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

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

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

**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

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))
INTRODUCTION

Working with Specs

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

\[ \text{.tile}(m,n) \]

**Current Spec**

\[
\text{MatMul}(\text{Kernel}, \\
A: \text{Matrix}(1536, 2048, \text{GL, FP32, RowMajor}), \\
B: \text{Matrix}(2048, 1024, \text{GL, FP32, ColMajor}), \\
C: \text{Matrix}(1536, 1024, \text{GL, FP32, ColMajor}))
\]

**New Spec**

\[
\text{MatMul}(\text{Kernel}, \\
A: \text{Matrix}(128, 2048, \text{GL, FP32, RowMajor}), \\
B: \text{Matrix}(2048, 128, \text{GL, FP32, ColMajor}), \\
C: \text{Matrix}(128, 128, \text{GL, FP32, ColMajor}))
\]
DESCENDING THE COMPUTE HIERARCHY

\texttt{.tile(m,n)}

\texttt{.tile(128,128)}

\texttt{.to(Block)}

“Refinement”: Adding implementation details
OUTER PRODUCT BLOCKED GEMM

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))

.split(8)
DESCENDING THE MEMORY HIERARCHY

\texttt{.load(Matrix, Location, Strategy)}

\texttt{.load(A, SH, strategy)}

\texttt{MatMul(128, 128, 8)(GL, GL, GL)(Block)}

\texttt{for(\text{/\* iterate over A */}) { }

\texttt{Move(A)(GL\rightarrow SH)(Block) }

\texttt{\_sycnthreads();}

\texttt{MatMul(128, 128, 8)(SH, GL, GL)(Block) }

new Spec describing data movement

this spec is decomposed with the given strategy
WMMA IN FIREIRON

adding support for CUDA’s WMMA API

“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.”

```
// 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;
```
fp16 performance on Volta
QUESTIONS?