Derek Alyne (dalyne2)
Joshua Sanchez (jsanch84)
Howard Shan (howards2)
ECE 408
Professor Sanjay Patel
8 March 2019

Report

I. Milestone 1
   A. Include a list of all kernels that collectively consume more than 90% of the program time.
      - [CUDA memcpy HtoD]
      - void cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>(int, int, int, float const *, int, float*, cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, kernel_conv_params, int, float, float, int, float, float, int, int)
      - volta_cgemm_64x32_tn
      - void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)
      - void fft2d_c2r_32x32<float, bool=0, bool=0, unsigned int=1, bool=0, bool=0>(float*, float2 const *, int, int, int, int, int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*, int2, int, int)
      - Volta_sgemm_128x128_tn
      - void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)

- `void fft2d_r2c_32x32<float, bool=0, unsigned int=0, bool=0>(float2*, float const *, int, int, int, int, int, int, int, int, int, cudnn::reduced_divisor, bool, int2, int, int)`

B. Report: Include a list of all CUDA API calls that collectively consume more than 90% of the program time.
   - `cudaStreamCreateWithFlags`
   - `cudaMemGetInfo`
   - `cudaFree`

C. Report: Include an explanation of the difference between kernels and API calls
   - API calls are functions defined by the CUDA library, such as `cudaMalloc` and `cudaMemcpy`, while kernels are functions that the programmer (or other libraries) defines to run on the gpu. They are a minimal set of extensions to the C language and a runtime library to help the programmer interface with the gpu. Kernel functions are typically run a large number of times in parallel, using multiple blocks and threads. According to the CUDA documentation, "A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>>execution configuration syntax."

D. Report: Show output of rai running MXNet on the CPU
   - `* Running /usr/bin/time python m1.1.py`
   - `Loading fashion-mnist data... done`
   - `Loading model... done`
   - `New Inference`
   - `EvalMetric: {'accuracy': 0.8236}`
   - `8.91user 3.64system 0:05.11elapsed 245%CPU (0avgtext+0avgdata 2470716maxresident)k`
   - `0inp`
   - `uts+2824outputs (0major+666444minor)pagefaults 0swaps`

E. Report: List program run time
   - 5.11 seconds

F. Report: Show output of rai running MXNet on the GPU
   - `* Running /usr/bin/time python m1.2.py`
   - `Loading fashion-mnist data... done`
   - `Loading model... done`
   - `New Inference`
   - `EvalMetric: {'accuracy': 0.8236}`
   - `4.43user 3.37system 0:04.33elapsed 180%CPU (0avgtext+0av`
   - `gdata 2841612maxresident)k`
   - `8inputs+1728outputs (0major+660933minor)pagefaults 0swaps`

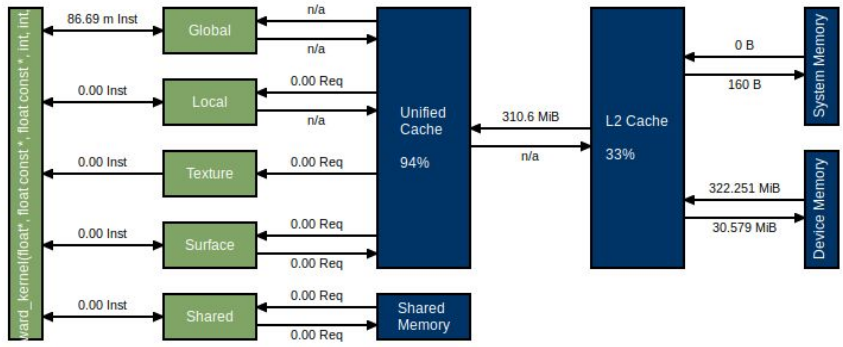
G. Report: List program run time

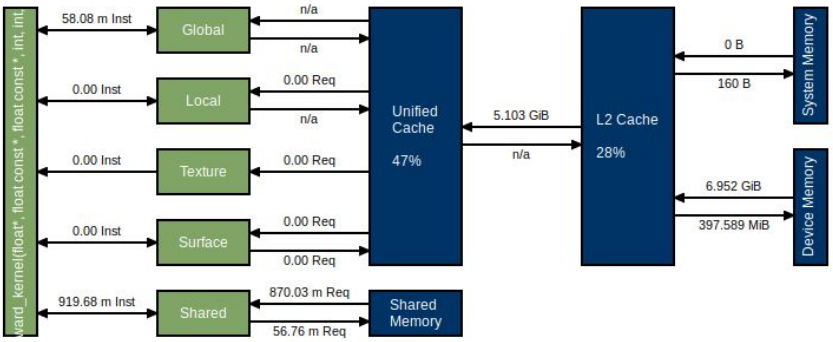
- 4.33 seconds

II. Milestone 2
  A. Whole Program Execution Time
    - 12.13 seconds
  B. OpTimes:
    - 2.583861 seconds
    - 7.785734 seconds

III. Milestone 3
  A. N/A

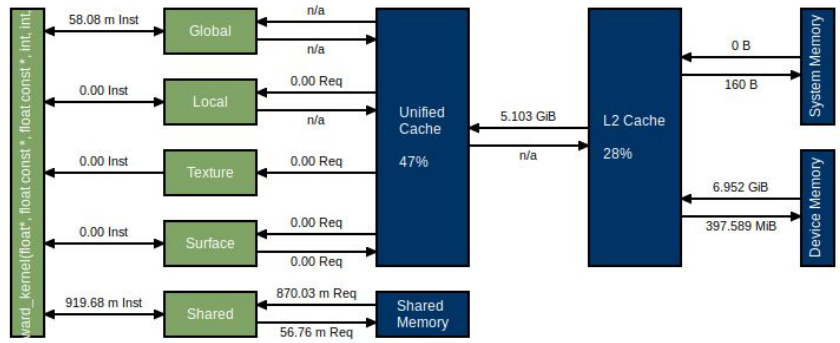
IV. Milestone 4
  A. Shared memory convolution

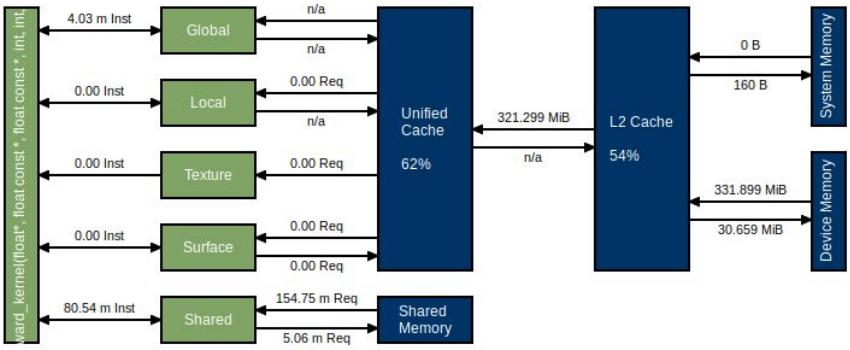




- The top diagram is the memory statistics for the basic forward convolution implementation, and the second one is for the shared memory convolution optimization. As can be seen near the bottom of both diagrams, the bottom one uses a significant amount of shared memory (in contrast with the top one which does not use any shared memory), which results in faster access of elements.

B. Constant Memory for Mask weights

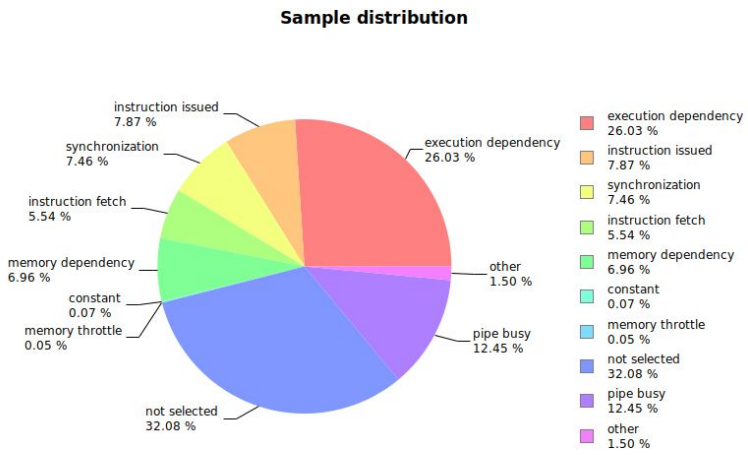


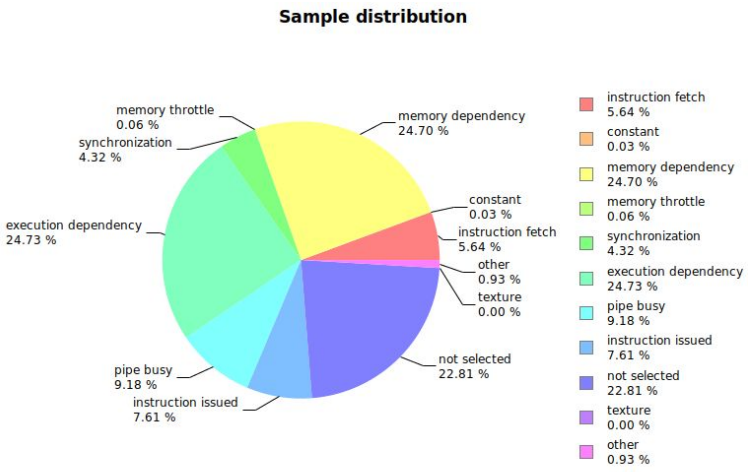


- The top diagram shows the memory statistics for shared memory convolution without using constant memory, and the bottom diagram shows the memory statistics with the constant memory. As can be seen, the amount of data retrieved from the device memory from almost 7GB to around 330MB. This is a very large decrease that helped increase the performance of our convolution layer.

C. Double buffering in kernel

- After using shared memory and utilizing constant memory, we were making two __syncthreads() calls in our kernel code for every for loop iteration. We realized that we could reduce this to only one __syncthreads() call by using a double shared memory buffer. This means that we are using two shared memory buffers, and alternating with every iteration so that we can write to the new buffer without worrying about overwriting data from threads that still need it.





- The top pie chart is from the code without using a double buffer, and the bottom one is with code that does utilize a double buffer. As can be seen, the synchronization stalling percentage decreases from 7.46% to 4.32% - over a 30% reduction.