Fast parentheses matching on GPU

Raph Levien
Google

January 4, 2022

Abstract

Parentheses matching is an important fundamental algorithm, with applications in processing of tree-structured data, parsing, and others. Theoretically efficient parallel algorithms have been known for some time, but adapating these to actual GPU hardware is not straightforward, in part because of the complexity of these algorithms. This paper restates the parentheses matching problem in terms of two monoids, both of which reveal parentheses matches but with different time-space tradeoffs, and presents a work-efficient solution that interleaves the two. This algorithm maps well to compute shaders running on a wide range of actual GPU hardware, and we present performance results.

1 Introduction

The parentheses matching problem has a simple, efficient sequential algorithm, and parallel algorithms have been well-studied as well. The literature contains a variety of work-efficient algorithms specified in terms of a parallel random-access machine (PRAM) architecture. These results demonstrate that the problem contains inherent parallelism, and suggest that implementation on GPU hardware may be possible, but it is not obvious how to adapt any of these published algorithms to actual GPU code.

This paper presents a new algorithm, with some similarities to the published algorithms, but carefully crafted to match the hierarchy of GPU hardware. Specifically, it maximizes the amount of work done within a workgroup, thus taking advantage of the fast communication and synchronization within that boundary. We present the algorithm in two versions: a simple version that is reasonably fast, and a work-efficient variant that increases performance further. The difference between the two is the number of elements processed per thread. In the simple version, each thread processes one element, while in the more sophisticated version it processes more than one ($k > 1$).

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.

```
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()
```

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).

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 increases.

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.

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

5 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.

5.1 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 this simpler variant, each thread handles one element, and a simple Hillis-Steele scan[HS86].

5.2 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 forward 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.

6 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. 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] for more discussion of efficient GPU implementation of scan.

6.1 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 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 actual shader code is straightforward, and the speedup significant. (TODO: probably refer to quantitative measurements later)

6.2 Work-efficient matching

This section needs to be expanded to be clearer, but the basic ideas are presented in bullet form:

- Stream compaction of stack. Breaks down into work-efficient reverse scan of bicyclic semigroup; create bitmaps of size k to identify non-empty segments; create linked list data structure (scan of max operation) of non-empty segments; finally generate output, where each step either consumes one element from a segment, steps to the next bit in the bitmap, or follows the linked list (all O(1)). Result is the same stack slice as before.
- Build binary tree of bicyclic semigroup. This is the same as before, except that we start by a sequential reduction of the monoid for k elements. Also build bitmap representing stack monoid reduction (k bits per thread).
- Do *two* searches of tree. One for the first element in the k-chunk, the second for the first unclosed open parenthesis in the k-chunk. The second induces a linked list data structure to reconstruct all stack snapshots at k granularity.
- Sequentially walk input elements. Start with search based on the first element of the chunk. Maintain a local stack. For each element, push and pop local stack as in sequential algorithm. When local stack is empty, use hierarchy to resolve: next bit in bitmap, follow linked list induced by second search, and, when it steps out of the partition, resolve in prefix stack.

7 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 DX12.

8 Performance results

This section will need to be filled in with detailed performance results, especially graphs of throughput over problem size.

Preliminary numbers: For the $k = 1$ case, with $w = 512$ so the problem size is limited to 2^{18}, we see 3.9G elements/s on AMD 5700 XT. For $k = 8$, on same problem size, throughput increases to 8.5G for that problem size, and 12.9G as the problem size reaches the maximum of 2^{24}. This latter number is approximately 40% of the "speed of light" on that hardware, meaning the amount of time it takes to read the input twice and write the output.

9 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 rather than CREW. 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.

The best presentation of a work-efficient algorithm is [PDC94].

This algorithm is similar to generalized prefix sum, described in [Ble90].

References

[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, dec 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.

[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.