深入浅出谈 CUDA

Hotball

CUDA 是 NVIDIA 的 GPGPU 模型，它使用 C 语言为基础，可以直接以大多数人熟悉的 C 语言，写出在显示芯片上执行的程序，而不需要去学习特定的显示芯片的指令或是特殊的结构。”

CUDA是什么？能吃吗？

编者注：NVIDIA的GeFoce 8800GTX发布后，它的通用计算架构CUDA经过一年多的推广后，现在已经在有相当多的论文发表，在商业应用软件等方面也初步出现了视频编解码、金融、地质勘探、科学计算等领域的产品，是时候让我们对其作更深一步的了解。为了让大家更容易了解CUDA，我们征得Hotball的本人同意，发表他最近亲自撰写的本文。这篇文章的特点是深入浅出，也包含了hotball本人编写一些简单CUDA程序的亲身体验，对于希望了解CUDA的读者来说是非常不错的入门文章，PCINLIFE对本文的发表没有作任何的删减，主要是把一些台湾的词汇转换成大陆的词汇以及作了若干"编者注"的注释。

现代的显示芯片已经具有高度的可程序化能力，由于显示芯片通常具有相当高的内存带宽，以及大量的执行单元，因此开始有利用显示芯片来帮助进行一些计算工作的想法，即 GPGPU。CUDA 即是 NVIDIA 的 GPGPU 模型。

NVIDIA 的新一代显示芯片，包括 GeForce 8 系列及更新的显示芯片都支持 CUDA。NVIDIA 免费提供 CUDA 的开发工具（包括 Windows 版本和 Linux 版本）、程序范例、文件等等，可以在 CUDA Zone 下载。

GPGPU 的优缺点

使用显示芯片来进行运算工作，和使用 CPU 相比，主要有几个好处：

1. 显示芯片通常具有更大的内存带宽。例如，NVIDIA 的 GeForce 8800GTX 具有超过 50GB/s 的内存带宽，而目前高阶 CPU 的内存带宽则在 10GB/s 左右。
2. 显示芯片具有更大量的执行单元。例如 GeForce 8800GTX 具有 128 个 "stream processors"，频率为 1.35GHz。CPU 频率通常较高，但是执行单元的数目则要少得多。
3. 和高阶 CPU 相比，显卡的价格较为低廉。例如目前一张 GeForce 8800GT 包括 512MB 内存的价格，和一颗 2.4GHz 四核心 CPU 的价格相若。

当然，使用显示芯片也有它的一些缺点：

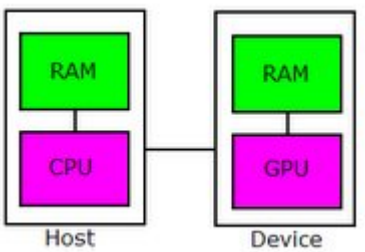
1. 显示芯片的运算单元数量很多，因此对于不能高度并行化的工作，所能带来的帮助就不大。
2. 显示芯片目前通常只支持 32 bits 浮点数，且多半不能完全支持 IEEE 754 规格， 有些运算的精确度可能较低。目前许多显示芯片并没有分开的整数运算单元，因此整数运算的效率较差。
3. 显示芯片通常不具有分支预测等复杂的流程控制单元，因此对于具有高度分支的程序，效率会比较差。
4. 目前 GPGPU 的程序模型仍不成熟，也还没有公认的标准。例如 NVIDIA 和 AMD/ATI 就有各自不同的程序模型。

整体来说，显示芯片的性质类似 stream processor，适合一次进行大量相同的工作。CPU 则比较有弹性，能同时进行变化较多的工作。

CUDA 架构

CUDA 是 NVIDIA 的 GPGPU 模型，它使用 C 语言为基础，可以直接以大多数人熟悉的 C 语言，写出在显示芯片上执行的程序，而不需要去学习特定的显示芯片的指令或是特殊的结构。

在 CUDA 的架构下，一个程序分为两个部份：host 端和 device 端。Host 端是指在 CPU 上执行的部份，而 device 端则是在显示芯片上执行的部份。Device 端的程序又称为 "kernel"。通常 host 端程序会将数据准备好后，复制到显卡的内存中，再由显示芯片执行 device 端程序，完成后再由 host 端程序将结果从显卡的内存中取回。

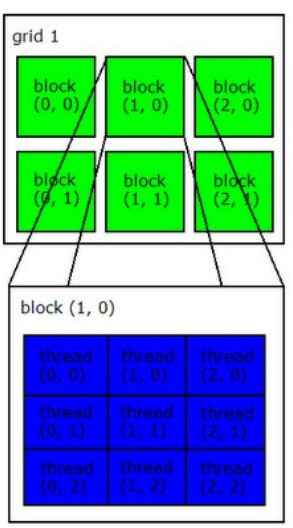


由于 CPU 存取显卡内存时只能透过 PCI Express 接口，因此速度较慢（PCI Express x16 的理论带宽是双向各 4GB/s），因此不能太常进行这类动作，以免降低效率。

在 CUDA 架构下，显示芯片执行时的最小单位是 **thread**。数个 thread 可以组成一个 **block**。一个 block 中的 thread 能存取同一块共享的内存，而且可以快速进行同步的动作。

每一个 block 所能包含的 thread 数目是有限的。不过，执行相同程序的 block，可以组成 **grid**。不同 block 中的 thread 无法存取同一个共享的内存，因此无法直接互通或进行同步。因此，不同 block 中的 thread 能合作的程度是比较低的。不过，利用这个模式，可以让程序不用担心显示芯片实际上能同时执行的 thread 数目限制。例如，一个具有很少量执行单元的显示芯片，可能会把各个 block 中的 thread 顺序执行，而非同时执行。不同的 grid 则可以执行不同的程序（即 kernel）。

Grid、block 和 thread 的关系，如下图所示：



每个 thread 都有自己的一份 register 和 local memory 的空间。同一个 block 中的每个 thread 则有共享的一份 share memory。此外，所有的 thread（包括不同 block 的 thread）都共享一份 global memory、constant memory、和 texture memory。不同的 grid 则有各自的 global memory、constant memory 和 texture memory。这些不同的内存的差别，会在之后讨论。

执行模式

由于显示芯片大量并行计算的特性，它处理一些问题的方式，和一般 CPU 是不同的。主要的特点包括：

1. 内存存取 latency 的问题：CPU 通常使用 cache 来减少存取主内存的次数，以避免内存 latency 影响到执行效率。显示芯片则多半没有 cache（或很小），而利用并行化执行的方式来隐藏内存的 latency（即，当第一个 thread 需要等待内存读取结果时，则开始执行第二个 thread，依此类推）。
2. 分支指令的问题：CPU 通常利用分支预测等方式来减少分支指令造成的 pipeline bubble。显示芯片则多半使用类似处理内存 latency 的方式。不过，通常显示芯片处理分支的效率会比较差。

因此，最适合利用 CUDA 处理的问题，是可以大量并行化的问题，才能有效隐藏内存的 latency，并有效利用显示芯片上的大量执行单元。使用 CUDA 时，同时有上千个 thread 在执行是很正常的。因此，如果不能大量并行化的问题，使用 CUDA 就没办法达到最好的效率了。

CUDA Toolkit的安装

目前 NVIDIA 提供的 CUDA Toolkit（可从这里下载）支持 Windows （32 bits 及 64 bits 版本）及许多不同的 Linux 版本。

CUDA Toolkit 需要配合 C/C++ compiler。在 Windows 下，目前只支持 Visual Studio 7.x 及 Visual Studio 8（包括免费的 Visual Studio C++ 2005 Express）。Visual Studio 6 和 gcc 在 Windows 下是不支援的。在 Linux 下则只支援 gcc。

这里简单介绍一下在 Windows 下设定并使用 CUDA 的方式。

下载及安装

在 Windows 下，CUDA Toolkit 和 CUDA SDK 都是由安装程序的形式安装的。CUDA Toolkit 包括 CUDA 的基本工具，而 CUDA SDK 则包括许多范例程序以及链接库。基本上要写 CUDA 的程序，只需要安装 CUDA Toolkit 即可。不过 CUDA SDK 仍值得安装，因为里面的许多范例程序和链接库都相当有用。

CUDA Toolkit 安装完后，预设会安装在 C:\CUDA 目录里。其中包括几个目录：

- bin -- 工具程序及动态链接库
- doc -- 文件
- include -- header 檔
- lib -- 链接库档案
- open64 -- 基于 Open64 的 CUDA compiler
- src -- 一些原始码

安装程序也会设定一些环境变量，包括：

- CUDA_BIN_PATH -- 工具程序的目录，默认为 C:\CUDA\bin
- CUDA_INC_PATH -- header 文件的目录，默认为 C:\CUDA\inc

- CUDA_LIB_PATH -- 链接库文件的目录，默认为 C:\CUDA\lib

在 Visual Studio 中使用 CUDA

CUDA 的主要工具是 nvcc，它会执行所需要的程序，将 CUDA 程序代码编译成执行档 (或 object 檔) 。在 Visual Studio 下，我们透过设定 custom build tool 的方式，让 Visual Studio 会自动执行 nvcc。

这里以 Visual Studio 2005 为例：

1. 首先，建立一个 Win32 Console 模式的 project（在 Application Settings 中记得勾选 Empty project），并新增一个档案，例如 main.cu。
2. 在 main.cu 上右键单击，并选择 **Properties**。点选 **General**，确定 **Tool** 的部份是选择 **Custom Build Tool**。
3. 选择 Custom Build Step，在 Command Line 使用以下设定：
   - **Release 模式**： "$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)bin" -c -DWIN32 -D_CONSOLE -D_MBCS -Xcompiler /EHsc,/W3,/nologo,/Wp64,/O2,/Zi,/MT -I"$(CUDA_INC_PATH)" -o $(ConfigurationName)\$(InputName).obj $(InputFileName)
   - **Debug 模式**： "$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)bin" -c -D_DEBUG -DWIN32 -D_CONSOLE -D_MBCS -Xcompiler /EHsc,/W3,/nologo,/Wp64,/Od,/Zi,/RTC1,/MTd -I"$(CUDA_INC_PATH)" -o $(ConfigurationName)\$(InputName).obj $(InputFileName)
4. 如果想要使用软件仿真的模式，可以新增两个额外的设定：
   - **EmuRelease 模式**： "$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)bin" -deviceemu -c -DWIN32 -D_CONSOLE -D_MBCS -Xcompiler /EHsc,/W3,/nologo,/Wp64,/O2,/Zi,/MT -I"$(CUDA_INC_PATH)" -o $(ConfigurationName)\$(InputName).obj $(InputFileName)
   - **EmuDebug 模式**： "$(CUDA_BIN_PATH)\nvcc.exe" -ccbin "$(VCInstallDir)bin" -deviceemu -c -D_DEBUG -DWIN32 -D_CONSOLE -D_MBCS -Xcompiler /EHsc,/W3,/nologo,/Wp64,/Od,/Zi,/RTC1,/MTd -I"$(CUDA_INC_PATH)" -o $(ConfigurationName)\$(InputName).obj $(InputFileName)
5. 对所有的配置文件，在 **Custom Build Step** 的 **Outputs** 中加入 $(ConfigurationName)\$(InputName).obj。
6. 选择 project，右键单击选择 **Properties**，再点选 **Linker**。对所有的配置文件修改以下设定：
   - General/Enable Incremental Linking：No
   - General/Additional Library Directories：$(CUDA_LIB_PATH)
   - Input/Additional Dependencies：cudart.lib

这样应该就可以直接在 Visual Studio 的 IDE 中，编辑 CUDA 程序后，直接 build 以及执行程序了。

第一个CUDA程序

CUDA 目前有两种不同的 API：Runtime API 和 Driver API，两种 API 各有其适用的范围。由于 runtime API 较容易使用，一开始我们会以 runetime API 为主。

CUDA 的初始化

首先，先建立一个档案 first_cuda.cu。如果是使用 Visual Studio 的话，则请先按照这里的设定方式设定 project。

要使用 runtime API 的时候，需要 include cuda_runtime.h。所以，在程序的最前面，加上

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

接下来是一个 InitCUDA 函式，会呼叫 runtime API 中，有关初始化 CUDA 的功能：

```
bool InitCUDA()
{
    int count;

    cudaGetDeviceCount(&count);
    if(count == 0) {
        fprintf(stderr, "There is no device.\n");
        return false;
    }

    int i;
    for(i = 0; i < count; i++) {
        cudaDeviceProp prop;
        if(cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
            if(prop.major >= 1) {
                break;
            }
        }
    }

    if(i == count) {
        fprintf(stderr, "There is no device supporting CUDA 1.x.\n");
        return false;
    }

    cudaSetDevice(i);

    return true;
}
```

这个函式会先呼叫 cudaGetDeviceCount 函式，取得支持 CUDA 的装置的数目。如果系统上没有支持 CUDA 的装置，则它会传回 1，而 device 0 会是一个仿真的装置，但不支持 CUDA 1.0 以上的功能。所以，要确定系统上是否有支持 CUDA 的装置，需要对每个 device 呼叫 cudaGetDeviceProperties 函式，取得装置的各项数据，并判断装置支持的 CUDA 版本（prop.major 和 prop.minor 分别代表装置支持的版本号码，例如 1.0 则 prop.major 为 1 而 prop.minor 为 0）。

透过 cudaGetDeviceProperties 函式可以取得许多数据，除了装置支持的 CUDA 版本之外，还有装置的名称、内存的大小、最大的 thread 数目、执行单元的频率等等。详情可参考 NVIDIA 的 CUDA Programming Guide。

在找到支持 CUDA 1.0 以上的装置之后，就可以呼叫 cudaSetDevice 函式，把它设为目前要使用的装置。

最后是 main 函式。在 main 函式中我们直接呼叫刚才的 InitCUDA 函式，并显示适当的讯息：

```
int main()
{
    if(!InitCUDA()) {
        return 0;
    }

    printf("CUDA initialized.\n");

    return 0;
}
```

这样就可以利用 nvcc 来 compile 这个程序了。使用 Visual Studio 的话，若按照先前的设定方式，可以直接 Build Project 并执行。

nvcc 是 CUDA 的 compile 工具，它会将 .cu 檔拆解出在 GPU 上执行的部份，及在 host 上执行的部份，并呼叫适当的程序进行 compile 动作。在 GPU 执行的部份会透过 NVIDIA 提供的 compiler 编译成中介码，而 host 执行的部份则会透过系统上的 C++ compiler 编译（在 Windows 上使用 Visual C++ 而在 Linux 上使用 gcc）。

编译后的程序，执行时如果系统上有支持 CUDA 的装置，应该会显示 CUDA initialized. 的讯息，否则会显示相关的错误讯息。

利用 CUDA 进行运算

到目前为止，我们的程序并没有做什么有用的工作。所以，现在我们加入一个简单的动作，就是把一大堆数字，计算出它的平方和。

首先，把程序最前面的 include 部份改成：

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define DATA_SIZE 1048576

int data[DATA_SIZE];
```

并加入一个新函式 GenerateNumbers：

```
void GenerateNumbers(int *number, int size)
{
    for(int i = 0; i < size; i++) {
        number[i] = rand() % 10;
    }
}
```

这个函式会产生一大堆 0 ~ 9 之间的随机数。

要利用 CUDA 进行计算之前，要先把数据复制到显卡内存中，才能让显示芯片使用。因此，需要取得一块适当大小的显卡内存，再把产生好的数据复制进去。在 main 函式中加入：

```
    GenerateNumbers(data, DATA_SIZE);

    int* gpudata, *result;
    cudaMalloc((void**) &gpudata, sizeof(int) * DATA_SIZE);
    cudaMalloc((void**) &result, sizeof(int));
```

```
    cudaMemcpy(gpudata, data, sizeof(int) * DATA_SIZE,
        cudaMemcpyHostToDevice);
```

上面这段程序会先呼叫 GenerateNumbers 产生随机数，并呼叫 cudaMalloc 取得一块显卡内存（result 则是用来存取计算结果，在稍后会用到），并透过 cudaMemcpy 将产生的随机数复制到显卡内存中。cudaMalloc 和 cudaMemcpy 的用法和一般的 malloc 及 memcpy 类似，不过 cudaMemcpy 则多出一个参数，指示复制内存的方向。在这里因为是从主内存复制到显卡内存，所以使用 cudaMemcpyHostToDevice。如果是从显卡内存到主内存，则使用 cudaMemcpyDeviceToHost。这在之后会用到。

接下来是要写在显示芯片上执行的程序。在 CUDA 中，在函式前面加上 __global__ 表示这个函式是要在显示芯片上执行的。因此，加入以下的函式：

```
__global__ static void sumOfSquares(int *num, int* result)
{
    int sum = 0;
    int i;
    for(i = 0; i < DATA_SIZE; i++) {
        sum += num[i] * num[i];
    }

    *result = sum;
}
```

在显示芯片上执行的程序有一些限制，例如它不能有传回值。其它的限制会在之后提到。

接下来是要让 CUDA 执行这个函式。在 CUDA 中，要执行一个函式，使用以下的语法：

函式名称<<<block 数目, thread 数目, shared memory 大小>>>(参数...);

呼叫完后，还要把结果从显示芯片复制回主内存上。在 main 函式中加入以下的程序：

```
    sumOfSquares<<<1, 1, 0>>>(gpudata, result);

    int sum;
    cudaMemcpy(&sum, result, sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(gpudata);
    cudaFree(result);

    printf("sum: %d\n", sum);
```

因为这个程序只使用一个 thread，所以 block 数目、thread 数目都是 1。我们也没有使用到任何 shared memory，所以设为 0。编译后执行，应该可以看到执行的结果。

为了确定执行的结果正确，我们可以加上一段以 CPU 执行的程序代码，来验证结果：

```
    sum = 0;
    for(int i = 0; i < DATA_SIZE; i++) {
        sum += data[i] * data[i];
    }
    printf("sum (CPU): %d\n", sum);
```

编译后执行，确认两个结果相同。

计算运行时间

CUDA 提供了一个 clock 函式，可以取得目前的 timestamp，很适合用来判断一段程序执行所花费的时间（单位为 GPU 执行单元的频率）。这对程序的优化也相当有用。要在我们的程序中记录时间，把 sumOfSquares 函式改成：

```
__global__ static void sumOfSquares(int *num, int* result,
    clock_t* time)
{
    int sum = 0;
    int i;
    clock_t start = clock();
    for(i = 0; i < DATA_SIZE; i++) {
        sum += num[i] * num[i];
    }

    *result = sum;
    *time = clock() - start;
}
```

把 main 函式中间部份改成：

```
    int* gpudata, *result;
    clock_t* time;
    cudaMalloc((void**) &gpudata, sizeof(int) * DATA_SIZE);
    cudaMalloc((void**) &result, sizeof(int));
    cudaMalloc((void**) &time, sizeof(clock_t));
    cudaMemcpy(gpudata, data, sizeof(int) * DATA_SIZE,
        cudaMemcpyHostToDevice);
    sumOfSquares<<<1, 1, 0>>>(gpudata, result, time);

    int sum;
    clock_t time_used;
    cudaMemcpy(&sum, result, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&time_used, time, sizeof(clock_t),
        cudaMemcpyDeviceToHost);
    cudaFree(gpudata);
    cudaFree(result);

    printf("sum: %d time: %d\n", sum, time_used);
```

编译后执行，就可以看到执行所花费的时间了。

如果计算实际运行时间的话，可能会注意到它的执行效率并不好。这是因为我们的程序并没有利用到 CUDA 的主要的优势，即并行化执行。在下一段文章中，会讨论如何进行优化的动作。

改良第一个 CUDA程序

在上一篇文章中，我们做了一个计算一大堆数字的平方和的程序。不过，我们也提到这个程序的执行效率并不理想。当然，实际上来说，如果只是要做计算平方和的动作，用 CPU 做会比用 GPU 快得多。这是因为平方和的计算并不需要太多运算能力，所以几乎都是被内存带宽所限制。因此，光是把数据复制到显卡内存上的这个动作，所需要的时间，可能已经和直接在 CPU 上进行计算差不多了。

不过，如果进行平方和的计算，只是一个更复杂的计算过程的一部份的话，那么当然在 GPU 上计算还是有它的好处的。而且，如果数据已经在显卡内存上（例如在 GPU 上透过某种算法产生），那么，使用 GPU 进行这样的运算，还是会比较快的。

刚才也提到了，由于这个计算的主要瓶颈是内存带宽，所以，理论上显卡的内存带宽是相当大的。这里我们就来看看，倒底我们的第一个程序，能利用到多少内存带宽。

程序的并行化

我们的第一个程序，并没有利用到任何并行化的功能。整个程序只有一个 thread。在 GeForce 8800GT 上面，在 GPU 上执行的部份（称为 "**kernel**"）大约花费 640M 个频率。GeForce 8800GT 的执行单元的频率是 1.5GHz，因此这表示它花费了约 0.43 秒的时间。1M 个 32 bits 数字的数据量是 4MB，因此，这个程序实际上使用的内存带宽，只有 9.3MB/s 左右！这是非常糟糕的表现。

为什么会有这样差的表现呢？这是因为 GPU 的架构特性所造成的。在 CUDA 中，一般的数据复制到的显卡内存的部份，称为 **global memory**。这些内存是没有 cache 的，而且，存取 global memory 所需要的时间（即 latency）是非常长的，通常是数百个 cycles。由于我们的程序只有一个 thread，所以每次它读取 global memory 的内容，就要等到实际读取到数据、累加到 sum 之后，才能进行下一步。这就是为什么它的表现会这么的差。

由于 global memory 并没有 cache，所以要避开巨大的 latency 的方法，就是要利用大量的 threads。假设现在有大量的 threads 在同时执行，那么当一个 thread 读取内存，开始等待结果的时候，GPU 就可以立刻切换到下一个 thread，并读取下一个内存位置。因此，理想上当 thread 的数目够多的时候，就可以完全把 global memory 的巨大 latency 隐藏起来了。

要怎么把计算平方和的程序并行化呢？最简单的方法，似乎就是把数字分成若干组，把各组数字分别计算平方和后，最后再把每组的和加总起来就可以了。一开始，我们可以把最后加总的动作，由 CPU 来进行。

首先，在 first_cuda.cu 中，在 #define DATA_SIZE 的后面增加一个 #define，设定 thread 的数目：

```
#define DATA_SIZE    1048576
#define THREAD_NUM    256
```

接着，把 kernel 程序改成：

```
__global__ static void sumOfSquares(int *num, int* result,
    clock_t* time)
{
    const int tid = threadIdx.x;
    const int size = DATA_SIZE / THREAD_NUM;
    int sum = 0;
    int i;
    clock_t start;
    if(tid == 0) start = clock();
    for(i = tid * size; i < (tid + 1) * size; i++) {
      sum += num[i] * num[i];
    }

    result[tid] = sum;
    if(tid == 0) *time = clock() - start;
}
```

程序里的 threadIdx 是 CUDA 的一个内建的变量，表示目前的 thread 是第几个 thread（由 0 开始计算）。以我们的例子来说，会有 256 个 threads，所以同时会有 256 个 sumOfSquares 函式在执行，但每一个的 threadIdx.x 则分别会是 0 ~ 255。利用这个变量，我们就可以让每

一份函式执行时，对整个数据不同的部份计算平方和。另外，我们也让计算时间的动作，只在 thread 0（即 threadIdx.x = 0 的时候）进行。

同样的，由于会有 256 个计算结果，所以原来存放 result 的内存位置也要扩大。把 main 函式中的中间部份改成：

```
int* gpudata, *result;
clock_t* time;
cudaMalloc((void**) &gpudata, sizeof(int) * DATA_SIZE);
cudaMalloc((void**) &result, sizeof(int) * THREAD_NUM);
cudaMalloc((void**) &time, sizeof(clock_t));
cudaMemcpy(gpudata, data, sizeof(int) * DATA_SIZE,
    cudaMemcpyHostToDevice);

sumOfSquares<<<1, THREAD_NUM, 0>>>(gpudata, result, time);

int sum[THREAD_NUM];
clock_t time_used;
cudaMemcpy(&sum, result, sizeof(int) * THREAD_NUM,
    cudaMemcpyDeviceToHost);
cudaMemcpy(&time_used, time, sizeof(clock_t),
    cudaMemcpyDeviceToHost);
cudaFree(gpudata);
cudaFree(result);
cudaFree(time);
```

可以注意到我们在呼叫 sumOfSquares 函式时，指定 THREAD_NUM 为 thread 的数目。最后，在 CPU 端把计算好的各组数据的平方和进行加总：

```
int final_sum = 0;
for(int i = 0; i < THREAD_NUM; i++) {
    final_sum += sum[i];
}

printf("sum: %d   time: %d\n", final_sum, time_used);

final_sum = 0;
for(int i = 0; i < DATA_SIZE; i++) {
   sum += data[i] * data[i];
}
printf("sum (CPU): %d\n", final_sum);
```

编译后执行，确认结果和原来相同。

这个版本的程序，在 GeForce 8800GT 上执行，只需要约 8.3M cycles，比前一版程序快了 77 倍！这就是透过大量 thread 来隐藏 latency 所带来的效果。

不过，如果计算一下它使用的内存带宽，就会发现其实仍不是很理想，大约只有 723MB/s 而已。这和 GeForce 8800GT 所具有的内存带宽是很大的差距。为什么会这样呢？

内存的存取模式

显卡上的内存是 DRAM，因此最有效率的存取方式，是以连续的方式存取。前面的程序，虽然看起来是连续存取内存位置（每个 thread 对一块连续的数字计算平方和），但是我们要考

虑到实际上 thread 的执行方式。前面提过，当一个 thread 在等待内存的数据时，GPU 会切换到下一个 thread。也就是说，实际上执行的顺序是类似

thread 0 -> thread 1 -> thread 2 -> ...

因此，在同一个 thread 中连续存取内存，在实际执行时反而不是连续了。要让实际执行结果是连续的存取，我们应该要让 thread 0 读取第一个数字，thread 1 读取第二个数字…依此类推。所以，我们可以把 kernel 程序改成如下：

```
__global__ static void sumOfSquares(int *num, int* result,
    clock_t* time)
{
    const int tid = threadIdx.x;
    int sum = 0;
    int i;
    clock_t start;
    if(tid == 0) start = clock();
    for(i = tid; i < DATA_SIZE; i += THREAD_NUM) {
      sum += num[i] * num[i];
    }

    result[tid] = sum;
    if(tid == 0) *time = clock() - start;
}
```

编译后执行，确认结果相同。

仅仅是这样简单的修改，实际执行的效率就有很大的差别。在 GeForce 8800GT 上，上面的程序执行需要的频率是 2.6M cycles，又比前一版程序快了三倍。不过，这样仍只有 2.3GB/s 的带宽而已。

这是因为我们使用的 thread 数目还是不够多的原因。理论上 256 个 threads 最多只能隐藏 256 cycles 的 latency。但是 GPU 存取 global memory 时的 latency 可能高达 500 cycles 以上。如果增加 thread 数目，就可以看到更好的效率。例如，可以把 THREAD_NUM 改成 512。在 GeForce 8800GT 上，这可以让执行花费的时间减少到 1.95M cycles。有些改进，但是仍不够大。不幸的是，目前 GeForce 8800GT 一个 block 最多只能有 512 个 threads，所以不能再增加了，而且，如果 thread 数目增加太多，那么在 CPU 端要做的最后加总工作也会变多。

更多的并行化

前面提到了 block。在之前介绍呼叫 CUDA 函式时，也有提到 "block 数目" 这个参数。到目前为止，我们都只使用一个 block。究竟 block 是什么呢？

在 CUDA 中，thread 是可以分组的，也就是 block。一个 block 中的 thread，具有一个共享的 shared memory，也可以进行同步工作。不同 block 之间的 thread 则不行。在我们的程序中，其实不太需要进行 thread 的同步动作，因此我们可以使用多个 block 来进一步增加 thread 的数目。

首先，在 #define DATA_SIZE 的地方，改成如下：

```
#define DATA_SIZE    1048576
#define BLOCK_NUM    32
#define THREAD_NUM    256
```

这表示我们会建立 32 个 blocks，每个 blocks 有 256 个 threads，总共有 32*256 = 8192 个 threads。

接着，我们把 kernel 部份改成：

```
__global__ static void sumOfSquares(int *num, int* result,
    clock_t* time)
{
    const int tid = threadIdx.x;
    const int bid = blockIdx.x;
    int sum = 0;
    int i;
    if(tid == 0) time[bid] = clock();
    for(i = bid * THREAD_NUM + tid; i < DATA_SIZE;
        i += BLOCK_NUM * THREAD_NUM) {
      sum += num[i] * num[i];
    }

    result[bid * THREAD_NUM + tid] = sum;
    if(tid == 0) time[bid + BLOCK_NUM] = clock();
}
```

blockIdx.x 和 threadIdx.x 一样是 CUDA 内建的变量，它表示的是目前的 block 编号。另外，注意到我们把计算时间的方式改成每个 block 都会记录开始时间及结束时间。

main 函式部份，修改成：

```
    int* gpudata, *result;
    clock_t* time;
    cudaMalloc((void**) &gpudata, sizeof(int) * DATA_SIZE);
    cudaMalloc((void**) &result,
        sizeof(int) * THREAD_NUM * BLOCK_NUM);
    cudaMalloc((void**) &time, sizeof(clock_t) * BLOCK_NUM * 2);
    cudaMemcpy(gpudata, data, sizeof(int) * DATA_SIZE,
        cudaMemcpyHostToDevice);

    sumOfSquares<<<BLOCK_NUM, THREAD_NUM, 0>>>(gpudata, result,
        time);

    int sum[THREAD_NUM * BLOCK_NUM];
    clock_t time_used[BLOCK_NUM * 2];
    cudaMemcpy(&sum, result, sizeof(int) * THREAD_NUM * BLOCK_NUM,
        cudaMemcpyDeviceToHost);
    cudaMemcpy(&time_used, time, sizeof(clock_t) * BLOCK_NUM * 2,
        cudaMemcpyDeviceToHost);
    cudaFree(gpudata);
    cudaFree(result);
    cudaFree(time);

    int final_sum = 0;
    for(int i = 0; i < THREAD_NUM * BLOCK_NUM; i++) {
        final_sum += sum[i];
    }
```

```
    clock_t min_start, max_end;
    min_start = time_used[0];
    max_end = time_used[BLOCK_NUM];
    for(int i = 1; i < BLOCK_NUM; i++) {
        if(min_start > time_used[i])
            min_start = time_used[i];
        if(max_end < time_used[i + BLOCK_NUM])
            max_end = time_used[i + BLOCK_NUM];
    }

    printf("sum: %d   time: %d\n", final_sum, max_end - min_start);
```

基本上我们只是把 result 的大小变大，并修改计算时间的方式，把每个 block 最早的开始时间，和最晚的结束时间相减，取得总运行时间。

这个版本的程序，执行的时间减少很多，在 GeForce 8800GT 上只需要约 150K cycles，相当于 40GB/s 左右的带宽。不过，它在 CPU 上执行的部份，需要的时间加长了（因为 CPU 现在需要加总 8192 个数字）。为了避免这个问题，我们可以让每个 block 把自己的每个 thread 的计算结果进行加总。

Thread 的同步

前面提过，一个 block 内的 thread 可以有共享的内存，也可以进行同步。我们可以利用这一点，让每个 block 内的所有 thread 把自己计算的结果加总起来。把 kernel 改成如下：

```
__global__ static void sumOfSquares(int *num, int* result,
    clock_t* time)
{
    extern __shared__ int shared[];
    const int tid = threadIdx.x;
    const int bid = blockIdx.x;
    int i;
    if(tid == 0) time[bid] = clock();
    shared[tid] = 0;
    for(i = bid * THREAD_NUM + tid; i < DATA_SIZE;
        i += BLOCK_NUM * THREAD_NUM) {
      shared[tid] += num[i] * num[i];
    }

    __syncthreads();
    if(tid == 0) {
        for(i = 1; i < THREAD_NUM; i++) {
            shared[0] += shared[i];
        }
        result[bid] = shared[0];
    }

    if(tid == 0) time[bid + BLOCK_NUM] = clock();
}
```

利用 __shared__ 声明的变量表示这是 shared memory，是一个 block 中每个 thread 都共享的内存。它会使用在 GPU 上的内存，所以存取的速度相当快，不需要担心 latency 的问题。__syncthreads() 是一个 CUDA 的内部函数，表示 block 中所有的 thread 都要同步到这个点，才能继续执行。在我们的例子中，由于之后要把所有 thread 计算的结果进行加总，所以我们需要确定每个 thread 都已经把结果写到 shared[tid] 里面了。

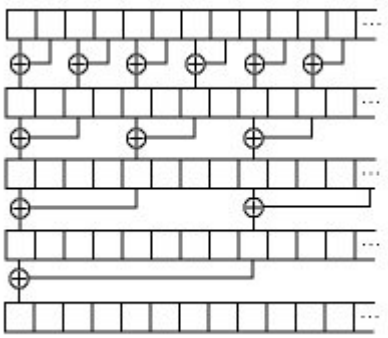
接下来，把 main 函式的一部份改成：

```
int* gpudata, *result;
clock_t* time;
cudaMalloc((void**) &gpudata, sizeof(int) * DATA_SIZE);
cudaMalloc((void**) &result, sizeof(int) * BLOCK_NUM);
cudaMalloc((void**) &time, sizeof(clock_t) * BLOCK_NUM * 2);
cudaMemcpy(gpudata, data, sizeof(int) * DATA_SIZE,
    cudaMemcpyHostToDevice);

sumOfSquares<<<BLOCK_NUM, THREAD_NUM,
    THREAD_NUM * sizeof(int)>>>(gpudata, result, time);

int sum[BLOCK_NUM];
clock_t time_used[BLOCK_NUM * 2];
cudaMemcpy(&sum, result, sizeof(int) * BLOCK_NUM,
    cudaMemcpyDeviceToHost);
cudaMemcpy(&time_used, time, sizeof(clock_t) * BLOCK_NUM * 2,
    cudaMemcpyDeviceToHost);
cudaFree(gpudata);
cudaFree(result);
cudaFree(time);

int final_sum = 0;
for(int i = 0; i < BLOCK_NUM; i++) {
    final_sum += sum[i];
}
```

可以注意到，现在 CPU 只需要加总 BLOCK_NUM 也就是 32 个数字就可以了。

不过，这个程序由于在 GPU 上多做了一些动作，所以它的效率会比较差一些。在 GeForce 8800GT 上，它需要约 164K cycles。

当然，效率会变差的一个原因是，在这一版的程序中，最后加总的工作，只由每个 block 的 thread 0 来进行，但这并不是最有效率的方法。理论上，把 256 个数字加总的动作，是可以并行化的。最常见的方法，是透过树状的加法：



把 kernel 改成如下：

```
__global__ static void sumOfSquares(int *num, int* result,
    clock_t* time)
{
    extern __shared__ int shared[];
    const int tid = threadIdx.x;
    const int bid = blockIdx.x;
    int i;
    int offset = 1, mask = 1;
    if(tid == 0) time[bid] = clock();
    shared[tid] = 0;
    for(i = bid * THREAD_NUM + tid; i < DATA_SIZE;
        i += BLOCK_NUM * THREAD_NUM) {
      shared[tid] += num[i] * num[i];
    }

    __syncthreads();
    while(offset < THREAD_NUM) {
        if((tid & mask) == 0) {
            shared[tid] += shared[tid + offset];
        }
        offset += offset;
        mask = offset + mask;
        __syncthreads();
    }

    if(tid == 0) {
        result[bid] = shared[0];
        time[bid + BLOCK_NUM] = clock();
    }
}
```

后面的 while 循环就是进行树状加法。main 函式则不需要修改。

这一版的程序，在 GeForce 8800GT 上执行需要的时间，大约是 140K cycles（相当于约 43GB/s），比完全不在 GPU 上进行加总的版本还快！这是因为，在完全不在 GPU 上进行

加总的版本，写入到 global memory 的数据数量很大（8192 个数字），也对效率会有影响。所以，这一版程序不但在 CPU 上的运算需求降低，在 GPU 上也能跑的更快。

进一步改善

上一个版本的树状加法是一般的写法，但是它在 GPU 上执行的时候，会有 share memory 的 bank conflict 的问题（详情在后面介绍 GPU 架构时会提到）。采用下面的方法，可以避免这个问题：

```
offset = THREAD_NUM / 2;
while(offset > 0) {
    if(tid < offset) {
        shared[tid] += shared[tid + offset];
    }
    offset >>= 1;
    __syncthreads();
}
```

这样同时也省去了 mask 变数。因此，这个版本的执行的效率就可以再提高一些。在 GeForce 8800GT 上，这个版本执行的时间是约 137K cycles。当然，这时差别已经很小了。如果还要再提高效率，可以把树状加法整个展开：

```
if(tid < 128) { shared[tid] += shared[tid + 128]; }
__syncthreads();
if(tid < 64) { shared[tid] += shared[tid + 64]; }
__syncthreads();
if(tid < 32) { shared[tid] += shared[tid + 32]; }
__syncthreads();
if(tid < 16) { shared[tid] += shared[tid + 16]; }
__syncthreads();
if(tid < 8) { shared[tid] += shared[tid + 8]; }
__syncthreads();
if(tid < 4) { shared[tid] += shared[tid + 4]; }
__syncthreads();
if(tid < 2) { shared[tid] += shared[tid + 2]; }
__syncthreads();
if(tid < 1) { shared[tid] += shared[tid + 1]; }
__syncthreads();
```

当然这只适用于 THREAD_NUM 是 256 的情形。这样可以再省下约 1000 cycles 左右（约 44GB/s）。最后完整的程序文件可以从这里下载。

第二个 CUDA程序

前面介绍的计算平方和的程序，似乎没有什么实用价值。所以我们的第二个 CUDA 程序，要做一个确实有（某些）实用价值的程序，也就是进行矩阵乘法。而且，这次我们会使用浮点数。

虽然矩阵乘法有点老套，不过因为它相当简单，而且也可以用来介绍一些有关 CUDA 的有趣性质。

矩阵乘法

为了单纯起见，我们这里以方形的矩阵为例子。基本上，假设有两个矩阵 A 和 B，则计算 AB = C 的方法如下：

```
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            C[i][j] = 0;
            for(k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
```

一开始，我们先准备好产生数据、设定 CUDA 等等的工作：

```
    int main()
    {
        float *a, *b, *c, *d;
        int n = 1000;

        if(!InitCUDA()) return 0;

        a = (float*) malloc(sizeof(float) * n * n);
        b = (float*) malloc(sizeof(float) * n * n);
        c = (float*) malloc(sizeof(float) * n * n);
        d = (float*) malloc(sizeof(float) * n * n);

        srand(0);

        matgen(a, n, n);
        matgen(b, n, n);

        clock_t time = matmultCUDA(a, n, b, n, c, n, n);

        matmult(a, n, b, n, d, n, n);
        compare_mat(c, n, d, n, n);

        double sec = (double) time / CLOCKS_PER_SEC;
        printf("Time used: %.2f (%.2lf GFLOPS)\n", sec,
            2.0 * n * n * n / (sec * 1E9));

        return 0;
    }
```

InitCUDA 函式和第一个 CUDA 程序一样，可以直接参考前面的文章。以下是上面用到的一些其它的函式：

产生矩阵：

```
    void matgen(float* a, int lda, int n)
    {
        int i, j;

        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
```

```
            a[i * lda + j] = (float) rand() / RAND_MAX +
                (float) rand() / (RAND_MAX * RAND_MAX);
        }
    }
}
```

这个函式只是利用随机数生成器把矩阵填满 0 ~ 1 之间的数字。特别注意到因为 C 语言中无法声明变动大小的二维矩阵，所以我们使用 i * lda + j 的方式。

进行矩阵乘法：

```
void matmult(const float* a, int lda, const float* b, int ldb,
    float* c, int ldc, int n)
{
    int i, j, k;

    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            double t = 0;
            for(k = 0; k < n; k++) {
                t += a[i * lda + k] * b[k * ldb + j];
            }
            c[i * ldc + j] = t;
        }
    }
}
```

这是以 CPU 进行矩阵乘法、用来进行验证答案正确与否的程序。特别注意到它用 double 来储存暂时的计算结果，以提高精确度。

验证结果：

```
void compare_mat(const float* a, int lda,
    const float* b, int ldb, int n)
{
    float max_err = 0;
     float average_err = 0;
    int i, j;

    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            if(b[i * ldb + j] != 0) {
                float err = fabs((a[i * lda + j] -
                    b[i * ldb + j]) / b[i * ldb + j]);
                if(max_err < err) max_err = err;
                average_err += err;
            }
        }
    }

    printf("Max error: %g Average error: %g\n",
```

```
        max_err, average_err / (n * n));
}
```

这个函式计算两个矩阵的最大相对误差和平均相对误差，并把结果印出来。

最后是 CUDA 的矩阵乘法的部份：

```
#define NUM_THREADS 256

clock_t matmultCUDA(const float* a, int lda,
    const float* b, int ldb, float* c, int ldc, int n)
{
    float *ac, *bc, *cc;
    clock_t start, end;

    start = clock();
    cudaMalloc((void**) &ac, sizeof(float) * n * n);
    cudaMalloc((void**) &bc, sizeof(float) * n * n);
    cudaMalloc((void**) &cc, sizeof(float) * n * n);

    cudaMemcpy2D(ac, sizeof(float) * n, a, sizeof(float) * lda,
        sizeof(float) * n, n, cudaMemcpyHostToDevice);
    cudaMemcpy2D(bc, sizeof(float) * n, b, sizeof(float) * ldb,
        sizeof(float) * n, n, cudaMemcpyHostToDevice);

    int blocks = (n + NUM_THREADS - 1) / NUM_THREADS;
    matMultCUDA<<<blocks * n, NUM_THREADS>>>
        (ac, n, bc, n, cc, n, n);

    cudaMemcpy2D(c, sizeof(float) * ldc, cc, sizeof(float) * n,
    sizeof(float) * n, n, cudaMemcpyDeviceToHost);

    cudaFree(ac);
    cudaFree(bc);
    cudaFree(cc);

    end = clock();

    return end - start;
}
```

这个函式相当单纯，就是在显卡内存中配置存放矩阵的内存，然后把主内存中的矩阵数据复制到显卡内存上。不过，因为我们的矩阵乘法函式可以指定 pitch（即 lda、ldb、和 ldc），所以如果用一般的 cudaMemcpy 函式来复制内存的话，会需要每个 row 都分开复制，那会需要呼叫很多次 cudaMemcpy 函式，会使效率变得很差。因此，在这里我们用了一个新的 cudaMemcpy2D 函式，它是用来复制二维数组，可以指定数组的 pitch。这样就可以透过一次函数调用就可以了。

进行计算的 kernel 如下：

```
__global__ static void matMultCUDA(const float* a, size_t lda,
    const float* b, size_t ldb, float* c, size_t ldc, int n)
```

```
{
    const int tid = threadIdx.x;
    const int bid = blockIdx.x;
    const int idx = bid * blockDim.x + tid;
    const int row = idx / n;
    const int column = idx % n;
    int i;

    if(row < n && column < n) {
        float t = 0;
        for(i = 0; i < n; i++) {
            t += a[row * lda + i] * b[i * ldb + column];
        }
        c[row * ldc + column] = t;
    }
}
```

这个函式一开始先从 bid 和 tid 计算出这个 thread 应该计算的 row 和 column，在判断 row 和 column 在范围内之后，就直接进行计算，并把结果写到 c 矩阵中，是非常单纯的函式。在 GeForce 8800GT 上实际执行的结果如下：

Max error: 2.01484e-006 Average error: 3.36637e-007

Time used: 1.1560 (1.73 GFLOPS)

可以看到两个问题：

1. 很明显的，执行效率相当低落。
2. 最大相对误差偏高。理想上应该要低于 1e-6。

计算结果的误差偏高的原因是，在 CPU 上进行计算时，我们使用 double（即 64 bits 浮点数）来累进计算过程，而在 GPU 上则只能用 float（32 bits 浮点数）。在累加大量数字的时候，由于累加结果很快会变大，因此后面的数字很容易被舍去过多的位数。

由于 CUDA 的浮点数运算，在进行加、减、乘法时是符合 IEEE 754 规定的精确度的，因此，我们可以利用 Kahan's Summation Formula 来提高精确度。把程序改成：

```
if(row < n && column < n) {
    float t = 0;
    float y = 0;
    for(i = 0; i < n; i++) {
        float r;
        y -= a[row * lda + i] * b[i * ldb + column];
        r = t - y;
        y = (r - t) + y;
        t = r;
    }
}
```

修改后的程序的执行结果是：

Max error: 1.19209e-007 Average error: 4.22751e-008

Time used: 1.1560 (1.73 GFLOPS)

可以看到相对误差有很大的改善，效率则没什么变化。

由于 Kahan's Summation Formula 需要的运算量提高，但是效率却没有什么改变，可以看出这个 kernel 主要的瓶颈应该是在内存的存取动作上。这是因为有大量的内存读取是重复的。例如，矩阵 a 的一个 row 在每次进行计算时都被重复读入，但这是相当浪费的。这样的计算方式，总共需要读取 $2*n^3$ 次内存。如果让一个 row 只需要读入一次的话，就可以减到为 n^3+n^2 次。

第一个改良

和我们的第一个 CUDA 程序一样，我们可以利用 shared memory 来储存每个 row 的数据。不过，因为只有同一个 block 的 threads 可以共享 shared memory，因此现在一个 row 只能由同一个 block 的 threads 来进行计算。另外我们也需要能存放一整个 row 的 shared memory。因此，把先把呼叫 kernel 的部份改成：

```
matMultCUDA<<<n, NUM_THREADS, sizeof(float) * n>>>
    (ac, n, bc, n, cc, n, n);
```

kernel 的部份则改成：

```
__global__ static void matMultCUDA(const float* a, size_t lda,
    const float* b, size_t ldb, float* c, size_t ldc, int n)
{
    extern __shared__ float data[];
    const int tid = threadIdx.x;
    const int row = blockIdx.x;
    int i, j;

    for(i = tid; i < n; i += blockDim.x) {
        data[i] = a[row * lda + i];
    }

    __syncthreads();

    for(j = tid; j < n; j += blockDim.x) {
        float t = 0;
        float y = 0;
        for(i = 0; i < n; i++) {
            float r;
            y -= data[i] * b[i * ldb + j];
            r = t - y;
            y = (r - t) + y;
            t = r;
        }
        c[row * ldc + j] = t;
    }
}
```

第一个部份先把整个 row 读到 shared memory 中，而第二个部份则进行计算，并没有太大的变化。主要的差别是现在一个 row 只由一个 block 进行计算。

在 GeForce 8800GT 上，执行的结果是：

Max error: 1.19209e-007 Average error: 4.22751e-008

Time used: 0.4220 (4.74 GFLOPS)

很明显的，计算的结果并没有改变，不过速度则提高了超过一倍。虽然如此，但是这样的效率仍不尽理想，因为理论上 GeForce 8800GT 有超过 300GFLOPS 的运算性能。即使是把 Kahan's Summation Formula 所需要的额外运算考虑进去，这样的效率仍然连理论最大值的十分之一都不到。

会有这样的结果，原因其实还是同样的：对内存的存取次数太多了。虽然现在 A 矩阵的 row 的数据已经不再需要重复读取，但是 B 矩阵的 column 的数据仍然一直被重复读取。

另一个问题比较不是那么明显：对 B 矩阵的读取，虽然看起来不连续，但实际上它是连续的。这是因为不同的 thread 会读取不同的 column，因此同时间每个 thread 读取的各个 column 加起来，就是一个连续的内存区块。那么，为什么效率还是不佳呢？这是因为，GPU 上的内存控制器，从某个固定的倍数地址开始读取，才会有最高的效率（例如 16 bytes 的倍数）。由于矩阵大小并不是 16 的倍数（这里使用的是 1000x1000 的矩阵），所以造成效率不佳的情形。

要解决这个问题，我们可以在 cudaMalloc 的时候稍微修改一下，让宽度变成 适当的倍数就可以了。但是，适当的倍数是多少呢？幸运的是，我们并不需要知道这些细节。CUDA 提供了一个 cudaMallocPitch 的函式，可以自动以最佳的倍数来配置内存。因此，我们可以把 cudaMalloc 的部份改成：

```
size_t pitch_a, pitch_b, pitch_c;
cudaMallocPitch((void**) &ac, &pitch_a, sizeof(float) * n, n);
cudaMallocPitch((void**) &bc, &pitch_b, sizeof(float) * n, n);
cudaMallocPitch((void**) &cc, &pitch_c, sizeof(float) * n, n);
```

cudaMallocPitch 函式会以适当的倍数配置内存，并把配置的宽度传回。因此，在把矩阵复制到显卡内存上时，要使用它传回的宽度：

```
cudaMemcpy2D(ac, pitch_a, a, sizeof(float) * lda,
    sizeof(float) * n, n, cudaMemcpyHostToDevice);
cudaMemcpy2D(bc, pitch_b, b, sizeof(float) * ldb,
    sizeof(float) * n, n, cudaMemcpyHostToDevice);
```

呼叫 kernel 的部份也需要修改：

```
matMultCUDA<<<n, NUM_THREADS, sizeof(float) * n>>>
    (ac, pitch_a / sizeof(float), bc, pitch_b / sizeof(float),
    cc, pitch_c / sizeof(float), n);
```

同样的，把计算结果复制回到主内存时，也要使用传回的宽度值：

```
cudaMemcpy2D(c, sizeof(float) * ldc, cc, pitch_c,
    sizeof(float) * n, n, cudaMemcpyDeviceToHost);
```

这样就完成了。Kernel 部份则不需要修改。

这样的修改有多大的效果呢？在 GeForce 8800GT 上执行，结果如下：

Max error: 1.19209e-007 Average error: 4.22751e-008

Time used: 0.1250 (16.00 GFLOPS)

可以看到，执行速度又再大幅提高了三倍多！而这只是把内存的配置方式稍微修改一下而已。虽然执行速度提高了很多，但是，和前面提到的理论值相比，其实还是有相当的差距。这是因为，前面也提到过，这样的做法需要 n^3+n^2 次的内存读取，和 n^2 次的内存写入动作。由于 n = 1000，每个数字的大小是 32 bits，所以总共的内存存取数据量约为 4GB。除以实际执行的时间 0.125 秒，得到的带宽数值是约 32GB/s，这已经接近 GeForce 8800GT 显卡内存的带宽了。由于我们计算时间的时候，把配置内存、以及数据的复制动作也计算进去，因此实际上

花费在 kernel 的时间是更短的（约 0.09 秒）。因此，可以很明显的看出，这个程序的效率，是受限于内存带宽的。

进一步的改良

上一节的结论显示出，矩阵乘法的程序，效率是受限于内存带宽的。那有没有办法降低内存的存取次数呢？答案当然是有的，不然就不会有这一节了 :)

要进一步降低内存带宽的使用，可以注意到，在上一节的方法中，虽然 A 矩阵的存取次数被减至最低，但是 B 矩阵的存取次数并没有减少。这是因为我们只将 A 矩阵的 row 加载到 shared memory 中，但是 B 矩阵的 column 也是有被重复使用的。理想上应该也可以避免重复加载才对。不过，由于 B 矩阵的 column 使用的时机，和 A 矩阵的 row 是不同的，所以并不能直接这样做。

解决方法是 "blocking"。也就是把整个矩阵乘法的动作，切割成很多小矩阵的乘法。例如，要计算 C 矩阵的 (0, 0) ~ (15, 15) 的值，可以把它想成：

A(0~15, 0~15) * B(0~15, 0~15) + A(0~15,16~31) * B(16~31, 0~15)
+ A(0~15, 32~47) * B(32~47, 0~15) + ...

这样一来，我们就可以把两个小矩阵加载到 shared memory，则小矩阵本身的乘法就不需要再存取任何外部的内存了！这样一来，假设小矩阵的大小是 k，则实际上需要的内存存取次数就会变成约 $2k^2(n/k)^3 = 2n^3/k$。

由于目前 CUDA 每个 block 的 thread 数目最多是 512，因此 k = 16 似乎是一个相当理想的数字（共 256 个 threads）。因此，对于一个 n = 1000 的矩阵来说，我们可以把内存存取的量减少到约 500MB，也就是上一节的存取量的 1/8。理论上，这样应该可以让效率提高八倍（假设没有遇到别的瓶颈）。

为了方便进行区块的计算，我们让每个 block 有 16x16 个 threads，再建立 (n/16)x(n/16) 个 blocks。把呼叫 kernel 的地方改成：

```
int bx = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 blocks(bx, bx);
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
matMultCUDA<<<blocks, threads>>>(ac, pitch_a / sizeof(float),
    bc, pitch_b / sizeof(float), cc, pitch_c / sizeof(float), n);
```

BLOCK_SIZE 则是定义成 16。dim3 是 CUDA 的一种数据型态，表示一个 3D 的向量。在这里，我们透过 dim3 来建立 16x16 个 threads 的 block，和 (n/16)x(n/16) 个 blocks。

Kernel 程序的部份，则改成：

```
__global__ static void matMultCUDA(const float* a, size_t lda,
    const float* b, size_t ldb, float* c, size_t ldc, int n)
{
    __shared__ float matA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float matB[BLOCK_SIZE][BLOCK_SIZE];
    const int tidc = threadIdx.x;
    const int tidr = threadIdx.y;
    const int bidc = blockIdx.x * BLOCK_SIZE;
    const int bidr = blockIdx.y * BLOCK_SIZE;
    int i, j;

    float results = 0;
    float comp = 0;
```

```
    for(j = 0; j < n; j += BLOCK_SIZE) {
        if(tidr + bidr < n && tidc + j < n) {
            matA[tidr][tidc] = a[(tidr + bidr) * lda + tidc + j];
        }
        else {
            matA[tidr][tidc] = 0;
        }

        if(tidr + j < n && tidc + bidc < n) {
            matB[tidr][tidc] = b[(tidr + j) * ldb + tidc + bidc];
        }
        else {
            matB[tidr][tidc] = 0;
        }

        __syncthreads();

        for(i = 0; i < BLOCK_SIZE; i++) {
            float t;
            comp -= matA[tidr][i] * matB[i][tidc];
            t = results - comp;
            comp = (t - results) + comp;
            results = t;
        }

        __syncthreads();
    }

    if(tidr + bidr < n && tidc + bidc < n) {
        c[(tidr + bidr) * ldc + tidc + bidc] = results;
    }
}
```

注意到因为我们现在使用 16x16 的 threads，因此 threadIdx 变量可以取得 threadIdx.x 和 threadIdx.y，范围分别是 0 ~ 15。blockIdx.x 和 blockIdx.y 变量也是同样的情形，范围分别是 0 ~ n/16。

在程序中，因为矩阵的大小不一定会是 16 的倍数，因此需要使用 if 判断式检查是否超出矩阵范围。

这个版本在 GeForce 8800GT 上的执行结果如下：

Max error: 1.19209e-007 Average error: 4.22751e-008

Time used: 0.0780 (25.64 GFLOPS)

速度虽然提高了，但是似乎并没有达到预期中的八倍。当然，前面提到过，我们在计算时间时，把一些复制内存、配置内存的动作也计算在内，这些动作的时间并不会缩短。实际上 kernel 的运行时间，大约是 0.053 秒左右（约略相当于 38GFLOPS），比上一节的版本快了将近一倍。

如果这一版程序已经不再限于内存带宽，那为什么没有达到预期的效率呢？这是因为这一版程序已经是限于指令周期了。除了使用 Kahan's Summation Formula 会需要更多的运算之外，

程序中也有大量计算矩阵地址的乘法等等，这都会需要花费运算资源。另外，那些用来判断超出矩阵范围的 if 判断式，也会有一定的影响。

要把那些 if 判断式去掉，有一个方法是，在配置内存时，就配置成 16 的倍数，并在复制矩阵到显卡内存之前，先将它清为 0。如下所示：

```
int newn = ((n + BLOCK_SIZE - 1) / BLOCK_SIZE) * BLOCK_SIZE;

cudaMallocPitch((void**) &ac, &pitch_a,
    sizeof(float) * newn, newn);
cudaMallocPitch((void**) &bc, &pitch_b,
    sizeof(float) * newn, newn);
cudaMallocPitch((void**) &cc, &pitch_c,
    sizeof(float) * newn, newn);

cudaMemset(ac, 0, pitch_a * newn);
cudaMemset(bc, 0, pitch_b * newn);
```

这样一来，我们就可以把 kernel 中的 if 判断式都移除了：

```
__global__ static void matMultCUDA(const float* a, size_t lda,
    const float* b, size_t ldb, float* c, size_t ldc, int n)
{
    __shared__ float matA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float matB[BLOCK_SIZE][BLOCK_SIZE];
    const int tidc = threadIdx.x;
    const int tidr = threadIdx.y;
    const int bidc = blockIdx.x * BLOCK_SIZE;
    const int bidr = blockIdx.y * BLOCK_SIZE;
    int i, j;

    float results = 0;
    float comp = 0;

    for(j = 0; j < n; j += BLOCK_SIZE) {
        matA[tidr][tidc] = a[(tidr + bidr) * lda + tidc + j];
       matB[tidr][tidc] = b[(tidr + j) * ldb + tidc + bidc];

       __syncthreads();

        for(i = 0; i < BLOCK_SIZE; i++) {
            float t;
           comp -= matA[tidr][i] * matB[i][tidc];
           t = results - comp;
            comp = (t - results) + comp;
            results = t;
       }

        __syncthreads();
    }
```

```
    c[(tidr + bidr) * ldc + tidc + bidc] = results;
}
```

这个版本的执行结果是：

```
Max error: 1.19209e-007  Average error: 4.22751e-008
Time used: 0.0780  (25.64 GFLOPS)
```

似乎没有改善。不过，实际上 kernel 的运行时间已经减少到 0.042 秒（约略相当于 48GFLOPS）。

结论

有些读者可能会想，如果把 block 再变得更大（例如 32x32）是否会有帮助呢？当然，由于最后的程序已经不再是受限于内存带宽（在 0.042 秒内存取 500MB 的数据约相当于 12GB/s 的带宽），所以把 block 再加大并不会有帮助了。而且，由于一个 block 内的 thread 数目最多只能到 512 个，将 block 变大也会造成很多额外负担。而且 shared memory 的大小也有限制（GeForce 8800GT 的 shared memory 大小限制是 16384 bytes），所以也不能任意增加 block 的大小。

最后一版程序的完整档案可以从这里下载。

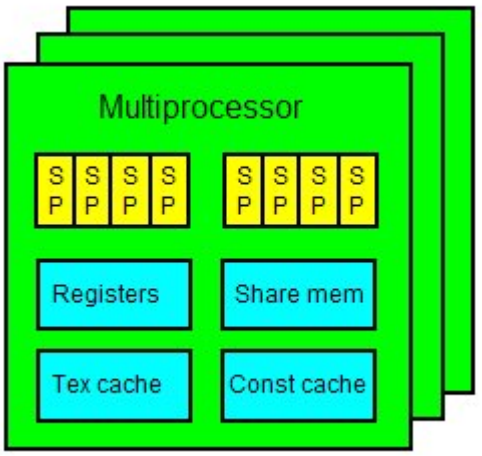
GPU 的硬件架构

这里我们会简单介绍，NVIDIA 目前支持 CUDA 的 GPU，其在执行 CUDA 程序的部份（基本上就是其 shader 单元）的架构。这里的数据是综合 NVIDIA 所公布的信息，以及 NVIDIA 在各个研讨会、学校课程等所提供的数据，因此有可能会有不正确的地方。主要的数据源包括 NVIDIA 的 CUDA Programming Guide 1.1、NVIDIA 在 Supercomputing '07 介绍 CUDA 的 session，以及 UIUC 的 CUDA 课程。

GPU 的基本介绍

目前 NVIDIA 推出的显示芯片，支持 CUDA 的是 G80 系列的显示芯片。其中 G80 显示芯片支持 CUDA 1.0 版，而 G84、G86、G92、G94、G96 则支援 CUDA 1.1 版。基本上，除了最早的 GeForce 8800 Ultra/GTX 及 320MB/640MB 版本的 GeForce 8800GTS、Tesla 等显卡是 CUDA 1.0 版之外，其它 GeForce 8 系列及 9 系列显卡都支持 CUDA 1.1。详细情形可以参考 CUDA Programming Guide 1.1 的 Appendix A。

所有目前支持 CUDA 的 NVIDIA 显示芯片，其 shader 部份都是由多个 **multiprocessors** 组成。每个 multiprocessor 里包含了八个 **stream processors**，其组成是四个四个一组，也就是说实际上可以看成是有两组 4D 的 SIMD 处理器。此外，每个 multiprocessor 还具有 8192 个寄存器，16KB 的 share memory，以及 texture cache 和 constant cache。大致上如下图所示：



详细的 multiprocessor 信息，都可以透过 CUDA 的 cudaGetDeviceProperties() 函式或 cuDeviceGetProperties() 函式取得。不过，目前还没有办法直接取得一个显示芯片中有多少 multiprocessor 的信息。

在 CUDA 中，大部份基本的运算动作，都可以由 stream processor 进行。每个 stream processor 都包含一个 FMA（fused-multiply-add）单元，可以进行一个乘法和一个加法。比较复杂的运算则会需要比较长的时间。

执行过程

在执行 CUDA 程序的时候，每个 stream processor 就是对应一个 thread。每个 multiprocessor 则对应一个 block。从之前的文章中，可以注意到一个 block 经常有很多个 thread（例如 256 个），远超过一个 multiprocessor 所有的 stream processor 数目。这又是怎么回事呢？

实际上，虽然一个 multiprocessor 只有八个 stream processor，但是由于 stream processor 进行各种运算都有 latency，更不用提内存存取的 latency，因此 CUDA 在执行程序的时候，是以 **warp** 为单位。目前的 CUDA 装置，一个 warp 里面有 32 个 threads，分成两组 16 threads 的 half-warp。由于 stream processor 的运算至少有 4 cycles 的 latency，因此对一个 4D 的 stream processors 来说，一次至少执行 16 个 threads（即 half-warp）才能有效隐藏各种运算的 latency。

由于 multiprocessor 中并没有太多别的内存，因此每个 thread 的状态都是直接保存在 multiprocessor 的寄存器中。所以，如果一个 multiprocessor 同时有愈多的 thread 要执行，就会需要愈多的寄存器空间。例如，假设一个 block 里面有 256 个 threads，每个 thread 用到 20 个寄存器，那么总共就需要 256x20 = 5,120 个寄存器才能保存每个 thread 的状态。

目前 CUDA 装置中每个 multiprocessor 有 8,192 个寄存器，因此，如果每个 thread 使用到 16 个寄存器，那就表示一个 multiprocessor 同时最多只能维持 512 个 thread 的执行。如果同时进行的 thread 数目超过这个数字，那么就会需要把一部份的数据储存在显卡内存中，就会降低执行的效率了。

编者注：在 NVIDIA GT200 中的 Register File 大小增加了一倍，在 FP32 下可用的 register file 为 16K，FP64 下是 8K。

Shared memory

目前 CUDA 装置中，每个 multiprocessor 有 16KB 的 shared memory。Shared memory 分成 16 个 bank。如果同时每个 thread 是存取不同的 bank，就不会产生任何问题，存取 shared memory 的速度和存取寄存器相同。不过，如果同时有两个（或更多个） threads 存取同一个 bank 的数据，就会发生 bank conflict，这些 threads 就必须照顺序去存取，而无法同时存取 shared memory 了。

Shared memory 是以 4 bytes 为单位分成 banks。因此，假设以下的数据：

```
__shared__ int data[128];
```

那么，data[0] 是 bank 0、data[1] 是 bank 1、data[2] 是 bank 2、…、data[15] 是 bank 15，而 data[16] 又回到 bank 0。由于 warp 在执行时是以 half-warp 的方式执行，因此分属于不同的 half warp 的 threads，不会造成 bank conflict。

因此，如果程序在存取 shared memory 的时候，使用以下的方式：

```
int number = data[base + tid];
```

那就不会有任何 bank conflict，可以达到最高的效率。但是，如果是以下的方式：

```
int number = data[base + 4 * tid];
```

那么，thread 0 和 thread 4 就会存取到同一个 bank，thread 1 和 thread 5 也是同样，这样就会造成 bank conflict。在这个例子中，一个 half warp 的 16 个 threads 会有四个 threads 存取同一个 bank，因此存取 share memory 的速度会变成原来的 1/4。

一个重要的例外是，当多个 thread 存取到同一个 shared memory 的地址时，shared memory 可以将这个地址的 32 bits 数据「广播」到所有读取的 threads，因此不会造成 bank conflict。例如：

```
int number = data[3];
```

这样不会造成 bank conflict，因为所有的 thread 都读取同一个地址的数据。

很多时候 shared memory 的 bank conflict 可以透过修改数据存放的方式来解决。例如，以下的程序：

```
data[tid] = global_data[tid];
...
int number = data[16 * tid];
```

会造成严重的 bank conflict，为了避免这个问题，可以把数据的排列方式稍加修改，把存取方式改成：

```
int row = tid / 16;
int column = tid % 16;
```

```
data[row * 17 + column] = global_data[tid];
...
int number = data[17 * tid];
```

这样就不会造成 bank conflict 了。

编者注：share memory 在 NVIDIA 的文档中其实还有不同的叫法，例如 PDC（Parallel Data Cache）、PBSM（per-block share memory）。

Global memory

由于 multiprocessor 并没有对 global memory 做 cache（如果每个 multiprocessor 都有自己的 global memory cache，将会需要 cache coherence protocol，会大幅增加 cache 的复杂度），所以 global memory 存取的 latency 非常的长。除此之外，前面的文章中也提到过 global memory 的存取，要尽可能的连续。这是因为 DRAM 存取的特性所造成的结果。

更精确的说，global memory 的存取，需要是 "coalesced"。所谓的 coalesced，是表示除了连续之外，而且它开始的地址，必须是每个 thread 所存取的大小的 16 倍。例如，如果每个 thread 都读取 32 bits 的数据，那么第一个 thread 读取的地址，必须是 16*4 = 64 bytes 的倍数。

如果有一部份的 thread 没有读取内存，并不会影响到其它的 thread 速行 coalesced 的存取。例如：

```
if(tid != 3) {
    int number = data[tid];
}
```

虽然 thread 3 并没有读取数据，但是由于其它的 thread 仍符合 coalesced 的条件（假设 data 的地址是 64 bytes 的倍数），这样的内存读取仍会符合 coalesced 的条件。

在目前的 CUDA 1.1 装置中，每个 thread 一次读取的内存数据量，可以是 32 bits、64 bits、或 128 bits。不过，32 bits 的效率是最好的。64 bits 的效率会稍差，而一次读取 128 bits 的效率则比一次读取 32 bits 要显著来得低（但仍比 non-coalesced 的存取要好）。

如果每个 thread 一次存取的数据并不是 32 bits、64 bits、或 128 bits，那就无法符合 coalesced 的条件。例如，以下的程序：

```
struct vec3d { float x, y, z; };
...
__global__ void func(struct vec3d* data, float* output)
{
        output[tid] = data[tid].x * data[tid].x +
         data[tid].y * data[tid].y +
         data[tid].z * data[tid].z;
}
```

并不是 coalesced 的读取，因为 vec3d 的大小是 12 bytes，而非 4 bytes、8 bytes、或 16 bytes。要解决这个问题，可以使用 __align(n)__ 的指示，例如：

```
struct __align__(16) vec3d { float x, y, z; };
```

这会让 compiler 在 vec3d 后面加上一个空的 4 bytes，以补齐 16 bytes。另一个方法，是把数据结构转换成三个连续的数组，例如：

```
__global__ void func(float* x, float* y, float* z, float* output)
{
    output[tid] = x[tid] * x[tid] + y[tid] * y[tid] +
        z[tid] * z[tid];
}
```

如果因为其它原因使数据结构无法这样调整，也可以考虑利用 shared memory 在 GPU 上做结构的调整。例如：

```
__global__ void func(struct vec3d* data, float* output)
{
    __shared__ float temp[THREAD_NUM * 3];
    const float* fdata = (float*) data;
    temp[tid] = fdata[tid];
    temp[tid + THREAD_NUM] = fdata[tid + THREAD_NUM];
    temp[tid + THREAD_NUM*2] = fdata[tid + THREAD_NUM*2];
    __syncthreads();
    output[tid] = temp[tid*3] * temp[tid*3] +
        temp[tid*3+1] * temp[tid*3+1] +
        temp[tid*3+2] * temp[tid*3+2];
}
```

在上面的例子中，我们先用连续的方式，把数据从 global memory 读到 shared memory。由于 shared memory 不需要担心存取顺序（但要注意 bank conflict 问题，参照前一节），所以可以避开 non-coalesced 读取的问题。

Texture

CUDA 支援 texture。在 CUDA 的 kernel 程序中，可以利用显示芯片的 texture 单元，读取 texture 的数据。使用 texture 和 global memory 最大的差别在于 texture 只能读取，不能写入，而且显示芯片上有一定大小的 texture cache。因此，读取 texture 的时候，不需要符合 coalesced 的规则，也可以达到不错的效率。此外，读取 texture 时，也可以利用显示芯片中的 texture filtering 功能（例如 bilinear filtering），也可以快速转换数据型态，例如可以直接将 32 bits RGBA 的数据转换成四个 32 bits 浮点数。

显示芯片上的 texture cache 是针对一般绘图应用所设计，因此它仍最适合有区块性质的存取动作，而非随机的存取。因此，同一个 warp 中的各个 thread 最好是读取地址相近的数据，才能达到最高的效率。

对于已经能符合 coalesced 规则的数据，使用 global memory 通常会比使用 texture 要来得快。

运算单元

Stream processor 里的运算单元，基本上是一个浮点数的 fused multiply-add 单元，也就是说它可以进行一次乘法和一次加法，如下所示：

```
a = b * c + d;
```

compiler 会自动把适当的加法和乘法运算，结合成一个 fmad 指令。

除了浮点数的加法及乘法之外，整数的加法、位运算、比较、取最小值、取最大值、及以型态的转换（浮点数转整数或整数转浮点数）都是可以全速进行的。整数的乘法则无法全速进行，但 24 bits 的乘法则可以。在 CUDA 中可以利用内建的 __mul24 和 __umul24 函式来进行 24 bits 的整数乘法。

浮点数的除法是利用先取倒数，再相乘的方式计算，因此精确度并不能达到 IEEE 754 的规范（最大误差为 2 ulp）。内建的 __fdividef(x,y) 提供更快速的除法，和一般的除法有相同的精确度，但是在 $2^{216} < y < 2^{218}$ 时会得到错误的结果。

此外 CUDA 还提供了一些精确度较低的内部函数，包括 __expf、__logf、__sinf、__cosf、__powf 等等。这些函式的速度较快，但精确度不如标准的函式。详细的数据可以参考 CUDA Programming Guide 1.1 的 Appendix B。

和主内存间的数据传输

在 CUDA 中，GPU 不能直接存取主内存，只能存取显卡上的显示内存。因此，会需要将数据从主内存先复制到显卡内存中，进行运算后，再将结果从显卡内存中复制到主内存中。这些复制的动作会限于 PCI Express 的速度。使用 PCI Express x16 时，PCI Express 1.0 可以提供双向各 4GB/s 的带宽，而 PCI Express 2.0 则可提供 8GB/s 的带宽。当然这都是理论值。

从一般的内存复制数据到显卡内存的时候，由于一般的内存可能随时会被操作系统搬动，因此 CUDA 会先将数据复制到一块内部的内存中，才能利用 DMA 将数据复制到显卡内存中。如果想要避免这个重复的复制动作，可以使用 cudaMallocHost 函式，在主内存中取得一块 page locked 的内存。不过，如果要求太大量的 page locked 的内存，将会影响到操作系统对内存的管理，可能会减低系统的效率。