Extending GRAMPS Shaders

Jeremy Sugerman
June 2, 2009
FLASHG
Agenda

- GRAMPS Reminder (quick!)
- Reductions
- Reductions and more with GRAMPS Shaders
GRAMPS Reminder

Map-Reduce

Produce

Initial Tuples

Map

Intermediate Tuples

Combine (Optional)

Intermediate Tuples

Reduce

Intermediate Tuples

Sort

Final Tuples

Output

Ray Queue

Camera

Ray Tracer

Intersect

Ray Hit Queue

Fragment Queue

Shade

Intermediate Tuples

FB Blend

Frame Buffer

Intermediate Tuples

Stage Output

Thread Stage

Queue

Shader Stage

Push Output
GRAMPS Shaders

- Facilitate data parallelism
- Benefits:
  - auto-instancing, queue management, implicit parallelism, mapping to ‘shader cores’
- Constraints:
  - 1 input queue, 1 input element and 1 output element per queue (plus push).

- Effectively limits kernels to “map”-like usage.
Reductions

- Central to Map-Reduce (duh), many parallel apps
- Strict form: sequential, requires arbitrary buffering
  - E.g., compute median, depth order transparency

- Associativity, commutativity enable parallel incremental reductions
  - In practice, many of the reductions actually used (all Brook / GPGPU, most Map-Reduce)
Logarithmic Parallel Reduction
Simple GRAMPS Reduction

- Strict reduction
- All stages are threads, no shaders
Strict Reduction Program

sumThreadMain(GrEnv *env) {
    sum = 0;
    /* Block for entire input */
    GrReserve(inputQ, -1);
    for (i = 0 to numPackets) {
        sum += input[i];
    }
    GrCommit(inputQ, numPackets);

    /* Write sum to buffer or outputQ */
}

Incremental/Partial Reduction

```c
sumThreadMain(GrEnv *env) {
    sum = 0;
    /* Consume one packet at a time */
    while (GrReserve(inputQ, 1) != NOMORE) {
        sum += input[i];
        GrCommit(inputQ, 1);
    }
    /* Write sum to buffer or outputQ */
}
```

Note: Still single threaded!
Shaders for Partial Reduction?

- **Appeal:**
  - Stream, GPU languages offer support
  - Take advantage of shader cores
  - Remove programmer boiler plate
  - Automatic parallelism and instancing

- **Obstacles:**
  - Location for partial / incremental result
  - Multiple input elements (spanning packets)
  - Detecting termination
  - Proliferation of stage / program types.
Shader Enhancements

- Stage / kernel takes N inputs per invocation
  - Must handle < N being available (for N > 1)
- Invocation reduces all input to a single output
  - Stored as an output key?
- GRAMPS can (will) merge input across packets
  - No guarantees on shared packet headers!

- Not a completely new type of shader
- General filtering, not just GPGPU reduce
GRAMPS Shader Reduction

- Combination of N:1 shader and graph cycle (in-place).
- Input “Queue” to validate only gets NOMORE
Scheduling Reduction Shaders

- Highly correlated with graph cycles.
  - Given reduction, preempt upstream under footprint.
- Free space in input gates possible parallelism
  - $1/N$th free is the most that can be used.
  - One free entry is the minimum required for forward progress.
- Logarithmic versus linear reduction is entirely a scheduler / GRAMPS decision.
Other Thoughts

- (As mentioned) Enables filtering. What else?
- How interesting are graphs without loops?

- Are there other alternatives? Would a separate “reduce” / “combine” stage be better?

- Questions?