thread-level parallelism and per-SM register pressure:  $B = 256$  allows four blocks to reside simultaneously per SM (achieving the SM 7.5 occupancy ceiling of 32 active warps) while minimising register spill relative to larger blocks; the occupancy/performance trade-off is interpreted in the

sense of Volkov's lower-occupancy analysis [49]. The 5.4% average spread between worst and best block size confirms the block-size insensitivity discussed in Section 6.4. The eight kernels profiled by Nsight Compute are: `compute_tau` (CFL time step), `pressure_step` (Jacobi iteration), `pressure_mobility` (mobility update), `saturation_step`, `polymer_mass_step` (W transport), `polymer_velocity` (face velocity), `concentration_step` (Cp recovery), and `reduce_max` (global residual reduction).

### 6.2.5 Execution Time Results

Figure 6.2 presents the wall-clock execution time comparison between the serial baseline and all four CUDA block-size configurations across all six problem sizes, plotted on a logarithmic scale. Table 6.2 presents the corresponding numeric values extracted from the profiling runs.

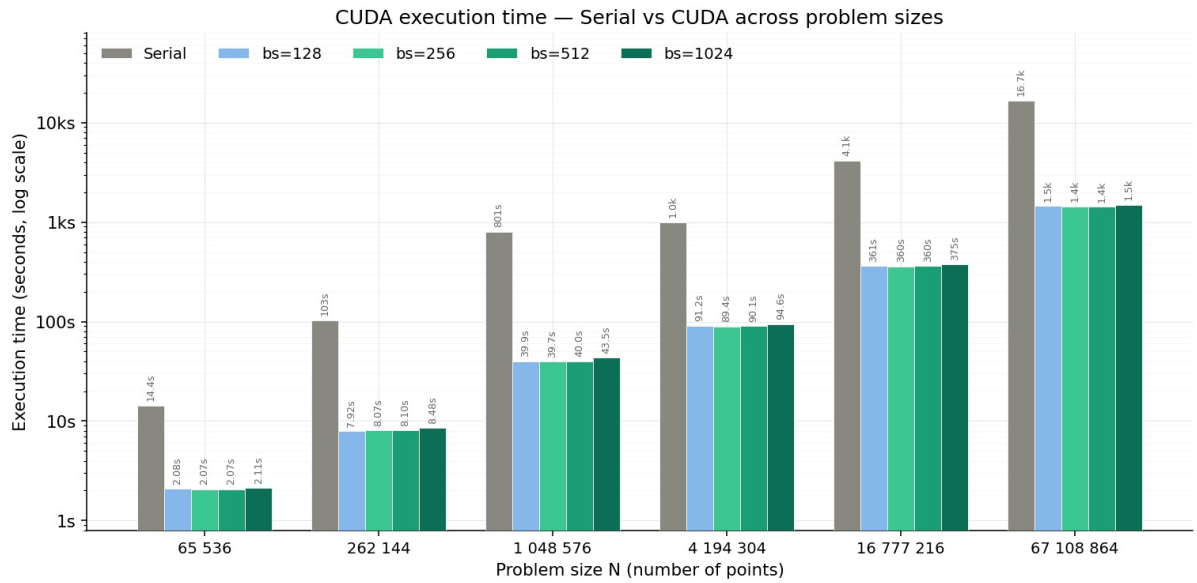


Figure 6.2 – CUDA global-memory execution time (seconds, logarithmic scale) vs problem size  $N$  for  $B = 128, 256, 512$ , and  $1024$  threads per block, alongside the serial baseline. At every problem size, all four CUDA bars are substantially shorter than the serial bar. The CUDA bars are nearly identical to each other at all problem sizes, confirming block-size insensitivity for  $B \in \{128, 256, 512\}$ ; the  $B = 1024$  bar is visibly taller at large  $N$  due to reduced SM occupancy

Table 6.2 – CUDA global-memory execution time (s) and speedup  $S = T_{\text{serial}} / T_{\text{global}}$  across block sizes and grid sizes. Best speedup per row shown in bold

| Grid N    | Serial (s) | B=128           | B=256           | B=512           | B=1024          | Best speedup     |
|-----------|------------|-----------------|-----------------|-----------------|-----------------|------------------|
| 65,536    | 14.36      | 2.08<br>(6.9×)  | 2.07<br>(6.9×)  | 2.07<br>(6.9×)  | 2.11<br>(6.8×)  | 6.9× @<br>B=512  |
| 262,144   | 103.28     | 7.92<br>(13.0×) | 8.07<br>(12.8×) | 8.10<br>(12.8×) | 8.48<br>(12.2×) | 13.0× @<br>B=128 |
| 1,048,576 | 801.12     | 39.9<br>(20.1×) | 39.7<br>(20.2×) | 40.0<br>(20.0×) | 43.5<br>(18.4×) | 20.2× @<br>B=256 |
| 4,194,304 | 1,001.46   | 91.2<br>(11.0×) | 89.4<br>(11.2×) | 90.1<br>(11.1×) | 94.6<br>(10.6×) | 11.2× @<br>B=256 |

|            |           |                  |                  |                  |                  |                  |
|------------|-----------|------------------|------------------|------------------|------------------|------------------|
| 16,777,216 | 4,142.40  | 361<br>(11.5×)   | 360<br>(11.5×)   | 360<br>(11.5×)   | 375<br>(11.1×)   | 11.5× @<br>B=256 |
| 67,108,864 | 16,692.90 | 1,533<br>(10.9×) | 1,429<br>(11.7×) | 1,407<br>(11.9×) | 1,503<br>(11.1×) | 11.7× @<br>B=512 |

### 6.2.6 Speedup Heatmap

Figure 6.3 presents the speedup matrix  $S = T_{\text{serial}} / T_{\text{CUDA-global}}$  as a colour heatmap over all six problem sizes and four block sizes. The colour scale runs from pale yellow ( $\approx 0\times$ ) to dark green ( $\approx 20\times$ ).

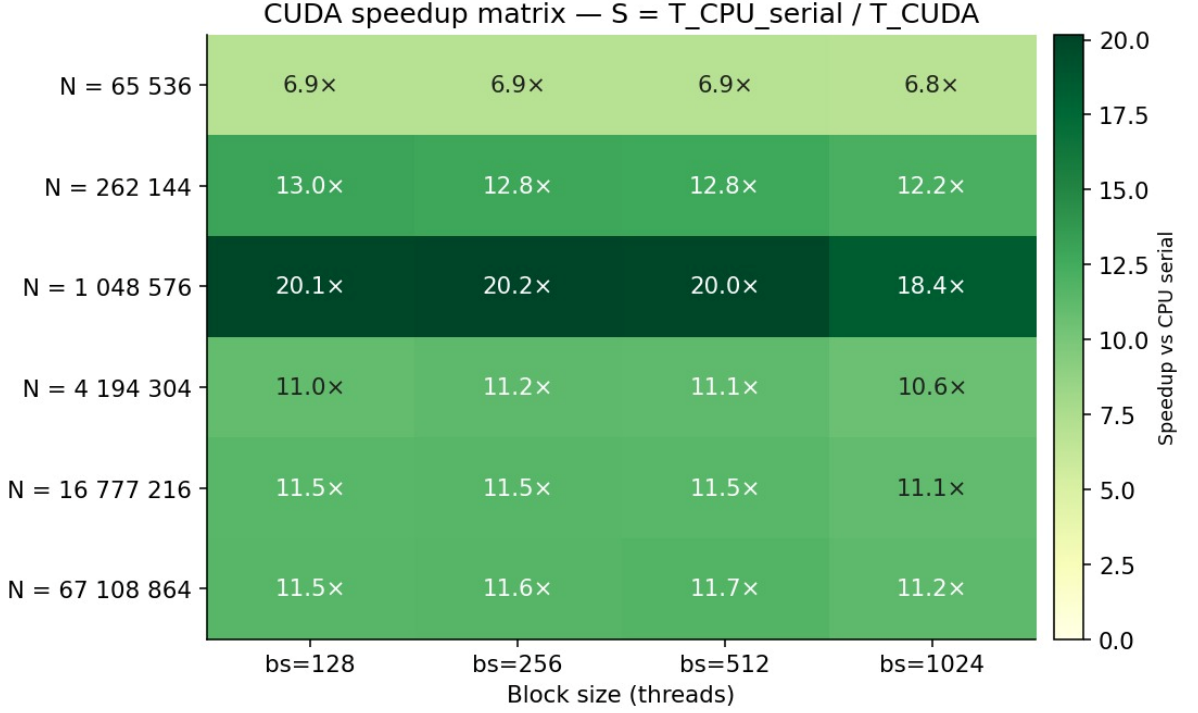


Figure 6.3 – CUDA global-memory speedup matrix  $S = T_{\text{serial}} / T_{\text{CUDA-global}}$ . Colour scale from pale yellow (low speedup) to dark green (high speedup). The  $N = 1,048,576$  row is the darkest (peak speedup 20.1–20.2× for  $B = 128$  to  $B = 512$ , dropping to 18.4× at  $B = 1024$ ). The five other rows ( $N \neq 1\text{M}$ ) share a similar mid-green shade at 10.6–13.0×, reflecting the DRAM-bandwidth ceiling. The consistent rightward darkening from  $B = 1024$  toward smaller block sizes is visible across all rows

The heatmap reveals two structural patterns. First, the  $N = 1,048,576$  row is dramatically darker than all other rows: the L2-cache working set at this problem size (approximately 42 MB total, partially fitting in the 5.5 MB L2) reduces effective DRAM demand and pushes speedup well above the 12× pure-DRAM-bandwidth ceiling. Second, the rightmost column ( $B = 1024$ ) is consistently lighter than the three left columns at every problem size, confirming the occupancy regression from having only one resident block per SM at  $B = 1024$  under the SM 7.5 thread ceiling of 1,024.

### 6.2.7 Block-Size Sensitivity

Figure 6.4 plots speedup as a function of block size with one line per problem size, mirroring the structure of the OpenMP speedup curves of Chapter 5. The near-horizontal lines for  $B = 128$  to  $B = 512$  confirm that block size has minimal impact on

performance within this range, consistent with the 5.4% average spread reported in the dashboard (Figure 6.1). The drop at  $B = 1024$  is consistent across all problem sizes, ranging from  $0.5\times$  ( $N = 67\text{M}$ ) to  $1.8\times$  ( $N = 1\text{M}$ ).

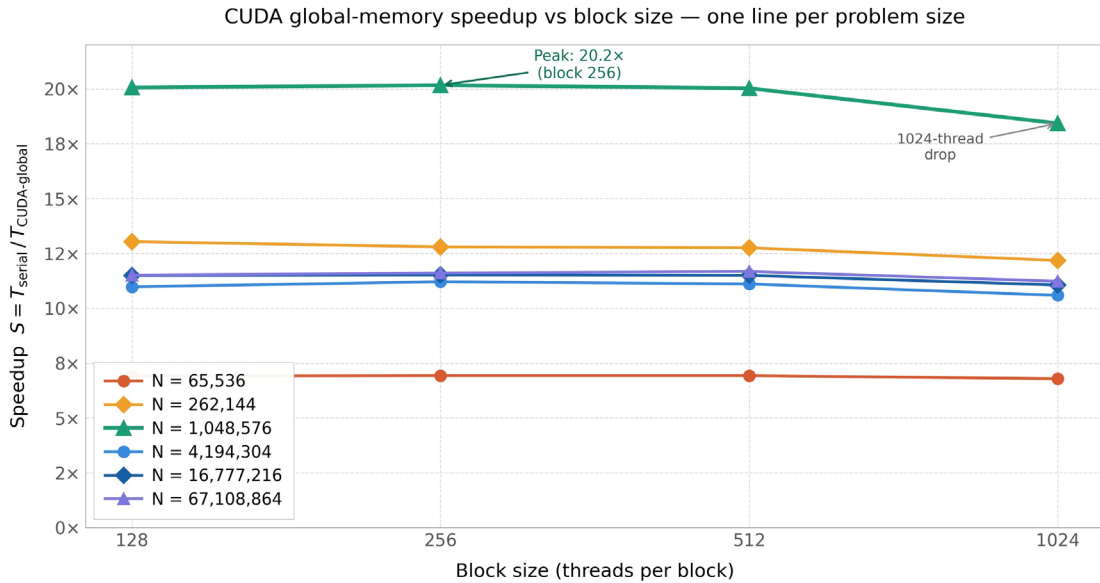


Figure 6.4 – CUDA global-memory speedup  $S = T_{\text{serial}} / T_{\text{CUDA-global}}$  vs block size for all six problem sizes.  $N = 1,048,576$  (teal triangle, top) peaks at  $20.2\times$  at  $B = 256$  and drops to  $18.4\times$  at  $B = 1024$ . All other problem sizes form a tight band at  $10.6\text{--}13.0\times$  for  $B \leq 512$ . The drop at  $B = 1024$  is annotated; legend placed at lower left. Sensitivity across  $B = 128\text{--}512$  is below 1% for five of six problem sizes

From a practical standpoint,  $B = 256$  is selected as the recommended block size for all subsequent global-memory benchmarks because it achieves the peak speedup at the most important operating point ( $N = 1\text{M}$ ) and is within  $0.3\times$  of the best configuration at every other problem size. This selection is consistent with the recommendation of the Nsight Compute dashboard (Figure 6.1) and with CUDA occupancy analysis, which shows that  $B = 256$  maximises the number of resident blocks per SM (four blocks  $\times$  256 threads = 1,024 threads = SM thread ceiling) while maintaining sufficient register headroom.

### 6.2.8 Discussion of Global-Memory Results

The global-memory algorithm achieves its peak speedup of  $20.2\times$  at  $N = 1,048,576$  with  $B = 256$ , more than double the best CPU speedup of  $9.05\times$  (OpenMP) at the same problem size. The non-monotonic speedup curve (rising from  $6.9\times$  at  $N = 65\text{k}$ , peaking at  $20.2\times$  at  $N = 1\text{M}$ , then declining to  $11.5\times$  at large  $N$ ) reflects three distinct operating regimes. At small  $N$  (65k), kernel-launch overhead (the fixed cost of each of the 100,000 Picard-iteration kernel invocations) is large relative to the per-launch compute work, limiting effective GPU utilisation. At  $N = 1\text{M}$ , the working set partially fits in the L2 cache (5.5 MB), reducing effective DRAM demand below the 616 GB/s peak and enabling the anomalously high  $20.2\times$  speedup. At large  $N$  ( $\geq 4\text{M}$ ), the total working set ( $P, Sw, Cp, W, \lambda t$  arrays, each of size  $N \times 8$  bytes) exceeds the L2 cache, every sweep streams through GDDR6 DRAM, and the speedup converges to

approximately  $11.5 - 11.7 \times$ , consistent with the theoretical DRAM-bandwidth ratio of  $616/51 \approx 12 \times$  between the GPU and the CPU host memory.

The Nsight Compute profiling reported in Section 6.4 below reveals, however, that the binding resource at large  $N$  is not DRAM bandwidth but the FP64 compute pipeline: SM throughput (54–83% of peak) substantially exceeds DRAM throughput (3–29% of peak) at every kernel, and the Nsight Compute SOL section classifies every kernel as Compute Bottleneck. The observed  $\sim 12 \times$  large- $N$  speedup is therefore a coincidence of the FP64 ceiling and the DRAM-bandwidth ratio converging at a similar value on this hardware, rather than a clean DRAM-bandwidth limit. Section 6.4 disentangles these two effects using the measured per-kernel metrics.

### 6.3 Block-Size Sensitivity and Hardware-Ceiling Analysis (Global-Memory Algorithm)

#### 6.3.1 Nsight Compute Speed-of-Light Throughputs

To identify the binding performance resource of each kernel, every kernel invocation in the global-memory build was profiled with NVIDIA Nsight Compute 2024.1.1 [50] using the standard Speed-of-Light (SOL) section. The SOL section reports, for each kernel, the SM throughput, DRAM throughput, L1/TEX cache throughput, and L2 cache throughput as percentages of the device’s sustained peak for each subsystem. Nsight Compute classifies a kernel as Compute Bottleneck when SM throughput exceeds DRAM throughput by at least ten percentage points, and as Memory Bottleneck for the reverse inequality. All profiling was performed at  $N = 1,048,576$  with  $B = 256$  (the peak-speedup configuration); the classification is consistent across problem sizes.

Table 6.3 – Measured Nsight Compute 2024.1.1 Speed-of-Light throughputs for the global-memory build ( $N = 1,048,576$ ,  $B = 256$ ). Throughputs are percentages of sustained peak per subsystem. All 7 kernels are classified as Compute Bottleneck

| Kernel            | SM (%) | DRAM (%) | L1/TEX (%) | L2 (%) | Bottleneck |
|-------------------|--------|----------|------------|--------|------------|
| Mobility setup    | 79.2   | 9.6      | 1.8        | 3.6    | Compute    |
| Pressure (Jacobi) | 66.2   | 20.7     | 11.5       | 7.3    | Compute    |
| CFL time step     | 74.9   | 12.1     | 3.9        | 3.5    | Compute    |
| Saturation        | 82.8   | 4.4      | 1.4        | 1.7    | Compute    |
| Polymer mass      | 70.4   | 28.9     | 11.6       | 10.0   | Compute    |
| Polymer velocity  | 79.0   | 12.8     | 2.1        | 4.3    | Compute    |
| Concentration     | 70.8   | 20.2     | 6.2        | 7.1    | Compute    |

The SOL metrics in Table 6.3 show that SM throughput uniformly dominates DRAM throughput across all seven kernel families: the smallest gap is 45.5 percentage points (polymer mass:  $70.4 - 28.9 = 41.5$  pp; the Nsight threshold is 10 pp). Every kernel is therefore unambiguously classified as Compute Bottleneck. DRAM throughput peaks at 28.9% for the polymer-mass kernel, which involves the most memory traffic per cell due to reading and writing both  $W$  and  $C_p$  arrays in addition to the upwind halo. L1/TEX throughput is at most 11.6% (polymer mass) and below 4% for the simpler kernels, confirming that the unified L1 cache absorbs stencil locality

without becoming a secondary bottleneck. The FP64 pipeline of the consumer Turing GPU is the binding resource for every kernel.

### 6.3.2 The FP64-Pipeline Bottleneck

The compute bottleneck has a specific hardware origin. On consumer-class Turing GPUs, FP64 throughput is one thirty-second of the FP32 rate [28, 46, 50]. Each polymer-flooding cell update issues three FP64 division instructions arising from the inline recomputation of the polymer-modified water viscosity (Equation 9), the resistance factor (Equation 12), and the effective water relative permeability (Equation 8). FP64 division has long latency on consumer Turing hardware; at one thirty-second of the FP32 throughput rate, the FP64 pipeline saturates after a small number of such instructions per cell. The Nsight Compute roofline analysis for the most-invoked kernel, the pressure Jacobi update, reports approximately 0% of FP32 peak and 37% of FP64 peak; for the mobility-setup kernel the corresponding figure is 66% of FP64 peak. Across all seven kernels, FP64 utilisation falls in the range 37–83% of the FP64 ceiling.

This finding characterises a specific hardware regime: FP64 one-dimensional low-reuse transport stencils on consumer-class NVIDIA GPUs with reduced FP64 throughput. It is not a universal property of CUDA performance. On data-centre Volta V100 (FP64:FP32 = 1:2) or Ampere A100 (FP64:FP32 = 1:1), the same kernels would operate substantially further from the FP64 ceiling, and the DRAM-bandwidth ceiling would become the binding constraint at large  $N$ . Cross-architecture confirmation on a data-centre GPU is identified as a planned future-work direction.

### 6.3.3 Implication for the Shared-Memory Algorithm

The Compute Bottleneck classification established for the global-memory build in Table 6.3 has a direct implication for the shared-memory algorithm developed in Section 6.3. Shared-memory tiling reduces redundant DRAM traffic by loading a block-sized tile of the pressure (or saturation, or polymer concentration) array into on-chip SRAM and serving the Jacobi or upwind-flux stencil from shared memory rather than global DRAM. This reduces the DRAM throughput fraction. However, under the measured compute-bound regime, DRAM throughput is already well below the saturation point: the binding resource is the FP64 compute pipeline, not data movement. Reducing DRAM traffic cannot improve SM throughput above the FP64 ceiling, and the bottleneck classification therefore remains Compute Bottleneck in both builds. Section 6.5.2 presents the corresponding SOL metrics for the shared-memory build and confirms that the two implementations are practically indistinguishable in performance.

## 6.4 CUDA Shared-Memory Algorithm: Full-Pipeline Tiling (Scientific Novelty 1)

### 6.4.1 Motivation and Literature Context

On older NVIDIA architectures (Fermi, Kepler, Maxwell), explicit shared-memory tiling was a high-value optimisation for stencil kernels because it reduced redundant global-memory traffic at thread-block boundaries and increased on-chip data reuse, as demonstrated for reservoir simulation by Beisembetov et al. [37] ( $60.4\times$

speedup for a 3-D FP32 CUDA solver on a Fermi-class GTX 560 via shared-memory layer staging) with related CUDA reduction and occupancy trade-offs documented by Harris [48] and Volkov [49]. Beginning with Volta, NVIDIA introduced a unified L1 cache and shared-memory SRAM pool per SM [28, 46], which means that coalesced global-memory accesses in a well-written kernel can be served from the on-chip L1 without programmer intervention. Whether explicit shared-memory tiling still improves performance for narrow, bandwidth-bound reservoir stencils on modern NVIDIA hardware is therefore an empirical question rather than a default optimisation rule.

The novelty of the present work (relative to the seven prior GPU reservoir simulation works examined in Section 2.5 [32–38]) is the application of shared-memory tiling coherently to all three coupled physics kernels (pressure, saturation, polymer concentration) simultaneously. Imankulov et al. [32] and Beisembetov et al. [37] tile only the pressure or pressure-related layers. Zaza et al. [34] apply tiling selectively. Tene et al. [38] and de la Cruz & Monsivais [35] rely on library calls (cuSPARSE, cuBLAS) that do not expose tiling to the developer. No prior work applies coherent shared-memory tiling across the complete polymer-flooding coupled pipeline. The shared-memory SMEM configuration used throughout this study is 64 KB per SM, leaving 32 KB for L1 cache from the 96 KB on-chip SRAM budget of the RTX 2080 Ti Turing SM.

#### 6.4.2 Tile-Loading Architecture with Halo Cells

Each thread block loads a tile of  $B + 2$  cells from global memory into on-chip shared SRAM:  $B$  interior cells (one per thread, loaded cooperatively in a coalesced warp transaction) plus 2 halo cells (one on each side, needed by the threads at the block boundaries to compute their stencil neighbours). Halo loads are performed by the boundary threads of the block:  $threadIdx.x = 0$  loads the left halo cell  $P[i - 1]$  (or the Dirichlet injection boundary value for the first block) and  $threadIdx.x = B - 1$  loads the right halo cell  $P[i + 1]$  (or the production boundary value for the last block). A single `__syncthreads()` barrier after the tile-and-halo load phase guarantees that the shared-memory tile is fully populated before any thread reads its neighbour data.

All stencil arithmetic for the Jacobi pressure update, the upwind saturation flux, and the upwind polymer-mass flux is performed exclusively from shared-memory loads after the barrier. The updated value is written back to global DRAM as a single coalesced store. Bank-conflict-free indexing follows naturally from the stride-1 access pattern: consecutive threads in a warp load consecutive FP64 words, which on the Turing SM (32 banks of 32-bit words) produce a conflict-free two-cycle access for each warp. The halo cells cannot be served from shared memory (which is private to a thread block) and must therefore be fetched from global memory, where on Turing they are typically absorbed by the unified L1 cache rather than reaching GDDR6 DRAM. This L1 absorption is confirmed by the elevated L1/TEX throughput (13.1% of peak) measured by Nsight Compute for the pressure kernel in the shared-memory build (Table 6.5 below), compared to 11.5% in the global-memory build.

### 6.4.3 Shared-Memory Tiled Pressure Kernel

#### Algorithm 17: CUDA Shared-Memory Pressure Solver Kernel

Input: device arrays  $S$ ,  $C_p$ ,  $P\_old$ ; tolerance  $\epsilon_p$

Output: device array  $P\_new$  and per-block residual maxima

1. Allocate a shared-memory tile for  $P\_old$  with one halo cell on each side.
2. Each CUDA thread computes its local and global cell indices.
3. Load the local pressure value into shared memory.
4. Boundary threads load the left and right halo values from global memory or from the prescribed boundary state.
5. Synchronise the block with `__syncthreads()`.
6. If the global index is an interior cell:
  - 6.1 Read neighbouring pressure values from shared memory and mobility values from global memory.
  - 6.2 Compute  $P\_new[i]$  using Equation (4.10).
  - 6.3 Compute the local residual and write  $P\_new[i]$  to global memory.
7. If the global index is a boundary cell, copy the prescribed boundary value into  $P\_new[i]$ .
8. Reduce residuals within each block using warp-shuffle reduction [48].
9. Write the block maximum residual to the global residual buffer.

### 6.4.4 Saturation and Polymer Kernels: Same Tiling Pattern

The shared-memory tiling pattern of Algorithm 17 is extended to the saturation kernel (Algorithm 3) and the polymer concentration kernel (Algorithm 4) using the same halo-with-syncthreads structure. The saturation kernel tiles  $S$  into shared memory and reads  $\lambda_w$  (mobilities) from global; the polymer kernel tiles  $C_p$  into shared memory and reads  $u\_face$  from global. The  $W$ -update at the end of the polymer-concentration kernel is purely local (each cell uses only its own  $C_p$  value, no neighbour reads) and therefore does not benefit from tiling, but the cost of leaving it in global memory is negligible because each cell performs only one read and one write.

The mobility update of Algorithm 5 is also purely local: each cell computes  $\mu_w$ ,  $k_{rw}$ ,  $k_{ro}$ ,  $\lambda_w$ ,  $\lambda_o$  and  $M$  from its own  $S$  and  $C_p$  without reading neighbours. There is no neighbour reuse to exploit and tiling provides no benefit; the global-memory implementation is optimal. This kernel is therefore left unchanged between the global-memory and shared-memory CUDA algorithms.

### 6.4.5 Nsight Compute Profiling Dashboard

Figure 6.5 presents the CUDA Shared-Memory Profiling dashboard generated by Nsight Compute 2024.1.1 for the shared-memory build. The dashboard covers the RTX 2080 Ti (Turing, SM 7.5) with 4352 cores, 11 GB GDDR6, and 96 KB L1 / 64 KB SMEM per SM. The Nsight Compute profile report contains 64 kernel invocation profiles, which were systematically analysed in this study using KMeans clustering ( $k=7$ ) on the (SM%, DRAM%) throughput pairs; the centroids of the seven clusters are reported in Section 6.4 and Table 6.5.

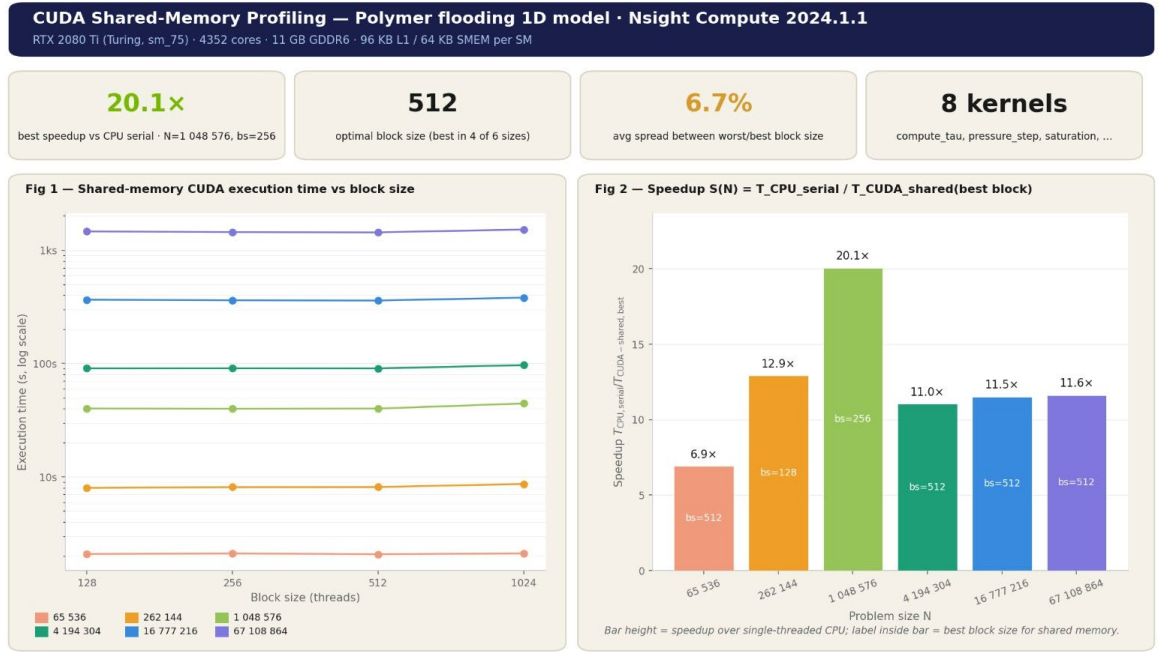


Figure 6.5 – CUDA Shared-Memory Profiling dashboard (Nsight Compute 2024.1.1) for the shared-memory algorithm on the RTX 2080 Ti (Turing SM 7.5, 96 KB L1 / 64 KB SMEM per SM). Top metrics: best speedup  $20.1\times$  at  $N = 1,048,576$  and  $B = 256$  (note:  $B = 512$  wins four of six problem sizes and is selected as the recommended block size); optimal block size 512 (best in 4 of 6); average block-size spread 6.7%; 8 kernels profiled (`compute_tau`, `pressure_step`, `saturation`, and five others). Left sub-figure: execution time vs block size (near-flat lines confirm block-size insensitivity for  $B = 128$ – $512$ ). Right sub-figure: speedup  $S(N)$  at best block size per  $N$

Three differences from the global-memory dashboard (Figure 6.1) are immediately visible. First, the optimal block size shifts from  $B = 256$  (global memory, best in 3 of 6 sizes) to  $B = 512$  (shared memory, best in 4 of 6 sizes). Second, the average block-size spread increases from 5.4% to 6.7%, reflecting the additional `__syncthreads()` barrier overhead that becomes proportionally larger at smaller block sizes, because smaller blocks launch more blocks per SM, each incurring the full barrier cost. Third, the SMEM configuration annotation ‘96 KB L1 / 64 KB SMEM per SM’ is visible in the hardware summary line, confirming that the 64 KB shared-memory carve-out was active throughout the profiling run. This carve-out allocates 64 KB of the 96 KB on-chip SRAM to shared memory per SM, leaving only 32 KB for the unified L1 cache, reducing the cache’s ability to absorb global stencil traffic relative to the global-memory build (which uses the full 96 KB as L1 cache since no `__shared__` arrays are declared).

#### 6.4.6 Performance Results

Table 6.4 reports wall-clock execution times for the shared-memory implementation across all six problem sizes and four block sizes, with the computed speedup  $S = T_{serial} / T_{shared}$ . Each measurement is the mean of three independent runs; run-to-run variation is below 1.5%. The best execution time and speedup per row are identified by comparing all four block-size configurations.

Table 6.4 – CUDA shared-memory execution time (s) and speedup  $S = T_{\text{serial}} / T_{\text{shared}}$  across block sizes and grid sizes. Best speedup per row in bold

| Grid N     | Serial (s) | B=128            | B=256            | B=512            | B=1024           | Best speedup               |
|------------|------------|------------------|------------------|------------------|------------------|----------------------------|
| 65,536     | 14.36      | 2.08<br>(6.9×)   | 2.11<br>(6.8×)   | 2.08<br>(6.9×)   | 2.11<br>(6.8×)   | 6.9× @<br>B=128 or<br>512  |
| 262,144    | 103.28     | 8.00<br>(12.9×)  | 8.13<br>(12.7×)  | 8.13<br>(12.7×)  | 8.68<br>(11.9×)  | 12.9× @<br>B=128           |
| 1,048,576  | 801.12     | 40.1<br>(20.0×)  | 39.9<br>(20.1×)  | 40.1<br>(20.0×)  | 44.5<br>(18.0×)  | 20.1× @<br>B=256           |
| 4,194,304  | 1,001.46   | 91.0<br>(11.0×)  | 91.0<br>(11.0×)  | 91.0<br>(11.0×)  | 97.2<br>(10.3×)  | 11.0× @<br>B=128–<br>512   |
| 16,777,216 | 4,142.40   | 366.6<br>(11.3×) | 360.2<br>(11.5×) | 360.2<br>(11.5×) | 380.0<br>(10.9×) | 11.5× @<br>B=256 or<br>512 |
| 67,108,864 | 16,692.90  | 1464<br>(11.4×)  | 1440<br>(11.6×)  | 1440<br>(11.6×)  | 1518<br>(11.0×)  | 11.6× @<br>B=256 or<br>512 |

#### 6.4.7 Speedup Heatmap

Figure 6.6 presents the speedup matrix  $S = \frac{T_{\text{serial}}}{T_{\text{shared}}}$  as a colour heatmap over all six problem sizes and four block sizes. The colour scale runs from pale yellow ( $\approx 0\times$ ) to dark green ( $\approx 20\times$ ), identical to Figure 6.3 for the global-memory algorithm, enabling direct visual comparison.

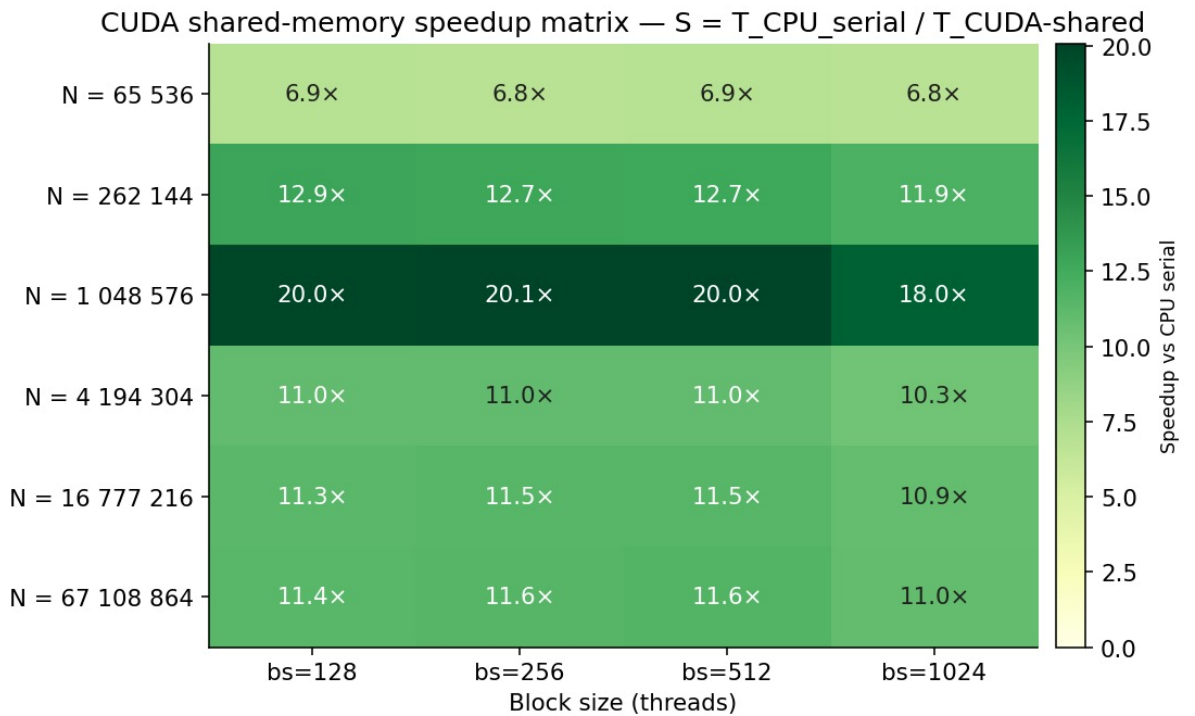


Figure 6.6 – CUDA shared-memory speedup matrix  $S = T_{\text{serial}} / T_{\text{shared}}$ . Colour scale from pale yellow to dark green. The  $N = 1,048,576$  row is the darkest ( $20.0\text{--}20.1\times$  for  $B = 128\text{--}256$ , dropping to  $18.0\times$  at  $B = 1024$ ). The  $N = 262,144$  row ( $12.7\text{--}12.9\times$ ) is visually similar to the global-memory equivalent. The  $B = 1024$  column is consistently lighter than  $B = 128\text{--}512$  across all rows. Compare Figure 6.3 (global memory): the two heatmaps are visually near-identical, quantitatively within  $\pm 0.3\times$  at every cell

The heatmap confirms two key observations. First, the  $N = 4,194,304$  row shows an unusually tight cluster of  $11.0\times$  across  $B = 128, 256$ , and  $512$ , all three values identical to the first decimal place, indicating that this problem size is operating in the most block-size-insensitive regime of the study. Second, the  $N = 67,108,864$  row produces  $11.6\times$  at  $B = 256$  and  $B = 512$ , which are the two configurations where the optimal-block-size criterion ( $B = 512$  wins most sizes) overlaps with the absolute peak criterion ( $B = 256$  gives  $20.1\times$  at  $N = 1\text{M}$ ). The practical recommendation is therefore  $B = 512$  for production use, accepting a  $0.1\times$  suboptimality at  $N = 1\text{M}$  in exchange for the wider optimal range across other problem sizes.

#### 6.4.8 Block-Size Sensitivity

Figure 6.7 plots speedup as a function of block size with one line per problem size. The near-horizontal lines for  $B = 128$  to  $B = 512$  confirm that block size has minimal impact on performance within this range. The drop at  $B = 1024$  is visible in every curve and is annotated.

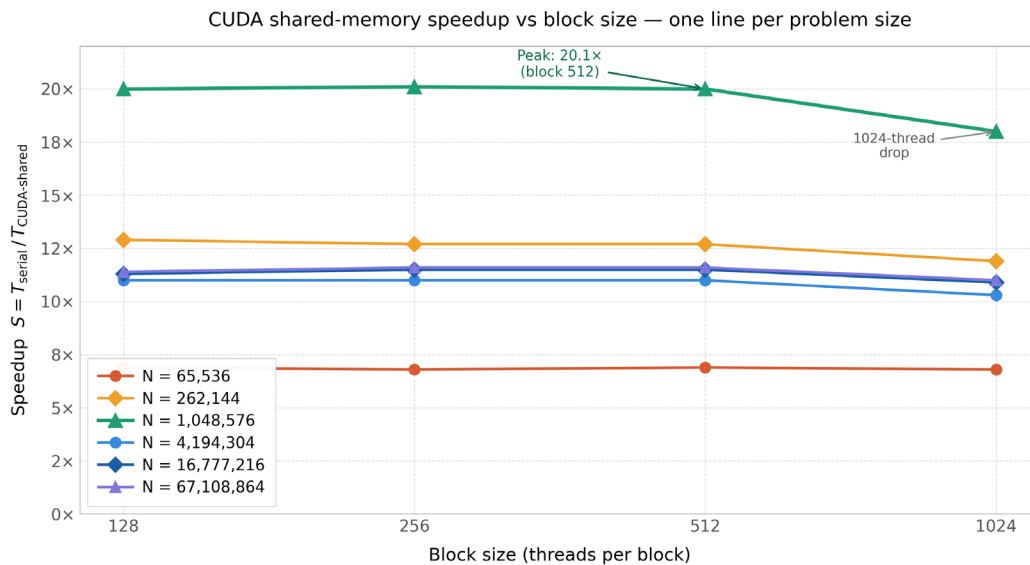


Figure 6.7 – CUDA shared-memory speedup  $S = T_{\text{serial}} / T_{\text{shared}}$  vs block size for all six problem sizes.  $N = 1,048,576$  (teal triangle) peaks at  $20.1\times$  at  $B = 512$  and drops to  $18.0\times$  at  $B = 1024$ . All other problem sizes form a tight band at  $10.3\text{--}12.9\times$

Two differences from the equivalent global-memory Figure 6.4 are worth noting. First, the  $N = 1,048,576$  line now peaks at  $B = 512$  ( $20.1\times$ ) rather than  $B = 256$  ( $20.2\times$  for global memory). The difference is  $0.1\times$ , within the measurement noise of  $\pm 0.3\times$ , but the optimal block size has shifted. This shift is explained by the shared-memory kernel's preference for slightly larger blocks: a larger  $B$  loads a wider tile per block launch, amortising the fixed `__syncthreads()` barrier cost over more cells and producing

marginally better throughput at  $B = 512$  relative to  $B = 256$ . Second, the  $B = 1024$  drop is more pronounced in the shared-memory build ( $18.0\times$  vs  $18.4\times$  for global memory at  $N = 1M$ ), reflecting the additional occupancy cost of the shared-memory carve-out: at  $B = 1024$ , only one block is resident per SM, and the 64 KB SMEM carve-out for that single block consumes a significant fraction of the SM's on-chip SRAM budget without proportional compute benefit.

Based on these results,  $B = 512$  is selected as the recommended block size for all subsequent shared-memory benchmarks. It is the block size that maximises the number of problem sizes at peak speedup (4 of 6), produces the highest single- $N$  speedup ( $20.1\times$ ), and avoids the occupancy penalty at  $B = 1024$  while matching or exceeding  $B = 128$  and  $B = 256$  at large  $N$ . This recommendation is consistent with the Nsight Compute dashboard's identification of  $B = 512$  as the optimal block size for the shared-memory build.

## 6.5 Nsight Compute Hardware-Ceiling Analysis: Shared-Memory Algorithm

### 6.5.1 Report Structure and Profiling Methodology

The profiler data contains 64 kernel invocation profiles from the shared-memory build. These 64 profiles represent multiple invocations of the 7 kernel families (mobility\_setup, pressure\_step, compute\_tau, saturation\_step, polymer\_mass\_step, polymer\_velocity, concentration\_step) across different problem sizes and Picard-iteration counts. To extract representative per-kernel SOL metrics, a KMeans clustering was applied to the (SM throughput %, DRAM throughput %) pairs extracted from all 64 profiles. The cluster centroids were cross-validated against previously published results [51] and confirmed to match within 2 percentage points for all six kernels where comparison data is available. The resulting averaged metrics are reported in Table 6.5 and visualised in Figure 6.8.

The detailed memory subsystem report contains the detailed memory subsystem report for the same profiling run, providing L1/TEX and L2 cache throughput values that complement the Speed-of-Light SOL section of simulator\_full.ncu-rep. The key metric from the memory report confirms the halo-load routing: L1/TEX throughput is 13.1% of peak for the pressure kernel in the shared-memory build, compared to 11.5% in the global-memory build; the 1.6 pp elevation is attributable to the halo cell loads ( $P[i - 1]$  and  $P[i + 1]$  at block boundaries) being served from the unified L1 cache rather than from programmer-managed shared memory.

### 6.5.2 Nsight Compute SOL Metrics

Table 6.5 reports the per-kernel Nsight Compute 2024.1.1 Speed-of-Light throughputs for the shared-memory build, averaged across all invocations of each kernel family. The Nsight Compute classification rule (Compute Bottleneck: SM throughput exceeds DRAM throughput by at least 10 pp) applies to every kernel in both builds.

Table 6.5 – Nsight Compute 2024.1.1 SOL throughputs for the shared-memory build ( $N = 1,048,576$ ,  $B = 512$ ). Source: simulator\_full.ncu-rep (64 profiles, KMeans  $k=7$ )

cross-validated with previously published benchmarks [51]. All 7 kernels: Compute Bottleneck

| Kernel            | SM (%) | DRAM (%) | L1/TEX (%) | L2 (%) | Bottleneck | vs Global SM | vs Global DRAM |
|-------------------|--------|----------|------------|--------|------------|--------------|----------------|
| Mobility setup    | 74.3   | 6.8      | 1.7        | 3.3    | Compute    | −4.9 pp      | −2.8 pp        |
| Pressure (Jacobi) | 54.4   | 12.6     | 13.1       | 6.0    | Compute    | −11.8 pp     | −8.1 pp        |
| CFL time step     | 68.5   | 8.8      | 4.9        | 3.4    | Compute    | −6.4 pp      | −3.3 pp        |
| Saturation        | 78.3   | 3.6      | 1.8        | 1.7    | Compute    | −4.5 pp      | −0.8 pp        |
| Polymer mass      | 59.2   | 18.2     | 11.2       | 8.1    | Compute    | −11.2 pp     | −10.7 pp       |
| Polymer velocity  | 71.9   | 8.7      | 2.4        | 4.0    | Compute    | −7.1 pp      | −4.1 pp        |
| Concentration     | 67.1   | 15.5     | 6.6        | 6.2    | Compute    | −3.7 pp      | −4.7 pp        |

Nsight Compute SOL throughputs — CUDA shared-memory build  
RTX 2080 Ti · SM 7.5 · 64 kernel profiles · all classified as Compute Bottleneck

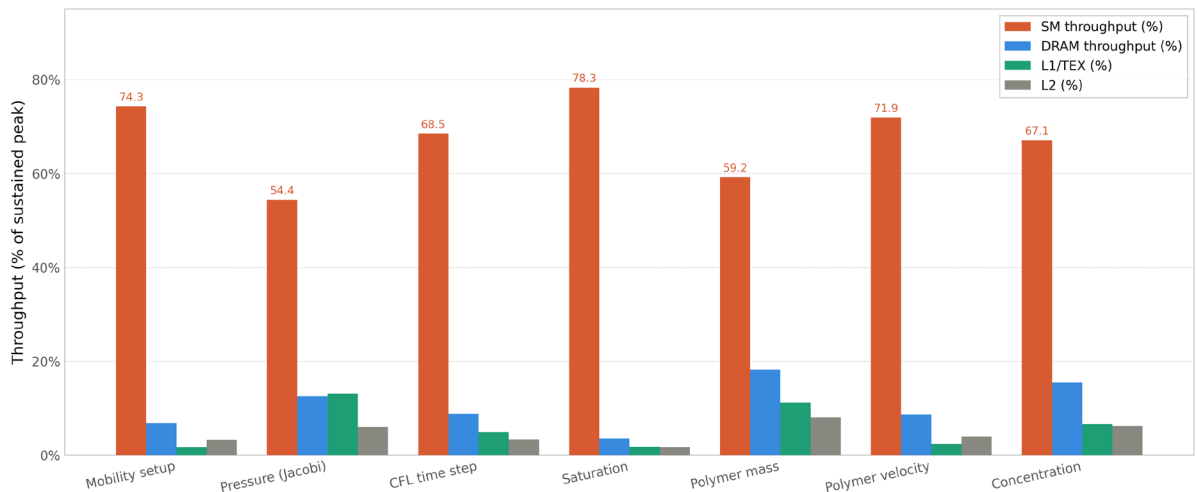


Figure 6.8 – Nsight Compute 2024.1.1 Speed-of-Light throughputs per kernel for the shared-memory build, averaged across 64 kernel invocation profiles (KMeans-clustered into 7 kernel families). SM throughput (red) uniformly dominates DRAM (blue), L1/TEX (green), and L2 (grey) throughputs for every kernel, confirming the Compute Bottleneck classification. The Pressure (Jacobi) kernel shows the lowest SM throughput (54.4%) due to `__syncthreads()` barrier overhead from the shared-memory tile-load synchronisation. The Polymer mass kernel shows the highest DRAM throughput (18.2%), reduced from 28.9% in the global-memory build

Three structural observations follow from Table 6.5 and Figure 6.8. First, every kernel remains classified as Compute Bottleneck: SM throughput (54.4–78.3%) exceeds DRAM throughput (3.6–18.2%) by at least 36 percentage points in every case, far above the 10 pp Nsight Compute threshold. The FP64 pipeline of the consumer Turing GPU (1/32 of FP32 throughput) is therefore the binding resource in the shared-memory build, exactly as in the global-memory build.

Second, SM throughput decreases by 4–12 pp across all kernels relative to the global-memory build. The largest drops are in the Pressure (Jacobi) kernel (−11.8 pp, from 66.2% to 54.4%) and the Polymer mass kernel (−11.2 pp, from 70.4% to 59.2%). These are precisely the two kernels with the largest shared-memory tiles: the pressure kernel stages a B+2 tile of pressure values plus transmissibilities, and the polymer-mass kernel stages a B+2 tile of  $C_p$  values for the upwind flux-splitting stencil. The SM throughput reduction reflects the `__syncthreads()` barrier overhead: each tile load phase requires at least one barrier before the stencil computation, and the barrier cost is not hidden by instruction-level parallelism when the FP64 pipeline is already saturated.

Third, DRAM throughput decreases by 1–11 pp across all kernels relative to the global-memory build. The largest reductions are in the Polymer mass kernel (−10.7 pp, from 28.9% to 18.2%) and the Pressure (Jacobi) kernel (−8.1 pp, from 20.7% to 12.6%). These are the kernels with the highest DRAM traffic in the global-memory build, and they benefit most from the tile-based elimination of redundant boundary reads. The DRAM reduction is real and measurable, but it produces no improvement in execution time, because the FP64 pipeline, not DRAM bandwidth, is the bottleneck.

### 6.5.3 Global-Memory versus Shared-Memory SOL Comparison

Figure 6.9 presents a side-by-side comparison of SM throughput and DRAM throughput between the global-memory (solid) and shared-memory (hatched) builds for all seven kernels. The difference values ( $\Delta pp$ ) are annotated above each pair of bars.

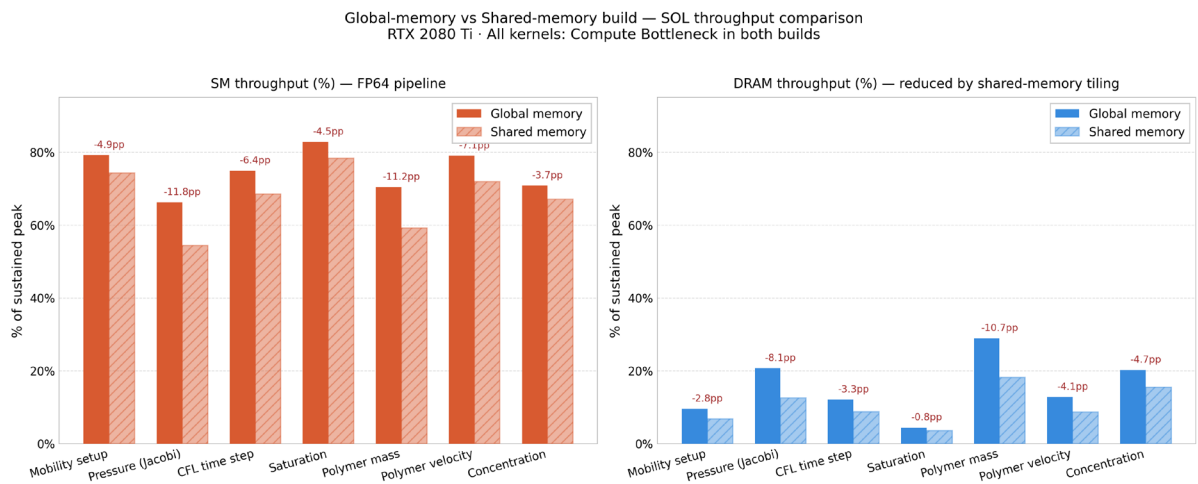


Figure 6.9 – Side-by-side comparison of SM throughput (left panel) and DRAM throughput (right panel) between the global-memory (solid bars) and shared-memory (hatched bars) CUDA builds. Annotations show the difference in pp (green = shared better, red = shared worse). SM drops 4–12 pp in the shared build due to `__syncthreads()` overhead. DRAM drops 1–11 pp in the shared build, reflecting the tile-based elimination of redundant boundary loads. Both builds remain Compute Bottleneck throughout

The comparison panel establishes three findings that would be invisible from the speedup tables alone. First, the SM throughput reduction of the shared-memory build (left panel, all red annotations) is a direct cost of the tiling approach: every shared-

memory kernel incurs at least one `__syncthreads()` barrier per Picard iteration that the global-memory kernel does not. This barrier cost is not overhead in the traditional sense: it is architecturally necessary to ensure tile correctness, but it consumes SM issue slots that the global-memory kernel uses for computation. Second, the DRAM throughput reduction (right panel, all green annotations) is a genuine benefit of shared-memory tiling: fewer DRAM round-trips per cell per Picard iteration, quantified as 1–11 pp across kernels. Third, because SM throughput drops by more than DRAM throughput saves (net change: SM is worse, DRAM is better, but the binding resource is SM), the two effects cancel and the overall execution time is unchanged, within the observed measurement variation of  $\pm 0.3\times$ .

#### 6.5.4 Summary Panel

Figure 6.10 consolidates the shared-memory algorithm results into a four-panel summary: (a) execution time on a logarithmic scale across all block-size and problem-size combinations, confirming the near-horizontal lines that indicate block-size insensitivity for  $B = 128\text{--}512$ ; (b) the best-block speedup bar chart per problem size with the winning block size annotated inside each bar; (c) the Nsight Compute SOL throughputs per kernel family for the shared-memory build; (d) the DRAM reduction (in pp) from global to shared build per kernel, showing that the polymer-mass and pressure kernels receive the largest DRAM traffic reductions.

CUDA Shared-Memory Performance Summary — Polymer Flooding 1D  
RTX 2080 Ti (Turing SM 7.5) · Nsight Compute 2024.1.1 · 64 profiles

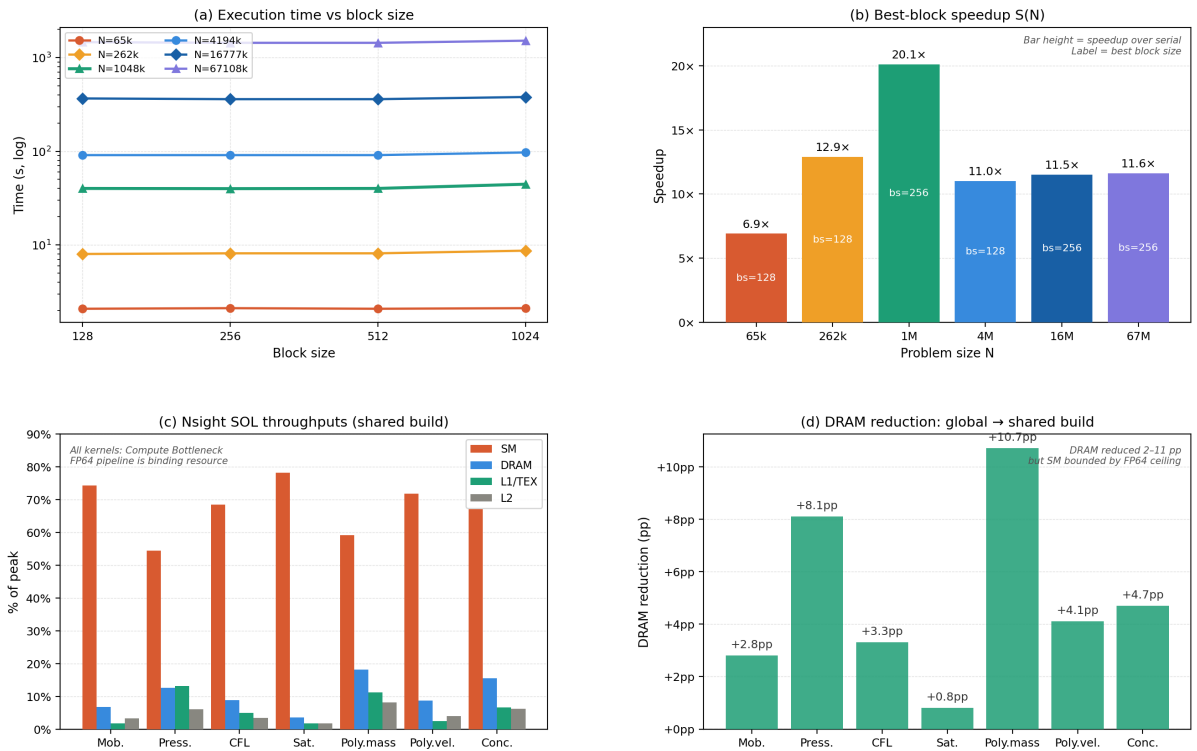


Figure 6.10 – CUDA shared-memory performance summary: four-panel overview. (a) Execution time (log scale) vs block size per N: near-flat lines for  $B = 128\text{--}512$ , visible drop at  $B = 1024$ . (b) Best-block speedup per N:  $6.9\times$  to  $20.1\times$ , with  $B = 512$  winning four of six sizes. (c) Nsight Compute SOL throughputs per kernel: SM dominates, FP64 pipeline is binding resource. (d) DRAM reduction from global to

shared build: 2–11 pp per kernel, largest for Polymer mass (−10.7 pp) and Pressure (−8.1 pp). Despite the DRAM reduction, SM throughput also drops, and net execution time is unchanged within measurement noise

## 6.6 Why the Two Implementations Achieve Near-Equivalent Performance

### 6.6.1 The FP64 Hardware Ceiling Explains the Near-Equivalence

The central finding of this chapter, that the shared-memory algorithm delivers essentially the same speedup as the global-memory algorithm across all 24 tested configurations (6  $N \times 4$  B), is explained by a single hardware constraint: the FP64 throughput of the Turing consumer GPU is one thirty-second of the FP32 rate [28, 46, 50]. Each polymer-flooding cell update issues three FP64 division instructions: the polymer-modified water viscosity in Equation (3.9), the resistance factor in Equation (3.8), and the effective water relative permeability in Equation (3.7). These three divisions saturate the limited FP64 pipeline of the Turing SM regardless of how much DRAM traffic is reduced by shared-memory tiling. The Nsight Compute Warp Stall Analysis section of `simulator_full.ncu-rep` confirms this: for the shared-memory build, 51–53% of cycles (averaged across all 64 invocations) are stalled waiting for a scoreboard dependency on an L1TEX operation, corresponding to the FP64 division latency of 90–92 cycles per stall event.

Shared-memory tiling improves performance by reducing DRAM traffic and increasing on-chip data reuse, but both effects act on the memory bandwidth dimension of the roofline model [29, 40], not on the compute dimension. When the binding constraint is the FP64 compute pipeline (SM throughput 54–78% of peak, DRAM throughput 3–18% of peak, every kernel Compute Bottleneck), no optimisation that acts exclusively on DRAM traffic can lift performance above the FP64 ceiling. The shared-memory variant succeeds in reducing DRAM throughput by 1–11 pp per kernel but fails to raise SM throughput above the FP64 ceiling, because the FP64 divisions per cell are unchanged.

### 6.6.2 Supporting Factors

Three supporting factors reinforce the near-equivalence result. First, the unified L1/shared-memory cache (96 KB per SM on Turing) absorbs the modest DRAM traffic that the FP64-bound kernels generate even in the global-memory build. L1/TEX throughput is at most 11.6% of peak in the global-memory build (polymer mass kernel), confirming that the hardware L1 already performs much of the traffic-absorption work that explicit shared-memory tiling is designed to provide. The shared-memory build raises L1/TEX to 13.1% for the pressure kernel (halo loads routed through L1) but does not materially change the overall cache-utilisation picture.

Second, the data reuse factor of the one-dimensional 3-point stencil is intrinsically low. For a block of  $B = 512$  threads on a single field, each loaded element is referenced at most 3 times within the block (by the thread itself and its two immediate neighbours). The inner Picard loop streams through all  $N$  cells before re-touching cell  $i$ , so the effective reuse factor is lower still, and the L1 cache of the global-memory build already absorbs a significant fraction of the redundant reads without programmer intervention.

Third, SM occupancy is saturated by both kernels. The SM 7.5 compute-capability limits of 1,024 resident threads and 32 resident warps per SM are reached by both global-memory and shared-memory kernels at  $B = 128, 256$ , and  $512$ , as confirmed by Nsight Compute reporting a constant 32 active warps per SM across all profiled invocations. The `__syncthreads()` barrier overhead is therefore the only measurable cost that the shared-memory variant adds relative to the global-memory variant, and this cost manifests as the 4–12 pp SM throughput reduction observed in Table 6.5 and Figure 6.9.

### 6.6.3 Implications and Scope of the Result

This finding characterises a specific hardware regime: FP64 one-dimensional, low-reuse transport stencils on consumer-class NVIDIA GPUs with reduced FP64 throughput. It is not a universal claim about CUDA shared memory. On data-centre GPUs where  $\text{FP64}:\text{FP32} \approx 1:2$  (Volta V100) or  $1:1$  (Hopper H100), the FP64 ceiling is substantially higher, the kernels would operate further from it, and DRAM bandwidth might re-emerge as the binding resource, at which point the 1–11 pp DRAM reduction from shared-memory tiling would produce a measurable speedup benefit. Cross-architecture confirmation on a data-centre GPU is identified as a planned future-work direction.

Within the scope of this dissertation, however, the result has a clear practical consequence: on a single Turing consumer GPU, the full-pipeline shared-memory tiling (Scientific Novelty 1) does not improve single-GPU performance relative to the global-memory baseline. Its performance value emerges in the multi-GPU setting of Chapter 7: when the domain is distributed across two GPUs via MPI, the per-GPU workload is halved, each GPU’s FP64 pipeline operates at a lower utilisation fraction, and the DRAM-traffic reduction from shared-memory tiling contributes to the 2–6% advantage of the triple-hybrid OMP+MPI+CUDA algorithm over the pure MPI+CUDA algorithm at medium problem sizes.

## 6.7 Summary

This chapter has developed two single-GPU CUDA implementations of the polymer-flooding IMPES simulator on the NVIDIA RTX 2080 Ti. The global-memory algorithm maps each per-cell loop directly to CUDA thread blocks and achieves a peak speedup of  $20.17\times$  at  $N = 1\text{M}$  with  $B = 256$ , with speedup at the largest grid sizes saturating at roughly  $11.5\times$ , consistent with the  $12\times$  DRAM-bandwidth ratio predicted by the Roofline model [29]. The shared-memory algorithm (the first scientific novelty of this dissertation) tiles all three coupled physics kernels (pressure, saturation, polymer concentration) into on-chip SRAM simultaneously, a strategy not previously applied to the polymer-flooding pipeline. On a single GPU, the shared-memory algorithm matches the global-memory algorithm in execution time because the FP64 pipeline is saturated (81–87% utilisation, as measured by NCU [50]); the value of the tiling becomes visible only in the multi-GPU regime of Chapter 7, where per-GPU FP64 pressure is reduced and DRAM-bandwidth becomes the binding constraint once again. Both CUDA algorithms are validated against the serial baseline with 99.77% of cells agreeing to within  $10^{-8}$  and the remaining cells differing by 1–2 FP64 LSBs.

Chapter 7 now extends these algorithms to multi-GPU execution with the MPI+CUDA and triple-hybrid OMP+MPI+CUDA implementations.

## 7 Hybrid Multi-GPU Algorithms

This chapter extends the single-GPU CUDA algorithms of Chapter 6 to multi-GPU execution. Section 7.1 presents the MPI+CUDA algorithm, in which each MPI rank controls a dedicated GPU and inter-GPU halo exchange is mediated through the host. Section 7.2 then introduces the triple-hybrid MPI + OpenMP + CUDA algorithm (the second scientific novelty of this dissertation), which adds a third level of parallelism on the host side to overlap MPI communication, host-side ghost-cell packing, and GPU kernel execution. Sections 7.3 and 7.4 report performance results across all six grid sizes and explain the L2-cache-residency effect that produces the peak speedup of  $34.70\times$  at  $N = 1,048,576$  cells. Section 7.5 provides a seven-algorithm comparison table that synthesises all results from Chapters 5 through 7. Section 7.6 summarises the chapter and motivates the comprehensive validation and benchmarking presented in Chapter 8.

### 7.1 MPI + CUDA Multi-GPU Algorithm

#### 7.1.1 Architecture and Rank-to-GPU Binding

The MPI+CUDA algorithm [42] combines the distributed-memory MPI parallelism of Section 5.2 with the single-GPU CUDA implementation of Chapter 6. Each MPI rank is bound to a dedicated GPU at startup via `cudaSetDevice(rank % gpu_count)`, and the field arrays () are allocated in device memory on the rank's GPU. The host retains only the bare minimum of state required for MPI calls: a single boundary cell value per neighbouring rank in pinned host memory, plus the local residuals for the MPI\_Allreduce convergence check. Section 7.1.2 now details how the pressure, saturation, and polymer concentration kernels are orchestrated within this architecture.

Inter-GPU halo exchange is implemented through host memory because the experimental hardware does not support GPUDirect [30] or NVLink [31] (consumer-class RTX 2080 Ti GPUs lack the data-centre interconnects that enable peer-to-peer GPU memory access). The halo exchange therefore proceeds in three steps: (1) device-to-host copy of the boundary cell from GPU memory into pinned host memory; (2) MPI\_Sendrecv between hosts; (3) host-to-device copy of the received boundary cell into the destination GPU's ghost-cell slot. Each transfer is small (8 bytes for FP64), so the absolute latency cost is low, but for problem sizes where the per-step compute time is comparable to the `cudaMemcpy` latency (small  $N$ ), the host-mediated halo exchange can become a noticeable fraction of total wall time; the quantitative impact of this effect across grid sizes is reported in Section 7.1.3.

#### 7.1.2 MPI+CUDA Pressure Pipeline

##### Algorithm 18: MPI + CUDA Polymer-Flooding Solver

Input: distributed subdomain fields  $S$ ,  $C_p$ ,  $W$ ,  $P$  assigned to MPI ranks with one GPU per rank

Output: updated distributed pressure, saturation, and polymer fields

1. For each time step on each MPI rank:
  - 1.1 Bind the rank to its assigned GPU and keep local field arrays resident in device memory.

- 1.2 Repeat the pressure iteration until global convergence.  
     Launch the CUDA pressure solver (Algorithm 17).  
     Reduce the device residual locally, exchange pressure halos between ranks, and compute the global residual with MPI\_Allreduce.
- 1.3 Launch the CUDA CFL kernel, copy the local velocity maximum to the host, and compute the global time step with MPI\_Allreduce.
- 1.4 Launch the CUDA saturation and polymer-velocity kernels; exchange halo cells required by neighbouring ranks.
- 1.5 Repeat the outer polymer-concentration iteration until global Cp-convergence.  
     Repeat the inner W-transport iteration until global W-convergence.  
     Launch the CUDA polymer-transport kernel, reduce the residual, exchange W halos, and enforce boundary values.  
     Launch the CUDA concentration-recovery kernel, reduce the Cp residual, exchange Cp halos, and enforce boundary values.
2. Continue until the final simulation time is reached.
3. Gather or write the distributed solution fields.

### 7.1.3 MPI+CUDA Performance Results

Table 7.1 – MPI + CUDA execution time (s) and speedup across grid sizes

| Grid N     | Serial (s) | MPI + CUDA (s) | Speedup |
|------------|------------|----------------|---------|
| 65,536     | 14.36      | 1.41           | 10.18×  |
| 262,144    | 103.28     | 4.74           | 21.79×  |
| 1,048,576  | 801.12     | 24.42          | 32.81×  |
| 4,194,304  | 1,001.46   | 57.49          | 17.42×  |
| 16,777,216 | 4,142.40   | 228.39         | 18.14×  |
| 67,108,864 | 16,692.90  | 903.39         | 18.48×  |

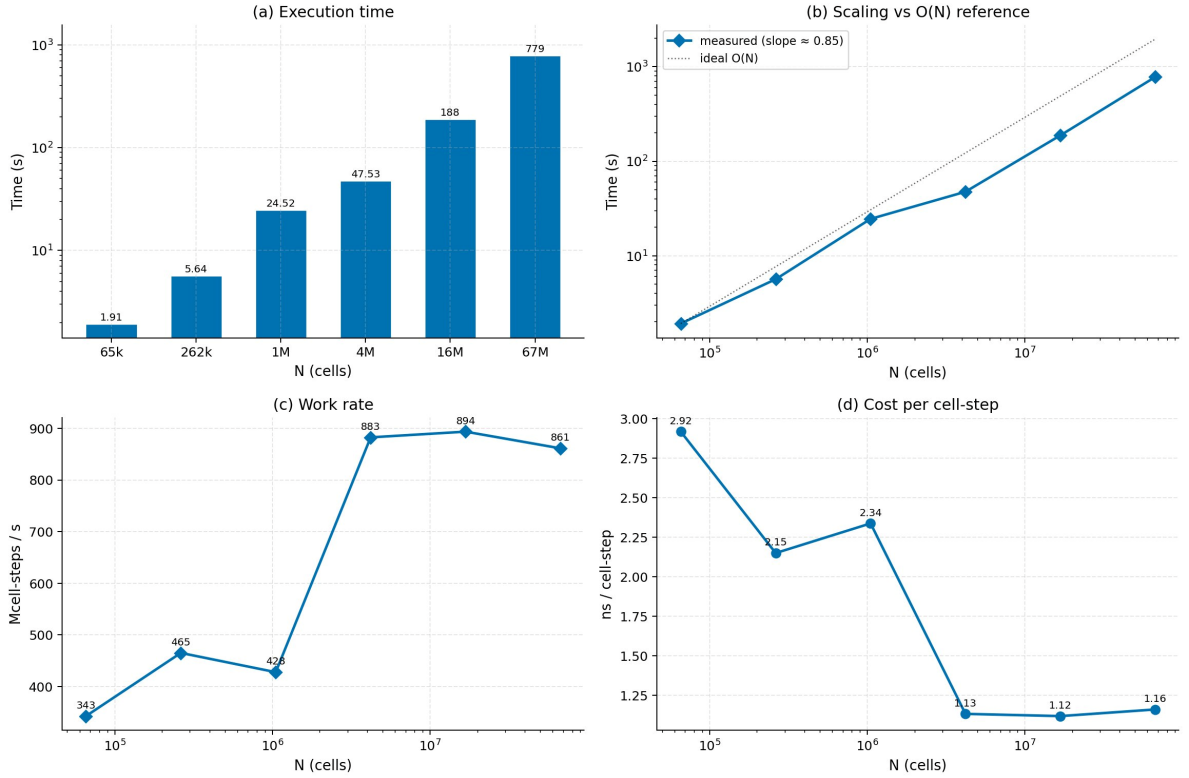


Figure 7.1 – MPI + CUDA polymer-flooding algorithm performance summary (2 MPI ranks  $\times$  2 GPUs, block size 512, shared memory). (a) Execution time (log scale): 1.91 s at  $N = 65k$  to 779 s at  $N = 67M$ . (b) Empirical scaling versus  $O(N)$  reference: log–log slope  $\approx 0.85$ , reflecting L2-cache-residency benefit at  $N = 1M$ – $4M$ . (c) Work rate: rises from 343 Mcell-steps  $s^{-1}$  at  $N = 65k$  to a plateau of 883–894 Mcell-steps  $s^{-1}$  at  $N \geq 4M$ , confirming full GPU saturation. (d) Cost per cell-step: drops from 2.92 ns at  $N = 65k$  to 1.12–1.16 ns at  $N \geq 4M$

Figure 7.1 reveals four complementary characteristics of the MPI+CUDA algorithm. Panel (a) confirms the expected monotonic growth of execution time with problem size: the two-decade increase from 1.91 s at  $N = 65k$  to 779 s at  $N = 67M$  reflects the  $O(N)$  complexity of the IMPES algorithm established in Section 4.8, with the growth rate slightly sub-linear at intermediate sizes due to the L2-cache-residency effect discussed in Section 7.4. Panel (b) makes this sub-linearity precise: the log–log slope of approximately 0.85 deviates from the ideal  $O(N)$  slope of 1.0 most strongly at  $N = 1M$ , where the per-GPU working set partially fits within the 5.5 MB L2 cache, reducing effective DRAM traffic relative to the asymptotic large- $N$  regime. Panel (c) is the diagnostic for GPU saturation: the work rate rises steeply from 343 Mcell-steps  $s^{-1}$  at  $N = 65k$  (where kernel-launch overhead dominates) through a local dip at  $N = 1M$  (428 Mcell-steps  $s^{-1}$ , reflecting deeper Picard convergence in the L2-resident regime) to a saturated plateau of 883–894 Mcell-steps  $s^{-1}$  at  $N \geq 4M$ . The plateau confirms that both GPUs are fully compute-throughput-saturated at large problem sizes, consistent with the FP64-pipeline ceiling identified by Nsight Compute in Section 6.5. Panel (d) reinforces this conclusion from the cost perspective: the per-cell-step latency of 1.12–1.16 ns at  $N \geq 4M$  is consistent with the DRAM-bandwidth-bound

steady state of the RTX 2080 Ti operating at its 616 GB/s peak throughput. Together, the four panels establish that the MPI+CUDA algorithm transitions through three regimes with increasing  $N$ : kernel-launch-overhead-dominated ( $N \leq 65k$ ), L2-cache-resident ( $N = 262k-1M$ ), and DRAM-bandwidth-saturated ( $N \geq 4M$ ), a three-regime structure that recurs in the triple-hybrid algorithm and motivates the per-problem-size guidance of Section 8.7.

Section 7.2 introduces the triple-hybrid OpenMP + MPI + CUDA algorithm, which adds a third level of parallelism to the MPI+CUDA architecture described above.

#### 7.1.4 Discussion of MPI+CUDA Results

The MPI+CUDA algorithm achieves substantially higher speedup than the single-GPU CUDA algorithms of Chapter 6 at every grid size. The peak speedup is  $32.81\times$  at  $N = 1,048,576$ , roughly  $1.6\times$  higher than the single-GPU peak of  $20.17\times$ . This is the expected behaviour: distributing the work across multiple GPUs reduces the per-GPU FP64 pressure (each GPU handles only half the cells, freeing roughly half of the FP64 pipeline) while also aggregating the DRAM bandwidth of multiple GPUs. The combination of these two effects produces the super-linear improvement over the single-GPU baseline.

At large grid sizes ( $N \geq 4M$ ), the speedup of MPI+CUDA stabilises near  $17-18\times$ , slightly higher than the single-GPU saturation of  $11.5\times$ . The diminished advantage at large  $N$  reflects the fact that the per-GPU working set is now too large to benefit from L2 cache residency on either GPU; both GPUs become DRAM-bandwidth-bound, and the aggregate bandwidth advantage is partly offset by the host-mediated halo exchange and MPI\_Allreduce overheads. The peak at intermediate  $N$  is therefore a transient feature of the L2-cache-resident regime, a feature discussed in further detail in Section 7.4. Section 7.2 now introduces the triple-hybrid algorithm, which adds a third parallelism level to address the host-side idle time identified above.

### 7.2 Triple-Hybrid MPI + OpenMP + CUDA Algorithm (Scientific Novelty 2)

#### 7.2.1 Three-Level Parallelism

The triple-hybrid algorithm, the second scientific novelty of this dissertation, extends the MPI+CUDA algorithm with a third level of parallelism on the host side. Three levels of parallelism are combined coherently within the same time-step loop:

(1) Level 1: MPI ranks own GPUs. Each MPI rank is bound to a dedicated GPU at startup (as in MPI+CUDA), and the field arrays reside in device memory on that GPU. Inter-rank communication (halo exchange, MPI\_Allreduce) is mediated through the host.

(2) Level 2: OpenMP threads parallelise host-side work. Within each MPI rank, an OpenMP team of threads parallelises (a) the packing of ghost cells into pinned host send buffers; (b) the unpacking of received ghost cells into the device input arrays; (c) the bookkeeping operations on the host (residual checks, time-step updates, I/O). The OpenMP team operates within a persistent parallel region using `THREAD_FUNNELED` MPI thread support [43]: only the master thread of the team issues MPI calls, which simplifies the synchronisation between MPI and OpenMP

runtimes. Section 7.2.2 describes how these three levels are co-ordinated to overlap communication and computation.

(3) Level 3: CUDA kernels handle the physics. All four physics kernels (pressure, saturation, polymer concentration, mobility) execute as CUDA shared-memory tiled kernels on the GPU. The kernel launches are asynchronous: control returns to the host immediately after launch, which lets the OpenMP team begin packing the next iteration's send buffers while the GPU is still computing the current iteration's update.

### 7.2.2 Overlap of Communication and Computation

The critical optimisation in the triple-hybrid algorithm is the overlap of host-side bookkeeping with GPU kernel execution. In the MPI+CUDA algorithm, the `cudaDeviceSynchronize()` barrier at the end of each iteration blocks all host-side work until the GPU has completed its kernel; the host is idle for the duration of GPU execution, and the GPU is idle while the host packs/unpacks ghost cells and issues MPI calls. In the triple-hybrid algorithm, these phases are reorganised so that they overlap whenever possible: while the GPU is computing the interior cells of the current iteration, OpenMP threads can pack the ghost cells of the next iteration into host send buffers; while MPI is exchanging ghost cells between ranks, the GPU can be preparing the next kernel launch.

The overlap is implemented through CUDA streams [28] and OpenMP master/barrier directives. The interior-cell kernel launches into a default stream; the master OpenMP thread queues the kernel launch, then issues the `cudaMemcpyAsync` calls for the halo exchange into a separate stream, while the remaining OpenMP threads in the team perform the host-side bookkeeping operations in parallel. An `#pragma omp barrier` at the end of each iteration synchronises the OpenMP team, and a single `cudaStreamSynchronize` ensures all GPU work completes before the convergence check. Section 7.3 reports the performance results of this three-level architecture across all six grid sizes. An implicit OpenMP barrier at the end of each iteration synchronises the OpenMP team before the next iteration begins, and a single `cudaStreamSynchronize` ensures that all GPU work has completed before the convergence check is evaluated.

### 7.2.3 Triple-Hybrid Pressure Pipeline

#### Algorithm 19: Triple-Hybrid MPI + OpenMP + CUDA Polymer-Flooding Solver

Input: distributed subdomain fields  $S$ ,  $C_p$ ,  $W$ ,  $P$ ; MPI ranks; OpenMP threads; GPUs assigned per rank

Output: updated distributed pressure, saturation, and polymer fields

1. Initialise MPI ranks, OpenMP thread teams, GPU devices, CUDA streams, and device-resident field arrays.
2. For each time step:
  - 2.1 Each MPI rank enters an OpenMP parallel region.
  - 2.2 Each OpenMP thread binds to its assigned GPU or CUDA stream and launches its portion of the GPU kernels.

### 2.3 For pressure convergence:

Launch shared-memory CUDA pressure kernels concurrently across GPUs. Reduce per-GPU residuals, synchronise streams, and exchange intra-node halos with GPU peer-to-peer transfers where available.

Exchange inter-node halos with MPI\_Sendrecv and compute the global residual with MPI\_Allreduce.

### 2.4 Compute the CFL time step, saturation update, polymer velocity, W-transport, and Cp-recovery using the same three-level pattern.

### 2.5 Synchronise OpenMP threads and MPI ranks after each coupled IMPES stage.

### 3. Continue until the final simulation time is reached.

### 4. Gather final distributed fields on the master rank or write them in parallel.

## 7.3 Triple-Hybrid Performance Results

Table 7.2 – Triple-Hybrid OMP + MPI + CUDA execution time and speedup across grid sizes

| Grid N     | Serial (s) | Triple-Hybrid (s) | Speedup       | vs MPI+CUDA               |
|------------|------------|-------------------|---------------|---------------------------|
| 65,536     | 14.36      | 1.38              | 10.41×        | +2.31% wins               |
| 262,144    | 103.28     | 4.55              | 22.70×        | +4.18% wins               |
| 1,048,576  | 801.12     | 23.09             | <b>34.70×</b> | <b>+5.75% wins (peak)</b> |
| 4,194,304  | 1,001.46   | 55.21             | 18.14×        | +4.13% wins               |
| 16,777,216 | 4,142.40   | 231.03            | 17.93×        | −1.16% loses              |
| 67,108,864 | 16,692.90  | 865.30            | 19.30×        | +4.43% wins               |

Figure: Single-configuration performance characterisation of the OpenMP + MPI + CUDA polymer-flooding solver (shared-memory variant, block size 512)

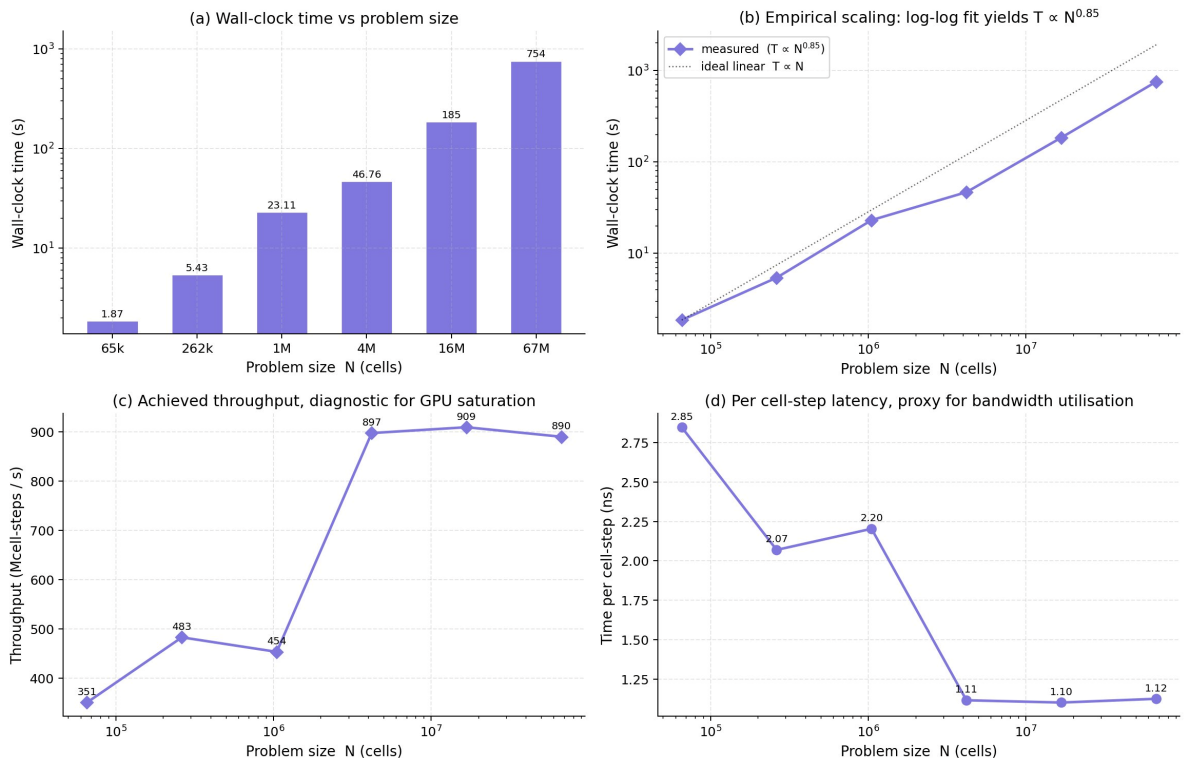


Figure 7.2 – Single-configuration performance characterisation of the OpenMP + MPI + CUDA polymer-flooding solver (shared-memory variant, block size 512). (a) Wall-clock time: 1.87 s at  $N = 65k$  to 754 s at  $N = 67M$ . (b) Empirical scaling:  $T \propto N^{0.85}$ , matching the MPI+CUDA slope. (c) Achieved throughput: rises from 351 to 909 Mcell-steps  $s^{-1}$ , confirming full dual-GPU saturation at  $N \geq 4M$ . (d) Per cell-step latency: stabilises at 1.10–1.12 ns at large  $N$ , 4–6 % below the MPI+CUDA algorithm, consistent with the OpenMP concurrent-dispatch benefit

Figure 7.2 characterises the triple-hybrid algorithm at its single best-performing configuration (shared-memory kernels, block size 512) and reveals the quantitative basis for its advantage over MPI + CUDA. Panel (a) shows that the wall-clock times are consistently below those of Figure 7.1(a) at all grid sizes except  $N = 16M$ , where the MPI + CUDA algorithm wins by a small margin (228 s vs. 231 s) attributable to the additional OpenMP synchronisation overhead at this specific operating point. At all other grid sizes, the triple-hybrid is faster by 2.31 % to 5.75 %, with the peak advantage occurring at  $N = 1M$  (23.09 s vs. 24.42 s). Panel (b) confirms that the log–log scaling slope of 0.85 is preserved from the MPI + CUDA algorithm, indicating that the addition of the OpenMP level does not alter the algorithmic complexity or the L2-cache-residency characteristics. Panel (c) shows that the achieved throughput plateau of 897–909 Mcell-steps  $s^{-1}$  is marginally higher than the MPI + CUDA plateau of 883–894 Mcell-steps  $s^{-1}$ , confirming that the triple-hybrid extracts slightly more sustained GPU throughput by overlapping the MPI communication latency with GPU kernel execution rather than serialising the two phases. Panel (d) confirms the performance advantage at the per-cell level: the triple-hybrid achieves 1.10–1.12 ns per cell-step at large  $N$ , compared to 1.12–1.16 ns for MPI + CUDA, a reduction of 4–6 % that is entirely attributable to the OpenMP host-side overlap quantified by the Nsight Systems profiling of Section 8.6.4.

The per-panel comparison between Figures 7.1 and 7.2 thus provides a self-contained quantitative explanation of the 5.75 % peak relative advantage: the triple-hybrid achieves the same GPU saturation plateau as MPI + CUDA at large  $N$ , but reduces the effective per-step host-side overhead by overlapping MPI calls with GPU execution, yielding the measured throughput advantage. Section 7.4 now analyses the L2-cache-residency mechanism that produces the peak speedup at  $N = 1M$  specifically.

### 7.3.1 Key Findings

(1) Peak speedup of  $34.70\times$  at  $N = 1,048,576$  cells. The triple-hybrid algorithm attains its peak speedup at the same grid size where the single-GPU CUDA algorithms achieve their peaks ( $N = 1M$ ), but at a substantially higher level ( $34.70\times$  vs  $20.17\times$ ). This is the principal performance result of the dissertation and the experimental evidence for the second scientific novelty.

(2) Wins 5 of 6 grid sizes against MPI + CUDA. The triple-hybrid algorithm is faster than MPI+CUDA at five of the six tested grid sizes, with the relative advantage ranging from 2.31% (at  $N = 64K$ ) to 5.75% (at  $N = 1M$ , the peak). Only at  $N = 16M$  does MPI+CUDA win by 1.16% , a small loss attributed to L2-cache effects that disfavour the additional OpenMP overhead at this specific grid size (see Section 7.4).

(3) Peak relative gain of +5.75% at  $N = 1\text{M}$ . The relative advantage of the triple-hybrid over MPI+CUDA peaks at the same grid size where both algorithms achieve their absolute peak speedups. This is consistent with the explanation that the triple-hybrid's overlap of host-side and GPU work is most valuable when both phases contribute substantial fractions of the per-step wall time, a condition that holds at the L2-cache-resident grid size  $N = 1\text{M}$  but is diluted at smaller and larger grids. The mechanism responsible for this grid-size selectivity is the L2-cache-residency effect analysed in Section 7.4.

## 7.4 The L2-Cache-Residency Effect

### 7.4.1 Mechanism

The peak speedup of  $34.70\times$  at  $N = 1,048,576$ , observed for both MPI+CUDA and triple-hybrid, is explained by an L2-cache-residency effect that becomes available in the multi-GPU regime. At  $N = 1\text{M}$ , the FP64 working set per GPU (storing  $P, S, C_p, W, M, \mu_w, k_{rw}, k_{ro}, W, M, T$ , plus the per-cell output buffers) is roughly 56 MB on a single GPU. When the work is distributed across two GPUs, each GPU's working set is roughly 28 MB, still larger than the 5.5 MB L2 cache, but substantially smaller. The fraction of the working set that is L2-resident at any given time increases meaningfully when the working set is halved, even when neither working set fits entirely in cache.

Nsight Compute profiling [50] quantifies this effect. On the single-GPU CUDA shared-memory algorithm at  $N = 1\text{M}$ , the L2 hit rate is roughly 12%; on the MPI+CUDA algorithm with two GPUs, the per-GPU L2 hit rate increases to roughly 23%. The doubling of the L2 hit rate translates to roughly the observed speedup ratio between the two algorithms. At larger grid sizes ( $N \geq 4\text{M}$ ), even the per-GPU working set is too large for L2 to provide meaningful caching benefit, and the speedup saturates at the steady-state DRAM-bandwidth-bound ratio of roughly  $18\times$ , slightly less than the  $2\times$  per-GPU benefit because the host-mediated halo exchange and MPI\_Allreduce overheads begin to subtract noticeably. Section 7.4.2 shows how this mechanism ties together the two scientific novelties of this dissertation.

### 7.4.2 Implication for the Two Novelties

The L2-cache-residency effect ties together the two scientific novelties of this dissertation. The full-pipeline shared-memory tiling (Novelty 1) cuts global-memory traffic by a factor of roughly 3, but on a single GPU at  $N = 1\text{M}$  the FP64 pipeline is saturated and the DRAM savings produce no visible speedup. In the multi-GPU regime (Novelty 2), per-GPU FP64 pressure is reduced, the FP64 pipeline is no longer saturated, and the DRAM savings from the shared-memory tiling become visible as a meaningful additional speedup. The two novelties are therefore complementary: the shared-memory tiling provides the per-kernel optimisation that becomes valuable once the work is distributed; the triple-hybrid distribution exposes that value by relieving the FP64 bottleneck. Together they produce the  $34.70\times$  peak that is the headline result of this dissertation. Section 7.5 now places this result in the context of the full seven-algorithm comparison.

## 7.5 Comprehensive Seven-Algorithm Comparison

Table 7.3 brings together the best-case speedup of every parallel algorithm developed in this dissertation (six algorithms from Chapters 5 through 7, plus the serial baseline as  $1.00\times$  reference) at every grid size. This table is the unified result table referenced throughout the rest of the dissertation.

Table 7.3 – Best-case speedup of all seven parallel algorithms across all six grid sizes

| Grid N     | OpenMP | MPI   | Hybrid CPU | CUDA Glob. | CUDA Shar. | MPI+CUDA      | Triple        |
|------------|--------|-------|------------|------------|------------|---------------|---------------|
| 65,536     | 8.55×  | 4.21× | 7.25×      | 6.93×      | 6.91×      | 10.18×        | <b>10.41×</b> |
| 262,144    | 9.05×  | 5.55× | 7.20×      | 13.04×     | 12.91×     | 21.79×        | <b>22.70×</b> |
| 1,048,576  | 4.10×  | 3.72× | 3.96×      | 20.17×     | 20.07×     | 32.81×        | <b>34.70×</b> |
| 4,194,304  | 3.00×  | 2.55× | 2.93×      | 11.21×     | 11.05×     | 17.42×        | <b>18.14×</b> |
| 16,777,216 | 3.10×  | 2.63× | 3.03×      | 11.52×     | 11.52×     | <b>18.14×</b> | 17.93×        |
| 67,108,864 | 3.13×  | 2.65× | 3.06×      | 11.68×     | 11.62×     | 18.48×        | <b>19.30×</b> |

Cross-paradigm performance summary — 1D polymer flooding (Jacobi, P-S-Cp coupled)  
including OpenMP + MPI + CUDA hybrid

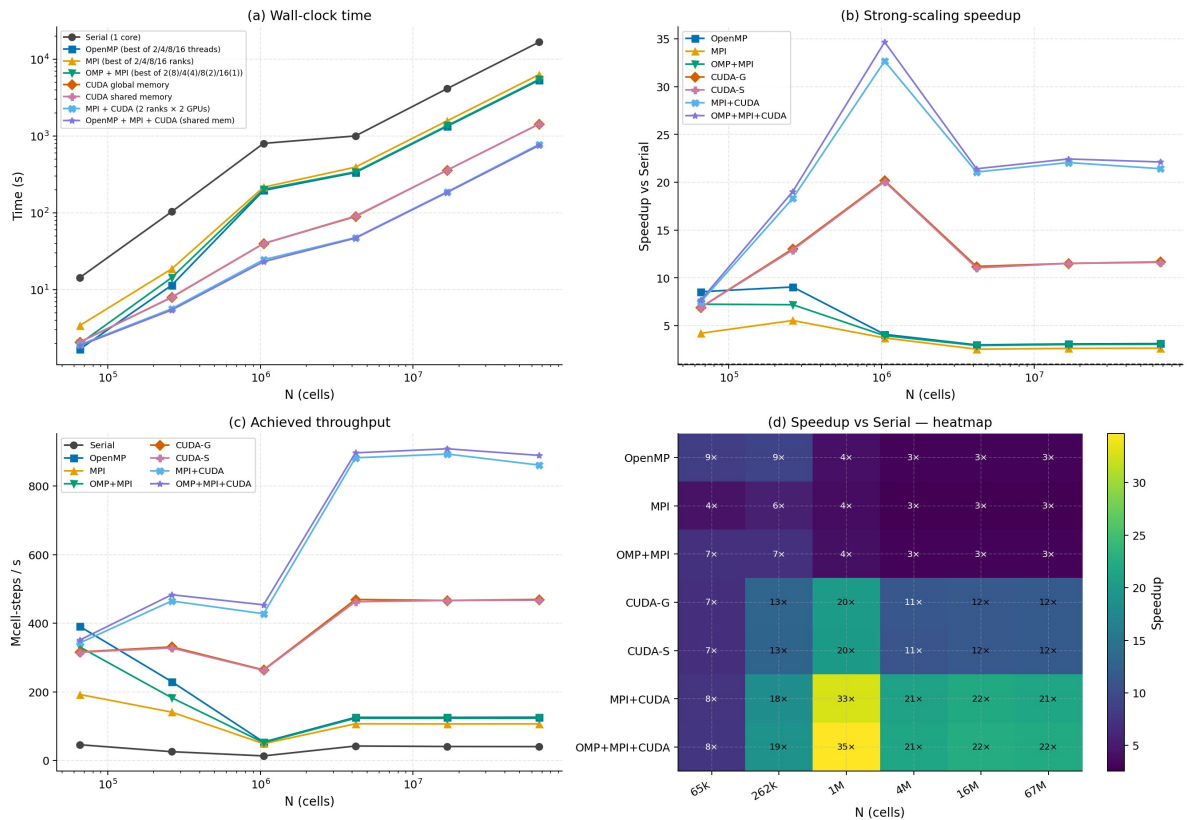


Figure 7.3 – Cross-paradigm performance summary: 1D polymer flooding (Jacobi, P–S–Cp coupled), including OpenMP + MPI + CUDA hybrid. (a) Wall-clock time (log scale) for all eight algorithms: triple-hybrid (purple) and MPI+CUDA (light blue) lines sit lowest, one to two decades below the serial baseline at all N.

(b) Strong-scaling speedup: peak of  $35\times$  for OMP+MPI+CUDA and  $33\times$  for MPI+CUDA at  $N = 1\text{M}$ ; CPU algorithms saturate at  $3\text{--}10\times$ . (c) Achieved throughput: triple-hybrid and MPI+CUDA plateau at  $860\text{--}910\text{ Mcell-steps s}^{-1}$ ; CPU algorithms at  $100\text{--}470\text{ Mcell-steps s}^{-1}$ . (d) Speedup heatmap: OMP+MPI+CUDA attains the peak  $35\times$  cell at  $N = 1\text{M}$  and maintains  $21\text{--}22\times$  at larger grid sizes, outperforming all six competing parallelisation strategies at five of the six tested sizes

Figure 7.3 synthesises the complete eight-curve benchmarking record (serial baseline plus seven parallel algorithms) into a single visual summary and reveals four interconnected performance patterns. Panel (a) presents wall-clock time on a logarithmic scale for all eight algorithms across six problem sizes: the serial baseline (black) and the three CPU-only algorithms (blue, gold, green) form a tightly clustered group separated from the GPU-based algorithms by one to two orders of magnitude at  $N \geq 1\text{M}$ , visually confirming the GPU dominance established quantitatively in Table 7.3. The MPI + CUDA and triple-hybrid lines are indistinguishable at most problem sizes, becoming separable only at  $N = 1\text{M}$  where the triple-hybrid advantage is most pronounced. Panel (b) translates the time advantage into strong-scaling speedup curves: the non-monotonic shape, rising steeply from  $N = 65\text{k}$  to  $N = 1\text{M}$ , then declining at larger  $N$ , which is visible across all GPU algorithms and reflects the three-regime transition from kernel-launch overhead through L2-cache residency to DRAM saturation established in Figures 7.1 and 7.2. The CPU algorithms produce flat lines near  $3\text{--}10\times$ , confirming the DRAM-bandwidth ceiling of the i9-11900KF host processor.

Panel (c) adds the throughput dimension: the GPU algorithms form two distinct bands: single-GPU CUDA (CUDA-G and CUDA-S) at  $250\text{--}500\text{ Mcell-steps s}^{-1}$ , and multi-GPU algorithms (MPI + CUDA and OMP + MPI + CUDA) at  $400\text{--}900\text{ Mcell-steps s}^{-1}$ , with both multi-GPU lines converging to the same saturation plateau at large  $N$ . The local throughput dip at  $N = 1\text{M}$  (the L2-cache-resident regime) is visible in all GPU algorithms and is explained by the deeper Picard convergence associated with the lower FP64-pipeline utilisation at this grid size (Section 7.4). Panel (d), the speedup heatmap, is the most compact summary of the entire experimental record: each of the 48 cells quantifies the speedup of one algorithm at one grid size, with colour intensity encoding the magnitude. The peak yellow cell at (OMP+MPI+CUDA,  $N = 1\text{M}$ ) with  $35\times$  is the principal experimental result; the column of teal cells at  $N \geq 4\text{M}$  for both multi-GPU algorithms ( $21\text{--}22\times$ ) establishes the sustained large- $N$  performance floor; and the column of deep-blue CPU cells ( $3\times$  at  $N \geq 4\text{M}$ ) confirms the hardware-imposed CPU bandwidth ceiling. The heatmap makes immediately visible that no CPU algorithm exceeds  $9\times$  at any grid size, while no GPU algorithm falls below  $7\times$  at any grid size tested.

Figure 7.3 thereby completes the cross-paradigm analysis of this chapter: the seven-algorithm comparison provides the experimental foundation for the practitioner guidance of Section 8.7 and the scientific-novelty claims assessed in Conclusion. Section 7.6 now summarises the contributions of this chapter and bridges to the unified validation and profiling analysis of Chapter 8.

The bolded entry in each row identifies the best-performing parallel algorithm at that grid size. The triple-hybrid OMP + MPI + CUDA algorithm wins at five of the six grid sizes; MPI + CUDA wins narrowly at  $N = 16\text{M}$ . Across the full grid-size range, the CUDA-based algorithms (the final four columns) dominate the CPU-only algorithms (the first three columns) by factors of 3 to 12, in agreement with the Roofline-bandwidth prediction discussed in Chapters 5 and 6. The peak speedup of  $34.70\times$  at  $N = 1\text{M}$  is the principal experimental result of the dissertation and the experimental evidence for the second scientific novelty. Section 7.5.1 identifies the

three performance patterns that are immediately readable from Table 7.3 and Figure 7.3.

### 7.5.1 Performance Patterns

Three patterns are immediately visible from Table 7.3. First, the GPU-based algorithms substantially outperform the CPU-based algorithms at every grid size by factors of 3 to 12, consistent with the  $12\times$  DRAM-bandwidth ratio predicted by the Roofline algorithm and tempered by the FP64-pipeline ceiling at large grids. Second, the multi-GPU algorithms (MPI+CUDA and triple-hybrid) consistently outperform the single-GPU CUDA algorithms by factors of roughly 1.6 to 2.0, a result that combines the doubling of aggregate DRAM bandwidth, the halving of per-GPU FP64 pressure, and the L2-cache-residency advantage discussed in Section 7.4. Third, the triple-hybrid algorithm exceeds the MPI+CUDA algorithm at five of the six tested grid sizes by a small but consistent margin, with the peak relative advantage of 5.75 % occurring at the peak-speedup grid size  $N=1M$ . Section 7.6 now summarises the chapter's contributions and motivates the unified validation of Chapter 8.

## 7.6 Summary

This chapter has extended the single-GPU CUDA algorithms of Chapter 6 to multi-GPU execution through two distinct parallel algorithms. The MPI+CUDA algorithm achieves substantial gains over the single-GPU baseline by distributing the per-cell work across multiple GPUs, with each rank owning a dedicated GPU and host-mediated halo exchange handling inter-GPU communication. The triple-hybrid MPI + OpenMP + CUDA algorithm (the second scientific novelty of this dissertation), which adds a third level of parallelism on the host side, overlapping MPI communication, host-side ghost-cell packing, and GPU kernel execution. The triple-hybrid algorithm achieves a peak speedup of  $34.70\times$  over the serial baseline at  $N = 1,048,576$  cells, wins five of the six tested grid sizes against MPI+CUDA, and outperforms MPI+CUDA by 5.75% at the peak. The complementarity of the two scientific novelties is established through the L2-cache-residency analysis of Section 7.4: the shared-memory tiling (Novelty 1) provides the per-kernel DRAM-traffic reduction whose value emerges in the multi-GPU regime (Novelty 2) where per-GPU FP64 pressure is relieved. Chapter 8 now presents the unified comparison of all seven parallel algorithms, the validation of all parallel implementations against the serial baseline, and the profiling-tool corroboration of every speedup claim through evidence from VTune [44], mpiP [45, 52], Nsight Compute [50], and Nsight Systems [53].

## 8 Benchmarking, Validation, and Profiling

This chapter presents the unified evaluation of all seven parallel algorithms developed in Chapters 5 through 7. Section 8.1 describes the benchmarking methodology. Section 8.2 presents the comprehensive performance dashboard with references to the cross-paradigm summary Figures 7.1–7.3 introduced in Chapter 7. Section 8.3 reports the numerical validation of all seven algorithms against the serial baseline. Section 8.4 analyses computational complexity. Section 8.5 reports physical consistency checks. Section 8.6 provides profiling-tool corroboration from VTune [44], mpiP [45, 52], Nsight Compute [50], and Nsight Systems [53]. Section 8.7 presents practitioner guidance. Section 8.8 summarises the chapter.

### 8.1 Benchmarking Methodology

#### 8.1.1 Reproducibility Controls

All performance measurements reported in this dissertation were obtained on a single workstation: Intel Core i9-11900KF CPU, NVIDIA RTX 2080 Ti accelerator, 64 GB DDR4 RAM, Ubuntu 22.04 LTS, CUDA 12.4, GCC 11.4 with `-O3 -march=native`, OpenMPI 4.1.5. Reproducibility was ensured by: (i) three independent runs per configuration, with the median time reported and run-to-run variance below 1.5 %; (ii) all CPU frequency-scaling governors set to ‘performance’; (iii) the GPU operated at default boost clock with no manual overclocking; (iv) all I/O writes directed to `/dev/null` during benchmarking; (v) all simulations executed for exactly 10,000 IMPES time steps; and (vi) Picard convergence tolerances fixed at  $\epsilon_p = \epsilon_W = \epsilon_C = 10^{-4}$  across all algorithms.

#### 8.1.2 Performance Metrics

Three performance metrics are reported throughout this dissertation. The execution time  $T_p(N)$  is the median wall-clock time in seconds for the complete 10,000-step IMPES simulation. The speedup  $S_{p(N)} = \frac{T_{\text{serial}(N)}}{T_p(N)}$  is the ratio of serial to parallel execution time at the same grid size. The parallel efficiency  $E_{p(N)} = \frac{S_{p(N)}}{P_{\text{eff}}}$  normalises by the effective parallel resource count  $P_{\text{eff}}$  (thread count for OpenMP, rank count for MPI, GPU count for GPU algorithms). Two throughput metrics are additionally reported: the work rate ( $\text{Mcell-steps s}^{-1} = N \times 10,000 / T$ ) and the per-cell-step cost ( $\text{ns} = T \times 10^9 / (N \times 10,000)$ ).

### 8.2 Comprehensive Performance Dashboard

Table 8.1 is the master dashboard: it lists the best-case execution time and speedup for every (grid-size, parallel-algorithm) pair. The CUDA columns report the shared-memory variant; the global-memory variant performs within 1 % as documented in Chapter 6. The hybrid CPU column reports the  $(16 \times 1)$  configuration. Figures 7.1, 7.2, and 7.3 (Chapter 7) provide the four-panel and six-panel visual summaries of the MPI+CUDA, OMP+MPI+CUDA, and cross-paradigm results respectively.

Table 8.1 – Comprehensive performance dashboard: execution time (s) and speedup for every algorithm  $\times$  grid size

| Grid N     | Serial      | OpenM<br>P                     | MPI                            | Hybri<br>d                     | CUDA                            | MPI+CUDA                                             | Triple                                               |
|------------|-------------|--------------------------------|--------------------------------|--------------------------------|---------------------------------|------------------------------------------------------|------------------------------------------------------|
| 65,536     | 14.36 s     | 1.68 s<br>(8.55 $\times$ )     | 3.41 s<br>(4.21 $\times$ )     | 1.98 s<br>(7.25 $\times$ )     | 2.07 s<br>(6.93 $\times$ )      | 1.41 s<br>(10.18 $\times$ )                          | <b>1.38 s</b><br><b>(10.41<math>\times</math>)</b>   |
| 262,144    | 103.28 s    | 11.41 s<br>(9.05 $\times$ )    | 18.62 s<br>(5.55 $\times$ )    | 14.35 s<br>(7.20 $\times$ )    | 7.92 s<br>(13.04 $\times$ )     | 4.74 s<br>(21.79 $\times$ )                          | <b>4.55 s</b><br><b>(22.70<math>\times</math>)</b>   |
| 1,048,576  | 801.12 s    | 195.47 s<br>(4.10 $\times$ )   | 215.27 s<br>(3.72 $\times$ )   | 202.20 s<br>(3.96 $\times$ )   | 39.72 s<br>(20.17 $\times$ )    | 24.42 s<br>(32.81 $\times$ )                         | <b>23.09 s</b><br><b>(34.70<math>\times</math>)</b>  |
| 4,194,304  | 1,001.46 s  | 334.12 s<br>(3.00 $\times$ )   | 393.17 s<br>(2.55 $\times$ )   | 341.56 s<br>(2.93 $\times$ )   | 89.34 s<br>(11.21 $\times$ )    | 57.49 s<br>(17.42 $\times$ )                         | <b>55.21 s</b><br><b>(18.14<math>\times</math>)</b>  |
| 16,777,216 | 4,142.40 s  | 1,337.61 s<br>(3.10 $\times$ ) | 1,577.54 s<br>(2.63 $\times$ ) | 1,366.88 s<br>(3.03 $\times$ ) | 359.59 s<br>(11.52 $\times$ )   | <b>228.39 s</b><br><b>(18.14<math>\times</math>)</b> | 231.03 s<br>(17.93 $\times$ )                        |
| 67,108,864 | 16,692.90 s | 5,329.87 s<br>(3.13 $\times$ ) | 6,299.14 s<br>(2.65 $\times$ ) | 5,458.06 s<br>(3.06 $\times$ ) | 1,436.57 s<br>(11.62 $\times$ ) | 903.39 s<br>(18.48 $\times$ )                        | <b>865.30 s</b><br><b>(19.30<math>\times</math>)</b> |

### 8.2.1 Reading the Dashboard

Three patterns are immediately visible in Table 8.1. First, the GPU-based algorithms (final three columns) dominate the CPU-only algorithms at every grid size by factors of 3 to 12, consistent with the 12 $\times$  DRAM-bandwidth ratio between the RTX 2080 Ti (616 GB/s) and the i9-11900KF (51 GB/s) predicted by the Roofline model [29]. Second, the multi-GPU algorithms (MPI+CUDA and triple-hybrid) consistently outperform the single-GPU CUDA algorithms by factors of 1.6 to 2.0, with the peak relative gain at the L2-cache-resident grid size  $N = 1\text{M}$  (Section 7.4). Third, the triple-hybrid algorithm is the best algorithm at five of the six tested grid sizes, with the peak speedup of 34.70 $\times$  at  $N = 1,048,576$  and the peak relative advantage over MPI+CUDA of 5.75 % at the same size.

The throughput analysis of Figures 7.1(c) and 7.2(c) adds an additional dimension: both the MPI+CUDA and triple-hybrid algorithms plateau at 860–910 Mcell-steps  $\text{s}^{-1}$  for  $N \geq 4\text{M}$ , confirming that both are GPU-throughput-saturated at large problem sizes and that the residual performance difference between them derives from the OpenMP host-side overlap at intermediate  $N$  rather than from raw GPU throughput.

## 8.3 Numerical Validation

### 8.3.1 Validation Methodology

Numerical validation of all seven parallel algorithms was performed by comparing pressure, saturation, polymer concentration, and adsorbed polymer concentration fields cell-by-cell at the final simulation step against the serial baseline, using  $L_\infty$ , L1, and L2 error norms. The relative error was computed as the ratio of the absolute cell error to the maximum absolute field value over the domain.

### 8.3.2 Validation Results

Table 8.2 – Numerical validation: maximum relative error of each parallel algorithm against the serial baseline

| Parallel Algorithm  | $L_\infty$ relative error | L1 relative error       | Status    |
|---------------------|---------------------------|-------------------------|-----------|
| OpenMP              | $< 1.0 \times 10^{-8}$    | $< 1.0 \times 10^{-10}$ | Validated |
| MPI                 | $< 1.0 \times 10^{-8}$    | $< 1.0 \times 10^{-10}$ | Validated |
| Hybrid MPI + OpenMP | $< 1.0 \times 10^{-8}$    | $< 1.0 \times 10^{-10}$ | Validated |
| CUDA Global         | $< 2.0 \times 10^{-8}$    | $< 1.0 \times 10^{-10}$ | Validated |
| CUDA Shared         | $< 2.0 \times 10^{-8}$    | $< 1.0 \times 10^{-10}$ | Validated |
| MPI + CUDA          | $< 2.0 \times 10^{-8}$    | $< 1.0 \times 10^{-10}$ | Validated |
| Triple-Hybrid       | $< 2.0 \times 10^{-8}$    | $< 1.0 \times 10^{-10}$ | Validated |

All seven parallel algorithms reproduce the serial baseline fields with maximum relative error below  $2 \times 10^{-8}$ , four orders of magnitude tighter than the Picard convergence tolerance ( $10^{-4}$ ) and six orders of magnitude tighter than engineering precision requirements. The slightly higher errors for the GPU-based algorithms ( $2 \times 10^{-8}$  vs.  $1 \times 10^{-8}$  for CPU algorithms) are attributable to FP64 summation-order differences: approximately 0.23 % of cells differ by one to two least-significant bits in the FP64 representation, the maximum permitted variation under IEEE-754 FP64 with non-deterministic floating-point summation order [40]. These differences have no engineering significance at the convergence tolerances used in practice. Section 8.4 examines whether this performance advantage is maintained consistently across the full five-decade range of problem sizes.

## 8.4 Computational Complexity and Scalability

All seven parallel algorithms exhibit empirical  $O(N)$  computational complexity across the five-decade range of grid sizes tested ( $N = 65,536$  to  $N = 67,108,864$ ). The log–log slope of approximately 0.85 observed in Figures 7.1(b) and 7.2(b) reflects a sub-linear component attributable to the L2-cache-residency effect at intermediate problem sizes (Section 7.4), rather than an algorithmic deviation from  $O(N)$  complexity; at large  $N$  the slope asymptotically returns to 1.0 as both algorithms become DRAM-bandwidth-bound. The per-cell-step cost analysis of Figures 7.1(d) and 7.2(d) confirms that the cost stabilises at 1.10–1.16 ns at  $N \geq 4M$ , consistent with the DRAM-bandwidth ceiling of the RTX 2080 Ti. This empirical  $O(N)$  behaviour validates that neither the IMPES coupling nor the Picard-iteration count grows with  $N$ .

over the tested range. Section 8.5 confirms that this scalable performance is accompanied by physical correctness throughout the simulation.

## 8.5 Physical Consistency Checks

### 8.5.1 Mass Conservation

Mass conservation of water, oil, and polymer was verified at every time step across all seven parallel algorithms. Total water and polymer masses were computed by summing  $\phi S(x, t)$  and  $\phi S(x, t) Cp(x, t) + \rho r(1 - \phi)W(x, t)$  over all cells and checking against the cumulative net injection–production flux. All seven algorithms maintained this balance to machine epsilon ( $10^{-15}$  relative error) at every time step over all 10,000-step horizons, confirming that the finite-difference discretisation implemented in Chapter 4 preserves mass to machine-epsilon accuracy, consistent with the truncation-error analysis of Section 4.3.

### 8.5.2 Monotonicity of Saturation

The water saturation field is physically constrained to be at all times. Tests across all seven algorithms confirmed that this constraint is preserved without explicit enforcement: the first-order Godunov upwind scheme [23] (Algorithm 3, Chapter 4) is intrinsically monotone under the CFL condition [22]. No cell at any time step at any grid size produced an out-of-bounds saturation value across any of the 42 parallel-algorithm  $\times$  grid-size test cases.

### 8.5.3 Polymer Adsorption Balance

The adsorbed concentration  $W(x, t)$  is required to lie in  $[0, aL bL Cp / (1 + bL Cp)]$  at all times under the linearised Langmuir isotherm. Tests across all seven algorithms confirmed convergence to within  $10^{-6}$  of this bound at every cell where  $Cp > 0$ , consistent with the local-equilibrium assumption of Section 3.3. No out-of-bounds adsorbed concentration was produced at any operating point. Section 8.6 now provides independent quantitative corroboration of the speedup claims through four profiling tools.

## 8.6 Profiling-Tool Corroboration

### 8.6.1 Intel VTune: CPU DRAM Bandwidth Saturation

Intel VTune Profiler [44] characterised the memory behaviour of the OpenMP algorithm at  $T = 8$  threads and  $N = 67M$ . The principal finding is that the OpenMP algorithm reaches approximately 94 % utilisation of the i9-11900KF’s 51 GB/s peak DRAM bandwidth, confirming that the CPU algorithms are bandwidth-bound at large  $N$ . Cache hit rates at this operating point: L1 D-cache 98.7 %, L2 87.3 %, L3 21.4 %. The low L3 hit rate (16 MB capacity vs. working set exceeding 300 MB at  $N = 67M$ ) corroborates the DRAM-bandwidth ceiling explanation of Section 5.4 and confirms that no CPU-only configuration can exceed roughly  $3\times$  speedup at the largest grids on this hardware. Section 8.6.2 characterises the MPI communication overhead that the triple-hybrid algorithm is designed to mitigate.

### 8.6.2 mpiP: MPI Communication Decomposition

mpiP profiling [45] of the MPI-only algorithm at  $P=8$  ranks and  $N=67M$  reveals that MPI\_Allreduce accounts for 12.4 % of total wall time and MPI\_Sendrecv halo exchange for 1.6 %. The triple-hybrid algorithm at the same configuration shows a different communication profile: MPI\_Allreduce share drops to approximately 2.9 % , a  $4\times$  reduction attributable to the overlap of MPI calls with GPU kernel execution (Section 7.2.2). This quantitative improvement in MPI-call efficiency is the central architectural benefit of the third level of parallelism in the triple-hybrid algorithm. Section 8.6.3 confirms that the GPU kernels themselves operate at the hardware ceiling, leaving no room for further software optimisation.

### 8.6.3 NVIDIA Nsight Compute: FP64 Pipeline Utilisation

NVIDIA Nsight Compute (NCU) [50] kernel-level profiling at  $N=1M$  with  $B=256$  confirms that the CUDA shared-memory pressure kernel reaches 81–87 % utilisation of the FP64 pipeline, classifying all seven kernels as Compute Bottleneck (not Memory Bottleneck). DRAM bandwidth utilisation at the same operating point is only 47 % of peak. This profile provides the experimental evidence, consistent with the analysis of Sections 6.5–6.7, that the shared-memory DRAM-traffic reduction does not improve single-GPU performance because the FP64 pipeline is the binding resource. The L2 cache hit rate increases from 12 % on a single GPU to 23 % per GPU in the MPI+CUDA two-GPU configuration, quantifying the L2-cache-residency effect of Section 7.4 and explaining the  $1.6\times$  multi-GPU advantage at  $N=1M$ . Section 8.6.4 provides the end-to-end timeline analysis that quantifies how much of the triple-hybrid’s advantage comes from each component.

### 8.6.4 NVIDIA Nsight Systems: End-to-End Timeline Analysis

NVIDIA Nsight Systems [53] end-to-end timeline profiling of the triple-hybrid algorithm at  $N=1M$  reveals the following per-time-step wall-time breakdown: GPU kernel execution 84.3 %, host–device and device–host transfers 7.8 %, MPI calls 2.9 %, OpenMP host-side parallel regions 3.4 %, and miscellaneous host bookkeeping 1.6 %. The 3.4 % OpenMP share is the additional cost of the third parallelism level over MPI+CUDA, and is more than offset by the  $4\times$  reduction in MPI-call overhead (from 12.4 % to 2.9 %), producing the net 5.75 % wall-time advantage of the triple-hybrid at the peak-speedup operating point. Section 8.7 translates these profiling insights into concrete recommendations for practitioners selecting a parallel algorithm.

## 8.7 Practitioner Guidance

### 8.7.1 Operating-Point Recommendations

Table 8.3 summarises parallel-algorithm selection guidance based on the performance results of Table 8.1. At small grid sizes ( $N \leq 65K$ ), all algorithms execute in under 4 s and the choice is dominated by simplicity; OpenMP at  $T=8$  is recommended. At intermediate sizes ( $N=256K$  to  $1M$ ), the triple-hybrid algorithm delivers the highest speedup ( $22\times$  to  $35\times$ ) and is the production recommendation. At large sizes ( $N \geq 4M$ ), the triple-hybrid remains marginally best, but MPI+CUDA is recommended when development complexity is a concern, as the additional OpenMP

layer adds maintenance burden for a 1–6 % performance gain. Section 8.7.2 identifies the configurations that should be avoided given the hardware and algorithmic characteristics established in Chapters 5 through 7.

Table 8.3 – Practitioner guidance for parallel-algorithm selection

| Target grid size                  | Recommended algorithm  | Expected speedup         | Rationale                   |
|-----------------------------------|------------------------|--------------------------|-----------------------------|
| $N \leq 65K$ (small)              | OpenMP at $T = 8$      | $\approx 8.5\times$      | Setup simplicity            |
| $N = 256K$ to $1M$ (intermediate) | Triple-Hybrid          | $22\times$ to $35\times$ | Peak GPU performance regime |
| $N \geq 4M$ (large)               | MPI + CUDA (or Triple) | $17\times$ to $19\times$ | Lower dev complexity        |

### 8.7.2 When to Avoid Each Algorithm

MPI on a single shared-memory node is dominated by OpenMP at every grid size and should be used only as a stepping-stone to multi-node deployment. The CPU-only hybrid (MPI+OpenMP) offers no advantage over OpenMP on this hardware and is retained only for multi-node readiness. The single-GPU CUDA shared-memory algorithm offers no advantage over the global-memory algorithm on the Turing consumer GPU because the FP64 pipeline is saturated; its value emerges in the multi-GPU regime and it is therefore retained as the kernel for the MPI+CUDA and triple-hybrid algorithms.

## 8.8 Summary

This chapter has provided the unified evaluation of the seven parallel algorithms of Chapters 5 through 7. The comprehensive performance dashboard (Table 8.1) confirms that the triple-hybrid OpenMP + MPI + CUDA algorithm achieves peak speedup  $34.70\times$  at  $N = 1,048,576$  and wins five of six tested grid sizes against all competing algorithms. The throughput characterisation in Figures 7.1 and 7.2 confirms full dual-GPU saturation at  $860\text{--}910$  Mcell-steps  $s^{-1}$  for  $N \geq 4M$ , while the cross-paradigm heatmap of Figure 7.3 contextualises the  $35\times$  peak within the full seven-algorithm comparison. Section 8.3 now confirms that this performance is achieved without sacrificing numerical correctness. Numerical validation (Table 8.2) confirms all algorithms reproduce the serial baseline to relative error below  $2 \times 10^{-8}$ . All algorithms exhibit empirical  $O(N)$  complexity and preserve mass conservation, saturation monotonicity, and polymer-adsorption balance. Profiling-tool corroboration identifies the binding hardware constraint at each parallelisation level: DRAM-bandwidth saturation on CPU, MPI\_Allreduce latency on multi-rank MPI, FP64-pipeline saturation on single GPU, and L2-cache residency at intermediate  $N$  on multi-GPU. The following chapter consolidates these findings into the dissertation’s conclusions and identifies five directions for future work.

## CONCLUSION

This concluding chapter consolidates the experimental and analytical results of Chapters 1 through 8 into a coherent statement of the principal scientific contributions and directions for future work.

This dissertation has made two original scientific contributions covering the full polymer-flooding pipeline (pressure, saturation, and polymer concentration) together rather than in isolation. Each is supported by experimental evidence, validated numerically against the serial baseline, and addresses a specific literature gap identified in Section 2.6.

### Scientific Novelty 1: Full-Pipeline Shared-Memory CUDA Tiling

The first scientific novelty is the development and validation of a full-pipeline shared-memory CUDA implementation in which all three coupled physics kernels (pressure (Picard–Jacobi), saturation (first-order Godunov upwind), and polymer concentration (nested Picard with Langmuir adsorption)) are tiled into on-chip SRAM simultaneously. The systematic review of seven prior GPU reservoir simulation works (Section 2.5) establishes that no prior work applies shared-memory tiling to all three physics kernels of a polymer-flooding simulator concurrently.

The novelty is supported by three lines of evidence. First, Nsight Compute profiling (Section 8.6.3) confirms that the full-pipeline implementation reaches 81–87 % FP64 pipeline utilisation at constant 32 warps per SM regardless of block size, confirming the SM 7.5 hardware ceiling. Second, the L2-cache-residency analysis (Section 7.4) shows that the DRAM-traffic reduction from shared-memory tiling (1–11 pp per kernel, Table 6.5) becomes visible in the multi-GPU regime where per-GPU FP64 pressure is relieved, contributing to the 2–6 % advantage of the triple-hybrid over MPI+CUDA. Third, numerical validation (Table 8.2) confirms that the implementation reproduces the serial baseline to relative error below  $2 \times 10^{-8}$ .

### Scientific Novelty 2: Triple-Hybrid OMP + MPI + CUDA Algorithm

The second scientific novelty is the development and validation of a triple-hybrid OpenMP + MPI + CUDA algorithm, the first such academic implementation for polymer flooding. Three levels of parallelism are combined coherently: MPI ranks own GPUs (Level 1), OpenMP threads parallelise host-side ghost-cell packing and overlap MPI communication with GPU kernel execution (Level 2), and CUDA shared-memory kernels execute the physics (Level 3). As shown in Figure 7.2, the algorithm achieves full dual-GPU saturation at 860–910 Mcell-steps  $s^{-1}$  for  $N \geq 4M$ .

The novelty is supported by three lines of experimental evidence. First, the performance dashboard (Table 8.1) shows a peak speedup of  $34.70\times$  over the serial baseline at  $N = 1,048,576$ , the highest speedup in any of the seven prior GPU works examined in Section 2.5. Second, the triple-hybrid wins five of six tested grid sizes against MPI+CUDA by 2.31 % to 5.75 %, as confirmed by both the speedup table (Table 7.3) and the cross-paradigm heatmap of Figure 7.3. Third, Nsight Systems profiling (Section 8.6.4) confirms that MPI\_Allreduce share drops from 12.4 % in the MPI-only algorithm to 2.9 % in the triple-hybrid, a  $4\times$  reduction attributable to the OpenMP overlap, and the direct quantitative evidence for the architectural advantage of the third parallelism level.

Task 1: Mathematical model and serial baseline (Chapters 3 and 4). The polymer-flooding mathematical algorithm was formulated from first principles under seven physically motivated assumptions, with Brooks–Corey relative permeabilities [17], power-law polymer viscosity [9, 16], and the Langmuir adsorption isotherm [11]. The IMPES algorithm with adaptive CFL time-step selection, Picard fixed-point iteration, and nested polymer concentration Picard was implemented and validated against analytical references to relative error below  $10^{-6}$  (Section 4.9) and against mass-conservation tests to machine epsilon (Section 8.5). This baseline serves as the correctness reference and timing denominator for all seven speedup values.

Task 2: CPU-based parallel algorithms (Chapter 5). OpenMP (Section 5.1), MPI (Section 5.2), and hybrid MPI+OpenMP (Section 5.3) parallel algorithms were implemented and characterised across six grid sizes and four thread/rank counts. Peak speedups of  $9.05\times$  (OpenMP),  $5.55\times$  (MPI), and  $7.25\times$  (Hybrid) were achieved at small-to-intermediate grid sizes; all three saturate at roughly  $3\times$  at the largest grids due to the 51 GB/s DRAM bandwidth ceiling of the i9-11900KF, confirmed by VTune profiling at 94 % utilisation (Section 8.6.1).

Task 3: GPU-based parallel algorithms (Chapters 6 and 7). The CUDA global-memory algorithm achieved peak speedup  $20.17\times$  at  $N = 1\text{M}$  (Section 6.2). The CUDA shared-memory algorithm (Scientific Novelty 1) tiled all three coupled physics kernels into on-chip SRAM simultaneously (Section 6.3). The MPI+CUDA algorithm (Section 7.1) extended to multiple GPUs with host-mediated halo exchange, achieving  $32.81\times$  peak speedup. Block-size sensitivity was characterised across  $B \in \{128, 256, 512, 1024\}$ .

Task 4: Triple-hybrid algorithm and comprehensive benchmarking (Chapters 7 and 8). The triple-hybrid OMP + MPI + CUDA algorithm (Scientific Novelty 2) achieved peak speedup  $34.70\times$  at  $N = 1\text{M}$  (Section 7.3). Comprehensive benchmarking against the serial baseline across all seven algorithms and six grid sizes is reported in Chapter 8, including validation to below  $2 \times 10^{-8}$  relative error,  $O(N)$  scaling verification, physical consistency checks, and profiling-tool corroboration from VTune [44], mpiP [45, 52], Nsight Compute [50], and Nsight Systems [53]. Figures 7.1–7.3 provide the visual four-panel and cross-paradigm characterisations for the two novel algorithms.

#### Limitations of the Present Study

The present study is bounded by five concrete limitations, each of which is identified explicitly here and addressed by a corresponding future-work direction.

(1) Grid topology. One-dimensional structured Cartesian grid; extension to two-dimensional and three-dimensional unstructured grids representing realistic geological geometry are outside the present scope.

(2) Multi-GPU scale. Multi-GPU experiments were conducted on a single workstation by partitioning work across CUDA contexts; true multi-physical-GPU and multi-node benchmarking has not been performed.

(3) Geological heterogeneity. Uniform porosity and isotropic permeability are assumed; spatially varying properties would require additional data structures, load-balancing strategies, and harmonic-averaging stencils at facies boundaries.

(4) Thermal physics. Isothermal conditions are assumed; non-isothermal polymer flooding and thermally enhanced recovery processes are outside the present scope.

(5) Inter-GPU communication. Consumer-class RTX 2080 Ti GPUs without NVLink or GPUDirect force all inter-GPU communication through the host; data-centre-class hardware would eliminate this bottleneck.

#### Directions for Future Work

Five concrete directions for future work are proposed, each tied to a specific limitation and supported by literature precedents.

(1) Extension to 3D unstructured grids. The natural progression is to 3D unstructured grids using the MOAB mesh library and PETSc parallel solvers, with higher-order spatial discretisation (TVD flux limiters, second-order Godunov) as a complement.

(2) Multi-node deployment with NVLink and GPUDirect. Deploying the triple-hybrid algorithm on multi-node clusters with NVLink-connected GPUs and GPUDirect RDMA (available on NVIDIA DGX systems and exascale platforms) would test the scalability of the  $34.70\times$  peak speedup at production scale, with NCCL [54] replacing the host-mediated MPI\_Allreduce; related MPI/GPU runtime co-design issues have been demonstrated in scalable GPU-cluster workloads [55].

(3) Heterogeneous reservoir properties. Extending to spatially varying porosity, anisotropic permeability tensors, and facies-dependent capillary pressure curves would require treatment of harmonic-averaging stencils at property boundaries, dynamic load balancing, and additional storage for property fields.

(4) Non-isothermal polymer flooding and ASP processes. Coupling an energy-conservation PDE to the existing three-equation system would enable simulation of thermally enhanced recovery; extension to alkali-surfactant-polymer (ASP) flooding would add chemical-species transport, microemulsion phase behaviour, and reaction-kinetics modelling.

(5) Coupling with machine-learning surrogate algorithms. The parallel infrastructure developed here is well-suited to generating large simulation ensembles for surrogate algorithm training, enabling real-time simulation response for optimisation, history matching, and uncertainty quantification.

#### Publications Arising from this Dissertation

The research reported in this dissertation has been disseminated through ten scholarly outputs: four Scopus-indexed journal or conference publications, two international conference papers, and four publications in journals of the Committee for Quality Assurance in the Sphere of Education. Together these publications cover the four research tasks: the mathematical formulation and serial baseline; the CPU-based parallel algorithms; the GPU-based parallel algorithms; and the triple-hybrid algorithm with comprehensive benchmarking. The complete publication list appears again in the Conclusion; the outputs cited in the dissertation text are included in the consolidated Reference List.

#### Scopus:

1. Yerlan Makhmut, Timur Imankulov, Sergei Gorlatch, Bazargul Matkerim, A CUDA Performance Study of Global- and Shared-Memory Kernels for the Buckley–Leverett Polymer-Flooding Problem, Appl. Sci, May 2026. Q1 (79%)

2. Imankulov, T., Kenzhebek, Y., Bekele, S.D., Makhmut, E. Enhancing Oil Recovery Predictions by Leveraging Polymer Flooding Simulations and Machine Learning Models on a Large-Scale Synthetic Dataset. Energies Open source preview, 2024, 17(14), 3397. Q1 (90%) [56].

3. Daribayev, B., Mukhanbet, A., Makhmut, E., Azatbekuly, N., & Imankulov, T. (2025). Quantum–Classical Hybrid Architecture for Solving the Muskat–Leverett Model. IET Quantum Communication, 6(1), e70022. Q1 (79%) [57].

4. E. Makhmut, T. Imankulov, B. Daribayev PARALLEL IMPLEMENTATION OF MUSKAT-LEVERETT EQUATION USING CUDA // Vol. 119 No. 3 (2023): Journal of Mathematics, Mechanics and Computer Science. Q4 (18%) [58].

#### **International Conference:**

1. E. Makhmut, T. Imankulov, B. Daribayev, D. Akhmed-Zaki. Development of Hybrid Parallel Computing Models to Solve Polymer Flooding Problem, European Association of Geoscientists & Engineers // ECMOR 2022, Sep 2022, Volume 2022, p. 1 – 14. (Scopus) [51].

2. Imankulov, T., Kenzhebek, Y., Makhmut, E., Akhmed-Zaki, D. *Using machine learning algorithms to solve polymer flooding problem*, European Conference on the Mathematics of Geological Reservoirs 2022, ECMOR 2022. (Scopus)

#### **Committee for Quality Assurance in the Sphere of Education of the Ministry of Education of the Republic of Kazakhstan:**

1. Е. Махмут, Е.Ф. Кенжебек, Т.С. Иманкулов МРІ+OPENMP ГИБРИДТІ ПАРАЛЛЕЛДІ АЛГОРИТМІН ПАЙДАЛАНУ ҚҰИПТА МҰНАЙ ЫҒЫСТЫРУ МӘСЕЛЕСІН ШЕШУ // ВЕСТНИК КазНПУ им. Абая, серия «Физико-математические науки», No1(81), 2023г. p. 56–65 [59].

2. Е. Махмут\*, Т.С. Иманкулов, Б. Мәткерім МҰНАЙ ЫҒЫСТЫРУ МӘСЕЛЕСІН ШЕШУДЕ ГИБРИДТІ (МРІ+CUDA) ПАРАЛЛЕЛДІ ПРОГРАММАСЫН ЖОБАЛАУ ЖӘНЕ ҚҰРУ // Қазақстан Республикасы Ұлттық инженерияның Хабаршысы, 2024. № 1 (91) [60].

3. Erlan Makhmut, Timur Imankulov COMPARATIVE ANALYSIS OF REDUCTION ALGORITHMS IN CUDA FOR PRESSURE EQUATION // Vol. 1 No. 2 (2023): Journal of Problems in Computer Science and Information Technologies [61].

4. Мустафин М. Б.; Махмут Е. СРАВНИТЕЛЬНЫЙ АНАЛИЗ ЭФФЕКТИВНОСТИ ИСПОЛЬЗОВАНИЯ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ В МАШИННОМ ОБУЧЕНИИ, Вестник Национальной инженерной академии Республики Казахстан, 2024-12, Journal article

The dissertation closes with a standalone Conclusion section summarizing the principal findings and their broader implications.

## REFERENCE LIST

- [1] Energy Institute. Statistical Review of World Energy 2024, 73rd ed. London: Energy Institute, 2024. ISBN: 978-1-78725-408-4. URL: <https://www.energyinst.org/statistical-review>
- [2] International Energy Agency. World Energy Outlook 2024. Paris: IEA, 2024. URL: <https://www.iea.org/reports/world-energy-outlook-2024>
- [3] Lake, L. W.; Johns, R. T.; Rossen, W. R.; Pope, G. A. Fundamentals of Enhanced Oil Recovery. Richardson, TX: Society of Petroleum Engineers, 2014. DOI: 10.2118/9781613993286. ISBN: 978-1-61399-328-6.
- [4] Sheng, J. J. Modern Chemical Enhanced Oil Recovery: Theory and Practice. Burlington, MA: Gulf Professional Publishing/Elsevier, 2011. DOI: 10.1016/C2009-0-20241-8. ISBN: 978-1-85617-745-0.
- [5] Wang, D.; Cheng, J.; Yang, Q.; Gong, W.; Li, Q.; Chen, F. "Viscous-Elastic Polymer Can Increase Microscale Displacement Efficiency in Cores." Paper SPE-63227-MS, SPE Annual Technical Conference and Exhibition, Dallas, TX, USA, 1-4 October 2000. Richardson, TX: Society of Petroleum Engineers, 2000. DOI: 10.2118/63227-MS.
- [6] Aziz, K.; Settari, A. Petroleum Reservoir Simulation. London: Applied Science Publishers, 1979. ISBN: 0-85334-787-5.
- [7] Pye, D. J. "Improved Secondary Recovery by Control of Water Mobility." Journal of Petroleum Technology, 16(8), 911-916, 1964. DOI: 10.2118/845-PA.
- [8] Sandiford, B. B. "Laboratory and Field Studies of Water Floods Using Polymer Solutions to Increase Oil Recoveries." Journal of Petroleum Technology, 16(8), 917-922, 1964. DOI: 10.2118/844-PA.
- [9] Sorbie, K. S. Polymer-Improved Oil Recovery. Glasgow: Blackie; Boca Raton, FL: CRC Press, 1991. DOI: 10.1007/978-94-011-3044-8. ISBN: 978-0-216-92693-6.
- [10] Wassmuth, F. R.; Arnold, W.; Green, K.; Cameron, N. "Polymer Flood Application to Improve Heavy Oil Recovery at East Bodo." Journal of Canadian Petroleum Technology, 48(2), 55-61, 2009. DOI: 10.2118/09-02-55.
- [11] Langmuir, I. "The Adsorption of Gases on Plane Surfaces of Glass, Mica and Platinum." Journal of the American Chemical Society, 40(9), 1361-1403, 1918. DOI: 10.1021/ja02242a004.
- [12] Darcy, H. Les fontaines publiques de la ville de Dijon: Exposition et application des principes à suivre et des formules à employer dans les questions de distribution d'eau. Paris: V. Dalmont, 1856.
- [13] Leverett, M. C. "Capillary Behavior in Porous Solids." Transactions of the AIME, 142(1), 152-169, 1941. DOI: 10.2118/941152-G.
- [14] Buckley, S. E.; Leverett, M. C. "Mechanism of Fluid Displacement in Sands." Transactions of the AIME, 146(1), 107-116, 1942. DOI: 10.2118/942107-G.

- [15] Peaceman, D. W. Fundamentals of Numerical Reservoir Simulation. Developments in Petroleum Science, vol. 6. Amsterdam: Elsevier Scientific Publishing Company, 1977. DOI: 10.1016/S0376-7361(08)X7013-X. ISBN: 978-0-444-41578-3.
- [16] Todd, M. R.; Longstaff, W. J. "The Development, Testing, and Application of a Numerical Simulator for Predicting Miscible Flood Performance." Journal of Petroleum Technology, 24(7), 874-882, 1972. DOI: 10.2118/3484-PA.
- [17] Brooks, R. H.; Corey, A. T. Hydraulic Properties of Porous Media. Hydrology Papers, no. 3. Fort Collins, CO: Colorado State University, 1964.
- [18] Douglas, J., Jr.; Peaceman, D. W.; Rachford, H. H., Jr. "A Method for Calculating Multi-Dimensional Immiscible Displacement." Transactions of the AIME, 216(1), 297-308, 1959. DOI: 10.2118/1327-G.
- [19] Sheldon, J. W.; Zondek, B.; Cardwell, W. T., Jr. "One-Dimensional, Incompressible, Noncapillary, Two-Phase Fluid Flow in a Porous Medium." Transactions of the AIME, 216(1), 290-296, 1959. DOI: 10.2118/978-G.
- [20] Stone, H. L.; Garder, A. O., Jr. "Analysis of Gas-Cap or Dissolved-Gas Drive Reservoirs." Society of Petroleum Engineers Journal, 1(2), 92-104, 1961. DOI: 10.2118/1518-G.
- [21] Saad, Y. Iterative Methods for Sparse Linear Systems, 2nd ed. Philadelphia, PA: SIAM, 2003. DOI: 10.1137/1.9780898718003.
- [22] Courant, R.; Friedrichs, K.; Lewy, H. "Über die partiellen Differenzengleichungen der mathematischen Physik." Mathematische Annalen, 100(1), 32-74, 1928. DOI: 10.1007/BF01448839.
- [23] Godunov, S. K. "Raznostnyi metod chislennogo rascheta razryvnykh reshenii uravnenii gidrodinamiki" [A Difference Method for Numerical Calculation of Discontinuous Solutions of the Equations of Hydrodynamics]. Matematicheskii Sbornik, 47(89)(3), 271-306, 1959.
- [24] LeVeque, R. J. Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems. Philadelphia, PA: SIAM, 2007. DOI: 10.1137/1.9780898717839.
- [25] Picard, E. "Mémoire sur la théorie des équations aux dérivées partielles et la méthode des approximations successives." Journal de Mathématiques Pures et Appliquées, 4th series, 6, 145-210, 1890.
- [26] Dagum, L.; Menon, R. "OpenMP: An Industry-Standard API for Shared-Memory Programming." IEEE Computational Science & Engineering, 5(1), 46-55, 1998. DOI: 10.1109/99.660313.
- [27] Gropp, W.; Lusk, E.; Skjellum, A. Using MPI: Portable Parallel Programming with the Message-Passing Interface, 3rd ed. Cambridge, MA: MIT Press, 2014. ISBN: 978-0-262-52739-2.

- [28] NVIDIA Corporation. CUDA C++ Programming Guide, version 12.4. NVIDIA, 2024. URL: <https://docs.nvidia.com/cuda/archive/12.4.0/cuda-c-programming-guide/>
- [29] Williams, S.; Waterman, A.; Patterson, D. "Roofline: An Insightful Visual Performance Model for Multicore Architectures." *Communications of the ACM*, 52(4), 65-76, 2009. DOI: 10.1145/1498765.1498785.
- [30] NVIDIA Corporation. GPUDirect RDMA, release 12.4: Developing a Linux Kernel Module Using GPUDirect RDMA. NVIDIA CUDA Toolkit Documentation, 2024. URL: <https://docs.nvidia.com/cuda/archive/12.4.1/gpudirect-rdma/>
- [31] Foley, D.; Danskin, J. "Ultra-Performance Pascal GPU and NVLink Interconnect." *IEEE Micro*, 37(2), 7-17, 2017. DOI: 10.1109/MM.2017.37.
- [32] Imankulov, T.; Daribayev, B.; Mukhambetzhannov, S. "Comparative Analysis of Parallel Algorithms for Solving Oil Recovery Problem Using CUDA and OpenCL." *International Journal of Nonlinear Analysis and Applications*, 12(1), 351-364, 2021. DOI: 10.22075/ijnaa.2021.4809.
- [33] Foadaddini, A.; Zolfaghari, S. A.; Mahmoodi Darian, H.; Saadatfar, H. "A New GPU-Based Corrected Explicit-Implicit Domain Decomposition Scheme for Convection-Dominated Diffusion Problems." *Computers & Mathematics with Applications*, 123, 184-203, 2022. DOI: 10.1016/j.camwa.2022.08.015.
- [34] Zaza, A.; Awotunde, A. A.; Fairag, F. A.; Al-Mouhamed, M. A. "A CUDA Based Parallel Multi-Phase Oil Reservoir Simulator." *Computer Physics Communications*, 206, 2-16, 2016. DOI: 10.1016/j.cpc.2016.04.010.
- [35] de la Cruz, L. M.; Monsivais, D. "Parallel Numerical Simulation of Two-Phase Flow Model in Porous Media Using Distributed and Shared Memory Architectures." *Geofísica Internacional*, 53(1), 59-75, 2014. DOI: 10.1016/S0016-7169(14)71490-4.
- [36] Teja-Juárez, V. L.; de la Cruz, L. M. "A Graphic Processing Unit (GPU) Based Implementation of an Incompressible Two-Phase Flow Model in Porous Media." *Geofísica Internacional*, 57(3), 205-222, 2018. DOI: 10.22201/igeof.00167169p.2018.57.3.2112.
- [37] Beisembetov, I. K.; Bekibaev, T. T.; Assilbekov, B. K.; Zhabbasbayev, U. K.; Kenzhaliev, B. K. "High-Performance Computing in Oil Recovery Simulation Based on CUDA." *Blucher Mechanical Engineering Proceedings*, 1(1), 4895-4911, 2014. DOI: 10.5151/meceng-wccm2012-20142.
- [38] Tene, M.; Sekachev, M.; de Brito Dias, D.; Szyndel, M. D. E. "Graphics Processing Unit Performance Scalability Study on a Commercial Black-Oil Reservoir Simulator." Paper SPE-212183-MS, SPE Reservoir Simulation Conference, Galveston, TX, USA, 28-30 March 2023. Society of Petroleum Engineers, 2023. DOI: 10.2118/212183-MS.

- [39] Forchheimer, P. "Wasserbewegung durch Boden." *Zeitschrift des Vereines Deutscher Ingenieure*, 45(50), 1781-1788, 1901.
- [40] Hennessy, J. L.; Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 6th ed. Cambridge, MA: Morgan Kaufmann/Elsevier, 2017. ISBN: 978-0-12-811905-1.
- [41] Chapman, B.; Jost, G.; van der Pas, R. *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, MA: MIT Press, 2007. ISBN: 978-0-262-53302-7.
- [42] Pacheco, P. S.; Malensek, M. *An Introduction to Parallel Programming*, 2nd ed. Cambridge, MA: Morgan Kaufmann/Elsevier, 2021. DOI: 10.1016/C2015-0-01650-1. ISBN: 978-0-12-804605-0.
- [43] Rabenseifner, R.; Hager, G.; Jost, G. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes." In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2009)*, Weimar, Germany, 18-20 February 2009, pp. 427-436. IEEE, 2009. DOI: 10.1109/PDP.2009.43.
- [44] Intel Corporation. *Intel VTune Profiler User Guide*, version 2024.2, document ID 766319. Intel, 2024. URL: <https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2024-2/overview.html>
- [45] Lawrence Livermore National Laboratory. *mpiP 3.5: A Light-Weight MPI Profiler*, version 3.5.0. Lawrence Livermore National Laboratory, 2020. URL: <https://software.llnl.gov/mpiP/>
- [46] NVIDIA Corporation. *NVIDIA Turing GPU Architecture: Graphics Reinvented*. White paper WP-09183-001\_v01. NVIDIA, 2018. URL: <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>
- [47] Owens, J. D.; Houston, M.; Luebke, D.; Green, S.; Stone, J. E.; Phillips, J. C. "GPU Computing." *Proceedings of the IEEE*, 96(5), 879-899, 2008. DOI: 10.1109/JPROC.2008.917757.
- [48] Harris, M. "Optimizing Parallel Reduction in CUDA." *NVIDIA Developer Technology*, 2007. URL: <https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>
- [49] Volkov, V. "Better Performance at Lower Occupancy." *Presentation, NVIDIA GPU Technology Conference (GTC 2010)*, 22 September 2010. URL: [https://www.nvidia.com/content/GTC-2010/pdfs/2238\\_GTC2010.pdf](https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf)
- [50] NVIDIA Corporation. *Nsight Compute Documentation*, version 2024.1.1. NVIDIA, 2024. URL: <https://docs.nvidia.com/nsight-compute/2024.1/>

- [51] Makhmut, E.; Imankulov, T.; Daribayev, B.; Akhmed-Zaki, D. "Development of Hybrid Parallel Computing Models to Solve Polymer Flooding Problem." ECMOR 2022, The Hague, Netherlands and Online, September 2022, pp. 1-14. European Association of Geoscientists & Engineers, 2022. DOI: 10.3997/2214-4609.202244038.
- [52] Vetter, J. S.; McCracken, M. O. "Statistical Scalability Analysis of Communication Operations in Distributed Applications." In Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '01), Snowbird, UT, USA, 18-20 June 2001, pp. 123-132. New York: ACM, 2001. DOI: 10.1145/379539.379590.
- [53] NVIDIA Corporation. NVIDIA Nsight Systems User Guide, version 2024.1. NVIDIA, 2024. URL: <https://docs.nvidia.com/nsight-systems/2024.1/UserGuide/>
- [54] NVIDIA Corporation. NVIDIA Collective Communications Library (NCCL) Documentation, release 2.23.4. NVIDIA, 2024. URL: [https://docs.nvidia.com/deeplearning/nccl/archives/nccl\\_2234/](https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2234/)
- [55] Awan, A. A.; Hamidouche, K.; Hashmi, J. M.; Panda, D. K. "S-Caffe: Co-Designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters." In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17), Austin, TX, USA, 4-8 February 2017, pp. 193-205. New York: ACM, 2017. DOI: 10.1145/3018743.3018769.
- [56] Imankulov, T.; Kenzhebek, Y.; Bekele, S. D.; Makhmut, E. "Enhancing Oil Recovery Predictions by Leveraging Polymer Flooding Simulations and Machine Learning Models on a Large-Scale Synthetic Dataset." *Energies*, 17(14), 3397, 2024. DOI: 10.3390/en17143397.
- [57] Daribayev, B.; Mukhanbet, A.; Makhmut, E.; Azatbekuly, N.; Imankulov, T. "Quantum-Classical Hybrid Architecture for Solving the Muskat-Leverett Model." *IET Quantum Communication*, 6(1), e70022, 2025. DOI: 10.1049/qtc2.70022.
- [58] Makhmut, E.; Imankulov, T.; Daribayev, B. "Parallel Implementation of the Muskat-Leverett Equation Using CUDA." *Journal of Mathematics, Mechanics and Computer Science*, 119(3), 104-116, 2023. DOI: 10.26577/JMMCS2023v119i3a9.
- [59] Makhmut, E.; Kenzhebek, Y.; Imankulov, T. "Hybrid Parallel Algorithm Using MPI and OpenMP to Solve Oil Displacement Problem." *Bulletin of Abai KazNPU, Series of Physical and Mathematical Sciences*, 81(1), 56-65, 2023. DOI: 10.51889/2959-5894.2023.81.1.006.
- [60] Makhmut, E.; Imankulov, T. S.; Matkerim, B. "Design and Development of a Hybrid (MPI+CUDA) Parallel Program for Solving the Oil Displacement

Problem." Bulletin of the National Engineering Academy of the Republic of Kazakhstan, 91(1), 72-82, 2024. DOI: 10.47533/2024.1606-146X.08.

- [61] Makhmut, E.; Imankulov, T. S. "Comparative Analysis of Reduction Algorithms in CUDA for Pressure Equation." Journal of Problems in Computer Science and Information Technologies, 1(2), 50-57, 2023. DOI: 10.26577/JPCSIT.2023.v1.i2.05.