GPU Programming with CUDA and Triton

Assignment 01

V S Vishnu

23B3909

Overview

In this week's assignment we were asked to analyze a workload that we may want to accelerate later in the course and understand how it fits into the GPU execution model without writing CUDA code yet. I have tried to do the same here with an assignment problem I had in my `EE353` (Introduction to DS/ML) course. This part of the assignment was to train an Artificial Neural Network (ANN) on the Hitters dataset.

Operation Breakdown

As this was a machine learning problem, there were multiple operations to chose from. I am choosing the Linear + ReLU forward for analysis here. Given an input matrix X, weight matrix W and bias b, the output is Y=max(0, XW+b).

Data shape or tensor dimension for the ANN model is as follows:

- Input X $\in \mathbb{R}^{263\times 20}$
- Weights W $\in \mathbb{R}^{20\times 5}$
- Bias b $\in \mathbb{R}^{5}$
- Output Y $\in \mathbb{R}^{263\times 5}$

This implies that the batch size is 263, number of input features is 20 and number of hidden nodes is 5.
Now the opportunities for parallelism identified by me are the following:

- The system is embarassingly parallel across samples (i) and the hidden nodes (j) as every element in the sets i and j are independent of each other for computations.
- In the inner loop k, as the partial sums are dependent on previous additions, this has limited parallelism (can be parallelised to a certain extent with tree reduction).

Pseudocode for the implementation:

Algorithm 1 Feedforward Layer Computation

```
for $i = 0$ to $N - 1$ do
    for $j = 0$ to $H - 1$ do
        $sum \leftarrow 0$
        for $k = 0$ to $D - 1$ do
            $sum \leftarrow sum + X[i][k] \cdot W[k][j]$
        end for
        $sum \leftarrow sum + b[j]$
        $Y[i][j] \leftarrow \max(0, sum)$
    end for
end for
```

Compute vs Memory Behavior

Here for each individual output element, these are the details regarding the arithmetic intensity: **Computation** has 20 multiplications, 19 additions, 1 bias addition and 1 ReLU comparison ie, a total of 41 floating point operations.

Memory Access has to read 20 elements from X, 20 elements from W, 1 bias and write 1 output ie, a total of 42 memory accesses. This implies that number of FLOPs per memory access is around 1. Hence, for larger feature dimensions, the same operation would become compute-bound; however, for the Hitters dataset with only 20 input features, memory access and kernel launch overhead are significant contributors to runtime and hence makes this case memory bound.

Regarding data **reuse opportunities**: weights W[k][j] are being used across all 263 samples and hence is an excellent reuse candidate and the input matrix X[i][k] is being used across all 5 hidden units and hence has moderate reuse.

Also, the computation contains **no cross-output dependencies**, allowing independent threads to compute different output elements without synchronization.

Expected GPU Mapping

- Thread mapping: As each sample and hidden node is independent of one another, GPU threads can compute individual output element Y[i][j], which can be indexed as a 2D grid.
- Block mapping: Blocks, group together threads computing nearby output elements so that they can reuse the same portions of the input. A problem here is that modern GPU's can run hundreds or thousands of threads concurrently but 263×5 is relatively a small number and hence the kernel won't fully occupy the GPU. Hence further scaling of variable can be done here.
- We already talked about reusability in the previous section. Weight matrix can be cached in the shared memory so as to reduce global memory accesses significantly.
- There isn't much worry regarding divergence in this case though as the only real chance of divergence occurs at the ReLu activation function. Although threads within a warp could diverge here, GPUs will be able to handle this simple divergence.
- Memory access: Input matrix stored as row major as this is best for caching due to minimal access time. Weight matrix is accessed column wise for the matrix multiplication so we will have to save W^T instead of W in the shared memory for ease of access.
- While the current configuration underutilizes the GPU, increasing batch size, feature dimension, or hidden units leads to improved occupancy and compute efficiency, making the mapping suitable for larger neural networks.

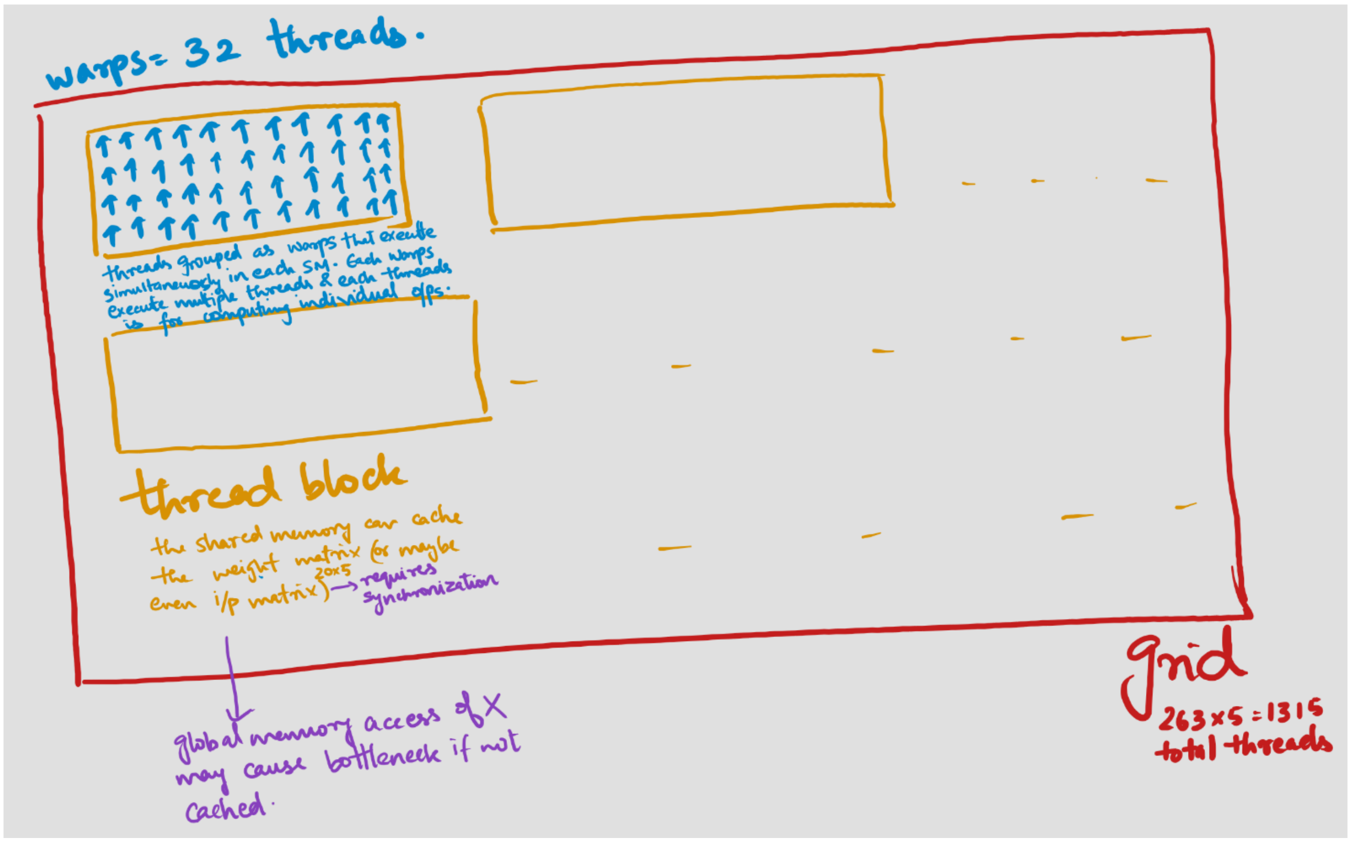
warps= 32 threads.
threads grouped as warps that execute simultaneuosly in each SM. Each warps execute multiple threads & each threads is for computing individual ops.
thread block
the shared memory can cache the weight matrix (or maybe even i/p matrix)
20x5
→ requires synchronization
global memory access of X may cause bottleneck if not cached.
grid
263x5=1315 total threads

Figure 1: CUDA Execution Model Diagram

The grid represents a logical iteration space; blocks are scheduled dynamically onto SMs.