High Performance GPU Tensor Core Code Generation for Matmul Using MLIR

Navdeep Katel and Uday Bondhugula
PolyMage Labs & Indian Institute of Science

Outline

Motivation

- The state of the art in high-performance deep learning is driven by highly tuned libraries.
- Often hand-optimized and tuned by expert programmers using low-level languages and models with significant effort.
- Lot of effort may need to be repeated for similar hardware.

What can be done?

- Make the process modular, systematic and automatic as much as possible.
- Build a set of abstractions, transformations and optimizations to generate code **automatically**. Possible using MLIR.

Outline

GPUs: Hardware (Compute Units)

- GPU (Graphics Processing Unit) is a highly parallel specialized piece of hardware. Primarily Used for accelerating Deep Learning and Machine learning workloads. Volta -> Turing -> Ampere, are some recent architectures.
- It is a collection of Streaming Multiprocessors (SMs).
- SM is further partitioned into processing blocks.
- L1-D cache shared among all processing blocks.
- Each processing block has:-
  - Warp schedulers + Dispatch Units.
  - CUDA cores for general math instructions.
  - Tensor Cores for MMA instructions.
  - SFUs for special math functions like __log2f(), __sinf(), __cosf().
  - Dedicated LD/ST units.
  - A register file.
  - L0-I cache.

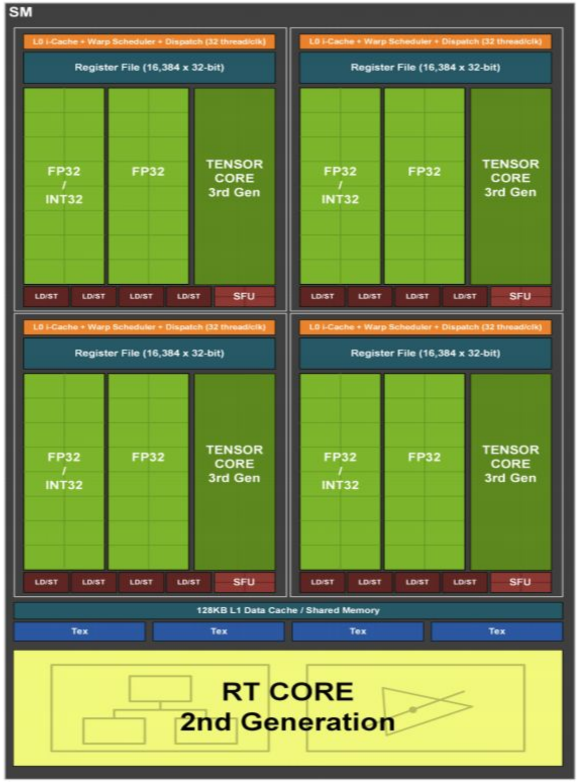

SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE 2nd Generation

Figure 6. SM on GA102(Ampere).
Fig. courtesy GA102 whitepaper.

GPUs: Hardware (Memory Hierarchy)

- GPU memory can be described as a 4-level memory hierarchy.
- Additionally, Each level has restrictions on what entity can access that memory.
- 4-Levels of the hierarchy:-
  - Registers.
  - L1-cache/Scratchpad memory.
  - L2-cache, shared among all the SMs.
  - Global memory (DRAM), Shared among all the SMs.
- L1-cache can be partitioned to have a programmer controlled scratchpad memory, called Shared Memory. Useful to store frequently accessed data.

GPUs: Tensor Cores

- Tensor cores are special units for matrix multiply accumulate.
- Performs a small matrix matrix multiplications in one cycle.
- Have very high performance as compared to regular CUDA cores.
- They operate on FP16 inputs, while accumulation can be in FP16 or FP32(new data types also supported now).

	Turing RTX 2080 Ti		**Ampere RTX 3090**	
	FP16	**FP32**	**FP16**	**FP32**
Non TC(TFLOPS)	28.5	14.2	35.6	35.6
TC(TFLOPS)	113.8	56.9	142	71

GPUs: Programming the Tensor Cores

- Tensor Cores can be be programmed/accessed in three ways:-
  - Use High level vendor libraries like cuBLAS, CuDNN.
    - Usually the fastest.
    - Limited support for operator fusion.
  - Program tensor cores using the (Warp Matrix Multiply Accumulate)WMMA API.
    - Provides utility functions for loading and storing matrix fragments.
    - Cannot totally avoid bank-conflicts.
    - Fused kernels can be implemented.
    - Offload the lowering to target instructions on NVIDIA's compiler.
  - Program the explicitly using **`mma_sync`** PTX instruction.
    - No utility functions for loading and storing matrices.
    - Different versions for devices of different compute capabilities.
    - Better control over bank conflicts by using swizzled layouts in shared memory.

Outline

GPUs: Programming Model

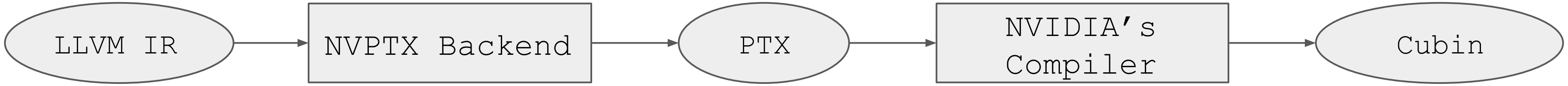
- Basic executing entity on a GPU is a thread.
- 32 threads are grouped together into a warp.
- A warp executes in a lock-step manner.
- Warps are further grouped into thread blocks. Bound to a single SM.
- Registers belonging to a thread are private to it.
- Threads in same thread block can communicate using the shared memory.
- Threads form different thread blocks need to communicate using the global memory.
- Synchronizations present at thread block and warp level.

Outline

- Motivation
- GPUs
  - Hardware
  - Programming Model
  - Compilation Path
- Matrix Multiplication on GPUs
- Code Generation for Tensor Core Matmul
  - Performance Guidelines
  - Pipeline for Efficient Codegen
- Results
  - Matmul
  - Fusion
- Conclusion, Collaborations and Future Directions

GPUs: Compilation Path

- Compilation path when using MLIR goes through LLVM.
- MLIR is converted into LLVM IR.
- LLVM NVPTX backend emits PTX code for given LLVM IR.
- PTX taken up by NVIDIA's assembler to convert it into SASS and subsequently CUBIN.



Outline

Matrix-Matrix Multiplication on GPUs

- Matmul in a general form is represented by the equation, $D = \boldsymbol{\alpha}.A.B + \boldsymbol{\beta}.C$, where α and $\boldsymbol{\beta}$ are scalars and dimensions of **A**, **B** and **C** are **M×K**, **K×N** and **M×N** respectively.
- Has O(n^3) computation on O(n^2) data. Roughly O(n) reuse.
- At the heart of many Deep learning models such as BERT.
- GPU vendor libraries like cuBLAS, usually provide the fastest implementations.
- Other works like CUTLASS achieve performance comparable/better than cuBLAS.
- CUTLASS highlights the recipe for efficient matmul on GPUs.

Matrix-Matrix Multiplication on GPUs

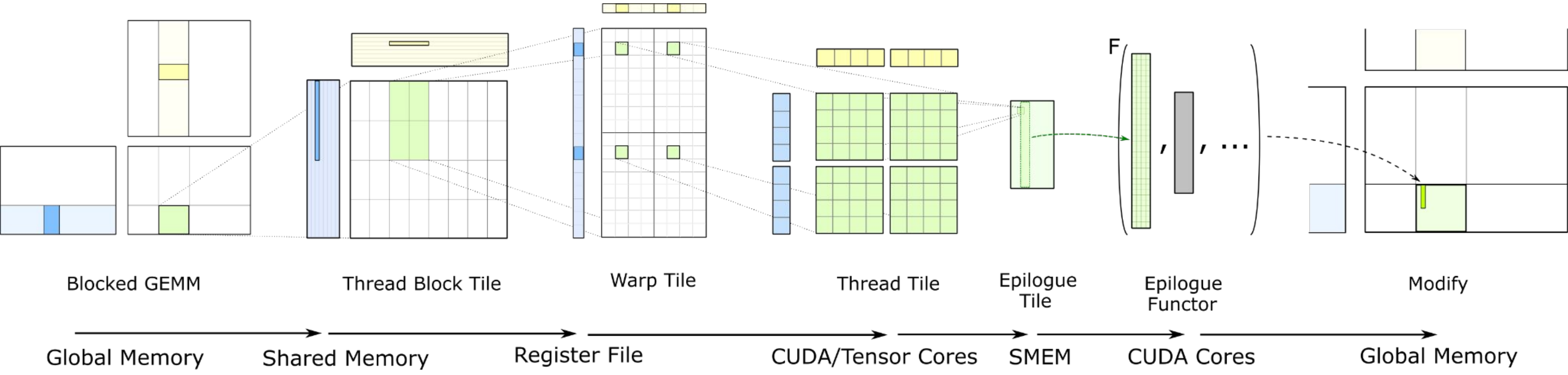


Figure 7. Recipe for efficient matmul on GPUs. ***Fig. courtesy CUTLASS***

Outline

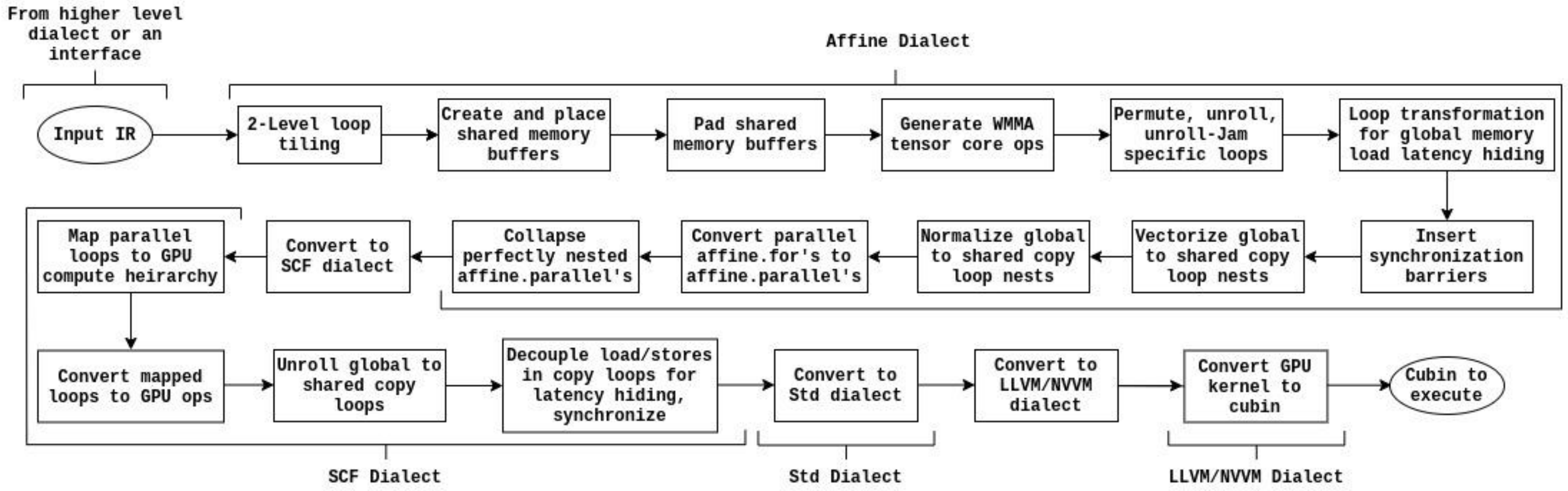
Code Generation for Tensor Core Matmul

Performance Guidelines:-

- Two-level tiling:
  - Thread Block Tiling
  - Warp Tiling
- Parallelize independent tasks, i.e. calculate independent tiles of the output matrix using different thread blocks.
- Efficient access to global memory by ensuring coalesced accesses.
- Efficient accesses to shared memory by reducing bank conflicts.
- Overlap memory transfers with compute, i.e. hide latency of global memory loads.
- Use vectorized memory accesses.

Outline

Code Generation for Tensor Core Matmul



From higher level dialect or an interface
Affine Dialect
Input IR
2-Level loop tiling
Create and place shared memory buffers
Pad shared memory buffers
Generate WMMA tensor core ops
Permute, unroll, unroll-Jam specific loops
Loop transformation for global memory load latency hiding
Insert synchronization barriers
Vectorize global to shared copy loop nests
Normalize global to shared copy loop nests
Convert parallel affine.for's to affine.parallel's
Collapse perfectly nested affine.parallel's
Convert to SCF dialect
Map parallel loops to GPU compute heirarchy
Convert mapped loops to GPU ops
Unroll global to shared copy loops
Decouple load/stores in copy loops for latency hiding, synchronize
Convert to Std dialect
Convert to LLVM/NVVM dialect
Convert GPU kernel to cubin
Cubin to execute
SCF Dialect
Std Dialect
LLVM/NVVM Dialect

Figure 8. MLIR Code generation Pipeline for tensor core matmul.

Code Generation for Tensor Core Matmul

WMMA Ops in MLIR:-

- WMMA ops in MLIR were introduced in gpu dialect.
- To complete the code generation path, ops also introduced in nvvm dialect.
- Ops model the WMMA intrinsics present in NVPTX backend.
- A supporting opaque type *gpu.mma_matrix*, for WMMA to operate on was also introduced.
- Ops introduced include *gpu.subgroup_mma_load_matrix*, *gpu.subgroup_mma_store_matrix*, and *gpu.subgroup_mma_compute*.

Code Generation for Tensor Core Matmul

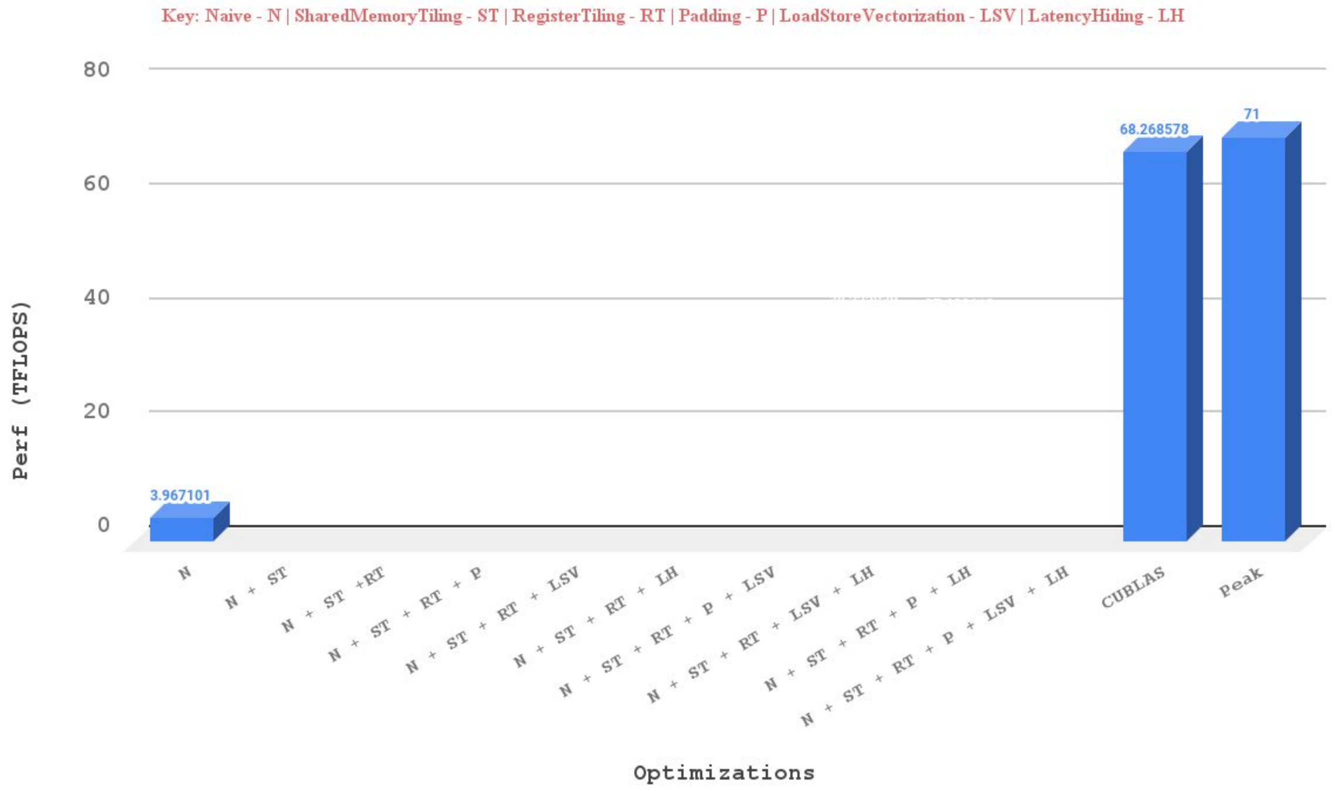
```
// Load operands for WMMA operation.

%a = gpu.subgroup_mma_load_matrix %A[%c0, %c0] {leadDimension = 1024 : index} : ←-

memref<1024x1024xf16> -> !gpu.mma_matrix<16x16xf16, "AOp">

...

// Perform the WMMA operation.

%res = gpu.subgroup_mma_compute %a, %b, %c : !gpu.mma_matrix<16x16xf16, "AOp">, ←-

!gpu.mma_matrix<16x16xf16, "BOp"> -> !gpu.mma_matrix<16x16xf32, "COp">

// Store back the result.

gpu.subgroup_mma_store_matrix %res, %C[%c0, %c0] {leadDimension = 1024 : index}: ←-

!gpu.mma_matrix<16x16xf32, "COp">, memref<1024x1024xf32>
```

Code Generation for Tensor Core Matmul

The beginning

```
affine.for %i = 0 to %M {
  affine.for %j = 0 to %N {
    affine.for %l = 0 to %K {
      %a = affine.load %gpu_A[%i, %l] : memref<8192x8192xf16>
      %b = affine.load %gpu_B[%l, %j] : memref<8192x8192xf16>
      %c = affine.load %gpu_C[%i, %j] : memref<8192x8192xf32>
      %aq = fpext %a : f16 to f32
      %bq = fpext %b : f16 to f32
      %p = mulf %aq, %bq : f32
      %co = addf %c, %p : f32
      affine.store %co, %gpu_C[%i, %j] : memref<8192x8192xf32>
    }
  }
}
```

Code Generation for Tensor Core Matmul

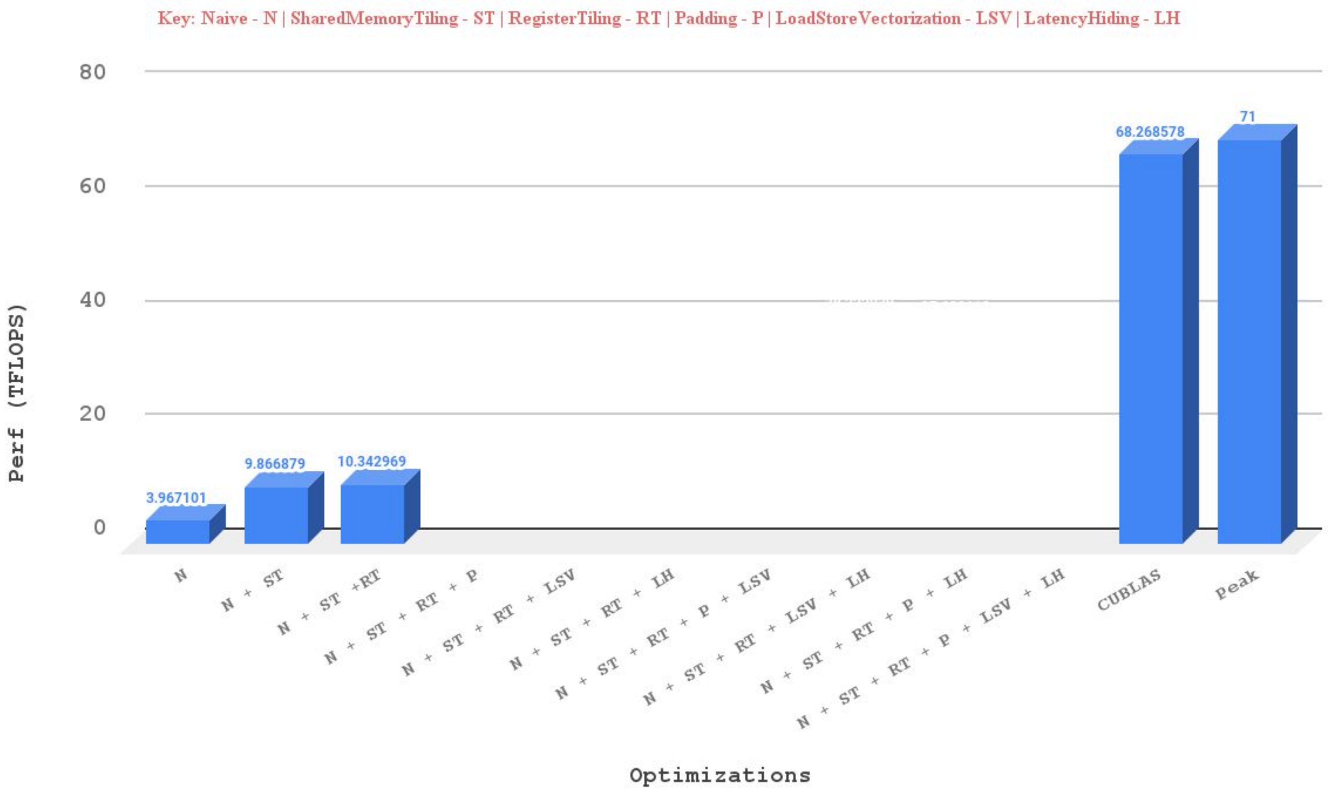


Code Generation for Tensor Core Matmul

Shared Memory and Register Blocking

- Matmul involves redundant accesses to the input matrices to calculate different elements of output matrices.
- Tile and distribute the computation among different thread blocks and warps. All proceed in parallel.
- Move the relevant tiles of input matrices into shared memory and registers.
- Redundant accesses are eliminated.
- Multi-level tiling using the affineLoopTiling, followed by affineDataCopyGenerate to create copy nests for global to shared data copy.

Code Generation for Tensor Core Matmul

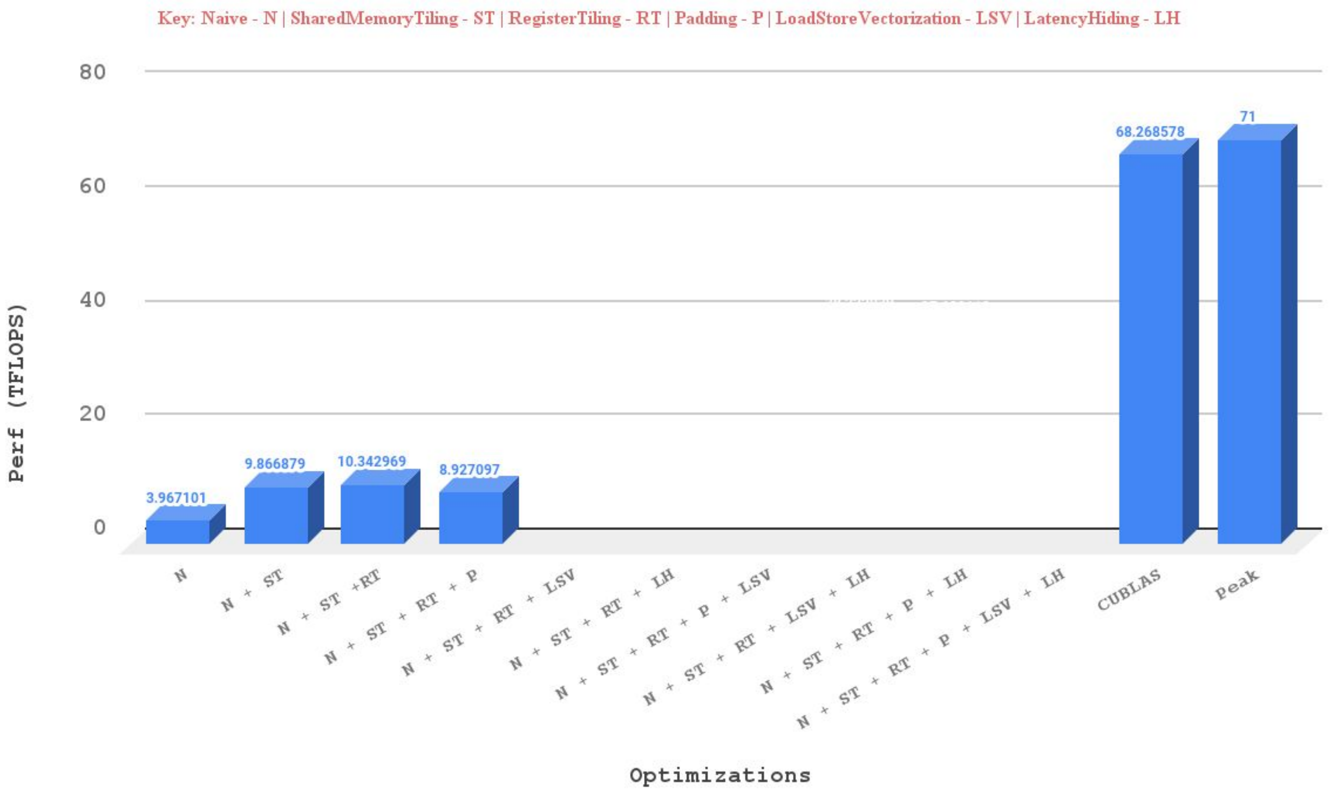


Code Generation for Tensor Core Matmul

Bank Conflicts and Padding

- Shared memory is arranged in banks, usually 32 banks each 4-byte wide.
- 32 Threads from a warp can access shared memory in parallel.
- Conflict occurs when two or more threads in the same warp access different 4-byte words in the same bank.
- The accesses in a conflict are serialized, hence reduction in bandwidth.
- Padding is an effective technique to reduce shared memory bank conflicts.
- Padding done by changing the leading dimension of shared memory memref from (*leadingDimension*) to (*leadingDimension + paddingFactor*).

Code Generation for Tensor Core Matmul



Code Generation for Tensor Core Matmul

Global to Shared Copy Vectorization:-

- Tiles of A and B being copied from global to shared to shared memory, element by element.
- Vector load-stores often help in utilizing the available bandwidth efficiently.
- Vector width upto 128-bit supported on NVIDIA GPUs.
- Experimenting with different vector widths, 32, 64, 128-bit revealed that 128-bit works the best.
- Vectorization utility borrowed from MLIRX.

Code Generation for Tensor Core Matmul

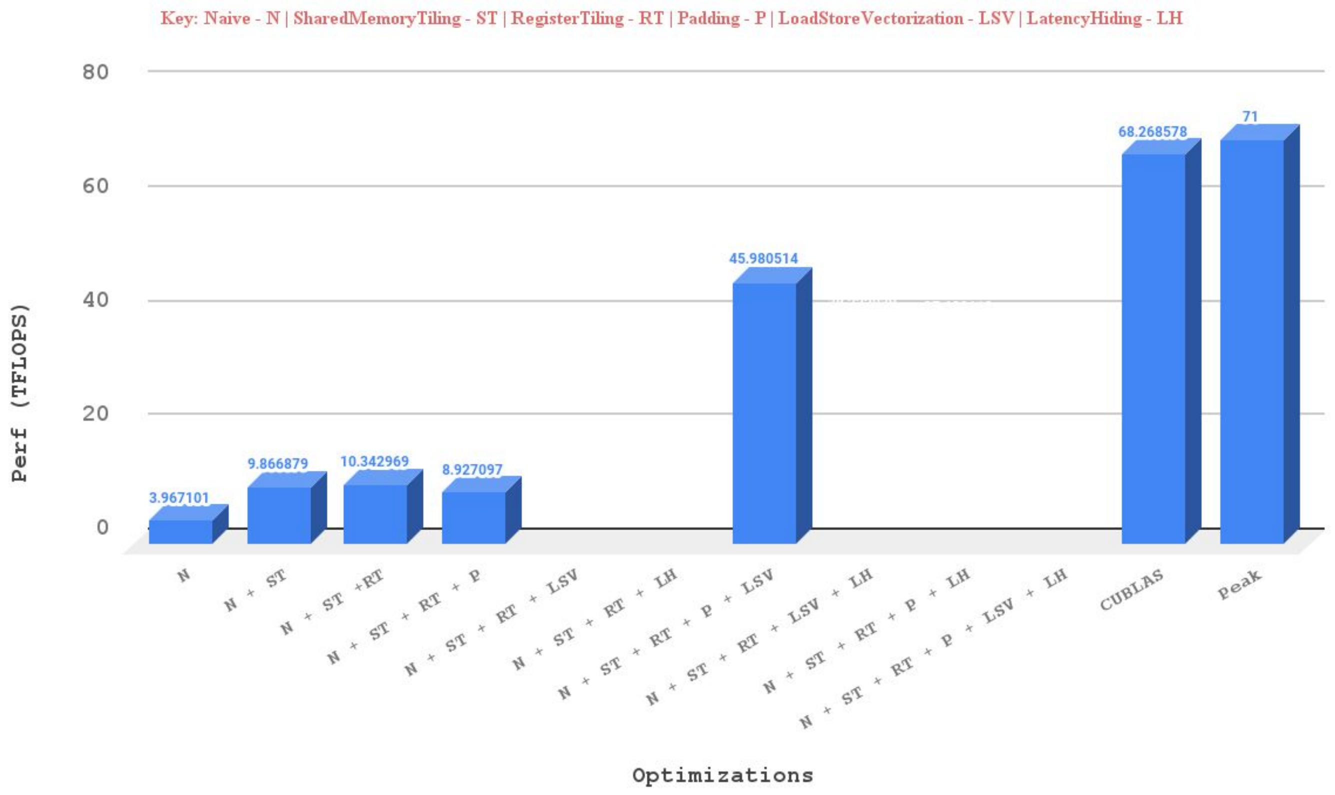
```
affine.for %arg21 = 0 to 32 {
     affine.for %arg22 = 0 to 256 {
        %138 = affine.load %B[%arg21, %arg22] : memref<8192x8192xf16>
        affine.store %138, %frag_B_padded[%arg21 - %arg4, %arg22 - %arg1] :
                                                    memref<32x264xf16, 3>
      }
}
```

Scalar copy loop

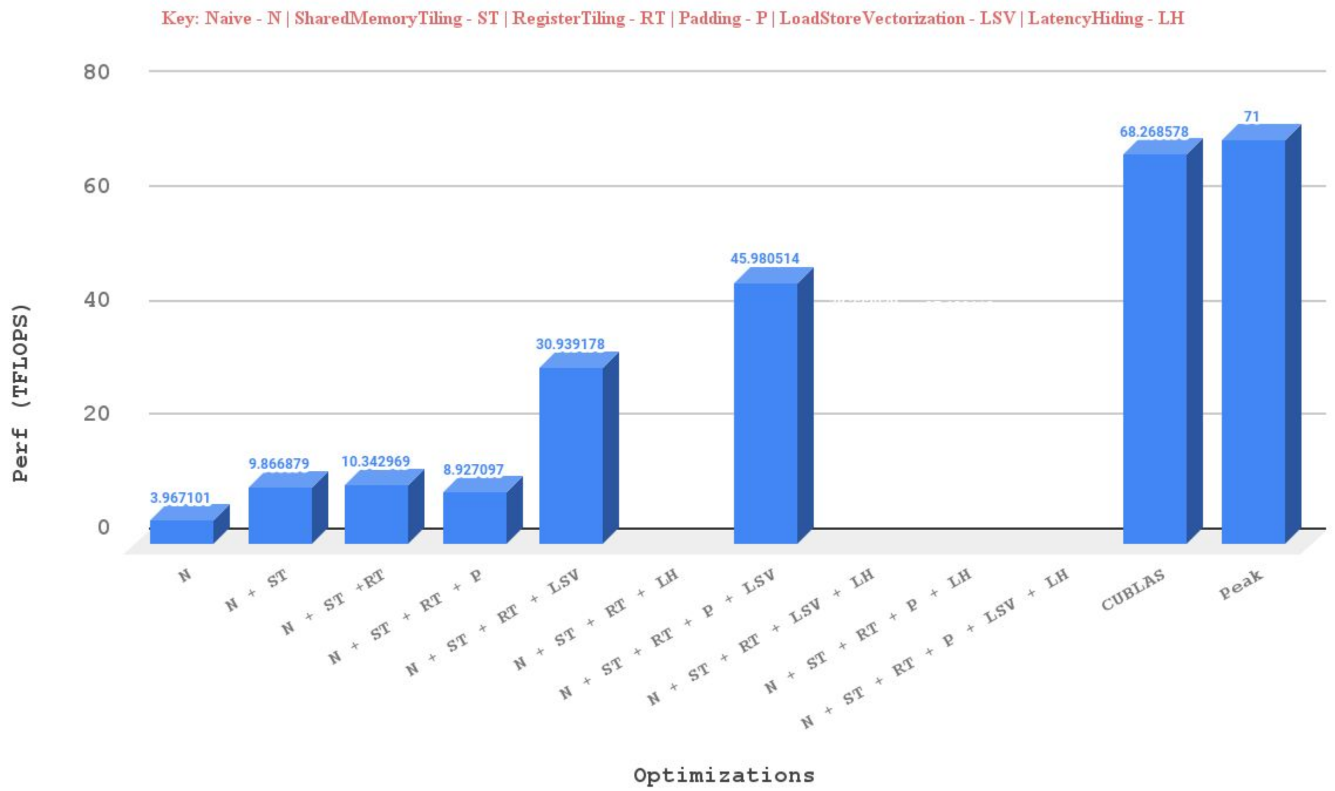
```
%7 = memref_vector_cast %memref_0 : memref<8192x8192xf16> to
                                            memref<8192x1024xvector<8xf16>>
%15 = memref_vector_cast %13 : memref<32x264xf16, 3> to memref<32x33xvector<8xf16>, 3>>
affine.for %arg21 = 0 to 32 {
     affine.for %arg22 = 0 to 256 step 8 {
          %147 = affine.load %7[%arg21, %arg22 floordiv 8] :
                                                memref<8192x1024xvector<8xf16>>
          affine.store %147, %15[%arg21 - %arg4, (%arg22 - %arg1) floordiv 8] :
                                            memref<32x33xvector<8xf16>, 3>>
      }
}
```

Vectorized copy loop

Code Generation for Tensor Core Matmul

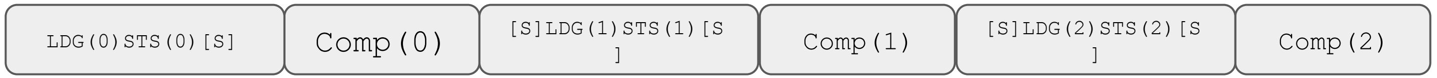


Code Generation for Tensor Core Matmul - Knobs

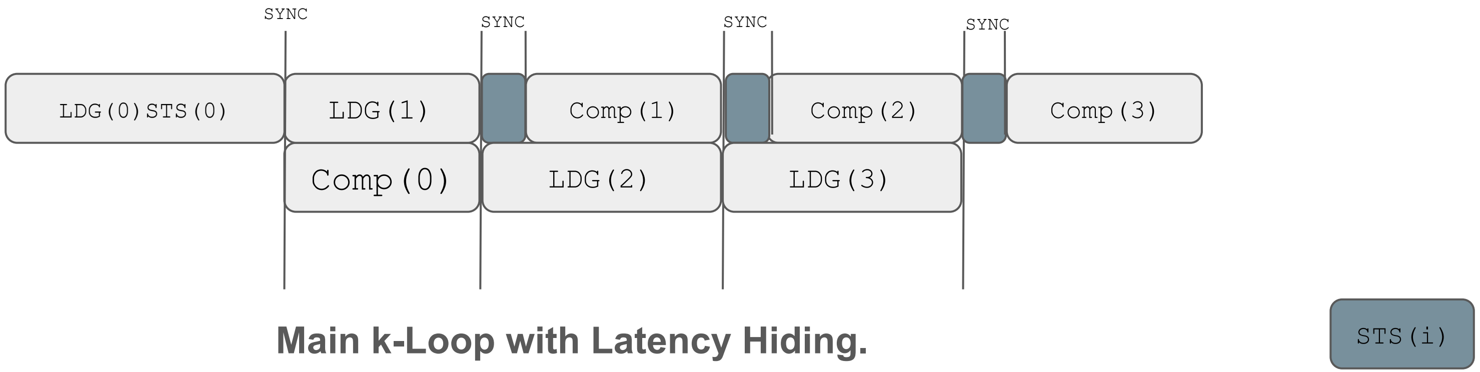


Code Generation for Tensor Core Matmul

Global Memory Load Latency Hiding:-



Main K-Loop of Matmul Kernel.

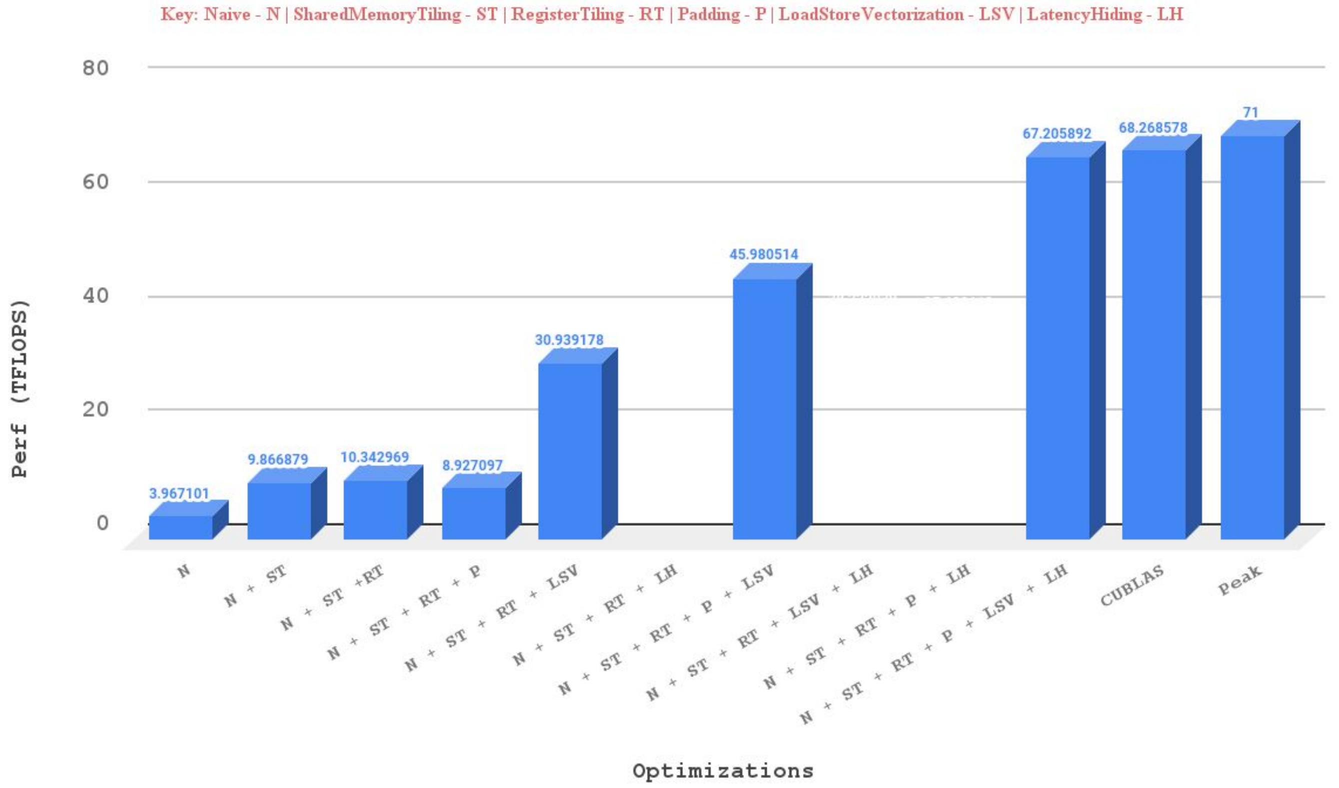


Main k-Loop with Latency Hiding.

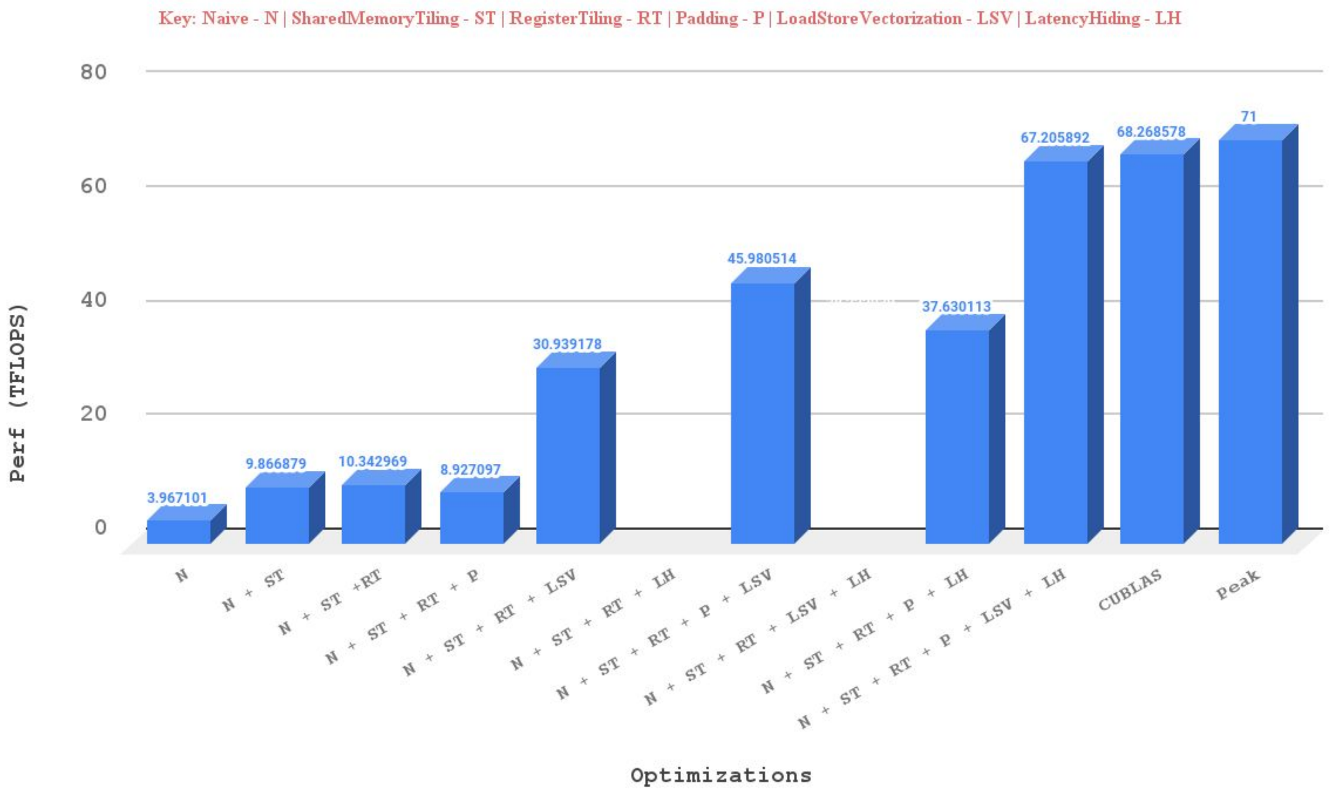
Code Generation for Tensor Core Matmul - Skeleton of Generated Kernel

```
gpu.launch blocks(...) {
       %28 = gpu.subgroup_mma_load_matrix  %C[%26, %27] {leadDimension = 8192 : index}
                                            : memref<8192x8192xf32> -> !gpu.mma_matrix<16x16xf32, "COp">
       %30 = gpu.subgroup_mma_load_matrix  %C[%29, %27] {leadDimension = 8192 : index} :
                                            memref<8192x8192xf32> -> !gpu.mma_matrix<16x16xf32, "COp">
       ...
       // Copy loops for iteration 0 of k-loop.
       scf.for %arg14 = %c0 to %c4 step %c1 {
              %62 = memref.load %B[%53, %61] : memref<8192x1024xvector<8xf16>>
              memref.store %62, %b_smem_cast[%53, %68] : memref<64x17xvector<8xf16>, 3>
       }
       ...
       gpu.barrier
       // Thread block k-loop.
       %49#2 = scf.for %arg14 = %c0 to %c8128 step %c64 iter_args(%arg14 = %28, %arg15 = %30)
                          -> (!gpu.mma_matrix<16x16xf32, "COp">, !gpu.mma_matrix<16x16xf32, "COp"> {
              gpu.barrier
              // Global memory loads for iteration i + 1 of k-loop.
              %82 = memref.load %A[%74, %81] : memref<8192x1024xvector<8xf16>>
              %102 = memref.load %B[%94, %101] : memref<8192x1024xvector<8xf16>>
              …
              // Compute for iteration i of k-loop.
              scf.for %arg31 = %c0 to %c64 step %c32 iter_args(%arg16 = %arg14, %arg17 = %arg15)
                          -> (!gpu.mma_matrix<16x16xf32, "COp">, !gpu.mma_matrix<16x16xf32, "COp"> {...}
              gpu.barrier
              // Shared memory stores for iteration i + 1 of k-loop.
              memref.store %102, %b_smem_cast[%51, %68] : memref<64x17xvector<8xf16>, 3>
              memref.store %82, %a_smem_cast[%150, %167] : memref<128x9xvector<8xf16>, 3>
              ...
       }
       // Compute loop for iteration n-1 of k-loop.
       scf.for %arg14 = %c0 to %c64 step %c32 {
              ...
       }
       gpu.subgroup_mma_store_matrix  %49#0, %C[%26, %27] {leadDimension = 8192 : index} :
                                            !gpu.mma_matrix<16x16xf32, "COp">, memref<8192x8192xf32>
       gpu.subgroup_mma_store_matrix  %49#1, %C[%29, %27] {leadDimension = 8192 : index} :
                                            !gpu.mma_matrix<16x16xf32, "COp">, memref<8192x8192xf32>
}
```

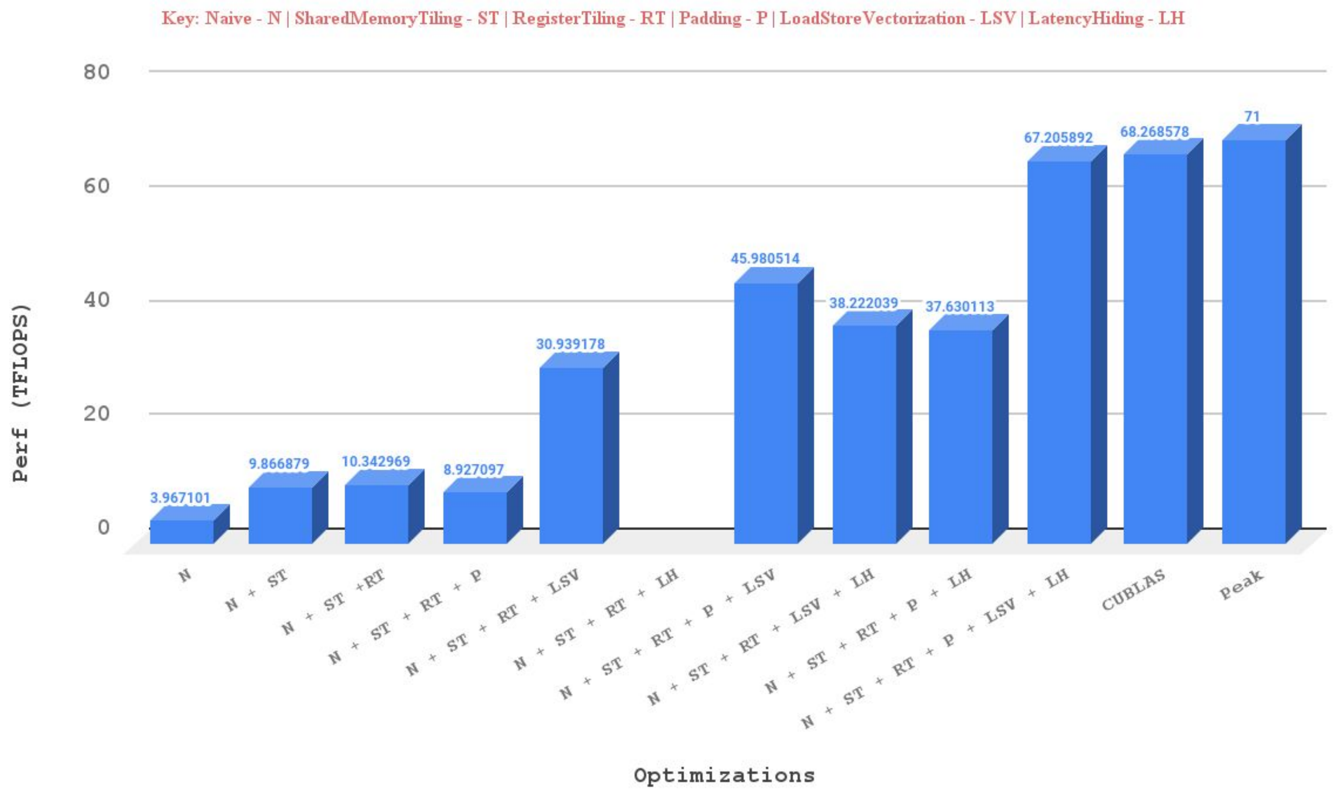
Code Generation for Tensor Core Matmul



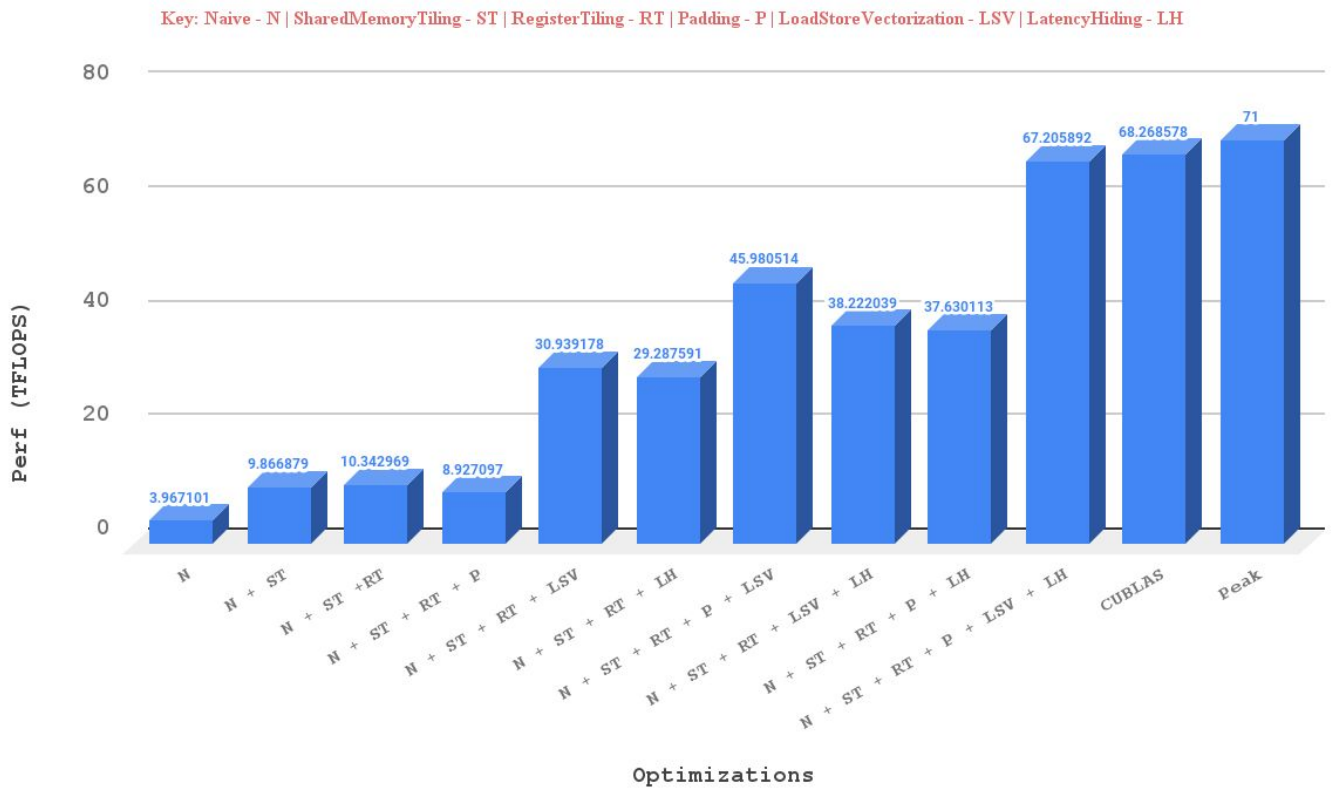
Code Generation for Tensor Core Matmul - Knobs



Code Generation for Tensor Core Matmul - Knobs

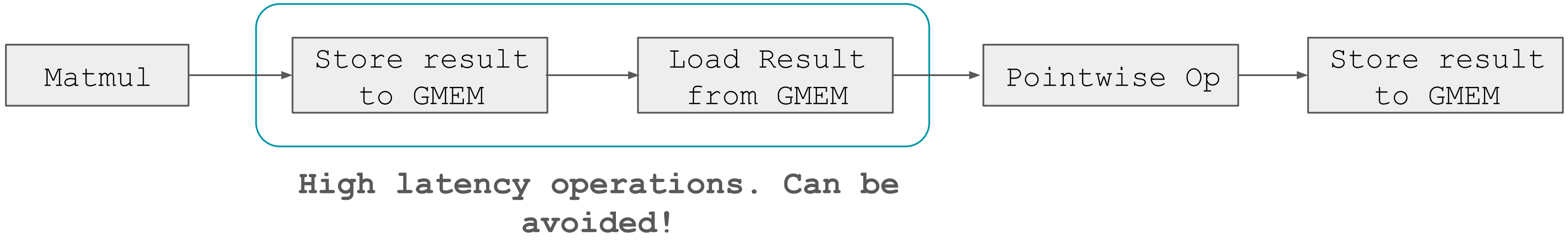


Code Generation for Tensor Core Matmul - Knobs



Code Generation for Tensor Core Matmul - What else?

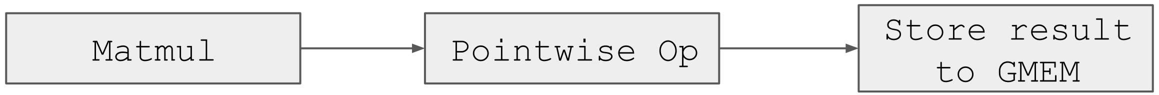
- In deep learning models, a commonly occurring patterns is matmul followed by some pointwise operation such as ReLU.
- Usually the operations that follow are O(n^2) and have RAW dependency on the result of matmul.
- When using frameworks such as PyTorch, the following thing happens:-



- By avoiding this, we can get real benefits on smaller kernels.

Code Generation for Tensor Core Matmul - What else?

- Because we are automatically generating our kernels we can take the opportunity to do Operator Fusion.
- **Idea** - Do not store the intermediate result of matmul to GMEM, instead keep them in the registers and apply the pointwise operation. Redundant loads-stores avoided.



Pointwise operation fused with matmul in registers.

Outline

- Motivation
- GPUs
  - Hardware
  - Programming Model
  - Compilation Path
- Matrix-Matrix Multiplication on GPUs
- Code Generation for Tensor Core Matmul
  - Performance Guidelines
  - Pipeline for Efficient Codegen
- Evaluation
  - Matmul
  - Fusion
- Conclusion, Collaborations and Future Directions

Evaluation

- Evaluation on two devices:-
  - Ampere based RTX 3090.
  - Turing based RTX 2080 Ti.
- The SM clocks were set to the boost frequency mentioned in the whitepaper for all the experiments.
- Only statically allocated shared memory used.
- The maximum number of registers per thread is set to 255.
- We consider matmul of the form C = AB + C.
- All three matrices are stored in a row-major layout.
- Square problem sizes ranging from 1024 to 16384 with step of 256.
- Raw kernel execution times used for calculating TFLOPS.

Evaluation

- In general, Results are on par with CuBLAS.
- Comparison against CuBLASLt for fusion with ReLU op.
- Significantly better in FP16 mode than cuBLAS on large sizes on Ampere, we attribute this to:-
  - cuBLAS kernels were using slightly smaller tile sizes in shared memory and might not be well tuned.
  - Had more stalls on global memory loads.
- Better perf in FP32 mode on Ampere than Turing when compared with cuBLAS:-
  - cuBLAS is better tuned for Turing than Ampere. Almost hits the Peak.
  - Optimal tile sizes found had bank conflicts on Turing.
  - Comparing with absolute device peak we hit 90% on Turing and 94% on Ampere.
- Better perf in FP32 mode than FP16 version on Turing, we attribute this to:-
  - FP16 Version - Large warp-tiles for better reuse in the register lead to register spills, and degraded performance. Same warp-tiles worked on Ampere without register spills.

Evaluation

	Mixed Precision		Half Precision	
	MIN	MAX	MIN	MAX
Ampere	95.34%	119.15%	80.21%	160.24%
Turing	86.78%	111.24%	72.09%	89.06%

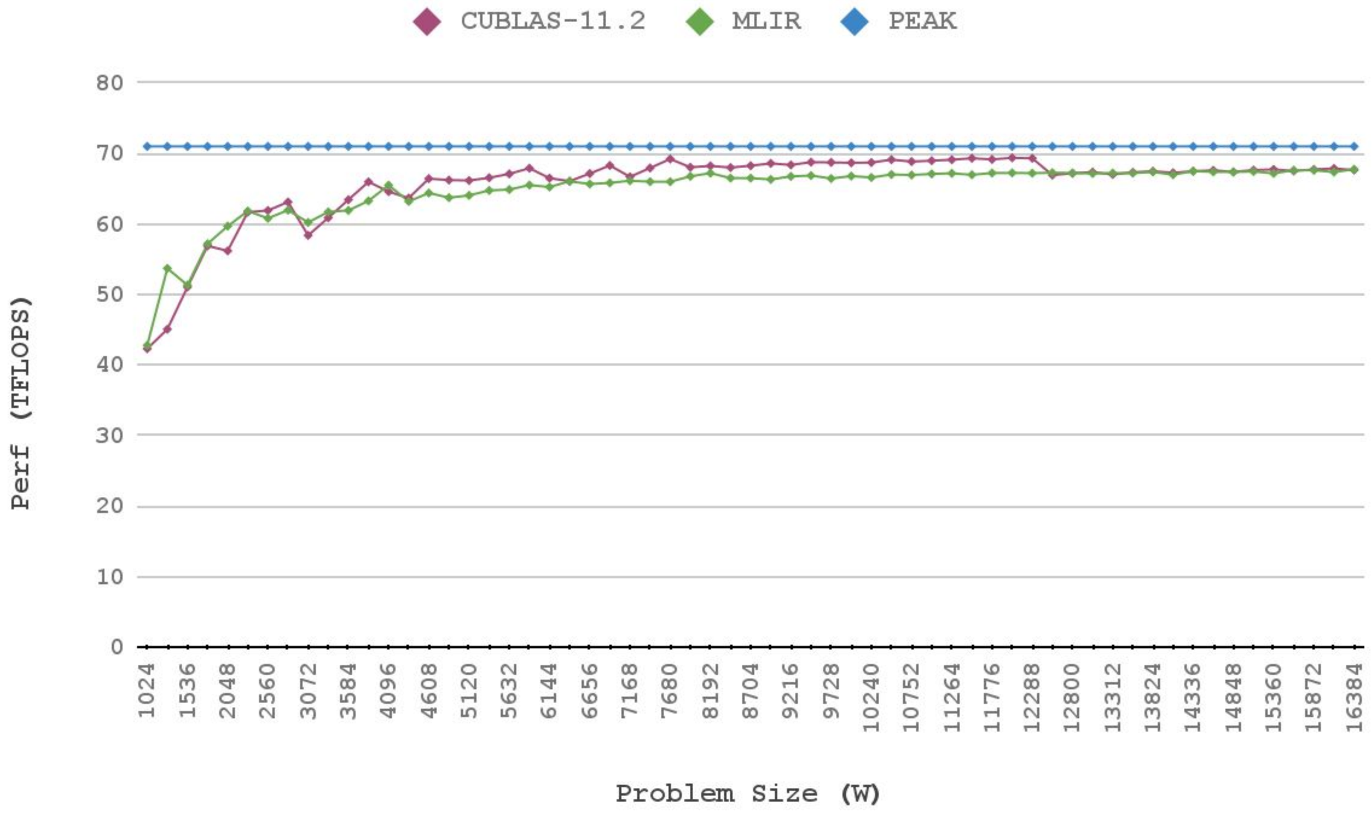
Figure 11. Summary of Matmul Perf Against CuBLAS v11.2

	Fused ReLU Vs. CuBLASLt		Fused Constant Addition Vs. CUBLAS + CUDA		Fused Matrix Addition Vs. CUBLAS + CUDA		Fused matrix addition + ReLU Vs. CUBLAS + CUDA	
	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
Ampere	95.32%	118.77%	99.5%	140.45%	100.97%	145.12%	103.63%	167.19%

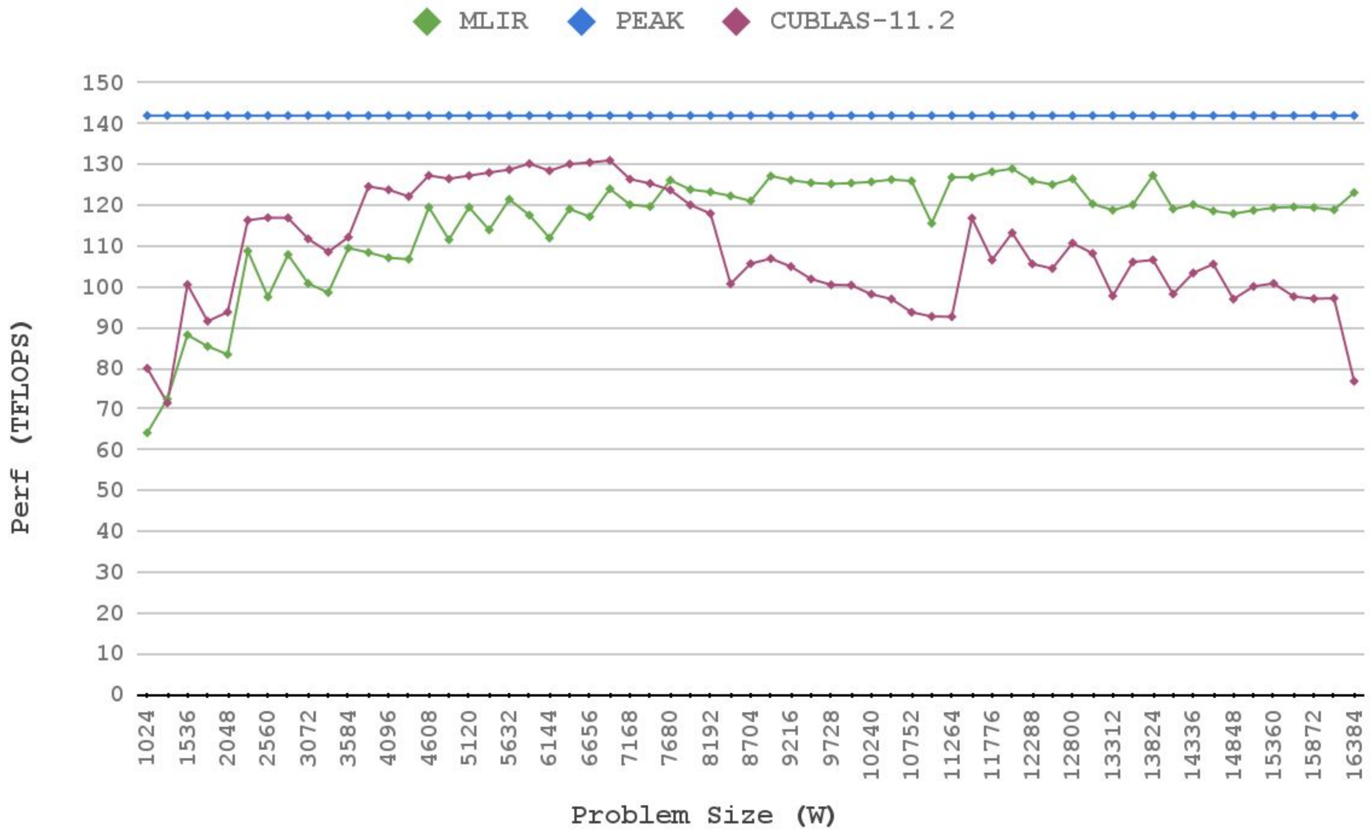
Figure 12. Summary of Ops Fused with Matmul in Mixed Precision

Outline

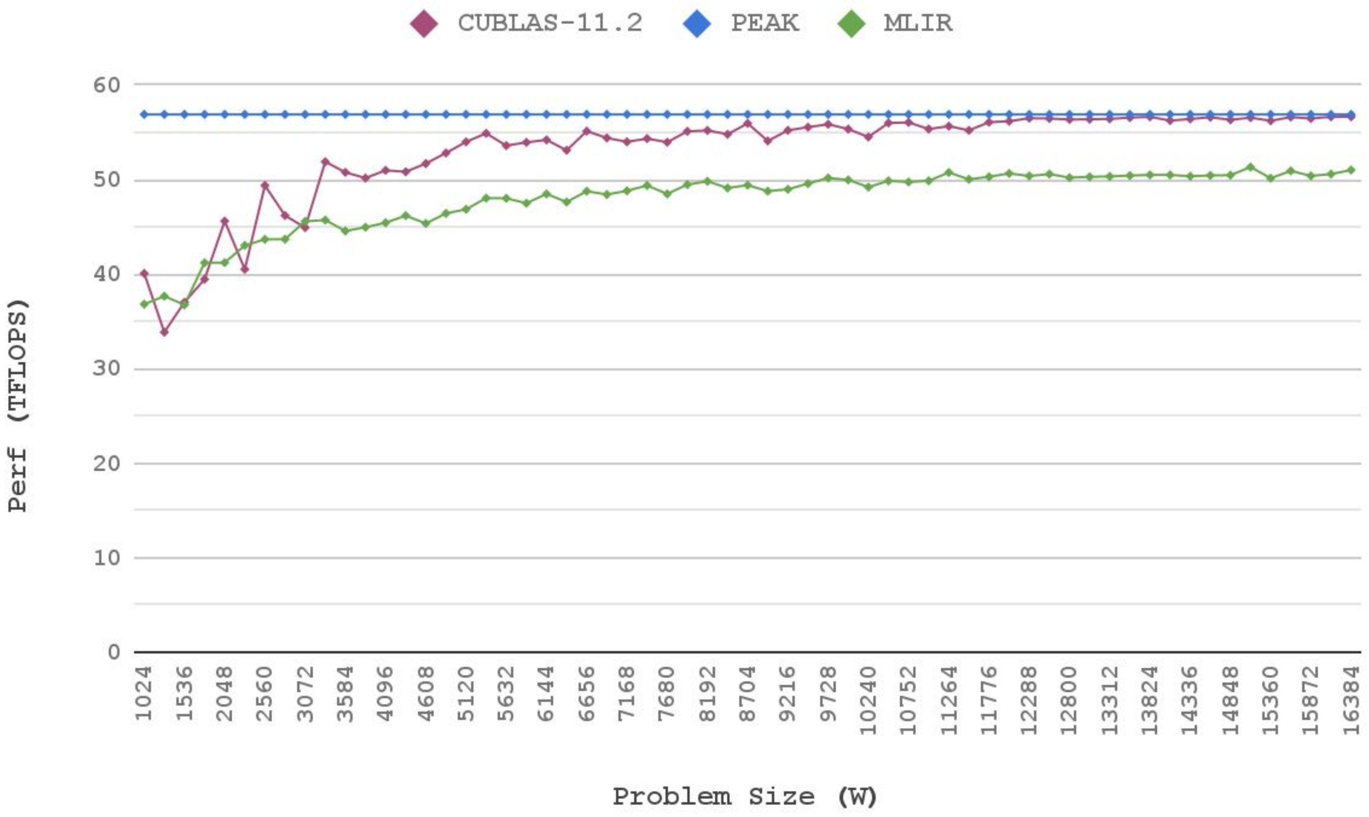
Mixed-Precision Performance on Ampere



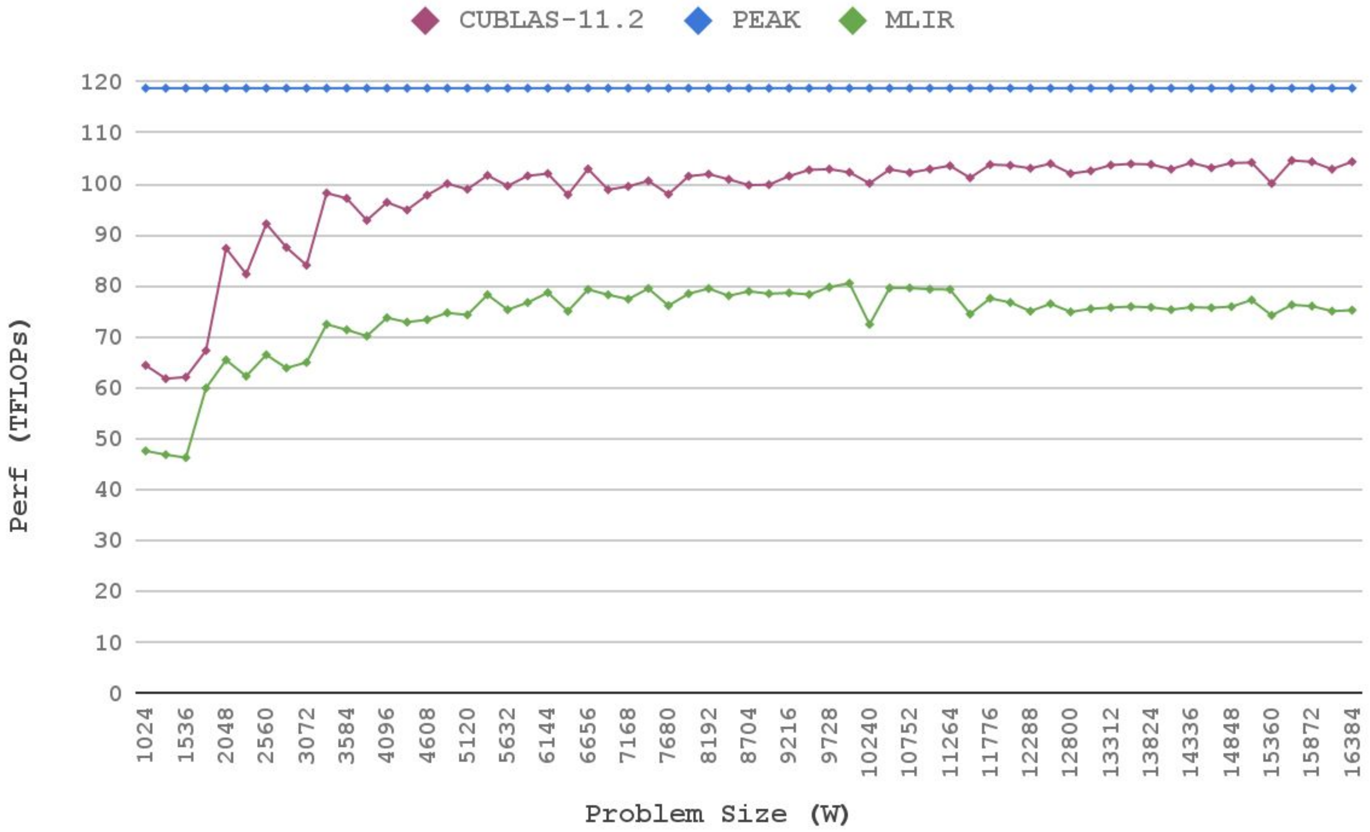
Half-Precision Performance on Ampere



Mixed-Precision Performance on Turing

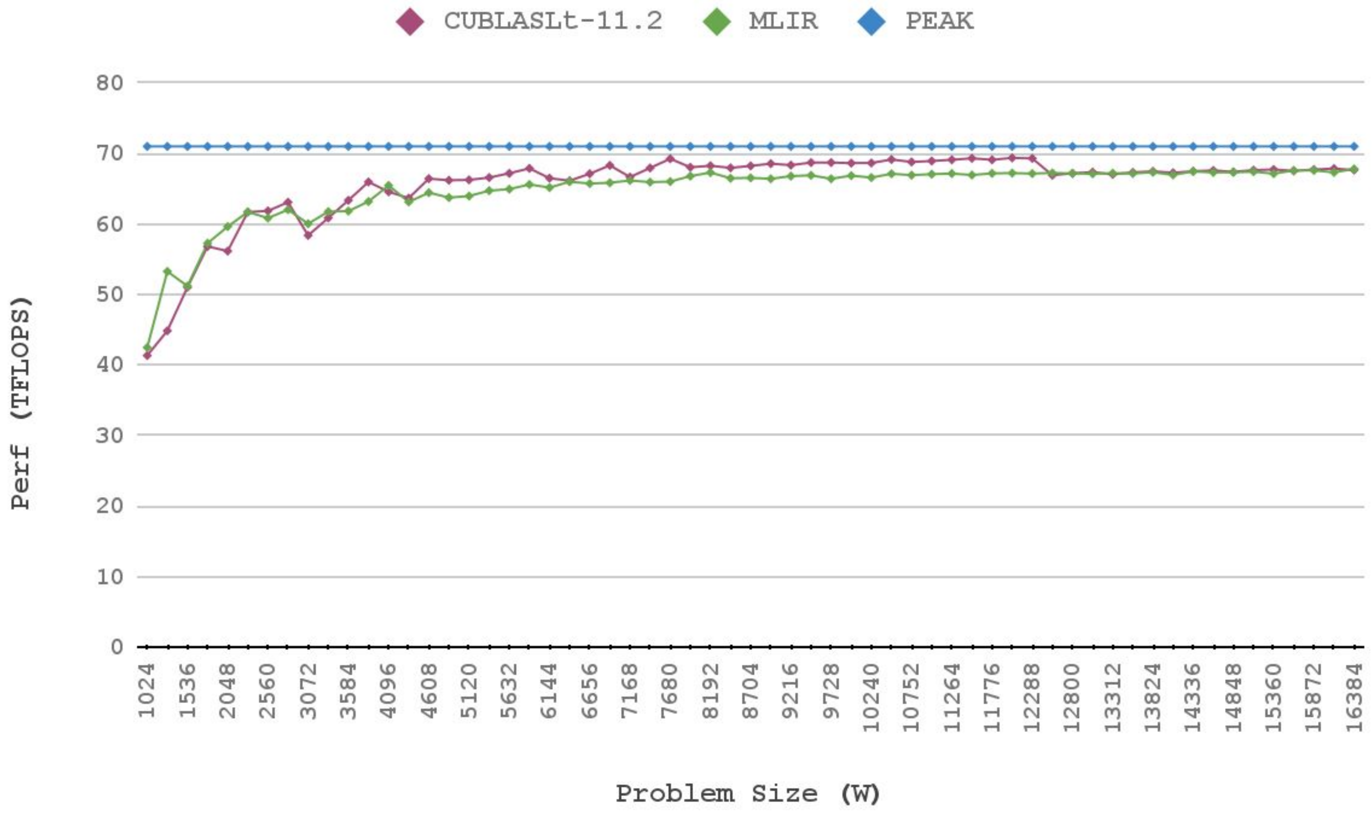


Half-Precision Performance on Turing

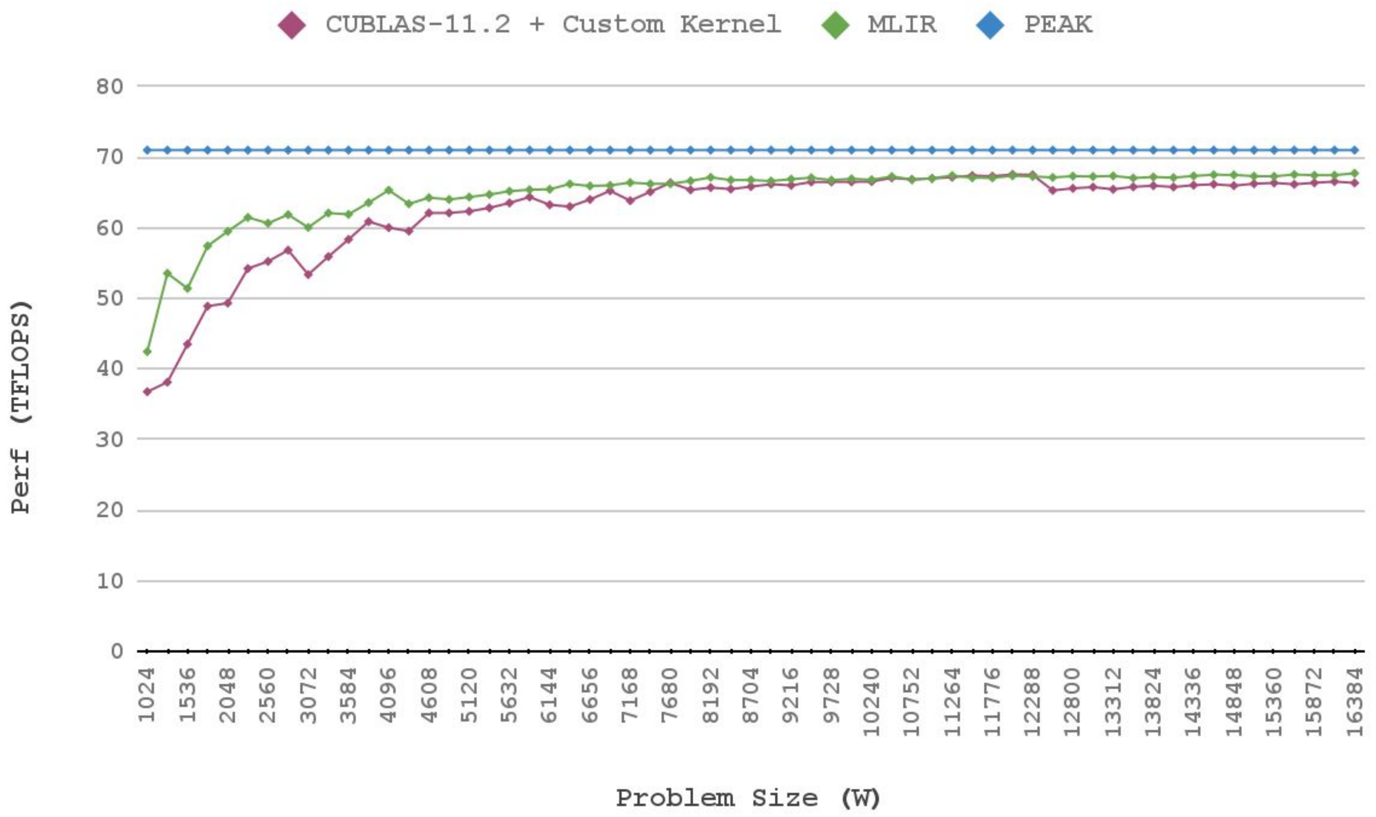


Outline

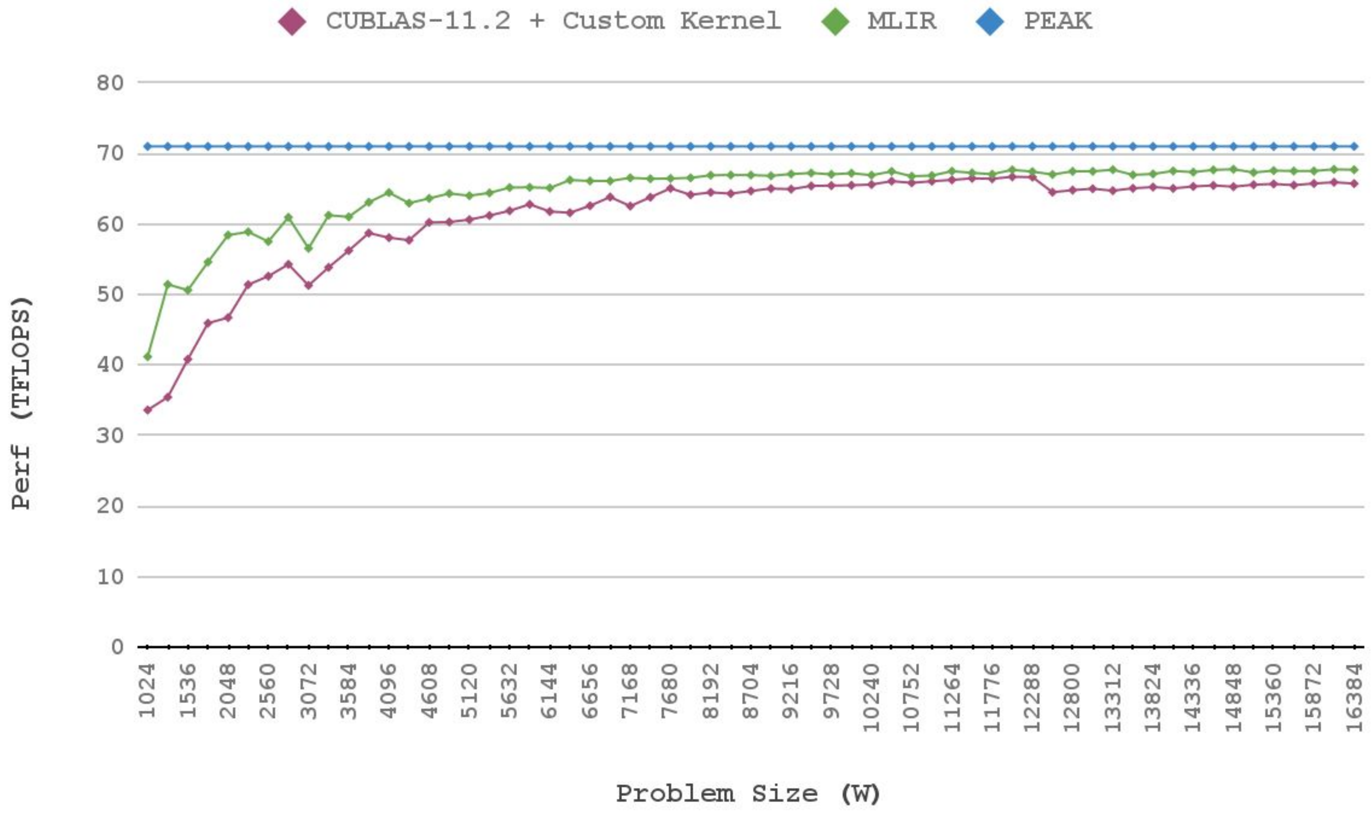
Mixed-Precision Fusion with ReLU on Ampere



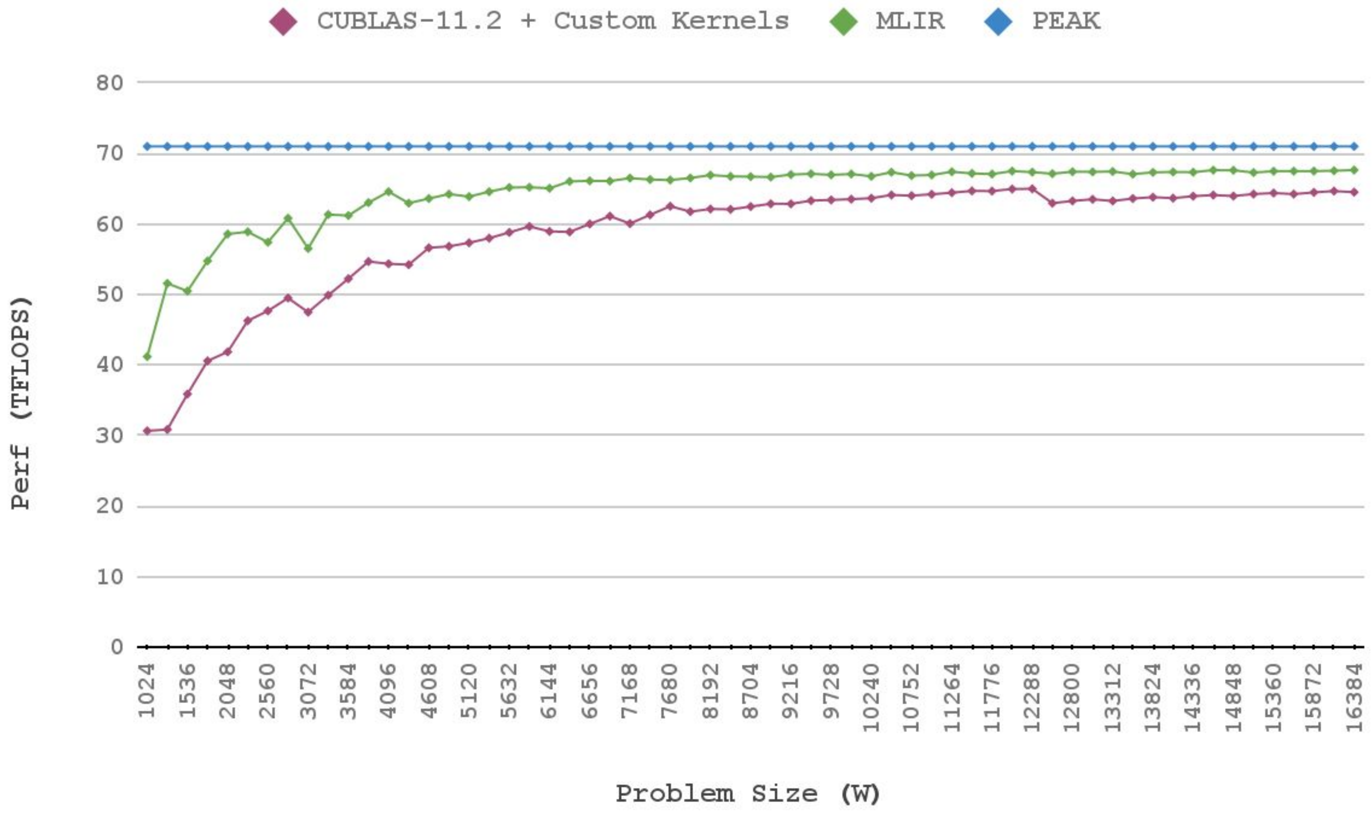
Mixed-Precision Fusion with Constant Add on Ampere



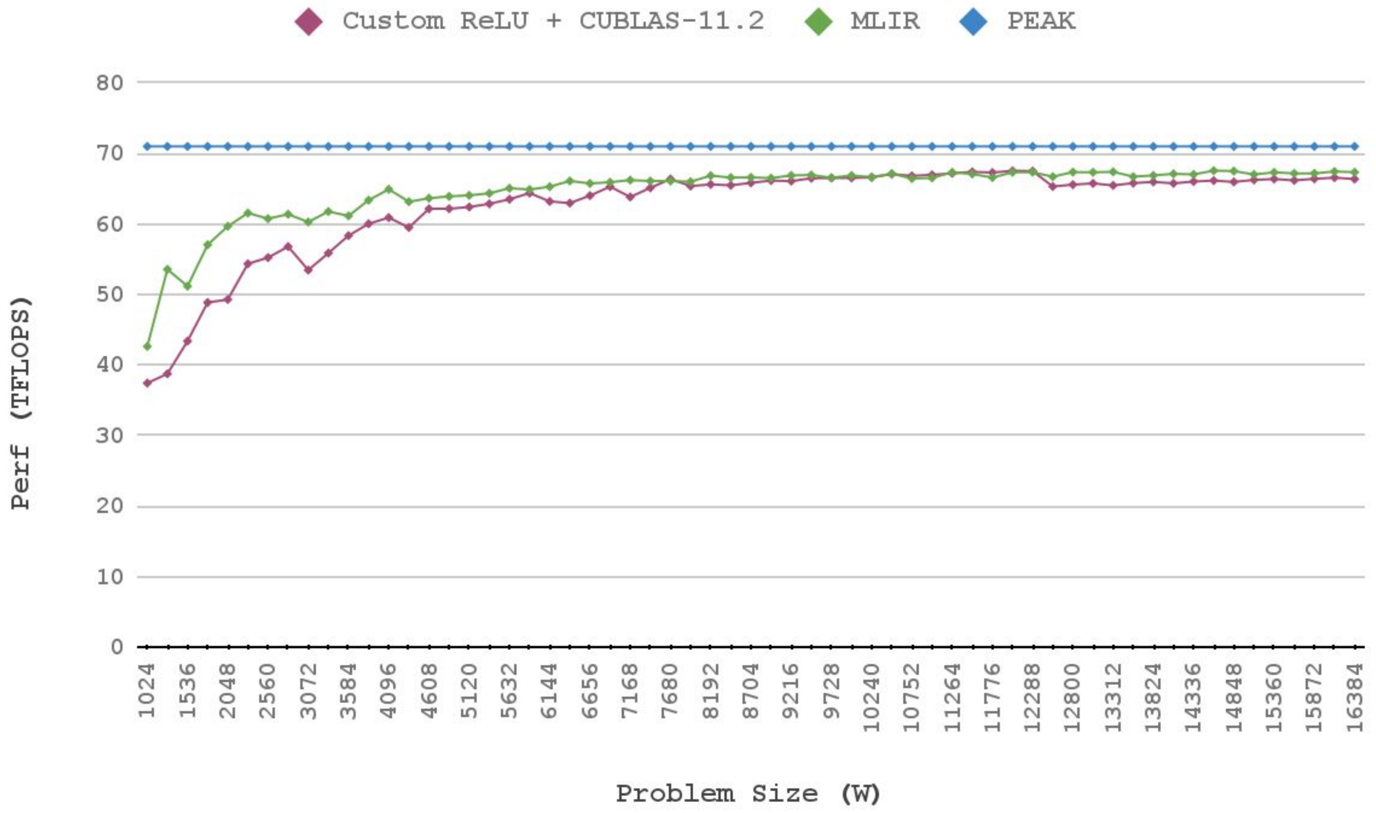
Mixed-Precision Fusion with matrix Add on Ampere



Mixed-Precision Fusion with matrix Add + ReLU on Ampere



Mixed precision Fusion with input from ReLU on Ampere



Outline

- Motivation
- GPUs
    - Hardware
    - Programming Model
    - Compilation Path
- Matrix-Matrix Multiplication on GPUs
- Code Generation for Tensor Core Matmul
    - Performance Guidelines
    - Pipeline for Efficient Codegen
- Evaluation
    - Matmul
    - Fusion
- Conclusion, Collaborations and Future Directions

Conclusion, Collaborations and Future Directions

- Automatic code generation for GPU tensor cores using MLIR appears promising
- Basic operations required to program tensor cores were contributed upstream.
- Plan to work and contribute other layers starting from the lower ones.
- Discuss to add a general scf to gpu mapping with warp level support.
- Mapping Affine operations to GPU WMMA operations.
- Plan to generalize operator fusion to handle a variety of fusion scenarios.
- **Ultimately, modular, and reusable code generation building blocks (beyond just GEMM).**

Thanks!