

CUDA Stream和Event





课程目标

理论部分



学习使用CUDA Stream和Event



学习使用NVVP工具

技能部分



CUDA Stream和Event的使用技巧和经验



NVVP的使用技巧和经验



CUDA Stream

CUDA stream是GPU上task 的执行队列，所有CUDA操作（kernel，内存拷贝等）都是在stream上执行的。

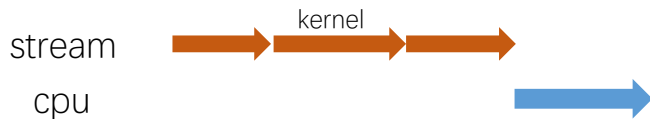
CUDA stream有两种

- 隐式流，又叫默认流，NULL流

所有的CUDA操作默认运行在隐式流里。隐式流里的GPU task和CPU端计算是同步的。

举例： $n = 1$ 这行代码，必须等上面三行都执行完，才会执行它。

```
cudaMemcpy(..., cudaMemcpyHostToDevice);  
kernel<<<grid, block>>>(...);  
cudaMemcpy(..., cudaMemcpyDeviceToHost)  
int n = 1;
```





CUDA Stream

CUDA stream是GPU上task 的执行队列，所有CUDA操作（kernel，内存拷贝等）都是在stream上执行的。

CUDA stream有两种

- 显式流：显式申请的流

显式流里的GPU task和CPU端计算是**异步**的。不同显式流内的GPU task执行也是异步的。

```
cudaStream_t stream;  
cudaStreamCreate(stream);  
cudaMemcpyAsync(..., cudaMemcpyHostToDevice, stream);  
kernel<<<grid, block, 0, stream>>>(...);  
cudaMemcpyAsync(..., cudaMemcpyDeviceToHost, stream)  
int n = 1;|
```

stream



cpu



并行的，异步的执行



CUDA Stream API

- 定义

```
cudaStream_t stream;
```

- 创建

```
cudaStreamCreate(&stream);
```

- 数据传输

```
cudaMemcpyAsync(dst, src, size, type, stream)
```

- kernel在流中执行

```
kernel_name<<<grid, block, sharedMemSize, stream>>>(argument list);
```

- 同步和查询

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream)
```

```
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

- 销毁

```
cudaError_t cudaStreamDestroy(cudaStream_t stream);
```



CUDA Stream demo

```
//创建两个流
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
...
//两个流，每个流有三个命令
for (int i = 0; i < 2; ++i) {
    //从主机内存复制数据到设备内存
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice,
stream[i]);
    //执行Kernel处理
    MyKernel <<<grid, block, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    //从设备内存到主机内存
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost,
stream[i]);
}
// 同步流
for (int i = 0; i < 2; i++)
    cudaStreamSynchronize(stream[i]);
...
//销毁流
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

定义

cudaStream_t stream;



CUDA Stream demo

```
//创建两个流
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
...
//两个流，每个流有三个命令
for (int i = 0; i < 2; ++i) {
    //从主机内存复制数据到设备内存
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice,
stream[i]);
    //执行Kernel处理
    MyKernel <<<grid, block, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    //从设备内存到主机内存
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost,
stream[i]);
}
// 同步流
for (int i = 0; i < 2; i++)
    cudaStreamSynchronize(stream[i]);
...
//销毁流
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

创建

cudaStreamCreate(&stream);



CUDA Stream demo

```
//创建两个流
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
...
//两个流，每个流有三个命令
for (int i = 0; i < 2; ++i) {
    //从主机内存复制数据到设备内存
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice,
stream[i]);
    //执行Kernel处理
    MyKernel <<<grid, block, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    //从设备内存到主机内存
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost,
stream[i]);
}
// 同步流
for (int i = 0; i < 2; i++)
    cudaStreamSynchronize(stream[i]);
...
//销毁流
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

数据传输

cudaMemcpyAsync(dst, src, size, type, stream)



CUDA Stream demo

```
//创建两个流
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
...
//两个流，每个流有三个命令
for (int i = 0; i < 2; ++i) {
    //从主机内存复制数据到设备内存
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice,
stream[i]);
    //执行Kernel处理
    MyKernel <<<grid, block, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    //从设备内存到主机内存
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost,
stream[i]);
}
// 同步流
for (int i = 0; i < 2; i++)
    cudaStreamSynchronize(stream[i]);
...
//销毁流
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

kernel在流中执行
kernel_name<<<grid, block, sharedMemSize, stream>>>();



CUDA Stream demo

```
//创建两个流
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)      同步
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
...
//两个流，每个流有三个命令
for (int i = 0; i < 2; ++i) {
    //从主机内存复制数据到设备内存
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice,
stream[i]);
    //执行Kernel处理
    MyKernel <<<grid, block, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    //从设备内存到主机内存
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost,
stream[i]);
}
// 同步流
for (int i = 0; i < 2; i++)
    cudaStreamSynchronize(stream[i]);
...
//销毁流
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```



CUDA Stream demo

```
//创建两个流
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)          销毁
    cudaStreamCreate(&stream[i]);    cudaError_t cudaStreamDestroy(cudaStream_t stream);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
...
//两个流，每个流有三个命令
for (int i = 0; i < 2; ++i) {
    //从主机内存复制数据到设备内存
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice,
stream[i]);
    //执行Kernel处理
    MyKernel <<<grid, block, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    //从设备内存到主机内存
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost,
stream[i]);
}
// 同步流
for (int i = 0; i < 2; i++)
    cudaStreamSynchronize(stream[i]);
...
//销毁流
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```



CUDA Stream 优点

- CPU计算和kernel计算并行
- CPU计算和数据传输并行
- 数据传输和kernel计算并行
- kernel计算并行



CUDA Stream 优点

- CPU计算和kernel计算并行
- CPU计算和数据传输并行
- 数据传输和kernel计算并行
- kernel计算并行

强调知识点

显式流里的GPU task与CPU端 task 的执行是异步的，使用stream一定要注意同步！

`cudaStreamSynchronize()` 同步一个流

`cudaDeviceSynchronize()` 同步该设备上的所有流

`cudaStreamQuery()` 查询一个流任务是否完成



CUDA Stream-数据传输和GPU计算重叠 demo

Serial

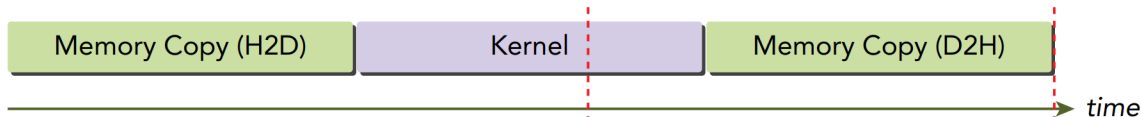
```
cudaMemcpy(..., cudaMemcpyHostToDevice);  
kernel<<<grid, block>>>(...);  
cudaMemcpy(..., cudaMemcpyDeviceToHost)
```

Tips: 数据量和计算量要足够大

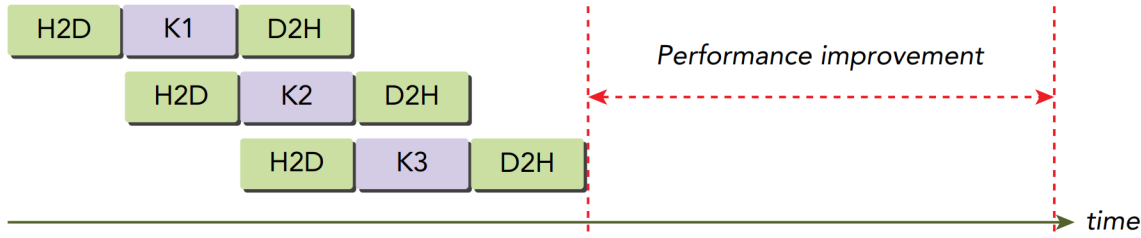
CUDA stream

```
for (int i = 0; i < nStreams; i++) {  
    int offset = i * bytesPerStream;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], bytesPerStream, streams[i]);  
    kernel<<<grid, block, 0, streams[i]>>>(&d_a[offset]);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], bytesPerStream, streams[i]);  
}  
for (int i = 0; i < nStreams; i++) {  
    cudaStreamSynchronize(streams[i]);  
}
```

Serial

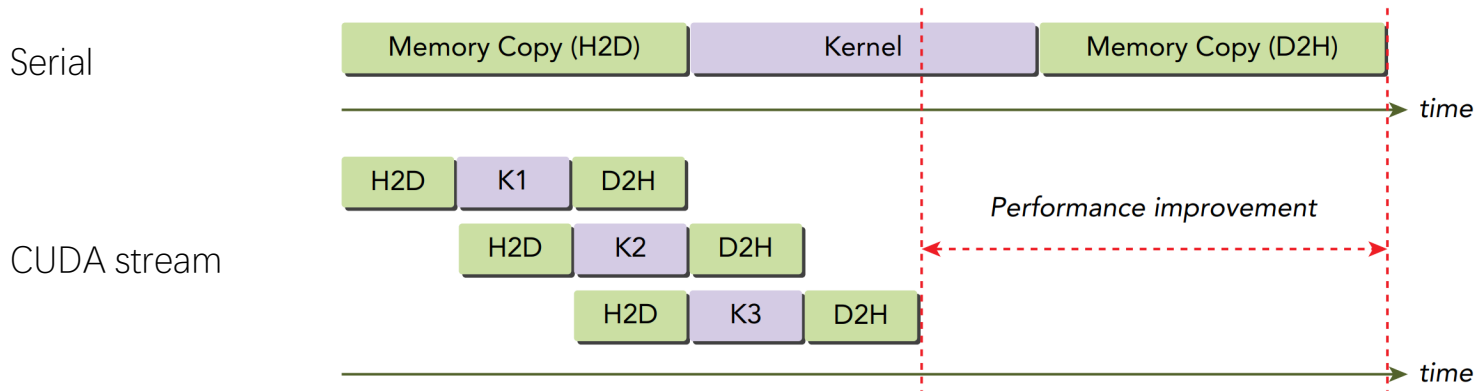


CUDA stream





CUDA Stream-数据传输和GPU计算重叠



H2D 和 D2H 为什么没有重叠？它们已经在不同stream上了。

因为CPU和GPU的数据传输是经过PCIe总线的，PCIe上的操作是顺序的。

带有双工PCIe总线的设备可以重叠两个数据传输，但它们必须在不同的流和不同的方向上。



CUDA Stream 优先级

GPU 算力 3.5 及以上，即Kepler架构及以上

API

```
cudaError_t cudaStreamCreateWithPriority(cudaStream_t* pStream, unsigned int flags, int priority);
```

```
cudaError_t cudaDeviceGetStreamPriorityRange(int *leastPriority, int *greatestPriority);
```

- 只对kernel有效
- 较低的整数值表示较高的流优先级。



CUDA Stream 为什么有效?

多流为什么会有效，流越多越好么？

一、PCIe总线传输速度慢，是瓶颈，会导致传输数据的时候GPU处于空闲等待状态。

多流可以实现数据传输与kernel计算的并行。

二、一个kernel往往用不了整个GPU的算力。多流可以让多个kernel同时计算，充分利用GPU算力。

三、不是流越多越好。GPU内可同时并行执行的流数量是有限的。

CUDA加速，kernel合并，将小任务合并成大任务，更有效。



CUDA Stream 为什么有效?

CUDA加速, kernel合并, 将小任务合并成大任务, 更有效。

为什么?

思考: GPU kernel耗时最大在哪里?

计算密集型: 耗时在计算, 一次访存, 数十次甚至上百次计算

访存密集型: 耗时在访存, 一次访存, 几次计算



CUDA Stream 为什么有效?

CUDA加速，kernel合并，将小任务合并成大任务，更有效。

思考：GPU kernel耗时最大在哪里？

计算密集型：耗时在计算，一次访存，数十次甚至上百次计算

访存密集型：耗时在访存，一次访存，几次计算

GPU一般处理简单可并行计算，大部分kernel都是访存密集型



CUDA Stream 为什么有效?

CUDA加速, kernel合并, 将小任务合并成大任务, 更有效。

向量 A 、 B 、 C , 大小都为 n 。

$$A * B = D; A * C = E; E + D = O;$$

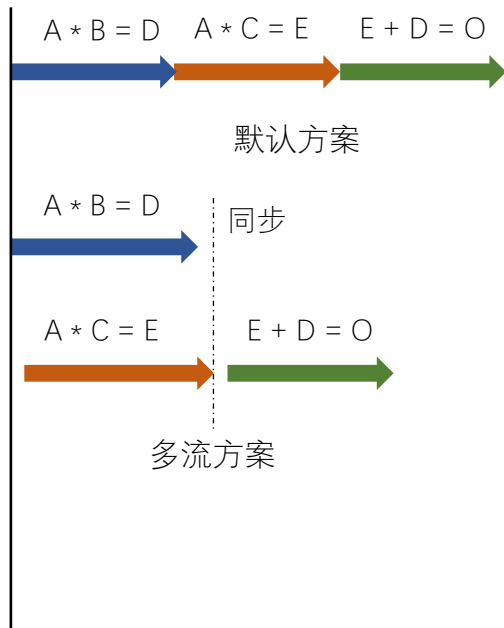


CUDA Stream 为什么有效?

CUDA加速, kernel合并, 将小任务合并成大任务, 更有效。

向量 A 、 B 、 C , 大小都为 n 。

$A * B = D; A * C = E; E + D = O;$



访问两次 A , 一次 B , 一次 C ,
两次 E , 两次 D , 一次 O
共9次读写
三次计算

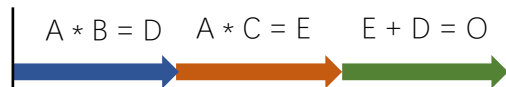


CUDA Stream 为什么有效?

CUDA加速, kernel合并, 将小任务合并成大任务, 更有效。

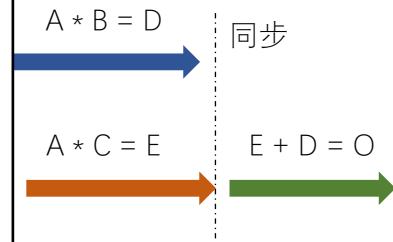
向量 A 、 B 、 C , 大小都为 n 。

$A * B = D; A * C = E; E + D = O;$



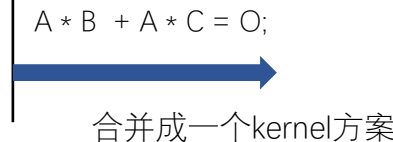
默认方案

访问两次A, 一次B, 一次C,
两次E, 两次D, 一次O
共9次读写
三次计算



多流方案

访问一次A, 一次B, 一次C,
一次O
共4次读写
三次计算



合并成一个kernel方案



CUDA Stream 默认流的表现

```
int main()
{
    const int num_streams = 8;

    cudaStream_t streams[num_streams];
    float *data[num_streams];

    for (int i = 0; i < num_streams; i++) {
        cudaStreamCreate(&streams[i]);
        cudaMalloc(&data[i], N * sizeof(float));
    }

    for (int i = 0; i < num_streams; i++) {
        // launch one worker kernel per stream
        kernel<<<1, 64, 0, streams[i]>>>(data[i], N);

        // launch a dummy kernel on the default stream
        kernel<<<1, 1>>>(0, 0);
    }

    return 0;
}
```

```
nvcc ./stream_test.cu -o stream_legacy
```

<https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>



CUDA Stream 默认流的表现

```
int main()
{
    const int num_streams = 8;

    cudaStream_t streams[num_streams];
    float *data[num_streams];

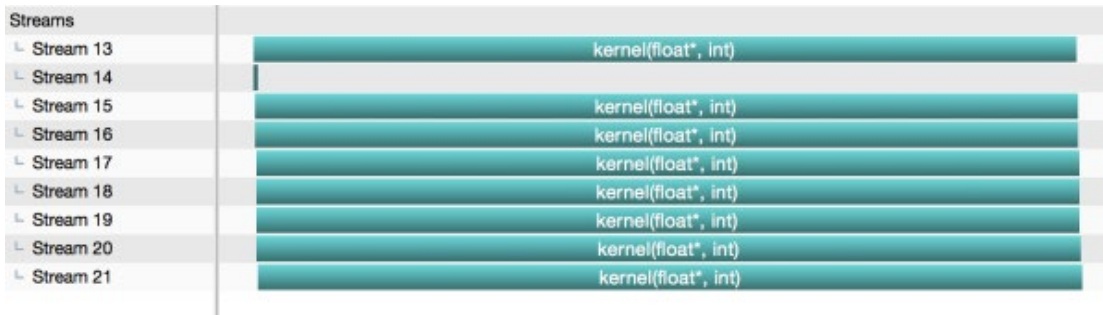
    for (int i = 0; i < num_streams; i++) {
        cudaStreamCreate(&streams[i]);
        cudaMalloc(&data[i], N * sizeof(float));
    }

    for (int i = 0; i < num_streams; i++) {
        // launch one worker kernel per stream
        kernel<<<1, 64, 0, streams[i]>>>(data[i], N);

        // launch a dummy kernel on the default stream
        kernel<<<1, 1>>>(0, 0);
    }

    return 0;
}
```

期望运行结果



```
nvcc ./stream_test.cu -o stream_legacy
```

<https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>



CUDA Stream 默认流的表现

单线程内，默认流的执行是**同步**的，显式流的执行是异步的

```
int main()
{
    const int num_streams = 8;

    cudaStream_t streams[num_streams];
    float *data[num_streams];

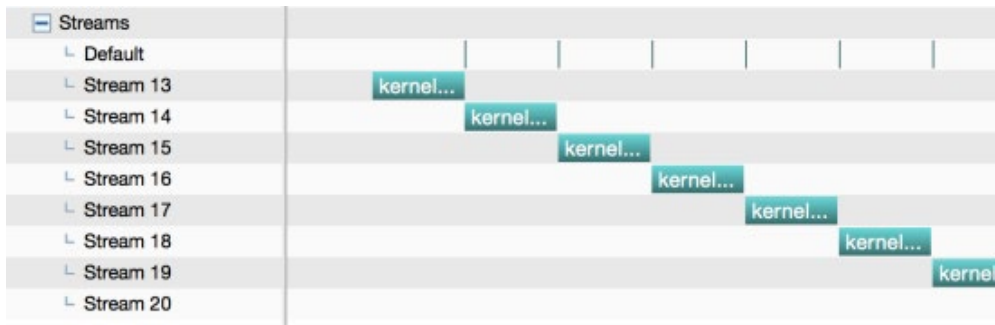
    for (int i = 0; i < num_streams; i++) {
        cudaStreamCreate(&streams[i]);
        cudaMalloc(&data[i], N * sizeof(float));
    }

    for (int i = 0; i < num_streams; i++) {
        // launch one worker kernel per stream
        kernel<<<1, 64, 0, streams[i]>>>(data[i], N);

        // launch a dummy kernel on the default stream
        kernel<<<1, 1>>>(0, 0);
    }

    return 0;
}
```

实际运行结果



```
nvcc ./stream_test.cu -o stream_legacy
```

<https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>

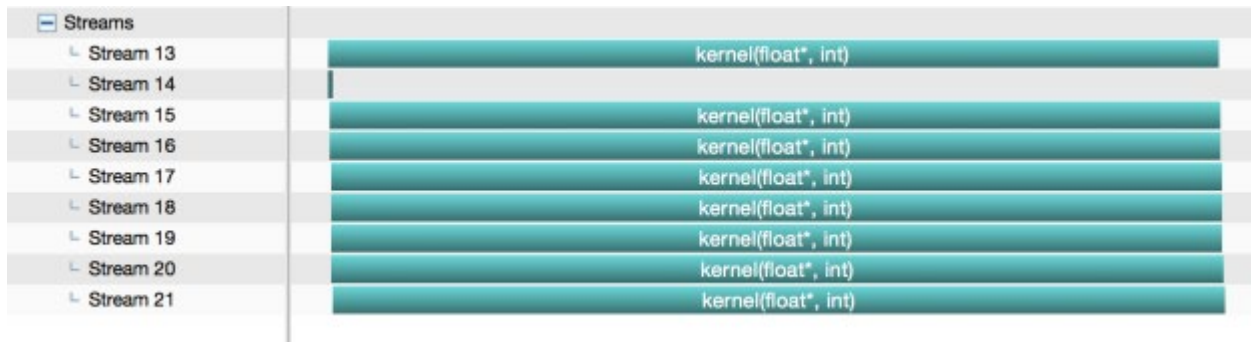


CUDA Stream 默认流的表现

单线程内，编译加上--default-stream per-thread 后

默认流的执行是异步的，显式流的执行是异步的

```
nvcc --default-stream per-thread ./stream_test.cu -o stream_per-thread
```



<https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>



CUDA Stream 默认流的表现

多线程下，默认流的表现是什么呢？是一个默认流还是多个默认流？

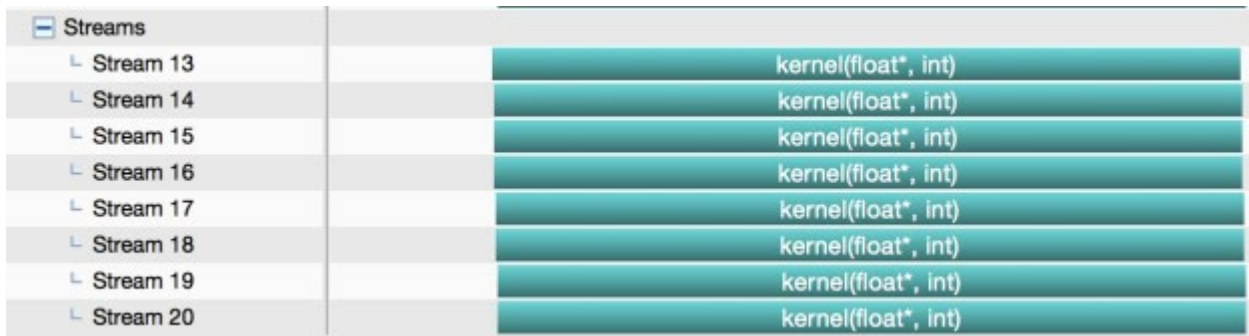
```
nvcc ./pthread_test.cu -o pthreads_legacy
```



Streams	
Default	kernel(...) kernel(...) kernel(...) kernel(...) kernel(...) kernel(...) kernel(...) kernel(...)

默认多线程共享一个默认流

```
nvcc --default-stream per-thread ./pthread_test.cu -o pthreads_per_thread
```



Streams	
Stream 13	kernel(float*, int)
Stream 14	kernel(float*, int)
Stream 15	kernel(float*, int)
Stream 16	kernel(float*, int)
Stream 17	kernel(float*, int)
Stream 18	kernel(float*, int)
Stream 19	kernel(float*, int)
Stream 20	kernel(float*, int)

每个线程都有一个默认流

<https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>



CUDA Event API

CUDA Event, 在stream中插入一个事件, 类似于打一个标记位, 用来记录stream是否执行到当前位置。Event有两个状态, 已被执行和未被执行。

- 定义

```
cudaEvent_t event
```

- 创建

```
cudaError_t cudaEventCreate(cudaEvent_t* event);
```

- 插入流中

```
cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0);
```

- 销毁

```
cudaError_t cudaEventDestroy(cudaEvent_t event);
```

- 同步和查询

```
cudaError_t cudaEventSynchronize(cudaEvent_t event);
```

```
cudaError_t cudaEventQuery(cudaEvent_t event);
```

- 进阶同步函数

```
cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event);
```



CUDA Event

最常用的用法是测时间

```
//使用event计算时间
float time_elapsed=0;
cudaEvent_t start,stop;
cudaEventCreate(&start);    //创建Event
cudaEventCreate(&stop);

cudaEventRecord( start,0);    //记录当前时间
mul<<<blocks, threads, 0, 0>>>(dev_a,NUM);
cudaEventRecord( stop,0);    //记录当前时间

cudaEventSynchronize(start);    //Waits for an event to complete.
cudaEventSynchronize(stop);    //Waits for an event to complete.Record之前的任务
cudaEventElapsedTime(&time_elapsed,start,stop);    //计算时间差

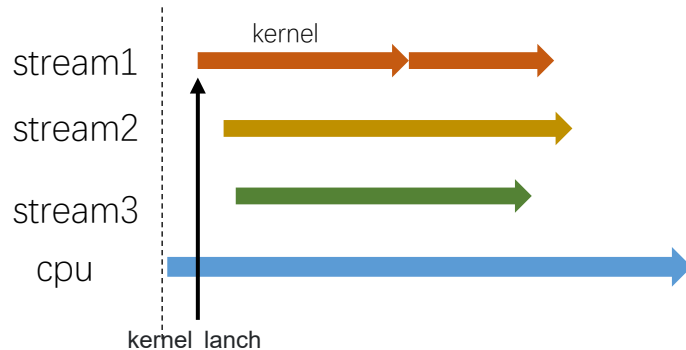
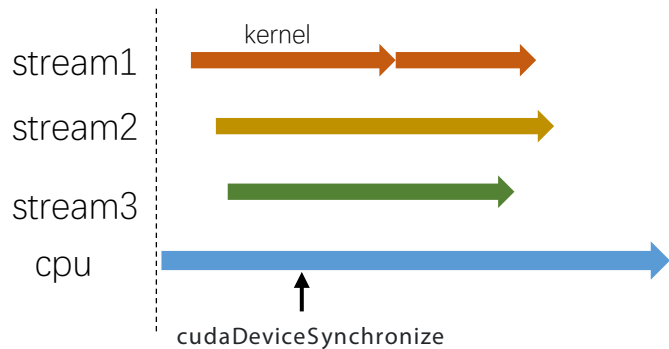
cudaEventDestroy(start);    //destory the event
cudaEventDestroy(stop);
printf("执行时间: %f(ms)\n",time_elapsed);
```



CUDA 同步操作

CUDA中的显式同步按粒度可以分为四类

- `device synchronize` 影响很大

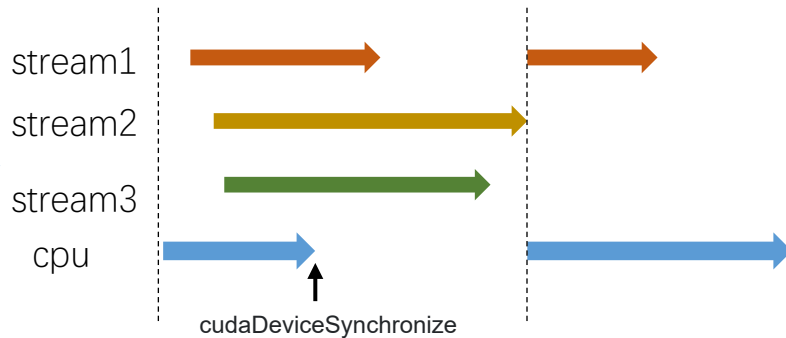
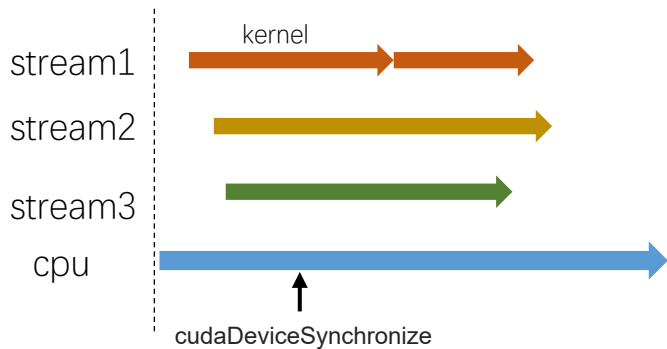
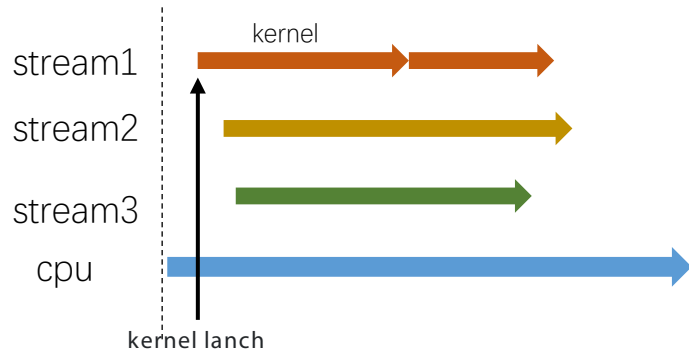




CUDA 同步操作

CUDA中的显式同步按粒度可以分为四类

- `device synchronize` 影响很大

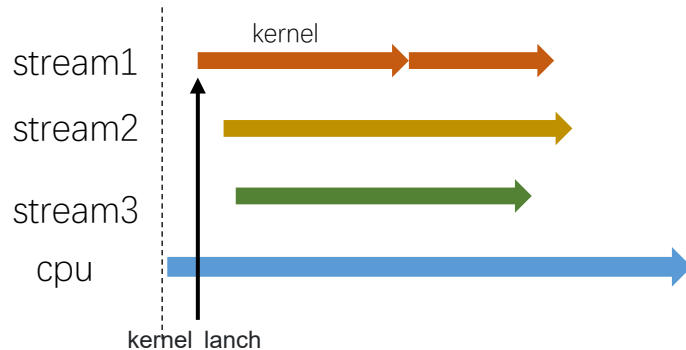




CUDA 同步操作

CUDA中的显式同步按粒度可以分为四类

- device synchronize 影响很大
- stream synchronize 影响单个流和CPU

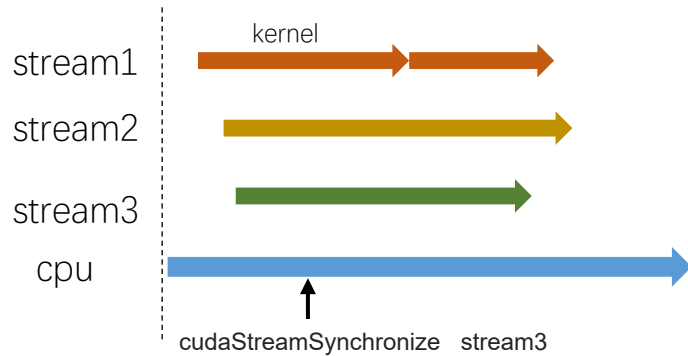




CUDA 同步操作

CUDA中的显式同步按粒度可以分为四类

- device synchronize 影响很大
- stream synchronize 影响单个流和CPU

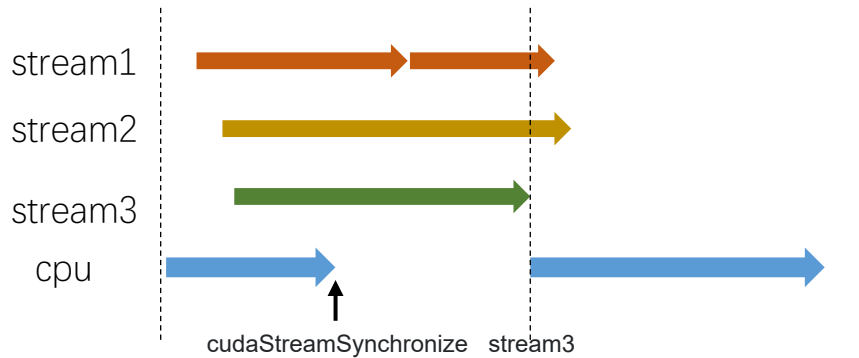
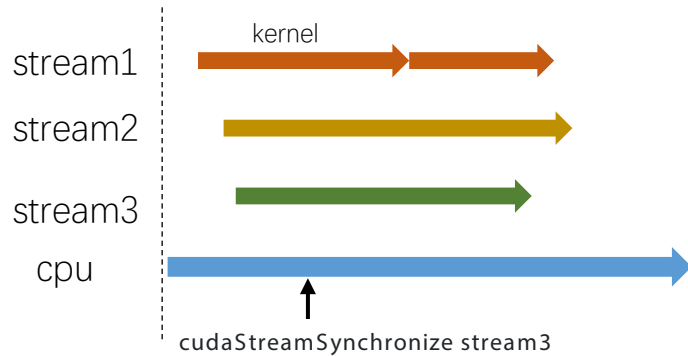




CUDA 同步操作

CUDA中的显式同步按粒度可以分为四类

- device synchronize 影响很大
- stream synchronize 影响单个流和CPU

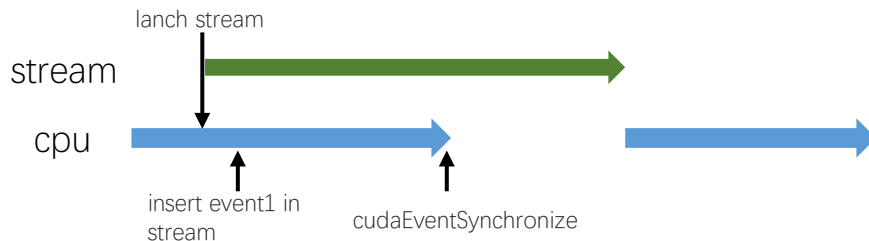




CUDA 同步操作

CUDA中的显式同步按粒度可以分为四类

- device synchronize 影响很大
- stream synchronize 影响单个流和CPU
- event synchronize 影响CPU，更细粒度的同步

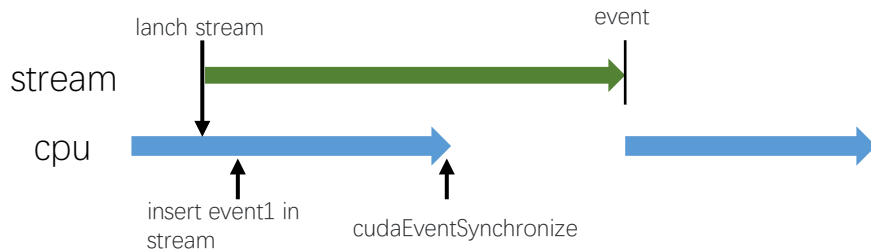




CUDA 同步操作

CUDA中的显式同步按粒度可以分为四类

- device synchronize 影响很大
- stream synchronize 影响单个流和CPU
- event synchronize 影响CPU，更细粒度的同步





CUDA 同步操作

CUDA中的显式同步按粒度可以分为四类

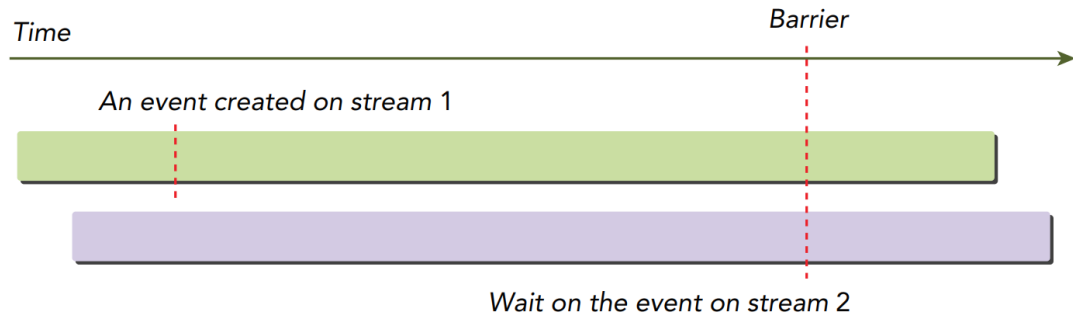
- device synchronize 影响很大
- stream synchronize 影响单个流和CPU
- event synchronize 影响CPU，更细粒度的同步
- synchronizing across streams using an event



CUDA 同步操作

```
cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event);
```

该函数会指定该stream等待特定的event，该event可以关联到相同或者不同的stream



Stream2会等待stream1中的event完成后继续执行。



NVVP

NVIDIA Visual Profiler (NVVP) 是NVIDIA推出的跨平台的CUDA程序性能分析工具。

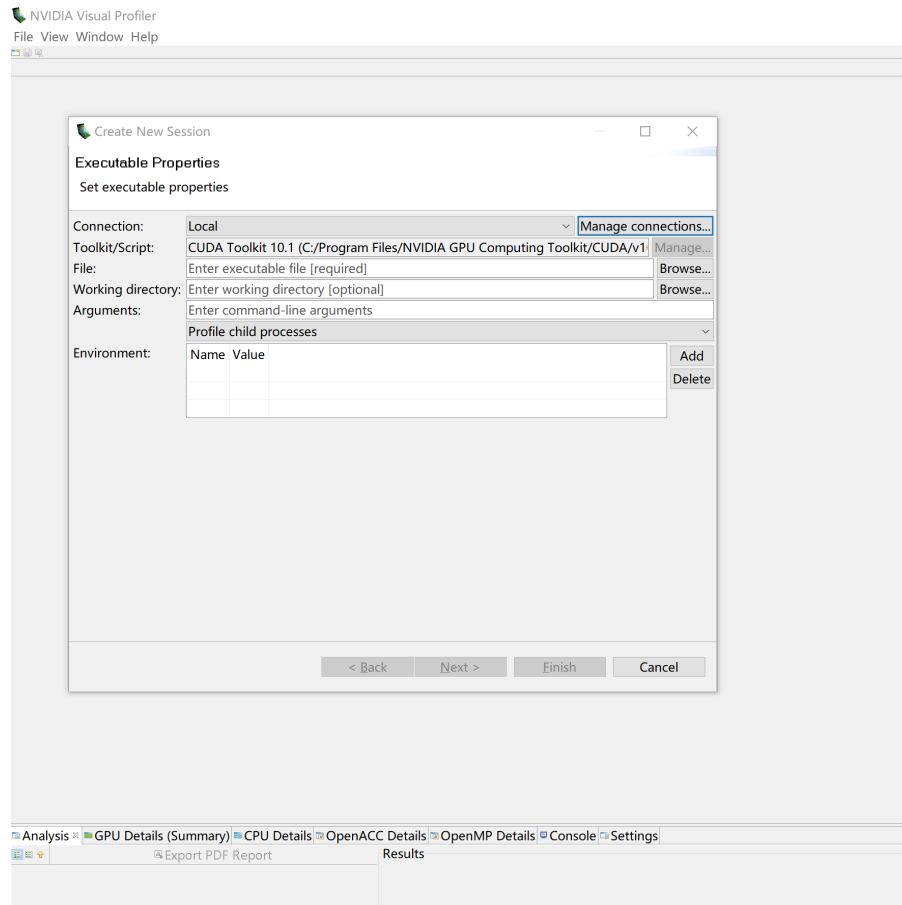
- 随CUDA安装，不需要额外安装。
- 可自定义配置+图形化界面，可以快速找到程序中的性能瓶颈。
- 以时间线的形式展示CPU和GPU操作。
- 可以查看数据传输和kernel的各种软件参数（速度，kernel启动时间等）和硬件参数（L1 cache命中率等）。



NVVP基本用法

图形化界面用法
Windows linux通用

- 打开nvvp
- File->New Session
- 在File里选择CUDA程序bin
- 选择执行





NVVP基本用法

Linux命令行用法

1. nvprof ./cuda_bin

```
==9261== Profiling application: ./tHogbomCleanHemi
==9261== Profiling result:

Time(%)      Time       Calls      Avg       Min       Max    Name
58.73%  737.97ms     1000  737.97us  424.77us  1.1405ms subtractPSFLoop_kernel(float const *,
int, float*, int, int, int, int, int, int, float, float)
38.39%  482.31ms     1001  481.83us  475.74us  492.16us findPeakLoop_kernel(MaxCandidate*, float
const *, int)
1.87%   23.450ms         2  11.725ms  11.721ms  11.728ms [CUDA memcpy HtoD]
1.01%   12.715ms     1002  12.689us  2.1760us  10.502ms [CUDA memcpy DtoH]
```

2. nvprof -o cuda_bin.nvvp ./cuda_bin

将cuda_bin.nvvp传回windows，使用NVVP打开

3.使用 nvprof 进行远程分析

感谢聆听！

Thanks for Listening

