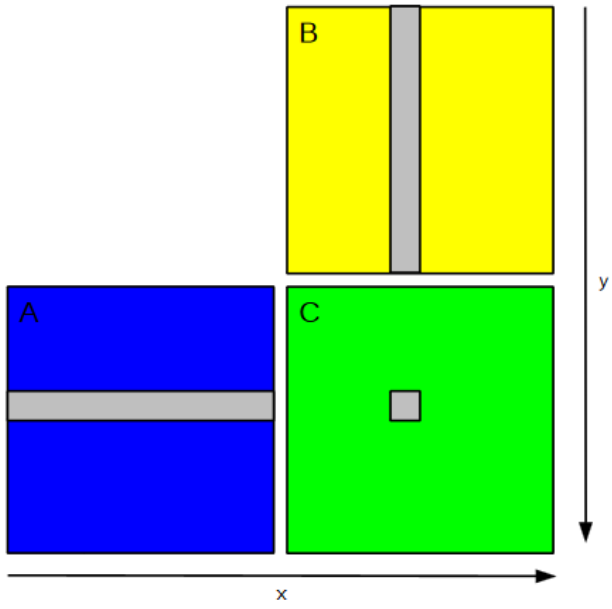
به نام خدا

برنامه‌نویسی چندهسته‌ای

دستور کار آزمایشگاه ۶

پیش‌تر با موازی‌سازی عمل ضرب دو ماتریس بر روی CPU آشنا شدیم. در این آزمایش قصد داریم تا عمل ضریب دو ماتریس $(C = A \times B)$ 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 $n \times n$ فرض شده‌اند.

گام اول



شکل ۱ توزیع کار در گام اول

در گام نخست، همان‌گونه که در شکل ۱ 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 $n = 32$ بررسی کنید (خطوط ۱۱۰ تا ۱۱۱ را ببینید).

گام دوم

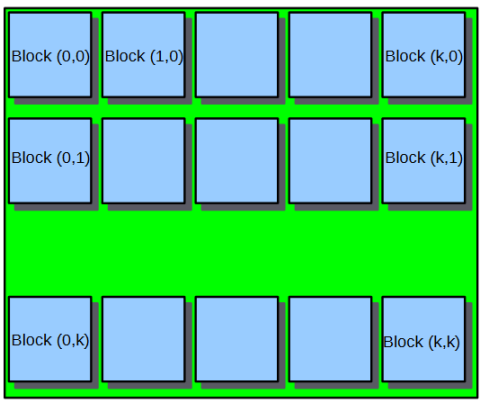
همان‌گونه که مشاهده می‌کنید kernel نوشته شده در گام اول از threadIdx برای آدرس‌دهی استفاده می‌کند. این اندیس دهی ما را به داشتن تنها یک block محدود می‌کند. از طرفی یک block نمی‌تواند بیشتر از ۱۰۲۴ نخ داشته باشد. بنابراین برای ضرب ماتریس‌هایی با ابعاد بزرگ‌تر از 32×32 دچار مشکل می‌شویم. برای حل این مساله دو راه داریم. ۱) هر نخ کار بیشتری انجام دهد. ۲) از چند block استفاده کنیم (شکل ۲).

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)

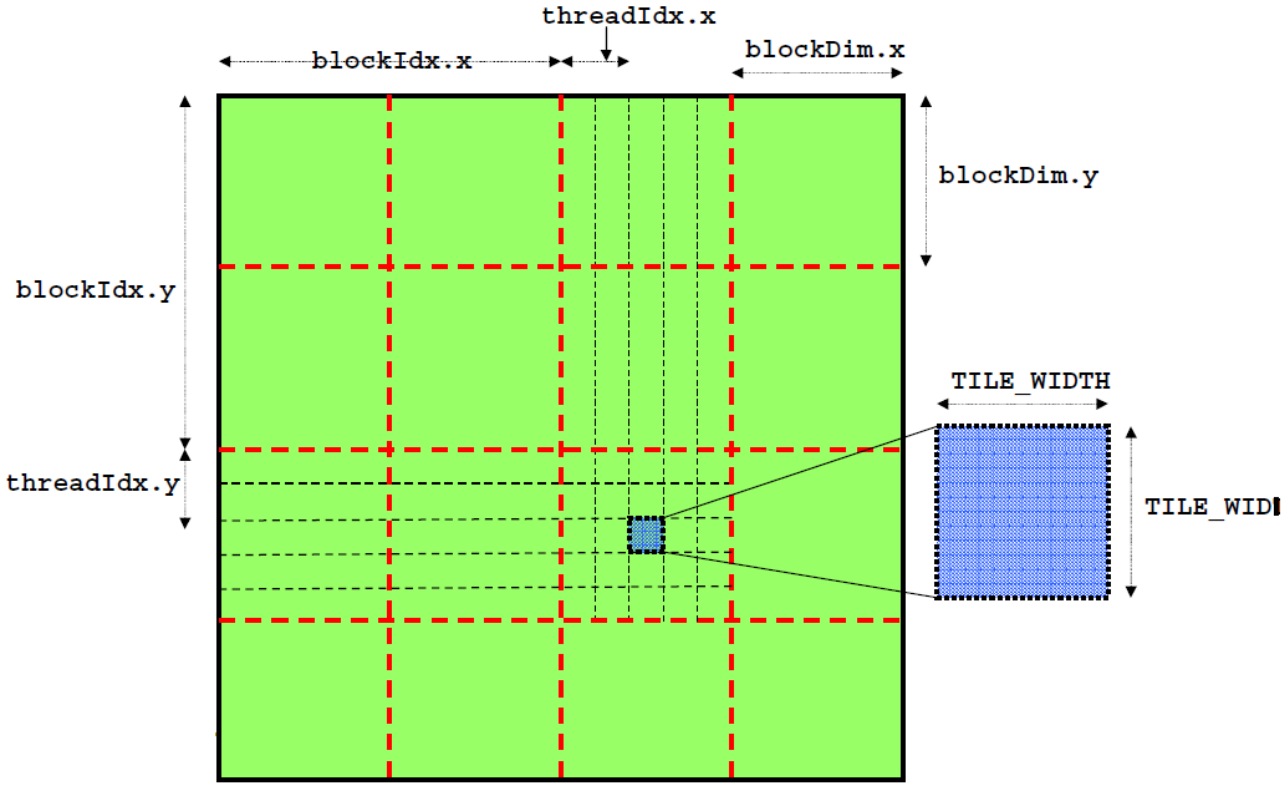
```
__global__ void
matrixMulCUDA(float *C, float *A, float *B, int n)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;
	int col = blockIdx.x * blockDim.x + threadIdx.x;
	float C_val = 0;
	for (int k = 0; k < n; ++k) {
		float A_elem = A[row * n + k];
		float B_elem = B[k * n + col];
		C_val += A_elem * B_elem;
	}
	C[row*n + col] = C_val;
}
```



شکل ۲

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 ۳).

```
#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 n زمان‌ها را با یکدیگر مقایسه کنید. بدیهی است که ممکن است بسته به مقدار n نیازمند تغییر پارامترهایی مانند TILE_WIDTH باشید.

گام سوم

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