CSE 591: GPU Programming

Optimizing Your Application

Klaus Mueller

Computer Science Department

Stony Brook University

Code examples from Shane Cook "CUDA Programming"

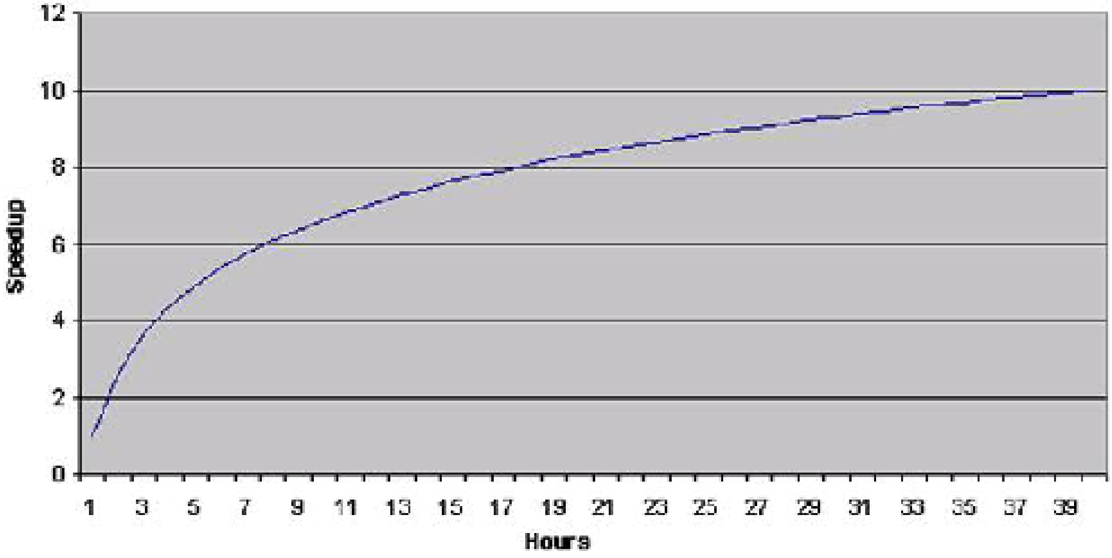
Code Development Time

Speed-ups of a factor of two:

- may just obtain it by upgrading hardware
- may not need a GPU solution

Before laborious optimization consider

- development time is expensive



- also recall slides on Amdahl’s law from an earlier lecture

Dependencies

Can take two forms:

- one element is dependent on one or more elements around it
- in a multi-pass program, a dependency from one pass to the next

```
extern int a,c,d;
extern const int b;
extern const int e;

void some_func_with_dependencies(void)
{
 a = b * 100;
 c = b * 1000;
 d = (a + c) * e;
}
```

- *a* and *c* have a dependency on *b*
- *d* has a dependency on *a* and *c*
- which can be computed in parallel?

- *a* and *b* must complete before *d* can be computed
- this can cause delays

Latency (and Dependency) Hiding

We heard about warp switching

- are there other ways to do this?

Yes – insert (overlap with) independent instructions

- this hides arithmetic execution latency

```
extern int a,c,d,f,g,h,i,j;
extern const int b;
extern const int e;

void some_func_with_dependencies(void)
{
 a = b * 100;
 c = b * 1000;

 f = b * 101;
 g = b * 1001;

 d = (a + c) * e;
 h = (f + g) * e;

 i = d * 10;
 j = h * 10;
}
```

Loop Fusion (1)

Non-fused loop vs. fused

```
void loop_fusion_example_unfused(void)
{
 unsigned int i,j;

 a = 0;
 for (i=0; i<100; i++)  /* 100 iterations */
 {
  a + = b * c * i;
 }

 d = 0;
 for (j=0; j<200; j++)  /* 200 iterations */
 {
  d += e * f * j;
 }
}
```

```
void loop_fusion_example_fused_01(void)
{
 unsigned int i;   /* Notice j is eliminated */

 a = 0;
 d = 0;
 for (i=0; i<100; i++)  /* 100 iterations */
 {
  a += b * c * i;
  d += e * f * i;
 }

 for (i=100; i<200; i++)  /* 100 iterations */
 {
  d += e * f * i;
 }
}
```

- fused example saves 1/3 of the loop iterations (which are empty work)

Loop Fusion (2)

Non-fused loop vs. fused

```
void loop_fusion_example_unfused(void)
{
 unsigned int i,j;

 a = 0;
 for (i=0; i<100; i++)  /* 100 iterations */
 {
  a + = b * c * i;
 }

 d = 0;
 for (j=0; j<200; j++)  /* 200 iterations */
 {
  d += e * f * j;
 }
}
```

```
void loop_fusion_example_fused_02(void)
{
 unsigned int i;   /* Notice j is eliminated */

 a = 0;
 d = 0;
 for (i=0; i<100; i++)  /* 100 iterations */
 {
  a += b * c * i;
  d += e * f * i;
  d += e * f * (i*2);
 }
}
```

- completely eliminates the second loop and creates additional independent work in the loop

Some Words of Caution

In a GPU implementation

- loops would be parallel threads

Adding more work into a thread

- will decrease parallelism
- will expand register use

Also, try to avoid multi-pass algorithms

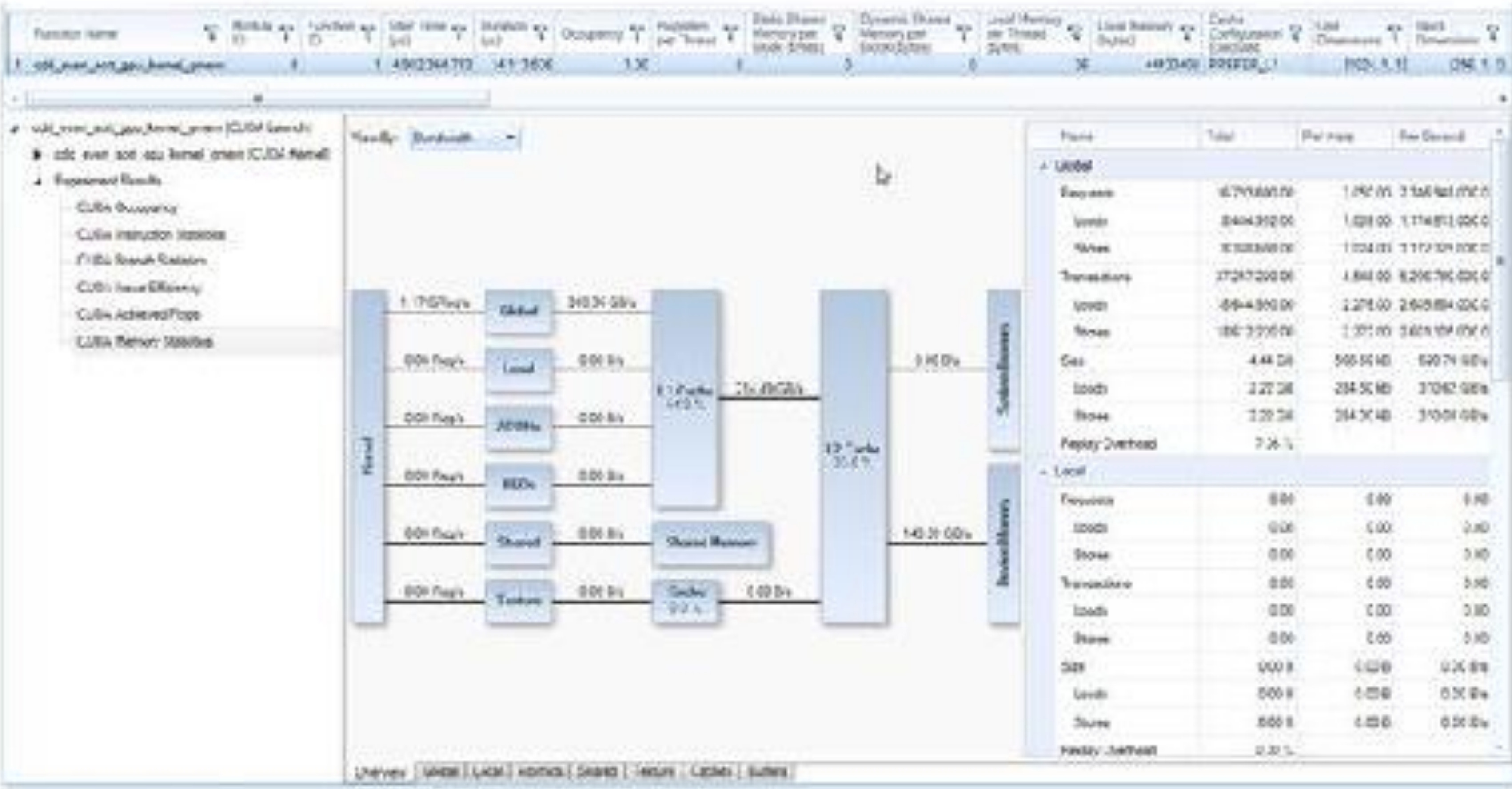
- they may require reading expensive transfers from and to slower memory
- a single pass will enable it all to be kept in shared memory

Profiling

Best way to find out where you spend you time optimizing

- find bottlenecks
- find occupancy and memory bandwidth
- find where code spends most of its time
- usually 20% of the code spends 80% of the time
- optimize these 20% (and use the profiler to find them)
- use NVIDIA Parallel Nsight

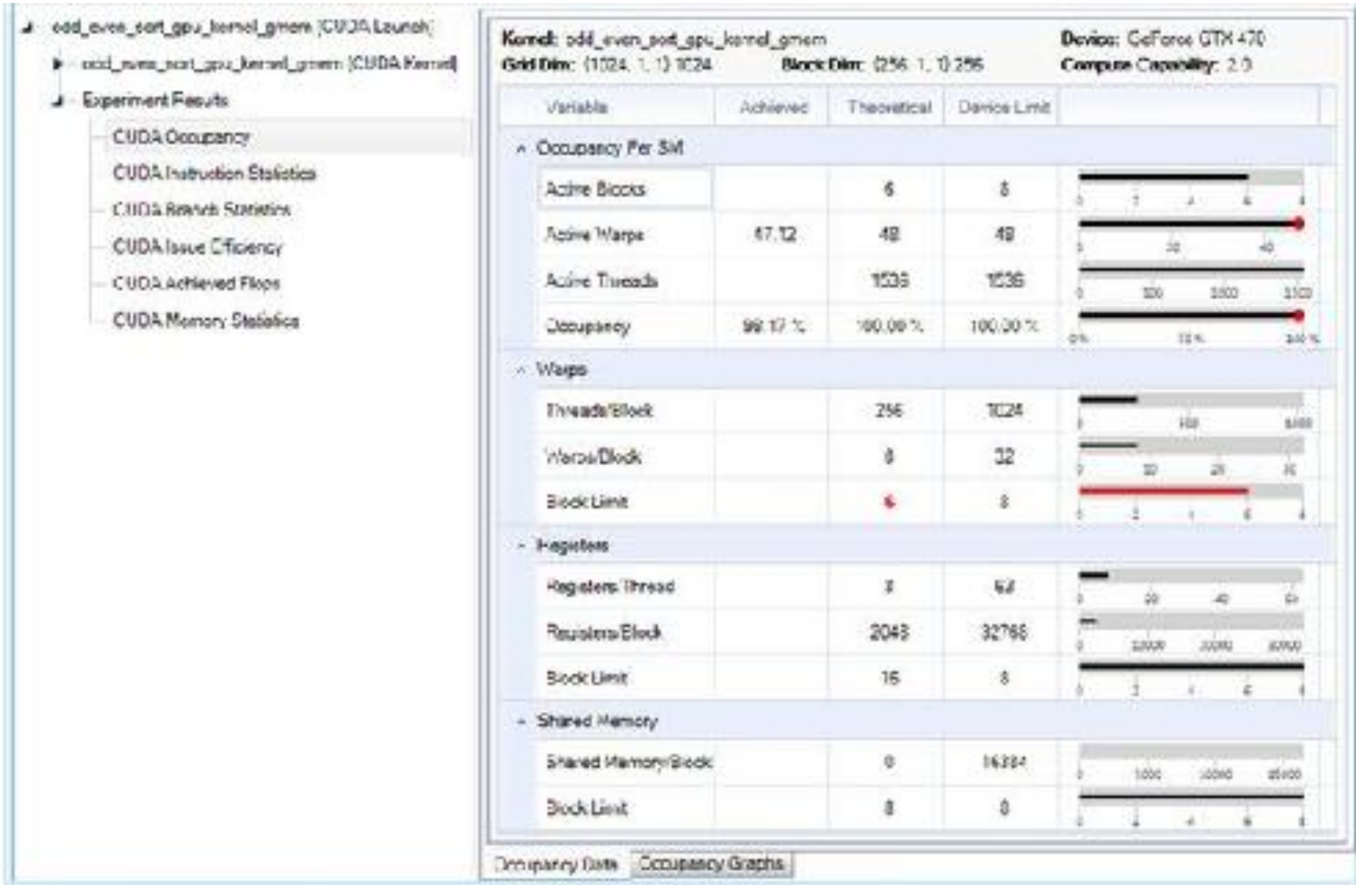
Parallel Nsight: Memory Overview



Observations

- 54% hit ratio in L1 achieves about 310 GB/s bandwidth to global memory (double than what is available)
- could lower the number of transactions for better coalescing

Parallel Nsight: Occupancy Rate



Observations:

- limiting factors will be highlighted in red (here: number of blocks/SM=6)
- via the graphs we see that # threads should be cut from 256 to 192
- this way we can get 8 blocks and so improve instruction mix

Parallel Nsight: Occupancy Rate (2)

Increasing # blocks improves occupancy from 98.17% to 98.22%

- just OK

But execution time drops from 14ms to just 10ms

- with 192 threads per block a smaller range of addresses is accessed
- this increases the locality of the accesses and this improves cache utilization
- the total number of memory transactions needed by each SM drops by about one-quarter and we see a proportional drop in execution time