علی عسگری
40032223

هدف ما در این تمرین محاسبه جمع دو ماتریس به کمک GPU است که مراحل آن را در ادامه ذکر میکنم. همکاری بین GPU و CPU ، انتظار CPU برای محاسبات GPU ، انتقال دیتا از CPU به GPU و برعکس از مواردیست که در این برنامه بررسی شده است.
ما دو فانکشن اصلی داریم که یکی از آنها وظیفه دریافت دیتا از کاربر و تثبیت دو ماتریس را برعهده دارد و فانکشن بعدی جمع دو ماتریس و نمایششان را شبیه سازی میکند.

در حالت کلی مراحل زیر باید طی بشود.

1) تخصیص حافظه به میزبان یا همان CPU
2) تخصیص حافظه روی دستگاه یا همان GPU
3) مقدار دهی اولیه حافظه روی میزبان
4) حافظه را از میزبان به روی دستگاه کپی کنید
5) اجرای کرنل روی دستگاه
6) نتیجه را از دستگاه به میزبان کپی کنید
7) آزاد سازی حافظه ها

در ابتدا فانکشن اصلی را بررسی میکنیم که به فرآیند راه اندازی یا initial کردن کرنل بر روی GPU میپردازد .

```
__global__ void matrix_sum(float A[], float B[], float C[], int m, int n) {
   /* blockDim.x = threads_per_block                            */
   /* First block gets first threads_per_block components.      */
   /* Second block gets next threads_per_block components, etc. */
   int my_ij = blockDim.x * blockIdx.x + threadIdx.x;

   /* The test shouldn't be necessary */
   if (blockIdx.x < m && threadIdx.x < n)
      C[my_ij] = A[my_ij] + B[my_ij];
}  /* matrix_sum */
```

فانکشن بعدی وظیفه دریافت ورودی از کاربر و ایجاد دو ماتریس و نمایش آنهاست.

```
 */
void Print_matrix(char title[], float A[], int m, int n) {
   int i, j;

   printf("%s\n", title);
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%.1f ", A[(i*n)+j]);
      printf("\n");
   }
}  /* Print_matrix */
```

در ادامه بحث تخصیص مموری به CPU و ایجاد متغیر را شاهد هستیم

```
/* declare pointers to vectors in device memory and allocate memory */
h_A = (float*) malloc(size);
h_B = (float*) malloc(size);
h_C = (float*) malloc(size);
```

تخصیص مموری به GPU هم در تصویر زیر مشهود است

```
/* Allocate matrices in device memory */
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);
cudaMalloc(&d_C, size);
```

```
/* Invoke kernel using m thread blocks, each of    */
/* which contains n threads                        */
dim3 block_size( 16, 16 );
dim3 num_blocks( ( n - 1 + block_size.x ) / block_size.x,
                 ( m - 1 + block_size.y ) / block_size.y );

matrix_sum<<<block_size, num_blocks>>>(d_A, d_B, d_C, m, n);

/* Wait for the kernel to complete */
cudaThreadSynchronize();
```

در این بخش از کد ، ما دو بحث را هندل کرده ایم ، بحث اول سایز و ابعاد بلاک را تعریف کرده ایم وتعداد بلاک ها را نیز به صورت داینامیک محاسبه میکنیم.

در ادامه نیز کرنل GPU را به کمک matrix_sum فراخوانی می کنیم.

بحث cudaThreadSynchronize هم مسئله انتظار CPU برای به اتمام رسیدن محاسبات GPU را هندل میکند.

مسئله بعدی که بسیار مهم است بحث نقل و انتقال دیتا از هات به دیوایس و برعکس است که به عنوان مثال با مشاهده تصویر زیر ، طریقه انجام این فرآیند قابل مشاهده است.

```
/* Copy matrices from host memory to device memory */
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
/* Copy result from device memory to host memory */
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

بحث آزاد سازی منابع هم به شکل زیر انجام شده است

```
/* Free device memory */
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

/* Free host memory */
free(h_A);
free(h_B);
free(h_C);

return 0;
} /* main */
```

خروجی و اجرای برنامه هم به شکل زیر است. با دریافت دو ماتریس 2*2 شبیه سازی جمع را انجام داده ایم.