计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(5):351−357 [doi: 10.15888/j.cnki.csa.008449]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

一种 Fortran 到 CUDA C 的转换方法①

刘颖辉[1,2], 迟学斌[1,2], 姜金荣[1,2], 张　峰[1,2]

[1](中国科学院 计算机网络信息中心, 北京 100190)
[2](中国科学院大学, 北京 100049)
通信作者: 迟学斌, E-mail: chi@sccas.cn

摘　要: 基于 GPU 的异构计算逐渐成为主流计算方法, 但限于科学计算编程的历史发展, 大量的数值计算软件仍以 Fortran 语言实现. 为了提高计算速度, 大量的软件需要移植为 CUDA C, 但人工实现程序移植是一项浩繁的工程. 若能实现从 Fortran 到 CUDA C 的自动转换, 可以极大的提高程序开发效率. 本文设计了将 Fortran 转换为 CUDA C 的算法, 并基于正则表达式和 shell 脚本实现了该算法, 编写测试用例进行了验证. 实验表明, 该算法可靠稳定兼容性好, 在大型程序的移植过程中, 能够自动筛选并建立变量信息表, 生成 CUDA 相关操作函数, 且结果代码可读性较好, 转化正确率达 80% 以上, 有效减少了移植的工作量.

关键词: Fortran; CUDA C; 并行计算; 正则表达式

引用格式: 刘颖辉,迟学斌,姜金荣,张峰.一种 Fortran 到 CUDA C 的转换方法.计算机系统应用,2022,31(5):351–357. http://www.c-s-a.org.cn/1003-3254/8449.html

Conversion Method from Fortran to CUDA C

LIU Ying-Hui[1,2], CHI Xue-Bin[1,2], JIANG Jin-Rong[1,2], ZHANG Feng[1,2]

[1](Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Graphic processing unit (GPU)-based heterogeneous computing has gradually become the mainstream computing method. Nevertheless, due to the limited historical development of scientific computing programming, a lot of numerical computing software is still implemented in Fortran. In terms of increasing the computing speed, a large amount of software needs to be transplanted onto compute unified device architecture (CUDA) C. However, it would be a complicated and massive project to manually implement the program transplant. If automatic conversion from Fortran to CUDA C can be achieved, the efficiency of program development would be greatly improved. This study designs an algorithm converting Fortran to CUDA C, implements the algorithm through regular expressions and shell scripts, and verifies it by programming test cases. Experimental results show that this tool is reliable, stable, and compatible. In the transplant process of large programs, it can automatically filter and establish variable information tables and generate CUDA-related operation functions. The resulting code possesses good readability, and the conversion accuracy is more than 80%. The workload of the transplant is effectively reduced.

Key words: Fortran; compute unified device architecture (CUDA) C; parallel computing; regular expression

1　背景介绍

随着计算机科学的发展, 不同特色的程序语言不断涌现, 由于 Fortran 更接近数学语言, 执行效率较高, 有着较低的开发成本, 因此在数值计算、科学和工程技术领域, Fortran 依旧占据主流地位; 大量的数值计算工程软件, 开发语言同样仍以 Fortran 为主. 但在气

① 基金项目: 国家重点研发计划 (2016YFB0200800); 国家自然科学重点基金 (41931183); 国家重大科技基础设施项目“地球系统数值模拟装置”
收稿时间: 2021-07-09; 修改时间: 2021-08-04; 采用时间: 2021-08-17; csa 在线出版时间: 2022-04-11

候、流体力学等多个领域的模拟过程中, 随着计算规模的扩大以及对于计算精度的更高要求, 程序计算所需的时间也在不断延长, 部分大型程序需要几天甚至几十天才能完成计算. 传统的优化方式是将 Fortran 程序与 MPI 进行结合, 通过实现粗粒度的并行来提高计算效率. 近些年来, 随着高性能计算机的迅速发展, 通过 GPU 和 CPU 的异构计算平台[1] 来实现细粒度的并行这一课题成为了学术界和工业界的研究热点. 基于 Fortran 所开发的科学计算软件及程序, 通过异构并行来实现性能的巨大提升, 在工程上有着急切的需求. 在异构计算平台上, 以用于 GPU 编程的 CUDA C 语法为主流的行业标准, AMD 平台使用的 ROCm 语法也参照 CUDA C 规范. 但是从 Fortran 到 CUDA C 的移植过程, 不仅要求开发人员对 Fortran 与 CUDA C 均有较高的熟悉度, 而且在复杂的文件和模块中检索相关变量的信息并编写相应 CUDA 函数往往有着巨大的工程量, 如果完全通过人工完成, 不仅效率低下、极易出错, 且后期难以维护和调试. 因此从 Fortran 到 CUDA C 的自动转码工具, 有着极大的需求.

现如今仅有一些将 C 语言转换为 CUDA 的工具, 加拿大多伦多大学的 Han 等人设计了一种基于指令的 CUDA 编程语言 hiCUDA[2], 西安交大的 Li 等人实现了一种源到源自动并行化工具 GPU-S2S[3], 孙香玉提出了一种面向 CUDA 的源到源并行化架构 STS-CUDA[4], 三者都是用于实现 C 语言到 CUDA 的自动转换, 但对于大型计算程序来说, 需要人工检索程序中大量变量的相关信息, 再对内存的分配拷贝释放等诸多操作插入编译指导语句. 通过这种方式来进行转化所需的工作量同样是十分巨大的. 荷兰埃因霍芬理工大学的 Nugteren 等人设计了自动转换编译器 Bones[5], Bones 是一个概念性的验证而非工业性的编译器, 其侧重点在于对特定结构的程序进行转换以及自动优化, 对于实际程序中的复杂情况不能很好的支持. 除此之外, 部分学者设计了与 CUDA 转化相关的编译器[6–8], 取得了不错的效果. 而从 Fortran 向 CUDA C 的转化工具, 仅美国国家海洋和大气管理局地球系统研究实验室开发了一种名为 F2C-ACC 的编译器[9], 但作者并未给出具体实现细节以及实际效果.

2 应用技术简介

2.1 正则表达式

正则表达式是对字符串进行逻辑过滤的一种逻辑操作[10]. 将某类字符抽象为一些特定的字符, 通过某些特定字符的组合来描述字符串匹配的模式.

正则表达式具有强大的灵活性与功能性, 仅需非常简便的代码就可以实现复杂的字符串操作. 通过正则表达式, 可以实现如下功能: (1) 匹配. 判断目标字符串是否与描述的模式相匹配; (2) 获取. 从字符串中提取需要的特定信息; (3) 编辑. 对字符串的子集进行切割、替换等操作.

2.2 shell 脚本

shell 脚本是一种为 shell 编写的脚本程序, 能够轻易的处理文件与目录之类的对象. 通过结合正则表达式以及 sed、grep、awk 等命令可以在短时间内完成一个功能强大又好用的脚本, 如下代码实现了获取使用操作符“( )”的变量名的功能:

```
cat filename.F90 |\
grep -oi "[a-z_]\+\w*[ ]*([^()]*)" | grep -vi "float" |\
awk -F "[(]" '{print $1"("}'|\
sed -e "s/[A-Z]/\l&/g" -e "s/=//" -e "s/ //g" |\
sort | uniq > result
```

如果采用 C 语言编写实现上述功能, 可能需要几十行甚至上百行的代码, 由此可见 shell 脚本在字符串处理方面的强大功能.

3 转换算法的设计

3.1 转换难点分析

Fortran 与 CUDA C 在语法规范、内存分配等方面均有较大的差异, 直接从 Fortran 转换为 CUDA C 有着较大的难度. 而 C 语言与 Fortran 无论是从结构方面还是语法的角度都比较相似, 同时在 CUDA C 程序中, 运行在主机端的代码仍然采用 C 语言进行编写, C 语言很好的建立了从 Fortran 到 CUDA C 的桥梁. 因此将 Fortran 程序转换为 CUDA C 的过程分为如下两部分: (1) Fortran 语言到 C 语言的转换; (2) 从 C 语言转化为 CUDA C. 结合 Fortran 与 CUDA C 两种语言的特点, 转换过程需要解决如下问题:

(1) 数组的处理. Fortran 中数组下标可以采用形如 A(index1:index2) 形式来自定义起始下标与终止下标, 若采用默认定义, 则起始下标为 1[11]; 而在 C 语言中所有数组下标均从 0 开始, 且不支持自定义的方式来访问元素. 对于多维数组, Fortran 采用列优先的方式进行存储, 而 C 语言为行优先. 因此如何完成下标映射

将成为转换过程中的一大难点. 对于动态分配的数组, 定义中的维度用“:”来进行, 需要额外检索 allocate 语句来获取对应的数组维度. Fortran 对数组的访问也更加灵活, 如对于维度相同的一维数组 A、B, 则可以直接通过 A(:)=B(:) 来完成对应元素的赋值. 因此对于使用“:”作为维度的相关语句需要进行较为复杂的处理.

(2) 引用外部模块. Fortran90 采用 module 将一系列的数据与函数封装起来, 任何程序都可以通过 use 语句来引用该 module 中的内容; 而在 C 语言中引用其他变量通过 include 或者 extern 关键字来实现. 因此转换过程中, 文件的检索范围不仅限于本模块, 还需要包含引用的所有外部模块. 同时为了将转换后的 CUDA C 程序与原有的 Fortran 程序链接起来, 对变量的名称需要根据编译器进行额外的处理, 因此如何将引用的外部模块转换为同时兼容 C 程序、Fortran 程序的形式将是一大难点.

(3) CUDA 函数的处理. 在异构计算平台上, CPU 用于控制计算过程和实施存储策略, 而具体的计算过程则由 GPU 进行[12]. 而在 GPU 上不能直接访问 CPU 端的内存, 因此 GPU 计算所需的数据以及计算得到的结果必须显式地与 CPU 进行数据的传输. 除此之外, 在 GPU 端执行的代码需针对并行做相应的修改. 因此针对计算过程中的大量变量, 如何自动生成函数实现内存的分配、拷贝以及释放将是转换过程中的一大挑战.

针对以上难点, 本文进行了相关算法设计和实现.

3.2 算法设计与实现

3.2.1 Fortran 语言到 C 语言的转换

在 Fortran 到 C 语言的转换过程中, 一方面需要完成相关语法、关键字等内容的转换, 另一方面对于每个变量, 需要从本文件或外部模块中检索来建立一张变量信息表, 表中需包含文件名称、变量名称、变量类型、维度等转换中必要的信息. 转换流程如图 1 所示.

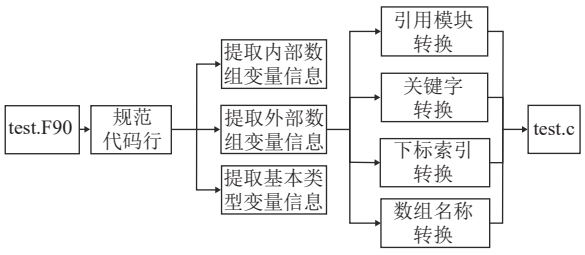

图 1 Fortran 到 C 的转换流程图

(1) 规范代码行

处理源文件中的注释与跨行. 为保证程序的可读性, 源程序的注释不删除, 将“!”所注释的内容均修改为 C 的“//”形式; 删除跨行符“&”并合并多行为一行, 删除文件中的空行.

(2) 提取变量信息

1) 数组型变量

首先检索源文件中的变量定义语句, 获取所有的内部数组变量名称. 根据下标运算符“( )”提取所有可能的数组变量, 去除关键字、函数名以及内部数组变量, 得到所有外部数组类型的变量名称.

通过对源文件进行检索, 截取数组变量的定义或 allocate 语句的相关信息, 建立内部数组变量与相应维度、类型的映射关系. 同理, 根据 use 语句提取源文件引用的 module 名称, 在相应文件中查找并建立外部数组变量信息表.

2) 基本数据类型变量

由于直接从源文件筛选可能的变量名不仅效率低下, 而且准确率较低, 因此从引用的模块中提取出所有的变量定义 (不含数组类型变量), 依次进行过滤, 仅保留在源文件中使用的子集, 从而建立外部变量与相应类型的映射关系.

(3) 引用模块转换

对源文件中引用的每一个模块, 生成相应的头文件, 该文件中对使用到的相关变量进行声明. 对于非数组型变量直接将其转换为 C 语言的形式. 对于数组型变量, 由于 C 程序数组的处理与 Fortran 存在较大差异, 如 C 程序数组维度仅能通过常量定义, 起始下标为 0, 不支持自定义下标等, 因此将其声明为同类型的指针变量, 通过一维数组的方式进行访问.

(4) 结构转换

1) 关键字以及语句的转换

将 Fortran 中的关键字与语句均转换为 C 语言中对应的形式, 包括但不限于表 1 的内容. 注释在 C 程序中无需使用的语句, 最大限度保证程序的可读性.

2) 数组下标索引的转换

对于 Fortran 中定义的数组, 如 real(8) dimension (index1) :: A, 则数组 A 的维度大小为 index1, 由于 C 语言中起始下标默认为 0, 当通过下标来进行元素的访问时需要减去相应的偏移量, 如 A(i) 应转换为 A[i–1]. 更一般的, 对于 Fortran 中自定义下标的数组,

如: real(8) dimension(index1:index2) ::B, 则数组 B 的维度大小为 index2–index1+1, 相应的访问 B(*i*) 应转换为 B[*i*–index1].

对于多维数组, 如 Fortran 中定义数组 real(8), dimension (dimx, dimy) :: D, 由于行列优先的不同, 对应 C 程序的数组定义为: double D[dimy][dimx] , 相应下标访问 D(*i*, *j*) 对应 C 程序中的 D[*j*–1][*i*–1]; 根据上文所述, 将多维数组均视为一维数组进行相关运算, 下标访问需进一步转换为 D[(*j* – 1)*dimy+(*i*–1)].

表 1 Fortran 与 C 关键字及语句对照表

Fortran	C	Fortran	C
use param	#include"param.h"	integer	int
if then	if else	real	double
end if	删除	logical	bool
call fun	fun();	.and. .or. .not.	&&|| !
return	return ;	.gt. .lt. .eq.	><=

3) 数组名称的转换

代码的前后处理在 CPU 上进行, 一般使用原有代码, 这就需要实现 C 和 Fortran 的混合编程, 主要是实现数组或变量的互相访问. 以 ifort 编译器下的文件 param.F90 中定义的外部数组变量 b 为例, 将数组名称 b 转换为 param_mp_b_, 所有外部数组变量均要做相应的替换.

Fortran 到 C 的整体转换算法如算法 1 所示.

算法 1. Fortran 到 C 的转换

```
输入 Fortran 文件
  预处理
  匹配 call 语句, 提取函数名称
  匹配 use 语句, 提取引用模块名称
  for file in 源文件, 引用模块文件
    匹配变量 define 语句、allocate 语句
    存储变量名称、所在文件、类型、维度等信息
  end for
  for var in 变量信息表
    if var in 引用模块文件
    //以 ifort 编译器为例
      替换变量名称: var → filename_mp_var_
    end if
  end for
  转换 Fortran 关键字、语句为 C 形式
  创建头文件
输出 C 文件
```

3.2.2 C 语言到 CUDA C 的转换

CUDA C 语法规定用__global__修饰符所修饰的函数为核函数, 核函数运行在设备端, 而运行在主机端的代码与 C 语言是完全兼容的. 因此从 C 语言到 CUDA C 的转换仅需考虑在设备端运行的部分代码即可, 这部分代码所涉及的数据以及得出的结果均需要在主机与设备内存间显式的进行复制[13], 即需要额外生成对应的拷贝函数. 除此之外, 由于核函数只能在主机端调用, 设备端执行[14], 因此主机端需要显式的进行核函数的调用, 并传递运行时所需参数, 核函数的内容需结合多线程并行的特点做出相应的调整. 转换流程如图 2 所示.

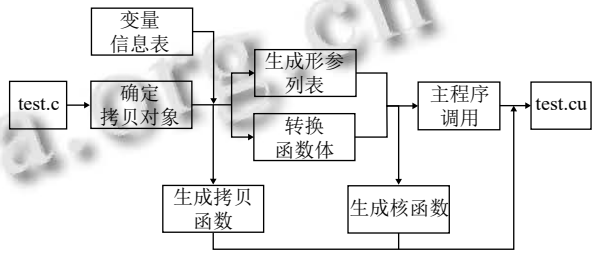

图 2 C 到 CUDA C 的转换流程图

(1) 内存拷贝函数的生成

1) 确定拷贝对象

核函数中参与运算的所有数组变量均需要在设备端显式的进行内存的分配. 通过查找上文建立的变量信息表确定类型与维度, 声明同类型空指针, 分别利用 cudaMalloc、cudaFree 进行 GPU 内存的分配与释放. 如参与运算的某一数组定义为 real, dimension(m, d):: a, 则需生成如下 CUDA C 语句:

```
double param_mp_a_[d][m];//转换后 C 形式的定义
double *d_a=NULL;//同类型指针声明
//GPU 内存的分配
CHECK(cudaMalloc(&d_a, d*m*sizeof(double)));
CHECK(cudaFree(d_a));//GPU 内存的释放
```

其中内存分配与释放语句中的 CHECK 为宏定义, 用于接收 cudaError 并输出提示信息, 保证程序的健壮性.

2) 生成拷贝函数

以所有外部数组变量为全集, 检索源文件中赋值运算符"=", 出现在运算符左侧的所有变量将其从设备端拷贝到主机端, 相应拷贝函数封装为 test_DtoH(), 出现在运算符右侧的变量将其从主机端拷贝到设备端, 相应拷贝函数封装为 test_HtoD(). 如对于 c(j, i)=c(j, i)+a(j, k)*b(k, i), 则应有如下函数被生成:

```
extern "C" void test_DtoH(){//从 GPU 到 CPU 的内存拷贝
```

```
CHECK(cudaMemcpy(param_mp_c_, d_c,
        n*m*sizeof(double), cudaMemcpyDeviceToHost));
}
```

(2) 核函数的转换

1) 核函数范围的确定

一般情况下, 建议用户将核函数部分放在一个单独的 Fortran 文件中, 使用 subroutine 或者 function 来将其封装, 在主程序中通过 call 语句来对其进行调用.

同时, 我们允许手动指定核函数的范围, 在对应的代码块加入如下指导语句:

```
#pragma test kernel begin
  kernel code
#pragma test kernel end
```

二者之间的代码被认为是一个完整的名为 test 的核函数.

2) 形参列表的确立

核函数中所用到的变量均需要作为形参列表的一部分来进行传递, 具体分类如下: ① 对于基本数据类型: 整型、实型、字符型、逻辑型, 直接将其转换为 C 中所对应的类型, 作为参数来进行传递. ② 对于数组型变量, 将其均视为一维数组, 相应类型的指针作为形参来进行传递. ③ parameter 关键字定义的常量采用在头文件中宏定义的方式进行处理, 无需进行参数的传递.

3) 函数体的转换

GPU 上的计算以 kernel 函数为主, 科学计算软件中主要的计算代码是 do 循环计算 (模板计算), 图 3 给出了模板计算的 kernel 函数生成. 首先是插入了线程号计算代码, 然后根据线程号将循环计算任务分配给相应的线程. 由于控制循环的变量被修改, 因此原先由循环变量控制的数组下标也要做出相应的修改. 在 Fortran 程序中由数组下标 i 控制的循环, 在循环展开后 i 的含义变为 CUDA C 程序中的线程号. 多维情况与一维情况类似, 不再赘述.

```
Fortran 程序:
do i=begin,finish,step
! for(i=begin;i<=finish;i+=step)
  A(i) =1
end do

CUDA C 程序:
i=blockIdx.x*blockDim.x+
  threadIdx.x;
if (begin+step*i>= begin &&
    begin-step*i <=finish){
    A[begin+step*i-index1]=-1;
}
```

图 3 循环展开格式

C 到 CUDA C 的整体转换算法如算法 2 所示.

算法 2. C 到 CUDA C 的转换

```
1.  输入 C 文件
2.    for var in 变量信息表
3.      if var in 引用模块文件
4.        声明设备端相应变量, 创建 cudaMalloc 函数
5.      if var match 赋值语句左侧
6.        创建 GPU 到 CPU 的 cudaMemcpy 函数
7.      else if var match 赋值语句右侧
8.        创建 CPU 到 GPU 的 cudaMemcpy 函数
9.      end if
10.       创建 cudaFree 函数
11.   end if
12.  end for
13.  确立形参列表, 生成核函数
14.  转换核函数函数体
15. 输出 CUDA C 文件
```

4 实验验证

本文实验环境如表 2 所示.

表 2 实验环境配置

类型	版本
操作系统	CentOS Linux release 7.4.1708
GPU	NVIDIA Quadro GV100
CPU	Intel Xeon Gold 6148 CPU @ 2.40 GHz
CUDA工具包	V11.0
MPI	Intel MPI 3.1

4.1 矩阵乘法的异构代码自动生成

稠密矩阵的乘法是典型的计算密集型问题, 且具有较好的并行性. 以计算矩阵乘法: $A_{m\times d}B_{d\times n}=C_{m\times n}$为例进行测试, 输入矩阵 A、B 均为随机生成. 编写名为 MatrixMul 的串行矩阵乘法 Fortran 程序, 计算用到的数组以及相关变量定义于外部模块 (源程序位于 param.F90 中). 基于正则表达式与 shell 语言实现本文提出的转换算法并进行验证, 转换所得结果与原 Fortran 程序计算结果相同, 验证了转换方法的正确性.

转换后程序的性能表现如图 4 所示, 其中横坐标是矩阵规模 ($m\times d\times n$), 纵坐标是计算时间. 点状柱是原 MPI 程序在 1 个节点 (2 颗 CPU) 下的计算时间, 条纹柱与灰柱分别对应使用本文算法转换完成和手动实现的 CUDA C 代码在同 1 张 GPU 卡上的计算时间. 从计算结果中可以得出, 由本文算法自动转换得到的 CUDA C 矩阵乘法在性能上与人工编写的 CUDA C 矩阵乘法相当, 较原 MPI 程序平均加速了 1.83 倍. GPU+CPU 的异

构计算较原 CPU 并行能够取得更好的加速效果.

4.2 海洋环流模式 LICOM 的异构代码生成

LICOM 是由中科院大气物理研究所 LASG 国家重点实验室发展的全球海洋环流模式. 它是中国科学院地球系统模式 CAS-ESM 的重要组成部分[15].

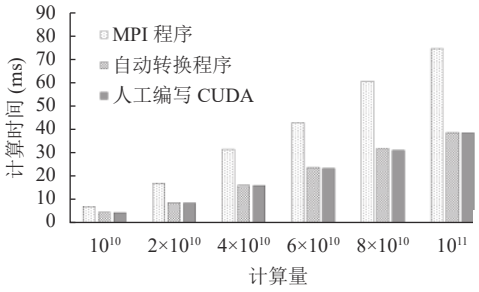

图 4 矩阵乘法性能对比

源程序采用 Fortran 语言编写, 核心计算程序普遍引用了多个模块, 大量的数组变量通过 allocate 语句来分配内存, 以 readyt 为例, 核函数部分计算变量累计 86 个, 其中数组型变量共 52 个, 引用自 14 个模块中数组变量共 40 个, 采用本文算法进行代码自动转换, 生成的文件结构及功能如图 5 所示.

```
c
└─readyt. c ---------- 转换生成的. c 文件, 可用于后期调试
cuda
├─readyt. cu ---------- 转换生成的. cu 文件
├─readyt_cu. h ---------- 新引入的变量定义头文件
├─readyt_DtoH. cu ---------- GPU 到 CPU 的拷贝函数
├─readyt_finalize. cu --------- GPU 内存回收
├─readyt_HtoD. cu ---------- CPU 到 GPU 的拷贝函数
└─ready_init. cu ---------- 变量初始化及 GPU 内存分配
error log
headfile
├─def_undef. h ---------- 源程序引入的头文件
├─block. h ----------
├─ …… ---------- } 根据引入模块转换生成的头文件
└─work_mod. h ----------
otherfile
├─ readyt. F90. allglobal ------- 所有变量名及类型
├─ readyt. F90. call ---------- 调用函数的名称
├─ readyt. F90. varin ---------- 内部数组变量名称
├─ readyt. F90. varlist --------- 数组变量名称
├─ readyt. F90. varmap -------- 外部数组变量与文件名的映射
├─ readyt. F90. varmap.sav------ 外部数组变量与类型维度的映射
└─ readyt. F90. varout ---------- 外部数组变量名称
```

图 5 readyt 转换文件结构及功能说明

实验证明: 对于复杂的大型程序, 本文提出的转换算法仍能够自动检索核函数所需变量, 从对应文件中提取变量的相关信息, 自动生成内存分配、拷贝、释放等一系列的函数, 结合现有的变量信息表以及转化完成的代码进行调试, 手动修改少量代码后即可编译通过, 转化代码的正确率高达 80% 以上, 初步完成了 LICOM3 CUDA C 异构代码的开发.

进一步分析转换算法的输出结果, 探究转换过程中出错的主要原因: (1) 计算中含有冒号的数组索引未能进行相应的展开; (2) readyt 内部定义的部分数组变量在实际的 CUDA C 程序中仍需进行额外的处理. 若针对上述两点预先对 Fortran 程序进行简单的修改, 则转换正确率可达 90% 以上.

经过测试, Fortran 程序中 readyt 函数平均耗时 72.32 s, 转换完成的 CUDA C 程序中该函数平均耗时 1.29 s, 加速了 56.06 倍.

5 结论展望

本文在深入分析 Fortran 语言以及 CUDA C 相关特征的基础上, 使用正则表达式和 shell 语言, 实现了一套逻辑清晰、功能强大的 Fortran 到 CUDA C 的转换方法. 经过测试, 该方法转换正确率高达 80% 以上, 转换后的代码性能与人工编写的 CUDA C 代码相当, 能够有效节省大型程序的移植时间. 本文的实现存在一定的局限性, 对于较复杂的软件代码得到的转换结果仍需开发人员手动进行调试, 且实现的代码还需要进行深度优化.

参考文献

1 Li Y, Dai ZT. Design and implementation of hardware accelerator for recommendation system based on heterogeneous computing platform. Proceedings of the 3rd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019). Dalian: Wuhan Zhicheng Times Culture Development Co. Ltd., 2019. 966–971.

2 Han TD, Abdelrahman TS. hiCUDA: High-level GPGPU programming. IEEE Transactions on Parallel and Distributed Systems, 2011, 22(1): 78–90. [doi: 10.1109/TPDS.2010.62]

3 Li D, Cao HJ, Dong XS, *et al*. GPU-S2S: A compiler for source-to-source translation on GPU. Proceedings of the 3rd International Symposium on Parallel Architectures, Algorithms and Programming. Dalian: IEEE, 2010. 144–148.

4 孙香玉. 面向 CUDA 的循环语句源到源并行化研究 [硕士学位论文]. 兰州: 西北师范大学, 2014.

5 Nugteren C, Corporaal H. Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs. ACM Transactions on Architecture and Code Optimization, 2015, 11(4): 35.

6 Yu CL, Royuela S, Quiñones E. OpenMP to CUDA graphs:

A compiler-based transformation to enhance the programmability of NVIDIA devices. Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems. St. Goar: ACM, 2020. 42–47.

7 Lee S, Min SJ, Eigenmann R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. ACM SIGPLAN Notices, 2009, 44(4): 101–110. [doi: 10.1145/1594835.1504194]

8 Stauber T, Sommerlad P. Parsing CUDA for Transformation to SYCL in an IDE. Proceedings of the International Workshop on OpenCL. Boston: ACM, 2019. 20.

9 Govett MW, Middlecoff J, Henderson T. Running the NIM next-generation weather model on GPUs. Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. Melbourne: IEEE, 2010. 792–796.

10 高阳阳, 徐烈伟, 俞剑, 等. 一种新型动态可重构的正则表达式匹配引擎设计. 复旦学报 (自然科学版), 2019, 58(6): 706–718.

11 段红英. Fortran 程序 CUDA 并行化总结. 物联网技术, 2015, 5(11): 92–93. [doi: 10.3969/j.issn.2095-1302.2015.11.033]

12 Brodtkorb AR, Hagen TR, Sætra ML. Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing, 2013, 73(1): 4–13. [doi: 10.1016/j.jpdc.2012.04.003]

13 韩雪. 面向 CPU_GPU 异构系统的通用计算模型研究 [硕士学位论文]. 沈阳: 东北大学, 2015.

14 王泽寰, 王鹏. GPU 并行计算编程技术介绍. 科研信息化技术与应用, 2013, 4(1): 81–87.

15 王文浩, 姜金荣, 王玉柱, 等. 海洋模式 LICOM 的 MIC 并行优化. 科研信息化技术与应用, 2015, 6(3): 60–67.