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[11] 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.

48 COMMUNICATIONS OF THE ACM | FEBRUARY 2019 | VOL. 62 | NO. 2

High-Level DSLs and Code Generators

Promise

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

Challenge

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

Halide

Futhark

tvm

Accelerate

Tiramisu-Compiler / tiramisu

Fireiron NVIDIA

LIFT

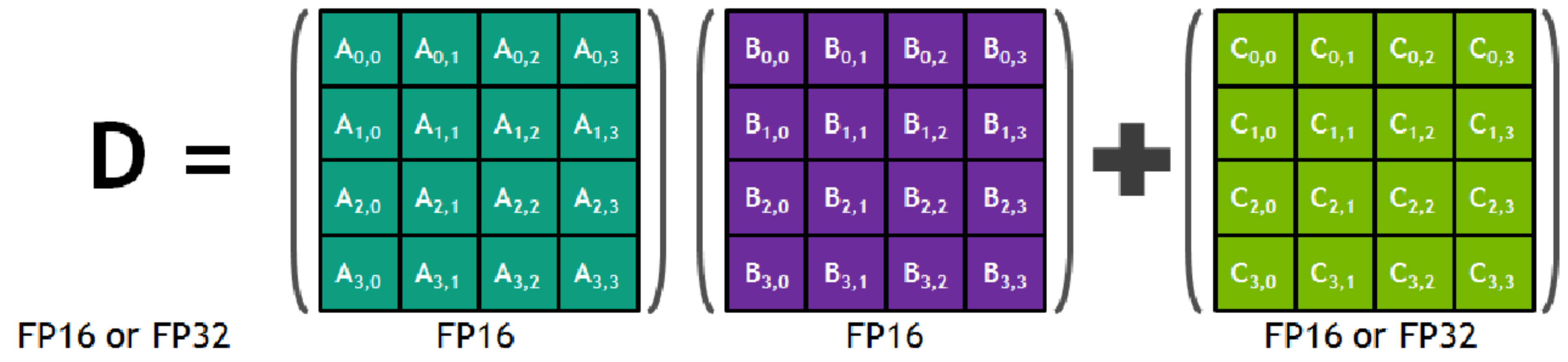
Dex Google Research

RISE

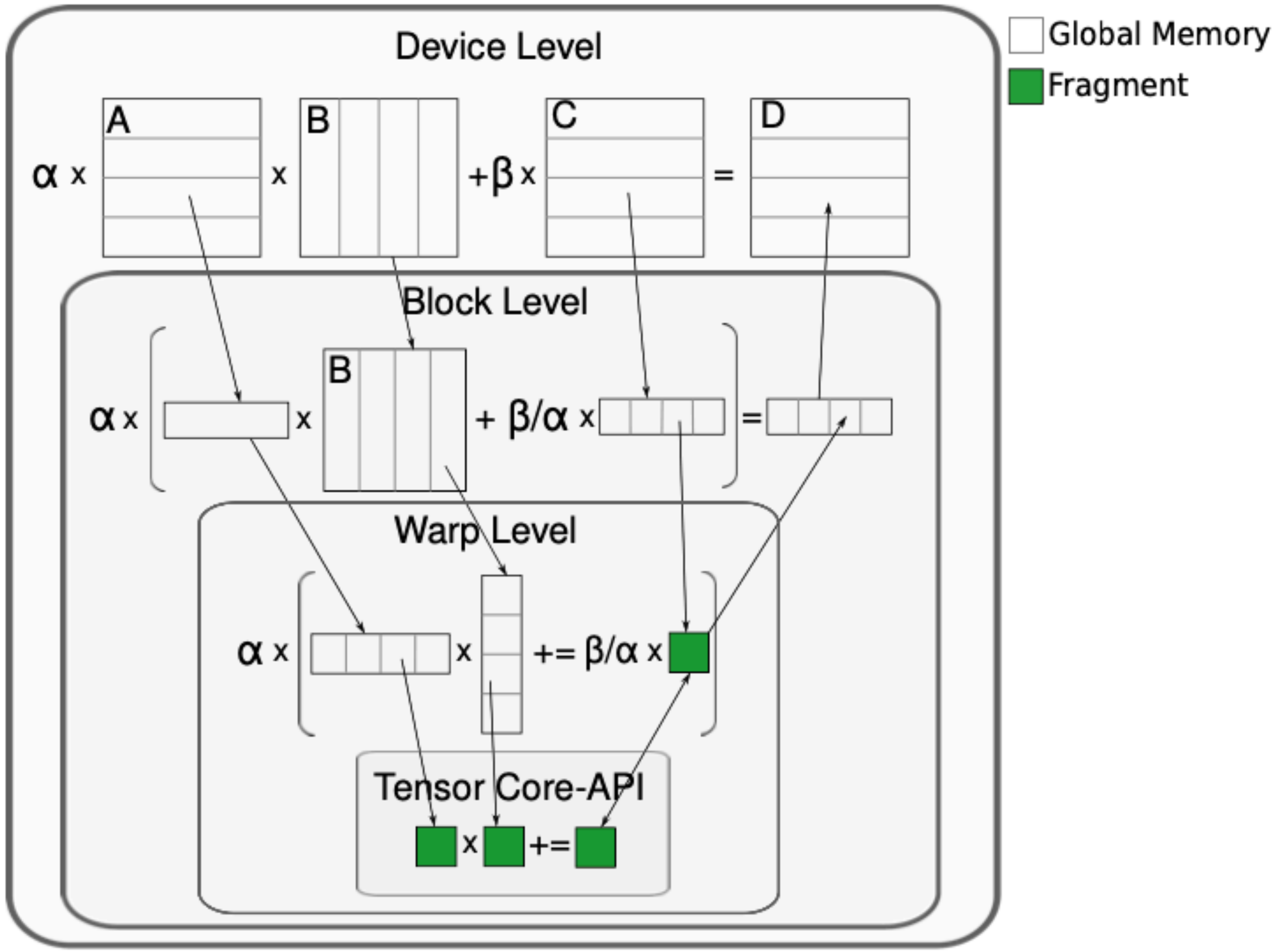
NVIDIA Tensor Cores

A case study of specialised hardware

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



- 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

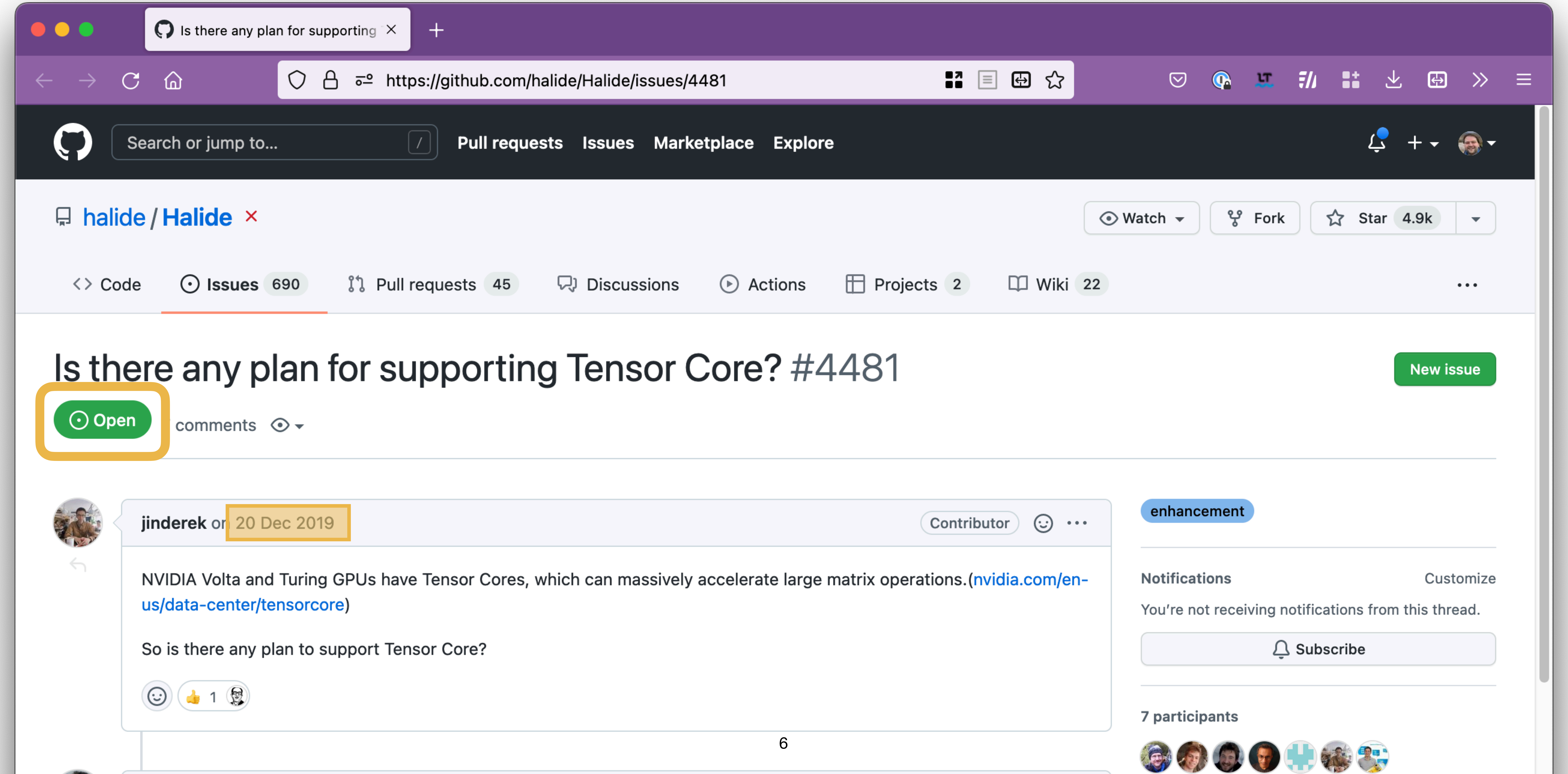
CUDA API for Tensor Cores

- 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

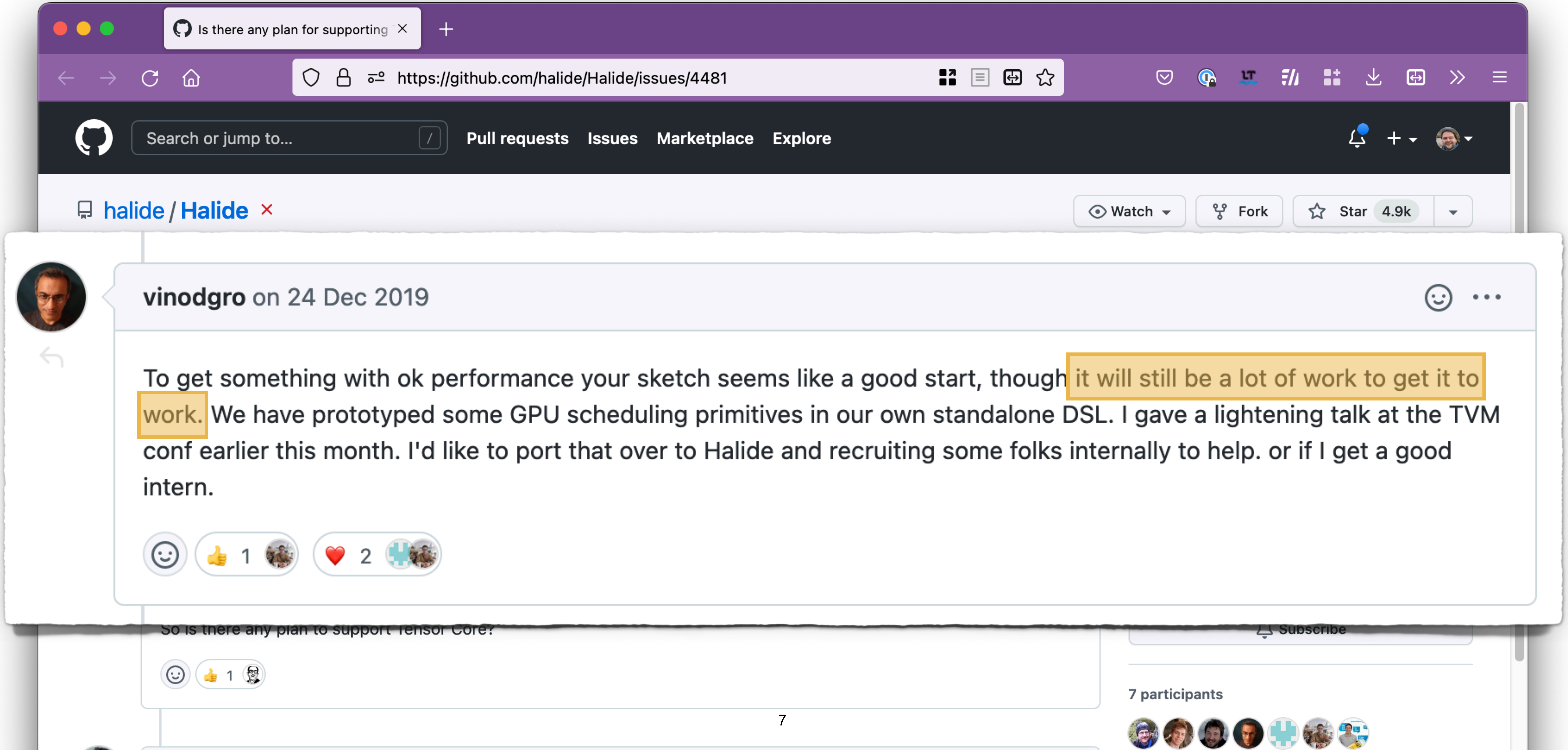
```
template<typename FragmKind, int m, int n, int k,
 typename T, typename Layout=void> class fragment;

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);
```

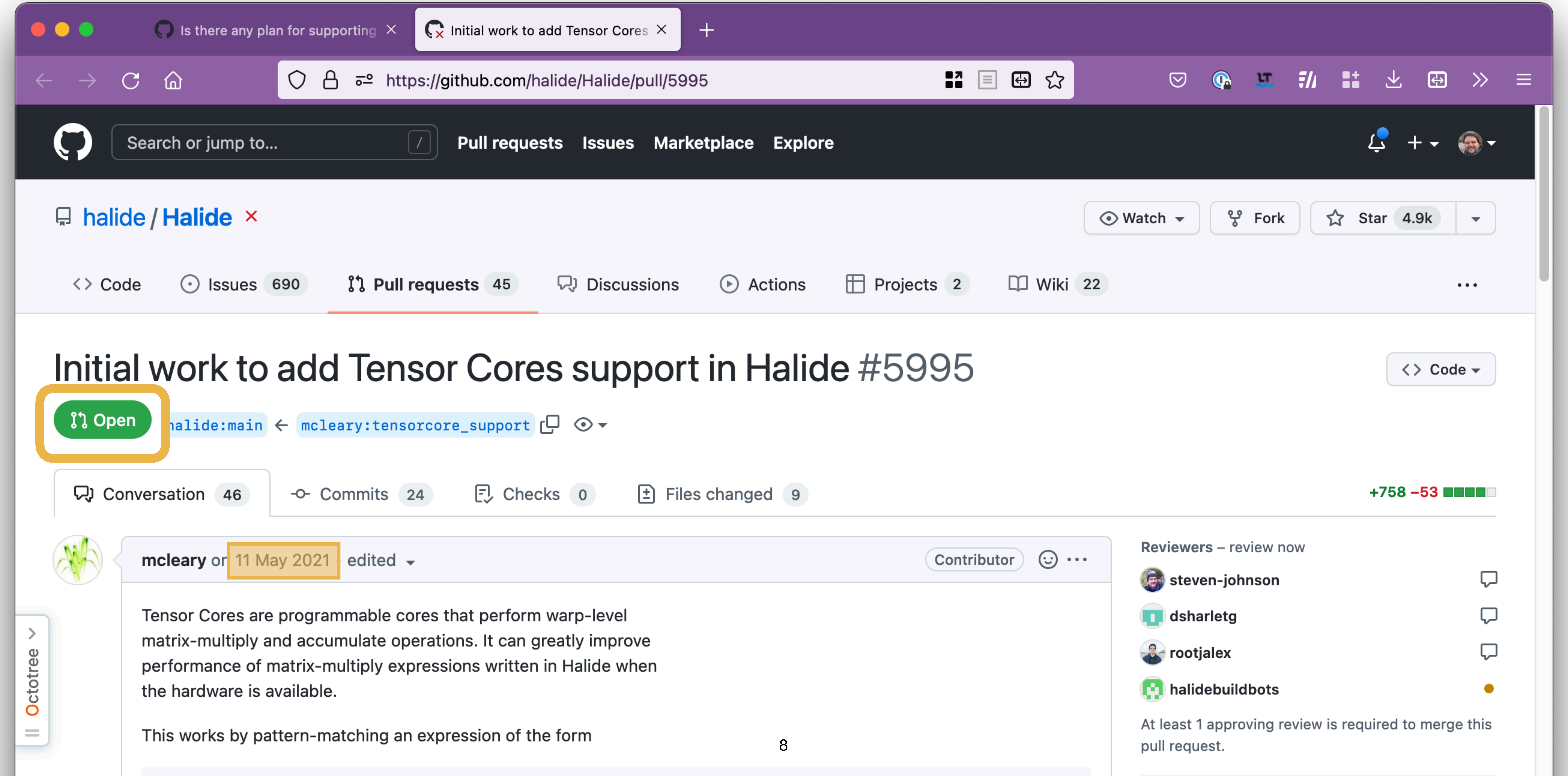
Adding support for Tensor Cores in Halide?



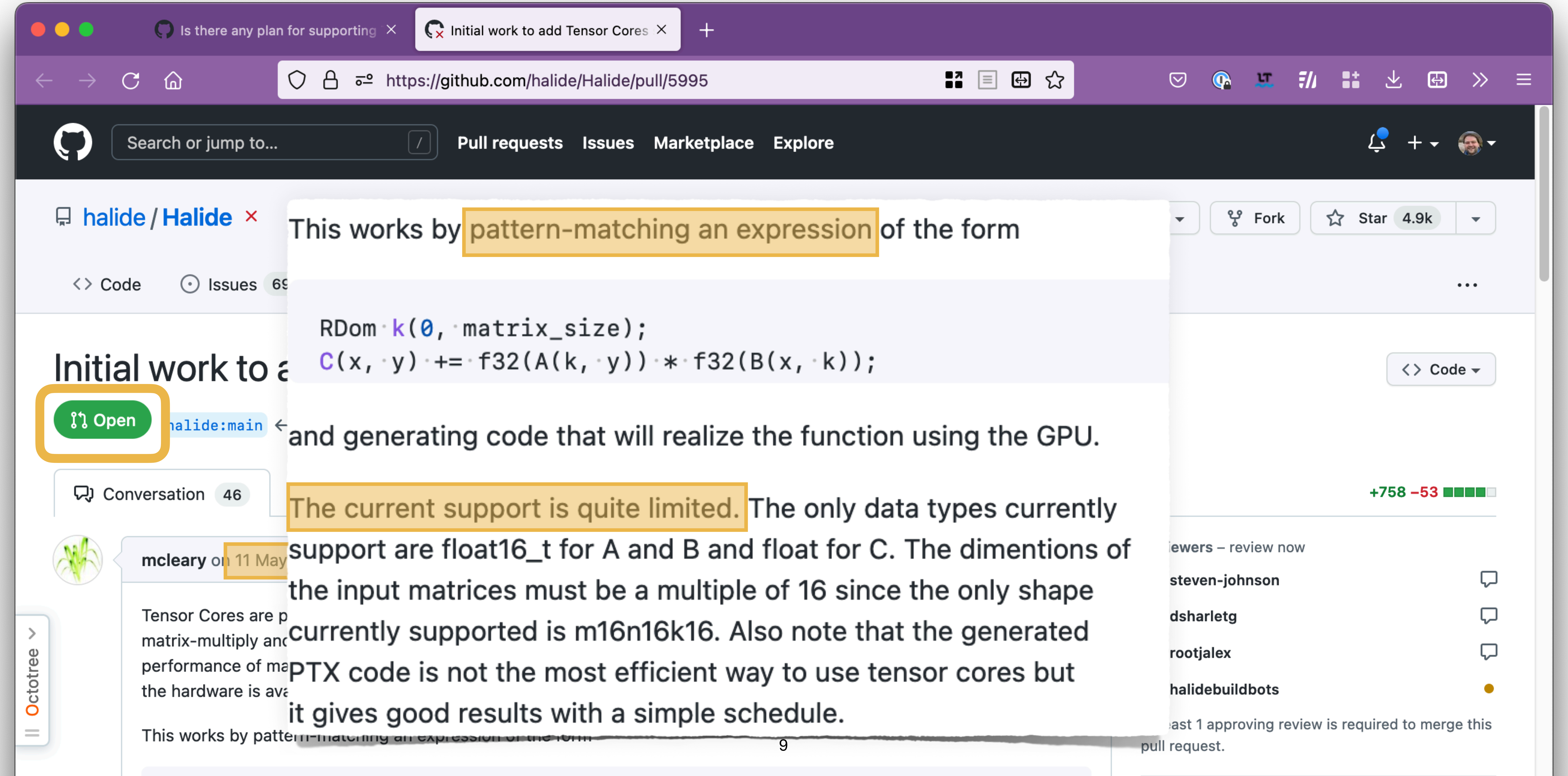
Adding support for Tensor Cores in Halide?



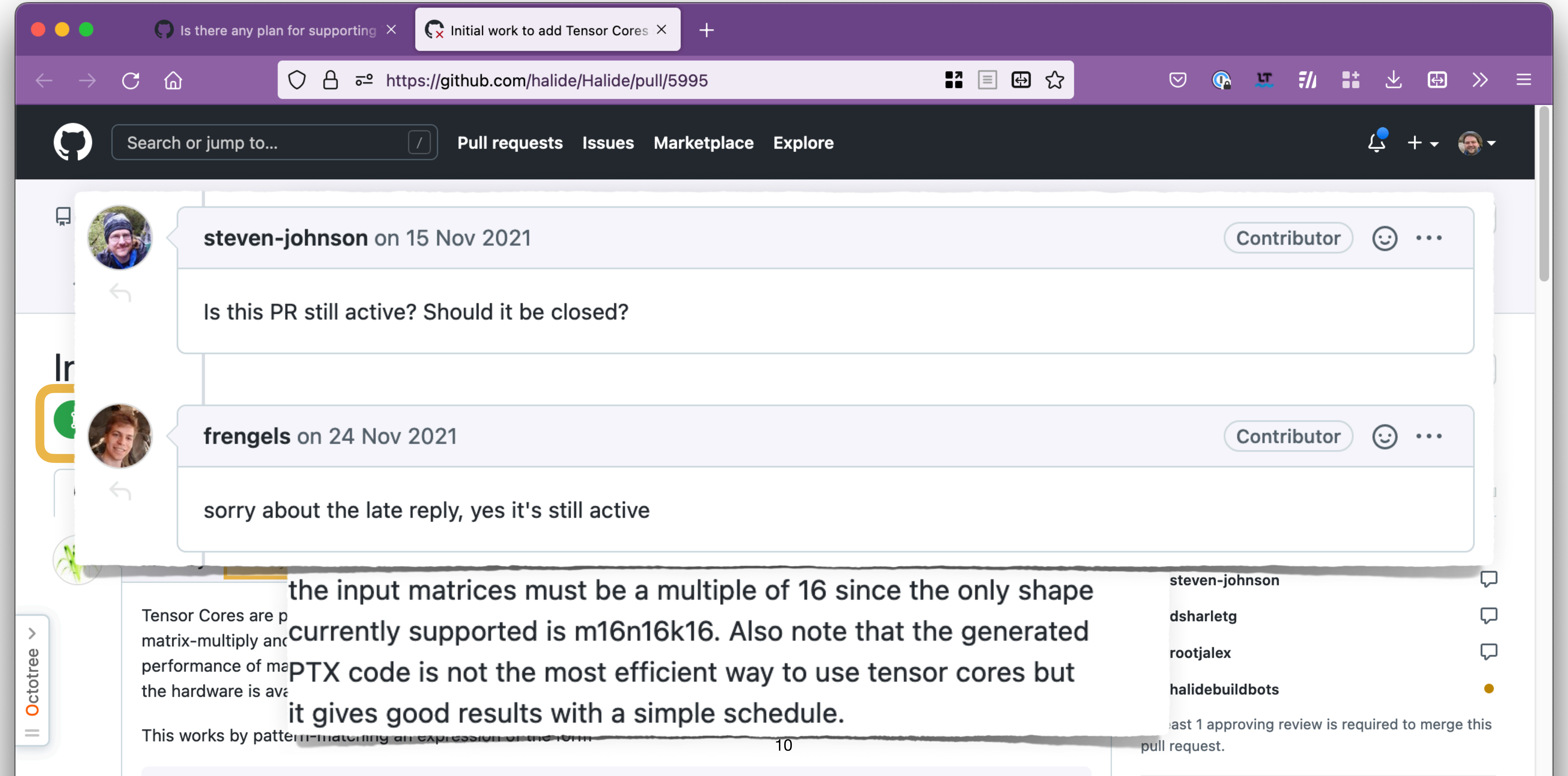
Adding support for Tensor Cores in Halide?



Adding support for Tensor Cores in Halide?



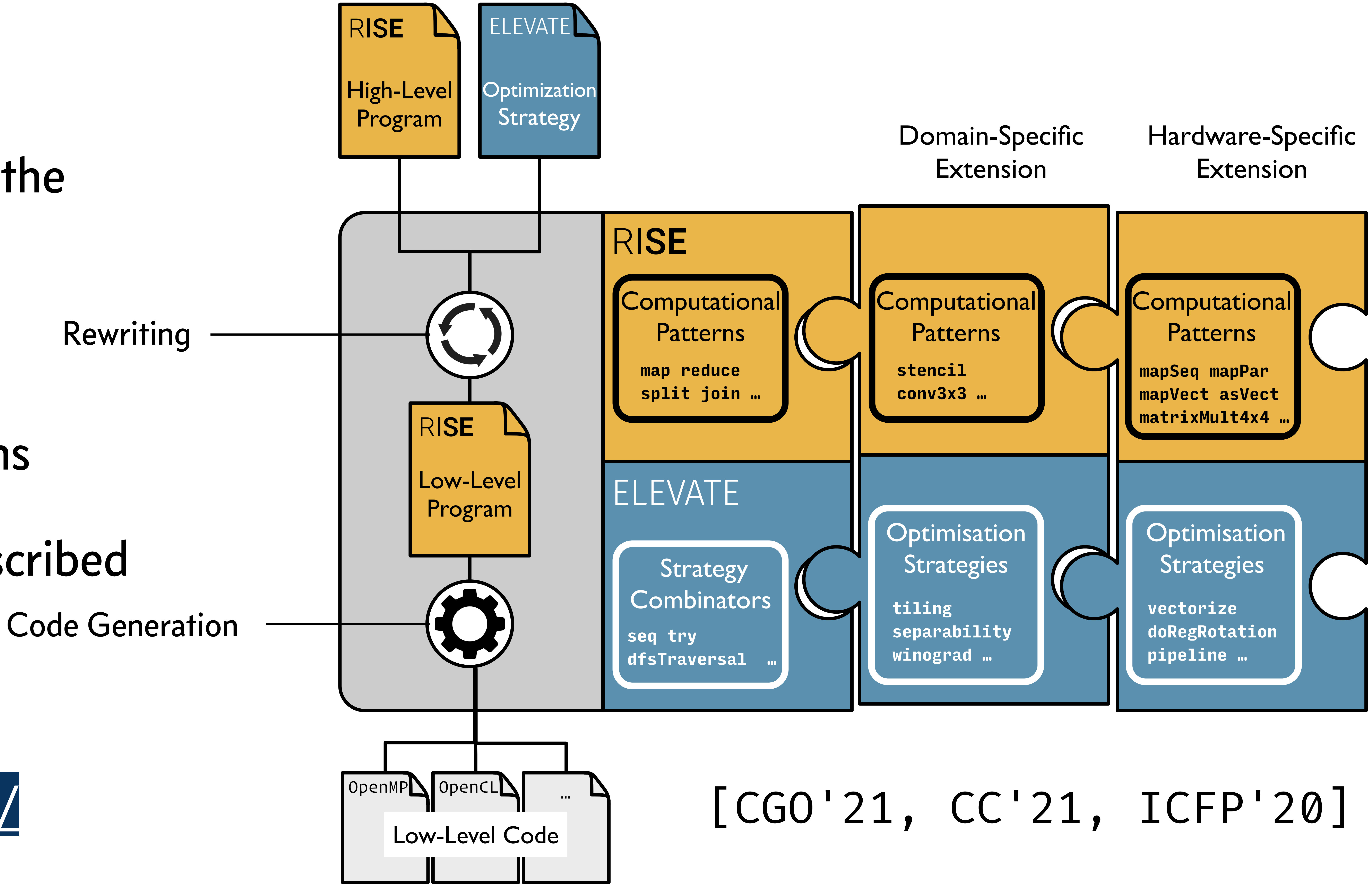
Adding support for Tensor Cores in Halide?



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

```
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

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

Michel Steuwer* Thomas Kœhler† 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

Translation

```
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

3. Add rewrite rules to enable Rewriting

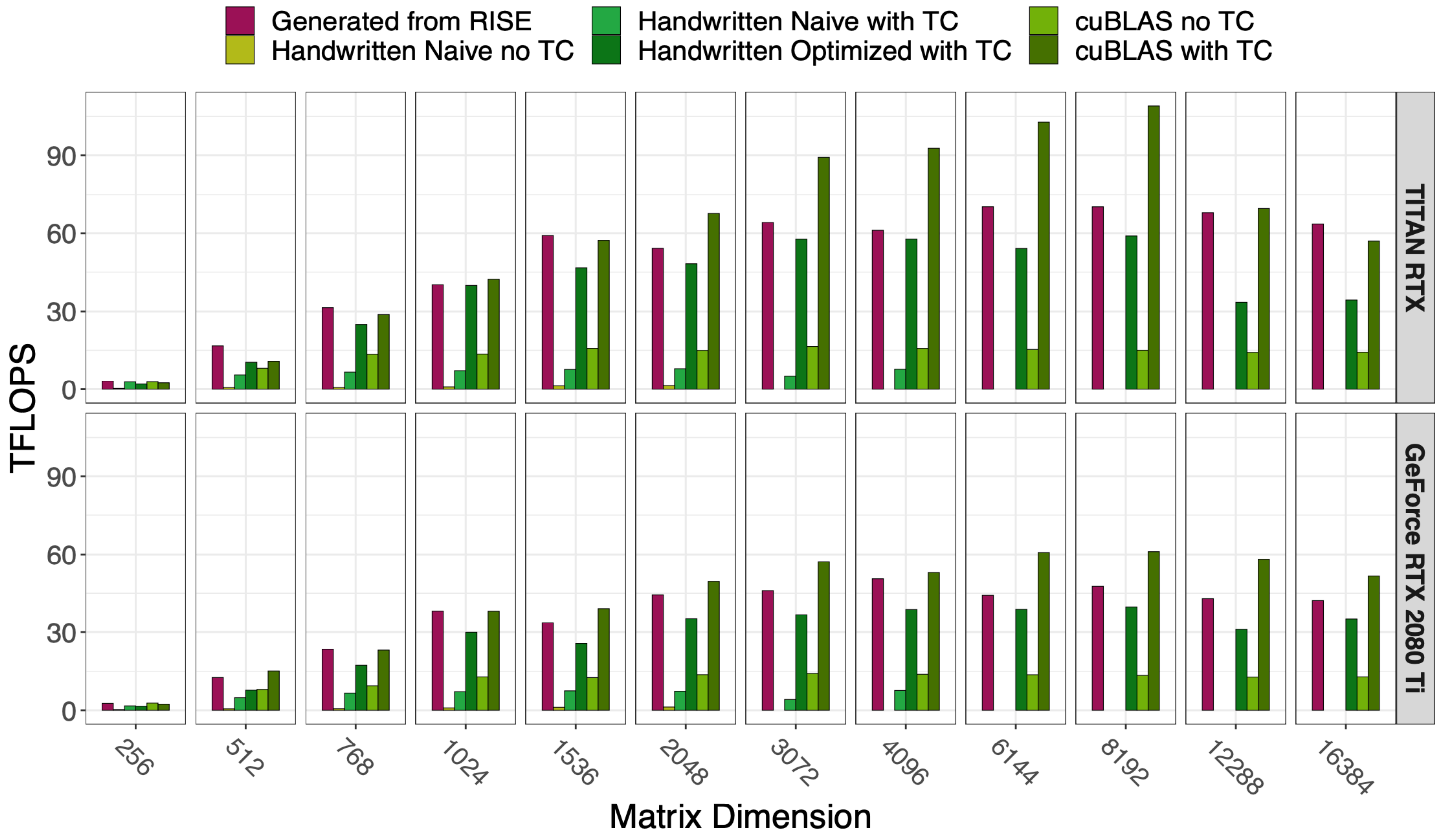
- Rewrite rules enable automatic exploitation of Tensor Cores
- Examples shows automatic use of Tensor Cores for high-level matrix multiplication code
- Rewrite rules can be applied *automatically* [GPGPU'16, ICFP'15], *manually* [ICFP'20], or *guided* [arXiv:2111.13040].

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

↧

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

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

- RI**SE** 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 https://michel.steuwer.info/ https://rise-lang.org/