Parallel Computing

#1 Homework Solutions

محمد حسین خوشه چین - ۹۹۲۱۰۱۶۴

۷ فروردین ۱۴۰۰

توضیحات

برای حل سوالات این سری از تمارین، از دو کتاب [۱] و [۲] کمک گرفته شده است.

تمرین یک : الگوریتم ضرب ۲ ماتریس

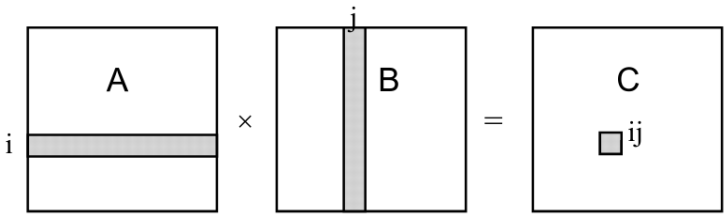
در بخش ۵.۶ کتاب Parhami الگوریتمی برای ضرب ماتریس ها در مدل PRAM با رویکرد Data Parallelism ارائه شده است که ابتدا به شرح آن می پردازیم و سپس این الگوریتم را به زبان CUDA می نویسیم. این الگوریتم در شکل ۱ آمده است.

PRAM matrix multiplication algorithm using m² processors

Processor $(i, j), 0 \leq i, j < m,$ do
begin
  $t := 0$
  for $k = 0$ to $m - 1$ do
    $t := t + a_{ik} b_{kj}$
  endfor
  $c_{ij} := t$
end

شکل ۱: الگوریتم ضرب ماتریس ها

بر اساس این الگوریتم برای بدست آوردن حاصل ضرب دو ماتریس بدین شکل عمل میکنیم که هر پراسسور وظیفه دارد تا حاصل یکی از خانه های ماتریس نهایی را محاسبه کند. در نتیجه با توجه به اینکه سایز ماتریس نهایی $n*n$ است در نهایت ما n^2 خانه داریم پس باید n^2 پراسسور داشته باشیم تا بتوانیم حاصل ضرب این دو ماتریس را به دست آوریم . در شکل ۲ میتوانید یک شمای کلی از این عملیات را مشاهده کنید.



شکل ۲: هر پراسسور وظیفه دارد تا حاصل ضرب یک خانه از ماتریس نهایی را بدست آورد

حال این الگوریتم را به زبان CUDAمینویسیم.اما پیش از آن لازم است ذکر کنم این برنامه را بر روی GPU سیستم خودم اجرا کرده ام. کارت گرافیک سیستم بنده GeForce سری 610m می باشد و مشخصات آن با اجرا کردن برنامه DeviceQuery در شکل ۳ قابل مشاهده است.

```
C:\Windows\system32\cmd.exe
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.5\1_Utilities\deviceQuery\../../bin/win64/Debug/deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce 610M"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 1) Multiprocessors, ( 48) CUDA Cores/MP:     48 CUDA Cores
  GPU Max Clock rate:                            950 MHz (0.95 GHz)
  Memory Clock rate:                             800 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 65536 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.5, CUDA Runtime Version = 7.5, NumDevs = 1, Device0 = GeForce 610M
Result = PASS
Press any key to continue . . .
```

شکل ۳: خروجی برنامه DeviceQuery بر روی سیستم

همانطور که مشخص است با توجه به نوع کارت گرافیک سیستم بنده مجبورم از CUDA ورژن ۷.۵استفاده کنم. برنامه نوشته شده که از کتاب دوم الهام گرفته شده است در شکل ۴ و ۵ آورده شده است.

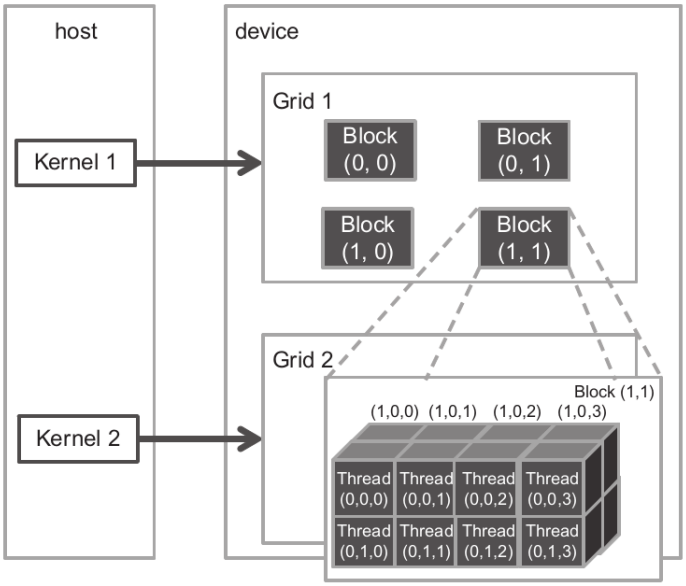
```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
using namespace std;


__global__ void MatricesMultiply(float* x, float* y, float* z, int s) {

    int C = (blockIdx.x * blockDim.x) + threadIdx.x;
    int R = (blockIdx.y * blockDim.y) + threadIdx.y;

    if ((R<s) && (C<s)) {
        float result = 0;
        for (int i = 0; i < s; i++) {
            result += x[R * s + i] * y[i * s + C];
        }
        z[R * s + C] = result;
    }
}

int main(void) {
    int n = 6;
    float* a, * b, * c, * a_d, * b_d, * c_d;

    a = (float*)malloc(sizeof(float) * n * n);
    b = (float*)malloc(sizeof(float) * n * n);
    c = (float*)malloc(sizeof(float) * n * n);

    cudaSetDevice(0);

    cudaMalloc((void**)&a_d, sizeof(float) * n * n);
    cudaMalloc((void**)&b_d, sizeof(float) * n * n);
    cudaMalloc((void**)&c_d, sizeof(float) * n * n);

```

شکل ۴: بخش اول برنامه

```
37      for (int i = 0; i < n * n; i++) {
38          a[i] = i;
39          b[i] = i+1;
40      }
41
42      cudaMemcpy(a_d, a, sizeof(float) * n * n, cudaMemcpyHostToDevice);
43      cudaMemcpy(b_d, b, sizeof(float) * n * n, cudaMemcpyHostToDevice);
44      dim3 dimGrid(std::ceil(n/32.0), std::ceil(n/32.0), 1);
45      dim3 dimBlock(32, 32, 1);
46      MatricesMultiply <<<dimGrid, dimBlock>>>(a_d, b_d, c_d, n);
47      cudaDeviceSynchronize();
48      cudaMemcpy(c, c_d, sizeof(float) * n * n, cudaMemcpyDeviceToHost);
49      cudaFree(a_d);
50      cudaFree(b_d);
51      cudaFree(c_d);
52      printf("The Result is\n\n");
53      int counter = 0;
54      printf("\t | ");
55      for (int i = 0; i < n*n; i++){
56          if(counter == n){
57              printf("\n");
58              printf("\t | ");
59              printf("%f\t", c[i]);
60              counter = 1;
61          }else{
62              printf("%f\t", c[i]);
63              counter++;
64          }
65
66      }
67      printf("\n\n");
68      return 0;
69  }
```

شکل ۵: بخش دوم برنامه

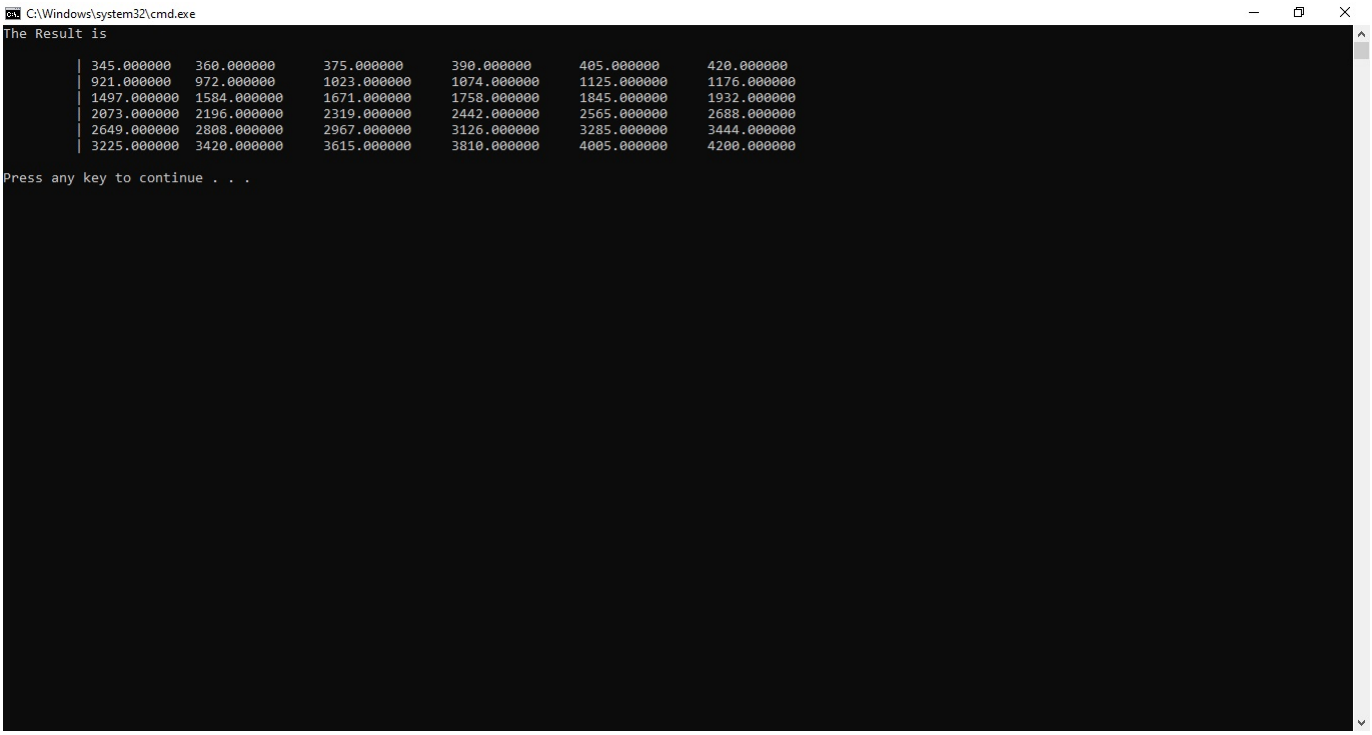
برنامه ما از دو بخش تشکیل شده است. بخش اول که تابع MatricesMultiply می باشد و با کلمه global مشخص شده است را بخش device می گویند و این بخش از کد بر روی GPU اجرا می شود. بخش دوم کد ما که تابع main می باشد را بخش host می گویند که بر روی CPU اجرا می شود. تابع MatricesMultiply را باید در بخش host صدا بزنیم.منتها پیش از آن که بحثمان را ادامه بدهیم باید بگوییم که زمانی که تابع هایی که به عنوان بخش device مینویسم که به آن ها kernel می گویند به چه شکل اجرا می شوند. به شکل ۶ توجه کنید.



شکل ۶: نحوه سازماندهی و شاخص گذاری thread ها در CUDA

در CUDA توابع kernel ما توسط یک تعداد thread که تعداد آن ها را در هنگام صدا زدن توابع مشخص میکنیم اجرا می شود . این thread ها در block هایی قرار میگیرند و این block ها در یک grid قرار می گیرند. هر تابع kernel برای اجرا شدن grid خودش را دارد. هر thread یک ID دارد که از ۶ بخش تشکیل می شود . سه بخش اول آن مشخص میکند که یک thread در کدام block از grid قرار دارد به این شکل که هر block در یک مختصات سه بعدی در نظر گرفته می شود و مختصات آن در محور طول و عرض و ارتفاع مشخص کننده آن block در grid است و به این بخش در کد blockIndex میگوییم. بخش دوم شناسه یک thread نشان دهنده جایگاه آن thread در یک فضای سه بعدی در داخل یک block است و به این بخش در کد threadIndex میگوییم. از این شناسه ۶ بخشی برای assign کردن یک thread به یک خونه از ماتریس برای بدست آوردن جواب نهایی استفاده می کنیم. در برنامه

ای که نوشتیم یک مولفه دیگر وجود دارد به اسم blockDim که به ما نشان میدهد در راستای محور x و y و z چه تعداد thread داریم. چون ما میخواهیم دو ماتریس را در هم ضرب کنیم و هر ماتریس دو بعد دارد می آییم هنگام صدا زدن تابع kernel تعداد block ها در grid را دو بعدی و تعداد thread ها را در یک block به شکل دو بعدی در نظر میگیریم بدین صورت که در سه تایی dimGrid و dimBlock مولفه سوم را ۱ میگذاریم تا از بعد سوم آن دو صرف نظر کنیم. در سه تایی dimBlock مولفه اول و دوم را 32 می گذاریم بدین معنا که در هر block در راستای محور x ها ۳۲ تا thread داریم و در راستای محور y ها هم ۳۲ تا thread داریم یعنی در مجموع ۳۲×۳۲ تا thread در هر block داریم که می شود ۱۰۲۴ تا thread. دقت کنید که در خروجی deviceQuery حداکثر thread ها ممکن در هر block هم ۱۰۲۴ تا بود. حال میخواهیم مولفه های اول و دوم سه تایی dimGrid را مخشص کنیم. این را می دانیم که سایز هر block ثابت است و نمی توانیم block هایی با سایز های متفاوت داشته باشیم. حال ممکن است زمانیکه مابخواهیم به هر خانه ماتریس یک thread بدهیم در نهایت مجبور باشیم تعدادی block داشته باشیم که در آن thread هایی وجود داشته باشد که به هیچ خانه ای assign نکنیم شان. به نوعی آن ها thread های اضافی یک block هستند. حال برای اینکه تعداد block ها در راستاهای x و y یک grid را مشخص کنیم می آییم سایز ماتریس را به ۳۲ تقسیم میکنیم و برای اینکه بتوانیم همه thread های لازم را بسازیم می آییم سقف آن حاصل تقسیم را در نظر میگیریم. نکته دیگری که وجود دارد این است که بخاطر استفاده از تابع malloc باید ماتریس های مان را یک بعدی در نظر بگیریم. بنابر این ماتریس دو بعدی n*n را در یک آرایه یک بعدی با n^2 خانه ذخیره میکنیم. نحوه ذخیره سازی این ماتریس در آرایه یک بعدی از نوع row major است. در این برنامه ای که نوشتیم آمدیم به تعداد خانه های ماتریس جوابمان thread ساختیم . حال باید بدست آوردن حاصل هر خانه از این ماتریس را به یک thread واگذار کنیم. برای اینکار از ۶ شناسه ای که یک thread دارد استفاده میکنیم تا آن خانه ای که در ماتریس دارای مختصاتی برابر با مختصات thread در grid است را پیدا کنیم . این کار را در خطوط ۱۱ و ۱۲ انجام داده ایم. و در خطوط ۱۴ تا ۲۰ هر thread حاصل خانه مورد نظرش را بدست می آورد.متغیر n در این برنامه نشان دهنده سایز ماتریس است که من اینجا آن را ۶ گذاشتم تا بتوانم خروجی دو ماتریس ۶×۶ را محاسبه کنم. در خطوط ۳۷ تا ۴۰ این دو ماتریس را به شکل مشخص شده مقداردهی می کنیم. توابع cudaMemcpy برای تبادل دیتا بین حافظه اصلی سیستم و حافظه اصلی GPU استفاده می شود که به حافظه اصلی GPU نیز global memory می گویند. در خطوط ۵۵ تا ۶۴ برنامه نیز خروجی حاصل ضرب را در محیط کنسول چاپ میکنیم. خروجی این برنامه را در شکل ۷ مشاهده میکنید.



شکل ۷: خروجی برنامه نوشته شده

تمرین دوم : حداکثر اندازه ماتریس

با توجه به تحقیقاتی که انجام دادم به ۳ نتیجه رسیدم که هر ۳ آن را مجزا شرح میدهم

یک

یکی از مواردی که می تواند تعیین کننده حداکثر سایز ماتریس باشد global memory است. به طور مثال همانطور که در خروجی برنامه DeviceQuery مشخص است میزان حافظه global memory کارت گرافیک سیستم من 2 Gig می باشد. هر خانه از حافظه ماتریس ما که از جنس float است 4 Byte اندازه دارد. حال اگر اندازه global memory را بر ۴ تقسیم کنم به عدد $5 * 10^8$ می رسیم. در نتیجه حداکثر اندازه جذر این عدد می تواند سایز ماتریس ما باشد.

دو

حال اگر مشکل حافظه را در نظر نگیریم می توانیم محاسبه کنیم که کارت گرافیک سیستم من در نهایت چند thread میتواند ایجاد کند و چون در الگوریتمی که نوشتم هر خانه از ماتریس به یک thread داده می شود در نتیجه حداکثر اندازه ماتریس بدست می آید. طبق آن چیزی که در نتیجه deviceQuery آمده است هر grid میتواند در دو بعد 65535*65535 تعداد block داشته باشد . هر block هم ۱۰۲۴ thread میتواند داشته پس در نهایت $2^{16} * 2^{16} * 2^{10} = 2^{42}$ تعداد thread می توانیم داشته باشیم. پس حداکثر سایز ماتریس ما 2^{21} می تواند باشد. بنابر این این محدودیت بخاطر الگوریتمی که انتخاب کرده این به وجود می آید.

سه

دیدگاه سوم به مقادیر داده های درون ماتریس ها بستگی دارد. زمانی که سایز ماتریس بسیار بزرگ باشد ممکن است مقدار یک خانه از ماتریس نهایی به قدری زیاد شود که نتوان آن را در ۴ بایت ذخیره کرد و چون این محدودیت وابسته به داده است نمی توان حداکثری برای سایز ماتریس در نظر گرفت.

مراجع

[۱] Behrooz Parhami. Introduction to Parallel Processing Algorithms & Architectures. Plenum Series in Computer Science, Kluwer Academic Publishers, 2002.

[۲] David B. Kirk, Wen-Mei W. Hwu. Programming Massively Parallel Processors A Hands-on Approach. 3rd Edition, Morgan Kauffmann, 2017.