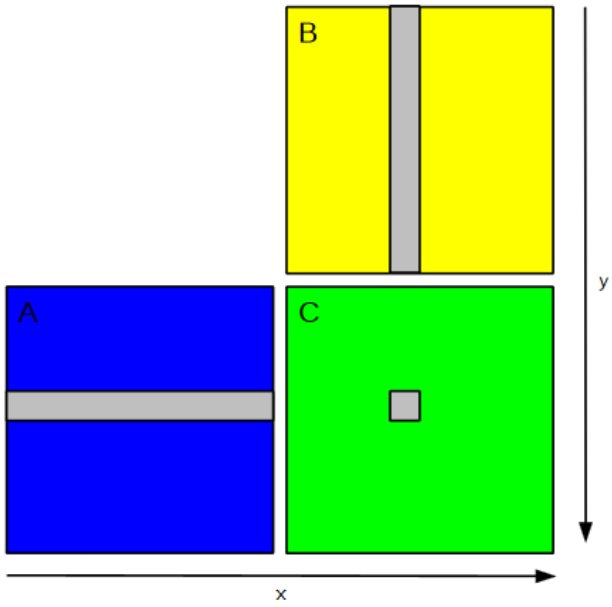
به نام خدا

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 ضمیمه‌شده 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;
}
```

همان‌گونه که مشاهده می‌کنید kernel نوشته‌شده 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.

این تابع را پیاده‌سازی کرده و سپس صحت خروجی را برای $n = 32$ بررسی کنید.

گام دوم

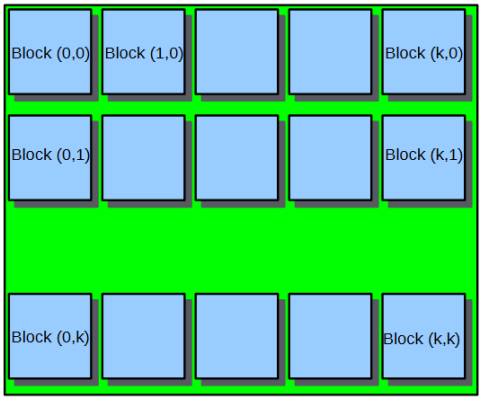
برای ضرب ماتریس‌هایی با ابعاد بزرگ‌تر از 32×32 دو راه داریم. ۱) هر نخ کار بیشتری انجام دهد. ۲) از چند block استفاده کنیم (شکل ۲).

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)

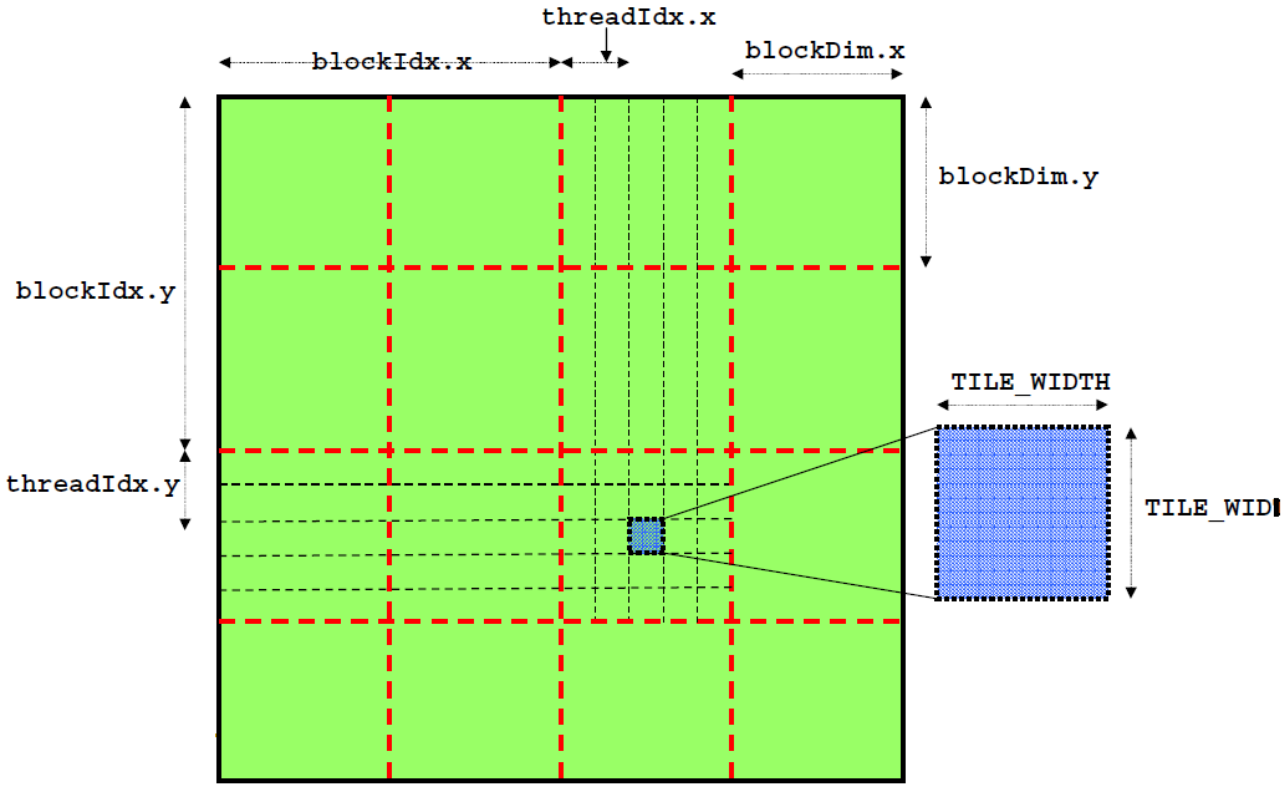
```
__global__ void
matrixMulCUDA(float *C, float *A, float *B, int n)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;
	int col = blockIdx.x * blockDim.x + threadIdx.x;
	float C_val = 0;
	for (int k = 0; k < n; ++k) {
		float A_elem = A[row * n + k];
		float B_elem = B[k * n + col];
		C_val += A_elem * 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). توجه کنید که این روش tiling با آنچه در بخش shared memory بحث شد متفاوت است. در این روش هر tile بخشی از داده است که محاسبات مربوط به آن بر عهده یک نخ است (هر نخ چند درایه ماتریس خروجی را محاسبه می‌کند). اما tile مربوط به بخش shared memory ، بخشی از داده است که به‌صورت یکجا بر روی shared memory قرار می‌گیرد و همه نخ‌های بلوک روی آن کار می‌کنند (هر نخ یک درایه ماتریس خروجی را محاسبه می‌کند).

```
#define TILE_WIDTH 16
__global__ void
matrixMulCUDA(float *C, float *A, float *B, int n)
{
	int start_row = blockDim.y * blockIdx.y * TILE_WIDTH + threadIdx.y * TILE_WIDTH;
	int end_row = start_row + TILE_WIDTH;
	int start_col = blockDim.x * blockIdx.x * TILE_WIDTH + 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;
		}
	}
}
```



شکل3

سه راه حل فوق را پیاده‌سازی کرده و صحت خروجی را بررسی کنید. سپس 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).

جدول زیر را نیز برای راه حل دوم تکمیل کنید (برای تسریع زمان اجرای سریال را در نظر بگیرید و برای محاسبه Occupancy از visual profiler یا nsight استفاده کنید).

	Block size	Grid size	Block size	Grid size	Block size	Grid size
	4x4	64x64x1	8x8	32x32x1	32x32	128x128x1
Elapsed time						
Speed up						
Occupancy						

گام سوم

در بستر و میانای برنامه‌ی کاربردی[۱] CUDA کتابخانه‌های بسیاری برای تسهیل کار برنامه‌ی نویس‌ها و تسریع سرعت توسعه و پیاده‌سازی برنامه وجود دارد. در این کتابخانه‌ها اکثرا توابع و اعمال پرکاربرد علمی پیاده‌سازی شده است. برخی از این کتابخانه‌ها عبارتند از:

- cuBLAS: در این کتابخانه پیاده‌سازی کتابخانه‌ی BLAS برای دسترسی به قابلیت‌های پردازشی GPUهای NVIDIA انجام‌شده است. با استفاده از این کتابخانه می‌توان عملیات FMA روی ماتریس‌ها انجام داد
- cuSPARSE: شامل برخی اعمال جبری برای ماتریس‌های تنک[۲] است
- cuFFT: برای تبدیل فوریه گرفتن از داده‌ها استفاده می‌شود
- cuSOLVER: با پیاده‌سازی بر اساس cuBLAS و cuSPARSE برای فاکتورگیری های ماتریسی رایج و روش‌های حل مثلثی برای ماتریس‌های چگال استفاده می‌شود.
- cuRAND: برای تولید اعداد شبه‌تصادفی[۳] و تصادفی‌نما[۴] روی GPU استفاده می‌شود

در این گام شما با استفاده از کتابخانه‌ی cuBLAS عمل ضرب دو ماتریس را پیاده‌سازی کرده و میزان افزایش سرعت با پیاده‌سازی خود در گام دوم در اندازه‌های ۳۲، ۱۲۸، ۲۵۶، ۵۱۲ مقایسه کنید.

	۳۲در ۳۲	۱۲۸ در ۱۲۸	۲۵۶ در ۲۵۶	۵۱۲ در ۵۱۲
پیاده‌سازی گام دوم				
cuBLAS				
افزایش سرعت				

[۱] Application program interface (API)

[۲] Sparse

[۳] Quasi-random

[۴] Pseudo-random