گزارش تمرین پنجم

تینا صداقت ۹۳۳۱۰۴۴

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];
```

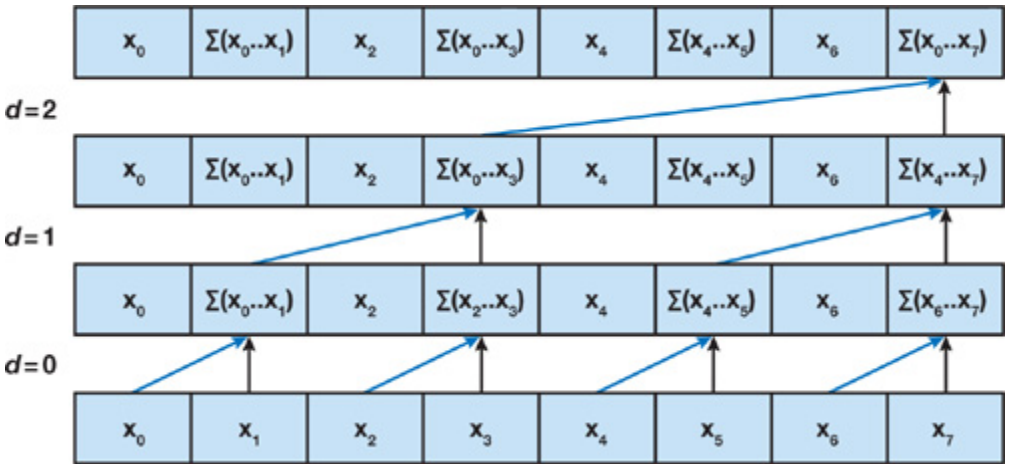
.۱

الگوریتم موازی‌سازی:

از الگوریتم balanced tree با n برگ که $d= \log^n_2$ سطح و هر سطح 2^d گره دارد، استفاده می‌کنیم. از حافظه اشتراکی کمک می‌گیریم. دو فاز داریم: ۱. فاز up-sweep و ۲. فازdown-sweep

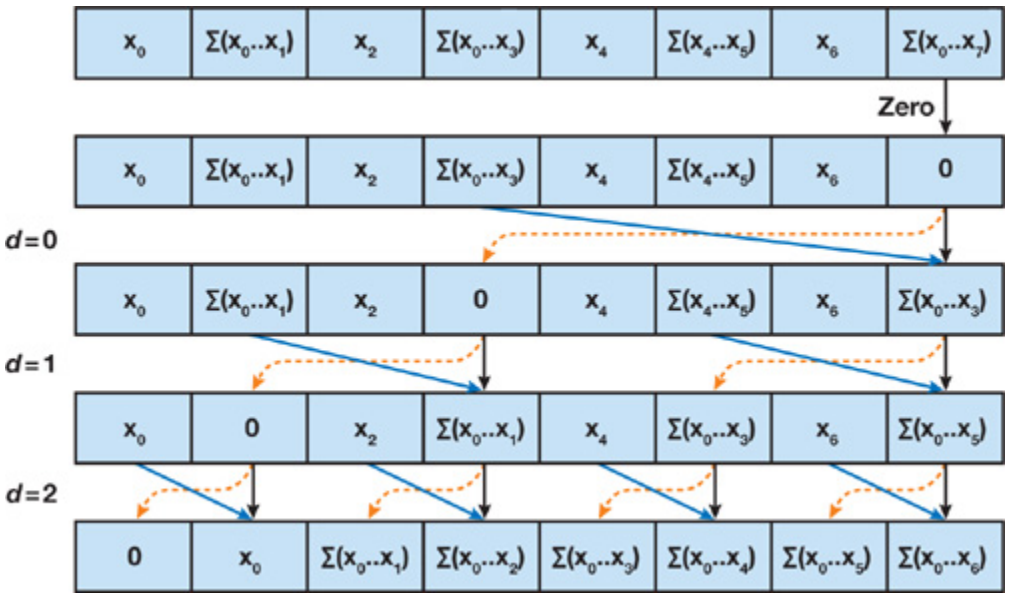
در فاز اول، درخت را از برگ‌ها به ریشه پیمایش می‌کنیم و جمع‌های جزئی در هر گره حساب می‌شوند. ریشه در پایان این فاز، مقدار جمع همه‌ی گره ها را دارد. شکل و Pseudo code این فاز را در زیر می‌بینیم:

1: **for** d = 0 to $\log_2 n - 1$ **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 ۰ 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: $x[n-1] \leftarrow 0$
2: **for** $d = \log_2 n - 1$ 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: $x[k + 2^{d+1} - 1] = x[k + 2^d + 1 - 1]$
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 ۳ 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 ۰ است، خانه‌ی آخر temp را ۰ 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 ۰ام، خانه‌های ۰ تا thread_num را در خروجی پر می‌کند، آرایه‌ی temp برای بلوک ۱ام، 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 ۸ باشد و دو تا بلوک ۴ 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 ۰ و۱و۳و۶ را دارد که مجموعشان ۱۰ می‌شود پس نتیجه درواقع جمع خانه های ۰و۵و۱۱و۱۸ با عدد ۱۰ است که می‌شود:

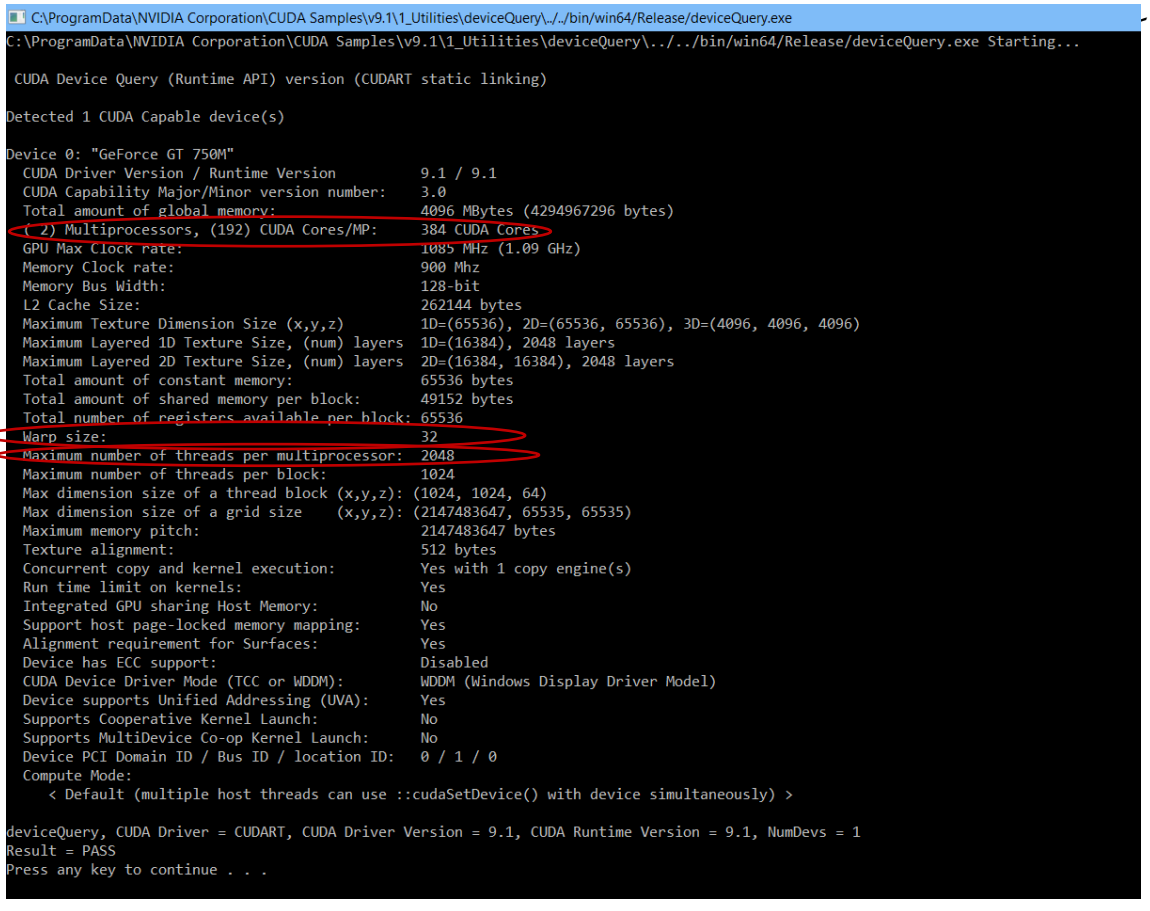
0 ,1, 3, 6, 10, 15, 21, 28

۲. به ازای ابعاد مختلف امتحان می‌کنیم(زمان اجرای سریال از آزمایش ۴ 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			

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



همانطور که می‌بینیم این GPU دارای دو microprocessor یا همان SM است. سایز هر warp ۳۲ تا نخ است و بیشترین تعداد نخی که در هر SM می‌تواند اجرا شود ۲۰۴۸ تا است. همچنین در هر بلوک حداکثر می‌توان ۱۰۲۴ نخ جا داد. (با فرض اینکه هر SM بتواند حداکثر ۸ 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 ۱۰۲۴ نخ در هر بلوک داشت و چون هر SM تنها ۲۰۴۸ بلوک را می‌تواند اجرا کند پس دو تا بلوک را می‌تواند داشته باشد. پس در کل ۴ بلوک فعال داریم و بقیه بلوک ها باید منتظر بمانند. اما در صورتی که تعداد نخ ها را کمتر کنیم، مثلا اگر ۵۱۲ نخ در هر بلوک داشته باشیم، هر SM می‌تواند ۴ بلوک فعال داشته باشد پس در مجموع ۸ 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.

a. 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.
b. 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 ۲ به بیشتر بلوک استفاده کنیم، هر SM کاری برای انجام دادن دارد، اما باید طوری پیاده سازی شود که ۲۰۴۸ نخ در هر SM فعال باشند تا استفاده حداکثری از SM شود.

۴. دو الگوریتم موازی سازی شده با CPU در پیوست آمده است و نتایج زمانی را در زیر می‌بینیم:

روش اول CPU:

تعداد نخ ها- سایز آرایه	100000	1000000	10000000	100000000
2	P: 0.001054	P: 0.008606	P: 0.025727	P: 0.256765
	S: 0.001055	S: 0.004224	S: 0.026230	S: 0.258038
4	P: 0.001057	P: 0.010601	P: 0.025398	P: 0.257758
	S: 0.001057	S: 0.010290	S: 0.025783	S: 0.257609

روش دوم CPU:

تعداد نخ ها- سایز آرایه	100000	1000000	10000000	100000000
2	P: 0.018858	P: 0.087240	P: 0.026232	P: 7.106513
	S: 0.001180	S: 0.002890	S: 0.028895	S: 0.282532
4	P: 0.011320	P: 0.061503	P: 0.569756	P: 6.188241
	S: 0.000944	S: 0.002938	S: 0.029425	S: 0.280924

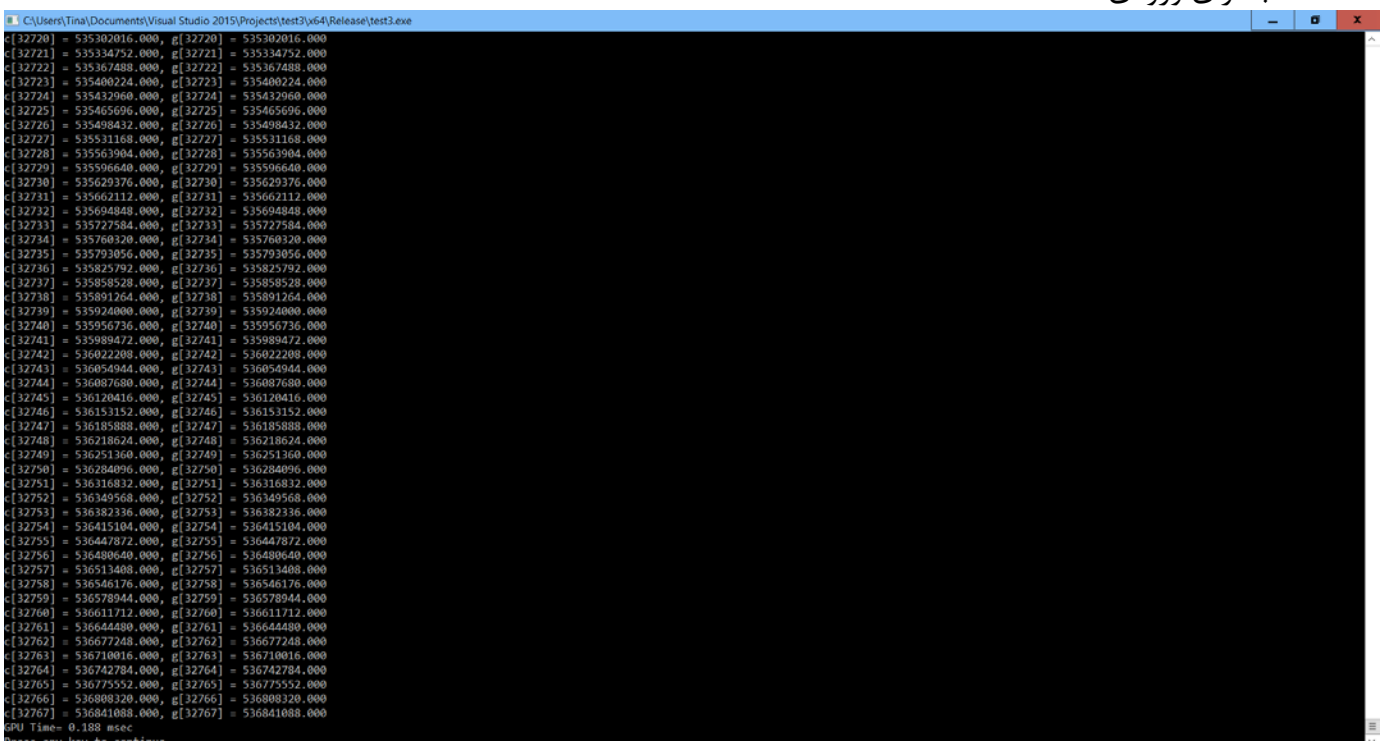
تقریبا می‌توان گفت که الگوریتم موازی سازی شده با CPU در روش اول بهتر از الگوریتم GPU است اما الگوریتم GPU از الگوریتم موازی سازی شده با CPU در روش دوم بهتر است. توجیه آن درواقع به نحوه پیاده سازی آنها، تعداد نخ ها و همچنین نحوه تخصیص کارها به نخ ها بستگی دارد. اما روش دوم CPU و الگوریتم ارائه شده در اینجا (GPU) چون هر دو از الگوریتمHillis 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.

۵. 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:



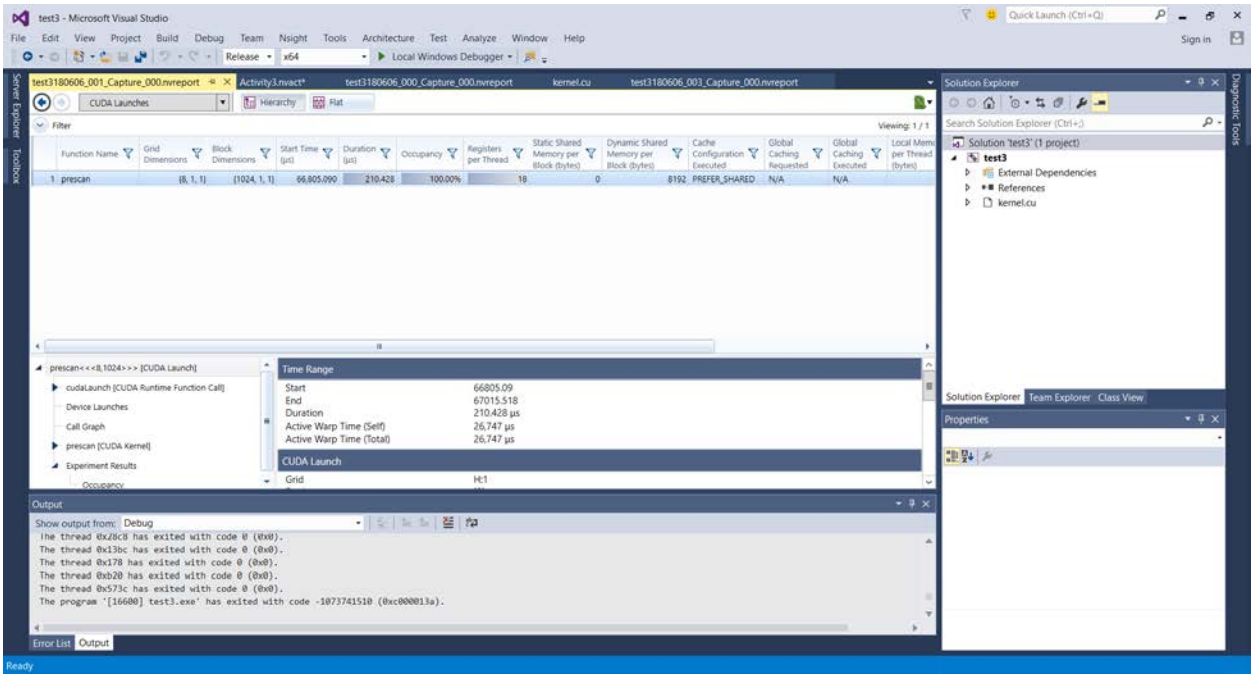
به ازای N=2048:

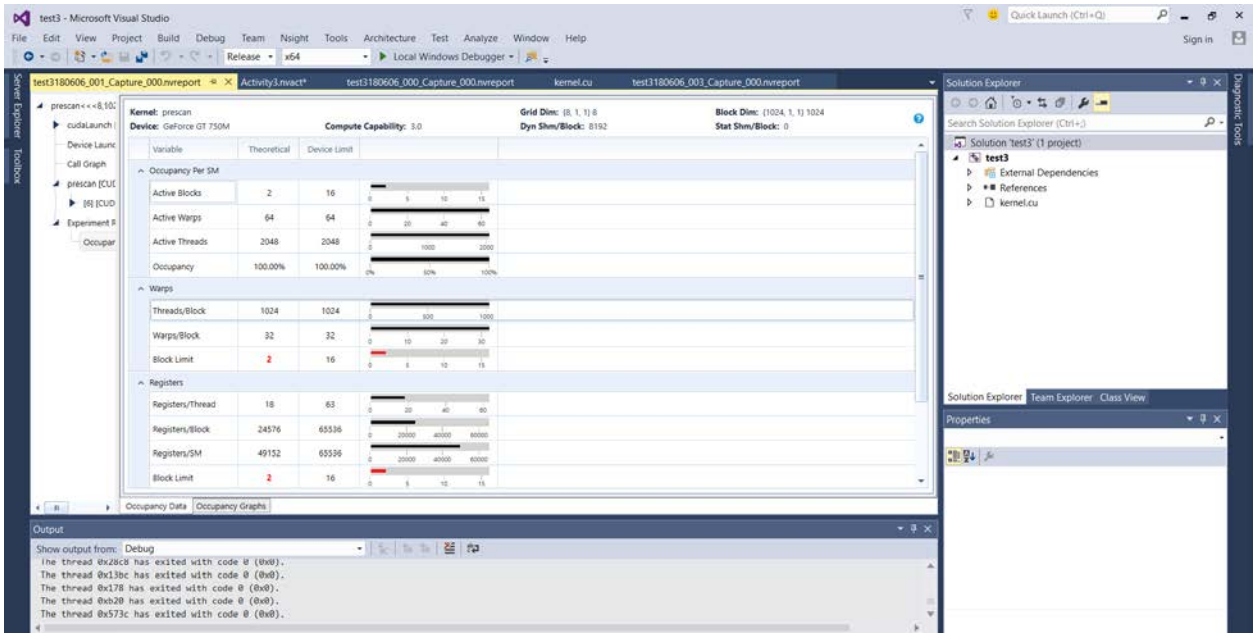
```
C:\Users\Tina\Documents\Visual Studio 2015\Projects\test3\x64\Release\test3.exe
c[2000] = 2001000.000, g[2000] = 2001000.000
c[2001] = 2003001.000, g[2001] = 2003001.000
c[2002] = 2005003.000, g[2002] = 2005003.000
c[2003] = 2007006.000, g[2003] = 2007006.000
c[2004] = 2009010.000, g[2004] = 2009010.000
c[2005] = 2011015.000, g[2005] = 2011015.000
c[2006] = 2013021.000, g[2006] = 2013021.000
c[2007] = 2015028.000, g[2007] = 2015028.000
c[2008] = 2017036.000, g[2008] = 2017036.000
c[2009] = 2019045.000, g[2009] = 2019045.000
c[2010] = 2021055.000, g[2010] = 2021055.000
c[2011] = 2023066.000, g[2011] = 2023066.000
c[2012] = 2025078.000, g[2012] = 2025078.000
c[2013] = 2027091.000, g[2013] = 2027091.000
c[2014] = 2029105.000, g[2014] = 2029105.000
c[2015] = 2031120.000, g[2015] = 2031120.000
c[2016] = 2033136.000, g[2016] = 2033136.000
c[2017] = 2035153.000, g[2017] = 2035153.000
c[2018] = 2037171.000, g[2018] = 2037171.000
c[2019] = 2039190.000, g[2019] = 2039190.000
c[2020] = 2041210.000, g[2020] = 2041210.000
c[2021] = 2043231.000, g[2021] = 2043231.000
c[2022] = 2045253.000, g[2022] = 2045253.000
c[2023] = 2047276.000, g[2023] = 2047276.000
c[2024] = 2049300.000, g[2024] = 2049300.000
c[2025] = 2051325.000, g[2025] = 2051325.000
c[2026] = 2053351.000, g[2026] = 2053351.000
c[2027] = 2055378.000, g[2027] = 2055378.000
c[2028] = 2057406.000, g[2028] = 2057406.000
c[2029] = 2059435.000, g[2029] = 2059435.000
c[2030] = 2061465.000, g[2030] = 2061465.000
c[2031] = 2063496.000, g[2031] = 2063496.000
c[2032] = 2065528.000, g[2032] = 2065528.000
c[2033] = 2067561.000, g[2033] = 2067561.000
c[2034] = 2069595.000, g[2034] = 2069595.000
c[2035] = 2071630.000, g[2035] = 2071630.000
c[2036] = 2073666.000, g[2036] = 2073666.000
c[2037] = 2075703.000, g[2037] = 2075703.000
c[2038] = 2077741.000, g[2038] = 2077741.000
c[2039] = 2079780.000, g[2039] = 2079780.000
c[2040] = 2081820.000, g[2040] = 2081820.000
c[2041] = 2083861.000, g[2041] = 2083861.000
c[2042] = 2085903.000, g[2042] = 2085903.000
c[2043] = 2087946.000, g[2043] = 2087946.000
c[2044] = 2089990.000, g[2044] = 2089990.000
c[2045] = 2092035.000, g[2045] = 2092035.000
c[2046] = 2094081.000, g[2046] = 2094081.000
c[2047] = 2096128.000, g[2047] = 2096128.000
GPU Time= 0.090 msec
Press any key to continue . . .
```

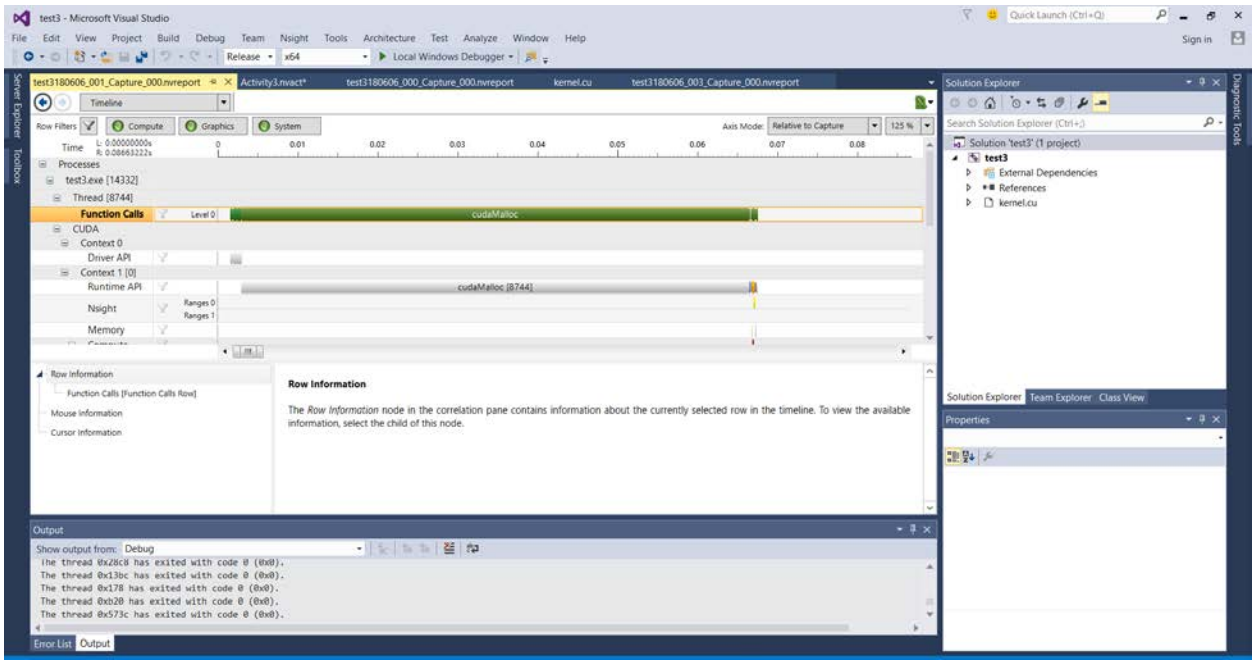
به ازای N=1024:

```
C:\Users\Tina\Documents\Visual Studio 2015\Projects\test3\x64\Release\test3.exe
c[996] = 496506.000, g[996] = 496506.000
c[997] = 497503.000, g[997] = 497503.000
c[998] = 498501.000, g[998] = 498501.000
c[999] = 499500.000, g[999] = 499500.000
c[1000] = 500500.000, g[1000] = 500500.000
c[1001] = 501501.000, g[1001] = 501501.000
c[1002] = 502503.000, g[1002] = 502503.000
c[1003] = 503506.000, g[1003] = 503506.000
c[1004] = 504510.000, g[1004] = 504510.000
c[1005] = 505515.000, g[1005] = 505515.000
c[1006] = 506521.000, g[1006] = 506521.000
c[1007] = 507528.000, g[1007] = 507528.000
c[1008] = 508536.000, g[1008] = 508536.000
c[1009] = 509545.000, g[1009] = 509545.000
c[1010] = 510555.000, g[1010] = 510555.000
c[1011] = 511566.000, g[1011] = 511566.000
c[1012] = 512578.000, g[1012] = 512578.000
c[1013] = 513591.000, g[1013] = 513591.000
c[1014] = 514605.000, g[1014] = 514605.000
c[1015] = 515620.000, g[1015] = 515620.000
c[1016] = 516636.000, g[1016] = 516636.000
c[1017] = 517653.000, g[1017] = 517653.000
c[1018] = 518671.000, g[1018] = 518671.000
c[1019] = 519690.000, g[1019] = 519690.000
c[1020] = 520710.000, g[1020] = 520710.000
c[1021] = 521731.000, g[1021] = 521731.000
c[1022] = 522753.000, g[1022] = 522753.000
c[1023] = 523776.000, g[1023] = 523776.000
GPU Time= 0.076 msec
Press any key to continue . . .
```

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:






test3 - Microsoft Visual Studio
File Edit View Project Build Debug Team Nsight Tools Architecture Test Analyze Window Help
Release x64 Local Windows Debugger
test3180606_001_Capture_000.nvreport
Activity3.nvact*
test3180606_000_Capture_000.nvreport
kernel.cu
test3180606_003_Capture_000.nvreport
Timeline
Row Filters Compute Graphics System
Axis Mode: Relative to Capture
Processes
test3.exe [14332]
Thread [8744]
Function Calls
cudaMalloc
CUDA
Context 0
Driver API
Context 1 [0]
Runtime API
cudaMalloc [8744]
Nsight
Memory
Row Information
Function Calls [Function Calls Row]
Mouse Information
Cursor Information
Row Information
The Row Information node in the correlation pane contains information about the currently selected row in the timeline. To view the available information, select the child of this node.
Output
Show output from: Debug
The thread 0x28c8 has exited with code 0 (0x0).
The thread 0x13bc has exited with code 0 (0x0).
The thread 0x178 has exited with code 0 (0x0).
The thread 0xb20 has exited with code 0 (0x0).
The thread 0x573c has exited with code 0 (0x0).
Error List Output
Solution Explorer
Solution 'test3' (1 project)
test3
External Dependencies
References
kernel.cu
Solution Explorer Team Explorer Class View
Properties