使用GPU加速通用科学计算- CUDA技术解析

刘 勇
(菏泽学院计算机与信息工程系 山东 菏泽 274015)

【摘 要】GPU是图形加速卡的处理单元,具有大量的并行流水线,通常,其浮点运算能力是同代的CPU的10倍以上。本文介绍了一种尚在完善中的利用GPU强大的浮点运算能力来加速通用科学计算的编程模型CUDA。CUDA是用于GPU计算的开发环境,它是一个全新的软硬件架构,可以将GPU视为一个并行数据计算的设备,对所进行的计算进行分配和管理。

【关键词】CUDA; GPU; 并行计算

一、引言

CUDA是用于GPU计算的开发环境,它是一个全新的软硬件架构,可以将GPU视为一个并行数据计算的设备,对所进行的计算进行分配和管理。在CUDA的架构中,这些计算不再像过去所谓的GPGPU架构那样必须将计算映射到图形API(OpenGL和Direct 3D)中,因此对于开发者来说,CUDA的开发门槛大大降低了。CUDA的GPU编程语言基于标准的C语言,因此任何有C语言基础的用户都很容易地开发CUDA的应用程序。目前为止,CUDA支持的GPU只包括NVIDIA公司的硬件支持Direct X 10编程接口的GeForce 8系列以及更新型号,考虑到较旧的GPU型号其像素渲染管线和顶点渲染管线是分离的且管线数量较少,所以在这些GPU上执行通用计算其性能并不会比当前最好的CPU显著提高。目前,最新的NVIDIA公司的GPU产品GeForce GTX 280具有240个流处理器,运行在1300MHz频率上,浮点运算性能933GFlops,而AMD的Radeon HD 4870则具有800个850MHz的流处理器,浮点运算性能1003GFlops,达到了1TFlops级别。而Intel公司最新的四核心处理器浮点运算性能只能达到100GFlops。斯坦福大学的分布式运算项目Folding@Home使用GeForce GTX 280的运算效率可达Core 2 Quad 3GHz四核心处理器的45倍。考虑到图形加速卡低廉的价格及其普及程度,利用GPU强大的处理能力来加速通用科学计算以及普通应用程序的计算密集部分具有非常好的应用前景。

二、编程模型

通过CUDA编程时,将GPU看作可以并行执行非常多个线程的计算设备。GPU相当于是协处理器,处理大量的并行运算,由于GPU硬件本身的结构特性决定了它的分支控制性能低下,但其数学运算能力巨大。所以最终的运算模式是用CPU做控制程序,然后把需要大量并行处理的计算密集任务放到GPU上来处理。CPU和GPU都保留自己的DRAM存储器,分别称为主机内存(host memory)和设备内存(device memory)。用户可以通过优化的API调用将数据从一个DRAM复制到其他DRAM中,而优化的API调用使用了设备的高性能直接内存访问(DMA)引擎。

在CUDA的软件层面,NVIDIA C编译器是其中的核心。CUDA程序是GPU和CPU的混合代码,它首先由NVIDIA C编译器进行编译。经过编译后,GPU和CPU的代码将被分离,GPU代码被编译成GPU计算的机器码,而CPU的C代码输出由标准的C编译器进行编译。因此一个完整的CUDA软件开发环境还需要有一个面向CPU的C编译器。CUDA可以支持多种运行在Windows XP和Linux操作系统下的C开发系统诸如Microsoft Visual C++等。CUDA通过允许程序员定义一种叫kernel的函数来扩展C语言以支持CUDA技术。当一个这种函数被调用时,会并行的执行N遍在N个不同的CUDA线程上,与此对应,普通的C语言函数执行时只执行一遍。定义一个kernel函数要用到“__global__”限定符,而每次调用所并行执行的线程数量用扩展的“<<< >>>”语法来指定。

CUDA的函数类型限定符有三种,“__device__”限定符声明函数在GPU设备上执行,只能从GPU中调用。“__global__”限定符将函数声明为内核,这种函数在设备上执行,只能从CPU主机中调用,而“__host__”限定符声明函数在主机上执行且只能从主机中调用。如:

```
__global__ void incrementArrayOnDevice(float *a, int N){……};
```

就是定义一个在GPU上并行执行,且只能从CPU中调用的函数。而

```
incrementArrayOnDevice <<< 4, 256 >>> (a_d, N);
```

是对该函数的调用,共有4X256=1024个独立的并行线程来执行该函数。

三、程序实例

该实例分别在CPU和CUDA设备上进行简单的计算,在浮点数组中以1为增量增加每个元素,最终比对两者的结果看是否相等。

```
#include <stdio.h>
#include <assert.h>
#include <cuda.h>
void incrementArrayOnHost(float *a, int N)
{
  int i;
  for (i=0; i < N; i++) a[i] = a[i]+1.f;
}
__global__ void incrementArrayOnDevice(float *a, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx]+1.f;
}
int main(void)
{
  float *a_h, *b_h;  // 到主机内存的指针
  float *a_d;        // 到显卡内存的指针
  int i, N = 10;
  size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  b_h = (float *)malloc(size);
  cudaMalloc((void **) &a_d, size);
  for (i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
  incrementArrayOnHost(a_h, N);
  int blockSize = 4;
  int nBlocks = N/blockSize + (N%blockSize == 0? 0:1);
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);
  cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);
free(a_h); free(b_h); cudaFree(a_d);
```

我们注意到经典的串行算法是使用常用的for语句进行迭代循环,而__global__函数用nBlocks×blockSize个独立线程并使每个线程只完成一个加法,使得GPU函数代码里没有一个循环但也可以正确的完成同样的功能。这体现出了CUDA并行和串行代码的编程策略的不同,因为GPU固有的弱控制强计算性质,使得在GPU上执行的代码尽量设计的少用循环或使得每次循环工作多而循环及分支次数少,并且尽量使用多线程。同时我们也发现,每次的计算都要通过使用cudaMemcpy函数将数据从主内存拷贝到GPU内存上,运算完成之后,再用相同的函数将结果从GPU内存拷贝回主内存中。

四、总结

CUDA是一项以提高计算性能为目的的新兴技术,但是其影响将远远超出‘计算’的范围。GPGPU是一个神奇的事物,因为如果处理得好的话,那么可以几千倍的打破摩尔定律,而跳跃到下一个时代。GPGPU的问题就在于编程的痛苦性太高,而CUDA就是为此目的而设计。目前不止是GPU是并行处理器,就连传统的(下转第411页)

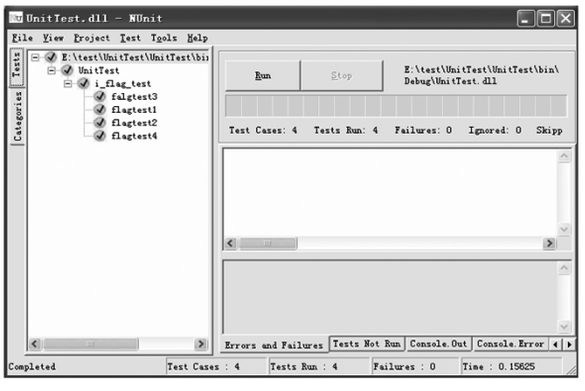
```
using SystemText;
using NUnit.Framework;

namespace UnitTest
{
    //程序代码
    public class i_flag
    {
        public int flag(int i_flag, int icount)
        {
            int i_result = 0;
            while (icount > 0)
            {
                if (i_flag == 0)
                {
                    i_result = icount + 100;
                    break;
                }
                else if (i_flag == 1)
                    i_result = icount * 10;
                else
                    i_result = icount * 20;
                icount- - ;
            }
            return i_result;
        }
    }
//测试代码
[TestFixture]
    public class i_flag_test
    {
        //路径一
        [Test]
        public void flagtest1()
        {
            int icount1 = 0;
            int i_flag1 = 0;
            i_flag my = new i_flag();
            Assert.AreEqual(0, my.flag(i_flag1, icount1));
        }
        //路径二
        [Test]
        public void flagtest2()
        {
            int icount1 = 1;
            int i_flag1 = 0;
            i_flag my = new i_flag();
            Assert.AreEqual(101, my.flag(i_flag1, icount1));
        }
        //路径三
        [Test]
        public void falgtest3()
        {
            int icount1 = 1;
            int i_flag1 = 1;
            i_flag my = new i_flag();
            Assert.AreEqual(10, my.flag(i_flag1, icount1));
        }
        //路径四
        [Test]
        public void flagtest4()
        {
            int icount1 = 1;
            int i_flag1 = 2;
            i_flag my = new i_flag();
            Assert.AreEqual(20, my.flag(i_flag1, icount1));
        }
    }
}
```

代码编写完成后，编译检查语法是否正确，等编译通过，在VS2005中，直接生成解决方案，则在该项目工程的bin文件夹下会有一个UnitTest.dll文件。

第五步：在NUnit中运行测试代码

运行NUnit图形测试工具，打开编译好的UnitTest.dll文件，点"Run"按钮，就可以看到如下画面：



如图显示均是绿色，表示测试通过。

通过该实例，相信大家对该如何在C#中进行单元测试有了一个基本的认识。当然，NUnit并不只针对C#语言，事实上，你可以在任何.net语言中使用NUnit来测试你的单元，方法都一样。

四、总结

单元测试看上去虽然有点麻烦，但是它为开发人员提供了一个安全的观点，让开发人员对自己的程序更加有信心，在减少开发后期进行频繁Debug所耗费时间的同时也为应用软件提供了第一道安全防护网，因此，单元测试是提高开发效率和软件品质的一个重要手段。科

参考文献】

[1]（美）托马斯，陈伟桩\陶文译，《单元测试之道C#版—使用UNnit》，电子工业出版社，2005年1月.
[2]曲朝阳，《软件测试技术》，中国水利水电出版社，2006年8月.

[责任编辑：张艳芳]

（上接第394页）CPU也由于受到主频难以提高的限制而大力向多核心方向发展，所以未来的整个软件设计世界正在经历着前所未有的整体的由串行技术迅速向并行化的转变，并行程序设计是困难的，所以我们迫切需要有一个好的并行设计架构来简化工作，CUDA的出现无疑为我们提供了一种优雅的解决方案。科

参考文献】

[1] NVIDIA Corporation,NVIDIA CUDA Programming Guide 2.0.2008年6月.
[2]邓劲,图形处理器上的快速傅里叶变换,现代电子技术.2007年10期.

作者简介：刘勇（1980—），男，山东菏泽人。菏泽学院计算机与信息工程系教师，主要研究方向为数据库技术及并行算法。

[责任编辑：田瑞鑫]