Project Report

Names: Yuan Zhang, Jingde Chen, Yu Wang
Netids: yuanz4, jingdec2, yuw14
Team Name: x_correlation
School Affiliation: on campus students

Final Optimizations

1. Introduction

The optimizations we implemented for the final submission are:

- Exploiting parallelism in input images, input channels, and output channels (UNROLL_EXPLICIT)
- Kernel fusion for unrolling and matrix-multiplication (UNROLL_IMPLICIT)
- Tuning with restrict and loop unrolling (SHARED)
- Sweeping various parameters to find best values (UNROLL_IMPLICIT)
- Multiple kernels for different layer sizes (UNROLL_IMPLICIT)

We also tried:

- TensorCores or FP16 implementation (WMMA)

All of the optimizations except WMMA achieves the correct accuracy: 0.7653 for 10000 images, 0.767 for 1000 images, 0.76 for 100 images. And we will discuss their performance, explain the details of approaches and results in the following sections.

We implemented all optimizations in *new-forward.cuh*. You can choose to run with each by defining different macros at the beginning of the file with "#define". Apart from those implemented in Milestone 4, there are 5 additional options:

- UNROLL_EXPLICIT: an improved version of UNROLL, more parallelism on input images and load division
- UNROLL_IMPLICIT: kernel fusion of unroll step and matrix multiplication step, combined with multiple kernels for different layer sizes, also used for parameters sweeping
- SHARED: add restrict and loop unrolling pragma, based on shared memory convolution, so combined with SHARED optimization in Milestone 4
- UNROLL_IMPLICIT_CONSTANT: based on UNROLL_IMPLICIT, store filter in constant memory
- WMMA: use FP16 to compute product of two elements

2. Op Time

Each option contains the op time of two different layers:

	10000 images	1000 images	100 images
UNROLL_EXPLICIT	0.027682, 0.076220	0.003689, 0.010597	0.001332, 0.003672
UNROLL_IMPLICIT	0.021536, 0.071667	0.002176, 0.007195	0.000239, 0.000743
SHARED	0.028862, 0.072565	0.002877, 0.007612	0.000313, 0.000798
UNROLL_IMPLICIT (Multiple kernel)	0.021692, 0.045428	0.001965, 0.004571	0.000237, 0.000534

Analysis: Exploiting more parallelism in the original unroll_explicit algorithm does improve the performance (to 104ms). But it still runs slower than doing the unroll process implicitly as in unroll_implicit due to excessive loops. Applying __restrict__ and unroll pragmas reduces the runtime of shared-memory-convolution significantly (from 120ms to 100ms) while it has almost no impact on unroll_implicit. Further tuning of tile sizes and implementation of different kernels for different layers boost our performance again. For our fastest implementation unroll_implicit the runtime drops from an average of 90ms to 70ms.

3. Implementation Details

Optimization 4: Exploiting parallelism in input images, input channels, and output channels

In the original unroll method, we cannot load the entire unrolled matrix into memory, so we have to divide them into B iterations, where each iteration is correspondent to an image. But it is very inefficient, so we use the y-dimension of grid to compute BATCH_SIZE images in parallel. Apart from that, in the original unroll, we use each thread to load one elements. But in the explicit unroll we make every thread to load K*K elements.

When BATCH_SIZE is 1000, meaning total 10 iterations, the time decreases to about 105ms, which is fruitful.

Optimization 5: Kernel fusion for unrolling and matrix-multiplication

Instead of using a kernel to convert input image to unroll matrix, we can directly calculate index of specific input and load them when doing matrix multiplication. The advantage is that we don't need additional memory to store the unrolled matrix, so now it can be done for all images in one iteration. And now we also don't need additional kernel launch. For global memory access, the fusion kernel can save one global read and one global write for each of H_out*W_out*C*K*K threads, which is a big decrease.

With this optimization, the time is much shorter, reaches about 85ms.

Optimization 6: Tuning with restrict and loop unrolling

We also tried tuning the kernel with restrict and loop unrolling to further improve the kernel performance. Basically, we used __restrict__ decorations to inform nvcc that the three pointers (*x as the input features maps, *y as the output feature maps, *k as the filters) do not alias one another. Since they do not overlap, the compiler is free to optimize the code to use the parallel instructions.

In our code, we add const and __restrict__ before the pointers which point to the input feature maps and the filters matrices, and __restrict__ before pointers which point to the output feature maps. Thus, the kernel can take advantage of the read-only data cache on the GPUs. After that, in order to control the loop unrolling, we briefly add #pragma unroll before the for loops.

We observed that this optimization has the most significant effect on the shared memory convolution since there are several for loops within that kernel, and we could fully utilize the advantage of loop unrolling. Before we use this strategy, the runtime of two layers on dataset 10000 is 33.110ms, 87.624ms respectively, 120.734ms in

total. After we implement this optimization, the runtime of 2 layers is reduced to 28.862ms .72.565ms respectively, 101.427ms in total. Thus, tuning with restrict and loop unrolling has greatly improved the shared memory convolution kernel execution speed.

Optimization 7: Sweeping various parameters to find best values

After we determined that our implicit unroll algorithm has the best performance, we moved on to tune the hyperparameters of the algorithm. We focused on tuning the TILE_WIDTH used in unrolled matrix multiplication. We swept through width values allowed in a block from 4 to 32 with a step of 4 and compare the resulting operation times. The sweeping results showed that a TILE_WIDTH of 16 produces the best run time if we use the same tile_width for the two layers, with an average of 84ms.

	4	8	12	16	20	24	28	32
OpTime1	0.0785	0.0308	0.0212	0.0199	0.0332	0.0406	0.0343	0.0388
OpTime2	0.2981	0.0805	0.0655	0.0647	0.0854	0.0457	0.0582	0.0600
Total	0.376	0.1113	0.0867	0.0846	0.1186	0.0863	0.0925	0.0988

Optimization 8: Multiple kernel implementations for different layer sizes

From the table of the sweeping optimization above, we also observed that runtime dependency of the two layers on the tile width varies. Operation 1 performs best with a tile width of 16 finishing in around 20ms, and operation 2 performs best with a tile width of 24 finishing in around 45 ms. We propose that using a combination of different tile widths (16 and 24) for the two layers could produce a better result of runtime around 20 + 45 = 65ms. We further implemented this optimization for our implicit unroll MM algorithm, using kernels with different TILE_WIDTH for different layer sizes. The analysis of this optimization proved our assumption to be correct. Using a tile width of 16 for the first layer with 12 output channels and a tile width of 24 for the second layer with 24 output channels results in a better total runtime of around 21+48 = 69ms.

Optimization 9: TensorCores or FP16 implementation

We try to use TensorCores to accelerate the speed of matrix multiplication, where Nvidia multiple two half precision matrix and add one float matrix. It is widely used in many deep learning framework, but not very suitable for our project. We want to integrate TensorCores with explicit unroll when doing matrix multiplication, but we encounter several problems and decide not to use that.

The first problem is that TensorCores only support the multiplication of half type (16 bits), but the type in our project is float (32 bits). I convert float to half to do multiplication and convert back to float when storing to output. But I notice that it sacrifices some accuracy, so the correctness for 10000 images is 0.7655 (different from expected 0.7653).

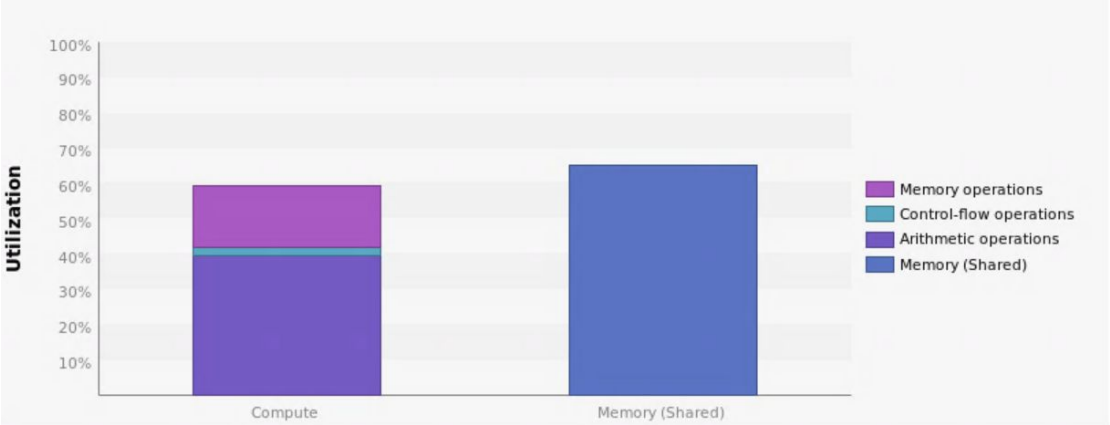
The second problem is that the dimension of matrix M and matrix N must be a multiple of 16, which can't be satisfied in this project because dimension is defined by the layer configuration.

4. NVVP Analysis

We will analyze the performance of the first 3 optimizations by looking at NVVP statistics of the first forward layer of each implementation when image size is 100.

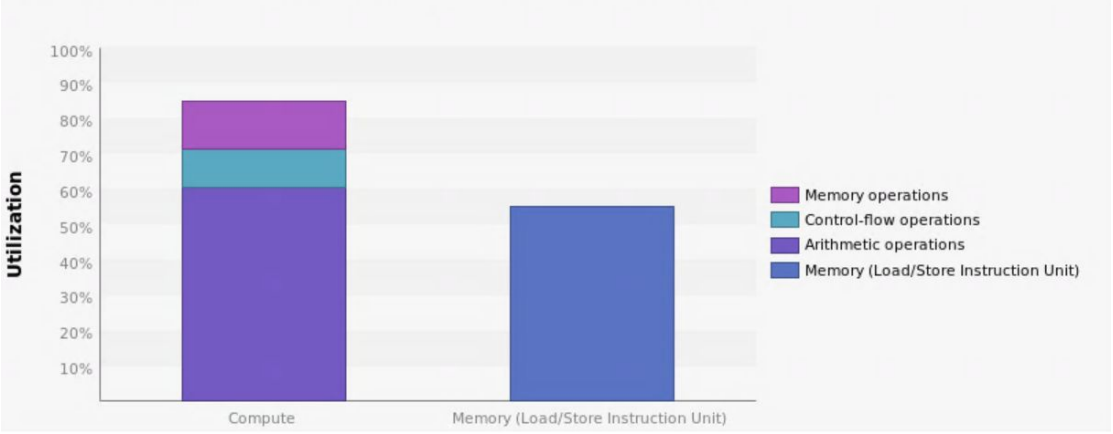
4.1 Kernel performance limiter:

UNROLL_EXPLICIT:



UNROLL_IMPLICIT:

SHARED:



Analysis:

- In the explicit optimization, the performance of kernel is most likely limited by the memory system, and the biggest limiting factor is shared memory

bandwidth.

- In the implicit optimization, the performance is most limited by computation on the SMs.
- In the shared optimization, the arithmetic computation on the SMs also is the biggest limiting factor.

4.2 Divergent branches:

UNROLL_EXPLICIT: No

UNROLL_IMPLICIT: 6.5% (whether loading filter)

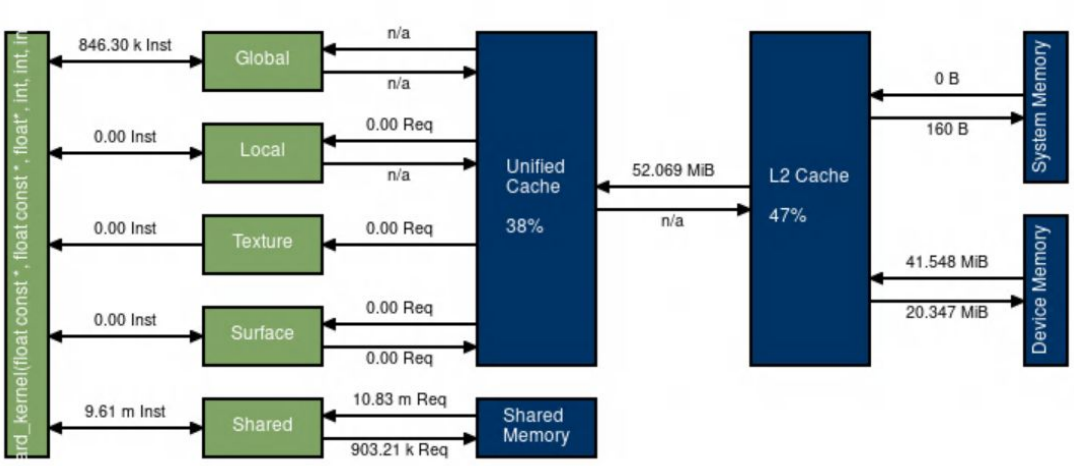
SHARED: 18.5% (whether loading input image), 64.3% (whether do computation)

Analysis:

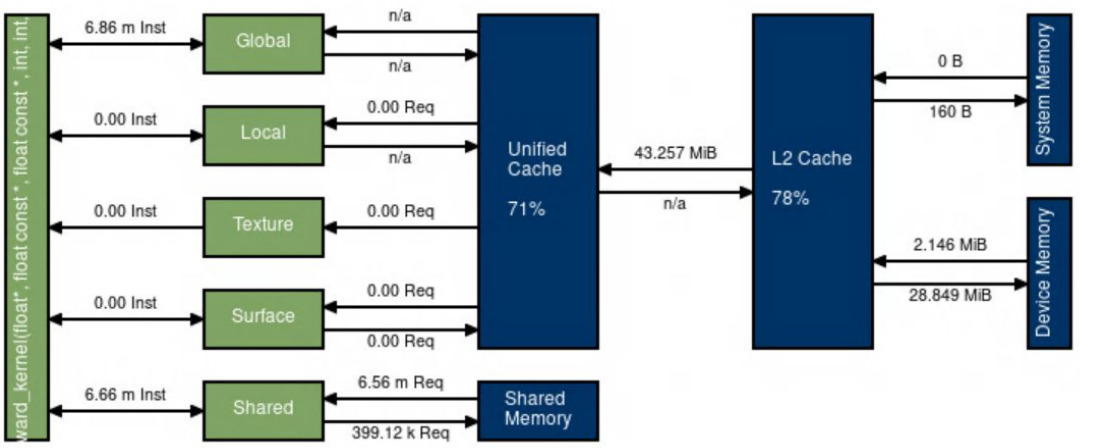
- In the explicit optimization, there is no control divergence. All threads will compute K*K elements.
- In the implicit optimization, during conceptual unrolling step, only a small part of threads will load the filter and all threads will load input image.
- In the shared optimization, some threads at boundary may not load input image, resulting in the first control divergence. When computing, the threads loading halo elements are inactive.

4.3 Memory statistics:

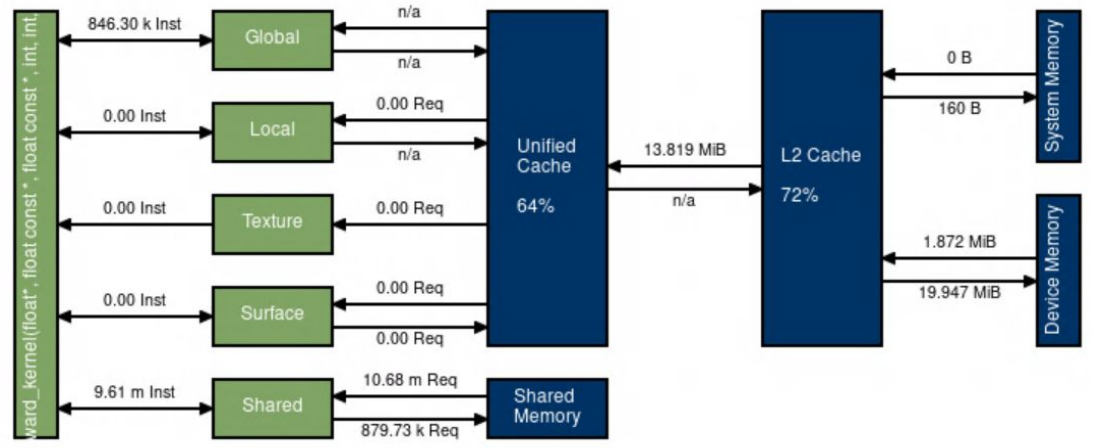
UNROLL_EXPLICIT:



UNROLL_IMPLICIT:



SHARED:

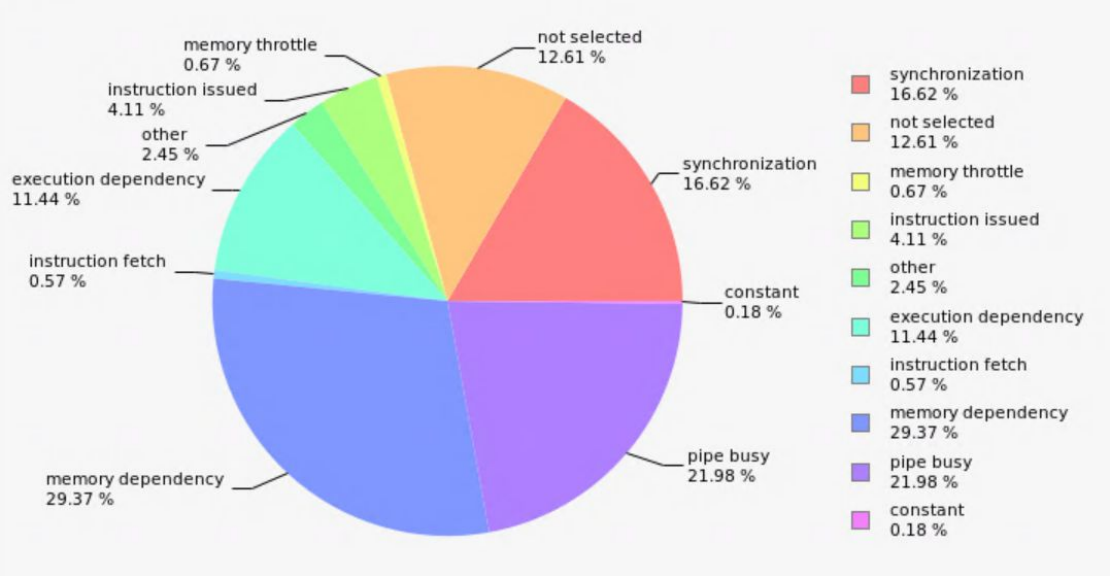


Analysis:

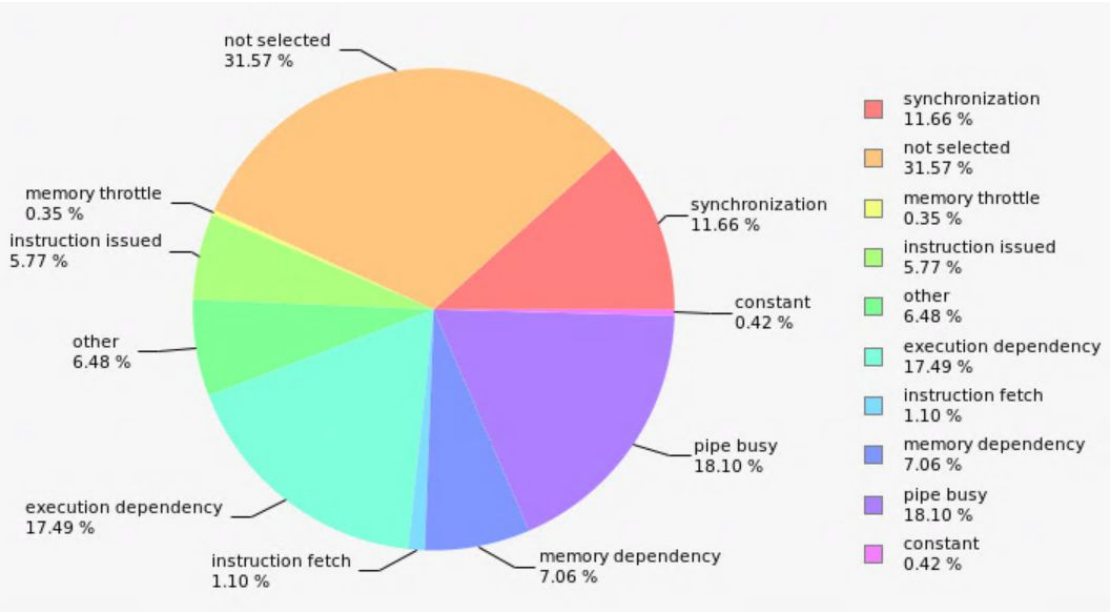
- On memory usages, shared-memory-convolution uses the most significant amount of caches among the algorithms we implemented. This explains its relatively good performance in spite of the loops it use.
- Implicit unroll MM algorithm uses more unified and L2 cache than explicit unroll MM algorithm because it performs the unroll process inside the device kernel, trading device memory for smaller overhead in unroll process.

4.4 Kernel profile - PC sampling:

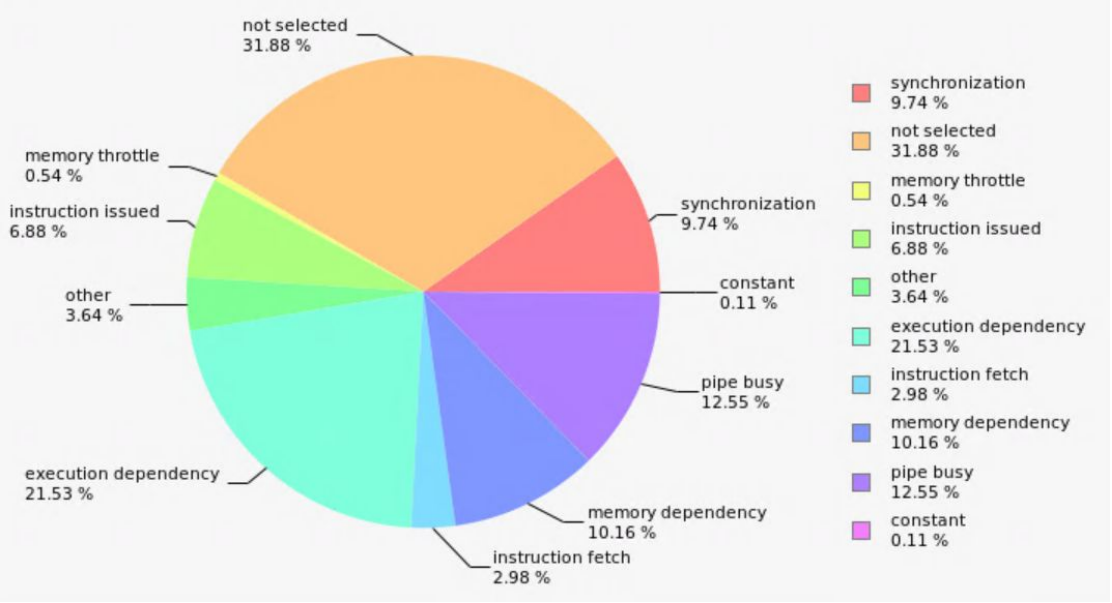
UNROLL_EXPLICIT:



UNROLL_IMPLICIT:



SHARED:



Analysis:

- In the explicit optimization, there is a big part of memory dependency, meaning the memory bandwidth limits the performance.
- In the implicit optimization, the main limitation comes from not selected, where warp is ready to issue, but some other warp issued instead.
- In the shared optimization, very similar to that of UNROLL_IMPLICIT, but a slightly higher execution dependency.

5. How our team organized and divided up this work

Each of us took in charge of several optimizations, including coding and analysis.

1. Unroll + shared-memory Matrix multiply : Yuan Zhang, Yu Wang, Jingde Chen
2&3. Shared-memory convolution & Weight matrix (kernel values) in constant memory : Jingde Chen
4. Exploiting parallelism in input images, input channels, and output channels: Yuan Zhang, Yu Wang, Jingde Chen
5. Kernel fusion for unrolling and matrix-multiplication: Yuan Zhang
6. Tuning with restrict and loop unrolling: Yu Wang
7. Sweeping various parameters to find best values: Jingde Chen, Yu Wang
8. Multiple kernel implementations for different layer sizes: Yu Wang
9. TensorCores or FP16 implementation: Yuan Zhang

6. References

- David H Bartley, Performance Tuning with “Restrict” Keyword, Systems and Applications R&D Center, Texas Instruments, 8 September 2010
- https://devtalk.nvidia.com/default/topic/1035783/does-the-use-of-16-bit-__restrict__-const-kernel-arguments-hurt-performance-/
- https://stackoverflow.com/questions/43235899/cuda-restrict-tag-usage
- http://www.orangeowlsolutions.com/archives/310
- https://devblogs.nvidia.com/programming-tensor-cores-cuda-9

Milestone 4

1. Introduction

The 3 optimizations we implement are:

- Unroll + shared-memory Matrix multiply (UNROLL)
- Shared Memory convolution (SHARED)
- Weight matrix (kernel values) in constant memory (CONSTANT)

We implemented all three optimizations in *new-forward.cuh*. You can choose to run with each by defining different macros at the beginning of the file with "#define". There are 5 options:

- ORIGINAL: the paralleled convolution without optimization
- UNROLL: input unrolled plus shared-memory matrix multiplication
- CONSTANT: weight matrix (kernel values) in constant memory
- SHARED: shared memory convolution
- CONSTANT + SHARED: hybrid of constant and shared memory

All of those optimizations achieves the correct accuracy: 0.7653 for 10000 images, 0.767 for 1000 images, 0.76 for 100 images. And we will discuss their performance in the following sections.

2. Op Time

Each option contains the op time of two different layers:

	10000 images	1000 images	100 images
ORIGINAL	0.030696, 0.095774	0.002909, 0.009462	0.000252, 0.000842
UNROLL	0.101361, 0.186982	0.017616, 0.020104	0.001849, 0.002538
CONSTANT	0.030431, 0.091505	0.002937, 0.009062	0.000256, 0.000767
SHARED	0.037284, 0.105367	0.003742, 0.011037	0.000411, 0.001148
CONSTANT + SHARED	0.033013, 0.094401	0.003340, 0.009504	0.000384, 0.001005

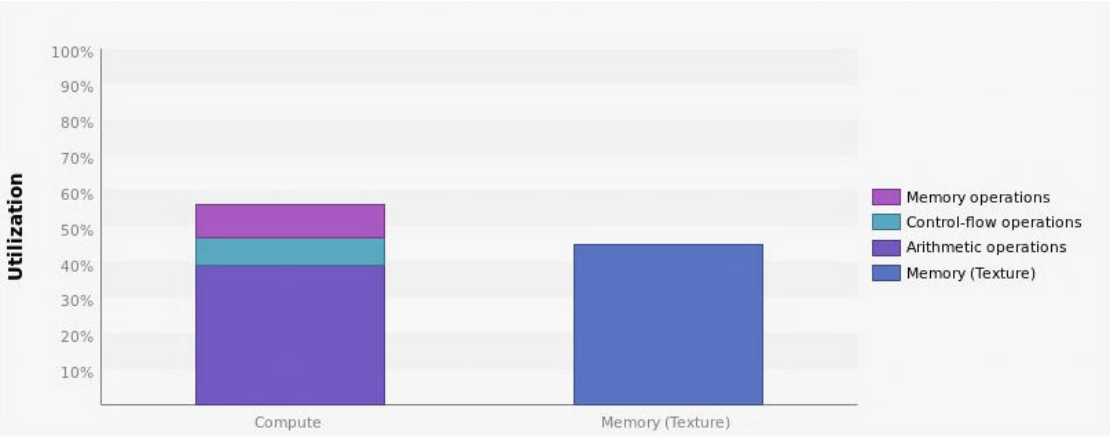
Analysis: Using constant memory does reduce our runtime slightly. The block structure of our shared-memory convolution can still be optimized for better parallelism. The performance of the unrolled matrix multiplication method does not meet our expectations. This is because due to the limited size of the grid, the operation for a batch cannot be paralleled and had to be done in a for loop, which is slow due to low level of parallelism.

3. NVVP Analysis

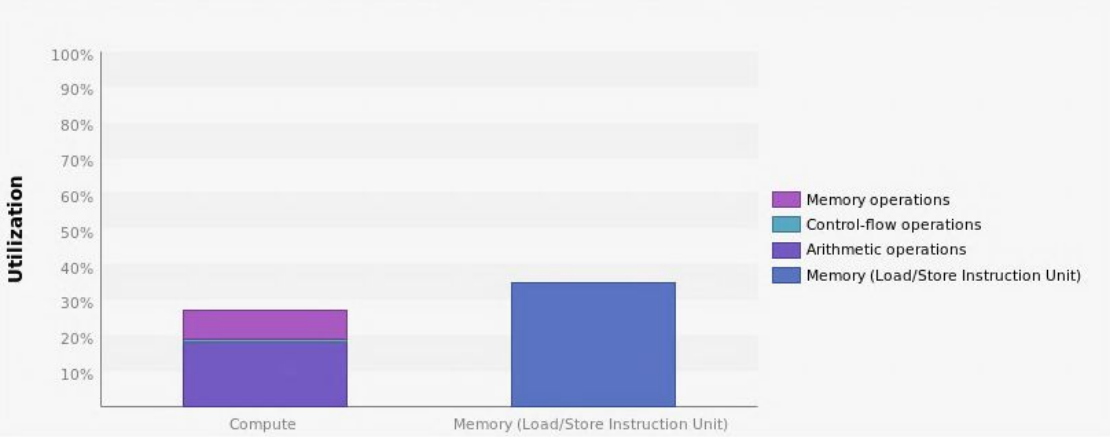
We will analyze the performance of the optimizations by looking at NVVP statistics of the first forward of each implementation.

3.1 Kernel performance limiter:

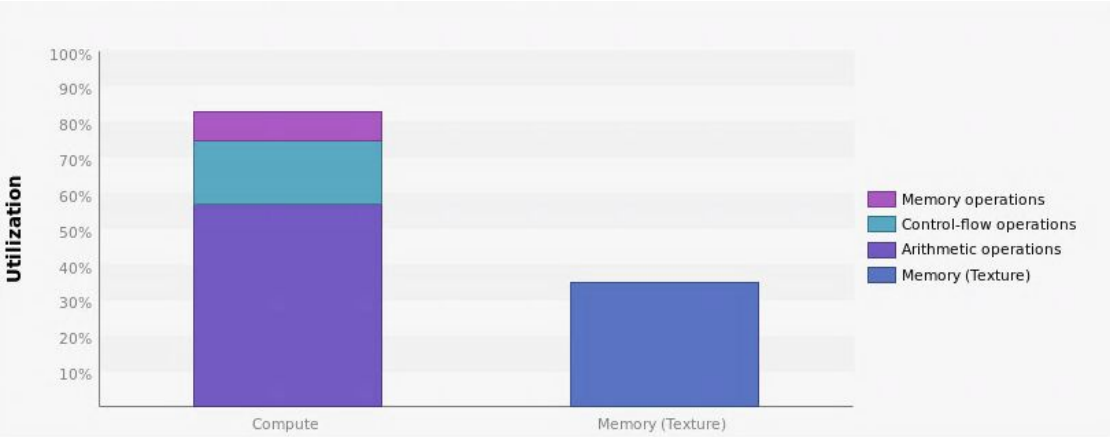
ORIGINAL:



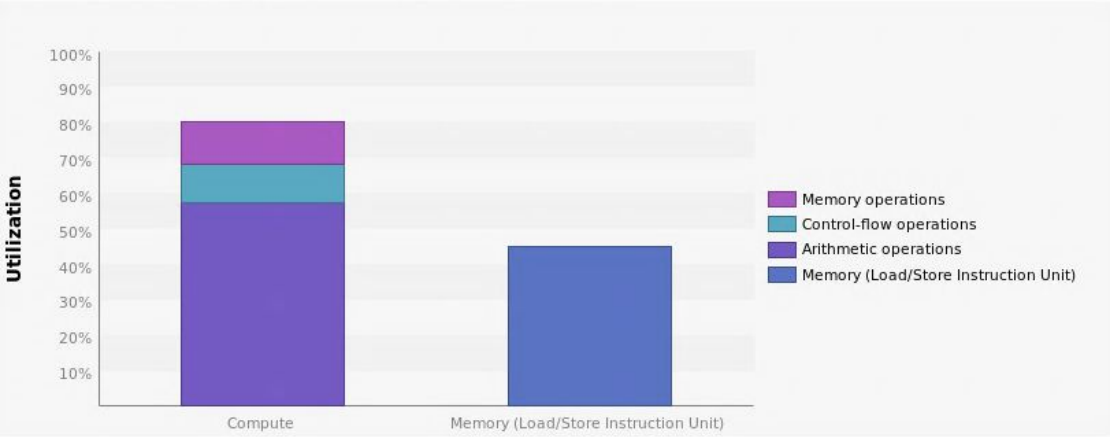
UNROLL:



CONSTANT:



SHARED:

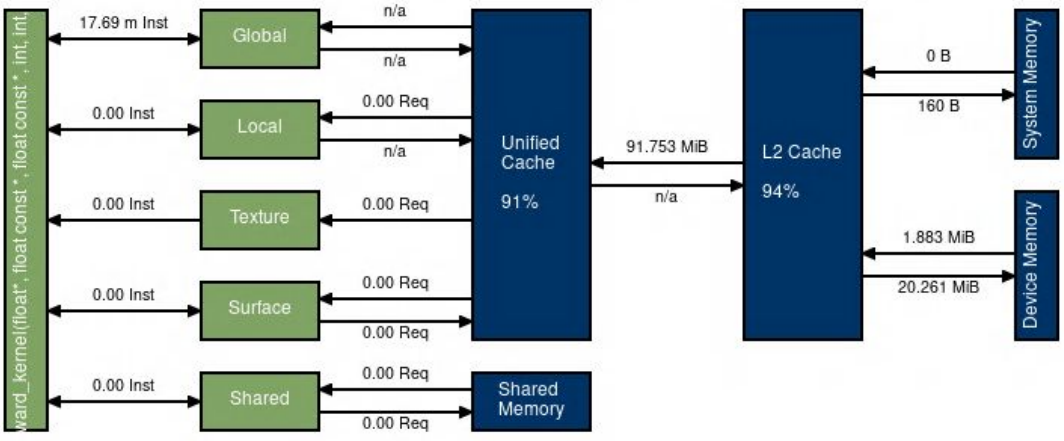


Analysis: The computation utilization of CONSTANT and SHARED are higher than ORIGINAL, which is a good improvement. But that of UNROLL is very limited. It indicates the performance of kernel is most likely limited by arithmetic operations and memory operations. The main parts in UNROLL are executed sequentially, which may lead to this issue.

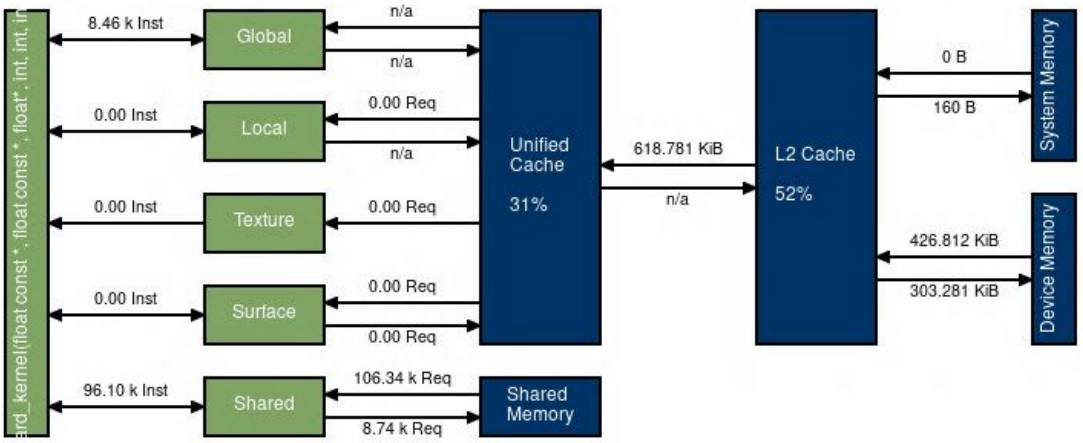
3.2 Divergent branches:
ORIGINAL: 11.4%
UNROLL: 0
CONSTANT: 16.5%
SHARED: 64.3%

Analysis: We can notice that the control divergence in the shared memory convolution is very high. We use the strategy 2, where all threads will load image data into shared memory but only some of them will do the computation. Those active and inactive threads in the same wrap may be the reason of this control divergence. In the following optimization, we can use other strategies to solve this problem.

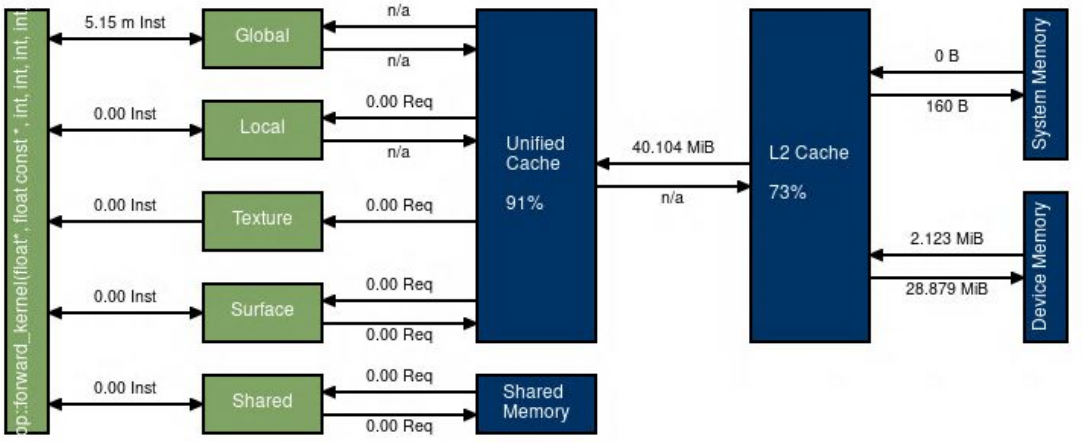
3.3 Memory statistics:
ORIGINAL:



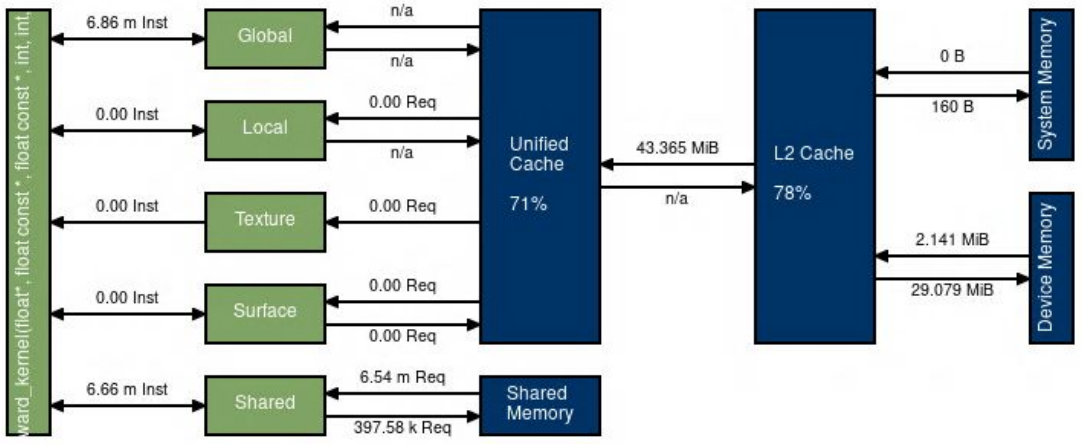
UNROLL:



CONSTANT:



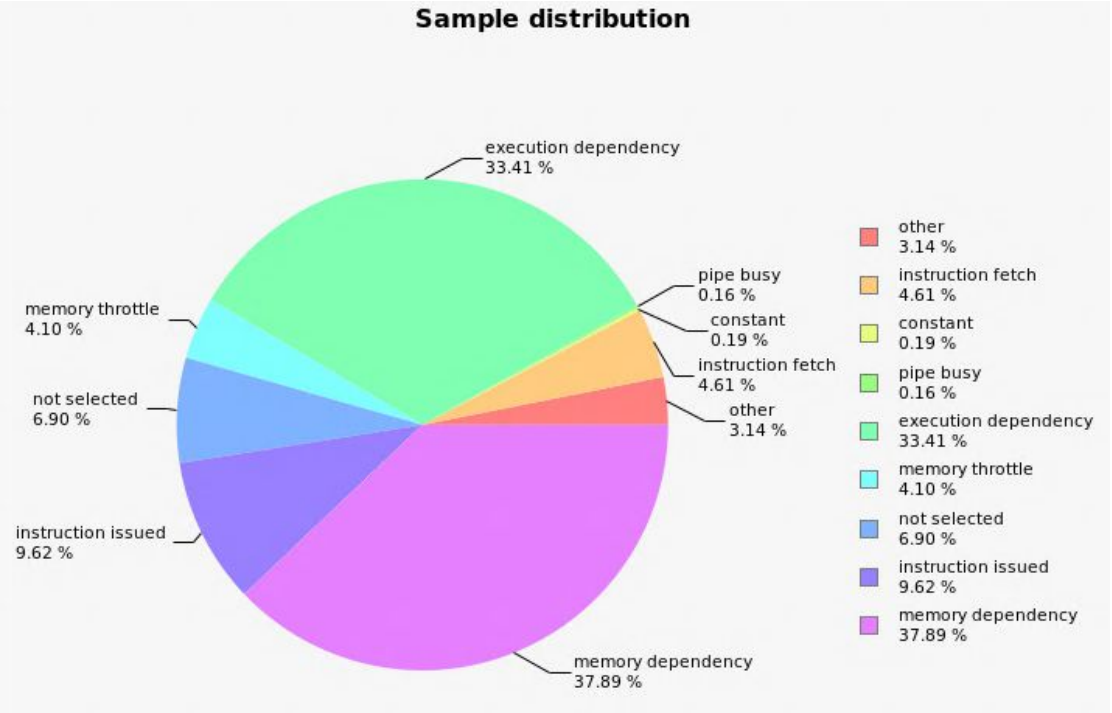
SHARED:



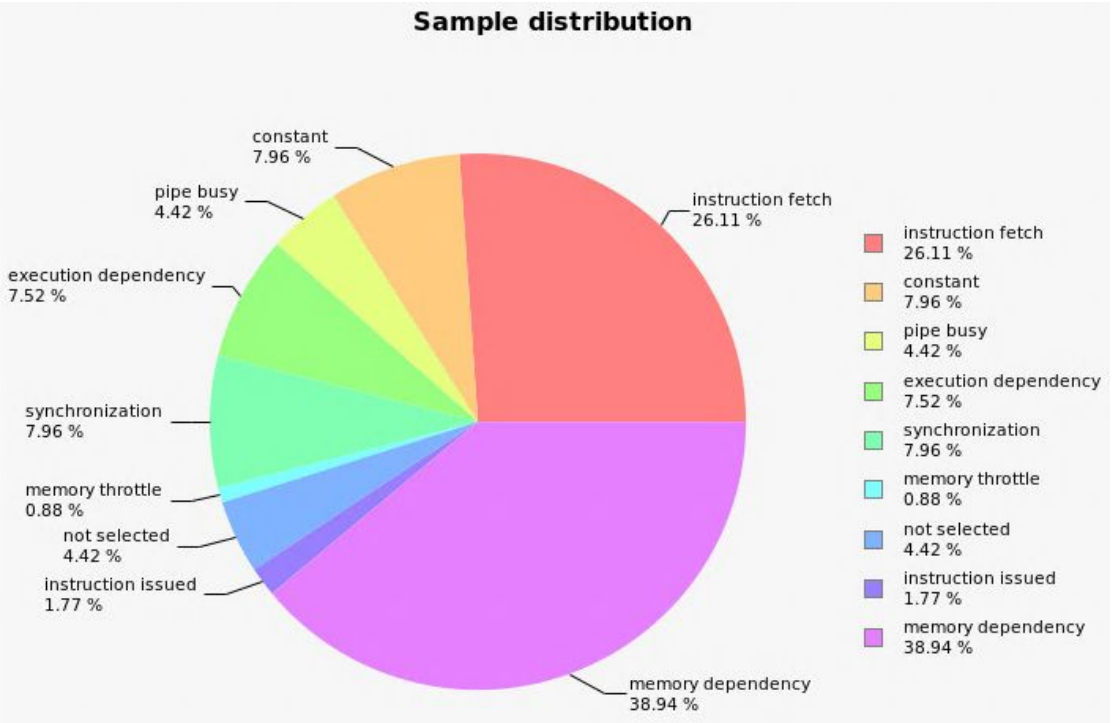
Analysis: We can notice that the UNROLL and SHARED use much shared memory, and SHARED has higher request rate than UNROLL. We can also find that the utilization of unified cache and L2 cache in UNROLL are not high, because the size of each unrolled image multiplication is not big, so it doesn't need to transfer much data.

3.4 Kernel profile - PC sampling:

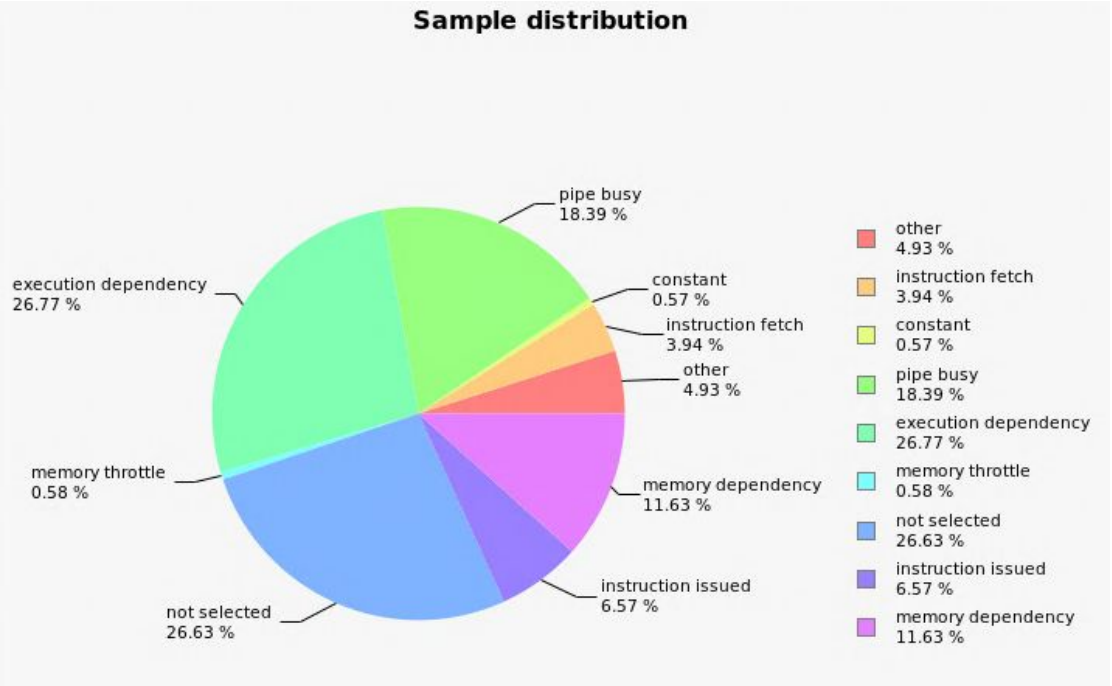
ORIGINAL:



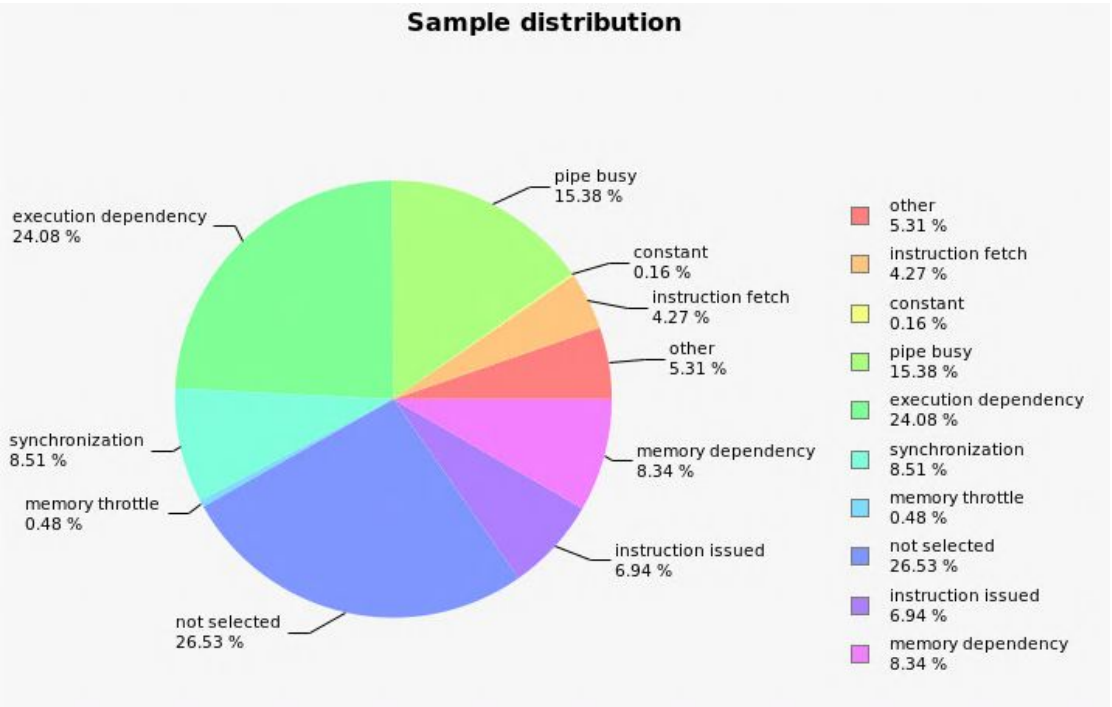
UNROLL:



CONSTANT:



SHARED:



Analysis: The synchronization section in UNROLL and SHARED takes relatively much time in the whole process, because they all have to set a barrier after loading shared memory. And we can find that the instruction fetch takes a long time in the UNROLL, because there is a loop of image size. In each iteration we will launch 2 kernel: one for unroll input, one for matrix multiplication. In the future optimization, we can increase the number of blocks in Z dimension and decrease the iteration of loop.

4. Implementation Details

Unroll + shared-memory Matrix multiply

In this optimization, we built a convolutional layer that uses shared-memory Matrix multiply by unrolling the inputs into matrices. The central idea is to linearize the Input Features (unroll) so that multiplying the two matrices gives us our output. We launched 2 kernels in order, *unroll_input* and *forward_kernel*, separately for unrolling input matrix and doing the matrix multiplication.
Why is it efficient in this way?
First, the replicated input features are shared among different output channels. Second, the convolution filter weights are reused within each output tile. In addition, the memory bandwidth is reduced by using shared-memory in Matrix Multiply. Implementing convolutions with matrix multiplication can be very efficient, since matrix multiplication is highly optimized on all hardware platforms.

Shared-memory convolution & Weight matrix (kernel values) in constant memory

We first improved the original convolution implementation by storing the weight matrix(kernel values) into constant memory. Because the constant memory allocation has to be static with constant size, we allocated a float array of length 10000. This array takes 10000 * 4 = 40KB which is smaller than the constant memory limit 64KB and is enough for the largest weight matrix of size M * C * K * K = 24 * 12 * 5 * 5 = 7200 floats. We then modified the macro in the kernel code to use the constant array instead of the matrix k when desired.
Further, we implemented a shared-memory 3d convolution similar to the one in mp4. We use the strategy 3 to load our shared memory, which means that the Block size covers input tile and after we load the input tile in one step, the extra threads are turned off when calculating the output. The allocation of shared memory is also required to be constant. We allocated a shared memory that is large enough to hold the largest input tile we have, which has 12 channels. Since the width of filter is 5, block size should be TILE_WIDTH + 5 - 1.

Benefits of shared memory convolution:
Taking advantage of cached memories. Reducing the memory bandwidth by reusing the input data within each block.

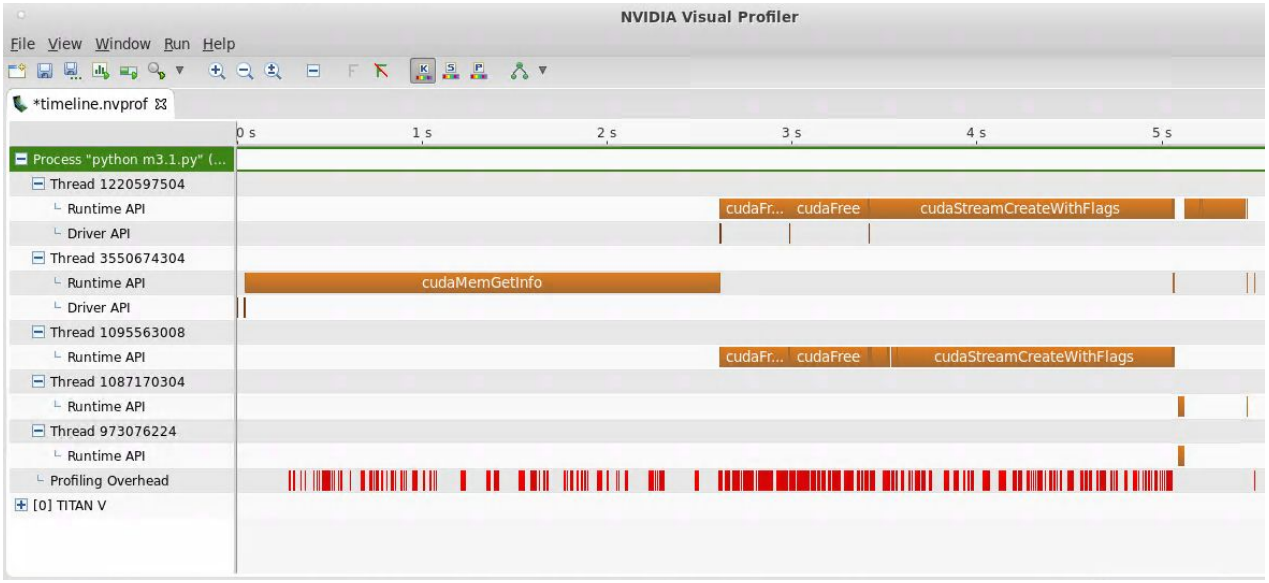
Benefits of weight matrix in constant memory:
Since mask is used by all threads and not modified, we can make sure all threads in a warp access the same locations every time. Taking advantage of the constant memory we can reduce global memory access without consuming shared memory.

Milestone 3

1. Demonstrate nvprof profiling the execution and result of nvvp

Deliverables
1. All kernels that collectively consume more than 90% of the program time:
mxnet::op::forward_kernel mshadow::cuda::MapPlanLargeKernel volta_sgemm_128x128_tn op_generic_tensor_kernel cudnn::detail::pooling_fw_4d_kernel
2. All CUDA API calls that collectively consume more than 90% of the program time:
cudaStreamCreateWithFlags cudaMemGetInfo cudaFree
3. Program run time of our GPU implementation with m3.1.py
user: 5.16, system: 3.33 , elapsed: 0:06.61
4. Results and Op Times:
python m3.1.py 10000
Op Time: 0.032341 Op Time: 0.093268 Correctness: 0.7653
python m3.1.py 1000
Op Time: 0.003130 Op Time: 0.009871 Correctness: 0.767
python m3.1.py 1000
Op Time: 0.000272 Op Time: 0.000943 Correctness: 0.76

This is the result of program execution.



It is the result of "::forward:1". We can find that the top thread is the CPU and the following thread is GPU. It shows the time consumed for each session.

Milestone 2

1. All kernels that collectively consume more than 90% of the program time:

- [CUDA memcpy HtoD]
- volta_scudnn_128x64_relu_interior_nn_v1
- volta_gcgemm_64x32_nt
- void fft2d_c2r_32x32
- volta_sgemm_128x128_tn
- void op_generic_tensor_kernel
- void fft2d_r2c_32x32

2. All CUDA API calls that collectively consume more than 90% of the program time:

- cudaStreamCreateWithFlags
- cudaMemGetInfo
- cudaFree

3. Explanation of the difference between kernels and API calls:

Kernel calls are activities executed in parallel by CUDA threads. And the time here represents the execution time on the GPU.
API calls list CUDA Runtime/Driver API calls. And time information here means the execution time on the host.
The time consumed by API calls are much more than that of kernel calls, because CUDA kernel launches are asynchronous and the host code need to wait for kernels to finish.

4. Output of rai running MXNet on the CPU:

Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}

5. Program run time of CPU:

user: 19.24, system: 6.74, elapsed: 9.60

6. Output of rai running MXNet on the GPU:

Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}

7. Program run time of GPU:

user: 4.92, system: 3.21, elapsed: 4.69

8. Whole program execution time:

mp1.1: 8.96s, mp1.2: 4.66s, mp2.1: 65.33s

9. Op Times:

Op Time: 12.348230
Op Time: 78.750367