Part B : Implement and optimize DC in CUDA (GPU)

Himanshu Devrani
Computer Science and Automation
Indian Institue of Science
Sr No : 23106
himanshud@iisc.ac.in

Dev Gandhi
Computer Science and Automation
Indian Institue of Science
Sr No : 22634
devtejas@iisc.ac.in

***Abstract*—This report presents the implementation of a Dilated Convolution Algorithm on a GPU using the CUDA framework. This Report Consists of design considerations of kernel and thread blocks and also evaluates the achieved performance gains.**

I. Introduction

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the *host* refers to the CPU and its memory, while the *device* refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches *kernels* which are functions executed on the device. These kernels are executed by many GPU threads in parallel.

A. GPU Configuration

We have performed this task in **NVDIA GetForce RTX 3060 Laptop GPU**.

Some Technical Specification are as follows.

- Compute Capability : 8.6
- No of Streaming Multiprocessors: 30
- CUDA Cores/SM : 128
- Maximum No of Threads per Thread Block : 1024
- Maximum number of resident blocks per SM : 16
- Warps size : 32 Threads
- Maximum Warps per SM : 48 warps

B. Dilated Convolution in CUDA

We Have seen definition of DC(Dialleted Convolution) previously. we want to find output matrix. And the logic is we create threads and each thread computes an element of the output matrix.So at a time parallely we could compute the output matrix.

Means No of threads = No of elements in output matrix

II. Implementation Details

A. Thread Blocks And Grid Size

We visualize the output matrix in 2D so it is more convenient to construct a thread block in 2 dimensions which contains threads in both axes. We obtain results for multiple size of thread blocks.(Ref Table 1) We need **a number of threads = number of output cells in the output matrix**.

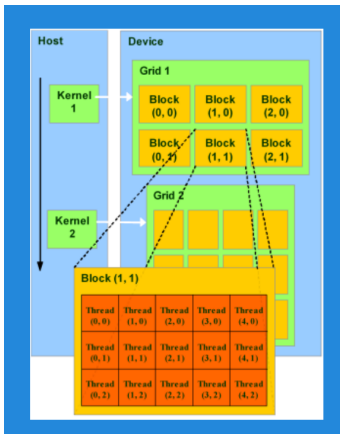

Fig. 1. 2d Grid and 2d Thread Block Visulization

To compute the size of the 2-dimensional grid covering the entire output matrix, we can use the following:

$$\text{dimBlock} : \text{dim3}(32, 32, 1)$$

$$\text{dimGrid} : \text{dim3}\left(\lceil\frac{\text{outputCol}}{32.0}\rceil, \lceil\frac{\text{outputRow}}{32.0}\rceil, 1\right)$$

Here, we assume that the thread block size is 32×32. The use of $\lceil\frac{\text{outputCol}}{32.0}\rceil$ and $\lceil\frac{\text{outputRow}}{32.0}\rceil$ ensures that the grid dimensions cover the entire output array. Note that the ceiling function is used to account for cases where the array size is not an exact multiple of the thread block size, so some threads may not map to the output array for those indices.

For example, consider an output array with dimensions outputCol = 70 and outputRow = 45.We Assumed Thread Block Dimension 32x32.The resulting grid dimensions would be 70/32.0 = 3 for columns and 45/32.0 = 2 for rows. Consequently, the grid would be organized as 3×2 thread blocks.

III. Performance Analysis

We conducted performance testing on the CUDA implementation by comparing the execution times between the standard unoptimized code and the CUDA code.

TABLE I
EXECUTION TIME FOR CUDA AND NORMAL IMPLEMENTATION

Input Matrix	Cuda Implementation (ms)	Normal Code (ms)
128 (I) & 64 (K)	1411.55	190.3
4096 (I) & 13(K)	1548.76	27911.9
4096 (I) & 128 (K)	2541	2241080
8192 (I) & 64(K)	2685.73	1063800
16384 (I) & 13 (K)	3019.4	183041

We have taken 32x24 size thread block because It is most optimal dimension among all possible ones.We have tested and analyzed the same CUDA implementation with different Thread Block Dimensions. More details are given in subsequent sections.

Here we executed normal unoptimized code and CUDA Implimentation (Without Constant Memory) Code with various Input Matrix (I) and Kernel Matrix (K).

In 128(I) and 64(K) We got more time in Cuda and less in Normal exuction. Because launching a kernel on a GPU carries and "approximately" fixed cost overhead, per kernel launch. This means that if you size the amount of work such that your CPU can do it in less than that amount of time, there is no way the GPU can be faster.But generally we use GPU for Bigger Inputs so it can be ignored.

Here We Compared the Difference between Normal execution time and Cuda Implementation.

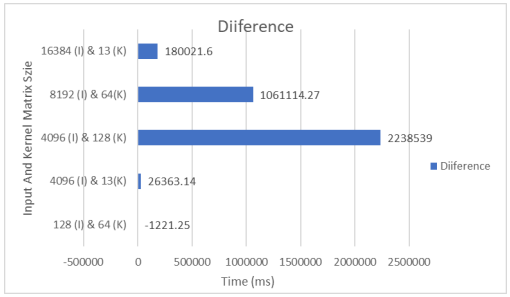

Fig. 2. Difference between Normal execution - Cuda Implimentation

A. *Optimizing Thread Block Dimension*

Different Thread Block sizes affect the execution time of the code.It is because it will impact the warp occupancy.

Occupancy: Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that are actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. These are some results from Nsight compute and we get similar patterns for all the inputs.

So here we see that for 768 threads it is giving maximum warp occupancy. Having more threads doesn't always mean better performance One of the reasons 1024 threads is not giving good warp occupancy is Hence we consider 768 threads (i.e. 32x24).

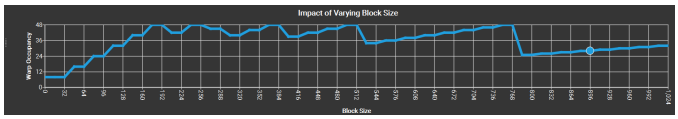

Fig. 3. 4096x4096 Input Matrix And 64x64 Kernel Matrix

We Also perform Cuda Code with different Thread Block Size and the result are as follows.(Fig 4 & 5)

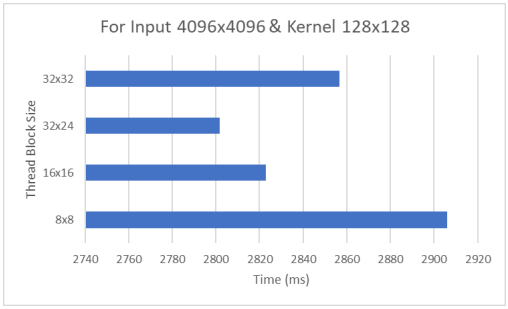

Fig. 4. 4096x4096 Input Matrix And 128x128 Kernel Matrix

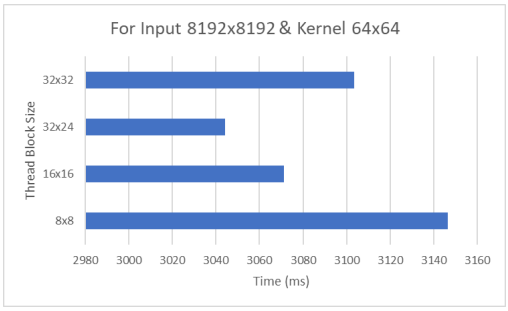

Fig. 5. 8192x8192 Input Matrix And 64x64 Kernel Matrix

So 32x24 is a better choice of thread block size for this program with this configuration of GPU. Because for the majority of the input we have seen the same pattern.

B. *Implimenting Constant Memory For Further Optimization*

The CUDA language makes available another kind of memory known as constant memory. As the name may indicate, we use constant memory for data that will not change over the course of a kernel execution.
NVIDIA hardware provides 64KB of constant memory that it treats differently than it treats standard global memory. In some situations, using constant memory rather than global memory will reduce the required memory bandwidth.

A couple of things to notice about the convolutional operation are that the convolutional kernel is never modified and that it is almost always fairly small. For these reasons, we can increase efficiency by putting the convolutional kernel in constant memory.

The CUDA runtime will initially read the convolutional kernel from global memory as before however now it will cache it since it knows it will never be modified. The disadvantage of constant memory is that it is small (only 64 KB) but since our convolutional kernel is also small this shouldn't be a problem. Very few changes need to be made to put M in constant memory.

If Kernel Matrix size $>$ 64KB then we can't put kernel matrix in constant memory so in that case we have to put it in global memory but still it can perform well than normal code.(Table 1)

Here Are some result that compares with constant memory and without constant memory implementation.(Ref Fig 6)
Here thread block size is 32x24 for all cuda programs.

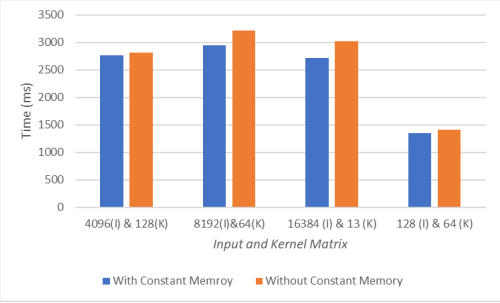

Fig. 6. Analysis of With and Without Constant Memory

IV. Conclusion

When dealing with a relatively small kernel size, which is often the scenario, In such cases, the better approach is storing the kernel in constant memory. Constant memory, designed specifically for data that remains unchanged throughout the execution of a kernel, offers faster access times.

Dimension of Thread Block size also affect execution time.Here 768 thread size is giving most optimal answer.It depends on type of application and also depends on configuration of GPU.

References

[1] https://cs.brown.edu/courses/cs195v/lecture/week10.pdf
[2] J https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/
[3] https://selkie.macalester.edu/csinparallel/modules/CUDAArchitecture/build/html/2-Findings/Findings.html
[4] https://www.bu.edu/pasi/files/2011/07/Lecture2.pdf
[5] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlarchitecture-8-x
[6] https://cuda-programming.blogspot.com/2013/01/what-is-constant-memory-in-cuda.html
[7] https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html