



南開大學

Nankai University

计算机学院
并行程序设计报告

基于 Devcloud OneAPI 平台的 GPU
编程学习

姓名：赵元鸣

学号：2211757

专业：计算机科学与技术

2024 年 6 月 12 日

目录

1	基本要求	2
1.1	SYCL_oneApi_intro	2
1.1.1	学习内容	2
1.1.2	学习结果	3
1.2	SYCL_Program_Structure	4
1.2.1	学习内容	4
1.2.2	学习结果	5
1.3	Unified_Shared_Memory	7
1.3.1	学习内容	7
1.3.2	学习结果	8
2	进阶课程学习	10
2.1	Sub_Groups	10
2.1.1	课程内容	10
2.1.2	练习题答案与结果	11
2.2	Intel_Advisor	12
2.2.1	Roofline Analysis	12
2.2.2	Offload Advisor	13

1 基本要求

1.1 SYCL_oneApi_intro

1.1.1 学习内容

oneAPI 编程模型提供了一套全面且统一的开发者工具组合，可用于跨硬件目标的开发。这些工具包括一系列性能库，覆盖了多个工作负载领域。每个库中的函数都针对不同的目标架构进行了定制编码，因此相同的函数调用可以在支持的架构上提供优化的性能。DPC++ 基于行业标准和开放规范，旨在鼓励生态系统的协作和创新。

目前，在数据中心领域，专用工作负载的增长显著。每种类型的数据中心硬件通常需要使用不同的语言和库进行编程，因为没有通用的编程语言或 API，这需要维护独立的代码库。开发者必须学习一整套不同的工具，因为各平台的工具支持不一致。oneAPI 旨在提供统一的编程模型，以简化跨多种架构的开发。它包括统一且简化的语言和库，用于表达并行性。

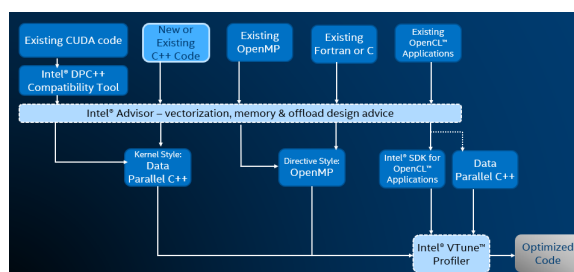


图 1.1: 优化过程

而 SYCL 在向行业标准化努力，支持 C++ 的数据并行编程。SYCL 标准由 Khronos Group 管理，构建在 OpenCL 之上，提供跨平台抽象层，使得异构处理器代码可以用 C++ “单源”风格编写。与 OpenCL 不同，SYCL 包含模板和 lambda 函数，使高层次应用软件可以干净地编码，同时优化内核代码的加速。

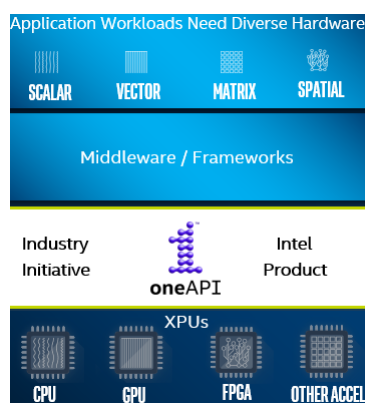


图 1.2: OneAPI 实现的行业标准化

SYCL 的执行模型定义了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与设备之间的执行和数据管理。队列可以按顺序或无序执行，由程序控制。内核代码代表了一个工作项的执行内容，代码外部控制并行执行的数量和分布。内存模型基于 SYCL 内存模型，定义了主机和设备如何与内存交互。内存分配和管理通过缓冲区和访问器进行协调。内核编程模型基于 SYCL 内核编程模型，支持主机与设备之间的显式并行。

演示代码

```
1
2 #include <sycl/sycl.hpp>
3 using namespace sycl;
4 static const int N = 16;
5 int main(){
6     // # define queue which has default device associated for offload
7     queue q;
8     std::cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";
9     // # Unified Shared Memory Allocation enables data access on host and device
10    int *data = malloc_shared<int>(N, q);
11    // # Initialization
12    for(int i=0; i<N; i++) data[i] = i;
13    // # Offload parallel computation to device
14    q.parallel_for(range<1>(N), [=] (id<1> i){
15        data[i] *= 2;
16    }).wait();
17    // # Print Output
18    for(int i=0; i<N; i++) std::cout << data[i] << "\n";
19    free(data, q);
20    return 0;
21 }
```

接着课程向我们展示了一段代码，告诉我们如何在设备上进行并行计算与内存管理。程序创建一个 SYCL 队列，选择默认计算设备。用统一共享内存（USM）为大小为 16 的整数数组分配内存，然后在设备上并行地将数组中的每个元素乘以 2。再输出计算后的数组元素并释放内存。

1.1.2 学习结果

这部分我学习到了以下内容：

- oneAPI 如何解决异构世界中的编程挑战
- 利用 oneAPI 解决方案来支持你的工作流程
- 使用 Intel® DevCloud 试用 oneAPI 工具和库
- SYCL 语言和编程模型的基础知识
- 通过在上下文中编辑源代码来熟悉 Jupyter 笔记本的使用

以下是代码运行截图：



图 1.3: 代码执行

1.2 SYCL_Program_Structure

1.2.1 学习内容

本章当中学习使用 SYCL 进行编程的方法. 首先, GPU 编程就需要对应的 GPU 设备, 因此 OneApi 提供了一些能够获取 GPU 基本信息的相关函数. 之后, 在第一节课中提到的命令组概念就是以队列方式实现的, 一个 SYCL 程序首先就需要创建队列 q, 队列是一种将工作提交给设备的机制, 一个队列或多个队列都可以映射到同一个设备, 创建过程也可以指定要使用的 GPU、FPGA、CPU。对应的, 内核类封装了在实例化命令组时在设备上执行代码的方法和数据。内核对象不是由用户明确构造的, 而是在调用内核调度函数 (例如 `parallel_for`) 时构造的, 内核类要作为队列的 `submit` 函数参数传入。

一个 SYCL 应用程序可以使用以下两种内存模型之一编写: 统一共享内存模型 (USM) 缓冲区内内存模型统一共享内存模型是一种基于指针的内存模型方法, 类似于 C/C++ 基于指针的内存分配。这使得将 C/C++/CUDA 应用程序迁移到 SYCL 变得更容易。缓冲区内内存模型允许使用一种名为缓冲区的新内存抽象, 并使用访问器进行访问, 访问器允许设置读写权限和其他内存属性。允许数据以 1、2 或 3 维表示, 并使使用 2/3 维数据的内核编程更容易。在 USM 中多个内核之间的依赖关系是显式的, 而在后者当中是隐式的。随后作者给出了两向量对应加和的代码示例, 分析在实验结果中给出。同时, 这里需要复制内存的原因是设备和主机可以共享物理内存, 也可以拥有不同的内存。当内存不同时, 卸载计算需要在主机和设备之间复制数据。对应的通过创建缓冲区和访问器的方式, SYCL 可以让主机和设备无需过多操作即可获得数据。

教程首先介绍了设备选择器类。这些类允许在运行时根据用户提供的启发式方法选择特定设备来执行内核。

演示代码

```

1 #include <sycl/sycl.hpp>
2 using namespace sycl;
3 int main() {
4     queue q(gpu_selector_v);
5     //queue q(cpu_selector_v);
6     //queue q(accelerator_selector_v);
7     //queue q(default_selector_v);
8     //queue q;
9     std::cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";
10    return 0;
11 }

```

然后课程介绍了并行内核。并行内核允许在加速器上并行执行操作, 利用 ‘`parallel_for`’ 函数可以

将独立的 for 循环并行化。基本并行内核通过 ‘range’ 和 ‘id’ 类描述和索引迭代空间，而 ‘item’ 类提供额外的迭代信息。ND-Range 内核通过 ‘nd_range’ 和 ‘nd_item’ 类进一步优化性能，通过分组执行和对硬件资源的精细控制来提高效率。

演示代码

```

1 | q.parallel_for(range<1>(1024), [=](item<1> item){
2 |     auto i = item.get_id();
3 |     auto R = item.get_range();
4 | });

```

1.2.2 学习结果

本节我学到的内容有：

- 如何选择设备以卸载内核工作负载
- 如何使用缓冲区、访问器、命令组处理程序和内核编写 SYCL 程序
- 如何使用主机访问器和缓冲区销毁进行同步

首先对教程的示例代码都进行了学习和验证。在向量加和对比当中，可以看到两种内存模式的区别在于：USM 使用 malloc 方法，而缓冲区使用 buffer 方法申请内存；USM 的代码也更具有 C++ 风格，依次内存数据迁移到 GPU 设备（等待）、创建并行内核执行任务、数据回迁、释放内存这几个步骤，而缓冲区代码将这几步省略，只留下了 submit 步，在设备的缓冲数据上创建缓冲处理器，再传入并行内核。也可以看到共同点在于都需要申请内存、创建并行内核这两步骤，最后两种方法的输出都是一致的，即 $1+2=3$ 。

[illegible]

图 1.4: 不同内存模式下向量加和执行结果

然后教程展示了如何使用 SYCL 中的 Host Accessor 来实现数据同步。代码首先初始化了一个包含 16 个元素的向量，每个元素的值为 10。然后创建了一个 SYCL 队列和一个与向量关联的 buffer 对象。在提交的命令组中，代码通过 accessor 对象访问 buffer，并在并行 for 循环中将每个元素的值减去 2。接着，代码创建了一个 host accessor 对象来同步数据回主机。Host accessor 的创建是一个阻塞调用，只有在所有修改同一 buffer 的 SYCL 内核执行完毕后，数据才会通过 host accessor 可用。最后，代码通过 host accessor 读取并打印 buffer 中的每个元素，展示了数据同步后的结果。

```
1 1.6.3.3. Build and Run
Select the cell below and click run  to compile and execute the code:

1 | chmod 755 ./; chmod 755 ./run_test_accessors.sh; if [ -x "$CODEDIR -> qsub" ]; then ./q_run_test_accessors.sh; else ./run_test_accessors.sh; fi

2 | m uffd9f780a697924c316a078a1 is compiling SYCL_Essentials Module: - SYCL Program Structure sample - 3 of 7 test_accessors_sample.cpp
3 | ./run_test_accessors_sample.cpp:120:44: error: use of undeclared identifier 'b'
4 |     for (int i = 0; i < N; ++i) std::cout << B[i] << " ";
5 |                                     ^
6 |
```

图 1.5: Host Accessor

然后教程演示了如何通过缓冲区销毁（Buffer Destruction）来实现数据同步。缓冲区的创建发生在一个单独的函数作用域内。当执行超出这个函数作用域时，缓冲区的析构函数被调用，释放数据的所

有权并将数据复制回主机内存。随后作者总结以上方法，进行了复数矩阵相乘程序的并行化实验，代码细节都已经在先前提过，为了验证并行的争取性，作者对比了直接相乘与并行相乘的结果，最终结果说明并行成功。



图 1.6: GPU 选择

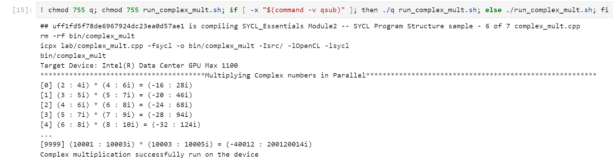


图 1.7: 矩阵结果对比

以下是我补充的练习代码和执行结果：

练习代码 2

```
1 int main() {
2     const int N = 256;
3     // # Initialize a vector and print values
4     std::vector<int> vector1(N, 10);
5     std::cout<<"\nInput Vector1: ";
6     for (int i = 0; i < N; i++) std::cout << vector1[i] << " ";
7     // # STEP 1 : Create second vector, initialize to 20 and print values
8     std::vector<int> vector2(N, 20);
9     std::cout<<"\nInput Vector2: ";
10    for (int i = 0; i < N; i++) std::cout << vector2[i] << " ";
11    // # Create Buffer
12    buffer vector1_buffer(vector1);
13    // # STEP 2 : Create buffer for second vector
14    buffer vector2_buffer(vector2);
15    // # Submit task to add vector
16    queue q;
17    q.submit([&](handler &h) {
18        // # Create accessor for vector1_buffer
19        accessor vector1_accessor (vector1_buffer, h);
20
21        // # STEP 3 - add second accessor for second buffer
22        accessor vector2_accessor (vector2_buffer, h, read_only);
23
24        h.parallel_for(range<1>(N), [=](id<1> index) {
25            // # STEP 4 : Modify the code below to add the second vector to first one
26            vector1_accessor[index] += vector2_accessor[index];
27        });
28    });
```

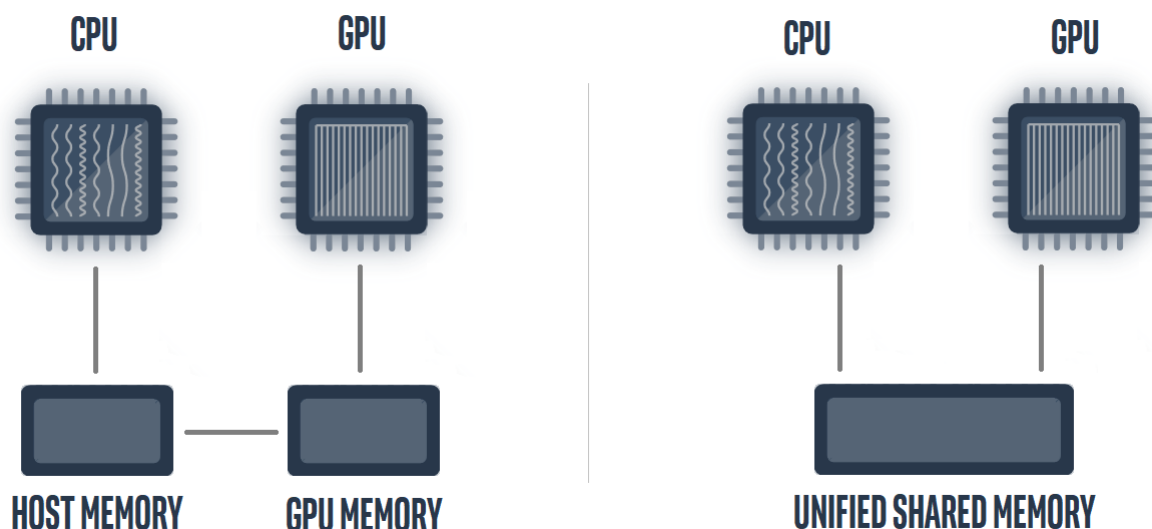




图 1.9: 开发者视角下是否使用 USM 的内存区别

1.3.2 学习结果

首先教程展示了使用 `malloc_shared` 实现 USM（统一共享内存），其中数据在主机和设备之间的移动是隐式进行的。这种方法有助于以最少的代码量快速实现功能，开发者无需担心在主机和设备之间移动内存。另一种情况主机和设备之间的数据移动需要开发者使用 `memcpy` 显式完成。这使得开发者可以更精确地控制主机和设备之间的数据移动。

1.5.1.1. Build and Run

Select the cell below and click run  to compile and execute the code:

```
! chmod 755 q; chmod 755 run_usm.sh; if [ -x "$(command -v qsub)" ]; then ./q run_usm.sh; else ./run_usm.sh; fi

## u304842ab3f4ceb1a2d679cdf88417ca is compiling SYCL_Essentials Module3 -- SYCL Unified Shared Memory - 1 of 5 usm.cpp
Device : Intel(R) Data Center GPU Max 1100
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
```

图 1.10: 显式（隐式）移动

然后教程展示单项依赖，使用 USM，并将三个内核提交到设备。每个内核都修改同一个数据数组。由于这三个队列提交之间存在数据依赖关系，因此需要修正代码以获得期望的输出 20。有三种解决方案：使用 `in_order` 队列属性，使用 `wait()` 事件，或使用 `depends_on()` 方法。接着介绍多项依赖，对每个被依赖的内核都命名，之后使用列表的方式以参数形式传入依赖其他内核的核当中，得到正确结果 25。

然后教程解释了子组如何映射到 GPU 以及为什么使用子组。并介绍了子组类和类函数及其使用的一些算法。你可以使用子组类的成员函数获取子组的大小以及其他关于子组的信息。

接着我们能了解到子组最有用的几个性质：shuffle, reduce 和 Broad-cast。Shuffle 是最有用的特性之一，其能够在各个工作项之间直接通信，而无需显式的内存操作。Shuffle 操使我们能够从内核中移除工作组局部内存的使用，或者避免不必要的重复访问全局内存。而 reduce_over_group 函数对子组中的所有项进行归约操作。归约操作在并行计算中是一种常见的模式，用于将一组数据通过某种操作如求和、求最大值、求最小值等汇总成一个单一的结果。而 Broadcast 是一种并行计算中的操作，用于让一个 work-item 将其变量的值共享给同一 group 中的所有其他工作项。通过广播操作，可以有效地在工作项之间传递数据，而无需每个工作项单独进行数据传递。

以下是其中代码的执行结果:


[illegible]

图 2.15: shuffle

[illegible]

图 2.16: reduce

```
10.3.1. Build and Run
```

Select the cell below and click  to compile and execute the code:

```
[ chmod 755 ./run_sub_group_broadcast.sh; if [[ "$1" == "command" || -q$1" ]]; then ./run_sub_group_broadcast.sh; else ./run_sub_group_broadcast.sh; fi  
rm -rf $HOME/aiida26797cf8f8147c is compiling SVL_Essentials under SVL Sub groups - 5 of 7 sub_group_broadcast.sh  
Device: icelad26797cf8f8147c GPU Max: 136  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68  
70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125  
126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175  
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224  
225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255  
  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68  
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125  
126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175  
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224  
225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255  
  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68  
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125  
126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175  
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224  
225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255  
  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68  
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125  
126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175  
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224  
225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255  
  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
```

图 2.17: Broadcast

2.1.2 练习题答案与结果

练习题使用子组中的 `reduce` 方法实现从 1 到 1024 的求和，首先使用子组包含的方法获得当前工作项的子组对象以及对应的 `id`，然后使用 `reduce` 对子组的 32 个工作项进行归约操作，计算当前子组内所有工作项之和，然后将获得的归约结果放入 `sg_data` 数组当中，下标逻辑为当前工作组的 `ID`+ 每个工作组中的子组数量 + 当前子组在其工作组中的 `ID`，最后将 `sg_data` 求和输出。

练习代码 2

```
1  int main () {
2      queue q ;
```

```

3      cout << "Device : " << q . get_device () . get_info<info :: device :: name>()
        << "\n" ;
4      int *data = malloc_shared<int>(N, q) ;
5      int *sg_data = malloc_shared<int>(N/S , q) ;
6      for (int i = 0; i < N; i++) data [i] = i ;
7      for (int i = 0; i < N; i++) std :: cout << data [i] << " " ;
8      cout << "\n\n" ;
9      q . p a r a l l e l _ f o r ( nd_range<1>(N, B) , [=]( nd_item<1> item )
10     [[intel :: reqd_sub_group_size (S) ] ] {
11      auto sg = item get_sub_group () ;
12      auto i = item.get_global_id (0) ;
13      int result = reduce_over_group ( sg , data [i] , plus <>()) ;
14      if(sg.leader()) {
15      sg_data [item . get_group (0) * (B / S) + sg . get_group_id () [0]] = r e s u l t ;
16      }
17      } . wait () ;
18      for (int i = 0; i < N/S ; i++) std :: cout << sg_data [i] << " " ;
19      cout << "\n" ;
20      int sum = 0;
21      for (int i = 0; i < N/S ; i++) {
22      sum += sg_data [i] ;
23      }
24      cout << "\nSum = " << sum << "\n" ;
25      free (data ,q) ;
26      free (sg_data ,q) ;
27      return 0;

```

运行结果如下:

```

Select the cell below and click run to compile and execute the code.
1 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000
Device : Intel(R) Xeon Phi 9401Y Processor
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999
Sum : 524288

```

图 2.18: reduce 向量和

2.2 Intel_Advisor

2.2.1 Roofline Analysis

Roofline 模型是一种图表，用于直观地展示应用性能与硬件限制（如内存带宽和计算峰值）之间的关系。Intel Advisor 提供了一个自动生成屋顶线图的工具，能够自动测量和绘制图表，我们用户只需读取和分析图表即可。通过这个图表，我们可以识别性能瓶颈的位置，了解产生瓶颈的可能原因，并确定可以优化哪些性能瓶颈能带来最大的性能提升。

以下是实验平台为我的环境测出的一个 Roofline 图表：

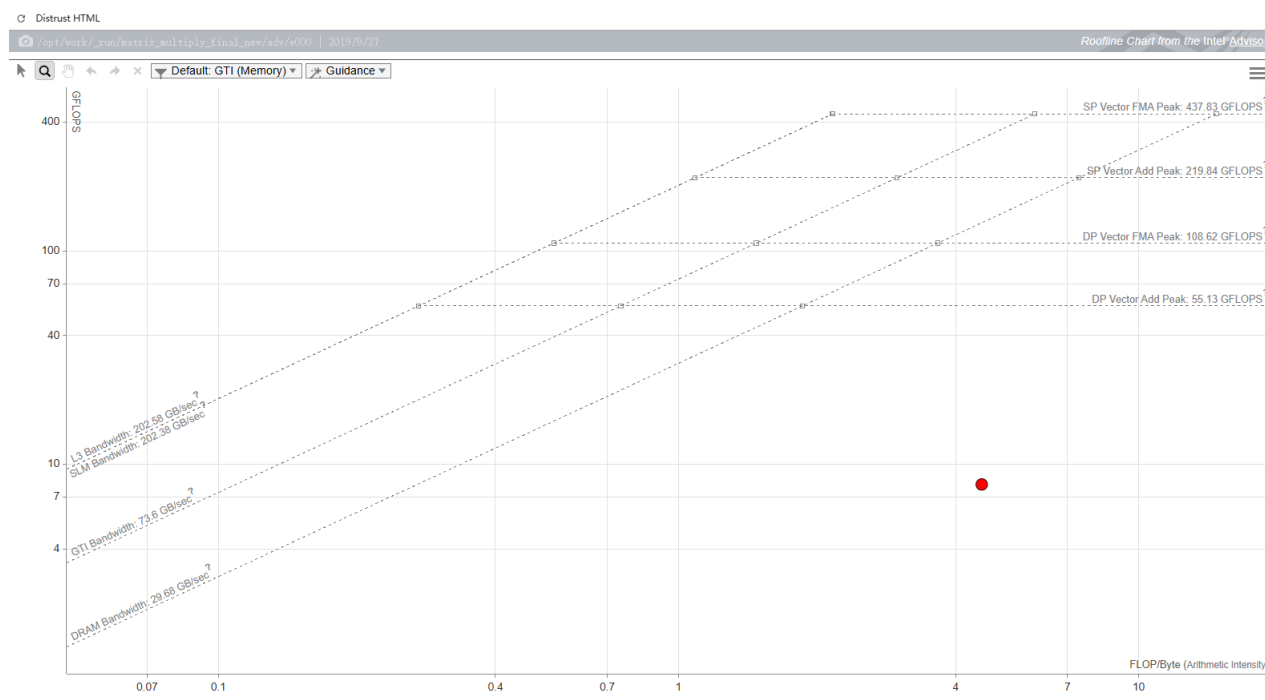


图 2.19: Roofline 图表

然后教程介绍了如何从这个图表上看起来性能较差的 loop，还有每个循环可以改进的“性能提升空间”和值得改进的地方。并且 roofline 报告还显示了可能导致性能瓶颈的出现的原因，比如内存受限还是计算受限，并提出了下一步的优化建议。

2.2.2 Offload Advisor

本章的学习内容有：

- 了解了什么是 Offload Advisor。运行了 Offload Advisor 报告。
- 分析了 Offload Advisor 的各种输出。
- 学习了 offloadAdvisor 命令行指令以及如何加快 collection 时间。

Offload Advisor 的主要功能是帮助开发者决定代码的哪些部分应该被加载到 GPU 上，以优化性能。Offload Advisor 可以：收集性能预测数据，除了常规的性能分析外，Offload Advisor 还可以收集预测数据，帮助评估卸载的潜在性能提升。同时生成详细报告，输出文件包含各种指标和性能数据，例如总加速比、被加速的代码比例、卸载的循环和函数数量等。并且分析调用树，提供一个调用树视图，展示哪些区域可以卸载和加速，从而帮助开发者更直观地理解代码的性能瓶颈和优化机会。

以下是 Offload Advisor 对我的实验环境生成的一份 html 报告。

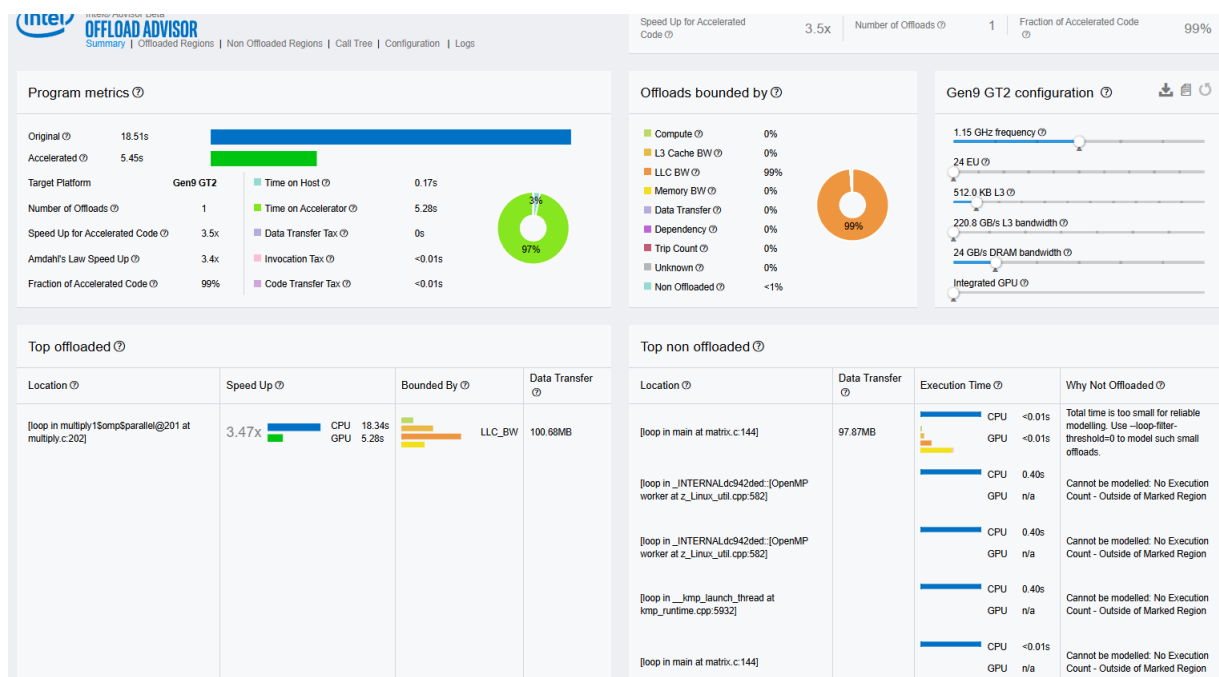


图 2.20: Offload Advisor 报告

并且 Offload Advisor 还可以帮助你提高计算性能。使用 Intel® Advisor 提升性能帮助优化向量化和内存使用 (适用于 CPU 和 GPU), 识别适合卸载的循环并预测在目标加速器上的性能。并且 Offload Advisor 可以帮助确定哪些内核应该被卸载, 并预测可能的加速效果。开发者可以使用 Intel® DPC++ Compatibility 工具进行一次性迁移, 从 CUDA 转换到 Data Parallel C++ (DPC++)。现有的 Fortran 应用程序可以使用基于 OpenMP 的指令风格, 而现有的 C++ 应用程序可以选择内核风格或基于指令的风格。

编写 SYCL 代码后, GPU Roofline 分析有助于制定优化策略, 并识别相对于目标最大性能的潜在瓶颈。最终, 通过 VTune 进行 GPU 分析, 可以进一步针对目标硬件进行优化。

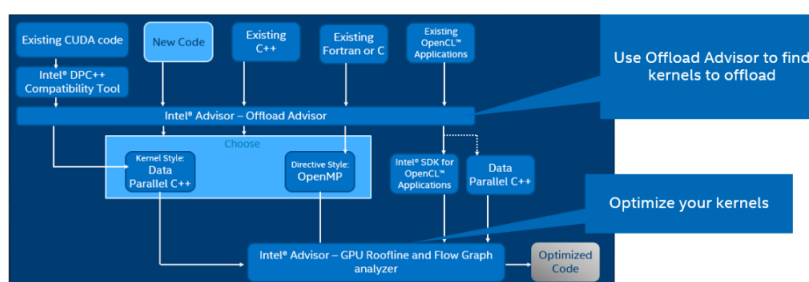


图 2.21: Offload Advisor 优化流程

然后教程介绍了如何找到可以有效加载的代码, 并给出展示图, 显示工作负载加速, 工作负载运行时间大幅减少。然后告诉我们这样加载是否会提高性能? 并且告诉我们哪些是适合加载的候选者, 哪些是不适合加载的候选者, 还可有工作负载的限制因素。然后教程提供每个适合加载的 loop 循环的详细描述。从其中可以查看时间 (总时间、加速器上的时间、加速比)、加载指标 (如加载开销和数据传输)、内存流量和循环次数。以及哪些内核不应该被加载?