Noname manuscript No.
(will be inserted by the editor)

BPLG-BMCS: GPU-Sorting Algorithm using a Tuning Skeleton Library

Adrián P. Diéguez · Margarita Amor · Ramón Doallo

Received: date / Accepted: date

Abstract In this work, we present an efficient and portable sorting operator for GPUs. Specifically, we propose an algorithmic variant of the Bitonic Merge Sort (BMS) which reduces the number of processing stages and internal steps, increasing the workload per thread and focusing on a multi-batch execution for multiple problems of a small size. This proposal is well matched to current GPU architectures and we apply different CUDA optimizations to improve performance. For portability, we use a library based on tuning building blocks. Thanks to this parametrization, the library can easily be tuned for different CUDA *GPU* architectures. Our proposals obtain competitive performance on two recent NVIDIA GPU architectures, providing an improvement of up to 11,794x over *CUDPP* and up to 6,467x over *ModernGPU*.

Keywords GPU · CUDA · Tuning · Building Blocks · Bitonic Merge Sort

1 Introduction

Sorting is a computational building block of high importance, being one of the most studied algorithms due to its impact. Many algorithms rely on the efficiency of sorting routines as core pillars of their own efficiency. For example, sorting is widely used in computer graphics and geographic information systems for building spatial data structures, and also acts as a basis for solving sparse matrix operations or MapReduce patterns [4].

There are several parallel sorting algorithms such as Radix sort [18], Mergesort [8], Bitonic sort [1], and Quicksort [7]. Furthermore, many of these algorithms have been developed for GPUs. Radix sort for GPUs was efficiently

A.P. Diéguez - M. Amor - R. Doallo
Grupo de Arquitectura de Computadores (GAC), Departamento de Electrónica e Sistemas
Facultade de Informática, Universidade da Coruña
Campus da Coruña, 15071 A Coruña, España
E-mail: adrian.perez.dieguez@udc.es, margamor@udc.es, doallo@udc.es

implemented in [6]. Quicksort algorithm in GPU was first implemented in [16], being improved in [3]. A hybrid algorithm that combines Mergesort and Bucketsort [2] was presented in [17] whereas new implementations based on Radix sort and Mergesort were developed in [15]. There are several *accelerated* libraries that integrate sorting within a set of different algorithms. As an example of these libraries, we can find CUDPP [13], CUB [14] and ModernGPU [12]. Performing a comparison of sorting primitives performance, currently ModernGPU is the fastest on small problem sizes, although all of these libraries were developed with large problem sizes in mind.

In [5], we proposed an algorithmic variant of BMS, called Bitonic Merge Comb Sort (BMCS), for solving limited-size problems that fit directly into the high bandwidth of GPU scratchpad memory (called shared memory in CUDA). Nowadays, the simultaneous execution of several problems of small size has a large impact in the most complicated simulations. For example, combustion, chemical and high-order finite-element models take advantage of this execution. On the other hand, BPLG[11] is a library which uses a tuned template library with a simple and unified interface for *parallel prefix algorithms* [9] with a set of skeletons or building blocks as a basis. These building blocks, predefined generic components, are parameterized where efficient implementation and specialization may exist for given CUDA architectures and algorithms, constraining programmers to using only the given set of skeletons. In this work, we propose a new version called BPLG-BMCS, that has been built based on tuning building blocks with BPLG and it has been improved by considering different CUDA optimizations. Our experimental results demonstrate that our implementation is faster than all previously published GPU sorting techniques for a multi-batch execution of small problem sizes. Future work will entail developing our library focused on dealing with many problems of medium and large sizes.

The remainder of this paper is organized as follows: Section 2 presents BMCS and its CUDA implementation. Section 3 presents our new BMCS proposal using BPLG library. Experimental results are discussed in Section 4 whereas the main conclusions of our work are explained in Section 5.

2 Bitonic Merge Comb Sort (BMCS)

In this section, we present an algorithmic variant of Bitonic Merge Sort (BMS), called Bitonic Merge Comb Sort (BMCS), which achieves high parallelism and efficiency in GPU using CUDA. An initial version of BMCS was proposed in [5], matching well to current NVIDIA GPU architectures. Furthermore, it adapts thread workload to available registers, obtains coalescing accesses, avoids bank conflicts, reduces synchronization barriers and uses shuffle instructions.

BMS is a parallel algorithm for sorting [1]. The classic complexity is of $N \cdot (logN)^2$. Figure 1 shows the classic algorithm for $N = 16$ where each horizontal line represents a key value, starting on the left and finishing at the outputs on the right. Vertical segments are comparators which make the

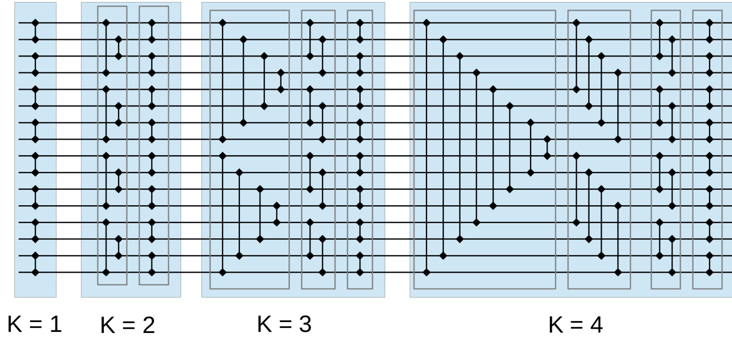

Fig. 1 Bitonic Merge Sort Algorithm for $N = 16$.

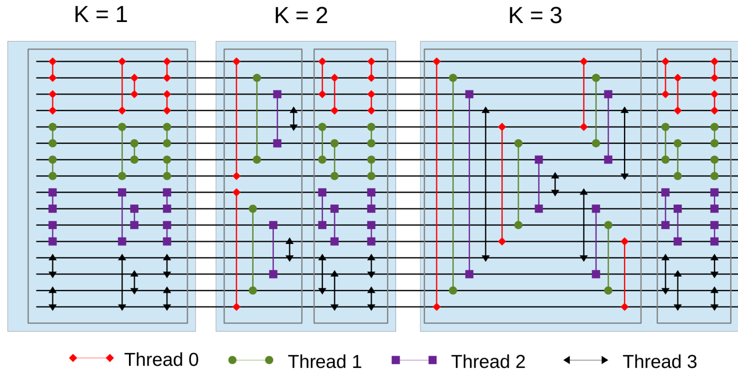

Fig. 2 Bitonic Merge Comb Sort Algorithm for $N = 16$.

comparison of the two selected keys, swapping their values if necessary. The sorting is processed along $log_2 N$ stages (grey boxes) where stage k has also k internal steps (marked by rectangles inside each stage).

The *Node* operator is the responsible for performing computation in parallel prefix algorithms. In more detail, the node $Node_{ALG}$ is a computational operator defined by four aspects: *fan_in* which is the number of input data; *fan_out*, the number of output data; *sizeof_data* which represents the size in bytes of each data, and the specific operation depending on the ALG algorithm. Furthermore, R is a factor specific to each algorithm where $\frac{N}{R}$ indicates the number of nodes per stage.

In BMS, the *node fan_in* and *fan_out* are both 2, processing 2 elements per thread. The best results are achieved when each thread works with 4 elements in order to increase thread workload, hence we have modified the algorithm, so *fan_in* and *fan_out* are both 4. We propose an algorithm with $log_2 N - 1$ external stages, each with half the number of the internal steps. In the case of external stages, 4 consecutive elements from global memory are read and

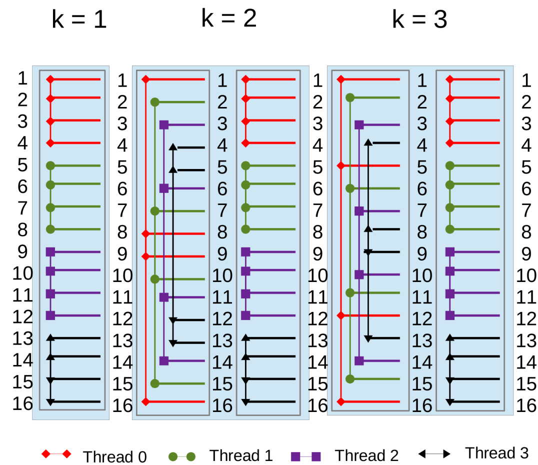

Fig. 3 Bitonic Merge Comb Sort Algorithm for $N = 16$.

computed directly in registers, thus first two external stages are reduced to one. Figure 2 contains a scheme of our algorithm for a problem of $N = 16$ with 4 threads where each thread computes a node of *fan_in=fan_out=4*. Reducing the number of stages and steps involves reducing synchronization barriers. Furthermore, increasing the *fan_in* and *fan_out* implies reducing the amount of threads needed per problem. Nevertheless, *fan_in* or *fan_out* in excess of 4 results in the use of more registers, decreasing occupancy and performance.

In order to develop this algorithm, we have taken into account avoiding dynamic indexing as well as achieving coalesced global memory accesses and preventing shared memory bank conflicts. When working with arrays as in this case, attention needs to be paid to accesses. Here we should point out that highest performance is achieved with static accesses where the compiler can derive constant indices in all accesses, placing elements into registers. If the compiler cannot determine the index at compile-time, array elements are placed in local memory in a process known as dynamic indexing. Our proposal avoids dynamic indexing with index calculations that can be understood as static accesses by compiler.

On the other hand, if each thread computes several elements, it is also necessary to specify which elements are accessed per thread in order to avoid bank conflicts or uncoalesced accesses (see Figure 2). The first and last steps are designed to enable the use of customized data types such as *Int4* which reduce the number of memory transactions and allow coalescing accesses.

The matching of threads with elements is not trivial in remaining stages in order to reduce the number of internal steps and to avoid shared memory bank conflicts. Depending on the specific external stage, the number of internal steps could be a non-power of two. In this case, it is necessary a mixed computation of *fan_in 2* is necessary in the first step of stage k, whereas the remaining steps

are processed by nodes of *fan_in 4*. Furthermore, each thread has to operate with the corresponding four elements which makes it possible to reduce the two steps in the *fan_in 2* naive approach to one step in the *fan_in 4* approach. This is shown clearly in Figure 2 when $k = 3$. In the first step of stage $k = 3$, the first, fifth, twelfth and sixteenth elements have to be processed by thread zero in order to reduce the first two steps of the naive algorithm to just one in our algorithm. Actually, thread zero could select other elements to perform the computation in other steps. However, this pattern is selected in order to reduce the number of shared memory bank conflicts. Figure 2 clearly shows how the pattern used in the internal steps allows consecutive threads to access to adjacent shared memory banks (without bank conflicts), except in last internal step where each thread loads 4 consecutive elements.

Figure 3 represents *BMCS* but using nodes $Node_{BMCS} = \{4, 4, 4, operator\}$. In this figure, the "comb" configuration can easily be seen. Increasing *fan_in/out* entails reducing the number of stages, i.e., reducing the number of synchronization barriers. It should be noticed that the naive algorithm has $n + \frac{n(n+1)}{2}$ synchronization barriers where $N = 2^n$, whereas our proposal has $n - 1 + \frac{n/2(n/2+1)}{2}$. In general, performance is increased by reducing the number of synchronization barriers. Here, we should emphasise the case of Fermi CUDA architectures where each SM has only 32 SPs. In this case reducing the number of barriers has a greater impact, synchronizing large thread block sizes implies a lot of warp-context switchings per SM, wasting a lot of time on this task.

Finally, it is also possible to reduce synchronization barriers and memory latencies with *shuffle* instructions which allow information to be exchanged between threads in the same warp using registers instead of shared memory. This approach avoids warp-synchronization barriers, although they are limited to the warp scope. BMCS consists of a hybrid strategy in which the 6 initial stages are computed using *shuffle* instructions, sorting $fan_in \times warpSize$ elements in each warp, and then using shared memory as a communication channel between warps. Therefore, 128-data chunks are sorted without synchronization barriers. Thus, if $N \leq 128$, then any synchronization barrier is needed in the execution. For larger sizes, this technique saves 9 synchronization barriers as this is the number of barriers there are in the 6 initial stages. Even though it was also possible to use shuffle instructions in internal steps with chunks of up to 128 elements, empirically greater performance was obtained working with shared memory.

3 BPLG-BMCS

We are currently developing an efficient library for parallel prefix algorithms based on tuning building blocks, called Butterfly Processing Library for GPUs (BPLG) [11]. This library enables algorithms to be designed with flexibility and adaptability, so far other parallel algorithms, such as FFT or tridiagonal system solvers, have been implemented in BPLG [10]. To this end, we use a set of functions as building blocks, high level blocks of code in several layers

where each function computes a small part of the whole work. This flexibility does not compromise performance as a tuned strategy is followed, obtaining a set of parameters in order to achieve the optimal level of parallelism to be exploited on each GPU architecture.

The idea behind tuning building blocks is to write the same kernel or a very similar one for all parallel prefix algorithms using skeletons,with the body of kernel remaining constant throughout different algorithms, but changing the implementation of the *compute* building block. Both computation and manipulation building blocks have common features, such as the use of templates, which allows generic programming and template metaprogramming. Thanks to this behaviour many optimizations take place at compile time, for example fully unrolling statics loops. Additionally, all functions have been designed to operate in any GPU memory space. Data are loaded from one bufer to another with the *copy* function. In contrast, computing blocks modify their input data. The fundamental function of computing building blocks is the *compute* function, which performs computations in each stage. The interface of the *compute* function is implemented by different algorithms, each one with its corresponding operation, which can be defined either recursively or using specializations depending on N.

The current BPLG version focuses on a multi-batch execution for multiple problems of small size. Specifically, G problems of size N are simultaneously computed. A kernel is invoked with B number of blocks where each block is executed with L threads. Each thread computes Q nodes per stage; therefore, each problem is processed by $\frac{N}{R \times Q}$ threads where R is a specific factor in which $\frac{N}{R}$ indicates the number of nodes per stage. However, instead of executing only one problem per block, L_G problems are computed per block using batch execution in order to increase the thread parallelism. Thus, the total number of threads per block is expressed as $L = \frac{N}{R \times Q} \times L_G$. In addition to aforesaid definitions, threads within a block have access to S data stored in shared memory and P data stored in private registers. These parameters can be related as follows: $P = fan_in \times sizeof_data \times R \times Q$ whereas $S = P \times L$ and $B = \frac{G}{L_G}$. In comparison to other libraries, multi-batch problems are performed by a single kernel invocation, where each thread performs the computation associated to each node. Data are processed and stored in registers. After computation, threads collaborate with each other in exchanging data prior to the next stage.

Algorithm 1 presents a first version of the code for the BMS in terms of tuning building blocks, called *BPLG-BMCS-Naive*. The read-only buffers are declared as *const_restrict* pointers, allowing the memory accesses to be optimized using texture cache. This code uses two tuning building blocks, *copy* and *compute*. The code can be divided into four main sections:

– Initialization section (lines 3-7). Defines a number of identifiers and some memory offsets for loading and storing operations. Furthermore, registers and shared memory are also allocated.
– Load data from global memory (lines 8 - 9) and first compute stage (lines 10-11). Loads coalescent data using a 64- bit load to obtain 2 consecutive

```
1  template<int N, int Q, int S> __global__ void
2  BPLG_Bitonic ( const int* __restrict__ data) {
3    // Obtain group-1D, thread-X and batch-Y identifiers
4    ...
5    // Statically allocate registers and shared memory
6    int reg[Q*2];
7    __shared__ int shm[N > Q ? S : 1];
8    //Load data from global memory to registers
9    copy<2,Q>(reg, data, ...);
10   //First compute stage
11   compute<Q>(reg);
12
13   for(int accR=2; accR < N ; accR*=2) {
14     //Obtains strides and offsets from the iteration and threadId
15     int readOffset = ..., readStride = ... ;
16     //Reg-> Shm -> Reg
17     if(accR>MixR) __syncthreads();
18     copy<2,Q>(shm+2*threadId, 1,reg, ...);
19     __syncthreads();
20     copy<2,Q>(reg,shm+readOffset,readStride, ...);
21     compute<Q>(reg); //Computation in registers of first internal stage
22     //Internal stages
23     for(int j=accRad; j>1; j/=2) {
24       int readOffset = ..., readStride = ... ;
25       int writeOffset = ..., writeStride = ... ;
26       if (j<accRad) __syncthreads();
27       copy<2,Q>(shm+writeOffset, writeStride,reg, ...);
28       __syncthreads();
29       copy<2,Q>(reg,shm+readOffset,readStride, ...);
30       compute<Q>(reg);
31     }
32   }
33   //copy to global memory
34   copy<2,Q>(data,reg,...);
35 }
```

Algorithm 1: Kernel code for BMS algorithm using tuning building blocks, *BPLG-BMS-Naive.*

elements instead of accessing a single data element per memory request. In the code, *copy* $< 2, Q >$ (...) loads 2 consecutive elements Q times per thread. Then, elements are directly processed in registers by *compute* function. This building block compares and swaps values.

- Compute stages of the algorithm (lines 13-32). The loop computes the remaining stages of the algorithm with its internal steps. To this end, the loop loads the corresponding data into registers using shared memory and synchronization barriers. The synchronization barrier in line 17 is avoided in first iteration as data is already in registers. The same behaviour occurs in the internal loop on line 26 where results are returned in registers.
- Store data to global memory (lines 33-34). The final iteration of loop stores the results into registers, thus final result is moved from register to global memory using 64-bit stores, reducing the number of memory transactions.

In order to achieve high portability and efficiency in GPUs, this code has been slightly modified. Firstly, we generalize the data type to be sorted from integers into any generic $DTYPE$ using a new *template* parameter class, allowing for easy portability. The code structure remains constant but even so

	Platform 1	Platform 2
CPU	Intel Xeon E5-2660 CPU 2.2 GHz	Intel Core i7-2600 3.4 GHz
Memory	64 GB DDR3 1600	8 GB DDR3 1333
OS	CentOS 6.4	Ubuntu 12.04 LTS
Compiler	GCC 4.4.7	GCC 4.6.3
GPU	Nvidia Tesla K20 GPU	Nvidia GeForce GTX980
Driver	340.58, SDK 6.0	343.22, SDK 6.5

Table 1 Description of the test platforms

it provides portability, by simply changing the $DTYPE$ comparator implementation inside the *compute* function if necessary. Furthermore, our building blocks such as *copy* or *compute* have been carefully designed in order to avoid dynamic indexing.

The effectiveness of BMCS has been demonstrated in [5]. Thus, BMCS is adapted to BPLG, also providing specialization kernels for some $DTYPE$ and Q values, with specific code for each case which exploits the maximum parallelism of each proposal. For example, specialization kernels for enabling the use of customized data types as *Float2* or *Int4* has been implemented, reducing the number of memory requests and improving performance. This approach has been also optimized with a hybrid implementation; initial stages are computed using *shuffle* instructions, sorting $fan_in \times warpSize$ elements in each warp, and the other stages use shared memory as a communication channel between warps. If $N \leq 128$, then there is no synchronization barrier in the execution. Finally, *BPLG-BMCS* is the result of adapting BMCS to BPLG and applying all previous optimizations.

4 Experimental Results

In this section, we present the results of our proposals on different *NVIDIA* GPU architectures. All tests were run using integers as data type. All the data initially reside in the GPU memory, so there are no data transfers to CPU during benchmarks. The test platforms used in our experiments are described in Table 1, where Platform 1 has a representative GPU of Kepler architecture whereas Platform 2 has a Maxwell GPU. All these algorithms were developed to take advantage of the read-only data cache, which slightly improves global memory read bandwidth. The performance of these experiments is measured in million data processed per second, MData/s. The size of the batch depends on the input size and is given by the expression $G = 2^{24}/N$. Thus, MData/s value is performed using the expression $N \times G \times 10^{-6}/t$. In order to demonstrate the portion of effectiveness of the algorithm, tuning building blocks and other optimizations, all proposals were tuned to each architecture.

Firstly, Figure 4 depicts a performance comparison of all our proposals presented for Platform 1 in Sections 2 and 3. The *BMS* tag refers to an optimized Bitonic Merge Sort implementation whereas *BMCS* represents the implementatiton of our algorithmic variant with shuffle communications presented in

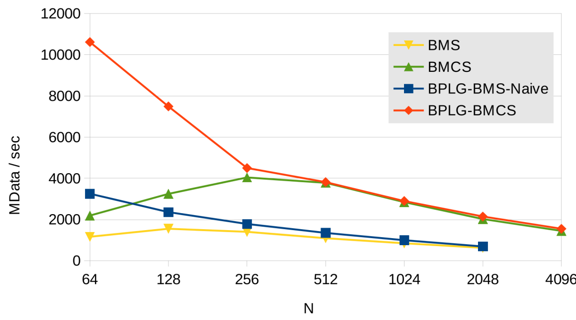

Fig. 4 Comparison of our proposal optimizations on Platform 1.

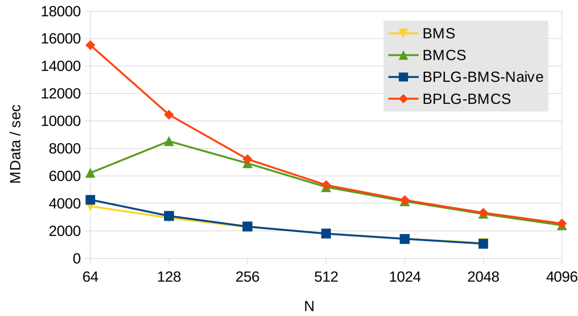

Fig. 5 Comparison of our proposal optimizations on Platform 2.

Section 2. *BPLG-BMS-Naive* denotes a naive implementation for $Q = 1$ in terms of tuning building blocks. *BPLG-BMCS* is a kernel specialization for $Q = 2$, with the optimizations presented in Section 3. As this implementation obtains the best performance, it is the one which we use in our BPLG library. In general, while shared memory is not an expensive resource, tuning building blocks implementations are better since they execute several batches per block. Until $N = 256$, BPLG-BMCS is better than BMCS, as each block executes several *batches* in parallel. The number of batches per block is obtained by tuning, and it guarantees a high occupancy. As N increases, the number of batches per block is reduced in BPLG, and performance is very similar with BMCS. In the case of BMCS, peak of performance is obtained with $N = 256$ or $N = 512$ as occupancy is maximum with these values. In problem sizes that are larger than $N = 512$, both BMCS and BPLG-BMCS can only execute one problem per block, owing to resource consumption. Even in this case, BPLG-BMCS is slightly better than BMCS. Tuning building blocks offer a simple way of programming, obtaining the same (or higher) performance than other complex-optimized verbose kernels for the same task, such as BMCS. In general, shared memory becomes a limiting factor. Figure 5 shows the same

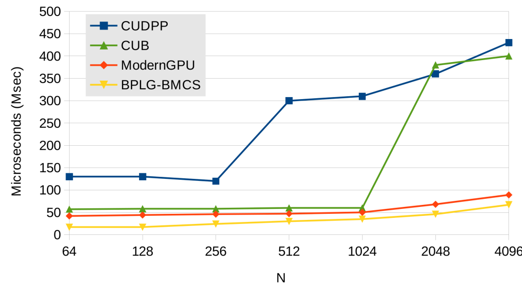

Fig. 6 Comparison of GPU sorting implementations for one batch on Platform 1.

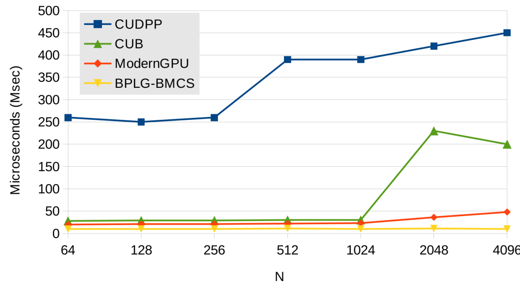

Fig. 7 Comparison of GPU sorting implementations for one batch on Platform 2.

comparison on Platform 2, maintaining the same nomenclature. The MData/s achieved on this platform is higher, which can be ascribed to the fact that Maxwell presents a power-efficient performance which provides a higher delivered performance per CUDA core than Kepler owing to its new datapath organization, new improved instruction scheduler, new memory hierarchy and bandwidth, obtaining a higher number of actives blocks per Streaming Multiprocessor. The architecture doubles the number of blocks per SM, up to 32 blocks (double that of Kepler), although the available shared memory per block remains the same. Owing to this behaviour, BMCS occupancy is maximum with $N = 128$ in Maxwell ($N = 256$ in Kepler).

Figure 6 compares BPLG-BMCS with respect to CUDPP implementation [13], CUB and ModernGPU library, all of them developed by *NVIDIA*, with *ModernGPU* being the reference library for sorting when small problem sizes are considered. It should be noted that this comparison is made in terms of execution time for only one batch. CUDPP shows the worst results for small problem sizes that can be directly processed in shared memory. Our proposal, *BPLG-BMCS*, provides highly competitive results compared to ModernGPU. This performance is achieved implementing *BPLG-BMCS*, obtaining an improvement of up to 10x over *CUDPP*, up to 8.26x over *CUB* and up to 2.6x over *ModernGPU*. On the other hand, Figure 7 presents the same comparison on Platform 2, Maxwell architecture. Results are similar to Platform 1, ob-

		Platform 1				Platform 2			
N	G	BPLG	ModernGPU	CUDPP	CUB	BPLG	ModernGPU	CUDPP	CUB
64	262144	10614	2	0.9	1.4	15521	4.3	2.4	2.9
128	131072	7492	3.7	1.8	2.7	10457	8.3	4.9	6.1
256	65536	4496	7.1	4	5.3	7218	16.3	10	12.3
512	32768	3817	13.4	7.9	10.2	5333	31.4	19.6	23.5
1024	16384	2897	25.2	14.7	20.4	4234	61.2	35.9	43.1
2048	8192	2144	34.4	16.1	5.5	3314	66.5	33.4	24.2
4096	4096	1549	50.4	16	10.4	2534	97.2	48	40.1

Table 2 MData/s comparison of GPU Multibatch Sorting Algorithms.

taining up to 40x in comparison to *CUDPP*, up to 20.9x over *CUB* and up to 4.8x over *ModernGPU*.

Many applications need to solve G batch problems in parallel. Therefore, we used a batch execution to compute G problems each time. Table 2 compares our best proposal BPLG-BMCS to CUDPP, CUB and ModernGPU ones. CUB, ModernGPU and CUDPP prove to be extremely ineffecient with problems where many *batches* of small size are processed in parallel, since they were designed for solving just one large-size problem. In order to solve G problems of size N, these two libraries have to launch G *light* kernels. Our proposal is up to 11,794x faster than *CUDPP* library, up to 7,581x over *CUB* and up to 5,307x over *ModernGPU* on Platform 1. Table 2 shows the MData/s obtained in Platform 2. The MData/s for Platform 2 are higher because Maxwell presents a power-efficient performance which provides a higher delivered performance per CUDA core than Kepler thanks to new datapath organization, new improved instruction scheduler, new memory hierarchy and bandwidth, and more actives blocks per SM. Our proposal is up to 6,467x faster than *CUDPP*, up to 5,352x over *CUB* and up to 3,609x than *ModernGPU*.

5 Conclusions

Sorting is a highly important kernel which takes part in many scientific and engineering applications. In terms of efficiency, a GPU implementation of this operator is not a simple task since hardware requirements have to be considered. In this paper, we provide a new proposal for solving sorting problems that match well to the new GPU architectures. Specifically, this proposal is an algorithmic variant of Bitonic Merge Sort (BMS), called Bitonic Merge Comb Sort (BMCS). Furthermore, we obtain a new proposal BPLG-BMCS using a tuning methodology based on tuning building blocks for parallel prefix algorithms. In addition to tuning building blocks, BPLG-BMCS is also optimized with different CUDA techniques such as specialized templates, shuffle instructions or customized datatypes.

Despite its mathematical complexity, the performance of the resulting proposal obtains very competitive results compared to other well-known sorting libraries, such as *ModernGPU*, *CUB* and *CUDPP*, achieving an improvement of up to 40x for single-batch execution and up to 11,794x for multibatch execution.

As future work, our primary focus will be to extend our proposal for large-size arrays that cannot be stored in shared memory.

Acknowledgements

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); by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P) and by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

References

1. Batcher, K.E.: Sorting Networks and their Applications. In: Proceedings of Spring Joint Computer Conference, AFIPS '68 (Spring), pp. 307–314 (1968)
2. Corwin, E., Logar, A.: Sorting in Linear Time - Variations on the Bucket Sort. Journal of Computing Sciences on Colleges **20**(1), pp. 197–202 (2004)
3. Cederman, D., Tsigas, P.: GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. J. Exp. Algorithmics **14**, 4:1.4–4:1.24 (2010)
4. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM **51**(1), 107–113 (2008)
5. Diéguez, A.P., Amor, M., Doallo, R.: BS-Comb: An Efficient Sorting Algorithm for GPUs. In: Proceedings of the 15th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2015, pp. 461–473 (2015)
6. Harris, M., Sengupta, S., Owens, J.D.: Parallel Prefix Sum (Scan) with CUDA. GPU Gems **3**(39), pp. 851–876 (2007)
7. Hoare, C.A.R.: Algorithm 64: Quicksort. Commun. ACM **4**(7) (1961)
8. Kipfer, P., Westermann, R.: GPU Gems 2-Chapter 46. Improved GPU Sorting (2005)
9. Ladner, R.E., Fischer, M.J.: Parallel Prefix Computation. J. ACM **27**(4), pp. 831–838 (1980)
10. Lobeiras, J., Amor, M., Doallo, R.: Designing Efficient Index-Digit Algorithms for CUDA GPU Architectures. IEEE Transactions on Parallel and Distributed Systems (In Press)
11. Lobeiras, J., Amor, M., Doallo, R.: BPLG: A Tuned Butterfly Processing Library for GPU Architectures. International Journal of Parallel Programming **43**(6), pp. 1078–1102 (2015)
12. Nvidia Comp.: Modern GPU library (2013). URL https://github.com/NVlabs/moderngpu
13. Nvidia Comp.: CUDPP: CUDA Data Parallel Primitives Library (2014). URL http://cudpp.github.io/
14. Nvidia Comp.: CUB library (2015). URL http://nvlabs.github.io/cub/
15. Satish, N., Harris, M., Garland, M.: Designing Efficient Sorting Algorithms for Manycore GPUs. In: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IPDPS '09, pp. 1–10 (2009)
16. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH '07, pp. 97–106 (2007)
17. Sintorn, E., Assarsson, U.: Fast Parallel GPU-sorting Using a Hybrid Algorithm. J. Parallel Distrib. Comput. **68**(10), pp. 1381–1388 (2008)
18. Zagha, M., Blelloch, G.E.: Radix sort for vector multiprocessors. In: Proceedings Supercomputing '91, pp. 712–721 (1991)