软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(7):1655−1675 [doi: 10.13328/j.cnki.jos.005241]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

ParaC:面向 GPU 平台的图像处理领域的编程框架*

卢兴敬[1,2], 刘 雷[1], 贾海鹏[1], 冯晓兵[1], 武成岗[1]

[1](体系结构国家重点实验室(中国科学院 计算技术研究所),北京 100190)

[2](中国科学院大学,北京 100049)

通讯作者: 卢兴敬, E-mail: xingjinglu@gmail.com

摘 要: GPGPU 加速器是当前提高图像处理算法性能的主流加速平台,但在 GPGPU 平台上,同一个程序充分利用硬件体系结构特征和软件特征的优化版本与简单实现版本在性能上会有数量级的差异.GPGPU 加速器具有多维多层的大量执行线程和层次化存储体系结构,后者的不同层次具有不同的容量、带宽、延迟和访问权限.同时,图像处理应用程序具有复杂的计算操作、边界处理规则和数据访问特性.因此,任务的并发执行模式、线程的组织方式和并发任务到设备的映射不仅影响到程序的并发度、调度、通信和同步等特性,而且也会影响到访存的带宽、延迟等.因此,GPGPU 平台上的程序优化是一个困难、复杂且效率较低的过程.提出基于语言扩展的领域编程模型:ParaC.ParaC 编程环境利用高层语言扩展描述的程序语义信息,自动分析获取应用程序的操作信息、并发任务间的数据重用信息和访存信息等程序特征,同时结合硬件平台特征,利用基于领域先验知识驱动的编译优化模型自动生成 GPGPU 平台上的优化代码,最后,利用源源变换编译器生成标准 OpenCL 程序.在测试用例上的实验结果表明,ParaC 在 GPGPU 平台上自动生成的优化版本相对于手工优化版本的加速比最高达到 3.22 倍,但代码行数只是后者的 1.2%~39.68%.

关键词: 图像处理;通用 GPU 加速器;领域编程语言;编译优化;源源变换
中图法分类号: TP314

中文引用格式: 卢兴敬,刘雷,贾海鹏,冯晓兵,武成岗.ParaC:面向 GPU 平台的图像处理领域的编程框架.软件学报,2017,28(7):1655−1675. http://www.jos.org.cn/1000-9825/5241.htm

英文引用格式: Lu XJ, Liu L, Jia HP, Feng XB, Wu CG. ParaC: A domain programming framework of image processing on GPU accelerators. Ruan Jian Xue Bao/Journal of Software, 2017,28(7):1655−1675 (in Chinese). http://www.jos.org.cn/1000-9825/5241.htm

ParaC: A Domain Programming Framework of Image Processing on GPU Accelerators

LU Xing-Jing[1,2], LIU Lei[1], JIA Hai-Peng[1], FENG Xiao-Bing[1], WU Cheng-Gang[1]

[1](State Key Laboratory of Computer Architecture (Institute of Computing Technology, The Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Image processing algorithms take the GPU accelerators as the main speedup solution. However, the performance difference between a naïve implementation and a highly optimized one on the same GPU accelerators is frequently an order of magnitude or more. The GPGPU platform features complicated hardware architecture characteristics, such as the large amount of multi-dimension and multi

* 基金项目: 国家自然科学基金(61432018, 61402445, 61502452, 61602443, 61432018); 国家重点研发计划(2016YFB1000402); 数学工程与先进计算国家重点实验室开放基金(2016A03); 北京市科委计划(D161100001216002)

Foundation item: National Natural Science Foundation of China (61432018, 61402445, 61502452, 61602443, 61432018); National Key R&D Program of China (2016YFB1000402); State Key Laboratory of Mathematical Engineering and Advanced Computing Open Foundation (2016A03); Beijing Municipal Science & Technology Commission Program (D161100001216002)

收稿时间: 2016-09-05; 修改时间: 2016-10-14; 采用时间: 2016-11-14; jos 在线出版时间: 2016-11-24

CNKI 网络优先出版: 2016-11-24 13:41:16, http://www.cnki.net/kcms/detail/11.2560.TP.20161124.1341.006.html

-level threads and the deep hierarchy memory system, while the different part of the latter features different capacity, bandwidth, latency and access authority. Additionally, image processing algorithms have complex operations, border data accessing rules and memory accessing patterns. Therefore, parallel execution model of tasks, organization of threads and parallel tasks to device mapping not only have big impact on the scalability, scheduling, communication and synchronization, but also affect the efficiency of memory accessing. In a word, the algorithm optimization methods on GPGPU platforms are difficult, complicated and less efficient. This paper proposes a domain specific language, ParaC, which can provide high level program semantics through the new language extensions. It obtains the applications' software characteristics, such as the operation information, the data reuse among parallel tasks and the memory access patterns, along with hardware platform information and the domain pre-knowledge driven optimization mechanism, to generate high performance GPGPU code automatically. The source-to-source compiler is then used to output the standard OpenCL programs. Experiment results on test cases show that ParaC automatically generated optimization version has gained 3.22 speedup compared to the hand-tuned version for the best case, while the number of lines of the former is just 1.2% to 39.68% of the latter.

Key words: image processing; GPGPU accelerator; domain specific language; compiler optimization, source-to-source translation

图像处理算法广泛应用在多媒体智能计算、游戏和人工智能等领域,但是在实际应用中经常受到运行效率的制约,无法满足实际应用场景对吞吐率或者实时性的要求.当前,利用 GPU 加速器来提高图像处理算法的性能是解决该问题的主要方法.但是,GPU 编程具有"实现容易、优化困难"的特点,GPGPU 在带来高性能的同时,也给用户带来了性能优化复杂困难、开发效率低等挑战.

在GPU加速器上开发高性能应用程序复杂困难且开发效率较低.首先,高性能收益依赖于对GPU加速器底层硬件特征的充分利用.GPU 加速器具有复杂的体系结构,包括:(1) 多层多维的处理器核,不同层次的处理器核之间具有不同的共享存储空间、同步和通信机制;(2) 层次化存储体系结构,包括全局内存、片上共享内存、私有内存、片上缓存、寄存器和常数内存,不同的存储空间具有不同的访问权限、容量、延迟和带宽,同时还可能影响程序的执行并发度.用户必须熟悉底层硬件特性并为核心函数选择恰当的任务组织形式和数据分布模式来适应这些特征.其次,程序员需要制定优化方案并确定其优化参数空间,该过程需要丰富的调优经验以及不断的尝试来实现.最后,上述过程开发的应用程序与硬件特征紧耦合,限制了程序在平台间的可移植性,例如,针对某个硬件平台的优化程序,可能无法保证性能的移植到不同厂商甚至相同厂商不同代的 GPU 加速器上,需要用户针对新平台进行重新开发及优化.

目前,为了解决 GPGPU 平台上的软件开发效率、性能优化和平台可移植性问题,学术界和产业界已经提出了多个 GPGPU 平台上的并行软件开发环境.为了充分利用 GPU 加速器复杂的底层硬件特征,GPU 加速器生产厂商提供了通用的底层编程框架,例如 OpenCL[1]和 CUDA[2],这些框架支持用户直接针对底层硬件特征进行编程,因此具有很大的性能优化空间,用户通过充分的手工优化可以获得最优的性能.为降低手工开发与优化带来的编程负担,提出了基于制导的通用编程环境,例如 OpenMP[3]和 OpenACC[4]等,这些编程环境利用程序员给出的制导信息,通过编译器和运行时自动完成部分代码分析和程序优化变换,从而提高了程序员的开发效率.但是,程序高层语义的缺失,限制了编译环境的自动调优空间,为此引入了能够提供程序高层语义的领域编程模型,例如 Halide[5,6]和 HIPA^{cc}[7,8].Halide[5,6]是一个针对图像处理算法的领域编程模型,其核心思想是:首先,算法和调度的分开描述提高了算法的可读性及可移植性;其次,由用户指定算法的调度策略;最后,由编译器和运行时根据调度策略自动完成程序变换和代码生成,从而减轻了程序员的性能优化负担.HIPA^{cc}[7,8]是基于 C++类概念实现的图像领域编程模型,其设计理念是:首先,将数据访问和程序执行分离,由用户编写程序的核心函数以及数据访问描述符;其次,针对图像处理算法中的数据存储、访问、边界处理和操作符抽象出了对应的高层语言扩展;最后,HIPA^{cc}编译器利用用户提供的数据访问信息和高层语义来完成性能优化.在上述编程模型的基础上,本文设计实现了针对图像处理的领域编程模型:ParaC.首先,该模型提供了描述处理对象、通用边界处理和高层算子运算符的语言扩展,该扩展能够简化用户的编程;其次,该编译器能够分析获取程序的高层语义和领域知识,并结合优化模型制定程序优化方案,不需要用户描述访存信息或者调度方案;再次,其编译器能够自动完成所有的程序分析和优化变换工作,从而降低了 GPGPU 平台上的编程复杂度;最后,源源翻译编译器生成标准 OpenCL 程序,最终利用硬件厂商提供的 OpenCL 编译器生成目标代码.

本文的主要贡献包括:

1) 提出了针对图像处理的领域编程语言:ParaC,该编程语言是基于标准 C 的进一步扩展,是对图像处理领域的数据对象、边界处理和数值计算的高层抽象,能够简化程序员的编程并为后续的编译优化提供领域知识.

2) 设计实现了先验领域知识驱动的编译优化模型,基于该模型的 ParaC 编译器自动实现了 GPU 加速器平台上的数据分布、访存和并行调度等优化.

3) 一个将 ParaC 程序转换成 GPU 加速器上 OpenCL 程序的源源翻译编译器.

4) 用 ParaC 实现了部分具有代表性的图像处理算法,并与手工编写的优化版本进行了性能对比,通过具体的性能评估,实验结果表明,ParaC 能够在提高用户编程效率的同时提供非常好的性能.

本文第 1 节介绍 ParaC 语言扩展和用法示例.第 2 节介绍编译框架,主要包括主要工作流程和编译优化技术.第 3 节讨论 ParaC 编程环境的实验结果,并具体分析编程效率和性能.第 4 节介绍相关工作.第 5 节对本文工作进行总结.

1 ParaC 语言设计

在完成对图像处理算法的领域特征分析和算法分类后,设计了既简洁又具有较强表达性的领域编程语言:ParaC.目前,已有多个对图像处理算法进行分类的工作.Bankman[9]从被用于计算目标图像的源操作数信息的角度将图像操作划分成 3 类:(1) 像素点操作:用源图像的一个点更新目标图像;(2) 局部块操作:用源图像上包含邻居像素点的多个像素点来更新目标图像;(3) 多图像操作:用多个图像来更新目标图像.Russ[10]从操作的目标操作数信息这一角度将图像操作划分成多个类,包括提高某特征可见度的空域图像增强和提高可计算性的频域图像处理等.Klette[11]采用了与 Bankman 类似的划分方法,只是将多图像操作替换成全局操作符.

本文采用了 Klette 的划分方法,按照源操作数信息将操作划分成点操作、局部块操作和全局操作.本文从图像对象的描述、执行模型和边界处理 3 个方面对标准 C 语言进行了扩展.

1.1 数据类型扩展

作为图像处理算法输入的二维图像,其通道和像素点集合恰好构成三维矩阵,最高维对应于通道,低两维用于描述平面上的像素点,针对图像的这一特征,本文引入了多维矩阵数据类型来表示图像对象.另外,有些图像操作可能会按行或者列对图像进行处理,为了便于描述此时的操作数对象,本文引入了向量数据类型.

图 1 给出了 ParaC 中新数据类型的语法规则,包括变量的声明、访问、引用和与 C 语言的交互.类型为 *parac_matrix* 的变量表示图像,目前最高支持三维,未来可根据需求扩展到更高维度.当 *parac_matrix* 变量是三维时,其最高纬度表示图像通道,低两维表示图像平面上的像素点;否则,该变量表示单通道图像.类型 *parac_vector* 表示图像上的行或者列对象,不包含通道信息.在声明上述两种类型的变量时,需要显式指明每个维度的长度,例如 *int parac_matrix* ImgMat[C][H][W]/ImgMat[H][W].

在数据访问语法方面,用户既可以访问图像上的任意单个像素点,也可以访问图像相邻区域的多个像素点,前者通过下标方式进行访问,例如 ImgMat[*x*][*y*],表示访问坐标为(*x*,*y*)的像素点;后者通过散列方式来进行描述,例如 ImgMat[*x*:5:1][*y*:5:1],该散列表达式表示以坐标值为(*x*,*y*)的像素点为起点,在 *X*/*Y* 方向上长度分别为 5 且相邻像素点间隔为 1 的矩形窗口.全局操作既可以通过迭代器和上述两种方式相结合的方法表达,也可以通过隐式的全局图像遍历来表达,例如后文图 6 所示第 1 行代码中的 *pEdge* 变量.*parac_vector* 类型的变量有类似的访问语法.

ParaC 支持声明 *parac_matrix*/*parac_vector* 类型的引用对象,且必须在声明时初始化为 C 数据类型的变量,ParaC 以引用的方式实现与 C 语言中数组或指针所对应存储空间的互操作,例如 *char**ImgSrc=(*char**) malloc (sizeof(*char*)*ROWS*COLS);*char parac_matrix* &ImgMat[ROWS][COLS]=ImgSrc,这时,ImgMat 与 ImgSrc 指向相同的存储区域.

1.2 执行模型扩展

图像算法通常遍历源图像的感兴趣区域(ROI)并完成特定的计算,ParaC 引入迭代器和算子来支持和简化

ROI 和常用特定计算的实现,后者也隐式地给出了算法的领域知识.图 2 给出了迭代器和算子的语法规则,迭代器可用于描述 ROI,并支持用户指定可能的边界处理规则;算子运算符的输入是矩阵、向量和两者的下标访问或者散列区间,算子运算符的参数或者操作数隐式定义了单次迭代计算时的操作数访问区间.

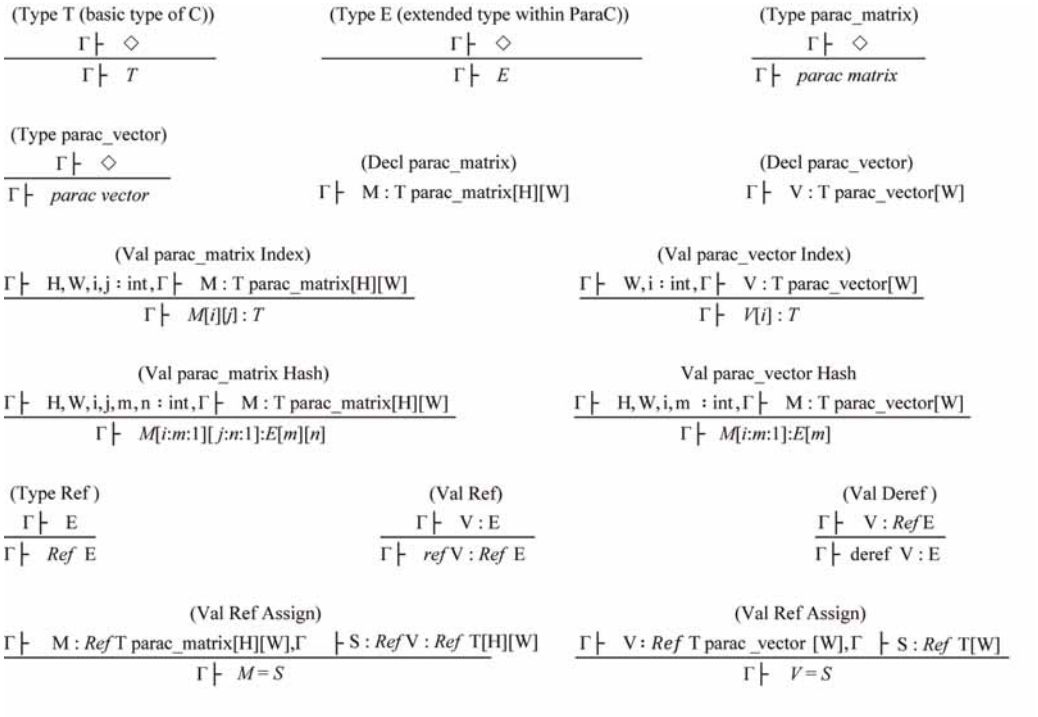

Fig.1 The grammar rules of new extended data types within ParaC

图 1 ParaC 新扩展数据类型的语法规则

```
IterStmt::=IterStmt Br Stmt|parac_iterator id(Expr:Expr:Expr;Expr:Expr:Expr)      //迭代器
Br::=parac_border_rule(id)|ε                                                      //边界规则
Stmt::={Stmt}|Stmt|Expr|ParaCExpr
ParacExpr::=Operator|ParaCExpr+ParaCExpr|ParaCExpr–ParaCExpr                      //矩阵、向量点操作
            |ParaCExpr.*ParaCExpr|id                                              //id 表示矩阵、向量或者对应的散列
Expr::=Expr|Expr+Expr|Expr–Expr|Expr*Expr|Expr/Expr|id|ε                          //C 语言表达式
Operator::=parac_matrix_sum(Params)|parac_matrix_min(Params)                      //算子运算符
            |parac_matrix_max(Params)|parac_matrix_mean(Params)
            |parac_convolution(Params, Params)
Params::=ParaCExpr                                                                //算子的参数
```

Fig.2 The grammer rule of exented operators within ParaC

图 2 ParaC 中迭代器、算子的语法规则

图 3 是一个用迭代器描述并行区域的代码示例,第 1 行代码中的迭代器 *piter* 定义了每个维度的下界、上界和间隔,其中,上、下界可以是表达式或者常数,但是间隔必须是编译时常量.目前,迭代空间只支持一维到三维,将来可以根据需求扩展到更高维度.*parac_iterator* 除了定义迭代空间外,还隐式地表示迭代体内的计算不存在迭代间真依赖,可以并发执行.另外,ParaC 不支持 *parac_iterator* 之间的嵌套.第 4 行代码调用了 *parac_matrix_sum* 算子,其输入是矩阵 *matA* 上的散列空间,表示对该散列空间上的点进行求和.

```
1 parac_iterator piter(lbx:ubx:stridex; lby:uby:stridey)
2 {
3   matA[itx][ity] = matB[itx][ity] + matC[itx][ity];
4   matD[itx][ity] = parac_matrix_sum(matA[itx][ity+offsety:5:1])/5;
5 }
```

Fig.3 An example of parac_iterator rule

图 3 迭代器语法规则示例

1.3 边界处理

如图 4 所示,图像处理算法中的局部块操作利用源图像的多个相邻像素点来计算目标图像的对应像素点,所以当计算目标图像边界像素点值时,可能会存在访问源图像越界像素点的情况.图 4(a)和图 4(b)表示,当局部块计算需要行或者列方向上的相邻像素点时,计算边界上的点会访问列或行方向上的越界数据.图 4(c)表示,当局部块计算需要二维相邻数据块时,在行和列上同时会访问越界数据,边界规则会被分成 8 个子区域进行描述.

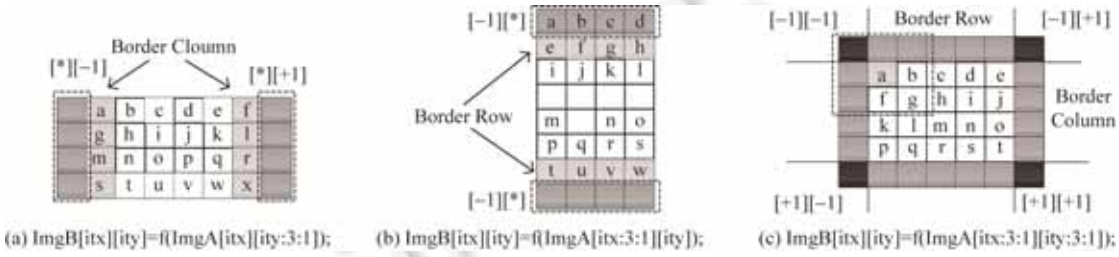

Fig.4 Type of boundary data accessing

图 4 图像边界数据访问类型

为了避免对越界数据的访问,用户可以通过增加分支控制语句对越界访存作特殊处理.但是,控制流分支既降低了程序员的开发效率和程序的可读性,又增加了后续编译分析优化的难度.ParaC 设计的边界处理规则只要求程序员显式地描述越界数据的计算公式,由编译器自动地在合适的程序点生成并插入相关数据访存操作的中间表示.

图 5 给出了边界处理规则的语法定义,边界外像素点的值表示成其他像素点值构成的表达式,该设计来自于对图像领域特征的观察总结:对于图像边界外的数据,通常是常数或者由其他内部像素点值计算得到的值.另外,ParaC 的边界规则的语法是通用的,但是当前编译器实现不支持边界值依赖于其他边界值的情况.编译器会保证边界值计算过程中的数据一致性,并且会对程序员给出的边界宽度进行静态检查,以保证其值不小于程序实际执行过程中需要的最大边界宽度.

```
1. 常整数:CONSTANT;图像:Src[M][N],其中,M,N 表示高和宽
2. *:正则表达式通配符,其值范围为[0,M-1]或者[0,N-1];
3. -CONSTANT:表示图像上侧或者左侧的越界数据;
4. +CONSTANT:表示图像下侧或者右侧的越界数据;
5. Src[*][-1]=CONSTANT; //左侧第 1 列边界的值赋给常数值
6. Src[*][+1]=Src[*][N−2]*2-Src[*][N-4]; //右侧第 1 列边界的值
7. Src[-1][*]=CONSTANT; //上侧第 1 行的边界值
8. Src[+1][*]=Src[N-1]+Src[N-2]*3; //下侧第 1 行的边界值
     //依次为左上角,左下角,右上角和右下角的边界值
9. Src[-1][-1]=Src[-1][+1]=Src[+1][-1]=Src[+1][+1]=CONSTANT;
```

Fig.5 Rule of boundary data accessing

图 5 定义边界处理规则语法定义

1.4 ParaC编程示例

为了便于理解 ParaC 程序,本文对常用的术语进行了定义:由 *parac_iterator* 描述的代码区域为迭代区,迭代区内的算子运算符为嵌套算子;没有被包含在任意迭代区内的算子运算符为独立算子;由新扩展数据类型修饰的变量声明、定义和引用为 ParaC 变量;访问 ParaC 变量的语句、嵌套算子、独立算子和迭代区统称为 ParaC 语句.所以,ParaC 程序是包含 ParaC 变量和 ParaC 语句的代码区域.

图 6 给出了用 ParaC 实现的部分图像锐化算法模块,根据上述定义,图 6 中第 1 行代码中的 *parac_matrix_sum* 是独立算子,表示由其参数 *img* 隐式确定迭代空间内所有像素点的求和操作;第 3 行~第 4 行、第 8 行~第 16 行、第 17 行~第 23 行和第 25 行~第 26 行代码区域都是迭代区,其中,第 3 行~第 4 行和第 25 行~第 26 行所定义的迭代区在两个维度上都存在并行性,第 17 行~第 24 行和第 25 行~第 26 行两个迭代区分别在行和列方向上具有并行性.

```
1 short mean=parac_matrix_sum(pEdge) /(M*N)+2/2;
2 int DownScaleNewX16 (unsigned char parac_matrix &pImageSrc[M][N], unsigned char parac_matrix &pImageDst[M/4][N/4]){
3   parac_iterator it(iter_element; 0:M:4; 0:N:4){
4         pImageDst[itx/4][ity/4]=(parac_matrix_sum(pImageSrc[itx:4:1][ity:4:1])+8)>>4;}}
5 int UpScaleNewX16 (unsigned char parac_matrix &srcImage[M/4][N/4], unsigned char parac_matrix &dstImage[M][N]){
6   float parac_matrix border_mp_0[4][1]={1.0, 3/4.0f, 2/4.0f, 1/4.0f};
7   float parac_matrix border_mp_1[4][1]={0.0, 1/4.0f, 2/4.0f, 3/4.0f};
8   parac_iterator it1(0:M/4:1;0:1:1){
9     if (it1x==M/4-1){
10        dstImage[it1x*4][0]=dstImage[it1x*4+1][0]=dstImage[it1x*4+2][0]=dstImage[it1x*4+3][0]=srcImage[it1x][0];
11        dstImage[it1x*4][1]=dstImage[it1x*4+1][1]=dstImage[it1x*4+2][1]=dstImage[it1x*4+3][1]=srcImage[it1x][0];
12        dstImage[it1x*4][M-2]=dstImage[it1x*4+1][M-2]=
                              dstImage[it1x*4+2][M-2]=dstImage[it1x*4+3][M-2]=srcImage[it1x][M/4-1];
13        dstImage[it1x*4][M-1]=dstImage[it1x*4+1][M-1]=
                              dstImage[it1x*4+2][M-1]=dstImage[it1x*4+3][M-1]=srcImage[it1x][M/4-1];
14    } else{
15        dstImage[it1x*4:4:1][0]=dstImage[it1x*4:4:1][1]=srcImage[it1x][0]*border_mp_0+srcImage[it1x+1][0]*border_mp_1;
16        dstImage[it1x*4:4:1][M-2]=dstImage[it1x*4:4:1][M-1]=srcImage[it1x][M/4-1]*border_mp_0+srcImage[it1x+1][M/4-1]*
          border_mp_1;}}
17  parac_iterator it2(;0:N/4:1){
18    if (it2y==N/4-1){
19        dstImage[0][it2y*4]=dstImage[0][it2y*4+1]=dstImage[1][it2y*4]=dstImage[1][it2y*4+1]=srcImage[0][it2y];
20        dstImage[N-2][it2y*4]=dstImage[N-2][it2y*4+1]=dstImage[N-1][it2y*4]
                                                    =dstImage[N-1][it2y*4+1]=srcImage[N/4-1][it2y];
21    } else{
22        dstImage[0][it2y*4:4:1]=dstImage[1][it2y*4:4:1]=srcImage[0][it2y]*transe(border_mp_0)
                                                    +srcImage[0][it2y+1]*transe(border_mp_1);
23        dstImage[N-2][it2y*4:4:1]=dstImage[N-1][it2y*4:4:1]=srcImage[N/4-1][it2y]*transe(border_mp_0)+
           srcImage[N/4-1][it2y+1]*transe(border_mp_1);}}
24  float parac_matrix mp[4][2]={7/8.0f, 1/8.0f, 5/8.0f, 3/8.0f, 3/8.0f, 5/8.0f, 1/8.0f, 7/8.0f};
25  parac_iterator it(0:M/4-1:1; 0:N/4-1:1){
26    dstImage[itx*4+2:4:1][ity*4+2:4:1]=mp*srcImage[itx:2:1][ity:2:1]*transe(mp);}}
```

Fig.6 The mean, downsample and upsample of sharpness algorithm written with ParaC

图 6 图像锐化算法中的均值、下采样和上采样算法的 ParaC 实现

图 7 展示了用 ParaC 来实现的拉普拉斯金字塔算法.如果迭代区计算涉及到边界数据处理,则用户需要在迭

代器上增加对相关边界处理规则的引用.例如,图 7 中的第 11 行~第 12 行代码所定义的迭代区标记了其用到的边界规则 *br*,该规则由代码第 6 行~第 10 行定义,该迭代区描述了在水平方向上的滤波操作.第 18 行~第 19 行代码所定义的迭代区描述了垂直方向上的滤波操作和下采样运算,其用到的边界规则是 *br*1,该越界规则由第 13 行~第 17 行代码定义;第 22 行~第 40 行代码定义了上采样运算.

```
1  int Laplacian(unsigned char parac_matrix &Src[M][N],
   unsigned char parac_matrix &layer[M][N]){
2  unsigned char parac_matrix filter_horizon[1][5]={1,4,6,4,1};
3  unsigned char parac_matrix filter_vertical[5][1]={1,4,6,4,1};
4  unsigned char parac_matrix dst_horizon[M][N],
   dst_vertical[M][N];
5  unsigned char parac_matrix dst_ds[(M+1)/2][(N+1)/2];
6  parac_border_rule br{
7     Src[*][-2]=Src[*][3]*4-Src[*][1]*4+Src[*][0]*2-Src[*][2];
8     Src[*][-1]=Src[*][1]*2-Src[*][3];
9     Src[*][+1]=Src[*][N-2]*2-Src[*][N-4];
10     Src[*][+2]=Src[*][N-4]*4-Src[*][N-2]*4+Src[*][N-1]*2-
       Src[*][N-3];}
11 parac_iterator it(0:M:1; 0:N:1) parac_border_rule(br){
12   dst_horizon[itx][ity]=(parac_convolution(filter_horizon,
     Src[itx][ity-2:5:1])+8)/16;}
13 parac_border_rule br1 {
14     dst_horizon[-2][*]=dst_horizon[3][*]*4-dst_horizon
       [1][*]*4 + dst_horizon[0][*]*2-dst_horizon[2][*];
15     dst_horizon[-1][*]=dst_horizon[1][*]*2-dst_horizon[3][*];
16     dst_horizon[+1][*]=dst_horizon [M-2][*]*2-dst_horizon
       [M-4][*];
17     dst_horizon[+2][*]=
       dst_horizon[M-4][*]*4-dst_horizon[M-2][*]*4+
       dst_horizon[M-1][*]*2-dst_horizon [M-3][*];}
18 parac_iterator it1(0:(M+1)/2:1; 0:(N+1)/2:1)
   parac_border_rule(br1){
19     dst_ds[it1x][it1y]=(parac_convolution(filter_vertical,
                          dst_horizon [it1x*2-2:5:1][it1y*2])+8)
                           /16; }
20 int halfWidth=N/2, halfHeight=M/2;
21 int widthOdd=N-(halfWidth*2), heightOdd=M-
   (halfHeight*2);
```

```
22 parac_iterator it(0:(M+1)/2:1; 0:(N+1)/2:1){
23   if (itx==0){
24     layer[0][ity*2]=Src[0][ity*2]-dst_ds[0][ity];
25    if (ity !=(N+1)/2-1) layer[0][ity*2+1]=Src[0][ity*2+1]-
      (dst_ds[0][ity]+dst_ds[0][ity+1]+1)/2;
26     if (ity==(N+1)/2-1 && widthOdd==0)
27        layer[0][ity*2+1]=Src[0][ity*2+1]-dst_ds[0][ity];}
28    if (heightOdd==0 && itx==(M+1)/2-1){
29      layer[M-1][ity*2]=Src[M-1][ity*2]-
        dst_ds[(M+1)/2-1][ity];
30      if (ity !=(N+1)/2-1) layer[M-1][ity*2+1]=
        Src[M-1][ity*2+1]-(dst_ds[(M+1)/2-1][ity]+dst_ds
        [(M+1)/2-1][ity+1]+1)/2;
31      if (ity==(N+1)/2-1 && widthOdd==0)
          layer[M-1][ity*2+1]=Src[M-1][ity*2+1]-
          dst_ds[M-1][ity];}
32    if (itx!=0){
33      layer[2*itx-1][2*ity]=Src[2*itx-1][2*ity]-
        (dst_ds[itx-1][ity]+dst_ds[itx][ity]+1)/2;
34      layer[2*itx][2*ity]=Src[2*itx][2*ity]-dst_ds[itx][ity];
35      if (ity !=(N+1)/2-1){
36        layer[2*itx-1][2*ity+1]=Src[2*itx-1][2*ity+1]-
          (dst_ds[itx-1][ity]+dst_ds[itx-1][ity+1]+
          dst_ds[itx][ity]+dst_ds[itx][ity+1]+2)/4;
37        layer[2*itx][2*ity+1]=
          Src[2*itx][2*ity+1]-(dst_ds[itx][ity]+
                              dst_ds[itx][ity+1]+1)/2;}
38      if (widthOdd==0 && ity==(N+1)/2-1){
39        layer[2*itx-1][2*ity+1]=Src[2*itx-1][2*ity]-
          (dst_ds[itx-1][ity]+dst_ds[itx][ity]+1)/2;
40        layer[2*itx][2*ity+1]=
          Src[2*itx][2*ity]-dst_ds[itx][ity];}}}
41 return 0;}
```

Fig.7 The ParaC code of Laplacian pyramid algorithm

图 7 ParaC 版本的拉普拉斯金字塔程序

2 编译框架

本文基于 Clang[12]编译器实现了一个从 ParaC 程序到 OpenCL 程序的编译器,首先,该编译器在 Clang 前端基础上增加了支持 ParaC 新语言特性的语法分析和语义分析模块;其次,在原有 Clang 中间表示(AST)之上增加了新的高层中间表示(简称为 HAST),用于表示程序的高层语义,如迭代器、算子运算符、数据访问模式等信息,该 HAST 中间表示在语义分析阶段生成;再次,在 HAST 中间表示上进行程序特征分析和优化变换,并下降到 AST 中间表示;最后经由源源变换生成 OpenCL 程序,再借助于目标硬件平台的 OpenCL 编译器生成最终的可

执行程序.

2.1 编译器工作流程

图 8 描述了编译器的工作流程.

(1) 编译器前端把标准 C 和 ParaC 共同描述的程序翻译成 Clang 的高层中间表示(HAST),然后遍历该高层中间,将程序划分为分别由标准 C 和 ParaC 实现的程序区域.

(2) 遍历 ParaC 程序的中间表示并将其划分成主机端和设备端.

(3) 对于设备端的程序,编译器依次进行程序特征分析、优化变换和源源变换;最后生成目标平台的 OpenCL 程序.

(4) 主机端程序包括 C 程序和 ParaC 程序中的主机端部分,结合设备端的程序特征分析结果,生成控制任务映射、数据传输、内存分配和核心函数建立等功能的代码.

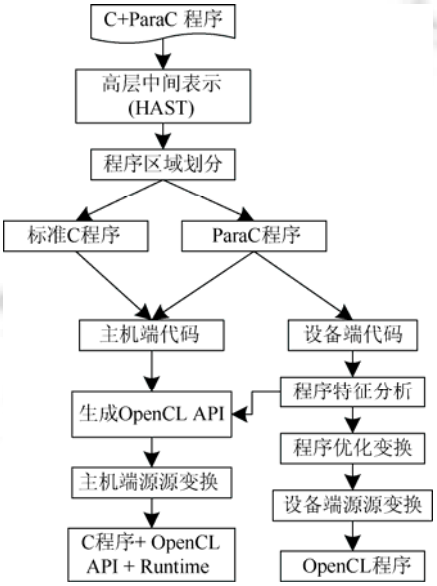

Fig.8 The work flow of ParaC compiler

图 8 ParaC 编译器的整体流程图

2.2 先验领域知识驱动的优化模型

本文构建了包含图像处理算法的领域特征和相关优化经验的先验领域知识库,并据此设计了先验领域知识驱动的优化模型,如算法 1 所示.该模型首先根据程序的高层语义和领域知识将运算划分成不同类型;其次,分析其关键程序特征,并根据先验领域优化知识初步确定可用优化策略集合;再次,根据硬件特征确定并实例化每种优化方案;最后,利用搜索算法通过对比不同候选优化方案所生成代码的执行效率选出最优的实现.通过在优化模型中集成硬件特征,能够使得 ParaC 编译器生成适合目标硬件平台的程序.

该模型利用程序的高层语义和领域知识,自动地将 ParaC 程序划分为点运算、局部块运算和全局运算.根据第 1.4 节中的定义,独立算子定义的并行区域属于全局运算;包含在迭代区内的 ParaC 语句可能构成局部块运算;点运算既有可能是全局运算,也可能是包含在迭代区内的 ParaC 语句.

图 7 所示的第 33 行代码中的源操作数为图像中的单个像素,所以被划分为点运算,该类型运算由不存在数据重用的细粒度数据并行任务构成.优化点运算的重点是任务的组织和映射,不需要进行数据分布等其他复杂优化,并且不会影响其他运算类型优化策略的选择.

图 9 给出了局部块运算和全局运算可能的优化方案的示例.第 1 个操作是 *parac_convolution* 算子,其运算类型、任务的维度和每个维度的大小由其参数决定,在该例子中,其参数为 3×3 的矩形区域,所以可知该算子是

大小均为 3 的二维局部块操作.然后根据描述数据访问的散列表达式确定其任务间存在数据重用;再根据其可在水平方向上并行执行确定利用向量化并行来减少访存次数.第 2 个操作为相邻点分别与向量相乘后的向量和,其源操作数为[itx][0]和[itx+1][0],同理得知其为大小为 2 的单维局部块操作.然后根据其访问下标确定其任务存在垂直方向上的数据重用,由于重用的数据间是不连续的,根据先验优化知识,此时采用共享内存优化效果较好.第 3 个操作是 *parac_matrix_sum* 独立算子,根据其高层语义可知,其为全局运算,大小和维度与图像一致,根据先验优化知识,对其采用分层次的树形归约优化来增加并行度及降低同步开销.

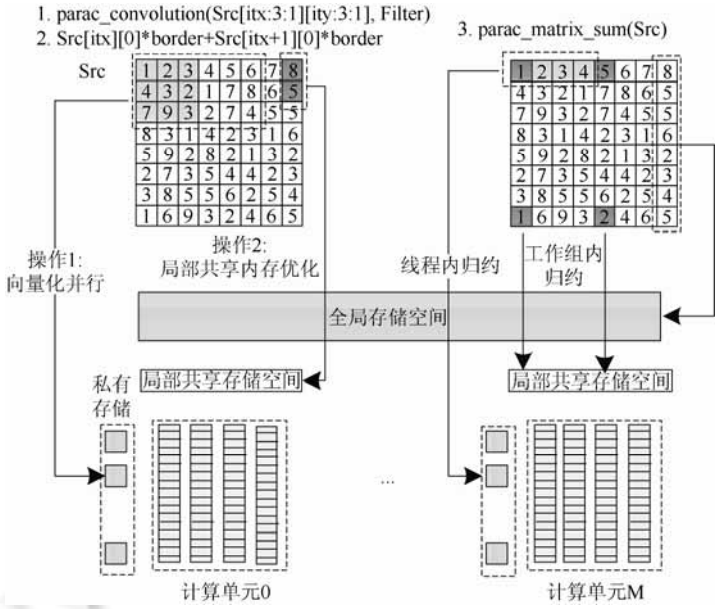

Fig.9 The mapping to memory hierarchies of GPGPU of application variables under different optimization algorithms

图 9 在不同优化算法中,程序数据到 GPGPU 上不同存储空间的映射

算法 1. 领域先验知识驱动的优化模型.

输入:程序特征和硬件特征;

输出:根据优化模型制定的优化方案.

```
FOR EACH ParNode in ParScope
  FOR EACH CVar in ChunkVar of ParNode
    IF CVar→HReuse==1                          //水平方向数据重用
    THEN insert CVar into the set VectAcc;     //对变量 CVar 进行向量化访存
         ParInfo→VectorHParFlag=1;             //水平方向向量化并行
         IF ReuseLen>UpRULen
         THEN ParInfo→LDSFlag=1;               //LDS 优化
    IF CVar→VReuse==1
    THEN CVar→LDSFlag=1;                       //存在垂直方向数据重用,LDS 优化
         insert CVar into set ClsAcc;          //Coalescing 优化
  IF ParGWS[0]*ParGWS[1]⇐LowGTS*VectorParLen
  THEN LDSFlag=LDSFlag||VectorParFlag;         //LDS 优化
  ELSE IF CVar→VReuse==1 && CVar→HReuse==0
```

THEN *ParInfo*→*VectorVParFlag*=1; //只在垂直方向存在重用,水平方向向量化并行

//Optimization of reduction operation.

FOR EACH reduction operation *ReducOpr*

IF *ReducOpr* is local reduction operation

THEN do reduction operation serially

ELSE IF *ReducOpr* is global reduction operation

THEN *GRTOptFlag*=1

在执行优化模型之前,编译器首先根据算法 2 分析程序特征.分析收集的程序特征包括并行性信息、全局任务维度及空间和数据存储访问模式等信息.并行性信息包括独立算子运算符和迭代区的集合,通过分析其高层语义能够获取每个并行区域的全局任务维度和每个维度的大小.分析程序访存信息的步骤包括:(1) 收集每个引用点的访存模式;(2) 将每个引用点的信息合并到对应的变量上,从而得到每个变量在并行区域内的访问信息;(3) 在变量的访问信息集合的基础上获取变量的访问模式,例如点数据访问还是块数据访问.在获取这些访存信息的基础上,算法 2 还可以进一步分析数据在任务间的数据重用情况.

算法 2. 程序特征分析.

输入:ParaC 程序经前端生成 HAST 中间表示;

输出:程序特征集合(ParScope:并行区域集合;ParDim:全局任务空间维度;ParGWS[ParDim]:全局任务空间的大小).

//Step 1. Analysis of parallelism information.

FOR EACH *Node* in HAST IR

IF *Node* is an IteratorScope||IndependentOperator

THEN Insert *Node* into the set of parallel scope: *ParScope*

FOR EACH *ParNode* in *ParScope*

setup variable *ParInfo* to store characteristics of *ParNode*

IF *ParNode* is IteratorScope

THEN get the header info of the iterator

get *ParDim* and *ParGWS*[*ParDim*] *information* of global tasks

ELSE IF *ParNode* is IndependentOperator

THEN get high level information the IndependentOperator

get *ParDim* and *ParGWS*[*ParDim*] information of global tasks

//Step 2. Analysis of memory accessing pattern

FOR EACH *Stmt* in *ParNode*

IF *Stmt* is Operator of ParaC

THEN for each parameter *Var* of the operator

AnalysisAndRecordAccInfo(*ParInfo*, *Var*)

ELSE IF *Stmt* is C operation

THEN for each operand *Var* of Stmt

AnalysisAndRecordAccInfo(*ParInfo*, *Var*)

FOR EACH *Var* in *DecVarRefInfo*

Init Memory Access Type:*PointFlag*=1 and insert *Var* into set *PointVar*;

FOR EACH reference point *RefPoint* of *Var*

IF *RefPoint* is chunk operation and *PointFlag*==1 //局部块操作

THEN *PointFlag*=0 and delete *Var* from set *PointVar*;

```
            insert Var into set ChunkVar;
    FOR EACH ChVar in ChunkVar
      IF ChVar is reused among different iterations
      THEN
        IF the data reuse of ChVar is in the horizontal direction THEN HReuse=1;
       ELSE IF the data reuse of ChVar is in the vertical direction THEN VReuse=1;
    AnalysisAndRecordAccInfo(ParInfo, Var){
      Analyzing memory accessing information of Var, such as def/use, ref, and access scope;
      Setup the map between reference_point and access_info for each reference point of Var: VarRefInfo
      Setup the map between DeclVar and RefPointInfo for each Var: DecVarRefInfo
      }
```

算法 3 描述了根据优化模型制定的优化方案进行相关程序变换的过程.对于需要进行向量化并行变换的算法,首先根据向量化并行宽度重新计算全局任务在低维度的大小,另外还需要对其他变量作类型或者操作符的提升,使其变成相应的向量化并行操作.对于需要共享内存优化的代码区域,需要编译器自动设定局部工作组的维度和对应大小,并且在核心函数的参数或者函数体内声明相应大小的局部共享变量.对于需要 coalescing 优化的访存,需要计算局部工作内所有任务需要的数据总量,并在任务间平均划分读取的数据大小.对于全局归约操作,会生成对应的线程内、工作组内和工作组间的归约操作.

算法 3. 优化变换及代码生成.

输入:HAST 中间表示和优化策略;

输出:代码变换后的中间表示.

```
FOR EACH ParNode in ParScope
  IF ParNode→ParInfo→VectorParFlag==1
  THEN ParGWS[0]=ParGWS[0]/VectorParLen;
  IF ParNode→ParInfo→LDSFlag==1
  THEN initializing LWS[ParDIM] for local work group
  //Vector parallelizing code generation.
  IF ParNode→ParInfo→VectorParFlag==1
  THEN
      FOR EACH Var in ParNode
        compute accessing scope of Var
        promote Var to corresponding vector type;
        generate load/store operations for Var;
        IF Var is in VectAcc
        THEN generate Vector memory access operations
             Transform original operations into vector types
  IF ParNode→ParInfo→LDSFlag==1
  THEN
      FOR EACH Var in ParNode
        generate memory access operation for Var
        IF Var is in ClsAcc
        THEN compute Acess_Scope for each task
                generate memory access operation
```

insert synchronization operation;

IF *ParNode→ParInfo→GRTOptFlag*

THEN

generate reduction operation for each thread;

generate reduction operation with each work group;

generate reduction operation among work groups;

2.3 编译优化变换

2.3.1 自动填充技术

数据是否对齐不仅会影响到访存效率,还会影响到后续向量化并行、向量化访存以及高效访存指令的选择,但是根据 OpenCL[1]的定义,当前的 OpenCL 编译器只能保证变量在起始地址对齐,因此对于长宽分别为 *m* 和 *n* 的图像 *ImgMat*,如果该数据元素类型为 short 且 *n* 不是 32 的整数倍,则会存在首地址非对齐的行,从而产生访问非对齐地址的问题.自动填充技术可以保证 GPU 全局地址空间上图像数据的每行起始地址是对齐的.

ParaC 编译器通过在每行末尾增加一定的存储空间来保证其起始地址都是 64 字节对齐的.填充后的每行字节数 *PadWidthLen* 的计算公式如下,*Width* 和 *WidthLen* 分别表示原变量每行的元素个数和字节数,*CLineLen* 表示当前机器的 cache line 长度.

$$WidthLen = sizeof(TyName) \times Width,$$

$$RemainLen = WidthLen \% CLineLen,$$

$$PadLen = RemainLen\ ?\ (CLineLen - RemainLen) : 0,$$

$$PadWidthLen = WidthLen + PadLen.$$

2.3.2 共享内存优化

根据基于先验领域知识的优化模型,ParaC 编译环境会利用共享内存优化解决两类问题:(a) 组内通信:利用共享内存和组内操作实现工作项之间的通信,并保证数据的一致性;(b) 降低程序的访存带宽需求:当组内工作项之间存在数据重叠时,利用局部共享内存减少工作组的总体访存次数.例如,对图 9 所示的第 3 个操作 *parac_ matrix_sum* 进行树形归约优化中利用局部共享内存实现通信.另外,在利用共享内存降低带宽需求时,会进一步利用 coalescing 优化提高访存效率.

但是,该优化也会受到硬件特性的约束,例如,在 AMD FirePro W8000 平 GPU 加速器平台上,单个计算单元的最大共享内存是 64KB,单个工作组最多可以访问 32KB 的共享内存.因此,ParaC 编译环境在实现局部共享内存优化时,会获取硬件特征并分析并行区域内的局部共享内存的需求量,协调组内工作项数目和单个工作项需要的局部共享内存空间来使得相关优化满足约束.

2.3.3 向量化并行

优化模型中用向量化并行减少工作组的整体访存次数和工组项数目.对于局部块操作,例如上采样、下采样、横向滤波、纵向滤波和卷积等算法,其相邻工作项之间可能会存在数据重用,因此可以利用向量化并行优化减少工作组的总访存次数,从而降低带宽需求.

另外,随着计算任务的增加,如果每个工作项只计算单个任务,会增加工作组的数目.GPU 加速器是以工作组作为 CU 上的基本调度单位,因为单个 GPU 加速器上的 CU 数目是固定的,也就是说,同时并行执行的工作组数目是固定的,所以,随着工作组数目的增加,其调度开销也会增加.为了减少线程数目,通常增加每个线程的任务数量.当每个 GPU 加速器上的 CU 单元同时运行的工作组数目能够充分隐藏通信开销时,继续增加更多的工作组不仅不会提高效率,反而会带来额外的调度开销.所以,为了降低调度开销,ParaC 利用向量化并行减少工作组的数量.

2.3.4 树形归约优化

根据算法 1 中定义的优化模型,ParaC 编译环境会对全局归约运算进行树形归约优化.例如,图 9 中的 *parac_matrix_sum* 算子,该优化分成两步进行:(1) 工作组内归约:工作项在局部共享内存空间内递归地进行多次局部归约,直至将工作组内的所有数据归约到一个点,并将该值写到全局地址空间,该归约过程利用工作组内

的 barrier 实现同步操作;(2) 工作组间归约:工作组之间在设备的全局存储空间上进行全局归约,利用原子操作保证数据的一致性.

相较于在全局地址空间上实现所有的归约计算,两步归约优化的好处是:(1) 利用了局部共享内存的延迟优于全局地址空间的特性;(2) 用局部的 barrier 操作替代了全局的原子操作,降低了同步开销;(3) 利用调度单元内的隐式同步机制,能够进一步降低同步开销.但是,在具体实现过程中,需要考虑不同硬件对局部隐式同步支持的不同,例如 AMD GPGPU 是 64 线程内不需要显式同步;NVIDIA GPGPU 是在 32 线程内不需要显式同步.

2.4 代码生成

ParaC 编译器会为 ParaC 程序的每个并行区域生成包含主机端和设备端的标准 OpenCL 程序.其中,主机端程序除了硬件平台查找、上下文创建和程序对象构造外,还包括对应每个并行区域的并发任务组织、主机设备间通信、内存分配管理和核心函数的创建和管理.设备端程序即是 OpenCL 的__kernel 对象,本文称为核心函数.

2.4.1 主机端代码生成

本节主要讨论并发任务的组织、内存分配管理以及主机与设备间的通信.

1) 并发任务组织

算法 2 分析了获取由迭代器头部信息或者独立算子高层语义所确定的迭代空间的维度和大小,即为该核心函数全局任务集合的初始维度和大小.在完成算法 1 确定的优化变换后,根据相应的优化集合更新全局任务集合的最终维度和大小,例如,向量化并行优化会减少任务的数目.工作组维度一般与全局任务集合的维度一致,其大小可由编译器自动确定或者采用默认值,这些值需满足硬件平台的约束.例如,AMD GPGPU 要求局部工作大小是 64 的整数倍但上限是 256.通过调整局部工作组的大小能够影响程序的性能,ParaC 编译器会通过比较不同配置时程序的性能来搜索出较优的任务组织形式.

2) 存储空间管理

ParaC 程序中声明的 *parac_matrix* 和 *parac_vector* 变量在输出的 OpenCL 程序中,是指向多维数组的指针.如果变量是单独声明的变量,ParaC 编译器会为其在堆中分配内存;如果是引用,则通过显式数据类型转换使其指向所引用变量指向的内存区域.

另外,如果某个数据对象需要进行自动填充,则在 GPGPU 上分配存储空间时按照填充后的大小进行分配.

3) 主机与设备间通信

主机与设备间是分布式存储结构,所以需要进行显式通信.为了降低通信开销,只传输必需的数据:(a) 在主机端定值或者初始化而在设备端被引用的变量,在核心函数执行之前从主机端复制到设备端;(b) 在设备端定值但又被该核心函数的后续主机端引用的变量,在主机端执行该访存语句之前将数据读到主机端.

4) 矩阵转置

矩阵转置操作在主机端实现,由编译器自动实现相关的存储管理和数据移动.但当矩阵规模较小且引用次数较少时,为了避免转置操作带来的额外内存复制开销,通过交换数据访问下标表达式来加以实现.

2.4.2 设备端代码生成

1) 向量化并行代码的生成

数据类型提升:向量化并行代码会同时读取或者操作 $M(M>1)$个操作数.对于原程序中的标量数据类型需要提升为向量数据类型,其宽度是向量化并行的宽度.对于向量或者矩阵类型的变量,需要将其提升为更高维度的矩阵,提升后的矩阵最低维度的宽度为向量化并行的宽度.

操作符的变换:对于 C 语言中的标量加、减、乘和除等运算,OpenCL 支持向量操作,所以只需要将操作数转换为对应的操作的操作数即可.对于 ParaC 中定义的操作符,需要根据操作符定义生成对应的多个操作.

2) 边界数据访问

当对某个变量的访问存在边界数据规则时,按照对边界数据的访问模式的不同,分别生成相应的越界值计算表达式和计算代码块,避免了为每个数据访问点插入判断是否为越界的分支条件.如果能够静态得到需要边界数据的计算任务及其边界数据访问信息的情况,则直接生成针对该计算代码块的控制条件;否则,为越界访问

的每种形式生成相应的控制条件表达式和计算代码块.

3) 向量化访存

向量化访存能够提高访存带宽利用率,同时能够为 OpenCL 编译器提供更多的优化机会.ParaC 编译器的自动填充技术已经在代码变换阶段保证图像的每行数据都是对齐的,因此,当 PE 需要访问块数据时,将尽量多的数据通过向量化读出或者写入.另外,当对多个相邻的地址写入数据时,也可以将写操作移到最后进行,从而生成向量写操作.

4) 高效指令

OpenCL 提供了很多支持高效操作的指令,例如 mad24,max 和 min 等指令.利用窥孔优化尽可能地将源程序中的相应操作转换成这些高效指令,能够显著提高核心的执行性能.

5) 分支控制流优化

GPGPU 平台的执行模式是 SIMD,同一个组内的 PE 同步执行核心内的所有指令,所以,多个条件分支内同时存在的相同计算、访存会严重影响程序的性能.所以,ParaC 编译器会将不同分支间的公共访存和计算移到控制流的前面来执行,在该过程中,通过定值引用分析、活跃变量分析和定值到达分析来保证变换结果的正确性.

3 实验结果与分析

本节对比了部分图像处理算法的 ParaC 版本和手工优化实现的 OpenCL 版本的执行效率和代码行数,以此来评估分析 ParaC 的性能与产能.另外,对比了部分图像处理算法的 ParaC 版本和 Halide[5]版本的执行效率和代码行数,从而评估分析 ParaC 相对于其他已有工作的性能和产能,其中,Halide 版本的调度策略都是其作者在论文中采用的最佳调度策略,并得到了论文作者的认可.

实验平台包括一台 X86 服务器和两台带有 GPU 加速器的异构服务器.前者是由 Intel Xeon CPU E7-4807 构成的 4 路 48(启动超线程)线程服务器,单核峰值性能为 14.96GFlops.其中一台 GPU 服务器是由 AMD Radeon W8000 GPU 加速器构成的异构服务器,该 GPU 加速器共有 28 个 Compute Unit,1 792 个核,每个核的工作主频是 0.88GHz,整个 GPGPU 卡的峰值性能为 3.13TFlops,最大带宽是 176GB/s.另外,GPU 服务器是由 NVIDIA Tesla K40m 加速器构成的服务器,共有 2 880 个处理器核,每个核的工作主频是 745MHz,整个 GPGPU 卡的单精度峰值性能为 4.29Tflops.

3.1 应用程序介绍

3.1.1 拉普拉斯金字塔算法

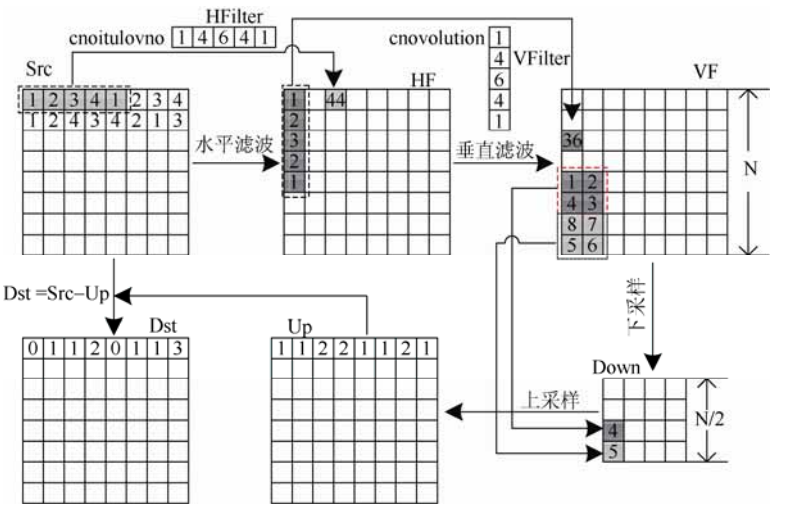

Fig.10 The work flow of Laplacian algorithm

图 10 单层上的拉普斯金塔算法

拉普拉斯金子塔算法可以用于图像的压缩、修复、合成和增强等应用中.拉普拉斯金子塔算法在运算过程中,会生成多个不同大小的层,但在每层都执行相同的计算步骤,最后将各层的计算结果进行合并.为了简化问题,本文所说的拉普拉斯金字塔算法都是指单层上的计算过程.

图 10 给出了拉普拉斯金字塔算法每层上的主要计算过程,共分为 5 步,包括:水平滤波、垂直滤波、下采样、上采样和求差值.

Src 是每层上的输入图像,

HF 是水平滤波后的结果,VF 是垂直滤波后的图像,Down 是下采样后的图像,Up 是上采样计算后的图像,Dst 是完成差值计算后的图像,*Src*(*itx*,*ity*)表示图像 Src 中横坐标为 *ity*,纵坐标为 *itx* 的像素点.

1) 水平滤波:对于目标图像 HF 中的每个像素点,需要用到源图像 Src 上以对应像素点为中心的长度为 5 的一个向量 *SrcHVect*,然后再计算出 *SrcHVect* 与滤波算子 *HFilter* 的卷积值 *HVal*,该卷积值即为图像 HF 中该像素点的值.

2) 垂直滤波:对于目标图像 VF 中的每个像素点,需要用到图像 HF 中以对应像素点为中心的长度为 5 的一个向量 *HFVVect*,然后求 *HFVVect* 与滤波算子 *VFilter* 的卷积值 *VVal*,即为图像 VF 中该像素点的值.

3) 下采样:将图像 VF 的长宽缩小到源图像的 1/2,在计算目标图像的过程中,目标图像中对应像素点 *Down*(*itx*,*ity*)的值分别是源图像中坐标为 *VF*(2×*itx*,2×*ity*)的值.

4) 上采样:将图像 Down 的长宽分别扩大到原来大小的 2 倍,这时候得到的目标图像 Up 的长宽与源图像 Src 一致.

5) 求差值:用源图像 Src 中每个坐标对应的值减去图像 Up 中对应坐标的值,作为目标图像在该坐标上的值.

在拉普拉斯金子塔算法中,存在点操作和局部块操作两种类型,其中,步骤 1~步骤 4 是局部块操作,需要进行边界处理,步骤 5 是点操作.

3.1.2 图像锐化算法

如图 11 所示,图像锐化算法主要分为 4 个步骤:下采样、上采样、图像差值和锐化操作,其中,锐化操作本身又分为 3 个子步骤.

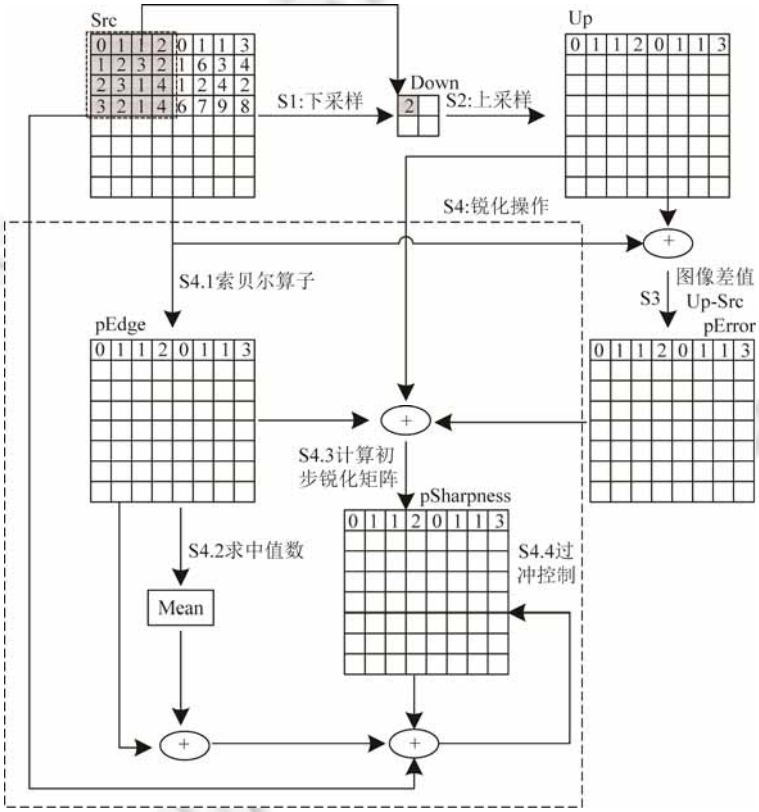

Fig.11 The work flow of sharpness

图 11 图像锐化算法的流程图

1) 下采样:目的是为了降低图像的大小,目标图像 Down 的长宽分别是源图像 Src 的 1/4.例如,图像 Down

中左上角灰色像素点的值是源图像 Src 中左上角相邻的 16 个灰色点的均值(按照四舍五入原则取整数).

2) 上采样:将下采样后的图像 Down 再通过上采样恢复到源图像大小,该过程的输出图像称为 Up.上采样在实现过程中分为 3 个部分,首先是图像 Up 的行边界;其次是计算列边界;最后是计算图像内部的像素点.

3) 图像差值:计算图像 Up 与源图像 Src 之间的差值矩阵 *pError*,该差值矩阵将会用于后续锐化操作中的索贝尔算子.

4) 锐化操作:该计算过程是包含多个子算法过程的复杂操作序列,包括索贝尔算子、求中值数操作、计算初步锐化矩阵和过冲控制操作.

a) 索贝尔算子:用于计算源图像矩阵亮度值的一阶梯度,完整的索贝尔算子包括两个步骤,分别是计算边界和图像内部像素点,索贝尔算子本质上是在源图像进行窗口为 3×3 大小的卷积计算,该算子的输出是 *pEdge* 图像矩阵.

b) 求中值数操作:其输入为 *pEdge* 矩阵,该计算过程就是计算图像上所有像素点的和,然后再求其平均值,主要是归约计算.

c) 计算初步锐化矩阵:利用步骤 b)计算到的中值数再结合其他用户参数,计算出图像亮度的强度值.然后,再用该强度值调整 *pEdge* 矩阵值来控制最终锐化后图像的锐化等级.最后,根据 *pEdge* 矩阵、*pError* 矩阵和上采样后的矩阵 *Up* 进行计算,得到初步的锐化后矩阵.

d) 过冲控制操作:该操作用来去掉不需要的效果,例如放大后的噪声.该过程以初步锐化后的矩阵作为输入,以最终计算得到锐化后的矩阵 *pSharpness* 作为输出.

3.1.3 图像模糊算法

图像模糊算法主要分为两个步骤,分别在水平和垂直方向上进行连续的 stencil 计算,在计算过程中分别用形状为 3×1 和 1×3 的核心在水平方向和垂直方向上进行计算.

3.1.4 反锐化掩膜算法

如图 12 所示,反锐化掩膜算法共分为 5 个步骤,分别是生成灰色图、水平方向上的模糊处理、垂直方向上的模糊处理、图像差值和图像锐化操作.

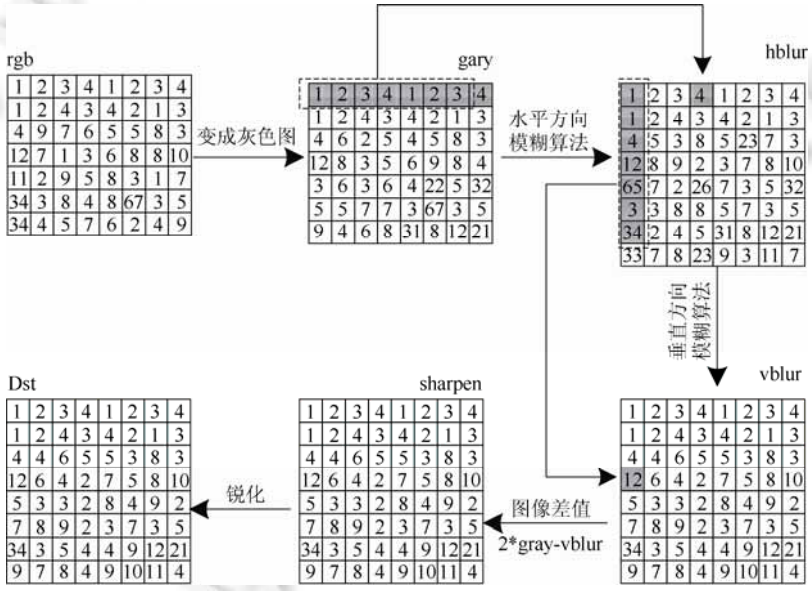

Fig.12 The workflow of unsharp mask algorithm

图 12 反锐化掩膜算法流程图

1) 获得灰色图:将原图像从 RGB 格式转换为灰色图格式,并在边界按照重复图像边缘模式对原图像进行

填充处理,获得灰色图像 gray.

2) 水平方向模糊处理:对于图像 gray,对每行上的像素进行长度为 7 的 stencil 计算,以求得模糊后图像 hblur.

3) 垂直方向模糊处理:对于图像 hblur,对每列上的像素进行长度为 7 的 stencil 计算,以求得模糊后图像 vblur.

4) 计算图像差值:对于模糊处理后的图像 vblur,与原图像进行差值计算得到图像 sharpen,该过程过滤掉低频部分,得到原图像的高频部分.

5) 图像锐化操作:利用图像 sharpen 和 gray 计算出修正因子 ratio,然后再将该修正因子与原图像相乘得到锐化后的图像 Dst.

3.2 应用程序优化分析

点算法:拉普拉斯金字塔算法中的步骤 5 和图像锐化算法中的步骤 3 都属于点算法.每个并行任务只需要源图像的单个像素点,同时,相邻并行任务间不存在数据重用,所以,ParaC 编译器未对这类算法进行优化.

局部块算法:滤波算法、采样算法和索贝尔算子都属于局部块算法.首先,单个任务需要访问多个数据,因此,在任务内采用向量化访存可能会提升访存效率.对于上采样或者下采样算法,存在二维数据重用,所以,并发任务可以在两个维度上同时读取数据,此时,为了保证每行的数据对齐,需要进行自动填充优化以保证每行上的数据都是对齐的.图 13 给出了自动填充优化以及向量化访存带来的程序加速,获得了相对于优化前 2.6%~57.2%的性能收益.

其次,不同并发任务间存在数据重用,向量化并行能够减少所有任务的总访存数量,从而降低带宽需求.图 14 展示了向量化并行对程序的加速情况,获得了相对于优化前程序执行时间 9.5%~22.3%的性能收益.另外,在索贝尔算子中,编译器还进行了循环展开优化,该优化相对于源程序能够减少 24.6%的时间开销.

全局算法:如图 11 所示,图像锐化算法步骤 S4.2 所表示的求中值数操作属于全局算法,在该算法中,ParaC 编译器在代码生成过程中采用了树形归约优化,增加了程序的并行性,同时在代码生成过程中,利用硬件平台特征消除部分同步操作.

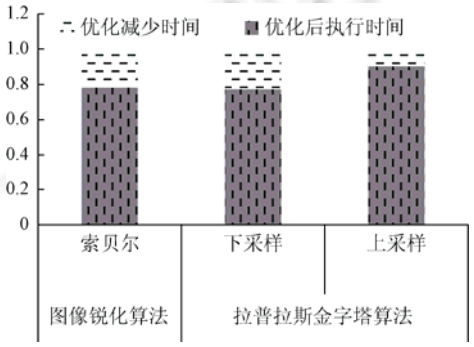

Fig.13 The performance benefit of vectorized access after automatic padding

图 13 自动填充后向量化访存优化获取的性能收益

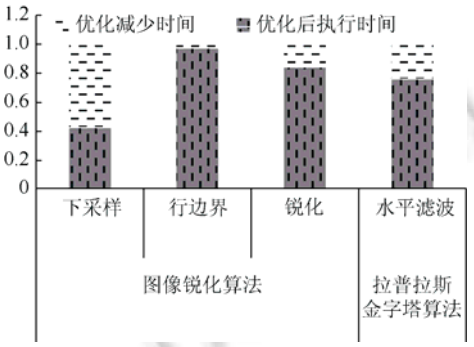

Fig.14 The performance benefit of vector parallelization

图 14 向量并行化优化后的性能收益

3.3 性能与产能分析

3.3.1 ParaC 与人工优化的效率对比

图 15 所示的实验结果表明,ParaC 编译器自动生成的优化程序 OpenCL 程序在不同的输入规模上都具有较好的扩展性.对于图像锐化中的下采样算法模块,ParaC 版本相对于专家手工优化版本在输入规模大于 1 024 时,加速比达到了 18 倍以上,通过分析两个版本的 OpenCL 核心函数发现,手工优化版本使用了 *ALLOC_HOST_PTR* 方式为变量 *pad_yPlane0Buffer* 分配内存空间,导致核心函数访问该变量的开销有所增加.当将其修改为 *CL_MEM_READ_WRITE* 模式时,ParaC 的加速比分别为在输入为 1024 和 2048 时的 1.086 和 2.535.

在优化拉普拉斯算法中的下采样垂直滤波模块时,ParaC 编译器通过搜索比较不同优化方案的性能,最终

采用了水平方向上的向量化并行优化方案,相较于人工优化版本采用的垂直方向上的向量化并行,前者具有更好的访存效率,而且随着输入规模的扩大,其性能优势更加明显,最高获得了相对于后者 1.76 倍的加速比.

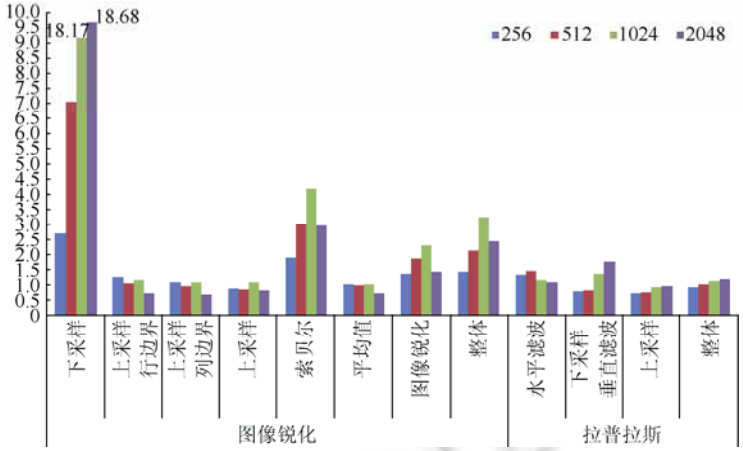

Fig.15 The scalability under different input size of ParaC

图 15 分析 ParaC 算法在不同规模输入下的性能

图 15 中横轴表示不同的算法模块及整体执行时间,纵坐标轴表示各算法核心函数的 ParaC 版本相对于专家优化版本的加速比(专家优化版本的执行时间/ParaC 版本的执行时间),其中,整体执行时间是指两个程序的所有算法模块核心函数执行时间之和.作为基准版本的 OpenCL 程序均由专家手工优化获得,其中图像锐化算法手工优化版本来自于公开发表的文献[13],拉普拉斯算法同样由文献[13]的作者通过手工优化获得,且两者的性能作为合作项目的输出成果都得到某业内领先通信与移动设备制造公司的认可.

对于图像锐化算法中的上采样行边界、上采样列边界和拉普拉斯算法中的上采样算法中存在很多的控制流分支,手工优化版本对这些分支作了更细的优化,例如通过合并分支条件消除不必要的分支,所以性能略好于 ParaC 自动生成版本.

通过搜索算法优化任务的组织形式,ParaC 版本的代码在拉普拉斯的水平滤波、上采样和图像锐化算法中的上采样列边界模块有较大的性能收益,相对于默认的任务组织形式,优化后的版本分别获取 8.51%、4.34%和 20.24%的加速比.

从代码行数角度来评估 ParaC 的产能,表 1 中的数据是拉普拉斯金子塔算法和图像锐化算法的 ParaC 版本与专家优化版本在 GPGPU 平台上的代码行数,从中看到,专家优化版本的代码行数是 ParaC 版本的 3.44 倍~82 倍.这是因为,ParaC 提供了针对领域特征的算子运算符,例如 *parac_matrix_sum* 运算,手工优化版本的代码行数会是 ParaC 的近百倍.另外,ParaC 编程环境能够自动制定程序的优化策略、完成代码变换和输出标准的 OpenCL 程序,所以相较于手工开发优化的 OpenCL 程序,ParaC 能够显著提高用户的开发效率.

Table 1 Comparing the number of code lines between ParaC and OpenCL on Laplacian and sharpness

表 1 比较相应算法模块的 ParaC 版本与专家优化版本的代码行数

	拉普拉斯算法			图像锐化算法						
	水平滤波	下采样与垂直滤波	上采样	下采样	上采样列边界	上采样行边界	上采样	索贝尔	平均值	锐化
专家优化	106	199	110	75	68	72	73	93	82	149
ParaC	16	11	32	5	28	15	5	8	1	34
专家优化/ParaC	6.63	18.09	3.44	15	2.52	4.8	14.6	11.63	82	4.38

为了分析 ParaC 优化算法在不同厂商硬件平台上的适用性,在 NVIDIA GPU 平台上评估了 ParaC 版本和手工优化版本的性能是,实验数据如图 16 所示.从实验结果来看,在部分热点函数上手工优化版本的性能优于 ParaC 版本;但是对于整体性能而言,在多数情况下,ParaC 版本略优于手工优化版本.

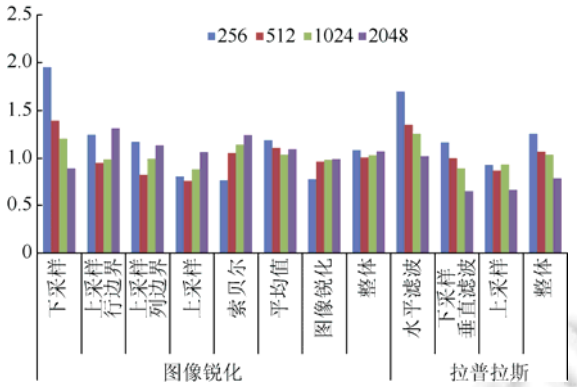

Fig.16 Comparing the performance between ParaC and manual optimization version on NVIDIA K40m GPU platform

图 16 比较 ParaC 版本和手工优化版本在英伟达 GPU 平台上的性能

图 16 中,横轴表示不同的算法模块及整体执行时间,纵坐标轴表示各算法核心函数的 ParaC 版本相对于专家优化版本的加速比(专家优化版本的执行时间/ParaC 版本的执行时间),其中,整体执行时间是指两个程序的所有算法模块核心函数执行时间之和.

3.3.2 ParaC 与 Halide 系统的对比

从代码行数来评估对比 ParaC 与 Halide 的产能,见表 2,其中,OpenCL 版本为由 ParaC 编译器自动生成的 OpenCL 代码,ParaC 版本只包含其核心算法代码行数,Halide 版本代码除核心算法外还包含其对应的调度策略代码.ParaC 和 Halide 都是针对图像领域的高层抽象语言,所以在表达核心算法本身的能力方面,两者非常接近,在反锐化掩膜应用程序中,Halide 版本由于需要设置较为复杂的调度策略,所以其代码行数略高于 ParaC.

Table 2 Comparing the number of code lines between ParaC and Halide, OpenCL on blur and unsharp mask applications

表 2 比较相应算法模块的 ParaC 版本与 Halide 版本、OpenCL 版本的代码行数

	图像模糊应用程序			反锐化掩膜应用程序	
	水平 stencil	垂直 stencil	水平模糊算法	垂直模糊算法	图像锐化
OpenCL	149	206	157	146	166
ParaC	3	3	3	3	8
Halide	3	3	3	3	9
Halide/ParaC	1	1	1	1	1.125

图 17 所示的实验结果表明,在当前的测试用例中,ParaC 获得了相对于 Halide 较优的性能.在图像模糊算法中,Halide 采用了两种优化调度设置,第 2 种调度策略是由 Halide 作者建议的最佳优化方案,是在第 1 种方案的基础上做了内核函数合并,最终只有 1 个内核函数.实验结果表明,ParaC 相对于第 1 种调度策略的 Halide 版本有大于 1.49 倍的加速比,其原因是 ParaC 除了采用与 Halide 相似的优化方案,还针对程序具有数据重用的特征,根据优化模型完成了对内核函数的向量化并行,所以性能优于 Halide.Halide 的第 2 个版本通过内核函数合并提高了性能,但是 ParaC 仍然具有 1.22 倍的加速比.在反锐化掩膜算法中,Halide 调度策略进行了激进的内核函数合并优化,将所有核心计算合并成统一的内核,该优化能够显著减少全局存储空间访问次数,其性能优于 ParaC 版本.综合两个应用程序来看,ParaC

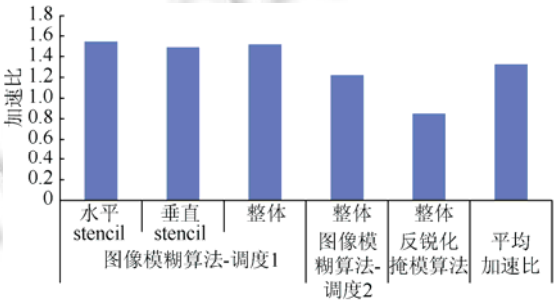

Fig.17 The speedup of ParaC to Halide

图 17 ParaC 版本相对于 Halide 的加速比

相对于 Halide 系统,在当前的测试用例下获得了较优的性能.

4 相关工作

通用编程语言:CUDA[2]和 OpenCL[1]都是支持 GPGPU 加速器平台的通用编程语言,前者只支持英伟达公司的硬件平台,后者是开放的具有平台可移植性的异构平台编程语言规范.CUDA 和 OpenCL 都属于底层编程语言,支持用户针对具体的硬件特征进行编程,其上的优化变换也完全由程序员负责,所以具有很高的编程复杂度且门槛较高.OpenACC[4]和 OpenMP[3]都是支持 GPGPU 平台的基于 pragma 的通用编程语言,能够支持 GPGPU 和 SIMD 等加速部件,两者都能够根据用户的制导自动完成相关的优化变换和代码生成,减轻了程序员的编程负担,但是仍然依赖于程序员给出具体的优化策略.

领域编程语言:领域编程语言包含了领域应用的高层抽象特征,使应用算法的描述比较简洁,同时能够隐藏底层硬件特征,同一份代码可以运行在不同的异构平台.因此,一些针对图像算法的领域编程语言被提出来用于降低 GPGPU 等异构加速平台的编程复杂度.Gordon 等人[14]提出一个支持 CUDA 平台的针对视觉特效的领域编程模型,该模型利用程序员提供的 Indexer Metadata 信息在编译时和运行时进行优化以获取尽可能高的性能.Howes[15]提出了支持 Cell 平台的图像领域编程方法,该方法利用程序员提供的数据访问信息和迭代执行顺序自动地完成数据访问的优化实现.Halide[5]从计算特点的角度将图像处理算法看成是流计算和 stencil 计算的综合,任务图像处理应用程序的优化过程需要处理并行度、局部性和冗余计算之间的冲突,是一个复杂困难的过程.为了简化编程,Halide 提出了将算法描述和调度分离的编程框架,用户给出调度空间的描述,编译器通过机器学习方法自动地搜索出最佳的实现方案.但是,Halide 在优化过程中未考虑领域算法特点,而且将边界处理当成独立的函数作用在每个像素点上,会带来额外的开销.HIPA^{cc}[8]根据图像处理领域的算法特征基于 C++的类定义了一组 DSL 组件,用来描述图像的存储类型、数据访问操作、边界处理、滑动窗口以及操作符,通过这些组件,程序员能够非常简洁地写出基于 HIPA^{cc} 的图像处理程序.但是,HIPA^{cc} 的边界处理不能描述多个点计算一个的过程,例如 Imag[−1][0]=(Imag[0][0]+Imag[1][0])的情况.另外,HIPA^{cc} 在进行高层抽象时,只从算法角度进行考虑,未考虑对编译优化的影响.

流程序的编译优化:在图像的实际处理过程中,同一个图像会经过不同的算法处理阶段,所以图像处理程序属于流式应用程序的一种.StreamIT[14,16]只考虑了 1D 的流处理,而在图像处理过程中,通常是至少为 2D 的流.另外,在实现过程中未针对冗余计算进行优化.Li[17]利用编译指导的方式进行共享内存优化.Li[18]设计了两层任务调度来实现非规则算法在 GPU 加速器平台上的优化.

5 结 论

本文中,我们引入了面向图像处理的领域编程语言 ParaC,并设计和实现了支持 ParaC 的编译器环境,该编译器利用先验知识驱动的编译优化机制,结合软件的程序特征和硬件特征,生成高性能的 OpenCL 程序.ParaC 语言提供了对典型图像操作的高层简化编程接口,例如 stencil 计算、滤波操作和归约操作等;对于其他一般性的算法,ParaC 也提供了迭代器等语言扩展支持程序开发.

在本文所评估应用程序上的实验结果表明,在性能方面,ParaC 编译器自动生成 OpenCL 程序,在性能上超过了由专家手工优化得到的对应程序,与 Halide 系统相比,在总体平均性能上也有一定的优势.同时,在不同输入规模上的实验结果说明,ParaC 编译器自动生成的 GPGPU 程序具有可扩展性.另外,通过对比 ParaC 程序和手工优化版本程序、Halide 版本程序的代码行数和程序员承担的开发责任,也表明 ParaC 具有较高的产能.

ParaC 的优点是通过在语言扩展中引入的高层语言抽象和基于先验知识驱动的优化模型能够在生成高性能代码的同时提高异构平台上的开发效率.其不足和未来需要进一步开展的工作是,进一步加强编译优化能力,例如支持更好的核心函数合并优化;在更多的应用程序和硬件平台上评估分析 ParaC 的能力.

References:

[1] Khronos OpenCL Working Group. The OpenCL specification. Version 1.2, 2012.

[2] NVIDIA. CUDA toolkit documentation. Version 8.0.61, 2017.

[3] OpenMP ARB. OpenMP application program interface. Version 4.0, 2013.

[4] OpenACC.Org. The OpenACC application programming interface. Version 2.5, 2015.

[5] Jonathan RK, Connelly B, Andrew A, Sylvain P, Frédo D, Saman A. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2013). New York: ACM, 2013. 519−530. http://dx.doi.org/10.1145/2491956.2462176

[6] Jonathan RK, Andrew A, Sylvain P, Marc L, Saman A, Frédo D. Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. on Graphics (TOG), 2012,31(4):1−12. [doi: 10.1145/2185520.2185528]

[7] Membarth R, Hannig F, Teich J, Körner M, Eckert W. Generating device-specific GPU code for local operators in medical imaging. In: Proc. of the 26th IEEE Int'l Conf. on Parallel & Distributed Processing Symposium (IPDPS). Shanghai, 2012. 569−581. [doi: 10.1109/IPDPS.2012.59]

[8] Membarth R, Reiche O, Hannig F, Teich J, Körner M, Eckert W. HIPA^{cc}: A domain-specific language and compiler for image processing. IEEE Trans. on Parallel and Distributed Systems, 2016,27(1):210−224. [doi: 10.1109/TPDS.2015.2394802]

[9] Bankman IN. Handbook of Medical Image Processing and Analysis. 2th ed., New York: Academic Press, 2008. 3−18.

[10] Russ JC. The Image Processing Handbook. 6th ed., Boca Raton: CRC Press, 2006. 288−355.

[11] Klette R, Zamperoni P. Handbook of Image Processing Operators, Vol. 1. Hoboken: Wiley, 1996. 38−42.

[12] Clang: A C language family frontend for LLVM. http://clang.llvm.org

[13] Fan M, Jia H, Zhang Y, An X, Cao T. Optimizing image sharpening algorithm on GPU. In: Proc. of the 44th Int'l Conf. on Parallel Processing (ICPP). Beijing, 2015. 230−239. [doi: 10.1109/ICPP.2015.32]

[14] Gordon MI, Thies W, Karczmarek M, Lin J, Meli AS, Lamb AA, Leger C, Wong J, Hoffmann H, Maze D, Amarasinghe S. A stream compiler for communication-exposed architectures. ACM SIGPLAN Notices, 2002,37(10):291−303. http://dx.doi.org/10.1145/605432.605428

[15] Howes L, Lokhmotov A, Donaldson A, Kelly PHJ. Deriving efficient data movement from decoupled access/execute specifications. In: Proc. of the 4th Int'l Conf. High-Perform. Embedded Archit. Compilers. 2009. 168−182. [doi: 10.1007/978-3-540-92990-1_14]

[16] Thies W, Karczmarek M, Amarasinghe S. StreamIt: A language for streaming applications. In: Compiler Construction. Berlin, Heidelberg: Springer-Verlag, 2002. [doi: 10.1007/3-540-45937-5_14]

[17] Li J, Liu L, Wu Y, Liu XH, Gao Y, Feng XB, Wu CY. Pragma directed shared memory centric optimizations on GPUs. Journal of Computer Science and Technology, 2016,31:235. [doi: 10.1007/s11390-016-1624-8]

[18] Li J, Liu L, Wu Y, Feng XB, Wu CY. Two-Level task scheduling for irregular applications on GPU platform. Int'l Journal of Parallel Programming, 2015. [doi: 10.1007/s10766-015-0387-0]

卢兴敬(1983－),男,山东临沂人,博士生,工程师,CCF 专业会员,主要研究领域为并行编程,程序并行优化及分析.

刘雷(1980－),男,博士,助理研究员,CCF 专业会员,主要研究领域为编程语言,编译优化.

贾海鹏(1983－),男,博士,助理研究员,CCF 专业会员,主要研究领域为高性能计算,面向多核/众核的编程方法与优化技术.

冯晓兵(1969－),男,博士,研究员,博士生导师,CCF 杰出会员,主要研究领域为编程模型,编译优化.

武成岗(1969－),男,博士,正高级工程师,博士生导师,CCF 高级会员,主要研究领域为计算机系统结构,编译技术.