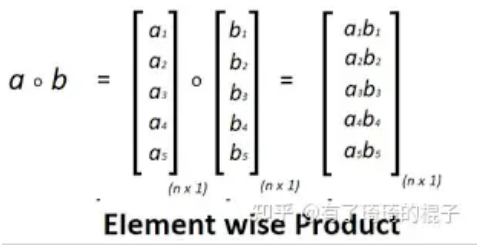
深入浅出GPU优化系列：elementwise优化及CUDA工具链介绍

本篇文章是深入浅出GPU优化系列的第3个专题，主要是介绍如何对GPU中的elementwise类OP进行优化。这其实是一个比较简单的话题，所以在说完kernel优化之后会再说点别的，主要是CUDA的二进制工具链，以及nsight工具的使用。后者在做profiling的时候会起到一个非常重要的作用。

总的来说，本篇文章分为三部分介绍，第一部分会介绍一下elementwise_add的优化，第二部分介绍CUDA的二进制工具，即cuobjdump和nvdisams的使用，第三部分主要是介绍nsight工具的使用，告诉大家怎么去做profiling。

一、elementwise优化

elementwise指的是需要逐位进行运行的op。举个例子，假设有数组a和数组b，数组a和数组b中的元素逐个进行乘法，得到其结果，示意图如下



elementwise

与reduce、sgemm等kernel的优化不一样，elementwise类的kernel其实并没有太多优化的技巧，只需要正常地设置block和thread，以及使用好向量化访存就可以获得接近于理论极限的性能。以elementwise_add为例。

不采用向量化访存的kernel如下：

```
__global__ void add(float* a, float* b, float* c)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    c[idx] = a[idx] + b[idx];
}
```

采用了float2进行访存的kernel如下：

```
#define FETCH_FLOAT2(pointer) (reinterpret_cast<float2*>(&(pointer))[0])
__global__ void vec2_add(float* a, float* b, float* c)
{
    int idx = (threadIdx.x + blockIdx.x * blockDim.x)*2;
    float2 reg_a = FETCH_FLOAT2(a[idx]);
    float2 reg_b = FETCH_FLOAT2(b[idx]);
    float2 reg_c;
    reg_c.x = reg_a.x + reg_b.x;
    reg_c.y = reg_a.y + reg_b.y;
```

```
    FETCH_FLOAT2(c[idx]) = reg_c;
}
```

采用了float4进行访存的kernel如下：

```
#define FETCH_FLOAT4(pointer) (reinterpret_cast<float4*>(&(pointer))[0])
__global__ void vec4_add(float* a, float* b, float* c)
{
    int idx = (threadIdx.x + blockIdx.x * blockDim.x)*4;
    float4 reg_a = FETCH_FLOAT4(a[idx]);
    float4 reg_b = FETCH_FLOAT4(b[idx]);
    float4 reg_c;
    reg_c.x = reg_a.x + reg_b.x;
    reg_c.y = reg_a.y + reg_b.y;
    reg_c.z = reg_a.z + reg_b.z;
    reg_c.w = reg_a.w + reg_b.w;
    FETCH_FLOAT4(c[idx]) = reg_c;
}
```

对3个kernel进行性能测试，其实验结果如下：

访存类型	带宽(GB/s)	带宽利用率
float	827	91.9%
float2	838	93.1%
float4	844	93.8%

结论：在V100上，global memory的带宽是900GB/s。可以从中看出，采用最简单的访存方式就已经能够达到827GB/s的带宽了，在使用float4进行向量化访存之后，性能提高了2%，到达844GB/s的带宽。当然，测试的case是一种比较简单的情况，对于一些更加复杂，取数有跳跃性的情况，从global取数还得考虑一下合并访存。

二、CUDA的二进制工具

为了让开发者可能窥探CUDA底层的实现细节，NV给出了两个主要的二进制工具，分别是cuobjdump和nvdisasm。官网链接如下

cuobjdump可以用来分析cubin文件和host文件。而nvdisasm只能用来分析cubin文件，但是可以得到更多的输出信息。我用的比较多的是nvdisasm。用来看代码的控制流图。

具体的命令如下：

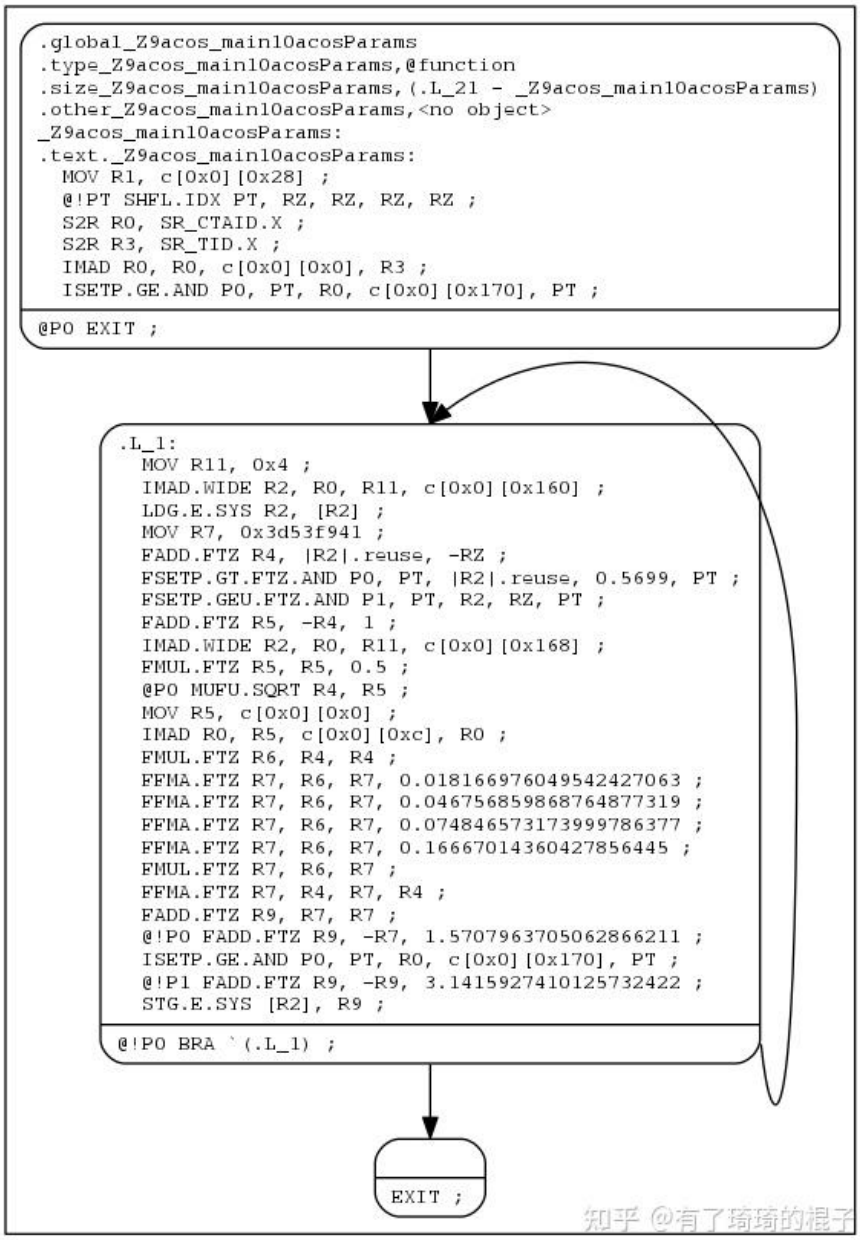
先得到cubin文件，这条命令会把所有的中间文件保留下来

```
    nvcc -o cudatest cudatest.cu -gencode=arch=compute_75,code=\"sm_75,compute_75\" --keep
```

然后用nvdisasm进行分析

```
nvdisasm -bbcfg a.cubin
```

将屏幕输出的信息放置在一个.dot文件中，而后用Vscode安装相关插件，直接打开。效果如下



nvdisasm生成的Control Flow Graph

而后其进行分析，除了具体的汇编码以外，主要是看看里面的分支是不是太多，也就是转移指令是不是太多。太多的话，代码效率会比较差。然后每个块中的代码行数也不能太多。这个主要是因为GPU中的L0 指令cache有限。指令数量多了会导致cache miss，也会对性能有所影响。NV还有一个比较有意思的东西，叫做control code。这个是没办法用NV的工具看到。而且关于底层的SASS汇编码，NV也没有官方的汇编器。意思就是，你可以看汇编，但你不能真正地去改汇编。所以有一系列的开源汇编器，但是没有公司投资金，开源的那些汇编器大多不太好用。

三、CUDA的profiling工具

第三部分主要是介绍一下nsight。Nsight有两个比较好用的工具，分别是systems和compute。

systems主要是用来看timeline，看看程序中的瓶颈是什么。下载安装链接

用户指南如下

效果图大概是下面这样

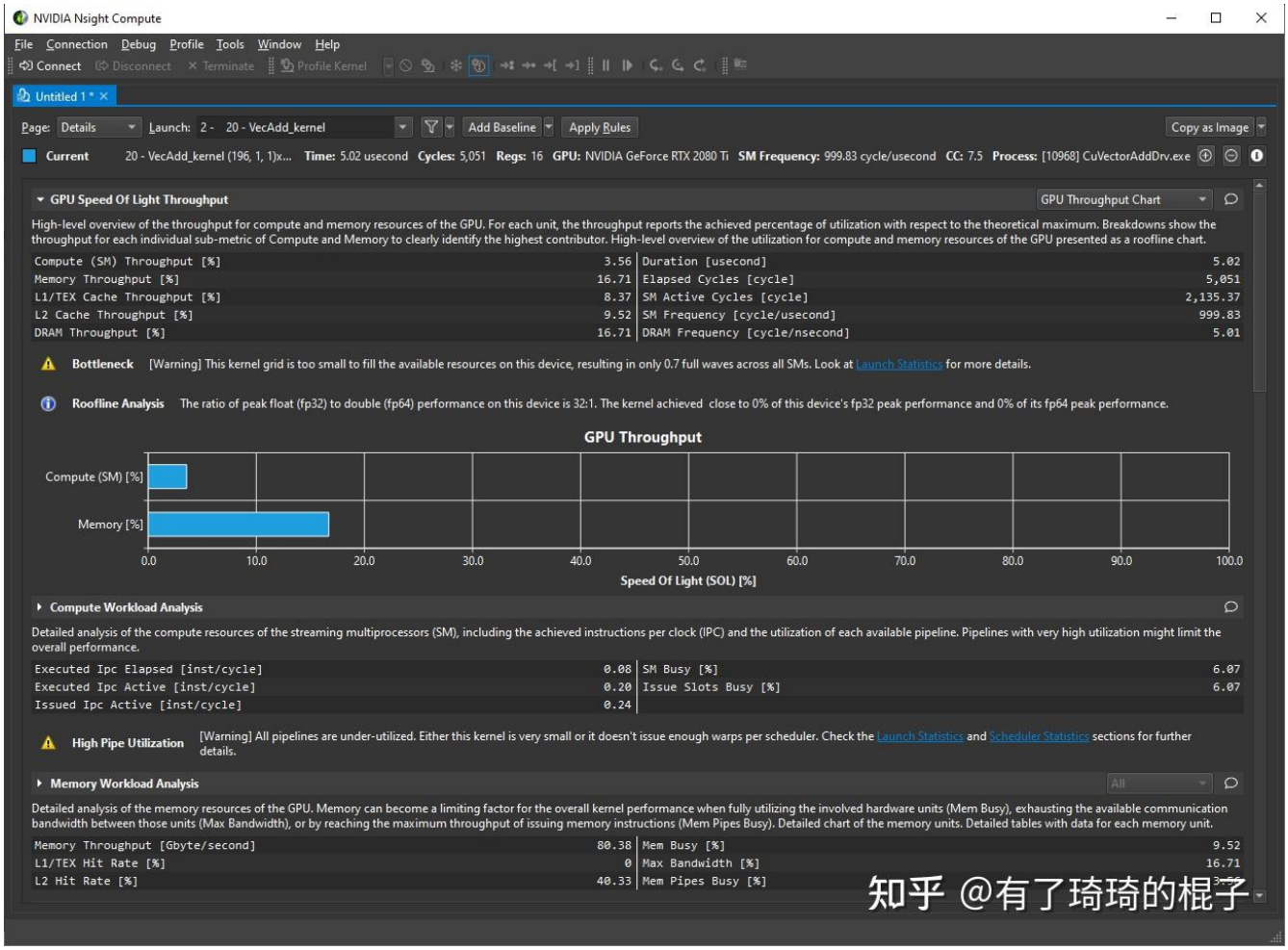
nsight systems timeline

如果是timeline中GPU kernel的占比很小，CPU占比很大，那说明瓶颈在CPU侧，需要注意是不是数据读取花了太多时间。如果GPU kernel的占比很大，说明瓶颈在GPU侧，需要重点花精力去优化GPU kernel实现。还有一种情况是，如果数据一直放在GPU上，但是kernel的时间占比不是特别多，那可能是因为kernel本身不太耗时，可能只运行了4us。但kernel lauch就花了6us。这个时间就要想着采用kernel fusion的方式，尽可能地在一个kernel里面多干点活。

Compute主要是用来分析具体的kernel实现瓶颈。下载安装链接见

用户指南见

在服务器上用ncu分析出report之后，用自己的电脑上安装的nisght compute打开图形界面，看到的界面大致是这个样子。



nsight compute界面

在page处选detail的话，可以先看看的大致的Roofline分析，compute会给出一些建议。然后再看看计算的workload分析和访存的workload分析，针对不同的指标判断kernel是哪里有了瓶颈，随后再针对性地进行优化。

最后，关于element类的OP优化以及CUDA的工具链介绍大致就是这么多，时间有点仓促。后面这篇文章会不断地去补充相关细节，主要是工具的使用方面，会把具体的使用命令再详细讲一下。当然，大家也可以直接看NV的官方用户指南。关于深入浅出GPU系列，还会持续更新。

欢迎大家关注哈:)