FIREIRON: A SCHEDULING LANGUAGE FOR HIGH-PERFORMANCE LINEAR ALGEBRA ON GPUS

BASTIAN HAGEDORN | SAM ELLIOTT | HENRIK BARTHELS | RAS BODIK | VINOD GROVER
University of Münster, b.hagedorn@wwu.de | lowRISC | RWTH Aachen University | Uni of Washington | NVIDIA

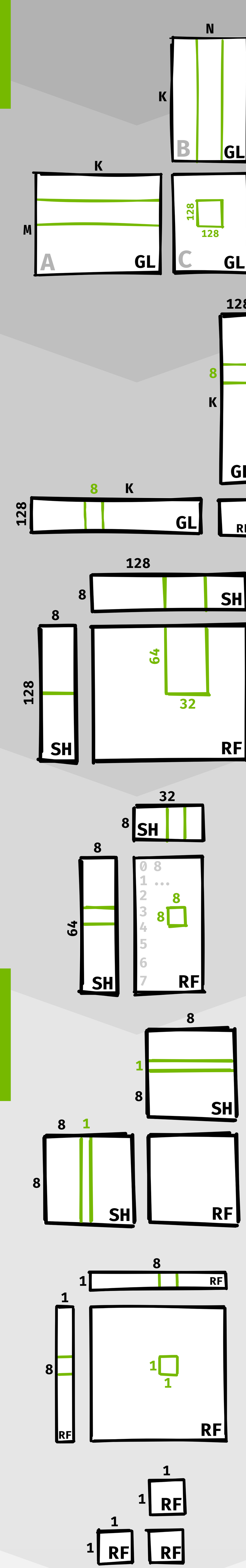
01 EXPLOITING HIERARCHICAL STRUCTURE

IN HIGH-PERFORMANCE LINEAR ALGEBRA GPU IMPLEMENTATIONS

Most high-performance kernels for GPUs are written in a hierarchical style. The original problem to be computed is decomposable into smaller sub-problems of the same kind. These sub-problems are then assigned to and computed by the different levels of the compute hierarchy. The figure below visualizes this observation using a simple matrix multiplication kernel. Step by step, the matrix multiplication is decomposed into a hierarchy of tiles and data is transferred to lower levels of the memory hierarchy until eventually every thread computes a single FMA instruction. Here, FMA can be viewed as a matrix multiplication of matrices which only contain a single element.

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

implements → *sizes of matrices* / *location of A, B and C* / *responsible level of compute hierarchy*

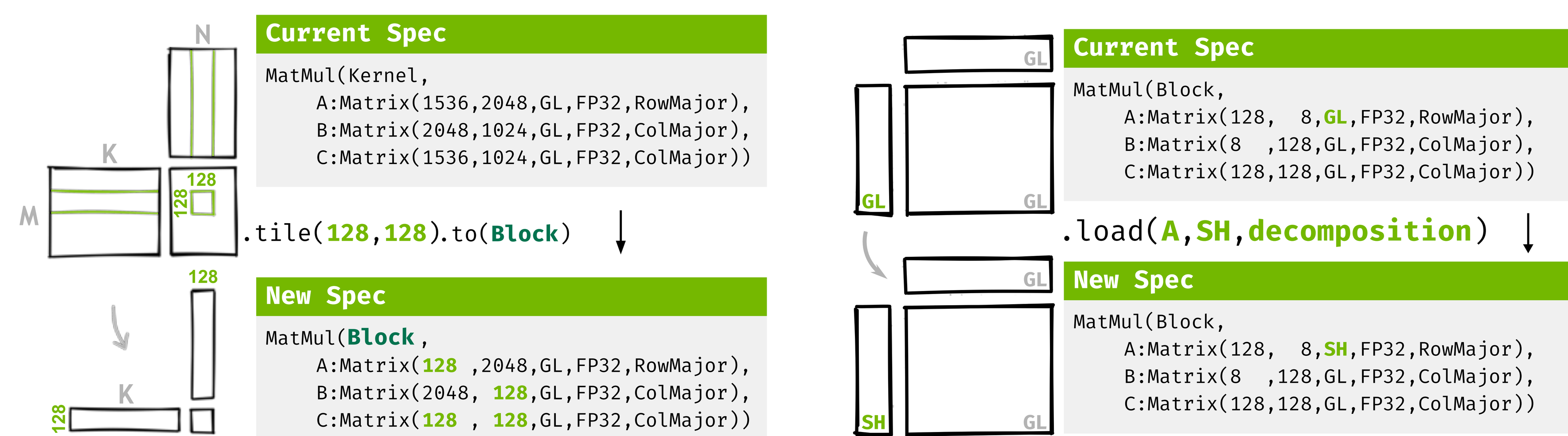


02 FIREIRON: A SCHEDULING LANGUAGE

EXPRESS COMPLEX IMPLEMENTATIONS AS COMPOSITIONS OF SIMPLE PRIMITIVES

Fireiron introduces two main concepts: Specifications and Decompositions. A Specification (spec) is a data-structure describing the computation to implement. A spec contains enough information such that a programmer would be able to manually write an implementation. This especially entails that a spec keeps track of the shapes, locations and storage layouts of its input and output tensors, as well as which level of the compute hierarchy (i.e., Kernel, Block, Warp or Thread) is responsible for computing this operation.

A Decomposition describes how to (partially) implement a given spec. More specifically, a decomposition is a function Spec → Spec which, given a spec, returns a new spec that represents the smaller decomposed sub-problem. Fireiron provides two main decompositions, tile and load, which allow implementations to use the compute and memory hierarchy of a GPU.



03 DECOMPOSING MATRIX MULTIPLICATION

GRADUALLY DESCENDING THE COMPUTE AND MEMORY HIERARCHY

KERNEL-LEVEL

In order to generate a high-performance Matrix Multiplication kernel with Fireiron, we start with a kernel-level MatMul specification...

```
MatMul(ComputeHierarchy: Kernel,
       A: Matrix((M x K), FP16, GL, ColMajor),
       B: Matrix((K x N), FP16, GL, ColMajor),
       C: Matrix((M x N), FP16, GL, ColMajor))
.tile(128,128).to(Block)
```

Storage Layout — Location in Memory Hierarchy — Shape — Scalar Data Type

BLOCK-LEVEL

MatMul-specific decompositions: In order to express advanced decompositions for storing computed results, and accumulating intermediate results in registers, we introduce a new decomposition .epilog(l,i,d). Similar to load, i and d are decompositions. Here, i describes the initialization of a new buffer in location l (usually in registers) used for accumulating the results. The .split(k) decomposition is used to create tiles in the K-dimension. This allows efficient use of shared memory.

```
.epilog(RF, init, store)
.split(8)
.load(A, SH, contiguousPrefetch)  // == contiguousPrefetch = Move
.load(B, SH, crosswisePrefetch)   //      .tile(128,1).to(Warp)
.tile(64,32).to(Warp)e            //      .tile(64,1).unroll
                                  //      .tile(2,1).to(Lane)
                                  //      .done
```

These are Fireiron-Decompositions too. We can reuse efficient (partial) implementations!:

Move: Another Spec representing data movement between levels of the memory hierarchy. **.unroll**: A Refinement, more in a second....

WARP-LEVEL

```
.tile(8,8).to(Thread)
.layout(ColMajor)
```

We also support **Tensor Cores**! In order to use them, computations must be decomposed to **warp-level WMMA** or **quad-pair mma** specifications

Refinements allow fine-grained control in decompositions. For example, **.unoll** is used to indicate loops shall be unrolled during code generation, and **.layout** is used to specify the mapping between tiles and the elements of the current compute hierarchy.

THREAD-LEVEL

```
.split(1)
.load(A, RF, aToRF)
.load(B, RF, bToRF)
.tile(1,1)
.done
```

04 PERFORMANCE

GENERATING FAST KERNELS COMPETITVE TO cuBLAS

We evaluate high-performance decompositions expressing similar implementations as provided in cuBLAS. We are targeting the Volta architecture and make heavy use of Tensor Cores. We achieve speedups due to smaller tile size selections.