【CUDA 基础】4.5 使用统一内存的向量加法

2018-05-14 | CUDA , Freshman | 0 |

Abstract: 使用统一内存的CUDA程序——向量加法

Keywords: 统一内存，Uniform Memory

使用统一内存的向量加法

本文是前面关于统一内存的补充

参考：https://face2ai.com/CUDA-F-4-2-%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86/

统一内存矩阵加法

统一内存的基本思路就是减少指向同一个地址的指针，比如我们经常见到的，在本地分配内存，然后传输到设备，然后在从设备传输回来，使用统一内存，就没有这些显式的需求了，而是驱动程序帮我们完成。具体的做法就是：

```
CHECK(cudaMallocManaged((float**)&a_d,nByte));
CHECK(cudaMallocManaged((float**)&b_d,nByte));
CHECK(cudaMallocManaged((float**)&res_d,nByte));
```

使用cudaMallocManaged 来分配内存，这种内存在表面上看在设备和主机端都能访问，但是内部过程和我们前面手动copy过来copy过去是一样的，也就是memcopy是本质，而这个只是封装了一下。

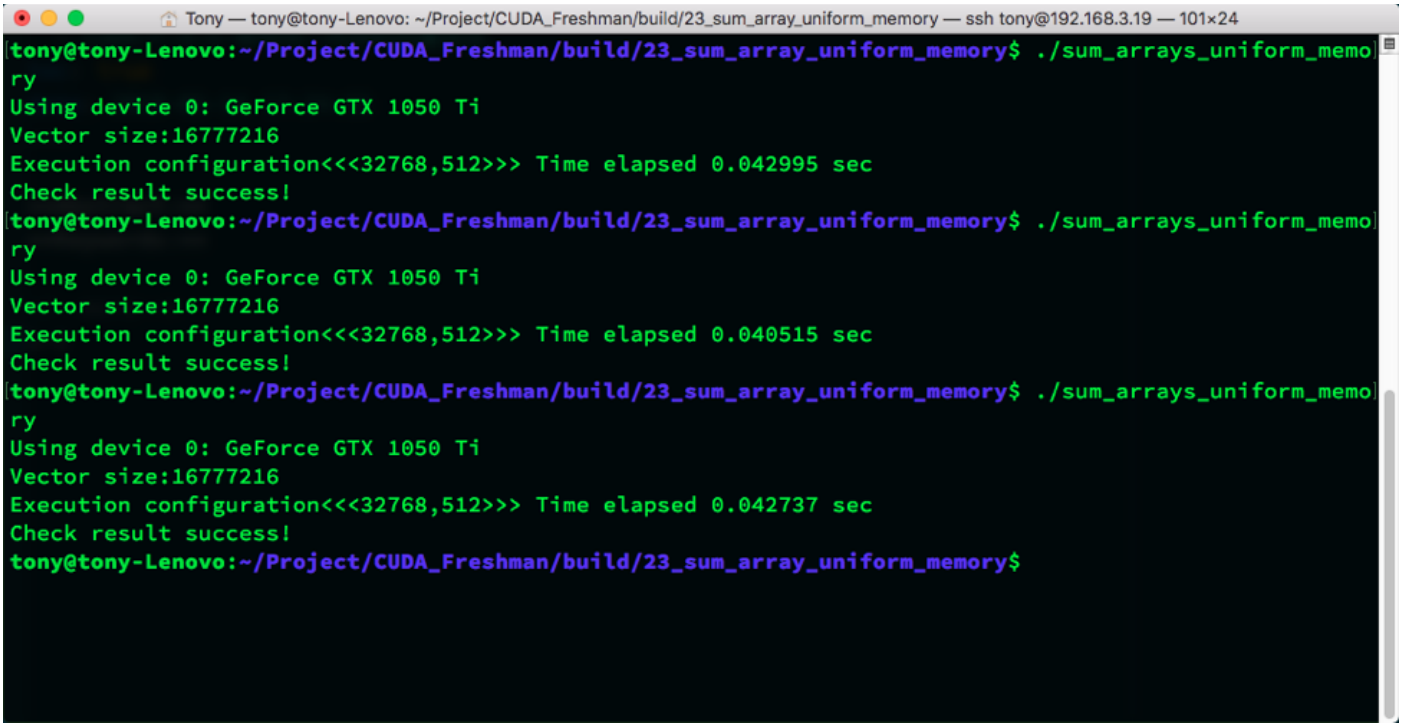
我们来看看完整的代码：

```
#include <cuda_runtime.h>
#include <stdio.h>
#include "freshman.h"



void sumArrays(float * a,float * b,float * res,const int size)
{
  for(int i=0;i<size;i+=4)
  {
    res[i]=a[i]+b[i];
    res[i+1]=a[i+1]+b[i+1];
    res[i+2]=a[i+2]+b[i+2];
    res[i+3]=a[i+3]+b[i+3];
  }
}
__global__ void sumArraysGPU(float*a,float*b,float*res,int N)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  if(i < N)
    res[i]=a[i]+b[i];
}
int main(int argc,char **argv)
{
  // set up device
  initDevice(0);

  int nElem=1<<24;
```

```
    printf("Vector size:%d\n",nElem);
    int nByte=sizeof(float)*nElem;
    float *res_h=(float*)malloc(nByte);
    memset(res_h,0,nByte);
    memset(res_from_gpu_h,0,nByte);

    float *a_d,*b_d,*res_d;
    CHECK(cudaMallocManaged((float**)&a_d,nByte));
    CHECK(cudaMallocManaged((float**)&b_d,nByte));
    CHECK(cudaMallocManaged((float**)&res_d,nByte));

    initialData(a_d,nElem);
    initialData(b_d,nElem);

    //CHECK(cudaMemcpy(a_d,a_h,nByte,cudaMemcpyHostToDevice));
    //CHECK(cudaMemcpy(b_d,b_h,nByte,cudaMemcpyHostToDevice));

    dim3 block(512);
    dim3 grid((nElem-1)/block.x+1);

    double iStart,iElaps;
    iStart=cpuSecond();
    sumArraysGPU<<<grid,block>>>(a_d,b_d,res_d,nElem);
    cudaDeviceSynchronize();
    iElaps=cpuSecond()-iStart;
    printf("Execution configuration<<<%d,%d>>> Time elapsed %f sec\n",grid.x,block.x,

    //CHECK(cudaMemcpy(res_from_gpu_h,res_d,nByte,cudaMemcpyDeviceToHost));
    sumArrays(b_d,b_d,res_h,nElem);

    checkResult(res_h,res_d,nElem);
    cudaFree(a_d);
    cudaFree(b_d);
    cudaFree(res_d);

    free(res_h);

    return 0;
}
```

注意我们注释掉的，这就是省去的代码部分、

运行结果：



就这个代码而言，使用统一内存还是手动控制，运行速度差不多。

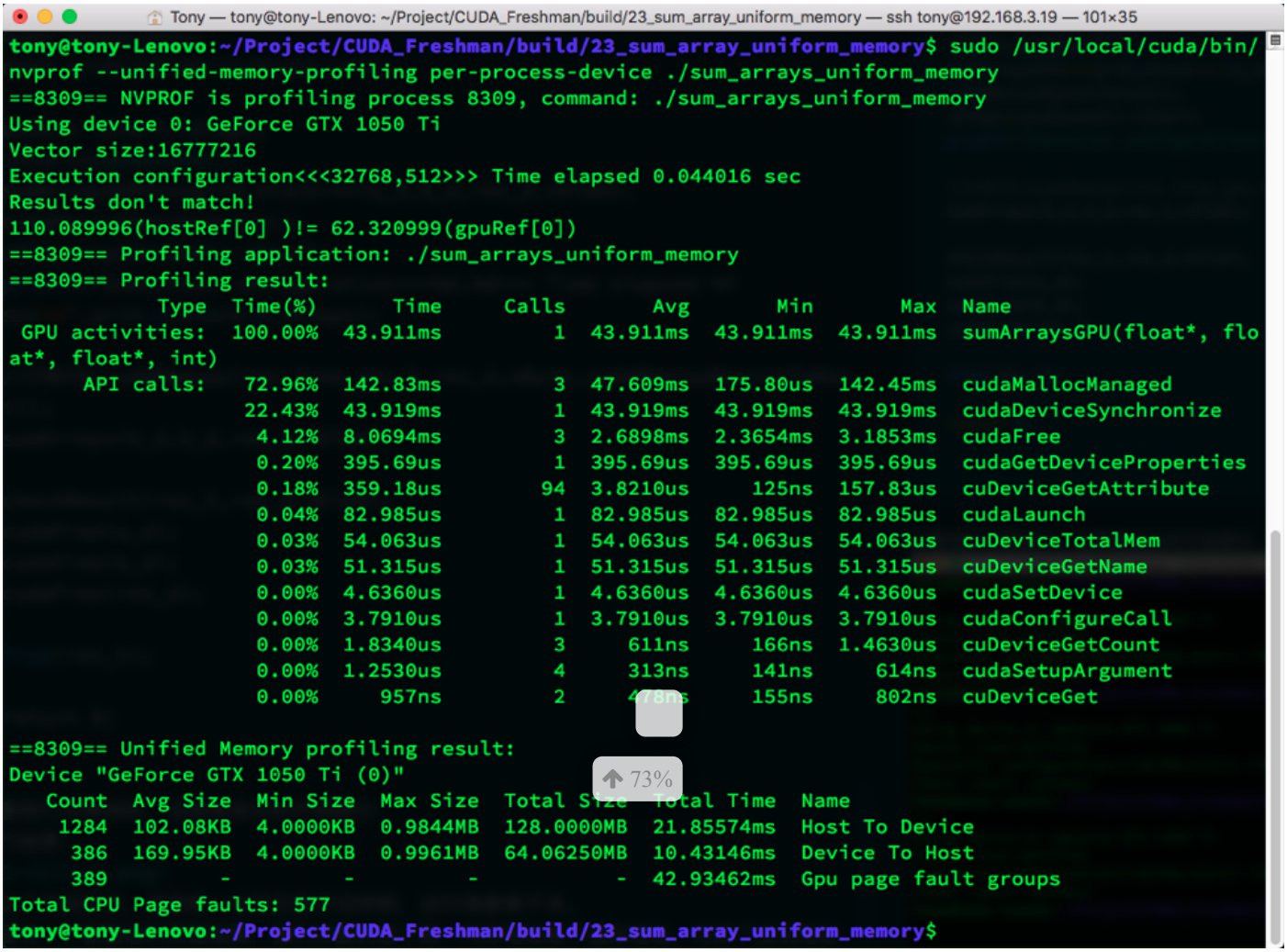
这里有一个新概念叫页面故障，我们分配的这个统一内存地址是个虚拟地址，对应了主机地址和GPU地址，当我们的主机访问这个虚拟地址的时候，会出现一个页面故障，当CPU要访问位于GPU上的托管内存时，统一内存使用CPU页面故障来出发设备到CPU的数据传输，这里的故障不是坏掉了，而是一种通信方式，类似于中断。

故障数和传输数据的大小直接相关。

使用

```
nvprof --unified-memory-profiling per-process-device ./sum_arrays_uniform_memory
```

可以查看到实际参数



也可以使用 nvvp来查看，效果类似。

总结

虽然统一内存管理给我们写代码带来了方便而且速度也很快，但是实验表明，手动控制还是要优于统一内存管理，换句话说，人脑的控制比编译器和目前的设备更有效，所以，为了效率，大家还是手动控制内存吧，把命运掌握在自己手里。

相关文章