**CUDA编程框架**

1.使用cudaSetDevice(0)选择GPU

2.使用cudaMalloc函数为device上的变量分配内存

3.使用cudaMemcpy函数将host上的数据拷贝到GPU上，如：

cudaStatus = cudaMemcpy(dev\_a, host\_a, size \* sizeof(int), cudaMemcpyHostToDevice);

4.调用核函数，格式：addKernel<<<1, size>>>(dev\_c, dev\_a, dev\_b);其中，括号中第一个参数为num\_blocks，第二个参数为num\_threads

核函数的定义：

\_\_global\_\_ void addKernel(int \*c, const int \*a, const int \*b)

{

int i = threadIdx.x;

c[i] = a[i] + b[i];

}

5.使用cudaDeviceSynchronize()函数等待kernel完成计算

6.调用cudaMemcpy函数将device上的数据拷贝到CPU上，如：

cudaStatus = cudaMemcpy(host\_c, dev\_c, size \* sizeof(int), cudaMemcpyDeviceToHost);

7.实例：

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

cudaError\_t addWithCuda(int \*c, const int \*a, const int \*b, unsigned int size);

\_\_global\_\_ void addKernel(int \*c, const int \*a, const int \*b)

{

int i = threadIdx.x;

c[i] = a[i] + b[i];

}

int main()

{

const int arraySize = 5;

const int a[arraySize] = { 1, 2, 3, 4, 5 };

const int b[arraySize] = { 10, 20, 30, 40, 50 };

int c[arraySize] = { 0 };

// Add vectors in parallel.

cudaError\_t cudaStatus = addWithCuda(c, a, b, arraySize);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addWithCuda failed!");

return 1;

}

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",

c[0], c[1], c[2], c[3], c[4]);

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaStatus = cudaDeviceReset();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceReset failed!");

return 1;

}

return 0;

}

// Helper function for using CUDA to add vectors in parallel.

cudaError\_t addWithCuda(int \*c, const int \*a, const int \*b, unsigned int size)

{

int \*dev\_a = 0;

int \*dev\_b = 0;

int \*dev\_c = 0;

cudaError\_t cudaStatus;

// Choose which GPU to run on, change this on a multi-GPU system.

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");

goto Error;

}

// Allocate GPU buffers for three vectors (two input, one output) .

cudaStatus = cudaMalloc((void\*\*)&dev\_c, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_a, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_b, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

// Copy input vectors from host memory to GPU buffers.

cudaStatus = cudaMemcpy(dev\_a, a, size \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

cudaStatus = cudaMemcpy(dev\_b, b, size \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

// Launch a kernel on the GPU with one thread for each element.

addKernel<<<1, size>>>(dev\_c, dev\_a, dev\_b);

// Check for any errors launching the kernel

cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));

goto Error;

}

// cudaDeviceSynchronize waits for the kernel to finish, and returns

// any errors encountered during the launch.

cudaStatus = cudaDeviceSynchronize();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);

goto Error;

}

// Copy output vector from GPU buffer to host memory.

cudaStatus = cudaMemcpy(c, dev\_c, size \* sizeof(int), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

Error:

cudaFree(dev\_c);

cudaFree(dev\_a);

cudaFree(dev\_b);

return cudaStatus;

}