DOI：10.7652/xjtuxb201310008

应用动态生成树的 GPU 显存数据复用优化

李亮[1]，王恩东[2]，朱正东[1]，颜康[1]，张保[1]，董小社[1]
（1. 西安交通大学电子与信息工程学院，710049，西安；
2. 浪潮集团高效能服务器和存储技术国家重点实验室，250013，济南）

摘要：针对手工优化 GPU(Graphic Processing Unit)显存级数据复用过程复杂和编译时优化数据复用开销过大的问题，提出了一种基于动态生成树在运行时进行数据复用的优化方法，可为程序员提供一种透明且高效简单的优化方式。该方法将已经执行的 GPU 计算任务的数据访问抽象为生成树的叶子节点，利用动态生成树管理 GPU 数据访问信息，实现了运行时 GPU 显存级数据的复用优化，并通过运行时对生成树的搜索和维护，动态地发掘和优化 GPU 显存级数据复用，因此，在编程时不需要进行复杂且困难的数据复用分析，直接调用文中提出的运行时库就能有效减少程序执行过程中 CPU 内存和 GPU 显存之间的冗余数据传输次数，从而提升应用的运行性能。实验结果表明，使用文中提出的优化方法可有效消除未进行数据复用优化的 CPU-GPU 应用程序中的冗余数据传输，最大加速比达原始执行的 3～10 倍，额外开销不到优化后程序总执行时间的 5%。

关键词：GPU 显存；动态生成树；数据复用；数据传输
中图分类号：TP399 **文献标志码**：A **文章编号**：0253-987X(2013)10-0044-07

Optimization Method for Reusing GPU Data Based on Dynamic Spanning Tree

LI Liang[1], WANG Endong[2], ZHU Zhengdong[1], YAN Kang[1], ZHANG Bao[1], DONG Xiaoshe[1]
(1. School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China;
2. State Key Laboratory of High-End Server and Storage Technology, Inspur Group Co. Ltd., Ji'nan 250013, China)

Abstract: An optimization method for reusing graphic processing unit (GPU) data is proposed based on the dynamic spanning tree to solve the problem that the reusing process with manual optimization is complicated and compilation overhead in optimization is expensive. The proposed method is transparent to programmers with simplicity and effectiveness. The approach abstracts the executed accesses of GPU tasks as leaf nodes of a spanning tree, and uses the tree to dynamically manage the access information. Then the identification and optimization for the data reusing is realized by searching and managing the tree. The complex and challenging data reuse analysis is not required since invoking the run-time library is sufficient to reduce the CPU-GPU data transfers. Experiments show that the proposed optimization method can eliminate redundant CPU-GPU data transfers in non-reuse CPU-GPU application programs and can achieve a speedup as large as 3 to 10 times over the original execution. Moreover, the additional cost is just less than 5% of the execution time of the program adopting the proposed optimization.

Keywords: GPU memory; dynamic spanning tree; data reuse; data transfer

---

收稿日期：2012-12-31。 作者简介：李亮(1987—)，男，博士生；董小社(通信作者)，男，教授，博士生导师。 基金项目：国家自然科学基金资助项目(61173039)；国家"863 计划"资助项目(2012AA010904，2012AA01A306)；国家科技支撑计划资助项目(2011BAH04B03)。
网络出版时间：2013-08-23 网络出版地址：http://www.cnki.net/kcms/detail/61.1069.T.20130823.1008.004.html

近年来，GPU（Graphic Processing Unit）由于集成了大量的加速部件，可提供强大的浮点计算能力，已被作为协处理器广泛应用于 CPU-GPU 异构并行系统中，如国防科技大学研制的“天河一号”超级计算机（TH-1A）[1]。然而，将应用的计算密集型任务移植到 GPU 上进行加速时，需要主机-设备端的数据传输[2]，这会造成相对于同构平台上实现应用时的额外通信开销。由于 CPU 和 GPU 间采用数据总线连接，传输带宽小，导致 CPU 和 GPU 间的通信经常成为 CPU-GPU 系统中应用性能提升的瓶颈。同时，程序员在基于 CPU-GPU 异构并行系统实现复杂应用时，往往忽略了 GPU 不同计算任务之间复用数据的可能性，导致可复用数据在主机-设备端进行多次冗余传输，影响了应用整体性能的提升。

针对上述问题，本文提出一种基于动态生成树的运行时数据复用优化方法，将已经执行的 GPU 计算任务的数据访问抽象为生成树的叶子节点，通过运行时对生成树的搜索和维护，动态发掘和优化 GPU 显存级数据复用，可消除 CPU 内存和 GPU 显存间冗余的数据传输，进而提升应用的运行性能。

1　相关工作

目前，GPU 显存级数据复用优化主要采用手工优化[2-4]。程序员手工识别多个 GPU 计算任务复用的数据，提前将其输入显存；对于可复用的计算结果，延迟其写回 CPU 内存的时刻。手工优化虽然可以获得良好的性能，但要求程序员熟悉应用和系统架构，需要大量调整应用代码，操作复杂且优化开销大。

自动数据复用目前主要采用编译优化的方法。例如：针对传统多核处理器的 cache 架构存储，通过置换、翻转、倾斜和分块等技术重构循环，提高 cache 命中率[5-6]；针对异构并行协处理器软件管理的层次化存储，如 GPU 显存、SPM（Scratch Pad Memory），采用图着色 SRF 分派的编译优化方法[7]，识别程序中存在数据复用，消去不同存储层次间冗余的数据传输；针对输入-输入复用，采用路径合并技术；针对输出-输入复用，采用路径消解技术。由于通过分析图获得最优数据分配和任务调度是 NP 问题，因此编译优化的时间开销和空间开销很大。

PGI[8]、HiCUDA[9] 等工具则采用编译优化和手工优化相结合的方法，即编译制导优化。例如，PGI 通过 allocatable 等制导语句显式定义存在数据复用的数组[10]，在编译时自动消除冗余的数据传输。编译优化和手工优化相结合的方法虽然可自动识别数据复用并消除部分冗余数据传输，但需要程序员声明程序中存在数据复用的数组，其实质为手工优化。另外，冯国富等使用软件的方法将 Cell BE 的 LS（Local Store）改造成对用户透明的 cache 存储，以降低用户管理和优化数据复用的难度[11]，然而由于硬件架构的差异，GPU 显存不适宜抽象为 cache。

本文提出的基于动态生成树的 GPU 显存级数据复用优化方法，在运行时执行 GPU 显存级数据复用优化，并以运行时库的形式提供给上层用户，可有效降低程序员优化 CPU-GPU 程序的难度，简化数据复用优化编译器的设计。

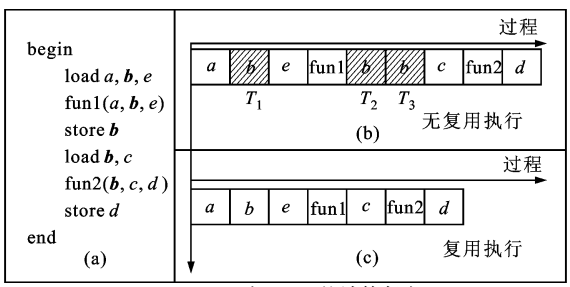
2　GPU 显存级数据复用优化

从应用移植到 GPU 加速执行一般由以下几步构成：①GPU 计算任务中待处理的数据由 CPU 内存传输到 GPU 显存；②GPU 的计算单元对显存上的数据进行处理；③数据处理完毕后，计算任务产生的数据由 GPU 显存传输回 CPU 内存。该方式虽然简便易行，但当 GPU 计算任务之间存在数据共享（即数据复用）时，会导致大量的 CPU-GPU 冗余数据传输。

2.1　数据复用优化

GPU 显存级数据复用可有效消除 CPU-GPU 间的冗余数据传输，提高 GPU 加速应用程序执行的效率。

针对图 1 中（a）所示的程序，未考虑数据复用的执行流程如图 1 中（b）所示，T_1 时刻 fun1 的输入（load b）和 T_2 时刻 fun1 的输出（store b）传输的是同一数据 b，T_2 时刻 fun1 的输出和 T_3 时刻 fun2 的输入传输的也是同一数据 b，显然 fun1 和 fun2 之间存在数据复用，且 T_2 和 T_3 时刻的数据传输是冗余



fun1、fun2 为 GPU 的计算任务

图 1　数据复用示例

的。考虑上述数据复用后，图 1 中(a)所示程序的执行流程如图 1 中的(c)所示。对于图 1 中(a)所示的 GPU 计算任务较少的简单程序，手工优化或编译时自动优化是可行的，然而当程序规模较大时，由于计算任务之间的数据依赖关系复杂，因而手工优化难度大且容易出错，编译时自动优化效率也较低。

2.2 GPU 显存级数据复用框架

针对上述问题，本文提出一种基于动态生成树的运行时 GPU 显存级数据复用优化框架，如图 2 所示。生成树记录 GPU 显存所缓存的有效数据副本对应的数据传输。对于待执行的数据输入/输出或 CPU 内存访问，通过搜索生成树、数据复用分析和冗余分析，判断当前的数据输入/输出是否冗余或 CPU 内存数据是否有效，并控制 GPU 显存和 CPU 内存之间的数据传输，更新生成树中的节点。

本框架在不需要程序员手工优化数据复用且不改变其编程习惯的前提下，基于懒惰一致性[12]和动态生成树的方法，在运行时动态消除 GPU 显存和 CPU 内存之间冗余的数据输入/输出。

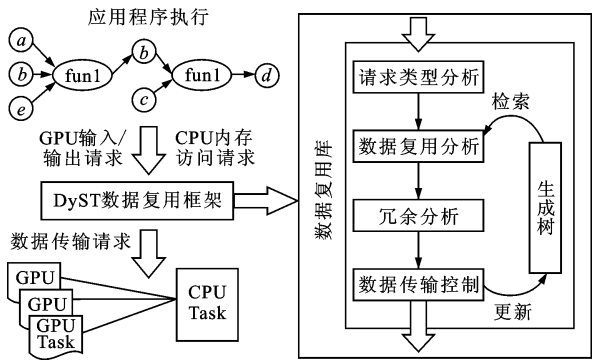


图 2 GPU 显存级数据复用框架

3 基于动态生成树的数据复用优化

由上节可知，在运行时识别 GPU 显存级数据复用和消除冗余数据传输，需要通过搜索生成树判断 GPU 显存中数据的有效性。本节通过动态生成树反映 GPU 显存上数据的有效性，使用动态搜索的方法实现数据复用优化。

3.1 动态生成树

以程序基本结构为框架组织动态生成树中的输入/输出叶子节点，其中执行时间相近的输入/输出所属的叶子节点被组织到同一局部子树中。动态生成树的各节点以及节点之间的关系定义如下：

transfer 节点：生成树的叶子节点，表示 GPU 显存和 CPU 内存之间的一次数据输入/输出，其值指向 GPU 显存中数据副本对应的 CPU 内存中的源数据。

funExec 节点：生成树的中间节点，表示 GPU 计算任务的一次执行，对应的数据输入/输出则通过其子节点 transfer 节点表示。如图 3a 所示，3 个 transfer 节点表示 funExec i 的 GPU 计算任务使用或生成的数据 a、b 和 c 所对应的数据输入/输出。

funWrap 节点和 iteration 节点：生成树的中间节点。funWrap 表示某一层循环的所有 GPU 计算任务和子循环，或主程序中某一段不包含且不属于任何循环的子程序，本文称该组合体为程序块；iteration节点表示该程序块的一次迭代执行。如图 3c 所示，funWrap i 的子节点 iteratoin i 表示 funWrap i 所指程序块的第 i 次迭代；如图 3b 所示，iteration i 的执行内容包括 funExec i 所指 GPU 计算任务的执行、funWrap i_k 所指子循环的执行等。

program 节点：为生成树的根节点，通过 funWrap 节点指向程序中包含 GPU 计算的所有循环。

图 4 详细地表示了以 funExec、funWrap 和 iteration 为根节点的各子树之间的关系。以 funExec 为根节点的 funExec1 和 funExec3 子树表示计算任务 fun1 和 fun3 的执行，只能隶属于程序块 funWrap 的某次迭代，即 iteration i 和 iteration j 的子树。funWrap 子树表示程序块执行，通过 iteration

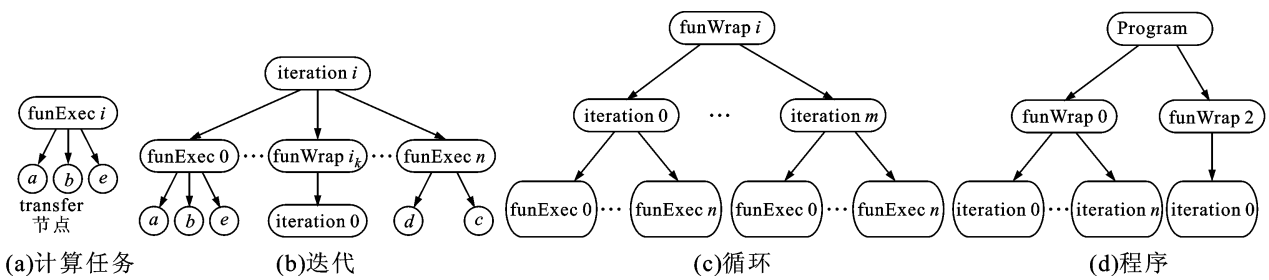


图 3 动态生成树

子树表示其某次迭代。如果程序块为循环嵌套，则表示内层循环的 funWrap j 为表示外层循环的 funWrap i 的某次迭代 iteration i 的子树。

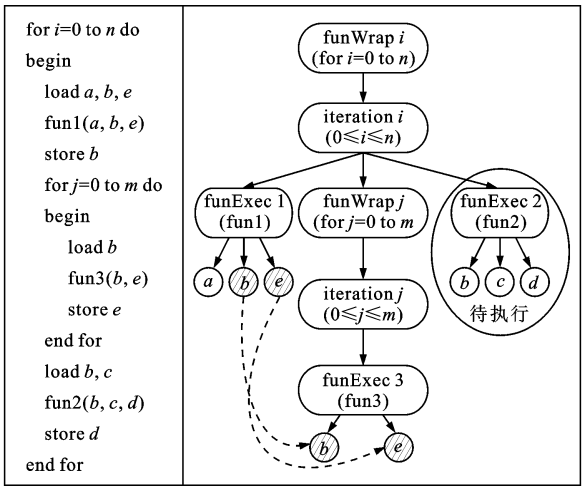


图 4　动态搜索示例

在程序开始执行时，生成树中只有 program 节点。在程序执行过程中，funWrap、iteration、funExec 和 transfer 等随着循环、迭代、GPU 计算任务和数据输入/输出的执行加入到搜索树（具体搜索树生成过程见 3.3 节）。存在的问题是：针对同一内存地址所指数据，生成树可能保存多个 transfer 节点。

为减少运行时的搜索时间开销和维持空间开销，针对同一 CPU 内存地址所指数据 d 的输入/输出，即存在数据复用的输入/输出，在生成树中只保留 1 个 transfer 叶子节点。当搜索到存在数据复用的 transfer 叶子节点时，将该叶子节点从原来的 funExec子树移动到当前 funExec 子树。例如对图 4 所示的程序，当执行到 funExec3（fun3）时，由于 fun3 的输入（load b）与之前 funExec1（fun1）子树复用同一数据 b，所以如图 4 的虚线所示，将 funExec1 下传输 b 的叶子节点从 funExec1 子树移动到 funExec3 子树。

由于每个 GPU 显存上有效的数据副本在树中对应唯一的 transfer 叶子节点，因此对于任意数据 d，如果能在生成树中找到对应的 transfer 叶子节点，说明 d 在 GPU 显存中的数据副本有效，否则 d 在显存中的数据副本无效。

3.2　回溯搜索

本文通过运行时的动态搜索分析当前传输数据在 GPU 显存的有效性。针对提出的动态生成树，采用算法 1 所示的回溯搜索，从最后执行的 funExec 节点出发逆向回溯搜索生成树。这是因为基于数据访问的局部性，可复用的 transfer 节点很可能位于最后执行的 funExec 子树或相邻子树，而回溯搜索可有效利用这一特性来降低搜索开销。

算法 1　DyST_BTS(DyST Back Trace Search)

```
1. input: d   → 显存上待输入/输出的数据
2. input: f   → 当前未命中子树的根节点
3. output: trans   → 与当前输入/输出存在数据复
                     用的 transfer 节点
4. begin
5. if f is program 节点 then
6.    return null node
7. end if
8. pa=f 的父节点;   //记录子树的父节点
9. for each child f1 of pa do
10.    if f1 is not f then
11.       trans=search_in_subtree (d, f1)   //搜
          索 f1 子树
12.       if (trans is not null) then
13.          return trans;
14.       end if
15.    end if
16. end for
17. return DyST_BTS (r, pa)   //递归回溯搜索
18. end
```

例如，对于图 4 所示的程序：当外层循环执行到第 $i=1$ 次迭代、内层循环执行到第 $j=0$ 次迭代时，对于 fun3 的输入 b，首先搜索 fun3 子树；由于在 fun3 子树没有找到复用节点，因此执行 DyST_BTS 算法中的过程，搜索 fun3 的父节点 iteartion 的其他子树，由于没有找到，则继续扩大搜索范围；当回溯到 iteration i 时，在 fun1 找到使 fun3 的输入 b 命中的 transfer 叶子节点 b。

回溯搜索利用程序的局部性收缩搜索范围，使得大部分的搜索在 iteration 子树和 funWap 子树就可以找到。只有使用频率较低的数据对应的输入/输出，以及生成树未命中的输入/输出，才需要搜索整棵生成树，因此对于所有的输入/输出，平均搜索开销维持在较低水平。

3.3　基于动态搜索的 GPU 显存数据复用优化

GPU 计算任务的输入/输出如果在生成树命中，说明与之前的 GPU 输入/输出存在数据复用，且可复用 GPU 显存数据；否则，需要依据传输类型决定 CPU 和 GPU 之间的数据传输。本文对每个 GPU 计算任务使用和生成数据对应的输入和输出

处理如下。

(1)针对数据输入 e_{in},如果在生成树中查找到使其命中的叶子节点,说明 e_{in} 与之前的数据传输存在数据复用,且在它们之间 CPU 没有写数据,GPU 显存的数据副本有效,即 e_{in} 冗余,此时只需将找到的叶子节点移动到当前 funExec 子树下;否则,认为 e_{in} 传输的数据是首次被 GPU 访问或 GPU 显存对应的数据副本无效,需要为 e_{in} 创建新的 transfer 叶子节点,并将数据输入到 GPU 显存。

(2)针对输出 e_{out},如果搜索到与 e_{out} 存在数据复用的叶子节点,则将找到的叶子节点移动到当前子树下;否则,需要为 e_{out} 创建新的叶子节点。并且,e_{out} 输出数据可能不会被 CPU 计算任务立即使用,本文将数据的实际输出推迟到 CPU 计算任务访问之前,以减少 CPU-GPU 数据的传输次数。

对于命中的输入/输出,将命中叶子节点从原来位置移动到当前 funExec 子树下时,可能会导致部分 funExec 等中间节点变为空节点。如果出现空节点,需要将这些节点删除,以消除对这些节点的无效遍历。对于任意输入/输出,除创建叶子节点或移动叶子节点外,还需要创建 funExec、iteration 和 funWrap 等中间节点。

针对 CPU 访问复用数据,本文通过如下方式确保 CPU 访问到复用数据在内存的有效副本。

(1)当 CPU 读 GPU 计算任务生成的数据 d 时,需要保证数据 d 在内存有效。通过回溯搜索判断对应的 GPU 显存中数据副本是否有效,从而间接判断内存数据的有效性。如果未找到数据 d 对应的叶子节点,说明内存数据有效;否则,需要执行 GPU 显存的数据输出以更新内存数据。

(2)当 CPU 写内存数据 d 时,d 在 GPU 显存中的数据副本变为无效。采用推迟更新的方法,先不更新数据 d 在 GPU 显存中的数据副本,而是将生成树中表示 d 有效的 transfer 叶子节点删除,使表示数据 d 在 GPU 显存中的数据副本变为无效;GPU 再次使用到数据 d 时,由于在生成树找不到表示 d 有效的叶子节点,从而执行数据输入来更新 d 在显存中的数据副本。

4 实验验证

本文面向 CPU-GPU 异构并行系统,构建了基于动态生成树的 GPU 显存级数据复用优化原型系统,并使用 FFT、LU、FT 和 CG 等科学计算中广泛使用的测试用例,在 TH-1A 平台对本文方法进行测试和验证。

TH-1A 平台的每个计算节点由 2 个 6 核 Intel Xeon X5670 CPU 和 1 个 Tesla C2050 GPU 组成,Tesla C2050 GPU 采用 PCI-E 2.0×16 总线与 CPU 互连,实测峰值带宽为 8 GB/s。本文针对 TH-1A 平台使用 C++、CUDA 等编程工具,实现了基于动态生成树的 GPU 显存级数据复用库,在运行时动态消除 GPU 显存上冗余的数据输入/输出,并将未消除的数据输入/输出重新定向为对 CUDA Runtime API 的调用。数据复用分析模块由 g++编译,数据传输模块由 CUDA 提供的 nvcc 编译。

4.1 测试用例

实验使用的测试用例包括 FFT、LU、FT 和 CG。FFT 采用原位变换算法,其多次迭代被划分到 9 个 GPU 计算核心;LU 的矩阵分解采用杜里特算法(Doolittle algorithm),通过初等行变换将矩阵变成一个上三角矩阵,计算部分被划分到 2 个 GPU 计算核心,执行时交替调用这 2 个计算核心直到结束;FT 和 CG 为经典测试集 NPB[13] 的子程序,本文使用的 FT 和 CG 来源于 HPCGPU[14]。测试用例的主要参数如表 1 所示,其中 FFT 和 LU 计算规模的基本单位为单精度浮点数,FT 和 CG 使用 NPB 定义的计算规模 S、W 和 A[13]。

表 1 测试用例主要参数

用例	GPU 任务数	GPU 输入次数	GPU 输出次数	计算规模
FFT	9	18	9	$256K\sim16M$
LU	1 022～7 166	1 022～7 166	1 022～7 166	512×512～3 585×3 584
FT	76	246	193	S, W, A
CG	15	75	60	S, W, A

注:$K=1\ 024$; $M=1\ 024^2$。

设定 non-opt 是不考虑 GPU 显存级数据复用的程序执行结果;dystr-opt 是针对 non-opt 程序、使用本文数据复用库进行数据访问优化后的执行结果;manual-opt 是手工优化数据复用和消除冗余数据传输后的执行结果。FFT 的程序来自参考文献[15];FT 和 CG 的程序来自 HPCGPU;LU 的程序为由作者手工将数据传输语句提取到循环外,以消除冗余数据传输。

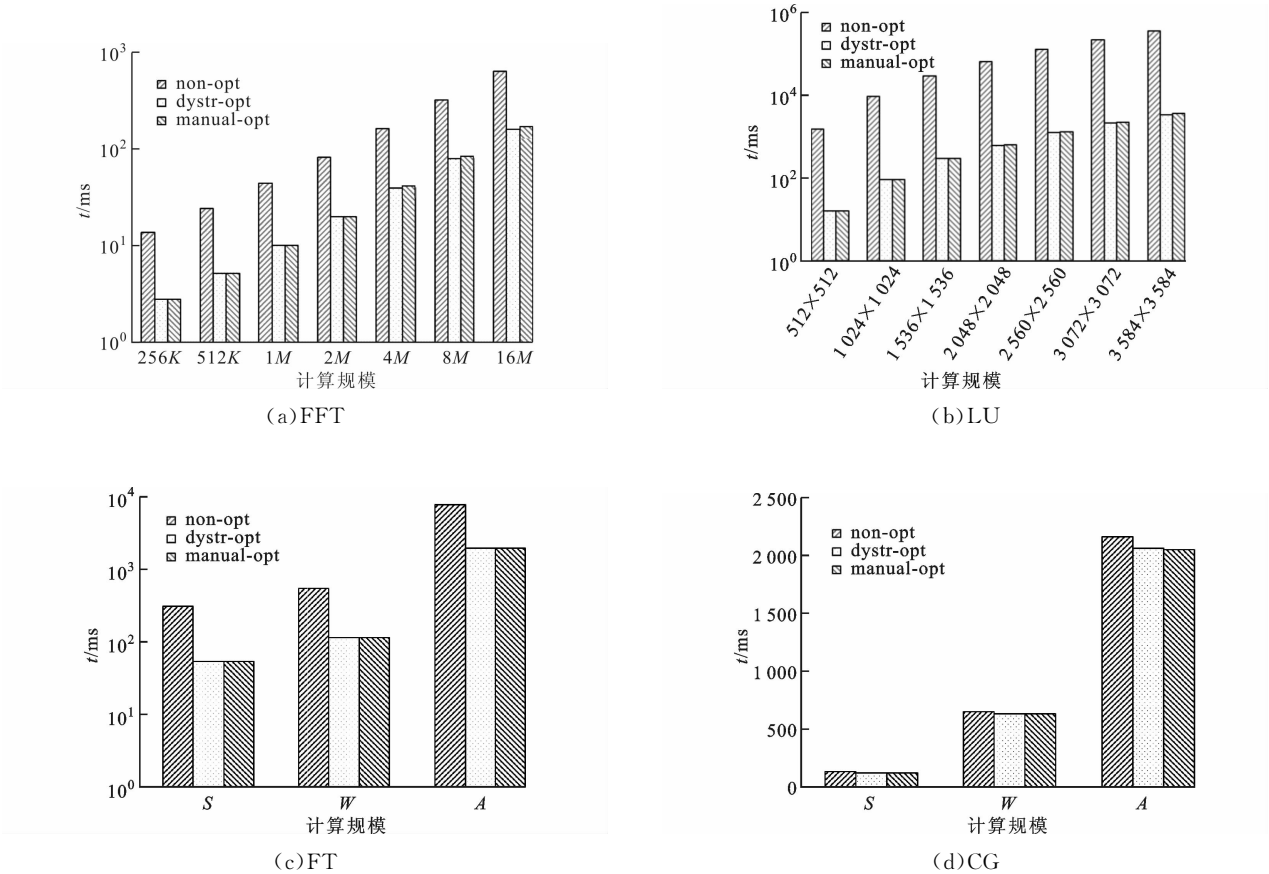
4.2 实验测试结果

针对表 1 所示的计算规模,分别执行 FFT、LU、FT、CG 等应用的 non-opt 程序、dystr-opt 程序

和 manual-opt 程序，实验结果如表 2 和图 5 所示。

表 2 是每个应用在小规模、中等规模和大规模计算时的数据传输次数。与 non-opt 相比，dystr-opt 程序的执行消除了 90%以上的数据传输；与 manual-opt 相比，dystr-opt 没有增加多余的数据传输。

由图 5 可以看到：dystr-opt 执行与 non-opt 执行相比，对 LU 快 10 倍以上，对 FFT 快 5 倍以上，对 FT 快 3 倍以上；对于 CG，由于数据传输时间占总时间的比例较小，dystr-opt 执行虽然消除了 GPU 显存和 CPU 内存之间的冗余数据传输，但与 non-opt 相比没有获得明显的性能提升，而 dystr-opt 的执行时间非常接近于 manual-opt 的执行时间。如果使用 $(T_{\text{dystr-opt}}-T_{\text{manual-opt}})/T_{\text{dystr-opt}}$ 表示 dystr-opt 和 manual-opt 的性能差距（即本文优化方法的额外开销），对于 FFT、LU、FT 和 CG 的大部分计算规模来说，二者的性能差距仅在 0～5%之间。



(a)FFT　(b)LU　(c)FT　(d)CG

图 5　FFT、LU、FT 和 CG 的执行时间对比

表 2　GPU 显存和 CPU 内存之间在不同计算规模下的数据传输次数

用例	N_{FFT}			N_{LU}			N_{FT}			N_{CG}		
	256K	2M	16M	512^2	$2\,048^2$	$3\,584^2$	S	W	A	S	W	A
non-opt	18	18	18	2 044	8 188	14 332	439	439	439	135	135	135
dystr-opt	2	2	2	2	2	2	7	7	7	7	7	7
manual-opt	2	2	2	2	2	2	7	7	7	7	7	7

注：$K=1\,024$；$M=1\,024^2$。

5　结　语

本文提出了一种基于动态生成树的 GPU 显存级数据复用优化方法，通过将 GPU 显存级数据复用识别和冗余数据传输消除抽象为对动态生成树的搜索和维护，在省去编程时复杂且困难的数据复用分析的同时，有效减少了运行时 CPU 内存和 GPU 显存之间的数据传输次数。在 TH-1A 平台上的实

验结果表明,该方法可使未优化程序获得与手工优化的执行相接近的性能,降低了将应用移植到GPU进行加速计算的难度。

本文下一步的工作将在目前对CPU-GPU异构并行系统平台研究的基础上,对该优化方法做进一步的扩展和完善,并应用于Intel MIC平台,使优化方法更具可扩展性和普适性。

致谢 本文工作由高效能服务器和存储技术国家重点实验室提供支持,实验部分在国家超级计算天津中心(NSCC-TJ)的TH-1A上完成,谨致谢忱。

参考文献:

[1] YANG Xuejun, LIAO Xiangke, LU Kai, et al. The TianHe-1A supercomputer: its hardware and software [J]. Journal of Computer Science and Technology, 2011, 26(3): 344-351.

[2] ZHU Xiaoqian, LIU Xin, MENG Xiangfei, et al. Performance analysis and optimization of gyrokinetic toroidal code on TH-1A supercomputer [C]// Proceedings of 2nd International Conference on Electrical and Control Engineering. Piscataway, NJ, USA: IEEE, 2011: 6027-6031.

[3] FENG Xiaowen, JIN Hai, ZHENG Ran, et al. Optimization of sparse matrix-vector multiplication with variant CSR on GPUs [C]// Proceedings of 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS). Piscataway, NJ, USA: IEEE, 2011: 165-172.

[4] WU Haicheng, DIAMOS G, Wang Jin, et al. Optimizing data warehousing applications for GPUs using kernel fusion/fission [C]// Proceedings of IEEE 26th International Parallel and Distributed Processing Symposium, Workshops & PhD Forum (IPDPSW). Piscataway, NJ, USA: IEEE, 2011: 2433-2442.

[5] WOLF M E, LAM M S. A loop transformation theory and an algorithm to maximize parallelism [J]. IEEE Trans on Parallel Distrib Syst, 1991, 2(4): 452-471.

[6] WOLF M E, LAM M S. A data locality optimizing algorithm [C]// Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI). Washington, DC, USA: ACM, 1991: 30-44.

[7] SMITH M D, RAMSEY N, HOLLOWAY G H. A generalized algorithm for graph-coloring register allocation [C]// Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI). Washington, DC, USA: ACM, 2004: 277-288.

[8] WOLFE M. Implementing the PGI accelerator model [C]// Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU). Washington, DC, USA: ACM, 2010: 43-50.

[9] HAN T D, ABDELRAHMAN T S. hiCUDA: high-level GPGPU programming [J]. IEEE Trans on Parallel Distrib Syst, 2011, 22(1): 78-90.

[10] WOLFE M. Optimizing data movement in the PGI accelerator programming model [EB/OL]. [2012-06-09]. http://www.pgroup.com/lit/articles/insider/v3n1a1.htm.

[11] 冯国富,董小社,胡冰,等. 一种支持多种访存技术的CBEA片上多核MPI并行编程模型 [J]. 计算机学报, 2008, 11(31): 1965-1974.
FENG Guofu, DONG Xiaoshe, HU Bing, et al. A MPI parallel programming model for CBEA based on hybrid memory access technology [J]. Chinese Journal of Computers, 2008, 31(11): 1965-1974.

[12] IOSEVICH V, SCHUSTER A. A comparison of sequential consistency with home-based lazy release consistency for software distributed shared memory [C]// Proceedings of 18th Annual ACM International Conference on Supercomputing (ICS). Washington, DC, USA: ACM, 2004: 306-315.

[13] NAS Parallel Benchmarks Team. Problem sizes and parameters in NAS Parallel Benchmarks [EB/OL]. [2012-06-11]. http://www.nas.nasa.gov/publications/npb_problem_sizes.html.

[14] YARROW M, KUSZMAUL C. NPB2.3-FT Benchmark-C+CUDA [EB/OL]. [2012-06-11]. http://hpc gpu.codeplex.com/releases/view/34770.

[15] 张宝,董小社,白秀秀,等. CPU-GPU系统中基于剖分的全局性能优化方法 [J]. 西安交通大学学报, 2012, 46(2): 17-23.
ZHANG Bao, DONG Xiaoshe, BAI Xiuxiu et al. Profiling based optimization method for CPU-GPU heterogeneous parallel processing systems [J]. Journal of Xi'an Jiaotong University, 2012, 46(2): 17-23.

(编辑 葛赵青 武红江)