7 June 2022

# RISE & Shine
## Language-Oriented Compiler Design

**Michel Steuwer, Thomas Kœhler, Bastian Köpcke, Federico Pizzuti**
with contributions from **Martin Lücke, Johannes Lenfers, Rongxiao Fu, Xueying Qin**

# Compiler Design With MLIR

## Representation-Oriented Compiler Design

- MLIR focuses on **Representation** (i.e. Syntax)
  - *Operations* have *operands*, *attributes*, and a *type*
  - *Operations* are organised in SSA form in *Blocks*
  - *Blocks* form CFGs and are part of *Regions (which can be nested in Operations)*
- Consistency is ensured by implementing and calling `verify` methods on operations, blocks, and regions …
- **Semantics** is given informally by how representations are transformed in each other

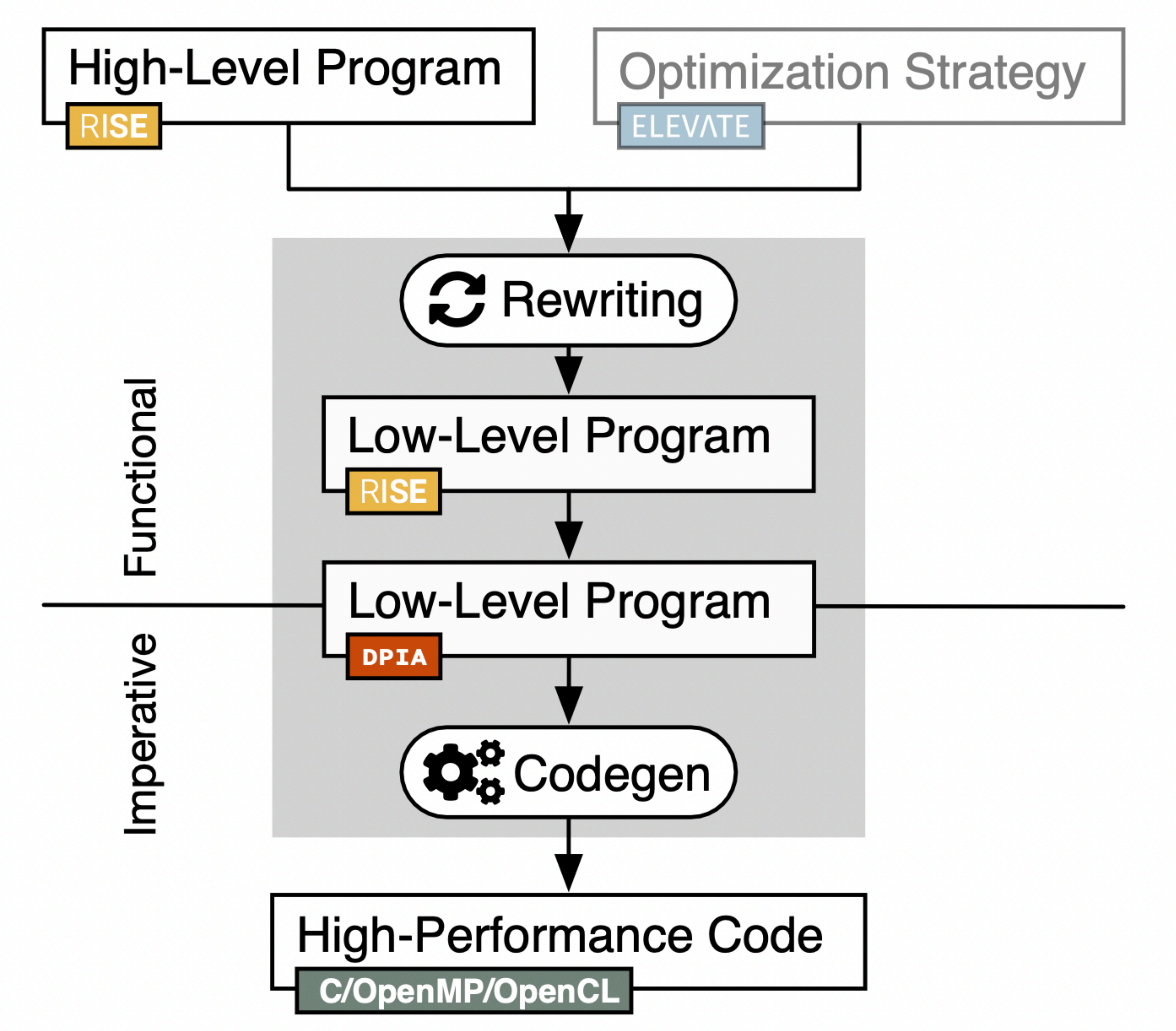
**How should we design *good* IRs that:**
***1)*** **have a clear purpose and semantics;**
***2)*** **have formally checked invariants and assumptions;**
***3)*** **are easily extensible?**

# Shine Compiler

## Language-Oriented Compiler Design

We advocate for:

- Clear separation of concerns between *optimizing* and *code generation*
- Formalisation of invariants and assumptions about IRs in *type systems*
- *Extensibility* at each level in the compiler



# GEMM in RISE

### High-Level GEMM

```
depFun((m:Nat,n:Nat,k:Nat) =>
 fun((A: Array[m,Array[k,f32]], B: Array[k,Array[n,f32]],
      C: Array[m,Array[n,f32]], alpha: f32, beta: f32) =>
 zip(A)(C) |> map(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |> map(fun(colBC =>
   zip(fst(rowAC))(fst(colBC)) |>
   map(fun((a, b) => a * b)) |> reduce(+, 0) |>
   fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

Optimization Strategy

ELEVATE

Rewriting

### Low-Level GEMM

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
 zip(A)(C) |> mapBlock(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |>
  mapThreads(fun(colBC => zip(fst(rowAC))(fst(colBC)) |>
     reduceSeq(Local)(fun((acc,ab) =>
        acc + fst(ab) * snd(ab)),0) |>
     fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

Translation

### Imperative GEMM

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
 parForBlock(m,Array[n,f16], output, fun(rowIdx,outRow =>
  parForThreads(n,f16, outRow, fun(colIdx,outElem =>
   new(Local,f32, fun((accumExp, accumAcc) =>
    accumAcc = 0.0f;
    for(k, fun(i => accumAcc = accumExp +
     fst(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
      fst(idx(colIdx, zip(transpose(B),
       snd(idx(rowIdx, zip(A,C))))))))) *
     snd(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
      fst(idx(colIdx, zip(transpose(B),
       snd(idx(rowIdx, zip(A,C))))))))));
    outElem = alpha * accumExp + beta *
     snd(idx(colIdx, zip(transpose(B),
      snd(idx(rowIdx, zip(A,C)))))) ))));
  syncThreads())))
```

Codegen

```
__global__ void gemm_kernel(float* __restrict__ output,
 int m, int n, int k, const __half* __restrict__ A,
 const __half* __restrict__ B,
 const float* __restrict__ C, float alpha, float beta) {
  for(int rowIdx=blockIdx.x;
      blockIdx.x<m; rowIdx += gridDim.x) {
   for(int colIdx=threadIdx.x;
       threadIdx.x<n; rowIdx += blockDim.x) {
    float accum = 0;
    for (int i = 0; i < k; i++) {
     accum = accum + A[i + rowIdx*k] * B[colIdx + i*n];
    }
    output[colIdx + rowIdx * n] =
     alpha * accum + beta * C[colIdx + rowIdx*n];
   }
   __syncthreads(); }}
```

# GEMM in RISE

## RISE

### High-Level GEMM

```
depFun((m:Nat,n:Nat,k:Nat) =>
 fun((A: Array[m,Array[k,f32]], B: Array[k,Array[n,f32]],
      C: Array[m,Array[n,f32]], alpha: f32, beta: f32) =>
 zip(A)(C) |> map(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |> map(fun(colBC =>
   zip(fst(rowAC))(fst(colBC)) |>
   map(fun((a, b) => a * b)) |> reduce(+, 0) |>
   fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

Optimization Strategy ELEVATE

Rewriting

### Low-Level GEMM

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
 zip(A)(C) |> mapBlock(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |>
  mapThreads(fun(colBC => zip(fst(rowAC))(fst(colBC)) |>
    reduceSeq(Local)(fun((acc,ab) =>
       acc + fst(ab) * snd(ab)),0) |>
    fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

Translation

## DPIA

### Imperative GEMM

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
 parForBlock(m,Array[n,f16], output, fun(rowIdx,outRow =>
  parForThreads(n,f16, outRow, fun(colIdx,outElem =>
   new(Local,f32, fun((accumExp, accumAcc) =>
    accumAcc = 0.0f;
    for(k, fun(i => accumAcc = accumExp +
     fst(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
      fst(idx(colIdx, zip(transpose(B),
       snd(idx(rowIdx, zip(A,C))))))))) *
     snd(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
      fst(idx(colIdx, zip(transpose(B),
       snd(idx(rowIdx, zip(A,C)))))))))));
    outElem = alpha * accumExp + beta *
     snd(idx(colIdx, zip(transpose(B),
      snd(idx(rowIdx, zip(A,C)))))) ))));
  syncThreads()))))
```

Codegen

## C

```
__global__ void gemm_kernel(float* __restrict__ output,
 int m, int n, int k, const __half* __restrict__ A,
 const __half* __restrict__ B,
 const float* __restrict__ C, float alpha, float beta) {
  for(int rowIdx=blockIdx.x;
      blockIdx.x<m; rowIdx += gridDim.x) {
   for(int colIdx=threadIdx.x;
       threadIdx.x<n; rowIdx += blockDim.x) {
    float accum = 0;
    for (int i = 0; i < k; i++) {
     accum = accum + A[i + rowIdx*k] * B[colIdx + i*n];
    }
    output[colIdx + rowIdx * n] =
     alpha * accum + beta * C[colIdx + rowIdx*n];
   }
   __syncthreads(); }}
```

# GEMM in RISE

## RISE

### High-Level GEMM

```
depFun((m:Nat,n:Nat,k:Nat) =>
 fun((A: Array[m,Array[k,f32]], B: Array[k,Array[n,f32]],
      C: Array[m,Array[n,f32]], alpha: f32, beta: f32) =>
 zip(A)(C) |> map(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |> map(fun(colBC =>
   zip(fst(rowAC))(fst(colBC)) |>
   map(fun((a, b) => a * b)) |> reduce(+, 0) |>
   fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

Optimization Strategy

ELEVATE

Rewriting Optimization

### Low-Level GEMM

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
 zip(A)(C) |> mapBlock(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |>
  mapThreads(fun(colBC => zip(fst(rowAC))(fst(colBC)) |>
    reduceSeq(Local)(fun((acc,ab) =>
       acc + fst(ab) * snd(ab)),0) |>
    fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

Translation Translation

## DPIA

### Imperative GEMM

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta
 parForBlock(m,Array[n,f16], output, fun(rowIdx,outRow =>
  parForThreads(n,f16, outRow, fun(colIdx,outElem =>
   new(Local,f32, fun((accumExp, accumAcc) =>
    accumAcc = 0.0f;
    for(k, fun(i => accumAcc = accumExp +
     fst(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
      fst(idx(colIdx, zip(transpose(B),
       snd(idx(rowIdx, zip(A,C))))))))) *
     snd(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
      fst(idx(colIdx, zip(transpose(B),
       snd(idx(rowIdx, zip(A,C)))))))))));
    outElem = alpha * accumExp + beta *
     snd(idx(colIdx, zip(transpose(B),
      snd(idx(rowIdx, zip(A,C)))))) ))));
  syncThreads())))
```

Codegen Translation

## C

```
__global__ void gemm_kernel(float* __restrict__ output,
 int m, int n, int k, const __half* __restrict__ A,
 const __half* __restrict__ B,
 const float* __restrict__ C, float alpha, float beta) {
  for(int rowIdx=blockIdx.x;
      blockIdx.x<m; rowIdx += gridDim.x) {
   for(int colIdx=threadIdx.x;
       threadIdx.x<n; rowIdx += blockDim.x) {
    float accum = 0;
    for (int i = 0; i < k; i++) {
     accum = accum + A[i + rowIdx*k] * B[colIdx + i*n];
    }
    output[colIdx + rowIdx * n] =
     alpha * accum + beta * C[colIdx + rowIdx*n];
   }
   __syncthreads(); }}
```

# RISE: a Purely Functional Language for Optimizing via Rewriting

```
E := x | 0.0f |                          variables and literals
     fun(x => E) |                       function abstraction
     E |> E | E(E) |                     function application
     depFun(x: K => E) |                 dependent fun. abstraction
     E(N) | E(DT) |                      dependent fun. application
     map | reduce | zip | …              primitives

T := t | DT |                            type variables & data types
     T -> T | (x: K) -> T                function types

DT:= f32 | … |                           scalar types
     Array[N,DT] | Tuple[DT,DT]          array & tuple types

N := 0|1| … | N + N | N * N | …          natural numbers

K := Nat | DataType | AddrSp             Kinds
```

Type system enforces that no functions can be stored in memory

# RISE: a Purely Functional Language for Optimizing via Rewriting

```
map: {n: Nat} -> {s: DataType} -> {t: DataType} ->
     (s -> t) -> Array[n, s] -> Array[n, t]

reduce: {n: Nat} -> {t: DataType} ->
     (t -> t -> t) -> t -> Array[n, t] -> t

zip: {n: Nat} -> {s: DataType} -> {t: DataType} ->
     Array[n, s] -> Array[n, t] -> Array[n, Tuple[s, t]]
```

---

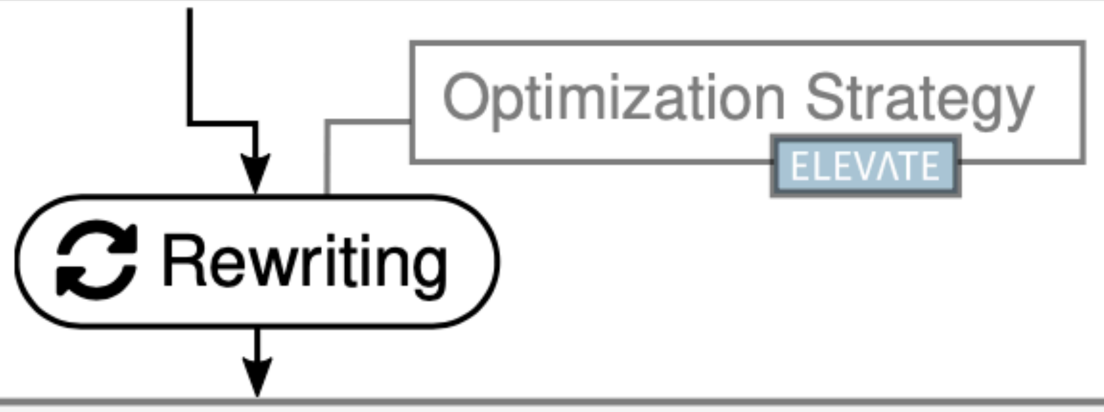
```
mapSeq: {n: Nat} -> {s: DataType} -> {t: DataType} ->
     (s -> t) -> Array[n, s] -> Array[n, t]
mapPar: {n: Nat} -> {s: DataType} -> {t: DataType} ->
     (s -> t) -> Array[n, s] -> Array[n, t]

reduceSeq: {n: Nat} -> {s: DataType} -> {t: DataType} ->
     (t -> s -> t) -> t -> Array[n, s] -> t
```

# Optimizing via Rewriting

**High-Level GEMM**

```
depFun((m:Nat,n:Nat,k:Nat) =>
 fun((A: Array[m,Array[k,f32]], B: Array[k,Array[n,f32]],
      C: Array[m,Array[n,f32]], alpha: f32, beta: f32) =>
 zip(A)(C) |> map(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |> map(fun(colBC =>
   zip(fst(rowAC))(fst(colBC)) |>
   map(fun((a, b) => a * b)) |> reduce(+, 0) |>
   fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```




Discussed 4 weeks ago

**Low-Level GEMM**

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
 zip(A)(C) |> mapBlock(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |>
  mapThreads(fun(colBC => zip(fst(rowAC))(fst(colBC)) |>
     reduceSeq(Local)(fun((acc,ab) =>
        acc + fst(ab) * snd(ab)),0) |>
     fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

# Correctness Proof of Rewrite Rule

```
mapSplit : (n: ℕ) → {m: ℕ} → {s t: Set} → (f: s → t) → (xs: Vec s (m * n)) →
           map (map f) (split n {m} xs) ≡ split n {m} (map f xs)
simplification : (n: ℕ) → {m: ℕ} → {t: Set} → (xs: Vec t (m*n)) → (join ∘ split n {m}) xs ≡ xs
{- Split-join rule proof -}
splitJoin : {m: ℕ} → {s: Set} → {t: Set} → (n: ℕ) → (f: s → t) → (xs: Vec s (m * n)) →
            (join ∘ map (map f) ∘ split n {m}) xs ≡ map f xs
splitJoin {m} n f xs =
    begin
      (join ∘ map (map f) ∘ split n {m}) xs
    ≡⟨⟩
      join (map (map f) (split n {m} xs))
    ≡⟨ cong join (mapSplit n {m} f xs) ⟩
      join (split n {m} (map f xs))
    ≡⟨ simplification n {m} (map f xs) ⟩
      map f xs
    ∎
```

Listing 3. Proof of correctness of the splitJoin rewrite rule in Agda

# DPIA: Combining Functional and Imperative

```
P := x | 0.0f |                          variables and literals
     fun(x => P) |                       function abstraction
     P |> P | P(P) |                     function application
     depFun(x: K’ => E) |                dependent fun. abstraction
     P(N) | P(DT) |                      dependent fun. application
     mapPar | reduceSeq | zip | …        functional primitives
     P = P | ; | new | parFor | …        imperative primitives

T’:= t | T’ -> T’ | (x: K’) -> T’        type var. & function types
     T’ x T’ |                           phrase pair type
     Exp[DT, RW] |                       expression type
     Acc[DT] |                           acceptor type
     Comm                                command type

RW  := Rd | Wr                           read-write annotations
K’  := K | RW                            kinds
```

Type system separates functional and imperative parts

# DPIA: Combining Functional and Imperative

```
mapPar(n: Nat, s: DataType, t: DataType,
       f: Exp[s, Rd] -> Exp[t, Wr],
       in: Exp[Array[n, s], Rd]): Exp[Array[n, t], Wr]

reduceSeq(n: Nat, s: DataType, t: DataType,
          f: Exp[t, Rd] -> Exp[s, Rd] -> Exp[t, Wr],
          init: Exp[t, Wr],
          in: Exp[Array[n, s], Rd]): Exp[t, Rd]
```

---

```
assign(t: DataType, lhs: Acc[t], rhs: Exp[t, Rd]): Comm

seq(c1: Comm, c2: Comm): Comm

new(t: DataType, body: (Exp[t, Rd] x Acc[t]) -> Comm): Comm

for(n: Nat, body: Exp[Idx[n], Rd] -> Comm): Comm

parFor(n: Nat, t: DataType, out: Acc[Array[n,t]],
       body: Exp[Idx[n], Rd] -> Acc[t] -> Comm): Comm
```

# Translating From RISE to DPIA

- Translation via two mutual recursive functions
  - **Acceptor translation:**
    translate an *expression E* into a command by writing the translated result into *acceptor A*:
    $$accT(E, A) \simeq A = E$$
  - **Continuation translation:**
    translate an *expression E* into a command by passing the translated result to a *continuation C*:
    $$conT(E, C) \simeq C(E)$$

Rewriting

Low-Level GEMM

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
 zip(A)(C) |> mapBlock(fun(rowAC =>
  zip(B |> transpose)(snd(rowAC)) |>
  mapThreads(fun(colBC => zip(fst(rowAC))(fst(colBC)) |>
     reduceSeq(Local)(fun((acc,ab) =>
        acc + fst(ab) * snd(ab)),0) |>
     fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

Translation

Imperative GEMM

```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
 parForBlock(m,Array[n,f16], output, fun(rowIdx,outRow =>
  parForThreads(n,f16, outRow, fun(colIdx,outElem =>
   new(Local,f32, fun((accumExp, accumAcc) =>
    accumAcc = 0.0f;
    for(k, fun(i => accumAcc = accumExp +
     fst(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
      fst(idx(colIdx, zip(transpose(B),
       snd(idx(rowIdx, zip(A,C))))))))) *
     snd(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
      fst(idx(colIdx, zip(transpose(B),
       snd(idx(rowIdx, zip(A,C))))))))));
    outElem = alpha * accumExp + beta *
     snd(idx(colIdx, zip(transpose(B),
      snd(idx(rowIdx, zip(A,C)))))) ))));
  syncThreads())))))
```

# Translating From RISE to DPIA

```
def accT(E: Phrase[Exp[t,Wr]],
         A: Phrase[Acc[t]]): Phrase[Comm] = expr match {
  case mapSeq(n, s, t, f, in) =>
      conT(in, fun(inT =>
        for(n, fun(i =>
          accT( f(inT[i]), A[i] ) ))))
  case mapPar(n, s, t, f, in) =>
      conT(in, fun(inT =>
        parFor(n, t, A, fun((i, a) =>
          accT( f(inT[i]), a ) ))))
  case …
}
```

# Translating From RISE to DPIA

```
def conT(E: Phrase[Exp[t,Rd]],
         C: Phrase[Expr[t,Rd]] -> Phrase[Comm]): Phrase[Comm] = expr match {
  case reduceSeq(n, t, s, f, init, in) =>
     conT(in, fun(inT =>
       new(t, fun((accumAcc, accumExp) =>
         accT(init, accumAcc) ;
         for(n, fun(i =>
           accT( f(accumExpr, inT[i]), accumAcc) )) ;
         conT(accumExp, C) )) ))
  case …
}
```

# Systematically Extending Shine With Support for Tensor Cores

**Bottom-up approach:**

1. Add new *low-level* **imperative** *primitives* corresponding to the CUDA Tensor Core API and implement Codegen for these primitives.

2. Add ***low-level functional*** *primitives* and implement Translation to their imperative counterparts

3. Add *rewrite* rules to enable exploiting Tensor Cores via Rewriting

# 1. Low-Level Imperative Primitives and Codegen

```
template<typename FragmKind, int m, int n, int k,
 typename T, typename Layout=void> class fragment;

void mma_sync(
  fragment<...> &D,
  const fragment<...> &A,
  const fragment<...> &B,
  const fragment<...> &C);
void load_matrix_sync(fragment<...> &A,
  const T* tile, unsigned l_dim, layout_t layout);
void store_matrix_sync(T* tile,
  const fragment<...> &A,
  unsigned l_dim, layout_t layout);
void fill_fragment(
  fragment<...> &A, const T& value);
```

```
Fragment[m: Nat, n: Nat, k: Nat, t: DataType, f: FragmKind]

def mmaFragment(m:Nat, n:Nat, k:Nat, s:DataType, t:DataType,
  A: Exp[Fragment[m,k,n,s,AMatrix], Rd],
  B: Exp[Fragment[k,n,m,s,BMatrix], Rd],
  C: Exp[Fragment[m,n,k,t,Accum], Rd],
  D: Acc[Fragment[m,n,k,t,Accum]]): Comm
def loadFragment(f:FragmKind, m:Nat, n:Nat, k:Nat, t:DataType,
  tile: Exp[Array[m,Array[n,t]], Rd], A: Acc[Fragment[m,n,k,t,f]]): Comm
def storeFragment(m:Nat, n:Nat, k:Nat, t:DataType,
  A: Exp[Fragment[m,n,k,t,Accum],Rd], tile: Acc[Array[m,Array[n,t]]]): Comm
def fillFragment(f:FragmKind, m:Nat, n:Nat, k:Nat, t:DataType,
  A: Acc[Fragment[m,n,k,t,f]], value: Exp[t, Rd]): Comm
```

- Direct representation of CUDA API as imperative primitives in RISE
- Fragment types needed to be added to RISE
- Code generation is straightforward

# 2. Low-Level Functional Primitives and Translation

**functional primitives**

```
tensorMatMulAdd: {m: Nat} -> {n: Nat} -> {k: Nat} ->
  {s: DataType} -> {t: DataType} ->
  Fragment[m,k,n,s, AMatrix] ->
  Fragment[k,m,n,s, BMatrix] ->
  Fragment[m,n,k,t, Accum] -> Fragment[m,n,k,t, Accum]
asFragment: {m: Nat} -> {n: Nat} -> {k: Nat} ->
  {t: DataType} -> {f: FragmKind} ->
  Array[m, Array[n, t]] -> Fragment[m,n,k,t, f]
asMatrix: {m: Nat} -> {n: Nat} -> {k: Nat} -> {t: DataType} ->
  Fragment[m,n,k,t, Accum] -> Array[m, Array[n, t]]
generateFragment: {m: Nat} -> {n: Nat} -> {k: Nat} ->
    {t: DataType} -> {f: FragmKind} ->
    t -> Fragment[m,n,k,t, f]
```

**imperative primitives**

```
Fragment[m: Nat, n: Nat, k: Nat, t: DataType, f: FragmKind]

def mmaFragment(m:Nat, n:Nat, k:Nat, s:DataType, t:DataType,
  A: Exp[Fragment[m,k,n,s,AMatrix], Rd],
  B: Exp[Fragment[k,n,m,s,BMatrix], Rd],
  C: Exp[Fragment[m,n,k,t,Accum], Rd],
  D: Acc[Fragment[m,n,k,t,Accum]]): Comm
def loadFragment(f:FragmKind, m:Nat, n:Nat, k:Nat, t:DataType,
  tile: Exp[Array[m,Array[n,t]], Rd], A: Acc[Fragment[m,n,k,t,f]]): Comm
def storeFragment(m:Nat, n:Nat, k:Nat, t:DataType,
  A: Exp[Fragment[m,n,k,t,Accum],Rd], tile: Acc[Array[m,Array[n,t]]]): Comm
def fillFragment(f:FragmKind, m:Nat, n:Nat, k:Nat, t:DataType,
  A: Acc[Fragment[m,n,k,t,f]], value: Exp[t, Rd]): Comm
```

- One *low-level functional primitive* per *imperative primitive*
- Functional primitives have return values, rather than returning nothing (i.e. `void`/`Comm`)
- loading / storing a fragment corresponds to turning a matrix into a fragment (and reverse)

# 2. Low-Level Functional Primitives and Translation

Translation

- Translation by a case for each low-level functional primitive

```
def accT(expr: Phrase[Exp[d,Wr]],
         output: Phrase[Acc[t]]): Phrase[Comm] = expr match {
case tensorMatMulAdd(m,n,k,dt,dtAcc,aMatrix,bMatrix,cMatrix)
  => conT(aMatrix, fun(aMatrix => conT(bMatrix,
  fun(bMatrix => conT(cMatrix, fun(cMatrix =>
    mmaFragment(m, n, k, dt,
     dtAcc, aMatrix, bMatrix, cMatrix, A)))))))
case asFragment(m, n, k, dt, f, tile)
  => conT(tile, fun(tile: =>
   loadFragment(f, m, n, k, dt, tile, A)))
case asMatrix(m, n, k, dt, frag)
  => conT(frag, fun(frag: =>
   storeFragment(m, n, k, dt, frag, A)))
case generateFragment(m, n, k, dt, f, fill)
  => conT(fill, fun(fill =>
   fillFragment(f, m, n, k, dt, fill, A)))
... }
```

# 3. Add Rewrite Rules To Enable Rewriting

- Rewrite rules enable automatic exploitation of Tensor Cores

- Examples shows automatic use of Tensor Cores for high-level matrix multiplication code

- Rewrite rules can be applied *automatically* [GPGPU'16, ICFP'15], *manually* [ICFP'20], or *guided* [arXiv:2111.13040].

```
aTile: Array[16,Array[16,f16]] |> map(fun(aRow =>
 bTile: Array[16,Array[16,f16]] |> map(fun(bCol =>
  zip(aRow, bCol) |>
  reduceSeq(fun(ac, ab =>
   add(ac, mul(fst(ab), snd(ab)))))(0.0)))))
```

↧

```
tensorMatMulAdd
 (aTile: Array[16,Array[16,f16]] |> asFragment |> toMem(Local))
 (bTile: Array[16,Array[16,f16]] |> transpose
  |> asFragment |> toMem(Local))
 (generateFragment(0.0) |> toMem(Local))
|> toMem(Local) |> asMatrix
```

# RISE & `Shine`: Language-Oriented Compiler Design

- `Shine` demonstrates an extensible compiler design allowing targeting specialised hardware

- Progressive compilation is a good idea:
  *High-level functional primitives* via Rewriting to
  *low-level functional primitives* via Translation to
  *low-level imperative primitives* via Codegen to
  *low-level imperative code*.

https://rise-lang.org/