Tridiagonal GPU Solver with Scaled Partial Pivoting at Maximum Bandwidth

Christoph Klein Robert Strzodka

50th International Conference on Parallel Processing

Heidelberg University
ZITI
Application Specific Computing

Introduction

Introduction

$$Ax = d$$

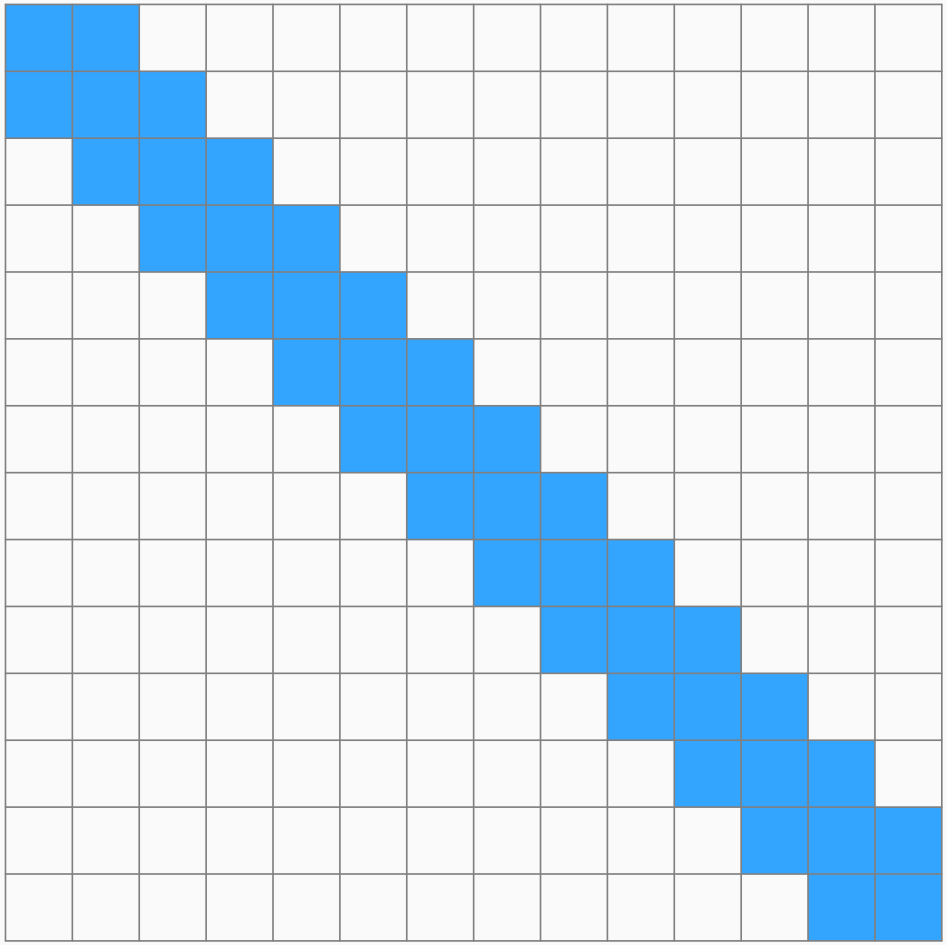
$$A = \begin{pmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & & \ddots & & \\ & & & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & & a_{N-1} & b_{N-1} \end{pmatrix}$$

Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

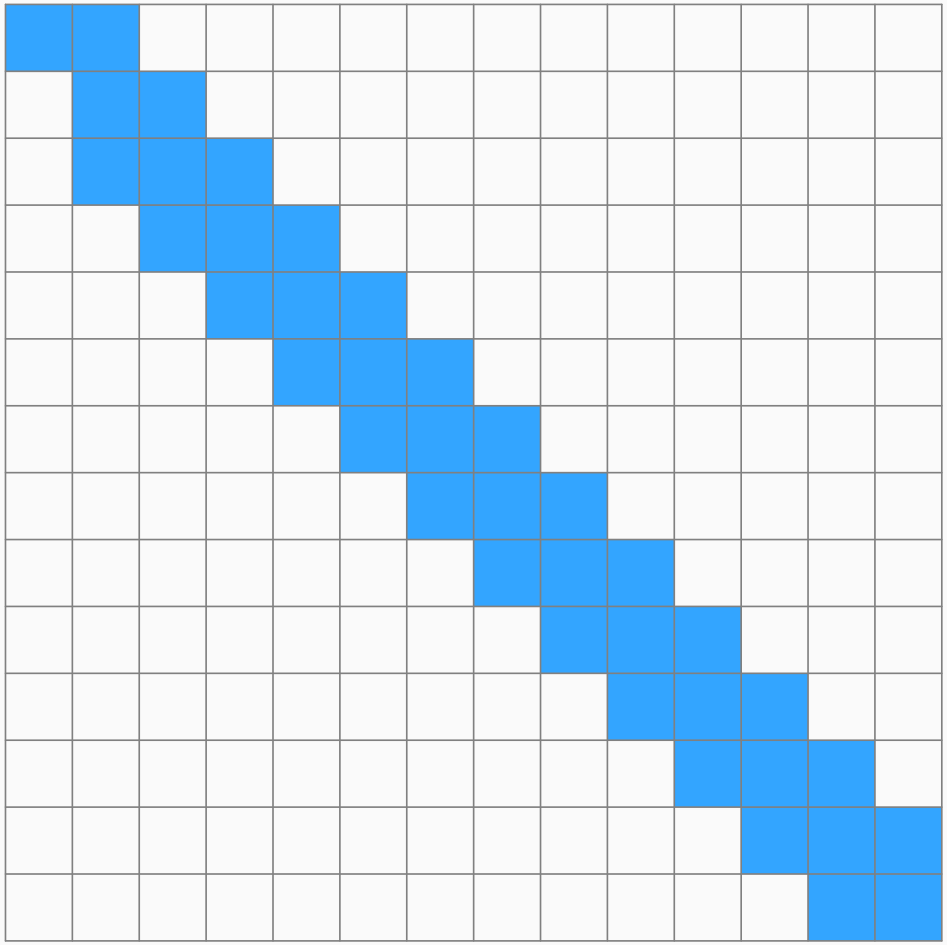


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

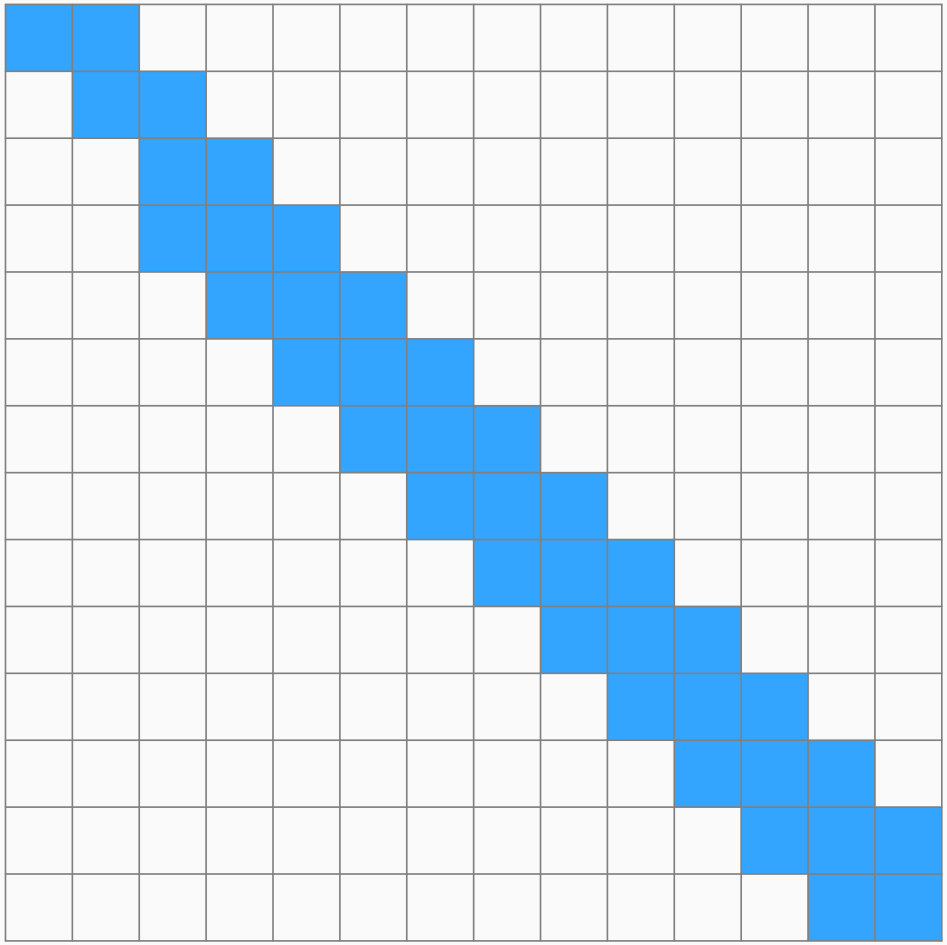


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

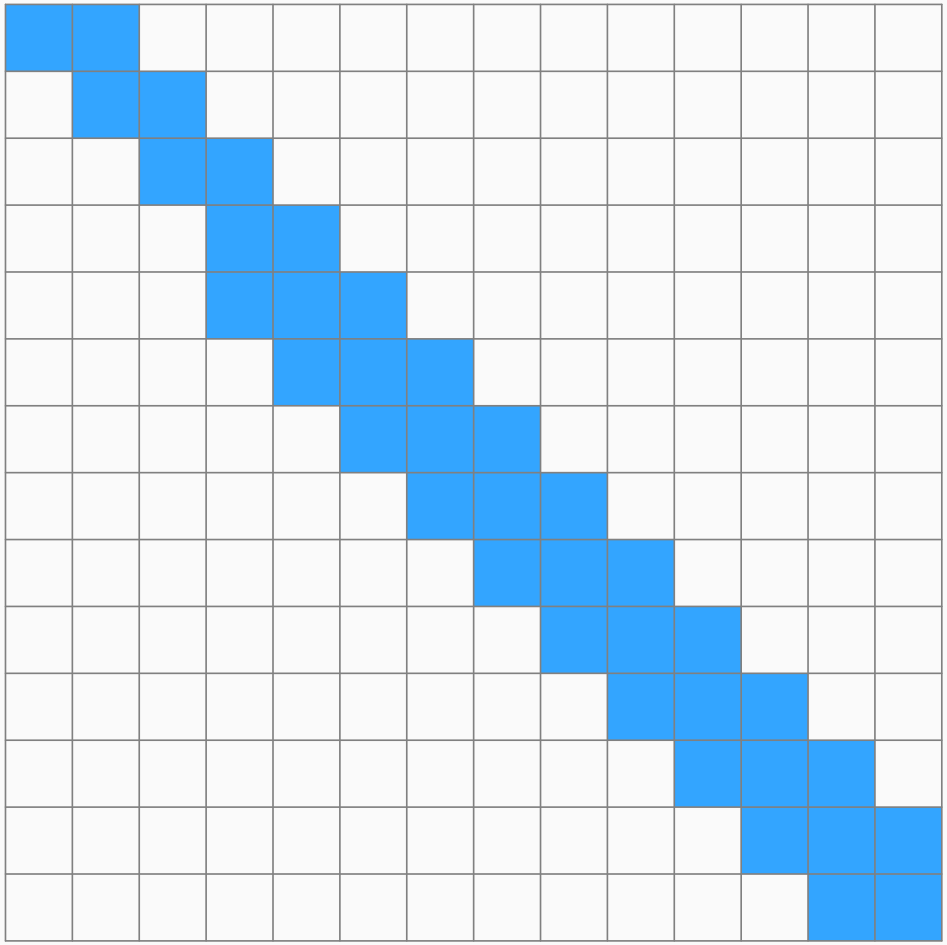


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

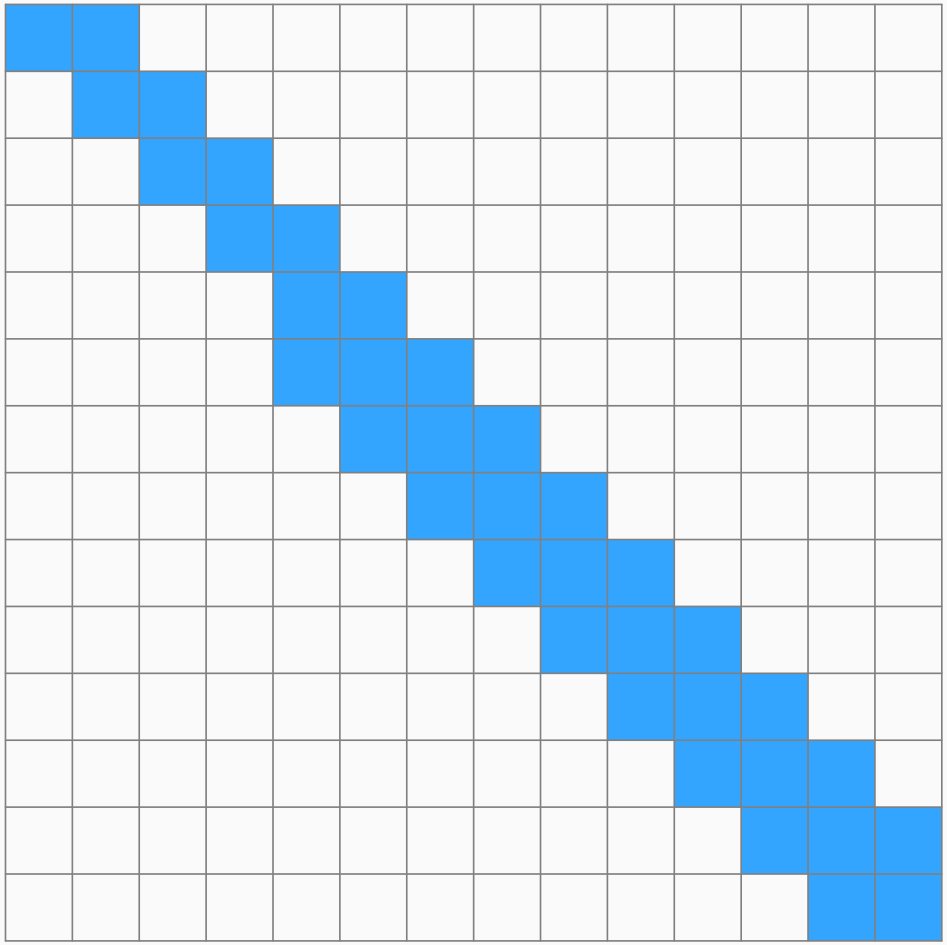


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

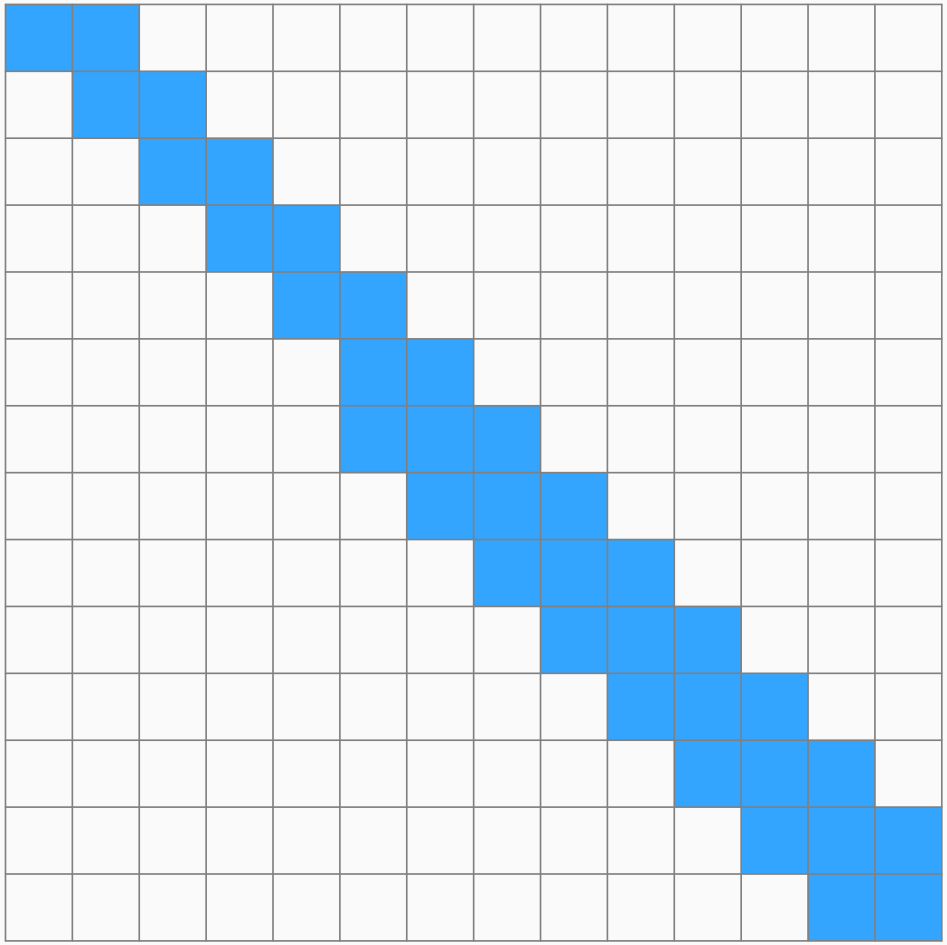


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

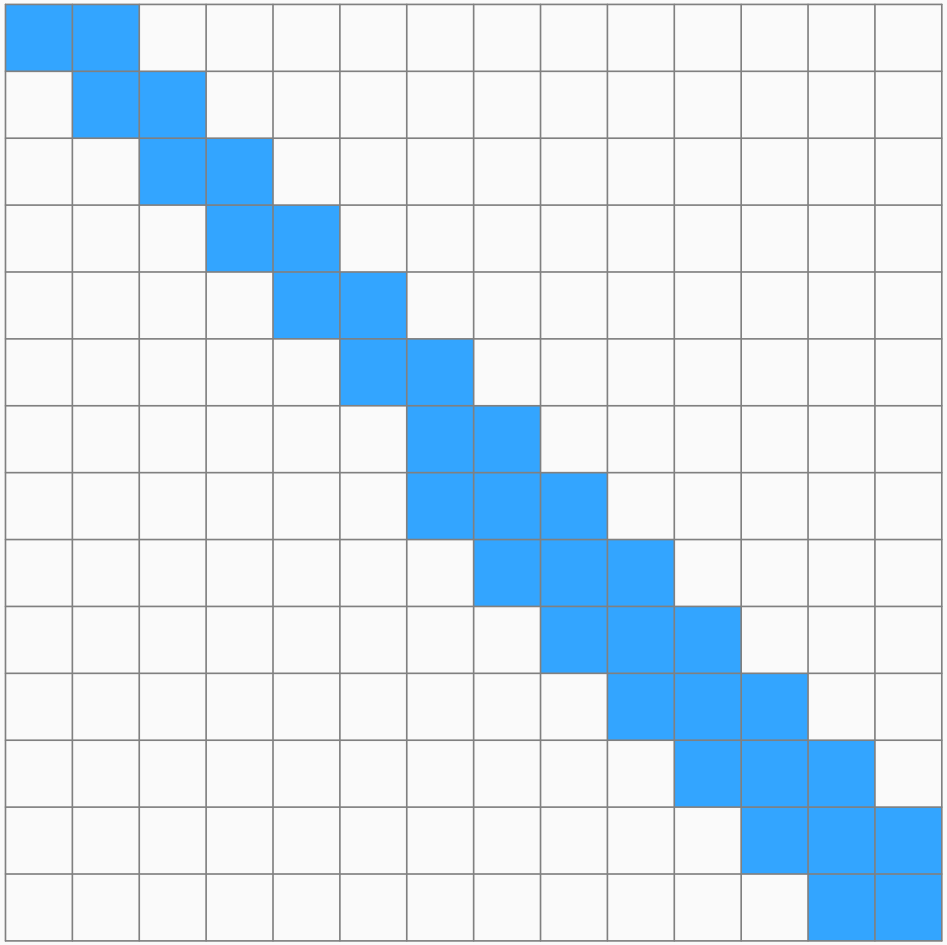


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

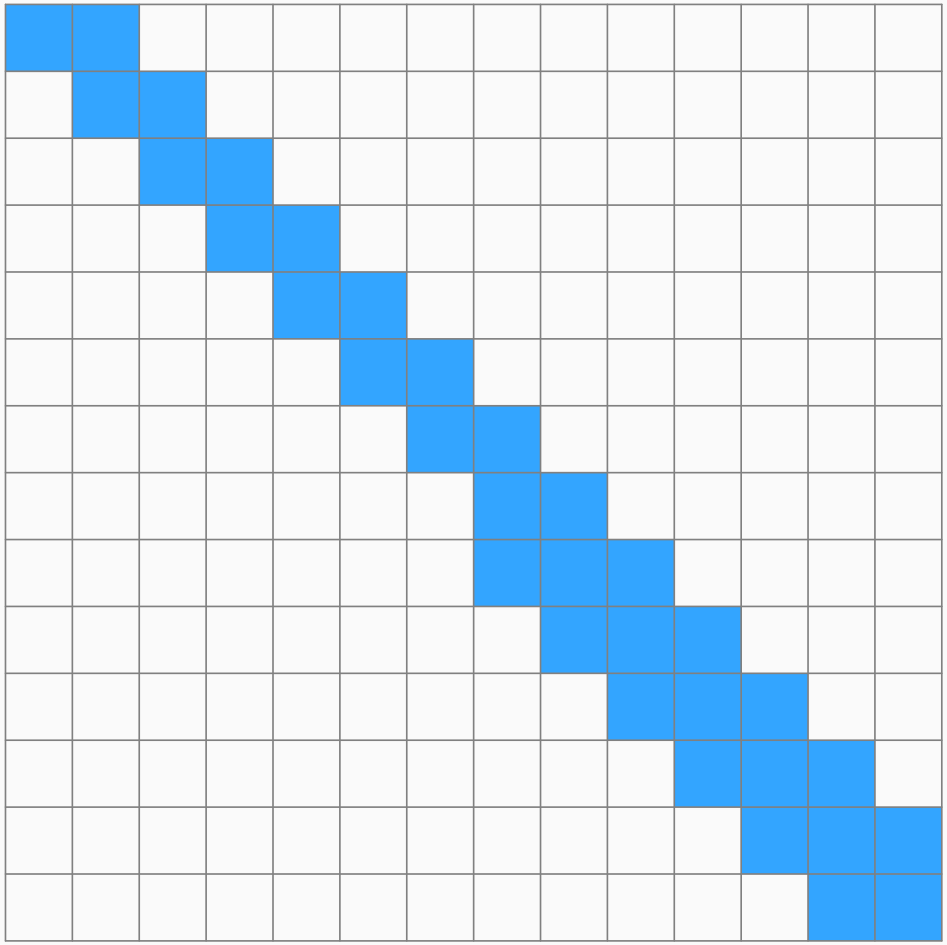


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

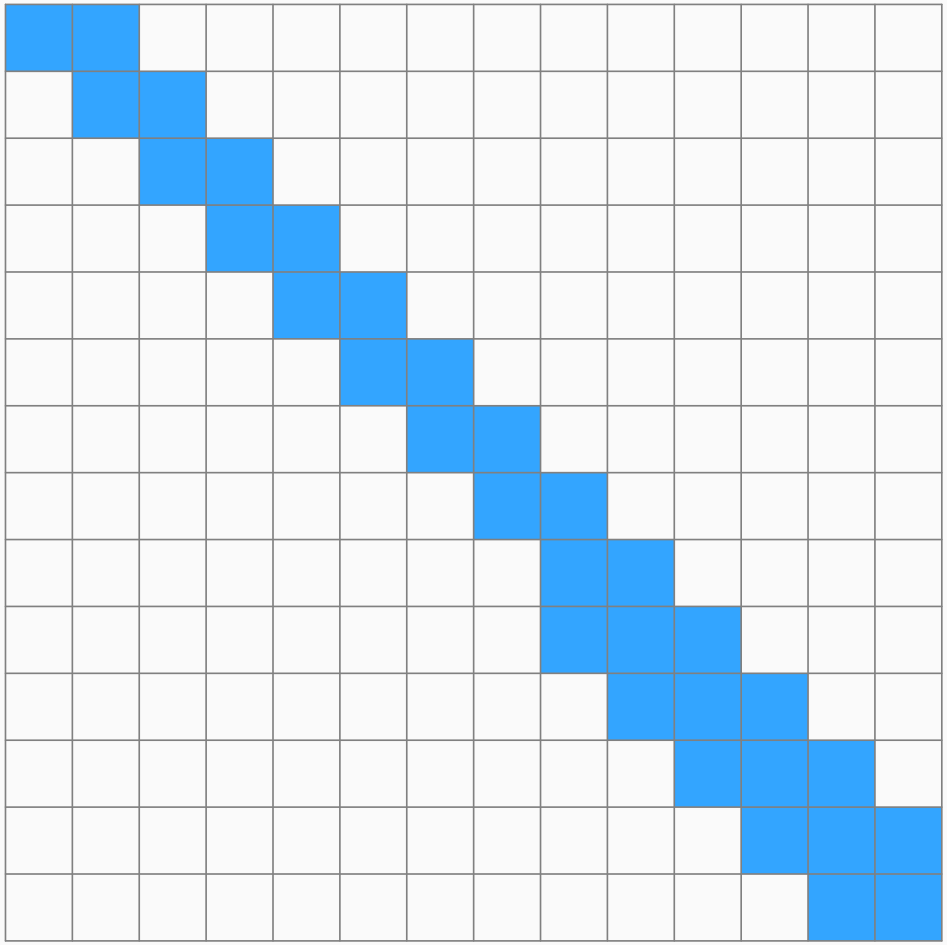


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

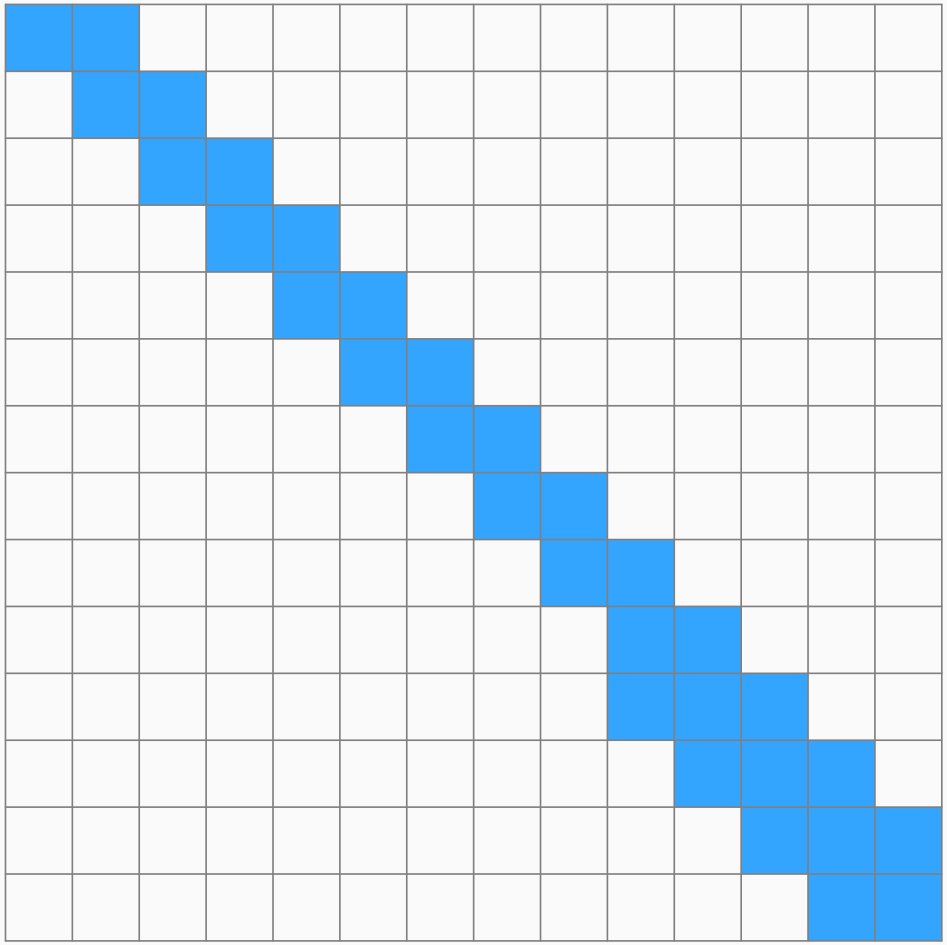


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

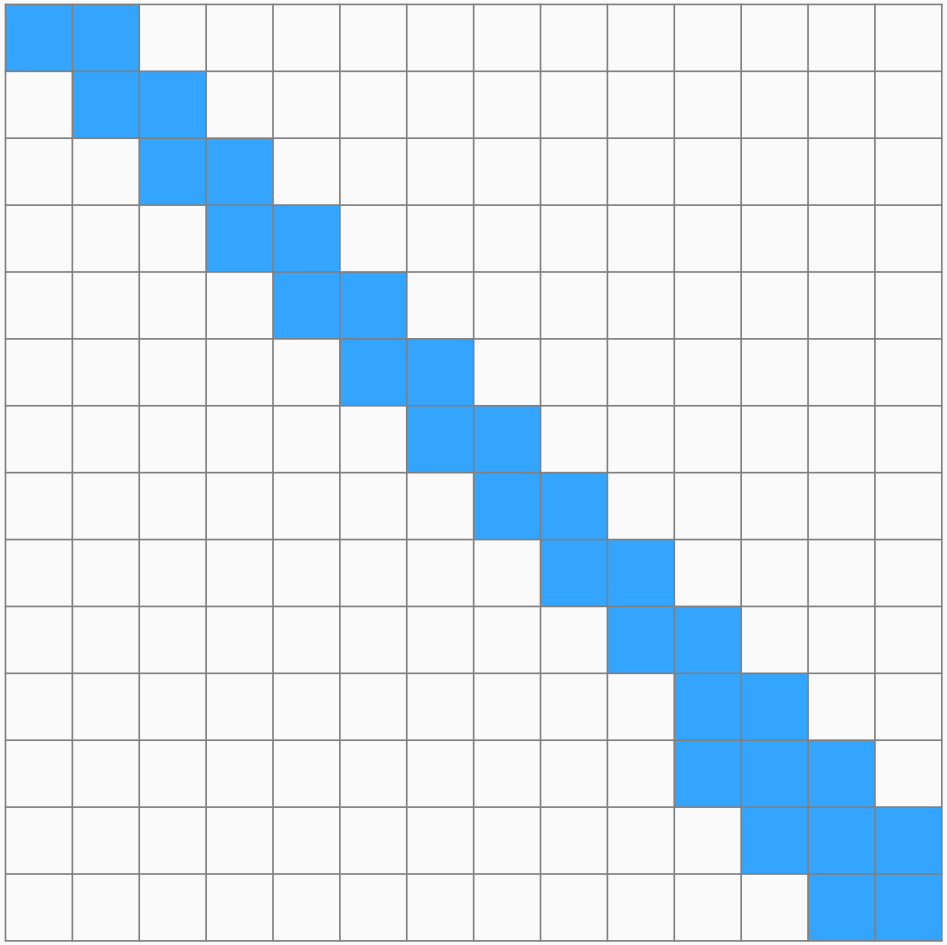


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

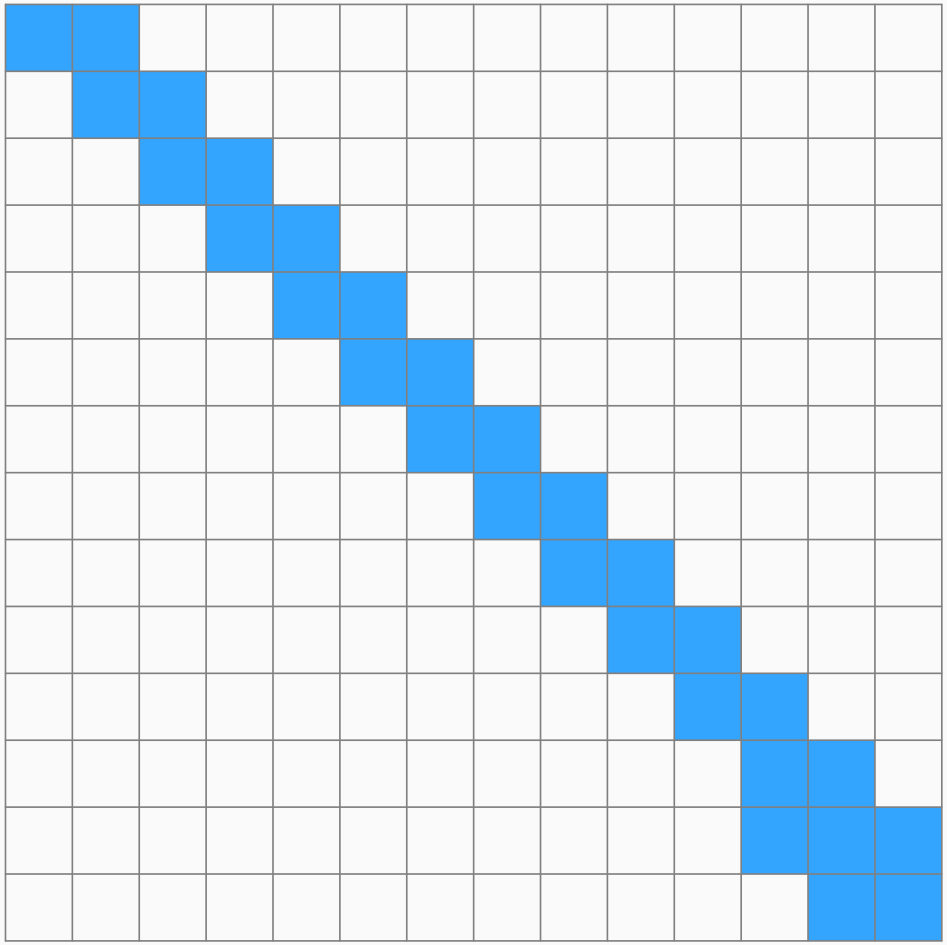


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

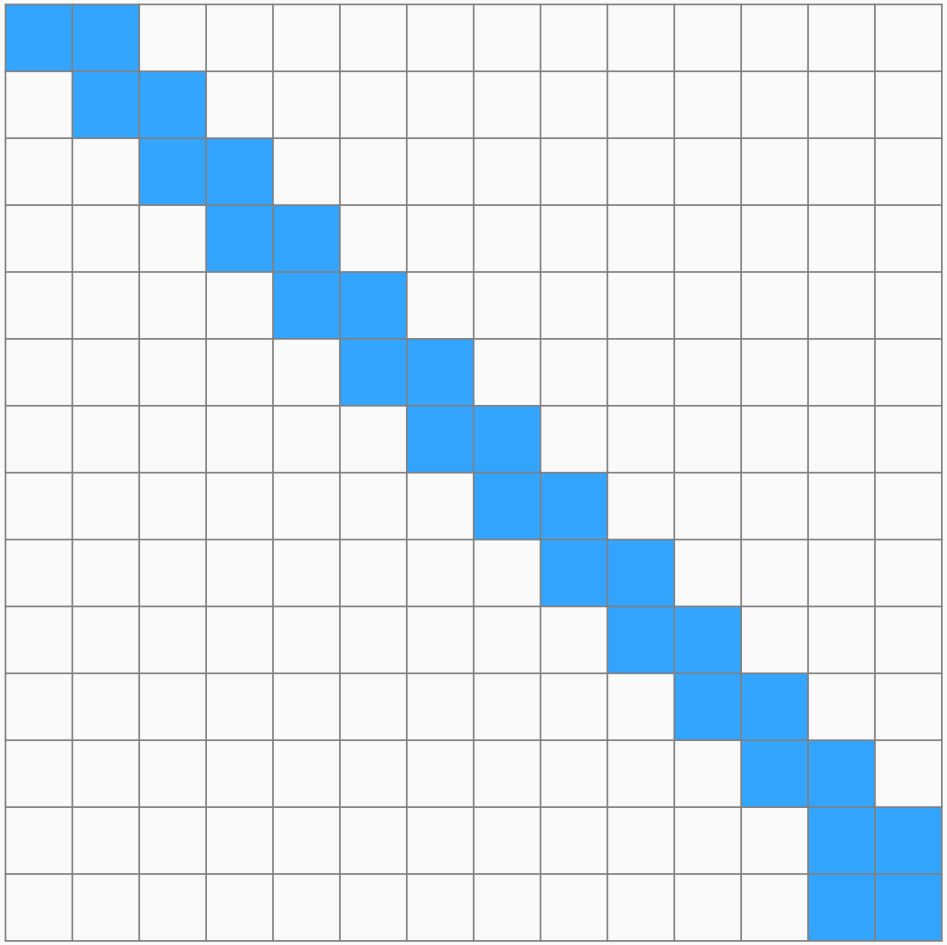


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

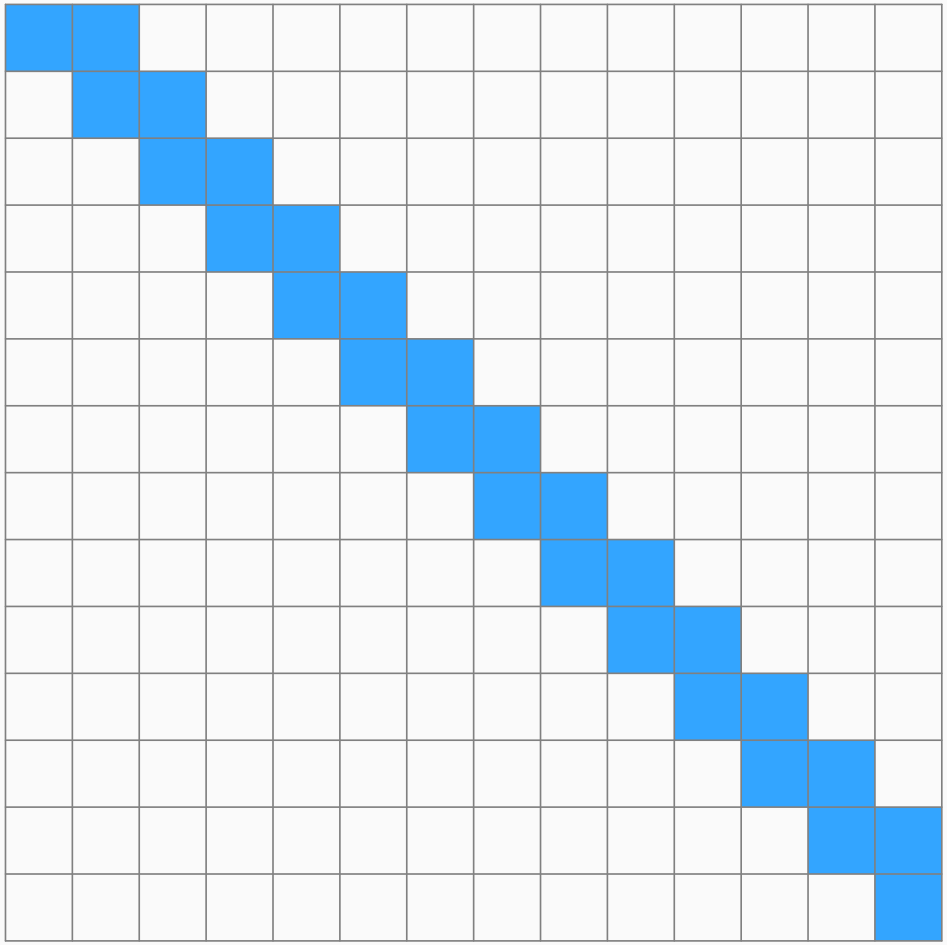


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

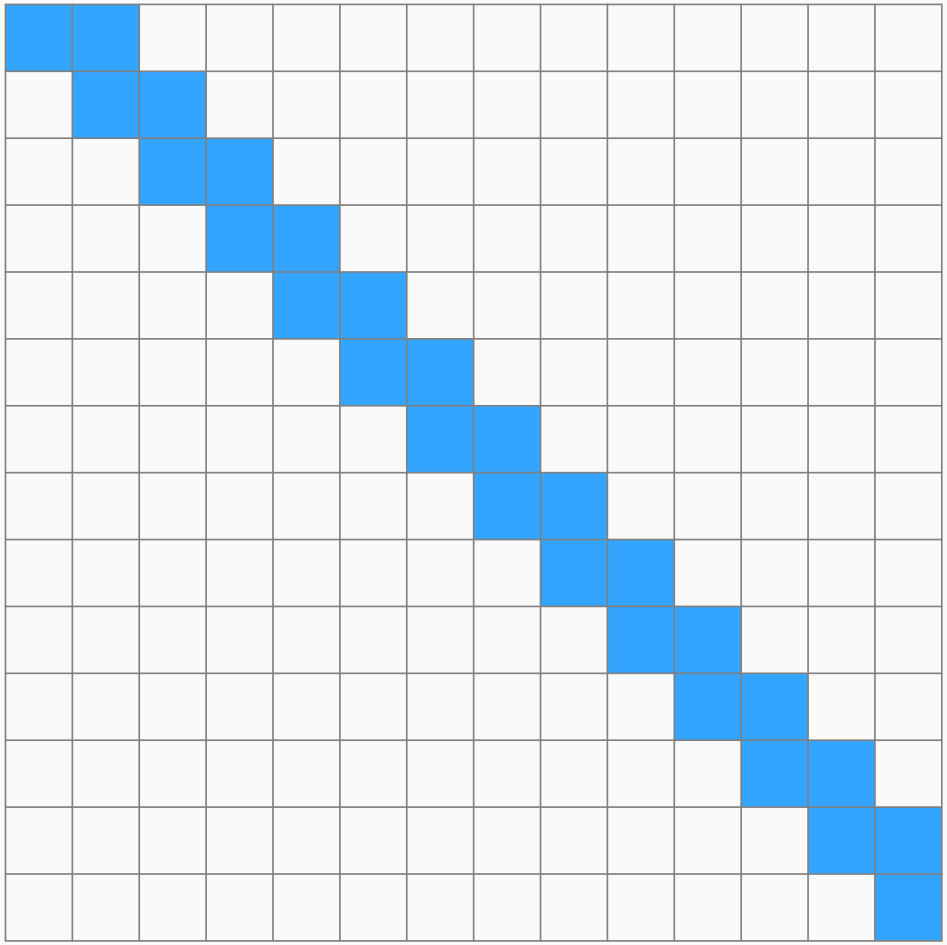


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

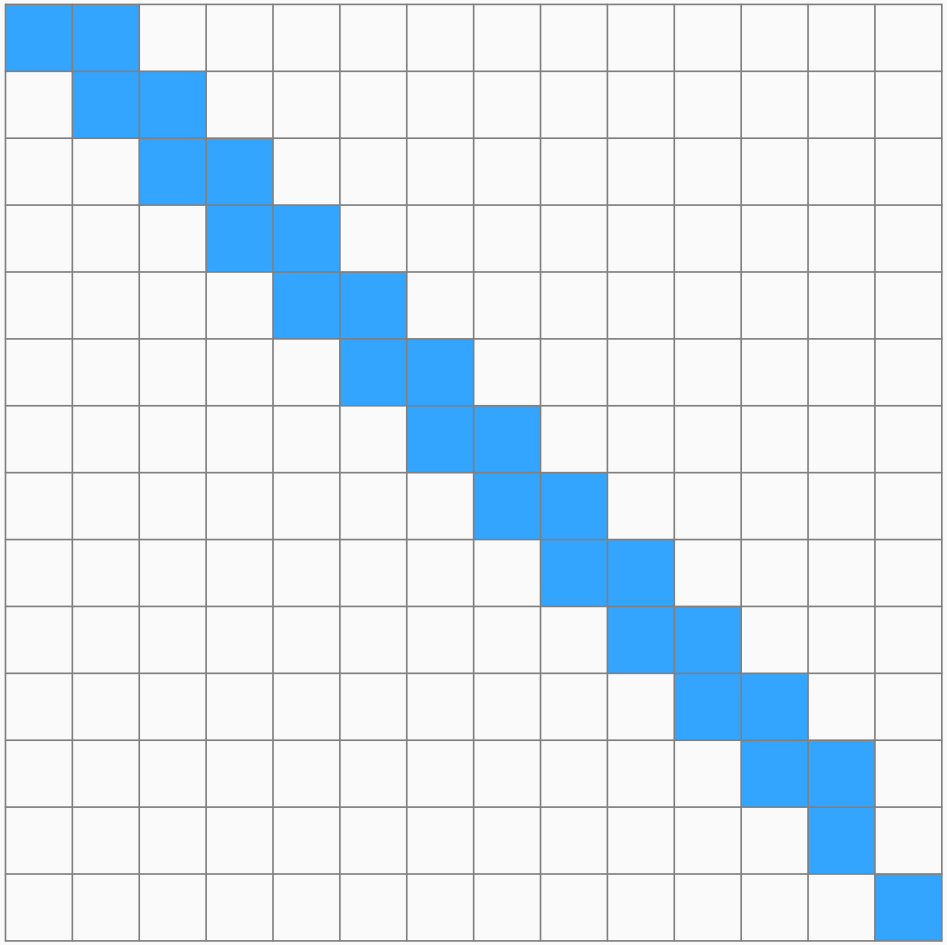


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

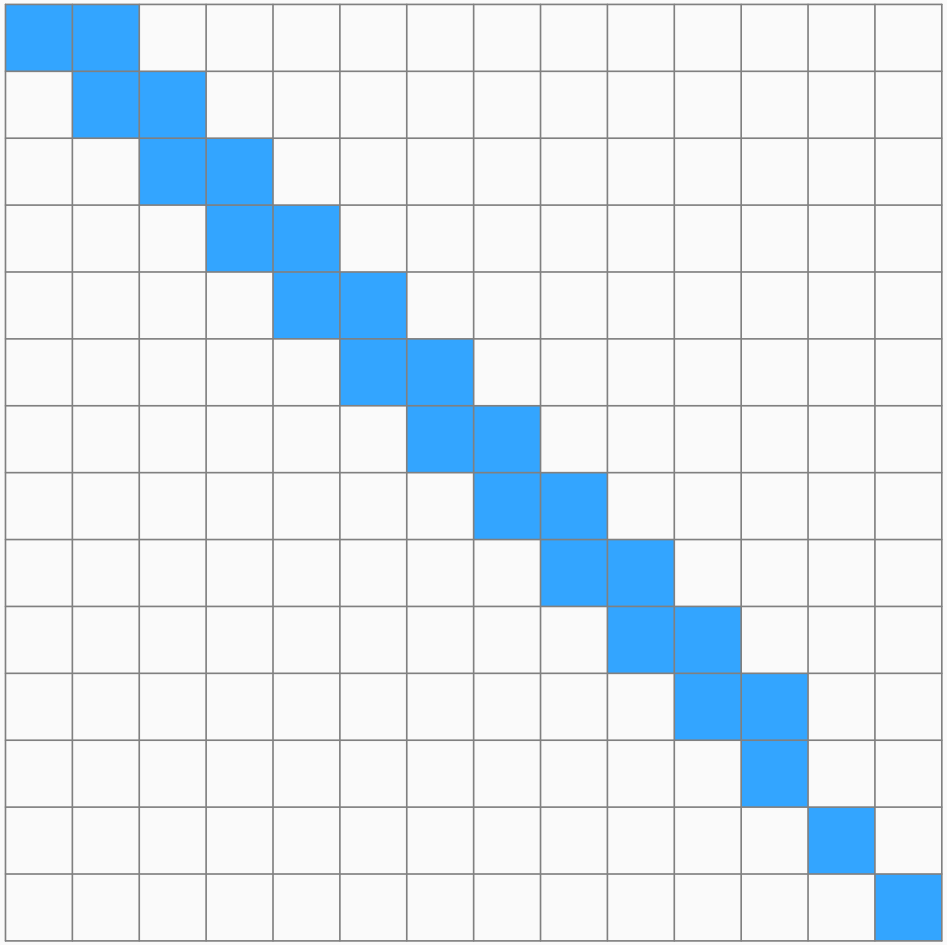


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

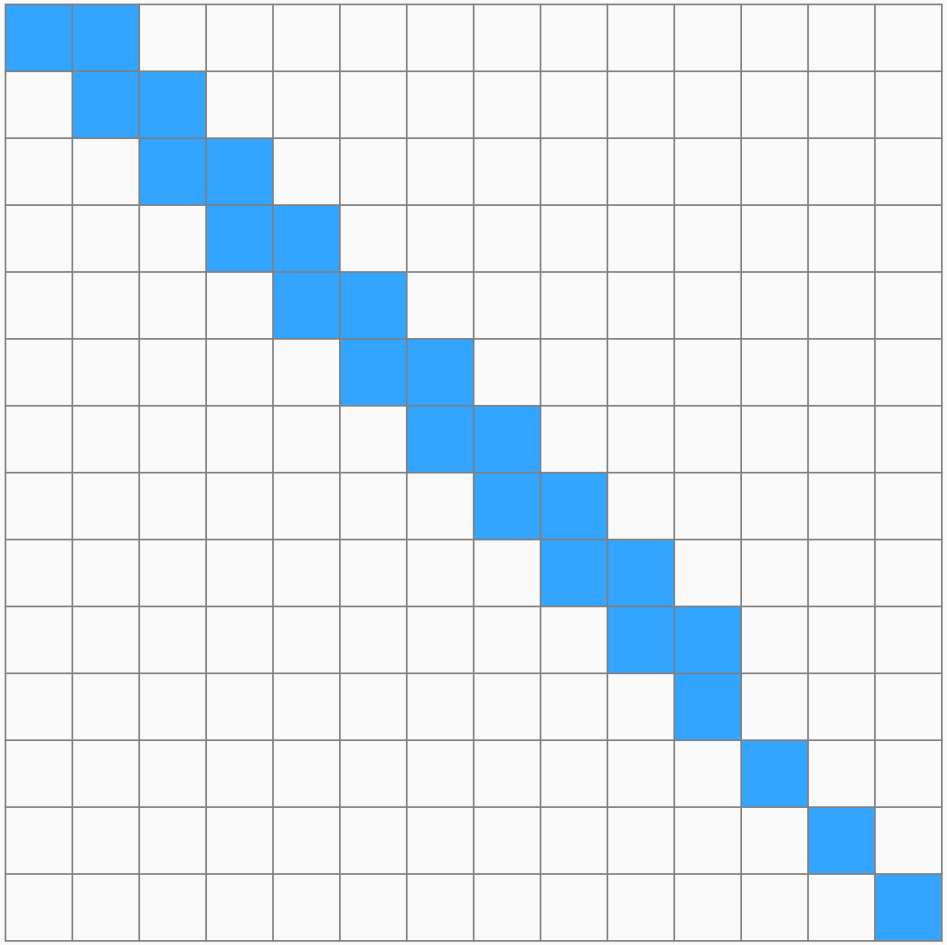


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

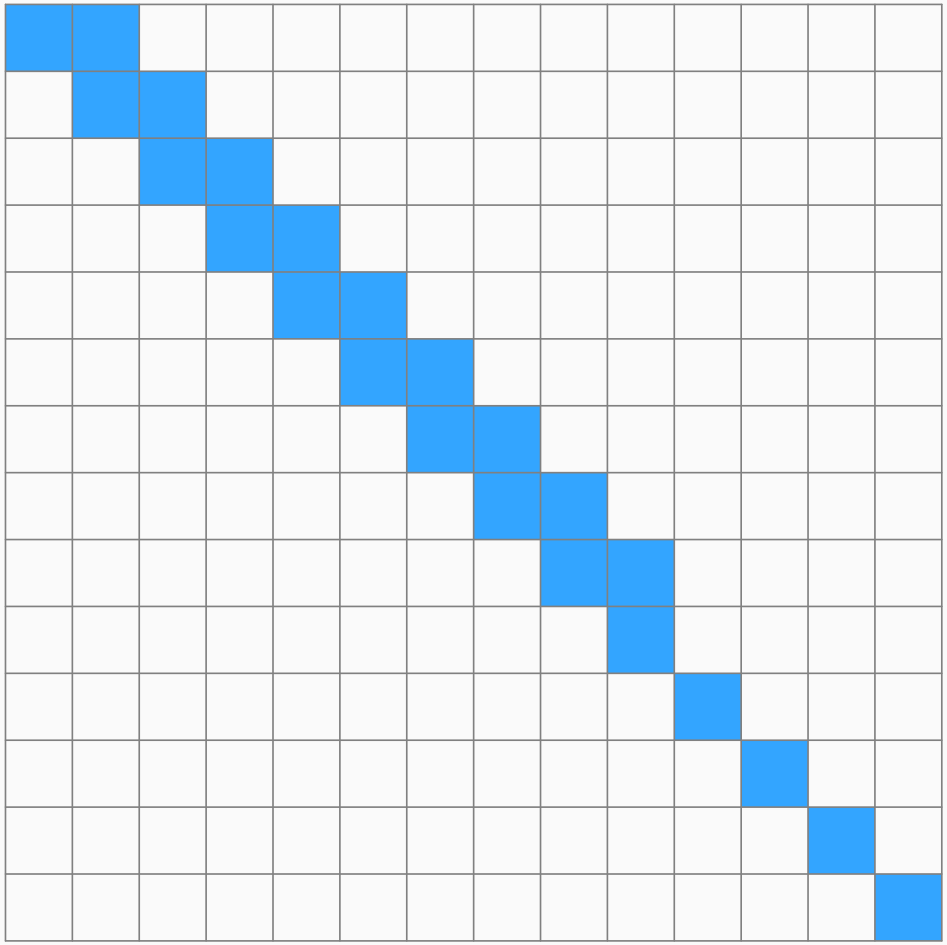


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

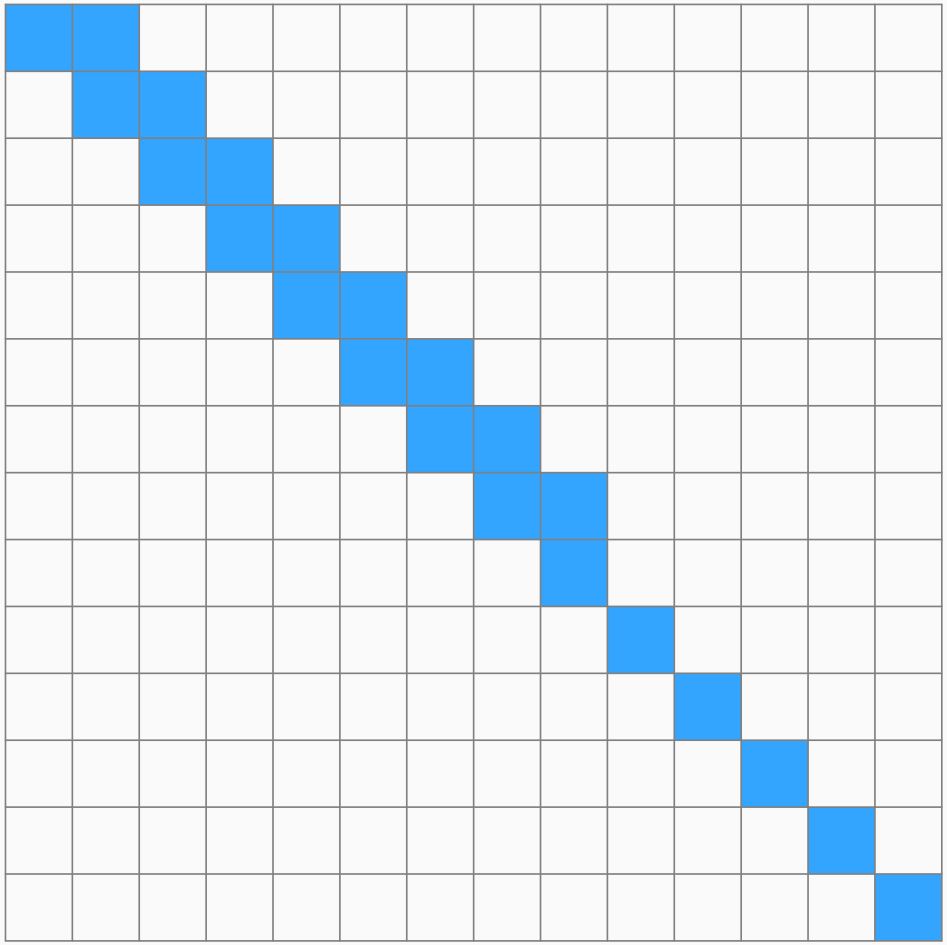


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

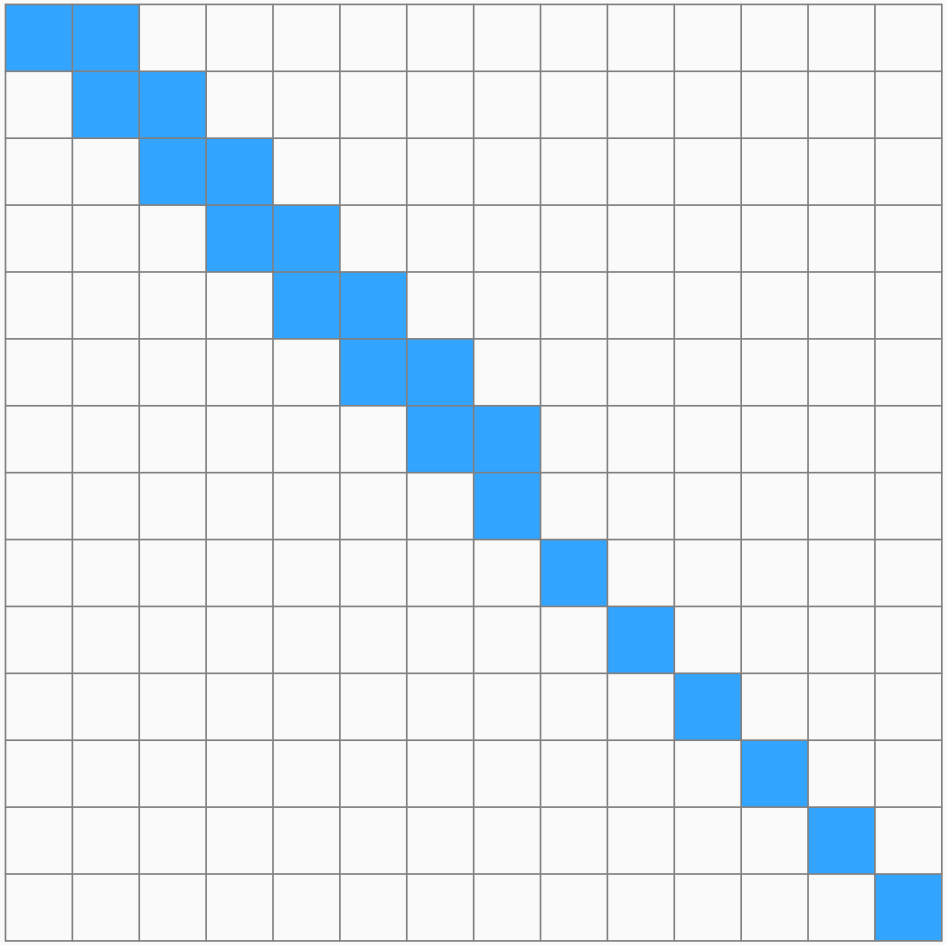


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

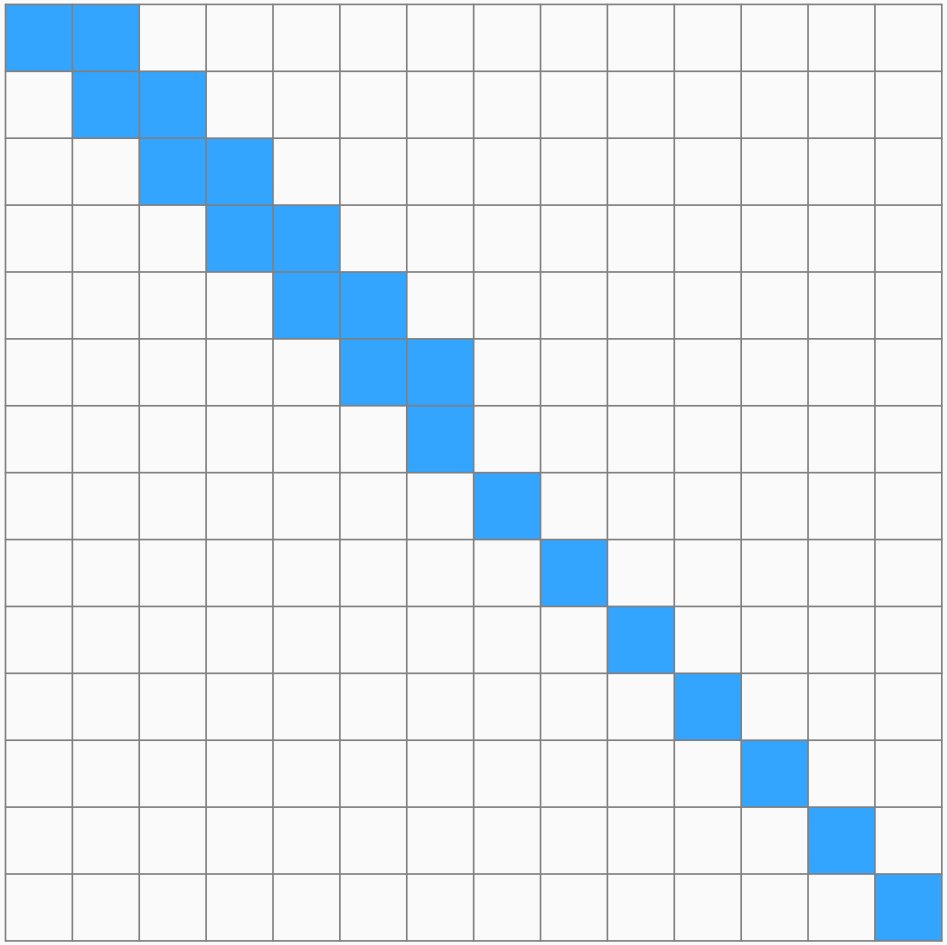


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

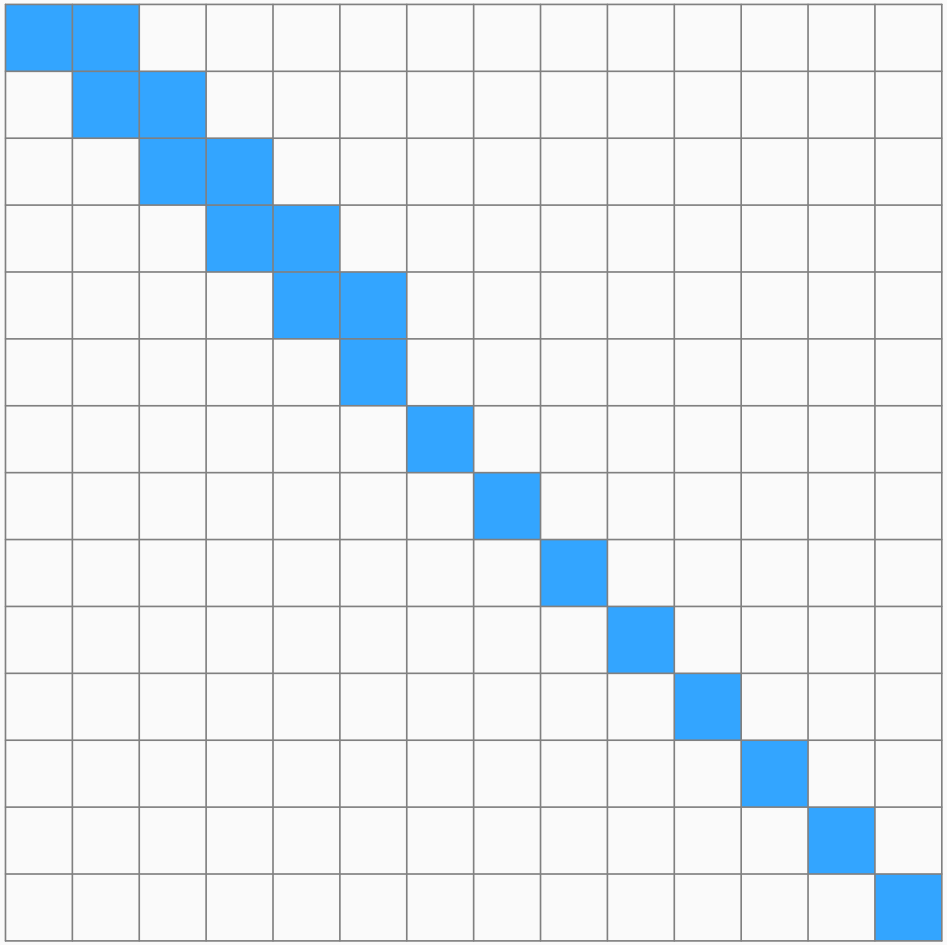


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

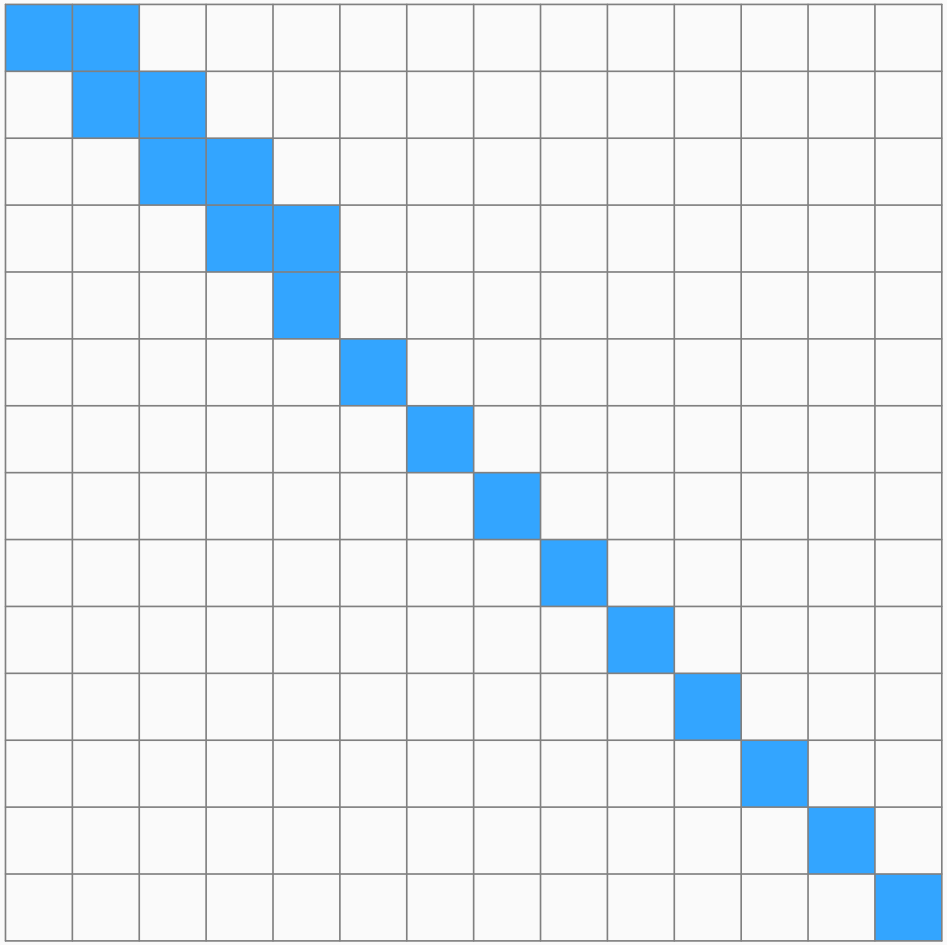


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

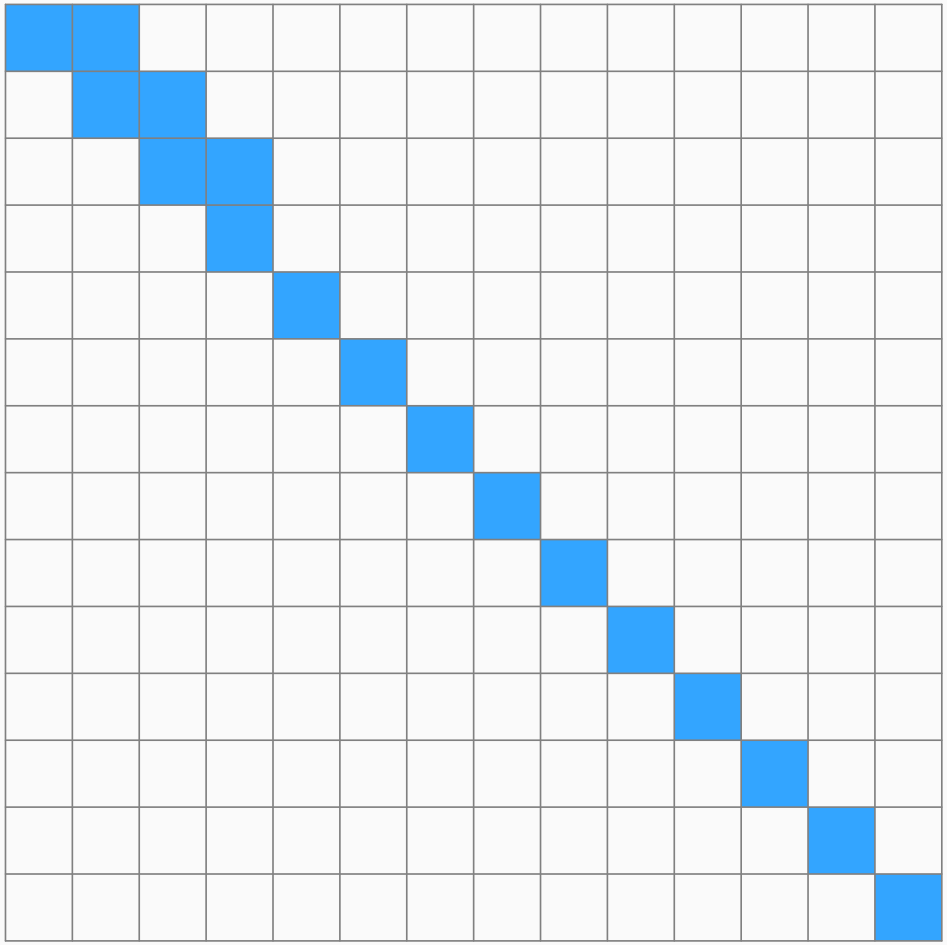


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

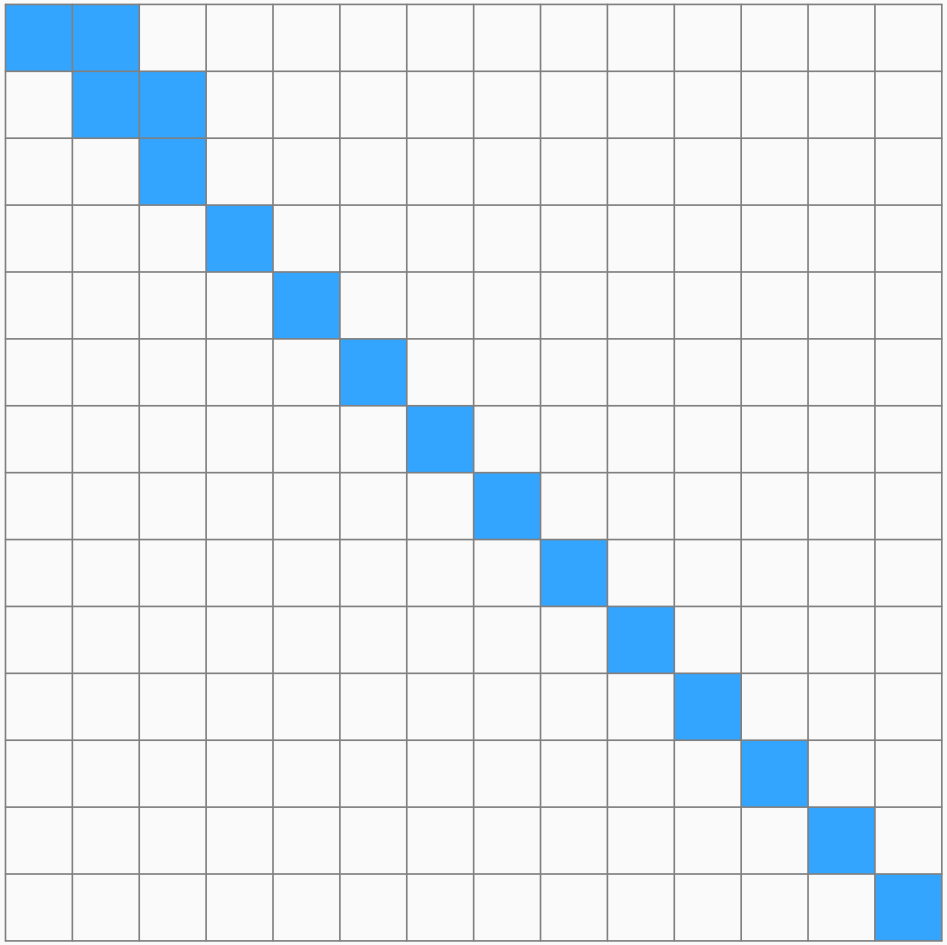


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

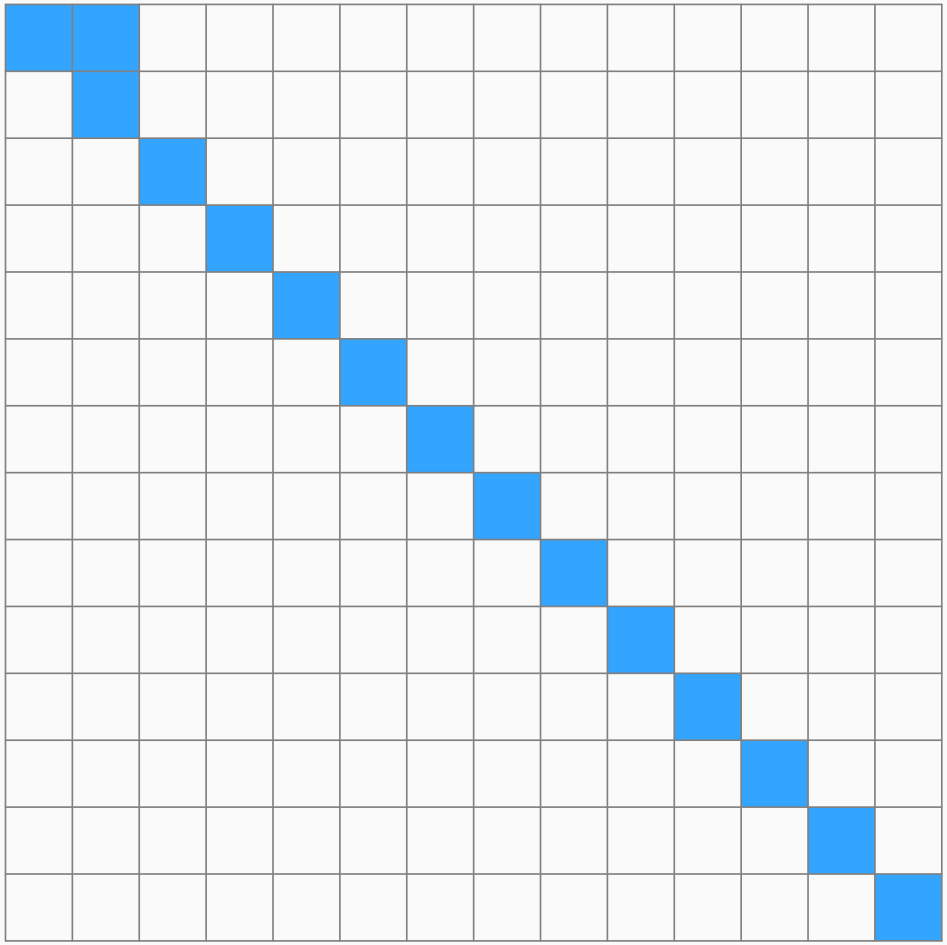


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$

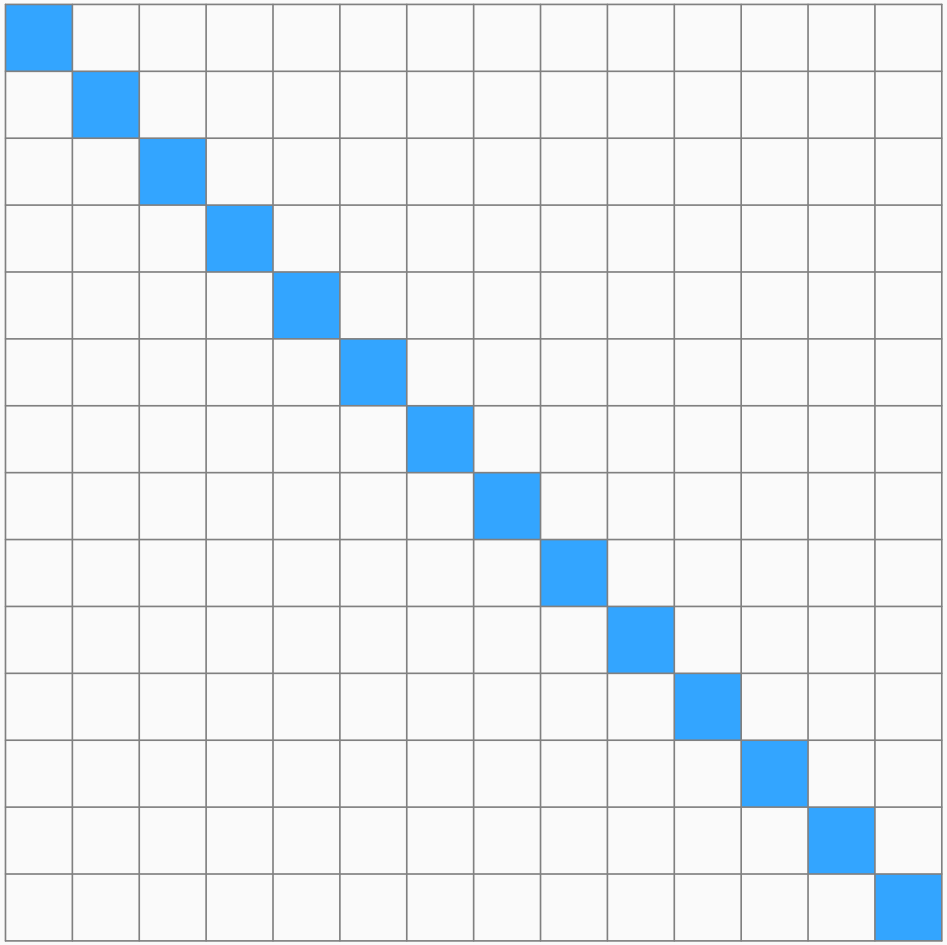


Thomas algorithm

Special case of Gaussian elimination:

$$Ax = d$$

$A =$



Contribution

Contribution

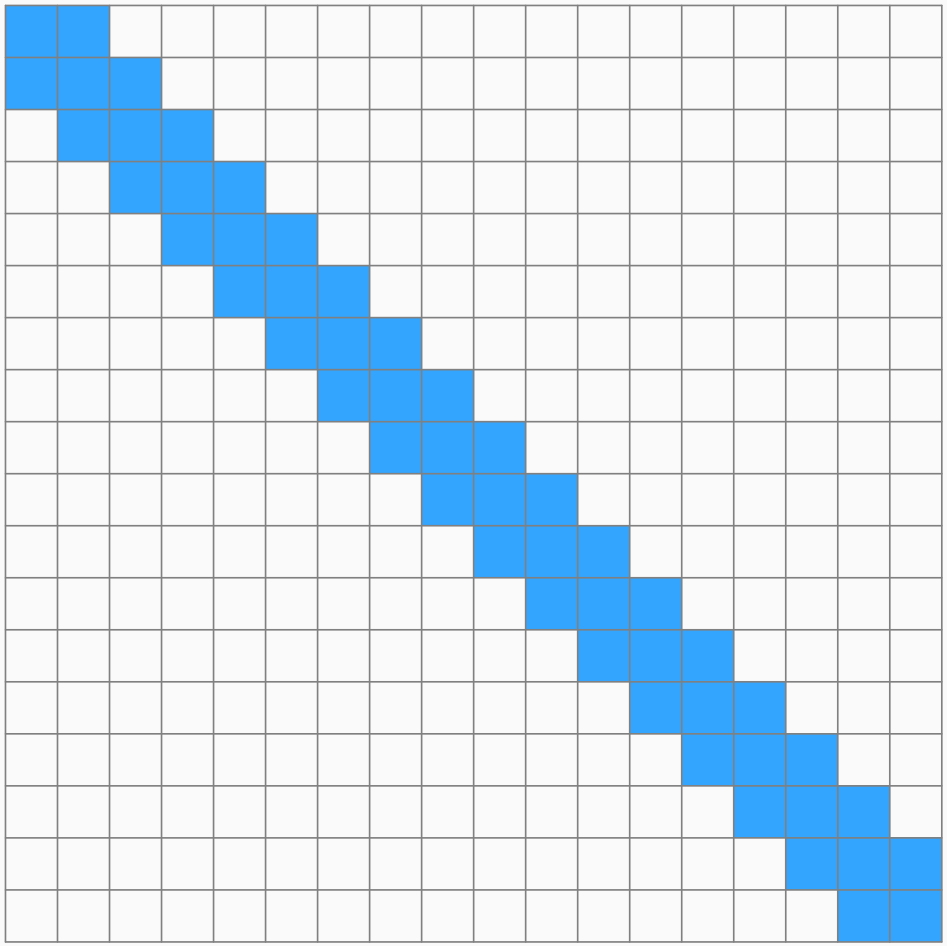
Recursive **P**artitioned **T**ridiagonal **S**chur Complement Algorithm (RPTS)

- hierarchical direct tridiagonal solver with scaled partial pivoting based on the Thomas algorithm, which runs in parallel on a GPU
- able to solve systems which exceed the shared memory of one block

Proposed algorithm

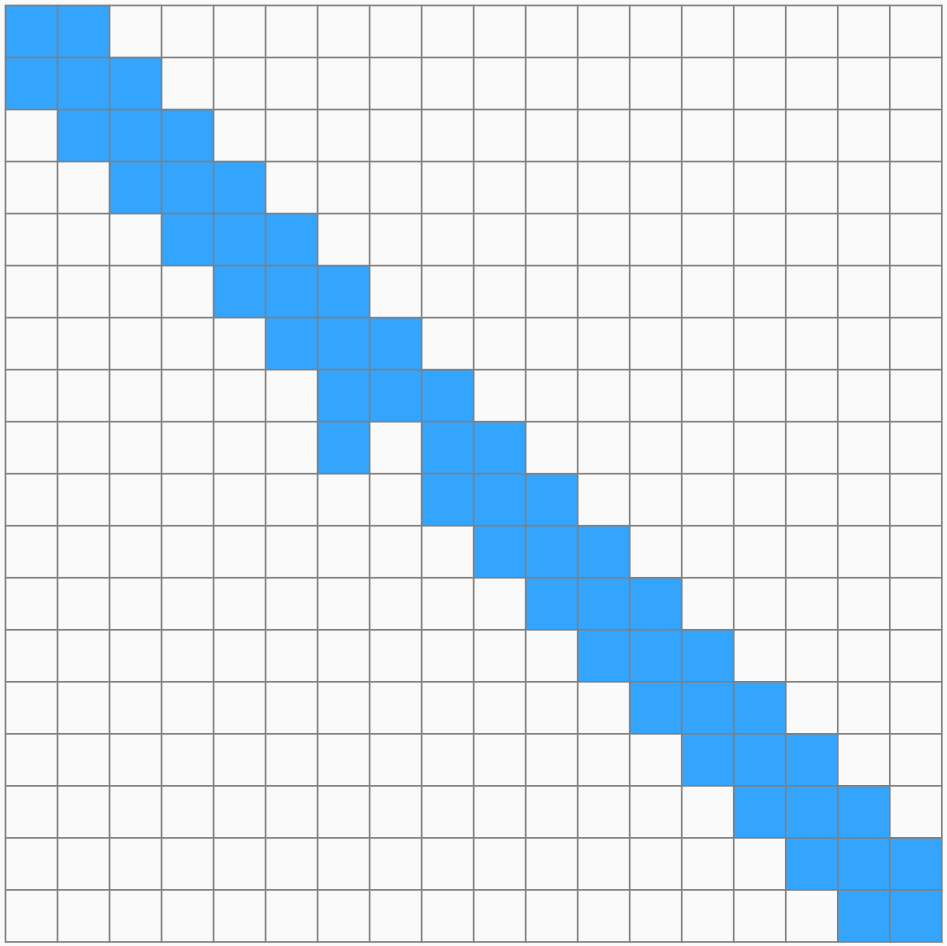
One thread

$A =$



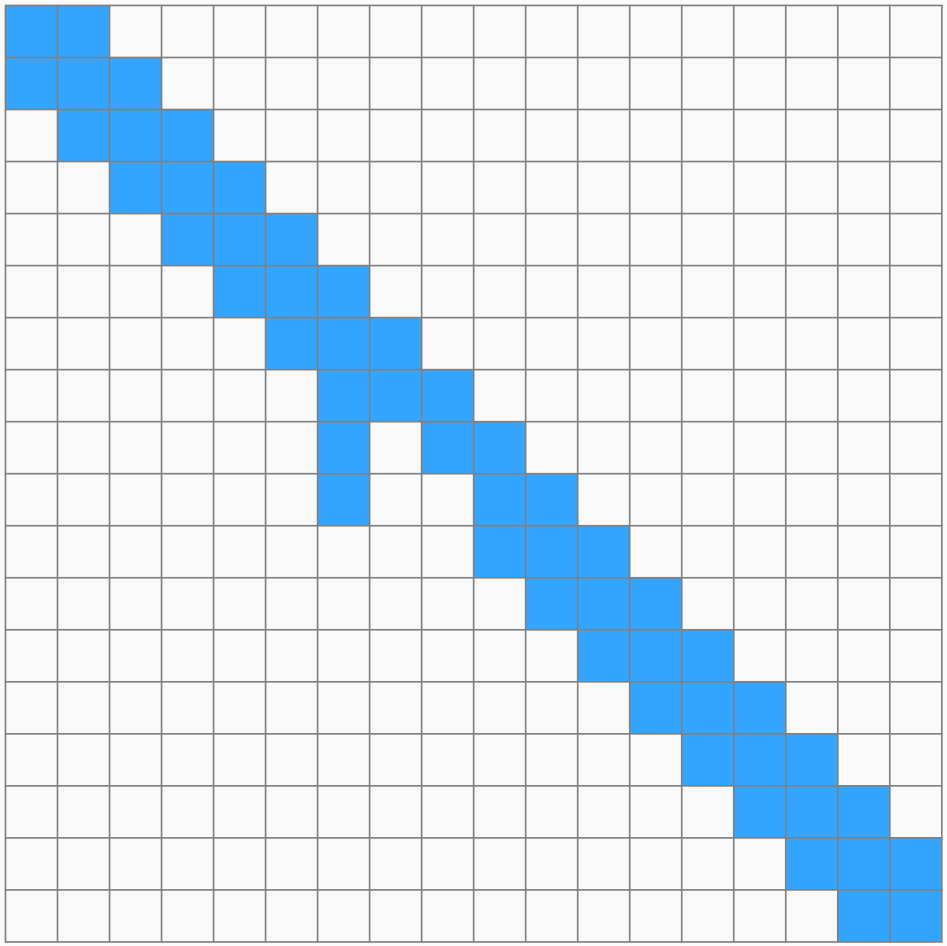
One thread

$A =$



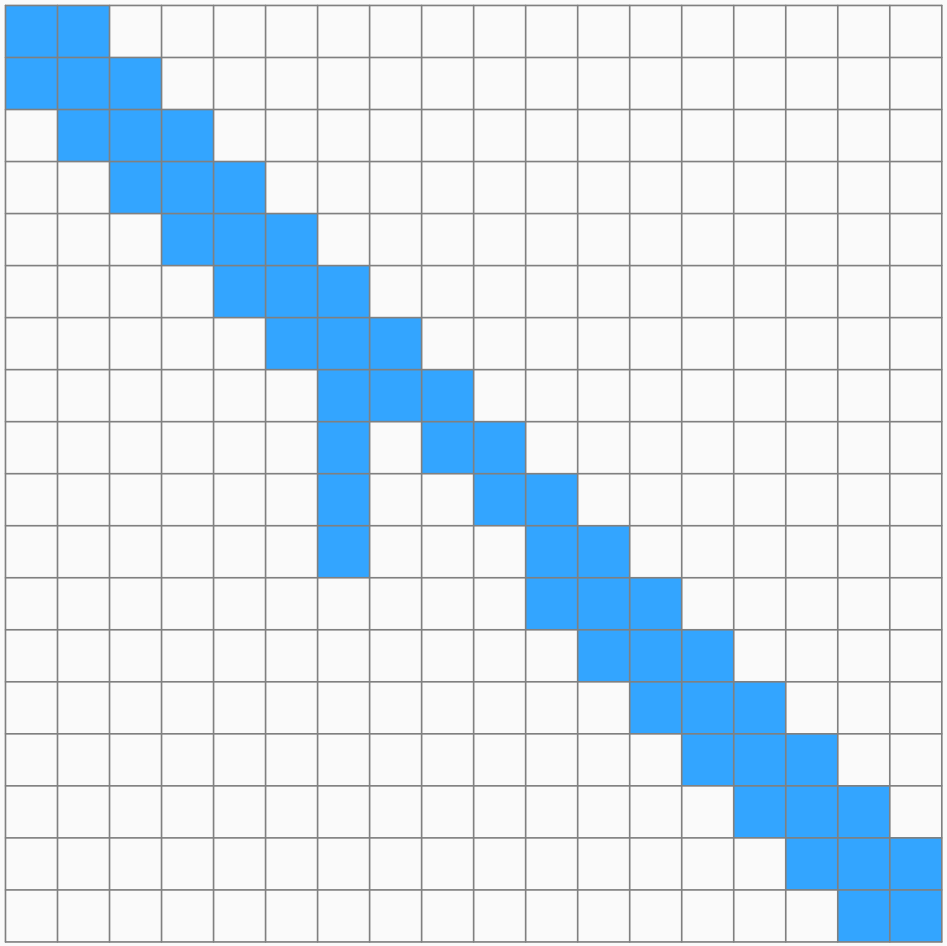
One thread

$A =$



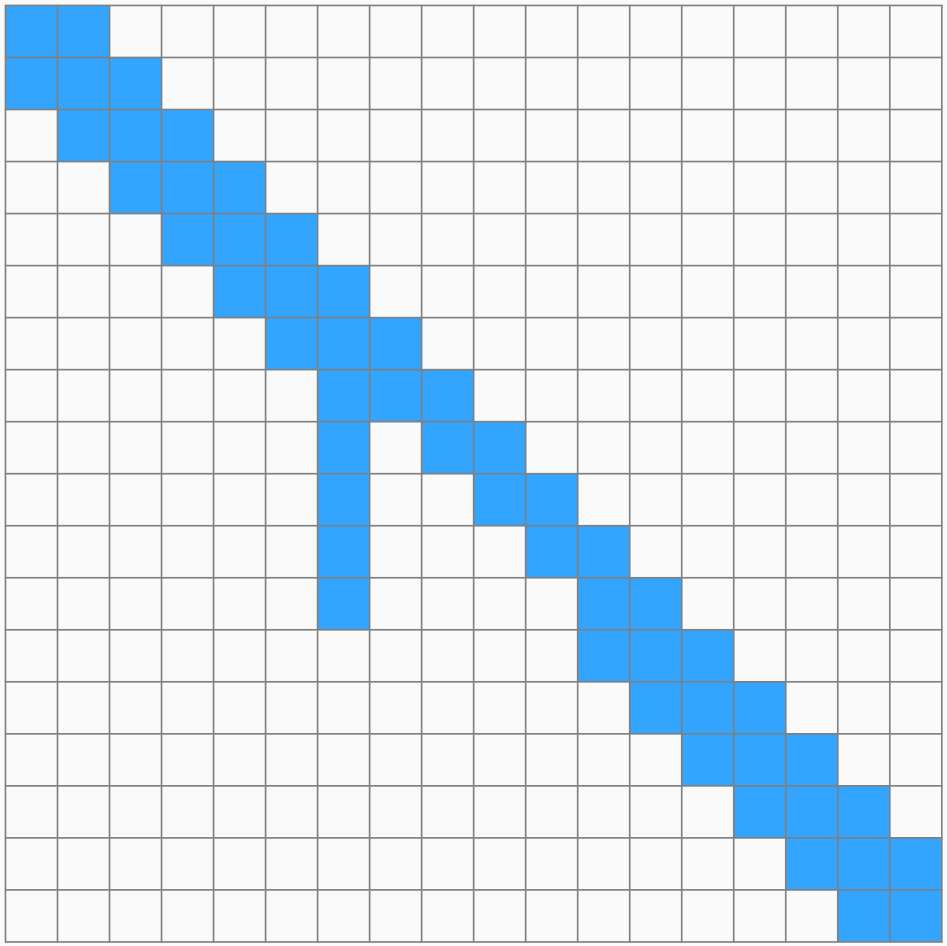
One thread

$A =$



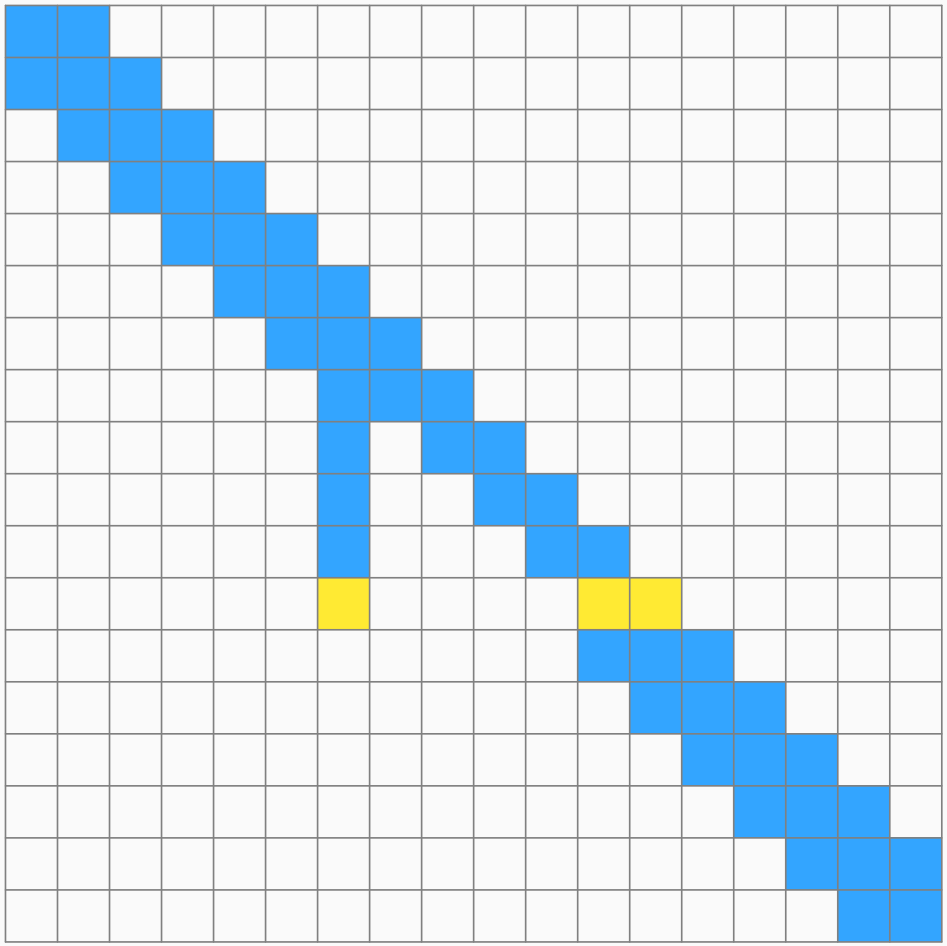
One thread

$A =$



One thread

$A =$

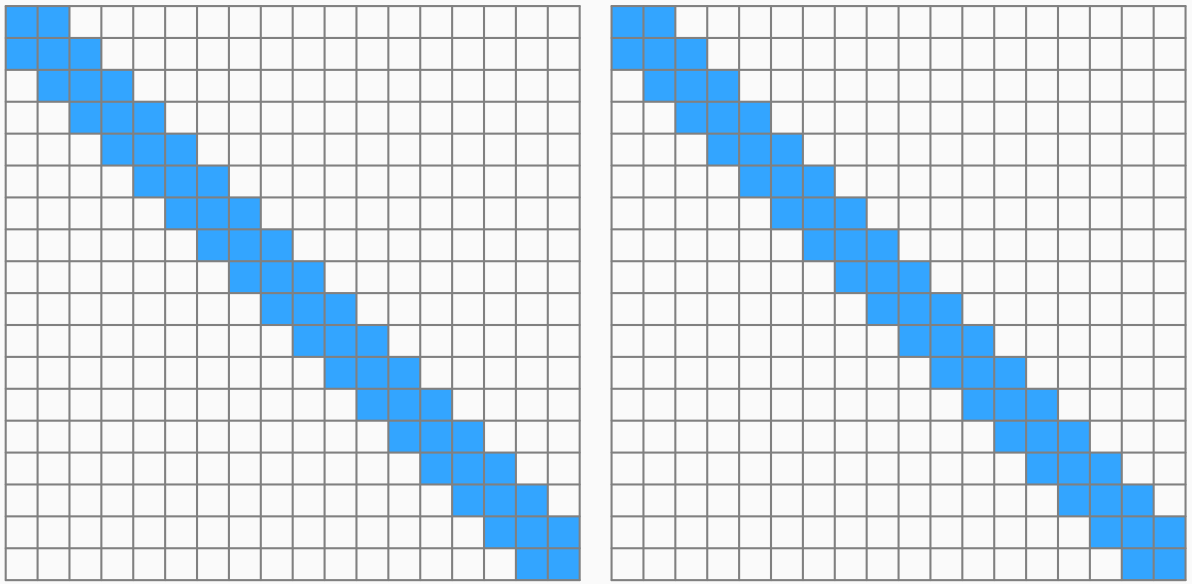


All threads

downwards elimination

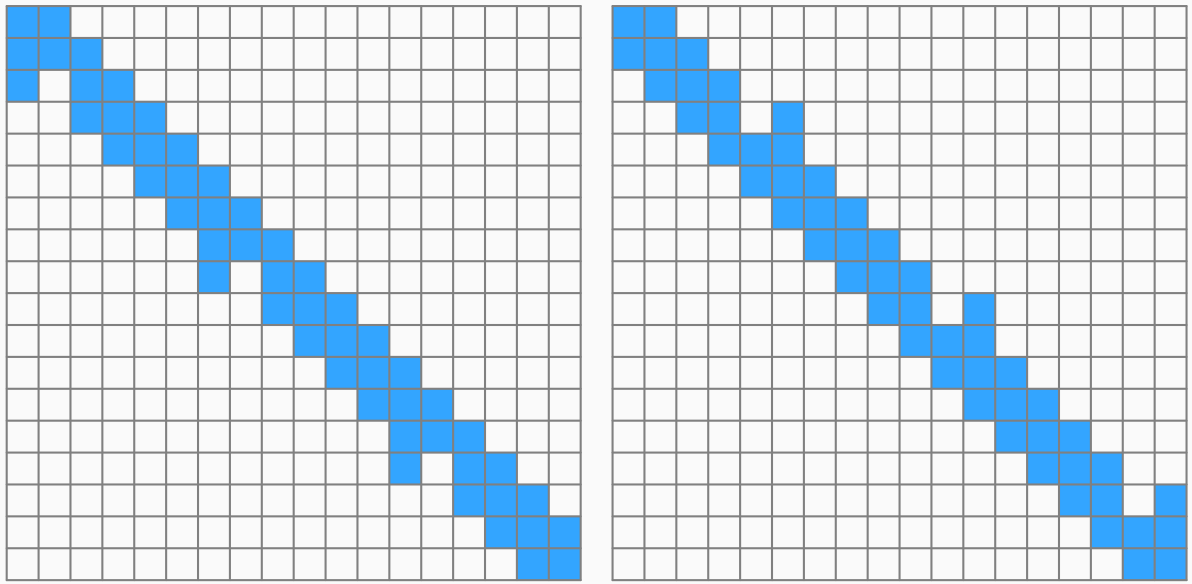
upwards elimination

$A =$



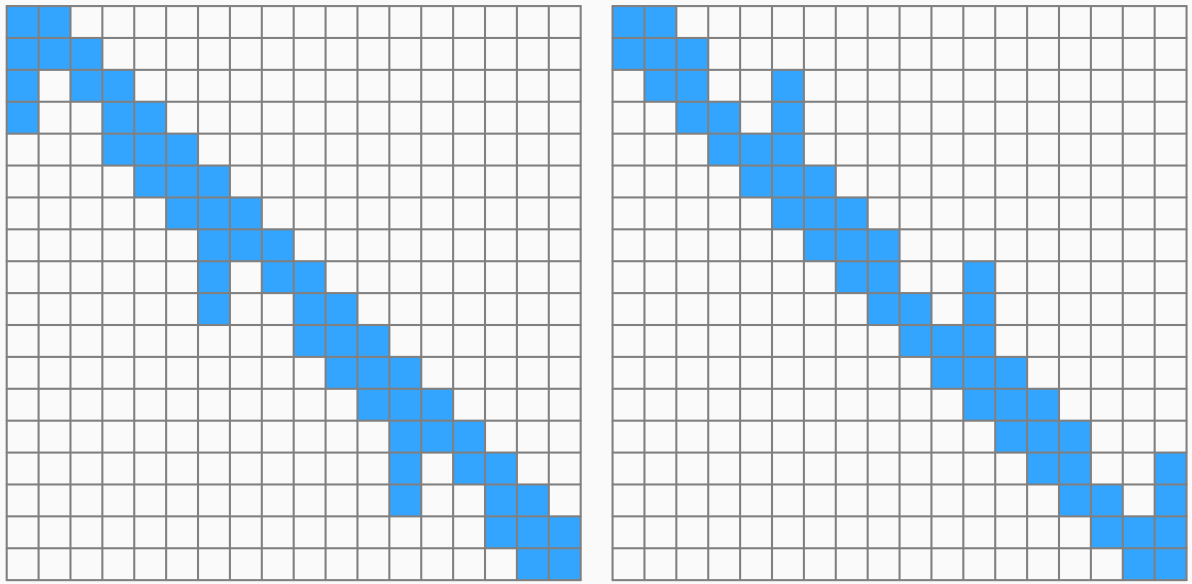
All threads

$A =$



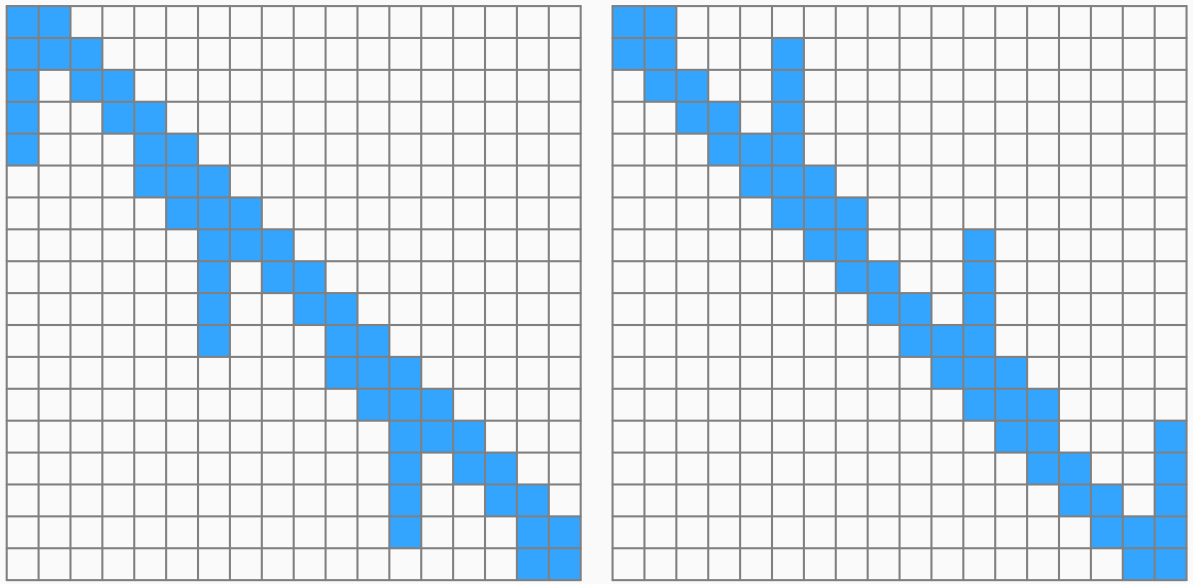
All threads

$A =$



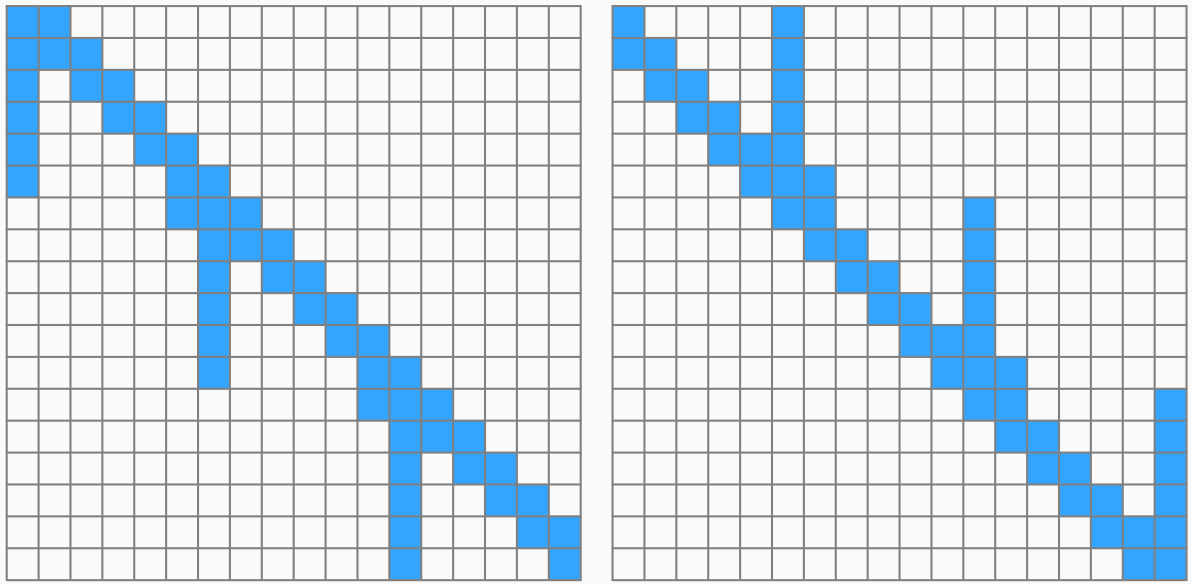
All threads

$A =$



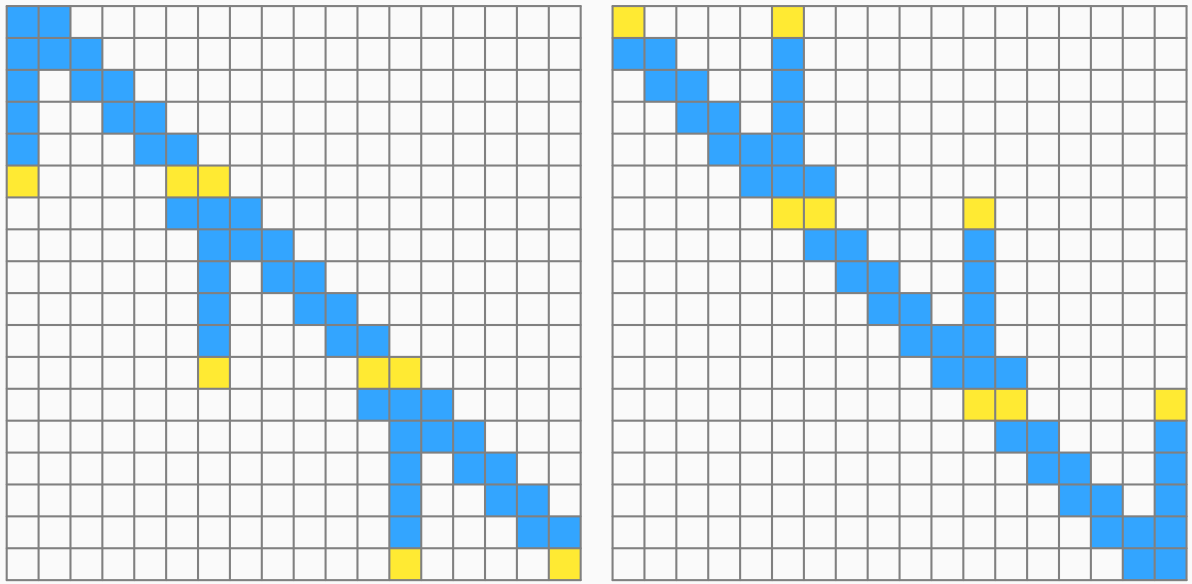
All threads

$A =$

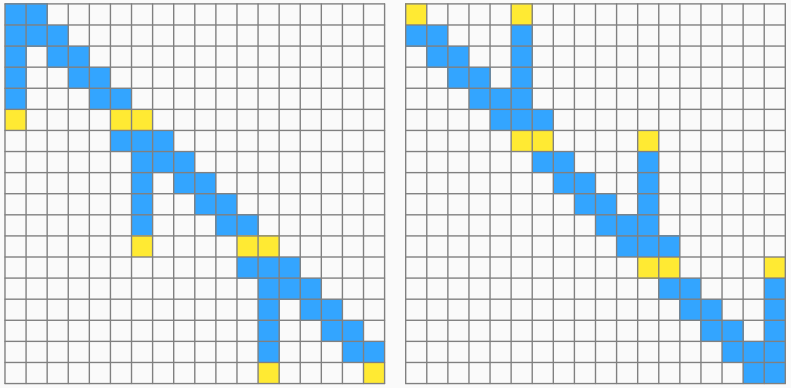
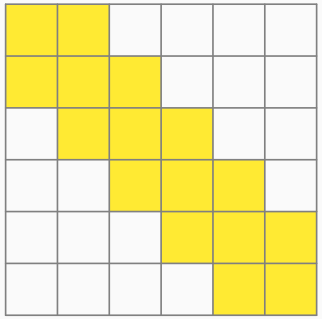


All threads

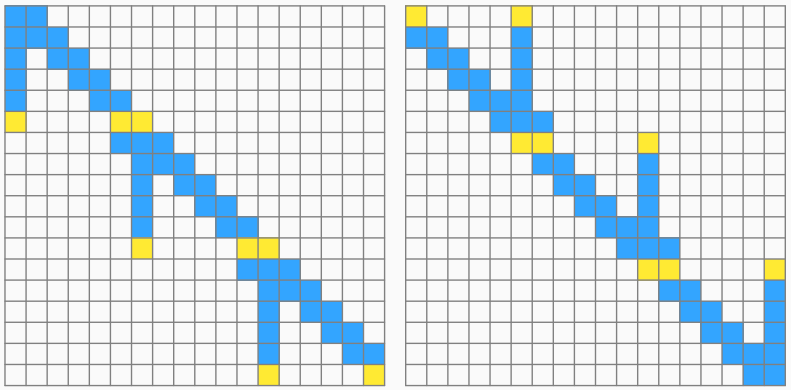
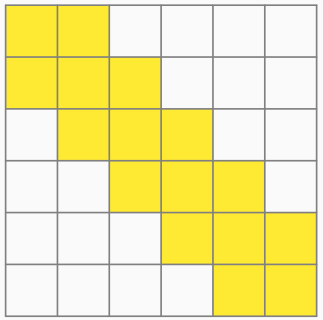
$A =$



All threads

$A =$  $\longrightarrow$ $A' =$ 

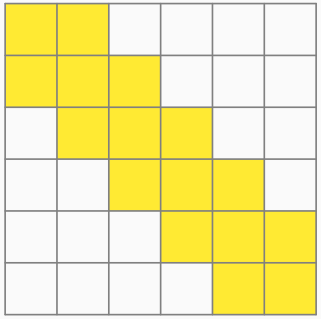
All threads

$A =$  $\longrightarrow$ $A' =$ 

apply recursively until system is small enough

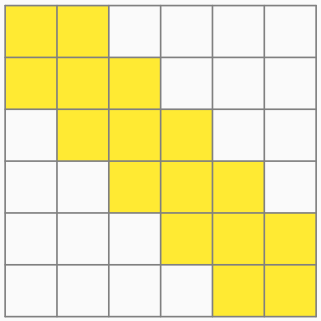
Smaller tridiagonal system

$A' =$

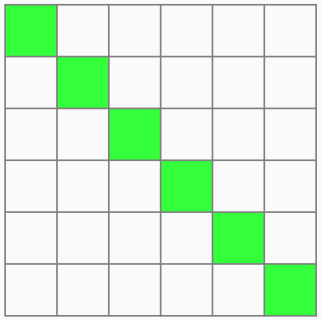


Smaller tridiagonal system

$A' =$

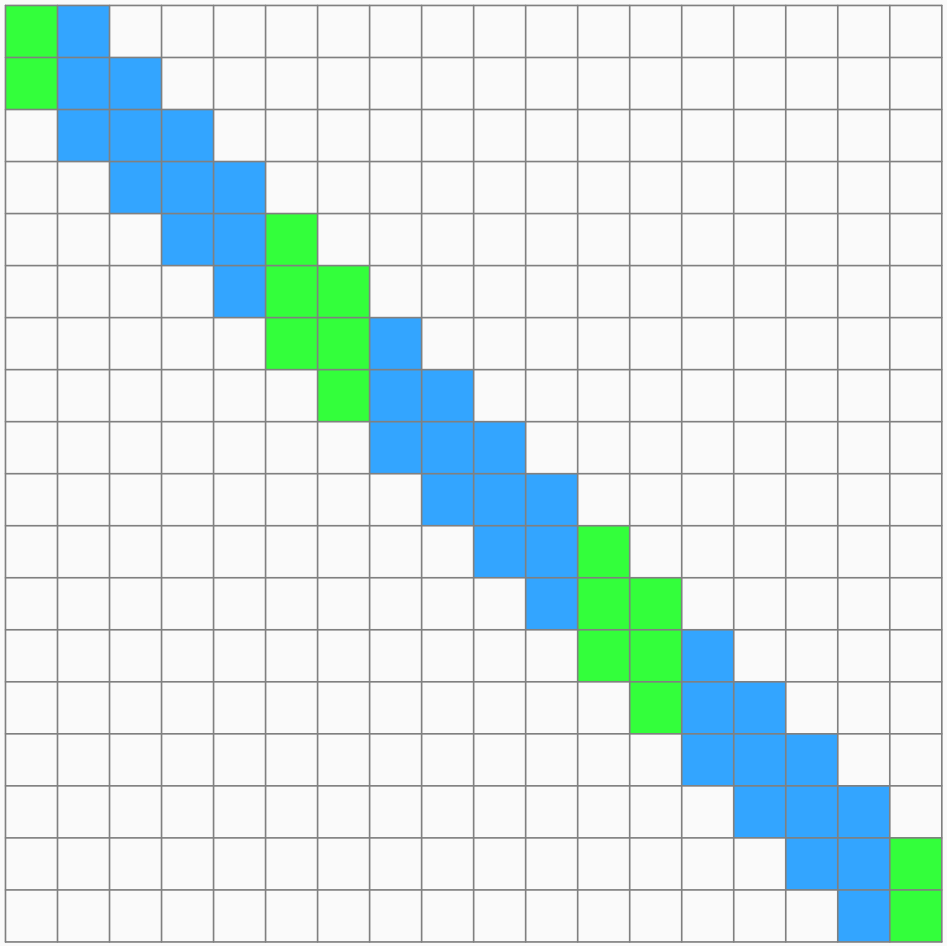


solution $=$



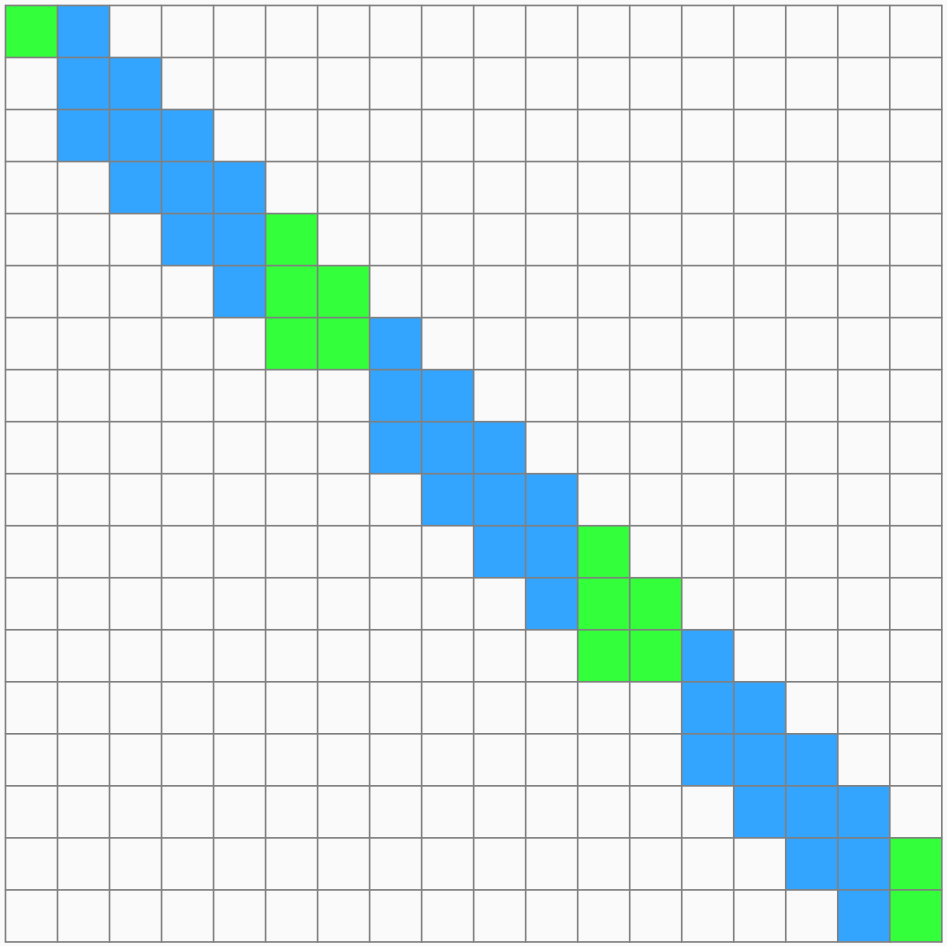
All threads use coarse solution

$A =$



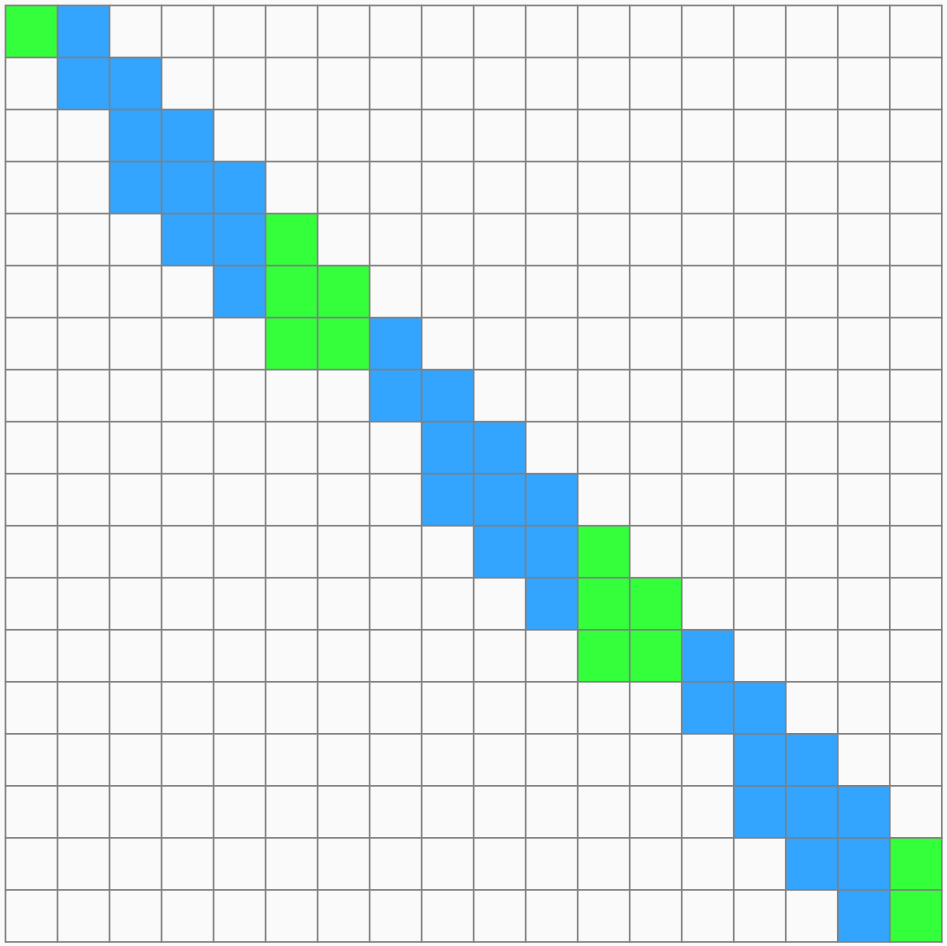
All threads use coarse solution

$A =$



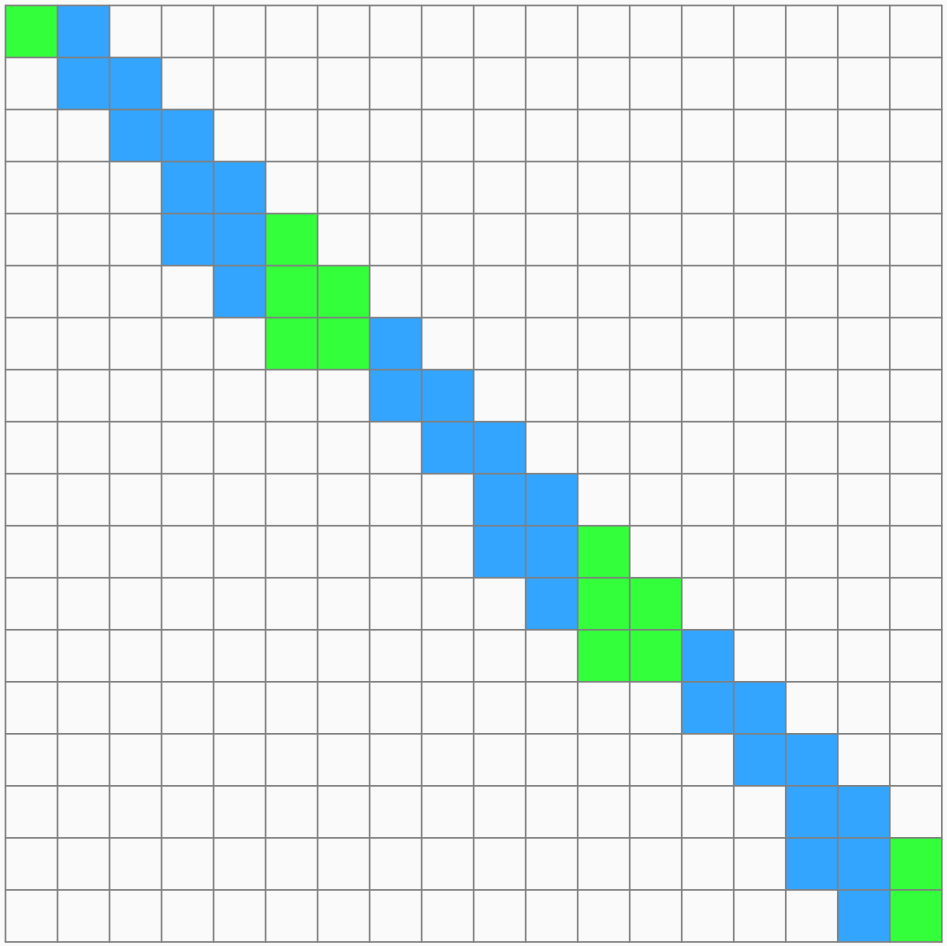
All threads use coarse solution

$A =$



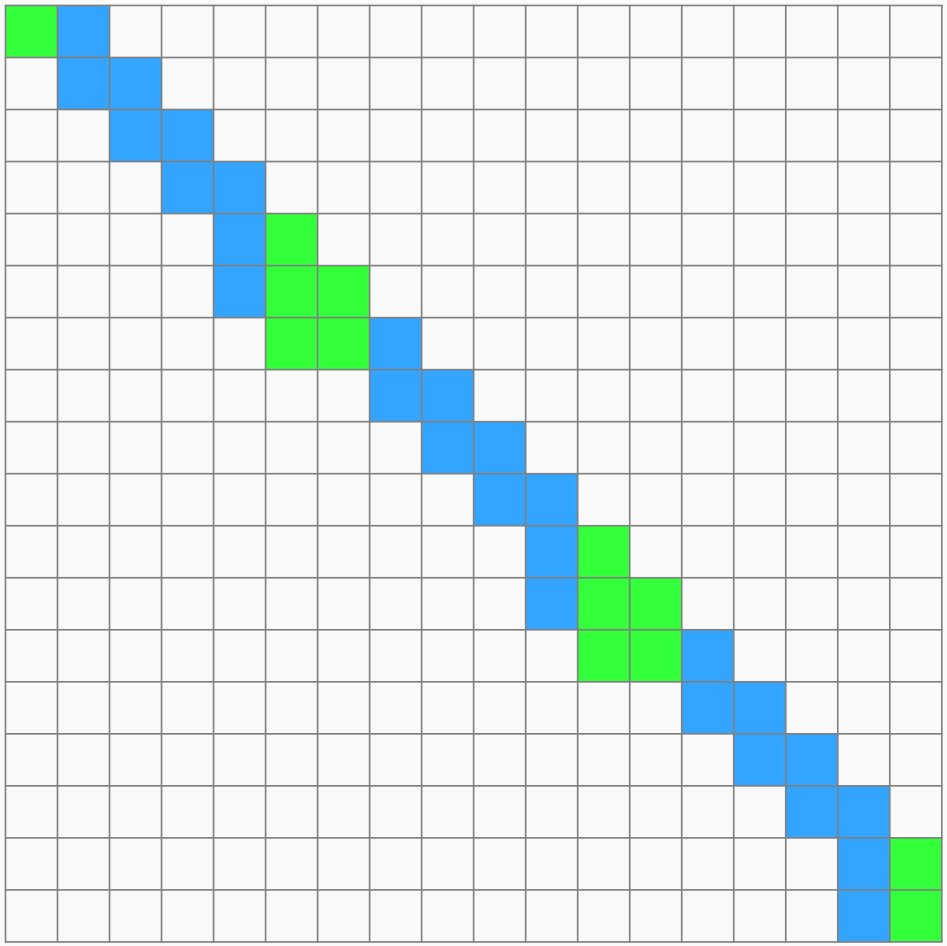
All threads use coarse solution

$A =$



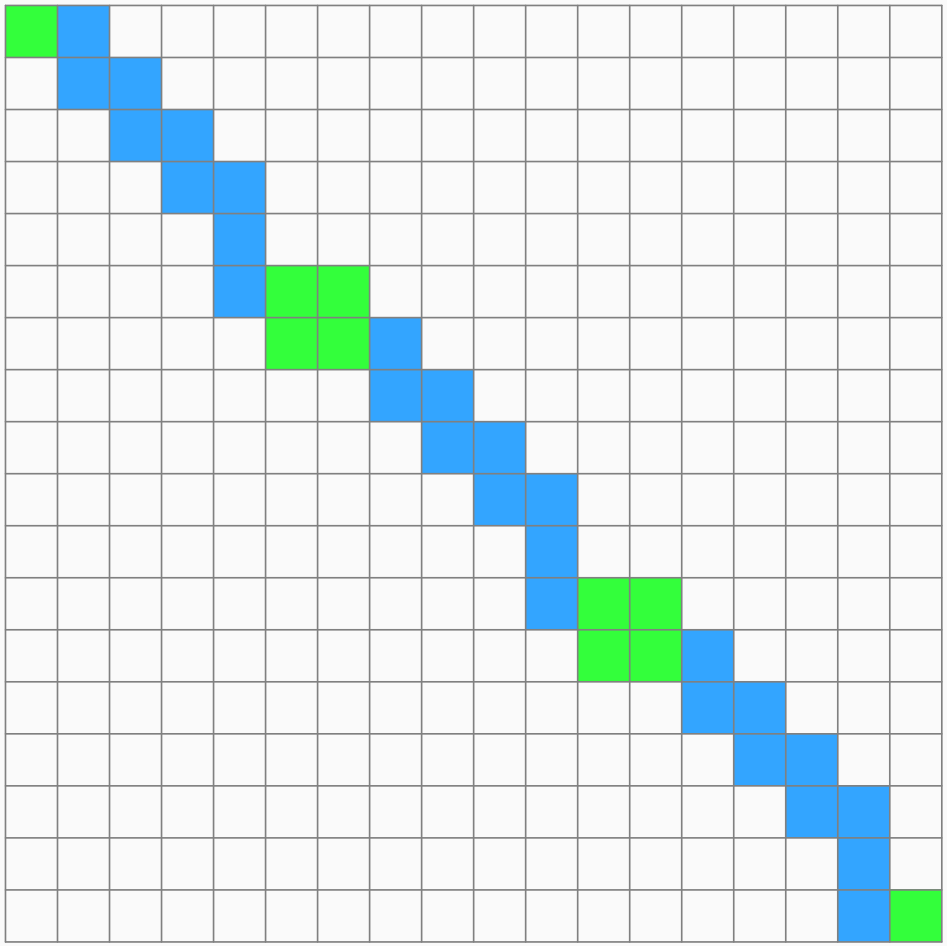
All threads use coarse solution

$A =$



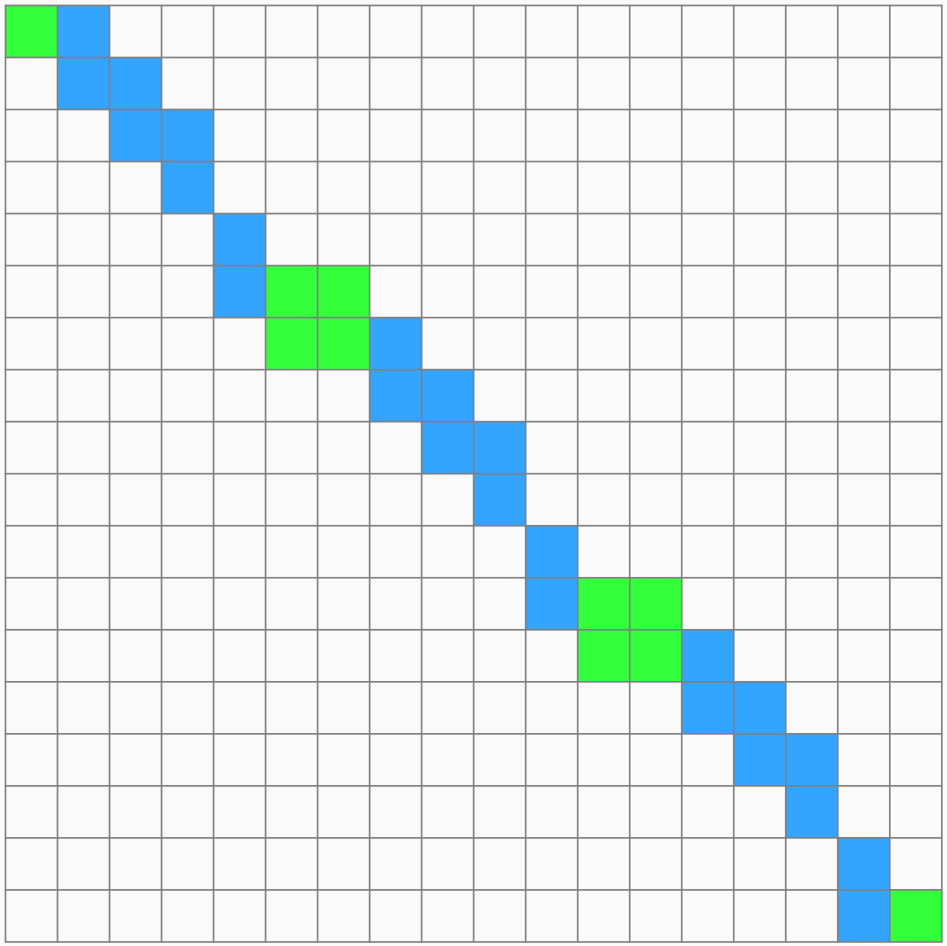
All threads use coarse solution

$A =$



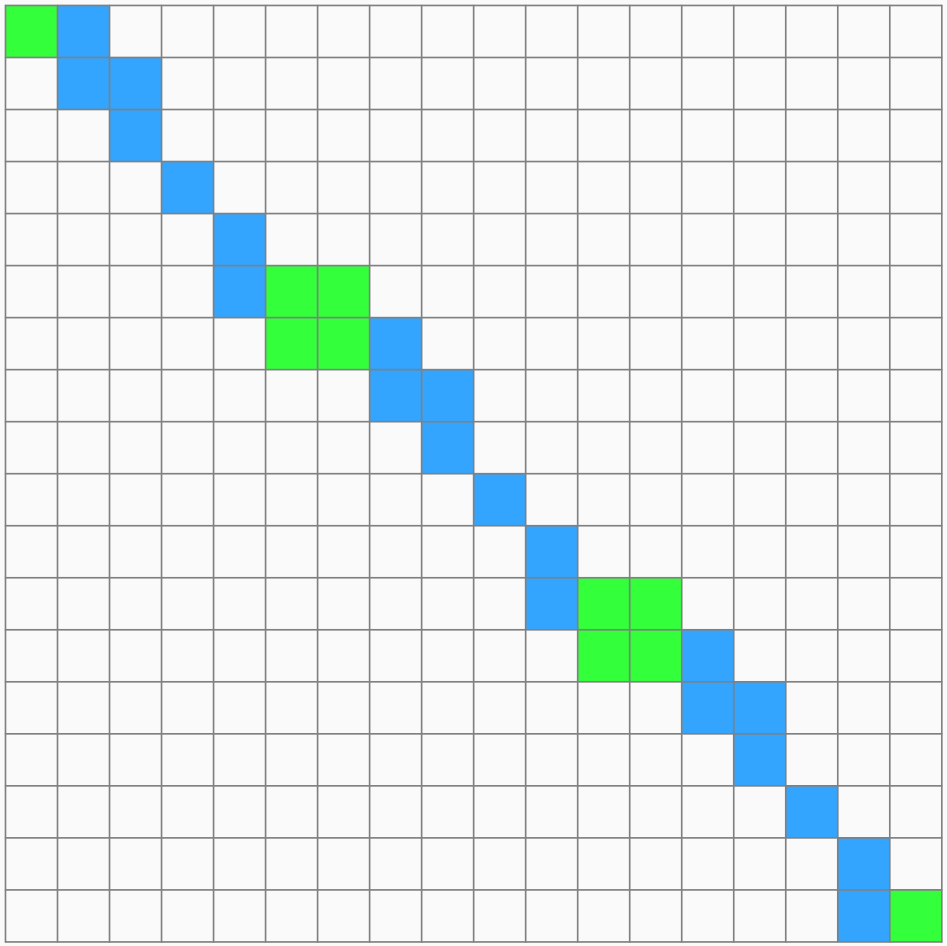
All threads use coarse solution

$A =$



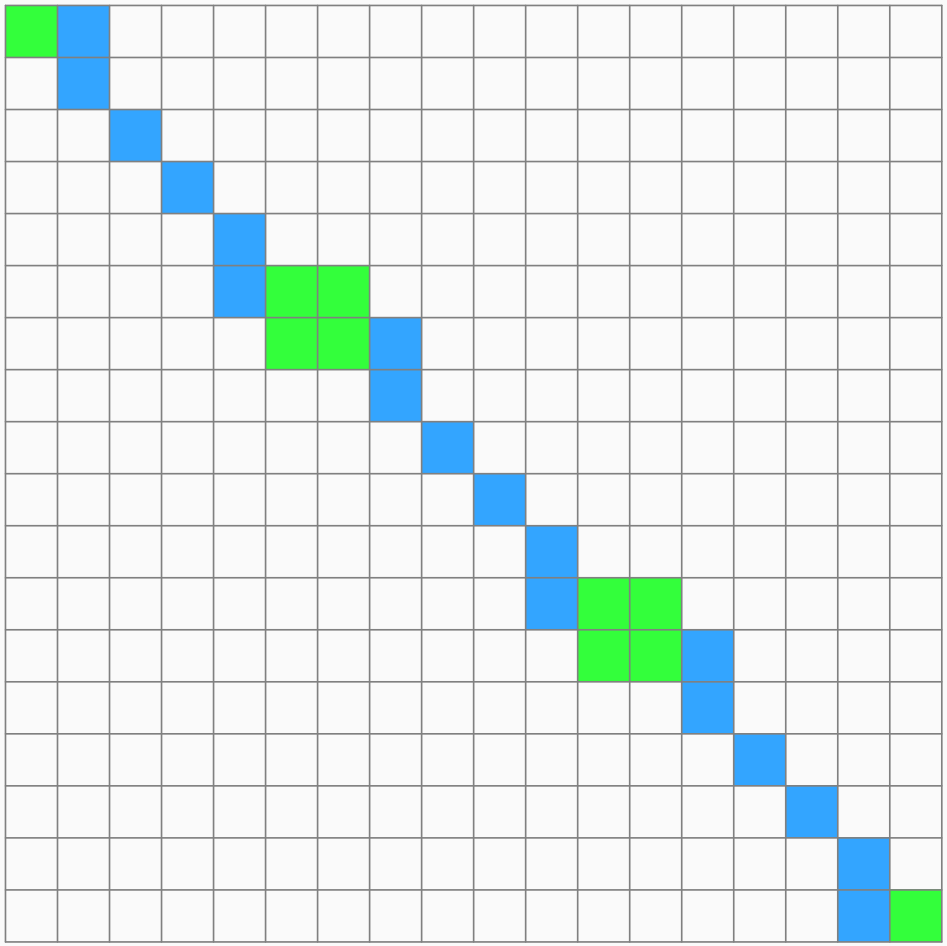
All threads use coarse solution

$A =$



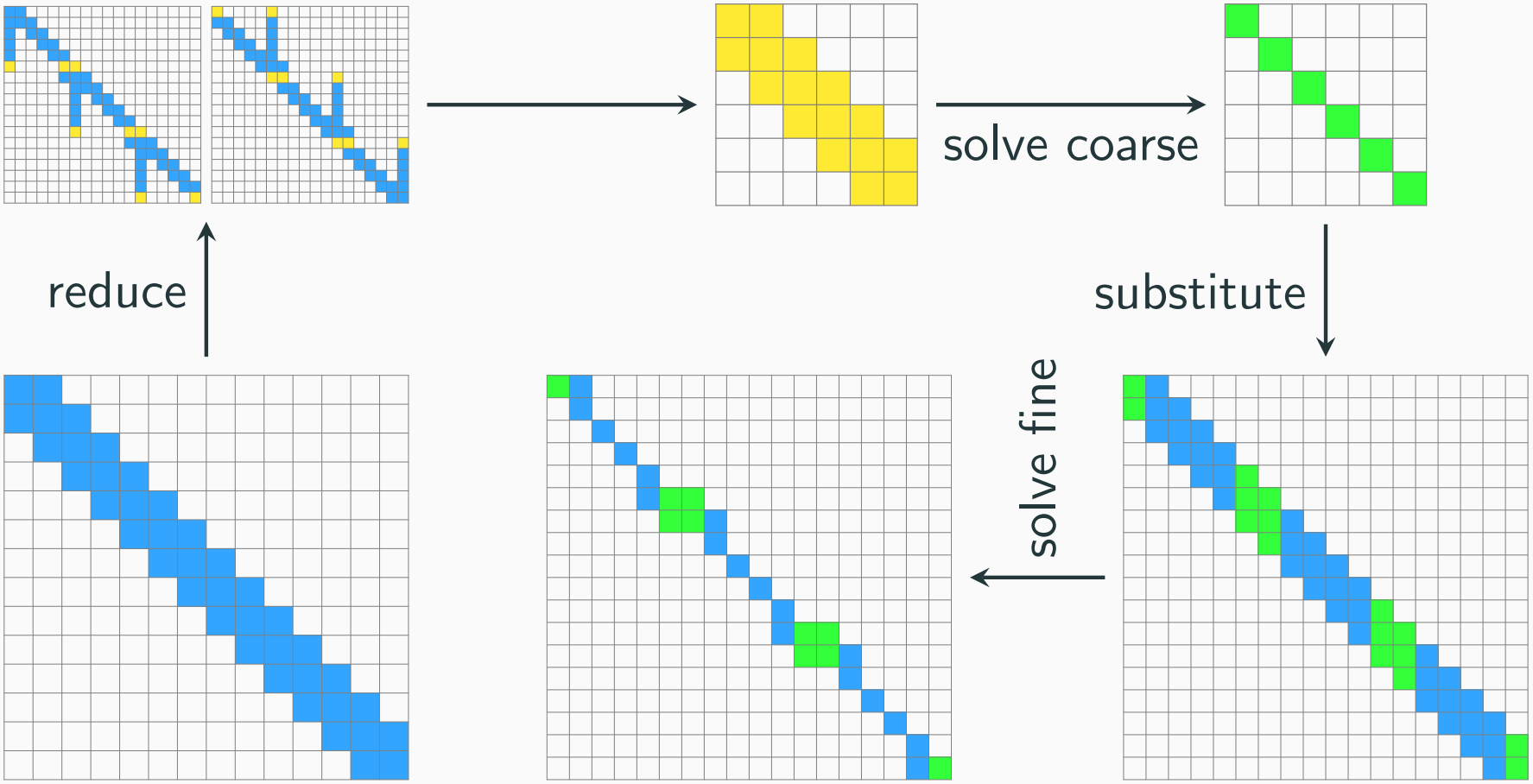
All threads use coarse solution

$A =$



✓

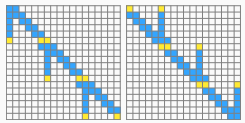
Overview



Pivoting implementation

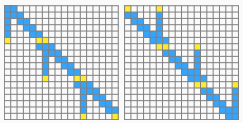
Pivoting implementation

- easy for reduction kernel
  - not necessary to save pivoting information

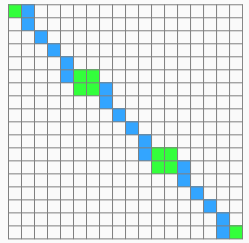


Pivoting implementation

- easy for reduction kernel
  - not necessary to save pivoting information

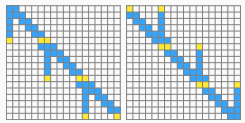


- difficult for substitution kernel
  - upwards oriented elimination requires pivoting information

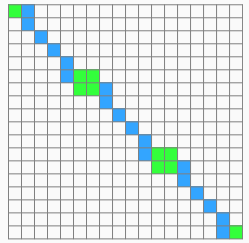


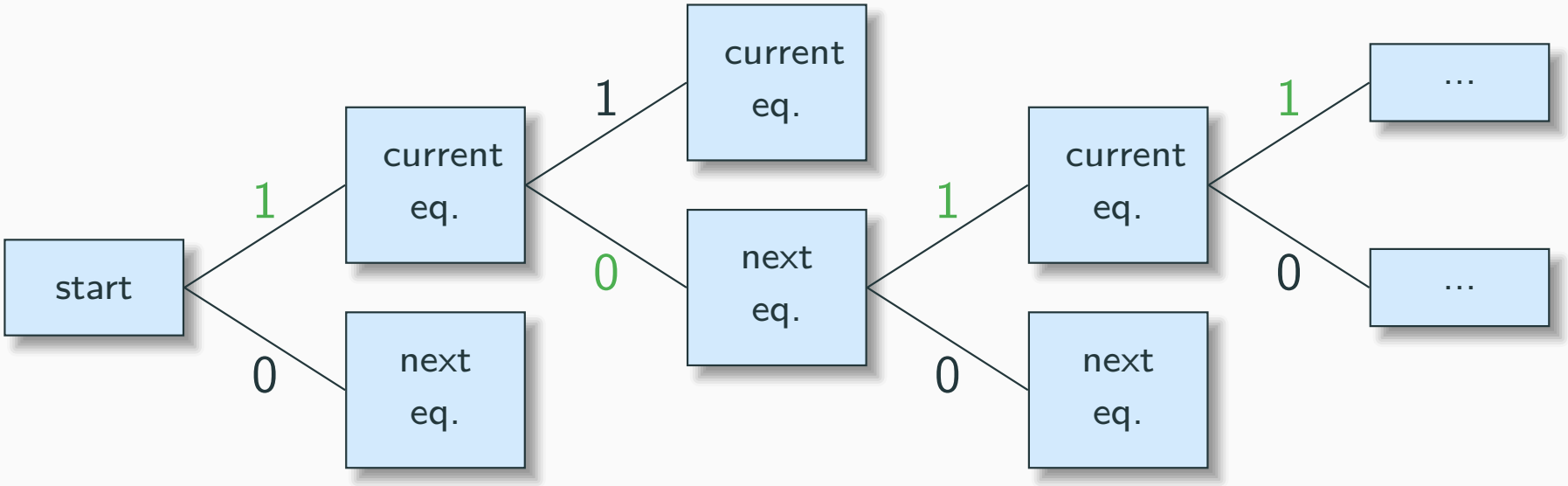
Pivoting implementation

- easy for reduction kernel
  - not necessary to save pivoting information



- difficult for substitution kernel
  - upwards oriented elimination requires pivoting information



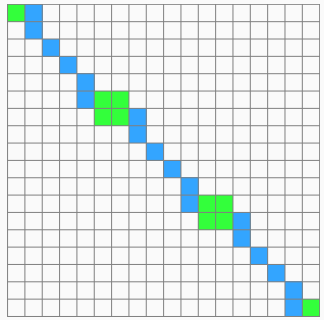


$\rightarrow$ save in bitset 0b000...1101

Substitution kernel declaration

Substitution kernel declaration

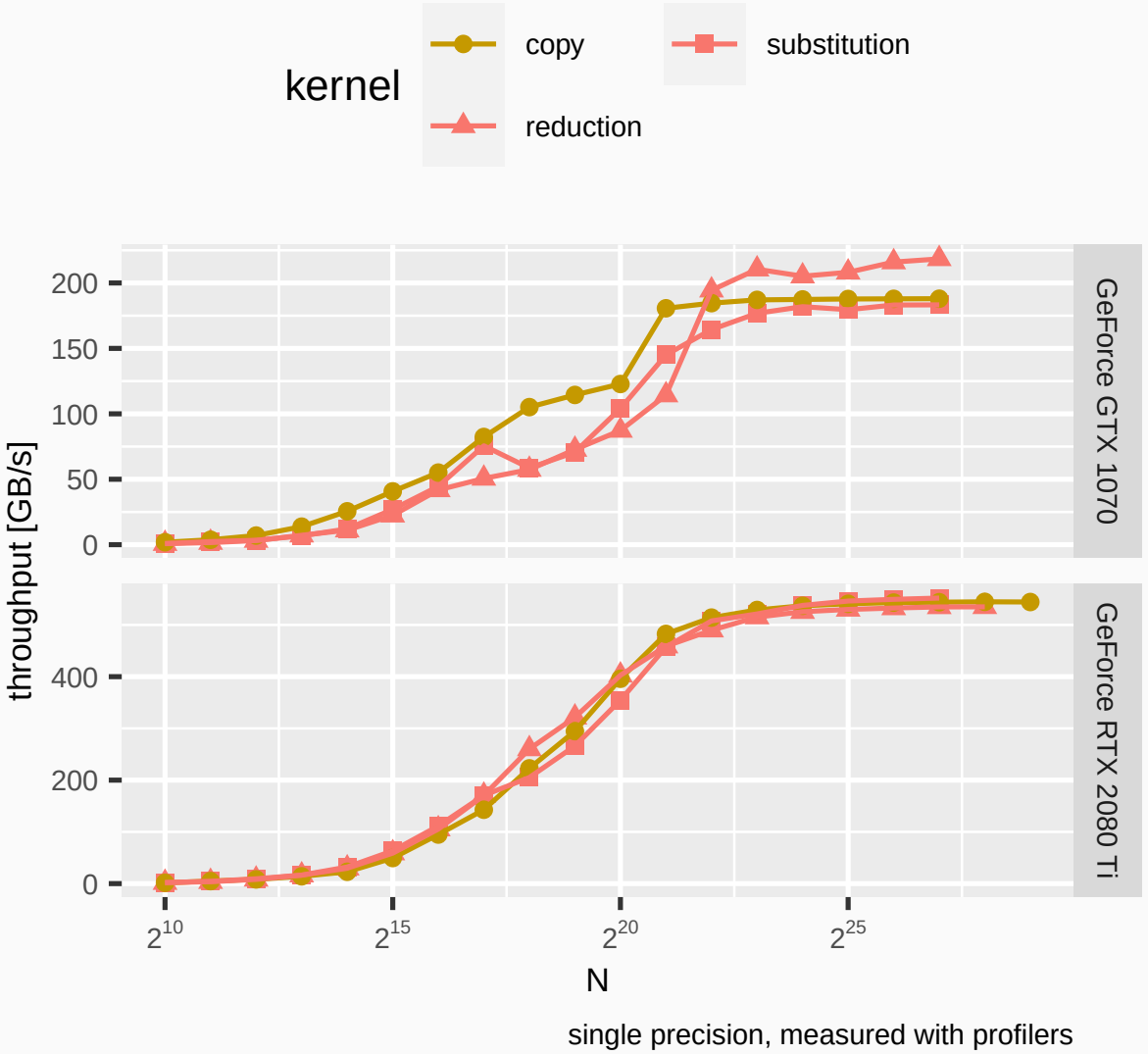
```
template <typename data_type,
          int num_equations_per_partition, // M
          int block_dim,
          int num_partitions_per_block> // L
__global__ void substitution_kernel(
    const data_type* coarse_solution,
    const data_type* fine_a, ///
    const data_type* fine_b, /// bands
    const data_type* fine_c, ///
    const data_type* fine_d,
    data_type* fine_solution,
    int N);
```



→ Compile time optimizations are essential for performance.

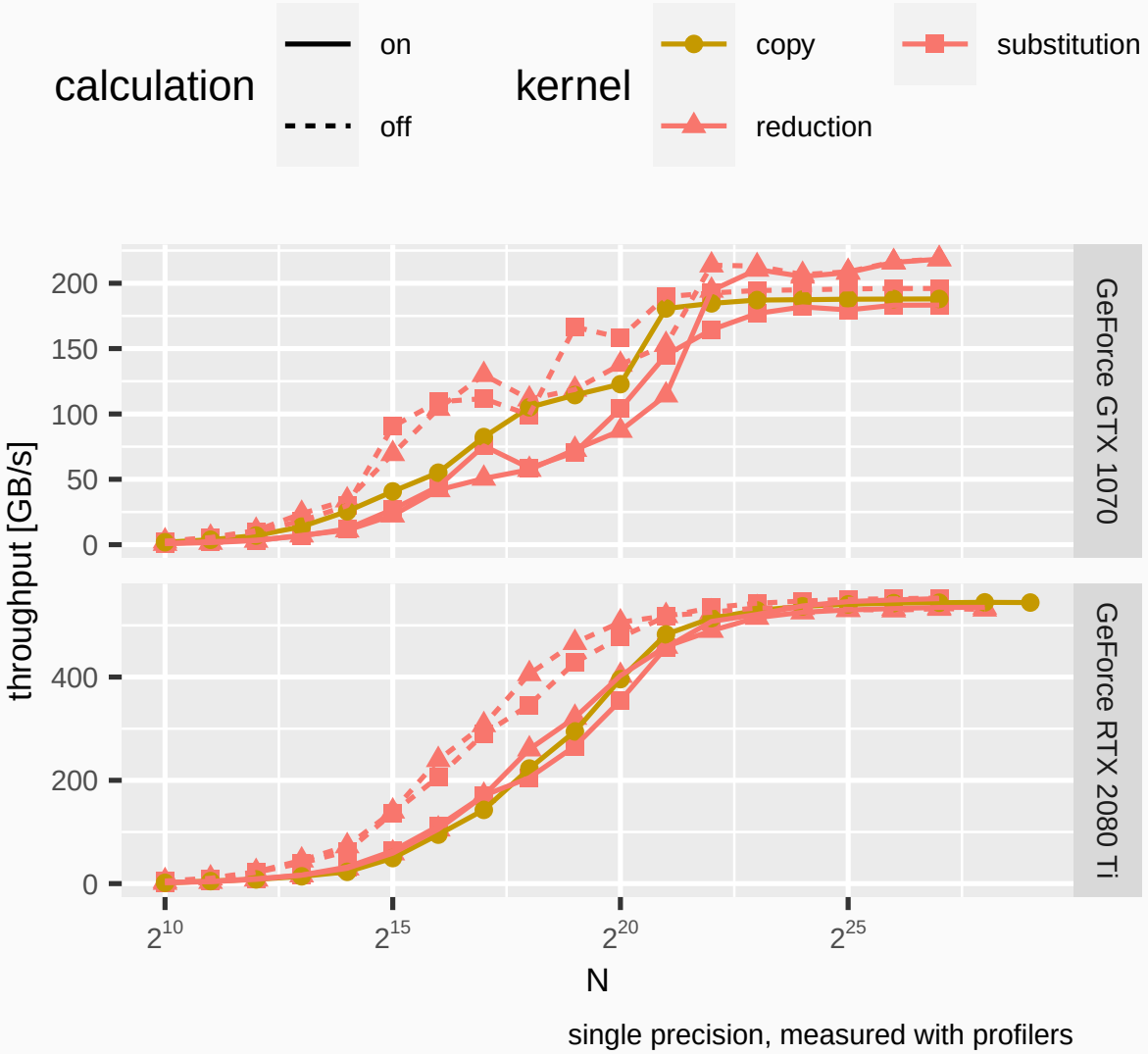
Benchmarks

Benchmarks



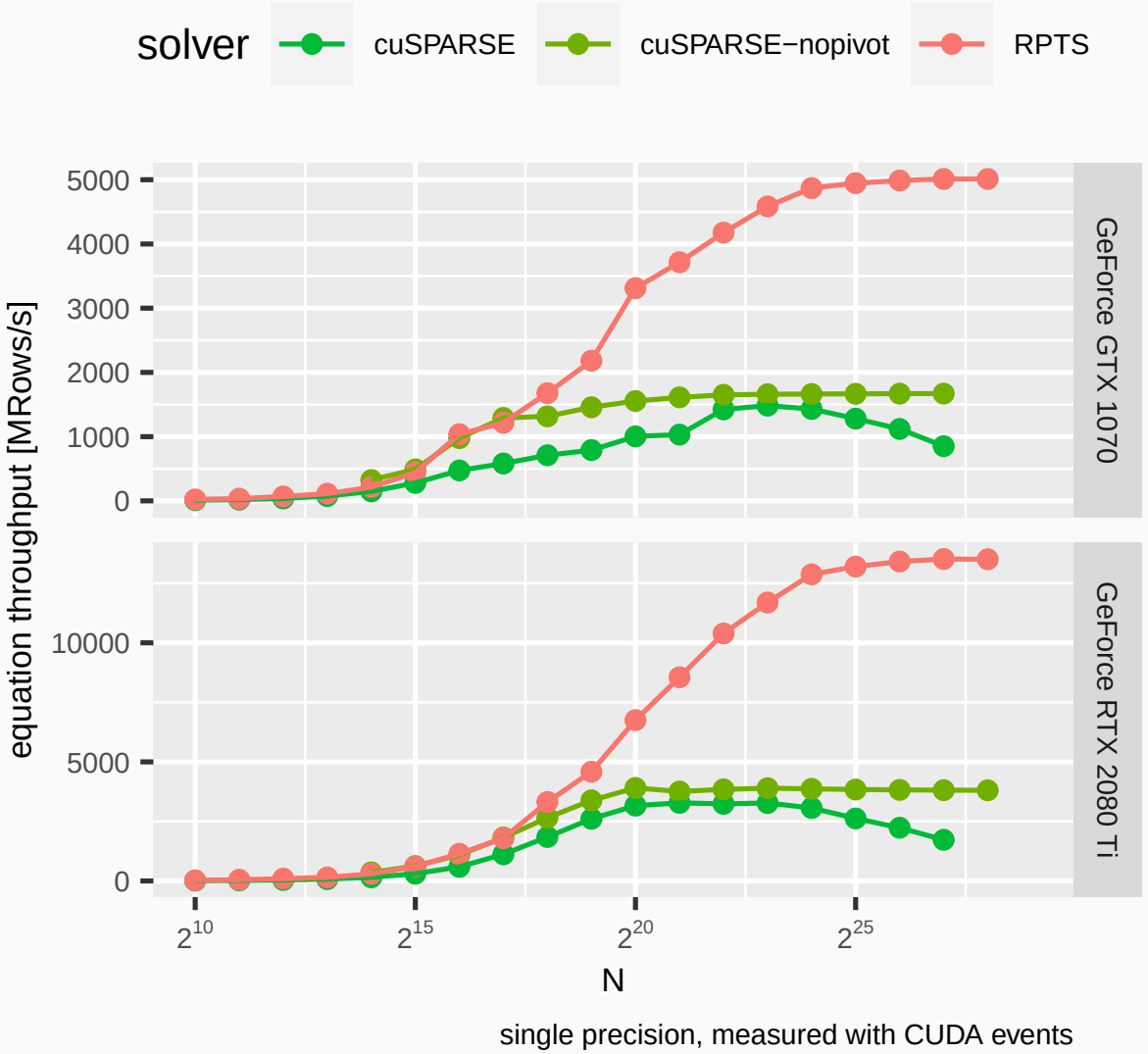
single precision, measured with profilers

Benchmarks



single precision, measured with profilers

Benchmarks


solver
cuSPARSE
cuSPARSE−nopivot
RPTS
equation throughput [MRows/s]
5000
4000
3000
2000
1000
0
10000
5000
0
GeForce GTX 1070
GeForce RTX 2080 Ti
2^{10}
2^{15}
2^{20}
2^{25}
N
single precision, measured with CUDA events

Kernel Optimization

How can this performance be achieved?

Kernel Optimization

How can this performance be achieved?

- templates, unrolling, compile time optimizations

Kernel Optimization

How can this performance be achieved?

- templates, unrolling, compile time optimizations
- custom data load and store functions to avoid register dependencies

Kernel Optimization

How can this performance be achieved?

- templates, unrolling, compile time optimizations
- custom data load and store functions to avoid register dependencies
- bit optimizations

Kernel Optimization

How can this performance be achieved?

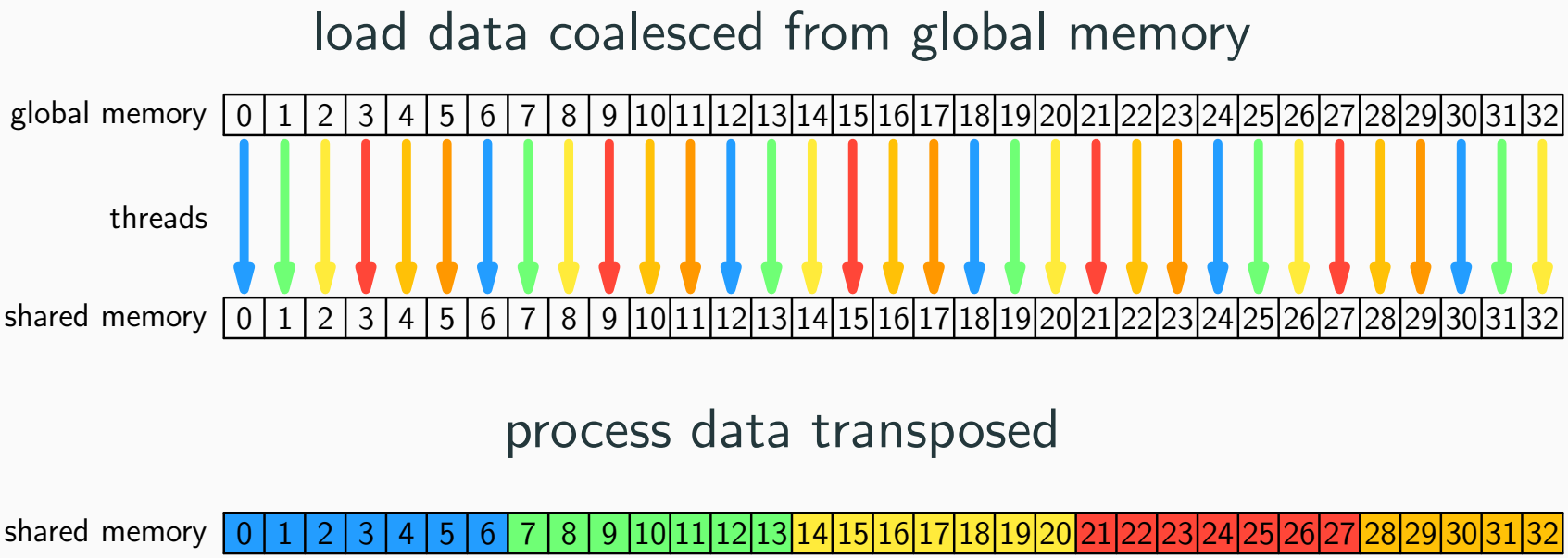
- templates, unrolling, compile time optimizations
- custom data load and store functions to avoid register dependencies
- bit optimizations
- avoid warp divergence (E.g. ternary operator `flag ? foo : bar;` instead of if-statement)

Kernel Optimization

How can this performance be achieved?

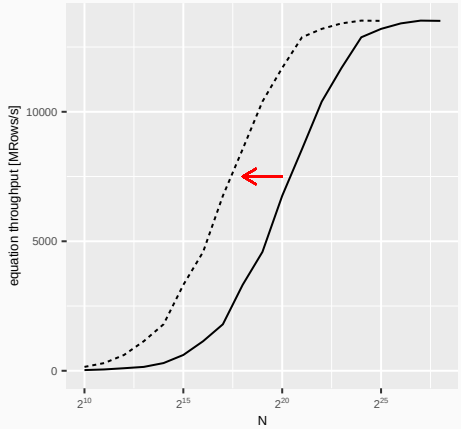
- templates, unrolling, compile time optimizations
- custom data load and store functions to avoid register dependencies
- bit optimizations
- avoid warp divergence (E.g. ternary operator `flag ? foo : bar;` instead of if-statement)
- shared memory usage

Shared memory layout

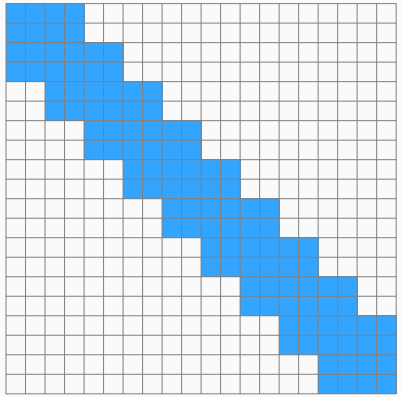


Future work

Future work



- increase GPU utilization for small system sizes
- solve block tridiagonal systems
- multiple right-hand sides



Take home message

Take home message

Strong privatization works great, if data set is large enough.

Code with CI, modern C++, and a large test suite

https://mp-force.ziti.uni-heidelberg.de/asc/code/tridigpu

Appendix

Schur Complement

$$A = \begin{pmatrix} A_{PP} & A_{PI} \\ A_{IP} & A_{II} \end{pmatrix} = \begin{pmatrix} I & 0 \\ A_{IP}A_{PP}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{PP} & A_{PI} \\ 0 & S \end{pmatrix} \tag{1}$$

$$S := A_{II} - A_{IP}A_{PP}^{-1}A_{PI} \tag{2}$$

The above factorization with the Schur-complement S allows the recursive solution of

$$Ax = \begin{pmatrix} A_{PP} & A_{PI} \\ A_{IP} & A_{II} \end{pmatrix} \begin{pmatrix} x_P \\ x_I \end{pmatrix} = \begin{pmatrix} d_P \\ d_I \end{pmatrix} = d \tag{3}$$

Pivoting

$$m_p = \max\left(|s_p[0]|, |s_p[1]|, |s_p[2]|\right) \tag{4}$$

$$m_c = \max\left(|s_c[1]|, |s_c[2]|, |s_c[3]|\right) \tag{5}$$

$$|s_c[1]| m_p \leq |s_p[1]| m_c \tag{6}$$