Weekly Research Progress Report

Student: 从 兴 **Date:** 2024/1/17 - 2024/2/6

List of accomplishments this week

1. 学习 C 语言依赖库的使用方式，掌握如何在有外库的条件下，编译链接源代码。
2. 学习 Lapack 的相关知识，通过 Cmake 构建了 Lapack 的使用 Demo。
3. 学习 HIP 编程的相关知识，熟悉常见的 HIP 编程（CUDA 编程）方法。
4. 完成在 ITMO 中 ML 课程的五项任务，获得结业证书。

Work summary

1. Lapack 相关

在 Lapack 学习过程中，我认为对我影响比较大的知识有以下几点：

（1）C 语言中外库的链接与使用

获得库的头文件路径以及依赖路径：

```
(base) buaaer@AirM1 ~ % brew list lapack
/opt/homebrew/Cellar/lapack/3.12.0/include/ (5 files) ------> 重要：头文件的路径
/opt/homebrew/Cellar/lapack/3.12.0/lib/libblas.3.12.0.dylib
/opt/homebrew/Cellar/lapack/3.12.0/lib/liblapack.3.12.0.dylib
/opt/homebrew/Cellar/lapack/3.12.0/lib/liblapacke.3.12.0.dylib
/opt/homebrew/Cellar/lapack/3.12.0/lib/cmake/ (8 files)
/opt/homebrew/Cellar/lapack/3.12.0/lib/pkgconfig/ (3 files)
/opt/homebrew/Cellar/lapack/3.12.0/lib/ (6 other files) -------->重要：头文件中函数的实现文件路径
```

图 1 获得库的头文件以及依赖路径

配置 CMake 文件：

```
cmake_minimum_required(VERSION 3.10)
project(learn_lapack) # 项目名称

include_directories(/opt/homebrew/Cellar/lapack/3.12.0/include/) # 头文件路径
link_directories(/opt/homebrew/Cellar/lapack/3.12.0/lib/) # lib库路径，实现头文件中的函数

find_package(LAPACK REQUIRED) # 查找LAPACK库

add_executable(hello main.c) # 添加可执行文件

target_link_libraries(hello PRIVATE -llapacke -llapack -lblas -lm) # 链接LAPACK库
```

图 2 配置相应的 CMake 文件

使用 CMake 成功构建 Lapack 的 Demo：

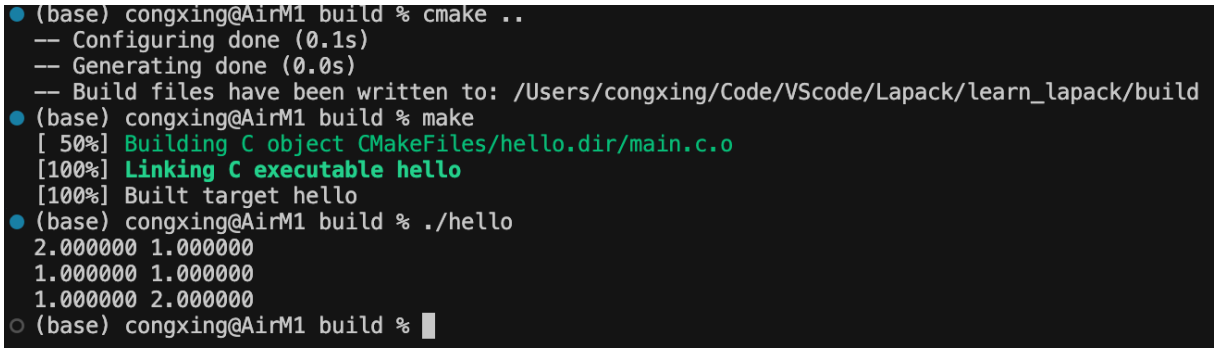

图 3 成功构建 Lapack Demo

（2）Lapack 的命名规则

所有函数都是以 **XYYZZZ** 的形式命名，对于某些函数，没有第六个字符，只是 XYYZZ 的形式。

- **X** - 第一个字符代表的是数据类型：
  - **S** ⟶ **REAL**，单精度实数
  - **D** ⟶ **DOUBLE PRECISION**，双精度实数
  - **C** ⟶ **COMPLEX**，单精度复数
  - **Z** ⟶ **COMPLEX*16** 或 **DOUBLE COMPLEX**

  注意：在新版 LAPACK 中含有使用重复迭代法的函数 **DSGESV** 和 **ZCDESV**。头 2 个字母表示使用的精度：**DS**，输入数据是 double 双精度，算法使用单精度。**ZC**，输入数据是 complex*16，算法使用 complex 单精度复数
- **YY** - 第二三个字符代表的是数组的类型：
  - **GE** ⟶ general 一般情形（即非对称，在有些情形下为矩形）
  - **DI** ⟶ diagonal，对角矩阵
  - 还有很多。
- **ZZZ** - 最后三个字母 ZZZ 代表计算方法：
  - 比如，**S GE BRD** 是一个单精度函数，用于把一个实数一般阵压缩为双对角阵（a bidiagonal reduction，即 BRD）。

关于详细的笔记，已经上传到网站：在 mac 上安装和使用 Lapack 和 Lapacke

2. 学习 HIP 编程

（1）AMD GPU 与 Nvidia GPU 硬件上的区别

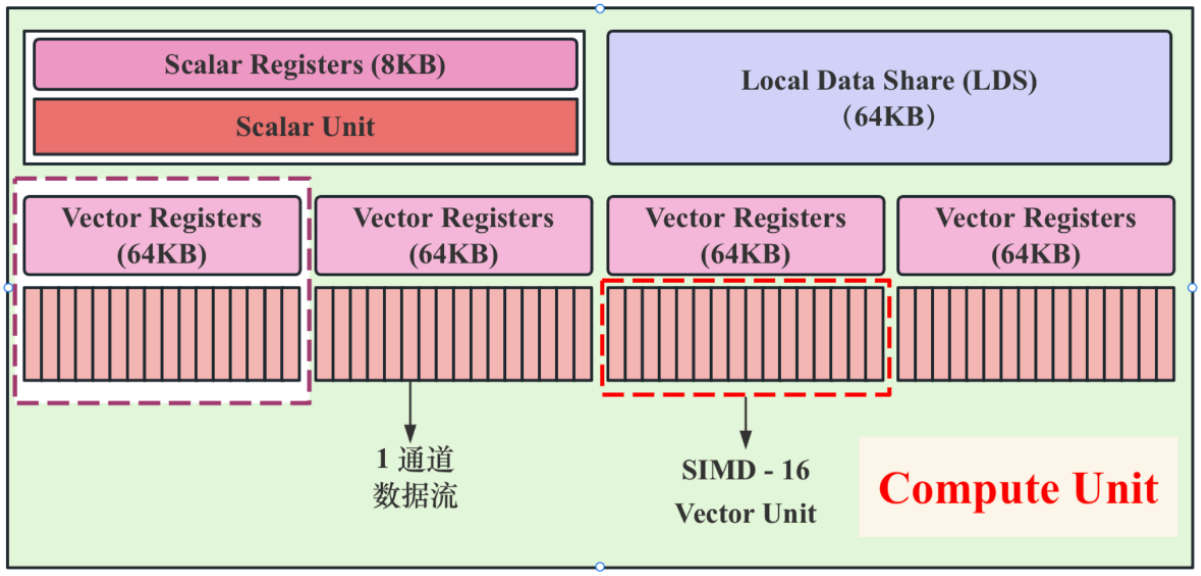

图 4 AMD GPU 中计算单元的内部结构

AMD 中一个计算单元（Compute Unit）有 4 个 SIMD，每个 SIMD 有 16 个通道，因此可以看作为 64 线程集成为一个单元，而在 Nvidia 中，每个计算单元只有 32 线程，因此，可能在具有较多分支指令的代码中，如果发生线程分散，会产生较大的性能差异。

（2）任务流的加速方式

分配小核到不同的流上去执行，提高并行度

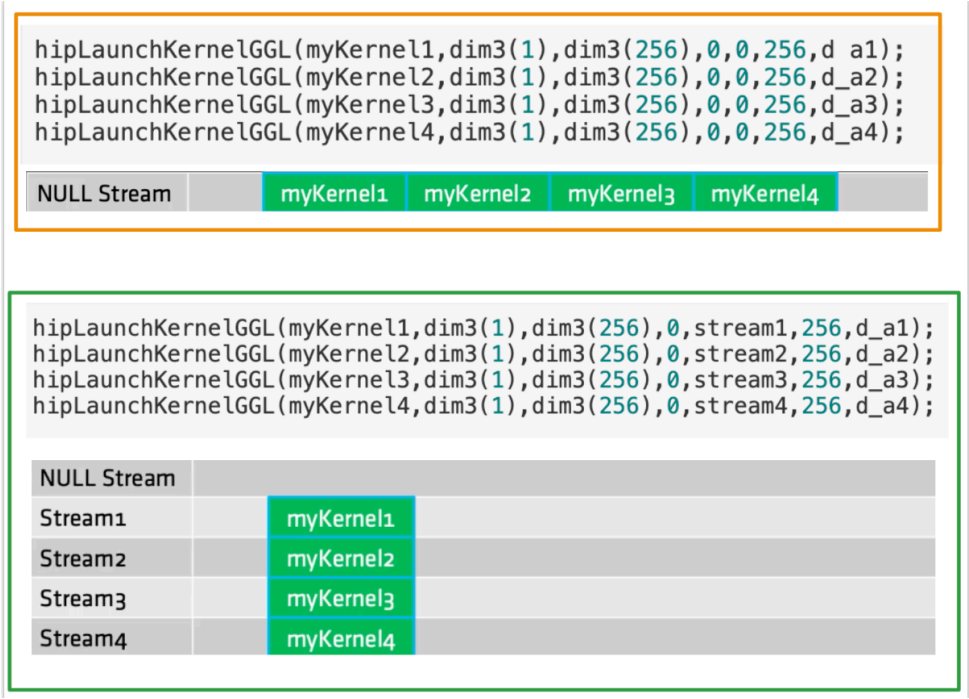

图 5 使用一个流和使用多个流执行多个任务的对比

在这样做时，需要保证各个内核不会修改内存中的共享部分，以免产生数据冲突。如果一个内核实际上占用了整个 GPU 资源，这时 GPU 已经没有多余的流分配给其他流去运行了，因此此时则不应该再使用此方法进行性能的优化。但是，对于多个小内核来说，这将会有帮助。

用计算内核时间覆盖数据移动的时间

GPU 具有独立的引擎，用于执行主机和设备之间的**内存移动**、**排队**和**计算内核**。在 AMD GPU 上，**允许这三种操作在不分割 GPU 资源的情况下重复使用**（前提是：这些操作都应在各自的非 NULL 流队列中，并且在主机和设备之间的复制操作所涉及的**主机内存都应被固定**）。

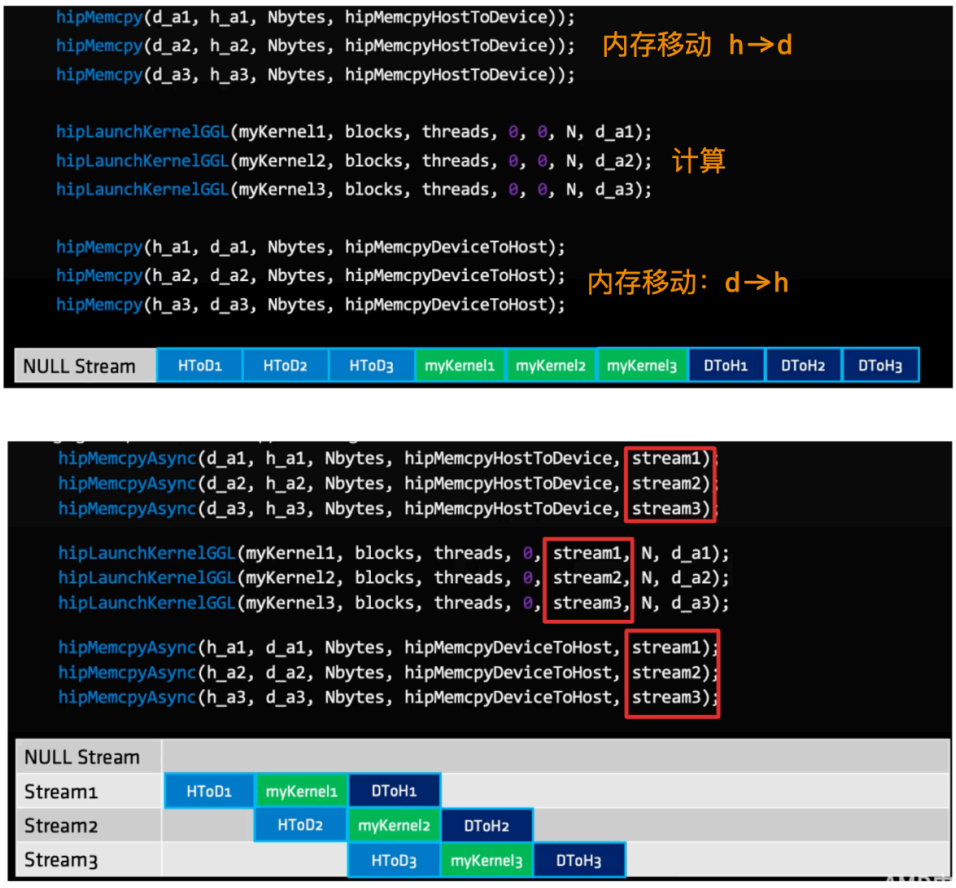

图 6 使用 Overlapping 提高并行度，进行加速操作

（3）AMD GPU 库文件

- **roc***: 以 roc 开头的库都是 AMD GCN 库，通常是用 HIP 编写的。
- **cu***: 以 cu 开头的库都是 Nvidia 库。
- **hip***: 以 hip 开头的库通常是位于 roc 或 cu 后段之上的接口层。
  - 例如，hipBLAS 是 rocBLAS 和 cuBLAS 之间的接口层，如果使用的是 Nvidia 设备，则会调用 cuBLAS，如果使用的是 AMD 设备，则会调用 rocBLAS。

- hip 库可由 hipcc 编译并生成相应设备的调用，hip 层只是一个简单的封装，用于调用正确的设备。
- 会替换 HIP 调用并内联 cuBLAS 调用，不会在 Nvidia 设备上产生任何运行开销。

注意：如果在乎代码的可移植性（既运行在 AMD 设备上又运行在 Nvidia 设备上），需要使用 hipBLAS，让编译器负责管理运行时间的调用。如果不在乎移植性，只想在 AMD 的设备上运行，直接使用 rocBLAS 即可。

因此，如果使用 hip 进行编程，是可以兼容 Nvidia 显卡的，但是只支持较低版本的 CUDA。

（4）从 CUDA 向 HIP 转变

由于两者的差别并不是很大，大部分只是命名的差别，因此，有两种自动化方案：**Hipify-Perl 和 Hipify-clang** 。它们被设计用于自动转换现有的 CUDA 代码，但它们不是万能的，不能自动转换所有的代码，不能自动转换部分的 CUDA 代码需要程序员手动去进行处理。

因此，对于开展的项目来说，利用上面这两个工具，结合 GPT，应该会容易很多。

详细的笔记已经放在网站上：在 AMDGPU 上使用 HIP 编程

Next

回国之后，登陆超算平台，搭建使用 HIP 编程的 Demo。

读 lapack 的源代码，在源代码的基础上进行相关论文实现的改进，观察效果。

阅读 Optimizing High-Performance Linpack for Exascale Accelerated Architectures 这篇论文。