علی قنبری - ۴۰۱۱۰۵۲۴

پروژه – پایگاه داده موازی

پایگاه‌های داده یکی از نرم‌افزارهای اساسی در همه برنامه‌های بزرگ هستند. یک پایگاه داده از تعدادی جدول که از تعدادی سطر که از تعدادی ویژگی تشکیل شده‌اند، تشکیل می‌شود.

مقدمه

پایگاه‌های داده، به غیر از ذخیره و بازیابی اطلاعات، برای تحلیل داده‌ها و دریافت گزارش نیز استفاده می‌شوند، مثلاً در جدول نمره‌های پایگاه داده دانشگاه صدها هزار نمره ثبت شده و مثلاً وقتی می‌خواهیم شماره دانشجویی کسی که بالاترین معدل را از ورودی ۱۴۰۰ داشته پیدا کنیم، از دستور زیر به زبان SQL برای انجام این کار استفاده می‌کنیم:

```
Select student_id from grades where entrance = 1400 group by student_id order by avg(grade) desc limit 1
```

در پیاده‌سازی‌های فعلی پایگاه داده‌های بدون شاخص، این کار بصورت سری صورت می‌گیرد یعنی ابتدا یک جستجوی خطی برای پیدا کردن نمرات دانشجوهای ۱۴۰۰ داریم، سپس نمرات با یکدیگر تجمیع می‌شوند و بعد از آن ردیف‌ها بر اساس نمرات بصورت نزولی مرتب‌سازی می‌شوند و در آخر فقط ردیف اول انتخاب می‌شود.

چون پایگاه‌های داده، داده‌ها را هم زمان با خواندن از دیسک، پردازش می‌کنند، معمولاً نیاز به پردازش موازی ندارند، چون سرعت پردازنده خیلی از سرعت دیسک بیشتر است. برای همین سعی شده در این پروژه داده‌ها در رم نگه داری شوند و عملگرهای پایگاه داده بصورت موازی پیاده سازی شوند.

پیاده‌سازی‌ها

این پایگاه‌داده ابتدا برای CPU با C++ نوشته شده بود(gator_db) و پس از آن یک نسخه دوم برای کودا(gator_db_cuda) نوشته شد. بدلیل محدودیت‌های محیط cuda، همه قابلیت های نسخه سی‌پی‌یو در نسخه جی‌پی‌یو وجود ندارند.

فرمت ورودی

فرمت فایل ورودی مجاز، فایل CSV است که در سطر اول نام ستون‌ها و در سطر دوم نوع ستون‌ها مشخص شده. نوع‌های مجاز عبارتند از: int و str

محدودیت

در نسخه gpu چون امکان استفاده از stringهای c++ وجود ندارد، فقط از int پشتیبانی می‌شود. از وارد کردن کوئری بصورت رشته SQL هم پشتیبانی نمی‌شود و کلاً امکانات عملگرها محدودتر هستند. مثلاً عملگر group و sort در cpu، از بیش از یک ستون پشتیبانی می‌کنند ولی در پیاده‌سازی gpu حداکثر می‌توان یک ستون را مشخص کرد. عملگر پرتو یا Projection فقط روی cpu پیاده سازی شده.

تفاوت‌های دیگر

در نسخه cpu از کلاس‌های کاستوم table, column, record, field استفاده شده که کار را نسبتاً آسان می‌کنند ولی در gpu امکان استفاده از کلاس‌های کاستوم وجود ندارد و جدول توسط یک آرایه دو بعدی(int[i][j]) نمایش داده می‌شود که بدلیل عدم امکان اختصاص آسان حافظه روی دوایس، از نمایش تخت آن استفاده شده(int[i * width + j]).

عملگرها

نحوه کار پایگاه‌های داده رابطه‌ای، قواعد ریاضی دارد. این قواعد ریاضی تحت عنوان "جبر رابطه‌ای" یاد می‌شوند. در جبر رابطه‌ای ۵ عملگر اصلی وجود دارد: Restrict, Project, Times, Minus, Union که در تعریف همه عملگرهای دیگر از این ۵ عملگر پایه استفاده می‌شود. از آن جایی که پایگاه پیاده‌سازی شده از نوع رابطه‌ای نیست، همه عملگرها پیاده‌سازی نشده‌اند.

عملگرهای پیاده سازی شده عبارتند از تحدید(Restrict)، تلخیص(Summerize) و مرتب‌سازی.

تحدید

این عملگر برای فیلتر کردن داده‌ها استفاده می‌شود. یک شرط می‌پزیرد و خروجی آن همه سطرهایی است که در شرط صدق می‌کنند، می‌باشد. در زبان SQL این عملگر با WHERE مشخص می‌شود.

در پیاده‌سازی سری، بر اساس اپراتور انتخاب شده، رکوردهایی که در شرط where صادق هستند، در یک آرایه جدید کپی می‌شوند.

در پیاده سازی موازی، هر ردیف به یک ترد داده می‌شود و ترد در آرایه‌ی match مشخص می‌کند که آیا آن ردیف شرط لازم را دارد یا خیر. پس از اتمام کار، سی‌پی‌یو ردیف‌هایی که توسط آرایه‌ی match مشخص شده اند را به ابتدای لیست رکوردها منتقل می‌کند(*شکل ۱*).

شکل ۱. نحوه کارکرد پیاده‌سازی موازی تحدید

تلخیص

این عملگر برای گروه بندی داده‌ها استفاده می‌شود. مثلاً جدول زیر لیست نمرات دانشجویان قبل از این عملگر می‌باشد:

student_id	grade
۹۷۰۳۳۲۴۲	۱۸
۹۸۰۶۵۴۲۱	۱۵
۹۸۰۶۵۴۲۱	۱۶
۹۷۰۳۳۲۴۲	۱۷
۹۷۰۳۳۲۴۲	۱۹
۹۷۰۳۳۲۴۲	۲۰
۹۶۰۶۵۴۲۱	۱۴

بعد از عملیات Summerize/Aggregation روی student_id به این صورت در می‌آید:

student_id	SUM(GRADE)	MAX(GRADE)	MIN(GRADE)	COUNT(GRADE)	AVG(GRADE)
۹۷۰۳۳۲۴۲	۷۴	۲۰	۱۷	۴	۱۸.۵
۹۸۰۶۵۴۲۱	۳۱	۱۶	۱۵	۲	۱۵.۵
۹۶۰۶۵۴۲۱	۱۴	۱۴	۱۴	۱	۱۴

در زبان SQL این عملیات با GROUP BY مشخص می‌شود. در cpu یک الگوریتم با مرتبه زمانی $O(n^2)$ پیاده‌سازی شده که کار تجمیع را روی یک هسته انجام می‌دهد. ستون های sum, count, min, max و avg برای هر ویژگی عددی اضافه می‌شود. در gpu، بدلیل سخت بودن موازی سازی الگوریتم تجمیع از الگوریتم مشابه با روش اجرای مشابه درخت باینری استفاده شده، به این صورت که در مرحله اول هر بلاک ۵۱۲ رکورد را تجمیع می‌کند، سپس در هر دور ۲ برابر می‌شود تا در مرحله آخر، یک بلاک، نتایج تجمیع همه بلاک های قبلی را تجمیع می‌کند.

نحوه دقیق پیاده سازی الگوریتم Group By در GPU

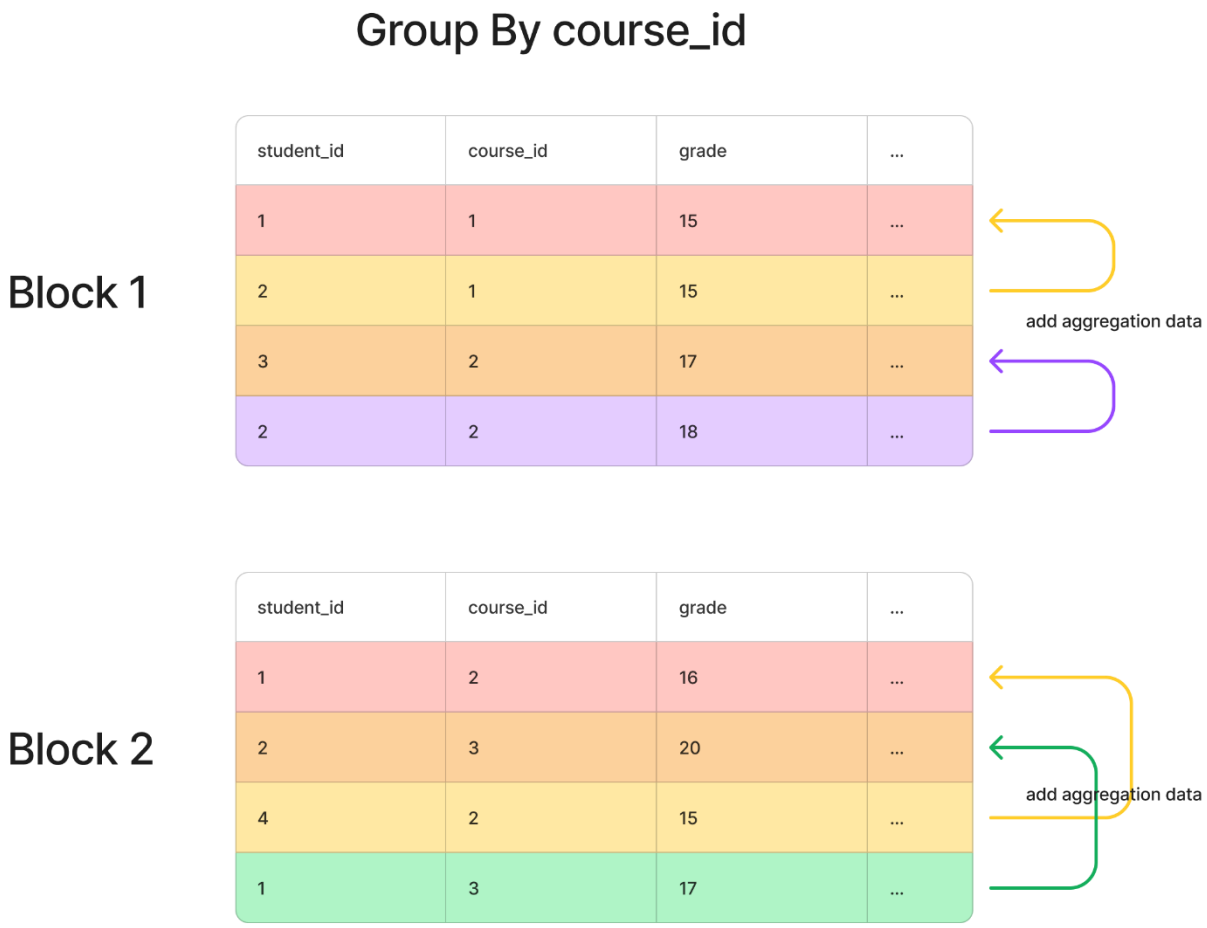
برای نمایش نحوه موازی سازی این عملکرد، فرض کنید که داده‌های زیر را داریم:

student_id	course_id	grade
1	1	15
2	1	15
3	2	17
2	2	18
1	2	16
2	3	20
4	2	15
1	3	17

می‌خواهیم عملیات تلخیص را روی ســـتون course_id انجام دهیم. برای این جدول، الگوریتم را با ۲ بلوک و ۴ ترد فراخوانی می‌کنیم. در اولین دور اجرا هر ترد ســـتون‌های مربوط به اطلاعات Aggregation را بر اساس اطلاعات ردیف خودش پر می‌کند.

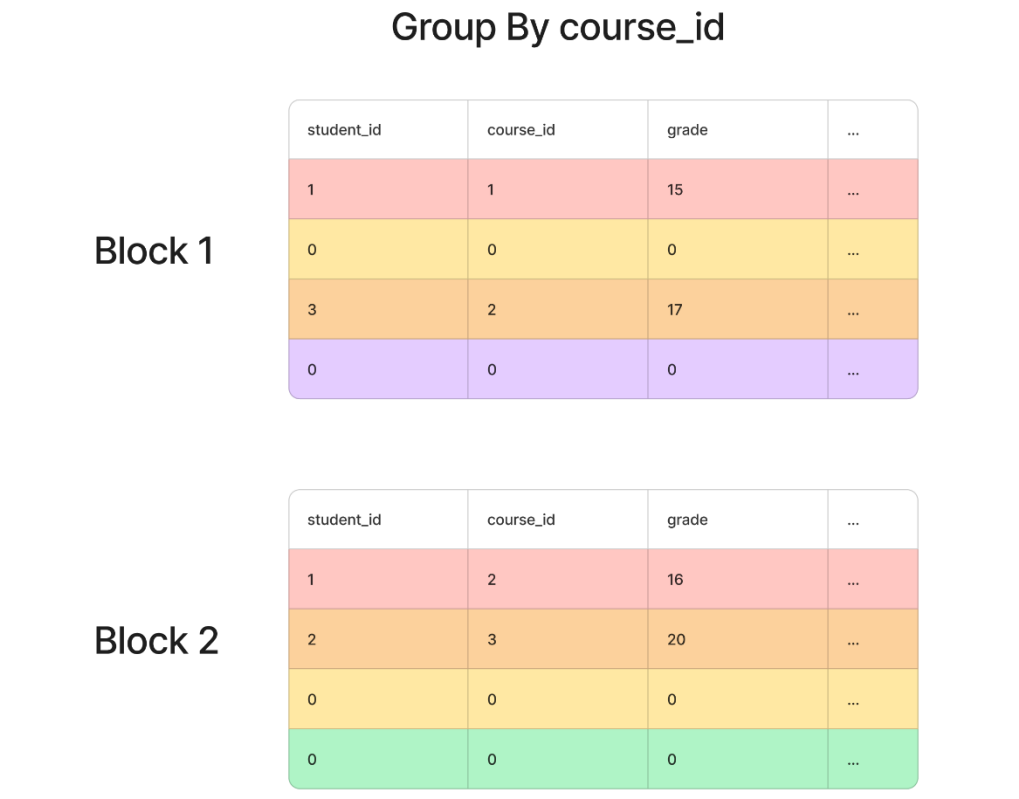
student_id	course_id	grade	sum	count	max	min	avg
1	1	15	15	1	15	15	15
2	1	15	15	1	15	15	15
3	2	17	17	1	17	17	17
2	2	18	18	1	18	18	18
1	2	16	16	1	16	16	16
2	3	20	20	1	20	20	20
4	2	15	15	1	15	15	15
1	3	17	17	1	17	17	17

سپس هر ترد در هر بلوک، **اولین ردیف قبل از خودش که کلید group_by آن با خودش برابر است را پیدا می‌کند**. اگر این ردیف پیدا شد، آن را به عنوان والد ذخیره می‌کند. اگر پیدا نشد یعنی خودش والد است.



سپس هر ردیف غیر parent، داده‌های Aggregation خود را به والدش اضافه می‌کند. مثلاً تعداد count ردیف هفت، ۱ می‌باشد. آن را به والدش اضافه می‌کند تا والدش نماینده ۲ رکورد باشد.

سپس رکوردهای غیر والد، داده‌های خود را صفر می‌کنند و فلگ پردازش شده خود را "۱" می‌کنند.

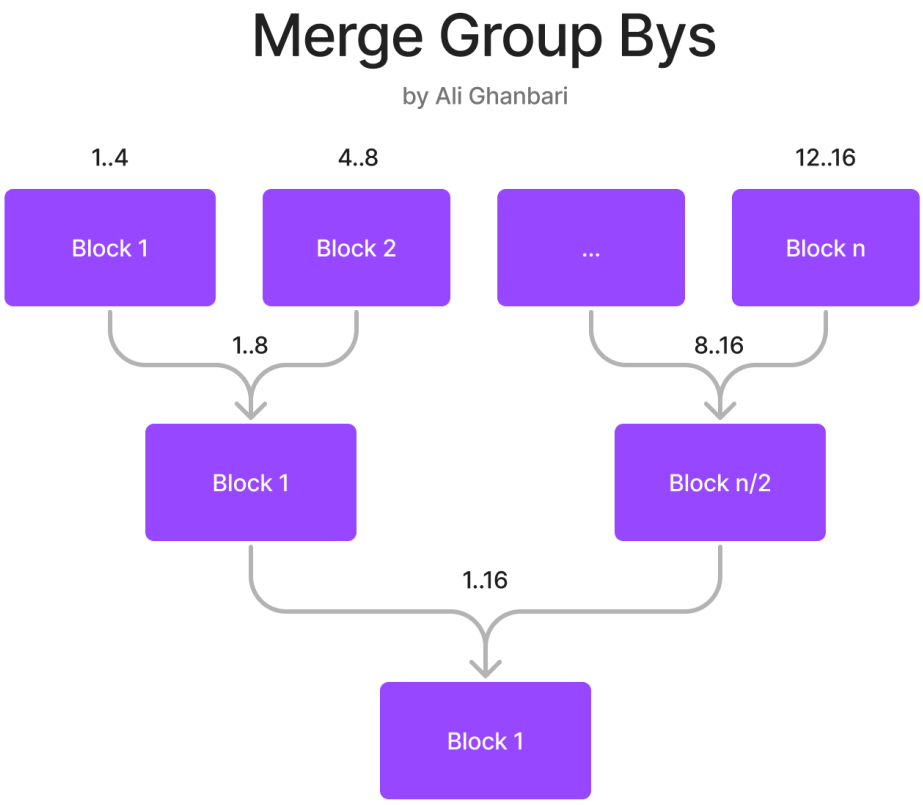


پس از این، یک مرحله از group_by تمام شده و دوباره با تعداد تردهای دوبرابر، group_by را فراخوانی می‌کنیم. با تعداد ۸ ترد، فقط ۱ بلوک نیاز داریم.

پس از اتمام این مرحله همه تجمیع ها روی داده انجام شده و فقط انتقال ردیف‌های پردازش نشده به ابتدای جدول باقی می‌ماند که توسط CPU انجام می‌شود.

student_id	course_id	grade	sum(student_id)	...	count(grade)	...	avg(grade)
1	1	15	...	...	2	...	...
3	2	17	...	...	4	...	...
2	3	20	...	...	2	...	...

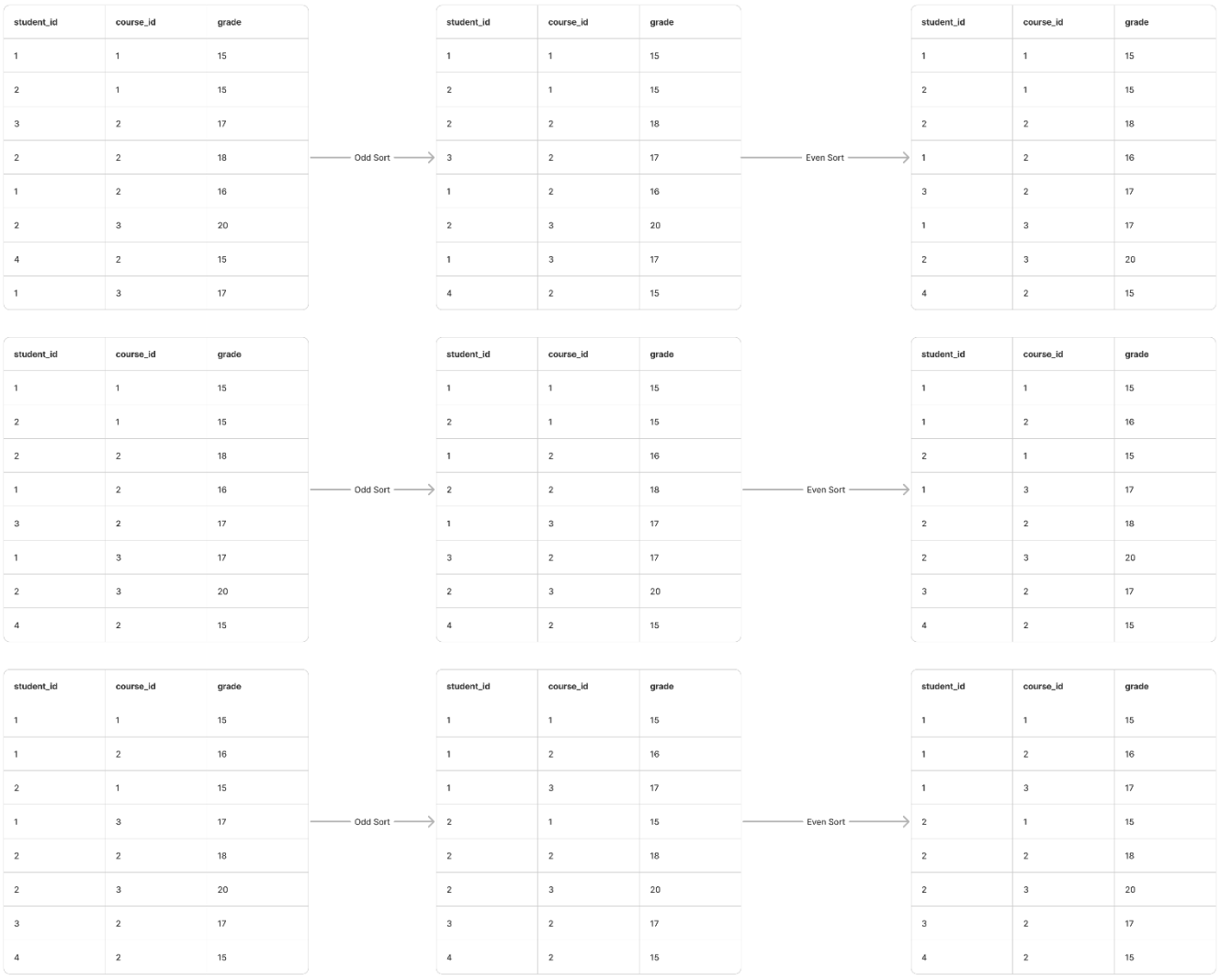
نمایی دیگر از نحوه اجرای این الگوریتم:



مرتب سازی

این عملگر برای مرتب‌سازی داده‌ها بر اساس یک ستون خاص و جهت مرتب سازی استفاده می‌شود.

برای پیاده‌سازی این عملگر از الگوریتم odd-even استفاده شده.



...
N times
...

شکل ۲. مرتب سازی odd-even روی مولفه student_id جدول

دیتاست

برای تولید دیتاست، یک اسکریپت با زبان پایتون نوشته شده، که فایل قابل خوانده شدن توسط برنامه نوشته شده را تولید می‌کند. دیتاست‌های متعددی با تعداد ۶۴k رکورد تا ۴۰۹۶k رکورد تولید شده‌اند.

	A	B	C	D
1	student_id	semester	course_id	grade
2	int	int	int	int
3	40110007	995	142	15
4	40110095	994	136	16
5	40110011	998	163	2
6	40110089	991	139	1
7	40110059	991	144	10
8	40110036	998	134	2
9	40110002	992	159	6
10	40110078	992	151	14
11	40110072	991	146	12
12	40110027	991	128	14
13	40110065	996	119	17
14	40110047	999	124	8
15	40110027	996	104	4
16	40110054	993	148	5
17	40110063	998	113	4
18	40110029	998	122	12
19	40110072	993	117	12
20	40110099	990	105	7

شکل ۳. نمونه داده تولید شده

آزمون

در این بخش کارایی الگوریتم‌های gpu را با معادل cpu آن‌ها مقایسه خواهند شد.

سخت افزار

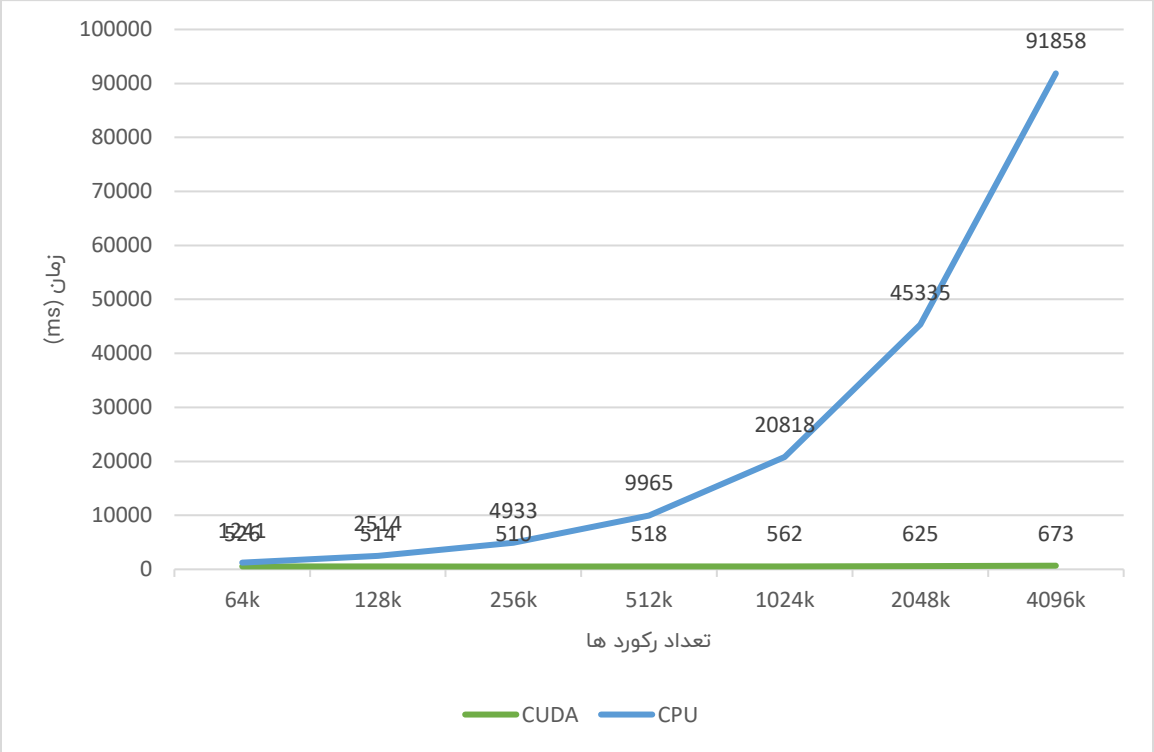
نسخه cpu و نسخه gpu روی دو سیستم متفاوت تست شده‌اند. مشخصات سیستم‌ها در جدول زیر وجود دارد:

	سیستم cpu	**سیستم gpu**
پردازنده	i۵ ۱۲th gen	i۵ ۵th gen
رم	۴۰ gb	۸ gb
کارت گرافیک	بدون کارت گرافیک مجزا	NVIDIA GeForce ۹۲۰M
VRAM	-	۲ gb

آزمون تحدید

برای تست تحدید(Restrict)، از دیتابیس خواسته شد تا فقط نمره‌های قبولی را نمایش دهد. این پرسش در زبان SQL به این صورت است:

```
Select * from grades64k.csv where grade > 9 limit 10
```



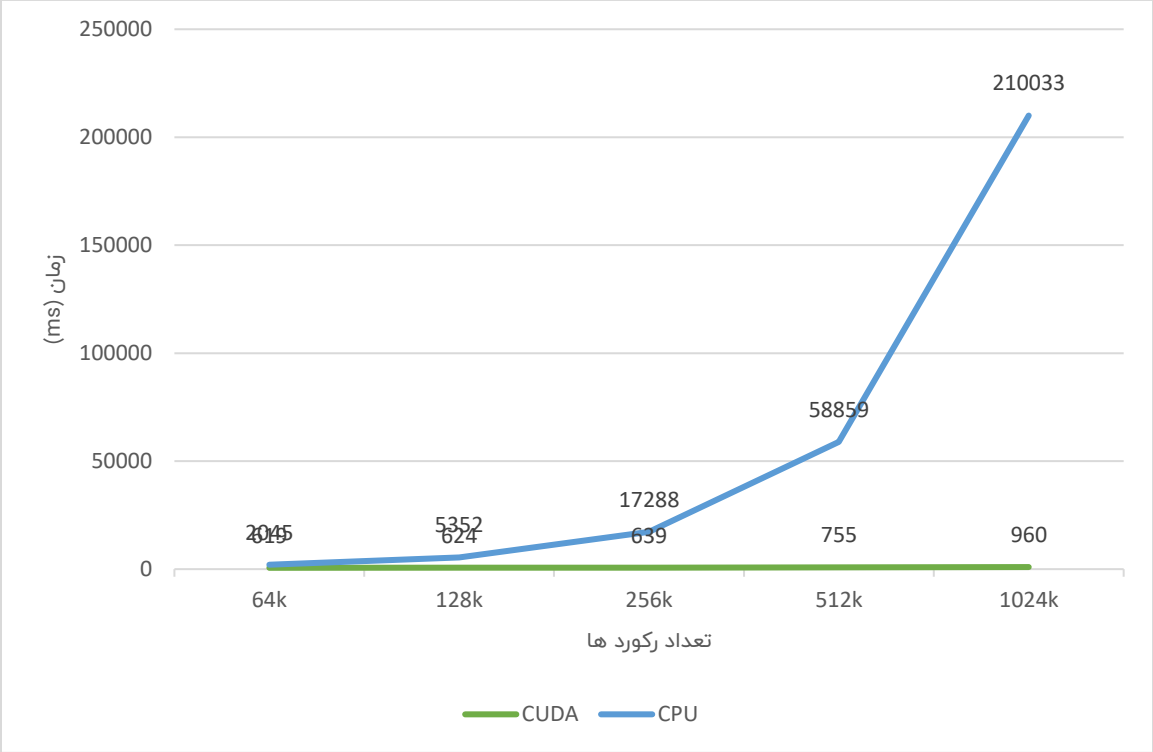
همانگونه که مشاهده می‌شود، زمان پردازش برای cuda تقریبا ثابت می‌ماند و روند افزایش خیلی کمی دارد ولی برای CPU بصورت خطی افزایش می‌یابد که منطقی است چون CPU از یک الگوریتم خطی(O(n)) استفاده می‌کند ولی در GPU به ازای هر رکورد یک ترد ساخته می‌شود که در صورتی که یک کارت گرافیک با تعداد بی نهایت ترد همزمان داشته باشیم، از مرتبه O(۱) می‌باشد.

آزمون تلخیص

برای تست تخلیص(Summerize)، از دیتابیس خواسته شد تا نمره‌های هر درس را تجمیع کند.

این پرسش در زبان SQL به این صورت است:

```
Select * from grades64k.csv group by course_id limit 10
```

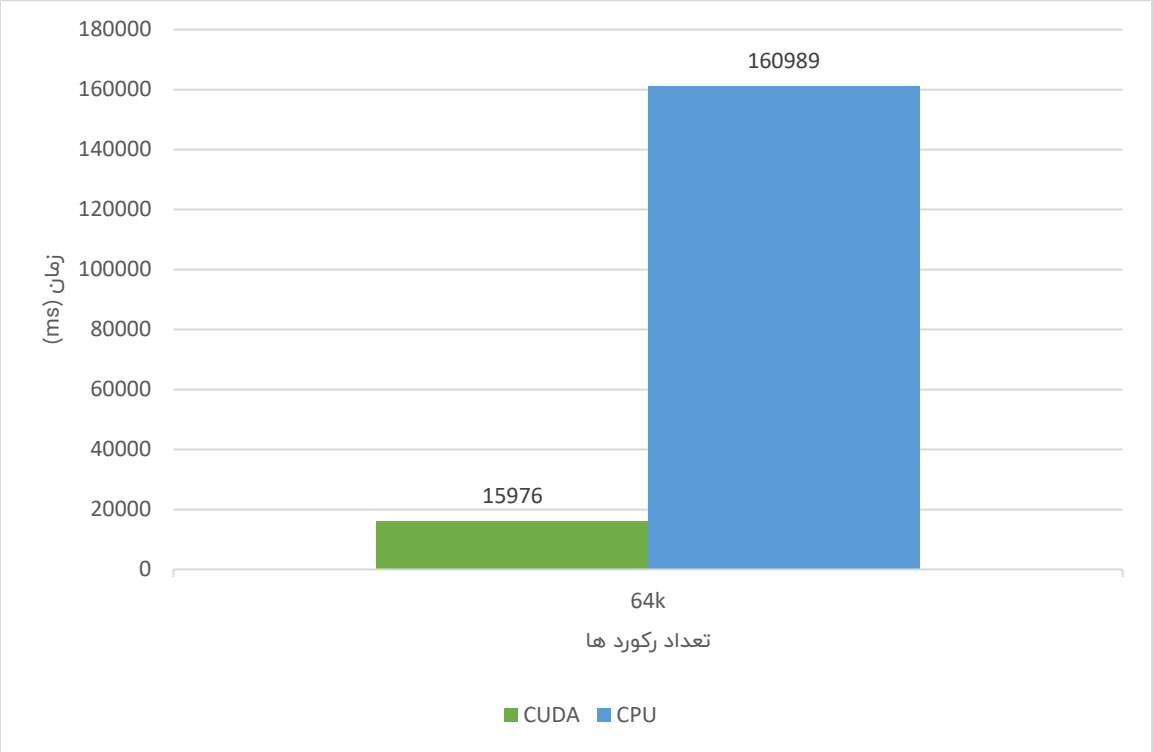


در این آزمون هم دیده می‌شود که پیاده‌سازی gpu خیلی سریعتر است. دلیل اصلی این است که الگوریتم GROUP BY در CPU از مرتبه $O(n^2)$ است ولی در GPU چون از ساختار درختی استفاده می‌کند می‌توانیم بگوییم از مرتبه O(Log n) است. برای همین رکوردهای بیشتر از ۱۰۲۴k تست نشدند چون زمان cpu آن‌ها احتمالاً بالای ۱۰۰۰ ثانیه است.

آزمون مرتب سازی

برای تست مرتب سازی، از دیتابیس خواسته شد تا رکوردها را بر اساس شماره ترم بصورت صعودی مرتب کن. این پرسش در زبان SQL به این صورت است:

Select * from grades64k.csv order by semester asc limit 10



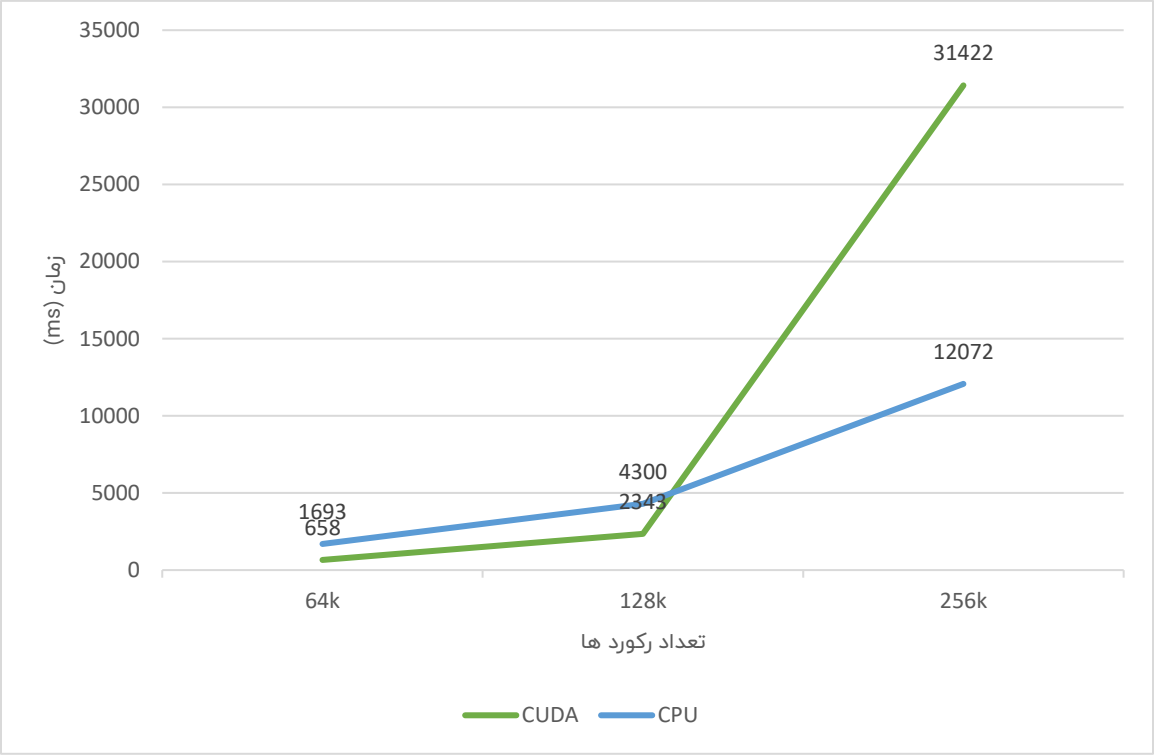
این الگوریتم برای ۶۴ هزار ردیف، روی gpu، ۱۰ برابر سریعتر از cpu است. برای تعدادهای بالاتر، زمان اجرای خیلی بالایی روی cpu دارد و بنابراین تست نشده است.

این الگوریتم از مرتبه $O(n^2)$ است و پرفورمنس خوبی ندارد. با این حال، شاید اگر از یک الگوریتم مرتب سازی دیگر از مرتبه $O(n^2)$ بصورت درخت باینری(merged) استفاده می‌شد، شاید به پرفورمنس بهتری دست یافته می‌شد.

آزمون تحدید، تخلیص و مرتب سازی

از مزایای پایگاه داده، امکان استفاده از چندین عملگر در یک پرسش است. برای تست هر سه عملگر در یک آزمون، از دیتابیس خواسته شد تا درس‌ها را بر اساس معدل نمره‌های قبولی بصورت نزولی مرتب کند. این پرسش در زبان SQL به این صورت است:

```
Select * from grades64k.csv where grade > 9 group by course_id order by avg(grade) desc limit 10
```



در این آزمون تا ۱۲۸k ردیف، پرفورمنس قابل انتظار نمایش داده می‌شود ولی بعد از آن اتفاق عجیبی می‌افتد و برنامه gpu بیش از ۳ برابر بیشتر نسبت به برنامه cpu برای پردازش درخواست طول می‌کشد. حدس من این است که پس از انجام عملیات تخلیص، عرض جدول از ۴ ستون به ۱۹ ستون افزایش می‌یابد و این افزایش حجم، هر گونه امکان قرارگیری بهینه در حافظه را از بین می‌برد. اگر دلیل واقعی این باشد باید به جای مرتب سازی مستقیم، فقط شاخصی از ردیف‌ها را در gpu مرتب سازی کرد و جابه‌جایی در cpu اتفاق بیافتد.

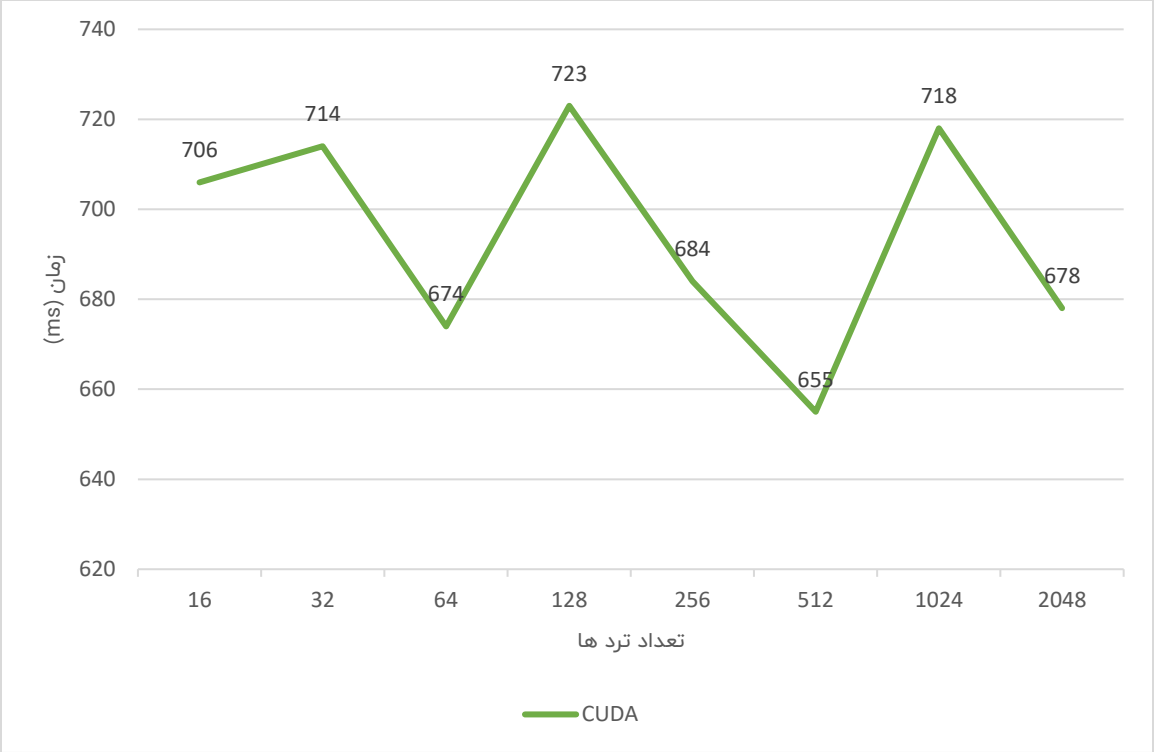
تغییر پارامترها

در این بخش تغییر تعداد ترد و بلاک‌ها را در عملکرد، عملگرهای ارائه شده بررسی می‌کنیم.

در همه الگوریتم‌ها تعداد بلوک‌ها از تقسیم تعداد رکوردها بر تردها بدست می‌آید.

تحدید

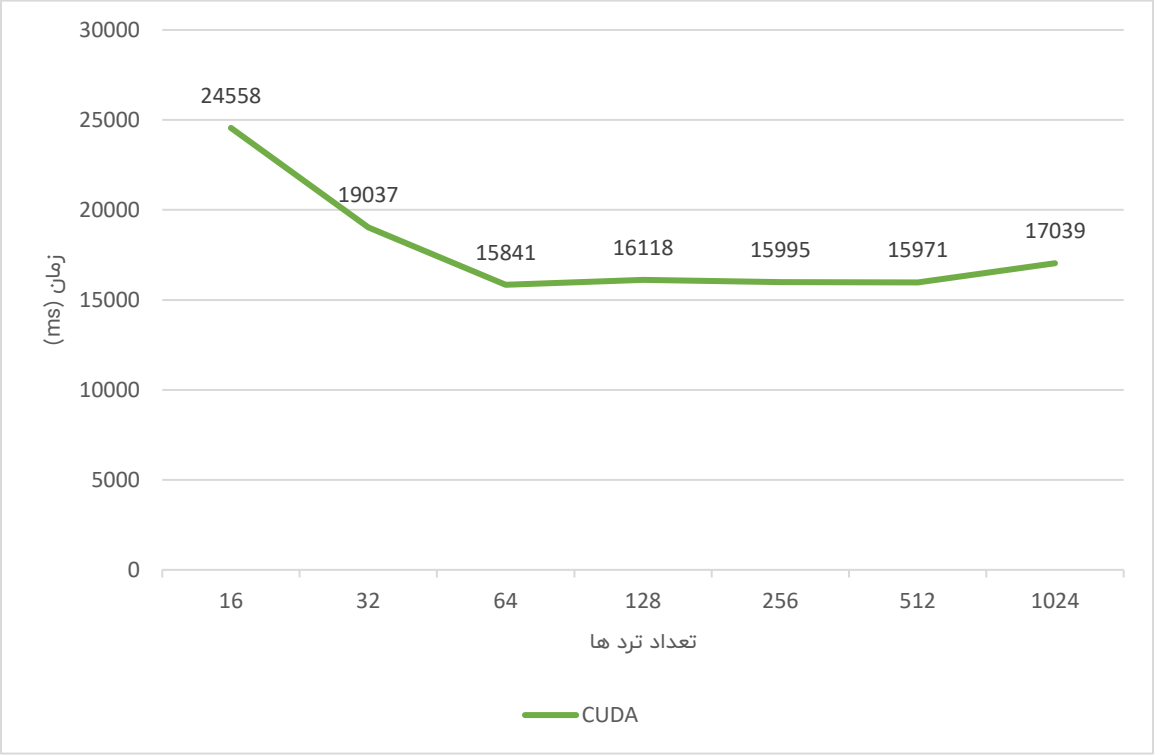
نتایج آزمون تعداد تردهای ۱۶ تا ۲۰۴۸ روی ۴۰۹۶k ردیف در نمودار زیر نمایش داده شده:



با تغییر تعداد تردها، هیچ تغییر خاصی مشاهده نمی‌شود. این تنها عملگری است که در آن تردها هیچ کاری به دیتای تردهای دیگر ندارند.

مرتب سازی

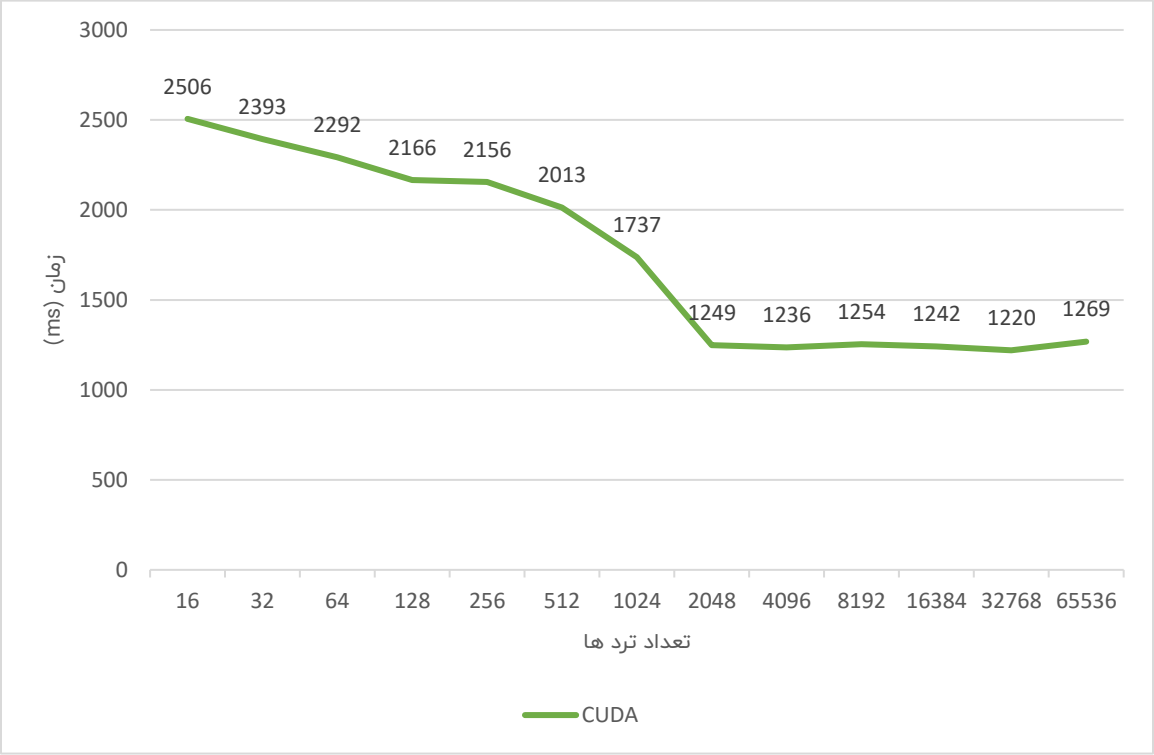
نتایج آزمون تعداد تردهای ۱۶ تا ۱۰۲۴ روی ۶۴k ردیف در نمودار زیر نمایش داده شده:



با تغییر تعداد تردها تا ۶۴تا کاهش قابل توجهی در زمان مرتب سازی وجود دارد ولی پس از آن تا ۵۱۲ ترد تغییری مشاهده نمی شود. از ۱۰۲۴ ترد به بعد زمان پردازش دوباره افزایش می‌یابد.

تلخیص

نتایج آزمون تعداد تردهای ۱۶ تا ۶۵۵۳۶ روی ۴۰۹۶k ردیف در نمودار زیر نمایش داده شده:



با افزایش تعداد تردها زمان اجرا روند نزولی پیدا می‌کند و بین ۲۰۴۸ تا ۴۰۹۶ ترد به ازای هر بلاک به کمترین زمان اجرا می‌رسد.

نتیجه گیری

پایگاه‌های داده یکی از مهمترین بخش‌های همه نرم‌افزارهای اطراف ما هستند. با این حال تاکنون همیشه روی CPU و بصورت سری اجرا می‌شدند. در این پروژه موازی سازی یک پایگاه داده روی GPU بررسی شد و به این نتیجه رسیده شد که امکان موازی سازی برخی عملگرهای آن‌ها وجود دارد. طبق آزمون‌های انجام شده، به این نتیجه رسیده شد که می‌توان برنامه‌ای نوشت که روی چندین پردازنده کوچک قدیمی(کارت گرافیک سطح پایین ۲۰۱۶) از یک پردازنده سریع خیلی جدید(پردازنده اینتل ۲۰۲۱) سریعتر کار کند.

ضمیمه

پیاده سازی سری تحدید(Restrict)

```
table restrict(table t, where restriction){

        int control_field = -1;
        for (size_t i = 0; i < t.width; i++)
        {
                if (t.columns[i].name == restriction.column) {
                        control_field = i;
                        break;
                }
        }

        std::vector<record> filtered;

        auto eq = [&restriction](field& f) {return (f == restriction.value); };
        auto neq = [&restriction](field& f) {return !(f == restriction.value); };
        auto lt = [&restriction](field& f) {return (f < restriction.value); };
        auto lte = [&restriction](field& f) {return (f <= restriction.value); };
        auto gt = [&restriction](field& f) {return (f > restriction.value); };
        auto gte = [&restriction](field& f) {return (f >= restriction.value); };

        if (restriction.op == "=") {
                std::copy_if(t.records, t.records + t.length - 2, std::back_inserter(filtered),
[control_field,&eq](record& record) {return  eq(record.fields[control_field]); });
        }
        else if (restriction.op == "!=" || restriction.op == "<>" || restriction.op == "~=") {
                std::copy_if(t.records, t.records + t.length, std::back_inserter(filtered),
[control_field, &neq](record& record) {return  neq(record.fields[control_field]); });
        }
        else if (restriction.op == "<") {
                std::copy_if(t.records, t.records + t.length - 2, std::back_inserter(filtered),
[control_field, &lt](record& record) {return  lt(record.fields[control_field]); });
        }
        else if (restriction.op == "<=") {
                std::copy_if(t.records, t.records + t.length - 2, std::back_inserter(filtered),
[control_field, &lte](record& record) {return  lte(record.fields[control_field]); });
        }
        else if (restriction.op == ">") {
                std::copy_if(t.records, t.records + t.length - 2, std::back_inserter(filtered),
[control_field, &gt](record& record) {return  gt(record.fields[control_field]); });
        }
        else if (restriction.op == ">=") {
                std::copy_if(t.records, t.records + t.length - 2, std::back_inserter(filtered),
[control_field, &gte](record& record) {return  gte(record.fields[control_field]); });
        }

        t.length = filtered.size();
        t.records = new record[t.length];
        for (size_t i = 0; i < t.length; i++)
        {
                t.records[i] = filtered[i];
        }

        return t;
}
```

پیاده سازی موازی تحدید(Restrict)

```
__global__ void filter_kernel(int* data, int width, int length, int filter_col, int op, int
value, int* match) {
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        if (tid < length)
        {
                int v = data[tid * width + filter_col];
                if (op == 0 /*=*/)
                {
                        match[tid] = (v == value);
                }
                else if (op == 1 /*>*/) {
                        match[tid] = (v > value);
                }
                else if (op == 2 /*<*/) {
                        match[tid] = (v < value);
                }
        }
}
```

```
cudaError_t filter(int* data, int width, int& length, int filter_col, int op, int value)
{
          int* device_table;
          int* device_matches;
          cudaError_t cudaStatus;

          cudaStatus = cudaSetDevice(0);
          if (cudaStatus != cudaSuccess) return cudaStatus;

          cudaStatus = cudaMalloc((void**)&device_table, sizeof(int) * length * width);
          if (cudaStatus != cudaSuccess) return cudaStatus;

          cudaStatus = cudaMalloc((void**)&device_matches, sizeof(int) * length);
          if (cudaStatus != cudaSuccess) return cudaStatus;

          cudaStatus = cudaMemcpy(device_table, data, sizeof(int) * length * width, cudaMemcpyHostToDevice);
          if (cudaStatus != cudaSuccess) return cudaStatus;

          int numThreads = 256;
          int numBlocks = (length + numThreads - 1) / numThreads;

          filter_kernel<<<numBlocks, numThreads >>>(device_table, width, length, filter_col, op, value, device_matches);

          cudaDeviceSynchronize();

          int* matches = new int[length];

          cudaStatus = cudaMemcpy(matches, device_matches, sizeof(int) * length, cudaMemcpyDeviceToHost);
          if (cudaStatus != cudaSuccess) return cudaStatus;

          cudaStatus = cudaFree(device_table);
          if (cudaStatus != cudaSuccess) return cudaStatus;

          cudaStatus = cudaFree(device_matches);
          if (cudaStatus != cudaSuccess) return cudaStatus;

          // fix matches order
          int m = 0;
          for (int i = 0; i < length; i++)
          {
                  if (matches[i]) {
                          for (int j = 0; j < width; j++)
                          {
                                  data[m * width + j] = data[i * width + j];
                          }
                          m++;
                  }
          }

          length = m;
          return cudaStatus;
}
```

پیاده سازی سری تلخیص(Summerize)

```
table aggregation(table t, std::vector<std::string> group_by) {
	std::string operations[] = { "sum", "count", "max", "min", "avg" };
	int old_width = t.width;

	std::vector<std::string> int_cols;
	std::vector<int> int_cols_indexes;
	std::vector<int> group_indexes;
	for (int j = 0; j < old_width; j++) {
		if (std::find(group_by.begin(), group_by.end(), t.columns[j].name) != group_by.end()) {
			group_indexes.push_back(j);
		}
		if (t.columns[j].type == DATA_TYPE_INT) {
			int_cols.push_back(t.columns[j].name);
			int_cols_indexes.push_back(j);
		}
	}

	t.width += int_cols.size() * 5;
	column* cols = new column[t.width];
	for (int j = 0; j < old_width; j++) {
		cols[j] = t.columns[j];
	}
	t.columns = cols;

	int i = old_width;
	for (auto& ic : int_cols)
	{
		for (int j = 0; j < 5; j++) {
			std::string& op = operations[j];
			column c;
			c.name = op + "(" + ic + ")";
			c.type = DATA_TYPE_INT;
			t.columns[i++] = c;
		}
	}

	// group by similar field
	int len = 0;
	record* records = new record[t.length];

	for (size_t i = 0; i < t.length - 2; i++) {
		auto current = t.records[i];
		if (current.fields[group_indexes[0]].get_type() == DATA_TYPE_NONE) {
			continue;
		}

		record merged;
		merged.fields = new field[t.width];

		for (size_t j = 0; j < old_width; j++) {
			merged.fields[j] = current.fields[j];
		}

		int* intColIndexArr = &int_cols_indexes[0];
		for (int j = 0; j < int_cols.size(); j++) {
			merged.fields[old_width + (j * 5) + 0] = current.fields[intColIndexArr[j]]; // sum
			merged.fields[old_width + (j * 5) + 1] = field(1); // count
			merged.fields[old_width + (j * 5) + 2] = current.fields[intColIndexArr[j]]; // max
			merged.fields[old_width + (j * 5) + 3] = current.fields[intColIndexArr[j]]; // min
			merged.fields[old_width + (j * 5) + 4] = 0; // avg
		}

		for (size_t j = i + 1; j < t.length - 2; j++)
		{
			auto next = t.records[j];
			int match = 1;
			for (int gi : group_indexes) {
				if (current.fields[gi] != next.fields[gi]) {
					match = 0;
					break;
				}
			}

			if (match) {
				for (size_t k = 0; k < int_cols.size(); k++)
				{
					field& f = next.fields[intColIndexArr[k]];
					merged.fields[old_width + (k * 5) + 0] = merged.fields[old_width + (k * 5) + 0] + f; // sum
					merged.fields[old_width + (k * 5) + 1] = merged.fields[old_width + (k * 5) + 1] + field(1);
// count
					merged.fields[old_width + (k * 5) + 2] = f.i() > merged.fields[old_width + (k * 5) + 2].i() ?
f : merged.fields[old_width + (k * 5) + 2]; // max
					merged.fields[old_width + (k * 5) + 3] = f.i() < merged.fields[old_width + (k * 5) + 2].i() ?
f : merged.fields[old_width + (k * 5) + 2]; // min
					merged.fields[old_width + (k * 5) + 4] = field(merged.fields[old_width + (k * 5) + 0].i() /
merged.fields[old_width + (k * 5) + 1].i()); // avg
				}
				next.fields[group_indexes[0]].make_null();
			}
		}

		records[len++] = merged;
		current.fields[group_indexes[0]].make_null();
	}

	t.length = len;
	t.records = records;

	return t;
}
```

پیاده سازی موازی تلخیص(Summerize)

```
__global__ void group_kernel(int* data, int raw_width, int width, int length, int group_by, int* proccessed, int init) {
	int block_start_index = blockIdx.x * blockDim.x;
	int block_end_index = block_start_index + blockDim.x;
	int global_index = block_start_index + threadIdx.x;
	const int debug_clear = false;
	__shared__ int parent_count[1];

	if (global_index >= length) return;
	if (proccessed[global_index]) return;

	if (init) {
		// init expansion data { "sum", "count", "max", "min", "avg" } with default values
		int a = 0;
		for (int k = 0; k < raw_width; k++)
		{
			if (k == group_by) {
				continue;
			}

			// sum
			data[(global_index * width) + raw_width + (a * 5) + SUM] = data[(global_index * width) + k];
			// count
			data[(global_index * width) + raw_width + (a * 5) + COUNT] = 1;
			// max
			data[(global_index * width) + raw_width + (a * 5) + MAX] = data[(global_index * width) + k];
			// min
			data[(global_index * width) + raw_width + (a * 5) + MIN] = data[(global_index * width) + k];
			// avg
			data[(global_index * width) + raw_width + (a * 5) + AVG] = 0;
			// next expansion position
			a++;
		}
	}

	// wait for all threads to copy data
	__syncthreads();

	// find parent thread id
	int parent_id = global_index;
	for (int i = block_start_index; i < global_index; i++)
	{
		if (data[i * width + group_by] == data[global_index * width + group_by]) {
			parent_id = i;
			break;
		}
	}

	// count parents
	if (parent_id == global_index) {
		atomicAdd(&parent_count[0], 1);
	}

	// add tid data to parent data
	if (parent_id != global_index) {
		// add expansion data { "sum", "count", "max", "min", "avg" } to parent row
		int a = 0;
		for (int k = 0; k < raw_width; k++)
		{
			if (k == group_by) {
				continue;
			}
			int self_expansion_start = global_index * width + raw_width;
			int parent_expansion_start = parent_id * width + raw_width;
			// sum
			int sum = data[self_expansion_start + (a * 5) + SUM];
			atomicAdd(&data[parent_expansion_start + (a * 5) + SUM], sum);
			// count
			int count = data[self_expansion_start + (a * 5) + COUNT];
			atomicAdd(&data[parent_expansion_start + (a * 5) + COUNT], count);
			// max
			int max = data[self_expansion_start + (a * 5) + MAX];
			atomicMax(&data[parent_expansion_start + (a * 5) + MAX], max);
			// min
			int min = data[self_expansion_start + (a * 5) + MIN];
			atomicMin(&data[parent_expansion_start + (a * 5) + MIN], min);

			// next expansion position
			a++;
		}

	}

	// wait for all threads to add their data to parent row
	__syncthreads();

	// calculate avg on parent rows
	if (global_index == parent_id) {
		int a = 0;
		for (int k = 0; k < raw_width; k++)
		{
			if (k == group_by) {
				continue;
			}

			int self_expansion_start = global_index * width + raw_width;
			// avg
			int avg = data[self_expansion_start + (a * 5) + SUM] / data[self_expansion_start + (a * 5) + COUNT];
			data[self_expansion_start + (a * 5) + AVG] = avg;

			// next expansion position
			a++;
		}

	}

	// zero out non parent rows
	if (global_index != parent_id) {
		for (int i = 0; i < width; i++)
		{
			data[global_index * width + i] = 0;
		}

		// mark used
		proccessed[global_index] = 1;
	}

	return;
}
```

```
cudaError_t group(int*& data, int& width, int& length, int group_by, std::string& header)
{
	// expand table width
	int expanded_width = width + ((width - 1) * 5);
	int* expanded = new int[length * expanded_width];

	// copy data
	for (int i = 0; i < length; i++)
	{
		for (int j = 0; j < expanded_width; j++)
		{
			if (j < width) {
				expanded[i * expanded_width + j] = data[i * width + j];
			}
			else {
				expanded[i * expanded_width + j] = 0;
			}
		}
	}

	// add headers
	std::string operations[] = { "sum", "count", "max", "min", "avg" };
	for (int i = 0; i < width; i++)
	{
		if (i == group_by) {
			continue;
		}
		for (int j = 0; j < 5; j++) {
			std::string& op = operations[j];
			std::string& name = op + "(" + std::to_string(i) + ")";
			header += ", " + name;
		}
	}

	// allocate device data
	int* device_table;
	int* device_processed;
	cudaError_t cudaStatus;

	cudaStatus = cudaSetDevice(0);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	cudaStatus = cudaMalloc((void**)&device_table, sizeof(int) * length * expanded_width);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	cudaStatus = cudaMalloc((void**)&device_processed, sizeof(int) * length);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	cudaStatus = cudaMemcpy(device_table, expanded, sizeof(int) * length * expanded_width, cudaMemcpyHostToDevice);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	cudaStatus = cudaMemset(device_processed, 0, sizeof(int) * length);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	// binary tree group by
	// start with small group bys than go to larger group bys

	int threads = length > 256 ? 256 : length / 8;
	int blocks = length / threads;

	// inital run
	group_kernel << <blocks, threads >> > (device_table, width, expanded_width, length, group_by, device_processed, true);

	do {
		threads *= 2;
		blocks = (length + threads - 1) / threads;
		group_kernel << <blocks, threads >> > (device_table, width, expanded_width, length, group_by, device_processed, false);
	} while (blocks != 1);

	cudaDeviceSynchronize();

	// copy to host
	int* processed = new int[length];

	cudaStatus = cudaMemcpy(processed, device_processed, sizeof(int) * length, cudaMemcpyDeviceToHost);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	cudaStatus = cudaMemcpy(expanded, device_table, sizeof(int) * length * expanded_width, cudaMemcpyDeviceToHost);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	cudaStatus = cudaFree(device_table);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	cudaStatus = cudaFree(device_processed);
	if (cudaStatus != cudaSuccess) return cudaStatus;

	// sort
	int len = 0;
	for (int i = 0; i < length; i++)
	{
		if (processed[i]) {
			continue;
		}

		if (i != len) {
			for (int j = 0; j < expanded_width; j++)
			{
				expanded[len * expanded_width + j] = expanded[i * expanded_width + j];
			}
		}
		len++;
	}

	data = expanded;
	width = expanded_width;
	length = len;

	return cudaStatus;
}
```

پیاده سازی سری مرتب سازی

```
table order(table t, std::vector<std::string> order_by, bool asc)
{
	std::vector<int> compare_fields;
	for (auto& order : order_by) {
		for (int i = 0; i < t.width; i++) {
			if (t.columns[i].name == order && t.columns[i].type == DATA_TYPE_INT) {
				compare_fields.push_back(i);
			}
		}
	}

	auto cmpr = [&compare_fields, asc](record a, record b) -> bool {
		int acc = 0;
		for (auto c : compare_fields) {
			acc += a.fields[c].i() - b.fields[c].i();
		}
		if (asc) {
			return acc < 0;
		}
		else {
			return acc > 0;
		}
	};

	int i, j, k;

	for (i = 0; i < t.length - 2; i++) {
		if (i % 2 == 0) {
			// Even phase
			for (j = 2; j < t.length - 2; j += 2) {
				if (cmpr(t.records[j], t.records[j-1])) {
					record temp = t.records[j];
					t.records[j] = t.records[j - 1];
					t.records[j - 1] = temp;
				}
			}
		}
			else {
				// Odd phase
				for (j = 1; j < t.length - 2; j += 2) {
					if (cmpr(t.records[j], t.records[j - 1])) {
						record temp = t.records[j];
						t.records[j] = t.records[j - 1];
						t.records[j - 1] = temp;
					}
				}
			}
	}

	// std::sort(t.records, t.records + t.length - 2, cmpr);

	return t;
}
```

پیاده سازی موازی مرتب سازی

```
__global__ void odd_sort_kernel(int* table, int width, int length, int sort_key, int asc) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int k, t;

        if (i % 2 != 0 && i < length) {
                // Odd phase
                if (compare_rows(table + (i * width), table + ((i - 1) * width), sort_key, asc))
{
                        // swap
                        for (k = 0; k < width; k++)
                        {
                                t = table[(i - 1) * width + k];
                                table[(i - 1) * width + k] = table[i * width + k];
                                table[i * width + k] = t;
                        }
                }
        }
}
```

```
__global__ void even_sort_kernel(int* table, int width, int length, int sort_key, int asc) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int k, t;

        if (i % 2 == 0 && i < length) {
                // Even phase
                if (compare_rows(table + (i * width), table + ((i - 1) * width), sort_key, asc))
{
                        // swap
                        for (k = 0; k < width; k++)
                        {
                                t = table[(i - 1) * width + k];
                                table[(i - 1) * width + k] = table[i * width + k];
                                table[i * width + k] = t;
                        }
                }
        }
}
```

```
cudaError_t sort(int* data, int width, int length, int order_col, int asc) {
            int* device_table;
            cudaError_t cudaStatus;

            cudaStatus = cudaSetDevice(0);
            if (cudaStatus != cudaSuccess) return cudaStatus;

            cudaStatus = cudaMalloc((void**)&device_table, sizeof(int) * length * width);
            if (cudaStatus != cudaSuccess) return cudaStatus;

            cudaStatus = cudaMemcpy(device_table, data, sizeof(int) * length * width, cudaMemcpyHostToDevice);
            if (cudaStatus != cudaSuccess) return cudaStatus;

            const int base_threads = 512;
            int threads = length > base_threads ? base_threads : length;
            int blocks = length / threads;

            for (int i = 0; i < length; i++)
            {
                        odd_sort_kernel << <blocks, threads >> > (device_table, width, length, order_col, asc);
                        even_sort_kernel << <blocks, threads >> > (device_table, width, length, order_col, asc);
            }

            cudaDeviceSynchronize();

            cudaStatus = cudaMemcpy(data, device_table, sizeof(int) * length * width, cudaMemcpyDeviceToHost);
            if (cudaStatus != cudaSuccess) return cudaStatus;

            cudaStatus = cudaFree(device_table);
            if (cudaStatus != cudaSuccess) return cudaStatus;

            return cudaStatus;
}
```