ISSN 1213-6972 (print)
ISSN 1213-6964 (online)
Journal of WSCG
http://www.wscg.eu
Vol.32, No-1-2, 2024

GPU Cache Flush Minimization In Render Graph Systems

Roman Sandu
Phystech School of Applied Mathematics and Computer Science
Moscow Institute of Physics and Technology
Institutskiy Pereulok, 9 Dolgoprudny, Moscow Oblast, 141701, Russia
sandu.ra@phystech.edu

Alexandr Shcherbakov
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, 119991, Russia
alex.shcherbakov@graphics.cs.msu.ru

ABSTRACT

Modern graphics APIs expose control over the infamously non-coherent GPU caches to application programmers through the mechanisms of pipeline barriers and render passes. A developer is then asked to group together their GPU computations based on memory access patterns such that cache flushes and invalidations are minimized, but render graph systems enable automation of this process. In this paper, we study the problem of finding an optimal execution order for a frame graph to minimize the amount of render pass breaks, which in turn minimizes cache control operations. We formulate and analyze a novel NP-complete problem MLGP and use it to propose an approach to render pass merging that results in 30% less render pass breaks when compared to previous works.

Keywords

frame graph, render graph, gpu, barrier, render pass, vulkan, dx12, tile based deferred renderer

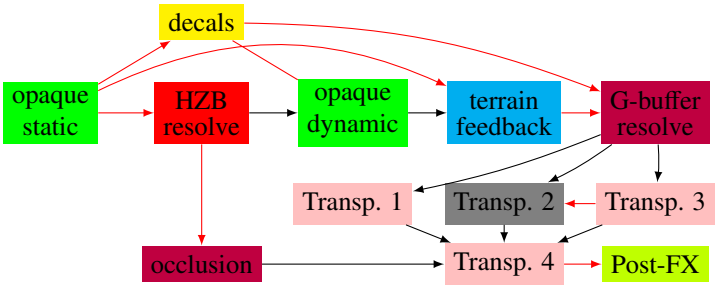

Figure 1: A contrived sample frame graph represented through a Λ graph. Nodes are computations, arrows represent causality or precedence, red edges represent barriers. Some precedence edges also require a barrier. Colors represent unique frame buffers requested by nodes. We are interested in finding a node execution order that enables optimal barrier batching and minimal render target changes.

1 INTRODUCTION

Frame Graphs and Cache Management

Throughout the history of computing, graphs have proven themselves to be an exceptionally effective tool for describing and working with computation. Most compilers use control flow graphs as one of the intermediate representations of a program; several approaches to concurrency use graphs explicitly to represent computations; various visual programming systems use graphs as their representation of choice, including graph-based shader authoring and gameplay programming systems as seen in industry-standard engines like the Unreal Engine [Epi]. Low-level graphics programming is no exception, most engines are either in the process of migrating from an immediate-style command dispatching architecture to a *frame graph* [ODo17] (also known as a *render graph* [Wih19]) based architecture, or have done so several years ago. A frame graph runtime library provides its user with an explicit node abstraction, which declaratively describes a computational atom. Different systems chose different granularity for the computations

abstracted inside nodes, from single dispatches of compute shaders or a fixed rendering algorithm, to an entire pass of rendering to a fixed set of render targets. The nodes communicate with each other through a resource system provided by the runtime, and the set of all user-created nodes constitute a computational graph that describes the process of computing a single application frame which is to be presented on the screen.

It is noteworthy that a principally different graph-based approach has recently appeared in the Godot engine [Ban24]. Godot uses an immediate-style user-facing API, but does not translate user-dispatched GPU commands directly into low-level graphics API command lists, instead constructing a data dependency graph from them, which is then processed analogously to traditional frame graphs. This approach is close in spirit to control flow graphs used by various compilers and allows to reap most benefits of a frame graph like system without breaking compatibility with existing application code.

The benefits of a graph-based approach are manyfold. User-facing frame graph systems provide a robust architectural framework for organizing graphics application code, and both user-facing and Godot-style graph systems enable automatic optimization of various aspects of an application: memory allocation, work scheduling, resource management, cache management, and several others. For this paper, we are interested in cache management and work scheduling in particular.

While the previous generation of graphics APIs have hidden all of cache management infrastructure away from an application programmer inside the device-level driver, such an approach has proven itself to be quite inefficient at times, especially on mobile tile-based deferred renderer architectures, and so the new generation of graphics APIs, including Vulkan and Direct3D 12, delegates cache management to the application developer. Focusing on the Vulkan API, this is manifested in two aspects: pipeline barriers and render passes. Both of these are not exclusively cache management tools, however. A pipeline barrier is a tool that controls 3 aspects of an application:

1) cache invalidation and flushing,
2) GPU pipeline synchronization,
3) texture layout transitions.

Render passes, on the other hand, are a tool for controlling the special rendering output merging hardware available in most GPUs. Several parts of the rasterization pipeline can be implemented in hardware more optimally than in software via general-purpose memory operations, including Z-testing, alpha-testing and simple rendering writes. In particular, tile-based deferred renderer GPU architectures store the textures which participate in the output merging stage, usually referred to as *attachments*, in a special tile cache, which is never flushed or invalidated throughout a single render pass.

Obviously, any performance-sensitive program should strive to maximize cache locality, or in other words, minimize cache flushes and invalidations. For programs that execute on the GPU, this problem is two-fold: cache access must be independently optimized inside shader code and inside the high-level command stream executed by the GPU. The former is similar to CPU cache access optimizations and is covered extensively in literature (e.g. [Bav14]), while the latter is a problem that is novel for application programmers and has been previously solved by proprietary means inside drivers for OpenGL, DirectX11 and similar.

Being interested in frame graphs, we focus on the latter. In terms of the Vulkan graphics API, this means minimizing the amount of render passes and barriers that are executed throughout a frame while preserving the functionality and correctness of a program. Various work items with compatible *frame buffers*, the sets of active attachments, should be grouped together into render passes. Barriers should be grouped together too, as testing shows that fine-grained cache control exposed by the Vulkan API is usually not leveraged by device drivers, and a flush of any memory region from a certain cache usually leads to a flush of the entire cache. Moreover, as barriers also synchronize various GPU pipeline stages, generally by preventing new work from being scheduled onto a processing unit before previous work was completed, grouping them also minimizes the amount of stalling throughout the frame. Finally, it must be noted that the Vulkan API prohibits barriers from being executed within render passes, so minimizing the amount of render passes by grouping work together also automatically minimizes the amount of *grace points* throughout the frame at which a barrier may be executed, and so naturally leads to grouping of barriers.

2 RELATED WORKS

One of the earlier public works on merging render passes inside a frame graph is Hans-Kristian Arntzen's *Granite* engine [Arn17]. In Granite, each node represents an entire logical render pass that is then translated into a Vulkan subpass. The algorithm employed for render pass merging is executed after the graph has been sorted into a linear execution order and tries to produce a new ordering that greedily maximizes reasonable scoring heuristics of subpass attachment reads being forcibly merged with the writer pass, dependent subpasses being as distant from each other as possible, and not ending a pass for as long as possible. This engineering approach to the problem fits mobile-oriented development quite naturally, as it focuses on various subpass interactions that often occur in mobile Vulkan-based applications, but lacks rigorous analysis of effectiveness of the algorithm.

The approach Granite takes for its user-facing API is what may be called a *coarse-grained* approach, meaning

that each node represents an entire Vulkan-style subpass, or even an entire render pass for engines that do not target mobile devices. As far as the authors are aware, most commercial engines take a similar approach to their frame graph API: Unity [Tec; TAC21], Unreal [Epi], Frostbite [ODo17], one of Activision's engines and AMD's RPS [Adv] all use coarse-grained nodes. Among these, it is publicly known that render pass merging is performed only for Activision's Task Graph Renderer, according to their talk at Rendering Engine Architecture Conference 2023 [Cha23], although algorithmic details are not available publicly. This is to be expected, as for a render graph runtime with coarse-grained nodes rendering code is already grouped together based on what render pass it belongs into, except for the case of mobile rendering, where still manual pass management is usually employed.

An alternative approach is what can be referred to as *fine-grained* nodes. With this approach, each node represents a computation that may be smaller than a single pass or subpass, and hence each pass in the resulting GPU command stream consists of commands recorded by multiple nodes. Consider an opaque G-buffer render pass. Coarse-grained approach suggests that there should exist a single node responsible for this pass that dispatches rendering of multiple systems, plugins or modules, while a fine-grained approach suggests that any system that needs to render opaque geometry to the G-buffer should create its own node with its own «virtual» render pass and let the frame graph runtime merge these nodes into a single pass. This approach provides architectural benefits to the developer:

— no additional event-like system needs to be employed to make passes present in the engine by default customizable by optional subsystems;
— if desired, it is possible for an optional subsystem to forcibly break a render pass mid-way to read an intermediate result of rendering without modifying other code, for example, to create a hierarchical culling Z-buffer [GKM93] that contains only static geometry.

Note that in the case of fine-grained nodes, it is crucial for the runtime to have a robust render pass merging algorithm, as a single logical pass might consist of dozens of subsystem nodes. On the other hand, while coarse-grained nodes do nott technically require such an algorithm, it is still desirable to have one, as extensibility of passes being handled outside of the frame graph runtime can lead to subsystems creating passes that are identical to other existing passes throughout a project's long lifetime or when the code that creates the initial pass is inaccessible for the developer.

As for the algorithmic problems described in this paper, to the best of authors knowledge, they are completely novel and have not been tackled previously.

Our Contribution

We focus on fine-grained frame graph runtimes and attempt to solve the render pass merging problem, although our results can also be applied to coarse-grained runtimes, or even Godot-like render graphs. We consider only the simplest case of each resulting render pass being constrained to have a single subpass, leaving multi-passes for future work. We formulate a rigorous statement of the *minimal lambda graph partition* problem that we consider to be general enough to be applicable to any frame graph system, the solution to which is then used to produce a node execution order that is optimal in its amount of render pass breaks. This problem is then analyzed and shown to be NP-complete even under some sensible regularity conditions. Finally, we present a greedy approach for solving this problem that results in graph execution orders that have about 30% less render passes than a baseline approach that closely matches that of the Granite engine.

3 EARLY STREAK SEGMENTATION

Mathematical model

For the purposes of this paper, the following mathematical model of a frame graph will be used.

Definition 1. A Λ-graph is a tuple (V,P,C,L), where (V,P) is a directed acyclic graph, (V,C) is an undirected graph, and $L : V \to \mathbb{N}$.

The set V represents nodes present in a frame graph; the set of directed edges P represents precedence order between nodes, constraining possible execution orders; the set of undirected edges C models pairs of nodes that are in conflict with each other and require a barrier to be executed between them; and finally the label function L represents unique frame buffers used by each node, or more generally, any mutually exclusive global state that is expensive to change on the target platform. A possible frame graph of an application represented through this definition can be seen on figure 1.

Informally, the problem we are trying to solve is as follows. Find a topological sort of (V,P) and a partition of it into as few contiguous sequences as possible, such that within each sequence there are no conflicting nodes and all labels are the same.

When faced with this problem, the first approach that comes to mind is choosing the next node to be executed to have the same frame buffer as the previous one, which is close to Granite's approach. It must however be noted that depending on the algorithm chosen for topologically sorting, the optimal solution may not be achievable at all. Both a depth-first and breadth-first traversal based topological sorting algorithms are restricted in the set of possible orders of traversals, and the optimal order may lie outside this restricted set. This motivates us to

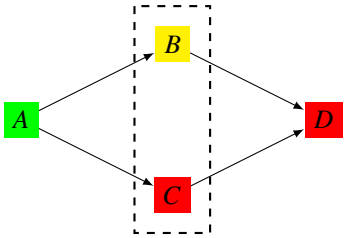

Figure 2: Simplest example where baseline Granite-like approach fails. Colors represent different frame buffers (including lack thereof), dashed outline represents the set of nodes among the which the algorithm must make a choice, node A already being scheduled as the first one. If C is selected as the next node to be executed, the next one has to be B, and so the resulting order is A, C, B, D and the algorithm has failed to merge C and D into a single pass.

always prefer Kahns algorithm [Kah62], which enables achieving any valid topological sort order from a graph by varying the strategy of selecting the next node to be processed. And so the first naive algorithm in its fullness would be using Khans topological sorting algorithm and prioritizing nodes with the same frame buffer as the last processed node when choosing the next node to process. This, however, does not yield satisfactory results on relatively straightforward examples, e.g. figure 2, which motivates the following formalization of the problem.

Definition 2. A *condensation* of a directed graph with respect to a partition of the node set $V = V_1 \sqcup ... \sqcup V_s$ is a directed graph (W, Q), where $W = (1, ..., s)$ and $Q = \left\{(i, j) \mid \exists v \in V_i, \exists w \in V_j, (v, w) \in P \wedge i \neq j\right\}$.

Definition 3. Call a partition of the node set *correct* for a Λ-graph if it satisfies the following:
1) $\forall i, \forall v, w \in V_i, \lambda(v) = \lambda(w)$;
2) $\forall i, \forall v, w \in V_i, \{v, w\} \notin C$;
3) and the condensation w.r.t. to it is acyclic.
The subsets V_i constituting the partition will be referred to as *buckets*.

Definition 4. The problem of finding a correct partition of minimal size s is called *minimal lambda graph partition* and denoted as MLGP.

The statement of MLGP can informally be understood as partitioning a frame graph into render passes before it is sorted, and correctness conditions correspond to the fact that it should be possible to produce a node execution order with render passes corresponding precisely to partition buckets.

In presence of a minimal partition for the lambda graph, a corresponding execution order can be achieved as shown in algorithm 1. This algorithm is de-facto equivalent to sorting the condensation graph first and then sorting nodes within each partition set according to edges from P afterwards.

Next, note that when an optimal solution to the initially stated informal problem of sorting the graph while minimizing the amount of render pass breaks is available, a correct partition that will yield the same amount of passes when sorted with algorithm 1 can clearly be constructed. Hence constructing a minimal partition and then sorting the graph while using it will indeed solve the initial informal problem.

This mathematical statement is not exclusive to render-pass related problems, as each set of the partition simply represents a contiguous streak of nodes that can be executed with no barriers or global state changes (e.g. frame buffer changes) as represented by L. Following this line of thought, we call this approach *early streak segmentation*, as opposed to segmenting the graph into streaks mid-flight while sorting it.

Analysis of MLGP

First of all, MLGP $\in$ NP, as checking the correctness of a partition can be done in polynomial time. Moreover, clearly,

Claim 1. MLGP is NP-complete.

Proof. Observe that a correct partition of a lambda graph $(V, \varnothing, C, \mathrm{id})$ is equivalent to finding the chromatic number of (V, C). □

Algorithm 1 Modification of Khan's algorithm that builds an execution order based on a correct partition of a Λ-graph.

```
for v ∈ V do
    D_v ← out-degree of v in (V, P)
end for
for i ← 1 to s do
    E_i ← num. edges (v, w) ∈ P s.t. v ∉ V_i and w ∈ V_i
end for
F ← nodes with out-degree 0
R ← empty list
Denote V_v^{-1} = i when v ∈ V_i
repeat
    v ← any element w of F s.t. E_{V_v^{-1}} = 0
    if R not empty then
        p ← last element of R
        v ← any element of F ∩ V_p^{-1}
    end if
    Append v to R
    for w → v do
        Decrement D_w
        Decrement E_{V_v^{-1}}
        if D_w = 0 then
            Add w to F
        end if
    end for
until no elements remain in F
return R
```

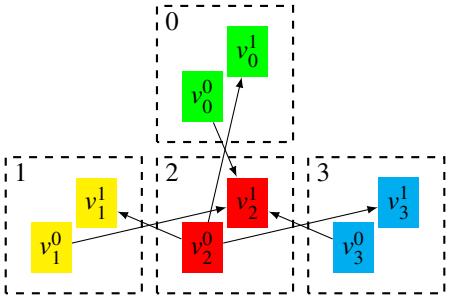

Figure 3: The construction described in claim 2. The initial graph is $V' = \{0,1,2,3\}$, $E' = \{\{0,2\},\{1,2\},\{2,3\}\}$. Nodes in V' are depicted through dashed rectangles, and the corresponding Λ graph nodes are located inside the rectangles, where colors represent different labels. The maximal anti-clique is $\{0,1,3\}$, and merging the corresponding nodes pairwise minimizes the partition.

Calculating chromatic numbers for arbitrary graph is known to be a notoriously hard problem, and so some regularity conditions must be enforced on Λ-graphs that we are to consider. Inspired by other fields of study, as well as our observation of graphs of various applications, we formulate a requirement analogous to lack of data races inside a multi-threaded program.

Regularity condition 1. $\forall\{v,w\} \in C, v \rightsquigarrow w \vee w \rightsquigarrow v$, where $\rightsquigarrow$ denotes reachability. In other words, any conflict is ordered by precedence edges.

We have observed this restriction to be sensible for actual applications, and so have enforced it in the design of the user API for our frame graph runtime. One case where enforcing it has proven to be a problematic is careless superfluous usage of decompressed read-only depth attachments in legacy code, but such problems can and should be resolved on the application programmer side and not inside a frame graph runtime, as they lead to performance issues in any case due to decompressing and recompressing the depth buffer.

This condition is not enough to make the problem tractable, although the proof is less trivial.

Claim 2. Even under the regularity condition 1, MLGP is NP-complete.

Proof. We shall present a polynomial time reduction of the maximal anti-clique problem to MLGP. Consider an arbitrary graph (V', E'). The construction is then as follows:
1) $V = \bigcup_{i \in V'} \{v_i^0, v_i^1\}$;
2) $E = \bigcup_{(i,j) \in E'} \{(v_i^0, v_j^1), (v_j^0, v_i^1)\}$;
3) $C = \varnothing$;
4) $L(v_i^0) = L(v_i^1) = i$.
A visual representation of this construction in a simple case is shown in figure 3.

Observe that any partition that satisfies the first 2 correctness conditions essentially selects a subset of V' by either merging the two nodes corresponding to some initial vertex or leaving them in separate partition buckets. Moreover, the set of vertices in V' whose pairs of nodes were merged form an anti-clique as long as the third correctness condition is satisfied, as otherwise a cycle would form in the condensation precisely along the edges generated for an initial edge in E' that prevented the set from being an anti-clique. On the other hand, for any anti-clique in (V', E'), the corresponding partition is correct by construction. Finally, note that the size of an anti-clique S and the size of the partition s are related as $|S| = |V| - s$. Hence, finding a minimal correct partition would indeed yield an anti-clique of maximal size in the initial graph. □

This fact, however, should not discourage one from trying to solve MLGP in practice, as the construction presented above, per our informal observations, is a characteristic representation of cases where a greedy approach presented below fails to construct a minimal partition. On the other hand, the construction is contrived, as such configurations rarely occur in practice and make little sense from the standpoint of computational graphs.

Greedy Algorithm

We propose a greedy algorithm that is based on the idea of *stacks*, but to justify it we require some additional facts about optimal solutions. Below, the notation $\rightsquigarrow$ is used to represent reachability along edges P or along edges of the condensation graph, and V_i^l is used to denote buckets whose nodes all have $L(v) = l$. Note also that the contents of this subsection all assume regularity condition 1 to hold.

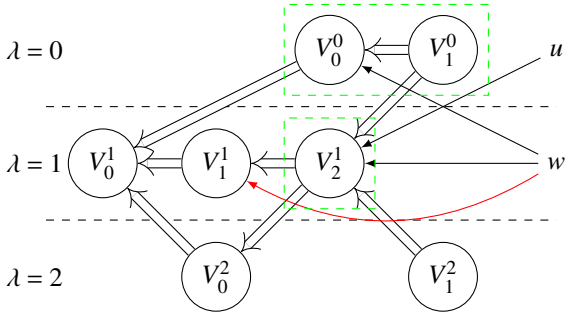

Figure 4: A visualization of the proposed approach. Circled nodes and thick arrows represent the condensation graph, u and w are nodes which are yet to be added into one of the sets, possibly a new one. Thin arrows represent precedence, red ones among them correspond to conflicts. Nodes and sets are laid out according to their labels as 3 stacks. Admissible sets for u and w are depicted using a green dashed outline. If the «deepest is best» greedy bucket selection strategy is used, w must be placed before u to avoid creating a new bucket.

Claim 3. In a minimal correct partition of a Λ graph, for any two buckets V_i^l and V_j^l with nodes in both having the same labels, either $V_i^l \rightsquigarrow V_j^l$ or $V_j^l \rightsquigarrow V_i^l$.

Proof. If two such buckets are not ordered, we are able to merge them together into a single bucket, which would contradict minimality. The resulting partition will still satisfy the first two correctness conditions, as the labels match, and any conflict between their nodes would have resulted in a path between them due to regularity condition 1. Finally, a cycle occurring in the condensation after the merge would imply a path between the two buckets, which is a contradiction. □

For this reason, we only try to construct partitions that have this property of buckets corresponding to a certain label being linearly ordered. This fact will also henceforth be abuse to assume that the notation V_i^l indexes buckets in this exact linear order, and so $i < j \Leftrightarrow V_i^l \rightsquigarrow V_j^l$. For each label l, the sequence of V_i^l is what we refer to as a *stack*. The proposed algorithm will then try to iteratively take nodes in a topological order of $v < w \Leftrightarrow v \leftsquigarrow w$ and attempt to place each node into one of the buckets in the stack without violating correctness conditions of the partially built condensation graph. Figure 4 should be referred to for visual intuition of these definitions and claims, as well as the algorithm itself.

Definition 5. When adding a new node v with $L(v) = l$, having only outgoing precedence edges and possibly some conflict edges, into an existing Λ-graph and its partition, we call a bucket V_i^l admissible if:

1) no nodes in V_i^l conflict with v;
2) for each previously existing node $w \in V_j^l$ such that $w \leftarrow v$, it holds that $j \leqslant i$;
3) for each previously existing node $w \in V_j^{l'}$ such that $l \neq l'$ and $w \leftarrow v$, it holds that $V_i^l \not\rightsquigarrow V_j^{l'}$.

Claim 4. Admissibility is monotonic along the stacks. In other words, if V_i^l is not admissible for v, then $\forall j < i$, V_j^l is not admissible too.

Proof. Observe that thanks to regularity condition 1, a conflict implies that there is an edge from v to some node in V_i^l. Inadmissibility due to violating the first condition then makes V_j^l inadmissible per the second condition. If V_i^l failed the second condition, clearly, V_j^l will too due to claim 3, and same reasoning holds for violating the third condition. □

This fact greatly aids in efficient implementation of algorithms, as a lot of bucket candidates can be immediately eliminated.

Finally, observe that if a minimal partition of the graph is already known, it is possible to reconstruct it by growing bucket stacks, iterating nodes in topological order and placing them only into admissible buckets, as a node being placed in an inadmissible bucket will definitely result in an incorrect partition. The only questions that remain is what exact order should nodes be considered in and what admissible bucket should be chosen. As we have shown the problem to be NP-complete, only heuristic approaches make sense, and the heuristics we have chosen is to use an arbitrary order and always prefer the «deepest» admissible bucket, i.e. the one with the smallest index. This is motivated by the fact that a sequence of buckets can accommodate an entire sequence of nodes which are not ordered with anything else but each require a barrier between them, which is a case that does indeed occur in practice for nodes that do compute dispatches. However, in our synthetic tests other «static» strategies like choosing the topmost or the midway admissible bucket did not have a significant influence on the results. Finally, the pseudo-code for the algorithm is shown in figure 2.

Algorithm 2 Proposed greedy solution to MLGP.

```
for l in the image of L do
    S_l ← empty list
end for
F ← nodes with in-degree 0
Chose any topsort order s.t. v < w ⇔ v ⇜⇝ w
for v ∈ V in chosen order do
    M ← ∅
    for w ← v do
        W ← bucket previously chosen for w
        Add all buckets reachable from W to M excluding W itself
    end for
    V_g ← deepest bucket in S_{L(v)} \ M
    if ∃w ∈ V_g s.t. (v, w) ∈ C then
        V_g ← next bucket in S_{L(v)}
    end if
    Put v into V_g
    if V_g ill-defined then
        Create new bucket and append it to S_{L(v)}
    end if
end for
return All buckets in S
```

4 EXPERIMENTAL RESULTS

The proposed algorithm was implemented in python with $O(|V|^3)$ time complexity and $O(|V|^2)$ space complexity. Random graphs were then used for comparing the proposed approach with a baseline and random sorting. The graph of one of released games that we have access to was observed to have a 0.025 probability of having an edge between two nodes, a 0.013 probability of having a conflict between nodes, and 0.2 proportion

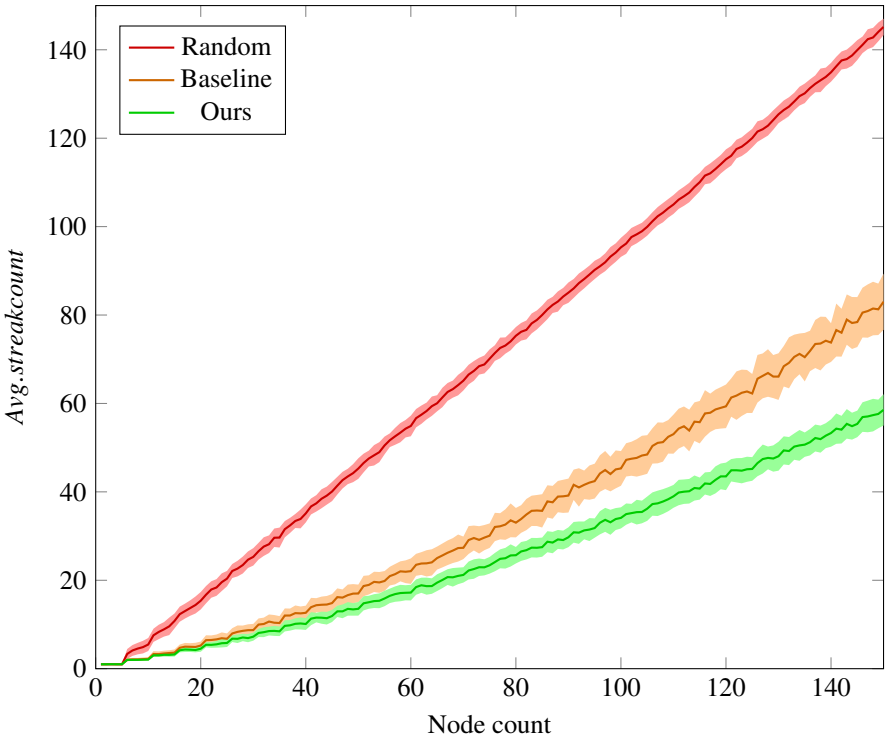

Figure 5: Comparison of proposed streak segmentation approach to baseline and random choice, less is better. For each graph size a sample set of 100 random graphs was generated and test results were averaged, confidence intervals shown as painted regions around plots.

of nodes having a unique frame buffer. The node count of this graph exceeds 100, which is characteristic of fine-grained systems. A sequence of sets of random graphs was then generated, distributed according to these parameters and three node sorting approaches were tested, see figure 5. As one might expect, randomized sorting results in render pass count being proportional to node count. For the baseline approach, Granite-like eager prioritization of continuing the current pass while sorting nodes using Khan's algorithm was chosen. The plots demonstrate that the suggested approach gives a significant improvement over the baseline of 30% on average. It is also surprising that the proposed algorithm results in lower standard deviation, which implies more predictable performance with respect to changes in the graph when used in practice.

Currently, two live-service games with principally different fine-grained frame graphs are available to us, one targeted at desktop computers, one at mobile devices. Both games currently use manual barrier placement to achieve best possible performance, and furthermore, the latter uses manual Vulkan render pass placement. To move away from this manual approach, a robust render pass merging algorithm was required, and when we attempted to do so using Granite's approach, we have observed the result to be less optimal than the manual approach, which made it impossible for us to gather practical performance data from mobile devices before a robust pass merging algorithm was devised and tested in isolation.

However, we were able to preliminarily integrate the algorithm into the desktop title and observe that it likely constructs the optimal render pass segmentation for the current frame graph of the game. The graph consists of 171 nodes and uses 36 unique frame buffers on ultra graphics presets. By randomizing the node sorting order and recompiling the graph 10000 times, we observed the streak (render pass) count to vary from 54 to 58 with mean $\mu = 56.22$ and variance $\sigma^2 = 0.6$ when no render pass merging approach is used. As nodes are declared in an unspecified order due to being distributed between various modules, plugins and subsystems of an application, any of these results might occur in practice, which leads to undesirable non-deterministic performance. When the baseline approach is used, the graph is sorted to have 54 streaks independent of further order randomization, which alleviates non-deterministic performance concerns. But with the proposed approach, the streak count decreases to exactly 52 independent of randomization, which is explained by situations as shown on figure 2 being present in the graph. The fact that randomization could not lower the streak count to 52 throughout the 10000 iterations can be explained by the fact that the amount of correct topological orders for a graph behaves asymptotically as $O(n!)$ in general, mak-

ing the probability of the optimal order being selected at random increasingly small.

It must be noted that this frame graph has a reasonable amount of legacy code which is yet to be integrated with the frame graph runtime, and so results are expected to improve in the future as more knowledge about the renderer's structure and more node reordering opportunities are made available to the runtime.

The preliminary C++ implementation of the algorithm stores reachability info of the condensation graph in a bit matrix and heavily uses SIMD instructions to alleviate the high asymptotic complexity of the algorithm, which results in partitioning being executed in under 200 µs on desktop PCs.

5 CONCLUSION

A seemingly simple problem of merging render passes appears to have much more depth to it than we initially anticipated. Implementing a two-pass approach of first partitioning the graph and only then sorting it enables saving almost a third of render pass breaks on synthetic tests, and hence tile cache flushes in practice. Moreover, although we initially started with render pass merging, we quickly found out that the rigorous problem statement through Λ-graphs naturally allows for grouping of barriers too, and is applicable not only to nodes that are parts of render passes, but to compute nodes too, as a contiguous streak of nodes without is a general enough to handle both.

Future Work

It is reasonable to think that the current version of the algorithm can be improved further. Time complexity can likely be improved without loss of quality, as well as the C++ implementation speed. Furthermore, it is reasonable to think that there are render pass breaks to be saved by using more advanced heuristics for choice of next node to be bucketed and choice of the bucket.

More work in the direction of handling multi-passes is to come, i.e. render passes that consist of multiple subpasses. Technically, any two nodes that do rendering and do not require a barrier on any paths between them in P can be merged together into a single render pass with two subpasses, even if their frame buffers completely different, but it is not clear how such an approach would affect performance, so we suspect that heuristics represented mathematically as a label «closeness» metric will be required. Subpass attachment reads is another optimization that is crucial for mobile performance but is yet to be incorporated into our approach.

Finally, we are interested in applying profile guided optimization to render pass merging, as well as to render graph runtimes in general. Different execution orders may be optimal for different platforms, and so measuring GPU timings of node execution and then using them as guides for a hybrid merging algorithm may lead to complete elimination of manual command list level cache optimizations from daily lives of rendering engineers, enabling development of applications at a higher abstraction level and increasing productivity.

6 REFERENCES

[Adv] Inc. Advanced Micro Devices. *AMD Render Pipeline Shaders source code*. URL: https://github.com/GPUOpen-LibrariesAndSDKs/RenderPipelineShaders.

[Arn17] Hans-Kristian Arntzen. *Render graphs and Vulkan — a deep dive*. 2017. URL: https://themaister.net/blog/2017/08/15/render-graphs-and-vulkan-a-deep-dive/.

[Ban24] Darío Banini. *GPU synchronization in Godot 4.3 is getting a major upgrade*. 2024. URL: https://godotengine.org/article/rendering-acyclic-graph/.

[Bav14] Louis Bavoil. In: (2014).

[Cha23] Francois Durand Charlie Birtwistle. "Task Graph Renderer at Activision talk at REAC conference". Rendering Engine Architecture Conference. 2023.

[Epi] Inc. Epic Games. *Unreal Engine Render Dependency Graph*. URL: https://docs.unrealengine.com/5.0/en-US/render-dependency-graph-in-unreal-engine/.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. "Hierarchical Z-buffer visibility". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, pp. 231–238.

[Kah62] A. B. Kahn. "Topological sorting of large networks". In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562.

[ODo17] Yuriy O'Donnell. "FrameGraph: Extensible Rendering Architecture in Frostbite". Game Developers Conference. 2017.

[TAC21] Natalya Tatarchuk, Sebastian Aaltonen, and Timothy Cooper. "Unity Rendering Architecture". SIGGRAPH 2021 REAC. 2021.

[Tec] Unity Technologies. *Unity render graph system*. URL: https://docs.unity3d.com/Packages/com.unity.render-pipelines.core@10.2/manual/render-graph-writing-a-render-pipeline.html.

[Wih19] Graham Wihlidal. "Halcyon: Rapid innovation using modern graphics". Reboot Develop. 2019.