

# Systematically Extending a High-Level Code Generator with Support for Tensor Cores

Lukas Siefke, Bastian Köpcke, Sergei Gorlatch (*University of Münster*),  
and Michel Steuwer (*University of Edinburgh*)

GPGPU 2022 - 3 April 2022

# A “new golden age of computer architecture”

*“The next decade will see a Cambrian explosion of novel computer architectures, meaning exiting times for computer in academia and in industry.”*

Hennessy and Patterson

**How are we going to program new specialised hardware architectures?**

turing lecture

DOI:10.1145/3282307

Innovations like domain-specific hardware, enhanced security, open instruction sets, and agile chip development will lead the way.

BY JOHN L. HENNESSY AND DAVID A. PATTERSON

## A New Golden Age for Computer Architecture

WE BEGAN OUR Turing Lecture June 4, 2018<sup>11</sup> with a review of computer architecture since the 1960s. In addition to that review, here, we highlight current challenges and identify future opportunities, projecting another golden age for the field of computer architecture in the next decade, much like the 1980s when we did the research that led to our award, delivering gains in cost, energy, and security, as well as performance.

*“Those who cannot remember the past are condemned to repeat it.”*  
—George Santayana, 1905

Software talks to hardware through a vocabulary called an instruction set architecture (ISA). By the early 1960s, IBM had four incompatible lines of computers, each with its own ISA, software stack, I/O system, and market niche—targeting small business, large business, scientific, and real time, respectively. IBM



engineers, including ACM A.M. Turing Award laureate Fred Brooks, Jr., thought they could create a single ISA that would efficiently unify all four of these ISA bases.

They needed a technical solution for how computers as inexpensive as

### » key insights

- Software advances can inspire architecture innovation.
- Elevating the hardware/software interface creates opportunities for architecture innovation.
- The marketplace ultimately settles architecture debates.

# High-Level DSLs and Code Generators

## Promise

- Programs are written in a *simple* high-level language
- achieve high-performance "for free"

## Halide



## Futhark

Accelerate

[Tiramisu-Compiler](#) / [tiramisu](#)

Fireiron NVIDIA.

## Challenge

***How to keep pace with the increasingly faster changing hardware architectures?***

LIFT

Dex

Google Research

RISE

# NVIDIA Tensor Cores

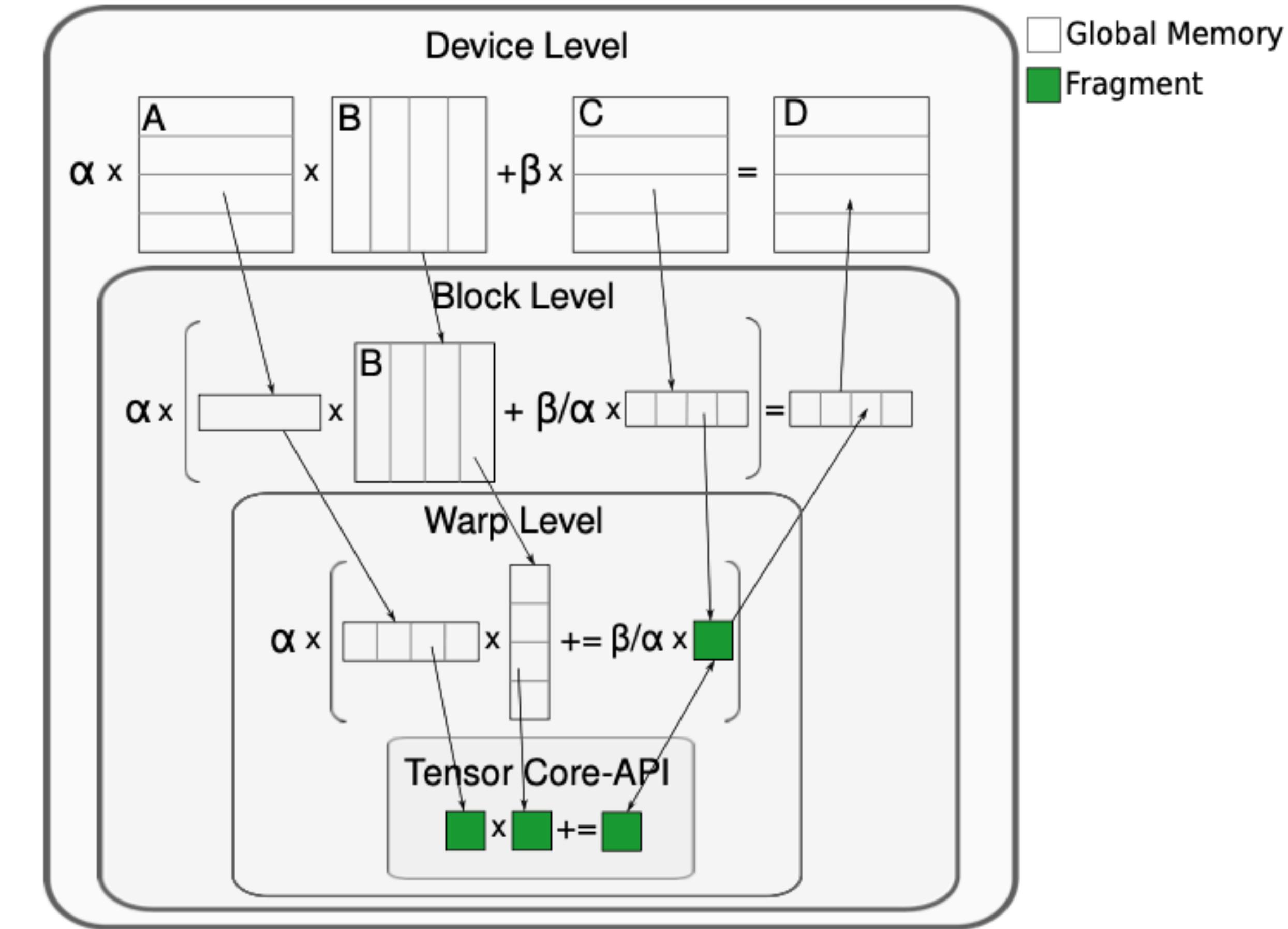
## A case study of specialised hardware

- Specialised hardware units that perform a 4x4 matrix-matrix-multiply-add

$$D = \left( \begin{array}{cccc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array} \right) \left( \begin{array}{cccc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array} \right) + \left( \begin{array}{cccc} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{array} \right)$$

FP16 or FP32      FP16      FP16 or FP32

- The V100 GPU with Tensor Cores can perform this calculation at **12x faster rate** than the Tesla P100 without Tensor Cores
- CUDA offers a *warp-level* API for exploiting Tensor Cores



# CUDA API for Tensor Cores

- Tensor Cores operate on *fragments* of the overall matrix
- `mma_sync` performs the matrix-matrix-multiply-add on the fragments
- `load/store_matrix_sync` load/store a fragment from global memory.
- `fill_fragment` writes a constant value into the fragment

```
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);
```

# Adding support for Tensor Cores in Halide?

A screenshot of a GitHub issue page for the Halide repository. The title of the issue is "Is there any plan for supporting Tensor Core? #4481". The "Open" button for the issue is highlighted with a yellow box. The issue was created by "jinderek" on 20 Dec 2019. The description of the issue states: "NVIDIA Volta and Turing GPUs have Tensor Cores, which can massively accelerate large matrix operations. ([nvidia.com/en-us/data-center/tensorcore](https://nvidia.com/en-us/data-center/tensorcore)) So is there any plan to support Tensor Core?" There is one like under the comment. To the right of the issue, there is a sidebar with "enhancement" and "Notifications" sections, and a "Subscribe" button. The GitHub interface includes a navigation bar with "Pull requests", "Issues", "Marketplace", and "Explore" tabs, and a top bar showing the URL https://github.com/halide/Halide/issues/4481.

Is there any plan for supporting Tensor Core? #4481

Open comments 0

jinderek on 20 Dec 2019

NVIDIA Volta and Turing GPUs have Tensor Cores, which can massively accelerate large matrix operations. ([nvidia.com/en-us/data-center/tensorcore](https://nvidia.com/en-us/data-center/tensorcore))

So is there any plan to support Tensor Core?

Contributor

enhancement

Notifications

You're not receiving notifications from this thread.

Customize

Subscribe

7 participants

# Adding support for Tensor Cores in Halide?

The screenshot shows a GitHub issue page for the Halide repository. The URL in the address bar is <https://github.com/halide/Halide/issues/4481>. The comment is from user **vinodgro** on 24 Dec 2019. The comment text is:

To get something with ok performance your sketch seems like a good start, though it will still be a lot of work to get it to work. We have prototyped some GPU scheduling primitives in our own standalone DSL. I gave a lightening talk at the TVM conf earlier this month. I'd like to port that over to Halide and recruiting some folks internally to help. or if I get a good intern.

Below the comment are reaction icons: a smiley face, a thumbs up (1), a heart (2), and a green plus sign.

At the bottom of the comment, there is a reply from another user asking: "So is there any plan to support tensor Core?". This reply has 1 like and 1 profile picture.

On the right side of the comment, there are options to "Subscribe" and "7 participants". Below the participants, there is a row of small profile pictures.

# Adding support for Tensor Cores in Halide?

The screenshot shows a GitHub pull request page for the Halide repository. The title of the pull request is "Initial work to add Tensor Cores". A green "Open" button is highlighted with a yellow border. The pull request has 45 reviews, 24 commits, and 9 files changed. It was last updated on 11 May 2021 by mcleary. The description explains that Tensor Cores are programmable cores that perform warp-level matrix-multiply and accumulate operations, improving performance of matrix-multiply expressions in Halide. The review section lists four reviewers: steven-johnson, dsharletg, rootjalex, and halidebuildbots. At least one approving review is required to merge the pull request.

Is there any plan for supporting tensor cores? X Initial work to add Tensor Cores X

https://github.com/halide/Halide/pull/5995

halide / Halide X

Pull requests Issues Marketplace Explore

Watch Fork Star 4.9k

Code Issues Pull requests 45 Discussions Actions Projects 2 Wiki 22

## Initial work to add Tensor Cores support in Halide #5995

**Open** halide:main ← mcleary:tensorcore\_support

Conversation 46 Commits 24 Checks 0 Files changed 9 +758 -53

mcleary on 11 May 2021 edited

Tensor Cores are programmable cores that perform warp-level matrix-multiply and accumulate operations. It can greatly improve performance of matrix-multiply expressions written in Halide when the hardware is available.

This works by pattern-matching an expression of the form

Reviewers – review now

- steven-johnson
- dsharletg
- rootjalex
- halidebuildbots

At least 1 approving review is required to merge this pull request.

# Adding support for Tensor Cores in Halide?

This works by pattern-matching an expression of the form

```
RDom k(0, matrix_size);  
C(x, y) += f32(A(k, y)) * f32(B(x, k));
```

Initial work to add Tensor Cores to Halide.

**Open**

halide:main ← and generating code that will realize the function using the GPU.

The current support is quite limited. The only data types currently supported are float16\_t for A and B and float for C. The dimensions of the input matrices must be a multiple of 16 since the only shape currently supported is m16n16k16. Also note that the generated PTX code is not the most efficient way to use tensor cores but it gives good results with a simple schedule.

This works by pattern-matching an expression of the form

OctoTree >

Conversation 46

mcleary on 11 May

Tensor Cores are planned for matrix-multiply and currently supported is m16n16k16. Also note that the generated performance of m16n16k16 PTX code is not the most efficient way to use tensor cores but the hardware is available.

reviewers – review now

+758 -53

steven-johnson

dsharletg

rootjalex

halidebuildbots

Last 1 approving review is required to merge this pull request.

# Adding support for Tensor Cores in Halide?

The screenshot shows a GitHub pull request page for a Halide repository. The title of the pull request is "Initial work to add Tensor Cores". The URL in the address bar is <https://github.com/halide/Halide/pull/5995>. The page displays a conversation between two contributors:

- steven-johnson** on 15 Nov 2021: Is this PR still active? Should it be closed?
- frengeles** on 24 Nov 2021: sorry about the late reply, yes it's still active

A large portion of the page is obscured by a dark gray redaction box, but some text is visible at the bottom:

the input matrices must be a multiple of 16 since the only shape  
Tensor Cores are p currently supported is m16n16k16. Also note that the generated  
matrix-multiply and performance of ma PTX code is not the most efficient way to use tensor cores but  
the hardware is av it gives good results with a simple schedule.

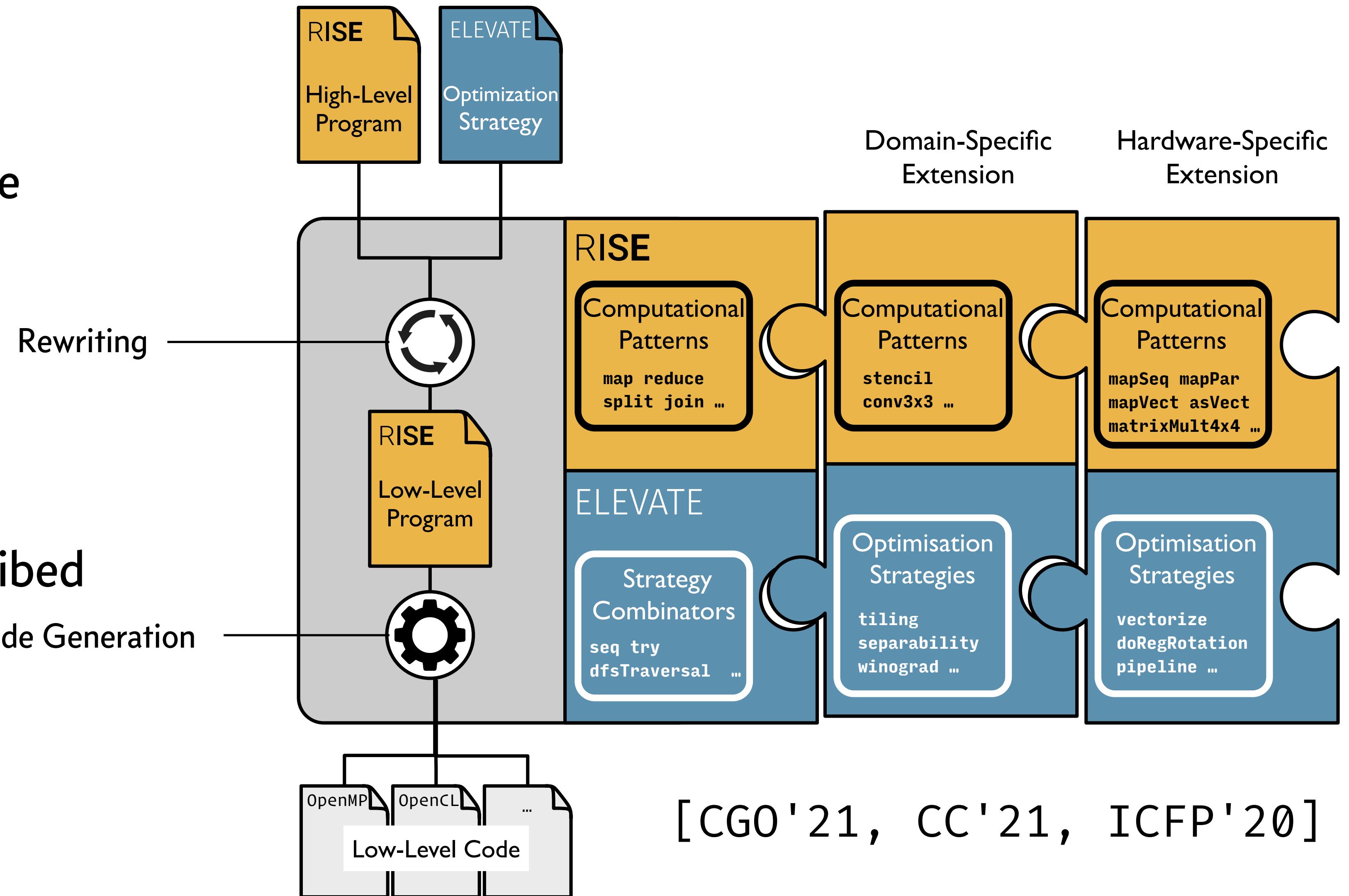
This works by pattern-matching an expression of the form

Contributor steven-johnson  
dsharletg  
rootjalex  
halidebuildbots

Last 1 approving review is required to merge this pull request.

# RISE & Shine an extensible compiler design

- Spiritual successor to the LIFT project
- Computations are expressed by computational patterns
- Optimisations are described as compositions of rewrite rules



<https://rise-lang.org/>

[CGO'21, CC'21, ICFP'20]

# GEMM in RISE

## High-Level GEMM

```

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

```

## Optimization Strategy

ELEVATE

## Rewriting

## Low-Level GEMM

```

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

```

## Translation

## Imperative GEMM

```

17 depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
18   parForBlock(m,Array[n,f16], output, fun(rowIndex,outRow =>
19     parForThreads(n, f16, outRow, fun(colIdx,outElem =>
20       new(Local,f32, fun((accumExp, accumAcc) =>
21         accumAcc = 0.0f;
22         for(k, fun(i => accumAcc = accumExp +
23           fst(idx(i, zip(fst(idx(rowIndex, zip(A,C)))), +
24             fst(idx(colIdx, zip(transpose(B),
25               snd(idx(rowIndex, zip(A,C)))))))) * *
26             snd(idx(i, zip(fst(idx(rowIndex, zip(A,C)))), +
27               fst(idx(colIdx, zip(transpose(B),
28                 snd(idx(rowIndex, zip(A,C))))))))));
29         outElem = alpha * accumExp + beta *
30           snd(idx(colIdx, zip(transpose(B),
31             snd(idx(rowIndex, zip(A,C))))))) ))));
32         syncThreads())))))

```

## Codegen

```

33 __global__ void gemm_kernel(float* __restrict__ output,
34   int m, int n, int k, const __half* __restrict__ A,
35   const __half* __restrict__ B,
36   const float* __restrict__ C, float alpha, float beta) {
37     for(int blockIdx=blockIdx.x;
38         blockIdx.x<m; blockIdx += gridDim.x) {
39       for(int colIdx=threadIdx.x;
40           threadIdx.x<n; threadIdx += blockDim.x) {
41         float accum = 0;
42         for (int i = 0; i < k; i++) {
43           accum = accum + A[i + blockIdx*k] * B[colIdx + i*n];
44         }
45         output[colIdx + blockIdx * n] =
46           alpha * accum + beta * C[colIdx + blockIdx*n];
47       }
48       __syncthreads(); }
}

```

# GEMM in RISE

## High-Level functional primitives

### High-Level GEMM

```

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

```

### Optimization Strategy

ELEVATE

### Rewriting

### Low-Level GEMM

```

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

```

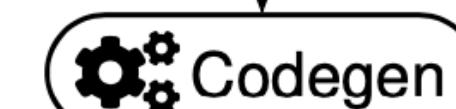


### Imperative GEMM

```

17 depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
18   parForBlock(m,Array[n,f16], output, fun(rowIndex,outRow =>
19     parForThreads(n, f16, outRow, fun(colIdx,outElem =>
20       new(Local,f32, fun((accumExp, accumAcc) =>
21         accumAcc = 0.0f;
22         for(k, fun(i => accumAcc = accumExp +
23           fst(idx(i, zip(fst(idx(rowIndex, zip(A,C)))), +
24             fst(idx(colIdx, zip(transpose(B),
25               snd(idx(rowIndex, zip(A,C)))))))) * +
26             snd(idx(i, zip(fst(idx(rowIndex, zip(A,C)))), +
27               fst(idx(colIdx, zip(transpose(B),
28                 snd(idx(rowIndex, zip(A,C))))))))));
29         outElem = alpha * accumExp + beta *
30           snd(idx(colIdx, zip(transpose(B),
31             snd(idx(rowIndex, zip(A,C))))))) ))));
32         syncThreads())))))

```



```

33 __global__ void gemm_kernel(float* __restrict__ output,
34   int m, int n, int k, const __half* __restrict__ A,
35   const __half* __restrict__ B,
36   const float* __restrict__ C, float alpha, float beta) {
37     for(int blockIdx=blockIdx.x;
38         blockIdx.x<m; blockIdx += gridDim.x) {
39       for(int colIdx=threadIdx.x;
40           threadIdx.x<n; threadIdx += blockDim.x) {
41         float accum = 0;
42         for (int i = 0; i < k; i++) {
43           accum = accum + A[i + blockIdx*k] * B[colIdx + i*n];
44         }
45         output[colIdx + blockIdx * n] =
46           alpha * accum + beta * C[colIdx + blockIdx*n];
47       }
48       __syncthreads(); }
}

```

# GEMM in RISE

## High-Level functional primitives

### High-Level GEMM

```

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

```

### Optimization Strategy

ELEVATE

### Rewriting

### Low-Level GEMM

```

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

```

## Low-Level functional primitives

### Translation

### Imperative GEMM

```

17 depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
18   parForBlock(m,Array[n,f16], output, fun(rowIndex,outRow =>
19     parForThreads(n, f16, outRow, fun(colIdx,outElem =>
20       new(Local,f32, fun((accumExp, accumAcc) =>
21         accumAcc = 0.0f;
22         for(k, fun(i => accumAcc = accumExp +
23           fst(idx(i, zip(fst(idx(rowIndex, zip(A,C)))), +
24             fst(idx(colIdx, zip(transpose(B),
25               snd(idx(rowIndex, zip(A,C)))))))) * +
26             snd(idx(i, zip(fst(idx(rowIndex, zip(A,C)))), +
27               fst(idx(colIdx, zip(transpose(B),
28                 snd(idx(rowIndex, zip(A,C))))))))));
29         outElem = alpha * accumExp + beta *
30           snd(idx(colIdx, zip(transpose(B),
31             snd(idx(rowIndex, zip(A,C))))))) ))));
32         syncThreads())))))

```

### Codegen

```

33 __global__ void gemm_kernel(float* __restrict__ output,
34   int m, int n, int k, const __half* __restrict__ A,
35   const __half* __restrict__ B,
36   const float* __restrict__ C, float alpha, float beta) {
37     for(int blockIdx=blockIdx.x;
38         blockIdx.x<m; blockIdx += gridDim.x) {
39       for(int colIdx=threadIdx.x;
40           threadIdx.x<n; threadIdx += blockDim.x) {
41         float accum = 0;
42         for (int i = 0; i < k; i++) {
43           accum = accum + A[i + blockIdx*k] * B[colIdx + i*n];
44         }
45         output[colIdx + blockIdx * n] =
46           alpha * accum + beta * C[colIdx + blockIdx*n];
47       }
48       __syncthreads(); }
}

```

# GEMM in RISE

*High-Level  
functional primitives*

High-Level GEMM

```

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

```

Optimization Strategy

ELEVATE

Rewriting

Low-Level GEMM

```

9 depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
10   zip(A)(C) |> mapBlock(fun(rowAC =>
11     zip(B |> transpose)(snd(rowAC)) |>
12     mapThreads(fun(colBC => zip(fst(rowAC))(fst(colBC)) |>
13       reduceSeq(local)(fun((acc,ab) =>
14         acc + fst(ab) * snd(ab)),0) |>
15         fun(r => (alpha * r) + (beta * snd(colBC)))) )))))

```

*Low-Level  
functional primitives*

*Low-Level  
imperative primitives*

Translation

Imperative GEMM

```

17 depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
18   parForBlock m, Array[n, f16], output, fun(rowIndex,outRow =>
19     parForThreads(n, f16, outRow, fun(colIdx,outElem =>
20       newLocal f32, fun((accumExp, accumAcc) =>
21         accumAcc = 0.0f;
22         for k, fun(i => accumAcc = accumExp +
23           fst(idx(i, zip(fst(rowIndex, zip(A,C)))), +
24             fst(idx(colIdx, zip(transpose(B),
25               snd(idx(rowIndex, zip(A,C))))))) * +
26             snd(idx(i, zip(fst(idx(rowIndex, zip(A,C)))), +
27               fst(idx(colIdx, zip(transpose(B),
28                 snd(idx(rowIndex, zip(A,C))))))) * +
29             outElem = alpha * accumExp + beta *
30               snd(idx(colIdx, zip(transpose(B),
31                 snd(idx(rowIndex, zip(A,C))))))) ))));
32 syncThreads())))

```

Codegen

```

33 __global__ void gemm_kernel(float* __restrict__ output,
34   int m, int n, int k, const __half* __restrict__ A,
35   const __half* __restrict__ B,
36   const float* __restrict__ C, float alpha, float beta) {
37   for(int blockIdx=blockIdx.x;
38       blockIdx.x<m; blockIdx += gridDim.x) {
39     for(int colIdx=threadIdx.x;
40         threadIdx.x<n; threadIdx += blockDim.x) {
41       float accum = 0;
42       for (int i = 0; i < k; i++) {
43         accum = accum + A[i + blockIdx*k] * B[colIdx + i*n];
44       }
45       output[colIdx + blockIdx * n] =
46         alpha * accum + beta * C[colIdx + blockIdx*n];
47     }
48     __syncthreads(); }
}

```

# GEMM in RISE

*High-Level  
functional primitives*

High-Level GEMM

```

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

```

Optimization Strategy

ELEVATE

Rewriting

Low-Level GEMM

```

9  depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
10   zip(A)(C) |> mapBlock(fun(rowAC =>
11     zip(B |> transpose)(snd(rowAC)) |>
12     mapThreads(fun(colBC => zip(fst(rowAC))(fst(colBC)) |>
13       reduceSeq(local)(fun((acc,ab) =>
14         acc + fst(ab) * snd(ab)),0) |>
15         fun(r => (alpha * r) + (beta * snd(colBC)))) )))))

```

*Low-Level  
functional primitives*

*Low-Level  
imperative primitives*

Translation

Imperative GEMM

```

17 depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
18   parForBlock m, Array[n, f16], output, fun(rowIndex,outRow =>
19     parForThreads(n, f16, outRow, fun(colIdx,outElem =>
20       newLocal f32, fun((accumExp, accumAcc) =>
21         accumAcc = 0.0f;
22         for k, fun(i => accumAcc = accumExp +
23           fst(idx(i, zip(fst(rowIndex, zip(A,C)))), +
24           fst(idx(colIdx, zip(transpose(B),
25             snd(idx(rowIndex, zip(A,C))))))) * +
26           snd(idx(i, zip(fst(idx(rowIndex, zip(A,C)))), +
27             fst(idx(colIdx, zip(transpose(B),
28               snd(idx(rowIndex, zip(A,C))))))) * +
29           outElem = alpha * accumExp + beta *
30             snd(idx(colIdx, zip(transpose(B),
31               snd(idx(rowIndex, zip(A,C))))))) ))));
32 syncThreads())))

```

Codegen

```

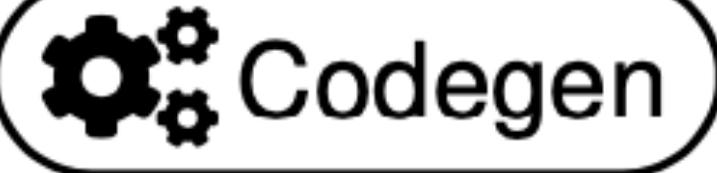
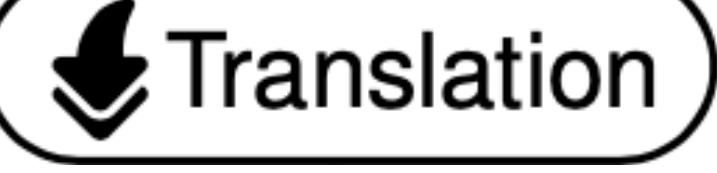
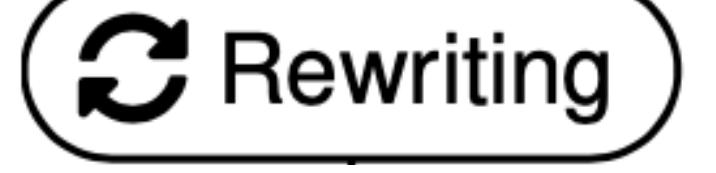
33 __global__ void gemm_kernel(float* __restrict__ output,
34   int m, int n, int k, const __half* __restrict__ A,
35   const __half* __restrict__ B,
36   const float* __restrict__ C, float alpha, float beta) {
37   for(int blockIdx=blockIdx.x;
38     blockIdx.x<m; blockIdx += gridDim.x) {
39     for(int colIdx=threadIdx.x;
40       threadIdx.x<n; threadIdx += blockDim.x) {
41       float accum = 0;
42       for (int i = 0; i < k; i++) {
43         accum = accum + A[i + blockIdx*k] * B[colIdx + i*n];
44       }
45       output[colIdx + blockIdx * n] =
46         alpha * accum + beta * C[colIdx + blockIdx * n];
47     }
48     __syncthreads(); }
}

```

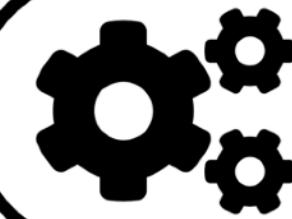
*Low-Level  
imperative code*

# Systematically Extending RISE 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 by adding one case for each low-level functional primitive
- The "acceptor translation" *accT* translates a functional expression who's result is written to output
- The "continuation translation" *conT* translates a functional expression by passing the translated expression to a *continuation* that continues the translation
- More details on the translations in

RISE & Shine: Language-Oriented Compiler Design

```
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)))  
  ... }
```

<https://arxiv.org/pdf/2201.03611.pdf>

Michel Steuwer\* Thomas Köhler† Bastian Köpcke‡ Federico Pizzuti\*

\*University of Edinburgh, Scotland, UK †University of Glasgow, Scotland, UK ‡University of Münster, Germany  
Email: michel.steuwer@ed.ac.uk, thomas.koehler@thok.eu, bastian.koepcke@wwu.de, federico.pizzuti@ed.ac.uk

### 3. Add rewrite rules to enable



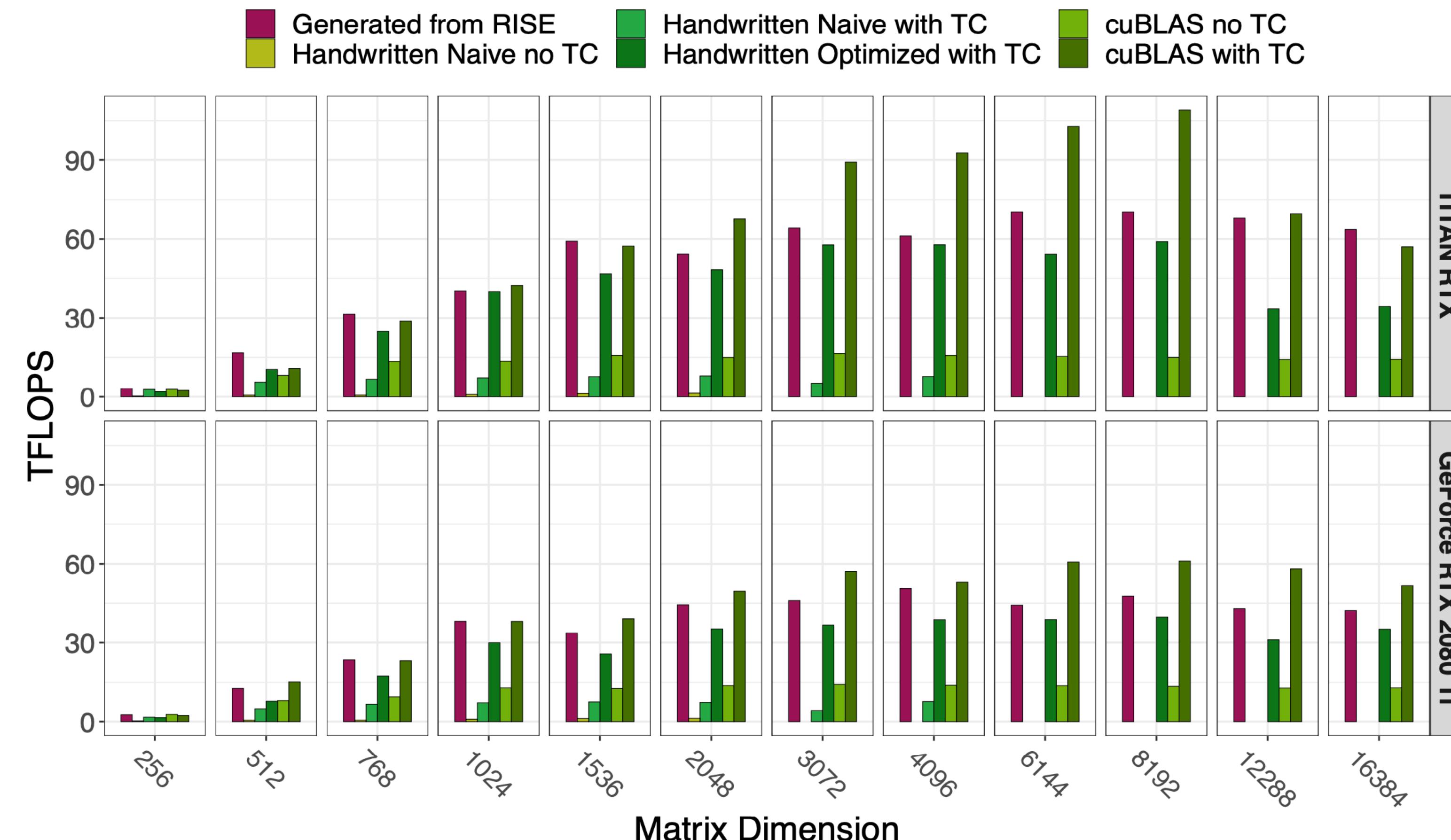
- Rewrite rules enable automatic exploitation of Tensor Cores
- Examples shows automatic use of Tensor Cores for high-level matrix multiplication code
- Rewrite rules can be applied *automatically* [GPGPU'16, ICFP'15], *manually* [ICFP'20], or *guided* [arXiv:2111.13040].

```
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
```

# Performance Evaluation



Competitive performance to manually optimised CUDA code.  
Within 36% of cuBLAS (on average only 10% slower).

# Systematically Extending a High-Level Code Generator with Support for Tensor Cores

- RISE 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*.
- Performance evaluation shows that automatically generated code is competitive to manually optimised code

Lukas Siefke, Bastian Köpcke, Sergei Gorlatch (*University of Münster*),  
and Michel Steuwer (*University of Edinburgh*)

[michel.steuwer@ed.ac.uk](mailto:michel.steuwer@ed.ac.uk)

<https://michel.steuwer.info/>

<https://rise-lang.org/>