UNIVERSITAT JAUME I

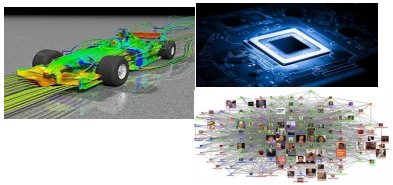
Sparse Linear System Solvers on GPUs

Parallel Preconditioning, Workload Balancing, and Communication Reduction

Goran Flegar

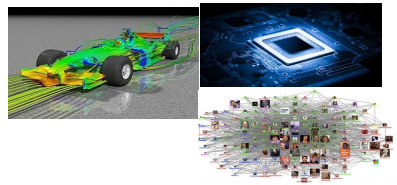
Introduction

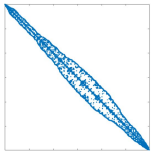
- Real-world problem transformed into a linear system:
  - Large number of unknowns, most coefficients zero



Introduction

- Real-world problem transformed into a linear system:
  - Large number of unknowns, most coefficients zero
- Possible approach: iterative methods
  - Krylov-subspace based linear solvers
    - Sparse matrix-vector product (SpMV) $y := Ax$
    - BLAS-1 operations $y := \alpha x + y$
      $\delta := x^T y$

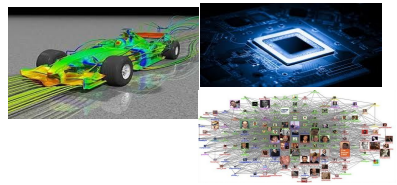


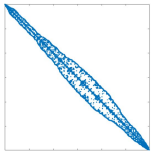


$$Ax = b$$

Introduction

- Real-world problem transformed into a linear system:
  - Large number of unknowns, most coefficients zero
- Possible approach: iterative methods
  - Krylov-subspace based linear solvers
    - Sparse matrix-vector product (SpMV) $y := Ax$
    - BLAS-1 operations $y := \alpha x + y$ $\delta := x^T y$
  - **Sparse matrix formats & SpMV**
    - accelerate each iteration of the solver

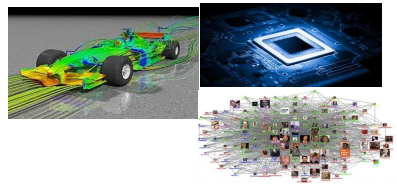


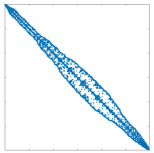


$$Ax = b$$

Introduction

- Real-world problem transformed into a linear system:
  - Large number of unknowns, most coefficients zero
- Possible approach: iterative methods
  - Krylov-subspace based linear solvers
    - Sparse matrix-vector product (SpMV) $y := Ax$
    - BLAS-1 operations $y := \alpha x + y$ $\delta := x^T y$
  - **Sparse matrix formats & SpMV**
    - accelerate each iteration of the solver
  - **Preconditioners**
    - reduce the number of iterations

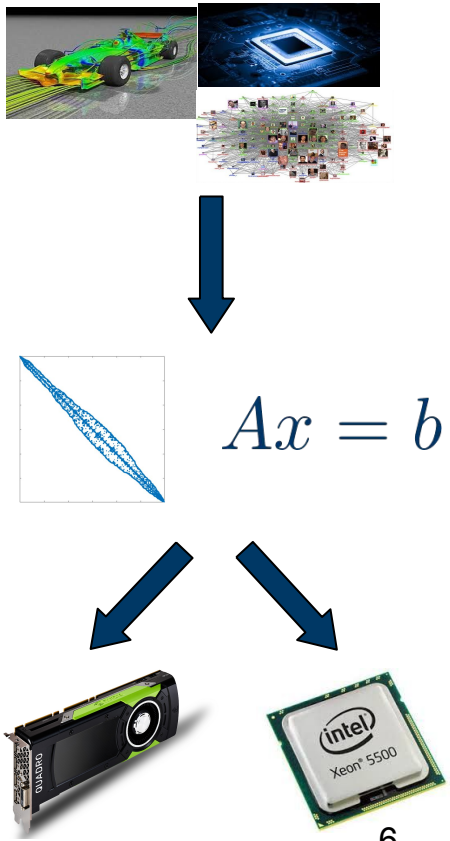




$$Ax = b$$

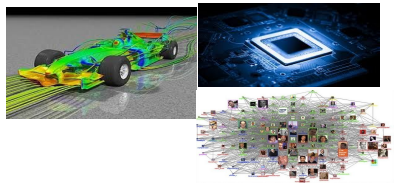
Introduction

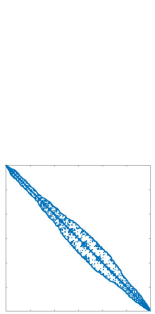
- Real-world problem transformed into a linear system:
  - Large number of unknowns, most coefficients zero
- Possible approach: iterative methods
  - Krylov-subspace based linear solvers
    - Sparse matrix-vector product (SpMV) $y := Ax$
    - BLAS-1 operations $y := \alpha x + y$ $\delta := x^T y$
  - **Sparse matrix formats & SpMV**
    - accelerate each iteration of the solver
  - **Preconditioners**
    - reduce the number of iterations
- Special hardware (e.g. GPUs)
  - Do not implement everything from scratch…



Introduction

- Real-world problem transformed into a linear system:
  - Large number of unknowns, most coefficients zero
- Possible approach: iterative methods
  - Krylov-subspace based linear solvers
    - Sparse matrix-vector product (SpMV) $y := Ax$
    - BLAS-1 operations $y := \alpha x + y$ $\delta := x^T y$
  - **Sparse matrix formats & SpMV**
    - accelerate each iteration of the solver
  - **Preconditioners**
    - reduce the number of iterations
- Special hardware (e.g. GPUs)
  - Do not implement everything from scratch…
  - **Use a library instead:** Ginkgo





$$Ax = b$$

Ginkgo

Sparse Matrix Formats and SpMV

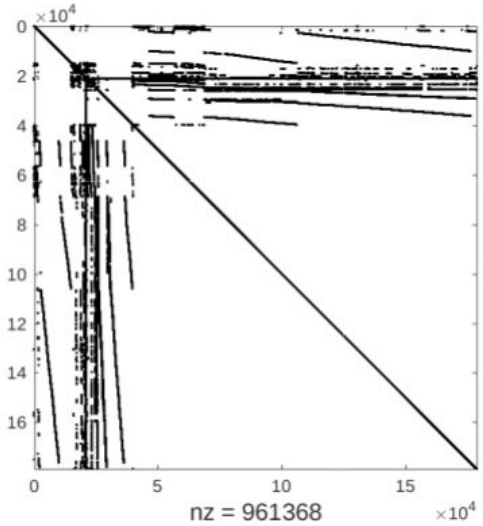
1. Balanced CSR SpMV
2. Balanced COO SpMV
3. Block-level SpMV

Flegar et al., *Balanced CSR sparse matrix-vector product on graphics processors*, Euro-Par 2017

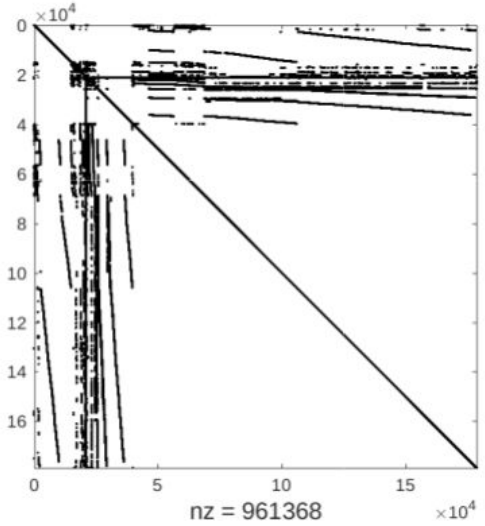
Flegar et al., *Overcoming load imbalance for irregular sparse matrices*, IA3 2017

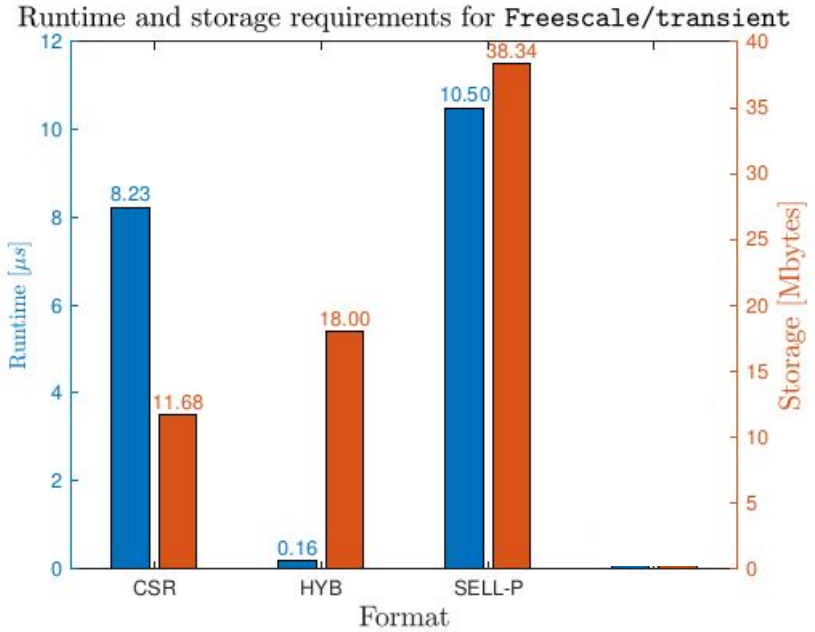
Anzt et al., *Flexible batched sparse matrix-vector product on GPUs*, ScalA 2017

Example



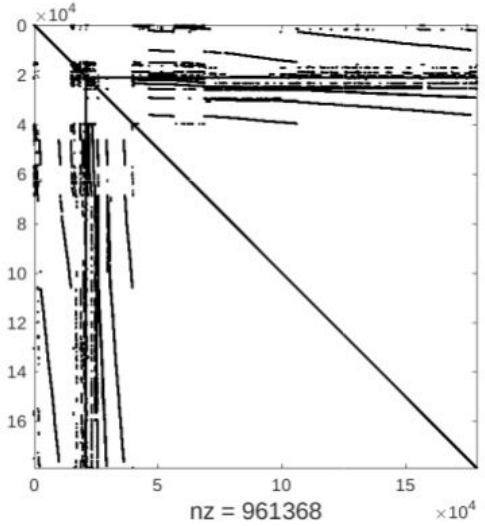
Example

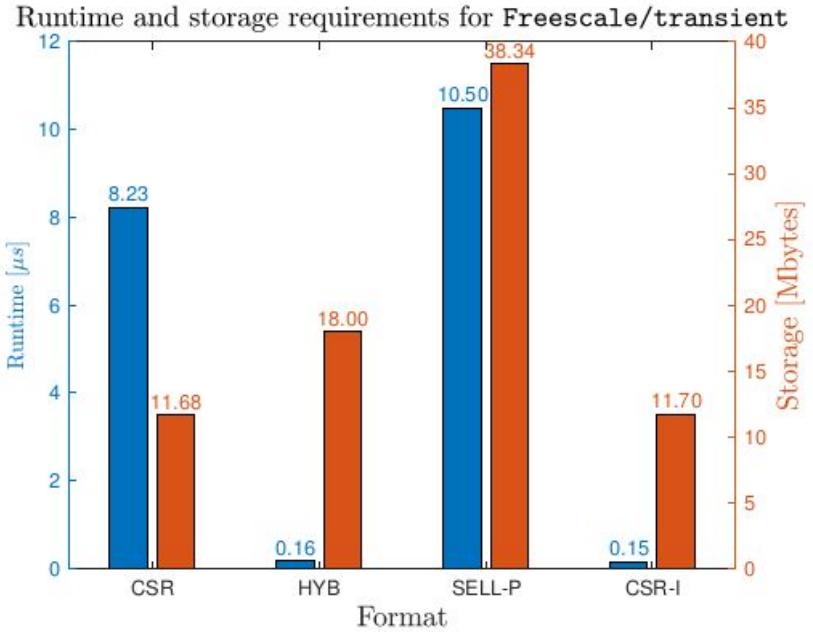




Can we do better than HYB using CSR?

Example



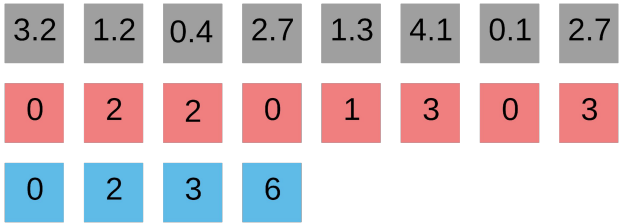


55x speedup

Can we do better than HYB using CSR?

YES!

CSR SpMV



Values (val)

Column indexes (colidx)

Row pointers (rowptr)

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

$$y := Ax$$

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] +=  val[j] * x [ colidx[j] ];
5   }
6 }
```

CSR SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	2	2	0	1	3	0	3
0	2	3	6				

Values (val)

Column indexes (colidx)

Row pointers (rowptr)

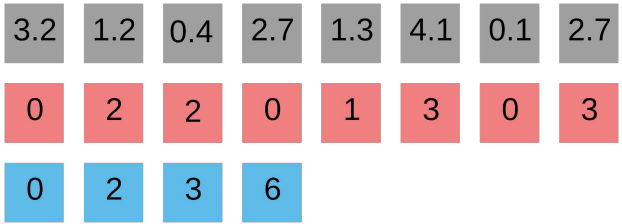
3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

$$y := Ax$$

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] +=  val[j] * x [ colidx[j] ];
5   }
6 }
```

Parallelize outer loop?

CSR SpMV



Values (val)

Column indexes (colidx)

Row pointers (rowptr)

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

$$y := Ax$$

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] +=  val[j] * x [ colidx[j] ];
5   }
6 }
```

Parallelize outer loop?

- Load imbalance
- Non-coalescence

CSR-I Algorithm

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] +=  val[j] * x [ colidx[j] ];
5   }
6 }
```

CSR-I Algorithm

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] +=  val[j] * x [ colidx[j] ];
5   }
6 }
```

Collapse loops.

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   int row = -1, next_row = 0, nnz = rowptr[m];
3   for (int i = 0; i < nnz; ++i) {
4       while (i >= next_row) next_row = rowptr[++row+1];
5       y[row] += val[i] * x[ colidx[i] ];
6 }}
```

CSR-I Algorithm

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] +=  val[j] * x [ colidx[j] ];
5   }
6 }
```

Collapse loops.

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   int row = -1, next_row = 0, nnz = rowptr[m];
3   for (int i = 0; i < nnz; ++i) {
4       while (i >= next_row) next_row = rowptr[++row+1];
5       y[row] += val[i] * x[ colidx[i] ];
6 }}
```

Split into equal chunks.

```
1 const int T = thread_count;
2 void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3   int row = -1, next_row = 0, nnz = rowptr[m];
4   for (int k = 0; k < T; ++k) {
5     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6       while (i >= next_row) next_row = rowptr[++row+1];
7       y[row] += val[i] * x[ colidx[i] ];
8 }}}
```

CSR-I Algorithm

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] +=  val[j] * x [ colidx[j] ];
5   }
6 }
```

Collapse loops.

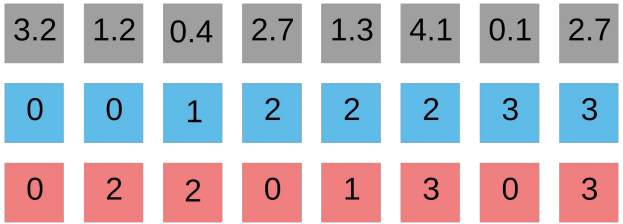
```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   int row = -1, next_row = 0, nnz = rowptr[m];
3   for (int i = 0; i < nnz; ++i) {
4       while (i >= next_row) next_row = rowptr[++row+1];
5       y[row] += val[i] * x[ colidx[i] ];
6 }}
```

Split into equal chunks.

```
1 const int T = thread_count;
2 void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3   int row = -1, next_row = 0, nnz = rowptr[m];
4   for (int k = 0; k < T; ++k) {
5     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6       while (i >= next_row) next_row = rowptr[++row+1];
7       y[row] += val[i] * x[ colidx[i] ];
8 }}}
```

Parallelize outer loop

COO SpMV



Values (val)

Row indexes (rowidx)

Column indexes (colidx)

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

```
1 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < nnz; ++i) {
3       y[rowidx[i]] += val[i] * x[colidx[i]];
4 }}
```

COO SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	0	1	2	2	2	3	3
0	2	2	0	1	3	0	3

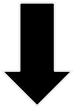
Values (val)

Row indexes (rowidx)

Column indexes (colidx)

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

```
1 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < nnz; ++i) {
3       y[rowidx[i]] += val[i] * x[colidx[i]];
4 }}
```



Split the loop into equal chunks.

```
1 const int T = thread_count;
2 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
3   for (int k = 0; k < T; ++k) {
4     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
5       y[rowidx[i]] += val[i] * x[colidx[i]];
6 }}
```

COO SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	0	1	2	2	2	3	3
0	2	2	0	1	3	0	3

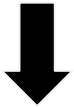
Values (val)

Row indexes (rowidx)

Column indexes (colidx)

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

```
1 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < nnz; ++i) {
3       y[rowidx[i]] += val[i] * x[colidx[i]];
4 }}
```



Split the loop into equal chunks.

```
1 const int T = thread_count;
2 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
3   for (int k = 0; k < T; ++k) {
4     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
5       y[rowidx[i]] += val[i] * x[colidx[i]];
6 }}
```

Parallelize this!

Parallelization

COO:

```
1 const int T = thread_count;
2 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
3   for (int k = 0; k < T; ++k) {
4     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
5       y[rowidx[i]] += val[i] * x[colidx[i]];
6 }}
```

CSR-I:

```
1 const int T = thread_count;
2 void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3   int row = -1, next_row = 0, nnz = rowptr[m];
4   for (int k = 0; k < T; ++k) {
5     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6       while (i >= next_row) next_row = rowptr[++row+1];
7       y[row] += val[i] * x[ colidx[i] ];
8 }}}
```

Parallelization

COO:

```
1 const int T = thread_count;
2 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
3   for (int k = 0; k < T; ++k) {
4     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
5       y[rowidx[i]] += val[i] * x[colidx[i]];
6 }}
```

CSR-I:

```
1 const int T = thread_count;
2 void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3   int row = -1, next_row = 0, nnz = rowptr[m];
4   for (int k = 0; k < T; ++k) {
5     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6       while (i >= next_row) next_row = rowptr[++row+1];
7       y[row] += val[i] * x[ colidx[i] ];
8 }}}
```

Race conditions!

Parallelization

COO:

```
1 const int T = thread_count;
2 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
3   for (int k = 0; k < T; ++k) {
4     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
5       y[rowidx[i]] += val[i] * x[colidx[i]];
6 }}
```

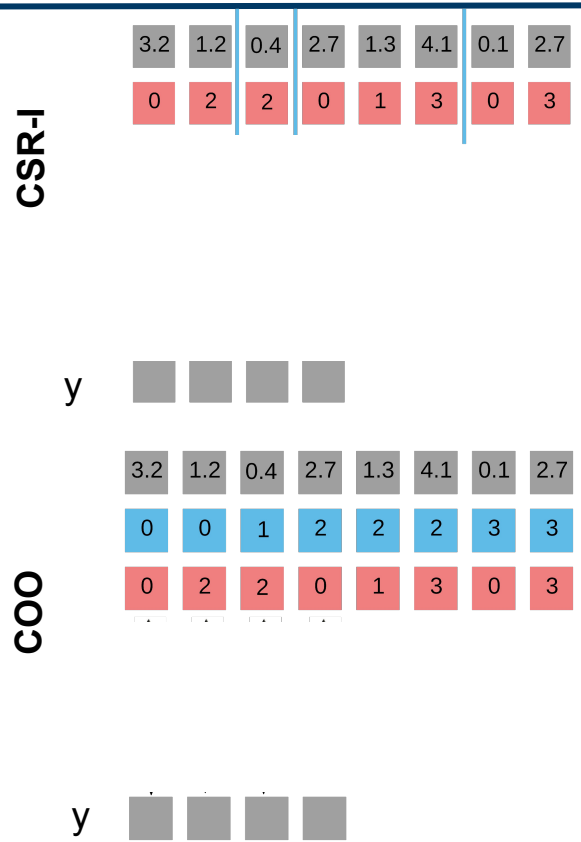
CSR-I:

```
1 const int T = thread_count;
2 void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3   int row = -1, next_row = 0, nnz = rowptr[m];
4   for (int k = 0; k < T; ++k) {
5     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6       while (i >= next_row) next_row = rowptr[++row+1];
7       y[row] += val[i] * x[ colidx[i] ];
8 }}}
```

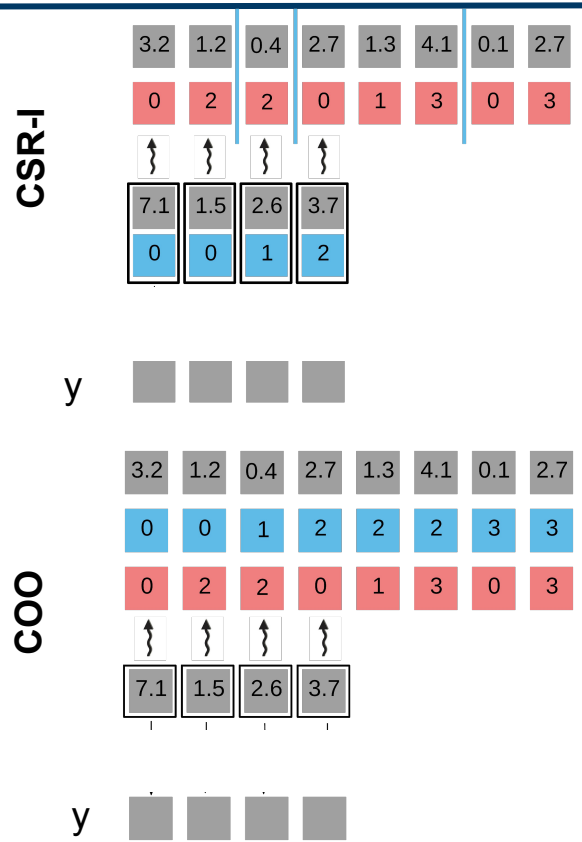
Race conditions!

- Use atomic operations
- Accumulate partial result into registers

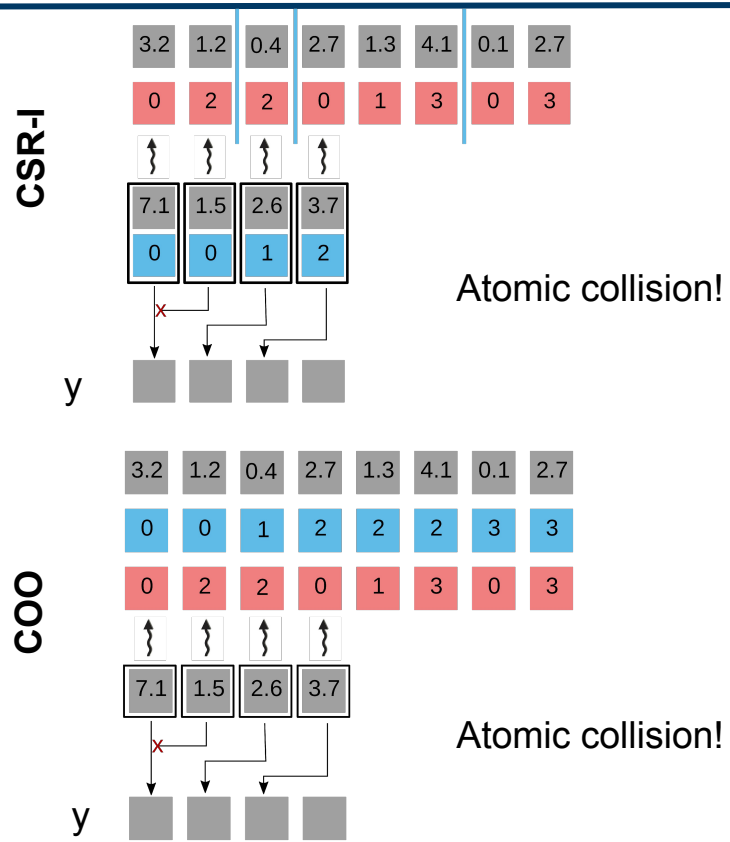
Vectorize within chunk for better access pattern



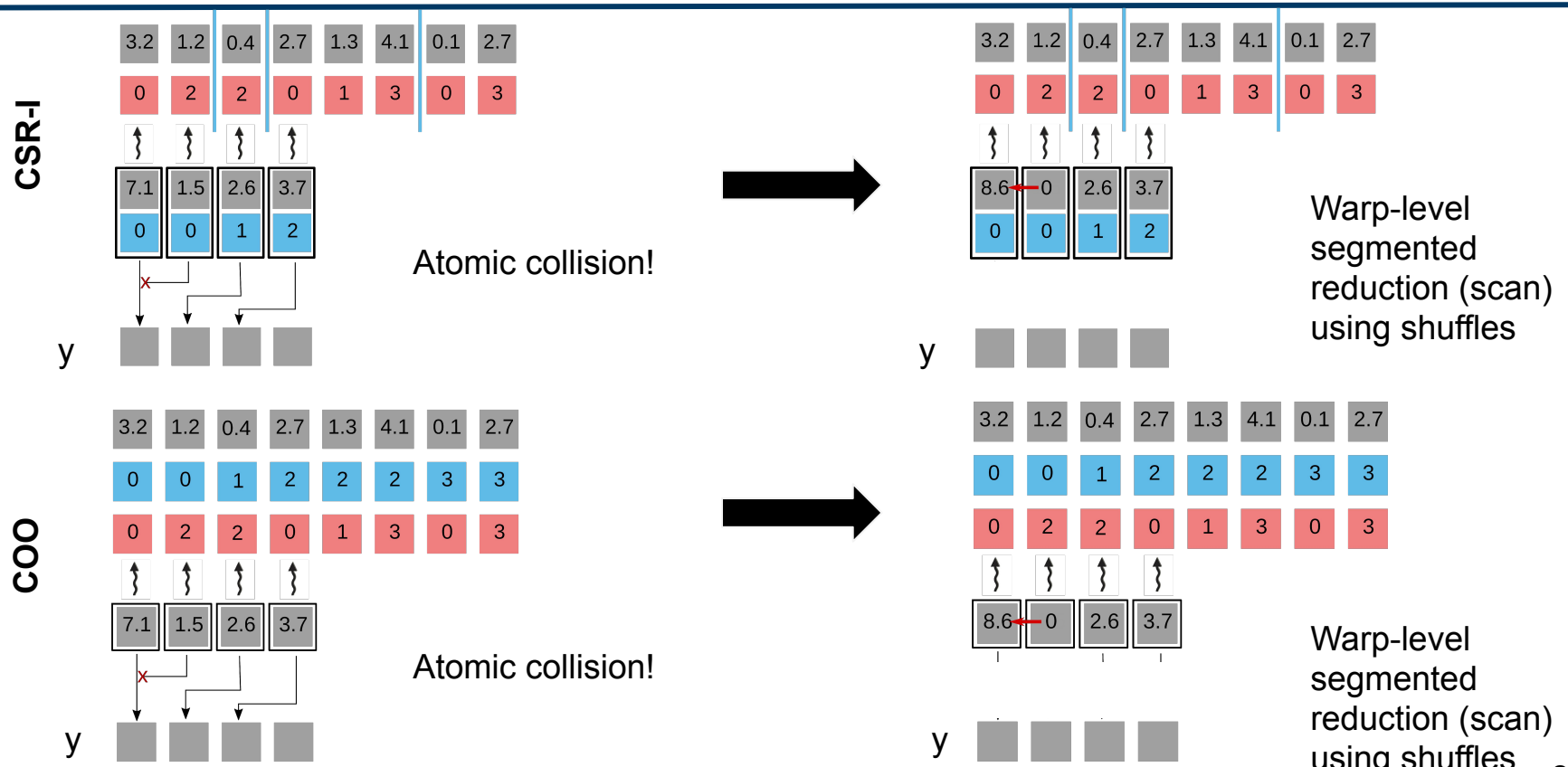
Vectorize within chunk for better access pattern



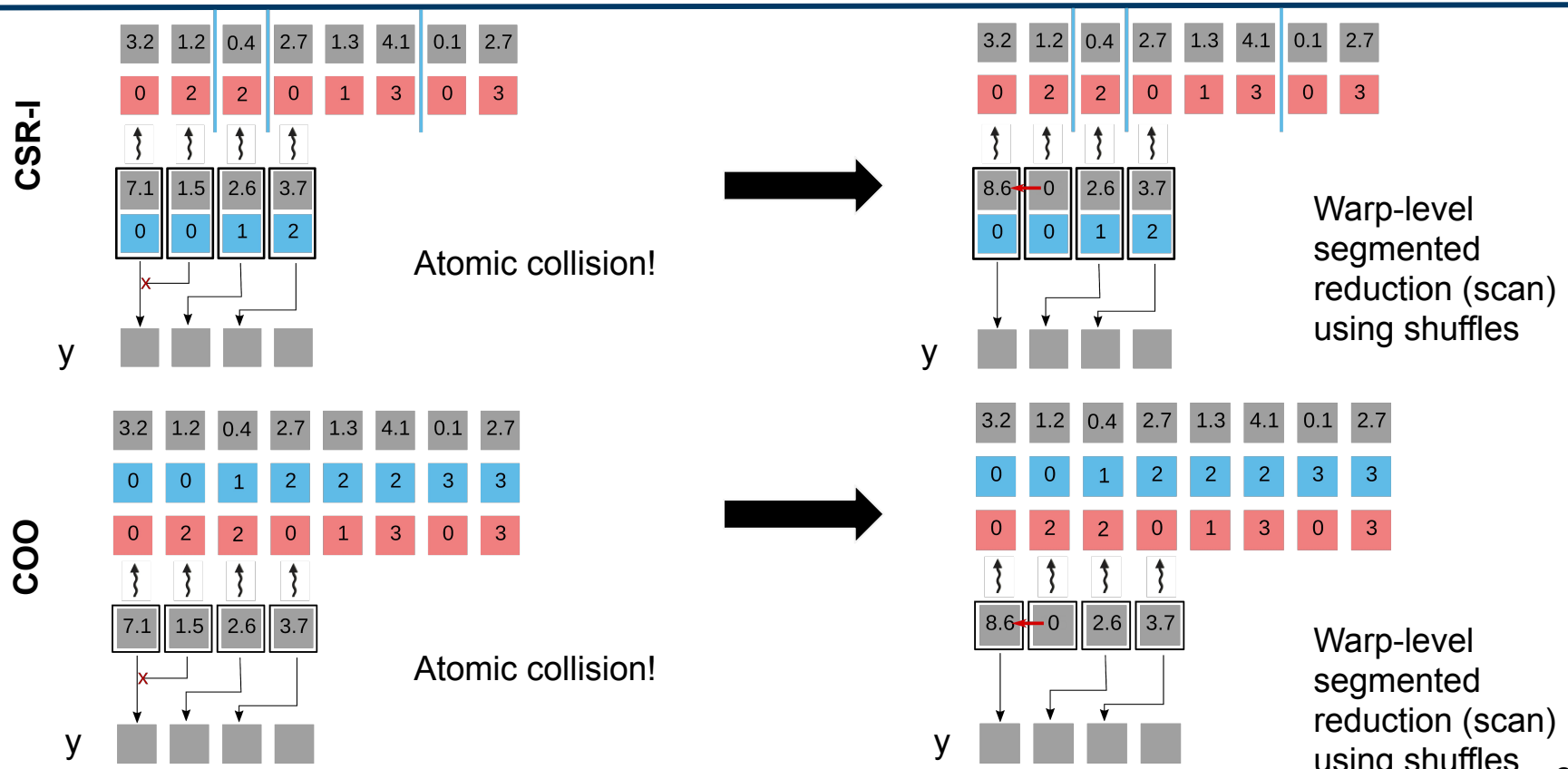
Vectorize within chunk for better access pattern



Vectorize within chunk for better access pattern



Vectorize within chunk for better access pattern



Performance

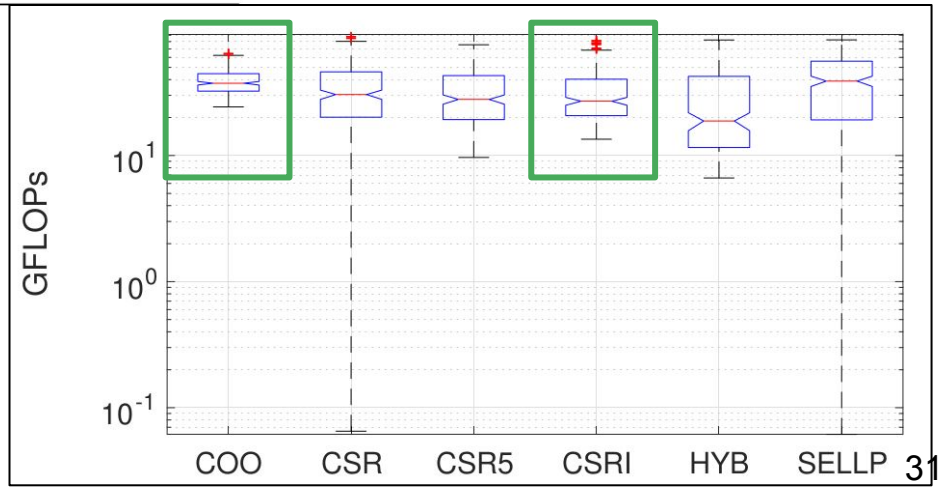
Kernel	min	max	average	median	standard-dev.
COO	**24.29**	**64.32**	**38.86**	**37.24**	**9.16**
CSRI	**13.47**	**81.21**	**31.85**	**26.84**	**14.44**
CSR	0.07	87.43	32.77	30.43	20.07
CSR5	9.66	75.56	31.79	27.15	15.58
HYB	6.64	82.43	27.98	18.74	20.22
SELLP	0.06	82.62	36.42	38.64	22.46

* in GFLOPs

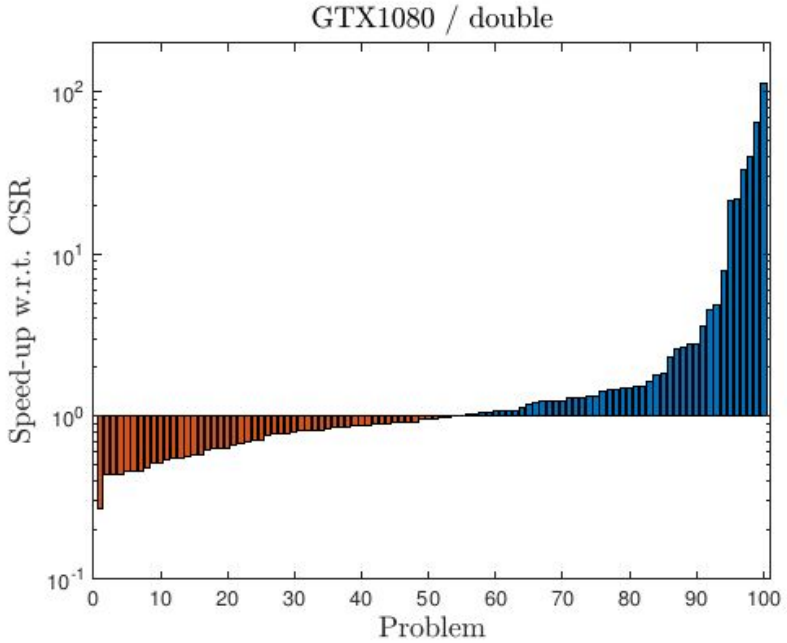
Performance

Kernel	min	max	average	median	standard-dev.
COO	**24.29**	**64.32**	**38.86**	**37.24**	**9.16**
CSRI	**13.47**	**81.21**	**31.85**	**26.84**	**14.44**
CSR	0.07	87.43	32.77	30.43	20.07
CSR5	9.66	75.56	31.79	27.15	15.58
HYB	6.64	82.43	27.98	18.74	20.22
SELLP	0.06	82.62	36.42	38.64	22.46

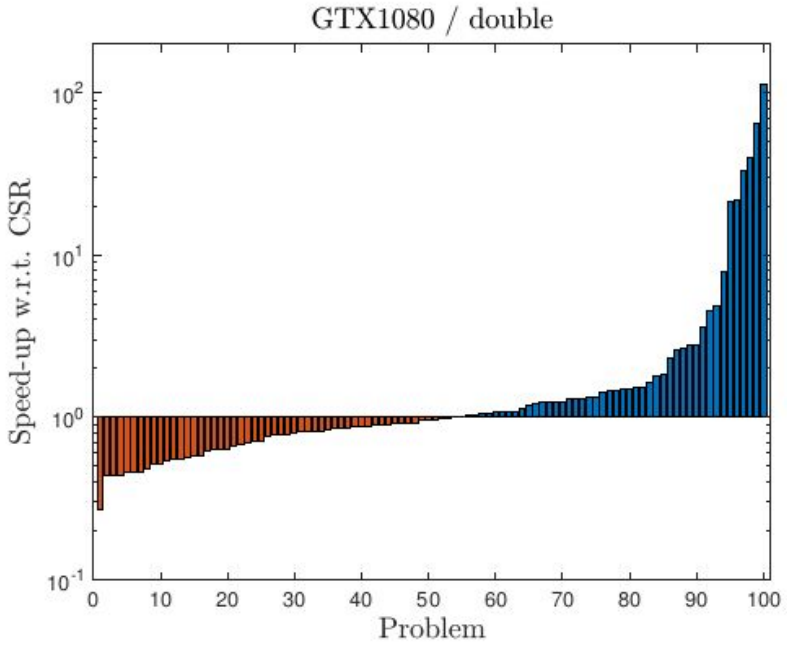
* in GFLOPs



CSR-I vs CSR


GTX1080 / double
Speed-up w.r.t. CSR
10^2
10^1
10^0
10^{-1}
0
10
20
30
40
50
60
70
80
90
100
Problem

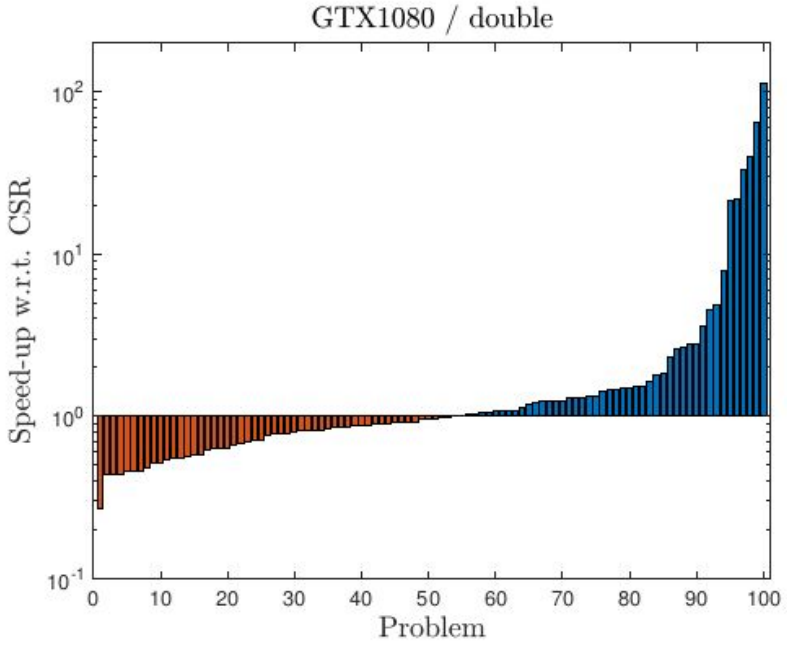
CSR-I vs CSR



No format conversion!

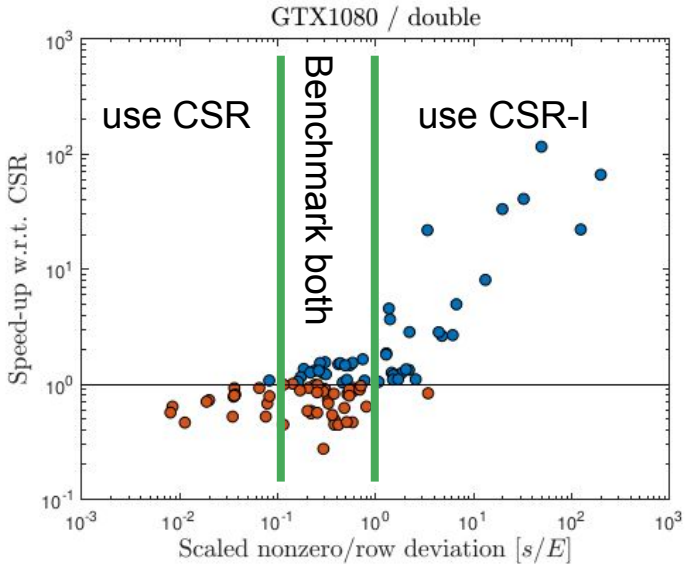
- try both, and use the fastest later on!
- sometimes 1 cuSPARSE SpMV = 100 CSR-I SpMVs

CSR-I vs CSR

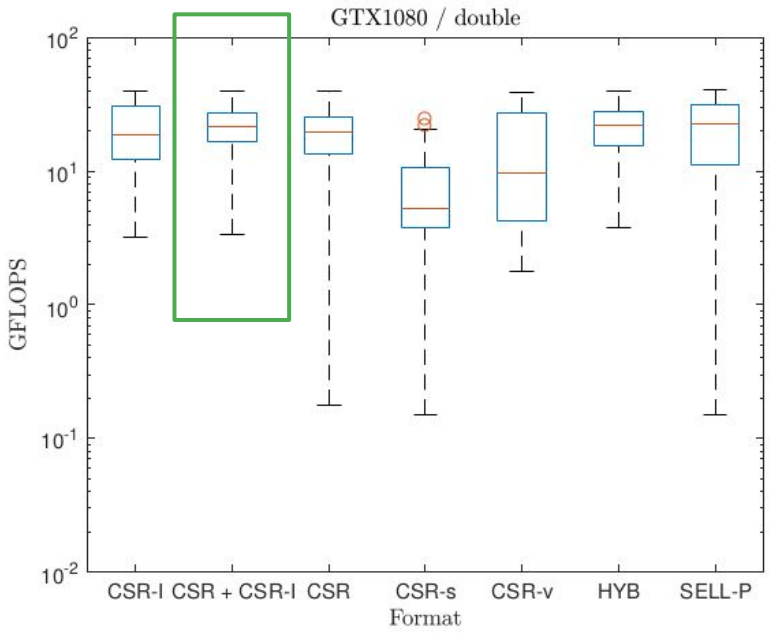


No format conversion!

- try both, and use the fastest later on!
- sometimes 1 cuSPARSE SpMV = 100 CSR-I SpMVs

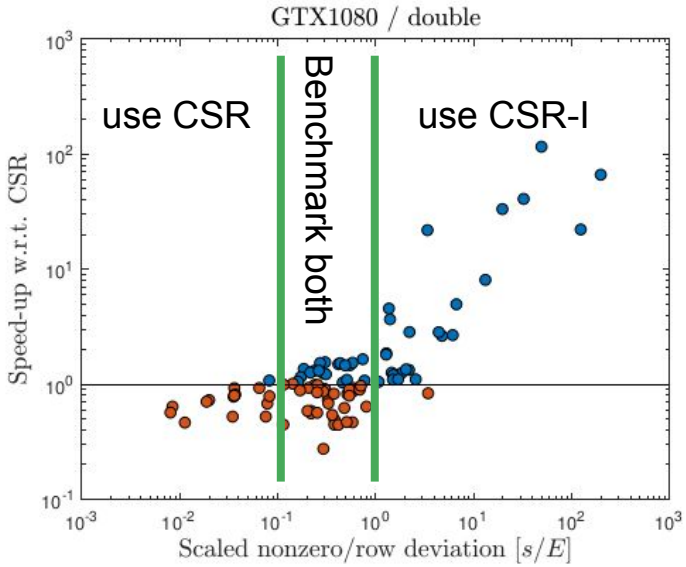


CSR-I vs CSR



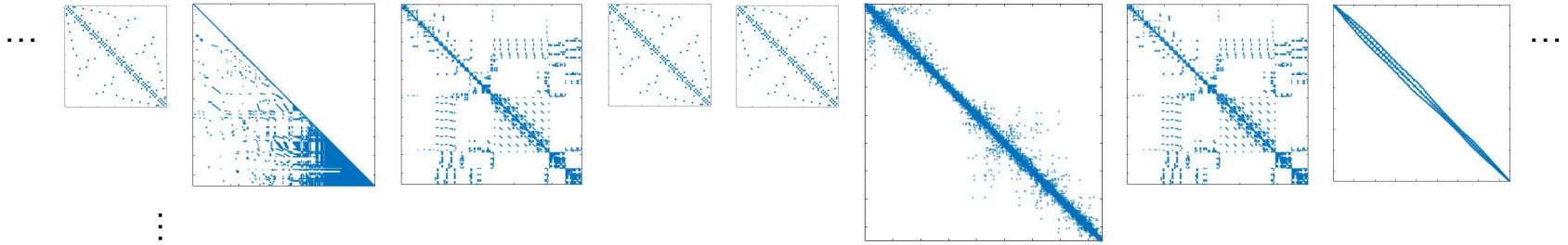
No format conversion!

- try both, and use the fastest later on!
- sometimes 1 cuSPARSE SpMV = 100 CSR-I SpMVs



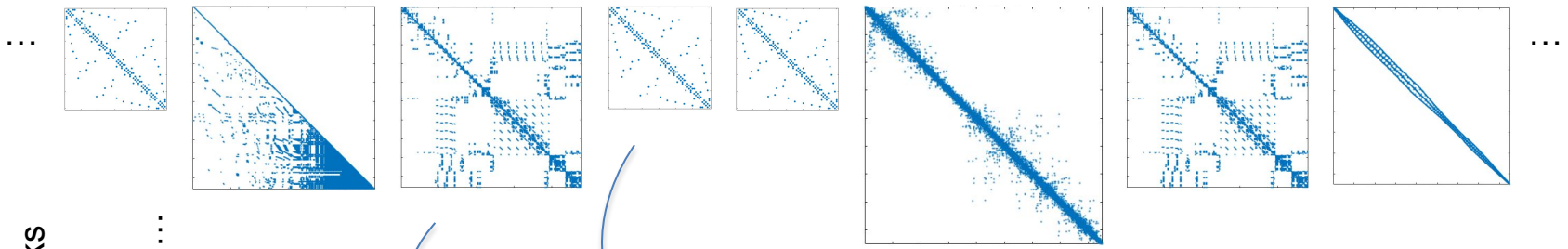
Block-level SpMV

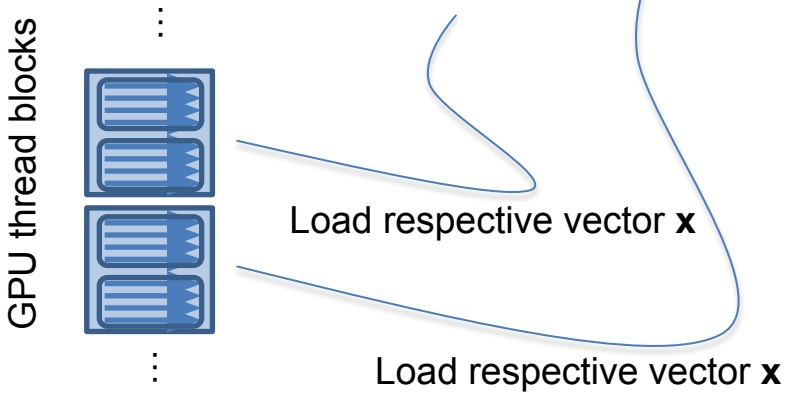
- What if we process many different matrices at a time? (Assume they are all small…)



Block-level SpMV

- What if we process many different matrices at a time? (Assume they are all small…)

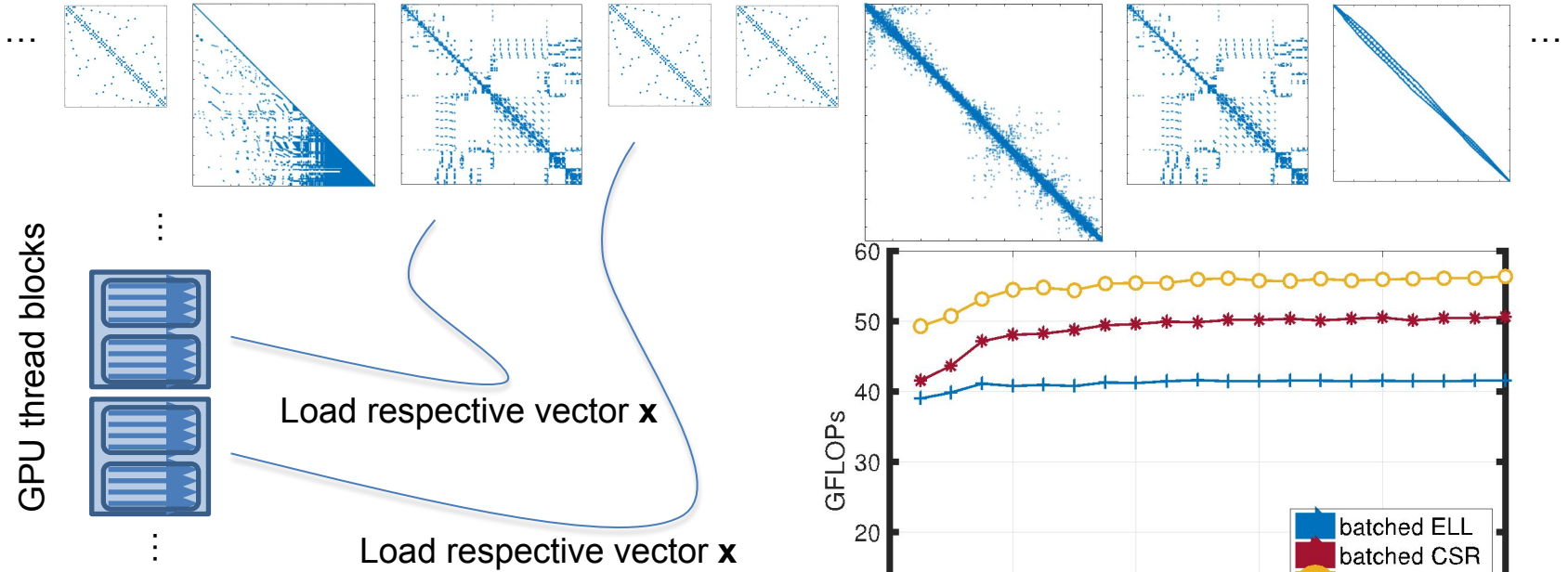




- All matrices stored in the same format
- Keep vectors in shared memory

Block-level SpMV

- What if we process many different matrices at a time? (Assume they are all small…)



Preconditioning

- Block-Jacobi preconditioner
  - Gauss-Jordan inversion
  - LU decomposition
  - Gauss-Huard decomposition
- Adaptive Block-Jacobi

Anzt et al., *Variable-size batched Gauss-Jordan elimination for block-Jacobi preconditioning on graphics processors*, ParCo 2019

Anzt et al., *Variable-size batched Gauss-Huard for block-Jacobi preconditioning*, ICCS 2017

Anzt et al., *Variable-size batched LU for small matrices and its integration into block-Jacobi preconditioning*, ICPP 2017

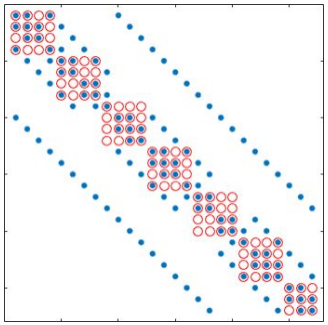
Anzt et al., *Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers*, CCPE 2019

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30

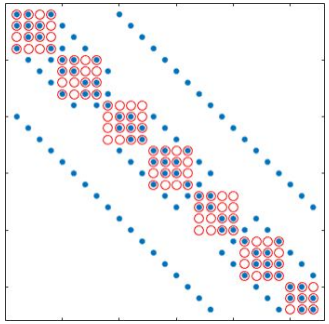


- Use only diagonal blocks for approximation

$$\mathrm{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \mathrm{diag}(D_1, \ldots, D_k)$$

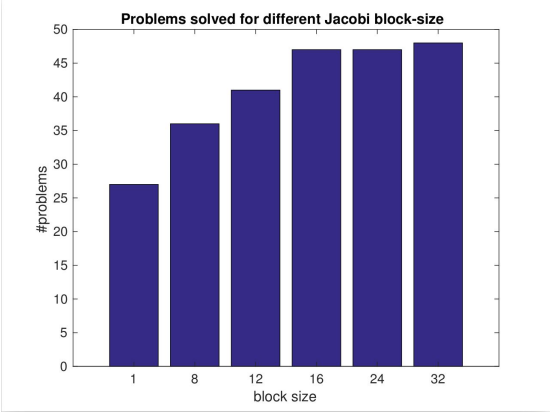
Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



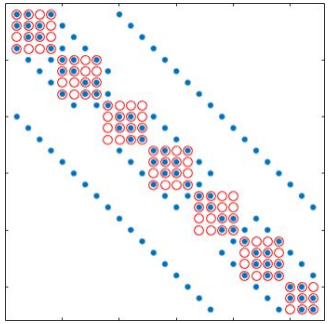
- Use only diagonal blocks for approximation

$$\text{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \text{diag}(D_1, \ldots, D_k)$$



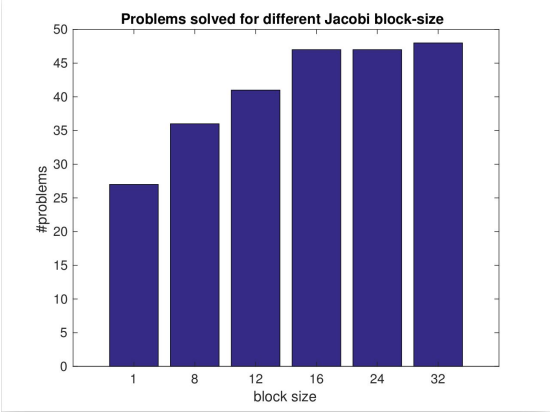
Block-Jacobi preconditioning

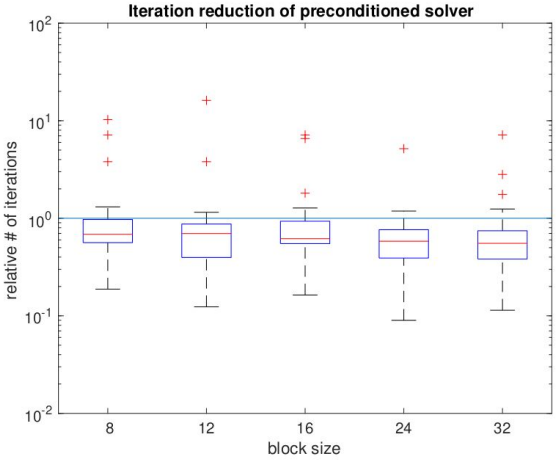
- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



- Use only diagonal blocks for approximation

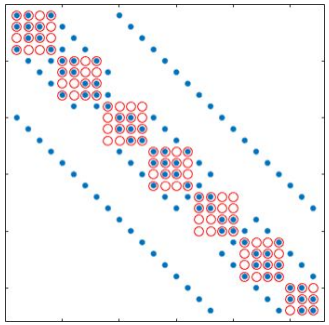
$$\text{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \text{diag}(D_1, \ldots, D_k)$$





Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



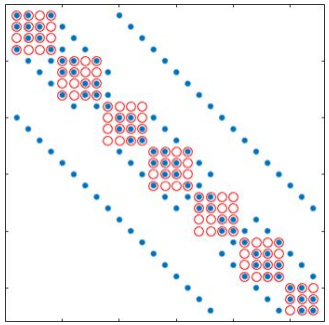
- Use only diagonal blocks for approximation

$$\mathrm{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \mathrm{diag}(D_1, \ldots, D_k)$$

$$y := M^{-1} z \quad \longrightarrow \quad y_i := D_i^{-1} z_i, \ \forall i$$

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



- Use only diagonal blocks for approximation

$$\text{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \text{diag}(D_1, \ldots, D_k)$$

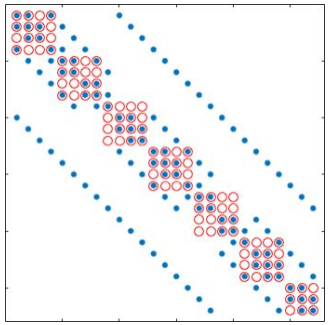
$$y := M^{-1} z \longrightarrow y_i := D_i^{-1} z_i, \ \forall i$$

$$\widetilde{D}_i := \text{inv}(D_i)$$

$$y_i := \widetilde{D}_i z_i$$

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



- Use only diagonal blocks for approximation

$$\text{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \text{diag}(D_1, \ldots, D_k)$$

$$y := M^{-1} z \quad \rightarrow \quad y_i := D_i^{-1} z_i, \ \forall i$$

$$\widetilde{D}_i := \text{inv}(D_i) \quad \textbf{Setup}$$

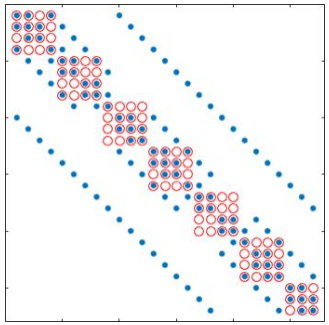
$$y_i := \widetilde{D}_i z_i \quad \textbf{Application}$$

Inversion-based block-Jacobi

inv = Gauss-Jordan elimination

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



- Use only diagonal blocks for approximation

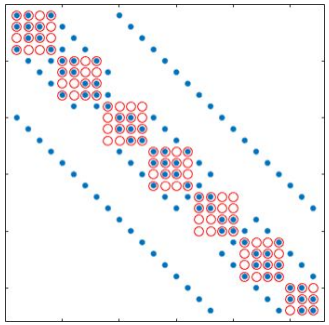
$$\text{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \text{diag}(D_1, \ldots, D_k)$$

$$y := M^{-1} z \quad \longrightarrow \quad y_i := D_i^{-1} z_i, \ \forall i$$

$$\longrightarrow \quad D_i y_i = z_i$$

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



- Use only diagonal blocks for approximation

$$\text{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \text{diag}(D_1, \ldots, D_k)$$

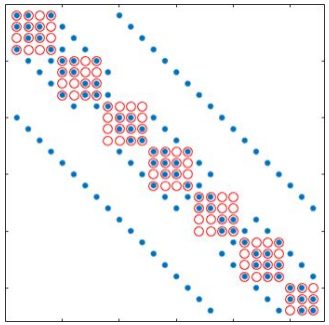
$$y := M^{-1} z \longrightarrow y_i := D_i^{-1} z_i, \ \forall i$$

$$\longrightarrow D_i y_i = z_i \longrightarrow D_i = L_i U_i$$

$$U_i y_i = w_i$$
$$L_i w_i = z_i$$

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



- Use only diagonal blocks for approximation

$$\text{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \text{diag}(D_1, \ldots, D_k)$$

$$y := M^{-1} z \longrightarrow y_i := D_i^{-1} z_i, \ \forall i$$

$$\longrightarrow D_i y_i = z_i \longrightarrow$$

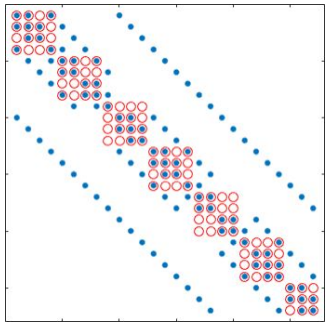
$$D_i = L_i U_i \quad \text{Setup}$$

$$U_i y_i = w_i$$
$$L_i w_i = z_i \quad \text{Application}$$

Decomposition-based block-Jacobi

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



- Use only diagonal blocks for approximation

$$\mathrm{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \mathrm{diag}(D_1, \ldots, D_k)$$

$$y := M^{-1} z \longrightarrow y_i := D_i^{-1} z_i, \ \forall i$$

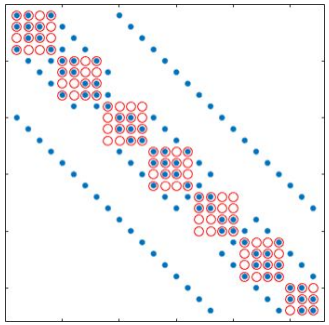
$$\longrightarrow D_i y_i = z_i \longrightarrow \widetilde{D}_i := \mathrm{gh}_d(D_i)$$

$$z_i := \mathrm{gh}_a(\widetilde{D}_i, y_i)$$

gh = Gauss-Huard decomposition / application

Block-Jacobi preconditioning

- Improve performance for problems with inherent block structure
  - Usually blocks are smaller than 30x30



- Use only diagonal blocks for approximation

$$\text{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \text{diag}(D_1, \ldots, D_k)$$

$$y := M^{-1} z \longrightarrow y_i := D_i^{-1} z_i, \ \forall i$$

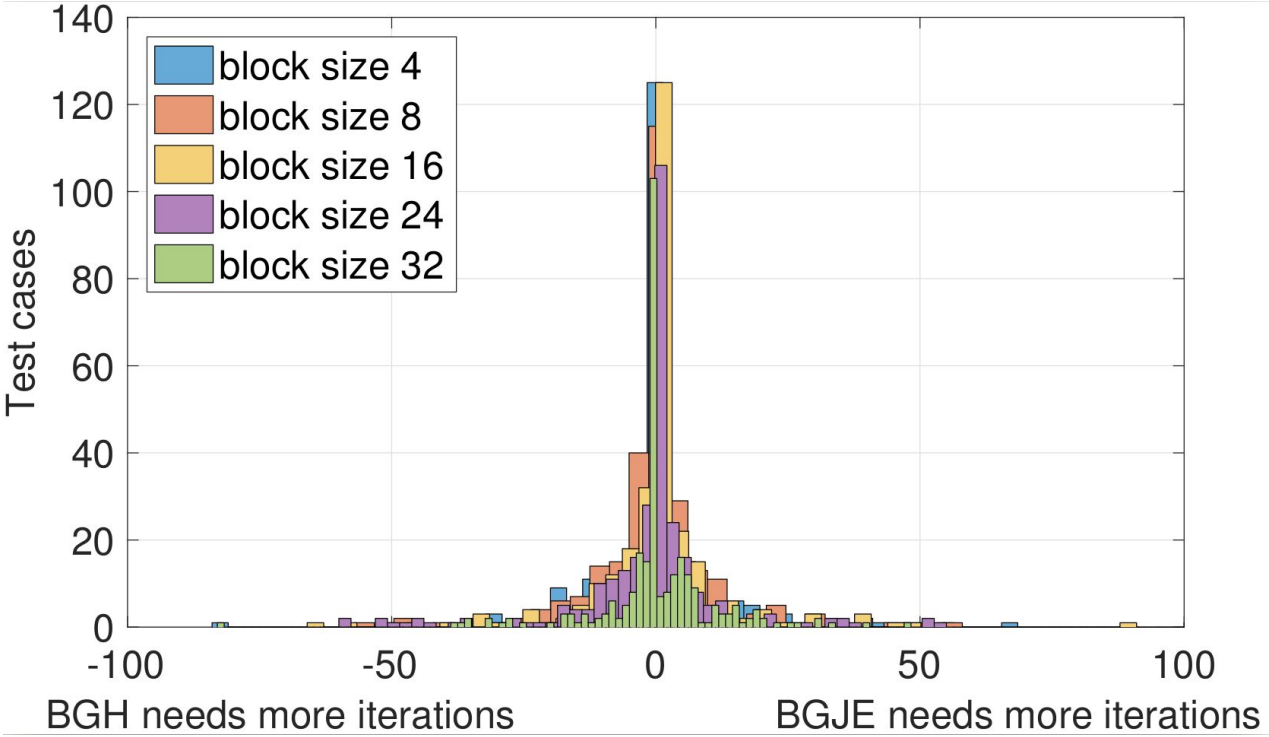
$$\longrightarrow D_i y_i = z_i \longrightarrow$$

$$\widetilde{D}_i := \text{gh}_d(D_i) \quad \text{Setup}$$

$$z_i := \text{gh}_a(\widetilde{D}_i, y_i) \quad \text{Application}$$

Decomposition-based block-Jacobi

Inversion vs decomposition stability



Block-Jacobi preconditioning

Inversion-based
($2n^3$ flops + $2n^2$ flops / iteration)

Decomposition-based
($2/3n^3$ flops + $2n^2$ flops / iteration)

Block-Jacobi preconditioning

Setup

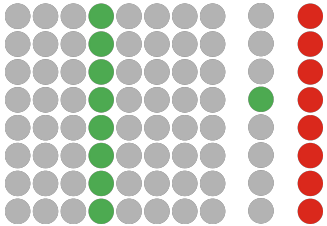
Gauss-Jordan inversion

Inversion-based
($2n^3$ flops + $2n^2$ flops / iteration)

Decomposition-based
($2/3n^3$ flops + $2n^2$ flops / iteration)

Application

matrix-vector multiply



n x AXPY

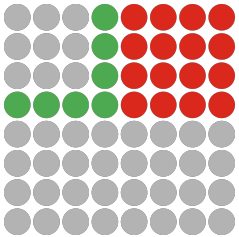
- read
- write

Block-Jacobi preconditioning

Setup

Gauss-Jordan inversion

Gauss-Huard decomposition

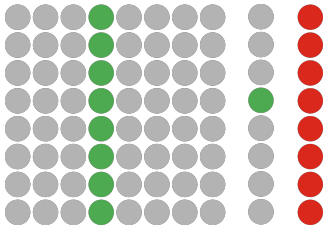


Inversion-based
($2n^3$ flops + $2n^2$ flops / iteration)

Decomposition-based
($2/3n^3$ flops + $2n^2$ flops / iteration)

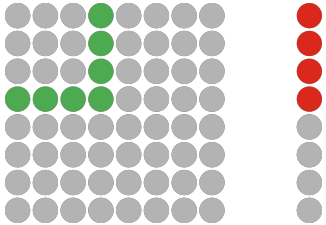
Application

matrix-vector multiply



n x AXPY

Gauss-Huard solve



n x AXPY
n x DOT

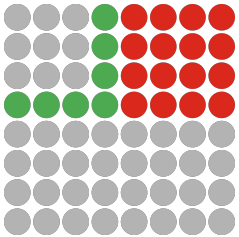
- read - write

Block-Jacobi preconditioning

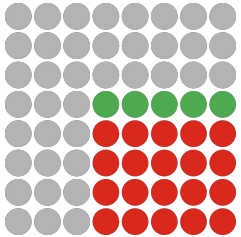
Setup

Gauss-Jordan inversion

Gauss-Huard decomposition



LU factorization

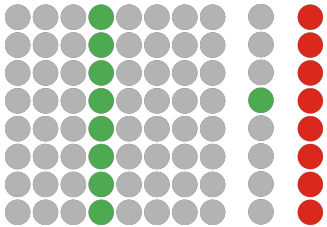


Inversion-based
($2n^3$ flops + $2n^2$ flops / iteration)

Decomposition-based
($2/3n^3$ flops + $2n^2$ flops / iteration)

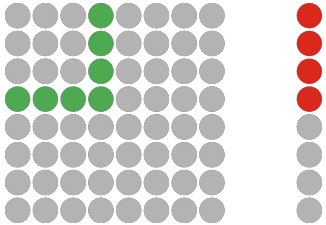
Application

matrix-vector multiply



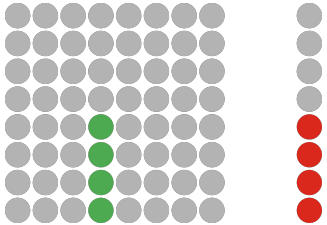
n x AXPY

Gauss-Huard solve



n x AXPY
n x DOT

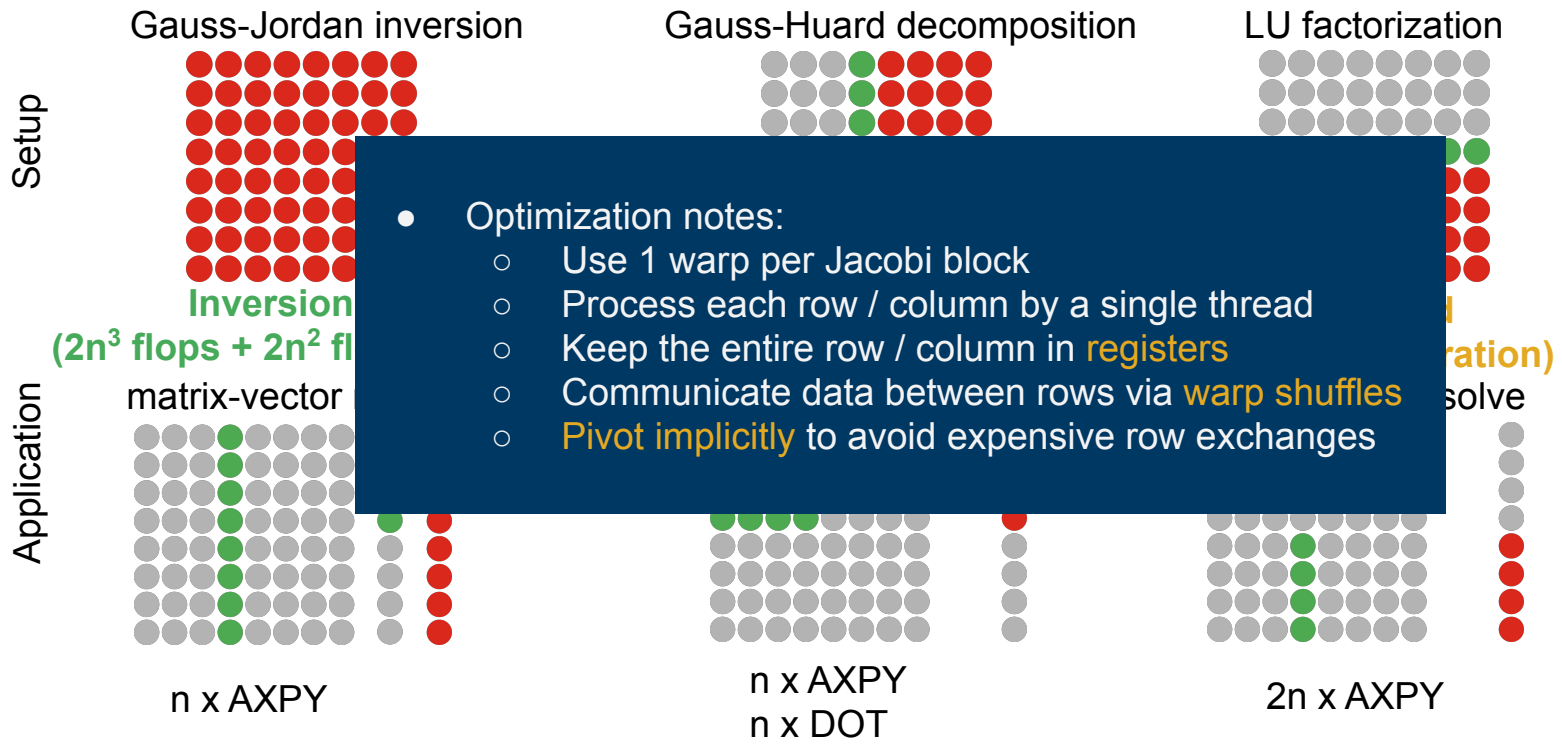
2x triangular solve



2n x AXPY

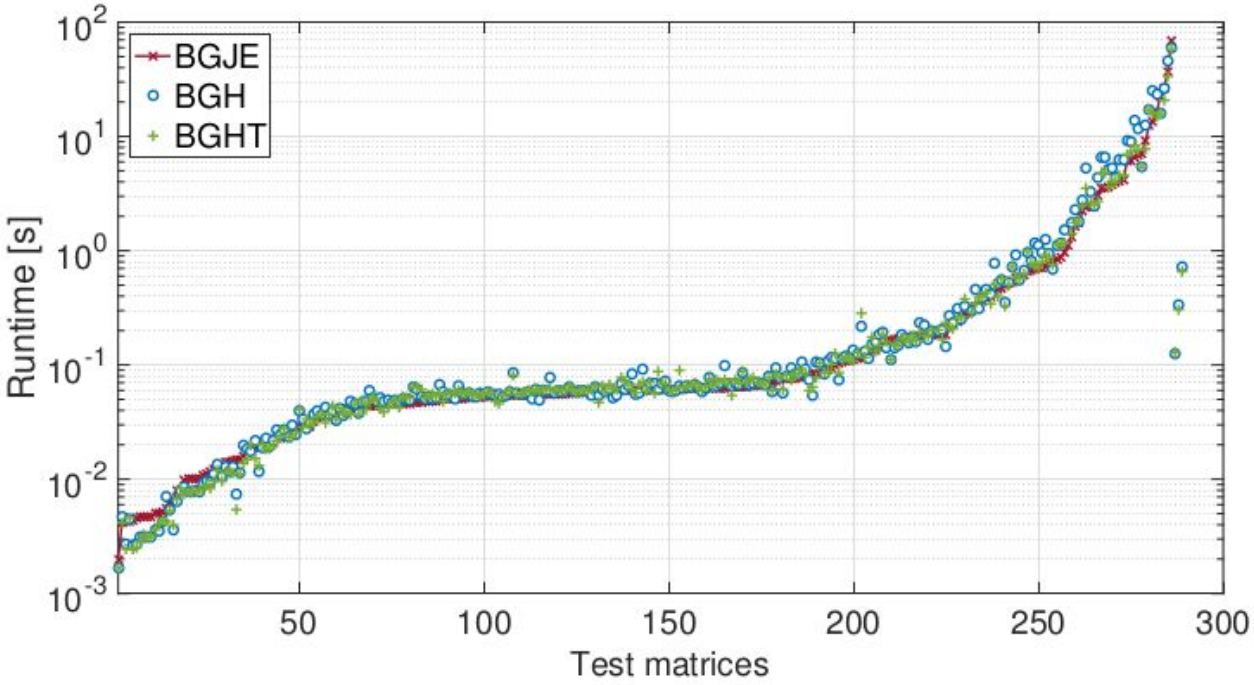
- read - write

Block-Jacobi preconditioning

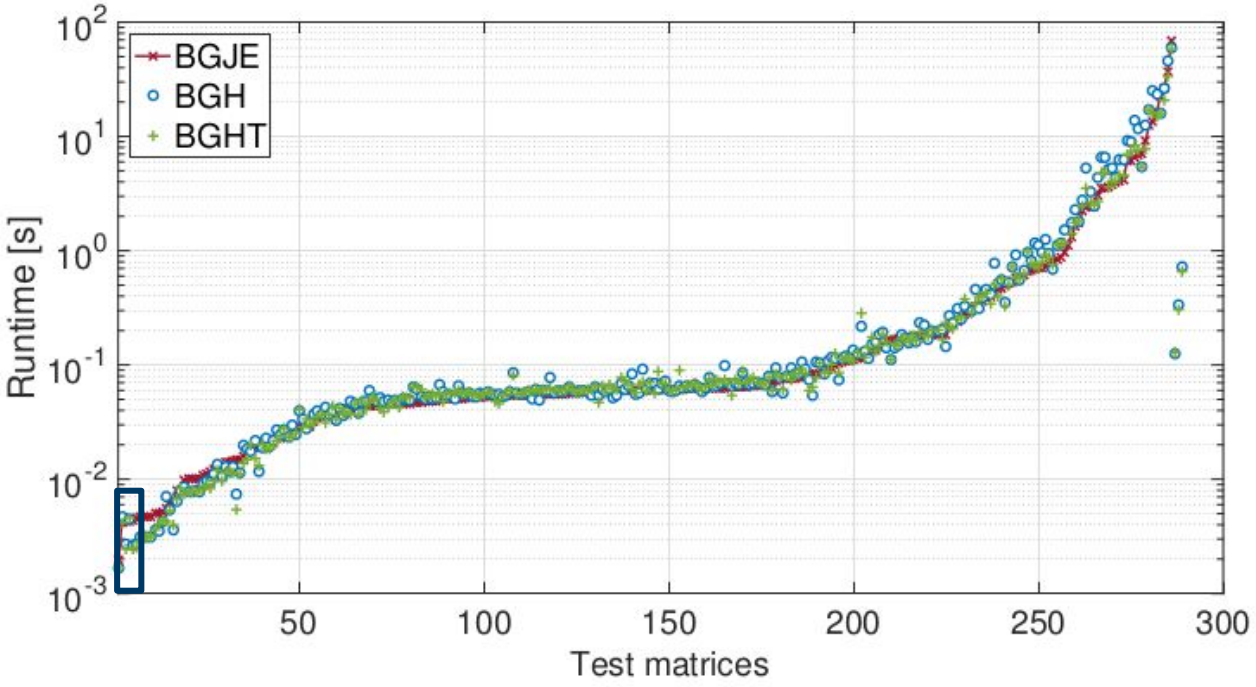


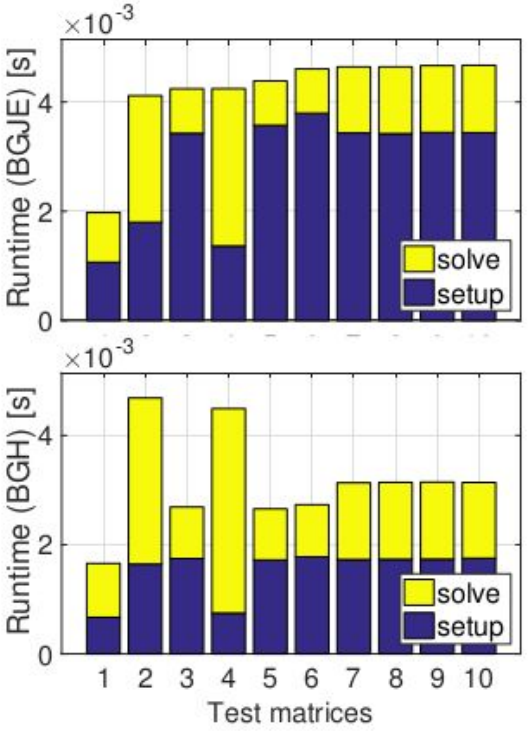
- Optimization notes:
  - Use 1 warp per Jacobi block
  - Process each row / column by a single thread
  - Keep the entire row / column in registers
  - Communicate data between rows via warp shuffles
  - Pivot implicitly to avoid expensive row exchanges

Performance



Performance





Adaptive precision block-Jacobi

Preconditioner is an approximation to the system matrix

- Applying a preconditioner inherently carries an error
- accuracy for block-Jacobi is usually around 0.1 - 0.01

$$z := M^{-1}y \approx A^{-1}y$$

Adaptive precision block-Jacobi

Preconditioner is an approximation to the system matrix

- Applying a preconditioner inherently carries an error
- accuracy for block-Jacobi is usually around 0.1 - 0.01

$$z := M^{-1}y \approx A^{-1}y$$

Preconditioner application is memory bound

- Most of the cost comes from reading the matrix from memory
- Idea: use lower precision to store the matrix

Adaptive precision block-Jacobi

Preconditioner is an approximation to the system matrix

- Applying a preconditioner inherently carries an error
- accuracy for block-Jacobi is usually around 0.1 - 0.01

$$z := M^{-1}y \approx A^{-1}y$$

Preconditioner application is memory bound

- Most of the cost comes from reading the matrix from memory
- Idea: use lower precision to store the matrix

Adaptive precision in inversion-based block-Jacobi:

- All computation is done in double precision
- Preconditioner matrix is stored in lower precision, with roundoff error “u”
- Error bound:

$$\frac{\|\delta z_i\|}{\|z_i\|} \leq u\,\kappa(D_i)$$

Adaptive precision block-Jacobi

Preconditioner is an approximation to the system matrix

- Applying a preconditioner inherently carries an error
- accuracy for block-Jacobi is usually around 0.1 - 0.01

$$z := M^{-1}y \approx A^{-1}y$$

Preconditioner application is memory bound

- Most of the cost comes from reading the matrix from memory
- Idea: use lower precision to store the matrix

Adaptive precision in inversion-based block-Jacobi:

- All computation is done in double precision
- Preconditioner matrix is stored in lower precision, with roundoff error “u”
- Error bound:

$$\frac{\|\delta z_i\|}{\|z_i\|} \leq u\,\kappa(D_i)$$

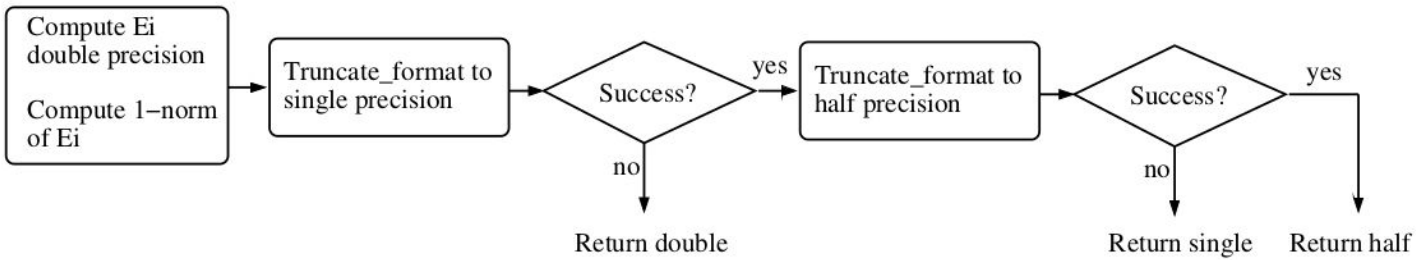
Choose the precision for each block **independently,** such that at least 1-2 digits of the result are correct

Handling overflow and underflow

- Storing the block in lower precision can cause underflow or overflow in some values.
- This is not accounted for by the numerical analysis shown in the previous slides.

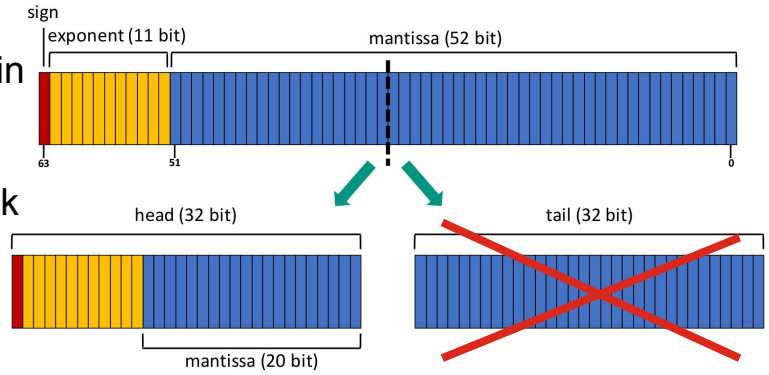
Handling overflow and underflow

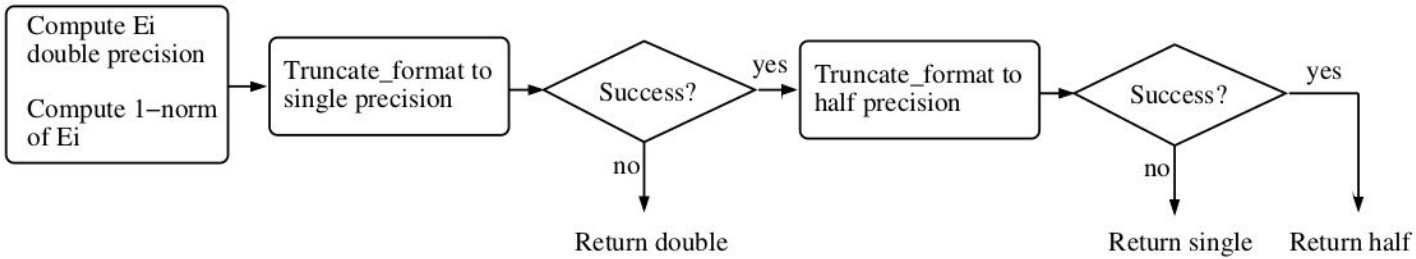
- Storing the block in lower precision can cause underflow or overflow in some values.
- This is not accounted for by the numerical analysis shown in the previous slides.
- Two options:
  1. Check the condition number of the low precision block to verify there were no catastrophic overflows or underflows.



Handling overflow and underflow

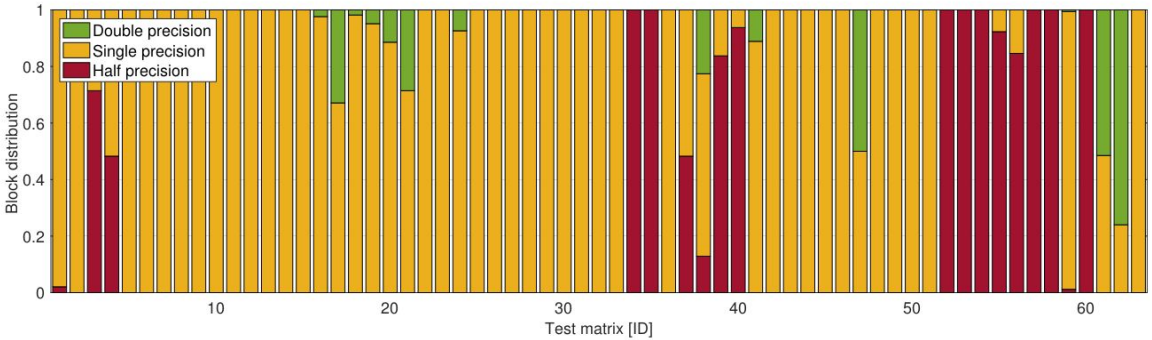
- Storing the block in lower precision can cause underflow or overflow in some values.
- This is not accounted for by the numerical analysis shown in the previous slides.
- Two options:
  1. Check the condition number of the low precision block to verify there were no catastrophic overflows or underflows.
  2. Use a custom storage format that preserves the number of exponent bits.





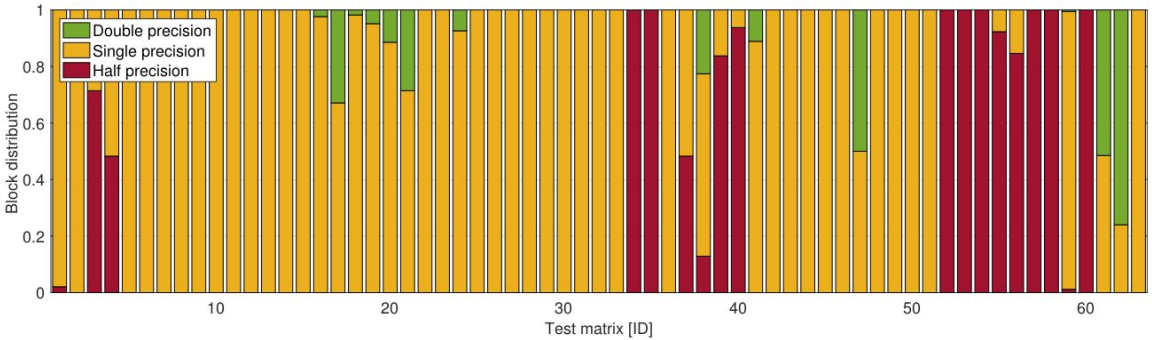
Experiments

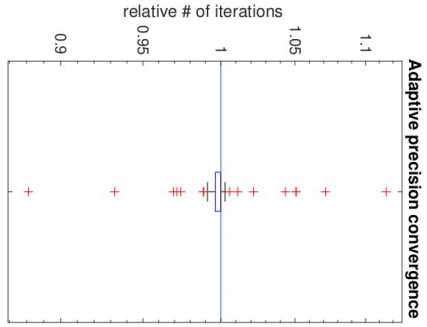
Can use fp32 for most blocks, and fp16 for some matrices:



Experiments

Can use fp32 for most blocks, and fp16 for some matrices:

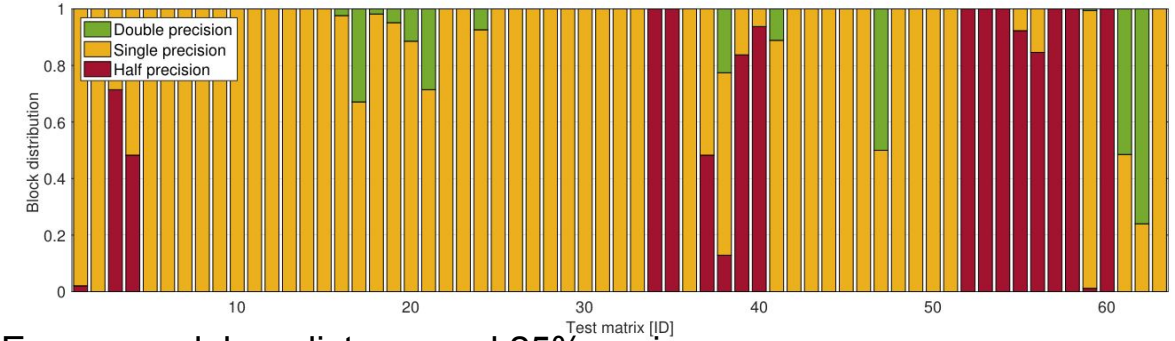


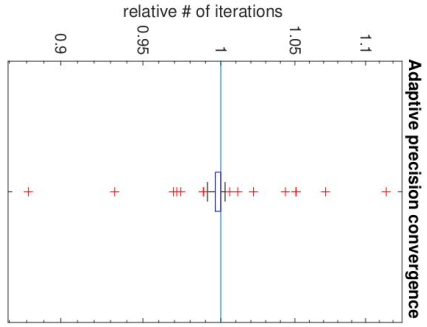


Usually < 5% iteration increase

Experiments

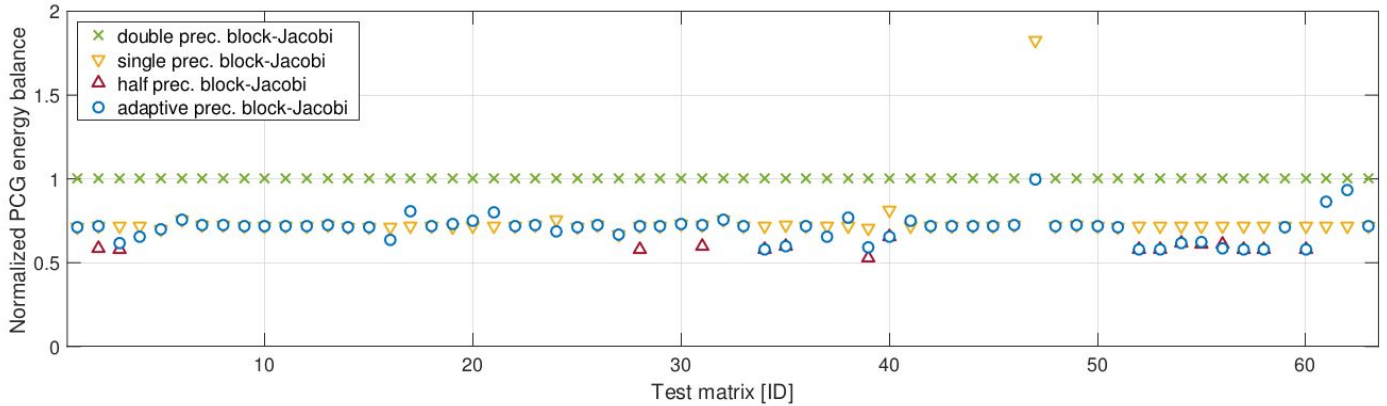
Can use fp32 for most blocks, and fp16 for some matrices:





Usually < 5% iteration increase

Energy model predicts around 25% savings:



Library Design for Sparse Computations

- Linear operators
- Operator factories
- The Ginkgo library

Interface of components for iterative solvers

Matrix

$$x = Ab$$

* can be realized as a (non-linear) approximation

Interface of components for iterative solvers

Matrix

$$x = Ab$$

Preconditioner

$$x = M^{-1}b$$

$$M \approx A$$

Interface of components for iterative solvers

Matrix

$$x = Ab$$

Preconditioner

$$x = M^{-1}b$$

$$M \approx A$$
$$M^{-1} = \Pi(A)$$

Interface of components for iterative solvers

Matrix

$$x = Ab$$

Preconditioner

$$x = M^{-1}b$$

$$M \approx A$$
$$M^{-1} = \Pi(A)$$

“Solver" operator

$$Ax = b$$

Interface of components for iterative solvers

Matrix	Preconditioner	“Solver" operator
		$Ax = b$
$x = Ab$	$x = M^{-1}b$	$x = Sb$
	$M \approx A$	$S = A^{-1}$
	$M^{-1} = \Pi(A)$	

Interface of components for iterative solvers

Matrix	Preconditioner	“Solver" operator
		$Ax = b$
$x = Ab$	$x = M^{-1}b$	$x = Sb$
	$M \approx A$	$S = A^{-1}$
	$M^{-1} = \Pi(A)$	$S = \Sigma(A)$

Interface of components for iterative solvers

Matrix	Preconditioner	“Solver" operator
		$Ax = b$
$x = Ab$	$x = M^{-1}b$	$x = Sb$
	$M \approx A$	$S = A^{-1}$
	$M^{-1} = \Pi(A)$	$S = \Sigma(A)$

All of them can be expressed as:

- applications of linear operators* (LinOp)

$$L : \mathbb{F}^m \rightarrow \mathbb{F}^n$$

* can be realized as a (non-linear) approximation

Interface of components for iterative solvers

Matrix	Preconditioner	“Solver" operator
		$Ax = b$
$x = Ab$	$x = M^{-1}b$	$x = Sb$
	$M \approx A$	$S = A^{-1}$
	$M^{-1} = \Pi(A)$	$S = \Sigma(A)$

All of them can be expressed as:

- applications of linear operators* (LinOp)
- (non-linear) transformations applied to linear operators (Factory)

$$L : \mathbb{F}^m \rightarrow \mathbb{F}^n$$
$$\Phi : \mathbb{L}^{mn}(\mathbb{F}) \rightarrow \mathbb{L}^{kl}(\mathbb{F})$$

* can be realized as a (non-linear) approximation

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

$$\mathrm{A_{CSR}} \quad \mathrm{b_D} \quad \mathrm{x_D}$$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

$$A_{CSR} \quad b_D \quad x_D$$

2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

$$BJ$$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

$$A_{CSR} \quad b_D \quad x_D$$

2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

$$BJ$$

3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)

$$CG_{BJ}$$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

$$\mathrm{A_{CSR}} \quad \mathrm{b_D} \quad \mathrm{x_D}$$

2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

$$\mathrm{BJ}$$

3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)

$$\mathrm{CG_{BJ}}$$

4. Generate the solver operator from the system matrix using the solver factory (preprocessing needed for the solver and preconditioner)

$$\mathrm{S_A} = \mathrm{CG_{BJ}}(\mathrm{A_{CSR}})$$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right-hand side and initial guess from file, standard input, or use existing data from the application

$$\mathrm{A_{CSR}} \quad \mathrm{b_D} \quad \mathrm{x_D}$$

2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

$$\mathrm{BJ}$$

3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)

$$\mathrm{CG_{BJ}}$$

4. Generate the solver operator from the system matrix using the solver factory (preprocessing needed for the solver and preconditioner)

$$\mathrm{S_A} = \mathrm{CG_{BJ}}(\mathrm{A_{CSR}})$$

5. Solve the system (actual computation)

$$\mathrm{x_D} = \mathrm{S_A}\,\mathrm{b_D}$$

The Ginkgo library

Ginkgo

https://ginkgo-project.github.io

Karlsruhe Institute of Technology

THE UNIVERSITY OF TENNESSEE KNOXVILLE

- Open-source C++ framework for sparse linear algebra.

The Ginkgo library

Ginkgo

https://ginkgo-project.github.io

Karlsruhe Institute of Technology

THE UNIVERSITY OF TENNESSEE KNOXVILLE

- Open-source C++ framework for sparse linear algebra.
- Sparse linear solvers, preconditioners, SpMV etc.

The Ginkgo library

Ginkgo

https://ginkgo-project.github.io

Karlsruhe Institute of Technology

THE UNIVERSITY OF TENNESSEE KNOXVILLE

- Open-source C++ framework for sparse linear algebra.
- Sparse linear solvers, preconditioners, SpMV etc.
- Generic algorithm implementation:
  - reference kernels for checking correctness;
  - architecture-specific highly optimized kernels.

The Ginkgo library

Ginkgo

https://ginkgo-project.github.io

Karlsruhe Institute of Technology

THE UNIVERSITY OF TENNESSEE KNOXVILLE

- Open-source C++ framework for sparse linear algebra.
- Sparse linear solvers, preconditioners, SpMV etc.
- Generic algorithm implementation:
  - reference kernels for checking correctness;
  - architecture-specific highly optimized kernels.
- Focused on GPU accelerators (i.e. NVIDIA GPUs).

The Ginkgo library

Ginkgo

https://ginkgo-project.github.io

Karlsruhe Institute of Technology

THE UNIVERSITY OF TENNESSEE KNOXVILLE

- Open-source C++ framework for sparse linear algebra.
- Sparse linear solvers, preconditioners, SpMV etc.
- Generic algorithm implementation:
  - reference kernels for checking correctness;
  - architecture-specific highly optimized kernels.
- Focused on GPU accelerators (i.e. NVIDIA GPUs).
- Software quality and sustainability efforts guided by xSDK community policies:

https://xsdk.info

The Ginkgo library

Ginkgo

https://ginkgo-project.github.io

Karlsruhe Institute of Technology

THE UNIVERSITY OF TENNESSEE KNOXVILLE

- Open-source C++ framework for sparse linear algebra.
- Sparse linear solvers, preconditioners, SpMV, etc.
- Generic algorithm implementation:
  - reference kernels for checking correctness;
  - architecture-specific highly optimized kernels.
- Focused on GPU accelerators (i.e. NVIDIA GPUs).
- Software quality and sustainability efforts guided by xSDK community policies:

https://xsdk.info

```
// Create solver
auto solver = Cg<>::build()
  .with_preconditioner(Jacobi<>::build().with_max_block_size(32).on(gpu))
  .with_criteria(
      Iteration::build().with_max_iters(1000u).on(gpu),
      ResidualNormReduction<>::build().with_reduction_factor(1e-9).on(gpu))
  .on(gpu);
// Solve system
solver->generate(give(A))->apply(lend(b), lend(x));
```

Summary and open research lines

Summary and open lines

- Handle irregularity by trading computation for load balancing.

Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.

Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.

Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.
- Combine preconditioning with reduced precision to gain performance without sacrificing accuracy.

Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.
- Combine preconditioning with reduced precision to gain performance without sacrificing accuracy.
- Design a common interface for solver components for flexibility and reusability.

Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.
- Combine preconditioning with reduced precision to gain performance without sacrificing accuracy.
- Design a common interface for solver components for flexibility and reusability.
- Extend to multi-GPU systems.
  - Maybe with runtime system help?
  - Terry Cojean @ KIT

Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.
- Combine preconditioning with reduced precision to gain performance without sacrificing accuracy.
- Design a common interface for solver components for flexibility and reusability.
- Extend to multi-GPU systems.
  - Maybe with runtime system help?
  - Terry Cojean @ KIT
- Implement block-level variants of other SpMV algorithms.

Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.
- Combine preconditioning with reduced precision to gain performance without sacrificing accuracy.
- Design a common interface for solver components for flexibility and reusability.
- Extend to multi-GPU systems.
  - Maybe with runtime system help?
  - Terry Cojean @ KIT
- Implement block-level variants of other SpMV algorithms.
- Accelerate other preconditioners (e.g. ILU)
  - Tobias Ribizel @ KIT

Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.
- Combine preconditioning with reduced precision to gain performance without sacrificing accuracy.
- Design a common interface for solver components for flexibility and reusability.
- Extend to multi-GPU systems.
  - Maybe with runtime system help?
  - Terry Cojean @ KIT
- Implement block-level variants of other SpMV algorithms.
- Accelerate other preconditioners (e.g. ILU)
  - Tobias Ribizel @ KIT
- Explore adaptive precision in the context of other preconditioners and solvers.
  - Thomas Grützmacher @ KIT

Summary and open lines

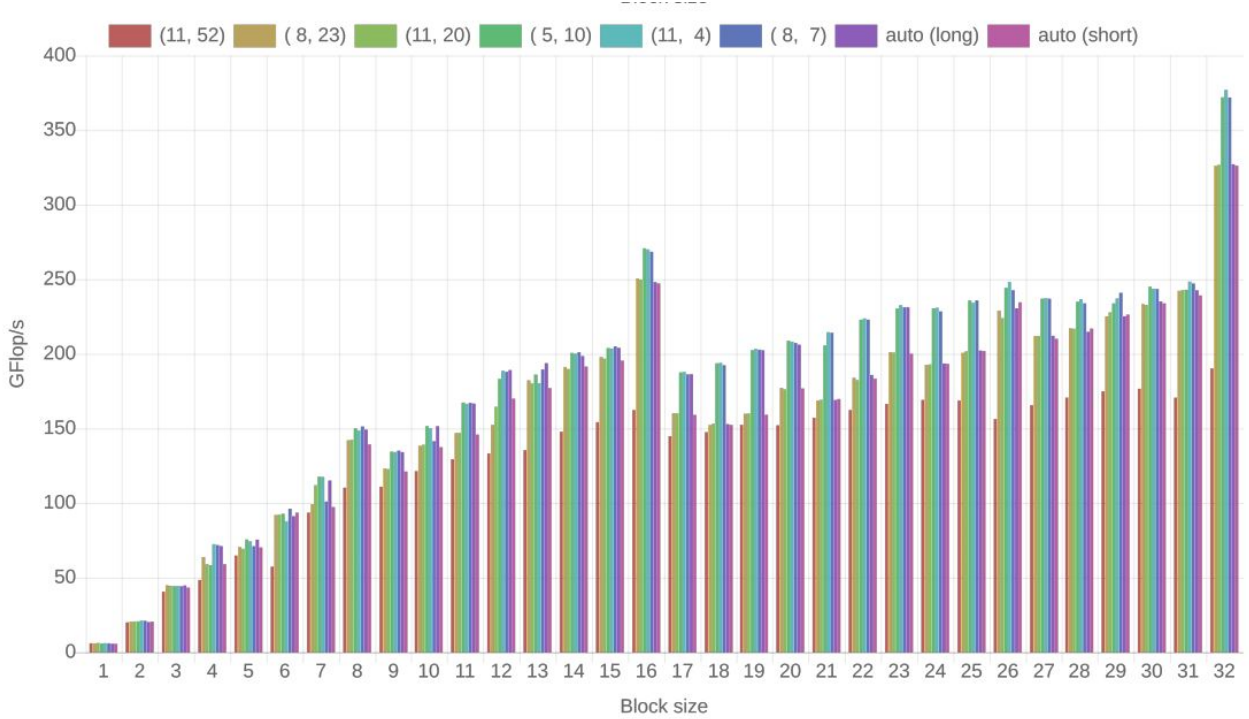
- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.
- Combine preconditioning with reduced precision to gain performance without sacrificing accuracy.
- Design a common interface for solver components for flexibility and reusability.
- Extend to multi-GPU systems.
  - Maybe with runtime system help?
  - Terry Cojean @ KIT
- Implement block-level variants of other SpMV algorithms.
- Accelerate other preconditioners (e.g. ILU)
  - Tobias Ribizel @ KIT
- Explore adaptive precision in the context of other preconditioners and solvers.
  - Thomas Grützmacher @ KIT
- Use these components in distributed applications.
  - Pratik Nayak @ KIT

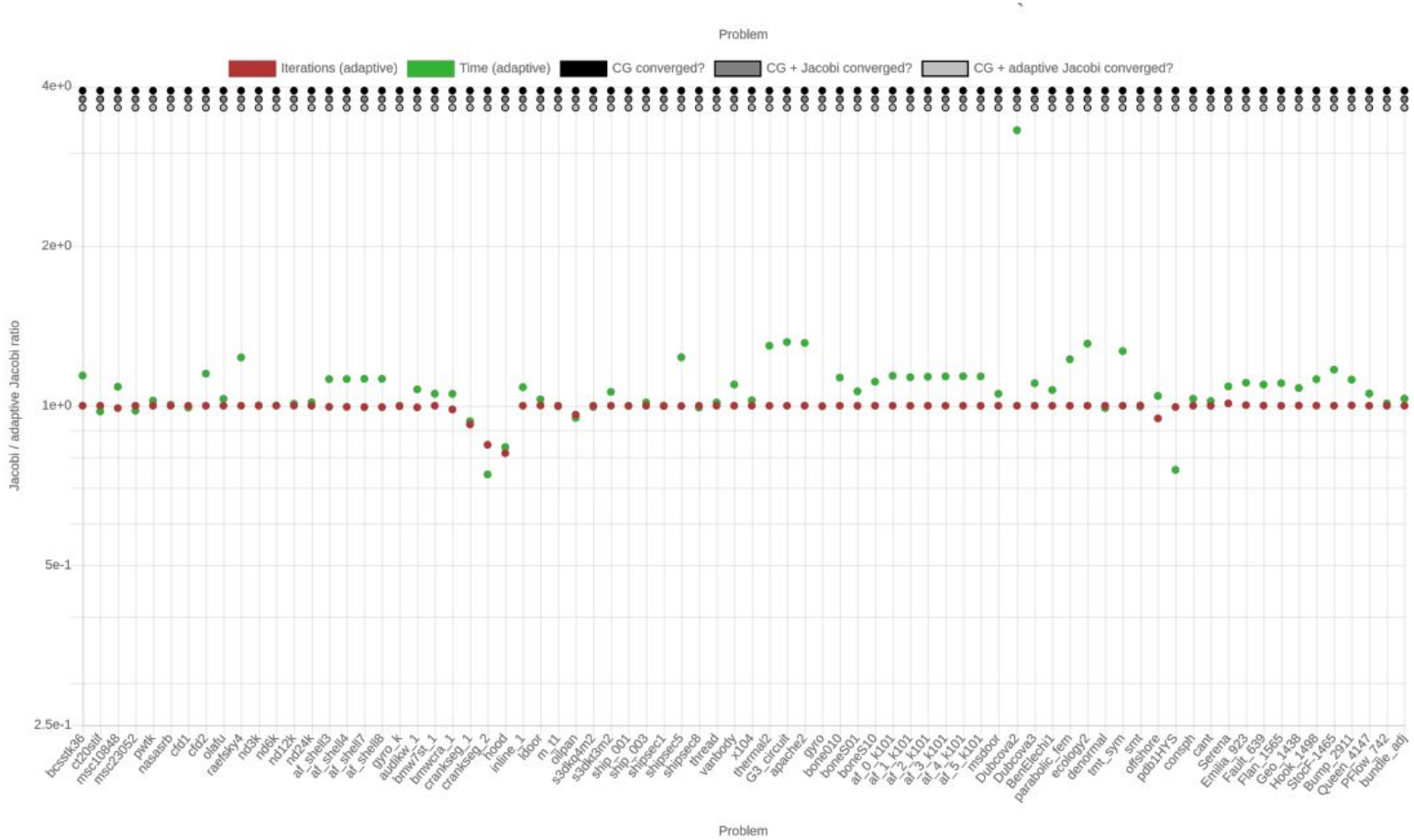
Summary and open lines

- Handle irregularity by trading computation for load balancing.
- Design components that target lower levels of the hierarchy (blocks, warps, threads, etc.) to process (many) small problems.
- Use highly-parallel preconditioners to accelerate Krylov solvers on GPUs.
- Combine preconditioning with reduced precision to gain performance without sacrificing accuracy.
- Design a common interface for solver components for flexibility and reusability.

- Extend to multi-GPU systems.
  - Maybe with runtime system help?
  - Terry Cojean @ KIT
- Implement block-level variants of other SpMV algorithms.
- Accelerate other preconditioners (e.g. ILU)
  - Tobias Ribizel @ KIT
- Explore adaptive precision in the context of other preconditioners and solvers.
  - Thomas Grützmacher @ KIT
- Use these components in distributed applications.
  - Pratik Nayak @ KIT

Thank you!
Questions?


(11, 52)
(8, 23)
(11, 20)
(5, 10)
(11, 4)
(8, 7)
auto (long)
auto (short)
GFlop/s
0
50
100
150
200
250
300
350
400
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Block size


Problem
Iterations (adaptive)
Time (adaptive)
CG converged?
CG + Jacobi converged?
CG + adaptive Jacobi converged?
4e+0
2e+0
1e+0
5e-1
2.5e-1
Jacobi / adaptive Jacobi ratio
bcsstk36
ct20stif
msc10848
msc23052
pwtk
nasasrb
cfd1
cfd2
oilfu
raefsky4
nd3k
nd6k
nd12k
nd24k
af_shell3
af_shell4
af_shell7
af_shell8
gyro_k
audikw_1
bmw7st_1
bmwcra_1
crankseg_1
crankseg_2
hood
inline_1
ldoor
m_t1
oilpan
s3dkq4m2
s3dkt3m2
ship_001
ship_003
shipsec1
shipsec5
shipsec8
thread
vanbody
x104
thermal2
G3_circuit
apache2
gyro
bone010
boneS01
boneS10
af_0_k101
af_1_k101
af_2_k101
af_3_k101
af_4_k101
af_5_k101
msdoor
Dubcova2
Dubcova3
BenElechi1
parabolic_fem
ecology2
denormal
tmt_sym
smt
offshore
pdb1HYS
consph
cant
Serena
Emilia_923
Fault_639
Flan_1565
Geo_1438
Hook_1498
StocF-1465
Bump_2911
Queen_4147
PFlow_742
bundle_adj
Problem

SpMV formats

Sparse matrix formats

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

Sparse matrix formats

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7

Sparse matrix formats

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

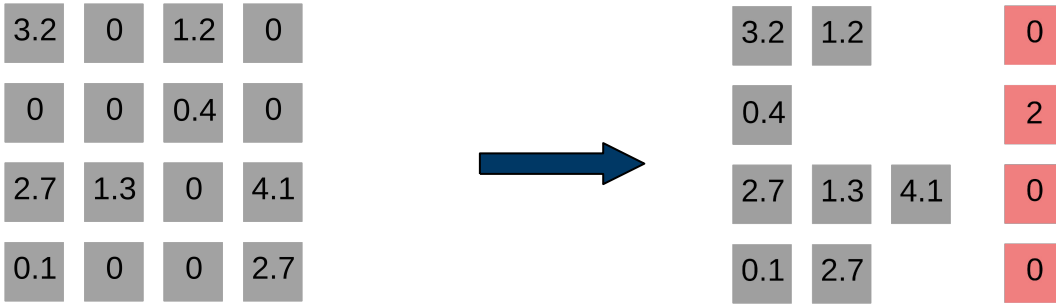
3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	0	1	2	2	2	3	3
0	2	2	0	1	3	0	3

row

column

COO

Sparse matrix formats

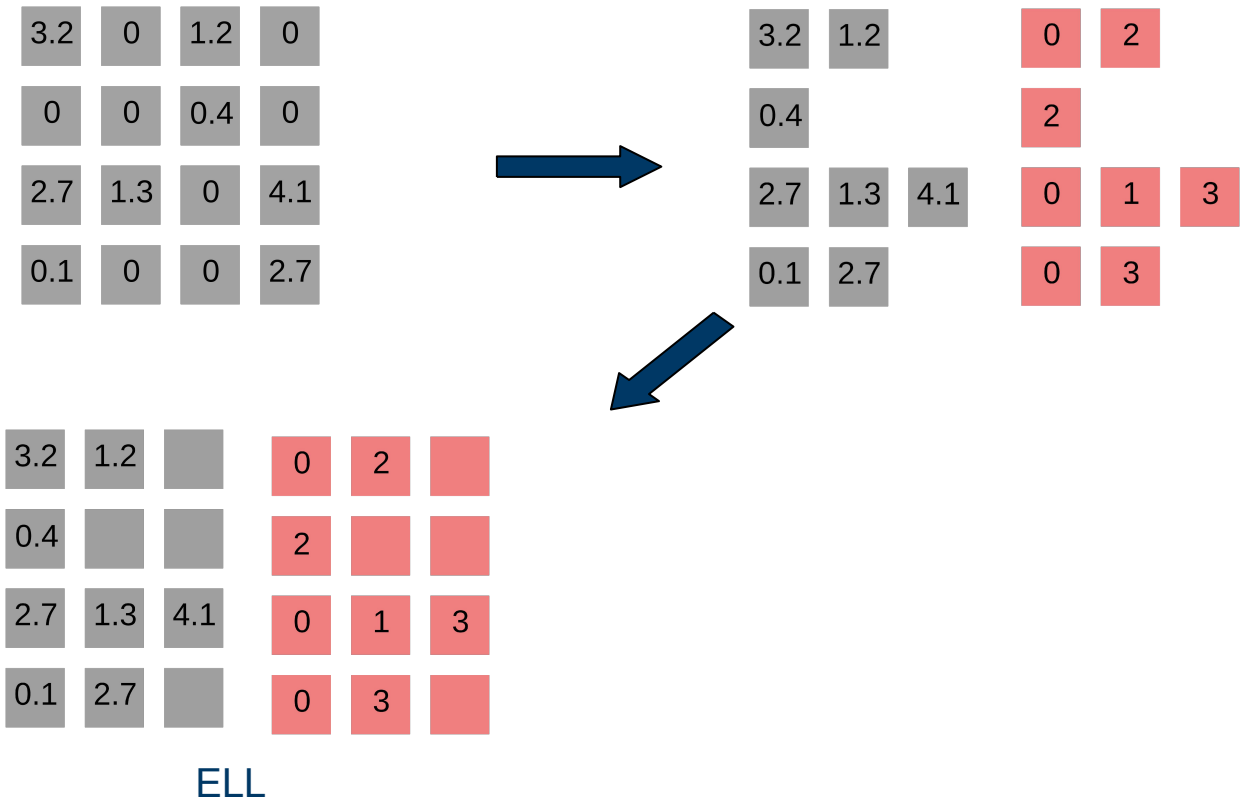


Sparse matrix formats

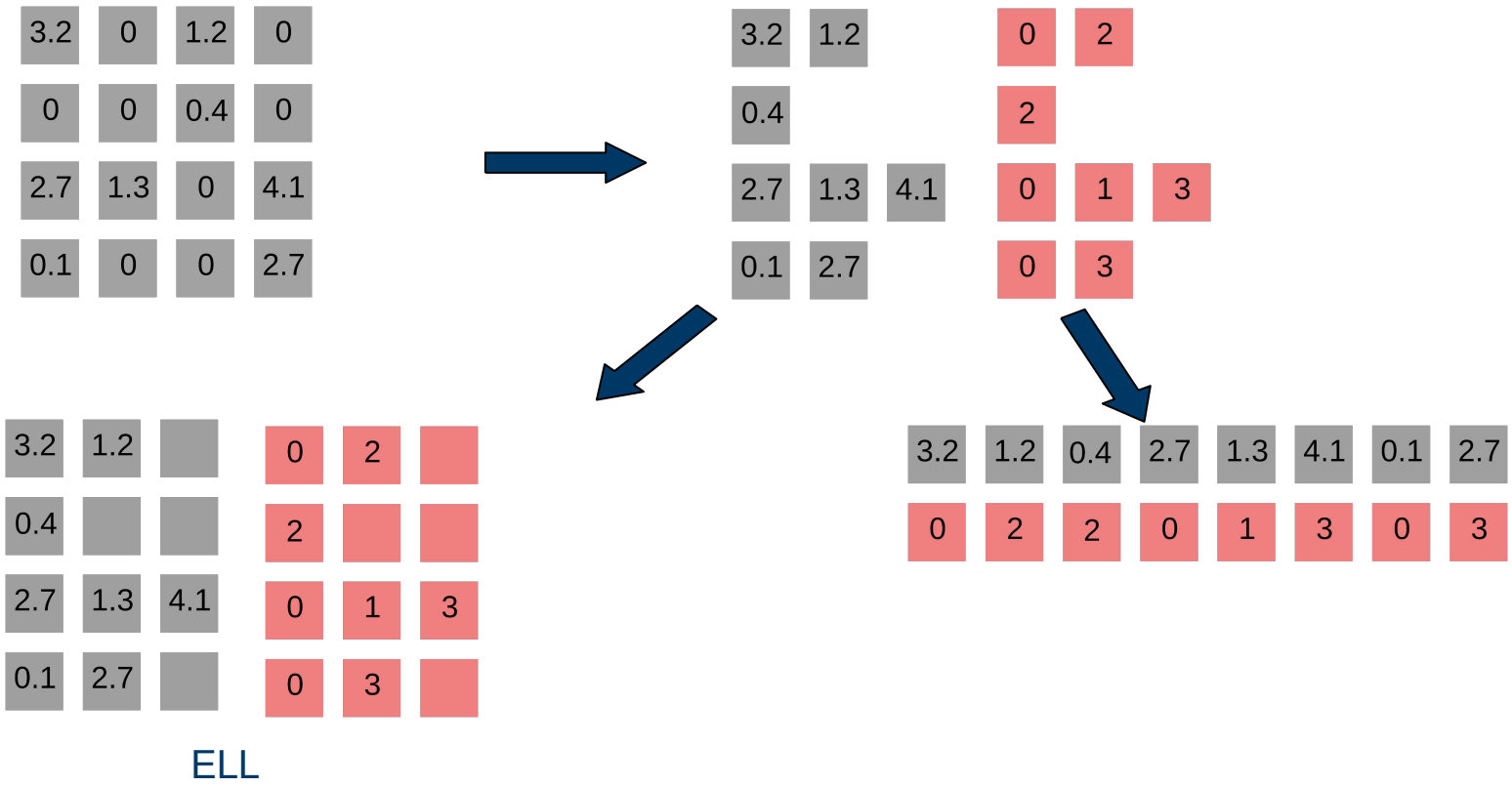
3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

3.2	1.2		0	2	
0.4			2		
2.7	1.3	4.1	0	1	3
0.1	2.7		0	3	

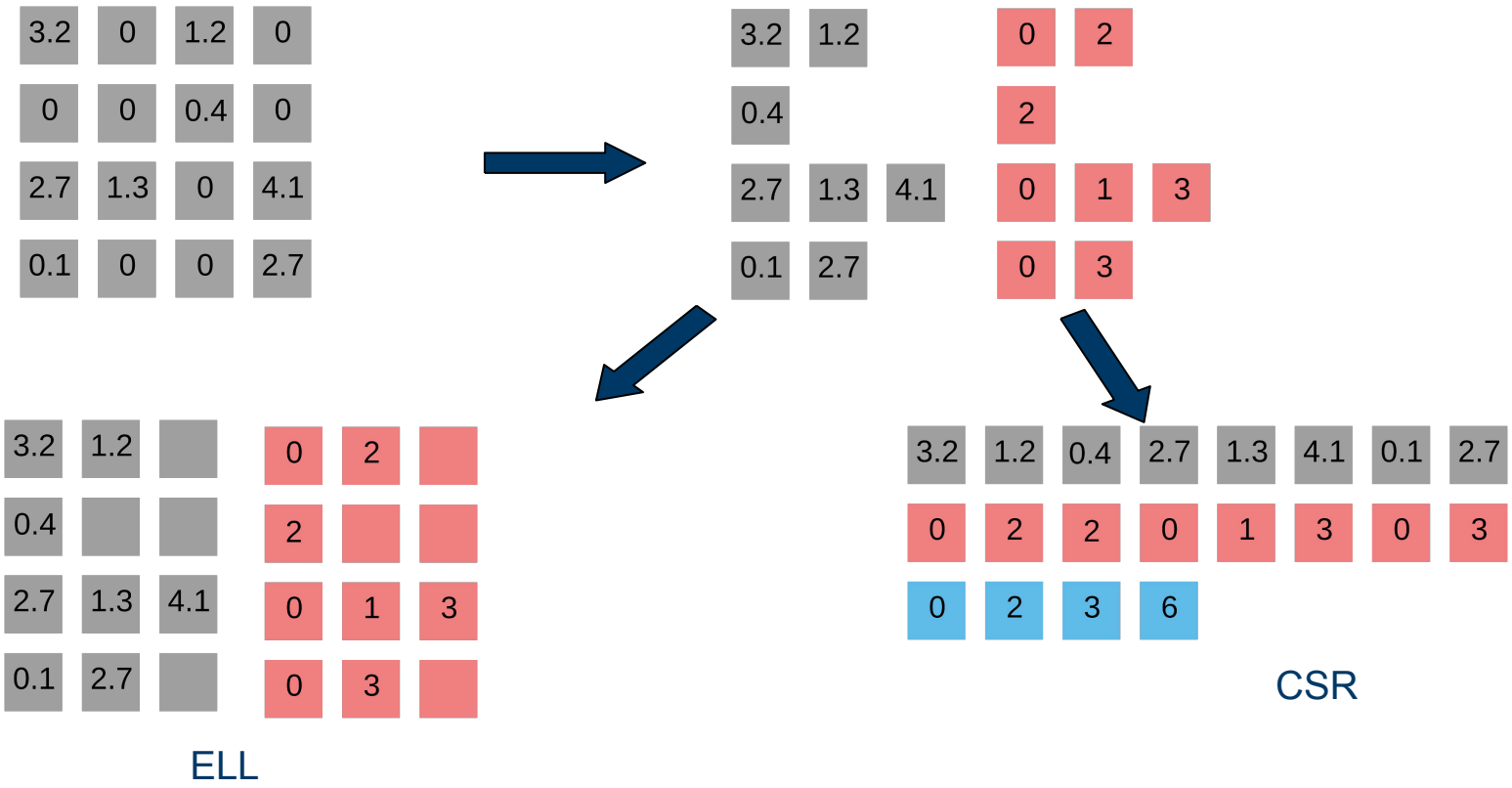
Sparse matrix formats



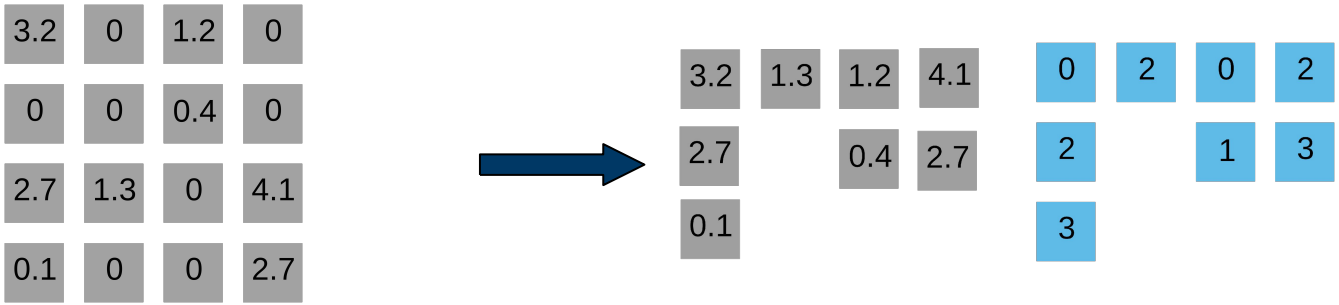
Sparse matrix formats



Sparse matrix formats

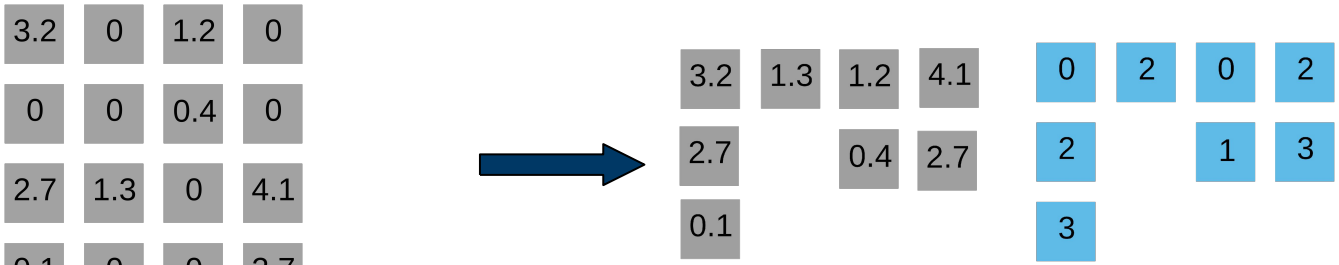


Sparse matrix formats

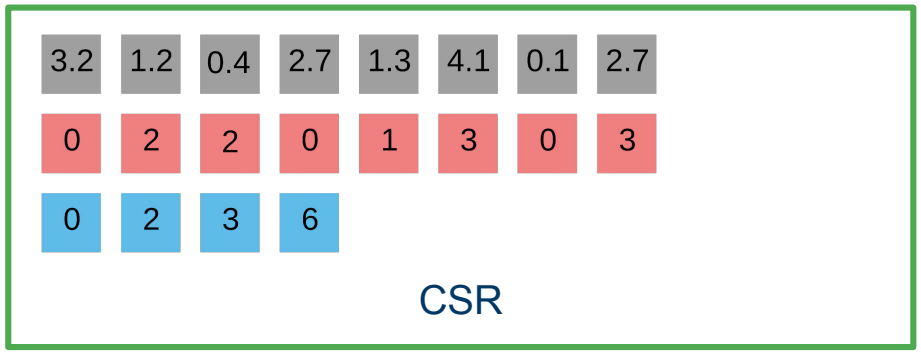


… leads to CSC

Sparse matrix formats



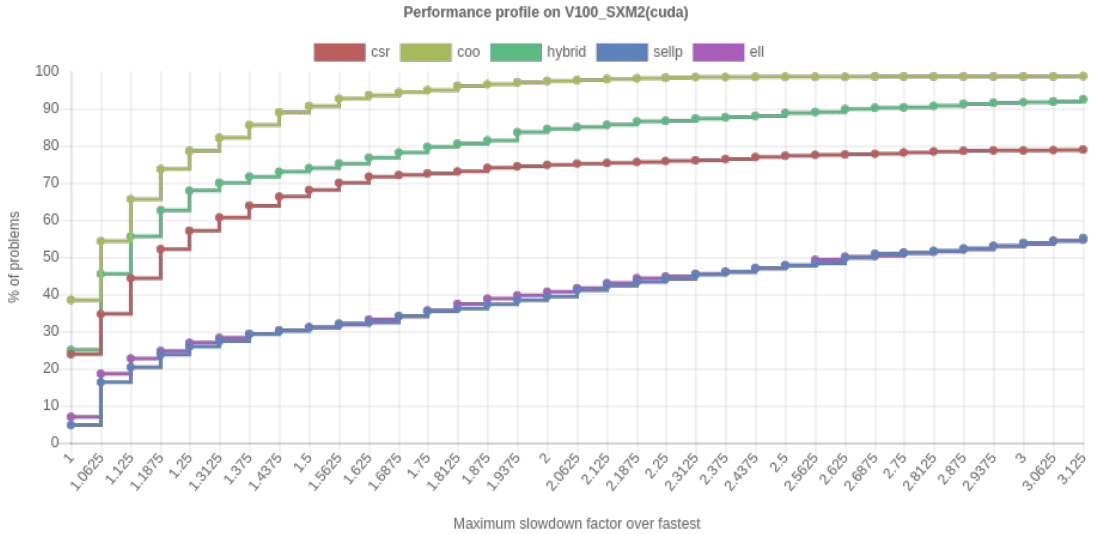
… leads to CSC



“Standard” approach

First things first

THERE IS NO “BEST” SPARSE MATRIX FORMAT / SpMV ALGORITHM



What Is Preconditioning

Preconditioning

$$Ax = b, \;\; A \in \mathbb{R}^{n \times n}$$

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \quad \Longrightarrow \quad M^{-1}Ax = M^{-1}b$$

- Replace the original system with an equivalent preconditioned system

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \quad \Longrightarrow \quad M^{-1}Ax = M^{-1}b$$

- Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \quad \Longrightarrow \quad M^{-1}Ax = M^{-1}b$$

- Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

~~$M^{-1}A$~~

- **Do not compute the preconditioned system matrix explicitly!**

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \quad \Longrightarrow \quad M^{-1}Ax = M^{-1}b$$

- Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

~~$M^{-1}A$~~

$$y := (M^{-1}A)x$$

- **Do not compute the preconditioned system matrix explicitly!**

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \quad \Rightarrow \quad M^{-1}Ax = M^{-1}b$$

- Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

~~$M^{-1}A$~~

- **Do not compute the preconditioned system matrix explicitly!**

$$y := (M^{-1}A)x$$

$$z := Ax$$

$$y := M^{-1}z$$

Preconditioner application

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \quad \Rightarrow \quad M^{-1}Ax = M^{-1}b$$

- Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

~~$M^{-1}A$~~

$$y := (M^{-1}A)x$$

- **Do not compute the preconditioned system matrix explicitly!**

Generate the preconditioner matrix, and store it in a form suitable for application

$$A \rightsquigarrow M^{-1}$$

Preconditioner setup

$$z := Ax$$

$$y := M^{-1}z$$

Preconditioner application

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \quad \Rightarrow \quad M^{-1}Ax = M^{-1}b$$

- Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

~~$M^{-1}A$~~

$$y := (M^{-1}A)x$$

- **Do not compute the preconditioned system matrix explicitly!**

Generate the preconditioner matrix, and store it in a form suitable for application

$$A \rightsquigarrow M^{-1}$$

Generation via factory

$$z := Ax$$

$$y := M^{-1}z$$

Linear operator application

Ginkgo

linear operator abstraction

Library Design Example

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

A_{CSR} b_D x_D

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application
2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

A_{CSR} b_D x_D

BJ

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application
2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)
3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)

$\mathrm{A_{CSR}}$ $\mathrm{b_D}$ $\mathrm{x_D}$

BJ

$\mathrm{CG_{BJ}}$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application
2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)
3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)
4. Generate the solver operator from the system matrix using the solver factory (preprocessing needed for the solver and preconditioner)

$$\mathrm{A_{CSR}} \quad \mathrm{b_D} \quad \mathrm{x_D}$$

$$\mathrm{BJ}$$

$$\mathrm{CG_{BJ}}$$

$$\mathrm{S_A} = \mathrm{CG_{BJ}}(\mathrm{A_{CSR}})$$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application
2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)
3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)
4. Generate the solver operator from the system matrix using the solver factory (preprocessing needed for the solver and preconditioner)
5. Solve the system (actual computation)

$$\mathrm{A_{CSR}} \quad \mathrm{b_D} \quad \mathrm{x_D}$$

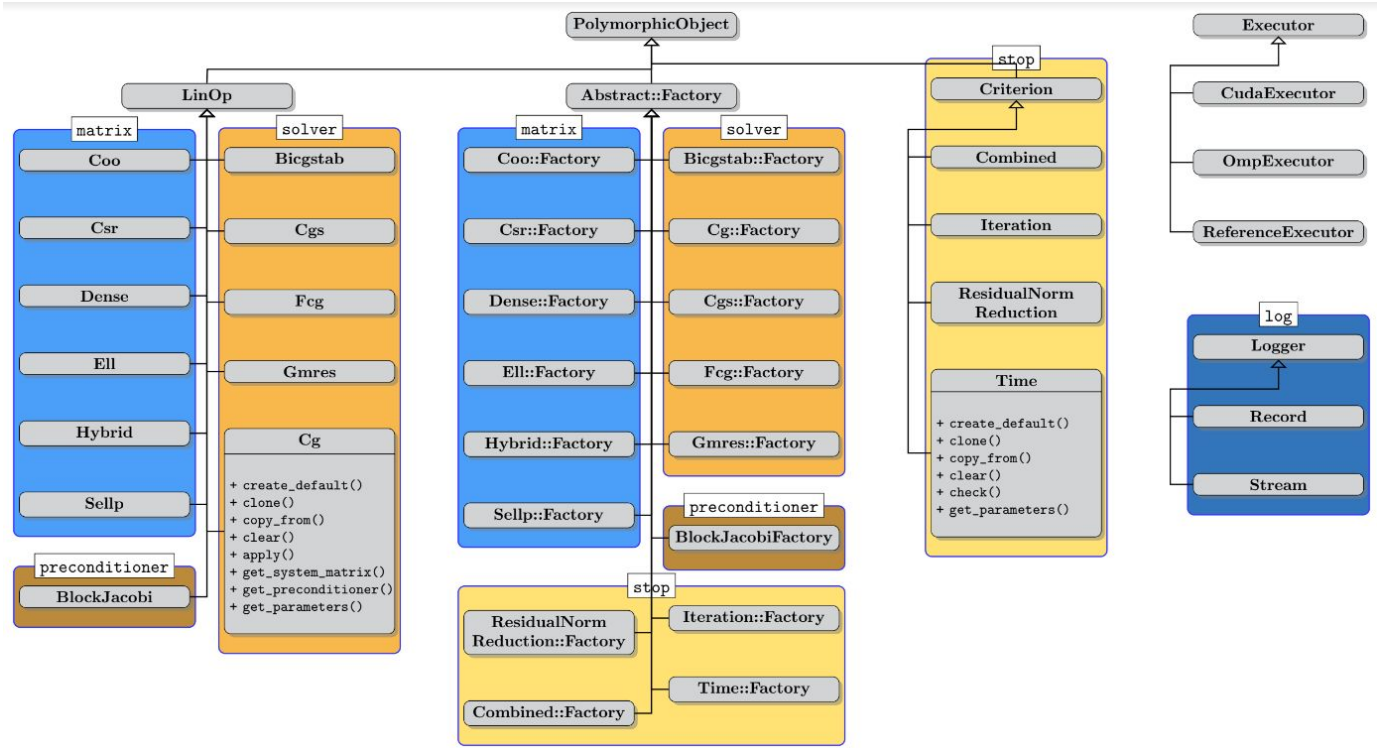
$$\mathrm{BJ}$$

$$\mathrm{CG_{BJ}}$$

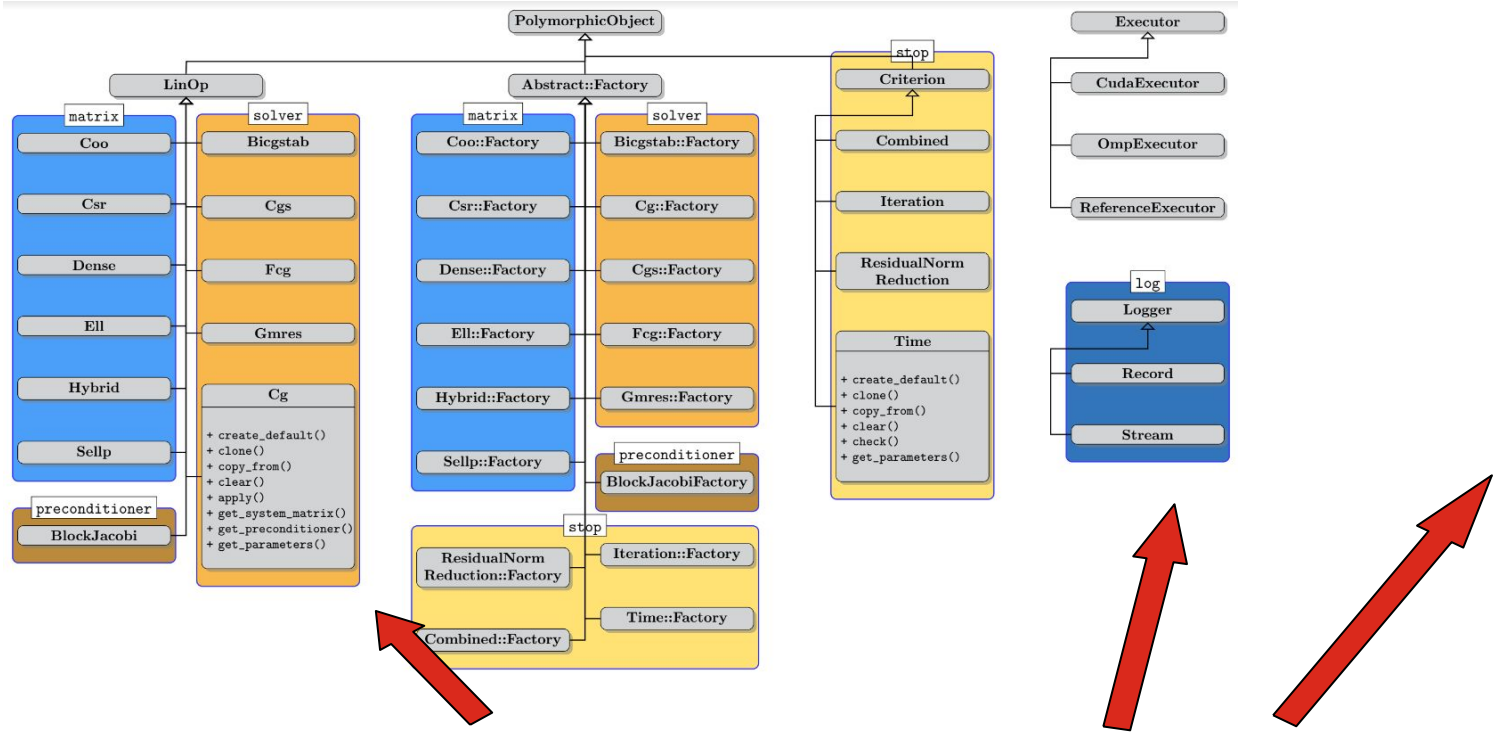
$$\mathrm{S_A} = \mathrm{CG_{BJ}}(\mathrm{A_{CSR}})$$

$$\mathrm{x_D} = \mathrm{S_A}\,\mathrm{b_D}$$

Library features



Library features: extensibility



users can provide new matrices, solvers, preconditioners, stopping criteria, loggers

Without recompiling the library!