

谭升的博客

人工智能基础



Do You Know What Huntington Disease Is? Take A Look

Most of these symptoms are often o
know the list!

Sponsored by: Huntington's | Sear... [LE](#)

【CUDA 基础】6.2 并发内核执行

📅 2018-06-18 | 📁 [CUDA](#) | [Freshman](#) | 💬 0 | 👁

Abstract: 本文介绍内核的并发执行，以及相关的知识

Keywords: 流，事件，深度优先，广度优先，硬件工作队列，默认流阻塞行为

并发内核执行

继续前面的内容，上文中我们说到了流，事件和同步等的概念，以及一些函数的用法，接下来的几个例子，介绍并发内核的几个基本问题，包括不限于以下几个方面：

- 使用深度优先或者广度优先方法的调度工作

- 调整硬件工作队列
- 在Kepler设备和Fermi设备上避免虚假的依赖关系
- 检查默认流的阻塞行为
- 在非默认流之间添加依赖关系
- 检查资源使用是如何影响并发的

非空流中的并发内核

本文我们开始使用NVIDIA提供的另一个可视化工具nvvp进行性能分析，其最大用途在于可视化并发核函数的执行，第一个例子中我们就能清楚地看到各个核函数是如何执行的，本例子中使用了同一个核函数，并将其复制多份，并确保每个核函数的计算要消耗足够的时间，保证执行过程能够被性能分析工具准确的捕捉到。

我们的核函数是：

```
1  __global__ void kernel_1()
2  {
3      double sum=0.0;
4      for(int i=0;i<N;i++)
5          sum=sum+tan(0.1)*tan(0.1);
6  }
7  __global__ void kernel_2()
8  {
9      double sum=0.0;
10     for(int i=0;i<N;i++)
11         sum=sum+tan(0.1)*tan(0.1);
12 }
13 __global__ void kernel_3()
14 {
15     double sum=0.0;
16     for(int i=0;i<N;i++)
17         sum=sum+tan(0.1)*tan(0.1);
18 }
19 __global__ void kernel_4()
20 {
21     double sum=0.0;
22     for(int i=0;i<N;i++)
23         sum=sum+tan(0.1)*tan(0.1);
24 }
```

四个核函数，N是100，tan计算在GPU中应该有优化过的高速版本，但是就算优化，这个也是相对耗时的，足够我们进行观察了。

接着我们按照上节课的套路，创建流，把不同的核函数或者指令放到不同的流中，然后看一下他们的表现。

本文完整的代码在github:https://github.com/Tony-Tan/CUDA_Freshman (欢迎随手star😘)

我们本章主要关注主机代码，下面是创建流的代码：

```
1  cudaStream_t *stream=(cudaStream_t*)malloc(n_stream*sizeof(cudaStream_t));
2  for(int i=0;i<n_stream;i++)
3  {
4      cudaStreamCreate(&stream[i]);
5  }
```

首先声明一个流的头结构，是malloc的注意后面要free掉

然后为每个流的头结构分配资源，也就是Create的过程，这样我们就有n_stream个流可以使用了，接着，我们添加核函数到流，并观察运行效果

```
1  dim3 block(1);
2  dim3 grid(1);
3  cudaEvent_t start,stop;
4  cudaEventCreate(&start);
5  cudaEventCreate(&stop);
6  cudaEventRecord(start);
7  for(int i=0;i<n_stream;i++)
8  {
9      kernel_1<<<grid,block,0,stream[i]>>>();
10     kernel_2<<<grid,block,0,stream[i]>>>();
11     kernel_3<<<grid,block,0,stream[i]>>>();
12     kernel_4<<<grid,block,0,stream[i]>>>();
13 }
14 cudaEventRecord(stop);
15 CHECK(cudaEventSynchronize(stop));
16 float elapsed_time;
17 cudaEventElapsedTime(&elapsed_time,start,stop);
18 printf("elapsed time:%f ms\n",elapsed_time);
```

这不是完整的代码，这个循环是将每个核函数都放入不同的流之中，也就是假设我们有10个流，那么这10

个流中每个流都要按照上面的顺序执行这4个核函数。

注意如果没有

```
1 cudaEventSynchronize(stop)
```

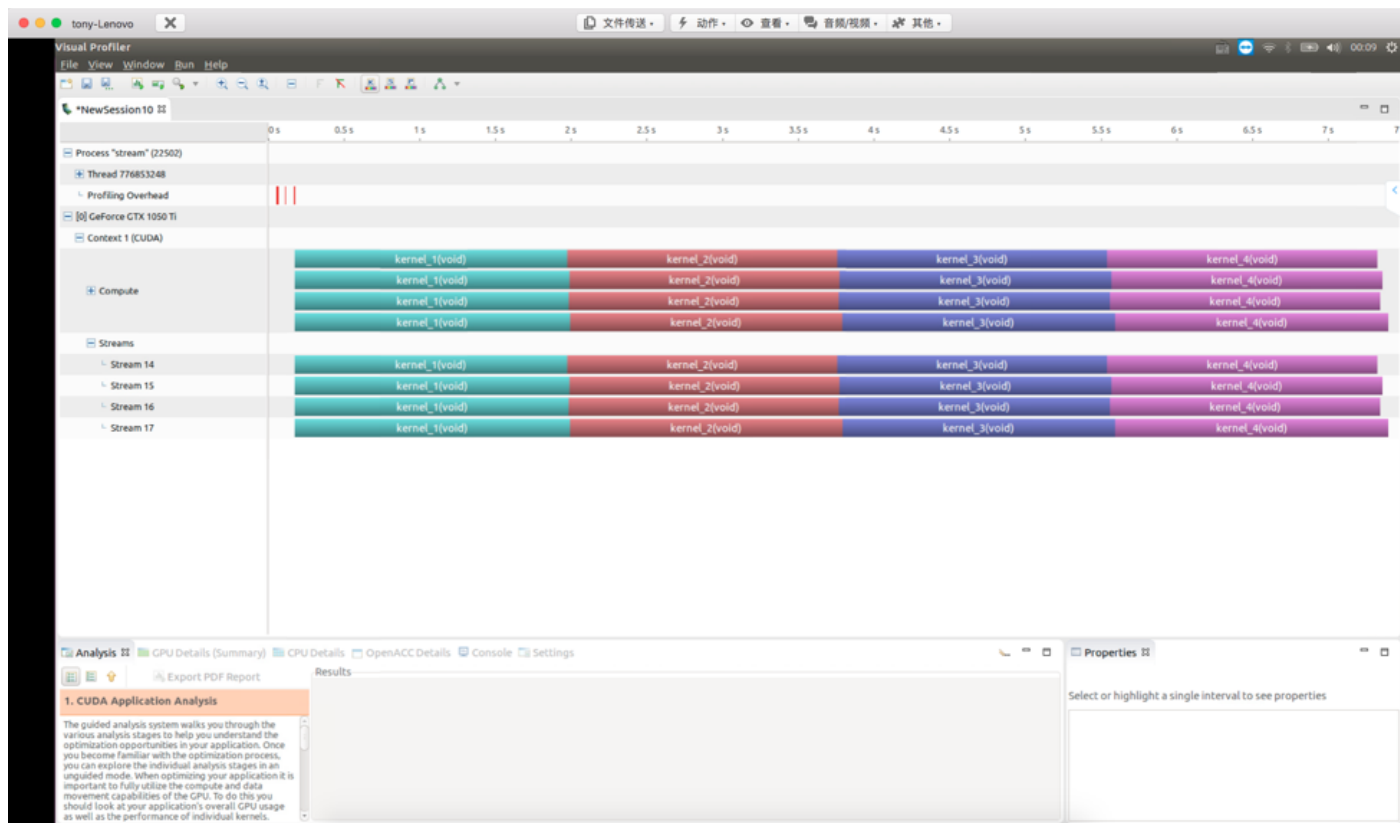
nvvp将会无法运行，因为所有这些都是异步操作，不会等到操作完再返回，而是启动后自动把控制权返回主机，如果没有一个阻塞指令，主机进程就会执行完毕推出，这样就跟设备失联了，nvvp也会相应的报错。

然后我们创建两个事件，然后记录事件之间的时间间隔。这个间隔是不太准确的，因为是异步的。

运行结果如下：

A terminal window screenshot showing a command prompt. The prompt is 'tony@tony-Lenovo:~/Project/CUDA_Freshman/build/30_stream\$'. The user has entered './stream' and the output is 'elapsed time:7168.597168 ms'. The prompt is now 'tony@tony-Lenovo:~/Project/CUDA_Freshman/build/30_stream\$' with a cursor. The terminal window title bar shows 'Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/30_stream — ssh tony@192.168.3.19 — 109x24'.

使用nvvp检测，结果如下：



Fermi GPU 上的虚假依赖关系

虚假依赖我们在上文中讲到过了，这种情况通常出现在只有在比较古老的Fermi架构上出现，原因是其只有一个硬件工作队列，由于我们现在很难找到Fermi架构的GPU了，所以，只能看看书上给出的nvvp结果图了：

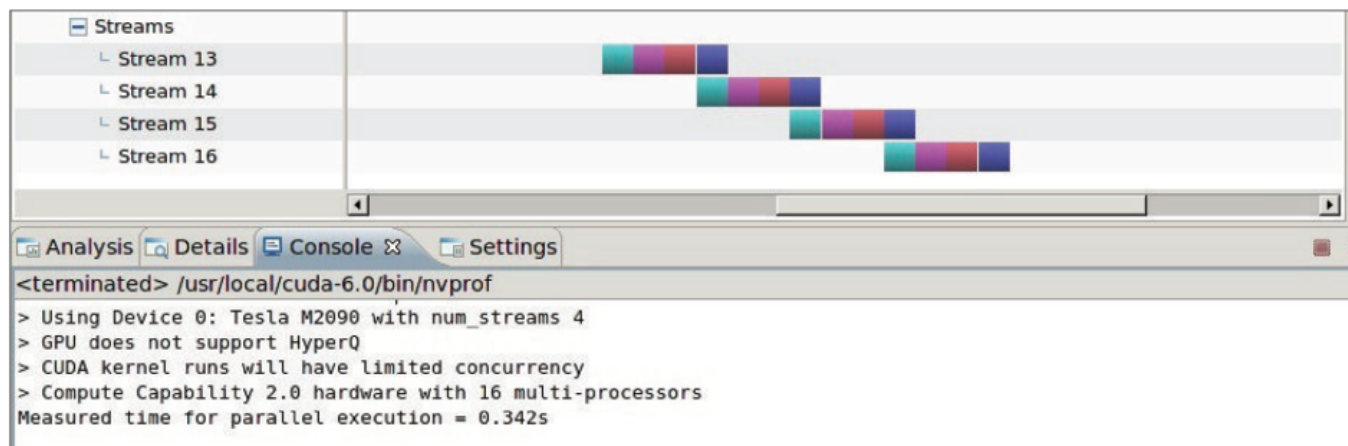


FIGURE 6-6

虚假依赖的问题我们在[流和事件概述](#)已经描述了引起此问题的理论原因，这里就不再解释了。如果你手头只有老机器，这种虚假依赖关系也是可以解决的，原理就是使用广度优先的方法，组织各任务

的方式如下：

```
1 // dispatch job with breadth first way
2 for (int i = 0; i < n_streams; i++)
3   kernel_1<<<grid, block, 0, streams[i]>>>();
4 for (int i = 0; i < n_streams; i++)
5   kernel_2<<<grid, block, 0, streams[i]>>>();
6 for (int i = 0; i < n_streams; i++)
7   kernel_3<<<grid, block, 0, streams[i]>>>();
8 for (int i = 0; i < n_streams; i++)
9   kernel_4<<<grid, block, 0, streams[i]>>>();
```

这样逻辑图就不是：

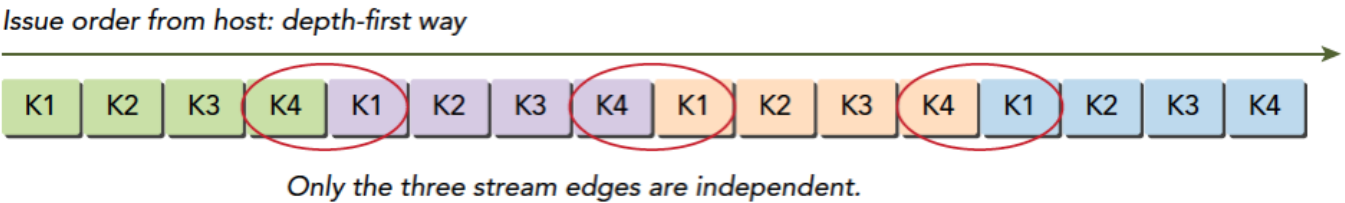


FIGURE 6-7

而是

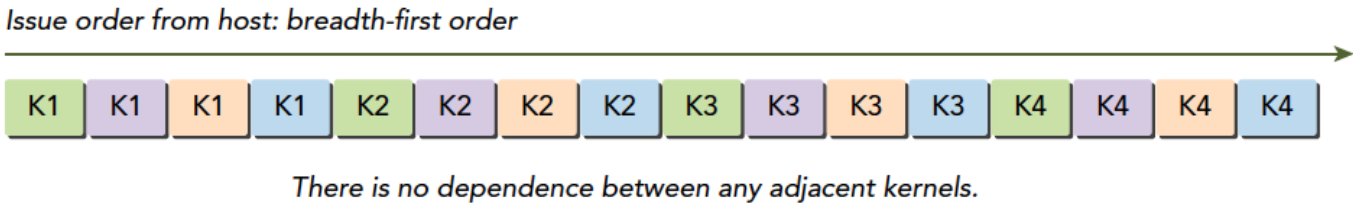


FIGURE 6-8

这样了，这就可以从抽象模型层面避免问题。

广度优先的nvvp结果是：

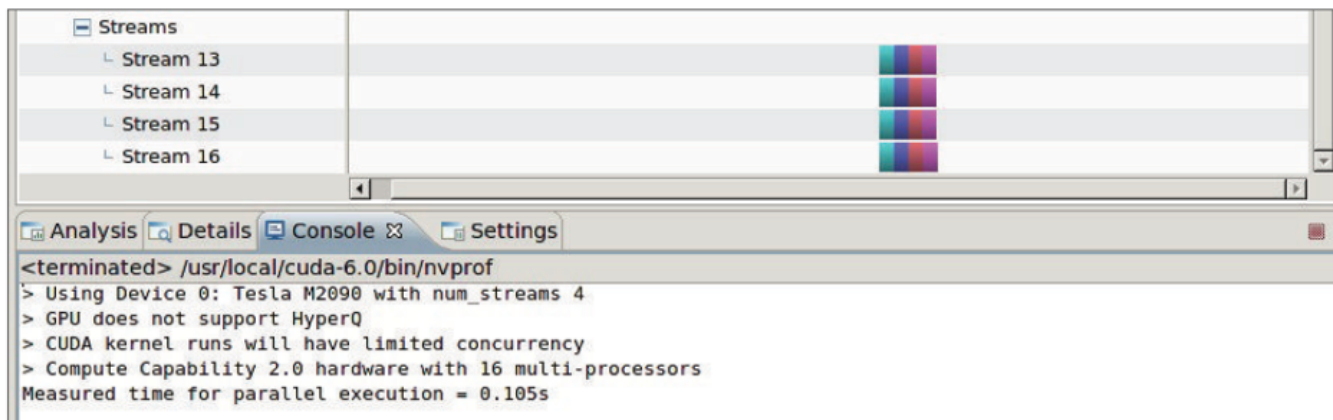


FIGURE 6-9

注意，以上结论都是我从书上原封不动弄下来的。

使用OpenMP的调度操作

OpenMP是一种非常好用的并行工具，比pthread更加好用，但是没有pthread那么灵活，这里我们不光要让核函数或者设备操作多个流处理，同时也让主机在多线程下工作，我们尝试使用每个线程来操作一个流：

```
1  omp_set_num_thread(n_stream);
2  #pragma omp parallel
3      {
4          int i=omp_get_thread_num();
5          kernel_1<<<grid,block,0,stream[i]>>>();
6          kernel_2<<<grid,block,0,stream[i]>>>();
7          kernel_3<<<grid,block,0,stream[i]>>>();
8          kernel_4<<<grid,block,0,stream[i]>>>();
9      }
```

解释下代码


```
1  omp_set_num_thread(n_stream);
2  #pragma omp parallel
```

调用OpenMP的API创建n_stream个线程，然后宏指令告诉编译器下面大括号中的部分就是每个线程都要执行的部分，有点类似于核函数，或者叫做并行单元。

其他代码和上面常规代码都一样，但是注意，OpenMP在mac上支持的不是很好，需要安装GCC，在Linux和

Windows下配置非常简单，Linux下只要连接库函数就行，Windows下如果使用vs系列IDE直接在属性中打开一个开关就可以，我们来观察一下运行结果，注意，这段代码没有使用cmake管理，而是使用命令行编译执行的：

```
1  nvcc -O3 -Xcompiler -fopenmp stream_omp.cu -o stream_omp -lgomp -I ../include/
```

A terminal window titled 'Tony — tony@tony-Lenovo: ~/Project/CUDA_Freshman/31_stream_omp — ssh tony@192.168.3.19 — 109x24'. The prompt is 'tony@tony-Lenovo:~/Project/CUDA_Freshman/31_stream_omp\$'. The user enters './stream_omp'. The output is 'elapsed time:0.463008 ms'. The prompt returns to 'tony@tony-Lenovo:~/Project/CUDA_Freshman/31_stream_omp\$' with a green cursor.

```
tony@tony-Lenovo:~/Project/CUDA_Freshman/31_stream_omp$ ./stream_omp
elapsed time:0.463008 ms
tony@tony-Lenovo:~/Project/CUDA_Freshman/31_stream_omp$
```

CUDA进阶系列还会有OpenMP和CUDA配合的部分，后面会详细说。

用环境变量调整流行为

Kepler支持的最大Hyper-Q 工作队列数是32，但是在默认情况下并不是全部开启，而是被限制成8个，原因是每个工作队列只要开启就会有资源消耗，如果用不到32个可以把资源留给需要的8个队列，修改这个配置的方法是修改主机系统的环境变量。

对于Linux系统中，修改方式如下：

```
1  #For Bash or Bourne Shell:
2  export CUDA_DEVICE_MAX_CONNECTIONS=32
3  #For C-Shell:
4  setenv CUDA_DEVICE_MAX_CONNECTIONS 32
```

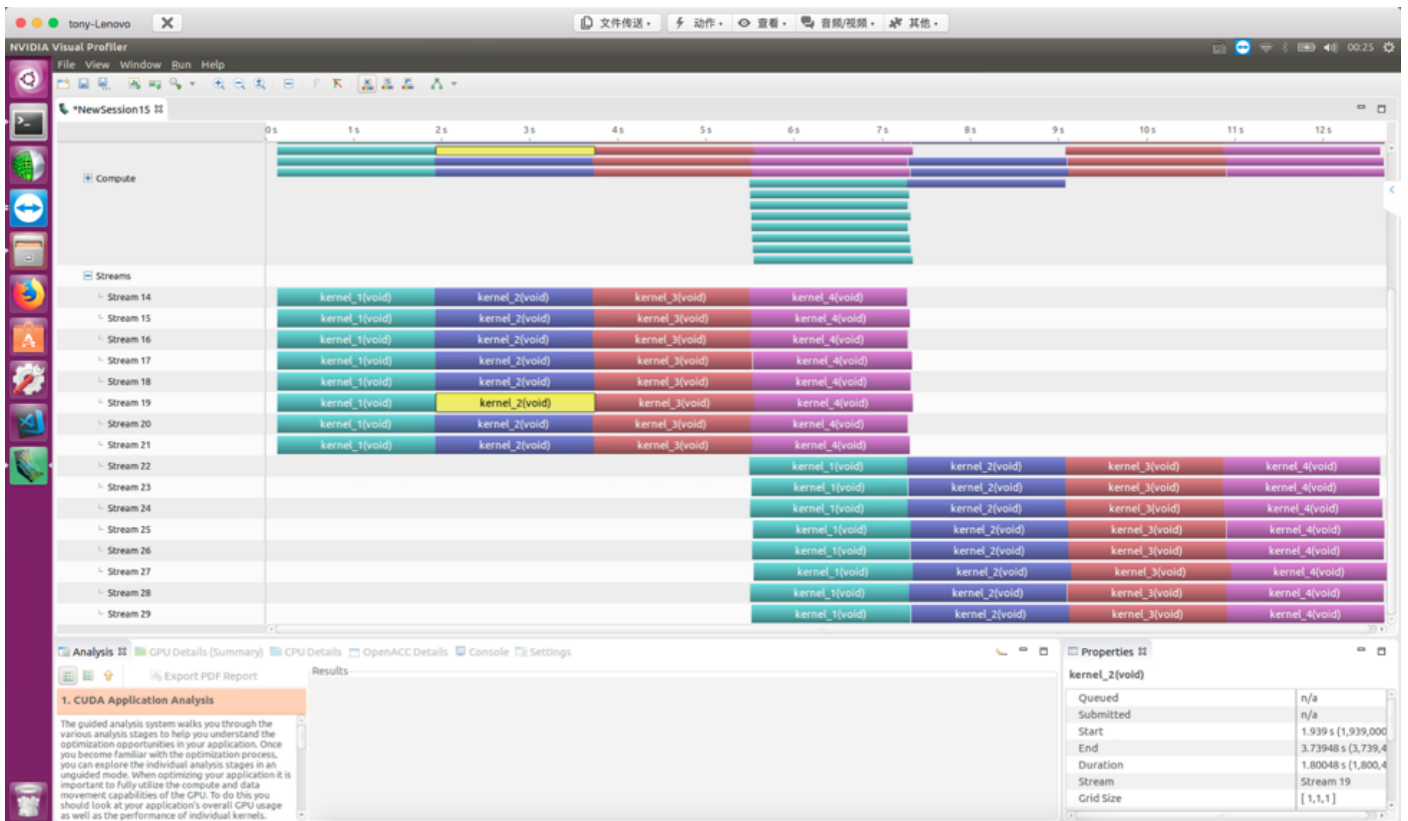
另一种修改方法是直接在程序里写，这种方法更好用通过底层驱动修改硬件配置：


```
1 setenv("CUDA_DEVICE_MAX_CONNECTIONS", "32", 1);
```

然后我们把前面的深度优先的代码改一下，加入上面这句指令,并把n_stream改成16，就可以得到如下结果：



16个流，8个工作队列的结果：

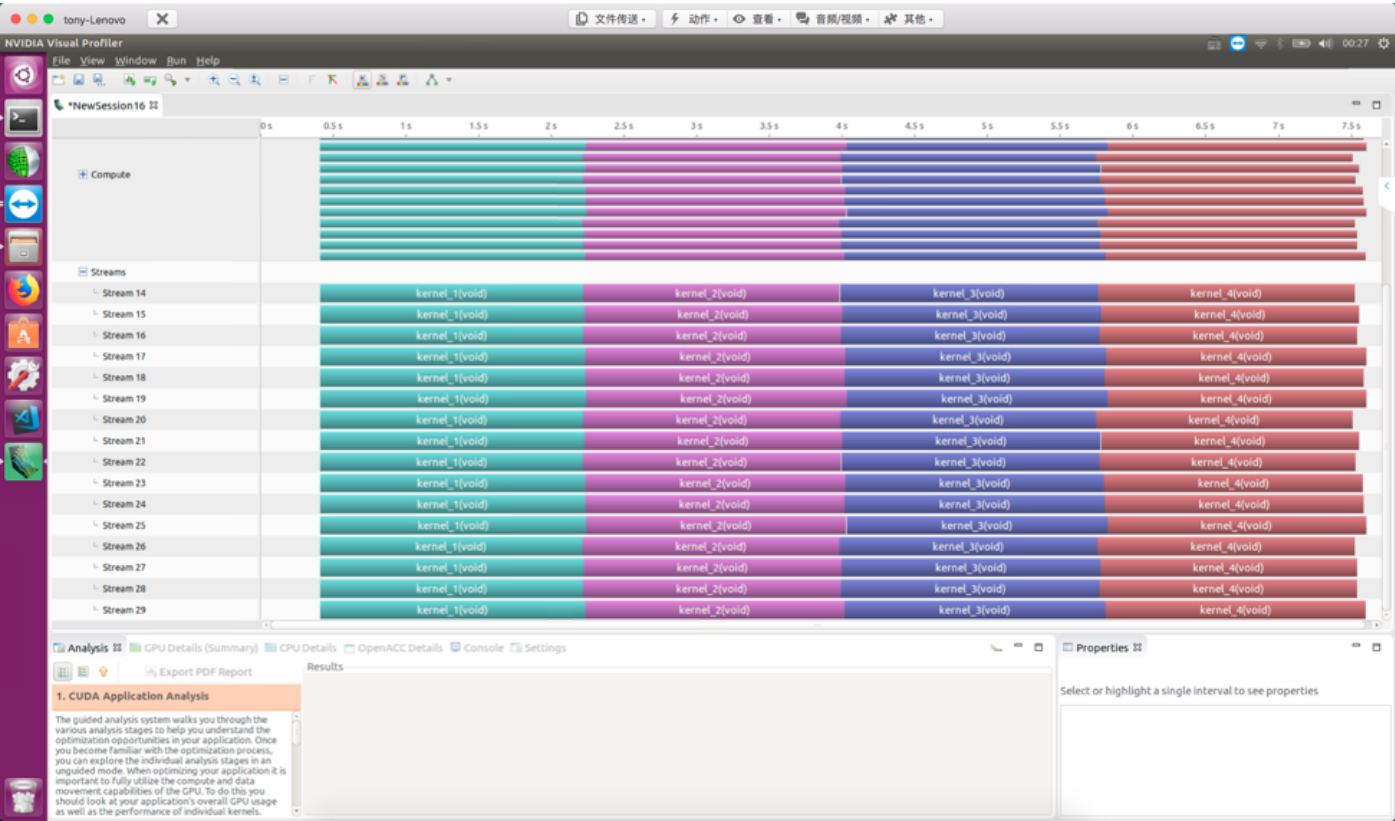


调用

```
1 setenv("CUDA_DEVICE_MAX_CONNECTIONS", "32", 1);
```

修改成32个队列

16个流，32个工作队列的结果：



GPU资源的并发限制

限制内核并发数量的最根本的还是GPU上面的资源，资源才是性能的极限，性能最高无非是在不考虑算法进化的前提下，资源利用率最高的结果。当每个内核的线程数增加的时候，内核级别的并行数量就会下降，比如，我们把

```
1 dim3 block(1);  
2 dim3 grid(1);
```

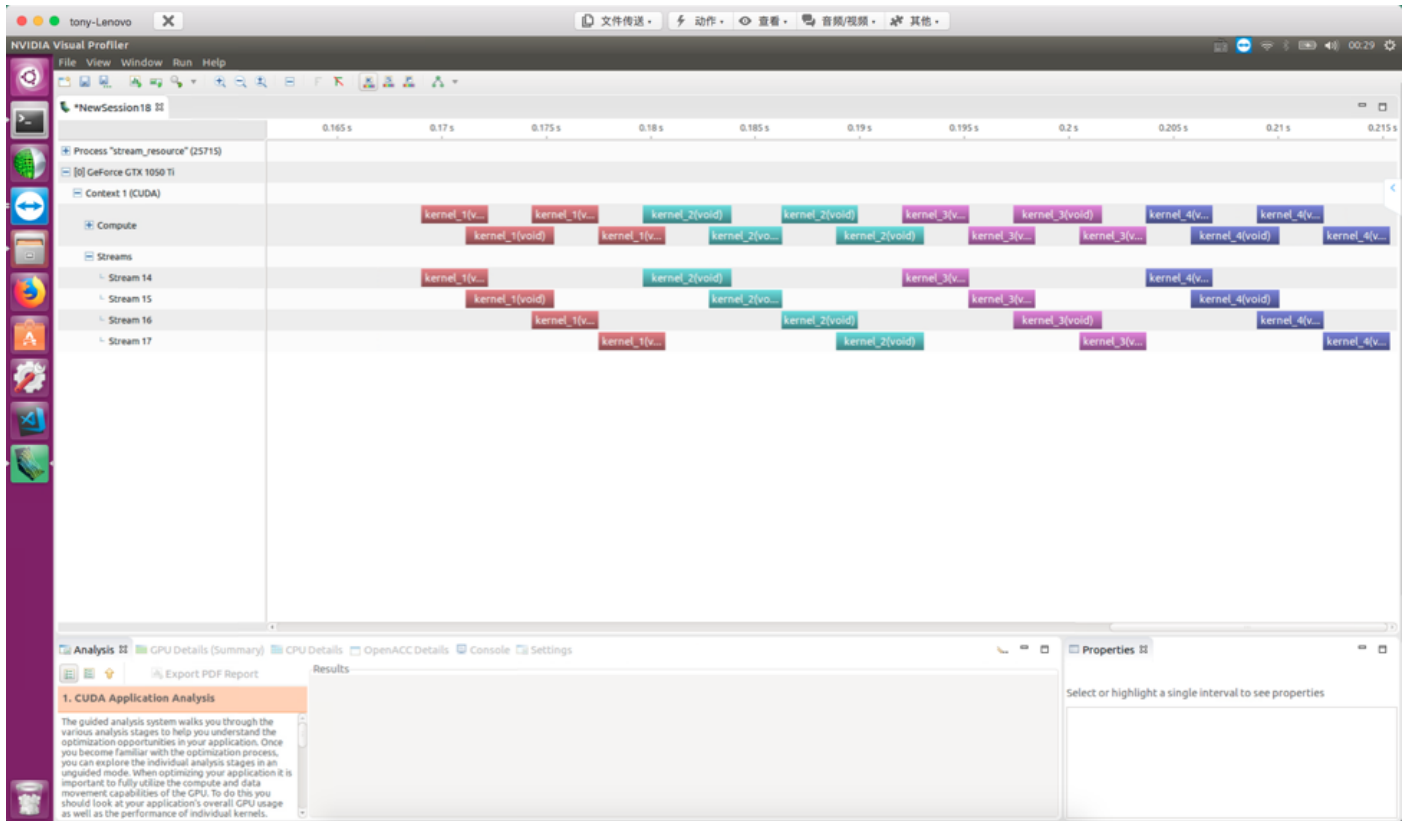
升级到

```
1 dim3 block(16,32);
```

↑ 79%

```
2 dim3 grid(32);
```

4个流，nvvp结果是：



默认流的阻塞行为

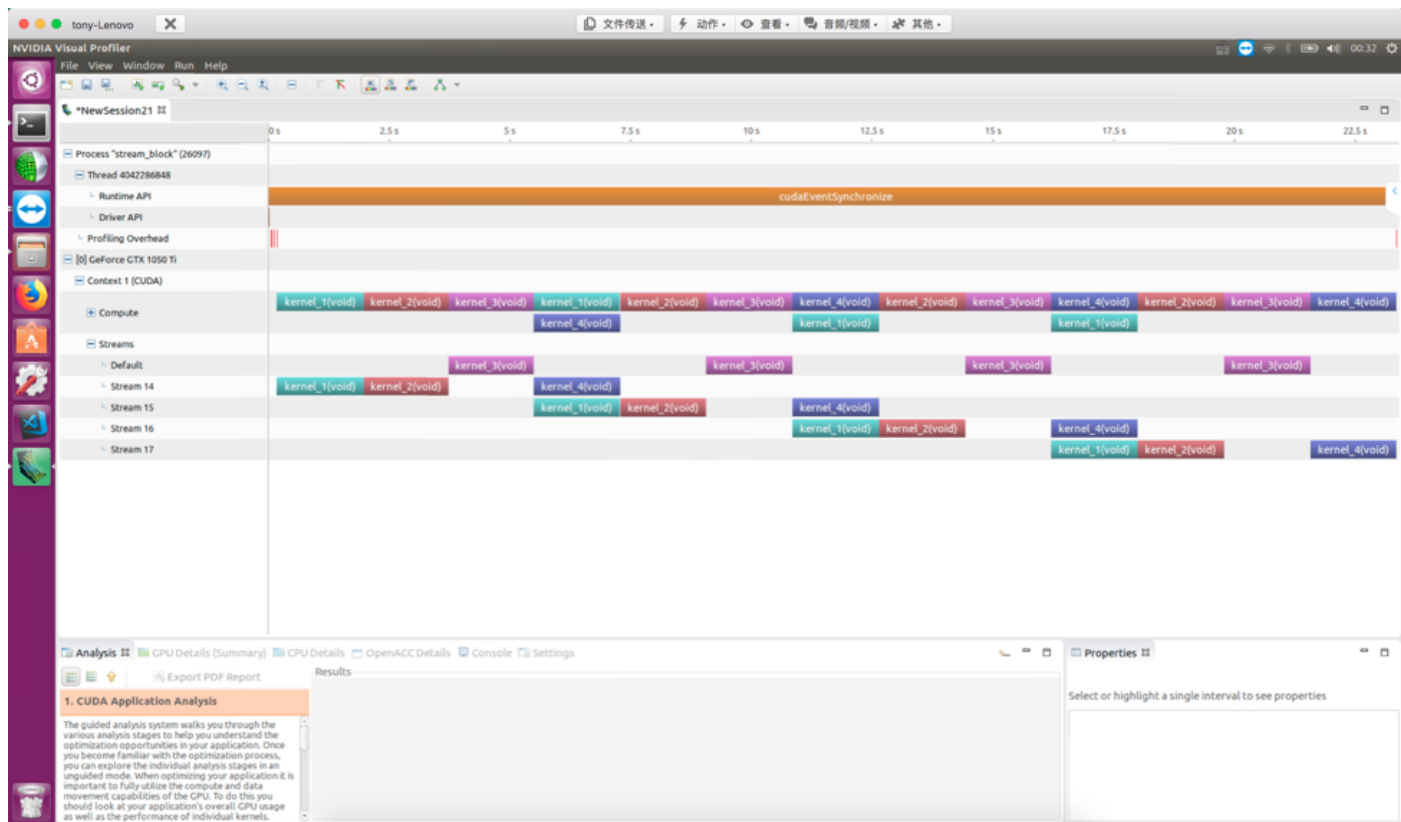
默认流也就是空流对于飞空流中的阻塞流是有阻塞作用的，这句话有点难懂，首先我们没有声明流的那些GPU操作指令，核函数是在空流上执行的，空流是阻塞流，同时我们声明定义的流如果没有特别指出，声明的也是阻塞流，换句话说，这些流的共同特点，无论空流与非空流，都是阻塞的。

那么这时候空流（默认流）对非空流的阻塞操作就要注意一下了。

```
1 for(int i=0;i<n_stream;i++)
2 {
3     kernel_1<<<grid,block,0,stream[i]>>>();
4     kernel_2<<<grid,block,0,stream[i]>>>();
5     kernel_3<<<grid,block>>>();
6     kernel_4<<<grid,block,0,stream[i]>>>();
7 }
```

注意，kernel_3是在空流（默认流）上的，从NVVP的结果中可以看出，所有kernel_3 启动以后，所有其他

的流中的操作全部被阻塞：



创建流间依赖关系

流之间的虚假依赖关系是需要避免的，而经过我们设计的依赖又可以保证流之间的同步性，避免内存竞争，这时候我们要使用的就是事件这个工具了，换句话说，我们可以让某个特定流等待某个特定的事件，这个事件可以再任何流中，只有此事件完成才能进一步执行等待此事件的流继续执行。

这种事件往往不用于计时，所以可以在生命的时候声明成 `cudaEventDisableTiming` 的同步事件：

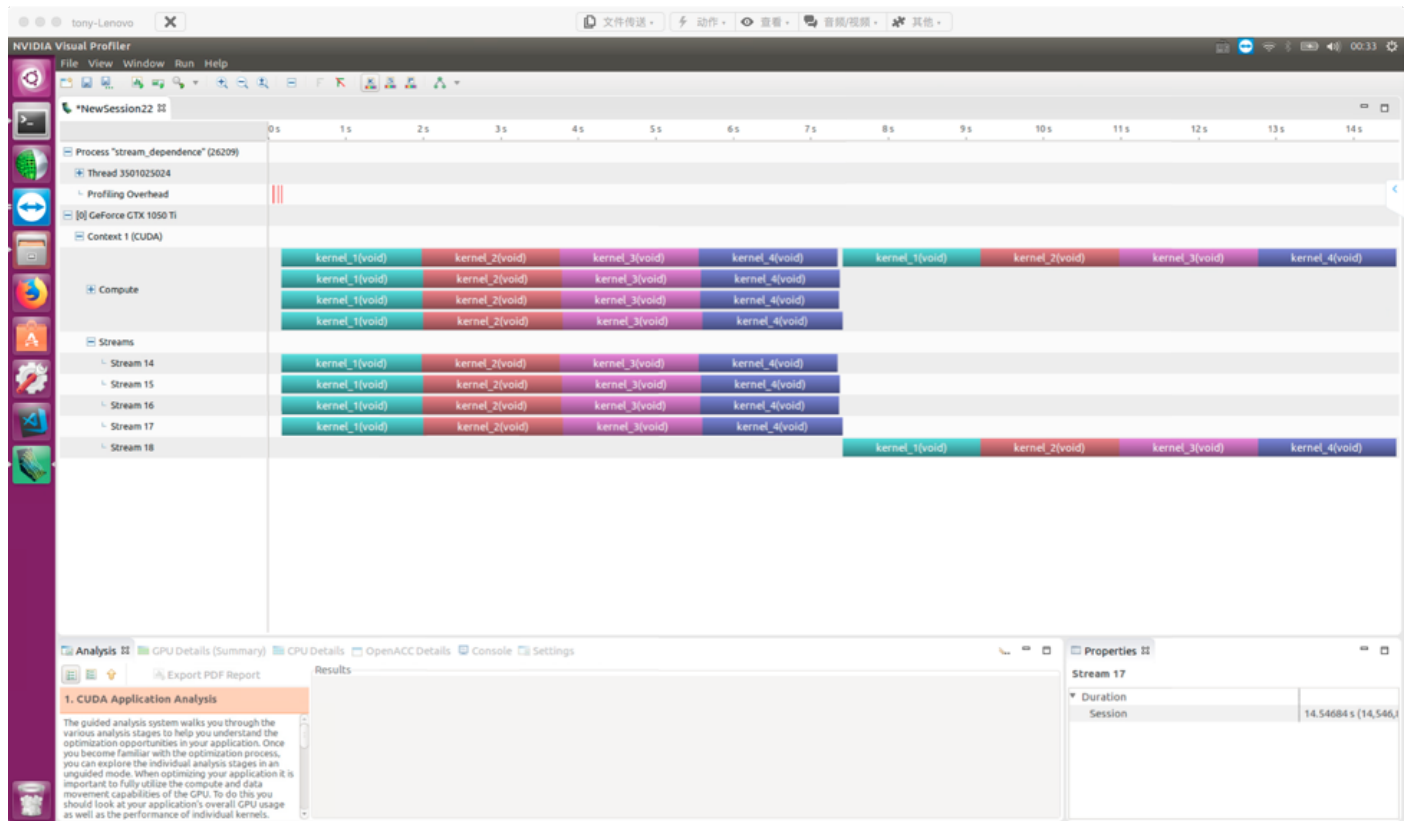
```
1  cudaEvent_t * event=(cudaEvent_t *)malloc(n_stream*sizeof(cudaEvent_t));
2  for(int i=0;i<n_stream;i++)
3  {
4      cudaEventCreateWithFlag(&event[i],cudaEventDisableTiming);
5  }
```

在流中加入指令：

```
1  for(int i=0;i<n_stream;i++)
2  {
3      kernel_1<<<grid,block,0,stream[i]>>>();
```

```
4 kernel_2<<<grid,block,0,stream[i]>>>();
5 kernel_3<<<grid,block,0,stream[i]>>>();
6 kernel_4<<<grid,block,0,stream[i]>>>();
7 cudaEventRecord(event[i],stream[i]);
8 cudaStreamWaitEvent(stream[n_stream-1],event[i],0);
9 }
```

这时候，最后一个流（第5个流）都会等到前面所有流中的事件完成，自己才会完成，nvvp结果如下



总结


本文研究了如何使用并发内核提高应用整体的效率，以及流阻塞的相关知识。

下一篇我们介绍一个更加实用的技术——通过流来屏蔽数据传输延迟。

本文作者： 谭升

本文链接：<https://face2ai.com/CUDA-F-6-2-并发内核执行/>

版权声明： 本博客所有文章除特别声明外，均采用 [CC BY-NC-SA 4.0](#) 许可协议。转载请注明出处！

 相关文章