南開大學
Nankai University

计算机学院

并行程序设计实验报告

GPU 优化高斯消元算法

姓名：张天歌

学号 :2211123

专业：计算机科学与技术

2024 年 6 月 16 日

目录

Abstract

本次实验主要完成了 SYCL 的学习，GPU 编熟悉程的范式和思想，进行 Lab Exercise 的代码实践；对于高斯消元部分进行 CUDA 加速优化，探索异构编程对于程序性能的影响。

关键字：并行；GPU；SYCL；高斯消元优化

1 Essentials of SYCL

1.1 oneAPI-Intro

1.1.1 SYCL 简介

SYCL 是一个开放标准，允许开发者使用现代 ISO C++（至少 C++ 17）编写异构计算和设备卸载代码，SYL 有较多优势：开放、基于标准的，支持多架构性能，避免厂商锁定，在 Nvidia GPU 上提供与原生 CUDA 相当的性能，C++ 语言的扩展，通过开源工具 SYCLomatic 加速代码迁移。

SYCL 的特性：

- 使用 SYCL 卸载计算到设备：修改原来 CPU 的 C++ 代码，使用 SYCL 将计算卸载到加速器设备。介绍了 SYCL 在设备选择、内存分配和提交计算任务方面的功能。
- SYCL 设备选择：使用 sycl::queue 来调度任务在设备上执行，可以选择不同的设备选择器来指定执行任务的设备类型。
- SYCL 内存分配：使用 sycl::malloc_shared 来分配可以被主机和设备访问的内存，数据移动是隐式的。
- 使用 SYCL 库卸载计算：介绍了如何使用 SYCL 库（如 oneDPL 或 oneAPI Data Parallel C++ Library）简化卸载计算到设备的过程，而无需深入了解 SYCL 编程。

1.1.2 Lab Exercise

课程提供了一个向量加法的练习，要求使用 SYCL 缓冲区和访问器概念来修改代码，将计算卸载到设备，根据 SYCL 提供的三个重要的功能：

设备选择用于卸载计算任务； 内存分配，以便主机和设备都能访问数据； 提交计算任务在设备上执行。

```
1  nclude <iostream>
2  nclude <sycl/sycl.hpp> // Include SYCL header
3
4  t main() {
5    // Create a SYCL queue using the SYCL 2020 device selector
6    sycl::queue q{sycl::default_selector_v};
7    std::cout << "Offload Device: " << q.get_device().get_info<sycl::info::device::name>() << "\n"
8
9    // Initialize some data array
10   const int N = 16;
11   float a[N], b[N], c[N];
```

```
12
13  for (int i = 0; i < N; i++) {
14      a[i] = 1;
15      b[i] = 2;
16      c[i] = 0;
17  }
18
19  // Allocate memory so that both host and device can access
20  {
21      sycl::buffer<float, 1> a_buf(a, sycl::range<1>(N));
22      sycl::buffer<float, 1> b_buf(b, sycl::range<1>(N));
23      sycl::buffer<float, 1> c_buf(c, sycl::range<1>(N));
24
25      // Submit computation to Offload device
26      q.submit([&](sycl::handler& h) {
27          // Create accessors
28          auto a_acc = a_buf.get_access<sycl::access::mode::read>(h);
29          auto b_acc = b_buf.get_access<sycl::access::mode::read>(h);
30          auto c_acc = c_buf.get_access<sycl::access::mode::write>(h);
31
32          // Define the kernel
33          h.parallel_for(sycl::range<1>(N), [=](sycl::id<1> i) {
34              c_acc[i] = a_acc[i] + b_acc[i];
35          });
36      }).wait(); // Wait for the computation to finish
37  }
38
39  // Print output
40  for (int i = 0; i < N; i++) std::cout << c[i] << "\n";
41
```

1.1.3 OneAPI 简介

oneAPI 旨在简化跨不同架构的开发，提供统一和简化的语言和库来表达并行性，支持 CPU、GPU、FPGA 等多种硬件，基于行业标准和开放规范，与现有的 HPC 编程模型互操作。

SYCL 和 DPC++

SYCL 是 C++ 的数据并行编程的行业标准化努力，类似于 OpenCL，由 Khronos Group 管理。它是一个跨平台的抽象层，使用 C++ 以单一源代码风格编写异构处理器的代码。DPC++ 是 oneAPI 对 SYCL 编译器的实现，利用现代 C++ 的生产力优势和熟悉的结构，结合 SYCL 标准进行数据并行和异构编程。

DPC++ 能够扩展 SYCL，提高生产力，简化表达，降低语言的□嗦和程序员的负担，还可以通过给程序员控制程序执行的能力和启用特定硬件的功能来提高性能。

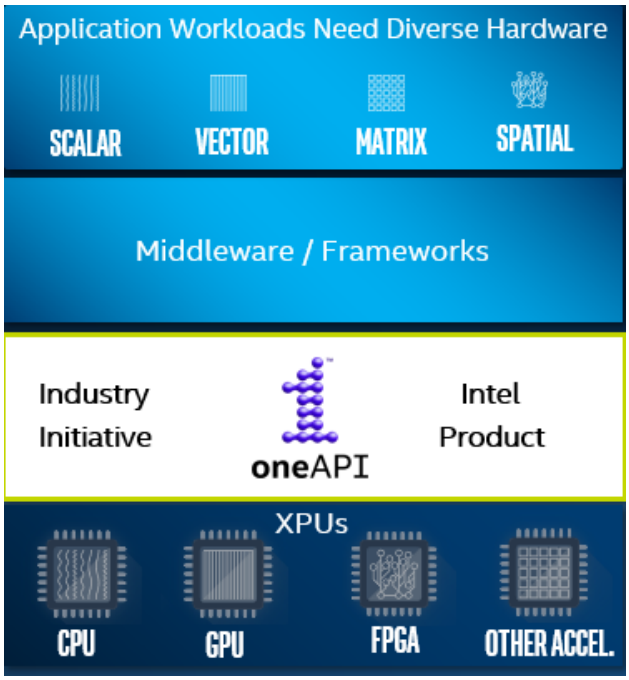


图 1.1: Oneapi 提供统一简化的库支持不同框架

HPC 单节点工作流与 oneAPI

在高性能计算领域，开发者可以根据自己的编程背景和需求，选择使用内核（SYCL）风格或指令（如 OpenMP）风格来加速他们的代码。Intel 提供了 DPC++ 兼容性工具，帮助从 CUDA 迁移到 SYCL，同时推荐使用 Intel® Advisor 工具来优化代码的向量化和内存管理，并识别适合加速器执行的循环，以提高性能。此外，还提供了针对不同起点的 HPC 开发者的推荐方法，以指导他们有效利用 oneAPI 和 SYCL 进行开发。OneAPI 的工作流如下图所示：

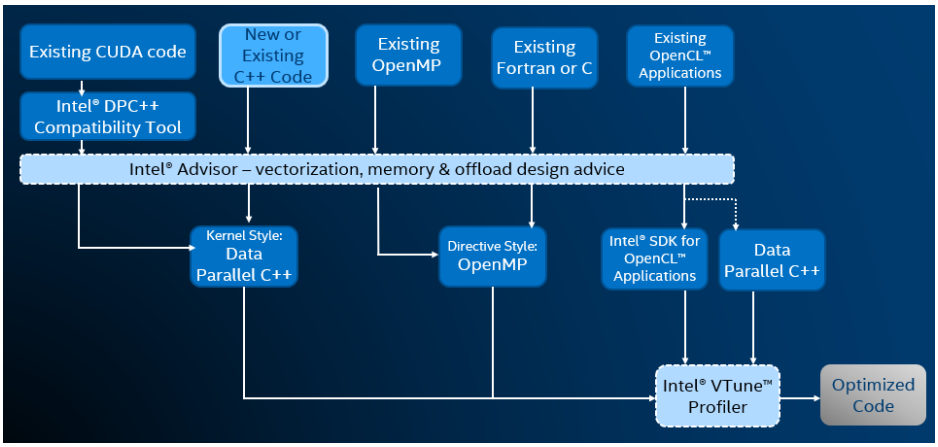


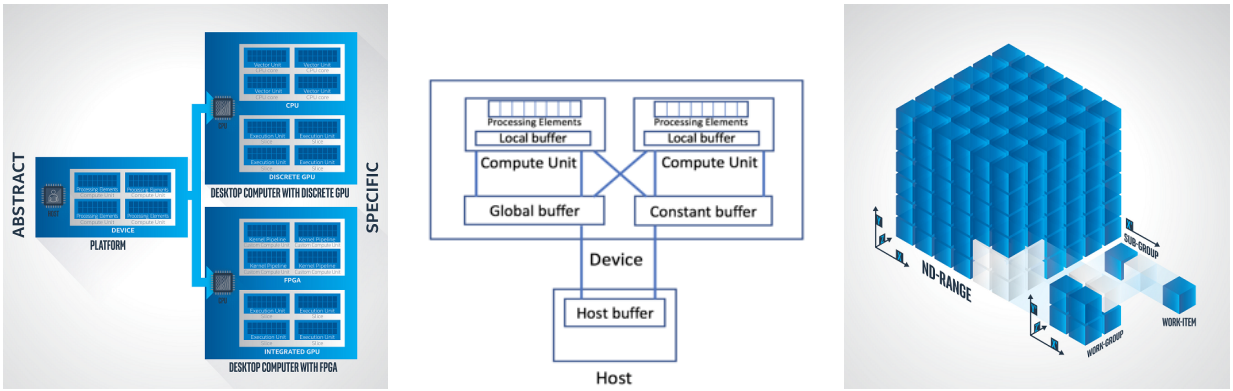
图 1.2: OneAPI 的工作流

Oneapi 的编程模型和如何编译运行项目

平台模型：oneAPI 的平台模型基于 SYCL* 平台模型。它指定控制一个或多个设备的主机。主机是计算机，通常是基于 CPU 的系统，执行程序的主要部分，特别是应用范围和命令组范围。主机协调并控制在设备上执行的计算工作。设备是加速器，是包含计算资源的专门组件，可以快速执行操作的子集，通常比系统中的 CPU 效率更高。

执行模型：执行模型基于 SYCL* 执行模型。它定义并指定代码（称为内核 kernel）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。命令组（由内核调用、访问器 accessor 等命令组成）被提交到执行队列。访问器 (accessor) 形式上是内存模型的一部分，它还传达执行的顺序要求。

内存模型：oneAPI 的内存模型基于 SYCL* 内存模型。它定义主机和设备如何与内存交互。它协调主机和设备之间内存的分配和管理。内存模型是一种抽象化，旨在泛化和适应不同主机和设备配置。



(a) oneAPI 平台模型

(b) ND-range、工作组、子组和工作项之间的关系图

(c) oneAPI 内存模型

图 1.3: OneAPI 编程模型

1.2 SYCL-Program-Structure

1.2.1 SYCL 基础类

SYCL（Standard Parallel Computing Language）是一个开放标准，允许开发者使用单一源代码编程异构设备。在本模块中，我们深入学习了以下几个基础类：

- Device: 表示系统中加速器的能力，可以查询设备的多种信息，如名称、供应商、版本等。
- Queue: 用于提交命令组给 SYCL 运行时执行，是工作提交到设备的机制。每个队列都关联一个设备，但同一个设备可以有多个队列。
- Kernel: 封装了在设备上执行的代码和数据。在 SYCL 中，用户不需要显式构造 Kernel 对象，而是在调用如 parallel_for 的内核调度函数时隐式构造。

1.2.2 设备选择

学习了如何使用标准设备选择器（如 default_selector_v、cpu_selector_v、gpu_selector_v）来选择执行内核的设备。此外，还探讨了如何编写自定义设备选择器，根据特定条件（如供应商名称、设备类型等）来选择设备。自定义选择器可以提高程序的灵活性和性能。

1.2.3 使用缓冲区、访问器、命令组处理器和内核编写 SYCL 程序

- 缓冲区（Buffer）：SYCL 中的缓冲区是对数据的抽象，可以表示一维、二维或三维的数据。缓冲区通过访问器来访问。

- 访问器（Accessor）：是访问缓冲区数据的机制，可以设置为只读、只写或读写模式，并具有同步数据的能力。
- 命令组处理器（Handler）：用于封装一系列命令，如内存分配、数据传输和内核执行。
- 内核（Kernel）：是在设备上并行执行的函数。我们学习了如何使用 parallel_for 函数来表达并行性，并通过 Lambda 表达式来编写内核代码。

1.2.4 使用主机访问器和缓冲区销毁进行同步

在异构计算中，数据在主机和设备之间同步是至关重要的，学习了两种主要的同步机制：

1. 主机访问器（Host Accessor）：允许在主机上同步和访问设备上的数据。创建主机访问器是一个阻塞调用，直到所有修改同一缓冲区的内核执行完成后才返回。
2. 缓冲区销毁: 当缓冲区超出作用域并被销毁时，它会自动将数据从设备同步回主机。

1.2.5 Lab Exercise

在 Code Sample 中通过一个复数乘法的示例，将所学知识综合应用，包括自定义数据类型、SYCL 并行执行以及结果验证。这个例子展示了如何将 SYCL 用于更复杂的数据结构和算法。在 Lab Exercise 中编写 SYCL 程序的示例，加深了对 SYCL 编程模型的理解，使用统一共享内存模型（USM）和缓冲区内存模型来实现**向量加法运算**。

```
1  nclude <sycl/sycl.hpp>
2  ing namespace sycl;
3  t main() {
4    const int N = 256;
5
6    //# Initialize a vector and print values
7    std::vector<int> vector1(N, 10);
8    std::cout<<"\nInput Vector1: ";
9    for (int i = 0; i < N; i++) std::cout << vector1[i] << " ";
10
11   //# STEP 1 : Create second vector, initialize to 20 and print values
12   //# YOUR CODE GOES HERE
13   std::vector<int> vector2(N, 20);
14   std::cout<<"\nInput Vector2: ";
15   for (int i = 0; i < N; i++) std::cout << vector2[i] << " ";
16
17   //# Create Buffer
18   buffer vector1_buffer(vector1);
19
20   //# STEP 2 : Create buffer for second vector
21   //# YOUR CODE GOES HERE
```

```
  buffer vector2_buffer(vector2);

  //# Submit task to add vector
  queue q;
  q.submit([&](handler &h) {
    //# Create accessor for vector1_buffer
    accessor vector1_accessor (vector1_buffer,h);

    //# STEP 3 - add second accessor for second buffer
    //# YOUR CODE GOES HERE
    accessor vector2_accessor (vector2_buffer, h, read_only);

    h.parallel_for(range<1>(N), [=](id<1> index) {

  //# STEP 4 : Modify the code below to add the second vector to first one
    vector1_accessor[index] += vector2_accessor[index];
    });
 });
//# Create a host accessor to copy data from device to host
host_accessor h_a(vector1_buffer,read_only);
//# Print Output values
std::cout<<"\nOutput Values: ";
for (int i = 0; i < N; i++) std::cout<< vector1[i] << " ";
std::cout<<"\n";
return 0;

```

运行结果正确，计算出相加结果为 30。

1.3 SYCL-Unified-Shared-Memory

USM 是 SYCL 中的基于指针的内存管理方式，对于习惯使用 C/C++ 中的 malloc 或 new 来分配内存的程序员来说，USM 提供了熟悉的编程模式，从而简化了将现有 C/C++ 代码移植到 SYCL 的开发过程。

1.3.1 USM 的类型和语法

USM 提供了显式和隐式两种内存管理模型：

- 设备（Device）：在设备上显式分配内存，仅在设备上可访问。
- 主机（Host）：在主机上隐式分配内存，主机和设备均可访问。
- 共享（Shared）：分配的内存可以在主机和设备之间迁移，主机和设备都可访问。

USM 的语法:
使用 malloc_shared 进行共享分配，示例如下:

```
t *data = malloc_shared<int>(N, q);
```

使用熟悉的 C++/C 风格 malloc 进行分配:

```
t *data = static_cast<int *>(malloc_shared(N * sizeof(int), q));
```

释放 USM 分配的内存:

```
ee(data, q);
```

1.3.2 USM 数据移动和使用

USM 数据移动:

- USM 隐式数据移动使用 malloc_shared 时，数据在主机和设备之间的移动是隐式的，这有助于快速实现功能，同时开发者不必担心内存的移动问题。
- USM 显式数据移动使用 malloc_device 时，开发者需要使用 memcpy 显式地在主机和设备之间移动数据，这种方式允许更细致地控制数据移动。

USM 的使用: 当抽象能够干净地应用在你的应用程序中，或者如果你的开发不受缓冲区干扰时，使用 SYCL 缓冲区；当移植 C++ 代码到 SYCL 时，USM 提供了一个熟悉的基于指针的 C++ 接口，可以最小化更改；当需要控制数据移动时，使用显式的 USM 分配。

1.3.3 USM 中的数据依赖性

在 Code Example 部分演示了如何使用 USM 和不同的数据依赖性管理方法，包括使用 in_order 队列属性、wait() 事件或 depends_on() 方法。

1.3.4 Lab Exercise

使用 USM 概念完成编码练习，包括创建 USM 设备分配、复制数据到设备、编写更新数据的内核任务、将数据从设备复制回主机并验证结果。

```
nclude <sycl/sycl.hpp>
ing namespace sycl;
atic const int N = 1024;

t main() {
queue q;
std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";
```

```
8
9  //intialize 2 arrays on host
10 int *data1 = static_cast<int *>(malloc(N * sizeof(int)));
11 int *data2 = static_cast<int *>(malloc(N * sizeof(int)));
12 for (int i = 0; i < N; i++) {
13   data1[i] = 25;
14   data2[i] = 49;
15 }
16
17 //# STEP 1 : Create USM device allocation for data1 and data2
18 //# YOUR CODE GOES HERE
19  int *dev_data1 = malloc_device<int>(N, q);
20  int *dev_data2 = malloc_device<int>(N, q);
21
22 //# STEP 2 : Copy data1 and data2 to USM device allocation
23 //# YOUR CODE GOES HERE
24  q.memcpy(dev_data1, data1, N * sizeof(int)).wait();
25  q.memcpy(dev_data2, data2, N * sizeof(int)).wait();
26
27
28 //# STEP 3 : Write kernel code to update data1 on device with square of its value
29 q.parallel_for(N, [=](auto i) {
30   dev_data1[i] = sycl::sqrt(static_cast<float>(dev_data1[i]));
31 });
32
33 //# STEP 3 : Write kernel code to update data2 on device with square of its value
34  q.parallel_for(N, [=](auto i) {
35   dev_data2[i] = sycl::sqrt(static_cast<float>(dev_data2[i]));
36 });
37
38 //# STEP 5 : Write kernel code to add data2 on device to data1
39 q.parallel_for(N, [=](auto i) {
40    dev_data1[i] += dev_data2[i];
41 });
42
43 //# STEP 6 : Copy data1 on device to host
44 q.memcpy(data1, dev_data1, N * sizeof(int)).wait();
45
46 //# verify results
47 int fail = 0;
48 for (int i = 0; i < N; i++) if(data1[i] != 12) {fail = 1; break;}
49 if(fail == 1) std::cout << " FAIL"; else std::cout << " PASS";
```

```
50 std::cout << "\n";
51
52 //# STEP 7 : Free USM device allocations
53 //# YOUR CODE GOES HERE
54  free(dev_data1, q);
55  free(dev_data2, q);
56
57 //# STEP 8 : Add event based kernel dependency for the Steps 2 - 6
58 return 0;
```

代码运行 pass 通过，正确实现了 USM。

1.4 SYCL-Sub-Groups

在许多现代硬件平台上，工作组中的一些工作项（work-items）可以同时执行或具有额外的调度保证。这些工作项的子集称为 Subgroups。利用 Subgroups 可以帮助将执行映射到低级硬件，并可能有助于实现更高的性能。

为什么要使用 Subgroups？

1. 子组中的工作项可以使用 shuffle 操作直接通信，无需显式内存操作。
2. 子组中的工作项可以使用子组屏障同步，并使用子组内存围栏保证内存一致性。
3. 子组中的工作项可以访问子组函数和算法，提供常见并行模式的快速实现。

1.4.1 sub group 信息

子组句柄可以查询以获取其他信息，例如子组中的工作项数，或工作组中的子组数，这对于使用子组实现内核代码的开发人员非常有用：

- get_local_id() 返回工作项在其子组内的索引
- get_local_range() 返回 sub_group 的大小
- get_group_id() 返回子组的索引
- get_group_range() 返回父工作组内的子组数

1.4.2 sub group 类和大小

可以使用 get_sub_group() 从 nd_item 获取子组句柄：

```
1 to sg = item.get_sub_group();
```

一旦获取了子组句柄，就可以查询有关子组的更多信息，执行 shuffle 操作或使用组算法

为了性能调整，可能需要将子组大小设置为特定值。例如，Intel(R) GPU 支持 8、16 和 32 的子组大小；默认情况下，编译器实现将选择最优的子组大小，但也可以强制使用特定值。

可以查询 GPU 支持的子组大小：

```
to sg_sizes = q.get_device().get_info<info::device::sub_group_sizes>();
```

1.4.3 sub group 主要功能

- Subgroup - Reduce：使用子组 reduce_over_group 函数对子组中的所有项执行归约。
- Subgroup - Broadcast：group_broadcast 函数使一个工作项能够与组中的所有其他工作项共享一个变量的值。
- Subgroup - Votes: any_of_group、all_of_group 和 none_of_group 函数（统称为“投票”函数）使工作项能够跨组比较布尔条件的结果。

1.4.4 Lab Exercise

下面是一个 sub group 的 exercise，运行结果 sum = 75712。

```
/# allocate USM shared allocation for input data array and sg_data array
int *data = malloc_shared<int>(N, q);
int *sg_data = malloc_shared<int>(N/S, q);

//# initialize input data array
for (int i = 0; i < N; i++) data[i] = i;
for (int i = 0; i < N; i++) std::cout << data[i] << " ";
std::cout << "\n\n";

//# Kernel task to compute sub-group sum and save to sg_data array
//# STEP 1 : set fixed sub_group size of value S in the kernel below

q.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item) {
  auto sg = item.get_sub_group();
  auto i = item.get_global_id(0);

  //# STEP 2: Add all elements in sub_group using sub_group reduce
  //# YOUR CODE GOES HERE
 int sg_sum = reduce_over_group(sg, data[i], std::plus<>());

  //# STEP 3 : save each sub-group sum to sg_data array
  //# YOUR CODE GOES HERE
  if (sg.get_local_id()[0] == 0) {
    sg_data[item.get_group(0)] = sg_sum;
  }
}).wait();

```

```
//# print sg_data array
for (int i = 0; i < N/S; i++) std::cout << sg_data[i] << " ";
std::cout << "\n";

//# STEP 4: compute sum of all elements in sg_data array
int sum = 0;
//# YOUR CODE GOES HERE
for (int i = 0; i < N/S; i++) {
  sum += sg_data[i];
 }

std::cout << "\nSum = " << sum << "\n";
//# free USM allocations
free(data, q);
free(sg_data, q);
```

2 CUDA 编程优化高斯消元

CUDA 是由 NVIDIA 推出的基于 GPGPU（General Purpose Computing on Graphics Processing Units）的并行计算平台，能够利用 NVIDIA 显卡的高性能并行计算能力，执行大规模复杂计算任务。

2.1 实验平台

- Visual Studio 2022 集成开发环境
- CUDA 11.7.1 windows10 版本
- 显卡：NVIDIA RTX3050Ti 移动端 GPU，CUDA 核心：2560，显存：4GB

2.2 CUDA 编程思想

首先对矩阵进行初始化，然后将其赋值到一维数组 *temp* 中，用于传输给 GPU 进行高斯消元，接下来的步骤包括：

- 调用 cudaMalloc() 函数在 GPU 端分配一个 sizeof(float)$\times N \times N$ 大小的内存空间，并以 *gpudata* 作为这一块空间的内存句柄。
- 使用 cudaMemcpy 函数将数据从 CPU 复制到 GPU 中。
- 使用 dimBlock() 和 dimGrid() 定义线程块和网格的维度。线程块的维度是 (BLOCK_SIZE, 1)，每个线程块包含 BLOCK_SIZE 个线程，但在 y 方向上只划分一个。线程网格 *dimGrid* 的维度是 $(1, 1)$，只有一个线程块组成线程网格。
- 使用 cudaEvent 计时器开始计时。

在 CUDA 优化代码中，定义了两个核函数 `division_kernel` 和 `eliminate_kernel`，分别用于高斯消元的除法和消元步骤。

`division_kernel` 核函数通过公式 $tid = \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x}$ 计算线程索引，并对矩阵对角线上的元素执行除法操作。`eliminate_kernel` 核函数根据线程索引 tx 进行矩阵消元，更新元素值。最终，使用 `cudaMemcpy()` 将结果传回 CPU，`cudaFree(gpudata)` 释放 GPU 内存。

2.3 实验结果和分析

实验结果如下表：（单位 ms）

CPU VS GPU	64	256	512	1024	2048	4096
CPU_time	0.19	10.63	71.13	210.78	1748.35	4850.21
GPU_time	5.01	6.56	12.85	20.35	58.33	118.30
加速比	0.04	1.68	5.86	10.55	30.01	41.00

表 1: CUDA 加速运行时间表

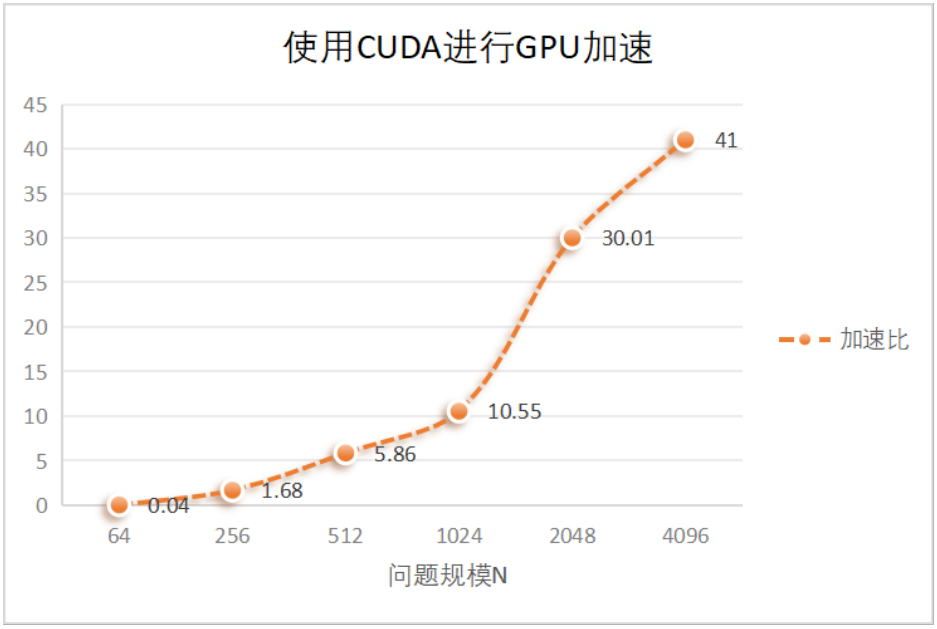


图 2.4: 使用 CUDA 进行 GPU 加速

根据实验结果和对图的分析，得出以下结论：

- **初期加速不明显：**在数据规模较小（例如 N=64）时，GPU 加速效果并不明显，甚至可能比 CPU 慢。这可能是因为小规模数据下，CPU 的计算能力已经足够处理，而 GPU 在这种情况下需要进行额外的线程管理和任务分配，导致效率降低。
- **加速比的非线性增长：**随着数据规模的增加，GPU 的加速比呈现出非线性增长的趋势。这表明 GPU 在处理大规模数据时，其并行处理的优势逐渐显现，加速效果越来越明显。
- **GPU 的计算优势：**GPU 拥有大量计算单元和高内存带宽，这使得它在处理大数据量时能够一次性读取和处理更多数据，从而实现高吞吐量和快速计算，达到最高加速比 41.0。相比之下，CPU 的内存带宽较小，数据读取能力有限，导致在大规模数据处理时性能受限。

- **架构差异的影响：**GPU 的架构天然适合并行计算，特别是在面对大量数据时，其并行加速效果尤为明显。而 CPU 由于架构和内存带宽的限制，在并行计算方面的表现不如 GPU。

3 总结与反思

本次实验进行了 SYCL 的课程学习，了解了 GPU 编程的基本范式并进行了简单的代码实践，在 SYCL 基础上又学习了 intel 架构下的一些并行编程工具，这对于编程实践有很大帮助；此外，本次实验采用 cuda 编程优化高斯消元算法，并进行了 gpu 和 cpu 的对比实验，得出结论 gpu 在大规模计算时算力和性能上都要更优于 cpu。

在 SYCL 课程学习时，进度较慢但是收获颇丰，学习了异构编程的思想和代码结构，对于一些编程工具有了更好的掌握，并且知道了 gpu 编程对于大规模计算的强有力优势。

代码详见：GPU 优化实验