一种针对异构计算平台的编译期优化方法

裴 根，朱振国，孔 颖
（重庆交通大学 信息与工程学院，重庆 404100）

摘 要：针对 OpenCL（open computing language）编译时期的特有模式，提出了一种新的针对异构计算平台的编译期优化方法。该方法根据设备端和主机端的各自特点，将设备端的一些冗余操作提到主机端或者新的设备端 kernel 中去执行，以达到降低存储器读写的目的。这种方法充分利用了异构计算平台的特点，较传统优化方法相对灵活。大多数情况下能有效提高 OpenCL 的运行速度，测试用例中在应用原有编译器优化的基础上最快提高了 270%。

关键词：编译优化；OpenCL；异构计算；冗余代码外提
中图分类号：TP314 **文献标志码**：A **文章编号**：1001-3695(2013)05-1402-03
doi:10.3969/j.issn.1001-3695.2013.05.030

Compile-time optimization method for heterogeneous computing platforms

PEI Gen, ZHU Zhen-guo, KONG Ying
(*School of Information & Engineering*, *Chongqing Jiaotong University*, *Chongqing* 404100, *China*)

Abstract: For the OpenCL(open computing language) peculiar mode, this paper proposed a new compile-time optimization methods for heterogeneous computing platform. According to the characteristics of each device-side and host-side, this method took the redundant operation of the device-end to the host-side or new device-side to operate, which reduced the times of read and write memory. This method took full advantage of the characteristics of the heterogeneous computing platforms, relatively flexible compared to traditional optimization methods. It can effectively improve the overall speed in many cases, the fastest increase of 270% in test samples.

Key words: compiler optimizations; OpenCL; heterogeneous computing; redundant code motion

单核 CPU 有效地推动了程序性能的提升与软件成本的降低。但能耗和散热问题限制了时钟频率的提高和单 CPU 在每个时钟周期的执行能力，因此仅靠硬件性能的提升来提高应用程序的运行速度已经远远不够。近几年来，凭借强大的计算能力和卓越的性价比，GPU 吸引了越来越多的通用计算。为充分发挥异构处理平台下各种设备性能的潜力，同时使程序具备可移植性，Khronos Group 推出了开放式计算语言 OpenCL，该语言是针对异构并行计算编程的免费工业标准。虽然 OpenCL 支持 Cell、DSP 等其他处理器，但其最初设计目的是为了方便 CPU 与 GPU 之间的异构编程。OpenCL 编程模型虽然一定程度上降低了 GPU 编程的学习曲线，但在编写高性能通用计算程序时仍需考虑如何将代码有效地映射到 GPU 硬件上以便于执行。并且由于并行计算的特点，一般程序员应显式地控制高速存储器的使用，直接访问全局存储器的时间通常达到几百个时钟周期，如果没有对 GPU 硬件进行深入的了解，很难写出高效的程序。另外，为了硬件成本与运行效率的折中，GPU 暂时没有提供强大的控制逻辑和有效的全局通信能力[1,2]，这使得在某些情况下程序运行十分耗时。

近几年来，针对异构平台的编译期优化方法技术有了较大的发展。针对单个 work-item 目前主要应用传统的串行程序优化方法，针对多个 work-item 之间的关系进行考虑的优化方法有多面体模型[3,4]、loop tiling[5] 等，都能获得较好的加速比，但未见有文献利用设备端与主机端之间关系进行优化的方法。如图 1(a) 所示的这段代码，如果直接将其并行化处理，即每个 work-item 负责计算一个 b 值，如图 1(b) 所示。这样做可以获得较高的加速比，但深入分析不难发现这段代码中每个 work-item 都要计算一遍变量 dis 的值，共需要读取 6 次全局存储器。虽然可以通过写入本地存储器在一定程度上降低访问时间，但本地存储器的大小是有限的，需要有效地利用。实际上，每个 work-group 只需计算一次 dis 的值就可以了，这样可以将计算 dis 的次数由 $M \times N$ 次降为 M 次。图 1(c) 和 (d) 分别表示将计算 dis 变量的代码进行外提，运行在主机端与设备端的情况。在执行图 1(e) 所示代码前，根据 work-group 的大小及 kernel 参数中内存的类型等运行时信息，动态指定提出的代码运行在设备端还是主机端，最终将读取全局存储器的次数从 $8 \times M \times N$ 次降低为 $7 \times M + 3 \times M \times N$ 次。这种方法在一定程度上降低了异构平台程序开发对硬件的依赖，结合主机端与设备端各自的优势，将存储器冗余读取部分进行外提，以产生良好的代码并提高程序运行的效率。

1 框架设计

1.1 程序模型

OpenCL 程序一般由主机端程序（C/C++）和设备端 kernel 程序（.cl）组成。Kernel 程序在主机端程序运行时编译。

收稿日期：2012-06-26；**修回日期**：2012-07-24

作者简介：裴根（1987-），男，吉林松原人，硕士研究生，主要研究方向为编译技术、机器学习（hurrican6@163.com）；朱振国（1969-），男，重庆人，副教授，硕导，硕士，主要研究方向为编译技术、高性能计算；孔颖（1989-），男，山东枣庄人，硕士研究生，主要研究方向为高性能计算.

主机端程序使用 g++、msvc++ 等 C/C++ 编译器即可编译,设备端代码需要 OpenCL 编译器进行编译,如图 2 所示。OpenCL 编译器的前端由标准 C 语言和相应的 OpenCL 扩展组成,中端大多数使用 LLVM[6] 低级虚拟机进行中间代码优化。Kernel 程序通过 OpenCL 运行时接口提交给编译器。前端将 OpenCL 源代码转换为 LLVM 的中间代码,同时前端提供了额外的数据类型支持、额外的关键字以及内建函数支持,它的输出与内建函数库由链接器链接,再通过优化器生成优化过的 LLVM 中间代码,之后根据设备端类型生成相应的机器代码,如 AMD IL、PTX、X86 等。

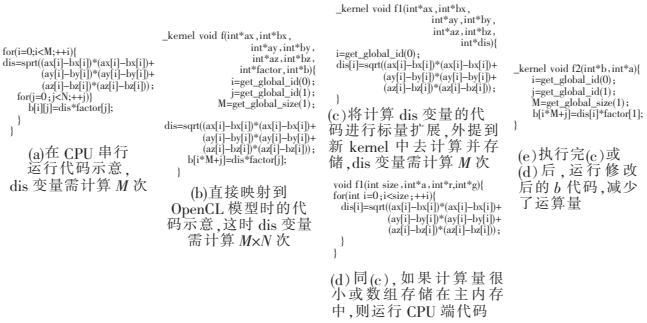

图 1　代码

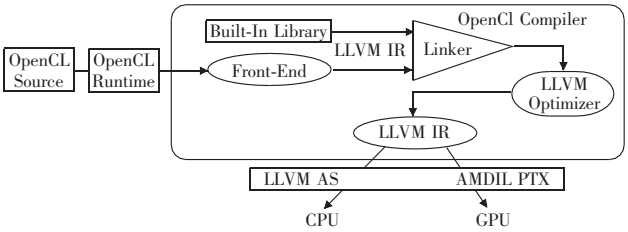

图 2　OpenCL 编译器工具链

本优化程序共分为两个模块,信息收集器模块用于收集各种运行时信息,优化器模块用于对生成的 LLVM 中间代码进行实际优化工作。

收集器仅进行关键信息的收集、保存,以及判断所提出的代码运行在主机端还是设备端,它是线性时间复杂度的轻量级收集器,占用的运行时开销比重非常小。如图 3 所示,本文在用户程序层与 OpenCL 运行时层之间多加了一层接口,这可以使用 HOOK 技术或提供的接口来实现。在用户层调用 OpenCL 运行 API 时进行截获,记录各种有用的运行时信息,比如 kernel 的个数,每个 kernel 的参数个数以及参数地址是在主机端内存还是在设备端内存等。它还负责检查 kernel 的参数地址是否相同,如果相同则不进行优化。

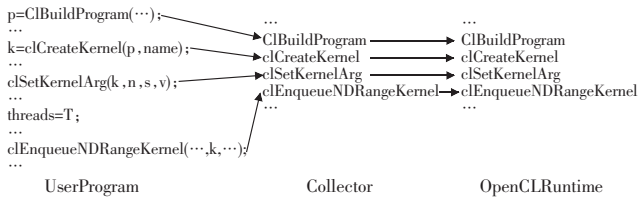

图 3　信息收集器示意图

优化器的主要工作是进行 kernel 代码的分析,找到可以进行外提的代码,将原程序修改,并把可外提代码生成新的程序。新的程序包括主机端程序和设备端程序两种。主机端程序生成为动态链接库,设备端程序生成为新的 kernel 函数。通过收集器得到的信息,在用户程序调用 OpenCL 的 clEnqueueNDRangeKernel 函数真正运行修改后的 kernel 程序之前,动态决定并运行新的主机端或者设备端程序。优化器在调用 OpenCL 运行时函数 clBuildProgram 时工作,理想情况下应该如图 4 所示,得到 LLVM 中间代码后进行优化工作,然后再传回优化后的代码并生成提出的代码。但由于没有接口,无法得到 LLVM 中间代码,因此使用了迂回办法。Libclc[7] 及 Clang[8] 是 LLVM 开源项目的开源子项目:Libclc 是 OpenCL kernel 的一个标准库,目前支持生成为 PTX[9] 低级代码;Clang 是一个生成 LLVM 中间代码的语言前端,目前支持 C/C++、Obj C、OpenCL 等语言。所以可以以离线的方式先将 kernel 代码进行优化,生成 PTX 代码。之后再通过调用 OpenCL 运行时的 clCreateProgramWithBinary 函数执行生成的 PTX 代码。

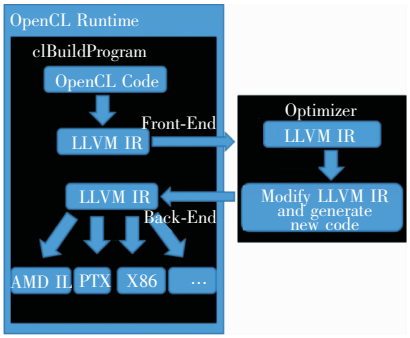

图 4　理想状况下的优化器示意图

1.2　运行代码的选择

在运行时,已经得到了 kernel 维度的大小、总共迭代的次数,以及参数地址是在 GPU 内存还是 CPU 内存等信息。为了确定运行在设备端还是主机端,主要参考以下几个方面:

目前主机端有些可以直接访问设备端全局存储器,有些不可以。如果不可以,则需要通过 PCI-E 总线从设备端存储器拷贝到主机端存储器,运行完主机端代码后,再拷贝回去,这是十分耗时的;如果可以,虽然不需要内存拷贝,但也要经过 PCI-E 总线。反过来设备端访问主机内存也存在同样的情况。当然,CPU 与 GPU 融合处理器,由于共同使用全局存储器,相互访问无须经过 PCI-E,一定程度上降低了内存间传输的瓶颈。

代码提出后可能计算量变得很小,如果运行设备端 kernel 代码,可能仅仅启动 kernel 的时间就会占据相当比重的总计算时间。根据目前设备的不同,计算单元最多不超过几千,如果迭代次数小于计算单元数,整个设备资源难以充分利用,因此这种情况建议在主机端运行。

综合以上信息,本文优先保证存储器访问不会出现主机端与设备端相互访问存储器的情况,其次再保证计算量的饱和度。

1.3　外提算法

本算法从循环不变代码外提[10](loop-invariant code motion)算法得到启示,循环不变代码外提用于标志循环中的每一个迭代都产生相同值的计算,并将它们移到循环之外,如图 5 所示。

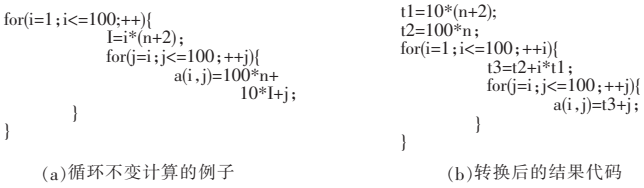

图 5　循环不变计算的例子与转换后的结果代码

假定已经进行过控制流分析,并标志出了各个循环,也通过数据流分析计算出了 ud 链,那么就知道哪些定值会对一个给定变量的使用有影响。计算循环不变指令集合的算法如下:

a)在下面的每一步中,记录指令被确定为循环的循环不

变量时的次序。

b)标志指令中的常数为循环不变量。

c)标志那些到达它们的定值都位于循环之外的所有操作数。

d)标志循环不变指令,它们是步骤 a)所有操作数都标志为循环不变量的指令,或步骤 b)只有一个到达定值且此定值已经标志为此循环中的循环不变量,并且在循环内该指令之前没有使用由该指令赋值的变量。

重复上面的步骤 b) ~ d),直到没有新的操作数或指令被标志为迭代中的不变量为止。

对于每一条已被标志为循环不变量的指令 v,如果它满足:a)定值 v 的语句所在的基本块是循环中所有使用了 v 的基本块的必经节点;b)定值 v 的语句所在的基本块是循环中所有出口基本块的必经节点,则将它们按被标志的次序移动到循环前置块。这里本文加入限制条件:c)如果 load 指令存在依赖或反依赖,或者 store 指令存在反依赖或输出依赖,则不进行外提。

对于 OpenCL 程序模型的特点,要应用上面的代码存在两个问题:a)由于 OpenCL 没有显式的循环语句,所以不能完全判断出哪些定值是在循环之外的;b)由于代码外提可能需要作标量扩展,这样会增加程序的内存需求且访问全局存储器可能比计算更加耗时,并非所有的代码外提都是有利的。

为了便于表述,只考虑使用全局存储器和寄存器的情况。在 OpenCL 编程模型中,相当于外层循环被硬件并行执行,维度是 1 ~ 3 维,因此假设循环全部是顺序执行同样可以保证执行结果的正确性。通过在外层新建立三层循环,每层循环的索引初始值是 0,终止值是 get_global_size(第 i 层 - 1) - 1 值,然后将所有的 get_global_id(dim)替换为相应维度循环的索引号,get_local_id(dim)替换为循环索引号% lsize[dim],变换后代码如下所示,可知新加入的外层循环是紧嵌循环且是规范化的。之后通过控制流分析得到各个循环,数据流分析计算出程序的 ud 链。通过循环不变代码外提算法分析新加入的几层循环,可以得到所有循环不变代码指令,如下所示,s4 为循环不变指令。

```
const size_t size[3] = {get_global_size(0),
                        get_global_size(1),
                        get_global_size(2)};
const size_t lsize[3] = {get_lobal_size(0),
                         get_lobal_size(1),
                         get_global_size(2)};
for(int idx[0] = 0;idx[0] < size[0]; ++idx[0]){
  for(int idx[1] = 0;idx[1] < size[1]; ++idx[1]){
    for(int idx[2] = 0;idx[2] < size[2]; ++idx[2]){
s1    i = idx[0];
s2    j = idx[1];
s3    M = size[1];
s4    dix = sqrt((ax[i] - bx[i]) * (ax[i] - bx[i]) +
                 (ay[i] - by[i]) * (ay[i] - by[i]) +
                 (az[i] - bz[i]) * (az[i] - bz[i]));
s5    b[i * M + j] + = dis * factor[j]
      }
    }
  }
```

定理 1 对所有循环不变代码进行标量扩展并外提所有循环不变代码,且按照外提的顺序执行,程序的执行结果不变。

证明 根据标量扩展的定义可知,对所有循环不变代码进行标量扩展[11]总是安全的。循环不变代码外提算法的限制条件 b)保证了所有的可外提代码一定不会被循环中条件语句包含,因此循环不变代码中只有标量和矢量两种变量的赋值语句且在同一个基本块。由限制条件 a)和 c)可知,所有循环不变代码中的定值是循环内所有引用此变量的唯一定值,且此变量的值在内循环不会改变。因为循环不变代码中矢量的索引都是由外循环携带的,且整个矢量的值在内循环不会改变,所以将其索引的维度扩展为外循环的层数并外提进行计算。这样相当于先将内循环需要使用的值按照外循环的迭代序号全部进行记录,每次使用时读取相应迭代序号内存的值,所以执行的结果仍然不变。对于标量,因为整个内循环每一次运行引用的都是同一个值,所以同样以外循环层数为索引的维度进行扩展,同理,执行结果也是不变的。

经过以上过程,会产生很多可选择的外提代码,根据定理 1 可知对所有的循环不变代码进行标量扩展[11](scalar expansion),就可以外提所有不变代码。但是并非外提任意的代码都是值得的,有些情况下访问全局存储器等待的时间会高于直接计算的时间,性能可能反而下降。可以定义需要外提的代码指令集合,即:

a)设循环不变指令集合为 T,T 中所有循环携带的变量(即数组形式,并以归纳变量为索引的变量)集合为 S。

b)在 T 中标记所有的变量 v 所在定值指令,如果 v 的最终定值存在 S 集合中元素的使用,记为 V。

c)如果存在 V 中的定值 v,在内层循环有对此定值的引用,则进行标量扩展,并将 v 加入到集合 N,T 变为 T-N, goto b);如果都不存在,goto d)。

d)计算 N 集合指令的 ud 链,向上递归找到 N 集合定值指令在循环不变指令集合中所有的使用,将那些给使用定值的指令加入到 N 集合中。如果给使用定值的指令在内层循环没有使用,则将指令从 T 集合中移出。

e)将 N 集合中的指令进行外提生成新设备端或主机端代码并修改原来代码。T-N 集合的指令不进行任何更改。

2 实验结果及分析

实验环境:CPU 为 AMD Phenom(tm) Ⅱ ×4 945 处理器 3.0 GHz,8 GB 1333 MHz 内存,NVIDIA GeForce GTX 460 显卡(768 MB 独立显存),在 Ubuntu 11.04 ×64 系统下测试。

表 1 为相同计算规模下进行外提后效率的提升。从表 1 可看出,在代码可外提的情况下,可有效提升程序的运行效率,优化器耗时在 10 ~ 150 ms。在 MapKernel 函数中,y_low、y_up 与 delta_alpha_low、delta_alpha_up 相乘表达式可直接外提计算出结果。在 Becke 函数中,可将计算 points[i]与 atomA 距离的表达式进行外提,但由于 atomA 与 points[i]在控制流下面的基本块还有使用,所以无法在原 kernel 中删除,提升效率较低。MRI-FHd 则是由于程序的主要瓶颈在累加 rFHd 和 iFHd 时使用了原子操作。

表 1 相同计算规模下进行外提后效率的提升

用例	global_size	local_size	效率提升/%
本文示例	8192 ×8192	1024 ×1	270
MapKernel[12]	6144 ×6144	256 ×1	20.4
MRI-FHd[13]	2048 ×524288	8 ×64	0.68
Becke[14]	262144 ×32	32 ×32	4.7

3 结束语

本文通过结合主机端与设备端各自的优势,将存储器冗余读取部分外提,并根据数据规模的大小,决定外提代码是在主机端运行还是在设备端运行。该方法在较大程度上提高了程序的运行效率,降低了编写高效代码的难度。GPU 的逻辑控制相对 CPU 来说比较薄弱,相信通过将复杂逻辑控制部分外提放入 CPU 端,也能达到很好的效果。 (下转第 1409 页)

和 Pig,而 Hive 和 Pig 的执行时间相差不大,这说明 ShareOpt 优化的性能优势在数据规模增长时仍然能够平稳地保持。

4 结束语

本文提出了一种基于查询共享的查询优化技术——ShareOpt 优化,将一组查询内具有相同模式的子查询合并在一个 MapReduce 作业中执行,从而减少了执行同样一组查询所需的作业数。通过与 Hive 和 Pig 对比发现,ShareOpt 优化具有更高的处理效率和良好的扩展性。当查询数量大于 2 时,ShareOpt 优化在查询处理时间上比 Hive 和 Pig 有明显的优势。下一步工作将查询的综合代价考虑在内,研究在 MapReduce 环境下基于综合代价估计的多查询优化技术。

参考文献:

[1] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[C]//Proc of the 6th Conference on Symposium on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association. 2004: 10-10.

[2] BU Ying-yi, HOWE B, BALAZINSKA M, *et al*. HaLoop: efficient iterative data processing on large clusters[J]. Proceedings of the VLDB Endowment, 2010,3(1-2): 285-296.

[3] PAN Wei, LI Zhan-huai, WU Sai, *et al*. Evaluating large graph processing in MapReduce based on message passing[J]. Chinese Journal of Computers, 2011,34(10): 1768-1784.

[4] ELTABAKH M Y, TIAN Yuan-yuan, ÖZCAN F, *et al*. CoHadoop: flexible data placement and its exploitation in Hadoop[J]. Proceedings of the VLDB Endowment, 2011,4(9): 575-585.

[5] KANG U, TSOURAKAKIS C E, FALOUTSOS C. PEGASUS: a peta-scale graph mining system implementation and observations[C]// Proc of the 9th IEEE International Conference on Data Mining. Washington DC: IEEE Computer Society, 2009: 229-238.

[6] LIN J, SCHATZ M. Design patterns for efficient graph algorithms in MapReduce[C]//Proc of the 8th Workshop on Mining and Learning with Graphs. New York: ACM Press, 2010: 78-85.

[7] Apache Software Foundation. Apache Mahout software[EB/OL]. (2012)[2012-05-01]. http://mahout.apache.org/.

[8] Apache Software Foundation. Apache Hadoop software[EB/OL]. (2012-03-19)[2012-05-01]. http://hadoop.apache.org/.

[9] ABOUZEID A, BAJDA-PAWLIKOWSKI K, ABADI D, *et al*. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads[J]. Proceedings of the VLDB Endowment, 2009,2(1): 922-933.

[10] THUSOO A, SARMA J S, JAIN N, *et al*. Hive: a warehousing solution over a Map-Reduce framework[J]//Proceedings of the VLDB Endowment, 2009,2(2): 1626-1629.

[11] OLSTON C, REED B, SRIVASTAVA U, *et al*. Pig latin: a not-so-foreign language for data processing[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2008: 1099-1110.

[12] DITTRICH J, JORGE-ARNULFO Q R, ALEKH J, *et al*. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)[J]. Proceedings of the VLDB Endowment, 2010,3(1-2): 515-529.

[13] LIN Yu-ting, AGRAWAL D, CHEN Chun, *et al*. Llama: leveraging columnar storage for scalable join processing in the MapReduce framework[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2011: 961-972.

[14] AFRATI F N, ULLMAN J D. Optimizing joins in a Map-Reduce environment[C]//Proc of the 13th International Conference on Extending Database Technology. New York: ACM Press, 2010: 99-110.

[15] Transaction Processing Performance Council. TPC benchmark H version 2.14.4[EB/OL]. (2011)[2012-05-01]. http://www.tpc.org/tpch/.

[16] LEE Ru-bao, LUO Tian, HUAI Yin, *et al*. YSmart: yet another SQL-to-MapReduce translator[C]//Proc of the 31st IEEE ICDCS. Washington DC: IEEE Computer Society, 2011: 25-36.

[17] Apache Software Foundation. Running TPC-H queries on Hive[EB/OL]. (2009-08-11)[2012-05-01]. https://issues.apache.org/jira/secure/attachment/12416257/TPC-H_on_Hive_2009-08-11.pdf.

[18] Apache Software Foundation. Running the TPC-H queries on Pig[EB/OL]. (2012-04-06)[2012-05-01]. https://issues.apache.org/jira/browse/PIG-2397.

(上接第 1404 页)

参考文献:

[1] The OpenCL specification v1.2[R]. [S.l.]: Khronos OpenCL Working Group, 2011.

[2] AMD accelerated parallel processing OpenCL programming guide[R]. [S.l.]: Advanced Micro Devices, Inc, 2011.

[3] BONDHUGULA U, HARTONO A, RAMANUJAM J, *et al*. A practical automatic polyhedral parallelizer and locality optimizer[C]//Proc of PLDI'08. New York: ACM Press, 2008: 101-113.

[4] DONG Guo-xing, CHEN Kai, ZHU Er-zhou, *et al*. A translation framework for virtual exection environment on CPU/GPU architecture[C]//Proc of PAAP'10 Conference Publishing Services. 2010: 130-137.

[5] RYOO S, RODRIGUES C, BAGHSORKHI S, *et al*. Optimization principles and application performance evaluation of a multithreaded GPU using cuda[C]//Proc of PPoPP'08. New York: ACM Press, 2008: 20-23.

[6] Computer Science Department of the University of UIUC. The LLVM compiler infrastructure[EB/OL]. (2012-05-22)[2012-06-21]. http://www.llvm.org.

[7] Computer Science Department of UIUC. Libclc[EB/OL]. (2012-06-15)[2012-06-21]. http://libclc.llvm.org.

[8] Computer Science Department of UIUC. Clang: a C language family frontend for LLVM[EB/OL]. (2012-05-21)[2012-06-21]. http://clang.llvm.org.

[9] NVIDIA. Parallel thread execution ISA v3.0. [R]. [S.l.]: NVIDIA Corporation, 2012.

[10] MUCHNICH S S. 高级编译器设计与实现[M]. 赵克佳,沈志宇,译. 北京:机械工业出版社,2005.

[11] ALLEN R, KENNEDY K. 现代体系结构的优化编译器[M]. 张兆庆,乔如良,冯晓兵,等译. 北京:机械工业出版社,2004.

[12] HERRERO-LOPEZ S. Multiclass support vector machine[M]//GPU Computing Gems Emerald Edition. Massachusetts: Morgan Kauffman Publishers, 2011: 293-311.

[13] KIRK D B, HWU W W. 大规模并行处理器编程实战[M]. 陈曙晖,熊淑华,译. 北京:清华大学出版社, 2010.

[14] LUEHR N, UFIMTSEV I S, MARTINEZ T J. Dynamical quadrature grids: application in density functional calculations[M]//GPU Computing Gems Emerald Edition. Massachusetts: Morgan Kauffman Publishers, 2011: 35-42.

[15] NVIDIA. OpenCL programming guide for the CUDA architecture[R]. [S.l.]: NVIDIA Corporation, 2012.