

به نام خدا

## برنامه‌نویسی چندهسته‌ای

### دستور کار آزمایشگاه ۵



## گام ۱

در این آزمایش قصد داریم تا با برنامه‌نویسی پردازنده‌های گرافیکی آشنا شویم. لازمه‌ی اینکه یک برنامه بتواند بر روی پردازنده‌ی گرافیکی اجرا شود، وجود پردازنده‌ی گرافیکی در سیستم است. CUDA این امکان را فراهم می‌کند تا GPUهای موجود در سیستم را پرس‌وجو کنیم و سپس برنامه خود را بر روی یک یا چند GPU اجرا کنیم. همچنین از طریق APIهایی که CUDA در اختیار قرار می‌دهد می‌توان توانایی‌های GPUهای موجود در سیستم را به دست آورد. این امر کمک می‌کند تا در صورت لزوم، پیش از اجرای محاسبات بر روی یک GPU، پارامترهای اجرا را به شکلی مناسب محاسبه کرد.

کد به دست آوردن لیست deviceهای موجود بر روی یک سیستم، با نام `deviceQuery.cu`، برای ویندوز و لینوکس در آدرس‌های زیر قرار دارد (برای لینوکس نیاز به نصب کدهای نمونه است):

`C:/ProgramData/NVIDIA Corporation/CUDA Samples`

`/usr/local/cuda-10.1/samples`

برای `compile` کردن این کد نیاز به ساختن یک Project در Visual Studio دارید. از آدرس ذکر شده، پروژه متناسب با ورژن Visual Studio خود را انتخاب کنید (به‌عنوان مثال `deviceQuery_vs2017.sln`) و کد ذکر شده را در آن `compile` کنید (`Ctrl + B`). خروجی در آدرس زیر قرار می‌گیرد:

`C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.1\bin\win64\[release|debug]`

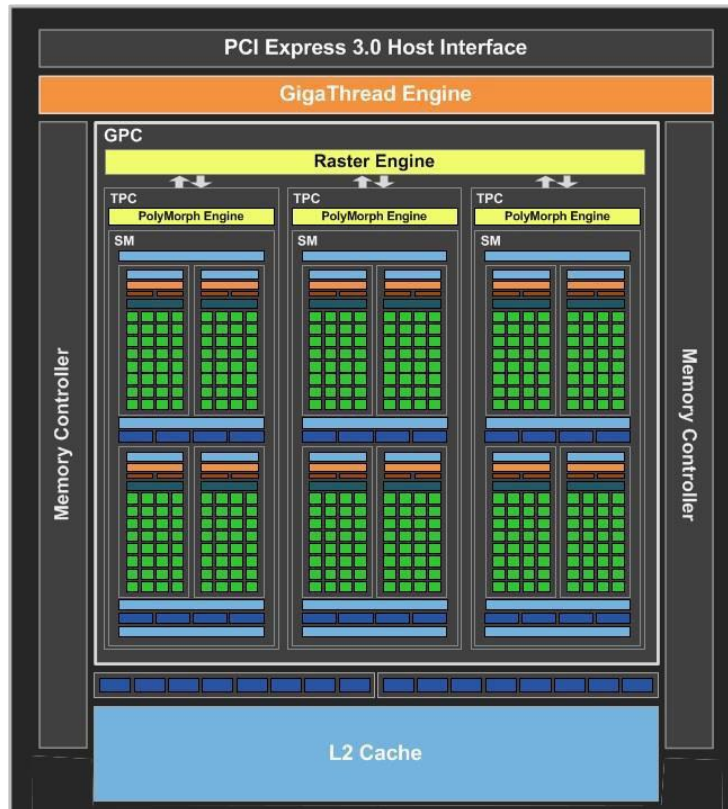
در صورتی که سیستم شما مجهز به پردازنده گرافیکی NVIDIA نیست می‌توانید از ماشین مجازی برای آزمودن برنامه‌های نیازمند به GPU استفاده کنید. برای تقاضای اطلاعات اتصال به استاد درس یا دستیار تدریس مراجعه کنید. توجه فرمایید که مسئولیت منابعی که دسترسی آن‌ها به شما داده شده است بر عهده شما است، بنابراین از انجام کارهایی غیر از اجرای برنامه‌ها خودداری کنید.

اگر همه مراحل را صحیح انجام داده باشید، باید بتوانید خروجی شکل ۱ را مشاهده کنید.

```
There is 1 device supporting CUDA
Device 0: "GeForce GT 1030"
Major revision number:          6
Minor revision number:          1
Total amount of global memory:  2147483648 bytes
Number of multiprocessors:       3
Number of cores:                 24
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size:                       32
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 2147483647 x 65535 x 65535
Maximum memory pitch:            2147483647 bytes
Texture alignment:                512 bytes
Clock rate:                      1.51 GHz
Concurrent copy and execution:    Yes
TEST PASSED
```

شکل ۱ خروجی موفق کد `queryDevice.cu`

هر ماشین مجازی مجهز به یک device با نام `NVIDIA GeForce GT1030` است که مشخصات آن در شکل ۱ قابل مشاهده است. شکل ۲ layout مربوط به GPUی این کارت گرافیک را نشان می‌دهد. همچنین شکل ۳ اجزای درون یک SM را نشان می‌دهد. می‌توانید مشخصات به‌دست‌آمده را با تصاویر نشان داده‌شده انطباق دهید.



شکل ۲ پردازنده گرافیکی کارت NVIDIA GT1030 این GPU دارای ۳ عدد SM است.



شکل ۳ نمای بزرگنمایی شده از یک SM در پردازنده گرافیکی کارت NVIDIA GT1030 هر SM در این GPU دارای دو Warp Scheduler است. همچنین می‌توانید تعداد هسته (رنگ سبز) را مشاهده کنید.

برای دیدن لیست کامل fieldهای structureای که تابع `cudaGetDeviceProperties` برمی‌گرداند می‌توانید به [مرجع NVIDIA](#) مراجعه کنید. همچنین ابزارهای [GPU-Z](#) و [CUDA-Z](#) می‌توانند در این زمینه مفید باشند.

## گام ۲

در این بخش می‌خواهیم تا یک برنامه ساده و سریال جمع دو ماتریس ( $C = A + B$ ) را به کمک CUDA بر روی GPU موازی‌سازی کنیم. برای شروع کد سریال با نام MatAdd.cu را مطالعه کنید. برای موازی‌سازی این برنامه باید مراحل ذیل طی شوند:

۱. یک GPU برای اجرای محاسبات در سیستم انتخاب شود.
۲. برای ماتریس‌های A و B و C در پردازنده گرافیکی فضای حافظه اختصاص یابد.
۳. ماتریس‌های A و B به حافظه‌ی پردازنده گرافیکی کپی شوند.
۴. محاسبات جمع دو ماتریس در پردازنده گرافیکی انجام شود.
۵. ماتریس C به حافظه‌ی پردازنده مرکزی (RAM) کپی شود.
۶. فضای حافظه‌ی گرفته‌شده در پردازنده گرافیکی آزاد شود.

اکنون تابع `addWithCuda` را تعریف می‌کنیم. این تابع ماتریس‌ها را گرفته و نتیجه را پس از انجام محاسبات بر روی GPU برمی‌گرداند. Signature تابع به این صورت است:

```
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int matSizeX, unsigned int matSizeY);
```

c برابر با آرایه نتیجه، a و b برابر با ماتریس‌های ورودی و size اندازه ماتریس‌ها را مشخص می‌کند. همچنین در صورت خطا، تابع خطا را برمی‌گرداند. برای استفاده از توابع و داده ساختارهای تعریف‌شده در CUDA کتابخانه‌های ذیل را به ابتدای برنامه اضافه کنید:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
```

در ابتدای تابع متغیرهای ذیل را برای نگهداری آدرس شروع ماتریس‌ها در GPU و خطاهای احتمالی تعریف می‌کنیم:

```
int *dev_a = 0;
int *dev_b = 0;
int *dev_c = 0;
cudaError_t cudaStatus;
```

نگهداری اشاره‌گر به ماتریس‌ها در GPU بدان سبب است که فضای آدرس CPU و GPU از یکدیگر جداست. به عبارت دیگر، هم CPU و هم GPU خانه‌ی حافظه‌ای با شماره‌ی x دارند. خانه‌ی شماره‌ی x در CPU از GPU به صورت مستقیم قابل دسترسی نیست و بالعکس. اکنون نوبت به انتخاب device جهت انجام محاسبات می‌رسد:

```
cudaStatus = cudaSetDevice(0);
if (cudaStatus != cudaSuccess) {
    printf("cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");
}
```

از اجرای کد `queryDevice.cu` در گام ۱ می‌دانیم که تنها یک پردازنده گرافیکی در سیستم موجود است. به همین دلیل پردازنده 0ام وارد شده است. پس از انتخاب پردازنده گرافیکی، برای سه ماتریس a، b و c بر روی آن حافظه می‌گیریم:

```
cudaStatus = cudaMalloc((void**)&dev_c, matSizeX*matSizeY * sizeof(int));
if (cudaStatus != cudaSuccess) {
    printf("cudaMalloc failed!");
}

cudaStatus = cudaMalloc((void**)&dev_a, matSizeX*matSizeY * sizeof(int));
if (cudaStatus != cudaSuccess) {
    printf("cudaMalloc failed!");
}

cudaStatus = cudaMalloc((void**)&dev_b, matSizeX*matSizeY * sizeof(int));
if (cudaStatus != cudaSuccess) {
    printf("cudaMalloc failed!");
}
```

در هر مرحله بررسی می‌کنیم که خطایی رخ نداده باشد. اکنون دو ماتریس  $a$  و  $b$  را به فضای اختصاص داده‌شده در GPU کپی می‌کنیم:

```
cudaStatus = cudaMemcpy(dev_a, a, matSizeX*matSizeY * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
}

cudaStatus = cudaMemcpy(dev_b, b, matSizeX*matSizeY * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
}
```

قبل از ادامه دستور کار، پارامترهای ورودی به هر API از CUDA تا اینجا را به‌دقت مطالعه کنید. به مرحله‌ای رسیده‌ایم که برای هر سه ماتریس  $a$ ،  $b$  و  $c$  در GPU فضا گرفته‌شده است و داده‌های دو ماتریس  $a$  و  $b$  نیز به GPU کپی شده‌اند. اگرچه دیگر نیازی به دو ماتریس  $a$  و  $b$  بر روی حافظه CPU (RAM) نداریم اما از حذف آن‌ها صرف‌نظر می‌کنیم. پس‌ازاین کافی است GPU دو ماتریس  $a$  و  $b$  را از حافظه‌ی خود بخواند و حاصل جمع را محاسبه کند. اکنون به بررسی موازی‌سازی تابع جمع دو ماتریس می‌پردازیم. این تابع به شکل زیر تعریف‌شده است:

```
void addMat(int * a, int *b, int *c, int matSizeX, int matSizeY) {
    int i;
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            c[i*matSizeY + j] = a[i*matSizeY + j] + b[i*matSizeY + j];
    }
}
```

اگر قرار بود این تابع را بر روی CPU موازی‌سازی کنیم، با فرض چهار هسته بودن آن،  $\frac{n}{4}$  داده‌ها را به هر نخ می‌دادیم. به عبارت دیگر، کار بین ۴ نخ تقسیم می‌شد. به بیان ساده، این بدان سبب بود که اولاً بیشتر از چهار هسته وجود نداشت و ثانیاً context switch بین نخ‌ها سربار نسبتاً زیادی را به سیستم تحمیل می‌کرد؛ ما در GPU شرایط متفاوتی داریم. یکی از این شرایط وجود Register File بسیار بزرگ در هر SM در GPU است که سربار context switch را تقریباً صفر می‌کند. به‌هرحال در اینجا تصمیم می‌گیریم که این آرایه 1024 عنصری را بین 1024 نخ توزیع کنیم؛ بنابراین تابع جدید را به این شکل تعریف می‌کنیم:

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    c[i*blockDim.y+j] = a[i*blockDim.y + j] + b[i*blockDim.y + j];
}
```

اسم تابع را `addKernel` انتخاب می‌کنیم. Kernel به تابعی گفته می‌شود که در GPU اجرا می‌شود. هر نخ به کمک `threadIdx.x` و `threadIdx.y` اندیس خود را به دست آورده و در خط بعدی داده‌های مربوط به خود را از حافظه خوانده و در عنصر متناظر در آرایه  $c$  که در GPU قرار دارد می‌نویسد. برای آنکه مشخص کنیم تابع `addKernel` از CPU فراخوانی می‌شود و در GPU اجرا می‌شود از `__global__` پیش از تعریف آن استفاده‌شده است. این بدان سبب است که `compile` کدهای ماشین متفاوتی برای CPU و GPU تولید می‌کند. با این کار مشخص می‌کنیم که برای کدامیک از CPU و GPU کد ماشین تولید کند. دقت کنید که ورودی‌های kernel اشاره‌گرهایی به خانه‌های حافظه‌ی GPU هستند.

پس از تعریف kernel اکنون باید آن را در ادامه تابع `addWithCuda` فراخوانی کنیم. CUDA یک extension به زبان C اضافه می‌کند که امکان استفاده از دستورات جدیدی را فراهم می‌آورد. دستور اجرای kernel در GPU در CUDA به شکل ذیل است:

```
dim3 block_size = dim3(matSizeX, matSizeY, 1);
addKernel <<<1, block_size>>>(dev_c, dev_a, dev_b);
```

ابتدا نام kernel که باید روی GPU اجرا شود ذکر می‌شود. سپس پارامتر اول درون `<<<, >>>` تعداد بلوک‌ها را مشخص می‌کند. پس‌ازآن پارامتر دوم تعداد نخ‌های درون هر بلوک را مشخص می‌کند. از متغیر `block_size` که از نوع `dim3` است برای ساخت بلوک نخ دو و سه‌بعدی استفاده می‌شود. درنهایت نیز اشاره‌گرها

به ماتریس‌ها به kernel داده می‌شوند. در اینجا بررسی رخداد خطا به شکلی دیگر انجام می‌پذیرد. پس از دستور فوق می‌توان به کمک دستور ذیل رخداد خطا در شروع اجرا را بررسی کرد:

```
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    printf("addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
}
```

در اینجا توجه به یک نکته اهمیت دارد. CPU و GPU در اجرا نسبت به یکدیگر غیرهمگام هستند. یعنی، زمانی که CPU به GPU انجام یک محاسبه را می‌سپارد و پس‌از آن خط دیگری از برنامه را اجرا می‌کند، بدان معنا نیست که اجرای محاسبات روی GPU به اتمام رسیده است. بنابراین زمانی که به کمک خط بالا در حال بررسی رخداد خطا در اجرای GPU هستید، اجرای برنامه روی GPU یا هنوز شروع نشده، یا در حال اجراست و یا خاتمه یافته است. طبقاً با توجه به این مسئله علاوه بر اینکه نمی‌توان به نتیجه تابع فوق اعتماد کرد، نمی‌توان آرایه c را از روی GPU به روی CPU منتقل کرد. چراکه ممکن است محاسبات هنوز کامل نشده باشد. به کمک دستور ذیل اجرای CPU را تا اتمام پردازش روی GPU متوقف کرده و سپس مجدداً وجود خطا را بررسی می‌کنیم:

```
cudaStatus = cudaDeviceSynchronize();
if (cudaStatus != cudaSuccess) {
    printf("cudaDeviceSynchronize returned error code %d after launching addKernel!\n",
    cudaStatus);
}
```

اگر خطایی رخ نداده باشد، می‌توانیم ماتریس نتایج (dev\_c) را به حافظه‌ی اصلی CPU (c) منتقل کنیم:

```
cudaStatus = cudaMemcpy(c, dev_c, matSizeX*matSizeY * sizeof(int), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    printf("cudaMemcpy failed!");
}
```

حال نتایج را در اختیار داریم. فراموش نکنیم که فضای گرفته‌شده در حافظه‌ی GPU را آزاد کنیم و متغیر نگه‌دارنده رخ داد خطا را برگردانیم:

```
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);

return cudaStatus;
```

حال می‌توانید این تابع را به‌جای تابع جمع سریال استفاده کنید. مجدداً زمان‌گیری کنید و نتیجه را با حالت سریال مقایسه کنید.

برای اندازه‌گیری زمان ابتدا دو شیء event می‌سازیم:

```
cudaEvent_t start;
cudaEventCreate(&start);

cudaEvent_t stop;
cudaEventCreate(&stop);
```

سپس در ابتدا و انتهای بازه‌ای که قصد اندازه‌گیری زمان اجرای آن را داریم، اشیایی که قبلاً ساخته‌ایم را record می‌کنیم. توجه داشته باشید که این بازه باید شامل بخش اجرای تابع Kernel و کل بخش جابجایی داده بین Host و Device باشد.

```
cudaEventRecord(start, NULL);
cudaEventRecord(stop, NULL);
```

درنهایت تا پایان اجرای event صبر می‌کنیم و طول بازه اجرا را به دست می‌آوریم (اندازه‌گیری زمان را در حالت Release انجام دهید):

```
error = cudaEventSynchronize(stop);
float msecTotal = 0.0f;
error = cudaEventElapsedTime(&msecTotal, start, stop);
```

## گام ۳

بدیهی است مثال‌های دنیای واقعی بزرگ‌تر از اندازه‌ی داده‌ها در گام ۲ هستند. در گام ۲ عناصر آرایه را  $1024 \times n$  عنصر داشته باشیم. چه باید کرد؟ در این شرایط دو راه از راه‌های پیش رو عبارت‌اند از: ۱- هر  $n$  جمع انجام دهد. ۲-  $n$  بلوک  $1024$  تایی اجرا کنیم. این دو روش را پیاده‌سازی و زمان اجرا را برای  $n$  های به‌اندازه کافی بزرگ مقایسه کنید. برای دیدن لیست متغیرهای built-in به فایل `CUDA C Cheat Sheet - Kapeli.pdf` مراجعه کنید.

## گام ۴

در این بخش می‌خواهیم دسته‌بندی نخ‌ها در warp، block و grid را بررسی کنیم. در اینجا شما باید kernel‌ای بنویسید که نخ‌ی که آن را اجرا می‌کند، شماره warp خود، شماره block‌ای که در آن قرار دارد و اندیس سراسری خودش را محاسبه و اعلام کند. باید توجه داشته که کدی که روی GPU اجرا می‌شود با محدودیت‌هایی مواجه هست. برای نمونه، توابع کتابخانه‌ای که به هنگام برنامه‌نویسی C در اختیار دارید بر روی GPU قابل اجرا نیستند. یکی از این توابع، تابع printf است (هرچند بعدها این امکان فراهم آمد، اما در اینجا مجاز به استفاده از این تابع نیستیم). با فرض اینکه kernel را در ۲ بلوک ۶۴ نخ‌ی اجرا کرده باشیم، برنامه شما باید خروجی ذیل را تولید کند:

```
Calculated Thread: 0 - Block: 0 - Warp 0 - Thread 0
Calculated Thread: 1 - Block: 0 - Warp 0 - Thread 1
Calculated Thread: 2 - Block: 0 - Warp 0 - Thread 2
Calculated Thread: 3 - Block: 0 - Warp 0 - Thread 3
Calculated Thread: 4 - Block: 0 - Warp 0 - Thread 4
...
Calculated Thread: 30 - Block: 0 - Warp 0 - Thread 30
Calculated Thread: 31 - Block: 0 - Warp 0 - Thread 31
Calculated Thread: 32 - Block: 0 - Warp 1 - Thread 32
Calculated Thread: 33 - Block: 0 - Warp 1 - Thread 33
Calculated Thread: 34 - Block: 0 - Warp 1 - Thread 34
...
Calculated Thread: 62 - Block: 0 - Warp 1 - Thread 62
Calculated Thread: 63 - Block: 0 - Warp 1 - Thread 63
Calculated Thread: 64 - Block: 1 - Warp 0 - Thread 0
Calculated Thread: 65 - Block: 1 - Warp 0 - Thread 1
Calculated Thread: 66 - Block: 1 - Warp 0 - Thread 2
Calculated Thread: 67 - Block: 1 - Warp 0 - Thread 3
...
Calculated Thread: 94 - Block: 1 - Warp 0 - Thread 30
Calculated Thread: 95 - Block: 1 - Warp 0 - Thread 31
Calculated Thread: 96 - Block: 1 - Warp 1 - Thread 32
Calculated Thread: 97 - Block: 1 - Warp 1 - Thread 33
Calculated Thread: 98 - Block: 1 - Warp 1 - Thread 34
Calculated Thread: 99 - Block: 1 - Warp 1 - Thread 35
Calculated Thread: 100 - Block: 1 - Warp 1 - Thread 36
...
Calculated Thread: 126 - Block: 1 - Warp 1 - Thread 62
Calculated Thread: 127 - Block: 1 - Warp 1 - Thread 63
```

هر خط توسط یک نخ محاسبه شده. Calculated Thread اندیس سراسری نخ، Block شماره بلوک آن نخ، Warp شماره warp آن نخ و Thread اندیس محلی نخ است.