Efficient Stream Reduction on the GPU

David Roger
Grenoble University
Email: droger@inrialpes.fr

Ulf Assarsson
Chalmers University of Technology
Email: uffe@chalmers.se

Nicolas Holzschuch
Cornell University
Email: holzschu@cs.cornell.edu

***Abstract*—Stream reduction is the process of removing unwanted elements from a stream of outputs. It is a key component of many GPGPU algorithms, especially in multi-pass algorithms: the stream reduction is used to remove unwanted elements from the output of a previous pass before sending it as input for the next pass.**

In this paper, we present a new efficient algorithm for stream reduction on the GPU. Our algorithm works by splitting the input stream into smaller components of a fixed size, on which we run a standard stream reduction pass. We then concatenate the results of these stream reduction pass with line drawing. This last pass is very efficient because we already know the size of the data.

I. Introduction

Stream reduction [1], sometimes called stream compaction [2] is the process of removing unwanted elements from a data stream. It is especially useful for multi-pass GPGU algorithms, where the stream of data output from a pass is used as input in the next pass: in the first pass, the algorithm finds that some data is not required, and flags them for deletion. Since fragment shaders lack the ability to write data at specific places in memory (*scatter*), these elements are still present in the output stream, and the stream reduction pass must remove them before the next pass.

The stream reduction pass, while being essential in many GPGPU algorithms, is a difficult problem, because it implies writing data at specific locations. Given the specificity of the GPU architecture, it is often more efficient to do a stream reduction pass on the entire data set after a processing pass than to try to do scattered writing on the fly during the processing pass. The consequence is that the stream reduction pass must work efficiently on very large datasets.

Because of the size of the datasets and the difficulty to specify the address for writing memory access, stream reduction passes are the bottleneck in many GPGPU algorithms [1], [3].

In this paper, we present a new approach to stream reduction. Our algorithm is hierarchical, and splits the input stream into smaller components of a fixed size, s. A stream reduction pass is run on each of the components, and the results of each pass are concatenated using line drawing. The advantages of our hierarchical approach are numerous: stream reduction algorithms are more efficient on smaller data sets, and the final pass of concatenation has a very low cost, because we already know the size of the data sets.

In our study, we have found that the theoretical complexity of our hierarchical stream reduction algorithm is $O(n + n \log s)$, where s is a constant, compared to $O(n \log n)$ for previous work. Experimental studies confirm that our stream reduction algorithm is substantially faster than existing algorithms.

```
1: $i \leftarrow 0$
2: for $j = 0$ to $n - 1$ do
3:     if $x[j]$ must be kept then
4:         $x[i] \leftarrow x[j]$
5:         $i \leftarrow i + 1$
```

Fig. 1. On sequential processors, the stream reduction of an array x of n elements is trivially done with a single loop.

Recent GPUs (such as NVidia G80) offer the ability to easily remove elements from a stream of input data, through the geometry engine. Our experiments show that our stream reduction algorithm is also faster than this approach, especially for large datasets.

However, our algorithm is mostly destinated toward frameworks that lack scattering abilities, such as OpenGL. Other systems that are able to perform scatter operations, such as CUDA, can perform reduction through a more straightforward process, and would benefit less from our algorithm.

Our paper is organized as follows: in the next section we review previous work on stream reduction on the GPU. In section III, we then describe our algorithm in detail, and conduct a study of the theoretical complexity of our algorithm (section III-D). Section IV is a review of the different available prefix sum scan techniques, that we need as a subroutine. In section V, we study the experimental behavior of our algorithm as a function of input size and parameters, and compare it with state-of-the-art algorithms. In sectionVI, we present our conclusions and directions for future works.

II. Previous works

On sequential processors, stream reduction is trivially achieved through a single loop over the stream elements (see code in Figure 1). However, this sequential algorithm requires the ability to specify the position for writing memory access. For GPUs, stream reduction is a more complex task.

Although it is a fundamental element in many GPGPU applications, surprisingly little research has been published on stream reduction techniques. Horn [1] was the first to propose an efficient stream reduction algorithm for the GPU. He performs stream reduction in two steps (see Figure 2): first, he computes the offset (displacement) for each non-empty element in the stream, using a parallel prefix sum scan [4]

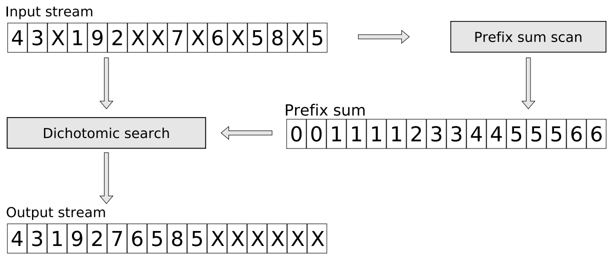

Fig. 2. Main steps of Horn's stream reduction illustrated with an example. The unwanted elements are the crosses. The prefix sum scan computes the displacement, and the dichotomic search performs it (*i.e.* moves each valid element to the left at a distance given by the prefix sum).

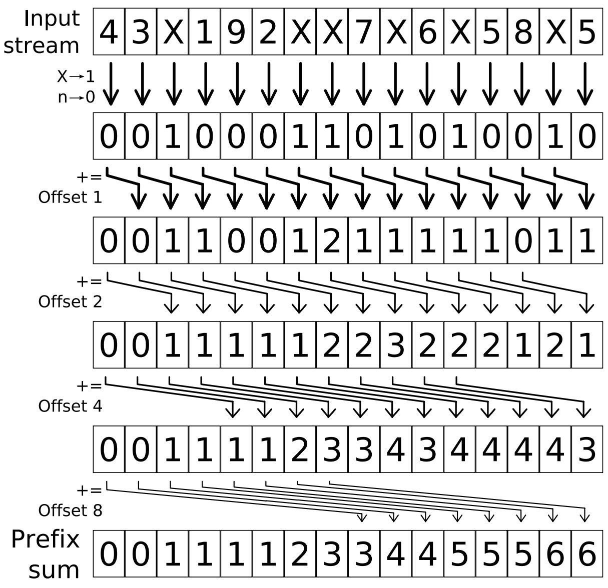

Fig. 3. The prefix sum scan used as the first part of Horn'algorithm has $\log n$ render passes. The first pass applies a boolean operation to each element of the input stream: if the element has to be discarded 1 is output, and 0 otherwise. Each subsequent pass performs the sum of two elements of the output of the previous pass. The prefix sum is the number of empty elements before each element, which is also the displacement that will be applied in the second part of the algorithm.

that counts the number of unwanted elements that are located before each element. Figure 3 details the different steps of the prefix sum scan. Second, each element is moved by the computed offset, using a logarithmic search. Both steps have $O(n \log n)$ complexity.

Sengupta *et al.* [2] improved this algorithm by using a different method for the prefix sum scan [5] of complexity $O(n)$.The second pass (moving each element by the computed offset) is left unchanged, so the overall complexity of their algorithm is still $O(n \log n)$.

Recent GPUs, such as NVidia's G80, support geometry shaders, which can trivially be used to remove unwanted elements from a stream of input. All the elements of the stream are stored as vertices in a vertex buffer object, which is given as input to a geometry program. This geometry program tests each element and discards the unwanted ones. The fragment program is skipped and the resulting

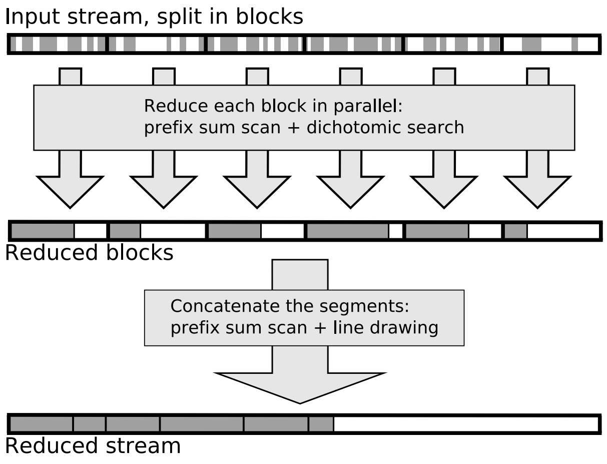
Input stream, split in blocks
Reduce each block in parallel:
prefix sum scan + dichotomic search
Reduced blocks
Concatenate the segments:
prefix sum scan + line drawing
Reduced stream

Fig. 4. We add a hierarchical step to the standard stream reduction algorithm: the input stream is divided in blocks, then each blocks is reduced, and finally the results are concatenated.

stream is directly output as a vertex buffer object, using the `NV_transform_feedback` OpenGL extension.

Sorting algorithms can trivially be used to perform stream reduction. The fastest GPU sorting algorithms have been based on bitonic sort [6]–[8].

Our theoretical and experimental studies show that our stream reduction method is much faster than Horn [1] and Sengupta *et al.* [2]. It is also significantly faster than the naive approach using geometry shaders, especially on large streams. Finally, because they address a simpler problem, all dedicated stream reduction methods outperform sorting algorithms by an order of magnitude.

In CUDA, fast stream reduction is possible, by adding a scatter step to an efficient prefix sum scan implementation, such as Sengupta *et al.* [9] or Harris *et al.* [10]. However, it requires scatter abilities. We will show how our algorithm could benefit from scattering too.

III. Algorithm

In this section, we present our stream-reduction algorithm. We begin by a presentation of the algorithm (section III-A), then we discuss specific details for each pass, in sections III-B and III-C. The overall complexity of our algorithm is summarized in section III-D.

A. Overview

Our algorithm builds upon existing stream reduction algorithms, using a divide-and-conquer approach. First, we divide the input stream into smaller blocks of size s, and we perform a stream reduction pass on each of these blocks. Then, we concatenate the results of the stream reduction pass, using line drawing (see Figure 4).

This hierarchical approach reduce the total number of operation and yields to a better asymptotic complexity. The blocks can be concatenated by openGL line drawing, and thus the algorithm is efficient and easy to implement.

B. Stream-reduction pass on each block

Once we have split the input stream into blocks of constant size s, we run a stream reduction pass on each of these blocks. We use recent stream reduction algorithms, running on the GPU [1], [2], in two passes: a prefix sum scan followed by a dichotomic search.

Here are the notations used in this section:

- x is one of the blocks.
- s is the size of the block.
- $pSum$ is the prefix sum, an array of size s.

We will now detail the two passes.

1) Prefix sum scan: First, we run a parallel prefix sum scan (*e.g* [4], [5]) on each block. For every element in the block, this computes the number of empty elements before it, which is also the length of the displacement to apply to this element. The prefix sum is stored in $pSum$. This step has a complexity of $O(s)$ using [5].

2) Dichotomic search: Each element is moved by the computed offset. As GPUs lack *scattering* abilities from the fast fragment shaders, this is achieved through *gathering*: for each position i of the output (*i.e. the reduced stream*), a dichotomic search is performed in order to find the element $x[j_i]$ of the input that has to be moved to i. This element is the only valid element of the input stream that satisfies $j_i - pSum[j_i] = i$.

As $j - pSum[j]$ is monotonously increasing with j, it is possible to find j_i by dichotomic search. The starting bounds of the research interval are given by:

$$i + pSum[i] \leq j_i \leq i + pSum[s-1]$$

On older GPUs, all the loops are constrained to have the same length, thus the algorithmic complexity for each element is $\log s$ (complexity of the worst case), leading to $s \log s$ for the whole block.

With recent GPUs such as the GeForce 8800, it is possible to slightly improve this algorithm. First, the fixed length loop (*for*) can be replaced by a conditional loop (*while*), as branching is more efficient. Second, the bounds of the research interval can be refined at each step of the loop in order to reduce the number of iterations. $j - pSum[j]$ is contracting with respect to j:

$$|j - j_i| \geq |j - pSum[j] - i|$$

Thus, at each step, the research interval is cut in half at a position j, and then further tightened by $|j - pSum[j] - i|$. The pseudocode of these improvements is given in Figure 5.

```
1: $lowerBound \leftarrow i + pSum[i]$
2: $upperBound \leftarrow i + pSum[s-1]$
3: **if** $upperBound > s-1$ **then**
4:     There is no element at position $i$. Stop here.
5: $j \leftarrow (lowerBound + upperBound)/2$
6: $found \leftarrow j - pSum[j] - i$
7: **while** ($found \neq 0$ **or** $x[j]$ is unwanted) **do**
8:     **if** $found < 0$ **then**
9:         $lowerBound \leftarrow j - found$
10:    **else**
11:        $upperBound \leftarrow j - \max(1, found)$
12:    $j \leftarrow (lowerBound + upperBound)/2$
13:    $found \leftarrow j - pSum[j] - i$
14: **return** $x[j]$
```

Fig. 5. Improved dichotomic search at the i^{th} position in a block x of size s: the program returns $x[j_i]$ such as $j_i - pSum[j_i] = i$. A conditionnal loop is used (line 7), and the bounds of the research interval are improved at each step of the loop (lines 9 and 11). The runing sum, previously computed, is stored in the $pSum$ array.

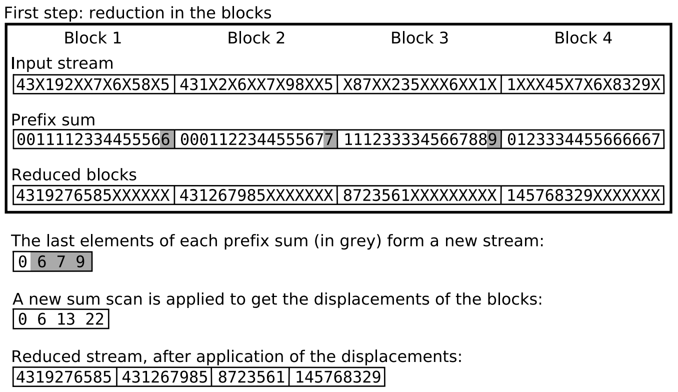

Fig. 6. In this example the block size s is 16. After the first part of the algorithm (reduction in the blocks), the last terms of the partial prefix sums, represented in grey color, are the number of empty elements in the blocks. These terms (except the last) form a stream of $\lceil n/s \rceil$ elements. By applying a new prefix sum scan, the displacement of each block is computed.

C. Concatenation of the intermediate results

After the reduction inside the blocks, we have $\lceil n/s \rceil$ blocks, each of them with at most s non-empty elements grouped at its beginning.

For each block, the number of empty elements is known, as a side result of the first pass of stream compaction: it is the last element of the prefix sum. Those $\lceil n/s \rceil$ last elements (one per block) form a sub-stream of the running sum. By running a second pass of sum scan on this sub-stream, we get a new stream describing the number of empty elements before each block, which is also the length of the displacement that has to be applied to the blocks, as seen on Figure 6. The complexity of this process is $O(\frac{n}{s})$ using [5].

Each block is then moved: the positions of the two ends of each segments are computed with the vertex shader using *scatter*, and the positions of the intermediate elements are linearly interpolated by rasterization. The vertex shader processes $2\lceil n/s \rceil$ elements, and the fragment shader processes all the elements. Thus the number of operations required decreases with block size: using too small blocks stresses the vertex engine and implies more memory transfers due to the large number of segments. The complexity of these displacements is $O(n)$.

As we store the streams in 2D textures, we have to convert the 1D stream into a 2D array, and thus we have to perform the wrapping of the segments, as illustrated on Figure 7. This

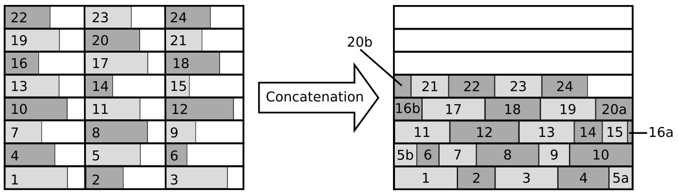

Fig. 7. Since we store the stream in a 2D texture, we have to wrap the line segments as we concatenate them. Segments 5, 16 and 20 have to be split, either by drawing them two times or in the geometry engine (if available).

can be done by sending all the segments twice: the first time draws the left part (or the full segment if it does not need wrapping), and the second draws the right part if needed or discards it otherwise. As most of the segments do not need such wrapping, it is better to use the geometry engine (if available) to split only when necessary.

D. Overall complexity

A summary of the algorithmic complexities of the different steps is given here:

1) Reduction of the blocks
   For one block:
   - Prefix sum scan using [5]: $O(s)$
   - dichotomic search in the block: $O(s \log s)$

   Total for the $\frac{n}{s}$ blocks: $O(n + n \log s)$
2) Concatenation of the line segments
   - Prefix sum scan using [5]: $O(\frac{n}{s})$
   - Line drawing: $O(n)$

The asymptotic complexity of our algorithm is $O(n + n \log s)$ (where s is a constant), and is to be compared with the $O(n \log n)$ complexity of the previous methods.

IV. Prefix sum scan techniques review

Our algorithm performs prefix sums computations at two different scales: inside the blocks and then across the blocks. Any technique can be used to compute those prefix sums.

We implemented both Hillis and Steele [4] and Blelloch [5]. In our experiments, despite its better asymptotic complexity, [5] has a higher overhead. [4] yielded better results for streams of 1 M elements and smaller, whereas [5] is slightly faster for 4 M and 16 M elements. However, the relative performance difference was less than 10%.

Sengupta *et al.* hybrid algorithm [2], which is a combination of Blelloch [5] and Hillis and Steele [4] method, could also be used for theoritically even faster results, although we have not tested this. However we would not expect a spectacular speed up, as we keep working on relatively small streams ($s = 64$) whereas [2] is tailored for large ones.

V. Results

A. Behavior

All the results of these section have been measured using a GeForce 8800 GTS, and an Intel Pentium IV 3 GHz with 2 Gb RAM. The algorithm was implemented using openGL.

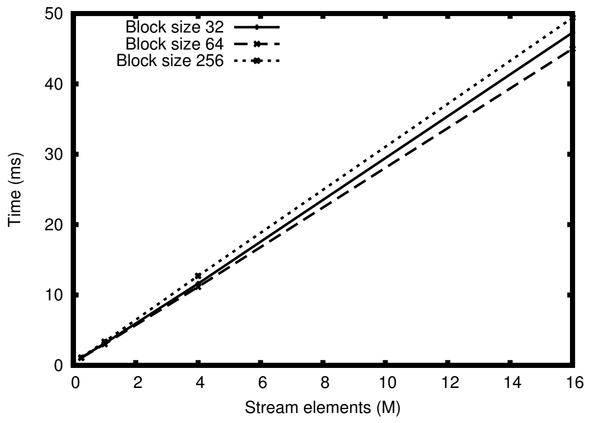

Fig. 8. Experimentally, our algorithm has a linear complexity with a slope depending on block size.

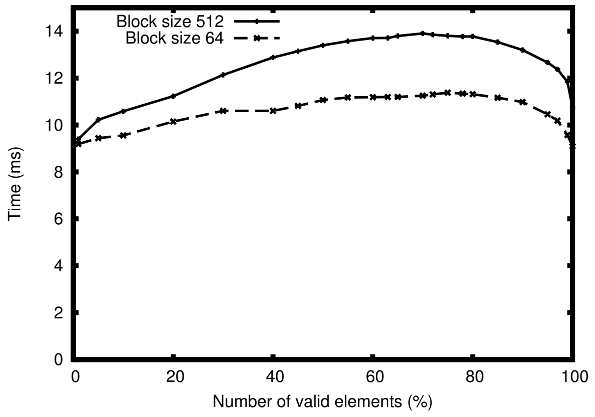

Fig. 9. Behavior of the algorithm with respect to the ratio of valid elements in a stream of size 4 M. The maximum execution time is reached at 70%.

As shown in section III-D, the asymptotic complexity of our algorithm is linear with respect to the number of elements in the stream. This is confirmed by our experiments (Figure 8).

The execution time also depends on the ratio of valid elements: it is faster when either most of the elements are valid or most are unwanted. The maximum time is reached around 70% of valid elements (Figure 9). However these variations are relatively small. If not stated otherwise, all the timings in this document are given for a 60% ratio of valid elements, which is a difficult case for our algorithm.

The size of the blocks is an obvious parameter for our algorithm. Small blocks are reduced faster (complexity $O(n + n \log s)$) but are concatened slower because of the complexity $O(n + \frac{n}{s})$ and the stress on the vertex engine. On the other hand, large blocks are reduced slower but concatened faster. Despite the lesser efficiency of the vertex engine, line drawing is not a bottleneck of the algorithm, except for very small block sizes (8 or less). Figure 10 shows the time repartition between the different steps of the algorithm for several block sizes, and Figure 11 shows the influence of the block size. We found experimentally that the best block size is 64 for streams

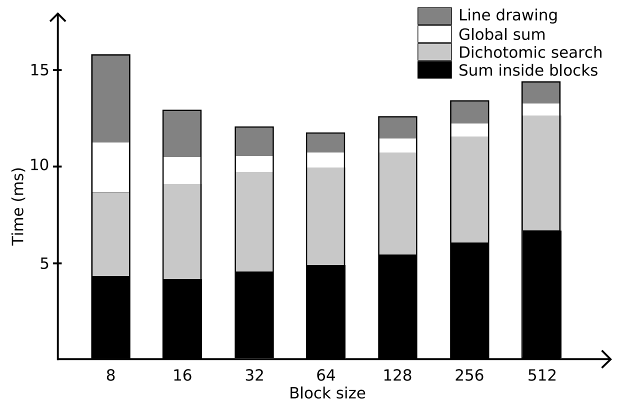

Fig. 10. Time spent in the different steps of the algorithm, for 4 M elements and various block sizes. Time spent in line drawing and global sum decreases with block size, whereas dichotomic search time and sum inside blocks time increases.

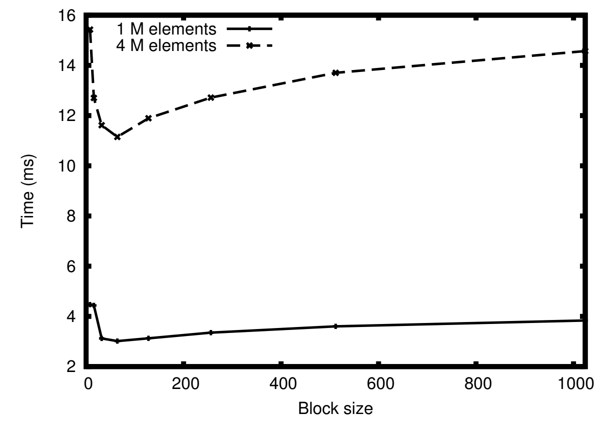

Fig. 11. Behavior of the algorithm with respect to block size. The best block size was 64 in all our experiments.

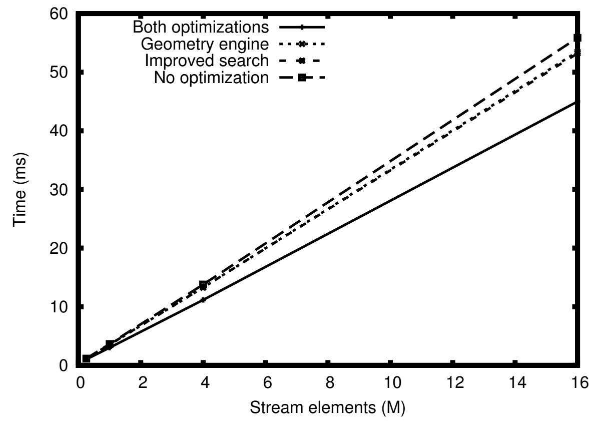

Fig. 12. Effect of two optimizations requiring a recent GPU such as the GeForce 8800: single line drawing and improved dichotomic search.

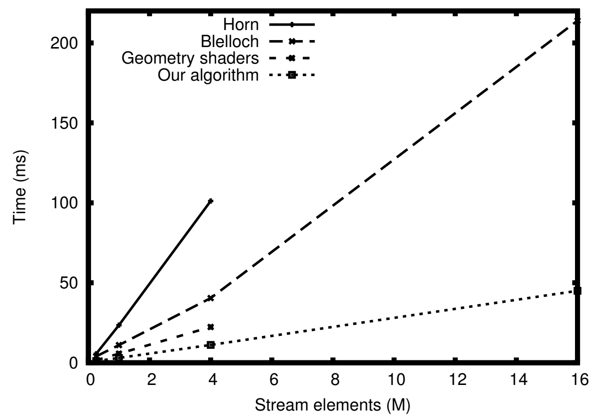

Fig. 13. Comparison of our algorithm against three other stream-reduction methods.

up to 16 M elements.

GeForce 8800 specific functionalities allows the two optimizations mentioned previously: single line drawing (section III-C) and improved dichotomic search (section III-B2). These optimizations save approximately 20% of execution time, as shown in Figure 12.

B. Comparison with other stream-reduction methods

We implemented the standard Horn's algorithm [1], and a modified version that uses Blelloch's prefix sum scan [5].

We also compared our algorithm to straightforward packing using GeForce 8800 advanced functionalities: elements are sent as vertices in a vertex buffer object, and the geometry shader discards the unwanted ones.

The comparison of these methods with ours is summarized in Figure 13 and Table I. Our algorithm is 9 times faster than Horn's on 4M elements stream and 2 times faster than using geometry shaders. Notice how the speed-up ratio increases with the stream size.

In CUDA, scatter abilities are available. Sengupta *et al.* [9] and Harris *et al.* [10] presented an optimized parallel prefix sum scan-algorithm, based on [5]. Thus stream reduction can be efficiently implemented in this framework by adding a fast scattering step to [9] or [10]. We have reported the timings of [10] in Table I: our algorithm is $\approx$2.5 X slower, however this CUDA algorithm requires scatter abilities whereas ours does not.

We did not implement Sengupta *et al.* [2], which achieves a 4 times speed up compared to Horn on streams of 1 M elements. As shown in Table I, our speed up is 7.8.

The largest stream on which we could run our algorithm had 16 M elements. We encountered the same limit with Blelloch [5]. We could not make Horn [1] and Geometry shaders work on streams with more than 4 M elements. In most cases, the limit is related to memory issues: such very large streams are taking a significant portion of the memory available on the card.

VI. Conclusion

In this paper, we have presented a new algorithm for stream reduction, which is an essential step of many GPGPU applications [1], [3].

#elements	Horn	Blelloch	G. Shaders	CUDA (sum only)
256k	5.5	3.9	1.5	0.32
1M	7.8	3.7	1.9	0.37
4M	9.1	3.6	2	0.36
16M	N.A.	4.8	N.A.	0.35

TABLE I
SPEED UP OF OUR ALGORITHM COMPARED TO OTHER STREAM-REDUCTION METHODS (*i.e.* RATIO OF THE EXECUTION TIMES). OUR ALGORITHM IS $\approx$2.5 X SLOWER THAN CUDA, WHICH USES SCATTER ABILITIES WHEREAS WE DO NOT.

Our algorithm works by a hierarchical approach: we divide the input stream into smaller blocks, perform a fast stream reduction pass on these smaller blocks, and concatenate the results. We thus achieve a new order of asymptotic complexity ($O(n)$ whereas previous methods where $O(n \log n)$) and an impressive speed up (Table I). Our algorithm even outperforms doing stream reduction using the geometry shaders.

As our algorithm can use any method to perform the prefix sum, it can benefit from research in this domain.

Furthermore, our work can be seen as a meta-algorithm that improves any stream reduction algorithm by adding a hierarchical step to it.

Our divide-and-conquer approach results in impressive speed-ups for stream-reduction. In the future, we expect that this approach can be used for improving other GPGPU algorithms.

As a future work, we would like to implement our algorithm in CUDA. This would allow to speed up several steps: the reduction of the blocks could be computed sequentially using the algorithm Figure 1 (instead of a sum scan and a dichotomic search), line drawing and the line wrapping would disappear. The algorithmic complexity would be then $O(n)$, and could be comprared more accurately to existing methods in CUDA.

REFERENCES

[1] D. Horn, "Stream reduction operations for GPGPU applications," in *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, 2005, ch. 36, pp. 573–589.

[2] S. Sengupta, A. E. Lefohn, and J. D. Owens, "A work-efficient step-efficient prefix sum algorithm," in *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, May 2006, pp. D–26–27.

[3] D. Roger, U. Assarson, and N. Holzschuch, "Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the GPU," in *Rendering Techniques 2007 (Eurographics Symposium on Rendering)*, jun 2007, pp. 99–110.

[4] W. D. Hillis and J. Guy L. Steele, "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[5] G. E. Blelloch, "Prefix sums and their applications," in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan Kaufmann, 1993. [Online]. Available: citeseer.ist.psu.edu/blelloch90prefix.html

[6] N. K. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha, "A cache-efficient sorting algorithm for database and data mining computations using graphics processors," University of North Carolina, Tech. Rep., 2005.

[7] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High performance graphics coprocessor sorting for large database management," in *ACM SIGMOD 2006*, 2006.

[8] A. Greß and G. Zachmann, "GPU-ABiSort: Optimal parallel sorting on stream architectures," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 25–29 April 2006.

[9] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware 2007*. ACM, Aug. 2007, pp. 97–106.

[10] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2007, ch. 39, pp. 851–876.