Mio:
Fast Multipass Partitioning via Priority-Based Instruction Scheduling

Andrew T. Riffel, Aaron E. Lefohn, Kiril Vidimce, Mark Leone, John D. Owens

University of California, Davis                Pixar Animation Studios

http://graphics.cs.ucdavis.edu/~lefohn/work/shadingLang/mio/
Programming for CPU

- Programming for CPU is easy
  - Focus on algorithm
  - Not on target hardware
- Compiler handles most complexities
  - Memory
  - Resource Allocation
Programming for GPU

- Programming for GPU is not easy
  - Focus on target hardware
  - Makes algorithm design hard
- Programmers must handle complexities
  - Instruction Counts
  - Register Usage
  - Multiplatform Programming
What happens when a shader is too big?

- Multipass rendering
  - Partition the shader into smaller shaders which do fit
  - Store intermediate results in texture memory, and then rerun the entire pipeline with the next partition

- Multipass rendering allows virtualization of programmable hardware resources
  - Virtualization allows programmers to abstract away the hardware resources
Multipass Partitioning Problem (MPP)

Definition:

Given a shader, generate partitions that will fit within the available hardware resource.
Who needs virtualization?

- General Purpose GPU (GPGPU) users
  - GPGPU algorithms use the hardware in unanticipated ways.
  - These algorithms stress the GPU differently than shaders.
- Film studios such as Pixar
  - Very large, complex shaders exceed GPU limits
- Multiplatform shader development
  - Backwards compatibility for previous hardware.
  - Development for future hardware.
- OpenGL Implementations
  - “[Implementations] virtualize resources that are not easy to count.”
    - OpenGL Shading Language Spec.
Goals

- New partitioning framework
  - Fits easily into existing compiler flows
- Fast algorithm
  - Targeting run-time compilers
  - $O(n \log n)$ time
- Robust
  - Shaders of arbitrary size
  - Support for different hardware
- Extensible
Mio

- Derived from the word meiosis
  - A process of cell division that produces child cells with half the number of chromosomes
  - Mio divides large programs into smaller partitions
Outline

- Recursive Dominator Split (RDS)
- List Scheduling
- Mio: Algorithm Design
- Results
- Conclusions and Contributions
RDS and the MPP

- Eric Chan et al. 2002

Recursive Dominator Split (RDS)

- \(O(n^3)\) and heuristic cousin RDS\(_h\) \(O(n^2)\)

- Solves MPP for hardware with differing constraints and performance characteristics
RDS limitations

- Runtime Complexity
  - \(O(n^3)\) and \(O(n^2)\) impractical at runtime for very large shaders

- No Support for Multiple Render Targets (MRT)
  - MRTs allow complex outputs
  - Deferred shading
  - Simplify the MPP problem

- Not very extensible
  - No control flow support
Minimization Criteria

- RDS
  - Number of passes
    - 16 instructions per pass
    - Pass overhead dominates performance
  - Mio
    - Number of operations
      - 1000 instructions per pass
      - Overhead of the operations dominates performance

Runtime of a 5,000 operation shader rendered in a 512x512 quad
Save vs. Recompute

- **RDS**
  - Save always results in a new pass
  - Recomputation = More operations
  - Minimize passes = Recompute often

- **Mio**
  - Save does not always result in a new pass
  - Recomputation = More operations
  - Minimize operations = Never recompute
Multiple Render Targets

- RDS assumes a single output per pass
  - Vector or Scalar
  - Merging Recursive Dominator Split (MRDS)
    - Tim Foley et al. 2004
    - Uses MRTs to gain significant increase in shader performance
- Mio uses all available MRTs
  - Packs scalars and vectors to fill all outputs
List Scheduling

- Input is a directed acyclic graph (DAG) of the dataflow within the program
- Nodes represent operations
- Edges represent ordering dependencies between operations
List Scheduling

- First-ready nodes are added to a ready list
- Highest priority node is selected and added to the schedule
List Scheduling

- Highest priority node is selected and added to the schedule
- Scheduled node is removed from ready list, and scheduling continues with next highest priority node
List Scheduling

- Highest priority node is selected and added to the schedule
- Scheduled node is removed from ready list, and scheduling continues with next highest priority node
List Scheduling

- Highest priority node is selected and added to the schedule
- Scheduled node is removed from ready list, and scheduling continues with next highest priority node
- Any new ready nodes are added to ready list
List Scheduling

- Any new ready nodes are added to ready list
- Scheduling of nodes continues until all nodes are scheduled
List Scheduling

- Any new ready nodes are added to ready list
- Scheduling of nodes continues until all nodes are scheduled
List Scheduling

- Any new ready nodes are added to ready list
- Scheduling of nodes continues until all nodes are scheduled
List Scheduling

- Any new ready nodes are added to ready list
- Scheduling of nodes continues until all nodes are scheduled
List Scheduling

- Any new ready nodes are added to ready list
- Scheduling of nodes continues until all nodes are scheduled
List Scheduling

- Any new ready nodes are added to ready list
- Scheduling of nodes continues until all nodes are scheduled
List Scheduling

- Any new ready nodes are added to ready list
- Scheduling of nodes continues until all nodes are scheduled
List Scheduling

- Any new ready nodes are added to ready list
- Scheduling of nodes continues until all nodes are scheduled
Scheduling = Partitioning

- Scheduling an operation
  - Adds that operation to the current partition

- Incremental resource estimation
  - Track resources used
  - Updated after every operation added
Mio Priorities

- Mio uses Sethi-Ullman Numbering
  - Produces optimal schedules for trees
    - Optimal = Minimum register pressure
    - Good heuristic for DAGs
  - Generates deep not wide
    - Wide traversals cause extra register pressure
    - Deep traversals minimize register pressure
Deep Not Wide

- Scheduling C cause 3 intermediate results
- Scheduling F results in only 1 intermediate result
- Intermediate Results = MRTs
Mio List Scheduling

Populate ready list with first ready instructions

Is Ready List Empty?

Yes

Is rollback stack empty?

Yes

Close current pass.

No

Remove nodes on rollback stack from the current pass.

Are there any nodes not scheduled in a pass?

Yes

Partitioning Complete

No

No

No

No

Yes

Does node fit within pass?

Yes

Schedule node in this pass and remove from ready list. Clear rollback stack.

Close Pass by clearing ready list.

No

Does node violate output constraints?

No

Schedule node in this pass and remove from ready list. Add node to rollback stack.

No

Attempt to schedule highest priority node.
Mio List Scheduling

1. Populate ready list with first ready instructions.
2. Is Ready List Empty?
   - Yes: Close current pass.
   - No: Attempt to schedule highest priority node.
3. Does node fit within pass?
   - Yes: Schedule node in this pass and remove from ready list. Clear rollback stack.
   - No: Does node violate output constraints?
4. Are there any nodes not scheduled in a pass?
   - Yes: Partitioning Complete.
   - No: Remove nodes on rollback stack from the current pass.
Mio Example

- Wood Shader
  - 57 Operations
  - Limited 16 operations per pass
  - 4 outputs
Experimental Setup

- Mio was integrated in ATI’s prototype Ashli compiler. Ashli implements $\text{RDS}_h$ which was used for comparisons.
- Measure performance with a variety of Renderman shader programs.
- The runtime tests were performed on a pre-release GeForce 6800 (NV40) graphics card.
  - Since most of the experimental shaders fit into a single pass on the NV40 we compiled the shaders with ATI 9800 limits.
Results

- Compiler Performance
- Overall Quality of the Partitions
- Shader Performance
Results

- **Compiler Performance**
  - Mio has superior theoretical compile-time performance.
  - Experimentation also shows that Mio has better compile-time performance scaling over a number of large shaders.

- **Overall Quality of the Partitions**

- **Shader Performance**
Results

- Compiler Performance
- Overall Quality of the Partitions
  - Fewer total operations
  - More texture operations
  - Equivalent number of passes
- Shader Performance
Results

- Compiler Performance
- Overall Quality of the Partitions
- Shader Performance
  - For small shaders with few partitions, we found equal performance between RDS and Mio.
  - However for larger shaders with more partitions, the memory footprint and texture cache thrashing caused a substantial hit to Mio performance.
    - The passes generated by Mio were not optimized to reduce intermediate buffers
    - Optimizations still needed
Future Work

- Development of open source Mio partitioner
  - Open source code will be available for academic and non-commercial use.

- Alternate priority schemes
  - Explore the tradeoffs between compile time and partition quality within Mio framework.

- Support for control flow
  - We are currently extending the Mio algorithm to handle shaders that include control flow.
Conclusion and Contributions

- Characterization of MPP in a list-scheduling framework
  - Easily integrated into code generation
  - Supports multiple render targets
  - Well suited for more complex shaders which include flow control
- Development of an efficient priority scheme
  - Fast compile time
  - Comparable partitions to RDS
Acknowledgements

- Arcot Preetham and Mark Segal (ATI)
- Craig Kolb (NVIDIA)
- Brian Smits, Alex Mohr, Fabio Pellacini (Pixar)

Project Supported by:
- ChevronTexaco
- NSF
- UC Davis

Equipment by:
- NVIDIA
- ATI
- ALIENWARE