高技术通讯 2012 年 第22卷 第4期:388 ~ 394

doi:10.3772/j.issn.1002-0470.2012.04.009

GPU-S2S:面向 GPU 的源到源翻译转化①

李 丹② 曹海军 董小社③ 张 保

(西安交通大学电子与信息工程学院 西安 710049)

摘 要 针对图形处理器(GPU)架构下的软件可移植性、可编程性差的问题,为了便于在 GPU 上开发并行程序,通过自动映射与静态编译相结合,提出了一种新的基于制导语句控制的编译优化方法,实现了一个源到源的自动转化工具 GPU-S2S,它能够将插入了制导语句的串行 C 程序转化为统一计算架构(CUDA)程序。实验结果表明,经 GPU-S2S 转化生成的代码和英伟达(NVIDIA)提供的基准测试代码具有相当的性能;与原串行程序在 CPU 上执行相比,转换后的并行程序在 GPU 上能够获取显著的性能提升。

关键词 图形处理器(GPU),制导语句控制,源到源转化

0 引 言

全新通用图形处理器(general purpose graphic process unit, GPGPU)的解决方案统一计算架构(compute unified device architecture, CUDA)的引入引发了图形处理器(GPU)通用计算的革命,CUDA 的引入在一定程度上降低了在 GPU 上开发通用计算的难度,然而,其复杂的存储管理体系、线程层次结构以及存储间的数据传输控制仍旧成为普通用户开发并行程序的巨大障碍。使用 CUDA 编程的开发人员仍需要掌握 GPU 架构方面的知识,且若要充分发挥 GPU 的并行处理能力,编程者还需要从负载平衡、存储区管理[1,2]等多个方面进行不断优化[3]。总而言之,计算密集型应用程序的移植需要花费程序开发者大量的时间和精力。目前许多研究学者投入了极大的热情研究编译器优化技术以解决 GPU 架构下的软件可移植性与可扩展性问题。

美国国家海洋和大气管理局(NOAA)地球系统研究实验室开发出了一种能够把采用 Fortran 语言编写的源代码转化为 CUDA 源程序的编译器——F2C-ACC[4]。加拿大滑铁卢大学推出了着色元编程(shader metaprogramming, SH)[5],其目的在于使用 GPU 进行绘图和通用计算。另外,普渡大学提出了把标准 OpenMP 应用程序转化为 CUDA 程序的源到源(source-to-source, S2S)编译框架[6]。然而,以上三个软件转化工具,虽然可以在一定程度上节约开发时间,但是,总体而言,对应用的移植要求也比较高,即使移植到 CUDA 环境,也需要程序员经过进一步的手动分析和优化才能达到较理想的性能,甚至需要基于新的计算模型重写编写代码。另外,针对 CUDA 编程,伊利诺伊大学研究组提出了一个通过插入制导语句自动完成存储优化的优化工具 CUDA-Lite[7]。虽然 CUDA-Lite 通过源到源转化生成优化后的代码,然而其仅仅局限于针对使用了全局存储区的程序的优化,且需要编程者事先写好一个完整的 CUDA 程序。

为了解决这些问题,本文通过自动映射与静态编译配置相结合,提出了一种新的基于制导语句控制的编译优化方法,实现了一个自动转化工具 GPU-S2S,它能够完成传统应用程序从同构平台到流处理架构 GPU 平台的源到源映射转化。

1 GPU-S2S

图 1 示出了本文实现的源到源自动转化工具 GPU-S2S 的编译流程。从图中看出,一个 C 语言编写的串行源程序通过 GPU-S2S 进行源到源编译,流程如下:(1) 编译验证:为了保证插入了制导语句的 C 串行程序(*.c 和 *.h 文件)的正确性,首先通

① 863 计划(2009AA01Z108,2009AA01A135,2006AA01A109)和中央高校基本科研业务费专项资金(08142007)资助项目。

② 女,1985 年生,硕士;研究方向:高性能计算,计算机体系结构,底层系统软件开发;E-mail: lidan2011@mail.xjtu.edu.cn

③ 通讯作者,E-mail: xsdong@mail.xjtu.edu.cn

(收稿日期:2010-09-02)

过 C 语言编译器对其进行编译验证。(2) GPU-S2S 转换:将通过了编译验证的 C 串行程序源码作为输入,GPU-S2S 对其进行源到源的编译,生成符合 CUDA 规范的并行程序源码(* .h、* .cu、* .c 文件)。(3) CUDA 编译:第二步生成的 CUDA 并行程序源码经过 CUDA 编译工具编译,生成 GPU 可执行的并行程序(* .o 文件)。

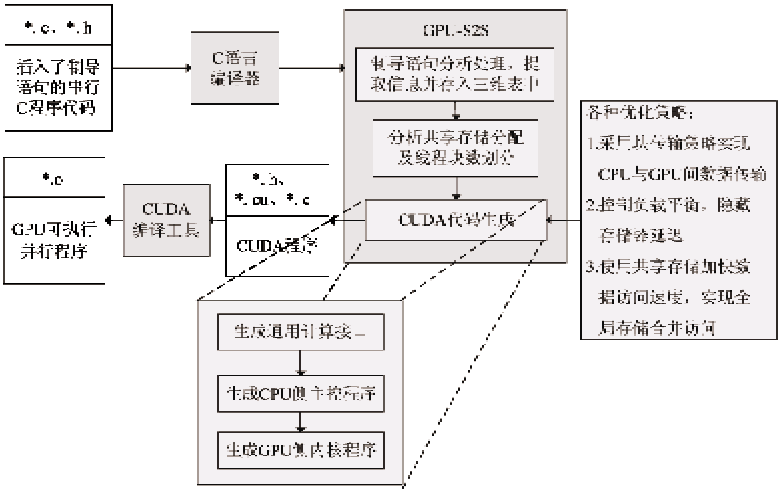

图 1　GPU-S2S 的编译流程

如图 1 所示,GPU-S2S 主要分为 3 个模块:制导语句处理模块、分析共享存储分配和线程块划分模块、CUDA 代码生成模块。具体来说,GPU-S2S 首先从 C 串行源程序中识别并分析出各类制导语句(制导语句规范见 1.1 节),提取相关信息。接着,提取并分析源程序中的共享存储及线程块数的分配信息(见 1.2 节)。由于 GPU 架构的特殊性,为了生成优化的 CUDA 程序,在代码生成阶段,GPU-S2S 根据提取出的信息,采用相对应的优化策略把插入了制导语句的串行程序翻译成加入了 CUDA 运行时库的优化程序。其中,GPU-S2S 生成 CUDA 并行程序的步骤见图 1。目前,如图 1 所示,GPU-S2S 支持三种优化策略。GPU-S2S 中具体转化策略见 1.2 节。

图2给出了GPU-S2S转化前后程序的标准框架。转化前后代码框架都分为用户自定义部分和调用共享库两大部分。

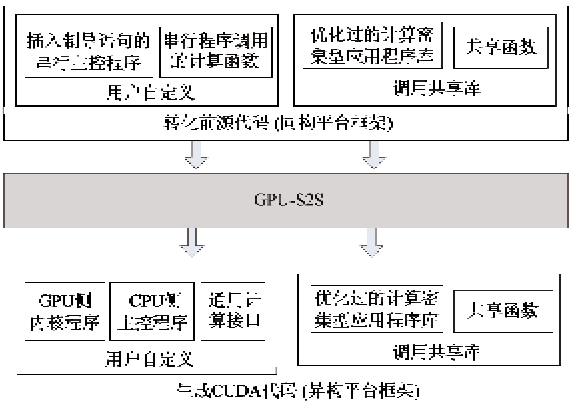

图 2　GPU-S2S 转化前后程序框架

1.1　制导语句

如前所述,用户在 C 串行程序中插入制导语句以提示 GPU-S2S 进行源到源转化。目前,GPU-S2S 提供了 5 种制导语句下面分别描述。

(1)存储映射制导语句

对于需要在 GPU 全局存储上分配空间的变量,可以通过存储映射制导语句对其声明。对变量通过存储映射制导语句进行声明的格式如下:

```
#pragma task input:\
Variable _ Type Variable _ Name Variable Space
#pragma task output:\
Variable _ Type Variable _ Name Variable Space
```

其中,input 表示数据需要从 CPU 存储空间传输到 GPU 全局存储空间;通过分析 input 制导语句,GPU-S2S 将在转换后的 CUDA 代码中插入变量存储分配语句、数据传输语句、空间释放语句(示例见第 2 节)。同理,output 表示数据从 GPU 全局存储空间传输到 CPU 存储空间。Variable _ Type 表示变量类型,Variable _ Name 为变量名,Variable _ Space 表示需要给该变量分配的存储空间的大小,其声明格式为

[Dimensions _ Size] {[Dimensions _ Size] * }

其中,Dimensions _ Size 表示每一维度的大小,{} 内表示在多维情况下还需增加的参数。

上述两类制导语句屏蔽了 CPU 与 GPU 间的数据传输过程,自动实现了全局存储变量对编程者的透明性。对于通过 input 制导语句声明了的变量,在全局存储空间允许的情况下,可采用批处理的方法一次性把所需数据传输到 GPU 全局存储器中。

(2)内核区域标识制导语句

内核区域标识制导语句的作用是在源 C 程序代码中界定出将要运行在 GPU 上的计算核心,该计算核心将被 GPU 上每个线程运行一次。具体定义如下:

```
#pragma kernel:tblock < ThreadBlocks _ Partition
>\
thread < Threads _ Partition >
……      //计算核心代码
#pragma kernel _ end:
```

其中,ThreadBlocks _ Partition 定义了一个网格中线程块的组织方式,声明格式为 ThreadBlocks _ Per _ Grid {, ThreadBlocks _ Per _ Grid *}

Threads _ Partition 定义了一个线程块中线程的组织方式,声明格式为

Threads _ Per _ Block {, Threads _ Per _ Block *}

上述参数是在考虑维度的情况下的数据规模和线程块大小的线性组合,是影响内核并行运行的重要因素。

(3)循环映射制导语句

在源 C 程序的计算核心中,在循环代码段前加入循环映射制导语句表示该循环可以被线程并发执行,其定义格式如下:

```
#pragma parallelizable loop part
```

对于在循环代码前插入了该制导语句的源代码,针对需要计算的变量的维度(例如 2 维或 1 维),GPU-S2S 将把循环变量转化为 CUDA 线程中对应的索引。需要注意的是,循环语句前未插入该制导语句的即为每个线程需要串行执行的部分。

例如,在傅里叶变换源代码中:

```
for(int L = 1; L < = level; L + +){      //表示每个
                                  //线程都需执行该循环
  ……
  #pragma parallelizable loop part://循环映射制导语句
  for(int k = 0; k < N / 2; k + +){   //表示每个
          //线程只执行该循环中的一次计算
  ……
  }
}
```

(4)共享存储制导语句

共享存储制导语句的作用是为内核函数中传入的 input 变量或 output 变量分配共享存储,其中 copyin 表示把数据从全局存储传入到共享存储,copyout to 表示反方向传输。定义格式如下:

```
#pragma shared alloc copyin: \
Variable _ Type Variable _ Name Variable _ Space
#pragma shared copyout to: \
Variable _ Type Variable _ Name Variable _ Space
```

其中,copyin 表示变量需从全局存储传输到共享存储,copyout to 则反之。Variable _ Type 表示变量类型;Variable _ Name 为变量名;Variable _ Space 表示需要给该变量分配的共享存储空间大小,其定义格式为:[Xdimension _ Relative _ StaAddress: Xdimension _ Relative _ StaAddress + Size]{[Ydimension _ Variable][Zdimension _ Variable]}或者{[Xdimension _ Variable]}[Ydimension _ Relative _ StaAddress: Ydimension _ Relative _ StaAddress + Size]{[Zdimension _ Variable]},其中,Xdimension _ Relative _ StaAddress 表示 X 维相对起始地址,Xdimension _ Relative _ StaAddress + Size 表示 X 维相对结束地址。

对于通过共享存储制导语句声明了的全局变量,GPU-S2S 在转化时会把其替换为共享变量。

(5)同步制导语句

同步制导语句的作用是保证一个线程块的线程全部计算完毕,实现线程块内的线程同步执行。其定义格式如下:

```
#pragma synchronous:
```

根据同步制导语句,GPU-S2S 将生成 CUDA 中对应的同步函数。

1.2 转化策略

如图 1 所示,当编程者在串行 C 程序代码中的合适位置插入制导语句后,GPU-S2S 的制导语句处理模块会逐条扫描程序源码,识别并分析各类制导语句,提取相关信息并将其存储在三维数组中,为 CUDA 代码生成模块做好准备。可以利用剖分指导优化的思想来确定计算核心,从而指导编程者插入内核区域标识制导语句。

GPU 架构有以下特征:(1)共享存储的大小有限,有些应用的输入数据不能一次性调入共享存储中;(2)每个流多处理器(streaming multiprocessor, SM)能同时运行的最大线程块和线程数目有限。因此,基于以上考虑,在 GPU-S2S 中,分析共享存储分

配和线程块划分模块做出了相关的判断,且为了保证性能,GPU-S2S 在转化中设定一个 SM 上至少能同时运行两个线程块。

判断共享存储以及线程块数分配是否合理的算法如下所示:

```
输入:共享存储制导语句集合 S
输出:一个 SM 上实际运行的线程块数目 Thread _ Block _ Num;
      一个 SM 上分配的总共享存储 Whole _ Share _ Size;
      串行执行内核数。
for(每个 s_i ∈ S)  //s_i 表示第 i 条共享存储制导语句
    提取第 i 条共享存储制导语句 s_i 中变量的共享空间分配表达式;
    扫描串行程序,将表达式中宏定义变量替换为对应数值,计算出表达式的值;
    查找变量表中该变量的类型大小,计算出为该变量分配的共享存储空间大小 Share _ Size;
    Share _ Per _ ThreadBlock + = Share _ Size;
    \Share _ Per _ ThreadBlock 表示每个线程块上为所有共
    \享存储制导语句中的变量分配的共享存间大小
end for
计算出一个 SM 上实际运行的线程块数目 Thread _ Block _ Num; Whole _ Share _ Size = Share _ Per _ ThreadBlock × Thread _ Block _ Num;
if (Whole _ Share _ Size > 共享存储区空间最大值)//分配不正确终止,重新分配;
else  //分配正确
  if(数据步长距离 > 一个线程块中线程所能计算的数据规模)分配计算核心到多个串行执行内核,输出总的串行执行内核数;
  else
    分配计算核心到一个内核,输出执行内核数为 1;
  end if
end if
```

在完成以上两步之后,GPU-S2S 的 CUDA 代码生成模块开始源到源的编译转化,主要分为 3 个步骤:(1) 根据转化前 CPU 平台的串行程序框架,生成面向 GPU 的调用共享库部分和通用计算接口;(2) 结合插入制导语句的串行主控程序,生成 CPU 侧主控程序,其中,根据存储映射制导语句,生成变量全局存储分配语句、CPU 与 GPU 间的数据传输语句、空间释放语句;(3) 生成 GPU 侧内核程序(图 3)。

生成 GPU 侧内核程序最为关键。根据内核区域标识制导语句,GPU-S2S确定将要运行在GPU上

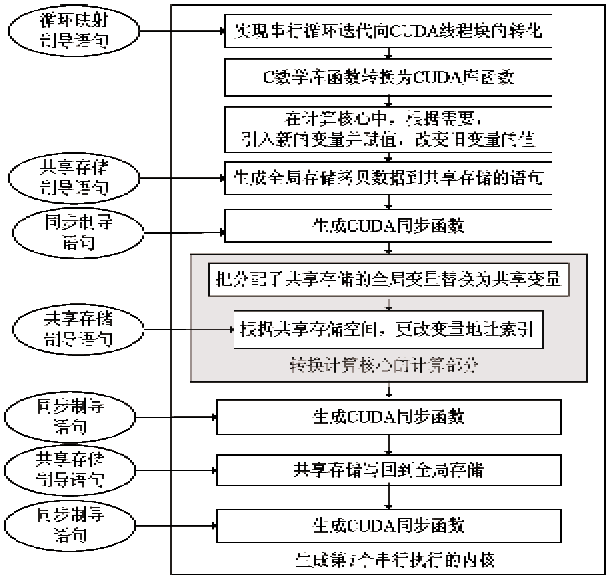

图 3　GPU 侧内核生成

的计算核心,然后扫描该计算核心,按照制导语句的类型分别把 C 串行程序代码段翻译转化为相应的 CUDA 代码。详细翻译转化机制如下:

(1)根据循环映射制导语句,实现串行循环迭代空间到 CUDA 线程块空间的转化。

(2)根据共享存储制导语句,若该制导语句有 copyin 标识,则将数据通过全局存储合并访问文献[8]传输到共享存储,即让线程把全局存储中某位置的数据写到共享存储中对应的位置;若有 copy-out 标识,则将数据从共享存储写回到全局存储。

(3)根据同步制导语句,生成相对应的 CUDA 同步函数。

除了根据以上制导语句转化,在生成 GPU 侧内核程序时,GPU-S2S 把 C 串行程序中调用的数学标准库函数转换为 CUDA 标准库函数;另外,在计算核心中,若源 C 串行程序中的变量不足以满足 CUDA 执行要求,GPU-S2S 将引入新的变量,并给予赋值,同时改变旧变量的值。

2　示　例

本节以计算密集型应用矩阵乘法(原理见文献[8])为例来说明 GPU-S2S 的转化。设输入矩阵 M、N、输出矩阵 P 都为 512 ×512 方阵。图 4 给出的是插入了制导语句之后的 C 源串行程序。

```
0 #pragma kernel: \
1 tblock < WIDTH/BLOCK _ SIZE, WIDTH/BLOCK _ SIZE > \
2 thread < BLOCK _ SIZE, BLOCK _ SIZE >
```

```
3 void matrixMul _ kernel (float * P,float * M,float * N,int width){
4 #pragma parallelizable loop part:
5   for (int row = 0;row < width;row + + ){
6     for(int col = 0;col < width;col + + ){
7       int k,m;
8       float sum = 0;
9       for(m = 0;m < width;m + = BLOCK _ SIZE){
10 #pragma shared alloc copyin: \
11        float N[m:m + BLOCK _ SIZE][col]
12 #pragma shared alloc copyin: \
13        float M[row][m:m + BLOCK _ SIZE]
14 #pragma synchronous:
15          for (k = 0;k < BLOCK _ SIZE;k + + ){
16          sum + = M[row * width + m + k] * N[(m + k) * width + col];
17          }
18 #pragma synchronous:
19       }
20       P[row * width + col] = sum;
21     }
22   }
23 }
24 #pragma kernel _ end:

25 #define BLOCK _ SIZE 16
26 #define WIDTH 512
27 int main(){
     ……//28-42 alloc and initialize input data on CPU
42 #pragma task input:float M[WIDTH][WIDTH]
43 #pragma task input:float N[WIDTH][WIDTH]
44 #pragma task output:float P[WIDTH][WIDTH]
45   matrixMul _ kernel (P,M,N,WIDTH);
     ……//46-48 free variable
49   return 0;
50 }
```

图 4 插入制导语句的串行程序

插入了制导语句的矩阵相乘 C 串行程序(图 4 所示)经过 GPU - S2S 转换后生成的 CUDA 并行程序如图 5 所示。

```
0 //compute kernel
1 __ global __ void matrixMul _ kernel0 (float * P,
float * M,float * N,int width){
2   int bx = blockIdx. x;
3   int by = blockIdx. y;
4   int tx = threadIdx. x;
5   int ty = threadIdx. y;
6   int row = by * BLOCK _ SIZE + ty;
7   int col = bx * BLOCK _ SIZE + tx;

8   int k,m;
9   float sum = 0;
10   for(m = 0;m < width;m + = BLOCK _ SIZE){
11     __ shared __ float Ns[BLOCK _ SIZE][BLOCK _ SIZE];
12     Ns[ty][tx] = N[(m + ty) * width + col];
13     __ shared __ float Ms[BLOCK _ SIZE][BLOCK _ SIZE];
14     Ms[ty][tx] = M[row * width + m + tx];
15     __ syncthreads();
16     for (k = 0;k < BLOCK _ SIZE;k + + ){
17       sum + = Ms[ty][k] * Ns[k][tx];
18     }
19     __ syncthreads();
20   }
21   P[row * width + col] = sum;
22 }

23 #define BLOCK _ SIZE 16
24 #define WIDTH 512
25 void runTest(int argc, char * * argv){ //control function
     ……//26-37 alloc and initialize input data on CPU
38   float * d _ M;
39   cutilSafeCall(cudaMalloc((void * * ) & d _ M,
                sizeof(float) * WIDTH * WIDTH));
40   float * d _ N;
41   cutilSafeCall(cudaMalloc((void * * ) & d _ N,
                sizeof(float) * WIDTH * WIDTH));
42   float * d _ P;
43   cutilSafeCall(cudaMalloc((void * * ) & d _ P,
                sizeof(float) * WIDTH * WIDTH));
44   cutilSafeCall(cudaMemcpy2D(d _ M, sizeof(float) * WIDTH,
          M, sizeof(float) * WIDTH, sizeof(float) * WIDTH,
                WIDTH, cudaMemcpyHostToDevice));
45   cutilSafeCall(cudaMemcpy2D(d _ N,sizeof(float) * WIDTH,
        N, sizeof(float) * WIDTH, sizeof(float) * WIDTH,
                WIDTH, cudaMemcpyHostToDevice));

46   dim3 dimBlock _ 0 _ 0(BLOCK _ SIZE,BLOCK _ SIZE);
47   dim3 dimGrid _ 0 _ 0(WIDTH/BLOCK _ SIZE,
                WIDTH/BLOCK _ SIZE);
48   matrixMul _ kernel0 < < < dimGrid _ 0 _ 0,
       dimBlock _ 0 _ 0 > > > (d _ P,d _ M,d _ N,WIDTH);

49  cutilSafeCall(cudaMemcpy2D(P, sizeof(float) * WIDTH,
    d _ P, sizeof(float) * WIDTH, sizeof(float) * WIDTH,
    WIDTH, cudaMemcpyDeviceToHost));

50   cutilSafeCall(cudaFree(d _ M));
51   cutilSafeCall(cudaFree(d _ N));
52   cutilSafeCall(cudaFree(d _ P));
     ……//53-55 free variable
56 }
```

图 5 GPU-S2S 生成的矩阵乘法代码

GPU-S2S 先扫描 C 源串行程序(见图 4)中的主控程序 main 函数,根据存储映射制导语句提示(图 4中第 42-44 行),生成相关存储分配传输(图 5 中第 38-45 行)以及释放语句(图 5 中第 50-52 行)。

接着,GPU-S2S 完成计算核心函数 matrixMul _ kernel 的转化,由内核区域标识制导语句(图 4 中第 0-2 行)可知,计算核心将在 GPU 上由(512/16) × (512/16) =32 ×32 个线程块并行执行,每个线程块有 16 ×16 个线程。对输入数据采用棋盘划分,每次从 M、N 矩阵中分别传入 16 ×16 大小的数据块。

GPU-S2S 根据循环映射制导语句(图 4 中第 4 行)和存储映射制导语句中变量的维度(图 4 中第 42-43 行)确定出串行程序中可并行的循环迭代语句(图 4 第 5-6 行)。索引为(0,0)到(15,15)的线

程处理矩阵 M、N 中 0-16 行和 0-16 列的矩阵相乘。依次类推。因此,转换后每个线程要计算的数据在矩阵中的位置如图 5 第 2-7 行所示。

根据共享存储制导语句(图 4 中第 10-13 行),GPU-S2S 生成分块矩阵(大小为 16 × 16)数据拷贝代码,即将数据从全局存储写到共享存储(图 5 中第 11-14 行)。如图 5 中 14 行第 i 个线程访问 M 中第 row × width + i 个元素,GPU-S2S 生成符合全局存储合并访问的 CUDA 并行程序。

3　验证与测试

本节以矩阵乘法和快速傅立叶变换(FFT)为测试用例,通过实验来分析验证 GPU-S2S 将 C 源串行程序转化成 CUDA 并行程序的正确性和性能情况。

实验平台选用浪潮英信 NF5588 服务器,其配置了两个 4 核的 Xeon CPU (主频 2.27GHz), 12GB 内存,配备了两块 NVIDIA Tesla C1062x 系列的 GPU;操作系统为 RedHat 企业服务器 5.3 版;采用 CUDA2.3 版本。

表 1 给出了矩阵乘法在不同输入规模下执行所花费的时间。表中 GPU-S2S 表示插入制导语句的串行 C 程序经过 GPU-S2S 转化后生成的 CUDA 程序,SDK-Example 表示 NVIDIA 提供的标准 CUDA 基准测试矩阵相乘程序,CPU 表示矩阵乘法串行程序。测试结果表明:在不同输入数据规模情况下,由 GPU-S2S 生成的 CUDA 程序和 NVIDIA 提供的标准 CUDA 基准测试矩阵相乘程序所花费的总执行时间非常接近,二者均采用了全局存储合并访问传输数据到共享存储;且随着数据规模的增大,串行程序的执行时间与 GPU-S2S 生成的并行程序的执行时间的比值(即加速比)越来越大(从 96 到 1677.5)。这主要是因为:GPU具备强大的并行处理能力和极高的存储器带宽,随着数据规模的增大,GPU 的并行处理能力会得到充分的发挥。

表 1　不同输入规模时矩阵相乘总执行时间(ms)

	256 × 256	512 × 512	1024 × 1024
GPU-S2S	0.5280	3.0632	14.2465
SDK-Example	0.5576	3.1129	14.2675
CPU	50.9270	447.3130	10577.0918
	2048 × 2048	4096 × 4096	8192 × 8192
GPU-S2S	96.7969	728.3226	5692.1143
SDK-Example	96.6832	727.6077	5691.3530
CPU	95450.2969	837221.25	954636

图 6 给出了在不同输入数据规模(16 倍数和非 16 倍数)时,GPU-S2S 生成的矩阵相乘并行程序执行时间比较。实验结果表明:在输入数据规模比较接近时,非 16 倍数的输入数据在 GPU 上比 16 倍数的输入数据反而要花费较多的执行时间,例如,1000 × 1000 的输入矩阵要比 1024 × 1024 的输入矩阵执行时间长,而且,随着数据规模的增大,这种趋势越来越明显。这主要是因为 GPU 的内存控制器总是从 16 字节的整数倍地址开始读取数据;同时,当输入矩阵规模不是半个线程组(half-warp,在 G80 上其包含 16 个线程)中线程的整数倍时,最后一个线程块中的一些线程闲置,其不参与计算,故导致总的计算时间延迟。

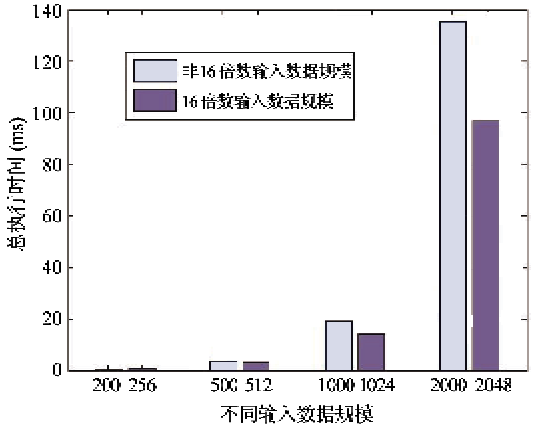

图 6　GPU-S2S 生成的并行程序执行时间比较

图 7 给出了 FFT 在不同输入规模下执行所花费的时间。图中 CUFFT 表示表示调用 NVIDIA 提供的函数库CUFFT进行傅立叶变换的程序。测试

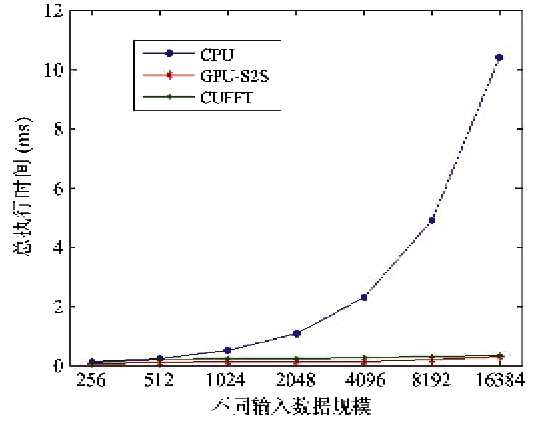

图 7　FFT 在不同数据规模下的总执行时间

结果表明,在不同输入数据规模情况下,由 GPU-S2S 生成的 CUDA 程序比 NVIDIA 提供的 CUFFT 傅立叶变换库函数执行时间稍快。且随着数据规模的增大,由 GPU-S2S 生成的程序和 CUFFT 的执行时间增长缓慢,而 CPU 串行程序的执行时间急剧增长。这充分体现了 GPU 强大的并行处理能力。

4 结 论

本文提出了一种新的借助制导语句的编译指导方法,实现从 C 语言串行源程序到性能良好的 CUDA 源程序的自动转化的工具 GPU-S2S,它支持现有应用开发习惯和遗留应用移植,能够降低编程者开发面向 GPU 的并行程序的难度,实现 GPU 部分底层架构对编程者的透明。实验结果表明,GPU-S2S 能够正确地将 C 源串行程序转化为 CUDA 程序,而且,借助 GPU-S2S 转化生成的程序和 NVIDIA 提供的标准 CUDA 基准测试程序具有相当的性能;与源串行程序在 CPU 上执行相比,转换后的并行程序在 GPU 上能够获取显著的性能提升。

下一阶段,我们将对 GPU-S2S 从两个方面做进一步的改进:(1) 减少制导语句的数目,以达到完全屏蔽 GPU 底层架构的目的;(2) 进一步采用剖分指导优化的思想,在串行及生成的 CUDA 程序中插桩以统计程序各分支的执行频率、各部分计算热点度等信息,基于这些信息来指导 GPU-S2S 生成更优化的代码。

参考文献

[1] Moazeni M, Bui A, Sarrafzadeh M. A memory optimization technique for software-managed scratchpad memory in GPUs. In: Proceedings of the IEEE 7th Symposium on Application Specific Processors, San Francisco, USA, 2009. 43-49

[2] Govindaraju N K, Larsen S, Gray J, et al. A memory model for scientific algorithms on graphics processors. In: Proceedings of the ACM/IEEE Conference on Supercomputing, Tampa, USA, 2006. 6-15

[3] Ryoo S, Rodrigues S S, Baghsorkhi C I, et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of Sympium on Principles and Practice of Parallel Programming, New York, USA, 2008. 73-82

[4] Govett M, Middlecoff J, Henderson T. Running the NIM next-generation weather model on GPUs. In: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, Australia, 2010. 792-796

[5] McCool M D, Qin Z, Popa T S. Shader metaprogramming. In: Proceedings of AMC SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Aire-la-Ville, Switzerland, 2002. 57-68

[6] Lee S, Min S J, Eigenmann R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, USA, 2009. 101-110

[7] Ueng S Z, Lathara M, Baghsorkhi S S, et al. CUDA-lite: reducing GPU programming complexity. In: Proceedings of the 21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC), 2008, LNCS 5335. 1-15, DOI: 10.1007/978-3-540-89740-8_1

[8] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide (Version 2.0). NVIDIA Corporation, July 2008

GPU-S2S: a source to source compiler for GPU

Li Dan, Cao Haijun, Dong Xiaoshe, Zhang Bao

(School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an 710049)

Abstract

To address the problem of poor software portability and programmability of a graphic processing unit (GPU), and to facilitate the development of parallel programs on GPU, this study proposed a novel directive based compiler guided approach, and then the GPU-S2S, a prototypic tool for automatic source-to-source translation, was implemented through combining automatic mapping with static compilation configuration, which is capable of translating a C sequential program with directives into a compute unified device architecture (CUDA) program. The experimental results show that CUDA codes generated by the GPU-S2S can achieve comparable performance to that of CUDA benchmarks provided by NVIDIA CUDA SDK, and have significant performance improvements compared to its original C sequential codes.

Key words: graphic processing unit (GPU), compiler directive, source to source translation