MobileNet V2 并行加速

Group 19

目录

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 1x1 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。

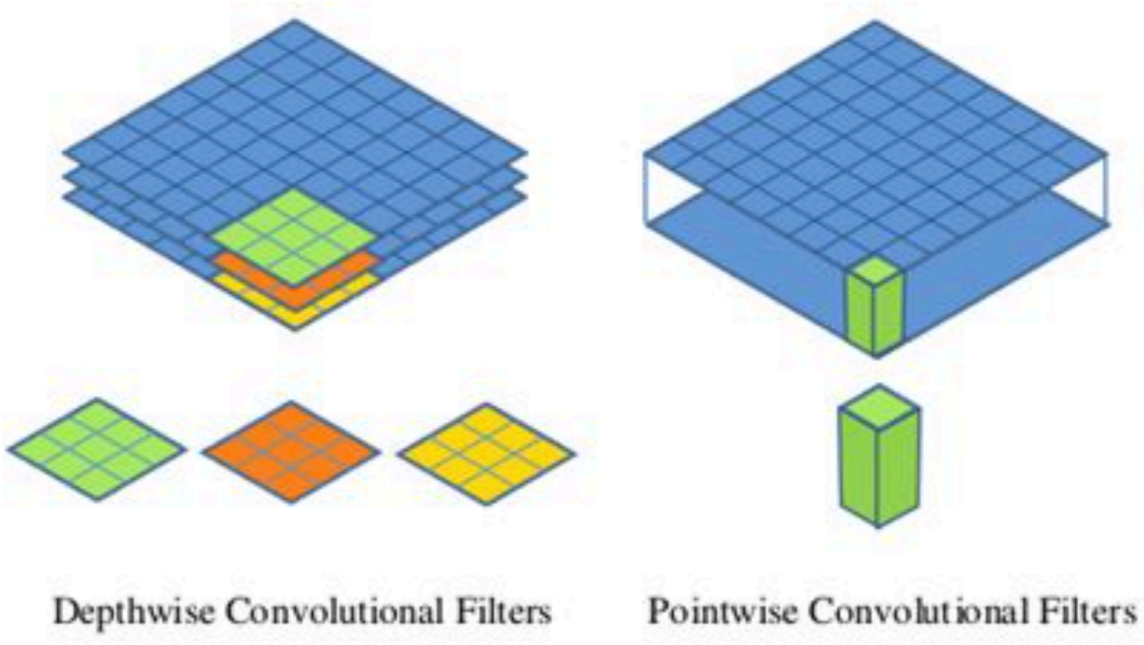


图 1: Mobilnet 卷积结构：Depthwise Separable Convolution

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

```
# 由于有多个相似目录，我们用通配符记录了文件名
+-- *_version/(模型实现代码，final_version下为最终实现)
    +-- add_*.cu （矩阵相加）
    +-- add_*.cuh
    +-- conv_*.cu （卷积）
    +-- conv_*.cuh
    +-- gemm_*.cu （矩阵相乘）
    +-- gemm_*.cuh
    +-- global_avg_*.cu （平均池化）
    +-- global_avg_*.cuh
    +-- main_*.cc （主函数，包含模型的拓扑结构实现）
    +-- Makefile
    +-- mobilenetInput.txt （输入）
    +-- mobilenetOutput.txt （参考输出）
    +-- 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

```
__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)
   {
    int output_row = blockIdx.x*blockDim.x+threadIdx.x;
```

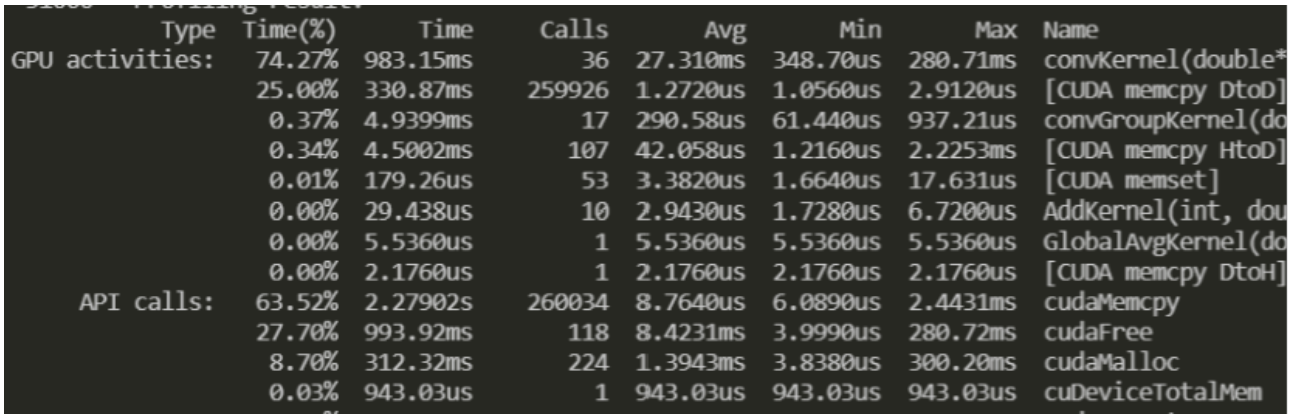
```
    int output_col = blockIdx.y*blockDim.y+threadIdx.y;
    int input_row = output_row*stride;
    int input_col = output_col*stride;
    if(output_row >= out_width || output_col >= out_width) return;

    int fnum = blockIdx.z;
    double tmp = 0.0;
    for(int d=0;d<input_depth;d++)
    for(int r=0;r<filter_width;r++)
    for(int c=0;c<filter_width;c++)
        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];

    tmp += bias[fnum];
    if(clip){
        if(tmp < 0) tmp = 0.0;
        else if(tmp > 6) tmp = 6.0;
    }
    output[fnum*out_width*out_width + output_row*out_width+
       output_col] = tmp;
}
```

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。

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

```
__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)
    {
     int output_row = blockIdx.x*blockDim.x+threadIdx.x;
     int output_col = blockIdx.y*blockDim.y+threadIdx.y;
     int input_row = output_row*stride;
     int input_col = output_col*stride;
     if(output_row >= out_width || output_col >= out_width) return;

     for(int fnum=0;fnum<filter_num;fnum++){
         double tmp = 0.0;
         for(int d=0;d<input_depth;d++)
         for(int r=0;r<filter_width;r++)
         for(int c=0;c<filter_width;c++)
             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+
               output_col] = tmp;
21      }
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

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。

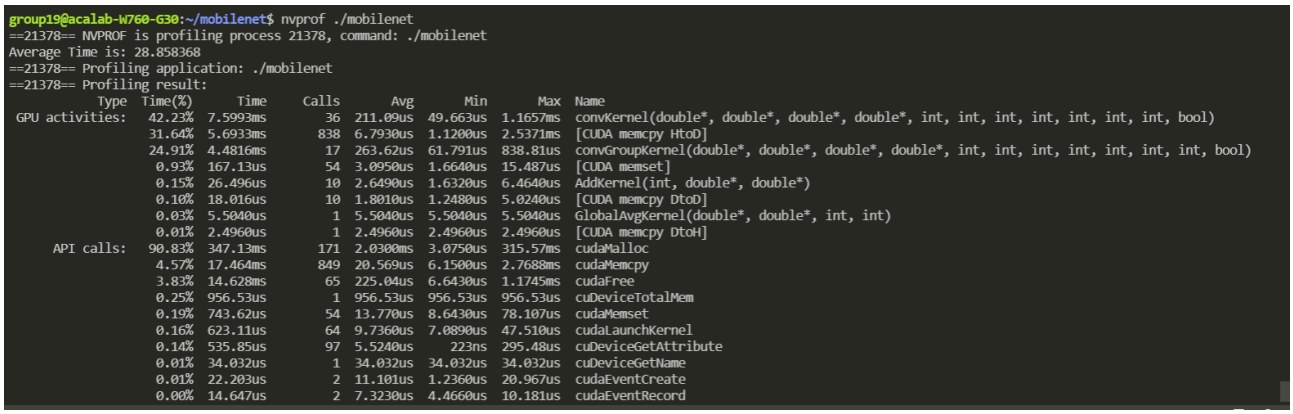


图 7: 版本 2.2：GPU Activity 分析

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。

图 10: 版本 3.2：单次运行速度

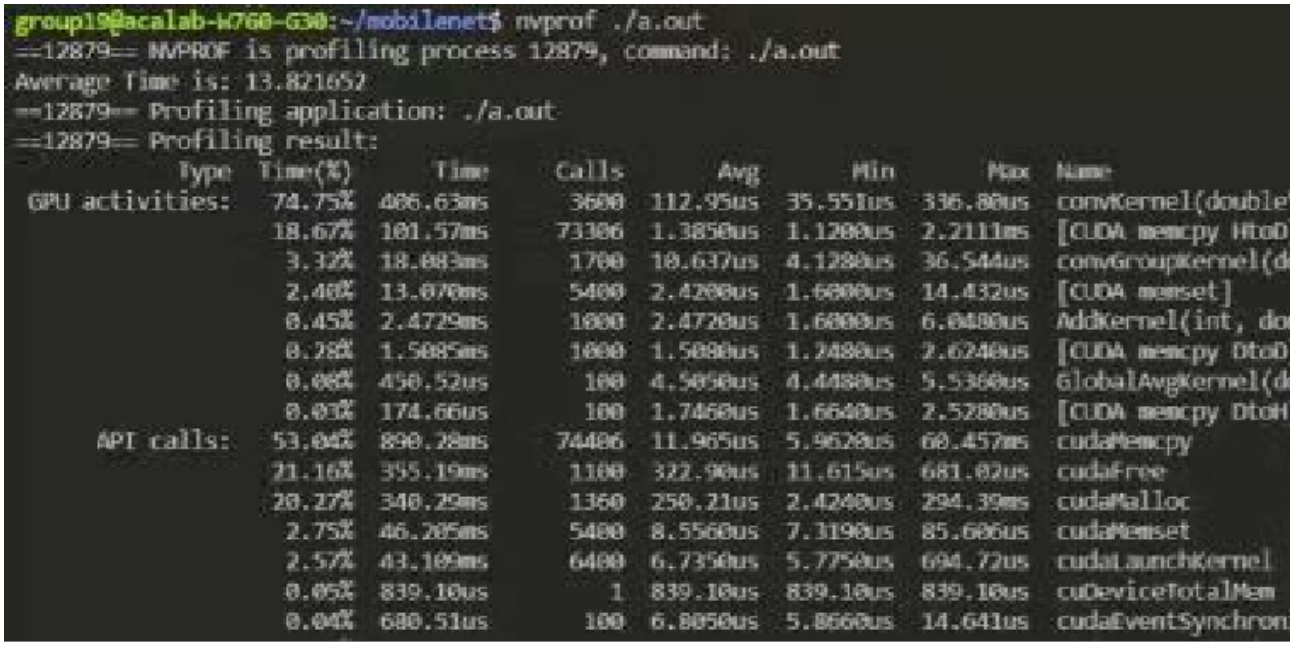


图 11: 版本 3.2：GPU Activity 分析

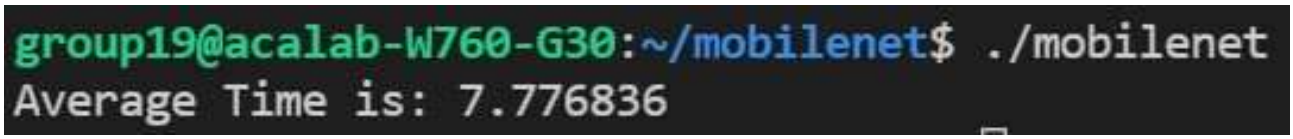


图 12: 版本 3.3：单次运行速度

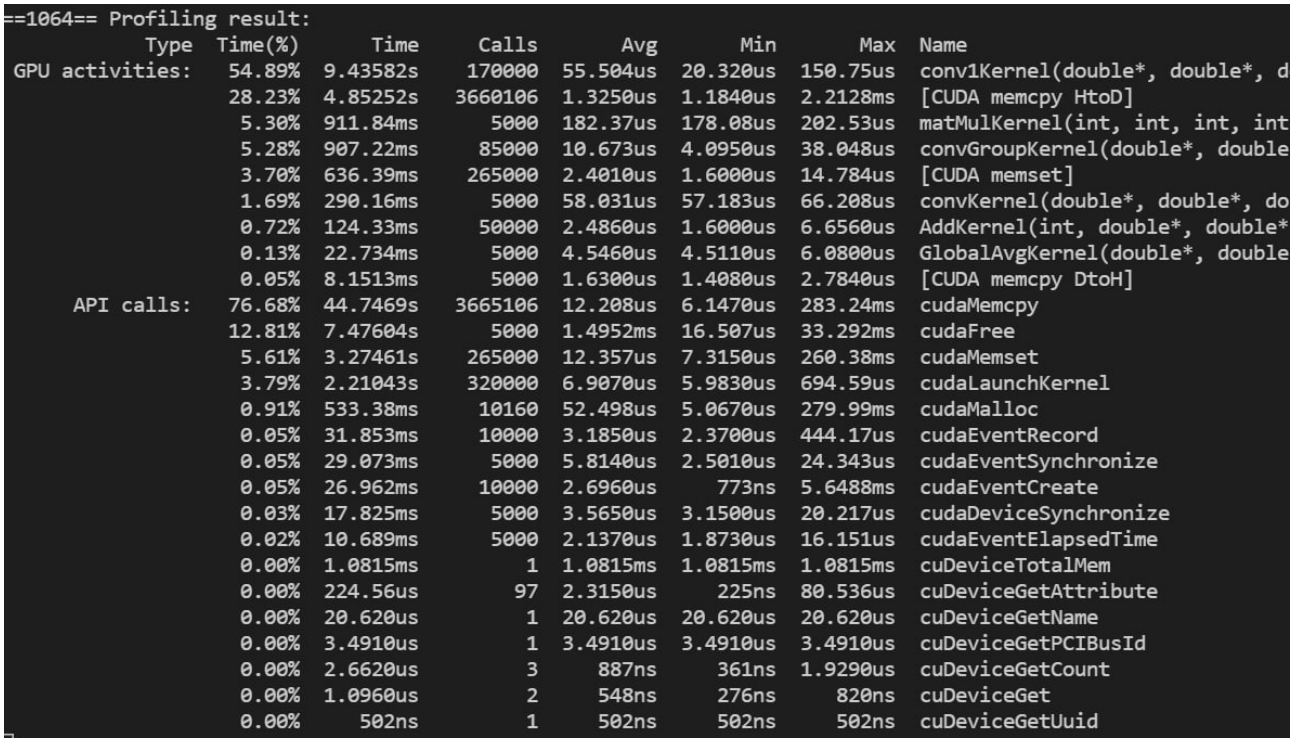


图 13: 版本 3.3：GPU Activity 分析

```
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, doubl
nNanPropagation_t=0, int=0>(cudnnTensorStruct, double*, cudnnTensorStruct, double const *, cudnnTensorStruct, double const *, d
, 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, doubl
nNanPropagation_t=1, int=1>(cudnnTensorStruct, double*, cudnnTensorStruct, double const *, cudnnTensorStruct, double const *, d
, reducedDivisorArray, int)
                    0.24%  30.818ms       552  55.830us  50.527us  70.847us  void precomputed_convolve_dgemm<int=128, int=5, in
, 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, doubl
nNanPropagation_t=0, int=0>(cudnnTensorStruct, double*, cudnnTensorStruct, double const *, cudnnTensorStruct, double const *, d
, reducedDivisorArray, int)
                    0.03%  4.5412ms       500  9.0820us  8.5440us  11.968us  void cudnn::ops::pooling_fw_4d_kernel<double, doub
 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！