دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مهندسی کامپیوتر

برنامه‌نویسی چندهسته‌ای

تمرین پنجم: آشنایی با مفاهیم CUDA

رادین شایانفر

شماره دانشجویی: ۹۷۳۱۰۳۲

بهار ۱۴۰۰

۱. اندیس مناسب با `i=blockIdx.x*blockDim.x+threadIdx.x` به دست می‌آید. توضیح آنکه هر نخ باید ابتدا به اندازه تعداد نخ‌هایی که در بلوک‌های پیشین آن هستند (`blockIdx.x*blockDim.x`) جلو برود و سپس اندیس آن در همان بلوک تعیین کننده است (`threadIdx.x`).

۲. در این صورت نیاز به ۸ بلوک برای پوشش دادن کل آرایه داریم. در نتیجه $1024 \times 8 = 8192$ نخ در گرید خواهیم داشت.

۳. مطابق مستندات کودا[1]، compute capability یک دستگاه (که با یک شماره ورژن مشخص می‌شود) مشخص‌کننده قابلیت‌های پشتیبانی شده توسط سخت‌افزار GPU است که به کمک آن برنامه‌ها می‌توانند در زمان اجرا تشخیص دهند چه دستورالعمل‌ها و قابلیت‌هایی بر روی آن GPU فعال است و با توجه به آن مسیر اجرای برنامه را مشخص کنند.

compute capability با استفاده از دو بخش major revision number (مثلا X) و minor revision number (مثلا Y) مشخص می‌شود و به شکل X.Y نوشته می‌شود. دستگاه‌هایی که major revision number آن‌ها برابر است، معماری هسته یکسانی دارند. minor revision number نیز تنها مشخص کننده برخی بهبودهای معماری هسته (مثلا قابلیت‌های جدیدتر) است.

۴. Parallel Thread Execution (یا به اختصار PTX)، یک ماشین مجازی سطح پایین و یک مجموعه دستورات (ISA) برای زبان CUDA شرکت Nvidia است.

کدهای device در کودا می‌توانند هم به زبان C++ و هم به اسمبلی PTX نوشته شوند. کامپایلر nvcc کدهای C++ را به PTX تبدیل می‌کند و در نهایت بخش‌های <<< . . . >>> در کد host با API callهای درایور کارت گرافیک جایگزین می‌شوند.

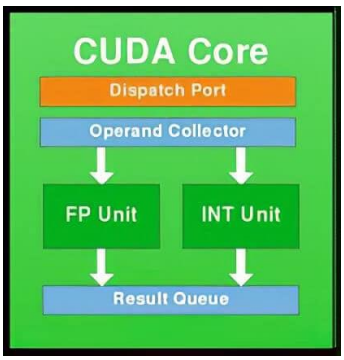
باید توجه داشت که کدهای PTX به طور مستقیم قابل اجرا بر روی کارت گرافیک نیستند. در واقع PTX در زمان اجرا توسط درایور دستگاه به کد باینری تبدیل می‌شود که به آن کامپایل just-in-time (با به اختصار JIT) می‌گویند. هر چند JIT باعث کندی لود شدن برنامه می‌شود اما اجازه بهره بردن برنامه از کامپایلرهای بهبود یافته در نسخه‌های جدیدتر درایور دستگاه را می‌دهد. همچنین JIT تنها راه اجرای برنامه‌ها روی دستگاه‌هایی است که در زمان کامپایل برنامه وجود نداشتند.

ساختار PTX در مستندات CUDA آمده است[2]. در اینجا به چند مورد اشاره می‌شود:

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability
[2] https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#syntax

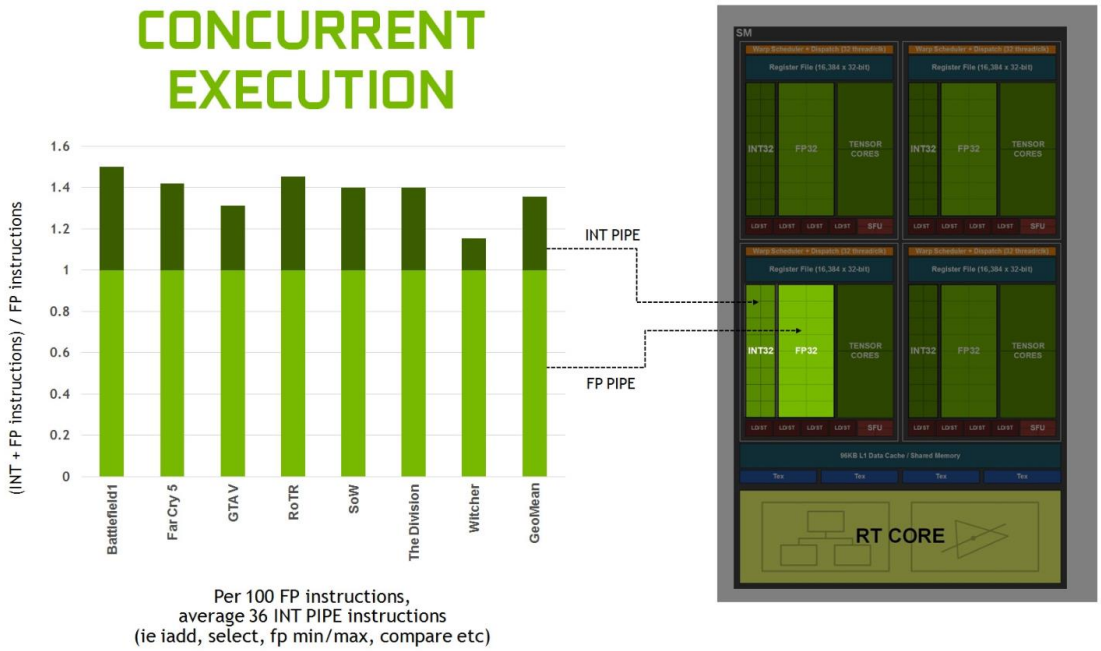
- PTX حساس به بزرگی و کوچکی حروف است و از حروف کوچک برای کلمات کلیدی استفاده می‌کند. همچنین whitespaceها در آن نادیده گرفته می‌شوند (مگر زمانی که برای جدا کردن قسمت‌های مختلف استفاده شود)
- در آن می‌توان از preprocessorهای زبان C (مانند `#include`، `#define` و ...) استفاده کرد. کامنت‌گذاری در آن نیز مانند زبان C انجام می‌شود.
- یک دستور در PTX یا یک راهنما (directive) است یا یک دستور (instruction). دستورات می‌توانند با یک لیبل اختیاری شروع شوند و با یک سمی‌کالن خاتمه می‌یابند.
- راهنماها با یک نقطه شروع می‌شوند. در نتیجه تداخلی با مواردی که کاربر تعریف می‌کند ندارد.
- دستورات در PTX با یک opcode شروع می‌شوند و صفر یا بیشتر عملوند پس از آن‌ها می‌آید. در انتها نیز با سمی‌کالن خاتمه می‌یابند. عملوندها می‌توانند متغیرهای داخل رجیسترها، عبارت‌های ثابت، آدرس یا نام یک لیبل باشند.

۵. در شکل زیر بلوک دیاگرام یک CUDA core را می‌بینیم.



در واقع هر هسته کودا دارای یک واحد ALU کوچک به همراه تعدادی بخش‌های کنترلی دیگر است. از معماری **Turing**، واحدهای عدد صحیح و ممیز شناور از هم جدا شدند. در نتیجه امکان اجرای همزمان دستورالعمل‌های صحیح و ممیز شناور در این هسته‌ها وجود دارد. در شکل زیر ساختار SMهای معماری Turing و تاثیر جدا بودن پایپ‌لاین واحدهای صحیح و ممیز شناور آن را می‌بینیم[۳].

[۳] منبع عکس: NVIDIA TURING GPU ARCHITECTURE ص. ۱۳



Profiling many workloads shows an average of 36 integer operations for every 100 floating point operations.

۶. هسته‌های تنسور هسته‌هایی هستند که می‌توانند عمل ضرب تنسورها (مثلا ماتریس را) با سرعت بالایی انجام دهند. از آنجا که ضرب تنسورها محاسبات زیادی دارد، این هسته‌ها به شکل بهینه و پرسرعت می‌توانند این کار را انجام دهند. به عنوان مثال در اولین نسل این هسته‌ها که در معماری Volta اضافه شد، امکان انجام عمل GEMM[۴] را بر روی ۶۴ ماتریس ۴ در ۴ (از نوع ممیز شناور ۱۶ بیتی) در یک کلاک را دارند. در نسل‌های اخیر هسته‌های تنسور تعداد ضرب‌ها به ۱۲۶ ضرب رسیده و همچنین نوع داده‌های جدیدتری نیز پشتیبانی می‌شوند.

استفاده از هسته‌های تنسور نیز به سادگی قابل انجام است. تنها کافی است فلگی را در استفاده از درایور مشخص کنیم (با استفاده از WMMA API) که از هسته‌های تنسور استفاده کند. در صورتی که نوع داده پشتیبانی شود و ابعاد ماتریس‌ها مضربی از ۸ باشد، بقیه کارها توسط سخت‌افزار انجام می‌شود.

در سطح اسمبلی نیز، با قرار دادن داده‌ها با فرمت مشخص در رجیسترها و اجرای دستور ضرب ماتریس، ضرب به صورت بهینه و با سرعت بالا (گاهی تا ۸ برابر سریع‌تر از هسته‌های عادی) انجام می‌شود.

۷. بله این کار ممکن است. با استفاده از سوییچ‌های زیر در nvcc امکان آن وجود دارد.

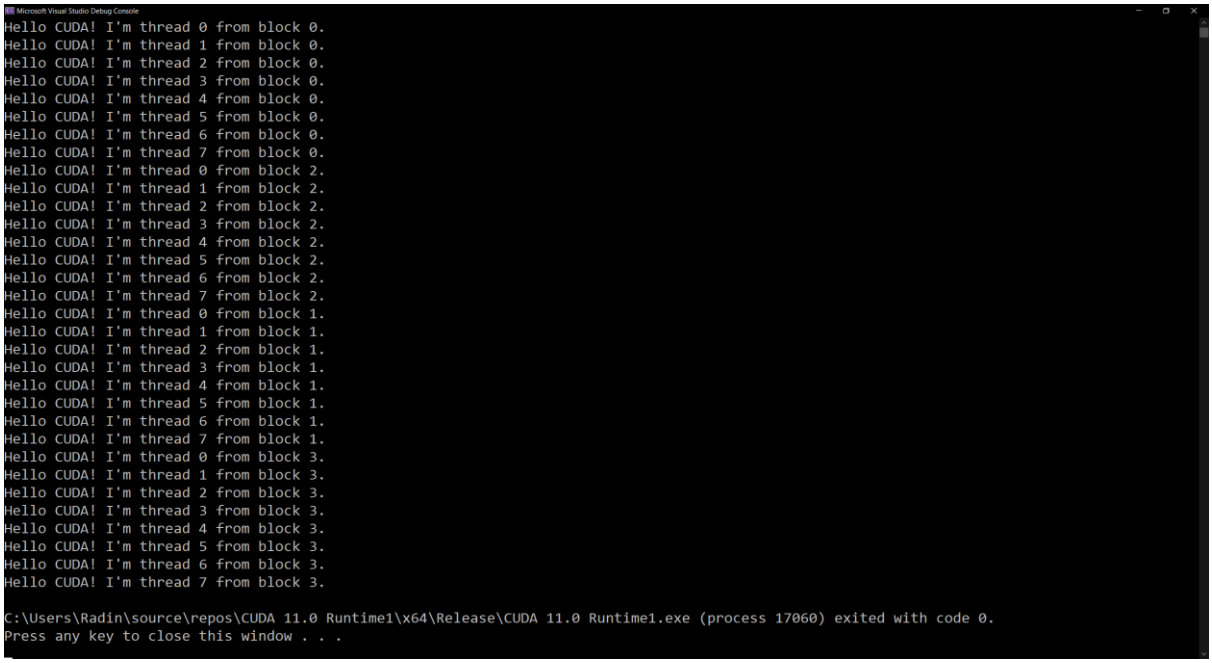
```
nvcc -Xcompiler -fopenmp -lgomp
```

[4] General Matrix Multiplication: $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$,

کاربرد این کار می‌تواند در مواردی باشد که همزمان با اینکه کارت گرافیک مشغول انجام کارهایی است که مشابه هم هستند (مثلا محاسبات ماتریسی که به خوبی با مدل SIMD کار می‌کند)، پردازنده هم بتواند برخی کارهایی که مستقل از هم هستند اما شباهتی به هم ندارند (مثلا استفاده از task در OpenMP) را به طور موازی انجام دهد تا سرعت اجرا بیشتر شود.

۸. کد برنامه در فایل Q8.cu قرار دارد.

در این برنامه در یک گرید ۴ بلوک قرار دارد و در هر بلوک ۸ نخ ساخته می‌شود. سپس هر نخ شماره `blockIdx.x` و `threadIdx.x` خود را چاپ می‌کند. یک نمونه از اجرای این کد در شکل زیر آمده است.



۹. کد برنامه در فایل Q9.cu آمده است. اندازه بردارها برابر ۱۰۰ میلیون در نظر گرفته شده و زمان‌های اجرا، تنها از میانگین‌گیری زمان ۱۰ عمل جمع (و نه پر و کپی کردن آرایه‌ها) محاسبه شده است. همچنین از GPU با Compute Capability برابر 6.1 استفاده شده است. لازم به ذکر است در انتهای تابع انجام محاسبات روی GPU، نتایج محاسبات توسط OpenMP به کمک چند نخ روی CPU بررسی می‌شوند و در صورت نادرست بودن نتایج جمع خطا داده می‌شود.

A. جمع سریال توسط تابع `serialAdd` انجام می‌شود. زمان اجرا در این حالت **۰.۱۴۶۳۱۱ ثانیه** به دست آمده است.

B. با موازی‌سازی کد سریال روی ۸ هسته به کمک OpenMP زمان اجرا به **۰.۰۸۶۱۰۹ ثانیه** کاهش می‌یابد. موازی‌سازی روی CPU در تابع `cpuAdd` آمده است.

C. تابع `gpuAdd` محاسبات را به کمک کودا بر روی GPU انجام می‌دهد. در این تابع ابتدا به هر نخ تنها یک المان برای محاسبه نسبت می‌دهیم. در نتیجه grid size برابر تعداد المان‌های بردارها (۱۰۰ میلیون) خواهد بود. با تغییر اندازه بلوک‌ها، جدول زیر را پر می‌کنیم.

جدول ۱ - زمان‌های اجرا با پردازش یک المان در هر نخ

	اندازه بلوک			
	128	256	512	1024
زمان اجرا (ثانیه)	0.008218	0.008090	0.008039	0.008113

به نظر می‌رسد اندازه بلوک بسیار کوچک (۱۲۸) و بسیار بزرگ (۱۰۲۴) چندان مناسب نیست. می‌دانیم که اندازه بلوک بسیار کوچک یا بسیار بزرگ به علت محدودیت‌های مختلف GPUها، نمی‌تواند به خوبی همه هسته‌های پردازشی آن را پر کند.

D. تابع مذکور را با اضافه کردن متغیر `ELEMENTS_PER_THREAD`، تغییر می‌دهیم. زمان‌های اجرا در جدول زیر پر می‌کنیم.

جدول ۲ - زمان‌های اجرا (ثانیه) با پردازش چند المان در هر نخ

تعداد المان پردازشی توسط هر نخ	اندازه بلوک	
	256	512
۱	0.008020	0.008072
۴	0.009032	0.008864
۸	0.012626	0.016394
۱۶	0.037284	0.056279

i. به وضوح **ریزدانگی** در این مسئله مناسب‌تر است. در واقع سریع‌تر بودن محاسبات روی GPU نسبت به CPU نیز به علت ریزدانگی بسیار بیشتر GPU نسبت به CPU است. این موضوع از آن‌جا ناشی می‌شود که در این مسئله، وظایف (taskها) شبیه به هم و ریز هستند و به خوبی با معماری SIMD در GPUها می‌توانند موازی شود.

ii. تعداد نخ‌ها و اندازه بلوک مناسب ارتباط مستقیمی با نوع مسئله و معماری GPU دارد. می‌دانیم که هر معماری GPU، محدودیت‌های مختلفی دارد. محدودیت‌هایی مانند تعداد نخ‌های هر بلوک، تعداد بلوک‌های قابل اجرای همزمان روی SMها، تعداد رجیسترهای مورد استفاده هر نخ و بلوک و اندازه shared memory قابل استفاده هر بلوک از عواملی هستند که می‌توانند موجب کاهش occupancy و کارایی شوند. گلوگاه شدن یا نشدن این محدودیت‌ها ارتباط مستقیمی با تعداد نخ‌ها و اندازه بلوک دارد.

۱۰.

A. کد کودای نوشته شده، پس از کامپایل توسط nvcc به PTX تبدیل می‌شود. PTX یک زبان شبه اسمبلی است که شباهت زیادی به زبان ماشین دارد، اما هنوز قابل اجرا روی دستگاه نیست. در زمان لود کردن برنامه و پیش از اجرای آن، درایور با کامپایل PTX و تبدیل آن به SASS، آن را به کد باینری قابل اجرا روی دستگاه تبدیل می‌کند.

B. در زبان C تابع `malloc` یک اشاره‌گر به حافظه تخصیص داده شده برمی‌گرداند. در `cudaMalloc` اما از آنجا که خروجی آن برای هدف دیگری (status اجرای تابع) استفاده می‌شود، لازم است تا خروجی ذکر شده (اشاره‌گر به حافظه اختصاص داده شده) در یکی از آرگومان‌های تابع ریخته شود. به همین دلیل و برای تغییر مقدار اشاره‌گر در scope تابعی که `cudaMalloc` را صدا زده است، نیاز به اشاره‌گر دوگانه داریم.