南開大學
Nankai University

计算机学院

并行程序设计作业报告

GPU 编程

姓名：徐文斌

学号：2010234

专业：计算机科学与技术

2022 年 6 月 15 日

目录

1 oneAPI 与 DPC++ 学习 **2**
1.1 实验介绍 . . . 2
1.2 oneAPI 编程模型概述 . . . 2
1.2.1 oneAPI 是什么 . . . 2
1.2.2 SYCL 规范 . . . 3
1.2.3 DPC++ 编程语言 . . . 3
1.3 oneAPI 编程框架 . . . 3
1.3.1 平台模型 . . . 3
1.3.2 执行模型 . . . 4
1.3.3 内存模型 . . . 4
1.3.4 内核编程模型 . . . 5
1.4 学习过程截图 . . . 5

2 NVIDIA 云端课程学习 **6**
2.1 实验介绍 . . . 6
2.2 Accelerating Applications with CUDA C/C++ 章节学习 . . . 6
2.2.1 核函数的声明和调用 . . . 6
2.2.2 CUDA 线程层次结构 . . . 7
2.2.3 CUDA 线程索引 . . . 7
2.2.4 CUDA 内存分配 . . . 8
2.2.5 CUDA 错误处理 . . . 8
2.3 Managing Accelerated Application Memory with CUDA C/C++ Unified Memory and nsys 章节学习 . . . 8
2.3.1 nsys 的使用 . . . 8
2.3.2 SM（Streaming Multiprocessor）的相关知识 . . . 8
2.3.3 Unified Memory 的一些细节 . . . 9
2.3.4 使用异步内存预取机制加速程序性能 . . . 9
2.4 Asynchronous Streaming, and Visual Profiling for Accelerated Applications with CUDA C/C++ 章节学习 . . . 9
2.4.1 nsys 图形化 profiling 使用 . . . 10
2.4.2 并发 CUDA 流 . . . 10
2.5 NVIDIA 网课学习总结与感想 . . . 11
2.6 普通高斯消去 CUDA 优化 . . . 11

3 实验源码项目 GitHub 链接 **12**

1 oneAPI 与 DPC++ 学习

1.1 实验介绍

本次实验我通过学习 DevCloud 平台 oneAPI 课程对 oneAPI 的基础知识有了一个大致的了解，下面我将说明本次学习过程和自己学习到的内容。

1.2 oneAPI 编程模型概述

首先是 oneAPI 的一些概念理解，此前一直理不清 oneAPI、SYCL 和 DPC++ 三者的关系，最终经过课程学习和网上查阅对这三者有了一个大致的理解。

1.2.1 oneAPI 是什么

在当今数据处理的异构架构下，不同的设备如 GPU 等的工作负载正在增长。而不同的数据处理的硬件往往需要使用不同的语言和库进行编程开发。我们却缺少通用的编程语言或 API，这就需要我们为不同的硬件维护单独的代码库。对于开发人员来说，由于缺少一致的跨平台工具，编程人员必须学习一整套不同的工具。此外，为每个硬件平台开发软件需要单独的投资，而在一中硬件架构下所做的工作几乎无法被重用在其他不同的架构下。开发人员还必须考虑各种硬件的细节需求。这些无疑对我们异构架构下软件的开发带来了许多问题。

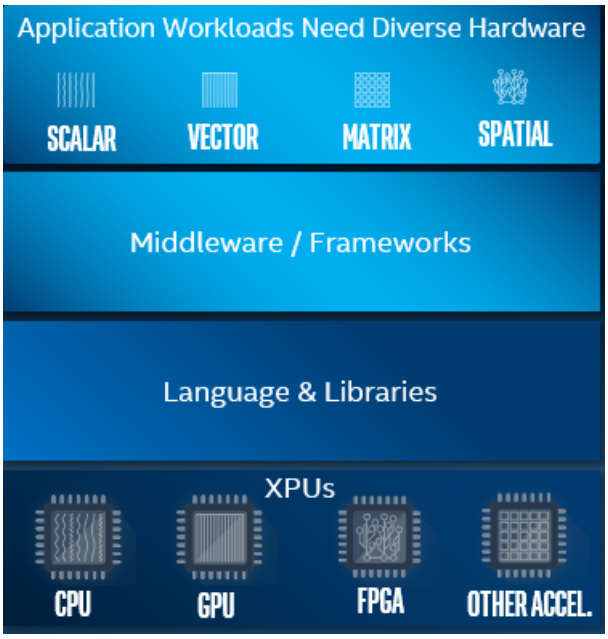

图 1.1: 传统异构编程模式

Intel oneAPI 正是为了解决上述问题而被开发出来的一种编程模型。其旨在提供一个适用于各类计算架构的统一编程模型和应用程序接口。通过使用 oneAPI 进行软件开发，应用程序的开发者只需要开发一次代码，就可以让代码在跨平台的异构系统上执。oneAPI 既可以大大地提升开发者的开发效率，又可以减少开发成本。

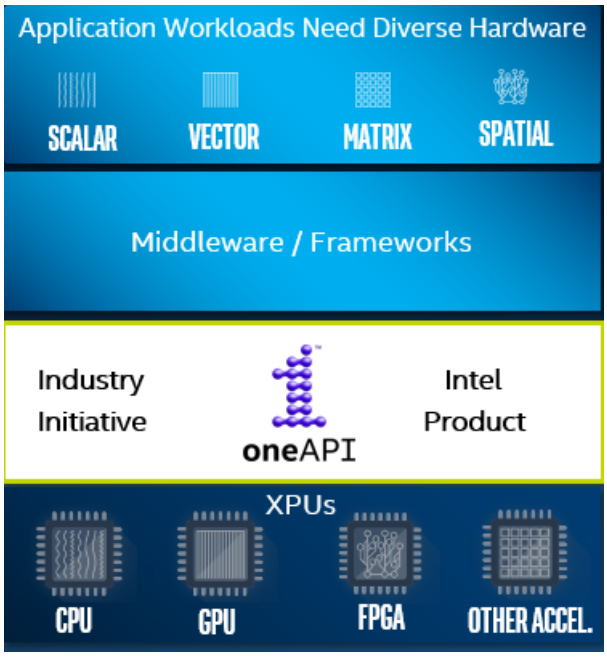

图 1.2: oneAPI 编程模式

1.2.2 SYCL 规范

SYCL 是一项 Khronos 标准，是一种基于 C++ 异构并行编程框架，用来加速高性能计算，机器学习，内嵌计算，以及在相当宽泛的处理器构架之上的计算量超大的桌面应用。SYCL 中增加了 oneAPI DPC 库（oneDPL），它为编程加速器提供了类似 STL 的功能。

1.2.3 DPC++ 编程语言

Data Parallel C++ (DPC++) 是 oneAPI 实现的一种全新的跨架构编程语言，DPC++ 基于 C++ 编写，由一组 C++ 类、模板与库组成，同时兼容 Kronos 的 SYCL 规范。DPC++ 是一种单一源码语言，其中主机代码和异构加速器内核可以混合在同一个源文件中。在主机上调用 DPC++ 程序并将计算卸载到加速器。程序员使用熟悉的 C++ 和库结构，并添加诸如工作目标队列、数据管理缓冲区和并行性并行的函数，以指导计算和数据的哪些部分应该被加载。

此外，Intel DPC++ 兼容性工具可以实现将 CUDA 代码迁移到 DPC++ 上，这在很大程度上减轻了开发人员代码移植的负担。

1.3 oneAPI 编程框架

oneAPI 有四个编程框架，分别为平台模型、执行模型、内存模型和内核编程模型，下面我将简要讲述自己对这四中模型的理解。

1.3.1 平台模型

我认为，平台模式和 CUDA 编程模式是较为相似的，只是设备不止局限于 GPU。具体来说，平台模式下，我们首先需要指定一个 Host 主机，由该主机执行程序的主要部分，且该主机负责进行对各个 Device 设备的协调和控制。Host 往往由 CPU 来担任，Device 则可以是 GPU、FPGA 等加速设备。当然，这不是固定的，我们也可以将 CPU 作为 Device。

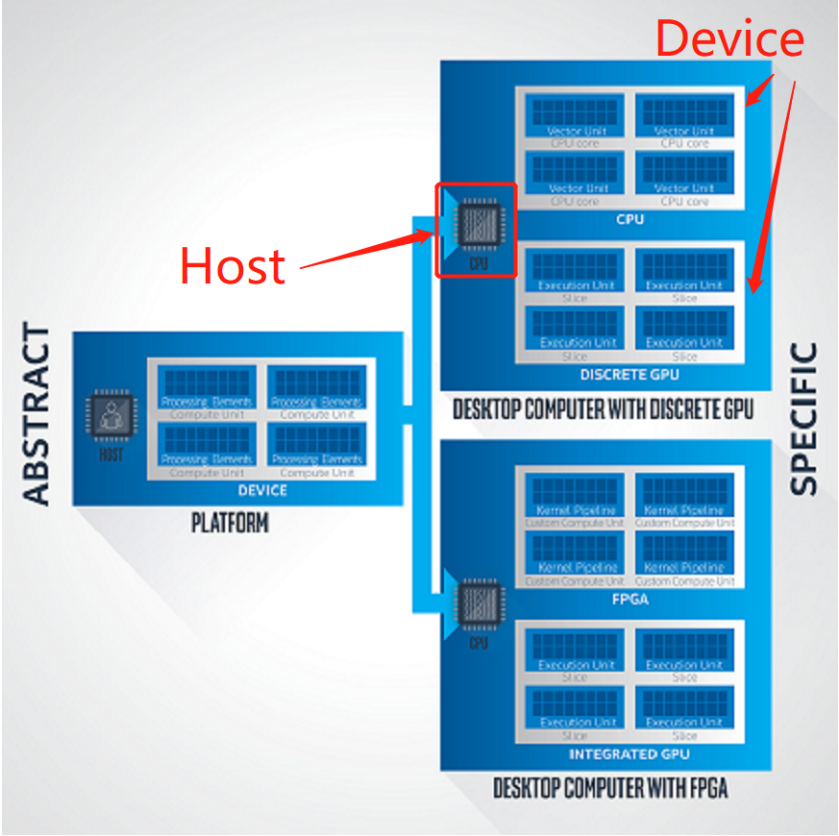

图 1.3: oneAPI 平台模型

1.3.2 执行模型

执行模型定义并指定代码（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。我对此部分理解还不够透彻，因此在此不对此模型做过多解释。

1.3.3 内存模型

oneAPI 的内存模型定义了主机和设备如何与内存进行交互，协调主机和设备之间的内存分配和管理。为了在不同的主机和设备都适用，内存模型是一种抽象化模型。在内存模型中，内存被通过以内存对象的形式指定，内存可以驻留在 Host 或者 Device 中，而 Host 和 Device 中的内存都被抽象成了内存对象。不同设备之间内存的交互也是以内存对象交互的形式进行的。内存模型中的内存对象有两种，分为 buffer 类型和 image 类型。此外，内存对象之间的通信需要通过访问器（accessor）对象来完成，访问器负责传达所需访问的位置（主机/设备）以及访问的模式（读/写）。

考虑一个具体的例子，当 Host 端申请一段内存时，在 Host 端将会创建一个 buffer 类型的内存对象。而当前 Device 和 Host 端进行通信时，访问器会将 Host 中 buffer 对象的类型、数目以及允许的访问模式传递给 Device 端，从而便于 Host 端和 Device 端的通信。

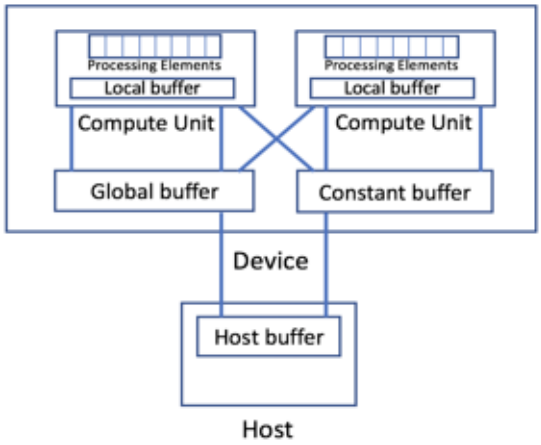

图 1.4: oneAPI 内存模型

1.3.4 内核编程模型

oneAPI 编程模型支持主机和设备之间的显示并行性。之所以是显示并行性，是因为 Host 端和 Device 端执行的代码不是自动分配的，而是程序员手动指定的。这一点与 CUDA 编程类似，由程序员来决定在 Device 端执行的核函数。我们在上面已经提到了，DPC++ 是单一源代码编程，Host 端的代码和 Device 端的代码可以在一个文件中，这也与 CUDA 编程类似。此外，与 CUDA 编程类似的是，Host 端代码的编写细节和 Device 端代码的编写细节也有些许差别。

1.4 学习过程截图

图1.5为 DevCloud 平台上《SYCL Hello World》实验的学习截图。

图 1.5: 学习截图

2 NVIDIA 云端课程学习

2.1 实验介绍

本次 GPU 实验中，我对 NVIDIA 的 Fundamentals of Accelerated Computing with CUDA C/C++ 云端课程进行了学习。此课程共分为三个部分，第一个部分主要讲述了 GPU 的架构和 CUDA 编程的一些基础概念和语法，后两个部分则通过向我们介绍更多的细节来一步一步引导我们对 CUDA 程序进行更进一步的优化。完成课程学习后，我使用本地的 NVIDIA GTX 1650Ti 显卡进行了普通高斯消去算法的 CUDA 编程优化实验。下面我将首先大致讲述每个章节学习过程和学习内容，最后再附上实验结果。

2.2 Accelerating Applications with CUDA C/C++ 章节学习

第一部分 Accelerating Applications with CUDA C/C++ 大致讲述了 CUDA 的编程风格，包括核函数如何声明和调用、CUDA 线程层次结构、CUDA 如何索引线程以及典型的线程任务分配方式、CUDA 内存分配、错误处理等等基本知识。下面我将大致讲述关于这几部分所学到的内容。

2.2.1 核函数的声明和调用

```
//函数声明
void CPUFunction() //CPU 函数
{
    printf("This function is defined to run on the CPU.\n");
}

__global__ void GPUFunction() //核函数
{
    printf("This function is defined to run on the GPU.\n");
}

int main()
{
    //函数调用
    CPUFunction();
    GPUFunction<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

核函数指的是在 GPU 端运行的函数，CUDA 核函数的声明与 C++ 函数的声明的主要区别有两个：

- 1 CUDA 核函数声明需要在开头添加 ___global___ 关键字，以标识该函数为核函数；
- 2 核函数的返回值必须为 void 类型，推测可能的原因有两个，一是核函数的执行和 Host 端程序的执行是并行的，如果不调用同步函数，Host 端无法得知核函数什么时候运行结束，这给使用核

函数的返回值带来一定的困扰；二是核函数将被 Device 端的多个线程执行，难以判定获得的返回值以哪一个线程为准。

函数调用方面，CUDA 核函数的调用必须指定执行配置，即代码中的 « <...> » 部分。以 GPU-Function« <1,1> »(); 为例，第一个 1 表示启用 1 个 block，第二个 1 表示每个 block 中启用 1 个线程。

需要注意的是，在调用过核函数后，核函数的执行与 Host 端是异步的，Host 端并不会等待核函数执行结束，而是会继续执行程序，因此我们需要调用 cudaDeviceSynchronize() 函数来使 Host 端显示地等待核函数执行结束。

我们可以使用 CUDA 的 nvcc 编译器对 CUDA 代码进行编译，其编译选项与 g++ 编译器类似，典型的编译指令如下所示，该指令编译并运行 CUDA 程序。

```
nvcc -arch=sm_70 -o out some-CUDA.cu -run
```

2.2.2 CUDA 线程层次结构

CPU 的一个进程内部可以开辟多个线程，每个线程可以运行线程函数，并行执行。与 CPU 类似的，GPU 中每个线程并行运行核函数。但 GPU 的线程层次结构较为特殊，若干线程组成一个 block，而若干个 block 组成一个 grid。我们在调用核函数时的执行配置的第一个参数就是对 grid 中 block 的结构的定义，第二个参数是每个 block 中的线程结构的定义，这就使得每个 block 中线程数量是相同的。

CUDA 还支持多维的 grid 和 block 的定义，最高支持定义三维的 gird 和 block，这需要使用 CUDA 的 dim3 类型数据，使用方式如下代码所示。多维 grid 和 block 的定义有利于我们对多维数据如矩阵等的处理。

```
dim3 threads_per_block(16, 16, 1);
dim3 number_of_blocks(16, 16, 1);
someKernel<<<number_of_blocks, threads_per_block>>>();
```

2.2.3 CUDA 线程索引

为了使不同的线程可以处理不同的数据，我们需要解决如何标识线程的问题。CUDA 中可以通过 threadIdx.x 变量来查询执行核函数的当前线程在所属 block 内的标号；通过 blockIdx.x 变量来查询当前线程所属的 block 在整个 grid 中的标号；通过 blockDim.x 变量查询线程所属的 block 中共有多少个线程。这样一个 grid 中的每个线程 ID 可以表示为如下公式：

```
index = threadIdx.x + blockIdx.x * blockDim.x;
```

多维情况下表达式也是类似的。

有了线程的索引方式，我们便可以对线程进行任务分配，根据具体的线程索引分配给线程相应的待处理数据。当线程数多于总的数据时，我们可以给一部分线程分配数据。当总数据多于总的线程数时，一个线程需要处理多个数据，这时我们可以将数据进行对线程的循环分配，我们使用 gridDim.x * blockDim.x 来定义其步长，其中 gridDim.x 变量存储 grid 中的 block 数目。

2.2.4 CUDA 内存分配

由于使用 malloc() 函数分配的内存只能在 CPU 端使用，CUDA 提供了新的内存分配函数 cudaMallocManaged()，其分配的内存可以在 Host 端和 Device 端共享并互相迁移，且移动过程可以自动实现，这给了程序员极大的便利。注意，使用 cudaMallocManaged() 函数分配的内存需要使用 cudaFree() 函数释放。

2.2.5 CUDA 错误处理

我认为这是本章节十分重要的一个内容。在传统的 C++ 编程中，如果程序出错较容易调试；但是 CUDA 编程中调试却相对困难，因此 CUDA 的错误处理较为重要。CUDA 中的一些函数的返回值为 cudaError_t 类型，我们可以接收这些函数的返回值，判断是否成功执行，若执行出错，我们可以利用该返回值得到错误信息。而我们在上边提到，核函数的返回值为 void 类型，因此如何判断核函数是否正确执行？CUDA 提供了 cudaGetLastError() 函数，可以获得上一次错误信息，返回 cudaError_t 类型的值，我们可以通过该函数来判断核函数是否正确运行。使用方法如下所示：

```
someKernel<<<1, -1>>>();  // -1 不是一个合法的线程数
cudaError_t err;
err = cudaGetLastError(); //得到上一次错误信息
if (err != cudaSuccess)
{
    printf("Error: %s\n", cudaGetErrorString(err));
}
```

2.3 Managing Accelerated Application Memory with CUDA C/C++ Unified Memory and nsys 章节学习

第二部分课程主要讲述了 CUDA 中 Profile 工具 nsys（Nsight Systems）的使用方式、SM（Streaming Multiprocessor）的一些相关知识、Unified Memory 的一些细节以及异步内存预取机制，从而引导着我们一步步地对程序进行更进一步地优化。

2.3.1 nsys 的使用

为了能够更好地加速程序的运行，对程序的 profile 是很有必要的，CUDA 向用户提供了 profile 工具 nsys，使用该工具我们可以轻松地得到 CUDA 程序运行时 CUDA API、CUDA 核函数、CUDA 内存操作等的统计数据。我们可以简单地使用如下指令运行 nsys 工具，这将会生成一个程序分析文件。

```
nsys profile --stats=true ./your_executable_file
```

2.3.2 SM（Streaming Multiprocessor）的相关知识

SMs（Streaming Multiprocessors）是 GPU 中的处理单元。当核函数执行时，block 交由 SMs 执行。GPU 每批次可以执行 SM 的数量个 block，因此，为了充分利用 SMs，我们编程时建议设置 block

的数量为 SM 数量的倍数，这样可以充分利用 SMs 而不留下空闲 SM。此外，SMs 在一个称为 warp 的块中创建、管理、调度和执行 32 个线程的分组。因此，设置每个 block 中的线程数量为 32 的倍数也有利于充分利用硬件单元。

不同的 GPU 中 SM 的数量可能不同，我们可以以编程的方式查看 GPU 中的 SM 数量，如下所示：

```
int deviceId;
int numberOfSMs;
cudaGetDevice(&deviceId); //获得 GPU 设备 ID
//获得 GPU 中 SM 的数量
cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);
```

2.3.3 Unified Memory 的一些细节

在课程第一部分中我们学习了一些 CUDA 的内存分配机制，使用 cudaMallocManaged() 函数分配统一内存，是 GPU 和 CPU 都可以使用其分配的内存，这种方式分配出的内存就是 Unified Memory，其可以自动地在 GPU 和 CPU 之间迁移。本部分课程则具体讲述了统一内存迁移是如何实现的。大致来讲，Unified Memory 实现了一种按需迁移的内存迁移方式，即当 Host 或者 Device 尝试访问某些内存数据但是该内存数据并未驻留在相应设备中时，将会发生页缺失，之后将会触发内存的迁移。这种机制对于程序运行前无法知道需要实际处理哪些数据时十分有效。

2.3.4 使用异步内存预取机制加速程序性能

UM 内存迁移机制虽然使用十分方便，但是页缺失以及按需内存迁移仍然会给程序带来不小的开销。为了尽量减少内存数据迁移对程序带来的性能影响，CUDA 提供了异步内存预取机制。对于那些事先知道的，且数据在内存中存储十分连续的数据内存迁移，我们可以通过调用 CUDA 的 cudaMemPrefetchAsync() 函数来实现在后台将 UM 内存迁移到 CPU 或 GPU，这样就可以减少内存页缺失和按需内存迁移的开销。预取过程中每次传输的数据块较大，数据传输次数较少。

如下是 CUDA 异步内存预取示例代码：

```
int deviceId;
cudaGetDevice(&deviceId);
//将数据异步预取到 GPU 中
cudaMemPrefetchAsync(pointerToSomeUMData, size, deviceId);
//将数据异步预取到 CPU 中
cudaMemPrefetchAsync(pointerToSomeUMData, size, cudaCpuDeviceId);
```

2.4 Asynchronous Streaming, and Visual Profiling for Accelerated Applications with CUDA C/C++ 章节学习

课程的最后一个章节，本章节中，我感到收获最大的是学习到了 nsys 图形化 profiling 的使用和了 CUDA 的 non-default 流机制。

2.4.1 nsys 图形化 profiling 使用

与我们之前使用的 profile 工具类似，nsys 也支持图形化显示，我们可以使用 nsys 工具分析 CUDA 程序运行的时间线，从而对 CUDA 程序运行过程中的各种行为及其消耗时间有一个更加清晰的认识，比如内存迁移过程、不同流中函数的执行顺序等等。从而使我们有一个更加明确的程序性能优化的方向。如图2.6所示，图形化工具的具体使用方法不再赘述。

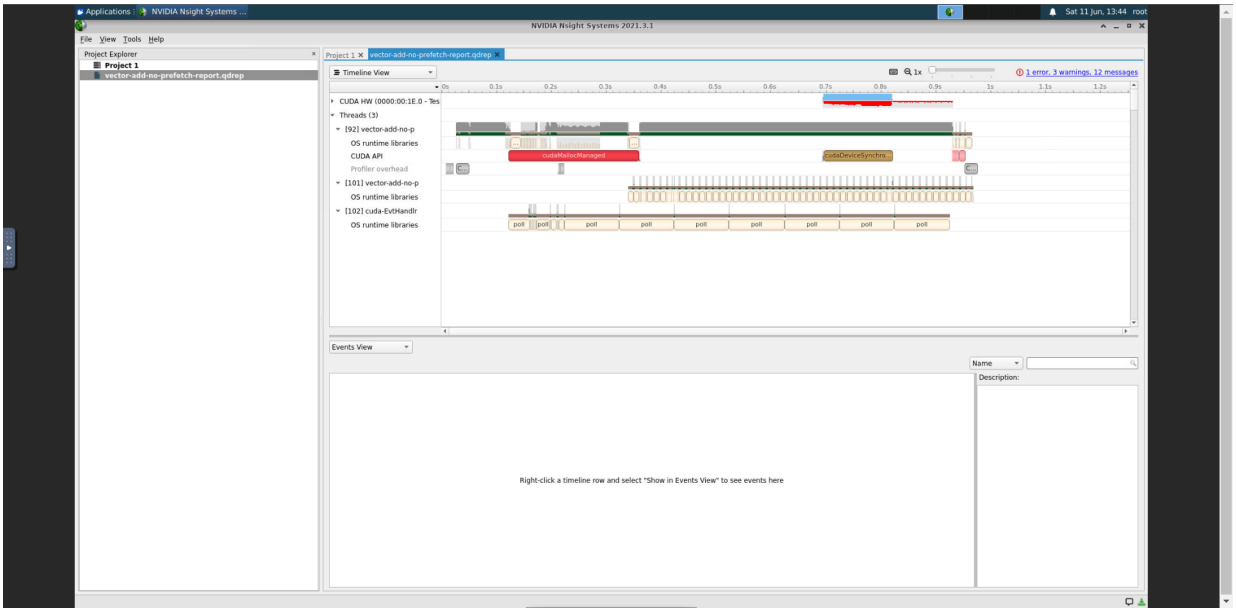

图 2.6: nsys 图形化分析工具

2.4.2 并发 CUDA 流

流是一个指令序列，在一个流中指令全部顺序执行，不可并行执行。CUDA 中存在一个默认的 default 流，核函数默认在该流中执行。为了提高并行性，CUDA 允许我们创建 non-default 流来运行核函数。在一个流中的若干核函数必须串行执行，但是在不同的 non-default 流中的核函数可以并行执行。而默认 default 流是比较特殊的，当默认流中执行核函数时，必须等待先于该核函数执行的其他 non-default 流中的核函数执行完才可以执行。而如果其他的 non-default 流希望执行核函数，也必须等待 default 流中的核函数执行完才可以执行。简而言之，default 流中函数的执行不会和 non-default 流中函数的执行并行进行。我认为这相当于向我们提供了一种同步机制。因此，如果我们想并行执行某些核函数，我们只需创建若干 non-default 流，在这些流中运行核函数即可。这样，我们再次使程序性能得到了提升。

如下是 non-default 流的创建和使用示例：

```
1 cudaStream_t stream;        // 定义流类型变量
2 cudaStreamCreate(&stream); //  调用函数创建 non-default 流
3 //将流变量作为核函数的执行配置的第四个参数，第三个参数不作讨论
4 someKernel<<<number_of_blocks, threads_per_block, 0, stream>>>();
5 cudaStreamDestroy(stream); // 记得销毁 non-default 流
```

2.5 NVIDIA 网课学习总结与感想

通过本次学习，我学习了 CUDA 编程的一些基本知识，如 CUDA 的一些基本语法及 API、GPU 的架构、UM 统一内存机制、nyns 优化工具的使用等等。从一开始的学习编写 CUDA 程序到一步步学习如何优化 CUDA 程序。课程学习中我也进一步认识到了提升程序的并行性和减少额外开销可以对程序性能带来较好的提升。我更加了解了并行编程的思想，也对 CUDA 语言的设计感到惊叹，其并不需要我们学习太多的语法知识，通过合理地利用 GPU 的各种机制我们便可以使程序得到性能得到较好的提升。此外，我也更加认识到了 profile 工具的重要性，一个好的 profile 工具对于我们并行程序设计有着十分大的帮助，可以对我们的优化起到一定的指导作用，通过不断的修改代码-profiling 的迭代，我们一步一步对程序进行优化。最后，网课的学习也锻炼了我的英文阅读能力，锻炼了我的耐心。

课程学习结果如图2.7所示。

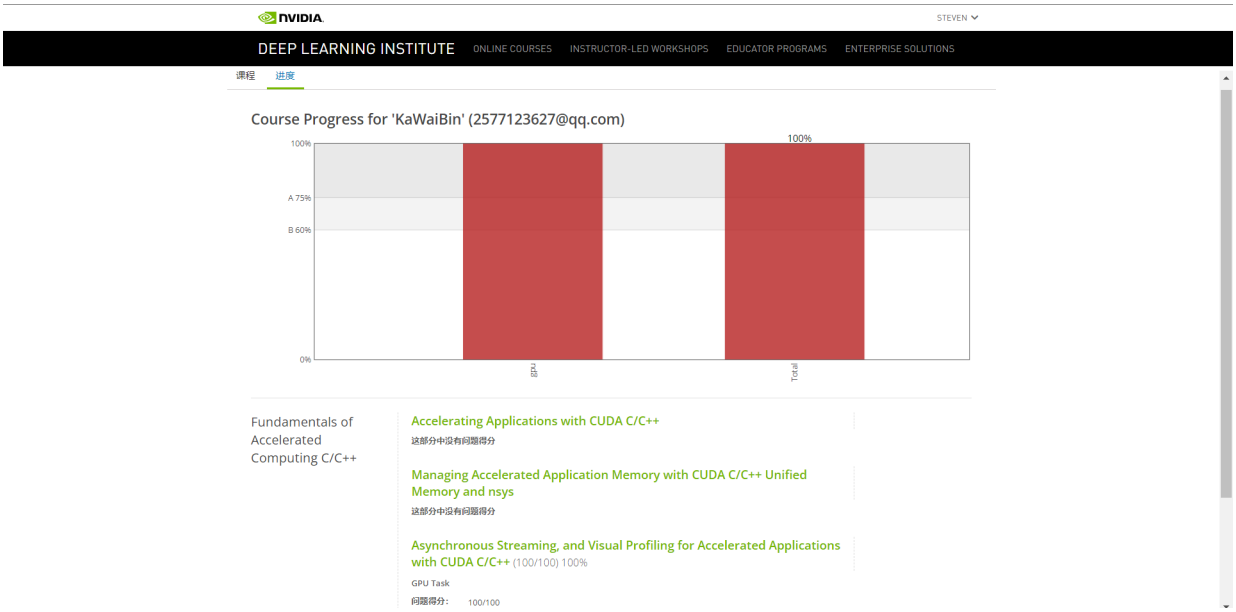

图 2.7: 课程学习截图

2.6 普通高斯消去 CUDA 优化

学习完 CUDA 课程后我使用学到的知识进行了普通高斯消去的 CUDA 优化实验，简单编写了普通高斯消去的 CUDA 程序，具体代码请见文末 Github 链接。程序运行环境为 x86 架构、Windows10 操作系统，显卡型号为 NVIDIA GTX 1650TI。

将不同问题规模下普通高斯消去算法耗时与 CUDA 优化高斯消去算法耗时进行比较，结果如下表1：

算法\规模	64	128	256	512	1028	2048
普通算法	0.447	4.307	27.859	221.831	1085.004	14807.893
CUDA 优化算法	2.583	6.986	12.420	30.507	110.051	1129.667

表 1: 普通算法与 CUDA 优化算法运行时间 (ms)

可以看到，当数据规模较小时，CUDA 优化算法运行时间要大于普通算法，这是很显然的，因为数据规模较小时，数据在 Host 端和 Device 端之间移动的开销所占比重较大。而当问题规模较大时，CUDA 算法达到了近 10 倍的加速比，得到的优化是十分显著的。由此可见 GPU 架构在并行数据运

算中的优势。

3 实验源码项目 GitHub 链接

普通高斯消去 CUDA 优化程序项目地址