

moffis

随笔 - 281, 文章 - 8, 评论 - 6, 引用 - 0

导航

- 博客园
- 首页
- 新随笔
- 联系
- 订阅
- 管理

< 2019年5月 >						
日	一	二	三	四	五	六
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

公告

昵称: moffis
园龄: 4年1个月
粉丝: 15
关注: 6
+加关注

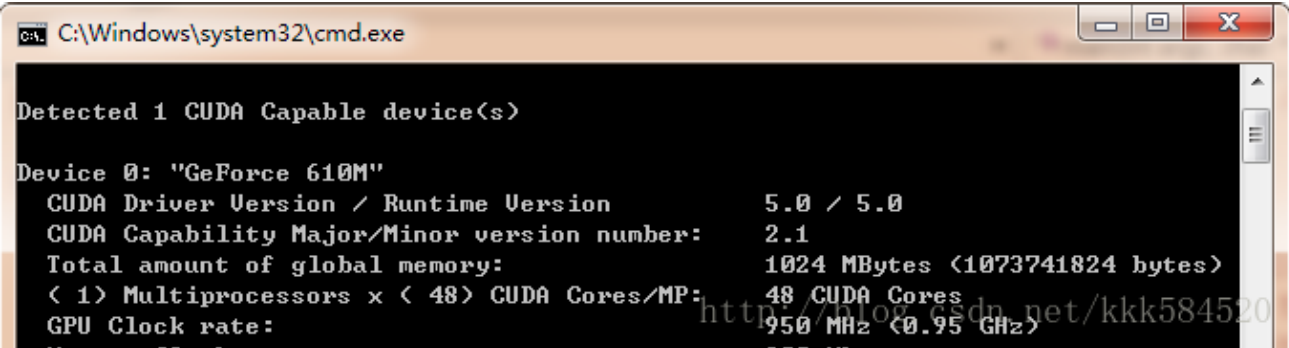
CUDA从入门到精通

转自<http://blog.csdn.net/kkk584520/article/details/9414191>

CUDA从入门到精通（一）：环境搭建

NVIDIA于2006年推出CUDA（Compute Unified Devices Architecture），可以利用其推出的GPU进行通用计算，将并行计算从大型集群扩展到了普通显卡，使得用户只需要一台带有Geforce显卡的笔记本就能跑较大规模的并行处理程序。

使用显卡的好处是，和大型集群相比功耗非常低，成本也不高，但性能很突出。以我的笔记本为例，Geforce 610M，用DeviceQuery程序测试，可得到如下硬件参数：



搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔档案

2015年11月 (2)
2015年10月 (8)
2015年9月 (11)
2015年8月 (14)
2015年7月 (38)
2015年6月 (109)
2015年5月 (30)
2015年4月 (32)
2015年3月 (19)
2015年2月 (1)
2014年12月 (15)
2014年11月 (2)

计算能力达 $48 \times 0.95 = 45.6$ GFLOPS。而笔记本的CPU参数如下：

系统

分级:	4.9 要求刷新 Windows 体验指数
处理器:	Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz 2.50 GHz
安装内存(RAM):	4.00 GB (3.85 GB 可用)
系统类型:	64 位操作系统
笔和触摸:	没有可用于此显示器的笔或触控输入

CPU计算能力为（4核）： $2.5G \times 4 = 10GFLOPS$ ，可见，显卡计算性能是4核i5 CPU的4~5倍，因此我们可以充分利用这一资源来对一些耗时的应用进行加速。

好了，工欲善其事必先利其器，为了使用CUDA对GPU进行编程，我们需要准备以下必备工具：

1. 硬件平台，就是显卡，如果你用的不是NVIDIA的显卡，那么只能说抱歉，其他都不支持CUDA。
2. 操作系统，我用过windows XP，Windows 7都没问题，本博客用Windows7。
3. C编译器，建议VS2008，和本博客一致。
4. CUDA编译器NVCC，可以免费免注册免license从官网下载CUDA Toolkit [CUDA下载](#)，最新版本为5.0，本博客用的就是该版本。
5. 其他工具（如Visual Assist，辅助代码高亮）

准备完毕，开始安装软件。VS2008安装比较费时间，建议安装完整版（NVIDIA官网说Express版也可

最新评论

1. Re:双边滤波器、高斯滤波
Ws(i,j)和Wr(i,j)的表达式指数是不是少了一个负号

--东东舒

2. Re:梯度下降与随机梯度下降

梯度下降法也是可以收敛到一个局部最小点的，因为梯度值会越来越小，它和固定的学习率相乘后的积也会越来越小。

这句话有问题吧！！

--视野

3. Re:第25章、
OnTouchListener触摸事件
(从零开始学Android)
ivwPicture.setOnTouchListener(new
PicOnTouchListener());
这一行中

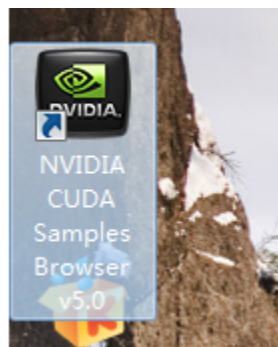
PicOnTouchListener老是显示红色名错误是什么原因呢

--ai06--hzh

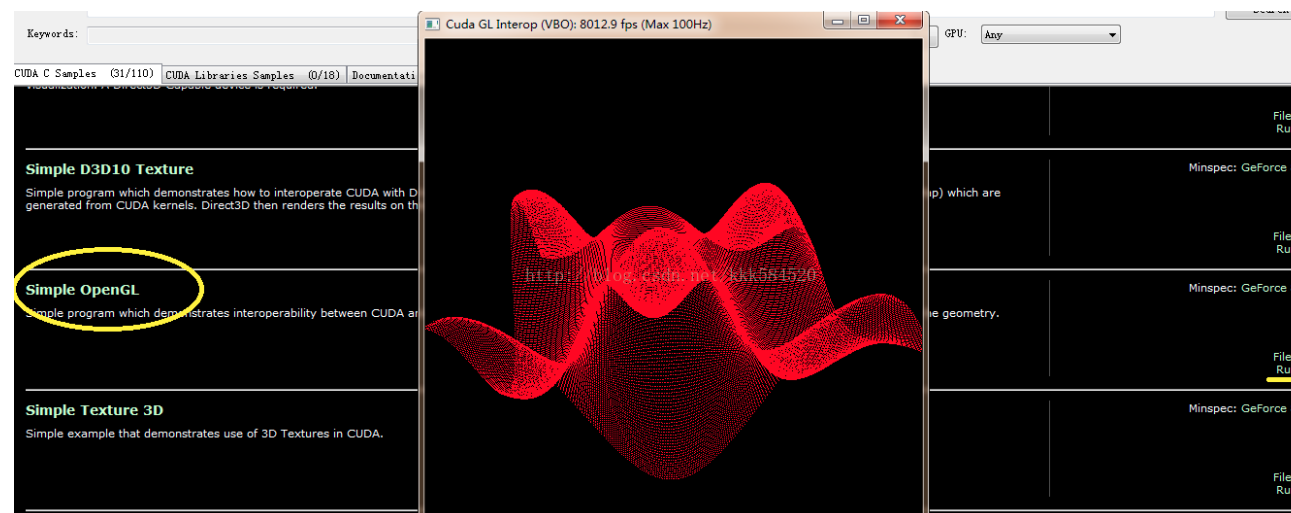
4. Re:Horn-Schunck光流法
你好。我现在要用Horn-Schunck光流法的强度和速度能不能给个demo呢

以)，过程不必详述。CUDA Toolkit 5.0里面包含了NVCC编译器、设计文档、设计例程、CUDA运行库、CUDA头文件等必备的原材料。

安装完毕，我们在桌面上发现这个图标：



不错，就是它，双击运行，可以看到一大堆例程。我们找到Simple OpenGL这个运行看看效果：



--sdsdd

5. Re:birch聚类算法

你好，请问你的代码是能够正确运行的是吗？

--qhl_finally

阅读排行榜

1. 梯度下降与随机梯度下降(16200)
2. 双边滤波器、高斯滤波(8576)
3. surf特征点检测(4701)
4. Horn-Schunck光流法(2722)
5. 粒子滤波(2331)

评论排行榜

1. 双边滤波器、高斯滤波(1)
2. 梯度下降与随机梯度下降(1)
3. 第25章、OnTouchListener触摸事件(从零开始学Android) (1)
4. Horn-Schunck光流法(1)
5. hough forest目标检测原理(1)

推荐排行榜

1. 在Activity中显示自定义View(1)

点右边黄线标记处的Run即可看到美妙的三维正弦曲面，鼠标左键拖动可以转换角度，右键拖动可以缩放。如果这个运行成功，说明你的环境基本搭建成功。

出现问题的可能：

1. 你使用远程桌面连接登录到另一台服务器，该服务器上有显卡支持CUDA，但你远程终端不能运行CUDA程序。这是因为远程登录使用的是你本地显卡资源，在远程登录时看不到服务器端的显卡，所以会报错：没有支持CUDA的显卡！解决方法：1. 远程服务器装两块显卡，一块只用于显示，另一块用于计算；2. 不要用图形界面登录，而是用命令行界面如telnet登录。
2. 有两个以上显卡都支持CUDA的情况，如何区分是在哪个显卡上运行？这个需要你在程序里控制，选择符合一定条件的显卡，如较高的时钟频率、较大的显存、较高的计算版本等。详细操作见后面的博客。

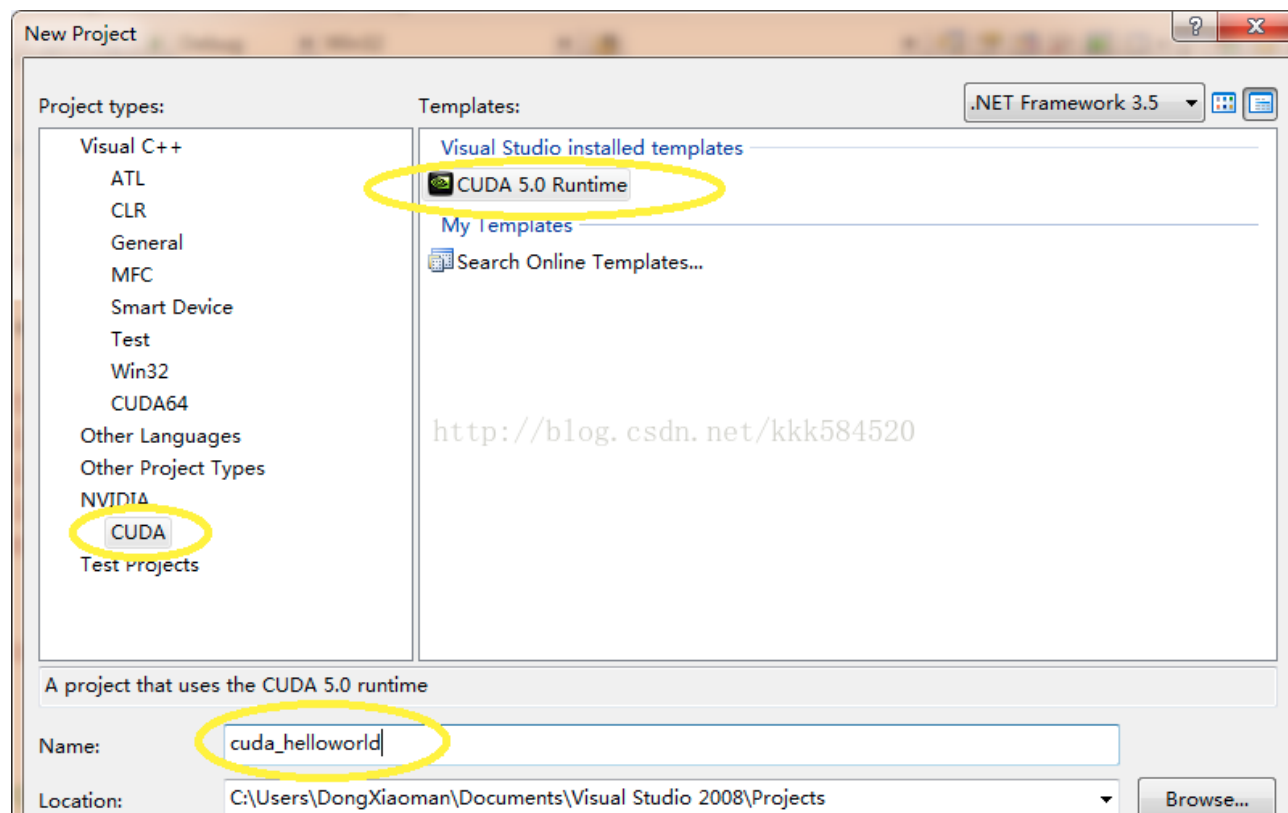
好了，先说这么多，下一节我们介绍如何在VS2008中给GPU编程。

CUDA从入门到精通（二）：第一个CUDA程序

书接上回，我们既然直接运行例程成功了，接下来就是了解如何实现例程中的每个环节。当然，我们先从简单的做起，一般编程语言都会找个helloworld例子，而我们的显卡是不会说话的，只能做一些简单的加减乘除运算。所以，CUDA程序的helloworld，我想应该最合适不过的就是向量加了。

打开VS2008，选择File->New->Project，弹出下面对话框，设置如下：

2. 双边滤波器、高斯滤波(1)
3. isodata聚类算法的实现(1)



之后点OK，直接进入工程界面。

工程中，我们看到只有一个.cu文件，内容如下：

```
[cpp] view plain copy print ?
01. #include "cuda_runtime.h"
02. #include "device_launch_parameters.h"
03.
04. #include <stdio.h>
05.
```

```
06.   cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size);
07.
08.   __global__ void addKernel(int *c, const int *a, const int *b)
09.   {
10.       int i = threadIdx.x;
11.       c[i] = a[i] + b[i];
12.   }
13.
14.   int main()
15.   {
16.       const int arraySize = 5;
17.       const int a[arraySize] = { 1, 2, 3, 4, 5 };
18.       const int b[arraySize] = { 10, 20, 30, 40, 50 };
19.       int c[arraySize] = { 0 };
20.
21.       // Add vectors in parallel.
22.       cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
23.       if (cudaStatus != cudaSuccess) {
24.           fprintf(stderr, "addWithCuda failed!");
25.           return 1;
26.       }
27.
28.       printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
29.             c[0], c[1], c[2], c[3], c[4]);
30.
31.       // cudaThreadExit must be called before exiting in order for profiling and
32.       // tracing tools such as Nsight and Visual Profiler to show complete traces.
33.       cudaStatus = cudaThreadExit();
34.       if (cudaStatus != cudaSuccess) {
35.           fprintf(stderr, "cudaThreadExit failed!");
36.           return 1;
37.       }
38.
39.       return 0;
40.   }
```

```
41.
42. // Helper function for using CUDA to add vectors in parallel.
43. cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size)
44. {
45.     int *dev_a = 0;
46.     int *dev_b = 0;
47.     int *dev_c = 0;
48.     cudaError_t cudaStatus;
49.
50.     // Choose which GPU to run on, change this on a multi-GPU system.
51.     cudaStatus = cudaSetDevice(0);
52.     if (cudaStatus != cudaSuccess) {
53.         fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-
54.         capable GPU installed?");
55.         goto Error;
56.     }
57.
58.     // Allocate GPU buffers for three vectors (two input, one output)
59.     cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
60.     if (cudaStatus != cudaSuccess) {
61.         fprintf(stderr, "cudaMalloc failed!");
62.         goto Error;
63.     }
64.
65.     cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
66.     if (cudaStatus != cudaSuccess) {
67.         fprintf(stderr, "cudaMalloc failed!");
68.         goto Error;
69.     }
70.
71.     cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
72.     if (cudaStatus != cudaSuccess) {
73.         fprintf(stderr, "cudaMalloc failed!");
74.         goto Error;
75.     }
```

```
75.  
76. // Copy input vectors from host memory to GPU buffers.  
77. cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);  
78. if (cudaStatus != cudaSuccess) {  
79.     fprintf(stderr, "cudaMemcpy failed!");  
80.     goto Error;  
81. }  
82.  
83. cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);  
84. if (cudaStatus != cudaSuccess) {  
85.     fprintf(stderr, "cudaMemcpy failed!");  
86.     goto Error;  
87. }  
88.  
89. // Launch a kernel on the GPU with one thread for each element.  
90. addKernel<<<1, size>>>(dev_c, dev_a, dev_b);  
91.  
92. // cudaThreadSynchronize waits for the kernel to finish, and returns  
93. // any errors encountered during the launch.  
94. cudaStatus = cudaThreadSynchronize();  
95. if (cudaStatus != cudaSuccess) {  
96.     fprintf(stderr, "cudaThreadSynchronize returned error code %d after launching a  
97.     ddKernel!\n", cudaStatus);  
98.     goto Error;  
99. }  
100. // Copy output vector from GPU buffer to host memory.  
101. cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);  
102. if (cudaStatus != cudaSuccess) {  
103.     fprintf(stderr, "cudaMemcpy failed!");  
104.     goto Error;  
105. }  
106.  
107. Error:  
108.     cudaFree(dev_c);
```



```
109.     cudaFree(dev_a);
110.     cudaFree(dev_b);
111.
112.     return cudaStatus;
113. }
```

可以看出，CUDA程序和C程序并无区别，只是多了一些以"cuda"开头的一些库函数和一个特殊声明的函数：

```
[cpp] view plain copy print ?
01.  __global__ void addKernel(int *c, const int *a, const int *b)
02.  {
03.      int i = threadIdx.x;
04.      c[i] = a[i] + b[i];
05.  }
```

这个函数就是在GPU上运行的函数，称之为核函数，英文名Kernel Function，注意要和操作系统内核函数区分开来。

我们直接按F7编译，可以得到如下输出：

```
[plain] view plain copy print ?
01.  1>----- Build started: Project: cuda_helloworld, Configuration: Debug Win32 -----
02.  1>Compiling with CUDA Build Rule...
03.  1>"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0
    \bin\nvcc.exe" -G -gencode=arch=compute_10,code=
    \"sm_10,compute_10\" -gencode=arch=compute_20,code=
```

```
\sm_20,compute_20\" --machine 32 -ccbin "C:\Program Files (x86)\Microsoft Visual Stud
io 9.0\VC\bin" -Xcompiler "/EHsc /W3 /nologo /O2 /Zi /MT " -I"C:\Program Files\N
VIDIA GPU Computing Toolkit\CUDA\v5.0\include" -maxrregcount=0 --compile -o "Debug
/kernel.cu.obj" kernel.cu
04. 1>tmpxft_000000ec_00000000-8_kernel.compute_10.cudafe1.gpu
05. 1>tmpxft_000000ec_00000000-14_kernel.compute_10.cudafe2.gpu
06. 1>tmpxft_000000ec_00000000-5_kernel.compute_20.cudafe1.gpu
07. 1>tmpxft_000000ec_00000000-17_kernel.compute_20.cudafe2.gpu
08. 1>kernel.cu
09. 1>kernel.cu
10. 1>tmpxft_000000ec_00000000-8_kernel.compute_10.cudafe1.cpp
11. 1>tmpxft_000000ec_00000000-24_kernel.compute_10.ii
12. 1>Linking...
13. 1>Embedding manifest...
14. 1>Performing Post-Build Event...
15. 1>copy "C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0
\\bin\cudart*.dll" "C:\Users\DongXiaoman\Documents\Visual Studio 2008\Projects
\cuda_helloworld\Debug"
16. 1>C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\\bin\cudart32_50_35.dll
17. 1>C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\\bin\cudart64_50_35.dll
18. 1>已复制 2 个文件。
19. 1>Build log was saved at "file:///c:/Users/DongXiaoman/Documents
\Visual Studio 2008\Projects\cuda_helloworld\cuda_helloworld\Debug\BuildLog.htm"
20. 1>cuda_helloworld - 0 error(s), 105 warning(s)
21. ===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

可见，编译.cu文件需要利用nvcc工具。该工具的详细使用见后面博客。

直接运行，可以得到结果图如下：

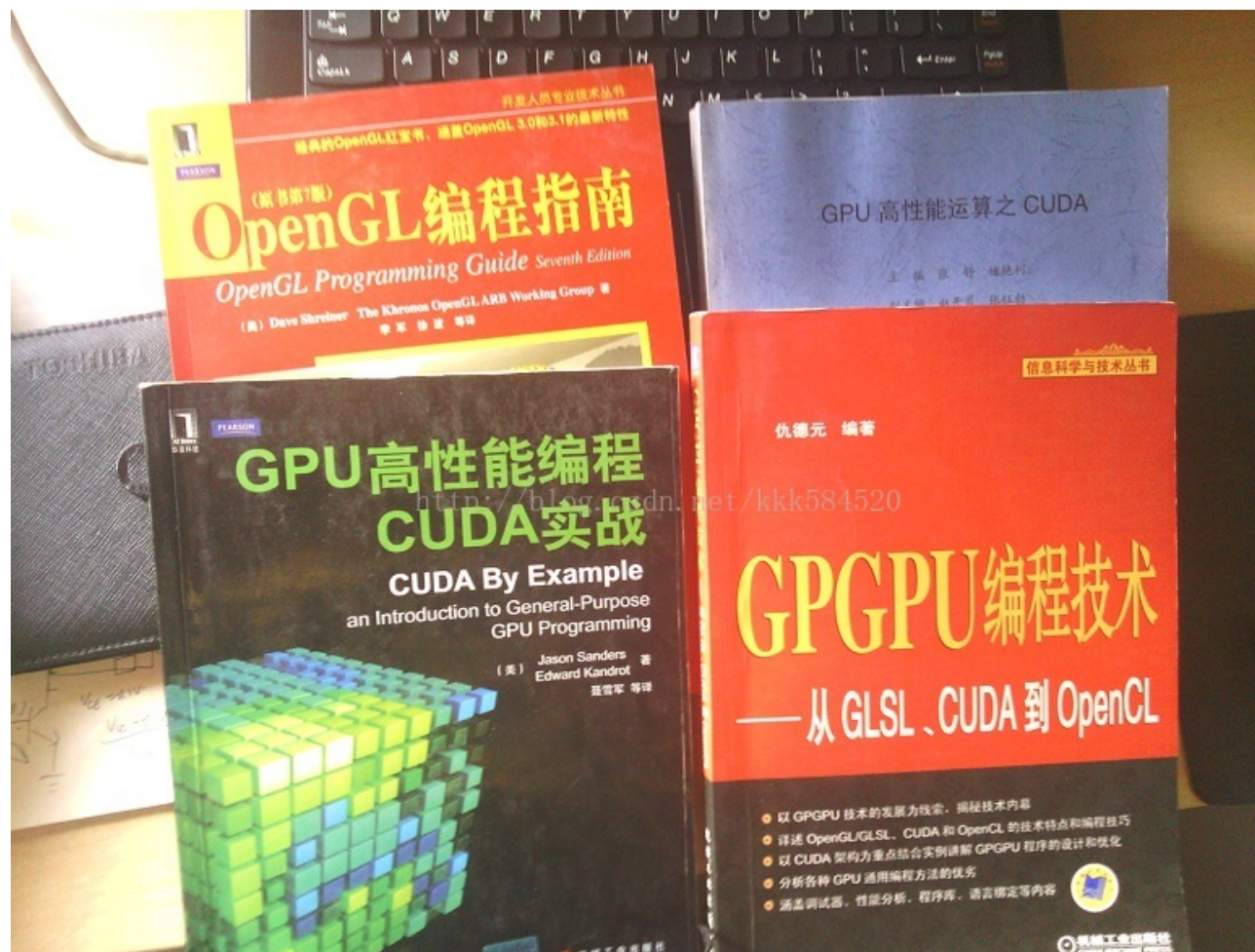
A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The first line shows a CUDA program output: $\{1, 2, 3, 4, 5\} + \{10, 20, 30, 40, 50\} = \{11, 22, 33, 44, 55\}$. The second line shows the prompt "请按任意键继续. . .". The third line shows a URL: <http://blog.csdn.net/kkk584520>. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

如果显示正确，那么我们的第一个程序宣告成功！

CUDA从入门到精通（三）：必备资料

刚入门CUDA，跑过几个官方提供的例程，看了看人家的代码，觉得并不难，但自己动手写代码时，总是不知道要先干什么，后干什么，也不知道从哪个知识点学起。这时就需要有一本能提供指导的书籍或者教程，一步步跟着做下去，直到真正掌握。

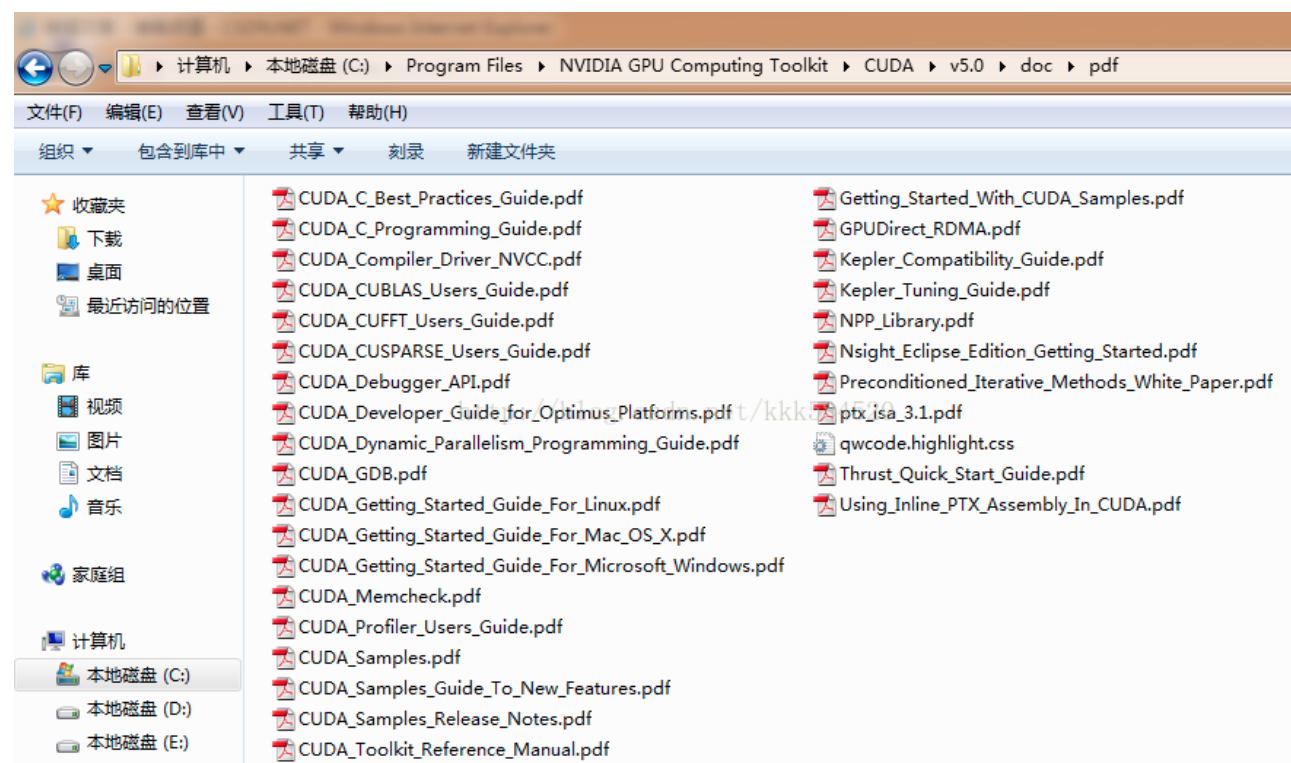
一般讲述CUDA的书，我认为不错的有下面这几本：



初学者可以先看美国人写的这本《GPU高性能编程CUDA实战》，可操作性很强，但不要期望能全看懂（Ps：里面有些概念其实我现在还是不怎么懂），但不影响你进一步学习。如果想更全面地学习CUDA，《GPGPU编程技术》比较客观详细地介绍了通用GPU编程的策略，看过这本书，可以对显卡有更深入的

了解，揭开GPU的神秘面纱。后面《OpenGL编程指南》完全是为了体验图形交互带来的乐趣，可以有选择地看；《GPU高性能运算之CUDA》这本是师兄给的，适合快速查询（感觉是将官方编程手册翻译了一遍）一些关键技术和概念。

有了这些指导材料还不够，我们在做项目的时候，遇到的问题在这些书上肯定找不到，所以还需要有下面这些利器：



这里面有很多工具的使用手册，如CUDA_GDB，Nsight，CUDA_Profiler等，方便调试程序；还有一些

有用的库，如CUFFT是专门用来做快速傅里叶变换的，CUBLAS是专用于线性代数（矩阵、向量计算）的，CUSPARSE是专用于稀疏矩阵表示和计算的库。这些库的使用可以降低我们设计算法的难度，提高开发效率。另外还有些入门教程也是值得一读的，你会对NVCC编译器有更近距离的接触。

好了，前言就这么多，本博主计划按如下顺序来讲述CUDA：

- 1.了解设备
- 2.线程并行
- 3.块并行
- 4.流并行
- 5.线程通信
- 6.线程通信实例：规约
- 7.存储模型
- 8.常数内存
- 9.纹理内存
- 10.主机页锁定内存
- 11.图形互操作
- 12.优化准则
- 13.CUDA与MATLAB接口

14.CUDA与MFC接口

CUDA从入门到精通（四）：加深对设备的认识

前面三节已经对CUDA做了一个简单的介绍，这一节开始真正进入编程环节。

首先，初学者应该对自己使用的设备有较为扎实的理解和掌握，这样对后面学习并行程序优化很有帮助，了解硬件详细参数可以通过上节介绍的几本书和官方资料获得，但如果仍然觉得不够直观，那么我们可以自己动手获得这些内容。

以第二节例程为模板，我们稍加改动的部分代码如下：

```
[cpp] view plain copy print ?
01.      // Add vectors in parallel.
02.      cudaError_t cudaStatus;
03.      int num = 0;
04.      cudaDeviceProp prop;
05.      cudaStatus = cudaGetDeviceCount(&num);
06.      for(int i = 0;i<num;i++)
07.      {
08.          cudaGetDeviceProperties(&prop,i);
09.      }
10.      cudaStatus = addWithCuda(c, a, b, arraySize);
```

这个改动的目的是让我们的程序自动通过调用cuda API函数获得设备数目和属性，所谓“知己知彼，百战不殆”。

cudaError_t 是cuda错误类型，取值为整数。

cudaDeviceProp为设备属性结构体，其定义可以从cuda Toolkit安装目录中找到，我的路径为：C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include\driver_types.h，找到定义为：

```
[cpp] view plain copy print ?
01.  /**
02.   * CUDA device properties
03.   */
04.  struct __device_builtin__ cudaDeviceProp
05.  {
06.      char    name[256];           /**< ASCII string identifying device */
07.      size_t  totalGlobalMem;      /**< Global memory available on device in bytes
08.   */
09.      size_t  sharedMemPerBlock;   /**< Shared memory available per block in bytes
10.   */
11.      int     regsPerBlock;        /**< 32-bit registers available per block */
12.      int     warpSize;           /**< Warp size in threads */
13.      size_t  memPitch;           /**< Maximum pitch in bytes allowed by memory co
14.   pies */
15.      int     maxThreadsPerBlock;  /**< Maximum number of threads per block */
16.      int     maxThreadsDim[3];    /**< Maximum size of each dimension of a block */
17.      /
18.      int     maxGridSize[3];     /**< Maximum size of each dimension of a grid */
19.
20.      int     clockRate;          /**< Clock frequency in kilohertz */
21.      size_t  totalConstMem;      /**< Constant memory available on device in byte
22.   s */
```



```

17.     int    major;                /**< Major compute capability */
18.     int    minor;                /**< Minor compute capability */
19.     size_t textureAlignment;      /**< Alignment requirement for textures */
20.     size_t texturePitchAlignment; /**< Pitch alignment requirement for texture ref
    erences bound to pitched memory */
21.     int    deviceOverlap;         /**< Device can concurrently copy memory and exe
    cute a kernel. Deprecated. Use instead asyncEngineCount. */
22.     int    multiProcessorCount;   /**< Number of multiprocessors on device */
23.     int    kernelExecTimeoutEnabled; /**< Specified whether there is a run time limit
    on kernels */
24.     int    integrated;            /**< Device is integrated as opposed to discrete
    */
25.     int    canMapHostMemory;      /**< Device can map host memory with cudaHostAll
    oc/cudaHostGetDevicePointer */
26.     int    computeMode;           /**< Compute mode (See ::cudaComputeMode) */
27.     int    maxTexture1D;          /**< Maximum 1D texture size */
28.     int    maxTexture1DMipmap;    /**< Maximum 1D mipmapped texture size */
29.     int    maxTexture1DLinear;    /**< Maximum size for 1D textures bound to linea
    r memory */
30.     int    maxTexture2D[2];       /**< Maximum 2D texture dimensions */
31.     int    maxTexture2DMipmap[2]; /**< Maximum 2D mipmapped texture dimensions */
32.     int    maxTexture2DLinear[3]; /**< Maximum dimensions (width, height, pitch) f
    or 2D textures bound to pitched memory */
33.     int    maxTexture2DGather[2]; /**< Maximum 2D texture dimensions if texture ga
    ther operations have to be performed */
34.     int    maxTexture3D[3];       /**< Maximum 3D texture dimensions */
35.     int    maxTextureCubemap;     /**< Maximum Cubemap texture dimensions */
36.     int    maxTexture1DLayered[2]; /**< Maximum 1D layered texture dimensions */
37.     int    maxTexture2DLayered[3]; /**< Maximum 2D layered texture dimensions */
38.     int    maxTextureCubemapLayered[2]; /**< Maximum Cubemap layered texture dimensions
    */
39.     int    maxSurface1D;          /**< Maximum 1D surface size */
40.     int    maxSurface2D[2];       /**< Maximum 2D surface dimensions */
41.     int    maxSurface3D[3];       /**< Maximum 3D surface dimensions */

```

```

42.     int    maxSurface1DLayered[2];    /**< Maximum 1D layered surface dimensions */
43.     int    maxSurface2DLayered[3];    /**< Maximum 2D layered surface dimensions */
44.     int    maxSurfaceCubemap;         /**< Maximum Cubemap surface dimensions */
45.     int    maxSurfaceCubemapLayered[2]; /**< Maximum Cubemap layered surface dimensions
    */
46.     size_t surfaceAlignment;          /**< Alignment requirements for surfaces */
47.     int    concurrentKernels;         /**< Device can possibly execute multiple kernel
    s concurrently */
48.     int    ECCEnabled;                /**< Device has ECC support enabled */
49.     int    pciBusID;                  /**< PCI bus ID of the device */
50.     int    pciDeviceID;               /**< PCI device ID of the device */
51.     int    pciDomainID;               /**< PCI domain ID of the device */
52.     int    tccDriver;                 /**< 1 if device is a Tesla device using TCC dri
    ver, 0 otherwise */
53.     int    asyncEngineCount;          /**< Number of asynchronous engines */
54.     int    unifiedAddressing;         /**< Device shares a unified address space with
    the host */
55.     int    memoryClockRate;           /**< Peak memory clock frequency in kilohertz */

56.     int    memoryBusWidth;            /**< Global memory bus width in bits */
57.     int    l2CacheSize;               /**< Size of L2 cache in bytes */
58.     int    maxThreadsPerMultiProcessor; /**< Maximum resident threads per multiprocessor
    */
59. };

```

后面的注释已经说明了其字段代表意义，可能有些术语对于初学者理解起来还是有一定困难，没关系，我们现在只需要关注以下几个指标：

name：就是设备名称；

totalGlobalMem：就是显存大小；

major,minor: CUDA设备版本号, 有1.1, 1.2, 1.3, 2.0, 2.1等多个版本;

clockRate: GPU时钟频率;

multiProcessorCount: GPU大核数, 一个大核 (专业点称为流多处理器, SM, Stream-Multiprocessor) 包含多个小核 (流处理器, SP, Stream-Processor)

编译, 运行, 我们在VS2008工程的cudaGetDeviceProperties()函数处放一个断点, 单步执行这一函数, 然后用Watch窗口, 切换到Auto页, 展开+, 在我的笔记本上得到如下结果:

Autos			
Name	Value	Type	
&prop	0x0034f998 {name=0x0034f998 "GeForce 610M"	cudaDeviceProp *	
name	0x0034f998 "GeForce 610M"	char [256]	
totalGlobalMem	1073741824	unsigned int	
sharedMemPerBlock	49152	unsigned int	
regsPerBlock	32768	int	
warpSize	32	int	
memPitch	2147483647	unsigned int	
maxThreadsPerBlock	1024	int	
maxThreadsDim	0x0034fab0	int [3]	
[0]	1024	int	
[1]	1024	int	
[2]	64	int	
maxGridSize	0x0034fab0	int [3]	
[0]	65535	int	
[1]	65535	int	
[2]	65535	int	
clockRate	950000	int	
totalConstMem	65536	unsigned int	
major	2	int	
minor	1	int	
textureAlignment	512	unsigned int	
texturePitchAlignment	32	unsigned int	
deviceOverlap	1	int	
multiProcessorCount	1	int	
kernelExecTimeoutEnabled	1	int	

可以看到，设备名为GeForce 610M，显存1GB，设备版本2.1（比较高端了，哈哈），时钟频率为950MHz（注意950000单位为kHz），大核数为1。在一些高性能GPU上（如Tesla，Kepler系列），大核数可能达到几十甚至上百，可以做更大规模的并行处理。

PS：今天看SDK代码时发现在help_cuda.h中有个函数实现从CUDA设备版本查询相应大核中小核的数

目, 觉得很有用, 以后编程序可以借鉴, 摘抄如下:

```
[cpp] view plain copy print ?
01. // Beginning of GPU Architecture definitions
02. inline int _ConvertSMVer2Cores(int major, int minor)
03. {
04.     // Defines for GPU Architecture types (using the SM version to determine the # of c
    ores per SM
05.     typedef struct
06.     {
07.         int SM; // 0xMm (hexidecimal notation), M = SM Major version, and m = SM minor
    version
08.         int Cores;
09.     } sSMtoCores;
10.
11.     sSMtoCores nGpuArchCoresPerSM[] =
12.     {
13.         { 0x10, 8 }, // Tesla Generation (SM 1.0) G80 class
14.         { 0x11, 8 }, // Tesla Generation (SM 1.1) G8x class
15.         { 0x12, 8 }, // Tesla Generation (SM 1.2) G9x class
16.         { 0x13, 8 }, // Tesla Generation (SM 1.3) GT200 class
17.         { 0x20, 32 }, // Fermi Generation (SM 2.0) GF100 class
18.         { 0x21, 48 }, // Fermi Generation (SM 2.1) GF10x class
19.         { 0x30, 192 }, // Kepler Generation (SM 3.0) GK10x class
20.         { 0x35, 192 }, // Kepler Generation (SM 3.5) GK11x class
21.         { -1, -1 }
22.     };
23.
24.     int index = 0;
25.
26.     while (nGpuArchCoresPerSM[index].SM != -1)
27.     {
28.         if (nGpuArchCoresPerSM[index].SM == ((major << 4) + minor))
29.         {
```

```
30.         return nGpuArchCoresPerSM[index].Cores;
31.     }
32.
33.     index++;
34. }
35.
36.     // If we don't find the values, we default use the previous one to run properly
37.     printf("MapSMtoCores for SM %d.%d is undefined. Default to use %d Cores/SM\n", maj
or, minor, nGpuArchCoresPerSM[7].Cores);
38.     return nGpuArchCoresPerSM[7].Cores;
39. }
40. // end of GPU Architecture definitions
```

可见，设备版本2.1的一个大核有48个小核，而版本3.0以上的一个大核有192个小核！

前文说到过，当我们用的电脑上有多个显卡支持CUDA时，怎么来区分在哪个上运行呢？这里我们看一下addWithCuda这个函数是怎么做的。

```
[cpp] view plain copy print ?
01.     cudaError_t cudaStatus;
02.
03.     // Choose which GPU to run on, change this on a multi-GPU system.
04.     cudaStatus = cudaSetDevice(0);
05.     if (cudaStatus != cudaSuccess) {
06.         fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-
capable GPU installed?");
07.         goto Error;
08.     }
```

使用了`cudaSetDevice(0)`这个操作，0表示能搜索到的第一个设备号，如果有多个设备，则编号为0,1,2...。

再看我们本节添加的代码，有个函数`cudaGetDeviceCount(&num)`，这个函数用来获取设备总数，这样我们选择运行CUDA程序的设备号取值就是0,1,...num-1，于是可以一个个枚举设备，利用`cudaGetDeviceProperties(&prop)`获得其属性,然后利用一定排序、筛选算法，找到最符合我们应用的那个设备号opt，然后调用`cudaSetDevice(opt)`即可选择该设备。选择标准可以从处理能力、版本控制、名称等各个角度出发。后面讲述流并发过程时，还要用到这些API。

如果希望了解更多硬件内容可以结合<http://www.geforce.cn/hardware>获取。

CUDA从入门到精通（五）：线程并行

多线程我们应该都不陌生，在操作系统中，进程是资源分配的基本单元，而线程是CPU时间调度的基本单元（这里假设只有1个CPU）。

将线程的概念引申到CUDA程序设计中，我们可以认为线程就是执行CUDA程序的最小单元，前面我们建立的工程代码中，有个核函数概念不知各位童鞋还记得没有，在GPU上每个线程都会运行一次该核函数。

但GPU上的线程调度方式与CPU有很大不同。CPU上会有优先级分配，从高到低，同样优先级的可以采用时间片轮转法实现线程调度。GPU上线程没有优先级概念，所有线程机会均等，线程状态只有等待资源和执行两种状态，如果资源未就绪，那么就等待；一旦就绪，立即执行。当GPU资源很充裕时，所有

线程都是并发执行的，这样加速效果很接近理论加速比；而GPU资源少于总线程个数时，有一部分线程就会等待前面执行的线程释放资源，从而变为串行化执行。

代码还是用上一节的吧，改动很少，再贴一遍：

```
[cpp] view plain copy print ?
01. #include "cuda_runtime.h"           //CUDA运行时API
02. #include "device_launch_parameters.h"
03. #include <stdio.h>
04. cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size);
05. __global__ void addKernel(int *c, const int *a, const int *b)
06. {
07.     int i = threadIdx.x;
08.     c[i] = a[i] + b[i];
09. }
10. int main()
11. {
12.     const int arraySize = 5;
13.     const int a[arraySize] = { 1, 2, 3, 4, 5 };
14.     const int b[arraySize] = { 10, 20, 30, 40, 50 };
15.     int c[arraySize] = { 0 };
16.     // Add vectors in parallel.
17.     cudaError_t cudaStatus;
18.     int num = 0;
19.     cudaDeviceProp prop;
20.     cudaStatus = cudaGetDeviceCount(&num);
21.     for(int i = 0; i < num; i++)
22.     {
23.         cudaGetDeviceProperties(&prop, i);
24.     }
25.     cudaStatus = addWithCuda(c, a, b, arraySize);
```



```
26.     if (cudaStatus != cudaSuccess)
27.     {
28.         fprintf(stderr, "addWithCuda failed!");
29.         return 1;
30.     }
31.     printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);
32.     // cudaThreadExit must be called before exiting in order for profiling and
33.     // tracing tools such as Nsight and Visual Profiler to show complete traces.
34.     cudaStatus = cudaThreadExit();
35.     if (cudaStatus != cudaSuccess)
36.     {
37.         fprintf(stderr, "cudaThreadExit failed!");
38.         return 1;
39.     }
40.     return 0;
41. }
42. // 重点理解这个函数
43. cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size)
44. {
45.     int *dev_a = 0; //GPU设备端数据指针
46.     int *dev_b = 0;
47.     int *dev_c = 0;
48.     cudaError_t cudaStatus;      //状态指示
49.
50.     // Choose which GPU to run on, change this on a multi-GPU system.
51.     cudaStatus = cudaSetDevice(0); //选择运行平台
52.     if (cudaStatus != cudaSuccess)
53.     {
54.         fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-
55.         capable GPU installed?");
56.         goto Error;
57.     }
58.     // 分配GPU设备端内存
59.     cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
```

```
59.     if (cudaStatus != cudaSuccess)
60.     {
61.         fprintf(stderr, "cudaMalloc failed!");
62.         goto Error;
63.     }
64.     cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
65.     if (cudaStatus != cudaSuccess)
66.     {
67.         fprintf(stderr, "cudaMalloc failed!");
68.         goto Error;
69.     }
70.     cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
71.     if (cudaStatus != cudaSuccess)
72.     {
73.         fprintf(stderr, "cudaMalloc failed!");
74.         goto Error;
75.     }
76.     // 拷贝数据到GPU
77.     cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
78.     if (cudaStatus != cudaSuccess)
79.     {
80.         fprintf(stderr, "cudaMemcpy failed!");
81.         goto Error;
82.     }
83.     cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
84.     if (cudaStatus != cudaSuccess)
85.     {
86.         fprintf(stderr, "cudaMemcpy failed!");
87.         goto Error;
88.     }
89.     // 运行核函数
90.     <span style="BACKGROUND-COLOR: #ff6666"><strong>    addKernel<<<1, size>>>
      (dev_c, dev_a, dev_b);</strong></span>
91.     </span>    // cudaThreadSynchronize waits for the kernel to finish, and returns
92.    // any errors encountered during the launch.
```

```
93.         cudaStatus = cudaThreadSynchronize();    //同步线程
94.         if (cudaStatus != cudaSuccess)
95.         {
96.             fprintf(stderr, "cudaThreadSynchronize returned error code %d after launching a
ddKernel!\n", cudaStatus);
97.             goto Error;
98.         }
99.         // Copy output vector from GPU buffer to host memory.
100.        cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
//拷贝结果回主机
101.        if (cudaStatus != cudaSuccess)
102.        {
103.            fprintf(stderr, "cudaMemcpy failed!");
104.            goto Error;
105.        }
106.    Error:
107.        cudaFree(dev_c);    //释放GPU设备端内存
108.        cudaFree(dev_a);
109.        cudaFree(dev_b);
110.        return cudaStatus;
111.    }
```

红色部分即启动核函数的调用过程，这里看到调用方式和C不太一样。<<<>>>表示运行时配置符号，里面1表示只分配一个线程组（又称线程块、Block），size表示每个线程组有size个线程（Thread）。本程序中size根据前面传递参数个数应该为5，所以运行的时候，核函数在5个GPU线程单元上分别运行了一次，总共运行了5次。这5个线程是如何知道自己“身份”的？是靠threadIdx这个内置变量，它是个dim3类型变量，接受<<<>>>中第二个参数，它包含x,y,z 3维坐标，而我们传入的参数只有一维，所以只有x值是有效的。通过核函数中int i = threadIdx.x;这一句，每个线程可以获得自身的id号，从而找到自己的任务去执行。

下节我们介绍块并行。

CUDA从入门到精通（六）：块并行

同一版本的代码用了这么多次，有点过意不去，于是这次我要做较大的改动大笑，大家要擦亮眼睛，拭目以待。

块并行相当于操作系统中多进程的情况，上节说到，CUDA有线程组（线程块）的概念，将一组线程组织到一起，共同分配一部分资源，然后内部调度执行。线程块与线程块之间，毫无瓜葛。这有利于做更粗粒度的并行。我们将上一节的代码改为块并行版本如下：

```
[cpp] view plain copy print ?
01. #include "cuda_runtime.h"
02. #include "device_launch_parameters.h"
03. #include <stdio.h>
04. cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size);
05. __global__ void addKernel(int *c, const int *a, const int *b)
06. {
07.     int i = blockIdx.x;
08.     c[i] = a[i] + b[i];
09. }
10. int main()
11. {
12.     const int arraySize = 5;
```

```
13.     const int a[arraySize] = { 1, 2, 3, 4, 5 };
14.     const int b[arraySize] = { 10, 20, 30, 40, 50 };
15.     int c[arraySize] = { 0 };
16.     // Add vectors in parallel.
17.     cudaError_t cudaStatus;
18.     int num = 0;
19.     cudaDeviceProp prop;
20.     cudaStatus = cudaGetDeviceCount(&num);
21.     for(int i = 0;i<num;i++)
22.     {
23.         cudaGetDeviceProperties(&prop,i);
24.     }
25.     cudaStatus = addWithCuda(c, a, b, arraySize);
26.     if (cudaStatus != cudaSuccess)
27.     {
28.         fprintf(stderr, "addWithCuda failed!");
29.         return 1;
30.     }
31.     printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0],c[1],c[2],c[3],c[4]);
32.     // cudaThreadExit must be called before exiting in order for profiling and
33.     // tracing tools such as Nsight and Visual Profiler to show complete traces.
34.     cudaStatus = cudaThreadExit();
35.     if (cudaStatus != cudaSuccess)
36.     {
37.         fprintf(stderr, "cudaThreadExit failed!");
38.         return 1;
39.     }
40.     return 0;
41. }
42. // Helper function for using CUDA to add vectors in parallel.
43. cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size)
44. {
45.     int *dev_a = 0;
46.     int *dev_b = 0;
```

```
47.     int *dev_c = 0;
48.     cudaError_t cudaStatus;
49.
50.     // Choose which GPU to run on, change this on a multi-GPU system.
51.     cudaStatus = cudaSetDevice(0);
52.     if (cudaStatus != cudaSuccess)
53.     {
54.         fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-
capable GPU installed?");
55.         goto Error;
56.     }
57.     // Allocate GPU buffers for three vectors (two input, one output)
58.     cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
59.     if (cudaStatus != cudaSuccess)
60.     {
61.         fprintf(stderr, "cudaMalloc failed!");
62.         goto Error;
63.     }
64.     cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
65.     if (cudaStatus != cudaSuccess)
66.     {
67.         fprintf(stderr, "cudaMalloc failed!");
68.         goto Error;
69.     }
70.     cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
71.     if (cudaStatus != cudaSuccess)
72.     {
73.         fprintf(stderr, "cudaMalloc failed!");
74.         goto Error;
75.     }
76.     // Copy input vectors from host memory to GPU buffers.
77.     cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
78.     if (cudaStatus != cudaSuccess)
79.     {
80.         fprintf(stderr, "cudaMemcpy failed!");
```

```
81.         goto Error;
82.     }
83.     cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
84.     if (cudaStatus != cudaSuccess)
85.     {
86.         fprintf(stderr, "cudaMemcpy failed!");
87.         goto Error;
88.     }
89.     // Launch a kernel on the GPU with one thread for each element.
90.     <span style="BACKGROUND-COLOR: #ff0000"> addKernel<<<size,1 >>>(dev_c, dev_a, dev_b);
91. </span>     // cudaThreadSynchronize waits for the kernel to finish, and returns
92.     // any errors encountered during the launch.
93.     cudaStatus = cudaThreadSynchronize();
94.     if (cudaStatus != cudaSuccess)
95.     {
96.         fprintf(stderr, "cudaThreadSynchronize returned error code %d after launching a
97.         ddKernel!\n", cudaStatus);
98.         goto Error;
99.     }
100.    // Copy output vector from GPU buffer to host memory.
101.    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
102.    if (cudaStatus != cudaSuccess)
103.    {
104.        fprintf(stderr, "cudaMemcpy failed!");
105.        goto Error;
106.    }
107.    Error:
108.    cudaFree(dev_c);
109.    cudaFree(dev_a);
110.    cudaFree(dev_b);
111.    return cudaStatus;
112. }
```

和上一节相比，只有这两行有改变，<<<>>>里第一个参数改成了size，第二个改成了1，表示我们分配size个线程块，每个线程块仅包含1个线程，总共还是有5个线程。这5个线程相互独立，执行核函数得到相应的结果，与上一节不同的是，每个线程获取id的方式变为`int i = blockIdx.x`；这是线程块ID。

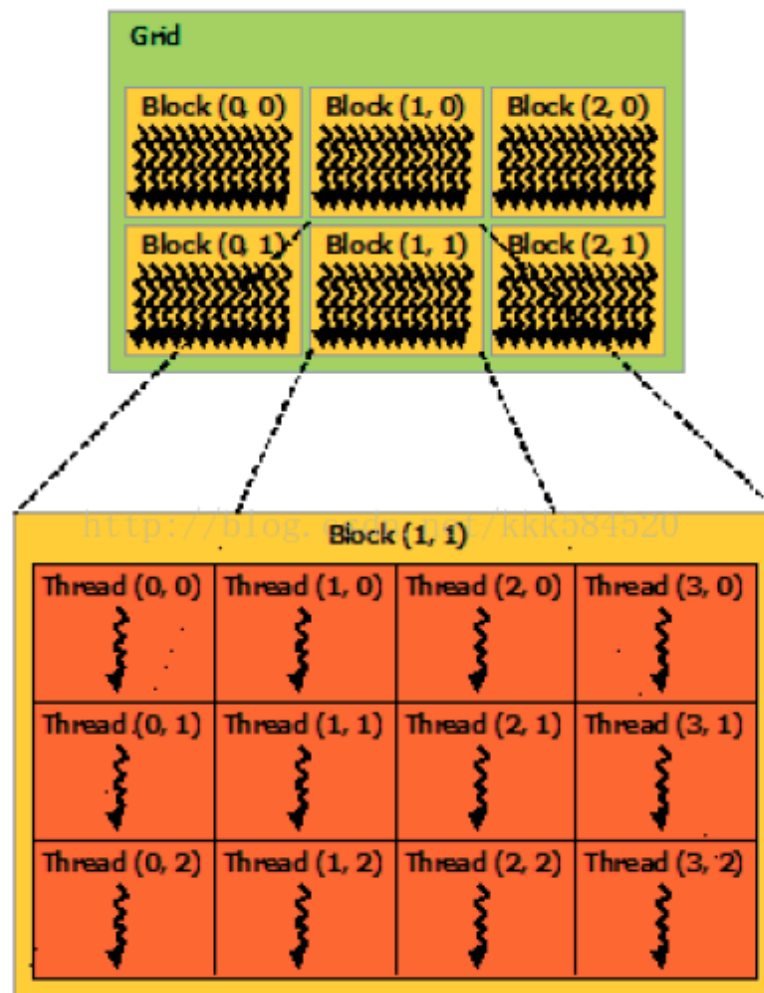
于是有童鞋提问了，线程并行和块并行的区别在哪里？

线程并行是细粒度并行，调度效率高；块并行是粗粒度并行，每次调度都要重新分配资源，有时资源只有一份，那么所有线程块都只能排成一队，串行执行。

那是不是我们所有时候都应该用线程并行，尽可能不用块并行？

当然不是，我们的任务有时可以采用分治法，将一个大问题分解为几个小规模问题，将这些小规模问题分别用一个线程块实现，线程块内可以采用细粒度的线程并行，而块之间为粗粒度并行，这样可以充分利用硬件资源，降低线程并行的计算复杂度。适当分解，降低规模，在一些矩阵乘法、向量内积计算应用中可以得到充分的展示。

实际应用中，常常是二者的结合。线程块、线程组织图如下所示。



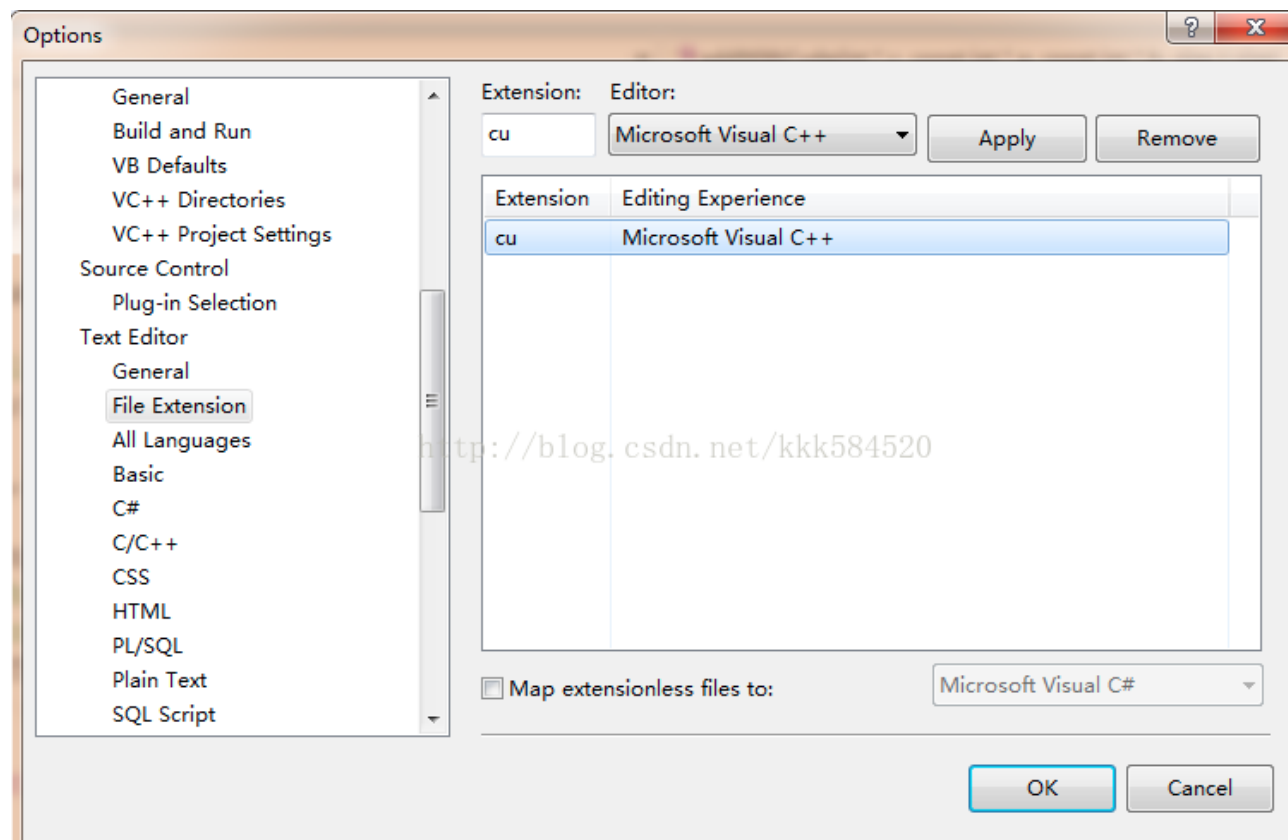
多个线程块组织成了一个Grid，称为线程格（经历了从一位线程，二维线程块到三维线程格的过程，立体感很强啊）。

好了，下一节我们介绍流并行，是更高层次的并行。

CUDA从入门到精通（七）：流并行

前面我们没有讲程序的结构，我想有些童鞋可能迫不及待想知道CUDA程序到底是怎么一个执行过程。好的，这一节在介绍流之前，先把CUDA程序结构简要说一下。

CUDA程序文件后缀为.cu，有些编译器可能不认识这个后缀的文件，我们可以在VS2008的Tools->Options->Text Editor->File Extension里添加cu后缀到VC++中，如下图：



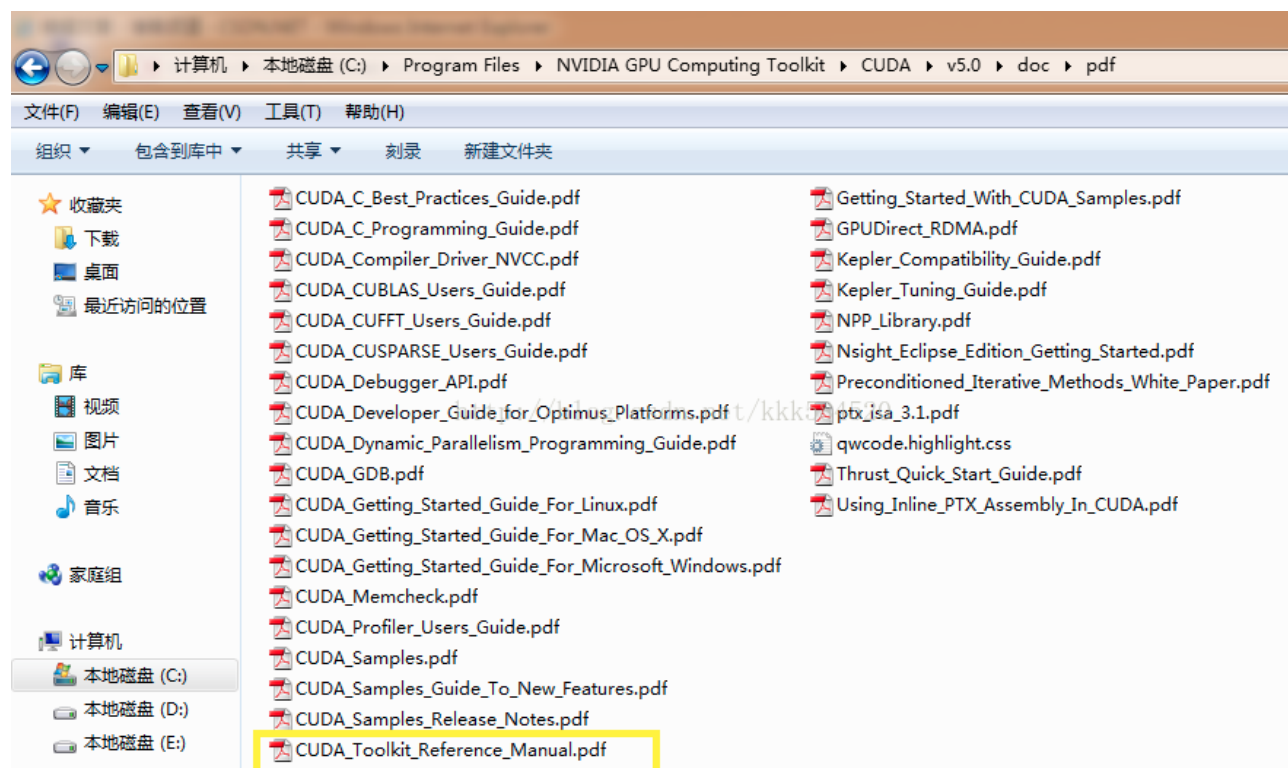
一个.cu文件内既包含CPU程序（称为主机程序），也包含GPU程序（称为设备程序）。如何区分主机程序和设备程序？根据声明，凡是挂有“__global__”或者“__device__”前缀的函数，都是在GPU上运行的设备程序，不同的是__global__设备程序可被主机程序调用，而__device__设备程序则只能被设备程序调用。

没有挂任何前缀的函数，都是主机程序。主机程序显示声明可以用__host__前缀。设备程序需要由

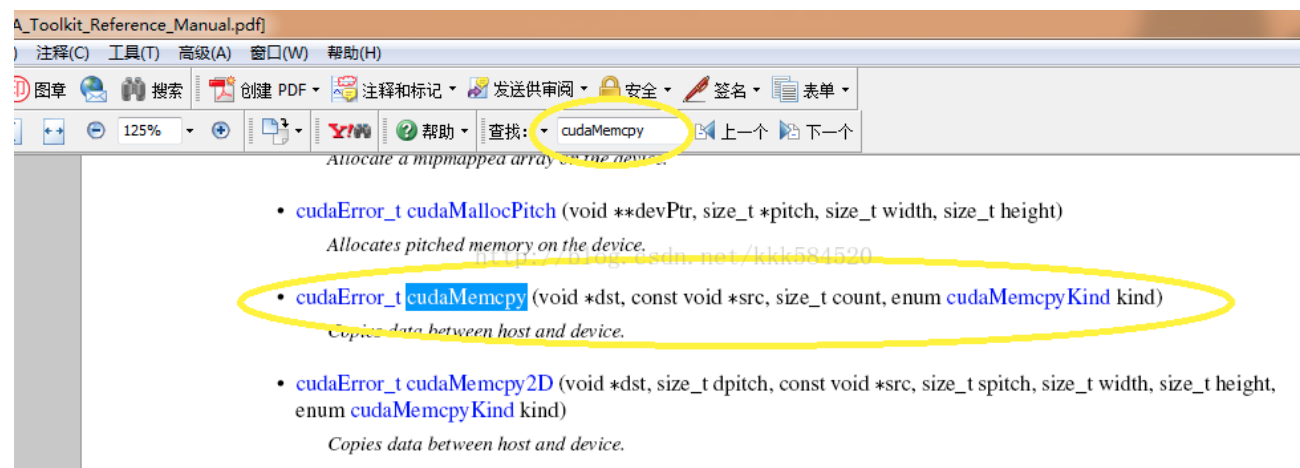
NVCC进行编译，而主机程序只需要由主机编译器（如VS2008中的cl.exe，Linux上的GCC）。主机程序主要完成设备环境初始化，数据传输等必备过程，设备程序只负责计算。

主机程序中，有一些“cuda”打头的函数，这些都是CUDA Runtime API，即运行时函数，主要负责完成设备的初始化、内存分配、内存拷贝等任务。我们前面第三节用到的函数

cudaGetDeviceCount(), cudaGetDeviceProperties(), cudaSetDevice()都是运行时API。这些函数的具体参数声明我们不必一一记下来，拿出第三节的官方利器就可以轻松查询，让我们打开这个文件：



打开后，在pdf搜索栏中输入一个运行时函数，例如cudaMemcpy，查到的结果如下：



可以看到，该API函数的参数形式为，第一个表示目的地，第二个表示来源地，第三个参数表示字节数，第四个表示类型。如果对类型不了解，直接点击超链接，得到详细解释如下：

5.28.3.10 enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

```
cudaMemcpyHostToHost  Host -> Host
cudaMemcpyHostToDevice Host -> Device
cudaMemcpyDeviceToHost Device -> Host
cudaMemcpyDeviceToDevice Device -> Device
cudaMemcpyDefault     Default based unified virtual address space
```

可见，该API可以实现从主机到主机、主机到设备、设备到主机、设备到设备的内存拷贝过程。同时可以发现，利用该API手册可以很方便地查询我们需要用的这些API函数，所以以后编CUDA程序一定要把它打开，随时准备查询，这样可以大大提高编程效率。

好了，进入今天的主题：流并行。

前面已经介绍了线程并行和块并行，知道了线程并行为细粒度的并行，而块并行为粗粒度的并行，同时也知道了CUDA的线程组织情况，即Grid-Block-Thread结构。一组线程并行处理可以组织为一个block，而一组block并行处理可以组织为一个Grid，很自然地想到，Grid只是一个网格，我们是否可以利用多个网格来完成并行处理呢？答案就是利用流。

流可以实现在一个设备上运行多个核函数。前面的块并行也好，线程并行也好，运行的核函数都是相同的（代码一样，传递参数也一样）。而流并行，可以执行不同的核函数，也可以实现对同一个核函数传递不同的参数，实现任务级别的并行。

CUDA中的流用`cudaStream_t`类型实现，用到的API有以下几个：`cudaStreamCreate(cudaStream_t *s)`用于创建流，`cudaStreamDestroy(cudaStream_t s)`用于销毁流，`cudaStreamSynchronize()`用于单个流同步，`cudaDeviceSynchronize()`用于整个设备上的所有流同步，`cudaStreamQuery()`用于查询一个流的任务是否已经完成。具体的含义可以查询API手册。

下面我们将前面的两个例子中的任务改用流实现，仍然是 $\{1,2,3,4,5\} + \{10,20,30,40,50\} = \{11,22,33,44,55\}$ 这个例子。代码如下：

[\[cpp\]](#) [view plain](#) [copy](#) [print ?](#)

```
01. #include "cuda_runtime.h"
02. #include "device_launch_parameters.h"
03. #include <stdio.h>
04. cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size);
05. __global__ void addKernel(int *c, const int *a, const int *b)
06. {
07.     int i = blockIdx.x;
08.     c[i] = a[i] + b[i];
09. }
10. int main()
11. {
12.     const int arraySize = 5;
13.     const int a[arraySize] = { 1, 2, 3, 4, 5 };
14.     const int b[arraySize] = { 10, 20, 30, 40, 50 };
15.     int c[arraySize] = { 0 };
16.     // Add vectors in parallel.
17.     cudaError_t cudaStatus;
18.     int num = 0;
19.     cudaDeviceProp prop;
20.     cudaStatus = cudaGetDeviceCount(&num);
21.     for(int i = 0; i < num; i++)
22.     {
23.         cudaGetDeviceProperties(&prop, i);
24.     }
25.     cudaStatus = addWithCuda(c, a, b, arraySize);
26.     if (cudaStatus != cudaSuccess)
27.     {
28.         fprintf(stderr, "addWithCuda failed!");
29.         return 1;
30.     }
31.     printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);
32.     // cudaThreadExit must be called before exiting in order for profiling and
```

```
33. // tracing tools such as Nsight and Visual Profiler to show complete traces.
34. cudaStatus = cudaThreadExit();
35. if (cudaStatus != cudaSuccess)
36. {
37.     fprintf(stderr, "cudaThreadExit failed!");
38.     return 1;
39. }
40. return 0;
41. }
42. // Helper function for using CUDA to add vectors in parallel.
43. cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size)
44. {
45.     int *dev_a = 0;
46.     int *dev_b = 0;
47.     int *dev_c = 0;
48.     cudaError_t cudaStatus;
49.
50.     // Choose which GPU to run on, change this on a multi-GPU system.
51.     cudaStatus = cudaSetDevice(0);
52.     if (cudaStatus != cudaSuccess)
53.     {
54.         fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-
55.         capable GPU installed?");
56.         goto Error;
57.     }
58.     // Allocate GPU buffers for three vectors (two input, one output)
59.     cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
60.     if (cudaStatus != cudaSuccess)
61.     {
62.         fprintf(stderr, "cudaMalloc failed!");
63.         goto Error;
64.     }
65.     cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
66.     if (cudaStatus != cudaSuccess)
67.     {
```



```
67.         fprintf(stderr, "cudaMalloc failed!");
68.         goto Error;
69.     }
70.     cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
71.     if (cudaStatus != cudaSuccess)
72.     {
73.         fprintf(stderr, "cudaMalloc failed!");
74.         goto Error;
75.     }
76.     // Copy input vectors from host memory to GPU buffers.
77.     cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
78.     if (cudaStatus != cudaSuccess)
79.     {
80.         fprintf(stderr, "cudaMemcpy failed!");
81.         goto Error;
82.     }
83.     cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
84.     if (cudaStatus != cudaSuccess)
85.     {
86.         fprintf(stderr, "cudaMemcpy failed!");
87.         goto Error;
88.     }
89.     cudaStream_t stream[5];
90.     for(int i = 0; i < 5; i++)
91.     {
92.         cudaStreamCreate(&stream[i]);    //创建流
93.     }
94.     // Launch a kernel on the GPU with one thread for each element.
95.     for(int i = 0; i < 5; i++)
96.     {
97.         addKernel<<<1,1,0,stream[i]>>>>(dev_c+i, dev_a+i, dev_b+i);    //执行流
98.     }
99.     cudaDeviceSynchronize();
100.    // cudaThreadSynchronize waits for the kernel to finish, and returns
101.    // any errors encountered during the launch.
```

```
102.     cudaStatus = cudaThreadSynchronize();
103.     if (cudaStatus != cudaSuccess)
104.     {
105.         fprintf(stderr, "cudaThreadSynchronize returned error code %d after launching a
ddKernel!\n", cudaStatus);
106.         goto Error;
107.     }
108.     // Copy output vector from GPU buffer to host memory.
109.     cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
110.     if (cudaStatus != cudaSuccess)
111.     {
112.         fprintf(stderr, "cudaMemcpy failed!");
113.         goto Error;
114.     }
115. Error:
116. <span style="BACKGROUND-COLOR: #ff6666">     for(int i = 0;i<5;i++)
117.     {
118.         cudaStreamDestroy(stream[i]);    //销毁流
119.     }
120. </span>     cudaFree(dev_c);
121.     cudaFree(dev_a);
122.     cudaFree(dev_b);
123.     return cudaStatus;
124. }
```

注意到，我们的核函数代码仍然和块并行的版本一样，只是在调用时做了改变，<<<>>>中的参数多了两个，其中前两个和块并行、线程并行中的意义相同，仍然是线程块数（这里为1）、每个线程块中线程数（这里也是1）。第三个为0表示每个block用到的共享内存大小，这个我们后面再讲；第四个为流对象，表示当前核函数在哪个流上运行。我们创建了5个流，每个流上都装载了一个核函数，同时传递参数有些不同，也就是每个核函数作用的对象也不同。这样就实现了任务级别的并行，当我们有几个互不相关的任

务时，可以写多个核函数，资源允许的情况下，我们将这些核函数装载到不同流上，然后执行，这样可以实现更粗粒度的并行。

好了，流并行就这么简单，我们处理任务时，可以根据需要，选择最适合的并行方式。

CUDA从入门到精通（八）：线程通信

我们前面几节主要介绍了三种利用GPU实现并行处理的方式：线程并行，块并行和流并行。在这些方法中，我们一再强调，各个线程所进行的处理是互不相关的，即两个线程不回产生交集，每个线程都只关注自己的一亩三分地，对其他线程毫无兴趣，就当不存在。。。。

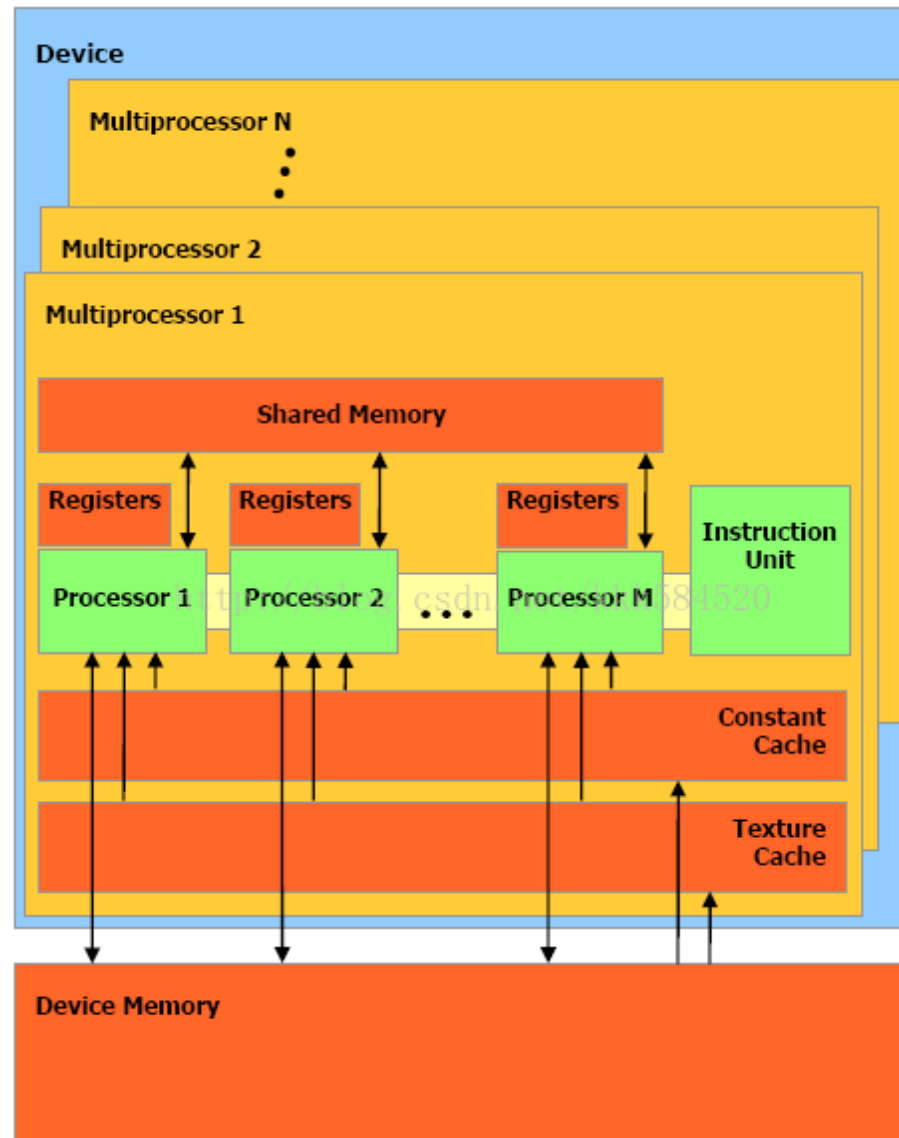
当然，实际应用中，这样的例子太少了，也就是遇到向量相加、向量对应点乘这类才会有如此高的并行度，而其他一些应用，如一组数求和，求最大（小）值，各个线程不再是相互独立的，而是产生一定关联，线程2可能会用到线程1的结果，这时就需要利用本节的线程通信技术了。

线程通信在CUDA中有三种实现方式：

1. 共享存储器；
2. 线程 同步；

3. 原子操作；

最常用的是前两种方式，共享存储器，术语Shared Memory，是位于SM中的特殊存储器。还记得SM吗，就是流多处理器，大核是也。一个SM中不仅包含若干个SP（流处理器，小核），还包括一部分高速Cache，寄存器组，共享内存等，结构如图所示：



从图中可看出，一个SM内有M个SP，Shared Memory由这M个SP共同占有。另外指令单元也被这M个SP共享，即SIMT架构（单指令多线程架构），一个SM中所有SP在同一时间执行同一代码。

为了实现线程通信，仅仅靠共享内存还不够，需要有同步机制才能使线程之间实现有序处理。通常情况是这样：当线程A需要线程B计算的结果作为输入时，需要确保线程B已经将结果写入共享内存中，然后线程A再从共享内存中读出。同步必不可少，否则，线程A可能读到的是无效的结果，造成计算错误。同步机制可以用CUDA内置函数：__syncthreads()；当某个线程执行到该函数时，进入等待状态，直到同一线程块（Block）中所有线程都执行到这个函数为止，即一个__syncthreads()相当于一个线程同步点，确保一个Block中所有线程都达到同步，然后线程进入运行状态。

综上两点，我们可以写一段线程通信的伪代码如下：

```
//Begin

if this is thread B

    write something to Shared Memory;

end if

__syncthreads();

if this is thread A

    read something from Shared Memory;
```

```
end if
```

```
//End
```

上面代码在CUDA中实现时，由于SIMT特性，所有线程都执行同样的代码，所以在线程中需要判断自己的身份，以免误操作。

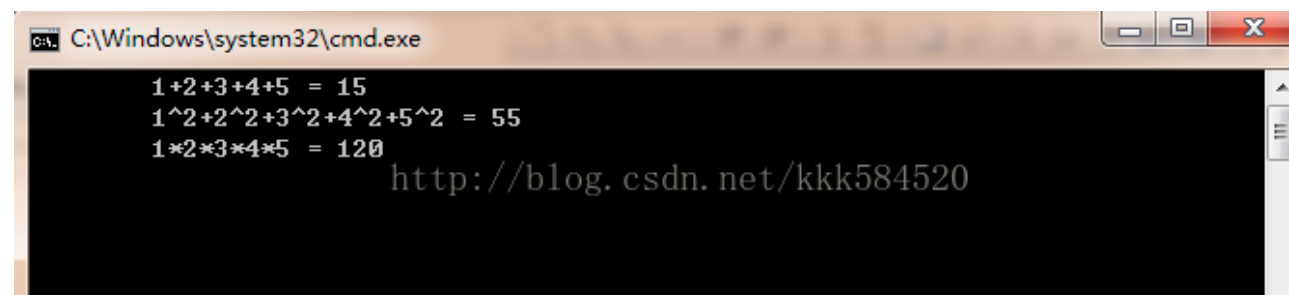
注意的是，位于同一个Block中的线程才能实现通信，不同Block中的线程不能通过共享内存、同步进行通信，而应采用原子操作或主机介入。

对于原子操作，如果感兴趣可以翻阅《GPU高性能编程CUDA实战》第九章“原子性”。

本节完。下节我们给出一个实例来看线程通信的代码怎么设计。

CUDA从入门到精通（九）：线程通信实例

接着上一节，我们利用刚学到的共享内存和线程同步技术，来做一个简单的例子。先看下效果吧：



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. It displays three mathematical calculations: $1+2+3+4+5 = 15$, $1^2+2^2+3^2+4^2+5^2 = 55$, and $1*2*3*4*5 = 120$. Below these calculations is a URL: <http://blog.csdn.net/kkk584520>. The window includes standard Windows window controls (minimize, maximize, close) in the top right corner.

很简单，就是分别求出1~5这5个数字的和，平方和，连乘积。相信学过C语言的童鞋都能用for循环做出同上面一样的效果，但为了学习CUDA共享内存和同步技术，我们还是要简单的东西复杂化(^_^)。

简要分析一下，上面例子的输入都是一样的，1,2,3,4,5这5个数，但计算过程有些变化，而且每个输出和所有输入都相关，不是前几节例子中那样，一个输出只和一个输入有关。所以我们在利用CUDA编程时，需要针对特殊问题做些让步，把一些步骤串行化实现。

输入数据原本位于主机内存，通过cudaMemcpy API已经拷贝到GPU显存（术语为全局存储器，Global Memory），每个线程运行时需要从Global Memory读取输入数据，然后完成计算，最后将结果写回Global Memory。当我们计算需要多次相同输入数据时，大家可能想到，每次都分别去Global Memory读数据好像有点浪费，如果数据很大，那么反复多次读数据会相当耗时间。索性我们把它从Global Memory一次性读到SM内部，然后在内部进行处理，这样可以节省反复读取的时间。

有了这个思路，结合上节看到的SM结构图，看到有一片存储器叫做Shared Memory，它位于SM内部，处理时访问速度相当快（差不多每个时钟周期读一次），而全局存储器读一次需要耗费几十甚至上百个时钟周期。于是，我们就制定A计划如下：

线程块数：1，块号为0；（只有一个线程块内的线程才能进行通信，所以我们只分配一个线程块，具体工作交给每个线程完成）

线程数：5，线程号分别为0~4；（线程并行，前面讲过）

共享存储器大小：5个int型变量大小（5*sizeof(int)）。

步骤一：读取输入数据。将Global Memory中的5个整数读入共享存储器，位置一一对应，和线程号也一一对应，所以可以同时完成。

步骤二：线程同步，确保所有线程都完成了工作。

步骤三：指定线程，对共享存储器中的输入数据完成相应处理。

代码如下：

```
[cpp] view plain copy print ?
01. #include "cuda_runtime.h"
02. #include "device_launch_parameters.h"
03.
04. #include <stdio.h>
05.
06. cudaError_t addWithCuda(int *c, const int *a, size_t size);
07.
08. __global__ void addKernel(int *c, const int *a)
09. {
10.     int i = threadIdx.x;
11.     extern __shared__ int smem[];
12.     smem[i] = a[i];
13.     __syncthreads();
14.     if(i == 0) //0号线程做平方和
15.     {
```

```
16.         c[0] = 0;
17.         for(int d = 0;d<5;d++)
18.         {
19.             c[0] += smem[d]*smem[d];
20.         }
21.     }
22.     if(i == 1)//1号线程做累加
23.     {
24.         c[1] = 0;
25.         for(int d = 0;d<5;d++)
26.         {
27.             c[1] += smem[d];
28.         }
29.     }
30.     if(i == 2) //2号线程做累乘
31.     {
32.         c[2] = 1;
33.         for(int d = 0;d<5;d++)
34.         {
35.             c[2] *= smem[d];
36.         }
37.     }
38. }
39.
40. int main()
41. {
42.     const int arraySize = 5;
43.     const int a[arraySize] = { 1, 2, 3, 4, 5 };
44.     int c[arraySize] = { 0 };
45.     // Add vectors in parallel.
46.     cudaError_t cudaStatus = addWithCuda(c, a, arraySize);
47.     if (cudaStatus != cudaSuccess)
48.     {
49.         fprintf(stderr, "addWithCuda failed!");
50.         return 1;
```

```
51.     }
52.     printf("\t1+2+3+4+5 = %d\n\t1^2+2^2+3^2+4^2+5^2 = %d\n\t1*2*3*4*5 = %d\n\n\n\n\n\n", c[1], c[0], c[2]);
53.     // cudaThreadExit must be called before exiting in order for profiling and
54.     // tracing tools such as Nsight and Visual Profiler to show complete traces.
55.     cudaStatus = cudaThreadExit();
56.     if (cudaStatus != cudaSuccess)
57.     {
58.         fprintf(stderr, "cudaThreadExit failed!");
59.         return 1;
60.     }
61.     return 0;
62. }
63.
64. // Helper function for using CUDA to add vectors in parallel.
65. cudaError_t addWithCuda(int *c, const int *a, size_t size)
66. {
67.     int *dev_a = 0;
68.     int *dev_c = 0;
69.     cudaError_t cudaStatus;
70.
71.     // Choose which GPU to run on, change this on a multi-GPU system.
72.     cudaStatus = cudaSetDevice(0);
73.     if (cudaStatus != cudaSuccess)
74.     {
75.         fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-
76.         capable GPU installed?");
77.         goto Error;
78.     }
79.     // Allocate GPU buffers for three vectors (two input, one output)
80.     cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
81.     if (cudaStatus != cudaSuccess)
82.     {
83.         fprintf(stderr, "cudaMalloc failed!");
```

```
84.         goto Error;
85.     }
86.
87.     cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
88.     if (cudaStatus != cudaSuccess)
89.     {
90.         fprintf(stderr, "cudaMalloc failed!");
91.         goto Error;
92.     }
93.     // Copy input vectors from host memory to GPU buffers.
94.     cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
95.     if (cudaStatus != cudaSuccess)
96.     {
97.         fprintf(stderr, "cudaMemcpy failed!");
98.         goto Error;
99.     }
100.    // Launch a kernel on the GPU with one thread for each element.
101.    <span style="font-size:24px;">
    <strong>    addKernel<<1, size, size*sizeof(int), 0>>>(dev_c, dev_a);</strong>
    </span>
102.    // cudaThreadSynchronize waits for the kernel to finish, and returns
103.    // any errors encountered during the launch.
104.    cudaStatus = cudaThreadSynchronize();
105.    if (cudaStatus != cudaSuccess)
106.    {
107.        fprintf(stderr, "cudaThreadSynchronize returned error code %d after launching a
108.        ddKernel!\n", cudaStatus);
109.        goto Error;
110.    }
111.
112.    // Copy output vector from GPU buffer to host memory.
113.    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
114.    if (cudaStatus != cudaSuccess)
115.    {
116.        fprintf(stderr, "cudaMemcpy failed!");
```

```
117.         goto Error;
118.     }
119.
120. Error:
121.     cudaFree(dev_c);
122.     cudaFree(dev_a);
123.     return cudaStatus;
124. }
```

从代码中看到执行配置<<<>>>中第三个参数为共享内存大小（字节数），这样我们就知道了全部4个执行配置参数的意义。恭喜，你的CUDA终于入门了！

CUDA从入门到精通（十）：性能剖析和Visual Profiler

入门后的进一步学习的内容，就是如何优化自己的代码。我们前面的例子没有考虑任何性能方面优化，是为了更好地学习基本知识点，而不是其他细节问题。从本节开始，我们要从性能出发考虑问题，不断优化代码，使执行速度提高是并行处理的唯一目的。

测试代码运行速度有很多方法，C语言里提供了类似于SystemTime()这样的API获得系统时间，然后计算两个事件之间的时长从而完成计时功能。在CUDA中，我们有专门测量设备运行时间的API，下面一一介绍。

翻开编程手册《CUDA_Toolkit_Reference_Manual》，随时准备查询不懂得API。我们在运行核函数前后，做如下操作：

```
[cpp] view plain copy print ?
01.  cudaEvent_t start, stop; //事件对象
02.  cudaEventCreate(&start); //创建事件
03.  cudaEventCreate(&stop); //创建事件
04.  cudaEventRecord(start, stream); //记录开始
05.  myKernel<<<dimg, dimb, size_smem, stream>>>(parameter list); //执行核函数
06.
07.  cudaEventRecord(stop, stream); //记录结束事件
08.  cudaEventSynchronize(stop); //事件同步，等待结束事件之前的设备操作均已完成
09.  float elapsedTime;
10.  cudaEventElapsedTime(&elapsedTime, start, stop); //计算两个事件之间时长（单位为ms）
```

核函数执行时间将被保存在变量elapsedTime中。通过这个值我们可以评估算法的性能。下面给一个例子，来看怎么使用计时功能。

前面的例子规模很小，只有5个元素，处理量太小不足以计时，下面将规模扩大为1024，此外将反复运行1000次计算总时间，这样估计不容易受随机扰动影响。我们通过这个例子对比线程并行和块并行的性能如何。代码如下：

```
[cpp] view plain copy print ?
01.  #include "cuda_runtime.h"
02.  #include "device_launch_parameters.h"
03.  #include <stdio.h>
04.  cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size);
05.  __global__ void addKernel_blk(int *c, const int *a, const int *b)
06.  {
```

```
07.     int i = blockIdx.x;
08.     c[i] = a[i]+ b[i];
09. }
10. __global__ void addKernel_thd(int *c, const int *a, const int *b)
11. {
12.     int i = threadIdx.x;
13.     c[i] = a[i]+ b[i];
14. }
15. int main()
16. {
17.     const int arraySize = 1024;
18.     int a[arraySize] = {0};
19.     int b[arraySize] = {0};
20.     for(int i = 0;i<arraySize;i++)
21.     {
22.         a[i] = i;
23.         b[i] = arraySize-i;
24.     }
25.     int c[arraySize] = {0};
26.     // Add vectors in parallel.
27.     cudaError_t cudaStatus;
28.     int num = 0;
29.     cudaDeviceProp prop;
30.     cudaStatus = cudaGetDeviceCount(&num);
31.     for(int i = 0;i<num;i++)
32.     {
33.         cudaGetDeviceProperties(&prop,i);
34.     }
35.     cudaStatus = addWithCuda(c, a, b, arraySize);
36.     if (cudaStatus != cudaSuccess)
37.     {
38.         fprintf(stderr, "addWithCuda failed!");
39.         return 1;
40.     }
41.
```

```
42. // cudaThreadExit must be called before exiting in order for profiling and
43. // tracing tools such as Nsight and Visual Profiler to show complete traces.
44. cudaStatus = cudaThreadExit();
45. if (cudaStatus != cudaSuccess)
46. {
47.     fprintf(stderr, "cudaThreadExit failed!");
48.     return 1;
49. }
50. for(int i = 0; i < arraySize; i++)
51. {
52.     if(c[i] != (a[i] + b[i]))
53.     {
54.         printf("Error in %d\n", i);
55.     }
56. }
57. return 0;
58. }
59. // Helper function for using CUDA to add vectors in parallel.
60. cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size)
61. {
62.     int *dev_a = 0;
63.     int *dev_b = 0;
64.     int *dev_c = 0;
65.     cudaError_t cudaStatus;
66.
67.     // Choose which GPU to run on, change this on a multi-GPU system.
68.     cudaStatus = cudaSetDevice(0);
69.     if (cudaStatus != cudaSuccess)
70.     {
71.         fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-
72.         capable GPU installed?");
73.         goto Error;
74.     }
75.     // Allocate GPU buffers for three vectors (two input, one output)
76.     cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
```

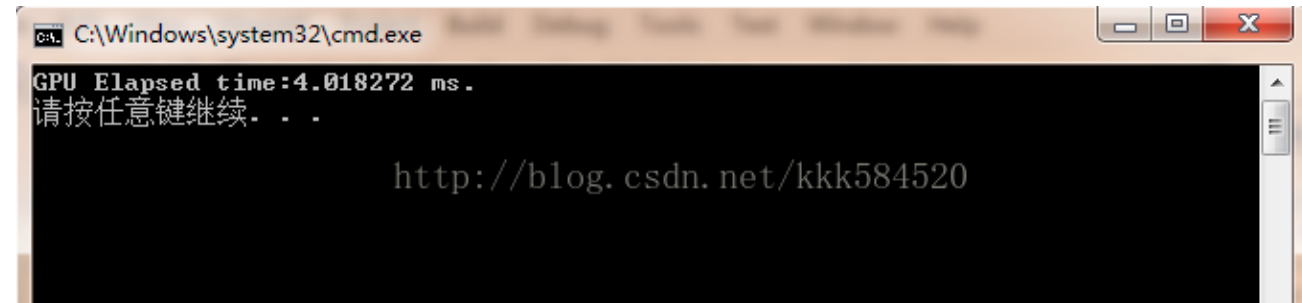


```
76.     if (cudaStatus != cudaSuccess)
77.     {
78.         fprintf(stderr, "cudaMalloc failed!");
79.         goto Error;
80.     }
81.     cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
82.     if (cudaStatus != cudaSuccess)
83.     {
84.         fprintf(stderr, "cudaMalloc failed!");
85.         goto Error;
86.     }
87.     cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
88.     if (cudaStatus != cudaSuccess)
89.     {
90.         fprintf(stderr, "cudaMalloc failed!");
91.         goto Error;
92.     }
93.     // Copy input vectors from host memory to GPU buffers.
94.     cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
95.     if (cudaStatus != cudaSuccess)
96.     {
97.         fprintf(stderr, "cudaMemcpy failed!");
98.         goto Error;
99.     }
100.    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
101.    if (cudaStatus != cudaSuccess)
102.    {
103.        fprintf(stderr, "cudaMemcpy failed!");
104.        goto Error;
105.    }
106.    cudaEvent_t start, stop;
107.    cudaEventCreate(&start);
108.    cudaEventCreate(&stop);
109.    cudaEventRecord(start, 0);
110.    for(int i = 0; i < 1000; i++)
```

```
111.     {
112.     //      addKernel_blk<<<size,1>>>(dev_c, dev_a, dev_b);
113.           addKernel_thd<<<1,size>>>(dev_c, dev_a, dev_b);
114.     }
115.     cudaEventRecord(stop,0);
116.     cudaEventSynchronize(stop);
117.     float tm;
118.     cudaEventElapsedTime(&tm,start,stop);
119.     printf("GPU Elapsed time:%.6f ms.\n",tm);
120.     // cudaThreadSynchronize waits for the kernel to finish, and returns
121.     // any errors encountered during the launch.
122.     cudaStatus = cudaThreadSynchronize();
123.     if (cudaStatus != cudaSuccess)
124.     {
125.         fprintf(stderr, "cudaThreadSynchronize returned error code %d after launching a
ddKernel!\n", cudaStatus);
126.         goto Error;
127.     }
128.     // Copy output vector from GPU buffer to host memory.
129.     cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
130.     if (cudaStatus != cudaSuccess)
131.     {
132.         fprintf(stderr, "cudaMemcpy failed!");
133.         goto Error;
134.     }
135. Error:
136.     cudaFree(dev_c);
137.     cudaFree(dev_a);
138.     cudaFree(dev_b);
139.     return cudaStatus;
140. }
```

addKernel_blk是采用块并行实现的向量相加操作，而addKernel_thd是采用线程并行实现的向量相加操作。分别运行，得到的结果如下图所示：

线程并行：



```
C:\Windows\system32\cmd.exe
GPU Elapsed time:4.018272 ms.
请按任意键继续. . .
http://blog.csdn.net/kkk584520
```

块并行：



```
C:\Windows\system32\cmd.exe
GPU Elapsed time:64.003296 ms.
请按任意键继续. . .
http://blog.csdn.net/kkk584520
```

可见性能竟然相差近16倍！因此选择并行处理方法时，如果问题规模不是很大，那么采用线程并行是比较合适的，而大问题分多个线程块处理时，每个块内线程数不要太少，像本文中的只有1个线程，这是对硬件资源的极大浪费。一个理想的方案是，分N个线程块，每个线程块包含512个线程，将问题分解处理，效率往往比单一的线程并行处理或单一块并行处理高很多。这也是CUDA编程的精髓。

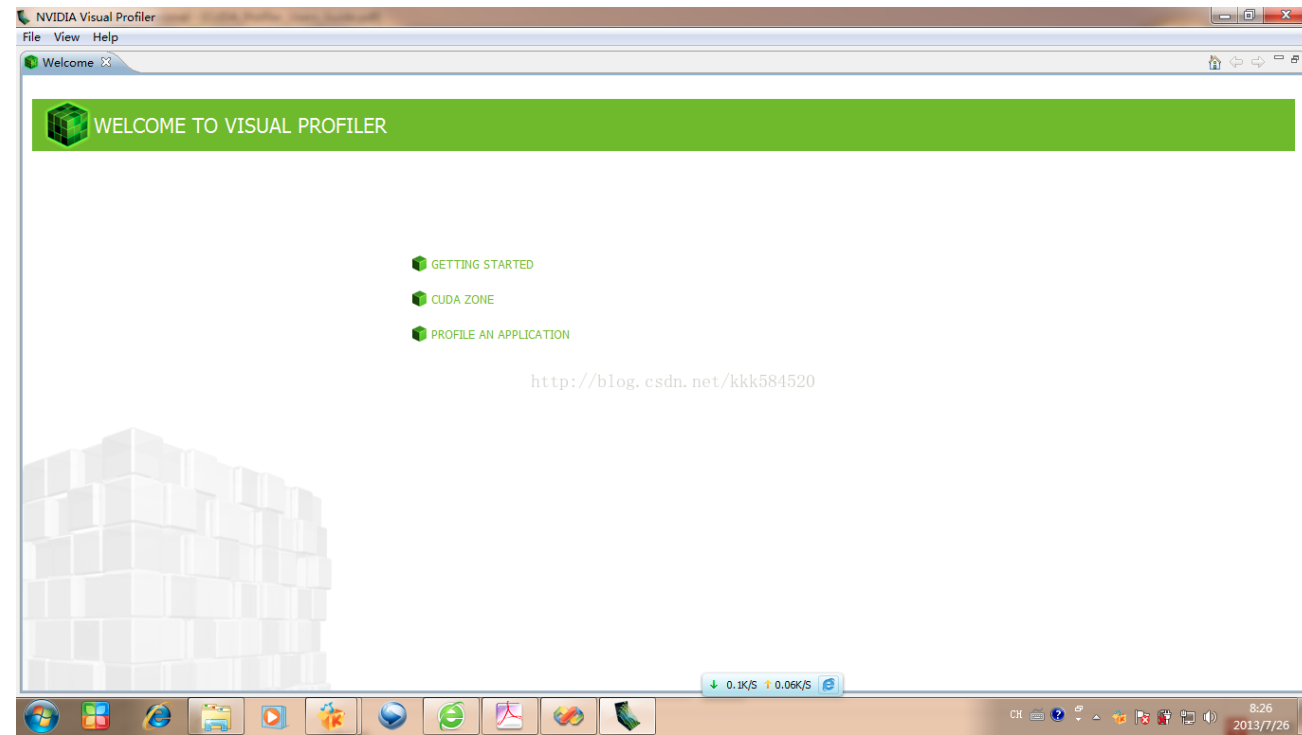
上面这种分析程序性能的方式比较粗糙，只知道大概运行时间长度，对于设备程序各部分代码执行时间没有一个深入的认识，这样我们就有个问题，如果对代码进行优化，那么优化哪一部分呢？是将线程数调节呢，还是改用共享内存？这个问题最好的解决方案就是利用Visual Profiler。以下内容摘自

《CUDA_Profiler_Users_Guide》

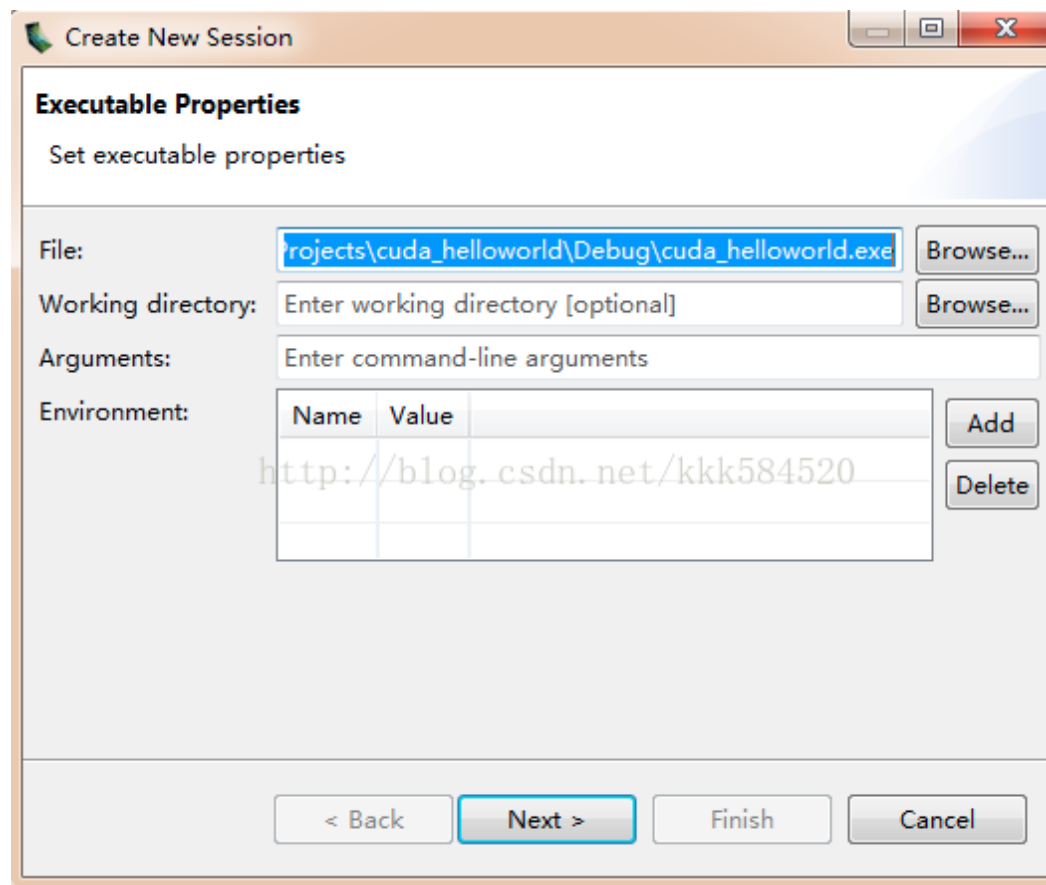
“Visual Profiler是一个图形化的剖析工具，可以显示你的应用程序中CPU和GPU的活动情况，利用分析引擎帮助你寻找优化的机会。”

其实除了可视化的界面，NVIDIA提供了命令行方式的剖析命令：nvprof。对于初学者，使用图形化的方式比较容易上手，所以本节使用Visual Profiler。

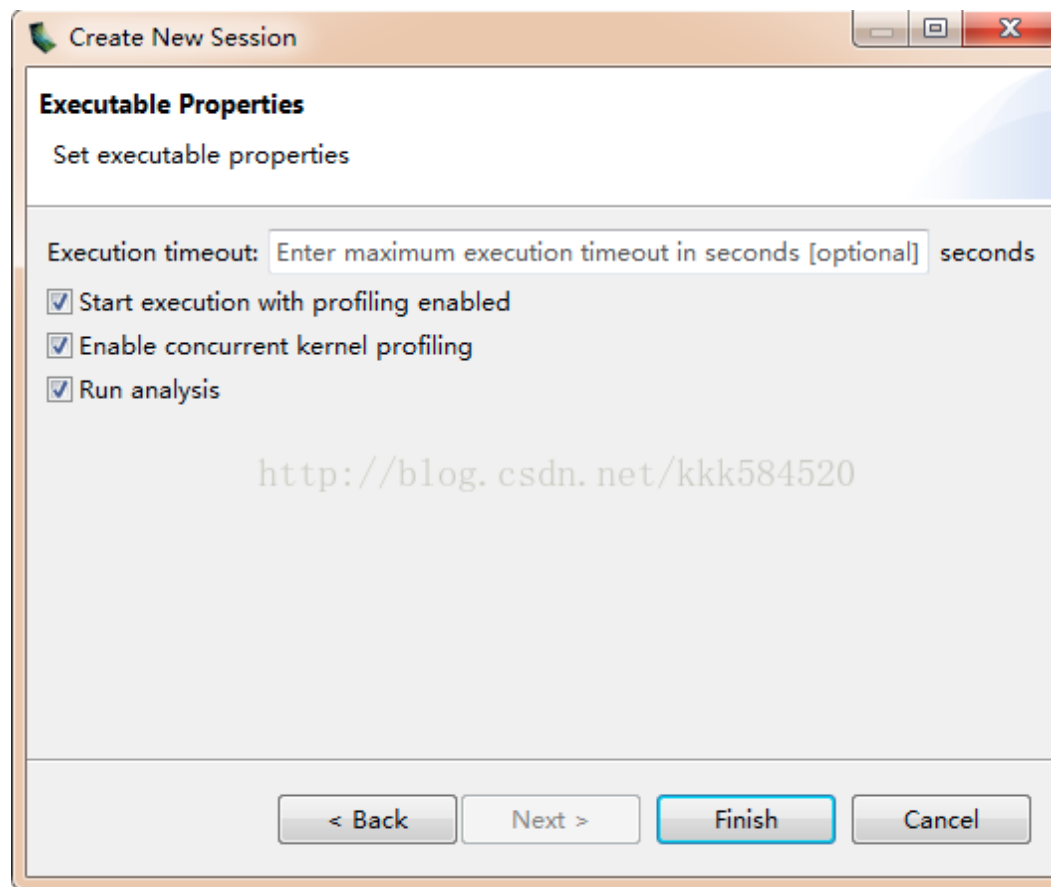
打开Visual Profiler，可以从CUDA Toolkit安装菜单处找到。主界面如下：



我们点击File->New Session，弹出新建会话对话框，如下图所示：

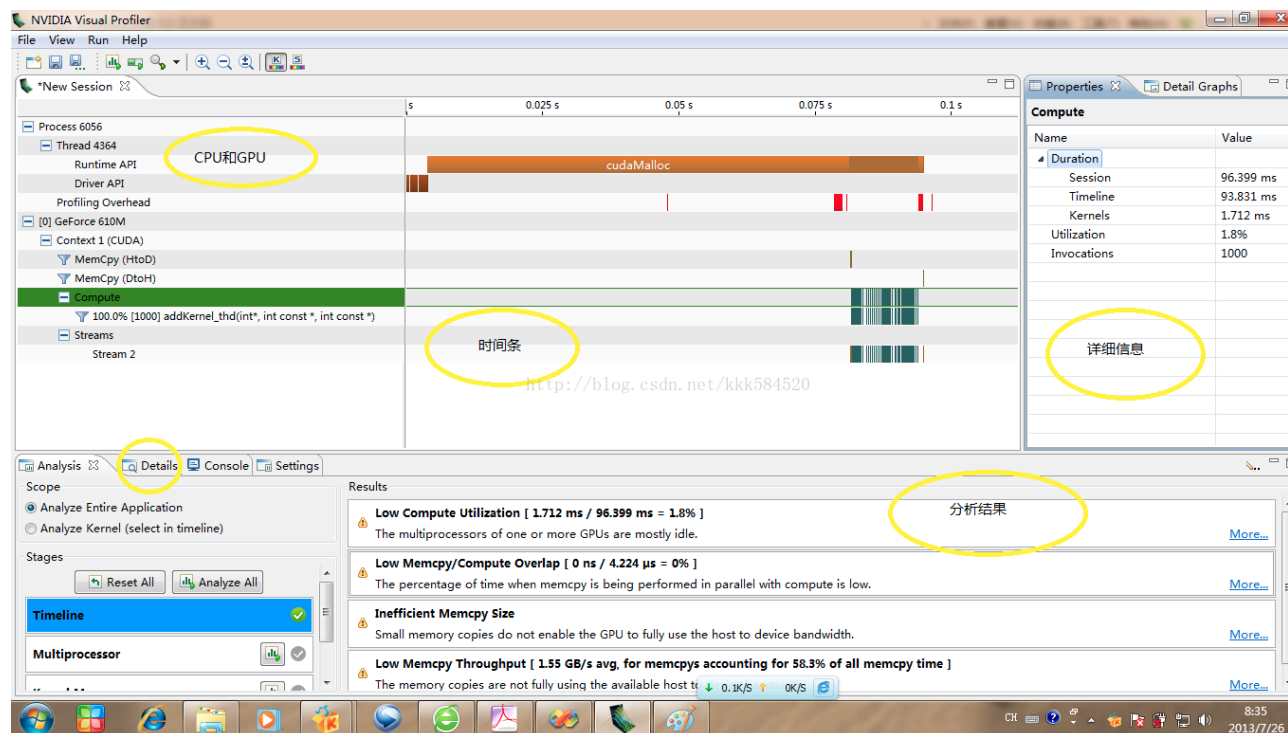


其中File一栏填入我们需要进行剖析的应用程序exe文件，后面可以都不填（如果需要命令行参数，可以在第三行填入），直接Next，见下图：



第一行为应用程序执行超时时间设定，可不填；后面三个单选框都勾上，这样我们分别使能了剖析，使能了并发核函数剖析，然后运行分析器。

点Finish，开始运行我们的应用程序并进行剖析、分析性能。



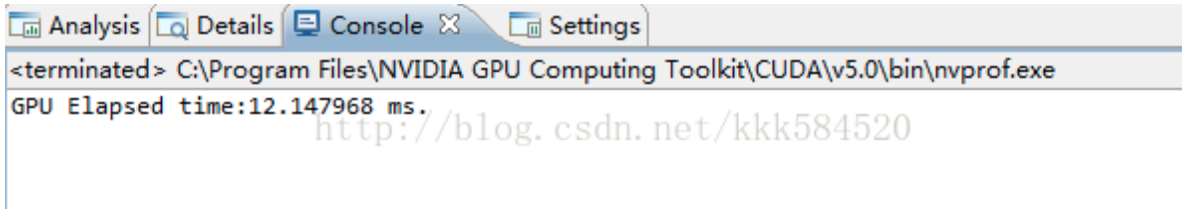
上图中，CPU和GPU部分显示了硬件和执行内容信息，点某一项则将时间条对应的部分高亮，便于观察，同时右边详细信息会显示运行时间信息。从时间条上看出，cudaMalloc占用了很大一部分时间。下面分析器给出了一些性能提升的关键点，包括：低计算利用率（计算时间只占总时间的1.8%，也难怪，加法计算复杂度本来就很低呀！）；低内存拷贝/计算交叠率（一点都没有交叠，完全是拷贝——计算——拷贝）；低存储拷贝尺寸（输入数据量太小了，相当于你淘宝买了个日记本，运费比实物价格还高！）；低存储拷贝吞吐率（只有1.55GB/s）。这些对我们进一步优化程序是非常有帮助的。

我们点一下Details，就在Analysis窗口旁边。得到结果如下所示：

Analysis Details Console Settings									
Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [sync]	81.351 ms	1.088 μs	n/a	n/a	n/a	n/a	n/a	4 ...	3.51 GB/s
Memcpy HtoD [sync]	81.683 ms	672 ns	n/a	n/a	n/a	n/a	n/a	4 ...	5.68 GB/s
addKernel_thd(int*, int const *, int const *)	81.687 ms	2.55 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.69 ms	1.731 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.693 ms	1.676 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.696 ms	1.682 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.699 ms	1.712 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.702 ms	1.704 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.705 ms	1.717 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.708 ms	1.704 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.711 ms	1.686 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.714 ms	1.725 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.717 ms	1.707 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.719 ms	1.729 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.722 ms	1.727 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.725 ms	1.653 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.728 ms	1.685 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.731 ms	1.7 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a
addKernel_thd(int*, int const *, int const *)	81.734 ms	1.725 μs	[1,1,1]	[1024,1,1]	6	0	0	n/a	n/a

通过这个窗口可以看到每个核函数执行时间，以及线程格、线程块尺寸，占用寄存器个数，静态共享内存、动态共享内存大小等参数，以及内存拷贝函数的执行情况。这个提供了比前面cudaEvent函数测时间更精确的方式，直接看到每一步的执行时间，精确到ns。

在Details后面还有一个Console，点一下看看。



这个其实就是命令行窗口，显示运行输出。看到加入了Profiler信息后，总执行时间变长了（原来线程并

行版本的程序运行时间只需4ms左右)。这也是“测不准定理”决定的，如果我们希望测量更细微的时间，那么总时间肯定是不准的；如果我们希望测量总时间，那么细微的时间就被忽略掉了。

后面Settings就是我们建立会话时的参数配置，不再详述。

通过本节，我们应该能对CUDA性能提升有了一些想法，好，下一节我们将讨论如何优化CUDA程序。

好文要顶

关注我

收藏该文

moffis

关注 - 6

粉丝 - 15

+加关注

0

0

« 上一篇: [CUDA编程指南阅读笔记](#)
» 下一篇: [CUDA学习](#)

posted on 2015-06-23 19:34 moffis 阅读(1046) 评论(0) 编辑 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】超50万C++/C#源码: 大型实时仿真组态图形源码

【推荐】[Java工作两年，一天竟收到33份面试通知](#)

【推荐】[程序员问答平台，解决您开发中遇到的技术难题](#)

Powered by:
博客园
Copyright © moffis