Algebraic Programming for Tensor Computations

Albert-Jan N. Yzelman[1] Denis Jelovina[1] Aristeidis Mastoras[1]
Xiaohe Niu[2] Daniele G. Spampinato[1]

[1]Computing Systems Laboratory, Huawei Research, Zürich

[2]Dept. of Mathematics, ETH Zürich

15th of May, MS83, SIAM LA24

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;
- allow compile-time introspection 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;
- allow compile-time introspection of algebraic information;
- automatically 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;
- allow compile-time introspection of algebraic information;
- automatically 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 into scalar or vector (fold), mxv.

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 into scalar or vector (fold), mxv.

Containers are similar 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 into scalar or vector (fold), mxv.

Containers are similar 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

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 into scalar or vector (fold), mxv.

Containers are similar 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**, containers have **capacities**

```
grb::Vector< std::pair< int, double > > pairs( n );
grb::Vector< bool > s( n, 1 );      // nz cap: one
grb::Matrix< void > L( n, n, nz ); // nz cap: nz
```

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 into scalar or vector (fold), mxv.

Containers are similar 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**, containers have **capacities** and **IDs**:

```
grb::Vector< std::pair< int, double > > pairs( n );
grb::Vector< bool > s( n, 1 );      // nz cap: one
grb::Matrix< void > L( n, n, nz ); // nz cap: nz
std::cout << "s has ID " << grb::getID( s ) << "\n";
```

Basics

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

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

Basics

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

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

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

- enables auto-parallelisation and other automatic optimisations.

Basics

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

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

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

- enables auto-parallelisation and other automatic optimisations.

More complex structures may be **composed**:

```
grb::Monoid<
   grb::operators::add< double >, grb::identities::zero
> addMon;
```

Basics

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

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

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

- enables auto-parallelisation and other automatic optimisations.

More complex structures may be **composed**:

```
grb::Monoid<
   grb::operators::add< double >, grb::identities::zero
> addMon;

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

Basics

Algebraic primitives operate on algebraic containers:

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

Basics

Algebraic primitives operate on algebraic containers:

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

Semantics may change based on **required** algebraic structures:

- grb::eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, x_i\}, \ \forall i$
- grb::eWiseApply(z, x, y, minOp); // $z_{n/2} = \min\{x_{n/2}, y_{n/2}\}$

Basics

Algebraic primitives operate on algebraic containers:

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

Semantics may change based on **required** algebraic structures:

- grb::eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, x_i\}, \ \forall i$
- grb::eWiseApply(z, x, y, minOp); // $z_{n/2} = \min\{x_{n/2}, y_{n/2}\}$
- grb::eWiseApply(z, x, y, addMon);// $z_{n/2} = x_i + y_i, \ z_{i \neq n/2} = x_i$

Basics

Algebraic primitives operate on algebraic containers:

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

Semantics may change based on **required** algebraic structures:

- grb::eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, x_i\}, \ \forall i$
- grb::eWiseApply(z, x, y, minOp); // $z_{n/2} = \min\{x_{n/2}, y_{n/2}\}$
- grb::eWiseApply(z, x, y, addMon);// $z_{n/2} = x_i + y_i, \ z_{i \neq n/2} = x_i$
- grb::mxv(y, A, x, mySemiring); // $y \ += Ax$; in-place

Basics

Algebraic primitives operate on algebraic containers:

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

Semantics may change based on **required** algebraic structures:

- grb::eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, x_i\}, \ \forall i$
- grb::eWiseApply(z, x, y, minOp); // $z_{n/2} = \min\{x_{n/2}, y_{n/2}\}$
- grb::eWiseApply(z, x, y, addMon);// $z_{n/2} = x_i + y_i, \ z_{i \neq n/2} = x_i$
- grb::mxv(y, A, x, mySemiring); // $y \ += Ax$; in-place

All primitives except 'getters' such as grb::{size,nrows,nnz,capacity}:

- allow input & output **masks**, **descriptors**, and **phase** arguments;

Basics

Algebraic primitives operate on algebraic containers:

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

Semantics may change based on **required** algebraic structures:

- grb::eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, x_i\}, \ \forall i$
- grb::eWiseApply(z, x, y, minOp); // $z_{n/2} = \min\{x_{n/2}, y_{n/2}\}$
- grb::eWiseApply(z, x, y, addMon);// $z_{n/2} = x_i + y_i, \ z_{i \neq n/2} = x_i$
- grb::mxv(y, A, x, mySemiring); // $y \ += Ax$; in-place

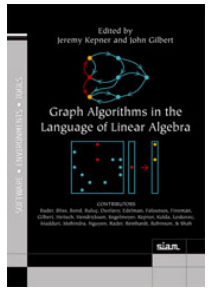
All primitives except 'getters' such as grb::{size,nrows,nnz,capacity}:

- allow input & output **masks**, **descriptors**, and **phase** arguments;
- return **error codes** such as for mismatching dimensions.

GraphBLAS

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

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

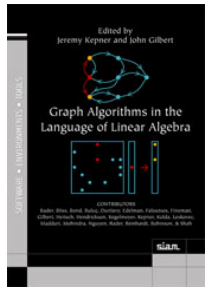



```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
                grb::Vector< VectorT >  &y,
          const grb::Matrix< NonzeroT > &A, const size_t k,
          const grb::Vector< VectorT >  &x,
                grb::Vector< VectorT >  &buffer,
          const Semiring                &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 );
      }
}
```

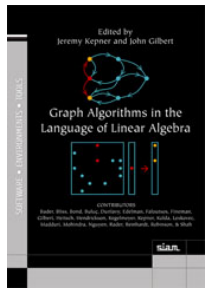
Solve different problems with different semirings:

- 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 );
      }
}
```

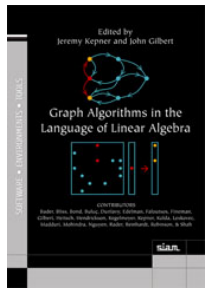
Solve different problems with different semirings:

- 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 );
      }
}
```

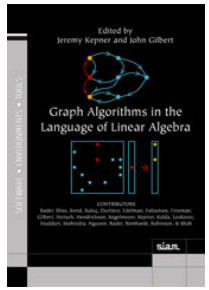
Solve different problems with different semirings:

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

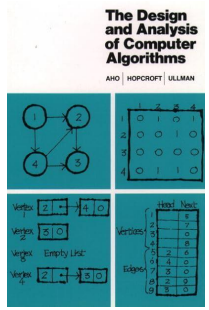
GraphBLAS

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.

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.

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.

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.

Beyond ALP/GraphBLAS

If generalised sparse linear algebra is so useful, what about:

- dense linear algebra,
- sparse multi-linear (tensor) algebra, and
- dense multi-linear algebra.

We will discuss each of these extensions in turn.

Beyond ALP/GraphBLAS

If generalised sparse linear algebra is so useful, what about:

- dense linear algebra,
- sparse multi-linear (tensor) algebra, and
- dense multi-linear algebra.

We will discuss each of these extensions in turn. Managing expections:

- the tensor extensions are still rather preliminary

ALP/Dense overview

Classic dense linear algebra:

ALP/Dense overview

Classic dense linear algebra:

- **submatrix selection**, **permutations**, random sampling, ...

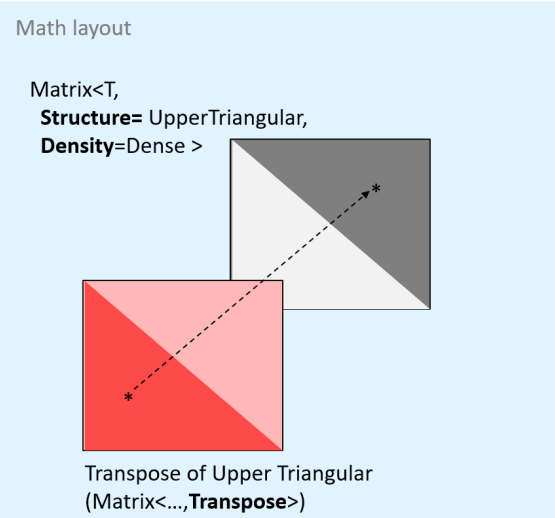
ALP/Dense overview

Classic dense linear algebra:

- **submatrix selection**, **permutations**, random sampling, ...

Requires ALP extensions: structures and views

- structures: general, triangular, banded, ... requires ontology



Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

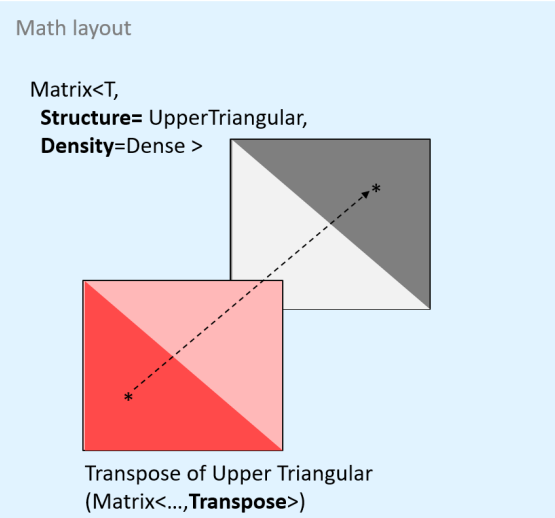
ALP/Dense overview

Classic dense linear algebra:

- **submatrix selection**, **permutations**, random sampling, ...

Requires ALP extensions: structures and views

- structures: general, triangular, banded, ... requires ontology
- views: transpose, masks, **permutation**, **submatrix selection**,
  - but also: outer products (two vectors), constant vectors (scalar)



Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

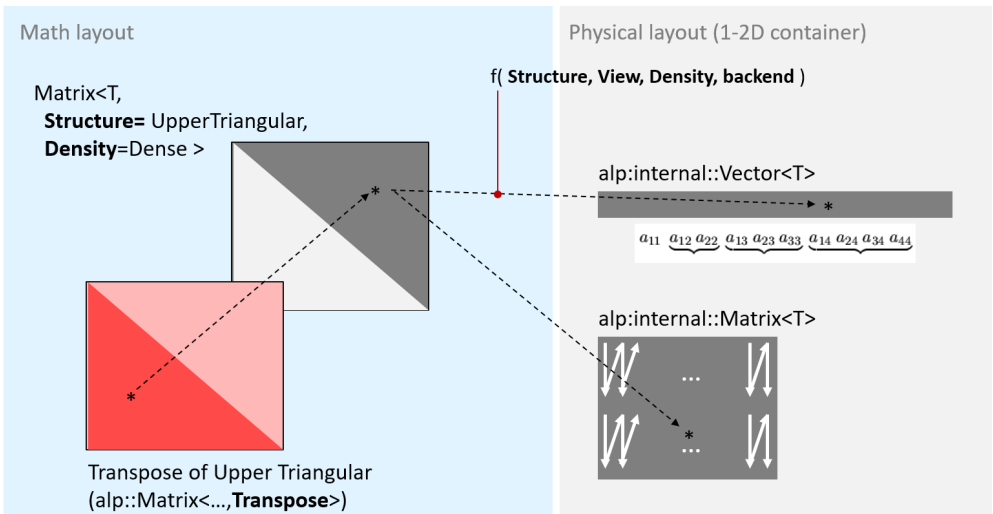
ALP/Dense overview

Classic dense linear algebra:

- **submatrix selection**, **permutations**, random sampling, ...

Requires ALP extensions: structures and views

- structures: general, triangular, banded, ... requires ontology
- views: transpose, masks, **permutation**, **submatrix selection**,
  - but also: outer products (two vectors), constant vectors (scalar)
- opaqueness: ALP controls layout



Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

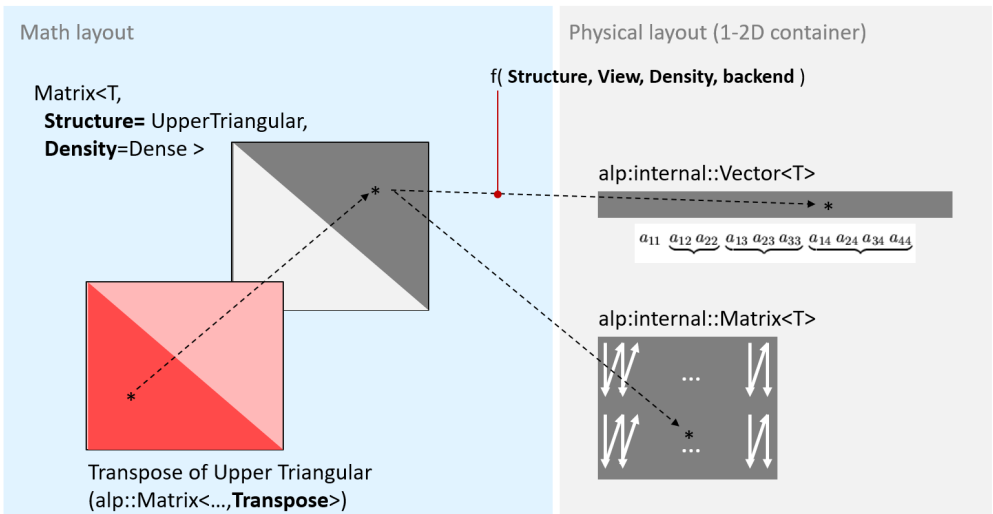
ALP/Dense overview

Classic dense linear algebra:

- **submatrix selection**, **permutations**, random sampling, ...

Requires ALP extensions: structures and views

- structures: general, triangular, banded, ... requires ontology
- views: transpose, masks, **permutation**, **submatrix selection**,
  - but also: outer products (two vectors), constant vectors (scalar)
- opaqueness: ALP controls layout, accesses via **parametric** xMFs



Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

ALP/Dense overview

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

ALP/Dense overview

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

ALP/Dense overview

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

- high-level MLIR dialect introducing, e.g., algebraic structures

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense overview

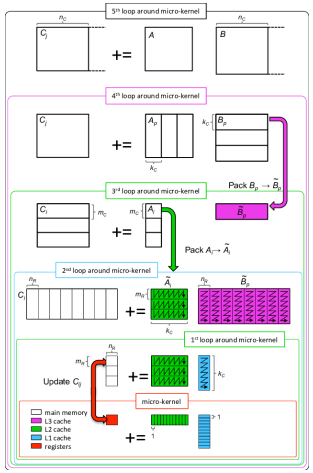
Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

- high-level MLIR dialect introducing, e.g., algebraic structures

3) BLIS-like approach to optimise MLIR or dispatch to BLAS:



Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense overview

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

- high-level MLIR dialect introducing, e.g., algebraic structures

3) BLIS-like approach to optimise MLIR or dispatch to BLAS:

- use offline (auto-)tuning, **once** per new architecture ✓

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense overview

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

- high-level MLIR dialect introducing, e.g., algebraic structures

3) BLIS-like approach to optimise MLIR or dispatch to BLAS:

- use offline (auto-)tuning, **once** per new architecture ✓

4) threads and/or processes **individually** follow steps 2-3;

- data distribution and parallelisation controlled by ALP.

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense overview

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)
2) when pipelines execute, instead first translate to MLIR;
   - high-level MLIR dialect introducing, e.g., algebraic structures
3) BLIS-like approach to optimise MLIR or dispatch to BLAS:
   - use offline (auto-)tuning, **once** per new architecture ✓
4) threads and/or processes **individually** follow steps 2-3;
   - data distribution and parallelisation controlled by ALP.

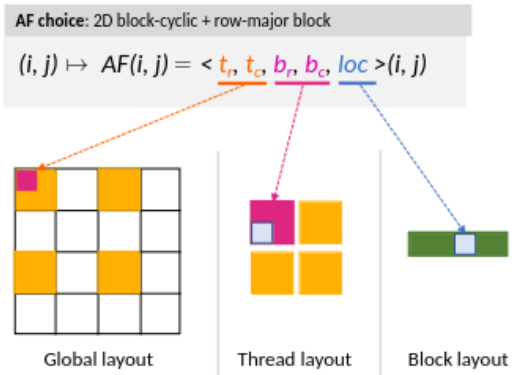
Between JIT and AOT: **delayed compilation or dispatching**

- high-level MLIR as an architecture-agnostic representation
- can be generated at run-time, following dynamic user control flow.

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense implementation

ALP/Dense employs **block-wise storage** via composable xMFs:




```
#pragma omp parallel
{
  thread = config::OMP::current_thread_ID();
  th_coo = da.getThreadCoords( thread );
  blk_grd = da.getLocalBlockGridDims( th_coo );

  for( auto &blk_coo: blk_grd ) {
    // Assuming shmem::seq == reference, Aloc has
   type
    // Matrix< ... , reference >
    auto Aloc = internal::get_view< shmem::seq >(
        A, th_coo, blk_coo
      );
      // Calls func on L.21
      eWiseMul( Aloc, alpha, beta, Sr );
} }
```

Fig.: storage illustration (left) and code sketch for $A \otimes \alpha \oplus \beta$ (right)

ALP/Dense:

- performs **compile-time simplification** of telescopic composed xMFs
- dispatches block computations to **sequential** hand-tuned BLAS
- performs **NUMA-aware allocation** of thread-owned blocks
- **blocking execution** within threads

Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

ALP/Dense evaluation

Expressivity: the API captures the following LA algorithms

- Householder tridiagonalisation, Cholesky, LU, QR, and more

Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

ALP/Dense evaluation

Expressivity: the API captures the following LA algorithms

- Householder tridiagonalisation, Cholesky, LU, QR, and more

Performance: 2× HiSilicon Kunpeng 920-**48**26 (4 NUMA domains)

- ALP/Dense (block size 128) compared to Netlib LAPACK;
- LAPACK linked to hand-tuned shared-memory parallel BLAS.

ALP/Dense evaluation

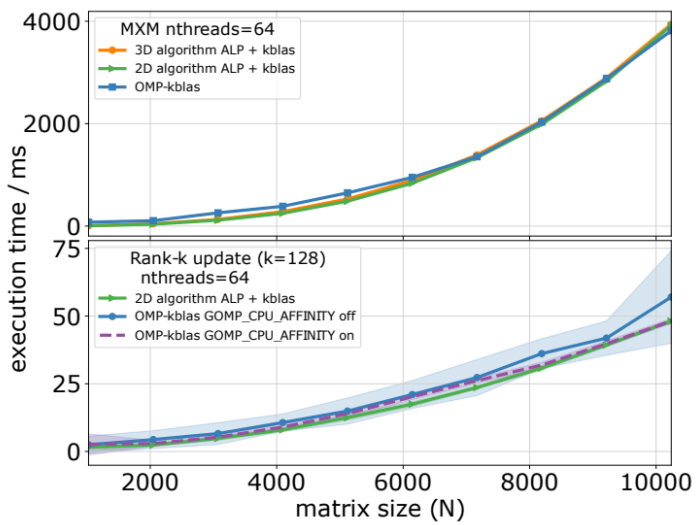
Expressivity: the API captures the following LA algorithms

- Householder tridiagonalisation, Cholesky, LU, QR, and more

Performance: 2× HiSilicon Kunpeng 920-**48**26 (4 NUMA domains)

- ALP/Dense (block size 128) compared to Netlib LAPACK;
- LAPACK linked to hand-tuned shared-memory parallel BLAS.

Dense matrix–matrix multiplication (left) and Cholesky decomposition (right)



ALP/Dense evaluation

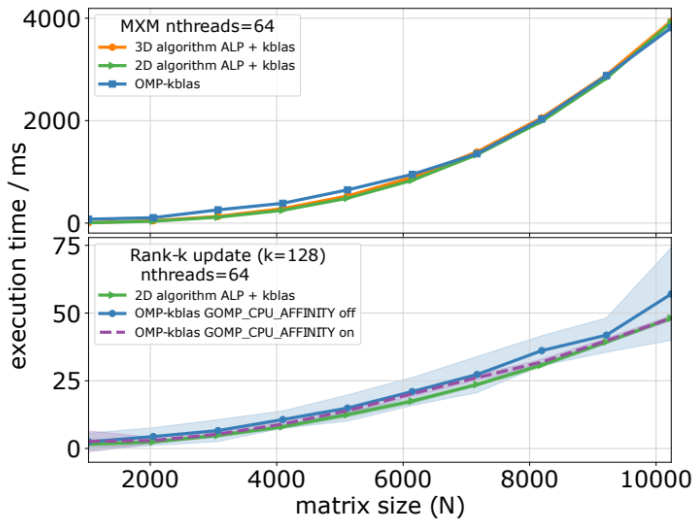
Expressivity: the API captures the following LA algorithms

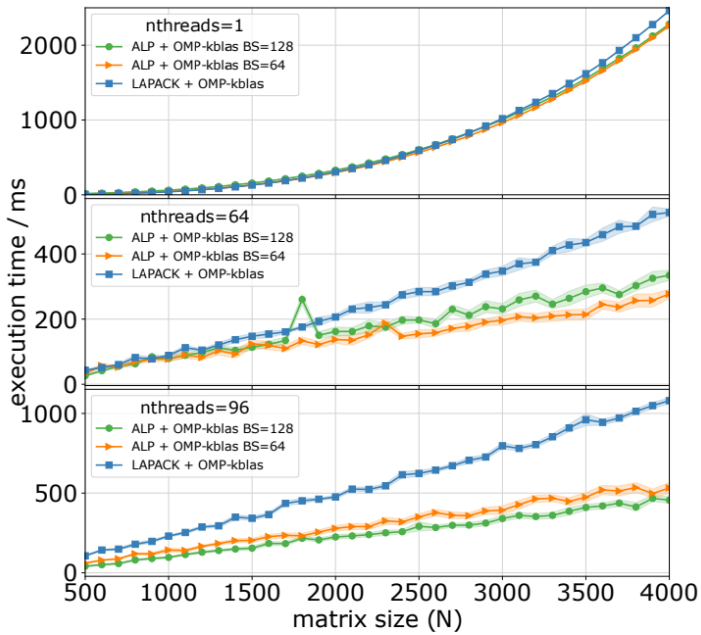
- Householder tridiagonalisation, Cholesky, LU, QR, and more

Performance: 2× HiSilicon Kunpeng 920-**48**26 (4 NUMA domains)

- ALP/Dense (block size 128) compared to Netlib LAPACK;
- LAPACK linked to hand-tuned shared-memory parallel BLAS.

Dense matrix–matrix multiplication (left) and Cholesky decomposition (right)





ALP/Dense evaluation

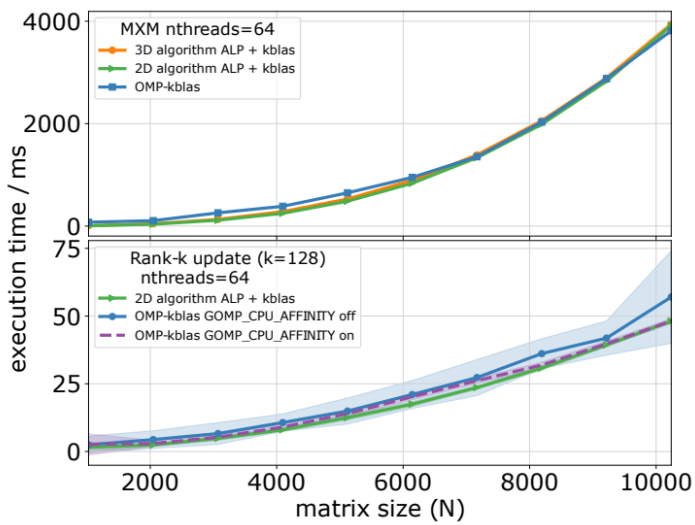
Expressivity: the API captures the following LA algorithms

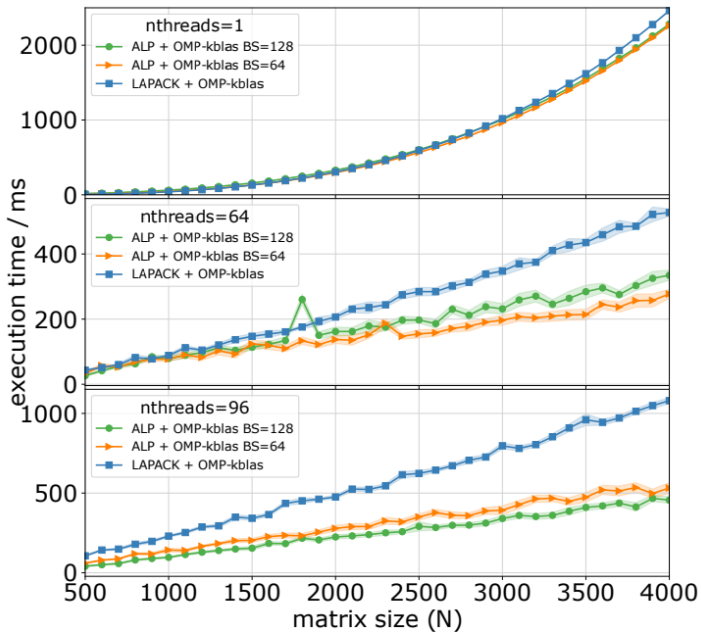
- Householder tridiagonalisation, Cholesky, LU, QR, and more

Performance: 2× HiSilicon Kunpeng 920-**48**26 (4 NUMA domains)

- ALP/Dense (block size 128) compared to Netlib LAPACK;
- LAPACK linked to hand-tuned shared-memory parallel BLAS.

Dense matrix–matrix multiplication (left) and Cholesky decomposition (right)





- ALP/Dense NUMA-aware auto-parallelisation: more stable, faster

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 Koster, Bisseling, and Y. on incremental matrix storages:

- 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:

- **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 Koster, Bisseling, and Y. on incremental matrix storages:

- 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:

- **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

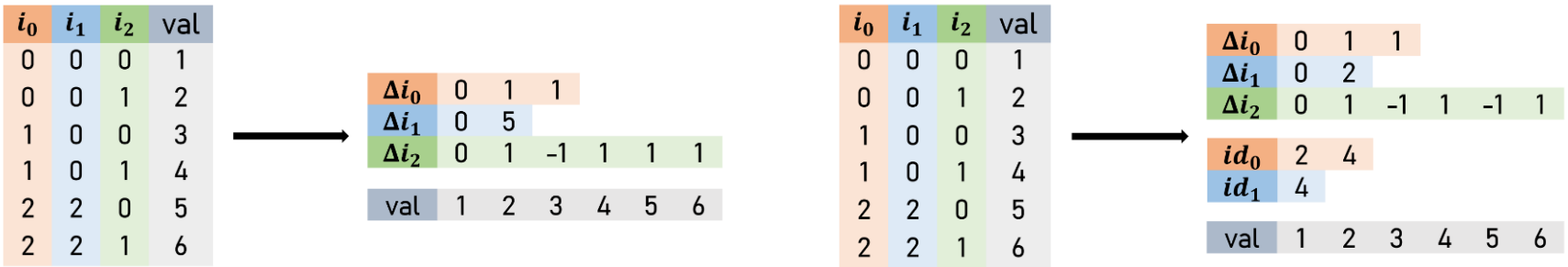


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position

Incremental sparse tensor storage schemes

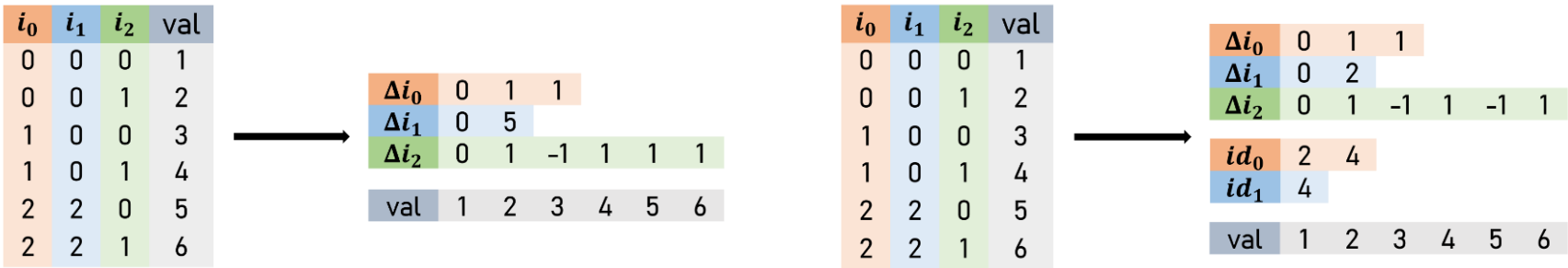


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position;

- start positions stored separately in 32 or 64 bits;
- increments may use 8 or 16 bits.

Incremental sparse tensor storage schemes

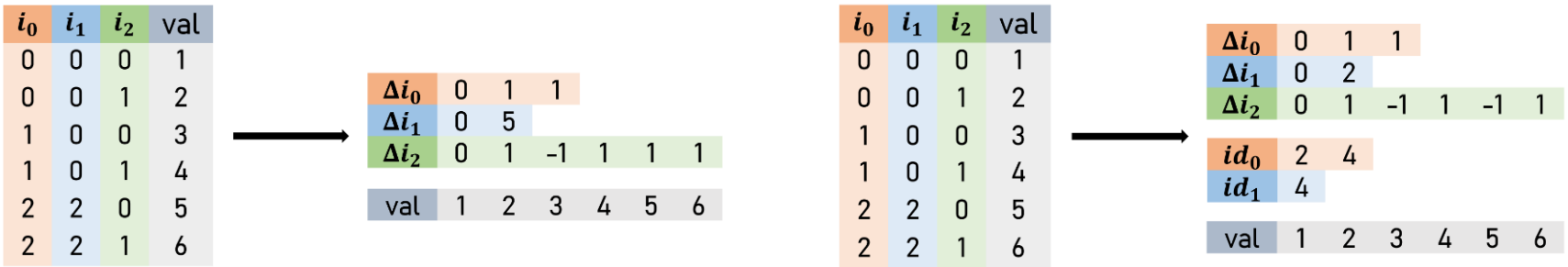


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position;

- start positions stored separately in 32 or 64 bits;
- increments may use 8 or 16 bits.

Different variants control which mode(s) change differently.

- MoSk-IF: via over- and under-flow (alike CBICRS, Y. et al.. '14);
- Ind-IF: per-mode array indicates when to jump (alike CSF)

Incremental sparse tensor storage schemes

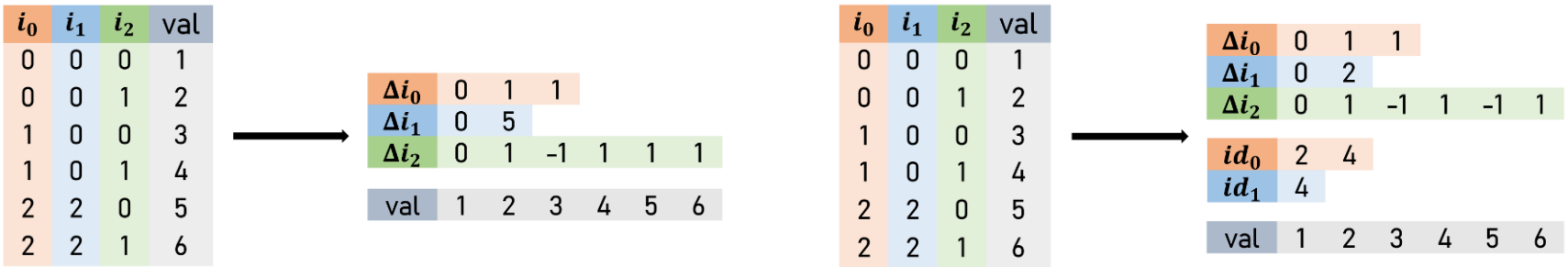


Fig. MoSk-IF (left) and Ind-IF (right)

Core idea: store coordinate increments vs. previous position;

- start positions stored separately in 32 or 64 bits;
- increments may use 8 or 16 bits.

Different variants control which mode(s) change differently.

- MoSk-IF: via over- and under-flow (alike CBICRS, Y. et al.. '14);
- 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

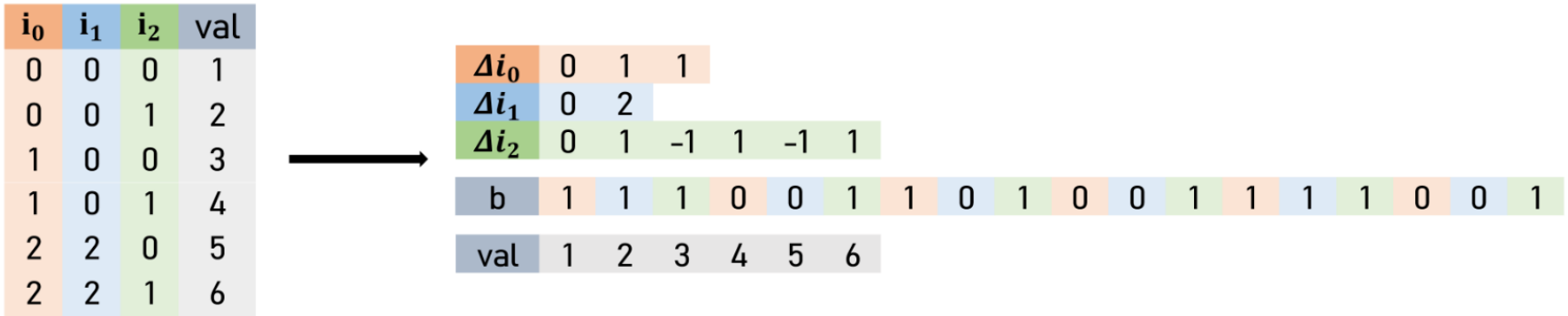


Fig. Bit-IF

Incremental sparse tensor storage schemes

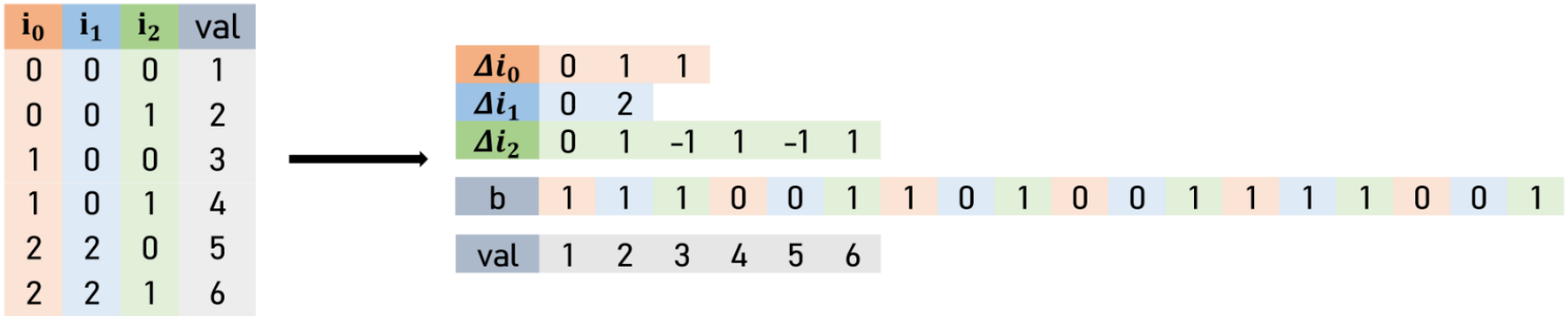


Fig. Bit-IF

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.

Incremental sparse tensor storage schemes

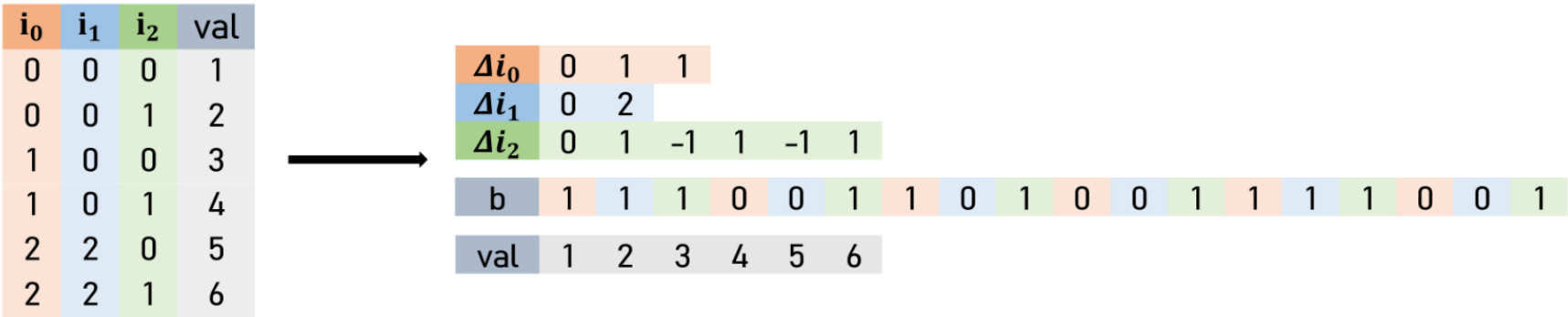


Fig. Bit-IF

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.

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

Incremental sparse tensor storage schemes

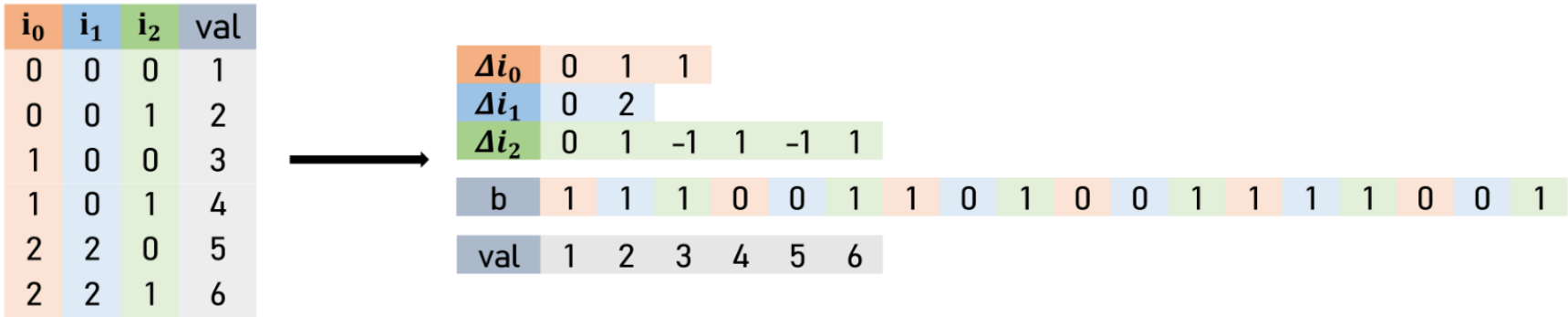


Fig. Bit-IF

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.

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

- MoSk: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-2} \tilde{q}_k \tilde{w}_{\text{inc}}\right] + \tilde{w}_{\text{inc}}\right)$, $\tilde{q}_k \geq q_k$, $\tilde{w}_{\text{inc}} \geq w_{\text{inc}}$;

Incremental sparse tensor storage schemes

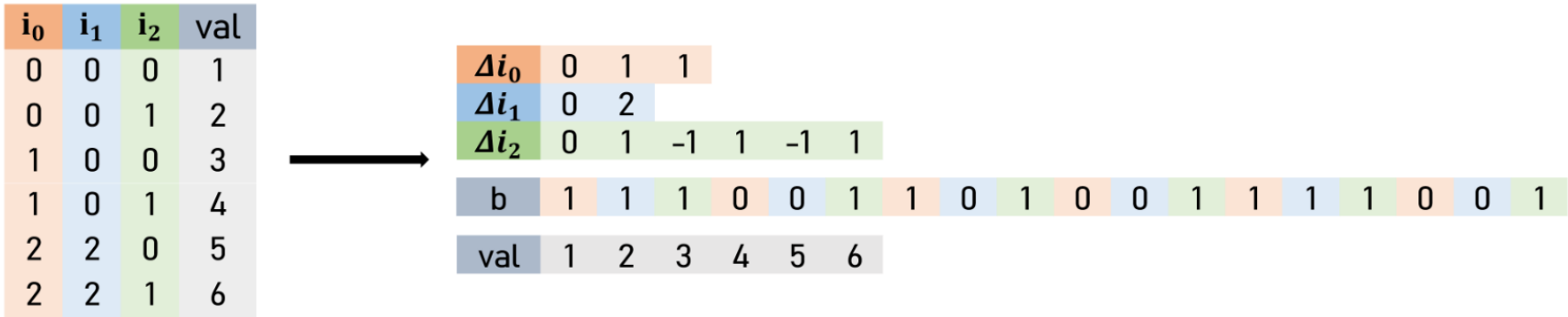


Fig. Bit-IF

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.

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: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-2} q_k w_{\text{inc}} + q'_k w_{\text{size\_t}}\right] + w_{\text{inc}}\right)$, $q'_k \leq q_k$;

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, $q_0 = 1/2, q_1 = 1/3, q_2 = 1/2$);
- let $w_{\text{val}} = 8$, $w_{\text{ind}} = 4$, $w_{\text{inc}} = 2$, and $w_{\text{size\_t}} = 8$ bytes.

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: $nnz\left(w_{\text{val}} + \left[\sum_{k=0}^{d-1} q_k w_{\text{inc}}\right] + d/8\right)$.

Results

From the analysis:

- all storage formats can improve on COO;
- if the $q_i \rightarrow 1$, Bit-IF is superior to MoSk-IF and Ind-IF
  - critical when blocking using Hilbert- or Z-curves

Results

From the analysis:

- all storage formats can improve on COO;
- if the $q_i \rightarrow 1$, Bit-IF is superior to MoSk-IF and Ind-IF
  - critical when blocking using Hilbert- or Z-curves;
- thesis contains similar analyses for data movement during TVM.

Results

From the analysis:

- all storage formats can improve on COO;
- if the $q_i \rightarrow 1$, Bit-IF is superior to MoSk-IF and Ind-IF
  - critical when blocking using Hilbert- or Z-curves;
- thesis contains similar analyses for data movement during TVM.

Actual compression on ten tensors from SPLATT:

Tensor	LBNL-Network	NIPS	Uber Pickups	$\cdots$	Delecious	NELL-1
COO	0.0443	0.0693	0.0740		3.1321	2.6748
Bit-IF	0.0693	0.0500	0.0519		2.6748	2.0816
Ratio	1.63	1.38	1.42		1.31	1.28

Table Storage requirement in Gbits, and the Bit-IF compression ratio

Results

From the analysis:

- all storage formats can improve on COO;
- if the $q_i \to 1$, Bit-IF is superior to MoSk-IF and Ind-IF
  - critical when blocking using Hilbert- or Z-curves;
- thesis contains similar analyses for data movement during TVM.

Actual compression on ten tensors from SPLATT:

Tensor	LBNL-Network	NIPS	Uber Pickups	$\cdots$	Delecious	NELL-1
COO	0.0443	0.0693	0.0740		3.1321	2.6748
Bit-IF	0.0693	0.0500	0.0519		2.6748	2.0816
Ratio	1.63	1.38	1.42		1.31	1.28

Table Storage requirement in Gbits, and the Bit-IF compression ratio

Aggregate compression rates vs. COO: 1.28–1.63×, geomean: 1.37×.

Results

Tensor–vector multiplication:

- **Bit-IF outperforms COO**
  - fastest in 10/10 cases on AMD EPYC 7720;
  - fastest in 7/10 cases on Intel 6238T.
    - $>$ 3 modes slower when lexicographic match and cacheable.

Results

Tensor–vector multiplication:

- **Bit-IF outperforms COO**
  - fastest in 10/10 cases on AMD EPYC 7720;
  - fastest in 7/10 cases on Intel 6238T.
    - $>$ 3 modes slower when lexicographic match and cacheable.
- **blocked Bit-IF** faster in 20 to 22 modes of TVM (40 total)
  - when the block size is chosen appropriately.

Results

Tensor–vector multiplication:

- **Bit-IF outperforms COO**
  - fastest in 10/10 cases on AMD EPYC 7720;
  - fastest in 7/10 cases on Intel 6238T.
    - $>$ 3 modes slower when lexicographic match and cacheable.
- **blocked Bit-IF** faster in 20 to 22 modes of TVM (40 total)
  - when the block size is chosen appropriately.
- blocked Bit-IF using **Hilbert order** outperforms Morton order.

Results

Tensor–vector multiplication:

- **Bit-IF outperforms COO**
  - fastest in 10/10 cases on AMD EPYC 7720;
  - fastest in 7/10 cases on Intel 6238T.
    - $>$ 3 modes slower when lexicographic match and cacheable.
- **blocked Bit-IF** faster in 20 to 22 modes of TVM (40 total)
  - when the block size is chosen appropriately.
- blocked Bit-IF using **Hilbert order** outperforms Morton order.

These results:

- include construction of output in matching storage;
- are pending additional performance optimisations

Results

Tensor–vector multiplication:

- **Bit-IF outperforms COO**
  - fastest in 10/10 cases on AMD EPYC 7720;
  - fastest in 7/10 cases on Intel 6238T.
    - $>$ 3 modes slower when lexicographic match and cacheable.
- **blocked Bit-IF** faster in 20 to 22 modes of TVM (40 total)
  - when the block size is chosen appropriately.
- blocked Bit-IF using **Hilbert order** outperforms Morton order.

These results:

- include construction of output in matching storage;
- are pending additional performance optimisations;
  - including for shared- and distributed-memory parallelisation

Results

Tensor–vector multiplication:

- **Bit-IF outperforms COO**
  - fastest in 10/10 cases on AMD EPYC 7720;
  - fastest in 7/10 cases on Intel 6238T.
    - $>$ 3 modes slower when lexicographic match and cacheable.
- **blocked Bit-IF** faster in 20 to 22 modes of TVM (40 total)
  - when the block size is chosen appropriately.
- blocked Bit-IF using **Hilbert order** outperforms Morton order.

These results:

- include construction of output in matching storage;
- are pending additional performance optimisations;
  - including for shared- and distributed-memory parallelisation
- pending comparison versus CSF and HiCOO.

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:

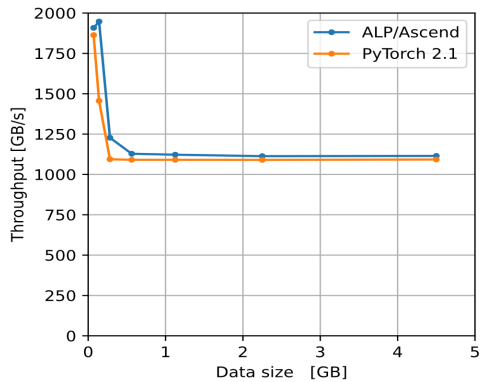
- **ALP/Ascend**, similar to OpenAI Triton (Tillet et al., '19 onwards):

```
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

Proposal

The ALPs:

- ALP/**{Graph,Sparse,Sp}BLAS**, ALP/{Solver, Dense, Ascend, ...}

Proposal

The ALPs:

- ALP/**{Graph,Sparse,Sp}BLAS**, ALP/{Solver, Dense, Ascend, ...}

Similar to GraphBLAS standard proposals,

- humble API for high-performance tensor computations?

Proposal

The ALPs:

- ALP/**{Graph,Sparse,Sp}BLAS**, ALP/{Solver, Dense, Ascend, ...}

Similar to GraphBLAS standard proposals,

- humble API for high-performance tensor computations?

For example, loosely based on internal ALP/Ascend representation:

```
Tensor< float , SPARSE > X( n1, ..., nk ), Y( n1, n2 ), Z( n3, .., nk );
Vector< float , DENSE > sum( n3 ); // an order-1 tensor
txt( Z["k", ..., "end"], X["i", "j", "k" , ..., "end"], Y["i", "j"], semiring );
foldl( sum[ 0 ], Z[ 0, 1, ..., n ], max ); // support eins *and* int axes?
```

Proposal

The ALPs:

- ALP/**{Graph,Sparse,Sp}BLAS**, ALP/{Solver, Dense, Ascend, ...}

Similar to GraphBLAS standard proposals,

- humble API for high-performance tensor computations?

For example, loosely based on internal ALP/Ascend representation:

```
Tensor< float , SPARSE > X( n1, ..., nk ), Y( n1, n2 ), Z( n3, .., nk );
Vector< float , DENSE > sum( n3 ); // an order-1 tensor
txt( Z["k", ..., "end"], X["i", "j", "k" , ..., "end"], Y["i", "j"], semiring );
foldl( sum[ 0 ], Z[ 0, 1, ..., n ], max ); // support eins *and* int axes?
```

- Auto-parallel? Sparse? Structures? Generalised LA? Interoperability?

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);
- Y.: Humble Heroes, Communications of Huawei Research (2023, to appear);
- 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).

Backup slides

Backup slides

TVM results

LBNL-Network, AMD EPYC 7720

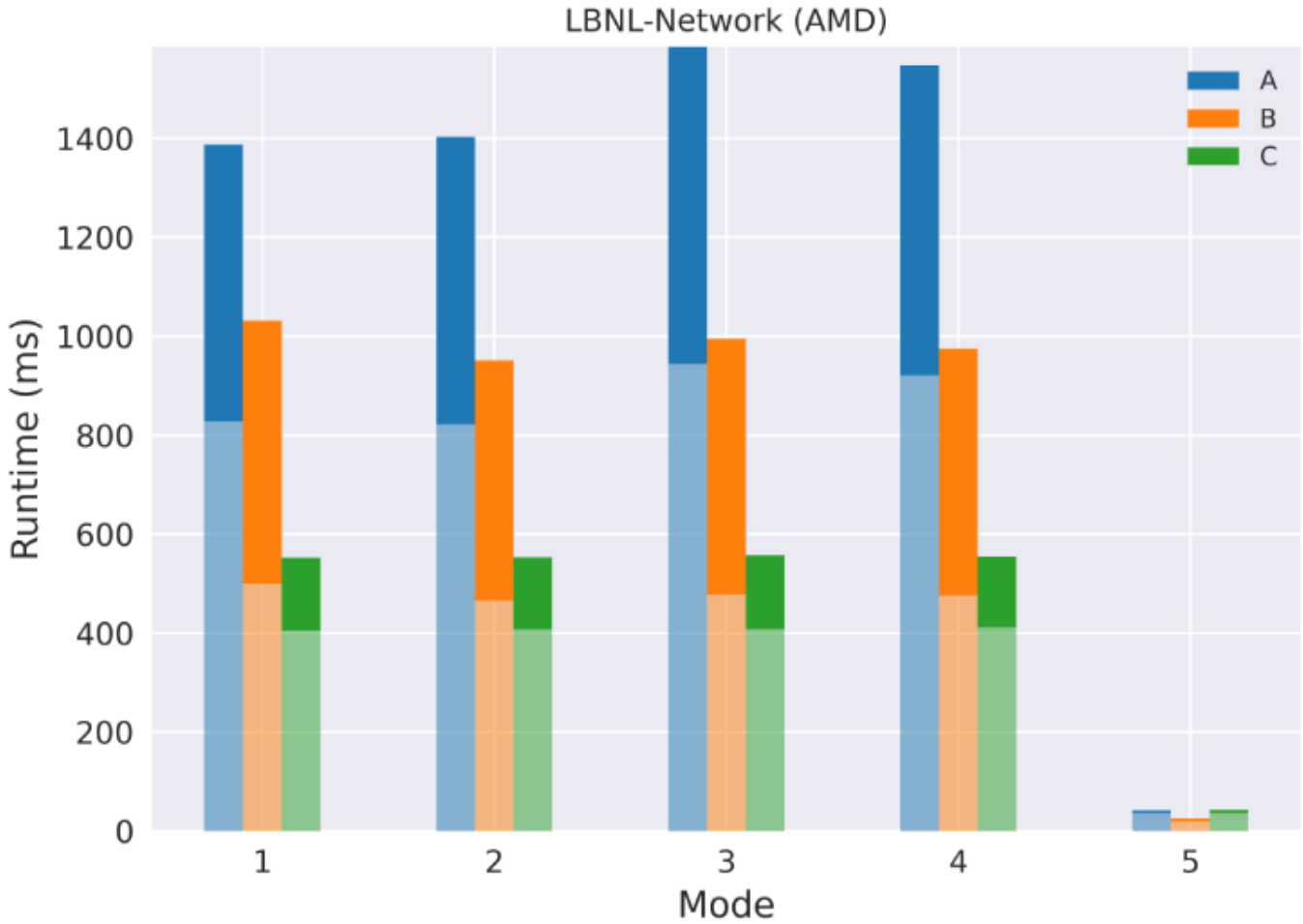


Fig. COO vs. Bit-IF vs. best blocked Bit-IF

TVM results

LBNL-Network, Intel 6238T

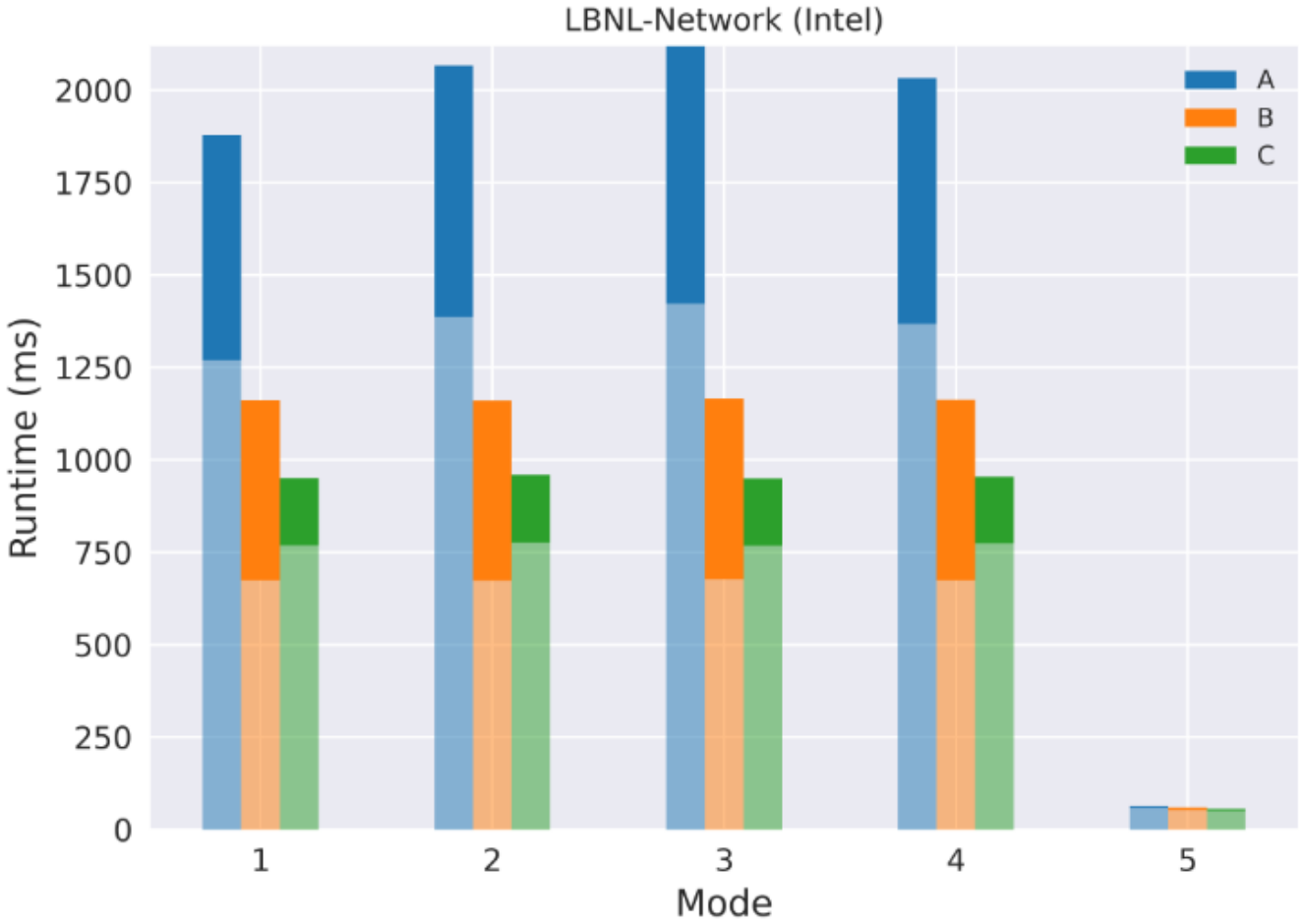


Fig. COO vs. Bit-IF vs. best blocked Bit-IF

TVM results

Enron, AMD EPYC 7720

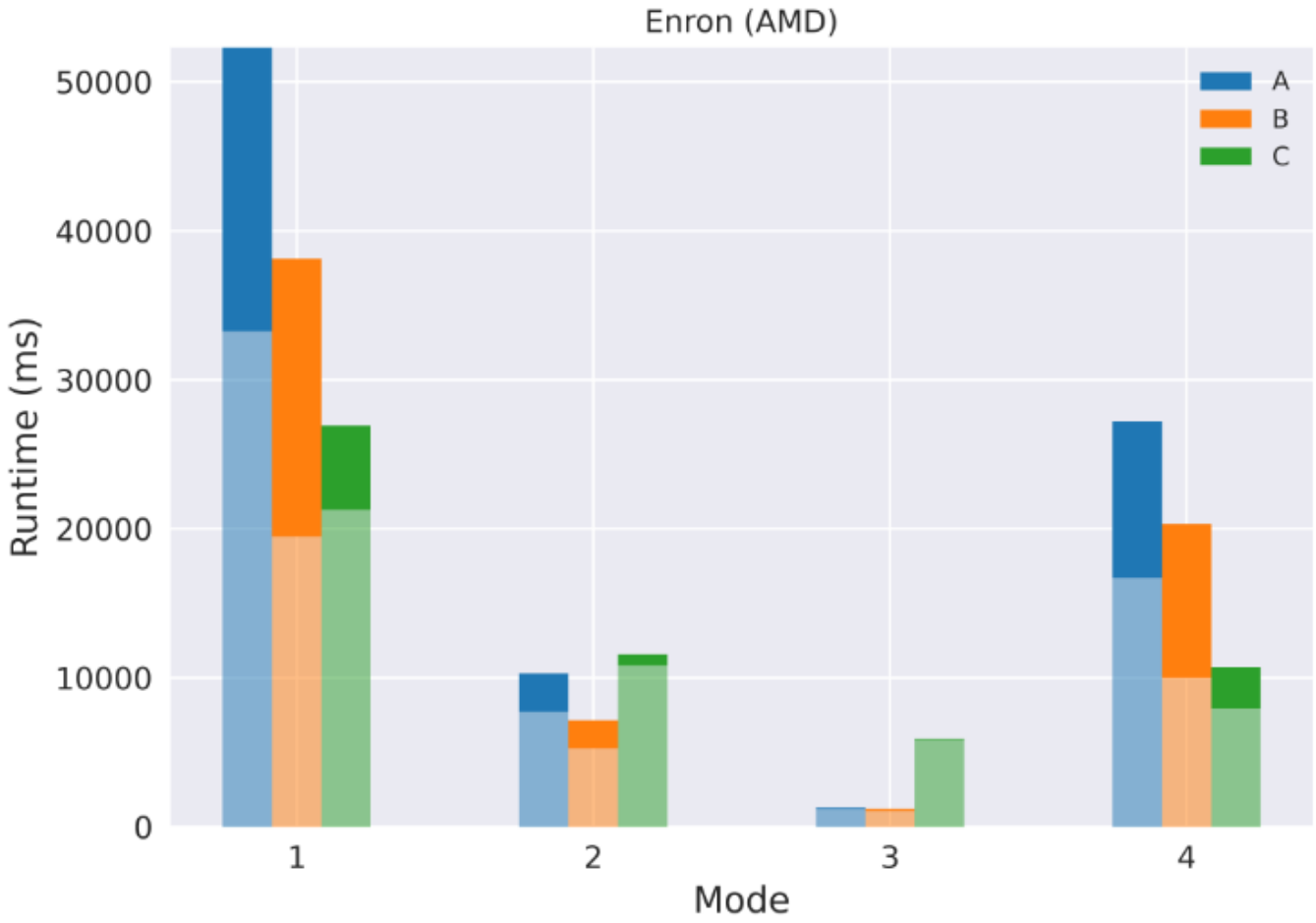


Fig. COO vs. Bit-IF vs. best blocked Bit-IF

TVM results

Enron, Intel 6238T

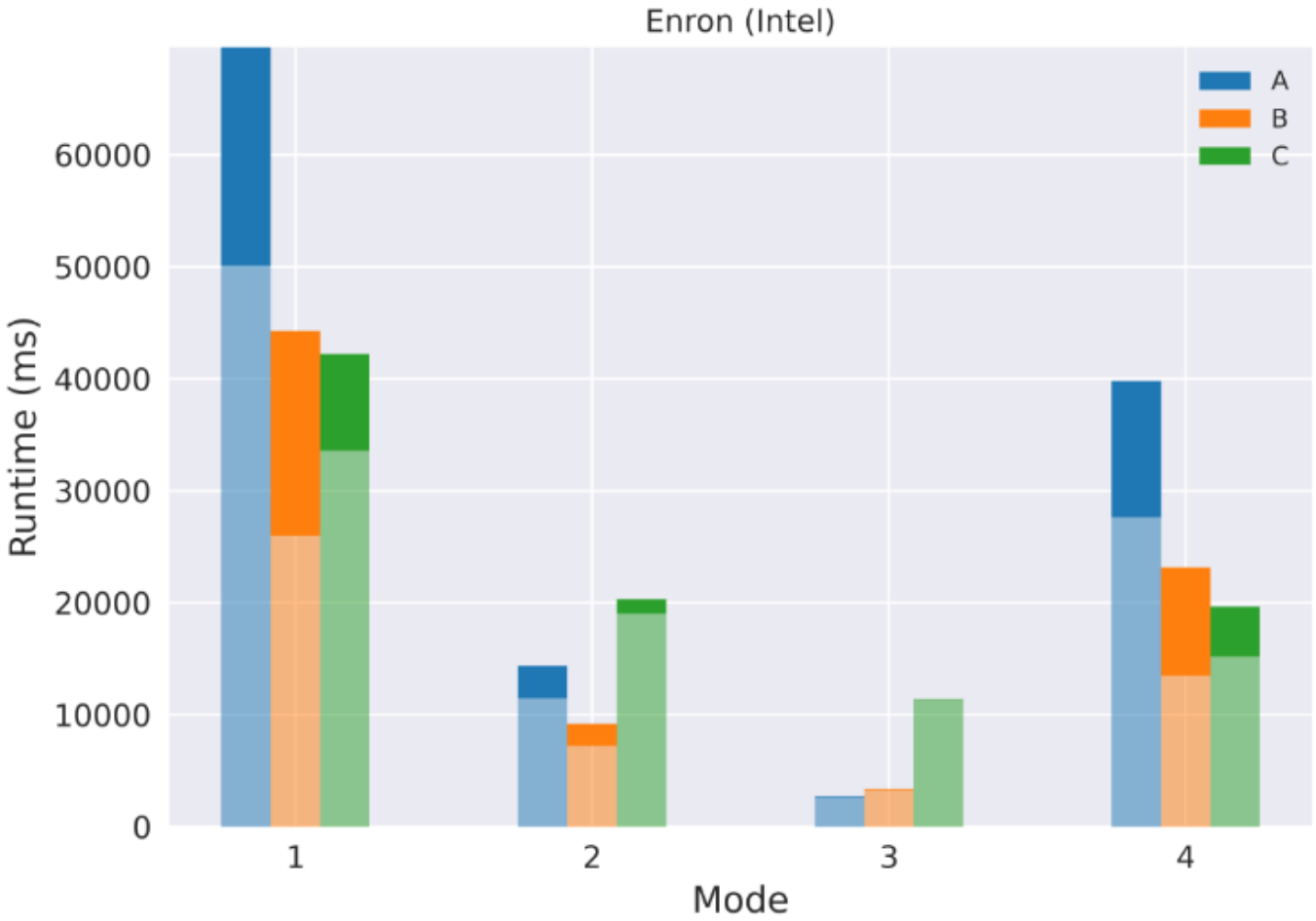


Fig. COO vs. Bit-IF vs. best blocked Bit-IF

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 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.

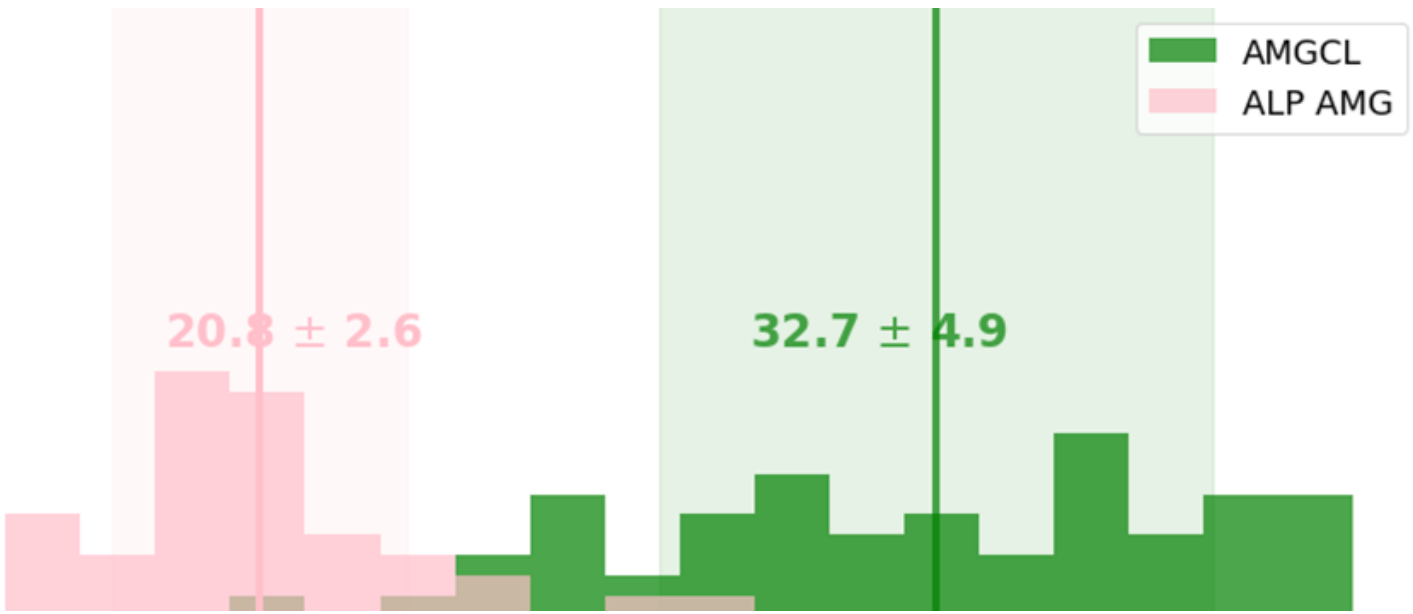
ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1
ALP	0.691	0.483	1.95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

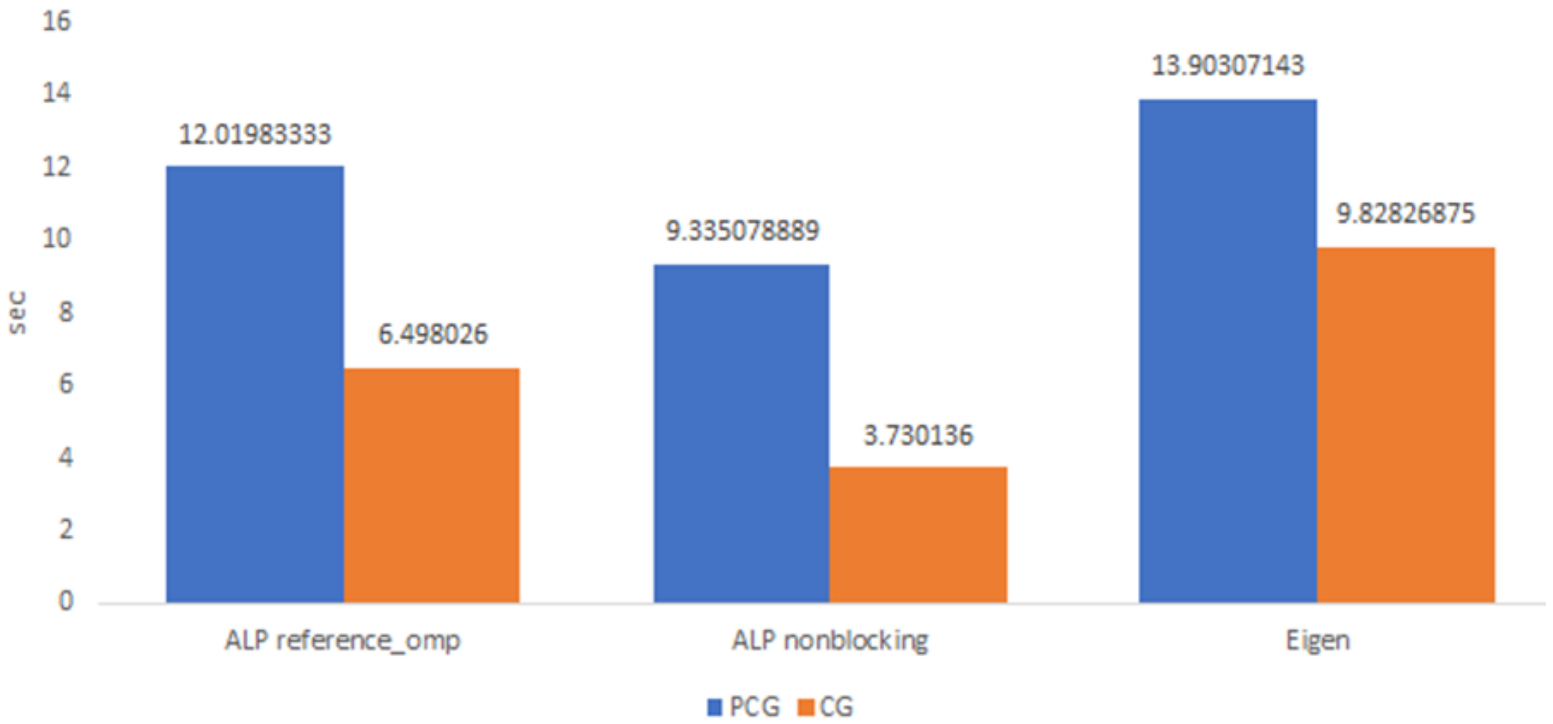
Tresholded ILU PCG in ALP vs. Eigen on the same internal problem:

- 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)

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
```

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; ...

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;

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`;
- Spark Clueweb: out of memory; Blogel (Ammar, Ozsu '18): 128 nodes

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`;
- Spark Clueweb: out of memory; Blogel (Ammar, Ozsu '18): 128 nodes
- results being refreshed for ARM and latest ALP, Scala, Spark, LPF
  - see ALP/Spark @ GitHub

Ref.: Suijlen and Y., "Lightweight Parallel Foundations", (2019; pre-v0.1 ALP; results are being refreshed).

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );          // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                        // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;          // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );             // t = u * L
foldl( t, gamma, add );                             // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                   // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );          // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                        // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;          // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );             // t = u * L
foldl( t, gamma, add );                             // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                   // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Easy to use: very close to MATLAB, Octave, Eigen, etc.

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );          // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                        // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;          // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );             // t = u * L
foldl( t, gamma, add );                             // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                   // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Easy to use: very close to MATLAB, Octave, Eigen, etc.

- also supports **mixed precision** and **complex** arithmetic:

```
typedef std::complex< double > T; // abstract value type
grb::Vector< T > x;
```

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );          // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                        // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;          // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );             // t = u * L
foldl( t, gamma, add );                             // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                   // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Easy to use: very close to MATLAB, Octave, Eigen, etc.

- also supports **mixed precision** and **complex** arithmetic:

```
typedef std::complex< double > T; // abstract value type
grb::Vector< T > x;
Semiring<
    operators::add< T >, operators::mul< T >,
    identites::zero, identities::one
> plusTimes;
```

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );          // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                        // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;          // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );             // t = u * L
foldl( t, gamma, add );                             // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                   // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Easy to use: very close to MATLAB, Octave, Eigen, etc.

- also supports **mixed precision** and **complex** arithmetic:

```
typedef std::complex< double > T; // abstract value type
grb::Vector< T > x;
Semiring<
    operators::add< T >, operators::mul< T >,
    identites::zero, identities::one
> plusTimes;
dot( beta, x, y, add, operators::conj_right_mul< T >() );
```