Particle simulation

Using CUDA

Dr. khunjush

Morteza sadeghi 9130202

توضیح

هدف این تمرین بهینه سازی اجرای برنامه particle simulation روی gpu است. ابتدا این برنامه را روی cpu اجرا میکنم و پروفایل میکنم، سپس آنرا روی gpu اجرا میکنم و بهینه سازی ها را اعمال میکنیم. برای اجرای بهینه روی gpu من 5 ایده درنظر گرفتم و آنها را بررسی کردم و نتیجه را در گزارش قرار دادم. ایده استفاده از sharing memory بهترین ایده بود.

Gpu سخت افزاری است که قادر است محاسبات موازی را با تعداد زیادی ترد انجام دهد. خوبی آن نسبت به سایر ابزارهای موازی سازی درجه موازی سازی بالایی است که دارد، تقریبا ده هزار ترد در یک gpu میتوانند همزمان کار کنند. برای اجرای برنامه ها روی gpu از کامپایلر cuda استفاده میکنیم که کد را قابل اجرا روی gpu میکند. از آنجا که تعداد زیادی ترد در gpu داریم، نحوه مدیریت آنها برای بدست آوردن حداکثر موازات خیلی مهم است. یک برنامه cuda را میتوان از نظر نحوه اختصاص تردها و نحوه اختصاص حافظه بهینه سازی کرد.

کد اصلی particle simulation از سایت barkely دانلود شده و من روی آن کمی بهینه سازی انجام داده ام. کد این برنامه شامل دو بخش میباشد، ایجاد particle ها و شبیه سازی. شبیه سازی در یک حلقه انجام میشود که دو بخش دارد : تاصیر نیروی ذرات برهم و حرکت دادن آنها. هرکدام از این سه بخش یادشده را میتوان روی gpu انجام داد: ایجاد ذرات و تاثیر نیروها و حرکت دادن شان.

اجرا روی CPU

کد دانلود شده برای هزار ذره، روی cpu در یک سیستم intel dual-core 2.2GHZ و ویندوز 7حدود 21 ثانیه طول میکشد. در این بخش برخی بهینه سازی ها جهت افزایش سرعت را اعمال میکنم.

ویرایش اول

یکی از دلالیل کندی برنامه شاید مقایسه ذره با همه ذرات دیگر بوده که اینهمه فراخوانی از حافظه و مقایسه فاصله ذره با آنها سرعت را کاهش میدهد. اگر میشد هر ذره بداند که با چه ذره هایی نیاز دارد کنش کند، آنوقت سرعت بهتر میشد. یعنی هر ذره بداند که کدام ذره ها از آن فاصله خیلی زیادی دارند و نباید محاسبه شوند. ولی چون ذره ها مدام در حال حرکت اند، پس این دانش نیز باید بصورت داینامیک کسب شود، واین خود یک سربار است، ولی باید آزمایش کنیم و ببینیم که این سربار میصرفد یا نه.

طرح این است که در هر چند iteration، مثلا هر 10 تا، برای هر ذره ای فاصله آنرا با همه ذرات دیگر محاسبه کنیم و ذراتی که فاصله آنها تقریبا نزدیک است را در یک لیست قرار دهیم، سپس در iteration های بعدی فقط ذرات همان لیست را بررسی کنیم. در اینصورت تعداد مراجعات به حافظه خیلی کمتر خواهند شد. و اما تعداد iteration ها و فاصله ای که به "تقریبا نزدیک" تعبیر میشود، دو فاکتوری هستند که با هم مرتبط اند، زیرا یک ذره اگر بتواند تا مثلا بیست iteration بعدی خود را به نزدیکی ذره موردنظر برساند، ولی ما تعداد iteration ها را 30 تا درنظر گرفته باشیم، آنوقت در محاسبه اشتباه شده است. حالا فرض من این است که در هر ده iteration ذره ها به فاصله ده برابر cutoff میرسند. Cutoff حداکثر فاصله ای است که در آن نیرو اعمال میشود. حال باید تاثیر این فرض را در اجرا دید. به دو فاکتوری که در بالا ذکر شد باید فاکتور تعداد ذره ها را هم افزود، چون موجب افزایش تراکم ذره ها میشود و این خود باعث کندی حرکت ذره ها میشود.

برای اینکه ببینیم این روش نتایج واقعی تولید میکنند، اجراها را باهم مقایسه کرده ام، طوریکه نتیجه نهایی شبیه سازی با هر دو روش را با هم مقایسه کرده ام ولی در نهایت آنها باهم برابر نبوده اند!

در کد زیر، تابع compute_force وظیفه محاسبه نیروهای وارده بر ذره ها را دارد، این بخش را بهینه سازی کرده ایم : (در فایل serial_test_edition1.c)

```
31  int *threadworks;
32  int *threadworksl;
33  void compute_forces_gpu(particle_t * particles, int n, int iteration)
34  {
35
36      if(iteration%10 == 0)
37      {
38          for(int tid=0;tid<n;tid++)
39          {
40              int idx = 0;
41              for(int j=0;j<n;j++)
42              {
43                  double dx = particles[tid].x - particles[j].x;
44                  double dy = particles[tid].y - particles[j].y;
45                  double r2 = dx * dx + dy * dy;
46                  if( r2 <= cutoff*cutoff*MaxDistance )
47                  {
48                      threadworks[tid*DENS+idx] = j;
49                      idx ++;
50                      if(idx == DENS)
51                          break;
52                  }
53              }
54              threadworksl[tid] = idx;
55          }
56      }
57
58      for(int tid=0;tid<n;tid++)
59      {
60          particles[tid].ax = particles[tid].ay = 0;
61          for(int j = 0 ; j < threadworksl[tid] ; j++)
62              apply_force_gpu(particles[tid], particles[ threadworks[tid*DENS+j] ]);
63      }
```

بهینه سازی تابع compute_force برای استفاده در cpu.

این تابع در فایل serial.cpp قرار دارد.

خانه i ام از متغیر threadworks شامل اندیس ذره هایی است که فاصله شان با ذره i ام حدود 20 برابر اندازه cutoff است، cutoff شعاع حداکثر اعمال نیرو است، مقدار 20 در متغیر MaxDistance ذخیره شده. آرایه threadworksl تعداد ذره های نزدیک را ذخیره میکند و در خط 61 استفاده میشود، متغیر DENS هم حداکثر ذره های نزدیک که میتوان ذخیره کرد را نشان میدهد، این مقدار را برای یک اجرای 1000 ذره ای ، برابر 50 ذره گرفته ام. طبق آزمایشی که انجام شد، این مقادیر (20و50) برای یک اجرای 1000 ذره ای درست عمل میکنند.

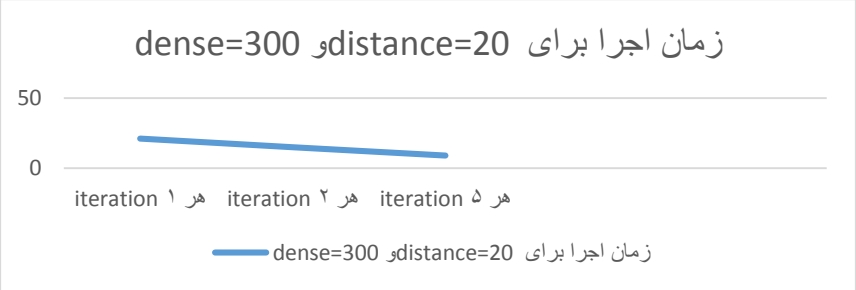
در این کد دو اجرا در تابع main با هم مقایسه میشوند .حلقه ی اول مربوط به اجرای عادی و حلقه دوم مربوط به اجرای بهینه سازی میباشد. در حلقه سوم این دو اجرا با هم مقایسه میشوند. برای 1000 ذره در 1000 گردش، مقادیر مختلف Dens و Maxdistance را امتحان کردم ولی هیچ کدام در جواب شان با

```
123     for( int step = 0; step < NSTEPS; step++ )
124     {
125         compute_forces(particles, n, step);
126         move(particles, n, size);
127     }
128     for( int step = 0; step < NSTEPS; step++ )
129     {
130         compute_forces_2(particles2, n, step);
131         move(particles2, n, size);
132     }
133     for(int i=0; i<n; i++)
134     {
135         if( particles[i]->x != particles2[i]->x ||
136             particles[i]->y != particles2[i]->y)
137         {
138             printf("\nare not the same!\n");
139             break;
140         }
141     }
```

هم مشابه نبودند، در حالی که باید باهم برابر باشند! علت در double بودن اعداد است، من این مورد را در کد زیر امتحان کرده ام، در این کد اعضای آرایه particles را کمی جابجا کردم و درنتیجه نتیجه نهایی محاسبات تغییر کرد! پس مشکل در double بودن اعداد است. بنابراین نمیتوان به نتایج این بهینه سازی اعتماد کرد!

```
113 particle_t *particles2 = (particle_t*) malloc( n * sizeof(particle_t) );
114 memcpy(particles2, &particles[500], 500 * sizeof(particle_t));
115 memcpy(&particles2[500], particles, 500 * sizeof(particle_t));
```

برای تنظیم مقادیر dens و maxdistance ، ابتدا یک مقدار maxdistance را درنظر میگیریم سپس بررسی میکنیم حداکثر ذرات نزدیک به چنین فاصله ای چقدر میشود، سپس آنرا بعنوان dens میگیریم. طبق آزمایشی که من کردم، برای مقدار maxdistance = 20 مقدار dens مناسب 200 خواهد بود. من اینجا کمترین تعداد گردش لازم برای ذخیره ذره ها را درنظر میگیرم : 2 iteration . برای این مقدار سرعت اجرا از 21 ثانیه به 15 ثانیه کاهش میابد و برای 5 گردش سرعت به 9 ثانیه میرسد.



زمان اجرا برای بهینه سازی نوع اول – در این روش در هر n گردش، داده هایی که احتمال نزدیکی شان زیاد است جمع آوری میشوند.

اجرا روی gpu

در این بخش کد دانلود شده را برای اجرا روی gpu بهینه سازی میکنم. در این بخش زیاد نمیتوان روی اندازه و ابعاد بلاک و و ترد مانور داد، زیرا سرعت اجرا تغییر چندان محسوسی نمیکند. سرعت اجرا در این مرحله نسبت به cpu حدود 60 برابر شده است (روی ماشین wave) و از 21 ثانیه به 0.36 ثانیه رسیده و تغییر بیشتر ممکن نیست. شاید اگر اندازه داده خیلی بزرگ میبود میشد تغییری اعمال کرد.

کد دانلود شده

در کد دانلود شده، تابع محاسبه نیرو اینگونه تعریف شده :

```
35  __global__ void compute_forces_gpu(particle_t * particles, int n)
36  {
37    // Get thread (particle) ID
38    int tid = threadIdx.x + blockIdx.x * blockDim.x;
39    if(tid >= n) return;
40
41    particles[tid].ax = particles[tid].ay = 0;
42    for(int j = 0 ; j < n ; j++)
43      apply_force_gpu(particles[tid], particles[j]);
44
45  }
```

تابع compute_force در کد دانلود شده در فایل gpu.cu

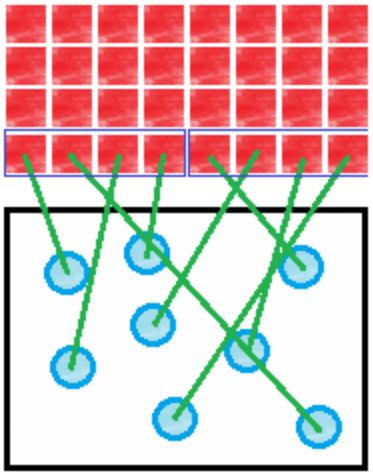
این تابع روی gpu اجرا میشود، فراخوانی آن در تابع main در یک حلقه بصورت زیر است :

```
13      for( int step = 0; step < NSTEPS; step++ )
14      {
15          //
16          //  compute forces
17          //
18
19      int blks = (n + NUM_THREADS - 1) / NUM_THREADS;
20      compute_forces_gpu <<< blks, NUM_THREADS >>> (d_particles, n);
21
22          //
23          //  move particles
24          //
25      move_gpu <<< blks, NUM_THREADS >>> (d_particles, n, size);
26
27      }
```

تابع main در کد دانلود شده، در فایل gpu.cu

در این پیاده سازی، تعدادی بلاک هرکدام با اندازه NUM_THREADS ترد به کد اختصاص داده میشوند. گرید شامل این بلاک ها، یک بعدی است و بلاک ها هم باتوجه به کد تابع compute_force_gpu یک بعدی هستند :

```
38    int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

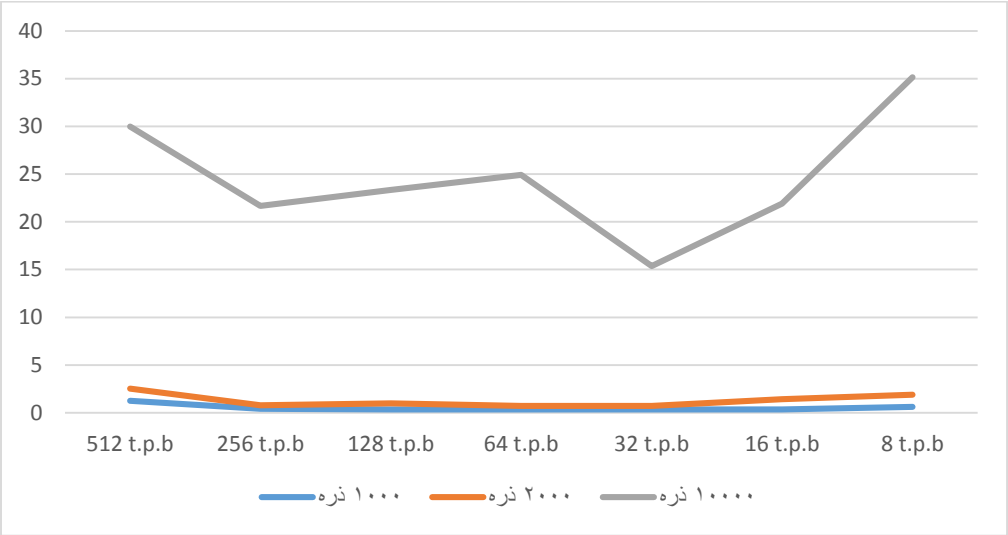


پس نحوه کار بدین صورت میشود که تردها یکی یکی به ذرات اختصاص داده میشوند و کار هر ذره بر عهده یک ترد است. بخش مهم آنجاست که هر ذره باید با همه ذرات دیگر کنش داشته باشد، یعنی هر ترد باید با همه ذرات دیگر ارتباط برقرار کند، و اولا لازم نیست با همه ذرات ارتباط داشته باشد چون تعداد زیادی شان در فاصله دوری از این ذره قرار دارند و دوما درخواست های تردهای مختلف پراکنده است و این دو مورد اتلاف زمان بوجود میاورد. این دو مورد هم قابل حل نیستند، مگر با ابزارهای خودکار که این بهینه سازی را انجام دهند و مدام چینش داده ها در حافظه را تغییر دهند.

زمان اجرا برای کد دانلود شده

در اینجا یک متغیر اصلی در کد داریم و آن تعداد تردها در بلاک است، اول این متغیر را امتحان میکنیم. سپس ازنظر تعداد ذرات آنرا بررسی میکنیم، برای هزار ذره 0.36 ثانیه طول میکشد، که حدود 60 برابر از اجرا روی cpu سریع تر است! سپس زمان اجرا برای بخش های مختلف برنامه را بررسی میکنیم.

تاثیر تعداد ذرات و اندازه بلاک در زمان اجرا



سرعت اجرا برای تعداد ذرات مختلف با اندازه بلاک های مختلف

با دقت در دیتای این جدول میبینیم که تعیین اندازه بلاک، برای تعداد ذرات مختلف فرق میکند، مثلا برای 1000 ذره، 64 ترد در بلاک بهترین اندازه بود ولی برای 2000 ذره، 256 ترد در بلاک بهترین است.

	1000 ذره	2000 ذره	10000 ذره
512 t.p.b	1.25	2.53	30
256 t.p.b	0.4	0.77	21.66
128 t.p.b	0.36	0.98	23.34
64 t.p.b	0.36	0.72	24.92
32 t.p.b	0.35	0.71	15.36
16 t.p.b	0.35	1.42	21.9
8 t.p.b	0.62	1.9	35.16

پس نتیجه میشود برای اندازه ذرات مختلف باید مقدار مناسب اندازه بلاک انتخاب شود. با دقت در جدول بالا میبینیم که برای هرکدام از تعداد ذرات، دو بار افت در سرعت و سپس افزایش تدریجی آنرا داریم، که علت این مساله را نمیدانم.

تاثیر بخش های مختلف برنامه بر زمان اجرا

در نرم افزار visual studio 2012 برنامه را بکمک ابزار performance analysis tool بررسی میکنیم تا ببینم کدام بخش برنامه نیاز به بهینه سازی دارد. در شکل زیر درصد اشتغال cpu به توابع نشان داده شده که تابع apply-force اشتغال بیشتری بوجود میاورد و آن بعلت ضرب متغیرها میباشد :

Functions Doing Most Individual Work

Name	Exclusive Samples %
apply_force_gpu	56.73
compute_forces2	37.79
move	3.02
@ILT+250(?apply_force_gpu@@YAXAAUparticle_t@@0@Z)	1.02
@ILT+355(__RTC_CheckEsp)	0.92

گزارش vs2012 از درصد اشتغال cpu به توابع برنامه serial.cpp

در داخل تابع apply_force بخشی که داده ا از حافظه لود میکند و بخشی که عمل ضرب را انجام میدهد زمان زیادی را صرف میکند :

```
        void apply_force_gpu(particle_t &particle, particle_t &neighbor)
 33.6 %{
  8.9 %  double dx = neighbor.x - particle.x;
  1.2 %  double dy = neighbor.y - particle.y;
  6.9 %  double r2 = dx * dx + dy * dy;
  3.3 %  if( r2 > cutoff*cutoff )
  0.9 %      return;
         //r2 = fmax( r2, min_r*min_r );
  0.1 %  r2 = (r2 > min_r*min_r) ? r2 : min_r*min_r;
< 0.1 %  double r = sqrt( r2 );

  0.3 %  double coef = ( 1 - cutoff / r ) / r2 / mass;
< 0.1 %  particle.ax += coef * dx;
         particle.ay += coef * dy;


  2.8 %}
```

گزارش vs2012 از تابع apply_force در فایل serial.cpp

درتابع compute_force نیز فراخوانی تابع apply_force زمان برترین بخش آن است که خب این طبیعی است :

```
< 0.1 %     for(int tid=0;tid<n;tid++)
            {
                particles[tid].ax = particles[tid].ay = 0;
  0.8 %         for(int j = 0 ; j < threadworksl[tid] ; j++)
 61.8 %             apply_force_gpu(particles[tid], particles[ threadworks[tid*DENS+j] ]);
            }
```

گزارش vs2012 ازتابع compute_force در فایل serial.cpp

درتابع main نیز دیده میشود که apply_force زمان بیشتری نسبت به move صرف میکند (حدود سه برابر) و نیاز بیشتری به بهینه سازی دارد :

```
        for( int step = 0; step < NSTEPS; step++ )
        {
97.0 %      compute_forces2(particles, n, step);
 3.0 %      move(particles, n, size);
        }
        free( particles );
```

گزارش vs2012 از تابع main در فایل serial.cpp

این گزارش ها در پوشه cpu-codes در فایل particle simulation130728.vsps ذخیره شده اند.

ویرایش اول : تغییر ابعاد بلاک ها

اینجا یک تغییر ساده در کد بوجود میاورم، و بلاک ها را دو بعدی میکنم تا تاثیر آنرا ببینم، اندازه هر بلاک را پیض فرض 16 * 8 ترد میگیرم و تعدادی بلاک اختصاص میدهم تا همه ذرات پوشش داده شوند. پس یک گرید دوبعدی با بلاک های دو بعدی داریم :

```
69  // Get thread (particle) ID
70  //int tid = threadIdx.x + blockIdx.x * blockDim.x;
71  int tid = (blockId.y*gridDim.x+blockId.x)*blockDim.y*blockDim.x +
72    threadId.y*blockDim.x + threadId.x;
73  if(tid >= n) return;
```

نحوه محاسبه id ترد در تابع compute_force .

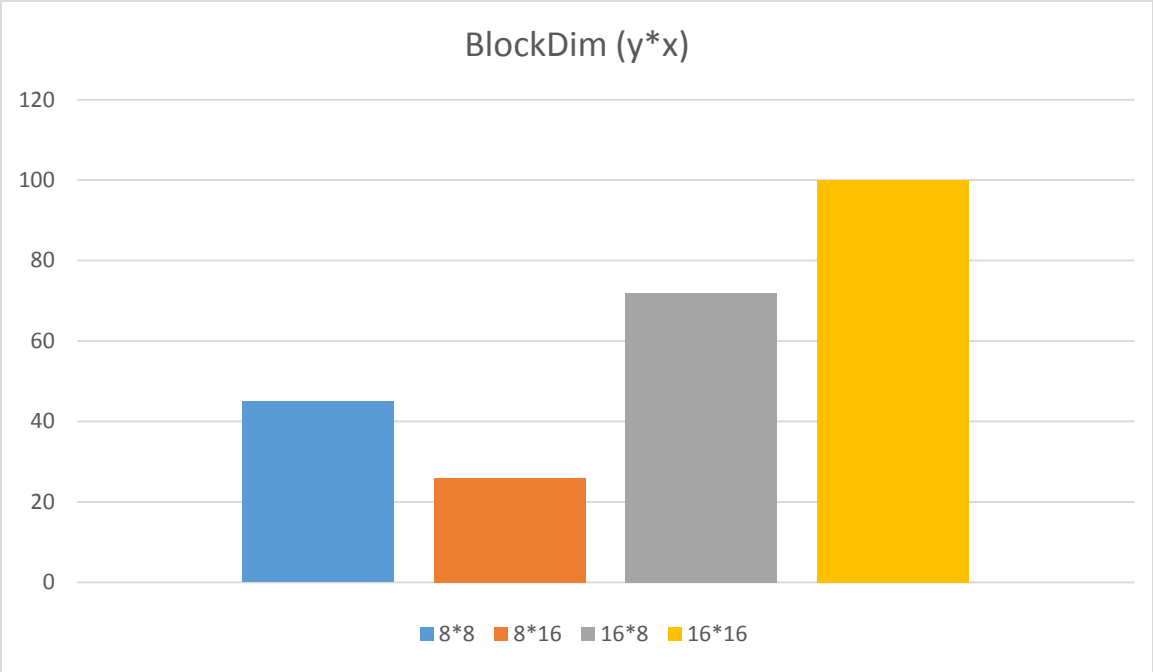
این تابع در فایل gpu1.cu قرار دارد.

زمان اجرا برای ویرایش اول

اینجا متغیر ما ابعاد بلاک است و میتوانیم با آن بازی کنیم تا نتیجه تاثیراتش را ببینیم. در ویرایش قبلی بلاک ها یک بعدی بودند و تردهایشان بترتیب به particle ها اختصاص داده میشدند، ولی اینجا میخواهیم نوعی بی نظمی در این تخصیص دادن ایجاد کنیم و تاثیر آنرا بر سرعت اجرا ببینیم :

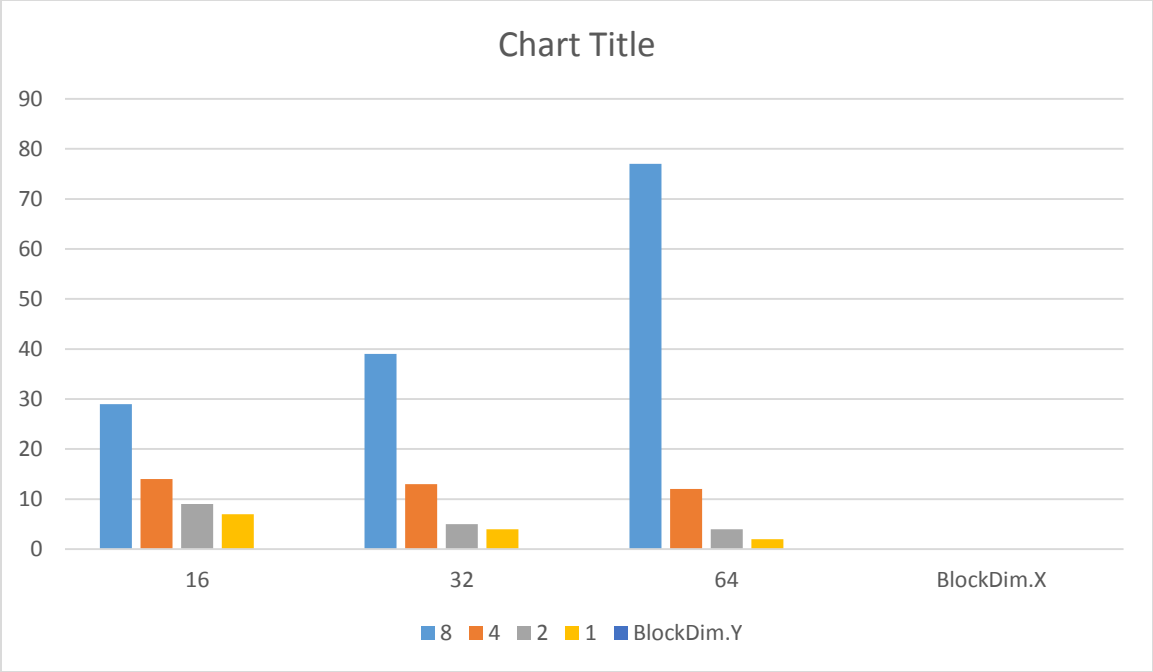
```
9130202@wave:~$ ./gpu1 -n 1000 -bx 16 -by 8
CPU-GPU copy time = 5.5e-05 seconds
n = 1000, simulation time = 26.4257 seconds
9130202@wave:~$ ./gpu1 -n 1000 -bx 8 -by 8
CPU-GPU copy time = 5.6e-05 seconds
n = 1000, simulation time = 45.8355 seconds
9130202@wave:~$ ./gpu1 -n 1000 -bx 16 -by 16
CPU-GPU copy time = 5.6e-05 seconds
n = 1000, simulation time = 100.089 seconds
9130202@wave:~$ ./gpu1 -n 1000 -bx 8 -by 16
CPU-GPU copy time = 5.6e-05 seconds
n = 1000, simulation time = 72.7987 seconds
9130202@wave:~$
```

اجرای ویرایش دوم روی ماشین wave



تاثیر ابعاد بلاک در زمان اجرا (به ثانیه)

آنطور که مشاهده میشود تاثیر خیلی بد بوده طوریکه از اجرا روی cpu هم بیشتر طول کشیده! علت بنظر من فقط یک چیز است : تغییر شکل بلاک ها باعث شده تا تردهای مجاور به خانه های غیرمجاور در حافظه دسترسی داشته باشند و این پراکندگی داده ها باعث کندی زمان اجرا شده. در کد ویرایش نشده، بلاک ها یک بعدی هستند و تردها پشت سر هم هستند و درخواست ها هم بهمین ترتیب مجاور میشوند، و اگر در جدول بالا دقت کنیم، هرچقدر شکل بلاک خوابیده تر باشد، این مجاورت هم بیشتر میشود و سرعت بیشتر میشود، مثلا در ستون دوم که ابعاد y از x کمتر است و تردهای مجاور در یک خط بیشتر هستند، سرعت اجرا نسبت به بقیه بیشتر شده است. برای امتحان درستی این گفته، بلاک ها با اشکال خوابیده را امتحان میکنم :



تاثیر ابعاد بلاک در زمان اجرا (به ثانیه)

رنگ ها مربوط به اندازه بعد Y هستند و اعداد نوشته شده در زیر دسته ستون ها مربوط به اندازه بعد X است.

	8	4	2	1
16	29	14	9	7
32	39	13	5	4
64	77	12	4	2

تاثیر ابعاد بلاک در زمان اجرا (به ثانیه)

همانطور که در این نمودار دیده میشود، هرچقدر اندازه X بیشتر میشود یعنی بلاک بیشتر شکل خوابیده پیدا میکند، کاهش اندازه Y تاثیر بیشتری در افزایش سرعت دارد، پس فرض ما براینکه خطی بودن شکل بلاک در این مساله سرعت بهتری در اجرا دارد، درست است. همچنین هرچقدر اندازه X بیشتر باشد، اندازه Y تاثیر بیشتری در کاهش سرعت دارد، مثلا اگر اندازه X برابر 1 باشد و اندازه Y برابر 16 باشد، زمان اجرا به بیشتر از 100 ثانیه میرسد! علت این را نمیدانم، فقط میدانم شکل خوابیده تر بهتر است.

یک تغییر دیگر هم در این مرحله میشود در کد ایجاد کرد : میتوان تابع compute_force را با بلاک های دوبعدی و move را با بلاک های یک بعدی انجام داد و بالعکس، این کار را انجام دادم، و متوجه شدم اجرای compute_force با بلاک های یک بعدی سرعت را بیشتر افزایش میدهد، یعنی این بخش از کد به موازات بیشتری نیاز دارد.

نتیجه

در این مساله، بلاک ها بهتر است یک بعدی باشند. زیرا particle ها هم در یک آرایه نگهداشته شده اند و قرار است هرکدام با همه مقایسه شوند، برای رفع پراکندگی مقایسات نمیتوان کاری کرد، ولی برای رفع پراکندگی اختصاص ترد به ذره بهتر است که تردها پشت سرهم به خانه های متوالی حافظه تخصیص داده شوند تا سرعت اجرا بهتر شود، یعنی شکل بلاک ها باید شبیه یک بعدی باشد.

ویرایش دوم : موازی سازی بخش اول کد (initialization)

حالا یک تغییر دیگر ایجاد میکنیم و آن کد مربوط به ایجاد ذرات است : init_Particles . این بخش از کد را هم میتوان موازی اجرا کرد زیرا در یک حلقه سعی دارد تا همه ذرات را مقدار دهی اولیه نماید :

```
49      for( int i = 0; i < n; i++ )
50      {
51          //
52          //  make sure particles are not spatially sorted
53          //
54          int j = lrand48()%(n-i);
55          int k = shuffle[j];
56          shuffle[j] = shuffle[n-i-1];
57
58          //
59          //  distribute particles evenly to ensure proper spacing
60          //
61          p[i].x = size*(1.+(k%sx))/(1+sx);
62          p[i].y = size*(1.+(k/sx))/(1+sy);
63
64          //
65          //  assign random velocities within a bound
66          //
67          p[i].vx = drand48()*2-1;
68          p[i].vy = drand48()*2-1;
69      }
70      free( shuffle );
71  }
```

این بخش از کد چون از توابعی همچون rand و srand و lrand48 استفاده کرده قابل اجرا روی gpu نیست.

ویرایش سوم : استفاده از shared memory

حالا بهتر است کل ذرات را از حافظه global به shared بیاورم تا تاثیر آنرا در زمان اجرا ببینم. در حالتی که داده ها روی global قرار داشتند، احتمالا درخواست های همزمان به یک بلاک حافظه زیاد بوده و زمان اجرا را کم میکرد و احتمالا cache دستگاه نتوانسته همه موارد مورد نیاز را هم cache کند، این احتمالات با قرار دادن داده در shared memory بررسی میشود.

باید در هربار فراخوانی تابع compute_force ، یکبار داده ها در داخل shared memory کپی شوند، البته این خود یک سربار اجتناب ناپذیر محسوب میشود که باید دید آیا میصرفد یا خیر. هر ترد به یک particle تخصیص میابد و باید خود را با همه particle های دیگر مقایسه کند، یعنی n بار به global memory مراجعه کند، ولی میتواند به sharing memory مراجعه کند. در کد زیر تابع cpmpute_force برای این منظرو بهینه سازی شده است :

```
35  __global__ void compute_forces_gpu(particle_t * particles, int n)
36  {
37      // copy to shareing memory
38      __shared__ particle_t *localparticles;
39
40      for(int i=0;i<n;i++)
41      {
42          localparticles[i].x = particles[i].x;
43          localparticles[i].y = particles[i].y;
44          localparticles[i].vx = particles[i].vx;
45          localparticles[i].vy = particles[i].vy;
46          localparticles[i].ax = particles[i].ax;
47          localparticles[i].ay = particles[i].ay;
48      }
49
50      // Get thread (particle) ID
51      int tid = threadIdx.x + blockIdx.x * blockDim.x;
52      if(tid >= n) return;
53
54      particles[tid].ax = particles[tid].ay = 0;
55      for(int j = 0 ; j < n ; j++)
56          apply_force_gpu(particles[tid], localparticles[j]);
57  }
```

تابع compute_force که برای استفاده از sharing memory ویرایش شده است. این تابع در فایل gpu3.cu قرار دارد.

زمان اجرا برای ویرایش سوم

همانطور که پیش بینی میشد سرعت اجرا بسیار بهبود یافته، سرعت اجرا برای کد بالا 0.16 ثانیه بوده است که نسبت به کد دانلود شده برای gpu، 36 بار سریعتر و برای کد دانلود شده برای cpu، حدود 2000 بار سریعتر شده است. فکرنمیکنم در کد اشکالی وجود داشته باشد، البته چندین warning هنگام کامپایل داده میشد که مربوط به استفاده از sharing memory میباشد.

نتیجه

پس ما هم به این باور میرسیم که استفاده از sharing memory همیشه و همه جا بسیار بدرد میخورد!

ویرایش چهارم: یک بار اجرای kernel

هر بار که کد در gpu فراخوانی میشود، باید تردها و بلاک ها مدیریت و برنامه ریزی شوند تا کد اجرا شود، بنا بر این فرض اگر کل حلقه را داخل تابع کرنل ببریم و یکبار این فراخوانی را انجام دهیم احتمالا سرعت بهتری خواهیم داشت.

بنابراین کد به این صورت تغییر میکند که در تابع main یکبار قراخوانی داریم :

```
81      double simulation_time = read_timer( );
82      run_simulation <<blks, NUM_THREADS >>> (d_particles, n, size);
83      cudaThreadSynchronize();
84      simulation_time = read_timer( ) - simulation_time;
```

و تابع run_simulation که جایگزین سه تابع قبلی شده به این صورت خواهد بود :

```
11  __global__ void run_simulation (particle_t * particles, int n, double size)
12  {
13
14      for(int step=0; step<n; step++)
15      {
16          // Get thread (particle) ID
17          int tid = threadIdx.x + blockIdx.x * blockDim.x;
18          if(tid >= n) return;
19
20          // compute force
21          particles[tid].ax = particles[tid].ay = 0;
22          for(int j = 0 ; j < n ; j++)
23          {
37
38          // move
39          particle_t * p = &particles[tid];
40          p->vx += p->ax * dt;
41          p->vy += p->ay * dt;
42          p->x  += p->vx * dt;
43          p->y  += p->vy * dt;
44
45          while( p->x < 0 || p->x > size )
46          {
50          while( p->y < 0 || p->y > size )
51          {
55
56          __syncthreads();
57      }
58  }
```

تابع run_simulation که همه توابع را داخل آن باز کرده ام.

این تابع در فایل gpu4.cu قرار دارد.

فقط اینجا نیاز هست که چندبار منتظر اتمام کار تردها بمانیم و تابع __syncthreads() را فراخوانی کنیم، زیرا باید کار یک قدم از شبیه سازی انجام شده باشد تا به قدم بعدی برویم و همچنین باید اول نیروها محاسبه شوند و بعد حرکت انجام شود.

زمان اجرا برای ویرایش چهارم:

زمان اجرا در اینجا برای 128 ترد چیزی باورنکردنی و برابر 4.6e-05 ثانیه بوده، احتمالا باید در کد مشکلی باشد، ولی من مشکلی ندیدم! پس حتما در کد یک مشکلی هست که چنین نتیجه ای میدهد، زیرا استفاده از sharing memory هم چنین نتیجه ای نداشته!

ویرایش پنجم : دوباره چینی داده ها در حافظه

کاری که در ویرایش اول برای کد cpu انجام دادم را اینجا برای gpu هم تکرار میکنم تا نتیجه را ببینم :

```
__global__ void compute_forces_gpu(particle_t * particles, int n, int iteration,
                                   int *threadworks, int *threadworksl)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if(tid >= n) return;

    if(iteration%2 == 0)
    {
        int idx = 0;
        for(int j=0;j<n;j++)
        {
            double dx = particles[tid].x - particles[j].x;
            double dy = particles[tid].y - particles[j].y;
            double r2 = dx * dx + dy * dy;
            if( r2 <= cutoff*cutoff*MaxDistance*MaxDistance )
            {
                threadworks[tid*DENS+idx] = j;
                idx ++;
                if(idx == DENS)
                    break;
            }
        }
        threadworksl[tid] = idx;
    }

    particles[tid].ax = particles[tid].ay = 0;
    for(int j = 0 ; j < threadworksl[tid] ; j++)
        apply_force_gpu(particles[tid], particles[ threadworks[tid*DENS+j] ]);

}
```

تابع compute_force بهینه سازی شده برای دوباره چینی داده ها در حافظه.

این تابع در فایل gpu5.cu قرار دارد.

زمان اجرا برای ویرایش پنجم

زمان اجرا برای بلاک های 218 تردی حدود 0.78 ثانیه بوده است، که نسبت به کد دانلودشده (بدون ویرایش) کندتر میباشد، درحالیکه اینجا هر 5گردش یکبار داده ها را دوباره مرتب میکنیم. علت این کندی استفاده بیشتر از حافظه global میباشد. حال اگر بجای global از shared استفاده کنیم، نباید تغییر محسوسی نسبت به shared در زمان اجرا داشته باشد ولی درعوض پیچیدگی کد بالا میرود.

مقایسه بهینه سازی های انجام شده با هم

در این جدول بهترین زمان هر بهنیه سازی آورده شده تا با هم مقایسه شوند :

v		Grid dim	Block dim	Run time for n=1000 (sec)
0	کد دانلودشده بدون تغییر	1*n	1*128	0.36
1	تغییر ابعاد بلاک	n*(n/64)	1*64	2
2	موازی سازی initialization			
3	Sharing mem	1*n	1*128	0.16
4	سرهم کردن کرنل	1*n	1*128	4.6e-05 !!
5	دوباره چینی داده در حافظه	1*n	1*128	0.78

بهترین تاثیر برای sharing memory بوده و سایر بهینه سازی ها تاثیری در افزایش سرعت نداشته اند.