【CUDA 基础】4.2 内存管理

2018-05-01 | CUDA , Freshman | 0 |

Abstract: 本文主要介绍CUDA内存管理，以及CUDA内存模型下的各种内存的特点。

Keywords: CUDA内存管理，CUDA内存分配和释放，CUDA内存传输，固定内存，零拷贝内存，统一虚拟寻址，统一内存寻址

内存管理

迷茫和困惑会影响我们的前进，彻底摆脱也许不太可能，但是我们必须肯定信仰的力量，专注你所热爱的，就会走出迷雾。

CUDA编程的目的是给我们的程序加速，尤其是机器学习，人工智能类的计算，CPU不能高效完成，说白

了，我们在控制硬件，控制硬件的语言属于底层语言，比如C语言，最头疼的就是管理内存，python，php这些语言有自己的内存管理机制，c语言的内存管理机制——程序员管理。这样的好处是学起来特别困难，但是学会了又会觉得特别爽，因为自由，你可以随意的控制计算机的计算过程。CUDA是C语言的扩展，内存方面基本集成了C语言的方式，由程序员控制CUDA内存，当然，这些内存的物理设备是在GPU上的，而且与CPU内存分配不同，CPU内存分配完就完事了，GPU还涉及到数据传输，主机和设备之间的传输。
接下来我们要了解的是：

- 分配释放设备内存
- 在主机和设备间传输内存

为达到最优性能，CUDA提供了在主机端准备设备内存的函数，并且显式地向设备传递数据，显式的从设备取回数据。

内存分配和释放

内存的分配和释放我们在前面已经用过很多次了，前面所有的要计算的例子都包含这一步：

```
cudaError_t cudaMalloc(void ** devPtr,size_t count)
```

这个函数用过很多次了，唯一要注意的是第一个参数，是指针的指针，一般的用法是首先我们生命一个指针变量，然后调用这个函数：

```
float * devMem=NULL;
cudaError_t cudaMalloc((float**) devMem, count)
```

这里是这样的，devMem是一个指针，定义时初始化指向NULL，这样做是安全的，避免出现野指针，cudaMalloc函数要修改devMem的值，所以必须把他的指针传递给函数，如果把devMem当做参数传递，经过函数后，指针的内容还是NULL。
不知道这个解释有没有听明白，通俗的讲，如果一个参数想要在函数中被修改，那么一定要传递他的地址给函数，如果只传递本身，函数是值传递的，不会改变参数的值。
内存分配支持所有的数据类型，什么int，float。。。这些都无所谓，因为他是按照字节分配的，只要是正数字节的变量都能分配，当然我们根本没有半个字节的东西。
函数执行失败返回：cudaErrorMemoryAllocation.
当分配完地址后，可以使用下面函数进行初始化：

```
cudaError_t cudaMemset(void * devPtr,int value,size_t count)
```

用法和Memset类似，但是注意，这些被我们操作的内存对应的物理内存都在GPU上。
当分配的内存不被使用时，使用下面语句释放程序。

```
cudaError_t cudaFree(void * devPtr)
```

注意这个参数一定是前面cudaMalloc类的函数（还有其他分配函数）分配到空间，如果输入非法指针参数，会返回 cudaErrorInvalidDevicePointer 错误，如果重复释放一个空间，也会报错。
目前为止，套路基本和C语言一致。但是，设备内存的分配和释放非常影响性能，所以，尽量重复利用！

内存传输

下面介绍点C语言没有的，C语言的内存分配完成后就可以直接读写了，但是对于异构计算，这样是不行的，因为主机线程不能访问设备内存，设备线程也不能访问主机内存，这时候我们要传送数据了：

```
cudaError_t cudaMemcpy(void *dst,const void * src,size_t count,enum cudaMemcpyKind k
```

这个函数我们前面也反复用到，注意这里的参数是指针，而不是指针的指针，第一个参数dst是目标地址，第二个参数src是原始地址，然后是拷贝的内存大小，最后是传输类型，传输类型包括以下几种：

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

四种方式，都写在字面上来，唯一有点问题的就是有个host 到host，不知道为啥存在，估计很多人跟我想法一样，可能后面有什么高级的用法。
这个例子也不用说了，前面随便找个有数据传输的都有这两步：从主机到设备，然后计算，最后从设备到主机。
代码省略，来张图：

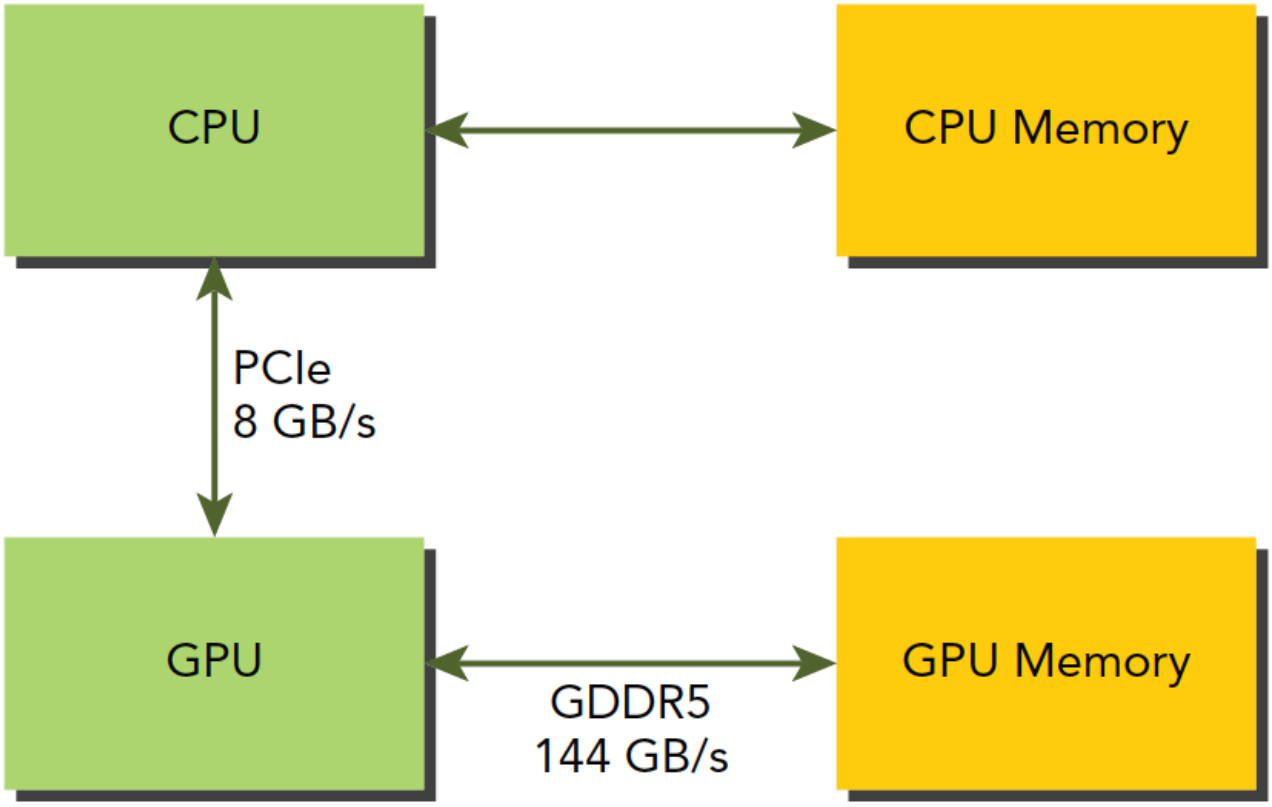


FIGURE 4-3

GPU的内存采用的DDR5制式，2011三星才做出来DDR4的主机内存，但是GPU却一直在使用DDR5，这个具体原因我也不清楚，有兴趣的同学自行去查询，但是我们要说的是GPU的内存理论峰值带宽非常高，对于Fermi C2050 有144GB/s，这个值估计现在的GPU应该都超过了，CPU和GPU之间通信要经过PCIe总线，总线的理论峰值要低很多——8GB/s左右，也就是说所，管理不当，算到半路需要从主机读数据，那效率瞬间全挂在PCIe上了。

CUDA编程需要大家减少主机和设备之间的内存传输。

固定内存

主机内存采用分页式管理，通俗的说法就是操作系统把物理内存分成一些“页”，然后给一个应用程序一大块内存，但是这一大块内存可能在一些不连续的页上，应用只能看到虚拟的内存地址，而操作系统可能随时更换物理地址的页（从原始地址复制到另一个地址）但是应用是不会差觉得，但是从主机传输到设备上的时候，如果此时发生了页面移动，对于传输操作来说是致命的，所以在数据传输之前，CUDA驱动会锁定页面，或者直接分配固定的主机内存，将主机源数据复制到固定内存上，然后从固定内存传输数据到设备上：

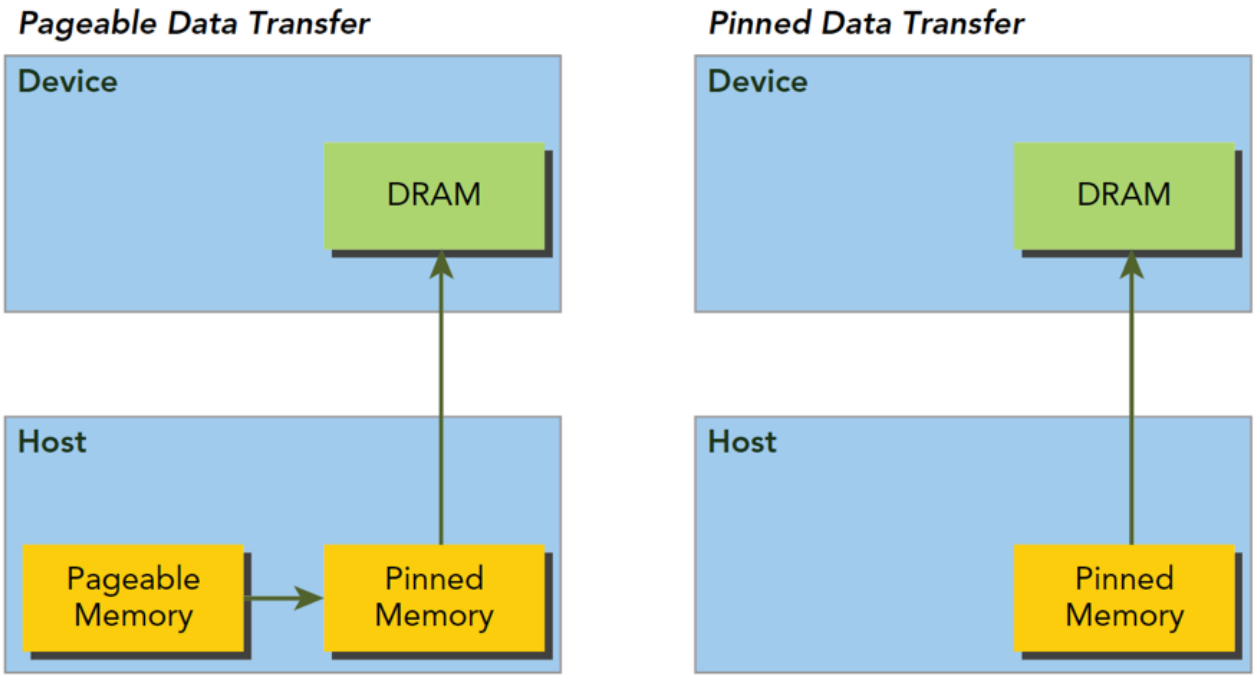


FIGURE 4-4

上图左边是正常分配内存，传输过程是：锁页-复制到固定内存-复制到设备

右边时分配时就是固定内存，直接传输到设备上。

下面函数用来分配固定内存：

```
cudaError_t cudaMallocHost(void ** devPtr,size_t count)
```

分配count字节的固定内存，这些内存是页面锁定的，可以直接传输到设备的（翻译的原文写的是：设备可访问的，英文原文是：Since the pinned memory can be accessed directly by the device。应该是翻译问题）这样就是的传输带宽变得高很多。

固定的主机内存释放使用：

```
cudaError_t cudaFreeHost(void *ptr)
```

我们可以测试一下固定内存和分页内存的传输效率，代码如下

```
#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 i=blockIdx.x*blockDim.x+threadIdx.x;
  res[i]=a[i]+b[i];
}
int main(int argc,char **argv)
{
  int dev = 0;
  cudaSetDevice(dev);

  int nElem=1<<14;
  printf("Vector size:%d\n",nElem);
  int nByte=sizeof(float)*nElem;
  float *a_h=(float*)malloc(nByte);
  float *b_h=(float*)malloc(nByte);
  float *res_h=(float*)malloc(nByte);
  float *res_from_gpu_h=(float*)malloc(nByte);
  memset(res_h,0,nByte);
  memset(res_from_gpu_h,0,nByte);

  float *a_d,*b_d,*res_d;
  // pine memory malloc
  CHECK(cudaMallocHost((float**)&a_d,nByte));
  CHECK(cudaMallocHost((float**)&b_d,nByte));
  CHECK(cudaMallocHost((float**)&res_d,nByte));

  initialData(a_h,nElem);
  initialData(b_h,nElem);

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

```

```
    dim3 block(1024);
    dim3 grid(nElem/block.x);
    sumArraysGPU<<<grid,block>>>(a_d,b_d,res_d);
    printf("Execution configuration<<<%d,%d>>>\n",grid.x,block.x);

    CHECK(cudaMemcpy(res_from_gpu_h,res_d,nByte,cudaMemcpyDeviceToHost));
    sumArrays(a_h,b_h,res_h,nElem);

    checkResult(res_h,res_from_gpu_h,nElem);
    cudaFreeHost(a_d);
    cudaFreeHost(b_d);
    cudaFreeHost(res_d);

    free(a_h);
    free(b_h);
    free(res_h);
    free(res_from_gpu_h);

    return 0;
}
```

注意这个核函数将会被本篇所有程序使用，今天的关键在于主机分配内存部分，所以核函数就选个最简单的。大家看看效率就好。

使用

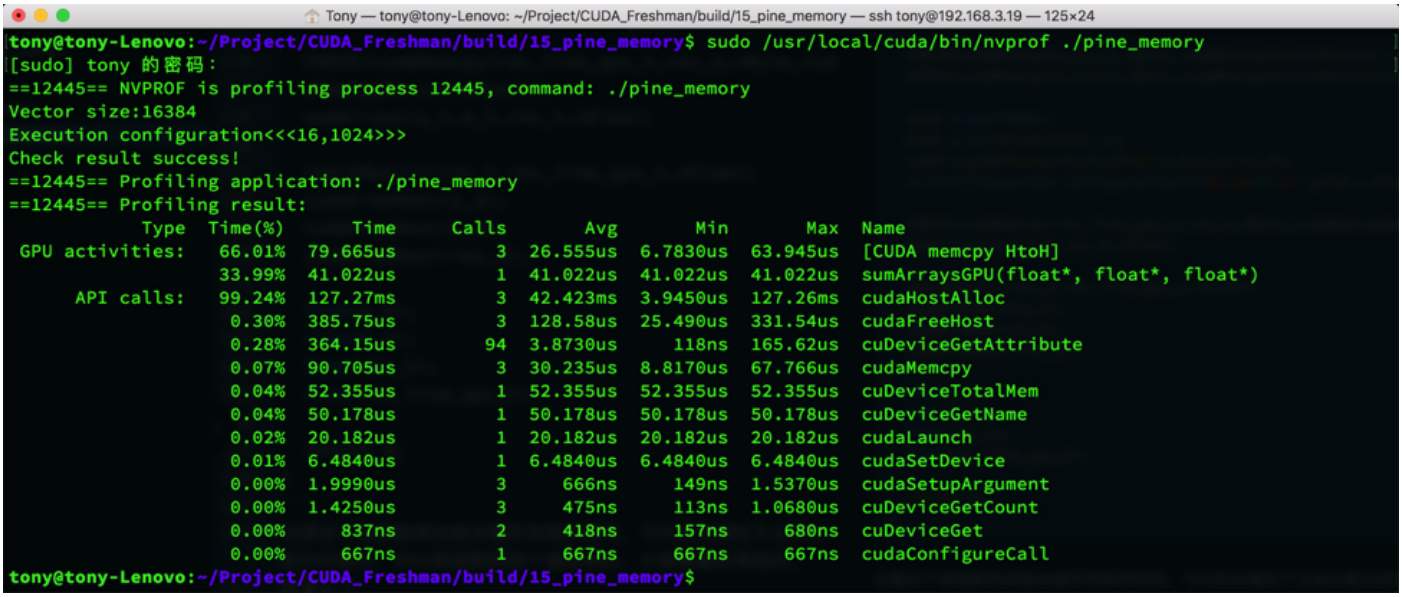
```
nvprof ./pine_memory
```

如果提示错误：

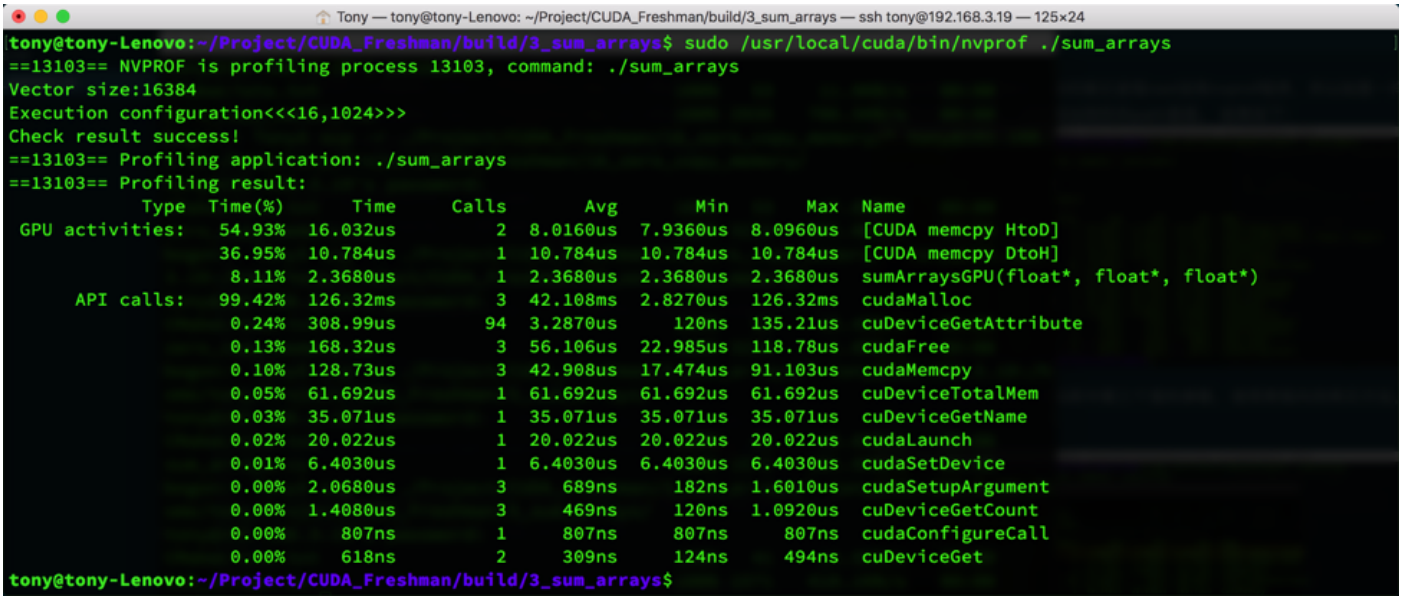
```
Error: CUDA profiling error.
```

可以改用root权限执行，这时候又发现root没有nvprof程序，所以如图一样，用完整路径执行就好，或者添加到你的path里面。

结果如下：



作为对比，我们改写了代码库中第三个里的参数，使用常规内存拷贝方法，得到的时间如下：



这个结果有点尴尬，固定内存的指标显示的是HtoH，也就是主机到主机的内存拷贝，而常规拷贝显示了HtoD。主机到设备但是看memcpy的速度能看出固定内存耗时确实少一些30:42。

同时也能看到cudaHostAlloc和cudaMalloc的时间接近，当数据增大的时候，这个就有区别了，cudaHostAlloc会慢很多。

结论：

固定内存的释放和分配成本比可分页内存要高很多，但是传输速度更快，所以对于大规模数据，固定内存效率更高。

尽量使用流来使内存传输和计算之间同时进行，第六章详细介绍这部分。

零拷贝内存

截止到目前，我们所接触到的内存知识的基础都是：主机直接不能访问设备内存，设备不能直接访问主机内存。对于早期设备，这是肯定的，但是后来，一个例外出现了——零拷贝内存。
GPU线程可以直接访问零拷贝内存，这部分内存在主机内存里面，CUDA核函数使用零拷贝内存有以下几种情况：

- 当设备内存不足的时候可以利用主机内存
- 避免主机和设备之间的显式内存传输
- 提高PCIe传输率

前面我们讲，注意线程之间的内存竞争，因为他们可以同时访问同一个内存地址，现在设备和主机可以同时访问同一个设备地址了，所以，我们要注意主机和设备的内存竞争——当使用零拷贝内存的时候。
零拷贝内存是固定内存，不可分页。可以通过以下函数创建零拷贝内存：

```
cudaError_t cudaHostAlloc(void ** pHost,size_t count,unsigned int flags)
```

最后一个标志参数，可以选择以下值：

- cudaHostAllocDefalt
- cudaHostAllocPortable
- cudaHostAllocWriteCombined
- cudaHostAllocMapped
  cudaHostAllocDefalt和cudaMallocHost函数一致，cudaHostAllocPortable函数返回能被所有CUDA上下文使用的固定内存，cudaHostAllocWriteCombined返回写结合内存，在某些设备上这种内存传输效率更高。cudaHostAllocMapped产生零拷贝内存。
  注意，零拷贝内存虽然不需要显式的传递到设备上，但是设备还不能通过pHost直接访问对应的内存地址，设备需要访问主机上的零拷贝内存，需要先获得另一个地址，这个地址帮助设备访问到主机对应的内存，方法是：

```
cudaError_t cudaHostGetDevicePointer(void ** pDevice,void * pHost,unsigned flag
```

pDevice就是设备上访问主机零拷贝内存的指针了！
此处flag必须设置为0，具体内容后面有介绍。

零拷贝内存可以当做比设备主存储器更慢的一个设备。

频繁的读写，零拷贝内存效率极低，这个非常容易理解，因为每次都要经过PCIe，千军万马堵在独木桥上，速度肯定慢，要是再有人来来回回走，那就更要命了。我们下面进行一个小实验，数组加法，改编自前面的代码，然后我们看看效果：

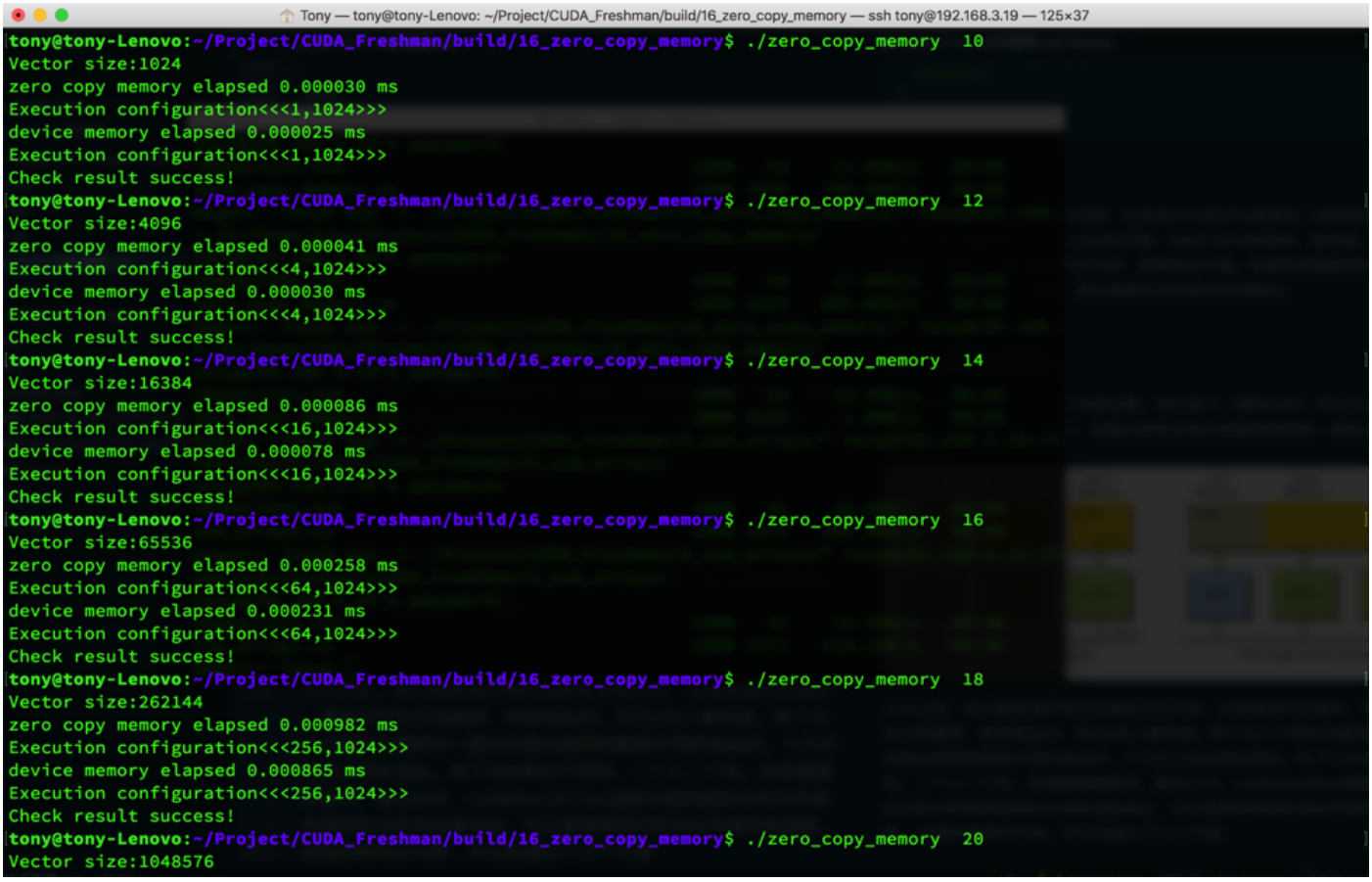
主函数代码，核函数如上节代码：

```
int main(int argc,char **argv)
{
  int dev = 0;
  cudaSetDevice(dev);
  int power=10;
  if(argc>=2)
    power=atoi(argv[1]);
  int nElem=1<<power;
  printf("Vector size:%d\n",nElem);
  int nByte=sizeof(float)*nElem;
  float *res_from_gpu_h=(float*)malloc(nByte);
  float *res_h=(float*)malloc(nByte);
  memset(res_h,0,nByte);
  memset(res_from_gpu_h,0,nByte);

  float *a_host,*b_host,*res_d;
  double iStart,iElaps;
  dim3 block(1024);
  dim3 grid(nElem/block.x);
  res_from_gpu_h=(float*)malloc(nByte);
  float *a_dev,*b_dev;
  CHECK(cudaHostAlloc((float**)&a_host,nByte,cudaHostAllocMapped));
  CHECK(cudaHostAlloc((float**)&b_host,nByte,cudaHostAllocMapped));
  CHECK(cudaMalloc((float**)&res_d,nByte));
  initialData(a_host,nElem);
  initialData(b_host,nElem);

 //=============================================================//
  iStart = cpuSecond();
  CHECK(cudaHostGetDevicePointer((void**)&a_dev,(void*) a_host,0));
  CHECK(cudaHostGetDevicePointer((void**)&b_dev,(void*) b_host,0));
  sumArraysGPU<<<grid,block>>>(a_dev,b_dev,res_d);
  CHECK(cudaMemcpy(res_from_gpu_h,res_d,nByte,cudaMemcpyDeviceToHost));
  iElaps = cpuSecond() - iStart;
 //=============================================================//
  printf("zero copy memory elapsed %lf ms \n", iElaps);
```

```
    printf("Execution configuration<<<%d,%d>>>\n",grid.x,block.x);
//-----------------------normal memory-----------------------------
    float *a_h_n=(float*)malloc(nByte);
    float *b_h_n=(float*)malloc(nByte);
    float *res_h_n=(float*)malloc(nByte);
    float *res_from_gpu_h_n=(float*)malloc(nByte);
    memset(res_h_n,0,nByte);
    memset(res_from_gpu_h_n,0,nByte);

    float *a_d_n,*b_d_n,*res_d_n;
    CHECK(cudaMalloc((float**)&a_d_n,nByte));
    CHECK(cudaMalloc((float**)&b_d_n,nByte));
    CHECK(cudaMalloc((float**)&res_d_n,nByte));

    initialData(a_h_n,nElem);
    initialData(b_h_n,nElem);
//=============================================================//
    iStart = cpuSecond();
    CHECK(cudaMemcpy(a_d_n,a_h_n,nByte,cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(b_d_n,b_h_n,nByte,cudaMemcpyHostToDevice));
    sumArraysGPU<<<grid,block>>>(a_d_n,b_d_n,res_d_n);
    CHECK(cudaMemcpy(res_from_gpu_h,res_d,nByte,cudaMemcpyDeviceToHost));
    iElaps = cpuSecond() - iStart;
//=============================================================//
    printf("device memory elapsed %lf ms \n", iElaps);
    printf("Execution configuration<<<%d,%d>>>\n",grid.x,block.x);
//-------------------------------------------------------------------

    sumArrays(a_host,b_host,res_h,nElem);
    checkResult(res_h,res_from_gpu_h,nElem);

    cudaFreeHost(a_host);
    cudaFreeHost(b_host);
    cudaFree(res_d);
    free(res_h);
    free(res_from_gpu_h);

    cudaFree(a_d_n);
    cudaFree(b_d_n);
    cudaFree(res_d_n);

    free(a_h_n);
    free(b_h_n);
    free(res_h_n);
```

```
81      free(res_from_gpu_h_n);
82      return 0;
83  }
```

结果：



我们把结果写在一个表里面：

数据规模n(2^n)	常规内存（us）	零拷贝内存（us）
10	2.5	3.0
12	3.0	4.1
14	7.8	8.6
16	23.1	25.8
18	86.5	98.2

20	290.9	310.5

这是通过观察运行时间得到的，当然也可以通过我们上面的nvprof得到内核执行时间：

数据规模n(2^n)	常规内存（us）	零拷贝内存（us）
10	1.088	4.257
12	1.056	8.00
14	1，920	24.578
16	4.544	86.63

直接上数据，图太多，没办法贴了，但是这种比较方法有点问题，因为零拷贝内存在执行内核的时候相当于还执行了内存传输工作，所以我觉得应该把内存传输也加上，那样看速度就基本差不多了，但是如果常规内存完成传输后可以重复利用，那又是另一回事了。

但是零拷贝内存也有例外的时候，比如当CPU和GPU继承在一起的时候，你别不信，我手里就有一个，Nvidia的平板，ARM+GPU的架构，这时候，他们的物理内存公用的，这时候零拷贝内存，效果相当不错。但是如果离散架构，主机和设备之间通过PCIe连接，那么零拷贝内存将会非常耗时。

统一虚拟寻址

设备架构2.0以后，Nvida又有新创意，他们搞了一套称为同一寻址方式（UVA）的内存机制，这样，设备内存和主机内存被映射到同一虚拟内存地址中。如图

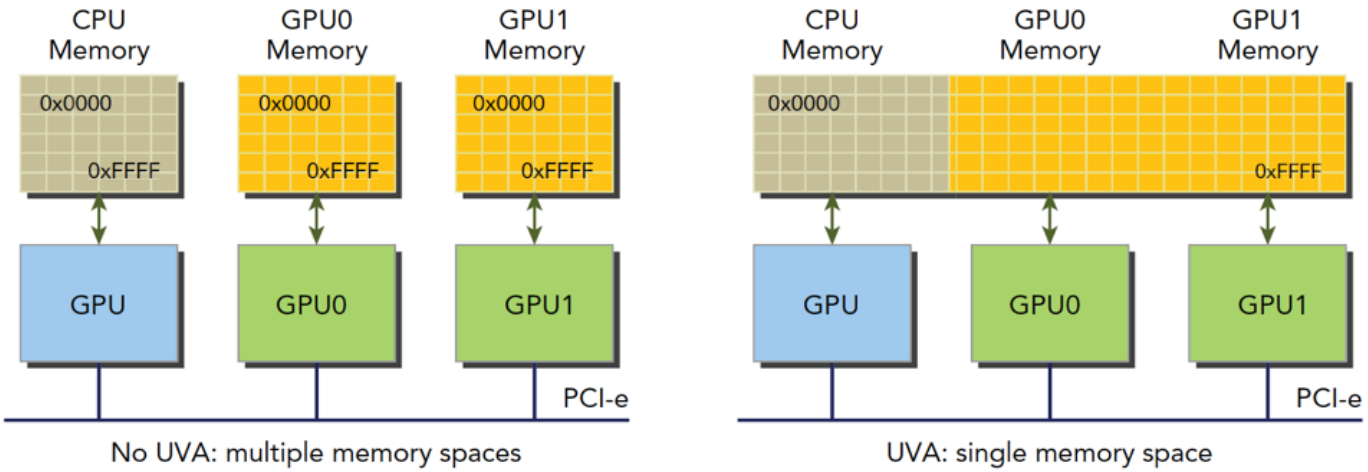


FIGURE 4-5

UVA之前，我们要管理所有的设备和主机内存，尤其是他们的指针，零拷贝内存尤其麻烦，很容易乱的，写过c的人都知道，弄个五六个指针在哪其中一部分还指向相同的数据不同的地址的，十几行之后必然会混乱。有了UVA再也不用怕，一个人一个名，走到哪里都能用，通过UVA，cudaHostAlloc函数分配的固定主机内存具有相同的主机和设备地址，可以直接将返回的地址传递给核函数。
前面的零拷贝内存，可以知道以下几个方面：

- 分配映射的固定主机内存
- 使用CUDA运行时函数获取映射到固定内存的设备指针
- 将设备指针传递给核函数

有了UVA，可以不用上面的那个获得设备上访问零拷贝内存的函数了：

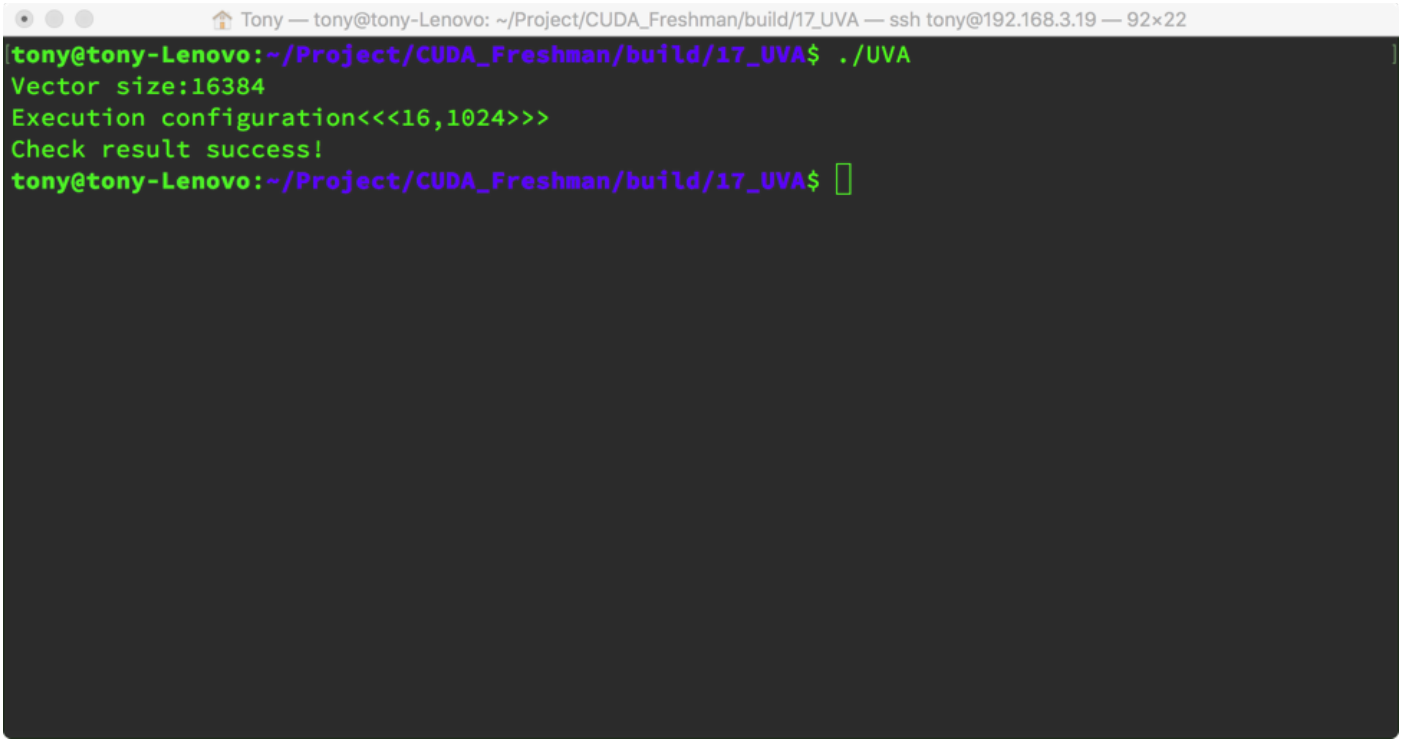
```
cudaError_t cudaHostGetDevicePointer(void ** pDevice,void * pHost,unsigned flags);
```

UVA来了以后，此函数基本失业了。
试验，代码：

```


    float *a_host,*b_host,*res_d;
    CHECK(cudaHostAlloc((float**)&a_host,nByte,cudaHostAllocMapped));
    CHECK(cudaHostAlloc((float**)&b_host,nByte,cudaHostAllocMapped));
    CHECK(cudaMalloc((float**)&res_d,nByte));
    res_from_gpu_h=(float*)malloc(nByte);

    initialData(a_host,nElem);
    initialData(b_host,nElem);

    dim3 block(1024);
    dim3 grid(nElem/block.x);
    sumArraysGPU<<<grid,block>>>(a_host,b_host,res_d);
}
```

UVA代码主要就是差个获取指针，UVA可以直接使用主机端的地址。

结果：



统一内存寻址

Nvidia的同志们还是不停的搞出新花样，CUDA6.0的时候又来了个统一内存寻址，注意不是同一虚拟寻址，提出的目的也是为了简化内存管理（我感觉是越简化越困难，因为套路多了）统一内存中创建一个托管内存池（CPU上有，GPU上也有），内存池中已分配的空间可以通过相同的指针直接被CPU和GPU访问，底层系统在统一的内存空间中自动的进行设备和主机间的传输。数据传输对应用是透明的，大大简化了代码。就是搞个内存池，这部分内存用一个指针同时表示主机和设备内存地址，依赖于UVA但是是完全不同的技术。

统一内存寻址提供了一个“指针到数据”的编程模型，概念上类似于零拷贝，但是零拷贝内存的分配是在主机上完成的，而且需要互相传输，但是统一寻址不同。

托管内存是指底层系统自动分配的统一内存，未托管内存就是我们自己分配的内存，这时候对于核函数，可以传递给他两种类型的内存，已托管和未托管内存，可以同时传递。

托管内存可以是静态的，也可以是动态的，添加 **managed** 关键字修饰托管内存变量。静态声明的托管内存作用域是文件，这一点可以注意一下。

托管内存分配方式：

```
1   cudaError_t cudaMallocManaged(void ** devPtr,size_t size,unsigned int flags=0)
```

这个函数和前面函数结构一致，注意函数名就好了，参数就不解释了，很明显了已经。