گزارش تمرین پنجم

تینا صداقت 9331044

الگوریتم سریال prefix-sum:

در این الگوریتم یک نخ روی همه‌ی المان‌های ورودی آرایه loop را اجرا می‌کند و هر بار خانه‌ی قبلی را با خانه‌ی فعلی جمع می‌کند و در آرایه‌ی نهایی قرار می‌دهد. این الگوریتم باید n بار عمل جمع را انجام دهد.

کد مربوط به الگوریتم سریال در ضمیمه آمده است.

f\_out[0] = 0;

for (int i = 1; i < i\_n; i++)

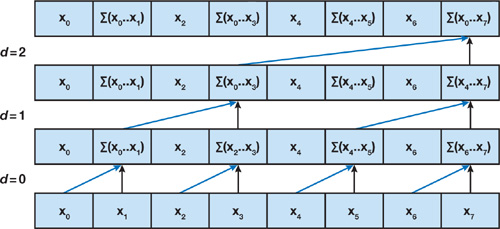
f\_out[i] = f\_out[i - 1] + f\_in[i - 1];

الگوریتم موازی‌سازی:

از الگوریتم balanced tree با n برگ که logn2 d= سطح و هر سطح d2 گره دارد، استفاده می‌کنیم. از حافظه اشتراکی کمک می‌گیریم. دو فاز داریم: 1. فاز up-sweep و 2. فاز down-sweep

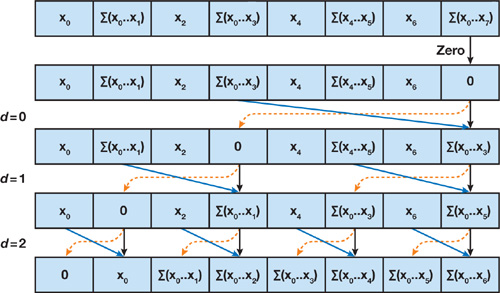
در فاز اول، درخت را از برگ‌ها به ریشه پیمایش می‌کنیم و جمع‌های جزئی در هر گره حساب می‌شوند. ریشه در پایان این فاز، مقدار جمع همه‌ی گره ها را دارد. شکل و Pseudo code این فاز را در زیر می‌بینیم:

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



در فاز دوم، از ریشه به درخت را پیمایش می‌کنیم و از جمع‌های جرئی تولید شده در فاز قبل، آرایه نهایی را تشکیل می‌دهیم. به این صورت که در ریشه 0 می‌گذاریم و در مرحله، هر گره مقدار خودش را به گره سمت چپ می‌دهد و جمع خودش با مقدار قبلی گره سمت چپی را به گره سمت راستش می‌دهد. شکل و pseudo code این فاز را در زیر می‌بینیم:

1: x[n – 1] U2190.GIF 0  
2: **for** d = log2 n – 1 down to 0 **do**  
3:       **for all** k = 0 to n – 1 by 2 d +1 in parallel **do**  
4:            t = x[k +  2 d  – 1]  
5:            x[k +  2 d  – 1] = x[k +  2 d +1 – 1]  
6:            x[k +  2 d +1 – 1] = t +  x[k +  2 d +1 – 1]



آرایه را به چندین بلوک تقسیم می‌کنیم که هر کدام می‌توانند توسط یک بلوکی از نخ‌ها اجرا شوند. و سپس در هر بلوک الگوریتم scan را اجرا می‌کنیم و جمع کلی هر بلوک را در آرایه دیگری از جمع بلوک‌ها می‌نویسیم. سپس برای بلوک‌ها با جمع کردن المان‌های بلوک های مربوطه، الگوریتم scan را اجرا می‌کنیم. به طور کلی اگر آرایه n تایی باشد و در هر بلوک b تا خانه بررسی شود، باید n/b بلوک نخ داشته باشیم که در هرکدام b/2 نخ باشد.

پیاده‌سازی تابع kernel ( توضیح کد بخش kernel):

آرگومان‌های این تابع 3 متغیر g\_odata (آرایه خروجی) و g\_idata (آرایه ورودی) و n(تعداد خانه‌های آرایه) می‌باشد. آرایه‌ی temp را به صورت حافظه اشتراکی(shared) تعریف می‌کنیم چون دسترسی به آن سریعتر از دسترسی به حافظه سراسری است. اندازه این حافظه اشتراکی باید به اندازه‌ی تعداد نخ‌های هر بلاک باشد. ماکسیمم سایز حافظه اشتراکی برابر 48KB است. پس محدودیت سایز حافظه اشتراکی داریم. شماره نخ در متغیر thid و شماره بلوک در متغیر bid ذخیره می‌شود.

extern \_\_shared\_\_ float temp[];

// allocated on invocation

int thid = threadIdx.x;

int bid = blockIdx.x;

درصورت کوچکتر بودن شماره نخ از تعداد خانه‌های آرایه(n)، خانه‌ی thid ام از آرایه‌ی حافظه اشتراکی temp را با مقدار خانه با اندیس شماره نخ(واقعی) آرایه ورودی پر می‌کنیم:

)bid: شماره بلوک , thread\_num: تعداد نخ‌های یک بلوک, thid: شماره نخ در بلوک (

int offset = 1;

if ((bid \* thread\_num + thid)<n) {

temp[thid] = g\_idata[bid \* thread\_num + thid];

}

else {

temp[thid] = 0;

} // Make the "empty" spots zeros, so it won't affect the final result.

سپس یک for با تکراری به اندازه‌ی تعداد نخ‌های بلوک تشکیل می‌دهیم. باید صبر کنیم همه‌ی نخ‌ها به این خط برسند و بعد درصورتی که آخرین نخ نباشد باید پوینتر دو خانه‌ی متوالی آن نخ را پیدا کنیم(aiوbi) و خانه‌ی جدید را با خانه‌ی قبلی جمع کنیم. هر بار offset را دو برابر می‌کنیم تا بتوانیم به سطح بعدی درخت برویم.

for (int d = thread\_num >> 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;

temp[bi] += temp[ai];

}

offset \*= 2;

}

اگر شماره نخ 0 است، خانه‌ی آخر temp را 0 می‌کنیم.

if (thid == 0)

{

temp[thread\_num - 1] = 0;

}

در فاز دوم درخت را به پایین پیمایش می‌کنیم و الگوریتم scan را انجام می‌دهیم.

// clear the last element

for (int d = 1; d < thread\_num; d \*= 2)

// traverse down tree & build scan

{

offset >>= 1;

\_\_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;

}

}

در آخر صبر می‌کنیم همه‌ی نخ‌های آن بلوک به این خط برسند و آرایه‌ی temp را در خروجی منتقل می‌کنیم. (آرایه‌ی temp برای بلوک 0ام، خانه‌های 0 تا thread\_num را در خروجی پر می‌کند،

آرایه‌ی temp برای بلوک 1ام، خانه‌های thread\_num تا thread\_num\*2 را در خروجی پر می‌کند و ...)

\_\_syncthreads();

g\_odata[bid \* thread\_num + thid] = temp[thid];

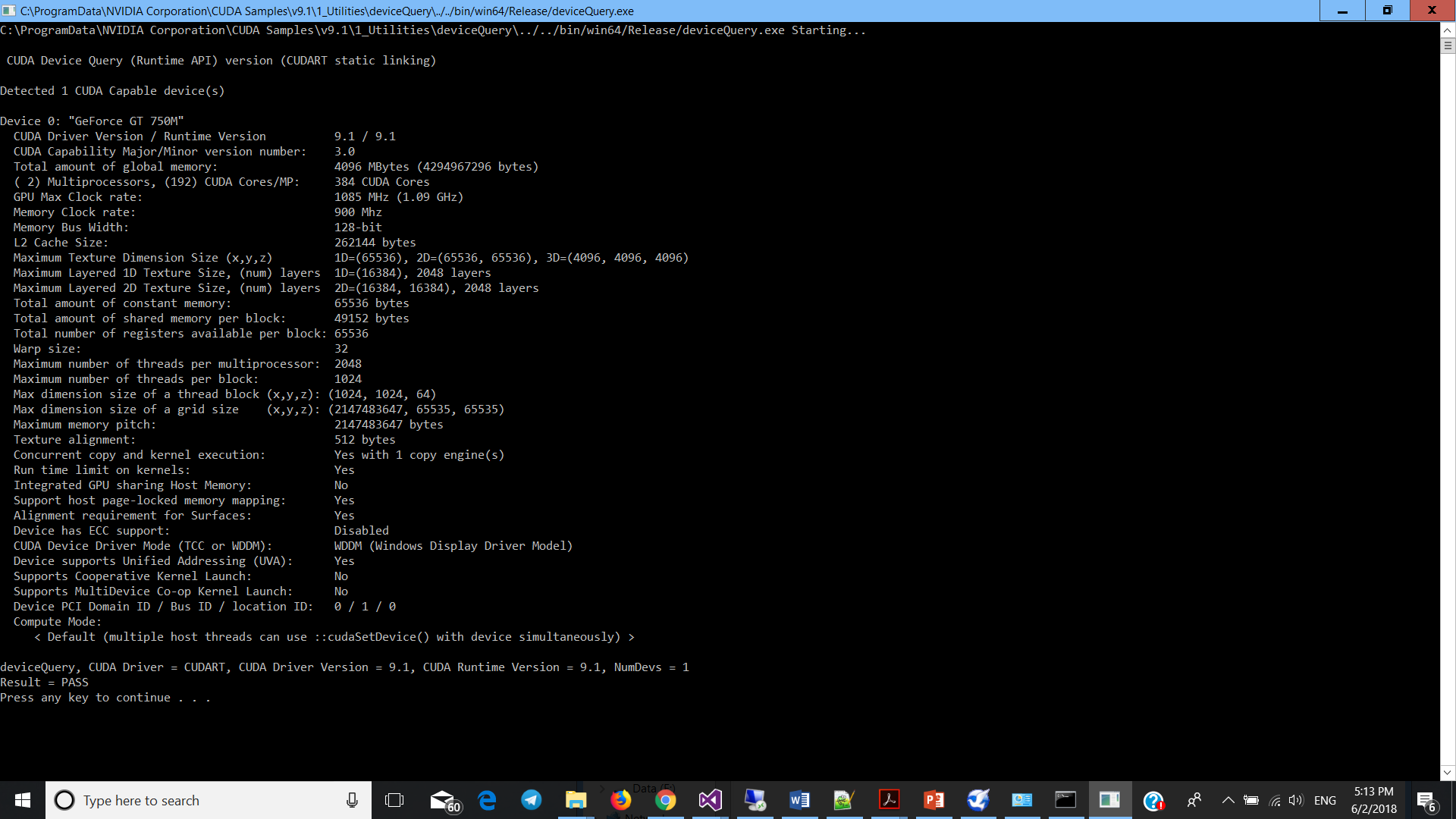
به طور مثال فرض کنید سایز آرایه 8 باشد و دو تا بلوک 4 نخی داشته باشیم. پس نتیجه شامل دو آرایه است که هر کدام توسط بلوک‌های مجزا حساب شده‌اند و نتیجه نهایی شامل جمع همه خانه‌های آرایه اول و سپس جمع عدد به‌دست آمده با همه‌ی خانه‌های آرایه دوم است. در اینجا آرایه اول خانه‌های 0 و1و3و6 را دارد که مجموعشان 10 می‌شود پس نتیجه درواقع جمع خانه های 0و5و11و18 با عدد 10 است که می‌شود:

0 ,1, 3, 6, 10, 15, 21, 28

1. به ازای ابعاد مختلف امتحان می‌کنیم(زمان اجرای سریال از آزمایش 4 حساب می‌شود):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| تعداد خانه‌های آرایهN | تعداد بلوک‌ها block\_num | تعداد نخ‌ها thread\_num | زمان موازی | زمان سریال |
| 8 | 2 | 8 | 0.047 | 0.0000 |
| 256 | 2 | 256 | 0.091 | 0.001 |
| 256 | 4 | 256 | 0.051 | 0.001 |
| 512 | 2 | 512 | 0.067 | 0.002 |
| 512 | 4 | 512 | 0.079 | 0.002 |
| 512 | 8 | 512 | 0.059 | 0.002 |
| 1024 | 4 | 512 | 0.080 | 0.004 |
| 1024 | 8 | 256 | 0.088 | 0.004 |
| 1024 | 16 | 128 | 0.084 | 0.004 |
| 2048 | 32 | 64 | 0.099 | 0.007 |
| 4096 | 4 | 1024 | 0.066 | 0.009 |
| 4096 | 8 | 512 | 0.062 | 0.009 |
| 4096 | 16 | 256 | 0.112 | 0.009 |
| 4096 | 32 | 128 | 0.076 | 0.009 |
| 4096 | 64 | 64 | 0.065 | 0.009 |
| 8192 | 8 | 1024 | 0.080 | 0.013 |
| 8192 | 16 | 512 | 0.090 | 0.013 |
| 8192 | 32 | 256 | 0.110 | 0.013 |
| 16384 | 16 | 1024 | 0.126 | 0.020 |
| 16384 | 32 | 512 | 0.112 | 0.020 |
| 16384 | 64 | 256 | 0.109 | 0.020 |
| 16384 | 128 | 128 | 0.065 | 0.020 |
| 32768 | 32 | 1024 | 0.206 | 0.10 |
| 32768 | 64 | 512 | 0.151 | 0.10 |
| 131072 | Stack over flow |  |  |  |

1. تعداد و اندازه بلوک انتخابی به عوامل مختلفی بستگی دارد:

کد آماده‌ی devicequery.exe را اجرا می‌کنیم تا اطلاعات GPU دستگاه را بررسی کنیم:

همانطور که می‌بینیم این GPU دارای دو microprocessor یا همان SM است. سایز هر warp 32 تا نخ است و بیشترین تعداد نخی که در هر SM می‌تواند اجرا شود 2048 تا است. همچنین در هر بلوک حداکثر می‌توان 1024 نخ جا داد. (با فرض اینکه هر SM بتواند حداکثر 8 بلوک فعال داشته باشد) با توجه به این مقادیر باید بهترین تعداد و اندازه بلوک‌ها را تعیین کنیم:

پس برای مقادیر بالای N، حداکثر می‌توان 1024 نخ در هر بلوک داشت و چون هر SM تنها 2048 بلوک را می‌تواند اجرا کند پس دو تا بلوک را می‌تواند داشته باشد. پس در کل 4 بلوک فعال داریم و بقیه بلوک ها باید منتطر بمانند. اما در صورتی که تعداد نخ ها را کمتر کنیم، مثلا اگر 512 نخ در هر بلوک داشته باشیم، هر SM می‌تواند 4 بلوک فعال داشته باشد پس در مجموع 8 بلوک فعال در GPU اجرا می‌شود که باعث سرعت بالاتر اجرا می‌شود.(چون در صورت آماده نبودن یک بلوک، فوری بلوک دیگری را SM اجرا می‌کند و بیکار نمی‌ماند) پس نتیجه می‌گیریم هر چه تعداد نخ ها در هر بلوک(البته با رعایت شرایط ذکر شده) کمتر باشد و تعداد بلوک بیشتر باشد( تا حدی که هر SM بتواند در خود اختصاص دهد) سرعت اجرای برنامه بیشتر می‌شود.

1. خیر فرق دارد. بسته به این که سایز آرایه چقدر باشد، باید با توجه به بالا، اندازه بلوک و تعداد نخ های آن بلوک تعیین گردد.
2. باید با امتحان کردن اندازه بلوک‎‌های مختلف، ببینیم که به ازای چه تعدادی از بلوک‌ها، از SM ها استفاده حداکثری می‌شود. البته چون دو SM داریم و درصورتی که از تعداد 2 به بیشتر بلوک استفاده کنیم، هر SM کاری برای انجام دادن دارد، اما باید طوری پیاده سازی شود که 2048 نخ در هر SM فعال باشند تا استفاده حداکثری از SM شود.
3. دو الگوریتم موازی سازی شده با CPU در پیوست آمده است و نتایج زمانی را در زیر می‌بینیم:

روش اول CPU:



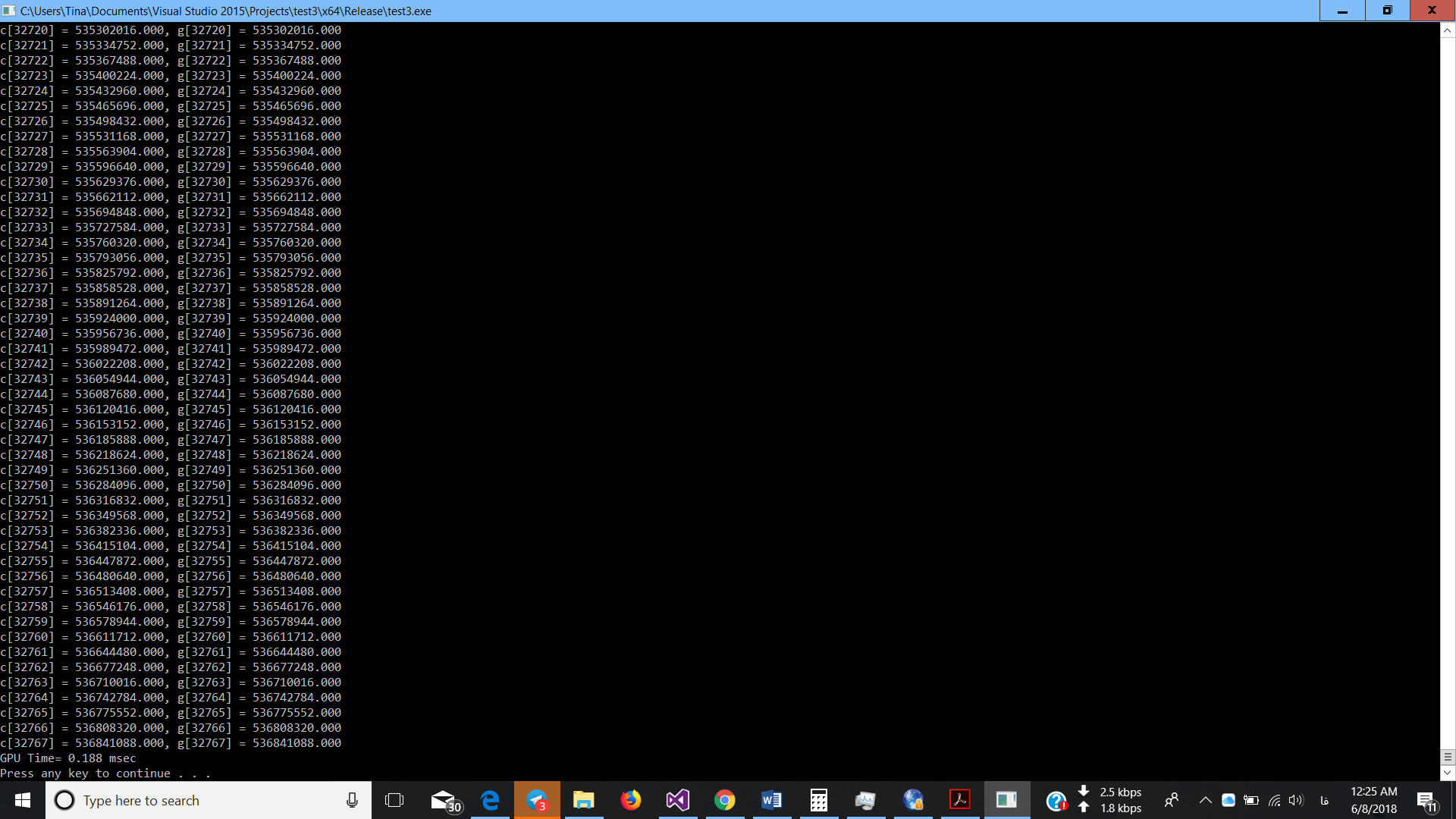
روش دوم CPU:

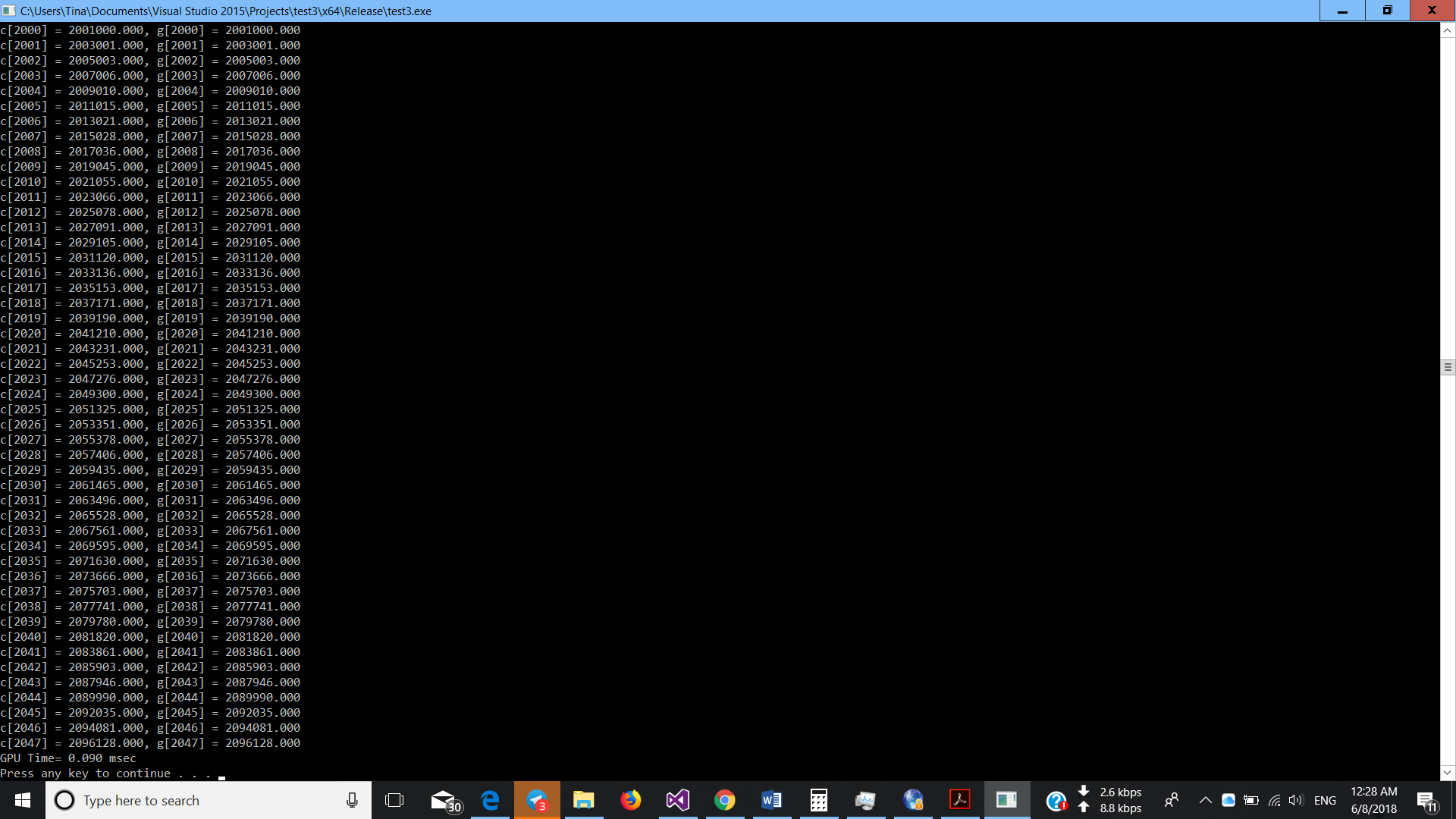
تقریبا می‌توان گفت که الگوریتم موازی سازی شده با CPU در روش اول بهتر از الگوریتم GPU است اما الگوریتم GPU از الگوریتم موازی سازی شده با CPU در روش دوم بهتر است. توجیه آن درواقع به نحوه پیاده سازی آنها، تعداد نخ ها و همچنین نحوه تخصیص کارها به نخ ها بستگی دارد. اما روش دوم CPU و الگوریتم ارائه شده در اینجا (GPU) چون هر دو از الگوریتم Hillis and Steele استفاده شده است و GPU قدرت تسریع سازی بیشتری به کمک نخ ها نسبت به CPU دارد پس سرعت بیشتری در اجرا داشته است.

1. این الگوریتم برای float تست شده بود. که در این سوال به جای آرایه ای از float ها، به ترتیب آرایه ای از int ها و double ها تشکیل دادیم. برای int مقدار کمی تسریع گرفتیم ( به دلیل سایز کمتر نسبت به float) .

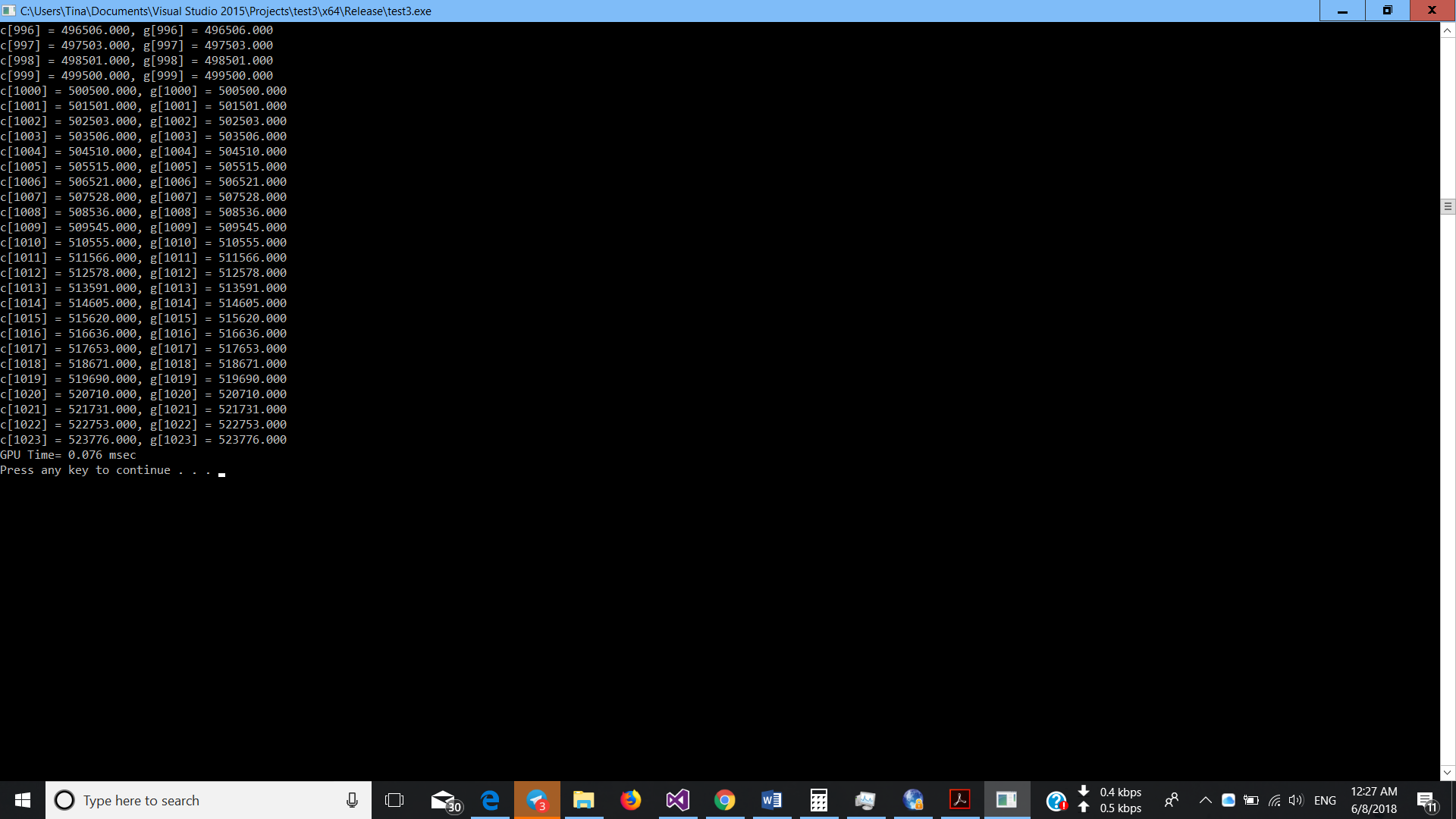
برای double مقدار نسبتا ناچیزی کم شدن در سرعت را داشتیم.

خروجی های برنامه درست هستند: به طور مثال:

به ازای ورودی N=32768:

به ازای N=2048:

به ازای N=1024:



برای تست از نرم افزار Nsight کمک گرفتم که در زیر screenshot هایی از اطلاعات مهم GPU و نخ ها و زمان ها را آورده ام:

