DESIGN AND ANALYSIS OF GPU-ACCELERATED 2D POISSON SOLVERS FOR PLASMA SIMULATION

Bálint Tóth and Zoltán Juhász

University of Pannonia, Department of Electrical Engineering and Information Systems

OUTLINE

1. Problem statement
2. Multigrid method: theory and CUDA implementation
3. Spectral method: theory and implementation with cuFFT
4. Implementation details
5. Results: runtime performance and numerical accuracy

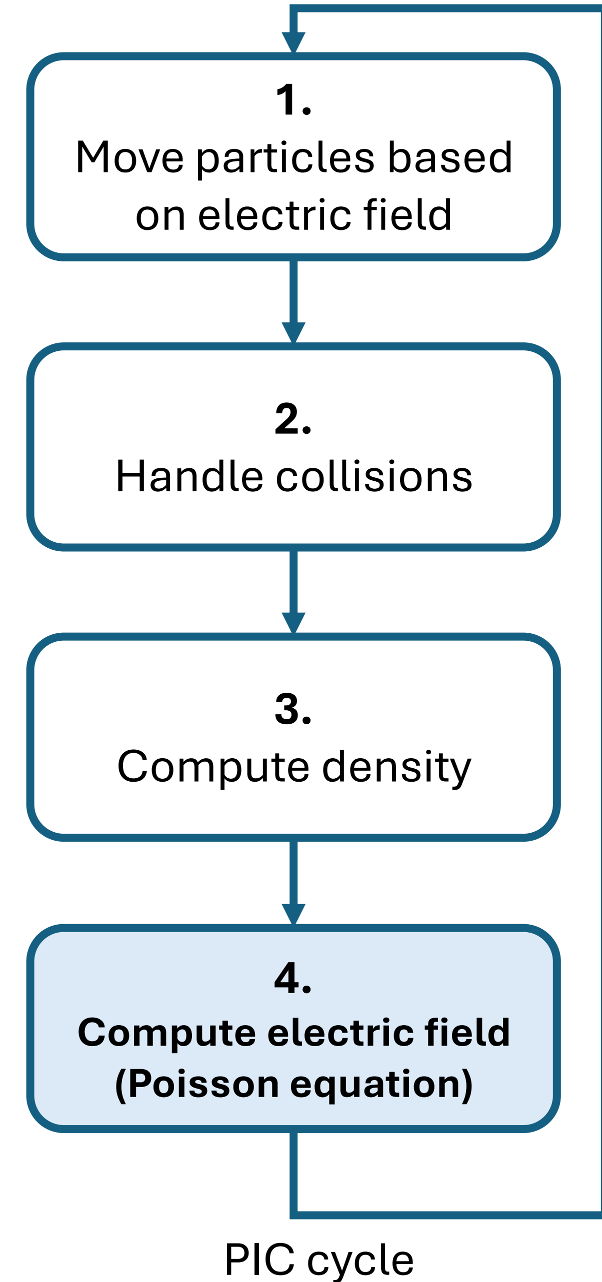
INTRODUCTION

Particle In Cell / Monte Carlo Collisions (PIC/MCC)

- Particle-Mesh technique
- Monte Carlo collision handling

Poisson equation solver is the bottleneck

- Grid size: 255×255
- Total PIC-cycle time: 22.5 ms
- **Poisson solver: 2×6 ms (54% of the cycle)**

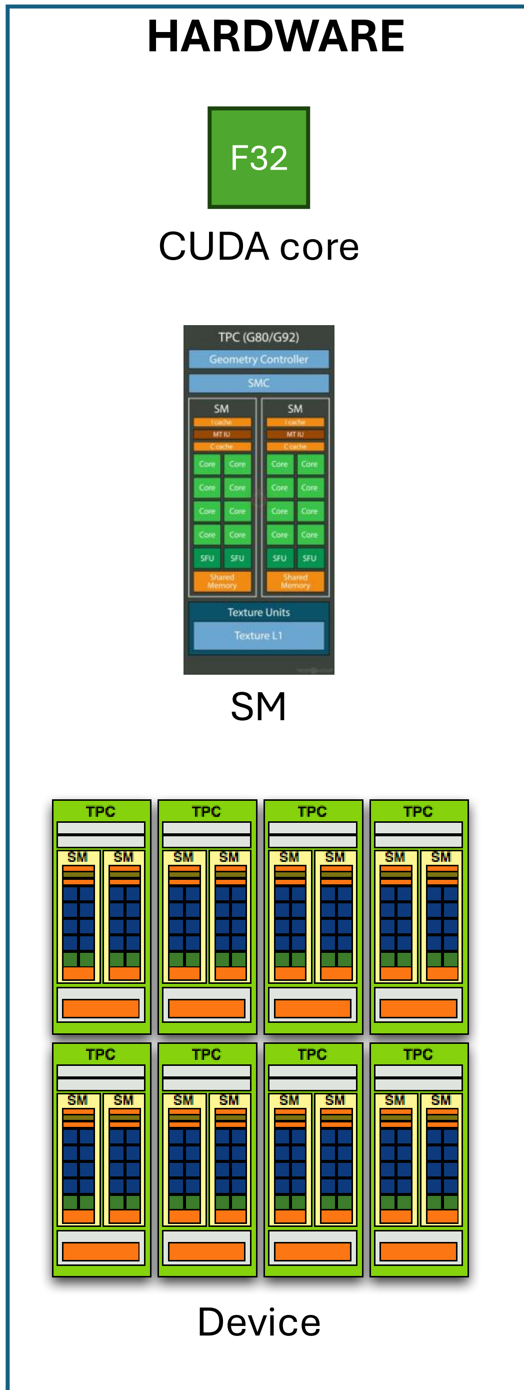


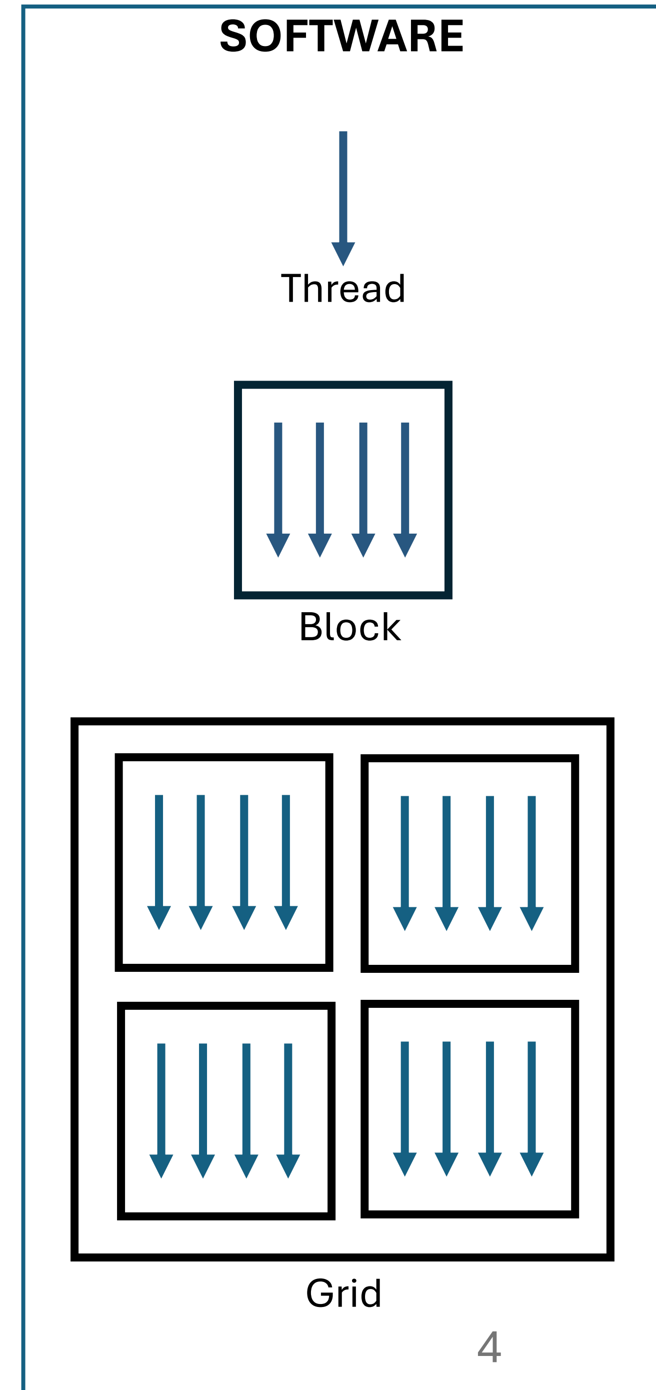
CUDA PROGRAMMING MODEL

Kernel: basic programmable unit, describing one thread

Hierarchy

- Hardware
- Threads
- Memory





POISSON-EQUATION SOLVERS

$$\nabla^2 \phi = \frac{\rho}{\varepsilon_0}$$

Direct

- Gaussian Elimination
- **Spectral**

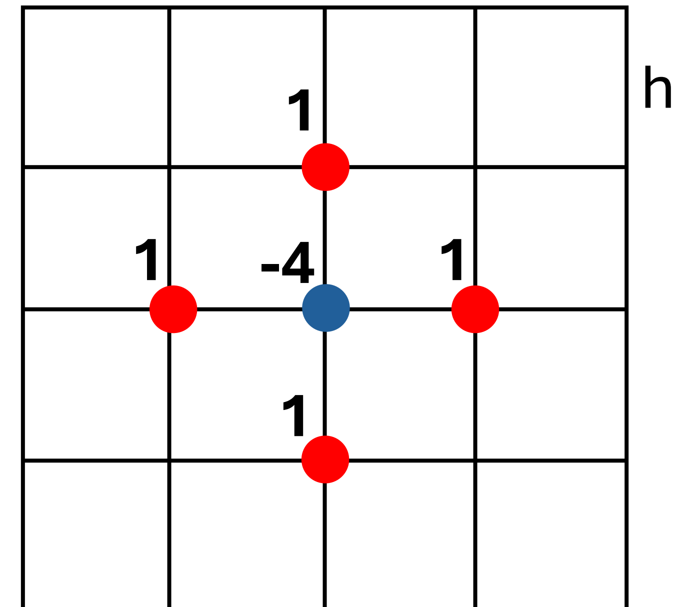
Iterative

- Jacobi iteration
- Gauss-Seidel iteration, Successive Over Relaxation
- **Multigrid**

Discretisation using **finite differences**:

$$\frac{1}{h^2}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}\right) = \frac{\rho}{\varepsilon_0}$$

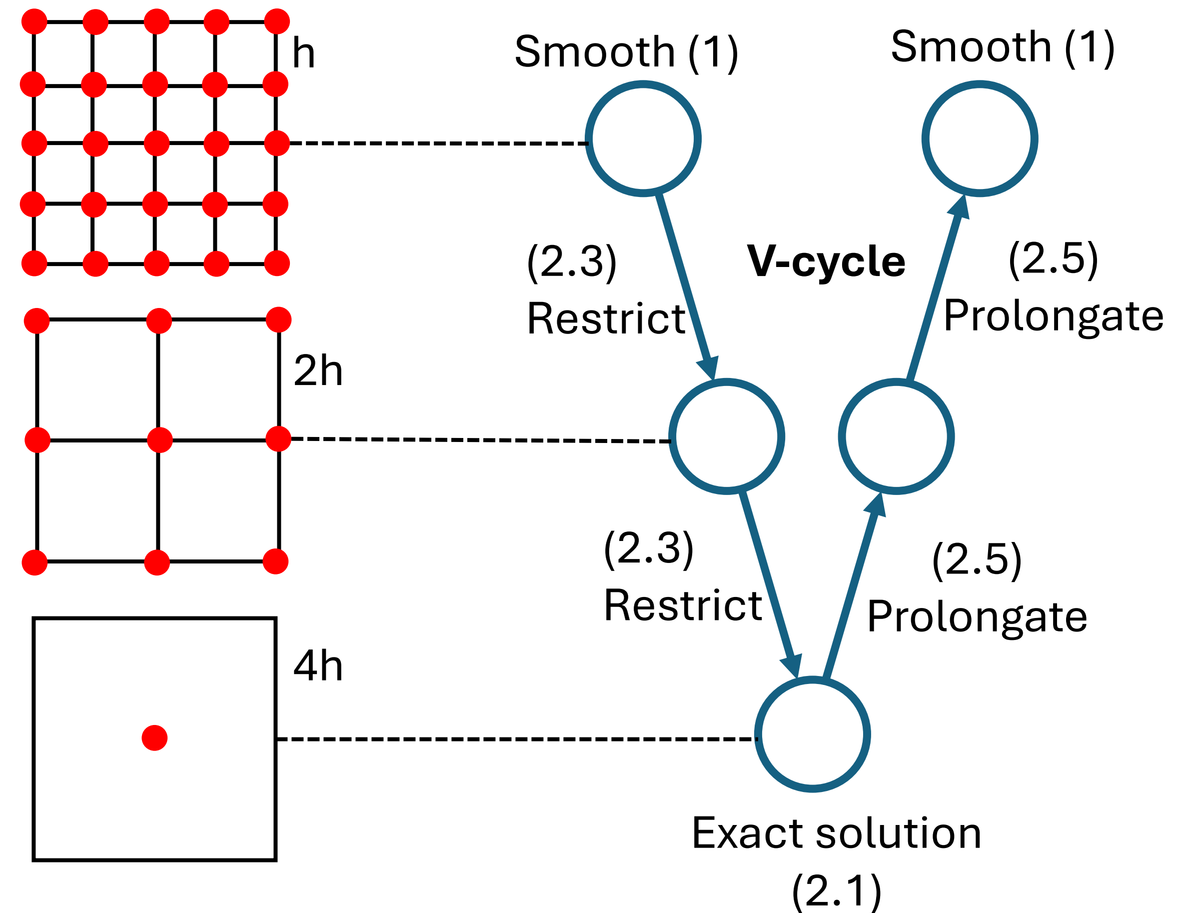
5-point star on the grid:



GEOMETRIC MULTIGRID METHOD[1]

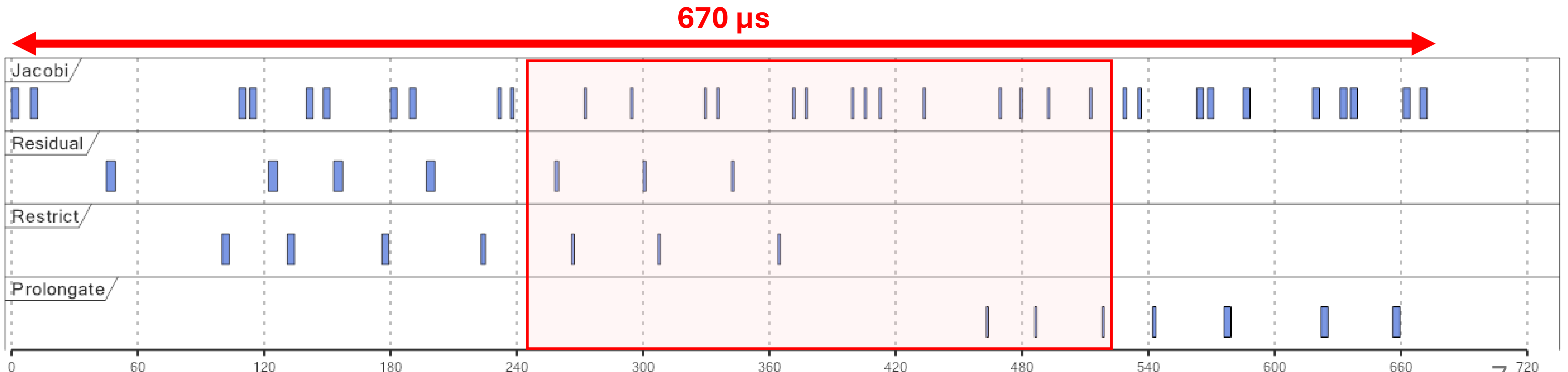
V-cycle:

```
procedure V_cycle
begin
1.   error smoothing
2.   if on coarsest grid then
2.1    compute exact solution
     else
2.2    compute residual
2.3    restrict to coarser grid
2.4  call V_cycle
2.5    prolongate to finer grid
     endif
3.   error smoothing
end
```



GPU IMPLEMENTATION

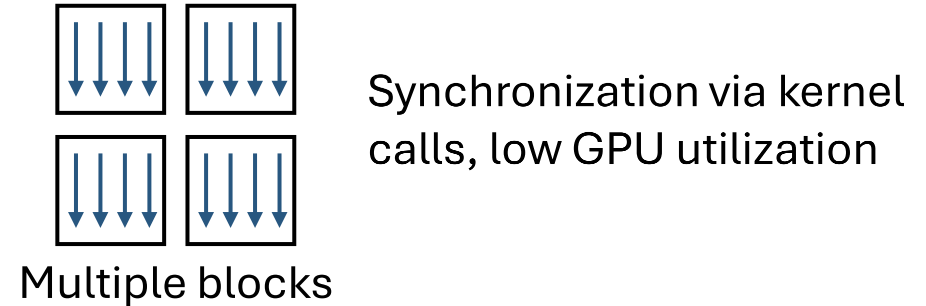
- Baseline sequential implementation (C++)
  - Matlab reference implementation[2]
- CUDA implementation
  - Each step runs as a GPU kernel
  - One thread - One grid point



FUSED-KERNEL OPTIMIZATION

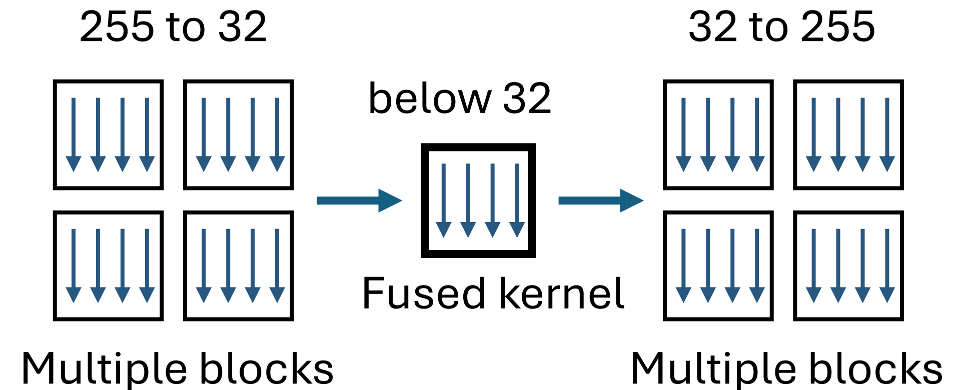
Naive implementation

- Many, small GPU grids
- Latencies, no parallelism

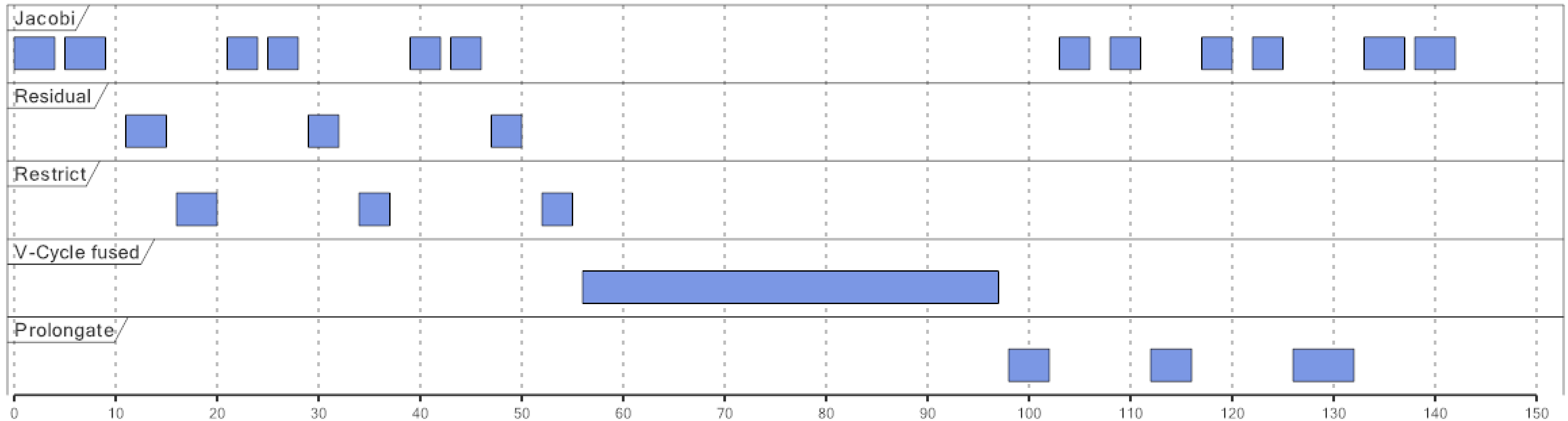


Fuse below 32×32

- **Single CUDA block** with internal synchronization
- Single kernel combined with
- 10x speedup for 32×32 and 64×64



Optimized GPU implementation



GPU utilization improvement

- Naive implementation: 20%
- Fused kernel implementation: 40 – 80%

V-Cycle runtimes and speedups

Grid size	Sequential baseline	CUDA Multigrid RTX6000 Ada generation		CUDA Multigrid A100	
	time [μs]	time [μs]	speedup	time [μs]	speedup
255×255	569.94	120.16	4.7	103.90	5.5
511×511	2550.70	184.35	13.8	146.13	17.5
1023×1023	12195.24	391.35	31.2	272.48	44.8

SPECTRAL METHOD[3]

Constant 0 Dirichlet boundary condition

$$\frac{\rho}{\varepsilon_0} = \frac{1}{h^2}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}\right)$$

Discretised Poisson-equation

$$\hat{\rho}_{m,n} = \frac{2}{J}\frac{2}{L}\sum_{j=1}^{J-1}\sum_{l=1}^{L-1}\rho_{j,l}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L}$$

1. Apply 2D forward Sine-transform

$$\hat{\phi}_{m,n} = \frac{h^2\hat{\rho}_{m,n}}{2\left(\cos\frac{\pi m}{J} + \cos\frac{\pi n}{L} - 2\right)\varepsilon_0}$$

2. Direct solution

$$\phi_{m,n} = \frac{2}{J}\frac{2}{L}\sum_{j=1}^{J-1}\sum_{l=1}^{L-1}\hat{\phi}_{j,l}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L}$$

3. Apply 2D inverse Sine-transform

Implementation using FFT

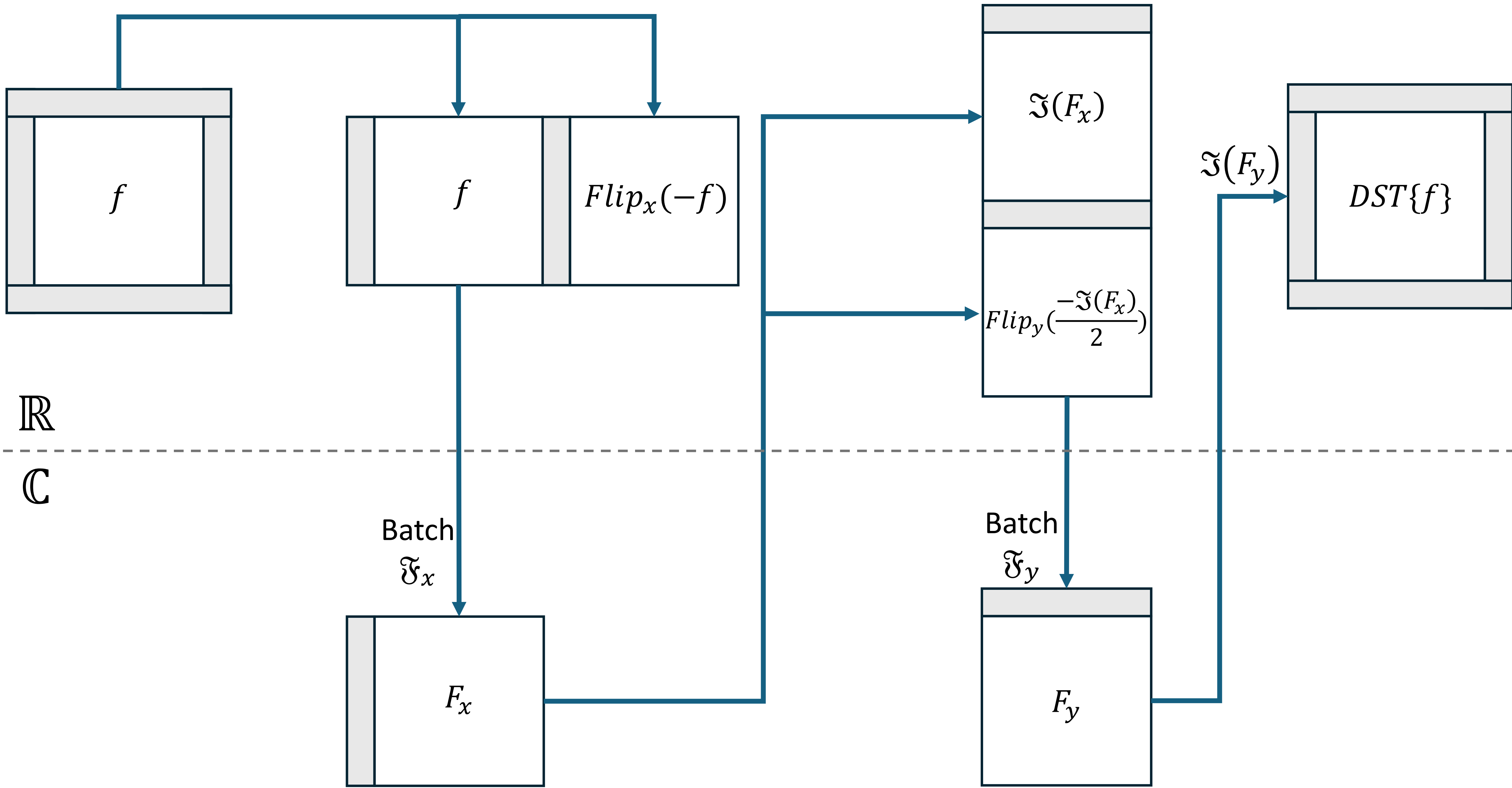
Fast Sine-transform:

- Extend the grid to get an ”odd signal”
- Use Batch-FFT

$$f'_{2N-j} := -f_j, j = 0,1, \dots, N-1$$

$$F_k = \underbrace{\sum_{j=0}^{2N-1} f_j e^{\frac{2\pi ijk}{2N}}}_{\text{DFT}} = 2i \underbrace{\sum_{j=0}^{N} f'_j \sin \frac{\pi jk}{N}}_{\text{DST}}$$

```
procedure Spectral
begin
    1.1 Horizontally duplicate grid
    1.2 Horizontal batch FFT (cuFFT)
DST 1.3 Vertically duplicate grid
    1.4 Vertical batch FFT (cuFFT)
    2.  Compute solution
     3.1 Horizontally duplicate grid
     3.2 Horizontal batch FFT (cuFFT)
IDST 3.3 Vertically duplicate grid
     3.4 Vertical batch FFT (cuFFT)
end
```



Buffering scheme for Fast-Sine transform

API

- Struct for configuration and workspace
- Functions for lifecycle: initialize, solve, destroy
- Both workspaces work with the same function names

```
SpectralSolverConfig cfg = { 0 };
cfg.block_size = 16;
cfg.grid_dimension_x = 1.0;
cfg.grid_dimension_y = 1.0;
cfg.N_G_x = 255;
cfg.N_G_y = 255;
```

Hyperparameters for the algorithm

```
SpectralSolverWorkspace wsp = { 0 };
init_workspace(&wsp, &cfg);
solve(&wsp, d_u, d_f);
destroy_workspace(&wsp);
```

Solve can be called any times with the same workspace

RESULTS

- Numerical correctness: 10^{-9} maximum absolute error

Grid size	CUDA Multigrid 1 V-cycle	CUDA Multigrid 20 V-cycles	CUDA Spectral (FFT) A100		
	time [μs]	time [μs]	time [μs]	speedup	cumulative speedup
255×255	103.90	2078.00	121.70	17.1	**~100x**
511×511	146.13	2922.60	144.64	20.2	
1023×1023	272.48	5449.60	883.91	6.2	

SUMMARY

New general purpose 2D Poisson solver implementations

- Speedups
  - Multigrid: up to 40x compared to a sequential C++ implementation
  - Spectral: 20x compared to the optimized CUDA Multigrid, 100x compared to the original solver
- Further development
  - Multigrid: convergence test, FMG-cycle
  - Spectral: Direct Fast DST instead of the FFTs

Acknowledgements: EKÖP, Code: 2025--2.1.1--EKÖP--2025--00029/46) of the National Fund for Research; Development and Innovation, DKF Ltd. for providing access to the Hungarian supercomputer Komondor; Nvidia Academic Grant Program.

References

1. W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial, Second Edition*. SIAM, 2000.
2. D. Appelhans, "*MultiGridMatlab.*" https://github.com/dappelha/MultiGridMatlab (accessed Mar. 01, 2026).
3. Teukolsky, S. A., Flannery, B. P., Press, W., & Vetterling, W. (1992). *Numerical recipes* in C. *SMR*, *693*(1)