2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing

Efficient Scan Operator Methods on a GPU

Adrián P. Diéguez
Computer Architecture Group (GAC)
University of A Coruña
Spain
Email:*adrian.perez.dieguez@udc.es*

Margarita Amor
Computer Architecture Group (GAC)
University of A Coruña
Spain
Email: *margarita.amor@udc.es*

Ramón Doallo
Computer Architecture Group (GAC)
University of A Coruña
Spain
Email: *doallo@udc.es*

***Abstract*—Current GPUs (*Graphics Processing Units*) offer high computational power at relatively low cost; nonetheless, this enhanced performance often comes at the expenses of flexibility and code complexity. Efficient GPU programming requires detailed knowledge on certain hardware aspects. The scan operator is an important building block for a wide range of algorithms. In this paper, we present a number of parallel scan methods based on the traditional cyclic reduction tridiagonal solver and the Ladner-Fischer parallel prefix adder. Futhermore, we analyze a set of new features introduced in the Kepler Nvidia architecture such as read-only data cache and shuffle instructions. Our methods provide an excellent performance in many cases, up to 48% improvement over the CUDA Data Parallel Primitives (CUDPP) library.**

I. Introduction

The scan operator is widely used in areas such as the construction of summed area tables [1], stream compaction [2], sorting [3], image filtering [1], Brownian values generation [4], polynomial evaluation [5] or cryptography [6], among many others.

Scan primitive in VLSI adders was proposed by Sklansky in 1960 [7]. Most implementations on GPU are based on either the Kogge-Stone or the Brent-Kung parallel prefix patterns. Figure 1 shows a taxonomy of these parallel prefix algorithms.

The Kogge-Stone VLSI adder pattern [8] was designed for a small VLSI area, being work inefficient. Hillis and Steele, in 1986, adapted this algorithm for supercomputing with $O(N \cdot \log_2 N)$ complexity. The Hillis and Steele algorithm was demonstrated for GPUs in 2005 by Horn [9] who used this scan for a non-uniform stream compaction operation as well as for a collision-detection application. Later, Hensley et al. proposed scan for summed-table area generation in [10], improving on Horn's implementation by pruning unnecesary work, but still getting $O(N \cdot \log_2 N)$ complexity. The first *O(N)* implementation was published in 2006 [2] by Sengupta who went to present an important improvement in 2008, emphasizing depth optimality as a key performance parameter in GPU implementation [11]. Finally, the popular CUDPP library [12] contains a scan implementation based on this pattern.

On the other hand, the Brent-Kung pattern is the VLSI area efficient method, using two balanced binary trees to obtain this efficiency. Although the use of two binary trees obtains *O(N)* complexity, the overall depth is increased where computations take twice as long to complete. Blelloch developed this primitive in an efficient way for supercomputing in 1990. This

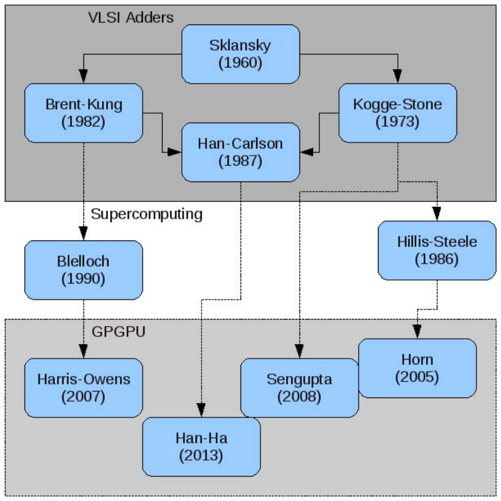


Fig. 1. Taxonomy of parallel algorithms for scan operator based on the parallel prefix adder patterns.

pattern was implemented for GPUs in 2007 by Harris-Owens (HO in the present paper) [3].

There exists another approach in VLSI adders, which was developed by Han-Carlson in 1987. This pattern combines both the Brent-Kung and the Kogge-Stone algorithms, seeking a tradeoff between area and time required. In 2013 [13], a GPU work efficient algorithm based on this third pattern was proposed.

Our work analyzes previous approaches toward GPU scan primitive implementation and adapts patterns from other fields to the scan operator. Futhermore, new Kepler architecture features are used and analyzed, such as read-only data cache, 8-bytes wide accesses to shared memory, shuffle instructions and CUDA Dynamic Parallelism.

The remainder of this paper is organized as follows: Section 2 presents the traditional parallel patterns for the scan computation, their GPU implementations and some proposals based on these patterns. In Section 3, we adapt other prefix graph patterns to scan operator and their GPU implementation, whereas a number of CUDA optimization techniques are presented in Section 4. Experimental results are discussed in Section 5, and our conclusions in Section 6.

II. Scan Operator Formulation with Prefix Graph

This section presents the scan operator and the classical algorithms used for solving this operator. The *scan* operator is defined as an associative and binary operator $\oplus$ with identity *I*, where, given an input array of *N* elements $[a_0, a_1, ..., a_{N-1}]$, it returns

$$[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{N-2})\ ]$$

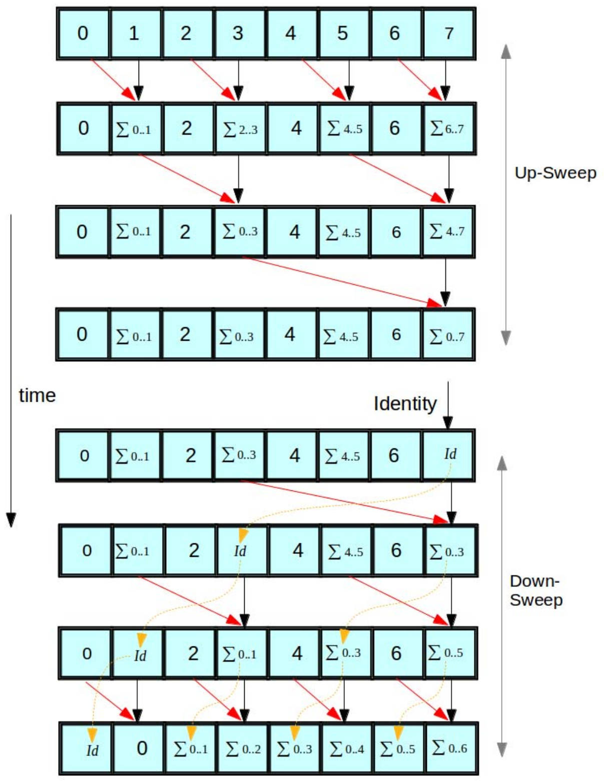
The scan operator defined here is an *exclusive* scan, since the a_j element is not taken into consideration for calculating the position *j* of the result. Otherwise, it would be an *inclusive* scan, but the transformation from one to the other is trivial. Hereinafter, we will use addition as scan operator in our examples.

The algorithm performs *N* adds for arrays of *N* elements; thus it possesses an *O(N) work complexity* and the calculation of the *i* element requires of the calculation of $i-1$ elements.

A parallel computation is *efficient* in terms of work when it does not perform more work than its sequential version; in other words, both versions have the same complexity. This approach is known as the *work and depth based model*. On the other hand, the *processor based model* takes into account computing costs, such as execution times, the number of processors, synchronization barriers or communications costs. Owing to this fact, this work does not analyze the complexity of the algorithms; only final execution times are considered, taking a *processor based* approach.

The analysis of different proposals of scan operator permits us to classify them in terms of their prefix graph, which enables us to describe the operations carried out on the data. In order to continue, certain concepts related with our proposals and CUDA threads notation [14] need to be introduced. *NVIDIA GPUs* comprise a set of *SMs* (Streaming Multiprocessors). Each *SM* consists of many *SPs* (Streaming Processors) and *SFUs* (Special *Function Units*), which are simple but energy-efficient processing units that execute instructions in a *SIMD* fashion (*Single Instruction Multiple Data*). The main memory of GPUs, called global memory, can be accessed from all the SPs in every SM. Furthermore, each SM has a set of registers and a small memory (shared memory) that can be accessed by threads from the same block. This shared memory is much faster than global memory but it offers less effective band- width than registers. Each kernel is invoked by a set of threads that are grouped into thread blocks. Each thread block is assigned to an SM and the shared memory of SM can be accessed by all threads of block. A small number of registers are assigned to each thread of block. A more detailed description of *NVIDIA's GPU* architecture can be found in [14].

A data sequence of size $N = r^n$ is executed in *M* stages, where L^k operations (butterflies) are performed in each stage, where *k=0,..,M-1*. The number of reductions (butterflies) computed by each thread is denoted by b, N_{tb} is the maximum number of threads per block, and C indicates the number of blocks.



0 1 2 3 4 5 6 7
0 Σ0..1 2 Σ2..3 4 Σ4..5 6 Σ6..7
0 Σ0..1 2 Σ0..3 4 Σ4..5 6 Σ4..7
0 Σ0..1 2 Σ0..3 4 Σ4..5 6 Σ0..7
Up-Sweep
time
Identity
0 Σ0..1 2 Σ0..3 4 Σ4..5 6 Id
0 Σ0..1 2 Id 4 Σ4..5 6 Σ0..3
0 Id 2 Σ0..1 4 Σ0..3 6 Σ0..5
Id 0 Σ0..1 Σ0..2 Σ0..3 Σ0..4 Σ0..5 Σ0..6
Down-Sweep

Fig. 2. Brent-Kung pattern for addition with N=8.

A. Brent-Kung Pattern

The Harris-Owens proposal (HO) is based on the Brent-Kung pattern [15] [16], reducing complexity through the use of two balanced binary trees, *O(N)*. In this case, $M = 2{\cdot}log_2 N$ and $r = 2$. Building the input tree, the next step is to sweep it to and from the root in two phases: *up-sweep phase* and *down-sweep phase*. For the up-sweep phase, $L^k = \frac{N}{2^{k+1}}$, and $L^k = 2^{k-\log_2 N}$ for the down-sweep phase. Figure 2 shows this pattern for $N = 8$ elements. The *up-sweep* phase computes partial sums at internal nodes, whereas the *down-sweep* phase uses previous partial sums for building the scan in place. In the *down-sweep* phase, there also exists a stage where values are passed to its child node (broken line notation in Figure 2).

For dealing with large arrays, thread blocks need to co-operate with each other. With this in mind, this proposal uses a recursive processing [3]: after each block computes its own scan, it saves the accumulative addition of the whole block in an extra-array element for the processing itself. When the extra-array can be computed by a single block, then the corresponding element is added to all elements of each block. Limiting factors are the single use of *radix* $r = 2$, since there are available free registers in many cases and *r* can be incresead, the existence of two phases for computing scan doubles the number of synchronization barriers, and the presence of warp divergence in some stages. This strategy is also similar to the method implemented in [17].

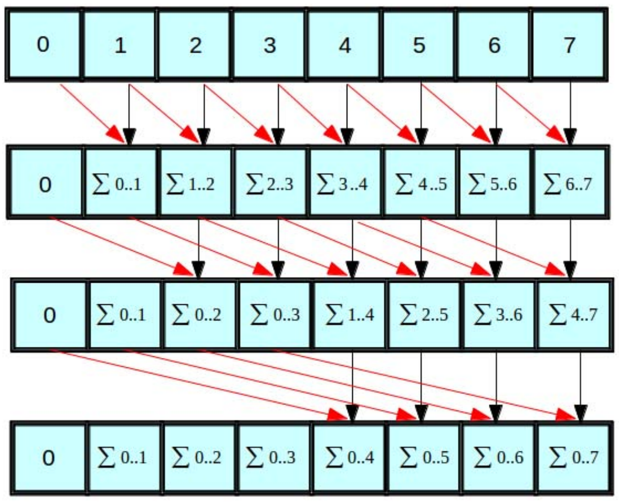


Fig. 3. Kogge-Stone pattern for addition with N=8.

B. *Kogge-Stone Pattern (KS)*

The Kogge-Stone pattern [8] is used in parallel prefix form carry look-ahead adder. This takes up more area than Brent-Kung adder but this layout has a minimum depth, which increases performance. The work complexity is observed as *O*($N \cdot \log_2 N$). This pattern has $M = \log_2 N$ stages, $r = 2$ and each stage has $L^k = N - 2^k$ computations. Figure 3 depicts the KS pattern for $N = 8$ elements. This pattern is considered work inefficient in comparison to the serial implementation which is bounded by *O(N)*.

Merrill [18] added the funcionality to combine the Kogge-Stone and Brent-Kung patterns, using performance configurations. However, an important change in his implementation is the thread-block execution. In order to produce fewer intermediate values, Merrill's implementation sets the number of thread blocks C before execution as a constant value. In the first phase (or kernel), each thread block computes the reduction for an input data set, which produces C intermediate values to be stored; in other words, there are C accesses to global memory. The second kernel reads back these C values and produces their scan. Finally, the third kernel reads in all of the input elements and updates them. Hence, the total amount of global memory required is $3N + 3C$. The value for C has to be small enough to compute the scan but large enough to obtain the maximum warp occupancy per SM.

On the other hand, Dotsenko's strategy [17] computes large arrays reducing to one partial sum per thread block, which is then prefix summed and sent to each thread block to be updated. In consequence, the number of thread blocks for invoking the kernel depends on the recursive level.

1) Kogge-Stone based proposals: We have developed two Kogge-Stone proposals, which hereinafter are referred to as KS-p and DA, respectively. The KS-p proposal presents coalesced access to global memory and avoids bank conflicts. Futhermore, each thread computes b butterflies in each stage in order to increase the workload per thread (instead of $b = 1$ for Horn's implementation). The main problem of this pattern is the presence of divergence in the final warp of each thread block, due to the fact that L^k decreases by a factor of 2^k in each stage. Since $L^k = N - 2^k$ and b is a power-of-2, the last thread would have to compute fewer than b butterflies in some stages, introducing more warp divergence with an extra conditional instruction in code. For example, if $b = 2$ and N is a power of 2, there are $N - 1$ binary reductions in first stage, but $N - 1$ is not divisible by b. To solve this, shared memory is increased in b elements as Figure 4 shows.

Our second proposal with CUDA vector types is called DA. This approach loads data from global memory using *float4* types to registers. In the first stage, each thread computes a 4-element chunk sequentally. The forth element of each thread is saved in shared memory to be processed by the KS algorithm. DA has coalesced accesses reducing the number of transactions to global memory and the amount of shared memory needed. Furthermore, memory latency is also reduced, since the first stage is completely performed in registers.

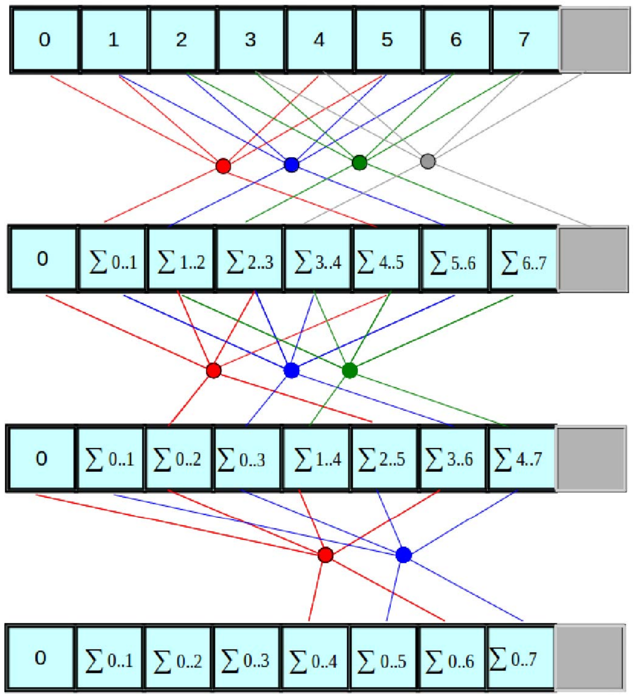


Fig. 4. KS-p proposal implementation for N=8 with b=2.

Finally, our thread block execution proposal is based on a cascade approach. This proposal combines three kernels for computing scan efficiently: *rd* refers to the kernel that is invoked with C threads blocks, where each block computes the reduction for a chunk of $\frac{N}{C}$ elements in the cascade approach, enabling template metaprogramming and unroll techniques. This entails computing the reduction over a loop where each iteration processes $b \cdot N_{tb}$ elements, using shared memory for communication, and storing the value in an auxiliar array. On the other hand, p is invoked with only one thread block and performs the scan of kernel *rd* result. Subsequently, the third kernel, denoted as *p2*, performs the global scan in C thread blocks, using previous kernel results and the cascade approach.

C. Han-Carlson Pattern

Han-Ha's proposal [13] is based on the Han-Carlson pattern [19], which combines both Kogge-Stone and Brent-Kung method. This pattern makes it possible to obtain a tradeoff between area and time required. At the beginning and at the end, there exists a Brent-Kunt stage, whereas Kogge-Stone works in the middle of the graph. However, this proposal also offers an $O(N \cdot \log_2 N)$ complexity. Figure 5 represents this pattern.

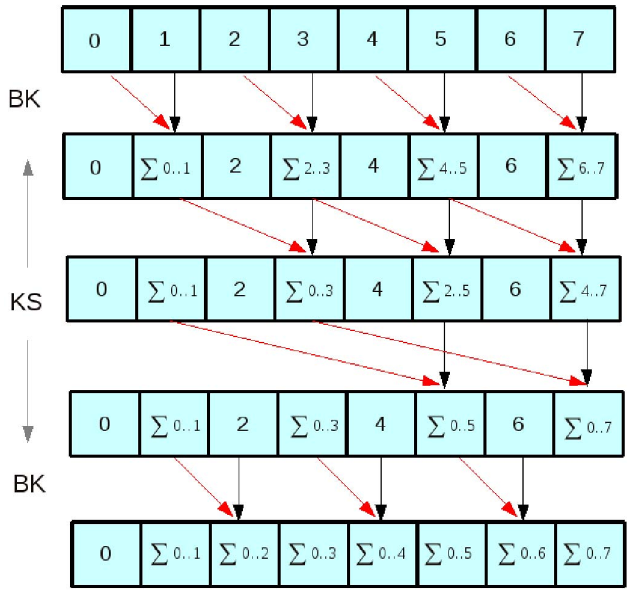


Fig. 5. Han-Carlson pattern for addition with N=8.

The major improvement in the Han-Ha proposal is the cascade thread block execution based on Merrill execution. Each thread block iterates over a loop, processing a chunk of data in parallel in each iteration, using shared memory for communication between iterations. The loop depth is also fixed prior to execution. The partial accumulative sum of the block is then shared with the adjacent block, which repeats previous procedure taking into account the partial accumulative sum. However, the biggest difference between Merrill and Han-Carlson is the absence of redundant global memory accesses at the beginning and the end of the thread blocks. Instead of using a homogeneous kernel across all of the SMs, Han-Carlson does not reduce the first thread block, since this reduced value can be obtained by scanning the block, avoiding the scanning process in third kernel call. Moreover, the reduced value for the last block is truncated as this result is no longer used. This approach takes $(3n + 3C) - (\frac{2n}{C} + 3)$ accesses to global memory.

III. Formulation of Scan Operator from other Prefix Graphs

In this section we consider other prefix graph patterns used in other fields, such as solving tridiagonal systems. These patterns have been modified in order to be used with the scan operator. In these proposals, we have implemented the same cascade strategy for dealing with large arrays as in KS-p and DA proposals.

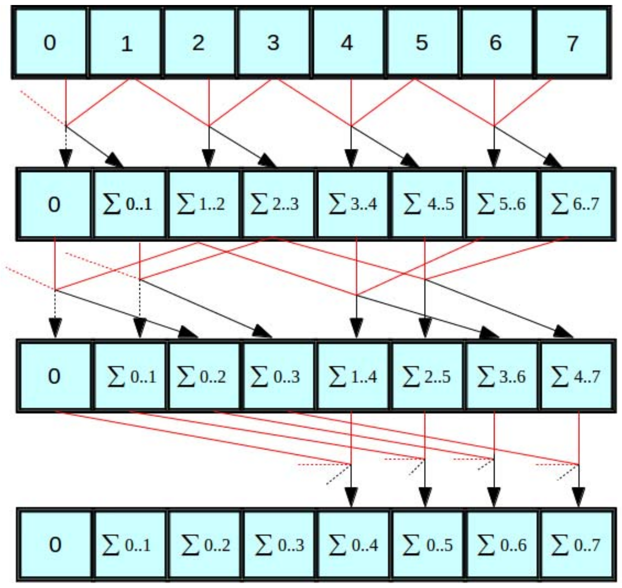


Fig. 6. HJ scan proposal for addition with N=8.

A. Hockney and Jesshope Pattern (HJ-scan)

Tridiagonal systems of equations (TS) are used to solve differential equations in partial derivatives, such as simple harmonic motion, Helhortz, Poisson or Laplace equations. There are many algorithms for solving TS, including the Cyclic Reduction (CR) proposed by Hockney and Jesshope [20]. CR consists of two phases: elimination and substitution. The substitution phase uses an inverse binary tree, while the elimination phase uses a complete direct binary tree that has three inputs and two outputs but only one of the outputs progresses to the next stage.

Our HJ-scan proposal is based on the elimination phase. The butterfly operator is defined as B^i_j, with $0 \leq i < M$, $M = \log_r N$, $r = 2$ and $0 \leq j < \frac{N}{r}$. If $j < 2^i$, it reads a set of 2 elements spaced 2^i positions and writes over only one element; whereas if $j \geq 2^i$ it reads a set of 3 and writes over two of them. To determine the elements to process, the butterfly j loads the central element with position $\frac{j}{2^i} \cdot 2^i + j$, the elements spaced 2^i positions to the left and 2^i positions to the right, and then saves the result of operating central and left elements in the central position, and the result of central and right elements is saved in the right position. Figure 6 illustrates the procedure for $N = 8$ with $r = 2$ and pseudocode is presented in Figure 7.

Concerning its implementation, it presents coalesced access to global memory. Furthermore, keeping the number of active threads during stages constant, warp divergence is avoided. This advantage is not free, it being neccesary to pay the price of bank conflicts. Empirically for this case, the overhead generated by implementing padding is not offset by the small bank conflict reduction achieved. Since Kepler supports 8-byte wide shared memory banks, the number of bank conflicts

```
for i:=0 to $\log_2(N)-1$ do
for all j=0 to N/r in parallel do
        index=$\frac{j}{2^i}*2^i+j$
        c:= x[index]
        if(index$-2^i$>=0) then
                x[index]$\oplus$=x[index$-2^i$]
        x[index$+2^i$]$\oplus$=c
```

Fig. 7. Pseudocode of HJ scan proposal.

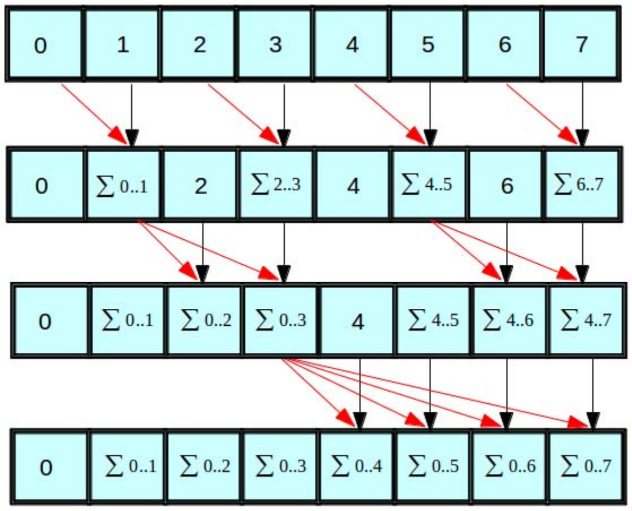


Fig. 8. LF-scan proposal for addition with N=8.

are minimal. This implementation allows different b values for increasing workload per thread while there exist available resources.

B. Ladner-Fischer Pattern (LF-scan)

The LF-scan is our own adaptation of the reduction algorithm, *Ladner-Fischer parallel prefix adder* [21], to scan operation. Figure 8 shows the scan operation over an array of 8 elements with addition. The algorithm is based on the Brent-Kung reading stage, but computing a block of 2^i adjacent positions with each element. Thus, the scan operation is performed in $M = \log_r M$ stages with $r = 2$. Unlike other algorithms, here the number of read and write operations remains constant over all stages. Figure 9 contains pseudocode for LF-scan pattern. Although index calculation in pseudocode would seem to be somewhat costly, its CUDA implementation is trivial using bit masks.

This implementation has coalesced accesses to global memory, allows the specification of any r (power of 2), can be used for large arrays and keeps the number of active threads in each stage constant. As in the HJ-scan, there are a small number of bank conflicts which, thanks to Kepler 8-bytes wide shared memory banks, do not need to be fixed. Unlike previous proposals, there are not two separate phases for read and write operations since elements read by threads are never overwritten in the same iteration by other threads. Hence, the LF-scan saves an extra synchronization barrier in each stage.

```
for i:=0 to $\log_2(N)-1$ do
for all $\left(j<\frac{N}{r}\right)$ in parallel do
x[$2^i+\left(\frac{j}{2^i}\right)*2^i+j$]$\oplus$= x[$2^i+\left(\frac{j}{2^i}\right)*2^{i+1}-1$]
```

Fig. 9. Pseudocode of LF-scan proposal.

IV. Optimization CUDA Techniques for Scan Primitive

In this section, different optimization techniques are applied to achieve an efficient implementation of our scan proposals.

Whereas there exist available registers, it is possible to increase the workload per thread; i.e., increasing radix. If the number of registers is insufficient, local memory is used, which involves latency penalizations. Moreover, a trade-off needs to be found between registers use and warp occupancy.

Moreover, increasing b needs to take into consideration shared memory bank conflicts. Since each thread computes b butterflies, it is important to define the elements the thread accesses. In general, accessing to adjacent pairs of elements per thread generates a high percentage of bank conflicts. CUDA implementation of HO, HJ-scan and LF-scan produces bank conflicts in shared memory. HO uses *padding* techniques for solving shared memory conflicts. However, HJ-scan and LF-scan access patterns are completely different to HO, it not being possible to find a padding function that solves all bank conflicts in all stages. Different strategies have been checked, introducing an overhead in execution times. These bank conflict rates practically disappear in Kepler due to its support for 8-byte wide accesses.

Techniques explained below, such as mapped memory (MM), in-place arrays (IP), read-only data Kepler cache, shuffle instructions or the use of CUDA vector types have achieved good performance in our tests.

Other methods, such as doubling the processing stages for saving global memory accesses or Cuda Dynamic Parallelism (CDP), have been also checked seeking performance, but empirical results have demonstrated that these techniques do not improve execution times. Although CDP is a new Kepler feature, it gave rise to poor performance owing to the use of local memory as heap in each *dynamic* call.

To identify which technique was used in each proposal, an acronym is added to the primary identifier (KS-p, DA, HJ-scan or LF-scan). Specifically, *Mapped Memory* adds *-MM*, *-IP* for *In-Place Arrays*, *CDP* for *CUDA Dynamic Parallelism*, *-shuffle* for *shuffle instructions* and finally, CUDA Vector types technique adds a number to the end of identifier, referring to the number of elements that compose the vector.

A. Mapped Memory [-MM]

Mapped Memory techniques have been applied to improving performance in KS-p, HJ-scan and LF-scan proposals. Sharing the same memory space between *host* and *device* brings benefits when the number of elements to compute is so small that launching a kernel is computationally more expensive than running the scan on CPU. The host structure is modified for computing the result in CPU when C is so small that the second kernel has an insufficient workload. In such a case, memory allocation is performed with the *cudaHostAlloc* function as input data is written by the previous kernel.

B. *In-Place Arrays [-IP]*

The KS-p, HJ-scan and LF-scan proposals were also tested using in-place arrays, improving perfomance thanks to the locality property. However, using the same array for input and output could be a problem for some real applications in which original input data is reused.

C. *Read-Only Data Cache in Kepler*

Previous proposals takes advantage of a read-only data cache when they are run in Kepler architecture. Kepler introduced a new method to load data easily through the textures (read-only) cache. A new instruction called LDG does not require the binding of textures and has no size restrictions. Since the texture cache is not coherent, safe use of LDG requires that an object is read-only throughout the entire lifetime of the kernel. To this end, constant array parameters are modified with *const __restrict__* keyword. Furthermore, this cache has a pipeline separated from L1, more bandwith and flexibility (no aligned accesses) than constant memory.

D. *Shuffle Instructions*

Shuffle instructions allow information to be shared using registers, instead of shared memory, between threads in the same warp. The use of these instructions frees up shared memory to be used for other data or to increase the occupancy. Moreover, shuffle instructions are faster since the require only one instruction instead of three (write, synchronize and read), avoiding warp-synchronization barriers. We have implemented our KS-p and LF-scan proposals in terms of shuffle instructions denoted as KS-shuffle and LF-shuffle.

Specifically, KS-shuffle uses *__shfl_up* instruction for computing KS scan in each warp. After this, each warp saves its partial sum in shared memory, and the process is repeated by only one warp over the values stored in shared memory. Finally, each block saves its accumulative sum in global memory as input data for the recursive process.

LF-shuffle is similar to KS-shuffle implementation, but uses *__shfl* instruction instead of *__shfl_up* for implementing the LF pattern.

E. *CUDA Vector Data Types*

In order to increase performance, CUDA provides built-in vector data types, such as *float2*, *float3* and *float4*. Global memory transactions are 128 bytes aligned, and are always performed by one full warp at a time. When a warp reaches a function that performs a memory transaction, it forces the chip to process as many transactions as needed to service all the threads in the warp. In the case of float types, if all accessed 32-bits coalesced values are within a single 128-byte line, only one transaction is necessary. Otherwise, multiple 128-byte transactions are performed, presenting a latency of 400 to 600 cycles per transaction.

After analyzing the use of CUDA vector data types in our proposals, the best performance was reached in LF-shuffle with *float4* types. The implementation of LF-shuffle with *float4* is denoted as LF-shuffle4 in our graphics. Futhermore, DA approach (see Section II-B1) loads data from global memory using *float4* types to registers.

V. EXPERIMENTAL RESULTS

In this section, the results of the comparision between our proposals, the CUDPP library and HO (Nvidia) algorithms are presented. All the tests were run in single precision using input arrays in the range $N = \{524288, 2097152, 8388608\}$ over 30 iterations. The test platform used in our experiments is composed of *Intel Xeon E5-2660 CPU* at 2.2 *GHz* with 64 *GB DDR3* at 1600 *MHz*, *CentOS 6.4 with GCC 4.4.7* as compiler and one *Nvidia Tesla K20 GPU* of *5 GB GDDR5* the driver version being *v304.54, SKD 5.0*. The kernels were executed setting the *cudaFuncCachePreferShared* cache configuration flag, in order to be able to use up to 48 *KB* of shared memory. The configurable K20 bank size was set to eight bytes (*cudaSharedMemBankSizeEightByte*). The ECC memory was enabled. To provide additional benchmark data, tests were also executed on a *GeForce GTX580* with driver *304.116, SDK 5.0* and *1.5 GB*. The desktop platform for this GPU is composed of an *Intel Core i7-2600 at 3.4 GHz* 8 *GB* with *Ubuntu 12.04 LTS* of 64-bits.

For each proposal, several thread block sizes, butterflies and radix parameters have been analyzed. Regarding the kernel profiler analysis, we used *NVIDIA Visual Profiler* to measure performance in tests. Table I shows detailed information on the configuration and optimization CUDA technique that provides the best performance for each proposed pattern in *Tesla K20 GPU*. Specifically, this table shows the execution time in seconds, the number of threads per block (N_{tb}), the number of blocks (C), the number of registers, the amount of shared memory in bytes and the shared memory reply rate (bank conflicts). Values displayed in brackets refer to the corresponding values of kernels *rd, p, p2* respectively. Note that there is no local memory spilling. This is confirmed by the local memory replay rate, which was not included in the table as the value thereof is always 0. Futhermore, the warp occupancy is maximum in all cases (64 warps per SM). Table II repeats the profiler analysis for the *GeForce GTX580 GPU*, presenting the same information as Table I, with no local memory spilling, although the warp occupancy is maximum (48 warps per SM) only in the DA proposal, being 32 warps per SM in the remaining cases.

Figure 10 compares our best results to CUDPP implementations in Tesla K20. Although Harris-Owens (HO) results are not represented graphically, as they are off scale with respect to the other methods results, its performance is also compared, reporting the percentage of improvement of those methods with respect to HO. HJ-scan proposal presents an improvement of up to 9.70% over CUDPP implementation and 60% over HO. The SM saturation is achieved with $b = 2$, $r = 2$ and $N_{tb} = 128$, and there is a small portion of bank conflicts due to the access pattern in shared memory. In comparison with other proposals, the first stage has to be computed in shared memory instead of registers as the same value is shared by several threads. With reference to the KS-shuffle, it shows a speedup

Proposal	N	Exec. time (sec)	N_{tb}	C	Reg	SH mem	SH reply (%)
KS-shuffle	524288	0.000108	128	256	[13,13,13]	[512,128,132]	0
	2097152	0.000256	256	512	[13,13,13]	[1024,128,132]	0
	8388608	0.000897	256	2048	[13,15,13]	[1024,128,132]	0
LF-shuffle4	524288	0.000058	128	256	[13,21,29]	[512,2180,132]	0
	2097152	0.000164	256	512	[13,21,29]	[1024,4228,132]	0
	8388608	0.000596	128	2048	[15,31,30]	[512,2180,132]	0
HJ-scan	524288	0.000076	128	256	[13,25,30]	[512,2048,1040]	[0,1.6,2.6]
	2097152	0.000216	128	1024	[13,29,30]	[512,2048,1040]	[0,1.7,2.6]
	8388608	0.000764	128	2048	[15,30,30]	[512,2048,1040]	[0,1.7,2.8]
DA	524288	0.000065	128	256	[13,22,20]	[512,2048,528]	0
	2097152	0.000174	256	512	[13,22,20]	[1024,4096,1040]	0
	8388608	0.000613	256	2048	[13,20,20]	[1024,4096,1040]	0

TABLE I. Scan performance and profiler analysis for the different proposals in Tesla K20. Notation in brackets refers to values of kernels [*rd,p,p2*]

.

Proposal	N	Exec. time (sec)	N_{tb}	C	Reg	SH mem	SH reply (%)
KS-p	524288	0.000078	128	256	[21,16,15]	[512,1028,1040]	0
	2097152	0.000283	128	1024	[21,20,15]	[512,1028,1040]	0
	8388608	0.001064	128	4096	[21,22,15]	[512,1028,1040]	0
LF-scan	524288	0.000051	128	256	[21,23,30]	[512,2048,1040]	[0,31.3.15.2]
	2097152	0.000171	128	512	[22,23,30]	[512,2048,1040]	[0,28.8,16.6]
	8388608	0.000628	128	1024	[23,27,30]	[512,2048,1040]	[0,29,17.4]
HJ-MM	524288	0.000065	128	128	[22,22,28]	[512,2048,1040]	[0,47.5,16]
	2097152	0.000216	128	512	[22,22,28]	[512,2048,1040]	[0,39.9,16.1]
	8388608	0.000801	128	128	[23,27,29]	[512,2048,1040]	[0,41.1,18]
DA	524288	0.000050	128	256	[21,16,18]	[512,2048,528]	0
	2097152	0.000162	128	512	[22,16,20]	[512,2048,528]	0
	8388608	0.000613	256	2048	[21,20,18]	[1024,4096,1040]	0

TABLE II. Scan performance and profiler analysis for the different proposals in GTX580. Notation in brackets refers to values of kernels [*rd,p,p2*]

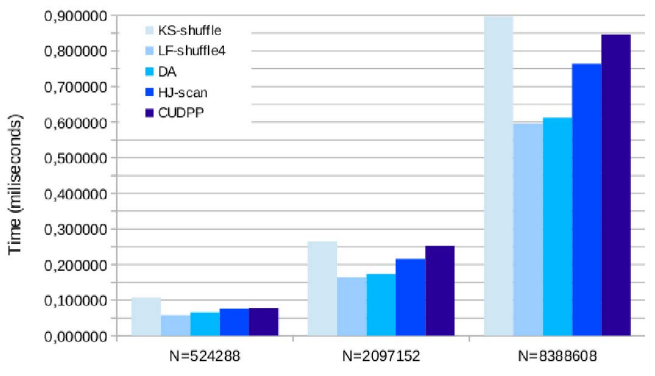


Fig. 10. Performance comparison of CUDPP library and proposed methods in Tesla K20.

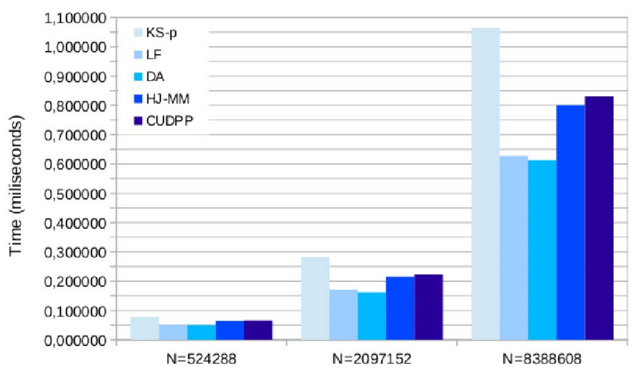


Fig. 11. Performance comparison of CUDPP library and proposed methods in GTX580.

of up to 53% over HO, but it does not improve CUDPP. The use of warp instructions here involves low latency accesses and avoids synchronization, using shared memory only for sharing partial sums among warps. With $N_{tb} = 128$, $r = 2$ and $b = 1$, the maximum warp occupancy is achieved. However, the best results are obtained with LF-shuffle4 and DA proposals. LF-shuffle4 is up to 69% faster than HO. Moreover, this proposal is 30% better than the CUDPP library for $N = 8388608$. This improvement is due to the use of shuffle instructions and the reduction of global memory transactions because of *float4* types and $b = 4$. Similar results are obtained for $r = 2$ and $r = 4$ in this algorithm. Finally, DA shows a 68% improvement over HO, and in the case of CUDPP up to 28%, combining $r = 4$ in first stage and $r = 2$ in the rest of stages with $b = 4$.

With reference to the optimization CUDA tecniques explained in the previous section, the efficiency of Mapped Memory technique depends on the number of elements to process in final recursive level. Empirically, the Mapped Memory technique presents bad performance in Kepler when the number of elements to process is greater than 300 owing to the performance of the PCI transactions. *Shuffle* instructions provide very good performance saving shared memory latency penalizations and synchronization barriers. All results in Table I were measured using the read-only data Kepler cache when passing read-only pointer arguments.

Figure 11 depicts a performance comparison between our best results and CUDPP library for GTX580 GPU. KS-p shows an improvement of 58 % over HO with $N = 8388608$, although it does not improve on the CUDPP library. HJ-MM produces a large percentage of bank conflicts, even so gives an improvement of up to 68% over HO and 4% in the case of CUDPP. Best results are also obtained with LF and DA proposals. In the case of LF, it shows an improvement of 75 % over HO and 25 % over CUDPP when $N = 8388608$. DA has the best performance in GTX580, up to 76 % over HO and 26 % over CUDPP.

To recap the experiments, the CUDPP library time was measured over 30 iterations, taking the initialization time into consideration only once. As this initialization time is high, our proposals achieve up to 40% of improvement in GTX580 and 48% in Tesla K20 when the experiments are repeated over only one iteration. Dividing this initialization time over 30 iterations, our proposals achieve up to 28% and 26% of improvement, respectively.

VI. Conclusions

The scan primivite is a highly important operation which participates in many applications. A GPU implementation of this operator in terms of efficency is not a simple task, since hardware capabilities have to be considered. In this paper we have presented several parallel scan proposals based on other reduction operators that are a good match for the scan primitive and to the GPU architecture. Furthermore, we have described the new Kepler architecture performance features. Based on these new features, several versions of each proposal were implemented, and all were analyzed in detail. New *shuffle* instructions have enhanced performance owing to their low latency and we highly recommend it for improving performance in intra-warp communication patterns. In addition, the new read-only data cache has also showed good performance in our tests, owing to its separate memory pipe and relaxed memory coalescing rules. In contrast, *CUDA Dynamic Paralelism* has shown poor performance in dynamic calls, using local memory as heap and long latency kernel invocations.

Finally, the performance of the resulting proposals was compared to other well-known scan implementations, such as the Harris-Owens method and the CUDPP library in two GPU architectures. With regard to Harris-Owens, improvements of up to 69% were achieved in Tesla K20 GPU and up to 74% in GTX580. For the CUDPP library, improvements of up to 48% and 40% (Tesla K20 and GTX580 respectively) were obtained for large arrays when the CUDPP library is measured over one iteration due to its high initialization time, and an overall improvement of performance of 28% and 26%, respectively.

Acknowledgments

This research has been supported by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Reference Groups, cofunded by FEDER funds of the EU (Ref. GRC2013/055); and by the Ministry of Economy and Competitiviness of Spain and FEDER funds of the EU (Project TIN2013-42148-P)

References

[1] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe, "GPU-efficient recursive filtering and summed-area tables," *ACM Transactions on Graphics (Proceedings of the ACM SIGGRAPH Asia 2011)*, vol. 30, no. 6, p. 176, 2011.

[2] S. Sengupta, A. E. Lefohn, and J. D. Owens, "A work-efficient step-efficient prefix sum algorithm," *Workshop on Edge Computing Using New Commodity Architectures*, 2006.

[3] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Aug. 2007.

[4] R. Pietersz and M. Regenmorte, "Bridging brownian libor," *Wilmott Magazine*, vol. 18, pp. 98–103, 2005.

[5] Ö. Eğecioğlu, E. Gallopoulos, and Ç. Koç, *A Parallel Method for Fast and Practical High-order Newton Interpolation*, ser. Center for Supercomputing Research and Development Urbana, Ill: CSRD report. University of Illinois at Urbana-Champaign, 1989.

[6] H. S. Warren, *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] J. Sklansky, "Conditional sum addition logic," *IRE Transactions on Electronic Computers*, 1960.

[8] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. 22, no. 8, pp. 786–793, Aug. 1973.

[9] D. Horn, *Stream Reduction Operations for GPGPU Applications in GPU Gems 2*. Addison-Wesley, 2005.

[10] Hensley, Justin, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra, "Fast summed-area table generation and its applications." *Computer Graphics Forum*, vol. 24, no. 3, pp. 547–555, 2005.

[11] D. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for gpus," *Technical Report*, Dec. 2008.

[12] N. Comp. (2014) Cudpp: Cuda data parallel primitives library. [Online]. Available: http://cudpp.github.io/

[13] S.-W. Ha and T.-D. Han, "A scalable work-efficient and depth-optimal parallel scan for the gpgpu environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2324–2333, 2013.

[14] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, June 2011.

[15] R. Brent and H. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. 31, no. 3, pp. 260–264, 1982.

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

[17] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*. ACM, 2008, pp. 205–213.

[18] D.Merrill and A. Grimshaw, "Parallel scan for stream architectures," in *Technical report*. Dept. of Computer Science, Univ. of Virginia, Dec. 2009.

[19] T. Han and D. Carlson, "Fast area-efficient vlsi adders," *Proc. Eighth Ann. Sym. Computer Arithmetic*, pp. 49–56, 1987.

[20] R. Hockney and C. Jesshope, *Parallel Computers 2: Architecture, Programming and Algorithms*, ser. Electronic computers. Taylor & Francis, 1988, no. v. 2.

[21] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.