FILiC：一种CUDA上的交互型库函数框架

吴 伟 卿 鹏 漆锋滨
（江南计算技术研究所 无锡 214083）

摘 要 CUDA是NVIDIA公司推出的GPU编程模型，它为高效利用GPU计算能力提供了强大的支持。但CUDA线程无法直接访问I/O设备、网卡等外围设备，在CUDA线程与外围设备的交互功能方面，目前CUDA的支持十分有限，仅支持非实时的屏幕打印(printf)。因此提出了一种交互型库函数框架FILiC，它通过设备和主机之间的巧妙交互，高效实现了CUDA线程实时的较完整I/O等函数；并且该框架具有很好的可扩展性，CUDA程序员或者编译器开发者可基于该框架按需求开发新的CUDA线程交互功能。

关键词 CUDA，FILiC，交互型库函数，可扩展性
中图法分类号 TP311 **文献标识码** A

FILiC：A Framework for Interactive Library on CUDA

WU Wei QING Peng QI Feng-bin
(Jiangnan Institute of Computing Technology, Wuxi 214083, China)

Abstract NVIDIA developed the CUDA programming model which provides a way to accelerate more general applications by GPU. But CUDA threads cann't access peripherals directly. As far as library functions interacting with peripherals, only 'printf' is allowed in CUDA threads by now. We described a framework named FILiC for interactive library, which implements I/O functions in CUDA threads efficiently by interactions between the device and the host. And FILiC is a framework with good scalability - CUDA progammers and compiler developers can use it to design some new library functions which interact with peripherals for CUDA threads.

Keywords CUDA, FILiC, Interactive library, Scalability

1 引言

GPU因其强大的计算能力而得到越来越广泛的应用，而CUDA则为应用开发人员有效利用GPU的强大性能提供了条件。石油勘测、天文计算、流体力学模拟、分子动力学仿真、生物计算、图像处理、音视频编解码等领域的许多应用，都利用CUDA在GPU上获得了几倍、几十倍，乃至上百倍的加速比[1]。但是，CUDA在设备端对输入、输出等基本库函数支持上的不足，以及高版本的CUDA不再支持设备模拟方式，使得调试CUDA程序比较困难。如常用的屏幕打印操作，虽然CUDA 3.1[2]以上的版本已经开始支持，但是所打印的内容是在核心函数(kernel)结束后才输出到屏幕，所以并没有实现与用户的实时交互。如果kernel出现挂死的现象，应用开发人员也无法通过屏幕打印获取反馈信息。

为了更好地调试和模拟CUDA程序，对Ocelot[3]编译调试框架以及WASTE CUDA[4]模拟器都进行了有益的研究和尝试。前者提供的PTX[5]模拟器能够在CPU上针对PTX指令进行功能模拟和调试，给应用程序的开发提供了便利。WASTE模拟器则致力于在Windows环境下实现CUDA原有的设备模拟功能，并且提供了一套用于调试的库函数。但是，Ocelot和WASTE这种模拟方式的效率不高，并且不能完全反映程序在GPU上的真实执行情况，而本文所提出的FILiC框架关注的正是真实环境下应用程序的开发调试。FILiC能够让应用开发人员在设备端直接使用屏幕打印、文件输入、输出等功能，并且在程序的真实运行过程中，就可以实时地进行这些交互。与WASTE类似，这些功能同样是以库的方式提供给应用开发人员使用。

另一方面，编译器开发和设计人员同样对GPU和CUDA产生了浓厚兴趣。许多结合已有的编程模型和编译器开展的研究工作，使得非CUDA程序经过编译后能够转化成CUDA程序，或者直接生成GPU上的可执行代码，例如HMPP[6]、PyCUDA[7]、CuPP[8]等。这些工作降低了用户在GPU上编程的难度，提高了应用开发效率，也使遗产代码能快速移植到GPU上。但是，编译器可能会遇到CUDA不支持遗产代码的某些库函数的问题，例如遗产代码的核心段有文件操作，而CUDA却不支持在kernel中读写文件。FILiC可以使这类问题得到有效缓解：FILiC已经为设计人员封装了一些常用的库函数，如基本输入和输出、文件操作、存储器操作、字符串操作等；而且利用FILiC框架提供的模板和接口，设计人员可以快捷地添加需要设备端与主机端交互的库函数。

本文第2节简要介绍CUDA与本文相关的一些知识；第

到稿日期：2011-04-21 返修日期：2011-07-13
吴 伟(1984—)，男，硕士，助理工程师，主要研究领域为并行编译、并行程序设计。

3 节展示 FILiC 框架的基本组成，主要介绍 FILiC 给用户提供的底层 API 功能；第 4 节描述 FILiC 交互型功能的典型实现实例；第 5 节说明用户如何通过 FILiC 设计新的交互型库函数；第 6 节通过 CUDA SDK[9] 程序对 FILiC 框架的执行效率进行实例测试；最后是本文的总结。

2 CUDA 的相关内容

在 CUDA 的执行模型中，主机端在 kernel 启动前进行数据准备和设备初始化的工作，以及在 kernel 之间进行一些串行计算，而设备端则专注于执行高度线程化的并行处理任务。kernel 以线程网格(grid)的形式组织，每个 grid 由若干个线程块(block)组成，而每个 block 又由若干个线程(thread)组成。各 block 并行执行，block 间无法通信，也没有执行顺序[10]。图 1 给出了 CUDA 的执行模型示意。

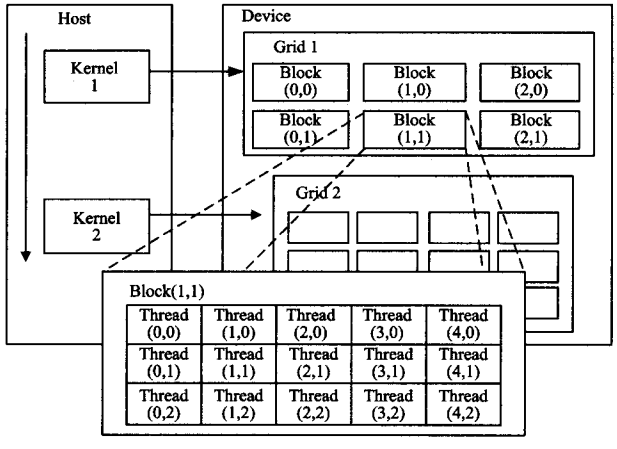

图 1 CUDA 的执行模型

一个流多处理器(Stream Multiprocessor，SM)上同时只能有一个 block 在执行，但是可以有多个活跃线程块在等待执行，当正在执行的 block 发生了高延迟操作时，活跃线程块中的一个 block 就可以被调度上 SM 执行。通过实验发现，调度器第一批次调度到各 SM 上的活动线程块是静态有序的；只要某 SM 执行完上一批的活跃线程块，调度器就调度下一批线程块到该 SM，并且与其他 SM 的状态无关。线程块的这种调度策略，决定了 FILiC 框架中一些函数的实现方式。

CUDA 提供的 mapped memory 拥有主机端和设备端两个地址，不需要在主机内存和设备显存间进行显式的数据拷贝。FILiC 利用了这种空间来实现设备端与主机端的交互型函数。首先，设备端将函数调用时的参数、数据等信息保存到 mapped memory，并通过标志通知主机端；然后主机端取出参数和数据，调用主机端相应的函数，代理设备端的请求，完成后再置标志通知设备端。将主机端和设备端之间通过置标志来交互的过程称为握手，而负责扫描并处理设备端请求的主机线程不同于原有的运行主机程序的进程，称之为交互线程。由于 mapped memory 可以从 CPU 和 GPU 两端访问，因此必须保证 CPU 和 GPU 对同一块存储器操作的顺序一致性，而 Fermi[11] 架构引入的数据 cache 增加了维护这种一致性的难度。FILiC 通过巧妙地利用设备端原子操作解决了这个问题。

3 FILiC 框架的基本组成

FILiC 框架的基本组成如图 2 所示，其主要包括主机端框架和设备端框架两部分，而图 3 则更为清晰地反映了 FILiC 框架的层次结构。

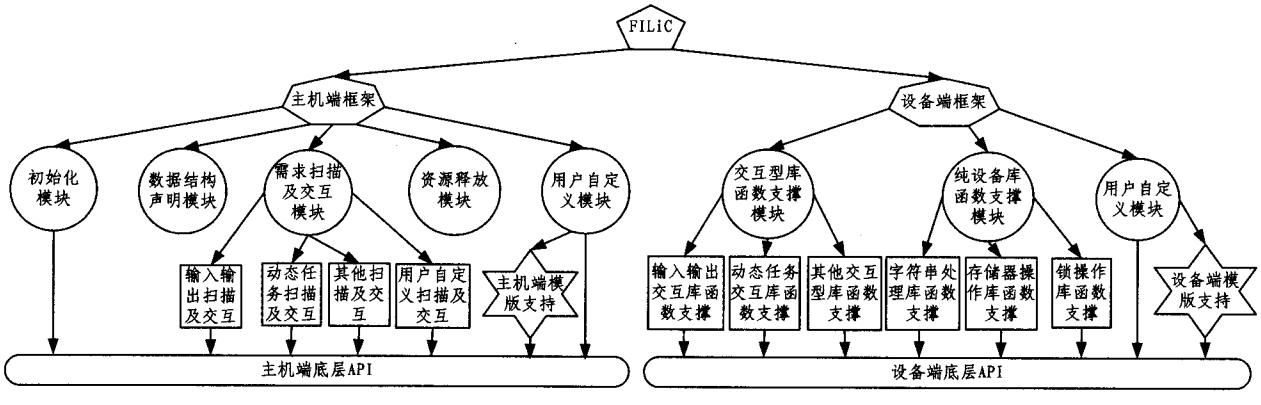

图 2 FILiC 的基本组成

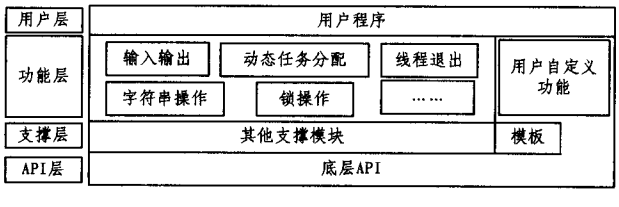

图 3 FILiC 框架的层次结构

3.1 主机端框架

在主机端框架中：初始化模块创建交互进程，并根据库函数的使用情况完成对相应变量的空间分配和初始化工作等；数据结构声明模块保留各库函数需要的数据结构的声明，其主要内容是将在 mapped memory 上使用的用于交互的数据结构；需求扫描及交互模块完成主机端对设备端的各种交互请求的实际处理工作；资源释放模块主要完成置 kernel 结束的标志、释放交互进程、各变量空间释放等工作；主机端底层 API 模块为初始化模块、需求扫描及交互模块及用户自定义模块的实现提供支持。

用户自定义的交互型库函数的主机端程序由用户自定义模块负责添加。用户可以直接使用完整的交互型模板，其中包括类似文件关闭功能的两次握手模板、类似打印功能的需要参数解析的多参数模板、类似读文件功能的多标志模板、类似动态任务调度[12] 功能的多次握手模板等。这些模板为用户提供了初始化、数据结构、需求扫描及交互等全套功能。用户在此基础上进行简单的修改就可以设计出自己的库函数主

机端程序。用户同样也可以采用更为灵活的方式，即结合底层 API 在已有的各模块添加自己的代码，完成整个主机端程序的添加。表 1 给出了常用 API 的具体功能说明。

表 1　主机端底层 API 功能

API 函数	功能
缓冲映射方式设置	FILiCAPI_buf_mapmode，用于设置缓冲队列的映射方式，宏 MAP_ONE_BY_ONE(或数字 0)表示一个线程对应一个缓冲，直接使用全局线程号访问相应缓冲；宏 MAP_SM_MAX(或数字 1)表示使用全局线程号对设备同时运行的最大线程数取模后的值访问相应缓冲。
格式参数解析	FILiCAPI_dsub_psf，用于对打印、文件打印等操作中的格式参数的解析。
错误检测和输出	FILiCAPI_error_out，判断某个条件是否成立，若不成立则输出指定的错误信息。
时间片轮转	FILiCAPI_sched，主机进程交出时间片。
mapped memory 变量的声明和初始化	FILiCAPI_map_val，在 mapped memory 上声明指定的变量，分配指定大小的空间并置 0；若是 CUDA 4.0[9] 以下的版本，则还返回此变量在设备上的指针以供设备端使用。
交互缓冲区大小设置	FILiCAPI_set_buflen，设置指定的交互缓冲队列的长度。
分配代理模式号	FILiCAPI_Hget_proxymode，为该库函数获得一个唯一的代理模式标识号。
从缓冲中取出参数值	FILiCAPI_read_parm，该 API 包含一系列的重载函数，根据参数的类型选择相应的函数执行，做出对应的类型转换后在缓冲中取出参数值。

3.2　设备端框架

设备端框架包含已实现的各库函数的设备端程序，包括需要与主机交互的交互型库函数支撑模块(包括输入输出、动态任务扫描和 exit 等)以及只在设备端运行的纯设备库函数支撑模块(包括字符串处理、存储器操作和锁操作)。同时，设备端也包含用户自定义模块，用于添加新的交互型库函数的设备端程序。同主机端框架一样，用户可以直接使用完整的库函数模板或结合丰富的设备端底层 API 完成整个设备端程序的添加。表 2 给出了常用 API 的具体功能说明。

表 2　设备端底层 API 功能

API 函数	功能
原子取数	FILiCAPI_atomic_get，保证读取指定变量值的原子性。
设备等待	FILiCAPI_wait_awhile，设备端程序等待一段时间。
握手等待	FILiCAPI_handshake，不时判断指定缓冲的设备端计数标志和主机端计数标志是否相等，等待双方握手成功。
分配代理模式号	FILiCAPI_Dget_proxymode，为该库函数获得一个唯一的代理模式标识号，该标识号会与主机端取得的标识号保持一致。
将参数值写入缓冲	FILiCAPI_write_parm，该 API 包含一系列的重载函数，根据参数的类型选择相应的函数执行，做出对应的类型转换后把用户提供的参数写入指定的缓冲中。
记录格式参数	FILiCAPI_record_fmt，用于对打印、文件打印等操作中，将需要解析的格式参数记录到交互缓冲的相应位置。

4　FILiC 交互型库函数的实现实例

在第 3 节中提到 4 种具有代表性的模板，用以支持用户添加自己的交互型库函数。下面就以它们为实例介绍 FILiC 的交互型功能的实现。

4.1　打印功能的实现

打印功能的使用最为广泛，并且其实现过程具有代表性特点，因此本节着重对打印功能进行介绍。图 4 展示了用户使用打印功能、打印请求与主机端交互、主机端处理打印请求并反馈的全过程。

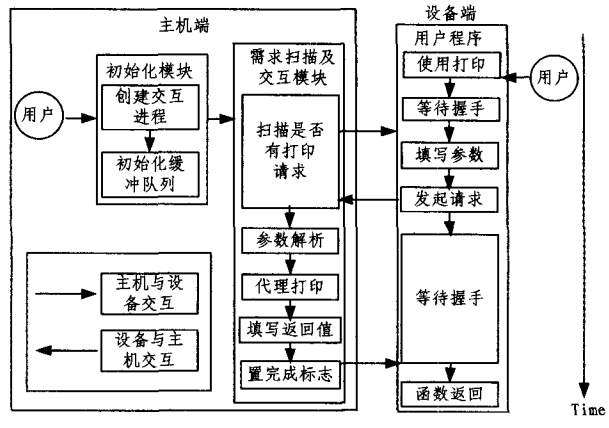

图 4　FILiC_printf 的交互处理过程

1)初始化模块创建交互进程，调用 FILiCAPI_buf_mapmode 设置映射方式。若是 MAP_ONE_BY_ONE 映射方式，则一个线程对应一个缓冲，当线程数很多时会产生很大的空间开销。若是 MAP_SM_MAX 方式，则当线程总数大于设备同时运行的最大线程数时，缓冲队列长度取为设备同时运行的最大线程数，这样可以大幅减少空间需求，不过用户需要保证各 SM 的工作量相当。因为当某个 SM 出现长延迟的操作而远远落后其他 SM 的执行时，其他 SM 并不会等待该 SM 完成，而是去动态地调度下一批 block 来执行。这些 block 中的 thread 对同时运行的最大线程数取模后，可能会映射到长延迟 SM 中 thread 对应的同一个缓冲上，造成打印信息不完整。实际上，cuda 的打印也无法处理好缓冲个数的问题，它通过一个接口指定缓冲容量，如果实际打印内容超过缓冲的容纳能力，则无法保证打印的正确性。本框架实际上提供了一个可供用户选择的对空间有较好优化的方式，而且绝大部分程序不会出现工作量很不均匀的情况，它还能比较好地保证打印的正确性。初始化模块接着调用 FILiCAPI_map_val 进行交互缓冲队列的初始化工作。

2)交互进程调用需求扫描及交互处理模块的扫描函数，开始扫描是否存在打印的请求，为了防止交互进程长时间占据 CPU，调用 FILiCAPI_sched 进行时间片轮转。

3)用户准备在设备端使用打印功能，由于没有 CUDA 编译器的支持，设备端函数无法实现可变参数，因此原有的一条打印语句可能需要多条语句实现。其分别是调用 FILiC_printf_begin 函数用以传递格式参数，多条 FILiCAPI_write_parm 语句用于交互需要打印的参数的值，最后是调用一条 FILiC_printf_end 来发出握手请求并等待交互完成、返回结果。

例如，printf("data %d%f\n",a,b)；将需要写成如下语句：

```
FILiC_printf_begin("data %d%f\n");
FILiCAPI_write_parm(a);
FILiCAPI_write_parm(b);
FILiC_printf_end();
```

如果是在编译器的设计和开发中，用户则可以使用正常的 printf 格式，编译器在产生 CUDA 代码时会为用户生成相应的语句。实际上，在针对 CUDA 的编译器开发中，已经证明了编译器的支持可以很方便地解决变参的问题。打印的变参处理的方式同样具有普适性，可以在文件操作等其他地方

使用。

在 FILiC_printf_begin 中，首先调用 FILiCAPI_handshake 等待握手成功，即等待同一缓冲上的之前的请求已经完成，可以处理当前的请求。接着给缓冲结构的线程号及代理标识号赋值，并调用 FILiCAPI_record_fmt 记录格式参数。在 FILiC_printf_end 中，因为参数均已传递完成，所以将缓冲结构的线程标志值加 1，发出握手请求，并调用 FILiCAPI_handshake 等待交互完成。

4)需求扫描函数检测到有代理请求，则根据请求类型转入对 printf 处理的主机端程序。该程序调用 FILiCAPI_dsub_psf 对格式参数进行解析，根据每次解析出的打印类型将缓冲结构中相应位置的数据经过类型转换后取出，调用主机端的 printf 函数并将返回结果累加。将格式参数解析完成后，主机端标志值加 1，标识代理已经完成。但是在 Fermi 架构上，由于要考虑数据 Cache 的一致性问题，需要再引入一个 host_num_only 标志，该标志只在主机端使用，代理结束时也加 1。在确定代理开始时需要判断线程标志值是否比 host_num_only 大 1，以防止因 Cache 一致性问题而导致重复打印。

5)FILiC_printf_end 函数发现交互完成后，通过缓冲结构得到最终的返回值，并返回结果。

在这个交互过程中，许多工作已由 FILiC 框架完成，应用开发和调试人员若想使用 FILiC 的打印功能，只需在主机端启动交互进程，然后在设备端调用打印函数(包括参数设置和打印结束函数)就可以了。

4.2 文件相关功能的实现

文件关闭功能的实现反映了一个最基本的两次握手的过程，不涉及变参和格式参数解析的处理，只涉及设备端等待握手成功、给缓冲结构相应变量赋值、发出请求、握手等待请求完成的过程。需要注意的特别之处在于，文件指针在设备端只被当作一个 long 型的地址来看待。因为该功能简单清晰，所以以此为基础的模板具有最好的适用性，也有利于用户理解 FILiC 的交互方式。

读文件功能从大框架上看也是一个两次握手的过程，其区别在于当主机端受理了设备端的请求之后，会经历若干次的“小”握手阶段。这是因为读取的数据量不确定，而存储数据的缓冲又不能无限制扩大，因此只能进行分批读取的操作。利用缓冲结构中空闲的参数缓冲充当新的交互标志来控制分批读取的交互过程。

4.3 动态任务调度功能的实现

动态任务调度功能能在 CUDA 本身 block 动态调度基础上提供一个 GPU 间动态调度的补充，这更有利于实现负载的均衡；并且动态任务调度的实现也具有一定的代表性，它是一个涉及到 GPU 间交互的多次握手过程。设备端发出申请任务号的请求，主机端从本地缓冲区返回任务号，如果任务为空则向主进程(比如 0 号 GPU)申请一批任务。设备端等待握手成功后，取出任务号再对线程标志值进行一次加 1 操作，通知主机端任务已取走，可以分配下一个任务。

5 基于 FILiC 设计新的交互型库函数

本节通过一个具体实例对利用 FILiC 设计新的交互型库函数进行简要的说明。C 标准库中的 isdigit 函数用于判断一个字符是否是数字，CUDA 本身不支持该函数。我们可以根据字符的 alpha 值判断其是否是数字，从而在纯设备库函数支撑模块中添加该函数。不过，为了展示设计交互型库函数的过程，或者用户想直接使用 C 标准库的效果，则可以通过交互的方式调用主机端的函数来实现。

isdigit 函数只需要传递一个字符参数给主机端，而且它符合基本的两次握手框架。因此，在主机端直接选择类似文件关闭功能的两次握手模板，该模板直接重用已有的输入、输出交互缓冲队列，调用 FILiCAPI_Hget_proxymode 来为 isdigit 取得标识号，并用一个宏 NEW_PROXY_MODE 来表示这个号。用户可以把该宏在程序里全部替换成 PROXY_MODE_ISDIGIT，实现更好的可读性。需求扫描及交互模块中的扫描及交互函数也已经生成，在处理请求的程序中已经通过 FILiCAPI_read_parm 取出了参数值，用户只需要把原有的调用 new_proxy_func 改成实际的 isdigit 即可。

在设备端同样使用类似文件关闭功能的两次握手模板，用户可以把生成的设备端函数名替换成 FILiC_isdigit，并把参数类型改成 char。设备端函数依次进行了 FILiCAPI_handshake、给线程号赋值、FILiCAPI_Dget_proxymode、FILiCAPI_write_parm、线程标志值自增和 FILiCAPI_handshake 等操作，这些内容用户都不需要进行任何修改。因为 isdigit 的返回类型是 bool，获取返回值时需要加一个类型转换。至此，交互型库函数 isdigit 的设计工作就已完成，用户在程序中可以直接使用 FILiC_isdigit。用户也可以不使用模板，直接在 FILiC 框架原有代码上使用底层 API 来完成同样的目标。

6 实例测试

本节以屏幕打印为例，通过实验来测试 FILiC 框架的执行效率。实验使用的 CPU 为 Intel Core2 Duo E7500，GPU 为 NVIDIA Tesla C2050，操作系统为 Red Hat Enterprise Linux Server release 5.3，CUDA 编译器为 nvcc 4.0。

测试的方法是在 CUDA SDK 4.0 程序中加入 FILiC 的屏幕打印函数(每个线程打印其线程号和一个字符串)，测试添加打印前后程序运行的时间，评估交互对原有程序执行的影响情况；同时将 FILiC 的屏幕打印性能与 CUDA 本身的屏幕打印相比较。实验结果如表 3 所列。

表 3 FILiC 框架针对 CUDA SDK 程序的测试结果

SDK 程序	执行时间(单位：ms)		影响比例	执行时间(单位：ms)		加速比
	原始时间	FILiC 屏幕打印		CUDA 屏幕打印	FILiC 屏幕打印	
mergeSort	693	771	11.3%	805	771	1.04
vectorAdd	387	457	18.1%	650	457	1.42
dct8x8	603	675	11.9%	687	675	1.02
scalarProd	428	478	11.7%	573	478	1.20
FDTD3d	8625	9048	4.9%	9196	9048	1.02
平均值	/	/	11.6%	/	/	1.14

从表 3 中可以看出，FILiC 屏幕打印的执行效率很高，使用 FILiC 屏幕打印增加的程序执行时间占原执行时间的比例平均为 11.6%，而与 CUDA 屏幕打印相比，平均能达到 1.14 的加速比。而对于一些大型应用，CUDA 的屏幕打印需要经过几小时甚至更长的时间才能得到结果，而 FILiC 屏幕打印却能实时地对用户的打印需求进行反馈，为应用开发调试提供了极大的便利。

结束语 本文在 CUDA 上设计了一种交互型的库函数

(下转第 138 页)

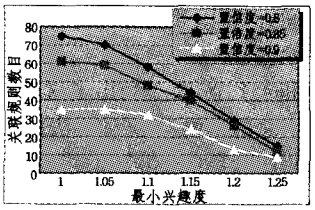

图 5　最小兴趣度与关联规则数目的关系

图 3 描述了最小支持度对关联规则产生数目的影响。在相同的置信度水平下，随着最小支持度的增大，发现规则的数目不断减少；当最小置信度处于一个较低的水平时，曲线的变化比较陡，说明此时最小支持度对最终产生的关联规则数目影响较大；而当最小置信度处于一个较高的水平时，曲线的变化比较平缓，说明此时最小支持度对产生规则的数目影响较小。随着最小支持度的取值趋向 1，产生的规则数目趋向 0。

图 4 描述了最小置信度对关联规则产生数目的影响。在相同的最小支持度水平下，随着最小置信度的增大，所得到的关联规则数目不断减少；当最小支持度设置较小时，曲线变化比较陡，此时最小置信度对最终产生的关联规则数目具有较大的影响；而当最小支持度设置较大时，曲线变化比较平缓，此时最小置信度对产生的关联规则数目影响较小。随着最小置信度的取值趋近 1，规则的数目趋向 0。

图 5 描述了最小兴趣度与产生的关联规则数目之间的关系（最小支持度为 0.55）。从图中可知，在相同的置信度水平下，随着兴趣度的增大，所产生的规则数目不断减少。

5.2 节的实验得出的各参数设置对挖掘结果的影响效果均符合预期。结合 5.1 节的实验，可以得出本文所提出的算法是有效的。

结束语　本文借鉴线性链表的存储结构的优点，提出了基于线性链表的模糊关联规则挖掘算法。算法能够充分利用前期计算的结果有效地减少存储和计算的花销，具有较高的效率；并且通过引入兴趣度剔除那些无意义或者意义不大的规则，很好地控制了产生规则的有效性和准确性。比较分析和实验表明，算法快速而有效。

然而，由于时间原因及条件限制，本文仅在模糊关联规则挖掘算法研究上做了一些工作。下一步，将在隶属函数的确定与优化方面做深入的研究，因为隶属函数选择的好坏对挖掘结果具有非常大的影响。

参 考 文 献

[1] Chan K C C, Au W-H. Mining fuzzy association rules [C]// Proceedings of the 6th ACM International Conference on Information and Knowledge Management. Las Vegas. Nevada, USA, 1997: 209-215

[2] Hong T P, Kuo C S, Chi S C. Mining association rules from quantitative data[J]. Intelligent Data Analysis, 1999, 3: 363-376

[3] Hong T P, Kuo C S, Wang S L. A fuzzy AprioriTid mining algorithm with reduced computational time[J]. Applied Soft Computing, 2004, 5: 1-10

[4] Lee Y C, Hong T P, Lin W Y. Mining fuzzy association rules with multiple minimum supports using maximum constraints [J]. Lecture Notes in Computer Science, 2004, 3214: 1283-1290

[5] Papadimitriou S, Mavroudi S. The frequent fuzzy pattern tree [C]// Proceeding of the 9th WSEAS International Conference on Computers. 2005

[6] Lin C W, Hong T P, Lu W H. Linguistic data mining with fuzzy FP-trees[J]. Expert Systems with Applications, 2010, 37: 4560-4567

[7] http://archive.ics.uci.edu/ml/machine-learning-databases/glass/

[8] 陆建江，张亚非，宋自林. 模糊关联规则的研究与应用[M]. 北京：科学出版社，2008

（上接第 127 页）

框架 FILiC。FILiC 能够在占用较少的 GPU 计算资源的情况下，实现 CUDA 设备端的打印、文件操作等交互函数。另一方面，FILiC 提供了一系列交互模板和底层 API，以支撑用户根据自身需求开发新的交互型库函数。FILiC 框架在交互过程中能够正确有效地完成用户的代理请求，而且执行效率很高，与 CUDA 打印相比平均有 1.14 的加速比，而且能够实时地对用户的打印给出反馈。

我们认为，FILiC 库函数框架的意义主要有两点：一是对于 CUDA 应用开发人员而言，本框架不仅提供了新的交互型库函数功能，而且提供了新的实时调试手段。二是编译器开发者如果需要将用户程序转换为 CUDA 代码，那么可以利用 FILiC 框架，来实现原用户程序中的 I/O 功能、消息发送接收、socket 操作甚至一个完整的 Web 服务器。可以说，FILiC 是对 CUDA 功能拓展的一次有益探索。

参 考 文 献

[1] 张舒，褚艳丽，赵开勇，等. GPU 高性能运算之 CUDA[M]. 北京：中国水利水电出版社，2009

[2] CUDA 3.1 Downloads. developer. nvidia. com/object/cuda_3_1_downloads. html

[3] Ocelot: An Open Source Debugging and Compilation Framework for CUDA[OL]. http://code.google.com/p/gpuocelot/

[4] cuda-waste: Why Another Simple Trivial Emulator for CUDA [OL]. http://code.google.com/p/cuda-waste/

[5] NVIDIA Compute——PTX: Parallel Thread Execution ISA Version 1.1[OL]. http://www.nvidia.com/object/io_1195170102263.html

[6] OpenHMPP[OL]. http://en.wikipedia.org/wiki/HMPP_Open_Standard

[7] PyCUDA[OL]. http://mathema.tician.de/software/pycuda

[8] Breitbart J. Cupp-a framework for easy cuda integration [C]// IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. Washington, DC, USA, IEEE Computer Society, 2009: 1-8

[9] CUDA 4.0 Downloads. developer. nvidia. com/object/cuda_4_0_downloads. html

[10] Kirk D B, Hwu W-M W. Programming Massively Parallel Processors: A Hands-on Approach[M]. ELSEVIER Press, 2010

[11] Next Generation CUDA Architecture [OL]. www.nvidia.com/object/fermi_architecture.html

[12] 尤洪涛，姜小成，陈左宁. 基于动态任务划分的降级机制[J]. 微计算机信息，2006，10(3)：72-76