基于OpenCL的连续数据无关访存密集型函数并行与优化研究

蒋丽媛[1,3] 张云泉[1,2] 龙国平[1] 贾海鹏[1,4]

(中国科学院软件研究所并行软件与计算科学实验室 北京 100190)[1]

(中国科学院软件研究所计算机科学国家重点实验室 北京 100190)[2]

(中国科学院研究生院 北京 100190)[3] (中国海洋大学信息科学与工程学院 青岛 266100)[4]

摘 要 连续的数据无关是指计算目标矩阵连续的元素时使用的源矩阵元素之间没有关系且也为连续的,访存密集型是指函数的计算量较小,但是有大量的数据传输操作。在OpenCL框架下,以bitwise函数为例,研究和实现了连续数据无关访存密集型函数在GPU平台上的并行与优化。在考察向量化、线程组织方式和指令选择优化等多个优化角度在不同的GPU硬件平台上对性能的影响之后,实现了这个函数的跨平台性能移植。实验结果表明,在不考虑数据传输的前提下,优化后的函数与这个函数在OpenCV库中的CPU版本相比,在AMD HD 5850 GPU达到了平均40倍的性能加速比;在AMD HD 7970 GPU达到了平均90倍的性能加速比;在NVIDIA Tesla C2050 GPU上达到了平均60倍的性能加速比;同时,与这个函数在OpenCV库中的CUDA实现相比,在NVIDIA Tesla C2050平台上也达到了1.5倍的性能加速。

关键词 GPU,OpenCL,向量化,ROI

中图法分类号 TP302 **文献标识码** A

Parallelism and Research on Functions with Continuously Independent Data and Intensive Memory Access Using OpenCL

JIANG Li-yuan[1,3] ZHANG Yun-quan[1,2] LONG Guo-ping[1] JIA Hai-peng[1,4]

(Laboratory of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[1]

(State Key Laboratory of Computing Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[2]

(Graduate University, Chinese Academy of Sciences, Beijing 100190, China)[3]

(School of Information Science and Technology, The Ocean University of China, Qingdao 266100, China)[4]

Abstract *Continuously independent data type means when calculating the continuous elements of destination matrix,* the used elements of source matrices are also continuous and there are no relationship among them. Intensive memory access function is the function that has less computation but a lot of data transfer operations. This paper took the bitwise function as the example, studied and implemented the parallel and the optimizing methods of the continuously independent data and intensive memory access function on GPU platforms. Based on the OpenCL framework, this paper studied and compared various optimizing methods, such as vectorizing, threads organizing, and instruction selecting, and finally used these methods to implement the cross-platform transfer of the bitwise function among different platforms. The study tested the function's execution time without data transfer both on AMD GPU and NVIDIA GPU platforms. On the AMD Radeon HD 5850 platform, the performance has reached 40 times faster than the CPU version in OpenCV library, 90 times faster on AMD Radeon HD 7970 platform, and 60 times faster on NVIDIA GPU Tesla C2050 platform. On NVIDIA GPU Tesla C2050 platform, the speedup is 1.5 comparing with the CUDA version in OpenCV library.

Keywords GPU, OpenCL, Vectorization, ROI

1 引言

随着计算能力和可编程性的不断提高,GPU在通用计算领域的应用也越来越广泛。OpenCL作为第一个面向异构系统的通用目的的并行编程模型,以其跨平台特性得到了越来越多厂商的支持。然而,由于GPU硬件架构的多样性和复杂性,使得在不同硬件平台上的性能移植成为难点。因此,本文以连续数据无关的访存密集型函数在不同GPU平台上的

到稿日期:2012-10-19 返修日期:2012-12-27 本文受国家自然科学基金资助项目(60303020,60533020),国家自然科学基金资助重点项目(60503020),国家自然科学基金青年基金课题(61100072),国家"863"计划基金资助项目(2012AA010902)和ISCAS-AMD联合fusion软件中心资助。

蒋丽媛(1987-),女,硕士生,CCF会员,主要研究方向为并行计算,E-mail:jlyuan001.good@163.com;张云泉(1973-),男,研究员,博士生导师,CCF会员,主要研究方向为高性能并行计算及并行数值软件;龙国平(1982-),男,博士,主要研究方向为计算机体系结构等。

并行和优化研究为例，研究访存密集型函数在不同 GPU 平台上的性能可移植性。

在 GPU 优化和性能移植方面，已经有很多的相关工作。文献[5]以 blur 函数为例，介绍了图像模糊化算法在 AMD 和 NVIDIA GPU 平台上的实现和优化，详细介绍了这个函数在 OpenCL 架构下存储访问的优化方法和选择策略，尤其是对 buffer 和 image 对象的选择方面，对本文很有启发和帮助；文献[11]以 minMax 函数为例详细讲述了归约算法在 AMD 和 NVIDIA GPU 平台上的实现和优化，文中分层次介绍了分别使用向量化、存储访问优化、线程组织方式和指令流优化后函数的加速情况，具有很高的参考价值；文献[10]以图像增强算法——拉普拉斯算法为例，详细介绍了函数的跨平台实现和优化技术，针对 AMD 和 NVIDIA GPU 的硬件架构的异同，总结了在 OpenCL 架构下相关的实现和优化方法。然而他们都没有针对连续数据无关的访存密集型函数进行研究。

本文的主要工作和贡献是研究了连续数据无关访存密集型函数在 OpenCL 架构下的优化方法。OpenCV 库有很多这种类型的函数，所在项目组已经改写的 bitiwse 系列的函数、比较函数、乘方函数和矩阵加减乘除 4 个算术运算函数等都属于连续数据无关访存密集型函数。这类函数在计算目标矩阵连续的元素时，使用的源矩阵元素之间没有关系且也为连续的，函数的计算量较小，但是有大量的数据传输操作，所以它们的优化方式非常相似，都可以采用 OpenCL 数据传输和存储方面的优化方法，所以研究这类函数的优化具有一定的代表性。本文以 bitwise 函数为例，通过在 AMD 和 NVIDIA GPU 平台上加速 bitwise 函数，研究了此类函数的平台移植和性能移植性。bitwise 函数是 OpenCV 库中的位操作函数，它的功能是对两个矩阵的相对应元素进行逻辑位运算，并将计算结果存储在相对应的结果矩阵中。逻辑位运算包括与、或、非、异或 4 种操作，这 4 种操作在 GPU 上的实现和优化大体相同。

本文第 2 节介绍 OpenCL 的编程框架和 GPU 架构；第 3 节介绍对于连续访存无关访存密集型函数在两个平台上的通用优化方法；第 4 节是在 OpenCL 框架下针对 bitwise 函数进行的优化；第 5 节分别在 AMD GPU 和 NVIDIA GPU 平台上进行测试。

2 OpenCL 与 GPU 架构

2.1 OpenCL 的编程框架

OpenCL 是一个异构平台并行计算的标准，此异构平台可以包括 CPU、GPU 和其他类计算设备。OpenCL 编程框架主要分为 4 个部分，即平台模型、内存模型、执行模型和编程模型[1]。

OpenCL 的平台模型是由一个主机连接一个或多个计算设备构成的。每个计算设备又可以分割成一个或多个计算单元(Compute Unit，CU)，而每个计算单元又可以由一个或多个处理单元(Processing Element，PE)组成。OpenCL 的存储模型分为主机和 OpenCL 设备两部分，OpenCL 设备上的内存分为 4 种，即全局内存(global memory)、常量内存(constant memory)、局部内存(local memory)和私有内存(private memory)。在读写速度上，私有内存＞常量内存＞局部内存＞全局内存。OpenCL 的执行模型也分为两部分，一部分是在 host 端的主程序，另一部分是在 OpenCL 设备上执行的内核程序(kernel)。OpenCL 编程模型支持数据级并行和任务级并行，同时也支持这 2 种方式的混合，OpenCL 的设计重点在于数据并行(任务并行主要针对多核 CPU)。

2.2 GPU 架构

目前，通用计算 GPU 架构主要包括两种：AMD 基于 SIMD(Single Instruction Multiple Data，单指令多数据)的向量架构；NVIDIA 基于 SIMT(Single Instruction Multiple Thread，单指令多线程)的标量架构。下面介绍 AMD GPU 的架构。

以 AMD Radeon™ HD 5850(Cypress 核心)为例，其包含 18 个 SIMD 引擎，每个 SIMD 引擎包含 16 个 SIMD 流处理核心，而每个 SIMD 处理核心又包含 5 个 ALU。针对 SIMD 的架构，AMD 采用 VLIW(Very Long Instruction Word，超长指令字)技术将短指令集成为长的 VLIW 指令来提高资源利用率，例如 5 条 1D 标量指令可以被集成为 1 条 VLIW 指令在一个周期完成。结果写回全局存储时，如果是 32 位整数倍的数据，则通过 Fast Path 写回，而小于 32 位的数据或者原子操作，则通过 Complete Path 写回。Fast Path 的速度比 Complete Path 的速度快 7～10 倍[2]。

NVIDIA GPU 是 SIMT 架构，从 G80 核心开始，NVIDIA 的 GPU 采用了统一的运算单元，并开始走彻底标量化路线。在 G80 GPU 内部，NVIDIA 将 ALU 拆分为最基本的 1D 标量运算单元，实现了 128 个标量流处理器，所有的运算指令被拆分为 1D 标量指令进行运算。

Fermi 是 NVIDIA 最新的 GPU 产品线，主要包括 GTX4xx 系列和 Tesla C2050。以 Tesla C2050 为例，它有 14 个 SM(Streaming Multiprocessor)，每个 SM 中有 32 个 SP(Streaming Processor，CUDA core)。

Fermi 架构每个 SM 中含有 128kB 大小的寄存器堆(32768 个 32-bit 的寄存器)；共享存储(shared memory)和 L1 Cache 共有 64kB，根据需要编程人员可以将两者配置为各占 48kB/16kB 或者 16kB/48kB；L2 Cache 大小为 768kB[3]。

3 连续访存无关访存密集型函数在两个平台上的通用优化方法

这类函数的特点是计算量小、数据传输较多，瓶颈在访存上，所以可以采用 OpenCL 数据传输和存储方面的优化方法，比如为了充分利用显存带宽，可以选择使用 128 位的向量来存取数据，因为 128 位数据正好可以充分利用访存请求的宽度[4]；函数的另一个特点是连续的数据无关性，即计算目标矩阵连续的元素时使用的源矩阵元素之间没有关系且也是连续的，所以可以同时处理尽可能多的元素，提高计算效率，可以采用向量化和线程组织方面的优化方法。另外还可以尝试 OpenCL 中的指令流优化和其他的优化方法。

3.1 内存对象的分析

选择恰当的存储方式可以提高访存效率，OpenCL 使用内存对象在主机和设备之间传输数据，OpenCL 中的内存对象包括 buffer 和 image object(图像对象)。buffer 是一维数据元素的集合。图像对象(image 对象)主要用来存储一维、二维、三维图像、纹理，图像对象中包括对地址模式和滤波模式的处理，并且是缓存的，该存储方式一次可以读取一个长度

为 4 的向量.

本文不使用 image 的原因是,使用 image 作为基本存储方式有两个缺点,1)image 对象即使在计算 uchar(unsigned char)类型时,也是将 uchar 转换成一个 int 来进行计算,非常浪费带宽;2)image 对象只能使用长度为 4 的向量。在带宽利用率不是很高的情况下,image 对象作为基本存储方式的缺点没有体现出来,但是在继续优化之后,就会发现 buffer 对象作为基本存储方式比 image 对象作为基本存储方式的速度更快。当图像为 4 通道、片上需要缓存的数据量小时,images 的优势才得以充分发挥,代码简洁且高效;而当图像不只是 4 通道时,使用 buffer 的方式通过合理安排数据的向量读取和向量化计算,能够更好地发挥 AMD GPU 硬件架构的性能[5]。所以选择使用 buffer 作为内存对象。

3.2 向量化的分析

在大部分 OpenCL 设备上,相邻线程读取连续数据时,会自动地对读取操作进行合并,因为 bitwise 函数是连续的数据无关,所以可以让相邻线程读取连续数据。比如读取 uchar4 时,如果需要读取的数据连续,则一次就可以将 4 个字节全部读出。

从全局存储连续读取数据时,要注意全局存储几个方面的优化包括:

1)合理组织数据的存储和线程访问数据的顺序,尽量避免存储体冲突。

2)尽量使用向量存取数据,AMD 和 NVIDIA GPU 最好分别使用 float4 和 float2 类型[6,7]。

3)尽量使用对齐的数据访问[2,8]。

4)合理组织线程访问数据的顺序,尽量使用合并读(Coalescing Read)和合并写(Coalescing Write)[9]。

5)在 AMD GPU 中写回时尽量使用 Fast Path[6]。

由于 bitwise 函数的存储访问模式是连续访存模式,因此,本文提高 Global Memory 的访存效率的最主要方法是向量化读取。

在实际算法优化中,采用向量化计算方法在两个平台上都能显著提高函数性能,这是因为在 AMD GPU 平台下,向量化可以充分利用 5 路 VLIW 计算资源。

由于 bitwise 函数的特点,需要向显存进行大量的写操作,使用向量化来提高计算部件的使用效率非常值得尝试。本文函数中使用了长度为 2,4,8,16 这 4 种向量化方式。

AMD 架构中结果写回全局存储时,如果是 32 位整数倍的数据,则通过 Fast Path 写回;而如果是小于 32 位的数据或者原子操作,则通过 Complete Path 写回。Fast Path 的速度比 Complete Path 的速度快 7～10 倍[2]。为了使用 Fast Path,不得不在程序中手工处理对齐的问题。

3.3 线程组织方式的分析

线程组织方面的优化要注意调整线程组织结构,使 work-group 的大小为 wavefront(AMD 平台)或者 warp(NVIDIA 平台)的整数倍;每个 CU 上同时并发运行足够数目的 wave-front 或者 warp,以隐藏访存延迟[10]。

在 AMD GPU 中,线程是以 wavefront 为单位进行执行的,在 HD 5850 中,每个 wavefront 中有 64 个工作项在并行执行。运行时可调度多个 work group 到一个 CU,因此 work group 这一层的调度能挖掘不同 CU 间的并行性。运行时硬件自动将 work group 切分为 1 个或多个 wavefront,以 wavefront 为单位将计算任务发射到一个 CU 内的 16 个核上执行。一个 CU 内可以同时保持多个 wavefront 的状态,任何一个 wavefront 因访存而堵塞时,能快速切换到下一个 wavefront 继续执行,从而可以很好地隐藏访存延迟。

在 AMD GPU 5850 上,只有 18 个 CU,由于 OpenCL 的一个工作组只能在一个 CU 上运行,因此在真实运行的过程中,同时运行的只有 18 个工作组。所以可以考虑将工作组数量固定为 18,工作组大小为 256。在计算过程中,通过对数据进行折叠,将数据规模降低到 18×256 个线程一次可以运行完成的大小[11]。

3.4 指令流优化和其他的优化方法分析

在精度允许的情况下,尽量使用内建本地函数代替标准函数;对于复杂的计算可以考虑用简单计算代替。比如对于乘 2^n 的操作,可以用左移 n 位来代替。如果可以,尽量用 24 位的乘和除操作。

4 在 OpenCL 框架下针对 bitwise 函数进行的优化

因为 bitwise 函数的与、或、非、异或 4 种操作在 GPU 上的实现和优化大体相同,所以本文将以与操作 bitwise_and 为例,介绍不同数据类型和通道数的 bitwise 函数在不同 GPU 平台上的优化方法。函数中的数据类型是 4 种通道(分别表示为 C1、C2、C3 和 C4)的 uchar(unsigned char),char,ushort(unsigned short),short,int,float,double 类型。

OpenCL 的执行模型分两部分,即 CPU 端的主程序和 GPU 端的执行程序 kernel。CPU 端的程序功能主要有设置 kernel 执行的上下文;设置工作空间 NDRange;传递参数值;执行 kernel 等。GPU 端是 kernel 程序的并行和优化。下面介绍对 bitwise 函数的实现和优化。

kernel 文件中使用到向量化处理,若使用 char4 作为向量长度,则有 $x=x<<2$ 语句,它使 x 方向的索引变为原来的 4 倍,即代表 4 个 char 数据一起处理。这条语句使用了左移语句代替乘法指令,因为左移、右移类语句的执行速度快于乘除指令,所以当可以用左移、右移等语句代替乘除等功能相同的指令时,要替换。

bitwise 函数支持 ROI(Region Of Interest)。ROI 的含义是指在一整幅图像中,只对其中的某一部分感兴趣,则只对感兴趣的部分进行处理。对 ROI 的支持给函数优化带来了很大的麻烦,这主要是因为在 GPU 中数据的访问必须是对齐的,而使用向量访问 ROI 中的数据时大部分情况下都是非对齐的。非对齐的数据访问从正确性和性能两个方面给函数优化工作带来了挑战。

AMD GPU 中当数据非对齐时,kernel 中使用 vload 实现数据的非对齐读。但是使用 vstore 写回数据时通过的是 Complete Path 而不是 Fast Path,这严重影响了性能。这是因为使用 vstore 写回数据时,即使数据大小大于 32 位,仍然通过 Complete Path 写回。为了使用 Fast Path,不得不在程序中手工处理对齐的问题。源矩阵的偏移量是根据目标矩阵的偏移量而决定的,如图 1 所示。如在处理 char4 类型的数据中语句 #define dst_align(dst_offset & 3)的使用,dst_off 是存在 ROI 时矩阵元素的偏移量,&3 的目的是调整偏移量为 4 字节的倍数。在建立源矩阵和目标矩阵的索引时,要前移

dst_align 个字节，以达到对齐访问的目的。

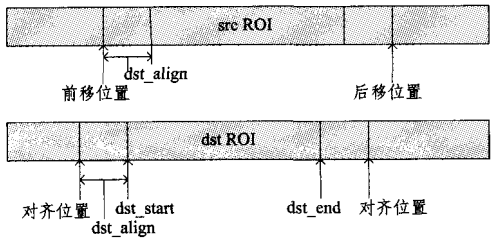

图 1　使用 ROI 时目标矩阵(dst)和源矩阵(src)的偏移

函数中对元素值的越界处理使用 dst_start 和 dst_end 来完成，只有写入的元素处于允许范围之内才可以写入。越界处理中使用了大量条件表达式?:语句代替 if、else 判断语句。

bitwise 函数支持 4 个通道的 uchar，char，ushort，short，int，float，double 7 种数据类型的计算，在 CPU 端要设定好每个通道每种数据类型使用的向量化长度。bitwise 函数含有矩阵和 scalar 的计算，在 CPU 端给 kernel 程序传递参数时，要注意当与 scalar 做逻辑操作时，为了提高传参效率，对于小的数据类型不使用创建 buffer 传递参数的方式，而是直接在 CPU 端将 scalar 类型以向量长度为 4 的参数的方式传递给 kernel。bitwise 的逻辑操作不支持浮点类型，所以 kernel 中表示 float 和 double 时使用的是 char 和 short 类型。传递 float 类型的 scalar 向量参数时容易犯错，比如当使用 char4 表示 float 类型时，需要传递的 scalar 为 float4。因为 float4 应该用 char16 表示，所以正确传递的 scalar 应该是 char16 而不是 char4。

本文的 bitwise 函数由于多通道和多种数据类型，使用了多种向量化方式。下面对 CPU 端和 kernel 中向量化长度的对应关系进行分析(以一个矩阵和一个 scalar 做运算的单通道函数为例)，表 1 是本文中在 CPU 端对单通道的各个数据类型的向量长度的设定(数字代表一次处理元素的个数)。

表 1　CPU 端向量长度的设定

通道数	数据类型						
	uchar	char	ushort	short	int	float	double
C1	4	4	2	2	1	1	1

对应的 kernel 文件中的向量长度设定如表 2 所列。

表 2　GPU 端向量长度的设定

通道数	数据类型						
	uchar	char	ushort	short	int	Float	double
C1	uchar4	char4	ushort2	short2	int	char4	short4

AMD GPU 的一个 PE 中有 4 个处理单元和 1 个 T 处理单元，利用 AMD 向量架构的超长指令字技术 VLIW，可以充分利用向量化提高程序的执行速度。在 bitwise 函数中，基本采用 uchar4，short2，int 的方式进行元素的向量运算。

当数据类型为单通道时，在 CPU 端设定 4 个 uchar 或者 char 数据，2 个 ushort(unsigned short)或者 short 数据，1 个 int、float 或者 double 数据一起处理。kernel 端使用的向量分别为 uchar4，char4，ushort2，short2，int，float，double。因为 bitwise 操作不支持浮点类型，所以 kernel 中分别使用 char4 和 short4 来表示 float 和 double。使用 short 而不是 char 表示 double 的原因是，当传递 scalar 参数时，double4 等于 char32，而向量长度最大为 16，所以不可以用 char 来表示 double。而 double4 表示成 short16 就没有问题。故可以使用 char 表示 float，而一定要使用 short 表示 double。综上，单通道时的数据处理方式设置如上。

5　分别在 AMD GPU 和 NVIDIA GPU 平台上进行测试及性能评估

在测试程序性能时，用 OpenCL 实现的函数与 OpenCV 库中的函数功能和接口均一致。OpenCV 库[12]中包含了 CPU 版本的函数实现和 CUDA 版本的实现。本文主要针对 AMD Radeon™ HD 5850 架构进行函数实现和优化。为了测试函数性能，将 OpenCL 版的函数分别与 CPU 版和 CUDA 版的函数进行对比。在此需要说明的是，OpenCV 库中 CPU 版本和 CUDA 版本的代码均是已经经过优化的高性能版本，其中 CUDA 版本的实现调用了 NVIDIA 开发的高性能 NPP 库[13]。

5.1　测试平台相关信息

设备参数如表 3 所列。

表 3　测试平台的设备参数

GPU	Cores	Peak single precision floating point performance	Peak double precision floating point performance	Memory bandwidth
HD 5850	288	2.09Tflops	0.418Tflops	128GB/sec
Tesla C2050	448	1.03Tflops	0.515Tflops	144GB/sec
HD 7970	128 * 16	3.79Tfops	0.947Tflops	264 GB/S

程序运行环境如表 4 所列。

表 4　测试程序的运行环境

Code version	CPU	GPU	GPU SDK	OS
CUDA	Intel® Xeon® X5550 2.66GHz	NVIDIA Tesla C2050	NVIDIA CUDA SDK 3.2.16	Ubuntu 10.10
OpenCL	AMD PhenomII X4940 0.8GHz	AMD RadeonTM HD 5850	ATIStream SDK 2.3	Ubuntu 9.04
227	Intel® Xeon® X5550 2.67GHz	AMD RadeonTM HD 7970	AMD APP SDK v2.6 windows 64	windows 7 ultimate 64bits

5.2　与 CPU 版本的性能对比

测试函数设定如下，bitwise 函数测试的是 scalar_mask (矩阵和向量操作，使用的 mask 参数)函数不含数据传输时的 GPU 时间和 CPU 时间，矩阵规模为 2560 * 2560，分别测试带 ROI 和不带 ROI 两种情况。

数据类型的表示意义是：8U，8S，16U，16S，32S，32F，64F 分别代表 unsigned char，char，unsigned short，short，int，float，double；C1，C2，C3，C4 分别代表单通道、2 通道、3 通道、4 通道。所以以 8UC1 为例，它代表的就是单通道的 unsigned char 数据类型。测试的数据类型为单通道的 8U，8S，16U，16S，32S，32F，64F 数据。

测试平台有 AMD HD 5850，HD 7970，NVIDIA Tesla C2050，在每个平台上测试 bitwise 函数的 CPU 时间和 GPU 时间，然后计算出 GPU 对 CPU 版本的时间加速比。性能评估和测试结果如下。

bitwise 函数的优化中使用了多种向量化方式，且各条并行指令间很少有依赖关系，可以灵活、充分地利用 AMD GPU 的向量架构，做到连续读取数据，达到较高的效率；内存对象

也选择了 buffer(相对于 image 对象的固定向量长度，buffer 这种存储方式更加灵活)来配合优化中使用的多种向量化操作；线程组织方面也是将工作组固定为 256 大小，即一个 CU 上至少有 2 个 wavefront，确保可以隐藏访存延迟；bitwise 函数的 kernel 程序中同样使用了很多的指令流方面的优化，例如使用了内建函数代替标准函数，使用了移位操作代替乘 2^n 的操作，使用了 24 位的乘法操作。综上可以预测，优化后的函数在 3 个平台上都具有较高的加速比。

图 2－图 4 是 bitwise 函数在 3 个测试平台上的 GPU 对 CPU 版本的时间加速比。图 2 是 bitwise 函数在 HD 5850 平台上的加速比，在带有 ROI 和不带 ROI 时都达到了平均约 40 倍的加速；图 3 是 bitwise 函数在 HD 7970 平台上的加速比，在带有 ROI 和不带 ROI 时都达到了平均约 90 倍的加速；图 4 是 bitwise 函数在 Tesla C2050 平台上的加速比，在带有 ROI 和不带 ROI 时都达到了平均约 60 倍的加速。综上，bitwise 函数性能与预测相符。

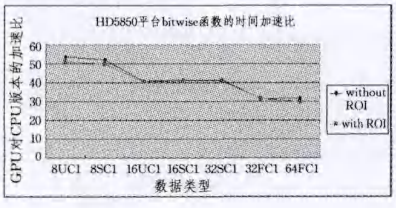

图 2　HD 5850 平台 bitwise 函数的时间加速比

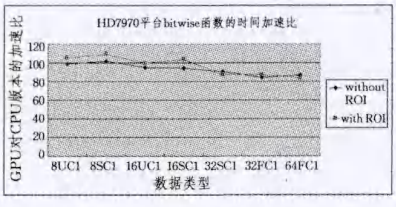

图 3　HD 7970 平台 bitwise 函数的时间加速比

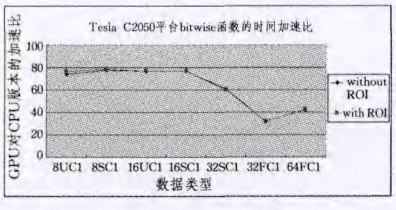

图 4　Tesla C2050 平台 bitwise 函数的时间加速比

5.3　与 CUDA 版本的性能对比

测试函数设定如下，bitwise 函数测试的是 scalar(矩阵和向量操作)函数不含数据传输时的 OpenCL 时间和 CUDA 时间，其中 scalar =(123，123，123，123)，矩阵规模为 2560 * 2560，不使用 ROI。测试的数据类型是 C1，C3，C4 通道的 8U，16U，32S 类型。测试平台是 Tesla C2050，在平台上分别测试 bitwise 函数的 OpenCL 时间和 CUDA 时间，然后计算出 OpenCL 版本对 CUDA 版本的时间加速比。性能评估和测试结果如下。

尽管 NVIDIA GPU 是标量化架构，AMD 是向量化架构，但是在函数优化中，使用向量化操作在两个平台上都能显著提高函数性能。这是因为，与在 AMD GPU 平台上向量化可以充分利用 5 路 VLIW 计算资源不同，在 NVIDIA GPU 平台上，向量指令都被转化为相互独立的标量指令执行，不会对程序性能造成实质的影响，但是向量化操作可以使每个线程处理的元素增多，提高了有效数据比，所以也可以提高函数性能[10]。综上可以预测，使用向量化在 Tesla C2050 平台上也应该可以提高函数性能。

图 5 是 bitwise 函数在 Tesla C2050 平台上 OpenCL 版本对 CUDA 版本的时间加速比，在带有 ROI 和不带 ROI 时都达到了平均约 1.5 倍的加速。测试结果与预期相符。

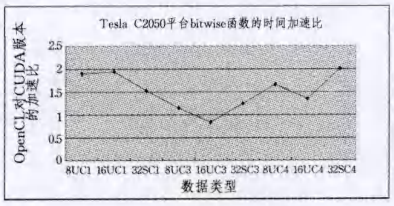

图 5　Tesla C2050 平台 bitwise 函数的时间加速比

5.4　测试结果总结

从结果可以看出，并行和优化后的函数与 CPU 版本相比，在 AMD HD 5850、AMD HD 7970 和 NVIDIA 平台上都有很好的加速比，分别有平均约 40，90 和 60 倍的加速比；在 NVIDIA 平台上与 CUDA 版本相比，也有平均约 1.5 倍的加速比，得到了很好的性能。

结束语　本文以 bitwise 函数为例，详细叙述了在 OpenCL框架下，从向量化、线程组织方式和指令选择优化等多个优化角度对连续数据无关访存密集型函数的优化策略。在 AMD 和 NVIDIA GPU 平台上，优化后函数的 OpenCL 版本与 CPU 版本和 CUDA 版本分别做了对比，从测试结果可以看出，其都获得了较好的性能加速。

参 考 文 献

[1] 陈钢，吴百锋. 面向 OpenCL 模型的 GPU 性能优化[J]. 计算机辅助设计与图形学学报，2011，23(4)

[2] ATI Stream SDK OpenCL Programming Guide[M]. rev1. 03，2010

[3] NVIDIA's Next Generation CUDA Architecture Whitepaper [M]. V1. 1，2009：8

[4] Herve CHEVANNE Dr. Ing，AMD. A Methodology For Optimizing Data Transfer in OpenCL[S]. 2011

[5] 张樱，张云泉，龙国平. 基于 OpenCL 的图像模糊化算法优化研究[J]. 计算机科学，2012，39(3)

[6] AMD Accelerated Parallel Processing OpenCL Programming Guide[M]. Rev 1. 3f，2011

[7] NVIDIA OpenCL Best Practice Guide[M]. Version 1. 0，2009

[8] NVIDIA OpneCL Programming Guide[M]. Version 4. 1，2012：56

[9] AMD 上海研发中心. 跨平台的多核与众核编程讲义 OpenCL 的方式[M]. 2010

[10] 贾海鹏，张云泉，龙国平，等. 基于 OpenCL 的拉普拉斯图像增强算法优化研究[J]. 计算机科学，2012，39(5)

[11] Yan S G，Zhang Y Q，Long G P，et al. Reduction algorithm optimization based on the OpenCL[J]. Journal of Software，2011，22 (Suppl. (2))：163-171

[12] OpenCV：Open Source Computer Vision library[OL]. http://opencv. willowgarage. com/wiki/

[13] NVIDIA Corporation. NPP：NVIDIA Performance Primitives library[OL]. http://developer. nvidia. com/npp