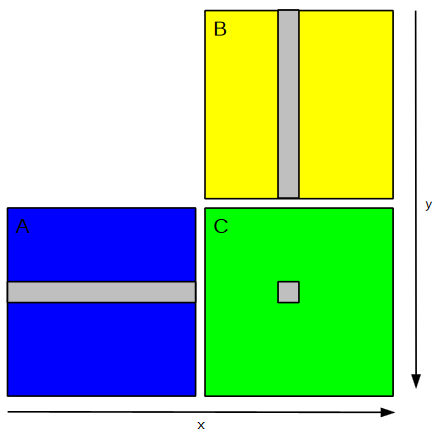
به نام خدا

برنامه‌نویسی چندهسته‌ای

دستور کار آزمایشگاه 6

پیش‌تر با موازی‌سازی عمل ضرب دو ماتریس بر روی CPU آشنا شدیم. در این آزمایش قصد داریم تا عمل ضریب دو ماتریس () را بر روی GPU پیاده‌سازی کنیم. این کار را ابتدا با یک روش ساده شروع کرده و مرحله‌به‌مرحله به جزئیات کار می‌افزاییم. برای ادامه کار می‌توانید از کد ضمیمه شده استفاده کنید. این کد تمام کارهای لازم را پیاده‌سازی کرده است و شما می‌توانید بر پیاده‌سازی kernel متمرکز شوید. برای سادگی ماتریس‌ها مربعی و با ابعاد فرض شده‌اند.

**گام اول**



شکل 1 توزیع کار در گام اول

در گام نخست، همان‌گونه که در شکل 1 مشاهده می‌کنید، هر نخ در block مسئول محاسبه‌ی یک خانه در ماتریس C (نتیجه) است. در اینجا ما بر روی کد محتوای kernel تمرکز می‌کنیم. برای این پیاده‌سازی kernel برابر است با

\_\_global\_\_ void

matrixMulCUDA(float \*C, float \*A, float \*B, int n)

{

int k;

int row = threadIdx.y, col = threadIdx.x;

float sum = 0.0f;

for (k = 0; k < n; ++k) {

sum += A[row \* n + k] \* B[k \* n + col];

}

C[row \* n + col] = sum;

}

این تابع را پیاده‌سازی کرده و سپس صحت خروجی را برای بررسی کنید (خطوط 110 تا 111 را ببینید).

**گام دوم**

همان‌گونه که مشاهده می‌کنید kernel نوشته شده در گام اول از threadIdx برای آدرس‌دهی استفاده می‌کند. این اندیس دهی ما را به داشتن تنها یک block محدود می‌کند. از طرفی یک block نمی‌تواند بیشتر از 1024 نخ داشته باشد. بنابراین برای ضرب ماتریس‌هایی با ابعاد بزرگ‌تر از دچار مشکل می‌شویم. برای حل این مساله دو راه داریم. 1) هر نخ کار بیشتری انجام دهد. 2) از چند block استفاده کنیم (شکل 2).

راه حل اول)

#define TILE\_WIDTH 16

\_\_global\_\_ void

matrixMulCUDA(float \*C, float \*A, float \*B, int n)

{

int start\_row = threadIdx.y \* TILE\_WIDTH;

int end\_row = start\_row + TILE\_WIDTH;

int start\_col = threadIdx.x \* TILE\_WIDTH;

int end\_col = start\_col + TILE\_WIDTH;

for (int row = start\_row; row < end\_row; row++) {

for (int col = start\_col; col < end\_col; col++) {

float C\_val = 0;

for (int k = 0; k < n; ++k) {

float A\_elem = A[row \* n + k];

float B\_elem = B[k \* n + col];

C\_val += A\_elem \* B\_elem;

}

C[row\*n + col] = C\_val;

}

}

}

راه حل دوم)

\_\_global\_\_ void

matrixMulCUDA(float \*C, float \*A, float \*B, int n)

{

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

float C\_val = 0;

for (int k = 0; k < n; ++k) {

float A\_elem = A[row \* n + k];

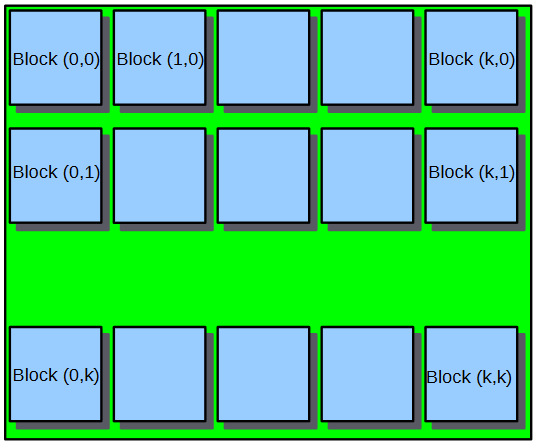
float B\_elem = B[k \* n + col];

C\_val += A\_elem \* B\_elem;

}

C[row\*n + col] = C\_val;

}



شکل 2

در راه حل اول ما عملاً تنها از یک SM استفاده می‌کنیم و در راه حل دوم به تعداد blockهای مجاز در GPU محدود می‌شویم. یک راه حل به نام tiling وجود دارد که از ترکیب دو راه حل بالا به دست می‌آید (شکل 3).

#define TILE\_WIDTH 16

\_\_global\_\_ void

matrixMulCUDA(float \*C, float \*A, float \*B, int n)

{

int start\_row = blockDim.y \* blockIdx.y + threadIdx.y \* TILE\_WIDTH;

int end\_row = start\_row + TILE\_WIDTH;

int start\_col = blockDim.x \* blockIdx.x + threadIdx.x \* TILE\_WIDTH;

int end\_col = start\_col + TILE\_WIDTH;

for (int row = start\_row; row < end\_row; row++) {

for (int col = start\_col; col < end\_col; col++) {

float C\_val = 0;

for (int k = 0; k < n; ++k) {

float A\_elem = A[row \* n + k];

float B\_elem = B[k \* n + col];

C\_val += A\_elem \* B\_elem;

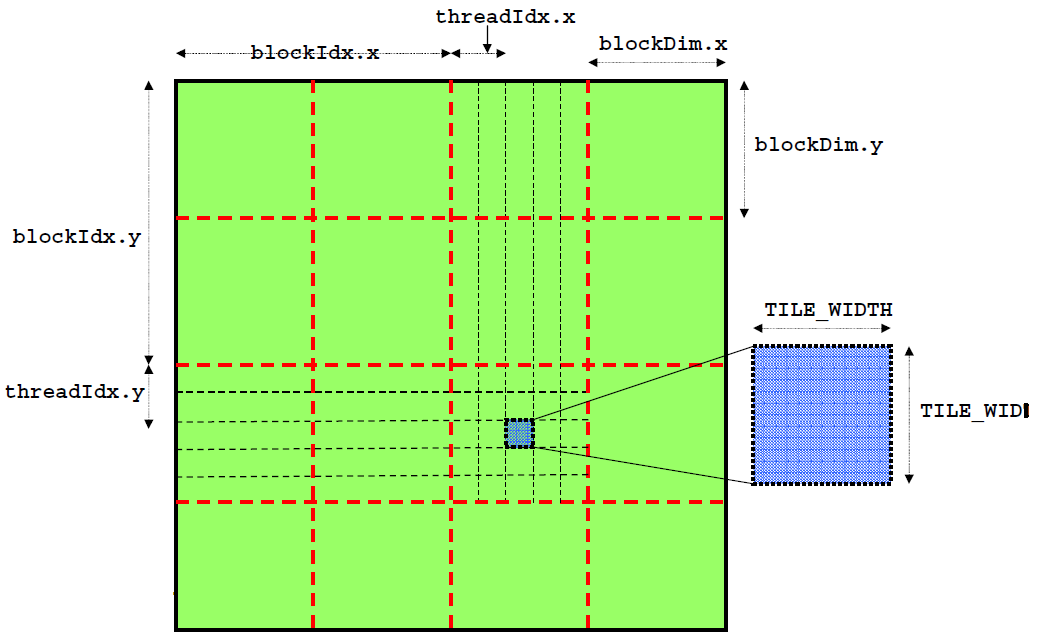
}

C[row\*n + col] = C\_val;

}

}

}



شکل 3

سه راه حل فوق را پیاده‌سازی کرده و برای مقدار به اندازه‌ی کافی بزرگ زمان‌ها را با یکدیگر مقایسه کنید. بدیهی است که ممکن است بسته به مقدار نیازمند تغییر پارامترهایی مانند TILE\_WIDTH باشید.

**گام سوم**

در این گام می‌بایست به کمک shared memory راه حل tiling را بهبود ببخشید.