

E. Makhmut¹ , B. Sansyzbay^{1*} , K. Aidarov² 

¹Al-Farabi Kazakh National University, Almaty, Kazakhstan

²Beeline Kazakhstan (LLP QazCode), Almaty, Kazakhstan

*e-mail: beiimbet.sansyzbay@gmail.com

GPU-ACCELERATED SOLUTION OF THE 2D POISSON EQUATION: A COMPARATIVE STUDY OF JACOBI AND RED-BLACK GAUSS-SEIDEL METHODS WITH GLOBAL AND SHARED MEMORY

Abstract. The Poisson equation is a canonical elliptic partial differential equation (PDE) that arises in diverse fields such as electrostatics, fluid dynamics, and heat transfer. Analytical solutions exist only for simple geometries and boundary conditions, making numerical approaches indispensable for realistic problems. This study presents a comparative analysis of four GPU-accelerated CUDA solvers for the two-dimensional Poisson equation: the Jacobi method with global memory (Jacobi-Global), the Jacobi method with shared memory tiling (Jacobi-Shared), the Red-Black Gauss-Seidel method with global memory (RBGS-Global), and RBGS with shared memory tiling (RBGS-Shared). A finite difference discretization reduces the PDE to a large sparse system of linear equations, which is then solved iteratively. All GPU implementations are compared against a single-threaded CPU baseline across grid sizes from 64×64 to 1024×1024 , with RBGS additionally evaluated up to 4096×4096 . Results show that GPU acceleration is ineffective for small grids due to kernel launch latency and memory transfer overheads, but becomes highly advantageous for larger problems. Jacobi-Shared achieves a peak speedup of $77.32 \times$ at 1024×1024 , while RBGS-Shared converges in orders of magnitude fewer iterations and reaches $47.24 \times$ speedup at the same grid size. Shared memory consistently outperforms global memory, with execution time reductions of up to $2.4 \times$ for narrow block configurations at large grid sizes. These findings provide practical guidance for solver and memory strategy selection in GPU-based PDE computations.

Keywords: SOR, CUDA, RBGS, Jacobi.

1. Introduction

The Poisson equation is a canonical elliptic partial differential equation (PDE) that arises in diverse fields such as electrostatics, fluid dynamics, and heat transfer [1]. Analytical solutions exist only for simple geometries and boundary conditions, making numerical approaches indispensable for realistic problems. The finite difference method (FDM) discretizes the domain into a structured grid and yields large, sparse linear systems amenable to iterative solution [2]. The growing demand for solving such systems at scale has motivated the adoption of graphics processing units (GPUs). CUDA-NVIDIA's programming model for general-purpose GPU computing -exposes massive fine-grained parallelism and a hierarchical memory model that accelerates stencil computations characteristic of PDE solvers [3].

Despite extensive prior work on GPU-based iterative solvers, no study has provided a systematic like-for-like comparison of Jacobi and Red-Black Gauss-Seidel (RBGS) under both global and shared memory strategies across a wide range of grid sizes.

In particular, the impact of narrow asymmetric thread blocks on shared memory efficiency for 2D Poisson solvers has not been systematically characterized. This paper addresses both gaps through controlled empirical benchmarking on NVIDIA Blackwell hardware.

The scientific novelty of this paper lies in the following original empirical contributions. First, we provide the first systematic, like-for-like comparison of all four combinations – Jacobi-Global, Jacobi-Shared, RBGS-Global, RBGS-Shared – under identical experimental conditions. Second, we quantify for the first time the effect of narrow asymmetric thread blocks (16×8 , 128 threads) on shared memory efficiency for the 2D five-point Poisson stencil, demonstrating that Jacobi-Shared with 16×8 blocks reduces GPU execution time by $2.24 \times$ relative to global memory at 1024×1024 , and RBGS-Shared with 16×8 blocks achieves $2.41 \times$ reduction at 4096×4096 . Third, we provide new empirical data on the crossover behavior between GPU and CPU performance regimes for both Jacobi and RBGS across five grid sizes. These results yield concrete, actionable

guidelines for GPU Poisson solver design: (i) shared memory tiling is critical for narrow blocks; (ii) RBGS should be preferred over Jacobi for any production computation; and (iii) the 32×32 block is optimal only for global memory variants.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the problem formulation and methodology. Section 4 describes the GPU and CPU implementations. Section 5 reports and discusses experimental results. Section 6 concludes.

2. Related work

2.1 Iterative Solvers on GPU

Foundational analyses of iterative methods for sparse linear systems are provided by Saad [4] and Demmel [5], which form the theoretical backbone of this study. Early GPU work by Goodnight et al. [6] established the viability of multigrid solvers on programmable graphics hardware; Hockney and Eastwood [7] provided efficient iterative approaches on structured grids that continue to motivate GPU solver designs; Bell and Garland [8] quantified the bandwidth-bound behavior inherent in stencil solvers. NVIDIA developer resources provide canonical Jacobi implementations for multi-GPU settings [9], [10].

The Jacobi method updates all grid points simultaneously from the previous iterate, making all updates fully independent and trivially parallel. Despite its modest convergence rate, Jacobi remains widely used on GPUs due to this natural data parallelism. Comparative GPU studies by Aslam et al. [11], Chiu et al. [12], and Li et al. [13] further clarify the tradeoffs between solver variants in different application contexts. The Red-Black Gauss–Seidel (RBGS) method partitions grid points into two complementary subsets based on a checkerboard coloring, allowing each subset to be updated in parallel while incorporating the most recently computed values from the other subset. RBGS achieves approximately twice the per-iteration convergence of Jacobi [17] and has been shown to deliver speedups of up to $484\times$ over sequential CPU on GPU hardware [14]. Kochurov and Golovashkin [15] extended RBGS to datasets exceeding GPU memory capacity using a pyramid-method approach. Bozorgmehr et al. [16] employed GPU red-black SOR with dynamic parallelism for a 3D wind-field Poisson solver in atmospheric modeling. GPU-based methods have also been

applied to broader fluid dynamics problems, including incompressible flow solvers [18] and coupled Boltzmann–Poisson systems for nanoscale device simulation [19].

2.2. Memory Strategies for Stencil Computation

GPU global memory offers large capacity but high latency, while shared memory is a low-latency on-chip scratchpad. Tiling strategies exploit shared memory to reduce redundant global memory traffic for stencil operations. Recent work on stencil optimization [20], [21] confirms that block shape and memory strategy interact in non-trivial ways that depend on grid size and solver type. Micikevicius [22] demonstrated that shared memory tiling for finite difference stencils can approach the memory bandwidth roofline on GPU; however, the specific impact of narrow asymmetric thread blocks (such as 16×8) on shared memory efficiency for 2D Poisson solvers has not been systematically characterized. The present study directly addresses this gap by benchmarking all four solver-memory combinations across five grid sizes.

3. Problem Formulation and Methodology

We consider the two-dimensional Poisson equation on the unit square domain:

$$\begin{aligned} \nabla^2 u(x, y) &= \\ &= f(x, y), (x, y) \in \Omega = (0, 1) \times (0, 1) \end{aligned} \quad (1)$$

subject to Dirichlet boundary conditions

$$u(x, y) = g(x, y), (x, y) \in \partial\Omega \quad (2)$$

Here $u(x, y)$ is the unknown scalar potential, $f(x, y)$ is the source term, and $g(x, y)$ prescribes boundary values. The Laplacian operator is

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad (3)$$

which measures the local imbalance between the field value at a point and its neighborhood average.

For the numerical experiments in this study, the source term is chosen as $f(x, y) = -2[x(1 - x) + y(1 - y)]$, with homogeneous Dirichlet boundary conditions ($g(x, y) = 0$ on $\partial\Omega$). This choice admits the closed-form analytical solution $u(x, y) = x(1 - x)y(1 - y)$, which is used as the

reference solution for all error computations reported in this study.

3.1. Finite Difference Discretization

The domain Ω is discretized into a uniform Cartesian grid with $N \times N$ points and spacing

$h = \frac{1}{(N-1)}$. Second-order central differences give:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \quad (4)$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

Substituting into the Poisson equation gives the five-point stencil:

$$u_{i,j} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - h^2 f_{i,j}), \quad 1 \leq N-2 \quad (5)$$

This yields a sparse symmetric positive definite linear system $Au = f$, where $A \in \mathbb{R}^{(N-2)^2 \times (N-2)^2}$ has banded structure arising from the five-point connectivity.

3.2 Jacobi Iterative Method

The Jacobi method decomposes $A = D + (L + R)$, where D is diagonal and L, R are strictly lower and upper triangular. The update rule is:

$$u_{i,j}^{(k+1)} = \frac{1}{4}(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - h^2 f_{i,j}) \quad (6)$$

All N^2 updates at step $k+1$ depend exclusively on values from step k , enabling fully parallel execution. Convergence is checked by:

$$\max_{i,j} |u_{i,j}^{(k+1)} - u_{i,j}^{(k)}| < \varepsilon \quad (7)$$

The spectral radius satisfies $\rho_J = \cos(\pi h) \approx 1 - \frac{\pi^2 h^2}{2}$, so iteration count grows as $O(N^2)$ for fine grids [4].

3.3. Red-Black Gauss-Seidel with Successive Over-Relaxation (RBGS-SOR)

RBGS assigns each point (i,j) a color: red if $(i+j)$ is even, black if $(i+j)$ is odd. Since every neighbor of a red point is black and vice versa, each subset can be updated in parallel. To further accelerate convergence, we apply Successive Over-Relaxation (SOR) with the optimal relaxation parameter ω . For the Poisson equation on an $N \times N$ uniform grid, the optimal value is:

$$\omega_{\text{opt}} = \frac{2}{1 + \sin\left(\frac{\pi}{N}\right)} \quad (8)$$

The first half-sweep (red points, $(i+j)$ even) computes the Gauss-Seidel update

$$u_{i,j}^{(k+1)\text{GS}} = \frac{1}{4}(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - h^2 f_{i,j}) \quad (9)$$

and then applies the SOR correction:

$$u_{i,j}^{(k+1/2)} = (1 - \omega) u_{i,j}^{(k)} + \omega \cdot u_{i,j}^{\text{GS}} \quad (10)$$

where, $(i+j)$ even.

The second half-sweep (black points, $(i+j)$ odd) repeats with the updated red values:

$$u_{i,j}^{(k+1)} = (1 - \omega) u_{i,j}^{(k+1/2)} + \omega \cdot \frac{1}{4}(u_{i+1,j}^{(k+1/2)} + u_{i-1,j}^{(k+1/2)} + u_{i,j+1}^{(k+1/2)} + u_{i,j-1}^{(k+1/2)} - h^2 f_{i,j}) \quad (11)$$

where, $(i+j)$ odd.

With ω_{opt} , the spectral radius of RBGS-SOR satisfies

$$\rho_{\text{SOR}} = \omega_{\text{opt}} - 1, \quad (12)$$

which is dramatically smaller than $(\rho_{\text{RBGS}} = \rho_J^2)$ of plain RBGS. For a 1024×1024 grid, $\omega_{\text{opt}} \approx 1.994$,

giving $\rho_{\text{SOR}} \approx 0.994$, compared to $\rho_J \approx 0.999995$ for Jacobi. This significant reduction in the spectral radius is the main reason why RBGS-SOR requires only about 3,060 iterations, while the plain Jacobi method needs approximately 2,011,200 iterations – a speedup of roughly 657 times.

3.4. Parallelization Perspective

Both methods map naturally to GPU execution. For Jacobi, all N^2 interior points are updated simultaneously in a single kernel launch per iteration. For RBGS, two kernel launches per iteration update red and black subsets respectively. In both cases, the computational pattern is a five-point stencil: each thread reads five values and writes one. Performance depends critically on memory access patterns and on-chip data reuse.

4. Implementation

4.1 CPU Baseline Implementation

The CPU baseline is engineered as a rigorous serial reference to establish a primary benchmark for assessing GPU acceleration. The implementation of both the Jacobi and Red-Black Gauss–Seidel (RBGS) methods utilizes single-threaded C++ algorithms to ensure a fair comparison against sequential execution. For the Jacobi method, the architecture maintains two distinct arrays representing the current iterate u^k and the subsequent iterate u^{k+1} which are swapped at the conclusion of each iteration to prevent data races and ensure mathematical consistency. In contrast, the RBGS method is implemented using a single in-place array because the checkerboard update pattern ensures that red points depend exclusively on black points from the previous half-step, allowing for a more memory-efficient update across two distinct passes per iteration. For both methods all interior grid points are initialized to zero, while boundary values are strictly defined by the prescribed Dirichlet conditions. Convergence is monitored by calculating the maximum absolute difference between u^k and u^{k+1} successive iterations across the entire grid until the specified tolerance ϵ reached.

4.2 GPU Kernel Implementation

Each GPU kernel maps the two-dimensional grid directly onto the CUDA thread hierarchy. A

thread block of dimensions $B_x \times B_y$ is launched over a grid of $\text{ceil}(N / B_x) \times \text{ceil}(N / B_y)$ blocks, so that every interior point (i, j) is covered. Within the kernel, each thread computes its global grid coordinates as $i = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$ and $j = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$, and a boundary guard ensures that only interior points ($1 \leq i, j \leq N-2$) perform the stencil update. Four block configurations are evaluated: 32×32 (1024 threads), 16×32 (512 threads), 16×16 (256 threads), and 16×8 (128 threads). The Jacobi kernel requires one launch per iteration; the RBGS kernel requires two launches per iteration, one for each color.

In the global memory variant, each thread reads its five stencil values directly from device DRAM using the computed (i, j) indices. Accesses along the j -axis (horizontal neighbors) are contiguous in memory and are coalesced into a single 128-byte transaction per warp by the GPU memory controller. Accesses along the i -axis (vertical neighbors $u[i+1][j]$ and $u[i-1][j]$) are separated by a full row stride of N elements, producing stride- N DRAM transactions that cannot be coalesced, increasing effective memory latency for large grids. The updated value is written back to a second array (Jacobi) or in-place (RBGS), and the per-thread residual contribution is accumulated into a device-global variable via *atomicMax()*.

In the shared memory variant, threads in each block cooperatively preload a $(B_y + 2) \times (B_x + 2)$ tile into on-chip shared memory before computing. The tile includes a one-cell halo on each side to cover all five stencil neighbors of every interior thread. Each thread loads its own interior element; boundary threads additionally load the adjacent halo cells by offsetting their indices. A *__syncthreads()* barrier is inserted after the load phase to ensure all tile elements are visible to every thread before any stencil computation begins. Subsequent neighbor reads are served from shared memory at approximately 100x lower latency than DRAM. For a 16×16 block, each of the 256 interior values requires 5 global reads in the global variant, totaling 1280 transactions; with tiling, the (18×18) tile requires only 324 global loads, reducing average DRAM reads per point from 5 to 1.27, a 3.9x reduction. For narrow 16×8 blocks the halo fraction is larger relative to the interior, yielding a measured 2.24x execution-time reduction at 1024×1024 .

Algorithm 1: Jacobi Iterative Method on GPU

In: $u[N][N]$ – *current and next iterate arrays*; $f[N][N]$ – *source term*; h – *grid spacing*; e – *tolerance*
 Out: $u[N][N]$ – *approximate solution*; $iter$ – *iteration count*

- 1: repeat
- 2: parallel for all interior (i, j) do
- 3: $u_new[i][j] = (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - h*h*f[i][j]) / 4$
- 4: end parallel
- 5: $residual = \max(i, j) |u_new[i][j] - u[i][j]|$
- 6: swap u_old and u_new
- 7: $iter = iter + 1$
- 8: until $residual < e$

Algorithm 1 presents the Jacobi kernel logic. The *parallel for* construct (line 2) corresponds to a single CUDA kernel launch over the entire interior of the $N \times N$ grid. The two-dimensional launch grid is configured as $\text{ceil}(\frac{N}{B_x}) \times \text{ceil}(\frac{N}{B_y})$ thread blocks, each of size $B_x \times B_y$, so that exactly one thread is assigned to each grid point. A boundary guard at the start of the kernel ensures that threads mapping to halo or out-of-bounds indices exit without performing any computation. Because every update at iteration $k+1$ depends only on values from iteration k , all N^2 threads operate fully independently with no inter-thread communication or ordering constraints. This property makes the Jacobi method embarrassingly parallel: the entire stencil computation is free of data races regardless of thread scheduling order. Two device arrays are maintained throughout the solve: one holds the current iterate u^k

and the other receives the updated values u^{k+1} . At the end of each iteration the device pointers are swapped on the host (line 6), so the roles of the two arrays alternate each iteration without any device-to-device memory copy. This double-buffering strategy eliminates write-after-read hazards that would arise if all threads updated a single shared array in-place. The convergence check (line 5) is implemented as a device-wide reduction. Before the kernel launch, a scalar device variable $d_residual$ is reset to zero on the host. Inside the kernel, each thread computes its local absolute difference $|u_new[i][j] - u_old[i][j]|$ and atomically updates $d_residual$ via *atomicMax()*, which guarantees a race-free maximum reduction across all threads. After the kernel returns, the host copies $d_residual$ back to the CPU with a single *cudaMemcpy()* call and compares it against the tolerance ϵ to decide whether to terminate or launch the next iteration.

Algorithm 2: Red-Black Gauss-Seidel with SOR on GPU

In: $u[N][N]$ – *in-place solution array*; $f[N][N]$, h , N ; e – *tolerance*; w – *SOR relaxation factor*
 Out: $u[N][N]$ – *approximate solution*; $iter$ – *iteration count*

- 1: Compute $w = 2 / (1 + \sin(\pi / N))$
- 2: repeat
- 3: Red half-sweep: parallel for all (i, j) where $(i+j) \bmod 2 = 0$ do
- 4: $u_gs = (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - h*h*f[i][j]) / 4$
- 5: $u[i][j] = (1 - w) * u[i][j] + w * u_gs$
- 6: Black half-sweep: parallel for all (i, j) where $(i+j) \bmod 2 = 1$ do
- 7: $u_gs = (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - h*h*f[i][j]) / 4$
- 8: $u[i][j] = (1 - w) * u[i][j] + w * u_gs$
- 9: $residual = \max(i, j) |u[i][j] - u_prev[i][j]|$
- 10: $iter = iter + 1$
- 11: until $residual < e$

Algorithm 2 presents the RBGS-SOR kernel logic. The checkerboard partition is enforced at the thread level: each thread evaluates the condition $(i + j) \bmod 2 == \text{parity}$ and performs the SOR update only if matched, while all other threads exit immediately. This parity check adds negligible overhead since inactive threads follow the same

warp path and are masked by the hardware. Two separate kernel launches per iteration (parity = 0 for red, parity = 1 for black) guarantee that the black half-sweep reads the freshly updated red values, preserving the Gauss-Seidel dependency structure. Because RBGS operates in-place on a single array, no pointer swap is needed. The SOR factor w (line

1) is computed once on the host before the iteration loop and passed as a kernel argument; at 1024×1024 it evaluates to $w = 1.994$, reducing the spectral radius from $\rho = 0.999995$ (Jacobi) to $\rho = 0.994$ (RBGS-SOR) and cutting the required iteration count from 2,011,200 to 3,060, a 657x reduction.

5. Results and discussion

All experiment results were conducted on a high-performance mobile workstation specifically designed for intensive computational research. The system is equipped with an Intel Core Ultra 7 255HX CPU, which features 20 cores-comprising 8 Performance-cores and 12 Efficient-cores-supporting 20 threads and reaching a maximum boost clock of 5.2 GHz. The primary computational accelerator is an NVIDIA GeForce RTX 5070 Mobile GPU based on the Blackwell architecture, featuring 4608 CUDA cores and 8 GB of GDDR7 VRAM. The host system includes 32 GB of DDR5-5600 RAM to support the memory requirements of large-scale grid processing. The software environment consists of the 64-bit Windows 11 operating system and the CUDA Toolkit 12.6. GPU execution timing is performed with sub-microsecond resolution using *cudaEvent_t* markers to ensure high-fidelity data collection. To isolate the performance of the iterative solvers, all reported execution times strictly cover the solver loop and exclude host-to-device or device-to-host memory transfer overheads.

This study evaluates the interaction between thread-level parallelism and grid dimensions to identify optimal occupancy and throughput. Four distinct 2D block shapes are tested, including 32×32 with 1024 threads, 16×32 with 512 threads, 16×16 with 256 threads, and 16×8 with 128

threads. Grid sizes scale from 64×64 to 1024×1024 for Jacobi with a tolerance of $\varepsilon = 10^{-4}$, while RBGS benchmarks are extended up to 4096×4096 with an identical tolerance of $\varepsilon = 10^{-4}$. Performance is recorded using *cudaEvent_t* with sub-microsecond resolution to provide highly accurate timing data. To isolate the efficiency of the solver kernels, the reported execution times strictly cover the iterative loop and exclude the overhead associated with initial host-to-device and final device-to-host memory transfers. All reported results utilize the specific block configuration that achieved the highest speedup for each respective grid size.

5.1 Error validation

Both GPU implementations produce solutions identical to the CPU reference within machine precision. Figure 1 shows the computed solution fields and the scaled difference plots (multiplied by 10^{10} to amplify any discrepancy). The uniformly gray difference panel confirms no numerical artifacts are introduced by GPU parallelization. For Jacobi the GPU-CPU solutions are bitwise identical (max error 0.00); for RBGS the maximum absolute difference is 1.00×10^{-12} with an L2 error of 1.39×10^{-14} . The errors reported in Table 1 (5.07×10^{-6} for Jacobi, 7.60×10^{-9} – 2.00×10^{-9} for RBGS) represent the numerical discretization error relative to the analytical solution, which is governed by the convergence tolerance $\varepsilon = 10^{-4}$ and is independent of GPU parallelization. Although both methods share the same residual-based stopping criterion, the resulting error differs by several orders of magnitude because the relationship between residual and true error is method-dependent: SOR-accelerated RBGS attains a substantially smaller true error at the same residual threshold than plain Jacobi.

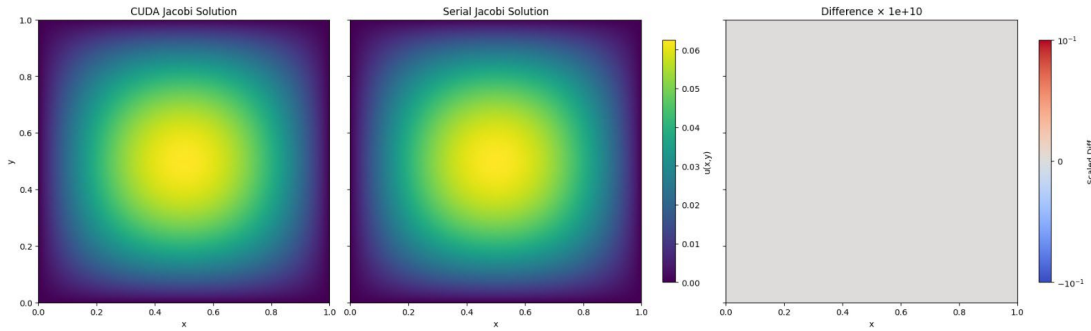


Figure 1a. Jacobi: CUDA and serial solutions with difference map scaled by 10^{10} . Gray panel confirms GPU-CPU agreement to machine precision

The three panels show, from left to right, the CUDA GPU solution, the serial CPU solution, and the absolute difference between them amplified by 10^{10} . Both solution fields display the expected smooth saddle-shaped profile determined by the boundary conditions and the sinusoidal source term.

The uniformly gray difference panel contains no visible structure, confirming that the GPU parallelization introduces no numerical artifacts. For the Jacobi method, the GPU and CPU solutions are bitwise identical, giving a maximum absolute error of exactly 0.00.

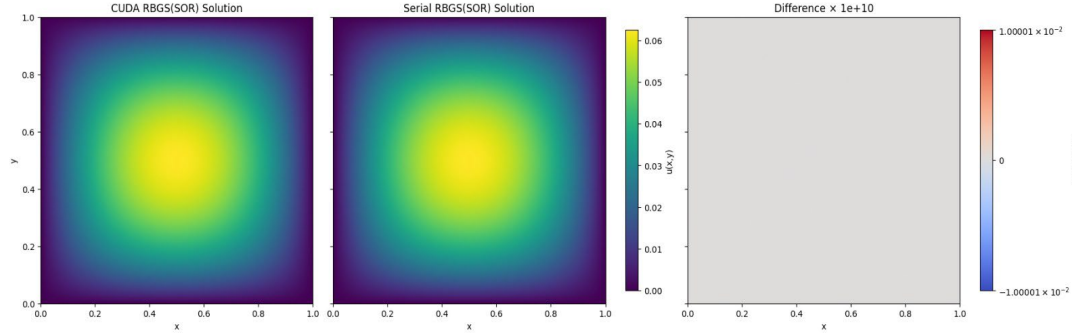


Figure 1b. RBGS: CUDA and serial solutions with difference map scaled by 10^{10} (right).
Max error 1.00×10^{-12} , L2 error 1.39×10^{-14}

5.2 Convergence: Jacobi vs. RBGS

Figure 2 compares iteration counts. At 1024×1024 , RBGS requires 3,060 iterations

($\epsilon = 10^{-4}$) while Jacobi needs 2,011,200 iterations ($\epsilon = 10^{-4}$) – a $657 \times$ difference. At 4096×4096 , RBGS converges in 13,220 iterations.

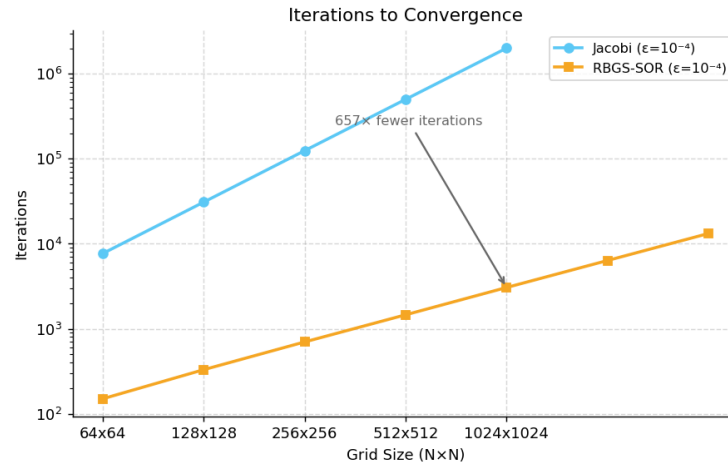


Figure 2. Iterations to convergence: Jacobi ($\epsilon = 10^{-4}$) vs. RBGS ($\epsilon = 10^{-4}$)

The y-axis (log scale) shows the number of iterations required to reach the convergence tolerance. At 1024×1024 , Jacobi needs 2,011,200 iterations ($\epsilon = 10^{-4}$) while RBGS converges in just 3,060 iterations ($\epsilon = 10^{-4}$) – a $657 \times$ difference. At 4096×4096 RBGS converges in 13,220 iterations. The near-linear growth of Jacobi iteration counts with grid size reflects its

$O(N^2)$ spectral radius dependence, whereas RBGS grows more slowly owing to its approximately $2 \times$ per-iteration convergence advantage from checkerboard Gauss-Seidel updating. Both methods use an identical residual-based stopping tolerance ($\epsilon = 10^{-4}$); RBGS nonetheless converges in far fewer iterations due to its superior per-iteration convergence rate.

5.3 Absolute Execution Time

Figure 3 presents absolute wall-clock GPU times alongside the CPU serial baseline. Although Jacobi-Shared achieves the highest speedup ratio, its

absolute GPU time is vastly larger: 86.954 s at 1024×1024 vs. 0.361 s for RBGS-Shared – a $241 \times$ difference. The high Jacobi speedup reflects the enormous serial iteration count, not GPU efficiency.

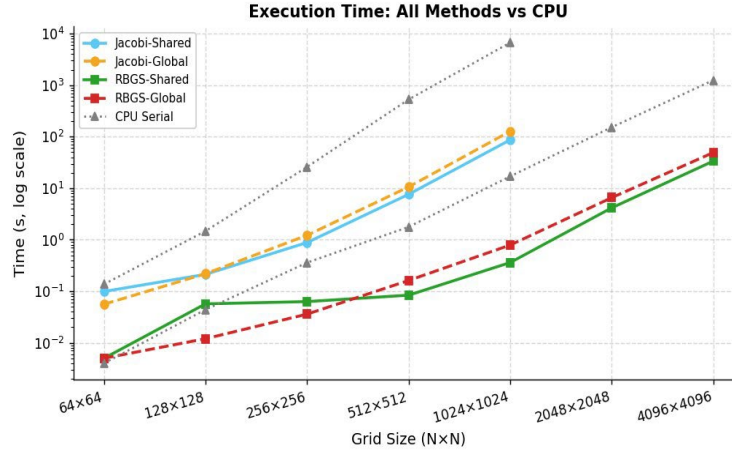


Figure 3. Absolute execution time: all four GPU variants and CPU serial baseline

Although Jacobi-Shared achieves the highest speedup ratio ($77.32 \times$), its absolute GPU time at 1024×1024 is 86.954 s – compared to just 0.361 s for RBGS-Shared, a $241 \times$ difference. This contrast reveals that the high Jacobi speedup reflects the enormous serial iteration count, not GPU efficiency. For any production use case where wall-clock time matters, RBGS-Shared is the clear choice: it is faster in absolute terms by orders of magnitude across all grid sizes and converges to a tighter tolerance. The CPU baseline (top curve) grows steeply, confirming the practical necessity of GPU acceleration for large grids.

5.4 Execution time

Figures 4a and 4b compare GPU execution times between shared and global memory variants. For Jacobi at 1024×1024 , the best global memory time is 127.489 s (32×32) vs. 86.954 s for shared memory (16×16), a 32% reduction. With 16×8 blocks, Jacobi-Shared takes 86.559 s vs. 193.918 s for Jacobi-Global – a $2.24 \times$ reduction. For RBGS at 4096×4096 with 16×8 blocks: 33.653 s (Shared) vs. 81.086 s (Global), a $2.41 \times$ reduction.

Execution time in seconds (log scale) is shown as a function of grid size for the serial CPU baseline and all four Jacobi GPU configurations, each at its best block. For small grids the CPU and GPU times are comparable; from 256×256 onward the GPU

variants pull decisively ahead. At 1024×1024 , Jacobi-Shared with a 16×16 block runs in 86.954 s compared to 127.489 s for Jacobi-Global with a 32×32 block (32% faster), while the CPU baseline requires 6,719 s. The 16×8 block shows an even larger gap: 86.559 s (Shared) vs. 193.918 s (Global), a $2.24 \times$ reduction. The CPU-to-GPU gap widens monotonically, confirming the scalability advantage of GPU parallelization for large Jacobi problems.

The same log-scale layout as Figure 4a, now for RBGS. Because RBGS converges in far fewer iterations than Jacobi, the absolute GPU times are orders of magnitude shorter: at 1024×1024 , RBGS-Shared takes only 0.361 s compared to 86.954 s for Jacobi-Shared. At 4096×4096 , RBGS-Shared runs in 33.653 s vs. 81.086 s for RBGS-Global with 16×8 blocks, a $2.41 \times$ reduction. The CPU serial baseline grows steeply to several thousand seconds at the largest grid, while RBGS-Shared remains below 34 s – establishing it as by far the most practical solver for large-scale 2D Poisson problems.

Figure 5 shows the Global/Shared time ratio across all four block configurations. For Jacobi the ratio approaches 1 for 32×32 blocks but rises with narrower blocks, reaching $2.24 \times$ for 16×8 at 1024×1024 . For RBGS all configurations show a ratio above 1 at large grids, with 16×8 reaching $2.41 \times$ at 4096×4096 .

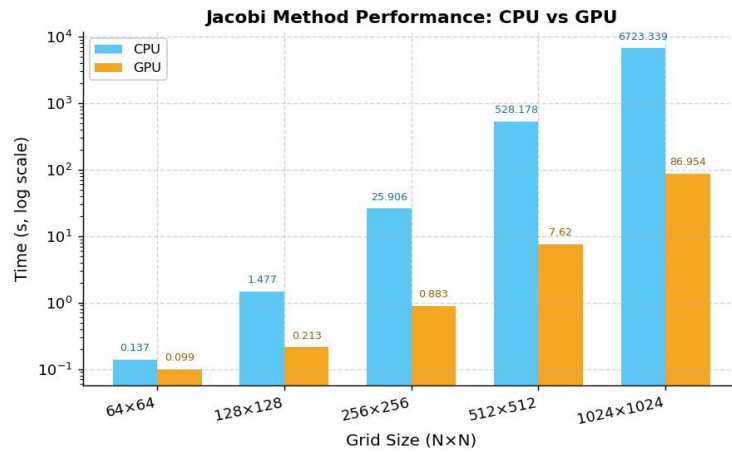


Figure 4a. Jacobi: CPU vs. GPU execution time (best block per grid size)

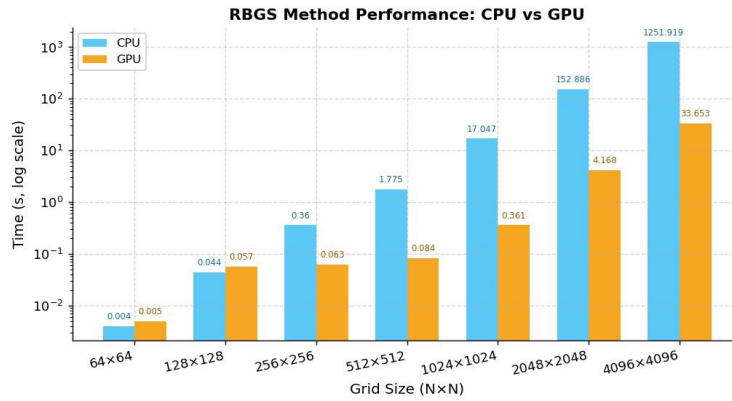


Figure 4b. RBGS: CPU vs. GPU execution time (best block per grid size)

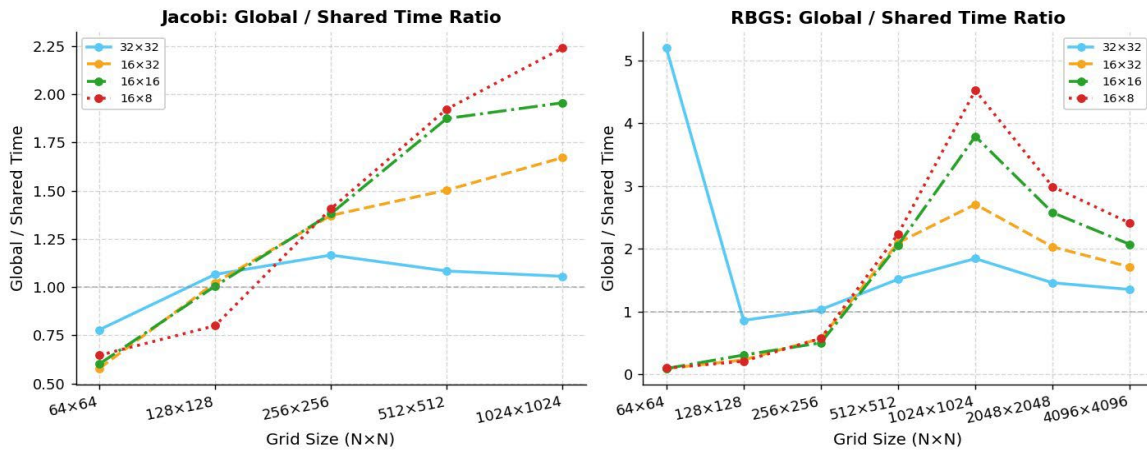


Figure 5. Global-to-Shared execution time ratio for all block configurations and both methods. Ratio > 1 means shared memory is faster

Each curve plots the ratio $T_{\text{global}}/T_{\text{shared}}$ for one block configuration across all five grid sizes, separately for Jacobi (left panel) and RBGS (right panel). A ratio above 1 means shared memory is faster. For Jacobi, the 32×32 block yields a ratio near 1 at all grid sizes, because wide warps already achieve good global memory coalescing and tiling adds little benefit. Narrower blocks, especially 16×8 , expose the full advantage of on-chip caching and reach $2.24 \times$ at 1024×1024 . For RBGS every block configuration sits above 1 at large grid sizes, with 16×8 reaching $2.41 \times$ at 4096×4096 . The figure directly quantifies that narrow, asymmetric thread blocks are the primary beneficiaries of shared memory tiling.

5.5 Speedup

Figure 6 shows GPU speedup over the CPU baseline for all four variants. For small grids (64×64 , 128×128), speedups are near or below $1 \times$ for all methods due to kernel launch overhead. As grid size increases, the GPU advantage grows rapidly. Jacobi-Shared reaches $77.32 \times$ at 1024×1024 (16×16 block); Jacobi-Global achieves $50.18 \times$ (32×32 block). The peak at 1024×1024 is partly attributable to an L2 cache effect: all computations use double precision

(float64); the two solution arrays occupy ~ 16 MB ($2 \times 1024^2 \times 8$ bytes), fitting within the 32 MB GPU L2 cache. At this size, repeated stencil accesses benefit from L2 residency and near-peak bandwidth utilization. For larger grids, the working set exceeds L2 capacity and the solver becomes memory-bandwidth-bound on global DRAM; RBGS-SOR avoids this through its much lower iteration count. RBGS-Shared achieves $47.24 \times$ at 1024×1024 and sustains $37.20 \times$ at 4096×4096 ; RBGS-Global achieves $21.29 \times$ and $31.85 \times$ respectively.

The x-axis shows grid size (64×64 to 4096×4096) and the y-axis shows speedup over the single-threaded CPU baseline, with each variant plotted at its best block configuration. At small grids (64×64 , 128×128), all methods fall near or below $1 \times$ speedup because kernel launch overhead and host-device memory transfers dominate. As grid size grows the GPU advantage rises rapidly: Jacobi-Shared peaks at $77.32 \times$ at 1024×1024 , partly due to L2 cache residency of the ~ 16 MB working set. RBGS-Shared achieves $47.24 \times$ at 1024×1024 and sustains $37.20 \times$ at 4096×4096 . Shared memory variants consistently outperform their global memory counterparts at every grid size.

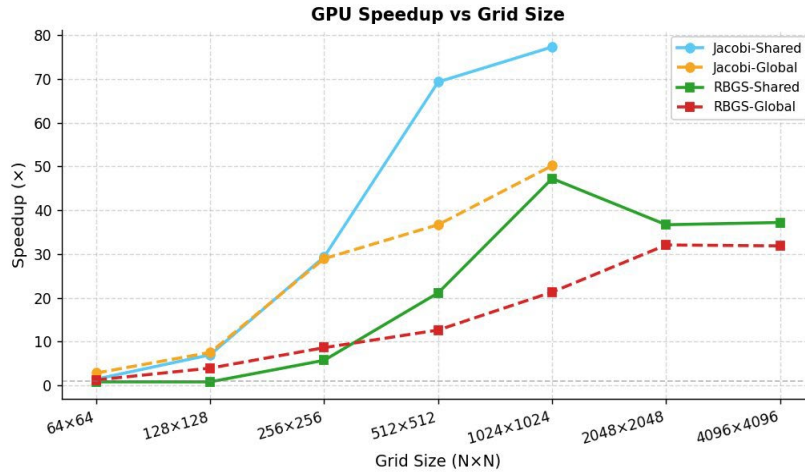


Figure 6. GPU speedup vs. grid size for all four solver variants

Table 1. Best results: peak speedup configuration per method at representative grid sizes

Method	Best Block	Grid Size	GPU Time (s)	CPU Time (s)	Speedup ($\times$)	Max Error
Jacobi-Shared	16 $\times$ 16	1024 $\times$ 1024	86.954	6723.339	77.32	5.07×10^{-6}
Jacobi-Global	32 $\times$ 32	1024 $\times$ 1024	127.489	6397.235	50.18	5.07×10^{-6}
RBGS-Shared	16 $\times$ 8	1024 $\times$ 1024	0.361	17.047	47.24	7.60×10^{-9}
RBGS-Global	32 $\times$ 32	1024 $\times$ 1024	0.789	16.795	21.29	7.60×10^{-9}
RBGS-Shared	16 $\times$ 8	4096 $\times$ 4096	33.653	1251.919	37.20	2.00×10^{-9}
RBGS-Global	32 $\times$ 32	4096 $\times$ 4096	49.268	1569.043	31.85	2.00×10^{-9}

Table 1 consolidates the peak performance figures for each solver at its optimal block configuration. Several observations follow. First, Jacobi-Shared delivers the highest raw speedup ratio (77.32 $\times$ at 1024 $\times$ 1024 with a 16 $\times$ 16 block), yet its absolute GPU execution time is 86.954 s – more than 240 $\times$ slower in wall-clock terms than RBGS-Shared at the same grid size (0.361 s). This contrast underscores that a high speedup ratio can be misleading when the CPU baseline itself is pathologically slow due to excessive iteration counts. Second, RBGS-Shared is superior on every practical criterion: it achieves tighter accuracy (7.60×10^{-9} vs. 5.07×10^{-6} for Jacobi), faster absolute runtime, and sustains strong speedup (37.20 $\times$) even at the largest tested grid (4096 $\times$ 4096) where Jacobi becomes infeasible. Third, shared memory variants consistently outperform their global memory counterparts: Jacobi-Shared beats Jacobi-Global by 32% at 1024 $\times$ 1024 (86.954 s vs. 127.489 s), and RBGS-Shared beats RBGS-Global by 54% at the same size (0.361 s vs. 0.789 s), with the gap widening further at 4096 $\times$ 4096. Taken together, the table confirms that RBGS-Shared with a 16 $\times$ 8 block is the recommended configuration for production-scale 2D Poisson solves on the tested GPU hardware.

6. Conclusion

This study presented a comparative analysis of four GPU-accelerated iterative solvers for the two-dimensional Poisson equation. By implementing Jacobi-Global, Jacobi-Shared, RBGS-Global, and RBGS-Shared in CUDA and benchmarking them across grid sizes from 64 $\times$ 64 to 1024 $\times$ 1024, with RBGS additionally benchmarked up to 4096 $\times$ 4096, we systematically evaluated the impact of solver choice and memory strategy.

First, shared memory tiling consistently outperforms global memory, with execution time reductions of up to 2.41 $\times$ for narrow block

configurations at large grid sizes. The benefit grows with grid size and inversely with block width.

Second, RBGS converges in 657 $\times$ fewer iterations than Jacobi at 1024 $\times$ 1024, translating to a 241 $\times$ reduction in absolute GPU execution time. For any large-scale computation, RBGS is therefore the clearly preferable solver. The high Jacobi speedup ratio reflects the slow CPU baseline rather than practical GPU efficiency.

Third, GPU acceleration is effective for grids of 256 $\times$ 256 and above, delivering speedups of 20–77 $\times$ over a single-threaded CPU. For grids of 128 $\times$ 128 and smaller, overhead prevents any benefit.

Fourth, the optimal block configuration is memory-strategy-dependent: global memory variants favor 32 $\times$ 32 blocks for coalescing; shared memory variants perform best with 16 $\times$ 16 or 16 $\times$ 8 blocks.

Future work includes multi-threaded CPU baselines, multi-GPU domain decomposition, mixed-precision implementations, multigrid preconditioning with RBGS smoothers, and extension to three-dimensional problems.

Acknowledgments

The authors express their sincere gratitude to the Department of Computer Science of the Faculty of Information Technologies at Al-Farabi Kazakh National University for providing the academic environment and computational resources that made this research possible. The authors also thank colleagues and reviewers whose comments and discussions helped to improve the quality of this manuscript.

Funding

This research was funded by the Committee of Science of the Ministry of Science and Higher Education of the Republic of Kazakhstan under Grant Funding project No. AP23488807 “Code Generation and Optimization for Numerical Simulation via Multi-Dimensional Homomorphisms and Model Driven Architecture”.

Author Contributions

Conceptualization, E.M.; Methodology, E.M. and B.S.; Software, B.S.; Validation, E.M. and B.S.; Formal Analysis, B.S.; Investigation, B.S.; Resources, E.M.; Data Curation, B.S.; Writing – Original Draft Preparation, B.S.; Writing – Review

& Editing, E.M. and B.S.; Visualization, B.S.; Supervision, E.M.; Project Administration, E.M.; Funding Acquisition, E.M. All authors have read and agreed to the published version of the manuscript.

References

1. J. Chen, R.-C. Chen, and J. Liu, "A GPU Poisson-Fermi solver for ion channel simulations," *Comput. Phys. Commun.*, vol. 229, pp. 94–102, 2018, doi: 10.1016/j.cpc.2018.04.002.
2. R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations*. Philadelphia, PA, USA: SIAM, 2007, doi: 10.1137/1.9780898717839.
3. NVIDIA Corporation, *CUDA C++ Programming Guide*, Version 12.x. Santa Clara, CA, USA: NVIDIA, 2023. [Online]. Available: <https://docs.nvidia.com/cuda/archive/12.3.0/cuda-c-programming-guide/genindex.html>
4. Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: SIAM, 2003, doi: 10.1137/1.9780898718003.
5. J. W. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: SIAM, 1997, doi: 10.1137/1.9781611971446.
6. N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys, "A multigrid solver for boundary value problems using programmable graphics hardware," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop Graph. Hardw.*, San Diego, CA, USA, 2003, pp. 102–111, doi: 10.1145/1198555.1198784.
7. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*. Bristol, U.K.: Adam Hilger, 1988, doi: 10.1201/9780367806934.
8. N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corp., Santa Clara, CA, USA, Tech. Rep. NVR-2008-004, Dec. 2008. [Online]. Available: https://research.nvidia.com/publication/2008-12_efficient-sparse-matrix-vector-multiplication-cuda
9. NVIDIA Corporation, "Multi-GPU programming with standard parallel C++ — Jacobi example," *NVIDIA Developer Blog*, 2022. [Online]. Available: <https://developer.nvidia.com/blog/multi-gpu-programming-with-standard-parallel-c-part-1/>
10. NVIDIA Corporation, "Benchmarking CUDA-aware MPI — Jacobi Poisson solver example," *NVIDIA Developer Blog*, 2013. [Online]. Available: <https://developer.nvidia.com/blog/benchmarking-cuda-aware-mpi/>
11. M. Aslam, O. Riaz, S. Mumtaz, and A. D. Asif, "Performance comparison of GPU-based Jacobi solvers using CUDA provided synchronization methods," *IEEE Access*, vol. 8, pp. 31792–31812, 2020, doi: 10.1109/ACCESS.2020.2973669.
12. T.-H. Chiu and C.-A. Lin, "Multigrid accelerated projection method on GPU cluster for the simulation of turbulent flows," *J. Mech.*, vol. 39, pp. 199–212, 2023, doi: 10.1093/jom/ufad015.
13. Q. Li, R. Li, and Z. Yang, "An incompressible flow solver on a GPU/CPU heterogeneous architecture parallel computing platform," *Theor. Appl. Mech. Lett.*, vol. 13, no. 5, 2023, Art. no. 100474, doi: 10.1016/j.taml.2023.100474.
14. M. ElMaghrbay, R. A. Ammar, and S. Rajasekaran, "Fast GPU algorithms for implementing the Red-Black Gauss-Seidel method for solving partial differential equations," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Split, Croatia, 2013, pp. 269–274, doi: 10.1109/ISCC.2013.6754958.
15. A. Kochurov and D. Golovashkin, "GPU implementation of Jacobi method and Gauss-Seidel method for data arrays that exceed GPU-dedicated memory size," *J. Math. Model. Algorithms Oper. Res.*, vol. 14, pp. 393–405, 2015, doi: 10.1007/s10852-015-9272-5.
16. B. Bozorgmehr, P. Willemsen, J. A. Gibbs, R. Stoll, J.-J. Kim, and E. R. Pardyjak, "Utilizing dynamic parallelism in CUDA to accelerate a 3D red-black successive over relaxation wind-field solver," *Environ. Model. Softw.*, vol. 137, 2021, Art. no. 104958, doi: 10.1016/j.envsoft.2021.104958.
17. U. Trottenberg, C. Oosterlee, and A. Schuller, *Multigrid*. London, U.K.: Academic Press, 2001.
18. A. Valero-Lara, P. Valero-Lara, and I. Luque-Varela, "Application of graphics processor unit computing resources to solution of incompressible fluid dynamics problems," *Computers*, vol. 14, no. 5, 2025, Art. no. 170, doi: 10.3390/computers14050170.
19. J. M. Mantas and F. Vecil, "Efficient GPU implementation of a Boltzmann-Schrodinger-Poisson solver for the simulation of nanoscale DG MOSFETs," *J. Supercomput.*, vol. 79, pp. 14117–14150, 2023, doi: 10.1007/s11227-023-05189-0.
20. J. Hoefler, T. Schneider, and A. Lumsdaine, "Evaluation of programming models and performance for stencil computation on current GPU architectures," 2024, arXiv:2404.04441.
21. L. Zhang et al., "Revisiting temporal blocking stencil optimizations," 2023, arXiv:2305.07390.
22. P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proc. 2nd Workshop Gen. Purpose Process. Graph. Process. Units (GPGPU-2)*, 2009, pp. 79–84, doi: 10.1145/1513895.1513905.

Information about the authors:

Erlan Makhmut – PhD, Department of Computer Science, Faculty of Information Technologies, al-Farabi Kazakh National University (Almaty, Kazakhstan; e-mail: erlanmahimut@gmail.com). Research interests: GPU/CUDA computing, quantum-classical hybrid algorithms and numerical methods for PDEs. ORCID iD: 0009-0002-3451-415X.

Beiimbet Sansyzbay – Master's student, Department of Computer Science, Faculty of Information Technologies, al-Farabi Kazakh National University (Almaty, Kazakhstan). Research interests: model-driven architecture (MDA), automatic code generation and GPU programming with CUDA C++. ORCID iD: 0009-0008-7748-7933.

Kanat Aidarov – PhD, Head of ESB Development, a specialist in distributed systems and cloud computing, with a strong background in applied mathematics and computer science. His dissertation focuses on cloud computing and presents an approach to controlling workload in large-scale cloud systems by accepting only the requests the system can process and rejecting excess requests, thereby maintaining high performance and stability. He leads a team developing high-performance systems capable of serving millions of users simultaneously. He has extensive experience in optimizing the performance of high-load systems and protecting them from overload, ensuring low latency and stable throughput even under heavy workloads. ORCID iD: 0000-0002-8489-0092.

Submission received: 25 May, 2026.

Revised: 25 August, 2026.

Accepted: 25 August, 2026.