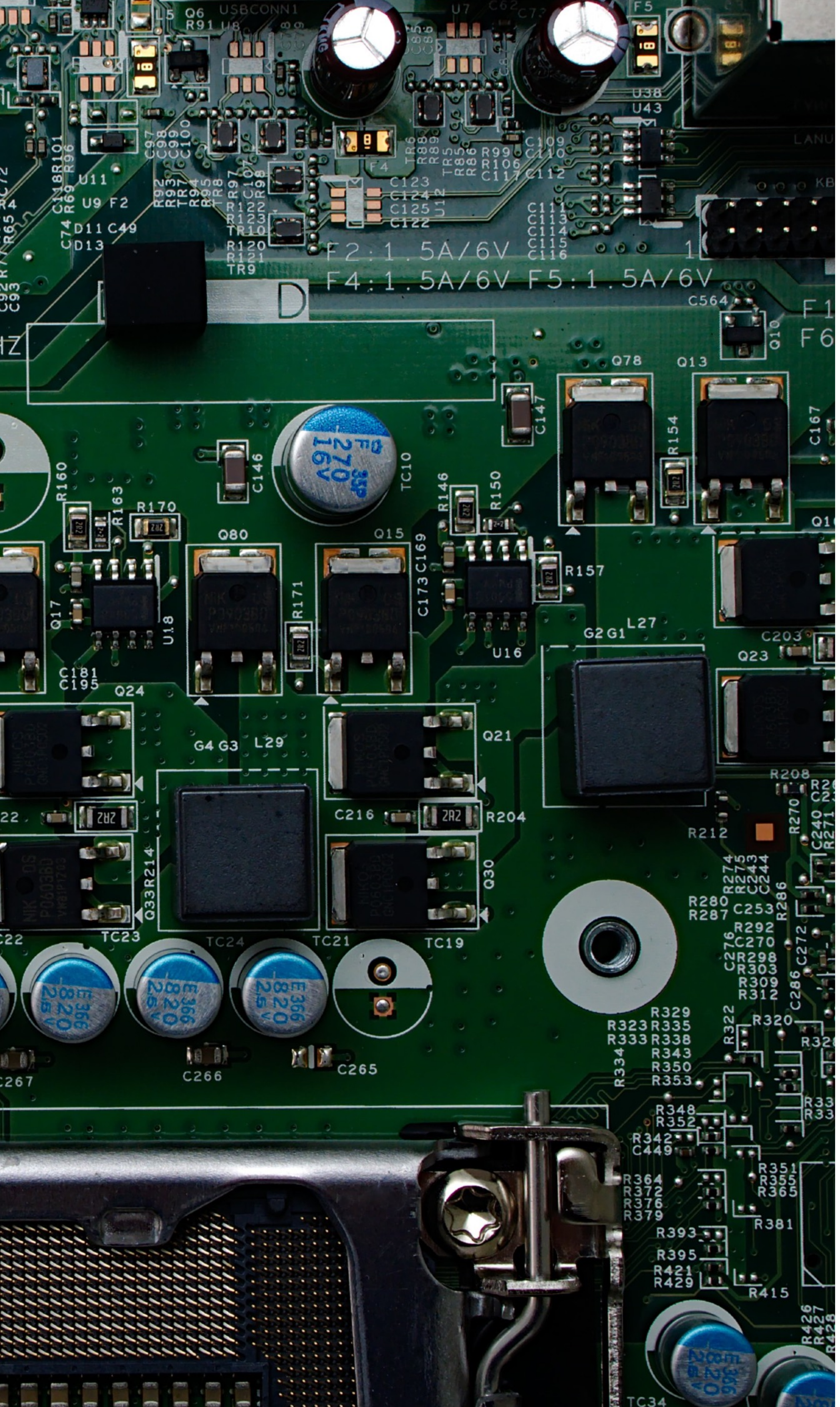


How to design the next 700 optimizing compilers

A framework for designing optimising domain-specific compilers for specialised hardware in the era of ML and AI

Michel Steuwer

THE UNIVERSITY of EDINBURGH

General purpose **Software** Specialised

THE C PROGRAMMING LANGUAGE

C++

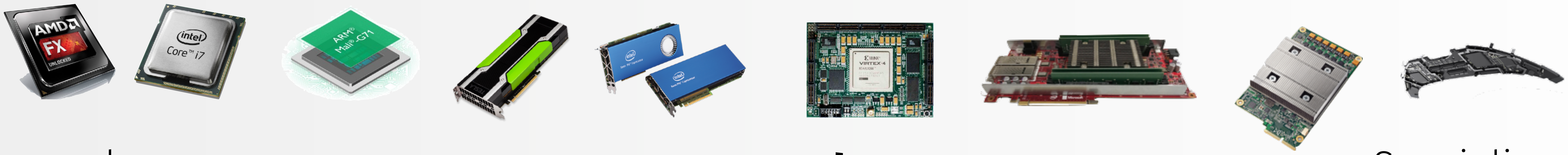
NVIDIA CUDA

TensorFlow

Stan

tvm

Halide



General purpose **Hardware** Specialised

General purpose **Software** Specialised

THE C PROGRAMMING LANGUAGE

C++

NVIDIA CUDA

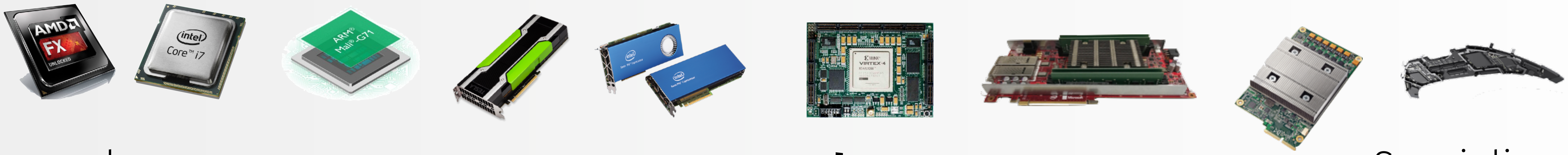
TensorFlow

Stan

tvm

Halide

How do we build compilers to (automatically) optimise specialised software for specialised hardware?



General purpose **Hardware** Specialised

COMPUTATION

OPTIMISATION

Domain Specific Example: TensorFlow

- > 500 different type of nodes in the TF IR
- > 50 different type of nodes in the XLA IR
- > 2.500.000 lines of code
- Support for custom hardware: TPU

◉ **Hughe effort to build still highly specialised**

- Problem solved?

[HotOS'19]

Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
Google Brain

Original authors of TensorFlow

Abstract

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on high-performance but inflexible kernels reinforces the dominant style of programming model, and argue these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress.

We conclude by noting promising directions in the field, and advocate steps to advance progress towards high-performance general purpose numerical computing systems on modern accelerators.

ACM Reference Format:

Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321441

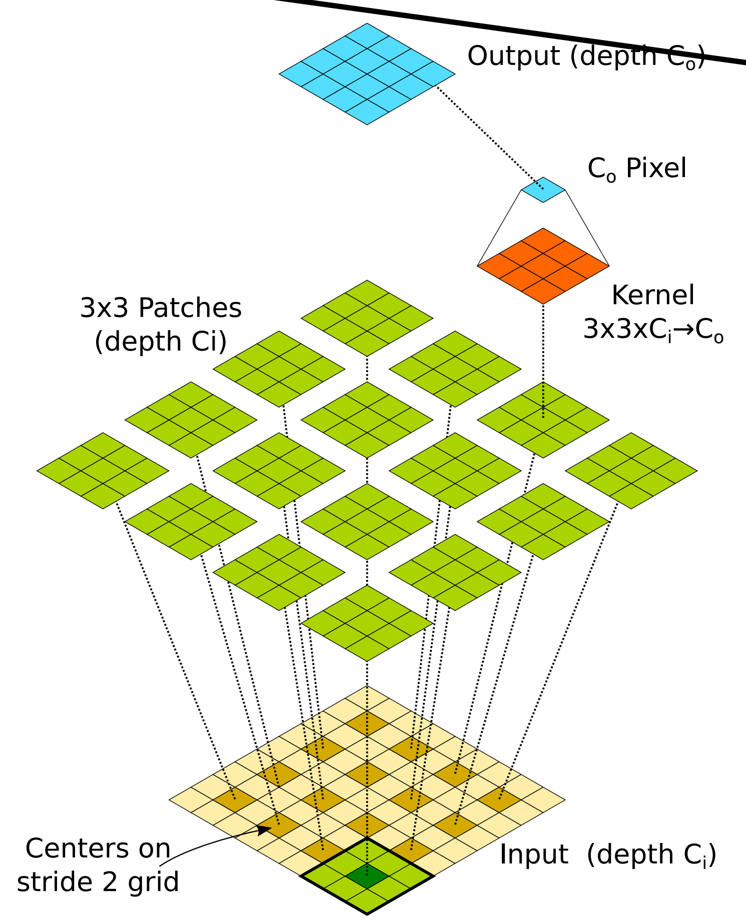


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convolutional neural network (CNN) we were comparing it to, implementations in both TensorFlow[2] and PyTorch[3] were much slower and ran out of memory with much smaller models. We wanted to understand why.

1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper[1], but for our purposes it is sufficient to consider a simplified form of the inner loop, which is

Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
Google Brain

Abstract

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on high-performance but inflexible kernels reinforces the dominant style of programming model, and argue these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress.

We conclude by noting promising directions in the field, and advocate steps to advance progress towards high-performance general purpose numerical computing systems on modern accelerators.

ACM Reference Format:
Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321441

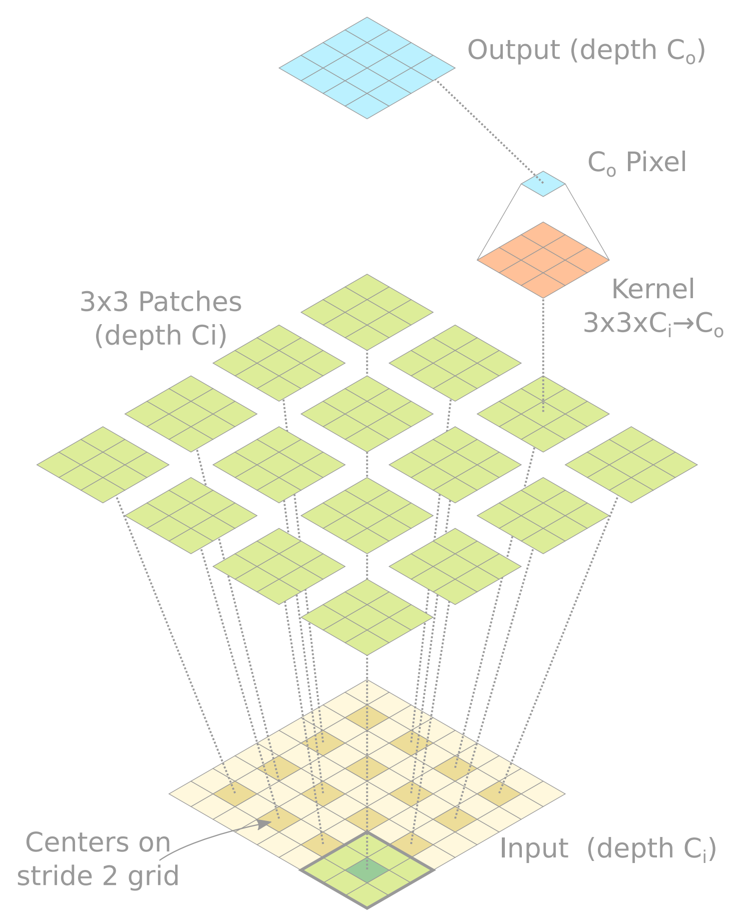


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convolutional neural network (CNN) we were comparing it to, implementations in both TensorFlow[2] and PyTorch[3] were much slower and ran out of memory with much smaller models. We wanted to understand why.

1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper[1], but for our purposes it is sufficient to consider a simplified form of the inner loop, which is

Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
Google Brain

Abstract

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on high-performance but inflexible kernels reinforces the dominant style of programming model, and argue these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress.

We conclude by noting promising directions in the field, and advocate steps to advance progress towards

We should aim for more principled higher level intermediate representations

Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321441

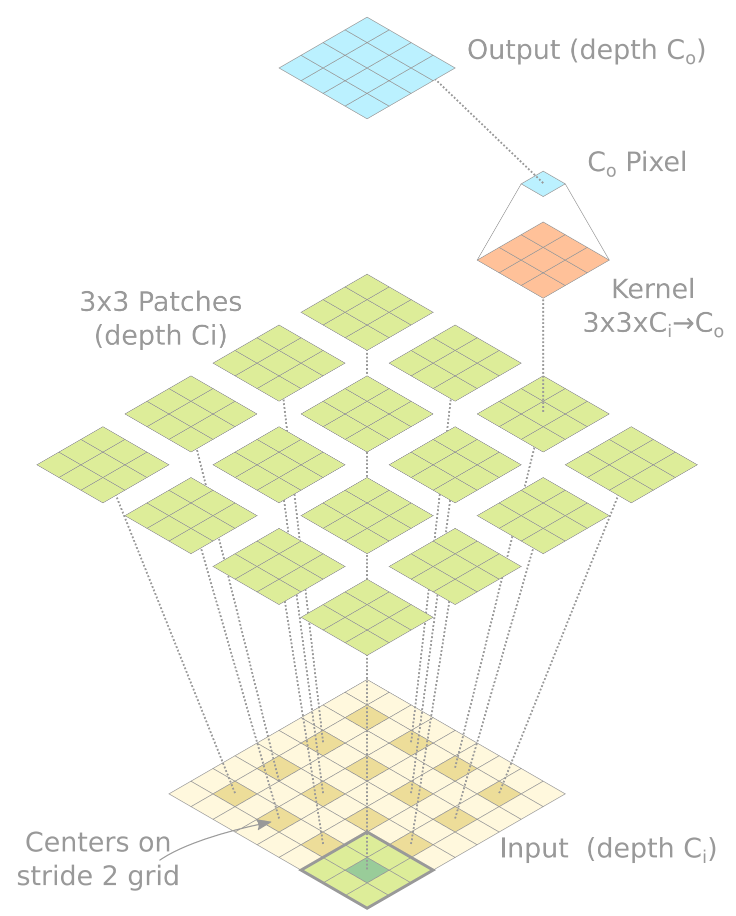


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convo-

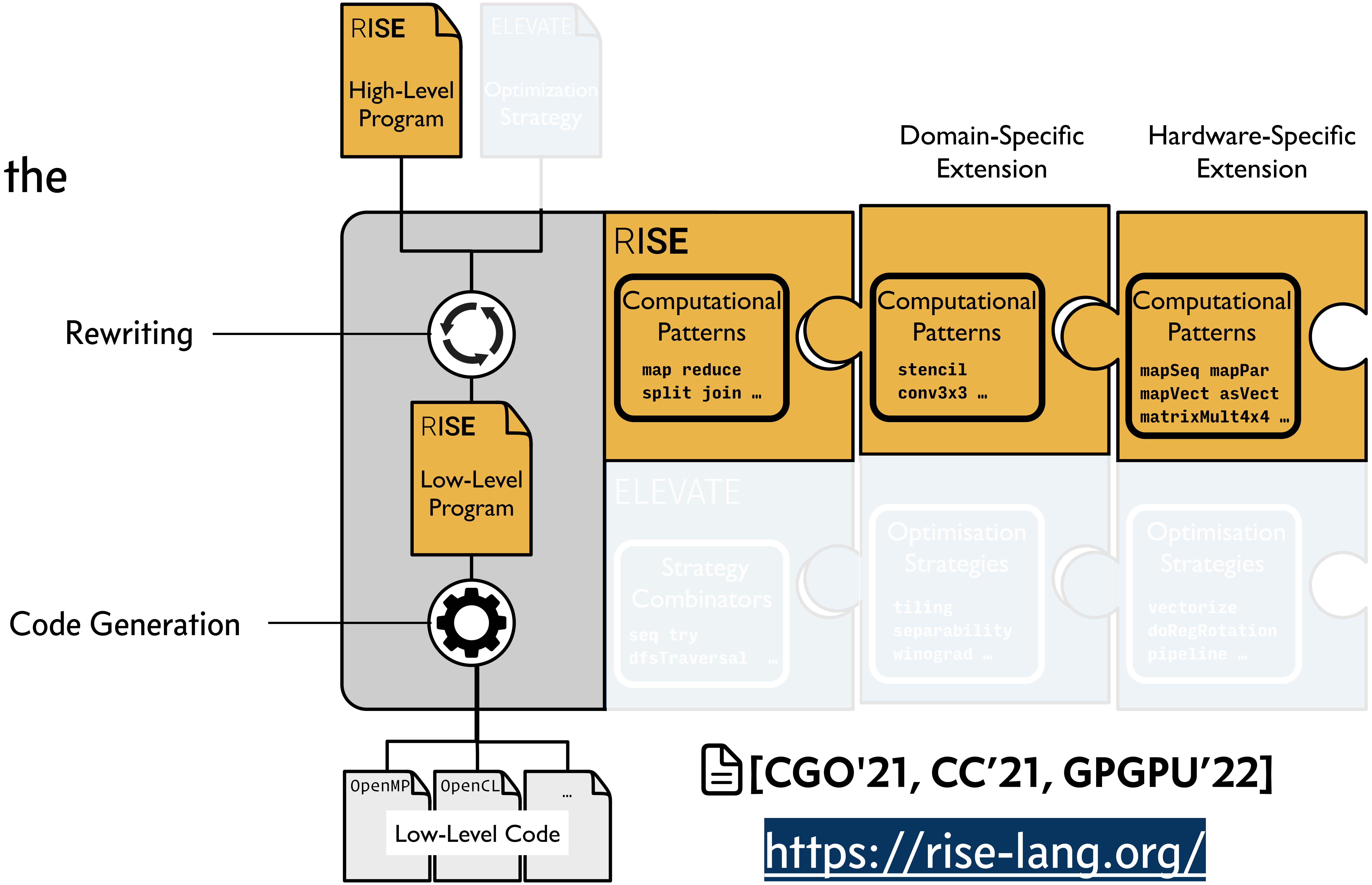
1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper[1], but for our purposes it is sufficient to consider a simplified form of the inner loop, which is

COMPUTATION

RISE & Shine an extensible compiler design

- Spiritual successor to the LIFT project
- Functional language as foundation
- Computations are expressed by computational patterns

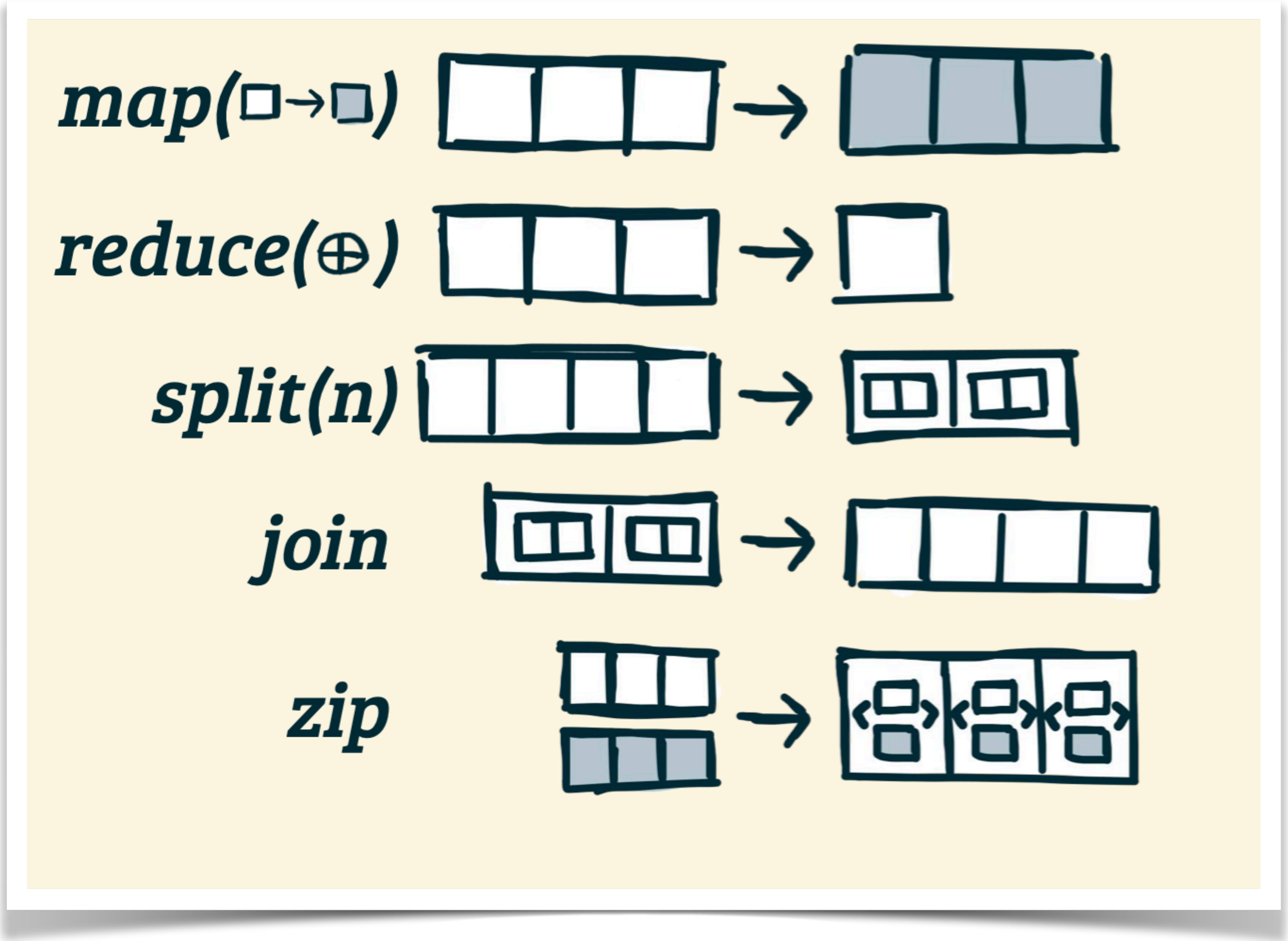


[CGO'21, CC'21, GPGPU'22]

https://rise-lang.org/

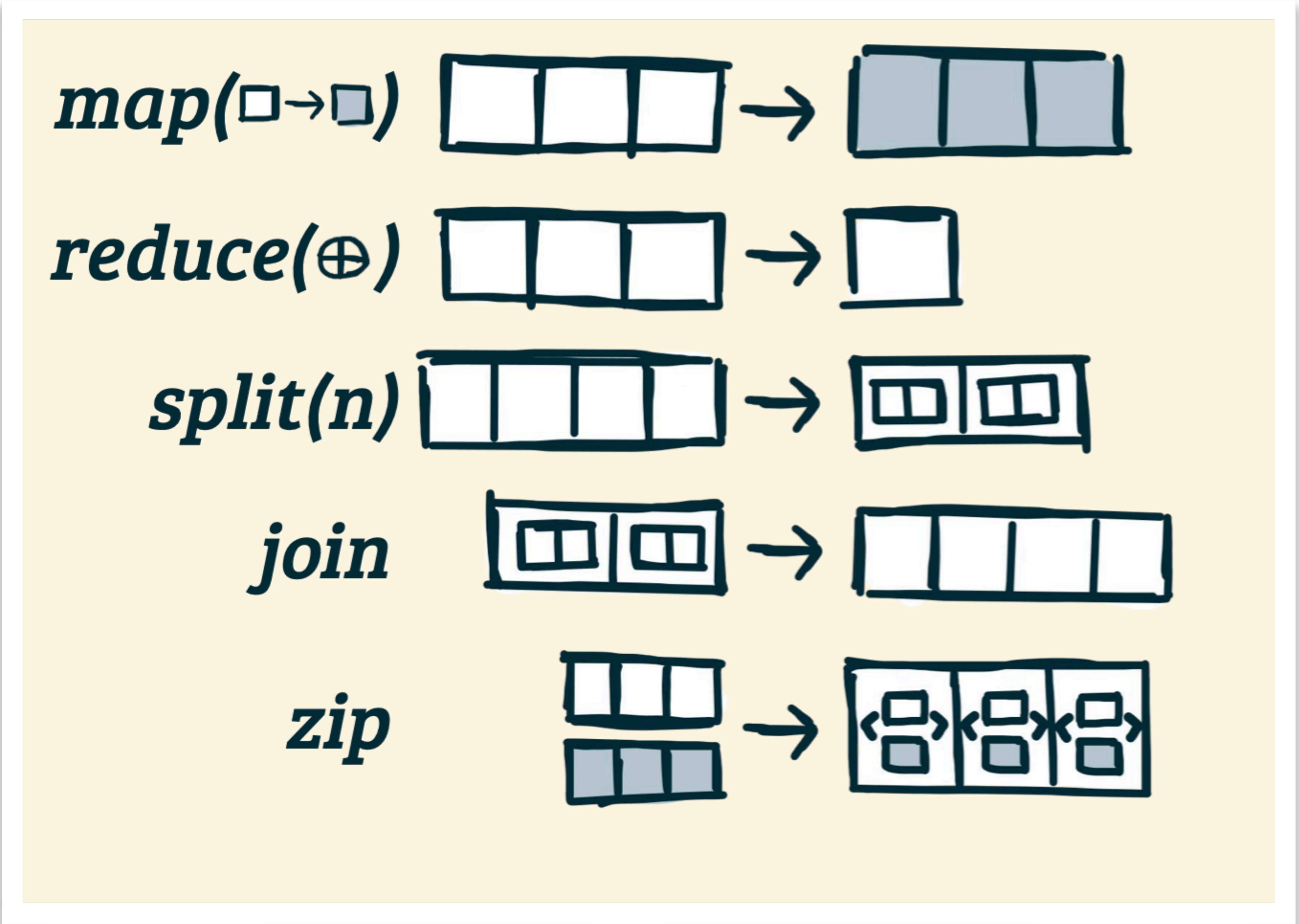
Computational Patterns

Data parallel patterns

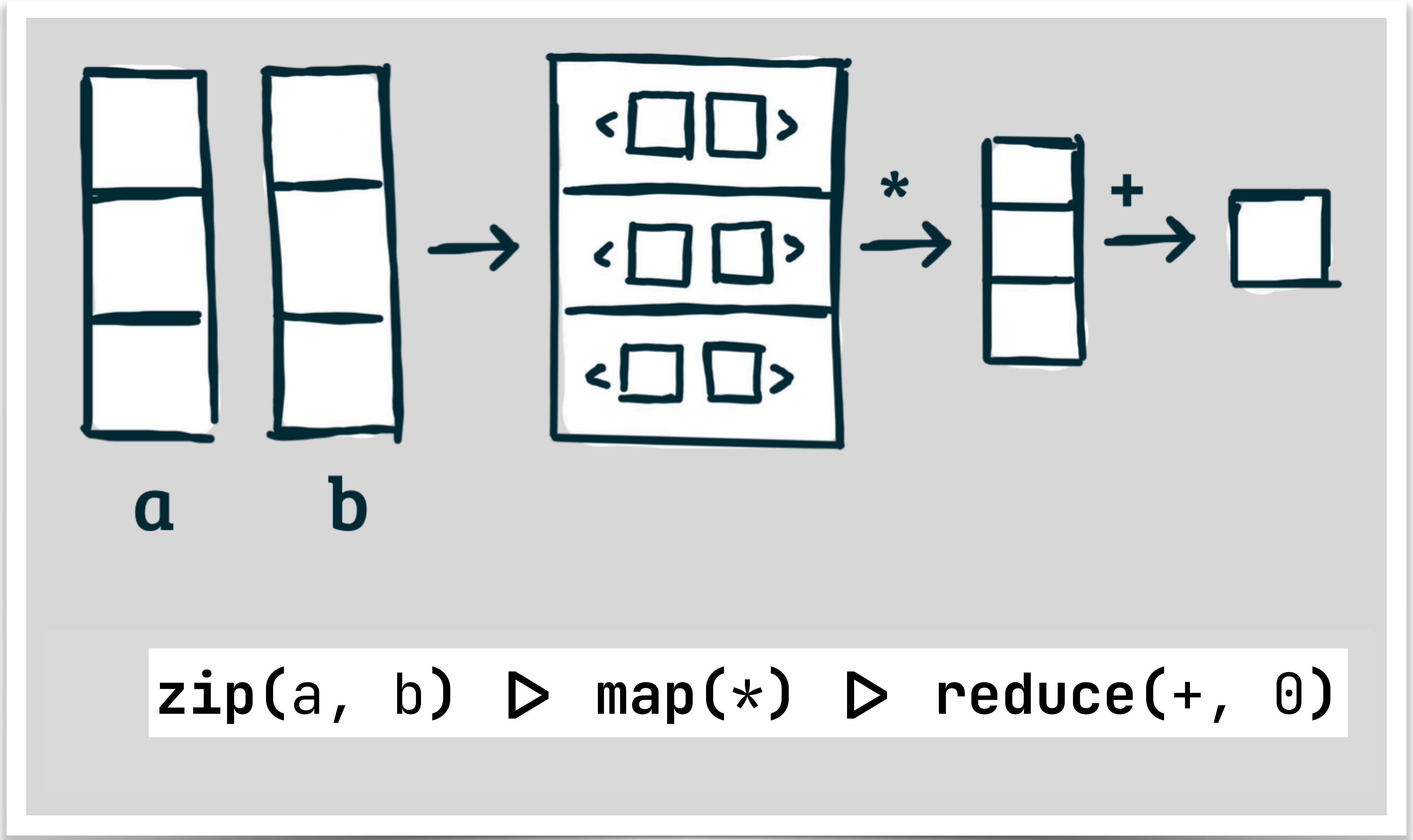


Computational Patterns

Data parallel patterns

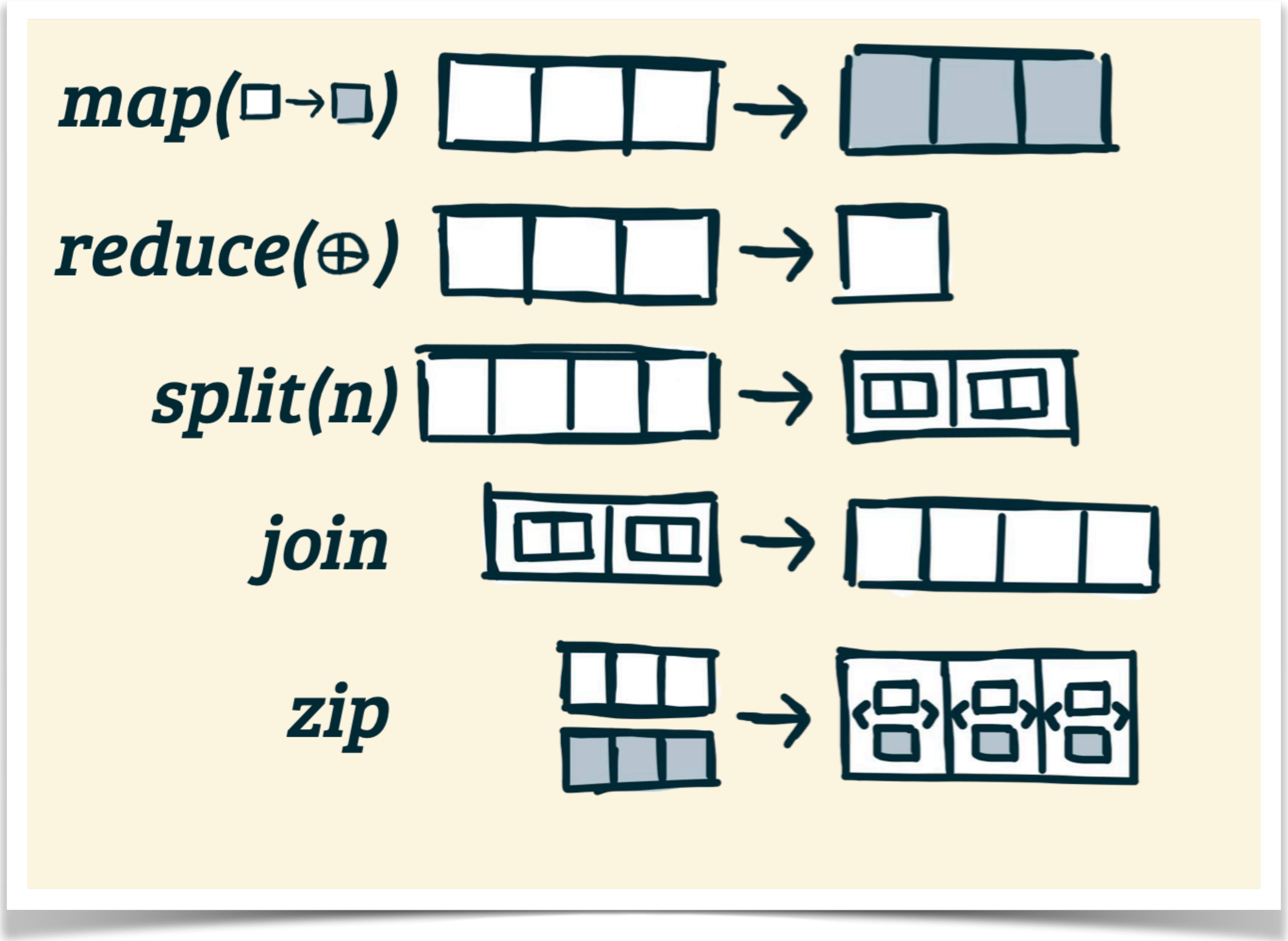


Dot product



Computational Patterns

Data parallel patterns



Matrix multiply

```
depFun((m:Nat, n:Nat, k:Nat) ⇒
 fun((A: Array[m,Array[k,f32]],
      B: Array[k,Array[n,f32]]) ⇒

   A ▷ map(fun(rowA ⇒
     B ▷ transpose ▷ map(fun(colB ⇒
       dot(rowA, colB)

))))))
```

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

High-Level functional primitives

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

High-Level functional primitives

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))) ))))))
```

Low-Level functional primitives

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

High-Level functional primitives

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))) ))))))
```

Low-Level functional primitives

Translation

Low-Level imperative primitives

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

High-Level functional primitives

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))) ))))))
```

Low-Level functional primitives

Translation

Low-Level imperative primitives

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 + row
    }
    __syncthreads(); }}
```

Low-Level imperative code

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

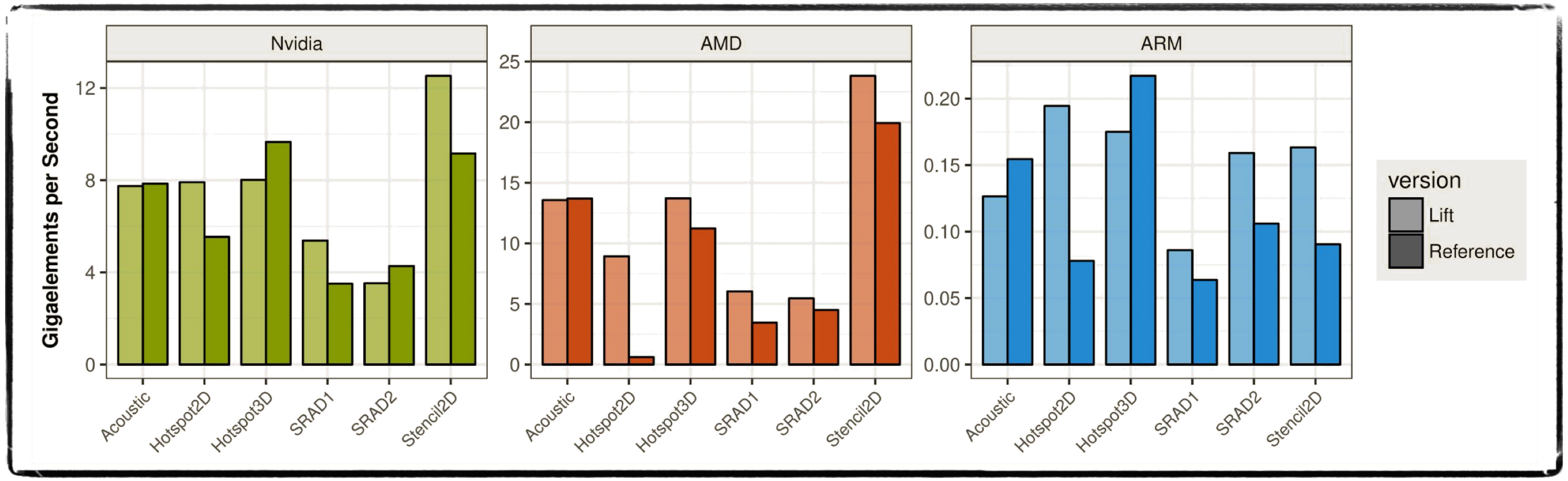
```
depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta
 parForBlock(m,Array[n,f16], output, fun(rowIdx,outRow =>
  parForThreads(n,f16, outRow, fun(colIdx,outElem =>
   new(Local,f32, fun((accumExp, accumAcc) =>
    accumAcc = 0.0f;
    for(k, fun(i => accumAcc = accumExp +
     fst(idx(i, zip(fst(idx(rowIdx,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

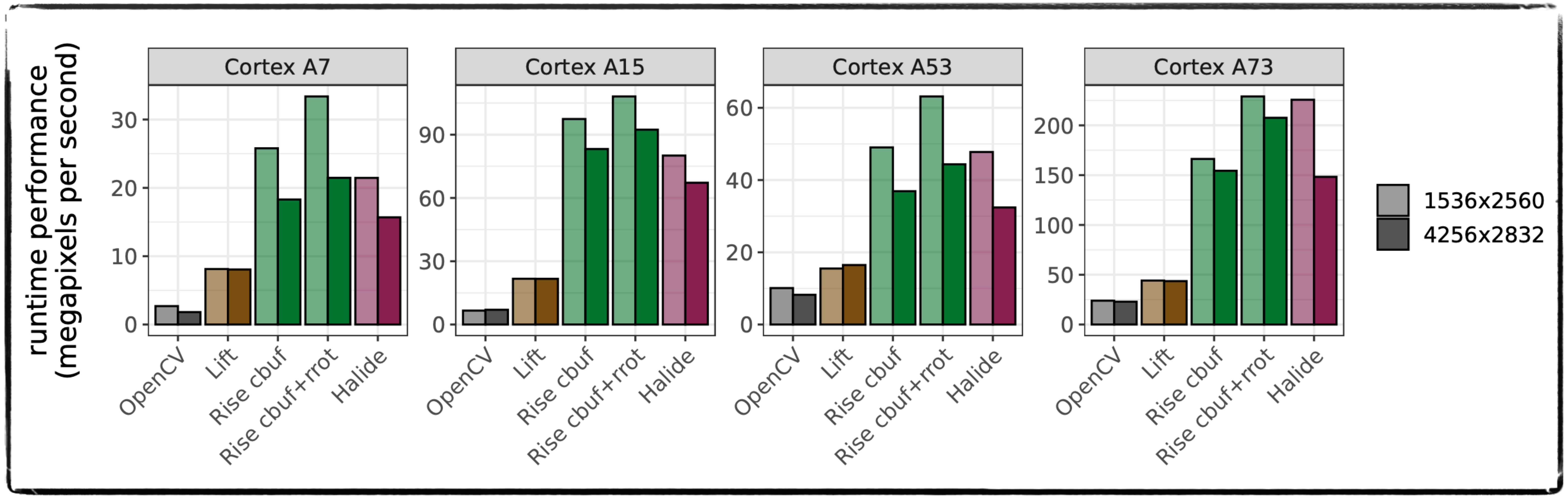
```
__global__ void gemm_kernel(float* __restrict__ output,
 int m, int n, int k, const __half* __restrict__ A,
 const __half* __restrict__ B,
 const float* __restrict__ C, float alpha, float beta) {
  for(int rowIdx=blockIdx.x;
      blockIdx.x<m; rowIdx += gridDim.x) {
   for(int colIdx=threadIdx.x;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

Performance Results



Same performance as hand-optimised code!

[CGO 2018]



Outperform Halide with two optimizations added as new patterns.

[CGO 2021]

Extensibility!

- New patterns can be added at each abstraction layer:
  - *Low-level imperative primitives* to capture a hardware details
  - *Low-level functional primitives* to lift these abstractions into the functional world
  - High-level functional primitives to make these abstractions available to rewriting

Low-level imperative primitives

```
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
```

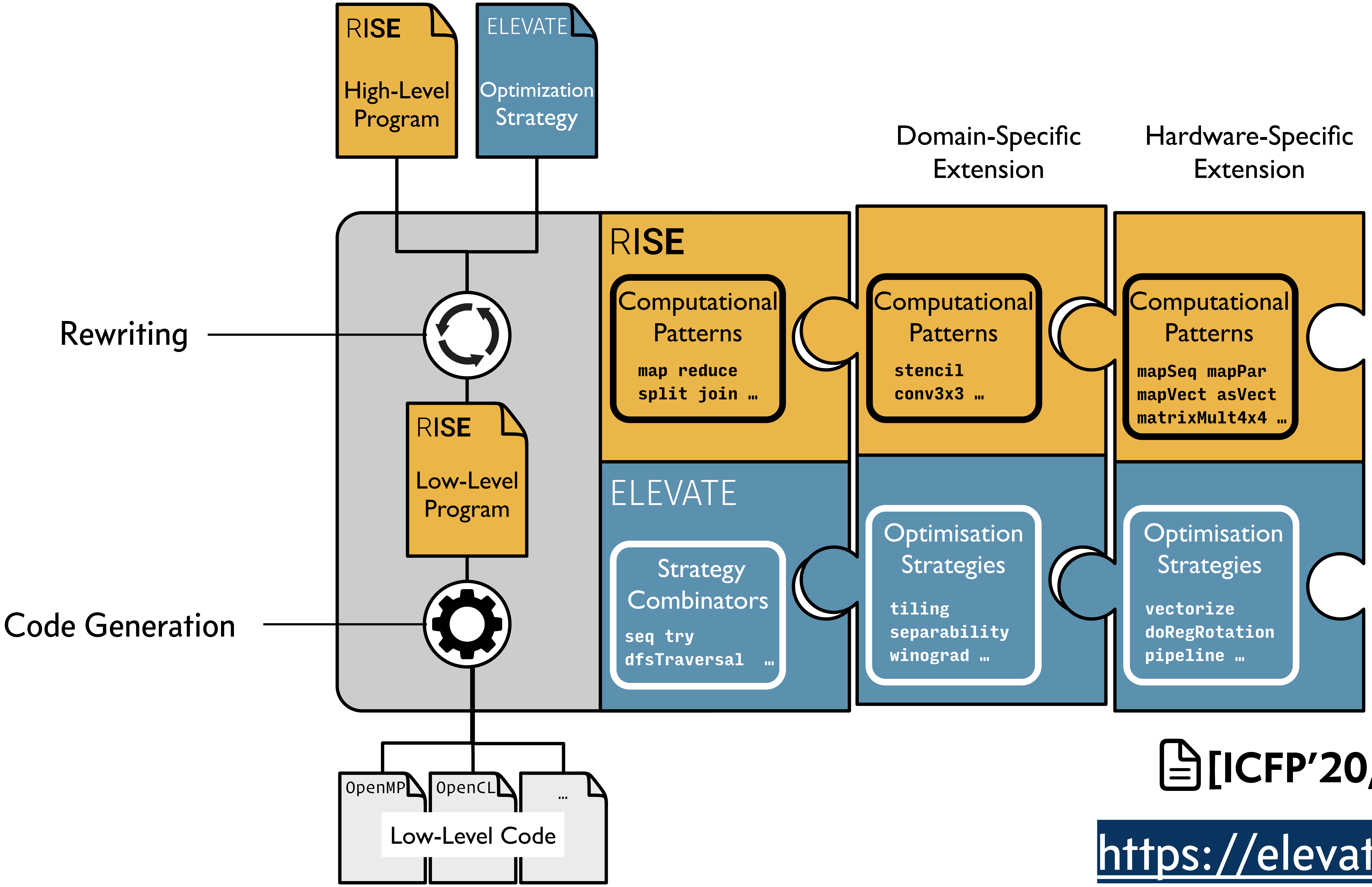
Low-level functional primitives

```
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]
```

[GPGPU'22]

OPTIMISATION

Extensible Optimizations via Rewriting



[ICFP'20, arXiv'22]

https://elevate-lang.org/

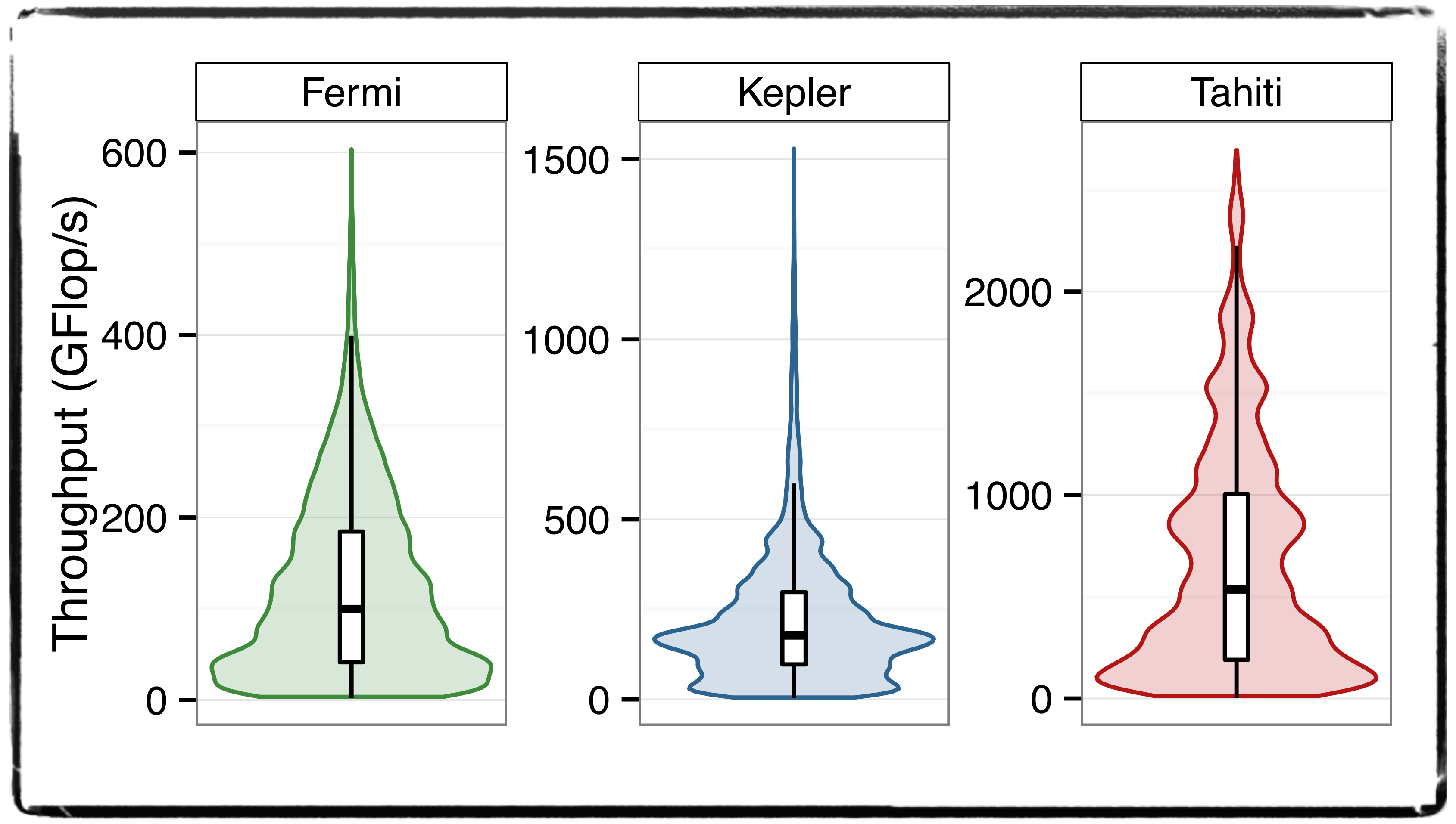
Tradeoffs when optimizing with rewriting

Automatic rewriting

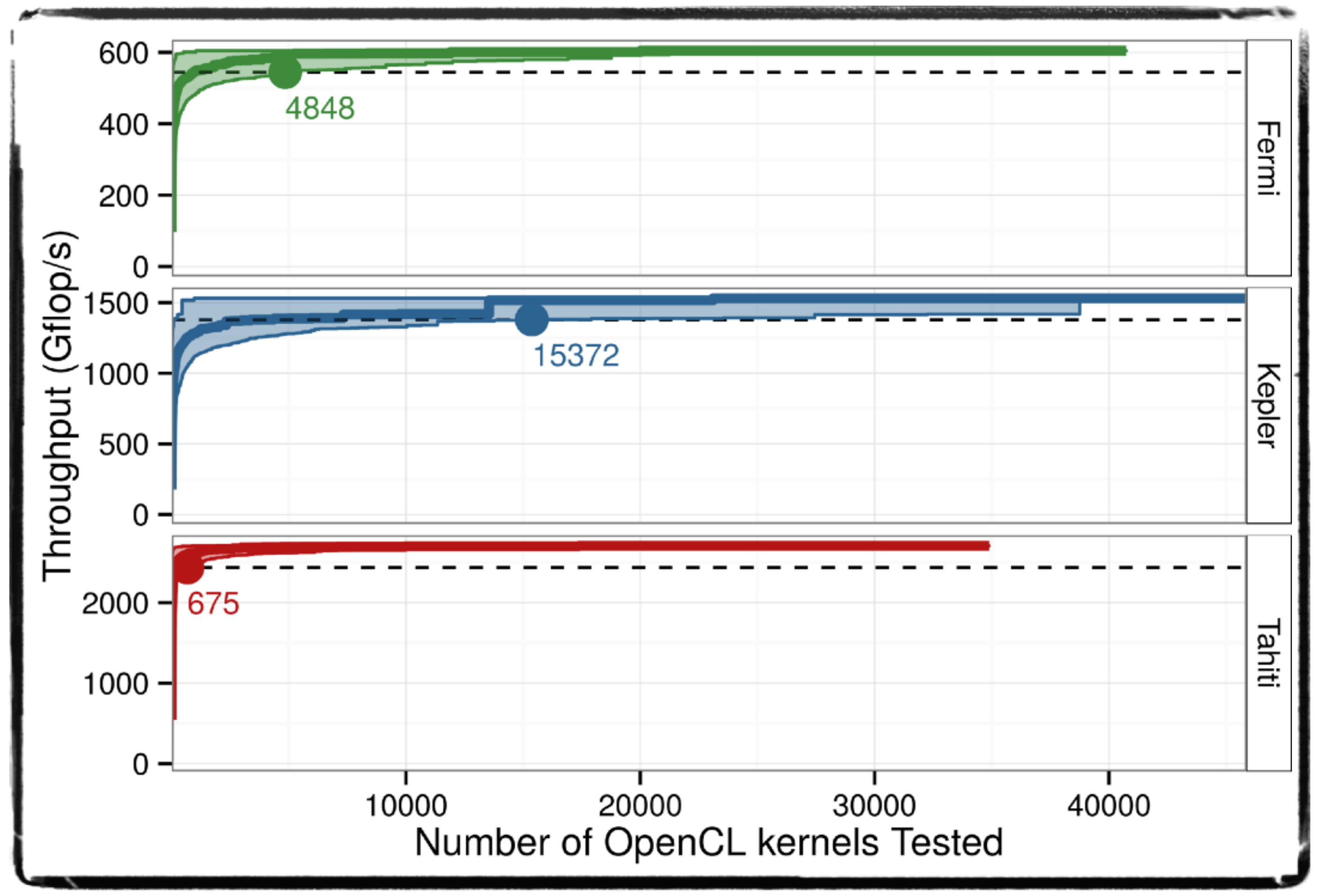
Manual rewriting

✅ No human needed in optimization process

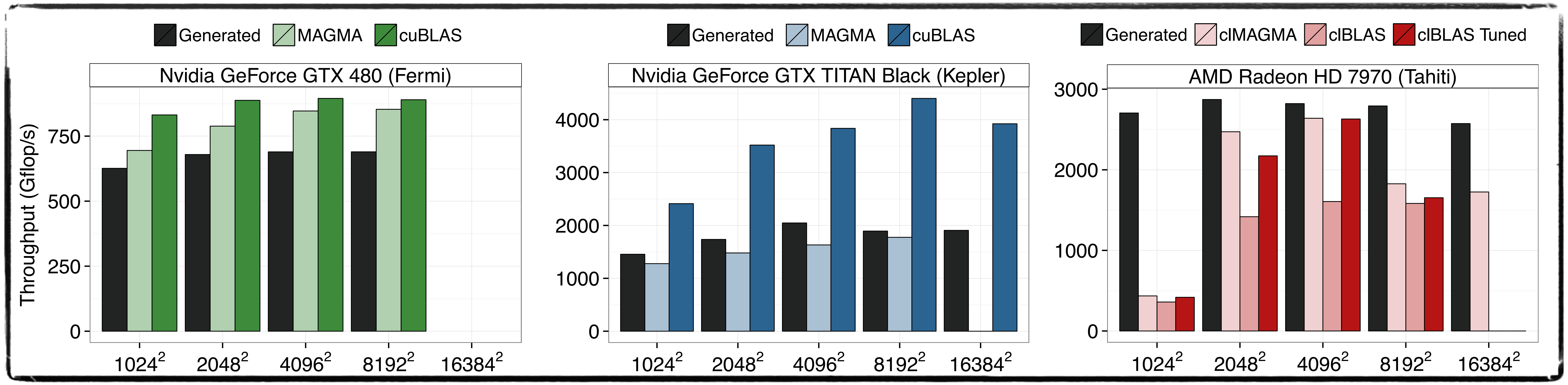
Automatic Rewriting for Matrix Multiplication



Only few generated code with very good performance



Still: One can expect to find a good performing kernel quickly!



Performance close or better than hand-tuned library code

[GPGPU’16]

Tradeoffs when optimizing with rewriting

- ✓ No human needed in optimization process
- ✗ **Costly & Lengthy search process**
- ✗ **Does not (yet) scale to complex programs**

Tradeoffs when optimizing with rewriting

Automatic rewriting

Manual rewriting

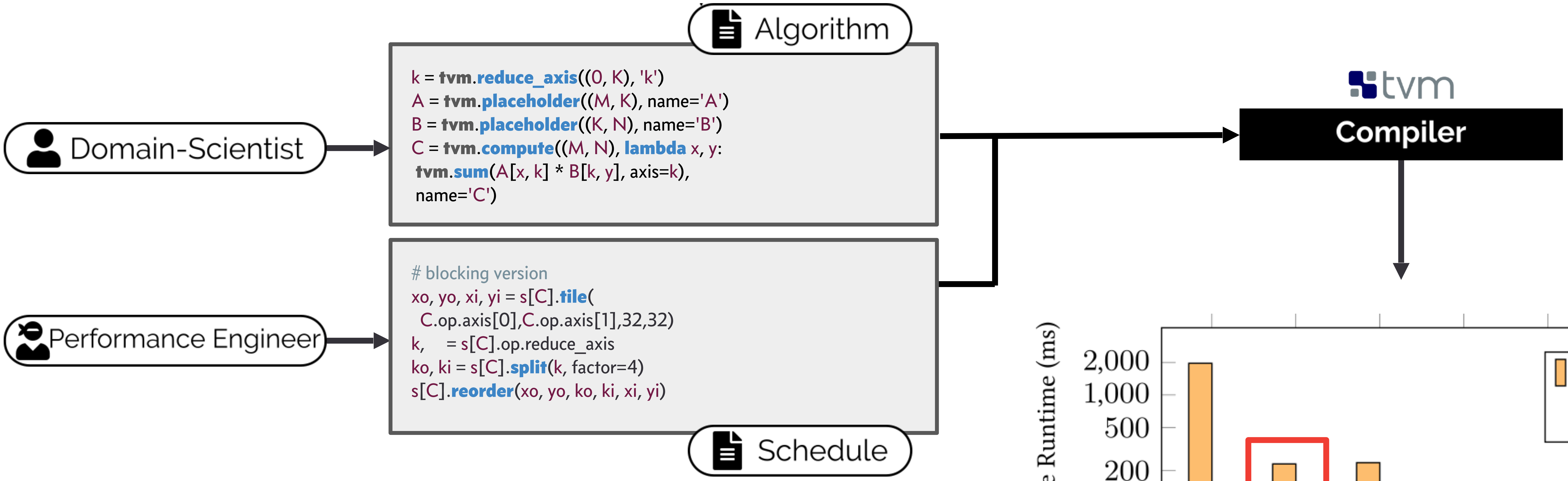
Automatic rewriting:

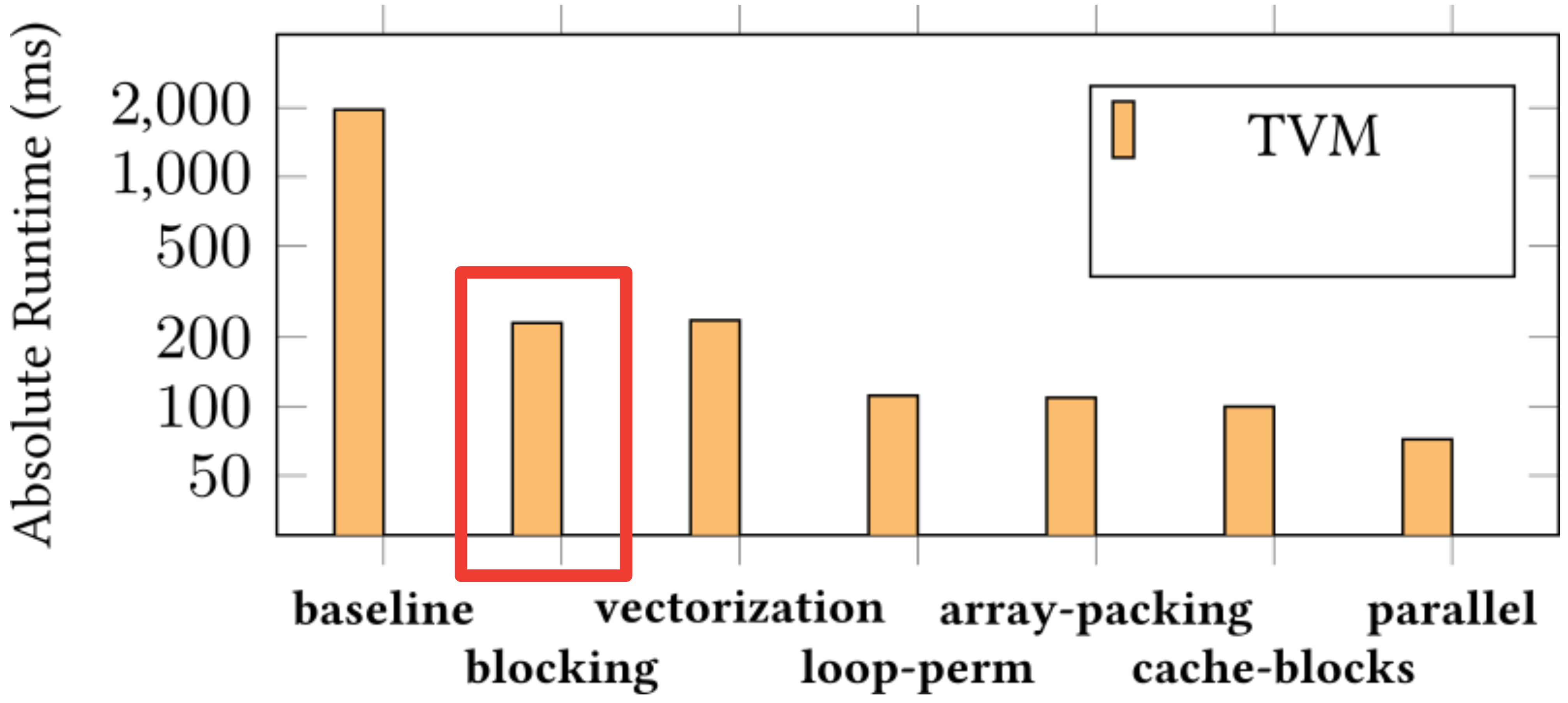
- ✓ No human needed in optimization process
- ✗ Costly & Lengthy search process
- ✗ Does not (yet) scale to all programs

Manual rewriting:

- ✗ Extensive human effort needed
- ✓ Expert is in control, no search required

Compilers with scheduling APIs





Compilers with scheduling APIs

Domain-Scientist

Algorithm

```
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N), lambda x, y:
 tvm.sum(A[x, k] * B[k, y], axis=k),
 name='C')
```

Performance Engineer

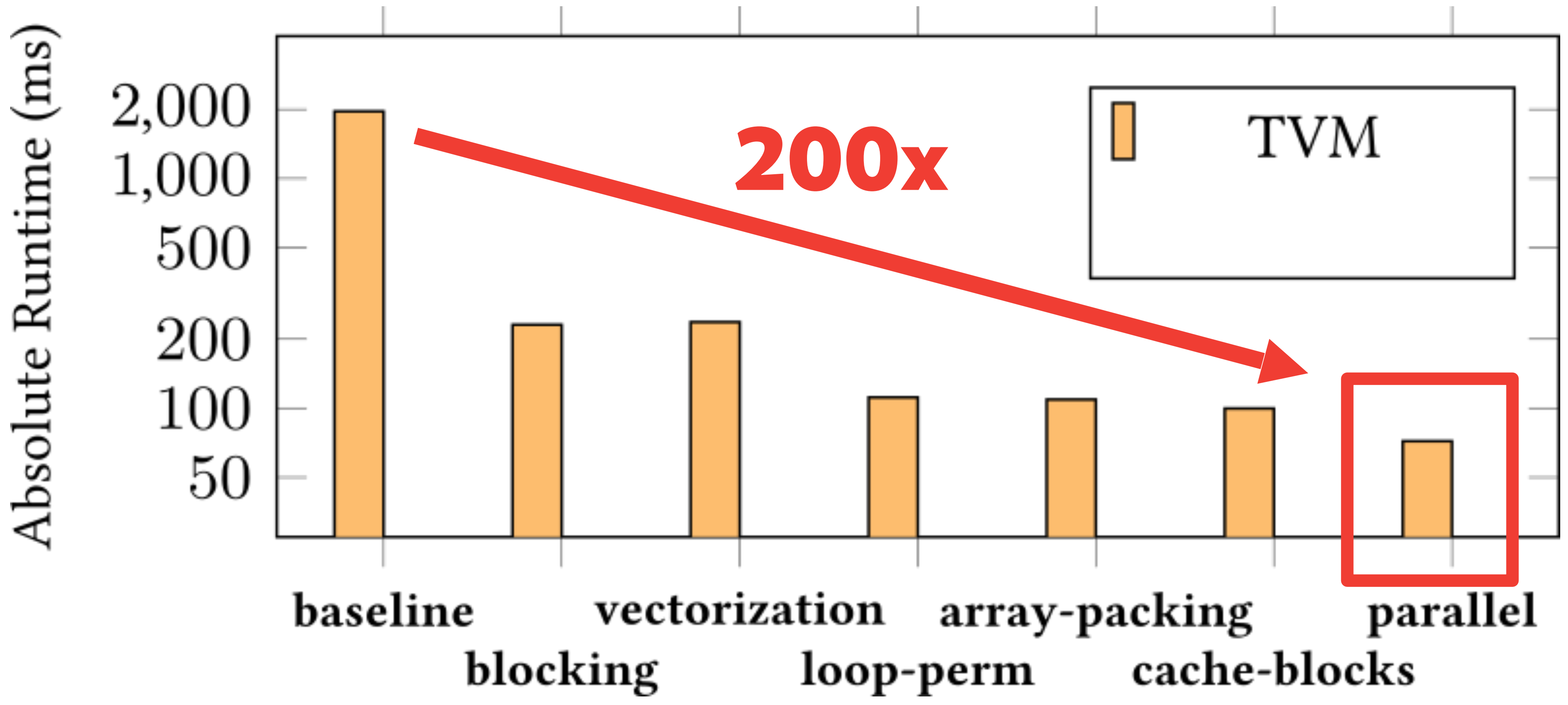
```
# "parallel schedule
s = tvm.create_schedule(C.op)
CC = s.cache_write(C, 'global')
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)

s[CC].compute_at(s[C], yo)
xc, yc = s[CC].op.axis
k, = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)
s[C].parallel(xo)
x, y, z = s[packedB].op.axis
s[packedB].vectorize(z)
s[packedB].parallel(x)
```

Schedule

tvm

Compiler



Compilers with scheduling APIs

Program

Algorithm

Domain-Scientist

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

Performance Engineer

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

Schedule

Optimisation Schedule

Halide compiler

Optimised Code

Compilers with scheduling APIs

Halide

tvm

Tiramisu-Compiler / **tiramisu**

Fireiron

NVIDIA

Problems with Scheduling APIs

Program

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
```

```
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
```

```
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Program

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

No clear separation

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
```

```
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
```

```
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Hinders reuse

Program

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
```

```
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
```

```
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Hinders reuse

Not well defined semantics

Program

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
```

```
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
```

```
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
    .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
    .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
    .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
    .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
    .unroll(xo).unroll(Ay);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Hinders reuse

Hinders understanding

Program

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
```

```
.store_in(MemoryType::Register).compute_at(out, x)
.split(x, xo, xi, warp_size * vec_size, RoundUp)
.split(y, ty, y, y_unroll)
.gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
.unroll(xo).unroll(y).update()
.split(x, xo, xi, warp_size * vec_size, RoundUp)
.split(y, ty, y, y_unroll)
.gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
.split(r.x, rxo, rxi, warp_size)
.unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
.unroll(xo).unroll(y);
```

```
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Hinders reuse

Hinders understanding

Not well-d… …ntics

Only fixed built-in optimisations

Program

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
```

```
.store_in(MemoryType::Register).compute_at(out, x)
.split(x, xo, xi, warp_size * vec_size, RoundUp)
.split(y, ty, y, y_unroll)
.gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
.unroll(xo).unroll(y).update()
.split(x, xo, xi, warp_size * vec_size, RoundUp)
.split(y, ty, y, y_unroll)
.gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
.split(r.x, rxo, rxi, warp_size)
.unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
.unroll(xo).unroll(y);
```

```
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
     .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
     .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
     .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
     .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
     .unroll(xo).unroll(Ay);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Hinders reuse

Hinders understanding

No extensibility

Program

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
```

```
.store_in(MemoryType::Register).compute_at(out, x)
.split(x, xo, xi, warp_size * vec_size, RoundUp)
.split(y, ty, y, y_unroll)
.gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
.unroll(xo).unroll(y).update()
.split(x, xo, xi, warp_size * vec_size, RoundUp)
.split(y, ty, y, y_unroll)
.gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
.split(r.x, rxo, rxi, warp_size)
.unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
.unroll(xo).unroll(y);
```

```
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

Optimisation Schedule

Halide compiler

`Optimised Code`

Problems with Scheduling APIs

Program

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);
```

```
// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
```

```
.store_in(MemoryType::Register).compute_at(out, x)
.split(x, xo, xi, warp_size * vec_size, RoundUp)
.split(y, ty, y, y_unroll)
.gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
.unroll(xo).unroll(y).update()
.split(x, xo, xi, warp_size * vec_size, RoundUp)
.split(y, ty, y, y_unroll)
.gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
.split(r.x, rxo, rxi, warp_size)
.unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
.unroll(xo).unroll(y);
```

```
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
```

```
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

Optimisation Schedule

Hinders reuse

Hinders understanding

No extensibility

Halide compiler

Code

We should aim for more principled ways to describe and apply optimisations

The Need for a Principled Way to Separate, Describe and Apply Optimizations

Our goals:

1. *Separate concerns*
   Computations should be expressed at a high abstraction level only.
   They should not be changed to express optimizations;

2. *Facilitate reuse*
   Optimization strategies should be defined clearly separated from the computational program facilitating reusability of computational programs and strategies;

3. *Enable composability*
   Computations *and* strategies should be written as compositions of user-defined building blocks (possibly domain-specific ones); both languages should facilitate the creation of higher-level abstractions;

4. *Allow reasoning*
   Computational patterns, but also especially strategies, should have a precise, well-defined semantics allowing reasoning about them;

5. *Be explicit*
   Implicit default behavior should be avoided to empower users to be in control.

The Need for a Principled Way to Separate, Describe and Apply Optimizations

Our goals:

1. *Separate concerns*

   Computations should be expressed at a high abstraction level only.

Fundamentally we argue that a more principled high-performance code generation approach should be holistic by considering *computation* and *optimization strategies* **equally important.**

As a consequence, a strategy language should be built with the same standards as a language describing computation.

Computational patterns, but also especially strategies, should have a precise, well-defined semantics allowing reasoning about them;

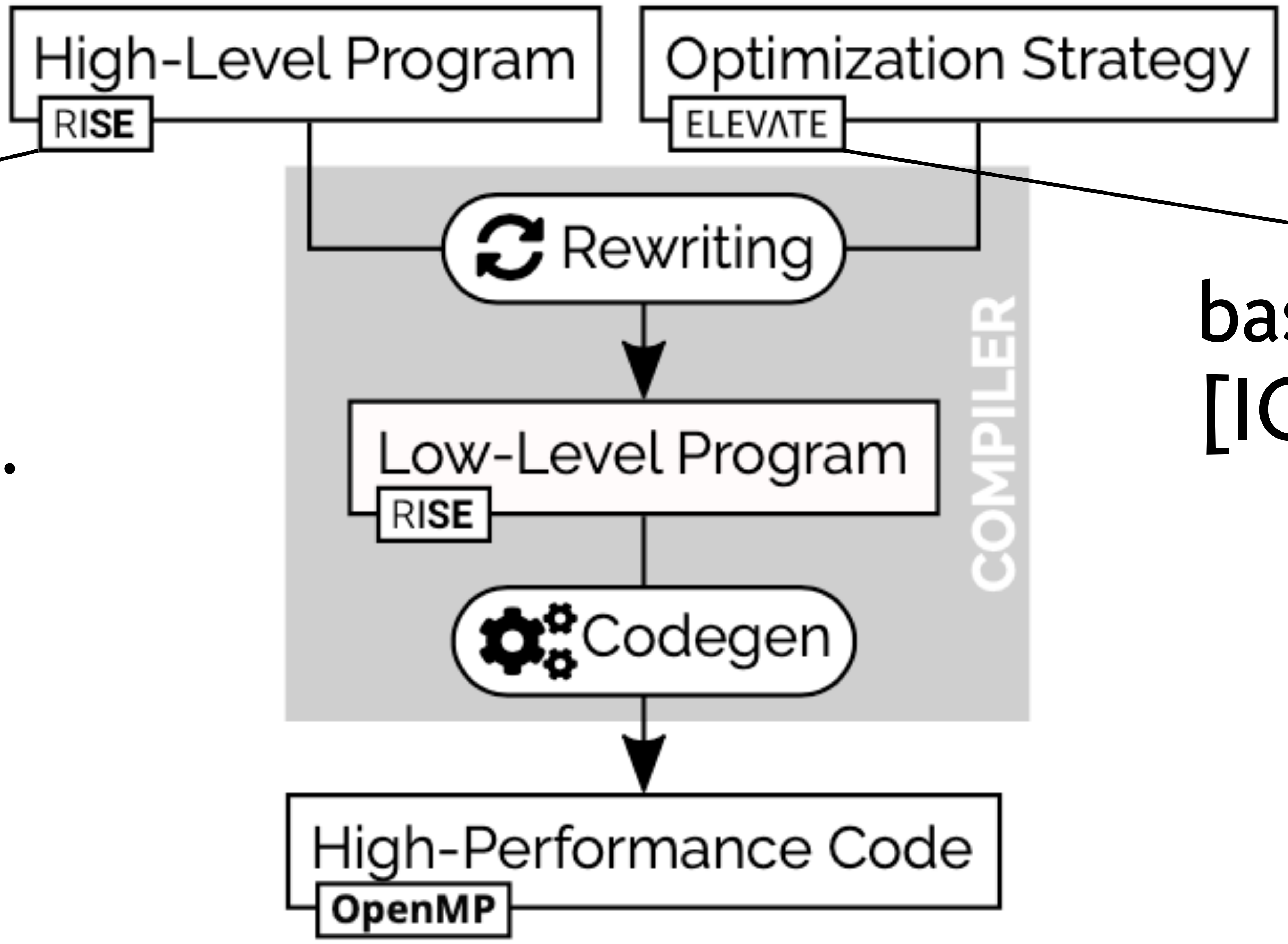
5. *Be explicit*

   Implicit default behavior should be avoided to empower users to be in control.

Achieving High-Performance the Functional Way

```
// Matrix Matrix Multiplication in RISE
val dot = fun(as, fun(bs,
  zip(as)(bs) |> map(fun(ab, mult(fst(ab))(snd(ab)))) |> reduce(add)(0) ) )
val mm  = fun(a : M.K.float, fun(b : K.N.float,
  a |> map(fun(arow,                    // iterating over M
    transpose(b) |> map(fun(bcol,       // iterating over N
      dot(arow)(bcol) )))) ) )          // iterating over K
```

```
val loopPerm = (
  tile(32,32)        ‘@‘ outermost(mapNest(2))     ‘;;‘
  fissionReduceMap   ‘@‘ outermost(appliedReduce)  ‘;;‘
  split(4)           ‘@‘ innermost(appliedReduce)  ‘;;‘
  reorder(Seq(1,2,5,3,6,4))                        ‘;;‘
  vectorize(32)      ‘@‘ innermost(isApp(isApp(isMap))))
(loopPerm ‘;‘ lowerToC)(mm)
```



based on Lift
[ICFP 2015] by Steuwer et. al.

based on Stratego
[ICFP 1998] by Visser et. al.

ELEVATE **A Language for Describing Optimisation Strategies**

- A **Strategy** encodes a program transformation as a function:

```
type Strategy[P] = P ⇒ RewriteResult[P]
```

- A **RewriteResult** encodes its success or failure:

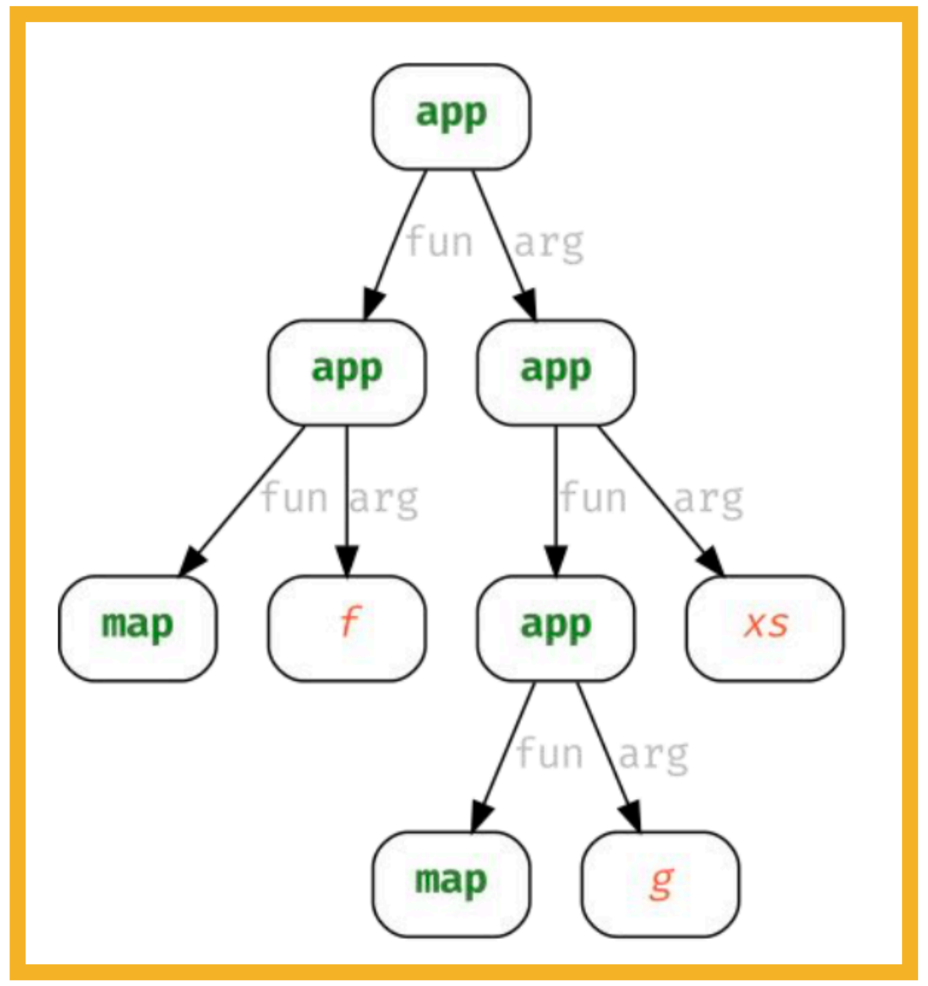
```
RewriteResult[P] = Success[P](p: P)
                 | Failure[P](s: Strategy[P])
```

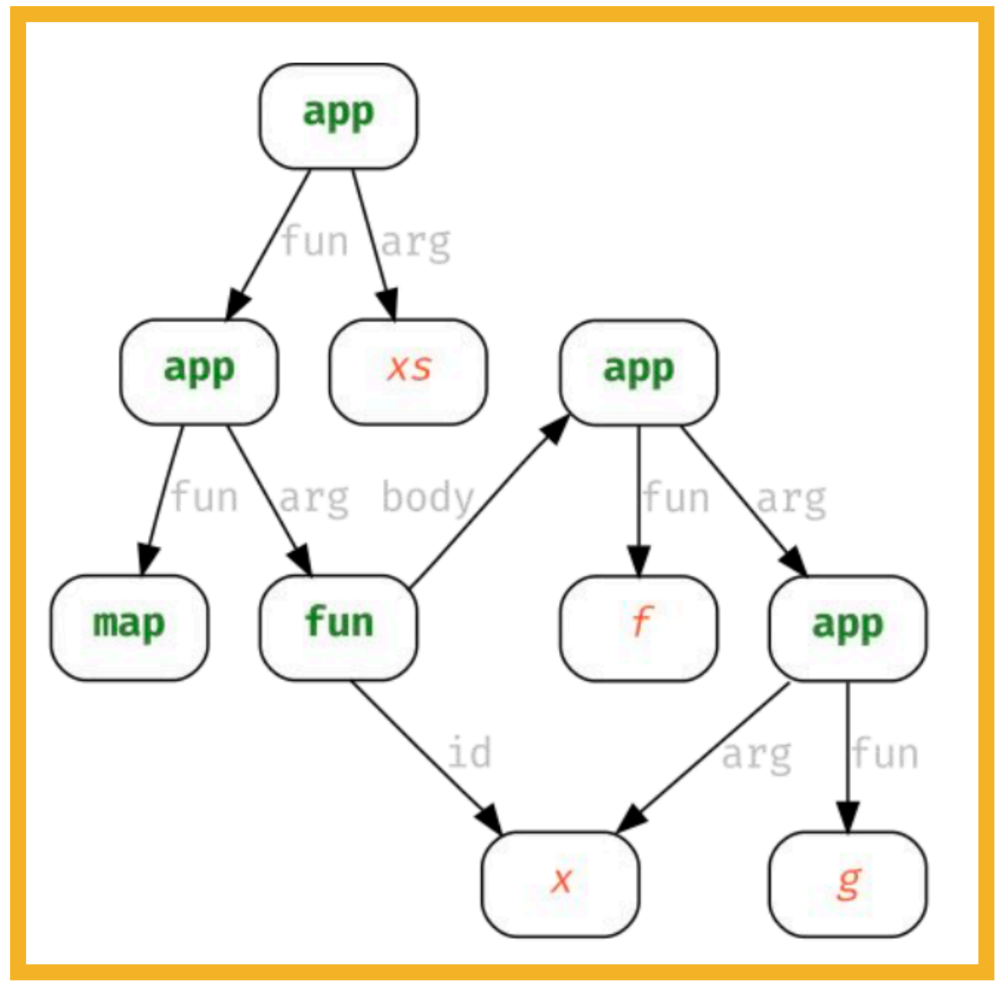
Rewrite Rules in ELEVATE

- *Rewrite rules* are basic strategies

$$\mathbf{map}(f) \ll \mathbf{map}(g) \rightsquigarrow \mathbf{map}(f \ll g)$$

```
def mapFusion: Strategy[Rise] =
  (p: Rise) ⇒ p match {
    case app(app(map, f),
             app(app(map, g), xs)) =
      Success( map(fun(x ⇒ f(g(x))), xs) )
    case _ = Failure(mapFusion)
}
```

mapFusion(

) = 

Combinators in ELEVATE

- Building more complex strategies from simpler once

- Sequential Composition (;)

```
def seq[P]: Strategy[P] ⇒ Strategy[P] ⇒ Strategy[P] =
        fs ⇒ ss ⇒ p ⇒ fs(p).flatMapSuccess(ss)
```

- Left Choice (<+)

```
def lChoice[P]: Strategy[P] ⇒ Strategy[P] ⇒ Strategy[P] =
        fs ⇒ ss ⇒ p ⇒ fs(p).flatMapFailure(_ ⇒ ss(p))
```

- Try

```
def try[P]: Strategy[P] ⇒ Strategy[P] =
        s ⇒ p ⇒ (s <+ id)(p)
```

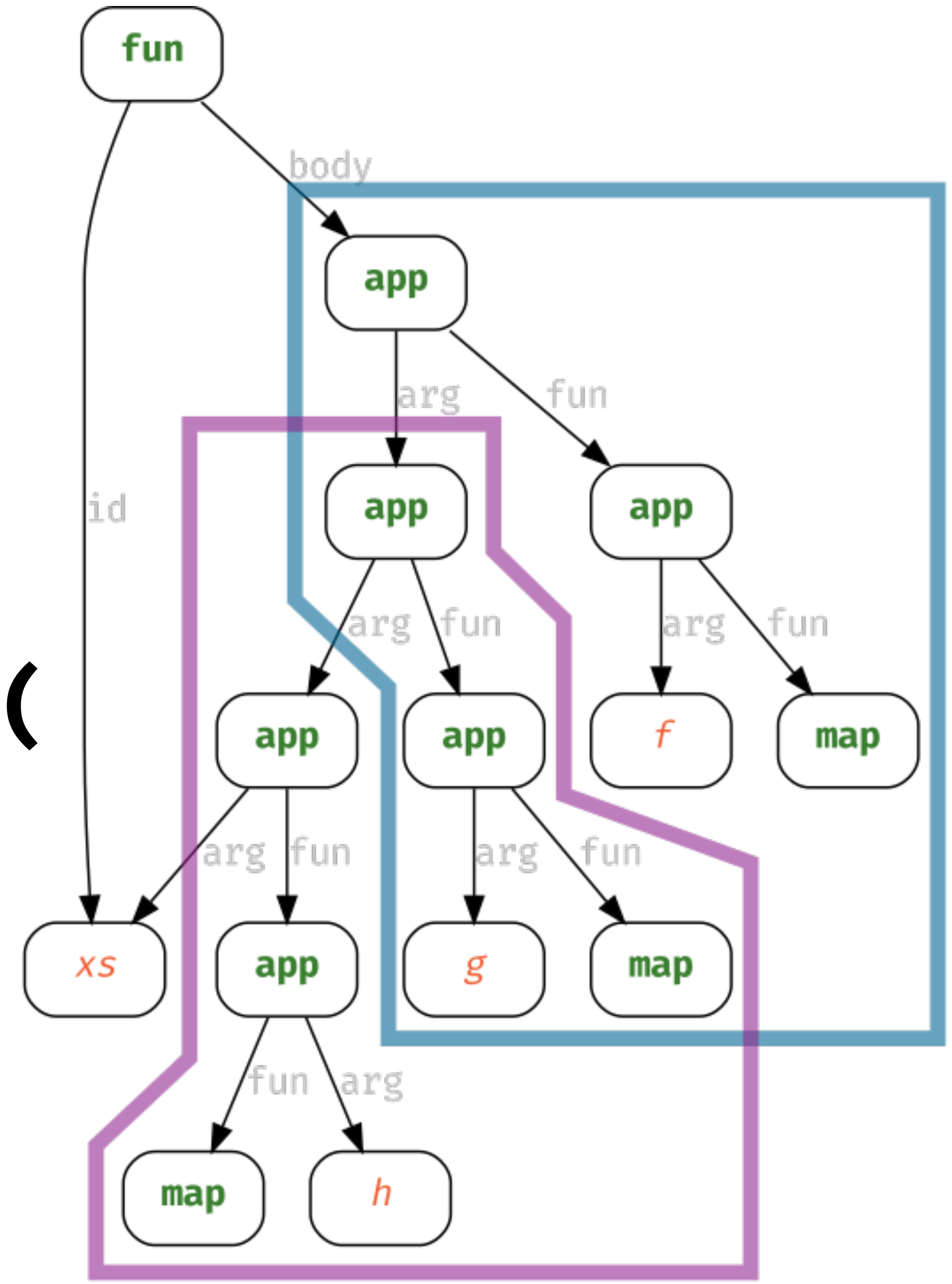
- Repeat

```
def repeat[P]: Strategy[P] ⇒ Strategy[P] =
        s ⇒ p ⇒ try(s ; repeat(s))(p)
```

Traversals in ELEVATE

- Describing Precise Locations

`mapFusion (`



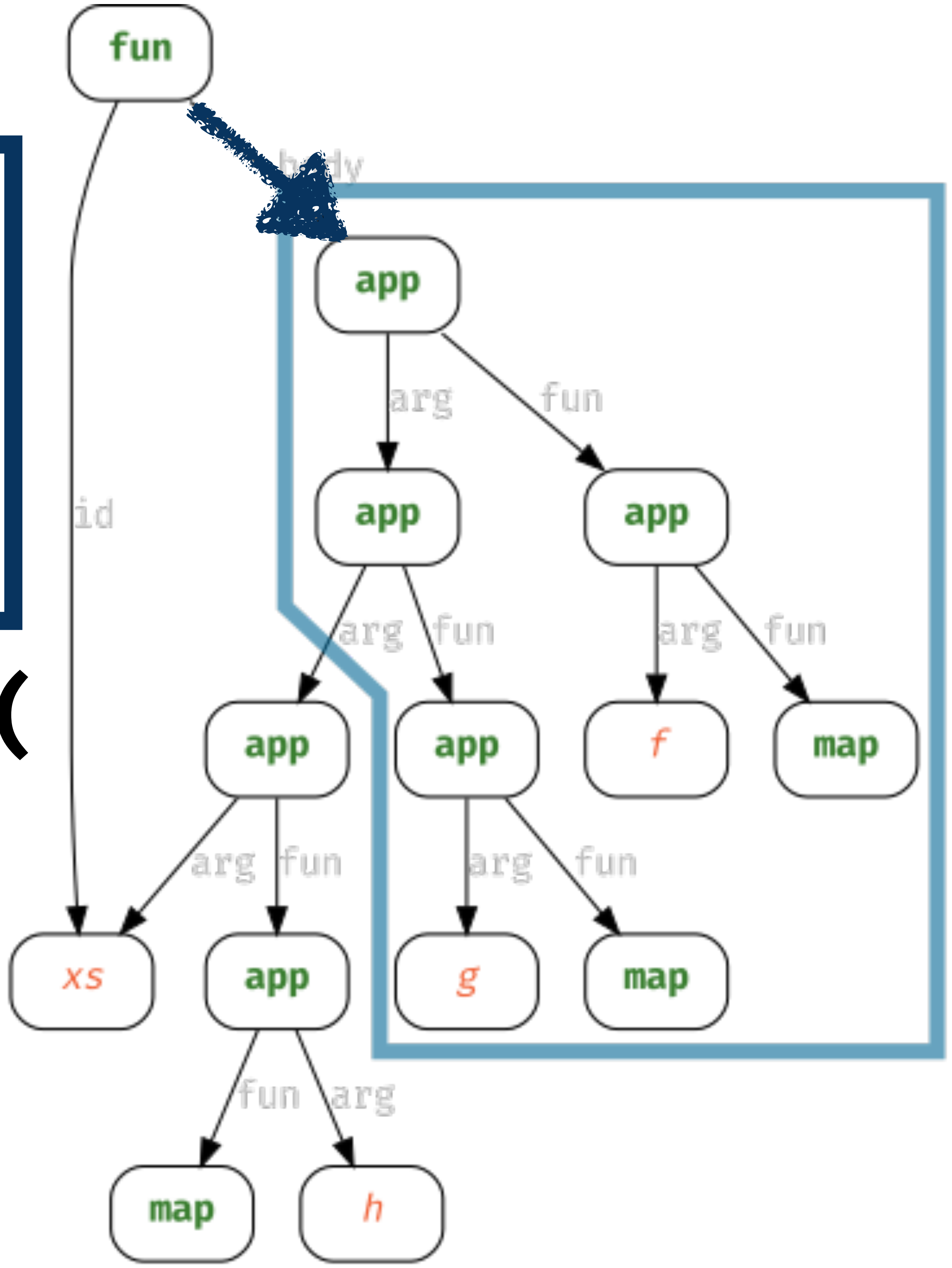
`) = ?`

```
threemaps =fun(xs ⇒ map(f)(map(g)(map(h)(xs))))
```

Traversals in ELEVATE

- Describing Precise Locations

```
def body: Strategy[Rise] => Strategy[Rise] =
 s => p => p match {
  case fun(x,b) => s(b).mapSuccess(nb =>
fun(x,nb))
  case _ => Failure( body(s) )
}
```

`body(mapFusion) (`  `) = ?`

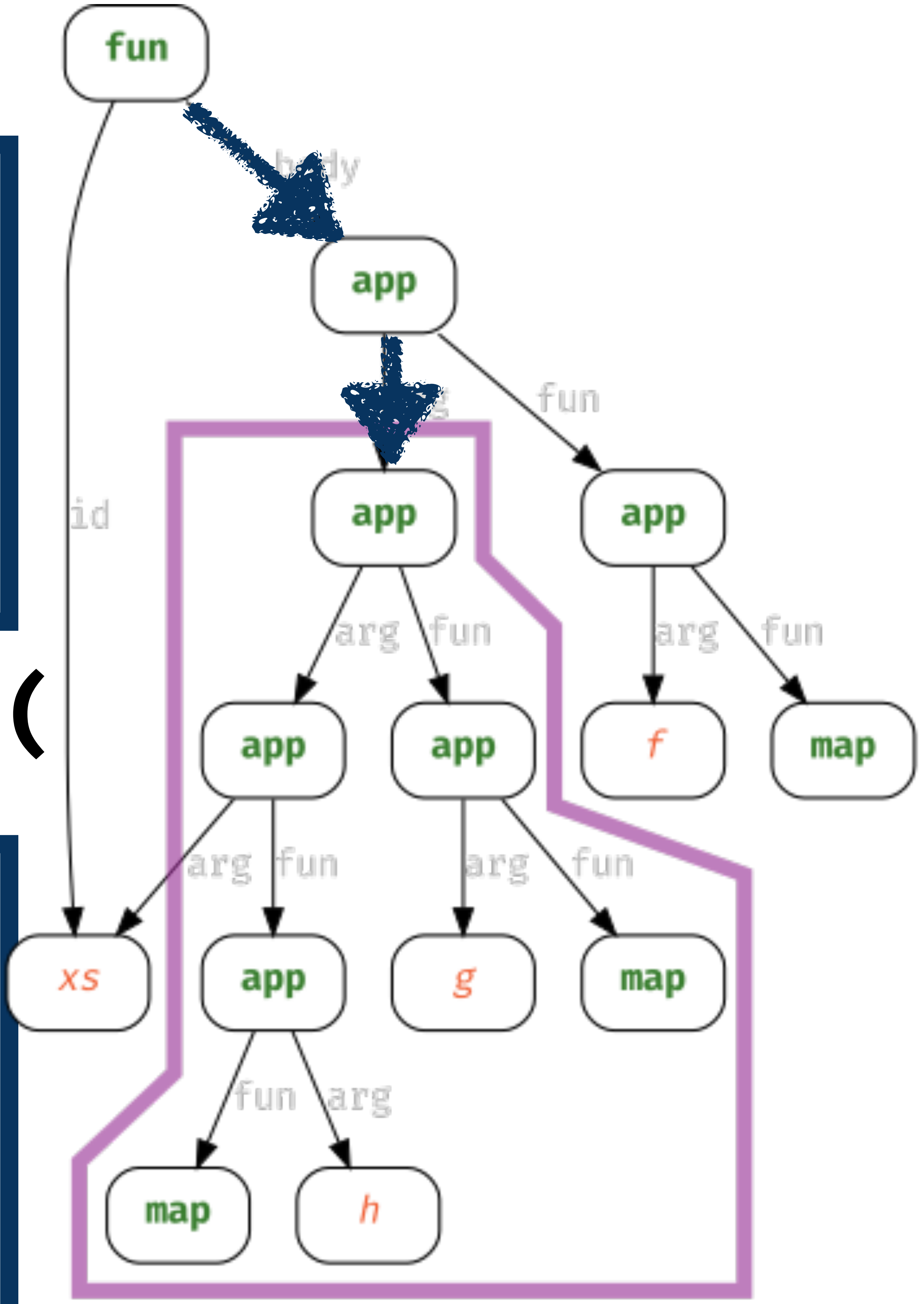
`threemaps = fun(xs, map(f)(map(g)(map(h)(xs))))`

Traversals in ELEVATE

- Describing Precise Locations

```
def body: Strategy[Rise] ⇒ Strategy[Rise] =
 s ⇒ p ⇒ p match {
  case fun(x,b) ⇒ s(b).mapSuccess(nb ⇒
fun(x,nb))
  case _ ⇒ Failure( body(s) )
}
```

```
body(argument(mapFusion)) (
```




```
) = ?
```

```
def argument: Strategy[Rise] ⇒ Strategy[Rise] =
 s ⇒ p ⇒ p match {
  case app(f,a) ⇒ s(a).mapSuccess(na ⇒
app(f,na))
  case _ ⇒ Failure( argument(s) )
}
```

```
threemaps = fun(xs, map(f)(map(g)(map(h)(xs))))
```

Complex Traversals + Normalization in ELEVATE

- With three basic generic traversals

```
type Traversal[P] = Strategy[P] => Strategy[P]
def all[P]: Traversal[P];    def one[P]: Traversal[P];    def some[P]: Traversal[P]
```

- we define more complex traversals:

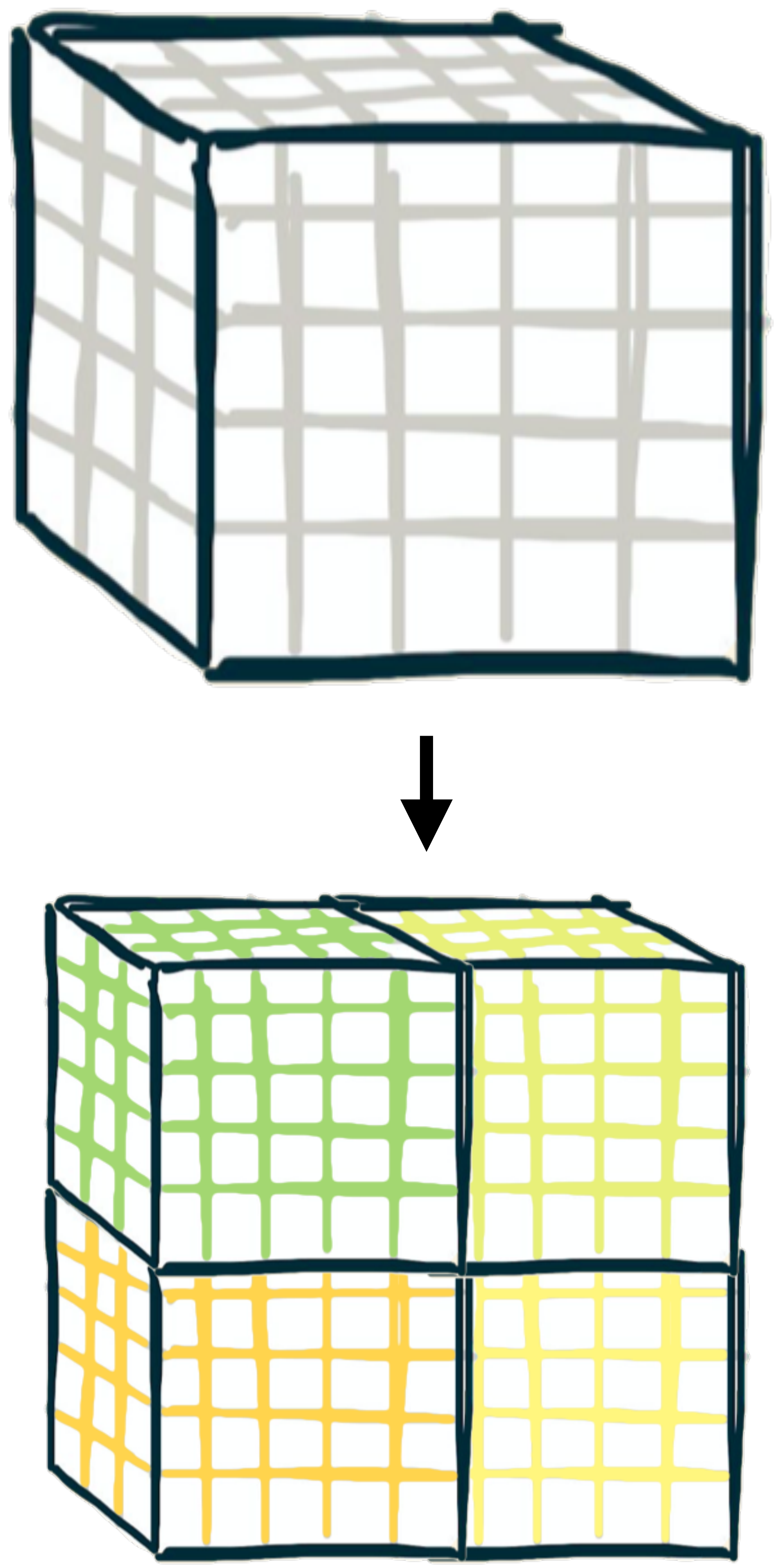
```
def      topDown[P]: Traversal[P] = s => p => (s <+ one(topDown(s)))(p)
def     bottomUp[P]: Traversal[P] = s => p => (one(bottomUp(s)) <+ s)(p)
def   allTopDown[P]: Traversal[P] = s => p => (s ‘;‘ all(allTopDown(s)))(p)
def allBottomUp[P]: Traversal[P] = s => p => (all(allBottomUp(s)) ‘;‘ s)(p)
def       tryAll[P]: Traversal[P] = s => p => (all(tryAll(try(s))) ‘;‘ try(s))(p)
```

- With these traversals we define normal forms, e.g. $\beta\eta$-normal-form:

```
def normalize[P]: Strategy[P] => Strategy[P] = s => p => repeat(topDown(s))(p)

def BENF = normalize(betaReduction <+ etaReduction)
```

Complex optimisations defined as strategies

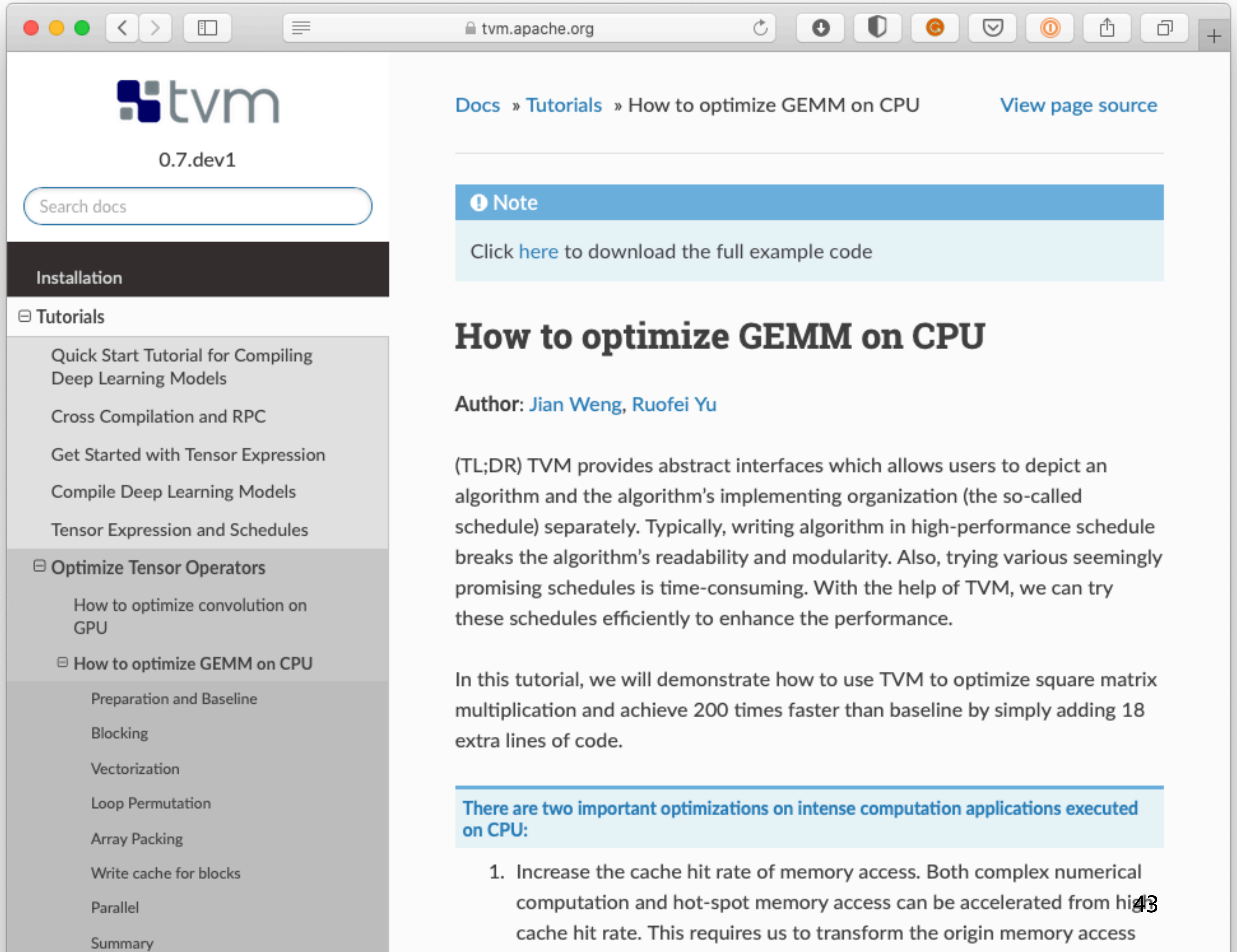


```
def tile: Int → Int → Strategy =
  (dim) ⇒ (n) ⇒ dim match {
    case 1 = function(splitJoin(n))
    case 2 = fmap(function(splitJoin(n))) ;
             function(splitJoin(n)) ; interchange(2)
    case i = fmap(tile(dim-1, n)) ;
             function(splitJoin(n)) ; interchange(n)
  }
```

Tiling defined as composition of rewrites not a built-in!

Case Study: Implementing TVM's Scheduling API

- We attempt to express the same optimizations described in the TVM tutorial:


tvm.apache.org

tvm
0.7.dev1

Search docs

Installation
Tutorials
Quick Start Tutorial for Compiling Deep Learning Models
Cross Compilation and RPC
Get Started with Tensor Expression
Compile Deep Learning Models
Tensor Expression and Schedules
Optimize Tensor Operators
How to optimize convolution on GPU
How to optimize GEMM on CPU
Preparation and Baseline
Blocking
Vectorization
Loop Permutation
Array Packing
Write cache for blocks
Parallel
Summary

Docs » Tutorials » How to optimize GEMM on CPU — View page source

Note

Click here to download the full example code

How to optimize GEMM on CPU

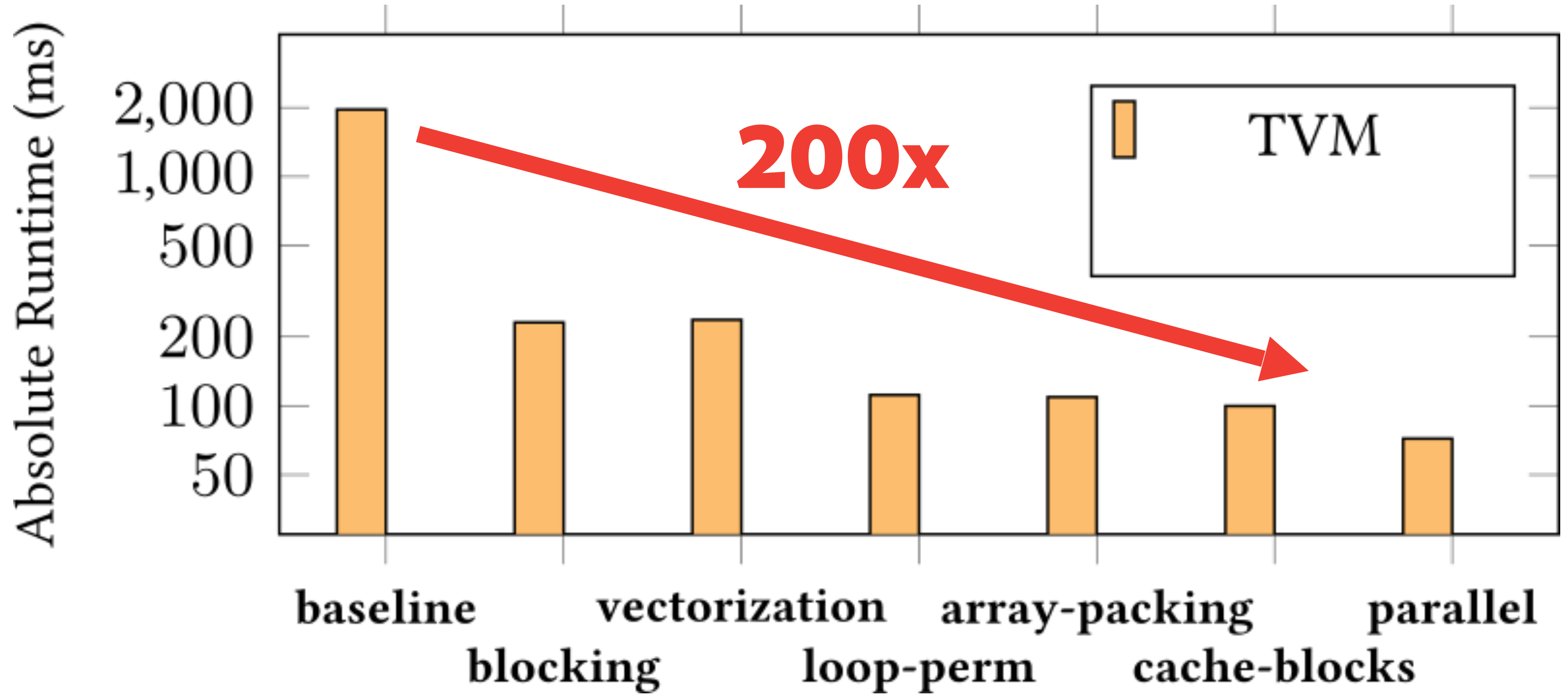
Author: Jian Weng, Ruofei Yu

(TL;DR) TVM provides abstract interfaces which allows users to depict an algorithm and the algorithm's implementing organization (the so-called schedule) separately. Typically, writing algorithm in high-performance schedule breaks the algorithm's readability and modularity. Also, trying various seemingly promising schedules is time-consuming. With the help of TVM, we can try these schedules efficiently to enhance the performance.

In this tutorial, we will demonstrate how to use TVM to optimize square matrix multiplication and achieve 200 times faster than baseline by simply adding 18 extra lines of code.

There are two important optimizations on intense computation applications executed on CPU:

1. Increase the cache hit rate of memory access. Both complex numerical computation and hot-spot memory access can be accelerated from high cache hit rate. This requires us to transform the origin memory access



Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

RISE

```
// matrix multiplication in RISE
val dot = fun(as, fun(bs, zip(as)(bs) |>
  map(fun(ab, mult(fst(ab))(snd(ab)))) |>
  reduce(add)(0) ) )
val mm = fun(a, fun(b, a |>
  map( fun(arow, transpose(b) |>
    map( fun(bcol,
      dot(arow)(bcol) )))) ))
```

```
// baseline strategy in ELEVATE
val baseline = ( DFNF ‘;‘
  fuseReduceMap ‘@‘ topDown )
(baseline ‘;‘ lowerToC)(mm)
```

ELEVATE

tvm

```
# Naive matrix multiplication algorithm
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N),lambda x, y:
    tvm.sum(A[x, k] * B[k, y],
    axis=k),name='C')





# TVM default schedule
s = tvm.create_schedule(C.op)
```

Baseline Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Clear separation of concerns

RISE

```
// matrix multiplication in RISE
val dot = fun(as, fun(bs, zip(as)(bs) |>
  map(fun(ab, mult(fst(ab))(snd(ab)))) |>
  reduce(add)(0) ) )
val mm = fun(a, fun(b, a |>
  map( fun(arow, transpose(b) |>
    map( fun(bcol,
      dot(arow)(bcol) )))) ))
```

```
// baseline strategy in ELEVATE
val baseline = ( DFNF ‘;‘
  fuseReduceMap ‘@‘ topDown )
(baseline ‘;‘ lowerToC)(mm)
```

ELEVATE

Be explicit

Enable composability

tvm

```
# Naive matrix multiplication algorithm
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N),lambda x, y:
      tvm.sum(A[x, k] * B[k, y],
      axis=k),name='C')




# TVM default schedule
s = tvm.create_schedule(C.op)
```

Implicit behavior

Baseline Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

ELEVATE

tvm

```
val loopPerm = (
 tile(32,32)        `@` outermost(mapNest(2))     `;;`
 fissionReduceMap `@` outermost(appliedReduce) `;;`
 split(4)           `@` innermost(appliedReduce) `;;`
 reorder(Seq(1,2,5,3,6,4))                        `;;`
 vectorize(32)      `@` innermost(isApp(isApp(isMap))))
(loopPerm `;` lowerToC)(mm)
```

```
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0],C.op.axis[1],32,32)
k,              = s[C].op.reduce_axis
ko, ki          = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi)
```

Loop Permutation with blocking Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE **Strategies**

ELEVATE

Facilitate reuse

User-defined vs. build in

tvm

```
val loopPerm = (
 tile(32,32)       ‘@‘ outermost(mapNest(2))    ‘;;‘
 fissionReduceMap ‘@‘ outermost(appliedReduce) ‘;;‘
 split(4)          ‘@‘ innermost(appliedReduce) ‘;;‘
 reorder(Seq(1,2,5,3,6,4))                      ‘;;‘
 vectorize(32)     ‘@‘ innermost(isApp(isApp(isMap))))
(loopPerm ‘;‘ lowerToC)(mm)
```

```
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0],C.op.axis[1],32,32)
k,              = s[C].op.reduce_axis
ko, ki          = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi)
```

No clear separation
of concerns

Loop Permutation with blocking Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

ELEVATE

```
val appliedMap = isApp(isApp(isMap))
val isTransposedB = isApp(isTranspose)

val packB = storeInMemory(isTransposedB,
 permuteB ‘;;‘
 vectorize(32) ‘@‘ innermost(appliedMap) ‘;;‘
 parallel      ‘@‘ outermost(isMap)
) ‘@‘ inLambda

val arrayPacking = packB ‘;;‘ loopPerm
(arrayPacking ‘;‘ lowerToC )(mm)
```

tvm

```
# Modified algorithm
bn = 32
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
pB = tvm.compute((N / bn, K, bn),
  lambda x, y, z: B[y, x * bn + z], name='pB')
C = tvm.compute((M,N), lambda x,y:
  tvm.sum(A[x,k] * pB[y//bn,k,
  tvm.indexmod(y,bn)], axis=k),name='C')
# Array packing schedule
s = tvm.create_schedule(C.op)
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)
k,              = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi)
x, y, z         = s[pB].op.axis
s[pB].vectorize(z)
s[pB].parallel(x)
```

Array Packing Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Clear separation of concerns vs No clear separation of concerns

ELEVATE

```
val appliedMap = isApp(isApp(isMap))
val isTransposedB = isApp(isTranspose)

val packB = storeInMemory(isTransposedB,
 permuteB `;;`
 vectorize(32) `@` innermost(appliedMap) `;;`
 parallel      `@` outermost(isMap)
) `@` inLambda

val arrayPacking = packB `;;` loopPerm
(arrayPacking `;` lowerToC )(mm)
```

Facilitate reuse

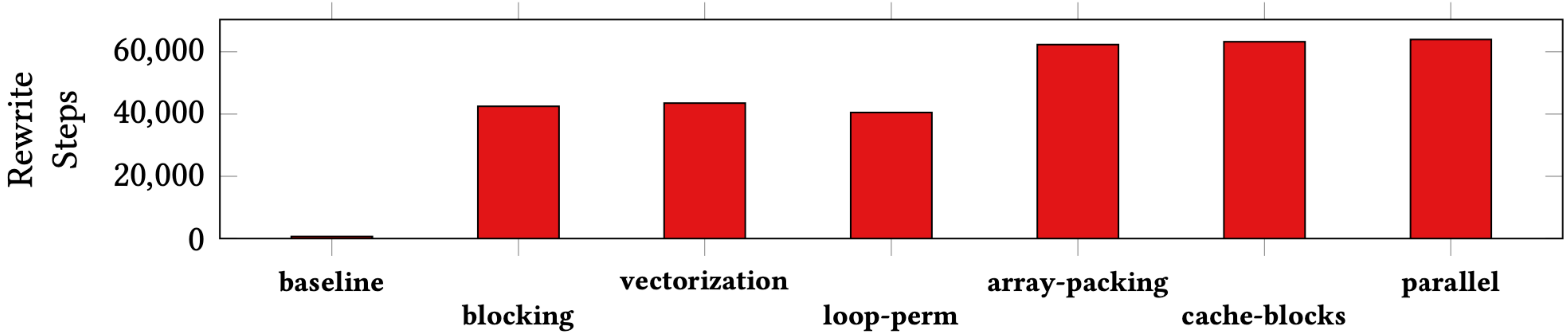
tvm

```
# Modified algorithm
bn = 32
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
pB = tvm.compute((N / bn, K, bn),
  lambda x, y, z: B[y, x * bn + z], name='pB')
C = tvm.compute((M,N), lambda x,y:
  tvm.sum(A[x,k] * pB[y//bn,k,
  tvm.indexmod(y,bn)], axis=k),name='C')
# Array packing schedule
s = tvm.create_schedule(C.op)
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)
k,              = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi)
x, y, z         = s[pB].op.axis
s[pB].vectorize(z)
s[pB].parallel(x)
```

Array Packing Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Number of successful rewrite steps

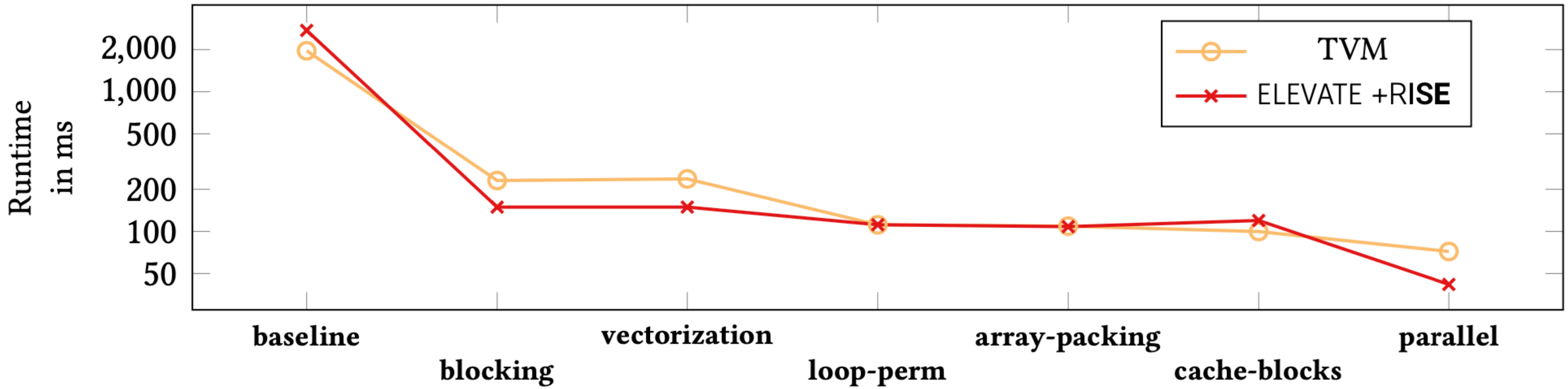


Rewriting took less than 2 seconds with our unoptimised implementation

Rewrite based approach scales to complex optimizations

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Performance of generated code



Competitive performance compared to TVM compiler

Tradeoffs when optimizing with rewriting

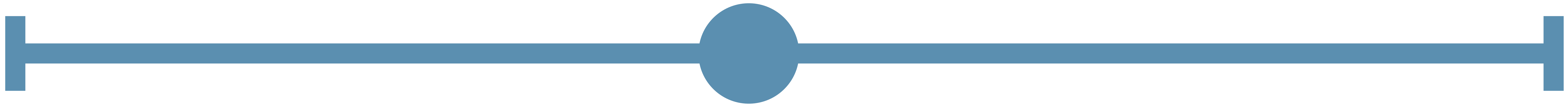
Automatic rewriting

Manual rewriting

- ✓ No human needed in optimization process
- ✗ Costly & Lengthy search process
- ✗ Does not (yet) scale to all programs

- Extensive human effort needed ✗
- Expert is in control, no search required ✓
- **Strategies are too sensitive ⇒ don't scale across applications** ✗

Tradeoffs when optimizing with rewriting



Automatic rewriting → ? ← Manual rewriting

✓ No human needed in optimization process

✗ Costly & Lengthy search process

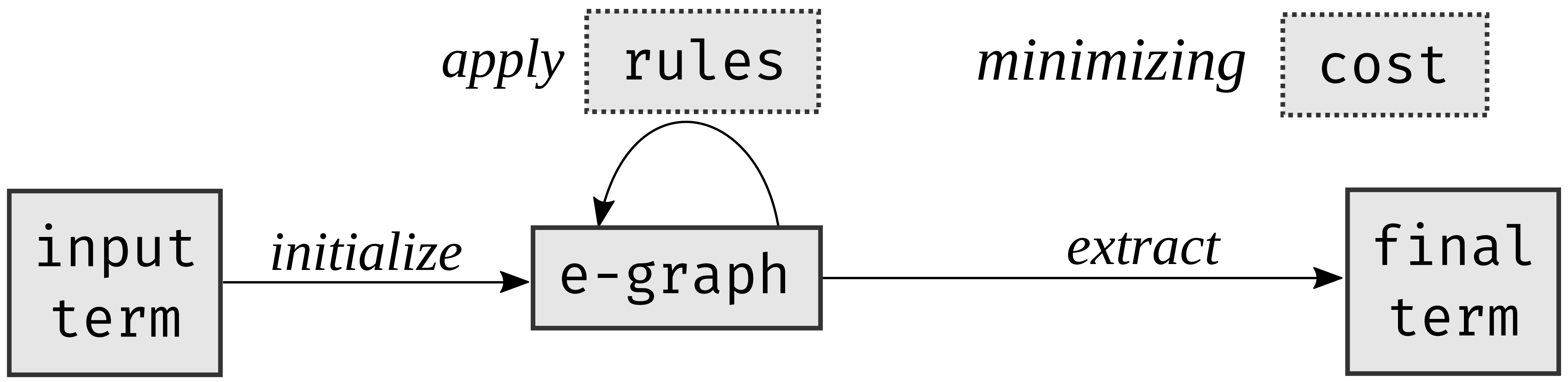
✗ Does not (yet) scale to all programs

Extensive human effort needed ✗

Human is in control, no search required ✓

Strategies are too sensitive
$\Rightarrow$ don't scale across applications ✗

Equality Saturation

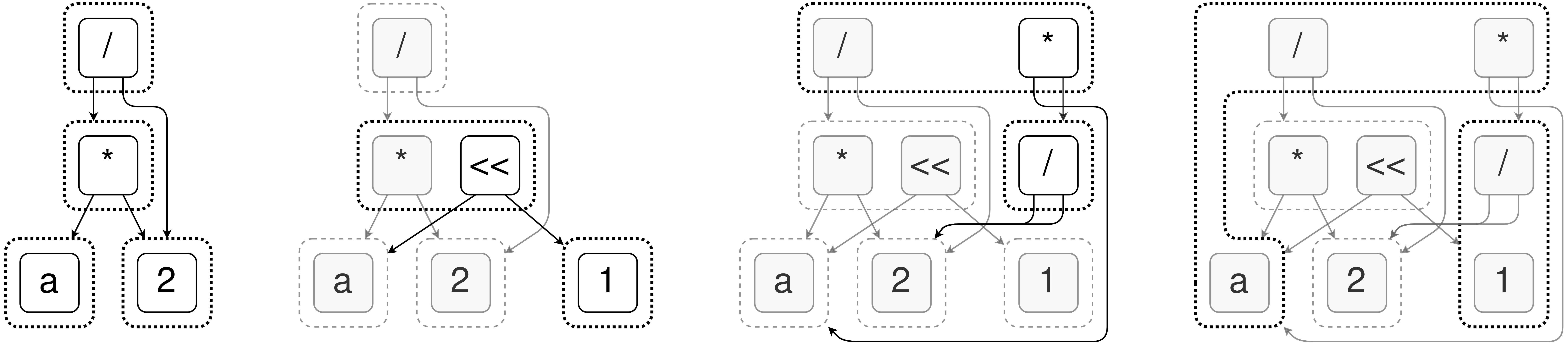


- Optimize programs by efficiently exploring many possible rewrites
- Many successful applications sparked from the recent `egg` library

E-Graph

Expression

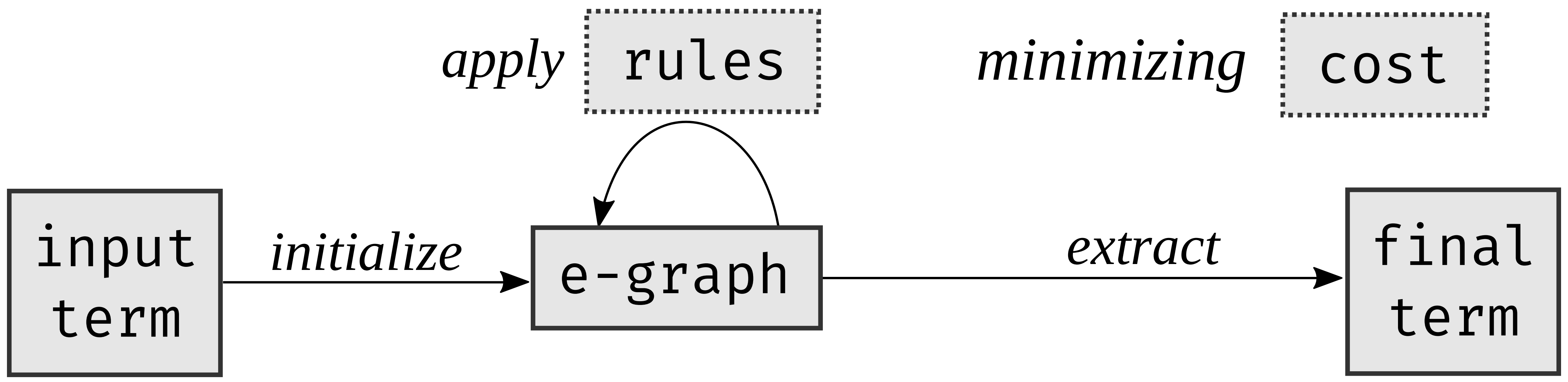
`(a*2)/2`



$x*2 \rightarrow x{<<}1$ $\qquad$ $(x*y)/z \rightarrow x*(y/z)$ $\qquad$ $x/x \rightarrow 1$, $1*x \rightarrow x$

After applying Rewrites

Equality Saturation



- Optimize programs by efficiently exploring many possible rewrites
- Many successful applications sparked from the recent `egg` library

Some optimizations remain out of reach as the e-graph grows too big

Case Study

Matrix Multiplication Optimizations for CPU:

- transform loops

  blocking, permutation, unrolling
- change data layout
- add parallelism

  vectorization, multi-threading

Case Study

Matrix Multiplication Optimizations for CPU:

- transform loops

  blocking, permutation, unrolling
- change data layout
- add parallelism

  vectorization, multi-threading

Space of equivalent programs to consider is huge

Case Study

- Rewritten language: RISE, a functional array language

Matrix Multiplication in RISE:

```
def mm a b =
  map (λaRow.                   | for aRow in a:
    map (λbCol.                 |   for bCol in transpose(b):
      dot aRow bCol)            |     ... = dot(aRow, bCol)
      (transpose b)) a

def dot xs ys =
  reduce + 0                    | for (x, y) in zip(xs, ys):
    (map (λ(x, y). x × y)       |   acc += x × y
      (zip xs ys))
```

Case Study

- Rewritten language: RISE, a functional array language

Matrix Multiplication in RISE:

```
def mm a b =
  map (λaRow.                 | for aRow in a:
    map (λbCol.               |   for bCol in transpose(b):
      dot aRow bCol)          |     ... = dot(aRow, bCol)
      (transpose b)) a

def dot xs ys =
  reduce + 0                  | for (x, y) in zip(xs, ys):
    (map (λ(x, y). x × y)     |   acc += x × y
      (zip xs ys))
```

RISE is designed for optimization via term rewriting

Case Study

- Achieve the same 7 optimization goals with equality saturation?[1]

goal	found?	runtime	RAM
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
vectorization	✗	>1h	>60 GB
loop-perm	✗	>1h	>60 GB
array-packing	✗	35mn	>60 GB
cache-blocks	✗	35mn	>60 GB
parallel	✗	35mn	>60 GB

- Most goals are not found before exhausting 60 GB.
- For comparison, rewriting strategies take <2s and <1GB.

[1] on Intel Xeon E5-2640 v2

Case Study

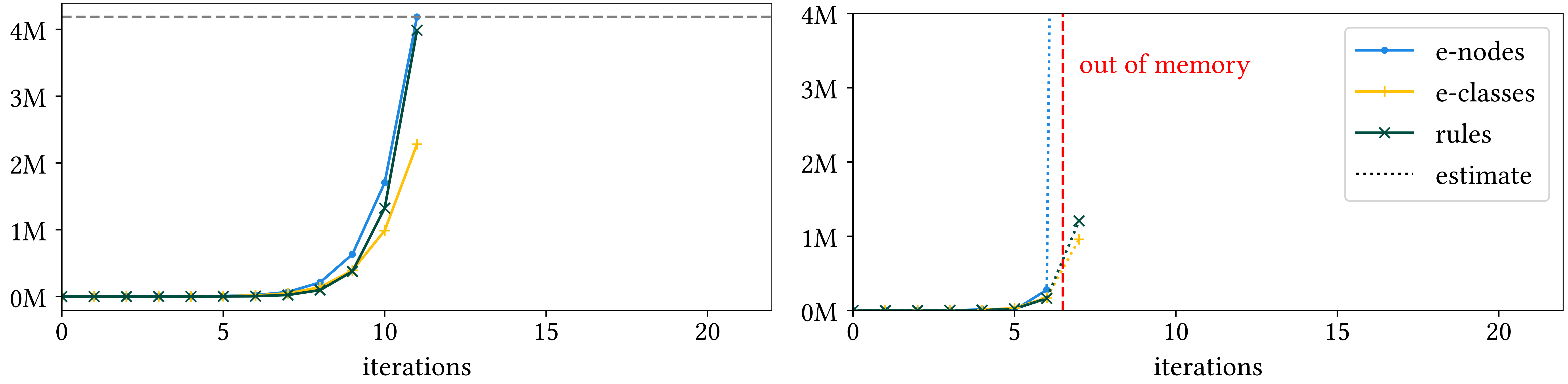
- Achieve the same 7 optimization goals with equality saturation?[1]

goal	found?	runtime	RAM
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
vectorization	✗	>1h	>60 GB
loop-perm	✗	>1h	>60 GB
array-packing	✗	35mn	>60 GB
cache-blocks	✗	35mn	>60 GB
parallel	✗	35mn	>60 GB

Standard equality saturation does not scale to this optimization space

[1] on Intel Xeon E5-2640 v2

E-Graph Evolution



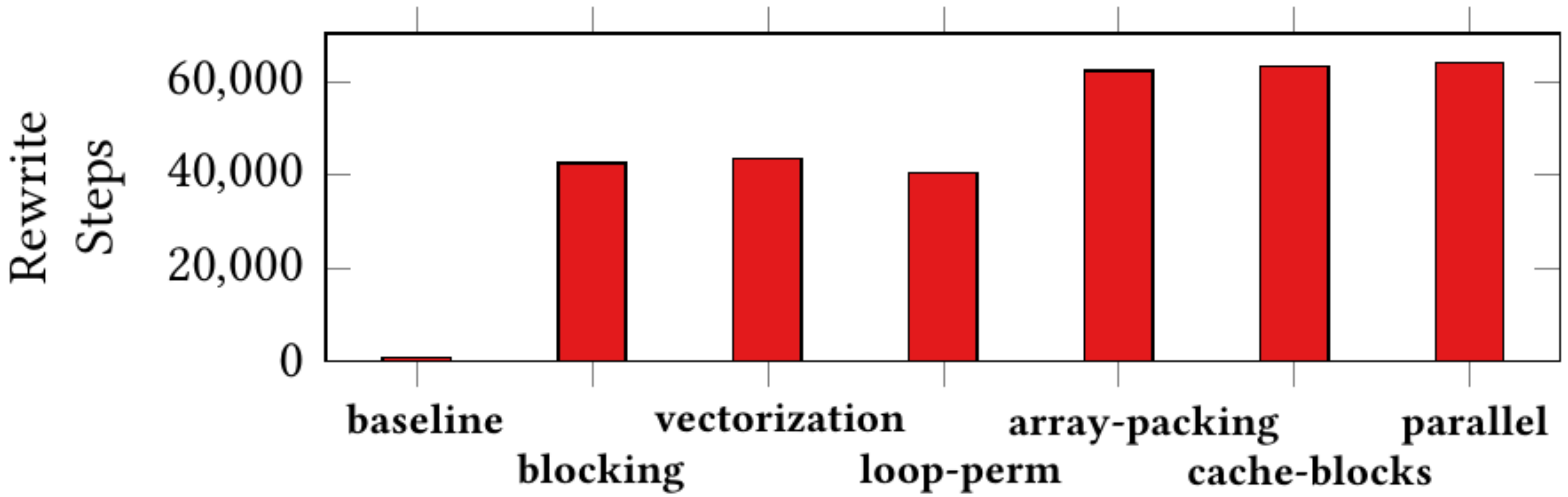
(a) *blocking*, found: ✓ (b) *parallel*, found: ✗

Two difficulties:

1. Long rewrite sequences $\implies$ many iterations are required
2. Explosive combination of rewrite rules $\implies$ exponential growth
   - millions of e-nodes and e-classes in less than 10 iterations
   - worse for *parallel*, memory is exhausted in the 7th iteration

Difficulty 1. Long Rewrite Sequences

Prior work *(not shortest path)*:



Difficulty 2. Explosive Combinations of Rewrite Rules

Two example rules that quickly generate many possibilities:

split-join:

```
    map f x                      | for m:
                                 |  ... = f(...)
      ↦
join
  (map                           | for m / n:
    (map f)                      |  for n:
    (split n x))                 |   ... = f(...)
```

transpose-around-map-map:

```
    map                          | for m:
      (map f) x                  |  for n:
                                 |   ... = f(...)
      ↦
transpose
  (map                           | for n:
    (map f)                      |  for m:
    (transpose x))               |   ... = f(...)
```

To overcome these difficulties, we came up with *sketch-guided equality saturation*

Sketch-Guided Equality Saturation

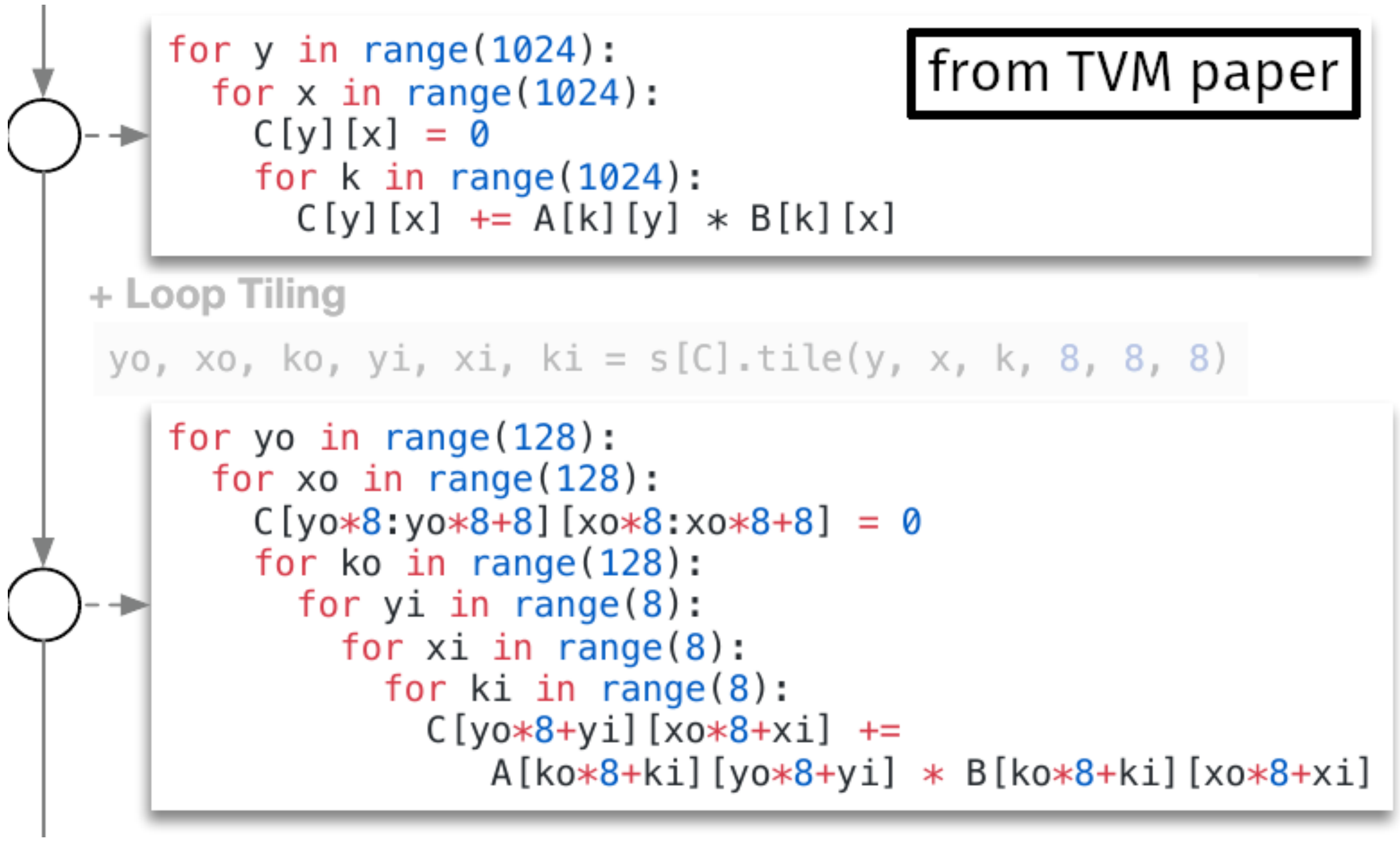
Observation:

- The *shape* of the optimised program is often used to explain optimizations:

```
for m:
 for n:
  for k:
   ..
```

$\mapsto^*$

```
for m / 32:
 for n / 32:
  for k / 4:
   for 4:
    for 32:
     for 32:
      ..
```



Sketch-Guided Equality Saturation

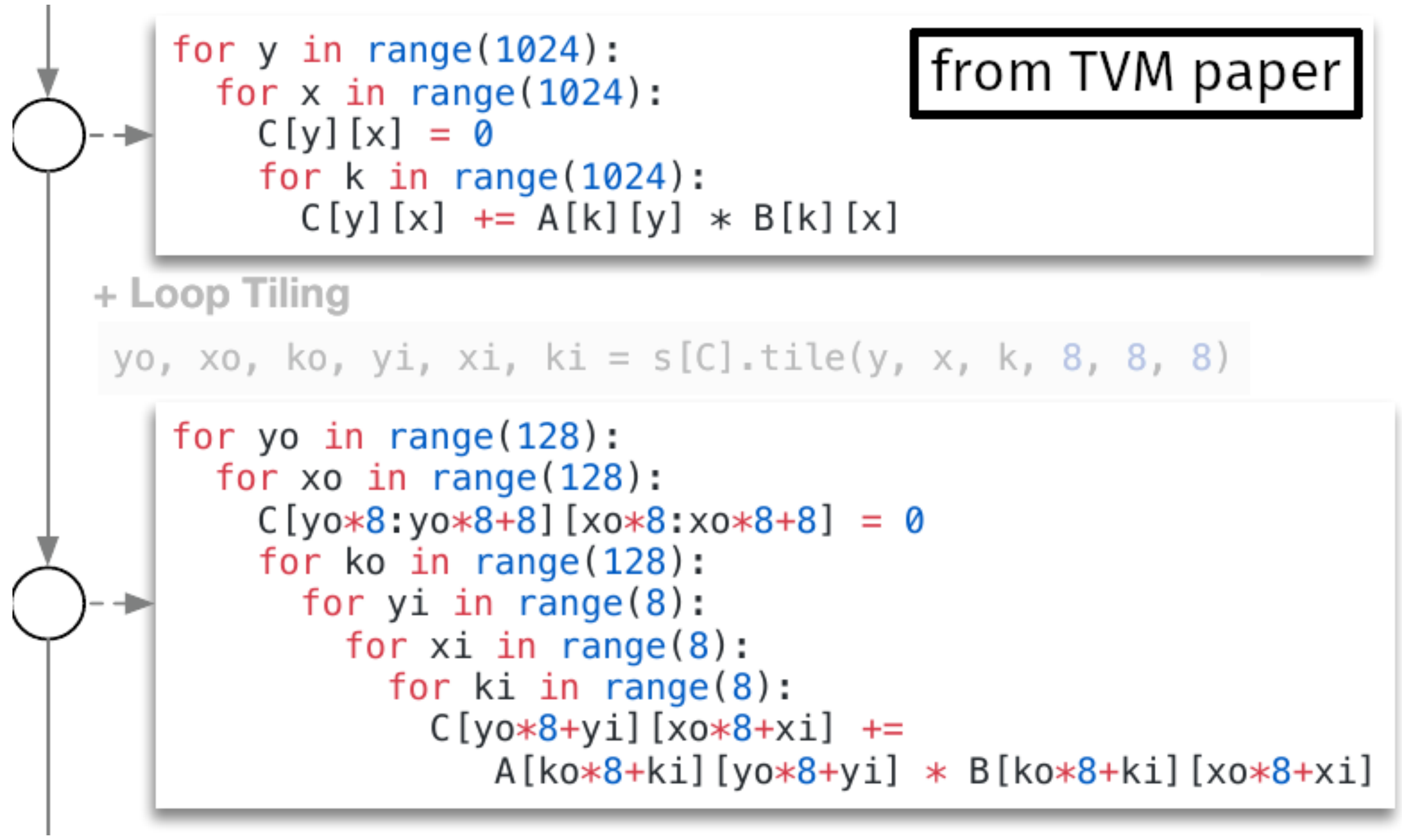
Observation:

- The *shape* of the optimised program is often used to explain optimizations:

```
for m:
 for n:
  for k:
   ..
```

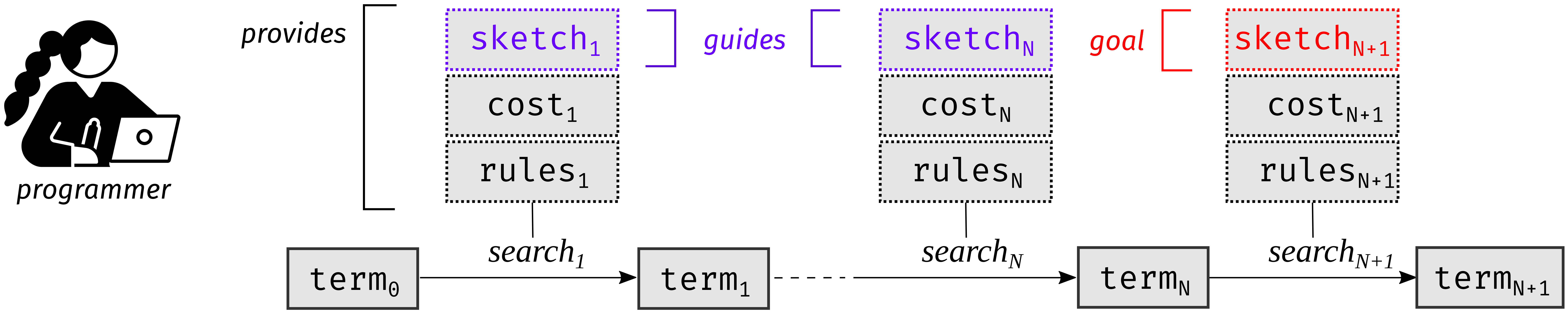
$\mapsto^*$

```
for m / 32:
 for n / 32:
  for k / 4:
   for 4:
    for 32:
     for 32:
      ..
```



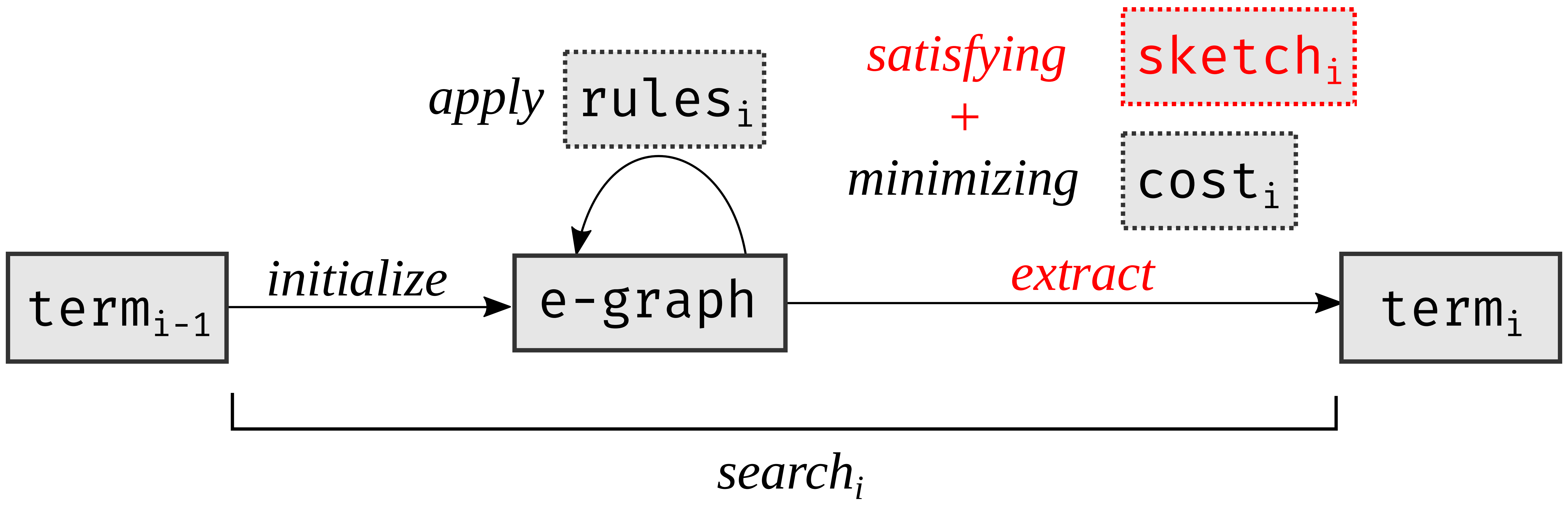
Explanatory shapes can be formalized as sketches and used to guide rewriting

Sketch-Guided Equality Saturation



- Factors an unfeasible search into a sequence of feasible ones:
    1. Break long rewrite sequences
    2. Isolate explosive combinations of rewrite rules

Sketch-Satisfying Equality Saturation



- Terminates as soon as a program satisfying the sketch is found

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                  | for m:
 containsMap(n,                 |  for n:
  containsReduceSeq(k,          |   for k:
   containsAddMul)))            |     .. + .. × ..
```

- Abstractions defined in terms of smaller building blocks:

```
def containsAddMul: Sketch =
  contains(app(app(+, ?), contains(×)))
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                   | for m:
 containsMap(n,                  |  for n:
  containsReduceSeq(k,           |   for k:
   containsAddMul)))             |     .. + .. × ..
```

- A sketch `s` is satisfied by a set of terms `R(s)`:

```
def containsAddMul: Sketch =
  contains(app(app(+, ?), contains(×)))

R(containsAddMul) = { R(app(app(+, ?), contains(×))) } ∪
  { F(t₁, .., tₙ) | ∃tᵢ ∈ R(containsAddMul) }
R(app(app(+, ?), contains(×))) = { app(app(+, t₁), t₂) | t₂ ∈ R(contains(×)) }
R(contains(×)) = { × } ∪ { F(t₁, .., tₙ) | ∃tᵢ ∈ R(contains(×)) }
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                   | for m:
 containsMap(n,                  |  for n:
  containsReduceSeq(k,           |   for k:
   containsAddMul)))             |     .. + .. × ..
```

blocking sketch:

```
containsMap(m / 32,              | for m / 32:
 containsMap(n / 32,             |  for n / 32:
  containsReduceSeq(k / 4,       |   for k / 4:
   containsReduceSeq(4,          |    for 4:
    containsMap(32,              |     for 32:
     containsMap(32,             |      for 32:
      containsAddMul))))))       |       .. + .. × ..
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                        | for m:
 containsMap(n,                       |  for n:
  containsReduceSeq(k,                |   for k:
   containsAddMul)))                  |    .. + .. × ..
```

sketch guide:

how to split the loops before reordering them?

```
containsMap(m / 32,                   | for m / 32:
 containsMap(32,                      |  for 32:
  containsMap(n / 32,                 |   for n / 32:
   containsMap(32,                    |    for 32:
    containsReduceSeq(k / 4,          |     for k / 4:
     containsReduceSeq(4,             |      for 4:
      containsAddMul))))))            |       .. + .. × ..
```

blocking sketch:

```
containsMap(m / 32,                   | for m / 32:
 containsMap(n / 32,                  |  for n / 32:
  containsReduceSeq(k / 4,            |   for k / 4:
   containsReduceSeq(4,               |    for 4:
    containsMap(32,                   |     for 32:
     containsMap(32,                  |      for 32:
      containsAddMul))))))            |       .. + .. × ..
```

Evaluation

- Equality Saturation without Sketch Guides[2]:

goal	**found?**	**runtime**	**RAM**
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

- Sketch-Guided Equality Saturation[3]:

goal	**sketch guides**	**found?**	**runtime**	**RAM**
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

[2] Intel Xeon E5-2640 v2

[3] AMD Ryzen 5 PRO 2500U

Evaluation

- Equality Saturation without Sketch Guides[2]:

goal	**found?**	**runtime**	**RAM**
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

- Sketch-Guided Equality Saturation[3]:

goal	**sketch guides**	**found?**	**runtime**	**RAM**
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

Sketch-guided equality saturation finds all 7 optimization goals

[2] Intel Xeon E5-2640 v2

[3] AMD Ryzen 5 PRO 2500U

Evaluation

- Equality Saturation without Sketch Guides[2]:

goal	found?	runtime	RAM
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

582x

116x

- Sketch-Guided Equality Saturation[3]:

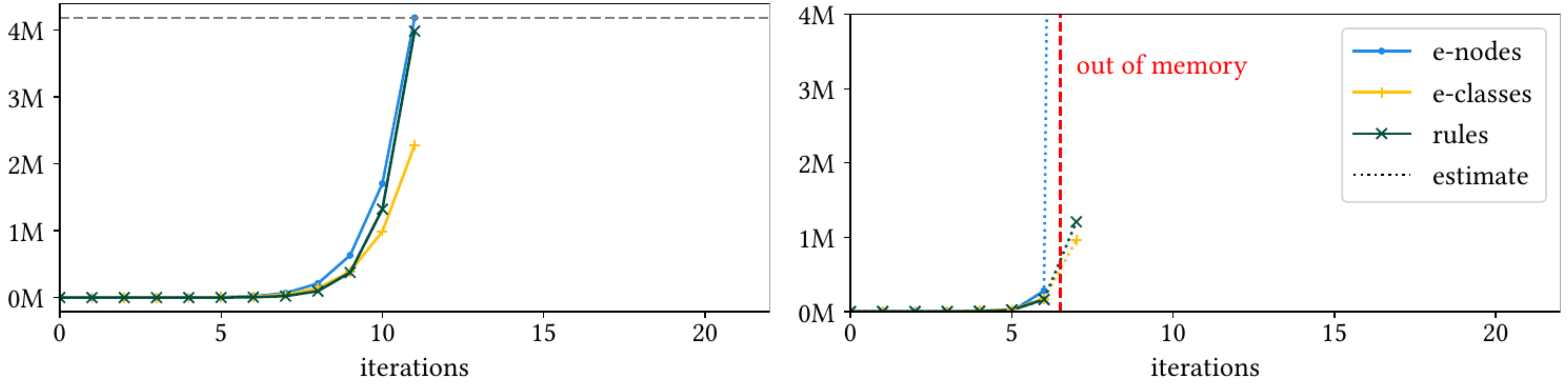
goal	sketch guides	found?	runtime	RAM
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

[2] Intel Xeon E5-2640 v2

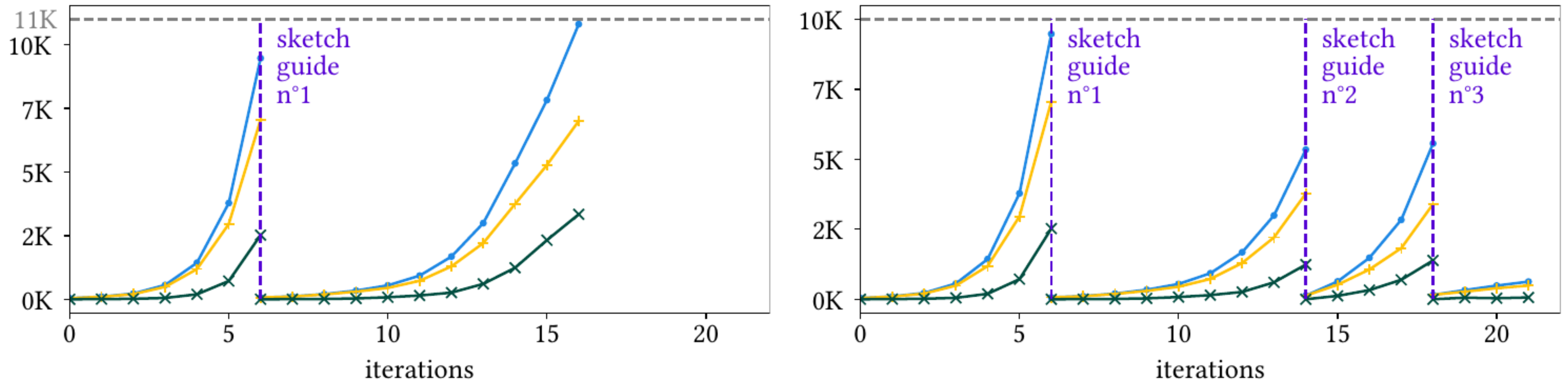
[3] AMD Ryzen 5 PRO 2500U

Evaluation

E-Graph Evolution



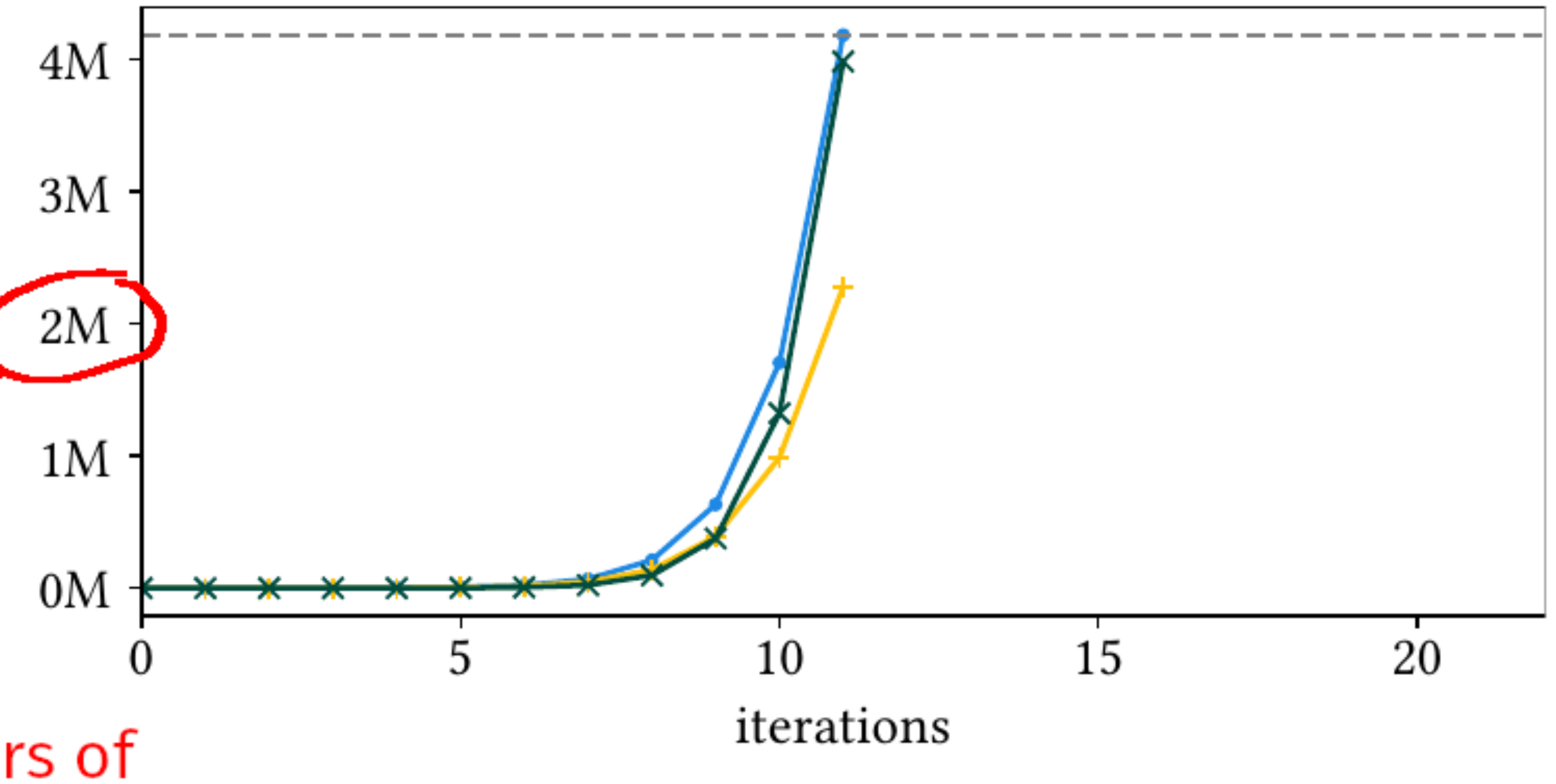
(a) unguided, *blocking*, found: ✓ (b) unguided, *parallel*, found: ✗



(c) sketch-guided, *blocking*, found: ✓ (d) sketch-guided, *parallel*, found: ✓

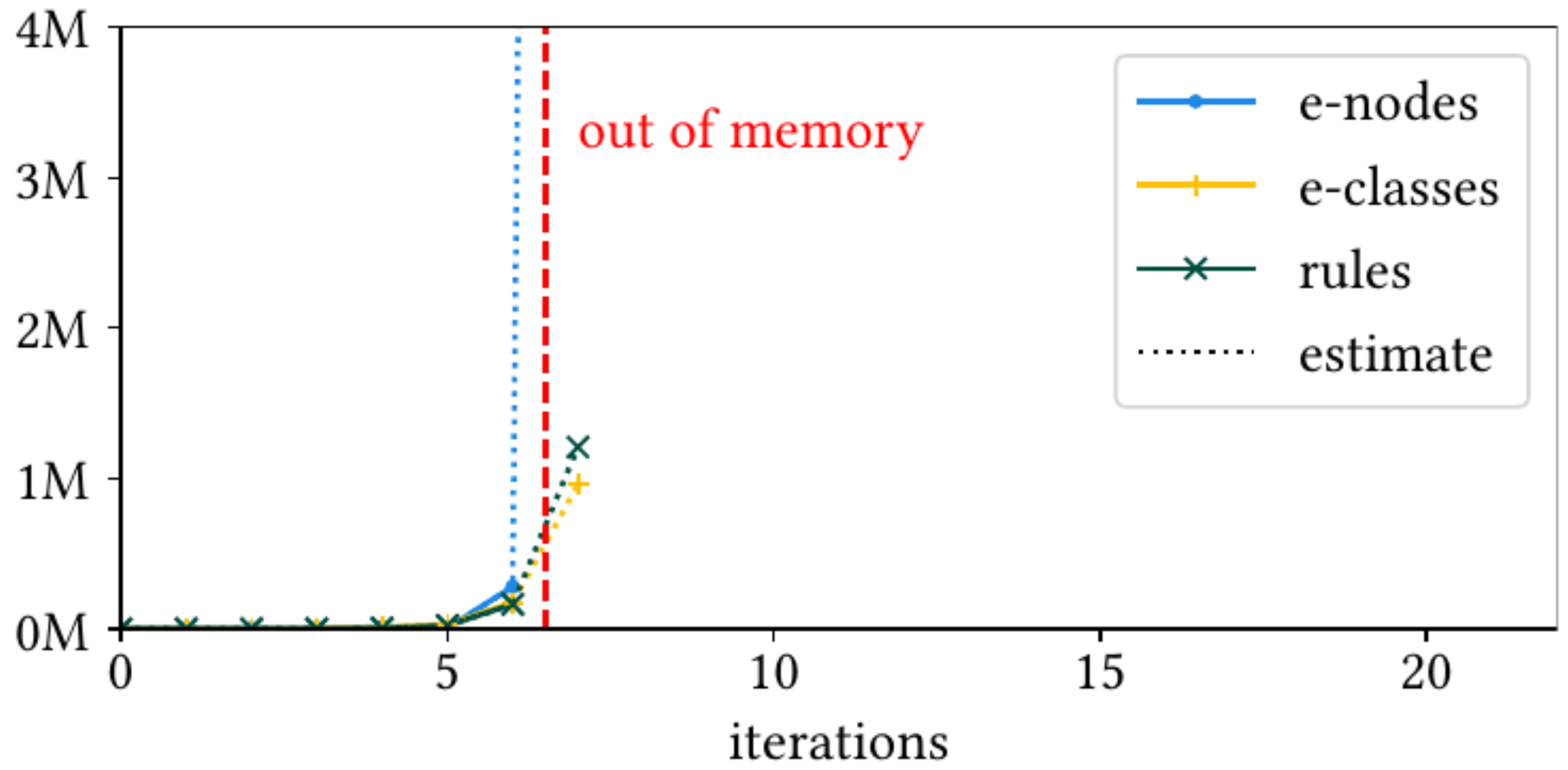
Evaluation

E-Graph Evolution

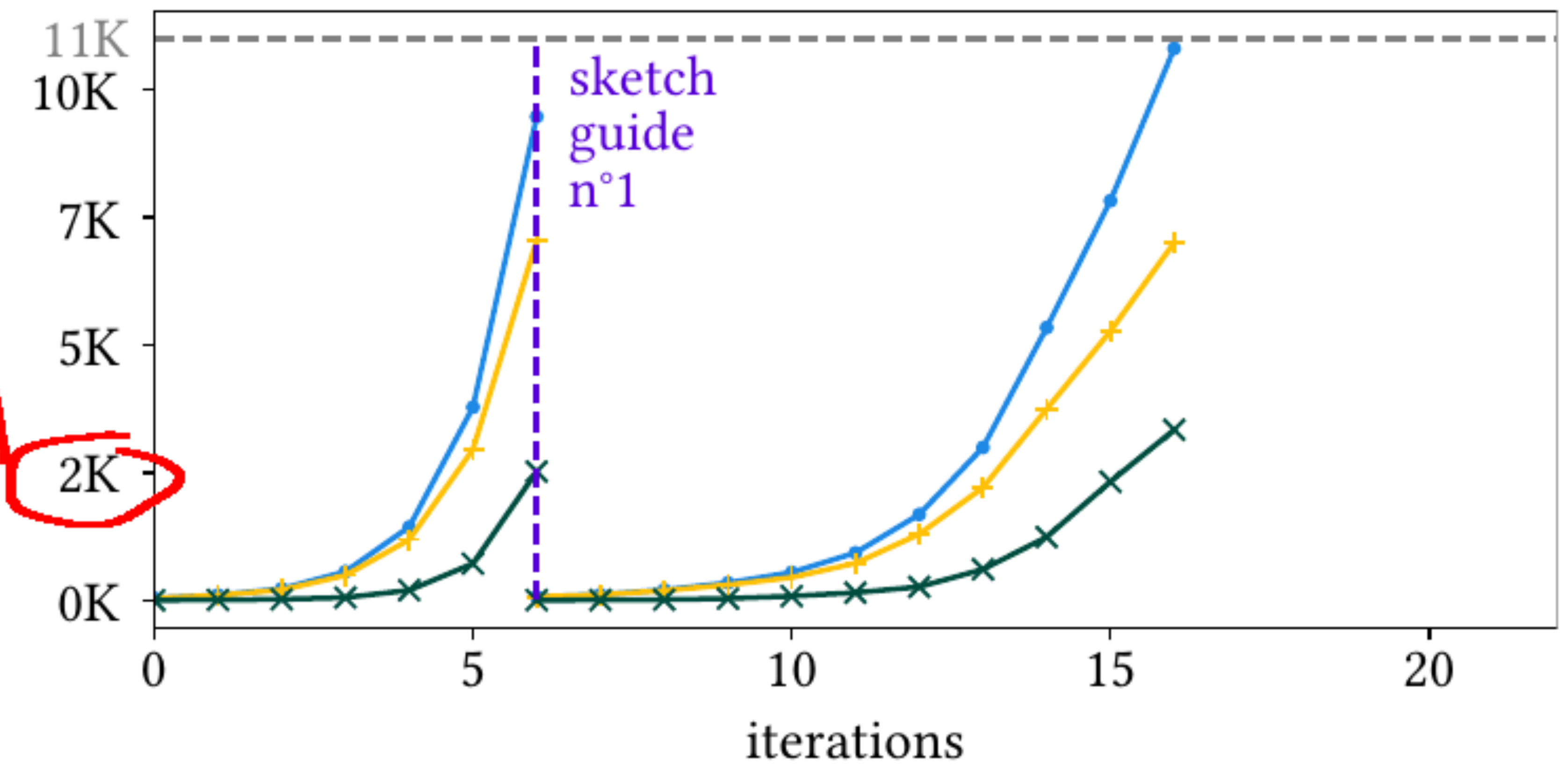


three orders of magnitude

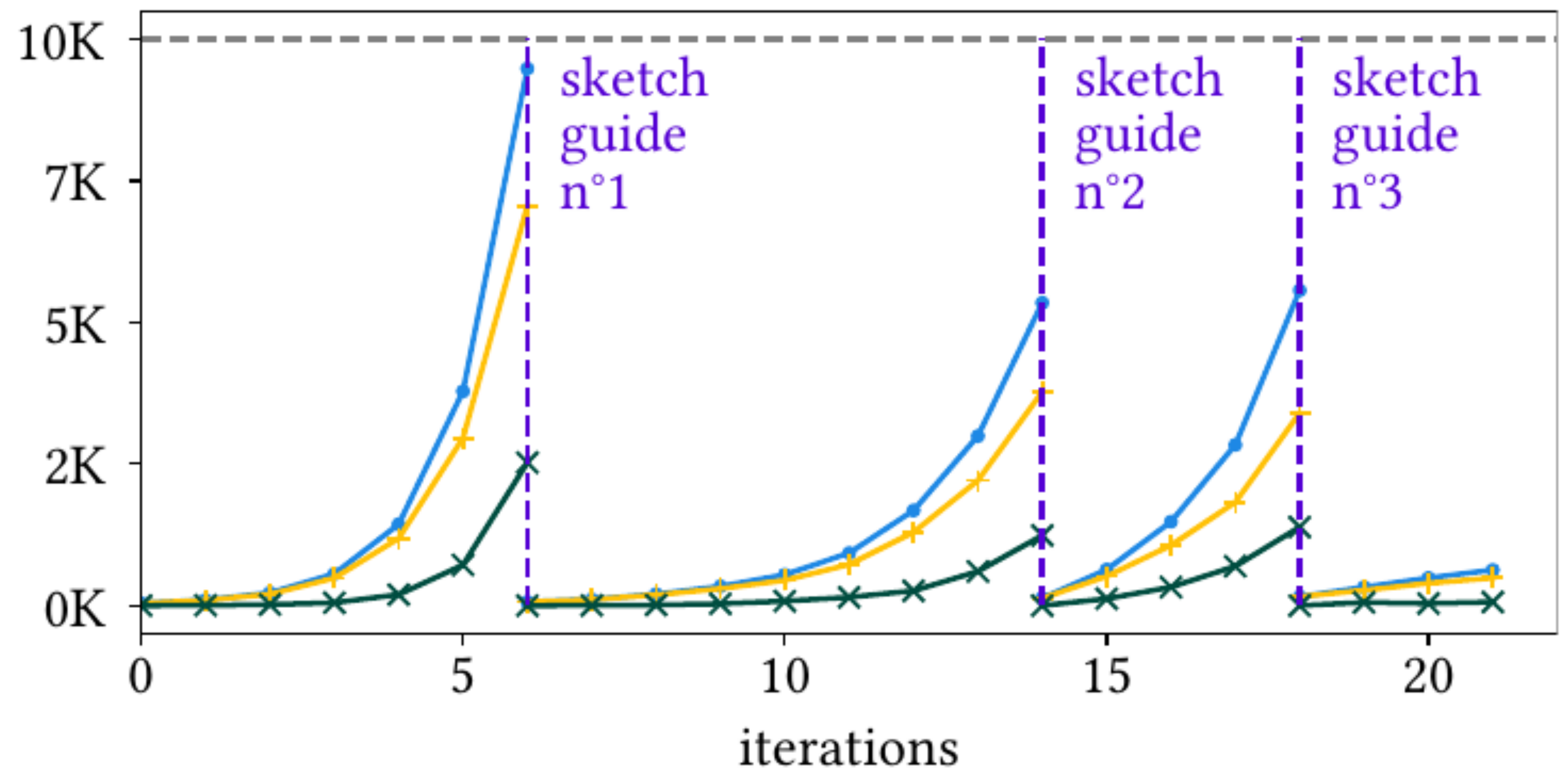
(a) unguided, *blocking*, found: ✓



(b) unguided, *parallel*, found: ✗



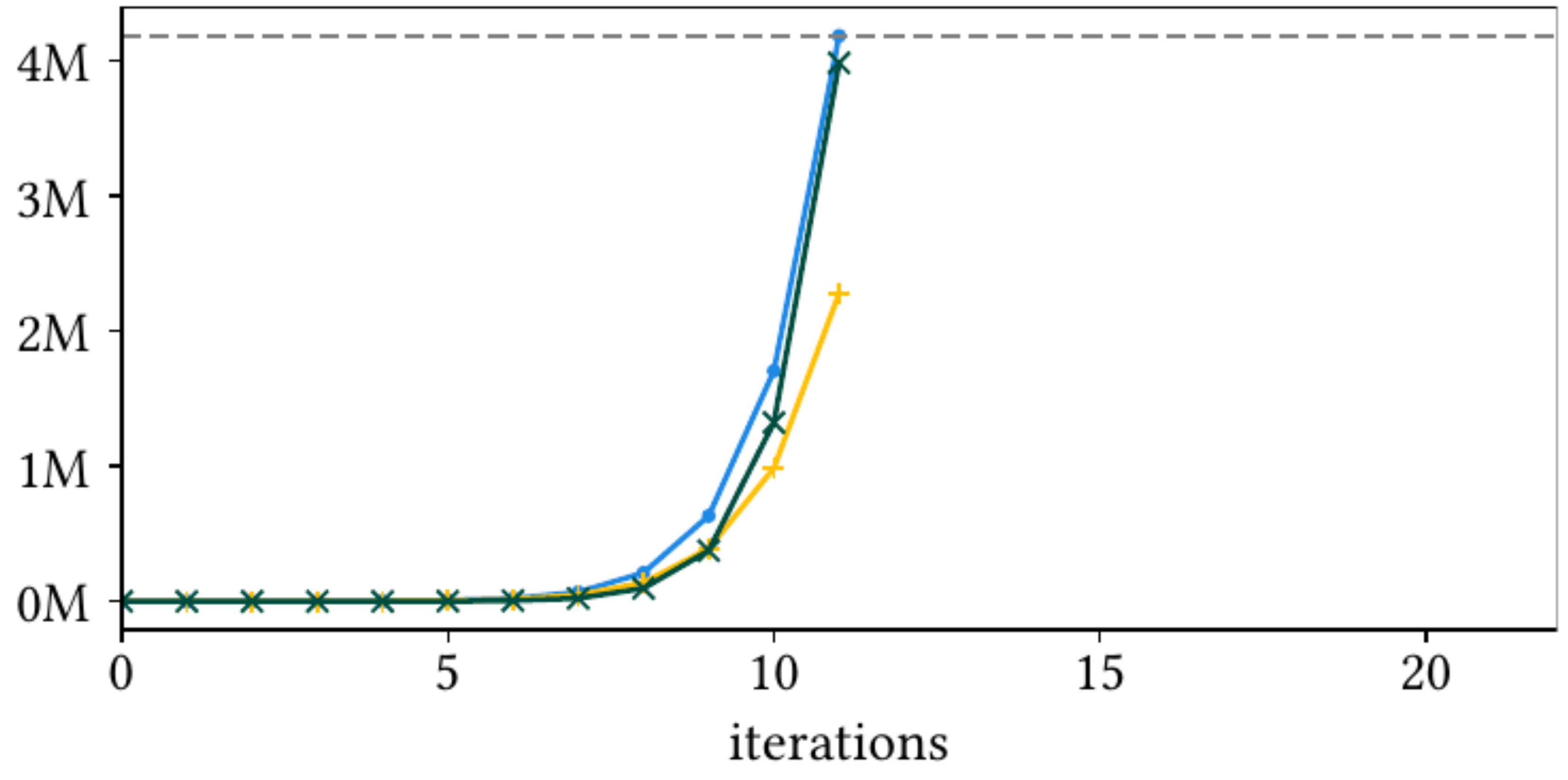
(c) sketch-guided, *blocking*, found: ✓



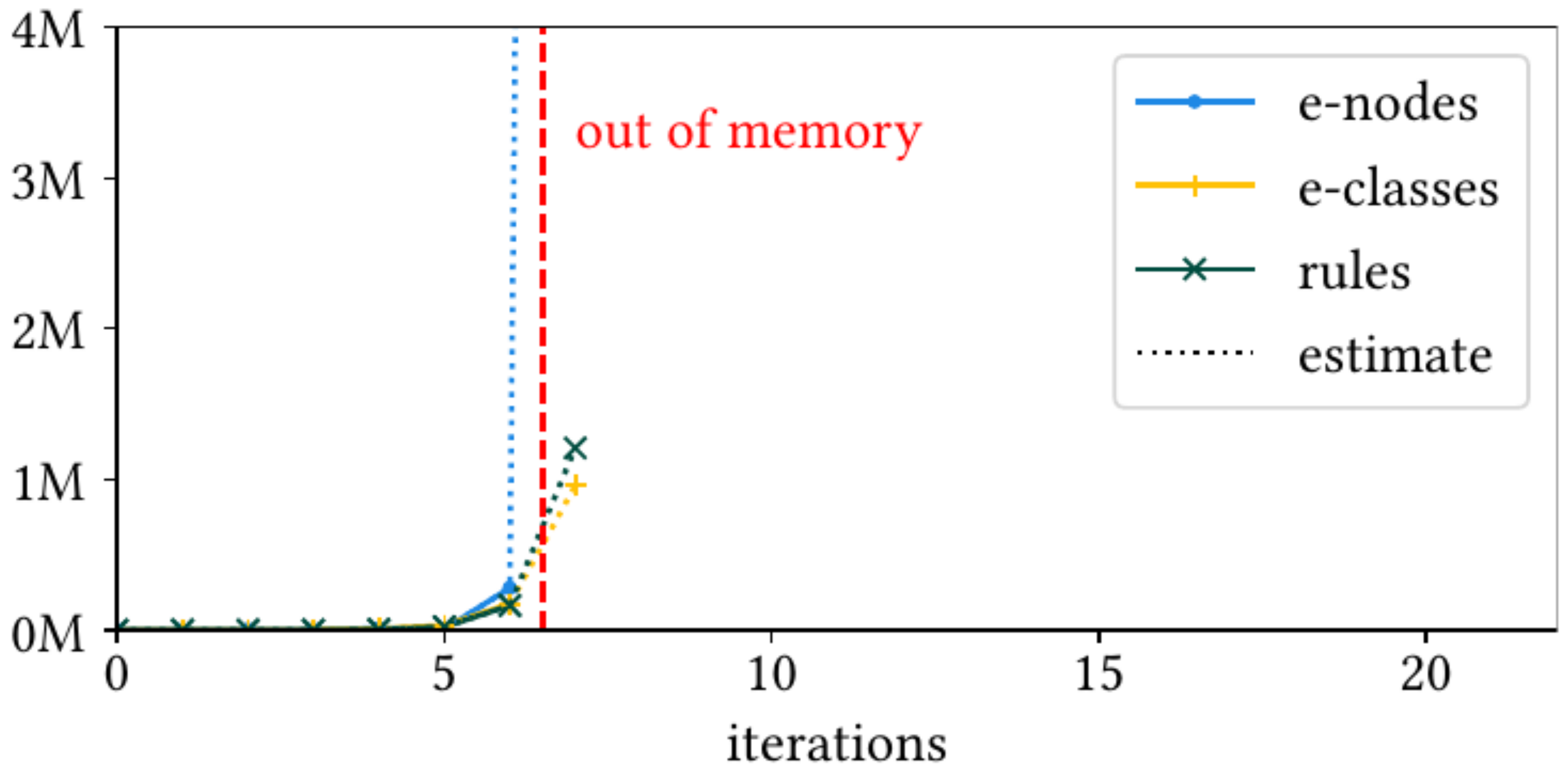
(d) sketch-guided, *parallel*, found: ✓

Evaluation

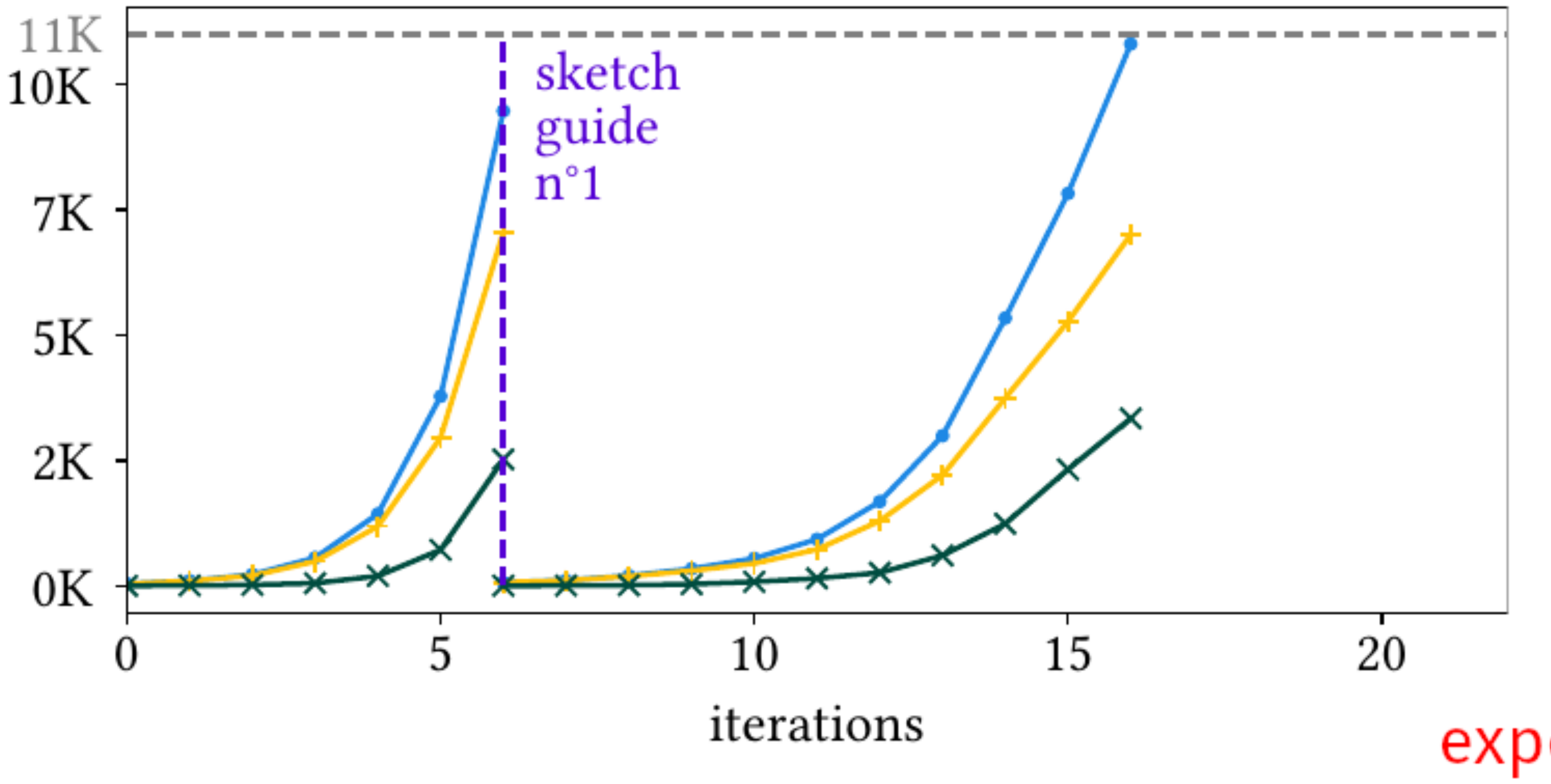
E-Graph Evolution



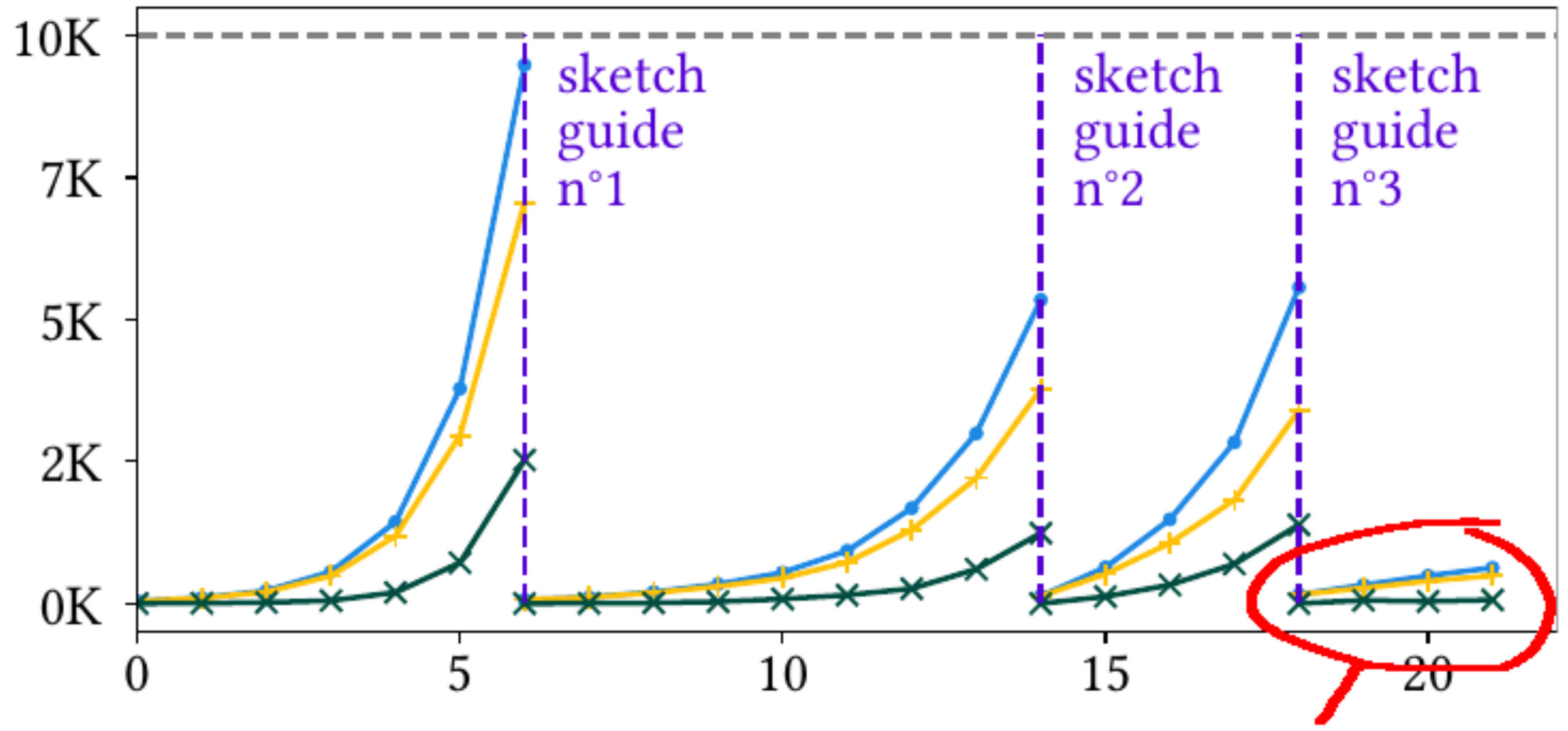
(a) unguided, *blocking*, found: ✓



(b) unguided, *parallel*, found: ✗



(c) sketch-guided, *blocking*, found: ✓



exponential growth, except linear here: no explosive rewrites

(d) sketch-guided, *parallel*, found: ✓

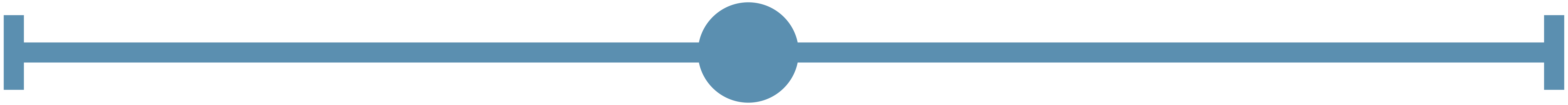
Evaluation

Sketches vs Full Program

goal	sketch guides	sketch goal	sketch sizes	program size
blocking	*split*	$reorder_1$	7	90
vectorization	*split* + $reorder_1$	$lower_1$	7	124
loop-perm	*split* + $reorder_2$	$lower_2$	7	104
array-packing	*split* + $reorder_2$ + *store*	$lower_3$	7-12	121
cache-blocks	*split* + $reorder_2$ + *store*	$lower_4$	7-12	121
parallel	*split* + $reorder_2$ + *store*	$lower_5$	7-12	121

- each sketch corresponds to a logical transformation step
- sketches elide around 90% of the program
- intricate details such as array reshaping patterns are not specified (e.g. `split`, `join`, `transpose`)

Tradeoffs when optimizing with rewriting



Automatic rewriting →

✓ Minimal human effort needed

✓ Human is in control, fast searches required

← Manual rewriting

✓ No human needed in optimization process

✗ Costly & Lengthy search process

✗ Does not (yet) scale to all programs

Extensive human effort needed ✗

Human is in control, no search required ✓

Strategies are too sensitive

⇒ don't scale across applications ✗

Thanks to all the PhD students

Johannes
Lenfers

Martin
Lücke

Federico
Pizzuti

Xueying
Qin

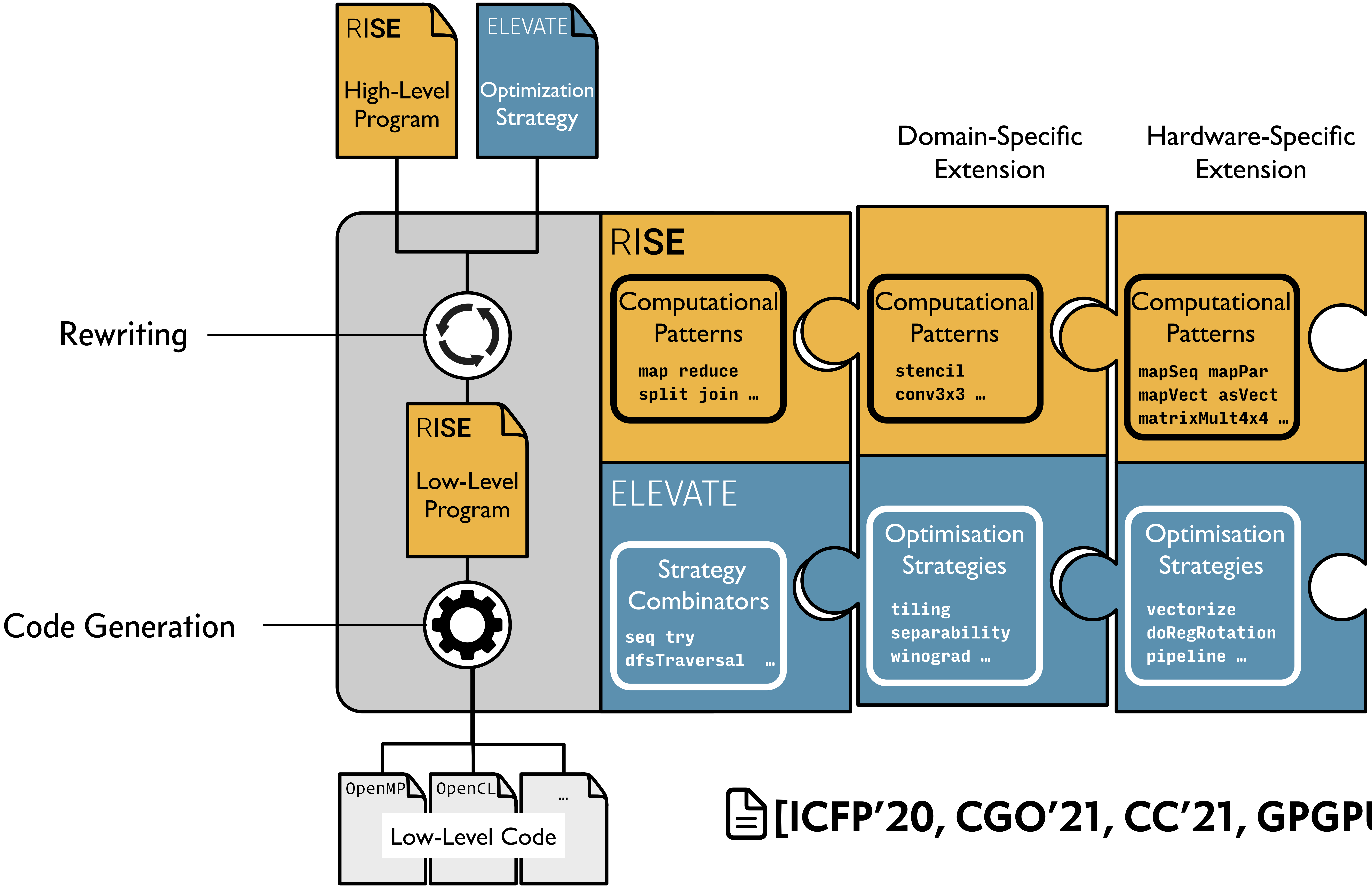
Bastian
Hagedorn

Thomas
Kœhler

Rongxiao
Fu

Bastian
Köpcke

How to design the next 700 optimizing compilers



Michel Steuwer

https://michel.steuwer.info

https://rise-lang.org/

https://elevate-lang.org/

[ICFP'20, CGO'21, CC'21, GPGPU'22, arXiv'22]