Introduction to Parallel and Distributed Computation Project Report

Jiaping Sun
1500017735
Yuanpei College, Peking University

Qianli Shen
1500017740
Yuanpei College, Peking University

1 Introduction

1.1 Tansposed Convolution

Transposed convolution, which is also called fractionally strided convolution, works by swapping the forward and backward passes of a convolution. For instance, the kernel w defines a convolution whose forward and backward passes are computed by multiplying with C and C^T respectively, but it also defines a transposed convolution whose forward and backward passes are computed by multiplying with C^T and $(C^T)^T = C$ respectively. The transposed convolution operation can be thought of as the gradient of some convolution with respect to its input, which is usually how transposed convolutions are implemented in practice.

Direct and transposed convolution is always an important method in image processing and computer vision. Convolution neural network (CNN), an image feature extracting method of the state of art, asks for huge amount of convolution and gradient computation. In this way, efficient algorithms and methods designed for transposed convolution appear to be significant.

In this project we implement transposed convolution by serial method and some parallel methods and measure their performance.

1.2 K-Nearest Neighbors

K-Nearest Neighbors is a simple but classic method in data mining, especially in classification problems. The purpose is to find the top k-nearest points of each point.

In practice, the main bottleneck of this method is its huge amount of computation, since the dimension and the number of points can be quite large. Efficient algorithms were designed to compute KNN, such as KD-tree, which constructs a high dimensional binary-search-tree-like data structure. While in this project, we implement direct and simple parallel algorithms to compute KNN using OpenCL and measure its performance.

2 Implementation

2.1 Transposed Convolution

In this project we regard transposed convolution as direct convolution computation with mapping, which is inefficient in practice due to many zero computation. We can easily derive a nested loops algorithm. Here we require direct convolution strides s as parameter. The transposed convolution strides are in fact $1/s$. Time cost of the algorithm is $O(WHCwh)$.

For this kind of task with nested loops, the key point of the parallel algorithm with GPU is how to distribute global and local nodes. In this project, we try three different ways to distribute tasks and respectively measure their performance in the following part. All three versions maintain the same global group size of (W, H). The local group sizes are (1), (C), (C, w, h) respectively.

Algorithm 1 Transposed Convolution Computation

```
Require: $input$ with shape $(W, H, C)$, $kernel$ with shape $(w, h)$, strides $s$
Ensure: single channel $output$ with shape $(Ws, Hs)$
  for $x = 0, 1, ..., W \cdot s - 1$ do
    for $y = 0, 1, ..., H \cdot s - 1$ do
      for $channel = 0, 1, ..., C$ do
        $output[x][y] = 0$
        for $i = 0, 1, ..., w - 1$ do
          for $j = 0, 1, ..., h - 1$ do
            $w = kernel[i][j]$
            $p = mapping(x, y, channel, i, j)$
            $output[x][y] = ouput[x][y] + w \cdot p$
          end for
        end for
      end for
    end for
  end for
  return
```

2.2 K-Nearest Neighbors

The most simple and intuitive algorithm is to divide the problem into two phases, first calculating the distance between every single two points, then select the smallest k within each points' neighbors. Therefore we use two OpenCL kernels to solve the problem.

For the first kernel, where we declare $n * n * dim$ work items, n denotes the number of points, and dim denotes the number of dimensions. Each work group calculates the distance between two points. It contains dim work items, and a single work item calculate the distance in one dimension, which is reduced to sum in the end.

The second phase is to select least K of each point. A naive way is to sort the whole array and then select the first K elements. But since we only need top K but not the whole ranks, sorting can be time-wasting. Therefore, we can simply go through the array K times, picking the smallest one every iteration. However, that's still $O(n^2)$ when n is large.

As an optimization, we use a heap structure, building a max-heap of size K with first K elements of array initially, and then iterate through the array once, replacing the top element if and only if it is smaller than the top element. The time complexity of phase 2 is totally $O(nlogK)$. This kernel requires n work item, the i^{th} work item build its own heap and select the K-smallest of the i^{th} point. If possible, we can parallelize this heaping-process as well.

Another trivial optimization is to reduce the "distance" matrix to triangular matrix, since the distance is symmetric. It frees half of the space and saves half of the computation.

3 Experiment

We do all our experiments on the appointed server with Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz and NVIDIA Corporation GM200 [GeForce GTX TITAN X]. All of our experient results are averaged ones.

3.1 Transposed Convolution

This part is mainly accomplished by Qianli Shen.

We run serial program and parallel program of all three versions on 4 tasks of different size. The size of tasks are shown in Table 1. Parameters are randomly generated according to the task size.

As we have mentioned above, what makes a difference among three versions of parallel program is the local item size. In v0, we do not distribute local items and in every global node the program do jobs of 3-nested loops. In v1, local item size is C and the depth of loop in local nodes reduces to 2. Finally in v2, local item size is C, w, h and in every node program only do a simple product calculation.

Table 2 shows total execution time of programs. When task size is small, although the kernel excution time cost is far smaller than serial program run time, parallel methods behave not that good in total execution time due to the cost for setting and communication. But when it comes to the large-scale tasks, parallel programs gain considerible speedup. And the larger the task size is, the better parallel programs behave.

Table 3 shows different performance of three versions of parallel methods. We can see v1 and v2 behave better than v0 because of better-designed task distribution. However, it is interesting to see that v2 has obvious better performance than v1 in tasks with small-scale filter but. But in large-scale filter tasks v1 and v0 have almost same execution time. It means that tasks distribution is not always the finer the better due to the limitation of device.

Table 1: Task Parameters

	task1	task2	task3	task4
W	200	200	1920	1920
H	200	200	1080	1080
C	3	32	3	32
w	3	5	3	5
h	3	5	3	5
s	2	2	2	2

Table 2: Total Execution Time(ms)

	task1	task2	task3	task4
serial	110	1.75K	3.2K	92.3K
v0	180	250	460	3.3K
v1	170	200	400	1.4K
v2	160	180	210	1.4K

Table 3: Kernel Execution Time(ms)

	task1	task2	task3	task4
v0	5.0	62.6	262	3.0K
v1	5.0	22.1	228	1.2K
v2	0.79	22.1	35.1	1.2K

3.2 K-Nearest Neighbors

This part is mainly accomplished by Jiaping Sun.

We compared serial program and two parallel algorithms mentioned above: the first is naive k-iterations, and the second is optimized heap-select. We mainly focus on the total executing time (excluding I/O time) as well as kernel executing time.

Table 4: Execution Time (n=10000, dim=120, K=18)

time/ms	total	kernel-1	kernel-2
serial	31620	-	-
k-iter	3030	865.042	179.558
heap-select	2480	565.944	60.960

Table 5: Execution Time (n=10000, dim=120, K=1000)

time/ms	total	kernel-1	kernel-2
serial	292K	-	-
k-iter	13.5K	870.6	107K
heap-select	2.73K	564.418	306.97

Table 6: Execution Time (n=100, dim=10, K=10)

time/ms	total	kernel-1	kernel-2
serial	0.0001	-	-
k-iter	1820	0.048	0.081
heap-select	1830	0.044	0.072

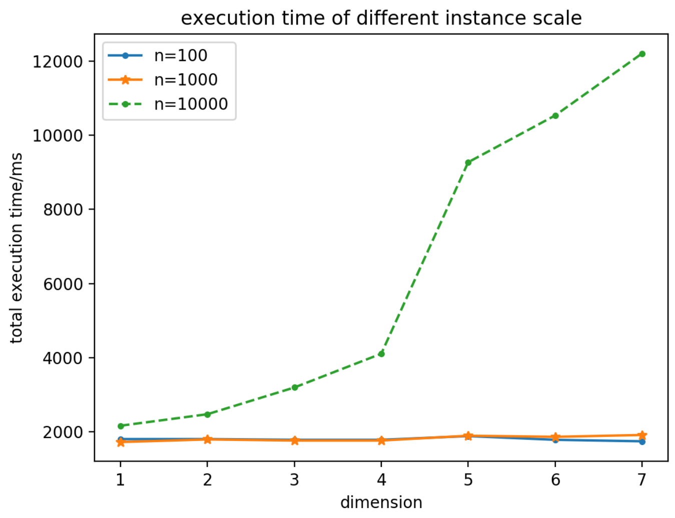

Figure 1: Execution time of different instance scale

Table 4 is the result of an instance which has 10000 points, 120 dimensions with K = 18. As we can see, the speedup is more than 10. However, the overhead time is far greater than the actual computing time (kernels) in both parallel program, including complex initialization of OpenCL program, such as requiring platforms and devices, creating kernels and programs. And the second algorithm performs better than the naive one in both two stages.

Table 5 suggests that when K gets larger, k-iter program gets slower, since its second kernel is inefficient, serial program is worse as well while heap algorithm remains fine.

When instance scale is small, for example, n = 100, dim = 10, k = 10, serial program performs far better than OpenCL because of the overhead, as Table 6 suggests.

As for a closer look at scalability, Figure 1 shows the total execution time related to different n and dim with heap-select algorithm. When n is small, overhead dominates the total time, thus it does not change with parameter dim. With greater n, execution time increases as dimension grows.

4 Summary

In this project we try to design parallel algorithms for two classical problem, transposed convolution and K-Nearest Neighbors(KNN). We run our programs with GPU and gain obviously better performance than serial ones.

However, there are still many complex but better serial algorithms on which designing parallel algorithm is much more difficult. We may have a try in the future.

5 Reference

[1]http://deeplearning.net/software/theano_versions/dev/tutorial/conv_arithmetic.html#transposed-convolution

[2] Li, S., & Amenta, N. (2015, October). Brute-force k-nearest neighbors search on the GPU. In International Conference on Similarity Search and Applications (pp. 259-270). Springer, Cham.

[3] Liang, S., Liu, Y., Wang, C., & Jian, L. (2009, October). A CUDA-based parallel implementation of K-nearest neighbor algorithm. In Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC'09. International Conference on (pp. 291-296). IEEE.