Lab 3 Report Yash Shah

CS 133 – 11/15. UID: 405-565-567

Please explain the parallelization strategies (how data is distributed and how computation is done) you applied for each loop in the CNN program (convolution, max pooling, etc) in this lab. What made you choose this strategy and how is it justified by the specs of our GPU device?

I used a 3D grid and block structure to parallelize the convolutional neural network (CNN) operations on a GPU. This structure is crucial for effectively managing the parallel computation power of the GPU. In my implementation, each thread in the GPU independently handles a specific segment of the input image and feature map. This approach ensures that the GPU's cores are efficiently utilized, with each core performing a part of the overall computation concurrently.

For the convolution operation, I employed nested loops within each thread. This setup allows each thread to compute the convolution for a particular output feature map. The convolution involves performing element-wise multiplications and additions across the input feature maps and the kernel.

Additionally, I integrated the max pooling operation within each thread. After the convolution, each thread performs max pooling to reduce the spatial dimensions of the convolved features. By handling this operation in-thread, I minimized the need for memory accesses and inter-thread communication, which are typically costly in terms of performance.

This strategy is particularly effective given the architecture of modern GPUs. GPUs are designed for high-throughput, low-latency operations, making them perfect for deep learning tasks such as CNNs. Their architecture allows for simultaneous processing of large data volumes, which is exactly what is needed for the computationally demanding tasks in CNNs. By distributing the workload across multiple threads and optimizing memory usage, my kernel can process large datasets quickly and efficiently, leveraging the full power of the GPU's architecture.

Please describe any optimization you have applied.

3D Grid and Block Structure: Efficiently mapped threads to the data in the input and output tensors, improving computational efficiency and memory access patterns.

Loop transformation: Transformed these loops to utilize fewer iterations while covering the same computation space. Doubling the increments of h and w variables ensured that each thread did more work.

In-Thread Max Pooling & ReLu: Performed pooling operations within each thread to minimize global memory accesses, enhancing performance.

Loop Unrolling: Employed loop unrolling techniques in the max pooling to decrease loop control overhead and fill the 2x2 convolutional grid.

Please report the dimension of threads per block and blocks per grid that provides the best performance for your kernel. Then please look up the number of multiprocessors and CUDA cores per multiprocessor in the Tesla M60 GPU. Does the GPU specification corroborate your chosen dimensions? If not, please discuss the probable reason.

My kernel configuration for the Tesla M60 GPU is as follows:

- Threads per Block: 1024
- Blocks per Grid: 3136

The Tesla M60 GPU has 20 multiprocessors with 128 CUDA cores per multiprocessor. This configuration aligns well with the GPU's capabilities:

- Threads per Block: The chosen number (1024) is the maximum allowed and efficiently utilizes each block's capacity.
- Blocks per Grid: The large number of blocks (3136) ensures continuous GPU utilization, keeping all multiprocessors busy.

Overall, the configuration effectively matches the Tesla M60's architecture.

Please evaluate the performance of at least four (4) different optimization techniques that you have incrementally applied and explain why such optimization improves the performance. Changing parameters does not count as one optimization. Significant code changes are needed between versions to be counted. In your report, please include the most important changes in your code for each optimization. Git commits, branches or tags make this process easy.

When optimizing the CUDA kernel for the convolutional neural network, I applied 3 main optimization techniques, each with significant code changes to enhance performance.

Initially, my focus was on optimizing memory access patterns. By ensuring that consecutive threads access consecutive memory locations, I could reduce memory access latency and improve the overall efficiency of the code. This change was primarily about rearranging how data was stored and accessed within the loops. This minimized the number of memory transactions, leading to better utilization of the GPU's memory bandwidth with GFlops around 23 GFlops.

Next, I transformed the loops to utilize fewer iterations while covering the same computation space. By doubling the increments of h and w variables and adjusting the conditions inside the if-statement, I ensured that each thread did more work. This change reduced the total number of thread invocations, leading to an increase in performance from 23 GFlops to around 40 GFlops.

Lastly, I implemented loop unrolling in the convolution operation. By manually expanding the loops for the convolution matrix computation and max pooling, I reduced the overhead associated with loop control. This optimization was particularly effective in increasing the instruction throughput, as it shot performance from 40 GFlops to around 160 GFlops.

If I were to have figured it out in time, I would have leveraged shared memory, which is much faster than global memory. I would modify the code to load frequently accessed data into shared memory before the convolution operation. This would require additional synchronization steps to ensure all threads had completed their memory operations before proceeding, but it would have significantly reduced the memory latency, most likely around the 200-300 GFlops range.

Please include a table that shows the performance for different threads per block and blocks per grid. Please report the performance for at least 3x3=9 different configurations including the one that provides the best performance. Your dimensions of thread blocks and the grid should probably be factors of loop bounds or powers of 2.

Blocks per Grid	Threads per Block	Performance (GFlops)
(16 14 14)	(16 8 8)	160
(16 7 28)	(16 16 4)	161
(16 28 7)	(16 4 16)	159
(32 14 7)	(8 8 16)	162
(64 14 7)	(4 8 16)	110
(8 28 14)	(32 4 8)	87
(8 14 28)	(32 8 4)	86
(4 28 28)	(64, 4, 4)	49
(2 56 28)	(128 2 4)	46