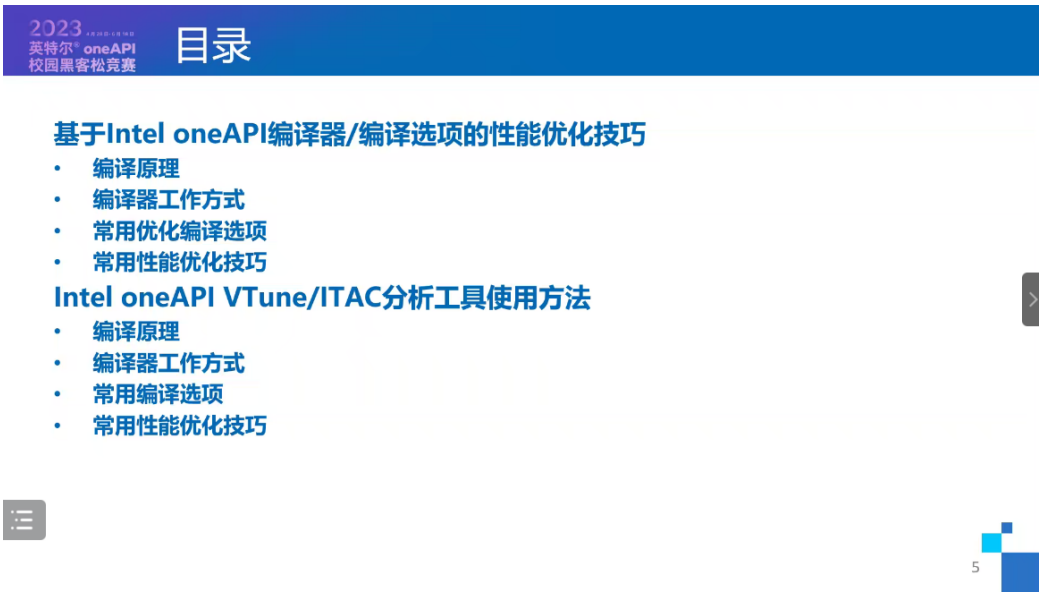
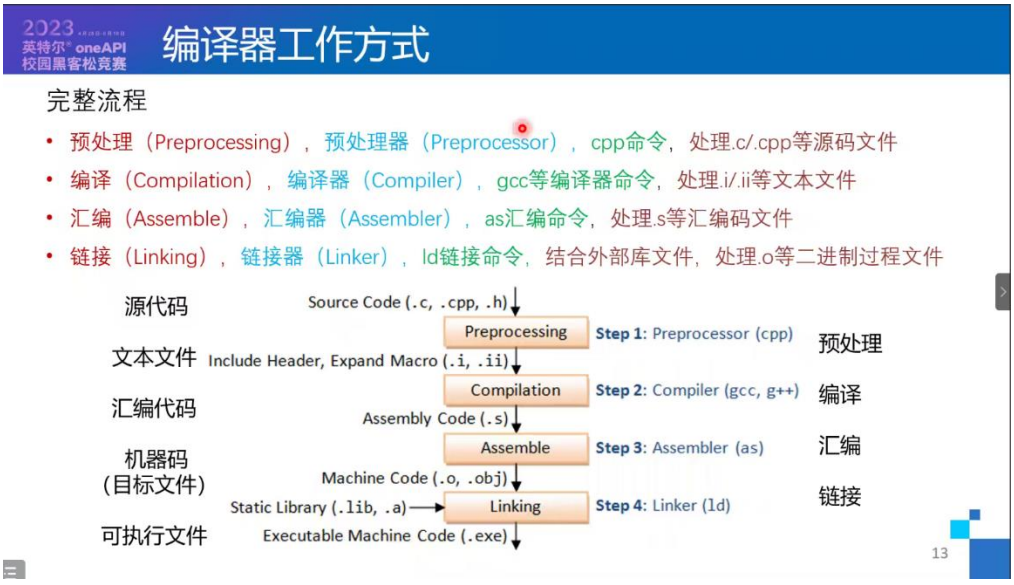
GPU 编程作业

学号：2120210492　　姓名：肖飞

一、OneAPI tools 培训内容

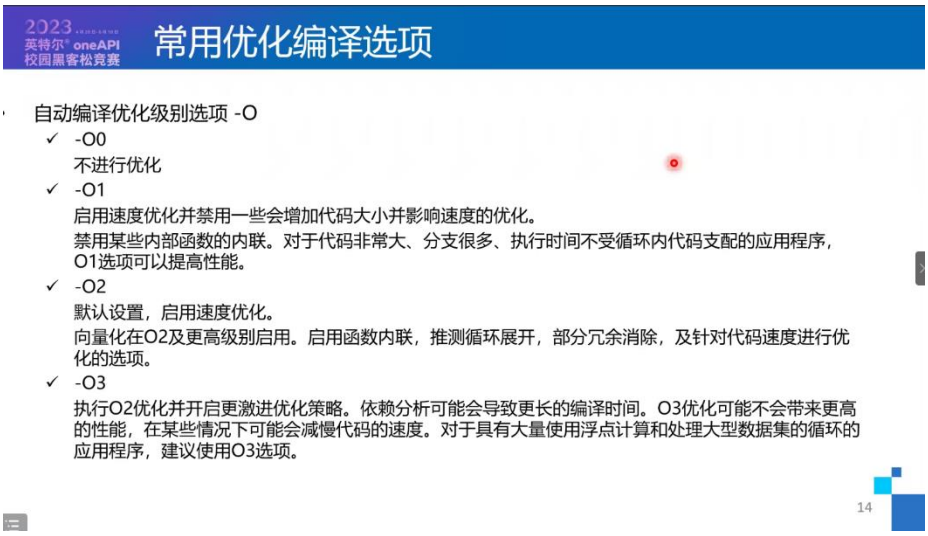


整体内容主要分为两部分，第一部分是基于 Intel oneAPI 编译器/编译选项的性能优化技巧，第二部分是如何通过分析工具对程序进行性能分析，从而找到可以进行优化的地方。

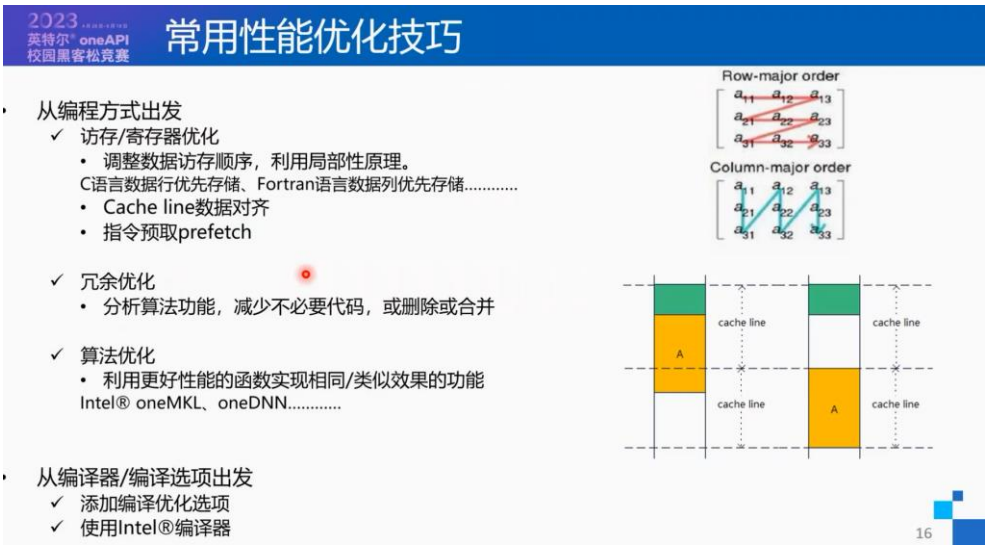


首先简单介绍了编译的过程。

Intel 的工具套件。

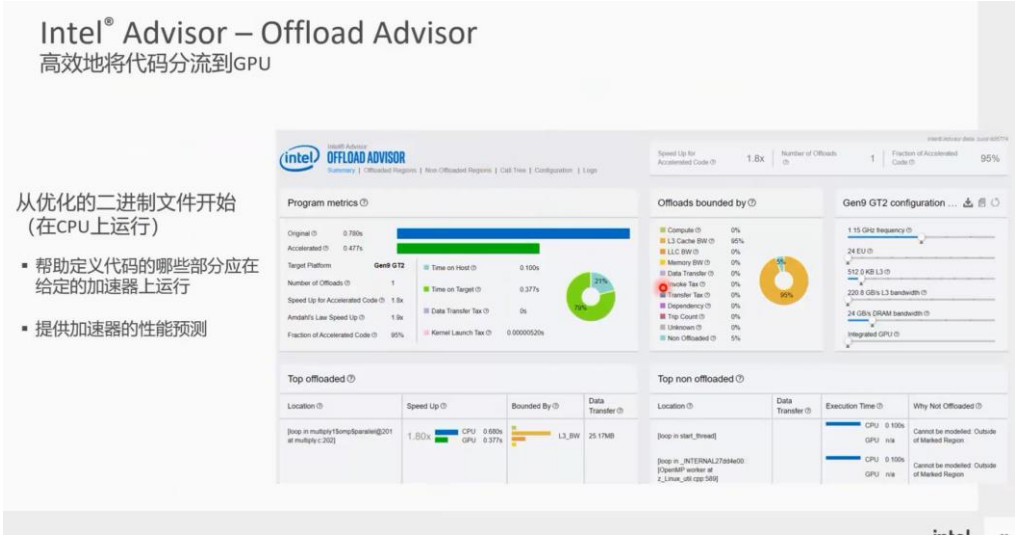


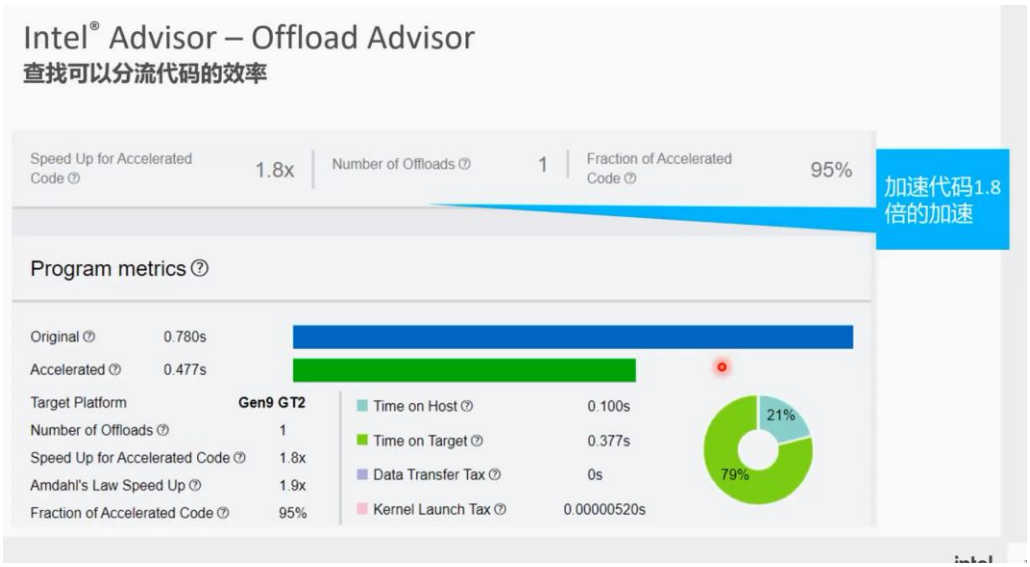
自动编译时进行一些优化选项的设置可能在不修改代码的情况下提高程序性能。



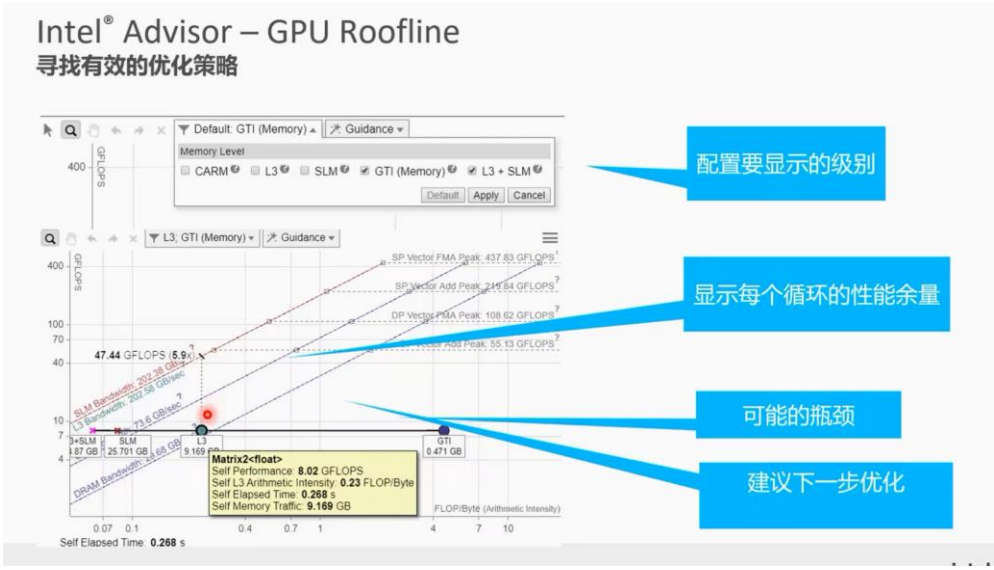
第二部分介绍的分析工具主要有以下几个:

Advisor 对于 python 的性能优化帮助可能不大，而 VTune 对于 python 代码也可以进行分析。

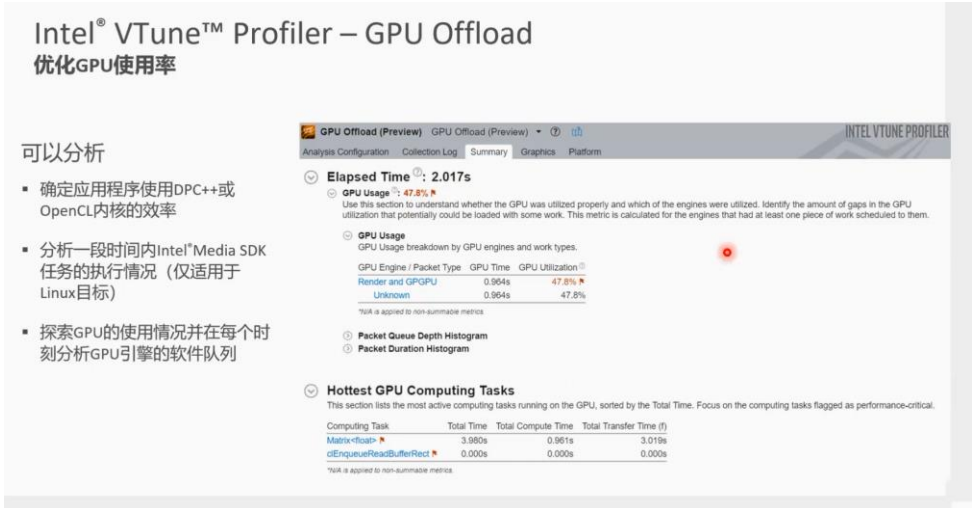


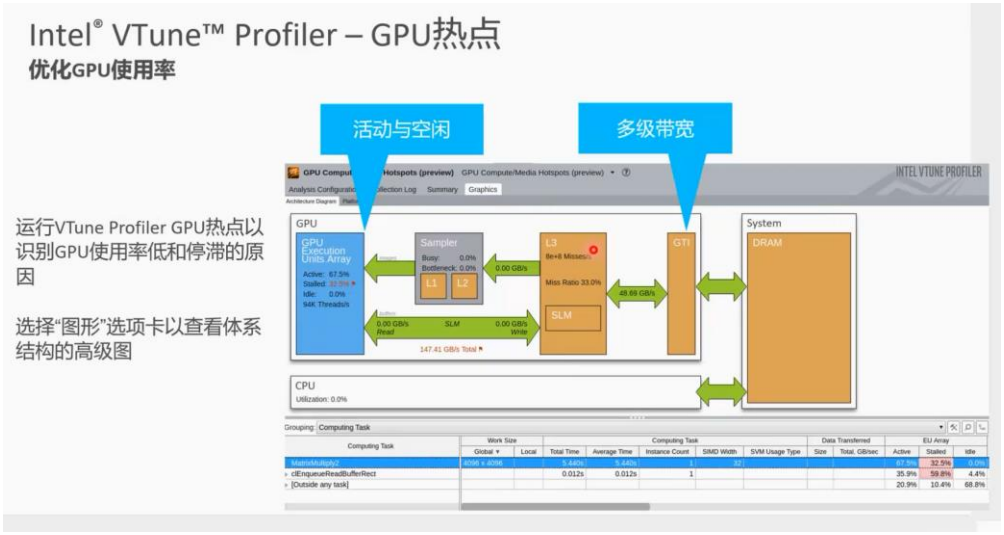


程序在 GPU 上的加速效果

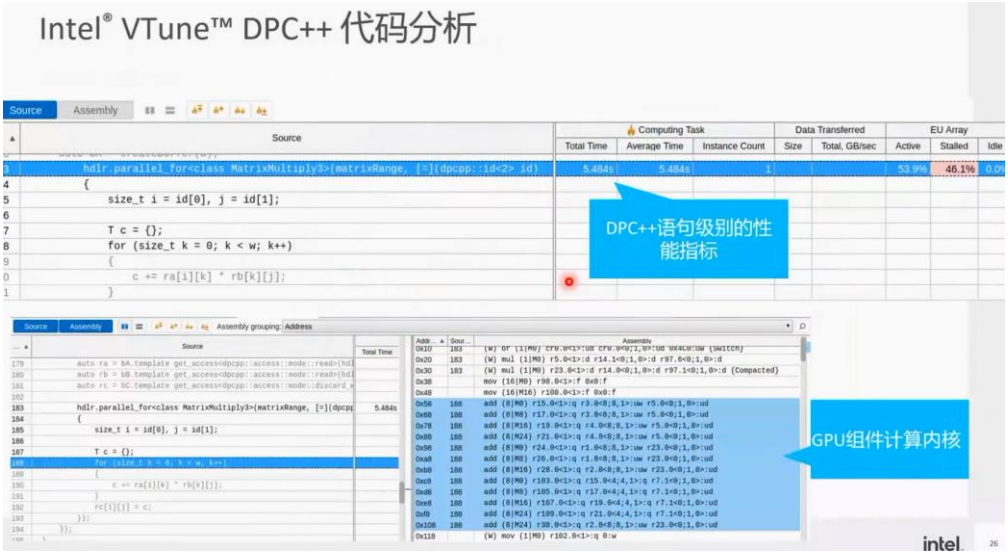


可以分析出程序中不同函数的时间等开销及可优化的空间。



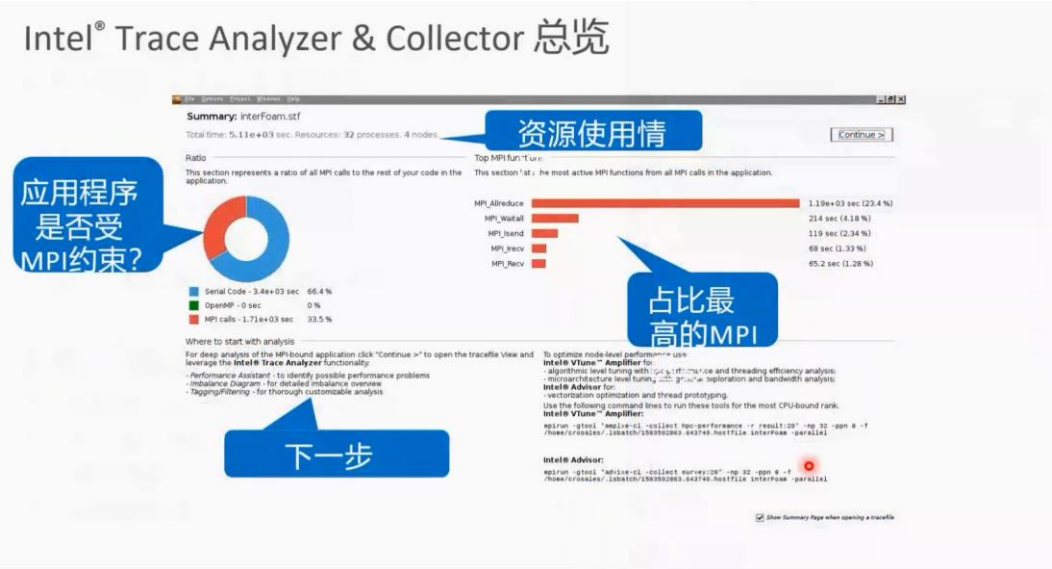


GPU 分析



具体代码语句的性能分析

MPI 程序行为分析



二、动手练习

实验环境如下：

实验平台：x64 平台

CPU：12th Gen Intel(R) Core(TM) i5-12500H　3.10 GHz

软件：Intel® oneAPI Base Toolkit & HPC Toolkit

操作系统：　Windows 11 系统

编译器：TDM-GCC & ICC 编译器

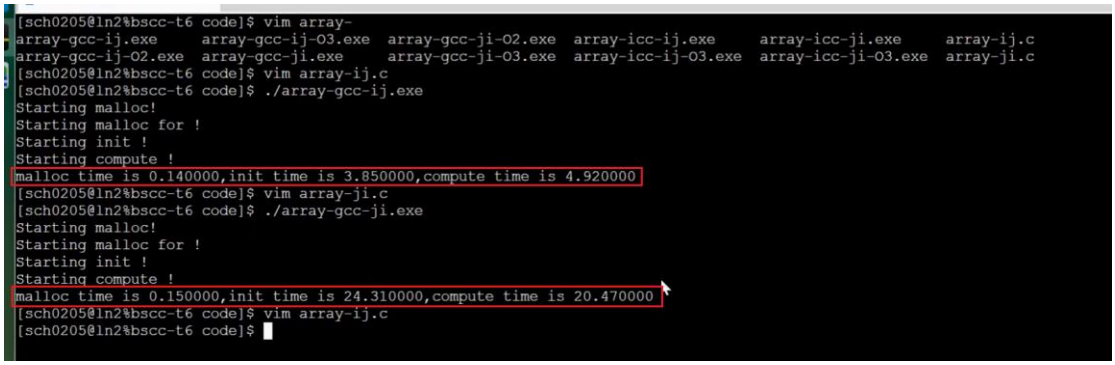
下面是一段矩阵加法的 C 代码：

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define N 20000
#define FIRST i
#define SECOND j

int main()
{
    int i,j,k;
    int **a,**b,**c;
    clock_t malloc_start,malloc_end,init_start,init_end,compute_start,compute_end;
    double malloc_time,init_time,compute_time;
    malloc_start=clock();
    printf("Starting malloc!\n");
    a=(int **)malloc(sizeof(int *) * N);
    b=(int **)malloc(sizeof(int *) * N);
    c=(int **)malloc(sizeof(int *) * N);
    printf("Starting malloc for !\n");
    for(i=0;i<N;i++)
    {
        a[i]=(int *)malloc(sizeof(int) * N);
        b[i]=(int *)malloc(sizeof(int) * N);
        c[i]=(int *)malloc(sizeof(int) * N);
    }
    malloc_end=clock();
    init_start=clock();
    printf("Starting init ! \n");
    for(FIRST=0;FIRST<N;FIRST++)
    {
"array-ij.c" 64L, 1645C
```

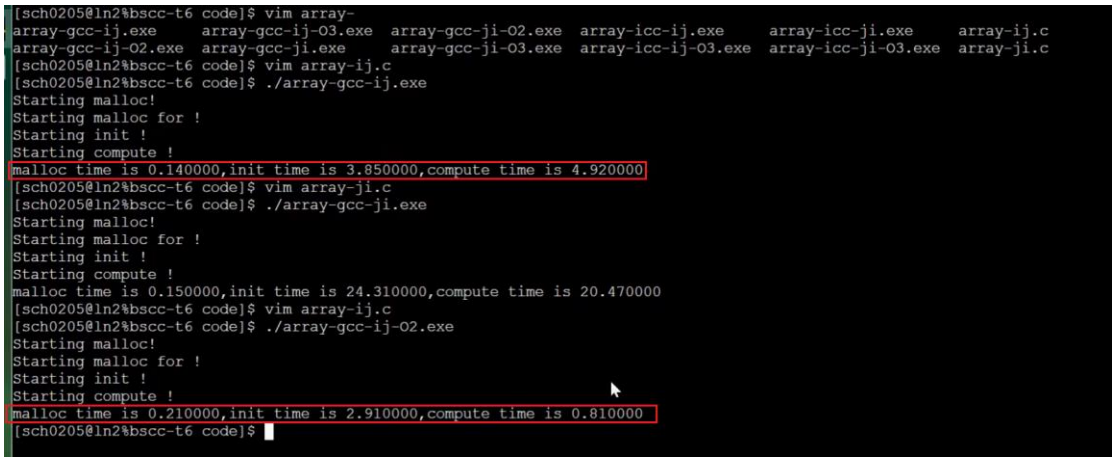
```c
    malloc_end=clock();
    init_start=clock();
    printf("Starting init ! \n");
    for(FIRST=0;FIRST<N;FIRST++)
    {
        for(SECOND=0;SECOND<N;SECOND++)
        {
            a[i][j]=i;
            b[i][j]=j;
            c[i][j]=0;
        }
    }
    init_end=clock();
    printf("Starting compute ! \n");
    compute_start=clock();
    for(FIRST=0;FIRST<N;FIRST++)
    {
        for(SECOND=0;SECOND<N;SECOND++)
        {
            c[i][j]=a[i][j] + b[i][j];
        }
    }
    compute_end=clock();
    malloc_time=(double)(malloc_end-malloc_start) / CLOCKS_PER_SEC;
    init_time=(double)(init_end-init_start) / CLOCKS_PER_SEC;
```

```
compute_end=clock();
malloc_time=(double)(malloc_end-malloc_start) / CLOCKS_PER_SEC;
init_time=(double)(init_end-init_start) / CLOCKS_PER_SEC;
compute_time=(double)(compute_end-compute_start) / CLOCKS_PER_SEC;
printf("malloc time is %lf,init time is %lf,compute time is %lf\n",malloc_time,init_time,compute_time);
for(i=0;i<N;i++)
{
    free(a[i]);
    free(b[i]);
    free(c[i]);
}
free(a);
free(b);
free(c);
return 0;
```

按照不同策略对上述代码进行优化，并编译运行测试不同阶段的时间开销，得到以下结果。



对于 C 程序，遵循行优先访存（第一次）的性能相比按照列优先访存（第二次）的性能由有较大的提升。



使用了-O2 编译优化选项后，各阶段代码性能也有了一定提升。

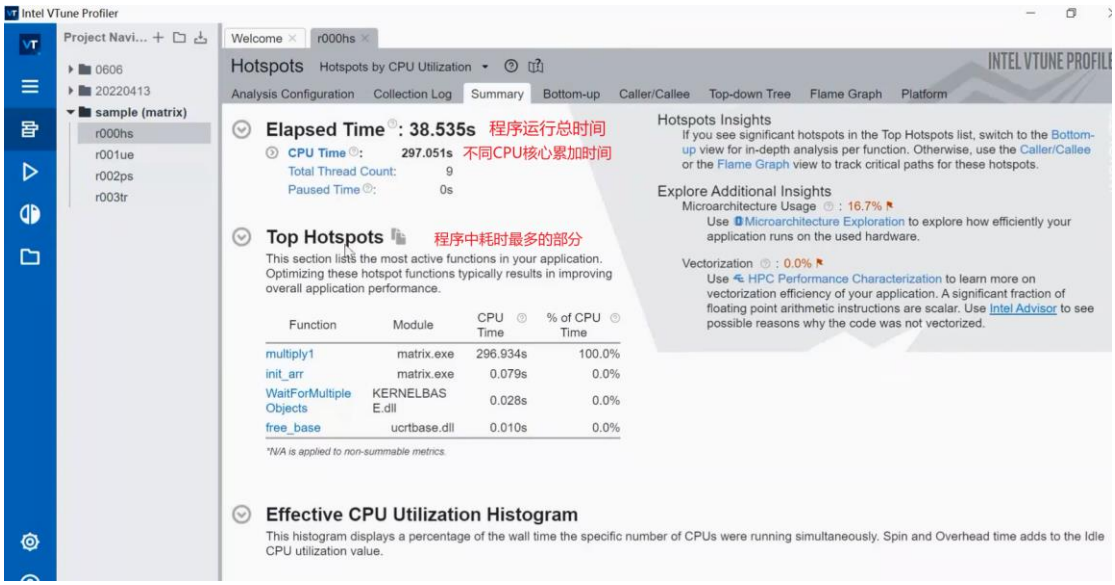
```
[sch0205@ln2%bscc-t6 code]$ vim array-ji.c
[sch0205@ln2%bscc-t6 code]$ ./array-gcc-ji.exe
Starting malloc!
Starting malloc for !
Starting init !
Starting compute !
malloc time is 0.150000,init time is 24.310000,compute time is 20.470000
[sch0205@ln2%bscc-t6 code]$ vim array-ij.c
[sch0205@ln2%bscc-t6 code]$ ./array-gcc-ij-O2.exe
Starting malloc!
Starting malloc for !
Starting init !
Starting compute !
malloc time is 0.210000,init time is 2.910000,compute time is 0.810000
[sch0205@ln2%bscc-t6 code]$ ./array-gcc-ji-O2.exe
Starting malloc!
Starting malloc for !
Starting init !
Starting compute !
malloc time is 0.160000,init time is 32.990000,compute time is 20.920000
[sch0205@ln2%bscc-t6 code]$
```

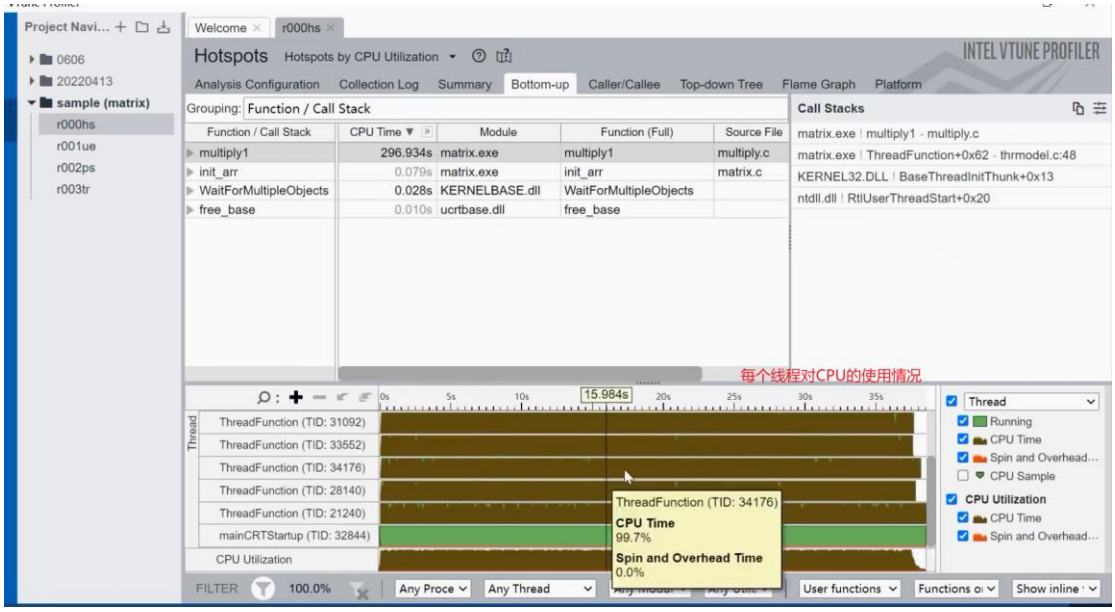
对于列优先访存的代码，使用-O2 选项编译反倒增加了时间开销。

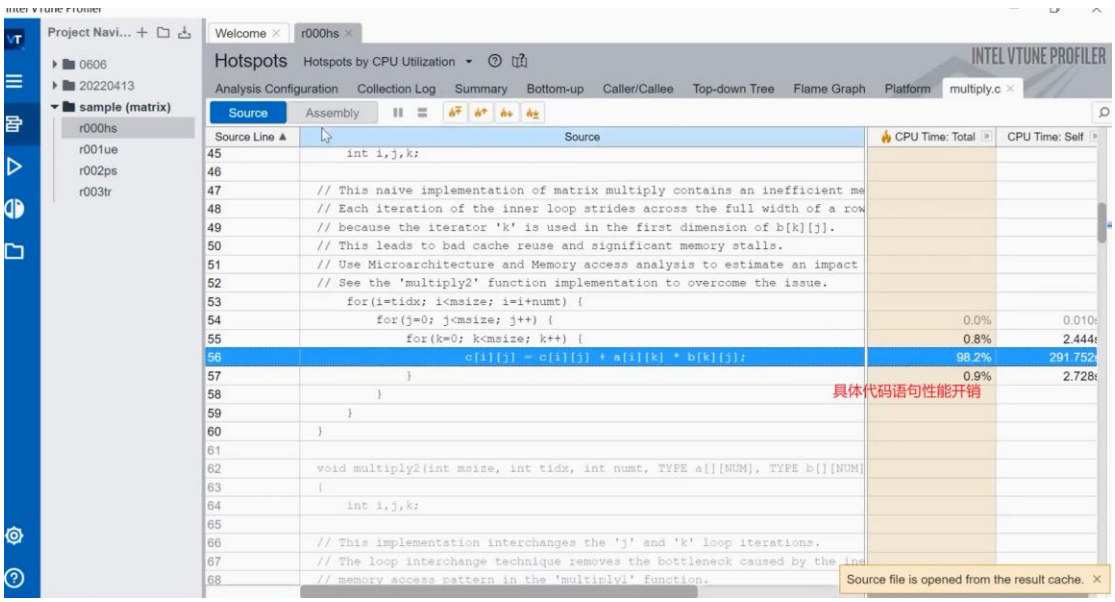
```
[sch0205@ln2%bscc-t6 code]$ ls
array-gcc-ij.exe     array-gcc-ji.exe     array-icc-ij.exe     array-icc-ji-O3.exe  compile.sh  run.sh
array-gcc-ij-O2.exe  array-gcc-ji-O2.exe  array-icc-ij-O3.exe  array-ij.c           matrix-c
array-gcc-ij-O3.exe  array-gcc-ji-O3.exe  array-icc-ji.exe     array-ji.c           matrix.c
[sch0205@ln2%bscc-t6 code]$ ./array-icc-ij.exe
Starting malloc!
Starting malloc for !
Starting init !
Starting compute !
malloc time is 0.150000,init time is 2.670000,compute time is 0.950000
[sch0205@ln2%bscc-t6 code]$ ./array-icc-ji.exe
Starting malloc!
Starting malloc for !
Starting init !
Starting compute !
malloc time is 0.150000,init time is 3.380000,compute time is 1.820000
[sch0205@ln2%bscc-t6 code]$ ./array-icc-ij-O3.exe
Starting malloc!
Starting malloc for !
Starting init !
Starting compute !
malloc time is 0.160000,init time is 2.830000,compute time is 0.930000
[sch0205@ln2%bscc-t6 code]$ ./array-icc-ji-O3.exe
Starting malloc!
Starting malloc for !
Starting init !
Starting compute !
malloc time is 0.150000,init time is 3.030000,compute time is 0.840000
[sch0205@ln2%bscc-t6 code]$
```

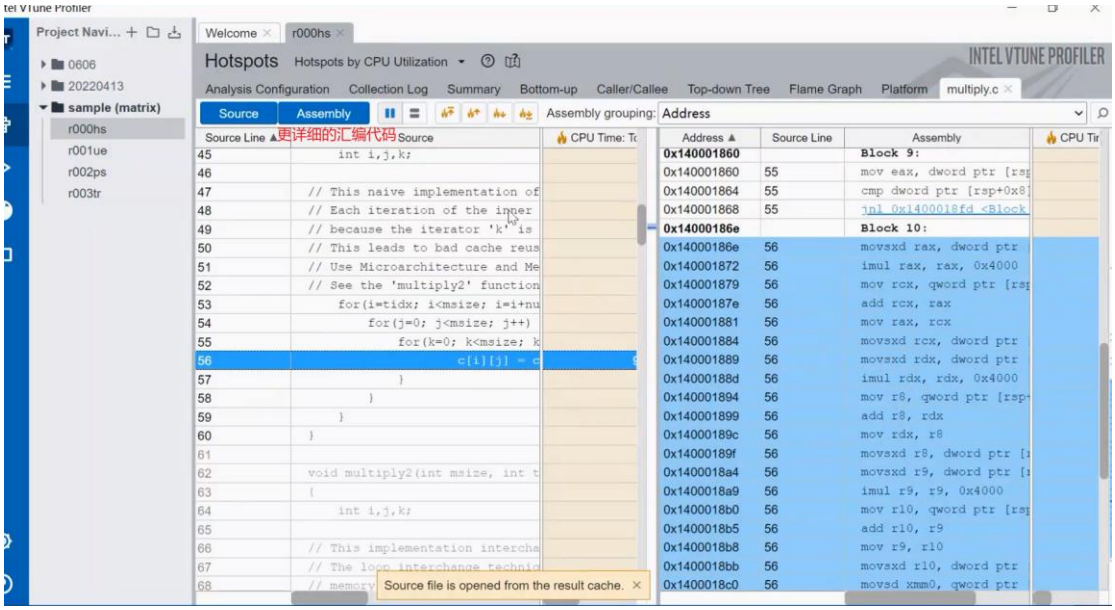
将 gcc 编译器更换为 Intel 的 icc 编译器后，同样的代码性能也有了一定提升，即便将行优先访存更改为列优先访存，性能也只是下降了一点，也就是说，使用 Intel 编译器可以在考虑较少代码优化细节（如访存顺序、编译优化选项等）的情况下仍旧保持不错的性能。

使用 VTune 工具（windows 版本）进行代码性能分析：


Intel VTune Profiler
Project Navi...
0606
20220413
sample (matrix)
r000hs
r001ue
r002ps
r003tr
Welcome
r000hs
Hotspots
Hotspots by CPU Utilization
INTEL VTUNE PROFILE
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
Elapsed Time: 38.535s
程序运行总时间
CPU Time: 297.051s
不同CPU核心累加时间
Total Thread Count: 9
Paused Time: 0s
Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.
Explore Additional Insights
Microarchitecture Usage : 16.7%
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.
Vectorization : 0.0%
Use HPC Performance Characterization to learn more on vectorization efficiency of your application. A significant fraction of floating point arithmetic instructions are scalar. Use Intel Advisor to see possible reasons why the code was not vectorized.
Top Hotspots
程序中耗时最多的部分
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.
Function | Module | CPU Time | % of CPU Time
multiply1 | matrix.exe | 296.934s | 100.0%
init_arr | matrix.exe | 0.079s | 0.0%
WaitForMultipleObjects | KERNELBASE.dll | 0.028s | 0.0%
free_base | ucrtbase.dll | 0.010s | 0.0%
*N/A is applied to non-summable metrics.
Effective CPU Utilization Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.


Project Navi...
Welcome
r000hs
Hotspots
Hotspots by CPU Utilization
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
Grouping: Function / Call Stack
Function / Call Stack | CPU Time | Module | Function (Full) | Source File
multiply1 | 296.934s | matrix.exe | multiply1 | multiply.c
init_arr | 0.079s | matrix.exe | init_arr | matrix.c
WaitForMultipleObjects | 0.028s | KERNELBASE.dll | WaitForMultipleObjects
free_base | 0.010s | ucrtbase.dll | free_base
Call Stacks
matrix.exe ! multiply1 - multiply.c
matrix.exe ! ThreadFunction+0x62 - thrmodel.c:48
KERNEL32.DLL ! BaseThreadInitThunk+0x13
ntdll.dll ! RtlUserThreadStart+0x20
每个线程对CPU的使用情况
15.984s
ThreadFunction (TID: 31092)
ThreadFunction (TID: 33552)
ThreadFunction (TID: 34176)
ThreadFunction (TID: 28140)
ThreadFunction (TID: 21240)
mainCRTStartup (TID: 32844)
CPU Utilization
ThreadFunction (TID: 34176)
CPU Time
99.7%
Spin and Overhead Time
0.0%
Thread
Running
CPU Time
Spin and Overhead...
CPU Sample
CPU Utilization
FILTER
100.0%
Any Proce
Any Thread
User functions
Show inline


Project Navi...
Welcome
r000hs
Hotspots
Hotspots by CPU Utilization
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
multiply.c
Source
Assembly
Source Line
Source
CPU Time: Total
CPU Time: Self
45 int i,j,k;
46
47 // This naive implementation of matrix multiply contains an inefficient me
48 // Each iteration of the inner loop strides across the full width of a row
49 // because the iterator 'k' is used in the first dimension of b[k][j].
50 // This leads to bad cache reuse and significant memory stalls.
51 // Use Microarchitecture and Memory access analysis to estimate an impact
52 // See the 'multiply2' function implementation to overcome the issue.
53 for(i=tidx; i<msize; i=i+numt) {
54 for(j=0; j<msize; j++) { 0.0% 0.010s
55 for(k=0; k<msize; k++) { 0.8% 2.444s
56 c[i][j] = c[i][j] + a[i][k] * b[k][j]; 98.2% 291.752s
57 } 0.9% 2.728s
58 }
59 }
60 }
61
62 void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM]
63 {
64 int i,j,k;
65
66 // This implementation interchanges the 'j' and 'k' loop iterations.
67 // The loop interchange technique removes the bottleneck caused by the ine
68 // memory access pattern in the 'multiply1' function.
具体代码语句性能开销
Source file is opened from the result cache.



三、项目地址

Git 项目链接：https://github.com/XFLasdf/Guass_GPU