به نام خدا

گزارش پروژه درس برنامه نویسی و ساختارهای موازی

Nearest-Neighbour Distance Histogram

دکتر متین هاشمی

نیمسال دوم ۹۸-۹۷

مهرسا پوریا – ۹۵۱۰۱۲۴۷

تینا بهنیا – ۹۵۱۰۱۱۹۶

1000000 (N_ref) نقطه رفرنس و 10000 (N) نقطه query در اختیار ما قرار داده شده است. قرار است با الگوریتم موازی N هیستوگرام در خروجی تحویل دهیم که تعداد binهای هیستوگرام توسط ورودی K تعیین می شود و K عددی بین 1 تا 5000 است. برای این کار چندین مرحله را طی می کنیم. در ادامه به توضیح مراحل پیاده شده در کد می پردازیم.

برای به دست آوردن هیستوگرام یک نقطه query باید ابتدا فاصله این نقطه را با تمام نقاط رفرنس به دست بیاوریم. سپس ماکسیمم و مینیمم اعداد محاسبه شده را پیدا کنیم و بر اساس مقدار K مرزهای هر bin در هیستوگرام را تعیین کنیم. در نهایت تعداد فواصلی که متعلق به یک بازه هستند را پیدا کرده و به این ترتیب مقدار هر بین هیستوگرام نقطه مورد نظر به دست می آید.

با توجه به این که حجم داده های مساله زیاد است و در حافظه GPU که تقریبا برابر با 2^{30}B است، جا نمی شوند، مجبوریم محاسبات داده ها را تکه تکه انجام دهیم. از طرفی برای به دست آوردن هیستوگرام یک نقطه query لازم داریم فاصله این نقطه را با تمام نقاط رفرنس داشته باشیم تا بتوانیم با به دست آوردن مینیمم و ماکسیمم فاصله مرزهای هر bin هیستوگرام را به طور صحیح به دست بیاوریم. پس برای این که مجبور به جا به جایی زیاد بین حافظه GPU و CPU نشویم ابتدا تمام نقاط رفرنس را در حافظه GPU کپی می کنیم. سپس در هر مرحله تعدادی از نقاط query را در GPU کپی کرده و تمام مراحل لازم برای به دست آوردن هیستوگرام این تعداد از نقاط را با چند kernel call انجام می دهیم و سپس برای نقاط query جدید را جایگزین نقاط قبلی می کنیم.

برای محاسبه هیستوگرام به چند کرنل مختلف نیاز داریم:

۱) Distcal : در این کرنل هدف به دست آوردن فاصله تک تک نقاط query با نقاط رفرنس است. پس آرایه ref_c که شامل تمام نقاط رفرنس است و آرایه query_c که شامل تعداد مشخصی از تقاط query است را به کرنل می دهیم. در نهایت فواصل این نقاط از هم در آرایه dist_c ذخیره می شود به طوری که N_ref فاصله مربوط به یک query پشت سر هم قرار بگیرند. محتوای این آرایه را تا محاسبه کامل هیستوگرام این دسته از نقاط query تغییر نمی دهیم.
۲) Minmax : در این کرنل هر بلاک را مسئول محاسبه مینیمم و ماکسیمم فاصله ی یک query می کنیم. برای این کار در هر بلاک ۶۴ ترد لانچ می کنیم که هر یک در ابتدا مسئول به دست آوردن مینیمم و ماکسیمم بین ۱۵۶۲۵ فاصله مربوط به query متناظر با بلاک هستند. این مقادیر مینیمم و ماکسیمم را در یک حافظه shared نگه می داریم. بعد از اتمام کار تمام تردها، ترد صفرم را مسئول پیدا کردن مینیمم (و ماکسیمم) بین ۶۴ عدد به دست آمده می کنیم. در نهایت مقادیر به دست آمده برای هر query را در آرایه border_c ذخیره می کنیم. دقت کنید که با داشتن مینیمم و ماکسیمم یک query و K مرزهای هیستوگرام مشخص است.
۳) Histcall : هر بلاک در این کرنل، با ۶۴ ترد، ۶۴ هیستوگرام برای هر query محاسبه می کند. به طوری که هر هیستوگرام مربوط به توزیع ۱۵۶۲۵ فاصله query باشد. پس هر ترد با توجه به مینیمم و ماکسیمم به دست آمده در بخش قبل، K عدد مربوط به بین های هیستوگرام ایجاد می کند و در نتیجه هر بلاک 64*K عدد محاسبه می کند. این اعداد را در آرایه hist_c2 ذخیره می کنیم.
۴) Adder : در نهایت ۲۰۰ هیستوگرام به دست آمده در بخش قبل را با هم جمع می کنیم تا هیستوگرام نهایی هر query به دست بیاید. این مقادیر را آرایه hist_c ذخیره می کنیم و در آرایه hist ، در CPU ذخیره می کنیم.

پس از کپی کردن هیستوگرام های به دست آمده به سراغ دسته جدیدی از query ها می رویم تا تمام هیستوگرام های لازم را محاسبه کنیم.

با توجه به توضیحات بالا به آرایه های زیر در حافظه گلوبال GPU نیاز داریم. (سایز و نوع هر یک از متغیرها در مقابل آن ذکر شده است. منظور از Q تعداد queryهایی است که در هر مرحله به کرنل ها می دهیم.):

Variable	type	Size
Ref_c	Float	N_ref × 128(dimension of data)
Query_c	Float	Q × 128
Dist_c	Float	Q × N_ref
Border_c	Float	Q × 2
Hist_c2	Int	Q × K × 64
Hist_c	Int	Q × K

ماکسیمم اندازه این متغیر ها مربوط به آرایه ref_c است که سایز آن مطابق زی به دست می آید:

$$size(ref - c) = 10^6 \times 2^7 \times 4(\text{sizeof(float)}) \approx 2^{29}$$

که از سایز GPU کمتر است. سایز سایر متغیر ها هم مرتبه کمتری از ۲ هستند پس می توانیم آن ها را در حافظه GPU جا کنیم.

برای پیاده سازی کرنل ها ما Q را برابر با ۳۲ در نظر گرفتیم. با توجه به اینکه تعداد queryها (۱۰۰۰۰) بر ۳۲ بخش پذیر نیست مجبوریم در مرحله آخر ۱۶ query را به کرنل ها بدهیم. پس ۳۱۳ بار باید کرنل ها را صدا بزنیم. در ۳۱۲ دفعه اول کرنل ها با ۳۲ query کار میکنند و در دفعه ۳۱۳ ام با ۱۶ query.

حال به جزییات پیاده سازی توابع می پردازیم:

۱) Distcal : در این تابع برای این که محاسبات سریعتر انجام شود و دسترسی به حافظه گلوبال کمتر شود، همه ی query ها و تعدادی از رفرنس ها را در shared memory هر بلاک وارد می کنیم و هر بلاک را مسئول محاسبه فاصله بین ۳۲ query و ۶۴ رفرنس می کنیم. به این ترتیب کرنل را با بلاک های 32 × 32 و grid های 512 × 512 لانچ می کنیم.هر ترد در بلاک در ابتدا ۴ مختصات از یک query و ۴ مختصات از دو رفرنس که مختصات آن ها توسط tx مشخص می شود را در متغیر q و r حافظه shared لود می کند. بعد از اتمام کپی شدن دیتاها، هر ترد مسئول محاسبه دو فاصله بین query و رفرنس ها می شود. اندیس این نقاط توسط qnum، rnum1، rnum2 تعیین می شوند.

تنها نکته ای که باید به آن توجه کنیم در مرحله آخر که ۱۶ query بررسی می کنیم، این تابع باید به جای ۳۲ query در هر بلاک ۱۶ تا کپی کنیم و نیاز به ۱۶ ترد به جای ۳۲ ترد داریم. به همین دلیل برای این حالت با اعمال این تغییرات تابع distcal2 را صدا می کنیم.

```
74  //------------------------------------------------------------------------------
75  __global__ void caldist(float* query_c, float* ref_c, float* dist_c){
76
77      __shared__ float q[128][32];
78      __shared__ float r[128][64];
79
80      int qnum = ty;
81      int rnum1 = 2 * tx;
82      int rnum2 = 2 * tx + 1;
83      int off1 = 2 * tx * 128 + 4 * ty + (by * 125 + bx) * 128 * 64;
84
85      float d1, d2, temp1 = 0, temp2 = 0;
86
87      for(int k = 0; k < 4; k++)
88          q[4 * ty + k][tx] = query_c[tx * 128 + 4 * ty + k];
89
90      for(int k = 0; k<4; k++){
91          r[4 * ty + k][2 * tx]     = ref_c[off1 + k];
92          r[4 * ty + k][2 * tx + 1] = ref_c[off1 + 128 + k];
93      }
94
95      __syncthreads();
96
97      for(int index = 0; index < 128; index++){
98          d1 = q[index][qnum] - r[index][rnum1];
99          d2 = q[index][qnum] - r[index][rnum2];
100         temp1 = temp1 + d1 * d1;
101         temp2 = temp2 + d2 * d2;
102     }
103
104     dist_c[qnum * 1000000 + 64*((by * 125 )+ bx) + rnum1] = sqrt(temp1);
105     dist_c[qnum * 1000000 + 64*((by * 125 )+ bx) + rnum2] = sqrt(temp2);
106
107 }
108
```

۲) Minmax : همانطور که توضیح داده شد در این کرنل ۶۴ ترد و ۳۲ بلاک لانچ می کنیم. (در مرحله آخر به جای ۳۲ بلاک، ۱۶ بلاک ایجاد می کنیم.) و با ۶۴ تکه کردن هر ۱۰۰۰۰۰۰ فاصله مربوط به یک کوئری پیدا کردن مینیمم و ماکسیمم را در دو مرحله انجام می دهیم. ۶۴ عدد مناسبی برای این کار است چون بزرگترین توان ۲ موجود در ۱۰۰۰۰۰۰ است پس تمام بلاک به صورت متقارن و با توان دو ترد اجرا می شوند. پس نسبی به sm ها بهینه تر است.

```
//------------------------------------------------------------------------------
__global__ void minmax(float* border_c, float* dist_c){ // 32(16) block 64 thread

    __shared__ float min[64];
    __shared__ float max[64];

    int i = tx;
    int offset = bx * 1000000;

    min[i] = dist_c[offset + i * 15625];
    max[i] = dist_c[offset + i * 15625];

    for(int j=1; j<15625; j++){
        if(min[i] > dist_c[offset + i * 15625 + j])
           min[i] = dist_c[offset + i * 15625 + j];
        if(max[i] < dist_c[offset + i * 15625 + j])
           max[i] = dist_c[offset + i * 15625 + j];
    }
    __syncthreads();


    if(i==0){
        float min1 = min[0];
        float max1 = max[0];

        for(int j=1; j<64; j++){
            if(min[j] < min1)
                min1 = min[j];
            if(max[j] > max1)
                max1 = max[j];
        }

        border_c[2 * bx] = min1;
        border_c[2 * bx + 1] = max1;
    }

}
```

۳) Histcal : با توجه به توضیحات اولیه این کرنل را با ۶۴ ترد و ۳۲ (۱۶ در مرحله آخر) بلاک لانچ می کنیم. برای این که تشخیص دهیم هر فاصله در کدام بین از هیستوگرام قرار می گیرد ابتدا با توجه به مینیمم و ماکسیمم محاسبه شده در بخش قبل طول هر بین را محاسبه می کنیم و با تقسیم کردن میزان اختلاف فاصله مورد نظر از مینیمم فاصله بر طول بین و در نظر گرفتن جز صحیح آن شماره بین مورد نظر را می یابیم. این عملیات در خط ۱۸۸ و ۲۰۲ مشخص شده است. دقت کنید که در کد نوشته شده به جای محاسبه طول بین، عکس طول بین محاسبه شده است و به جای عملیات تقسیم از عملیات ضرب استفاده شده است. در این حالت سرعت اجرا برنامه سریعتر خواهد بود:

```
182  //------------------------------------------------------------------------------------
183  __global__ void histcal(float* border_c, float* dist_c,int* hist_c2, unsigned int K){ // 32(16) block 64 thread
184
185      int length = 15625;
186      float min = border_c[2*bx];
187      float max = border_c[2*bx+1];
188      float bin_ilength = K/(max - min);
189      int index;
190      int index2;
191      int mul = bx*1000000 + tx*length;
192      int mul2 = 64*K*bx + K*tx;
193
194      for(int j=0; j<K; j++){
195          hist_c2[mul2 +j] = 0;
196      }
197
198      __syncthreads();
199
200      for(int i=0; i<length; i++){
201          index = mul + i;
202          index2 = (dist_c[index]-min) * bin_ilength;
203          if(index2 == K) index2=K-1;
204              atomicAdd(&hist_c2[mul2 +index2],1);
205
206      }
207  }
```

۴) Adder : در این تابع که تابع نهایی است، ۶۴ هیستوگرام تولی شده برای هر query را با هم جمع می کنیم.

```
208  //------------------------------------------------------------------------------------
209  __global__ void adder(int* hist_c, int* hist_c2, unsigned int K){ // K block 32 thread
210
211      hist_c[bx + tx*K] = 0;
212
213      for(int i = 0; i < 64; i++){
214          hist_c[bx + tx * K] = hist_c2[64 * K * tx + i * K + bx] + hist_c[tx * K + bx];
215      }
216
217  }
```

و برای ۳۲ query مورد نظر هیستوگرام محاسبه می شود. نتایج محاسبات را در متغیر hist در CPU کپی می کنیم و به سراغ query های باقیمانده می رویم.

نتیجه نهایی

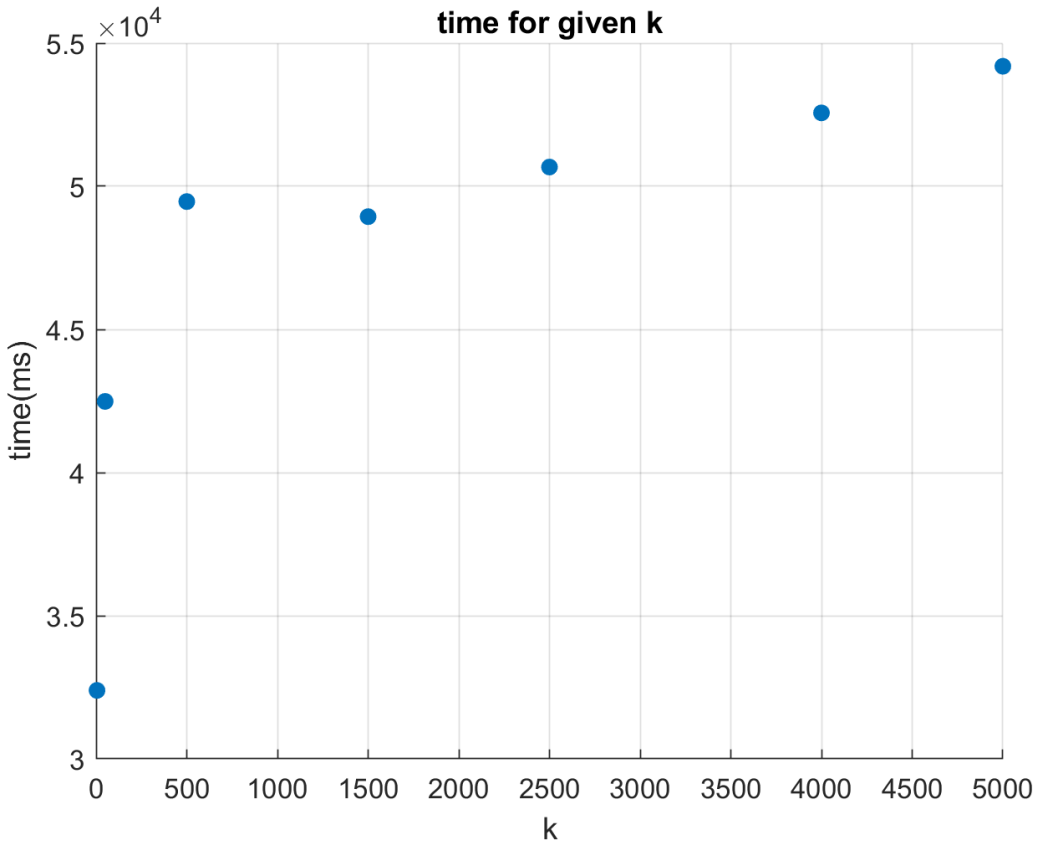
کد را برای K=5 اجرا می کنیم و با پاسخ صحیح مقایسه می کنیم تا از صحت الگوریتم استفاده شده مطمئن شویم. با توجه به اینکه در الگوریتم پیاده شده K فقط در تعداد بلاک ها تاثیر گذار است، می توان نتیجه گرفت در حالات دیگر هم الگوریتم به شکل صحیح اجرا می شود.

پس اجرای کد و مقایسه ۱۰۰۰۰ هیستوگرام به دست آمده، در ۵۷۷ نقطه اختلاف جزیی مشاهده می شود. این اختلاف می تواند ناشی از خطای جزیی در محاسبه مرزها در به وجود آمده باشد. (با توجه به اینکه عملیات floating point دقت کافی را ندارد.) و می توانیم از این خطا صرف نظر کنیم.

در این حالت زمان اجرا کد حدودا برابر با ۳۲ ثانیه است.

در حالت K=5000، زمان در حدود ۵۳ ثانیه است.

در انتها با بررسی زمان برای چند K متوجه میشویم که الگوریتم به K های کوچیک وابستگی بیشتری دارد:



(گزارش کار، گزارش کار های انجام شده تا این لحظه است و ممکن است کد نهایی با تغییراتی جزیی بهبود یابد و زمان ها اندکی بهتر شود.)