Fast parentheses matching on GPU

Raph Levien
Google
San Francisco, CA, USA

ABSTRACT

Parentheses matching is an important fundamental algorithm, with applications including parsing and processing of tree-structured data. Previous literature presents work-efficient parallel algorithms targeting an abstract PRAM machine, but does not address modern GPU hardware. This paper analyzes the parentheses matching problem using two monoids, the bijective semigroup and a novel "stack monoid," and presents a practical, fast algorithm interleaving these two monoids to map to the thread, workgroup, and dispatch levels of the GPU hierarchy. This algorithm is implemented portably using compute shaders, and performance results show that the algorithm operates at a significant fraction of the raw memory bandwidth of a typical GPU.

ACM Reference Format:
Raph Levien. 2022. Fast parentheses matching on GPU. In *Proceedings of High-Performance Parallel and Distributed Computing (HPDC'22).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper presents an efficient parallel solution to the *parentheses matching problem,* tuned for high throughput on GPU hardware. Parentheses matching is an important subproblem of parsing, and is a general building block for disparate other algorithms, including bin packing[AMW89] and tree pattern matching[PJ20]. The immediate motivation for the present work is calculating clipping rectangles for each node in a tree representing a 2D rendering task.

1.1 Motivation

There are many GPU algorithms operating on large, flat arrays of values, but fewer for processing tree-structured data. At the heart of many such tree problems, including parsing a sequential representation of a tree into a usable tree structure, is the *parentheses matching problem.* Since the '80s there have been theoretical results suggesting that efficient parallel implementation is possible, but the literature has not to date produced practical algorithms that run on contemporary GPU hardware.

The specific motivation for this work was tracking clip bounding boxes for a 2D renderering engine in which the scene is represented as a tree of drawing operations, with individual objects such as images and vector paths as leaf nodes, and operations such as clip, blend, and transform as parent nodes. An earlier version of the code tracked these bounding rectangles on the CPU, but that was a significant bottleneck, so we sought to move the tracking to the GPU. Parentheses matching is the key subproblem of that task, associating the end of the scope of each clip operation with the beginning, and deriving tree structure in a way that intersection of clip rectangles for all nodes insides a clip can be readily computed. The parentheses matching problem is general, however, and an efficient algorithm will help make parsing and other related tasks practical for GPU implementation.

1.2 Limitations of current state of the art

The literature on the parentheses matching problem goes back decades (at least to [BOV85]), but until now there is no known satisfactory solution running on actual GPU hardware. The literature falls into several categories:

- Theoretical presentations of work-efficient algorithms analyzed in terms of an abstract Parallel Random Access Machine PRAM) model but no clear mapping to an efficient GPU implementation ([BOV85], [LP92], [PDC94]).
- Practical algorithms which run on GPU but have a work factor dependent on maximum nesting depth ([Hsu19]).
- More limited GPU-based parsing algorithms which cannot handle arbitrary tree structure ([SJ19]). This category also includes the use of standard generalized prefix sum algorithms with a small fixed bound on nesting depth.

Thus, the prevailing wisdom remains that parsing of arbitrary tree structured data is inherently a serial problem and must be done on CPU rather than GPU.

1.3 Key insights and contributions

There are several key insights in this paper, culminating in presentation and empirical performance measurement of an algorithm that is fast and practical to implement on standard GPU hardware.

The first insight is that the parentheses matching problem can be expressed in terms of two monoids, both of which can be used to compute matches, but with different time/space tradeoffs. The first of these is the well-known bicyclic semigroup which is cheap to compute and can be queried by binary search, and the second is a "stack monoid" which takes more space but can be queried in $O(1)$ time. Either by itself can be used to derive an algorithm which is parallel but has $O(\log n)$ work factor.

The second insight is that *interleaving* these two approaches yields a work-efficient algorithm. Further, the two approaches map well to the hierarchical structure of actual GPU hardware. We present a simple algorithm consisting of reduction of the stack monoid (computing stack snapshots at partition granularity), followed by binary search of the bicyclic semigroup to resolve matches within a partition. The second step can be done within a workgroup, using efficient shared memory. It is reasonably fast but not truly work-efficient.

HPDC'22, June 2022, Minneapolis

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A faster version of the algorithm adds a third level of hierarchy: a sequence of k elements processed per thread, instead of just one as in the simpler algorithm. This technique is analogous to that used for high performance prefix sum implementations, but requires more sophistication. Stack monoid reduction is used for the smallest granularity, then binary search for the workgroup level, and then stack monoids again for finding matches across workgroup boundaries.

1.4 Experimental methodology and artifact availability

The primary empirical claim is that the proposed algorithm is fast on standard GPU hardware. To demonstrate this claim, we run the code on an AMD 5700 XT as Vulkan compute shaders. The test consists of a random sequence of parenthesis. The GPU time is measured with Vulkan timer queries.

All software is available on GitHub with a permissive Apache 2 open source license[1]. The infrastructure for running and measuring compute shader performance is cross-platform and runs on Metal and Direct3D 12 as well as Vulkan. Such cross-platform infrastructure is unusual for compute-centric tasks, though it is relatively common in game engines.

1.5 Limitations of the proposed approach

The main limitation of the proposed algorithm is that the presentation and implementation is a 2-dispatch pipeline and is limited to inputs of w^2k^2, where w is workgroup size and k is the number of elements processed per thread. In many cases it is possible to increase k to accommodate the problem size (as is the case for the motivating 2D graphics example), but as inputs scale up the algorithm would need to be extended to 3 or more dispatches.

Parentheses matching in isolation is not an especially useful task. To put this technique into practice will require integration with other subsystems that can utilize parentheses matching as a subtask. For example, parsing of textual tree-structured data formats such as XML and JSON would also require lexical analysis.

2 THE PARENTHESES MATCHING PROBLEM

The classical version of the parentheses matching problem is, for every index in the source string, find the index of the corresponding matching parenthesis. This paper actually considers a stronger version of the problem: for every closing parenthesis, find the index of the matching open parenthesis. But for every opening parenthesis, find the index of the immediately enclosing opening parenthesis. It is straightforward to reconstruct the traditional version, but the converse is not true.

One statement of the problem is as a simple sequential program which uses a stack, as shown in Figure 1.

We will be concerned with *snapshots* of the stack at step i. An appealing quality of this specific formulation of the parentheses-matching problem is that all stacks can be recovered from the output, just by repeatedly following references until the root is reached (here represented by a value of -1).

[1] `https://github.com/linebender/piet-gpu`

```
stack = [-1]
for i in range(len(s)):
    out[i] = stack[len(stack) - 1]
    if inp[i] == '(':
        stack.push(i)
    elif inp[i] == ')':
        stack.pop()
```

Figure 1: Sequential algorithm for parentheses matching

3 THE BICYCLIC SEMIGROUP

The theoretical derivation of the algorithm relies heavily on the *bicyclic semigroup,* actually a monoid, which is well known to model the balancing of parentheses. An element of the bicyclic semigroup Bic can be represented as a pair of nonnegative integers, with $(0, 0)$ as an identity and the following associative operator:

$$(a, b) \oplus (c, d) = (a + c - \min(b, c), b + d - \min(b, c))$$

An open parenthesis maps to $(0, 1)$ and a close parenthesis maps to $(1, 0)$. We will overload the function $\mathrm{Bic}(s)$ over a string to result in the $\oplus$-reduction of this mapping applied to the elements of the string; thus $\mathrm{Bic}(\texttt{'))()('}) = (2, 1)$. We will use slice notation on strings; $s[i..j]$ represents the substring beginning at index i of length $j - i$.

The bicyclic semigroup gives rise to an alternate definition of the parentheses matching problem. In particular, parenmatch(s)$[j]$ is the maximum value of i such that $\mathrm{Bic}(s[i..j]).b = 1$. This is one less than the minimum value such that the b field is 0. Note that $\mathrm{Bic}(s[i..j]).b$ is monotonically increasing as i decreases.

4 THE STACK MONOID

Another related monoid is the *stack monoid,* which is a sort of hybrid of the bicyclic semigroup and the free monoid. Essentially, rather than just counting the number of stack pushes, it contains the actual values pushed on the stack.

Like the bicyclic semigroup, the stack monoid can be represented as a 2-tuple. The first element in the tuple is the number of unmatched closing parentheses, the same as the bicyclic semigroup. The second element is a *sequence* of values corresponding to unmatched open parentheses, as opposed merely to their count as in the bicyclic semigroup. In the context of this paper, those values are typically the indices, though the monoid is free in that it can be defined over any sequence element type.

The empty stack monoid is $(0, [])$. The value corresponding to an open parenthesis with associated value x is $(0, [x])$, and the value corresponding to a close parenthesis is $(1, [])$. The combination rule is as below:

$$(a_0, l_0) \oplus (a_1, l_1) = (a_0 + a_1 - \min(|l_0|, a_1), l_0[..\max(0, |l_0| - a_1)] + l_1)$$

Like the bicyclic semigroup, the stack monoid lends itself to a straightforward definition of the parenthesis matching problem. A reduction of the stack monoid over a prefix of the input represents a snapshot of the stack, as computed by the sequential algorithm, up to the end of that slice. The result of the parentheses matching algorithm is then the top of the stack at each step.

The parenthesis match value at j is the topmost value of stack snapshot taken at position j. Here we use $enum(s)$ to represent the enumeration of the indices of the sequence s, for example, $enum$(')) (') is the sequence [(0, ')'), (1, ')'), (2, '(')].

$$parenmatch(s)[j] = last(\mathrm{Stk}(enum(s)[..j]))$$

The k-suffix of the stack monoid is simply the last k values.

The storage required by a single stack monoid value is unbounded, but that does not preclude efficient implementations. In particular, the combination of two values of size k can be done in-place by $2k$ processors in one step. This result generalizes to combination of a vector of values, which can be represented as a stream compaction.

When a sequence containing only unbalanced close parentheses (and no unbalanced open parentheses) is appended to a first sequence, the resulting stack monoid is a prefix of that of the first sequence. Stated more formally:

$$rev(\mathrm{Stk}(enum(s)[i_0..i_2]))[k] = rev(\mathrm{Stk}(enum(s)[i_0..i_1]))[k+j] \quad \text{where } \mathrm{Bic}(s[i_1..i_2]) = (j, 0)$$

The significance of this relation is that, given the value of the Bic monoid and a materialized stack slice, it is possible to resolve queries in $O(1)$ time. In cases where $\mathrm{Bic}([s_1..i_2])$ has a nonzero $.b$, the match is found within the slice $s[i_1..i_2]$; a general matching algorithm will resolve the match inside the slice if $.b \neq 0$, and use $.a$ to index into a stack monoid value in $s[i_0..i_1]$ when $.b = 0$.

5 CORE PARALLEL ALGORITHM

The core parallel algorithm is a binary search over the bicyclic semigroup. That algorithm by itself is fully parallel and reasonably efficient; it has a work factor of $O(\log n)$ for the binary search.

Before running this algorithm, a binary tree of bicyclic semigroup values is constructed; this is the same as the up-sweep phase of a standard parallel prefix sum implementation. Specifically, the leaf nodes of the tree are defined by $tree[0][i] = \mathrm{Bic}(s[i])$, and parent nodes by the relation $tree[j+1][i] = tree[j][2i] \oplus tree[j][2i+1]$. Construction of this tree takes $\lg n$ steps, and the tree itself requires storage of $2n - 2$ bicyclic semigroup elements.

Then, for each index i, the algorithm shown in Figure 2 searches the tree for a parentheses match.

On termination, i contains the smallest value such that $\mathrm{Bic}(s[i..i_1]).b = 0$, thus $i - 1$ is the solution to the parentheses matching problem.

Operation of the algorithm is illustrated in Figure 3. Here, i_1 is 14 (of a 16 element sequence), and the final value of i is 4, indicating that $\mathrm{Bic}(s[4..14]).b = 0$ but $\mathrm{Bic}(s[3..14]).b = 1$. There is an upward scanning pass followed by a downward scanning pass. At each level, one node from the tree is examined. If combining that node with b would preserve $.b = 0$, it is incorporated (and i adjusted to point to the beginning of the range covered by the node), otherwise it is rejected. Nodes incorporated are marked with a circle, nodes rejected by an X.

This binary search takes $2 \lg n$ steps in the worst case. Thus, while the algorithm is highly parallel, it cannot be considered work-efficient.

$i \leftarrow i_1$
$b \leftarrow (0, 0)$
$j \leftarrow 0$
while $j < \lg w$ **do**
    if i bitand $2^j \neq 0$ **then**
        $q \leftarrow tree[j][\lfloor i/2^j \rfloor - 1] \oplus b$
        if $q.b = 0$ **then**
            $b \leftarrow q$
            $i \leftarrow i - 2^j$
        else
            break
        end if
    end if
    $j \leftarrow j + 1$
end while
if $i > 0$ **then**
    while $j > 0$ **do**
        $j \leftarrow j - 1$
        $q \leftarrow tree[j][\lfloor i/2^j \rfloor - 1] \oplus b$
        if $q.b = 0$ **then**
            $b \leftarrow q$
            $i \leftarrow i - 2^j$
        end if
    end while
end if

Figure 2: Core parallel matching algorithm

6 SIMPLE ALGORITHM

In this section, we describe a simple algorithm which is not strictly work-efficient, but may be practical, especially if the problem is small or if the costs associated with code complexity are significant. For simplicity, it is presented as two dispatches, effective up to a problem size of w^2, where w is the size of a workgroup.

6.1 Stack slices

The first dispatch computes slices of the stack, with each workgroup computing a partition of w values. More precisely, each workgroup computes $\mathrm{Stk}(enum(s)[p..p+w])$, where p is the start of the partition, in this case $w \cdot i$.

This dispatch is very simple. We do a partition-wide reverse scan of the bicyclic semigroup on the mapping of the input elements, followed by a simple stream compaction step: the index is written if the $.a$ of the scan of all following elements is zero, and the memory location to write is derived from the $.b$ value of that scan.

In more detail, for each index i covering the input, index $p + i$ is written to the output at location $\mathrm{Bic}(s[p..p+w]).b - \mathrm{Bic}(s[p+i..p+w]).b$ if the element is an open parenthesis and $\mathrm{Bic}(s[p+i+1..p+w]).a = 0$. For example, in the sequence `')(()('`, the result of the reverse scan of the bicyclic semigroup is [(1, 2), (0, 2), (0, 1) (1, 1), (0, 1)], and the predicate is true at indices 1 and 4, representing the two unmatched open parentheses. The $.b$ values at these indices are 2 and 1, respectively, resulting in 1 being written to output slot 0 and 4 being written to output slot 1.

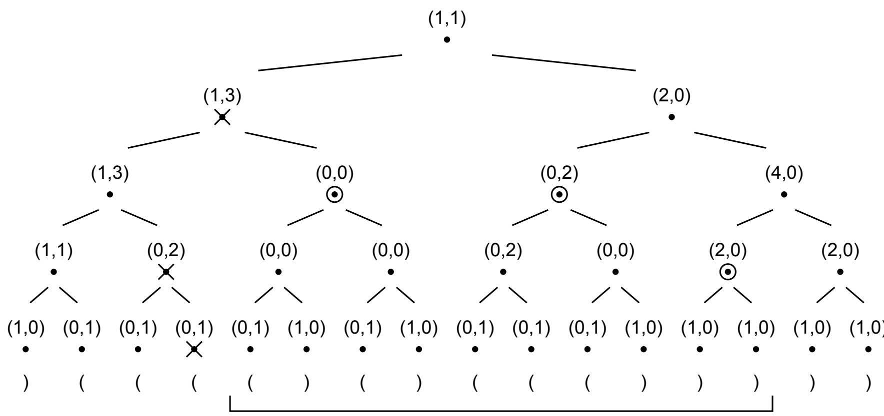

Figure 3: Binary tree search for matching parentheses

In this simpler variant, each thread handles one element, and a simple Hillis-Steele scan[HS86] is used to compute the scan of the bicyclic semigroup.

6.2 Main matching pass

The second dispatch performs the main parentheses-matching task, resolving all matches within the partition, and also using the stack slices generated by the previous dispatch for the rest. Each workgroup handles one partition independently, performing the following steps sequentially (separated by workgroup barriers).

- Materialize the stack for the prefix of the input up to the current partition. This results in the w-suffix of $\mathrm{Stk}(enum(s)[..p])$ in workgroup-shared memory. It consists of a reverse Hillis-Steele scan of the bicyclic semigroups produced in the previous step (up to i), followed by stream compaction which is a per-element binary search of the $.b$ values for the stack value.
- Compute a binary tree of the bicyclic semigroup from the elements in the partition. This is a simple up-sweep as described by [Ble90]. This binary tree requires storage of $2w-1$ bicyclic semigroup elements in shared memory storage, and $\lg w$ steps.
- For each element j, find the least value j such that $\mathrm{Bic}(s[p+i..p+j]).b = 0$, searching the binary tree in an upwards then a downwards pass. The algorithm is very similar to that given in [BOV85].
- If $i > 0$ then the match is found within the partition, and $p + i - 1$ is written to the output. If $i = 0$ then the match is in outside the partition, and the $\mathrm{Bic}(s[p..p + j]).a$ is used to index into the stack as materialized in the first step.

7 WORK-EFFICIENT ALGORITHM

In a PRAM model, a simple Hillis-Steele scan over n elements consists of n processors running $\lceil \lg n \rceil$ steps. Thus, it has a *work factor* of $\lceil \lg n \rceil$ compared to the sequential algorithm running in $O(n)$ steps on one processor.

There are a number of work-efficient variations of the basic Hillis-Steele scan. The most straightforward to implement on GPU is for each thread to process k elements sequentially, amortizing the logarithmic cost over these k elements. In a PRAM model, n/k processors each take $O(\lg(n/k))$ steps, which is work-efficient when $k \geq logn$. See [HSO07] and [SHG08] for more discussion of efficient GPU implementation of scan.

7.1 Work-efficient stack slices

The work-efficient version of the algorithm for producing stack slices is straightforward, and based on standard techniques. We will present it in a bit of detail, as other parts of the algorithm will use similar techniques.

Recall that production of a stack slice is a stream compaction based on a reverse scan of the bicyclic semigroup. The standard work-efficient algorithm for scan is for each thread to process k elements; this way the cost of the Hillis-Steele scan is amortized over k. On an actual GPU, each workgroup will have w threads, so will end up processing wk elements. An argument for the optimality of that approach on an EREW (exclusive read, exclusive write) PRAM is given at the end of section 1.2 of [Ble90].

Applying that technique, the first step is for each thread to do a sequential reduction of the bicyclic semigroup for k elements. Then a standard (reverse) Hillis-Steele reduction over the resulting w elements, which takes $\lg w$ steps. Lastly, each thread does a sequential walk (also in reverse), starting with the exclusive scan value. At each step, the value is written if the $.a$ field of the bicyclic semigroup is 0, and the location is determined from the $.b$ field.

The change to the shader code compared to the $k = 1$ case is modest, and the speedup is significant, contributing to a speedup of over 2.5 for peak throughput (see section 8).

7.2 Work-efficient matching

The work-efficient matching algorithm also processes k elements per thread. Significant attention to detail is required to ensure that sequential iteration over each of these k elements is $O(1)$, in addition to tree build and tree search stages which are $O(\log w)$.

The key steps of the algorithm are outlined as follows (with the reader referred to the commented source code for a more complete presentation):

(1) Reduction of the stack monoid for the prefix preceding this partition. This computes the same stack monoid reduction as in the $k = 1$ case, and is also a stream compaction. In the input stage, each thread processes k input stack monoids, resulting in a *segment* of subsequences. A bitmap records for each input whether it contributed any unmatched open parentheses. The segments with nonzero bitmaps are placed in a linked list data structure (computed as scan of the max operation). Next, a reverse scan of the bicyclic semigroups, one value per segment. In the output stage, each thread is resposible for generating k values. The first value is determined by binary search of the scan result, and successive values are determined by iteration: walking backward through the subsequence from the input monoid until the beginning, then finding the previous nonempty subsequence from the same segment (using the bitmap to identify nonempty subsequences), then following the linked list structure. Each of these queries is $O(1)$.

(2) Building a binary tree of bicyclic semigroup. This the same as the $k = 1$ case, except that each value is a reduction over k input elements. This stage also builds a bitmap for each sequence of k input elements, with a 1 bit for each unmatched open parenthesis in the sequence.

(3) Two binary searches of that tree. The first begins at the first element in the sequence of k input elements processed by the thread. The second begins at the first unmatched open parenthesis in that sequence. The results of these latter searches induces a linked list structure; walking them backwards reconstructs all stack snapshots at multiples of k elements.

(4) Production of parentheses match output. This is a sequential iteration over k elements as well. Maintain an index which might reference a location in the partition, or an index into the prefix stack monoid (the code uses negative values to represent the latter case). This index is initially the result of the first binary search. Also maintain a local stack, initially empty. At each step, if the stack is nonempty, output that value. If empty, use the index to resolve a match value (reading from the prefix stack monoid if outside the partition). Then process the input element. If an open parenthesis, push its index onto the stack. If a close parenthesis, use the hierarchy of previous computations to resolve the next element: if there are nonzero bits remaining in the bitmap (as computed in step 2), use those; otherwise, if the index is in the partition, follow the linked list link as computed in step 3, and lastly, if outside the partition, decrement the index so it references the next element down in the prefix stack monoid.

8 PORTABLE COMPUTE SHADERS

A goal of this work was to develop an algorithm that could be run efficiently and reliably on a wide range of GPU hardware. To this end, we avoided constructs that would pose problems, such as inter-workgroup communication. We also implemented the algorithm on the piet-gpu-hal infrastructure, which runs compute shaders on multiple back-ends, currently Vulkan, Metal, and Direct3D 12.

Our portability layer provides an abstraction over the multiple APIs available for submitting compute jobs to the GPU. The compute shaders are written in GLSL, and compiled to SPIR-V intermediate representation for the Vulkan back-end, as well as translation to HLSL and Metal Shading Language using the standard open-source `spirv-cross` tool for use on Direct3D 12 and Metal, respectively. All shader translation is done ahead of time, for minimal runtime overhead at execution time. The test executable is approximately 1.5 megabytes and requires no additional runtime dependencies aside from the GPU interface provided by the operating system.

The infrastructure allows for runtime query of specific GPU capabilities beyond a baseline set, so different permutations of shaders can be selected for performance or compatibility reasons. Even so, the final implementation of this algorithm was written with compatibility in mind, using workgroup shared memory for communication between threads, and not relying on subgroups or memory barriers, both of which can pose portability challenges.

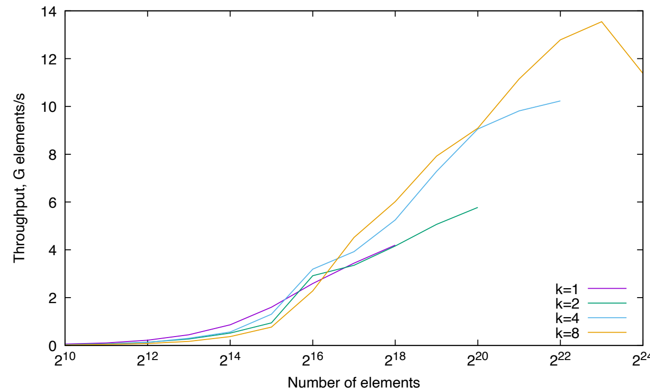

Figure 4: Performance results

9 PERFORMANCE RESULTS

Performance results are shown in Figure 4. All measurements were performed on an AMD 5700 XT graphics card on using the Vulkan API, and clocks set manually to 2GHz. For small problem sizes, throughput is dominated by the overhead of a dispatch. For small problems, $k = 1$ offers the best performance, as the ratio of available hardware threads to number of elements is favorable, so minimizing the amount of work done by any one thread is a winning strategy. As problem size increases to 2^{16} and beyond, larger values of k use the available throughput more efficiently. In addition, the maximum problem size is w^2k^2, so a larger value of k is required to handle problems at the larger end of the scale. An optimum scheduler may adaptively choose compute shaders specialized to a value of k optimum for the hardware.

The peak performance is 13.5 billion elements per second. Since each input and output word is 4 bytes, the memory bandwidth just to read and write the problem is 108GB/s, which is a substantial fraction (27%)of the approximately 400GB/s raw memory bandwidth available on the device. These numbers suggest that the algorithm is operating not far from the theoretical limit of how fast it could possibly go, even if considerably more clever optimizations were applied.

This section will be expanded in the final work; the algorithm has been ported to a wide variety of GPU hardware, but measurement and analysis is not yet complete.

10 RELATED WORK

There is an extensive literature of algorithms for parentheses matching described in terms of the PRAM model. We will briefly survey those. Generally, an algorithm that runs on n processors in $O(\log n)$ time is straightforward, but adaptations to make it work-efficient add significant complexity.

The first work-efficient algorithm in the literature is [BOV85]. The core of this algorithm is essentially equivalent to the up-sweep of the bicyclic semigroup followed by efficient binary search; they don't describe it in terms of a single semigroup, but rather do two passes, one a simple prefix sum for nesting depth, the second an up-sweep using a minimum operation. Certainly on modern GPUs the bicyclic semigroup formulation is superior, as a single pass is more efficient than two, and the calculation of the semigroup itself compiles to a small number of inexpensive machine operations. The work-efficient adaptation depends on scans in both directions.

Much of the following literature is concerned with efficient execution on weaker PRAM variants, in particular EREW (exclusive read, exclusive write) rather than CREW (concurrent read, exclusive write). These concerns don't map well to actual GPU hardware. Indeed, after a dispatch boundary, having many threads read from the same location is a potentially good for performance, due to caching.

An example of work targeting the EREW model is [PDC94]. While the refinement of theoretical execution model is of little interest when targeting actual GPU hardware, their Algorithm II is notable because it is effectively an in-place reduction of the stack monoid presented in this paper, though the monoidal structure was not identified as such in that paper.

The parentheses matching problem is very similar that of deriving parent and left sibling vectors from a depth vector. The depth vector is a representation of tree structure popular in the APL world, and it can readily be derived as a prefix sum of +1/-1 values corresponding to open and close parentheses in the input sequence, respectively. A highly parallel algorithm is given in Section 3.3 of [Hsu19]. This algorithm, however, is not work-efficient, but rather has an additional work factor proportional to the maximum depth of the tree. The present work has no such limitation, and tree depth is unbounded with no impact on performance.

The parentheses matching is related to prefix sum [Ble90]. The latter has both a solid theoretical foundation and a host of practical implementations on actual GPUs, of which a good early example is [SHG08]. We have liberally adapted techniques used for efficient implementation of prefix sum on GPU to develop the algorithm presented here.

11 DISCUSSION AND FUTURE WORK

The parentheses matching problem is similar in many ways to prefix sum, for which there is much work on efficient implementations. In particular, both can be represented as monoids, though the monoidal structure of parentheses matching is trickier than pure sums. In particular, for parentheses matching there are two monoids with different time/space tradeoffs, and only through their interleaving is a work-efficient algorithm possible. This algorithm is not merely theoretically work-efficient in a PRAM model, but maps well to efficient implementation on GPU using techniques similar to existing prefix sum implementations.

The decomposition into monoids has the advantage that the pure parentheses matching problem can readily be fused with other monoid operations. The main motivating example for this work is computing bounding boxes for clipping, which can be modeled as the intersection of rectangles on all paths from the root of a tree to the leaves. Reduction of the stack monoid generalizes easily as it is free and can be extended to include some other monoid such as rectangle intersection in addition to indices. Other related tasks such as parsing likely can be expressed in terms of monoids as well.

One possible extension of the algorithm presented in this paper is scaling up to larger problem sizes than can fit in w^2k^2. There are a few different approaches, depending on the exact application. If the nesting depth can be bounded by wk, then the most straightfoward approach is a standard tree reduction applied at the workgroup granularity approach; this is work-efficient and straightforward to implement. If unbounded nesting depth is required, other tradeoffs exist.

This work uses a portable fragment of the GPU computation model, using workgroup shared memory to communicate between threads within a dispatch, and multiple dispatches to pass data from one workgroup partition to another. The state of the art in high performance prefix sum implementations is decoupled look-back[MG16], which performs a single pass rather than multiple dispatches, and uses message passing between workgroups to propagate the partial results. The primary performance advantage of this technique is eliminating the need to read the input twice, which is important because prefix sum is usually limited by raw memory bandwidth. It may be interesting to apply such a single-pass technique to the parentheses matching problem, though it may be tricky, and it is not obvious that memory bandwidth is the limiting factor. Similarly, it is worth exploring whether subgroups can speed the communication of partial results between threads, compared with workgroup shared memory.

In any case, this work should help make parsing and other manipulation of tree-structured data practical for implementation on GPU, pushing past the common misconception that this work is inherently serial and must be run on CPU.

REFERENCES

[AMW89] Richard J. Anderson, Ernst W. Mayr, and Manfred K. Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation*, 82(3):262–277, 1989.

[Ble90] Guy E Blelloch. Prefix sums and their applications. In *Synthesis of parallel algorithms*, pages 35–60. Morgan Kaufmann Publishers Inc., 1990.

[BOV85] Ilan Bar-On and Uzi Vishkin. Optimal parallel generation of a computation tree form. *ACM Trans. Program. Lang. Syst.*, 7(2):348–357, apr 1985.

[HS86] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[Hsu19] Aaron W. Hsu. *A data parallel compiler hosted on the GPU.* PhD thesis, Indiana University, 2019.

[LP92] Christos Levcopoulos and Ola Petersson. Matching parentheses in parallel. *Discrete Applied Mathematics*, 40(3):423–431, 1992.

[MG16] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled lookback. Technical Report NVR-2016-002, Nvidia, March 2016.

[PDC94] S.K. Prasad, S.K. Das, and C.C.-Y. Chen. Efficient EREW PRAM algorithms for parentheses-matching. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):995–1008, 1994.

[PJ20] Štěpán Plachý and Jan Janoušek. On synchronizing tree automata and their work–optimal parallel run, usable for parallel tree pattern matching. In *SOFSEM 2020: Theory and Practice of Computer Science*, pages 576–586, Cham, 2020. Springer International Publishing.

[SHG08] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for gpus. Technical Report NVR-2008-003, Nvidia, 2008.

[SJ19] Elias Stehle and Hans-Arno Jacobsen. Parparaw: Massively parallel parsing of delimiter-separated raw data. *CoRR*, abs/1905.13415, 2019.