Homework 3
CSCI 680 GPU Architectures

Yu Chen

1. **What is the speed up between the non-Stream and Stream version of Vector Add? Where are the improvements coming from?**
   When I was using 1 million vector size, it didn't have much performance improvement. Thus I tried to increase the vector size to 10 million. The result showed as below:

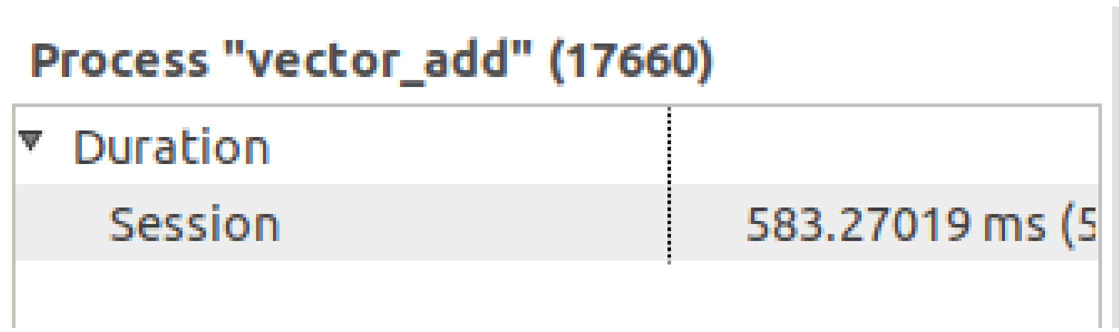

Figure 1: non-Stream

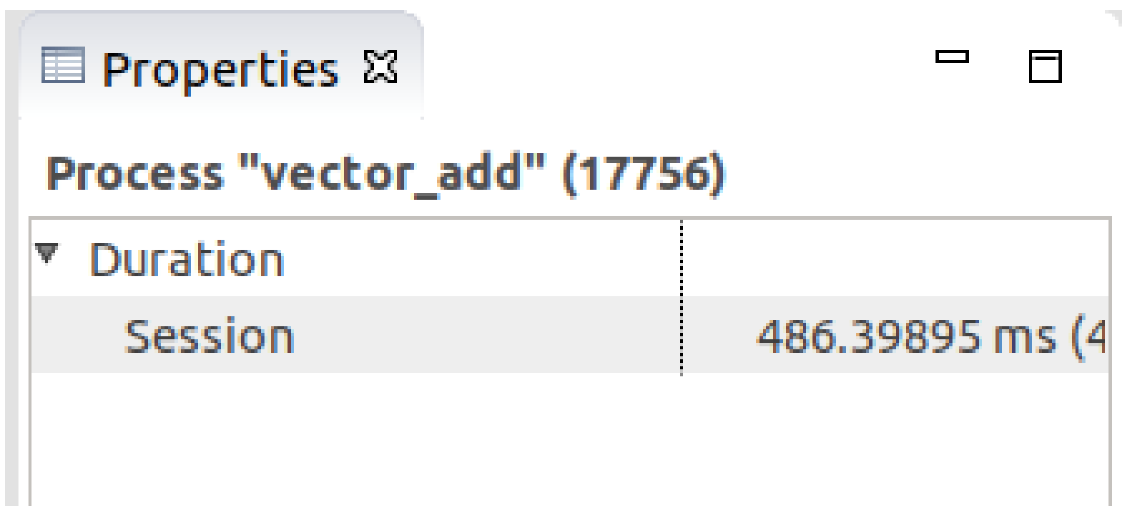

Figure 2: Stream

The Stream version has almost 25% performance improvement. The improvement came from the parallel between kernel and memcpy. We can see in figure 3, the kernel and memcpy ran sequential. However in figure 4, stream 18's kernel ran parallel with stream 17's memcpy. This parallel came from the cuda's async API because these APIs will not block.

2. **How can data transfers be further optimized?**
   Cite by NVIDIA office doc, "How to Overlap Data Transfers in CUDA C/C++ ", we have four ways to optimize the data transfer:

   - Minimize the amount of data transferred between host and device when possible, even if that means running kernels on the GPU that get little or no speed-up compared to running them on the host CPU.

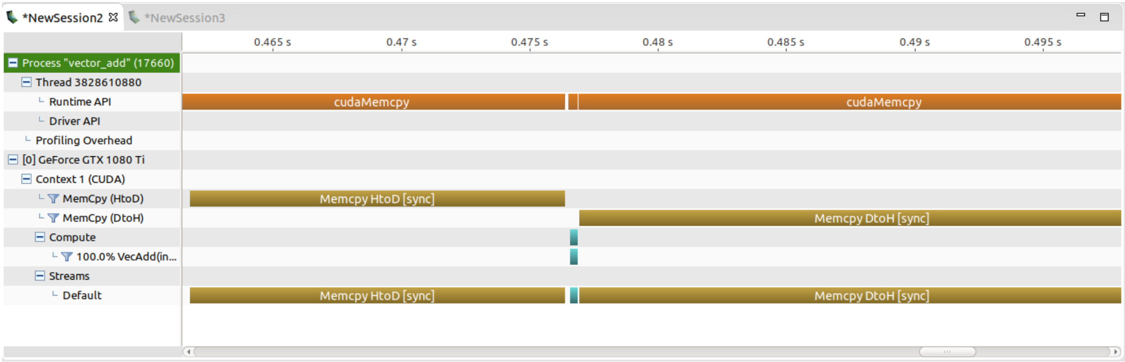

Figure 3: non-Stream detail

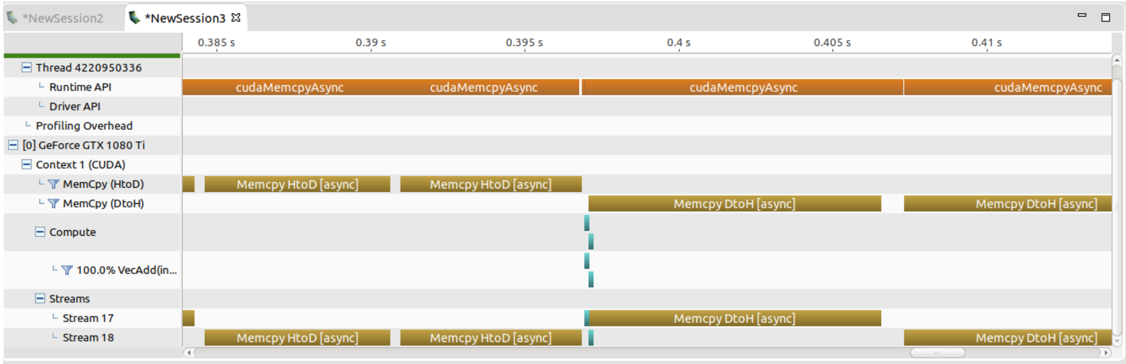

Figure 4: Stream detail

- Higher bandwidth is possible between the host and the device when using page-locked (or pinned) memory.
- Batching many small transfers into one larger transfer performs much better because it eliminates most of the per-transfer overhead.
- Data transfers between the host and device can sometimes be overlapped with kernel execution and other data transfers.

The first way can't be used in our benchmark. And we can use page-locked memory to achieve higher bandwidth. The third way and the fourth way will be a trade-off in our benchmarks because when we increase the streams' number, it will get more overlap between kernel and memcpy but too many streams will cause extra transfer overhead. Thus increasing streams to a appropriate number will optimize data transfer too.

3. **Do ordering of various CUDA API calls on the host side matter when implementing streams? Why or why not?**
Logically, it's no matter of CUDA API calls' ordering because calls in a same stream will be sequential and streams will be parallel. However, NVIDIA doc "How to Overlap Data Transfers in CUDA C/C++" mentions that the following two approaches perform very differently depending on the specific generation of GPU used.

```
for (int i=0; i<nStreams; ++i)
{
  cudaMemcpyAsync(A_d+i*s_size, A_h+i*s_size, s_byte, HtoD, streams[i]);
  cudaMemcpyAsync(B_d+i*s_size, B_h+i*s_size, s_byte, HtoD, streams[i])
  //kernel vector add
  cudaMemcpyAsync(C_h+i*s_size, C_d+i*s_size, s_byte, DtoH, streams[i])
}
```

Figure 5: approache 1

```
for (int i=0; i<nStreams; ++i)
{
  cudaMemcpyAsync(A_d+i*s_size, A_h+i*s_size, s_byte, HtoD, streams[i]);
  cudaMemcpyAsync(B_d+i*s_size, B_h+i*s_size, s_byte, HtoD, streams[i])
}
for (int i=0; i<nStreams; ++i)
{
  //kernel vector add
}
for (int i=0; i<nStreams; ++i)
{
  cudaMemcpyAsync(C_h+i*s_size, C_d+i*s_size, s_byte, DtoH, streams[i])
}
```

Figure 6: approache 2

4. **Implement Vector Add with 4 streams. Use nvprof to analyze the results. Write a couple of observations.**

- cudaStreamCreate and cudaStreamDestroy are very lightweight which will not give us a lot of overhead

```
==20444== Profiling application: ./vector_add
==20444== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   55.89%  20.713ms         4  5.1783ms  5.1314ms  5.2539ms  [CUDA memcpy HtoD]
                   43.26%  16.033ms         2  8.0166ms  7.7215ms  8.3118ms  [CUDA memcpy DtoH]
                    0.85%  315.05us         2  157.53us  153.22us  161.83us  VecAdd(int, float const
*, float const *, float*)
      API calls:   86.47%  318.96ms         3  106.32ms  317.26us  318.32ms  cudaMalloc
                   10.51%  38.784ms         6  6.4640ms  5.4058ms  8.6754ms  cudaMemcpyAsync
                    1.46%  5.3829ms       384  14.017us     293ns  634.65us  cuDeviceGetAttribute
                    0.98%  3.6113ms         3  1.2038ms  509.95us  1.5619ms  cudaFree
                    0.40%  1.4827ms         4  370.69us  366.78us  379.55us  cuDeviceTotalMem
                    0.12%  448.29us         4  112.07us  106.16us  129.60us  cuDeviceGetName
                    0.02%  91.988us         2  45.994us  15.354us  76.634us  cudaLaunchKernel
                    0.01%  37.682us         2  18.841us  10.391us  27.291us  cudaStreamCreate
                    0.01%  22.396us         2  11.198us  9.4020us  12.994us  cudaDeviceSynchronize
                    0.01%  20.694us         2  10.347us  3.0510us  17.643us  cudaStreamDestroy
                    0.01%  19.242us         4  4.8100us  3.4200us  8.2590us  cuDeviceGetPCIBusId
                    0.00%  4.0970us         8     512ns     302ns  1.2920us  cuDeviceGet
                    0.00%  3.0730us         3  1.0240us     272ns  2.0240us  cuDeviceGetCount
                    0.00%  1.6980us         4     424ns     354ns     560ns  cuDeviceGetUuid
ychen39@bg4:~/code/gpu/hw3-files>
```

Figure 7: 2 streams

```
==20400== Profiling application: ./vector_add
==20400== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.90%  19.579ms         8  2.4474ms  2.4046ms  2.4759ms  [CUDA memcpy HtoD]
                   42.16%  14.507ms         4  3.6269ms  3.4949ms  3.9704ms  [CUDA memcpy DtoH]
                    0.93%  320.30us         4  80.074us  76.802us  81.443us  VecAdd(int, float const
*, float const *, float*)
      API calls:   85.91%  299.67ms         3  99.891ms  303.98us  299.06ms  cudaMalloc
                   11.10%  38.714ms        12  3.2262ms  2.6462ms  4.7472ms  cudaMemcpyAsync
                    1.43%  4.9897ms       384  12.994us     228ns  598.09us  cuDeviceGetAttribute
                    1.03%  3.5808ms         3  1.1936ms  498.47us  1.5662ms  cudaFree
                    0.35%  1.2213ms         4  305.32us  302.82us  309.97us  cuDeviceTotalMem
                    0.11%  397.50us         4  99.375us  96.008us  108.93us  cuDeviceGetName
                    0.03%  111.49us         4  27.872us  11.442us  74.043us  cudaLaunchKernel
                    0.02%  60.551us         4  15.137us  8.4340us  32.349us  cudaStreamCreate
                    0.01%  26.090us         4  6.5220us  2.5440us  17.874us  cudaStreamDestroy
                    0.01%  19.868us         2  9.9340us  8.0050us  11.863us  cudaDeviceSynchronize
                    0.00%  12.982us         4  3.2450us  1.8880us  6.0720us  cuDeviceGetPCIBusId
                    0.00%  3.3900us         8     423ns     241ns  1.0050us  cuDeviceGet
                    0.00%  1.9680us         3     656ns     261ns  1.0640us  cuDeviceGetCount
                    0.00%  1.4050us         4     351ns     285ns     506ns  cuDeviceGetUuid
```

Figure 8: 4 streams

```
==20667== Dependency Analysis:
==20667== Analysis progress: 100%
Critical path(%)  Critical path  Waiting time  Name
          65.19%   302.997420ms           0ns  cudaMalloc
          24.63%   114.501427ms           0ns  <Other>
           4.63%    21.540644ms   14.706736ms  cudaMemcpyAsync
           3.16%    14.706736ms           0ns  [CUDA memcpy DtoH]
           1.23%     5.723398ms           0ns  cuDeviceGetAttribute
           0.77%     3.591335ms           0ns  cudaFree
           0.25%     1.167800ms           0ns  cuDeviceTotalMem_v2
           0.08%   392.694000us           0ns  cuDeviceGetName
           0.02%    76.099000us           0ns  VecAdd(int, float const *, float const *, float*)
           0.01%    55.785000us           0ns  cudaStreamCreate
           0.01%    25.915000us           0ns  cudaStreamDestroy_v5050
           0.00%    21.185000us           0ns  cudaDeviceSynchronize
           0.00%    11.927000us           0ns  cuDeviceGetPCIBusId
           0.00%     3.402000us           0ns  cuDeviceGet
           0.00%     1.748000us           0ns  cuDeviceGetCount
           0.00%     1.258000us           0ns  cuDeviceGetUuid
           0.00%            0ns           0ns  cudaLaunchKernel_v7000
           0.00%            0ns           0ns  [CUDA memcpy HtoD]
ychen39@bg4:~/code/gpu/hw3-files>
```

Figure 9: Dependency Analysis

- Increasing streams didn't decrease the total time of memcpy: $6 * 6.4640ms \approx 12 * 3.2262ms$
- Although we have increased the stream number, memcpy still is the bottleneck by dependency analysis