Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

Optimal Automatic Multi-pass Shader Partitioning by Dynamic Programming

Alan Heirich

Sony Computer Entertainment America
Foster City, California

Abstract

Complex shaders must be partitioned into multiple passes to execute on GPUs with limited hardware resources. Automatic partitioning gives rise to an NP-hard scheduling problem that can be solved by any number of established techniques. One such technique, Dynamic Programming (DP), is commonly used for instruction scheduling and register allocation in the code generation phase of compilers. Since automatic partitioning occurs during the shader compilation process it is natural to ask whether DP is useful for shader partitioning as well as for code generation. This paper demonstrates that these problems are Markovian and can be solved by DP techniques. It presents a DP algorithm for shader partitioning that can be adapted for use with any GPU architecture. Unlike solutions produced by other techniques DP solutions are globally optimal. Experimental results on a set of test cases with a commercial prerelease compiler for a popular high level shading language showed a DP algorithm had an average runtime cost of $\mathcal{O}(n^{1.14966})$ which is less than $\mathcal{O}(n \log n)$ on the region of interest in n. This demonstrates that efficient and optimal automatic shader partitioning can be an emergent byproduct of a DP-based code generator for a very high performance GPU.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture I.3.3 [Computer Graphics]: Picture/Image Generation I.3.6 [Computer Graphics]: Methodology and Techniques

1. Introduction

As the complexity of programmable shaders increases their demand for GPU resources also increases. Complex shaders may consume more registers and other hardware resources than are physically available in the GPU leading to a situation where the shader cannot be compiled for the target architecture. This situation may be resolved by virtualizing the GPU resources in time by partitioning the shader execution into passes that use the same resources for different purposes at different times. This virtualization process is known as *shader partitioning* and for best results occurs during the code generation phase of a shading language compiler. Chan et al. [CNS*02] have identified the problem of automatically finding an optimal partitioning as the *multi-pass partitioning problem (MPP)*. This problem has been explored by them and by other authors [FHH04, RLV*04].

Riffel et al. [RLV*04] first observed that, like instruction scheduling, MPP is an instance of well studied job-shop scheduling problems that are known to be NP-hard or NP-complete [GJ75, MSSU02]. They demonstrate that MPP can be solved approximately during the code generation phase of a compiler by using an $\mathcal{O}(n \log n)$ average time List Scheduling algorithm. They show that this approach has advantages over other approaches in terms of compile time cost and generality of solution. Although the List Scheduling algorithm only processes basic blocks, and therefore does not handle flow control, the extension of this solution to include flow control and other important features appears relatively straightforward.

Riffel et al. have provided a scalable approximate solution to the MPP problem. Their solution is scalable because it has a compile-time cost of $\mathcal{O}(n \log n)$ average time (although potentially non-scalable $\Omega(n^2)$ worst time). It is approximate because it produces locally optimal answers that may be globally suboptimal.

This paper reports an exact algorithmic solution to MPP that provides globally optimal answers. The algorithm solves problems with multiple render targets and can be extended

to support flow control. The algorithm is based on Dynamic Programming (DP) which is an algorithmic paradigm for solving Markovian problems with nonlinear objective functions. The paper demonstrates that the MPP problem is Markovian, presents a DP solution to it, and analyzes the compile time complexity of the solution. This scaling analysis shows that while the compile-time cost of the algorithm is asymptotically inferior to that of the List Scheduling solution there is a large interval of interest on which the two are comparable. On the range of shaders in current use the runtime complexity of the DP algorithm was actually better than List Scheduling. This analysis is validated by experiment.

The DP solution is motivated by a study of a very high performance GPU that supports large and complex shaders. The size of these shaders implies multi-pass execution and motivates the search for a scalable partitioning algorithm. Increased shader complexity also provides greater potential for sub-optimality in the result, and this coupled with very high performance goals motivates the search for a globally optimal solution.

The contributions of this paper are an algorithm (see figure 1) that may be adapted for use with any GPU architecture, and a contextualization of the MPP problem with respect to job-shop scheduling.

1.1. Prior art

For a current survey of job shop scheduling see for example the paper by Möhring et al. [MSSU02]. The authors discuss solutions of the general Resource-Constrained Project Scheduling Problem by Integer Programming, Linear Programming, minimum graph cuts, List Scheduling, longest paths, and other techniques. All of these solution techniques are in principle applicable to MPP. Branch-and-Bound search is also commonly used for NP-complete scheduling problems and is applicable to this problem. For an example of Branch-and-Bound techniques applied to job scheduling see the paper by Hartmann and Drexl [HD98]. Genetic Search is another algorithmic paradigm that is sometimes applied to job scheduling problems, see for example the original paper by Davis [Dav85].

1.1.1. *MIO* (List scheduling)

Riffel et al. [RLV*04] first observed that MPP is an instance of job shop scheduling. They defined an objective to minimize the number of operations in a compiled shader which they argued is equivalent to minimizing the shader runtime. They developed an algorithm *MIO* based on List Scheduling that minimizes register usage and operation count and applied it to large and small shaders targeted to two commercially available GPUs. List scheduling is a well-known greedy algorithm with average time $\mathcal{O}(n \log n)$ and worst-case time $\Omega(n^2)$. (For a review of List Scheduling see for example Allen & Kennedy [AK02].) When compared to previous methods, notably *RDS*, the List Scheduling approach was seen to be faster and more general although the quality of results was sometimes inferior.

The approach presented here is similar to that of Riffel et al. [RLV*04] in understanding MPP as a formal optimization problem and in co-solving the problem in conjunction with code generation. The proposed algorithm searches for a minimum-cost schedule by incrementally scheduling individual operations. For the target architecture per-pass overhead is on par with other operations and is usually negligible.

The current work differs in employing a Dynamic Programming technique rather than List Scheduling. DP algorithms differ from greedy algorithms by producing optimal solutions to problems with nonlinear objective functions. The DP algorithm described here minimizes a nonlinear cost function that accounts for register operations, memory loads and stores, texture accesses, instruction fetches and costs associated with pass boundaries. Such a function can model texture latencies, register load times, and other real-world factors.

The complexity of Dynamic Programming is algorithm-specific and depends on the efficiency of pruning the search space. The algorithm described here had an average complexity of $\mathcal{O}(n^{1.14966})$ on a set of example shaders. Within the region of interest in n this approximates $\mathcal{O}(n \log n)$ scalability.

1.1.2. *RDS*, RDS_h, *MRDS*, $MRDS_h$ (Minimal cut sets)

Chan, Foley and their collaborators have developed an approach to MPP based on finding a minimal set of cuts in a DAG that represents the compiled shader program [CNS*02, FHH04]. Each cut in the DAG corresponds to a pass in the partitioned shader and this approach assumes that minimizing the number of shader passes is equivalent to minimizing the shader runtime. Some of the resulting algorithms have been implemented in the *Ashli* shader development toolkit [BP03].

The resulting algorithms are greedy bottom-up merges of the DAG into passes. Operations are scheduled into passes in order to respect resource limitations of the GPU. At pass boundaries intermediate results are spilled into a form of intermediate storage from which they can be retrieved in a subsequent pass.

These algorithms also explore the option of recomputing multiply referenced nodes of the DAG rather than saving them to intermediate storage. This consideration is motivated by the high cost on desktop GPUs of intermediate storage in the form of textures. By recomputing rather than saving intermediate results the shader may execute in fewer passes at the cost of executing more instructions. On current desktop GPUs this usually leads to a faster runtime.

Experiments with a diversity of shaders targeted to several current desktop GPUs showed these algorithms produce relatively high quality partitions but have non-scalable compile

times with complexities of $\mathcal{O}(n^3)$ for *RDS* (which handles single render targets) and $\mathcal{O}(n^4)$ for *MRDS*(which handles multiple render targets). The simplest algorithm RDS_h had non-scalable complexity $\mathcal{O}(n^2)$ and produced solutions that were judged to be of inadequate quality [CNS*02].

The present work differs from the minimal cut set approach in several ways. MPP is addressed within an optimization framework and principled techniques are applied to its solution. The technique of Dynamic Programming is chosen in part because it finds globally optimal solutions for nonlinear problems with multiple local minima. As a result the solutions are better than the solutions found by greedy algorithms because they are globally rather than locally optimal.

Rather than searching for cuts in a DAG the proposed algorithm works by incrementally generating a schedule in order to minimize a cost function. The objective minimizes a nonlinear cost function that predicts the expected execution time for a compiled shader. The costs associated with pass boundaries are negligible and are dominated by the time to make a memory reference for each word of intermediate storage.

Unlike others [CNS*02,BP03,FHH04] this approach does not equate minimal runtime of the compiled shader with a minimum number of passes. Instead the cost of adding a pass is included as one component in the overall cost function, like the cost of texture fetches, register copies, etc. This approach subsumes the minimal cut set approach by finding minimum-pass solutions when these are the most efficient but allowing more passes in a solution if this leads to a lower runtime cost.

1.1.3. Precursors

Multi-pass shaders have long been used with OpenGL hardware to achieve effects like shadows and reflections. Some of the earliest work is in Diefenbach's dissertation [Die96]. The first effort to automatically partition multi-pass shaders appeared in an influential paper of Peercy et al. [POAU00]. It described a Dynamic Programming-based partitioner for shading languages built on top of an OpenGL-based scene graph. The partitioner groups shader operations into SIMD passes that are executed on the entire frame buffer contents. Pass boundaries are selected to minimize the execution time while preserving the shader semantics.

This work is similar to Peercy's in using DP for automatic partitioning. It differs in targeting a GPU rather than a scene graph. Rather than breaking a shader into passes of OpenGL operations that affect the entire frame buffer the partitioner breaks a shader for a single vertex or fragment into passes of GPU instructions that observe physical resource limits of the GPU while minimizing the predicted execution time.

2. Problem statement

This study is concerned with partitioning shaders for a very high performance GPU that has a very efficient intermediate storage mechanism. GPU resources that must be scheduled include live operands (register allocation), outstanding texture requests, instruction storage and rasterized interpolants. Every shader pass must observe the physical resource limits of the GPU with any excess storage requirements satisfied from intermediate storage between passes. Each of these resource types has architecture-specific considerations that influence the cost function and the location of pass boundaries.

2.1. MPP Problem

The MPP problem for the present purposes can be stated: *given a valid shader program in the form of a DAG and associated information, generate a schedule of DAG operations and partition the schedule into passes such that: the total runtime as predicted by a cost function is minimal (where runtime includes operation cost and pass overheads); each pass observes the physical resource limitations of the GPU; and the schedule observes the precedence relations of the DAG.*

For illustration consider a simplest example of compiling the expression $(A+B)*(C+D)$ for a hypothetical GPU processor with two registers. Regardless of which sum is scheduled first its result cannot be kept in a register and must be stored for use in the final multiplication. In a multi-pass solution a second pass computes the other sum, retrieves the first sum, and multiplies the two sums together.

This simplest example illustrates the general strategy taken in the DP algorithm for identifying pass boundaries during the search. The algorithm generates code and allocates resources until a physical resource like available registers is exhausted. When this happens the algorithm allocates a portion of state to intermediate storage. It then continues code generation until certain conditions are met (such as filling intermediate storage or exhausting interpolants) at which time it inserts a pass boundary into the schedule. This process continues until the entire DAG has been scheduled.

2.2. MPP is Markovian

Only problems that are Markovian are amenable to solution by Dynamic Programming. For a multi-stage problem to be Markovian the choices at any stage must depend only on the current state and not on any prior states, that is, not on the history of how the current state was reached. This is the *Principle of Optimality* [Bel57]:

> An optimal policy has the property that whatever the initial state and the initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

The simplest example illustrates this principle. In compiling $(A+B)*(C+D)$ it is first necessary to compile one of the two sums. Without loss of generality consider the case of compiling $(A+B)$. At least two different instruction sequences could be generated for this sum: LOAD A, LOAD B, ADD; or LOAD B, LOAD A, ADD. Regardless of which operand was loaded first both of these sequences lead to a state in which the sum is in one register and the other register contains a stale operand. Subsequent compilation decisions depend on that state but not on which operand was loaded first, that is, not on the history of how the state was reached. Any instruction selected to follow one of these instruction sequences would also be a valid successor of the other sequence.

This example illustrates that register allocation is Markovian. For another example consider the subproblem of generating a schedule of rasterized interpolants in fragment shaders. As the GPU rasterizes data into fragments it interpolates quantities that were generated during setup and vertex processing. When the number of interpolants consumed by a fragment shader exceeds the number supported by the hardware these interpolants must be scheduled across multiple shader passes. The interpolants to be scheduled in any pass are drawn from the set of interpolants that have not already been scheduled in any prior pass. Any number of different schedules could be generated for these prior passes. If these different schedules result in the same set of unscheduled interpolants then they are equivalent for the purpose of scheduling in the present pass.

Similar arguments may be used to demonstrate that texture access, instruction storage and other aspects of shader scheduling are Markovian. As a result the scheduling problem in general, and MPP in particular, can be solved by Dynamic Programming techniques. The practical importance of the Markov property is that it allows a multi-stage decision problem to be solved in reverse order. This is illustrated in the next section.

3. Solution by Dynamic Programming

Dynamic Programming is an algorithmic technique for solving multi-stage optimization problems, including job-shop scheduling problems, that have nonlinear objective functions and thus cannot be solved reliably by greedy methods. Any DP algorithm exploits the property of *optimal substructure* (see 3.2.1) in which the globally optimal solution to a problem can be obtained by finding optimal solutions to a series of subproblems.

MPP is solved with a DP algorithm by employing backward recursion which solves the problem in reverse beginning with the final stage. The algorithm exhaustively searches a subspace of the total search space of possible schedules. This subspace contains only schedules with optimal substructure that end in the desired final state. At each stage the search explores some number of possible prior stages. An entire tree of solutions is searched in this way and the cost of every path from root to leaf is evaluated in parallel as the search proceeds. At the end of this process the global solution is taken as the lowest cost path from the root to any leaf.

```
Cost DPMPP(stage, T) {
  P := {}
  (∀t ∈ T) do
    S_t := { valid prior transitions of t
           { with optimal substructure }
    min := ∞
    (∀s ∈ S_t) do
        s.cost := t.cost +
            COST(s.operation, s.postcondition)
        if (s.cost < min)
            min := s.cost
    enddo s
    (∀s ∈ S_t such that s.cost = min) do
        P := (P ⋃ s) uniquely by precondition
    enddo s
  enddo t
  if (stage > 1)
    return DPMPP(stage − 1, P)
  else
    return min_{p∈P}(p.cost)   }
```

Figure 1: *Algorithm DPMPP is a Dynamic Programming solution to the problem of automatically partitioning multi-pass shaders. It can be adapted for use with any GPU architecture by replacing the COST() function.*

3.1. DP algorithm for MPP

Algorithm *DPMPP* (figure 1) takes as arguments an integer *stage* and a set T of *transitions*. Each *transition* consists of a *precondition*, *postcondition*, *operation*, and *cost*. The *precondition* and *postcondition* are represented by snapshots of machine state, *operation* is one of the operations from the input DAG that are eligible for scheduling after *precondition*, or is a "new pass" operation, and *cost* is a scalar value. For any transition t the machine state in $t.postcondition$ represents the machine state in $t.precondition$ following application of $t.operator$.

DPMPP is first called with $stage = n$, the number of operations in the input DAG, and with T the set of valid final transitions to the desired end state. There is a single desired end state and this is the postcondition of every transition in the initial T. The desired end state has all output values in registers and T is the set of transitions that can occur in the final step of computing those output values. Each element of T has a unique precondition.

Upon entry algorithm *DPMPP* initializes an empty set of transitions P. P will represent a set of prior transitions that will be passed recursively to the preceding stage of the search. It corresponds to T of the current stage.

The algorithm next considers the set T which contains all of the transitions from the set of machine states under consideration in the current stage. For each element $t \in T$ the principle of optimal substructure is used to generate a new set S_t of transitions that may precede the current stage. For every element s in S_t, $s.postcondition = t.precondition$.

A "new pass" operation is considered only when it is made necessary by depletion of a GPU resource. For example, this can occur when all rasterized interpolants for the current pass have been allocated but another interpolant is referenced by the DAG. This can also occur if all general purpose registers are full but further progress requires discarding a live operand, or if any other physical resource (such as a maximum number of simultaneous pending texture requests) is exhausted.

For each s in S_t the algorithm updates $s.cost$ to reflect the best cost path from $s.precondition$ to the end state. This is the sum of $COST(s.operation, s.postcondition)$, the cost of the current operation preceding the current postcondition, and $t.cost$, the best cost among any path to the endstate that is reachable from $t.precondition$. Recall that $s.postcondition = t.precondition$ always. Note also that $COST()$ depends on the machine state in $s.postcondition$ as well as on $s.operation$. This cost update is applied to all elements of S_t.

$COST()$ is the function that reflects the cost model of the GPU and predicts the costs of individual factors that determine runtime of the compiled shader. For this study $COST()$ was defined in units of execution time and accounted for register loads, interpolant access, texture access times, DAG operation costs, new pass overhead and other factors. The schedule generated by the DP algorithm will be optimal with respect to the costs modeled by $COST()$. The quality of these schedules depends on the accuracy with which $COST()$ reflects the actual costs of the GPU.

After these cost updates the s with lowest cost are added to the set P. There may be more than one such s for any S_t. However there may be only one element of P with a given precondition. If there is already an element $p \in P$ such that $s.precondition = p.precondition$ then the costs of the two elements are compared and only the lowest cost element is retained.

These steps are then repeated for all of the remaining $t \in T$. In this way the double loop in s and t examines all possible prior transitions and retains the lowest cost set of paths from the current stage to the end state. The number of paths being simultaneously considered may change at each stage. This algorithm continues recursively until it reaches the initial stage at which time it returns the lowest cost among elements of P. The optimal schedule is the path from this leaf element p to the root of the search tree.

3.2. Algorithm correctness

Algorithm *DPMPP* is correct if it produces a globally optimal schedule that respects GPU resource limitations and DAG precedence relations. GPU resources are evaluated in the $COST()$ function which returns a value of infinity if a resource is exhausted. A prior transition s is valid if it does not violate resource limitations and if $s.operation$ is a valid precedent of $t.operation$. These conditions are trivial to check.

The resulting solution is globally optimal if it observes the Principle of Optimality [Bel57] at each stage. In the final stage the initial set T satisfies this principle trivially. Every element t of the initial T is an optimal path of length 1 leading from $t.precondition$ to the desired end state.

For each preceding stage the algorithm generates a new set P of transitions. At the end of the double loop in s and t, every transition $p \in P$ represents an optimal path from $p.precondition$ to the end state. To be eligible for inclusion in P every candidate transition s must be a lowest cost element of its set S_t. In addition the uniqueness condition requires that $\forall p \in P, p.precondition$ occurs uniquely and is the lowest over all S_t. As a result of these conditions P satisfies the Principle of Optimality at every stage.

Progress and termination of the algorithm follow directly from the use of backward recursion. When the first stage has been computed the set P satisfies the Principle of Optimality for paths of length n. The best $p \in P$ represents the globally optimal solution.

3.2.1. Optimal substructure

S_t is generated for each $t \in T$ using the principle of optimal substructure. Candidate transitions are considered if they satisfy an optimality condition that depends only on the current stage. Every globally optimal solution is also compatible with this optimal substructure condition. To derive such a condition consider the problem of generating code that runs in minimal time. This is equivalent to generating code that delivers the maximum computational throughput, which is equivalent to having the lowest amount of idle time of operands in registers.

At runtime the DAG is executed bottom-up. It executes most efficiently when operands are in registers so that no idle time is spent waiting for loads to complete. At compile time in the proposed algorithm the DAG is processed top-down because the operations are scheduled in reverse order. In order to minimize register loads the algorithm selects from among candidate operations those with the smallest number of descendents in the DAG. By choosing the smallest subtree the algorithm is on average minimizing the number of

registers that are required to compute the result. This leads to the lowest register idle time and fastest runtime execution.

Every globally optimal solution is compatible with this optimal substructure condition. Global solutions have minimal predicted runtime cost, and optimal substructure ensures minimal predicted cost for individual subtrees. Global solutions have minimal cost over all subtrees and therefore are compatible with the optimal substructure condition.

Although not completely equivalent this condition is similar to the Sethi-Ullman numbering used by algorithm MIO to select subtrees for scheduling. Sethi-Ullman numbering assigns a number at each DAG node that tells how many registers are required to evaluate the subtree rooted at that node. Subtrees are then scheduled by MIO in order of decreasing number. This results in a partial ordering by size with the largest subtrees scheduled first and is known to result in minimal register usage [SU70]. The optimal substructure condition of algorithm DPMPP results in a total ordering of subtrees by size with the smallest subtrees scheduled last. As in Sethi-Ullman numbering the goal is to minimize register usage and execution time.

The optimal substructure condition also has a dramatic beneficial effect on compile-time complexity. Since there is typically a unique minimum subtree the size of set S_t is most often 1 which leads to small branching factors. This can be seen in figure 2 and is responsible for the tractability of the DP algorithm.

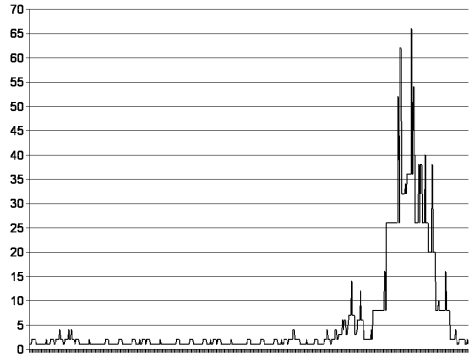

Figure 2: *Example search width w for a fragment shader with 490 operations. The branching factor b_i at each stage i is w_{i+1}/w_i. The average b over these 490 stages was 1.06091. Early in the search the final operations are being scheduled and the search width is small. For the first three quarters of the search b_i is most often 1 but also takes on the values $1/2$ and 2. Near the end of the search as the initial operations are being scheduled the search widens considerably. This pattern was observed in most of the test cases.*

3.3. Algorithm complexity

The complexity of the DP algorithm is equal to the number of stages in the schedule n raised to the power of the average branching factor of the search tree b. In order to be scalable b must be significantly less than 2 and ideally close to 1. The optimal substructure condition implies that b will be close to 1 although the exact value will vary for different DAGs and this was confirmed by experiment. The value of b was measured empirically on a set of test cases on which it averaged 1.14966. As a result the average compile time cost on these cases was $\mathcal{O}(n^{1.14966})$.

4. Experiment

A prerelease commercial compiler for a popular high-level shading language was augmented with the DP algorithm to perform operator scheduling and pass partitioning. A set of example shaders was compiled to verify the DP algorithm and measure the average branching factor b. A representative trace of the search width is shown in figure 2. The average branching factor on these test cases was always between 1.047 and 1.257 with an average of 1.14966.

The test set consisted of 10 vertex shaders and 3 fragment shaders. The vertex shaders were taken from public sources. They ranged between 36 and 179 operations in length with an average length of 89. Their branching factors ranged from 1.10088 to 1.25735 with an average of 1.17640.

The fragment shaders were 344, 490 and 563 operations in length with branching factors of 1.07336, 1.06091 and 1.04731 respectively. Two of these shaders were generated by a modeling package and perform Phong shading with bump maps and textures. The third was a self-shadowing shader obtained from public sources.

The average branching factor on this test set was 1.14966. The overall compile-time cost of algorithm *DPMPP* was therefore $\mathcal{O}(n^{1.14966})$. This is less than $\mathcal{O}(n \log n)$ up to $n = 12,053$ which is considerably larger than any of the shaders tested. This implies that algorithm *DPMPP* has compile-time cost comparable to algorithm *MIO* on current generation shaders. Shaders for real-time rendering are currently orders of magnitude smaller than this.

The fragment shaders showed a decreasing b with respect to n. This may be explained by the branching profile in figure 2. The branching factor remains close to 1 from the beginning of the search until near the end when it expands dramatically. The decrease in b with respect to n implies that this expansion occurs later (or with smaller magnitude) when n is larger. This suggests that early in the search the pruning was very efficient whereas near the end of the search there were a larger number of eligible DAG operations to consider. If this pattern persists for larger n then b will continue to decline from the values presented here implying an increase in scalability.

5. Summary and discussion

This paper has presented an algorithm for finding optimal solutions to the MPP problem. This algorithm can be adapted to any GPU architecture by modifying the $COST()$ function appropriately. On a variety of small and medium size test cases the algorithm had an average compile-time cost of $\mathcal{O}(n^{1.14966})$ making it comparable to the $\mathcal{O}(n \log n)$ scalability of algorithm *MIO* up to at least $n = 12,053$ which is as large as the largest shaders reported in the literature.

It was difficult to locate publicly available large shaders and this was the reason for generating the fragment shader test cases. In general purpose GPGPU applications shaders have been reported with over ten thousand operations. Unfortunately no such shaders in the language under study were available on the relevant public web sites.

This paper describes a study of a single algorithm targeted to a single very high performance GPU. As such it does not present a competitive analysis of different algorithms. Such an analysis would fill in the table in figure 3 with more of the candidate algorithms mentioned in section 1.1. These would be evaluated on multiple GPUs with larger shaders and compared for scalability and quality of results. Among these approaches Integer Programming and Branch-and-Bound solutions appear to have straightforward implementations.

Figure 3 contrasts the costs and benefits of the three algorithmic approaches discussed in this paper (minimal cut sets, List Scheduling and Dynamic Programming). Of the three the only one that is guaranteed to be scalable is List Scheduling but this produces sub-optimal solutions. The only approach that produces globally optimal solutions is Dynamic Programming but this is not asymptotically scalable. However on the range of interest algorithm *DPMPP* demonstrated compile-time costs comparable to algorithm *MIO*.

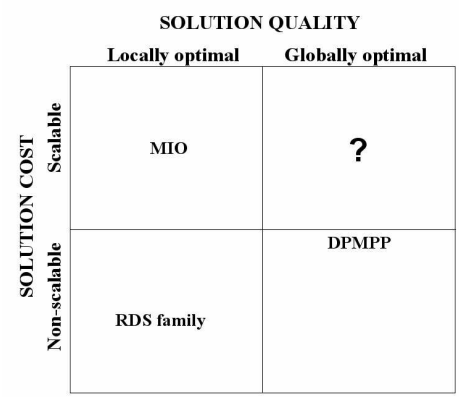

Figure 3: *Compile-time cost versus solution quality. Algorithm MIO uses List Scheduling, DPMPP uses Dynamic Programming, and the RDS family of algorithms search for minimal cut sets.*

	List Scheduling	DP
Direction:	*forward*	*reverse*
Paradigm:	*greedy*	*optimization*
Optimality:	*local*	*global*
Complexity:	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{1.14966})$
Numbering:	$Sethi-Ullman$	*subtree size*
Critical:	*scheduling priority*	*search pruning*
Policy:	*largest trees first*	*smallest trees last*
Affects:	*run time*	*compile&runtime*

Figure 4: *Comparison between List Scheduling (MIO) and Dynamic Programming (DPMPP) solutions to MPP.*

There are many similarities between algorithms *MIO* and *DPMPP*. Both algorithms generate a schedule incrementally in order to minimize a cost function. Algorithm *MIO* generates a schedule in order and schedules largest trees first in order to maximize the solution quality by minimizing register usage and therefore execution cost. Algorithm *DPMPP* generates a schedule in reverse order and schedules smallest trees last in order to minimize register loads and therefore execution cost. This has the additional benefit of drastically pruning the search space which is directly responsible for the algorithms scalability. Figure 4 summarizes these issues.

Both algorithms use DAG node numbering in order to maximize runtime efficiency of generated code. *MIO* uses Sethi-Ullman numbering while *DPMPP* uses subtree size which is computed in a similar way. It would be worth exploring Sethi-Ullman numbering for use in algorithm *DPMPP*.

Although the present algorithm does not address flow control this issue is largely orthogonal to it. Many aspects of flow control are relatively straightforward such as unrolling loops with predictable repetition counts and implementing medium-grained SIMD conditional operations. Other aspects of flow control are more problematic. For example, large loops with dynamically computed repetition counts could be too large to schedule within a single shader pass. Correct execution of such compiled code may require repeated execution of shader passes out of order. This is a difficult problem, and is a general problem for streaming computations apart from MPP.

6. Acknowledgements

Thanks to Gabor Nagy for providing two of the fragment shader examples. Axel Mamode gave a thoughtful reading to the original manuscript. John Owens provided insightful comments about the similarities between algorithms MIO and DPMPP.

References

[AK02] ALLEN R., KENNEDY K.: *Optimizing Compilers for Modern Architectures*. Morgan-Kaufmann, 2002.

[Bel57] BELLMAN R. E.: *Dynamic Programming*. Dover, 1957.

[BP03] BLEIWEISS A., PREETHAM A.: Ashli - advanced shading language interface. *ACM Siggraph Course Notes* (2003), http://www.ati.com/developer/SIGGRAPH03/AshliNotes.pdf.

[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 *Graphics Hardware 2002* (Sept. 2002), pp. 69–78.

[Dav85] DAVIS L.: Job shop scheduling with genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms* (Mahwah, NJ, USA, 1985), Lawrence Erlbaum Associates, Inc., pp. 136–140.

[Die96] DIEFENBACH P. J.: *Multi-pass pipeline rendering: interaction and realism through hardware provisions*. PhD thesis, Dept. Computer and Information Science, Univ. Pennsylvania, 1996. MS-CIS-96-26.

[FHH04] FOLEY T., HOUSTON M., HANRAHAN P.: Efficient partitioning of fragment shaders for multiple-output hardware. In *Graphics Hardware 2004* (Aug. 2004), pp. 45–54.

[GJ75] GAREY M. R., JOHNSON D. S.: Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Comput. 4* (1975), 397–411.

[HD98] HARTMANN S., DREXL A.: Project scheduling with multiple modes: A comparison of exact algorithms. *Networks 32* (1998), 283–297.

[MSSU02] MÖHRING R. H., SCHULZ A. S., STORK F., UETZ M.: Solving project scheduling problems by minimum cut computations. *Management Science 49*, 3 (2002), 330–350.

[POAU00] PEERCY M. S., OLANO M., AIREY J., UNGAR P. J.: Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000* (July 2000), Computer Graphics Proceedings, Annual Conference Series, pp. 425–432.

[RLV*04] RIFFEL A. T., LEFOHN A. E., VIDIMCE K., LEONE M., OWENS J. D.: Mio: Fast multipass partitioning via priority-based instruction scheduling. In *Graphics Hardware 2004* (Aug. 2004), pp. 35–44.

[SU70] SETHI R., ULLMAN J.: The generation of optimal code for arithmetic expresssions. *J. ACM 17*, 4 (1970), 715–728.