计算机学院

并行程序设计 GPU 作业

NVIIDA Accelerating Applications with CUDA C/C++ 课程

姓名：林语盈

学号：2012174

专业：计算机科学与技术

2022 年 6 月 11 日

摘要

本文介绍了 NVIIDA Accelerating Applications with CUDA C/C++ 课程的主要学习内容并附课程完成截图。

关键字: NVIDIA; CUDA; GPU

目录

1 学习过程和内容

1.1 nvcc 基本命令

查看 GPU 信息

```
!nvidia-smi
```

编译并运行

```
!nvcc -arch=sm_70 -o name path/hello.cu -run
//-run指定需运行
```

1.2 CUDA 基本编程框架

cpu 端即 host 端，gpu 端即 device 端。通过 host 端，使函数在 device 端运行并返回。使用 __global__ 指定 GPU 端的函数，使用 GPUFunctionblock_num, threads_per_block () 对 GPU 进行调用。若不加说明，CPU 与 GPU 两端的程序将同时执行, 即可能 GPU 尚未结束时，CPU 端已经结束程序并返回。故，使用 cudaDeviceSynchronize() 进行同步，使 CPU 端等待 GPU 返回。

GPUFunctionblock_num, threads_per_block () 的参数指明了分配的 block 数与 threads 数，总线程数为 block_num * threads_per_block。多个线程常常用于将循环展开, 假设循环数为 N, 要处理线程数与 N 不同的情况, 并重新设计循环, 注意每个 block 内的线程数存在上限。此外，使用 threadIdx.x 获得当前的线程编号，使用 blockIdx.x 获得当前的 block 编号，使用 blockDim.x 获得当前 block 维度，使用 gridDim.x 获得当前 grid 维度。实际上，其内存结构均为 2 维，可以设计 2 维的数据分配方式。

GPU 拥有独立的存储空间，使用 cudaMallocManaged(a, size) 进行分配，使用 cudaFree(a) 进行释放。

综上，CUDA 基本编程框架如下所示:

CUDA 基本编程框架

```
#include <stdio.h>
__global__
void GPUfunction(float *a, int N)
{
  printf("this is from GPU");
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;

  for(int i = index; i < N; i += stride)
  {
    //do something with a[i]
    a[i] = a[i]*2;
  }
}

void CPUfunction(float *a, int N)
{
```

```
    printf("this is from CPU");
    for(int i = 0; i < N; ++i)
    {
        a[i] = i;
    }
}

int main()
{
  const int N = 2<<20;
  size_t size = N * sizeof(float);
  float *a;

  cudaMallocManaged(&a, size);

  initWith(a, N);

  size_t threadsPerBlock = 256;
  size_t numberOfBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;

  addVectorsInto<<<numberOfBlocks, threadsPerBlock>>>(a, N);

  cudaDeviceSynchronize();

  //check(a, N);
  cudaFree(a)
}
```

对于每个 cuda 系统函数的调用，都应该进行 error 的处理，可使用如下函数进行。

error 的处理

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

inline cudaError_t checkCuda(cudaError_t result)
{
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
  return result;
}

int main()
{
...
  checkCuda( cudaMallocManaged(&a, size) );
...
  addVectorsInto<<<numberOfBlocks, threadsPerBlock>>>(...);
```

```

  checkCuda( cudaGetLastError() );
  checkCuda( cudaDeviceSynchronize() );
...
  checkCuda( cudaFree(a) );

}
```

1.3 nsys 性能分析工具

使用 nsys 进行性能分析

```
!nsys profile --stats=true ./hello
```

–stats=true 指定在屏幕上输出，./hello 为待分析的可执行程序。

1.4 CUDA 程序性能优化

1.4.1 SM

SM 即流多处理器，每个 SM 可以处理一个 block，一个 GPU 上拥有一定数量的 SM。性能提升策略：选择恰当的 grid_size，使之是给定的 GPU 上 SM 数的倍数。使用如下代码可以获取相关的系统参数。

获取相关的系统参数

```
int deviceId;
cudaGetDevice(&deviceId);                     // 获取当前GPU编号

cudaDeviceProp props;
cudaGetDeviceProperties(&props, deviceId); // 获取当前GPU相关参数

 //可以如下设计线程数，N=256*numberOfSMs * 32
  cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);
  int threads_per_block = 256;
  int number_of_blocks = numberOfSMs * 32;
```

props 中的相关参数具体可参见`https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html`

1.4.2 UM, 数据访存和移动开销

当数据在 host 或 device 端被初次访问时，需要进行数据移动，这造成了很大的开销，应当设计函数的运行位置（host、device）和顺序，尽量减少数据的移动。

1.4.3 asynchronous memory prefetching

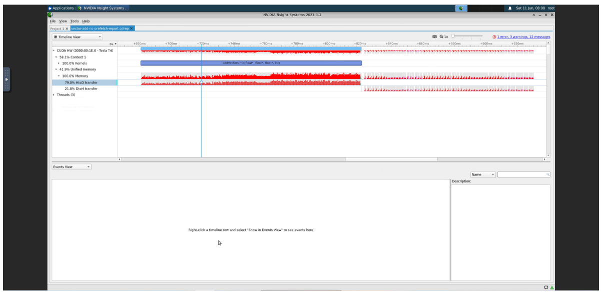
优化策略：无论 cpu 还是 gpu 端，在访问任何数据之前，首先进行异步内存预取。

异步内存预取

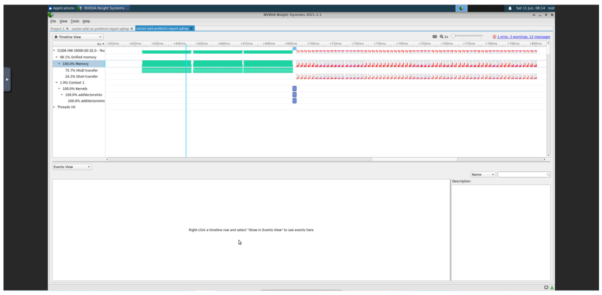
```
cudaMemPrefetchAsync(pointerToSomeUMData, size, deviceId);        // 预取数据至 to GPU device, deviceId由上述函数获取
cudaMemPrefetchAsync(pointerToSomeUMData, size, cudaCpuDeviceId); // 预取数据 to host. cudaCpuDeviceId是内置CUDA变量
```

1.5 Nsight Systems

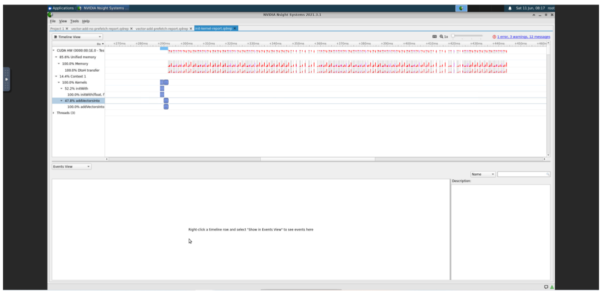
使用 Nsight Systems 工具加载 nsys profile 生成的.qdrep 文件，即可方便的将程序各部分的运行细节可视化并进行分析。附录中的几个程序分析截图如1.1所示。



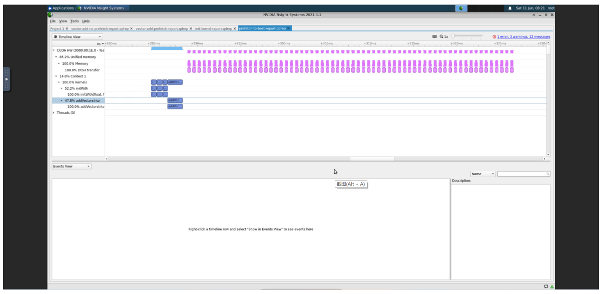
(a) 无预取



(b) 预取至 GPU



(c) 将 init 函数在 GPU 端运行



(d) check 前增加预取

图 1.1: 向量加法程序 Nsight Systems 分析

1.6 CUDA streams

在默认情况下，kernal 函数在 default stream 上运行。但 cuda 提供了对每个 kernal 函数指定特定 stream 的 API。在同一 stream 内，kernal 函数串行运行，但不同 stream 上的 kernal 函数可以并行运行。default stream 可以进行全部 streams 的同步操作。同样，Nsight Systems 工具可以查看每个 stream 的运行情况。

CUDA streams

```
    cudaStream_t stream;//定义stream
    cudaStreamCreate(&stream);//创建stream
    GPUfunction<<<1, 1, 0, stream>>>();//第四个参数指定了该核函数的stream
    //第三个参数指定分配给每个块的共享内存的字节数，这里为0
    cudaStreamDestroy(stream);//销毁stream
```

1.7 手动分配内存

CUDA 提供了手动分配 host 和 device 内存的 API, 使用 cudaMalloc 与 cudaMallocHost 而不再使用 cudaMallocManaged, 程序员可以手动在两端分配和回收内存, 并在需要的时候通过 cudaMemcpy 进行拷贝。

手动分配内存程序框架

```
int *host_a, *device_a;        // Define host-specific and device-specific arrays.
cudaMalloc(&device_a, size);   // 'device_a' is immediately available on the GPU.
cudaMallocHost(&host_a, size); // 'host_a' is immediately available on CPU, and is
    page-locked, or pinned.

initializeOnHost(host_a, N);   // No CPU page faulting since memory is already
    allocated on the host.

// cudaMemcpy takes the destination, source, size, and a CUDA-provided variable for
    the direction of the copy.
cudaMemcpy(device_a, host_a, size, cudaMemcpyHostToDevice);

kernel<<<blocks, threads, 0, someStream>>>(device_a, N);

// cudaMemcpy can also copy data from device to host.
cudaMemcpy(host_a, device_a, size, cudaMemcpyDeviceToHost);

verifyOnHost(host_a, N);

cudaFree(device_a);
cudaFreeHost(host_a);          // Free pinned memory like this.
```

1.8 异步内存拷贝

可以将 cudaMemcpy 替换为 cudaMemcpyAsync 进行异步内存拷贝，与数据的运算形成流水线，优化程序性能。但异步时的步长和数据输入应当精心设计，并分配新的 stream。下例为 4 步内存拷贝。

异步内存拷贝

```
/*
非异步的内存拷贝:
  GPUfunction<<<numberOfBlocks, threadsPerBlock>>>(c, a, N);
cudaMemcpy(h_c, c, size, cudaMemcpyDeviceToHost);
*/
  for (int i = 0; i < 4; ++i)
  {
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    GPUfunction<<<numberOfBlocks/4, threadsPerBlock, 0, stream>>>(&c[i*N/4],
        &a[i*N/4], N/4);
```

```
    cudaMemcpyAsync(&h_c[i*N/4], &c[i*N/4], size/4, cudaMemcpyDeviceToHost,
        stream);
    cudaStreamDestroy(stream);
  }
  /*
  const int numberOfSegments = 4;                    // This example demonstrates
      slicing the work into 4 segments.
int segmentN = N / numberOfSegments;               // A value for a segment's worth
    of N is needed.
size_t segmentSize = size / numberOfSegments;      // A value for a segment's worth
    of size is needed.

// For each of the 4 segments...
for (int i = 0; i < numberOfSegments; ++i)
{
  // Calculate the index where this particular segment should operate within the
      larger arrays.
  segmentOffset = i * segmentN;

  // Create a stream for this segment's worth of copy and work.
  cudaStream_t stream;
  cudaStreamCreate(&stream);

  // Asynchronously copy segment's worth of pinned host memory to device over
      non-default stream.
  cudaMemcpyAsync(&device_array[segmentOffset],  // Take care to access correct
      location in array.
                  &host_array[segmentOffset],    // Take care to access correct
                      location in array.
                  segmentSize,                   // Only copy a segment's worth of
                      memory.
                  cudaMemcpyHostToDevice,
                  stream);                       // Provide optional argument for
                      non-default stream.

  // Execute segment's worth of work over same non-default stream as memory copy.
  kernel<<<number_of_blocks, threads_per_block, 0,
      stream>>>(&device_array[segmentOffset], segmentN);

  // cudaStreamDestroy will return immediately (is non-blocking), but will not
      actually destroy stream until
  // all stream operations are complete.
  cudaStreamDestroy(stream);
}
  */
```

附录 A 课程学习截图证明

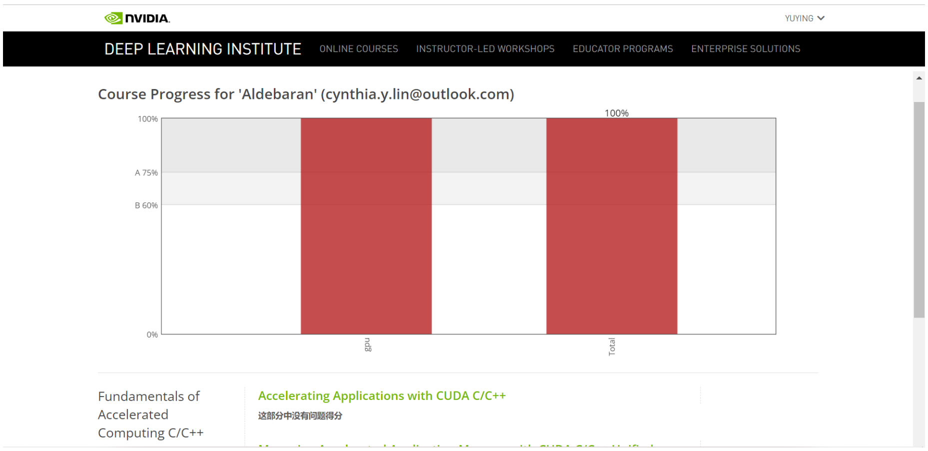


图 A.2: 课程学习截图证明 1

图 A.3: 课程学习截图证明 2

附录 B 向量加法代码

B.1 无预取

include <stdio.h>

void initWith(float num, float *a, int N) for(int i = 0; i < N; ++i) a[i] = num;

$^{global}{}_{voidaddVectorsInto(float*result,float*a,float*b,intN)intindex=threadIdx.x+blockIdx.x*blockDim.x;intstride=blockDim.x*gridDim.x;}$

for(int i = index; i < N; i += stride) result[i] = a[i] + b[i];

void checkElementsAre(float target, float *vector, int N) for(int i = 0; i < N; i++) if(vector[i]

!= target) printf("FAIL: vector[exit(1); printf("Success! All values calculated correctly.");

int main() int deviceId; int numberOfSMs;

cudaGetDevice(deviceId); cudaDeviceGetAttribute(numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);

const int N = 2«24; size$_t size = N * sizeof(float)$;

float *a; float *b; float *c;

cudaMallocManaged(a, size); cudaMallocManaged(b, size); cudaMallocManaged(c, size);

initWith(3, a, N); initWith(4, b, N); initWith(0, c, N);

size$_t threadsPerBlock; size_t numberOfBlocks$;

threadsPerBlock = 256; numberOfBlocks = 32 * numberOfSMs;

cudaError$_t addVectorsErr; cudaError_t asyncErr$;

addVectorsInto«<numberOfBlocks, threadsPerBlock»>(c, a, b, N);

addVectorsErr = cudaGetLastError(); if(addVectorsErr != cudaSuccess) printf("Error:

asyncErr = cudaDeviceSynchronize(); if(asyncErr != cudaSuccess) printf("Error:

checkElementsAre(7, c, N);

cudaFree(a); cudaFree(b); cudaFree(c);

B.2 取至 GPU

include <stdio.h>

void initWith(float num, float *a, int N) for(int i = 0; i < N; ++i) a[i] = num;

$_{global}{}_{voidaddVectorsInto(float*result,float*a,float*b,intN)intindex=threadIdx.x+blockIdx.x*blockDim.x;intstride=blockDim.x*gridDim.x;}$

for(int i = index; i < N; i += stride) result[i] = a[i] + b[i];

void checkElementsAre(float target, float *vector, int N) for(int i = 0; i < N; i++) if(vector[i] != target) printf("FAIL: vector[exit(1); printf("Success! All values calculated correctly.");

int main() int deviceId; int numberOfSMs;

cudaGetDevice(deviceId); cudaDeviceGetAttribute(numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);

const int N = 2«24; size$_t size = N * sizeof(float)$;

float *a; float *b; float *c;

cudaMallocManaged(a, size); cudaMallocManaged(b, size); cudaMallocManaged(c, size);

initWith(3, a, N); initWith(4, b, N); initWith(0, c, N);

cudaMemPrefetchAsync(a, size, deviceId); cudaMemPrefetchAsync(b, size, deviceId); cudaMemPrefetchAsync(c, size, deviceId);

size$_t threadsPerBlock; size_t numberOfBlocks$;

threadsPerBlock = 256; numberOfBlocks = 32 * numberOfSMs;

cudaError$_t addVectorsErr; cudaError_t asyncErr$;

addVectorsInto«<numberOfBlocks, threadsPerBlock»>(c, a, b, N);

addVectorsErr = cudaGetLastError(); if(addVectorsErr != cudaSuccess) printf("Error:

asyncErr = cudaDeviceSynchronize(); if(asyncErr != cudaSuccess) printf("Error:

checkElementsAre(7, c, N);

cudaFree(a); cudaFree(b); cudaFree(c);

B.3 将 init 函数在 GPU 端运行

include <stdio.h>

$_{g}^{lobal}{}_{voidinitWith(floatnum,float*a,intN)}$
int index = threadIdx.x + blockIdx.x * blockDim.x; int stride = blockDim.x * gridDim.x;

for(int i = index; i < N; i += stride) a[i] = num;

$_{g}^{lobal}{}_{voidaddVectorsInto(float*result,float*a,float*b,intN)intindex=threadIdx.x+blockIdx.x*blockDim.x;intstride=blockDim.x*gridDim.x;}$
for(int i = index; i < N; i += stride) result[i] = a[i] + b[i];

void checkElementsAre(float target, float *vector, int N) for(int i = 0; i < N; i++) if(vector[i] != target) printf("FAIL: vector[exit(1); printf("Success! All values calculated correctly.");

int main() int deviceId; int numberOfSMs;

cudaGetDevice(deviceId); cudaDeviceGetAttribute(numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);

const int N = 2«24; size$_tsize = N * sizeof(float)$;

float *a; float *b; float *c;

cudaMallocManaged(a, size); cudaMallocManaged(b, size); cudaMallocManaged(c, size);

cudaMemPrefetchAsync(a, size, deviceId); cudaMemPrefetchAsync(b, size, deviceId); cudaMemPrefetchAsync(c, size, deviceId);

size$_tthreadsPerBlock; size_tnumberOfBlocks$;

threadsPerBlock = 256; numberOfBlocks = 32 * numberOfSMs;

cudaError$_taddVectorsErr; cudaError_tasyncErr$;

initWith«<numberOfBlocks, threadsPerBlock»>(3, a, N); initWith«<numberOfBlocks, threadsPerBlock»>(4, b, N); initWith«<numberOfBlocks, threadsPerBlock»>(0, c, N);

addVectorsInto«<numberOfBlocks, threadsPerBlock»>(c, a, b, N);

addVectorsErr = cudaGetLastError(); if(addVectorsErr != cudaSuccess) printf("Error:

asyncErr = cudaDeviceSynchronize(); if(asyncErr != cudaSuccess) printf("Error:

checkElementsAre(7, c, N);

cudaFree(a); cudaFree(b); cudaFree(c);

B.4 heck 前增加预取

include <stdio.h>

$_{g}^{lobal}{}_{voidinitWith(floatnum,float*a,intN)}$
int index = threadIdx.x + blockIdx.x * blockDim.x; int stride = blockDim.x * gridDim.x;

for(int i = index; i < N; i += stride) a[i] = num;

$_{g}^{lobal}{}_{voidaddVectorsInto(float*result,float*a,float*b,intN)intindex=threadIdx.x+blockIdx.x*blockDim.x;intstride=blockDim.x*gridDim.x;}$
for(int i = index; i < N; i += stride) result[i] = a[i] + b[i];

void checkElementsAre(float target, float *vector, int N) for(int i = 0; i < N; i++) if(vector[i] != target) printf("FAIL: vector[exit(1); printf("Success! All values calculated correctly.");

int main() int deviceId; int numberOfSMs;

cudaGetDevice(deviceId); cudaDeviceGetAttribute(numberOfSMs, cudaDevAttrMultiProcessorCount, deviceId);

const int N = 2«24; size$_tsize = N * sizeof(float)$;

float *a; float *b; float *c;

cudaMallocManaged(a, size); cudaMallocManaged(b, size); cudaMallocManaged(c, size);

cudaMemPrefetchAsync(a, size, deviceId); cudaMemPrefetchAsync(b, size, deviceId); cudaMemPrefetchAsync(c, size, deviceId);

size$_t threadsPerBlock; size_t numberOfBlocks$;

threadsPerBlock = 256; numberOfBlocks = 32 * numberOfSMs;

cudaError$_t addVectorsErr; cudaError_t asyncErr$;

initWith«<numberOfBlocks, threadsPerBlock»>(3, a, N); initWith«<numberOfBlocks, threadsPerBlock»>(4, b, N); initWith«<numberOfBlocks, threadsPerBlock»>(0, c, N);

addVectorsInto«<numberOfBlocks, threadsPerBlock»>(c, a, b, N);

addVectorsErr = cudaGetLastError(); if(addVectorsErr != cudaSuccess) printf("Error:

asyncErr = cudaDeviceSynchronize(); if(asyncErr != cudaSuccess) printf("Error:

cudaMemPrefetchAsync(c, size, cudaCpuDeviceId);

checkElementsAre(7, c, N);

cudaFree(a); cudaFree(b); cudaFree(c);

参考文献