دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

گزارش تمرین سری اول پردازش موازی

عنوان:

پیاده‌سازی چند الگوریتم موازی با پلتفرم **CUDA**

نگارش:

ایمان قدیمی دیلمی

۹۹۲۱۰۷۴۲

استاد درس:

دکتر محمد قدسی

ارائه‌شده به:

جناب آقای مهران معینی جم

جناب آقای محمدحسین خوشه‌چین

۱۳ فروردین ۱۴۰۱

مقدمه

آیا پردازنده‌های گرافیکی سریع‌تر از پردازنده‌های مرکزی هستند؟ این یک سوال بسیار مهم است، اما پاسخ آن همیشه ساده نیست. در واقع برای اکثر محاسبات با هدف عمومی، یک CPU بسیار بهتر از یک GPU عمل می‌کند. دلیلش این است که CPU ها با هسته‌های پردازنده کمتری طراحی شده‌اند که سرعت کلاک بالاتری نسبت به هسته‌های گرافیکی دارند و به آنها اجازه می‌دهد تا یک سری کارها را خیلی سریع انجام دهند. از سوی دیگر، پردازنده‌های گرافیکی تعداد هسته‌های بسیار بیشتری دارند و برای اهداف متفاوتی طراحی شده‌اند. در ابتدا، GPU در اصل برای تسریع عملکرد رندر گرافیکی طراحی شده بود. این کار را میتوان با اجازه دادن به CPU برای تخلیه محاسبات سنگین و آزادکردن قدرت پردازش انجام داد. پردازنده‌های گرافیکی به دلیل معماری پردازش موازی، تصاویر را سریع‌تر از یک CPU ارائه می‌کنند که به آن اجازه می‌دهد تا چندین محاسبات را در جریان‌های داده به طور همزمان انجام دهد. CPU مغز عملیات است و مسئول ارائه دستورالعمل به بقیه سیستم است. امروزه، با کمک نرم‌افزارهای اضافی، قابلیت‌های پردازنده‌های گرافیکی افزایش یافته است تا زمان لازم برای تکمیل انواع خاصی از محاسبات مورد نیاز در مراحل مختلف علم داده را به میزان قابل توجهی کاهش دهد. مهم است که بر این نکته تاکید شود که GPU ها جایگزین CPU ها نمی شوند، بلکه به عنوان یک پردازنده مشترک برای تسریع محاسبات علمی و مهندسی عمل می کنند.

CUDA یک پلتفرم محاسباتی موازی و مدل برنامه نویسی است که توسط Nvidia برای محاسبات عمومی بر روی GPUs توسعه یافته است. CUDA به توسعه دهندگان این امکان را می‌دهد تا با استفاده از قدرت پردازنده‌های گرافیکی برای بخش قابل موازی‌سازی محاسبات، برنامه‌های کاربردی محاسباتی را سرعت بخشند.

۱. مرتب‌سازی مرتبه‌ای

الگوریتم مرتب‌سازی رتبه، تعداد کل عناصری را محاسبه می‌کند که کمتر از آن عدد هستند. این مقدار رتبه عنصر نامیده می‌شود و برای محاسبه آن، الگوریتم باید عنصر را با سایر مقادیر موجود در لیست مقایسه کند، بر این اساس است که هر پردازنده، با پیمایش آرایه، تعداد اعداد کوچک تر از خود را شمرده و بر مبنای آن، جایگاه مقدار خود را معین می‌کند و در انتها، آن را در خانه ی مربوطه می‌نویسد.

```
for all i in 0 .. n - 1:
    cnt = 0;
    myValue = array[i];
    for all j in 0 .. n - 1:
        if array[j] < myValue then cnt++;
    array[cnt] = myValue;
```

```
__global__ void rankSort(int * array, int * result, int k) {
  // How many numbers have a block to RANK.
  int a = k / gridDim.x;
  // How many numbers have a thread to COMPARE with the RANK number.
  int b = k / blockDim.x;
  __shared__ int tamBlocks; // Extra (rest) number(s) that lower blocks will have to RANK after.
  __shared__ int tamThreads; // Extra (rest) number(s) that lower threads will have to COMPARE with the actual RANK.
  __shared__ int miNumero; // Number to RANK.
  __shared__ int rank; // Rank (index) accumulation to sort an array position.
  int localRank; // Local thread Rank (index) accumulation (will be sum to the global one at the end of the thread comparissions)
  int comparador; // Number to compare with the RANK (miNumero).
  int range2 = threadIdx.x * b; // Second loop range distribution (comparissions)

  if(threadIdx.x == 0) {
    // Rest of the numbers (indexes) that don't fit with the block distribution to be RANK after with lower blocks IDS
    tamBlocks = k - (a * gridDim.x);
    tamThreads = k - (b * blockDim.x);
  }

  int range1 = blockIdx.x * a; // First loop range distribution (numbers to RANK)
  for(int i = range1; i < range1 + a; i++) {
    if(threadIdx.x == 0) {
      miNumero = array[i]; // We get the RANK number so we will let threads to make their comparissions
      rank = 0; // Initial shared rank
    }
    __syncthreads();

    localRank = 0; // Initial thead local rank
    for(int j = range2; j < range2 + b; j++) {
      comparador = array[j];
      if(comparador < miNumero || (comparador == miNumero && (j < i)))
        localRank += 1; // Local rank accumulation
    }

    // Let the lower threads ID's compute the 'rest' of the comparissions //
    if(threadIdx.x < tamThreads) {
      comparador = array[(blockDim.x * b) + threadIdx.x];
      if(comparador < miNumero || (comparador == miNumero && (((blockDim.x * b) + threadIdx.x) < i)))
        localRank += 1; // Local rank accumulation
    }
```

```
    atomicAdd(&rank, localRank); // Atomic shared rank accumulation

    __syncthreads();

    if(threadIdx.x == 0) {
      result[rank] = miNumero; // Placing the number in its sorted position
    }

    __syncthreads();
  }
  // Let the lower blocks ID's compute the 'rest' of the RANKS //
  if(blockIdx.x < tamBlocks) {
    if(threadIdx.x == 0) {
      miNumero = array[gridDim.x * a  + blockIdx.x];
      rank = 0;
    }
    __syncthreads();
    localRank = 0;
    for(int j = range2; j < range2 + b; j++) {
      comparador = array[j];
      if(comparador < miNumero || (comparador == miNumero && (j < (gridDim.x * a + blockIdx.x))))
        localRank += 1; // Local rank accumulation
    }

    // Let the lower threads ID's compute the 'rest' of the comparissions //
    if(threadIdx.x < tamThreads) {
      comparador = array[(blockDim.x * b) + threadIdx.x];
      if(comparador < miNumero || (comparador == miNumero && (((blockDim.x * b) + threadIdx.x) < gridDim.x * a  + blockIdx.x)))
        localRank += 1; // Local rank accumulation
    }
    atomicAdd(&rank, localRank); // Atomic shared rank accumulation
    __syncthreads();
    if(threadIdx.x == 0) {
      result[rank] = miNumero; // Placing the number in its sorted position
    }
  }
}
```

۲. مرتب‌سازی جابه‌جایی زوج/فرد

در محاسبات، مرتب‌سازی زوج و فرد یا مرتب‌سازی جابه‌جایی زوج و فرد، یک الگوریتم مرتب‌سازی نسبتاً ساده است که در اصل برای استفاده در پردازنده‌های موازی با اتصالات محلی ایجاد شده است. این یک مرتب‌سازی مقایسه‌ای مشابه با مرتب‌سازی حبابی است که با آن ویژگی‌های مشترک زیادی دارد. با اجرای الگوریتم مرتب‌سازی جابه‌جایی زوج/فرد، می توان اعداد را به شکل مرتب‌شده بر روی پردازنده‌ها قرار داد. اگر تعداد پردازنده‌های ما برابرp باشد آنگاه این الگوریتم به p گام نیاز دارد تا خروجی مدنظر را حاصل کند. الگوریتم بدین صورت است که در گام‌های فرد، پردازنده هایی

که شماره‌ی فرد دارند، مقدارشان را با پردازنده‌ی شماره‌ی زوج سمت راستی شان مقایسه می‌کنند. اگر مقدار آن‌ها خارج از ترتیب باشد، مقدارشان را با یکدیگر جابه‌جا می‌کنند. در گام‌های زوج نیز پردازنده‌های شماره‌ی زوج همین کار را به طور مشابه با پردازنده‌ی سمت راست خود انجام می‌دهند. همانطور که واضح است، این الگوریتم بر روی معماری آرایه‌ای قابل اجراست. حال ما قصد داریم که نسخه‌ای از این الگوریتم را بر روی معماری حافظه-مشترک پیاده‌سازی کنیم.

```
__global__ void oddeven(int* x,int I,int n)
{
  int id=blockIdx.x;
  if(I==0 && ((id*2+1)< n)){
    if(x[id*2]>x[id*2+1]){
      int X=x[id*2];
      x[id*2]=x[id*2+1];
      x[id*2+1]=X;
    }
  }
  if(I==1 && ((id*2+2)< n)){
    if(x[id*2+1]>x[id*2+2]){
      int X=x[id*2+1];
      x[id*2+1]=x[id*2+2];
      x[id*2+2]=X;
    }
  }
}
```

۳. جمع پیشوندی بهبودیافته

در مسئله ی جمع پیشوندی، هر خانه‌ی آرایه به یک پردازنده منسوب و دارای یک مقدار اولیه است و پس از اجرای الگوریتم، مقدار هر پردازنده باید جمع مقادیر همه ی پردازنده‌های با اندیس کوچک‌تر از خودش باشد. در نسخه ی ساده‌ی الگوریتم که در کلاس مورد بررسی قرار گرفت، پردازنده ی i بار هر فاز k بار ،مقدار خود را با مقدار پردازنده‌ی $2k+i$ با جمع می‌کند و در خانه‌ی مربوط به آن پردازنده در آرایه می‌نویسد. در الگوریتم بهبودیافته ای که شما باید آن را پیاده‌سازی کنید، فازها مطابق با شکل زیر، به دو بخش صعودی و نزولی تقسیم می شوند که در بخش نخست، تعداد

پردازنده‌های فعال هر بار نصف و در بخش دوم، در هر فاز دو برابر می شوند.

ما می خواهیم الگوریتمی پیدا کنیم که به کارایی الگوریتم ترتیبی نزدیک شود، در حالی که همچنان از موازی‌بودن در GPU استفاده می‌کند. هدف ما در این بخش ایجاد یک الگوریتم اسکن کارآمد برای CUDA است که از فاکتور اضافی $logn$ کار انجام شده توسط الگوریتم کلاسیک اجتناب کند. این الگوریتم بر اساس الگوریتم ارائه شده توسط Blelloch (1990) است. برای انجام این کار از یک الگوی الگوریتمی استفاده خواهیم کرد که اغلب در محاسبات موازی ایجاد می‌شود: درختان متعادل. ایده این است که یک درخت باینری متعادل روی داده‌های ورودی بسازیم و آن را به ریشه و از ریشه برای محاسبه مجموع پیشوندها جابجا کنیم. یک درخت باینری با n برگ دارای $d = logn$ سطح است و هر سطح d دارای $2d$ گره است. اگر به ازای هر گره یک جمع انجام دهیم، در یک پیمایش درخت، $O(n)$ تا جمع را انجام خواهیم داد.

درختی که ما می‌سازیم یک ساختار داده واقعی نیست، بلکه مفهومی است که برای تعیین اینکه هر رشته در هر مرحله از پیمایش چه کاری انجام می‌دهد استفاده می‌کنیم. در این الگوریتم اسکن کارآمد، عملیات را در جای خود روی یک آرایه در حافظه مشترک انجام می‌دهیم. این الگوریتم از دو فاز تشکیل شده است: فاز کاهش (همچنین به عنوان فاز رفت و برگشت شناخته می شود) و فاز پایین رفت و برگشت. همانطور که در Figure 3-39 نشان داده شده است، در مرحله کاهش، درخت را از برگ‌ها به ریشه پیمایش و جمع‌های میانی را در گره های داخلی درخت محاسبه می کنیم. این به عنوان کاهش موازی نیز شناخته می شود، زیرا پس از این مرحله، گره ریشه (آخرین گره در آرایه) مجموع تمام گره های آرایه را نگه می‌دارد. شبه کد برای فاز کاهش در الگوریتم 3 آورده شده است.

الگوریتم ۳:

```
for d = 0 to log2 n – 1 do :
  for all k = 0 to n – 1 by 2 d+1 in parallel do:
    x[k + 2 d+1 – 1] = x[k + 2 d – 1] + x[k + 2 d +1 – 1]
```

.

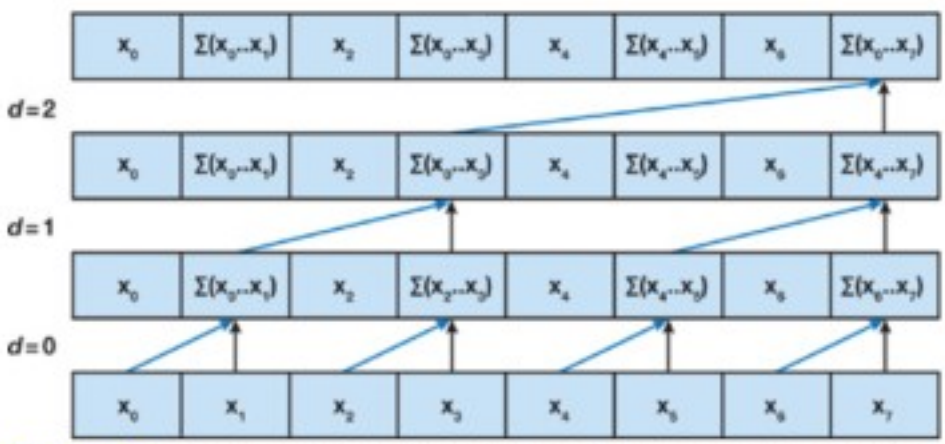

Figure 39-3 An Illustration of the Up-Sweep, or Reduce, Phase of a Work-Efficient Sum Scan Algorithm

در مرحله پایین رفت و برگشت، از ریشه درخت را به سمت پایین پیمایش می‌کنیم و با استفاده از جمع‌های میانی فاز کاهش، اسکن را در جای خود روی آرایه ایجاد می‌کنیم. با واردکردن صفر در ریشه درخت شروع می‌کنیم و در هر مرحله، هر گره در سطح فعلی مقدار خود را به فرزند چپ خود و مجموع مقدار خود و مقدار قبلی فرزند چپ خود را به فرزند راست خود منتقل می‌کند. . رو به پایین در شکل Figure 39-4 نشان داده شده است و شبه کد در الگوریتم 4 آورده شده است.

الگوریتم ۴:

for d = log2 n – 1 down to 0 do:

    for all k = 0 to n – 1 by 2 d +1 in parallel do:

        t = x[k + 2 d – 1]

        x[k + 2 d – 1] = x[k + 2 d +1 – 1]

        x[k + 2 d +1 – 1] = t + x[k + 2 d +1 – 1]

.

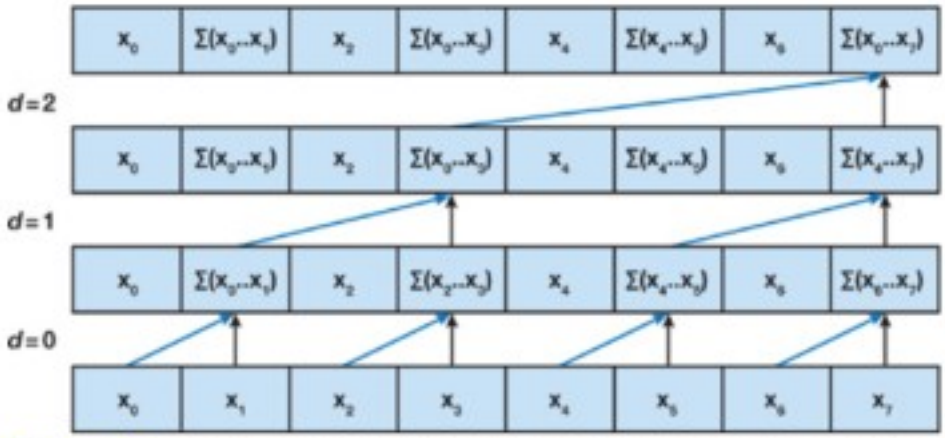

Figure 39-3 An Illustration of the Up-Sweep, or Reduce, Phase of a Work-Efficient Sum Scan Algorithm

```
__global__ void optimizedPrefixSum(float *out_data, float *in_data, int n) {

    extern __shared__ float temp[];

      // allocated on invocation

      int thid = threadIdx.x;
      int offset = 1;
      temp[2*thid] = in_data[2*thid];

      // load input into shared memory

      temp[2*thid+1] = in_data[2*thid+1];
      for (int d = n>>1; d > 0; d >>= 1)

        // build sum in place up the tree
            {
                __syncthreads();
             if (thid < d){
                 int ai = offset*(2*thid+1)-1;
                 int bi = offset*(2*thid+2)-1;
                 float t = temp[ai];
                 temp[ai] = temp[bi];
                 temp[bi] += t;
                  }
             }
              __syncthreads();

      out_data[2*thid] = temp[2*thid];

      // write results to device memory

      out_data[2*thid+1] = temp[2*thid+1];

      }
```