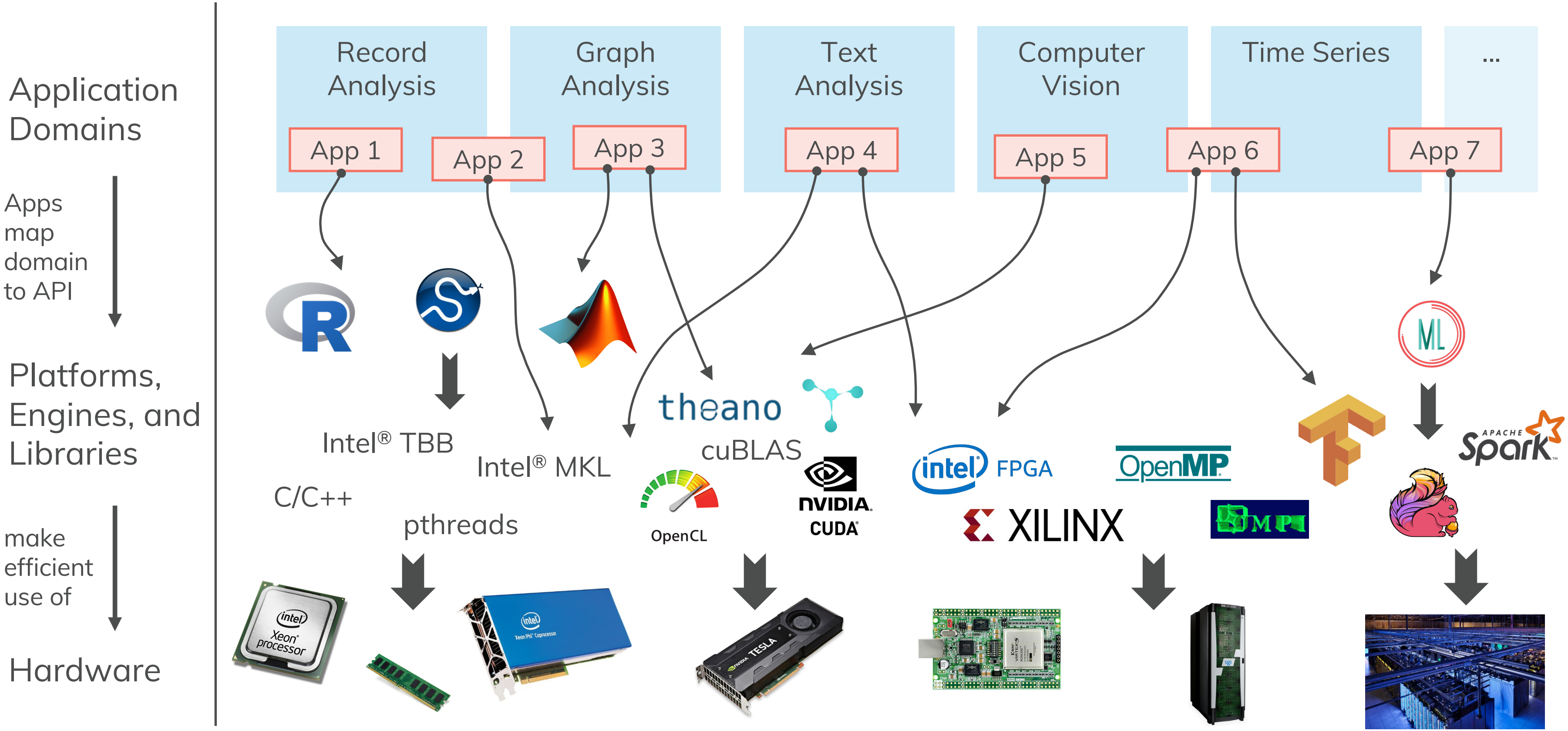
TECHNISCHE UNIVERSITÄT DRESDEN

Dresden Database Systems Group

AL: Unified analytics in domain specific terms

Johannes Luong, Dirk Habich, Wolfgang Lehner
Chair of Databases, Techinsche Universität Dresden

Analytical landscape



Analytical landscape

Dresden Database Systems Group

Hardwired execution path

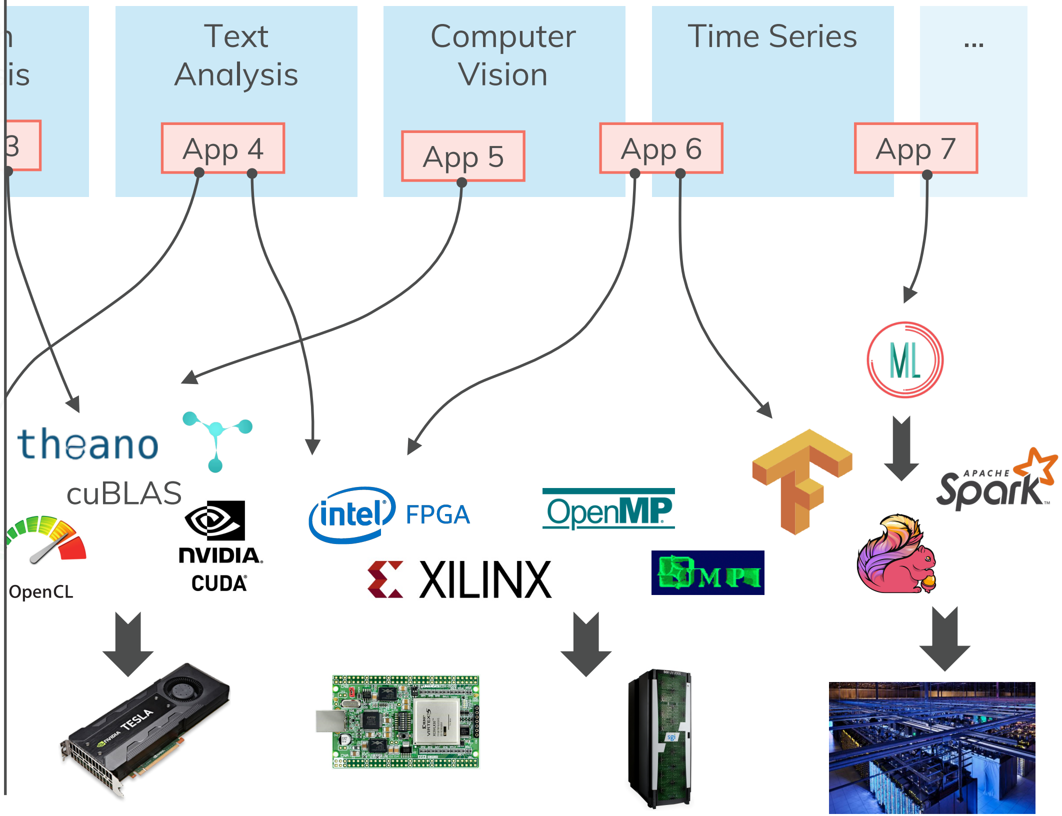
- Replacing an API requires rewrite
- No path to improved solutions

User guided translation

- *Use of generic or low-level APIs*
- Obscured domain logic
- Missed or misguided optimization

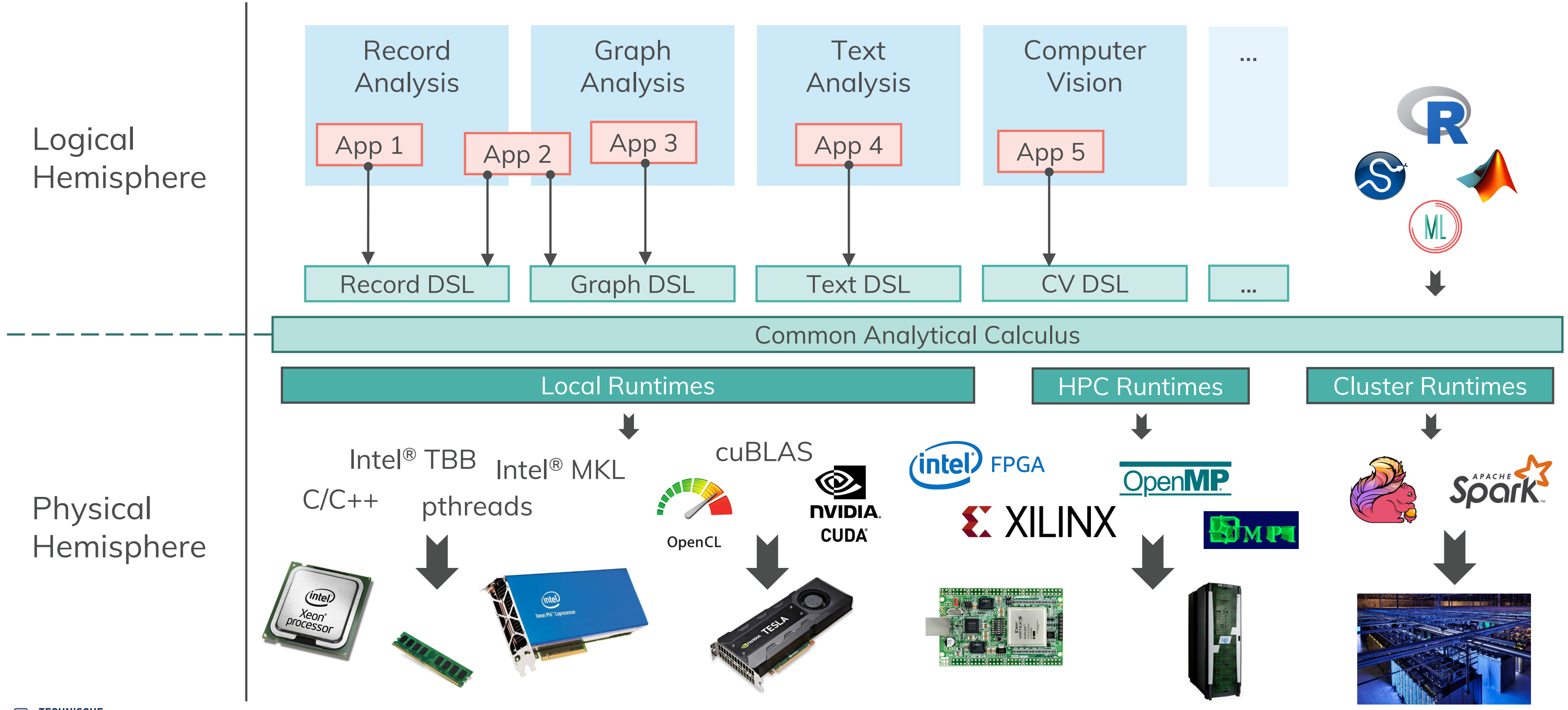
Resctricted domain composition

- *Use of domain specific engines*
- Costly cross engine data movement
- No cross domain optimization



An Analytical Abstraction

Dresden Database Systems Group



An Analytical Abstraction

Dresden Database Systems Group

Strong physical abstraction

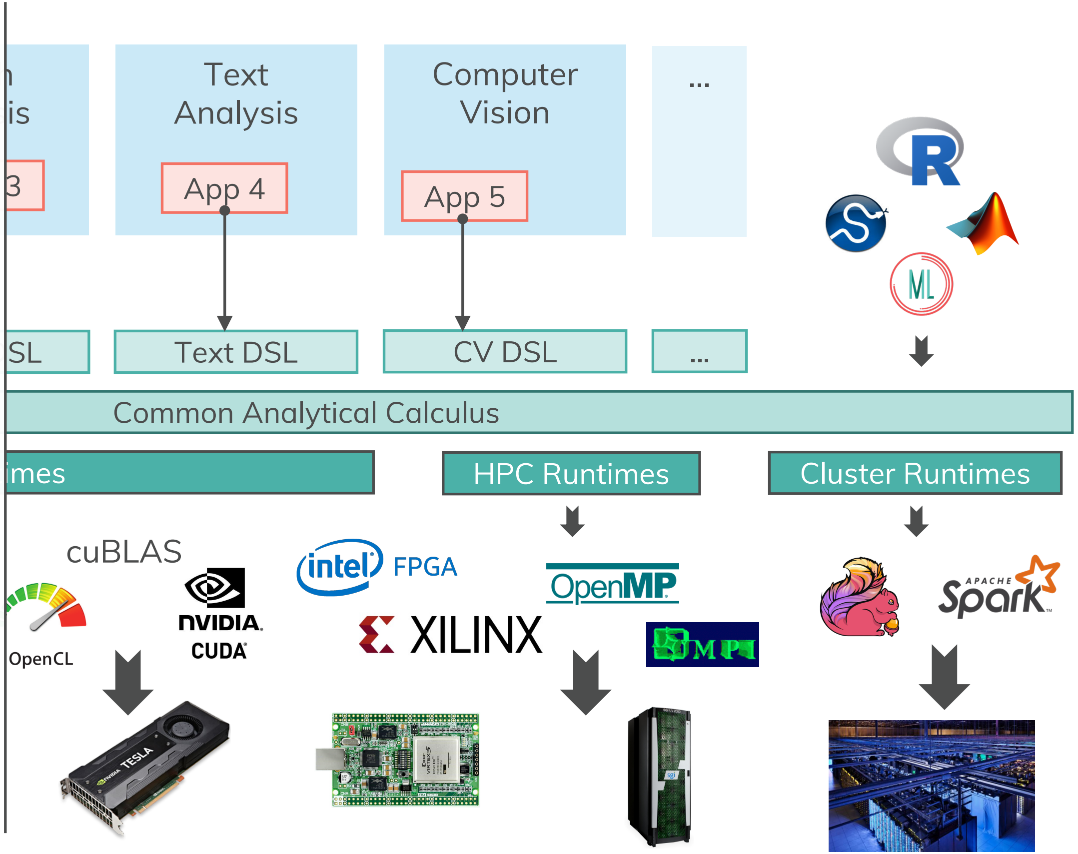
- Implementation independent apps
  - *portable performance*
- Large space of physical strategies

Domain specific applications

- Apps written in domain logic
- Engine guided compilation
  - Calculus generalizes DSL optimization

Flexible domain composition

- In the calculus, all domains are equal



Let‘s get real – AL & AIR

AL – Analytical Language

Dresden Database Systems Group

A container for domain languages

- Small „*functional*“ core
  - expressions, let bindings
  - no side effects
- A set of embedded DSLs
  - Record DSL, Graph DSL

Implemented as scala library

- Use Scala‘s parser as frontend
- Simple integration with Scala apps

Designed for extensibility

- Inherit some traits, create a new language

```
def getOfferings(userId: Long): ResultHandle =
  AL""""
    currentBill =
      SELECT(sum(price))
        .FROM(purchases_m)
        .WHERE(purchases_m.user_id == $userId)

    relatedProducts =
      MATCH(
        u[User](id == $userId) -visited-> p[Product],
        u2[User] -bought-> p,
        u2 -bought-> p2[Product]
      ).WHERE(
        abs(u.avg_mbill - u2.avg_mbill) < 100.0,
        p2.price < (u.avg_month_bill - currentBill)
      ).RETURN(
        p2.id as p_id
        p2.url as p_url)

      return relatedProducts
  """"
```

Record DSL

Graph DSL

The analytical calculus

Dresden Database Systems Group

Desired properties

- Independent of execution model
  - No opinion on data access / compute
  - Naturally parallel
- Expressive and flexible
  - capture complete algorithms
- Straightforward transformation
  - domain optimizations (join order et al.)
  - translation to instructions/operators

Restricted form lambda calculus

- Abstract, expressive, provable transformations
- Monad comprehensions for collection access
- Set of well known recursion patterns
  - Structural recursion on collections
  - *Tail call for iterative algorithms*

```
MATCH(
 u[User](id == $userId) -visited-> p[Product],
 u2[User] -bought-> p,
 u2 -bought-> p2[Product]
).RETURN(
 p2.id as p_id, p2.url as p_url
)
```

⬇

```
Set(
 Record(p_id: b2.id, p_url: b2.url) |
  b0 <- edges,
   b0.from.type = User, b0.from.id = $userId
   b0.property = visited
   b0.to.type = Product,
  b1 <- edges,
   b1.from.type = User,
   b1.property = bought
   b1.to = b0.to
  b2 <- edges,
   b2.from = b1.from,
   b2.property = bought
   b2.to.type = Product
)
```

AIR – Analytical Intermediate Representation

Dresden Database Systems Group

Form: tree of functions

- Let bindings
- Function calls
- Control structures

Semantics: library of functions

- Comprehension constructors
- Recursion constructors
- Future extensions

Creation: air builder

- Scala AIR builder library
- Reusable with other languages
- Create and run AIR directly

```
SELECT(price).FROM(purchases_m).WHERE(userId == 123)
```

```
Func({
  b0: Bag[Record[A]] = DataObject(„purchases_m“)

  b1: Record[A] => Double = Func(p0: Record[A], {
    b2: Double = RecordGet(p0, „price“)
    return b2
  })

  b3: Record[A] => Bool = Func(p0: Record[A], {
    b4: Long = RecordGet(p0, „userId“)
    b5: Long = IntLiteral(123) // $userID
    b6: Bool = Equals(b4, b5)
    return b6
  })

  b7: Bag[Double] = BagComprehension(b1, Seq(b0, b3))

  return b7
})
```

From RecordDSL to AIR

Transformation pipeline

1. Parse the AL program into an AST
   - Using the Scala Parser
2. Map the AST to AIR

AST matching and AIR builder

- Map AST node to AIR builder calls
- AST pattern matching with quasi quotes

Extending AL

- Add AstTraversial traits
- Compose traversal traits
- Delegate to super

```
trait RecordDSL extends AstTraversal {
 override def traverseTree(ast: Tree, ir: FuncBuilder): IRBuilder =
  ast match {
   case q"SELECT(..$expTs).FROM(..$tableTs)" =>
    // translate tableTs and expTs ...
    val comp = ir.addBagComp()(RecordType(...))
    comp.addBindings(tables)
    // create head and return comprehension identifier

   case q"$qry.WHERE(..$predicateTs)" =>
    // translate predicates ...
    val comp = getBuilder[BagComp](qry)
    // add filter to comprehension, return ref

   case _ =>
    super.traverseTree(ast, ir)
  }
}

trait GraphDSL extends AstTraversal { ... }

object AL extends AstTraversal with RecordDSL with GraphDSL
```

Let‘s get going– AIR runtime(s)

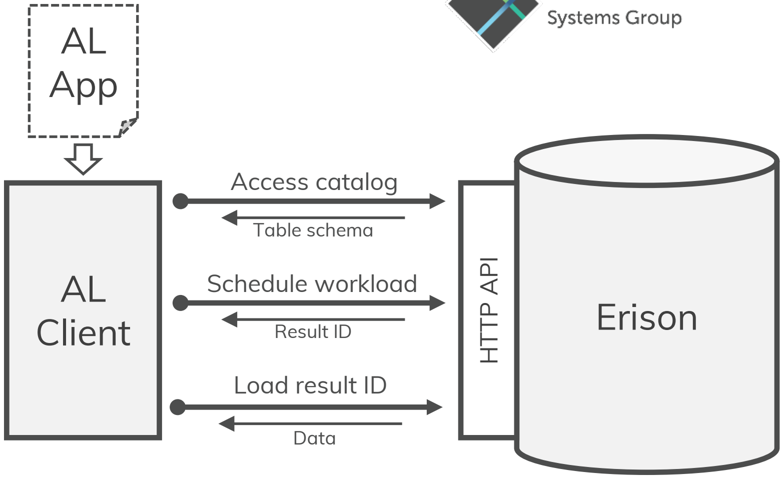
Scale up performance

Erison – shared memory processing

- In-memory data store
  - column oriented record format
- Catalog with schema information
- Work stealing task scheduler (Intel® TBB)
- HTTP interface

Task based parallelism

- Task scheduler
  - Spawns and manages worker threads
  - Accepts self-contained processing tasks
  - Assigns tasks to workers
- Deals with a large number of tasks
- Supports nesting and composition
  - No static #task to #thread ratio!



PFor(Range [0, 100) , lamda)

schedule(Task(Range [0, 25) , lamda))
schedule(Task(Range [25, 50) , lamda))
schedule(Task(Range [50, 75) , lamda))
schedule(Task(Range [75, 100) , lamda))

AIR on Erison

Execution pipeline

- Parse AIR from json serialization
- Build internal loop nest representation
- Optimize loop nest representation
  - join order, hash join, push down predicates
- Compile and execute loop nest program

Loop nest representation

- AIR with comprehensions replaced by loop nests

Loop nest execution

- JIT compile each loopnest into a lambda
- Schedule a PFor for each loop
  - Use the compiled lambda as PFor body

Bag(p.price | p <- purchases_m, p.userId = 123)

```
Loop(li : purchases_m)
    Guard(li.userId == 123)
    Emit(li.price)
```

```
PFor(0, size(purchases_m))
    JitLoop(li : purchases_m[from, to])
        Guard(li.userId == 123)
        Emit(li.price)
```

Wrap Up

Wrap Up

Dresden Database
Systems Group

An analytical abstraction

- Separate algorithms from implementation
- Portable logic
- Managed processing

Many languages, many engines

- AIR can be targetted by many languages
- AIR to Spark/Flink dataflow graph

Future work

- Extend the calculus
  - Tensor type?
- Optimization
  - Must optimization be done in the engine?
  - Cross domain optimization?

Thank You!