CSE 591: GPU Programming

Using CUDA in Practice

Klaus Mueller

Computer Science Department

Stony Brook University

Code examples from Shane Cook "CUDA Programming"

Lazy Evaluation

Related to:

- score boarding
- load and store

Consider the following code:

```
int sum=0;
for (int i=0; i< 128; i++)
{
 sum += src_array[i];
}
```

- is this efficient code?
- each operation is dependent on the one before
- compute new index and address, then load the data, add to sum
- not much memory latency hiding
- can we do it better?

Lazy Evaluation

Splitting into four independent sums:

```
int sum=0;
int sum1=0, sum2=0, sum3=0, sum4=0;
for (int i=0; i< 128; i+=4)
{
 sum1 += src_array[i];
 sum2 += src_array[i+1];
 sum3 += src_array[i+2];
 sum4 += src_array[i+3];
}
sum = sum1 + sum2 + sum3 + sum4;
```

- is this better?

Lazy Evaluation

How about this?

```
int sum=0;
int sum1=0, sum2=0, sum3=0, sum4=0;
for (int i=0; i< 128; i+=4)
{
 const int a1 = src_array[i];
 const int a2 = src_array[i+1];
 const int a3 = src_array[i+2];
 const int a4 = src_array[i+3];
 sum1 += a1;
 sum2 += a2;
 sum3 += a3;
 sum4 += a4;
}
sum = sum1 + sum2 + sum3 + sum4;
```

- compare this with the eager evaluation model of CPUs
- CPUs will stall at every read
- GPUs will delay the stall until actual use
- then will rapidly switch a thread

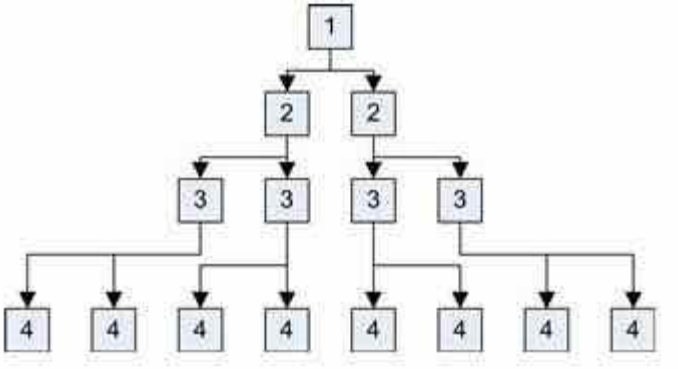
Recursion on GPUs

Leads to branch “explosion”

- less appropriate for GPUs since number of threads is fixed before kernel invocation
- very recent Kepler high-end K20 GPUs support dynamic parallelism

How else to implement it?

- use iterative methods instead of branch generation
- recall binary search in Sample Sort
- can also invoke new kernels at every level
- see book for more detail



CUDA Ballot Function

For compute 2.0 and above

- unsigned int __ballot(int *predicate*)
- when *predicate* evaluates to TRUE the function returns a value with the Nth bit set
- *N = threadIdx.x*
- C-implementation of non-atomic version (will work for all compute levels):

```
__device__ unsigned int __ballot_non_atom(int predicate)
{
 if (predicate != 0)
  return (1 << (threadIdx.x % 32));
 else
  return 0;
}
```

Intrinsics

What are (CUDA) intrinsics?

- a function known by the compiler that directly maps to a sequence of one or more assembly language instructions.
- are inherently more efficient than called functions because no calling linkage is required
- make the use of processor-specific enhancements easier because they provide a CUDA language interface to assembly instructions.
- in doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data

Other CUDA Intrinsics

AtomicOr:

- int atomicOr(int **address*, int *val*)
- reads the value pointed to by *address*
- performs a bitwise OR with *val*
- returns the result back in *address*

Example use:

```
volatile __shared__ u32 warp_shared_ballot[MAX_WARPS_PER_BLOCK];
// Current warp number - divide by 32
const u32 warp_num = threadIdx.x >> 5;
atomicOr( &warp_shared_ballot[warp_num],
                              __ballot(data[tid] > threshold) );
```

- what does this function do?
- will set a bit for every thread for which *data[tid]> threshold*
- what useful computation can this enable?

Other CUDA Intrinsics

Enables very fast counting given a condition (predicate)

- extend it using the __popc function
- returns the number of bits set within a 32-bit parameter
- can be used to accumulate a block-based sum for all warps in the block

```
atomicAdd(&block_shared_accumulate,
          __popc(warp_shared_ballot[warp_num]));
```

- can accumulate for a given CUDA block the number of threads in every warp that had the condition we used for the predicate set
- in this example, the condition is that the data value was larger than a threshold
- then must sum all the values across the blocks

See the book for more detail and applications

Profiling

We shall use Sample Sort as a running example

Major parameters

- number of samples
- number of threads

Explore the possible search space

- double the number of samples per iteration
- use 32, 64, 128, or 256 threads

Example Configuration (1)

```
ID:0 GeForce GTX 470: Test 16 - Selecting 16384 from 1048576 elements
using 64 blocks of 256 threads
  Num Threads:                              32     64    128    256
  Select Sample Time- CPU:    0.56  GPU: 0.56   0.19   0.06   0.38
  Sort Sample Time -  CPU:    5.06  GPU: 5.06   5.06   5.06   5.06
  Count Bins Time -   CPU: 196.88  GPU: 7.28   4.80   4.59   4.47
  Calc. Bin Idx Time- CPU:    0.13  GPU: 1.05   0.71   0.70   0.98
  Sort to Bins Time - CPU: 227.56  GPU: 7.63   4.85   4.62   4.49
  Sort Bins Time -    CPU:   58.06  GPU:64.77  47.88  60.58  54.51
  Total Time -        CPU: 488.25  GPU:86.34  63.49  75.61  69.88
  QSORT Time -        CPU: 340.44

ID:0 GeForce GTX 470: Test 16 - Selecting 32768 from 1048576 elements
using 128 blocks of 256 threads
  Num Threads:                              32     64     128    256
  Select Sample Time- CPU:    0.63  GPU: 0.63   0.63   0.75   0.38
  Sort Sample Time -  CPU:   10.88  GPU:10.88  11.06  10.63  10.69
  Count Bins Time -   CPU:  222.19  GPU: 7.85   5.51   5.39   5.22
  Calc. Bin Idx Time- CPU:    0.19  GPU: 1.76   0.99   0.98   1.16
  Sort to Bins Time - CPU:  266.06  GPU: 8.19   5.53   5.40   5.24
  Sort Bins Time -    CPU:   37.38  GPU:57.57  39.40  44.81  41.66
  Total Time -        CPU: 537.31  GPU:86.88 63.13  67.96  64.35
  QSORT Time -        CPU: 340.44
```

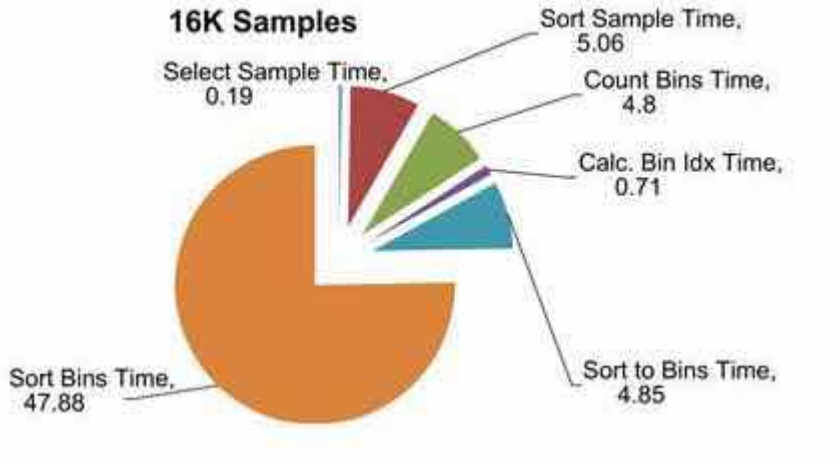
Example Configuration (2)

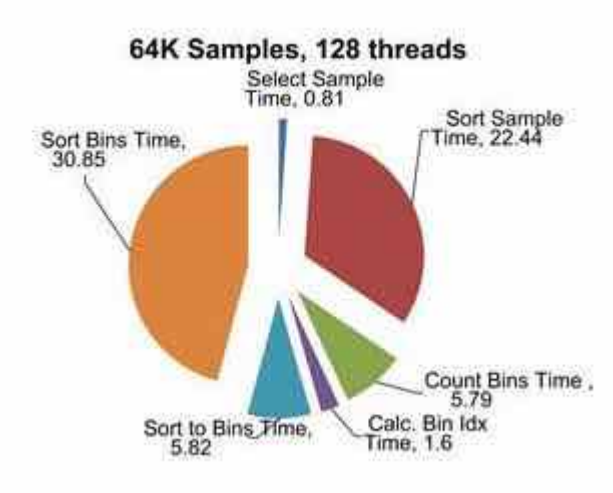
```
ID:0 GeForce GTX 470: Test 16 - Selecting 32768 from 1048576 elements
using 128 blocks of 256 threads
Num Threads:                              32      64     128     256
Select Sample Time- CPU:     0.63  GPU: 0.63    0.63    0.75    0.38
Sort Sample Time -  CPU:    10.88  GPU:10.88   11.06   10.63   10.69
Count Bins Time -   CPU:   222.19  GPU: 7.85    5.51    5.39    5.22
Calc. Bin Idx Time- CPU:     0.19  GPU: 1.76    0.99    0.98    1.16
Sort to Bins Time - CPU:   266.06  GPU: 8.19    5.53    5.40    5.24
Sort Bins Time -    CPU:    37.38  GPU:57.57   39.40   44.81   41.66
Total Time -        CPU: 537.31  GPU:86.88 63.13 67.96 64.35
QSORT Time -        CPU: 340.44

ID:0 GeForce GTX 470: Test 16 - Selecting 65536 from 1048576 elements
using 256 blocks of 256 threads
Num Threads:                              32      64     128     256
Select Sample Time- CPU:     1.00  GPU: 1.00    0.88    0.81    0.94
Sort Sample Time -  CPU:    22.69  GPU:22.69   22.50   22.44   23.00
Count Bins Time -   CPU:   239.75  GPU: 8.32    5.90    5.79    5.62
Calc. Bin Idx Time- CPU:     0.25  GPU: 1.49    1.98    1.60    1.65
Sort to Bins Time - CPU:   300.88  GPU: 8.69    5.97    5.82    5.67
Sort Bins Time -    CPU:    24.38  GPU:52.32   33.55   30.85   32.21
Total Time -        CPU: 588.94  GPU:94.50 70.78 67.32 69.09
QSORT Time -        CPU: 340.44
```

Hard to See? What We Do?

Visualize it!





Now differences and trends can be easily observed

For 16K samples

- ¾ of the time is used for sorting, ¼ for setting up the sample sort

For 64K samples

- suddenly the time to sort the sample jumps to around 1/3
- much variability on number of samples and the number of threads

NVIDIA Parallel Nsight

Free debugging and analysis tool

- incredibly useful for identifying bottlenecks
- look for “New Analysis Activity” feature

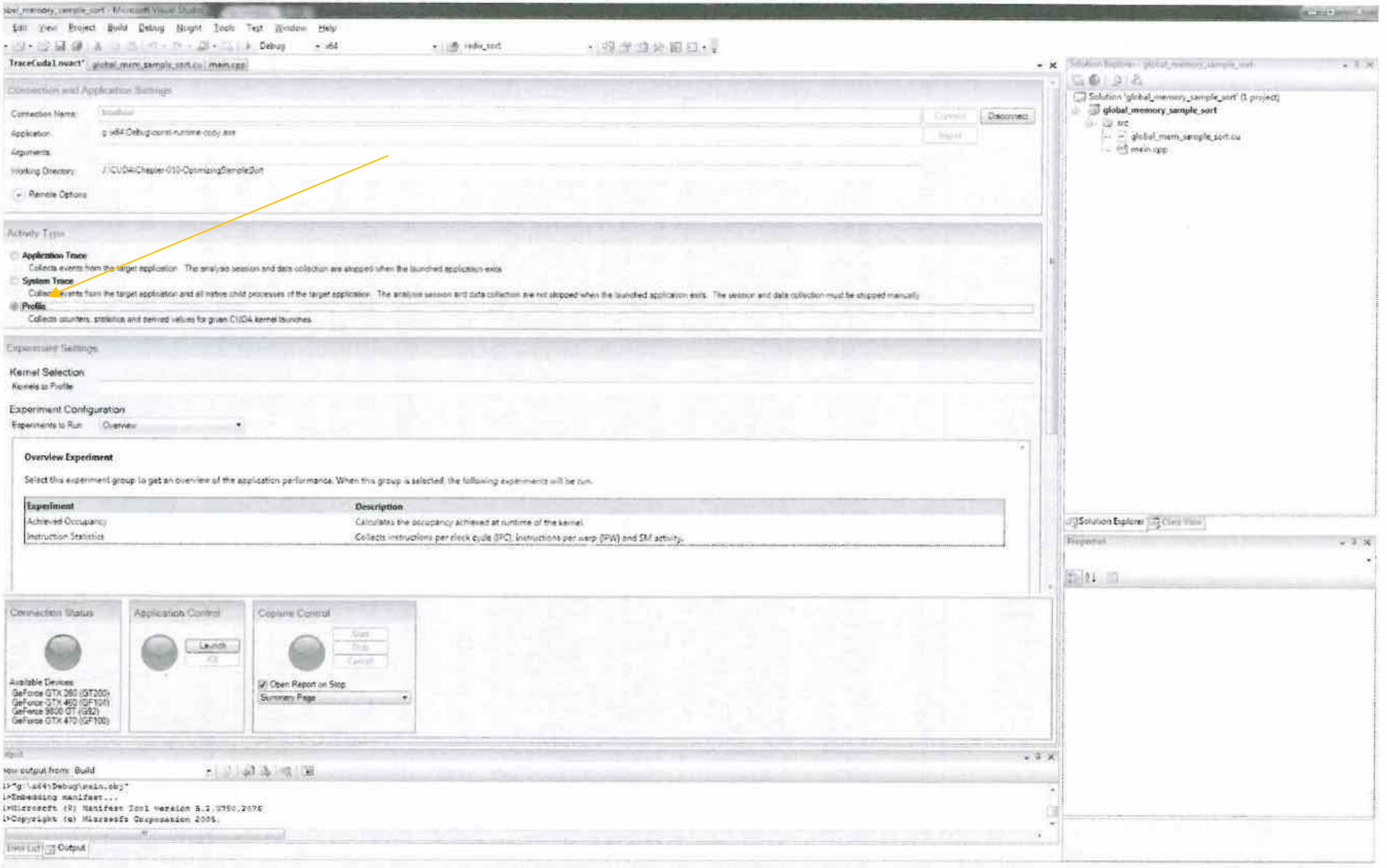
We shall focus on middle case

- 32 K samples

Choose “Profile” activity type (next slide)

- by default this will run a couple of experiments
- “Achieved Occupancy” and “Instruction Statistics”
- produces a summary
- at the top of the summary page is a dropdown box
- select “CUDA Launches” to get useful information

Parallel Nsight Launch Options



Parallel Nsight Analysis

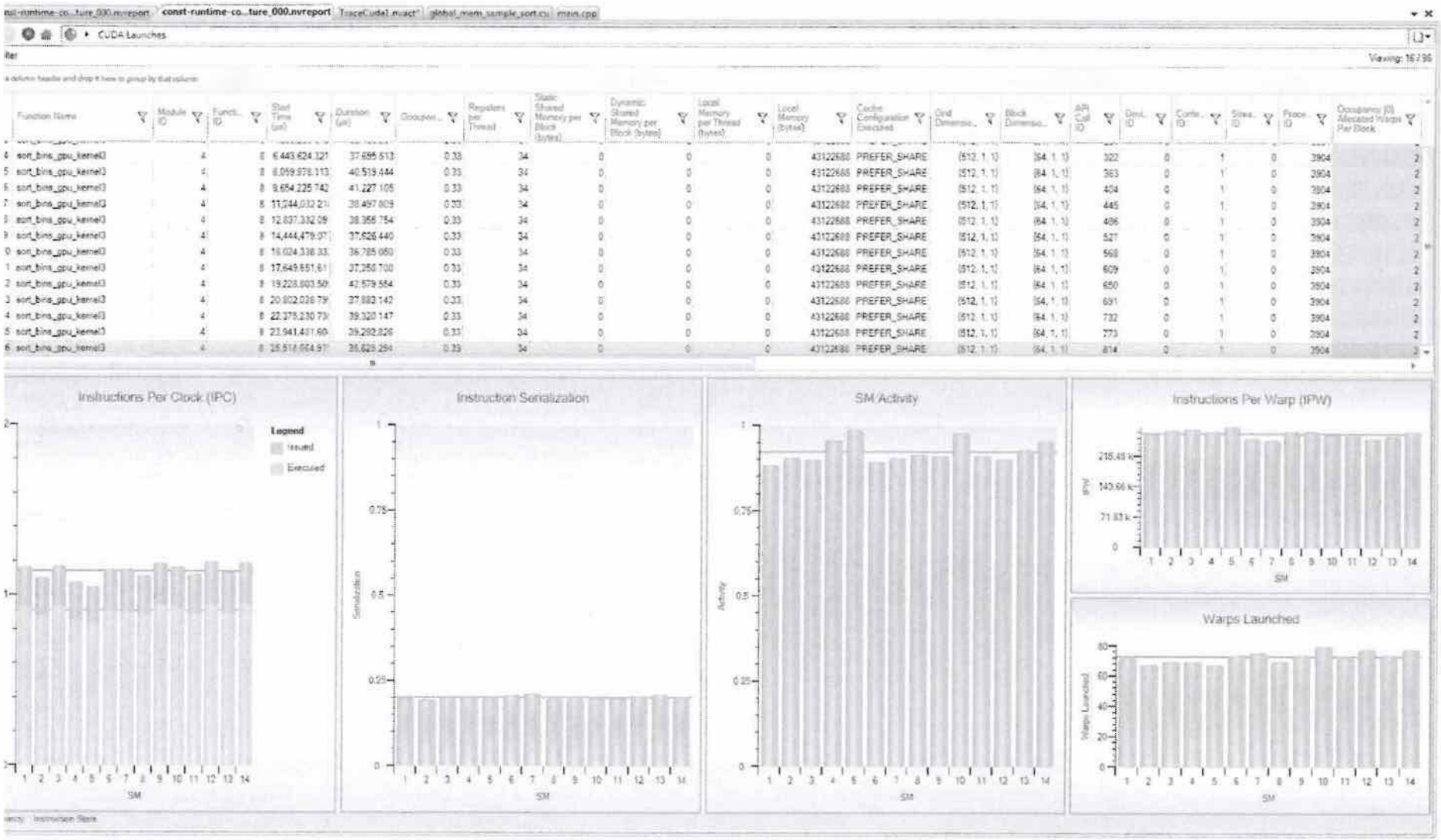
Analysis

Factors limiting occupancy are red-colored

Occupancy

- block limit (8 blocks) per device is limiting the maximum number of active warps on the device
- not enough warps means less memory latency can be hidden
- we launched around 16 warps (but could have up to 48)
- this yields 1/3 of the maximum occupancy
- so should we increase the number of warps (and threads)?
- turns out that this will have the opposite effect (see measured results)

More Parallel Nsight Analysis



Analysis

Some instructions were reissued (blue bars)

- due to serialization
- these are threads not able to execute as a complete warp
- due to divergent program flow, uncoalesced memory access, conflicts (shared memory, atomics)

Distribution of work across SMs is uneven

- some have more warps than others
- some also take longer due to uneven amount of work
- 14 SMs 512 blocks of 64 threads
- expect 36 blocks (72 warps)/SM
- but some get 68 warps
- others get 78 warps

Change in Parameters

When moving to 256 threads/block

- variability in issued vs. executed instructions grows
- number of scheduled blocks goes from eight to three due to the 34 registers allocated per thread
- nevertheless, the number of scheduled warps goes to 24 (from 16)
- this gives a 50% occupancy rate

Can we increase occupancy?

- we need to limit registers use via compiler option (set max to 32)
- this leads to a register use of 18
- occupancy is now 100%
- but now execution time grows from 63ms to 86ms – why?
- because now registers are pushed to local storage (L1 cache, or global memory for earlier devices)

Can we gain performance in a different way?

- can achieve a speedup of 4.4 over CPU-based QSort
- see book for details (mainly by better cache utilization)