

MobileNet V2 并行加速

Group 19

目录

1	模型介绍	2
2	模型实现	2
3	优化过程	3
3.1	版本一：基本实现	3
3.2	版本二：大幅优化	5
3.2.1	版本 2.1：卷积核函数优化	5
3.2.2	版本 2.2：DtoD 优化	6
3.3	版本三：小幅优化	7
3.3.1	版本 3.1：cudaMalloc 优化	7
3.3.2	版本 3.2：BlockSize 优化	7
3.3.3	版本 3.3：1x1 卷积核优化	8
4	Baseline 对比结果	8
4.1	cudnn 实现	8
4.2	cudnn 运行结果	10
5	总结与展望	10
5.1	项目总结	10
5.2	未来改进方向	10
5.3	致谢	11

1 模型介绍

在本项目中，我们使用 cuda 实现了经典的图卷积模型 MobileNetV2。MobileNet 是一个轻量级的图像识别模型，相较于其他模型如 CNN，resnet 等，参数量显著下降，推理速度加快，但是依旧在目标识别、细粒度分类等任务上有良好的表现。

模型特点 MobileNet 采用 Depthwise Separable Convolution（图 1）的方式代替传统的卷积。Depthwise Separable Convolution 可以被分解为 Depthwise Convolution 和 Pointwise Convolution。Depthwise Convolution 对不同的输入通道使用不同的卷积核，实现了深度层面的卷积；Pointwise Convolution 使用 1×1 的卷积核将多个输入通道合并。通过这两种卷积方式的结合，完成与传统卷积方式基本等效的卷积操作，同时降低了参数量。

实现思路 MobileNet 主要包括了卷积，矩阵求和，矩阵乘法和平均值池化几个操作，可以分别实现这些操作，然后按照网络的拓扑结构组织网络的推理。

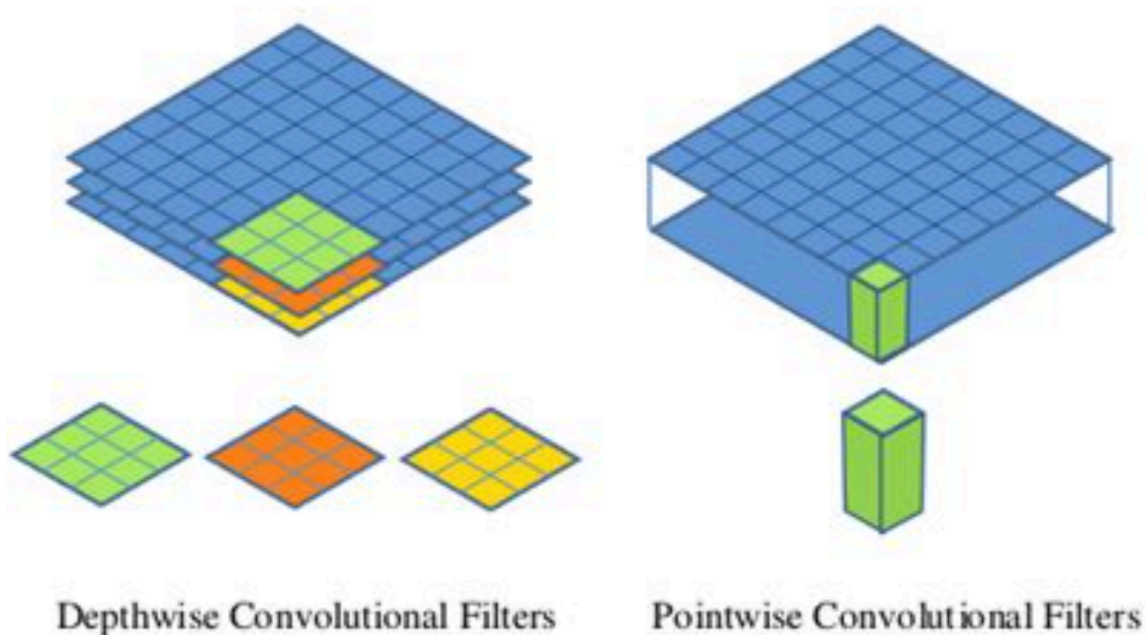


图 1: Mobilnet 卷积结构：Depthwise Separable Convolution

2 模型实现

代码结构 模型实现的代码结构如下，可以看到主要操作被我们分解为了多个的 cuda 文件分别实现，main.cc 中调用这些操作的接口实现网络的拓扑结构

```

1 # 由于有多个相似目录，我们用通配符记录了文件名
2 +-- *_version/(模型实现代码，final_version下为最终实现)
3     +-- add_*.cu (矩阵相加)
4     +-- add_*.cuh
5     +-- conv_*.cu (卷积)
6     +-- conv_*.cuh
7     +-- gemm_*.cu (矩阵相乘)
8     +-- gemm_*.cuh
9     +-- global_avg_*.cu (平均池化)
10    +-- global_avg_*.cuh
11    +-- main_*.cc (主函数，包含模型的拓扑结构实现)
12    +-- Makefile
13    +-- mobilenetInput.txt (输入)
14    +-- mobilenetOutput.txt (参考输出)
15    +-- params.txt (模型参数)

```

整体架构 首先调用 ‘initModel’ 函数，载入模型参数，同时在 GPU 上为模型参数开辟空间，并将参数复制到 GPU；然后调用 ‘inference’ 函数进行模型推理，将输入的图片依次通过模型各层进行操作，最终获得输出结果。通过多次多图 inference 测试，我们将输出结果与参考结果进行对比，保证误差小于 10^{-5} ，同时我们会测出单次 inference 的平均时间，以评估性能。

3 优化过程

3.1 版本一：基本实现

实现方法 卷积操作接受输入、卷积核等指针（期望是在 host memory 中），先拷贝到 GPU 的 global memory，再进行 padding，最后启用 kernel 函数开始并行化计算，为每个输出像素点开一个 thread。最终算完后再将其拷贝到 host memory 中，完成操作。由于一个 grid 能开的 thread 是有限的，我们设置了常量 BLOCKSIZE 作为 grid 中 thread 的 dimension。

核心代码 卷积核函数的实现如下

```

1 __global__ void convKernel(double *input, double *filter, double *
    output, double* bias, int filter_width, int filter_num, int
    input_width, int input_depth, int stride, int out_width, bool clip)
    {
2     int output_row = blockIdx.x*blockDim.x+threadIdx.x;

```

```

3      int output_col = blockIdx.y*blockDim.y+threadIdx.y;
4      int input_row = output_row*stride;
5      int input_col = output_col*stride;
6      if(output_row >= out_width || output_col >= out_width) return;
7
8      int fnum = blockIdx.z;
9      double tmp = 0.0;
10     for(int d=0;d<input_depth;d++)
11     for(int r=0;r<filter_width;r++)
12     for(int c=0;c<filter_width;c++)
13         tmp += input[d*input_width*input_width + (input_row+r)*
14                 input_width + input_col+c]*filter[fnum*input_depth*
15                 filter_width*filter_width + d*filter_width*filter_width
16                 + r*filter_width + c];
17
18     tmp += bias[fnum];
19     if(clip){
20         if(tmp < 0) tmp = 0.0;
21         else if(tmp > 6) tmp = 6.0;
22     }
23     output[fnum*out_width*out_width + output_row*out_width+
24           output_col] = tmp;
25 }

```

性能分析 最终运行时间见图 2，根据 GPU activities 见图 3的情况可以看出，convKernel 整体时间占用最多，为主要的性能瓶颈。这是因为在 convKernel 的实现中，包含了四重循环，并行程度并不理想，还可以进一步并行。

```

group19@acalab-w760-g30:~/mobilenet$ ./a.out
Average Time is: 2278.529297

```

图 2: 版本 1: 单次运行速度

	Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:		74.27%	983.15ms	36	27.310ms	348.70us	280.71ms	convKernel(double*
		25.00%	330.87ms	259926	1.2720us	1.0560us	2.9120us	[CUDA memcpy DtoD]
		0.37%	4.9399ms	17	290.58us	61.440us	937.21us	convGroupKernel(do
		0.34%	4.5002ms	107	42.058us	1.2160us	2.2253ms	[CUDA memcpy HtoD]
		0.01%	179.26us	53	3.3820us	1.6640us	17.631us	[CUDA memset]
		0.00%	29.438us	10	2.9430us	1.7280us	6.7200us	AddKernel(int, dou
		0.00%	5.5360us	1	5.5360us	5.5360us	5.5360us	GlobalAvgKernel(do
		0.00%	2.1760us	1	2.1760us	2.1760us	2.1760us	[CUDA memcpy DtoH]
API calls:		63.52%	2.27902s	260034	8.7640us	6.0890us	2.4431ms	cudaMemcpy
		27.70%	993.92ms	118	8.4231ms	3.9990us	280.72ms	cudaFree
		8.70%	312.32ms	224	1.3943ms	3.8380us	300.20ms	cudaMalloc
		0.03%	943.03us	1	943.03us	943.03us	943.03us	cuDeviceTotalMem

图 3: 版本 1: GPU Activity 分析

3.2 版本二：大幅优化

3.2.1 版本 2.1：卷积核函数优化

实现方法 采用更细粒度的并行。从原本的每个 thread (convKernel) 计算一个像素点 (包含多个 channel)，细粒度化为每个 thread (convKernel) 只计算一个像素点中的一个 channel。

核心代码 卷积核函数的实现如下

```

1  __global__ void convKernel(double *input,double *filter,double *
    output, double* bias, int filter_width,int filter_num,int
    input_width,int input_depth,int stride,int out_width, bool clip)
    {
2      int output_row = blockIdx.x*blockDim.x+threadIdx.x;
3      int output_col = blockIdx.y*blockDim.y+threadIdx.y;
4      int input_row = output_row*stride;
5      int input_col = output_col*stride;
6      if(output_row >= out_width || output_col >= out_width) return;
7
8      for(int fnum=0;fnum<filter_num;fnum++){
9          double tmp = 0.0;
10         for(int d=0;d<input_depth;d++)
11             for(int r=0;r<filter_width;r++)
12                 for(int c=0;c<filter_width;c++)
13                     tmp += input[d*input_width*input_width + (input_row+r)*
                        input_width + input_col+c]*filter[fnum*input_depth*
                        filter_width*filter_width + d*filter_width*
                        filter_width + r*filter_width + c];

```

```

14
15     tmp += bias[fnum];
16     if(clip){
17         if(tmp < 0) tmp = 0.0;
18         else if(tmp > 6) tmp = 6.0;
19     }
20     output[fnum*out_width*out_width + output_row*out_width+
21           output_col] = tmp;
22 }

```

性能分析 最终运行时间见图 4，根据 GPU activities 见图 5，增加 grid 的维度，线程总数正比于 channel，大大降低了 convKernel 的时间。CUDA memcpy DtoD 变成了几乎唯一的瓶颈。经过代码分析，我们发现 DtoD 的数据搬运主要来源于 padding。在我们的实现中，是额外开一块空间作为 padded image 的目的地，将原有数据 memcpy 进去，这个操作时非常耗时的。

```

group19@acalab-W760-G30:~/mobilenet$ ./a.out
Average Time is: 1312.240967

```

图 4: 版本 2.1: 单次运行速度

3.2.2 版本 2.2: DtoD 优化

实现方法 我们发现通过判断省去 padding 为 0 的拷贝之后，总体时间减少了一半，不过 DtoD 的内存搬运带来的时间消耗还是非常严重。为此我们采取了更加激进的办法：在上一层

	Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:		94.84%	342.53ms	259926	1.3170us	1.2150us	9.1200us	[CUDA memcpy DtoD]
		2.45%	8.8524ms	36	245.90us	61.759us	1.5655ms	convKernel(double*, d
		1.39%	5.0055ms	17	294.44us	68.479us	937.49us	convGroupKernel(doubl
		1.26%	4.5664ms	107	42.676us	1.1840us	2.2535ms	[CUDA memcpy HtoD]
		0.05%	179.87us	53	3.3930us	1.7280us	17.824us	[CUDA memset]
		0.01%	30.047us	10	3.0040us	1.7910us	7.2000us	AddKernel(int, double
		0.00%	6.0480us	1	6.0480us	6.0480us	6.0480us	GlobalAvgKernel(doubl
		0.00%	2.7200us	1	2.7200us	2.7200us	2.7200us	[CUDA memcpy DtoH]
API calls:		87.66%	2.26227s	260034	8.6990us	6.0490us	2.4750ms	cudaMemcpy
		11.48%	296.28ms	224	1.3227ms	3.0300us	282.70ms	cudaMalloc
		0.75%	19.308ms	118	163.63us	3.4890us	1.5752ms	cudaFree

图 5: 版本 2.1: GPU Activity 分析

conv 产生结果的同时进行本层 conv 的 padding。这样将本次的输出直接作为下一次卷积的输入，直接避免了 DtoD 的内存搬运。

性能分析 最终运行时间见图 6，根据 GPU activities 见图 7，目前 cudaMalloc 成为了新的瓶颈

```
group19@acalab-W760-G30:~/mobilenet$ ./mobilenet
Average Time is: 27.509665
```

图 6: 版本 2.2: 单次运行速度

3.3 版本三：小幅优化

3.3.1 版本 3.1: cudaMalloc 优化

实现方法 之前在每次卷积运算中，运算结果都会放在新申请的内存空间中，导致多次重复的 malloc 操作。可以将这部分空间在模型初始化时分配，后续只需要读写。

性能分析 最终运行时间见图 8，根据 GPU activities 见图 9，malloc 花费的时间显著下降，压力回到了卷积操作上。

3.3.2 版本 3.2: BlockSize 优化

实现方法 调整卷积操作中的 BLOCKSIZE，提升并行程度

性能分析 最终运行时间见图 10，根据 GPU activities 见图 11，convKernel 的时间变为原来的 1/2 左右，但是仍然是主要的性能瓶颈。

```
group19@acalab-W760-G30:~/mobilenet$ nvprof ./mobilenet
==21378== NVPROF is profiling process 21378, command: ./mobilenet
Average Time is: 28.858368
==21378== Profiling application: ./mobilenet
==21378== Profiling result:
Type      Time(%)      Time      Calls      Avg      Min      Max      Name
GPU activities: 42.23%  7.5993ms    36  211.09us  49.663us  1.1657ms convKernel(double*, double*, double*, double*, int, int, int, int, int, int, int, bool)
                31.64%  5.6933ms   838  6.7930us  1.1200us  2.5371ms [CUDA memcpy HtoD]
                24.91%  4.4816ms   17  263.62us  61.791us  838.81us convGroupKernel(double*, double*, double*, double*, int, int, int, int, int, int, bool)
                0.93%  167.13us   54  3.0950us  1.6640us  15.487us [CUDA memset]
                0.15%  26.496us   10  2.6490us  1.6320us  6.4640us AddKernel(int, double*, double*)
                0.10%  18.016us   10  1.8010us  1.2480us  5.0240us [CUDA memcpy DtoD]
                0.03%  5.5040us    1  5.5040us  5.5040us  5.5040us GlobalAvgKernel(double*, double*, int, int)
                0.01%  2.4960us    1  2.4960us  2.4960us  2.4960us [CUDA memcpy DtoH]
API calls:      90.83%  347.13ms   171  2.0300ms  3.0750us  315.57ms cudaMalloc
                4.57%  17.464ms   849  20.560us  6.1500us  2.7688ms cudaMemcpy
                3.83%  14.628ms   65  225.04us  6.6430us  1.1745ms cudaFree
                0.25%  956.53us    1  956.53us  956.53us  956.53us cuDeviceTotalMem
                0.19%  743.62us   54  13.770us  8.6430us  78.107us cudaMemcpy
                0.16%  623.11us   64  9.7360us  7.0890us  47.510us cudaLaunchKernel
                0.14%  535.85us   97  5.5240us      223ns  295.48us cuDeviceGetAttribute
                0.01%  34.032us    1  34.032us  34.032us  34.032us cuDeviceGetName
                0.01%  22.203us    2  11.101us  1.2360us  20.967us cudaEventCreate
                0.00%  14.647us    2  7.3230us  4.4660us  10.181us cudaEventRecord
```

图 7: 版本 2.2: GPU Activity 分析


```
Average Time is: 21.626221
```

图 8: 版本 3.1: 单次运行速度

	Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:		57.46%	774.44ms	3600	215.12us	52.511us	1.5711ms	convKernel(dou
		33.62%	453.06ms	1700	266.50us	61.151us	938.10us	convGroupKerne
		7.61%	102.56ms	73306	1.3990us	1.1840us	2.2548ms	[CUDA memcpy H
		0.96%	12.998ms	5400	2.4070us	1.6320us	14.240us	[CUDA memset]
		0.18%	2.4765ms	1000	2.4760us	1.5680us	6.6560us	AddKernel(int,
		0.11%	1.5382ms	1000	1.5380us	1.2480us	2.9760us	[CUDA memcpy D
		0.03%	455.45us	100	4.5540us	4.4790us	6.0790us	GlobalAvgKerne
		0.01%	194.21us	100	1.9420us	1.8240us	3.1680us	[CUDA memcpy D
	API calls:	45.62%	1.13526s	1100	1.0321ms	55.369us	35.542ms	cudaFree
		36.34%	904.19ms	74406	12.152us	6.0350us	62.473ms	cudaMemcpy
		14.28%	355.27ms	1360	261.23us	2.4220us	296.59ms	cudaMalloc
		1.88%	46.689ms	5400	8.6460us	7.3970us	71.241us	cudaMemset
		1.74%	43.384ms	6400	6.7780us	5.8840us	721.27us	cudaLaunchKern

图 9: 版本 3.1: GPU Activity 分析

3.3.3 版本 3.3: 1x1 卷积核优化

实现方法 通过对 MobileNet 的观察我们发现，大部分的卷积操作都是 1x1 的卷积核，且步长为 1，对于 1x1 的卷积核，没必要与 3*3 的卷积核一样进行多重循环来求卷积值，所以我们单独实现了 1x1 的卷积函数

性能分析 最终运行时间见图 12，根据 GPU activities 见图 13，convKernel 的时间变为原来的 1/2 左右，但是仍然是主要的性能瓶颈。

4 Baseline 对比结果

4.1 cudnn 实现

cuDnn 的卷积操作通过 cudnnConvolutionForward 函数进行，除了输入与输出之外，该函数还需要设定输入、输出、卷积操作等多种描述符，并自动计算出所需的算法和需要的缓存空间。加法和 clip 操作也需要同样的描述符，这些描述符对每次卷积都是特定的，需要每次创建和销毁。

```
group19@acalab-W760-G30:~/mobilenet$ ./a.out
Average Time is: 10.599330
```

图 10: 版本 3.2: 单次运行速度


```
group19@acalab-W760-G30:~/mobilenet$ nvprof ./a.out
==12879== NVPROF is profiling process 12879, command: ./a.out
Average Time is: 13.821652
==12879== Profiling application: ./a.out
==12879== Profiling result:
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	74.75%	486.63ms	3600	112.95us	35.551us	336.80us	convKernel(double
	18.67%	181.57ms	73306	1.3850us	1.1280us	2.2111ms	[CUDA memcpy HtoD]
	3.32%	18.083ms	1700	10.637us	4.1280us	36.544us	convGroupKernel(d
	2.40%	13.070ms	5400	2.4280us	1.6000us	14.432us	[CUDA memset]
	0.45%	2.4729ms	1800	2.4720us	1.6000us	6.0480us	AddKernel(int, do
	0.28%	1.5085ms	1800	1.5080us	1.2480us	2.6240us	[CUDA memcpy DtoD]
	0.06%	450.52us	100	4.5050us	4.4480us	5.5360us	GlobalAvgKernel(d
	0.03%	174.66us	100	1.7460us	1.6640us	2.5280us	[CUDA memcpy DtoH]
API calls:	53.04%	890.28ms	74406	11.965us	5.9620us	60.457ms	cudaMemcpy
	21.16%	355.19ms	1100	322.90us	11.615us	681.02us	cudaFree
	20.27%	340.29ms	1360	250.21us	2.4240us	294.39ms	cudaMalloc
	2.75%	46.205ms	5400	8.5560us	7.3190us	85.606us	cudaMemset
	2.57%	43.109ms	6400	6.7350us	5.7750us	694.72us	cudaLaunchKernel
	0.05%	839.10us	1	839.10us	839.10us	839.10us	cuDeviceTotalMem
	0.04%	680.51us	100	6.8050us	5.8660us	14.641us	cudaEventSynchron

图 11: 版本 3.2: GPU Activity 分析

```
group19@acalab-W760-G30:~/mobilenet$ ./mobilenet
Average Time is: 7.776836
```

图 12: 版本 3.3: 单次运行速度

```
==1064== Profiling result:
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	54.89%	9.43582s	170000	55.504us	20.320us	150.75us	conv1Kernel(double*, double*, d
	28.23%	4.85252s	3660106	1.3250us	1.1840us	2.2128ms	[CUDA memcpy HtoD]
	5.30%	911.84ms	5000	182.37us	178.08us	202.53us	matMulKernel(int, int, int
	5.28%	907.22ms	85000	10.673us	4.0950us	38.048us	convGroupKernel(double*, double
	3.70%	636.39ms	265000	2.4010us	1.6000us	14.784us	[CUDA memset]
	1.69%	290.16ms	5000	58.031us	57.183us	66.208us	convKernel(double*, double*, do
	0.72%	124.33ms	50000	2.4860us	1.6000us	6.6560us	AddKernel(int, double*, double*
	0.13%	22.734ms	5000	4.5460us	4.5110us	6.0800us	GlobalAvgKernel(double*, double
	0.05%	8.1513ms	5000	1.6300us	1.4080us	2.7840us	[CUDA memcpy DtoH]
API calls:	76.68%	44.7469s	3665106	12.208us	6.1470us	283.24ms	cudaMemcpy
	12.81%	7.47604s	5000	1.4952ms	16.507us	33.292ms	cudaFree
	5.61%	3.27461s	265000	12.357us	7.3150us	260.38ms	cudaMemset
	3.79%	2.21043s	320000	6.9070us	5.9830us	694.59us	cudaLaunchKernel
	0.91%	533.38ms	10160	52.498us	5.0670us	279.99ms	cudaMalloc
	0.05%	31.853ms	10000	3.1850us	2.3700us	444.17us	cudaEventRecord
	0.05%	29.073ms	5000	5.8140us	2.5010us	24.343us	cudaEventSynchronize
	0.05%	26.962ms	10000	2.6960us	773ns	5.6488ms	cudaEventCreate
	0.03%	17.825ms	5000	3.5650us	3.1500us	20.217us	cudaDeviceSynchronize
	0.02%	10.689ms	5000	2.1370us	1.8730us	16.151us	cudaEventElapsedTime
	0.00%	1.0815ms	1	1.0815ms	1.0815ms	1.0815ms	cuDeviceTotalMem
	0.00%	224.56us	97	2.3150us	225ns	80.536us	cuDeviceGetAttribute
	0.00%	20.620us	1	20.620us	20.620us	20.620us	cuDeviceGetName
	0.00%	3.4910us	1	3.4910us	3.4910us	3.4910us	cuDeviceGetPCIBusId
	0.00%	2.6620us	3	887ns	361ns	1.9290us	cuDeviceGetCount
	0.00%	1.0960us	2	548ns	276ns	820ns	cuDeviceGet
	0.00%	502ns	1	502ns	502ns	502ns	cuDeviceGetUuid

图 13: 版本 3.3: GPU Activity 分析

```
group19@acalab-W760-G30:~/mobilenet/cudnn_baseline$ ./cudnn.out
Average Time is: 116.481952
```

图 14: cuDnn: 单次运行速度

```
0.92% 119.61ms 26000 4.6000us 2.4000us 37.375us void op_generic_tensor_kernel<int=3, double, double>
, reducedDivisorArray, int)
0.79% 102.66ms 608 168.85us 1.1520us 2.2307ms [CUDA memcpy HtoD]
0.58% 75.912ms 17500 4.3370us 1.9520us 37.152us void op_generic_tensor_kernel<int=1, double, double>
, reducedDivisorArray, int)
0.24% 30.818ms 552 55.830us 50.527us 70.847us void precomputed_convolve_dgemm<int=128, int=5, int=
, int, int, double const *, int, double *, double const *, kernel_conv_params, __int64, int, double, double, int, double const *
0.13% 17.298ms 5500 3.1450us 2.5280us 7.9040us void op_generic_tensor_kernel<int=1, double, double>
, reducedDivisorArray, int)
0.03% 4.5412ms 500 9.0820us 8.5440us 11.968us void cudnn::ops::pooling_fw_4d_kernel<double, double>
cudnnPoolingMode_t=2, bool=0>(cudnnTensorStruct, double const *, cudnn::ops::pooling_fw_4d_kernel<double, double, cudnn::averp
e_t=2, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, double, cudnnPoolingStruct, int, cudnn::reduced_divisor, double)
0.01% 1.2290ms 500 2.4580us 2.0160us 7.4240us [CUDA memcpy DtoH]
0.01% 1.0086ms 552 1.8270us 1.6960us 4.8000us void cudnn::cnn::kern_precompute_indices<bool=0>(i
0.00% 8.6400us 4 2.1600us 1.6960us 3.4880us [CUDA memset]
API calls: 58.21% 68.8645s 124000 555.36us 3.5570us 47.749ms cudaEventSynchronize
20.31% 24.0334s 89669 268.02us 684ns 3.27541s cudaMalloc
10.32% 12.2034s 89505 136.34us 458ns 38.294ms cudaFree
4.41% 5.22249s 285647 18.283us 7.8860us 11.955ms cudaLaunchKernel
1.94% 2.29910s 152 15.126ms 5.3800us 424.45ms cuModuleUnload
1.71% 2.01991s 423020 4.7740us 731ns 2.9946ms cudaEventRecord
1.03% 1.21566s 8 151.96ms 2.1270us 1.21565s cudaStreamCreateWithFlags
0.62% 731.25ms 124000 5.8970us 2.7370us 2.4674ms cudaEventElapsedTime
```

图 15: cuDnn: GPU Activity 分析

4.2 cudnn 运行结果

最终运行时间见图 14，尽管已经优化了 malloc 操作，但 cuDNN 的速度仍然不及 cuda 版本。根据 GPU activities 见图 15，接下来可能的优化方向是改善每次都需要重复创建的描述符，在每次卷积中重用。

5 总结与展望

5.1 项目总结

Mobile net 本身属于较小的网络，计算量不高。由于其本身绝大多数运算为卷积，本项目着重优化卷积的相关操作。主要思路包括：通过多个线程分别计算结果的一部分提高并行度，避免不必要的内存数据搬运，内存申请等操作尽量放到模型初始化阶段等。通过这些优化，我们实现了从最初的每次推理需要 2 秒到最终每次仅需 7.7 毫秒。

5.2 未来改进方向

当前版本的瓶颈 参考 Mobile net 结构特点，根据 nvprof 的结果（图 13），目前的瓶颈应当是 conv1Kernel 内的唯一一层循环。原因：随着网络深度加深，越靠后的网络输入的层数越多，循环迭代次数（串行）越多。

可能的解决办法 将 conv1Kernel 中的 for 循环也分配给不同的线程进行，最后使用通用的 reduce 优化策略将求和结果存到指定位置。不过这也会带来一定的弊端，中间结果存放需要较大的内存开销，同时计算单层卷积的访存次数会变多。

理论性能分析 假设并行化资源充足，原本需要 for 循环迭代 C 次，最后访存一次写入结果，那么采用上述方法后会变成并行 C 个线程执行一次乘法并写入一块连续内存，后 reduce（通常优化为并行的 $\log C$ 次迭代）。因此从时间轴的角度来看，记一次乘法需要 m 时间，一次内存写入需要 n 时间，一次加法需要 a 时间，则现有版本需要 $C \times m + n$ ；采取上述策略后，需要 $m + n + \log C \times a + n$ ，其中 $m + n$ 为分线程计算乘法并写入内存，后两项为 reduce 的复杂度。

5.3 致谢

感谢本学期老师与助教们的悉心指导。通过课堂学习、随堂作业以及本次项目作业，我们对并行化程序的设计思想与技巧有了从零到一的提升。在优化推理模型的流程中，我们发现了最初实现中的诸多问题，一一分析后逐渐改进，使得模型性能有了巨大的提升。虽然由于时间所限，更多的优化想法未能继续实现，但对问题的分析流程让我们受益良多。再次感谢老师与助教们的指导！