高技术通讯 2013 年 第23 卷 第3 期:257 ~ 262

doi:10.3772/j.issn.1002-0470.2013.03.006

面向 GPU 的循环合并①

杨　扬②[*,**]　崔慧敏[*]　冯晓兵[*]

([*]中国科学院计算技术研究所计算机体系结构国家重点实验室　北京 100190)

([**]中国科学院研究生院　北京 100039)

摘　要　针对现有的将 C 或 Fortran 程序映射到通用图形处理单元(GPU)的自动转换工具主要关注将单个循环生成一个独立的 GPU 内核,从而阻碍了对循环间数据重用的利用的问题,提出一种新的面向 GPU 的循环合并的代码变换方法,该方法通过循环分块(strip mining)和冗余计算等手段达到消除迭代间数据依赖的目的,并可充分利用 GPU 片上的共享内存进行线程间数据交换,从而将此类程序高效地映射到 GPU 上。通过典型程序在 GPU 上的实验表明,该新方法由于能够减少对全局内存的访问,带来了最多高达 1.96 倍的加速比。

关键词　通用图形处理单元(GPU),循环合并,并行,CUDA,循环间数据重用

0　引　言

通用图形处理单元(GPGPU 以下称 GPU)拥有大量的并行运算部件和极高的算术运算能力,对于有较高并行度的程序而言,是一种高效的计算平台。研究[1,2]表明,跟 CPU 相比,拥有高并行度的程序在 GPU 上经过性能调优后可得到高达二到三个数量级的加速比。然而,由于 GPU 结构上的特异性以及编程语言(如 CUDA 等)的不同,要将原有的用 C 或 Fortran 写成的串行程序用 GPU 加速,程序员需要做许多的移植和调优工作。从程序性能调优的角度,Yang 等提出了一个优化编译器[3],对 CUDA 内核进行局部性和并行性方面的优化;Baskaran 等针对 GPU 特有的一些访存问题提出了一个优化框架[4]。为了解决从 C/Fortran 代码到 CUDA 代码移植的问题,以文献[5,6]为代表的工作分析了用户通过制导指定的循环,通过依赖分析判断其是否可以并行,并把可并行的循环转换成 CUDA 代码。上述性能调优和代码移植的工作主要关注对单个循环或内核进行优化,或将单个循环变换成一个独立的 GPU 内核。近期也有些针对循环间数据重用的优化的研究:Belter 等提出一种将用户写成的线性代数表达式编译成基本 BLAS 库函数组合的方法[7],该方法相对于分别调用基本 BLAS 函数的方式,减少了因中间数组引起的访存;Yang 等提出了一种基于流重用图的方法,以实现不同内核间重用数据[8],他们针对的是一种在内核间硬件可以保存数据的流处理器。而 GPU 的片上存储无法在内核间保存数据,因此文献[8]的方法并不适用。为每个循环生成单独内核的方法阻碍了对原有循环间的数据重用的利用。若要挖掘循环间的数据重用,一种方法是先将有数据重用的循环进行循环合并,再将合并后的循环变换成 CUDA 代码。但是,原本可并行的两个循环合并后生成的循环可能会引入迭代间的数据依赖,使得合并后的循环不可并行,从而不能通过现有的方法将这类合并后的循环转换为 CUDA 程序。针对科学计算中大量存在的这类程序,本文提出一种新的循环合并的代码变换方法,该方法能够将这类循环代码变换成相应的高效 CUDA 代码:通过在线程块(thread block)边界增加一些冗余计算来消除线程块间的数据依赖,从而满足在 GPU 上执行的代码在线程块间没有数据依赖的要求;在线程块内部,利用 GPU 片上的共享内存进行高效的通讯,以满足数据依赖关系;通过片上的共享内存的读写操作,保证了原有循环间的数据依赖关系,从而相对于

① 973 计划(2011CB302504, 2011ZX01028-001-002), 863 计划(2009AA01A129, 2012AA010902)和国家自然科学基金(60970024, 60925009, 60921002)资助项目。

② 男,1983 年生,博士生;研究方向:编译技术;联系人,E-mail: yangyang@ict.ac.cn

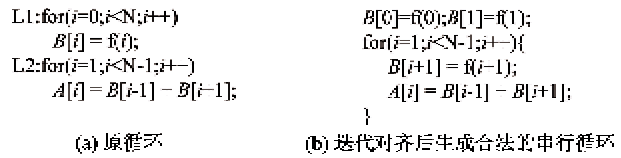
(收稿日期:2012-03-14)

为每个循环单独生成内核的方式，减少了对全局内存的访问，有效提高了性能。

1 背景介绍

GPU 的存储层次包括全局内存和共享内存等。全局内存在 GPU 中的地位相当于 CPU 的内存，而共享内存相当于 CPU 的缓存。但与缓存不同的是，共享内存是需要显式管理的 scratchpad memory。在不发生共享内存块冲突的情况下，线程访问共享内存的延迟很低，而带宽很高，访问效率远高于全局内存，因此应该利用共享内存高效地进行线程通信。GPU 上线程是分组的，固定数量的线程组成一个线程块。同一个线程块中的线程可以通过片上的共享内存进行通讯，属于不同线程块的线程在同一个内核内不能通讯。

图 1(a)中，循环 L1 到 L2 有两个真依赖，依赖向量的长度为常数。L1 和 L2 本身都可以并行，两个循环进行迭代对齐[9]后，可以生成一个合法的循环，但该循环有迭代间依赖，导致无法并行，如图 1(b)所示。L2 的每个迭代都需要用到 L1 的多个迭代产生的值，这类程序在科学计算中很常见。如果能够把这两个循环合并后生成到同一个 GPU 内核中，那么就有机会减少因为数组 B 引起的对全局内存的访问。如果数组 B 在 L2 后不再使用，那



(a) 原程序　　(b) 迭代对齐后生成合法的串行循环

图 1　循环合并后产生依赖的处理

么可以通过将数组 B 存储在共享内存中而完全消除由其引起的全局内存访问；如果 B 在 L2 后还会被读，至少也可以减少一次对该数组的全局内存读操作。

对于 CPU，此类合并后迭代间会产生依赖的循环的并行问题也有研究。Manjikian 等在文献[10]中提出了循环位移和剥离(shift-and-peel)的方法来消除在不同 CPU 上执行的迭代间的依赖，使得不同的 CPU 可以并行执行，但在每个 CPU 内部，还是串行地执行迭代。这样的串行执行保证了迭代间的数据关系能够得以满足。但在 GPU 上，所有的迭代都处于并行状态，无法通过串行执行序来保证依赖关系。Allen 等在文献[9]中提出了代码复制的方法。该方法令循环的每个迭代都做冗余计算，以消除迭代间的数据依赖，但会引入大量的重复计算的开销和冗余数组的空间开销。

对于图 1(a)所示的两个循环，我们希望能将它们生成到同一个 GPU 内核中，在 GPU 上并行执行。我们希望产生的代码遵守 GPU 所需的线程块间不能通讯的要求，而在线程块内部，线程通过共享内存高效地进行通讯。通过将两个循环生成到同一个内核中，并利用共享内存进行通讯，可以减少对全局内存的访问，从而提高性能。与文献[9]不同，我们并不增加每个迭代的计算量，而是在线程块的边界增加少量的冗余计算迭代，以保证每个线程块所需的数据都可完全由自己产生，从而消除线程块间的数据依赖。

2 代码变换

我们提出的代码变换方法对合并后会产生迭代间数据依赖的循环进行变换，然后进行循环合并，使得处理后的代码满足线程块间无数据交换，线程块内部的线程可以并行执行并通过共享内存进行数据交换的要求。基本思路是对所有循环进行循环分块，在每个块(strip)的边界补充一些迭代，以消除不同块间的数据依赖。在块内部，不同迭代间的数据交换利用片上的共享内存来进行。合并后的代码最后被翻译成 CUDA 代码，每个块的计算被映射到一个线程块，块内的每个迭代被映射到一个线程。

2.1 根据循环依赖图计算需要补充的迭代数

首先，我们要找出为了保证每个块不依赖于其他块的计算结果，需要为每个循环在块边界上补充多少迭代。然后，为要进行合并的循环建立循环依赖图。我们假设所有的循环都只包含一条定值语句。如果一个循环中包含多条定值语句，则对该循环进行循环分布，以产生符合上述条件的循环。由于原始的所有循环都是可并行的，因此一定可以合法地进行循环分布。循环依赖图中，每个节点对应一个循环，如果循环 A 到 B 有一个循环间的真依赖，则图上有一条从节点 A 指向节点 B 的有向边。边上的权表示引起依赖的 A、B 两循环迭代的迭代号之差，权可能为正、负或者零。例如，若 B 循环的迭代 i 依赖于 A 循环的 $i+1$ 迭代的计算结果，则权为 1。这样产生的图，两个节点间可能有多条边，我们对边做如下合并：如果两节点间有多条权值为负

的边,则将它们合并成一条边,边的权为原有负权中最小的一个;如果两节点间有多条权值为正的边,则将它们合并成一条边,边的权为原有正权中最大的一个。我们假设循环依赖图上没有依赖环,包含依赖环的情况本文暂不做讨论。

将循环依赖图上的边反向,对反向后的图进行拓扑遍历。我们赋予每个节点 2 个权,分别记录循环的两个边界所需要补充的迭代数,初始值都赋为零。从图中选择任意一个无入边的节点 X,对于任何与 X 有边相连的节点 Y,如果 X 到 Y 的边上的权为正,则将 X 的正权与边上权相加,并与 Y 的正权比较,将较大的更新为 Y 的正权;若 X 到 Y 的边上的权为负,则将 X 的负权与边上权相加,并与 Y 的负权比较,将较小的更新为 Y 的负权;若边上权为 0,则对正和负权都做相应操作。对所有与 X 有边相连的节点如上所述更新权。完成后,从图中去掉节点 X 和所有与其相连的边,并记录下 X 的正负权值。重复以上步骤,直到图中所有的边都被去掉。此时,每个节点的正和负权分别表示该循环在每个块的左和右边界需要补充的迭代数。同时,如果依赖图中边上的权不为 0,则其对应的数组元素的该次定值到引用间需要线程间通讯。过程如图 2 所示。

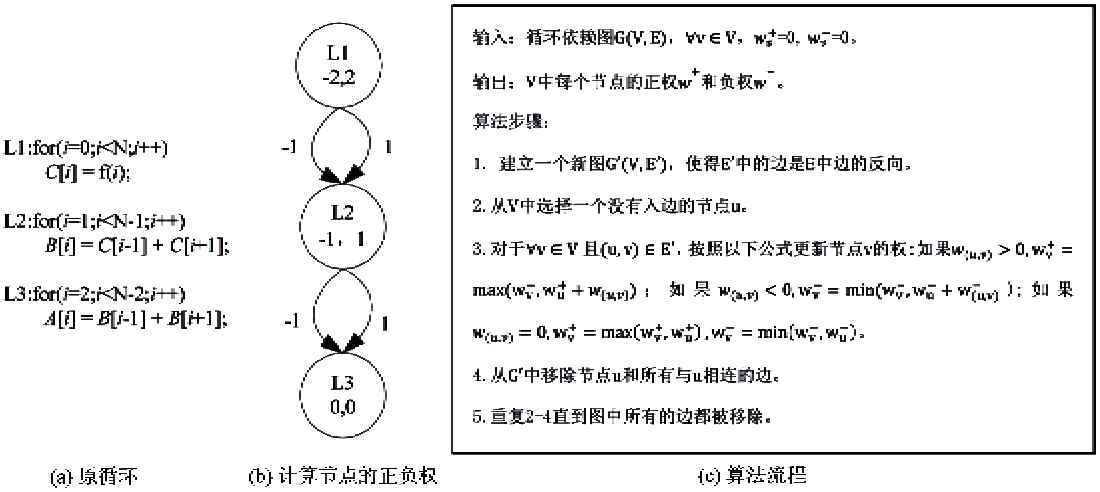


图 2　计算边界需要补充的迭代数

2.2　确定共享内存交换数组的数量和分配关系

接下来,我们需要找出在块内部,哪些数组需要线程间数据交换,并为这些数组在共享内存上分配空间。一个数组可能在不同的循环中被定值多次,但并非每次定值到引用都需要线程间通讯。我们记录下每个需要被通讯的定值以及该定值的最后一次引用对应的语句。我们需要确定共需用到多少共享内存数组作为交换空间,以及每个语句使用哪个数组进行数据交换。定义一个队列用来存放目前未被使用的共享内存交换数组,该队列初始化为空。我们按照原有的文本序扫描各个循环,如果循环产生一个需要线程间通讯的定值,则需为该定值分配一个数组用于线程间数据交换,如果此时交换数组队列不为空,则从中取一个数组分配给该定值,否则产生一个新的交换数组分配给该定值。如果循环的语句是最后一次对某定值的使用,则将该定值对应的交换数组加入上述队列(从而实现了对这些数组的复用)。扫描完后,我们可以得到所需的共享内存交换数组的个数,以及每次定值的时候使用哪个数组。

2.3　代码变换

获得了上述信息后,我们可以进行实际的代码变换,步骤如下:

(1)首先对要进行合并的循环做循环分块,所有的循环采用相同的块大小。(2)将循环分块后各循环的外层循环进行合并,并根据前面分析的结果,为内层循环补充相应的迭代。(3)将内层的循环进行合并。(4)将代码翻译成 CUDA 代码。主要是将循环结构的代码表示成以 GPU 线程为中心的代码。外层循环的一个迭代对应一个线程块,内层循环的每个迭代对应一个线程。循环的控制结构被去掉,循环体内原有的循环变量将用 *blockIdx. x* 和 *threadIdx. x* 的线性组合来表示,其中 *blockIdx. x* 表示原有外层循环的循环变量,*threadIdx. x* 表示原有内层循环的循环变量。(5)根据前面分析的结果,在共享内存中声明相应数量的数组作为线程交换数据的空间,并将原有的对全局内存的访问替换成对相应交换数组的访问。根据上面分析得到的每次定值和引

用对应的数组,在定值语句后插入一条将定值结果写入相应共享内存交换数组的语句,并在该共享内存写语句的前后各加一条__ syncthreads( )语句,以保证不同线程间读和写的顺序,并且保证对同一块空间的先读后写顺序(当一个交换数组第一次被写时,写语句前不需要__ syncthreads( )语句)。将相应的引用语句中对原有全局内存数组的访问改成对相应共享内存交换数组的访问。图 3(a)-(f)示例了代码的变换过程。

```
for(i=0;i<N;i++)
    B[i] = f(i);
for(i=1;i<N-1;i++)
    A[i] = B[i-1] + B[i+1];
```

(a) 原循环

```
for(ii=0;ii<N;ii+=S)
    for(i=ii;i<min(ii+S,N);i++)
        B[i] = f(i);
for(ii=1;ii<N-1;ii+=S)
    for(i=ii;i<min(ii+S,N-1);i++)
        A[i] = B[i-1] + B[i+1];
```

(b) 循环分块

```
for(ii=1;ii<N;ii+=S){
    for(i=ii-1;i<min(ii+S+1,N-1);i++)
        B[i] = f(i);
    for(i=ii;i<min(ii+S,N-1);i++)
        A[i] = B[i-1] + B[i+1];
}
```

(c) 合并外层循环

```
for(ii=1;ii<N;ii+=S)
    for(i=ii-1;i<min(ii+S+1,N);i++){
        B[i] = f(i);
        if(i != ii-1 && i != min(ii+S+1,N))
            A[i] = B[i-1] + B[i+1];
}
```

(d) 合并内层循环

```
int idx = blockIdx.x * (S+2) + threadIdx.x;
B[idx] = f(idx);
if(idx != blockIdx.x * (S-2) && idx != min((blockIdx.x + 1)* (S+2),N))
    A[idx] = B[idx-1] + B[idx+1];
```

(e) 等价为 CUDA 代码

```
__shared__ float swap[S-2];
int idx = blockIdx.x * (S+2) + threadIdx.x;
float tmp = f(idx);
B[idx] = tmp;
swap[threadIdx.x] = tmp;
__syncthreads();
if(idx != blockIdx.x * (S+2) && idx != min((blockIdx.x + 1)* (S-2),N))
    A[idx] = swap[threadIdx.x-1] + swap[threadIdx.x+1];
```

(f) 用对共享内存交换数组的访问替代对全局内存数组的访问

图 3 代码变换过程

以上我们假设循环是一维的情况。如果循环是多维的,且需要对多维循环都进行并行,则对每层要并行的循环都要如上述进行分析,循环分块和补充边界迭代。对两层循环同时并行形成的线程也为二维,同理,对三层循环同时并行会形成三维的线程。CUDA 本身最多只支持二维的线程,分别用 *threadIdx. x* 和 *threadIdx. y* 表示线程在线程中的坐标。可以将 *threadIdx. y* 坐标进一步分离为 *y* 和 *z* 方向的坐标,以实现三维的线程。二维和三维并行后,所需的共享内存交换数组也为二维和三维,数组在每个方向上的大小由该层循环的块大小和边界补充的迭代数之和决定。

3 实 验

我们采用了基于 fermi 架构的 NVIDIA Tesla C2050 来验证我们提出的代码变换方法对性能的影响,驱动版本为 260.19,NVCC 版本为 3.2。实验中采用的代码的详细信息见表 1。最大补充迭代数这一列记录了一系列循环中在两边边界上需要补充的迭代数的最大值,如果多层循环都可并行,则是补充迭代数最多的那个。交换数组的数量这列记录了每个程序所需要的共享内存交换数组的数量。该值取决于同时活跃的需要线程间数据交换的数组定值的数量。

表 1 实验用程序的详细信息

程序名称	语言	来源	循环个数	最大补充迭代数	交换数组的数量	输入规模	块大小
LL18	C	Livermore Loops	2	1	2	4096 ×4096	32 ×16
filter	Fortran	SPEC95 中的 hydro2d	10	4	2	128 ×128 ×128	32 ×16
y _ solve	Fortran	NPB3.3 中的 BT	2	2	2	4096 ×4096	64 ×8

对以上几个程序,我们产生了 2 个版本的代码:single-loop kernel (slk)和 multi-loop kernel (mlk)。single-loop kernel 版本为源程序中的每个循环生成一个内核,multi-loop kernel 版本将若干个循环生成同一个内核。对于 filter,我们还按照文献[9]中的代码复制方法生成一个版本 replicate 以供对比。在 mlk 和 replicate 版本中,如果一个中间结果数组在程序段执行完后不再使用,则不再使用全局内存保存该数组。y _ solve 中的数组原本采用的是适合 CPU 的结构体数组(array-of-structure)的形式,为了适应 GPU,slk 和 mlk 版本皆采用了数组结构体(structure-of-array)的形式,另外,在保证语义的情况下,我们对循环中原有的语句进行了重新排布,以缩短中间数组元素的活跃期,从而减少对共享内存交换数组的需求。在所有的版本中,我们都做了手工优化,使得对全局内存的访问满足了访存合并(coalescing)的要求。

表 2 显示了这 3 个程序的 2 个不同版本的性能和对全局内存的读写次数对比。我们用 GPU 硬件计数器的 gld _ 32bit 和 gst _ 32bit 两个事件来统计对全局内存的读写次数。

表 2　slk 和 mlk 版本对比

	时间(μs)	single-loop kernel 全局内存读	全局内存写	时间(μs)	multi-loop kernel 全局内存读	全局内存写	加速比
ll18	11513	167772160	67108864	10187	161441280	34594560	1.13
y _ solve	16974	116359168	242483200	8682	21233664	176947200	1.96
filter	27549	419430400	201331632	14877	180355072	33554432	1.85

由于中间数组的值在程序段执行完后不再使用,这些数组的值不必再写回全局内存,而是保存在片上的共享内存中,从而大大减少了对全局内存的读写,最终带来了性能的提升。y _ solve的性能基本受限于带宽,因而对全局内存访问的减少与执行时间的减少基本呈线性关系。ll18 和 filter 由于计算占有一定的比重,因此对全局内存访问的减少带来的性能提升相对小些。需要注意的是,补充的边界迭代也会增加一些额外的全局内存读操作,线程块的大小越小,这些额外读的比例越大。ll18 程序本身需要读取的数据较多,而通过循环合并能够减少的数组读取仅占较小比例,而且补充迭代会引入额外的读操作,因此其 mlk 版本对全局内存读操作的减少并不明显。

表 3 显示了 filter 例程用两种方法生成的代码的性能。filter 例程中包含 10 个循环,它们最终可以合并成 1 个循环。文献[9]中提出的代码复制方法,其开销会随着依赖链的增长而呈指数级增长。由于 filter 中 10 个循环的依赖链很长,文献[9]的方法性能远低于 multi-loop kernel,甚至低于 single-loop kernel 版本。文献[9]中的冗余计算大大增加了每个线程的代码的长度,使得运行时指令数大大增加,同时,冗余计算使更多的变量处于活跃状态,增加了寄存器压力,最终导致该版本的占有率只有 mlk 版本的一半。另一方面,由于需要额外的共享内存数组做线程间通讯,mlk 版本对共享内存的需求大于采用文献[9]方法的版本,但在该程序中并没有对占有率造成影响。

表 3　filter 例程的 mlk 和代码复制版本对比

	执行时间(μs)	占有率	共享内存(kB)	寄存器	指令数
mlk	14877	0.667	17860	24	10094392
replicate	32918	0.333	11236	46	23072400

4　结　论

本文提出了一种针对 GPU 体系结构的循环代码合并方法,使得循环合并后会产生迭代间数据依赖的循环能够高效地被映射到 GPU 上。这项技术使得更多的循环可以在合并后被转换为 CUDA 代码,从而有利于发掘原有循环间的数据重用,减少对全局内存的访问。实验数据表明,相对于为每个循环产生单独内核的方法,本文提出的方法生成的包含多个原有 C/Fortran 循环的内核实现了最多高达 1.96 倍的加速。

关于后续工作,我们有如下考虑:(1) 循环合并

会增加内核对共享内存的需求(例如引入了交换数组),过度循环合并可能引起占有率下降,从而导致合并后的内核性能反而不如单独生成内核,或者内核无法运行。我们希望在考虑共享内存资源限制的情况下,研究程序段中的哪些循环可以合并到一起。(2) 共享内存交换数组的数量与原有循环中语句的顺序有关(例如我们对 y _ solve 做的改动),我们希望在分配交换数组前,先对语句顺序进行优化,以减少对交换数组数量的需求。

参考文献

[1] Falcao G, Sousa L, Silva V. Massive parallel LDPC decoding on GPU. In: Proceedings of 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, USA,2008. 83-90

[2] Rehman M S, Kothapalli K, Narayanan P J. Fast and scalable list ranking on the GPU. In: Proceedings of the 23rd International Conference on Supercomputing, San Servolo Island, Italy, 2009. 235-243

[3] Yang Y, Xiang P, Kong J, et al. A GPGPU compiler for memory optimization and parallelism management. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Canada, 2010. 86-97

[4] Baskaran M M, Bondhugula U, Krishnamoorthy S, et al. A Compiler framework for optimization of affine loopnests for GPGPUs. In: Proceedings of the 22nd International Conference on Supercomputing, Island of Kos, Greece, 2008. 225-234

[5] Baskaran M M, Ramanujam J, Sadayappan P. Automatic C-to-CUDA code generation for affine programs. In: Proceedings of 19thinternational conference of Compiler Construction, Paphos, Cyprus: Springer, 2010. 244-263

[6] Wolfe M. The PGI Accelerator programming model on NVIDIA GPUs. http://www. pgroup. com/lit/articles/insider/v1n1a1. htm: The Portland Group, 2009

[7] Belter G, Jessup E R, Karlin I, et al. Automating the generation ofcomposed linear algebra kernels. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, USA, 2009

[8] Yang X, Zhang Y, Xue J, et al. Exploiting loop-dependent stream reuse for streamprocessors. In: Proceedings of 17th international conference on Parallel Architectures and Compilation Techniques, Toronto, Canada, 2008. 22-31

[9] Allen R, Callanhan D, Kennedy K. Automatic decomposition of scientific programs for parallel execution. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, Munich, Germany, 1987. 63-76

[10] Manjikian N, Abdelrahman T S. Fusion of loops for parallelism and locality. *IEEE Trans PDS*, 1997, 8(2): 193-209

A GPU-oriented loop fusion method

Yang Yang[*,**], Cui Huimin[*], FengXiaobing[*]

([*]SKL Computer Architecture,ICT,CAS, Beijing 100190)

([**] Graduate University of Chinese Academy of Sciences, Beijing 100039)

Abstract

To solve the problem that current tools for automatical mapping of C or Fortran programs onto a general purpose graphic processing unit (GPU) mainly aim at generating an independent GPU kernel for each individual loop, which hinders the exploitation of inter-loop data reuse, this paper presents a novel GPU-oriented code transformation approach for loop fusion. The approach integrates strip mining and redundant computation to eliminate data dependence between iterations, and takes advantage of GPU's on-chip shared memory to achieve inter-thread data exchange so as to map this kind of programs onto GPUs effectively. The experiment on various programs demonstrate that the proposed framework can achieve the 1.96-fold speedup because of its reduction of global memory access.

Key words: general purpose graphic processing unit (GPU), loop fusion, parallelization, CUDA, inter-loop data reuse