Efficient Solving of Scan Primitive on Multi-GPU Systems

Adrián P. Diéguez, Margarita Amor, Ramón Doallo
University of A Coruña
A Coruña, Spain
Email:{*adrian.perez.dieguez, margamor, doallo*}*@udc.es*

Akira Nukada, Satoshi Matsuoka
Tokyo Institute of Technology
Tokyo, Japan
Email: nukada@smg.is.titech.ac.jp, matsu@is.titech.ac.jp

***Abstract*—GPUs fulfill high computation demands, but it is necessary to develop code carefully, selecting algorithms well suited to the GPU architecture and applying different optimizations. This article presents a GPU-suitable algorithm and a tuning strategy for performing the scan primitive over large problem sizes in CUDA. This tuning strategy defines different performance premises to find the GPU execution parameters that maximize performance. Taking these premises into consideration, we easily develop the kernels using CUDA skeletons to ensure efficiency and portability. Based on this, we describe an optimal proposal analyzed over different multiple GPU environments, the first multiple-GPU batch scan proposal to the best of our knowledge. The resulting implementations outperform other well-known libraries in most cases, such as CUDPP, ModernGPU, Thrust, CUB and LightScan.**

1. Introduction

GPUs have played a huge role in high performance computing in recent years. Nevertheless, programmers need to develop suitable parallel algorithms and consider optimization techniques to achieve the desired performance.

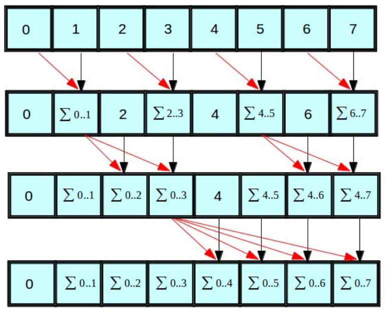


Figure 1: LF scan primitive for addition with N=8 elements.

The scan primitive, also known as prefix sum, is an operation that seems inherently sequential but for which there are many efficient parallel algorithms. Given a sequence of numbers and an operator, the primitive returns a sequence of the same size, where the i-element is the result of applying the operator from element 0 to element $i-1$, in the case of exclusive scan, or from element 0 to element i, in the case of inclusive scan. Figure 1 shows the inclusive scan primitive over an array of 8 elements.

The motivation of this work comes from the fact that the scan operator is widely used in different scientific disciplines [21] [23] and is the building block of different application. Nowadays, the success of Big Data depends on many of these applications and requires to use this primitive for large data sets, thus it is crucial to efficiently implement it for large problem sizes in a multiple GPU environment. Most of the scan implementations on the GPU were designed ad-hoc or for small-medium problem sizes. However, efficient solutions of large problem sizes are still an important challenge. Additionally, there are many cases where an application solves many instances of the same problem simultaneously [4], but there are not many implementations that solve a batch approach in a single invocation. The goal of this work is to provide a multiple-GPU library which obtains better performance thanks to use a tuning strategy for Multi-GPU and Multi-Node systems, as well as solving large data sets that cannot be solved in a single GPU.

In this work, an efficient multiple-GPU batch scan library is proposed. Firstly, a GPU-suitable algorithm is chosen and a tuning strategy defines a set of performance premises that aim at obtaining different GPU performance values which maximize the execution throughput. Then, CUDA kernels are implemented for the selected algorithm as parametrisable skeletons which receive the performance parameters values achieved from premises. Additionally, the execution is adapted to different environments: a computing node comprising several GPUs or several multiple-GPU nodes. Finally, the effectiveness of the strategy is empirically demonstrated, surpassing the state-of-the-art libraries.

1.1. Related Work

The scan parallel implementation was originally proposed for VLSI adders [12] [22]. The first GPU implementation was presented in [10]. In [9], the previous implementation was improved by pruning unnecessary work and then, the authors of [20] developed a simpler implementation. Other implementations can be found in [6] [7] [8] [25]. The Ladner-Fischer (LF) pattern also matches very well to GPU architectures [3], surpassing any other implementations. Currently the libraries that offer a high-optimized scan implementation are CUDPP [1], LightScan

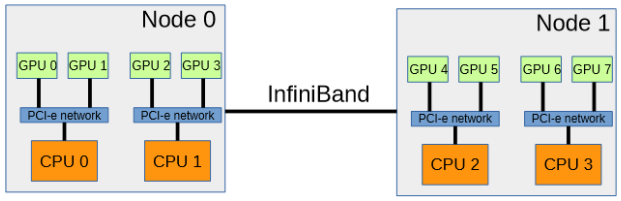


Figure 2: Multi-GPU topology within a Multi-Node environment.

[13], Thrust [15], CUB [16] and ModernGPU [19]. It should be also noted that CUB already runs at nearly the maximum theoretical rate for a single GPU [14]. In fact, all state-of-the-art solutions are based on a single GPU implementation. There is a parallel multiple-GPU library from NVIDIA, NCCL [17], although it has no implementation for scan.

2. Multi-GPU Resources Utilization

A *Multi-GPU* environment represents a single computing node composed of several GPUs, whereas if the system consists of several of these nodes connected through a low-latency bus, it is called a *Multi-Node* environment. When GPUs are arranged in several nodes (for instance communication between GPU 0 and 4 in Figure 2), a Multi-Node communication is required.

CUDA presents a number of features to facilitate Multi-GPU programming. *Unified Virtual Addressing* (UVA) provides one virtual address space for all CPU and GPU memories. Additionally, kernels executed under 64-bit applications on modern devices can directly access the global memory of any GPU connected to the same PCIe network using the CUDA peer-to-peer (P2P) API and avoiding communication via the host. Hence, data are copied between these devices asynchronously along the shortest PCI-e path, enabling communication-computation overlapping. A more detailed description of *NVIDIA's* architecture can be found in [11].

Using MPI, the contents of host memory can be transmitted directly by MPI functions. Instead of copying data from the device memory to the host buffers, and then calling the MPI API, it is possible to combine MPI and CUDA, sending data directly to the GPU buffers. This CUDA support is called *CUDA-Aware MPI*, enabling direct MPI communication between GPU global memories. Besides, *RDMA - GPU Direct* technology enables low-latency transfers over Infiniband between GPUs in different nodes using standard PCI-e adapters without host processor involvement, reducing CPU overhead and communication latency.

Specifically, the performance results of our tuning strategy were taken from a Multi-Node environment with 2 PCI-e networks per node, and 4 GPUs in each network (i.e, 8 GPUs per node). Table 1 describes this test platform, whereas performance results are discussed in Section 5.

	TSUBAME KFC Node
CPU	Xeon E5-2620 v2 (2.10 GHz, 6 cores) x2
Memory	64 GB
GPU	*4x Nvidia Tesla K80 (8 GPUs), 2 PCI-e networks*
Driver	375.51, SDK 8.0
Inter-node connection	InfiniBand FDR

TABLE 1: Description of a computing node in the test platform

Problem Parameters

$N = 2^n$	Problem size.
$G = 2^g$	Number of problems being solved simulteneously.

GPU Performance Parameters

$S = 2^s$	Number of shared-memory elements per block.
$P = 2^p$	Number of elements stored in registers per thread.
$B = 2^b$	Number of thread blocks executed per GPU, where $B = B_x \cdot B_y$.
$L = 2^l$	Number of threads that compose a block, where $L = L_x \cdot L_y$ and $S \leq P \cdot L$
K	Number of iterations per block in the cascade approach implementation. This determines the chunk size ($K \cdot P \cdot L_x$) when partitioning the problem into portions.

Node Performance Parameters

$Y = 2^y$	Number of PCI-e networks employed per node.
$V = 2^v$	Number of GPUs being executed within the same PCI-e network.
$W = 2^w$	Number of GPUs used per node, where $W = Y \cdot V$
$M = 2^m$	Number of nodes.

TABLE 2: Description of tuning strategy parameters.

2.1. Tuning Parameters

The success of our tuning strategy resides in representing the problem features in terms of GPU resources, that can be modeled as parameters whose performance should be maximized. All the parameters defined here are collected in Table 2. As it can be observed in the table, they are grouped into three categories: problem parameters given by application, performance parameters for GPU implementation and performance parameters related with hardware node configuration, respectively.

Our proposals consider the case of simultaneously executing $G = 2^g$ problems of N elements each, where $N = 2^n$. This *batch* computation is performed in a single invocation to the library, and the use of powers of two for defining G and N is merely to simplify the explanation here, since any other power base could be used. In any case, the problems are solved along n computational steps.

Problems are solved using $B = 2^b$ thread blocks per kernel, which can be scheduled into a two-dimensional distribution $B = B_x \cdot B_y$, ($b = b_x + b_y$). In our strategy, B_x represents the number of blocks used for computing each problem, whereas B_y represents the number of problems being simultaneously executed on that kernel. Each thread block comprises $L = 2^l$ threads, which can be distributed as $L = L_x \cdot L_y$, ($l = l_x + l_y$). L_x represents the number of threads per block working on the same problem, whereas L_y the number of problems being solved in that block. Each block has assigned an amount of shared memory that allows to exchange data among threads of the same block. Threads have a certain amount of private registers associated. Thus, each thread works with $P = 2^p$ elements of the problem in private registers, whereas all threads into a thread block can access to $S = 2^s$ elements in shared memory. There are cases where the data stored in registers have no copy in shared memory. For example, when using *shuffle* instructions (intra-warp communication via registers), shared memory is not needed for exchanging information inside a warp. In that case, shared memory is only used for exchanging data among different warps; thus, $s \leq p + l$. When working with several kernels, all previously defined parameters use a superscript number to identify the referred kernel. For example, B_x value of kernel 1 is represented by B_x^1, whereas B_x value of kernel 2, by B_x^2. In Section 3.1, K parameter will be deeply explained.

A typical NVIDIA Multi-Node environment is shown in Figure 2, where each computing node has several GPUs. The same node is composed of 4 GPUs grouped into two PCI-e networks. Each PCI-e network contains a CPU with two GPUs. The number of GPUs inside one computing node that are being used in the execution is represented by $W = 2^w$. In addition to this, $Y = 2^y$ is the number of PCI-e networks being employed in each computing node; whereas $V = 2^v$ shows the number of used GPUs connected to the same PCI-e network. Finally, $M = 2^m$ is the number of computing nodes being used. The first three parameters can be related as $w = y + v$. For example, an execution over Node 0 in figure, which uses the 4 GPUs available, implies $W = 4$, $Y = 2$, $V = 2$ and $M = 1$. Using only the GPU 0 and GPU 2 would involve $W = 2$, $Y = 2$, $V = 1$ and $M = 1$. In a Multi-Node configuration, $M = 2$ when using Node 0 and Node 1 with $W = 4$, $V = 2$ and $Y = 2$. It should be noted that W, Y, V and M values are defined by the tuning strategy, and are limited by the hardware distribution.

3. A Tuning Strategy for Scan Primitive Problems

Our tuning strategy is based on a set of premises that aim at obtaining different GPU performance parameters which maximize the execution throughput. The strategy is focused on solving large-size problems, partitioning the problem among several thread blocks. In this work, the scan primitive is optimized with this strategy and the employed algorithm is shown in Figure 1 with the addition operator. The pattern of the algorithm is called Ladner-Fischer pattern (LF) [18], and it has been chosen since matches very well to GPU architectures [3]. Henceforth, the addition operation is used in the scan primitive by default.

The exchange of information between blocks is performed through global memory. Blocks write their information in global memory, and after using any global synchronization mechanism, the remaining blocks will be able to read this information from global memory. This approach is highly efficient if data exchanges are properly optimized and the workload is properly balanced among the GPU resources.

3.1. Implementation Details

Based on current large-size scan problems solvers, data are divided into several data blocks. The reduction value of each data block is computed, stored in an auxiliary array

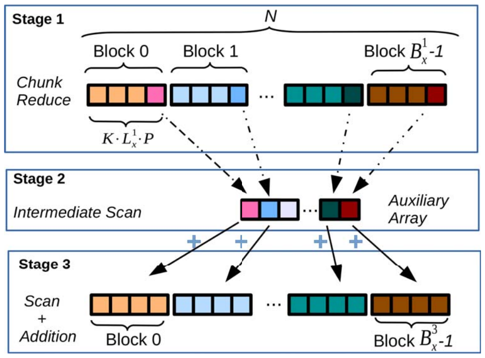


Figure 3: Three kernel execution for the scan primitive when $G = 1$ problems.

and then all elements in the auxiliary array are scanned. Data blocks compute then their local scan and add the corresponding value from the auxiliary array to their elements, completing the overall scan. This procedure is summarized in Figure 3.

From the CUDA perspective, the execution is divided into three stages (kernels). As it can be seen in Figure 3, firstly, the N elements are divided into chunks, where each block, represented by a color, processes one chunk. Thus, B_x^1 blocks work on the same problem, and $B_y^1 = G$ problems are being solved simultaneously. The first stage (*Chunk Reduce*) computes the reduction for each chunk (light colour square in figure). Note that reduction primitive means writing the cumulative sum for all elements into the last element, whereas the remaining elements are not modified. The result of each reduction is stored in an auxiliary array in global memory. Taking into account the fact that this is a memory-bound problem in current GPU architectures, storing one element per chunk and computing the scan later again is preferable to writing all elements in global memory twice. As each chunk writes its reduction in the auxiliary array, there are $G \cdot B_x^1$ elements in this array. Thus, a second stage (*Intermediate Scan*) computes the scan of these B_x^1 values for each problem using the LF scan pattern, explained above, in a single thread block. Finally, a third stage (*Scan+Addition*) performs the local scan for each chunk, adding the corresponding element from the global memory's auxiliary array, processed in the previous kernel, to all elements in its chunk. Note that the number of elements per problem processed in Stage 1 and Stage 3 is N, whereas it is B_x^1 for Stage 2. As Stage 1 and Stage 3 input sizes are equal and these stages share a very similar computational core, each problem is partitioned into the same number of chunks and both stages use the same amount of SM resources. Thus, $B_x^1 = B_x^3 = B_x^{1,3}$.

In our strategy, each thread reads P elements from global memory using the *int4* customized data type, facilitating coalescence and reducing memory transactions. These 4-elements are computed by each thread in registers, as the top of Figure 4 shows for $L_x = 4$ and $P = 4$. For example, if P is equal to 8, then two loads from global memory are performed by each thread and two 4-elements scans are computed.

The L_x threads are grouped into warps, therefore each warp computes $P \cdot warpSize$ elements ($warpSize = 32$ currently, although $warpSize = 4$ in Figure 4 for clarity). Hence, after the initial scan of P elements in a single step (red values in the figure), each warp computes $warpSize$ elements using *shuffle* instructions and the Ladner-Fischer access pattern. Once the shuffle-scan is performed, each thread adds the corresponding value to its 4-elements. Using the exclusive scan saves an extra communication step, otherwise each thread would have to send the inclusive result to its neighbor, instead of directly adding the value stored in its register. Computing the exclusive scan is fast, the initial value is subtracted from the scanned value. Finally, the last element of the $P \cdot warpSize$ data sequence is stored in shared memory in order to share this partial sum with other warps. Hence, shared memory has as many elements as warps, at most 32 in current architectures. A single warp will repeat this process over the 32 partial sums stored in shared memory in order to build the final result of the $P \cdot L_x$ elements. Note that, thanks to use shuffle instructions, $S \leq 32$ ($s \leq 5$).

In addition to this computational flow, this implementation also follows a cascade approach [7]. Each block executes K iterations, where each iteration computes $L_x \cdot P$ elements as explained in the previous paragraph. Thus, each block computes $K \cdot L_x \cdot P$ elements; i.e., the chunk size is equal to $K \cdot L_x \cdot P$ elements. Once one iteration has computed $L_x \cdot P$ elements, the last one is passed to the next iteration, adding this value to all $L_x \cdot P$ elements of that iteration. Figure 5 shows this approach, which avoids launching an excessive number of blocks, and allows thread information to be reused, generating fewer instructions and also using fewer temporal values. After K iterations, the scan of $K \cdot L_x \cdot P$ elements has been computed. In the case of Stage 1 (Chunk Reduction), the last element is written in the auxiliary array to be passed to Stage 2. As the chunk size is a power of two, K is also a power of two.

As already mentioned, there are G problems being simultaneously solved in each kernel. In Stage 1 (Chunk Reduction) and Stage 3 (Scan+Addition), all threads in a block work on the same chunk, i.e. on the same problem ($L_y^{1,3} = 1$), thus $L^{1,3} = L_x^{1,3} \cdot 1$. In Stage 1, each chunk writes its reduction in an auxiliary array, thus the number of chunks sets up the number of elements per problem to be processed in Stage 2, $B_x^{1,3}$. In the Intermediate Scan kernel, the same block must process elements from different problems, otherwise warp occupancy would be much too low, as Stage 2 executes much fewer elements. Therefore, all elements which come from the same problem have the same L_y^2 identifier, so $L_y^2 > 1$, $B_y^2 = G/L_y^2$ and $B_x^2 = 1$ in Stage 2.

All these operations are efficiently implemented using BPLG CUDA skeletons [2], which are carefully designed to attain high levels of efficiency in CUDA architectures. They are designed with templates, enabling the generation, at compile time, of tuned kernels according to the more suitable (s, p, l, K) tuple for the specific GPU architecture in which they are to be executed. The compile-time generation allows the use of generic programming and template metaprogramming, reducing code complexity, avoiding temporal registers for function calls and taking advantage of fully unrolling static loops and avoiding the dynamic addressing of register arrays.

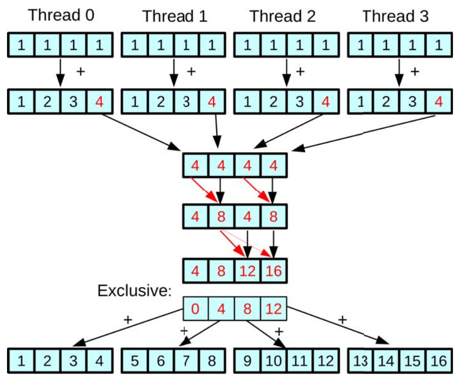


Figure 4: Scan computation in one warp, considering *warpSize=4*, $P = 4$ and $L_x = 4$.

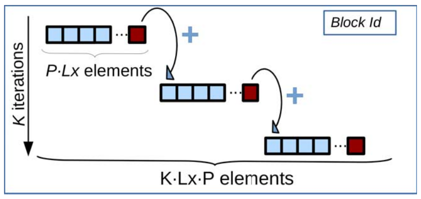


Figure 5: Cascade approach computation.

3.2. Premises for Performance Maximization

The values of (s, p, l, K) parameters determine the performance of the execution in a GPU. A set of premises are defined in our strategy in order to find their appropriate values which maximize execution performance:

Premise 1. *Balancing warp and block parallelism*. Higher parallelism obtains better performance since it hides latency from functional units and the access to memory. However, this parallelism is limited by the amount of common resources shared in an SM. More threads per block implies less resources in each block. In the GPU, the level of parallelism can be supported in terms of the number of blocks per *SM* (*SM* block parallelism), or the number of warps per *SM* (*SM* warp parallelism), so our strategy attempts to strike a balance between the maximization of both:

1. *The maximization of SM block parallelism in each stage* in order to keep processing the maximum amount of simultaneous blocks per *SM* (16 in the case of *Kepler* and 32 in the case of *Maxwell*-based *GPUs*). Factors that limit SM block parallelism are the number of registers used by each thread and the amount of shared memory required by a block. In fact, the *GPU* hardware is able to

Warps per block	Regs per thread	Shared mem per block	SM warp occupancy	SM number of blocks
1	256	7168	25%	16
2	128	7168	50%	16
4	**64**	**7168**	**100%**	**16**
8	64	14336	100%	8
16	64	28672	100%	4
32	64	49152	100%	2

TABLE 3: Performance parameters per SM on Kepler Platforms with compute capability 3.7

provide highly satisfactory performance even at lower warp occupancy (small *SM* warp parallelism) [24] [5].
2. *The maximization of SM warp parallelism in each stage*. This premise is focused on increasing the number of warps per *SM*, allowing the SM to hide latency among warps when one stalls.

In order to balance warp and block parallelism, it is necessary to limit the factors that decrease the SM parallelism, such as the number of registers used by each thread or the amount of shared memory required per block. Table 3 summarizes different GPU performance configurations in order to maximize SM warp and block parallelism in Kepler architectures with compute capabilities 3.7. It is easy to see that increasing warp parallelism reduces block parallelism, and vice versa. However, there is a configuration, marked as a bold row in the table, that maximizes both types of parallelism: when working with 4 warps, less than 7168 shared memory bytes and less than 64 registers per thread.

Premise 2. *Increase the computational load per thread*. The number of elements processed by each thread, P, influences the number of computing steps per stage and the number of threads that process a problem. A larger P delivers higher performance, as there are fewer shuffle exchanges and more elements are processed in each iteration. Nevertheless, the increase of P may also require too many registers per thread, reducing the block parallelism or generating memory spilling (high-latency memory usage when there are no registers available).

Premise 3. *Maximization of SM occupancy and minimization of global memory communications*. This strategy involves the invocation of 3 kernels. The number of elements in each kernel depends on the previous/next one, thus it is important to find a performance trade-off among them that maximizes global throughput. In this approach, this can be controlled by the K parameter. The number of elements per problem to be processed in Stage 2 is determined by B_x^1, which is the same as the number of chunks, $B_x^1 = \frac{N}{K^1 \cdot L_x^1 \cdot P^1}$, being L_x^1 and P^1 constant values. On the one hand, K^1 must be small in order to have a large number of elements in Stage 2 and exploit GPU parallelism. On the other hand, K^1 must be large in order to have fewer chunks and reduce the number of global memory transactions (reads and writes from/to global memory auxiliary array).

Since $B_x^1 = B_x^3$, and both Stage 1 and Stage 3 use the same amount of SM resources, $K^1 = K^3$. On the other hand, as the number of elements to be processed in Stage 2 is low, and in favor of exploiting the SM block parallelism for this Stage, $K^2 = 1$, increasing the number of blocks as much as possible in Stage 2. Therefore, it is necessary to calculate the optimal value of K^1, which depends on the total number of elements being processed, $N \cdot G$. To do so, our strategy considers that the total number of blocks processed in Stage 2 must be greater than the maximum number of blocks executed per SM; i.e., 16 for Kepler architecture. As the number of elements processed in Stage 2 is $[G \cdot \frac{N}{K^1 \cdot L_x^1 \cdot P^1}]$ and each block executes $P^2 \cdot L_x^2 \cdot L_y^2$ in Stage 2, then Premise 3 establishes $[G \cdot \frac{N}{K^1 \cdot L_x^1 \cdot P^1} / (P^2 \cdot L_x^2 \cdot L_y^2) \geq 16]$.

Note that $L = L_x \cdot L_y$, and $L_y^{1,3} = 1$. Then, K can be defined as:

$$1 \leq K^1 \leq \frac{G \cdot N}{16 \cdot P^1 \cdot P^2 \cdot L^1 \cdot L^2} \tag{1}$$

Thus, Equation 1 establishes the searching space for K^1 that looks for a trade-off between maximizing SM occupancy and minimizing global memory communications.

In summary, following Premise 1 and considering an architecture with compute capabilities 3.7, our kernels should use 128 threads (4 warps) per block ($l = 7$), and fewer than 7168 shared memory bytes per block ($s \leq 10$ in the case of integers) in order to achieve 16 active blocks and 100% of warp occupancy (bold row in Table 3). Note that, thanks to shuffle instructions, $s \leq 5$, as it has already been explained. On the other hand, according Premise 2, the value of p must be as high as possible, without surpassing 64 registers per thread. Considering integers, each element is stored in a single 32-bit register, thus $p \leq 6$. Following Premise 2, and also considering that auxiliary variables and index calculation consume many registers, $p = 3$ is defined. Premise 3 helps to establish the K value, although this parameter depends on the execution (G and N values) and is obtained following previous equations.

Premises 1, 2 and 3 determine the (s, p, l) performance parameters and trim the subspace to find the K parameter, creating the (s, p, l, K) tuple to be passed to the skeleton-based kernels in a single GPU environment. The optimal parameter K depends on the execution (G and N values) and other factors that are difficult to predict (such as the CUDA memory system management). Thus, once the (s, p, l) is determined using previous premises, all possible K values that meet Eq. 1 are tested. Currently, this search is not done automatically, but is part of the future work. In the next section, our strategy is analyzed for dealing with several GPUs, considering Premise 4 for that purpose. This new premise introduces new limits to the possible values of K for each (W, V, M) given configuration, reducing the search space. Thus, for each tuple (W, V, M) possible in the system, all K values from the corresponding search space are empirically tested, choosing the one which maximizes the global performance. It should be observed that K is a power of two, thus the number of suitable K values that meet the corresponding equations is not very huge and can be easily calculated. Additionally, we would like to point out that these premises are focused on this operation, but they can be easily extended to other algorithms.

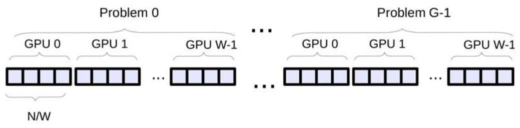


Figure 6: Multi-GPU Problem Scattering: Each problem is solved by W GPUs.

4. Tuning Scan Primitive for Multi-GPU and Multi-Node Systems

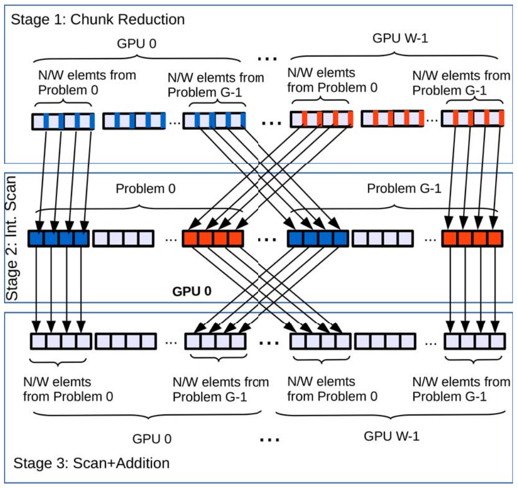
This section presents different approaches based on our tuning strategy, when working with several GPUs. The use of several GPUs can be motivated by two cases: (Case 1) Each problem can be perfectly stored in a single GPU memory but using each GPU to compute independently several problems may improve performance, and (Case 2) either the N elements of a single problem cannot be stored in a single GPU memory or performance can take advantage of distributing the same problem among several GPUs.

As described in Section 2, two kinds of environments using several GPUs are considered. A Multi-GPU environment, where multiple devices are placed in the same computing node, and a Multi-Node environment with multiple GPUs per computing node and several computing nodes connected by a low-latency bus. In the Multi-GPU environment, the communication through GPUs is performed by PCI-e networks inside the node. In the Multi-Node environment, MPI routines are employed for communicating nodes. Solving the Case 1 is trivial, simply executing the strategy analyzed in Section 3 through several GPUs, since there is no communication among GPUs. However, Case 2 requires collaboration among GPUs, and it is studied under the *Multi-GPU Problem Scattering* and *Multi-GPU Problem with Prioritized Communications* approaches.

4.1. Multi-GPU Problem Scattering (MPS)

This approach can handle Case 2 and is labeled as *Scan-MPS* proposal in Section 5. All GPUs participate in solving a portion of each problem, as Figure 6 shows for a Multi-GPU environment. Each problem is solved by W GPUs, where each GPU computes a portion of the problem ($\frac{N}{W}$ elements). If there are G problems being simultaneously solved, then each GPU works with G portions of N/W elements. In terms of performance, this approach is bounded by GPU-communication bandwidth in most cases.

Figure 7 shows the schema of this approach. In Stage 1 (Chunk Reduction), the N/W elements of each problem are divided into chunks of $K^1 \cdot L_x^1 \cdot P^1$ size for each GPU, as explained in Section 3, performing the chunk reduction. Note that problems are now divided among W GPUs, thus the number of chunks per problem in a Multi-GPU environment is $W \cdot B_x^1$, and each chunk is computed by one block. The resulting element of each chunk is stored in an auxiliary array of $G \cdot W \cdot B_x^1$ elements. Stage 2 (Intermediate Scan) performs the scan of these values. At this point, there


Stage 1: Chunk Reduction
GPU 0
GPU W-1
N/W elemts from Problem 0
N/W elemts from Problem G-1
N/W elemts from Problem 0
N/W elemts from Problem G-1
Stage 2: Int. Scan
Problem 0
Problem G-1
GPU 0
N/W elemts from Problem 0
N/W elemts from Problem G-1
N/W elemts from Problem 0
N/W elemts from Problem G-1
GPU 0
GPU W-1
Stage 3: Scan+Addition
Figure 7: Multi-GPU Problem Scattering on a Multi-GPU environment.

are two options: either a single GPU performs the auxiliary-array scan for all problems in its memory space, or several GPUs participate in this task, with each GPU performing the scan of a set of entire problems in its memory space. This depends on the node topology performance (if all GPUs are connected in the same PCI-e network or it is necessary to transfer through host memory), but empirically, executing this second kernel on a single GPU has better performance than splitting its execution into several GPUs. Finally, Stage 3 (Scan+Addition), which uses the same data distribution as Stage 1, performs the local scan over its chunks, also adding the corresponding elements from the auxiliary array to them.

In a Multi-Node environment, all GPUs participate in solving each problem, thus there is communication among $M \cdot W$ GPUs. As there are G problems simultaneously executed, each GPU works with G portions of $\frac{N}{M \cdot W}$ elements, and the number of chunks per problem is $M \cdot W \cdot B_x^1$. Basically, the same approach as that in Figure 7 is used. Firstly, each node divides these data among its W GPUs. One GPU in the system acts as a *master process* (GPU 0), allocating an additional array for processing the second stage on its device memory. After synchronizing all MPI processes, the first stage is executed. The goal of the first stage is to calculate the chunk reductions that are passed to the second stage. To do so, these values are collected from all GPUs by the master process with an *MPI_Gather* instruction. The master process computes the second stage in its memory and returns the resulting values to the corresponding GPUs through an *MPI_Scatter* instruction. Finally, each GPU executes the third stage, and the result is collected from GPUs to the node. Please, note that data transfers in the same node are performed using the MPI API, although if they are on the same PCI-e bus, peer-to-peer transfers are automatically used by the CUDA-aware MPI library.

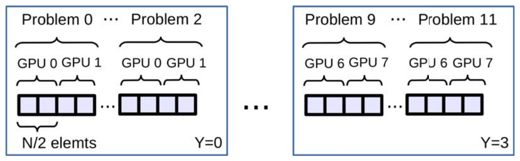


Figure 8: 12 problems being solved by 4 different PCI-e networks with 2 GPUs each.

4.1.1. Multi-GPU Problem with Prioritized Communications (MP-PC). The *Multi-GPU Problem with Prioritized Communications* proposal can be considered as a sub-case of the Multi-GPU Problem Scattering proposal, addressing Case 2, where the intra-node PCI-e network communications are prioritized. This approach is tagged as *Scan-MP-PC* proposal in the evaluation in Section 5, and Figure 8 shows this schema for a Multi-GPU environment.

This approach basically focuses on the *Multi-GPU Problem Scattering* approach but taking advantage of the PCI-e networks within the compute node. When two GPUs in the same node are not connected to the same PCI-e network, memory transfers are performed through host memory, losing a good deal of performance. In order to minimize this loss, the work is partitioned into the GPUs which belong to the same PCI-e network. Thus, V GPUs of each PCI-e network work on G/Y problems, partitioning each problem into V portions of size N/V. Communication is only performed among the V GPUs of the same PCI-e network node, whereas other PCI-e GPUs work on their problems, as it can be seen in Figure 8 for $V = 2$, $W = 8$, $Y = 4$ and $G = 12$. In terms of performance, this proposal improves on the *Problem Parallelism* proposal by avoiding memory copies through the host.

Regarding the Multi-Node version, each node solves several problems, and these problems are solved only by that node. Specifically, these problems are computed by V GPUs connected to the same PCI-e network. As there is no communication among nodes, just inside each node, this approach runs the same code as the Multi-GPU version, but being executed through several computing nodes. There is no MPI communication in this proposal.

4.2. Multi-GPU Performance Maximization

Using several GPUs introduces new variables to take into consideration when maximizing performance. Premise 4 takes these new variables into consideration:

Premise 4. *Prioritizing High-Bandwidth Communications.* Scan primitive scales very well when the number of GPUs rises, thus the number of participating GPUs should be as high as possible. However, it is necessary to pay attention to how these GPUs are connected, since communication latency should be reduced, as well as the amount of data to be transferred from/to each GPU (related with K^1 parameter). This premise defines the kind of communications that should be prioritized depending on the target environment.

It distinguishes between several scenarios depending on the problem size and the hardware distribution, as discussed in Section 5. On the one hand, if the participating GPUs in each problem belong to the same PCI-e network, the communication overhead is very low for both the Scan-MPS and the Scan-MP-PC proposals, since the computation is performed inside the same node and there is no communication among nodes. Hence, W, V, M must be maximized as much as the hardware allows. On the other hand, when the participating GPUs do not belong to the same PCI-e network, the computation can be distributed either along several PCI-e networks inside the same node, communicating via host memory and CUDA API, or across several nodes via InfiniBand and MPI. Empirically, if the amount of data is low, the communication via host memory performs better than via MPI, as MPI introduces a considerable overhead. Therefore, W, V must be maximized and M minimized in this case. Nevertheless, the computation of a huge amount of data performs better through several nodes via MPI - RDMA, since the MPI latency remains constant. Thus, W and M must be maximized.

Irrespective of the approach employed, (s, p, l) parameters remain constant from the analysis of Section 3, since they maximize performance in each GPU. Nevertheless, K^1 may vary slightly because it has an indirect bearing on the performance of the other GPUs. Premise 3 justifies the fact of maximizing K^1 with Equation 1. With several GPUs, a large K^1 will generate a low number of chunks and it implies fewer elements to be written in GPU 0 global memory from other GPUs. Equation 1 from Premise 3 is extended with the following equations when working in Multi-GPU or Multi-Node environments:

$$\frac{N}{K^1 \cdot L_x^1 \cdot P^1} \geq M \cdot W \tag{2}$$

$$\frac{N}{K^1 \cdot L_x^1 \cdot P^1} \geq V \tag{3}$$

The number of chunks, generated by splitting N into portions of $K^1 \cdot L_x^1 \cdot P^1$ elements, must be equal or greater than the number of GPUs employed in the case of MPS and MP-PC proposals, using Equation 2 and Equation 3, respectively, in order to ensure each GPU processes at least one chunk.

5. Performance Evaluation

In this section, our tuning strategy's performance is compared with state-of-the-art libraries, such as ModernGPU [19], Thrust [15], LightScan [13] and CUB [16]. The performance results were taken on the platform described in Table 1. All data elements are integers, and they were in GPUs memory prior to the GPU execution. Regarding the number of problems and their size, $N \leq 268,435,456$ ($n \leq 28$) is established. In all cases, the K^1 parameter for the given (W, V, M) configuration is set with the value which maximizes performance. This is obtained empirically for each problem size from the search space proposed in premises, whereas the employed (s, p, l) parameters are the ones obtained in Section 3.2.

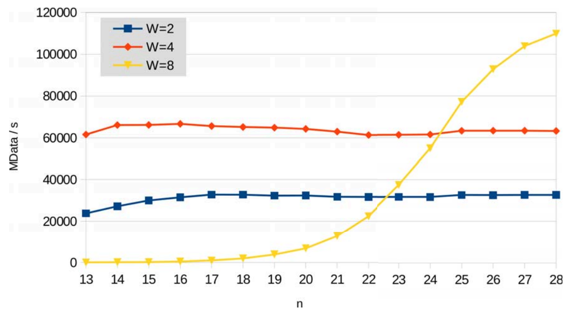


Figure 9: Performance analysis for the Multi-GPU Problem Scattering approach (Scan-MPS proposal) where $G = 2^{28}/N$.

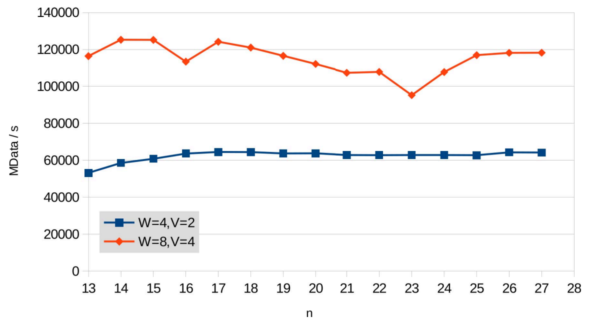


Figure 10: Performance analysis for the Multi-GPU Problem with Prioritized Communications approach (Scan-MP-PC proposal) where $G = 2^{28}/N$.

5.1. Multi-GPU Environment

Performance results for the *Multi-GPU Problem Scattering* approach (*Scan-MPS*) are considered in Figure 9 where 2^{28} data are solved, divided into $G = 2^{28}/N$ batches for each $N = 2^n$ problem. It should be noted that this platform has 2 PCI-e networks, each one with 4 GPUs; thus W can be configured as $1 \leq W \leq 8$, as well as $V \leq 4$ and $Y \leq 2$. According to Premise 4, if $W \leq 4$, then *V=W* in this case, since the throughput would scale along GPUs (*W=2,4*) due to the absence of host memory communications. However, when *W=8*, host memory transactions are used, as the node configuration only allows 4 GPUs connected to the same PCI-e. This is the reason why performance drops so markedly when *W=8*: there are G problems being executed simultaneously where each auxiliary array is written by 8 GPUs through host memory. As fast as N grows and G decreases, the number of auxiliary arrays being written is also reduced, raising performance. This analysis also shows that the GPU communication penalty is very low with P2P, demonstrating that the *Multi-GPU Problem Scattering* strategy works well when N cannot be stored in a single GPU and P2P API can be used.

Figure 10 depicts the *Multi-GPU Problem with Prioritized Communications* approach (*Scan-MP-PC*) with $G = 2^{28}/N$. There are communications among GPUs but performed with the P2P API, as each problem can be stored in

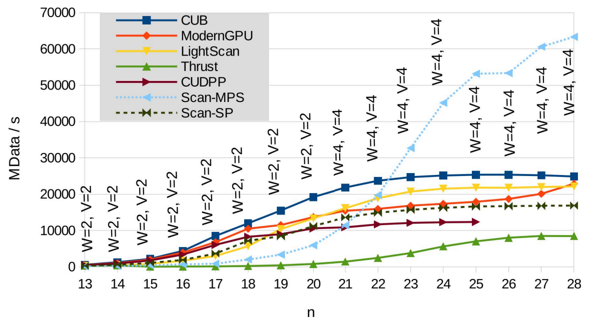


Figure 11: Performance analysis for our best Multi-GPU proposal when $G = 1$.

V GPUs of the same PCI-e network. Note that having *Y=2* in each node does not make sense with only *W=2* GPUs: if they are placed in different PCI-e networks, it represents the trivial case where no communication among GPUs is involved; otherwise, if they are connected to the same PCI-e network, it is the case of *Scan-MPS* proposal. Each node has 2 PCI-e networks with 4 GPUs connected to each network; thus we propose *W=4* and *V=2* for one test, and *W=8*, and *V=4* for a second test. As each problem is solved by V GPUs, when the number of problems, G, is lower than the number of PCI-e networks, Y, the number of PCI-e being used has to be reduced. In Figure 10, *n=28* is not shown since it is solved by a single PCI-e network.

Figure 11 depicts a performance comparison with respect to state-of-the-art libraries, where the number of problems solved is *G=1*. Our strategy relies on a massive parallelism for exploiting GPU SMs, therefore our strategy performance is not very impressive if the total number of elements being simultaneously executed is low, *G=1* in this case. It should be noted that having only 1 batch, the *Scan-MP-PC* proposal is executed on a *V=1* PCI-e network, which is the same as executing the *Scan-MPS* proposal. In this case, GPU computational power is underused, especially for Stage 2. Nonetheless, our proposal is still very competitive, being on average (averaging the speedup obtained for each data point) 1.21x faster than CUDPP, 7.8x against Thrust, 1.31x against ModernGPU, 1.31x with respect to LightScan and 1.04x against CUB. Please observe that each N is solved with the $(W, V) > 1$ parameters (attached in Figure 11) which achieve the best performance. Multi-GPU proposals cannot be competitive for small problem sizes when *G=1*, since the computation time is lower than the communication latency among GPUs. Our proposal running in a single GPU, called Scan Single-GPU Problem *Scan-SP*, is also shown in figure in order to compare the performance with the other libraries. As said, our proposal is focused on solving simultaneously many problems, thus the performance for the case of *G=1* is not very outstanding.

Figure 12 shows the performance achieved with *Scan-MP-PC*, our best proposal for $G = 2^{28}/N$ batches for each N value, as well as *Scan-SP*. A *Log10* performance scale has been adopted for readability. Although the most representative scenario of our proposal lies in solving several

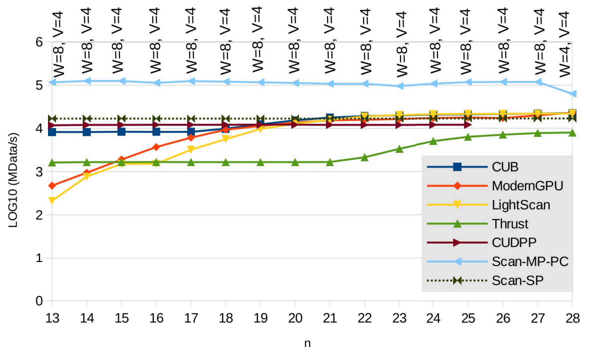


Figure 12: Performance analysis for our best Multi-GPU proposal when $G = 2^{28}/N$ problems.

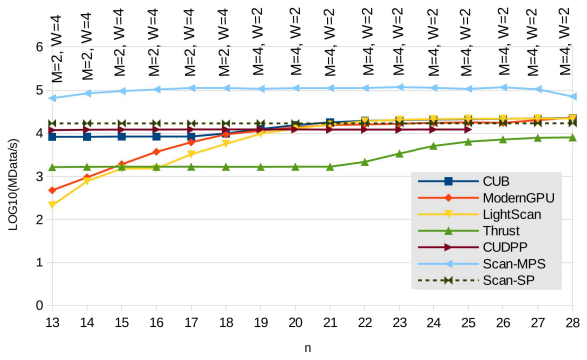


Figure 13: Performance analysis for our best Multi-Node proposal for $G = 2^{28}/N$ problems.

batches simultaneously, only CUDPP supports this feature with its *multiScan* function. Thrust provides a segmented operation, but it forces to carry an additional flag array, reducing performance. Also, a segmented scan can be implemented with CUB following [20], modifying the datatype and extending the sum operator with an additional condition. However, better performance has been obtained invoking the non-segmented function G times for *n¿21* in the case of Thrust, and *n¿17* in CUB. For fairness, we use the option that achieves the best performance for each data point. In the case of ModernGPU and LightScan libraries, the corresponding function is also invoked G times. All competing libraries are executing in a single GPU, since none of them provides a Multi-GPU support. Our Multi-GPU proposal is on average 9.48x faster than CUDPP, 49.81x against Thrust, 33.77x with respect to ModernGPU, 8.92x faster than CUB and 58.44x against LightScan under such scenario. It can be observed how performance increases in Thrust, ModernGPU, CUB and LightScan libraries in line with the rise in N (increasing N implies lower G, reducing the number of invocations). Specifically, when *G=32768* problems with *n=13*, our proposal is 245.54x times faster with respect to ModernGPU, 71.36x faster than Thrust, 14.28x against CUB and 549.79x with respect to LightScan. However, when *G=8* and *n=25*, this speedup is decreased to 6.59x for ModernGPU, 18.5x for Thrust, 5.55x for CUB and 5.44x for LightScan. Please, note that performance drops when *n=28*, as *G=1* and only one PCI-e network is used.

5.2. Multi-Node Environment

The performance when involving several computing nodes, where there is no communication among them, can be easily predicted. However in this environment, the *Multi-GPU Problem Scattering* proposal performs inter-node communications through MPI instructions, adding extra complexity to our model as well as new latency that affect to global performance. OpenMPI 1.8.5 with CUDA-aware and RDMA support are employed, and GPUs are connected through the same PCI-e network inside the computing node.

Different combinations of M and W can be used to compute the scan in a Multi-Node environment. Depending on the amount of data to be processed, the correct choice is key to obtaining the maximum performance. For example, in the case of using 8 GPUs in total, there are several M, W possible combinations ($M \times W = 8$). In our experiments, the best performance is achieved with $M = 2, W = 4$, obtaining the same performance results as $M = 4, W = 2$ at high N sizes, whereas $M = 8, W = 1$ obtains the worst results. This is due to the fact that MPI communications introduce an additional overhead in execution, thus the strategy would be to minimize the number of computing nodes as far as possible, maximizing the use of GPUs connected to the same PCI-e network in each node. However, as soon as the amount of data grows, the performance difference among different combinations is reduced. In the case of 2^{13} elements being solved per problem, the configuration $M = 2, W = 4$ is 1.48x faster than the configuration $M = 8, W = 1$, whereas in the case of 2^{28} elements, this speed-up is only 1.03x. This is due to the fact that, empirically, the MPI overhead is almost constant in spite of the amount of data, while GPU computation time is proportional to data size. It should also be noted that K^1 is a factor that has a bearing on global performance and must be small enough to have at least as many chunks as GPUs, $\frac{N}{K^1 \cdot L_x^1 \cdot P^1} \geq M \cdot W$.

Figure 13 depicts a performance study of our best Multi-Node proposal in comparison to state-of-the-art libraries for the *Multi-GPU Problem Scattering* approach, outperforming all of them. On average, it is 8.51x faster than CUDPP, 43.82x against Thrust, 24.85x in comparison to ModernGPU, 7.7x with respect to CUB and 41.2x for LightScan, when simultaneously solving $G = 2^{28}/N$ problems. In the case of low N, these speedups are greater for the libraries with no batch support: 50.37x for Thrust, 88.31x for ModernGPU, 10.13x for CUB and 109.12x for LightScan in the case of $n = 14$. However, they are smaller for high N values, since the number of memory transactions decreases with G: 8.85x for Thrust, 3.1x for ModernGPU, 3.13x for CUB and 3.22x for LightScan in the case of $n = 28$.

Finally, Figure 14 shows a breakdown of times spent on each problem size for $M = 2$ computing nodes of $W = 4$ GPUs each, executing 2^{28} elements split into $G = 2^{28}/N$ batches for each problem size. Since MPI communications

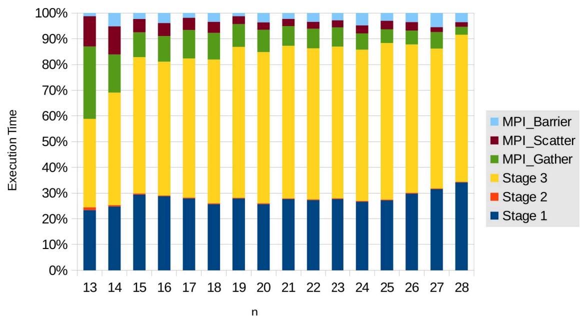


Figure 14: Breakdown of times spent on M=2 and W=4 for $G = 2^{28}/N$ problems.

are introduced in the execution, there is an additional overhead which remains almost constant independently of the amount of data to be processed. MPI barriers sometimes increase their time, as they are blocking collectives; thus the time of the collective in each MPI process also depends on how long the process has waited for the others. Note that the time spent on *MPI_Gather* and *MPI_Scatter* collectives is reduced when G is also decreased, since the number of elements to be processed by Stage 2 is also lessened and the MPI collectives work with fewer elements.

6. Conclusions

We develop a multiple-GPU tuning strategy to compute a batch scan operator for large-size problems on several GPUs. In our strategy, different GPU performance parameters are obtained according to a set of premises. Based on this strategy, an optimal proposal is studied over different NVIDIA environments: a Multi-GPU environment in a single computing node or a Multi-Node environment with several GPUs and computing nodes.

After analyzing performance results, we can conclude that our strategy efficiently solves the scan primitive in different NVIDIA environments, surpassing previous state-of-the-art libraries. Specifically, our proposal surpasses CUDPP, Thrust, ModernGPU, CUB and LightScan libraries and obtains better performance thanks to a novel collaborative strategy among several GPUs, which allows solving large data sets that cannot be solved by other libraries.

Acknowledgment

This work was cofunded by the Government of Galicia and ERDF funds from the EU, under the Consolidation Programme of Competitive Reference Groups [ED431C 2017/04] and Research Units [R2014/049 and R2016/037]; by the Ministry of Economy of Spain and ERDF funds [TIN2016-75845-P]; and by the Ministry of Education of Spain (FPU14/02801). This work is also partially supported by JST CREST [JPMJCR1303 and JPMJCR1687] and NVIDIA GPU Center of Excellence; and conducted as research activities of AIST-TokyoTech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL).

References

[1] *CUDA Data Parallel Primitives Library*, 2014, v2.2.

[2] *BPLG: Butterfly Processing Library for GPUs*, 2016. [Online]. Available: http://bplg.des.udc.es

[3] A. P. Diéguez, M. Amor, R. Doallo, "Efficient Scan Operator Methods on a GPU," *Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD'14)*, vol. 1, pp. 190–197, 2014.

[4] G. E. Blelloch, "Vector Models for Data-Parallel Computing," *Tech. Rep. in MIT Press*, 1990.

[5] A. P. Diéguez, M. Amor, J. Lobeiras, and R. Doallo, "Solving Large Problem Sizes of Index-Digit Algorithms on GPU," *IEEE Transactions on Computers*, vol. 67, no. 1, pp. 86–101, 2018.

[6] 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 (2008)*, 2008, pp. 205–213.

[7] 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.

[8] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (Scan) with CUDA," in *GPU Gems 3*. Addison Wesley, 2007.

[9] J. Hensley, 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.

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

[11] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2012.

[12] 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, 1973.

[13] Y. Liu and S. Aluru, "Lightscan: Faster scan primitive on CUDA compatible manycore processors," *CoRR*, vol. abs/1604.04815, 2016.

[14] D. Merrill, M. Garland, and A. Grimshaw, "Policy-based tuning for performance portability and library co-optimization," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[15] *Thrust Library*, NVIDIA, 2015, v1.8.1.

[16] *CUB Library*, Nvidia Comp., 2015, v5.0. [Online]. Available: http://nvlabs.github.io/cub

[17] *NCCL Library*, Nvidia Comp., 2016. [Online]. Available: https://github.com/NVIDIA/nccl

[18] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[19] *ModernGPU Library*, Sean Baxter, 2016, v2.0.

[20] S. Sengupta, M. Harris, and M. Garland, "Efficient Parallel Scan Algorithms for GPUs," *Technical Report*, 2008.

[21] 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*, pp. 539–552, 2006.

[22] J. Sklansky, "Conditional Sum Addition Logic," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 2, pp. 226–231, 1960.

[23] G. L. Steele and J.-B. Tristan, "Using Butterfly-Patterned Partial Sums to Draw from Discrete Distributions," *Proceedings of 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP'17)*, pp. 341–355, 2017.

[24] V. Volkov, "Better Performance at Lower Occupancy," in *Proceedings of GPU Technology Conference (GTC 2010)*, 2010.

[25] S. Yan, G. Long, and Y. Zhang, "StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization," *SIGPLAN Not.*, vol. 48, no. 8, pp. 229–238, 2013.