COL380 Assignment 3

Divyanka Chaudhari - 2019CS50429, Aradhye Agarwal - 2020CS50417

CUDA Kernels

Convolution Kernel (`convMultiChannelKernel`) The convolution kernel is designed to perform 2D convolution over multi-channel input, such as images with multiple color channels. This kernel handles convolution operations for both single and multiple input and output channels, making it versatile for different layers within a CNN architecture.

Kernel Design:

- **Parameters**:
  - `input`: The input feature map or image.
  - `kernels`: The convolutional filters.
  - `output`: The output feature map.
  - `inputHeight`, `inputWidth`: Dimensions of the input image.
  - `kernelSize`: The size of the convolutional kernel (assumed to be square).
  - `numInputChannels`: Number of channels in the input image.
  - `numOutputChannels`: Number of filters, determining the number of channels in the output feature map.
- **Computation**:
  - Each thread calculates one pixel of the output feature map.
  - It iterates over every input channel, applying the corresponding kernel and summing up the results to produce the output for each output channel.
  - The kernel involves a nested loop over the kernel dimensions, multiplying corresponding input pixels and kernel weights and accumulating the results.

Optimization Strategies:

- **Memory Access**: Utilizes coalesced memory accesses for reading the input and kernels, minimizing memory latency.
- **Shared Memory**: For larger kernels, using shared memory to cache input tiles can significantly reduce global memory accesses.
- **Loop Unrolling**: The inner loops over the kernel dimensions can be unrolled to improve instruction-level parallelism.

Max Pooling Kernel (`maxPoolingKernel`) The max pooling kernel reduces the spatial dimensions of the input feature map, a critical operation for down-sampling in CNNs.

Kernel Design:

- **Parameters**:

  - `input`: The input feature map.
  - `output`: The output feature map after pooling.
  - `inputHeight`, `inputWidth`: Dimensions of the input.
  - `outputHeight`, `outputWidth`: Dimensions of the output.
  - `numChannels`: Number of channels in the input and output (remains unchanged).
  - `stride`, `poolSize`: Stride and size of the pooling window.
- **Computation**:
  - Each thread computes one element of the output feature map by selecting the maximum value within the pooling window of the input feature map.
  - This involves iterating over the poolSize within the given input channel, finding the max value, and writing it to the output.

Optimization Strategies:

- **Parallel Reduction**: For finding the maximum within the pooling window, parallel reduction techniques can be employed to enhance performance.
- **Constant Memory**: Pooling parameters such as stride and poolSize can be stored in constant memory for faster access.

Activation Function Kernel (`reluKernel`) The ReLU (Rectified Linear Unit) kernel applies the ReLU activation function element-wise to the input data. This non-linear activation function is defined as `f(x) = max(0, x)`.

Kernel Design:

- **Parameters**:
  - `data`: The input and output data on which the ReLU function is applied.
  - `count`: The total number of elements in the input data.
- **Computation**:
  - Each thread processes one element of the input data, applying the ReLU function directly.

Optimization Strategies:

- **Thread Utilization**: The straightforward nature of ReLU allows for full utilization of threads, as each thread independently computes the ReLU function for one element.

Conclusion

The CUDA kernels described above form the computational backbone of the GPU-accelerated portions of the CNN. By leveraging the parallel computing capabilities of CUDA, significant speedups can be achieved compared to serial computations on the CPU. Each kernel is optimized to reduce memory latency, maximize hardware utilization, and ensure efficient computation, laying the groundwork for fast and scalable neural network training and inference.

CUDA Streams

CUDA streams allow for concurrent execution of kernels and memory operations, providing a mechanism to overlap data transfer with computation, thus improving the overall efficiency of the application.

- **Concurrent Kernel Execution**: By assigning different computational tasks to separate streams, we were able to execute multiple kernels in parallel, significantly reducing idle time on the GPU.
- **Overlap of Data Transfer and Computation**: Critical to achieving high throughput, we leveraged streams to overlap memory transfers (both H2D and D2H) with kernel execution. This was particularly beneficial in stages where the processing of one layer could proceed concurrently with the data transfer of the next.
- **Stream Synchronization**: Synchronization points were strategically placed to ensure data dependencies were respected, without unnecessarily stalling the pipeline. This required a careful analysis of the computation graph to identify independent operations.

Performance Impact

For 10,000 images, without Streams took ~23s and with stream took ~20s.