Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

Efficient Partitioning of Fragment Shaders for Multiple-Output Hardware

Tim Foley, Mike Houston and Pat Hanrahan †

Stanford University

Abstract

Partitioning fragment shaders into multiple rendering passes is an effective technique for virtualizing shading resource limits in graphics hardware. The Recursive Dominator Split (RDS) algorithm is a polynomial-time algorithm for partitioning fragment shaders for real-time rendering that has been shown to generate efficient partitions. RDS does not, however, work for shaders with multiple outputs, and does not optimize for hardware with support for multiple render targets.

We present Merging Recursive Dominator Split (MRDS), an extension of the RDS algorithm to shaders with arbitrary numbers of outputs which can efficiently utilize hardware support for multiple render targets, as well as a new cost metric for evaluating the quality of multipass partitions on modern consumer graphics hardware. We demonstrate that partitions generated by our algorithm execute more efficiently than those generated by RDS alone, and that our cost model is effective in predicting the relative performance of multipass partitions.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors G.2.2 [Mathematics of Computing]: Graph AlgorithmsTrees

1. Introduction

Real-time shading languages for graphics hardware simplify the task of writing shader code that is portable across a range of hardware and graphics APIs. However, most current high-level shading language compilers do not virtualize platform-specific resource limits such as number of instructions, input textures, or render targets. We can virtualize these hardware constraints for fragment shaders with *multipass partitioning* by dividing a large shader into multiple rendering passes.

The goal of multipass partitioning is to generate a set of passes that are equivalent to the input shader such that the cost of executing all the passes on the target hardware is minimized. The size of the search space is exponential in the number of operations, so good heuristic algorithms are necessary to find efficient partitions in reasonable time. The Recursive Dominator Split (RDS) algorithm is a polynomial-time heuristic algorithm for multipass partitioning that generates partitions that execute efficiently for a wide variety of shaders [CNS*02]. However, RDS is limited to operating on shaders with a single output color.

Recently, it has been shown that graphics hardware can also be used to run a large number of non-shading algorithms including ray tracing [PBMH02], fluid dynamics [HBSL03], and stream processing based applications [BFH*04]. These applications frequently require multiple outputs per pass, and hardware support for multiple render targets (MRT) makes it possible to more efficiently execute such shaders.

This paper presents Merging Recursive Dominator Split (MRDS), an extension of the RDS algorithm to support multiple-output shaders and graphics hardware with multiple render targets. The primary contributions of this paper are:

- The extension of the RDS algorithm to support shaders with multiple outputs. Our algorithm transforms the input DAG presented to RDS, allowing it to partition multiple output shaders, even for hardware which supports only a single render target.
- A pass-merging algorithm which allows partitions generated by RDS to be optimized for hardware with support

† {tfoley, mhouston, hanrahan}@graphics.stanford.edu

for multiple render targets. We derive two new algorithms for multipass partitioning, MRDS and MRDS′, that combine merging with RDS.
- Performance analysis demonstrating that MRDS produces more efficient partitions of fragment shaders than RDS. We show that even for shaders with only a single output value, partitions generated by MRDS can execute more efficiently on graphics hardware with MRT support.

2. Related Work

2.1. High-Level Shading Languages

Cook[Coo84] and Perlin[Per85] laid the groundwork for current shading languages. The Renderman shading language [HL90] is commonly used today for high-quality offline shading in software rendering systems. The Pixelflow shading system introduced pfman, a shading language for real-time rendering [OL98].

The Stanford Real-Time Shading Language (RTSL) system compiles a Renderman-like language for early programmable graphics hardware [PMTH01]. RTSL virtualizes "frequencies" of computation, allowing the compiler to select whether individual operations will be executed per-vertex or per-fragment.

NVIDIA's Cg [MGAK03] and Microsoft's HLSL [Mic03] are high-level languages for current graphics hardware that allow programs to be written for either the vertex or fragment processor and then compiled for a variety of hardware targets. While these languages do not encode any fixed resource limitations, the individual compiler targets have strict limits and will reject shaders that exceed them. Thus, programmers must write shader implementations for multiple hardware targets, manually breaking shaders that exceed resource limits into multiple passes.

The Sh system is a meta-compiler implemented in C++ for generating shaders at runtime and applying transformations to them [MQP02, MMT04]. The system does not explicitly specify resource limitations, although large shaders can currently fail to execute on graphics hardware.

The OpenGL Shading Language, GLSL, is a high-level languages for writing vertex and fragment processor shaders that makes it easy for programmable shaders to integrate with state from the fixed-function pipeline [KBR03]. The GLSL specification requires that implementations virtualize limits on number of instructions and temporary registers, but forces programmers to query and respect hardware-specific limitations on other resources.

2.2. Multipass Partitioning

Peercy, et al. map a subset of the Renderman language to a fixed-function OpenGL platform by abstracting the graphics hardware as a SIMD processor [POAU00]. Each rendering pass executes a single SIMD instruction, and a tree-matching approach is used to map shader computations to the small set of fixed-function operations available. While this technique generates good multipass partitions for fixed-function hardware, Proudfoot et al. demonstrate that tree-matching techniques are not sufficient for multipass partitioning on programmable hardware [PMTH01].

RTSL utilizes the RDS algorithm to virtualize fragment shading resource limits through multipass partitioning. As shown in [CNS*02], this allows for efficient partitioning of large shaders into multiple passes.

The Ashli system reads shaders in a number of high-level languages as input, including HLSL, GLSL, and Renderman, and generates low-level code to execute them on graphics hardware [ATI03a]. Ashli can generate multipass partitions for large shaders using RDS and provides the user with API calls to progressively render their scene. Ashli has demonstrated the effectiveness of RDS with multiple input languages.

Both the RTSL and Ashli systems are able to load balance shading computations between the vertex and fragment processors. Our implementation does not consider splitting shaders across the two shading units, only concentrating on partitioning shaders to run on the fragment processor.

2.3. Overview of the RDS Algorithm

Given a fragment shader represented as a DAG, the problem of generating a multipass partition is as follows: label a subset of the n DAG nodes as *splits*, intermediate values to be saved to texture memory, and then generate a partition of those splits into p rendering passes. Each pass then consists of shader code to generate its constituent splits as outputs, using texture fetches to restore the values of previously computed splits. Such a partition is *valid* if each of the passes can run on the target hardware and they can be ordered to preserve dependencies between splits. Among the many possible multipass partitions of a given shader, we wish to find one that is maximally efficient. However, the space of possible partitions is exponential, so the goal of multipass partitioning is to efficiently find a partition that is close to optimal.

The RDS algorithm combines top-down heuristic splitting, bottom-up greedy merging, and a limited search algorithm to mark nodes in a shader DAG as splits. We use the RDS algorithm as the base for our new techniques. RDS assumes that the target hardware can only output a single value per fragment in each rendering pass, and thus that the shader DAG has a single root. Along with this graph, RDS operates on its associated *dominator tree*. A DAG node x is a *dominator* of node y, written x dom y, if all paths from the root of the graph to y pass through x. The *immediate dominator* of node y is the unique $x \neq y$ such that x dom y and for all nodes $z \neq y$, z dom $y \Rightarrow z$ dom x. The dominator tree of a

DAG shares the node set of the graph, and connects each node as the direct child of its immediate dominator.

Some subset of the nodes in the DAG are *multiply-referenced* (MR) nodes, having more than one direct parent. The RDS algorithm is primarily concerned with whether these MR nodes should be marked as split locations. If a MR node is marked as a split, we subsequently incur the additional bandwidth costs of writing the value to texture memory and restoring that value in later passes. If the node is not marked as a split, we may incur additional computation costs from recomputing its value in multiple passes. Both of these additional costs can be eliminated if we can compute a MR node in the same pass as its immediate dominator, as all references to the MR node are then isolated to a single pass. The RDS algorithm attempts to eliminate save/recompute costs when possible and otherwise tries to chose the less expensive of the two options.

RDS uses two graph traversals to partition the DAG, a top-down subdivision over the dominator tree, and a bottom-up merge over the DAG. The subdivision traversal makes heuristic decisions about whether to save or recompute MR nodes, while the merge traversal greedily combines nodes with as many of their children as possible. These two traversals operate over the n nodes of the DAG and at each node use a low-level compiler to check the validity of a fixed number of subregions of the graph. Assuming that low-level compilation is a linear-time operation, this leads to an overall running time of $O(n^2)$. The graph traversals are wrapped in a limited search over the MR nodes of the DAG. The search algorithm forces successive MR nodes to be saved or recomputed (split, or unsplit) and compares the cost of partitions generated by the graph traversals under these two constraints. Each node is then fixed in whatever state led to the better partition and this information overrides the heuristic decisions made in the subdivide step. Using this limited search can increase the efficiency of generated partitions, but increases running times by a factor of n, making RDS an $O(n^3)$ algorithm.

3. The Algorithm

3.1. Multiple-Output Shaders

The RDS algorithm cannot operate on shaders with multiple output values since the dominator tree which drives the subdivision step is undefined when the shader DAG has multiple roots. A MR node m that can be reached from two different DAG roots (outputs) may have no immediate dominator, and thus would not be considered by the subdivide traversal of RDS.

We propose a simple solution to this problem that still allows us to take advantage of the information the dominator tree provides. Before applying RDS to a shader DAG with multiple root nodes, we insert a new node at the root of the DAG with operation *join* and having the shaders outputs as children. We define the join operation so that it compiles successfully if and only if all of its children are marked as splits. Figure 1 illustrates how this procedure is applied to a DAG with multiple roots. The left graphs represent shading DAGs and the right graphs are their associated dominator graphs. The addition of the join operation generate a single-rooted DAG from which a dominator tree can be derived. In this way we can partition multiple-output shaders into multiple single-output rendering passes while remaining transparent to the existing RDS algorithm.

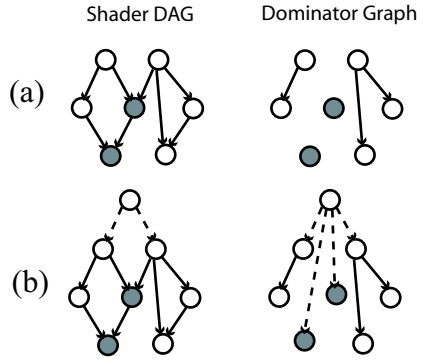

Figure 1: *(a) A multiple-output shader and its associated dominator graph. Note that the shaded intermediate nodes do not have immediate dominators, and the dominator graph is not a tree. (b) After adding a new root node to the DAG that joins the two outputs, the dominator graph is tree-structured.*

The result of the above algorithm will be a set of splits, and for hardware that can only write a single output per pass we can simply assign each split to its own rendering pass to achieve good results. However, on graphics hardware that supports multiple render targets, we may be able to produce more efficient partitions.

3.2. Multiple-Output Hardware

Typically, the outputs of multiple-output shaders will not be disjoint, and thus intermediate results may be shared between outputs. The immediate dominator of such intermediate value nodes is the root join operation, and thus the dominator tree cannot help us to eliminate the save/recompute costs for such nodes. However, if we can write all of the outputs that depend on such a node m in the same pass that we use to compute m, we can avoid these additional costs.

This situation is not unique to shaders with multiple outputs. As Figure 2 demonstrates, even in shaders with only a single output, there may be sets of intermediate values that can be computed together much more efficiently than apart. We can extend the concept of dominance in a graph to describe this situation:

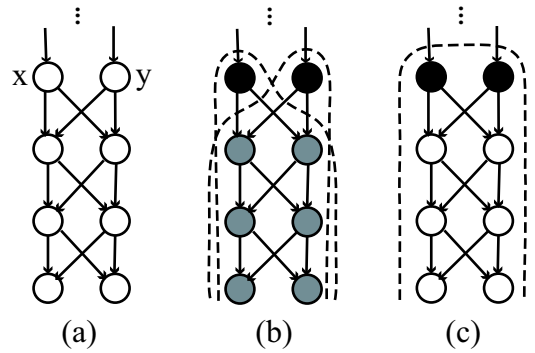

Figure 2: *A "ladder" configuration of intermediate value nodes. (a) The nodes x and y both depend on all previous intermediate values. (b) If x and y are marked as splits by a single-output partitioning algorithm, then the shaded nodes will have to be recomputed. (c) A MRT-aware algorithm can save both splits in a single pass and avoid the recomputation.*

Given a DAG node x and a set of nodes S, we say that x is a *dominating set* of x, written S dom x if and only if every path from a root of the DAG to node x passes through a node in S. The parents of a node x form a dominating set of x, and the set of shader outputs dominate every node in a shader DAG. Given this notation, we can generalize some of our previous statements about how dominance affects multipass partitioning: if a DAG node x is computed in a pass with output set S, where S dom x, then we can eliminate any save/recompute costs for x.

We can realize the benefits of this effect incrementally by merging the passes of an existing multipass partition. If an MR node n is being recomputed in many passes, and two of those passes are merged then some recomputation costs for m are eliminated. By repeatedly merging passes it may be possible to merge m with a dominating set and eliminate all of these costs. The following algorithm, MERGE-PASSES, uses greedy pass-merging to decrease recomputation costs in a multipass partition:

```
// input: T a set of splits as generated by RDS
MERGE-PASSES( T )
    // generate initial set of passes
    for each node n in T
        create a pass with output n and add it to the list of passes
    // create list of candidate merges
    for each pair (x,y) of nodes in T
        score ← TEST-MERGE( pass(x), pass(y) )
        if score ≥ 0 then
            add (score,x,y) to array A of potential merges
    sort A in decreasing order by score
    // attempt to execute merges from best to worst
    for each tuple (score,x,y) in A
        if pass(x) ≠ pass(y) then
            // score may have changed, so revalidate
            score ← TEST-MERGE( pass(x), pass(y) )
            if score ≥ 0 then
                EXECUTE-MERGE( pass(x), pass(y) )

// input: A, B passes (sets of nodes)
// output: a score measuring how "good" the merge is
TEST-MERGE( A, B )
    // if there is an ordering conflict, we cannot merge
    if ancestors(A) ∩ descendants(B) ≠ ∅
        ∨ descendants(A) ∩ ancestors(B) ≠ ∅
    then return −1
    M ← outputs(A) ∪ outputs(B)
    // use low-level compiler to check validity
    if pass-valid(M) then
        // score is how much the merge would improve partition cost
        cost_merged ← pass-cost(M)
        return cost(A) + cost(B) − cost_merged
    else return −1

// input: A, B passes (sets of nodes)
EXECUTE-MERGE( A, B )
    // create and initialize the merged pass
    create pass P
    outputs(P) ← outputs(A) ∪ outputs(B)
    cost(P) ← pass-cost(outputs(P))
    remove A, B from list of passes
    add P to list of passes
    // update membership information for nodes in this pass
    for each node n in outputs(P)
        pass(n) ← P
```

This algorithm operates on a list of splits and produces a list of passes. It begins by considering all possible pairwise merges between splits and determining whether they are valid. If there exists some split z such that z is an ancestor of one pass and a descendant of the other (i.e. z depends on the outputs of A and an output of B depends on z) we dismiss the merge as invalid. Otherwise we construct set M, the union of the output splits of passes A and B. If the elements of M cannot be generated in a single rendering pass on the target hardware, then the merge is invalid. For every valid merge we calculate a score measuring the improvement in cost of the merged pass over the two input passes, dismissing merges that yield negative scores. We iterate over the remaining potential merges in order of decreasing score and try to execute them. It is possible that earlier merges will have invalidated a potential merge or that it will no longer improve the overall score, so we re-check validity before executing any given merge.

The running time of this algorithm is dominated by the initial search for valid merges. For an n-node DAG and s splits, the search operates over s^2 pairs of splits and performs size-s set operations and an $O(n)$ compiler call. This yields an overall running time of $O(s^2(s+n))$ for MERGE-PASSES. In general, s is $O(n)$ in the number of DAG nodes, and thus the algorithm is $O(n^3)$, although in practice we expect s to be small relative to n.

If the MERGE-PASSES algorithm is run directly on the results of our multiple-output enabled RDS, then the result is an $O(n^3)$ algorithm for generating multipass partitions for MRT hardware. We call this algorithm MRDS, or Merging Recursive Dominator Split. We will demonstrate in the following section that MRDS can significantly improve the efficiency of multipass partitions generated by RDS.

However, it is possible that MRDS might produce poor partitions for certain shaders. Sets of splits that lead to expensive partitions when executed with one split per pass might yield highly optimized partitions after greedy merging. The search algorithm employed by RDS does not consider the potential for merging when evaluating the relative quality of partitions, and thus could make sub-optimal save/recompute decisions based on the incomplete information available to it. Thus, we introduce a second modified RDS algorithm, MRDS$'$.

The MRDS$'$ algorithm integrates greedy pass-merging directly into the search performed by RDS. When RDS evaluates the cost of a partition in order to make save/recompute decisions, we first apply pass-merging to collapse the candidate set of splits into a smaller number of passes. While this modification provides the RDS search with more accurate cost information, it comes with a penalty in asymptotic performance. The RDS algorithm wraps a linear search around an $O(n^2)$ graph traversal, and MRDS$'$ introduces the additional cost of $O(n^3)$ merging inside this search. Thus MRDS$'$ increases compile times by a factor of n over RDS, resulting in an $O(n^4)$ algorithm.

3.3. Implementation

We have implemented the RDS, MRDS and MRDS$'$ algorithms for multipass partitioning into the BrookGPU [BFH*04] system for stream computing on GPUs. Although the Brook language and the BrookGPU system are designed for stream computation rather than interactive shading, the implementation of our multipass partitioning algorithms are devoid of any concepts specific to streaming. The BrookGPU compiler transforms functions written in the Brook language into high-level fragment shaders, constructing a shader DAG from this representation. Our multipass partitioning algorithms operate on this DAG, using modularized compiler back-ends to encapsulate the validity testing and cost metrics required. In order to evaluate the effectiveness of MRDS and MRDS$'$ at generating multipass partitions for hardware with support for multiple render targets, we developed a back-end that generates DirectX 9 pixel shader code for the ATI Radeon 9800XT [Mic01, ATI03b]. This back-end uses the Microsoft HLSL compiler, fxc to perform validation and generate low-level code.

Whereas the RDS implementation used in RTSL operates over a DAG of low-level machine instructions, our system operates over a DAG of the high-level expression tree and relies on the external compiler to perform instruction selection. This makes implementing a back-end target much simpler as it need only pass high-level code to an external compiler. However, opportunities for optimization could be lost by operating on code before instruction selection. Our implementation will not split shader operations that are represented as primitive function calls, such as vector cross product, across passes even if they consist of multiple machine instructions. Furthermore, we generate DAG nodes for operations, such as swizzles and negation, which might be free on a particular hardware target.

3.4. Cost Metric

All of our multipass partitioning algorithms rely on a cost metric to provide predictions of shader performance on the target hardware. The original RDS implementation used a linear cost model that combined the total number of passes, arithmetic instructions and texture-fetch instructions in a given shader partition. In trying to model shader execution cost on our target platform, we have extended this model.

Chan et al. estimate the cost of each shader pass by measuring the time taken to render a single-pixel quad on the target hardware and convert this into units of GPU instruction cycles. The goal is to measure the cost of API overhead and setup to render a primitive, while making the cost of any per-fragment operations negligible [CNS*02]. However, because shader execution is typically asynchronous with API calls on the CPU, this overhead can be mitigated by running a shader on enough data so that GPU execution time dominates API overhead. We expect that in practice most applications of MRDS will apply to shaders being run on sufficiently large datasets to be GPU-limited rather than CPU-limited. Therefore we do not use the same procedure to derive per-pass overhead as Chan et al.

However, this is not to say that we do not account for per-pass overhead. Experimentally we have found that the dominant overhead of each rendering pass is in writing shader results to off-chip framebuffer memory. We measured this cost by comparing execution times $t0$ when running i instructions over n fragments and $t1$ when running $i/2$ instructions over $2n$ fragments, always operating on enough data to be GPU-limited. If we fit a cost model of the form $cost = c_i \cdot i \cdot n + c_p \cdot n$ then we expect to find $c_p = (t1 - t0)/n$. We have experimentally measured c_p, the per-fragment pass overhead, to be equivalent to approximately 10 instructions on our target platform.

We must also take into account the total number of outputs being written in a given pass, as each output incurs an additional bandwidth penalty. We have found that shader output writes appear to mask the instruction execution time of small shaders. We have determined that single-output shaders with only a single instruction take the same amount of time to execute as those with 6 instructions, and that this number of

instructions increases linearly as we increase the number of shader outputs.

We have measured the latency of a texture fetch on our target hardware to be equivalent to approximately 8 instructions for textures in float4 format. However, accounting for this delay with a linear term fails to take into account the potential for the hardware to schedule instructions to hide the latency of texture fetches. In the best case, the pipeline can completely hide texture latency and our execution time is $max(t_{textureFetch}, t_{instructionExecute})$, while in the worst case every texture fetch leads to a stall, and execution time is $t_{textureFetch} + t_{instructionExecute}$. Rather than try to inspect the texture-access behavior of a pass, we choose to model this term as a simple average of the best-case and worst-case forms.

The complete equation that we use to measure the cost c_{pass} of a shader pass consisting of i total instructions, t texture fetches and writing o output values is:

$$c_{pass} = c_p + max(c_o \cdot o, \frac{(i + c_t \cdot t) + max(i, c_t \cdot t)}{2})$$

where $c_p = 10$ is the per-fragment pass overhead, $c_o = 6$ is the number of instructions masked by each output write, and $c_t = 8$ is the latency of a texture fetch.

4. Results

In order to evaluate the quality of partitions generated by our new algorithms, we applied four different partitioning strategies to four different shaders written in the Brook language. The partitioning strategies were:

- Ideal - Represents a partitioning strategy that always puts all computation into a single rendering pass. The partitions generated by this strategy cannot be run on the target hardware, but are useful for comparison.
- RDS - The original RDS algorithm modified to accept multiple-output shaders.
- MRDS - The RDS strategy followed by our greedy pass-merging algorithm.
- MRDS′ - The RDS strategy with greedy pass-merging integrated into the RDS search step.

The applications were selected to provide a range of shader sizes, outputs, and uses. The individual applications are:

- Particle - A shader that advances a particle-system based cloth simulation. This shader uses two outputs to write the new position and velocity of each particle in the system. Each particle is constrained to up to 8 of its neighbors by spring forces. This application is representative of n-body dynamics simulations
- Fractal - A shader that computes a 40-iteration approximation to the Mandelbrot set. The shader is vectorized to process 4 adjacent points at once. This application was chosen to demonstrate that even shaders with a single output value may have strongly coupled intermediate values, and thus can benefit from hardware support for multiple outputs.
- Matrix - A shader that computes an 8 by 8 dense matrix multiply. Input and output matrices are packed with two float4 values per row. This shader shows how the merging algorithms, while not parameterized by the number of outputs supported in hardware, can partition shaders that use more than the number of supported outputs.
- Fire - A procedural volumetric fire shader [Ura02]. This shader uses 5 octaves of 4-dimensional Perlin noise, and demonstrates an extremely large shading computation with many levels of texture indirection.

Table 1 shows the results of partitioning and executing our applications under each of our partitioning strategies. These tables show the number of passes in each partition, along with its cost as given by our metric. The execution timing results represent the average per-fragment execution time, in nanoseconds, when running the partitioned shader on the graphics hardware. These values were generated by amortizing the time taken to shade a 1024 by 1024 pixel quad in the case of Particle and Fractal, and a 512 by 512 pixel quad in the case Matrix and Fire. All timing results were averaged over 1000 iterations. The compilation times represent the total time spent in the partitioning algorithm, including time spent waiting on the external compiler.

5. Discussion

Partitions generated by MRDS and MRDS′ outperform those generated by RDS for all of our shaders, including those that write only a single output. The relative increase in performance varies across the applications, and we will discuss these variations individually.

Although the Particle shader writes multiple output values, compiling for hardware with MRT support does not significantly increase its performance over a partition generated for single-output hardware. The majority of the instructions in the shader are used to calculate a sum of neighbor forces, and this running sum can be maintained with a single output per pass. The final integration of position and velocity is the only step that generates multiple values. Despite a 33% decrease in the number of passes, the merged partitions take only 5% less time to execute. This result indicates that per-pass overhead is not the dominant factor in shader performance.

The Fractal shader, despite writing only a single output value, shows a marked improvement in execution time when partitioned with MRDS or MRDS′ rather than RDS. The two merged partitions execute 47% more quickly than the RDS partition. We attribute this result to the fact that the Fractal shader maintains two live values throughout most of its body. These values are used in an iterative computation, with

Particle	Ideal	RDS	MRDS	MRDS′
Passes	1	6	4 (33%)	4 (33%)
Cost	276	449	389 (13%)	389 (13%)
ArithOps	170	210	208 (1%)	208 (1%)
TexOps	19	33	28 (15%)	28 (15%)
TimeExec	-	56	53 (5%)	53 (5%)
TimeComp	-	0.71	0.71	0.80

Fractal	Ideal	RDS	MRDS	MRDS′
Passes	1	10	4 (60%)	4 (60%)
Cost	225	666	324 (51%)	324 (51%)
ArithOps	205	355	211 (41%)	211 (41%)
TexOps	2	38	14 (63%)	14 (63%)
TimeExec	-	119	63 (47%)	63 (47%)
TimeComp	-	7.9	7.9	9.2

Matrix	Ideal	RDS	MRDS	MRDS′
Passes	1	32	10 (68%)	10 (68%)
Cost	401	2128	1028 (52%)	966 (55%)
ArithOps	208	656	326 (50%)	298 (55%)
TexOps	32	176	88 (50%)	82 (53%)
TimeExec	-	380	290 (24%)	279 (27%)
TimeComp	-	4.3	4.4	14

Fire	Ideal	RDS	MRDS	MRDS′
Passes	1	37	22 (36%)	22 (36%)
Cost	1630	3109	2799 (10%)	2722 (12%)
ArithOps	690	1334	1249 (6%)	1329 (0%)
TexOps	150	237	227 (4%)	213 (10%)
TimeExec	-	3172	2225 (30%)	1879 (41%)
TimeComp	-	78	78	244

Table 1: *Results of partitioning several shaders with RDS, MRDS and MRDS′. For each partition we list the number of passes, cost measure, number of arithmetic and texture operations,execution time (in nanoseconds per fragment), and compilation time (in minutes). Percentages measure relative improvement over the RDS partition.*

each successive value depending on both previous values, yielding the previously-described "ladder" configuration in the shader DAG. This ladder configuration forces the RDS partition to duplicate many calculations, leading to a 73% increase in the number of arithmetic operations over the Ideal partition. The merged partitions, on the other hand, introduce no additional arithmetic instructions beyond those needed to write intermediate outputs.

The Matrix shader shows modest improvements in execution time for both merged strategies. The MRDS and MRDS′ partitions are 24% and 27% more efficient, respectively, than the partition generated by RDS. As in the case of the Particle shader, a drastic decrease in the number of passes executed did not lead to a proportional increase in execution time. The Matrix shader also clearly demonstrates that our cost metric is not directly proportional to execution time. The cost of the RDS partition is approximately 2.2 times that of the MRDS′ partition, but takes only 36% more time to execute. However, it appears that relative differences in partition costs for a particular shader are predictive of relative performance.

The Fire shader also shows appreciable improvements of 30% and 40% over RDS when partitioned with MRDS and MRDS′. It is interesting to note that although the MRDS′ partition has a better score than the MRDS partition, it consists of the same number of passes, and actually executes more arithmetic instructions. Instead of minimizing instruction counts, the MRDS′ algorithm found a partition that reduced the total number of texture fetches at the expense of arithmetic operations. This is a sensible optimization for a bandwidth-limited shader, and our performance results show that these decisions yield a 16% increase in performance over MRDS.

Both MRDS and MRDS′ yield partitions of the Particle and Fractal shaders with the same overall cost and execution time, although the generated passes differ in the placement of a few instructions. For these relatively small shaders it appears that there exists a small set of nodes at which efficient splits can be made, and thus both MRDS and MRDS′ select similar points at which to split the DAG. For the larger Matrix and Fire shaders we find that the MRDS′ strategy generates partitions with slightly better performance. In the case of the Matrix shader, the MRDS′ partition has the same number of passes as the MRDS partition, but saves more intermediate values and thus avoids certain recomputation costs. Although the Fire shader saw modest gains in runtime performance from using MRDS′, we note that compilation with MRDS′ took over a factor of three times longer than with MRDS.

For all of our applications the RDS and MRDS compilations take a comparable amount of time. In the case of the Particle and Fractal shaders the MRDS′ algorithm increases these compile times by less than 17%. For these applications the number of splits is relatively small and the merge operations are quite efficient. In the case of the Matrix and Fire applications the number of splits is higher and the merges dominate the cost of the MRDS′ compiles. These results also demonstrate that the number of splits s is often much less than n, and the single merge step of MRDS is relatively inexpensive. We note that in every case between 94% and 97% of the compilation time was spent in the external compiler, fxc.

In general our results indicate that although the MRDS′ algorithm generates better partitions than MRDS for some shaders, the improvements in performance will often not be sufficient to justify the additional time complexity. However, the MRDS algorithm performs similarly to RDS, and improves performance on MRT hardware even for shaders that write only a single output value.

6. Conclusion and Future Work

We have described a set of modifications to the RDS algorithm that allow it to partition fragment shaders with multiple outputs, and to take advantage of hardware with support for multiple render targets. We have demonstrated that these modifications can allow partitions generated for hardware with multiple outputs to outperform partitions limited to single outputs, even for shaders producing only a single result. Although these modifications greatly expand the range of shaders to which multipass partitioning can be applied, a number of issues still remain.

While our implementation uses an external shader compiler to perform validation checks and measure resource usage, our results indicate that there is a clear benefit to integrating the MRDS algorithm for multipass partitioning into existing shader compilers. As the original RDS paper mentions, one way to improve the asymptotic bounds of the partitioning process is to utilize incremental compilation of shader passes. Using such a strategy we can effectively reduce the cost of compiler calls from $O(n)$ to $O(1)$.

It is useful to know how close the results of a heuristic algorithm, such as ours, are to optimality. The size of the search space involved, and the size of the shaders we use, make finding optimal partitions by brute force prohibitive. It may be possible to utilize directed search algorithms, such as A*, to find optimal partitions in far less time than a brute-force search. This would require the careful formulation of the state space to be searched, as well as the derivation of a good admissible heuristic to direct the search.

Our analysis of multipass partitioning assumes that a shader computation can be expressed as a single dataflow DAG, and thus does not handle loops, predication, or any data-dependent control flow. Future graphics hardware will support data-dependent branching in the fragment processor, and it is interesting to explore multipass partitioning algorithms that handle branching. It is possible that certain branching constructs will yield better performance when control flow is divided between the CPU and GPU, requiring partitioning algorithms to balance this load.

Although some hardware resource limits, such as instruction counts, have grown drastically with successive generations of programmable hardware, others, such as the number of available texture coordinate interpolants, have remained relatively low. Thus, shaders that use constrained resources will still require multipass partitioning. As instruction counts increase into the hundreds or thousands, a partitioning algorithm with running time polynomial in the number of shading instructions can be prohibitively expensive.

An interesting area for future research is to develop partitioning strategies that operate on dependency graphs of only the most constrained resources. This may be modeled as a constraint satisfaction problem, and limiting the search space in this way may drastically reduce compile times. Thinking of multipass partitioning in terms of resource allocation, this constraint search is closely related to register allocation and instruction scheduling for CPUs. Combining these two approaches, it may be possible to further improve the running time of multipass partitioning algorithms.

7. Acknowledgments

We would like to thank Eric Chan for the original RDS implementation from the Stanford Real-Time Shading Language system. Kekoa Proudfoot and Ren Ng provided invaluable feedback during the development of the MRDS algorithm. Kayvon Fatahalian provided the benchmarking results that drove our cost model.

This work was done on the Brook for GPUs system, which is supported by DARPA. Additional support has been provided by ATI, IBM, NVIDIA and SONY. The Brook programming language has been developed with support from Department of Energy (contract B527299-4), NNSA, under the ASCI Alliances program (contract LLL-B341491), the DARPA Smart Memories Project (contract MDA904-98-R-S855), and the DARPA Polymorphous Computing Architectures Project (contract F29601-00-2-0085).

References

[ATI03a] ATI: ASHLI - advanced shading language interface, 2003. http://www.ati.com/developer/ashli.html.

[ATI03b] ATI: Radeon 9800 technical specification, 2003. http://www.ati.com/products/radeon9800/radeon9800pro/specs.html.

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of ACM SIGGRAPH (to appear)* (2004).

[CNS*02] CHAN E., NG R., SEN P., PROUDFOOT K., HANRAHAN P.: Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Proceedings of the conference on Graphics hardware 2002* (2002), Eurographics Association, pp. 69–78.

[Coo84] COOK R. L.: Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), ACM Press, pp. 223–231.

[HBSL03] HARRIS M. J., BAXTER W. V., SCHEUERMANN T., LASTRA A.: Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 92–101.

[HL90] HANRAHAN P., LAWSON J.: A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (1990), ACM Press, pp. 289–298.

[KBR03] KESSENICH J., BALDWIN D., ROST R.: The OpenGL Shading Language, 2003. http://www.opengl.org/documentation/oglsl.html.

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics 22*, 3 (July 2003), 896–907.

[Mic01] MICROSOFT: DirectX product web site, 2001. http://www.microsoft.com/directx/.

[Mic03] MICROSOFT: High-level shader language, 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/Shaders/HighLevelShaderLanguage.asp.

[MMT04] MCCOOL M. D., MOULE K., TOIT S. D.: Sh: Embedded metapgramming language, 2004. http://libsh.sourceforge.net/.

[MQP02] MCCOOL M. D., QIN Z., POPA T. S.: Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), Eurographics Association, pp. 57–68. revised version.

[OL98] OLANO M., LASTRA A.: A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM Press, pp. 159–168.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[Per85] PERLIN K.: An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (1985), ACM Press, pp. 287–296.

[PMTH01] PROUDFOOT K., MARK W. R., TZVETKOV S., HANRAHAN P.: A real-time procedural shading system for programmable graphics hardware. *ACM Transactions on Graphics* (August 2001).

[POAU00] PEERCY M. S., OLANO M., AIREY J., UNGAR P. J.: Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 425–432.

[Ura02] URALSKY Y.: Volumetric fire cg shader, 2002. http://www.cgshaders.org/shaders/show.php?id=39.