Algebraic Programming for Graph & Machine Learning

Albert-Jan N. Yzelman Denis Jelovina Aristeidis Mastoras
Alberto Scolari

Computing Systems Laboratory, Zürich Research Center, Switzerland

4th of June, MS4F, ACM PASC, ETH Zürich, Switzerland

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Principles:

- explicitly **annotate** computations with algebraic information;

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Principles:

- explicitly **annotate** computations with algebraic information;
- compile-time **introspect**ion of algebraic information;

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Principles:

- explicitly **annotate** computations with algebraic information;
- compile-time **introspect**ion of algebraic information;
- auto-**optimise** code based on algebraic information;

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Principles:

- explicitly **annotate** computations with algebraic information;
- compile-time **introspect**ion of algebraic information;
- auto-**optimise** code based on algebraic information;
- **allow only** scalable expressions.

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;

2) structures:

3) primitives:

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;

2) structures: binary operators, monoids, and semirings; and

3) primitives:

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;

2) structures: binary operators, monoids, and semirings; and

3) primitives: eWiseApply, reduction (fold), mxv, mxm.

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;

2) structures: binary operators, monoids, and semirings; and

3) primitives: eWiseApply, reduction (fold), mxv, mxm.

Containers are simlar to the standard template library (STL):

```
grb::Vector< double > x( n ), y( n ), z( n );
grb::Matrix< double > A( n, n );
```

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;

2) structures: binary operators, monoids, and semirings; and

3) primitives: eWiseApply, reduction (fold), mxv, mxm.

Containers are simlar to the standard template library (STL):

```
grb::Vector< double > x( n ), y( n ), z( n );
grb::Matrix< double > A( n, n );
```

Elements may be **any POD type**:

```
grb::Vector< std::pair< int, double > > pairs( n );
```

Basics

Algebraic structures are types. E.g., min : $D_1 \times D_2 \rightarrow D_3$:

```
grb::operators::min< double, int, double > min;
```

Basics

Algebraic structures are types. E.g., min : $D_1 \times D_2 \rightarrow D_3$:

```
grb::operators::min< double, int, double > min;
```

More complex structures may be **composed**:

```
grb::Semiring<
    grb::operators::add< double >,
    grb::operators::mul< double >,
    grb::identities::zero,
    grb::identities::one
> plusTimes;
```

Basics

Algebraic structures are types. E.g., min : $D_1 \times D_2 \rightarrow D_3$:

```
grb::operators::min< double, int, double > min;
```

More complex structures may be **composed**:

```
grb::Semiring<
    grb::operators::add< double >,
    grb::operators::mul< double >,
    grb::identities::zero,
    grb::identities::one
> plusTimes;
```

Algebraic type traits: is_associative < OP >::value, is_commutative, ...

- type traits are C++11 **compile-time constants**

Basics

Algebraic primitives operate on algebraic containers:

- grb::set(x, 1.0); // $x_i = 1,\ \forall x_i \in x$
- grb::setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Basics

Algebraic primitives operate on algebraic containers:

- grb::set(x, 1.0); // $x_i = 1,\ \forall x_i \in x$
- grb::setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may depend on **algebraic structures**:

- grb::eWiseApply(z, x, x, min); // $z_i = \min\{x_i, x_i\},\ \forall i$

Basics

Algebraic primitives operate on algebraic containers:

- grb::set(x, 1.0); // $x_i = 1,\ \forall x_i \in x$
- grb::setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may depend on **algebraic structures**:

- grb::eWiseApply(z, x, x, min); // $z_i = \min\{x_i, x_i\},\ \forall i$
- grb::mxv(y, A, x, plusTimes); // $y \mathrel{+}= Ax$

Basics

Algebraic primitives operate on algebraic containers:

- grb::set(x, 1.0); // $x_i = 1,\ \forall x_i \in x$
- grb::setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may depend on **algebraic structures**:

- grb::eWiseApply(z, x, x, min); // $z_i = \min\{x_i, x_i\},\ \forall i$
- grb::mxv(y, A, x, plusTimes); // y += Ax

GraphBLAS also allows for **descriptors** and output **masking**. E.g.,

- Vector< bool > mask(n); setElement(mask, true, 0);
- mxv< transpose >(y, mask, A, x, plusTimes);,

Basics

Algebraic primitives operate on algebraic containers:

- grb::set(x, 1.0); // $x_i = 1,\ \forall x_i \in x$
- grb::setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may depend on **algebraic structures**:

- grb::eWiseApply(z, x, x, min); // $z_i = \min\{x_i, x_i\},\ \forall i$
- grb::mxv(y, A, x, plusTimes); // y += Ax

GraphBLAS also allows for **descriptors** and output **masking**. E.g.,

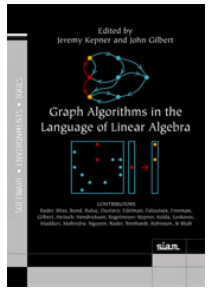
- Vector< bool > mask(n); setElement(mask, true, 0);
- mxv< transpose >(y, mask, A, x, plusTimes);,

computes $A^T x$ for the first element only, and adds it to y_0.

GraphBLAS

GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

- for example, $y = A^k x$, parametrised in a semiring:

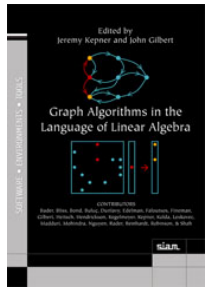



```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
            grb::Vector< VectorT >  &y,
      const grb::Matrix< NonzeroT > &A, const size_t k,
      const grb::Vector< VectorT >  &x,
            grb::Vector< VectorT >  &buffer,
      const Semiring                &ring = Semiring()
) {
      // error checking and error propagation omitted
      grb::vxm( y, x, A, ring );
      for( size_t i = 1; i < k; ++i ) {
            std::swap( y, buffer );
            grb::vxm( y, buffer, A, ring );
      }
}
```

GraphBLAS

GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

- for example, $y = A^k x$, parametrised in a semiring:




```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
            grb::Vector< VectorT >  &y,
      const grb::Matrix< NonzeroT > &A, const size_t k,
      const grb::Vector< VectorT >  &x,
            grb::Vector< VectorT >  &buffer,
      const Semiring                &ring = Semiring()
) {
      // error checking and error propagation omitted
      grb::vxm( y, x, A, ring );
      for( size_t i = 1; i < k; ++i ) {
            std::swap( y, buffer );
            grb::vxm( y, buffer, A, ring );
      }
}
```

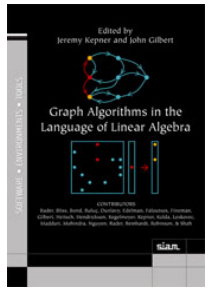
Solve different problems with different semirings:

- plus-times: numerical linear algebra,

GraphBLAS

GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

- for example, $y = A^k x$, parametrised in a semiring:




```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
            grb::Vector< VectorT >  &y,
      const grb::Matrix< NonzeroT > &A, const size_t k,
      const grb::Vector< VectorT >  &x,
            grb::Vector< VectorT >  &buffer,
      const Semiring                &ring = Semiring()
) {
      // error checking and error propagation omitted
      grb::vxm( y, x, A, ring );
      for( size_t i = 1; i < k; ++i ) {
            std::swap( y, buffer );
            grb::vxm( y, buffer, A, ring );
      }
}
```

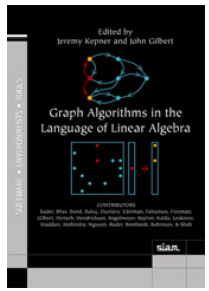
Solve different problems with different semirings:

- plus-times: numerical linear algebra,
- Boolean: reachability / connectivity,

GraphBLAS

GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

- for example, $y = A^k x$, parametrised in a semiring:




```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
            grb::Vector< VectorT >  &y,
      const grb::Matrix< NonzeroT > &A, const size_t k,
      const grb::Vector< VectorT >  &x,
            grb::Vector< VectorT >  &buffer,
      const Semiring                &ring = Semiring()
) {
      // error checking and error propagation omitted
      grb::vxm( y, x, A, ring );
      for( size_t i = 1; i < k; ++i ) {
            std::swap( y, buffer );
            grb::vxm( y, buffer, A, ring );
      }
}
```

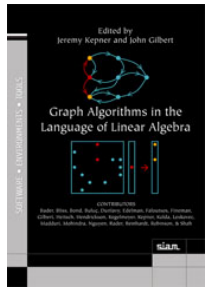
Solve different problems with different semirings:

- plus-times: numerical linear algebra,
- Boolean: reachability / connectivity,
- min-plus: shortest paths,

GraphBLAS

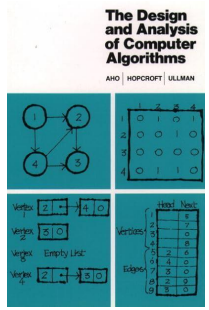
GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

- for example, $y = A^k x$, parametrised in a semiring:




```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
            grb::Vector< VectorT >  &y,
      const grb::Matrix< NonzeroT > &A, const size_t k,
      const grb::Vector< VectorT >  &x,
            grb::Vector< VectorT >  &buffer,
      const Semiring                &ring = Semiring()
) {
      // error checking and error propagation omitted
      grb::vxm( y, x, A, ring );
      for( size_t i = 1; i < k; ++i ) {
            std::swap( y, buffer );
            grb::vxm( y, buffer, A, ring );
      }
}
```

Solve different problems with different semirings:



- plus-times: numerical linear algebra,
- Boolean: reachability / connectivity,
- min-plus: shortest paths,
- ...and more – see e.g. Aho, Hopcroft, and Ullman '74; Kepner & Gilbert '11.

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one.

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one.

Use different backends **without ever changing the ALP programs**(!)

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one.

Use different backends **without ever changing the ALP programs**(!)

Selecting the **sequential** auto-vectorising backend:

```
grbcxx -o myProgram myProgram.cpp
grbrun ./myProgram datasets/west0497.mtx
```

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one.

Use different backends **without ever changing the ALP programs**(!)

Selecting the **shared-memory parallel** **auto-vectorising** backend:

```
grbcxx --backend reference_omp -o myProgram myProgram.cpp
grbrun -b reference_omp ./myProgram datasets/west0497.mtx
```

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one.

Use different backends **without ever changing the ALP programs**(!)

Selecting the **1D distributed-memory parallel** backend (4 nodes):

```
grbcxx -b bsp1d -o myProgram myProgram.cpp
grbrun -b bsp1d -np 4 ./myProgram datasets/west0497.mtx
```

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one.

Use different backends **without ever changing the ALP programs**(!)

Selecting the **hybrid shared and dist. parallel** backend (10 nodes):

```
grbcxx -b hybrid -o myProgram myProgram.cpp
grbrun -b hybrid -np 10 ./myProgram datasets/west0497.mtx
```

GraphBLAS

A *non-exclusive* list of graph algorithms expressed using GraphBLAS:

- strongly connected components,
- p-Laplacian spectral clustering,
- maximal independent set,
- minimum spanning tree,
- betweenness centrality,
- k-core decomposition,
- graph contraction,
- depth-first search,
- triangle counting,
- graph generation,
- shortest paths,
- ...and more– see `graphblas.org` for an up-to-date overview.

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

ALP v0.8 comes with the following prototypes:

- Sparse BLAS (Duff et al. '02, NIST BLAS tech. forum standard)
- SpBLAS (preceding non-compliant POC, de-facto standard)

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

ALP v0.8 comes with the following prototypes:

- Sparse BLAS (Duff et al. '02, NIST BLAS tech. forum standard)
- SpBLAS (preceding non-compliant POC, de-facto standard)
  - disadvantage: loses some ALP automatisms and performance

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

ALP v0.8 comes with the following prototypes:

- Sparse BLAS (Duff et al. '02, NIST BLAS tech. forum standard)
- SpBLAS (preceding non-compliant POC, de-facto standard)
  - disadvantage: loses some ALP automatisms and performance
- CRS-compatible sparse iterative solvers:

```
double *x, *b, *a_vals; int *a_cols, *a_offs;
// ...

sparse_cg_handle_t handle;
sparse_cg_init( &handle, n, a_vals, a_cols, a_offs );

sparse_cg_solve( handle, x, b );
sparse_cg_destroy( handle );
```

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

ALP v0.8 comes with the following prototypes:

- Sparse BLAS (Duff et al. '02, NIST BLAS tech. forum standard)
- SpBLAS (preceding non-compliant POC, de-facto standard)
  - disadvantage: loses some ALP automatisms and performance
- CRS-compatible solvers with user-defined preconditioning:

```
double *x, *b, *a_vals; int *a_cols, *a_offs;
int my_preconditioner( double * out, const double * in, void * data );
// ...
sparse_cg_handle_t handle;
sparse_cg_init( &handle, n, a_vals, a_cols, a_offs );
sparse_cg_set_preconditioner( handle, my_preconditioner, data );
sparse_cg_solve( handle, x, b );
sparse_cg_destroy( handle );
```

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1
ALP	0.691	0.483	1.95	5.29	7.86	36.0

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1
ALP	0.691	0.483	1.95	5.29	7.86	36.0

ALP up to $28.5\times$ faster vs. Eigen and 6.96, $2.01\times$ vs. hand-optimised.

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;
- data exchange between Spark and ALP happens within-process.

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;
- data exchange between Spark and ALP happens within-process.

Orders of magnitude improvements on **10 nodes** (hybrid backend):

				Spark				Spark with ALP/GraphBLAS			
	GB	Gnz	n_ϵ	$n=1$	$n=10$	$n=n_\epsilon$	s/it.	$n=1$	$n=10$	$n=n_\epsilon$	s/it.
uk-2002	4.7	0.3	73	168.6	1373.8	>4 hrs	133.9	8.7	13.9	48.7	0.56
clueweb12	786	42.5	45	-	-	-	-	658.8	963.2	1875.0	27.7

Pagerank performance in seconds using ten Ivy nodes with Infiniband EDR, Spark 2.3.1, and Hadoop 2.7.7.

- I/O: 19× faster, computation: 239× faster for `uk-2002`;

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;
- data exchange between Spark and ALP happens within-process.

Orders of magnitude improvements on **10 nodes** (hybrid backend):

				Spark				Spark with ALP/GraphBLAS			
	GB	Gnz	n_ϵ	$n=1$	$n=10$	$n=n_\epsilon$	s/it.	$n=1$	$n=10$	$n=n_\epsilon$	s/it.
uk-2002	4.7	0.3	73	168.6	1373.8	>4 hrs	133.9	8.7	13.9	48.7	0.56
clueweb12	786	42.5	45	-	-	-	-	658.8	963.2	1875.0	27.7

Pagerank performance in seconds using ten Ivy nodes with Infiniband EDR, Spark 2.3.1, and Hadoop 2.7.7.

- I/O: 19× faster, computation: 239× faster for `uk-2002`;
  - can handle 141× larger problems, 12× fewer resources

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;
- data exchange between Spark and ALP happens within-process.

Orders of magnitude improvements on **10 nodes** (hybrid backend):

				Spark				Spark with ALP/GraphBLAS			
	GB	Gnz	n_ϵ	$n=1$	$n=10$	$n=n_\epsilon$	s/it.	$n=1$	$n=10$	$n=n_\epsilon$	s/it.
uk-2002	4.7	0.3	73	168.6	1373.8	>4 hrs	133.9	8.7	13.9	48.7	0.56
clueweb12	786	42.5	45	-	-	-	-	658.8	963.2	1875.0	27.7

Pagerank performance in seconds using ten Ivy nodes with Infiniband EDR, Spark 2.3.1, and Hadoop 2.7.7.

- I/O: 19× faster, computation: 239× faster for `uk-2002`;
  - can handle 141× larger problems, 12× fewer resources
- v0.8 (prelim.), **3 nodes**, uk-2002, IB EDR, dual-socket ARM:
  - 113 ms./iter, 19.2× vs. GraphX. See ALP/Spark @ GitHub

Ref.: Suijlen and Y., "Lightweight Parallel Foundations", (2019); ALP v0.8-preview and ALP/Spark (2024).

Beyond ALP/GraphBLAS

How far can we take this type of programming?

The Alps

The Alps:

- Monte Rosa,
- Matterhorn,
- Weisshorn,
- Jungfrau,
- Rothorn,
- Dom,
- ...

The ALPs

Algebraic Programming

IRs, communication layers, domain-specific languages, libraries and everything in-between for realising Algebraic Programming

Switzerland https://algebraic-programming.gith...

The ALPs:

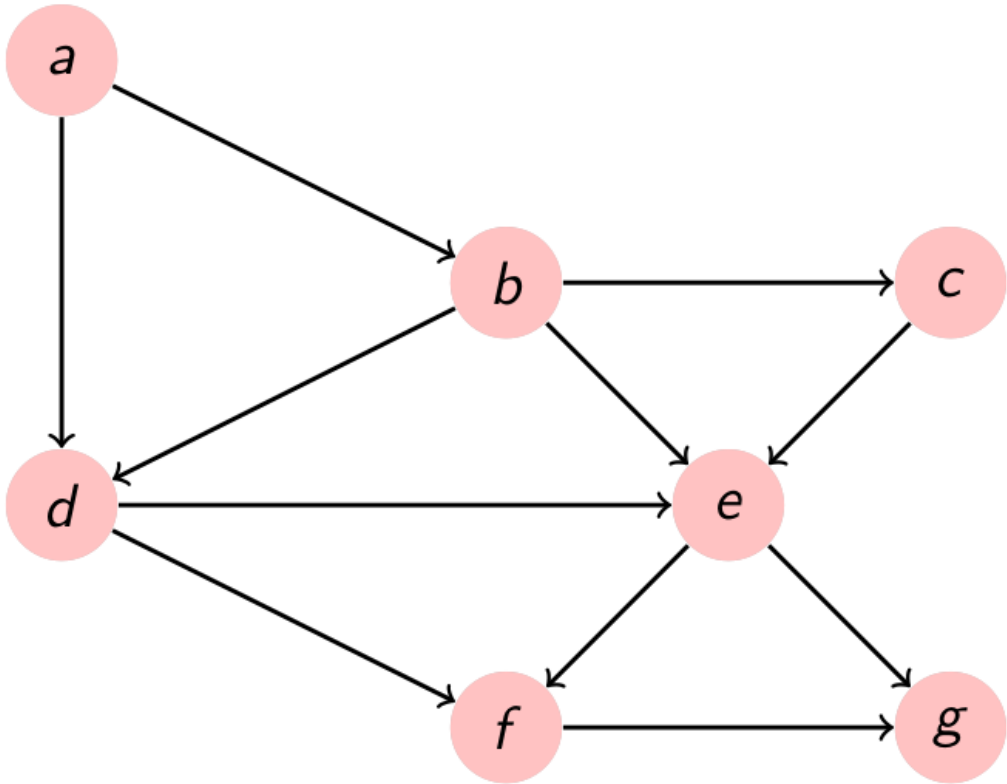
- linear algebra: **ALP/GraphBLAS** and **ALP/Dense**,
- vertex-centric programming: **ALP/Pregel**,
- towards tensor algebra: **ALP/Tensors** and **ALP/Ascend**,
- ...

Interoperability with existing software:

- **ALP/Spark**;
- **ALP/Solver**, **ALP/SparseBLAS**, **ALP/SpBLAS**.

ALP/Pregel

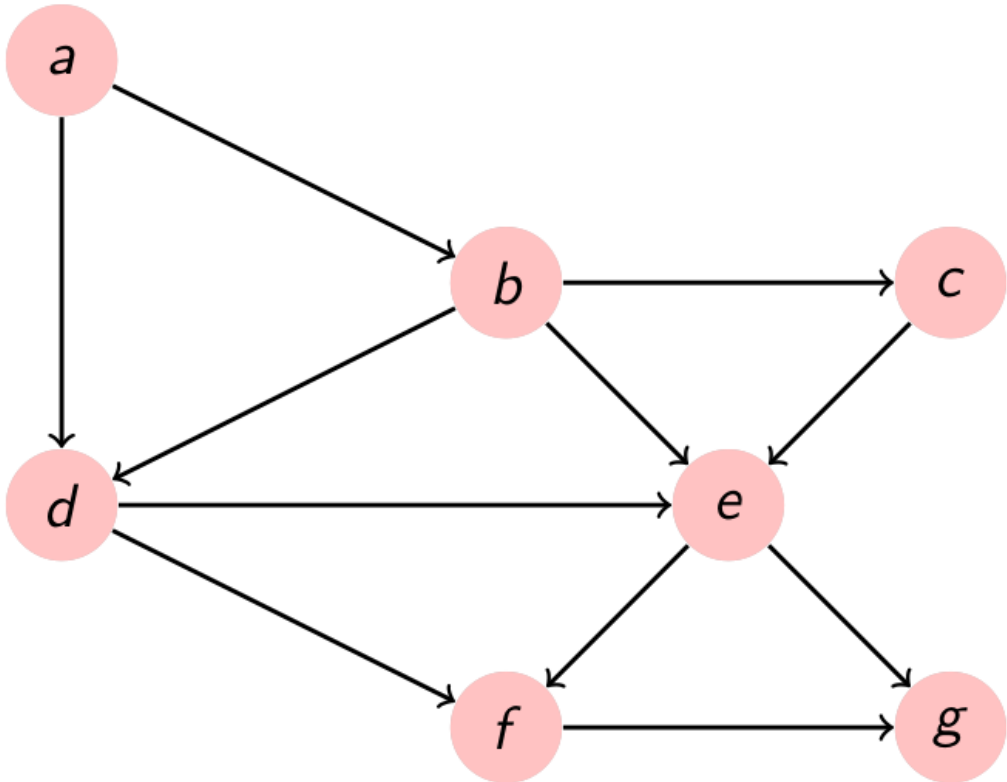
Pregel:



- Each vertex executes a round-based program;
- after each round, message exchange over edges.

ALP/Pregel

Pregel:



- Each vertex executes a round-based program;
- after each round, message exchange over edges.

Think like a vertex, Malewicz et al. '10.

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours;
  - if not, vote to halt program.

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours;
    - if not, vote to halt program.

Pregel **page ranking** over directed graphs:

- start with equally-distributed local score;
- divide it over the number of neighbours and broadcast;
- new score is $\alpha + (1 - \alpha)$ times the sum over incoming scores.

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours;
  - if not, vote to halt program.

Pregel **page ranking** over directed graphs:

- start with equally-distributed local score;
- divide it over the number of neighbours and broadcast;
- new score is $\alpha + (1 - \alpha)$ times the sum over incoming scores.
  - execute a fixed number of rounds, or
  - use local convergence detection and vote-to-halt.

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours;
  - if not, vote to halt program.

Pregel **page ranking** over directed graphs:

- start with equally-distributed local score;
- divide it over the number of neighbours and broadcast;
- new score is $\alpha + (1 - \alpha)$ times the sum over incoming scores.
  - execute a fixed number of rounds, or
  - use local convergence detection and vote-to-halt.
- while "PageRank-like", **not mathematically equivalent**!

ALP/Pregel

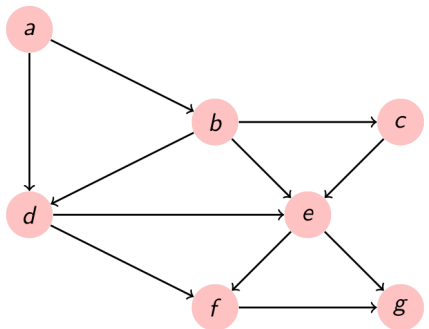
Expanding Pregel programs into ALP/GraphBLAS:

```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message, // IDs will propagate from neighbours
        VertexIDType &outgoing_message, // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) { // a larger ID has arrived; join the
      current_max_ID = incoming_message;        // component 'led' by this ID

    } else {                                    // otherwise no change: if everyone
      pregel.voteToHalt = true;                 // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;            // as long as we're running, keep
}                                               // broadcasting my component ID
```



ALP/Pregel

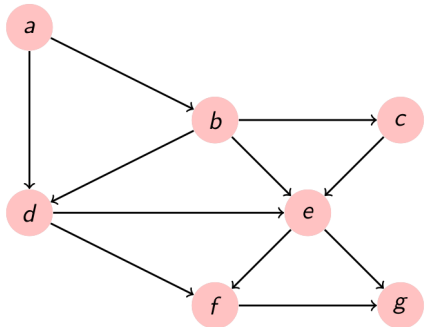
Expanding Pregel programs into ALP/GraphBLAS:

```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message, // IDs will propagate from neighbours
        VertexIDType &outgoing_message, // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) { // a larger ID has arrived; join the
      current_max_ID = incoming_message;        // component 'led' by this ID

    } else {                                    // otherwise no change: if everyone
      pregel.voteToHalt = true;                 // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;            // as long as we're running, keep
}                                               // broadcasting my component ID
```



- grb::eWiseLambda executes a vertex program;

ALP/Pregel

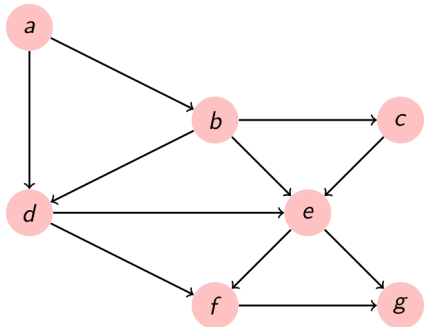
Expanding Pregel programs into ALP/GraphBLAS:

```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message, // IDs will propagate from neighbours
        VertexIDType &outgoing_message, // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) { // a larger ID has arrived; join the
      current_max_ID = incoming_message;        // component 'led' by this ID

    } else {                                    // otherwise no change: if everyone
      pregel.voteToHalt = true;                 // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;            // as long as we're running, keep
}                                               // broadcasting my component ID
```



- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;

ALP/Pregel

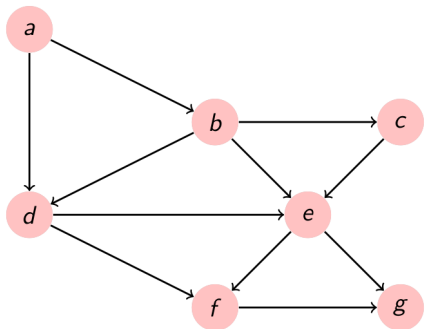
Expanding Pregel programs into ALP/GraphBLAS:

```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message, // IDs will propagate from neighbours
        VertexIDType &outgoing_message, // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) { // a larger ID has arrived; join the
      current_max_ID = incoming_message;        // component 'led' by this ID

    } else {                                    // otherwise no change: if everyone
      pregel.voteToHalt = true;                 // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;            // as long as we're running, keep
}                                               // broadcasting my component ID
```



- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;
- grb::Monoid performs reductions of incoming messages;

ALP/Pregel

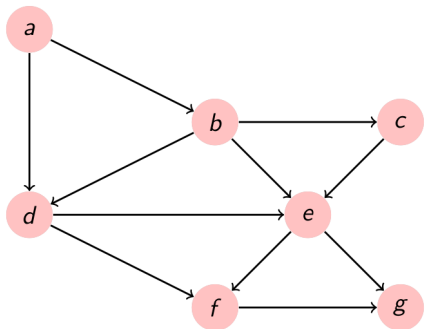
Expanding Pregel programs into ALP/GraphBLAS:

```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message, // IDs will propagate from neighbours
        VertexIDType &outgoing_message, // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) { // a larger ID has arrived; join the
      current_max_ID = incoming_message;        // component 'led' by this ID

    } else {                                    // otherwise no change: if everyone
      pregel.voteToHalt = true;                 // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;            // as long as we're running, keep
}                                               // broadcasting my component ID
```



- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;
- grb::Monoid performs reductions of incoming messages;
- grb::fold reduces halting votes to termination condition.

ALP/Pregel

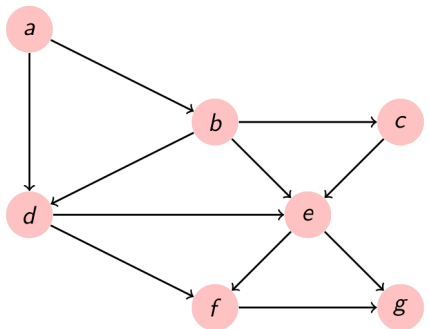
Expanding Pregel programs into ALP/GraphBLAS:

```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message, // IDs will propagate from neighbours
        VertexIDType &outgoing_message, // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) { // a larger ID has arrived; join the
      current_max_ID = incoming_message;        // component 'led' by this ID

    } else {                                    // otherwise no change: if everyone
      pregel.voteToHalt = true;                 // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;            // as long as we're running, keep
}                                               // broadcasting my component ID
```



- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;
- grb::Monoid performs reductions of incoming messages;
- grb::fold reduces halting votes to termination condition.

The translation is **automatic**, using the standard ALP software stack

ALP/Pregel

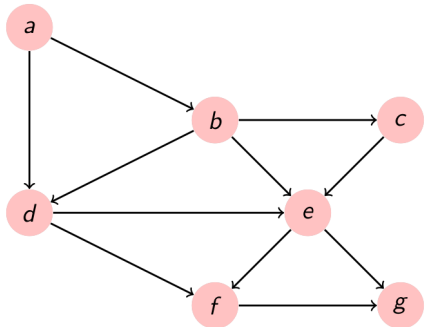
Expanding Pregel programs into ALP/GraphBLAS:

```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message, // IDs will propagate from neighbours
        VertexIDType &outgoing_message, // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) { // a larger ID has arrived; join the
      current_max_ID = incoming_message;        // component 'led' by this ID

    } else {                                    // otherwise no change: if everyone
      pregel.voteToHalt = true;                 // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;            // as long as we're running, keep
}                                               // broadcasting my component ID
```



- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;
- grb::Monoid performs reductions of incoming messages;
- grb::fold reduces halting votes to termination condition.

The translation is **automatic**, using the standard ALP software stack:

```
grbcxx -b hybrid myPregelAlgo pregelAlgo.cpp
grbrun -b hybrid -np 4 ./myPregelAlgo
```

ALP/Pregel

For the Pregel "page-ranking", **two variants**:

- global: terminate when all vertices are converged;
- local: disable locally converged vertices from any future rounds.

ALP/Pregel

For the Pregel "page-ranking", **two variants**:

- global: terminate when all vertices are converged;
- local: disable locally converged vertices from any future rounds.

Using masking to not incur overhead from inactive vertices;

- the more deactivated vertices, the faster each compute round.

ALP/Pregel

For the Pregel "page-ranking", **two variants**:

- global: terminate when all vertices are converged;
- local: disable locally converged vertices from any future rounds.

Using masking to not incur overhead from inactive vertices;

- the more deactivated vertices, the faster each compute round.

	ALP/Pregel		Sequential
Dataset	Global	Local	GraphBLAS
gyro_m	34.8 (40)	24.7 (39)	31.4 (52)
G2_circuit	175 (38)	78.8 (36)	90.0 (48)
bundle_adj	3 070 (66)	2 070 (51)	2 330 (60)
G3_circuit	1 960 (38)	987 (36)	*1 100* (48)
wiki-2007	40 500 (103)	11 400 (96)	18 100 (55)
uk-2002	153 000 (115)	*46 100* (104)	*72 100* (73)
road_usa	*87 600* (78)	58 800 (72)	62 200 (78)

Sequential performance in ms. Compares different page ranking algorithms.

ALP/Pregel

When using the (blocking) shared-memory parallel backend:

	ALP/Pregel		Blocking
Dataset	Global	Local	GraphBLAS
gyro_m	31.1 (40)	29.2 (39)	*37.6* (52)
G2_circuit	58.8 (38)	38.9 (36)	29.0 (48)
bundle_adj	280 (66)	*224* (51)	1 290 (60)
G3_circuit	367 (38)	243 (36)	87.8 (48)
wiki-2007	2 440 (103)	878 (96)	5 030 (55)
uk-2002	11 500 (115)	4 420 (104)	2 750 (73)
road_usa	9 800 (78)	7 560 (72)	2 680 (78)

- 0.84–13.0× speedup for the local Pregel page ranking;
- fastest 3 out of 7 times (5 out of 13 in the full paper)

Ref.: Y., "Humble Heroes", Communications of Huawei Research, to appear (2024).

ALP/Pregel

When using the (blocking) shared-memory parallel backend:

Dataset	ALP/Pregel Global	ALP/Pregel Local	Blocking GraphBLAS
gyro_m	31.1 (40)	29.2 (39)	*37.6* (52)
G2_circuit	58.8 (38)	38.9 (36)	29.0 (48)
bundle_adj	280 (66)	*224* (51)	1 290 (60)
G3_circuit	367 (38)	243 (36)	87.8 (48)
wiki-2007	2 440 (103)	878 (96)	5 030 (55)
uk-2002	11 500 (115)	4 420 (104)	2 750 (73)
road_usa	9 800 (78)	7 560 (72)	2 680 (78)

- 0.84–13.0× speedup for the local Pregel page ranking;
- fastest 3 out of 7 times (5 out of 13 in the full paper);
- 1.03–17.5× speedup for connected components algorithm.

Ref.: Y., "Humble Heroes", Communications of Huawei Research, to appear (2024).

ALP as a foundational programming model

The ALPs:

- ALP/GraphBLAS, ALP/Pregel, ALP/Dense, ALP/Ascend, ...

ALP as a foundational programming model

The ALPs:

- ALP/GraphBLAS, ALP/Pregel, ALP/Dense, ALP/Ascend, ...

Integrates with existing software:

- interoperability: ALP/Spark;
- transition paths: ALP/SparseBLAS, ALP/SpBLAS, ALP/Solver.

ALP as a foundational programming model

The ALPs:

- ALP/GraphBLAS, ALP/Pregel, ALP/Dense, ALP/Ascend, ...

Integrates with existing software:

- interoperability: ALP/Spark;
- transition paths: ALP/SparseBLAS, ALP/SpBLAS, ALP/Solver.

Future:

- what other humble APIs can ALP efficiently simulate?
- what other algebras to support for widened applicability?

ALP as a foundational programming model

The ALPs:

- ALP/GraphBLAS, ALP/Pregel, ALP/Dense, ALP/Ascend, ...

Integrates with existing software:

- interoperability: ALP/Spark;
- transition paths: ALP/SparseBLAS, ALP/SpBLAS, ALP/Solver.

Future:

- what other humble APIs can ALP efficiently simulate?
- what other algebras to support for widened applicability?

One software stack, many humble interfaces, hero performance

Ref.: Y., "Humble Heroes", Communications of Huawei Research.

It's open!

Open source, Apache 2.0, welcome to try, use, and collaborate!

- `https://github.com/Algebraic-Programming`
- `https://algebraic-programming.github.io`

Publications:

- Suijlen, Y.: Lightweight Parallel Foundations: a model-compliant communication layer (2019);
- Y., Di Nardo, Nash, Suijlen: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation (2020);
- Mastoras, Anagnostidis, Y.: Nonblocking execution in GraphBLAS, IPDPSW (2022);
- Chelini, Barthels, Bientinesi, Copic, Grosser, Spampinato: MOM: Matrix Operations in MLIR, HiPEAC IMPACT workshop (2022);
- Mastoras, Anagnostidis, Y.: Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance, ACM TACO (2023);
- Scolari, Y.: Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS, IPDPSW (GrAPL 2023);
- Spampinato, Jelovina, Zhuang, Y.: Towards Structured Algebraic Programming, ACM ARRAY (2023);
- Papp, Anegg, Y.: Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications, ACM SPAA (2023);
- Papp, Anegg, Y.: DAG scheduling in the BSP model (submitted, 2023);
- Pasadakis, et al., Nonlinear spectral clustering with C++ GraphBLAS, extended abstract, IEEE HPEC (2023, **outstanding short paper**);
- Papp, Anegg, Karanasiou, Y.: Efficient Multi-Processor Scheduling in Increasingly Realistic Models (accepted, SPAA 2024).
- Y.: Humble Heroes, Communications of Huawei Research (2024, to appear);

Backup slides

Backup slides

Historical context

- The Design and Analysis of Computer Algorithms,
  Aho, Hopcroft, Ullman (1974)
- Introduction to Algorithms (first edition only),
  Cormen, Leiserson, Rivest (1990)
- **Elements of Programming**,
  Alexander Stepanov & Paul McJones (2009)
- **Graph Algorithms in the Language of Linear Algebra**,
  Jeremy Kepner & John Gilbert (2011)
- **From Mathematics to Generic Programming**,
  Alexander Stepanov & Daniel Rose (2015)
- **GraphBLAS.org**, following work by Kepner & Gilbert,
  Kepner, Gilbert, Buluç, Mattson, et alii (2016)

Historical context

- The Design and Analysis of Computer Algorithms, Aho, Hopcroft, Ullman (1974)
- Introduction to Algorithms (first edition only), Cormen, Leiserson, Rivest (1990)
- **Elements of Programming**, Alexander Stepanov & Paul McJones (2009)
- **Graph Algorithms in the Language of Linear Algebra**, Jeremy Kepner & John Gilbert (2011)
- **From Mathematics to Generic Programming**, Alexander Stepanov & Daniel Rose (2015)
- **GraphBLAS.org**, following work by Kepner & Gilbert, Kepner, Gilbert, Buluç, Mattson, et alii (2016)
- A C++ GraphBLAS, Y., Suijlen, Di Nardo, Nash (2017)
- **Algebraic Programming** (2021 onwards)
- ...

Sparse tensor storage

Wish-list:

- arbitrary traversal, e.g., using Hilbert- or Z-curves;
- index compression using smaller word lengths;
- low memory footprints; and
- mode-obliviousness.

Sparse tensor storage

Wish-list:

- arbitrary traversal, e.g., using Hilbert- or Z-curves;
- index compression using smaller word lengths;
- low memory footprints; and
- mode-obliviousness.

Evaluate for bandwidth-bound kernels first;

- compute-bound kernels can follow sparse data structure.

Sparse tensor storage

Wish-list:

- arbitrary traversal, e.g., using Hilbert- or Z-curves;
- index compression using smaller word lengths;
- low memory footprints; and
- mode-obliviousness.

Evaluate for bandwidth-bound kernels first;

- compute-bound kernels can follow sparse data structure.

State-of-the-art:

- CSF (Smith et al.): generalisation of CRS;
- HiCOO (Li et al.): generalisation of CSB

Sparse tensor storage

Wish-list:

- arbitrary traversal, e.g., using Hilbert- or Z-curves;
- index compression using smaller word lengths;
- low memory footprints; and
- mode-obliviousness.

Evaluate for bandwidth-bound kernels first;

- compute-bound kernels can follow sparse data structure.

State-of-the-art:

- CSF (Smith et al.): generalisation of CRS;
- HiCOO (Li et al.): generalisation of CSB;
- dense bandwidth-bound: Pawłowski, Uçar, Y. ('19 JOCS, '20 HPEC); Martinez-Ferrer, Y., Beltran ('22 TPDS, '24 submitted).

Incremental sparse tensor storage schemes

Following work on **incremental** matrix storages, ICRS, (C)BICRS:

- mode-skipping incremental sparse fibers (MoSk-IF);
- index-identified incremental sparse fibers (Ind-IF);
- bit-encoded incremental sparse fibers (Bit-IF).

Master's thesis by Xiaohe Niu (with Y. and O. Schenk):

- **Ref.** *Rethinking Sparse Tensor Storage: Incremental Formats as a Path Towards Maximizing Tensor-Vector Multiplication Efficiency*, Master Thesis, ETH Zürich, May 2023;
- `https://github.com/xniuuu/SparseTensorComputations`, GPL v3.

Incremental sparse tensor storage schemes

Following work on **incremental** matrix storages, ICRS, (C)BICRS:

- mode-skipping incremental sparse fibers (MoSk-IF);
  - alike CBICRS
- index-identified incremental sparse fibers (Ind-IF);
  - alike CSF
- bit-encoded incremental sparse fibers (Bit-IF).

Master's thesis by Xiaohe Niu (with Y. and O. Schenk):

- **Ref.** *Rethinking Sparse Tensor Storage: Incremental Formats as a Path Towards Maximizing Tensor-Vector Multiplication Efficiency*, Master Thesis, ETH Zürich, May 2023;
- `https://github.com/xniuuu/SparseTensorComputations`, GPL v3.

Incremental sparse tensor storage schemes

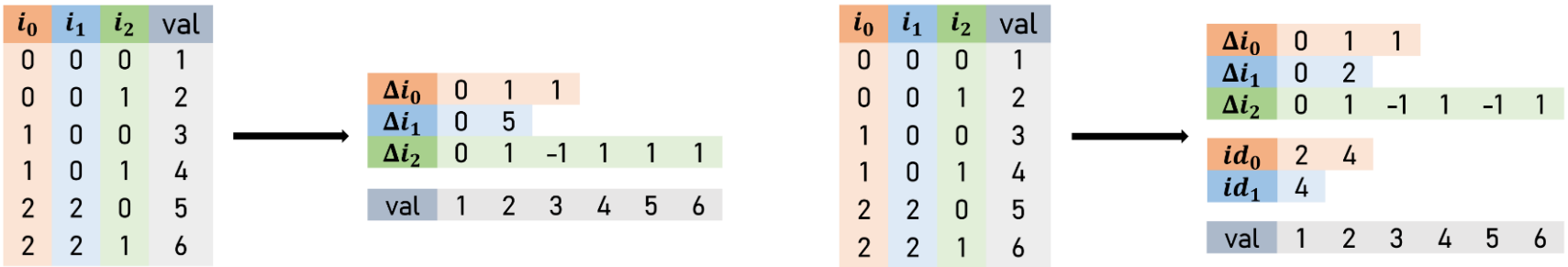


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position

Incremental sparse tensor storage schemes

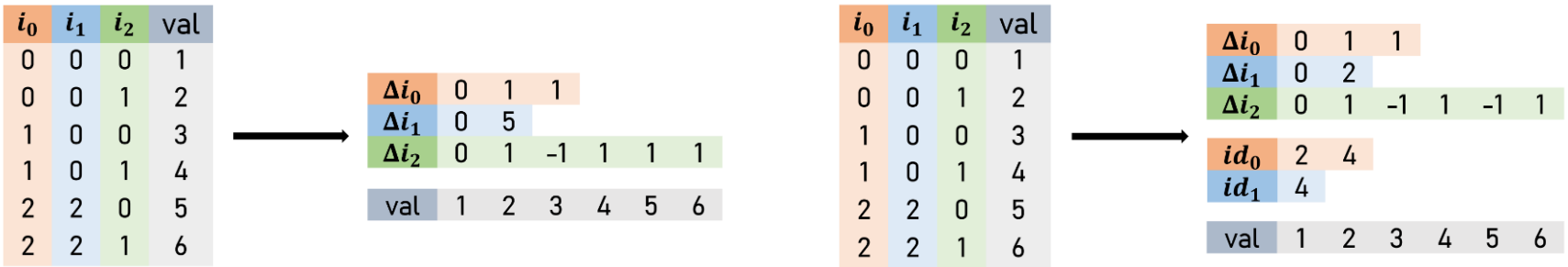


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position;

- start positions stored separately in 32 or 64 bits (depends on n_i);

Incremental sparse tensor storage schemes

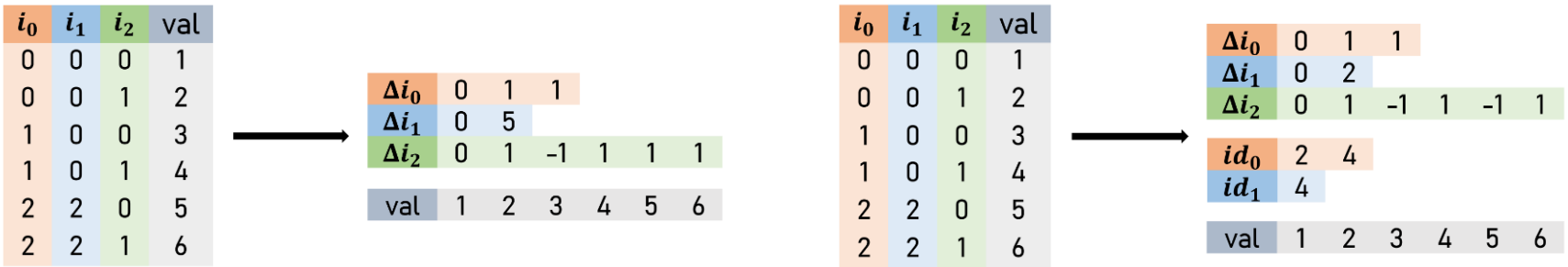


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position;

- start positions stored separately in 32 or 64 bits (depends on n_i);
- increments may use 8 or 16 bits (cf. CBICRS, CSB, HiCOO).

Incremental sparse tensor storage schemes

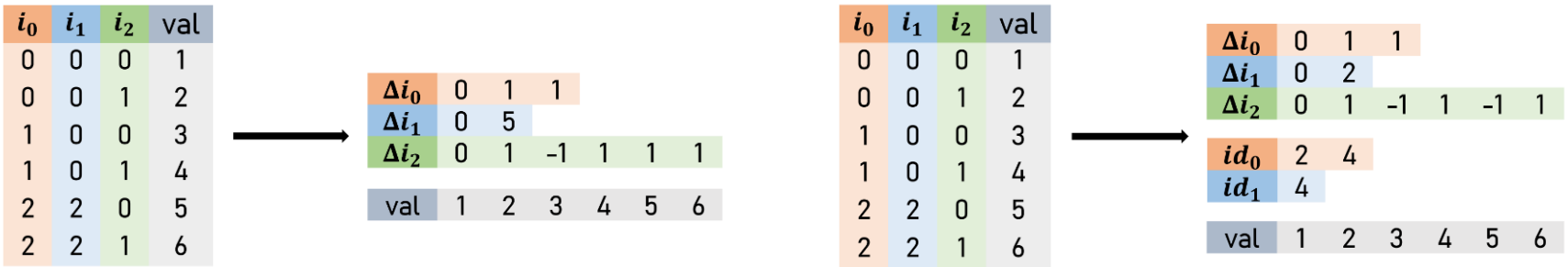


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position;

- start positions stored separately in 32 or 64 bits (depends on n_i);
- increments may use 8 or 16 bits (cf. CBICRS, CSB, HiCOO).

Different variants control which mode(s) change differently.

- MoSk-IF: via over- and under-flow (alike CBICRS);
- Ind-IF: per-mode array indicates when to jump (alike CSF);

Incremental sparse tensor storage schemes

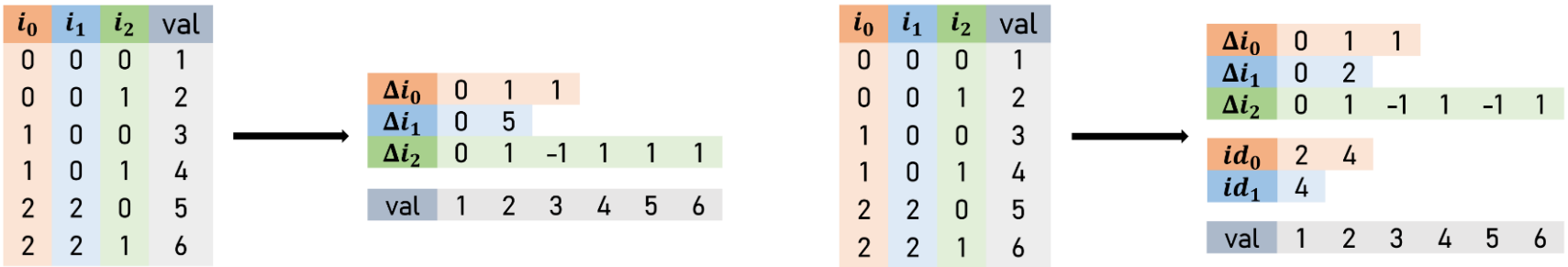


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position;

- start positions stored separately in 32 or 64 bits (depends on n_i);
- increments may use 8 or 16 bits (cf. CBICRS, CSB, HiCOO).

Different variants control which mode(s) change differently.

- MoSk-IF: via over- and under-flow (alike CBICRS);
- Ind-IF: per-mode array indicates when to jump (alike CSF);
- Bit-IF: bit array indicates which mode(s) change.

Incremental sparse tensor storage schemes

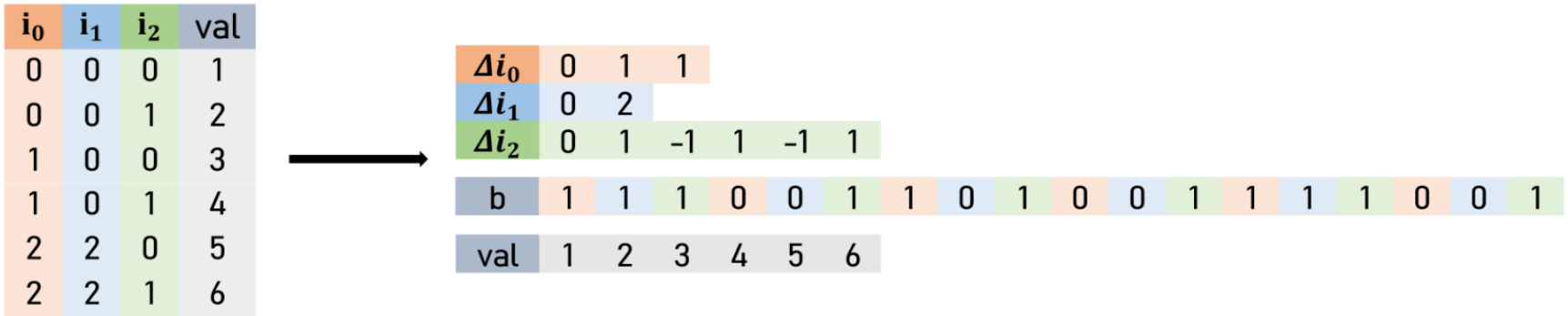


Fig. Bit-IF

Incremental sparse tensor storage schemes

i_0	i_1	i_2	val
0	0	0	1
0	0	1	2
1	0	0	3
1	0	1	4
2	2	0	5
2	2	1	6

Δi_0	0	1	1																
Δi_1	0	2																	
Δi_2	0	1	-1	1	-1	1													
b	1	1	1	0	0	1	1	0	1	0	0	1	1	1	1	0	0	1	
val	1	2	3	4	5	6													

Fig. Bit-IF

Memory footprint analysis – simplified, see thesis for full detail:

- let $q_i = \frac{\textit{jumps along mode } i}{nnz}$ (here, $q_0 = 1/2, q_1 = 1/3, q_2 = 1/2$);

Incremental sparse tensor storage schemes

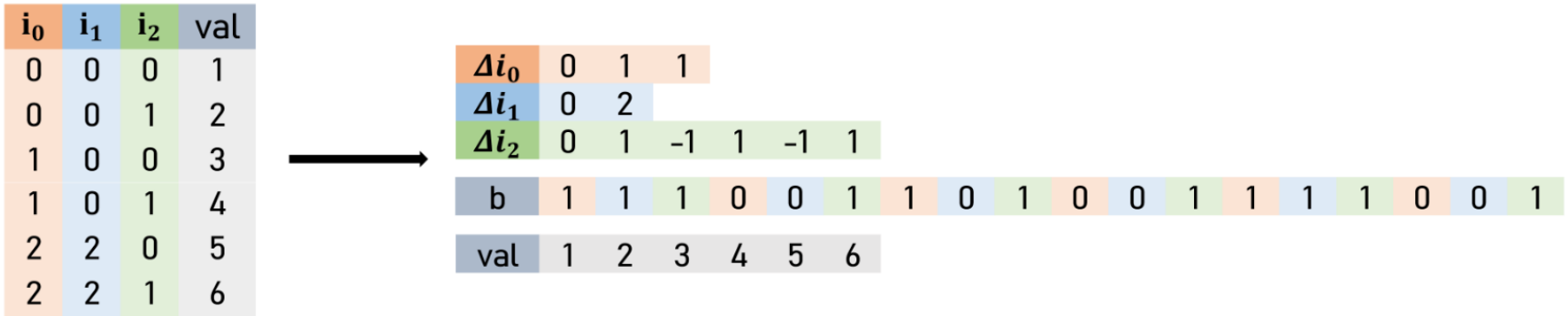


Fig. Bit-IF

Memory footprint analysis – simplified, see thesis for full detail:

- let $q_i = \frac{\textit{jumps along mode } i}{nnz}$ (here, $q_0 = 1/2, q_1 = 1/3, q_2 = 1/2$);
- let $w_{\text{val}} = 8$, $w_{\text{ind}} = 4$, $w_{\text{inc}} = 2$, and $w_{\text{size\_t}} = 8$ bytes.

Then COO requires $nnz(w_{\text{val}} + dw_{\text{ind}})$ bytes

Incremental sparse tensor storage schemes

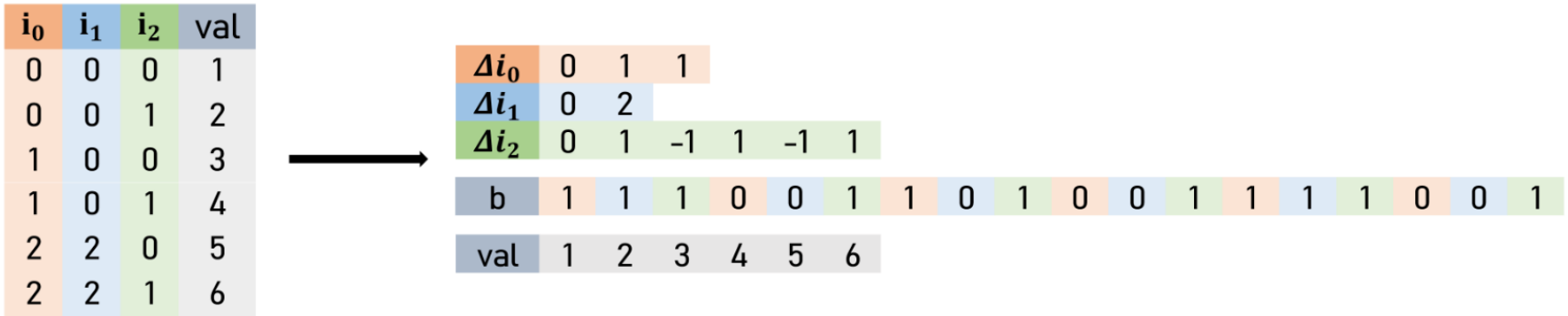


Fig. Bit-IF

Memory footprint analysis – simplified, see thesis for full detail:

- let $q_i = \frac{\textit{jumps along mode } i}{nnz}$ (here, $q_0 = 1/2, q_1 = 1/3, q_2 = 1/2$);
- let $w_{\text{val}} = 8$, $w_{\text{ind}} = 4$, $w_{\text{inc}} = 2$, and $w_{\text{size\_t}} = 8$ bytes.

Then COO requires $nnz(w_{\text{val}} + dw_{\text{ind}})$ bytes, while our X-IF variants:

- MoSk: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-2} \tilde{q}_k \tilde{w}_{\text{inc}}\right] + \tilde{w}_{\text{inc}}\right)$, $\tilde{q}_k \geq q_k$, $\tilde{w}_{\text{inc}} \geq w_{\text{inc}}$;
- Ind: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-2} q_k w_{\text{inc}} + q'_k w_{\text{size\_t}}\right] + w_{\text{inc}}\right)$, $q'_k \leq q_k$;
- Bit: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-1} q_k w_{\text{inc}}\right] + d/8\right)$.

Incremental sparse tensor storage schemes

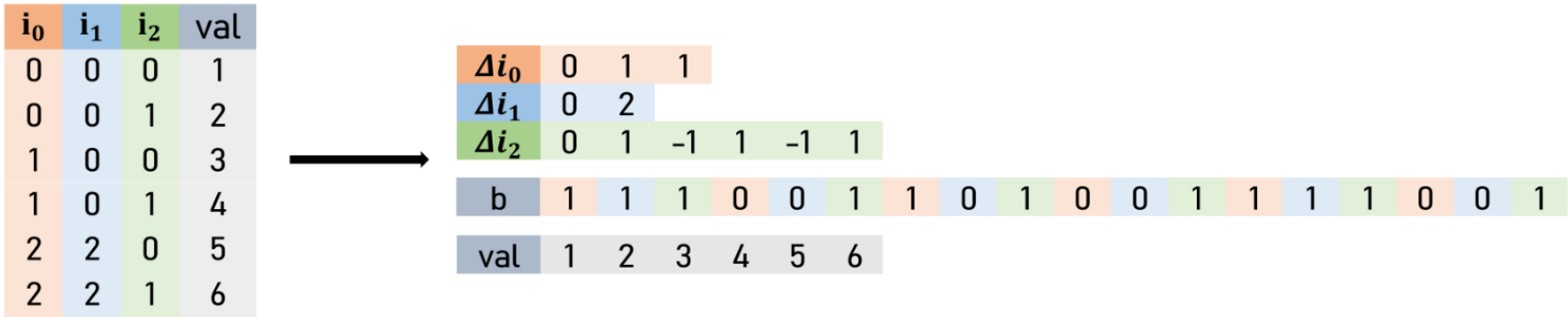


Fig. Bit-IF

Memory footprint analysis – simplified, see thesis for full detail:

- let $q_i = \frac{\textit{jumps along mode } i}{nnz}$ (here, $q_0 = 1/2, q_1 = 1/3, q_2 = 1/2$);
- let $w_{\text{val}} = 8$, $w_{\text{ind}} = 4$, $w_{\text{inc}} = 2$, and $w_{\text{size\_t}} = 8$ bytes.

Then COO requires $nnz(w_{\text{val}} + dw_{\text{ind}})$ bytes, while our X-IF variants:

- MoSk: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-2} \tilde{q}_k \tilde{w}_{\text{inc}}\right] + \tilde{w}_{\text{inc}}\right)$, $\tilde{q}_k \geq q_k$, $\tilde{w}_{\text{inc}} \geq w_{\text{inc}}$;
- Ind: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-2} q_k w_{\text{inc}} + q'_k w_{\text{size\_t}}\right] + w_{\text{inc}}\right)$, $q'_k \leq q_k$;
- Bit: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-1} q_k w_{\text{inc}}\right] + d/8\right)$. Usually best.

Results

10 SPLATT tensors, tensor–vector multiplication over each mode:

- output tensors use the same data structure as the input tensor.

Results

10 SPLATT tensors, tensor–vector multiplication over each mode:

- output tensors use the same data structure as the input tensor.

1) **Bit-IF outperforms COO**
   - $1.37\times$ geomean compression ratio, up to $1.63\times$

Results

10 SPLATT tensors, tensor–vector multiplication over each mode:

- output tensors use the same data structure as the input tensor.

1) **Bit-IF outperforms COO**
   - 1.37× geomean compression ratio, up to 1.63×
   - fastest for 10/10 tensors on AMD EPYC 7720;

Results

10 SPLATT tensors, tensor–vector multiplication over each mode:

- output tensors use the same data structure as the input tensor.

1) **Bit-IF outperforms COO**
   - 1.37× geomean compression ratio, up to 1.63×
   - fastest for 10/10 tensors on AMD EPYC 7720;
   - fastest for 7/10 tensors on Intel 6238T.
     > 3 modes slower when lexicographic match and cacheable.

Results

10 SPLATT tensors, tensor–vector multiplication over each mode:

- output tensors use the same data structure as the input tensor.

1) **Bit-IF outperforms COO**
   - $1.37\times$ geomean compression ratio, up to $1.63\times$
   - fastest for $10/10$ tensors on AMD EPYC 7720;
   - fastest for $7/10$ tensors on Intel 6238T.
     - $>$ 3 modes slower when lexicographic match and cacheable.

2) **blocked Bit-IF**: Hilbert order outperforms Morton
   - helps limit index jump range, lowers w_{inc}(!)

Results

10 SPLATT tensors, tensor–vector multiplication over each mode:

- output tensors use the same data structure as the input tensor.

1) **Bit-IF outperforms COO**
   - $1.37\times$ geomean compression ratio, up to $1.63\times$
   - fastest for $10/10$ tensors on AMD EPYC 7720;
   - fastest for $7/10$ tensors on Intel 6238T.
     - $>$ 3 modes slower when lexicographic match and cacheable.

2) **blocked Bit-IF**: Hilbert order outperforms Morton
   - helps limit index jump range, lowers w_{inc}(!)
   - faster in 20 to 22 TVMs (out of 40);

Results

10 SPLATT tensors, tensor–vector multiplication over each mode:

- output tensors use the same data structure as the input tensor.

1) **Bit-IF outperforms COO**
   - 1.37× geomean compression ratio, up to 1.63×
   - fastest for 10/10 tensors on AMD EPYC 7720;
   - fastest for 7/10 tensors on Intel 6238T.
     > 3 modes slower when lexicographic match and cacheable.

2) **blocked Bit-IF**: Hilbert order outperforms Morton, mixed results
   - helps limit index jump range, lowers w_{inc}(!)
   - faster in 20 to 22 TVMs (out of 40);
   - effective when the block size is chosen appropriately.

Results

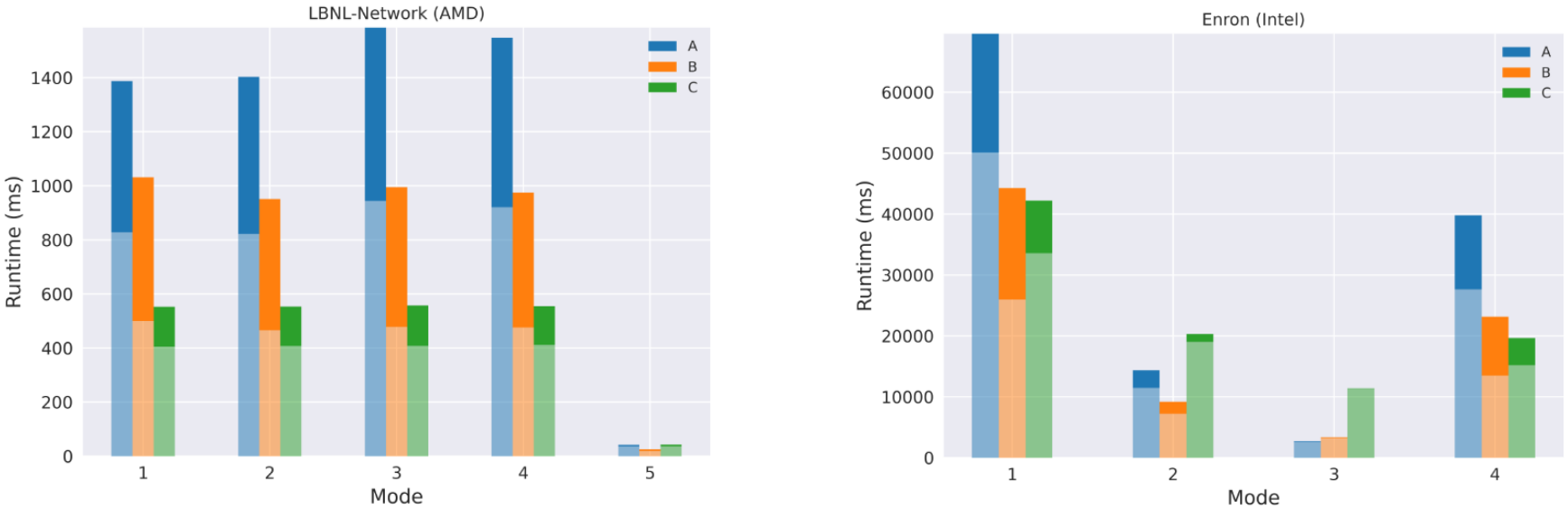


Fig. COO vs. Bit-IF vs. best blocked Bit-IF.
LBNL-Network on AMD EPYC 7720 (left) and Enron on Intel 6238T (right).

Next steps

These are pending:

- additional performance optimisations
  - including for shared-, distributed-memory parallelisation; and
- comparison versus CSF and HiCOO.

Next steps

These are pending:

- additional performance optimisations
  - including for shared-, distributed-memory parallelisation; and
- comparison versus CSF and HiCOO.

Upcoming:

- nonblocking execution & lazy tensors;
- unified sparse & dense interface;
- automatic block size selection;
- ALP framework integration.

TVM results

LBNL-Network, AMD EPYC 7720

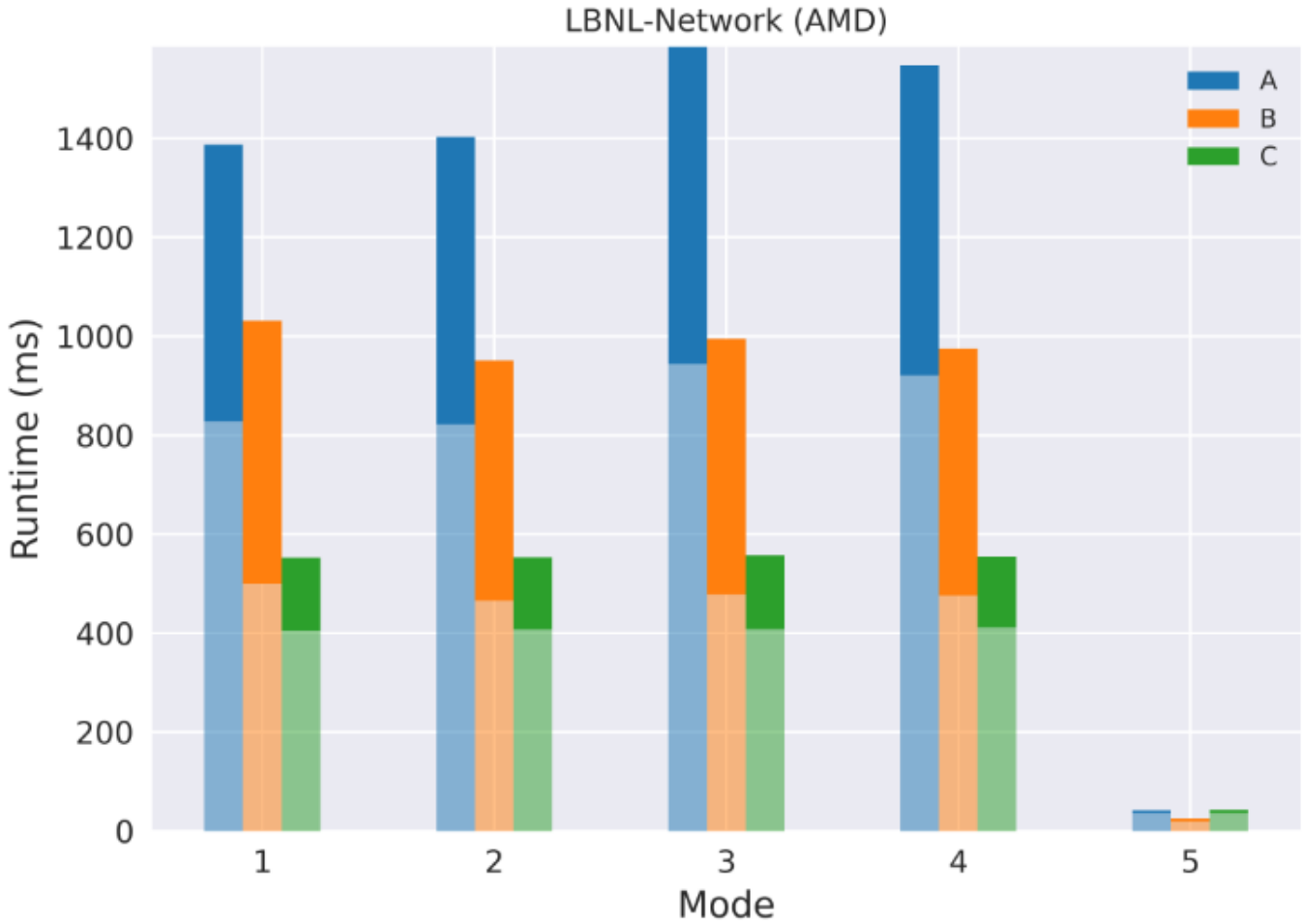


Fig. COO vs. Bit-IF vs. best blocked Bit-IF

TVM results

LBNL-Network, Intel 6238T

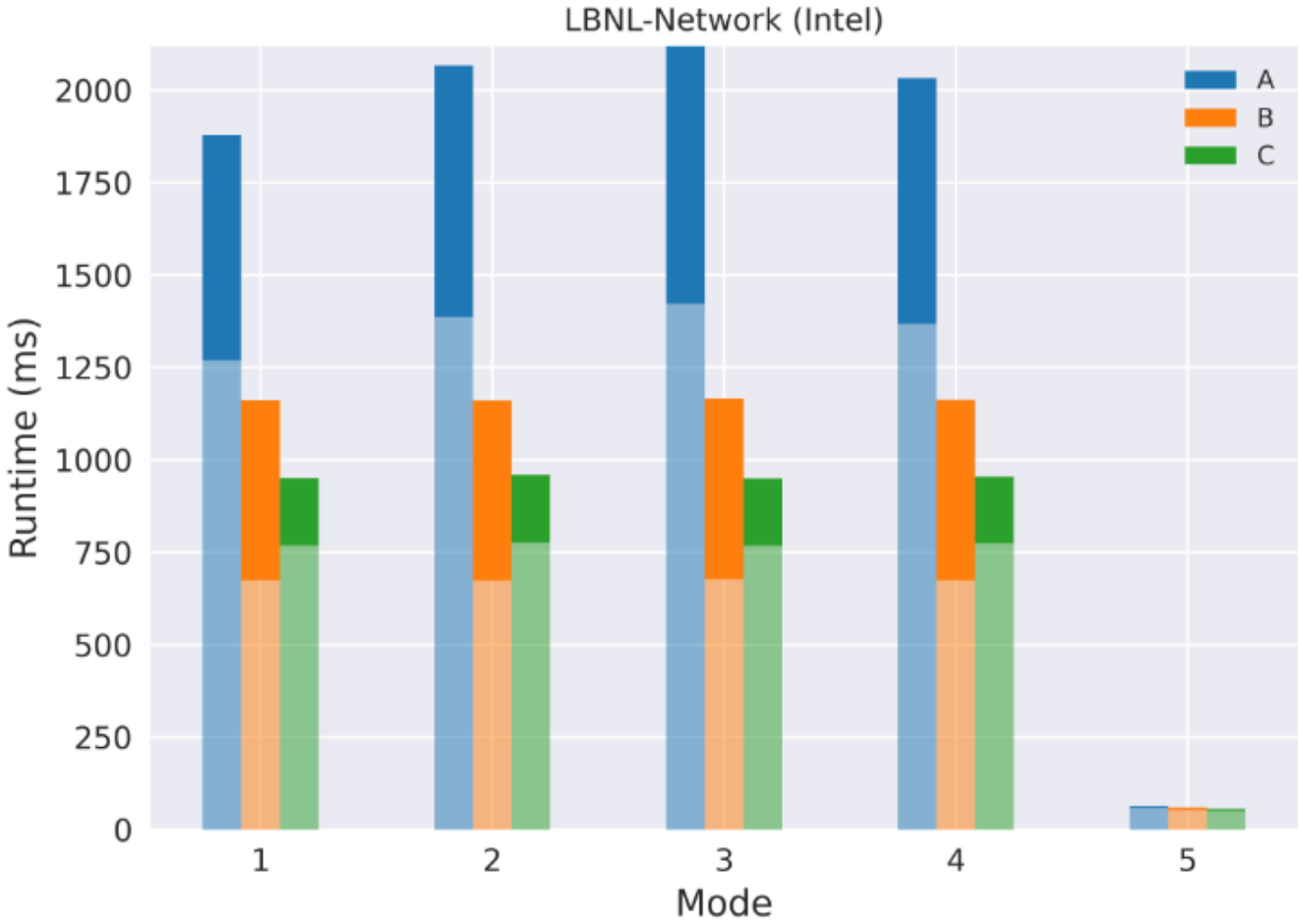


Fig. COO vs. Bit-IF vs. best blocked Bit-IF

TVM results

Enron, AMD EPYC 7720

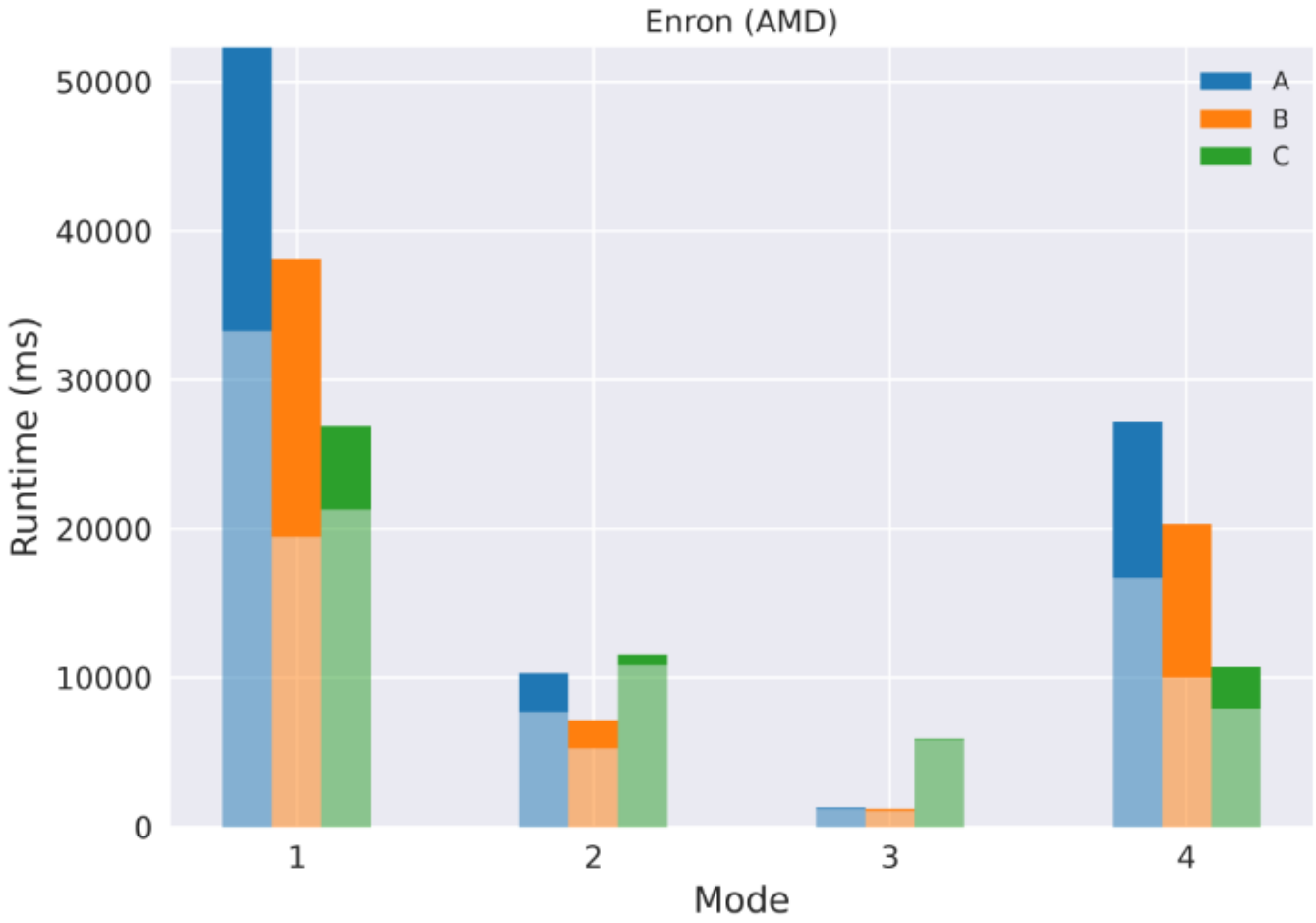


Fig. COO vs. Bit-IF vs. best blocked Bit-IF

TVM results

Enron, Intel 6238T

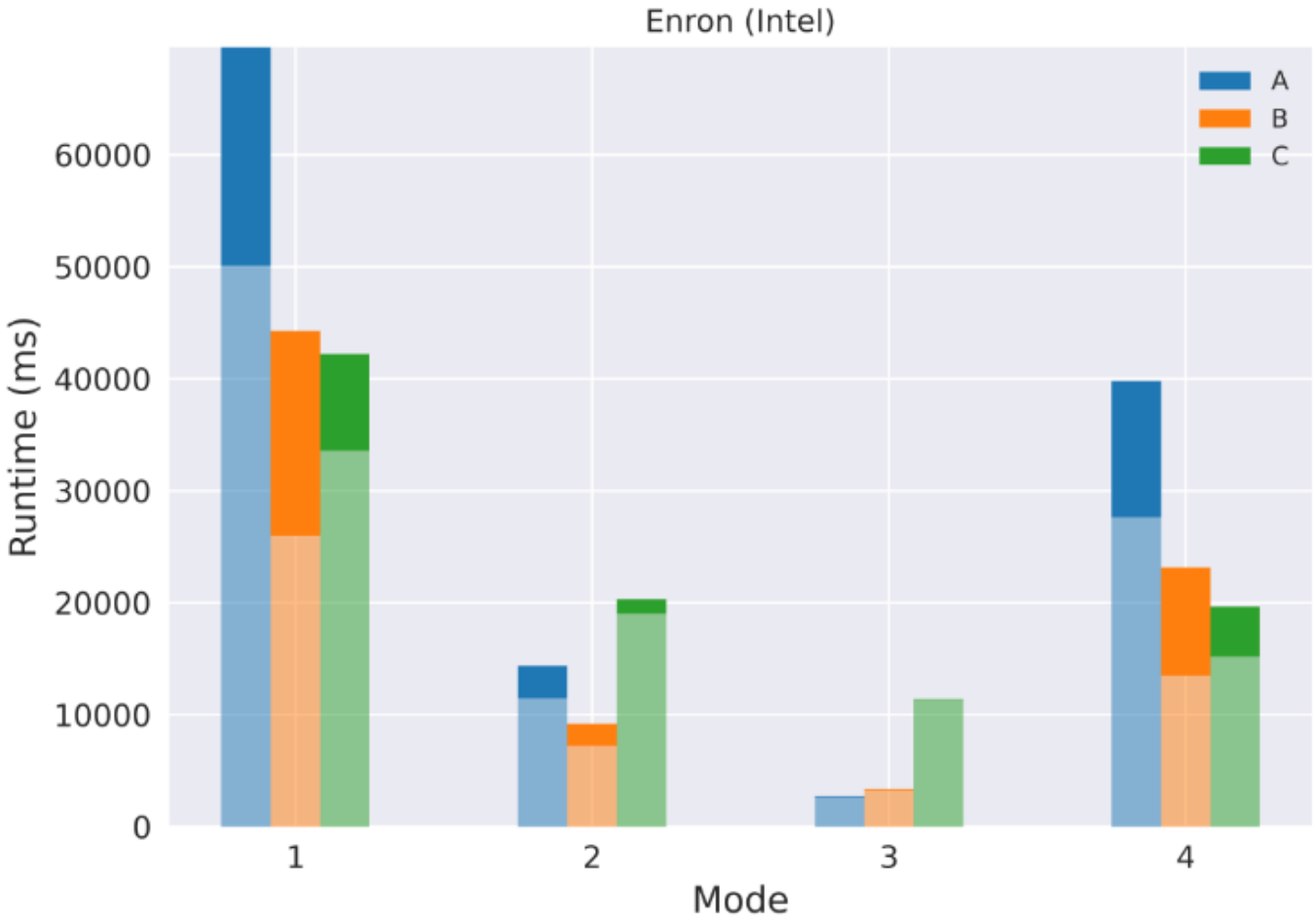


Fig. COO vs. Bit-IF vs. best blocked Bit-IF

ALP/Ascend

Preliminary work, **dense tensors** for Huawei Ascend AI platform:

ALP/Ascend

Preliminary work, **dense tensors** for Huawei Ascend AI platform:

- **ALP/Ascend**, similar to OpenAI Triton (Tillet et al., '19 onwards):

```
void ascend_code( const Grid< 1, 2 > &grid ) {
  Tensor x_global( FP16, make_axes( "i", "j" ) );
  Tensor y_global( FP16, make_axes( "i", "j" ) );
  Tensor z_global( FP16, make_axes( "i", "j" ) );

  rc = grid.forEach( make_axes( "i" ), [&] () {
    auto x_block = getView( x_global );
    auto y_block = getView( y_global );
    auto z_block = getView( z_global );

    z_block( "j" ) = add( x_block( "j" ), y_block( "j" ), "j" ); // z = x + y

    store( z_block );
  } );
}
```

Supports roughly half of current Triton operators.

ALP/Ascend

Preliminary work, **dense tensors** for Huawei Ascend AI platform:

- **ALP/Ascend**, similar to OpenAI Triton (Tillet et al., '19 onwards):

```
void ascend_code( const Grid< 1, 2 > &grid ) {
  Tensor x_global( FP16, make_axes( "i", "j" ) );
  Tensor y_global( FP16, make_axes( "i", "j" ) );
  Tensor z_global( FP16, make_axes( "i", "j" ) );

  rc = grid.forEach( make_axes( "i" ), [&] () {
    auto x_block = getView( x_global );
    auto y_block = getView( y_global );
    auto z_block = getView( z_global );

    z_block( "j" ) = add( x_block( "j" ), y_block( "j" ), "j" ); // z = x + y

    store( z_block );
  } );
}
```

Supports roughly half of current Triton operators. Compiling the above:

a) automatically translates to an ALP tensor dialect
   - element-wise, reduction, and multiplication ops

ALP/Ascend

Preliminary work, **dense tensors** for Huawei Ascend AI platform:

- **ALP/Ascend**, similar to OpenAI Triton (Tillet et al., '19 onwards):

```
void ascend_code( const Grid< 1, 2 > &grid ) {
  Tensor x_global( FP16, make_axes( "i", "j" ) );
  Tensor y_global( FP16, make_axes( "i", "j" ) );
  Tensor z_global( FP16, make_axes( "i", "j" ) );

  rc = grid.forEach( make_axes( "i" ), [&] () {
    auto x_block = getView( x_global );
    auto y_block = getView( y_global );
    auto z_block = getView( z_global );

    z_block( "j" ) = add( x_block( "j" ), y_block( "j" ), "j" ); // z = x + y

    store( z_block );
  } );
}
```

Supports roughly half of current Triton operators. Compiling the above:

a) automatically translates to an ALP tensor dialect
   - element-wise, reduction, and multiplication ops

b) use nonblocking backend to build operation pipelines

ALP/Ascend

Preliminary work, **dense tensors** for Huawei Ascend AI platform:

- **ALP/Ascend**, similar to OpenAI Triton (Tillet et al., '19 onwards):

```
void ascend_code( const Grid< 1, 2 > &grid ) {
  Tensor x_global( FP16, make_axes( "i", "j" ) );
  Tensor y_global( FP16, make_axes( "i", "j" ) );
  Tensor z_global( FP16, make_axes( "i", "j" ) );

  rc = grid.forEach( make_axes( "i" ), [&] () {
    auto x_block = getView( x_global );
    auto y_block = getView( y_global );
    auto z_block = getView( z_global );

    z_block( "j" ) = add( x_block( "j" ), y_block( "j" ), "j" ); // z = x + y

    store( z_block );
  } );
}
```

Supports roughly half of current Triton operators. Compiling the above:

a) automatically translates to an ALP tensor dialect
   - element-wise, reduction, and multiplication ops

b) use nonblocking backend to build operation pipelines

c) once pipeline representation is built, generate parallel Ascend code

ALP/Ascend

Preliminary work, **dense tensors** for Huawei Ascend AI platform:

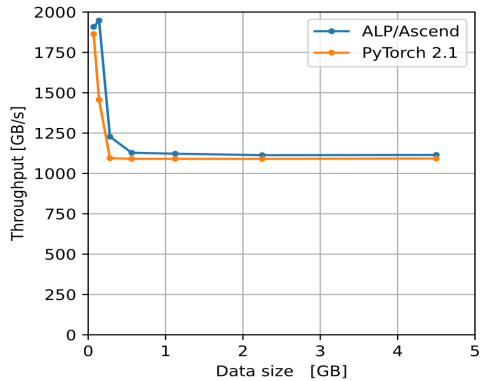
- **ALP/Ascend**, similar to OpenAI Triton (Tillet et al., '19 onwards):

```
void ascend_code( const Grid< 1, 2 > &grid ) {
  Tensor x_global( FP16, make_axes( "i", "j" ) );
  Tensor y_global( FP16, make_axes( "i", "j" ) );
  Tensor z_global( FP16, make_axes( "i", "j" ) );

  rc = grid.forEach( make_axes( "i" ), [&] () {
    auto x_block = getView( x_global );
    auto y_block = getView( y_global );
    auto z_block = getView( z_global );

    z_block( "j" ) = add( x_block( "j" ), y_block( "j" ), "j" ); // z = x + y

    store( z_block );
  } );
}
```



Supports roughly half of current Triton operators. Compiling the above:

a) automatically translates to an ALP tensor dialect
   - element-wise, reduction, and multiplication ops

b) use nonblocking backend to build operation pipelines

c) once pipeline representation is built, generate parallel Ascend code

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );` *// s = r*

2) `grb::eWiseMul( s, alpha, v, semiring );` *// s += alpha .* v*

Blocking execution: the vector s is accessed *twice*

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );` *// s = r*

2) `grb::eWiseMul( s, alpha, v, semiring );` *// s += alpha .* v*

Blocking execution: the vector s is accessed *twice*; performance ✗

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );` // $s = r$
2) `grb::eWiseMul( s, alpha, v, semiring );` // s += *alpha* .* v

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Ref.: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation by Y., D. Di Nardo, J. M. Nash, and W. J. Suijlen (2020).

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );` // $s = r$
2) `grb::eWiseMul( s, alpha, v, semiring );` // s += *alpha* .* v

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Ref.: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation by Y., D. Di Nardo, J. M. Nash, and W. J. Suijlen (2020).

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );` // $s = r$
2) `grb::eWiseMul( s, alpha, v, semiring );` // s += *alpha* .* v

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Automatic non-blocking mode (Mastoras et al., '22):

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );` // $s = r$
2) `grb::eWiseMul( s, alpha, v, semiring );` // s += *alpha* .* v

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Automatic non-blocking mode (Mastoras et al., '22):

- *lazily* evaluate ALP/GraphBLAS calls, no ALP program changes!

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );` // $s = r$
2) `grb::eWiseMul( s, alpha, v, semiring );` // s += *alpha* .* v

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Automatic non-blocking mode (Mastoras et al., '22): humble ✓

- *lazily* evaluate ALP/GraphBLAS calls, no ALP program changes!

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );` // $s = r$
2) `grb::eWiseMul( s, alpha, v, semiring );` // s += *alpha* .* v

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

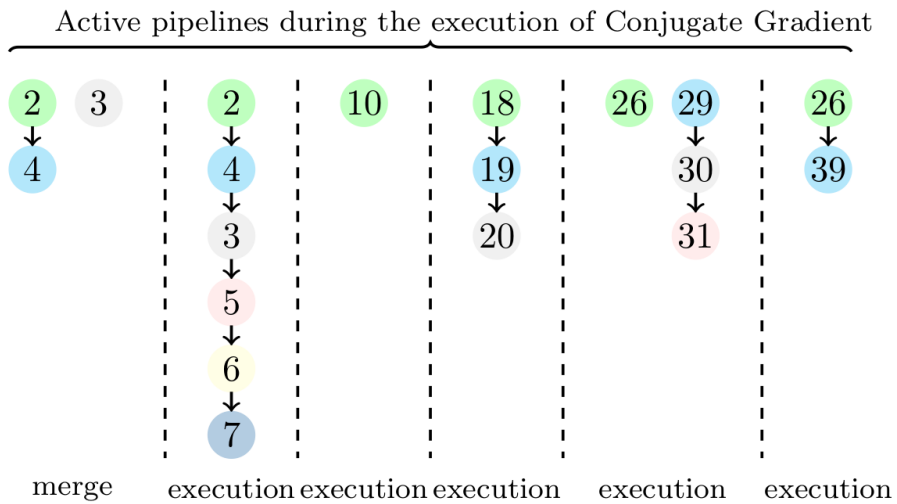
Automatic non-blocking mode (Mastoras et al., '22): humble ✓

- *lazily* evaluate ALP/GraphBLAS calls, no ALP program changes!
- dynamically trigger pipelines when required, **automatically fuse**.

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

The nonblocking backend

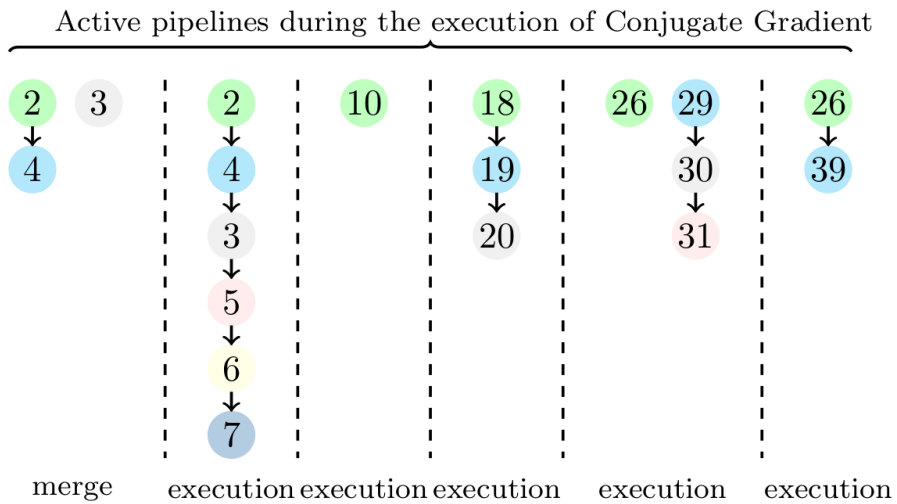
Dynamic **on-line** dependence analysis:




```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

The nonblocking backend

Dynamic **on-line** dependence analysis:



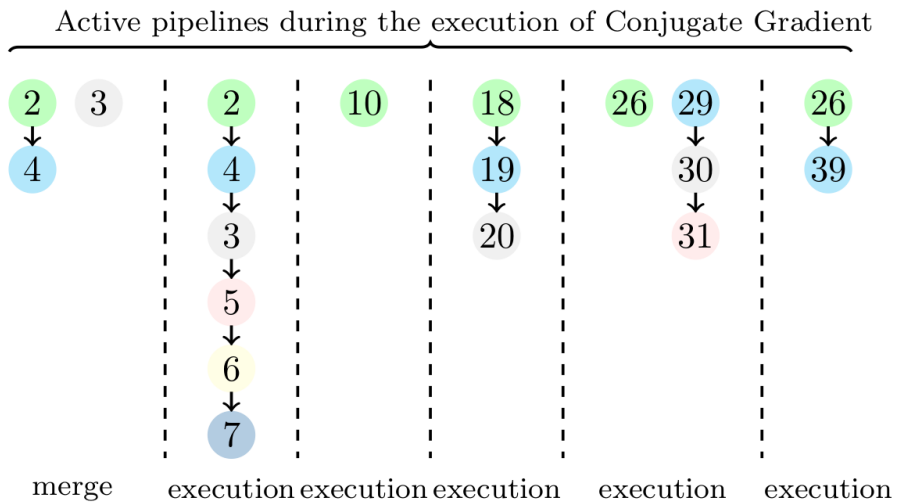
Fused execution can **cross control flow**:

- e.g., lines 26, 39 cross an if-statement;

```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

The nonblocking backend

Dynamic **on-line** dependence analysis:



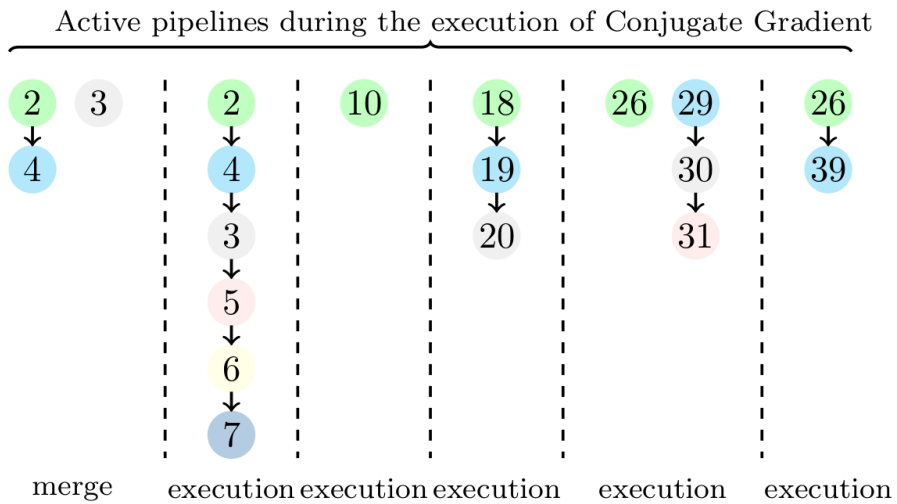
Fused execution can **cross control flow**:

- e.g., lines 26, 39 cross an if-statement;
- elect chunk size s.t. all vectors cached;

```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

The nonblocking backend

Dynamic **on-line** dependence analysis:



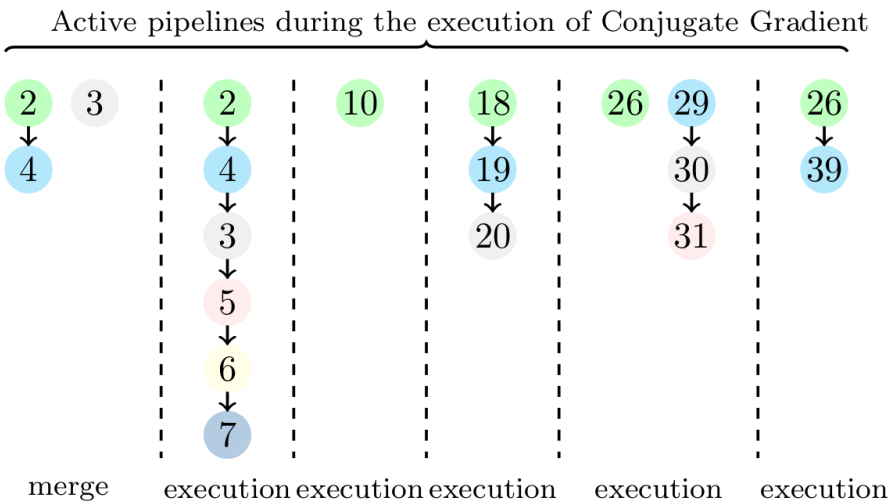
Fused execution can **cross control flow**:

- e.g., lines 26, 39 cross an if-statement;
- elect chunk size s.t. all vectors cached;
- reduce #threads if vectors too small;

```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

The nonblocking backend

Dynamic **on-line** dependence analysis:



Fused execution can **cross control flow**:

- e.g., lines 26, 39 cross an if-statement;
- elect chunk size s.t. all vectors cached;
- reduce #threads if vectors too small;
- **analytic model** automatically selects performance parameters: ✓.

```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

Performance

Speedup relative to sequential ALP (v0.5), vs. state-of-the-art

- Conjugate Gradient solve, two-socket x86, 44 cores:

	gyro_m	G2_circuit	bundle_adj	ecology2	Queen_4147
GSL	0.84	0.95	0.89	0.91	0.92
blocking ALP	2.30	4.53	**12.7**	6.91	17.5
SuiteSparse:GraphBLAS	1.57	1.11	5.82	3.52	11.6
Eigen	5.21	2.57	1.61	1.94	9.20
non-blocking ALP	**5.57**	**9.75**	2.87	**13.7**	**18.6**

Ref.: Mastoras, Anagnostidis, and Y., "Nonblocking execution in GraphBLAS", IEEE IPDPSW 2022.
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

Performance

Speedup relative to sequential ALP (v0.5), vs. state-of-the-art

- Conjugate Gradient solve, two-socket x86, 44 cores:

	gyro_m	G2_circuit	bundle_adj	ecology2	Queen_4147
GSL	0.84	0.95	0.89	0.91	0.92
blocking ALP	2.30	4.53	**12.7**	6.91	17.5
SuiteSparse:GraphBLAS	1.57	1.11	5.82	3.52	11.6
Eigen	5.21	2.57	1.61	1.94	9.20
non-blocking ALP	**5.57**	**9.75**	2.87	**13.7**	**18.6**

The nonblocking backend:

- speedup up to pipeline depth if $nz = \Theta(n)$;
- up to 2.43× vs. blocking, 0.49–8.78× vs. SuiteSparse:GraphBLAS;
- 2.87–7.06× vs. Eigen– which **also performs loop fusion**.

Ref.: Mastoras, Anagnostidis, and Y., "Nonblocking execution in GraphBLAS", IEEE IPDPSW 2022.
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

Performance

Speedup relative to sequential ALP (v0.5), vs. state-of-the-art

- Conjugate Gradient solve, two-socket x86, 44 cores:

	gyro_m	G2_circuit	bundle_adj	ecology2	Queen_4147
GSL	0.84	0.95	0.89	0.91	0.92
blocking ALP	2.30	4.53	**12.7**	6.91	17.5
SuiteSparse:GraphBLAS	1.57	1.11	5.82	3.52	11.6
Eigen	5.21	2.57	1.61	1.94	9.20
non-blocking ALP	**5.57**	**9.75**	2.87	**13.7**	**18.6**

The nonblocking backend:

- speedup up to pipeline depth if $nz = \Theta(n)$;
- up to 2.43× vs. blocking, 0.49–8.78× vs. SuiteSparse:GraphBLAS;
- 2.87–7.06× vs. Eigen– which **also performs loop fusion**.

Similar results for PageRank and sparse deep neural network inference.

Ref.: Mastoras, Anagnostidis, and Y., "Nonblocking execution in GraphBLAS", IEEE IPDPSW 2022.
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also `http://albert-jan.yzelman.net/alp/v0.8-preview/`)

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also `http://albert-jan.yzelman.net/alp/v0.8-preview/`)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
            grb::Vector< IOType >       &x,
      const grb::Matrix< NonzeroType > &A,
      const grb::Vector< InputType >   &b,
```

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also `http://albert-jan.yzelman.net/alp/v0.8-preview/`)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
            grb::Vector< IOType >       &x,
      const grb::Matrix< NonzeroType > &A,
      const grb::Vector< InputType >   &b,
      const std::function< grb::RC(
          grb::Vector< IOType > &,
          const grb::Vector< IOType > &
      ) > &Minv,
```

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also `http://albert-jan.yzelman.net/alp/v0.8-preview/`)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
            grb::Vector< IOType >       &x,
      const grb::Matrix< NonzeroType > &A,
      const grb::Vector< InputType >   &b,
      const std::function< grb::RC(
          grb::Vector< IOType > &,
          const grb::Vector< IOType > &
      ) > &Minv,
      const size_t max_iterations, ResidualType tol,
```

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also `http://albert-jan.yzelman.net/alp/v0.8-preview/`)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
            grb::Vector< IOType >       &x,
      const grb::Matrix< NonzeroType > &A,
      const grb::Vector< InputType >   &b,
      const std::function< grb::RC(
          grb::Vector< IOType > &,
          const grb::Vector< IOType > &
      ) > &Minv,
      const size_t max_iterations, ResidualType tol,
            size_t &iterations,    ResidualType &residual,
```

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also `http://albert-jan.yzelman.net/alp/v0.8-preview/`)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
            grb::Vector< IOType >       &x,
      const grb::Matrix< NonzeroType > &A,
      const grb::Vector< InputType >   &b,
      const std::function< grb::RC(
          grb::Vector< IOType > &,
          const grb::Vector< IOType > &
      ) > &Minv,
      const size_t max_iterations, ResidualType tol,
            size_t &iterations,    ResidualType &residual,
      grb::Vector< IOType > &buffer1, grb::Vector< IOType > &buffer2,
      grb::Vector< IOType > &buffer3, grb::Vector< IOType > &buffer4
);
```

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1
ALP	0.691	0.483	1.95	5.29	7.86	36.0

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

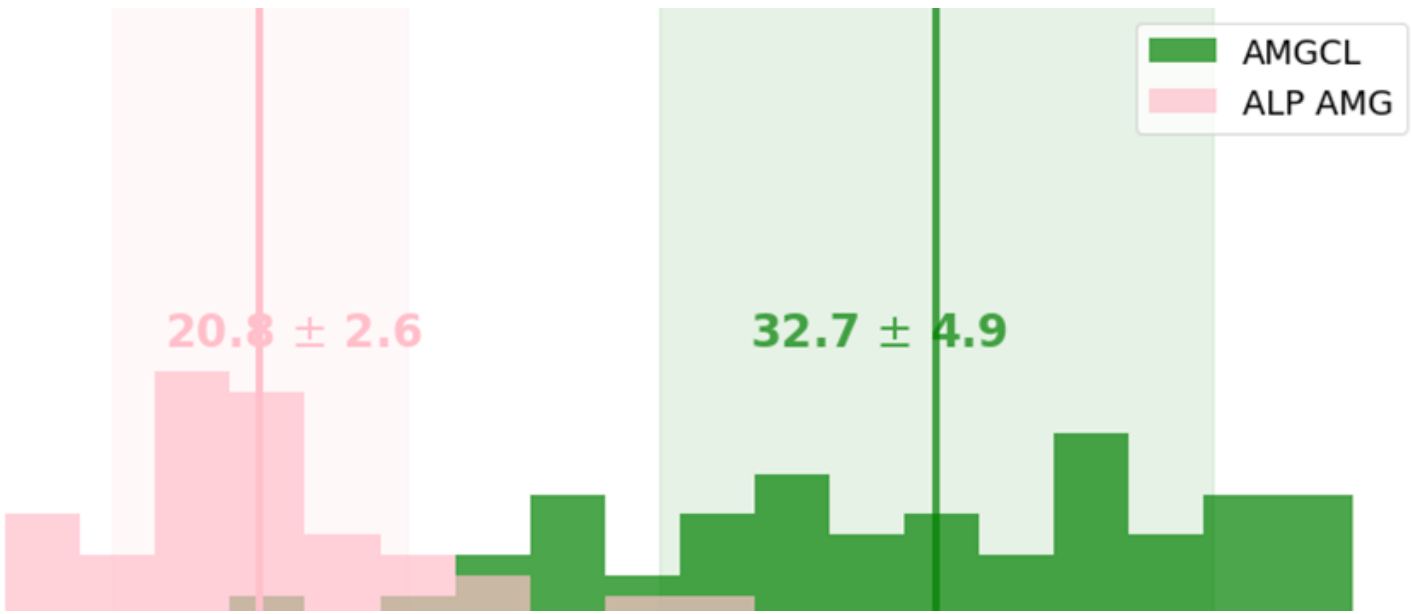
ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1
ALP	0.691	0.483	1.95	5.29	7.86	36.0

ALP up to $28.5\times$ faster vs. Eigen and 6.96, $2.01\times$ vs. hand-optimised.

ALP PCG with AMG preconditioning

AMG PCG in ALP vs. AMGCL on an internal problem:

- Matrix size n approx. $217M$, 460M nonzeroes.
- explicit coarsening / restriction matrices applied within ALP;
- compiled using the non-blocking backend.



(obtained using the 2-socket ARM machine, 96 cores; similar for x86)

ALP PCG with ILU(tresh)

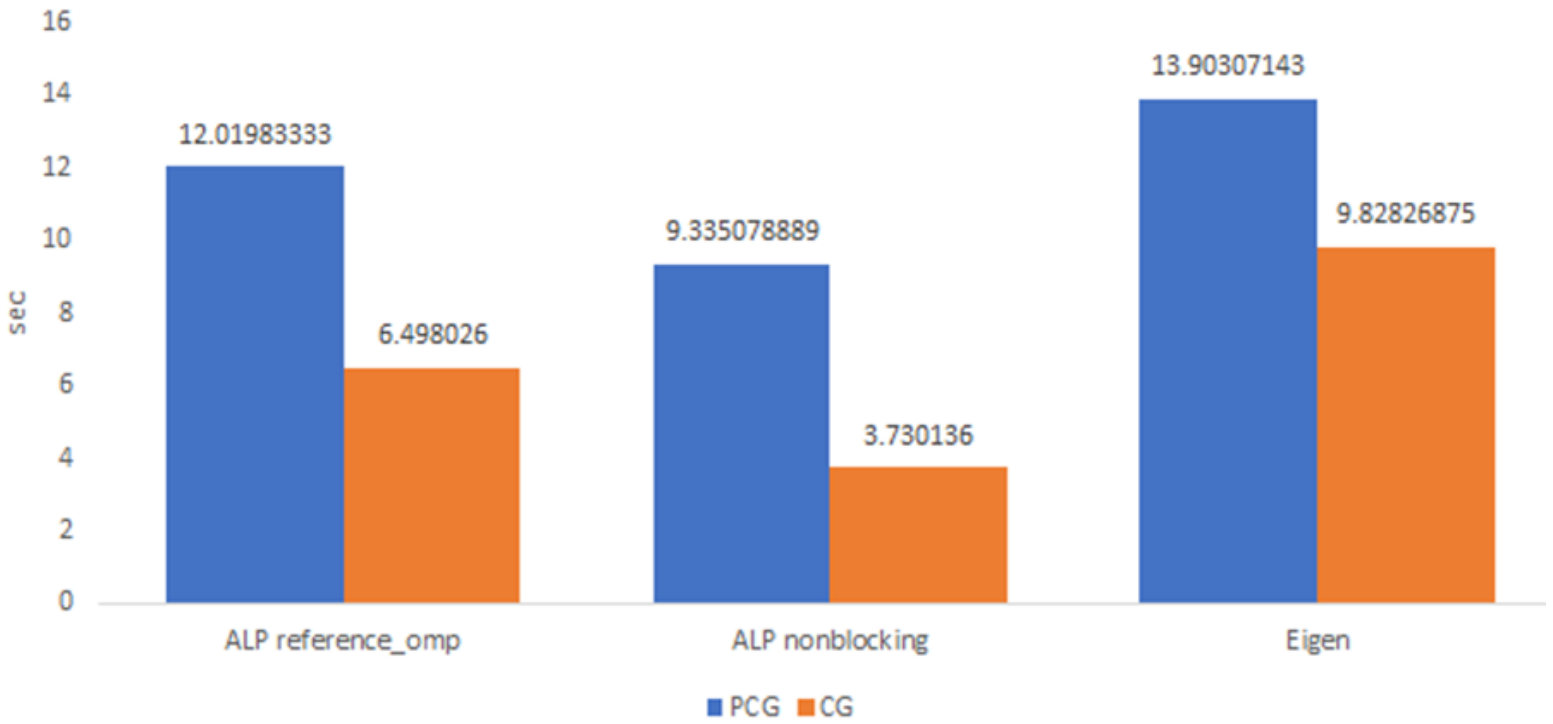
Tresholded ILU PCG in ALP vs. Eigen on the same internal problem:

- forward/backward solves $Lx = b$ and $L^T y = x$ are outside of ALP;
  - optimised using Sympiler/HDagg
- they are called from the standard ALP PCG algorithm.

ALP PCG with ILU(tresh)

Tresholded ILU PCG in ALP vs. Eigen on the same internal problem:

- forward/backward solves $Lx = b$ and $L^T y = x$ are outside of ALP;
  - optimised using Sympiler/HDagg
- they are called from the standard ALP PCG algorithm.

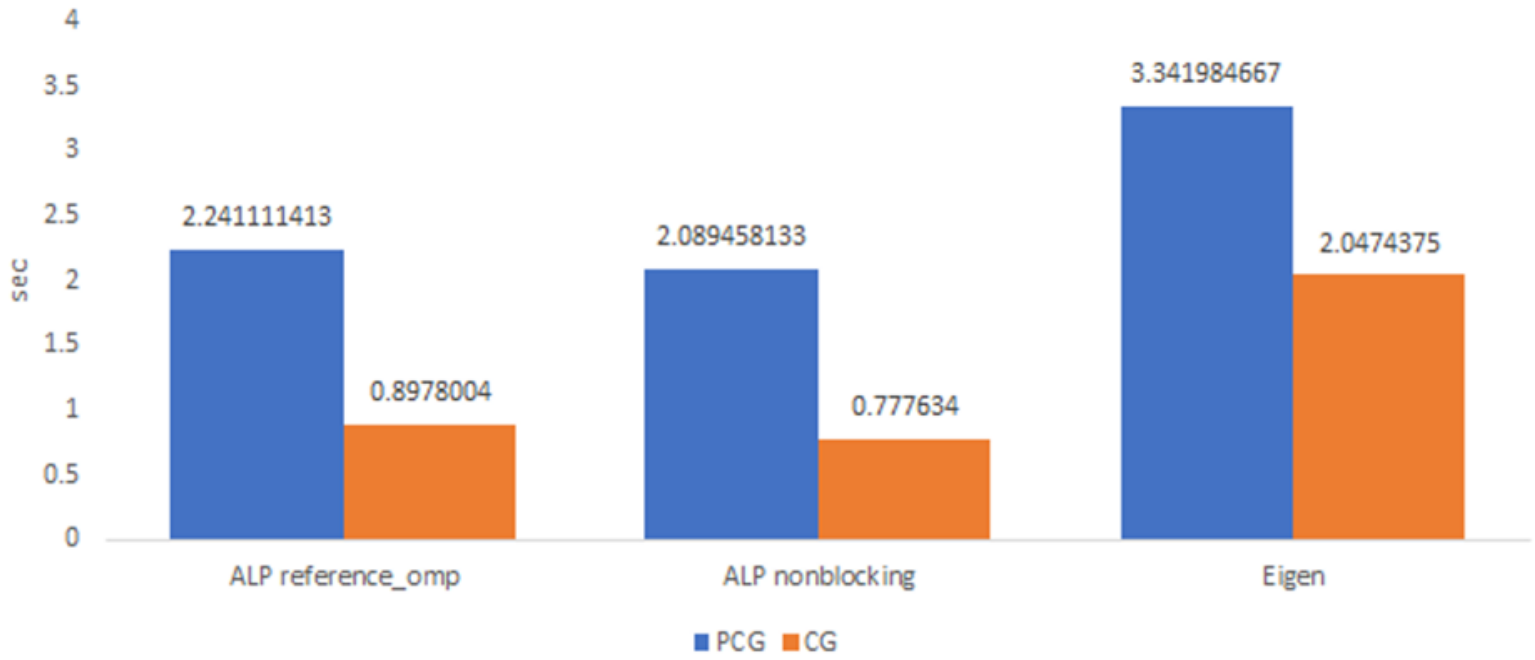


(obtained on a 2-socket Intel x86 machine, 32 threads)

ILU(tresh) PCG on x86

Two-socket Intel x86, smaller problem

- Matrix size n approx. 44M, 100M nonzeroes.

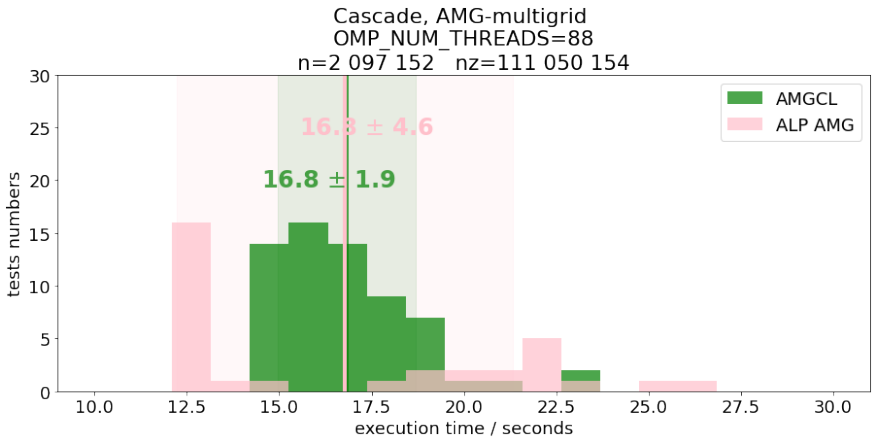


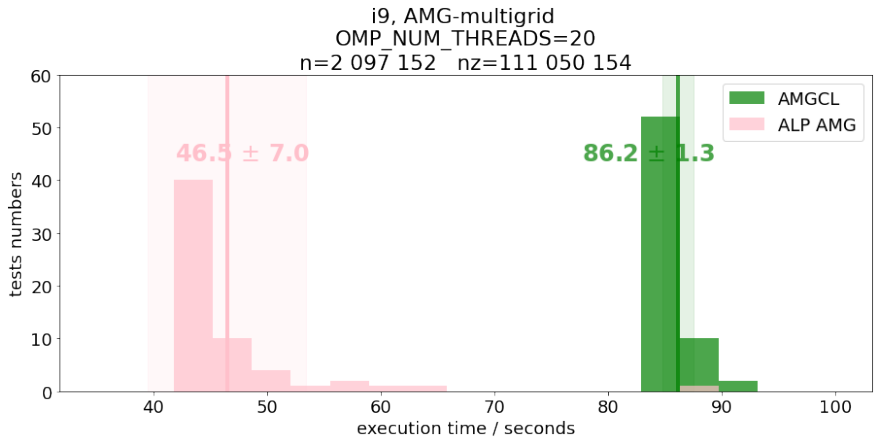
(obtained on a 2-socket Intel x86 machine, 32 threads)

Here, PCG Eigen: 3.36, Eigen+Hdagg: 2.08, ALP+Hdagg: 0.778 s.

AMG PCG on x86

Two-socket and one-socket performance on Intel x86:





- Cascade Lake, 64-byte vectorisation, default ALP settings

Performance semantics

Every backend defines **performance semantics**:

- work;

Performance semantics

Every backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;

Performance semantics

Every backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;
- whether system calls such as (de-)allocations may be made.

Performance semantics

Every backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;
- whether system calls such as (de-)allocations may be made.

Performance semantics help

- guide programmers to express the best possible algorithm;

Performance semantics

Every backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;
- whether system calls such as (de-)allocations may be made.

Performance semantics help

- guide programmers to express the best possible algorithm;
- gauge scalability: compute resources vs. problem size;
- expose trade-off opportunities: e.g., speed vs. memory;

Performance semantics

Every backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;
- whether system calls such as (de-)allocations may be made.

Performance semantics help

- guide programmers to express the best possible algorithm;
- gauge scalability: compute resources vs. problem size;
- expose trade-off opportunities: e.g., speed vs. memory;
- automatic choice of algorithms and backends.

Performance semantics

Every ALP program can be **systematically costed**:

Primitive	Work	Ops	Data movement	Reductions
setElement(x, y, i)	1	-	1	no
set(x, y)	$\min\{n, nz_x + nz_y\}$	-	$nz_x + nz_y$ or $n + nz_y$	no
clear(x)	nz_x	-	nz_x	no
apply$(z, x, y, \odot/M)$	$\min\{n, nz_x + nz_y\}$	$nz_{x\cap y}$	$2\min\{n, nz_x + nz_y\}$ $+ nz_{x\cup y}$	no
foldl$(y, x, \odot/M)$ foldr$(x, y, \odot/M)$	nz_x	$nz_{x\cap y}$	$2nz_x$	no
foldl$(y, \alpha, \odot/M)$ foldr$(\alpha, y, \odot/M)$	nz_y	nz_y	nz_y	no
foldl(α, y, M) foldr(y, α, M)	nz_y	nz_y	nz_y	yes
mul(z, x, y, R)	$\min\{nz_x, nz_y\}$	$nz_{x\cap y}$	$2\min\{nz_x, nz_y\} + nz_{x\cap y}$	no
dot$(z, x, y, (M, \odot))$	n	$2n$	$2n$	yes
dot(z, x, y, R)	$\min\{nz_x, nz_y\}$	$2 \cdot nz_{x\cap y}$	$2\min\{nz_x, nz_y\}$	

Level-1 primitives and their costs, excluding masking. Similar tables exist for level-2 and level-3 primitives.

Ref.: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation by Y., D. Di Nardo, J. M. Nash, and W. J. Suijlen (2020).

Performance semantics

Every container has **memory use semantics**:

- "static" costs proportional to container sizes;
- "dynamic" costs proportional to container capacities.

Performance semantics

Every container has **memory use semantics**:

- "static" costs proportional to container sizes;
- "dynamic" costs proportional to container capacities.

Capacities are **optional** during container construction:

```
grb::Vector< bool > s( n, 1 );
grb::Matrix< void > L( n, n, nz );
```

Out of memory errors throw exceptions; primitives return error codes.

Performance semantics

Every container has **memory use semantics**:

- "static" costs proportional to container sizes;
- "dynamic" costs proportional to container capacities.

Capacities are **optional** during container construction:

```
grb::Vector< bool > s( n, 1 );
grb::Matrix< void > L( n, n, nz );
```

Out of memory errors throw exceptions; primitives return error codes.

Capacities:

- are **lower bounds**; grb::capacity(s) ≥ 1;
- may **increase** through grb::resize, updates memory use semantics;
- Any request to decrease capacity thus **may be ignored**.

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

- grb :: is_associative < Operator >::value, `true` iff $(a \odot b) \odot c = a \odot (b \odot c)$;
- grb :: is_idempotent< Operator >::value, `true` iff $a \odot a = a$;
- grb :: is_monoid< T >::value, `true` iff T is a monoid;
- ...

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

- grb::is_associative< Operator >::value, `true` iff $(a \odot b) \odot c = a \odot (b \odot c)$;
- grb::is_idempotent< Operator >::value, `true` iff $a \odot a = a$;
- grb::is_monoid< T >::value, `true` iff T is a monoid;
- ...

Algebraic type traits transfer to richer algebraic structures:

- grb::is_commutative< grb::operators::add< **double** > >::value? ✓

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

- grb::is_associative< Operator >::value, `true` iff $(a \odot b) \odot c = a \odot (b \odot c)$;
- grb::is_idempotent< Operator >::value, `true` iff $a \odot a = a$;
- grb::is_monoid< T >::value, `true` iff T is a monoid;
- ...

Algebraic type traits transfer to richer algebraic structures:

- grb::is_commutative< grb::operators::add< **double** > >::value? ✓,
  therefore
  ```
  is_commutative< Monoid<
     operators::add< double >, identities::zero >
  >::value? ✓
  ```

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

- grb::is_associative< Operator >::value, `true` iff $(a \odot b) \odot c = a \odot (b \odot c)$;
- grb::is_idempotent< Operator >::value, `true` iff $a \odot a = a$;
- grb::is_monoid< T >::value, `true` iff T is a monoid;
- ...

Algebraic type traits transfer to richer algebraic structures:

- grb::is_commutative< grb::operators::add< **double** > >::value? ✓,
  therefore
  ```
  is_commutative< Monoid<
     operators::add< double >, identities::zero >
  >::value? ✓
  ```

These are all **compile-time constant** (through C++11 constexpr):

- similar to the standard C++11 *type traits*.

Algebraic type traits

Algebraic type traits help

- detect programmer errors,
- decide which optimisations are applicable, and
- reject expressions without recipe for auto-parallelisation.

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities ::one > myMonoid;
  - is_associative < operators:: divide< **int** > >::value? ✗

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities ::one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators :: right_assign<**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities ::one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators :: right_assign<**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗
- reducing a sparse vector to a scalar without a monoid structure.
  - operators :: add< **double** > addOp;
  - **double** alpha = 0; foldl (alpha, x, addOp);
  - is_monoid< addOp >::value? ✗

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities ::one > myMonoid;
  - is_associative < operators:: divide< **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators:: right_assign<**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗
- reducing a sparse vector to a scalar without a monoid structure.
  - operators :: add< **double** > addOp;
  - **double** alpha = 0; foldl (alpha, x, addOp);
  - is_monoid< addOp >::value? ✗, since
    parallelisation requires identity *and* associativity!
    foldl (alpha, x, addMon); ✓

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities ::one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators :: right_assign<**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗
- reducing a sparse vector to a scalar without a monoid structure.
  - operators :: add< **double** > addOp;
  - **double** alpha = 0; foldl (alpha, x, addOp);
  - is_monoid< addOp >::value? ✗, since parallelisation requires identity *and* associativity!

    foldl (alpha, x, addMon); ✓

Above errors are all **compile-time** (through C++11 static_assert), with

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities ::one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators :: right_assign<**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗
- reducing a sparse vector to a scalar without a monoid structure.
  - operators :: add< **double** > addOp;
  - **double** alpha = 0; foldl (alpha, x, addOp);
  - is_monoid< addOp >::value? ✗, since parallelisation requires identity *and* associativity!

    foldl (alpha, x, addMon); ✓

Above errors are all **compile-time** (through C++11 static_assert), with **clear error messages**.

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$?

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot$ = min? Replace eWiseApply with grb::set(z, x);!

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$? Replace eWiseApply with grb::set(z, x);!
  - every time S is **idempotent**;

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot$ = min? Replace eWiseApply with grb::set(z, x);!
  - every time S is **idempotent**;
- ...

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$? Replace eWiseApply with grb::set(z, x);!
  - every time S is **idempotent**;
- ...

These optimisations are applied at compile-time,

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$? Replace eWiseApply with grb::set(z, x);!
  - every time S is **idempotent**;
- ...

These optimisations are applied at compile-time,

without requiring programmer knowledge or intervention.

Ex.: Y & Bisseling '10; Y & Roose '14; Y, Bisseling, Roose, Meerbergen '14; Y, Roose, Meerbergen '14; ...