南開大學
Nankai University

计算机学院

GPU 编程实验报告

Grobner 基计算中的高斯消去

姓名：卢艺晗

学号：2213583

专业：计算机科学与技术

2024 年 6 月 7 日

目录

1 综述

1.1 摘要

本次 GPU 编程实验分为两部分。第一部分为 oneAPI 版本的线上学习，对于《C++ SYCL》模块的课程，完成了基础要求中的《Essentials of SYCL》课程三章内容的学习和编程练习，并在进阶要求中学习了《Introduction to GPU Optimization》课程。

第二部分为 CUDA 版本的编程练习，针对普通高斯消去，进行 GPU 加速实现，并对比了三种不同任务划分方式的性能优劣。实验表明，二维块划分方式的 GPU 加速算法性能最优，在大规模矩阵下能达到数百倍的加速比。

1.2 实验平台信息

指标	值
CPU 型号	Intel® Core™ i7-14650HX
CPU 核数	16
操作系统及版本	windows11 家庭中文版
编译器	Visual Studio 2022

表 1: CPU 实验平台信息

指标	值
显卡	NVIDIA GeForce RTX4060 Laptop GPU
显存大小	8GB
CUDA 版本	V12.1.66
编译器	Visual Studio 2022

表 2: GPU 实验平台信息

1.3 选题简介

普通高斯消元：高斯消元法（也称高斯消去法）是求解线性方程组的一个基本方法, 它可以用于决定线性方程组的解、矩阵的秩, 以及可逆矩阵的逆。高斯消元法是通过矩阵的行变换达到消元的目的, 从而将方程组的系数矩阵转化为三角矩阵, 最后获得方程组的解, 其主要包括 3 个过程: 除法, 消元和回代求解。

普通高斯消元法的串行算法具体实现已经在开题报告中体现，在此由于篇幅所限，不再赘述。

作为期末研究的 GPU 加速方向，本实验拟针对普通高斯消去，进行 CUDA 版本的 GPU 加速实现，并探索三种不同任务划分方式的性能优劣，以期在期末研究中将实验结论应用于特殊高斯消去。

特殊高斯消去：本实验选取 Grobner 基计算中的高斯消去这一特殊的高斯消元过程，进行串行算法的实现和 x86 平台上的 MPI 并行化实现。与普通高斯消去相比，该特殊高斯消去有三点不同之处：

（1）矩阵元素的值只能是 0 或 1；

（2）Grobner 基计算中的高斯消去只有异或运算：加法运算实际为异或运算（0+0=0，0+1=1，1+0=1，1+1=0）；减法运算为加法的逆运算，亦为异或运算；乘法运算（0*0=0，0*1=0，1*0=0，1*1=0）实际可消。

（3）矩阵行分为两类：“消元子”和“被消元行”，在输入时给定：在消去过程中，“消元子”充当减数，所有消元子首项位置均不同，但不涵盖所有对角线元素；“被消元子”充当被减数，若恰好包含消元子中缺失的对角线 1 元素，则升格为消元子。

在本次实验中，主要针对普通高斯消去进行 CUDA 版本的加速实现。对于特殊高斯消去，将在期末研究中应用本实验探究的得到的最佳算法策略。

1.4 github 链接

本实验的代码已经上传到 GitHub 的仓库上。

源码 GitHub 链接： GitHub/Lucyannn/Parallel Design

2 （基础要求）《Essentials of SYCL》课程学习

2.1 学习过程简述

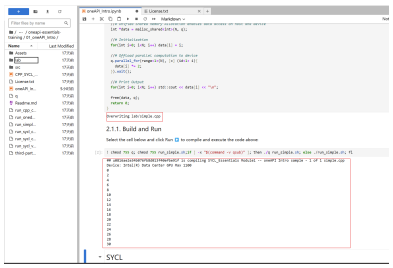
2.1.1 《01 oneAPI Intro》

第一节《Introduction to oneAPI and SYCL》主要对 oneAPI 和 SYCL 进行入门介绍，从多架构编程的挑战引出对 oneAPI 开发动机的介绍，进而讲述了什么是 oneAPI、什么是 DPC++、什么是 SYCL 等问题以及他们的关系。课程对 oneAPI 编程模型进行介绍，包括平台模型、执行模型、内存模型、核函数编程模型，并辅以图画，帮助学习者理解 oneAPI 的模型架构。此外，课程还通过示例讲述了如何在 Intel DevCloud 或本地编译并运行 SYCL 程序。并在课程末尾提供了一些网站和示例资源，以供学习者自取。

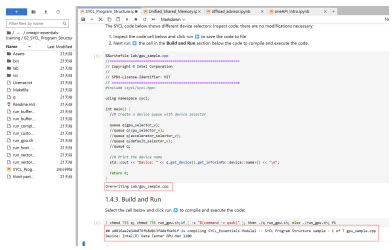
第二节《C++ SYCL Introduction》围绕计算卸载到加速器这一中心问题，介绍了一些加速器语言，并着重强调 SYCL 的优势。接着，对 SYCL 卸载到加速器的主要编程范式进行介绍。最后布置一道编程练习题来检测学习效果。

我的学习过程主要分为四个步骤：（1）阅读网页内容进行课程学习；（2）上网查阅相关资料进行深入理解；（3）运行和分析网站提供的代码示例，编写第二节布置的练习题，进行实践检验；（4）有选择的阅读课程末尾提供的 oneAPI 代码示例和 SYCL 2020 规范文档，对 oneAPI 和 SYCL 的编程方法进行初步了解。

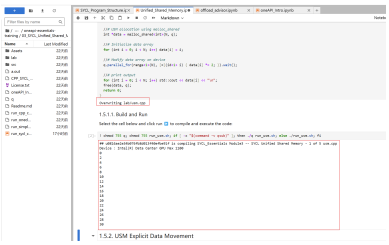
节选了其中一张检查、运行代码示例的学习记录如图2.1(a) 所示。



(a) 代码示例执行记录-第一章



(b) 代码示例执行记录-第二章



(c) 代码示例执行记录-第三章

图 2.1: 学习过程截图-代码示例的运行

2.1.2 《02 SYCL Program Structure》

该课程围绕 SYCL 程序结构，主要讲述了基本 SYCL 类及主要组件语法、选择卸载的设备、如何编写 SYCL 程序、使用 Host 访问器和 Buffer 销毁来进行同步等内容。通过学习课程网站内容以及编程实现该课程布置的练习题，我对于 SYCL 程序有了系统的掌握。

节选了其中一张检查、运行代码示例的学习记录如图2.1(b) 所示。

2.1.3 《03 SYCL Unified Shared Memory》

该课程围绕统一共享内存 USM，主要介绍了 USM 的特点和使用方法，以及如何在 USM 下管理数据依赖性。我学习课程网页内容，对需要运行的代码示例进行检查、运行，并完成了课程布置的两道练习题和一道作业题。

节选了其中一张检查、运行代码示例的学习记录如图2.1(c) 所示。

2.2 学习内容梳理

2.2.1 《01 oneAPI Intro》

（1）认识 oneAPI 和 SYCL

- **oneAPI** 是一个跨架构的编程模型，旨在简化在不同硬件（如 CPU、GPU、FPGA 等）上开发高性能应用程序的过程。它提供了一整套工具和库，使得开发者可以用统一的编程语言和接口来编写代码，而无需针对不同硬件进行大量修改。由于跨架构的兼容性、专为高性能计算设计以及统一的编程接口，oneAPI 体现出巨大的优势。
- **SYCL** 属于 oneAPI 的一部分，允许开发者使用 C++ 以 “单一源” 风格编写异构处理器的代码。不仅为开发者提供便利，还使得使编译器能够对整个程序进行分析和优化。
- **DPC++**（即 Data Parallel C++）是 SYCL 编译器的 oneAPI 实现，它利用 C++ 的效率和结构，并结合数据并行性和异构编程的 SYCL* 标准。不仅能够简化代码，还可以通过让程序员控制程序执行并启用特定硬件的功能来提高性能。

（2）理解 oneAPI 编程模型

- **平台模型**：平台模型描述计算系统的硬件结构和资源，包括计算设备（如 CPU、GPU、FPGA 等）和它们之间的通信方式。其中，“主机” 通常指计算节点中的 CPU，负责管理和调度任务；而 “设备” 往往指计算节点中的其他硬件（如 GPU、FPGA），负责执行具体的计算任务。一台主机与一台或多台设备通信，每个设备可以包含一个或多个计算单元，每个计算单元可以包含一个或多个处理元件。
- **执行模型**：执行模型定义了代码（内核）如何在设备上执行并与控制主机交互。主机执行模型通过命令组协调主机和设备之间的执行和数据管理；设备执行模型指定如何在加速器上完成计算。其中，命令组是诸如内核调用和访问器之类的命令分组，被提交到队列以供执行，使用执行模型的程序声明并实例化队列，队列可以根据程序要求进行有序或无序的命令执行。
- **内存模型**：内存模型是一种抽象，用于协调主机和设备之间的内存分配和管理。它定义了主机和设备之间的内存交互。内存通过内存对象（分为缓冲区和图像两种）的方式指定，主机或设备通过访问器指定访问位置（主机或设备）和访问方式（如读、写等）来进行内存交互。
- **核函数编程模型**：核函数编程模型描述了如何编写和执行核函数。核函数是在设备上运行的并行计算单元，通常用于处理大规模数据的并行计算。而命令组则用于定义核函数的执行和数据的管理，确保并行计算任务能够正确执行。

（3）学习 SYCL 程序的编译、运行方法

编译、运行 SYCL 程序需要三个步骤：初始化环境变量，编译源代码以及运行可执行文件。在 Intel DevCloud 上，可以通过编写脚本（包含上述三个步骤），将作业提交到 DevCloud 上的 GPU 节点执行，等待打印输出和错误信息。在本地，如果安装了英特尔 ® oneAPI Base Toolkit，也可以通过脚本运行。脚本示例如下：

```
source /opt/intel/inteloneapi/setvars.sh
icpx -fsycl simple.cpp -o simple
./simple
```

（4）了解计算卸载到加速器问题

高性能计算中的大型计算问题在 GPU 等专用硬件加速器上的运行速度比在 CPU 上快得多。GPU 等加速器可以利用硬件中的并行性同时运行许多较小的计算。因此，将主机上的计算卸载到加速器，有助于提高程序运行效率。

加速器编程语言有 CUDA、OpenCL、SYCL 等多种语言，其中 SYCL 相对其它语言的显著优势在于，它具有跨架构兼容性、高性能、编程语言简洁等特点。

（5）学习 C++ SYCL 编程范式

课程通过简单的程序示例讲解 SYCL 编程的主要语法。

%%writefile 用于保存文件；使用 SYCL 编程需要引用头文件 sycl/sycl.hpp 以及命名空间 sycl。

SYCL 支持三个重要功能：设备选择、内存分配和提交计算任务。

- 设备选择：sycl::queue 用于选择卸载计算的设备，在声明变量时通过构造参数指定选择的设备；
- 内存分配：sycl::malloc_shared 用于分配共享内存，主机和设备都可以访问；sycl::malloc_device 在设备上分配内存，其中数据显式移动；
- 提交计算任务：q.single_task 是提交任务在设备上执行的最基本方法，.wait() 方法用于通过等待任务完成来与主机同步；此外，q.parallel_for 允许提交任务并在设备上并行执行。

2.2.2 《02 SYCL Program Structure》

（1）SYCL 类 SYCL 语言由一组 C++ 类、模板和库组成。其中 C++ 的全部功能可在应用程序和命令组范围内使用，即可以在主机上完全使用，而在设备上使用受限。主要的 SYCL 类和功能总结如下：

- **设备**: 设备类代表加速器。该类中主要含有查询加速器各种信息的成员函数。
- **设备选择器**: 允许在运行时选择特定设备来根据用户提供的启发法执行内核。标准设备选择器有 default_selector_v、cpu_selector_v、gpu_selector_v、accelerator_selector_v。
- **队列**: 队列是一种将工作提交到设备的机制。队列提交要由 SYCL 运行时的命令组来执行。一个队列只能映射到一个设备，多个队列可以映射到同一设备。
- **内核**: 内核类封装了命令组被实例化时用于在设备上执行代码的方法和数据。内核对象不是用户显式构造，而是在调用内核调度函数（例如 parallel_for）时才构造。

  其中一种主要的内核是并行内核，也就是调用 parallel_for 时构造的内核。它允许一个操作的多个实例并行执行，对于卸载基本 for 循环的并行执行非常有效，且每次迭代都是独立且不需先后顺序的。并行内核又分为基本并行内核和 ND 范围内核。基本并行内核是并行化 for 循环的简单方法，但不允许在硬件级别优化性能。ND-Range 内核是表达并行性的另一种方式，它通过提供对本地内存的访问并将执行映射到硬件上的计算单元来实现低级性能调整。

（2）内存模型

内存模型分为统一共享内存模型 (USM) 和缓冲区内存模型两类。统一共享内存模型提供了更细粒度的内存控制和直接的指针访问，适合需要灵活内存管理的场景。缓冲区内存模型提供了更高层次的抽象，简化了内存管理和数据传输，适合数据并行计算和复杂的内存访问模式。

以使用统一共享内存模型为例，总结 SYCL 进行程序编写的主要结构流程。

SYCL 程序的语言是标准 C++。该程序在主机上调用，并将计算卸载到加速器。程序员使用 SYCL 的队列和内核抽象来指示应卸载哪些部分的计算和数据。

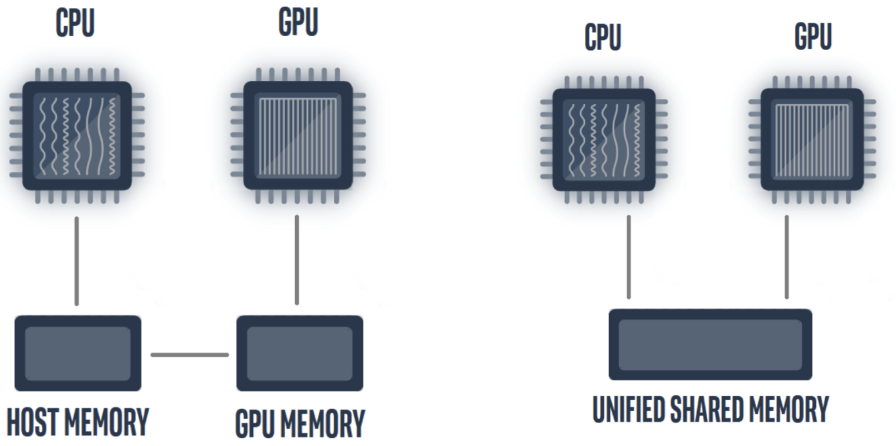
- 程序的第一步一般为创建队列，通过将任务提交到队列来将计算卸载到设备。程序员可以使用选择器选择 CPU、GPU、FPGA 等设备。
- 设备和主机可以共享物理内存或拥有不同的内存。当存储器不同时，卸载计算需要在主机和设备之间复制数据。在统一共享内存模型中，可使用 malloc_device 来在设备上分配共享内存，并使用 memcpy 方法在主机和设备之间复制数据；或使用 malloc_shared 分配共享内存，不需显式移动数据等。
- 在 SYCL 程序中，会定义一个或多个内核，应用于索引空间中的每个点，在加速器上执行功能。内核常封装在 C++ lambda 函数中。

（3）同步

将加速器上的数据同步回主机，一般有使用主机访问器和缓冲区销毁等方法。主机访问器同步通过访问器的创建和销毁自动管理数据同步，确保主机端访问的数据始终是最新的。缓冲区销毁同步方法则在缓冲区销毁时自动进行数据同步，确保设备上的数据正确传回主机，并且所有相关任务已经完成。

2.2.3 《03 SYCL Unified Shared Memory》

- **USM 定义及特点**: 统一共享内存（USM）是一种基于指针的内存管理，借助 USM，程序员可以在主机和设备上引用相同的内存对象。图**??**是内存结构的简化示意图，(a) 表示没有 USM 的内存结构，主机和设备上的数据是独立的；(b) 表示有 USM 的内存结构，可以共享内存对象。USM 使开发更加简洁。
- **两种数据移动方式**:USM 提供了管理内存的显式和隐式模型。显式模型常用 malloc_device 在设备上分配内存，它不可在主机上访问。在主机和设备之间复制数据时，程序员需要显式地使用 memcpy 方法。隐式模型可通过 malloc_host 在主机上分配内存，或使用 malloc_shared 分配共享内存，二者在主机和设备上均可访问。它们的数据移动隐式发生，即不需程序员使用 memcpy 进行数据复制，使得程序开发更加简洁。
- **使用 USM 下的数据依赖性管理**：由于任务是异步的，可以同时执行，因此使用 USM 时需要通过事件来维护数据间的依赖关系。主要有三种方法：
  1. 在内核中使用 wait(): 下一个任务开始之前，在内核任务上使用 wait() 等待其执行完毕，但是它将阻止主机上的执行。
  2. 使用队列的 in_order() 属性：声明队列的实例化对象时，指定 in_order() 属性，那么在此队列上创建的任务将按照创建次序依次执行。
  3. 使用 depends_on() 方法：在命令组中，使用 depends_on() 方法，确保该内核在指定内核任务完成之后执行。



(a) 无 USM 的内存结构 (b) 使用 USM 的内存结构

图 2.2: 内存结构简图

2.3 实践练习

第一章课程布置了一道“矢量相加”的练习题，该题目要求使用 SYCL 的编程语法，通过计算卸载到加速器的方式，实现适量相加的函数功能。我的编程实现具体展示如下：

```
#include <iostream>
//# STEP 1 : Include header for SYCL
#include <sycl/sycl.hpp>
using namespace sycl;

int main(){
//# STEP 2: Create a SYCL queue and device selection for offload
queue q;
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";
//# initialize some data array
const int N = 16;

//# STEP 3: Allocate memory so that both host and device can access
float *a = malloc_shared<float>(N, q);
float *b = malloc_shared<float>(N, q);
float *c = malloc_shared<float>(N, q);
for(int i=0;i<N;i++) {
    a[i] = 1;  b[i] = 2;  c[i] = 0;
}
//# STEP 4: Submit computation to Offload device
q.parallel_for(range<1>(N), [=](id<1> i) {
    c[i] = a[i] + b[i];
```

```
23  }).wait();
24  //# print output
25  for(int i=0;i<N;i++) std::cout << c[i] << "\n";
26
27
```

通过引用 sycl 库，使用 sycl::queue 选择设备，malloc_shared 为三个数组分配共享内存，q.parallel_for 方法和 lambda 函数结合实现计算任务提交给设备，该程序实现成功。运行输出为 16 个 “3”，每个结果均符合数组 a 与数组 b 对应元素求和的结果，如图2.3(a) 和 (b) 所示，验证了编程实现的正确性。

此外，第二章、第三章也分别布置了练习题，我均进行了编程并运行成功。由于篇幅所限，不再详细介绍。第二章练习题截图记录如图2.3(c) 和 (d) 所示，第三章练习题截图记录如图2.3(e) 和 (f) 所示。

图中，成功输出部分均用红色矩形框出。

(a) 第一章编程实现 (c) 第二章编程实现 (e) 第三章编程实现

(b) 第一章运行成功 (d) 第二章运行成功 (f) 第三章运行成功

图 2.3: 第一、二、三章编程练习及运行成功截图

3 （进阶要求）《C++ SYCL》模块其它课程学习

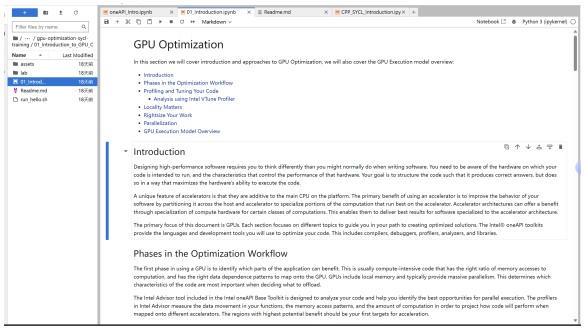
除了基础要求中的课程以外，我还学习了 C++ SYCL 模块的另一门课程，即《Introduction to GPU Optimization》，学习过程和收获简要记录如下。

3.1 学习过程简述

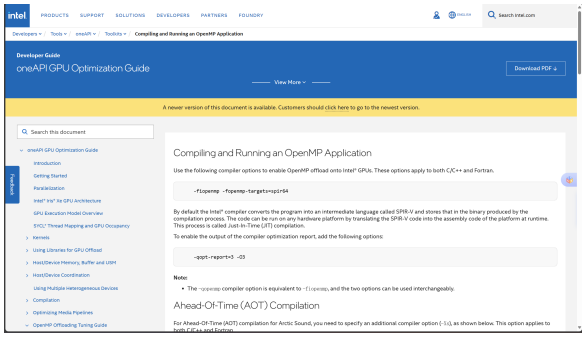
本课程围绕 GPU 优化这一主题，讲解了为什么要使用加速器、GPU 优化工作的主要流程、如何使用 VTune profiler 等工具对代码进行分析和调整、分区、工作规模、并行化等 GPU 优化的主要问题。此外，课程还对 GPU 执行模型进行概述，让学习者理解了 GPU 实现高性能计算的原理。

我的学习过程主要分为三步：（1）阅读课程网站，学习知识；（2）参照课程网站讲解，进行 Intel VTune profiler 的 GPU 测试的配置；（3）阅读课程网站提供的拓展学习资源，进行 OpenMP 等相关 GPU 并行优化的知识补充。

学习过程的截图记录如图3.4所示。



(a) 课程网页学习截图



(b) 拓展资源学习截图

图 3.4: 课程学习记录截图

3.2 学习内容梳理

使用加速器的主要优点在于，通过在主机和加速器之间进行分区来改进软件的行为，可以专门化在加速器上运行最佳的计算部分。加速器架构可以通过针对某些类别的计算专门化计算硬件来提供优势，使其能够为加速器架构专用的软件提供最佳的性能效果。本课程主要介绍 GPU 这一加速器，讲述如何进行 GPU 优化。

（1）GPU 优化流程：

使用 GPU 的第一个阶段是确定程序的哪些部分可以受益。这部分代码通常是计算密集型代码，且具有正确的内存访问与计算比率，以及映射到 GPU 的正确数据依赖模式。GPU 包含本地内存，通常提供大规模并行性。这决定了卸载代码时的重要特征。

在进行代码设计时，有一些优化的原则：

- **让所有计算资源保持忙碌状态**: 必须有足够的独立任务来使设备饱和并充分利用所有执行资源。例如，如果设备有 100 个计算核心，但程序只有一项任务，则 99% 的设备将处于空闲状态，这无法使 GPU 发挥其优势，还可能由于通信等开销降低性能。良好的设计应该创建比可用计算资源更多的独立任务，以便设备可以在先前任务完成后安排更多工作，使其优势得以充分发挥。

- **最大限度地减少主机和设备之间的同步**: 在程序执行时，主机在设备上启动内核并等待其完成，而启动内核会产生开销，因此需要精心设计程序，以最小化内核启动的次数，降低同步开销。

- **最大限度地减少主机和设备之间的数据传输**: 数据通常在主机上启动并复制到设备作为计算的输入，计算完成后，需要将结果传输回主机。数据传输也是一部分开销，为了获得最佳性能，要尽量将计算的中间结果保留在设备上，最大程度地减少数据传输。开发者可以通过重叠计算和数据移动来减少数据传输的影响，使计算核心永远不必等待数据。

- **将数据保存在更快的内存中并使用合适的访问模式**:GPU 架构具有不同类型的内存，并且这些内存具有不同的访问成本。寄存器、高速缓存和暂存器的访问成本比本地内存更低，但容量较小。当数据加载到寄存器、缓存行或内存页时，开发者可以使用一种访问模式，在移动到下一个块之

前使用所有数据。当内存中的数据写回硬盘时，要设计避免所有计算核心同时访问同一内存组的方式。

（2）GPU 优化策略：

优化 GPU 的关键在于充分利用其内存层次结构、合理分配工作量以及有效的并行化技术等。详述如下：

- **数据本地化**。加速器通常具有专门的内存，并且地址空间是分离的。内存层次结构中，访问效率为寄存器 > 缓存 > 主存。使数据尽可能靠近执行点有助于提高效率。有以下三个重点方法：
  1. 在加速器上分配数据并尽可能驻留。
  2. 执行内核时访问连续内存块。
  3. 将代码重组为具有更高数据重用性的块。
- **调整工作规模**。数据并行加速器通过多次复制执行单元来提高性能。若要充分利用并行处理器，需要将计算分解为大量并行活动，因此开发者需要设计合适的共工作规模来达到此目的。
- **并行化**。由于加速器包含许多能够并行执行代码的独立处理单元，因此程序的并行性对于有效使用加速器至关重要。开发并行代码有三种主要方法：使用并行编程语言或 API、使用并行编译器、使用并行库。

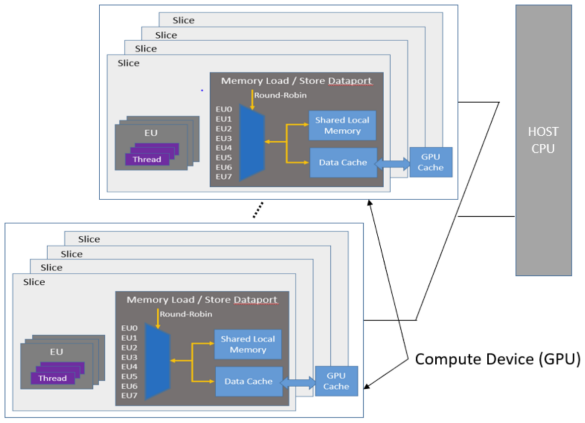


图 3.5: GPU 执行模型

（3）GPU 执行模型：

GPU 执行模型如图3.5所示。GPU 执行模型通过主机程序、内核、命令队列、工作组和工作项等机制实现高效并行计算。了解这些概念和机制对于优化和充分利用 GPU 的计算能力至关重要。通过合理调度和同步工作项，以及利用 SIMD 向量处理，GPU 可以大幅提升计算性能。

- **执行模型基本结构**：GPU 执行模型主要包括主机程序和内核两部分。主机程序通过命令队列与内核交互，并协调内核的执行；内核则负责在计算单元集群内执行特定任务。每个设备都有自己的命令队列。命令被提交到命令队列后，会检查依赖关系，然后在计算单元内的执行引擎（VE）上执行。执行完毕后，内核通过“线程结束”消息结束生命周期。
- **内核的调度与执行**：当提交一个内核执行命令时，定义了一个索引空间或 N 维范围。计算设备在索引空间的每个点上执行内核函数，称为一个工作项。多个工作项组成一个工作组。执行引擎可以将工作项组织成 SIMD 向量格式，运行相同的内核，从而加速计算。

- **执行顺序与同步**：设备可以任意顺序计算每个工作组，工作组内的工作项并发执行，但不保证执行顺序。关于同步机制，障碍函数用于工作组内的同步，命令队列中的命令可以依赖于其他命令的执行点进行同步，原子操作和内存屏障控制某个工作项的内存操作如何对其他工作项可见，提供了数据并行计算模型中的微观同步点。

4 （进阶要求）普通高斯消去的 CUDA 编程

4.1 普通高斯消去的 GPU 加速实现

（1）实验设计

本实验旨在对普通高斯消去算法进行 GPU 加速的编程实现。在程序设计中，首先将主机上的矩阵数据复制到设备上，然后执行高斯消去计算，最后所有线程同步后将结果传回主机。对于高斯消去过程中的两处循环（除法部分和消去部分），各使用一个核函数。具体的代码实现思路如下伪代码所示。

（2）编程实现

普通高斯消去的 CUDA 编程思路使用伪代码展示如下。算法1描述了 CUDA 版本高斯消去的主要流程；算法2描述第一个核函数的算法思路（即除法部分），算法3描述第二个核函数的算法思路（即消去部分）。

Algorithm 1 Gaussian Elimination(CUDA)

procedure GaussianElimination(h_data, N, thread_num)
    $d\_data \leftarrow$ CudaMalloc($N \times N \times$ sizeof(float)) ▷ 在设备上分配内存
    CudaMemcpy(d_data, h_data, $N \times N \times$ sizeof(float), HostToDevice) 将数据从主机复制到设备
    $threadsPerBlock \leftarrow thread\_num$ ▷ 每个块的线程数
    for $k \leftarrow 0$ **to** $N-1$ **do** ▷ 遍历矩阵的每一列
        $divideBlocks \leftarrow \lceil \frac{N-k-1}{threadsPerBlock} \rceil$ ▷ 计算划分块的数量
        Kernel(DivideLargeMatrixKernel, divideBlocks, threadsPerBlock, d_data, k, N) ▷ 启动划分内核
        CudaDeviceSynchronize ▷ 等待划分内核完成
        $eliminateBlocks \leftarrow N-k-1$ ▷ 计算消去块的数量
        Kernel(EliminateLargeMatrixKernel, eliminateBlocks, threadsPerBlock, d_data, k, N) ▷ 启动消去内核
        CudaDeviceSynchronize ▷ 等待消去内核完成
    end for
    CudaMemcpy(h_data, d_data, $N \times N \times$ sizeof(float), DeviceToHost) ▷ 将结果从设备复制回主机
    CudaFree(d_data) ▷ 释放设备上的内存
end procedure

Algorithm 2 Divide Large Matrix Kernel

procedure DivideLargeMatrixKernel(data, k, N)
    $tid \leftarrow blockDim.x \times blockIdx.x + threadIdx.x$
    $total\_cols \leftarrow N-k-1$
    for $i \leftarrow tid$ **to** $total\_cols$ **step** $blockDim.x \times gridDim.x$ **do**

```
        col ← k + 1 + i
        data[k × N + col] ← data[k × N + col]/data[k × N + k]
    end for
end procedure
```

Algorithm 3 Eliminate Large Matrix Kernel

```
procedure EliminateLargeMatrixKernel(data, k, N)
    row ← blockIdx.x + k + 1
    if row ≥ N then
        return
    end if
    col_start ← threadIdx.x
    total_cols ← N − k − 1
    for i ← col_start to total_cols step blockDim.x do
        col ← k + 1 + i
        data[row × N + col] ← data[row × N + col] − data[row × N + k] × data[k × N + col]
    end for
end procedure
```

（3）测试结果

测量 CPU 上串行算法与 CUDA 并行算法程序的执行时间，并计算加速比，数据记录见表格3。

矩阵规模	128	256	512	1024	2048	4096
CPU_time	0.00130	0.01029	0.08436	0.66203	7.48394	56.85620
GPU_time	0.18945	0.00539	0.01281	0.04821	0.20404	1.34956
加速比	0.00687	1.90785	6.58642	13.73113	36.67951	42.12943

表 3: 普通高斯消去 GPU 加速算法程序执行时间及加速比（保留 5 位小数）

将 GPU 加速算法程序的加速比绘制成散点图，如图4.6(a) 所示。

（4）结果分析

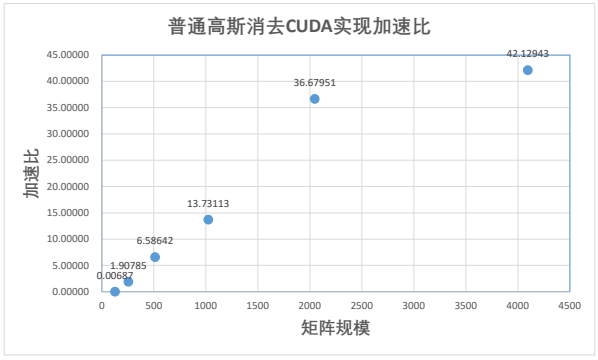
结合表3, 由图4.6(a) 可知, 在矩阵规模很小时 (N=128), GPU 版本的程序加速比远小于 1, 未实现加速; 原因是矩阵规模小导致程序本身耗时不多, 加之将数据迁移到设备上进行并行, 增加了数据移动和同步等时间开销。矩阵规模越大, 加速比越大, 因为将更多的计算任务分配到 GPU 的计算单元上, 更能充分利用其资源。矩阵规模 N=4096 时, 加速比达到了 42.12943, 是 CPU 上 MPI+OpenMP+AVX 混合并行化的最大加速比 15 的 3 倍多，可见 GPU 加速的效果显著。

4.2 不同任务划分方式对比

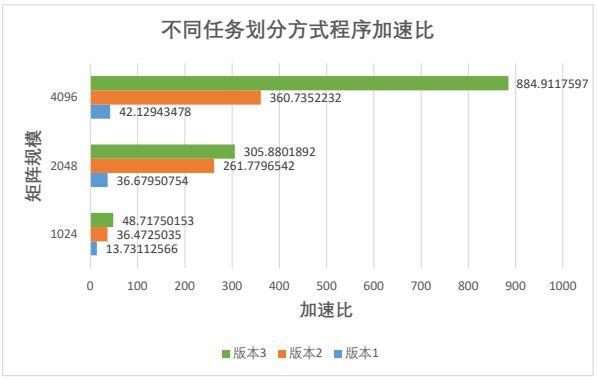
（1）实验设计

在 CUDA 中，每个块的线程数是有限的（经查询，本机 GPU 每个块的最大线程数为 1024）。当矩阵规模较大时，一个块内的线程可能不足以处理一整行或一整列的任务，这时需要采取一些策略来继续并行计算。

将计算任务分配给 GPU 的线程时，有多种分配方式。本实验旨在探究不同任务划分方式对加速效果的影响。实验设计了三种不同任务划分方式的普通高斯消去的 GPU 加速算法。划分方式如下:



(a) 普通高斯消去 GPU 加速算法程序加速比



(b) 不同任务划分方式程序加速比

图 4.6: 普通高斯消去 CUDA 版本程序加速比

1. 版本 1 采用循环的方式（即 4.1 中 GPU 加速编程实现采取的方式）。即在消去部分，每个块处理一行，对于每一行的所有列，先逐一分配给每个线程，余处的列再从第一个线程开始逐一分配；
2. 版本 2 采用一维块划分的方式。即在消去部分，每个块处理一行，对于每一行的所有列，先计算平均分配给每个线程的列数，在为每个线程分配连续的指定数目的列；
3. 版本 3 采用二维块划分的方式。即每个块处理一个计算好大小的二维区域，每个线程处理块中一个元素。

（2）测试结果

结合第 4.1 节实验结果可知，GPU 加速算法在矩阵规模较大时效果更显著。由于本实验重在比较不同任务划分方式的加速效果，因此仅对较大规模（N=1024、2048、4096）三个规模的矩阵进行测试和分析。程序执行时间和加速比如表4所示，加速比绘制成柱状图如图4.6(a) 所示。

矩阵规模	1024	2048	4096	矩阵规模	1024	2048	4096
	程序执行时间				加速比		
版本 1	0.04821	0.20404	1.34956	版本 1	13.73113	36.67951	42.12943
版本 2	0.01815	0.02859	0.15761	版本 2	36.47250	261.77965	360.73522
版本 3	0.01359	0.02447	0.06425	版本 3	48.71750	305.88019	884.91176

表 4: 不同任务划分方式程序执行时间和加速比（保留 5 位小数）

（3）结果分析

由表4中数据和图4.6(b) 可知，三种任务划分方式的加速比：版本 3（二维块划分）> 版本 2（一维块划分）> 版本 1（一维循环划分）。矩阵规模越大，加速比差距越明显。性能最优的二维块划分方式在 N=4096 时加速比达到了惊人的约 884!

由本实验可知，在大规模矩阵下，三种任务划分方式均实现了性能提升，且性能版本 3 最优、版本 2 次之，版本 1 相对最低。