A Work-Efficient Step-Efficient Prefix-Sum Algorithm

Shubhabrata Sengupta[*] Aaron E. Lefohn[†] John D. Owens[‡]

1 Introduction

The Prefix-sum algorithm [Hillis and Steele Jr. 1986] is one of the most important building blocks for data-parallel computation. Its applications include parallel implementations of deleting marked elements from an array (stream-compaction), radix-sort, solving recurrence equations, solving tri-diagonal linear systems, and quicksort. In addition to being a useful building block, the prefix-sum algorithm is a good example of a computation that seems inherently sequential, but for which there are efficient data-parallel algorithms.

2 Problem definition

The prefix-sum algorithm computes all the partial sums of an array of numbers. It is called prefix-sum because it computes sums over all prefixes of the array. For example, if one is to put into an array one's initial checkbook balance, followed by the amounts of the check one has written as negative numbers and deposits as positive numbers, then computing the parital sums produces all the intermediate and final balances.

2.1 Sequential implementation

It might seem that computing the partial sums is an inherently serial process, because one must add up the first k elements before adding in the element $k+1$. Indeed, with only a single processor, one might as well do it that way. Algorithm 1 shows the pseudo-code for such an implementation.

```
$s \leftarrow x[0]$
for $i \leftarrow 1$ to $n-1$ do
    $s \leftarrow s + x[i]$
    $x[i] \leftarrow s$
```

Algorithm 1: Sequential algorithm that computes the prefix-sum of an array x containing n elements.

However, with many processors at our disposal, one can do better. Assuming one has n processors, one can do $n \log n$ individual additions in $\log n$ time. Serialization is avoided by performing logically redundant additions.

2.2 Horn's data-parallel algorithm

Daniel Horn recently presented a GPU implementation for prefix-sum [Horn 2005]. The pseudo-code for his implementation is shown in Algorithm 2.

The input array is stored in a two-dimensional texture. Each fragment uses its screen-space position to index into this texture. It sums the value at its position and 2^{i-1} position to the left. This is written to a separate output texture which is used as input to the next pass. By having only two textures and switching their roles in each pass, we minimize the memory footprint.

```
for $i \leftarrow 1$ to $\log_2 n$ do
    for all $k$ in parallel do
        if $k \geq 2^i$ then
            $x[k] \leftarrow x[k-2^i] + x[k]$
        else
            $x[k] \leftarrow x[k]$
```

Algorithm 2: Horn's algorithm to compute the prefix-sum of an array x containing n elements.

We notice that in any iteration i, only $n/2^i$ fragments are doing useful work. However, Horn's algorithm does more computation in one pass, enabling it to finish the prefix-sum computation in $O(\log n)$ time, compared to $O(n)$ time taken by the sequential algorithm to compute the same result.

We also note that the $O(\log n)$ computation time is true only when there are n or more processors which can compute in parallel. However, the fragment pipeline can only execute a fixed maximum number of fragments in parallel. We call this limit the degree of parallelism. If we begin with an array which is greater than this limit, the fragment pipeline would have to break up the fragments into batches, which it executes sequentially. However, it would be transparent to the programmer since logically the fragment pipeline operates on all fragments in parallel. Thus we expect to see a decrease in execution time as we decrease the size of the array, until we reach a size which is less than or equal to the degree of parallelism.

2.3 A work-efficient data-parallel algorithm

To reduce wasteful computation in each pass, we break the algorithm [Blelloch 1990] into two stages. The first stage is called *reduce* and the second stage is called *down-sweep*.

2.3.1 Reduce

The *reduce* step reduces an input array to a single element, which is the sum of all the elements in the input array. At each stage, we reduce the array size by half by adding together non-overlapping, adjacent pairs of elements. Algorithm 3 shows the pseudo-code.

```
for $d \leftarrow 1$ to $\log_2 n$ do
    for $i \leftarrow 1$ to $n/2^d - 1$ in parallel do
        $a_d[i] \leftarrow a_{d-1}[2i] + a_{d-1}[2i+1]$
```

Algorithm 3: The *reduce* step for an array containing n elements.

Thus each fragment doubles the value of its position to index into the two-dimensional texture which holds the input array, and adds that value to the one to the left of it. We halve the size of the output texture in each pass by halving the height of the texture, since that results in an efficient address translation from the one-dimensional array index to the two-dimensional index into the texture. Due to this optimization, we never run the *reduce* step until we get an array

[*] e-mail: ssengupta@ucdavis.edu

[†] e-mail: lefohn@cs.ucdavis.edu

[‡] e-mail: jowens@ece.ucdavis.edu

containing a single element. We can still continue with the *down-sweep* step by taking a hybrid approach, as we will explain in Section 2.4.

We save all the intermediate textures a_d containing the partial sums, since they will be required in the *down-sweep* step.

2.3.2 Down-sweep

The *down-sweep* step uses the partial sums a_d generated by the *reduce* step to generate the final output. Algorithm 4 shows the pseudo-code.

```
for $d \leftarrow (\log_2 n) - 1$ downto $0$ do
    for $i \leftarrow 0$ to $n/2^d - 1$ in parallel do
        if $i > 0$ then
            if $(i \bmod 2) \neq 0$ then
                $a_d[i] \leftarrow a_{d+1}[i/2]$
            else
                $a_d[i] \leftarrow a_d[i] + a_{d+1}[(i/2) - 1]$
        else
            $a_d[i] \leftarrow a_d[i]$
```

Algorithm 4: The *down-sweep* step for an array containing n elements.

In contrast to Horn's algorithm, our method does not do wasteful computation in each pass. The size of the output texture is halved in each pass, which means there are fewer fragments to process. However, the number of passes are doubled. In spite of the increase in the number of passes, this algorithm would do less computation for large arrays. It can be easily shown that Horn's algorithm does $n \log n - 2n(1 - 2^n)$ more operations than our algorithm, where n is the size of the input array. Since the difference increases monotonically with n, our algorithm would be faster when the degree of parallelism is fixed.

2.4 A work-efficient, step-efficient hybrid algorithm

For arrays which have fewer elements than the degree of parallelism offered by the graphics processor, the algorithm described in Section 2.3 is slower since it does not utilize the parallelism in each pass and executes twice the number of passes as Horn's algorithm. Hence we use a hybrid algorithm which combines the best of both worlds. We execute the *reduce* step until the size of the array is less than or equal to the degree of parallelism. We then run Horn's algorithm to compute the prefix-sum of this smaller array. Finally, we run the *down-sweep* step to update the partial sums generated by the *reduce* step. The hybrid algorithm would be the fastest of the three, since it does not do wasteful computation for large arrays and it does not do wasteful passes when the array size is smaller than the degree of parallelism offered by the graphics processor.

3 Implementation

We implemented all three algorithms using Cg and OpenGL on a NVIDIA 7800 GTX graphics card. Since prefix-sum is a frequently used operation data-parallel computation, the code is highly optimized. We expect that a typical user would use the prefix-sum algorithm multiple times in his application. Hence Cg programs are statically allocated, so that they only need to be compiled once and reused with minimal overhead.

Both the *reduce* and *down-sweep* steps need to keep the partial sums around as textures. We have tried to minimize texture memory usage by reusing textures wherever possible. We also statically allocate textures to minimize reuse overhead. The hybrid algorithm has the same memory footprint as the Horn's stream-compaction algorithm.

Our implementation also aggressively uses framebuffer objects to write out the result of each pass to a texture. As per the framebuffer object specification, all the textures bound to a framebuffer object need to be of the same size. Thus the output texture for each *reduce* pass is bound to a different framebuffer object which implies that we have to switch framebuffer objects for each *reduce* pass. Happily enough, this hasn't proved to be a bottleneck and we expect the performance of framebuffer objects to improve as drivers mature.

4 Results

We implement Horn's stream-compaction algorithm using our hybrid prefix-sum algorithm, since we expect stream-compaction to be the most frequent use of prefix-sum. For a given array size, this adds a constant cost to the prefix-sum algorithm.

We notice that the time taken to run stream-compaction falls as we increase the number of *reduce* passes and it levels off at around 9 *reduce* steps. For 1,048,576 input elements, the hybrid algorithm is around 4 times faster (10 *reduce* steps) and for 262,144 elements, the hybrid algorithm is around 3 times faster (9 *reduce* steps). In neither case are we reading back the final result to the CPU. We expect the performance to drop as the number of valid (non-null) elements increase since the readbacks are slow, though large readbacks would affect both algorithms to the same degree.

We run Horn's stream-compaction algorithm on streams of increasing length and we notice a sharp rise in execution time when the input stream contains more than 4096 elements indicating that it is not beneficial to do wasteful computation on the GPU if the input stream is larger than 4096 elements.

5 Conclusion

We have implemented a hybrid prefix-scan algorithm which is faster than Horn's prefix-scan algorithm by a factor of 4 for array sizes up to 1,048,576 elements. For a fixed number of elements in the input array, the algorithm can be tuned to run the optimum number of *reduce* passes.

References

BLELLOCH, G. E. 1990. Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov.

HILLIS, W. D., AND STEELE JR., G. L. 1986. Data parallel algorithms. *Communications of the ACM 29*, 12 (Dec.), 1170–1183.

HORN, D. 2005. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Mar., ch. 36, 573–589.