

Fireiron - A Scheduling Language for GPUs.

Vinod Grover | Dec 5, 2019



# Acknowledgments

Joint work with :

- Bastian Hagedorn
- Sam Elliott
- Henrik Barthels
- Ras Bodik

And contributions from many others at NVIDIA.

# OVERVIEW

## High Performance DSL for linear algebra on GPUs

A hierarchical scheduling language based on Halide and TVM

- designed to express GPU optimizations for maximum performance

Can directly represent elements of

- storage hierarchy
  - registers, fragments, shared memory
- compute hierarchy
  - threads, warps, blocks, kernels

Can reason about tensorcore and machine level operations.

Suitable for auto-scheduling and auto-tuning

# DECOMPOSING MATMUL

## Exploiting Hierarchical Structure of GPU Kernels

```

1 __global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3   __shared__ float ASH[128][8], BSH[8][128];
4   float ARF[8][1], BRP[1][8], CRF[8][8];
5
6   iBlock ← 128 * blockIdx.x;
7   jBlock ← 128 * 8 * blockIdx.y;
8
9   CRF ← 0;
10
11   for (k ← 0; k < K / 8; k++) {
12
13     GlbToSh(A → ASH (8×128), start at (iBlock, jBlock))
14     GlbToSh(B → BSH (128×8), start at (iBlock, jBlock))
15     __syncthreads();
16
17     iWarp ← iBlock + warpIdx.x * 64;
18     jWarp ← jBlock + warpIdx.y * 32;
19
20     iThread ← iWarp + threadIdx.x * 8;
21     jThread ← jWarp + threadIdx.y * 8
22
23     for (kk ← 0; kk < 8; kk++)
24
25       ShToPvt(ASH → ARF (8×1), start at (iThread, jThread))
26       ShToPvt(BSH → BRP (1×8), start at (iThread, jThread))
27
28       for (i ← 0; i < 8; i++)
29         for (j ← 0; j < 8; j++)
30
31           CRF[i][j] += ARF[i][0] * BRP[0][j];
32
33       endfor
34     endfor
35
36   endfor
37
38   PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
39
40 } // end kernel

```

implements

sizes of matrices

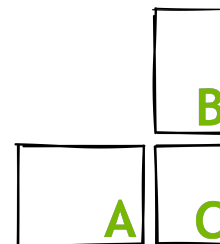
location of A, B and C

responsible level of compute hierarchy

MatMul(M,N,K)(GL,GL,GL)(Kernel)

Describing the problem this box implements

Hierarchical Structure:  
Original Problem is **decomposed** into  
“smaller” instances of the same type of problem



# DECOMPOSING MATMUL

## Exploiting Hierarchical Structure of GPU Kernels

```

1  __global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3  __shared__ float ASH[128][8], BSH[8][128];
4  float ARF[8][1], BRF[1][8], CRF[8][8];
5
6  iBlock ← 128 * blockIdx.x;
7  jBlock ← 128 * blockIdx.y;
8
9  CRF ← 0;
10
11  for (k ← 0; k < K / 8; k++) {
12
13    GlbToSh(A → ASH (8×128), start at (iBlock, jBlock))
14    GlbToSh(B → BSH (128×8), start at (iBlock, jBlock))
15    __syncthreads();
16
17    iWarp ← iBlock + warpIdx.x * 64;
18    jWarp ← jBlock + warpIdx.y * 32;
19
20    iThread ← iWarp + threadIdx.x * 8;
21    jThread ← jWarp + threadIdx.y * 8
22
23    for (kk ← 0; kk < 8; kk++)
24
25      ShToPvt(ASH → ARF (8×1), start at (iThread, jThread))
26      ShToPvt(BSH → BRF (1×8), start at (iThread, jThread))
27
28      for (i ← 0; i < 8; i++)
29        for (j ← 0; j < 8; j++)
30
31          CRF[i][j] += ARF[i][0] * BRF[0][j];
32
33      endfor
34    endfor
35
36  endfor
37
38  PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
39
40
41
42 } // end kernel

```

implements

sizes of matrices

location of A, B and C

responsible level of compute hierarchy

MatMul(M,N,K)(GL,GL,GL)(Kernel)

MatMul(128,128,K)(GL,GL,GL)(Block)

MatMul(128,128,K)(GL,GL,RF)(Block)

MatMul(128,128,8)(GL,GL,RF)(Block)

MatMul(128,128,8)(SH,SH,RF)(Block)

MatMul(64,32,8)(SH,SH,RF)(Warp)

MatMul(8,8,8)(SH,SH,RF)(Thread)

MatMul(8,8,1)(SH,SH,RF)(Thread)

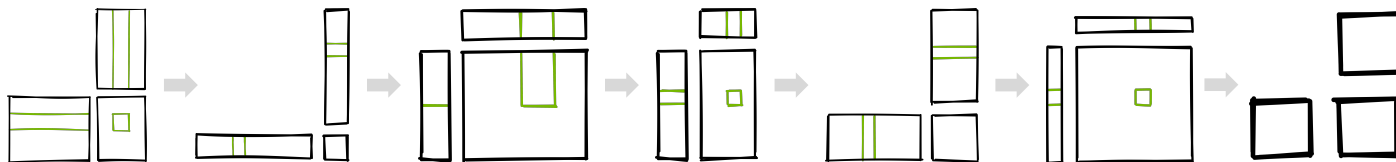
MatMul(8,8,1)(RF,RF,RF)(Thread)

MatMul(1,1,1)(RF,RF,RF)(Thread)

Describing the problem this box implements

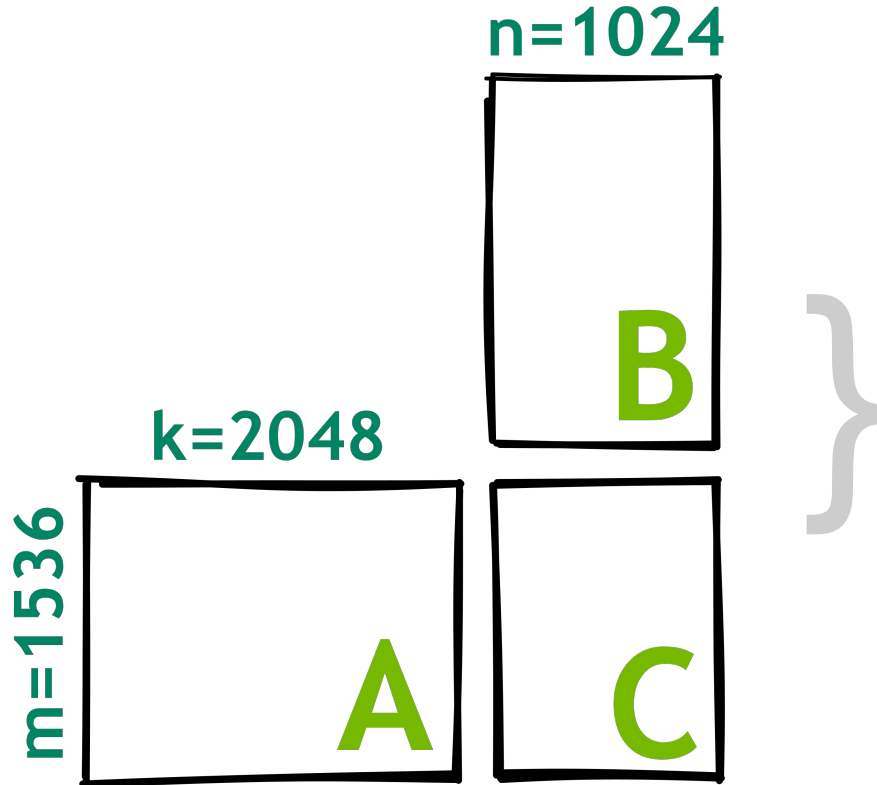
Hierarchical Structure:

Original Problem is **decomposed** into “smaller” instances of the same type of problem



# INTRODUCTION

## GEMM Spec(ification)



Specs define the current problem to optimize

### Fireiron MatMul Spec

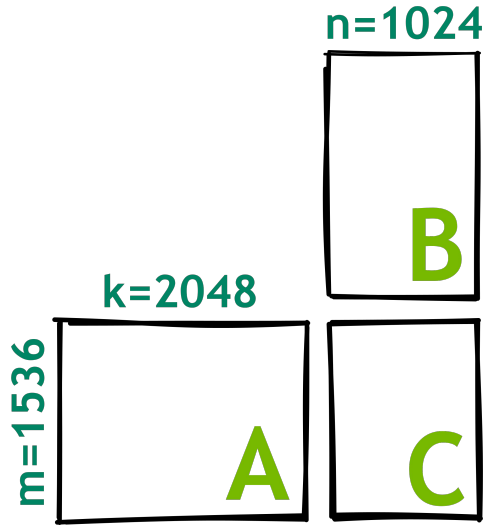
```
MatMul(Kernel,  
      A: Matrix(1536,2048,GL,FP32,RowMajor),  
      B: Matrix(2048,1024,GL,FP32,ColMajor),  
      C: Matrix(1536,1024,GL,FP32,ColMajor))
```

and contain enough information to fully describe it

*Idea: A programmer should be able to provide a valid implementation for a given spec!*

# INTRODUCTION

## Working with Specs



### Fireiron MatMul Spec

```
MatMul(Kernel,  
  A: Matrix(1536,2048,GL,FP32,RowMajor),  
  B: Matrix(2048,1024,GL,FP32,ColMajor),  
  C: Matrix(1536,1024,GL,FP32,ColMajor))
```

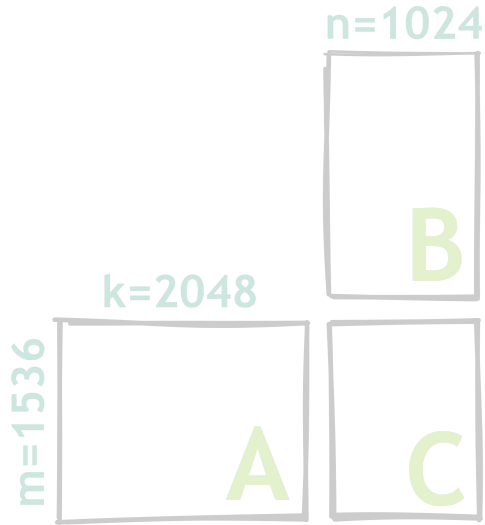
**Goal:** Generate high-performance MatMul Kernel  
-> We start with Kernel-level Spec

Given a Spec, you can:

- a) Provide a handwritten microkernel, or
- b) Arrive at an *executable* Spec, or
- c) *Decompose* it into a “smaller” spec

# INTRODUCTION

## Working with Specs



### Fireiron MatMul Spec

```
MatMul(Kernel,  
  A: Matrix(1536,2048,GL,FP32,RowMajor),  
  B: Matrix(2048,1024,GL,FP32,ColMajor),  
  C: Matrix(1536,1024,GL,FP32,ColMajor))
```

Goal: Generate high-performance MatMul Kernel  
-> We start with Kernel-level Spec

Given a Spec, you can:

- a) Provide a handwritten microkernel, or
- b) Arrive at an *executable* Spec, or
- c) **Decompose** it into a “smaller” spec



# DECOMPOSITIONS

## Halide-like transformations constructing the IR

Every *Decomposition*:

1. is a function: *Spec* -> *Spec* (returning a “smaller” subspec)
2. provides a *partial implementation* to our code generator

Two Main Decompositions:

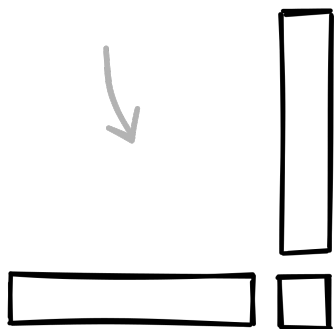
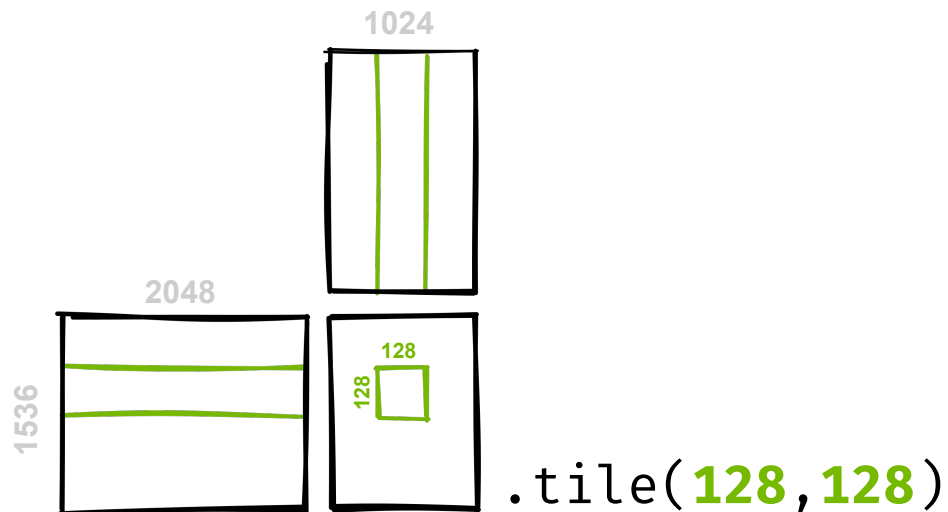
- **.tile(m,n)** - enables descending the compute-hierarchy
- .load(matrix, loc, impl) - enables descending the memory hierarchy

We allow to define operation-specific Decompositions:

- **.split(k)**
- .epilog(...)
- ...

# DESCENDING THE COMPUTE HIERARCHY

`.tile(m,n)`



## Current Spec

```
MatMul(Kernel,  
  A:Matrix(1536,2048,GL,FP32,RowMajor),  
  B:Matrix(2048,1024,GL,FP32,ColMajor),  
  C:Matrix(1536,1024,GL,FP32,ColMajor))
```

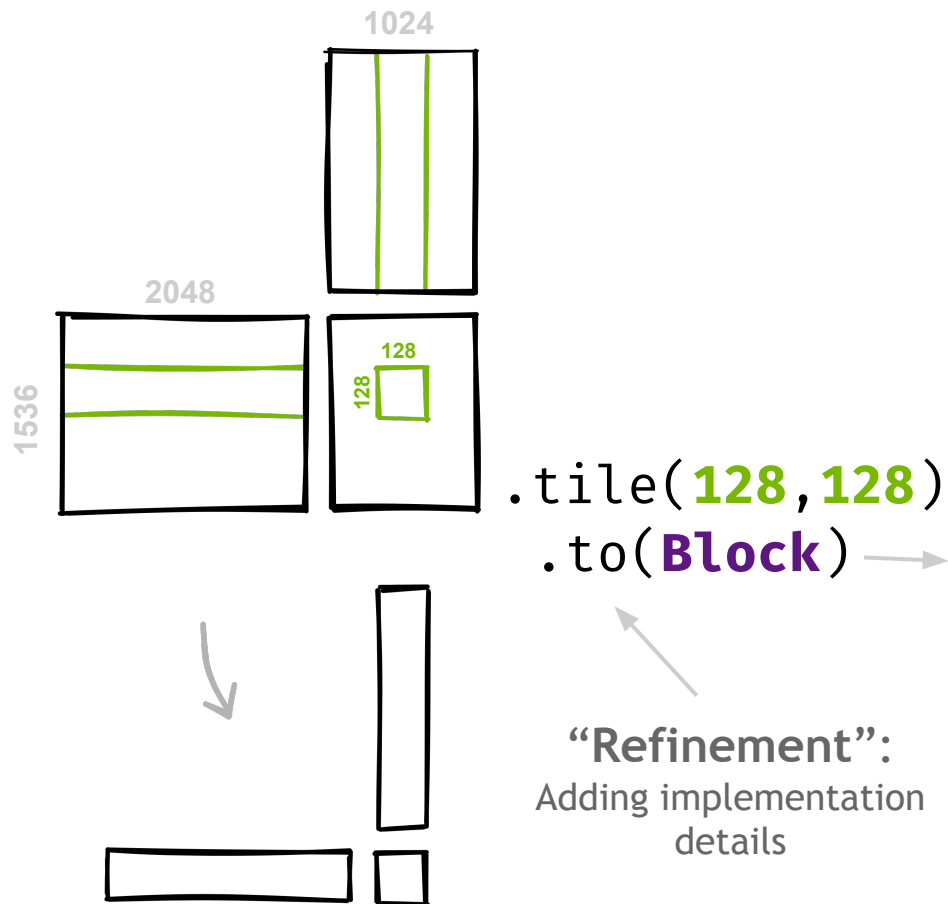


## New Spec

```
MatMul(Kernel,  
  A:Matrix(128,2048,GL,FP32,RowMajor),  
  B:Matrix(2048,128,GL,FP32,ColMajor),  
  C:Matrix(128,128,GL,FP32,ColMajor))
```

# DESCENDING THE COMPUTE HIERARCHY

`.tile(m,n)`

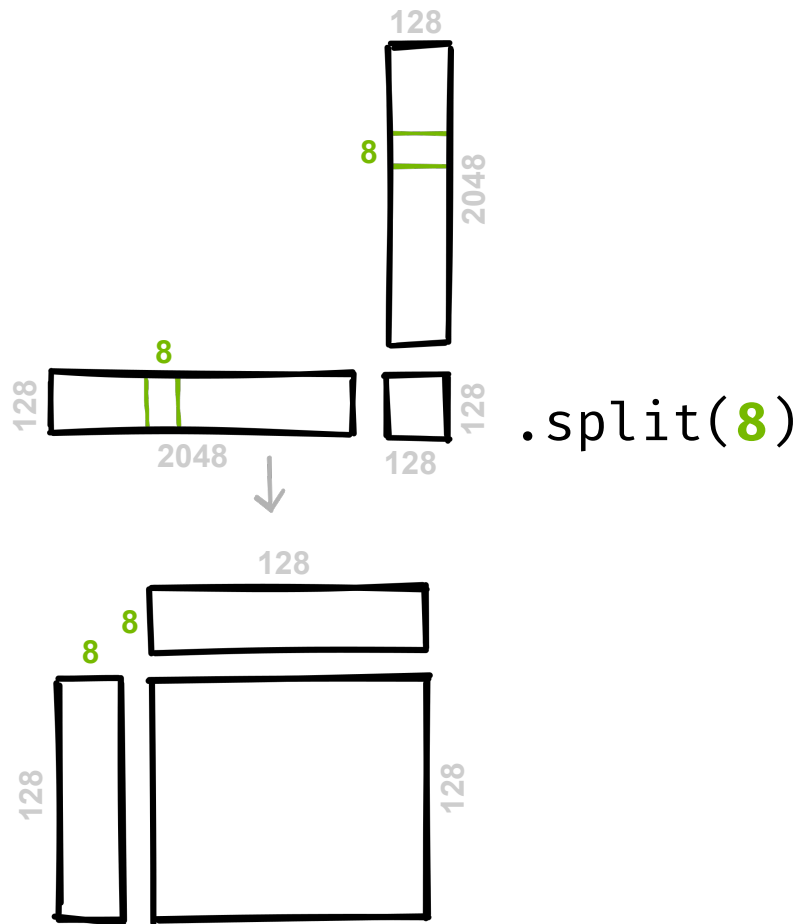


```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {  
  
    MatMul(M,N,K)(GL,GL,GL)(Kernel)  
  
}
```

```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {  
    iBlock ← 128 * blockIdx.x;  
    jBlock ← 128 * blockIdx.y;  
  
    MatMul(128,128,K)(GL,GL,GL)(Block)  
  
}
```

# OUTER PRODUCT BLOCKED GEMM

*.split(kBlock)*



## Current Spec

```
MatMul(Block,  
  A:Matrix(128,2048,GL,FP32,RowMajor),  
  B:Matrix(2048,128,GL,FP32,ColMajor),  
  C:Matrix(128 ,128,GL,FP32,ColMajor))
```

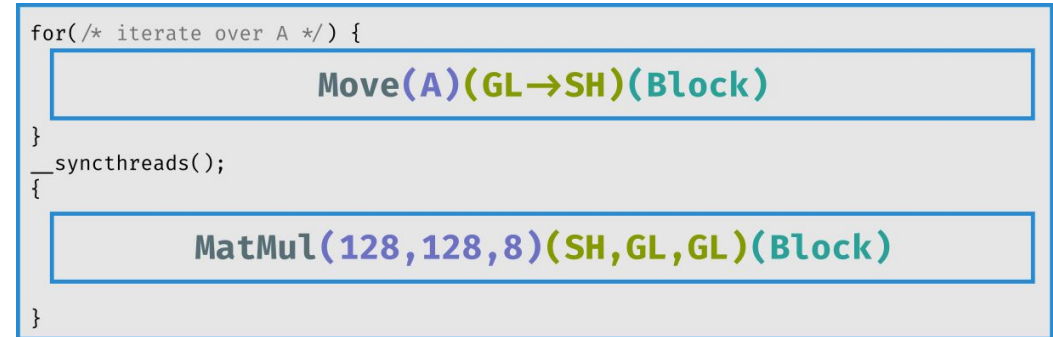
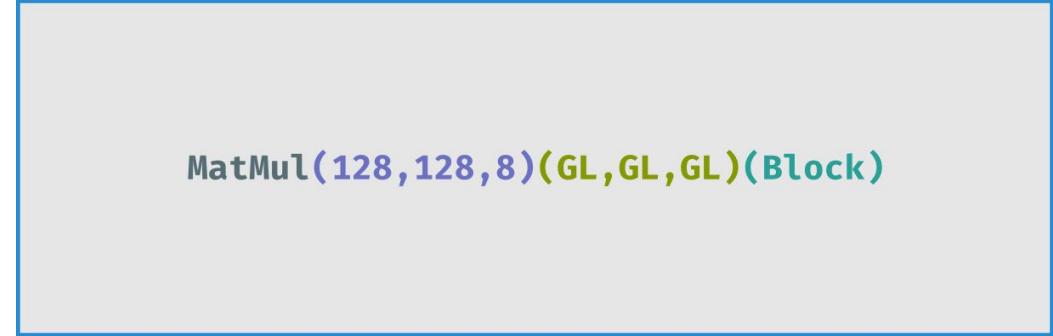
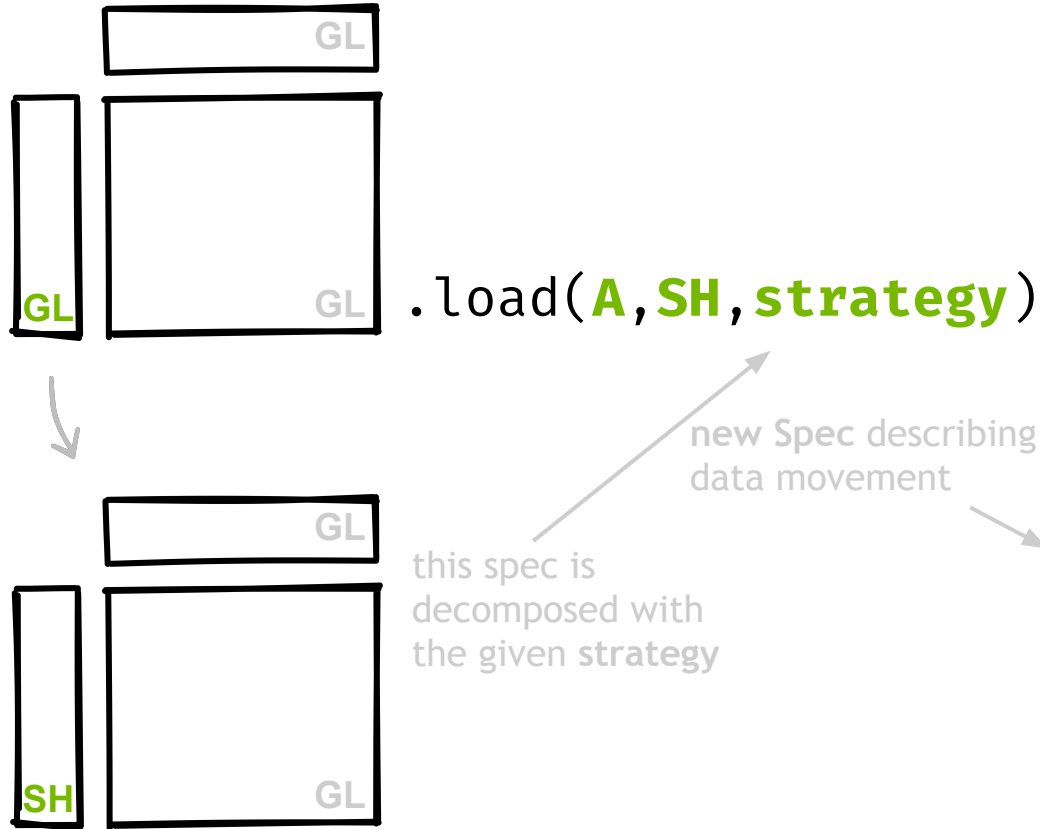


## New Spec

```
MatMul(Block,  
  A:Matrix(128, 8,GL,FP32,RowMajor),  
  B:Matrix(8 ,128,GL,FP32,ColMajor),  
  C:Matrix(128,128,GL,FP32,ColMajor))
```

# DESCENDING THE MEMORY HIERARCHY

*.load(Matrix, Location, Strategy)*



# WMMA IN FIREIRON

## adding support for CUDA's WMMA API

### Programmatic Access to Tensor Cores in CUDA 9.0

Access to Tensor Cores in devices via CUDA 9.0 is available as a preview feature. This means that the data structures, APIs and code described in this section are subject to change in future CUDA releases.

While cuBLAS and cuDNN cover many of the potential uses for Tensor Cores, you can also program them directly in CUDA C++. Tensor Cores are exposed in CUDA 9.0 via a set of functions and types in the `<nvcc>::wmma` namespace. These allow you to load or initialize values into the special format required by the tensor cores, perform matrix-multiply-accumulate (MMA) steps, and store values back out to memory. During program execution multiple Tensor Cores are used concurrently by a full warp. This allows the warp to perform a 16x16 MMA at very high throughput (Figure 5).

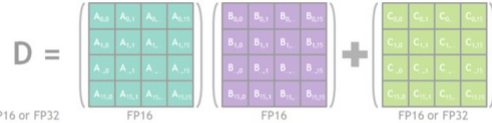


Figure 5: A warp performs  $D=A*B+C$  where A, B, C and D are 16x16 matrices. (Note the change in numbering from Figure 1: multiple Tensor Core operations are combined by the WMMA API to perform 16x16 matrix-multiply and accumulate operations.)

Let's go through a simple example that shows how you can use the WMMA API to perform matrix-multiply and accumulate operations. This example is not tuned for high performance and might not get better performance, check out the `CudaTensorFlow` project for more information.

#### Headers and Namespaces

The WMMA API is contained within the `<wmma>` header. In the code, we'll just be using the `<nvcc>` namespace, so we'll just be using the `<nvcc>` namespace.

```
#include <wmma>
using namespace nvcc;
```

#### Declarations and Initialization

The full GEMM specification allows the algorithm to be used with either row-major or column-major data. In this example, we'll assume that neither A nor B are transposed, and that the strategy we'll employ is to have a single warp responsible for the entire operation. This means we can effectively tile the warp across the two dimensions.

```
// The only dimensions currently supported by WMMA
const int WMMA_M = 16;
const int WMMA_N = 16;
const int WMMA_K = 16;
```

```
_global_ void wmma_example(half *a, half *b, float *c,
                           int M, int N, int K,
                           float alpha, float beta)
```

```
{
    // Loading dimensions. Packed with no transpositions.
    int lda = M;
    int ldb = K;
    int ldc = N;
```

```
// Tile using a 2D grid
int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
int warpN = (blockIdx.y * blockDim.y + threadIdx.y);
```

Before the MMA operation is performed the operand matrices must be represented in the registers of the GPU. As an MMA is a warp-wide operation these registers are distributed amongst the threads of a warp with each thread holding a *fragment* of the overall matrix. The mapping between individual matrix parameters to their fragments is shown, so your program should not make any assumptions about it.

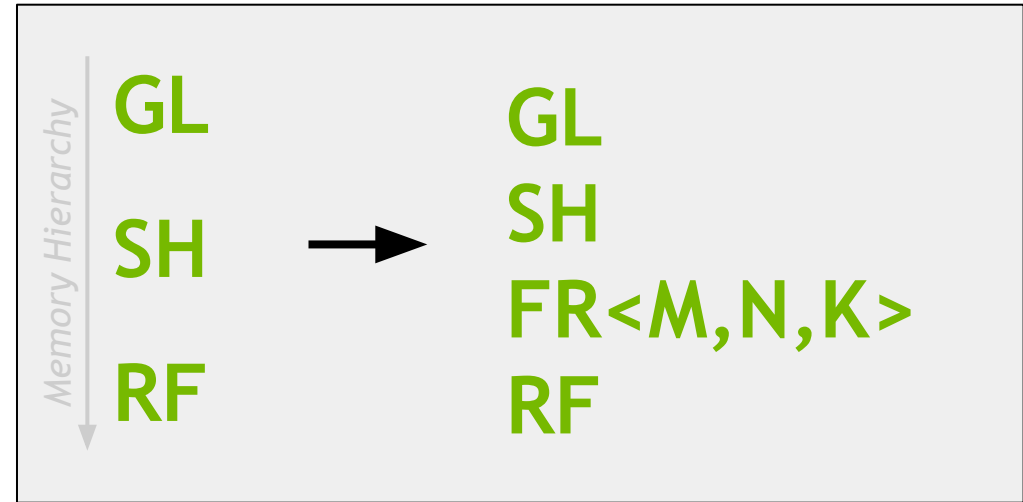
In CUDA a fragment is a 4x4 grid of 16 threads. Each thread holds a fragment of the overall matrix. The mapping between individual matrix parameters to their fragments is shown, so your program should not make any assumptions about it.

```
// Declare the fragments
wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
```

#### // Declare the fragments

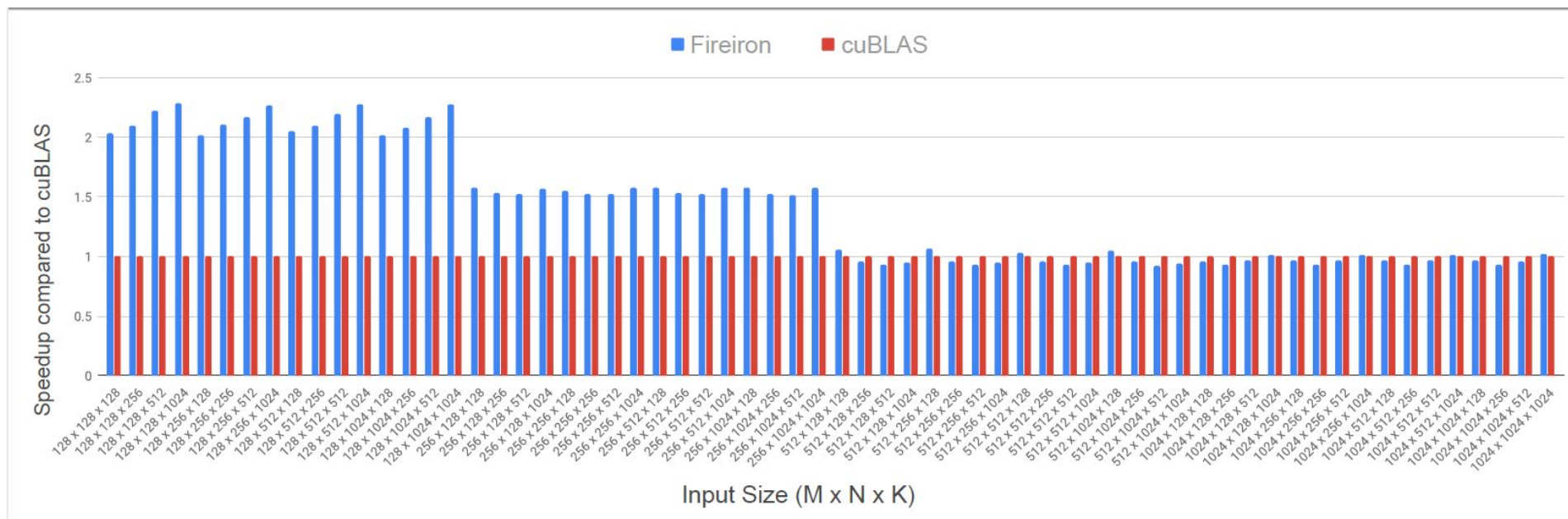
```
wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
```

“Before the MMA operation is performed the operand matrices must be represented in the registers of the GPU. As an MMA is a warp-wide operation these registers are distributed amongst the threads of a warp with each thread holding a *fragment* of the overall matrix.”



### Updating Fireiron's Memory Hierarchy

# fp16 performance on Volta



QUESTIONS?