تمرین ششم

تینا صداقت 9331044

درهر بلوک از روش درخت استفاده می‌شود. باید بتوانیم چندین بلوک از نخها داشته باشیم تا بتوان آرایه‌های بزرگ را پردازش کرد و همه‌ی پردازنده‌های GPU را مشغول نگه داشت و هر بلوک یک قسمت از پردازش آرایه را انجام ‌دهد. حال اینکه چطور این نتیجه‌های جزئی را حساب کنیم. اگر بتوانیم همه‌ی بلوکهای نخها را سنکرون کنیم: باید یک global sync بعد از تولید نتیجه هر بلوک داشته باشیم و هر وقت همه‌ی بلوکها به آن sync رسیدند به طور بازگشتی ادامه دهیم. اما کودا یک global synchronization ندارد به دلیل اینکه هزینه ساخت آن در سخت افزار زیاد است و برنامه نویس را مجبور می‌کند که تعداد کم تری بلوک داشته باشد تا از deadlock جلوگیری کند.

راه حل: به چندین کرنل تجزیه کنیم:

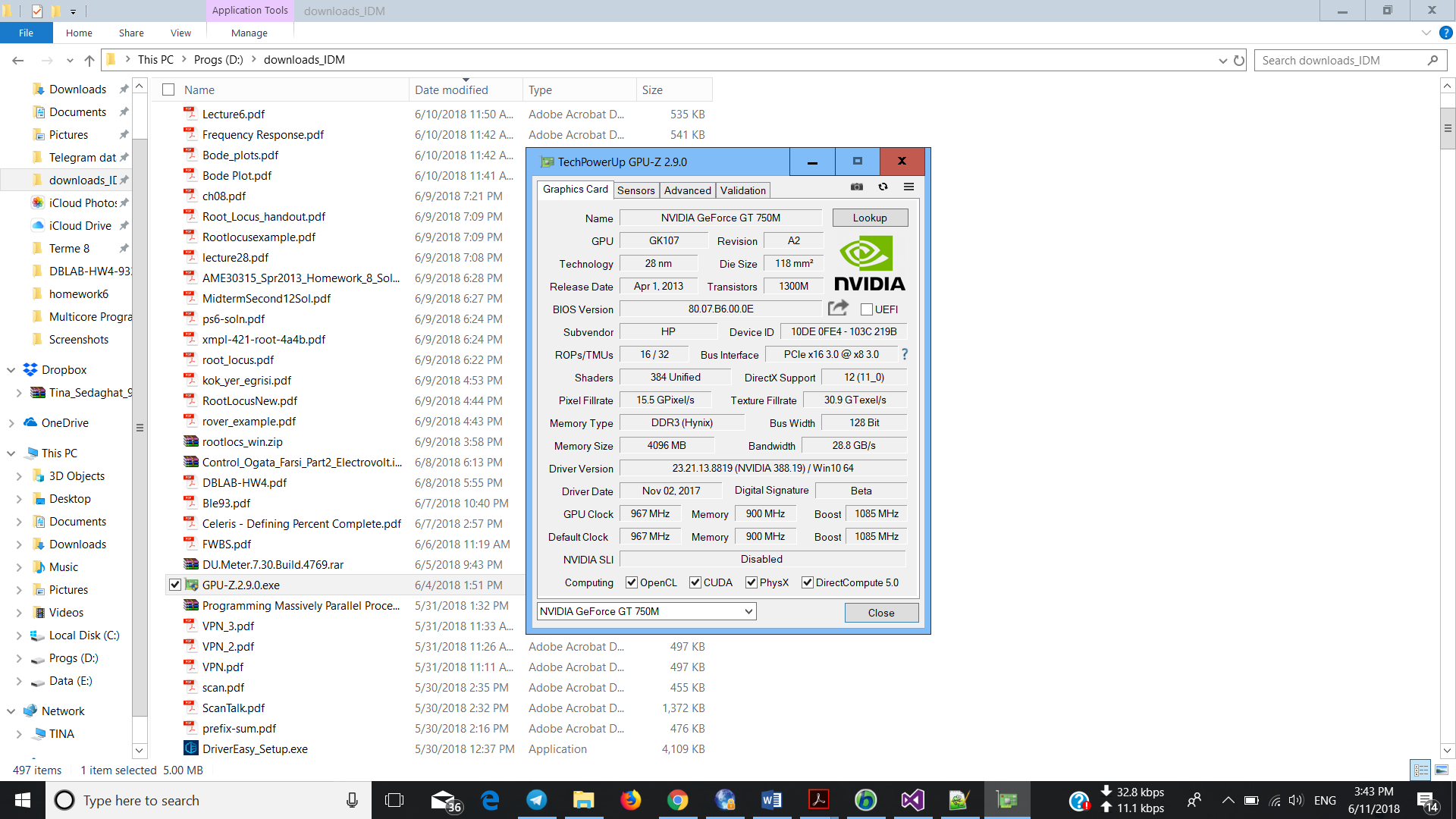
* که در آن اجرا شدن تابع کرنل خودش یک نقطه‌ی همگام سازی است.
* Overhead سخت‌افزار و نرم‌افزار قابل چشم پوشی است.

تجزیه به چندین کرنل باعث جلوگیری از global sync می‌شود. در خصوص reduction ها کد برای همه‌ی مراحل یکسان است.( فراخوانی کرنل بازگشتی)

هدف بهینه سازی:

* باید تلاش کنیم تا به ماکسیمم کارایی GPU برسیم.
* از متریک درست استفاده کنیم:
  + GFLOP/s: برای کرنل‌های محاسباتی
  + Bandwidth: برای کرنل‌های حافظه‌ای
* Reduction ها شدت محاسباتی خیلی کمی دارند.
  + یک flop برای لود کردن یک المان(پهنای باند بهینه)
* بنابراین باید برای پهنای باند زیاد تلاش کنیم.
* برای این تمرین از GeForce GT 750M استفاده شده است.
  + 384-bit memory interface و 900MHz DDR3

957 \* 106 \* (384/8) \* 2 / 109 = 91 GB/s



ابتدا کد سریال را چندین بار اجرا کردم و میانگین زمان سریال را از روی نتایج به دست آمده حساب کردم:

5004 , 3006 , 2704,7258,2767,5466 => 4367

Reduction#1: آدرس‌دهی برگ برگ شده:



در این الگوریتم هر نخ یک المان از آرایه‌ی global را در آرایه‌ی shared کپی می‌کند. در مرحله اول نخ‌هایی با اندیس‌های زوج دوتا دوتا عمل جمع را انجام می‌دهند و در خانه با اندیس کمتر قرار می‌دهند. در مرحله دوم همین کار با نخ‌های با مضرب 4 انجام می‌شود. در مرحله بعد ضریب 8 ...

نتیجه‌ی محاسبات هر بلوک در g\_outData[blockIdx.x] ذخیره می‎‌شود. بسته به تعداد بلوک‌هایی که داریم باید تعیین کنیم که چند بار تابع کرنل لانچ شود.

\_\_global\_\_ void reduce0(int \*g\_inData, int \*g\_outData)

{

extern \_\_shared\_\_ int sdata[];

// each thread loads one element from global to shared mem

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x \* blockDim.x + threadIdx.x;

sdata[tid] = g\_inData[i];

\_\_syncthreads();

// do reduction in shared mem

for (unsigned int s = 1; s < blockDim.x; s \*= 2) {

if (tid % (2 \* s) == 0) {

مشکل: branch های مختلف زیاد که کارایی را پایین می‌آورد.

sdata[tid] += sdata[tid + s];

}

\_\_syncthreads();

}

// Write the result for this block to global mem

if (tid == 0) {

g\_outData[blockIdx.x] = sdata[0];

}

}

یک تابع int reduction(int \*a) با ورودی پوینتر به شروع آرایه a داریم. در این آرایه اول پوینتر های ورودی و خروجی را تعریف می‌کنیم و برای آنها در device حافظه در نظر می‌گیریم. سایز آرایه‌ها باید به اندازه N باشد یعنی: int memSize = N \* sizeof(int);

اندازه گرید و سایز بلوک ها:

int grid\_x = N / BLOCK\_SIZE;

int block\_x = BLOCK\_SIZE;

اینها را به این شکل می‌نویسیم تا بعدا در تابع کرنل بیایند.

dim3 gridDimensions(grid\_x, 1, 1);

dim3 blockDimensions(block\_x, 1, 1);

تایمر های Start و stop را تعریف می‌کنیم. و تایمر شروع را اجرا می‌کنیم.

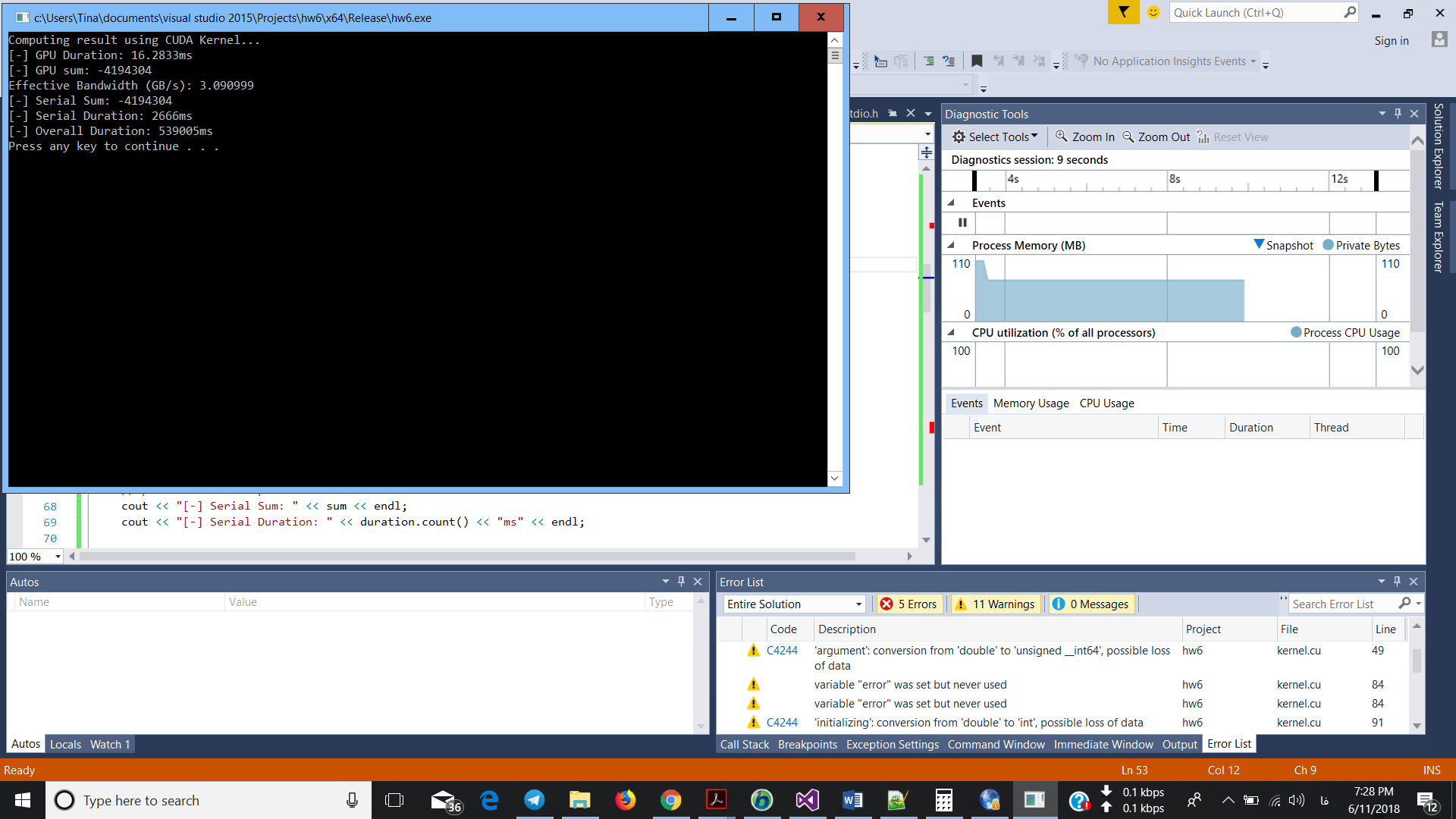
تابع کرنل به صورت زیر تعریف می‌شود که ابعاد گرید و بلوک و اختصاص حافظه ی بلوک در تعریف آن می‌آیند و همچنین ورودی d\_A و خروجی d\_Blocks است.

// Execute Kernels

reduce0 << <gridDimensions, blockDimensions, block\_x\*sizeof(int) >> > (d\_A, d\_Blocks);

اما اگر تعداد بلوک ها زیاد باشد باید این تابع کرنل چند بار صدا زده شود با این تفاوت که در دفعه های بعد ورودی d\_Blocks می‌شود. (reduction)

تایمر خروجی را فرا میخوانیم و زمان سپری شده را نمایش می‌دهیم. خروجی تولید شده از کرنل‌ها را در آرایه a در host کپی می‌کنیم و نشان می‌دهیم.



Reduction#2: آدرس‌دهی برگ برگ شده:

کد مثل قبل است با این تفاوت که در قسمتی که reduction در shared memory انجام می‌شود این کد را می‌نویسیم:

for (unsigned int s = 1; s < blockDim.x; s \*= 2) {

int index = 2 \* s \* tid;

if (index < blockDim.x) {

sdata[index] += sdata[index + s];

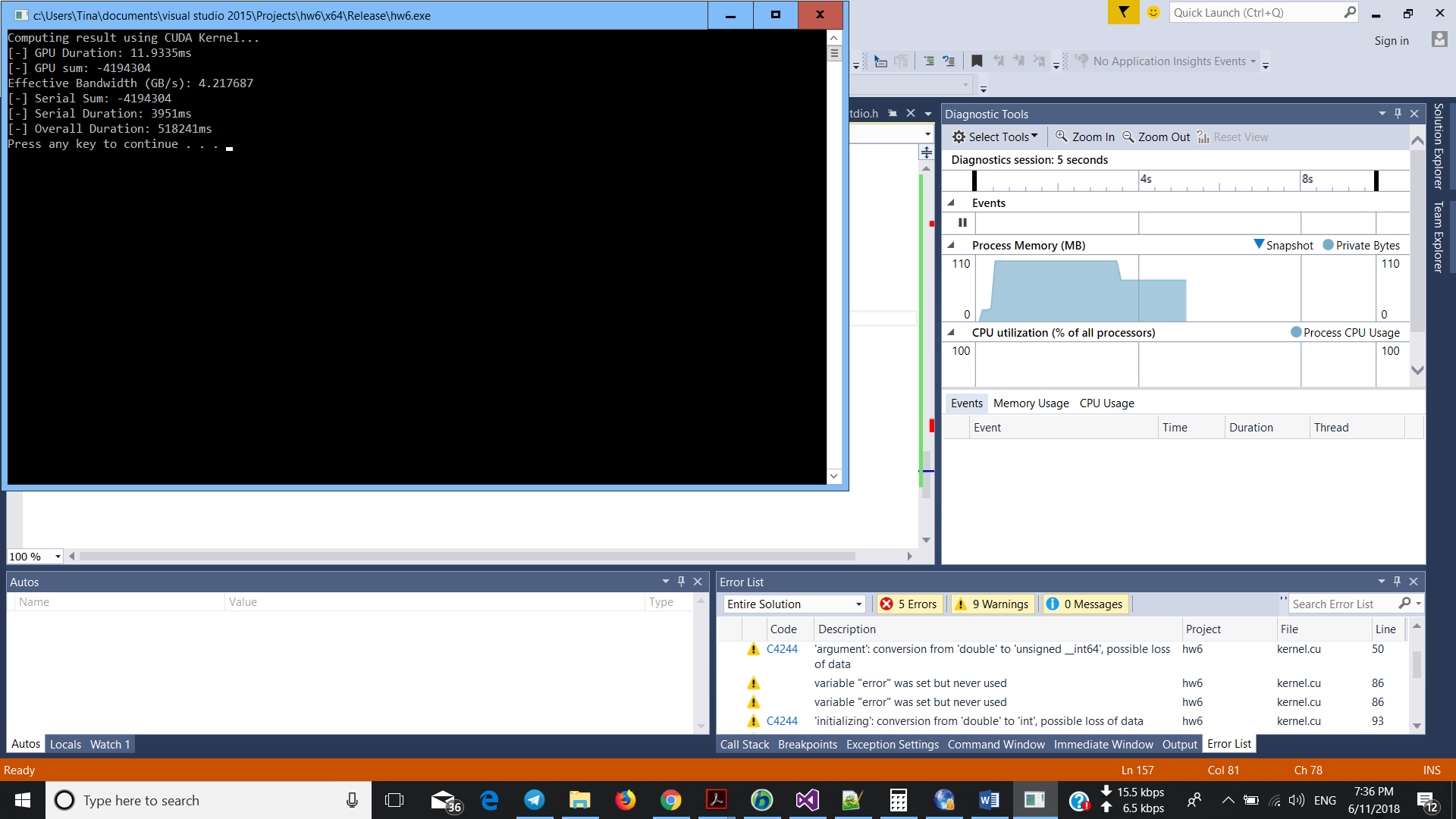
}

\_\_syncthreads();

}

به جای branchهای مختلف، index را به فرم int index = 2 \* s \* tid; تعریف می‌کنیم.

مشکل جدیدی به وجود می‌آید که همان shared memory bank conflict است.



Reduction#3: آدرس‌دهی سریالی:

\_\_global\_\_ void reduce3(int \*g\_inData, int \*g\_outData)

{

extern \_\_shared\_\_ int sdata[];

// Identification

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x \* blockDim.x + threadIdx.x;

// Fill up the shared memory

sdata[tid] = g\_inData[i];

// Fixed bank conflicts with sequential addressing

\_\_syncthreads();

for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {

if (tid < s)

sdata[tid] += sdata[tid + s];

\_\_syncthreads();

}

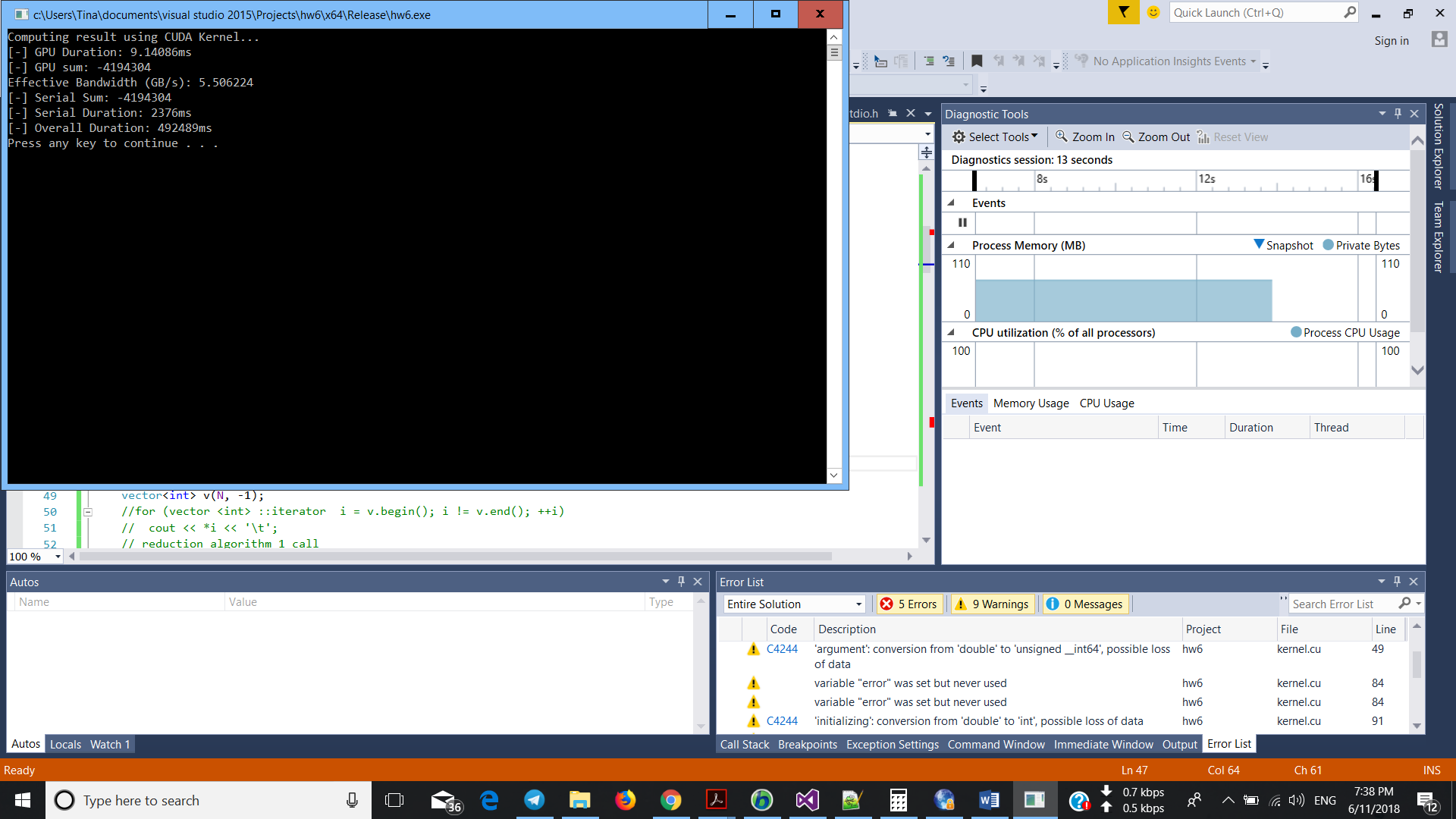
// Write the result for this block to global memory

if (tid == 0) {

g\_outData[blockIdx.x] = sdata[0];

}

}



**نخ های بیکار:**

**for (unsigned int s=blockDim.x/2; s>0; s>>=1) {**

**if (tid < s) {**

**sdata[tid] += sdata[tid + s];**

**}**

**\_\_syncthreads();**

**}**

در تکرار حلقه اول نصف نخ ها بیکار هستند!

Reduction#4: جمع اول هنگام لود:

تعداد بلوک ها را نصف می‌کنیم (64) و یک load تکی می‌گذاریم. یعنی به جای:

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x \* blockDim.x + threadIdx.x;

sdata[tid] = g\_inData[i];

\_\_syncthreads();

کد زیر را می‌نویسیم:

// perform first level of reduction,

// reading from global memory, writing to shared memory

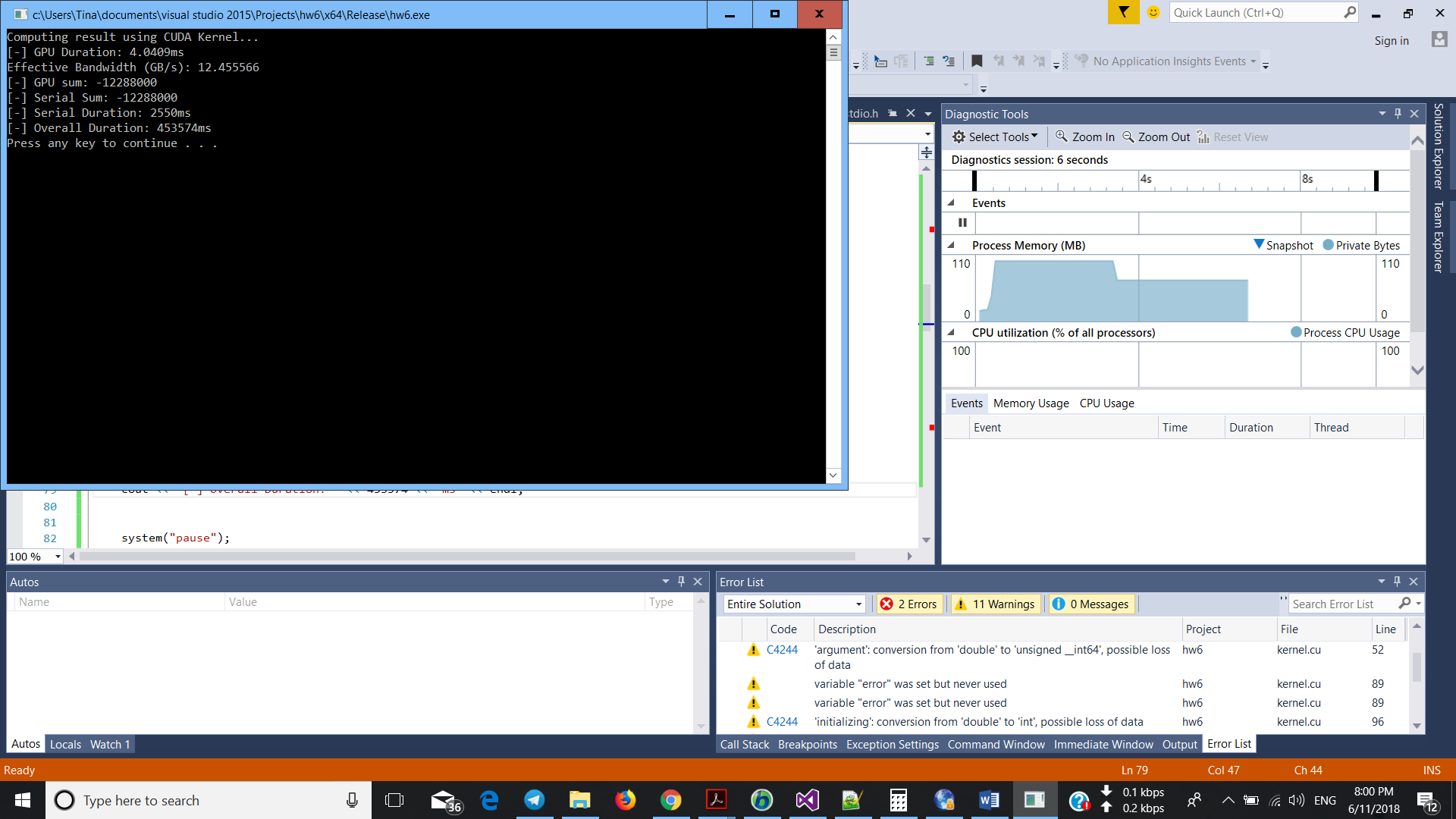
unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*(blockDim.x \* 2) + threadIdx.x;

// Fill up the shared memory

sdata[tid] = g\_inData[i] + g\_inData[i + blockDim.x];

\_\_syncthreads();



Reduction#5: آخرین warp را از حلقه بیرون می‌آوریم:

گلوگاه دستورها:

در 17GB/s ما خیلی دور از محدوده ی پهنای باند هستیم و می‌دانیم که Reduction شدت محاسباتی کمی دارد. پس احتمالا یک گلوگاه سربار دستور وجود دارد. دستورات فرعی ( غیر از load و store و محاسبات هسته) یعنی محاسبات آدرس‌ها و سربار حلقه

راه حل: حلقه را باز کنیم.

آخرین warp را باز کنیم:

همینطور که Reduction جلو می‌رود، تعداد نخ های فعال کم می‌شود. وقتی s<=32 شود، فقط یک warp باقی می‌ماند. دستورات در warp، single instrudtion multiple data هستند. یعنی وقتی s<=32 باشد لزومی به استفاده از \_\_ synchthread() نداریم. به بخش if(tid<s) هم نیازی نداریم چون هیچ کاری را سیو نمی‌کند. پس 6 تا تکرار آخر را از حلقه بیرون می‌آوریم.

for (unsigned int s = blockDim.x / 2; s>32; s >>= 1)

{

if (tid < s)

sdata[tid] += sdata[tid + s];

\_\_syncthreads();

}

if (tid < 32)

{

sdata[tid] += sdata[tid + 32];

sdata[tid] += sdata[tid + 16];

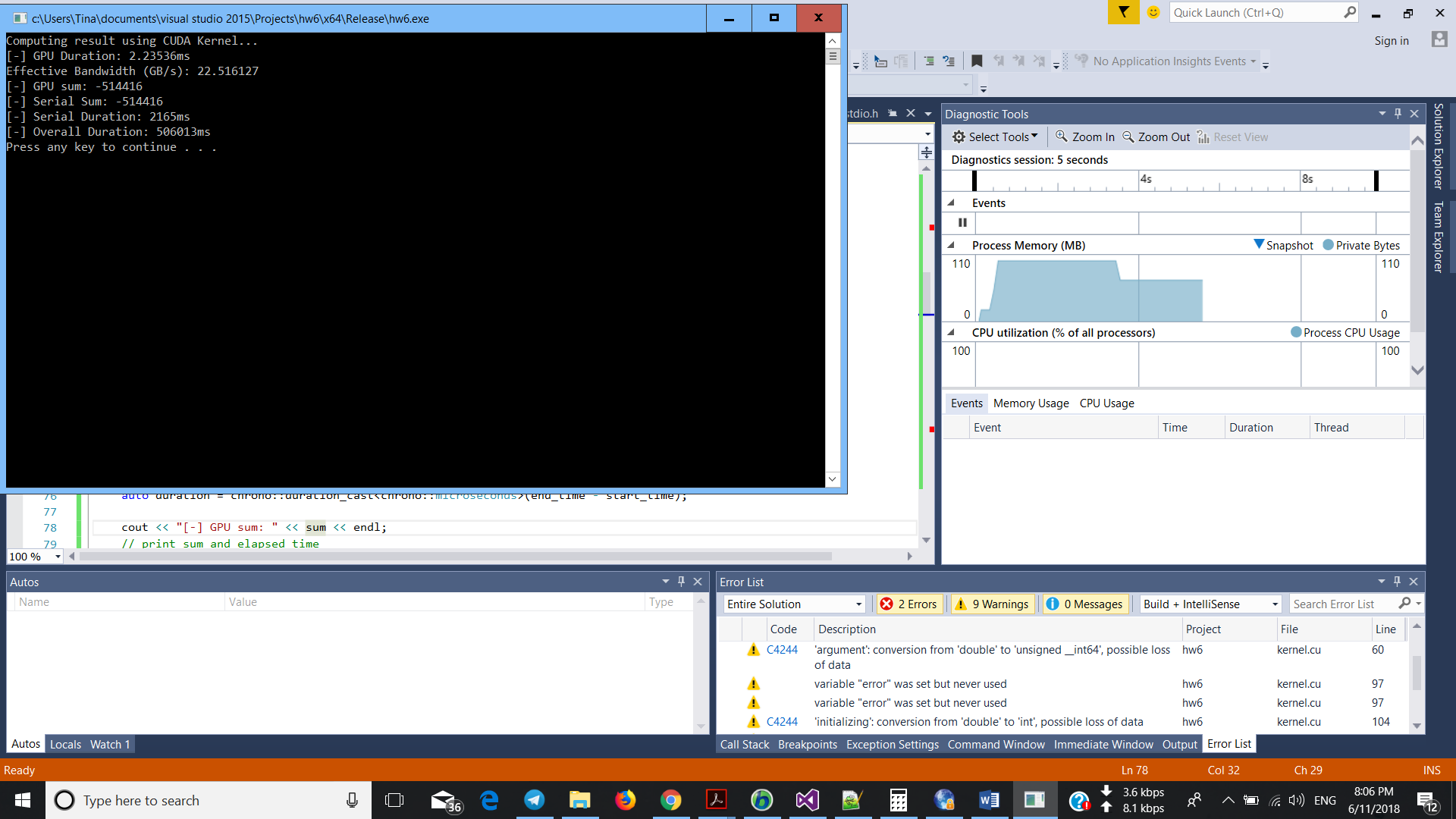
sdata[tid] += sdata[tid + 8];

sdata[tid] += sdata[tid + 4];

sdata[tid] += sdata[tid + 2];

sdata[tid] += sdata[tid + 1];

}



محاسبه bandwidth:

Bandwidth تئوری: (peak bandwidth)

Memory clock rate: 957 MHz

Memory interface: 384 bit

BWtheoretical = 957 \* 106 \* (384/8) \* 2 / 109 = 91 GB/s

تقسیم بر 8 برای اینکه بیت را به بایت تبدیل کنیم. دو سرعت داده دو برابر است ضربدر دو هم می‌شود. برای رسیدن به GB/s باید تقسیم بر 109 کنیم.

Bandwidth موثر:

BWEffective = (RB + WB) / (t \* 109)

RB تعداد بایت های خوانده شده از کرنل، و WB تعداد بایتهای نوشته شده در کرنل و t زمان سپری شده بر حسب ثانیه است.

کد محاسبه bandwidth:

printf("Effective Bandwidth (GB/s): %f \n", N \* 4 \* 3 / elapsed\_time / 1e6);

N\*4 تعداد بایت های خوانده شده یا نوشته شده از هر آرایه است. 3 هم نمایش دهنده خواندن d\_A و خواندن و نوشتن d\_Blocks است.

نتایج:

با توجه به pdf داده شده، من هم سایز بلوک ها را 128 گرفتم.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| speedup | bandwidth | زمان کل (محاسبات + انتقال داده‌ها) | زمان محاسبات |  |
| 4367/16.28=272.93 | 3.09 | 539005 | 16.2833 | Kernel1  Interleaved addressing with divergent branching |
| 4367/11.933=365.9 | 4.21 | 518241 | 11.9335 | Kernel2 interleaved addressing with bank conflicts |
| 4367/9.14=477 | 5.50 | 492489 | 9.14086 | Kernel 3 sequential addressing |
| 4367/4.04=1072 | 12.45 | 45327 | 4.0409 | Kernel4 first add during global load |
| 4367/2.33= 1858 | 22.51 | 43741 | 2.3553 | Kernel5 unroll last warp |