به نام خدا

برنامه‌نویسی چندهسته‌ای

دستور کار آزمایشگاه 5

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گام 1

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 (به عنوان 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. فایل اتصال به این سرورها در پوشه‌ی GPU VM Connection قرار دارد. به هر گروه یک ماشین مجازی و نام کاربری و کلمه عبور اتصال اختصاص داده می‌شود که اعضای گروه می‌توانند برای انجام آزمایش‌ها و همچنین تمرینات درس از آن استفاده کنند. 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.

پس از اتصال به ماشین مجازی، درایو C از رایانه شما به‌صورت خودکار به ماشین مجازی متصل خواهد شد. بنابراین می‌توانید برنامه خود را در ماشین مجازی به‌راحتی اجرا کنید. اگر همه مراحل را صحیح انجام داده باشید، باید بتوانید خروجی شکل 1را مشاهده کنید.

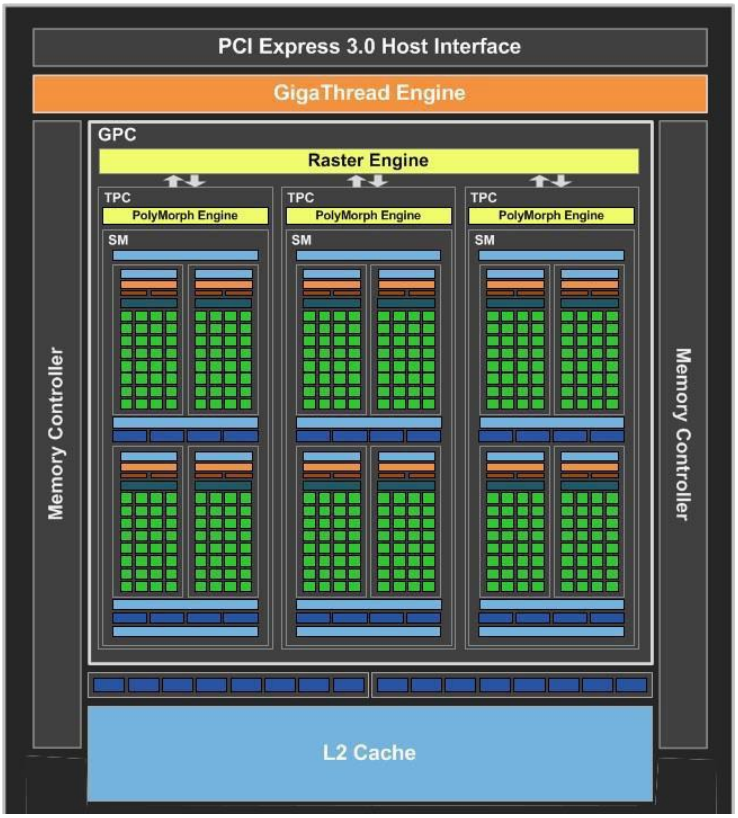
```
There is 1 device supporting CUDA

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 1 خروجی موفق کد queryDevice.cu

هر ماشین مجازی مجهز به یک device با نام NVIDIA GeForce GT1030 است که مشخصات آن در شکل 1 قابل مشاهده است. شکل 2 layout مربوط به GPUی این کارت گرافیک را نشان می‌دهد. همچنین شکل 3 اجزای درون یک SM را نشان می‌دهد. می‌توانید مشخصات به دست آمده را با تصاویر نشان داده شده انطباق دهید.



شکل 2 پردازنده گرافیکیِ کارت NVIDIA GT1030. این GPU دارای 3 عدد SM است.

شکل 3 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 2

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

1. یک GPU برای اجرای محاسبات در سیستم انتخاب شود.
2. برای بردارهای A و B و C در پردازنده گرافیکی فضای حافظه اختصاص یابد.
3. بردارهای A و B به حافظه‌ی پردازنده گرافیکی کپی شوند.
4. محاسبات جمع دو بردار در پردازنده گرافیکی انجام شود.
5. بردار C به حافظه‌ی پردازنده مرکزی (RAM) کپی شود.
6. فضای حافظه‌ی گرفته شده در پردازنده گرافیکی آزاد شود.

اکنون تابع addWithCuda را تعریف می‌کنیم. این تابع بردارها 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 size);
```

c برابر با آرایه نتیجه، a و b برابر با بردارهای ورودی و size اندازه بردارها را مشخص می‌کند. همچنین در صورت خطا، تابع خطا را برمی‌گرداند. برای استفاده از توابع و داده ساختارهای تعریف شده 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 بردارها 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 بردارها در GPU بدان سبب است که فضای آدرس CPU و GPU از یکدیگر جداست. به عبارت دیگر، هم GPU و هم CPU خانه‌ی حافظه‌ای با شماره‌ی x دارند. خانه‌ی شماره‌ی x در CPU از GPU به‌صورت مستقیم قابل دسترس 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 1 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 بردار a، b و c بر روی آن حافظه می‌گیریم:

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

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

cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
```

```
	printf("cudaMalloc failed!");
}
```

در هر مرحله بررسی می‌کنیم که خطایی رخ نداده باشد. اکنون دو بردار a و b را به فضای اختصاص داده شده در GPU کپی می‌کنیم:

```
cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
	printf("cudaMemcpy failed!");
}

cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
	printf("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 بردار a، b و c در GPU فضا گرفته شده است و داده‌های دو بردار a و b نیز به GPU کپی شده‌اند. اگرچه دیگر نیازی به دو بردار a و b بر روی حافظه CPU (RAM) نداریم اما از حذف آن‌ها صرف نظر می‌کنیم. پس از این کافی است GPU دو بردار a و b را از حافظه‌ی خود بخواند و حاصل جمع را محاسبه کند. اکنون به بررسی موازی‌سازی تابع جمع دو بردار می‌پردازیم. این تابع به شکل زیر تعریف شده است:

```
void addVector(int * a, int *b, int *c, size_t n) {
	int i;
	for (i = 0; i < n; i++) {
		c[i] = a[i] + b[i];
	}
}
```

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 4 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. اما 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;
	c[i] = a[i] + b[i];
}
```

اسم تابع را addKernel انتخاب می‌کنیم. Kernel به تابعی گفته می‌شود که در GPU اجرا می‌شود. هر نخ به کمک threadIdx.x 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 استفاده شده است. این بدان سبب است که compile کدهای ماشین متفاوتی برای CPU و GPU تولید می‌کند. با این کار مشخص می‌کنیم که برای کدام یک 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:

```
addKernel <<<1, 1024>>>(dev_c, dev_a, dev_b);
```

ابتدا نام kernelای که باید روی GPU اجرا شود ذکر می‌شود. سپس پارامتر اول درون <<< ,>>> تعداد بلوک‌ها را مشخص می‌کند. پس از آن پارامتر دوم تعداد نخ‌های درون هر بلوک را مشخص می‌کند. در نهایت نیز اشاره‌گرها به بردارها 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 پس از 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. طبعاً با توجه به این مساله علاوه بر اینکه نمی‌توان به نتیجه تابع فوق اعتماد کرد، نمی‌توان آرایه c را از روی GPU به روی CPU منتقل کرد. چرا که 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 بردار نتایج (dev_c) را به حافظه‌ی اصلی CPU (c) منتقل کنیم:

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

حال نتایج را در اختیار داریم. فراموش نکنیم که فضای گرفته شده 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 اندازه گیری 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 قبلا 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 بدست می‌آوریم (اندازه‌گیری زمان را در حالت Release انجام دهید):

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

گام 3

بدیهی است مثال‌های دنیای واقعی بزرگ‌تر از اندازه‌ی داده‌ها در گام 2 هستند. در گام 2 عناصر آرایه را 1024 فرض کردیم. این عدد بیشترین تعداد نخ‌هایی است که یک بلوک می‌تواند داشته باشد. حال فرض کنید $1024 \times n$ عنصر داشته باشیم. چه باید کرد؟ در این شرایط دو راه از راه‌های پیش رو عبارت‌اند از: 1- هر نخ n جمع انجام دهد. 2- n بلوک 1024تایی اجرا کنیم. این دو روش را پیاده‌سازی و زمان اجرا را برای nهای به‌اندازه کافی بزرگ مقایسه کنید. برای دیدن لیست متغیرهای built-in به فایل CUDA C Cheat Sheet - Kapeli.pdf مراجعه کنید.

گام 4

در این بخش می‌خواهیم دسته‌بندی نخ‌ها در warp، block ، grid و SMID 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، SM را که به ان اختصاص داده شده است و اندیس سراسری خودش را محاسبه و اعلام کند.برای به دست آوردن شماره SM اختصاص داده‌شده تابع مستقیمی وجود ندارد به همین دلیل باید در سطح پایین‌تری به بررسی بپردازیم. به همین جهت ما نیاز به PTX داریم. PTX یک زبان شبه اسمبلی است که کامپایلر NVCC کد CUDA را قبل از ترجمه به کد binary قابل اجرا روی GPU به آن ترجمه می‌کند. مانند C در CUDA نیز می‌توانیم در میان کد بلوک‌هایی از کد اسمبلی داشته باشیم. قسمت زیر کد موردنیاز برای به دست آوردن شماره SM اختصاص داده‌شده آمده است:

```
long int smid;
asm volatile("mov.u32 %0, %%smid;" : "=r"(smid));
```

asm برای وجود آوردن بلوک برای دستورالعمل اسمبلی است

volatile جهت اطمینان که در هنگام تولید PTX قسمت asm به جهت بهینه‌سازی جابجا یا حذف نشود

mov.u32 دستور موردنظر است که دستور جابجایی 32 بیتی است

%0 شماره عملوندی است که بعد از : آمده است

%smid رجیستر موردنظر است که شماره sm تخصیص‌یافته به بلوک را نگهداری می‌کند

"=r" به این معنی است که مقدار %0 را در smid قرار بدهد

مثال دیگر برای دستور PTX که علاوه بر مقدار خروجی، مقدار ورودی را نیز از خارج از بلوک asm می‌گیرد

```
asm volatile("add.s32 %0, %1, %2":"=r"(i):"r"(j),"r"(k))
```

همان‌طور که می‌بینید برای دادن مقدار از بیرون بلوک اسمبلی از **"r"** استفاده‌شده است به‌عبارت‌دیگر عبارت بالا مانند

```
add.s32 i,j,k
```

می‌توان خواند

باید توجه داشته که کدی که روی GPU اجرا می‌شود با محدودیت‌هایی مواجه هست. برای نمونه، توابع کتابخانه‌ای که به هنگام برنامه‌نویسی C در اختیاردارید بر روی GPU قابل‌اجرا 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 2 بلوک 64 نخی اجرا کرده باشیم، برنامه شما باید خروجی ذیل را تولید کند:

Worker: 0-SMid: 0-Block0-Warp: 0-Thread: 0

Worker: 1-SMid: 0-Block0-Warp: 0-Thread: 1

Worker: 2-SMid: 0-Block0-Warp: 0-Thread: 2

Worker: 3-SMid: 0-Block0-Warp: 0-Thread: 3

Worker: 4-SMid: 0-Block0-Warp: 0-Thread: 4

Worker: 5-SMid: 0-Block0-Warp: 0-Thread: 5

Worker: 6-SMid: 0-Block0-Warp: 0-Thread: 6

Worker: 7-SMid: 0-Block0-Warp: 0-Thread: 7

Worker: 8-SMid: 0-Block0-Warp: 0-Thread: 8

Worker: 84-SMid: 2-Block1-Warp: 0-Thread: 20

…

Worker: 85-SMid: 2-Block1-Warp: 0-Thread: 21

Worker: 94-SMid: 2-Block1-Warp: 0-Thread: 30

Worker: 95-SMid: 2-Block1-Warp: 0-Thread: 31

Worker: 96-SMid: 2-Block1-Warp: 1-Thread: 32

Worker: 97-SMid: 2-Block1-Warp: 1-Thread: 33

Worker: 98-SMid: 2-Block1-Warp: 1-Thread: 34

Worker: 99-SMid: 2-Block1-Warp: 1-Thread: 35

هر خط توسط یک نخ محاسبه‌شده. Worker اندیس سراسری نخ،SMid شمارهSM اختصاص داده‌شده به آن نخ، Block شماره بلوک آن نخ، Warp شماره warp آن نخ و Thread اندیس محلی نخ است.