



南開大學  
Nankai University

计算机学院  
并行程序设计实验报告

GPU 编程

姓名：黄天昊王旭尧  
学号：2011763 2012527  
专业：计算机科学与技术

2023 年 6 月 27 日

# 目录

<b>1 黑客松习题</b>	<b>2</b>
1.1 学习截图 . . . . .	2
1.2 实验环境 . . . . .	2
1.3 实验结果 . . . . .	2
1.3.1 实验三 . . . . .	2
1.3.2 实验四 . . . . .	2
1.4 结果分析 . . . . .	2
1.4.1 实验三 . . . . .	2
1.4.2 实验四 . . . . .	3
<b>2 Accelerating Applications with CUDA C/C++</b>	<b>3</b>
2.1 写 CUDA 程序 . . . . .	3
2.1.1 知识点 . . . . .	3
2.1.2 例题 . . . . .	4
2.2 Debug . . . . .	4
2.2.1 知识点 . . . . .	4
2.2.2 例题 . . . . .	5
2.3 总结 . . . . .	5
2.4 进阶: Grids and Blocks of 2 and 3 Dimensions . . . . .	6
2.4.1 知识点 . . . . .	6
2.4.2 例题 . . . . .	6
<b>3 Managing Accelerated Application Memory with CUDA Unified Memory and nsys</b>	<b>7</b>
3.1 Profile an Application with nsys . . . . .	7
3.2 Streaming Multiprocessors . . . . .	7
3.3 Query the Device . . . . .	8
3.4 Unified Memory . . . . .	9
3.5 Asynchronous Memory Prefetching . . . . .	9
3.6 Final Exercise . . . . .	10
<b>4 Asynchronous Streaming, and Visual Profiling for Accelerated Applications with CUDA C/C++</b>	<b>12</b>
4.1 Nsight Systems Profiling . . . . .	12
4.2 Stream . . . . .	13
4.2.1 知识点 . . . . .	13
4.2.2 例题 . . . . .	14
4.3 Final Exercise . . . . .	15
<b>5 高斯消去</b>	<b>15</b>

## 1 黑客松习题

### 1.1 学习截图

818 656 844  
**CUDA编程实验课**  
时间: 18:38 发起人: 孙辉

图 1.1: 学习截图 1

175 767 220 研讨  
**OneAPI tools 培训**  
时间: 18:56 发起人: IvyZ

图 1.2: 学习截图 2

### 1.2 实验环境

CPU:intel core i7  
GPU:NVIDIA GeForce RTX  
CUDA:11.7.1.1

### 1.3 实验结果

#### 1.3.1 实验三

scale	CPU	GPU
1*1	50.6246	38.4013
2*2	49.6235	22.4372
4*4	49.8923	4.9323
8*4	49.9234	20.5243

表 1: 不同 tile 结果

#### 1.3.2 实验四

register size	CPU	GPU
1	18.8543	6.5823
2	18.8602	3.8235
4	18.8609	3.3325
8	19.0934	3.3146
16	18.8732	3.2264

表 2: 不同 tile 结果

### 1.4 结果分析

#### 1.4.1 实验三

tile 大小对于 CPU 执行时间几乎没有影响: 经过多次测试, 得出的结果显示, CPU 的计算时间对于不同规模的问题影响较为一致。

tile 大小对于 GPU 执行时间有巨大影响：当问题的尺寸逐渐增加时，计算时间会先逐渐减少，然后再逐渐增加。这说明在一定范围内增加 tile 大小可以减少执行时间，但一旦超过一定的界限，性能就会变差。下面是对上述现象可能原因的推测：

- 计算资源：随着问题规模增加，硬件的计算资源逐步被充分利用，从而提高了性能。然而，一旦超过硬件资源的承载能力，资源竞争会导致计算时间增加。
- 局部性：问题规模的大小影响了数据的局部性利用效率。在问题规模初步增长时，由于局部性良好，数据可以有效获取，从而提高了执行效率。然而，当问题规模过大时，局部性变差，导致效率下降。

#### 1.4.2 实验四

- 提高内存访问效率：通过增加寄存器的大小，可以将更多的行和列数据一次性读取到寄存器中，减少对全局内存的访问次数。这样做可以显著降低内存访问的延迟，并提高数据的读取速度和处理效率。
- 减少数据依赖性：寄存器的扩大使得每个线程能够一次性读取多个元素的数据到寄存器中，并行地执行乘法和累加操作。由于每个线程都可以独立地从寄存器中获取所需的数据，减少了对其他线程计算结果的依赖。这种减少数据依赖性的特性可以提高并行度，并允许更多的计算任务同时进行，从而加快计算速度。

## 2 Accelerating Applications with CUDA C/C++

本课程是 CUDA 编程的入门基础内容，每个小知识点后都有一个 exercise，加上进阶内容共 11 个 exercise。下面是我对知识点和较难 exercise 的总结。

### 2.1 写 CUDA 程序

#### 2.1.1 知识点

1. 声明 kernel 函数：使用 `__global__` 关键字，且返回类型需为 `void`。
2. 调用 kernel 函数：使用  
`kernelFunc <<< NUMBER_OF_BLOCKS, NUMBER_OF_THREADS_PER_BLOCK >>>`  
`()`。
3. 若想成功执行，需要在调用后使用 `cudaDeviceSynchronize()` 来同步 CPU 和 GPU，否则 CPU 端执行后直接退出。
4. `gridDim.x`: grid 中的 block 数；`blockIdx.x`: grid 中某 block 的索引；`blockDim.x`: block 中的 thread 数；`threadIdx.x`: block 中某 thread 的索引。指定特定 thread 时可用 `if(threadIdx.x == 1023 && blockIdx.x == 255)` 类似语句。
5. 使用语句 `cudaMallocManaged(&a, size)` 初始化一个指向 `size` 大小空间的指针 `a`；使用 `cudaFree(a)` 释放 `a` 指向的内存地址。

6. 使用 `threadIdx.x+blockIdx.x*blockDim.x` 计算 thread 的索引, 使用 `gridDim.x*blockDim.x` 计算循环步长 (当线程数 < 待处理元素数时)。例如以下代码:

---

```

1  __global__ void kernel(int *a, int N)
2  {
3      int indexWithinTheGrid = threadIdx.x + blockIdx.x * blockDim.x;
4      int gridStride = gridDim.x * blockDim.x;
5
6      for (int i = indexWithinTheGrid; i < N; i += gridStride)
7      {
8          // do work on a[i];
9      }
10 }
```

---

7. 一个 block 内最多只能有 1024 个 thread。

### 2.1.2 例题

**控制顺序:** 使用 `cudaDeviceSynchronize()` 语句控制 CPU 和 GPU 的调用顺序, 达到 GPU 输出->CPU 输出->GPU 输出的结果。

---

```

1  int main()
2  {
3
4      helloGPU<<<1, 1>>>();
5      cudaDeviceSynchronize();
6
7      helloCPU();
8
9      helloGPU<<<1, 1>>>();
10     cudaDeviceSynchronize();
11 }
```

---

## 2.2 Debug

### 2.2.1 知识点

使用 `cudaGetLastError` 函数获取 synchronous error (一般在调用 CUDA 函数时出现); 使用 `cudaDeviceSynchronize` 函数获取 asynchronous error (一般在 kernel 执行时出现)。下面给出一个板子, 可以在别的 CUDA 程序中加入如下代码, 帮助检查错误。

---

```

1  #include <stdio.h>
2  #include <assert.h>
```

---

---

```

3  inline cudaError_t checkCuda(cudaError_t result)
4  {
5      if (result != cudaSuccess) {
6          fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
7          assert(result == cudaSuccess);
8      }
9      return result;
10 }
11 int main()
12 {
13     checkCuda( cudaDeviceSynchronize() )
14 }

```

---

### 2.2.2 例题

在如下代码中，需要打印出 synchronous 和 asynchronous 两种 error。

---

```

1  #include <stdio.h>
2  //在 main 函数中
3  cudaError_t syncErr, asyncErr; //声明两类 error
4
5  doubleElements<<<number_of_blocks, threads_per_block>>>(a, N);
6
7  /*
8   * Catch errors for both the kernel launch above and any
9   * errors that occur during the asynchronous `doubleElements`
10  * kernel execution.
11  */
12
13 syncErr = cudaGetLastError(); //使用 cudaGetLastError 获取 syncerr
14 asyncErr = cudaDeviceSynchronize(); //使用 cudaDeviceSynchronize 获取 asyncerr
15
16 if (syncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString(syncErr));
17 if (asyncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString(asyncErr));

```

---

## 2.3 总结

如图2.3所示，完成了 Final Exercise: Accelerate Vector Addition Application。

```
In [11]: !nvcc -arch=sm_70 -o vector-add 07-vector-add/01-vector-add.cu -run
SUCCESS! All values added correctly.
```

图 2.3: final exercise

## 2.4 进阶: Grids and Blocks of 2 and 3 Dimensions

### 2.4.1 知识点

当数据为高维 (如二维矩阵) 时, 使用高维的 grid 和 block 可以方便任务划分。声明方式如下:

---

```
1 //声明二维 block 和 grid
2 dim3 threads_per_block(16, 16, 1);
3 dim3 number_of_blocks(16, 16, 1);
4 someKernel<<<number_of_blocks, threads_per_block>>>();
5
6 //声明三维 block 和 grid
7 dim3 threads_per_block(16, 16, 2);
8 dim3 number_of_blocks(16, 16, 2);
9 someKernel<<<number_of_blocks, threads_per_block>>>();
```

---

当声明 dim3 类型变量的第 3 个参数为 1 时, 表明声明了一个实际为二维的变量; 参数 >1 时声明了三维变量。

### 2.4.2 例题

使用二维 grid 和 block 处理矩阵乘法的核心代码如下:

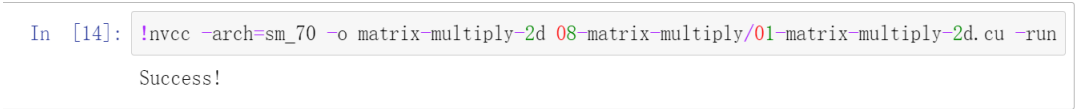
---

```
1 __global__ void matrixMulGPU( int * a, int * b, int * c )
2 {
3     int val = 0;
4
5     int row = blockIdx.x * blockDim.x + threadIdx.x;
6     int col = blockIdx.y * blockDim.y + threadIdx.y;
7
8     if (row < N && col < N)
9     {
10         for ( int k = 0; k < N; ++k )
11             val += a[row * N + k] * b[k * N + col];
12         c[row * N + col] = val;
13     }
14 }
15 void matrixMulCPU( int * a, int * b, int * c )
```

---

```
16 {
17     int val = 0;
18
19     for( int row = 0; row < N; ++row )
20         for( int col = 0; col < N; ++col )
21         {
22             val = 0;
23             for ( int k = 0; k < N; ++k )
24                 val += a[row * N + k] * b[k * N + col];
25             c[row * N + col] = val;
26         }
27 }
28
29 //main 函数中
30 dim3 threads_per_block (16, 16, 1); // A 16 x 16 block threads
31 dim3 number_of_blocks ((N / threads_per_block.x) + 1, (N / threads_per_block.y) + 1, 1);
32 matrixMulGPU <<< number_of_blocks, threads_per_block >>> ( a, b, c_gpu );
33 cudaDeviceSynchronize(); // Wait for the GPU to finish before proceeding
```

如图2.4所示，通过测试。



```
In [14]: !nvcc -arch=sm_70 -o matrix-multiply-2d 08-matrix-multiply/01-matrix-multiply-2d.cu -run
Success!
```

图 2.4: 进阶内容

## 3 Managing Accelerated Application Memory with CUDA Unified Memory and nsys

### 3.1 Profile an Application with nsys

本小节主要学习了使用 nsys 分析程序性能的方法。首先使用!nvcc -o multi-thread-vector-add 01-vector-add/01-vector-add.cu -run 编译.cu 文件，然后使用!nsys profile -stats=true ./multi-thread-vector-add 进行性能分析。在结果中可看到 CUDA API Statistics, CUDA Kernel Statistics, CUDA Memory Operation Statistics (by time), CUDA Memory Operation Statistics (by size in KiB) 和 Operating System Runtime API Statistics。在前部分实验中主要关注 CUDA Kernel Statistics，后半部分主要关注 CUDA Memory Operation Statistics。

### 3.2 Streaming Multiprocessors

Streaming Multiprocessors 又称 SMs，是 threads 执行的地方。当每个 grid 含有的 blocks 数是 GPU 的 SMs 数的倍数时，性能会有所提升；此外，每个 block 含有的 threads 数是 32 的倍数时性能也会提升。



### 3.3 Query the Device

获取 SMs 数可以使用如下代码：

---

```
1  int deviceId;
2  int numberOfSMs;
3
4  cudaGetDevice(&deviceId);
5  cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);
```

---

此外，还有一些获取设备信息的函数，使用方法如下：

---

```
1  #include <stdio.h>
2  int main()
3  {
4      /*
5       * Device ID is required first to query the device.
6       */
7      int deviceId;
8      cudaGetDevice(&deviceId);
9
10     cudaDeviceProp props;
11     cudaGetDeviceProperties(&props, deviceId);
12
13     /*
14      * `props` now contains several properties about the current device.
15      */
16
17     int computeCapabilityMajor = props.major;
18     int computeCapabilityMinor = props.minor;
19     int multiProcessorCount = props.multiProcessorCount;
20     int warpSize = props.warpSize;
21
22     printf("Device ID: %d\n", deviceId);
23     printf("Number of SMs: %d\n", multiProcessorCount);
24     printf("Compute Capability Major: %d\n", computeCapabilityMajor);
25     printf("Compute Capability Minor: %d\n", computeCapabilityMinor);
26     printf("Warp Size: %d\n", warpSize);
27 }
```

---

### 3.4 Unified Memory

当使用 `cudaMallocManaged` 函数分配内存时，它既不在 `host` 也不在 `device` 端，而是等到第一次调用它的端口（不失一般性，假设为 `device`）尝试获取该内存时，将该内存移动至 `device` 端并发生 `page fault`。此后，当 `host` 端需要使用时将移动该内存并发生 `page fault`。

在 `perf` 分析时，如果有很多小批量的内存移动操作，就说明发生了 `on-demand page fault`。`page fault` 会导致性能下降，因此在设计程序时需要尽量避免 `page fault`。有 2 种解决方法：

1. 如果只在 `device(host)` 使用数据，就将初始化函数也设为 `kernel(CPU)` 函数。
2. 使用数据预取。

### 3.5 Asynchronous Memory Prefetching

内存预取可以减少 `page fault` 和 `on-demand` 内存移动的开销。程序员可以在程序使用某内存前，将 UM 上的该内容迁移到任意 CPU 或 GPU 设备上。预取通常较大粒度地迁移数据，相比 `on-demand migration` 使用的步数更少。

下面一段 `main` 函数中的代码表示，在 GPU 初始化之前将所有内存预取到 `device` 端，在 GPU 执行完后将所有数据预取到 `host` 端，然后使用 CPU 检查各元素：

---

```
1  #include <stdio.h>
2  int main()
3  {
4      int deviceId;
5      int numberOfSMs;
6
7      cudaGetDevice(&deviceId);
8      cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);
9
10     const int N = 2<<24;
11     size_t size = N * sizeof(float);
12
13     float *a;
14
15     cudaMallocManaged(&a, size);
16
17     //预取到 device 端
18     cudaMemPrefetchAsync(a, size, deviceId);
19
20     size_t threadsPerBlock;
21     size_t numberOfBlocks;
22
23     threadsPerBlock = 256;
24     numberOfBlocks = 32 * numberOfSMs;
```

```
25
26     cudaError_t addVectorsErr;
27     cudaError_t asyncErr;
28
29     //GPU 处理
30     initWith<<<numberOfBlocks, threadsPerBlock>>>(3, a, N);
31
32     kernelFunc<<<numberOfBlocks, threadsPerBlock>>>(a, N);
33
34     addVectorsErr = cudaGetLastError();
35     if(addVectorsErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString(addVectorsErr));
36
37     asyncErr = cudaDeviceSynchronize();
38     if(asyncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString(asyncErr));
39
40     //预取到 host 端
41     cudaMemPrefetchAsync(a, size, cudaCpuDeviceId);
42
43     //CPU 处理
44     checkElementsAre(3, a, N);
45
46     cudaFree(a);
47 }
```

---

### 3.6 Final Exercise

final exercise 是一个 SAXPY (Single Precision  $a*x+b$ ) 问题。任务是：首先修复 bug，然后将 kernel 的执行时间缩短到 200us。

修复 bug 后： 执行时间为 19650.47us。

将单位 block 数设为 SMs 的倍数： 使用如下代码设置：

---

```
1  int deviceId;
2  int numberOfSMs;
3
4  cudaGetDevice(&deviceId);
5  cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);
6
7  int threads_per_block = 1024;
8  int number_of_blocks = 32 * numberOfSMs;
```

---

执行时间为 15812.579us，缩短为原来的 3/4。

**将初始化设置为 kernel 函数：** 执行时间为  $4321.273+436.118=4757.391\mu s$ ，相比最初版本性能提升了 5 倍。

**使用 prefetching：** 在 GPU 执行 init 前将数据预取到 device 端，在 CPU 验证前将数据预取到 host 端：

---

```

1  //prefetch to device
2  cudaMemPrefetchAsync(a, size, deviceId);
3  cudaMemPrefetchAsync(b, size, deviceId);
4  cudaMemPrefetchAsync(c, size, deviceId);
5
6  //GPU work
7
8  //prefetch to host
9  cudaMemPrefetchAsync(a, size, cudaCpuDeviceId);
10 cudaMemPrefetchAsync(b, size, cudaCpuDeviceId);
11 cudaMemPrefetchAsync(c, size, cudaCpuDeviceId);
12
13 //CPU work

```

---

如图3.5所示，执行时间为  $86.43+60.959=147.389\mu s < 200\mu s$ ，完成！

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
92.6	240866094	3	80288698.0	24783	240802191	cudaMallocManaged
6.1	15912368	6	2652061.3	71384	5218727	cudaMemPrefetchAsync
1.2	3102396	3	1034132.0	984431	1105844	cudaFree
0.1	151471	2	75735.5	62528	88943	cudaDeviceSynchronize
0.0	35372	2	17686.0	6802	28570	cudaLaunchKernel

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
58.6	86430	1	86430.0	86430	86430	saxpy(float*, float*, float*)
41.4	60959	1	60959.0	60959	60959	init(float*, float*, float*)

图 3.5: Final Exercise

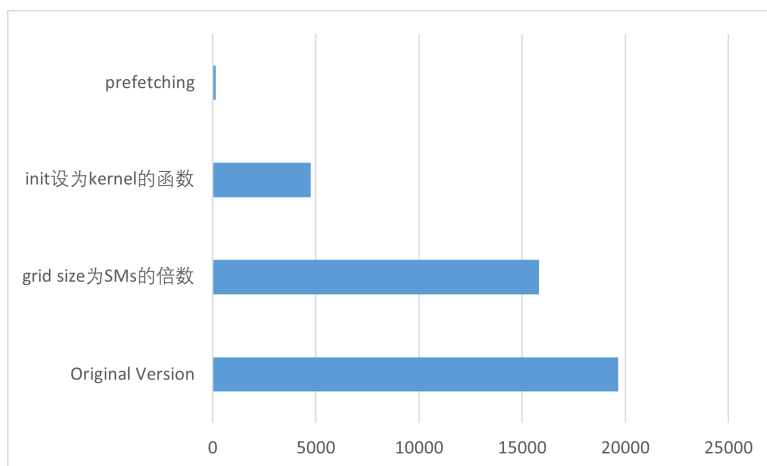


图 3.6: 性能分析

可以看到, 将 `init` 设为 `kernel` 函数对程序的影响最显著 (斜率最大), 数据预取的影响较大, 设置 `grid size` 的影响相对较小, 但上述三者均能大幅度提升性能。最后性能提升了 134 倍, 相比之前的 SIMD、Pthread 和 OpenMP 可看出 CUDA 编程极大的优越性。

## 4 Asynchronous Streaming, and Visual Profiling for Accelerated Applications with CUDA C/C++

### 4.1 Nsight Systems Profiling

在 Nsight System 中, 我比较了 3 个版本的代码: 1. 在 GPU 调用 `init` 核函数前没有将数据预取到 device, 在 CPU 调用检查函数前没有将数据预取到 host; 2. 在 GPU 调用 `init` 核函数前将数据预取到 device, 但 CPU 调用前没有预取到 host; 3. 在 GPU 调用前将数据预取到了 device, 在 CPU 调用前预取到了 host。下面使用 Nsight System 来图形化地比较这 3 个版本代码的执行区别。

图4.7为代码 1 的分析结果, 图4.8为代码 1 前半段的 page fault 以及 host to device 的数据传输, 图4.9为代码 1 后半段的 page fault 以及 device to host 的数据传输。

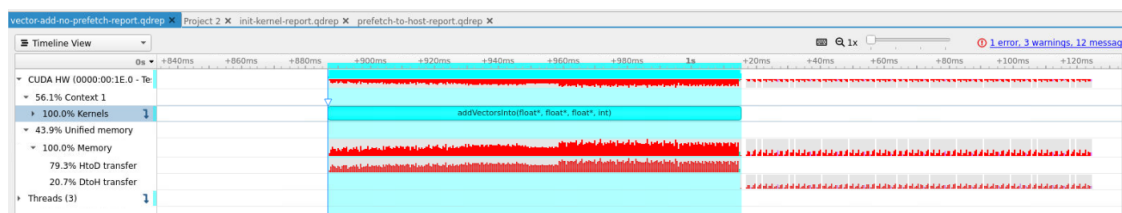


图 4.7: 代码 1 分析结果



图 4.8: 代码 1: HtoD



图 4.9: 代码 1: DtoH

图4.10为代码 2 的分析结果，图4.11为代码 2 后半段的 page fault 以及 device to host 的数据传输。

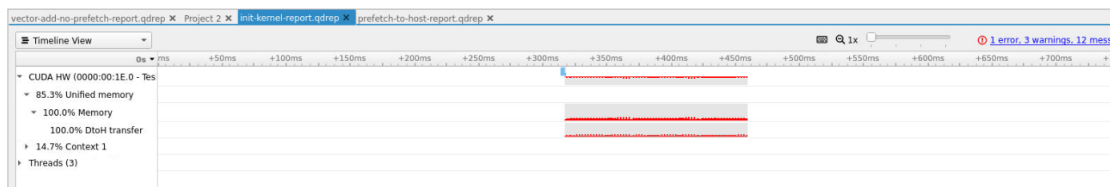


图 4.10: 代码 2 分析结果



图 4.11: 代码 2: DtoH

图4.12为代码 3 的分析结果，图4.13为代码 3 后半段的 page fault 以及 device to host 的数据传输。

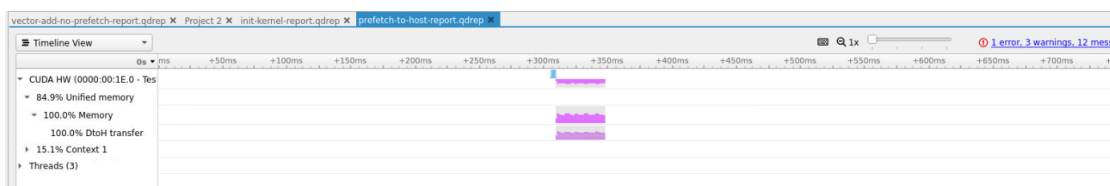


图 4.12: 代码 3 分析结果

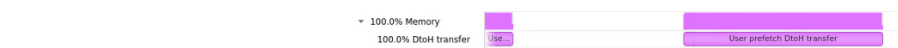


图 4.13: 代码 3: DtoH

通过上图可以看到，代码 1 和代码 2 中均有 page fault，是在 page fault 之后进行的相应 prefetch，且 prefetch 处的 migration cause 为”speculative prefetch”，表示程序自动预取；而代码 3 中没有 page fault，且数据传输为”user prefetch transfer”，表示用户自定义的数据传输。这个分析结果符合代码。

## 4.2 Stream

### 4.2.1 知识点

stream 指一系列顺序执行的指令，CUDA 中的 kernel 执行、内存移动都在 stream 上发生。当不显示声明、指定 stream 时，使用 default stream；用户也可以自己声明、指定 non-default stream。使用多个 streams 可以提高程序性能。

CUDA 中 stream 需要遵守如下规则：

1. 每个 stream 内的指令顺序执行；
2. 不同的 non-default streams 间的指令没有顺序要求；
3. default stream 上的指令执行前需等待其他 streams 上的指令执行完，且 default stream 上指令执行时其他 streams 上的指令不能执行。如图4.14所示。

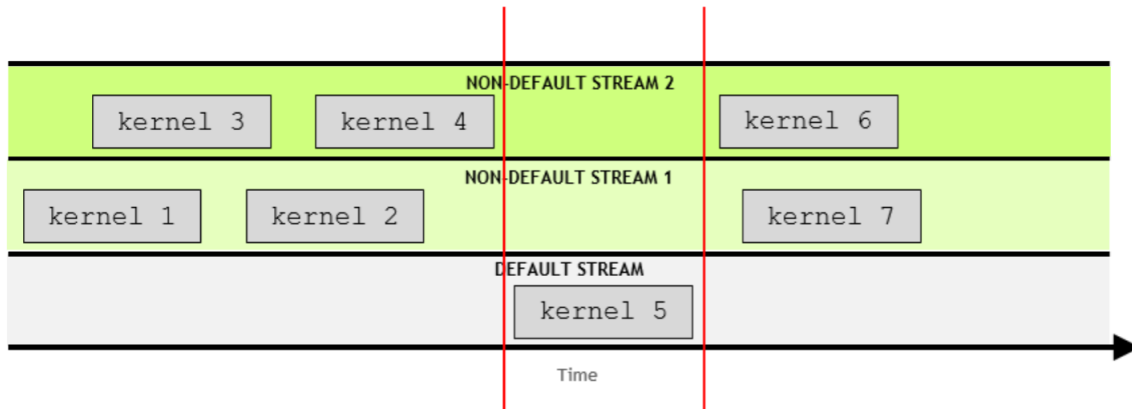


图 4.14: stream 工作规则

可以用如下板子生成 non-default stream：

---

```

1  cudaStream_t stream;           // CUDA streams are of type `cudaStream_t`.
2  cudaStreamCreate(&stream); // Note that a pointer must be passed to `cudaCreateStream`.
3
4  someKernel<<<number_of_blocks, threads_per_block, 0, stream>>>();
5  // `stream` is passed as 4th EC argument.
6
7  cudaStreamDestroy(stream);
8  // Note that a value, not a pointer, is passed to `cudaDestroyStream`.

```

---

#### 4.2.2 例题

如下代码，前一个在 default stream 上串行地执行 5 个 kernel 函数，后一个在 5 个 non-default streams 上并行地执行 5 个 kernel 函数：

---

```

1  //串行
2  for (int i = 0; i < 5; ++i){
3      printNumber<<<1, 1 >>>(i);
4  }
5  cudaDeviceSynchronize();
6  //并行
7  for (int i = 0; i < 5; ++i){
8      cudaStream_t stream;

```

```

9      cudaStreamCreate(&stream);
10     printNumber<<<1, 1, 0, stream>>>(i);
11     cudaStreamDestroy(stream);
12 }
13 cudaDeviceSynchronize();

```

图4.15展示了使用 Nsight Systems 分析并行 streams 的图形化结果。可以看到 5 个 streams 上各执行了一个 kernel 函数。

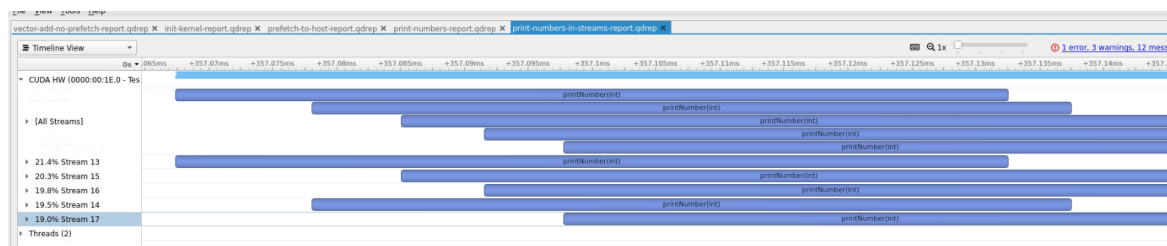


图 4.15: 多 streams 分析结果

### 4.3 Final Exercise

本 exercise 需要加速一个多体模拟器。我使用了 2 个加速方法：1. 将 bodyForce 和 add 都改为 kernel 函数；2. 获取 numberOfSMs，然后设置 numberOfBlocks = 32 \* numberOfSMs。如图4.16所示，通过测试。

```

In [22]: run_assessment()

Running nbody simulator with 4096 bodies
-----

Application should run faster than 0.9s
Your application ran in: 0.1602s
Your application reports 16.641 Billion Interactions / second

Your results are correct

Running nbody simulator with 65536 bodies
-----

Application should run faster than 1.3s
Your application ran in: 0.4954s
Your application reports 114.086 Billion Interactions / second

Your results are correct

Congratulations! You passed the assessment!
See instructions below to generate a certificate, and see if you can accelerate the simulator even more!

```

图 4.16: Final Exercise

除上述优化方法外，我还想到了 2 种优化方法：1. 使用数据预取，在 GPU 执行前预取到 device 端，在 CPU 执行前预取到 host 端；2. 使用多 streams 提高并发度。

## 5 高斯消去

使用 CUDA 加速高斯消去。

代码见 GitHub 链接<https://github.com/Skyyyyy0920/Parallel-Programming/tree/main/CUDA>。具体数据如图5.17所示。



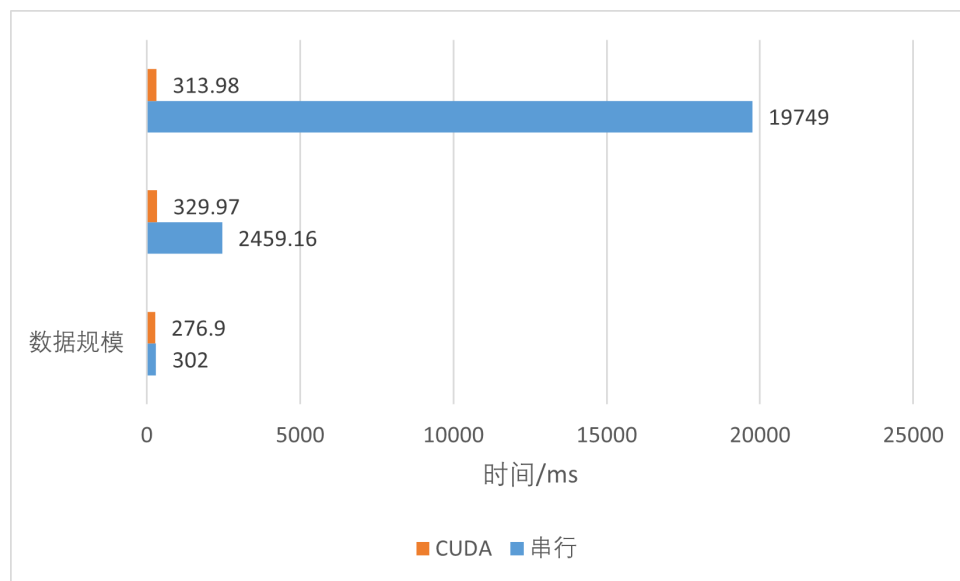


图 5.17: 高斯消去结果

可以看到，随着矩阵规模增大（500，1000，2000），加速比从 1.09 倍到 7.45 倍再到 62.89 倍。