پیاده‌سازی موازی الگوریتم K-nearest neighbor بر روی GPU با استفاده از CUDA

علی آجریان[1]، حسین باستان[2] و محمدعلی منتظری[3]
[1]دانشجوی کارشناسی ارشد مهندسی معماری کامپیوتر ، دانشکده برق و کامپیوتر ، دانشگاه صنعتی اصفهان ، اصفهان ،
arastoo.ajorian@gmail.com
[2]دانشجوی کارشناسی ارشد مهندسی معماری کامپیوتر ، دانشکده برق و کامپیوتر ، دانشگاه صنعتی اصفهان ، اصفهان ،
hossein.bastan@yahoo.com
[3]استادیار گروه کامپیوتر ، دانشکده برق و کامپیوتر ، دانشگاه صنعتی اصفهان ، اصفهان ،
montazeri_ma@yahoo.com

چکیده – الگوریتم *K-nearest neighbor* یکی از پرکاربردترین الگوریتم‌های کلاس‌بندی است که در زمینه‌های مختلفی مورد استفاده قرار گرفته است. یکی از چالش‌های پیش روی این الگوریتم زمان اجرای نسبتا بالای آن است. برای غلبه بر این موضوع عمدتا فضای مسئله را با نمونه‌گیری از داده‌های خام کوچکتر می‌کنند. این امر باعث می‌شود تا میزان خطای کلاس‌بندی افزایش یابد. بدیهی است که هر گونه راه حلی که سرعت اجرای این الگوریتم را بدون کاهش فضای داده‌ها افزایش دهد یک راه حل بهتر تلقی می‌شود. در این مقاله یک پیاده‌سازی موازی برای این الگوریتم بر روی معماری *CUDA* ارائه شده است که بدون کاهش فضای داده‌های مسئله، به *speed up* ای در حدود ۲۰ نایل می‌شود.

کلید واژه- کارت گرافیکی، GPU، CUDA، Streaming Multiprocessor ، k-nearest neighbors

۱- مقدمه

K-nearest neighbor (KNN) یکی از پرکاربردترین تکنیک‌های کلاس‌بندی[1] است [۱] و در بسیاری از زمینه های تحقیقاتی و صنعتی از جمله رندرینگ اشیاء سه بعدی، بازیابی تصاویر بر اساس محتوا[2]، محاسبات آماری (مانند تخمین آنتروپی و دیورژانس)، بیولوژی (مانند کلاسبندی ژن ها) و ... کاربرد دارد [۲].

در KNN برای کلاس‌بندی یک نقطه‌ی q از نزدیکترین k تا همسایه‌ی آن استفاده می‌شود، بنابراین مسئله‌ی اصلی در این روش پیدا کردن K نقطه‌ی $k_1,k_2,\ldots,k_K$ در مجموعه‌ی نقاط مرجع R است به نحوی که این نقاط کمترین فاصله را با q داشته باشند[2]. در این روش به منظور بهبود نرخ خطای کلاس‌بندی عمدتا اندازه‌ی مجموعه‌ی نقاط مرجع R را بزرگ فرض می‌کنیم[۳] که این امر باعث افزایش شدید حجم محاسبات می‌گردد. بنابراین ذات محاسباتی KNN نیازمند یک پیاده‌سازی با کارایی بالاست [۱]. در این مقاله یک پیاده‌سازی کارآمد KNN با بهره‌گیری از قدرت محاسباتی بالای عناصر پردازشی کارت‌های گرافیکی (GPU) مجهز به CUDA ارائه شده است.

CUDA یک معماری نرم‌افزای- سخت‌افزای برای محاسبات موازی است که کارت‌های گرافیکی NVIDIA را به عناصر پردازشی عام منظوره مبدل می‌کند[۴]. ایده‌ی اصلی در این معماری اجرای قسمت‌های سریال برنامه بر روی CPU و اجرای قسمت‌های موازی آن بر روی GPU است. C، C++، Fortran، OpenCL، DirectCompute و ... زبان‌هایی هستند که امکان برنامه‌نویسی برای معماری CDUA را مهیا می‌کنند[۴].

در ادامه و در قسمت ۲ به بررسی مسئله‌ی KNN می‌پردازیم، سپس در قسمت ۳ CUDA را معرفی می‌کنیم و در قسمت ۴ الگوریتم پیشنهادی برای بهبود کارایی KNN را تشریح می‌کنیم. در نهایت در قسمت ۵ نتایج حاصل از پیاده‌سازی این الگوریتم بررسی می‌شود.

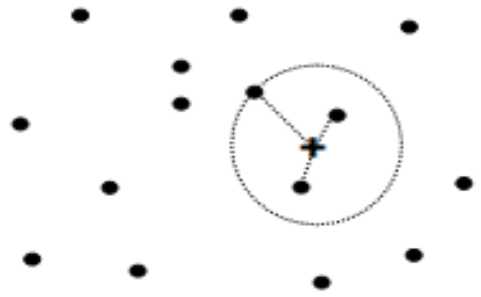
[1] classification
[2] Content-based image retrieval

۲- مسئله‌ی KNN

مجموعه‌ی $R=\{r_1,r_2,\ldots,r_m\}$ که دارای m نقطه‌ی مرجع در یک فضای d بُعدی به نام S است را در نظر می‌گیریم. نقطه‌ی q را نیز در این فضا به عنوان نقطه‌ی جستار[۳] فرض می‌کنیم. هدف از مسئله‌ی KNN تعیین k نقطه از مجموعه‌ی R است به گونه‌ای که این نقاط کمترین فاصله تا نقطه‌ی q را داشته باشند[۲].

در این‌جا فرض می‌کنیم که فضای S یک فضای گسسته است که بسته به حوزه‌ی مسئله می‌تواند از هر جنسی فرض شود. مثلا مجموعه‌ی همه‌ی تصاویر گرفته شده از یک صحنه‌ی خاص و یا مجموعه‌ی تمامی مداخل[۴] یک جدول از پایگاه داده‌ی دانشجویان و یا مجموعه‌ی فاکتورهای خرید مشریان یک فروشگاه و ... هر یک می‌تواند به عنوان فضای S فرض شود. بدیهی است که در هر یک از این موارد تعریف فاصله‌ی دو نقطه از یکدیگر متفاوت خواهد بود.

شکل ۱ یک نمونه از مسئله‌ی KNN را در فضای دو بُعدی برای K=3 نشان می‌دهد که در آن از فاصله‌ی اقلیدسی برای تعیین فاصله‌ی بین دو نقطه استفاده شده است[۲].



شکل ۱: مسئله‌ی KNN در فضای R^2 با K=3 و فاصله‌ی اقلیدسی

یک روش پایه برای محاسبه‌ی KNN روش brute-force است که در آن، کل فضا برای محاسبه‌ی نزدیکترین همسایگی‌ها در نظر گرفته می‌شود [۲]. ویژگی بارز این کار کاهش میزان خطای کلاس‌بندی به قیمت صرف هزینه محاسباتی بیشتر است. البته در پاره‌ای از موارد می‌توان برای کاهش این بار محاسباتی، یک زیر مجموعه‌ی R' از R را به عنوان مجموعه‌ی مرجع در نظر گرفت[۳].

مراحل الگوریتم KNN با روش brute-force به صورت زیر است:

فاصله‌ی بین نقطه‌ی q و تمامی نقاط R را محاسبه کن.

مجموعه‌ی فواصل یافته شده را مرتب کن.

از بین فواصل مرتب شده k عنصر ابتدایی را به عنوان نزدیکترین همسایه‌های q در نظر بگیر.

با دقت در الگوریتم فوق مشخص می‌شود که مرحله‌ی اول این الگوریتم قابلیت موازی سازی بسیار بالایی دارد. زیرا محاسبه‌ی فاصله‌ی هر یک از نقاط R تا نقطه‌ی q به سایر نقاط وابسته نیست. لذا می‌توان با استفاده از یک سخت‌افزار موازی با معماری SIMD[۵] میزان speed up زیادی بدست آورد. همچنین با ارائه‌ی یک الگوریتم مرتب‌سازی موازی کارا می‌توان مرحله‌ی دوم الگوریتم را نیز بر روی چنین سخت‌افزاری پیاده‌سازی کرد.

۳- معرفی معماری CUDA

GPU همواره به عنوان یک پردازنده با منابع محاسباتی بسیار زیاد مورد توجه بوده است[۷]. تعداد هسته‌های فراوان این پردازنده و همچنین پهنای باند زیاد حافظه‌ی کارت‌های گرافیکی قابلیت اجرای تعداد نخ‌های موازی بسیار زیادی را در مقایسه با CPU به این پردازنده داده است.

به طور کلی ساختار GPU به گونه‌ای است که با پردازش دسته-ای داده‌ها بازده‌ی[۶] محاسبات را افزایش می‌دهد، حال آنکه CPU اجرای یک دستور را تا حد امکان سرعت می بخشد [۶]. بعلاوه GPU در یک معماری SIMD اجرای تعداد زیادی نخ با وزن سبک را ممکن می‌سازد ولی معماری MIMD مربوط به CPU نخ‌های با وزن بیشتر و البته تعداد کمتر را فراهم می‌کند [۶].

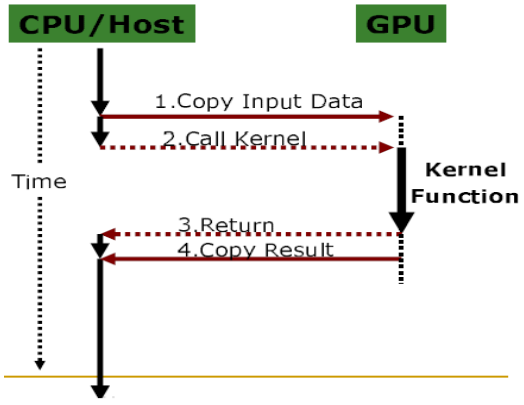
GPUها در ابتدا به صورت سخت‌افزارهایی با کارکرد ثابت و برای محاسبات گرافیکی بوجود آمدند، اما به مرور زمان قابلیت پیکربندی محدودی به آن‌ها افزوده شد. این امر باعث می‌شد تا با بیان مسائل مختلف به صورت یک مسئله‌ی گرافیکی بتوان از قابلیت پردازشی بالای آن‌ها بهره گرفت. تبدیل مسئله‌ی هدف به یک مسئله‌ی گرافیکی، خود موضوعی چالش برانگیز بود که استفاده‌ی از GPUها را مشکل و در نتیجه محدود می‌کرد. در نهایت سازندگان این تراشه‌ها به فکر ارائه‌ی یک معماری عام

[3] Query point
[4] entries
[5] Single Instruction Multiple Data
[6] throughput

منظوره به همراه یک زبان سطح بالای برنامه‌نویسی افتاده و مفهوم GPU همه‌منظوره یا GPGPU[۷] را معرفی کردند[۷]. یکی از مهمترین این معماری‌ها CUDA است که در آن هر GPU شامل چندین واحد پردازشی مستقل به نام Streaming multiprocessor است که هر یک از آن‌ها نیز حاوی چندین هسته‌ی اجرایی است. تعداد این هسته‌ها در یک GPU به چندصد هسته می‌رسد و در نتیجه تعداد نخ‌های اجرایی همزمان عدد چشم‌گیری خواهد بود [۵].

برنامه‌ای که مطابق با معماری CUDA نوشته شده باشد شامل دو جز سریال و موازی است که قسمت سریال آن بر روی CPU و قسمت موازی آن بر روی GPU و در قالب توابع خاصی با نام kernel اجرا می‌گردد. در حین اجرا ابتدا برنامه توسط CPU شروع به کار کرده و به صورت سریال و خط به خط جلو می‌رود. اگر یک خط خاص شامل یک فراخوانی kernel باشد GPU وارد عمل شده و تعداد زیادی نخ موازی را که تمامی آن‌ها تابع kernel را اجرا می‌کنند ایجاد می‌کند [۶].

از آنجایی که GPU حافظه‌ی مربوط به خود را داشته و محاسباتش را بر روی این حافظه انجام می‌دهد، قبل از فراخوانی تابع kernel باید داده‌های مورد نیاز را از حافظه‌ی اصلی به حافظه‌ی کارت گرافیکی منتقل کرد. همچنین پس از پایان محاسبات، نتیجه‌ی آن نیز در صورت لزوم باید به حافظه‌ی اصلی منتقل شود. شکل ۲ این امر را نمایش می‌دهد.



شکل ۲: نحوه‌ی اجرای یک برنامه‌ی CUDA

در معماری CUDA فراخوانی یک تابع kernel در قالب ایجاد یک تور[۸] از نخ‌ها صورت می‌گیرد. هر تور حاوی چندین بلوک بوده که هر بلوک نیز حاوی چندین نخ است. هر تور در واقع نحوه‌ی سازماندهی نخ‌ها بر روی streaming multiprocessor ها و هسته‌های آن‌ها را مشخص می‌کند.

۴- الگوریتم پیشنهادی

همان‌طور که در قسمت ۲ بیان گردید الگوریتم brute-force مطرح شده، در دو سطح قابلیت موازی‌سازی دارد. در سطح اول می‌توان با تعریف یک kernel، فاصله‌ی نقطه‌ی جستار q تا یک نقطه‌ی خاص از مجموعه‌ی مرجع R را محاسبه نمود. سپس یک تور که به تعداد نقاط مرجع، نخ همزمان دارد تعریف می‌شود. بدیهی است که با فراخوانی این kernel در قالب تور تعریف شده، فاصله‌ی نقطه‌ی q تا تمامی نقاط R به صورت همزمان محاسبه شده و در یک آرایه ذخیره می‌شود. شکل ۳ کد این kernel را نشان می‌دهد.

```
__global__ void distance_kernel(point*
points,point*query_point,float* distances)
{
  const int idx = blockDim.y*blockIdx.y + threadIdx.y;
  const float x_dif = (points[idx].x - query_point->x);
  const float y_dif = (points[idx].y - query_point->y);
  distances[idx] = (x_dif * x_dif) + (y_dif * y_dif);
}
```

شکل۳: کد kernel ای که فواصل نقاط را محاسبه می‌کند.

```
__global__void sort_kernel(float*a , float*b,int size)
{
        const int idx = blockDim.y*blockIdx.y +
threadIdx.y;
        int les = 0;
        int eq  = 0;
        for(int i=0 ; i < size ; i++ )
        {
                if(a[i] < a[idx])  les++;
                else if((a[i]== a[idx])&& i<idx)  eq++;
        }
        b[les+eq] = a[idx];
}
```

شکل ۴: کد kernel ای که مرتب‌سازی موازی را انجام می‌دهد.

در سطح دوم به منظور مرتب‌سازی آرایه‌ی حاصل یک تابع kernel دیگر تعریف می‌شود. در این تابع هر عنصر آرایه با تمامی عناصر دیگر آن مقایسه شده و مکان نهایی آن عنصر در آرایه‌ی مرتب شده پیدا می‌شود. در اینجا نیز با تعریف یک تور که به تعداد نقاط آرایه‌ی فواصل، نخ همزمان دارد می‌توان مکان تمامی عناصر را در آرایه‌ی مرتب شده به صورت همزمان محاسبه کرد.

[7] General Purpose GPU
[8] grid

شکل ۴ کد این kernel را نشان می‌دهد.

در تور تعریف شده هر نخ به یک عنصر از آرایه‌ی اولیه نگاشت می‌شود. این نخ وظیفه دارد تا مکان نهایی این عنصر را در آرایه-ی مرتب شده پیدا کند. بدین منظور مقدار عنصر متناظر خود را با تمامی عناصر آرایه مقایسه کرده و تعداد عناصری که از این عنصر کوچکترند را پیدا می‌کند. اگر تمامی عناصر آرایه غیر تکراری باشند عدد حاصل مکان نهایی این عنصر در آرایه‌ی مرتب شده را مشخص می‌کند. شکل زیر یک مثال از این دست را نشان می‌دهد.

عنصر	۱۰	۲۲	۲۰۰	۲۴۴	۱۴	۴
تعداد عناصر کوچکتر	۱	۳	۴	۵	۲	۰

آرایه‌ی مرتب شده	۴	۱۰	۱۴	۲۲	۲۰۰	۲۴۴

شکل۵: یک نمونه از مرتب سازی آرایه‌ای با عناصر غیر تکراری

چنانچه در آرایه‌ی اولیه عناصر تکراری وجود داشته باشد تعداد عناصر کوچکتر از این عناصر تکراری یکسان شده و در نتیجه چند عنصر آرایه به یک عنصر در آرایه‌ی مرتب شده‌ی نهایی نگاشت می‌شوند. پیش از بررسی راه حل این مسئله نماد e^t را به عنوان عنصر متناظر نخ و e^* را برای سایر عناصر آرایه قرارداد می‌کنیم.

به منظور حل مشکل فوق کافی است هر نخ تعداد عناصری که با e^t برابر بوده و اندیس آن‌ها از اندیس e^t کوچکتر است را بشمارد. محل نهایی e^t در آرایه‌ی مرتب شده مجموع این عدد با تعداد عناصر کوچکتر از e^t می‌باشد. شکل ۵ یک مثال از این حالت را نمایش می‌دهد.

اندیس در آرایه‌ی اولیه	۰	۱	۲	۳	۴	۵
عنصر	۴	۲۰۰	۱۴	۲۴۴	۱۴	۴
تعداد عناصر کوچکتر	۰	۴	۲	۵	۲	۰
تعداد عناصر برابر با e^t	۰	۰	۰	۰	۱	۱
محل عنصر در آرایه‌ی مرتب شده	۰+۰	۰+۴	۰+۲	۰+۵	۱+۲	۱+۰

آرایه‌ی مرتب شده	۴	۴	۱۴	۱۴	۲۰۰	۲۴۴

شکل۶ : یک نمونه از مرتب سازی آرایه‌ای با عناصر تکراری

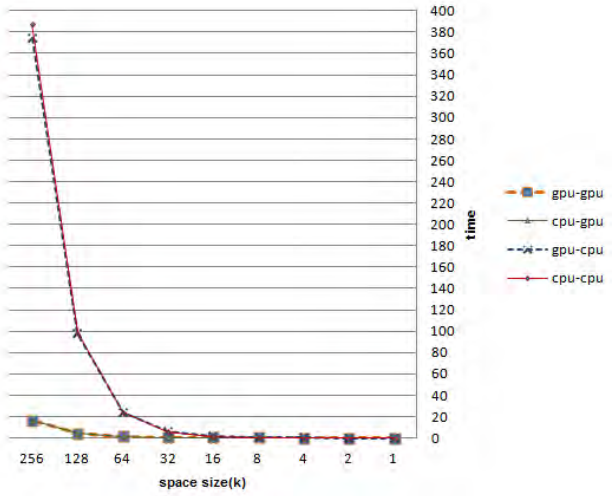
۵- ارزیابی الگوریتم پیشنهادی

به منظور ارزیابی الگوریتم پیشنهاد شده یک فضای S دو بُعدی که به صورت تصادفی ایجاد می‌شود در نظر گرفته شد. تعداد نقاط این فضا در آزمایش‌های مختلف در محدوده‌ی ۱۰۲۴ تا ۲۵۶*۱۰۲۴ تغییر یافت.

دو kernel اجرا کننده‌ی الگوریتم پیشنهادی نیز در دو تور مجزا محاسبات خود را به انجام می رسانند. در هر یک از این تورها تعداد بلوک‌ها متغیر، ولی تعداد نخ‌های هر یک از آن‌ها عدد ثابت ۱۰۲۴ فرض شد. بنابراین به منظور افزایش حجم فضای S تعداد بلوک‌های تور از ۱ تا ۲۵۶ افزایش یافته و در نتیجه حجم این فضا از ۱۰۲۴ تا ۱۰۲۴*۲۵۶ تغیر کرد.

نتایج حاصل از این آزمایش‌ها در شکل ۷ آورده شده است. این شکل حاوی چهار منحنی است که هر یک از آن‌ها معرف یک دسته آزمایش مجزاست.

در این آزمایش‌ها تابع محاسبه کننده‌ی فاصله‌ی نقاط و نیز تابع مرتب‌سازی فواصل به صورت مجزا در نظر گرفته شده است و هر چهار حالت ممکن اجرای این توابع بر روی CPU و GPU مورد کنکاش قرار گرفته است.



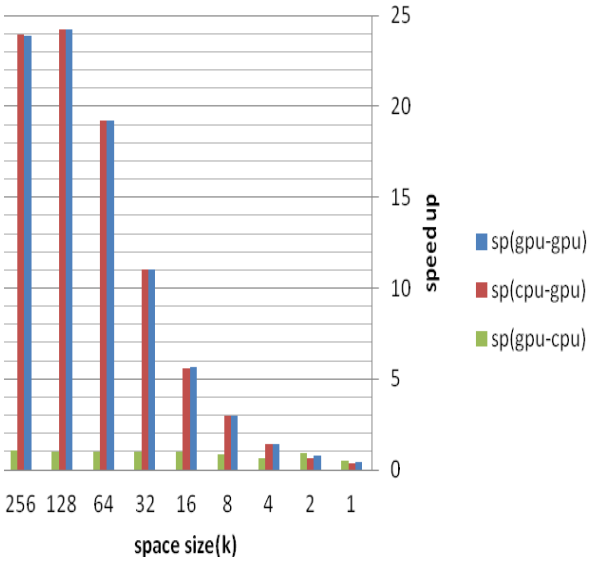
شکل ۷: نتایج حاصل از اجرای الگوریتم در حالت‌های مختلف اجرا

در شکل فوق و در هر یک از حالت‌های GPU-GPU، CPU-GPU، GPU-CPU و CPU-CPU عبارت سمت چپ معرف پلتفرمی است که تابع محاسبه‌ی فاصله‌ی نقاط بر روی آن اجرا شده است. عبارت سمت راست نیز پلتفرم اجرای تابع مرتب‌سازی را نشان می‌دهد. به عنوان مثال عبارت CPU-GPU نشان می-دهد که تابع محاسبه‌ی فاصله بر روی CPU و تابع مرتب‌سازی بر روی GPU اجرا شده است.

همان‌طور در شکل ۷ مشاهده می‌شود اختلاف زمان‌اجرای حالت‌های CPU-CPU و GPU-CPU بسیار اندک بوده و دو منحنی متناظر این حالات تقریبا بر روی هم قرار گرفته‌اند. همچنین در حالت‌های GPU-GPU و CPU-GPU نیز این اتفاق رخ داده است. این امر نشان می‌دهد که محاسبه‌ی فواصل نقاط بر روی CPU و GPU تقریبا زمان اجرای یکسانی داشته است که

علت آن سربار انتقال داده‌های مورد نیاز از حافظه‌ی اصلی به حافظه‌ی کارت گرافیک و برعکس می‌باشد.

شکل ۸ نیز میزان speed up را در هر یک از حالت‌های GPU-GPU، CPU-GPU، GPU-CPU در مقایسه با حالت CPU-CPU نشان می‌دهد.



شکل ۸: مقایسه‌ی میزان speed up در حالت‌های مختلف

۶- نتیجه‌گیری

در این مقاله یک الگوریتم کارا برای پیاده‌سازی KNN بر روی GPU پیشنهاد شد. این الگوریتم موازی سازی در دو سطح محاسبه‌ی فاصله‌ی نقاط و مرتب‌سازی فواصل را در نظر می‌گیرد. پس از انجام پیاده‌سازی این الگوریتم و بررسی نتایج مختلف مشخص شد که به علت سربار بالای انتقال داده‌ها در مقابل حجم اندک محاسبات صورت گرفته در هر یک از نخ‌هایی که فاصله‌ی نقاط را محاسبه می‌کنند عمده‌ی speed up حاصل از اجرای الگوریتم ناشی از اجرای موازی مرتب‌سازی بوده است.

مراجع

[1] Shenshen Liang, Cheng Wang, Ying Liu and Liheng Jian, "CUKNN: A parallel implementation of K-nearest neighbor on CUDA enabled GPU," Information, Computing and Telecommunication, pp. 415 – 418, 2009.

[2] Garcia V., Debreuve E., Nielsen F. and Barlaud, M., "K-nearest neighbor search: fast GPU-based implementations and application to high dimensional feature matching," Image Processing (ICIP),pp. 3757-3760, 2010.

[3] Bhattacharya B. and Kaller D, "Reference set thinning for the K-nearest neighbor decision rule," Pattern Recognition,pp. 238-242, 1998.

[4] Nickolls J. and Dally W.J., "The GPU computing era," Micro, IEEE,pp. 56-69, 2010.

[5] Yao Zhang and John D., "A quantitative performance analysis model for GPU architecture," HPCA '11 Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture,pp. 382-393, 2011.

[6] Song Jun Park, "An analysis of GPU parallel computing," DoD High Performance Computing Modernization Program Users Group Conference,pp. 365-369, 2010.

[7] Owens J.D., Houston M., Luebke D., Green S., Stone J.E. and Phillips J.C., "GPU computing," Proceedings of the IEEE,pp. 879-899, 2008.