به نام خدا

تهیه کننده: ابراهیم صدیقی شماره دانشجویی: ۹۹۳۱۰۹۸

:Q1

(a

PCIe (Peripheral Component Interconnect Express) یک استاندارد گذرگاه بسط کامپیوتر سریال پرسرعت است که با نسخه قبلی خود PCI که از گذرگاه موازی استفاده می کند متفاوت است. PCIe انتقال داده سریعتر و پهنای باند بالاتر را با نقطه به نقطه فراهم می کند. سوئیچ PCIe چندین دستگاه را به یک پورت PCIe متصل می‌کند، جریان داده را مدیریت می‌کند و به دستگاه‌ها اجازه می‌دهد مستقیماً بدون CPU ارتباط برقرار کنند.

(b

چهار معماری سیستم در این کتاب احتمالاً پیکربندی‌های مختلف GPU ،RAM و تجهیزات جانبی را مورد بحث قرار می‌دهند. سیستم‌های درجه یک مصرف‌کننده مدرن اغلب از معماری‌هایی استفاده می‌کنند که برای پهنای باند بالا و تأخیر کم، با اتصالات مستقیم CPU-GPU از طریق فناوری‌هایی مانند PCIe و NVMe برای انتقال سریع داده‌ها، بهینه می‌شوند.

(c

فناوری GPU-Direct به پردازنده‌های گرافیکی اجازه می‌دهد تا مستقیماً داده‌ها را به/از ذخیره‌سازی یا دیگر GPUها انتقال دهند و CPU را دور بزنند تا تأخیر را کاهش داده و پهنای باند را افزایش دهند. در سیستم های درجه مصرف کننده، این بدان معناست که GPU ها می توانند مستقیماً به حافظه سیستم دسترسی داشته باشند یا با سایر GPU های بدون CPU ارتباط برقرار کنند.

(d

پردازنده‌های گرافیکی انویدیا چندین نوع حافظه دارند:

• حافظه جهانی (Global Memory): بزرگ، آهسته و قابل دسترسی برای همه رشته ها. برای ساختارهای داده بزرگ استفاده می شود.

• حافظه مشترک (Shared Memory): روی تراشه، سریعتر از حافظه جهانی، به اشتراک گذاشته شده در بین رشته های یک بلوک. برای به اشتراک گذاری داده ها در یک بلوک استفاده می شود.

• حافظه محلی (Local Memory): خصوصی برای هر رشته، ذخیره شده در حافظه جهانی. برای متغیرهای محلی استفاده می شود.

• حافظه ثابت (Constant Memory): ذخیره شده، فقط خواندنی، قابل دسترسی برای همه رشته ها. برای ثابت ها استفاده می شود.

• حافظه بافت (Texture Memory): ذخیره شده، فقط خواندنی، طراحی شده برای واکشی بافت. در گرافیک و محاسبات محلی سازی شده استفاده می شود.

e) سلسله مراتب CUDA:

• Thread: کوچکترین واحد اجرا، یک نمونه از یک هسته را اجرا می کند.

• Warp: گروهی از 32 رشته که دستورات را در مرحله قفل اجرا می کنند.

• Thread Block: گروهی از رشته ها که می توانند با یکدیگر همکاری کرده و منابع را به اشتراک بگذارند.

• SM (Streaming Multiprocessor): یک یا چند بلوک را اجرا می کند و منابع محاسباتی را فراهم می کند.

• Grid: مجموعه ای از بلوک ها که یک هسته را در سراسر GPU اجرا می کنند.

هر سطح امکان اجرای موازی در دانه بندی های مختلف را فراهم می کند که به مقیاس پذیری برنامه نویسی GPU کمک می کند.

f) بانک‌های ثبات بزرگ در پردازنده‌های گرافیکی وضعیت بسیاری از رشته‌ها را ذخیره می‌کنند، که امکان تعویض سریع متن و اجرای موازی کارآمد را فراهم می‌کند. آنها متغیرهایی را که اغلب به آنها دسترسی دارند نگه می دارند و نیاز به دسترسی به انواع حافظه کندتر را کاهش می دهند.

g)برای تعیین قابلیت محاسبه CUDA، می توانید از ابزار deviceQuery موجود در جعبه ابزار CUDA استفاده کنید. برای اشکال زدایی و پروفایل، می توانید از ابزارهایی مانند Nsight Compute، Compute Sanitizer، CUDA-GDB و Nsight Systems استفاده کنید.

h) از منظر سخت افزاری، یک هسته CUDA بیشتر شبیه به یک هسته CPU است. از منظر نرم افزاری، یک رشته در CUDA شبیه به یک رشته در CPU به عنوان کوچکترین واحد اجرایی است که می تواند توسط سیستم عامل برنامه ریزی شود.

Q2

a) با توجه به اینکه هر رشته وظیفه محاسبه یک شاخص واحد را بر عهده دارد و حداکثر تعداد رشته ها در هر بلوک 1024 است، می توانید از بلوک های 1024 رشته برای حداکثر استفاده استفاده کنید. از آنجایی که شما 20000 شاخص دارید، برای پوشش همه شاخص‌ها به $$ \lceil \frac{1024} \rceil = 20\ $$ نیاز دارید.

b) پیکربندی در بخش (الف) لزوماً بهینه نیست، زیرا اگر تعداد بلوک‌ها مضربی از تعداد SM نباشد، ممکن است به طور کامل از تمام SMها استفاده نکند. برای بهینه سازی، می توانید بیش از یک ایندکس در هر رشته اختصاص دهید. به عنوان مثال، اگر هر رشته دو شاخص را پردازش کند، به نصف تعداد رشته ها و در نتیجه نصف تعداد بلوک نیاز خواهید داشت که می تواند منجر به استفاده بهتر از SM شود.

c) فرمول محاسبه شاخص آرایه برای یک شاخص در هر رشته (بخش الف) به صورت زیر خواهد بود:

```
index = blockIdx.x * blockDim.x + threadIdx.x
```

اگر هر نخ چندین شاخص را پردازش کند (بخش ب)، فرمول داخل یک حلقه ممکن است به صورت زیر باشد:

```
for(int i = 0; i < num_indices_per_thread; ++i) {
        int index = (blockIdx.x * blockDim.x + threadIdx.x) * num_indices_per_thread + i;
        // Check if index is within bounds before processing
        if(index < 20,000) {
        // Perform calculations
        }
}
```

تعداد واقعی شاخص هایی که هر رشته باید پردازش کند، num_indices_per_thread را جایگزین کنید.

d) برای هر مقدار x:
• 16 رشته / بلوک: هر SM می تواند 16 بلوک را اجرا کند، بنابراین 16 * 16 = 256 رشته در هر SM فعال خواهد بود. این 12.5٪ استفاده از ظرفیت threads/SM 2048 است.

• 20 رشته / بلوک: هر SM می تواند 16 بلوک را اجرا کند، بنابراین 16 * 20 = 320 رشته در هر SM فعال خواهد بود. این 15.625٪ استفاده است.

• 32 رشته / بلوک: هر SM می تواند 16 بلوک را اجرا کند، بنابراین 16 * 32 = 512 رشته در هر SM فعال خواهد بود. این 25 درصد استفاده است.

• 200 رشته / بلوک: هر SM می تواند 10 بلوک را اجرا کند، بنابراین 10 * 200 = 2000 رشته در هر SM فعال خواهد بود. این 97.65625٪ استفاده است.

• 256 رشته / بلوک: هر SM می تواند 8 بلوک را اجرا کند، بنابراین 8 * 256 = 2048 رشته در هر SM فعال خواهد بود. این استفاده 100٪ است.

• 1000 رشته / بلوک: هر SM می تواند 2 بلوک را اجرا کند، بنابراین 2 * 1000 = 2000 رشته در هر SM فعال خواهد بود. این 97.65625٪ استفاده است.

• 1024 رشته / بلوک: هر SM می تواند 2 بلوک را اجرا کند، بنابراین 2 * 1024 = 2048 رشته در هر SM فعال خواهد بود. این استفاده 100٪ است.

در پیکربندی‌هایی با 16، 20 و 32 رشته در هر بلوک، بلوک‌هایی در انتظار اتمام بلوک‌های دیگر برای برنامه‌ریزی هستند زیرا تعداد رشته‌های فعال کمتر از ظرفیت SM است. در پیکربندی‌هایی با 200، 256، 1000 و 1024 رشته در هر بلوک، با فرض وجود بلوک‌های کافی برای زمان‌بندی، همه رشته‌ها می‌توانند به طور همزمان بدون انتظار فعال باشند.

Q3
برای محاسبه میانگین متحرک وزنی در یک GPU، باید یک هسته CUDA بنویسیم که بتواند طول های مختلف آرایه ورودی را مدیریت کند. هسته میانگین متحرک وزنی را طبق فرمول محاسبه می کند:

$$ [output[i] = 0.14 \times input[i] + 0.29 \times input[i + 1] + 0.57 \times input[i + 2 $$

با توجه به مشخصات GPU (10 SM)، هر کدام با 128 هسته، حداکثر 1024 Threads/Block، 16 Blocks/SM، و 2048 Threads/SM)، می‌خواهیم تعداد رشته‌ها و بلاک‌ها را بهینه کنیم.

در اینجا یک هسته CUDA و منطق تعیین اندازه بلوک بهینه و تعداد بلوک وجود دارد:

```
__global__ void weightedMovingAverage(float *input, float *output, int len) {
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        if (index < len - 2) { // Ensure we don't read beyond the array
        output[index] = 0.14f * input[index] + 0.29f * input[index + 1] + 0.57f * input[index + 2];
        }
}
```

برای اندازه بلوک و تعداد بلوک، موارد زیر را در نظر می گیریم:
• هدف ما به حداکثر رساندن استفاده از GPU است.

• ما باید اطمینان حاصل کنیم که بلوک های کافی برای مشغول نگه داشتن تمام پیامک ها داریم.

• ما می خواهیم از داشتن موضوعات غیرفعال بیش از حد جلوگیری کنیم.

اندازه بلوک بهینه اغلب مضربی از اندازه تار (32) است، بنابراین ما با 1024 رشته در هر بلوک شروع می کنیم زیرا حداکثر مجاز است و مضربی از 32 است. این انتخاب اشغال هر بلوک را به حداکثر می رساند.

برای تعداد بلاک‌ها، ما می‌خواهیم حداقل به اندازه تعداد پیامک‌ها بلوک داشته باشیم تا همه آنها را مشغول نگه داریم. با این حال، ما همچنین باید تعداد کل رشته‌هایی را که می‌توانیم همزمان فعال داشته باشیم (2048 Threads/SM * 10 SMs = 20480 Threads) در نظر بگیریم.

در اینجا نحوه محاسبه تعداد بلوک ها به عنوان تابعی از اندازه آرایه ورودی آورده شده است:

```
int blockSize = 1024; // Max threads per block
int minBlocksPerSM = 16; // Min blocks per SM to keep the SMs busy
int maxThreadsPerSM = 2048; // Max threads per SM
int numSMs = 10; // Number of SMs in the GPU

int blockCount = (len + blockSize - 1) / blockSize; // Total blocks needed for the array
blockCount = max(blockCount, numSMs * minBlocksPerSM); // Ensure we have enough blocks
to keep all SMs busy

// Adjust blockCount to not exceed the total number of active threads
int maxActiveThreads = numSMs * maxThreadsPerSM;
if (blockCount * blockSize > maxActiveThreads) {
        blockCount = maxActiveThreads / blockSize;
}
```

این محاسبه تضمین می کند که:
• ما بلوک های کافی برای مشغول نگه داشتن تمام پیامک ها داریم.

• ما از حداکثر تعداد رشته های فعال در GPU تجاوز نمی کنیم.

برای اندازه‌های مختلف آرایه ورودی (256، 512، 1024، 4M ،20k)، تعداد بلاک‌ها مطابق با این تنظیم تنظیم می‌شود تا ضمن رعایت محدودیت‌های GPU، استفاده بالا حفظ شود.

این پیکربندی تقریباً برای GPU معین بهینه است، زیرا تعداد رشته‌های فعال را به حداکثر می‌رساند و تضمین می‌کند که همه SMها کاری برای انجام دارند، که باید به اشغال بالا و استفاده کارآمد از منابع GPU منجر شود. با این حال، پیکربندی بهینه واقعی ممکن است بر اساس عوامل اضافی مانند پهنای باند حافظه و ویژگی های اجرای هسته متفاوت باشد، که نیاز به پروفایل و تنظیم دقیق دارد.