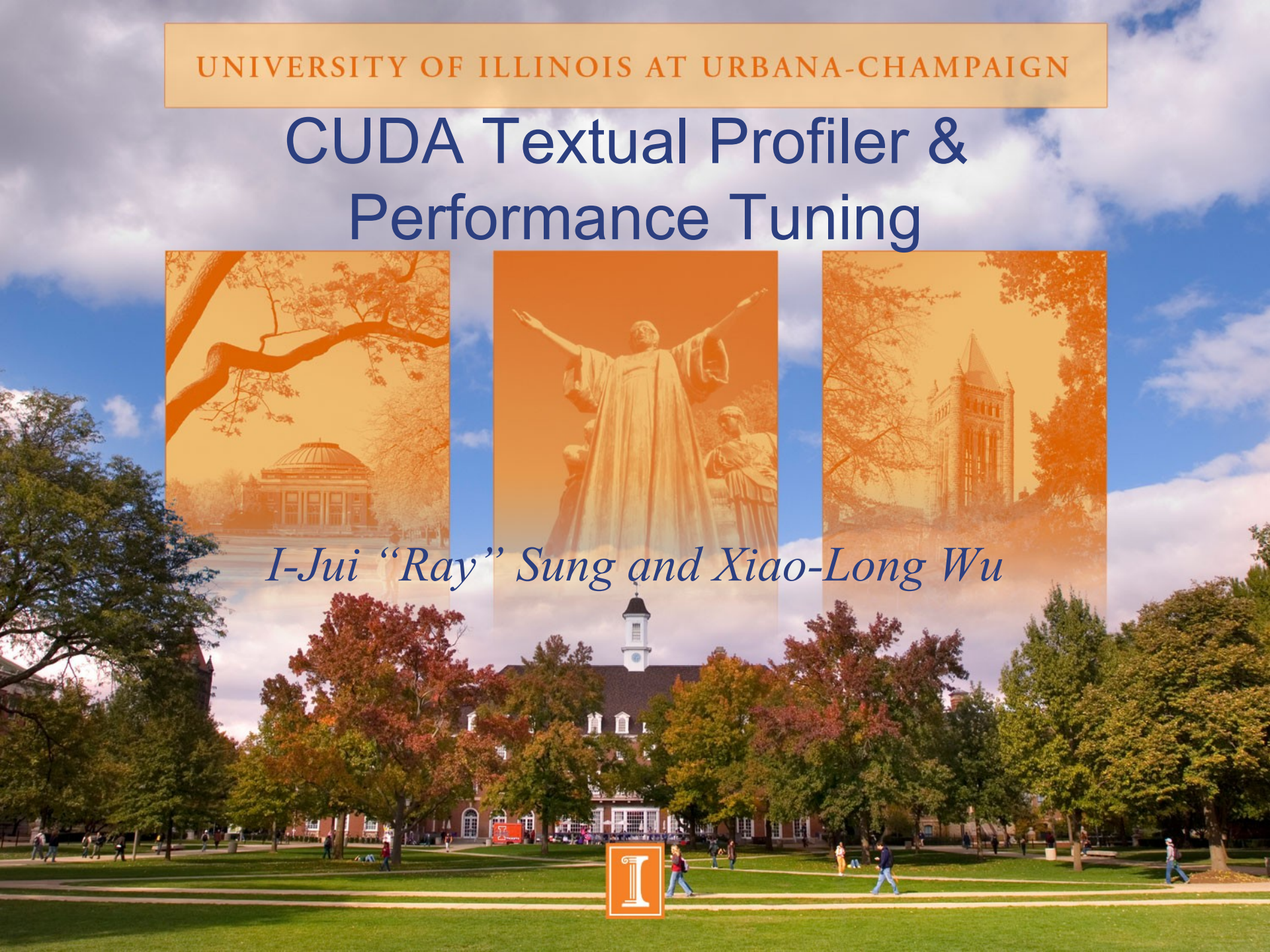


CUDA Textual Profiler & Performance Tuning

I-Jui "Ray" Sung and Xiao-Long Wu

Agenda

- CUDA Textual Profiler by I-Jui “Ray” Sung
- Performance Tuning by Xiao-Long Wu

The use of CUDA profiler

- The CUDA profile can be used when one wants to see the relative performance improvements of a particular optimization in terms of
  - # of Coalesced/uncoalesced global memory accesses
  - # of Shared memory bank conflicts
  - # of Diverging control flows

Enabling and Controlling Profiling

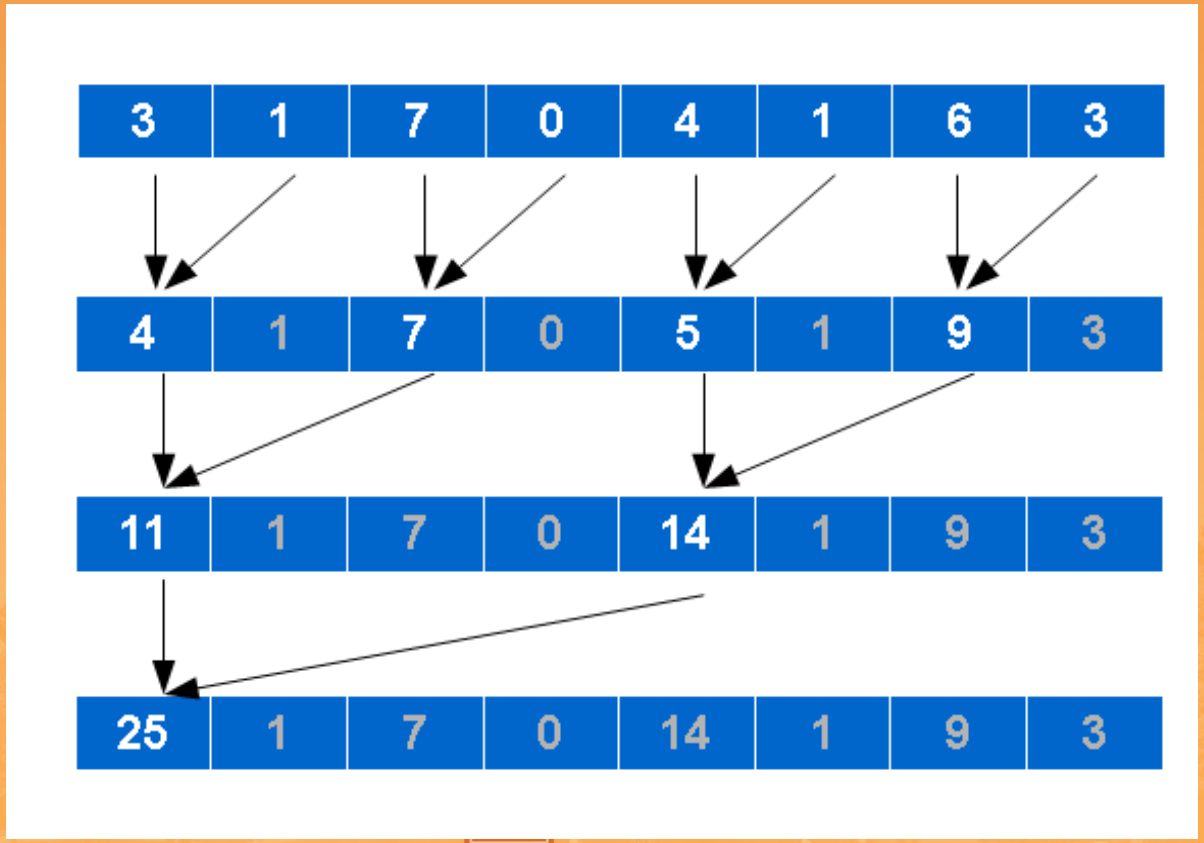
- Environmental variables that control the text version of the CUDA profiler are:
  - CUDA_PROFILE
  - CUDA_PROFILE_LOG
  - CUDA_PROFILE_CSV
  - CUDA_PROFILE_CONFIG

Controlling the CUDA profiler

- g{ld,st}_incoherent: Number of non-coalesced global memory loads/stores
- g{ld,st}_coherent: Number of coalesced global memory loads/stores
- local_{load,store}: Number of local memory loads/stores
- branch: Number of branch events taken by threads
- divergent_branch: Number of divergent branches within a warp
- warp_serialize: Number of threads in a warp that serialize based on address conflicts to shared or constant memory

Example: a parallel reduction with bank conflicts

- Consider an implementation of the following reduction scheme



Example: a parallel reduction with bank conflicts

- With the following profiler configuration in ./profile_config:

```
divergent_branch
warp_serialize
```

- And following command line (using bash):

```
CUDA_PROFILE=1 CUDA_PROFILE_CONFIG=./profile_config \
../../bin/linux/release/vector_reduction
```

- The generated cuda_profile.log may look like:

```
method,gputime,cputime,occupancy,divergent_branch,warp_serialize
method=[ _Z9reductionPfi ] gputime=[ 7.168 ] cputime=[ 64.000 ]
occupancy=[ 1.000 ] divergent_branch=[ 1 ] warp_serialize=[ 755 ]
```

Interpreting profiler counters

- A counter represents events within a warp; it does not correspond to individual thread activity in an SM
- To get consistent numbers, use 100+ blocks
- Best used to identify relative performance difference between optimized and unoptimized code

Performance Tuning

- Nonlinear optimizations
- Application specific
- Keep in mind
  - The ratio of memory access to data computation
  - Register usage
    - 8192 32-bit registers for G80, 16384 for GTX 280
  - Shared memory sizes on an SM
    - 16 KB for G80 and GTX 280
  - Cache working set for constant memory on an SM
    - 8 KB for G80 and GTX 280
  - Thread occupancy on an SM (# of active warps)
    - 768 active threads for G80, 1024 active threads for GTX 280
  - Block occupancy on an SM (# of active blocks)
    - 8 for G80 and GTX 280

Tiling

- A tile refers to the working set of data operated on by a block, etc.
- A tile can be a chunk of input data and/or output data
- Increasing tile size
  - Improves the input data reuse and reduce memory access
  - May lower # of active warps
- Decreasing tile size
  - Can use shared memory to reuse input data

Loop Unrolling

- The act of executing multiple copies of the loop body
- Pros
  - Reduce the number of branches
  - Reduce the number of evaluating loop termination conditions
- Cons
  - Require extra registers to store data for multiple loop bodies
  - May lower # of active warps
- May need “fixup” code

```
for (i = 0; i < N; i++) {
    sum += array[i];
}
```

```
for (i = 0; i < N; i=i+2) {
    sum += array[i];
    sum += array[i+1];
}
if (N mod 2 != 0)
    sum += array[N-1];
}
```

Working Size

- Working size is the amount of work done by a thread
- Increasing working size
  - Lowers the number of threads and blocks
  - Increases register usage
  - May increase the turn-around time for each thread
  - May increase data reuse in each thread
  - May lower # of active warps
- Decreasing working size
  - Has the opposite effects

Register Spilling

- Use shared memory as extra register space to relieve register usage
- Pros
  - Save additional memory accesses to global memory
- Cons
  - Manually coded by programmer
  - Takes extra operations to store/load to/from shared memory
  - Very application specific

Data Prefetching

- Fetch data for the next loop iteration(s)
- Leverage the asynchronous aspect of memory accesses in CUDA
  - Memory access operation is not blocked
- Pros
  - Memory access latency can be hidden
- Cons
  - Increase register usage
  - Lower # of active warps

```
for (i = 0; i < N; i++) {
    sum += array[i];
}
```

```
temp = array[0];
for (i = 0; i < N-1; i++) {
    temp2 = array[i+1];
    sum += temp;
    temp = temp2;
}
sum += temp;
```

Using Registers

- Store frequently reused data
- Pros
  - Save memory accesses to global memory
- Cons
  - Increase register usage
  - May lower # of active warps

```
for (k = 0; k < N; i++){
    for (i = 0; i < M; i++) {
        sum += v1[k]*v2[i];
    }
}
```

```
for (k = 0; k < N; i++){
    temp = v1[k];
    for (i = 0; i < M; i++) {
        sum += temp*v2[i];
    }
}
```

Questions?