Algorithmic Strategies for Optimizing the Parallel Reduction Primitive in CUDA

Pedro J. Martín, Luis F. Ayuso, Roberto Torres, Antonio Gavilanes
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain
{pjmartin@sip, lf.ayuso@fdi, r.torres@fdi, agav@sip}.ucm.es

Abstract—**Many general-purpose applications exploit Graphics Processing Units (GPUs) by executing a set of well-known data-parallel primitives. Those primitives are usually invoked from the host many times, so their throughput has a great impact on the performance of the overall system. Thus, the study of novel algorithmic strategies to optimize their implementation on current devices is an interesting topic to the GPU community. In this paper we focus on optimizing the reduction primitive, which merely reduces a data sequence into a single value using a binary associative operator. Although tree-based and sequential-based algorithms have been already implemented on GPUs, a comparison of both algorithm performance had not been carried out yet. Thus, our first contribution is to present an experimental study of state-of-the-art reduction algorithms on CUDA. Next we introduce two algorithmic optimizations that are integrated into the fastest solution (a sequential-based algorithm), improving its throughput even more. Finally, we replicate this methodology to the segmented version of the primitive, which applies when the input is composed of several independent segments. In this case, it is not clear which algorithm exhibits the best performance, since throughput deeply depends on the distribution of segments along the input. According to our results, tree-based algorithms run faster for small segments, while sequential methods are better for medium and large ones.**

Keywords- parallel reduction; segmented parallel reduction; data-parallel algorithms; GPGPU; CUDA.

I. INTRODUCTION

Nowadays Graphics Processing Units (GPUs) are used to accelerate a wide range of general-purpose applications by exploiting their high-performance many-core processors. GPUs usually contribute to the overall computation in two different ways: carrying out some specific tasks through user-designed kernels or executing some data-parallel primitives provided by a growing number of libraries (e.g. CUDPP, CLPP, GPULib, Thrust…). While programmers assume control of the throughput of their kernels, primitives are integrated as black boxes whose performance relies on the library. On the other hand, hardware improvements incorporate more functional units in each release, along with larger shared memories and sophisticated cache hierarchies. For those reasons, the study of algorithmic strategies to optimize the implementation of these primitives, and the adjustment to the features of the upcoming GPU architectures, are ever-interesting topics to the General-Purpose GPU community [8].

In this paper, we focus on optimizing the classic *reduction primitive*, paying attention to its two versions. Its *unsegmented* form takes a binary associative operator $\oplus$ (e.g. $+$, $\times$, min and max) and an array of N data [a_0, a_1, ..., a_{N-1}] as inputs, and it returns as output one value ($a_0 \oplus a_1 \oplus ... \oplus a_{N-1}$). In its *segmented* version, the input array is divided into segments of consecutive data, and the output is the individual reduction of each segment. Thus, the output size is the number of segments included in the given input. Observe that the segmented primitive could be easily implemented in terms of the unsegmented primitive, by extracting each segment and reducing it, isolated from the global input, with the unsegmented primitive. Nevertheless, this should be done many times (one for each segment), and some of the segments may be too small to justify further kernel executions. In consequence, this approach would not take the most of GPUs. On the contrary, the segmented solutions we illustrate will simultaneously perform separate parallel reductions on the segments of the input. For this reason unsegmented and segmented reductions are introduced as independent primitives along the paper.

The two reduction versions are useful building blocks for solving a wide variety of problems on GPU. For example and using CUDA, the unsegmented version has been successfully applied to solve the Single-Source Shortest-Path problem [12] and to build the Minimum Spanning Tree [16], while the segmented version to construct kd-trees on GPU for ray tracing [17] and to accelerate sparse-matrix multiplication [3].

Concerning the properties the operator $\oplus$ must fulfill, only associativity is essential. Actually, the correctness of all the algorithms described along this paper deeply relies on it. Most of the presented algorithms also make use of identity; so, $1_\oplus$ will denote an identity element for $\oplus$ from now on. Although the operator could be commutative as well, as it happens to the examples above, we will not suppose it in this paper, i.e. they are considered to be "non-commutative". The advantage of using commutativity, along with associativity, is that any pair of elements could be reduced regardless of their location on the input. However, these pairs must belong to the same segment in segmented reduction, which makes it difficult to exploit commutativity in this case. Hence, the exploitation of commutativity seems to come into conflict with the segment arrangement.

N	Number of elements in the input array
D	Size of a data-block
N_D	Number of data-blocks inside the input ($= ceil(N/D)$)
B	Number of threads in a CUDA-block
G	Number of CUDA-blocks in a grid
R	#Elements a thread loads in tree-based reductions ($= D/B$)
W	Width of the matrix in sequential matrix-reductions
H	Height of the matrix in sequential matrix-reductions
$MBPM$	Maximum number of resident blocks per multiprocessor
P	Number of producer warps in the producer-consumer scheme
C	Number of consumer warps in the producer-consumer scheme
$nBanks$	Number of banks in the shared memory of the device

Glossary. Parameters used in the paper.

One of the main aims of this paper is to compare two kinds of reduction approaches: recursive (tree-based) versus sequential algorithms. In fact, our first contribution is to experimentally confront the state-of-the-art algorithms of both types. Obviously, we have replicated the experimental study for the two versions of the primitive. As a second contribution, we propose two algorithmic optimizations that can be integrated into any of the previous algorithms, regardless their nature. We have tested them into the fastest solution, resulting in significant speed-up for unsegmented reduction. However, they did not lead to succeed in the segmented case since the resulting solutions are bounded by the shared memory size.

II. Related Work

The kernels presented by Harris [10] are the most popular CUDA implementations for the unsegmented reduction primitive. They are actually included as project examples in every CUDA SDK release. His document introduces seven kernels from a didactic perspective, in such a way that each kernel improves the performance of the previous one. Nevertheless, many of them require the operator to be commutative.

The two segmented reduction algorithms we have found are located at the previous references [17, 3], where the primitive is also applied to solve a specific problem. In both cases, those proposals are based on the works of Blelloch for the scan primitive below mentioned.

The reduction primitive is actually a part of the (*inclusive*) *scan* primitive, which has been studied more widely because of its great applicability [5]. Thus, we must describe in this section some of the progress made on the scan primitive. Scan also takes an array [a_0, a_1, ..., a_{N-1}] and a binary associative operator $\oplus$ as inputs, but it returns an array containing the reduction of all the prefixes [a_0, ($a_0 \oplus a_1$), ..., ($a_0 \oplus a_1 \oplus ... \oplus a_{N-1}$)]. Scan also accepts a *segmented* version. In this case, the output is the unsegmented scan of each segment.

The classic parallel algorithms for scan were studied by Blelloch [4, 5]. His formulations were based on recursive equations whose application described a full binary tree. For this reason, they are called *tree-based* algorithms. Later on, with the advent of GPUs, these formulations were adapted to the novel programming model. Thus, tabulation techniques were applied to replace the recursive nature of tree-based algorithms with an iterative processing. Horn [11] was the first developing GPU-based implementations of the scan primitive. The complexity $O(N\log N)$ of its formulation was improved by Gress et al. [9] and Sengupta et al. [13] to a linear algorithm. The latter presents a work-efficient step-efficient implementation on CUDA that was adapted to the segmented case a year later [14]. In order to improve performance of previous algorithm, the authors exploit shared memory usage. However, the implementations involve bank conflicts, and the kernels may not scale well with shared memory size.

Subsequently, Dotsenko et al. [7] presented work-efficient sequential algorithms for the unsegmented and segmented scan. They decompose the input into blocks that are arranged as matrices in shared memory. Each matrix is sequentially reduced by rows and partial results are stored in a smaller array, which is processed later on. The overhead of previous tree-based formulations concerning synchronization barriers is reduced since each thread reduces a row. Immediately, Sengupta et al. [15] improved their tree-based algorithms incorporating an intra-warp operation: each warp individually performs a scan over 32 elements. Next, one warp carries out another intra-warp execution over the previous results to generate the reduction of the whole block. The advantages of this operation are that many synchronization barriers become unnecessary. However that paper does not include a comparison with the work by Dotsenko et al, thus, the most recent and fastest implementations of both trends –the intra-warp scan [15] for the tree-based trend and the sequential scan [7] for the sequential family– have not been experimentally compared yet.

III. Unsegmented Reduction

Along the paper, we use the term block to denote two different concepts. A *CUDA-block* is a block of threads, while a *data-block* is a chunk of consecutive data that is individually reduced by a CUDA-block. As we will see later, a CUDA-block can reduce one or several data-blocks. We use B and D to denote the sizes of a CUDA-block and a data-block, respectively (see Glossary).

A data-block is loaded from global memory to shared memory by the corresponding CUDA-block, before being reduced. This is done exploiting coalesced readings. When the input size N is not a multiple of D, we append a virtual padding to the last data-block, which is filled with values $1_\oplus$. This is done through an *if*-statement during the loading stage. Thus, all the algorithms of this paper work on $N_D = ceil(N/D)$ data-blocks.

In order to reveal the main differences among the algorithms, we focus on how a data-block is reduced into a single result, and on how this value is handled afterwards. Thus, we will suppose that the array `s_data` holds a data-block in shared memory for subsequent reduction.

A. State-of-the-art Reductions

A common characteristic of these algorithms is that each CUDA-block exactly reduces a data-block. The result is then written back into global memory by a thread of the CUDA-block. Hence, the grid size is $G = N_D$, which also corresponds to the output size. In consequence, the output must be reduced again with another kernel launch, and so on, until a single result is left. This is usually known as a *multi-pass* approach.

1) Tree-based reductions: Supposing that `s_data` lays on the leaves of a full binary tree, tree-based algorithms reduce each pair of siblings using $\oplus$, in a bottom-up manner. Hence, they require D to be a power of two. The underlying recursive equations are:

$$\text{red}(n) = \begin{cases} \text{red}(\mathit{left}(n)) \oplus \text{red}(\mathit{right}(n)) & \mathit{if}\ n\ \mathit{is\ inner} \\ n & \mathit{if}\ n\ \mathit{is\ leaf} \end{cases} \tag{1}$$

Algorithm 1 is similar to the kernel#2 by Harris [10]. At each iteration, a thread reduces two elements at line 14 (with indices `ai` and `bi`), and stores the result in `ai`, which corresponds to a *left-storing* approach. Storing into `bi` would be also possible (*right-storing*). The first iteration requires a thread for each pair of elements, thus $D = 2B$ and B must be a power of two as well. So the algorithm executes the loop $log_2(2B)$ times. Also observe that in each iteration, the number of active threads is halved (line 7), reaching a single active thread in the last iteration.

In order to avoid bank conflicts in shared memory, Harris replaces this interleaved addressing access with a sequential addressing pattern in his kernel#3. This new approach requires $\oplus$ to be commutative, so this technique has not been considered in this paper. On the contrary, we overcome bank conflicts by including the usual padding of one element every *nBanks* elements, where *nBanks* is the number of banks in the shared memory of the device. Thus, the total padding is *2B/nBanks* elements. This is why we update the indices at lines 11 and 12. Incorporating this offset does not penalize occupancy on current devices, and the overhead due to the index arithmetic is negligible.

Tree-based algorithms are quite fast, but they suffer from too much synchronization. Notice that a barrier must be located between iterations (line 6). However, the number R of elements that a thread loads from global memory can be tuned to achieve a certain speed-up. Notice that $D = R * B$ then holds. We do not consider such improvements as algorithmic optimizations, but code optimizations, so we have not paid attention to them in this paper. Observe that R=2 in Algorithm 1. On the contrary, R=8 in the scan implementation of CUDPP 1.1.1 [6].

```
 1  void TB2_reduction (float* s_data){
 2    unsigned int thid = threadIdx.x;
 3    unsigned int stride = 1;
 4    unsigned int i, ai, bi;
 5    for(unsigned int d=B; d>0; d>>=1){
 6      __syncthreads();
 7      if(thid < d){
 8        i = 2*stride*thid;
 9        ai = i;
10        bi = ai + stride;
11        ai += (ai >> 5); //log2(nBanks)=5
12        bi += (bi >> 5); //log2(nBanks)=5
13        //Reduction
14        s_data[ai] = op(s_data[ai], s_data[bi]);
15      }//if
16      stride <<=1;
17    }//for
18    //The result is in s_data[0]
19  }
```

Algorithm 1. Tree-based reduction. R=2.

```
 1  void tb_reduceWarp( float* s_data,
 2                      unsigned int thid,
 3                      unsigned int lane,
 4                      float& target){
 5   //REDUCTION INTRA-WARP (Left-Storing)
 6   if( !(lane & 1)  ) s_data[thid]=
 7     op(s_data[thid], s_data[thid+1]); //%2=0
 8   if( !(lane & 3)  ) s_data[thid]=
 9     op(s_data[thid], s_data[thid+2]); //%4=0
10   if( !(lane & 7)  ) s_data[thid]=
11     op(s_data[thid], s_data[thid+4]); //%8=0
12   if( !(lane & 15) ) s_data[thid]=
13     op(s_data[thid], s_data[thid+8]); //%16=0
14   if( !(lane & 31) ) target=
15     op(s_data[thid], s_data[thid+16]);//%32=0
16  }
17  // **************************************
18  //D=1024, B=256 => Nwarps=8, 32 intermediate results
19  void TB4_Warp_reduction (float* s_data){
20   __shared__ float s_result[32];//32 results
21   unsigned int thid    = threadIdx.x;
22   unsigned int warpid = thid >> 5; //thid/32
23   unsigned int lane    = thid & 31; //thid%32
24
25   //Reduce s_data[kB+warpid*32, kB+(warpid+1)*32)
26   //and store the result into s_result[k*Nwarps+warpid
27   for(unsigned int k = 0; k<4; k++)
28     tb_reduceWarp(s_data, thid+k*blockDim.x, lane,
29                   s_result[warpid+k*Nwarps]);
30   __syncthreads();
31
32   if(warpid==0)
33     tb_reduceWarp(s_data, thid, lane, s_data[0]);
34   //The final result is in s_data[0]
35  }
```

Algorithm 2. Intra-warp tree-based reduction. B=2^8, D=2^{10}.

2) Intra-warp tree-based reductions: Sengupta et al. [15] improve the previous tree-based algorithm by working at the warp level. In order to explain this techique, let us extend the notation of CUDA-block and data-block to the case of warps. Briefly, a *CUDA-warp* is composed of 32 adjacent threads inside a CUDA-block, which run implicitely synchronized in SIMD fashion, while a *data-warp* is a chunk of 32 consecutive data inside a data-block. Sengupta et al. avoid bank conflicts and many synchronization barriers, using an intra-warp routine in which each data-warp is scanned by a CUDA-warp. The device function `tb_reduceWarp` of Algorithm 2 adapts this tecnique to the reduction primitive. During its execution, each CUDA-warp is responsible for reducing one data-warp, and sending the result to parameter `target`. The five iterations that are enough to reduce a data-warp have been unrolled, as the authors do. Notice that no explicit synchronization barriers are requiered due to the way CUDA-warps run in CUDA.

The kernel `TB4_Warp_reduction` of Algorithm 2 repeatedly invokes the previous function to reduce the whole data-block. It uses a shared array called `s_result` to hold the intermediate result each data-warp produces. We have used D=1024 and B=256, thus R=4, there are N_{warps}=8 CUDA-warps, which are responsible for reducing 4 data-warps, and the number of intermediate results is 32. Each data-warp is reduced at line 28, and the result is sent to `s_result[warpid+k*N`$_{\text{warps}}$`]`. Finally, intermediate results are reduced again at line 33 by the first CUDA-warp. The final result is sent to `s_data[0]` for the sake of clarity; in practice it is sent straight to global memory.

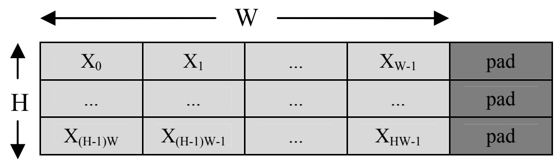


Figure 1. Arranging `s_data` as a matrix.

3) Sequential reductions: Dotsenko et al. [7] propose an algorithm for the scan primitive that is based on a matrix representation of `s_data`, as Fig. 1 shows. H and W respectively denote the height and width of this matrix, so $D = H \times W$. Algorithm 3 adapts their *MatrixScan* to the reduction primitive. Observe that one thread is responsible for sequentially reducing one row of W elements, thus only H threads are required to reduce the whole data-block (line 5). Nevertheless, all the threads inside the CUDA-block cooperated at the beginning to load the data, including these H threads.

Since reducing a row involves no synchronizations, the performance of the algorithm could be improved by maximizing W. Nevertheless, D must be small enough to fit in shared memory, thus W and H are forced to be rather small as well. In addition, H should be a multiple of the CUDA-warp size in order to avoid divergent warps. To sum up, Dotsenko et al. finally assign W and H to be the warp size ($W = H = 32$). A padding of one element is then added at the end of each row to avoid bank conflicts. Moreover, the H values that are obtained after reducing the H rows can be reduced using one intra-warp tree-based reduction (line 16). Also notice that only one warp continues beyond line 5 since $H = 32$, so no explicit synchronization is needed at line 15.

B. Algorithmic Optimizations

We present two algorithmic optimizations that can be integrated into any of the solutions described above.

1) Persistent blocks: We can force each CUDA-block to reduce multiple consecutive data-blocks, instead of a single one. Thus, the output size decreases since the number of CUDA-blocks (G) is smaller than the number of data-blocks. Moreover, the number of multipasses falls as G decreases. The

```
1   void Matrix_reduction (float* s_data){
2    unsigned int thid = threadIdx.x;
3    float current_red;
4    //Only the first threads reduce
5    if(thid < H){
6     unsigned int s_base = thid * (W + PADDING);
7     float* row = &s_data[s_base];
8     current_red = row[0];
9
10    for(unsigned int k=1; k<W; k++)
11       current_red = op(current_red, row[k]);
12
13    //Store the reduction of the row
14    s_data[thid] = current_red;
15
16    tb_reduceWarp(s_data, thid, thid&31, s_data[0]);
17   }//if
18  } //The final result is in s_data[0]
```

Algorithm 3. Sequential matrix-based reduction. H=W=32.

reductions of the data-blocks assigned to a CUDA-block are accumulated using a shared variable. Specifically, one of its threads accumulates the result of a data-block into the reduction of the previous data-blocks.

In order to distribute all the data-blocks among all the CUDA-blocks, we simply incorporate two new parameters into the kernel to indicate the quotient `q` and the remainder `r` of the division N_D/G. Then, a CUDA-block must reduce `q+1` data-blocks if `blockIdx.x<r`, or just `q` data-blocks otherwise.

In order to improve performance, the grid size must be carefully chosen according to the requirements of the kernel. Given a block size B, the maximum number of resident CUDA-blocks per multiprocessor (*MBPM*) is computed according to the CUDA Occupancy Calculator. Then G is fixed to *NUM_MULTIPROCESSORS*MBPM*. The underlying idea is to fill each multiprocessor with the maximum number of CUDA-blocks that can reside together on it. Thus, no CUDA-warp will wait for being allocated on a multiprocessor. We use the adjective *persistent* to denote these CUDA-blocks since they will be residing on the device until the whole input is processed.

Using persistent blocks results in a small grid size, since the number of multiprocessors is limited by the device, and *MBPM* cannot exceed 8 in any of the current CUDA compute capabilities. Thus, one kernel execution is almost enough to reduce the whole input. Indeed, the results after the first launch do not even complete a data-block because $G < D$. In consequence, we remove the usual recursion controlled by the host, and furthermore the cost of storing partial results into global memory between recursive calls.

The idea of persistent blocks has been already applied to other topics. For example, it was recently used to accelerate the traversal step of GPU-implemented ray tracers by Aila and Lane [1], under the term *persistent thread*. In fact, Harris [10] already proposed a reduction algorithm, the so-called *cascading* kernel#7, whose CUDA-blocks reduce many data-blocks, but during the load stage, rather than during the reduction stage, and using a commutative operator. Nevertheless, these authors do not explain how the grid size (G) is chosen in their respective papers.

2) Producer-consumer scheme: We add another optimization onto the persistent technique in order to help the schedulers to hide memory latency a little more. The idea is to classify the set of CUDA-warps into two groups: consumers and producers, of respective sizes C and P. At each iteration, consumer warps sequentially reduce the data-block loaded in the previous iteration, while producer warps load a new data-block. Thus, consumers can reduce at the same time producers load new data.

Algorithm 4 shows the code fragment that implements the producer-consumer scheme. Two data-blocks are held in shared memory, which are accessed through two pointers, `s_load` and `s_comp`. These pointers are swapped at the beginning of each iteration (line 10). Then, consumer warps reduce the data-block pointed by `s_comp` (line 13), while producer warps load the

```
1  //Load the first data-block
2  if(warpid>=C)
3    loadChunk(first, s_load);
4  __syncthreads();
5  first += D;
6
7  //Producer-consumer loop
8  for(unsigned int k=1; k<d; k++){
9    //swap shared buffer pointers
10   aux = s_load; s_load = s_comp; s_comp = aux;
11   //Each thread does its job
12   if(warpid<C)
13      reduceChunk(s_comp, s_current_red);
14   else
15      loadChunk(first, s_load);
16   __syncthreads();
17   first += D;
18 } //for
19
20 //Reduce the last data-block
21 if(warpid<C)
22   reduceChunk(s_load, s_current_red);
23 __syncthreads();
24 //The result is in s_current_red
```
Algorithm 4. Producer-consumer scheme.

data-block that is located at index `first` in global memory into the buffer pointed by `s_load` (line 15). Observe that a synchronization barrier is required (line 16) to prevent warps from overwriting the other buffer. Also notice that the first/last data-block is loaded/reduced before/after the loop. The number of iterations is controlled by variable `d`, which holds the number of data-blocks assigned to this CUDA-block. The result of reducing a data-block is accumulated into the shared variable `s_current_red` inside the `reduceChunk` routine.

Fig. 2 graphically exposes the advantages of this technique, using the sequential matrix-based solution presented in Algorithm 3 as the underlying reduction method. Remember that each data-block is arranged as a matrix of size $D = H \times W = 1024$, where $H = 32$ and $W = 32$ denote its height and width, respectively. The figure depicts how warps can be dispatched if the optimization is incorporated (on the left), and if is not (on the right). In the first case, the producer-consumer scheme has a configuration of $P = 8$ producers (warps from W_1 to W_8) versus $C = 1$ consumers (warp W_0). Observe that a single consumer warp is enough, since exactly 32 rows must be reduced. In addition, each producer is responsible for loading

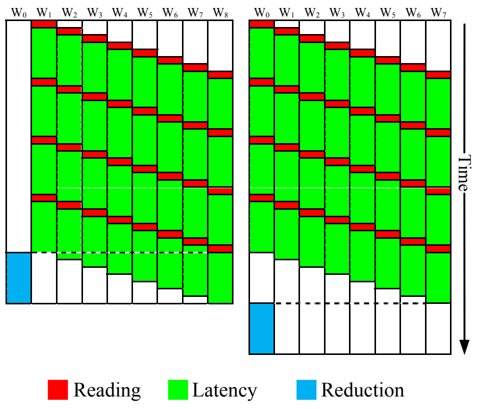


Figure 2. Producer-consumer scheme (left).
Persistent sequential matrix-based reduction (right).

$D/P = 128$ elements, which is done in $128/32 = 4$ coalesced readings. Whenever a producer has requested data from global memory, it must wait until those data are available. This waiting time can be used to request more data by another producer, or to reduce the previous data-block by the consumer.

On the right, the figure shows the execution of the persistent sequential matrix-based algorithm without the optimization. In this case we have 8 warps in a CUDA-block ($B = 256$) processing $D = 1024$ elements as before. Notice that warp W_0 starts reducing only when all the data-block are available in shared memory. Thus, each iteration is slightly longer than one using the producer-consumer scheme.

Although the producer-consumer paradigm is a well-known technique, its implementation on GPU is a recent issue. Besides our paper, Bauer et al. apply DMA techniques to solve in GPU other problems [2]. Our scheme can be compared to their "manual double-buffering" technique.

IV. Segmented Reduction

In segmented reduction (*s-reduction* in the sequel), the input is divided into *segments*. We demarcate them by using another array of size *N*, called `owner`, such that `owner[i]` holds the index of the segment of element `i`. Hence, `owner` is sorted in non-decreasing order. Notice that the output of the s-reduction is an array whose size is the number of segments. In the sequel, the variable `g_output` will denote such array, which is located on global memory.

Next we adapt the algorithms presented so far to the segmented case. Again we focus on the reduction step, since it exposes the differences among them. Thus, we suppose that data and owners already hold in shared memory, specifically in `s_data` and `s_owner`.

A. State-of-the-art Segmented Reductions

Zhou et al. [17] propose a tree-based algorithm by adapting (1) to the segmented case. The underlying recurrences are:

$$\text{s-red}(n) = \begin{cases} \text{s-red}(l(n)) \oplus \text{s-red}(r(n)) & \left[\begin{matrix} n \text{ is inner, and} \\ \text{owner}(l(n)) = \text{owner}(r(n)) \end{matrix}\right. \\ \text{s-red}(l(n)) & \left[\begin{matrix} n \text{ is inner, and} \\ \text{owner}(l(n)) \neq \text{owner}(r(n)) \end{matrix}\right. \dagger \\ n & [n \text{ is leaf} \end{cases}$$

$$\text{owner}(n) = \begin{cases} \text{owner}(l(n)) & n \text{ is inner} \\ \bar{n} & n \text{ is leaf} \end{cases}$$

Expressions $l(i)$ and $r(i)$ respectively denote the left and right children of an inner node i, and $\bar{n}$ is the owner of leaf n. Zhou et al. include an extra operation that must be applied when the siblings have different owners (label †). In this case, the s-reduction of the right child must be accumulated into the global solution. Specifically, each thread stores in `ai` the reduction of two elements, only if their owners are the same. Otherwise, only the left one is propagated, while the other is used to update `g_output`. Hence, the prefix of the processed chunk is propagated in a bottom-up manner, which agrees with this left-storing approach. Fig. 3 shows an example of a tree-

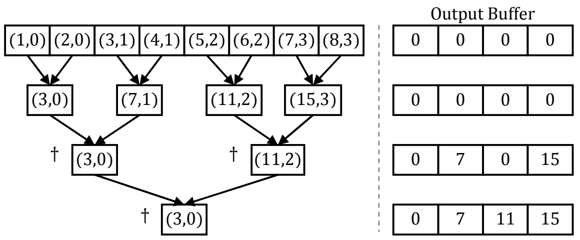


Figure 3. Example of a tree-based s-reduction. The reduction of the first segment has not been written to the output buffer yet.

based s-reduction for the + operator; on the left it illustrates the reduction process where each box contains a pair *(datum,owner)*, on the right it shows how the output buffer evolves as the algorithm progresses. Notice that the buffer is initialized with $1_\oplus$ values.

In order to adapt Algorithm 1 to the segmented case, it is enough to replace line 14 with the following code fragment:

```
14   unsigned int lo = s_owner[ai]; //left  owner
15   unsigned int ro = s_owner[bi]; //right owner
16   if(lo!=ro)
17      g_output[ro] = op(g_output[ro], s_data[bi]);
18   else
19      s_data[ai] = op(s_data[ai], s_data[bi]);
```

The partial result of the first segment inside this data-block ends at `s_data[0]` and `s_owner[0]`. The kernel is responsible for s-reducing a data-block, thus a multi-pass approach is required again to completely s-reduce a larger input. This is common to all the algorithms we present in this subsection.

Warps can also be exploited to improve Zhou et al.'s algorithm by avoiding some synchronizations. Basically, we must replace lines 7, 9, 11, 13 and 15 of Algorithm 2 with a proper call to the following device routine, in order to obtain `tb_s_reduceWarp`:

```
1    void s_reducePair_toTheLeft(float* g_output,
2           unsigned int oLeft,  float &dLeft,
3           unsigned int oRight, float dRight){
4      if(oLeft!=oRight)
5        g_output[oRight] = op(g_output[oRight], dRight);
6      else dLeft = op(dLeft, dRight);
7    }
```

Notice that prefixes are propagated again. Replacing in `TB4_Warp_reduction` any call to `tb_reduceWarp` with a call to `tb_s_reduceWarp` results in the intra-warp tree-based kernel for s-reduction.

Concerning sequential matrix-based reduction, we must replace lines 7-16 of Algorithm 3 with the following code fragment to cover the segmented case:

```
 7   float* dRow = &s_data[s_base];
 8   unsigned int* oRow = &s_owner[s_base];
 9   //The first element is specially processed
10   current_red   = dRow[0];
11   current_owner = oRow[0];
12   for(unsigned int k=1; k<W; k++){
13     if(current_owner!=oRow[k]){
14       g_output[current_owner] =
15             op(current_red, g_output[current_owner]);
16       current_owner = oRow[k];
17       current_red   = dRow[k];
18     }else
19       current_red = op(current_red, dRow[k]);
20   }//for
21   //Store the reduction of the row
22   s_data [thid] = current_red;
23   s_owner[thid] = current_owner;
24   tb_s_reduceWarp_toTheRight(s_data, s_owner, g_output,
25                 thid, thid&31, s_data[0], s_owner[0]);
```

Data layout is left-to-right, top-to-bottom, hence the suffix of the processed chunk inside the row is propagated now (line 12), and the tree-based s-reduction at lines 24-25 must be right-storing. The partial result of the last segment is sent to `s_data[0]` and `s_owner[0]`.

B. *Algorithmic Optimizations*

The techniques we proposed for the unsegmented case can be integrated into the previous state-of-the-art segmented algorithms. Specifically, we incorporate persistent blocks, and the producer-consumer scheme afterward, into the sequential matrix-based algorithm. This latter technique requires a considerable amount of shared memory, since two data-blocks for the elements and another two data-blocks for the owners are needed at the same time for a CUDA-block. Thus, occupancy decreases on current devices, which turns into a not competitive performance as we will see.

V. EMPIRICAL RESULTS AND DISCUSSION

We have used a NVIDIA GTX 480 (capability 2.0 -Fermi, 480 cores, 1536MB of GDDR5 global memory, configured as 48KB of shared memory per multiprocessor and 16KB of L1 cache), with driver 270.61, and the CUDA Toolkit, SDK and Compute Visual Profiler 4.0.

This paper focuses on empirically comparing algorithms by testing their straight implementations. Thus, we have not tuned the code as much as possible. In the experiments described below, the operator is the *minimum* function on *float* data (4 bytes). The timing information was obtained by reducing ten times a random input, and by taking their performance on average. The input size is $N = m * 2^{20}$, where m ranges from 16 to 31. We also tested other operators, e.g. + on *float* data, obtaining similar runtimes that are not included in the paper.

Concerning global memory accesses, Fermi architecture incorporates an on-chip cache hierarchy which is fairly configurable. Specifically, accesses can be cached in both L1 and L2, which is the default setting, or in L2 only. Since our implementations report similar runtimes under both configurations, the results we present below correspond to the default mode (L1 and L2).

A. *Unsegmented Reduction Results*

We have tested the five algorithms previously presented:

- `TB2:` the tree-based solution of Algorithm 1, with $R = 2$ and $D = 2B$.
- `TB4-warp:` the intra-warp tree-based solution included in Algorithm 2, with $R = 4$ and $D = 4B$.

- `Matrix:` the sequential solution of Algorithm 3, with $H = W = 32$, and $D = 1024$.
- `Persistent Matrix:` the previous `Matrix` solution incorporating persistent blocks.
- `Diffwarps:` the previous `Persistent Matrix` solution incorporating the producer-consumer scheme described in Algorithm 4. We use the term `Diffwarps` to express that warps carry out different tasks.

Notice that we have only incorporated the algorithmic optimizations into the sequential matrix-based reduction. The reason is that `Matrix` exhibits the best performance for unsegmented reduction.

The features of the five implementations are presented in Table I on the left. It includes *MBPM* for persistent solutions because it determines the corresponding grid size. Runtimes and effective bandwidths are shown in Fig. 4. `Diffwarps` exhibits the best throughput, with a bandwidth near to 104 GB/s. If B_r and B_w denote the number of bytes read and written by the algorithm, and $time$ is the runtime in seconds, effective bandwidth has been calculated as $((B_r + B_w)/10^9)/time$.

Table I on the right shows some empirical details for 30*2^{20} elements. These columns are especially relevant since they exhibit the behavior of each kernel execution individually. Notice that non-persistent solutions (`TB2`, `TB4_Warp` and `Matrix`) require three launches to completely reduce the input, while persistent ones (`P_Matrix` and `Diffwarps`) only need two. According to the Profiler reports, we include the following runtime information: grid size, global memory read throughput and average number of warps that are active on a multiprocessor per cycle *aw/ac*, which is calculated as (active warps)/(active cycles). Notice that bandwidth and *aw/ac* decrease as the reduction process advances. This is because the input for the first launch is larger than the input for consecutive launches. Hence, the GPU has less workload and becomes less efficient for subsequent launches, which explains why persistent solutions run faster.

B. *Segmented Reduction Results*

We have tested the corresponding five segmented algorithms. Their features are shown in Table II. Observe that the theoretical occupancy of most algorithms is smaller than those of their unsegmented counterparts. In the case of `Diffwarps`, it is so small (38%) that it is not competitive. Thus, we have added a variant which sequentially s-reduces a smaller matrix (H=32, W=16). Let `Diffwarps`$_{16}$ denote such solution. Notice that we then recover the 94% occupancy of the unsegmented version.

The way segments are distributed has a profound impact on the performance of all the algorithms, since the accesses to global memory, which are required when the left and right owners are different, called *irregular acce*sses in the sequel, are irregularly spread along execution. Thus, we have tested three scenarios: (a) a single huge segment covering the whole data, (b) segments of random size, ranging from 10 to 50, and (c) many small segments of size 3.

Fig. 5 shows the runtimes we have obtained for the three cases. Concerning state-of-the-art reductions, the figure shows two surprising facts: (1) `TB2` runs faster than `TB4-warp` in the three scenarios, and (2) tree-based solutions exhibit a better performance w.r.t. sequential ones as the segment size gets smaller. We will explain the reasons we find below. With regards to the algorithmic optimizations, `P_Matrix` and `Matrix` show similar performance in the three cases, and `Diffwarps` and `Diffwarps`$_{16}$ are not competitive in general, although the latter exhibits higher throughput.

Let us focus on scenario (a), that is, only one segment appears. Since irregular accesses do not take place, the results should be similar to those obtained for unsegmented reduction. Two facts remain: matrix-based beat tree-based methods, and `P_Matrix` improves `Matrix;` but two new issues come up. On the one hand, `Diffwarps`$_{16}$ is not among the fastest solutions in segmented reduction, while `Diffwarps` was the fastest in the unsegmented case. Each data-block requires loading the same amount of bytes in `Diffwarps`$_{16}$ (segmented) as in `Diffwarps` (unsegmented), since the matrix is halved (W=16) but owners are also loaded. On the contrary, the data size that is processed in a data-block is halved in `Diffwarps`$_{16}$ (W=16). Hence, read bandwidth gets halved, which finally results in a suboptimal performance. The Profiler endorses such claim since global memory read throughput falls from 104.17 GB/s in `Diffwarps` (unsegmented) to 53.67 GB/s in `Diffwarps`$_{16}$ (segmented), for the first launch on 30*2^{20} elements.

On the other hand, the relation between `TB2` and `TB4-warp` gets reversed from unsegmented to segmented reduction. `TB4-warp` has the advantage of requiring less explicit synchronization barriers, but it presents more intra-warp divergences because many threads inside a CUDA-warp are stalled inside function `tb_reduceWarp`. These divergences cause more harm to the segmented version of `TB4-warp` than to its unsegmented counterpart, because the divergent code is heavier in the segmented case due to `s_reducePair_toTheLeft`. Table III proves such assumption by showing the percentage of divergent branches the Profiler reports for 30*2^{20} elements. Observe that `TB2` and `TB4-warp` exhibit a similar divergence for unsegmented reduction, while `TB4-warp` is around ten times

TABLE I. KERNEL FEATURES FOR REDUCTION (LEFT). PROFILER'S REPORT FOR 30*2^{20} DATA (RIGHT).

Solution	#Registers	Shared Memory	Block Size	Theoretical Occupancy	MBPM	N= 30*2^{20}		
						#Blocks	Global Memory Read Throughput (GB/s)	aw/ac
TB2	9	2B+2B/nBanks	256	100%	-	61440, 120, 1	25.53, 17.58, 0.11	46.24, 36.37, 7.88
TB4-warp	13	4B+32	256	100%	-	30720, 30, 1	36.36, 16.45, 0.03	45.12, 22.21, 7.35
Matrix	10	H×(W+1)	256	100%	-	30720, 30, 1	92.88, 26.33, 0.01	39.48, 20.85, 6.24
Persistent Matrix	15	H×(W+1)	256	100%	6	90, 1	94.25, 0.68	45.25, 7.82
DiffWarps	15	2(H×(W+1))	288	94%	5	75, 1	104.17, 0.74	41.95, 8.83

TABLE II. KERNEL FEATURES FOR S-REDUCTION.

Solution	#Reg.	Shared Memory	Block size	Theoretical Occupancy	MBPM
TB2	10	2(2B+2B/nBanks)	256	100%	-
TB4-warp	14	2(4B+32)	256	83%	-
Matrix	11	2H×(W+1)	256	83%	-
Persistent Matrix	20	2H×(W+1)	256	83%	5
DiffWarps	25	4H×(W+1)	288	38%	2
DiffWarps_{16}	21	2H×(W+1)	288	94%	5

more divergent for segmented reduction.

With regard to the other scenarios, tree-based solutions are gaining positions as the segment size gets smaller. They overtake `Diffwarps` and `Diffwarps`$_{16}$ in case (b), and exhibit the best performance in case (c). The reason we find is that the probability of getting coalesced accesses, concerning irregular accesses, increases for tree-based reductions when segments are small. To explain it let us focus on segments of size 3, i.e. case (c). We have simulated the first launch of `TB2`, `TB4-warp` and `Matrix` to analyze the ratio of read data transfers to requested accesses. Transfers have been counted according to the way Fermi serves memory in chunks of 32 adjacent floats (128 bytes). Table IV examines what happens to the first CUDA-warp of the first CUDA-block. According to the values of D for each solution, `TB2` runs 9 recursive levels, `TB4-warp` requires 5 levels for intra-warp reductions, and `Matrix` processes 32 columns. Since the results of columns 1, 2 and 3 get repeated, only columns from 0 to 6 are presented in the table. In addition, `TB4-warp` and `Matrix` require 5 levels more to reduce intermediate results. They have been included in the table for `TB4-warp` (levels 5-9). Notice that `TB2` exhibits the lowest ratios, especially in the first levels, which indicates that irregular accesses are quite coalesced. On the contrary, the ratio for `Matrix` is always 1, that is, each request access is served with an independent transfer, which penalizes its performance.

VI. CONCLUSION AND FUTURE WORK

Sequential approaches have a better performance than tree-based ones for unsegmented reduction. With regards to the segmented case, performance depends on the distribution of segments. According to our results for regularly-spread segments, tree-based methods exhibit a higher bandwidth for small sizes, whereas sequential ones run faster for medium and large ones.

The two optimizations we have presented result in a speed-up for the unsegmented problem. Concerning the segmented case, performance is improved by using persistent blocks over segments of large size, while it remains the same for medium/small segments. `Diffwarps` is very shared-memory demanding, which makes its occupancy decrease. Thus, it is not competitive for segmented reduction on nowadays graphics hardware, although this could change for future devices since the current tendency has been to increase shared memory size.

TABLE III. PROFILER'S REPORT FOR $30*2^{20}$ ELEMENTS.

Solution	#Blocks	Divergent Branches (%)	
		Unsegmented	One Segment
TB2	61440, 120, 1	0.56, 0.70, 0.56	2.74, 2.74, 2.704
TB4-warp	30720, 30, 1	0.42, 0.42, 0.42	24.77, 24.81, 24.81

TABLE IV. ANALYZING IRREGULAR ACCESSES IN SEGMENTS OF SIZE 3. ONLY THE FIRST CUDA-WARP IS CONSIDERED.

TB-level/Matrix-column		0	1	2	3	4	5	6	7	8	9
TB2	accesses	11	21	32	32	16	8	4	2	1	-
	transfers	1	2	3	6	6	6	4	2	1	-
	trans./acc.	0.09	0.09	0.09	0.18	0.37	0.75	1	1	1	-
TB4-warp	accesses	5	5	4	2	1	16	8	4	2	1
	transfers	1	1	1	1	1	11	8	4	2	1
	trans./acc.	0.2	0.2	0.25	0.5	1	1	1	1	1	1
Matrix	accesses	11	10	11	11	10	11	11	…	…	…
	transfers	11	10	11	11	10	11	11	…	…	…
	trans./acc.	1	1	1	1	1	1	1	…	…	…

The optimizations only have been integrated into the sequential matrix-based algorithm, and we plan to test them onto the tree-based methods. Finally, the reduction is a part of the scan primitive and the optimizations presented in this paper could be embedded in the scan algorithms to improve their performance.

ACKNOWLEDGMENT

Research supported by the Spanish Projects CCG10-UCM/TIC-5476 and GR35/10-A-921547.

REFERENCES

[1] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," Proc. High Performance Graphics (HPG 09), ACM, 2009, pp. 145–149, DOI=10.1145/1572769.1572792.

[2] M. Bauer, H. Cook, and B. Khailany, "*CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization*," Proc. Int. Conference for High Performance Computing, Networking, Storage and Analysis (SC '11), ACM, November 2011, DOI=10.1145/2063384.2063400.

[3] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA TR NVR-2008-004, Dec. 2008.

[4] G. E. Blelloch, Vector Models for Data-Parallel Computing, MIT Press, 1990.

[5] G. E. Blelloch, "Prefix sums and their applications," Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1993.

[6] CUDA data parallel primitives library (CUDPP). http://gpgpu.org/developer/cudpp.

[7] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," Proc. *international conference on Supercomputing* (ICS 08), ACM, 2008, pp. 205-213, DOI=10.1145/1375527.1375559.

[8] General-Purpose Computation on Graphics Hardware http://gpgpu.org/

[9] A. Gress, M. Guthe and R. Klein, "GPU-based collision detection for deformable parameterized surfaces," Computer Graphics Forum 25, 2006, pp. 497–506, DOI: 10.1111/j.1467-8659.2006.00969.x.

[10] M. Harris, Optimizing Parallel Reduction in CUDA, (2007), http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf.

[11] D. Horn, "Stream reduction operations for GPGPU applications," GPU Gems 2, M. Pharr (ed.), Addison Wesley, 2005, pp. 573–589.

[12] P. J. Martín, R. Torres, and A. Gavilanes,. "CUDA Solutions for the SSSP Problem," Proc. International Conference on Computational

Science (ICCS 09), Springer-Verlag, 2009, pp. 904-913, DOI=10.1007/978-3-642-01970-8_91.

[13] S. Sengupta, A. E. Lefohn and J. D. Owens, “A work-efficient step-efficient prefix sum algorithm,” Proc. Edge Computing Using New Commodity Architectures, 2006, pp. 26–27.

[14] S. Sengupta, M. Harris, Y. Zhang, and J. Owens, “Scan Primitives for GPU Computing,” Proc. Graphics Hardware (GH 07), ACM, 2007, pp. 97-106.

[15] S. Sengupta, M. Harris, M. Garland, “Efficient Parallel Scan Algorithms for GPUs,” NVIDIA TR NVR-2008-003, Dec. 2008.

[16] W. Wang, Y. Huang and S. Guo, “Design and Implementation of GPU-Based Prim´s Algorithm,” Modern Education and Computer Science, 4, MECS-Press, 2011, pp 55-62.

[17] K. Zhou, Q. Hou, R. Wang and B. Guo, “Real-time kd-tree construction on graphics hardware,” Proc. ACM SIGGRAPH Asia 2008, ACM, 2008, pp. 1-11, DOI=10.1145/1457515.1409079.

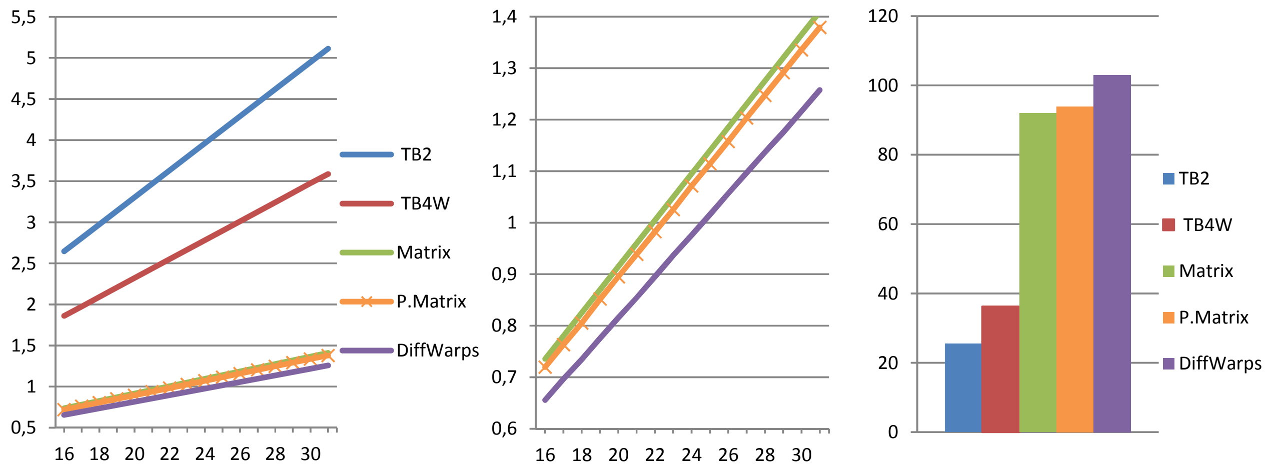
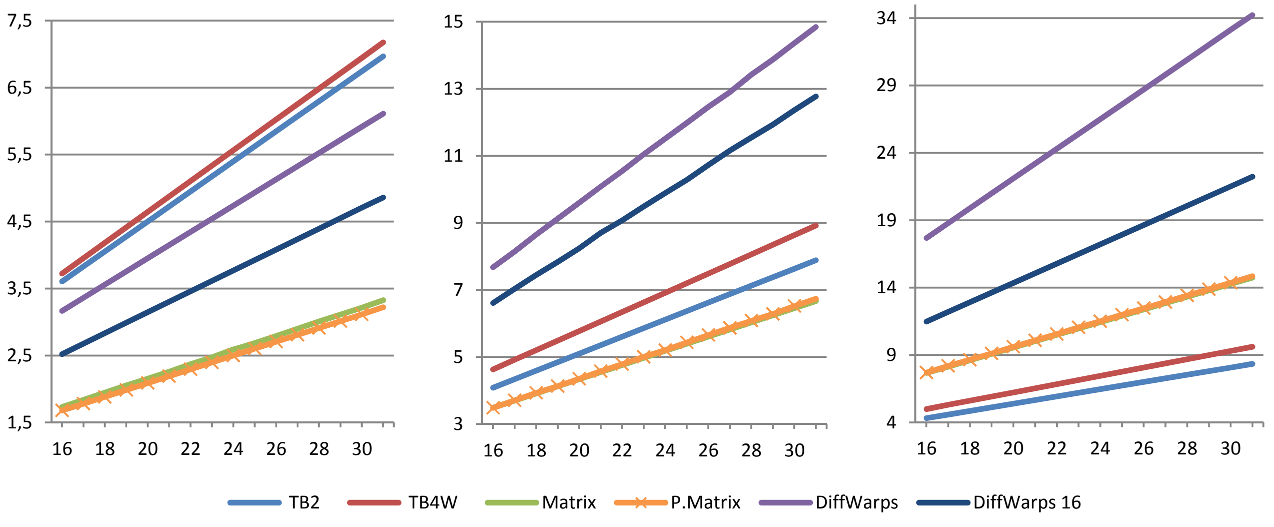


Figure 4. Results for unsegmented reduction. (Left and center) runtimes in ms (Y-axis). Input size is x*2^{20} elements (X-axis). (Right) effective bandwidths (GB/s).



(a) (b) (c)

Figure 5. Runtimes in ms (Y-axis) for s-reduction. Input size is x*2^{20} elements (X-axis). (a) Only one segment, (b) random sized segments, (c) segments of size 3.