2015 44th International Conference on Parallel Processing

In-Place Data Sliding Algorithms for Many-Core Architectures

Juan Gómez-Luna
Computer Architecture and Electronics
University of Córdoba
Córdoba, Spain
el1goluj@uco.es

Li-Wen Chang, and Wen-Mei W. Hwu
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
{*lchang20, w-hwu*}*@illinois.edu*

I-Jui Sung
MulticoreWare, Inc.
Champaign, Illinois, USA
ray@multicorewareinc.com

Nicolás Guil
Computer Architecture
University of Málaga
Málaga, Spain
nguil@uma.es

***Abstract*—In-place data manipulation is very desirable in many-core architectures with limited on-board memory. This paper deals with the in-place implementation of a class of primitives that perform data movements in one direction. We call these primitives Data Sliding (DS) algorithms. Notable among them are relational algebra primitives (such as *select* and *unique*), padding to insert empty elements in a data structure, and stream compaction to reduce memory requirements. Their in-place implementation in a bulk synchronous parallel model, such as GPUs, is specially challenging due to the difficulties in synchronizing threads executing on different compute units. Using a novel adjacent work-group synchronization technique, we propose two algorithmic schemes for regular and irregular DS algorithms. With a set of 5 benchmarks, we validate our approaches and compare them to the state-of-the-art implementations of these benchmarks. Our regular DS algorithms demonstrate up to 9.11× and 73.25× on NVIDIA and AMD GPUs, respectively, the throughput of their competitors. Our irregular DS algorithms outperform NVIDIA Thrust library by up to 3.24× on the three most recent generations of NVIDIA GPUs.**

***Keywords*-in-place; stream compaction; relational algebra.**

I. Introduction

We identify and introduce *Data Sliding (DS) algorithms* as a class of data manipulation primitives which perform unidirectional data shifting within a memory area. Essentially, they move all input array elements or some of them (e.g., valid elements are those that satisfy a certain condition) in only one direction. Additionally, they are often required to ensure stability, meaning that the relative order of input elements is maintained. Depending on how the shifting offset is calculated, they can be classified as regular or irregular. In regular DS algorithms, groups of consecutive input array elements are shifted by a constant amount of memory positions, which might be different for each group. For instance, while padding a matrix (i.e., adding extra columns), all elements of one row are moved forward the same number of positions. Irregular DS algorithms apply a varying shifting offset to input elements, which is typically data dependent. For instance, in stream compaction the offset depends on the number of unwanted data instances that will be removed.

A major reason for applying DS algorithms is to reshape data structures in order to gain performance benefits in later computation. On the one hand, when the unidirectional movement expands the input array elements, DS algorithms (e.g., padding) can potentially provide better memory system data movement and cache efficiency due to memory alignment. On the other hand, the movement can shrink the input array, thus reducing memory or storage consumption (e.g., stream compaction).

DS algorithms can be found from an important set of applications. First among them, as mentioned above, padding extra columns in a matrix is a regular DS algorithm to guarantee memory alignment for higher memory performance [1][2]. Matrix transposition can also benefit from padding. Due to a simplified algorithm for swapping symmetric elements along the diagonal in square matrices, rectangular matrices (specially near-square ones) can be transposed by padding with extra rows or columns to make the matrices square. Second, rational algebra operators [3][4], such as *select* and *unique*, can be considered as irregular DS algorithms as they shift elements that satisfy a certain predicate in one direction. For instance, *unique* maintains in the output array those elements that are different from the previous element. Third, also aforementioned, stream compaction removes unwanted elements in sparse data, thus irregular DS algorithms are often deployed in many applications, such as tree traversal [5], ray tracing [6], image processing [7], and sparse matrix-vector multiplication [8].

An important implementation decision is if the output array of a DS algorithm occupies the same physical space as the input array. In-place algorithms are desirable because they are less constrained by memory capacity. This is especially true for many-core architectures, such as GPUs, where memory capacity is much more limited than CPUs. However, working in-place is more challenging than working out-of-place, because of inherent memory space conflicts. A careful coordination is needed to avoid overwriting data. The need for in-place data transformations has led recent research efforts such as those about in-place transposition of rectangular matrices [9][10].

A challenge in implementing parallel in-place DS algo-

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.30

rithms for GPUs is dependency between copying out existing elements and filling in new elements at the same location. Previous works [11] exploit parallelism from dependency-free movable elements based on input and output memory locations. As dependency changes as it executes an algorithm, multiple shift operations take place. However, it uses expensive global synchronization between each kernel invocations despite that the scope of dependency is much narrower than the entire memory. Moreover, the parallelism drains quickly as the execution continues, thus degrading memory performance significantly.

We simplify the synchronization for avoiding space conflicts by introducing proper scheduling of work-groups. Furthermore, by using an efficient adjacent work-group synchronization mechanism, the approaches we present in this work enable tiling and parallel work-group execution to exploit Memory-Level Parallelism (MLP). MLP critically dominates performance, as DS algorithms are inherently memory-bound. Moreover, auxiliary on-chip memory space is used in a way where each work-item loads a certain number of input array elements, that is, the coarsening factor. Properly tuned coarsening maximizes both MLP and Instruction-Level Parallelism (ILP) and meanwhile minimizes the need for adjacent work-group synchronization.

This paper makes the following contributions:

- We identify a class of parallel primitives that all move array elements in one direction. We call them DS algorithms.
- We identify the limitations of traditional in-place approaches to DS algorithms.
- We propose a generic algorithm that guarantees in-place and stable implementations of these primitives.
- We demonstrate that our approach outperforms the state-of-the-art implementations (such as NVIDIA Thrust library [12]), and evaluate the performance portability across GPU and CPU architectures [1].

The rest of this paper is organized as follows. Section II identifies the limitations of the bulk synchronous parallel model to perform global synchronization. It also describes a motivating example, which is padding extra columns in a matrix. In Section III we present our algorithms for in-place Data Sliding algorithms. In Section IV we apply our algorithms to several primitives and compare them to state-of-the-art implementations. Section V presents related work. Finally, Section VI concludes the paper.

II. Background

A. Motivating Example: Padding Matrix Columns

Padding a matrix in row-major order with extra columns is an operation that fits our definition of DS algorithm. Every row of the matrix (except row 0) is moved forward in memory. The relative order of the row elements and the relative order of rows should remain invariant, which we refer to as stability.

A sequential implementation is straightforward, regardless whether it was out-of-place or in-place. In Figure 1(a), in-place padding involves slightly shifting each row: row i will be shifted by $C \times i$ where C is the number of columns to be padded to each row. The simplest way [13] to implement this in-place padding scheme is to move each row starting from the last one, i.e., move row 4 in Figure 1(a), then row 3, and so on. In the case of in-place padding, the extra memory space needs to be previously allocated adjacent to the matrix.

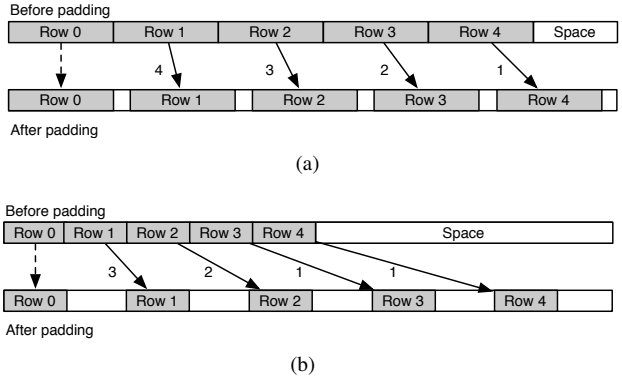


Figure 1. In the in-place padding of a row-major matrix composed of 5 rows. In general, it is a sequential operation (a). If there is enough room (b), multiple rows can be moved in parallel. In this example, Row 3 and Row 4 can be moved in parallel in iteration 1, but Row 2 has to be moved in iteration 2 as it overlaps with the space taken by Row 3 and Row 4. The number on each data movement arrow labels the iteration in which the movement can performed.

In the worse case, each row will be moved sequentially, due to space conflicts. However, depending on the extra memory space available, it might be possible to move multiple rows in parallel [11]. The number of rows that can be moved parallel can be determined using the original number of rows, the number of rows that have been moved to the destination, the number of columns to be padded and the number of elements in a row *before* padding.

Figure 1(b) shows such parallel in-place padding. Row 3 and Row 4 will be moved in parallel. However, as the execution continues, the number of movable rows in parallel decreases as the extra space decreases. Thus, at some point, the rest of rows will be still moved sequentially. In this example, Row 2 and Row 1 will be moved sequentially.

Figure 2 plots the performance of such padding scheme. The program is written in OpenCL and executed on an NVIDIA Tesla K20 GPU for a 5K $\times$ 4.9K matrix. Note that in this particular case, after 181 iterations, there are still 99 total rows to be moved, but the space is insufficient to move even more than one row in parallel. After this point, we have to move the rest of the rows sequentially. Here one work-group loads the content of an entire row to the

[1]Our codes are publicly available at https://bitbucket.org/gomezlun/in-place-ds-algorithms/

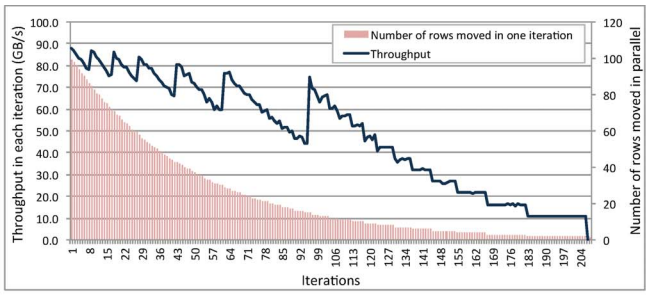


Figure 2. Throughput of parallel in-place padding a 5K × 4.9K matrix to square on a K20 GPU. The thin columns show the parallelism available in each iteration, i.e., the number of rows that can be moved in parallel.

temporary storage in the scratchpad memory synchronously, and then stores the entire row to the destination. As the peak memory bandwidth of Tesla K20 is roughly 208 GB/s, the performance of the in-place padding is actually quite good when there is enough rows to be moved. However, the performance degrades quickly and eventually goes to 10 GB/s in the latest stage (i.e., sequentially moving row-by-row due to limited space available). The effective throughput is 38.2 GB/s for this case, which is less than 20% of peak memory bandwidth. The limited number of concurrently executing work-groups constricts the available MLP, and consequently the global memory bandwidth is under exploited.

B. Insights into Limiting Factors

The primary limiting factor of parallelism in the previous example is that rows have dependency on other rows. For instance, processing Row 2 has dependency on processing Row 3 and 4, because output location of Row 2 overlaps with Row 3 and 4. In order to observe the dependency, previous implementations exploit parallelism by available extra space and iteratively execute this process. With this example, three such iterations take place, moving Row 3 and 4 together, then Row 2, and finally Row 1.

The iterative application of movement is done via multiple kernel invocations. Here, global synchronization between kernel calls is used as a mean of enforcing the dependency. However, the scope of dependency is much narrower as a work-group only needs to know which row it is processing and if processing of dependent rows are done.

In bulk synchronous parallel models, such as CUDA or OpenCL, there is no provided way to synchronize work-items belonging to different work-groups, unless the kernel is terminated. Global synchronization entails an inherent performance bottleneck. Such costly global synchronization can be avoided if it were possible to efficiently synchronize work items belonging to different work-groups.

III. Proposed Generic Algorithm

This section describes our proposed generic algorithm to carry out in-place stable unidirectional data movements. It consists of a loading stage and a storing stage, which are separated by a synchronization operation. It has two different versions (regular and irregular) depending on whether the sliding offset is data dependent.

Algorithm 1 Regular Data Sliding algorithm. A work-item wi_id loads into a number $\#coarsen\_factor$ array elements into on-chip memory, including registers and scratchpad memory, from locations pos_in. After Adjacent_wg_synchronization, these elements are stored in locations, pos_out, that are later calculated.

```
wg_id_ = Dynamic_work_group_id_allocation() (Figure 4)
pos_in = wg_id_ × coarsen_factor × wg_size + wi_id
for i = 0 to coarsen_factor do
  if pos_in < total_size then
    OnChipMem_i = Array[pos_in]
  end if
  pos_in+ = wg_size
end for
Adjacent_wg_synchronization(wg_id_) (Figure 3)
for i = 0 to coarsen_factor do
  Calculate pos_out
  Array[pos_out] = OnChipMem_i
end for
```

A. Regular Data Sliding

In Algorithm 1, in the loading stage, work-items of a work-group i load consecutive array elements into on-chip memory (registers and local scratchpad memory), which is used as a high-bandwidth auxiliary space for in-place data movement[2]. Synchronization among adjacent work-groups is performed after the loading stage to guarantee that data is read before the memory addresses are written by other data. By avoiding memory space conflicts among work-groups using adjacent synchronization, in the storing stage, data can be written in the corresponding memory addresses after adding or removing padding data.

Each work-group has an associated flag that will be set during the synchronization operation. When the loading stage finishes, work-group i waits for the flag $i-1$ to be set. As soon as this flag is set, work-group i sets its flag, and the storing stage starts. The code of the synchronization mechanism is in Figure 3. Flags should be read and written *atomically*. Similar synchronization procedures between adjacent work-groups are explained in [14][15].

The major difference is that adjacent synchronization in most previous works is used to resolve computed data communication between two adjacent work-groups, while in our cases, adjacent synchronization is applied to avoid memory space conflicts among work-groups (i.e., data overwriting). Since there is no real data communication, no memory fence is really required.

[2] In OpenCL stacks, especially CPUs, using local memory might not directly imply high-bandwidth on-chip memory. However, due to their scheduling policy implementation, local memory operations can induce cache hits, so they result in high bandwidth.

```
barrier(local memory fence);
if (wi_id == 0){
  // Wait
  while (atom_or(&flags[wg_id_ - 1], 0) == 0){;}
  // Set flag
  atom_or(&flags[wg_id_], 1);
}
barrier(global memory fence);
```

Figure 3. Code segment of adjacent work-group synchronization. The work-item with `wi_id == 0` executes the loop until the flag of the previous work-group is set. Then, it sets its associated flag. The initial local barrier ensures that all work-items of the same work-group have finished the loading stage. The final barrier forces other work-items in the work-group to wait until they are enabled to continue execution, and ensures correct ordering of global memory operations.

```
__local int wg_id_;
if (wi_id == 0)
  wg_id_ = atom_add(&S, 1);
barrier(local memory fence);
```

Figure 4. Code segment of dynamic work-group ID allocation. As soon as a work-group is scheduled, the first work-item `wi_id == 0` gets the dynamic work-group ID by incrementing a location `S` in global memory, which was initialized to 0. The dynamic work-group ID is stored in shared memory, so that it is visible to every work-item in the work-group after the synchronization.

The non-deterministic scheduling of work-groups might potentially cause deadlocks. Work-groups are scheduled onto compute units when there are free resources, but the scheduling order is not guaranteed. If work-group $i-1$ is launched after work-group i, deadlock might happen. This is avoided by a dynamic work-group ID allocation [14], shown in Figure 4.

It is notable that in our proposed method, each work-group synchronizes with only its previous work-group (e.g., $wg\_id\_ = 4$ synchronizes with $wg\_id\_ = 3$), instead of checking whether space conflicts happens or synchronizing with all overlapped work-groups. The key insight here is that by chaining dependencies between neighboring work-groups, the space conflicts are guaranteed not to happen because a storing stage will take place once all of its previous work-groups complete loading stages.

Figure 5 illustrates the runtime difference between our proposed approach and the previous work [11] in the case of padding matrix columns (Figure 1(b)). Figure 5 is a simplified representation, because it shows work-groups moving entire rows. Actually, our algorithm is oblivious to number of rows and row boundaries. The work-groups load all input elements, and then they take care of the boundaries when calculating the output positions. Our algorithm assigns a pre-defined order to the work-group scheduling, which is actually a sequential order (ascendant or descendant). While adjacent work-groups are in the loading or the storing stage, they carry out concurrent memory requests intensively. Thus, they leverage the available MLP to fully exploit the global memory bandwidth. Between the loading and the storing stages, the adjacent synchronization mechanism performs as a lightweight barrier, which guarantees the correct ordering

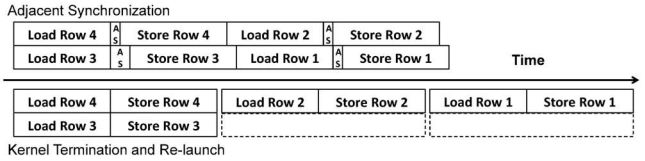


Figure 5. Comparison between the proposed approach using adjacent synchronization and the previous approach using kernel termination and re-launch in the example of Padding Matrix Columns (Figure 1(b)) with two concurrent work-groups: AS denotes adjacent synchronization including both polling and setting the flags.

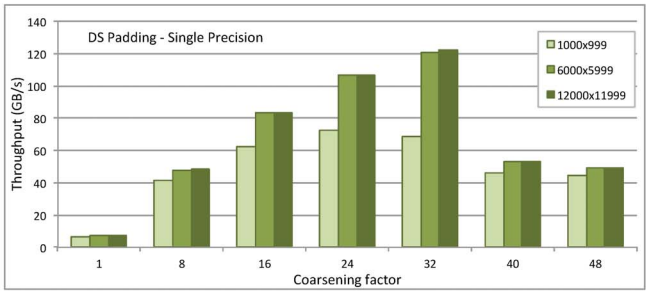


Figure 6. Throughput (GB/s) of OpenCL versions of DS Padding for single precision on NVIDIA GTX 980 (Maxwell). The coarsening factor changes between 1 and 48. The work-group size is 256. The legend presents matrix sizes.

of global memory loads and stores.

Additionally, the algorithm employs a tunable coarsening factor, which is the number of array elements that each work-item loads into on-chip memory, and work-group size. By properly adjusting them for each architecture, it can be possible to maximize the ILP and MLP (memory reads and writes by one work-item are independent of each other), and meanwhile minimize the number of adjacent work-group synchronizations (the higher the coarsening factor, the lower the number of work-groups).

The effects of coarsening are illustrated for our DS Padding algorithm by Figure 6 between 1 and 32. However, the coarsening factor cannot be arbitrarily large. For 40 and 48 the throughput falls down significantly, because the need for on-chip resources (registers and/or scratchpad memory) is too large and data must be spilled to off-chip memory. Our padding algorithm is more extensively evaluated in Section IV, where all primitives are tuned with the best coarsening factors.

B. Irregular Data Sliding

Irregular Data Sliding (Algorithm 2) requires additional steps to determine the exact output location for the storing stage. The loading stage is similar to the one in regular data sliding. After the loading stage, for those array elements that evaluate true a certain predicate, the corresponding work-items update a local counter. Then, a *binary* prefix sum operation within a work-group is required to calculate the sliding offset. The sliding offset of the last element within a work-group can be sent to the next work-group when adjacent synchronization is performed. Similar to previous

```
barrier(local memory fence);
if (wi_id == 0){
  // Wait
  while (atom_or(&flags[wg_id_ - 1], 0) == 0){;}
  // Set flag
  int flag = flags[wg_id_ - 1];
  atom_add(&flags[wg_id_], flag + count);
  count = flag;
}
barrier(global memory fence);
```

Figure 7. Code segment of adjacent work-group synchronization. The work-item with `wi_id == 0` executes the loop until the flag of the previous work-group is set. Then, it sets its associated flag by adding the previous flag and a counter in shared memory, which contains the result of the reduction operation. The previous flag is then stored in `count`, because it will be needed in subsequent calculations. The final barrier forces other work-items in the work-group wait until they are enabled to continue execution, and ensures correct ordering of global memory operations.

works [14][16], a reduction operation could replace the scan operation to quickly compute the sliding offset of the last element to reduce the critical path of adjacent synchronization. In this case, the binary prefix sum operation is performed after adjacent synchronization. Finally, in the storing stage, data can be written in the corresponding sliding offset.

Both reduction and binary prefix sum operations within a work-group are well studied. A reduction operation can be performed using the CUDA SDK reduction code [17]. In the more advanced GPUs, such as NVIDIA Kepler architecture or later, this code can further use shuffle instructions and perform more efficiently. In this paper, the reduction is by default performed in CUDA SDK reduction code re-written in OpenCL without extra notation. If extra optimizations are applied, a notation is further given.

An intra-work-group binary prefix sum operation can be performed as a balanced tree algorithm [18]. In NVIDIA Fermi architecture or later, as the values to be scanned are either 0 or 1, the operation can be implemented using the voting instruction `__ballot` and the bit manipulation instruction `__popc` [19]. In NVIDIA Kepler architecture or later, it can also use shuffle instructions as explained in [20]. In the paper, by default, the binary prefix sum is performed using the balanced tree algorithm without extra notation. A notation is given for further optimizations.

Figure 7 shows the modified adjacent synchronization for irregular DS algorithms. Different from ones for regular cases, the modified one not only avoids memory space conflicts but also passes computed memory address offset to adjacent work-groups. In this sense, the modified one is very close to [14][15]. Since only unsigned integers are used to represent memory address offsets, our adjacent synchronization does not need to support other data types or structures.

IV. Application and Evaluation

In this section, we apply our generic algorithms to several primitives that are widely-used in numerous applications. CUDA versions of these kernels are evaluated on

Algorithm 2 Irregular Data Sliding algorithm. A work-item wi_id loads into a number $\#coarsen\_factor$ array elements into on-chip memory from locations pos_in. A local counter increases if elements satisfy a certain predicate. After Adjacent_wg_synchronization, these elements are stored in locations, pos_out, that are later calculated, using binary prefix sum.

$wg\_id\_ =$ **Dynamic_work_id_allocation (Figure 4)**
$pos\_in = wg\_id\_ \times coarsen\_factor \times wg\_size + wi\_id$
for $i = 0$ **to** $coarsen\_factor$ **do**
  if $pos\_in < total\_size$ **then**
    $OnChipMem_i = Array[pos\_in]$
    if $Predicate(OnChipMem_i)$ **then**
      $local\_count++$
    end if
  end if
  $pos\_in += wg\_size$
end for
Reduction($local\_count$)
Adjacent_wg_synchronization($wg\_id\_$) **(Figure 7)**
for $i = 0$ **to** $coarsen\_factor$ **do**
  if $Predicate(OnChipMem_i)$ **then**
    Calculate pos_out **using binary_prefix_sum within the work-group**
    $Array[pos\_out] = OnChipMem_i$
  end if
end for

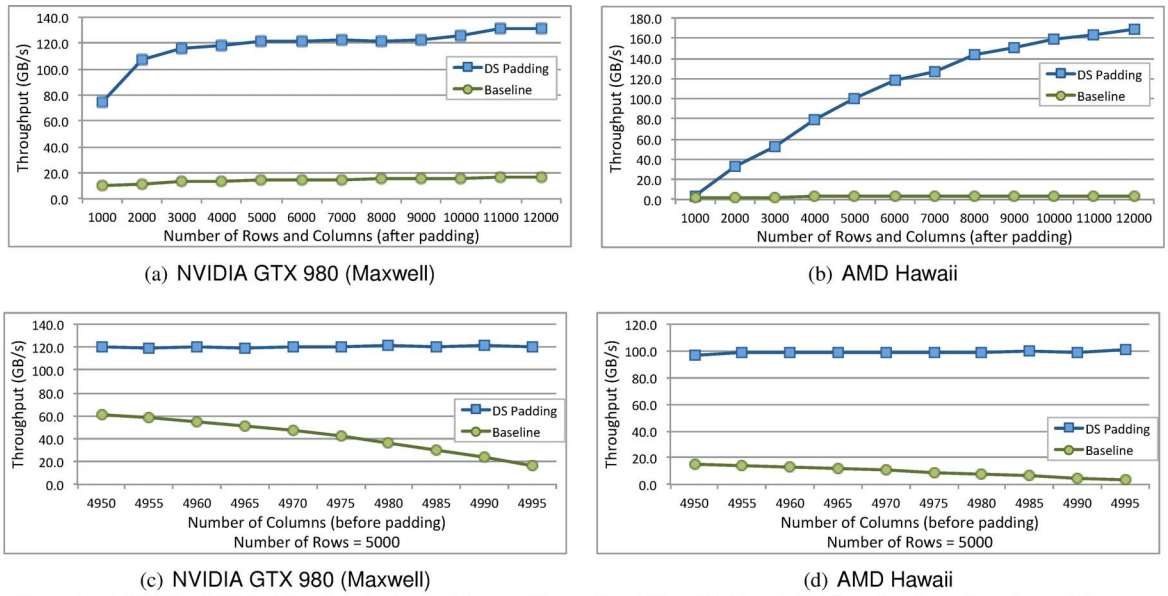
NVIDIA GPUs belonging to the three most recent architectures: Fermi (GeForce GTX 580), Kepler (Tesla K20), and Maxwell (GeForce GTX 980). OpenCL versions are evaluated on the same NVIDIA GPUs and AMD Hawaii and Kaveri GPUs. Tests on NVIDIA devices have been carried out with CUDA SDK 6.5, and on AMD devices with AMD SDK 2.9.1. OpenCL versions are also run on an Intel Core i7-3820 CPU (using only 4 out of 8 memory modules) with Intel OpenCL stack (driver version 1.2.0.8, Intel C Compiler version 14.0.1) and MxPA [21] compiler, in order to evaluate the performance portability across heterogenous devices. We compare our DS implementations with Sung's [11] and NVIDIA Thrust library (version 1.8.0) [12].

A. Padding and Unpadding

Introduced in Section II-A, padding extra columns in parallel can be difficult. In the implementation explained in [11], the amount of rows that can be moved concurrently depends on the amount of padding. A related primitive undoes the padding, that is, it removes the extra columns. The unpadding operation is even trickier, because there is no empty space in the beginning, as in the case of padding. The baseline implementation of unpadding we test below is using only one work-group.

Our DS Padding and DS Unpadding primitives follow Algorithm 1, because the number of positions that every element in a row of the matrix is shifted is the same.

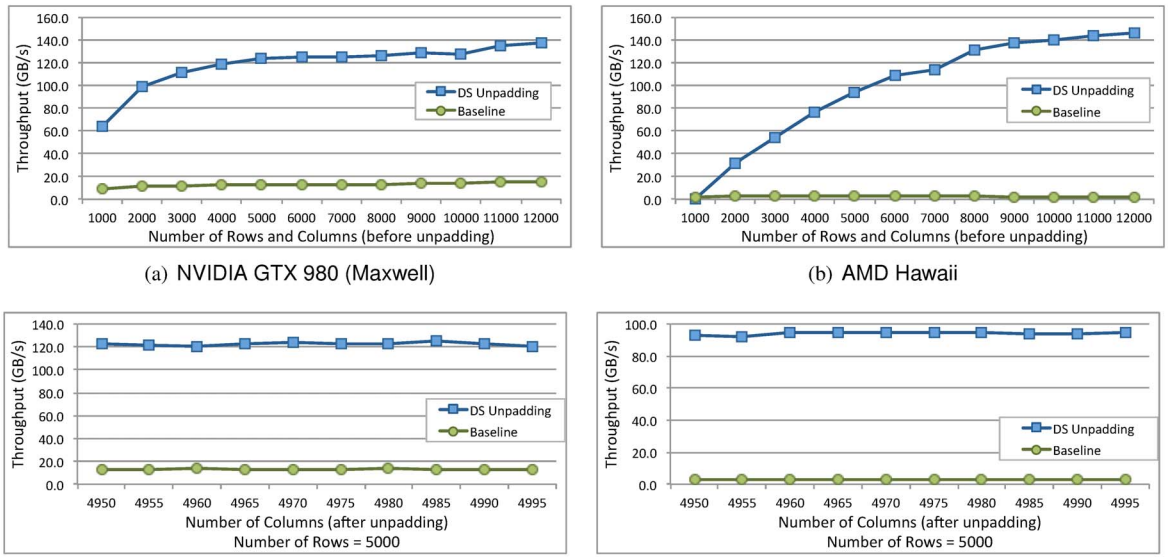
Experimental Evaluation. Figure 8 compares our



(a) NVIDIA GTX 980 (Maxwell) (b) AMD Hawaii

(c) NVIDIA GTX 980 (Maxwell) (d) AMD Hawaii

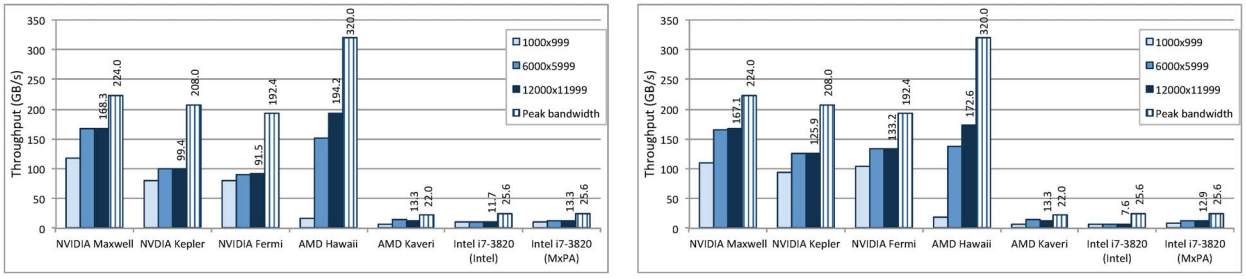
Figure 8. Throughput (GB/s) of DS Padding for single precision on Maxwell and Hawaii: (a) and (b) show the throughput for padding one column; (c) and (d) show the throughput for a varying number of padded columns. Baseline can use more than one work-group when there is enough extra space.



(a) NVIDIA GTX 980 (Maxwell) (b) AMD Hawaii

(c) NVIDIA GTX 980 (Maxwell) (d) AMD Hawaii

Figure 9. Throughput (GB/s) of DS Unpadding for single precision on Maxwell and Hawaii: (a) and (b) show the throughput for unpadding one column; (c) and (d) show the throughput for a varying number of padded columns. Baseline always uses one work-group.



(a) DS Padding (b) DS Unpadding

Figure 10. Throughput (GB/s) of OpenCL versions of DS Padding (a) and DS Unpadding (b) for double precision on NVIDIA GPUs, AMD GPUs and Intel CPU with two compilers (MxPA and Intel). The graphs show the throughput for padding or unpadding one column. Matrix sizes are in the legend. Figures present the results on Fermi, Kepler, Maxwell, Hawaii, Kaveri, and Intel CPU with MxPA and Intel compilers.

OpenCL version of DS Padding to the baseline [11] for single precision on NVIDIA and AMD GPUs. In Figures 8(a) and 8(b), the DS Padding is up to 8× faster than the baseline on Maxwell, and up to 63× on Hawaii. As in these tests only one column is padded, the limited amount of extra space constricts the number of concurrently executing work-groups in the baseline, causing the poor performance.

On the contrary, our DS Padding is completely independent of the extra memory space. This is observed in Figures 8(c) and 8(d), which compare DS Padding and the baseline for a variable number of extra columns. Note that the number of columns after padding is the same as the number of rows, which is set to 5,000 in Figures 8(c) and 8(d). Thus the number of columns before padding determines the amount of extra space added. The lower the number of extra columns, the lower the throughput of the baseline, because the number of work-groups that can run concurrently decreases. Thus, the speedup of DS Padding is between 1.95× and 7.32× on Maxwell, and between 6.45× and 29.71× on Hawaii. Similar results are shown in Figure 9 for the unpadding primitive. Here, the throughput of the baseline does not depend on the number of removed columns, because only one work-group is used in all cases. In Figures 9(a) to 9(d), we observe a speedup up to 9.11× on Maxwell, and up to 73.25× on Hawaii.

Figure 10 shows the throughput of the OpenCL version of our DS Padding and DS Unpadding for double precision on the NVIDIA, AMD GPUs, and Intel CPU. The throughput on the AMD GPUs is more dependent on the size of the matrix. MxPA compiler outperforms Intel compiler on the Intel CPU. Our DS Padding and Unpadding obtain a very significant fraction of peak bandwidth on all these architectures. For instance, the peak memory bandwidth of Maxwell is 224 GB/s, so our primitives achieve up to 75% of that peak. On Fermi and Kepler, up to 50% is attained. The peak bandwidth on Hawaii is 320 GB/s, so around 60% of the peak is achieved. Similar fraction is obtained on Kaveri.

On the Intel CPU, the peak memory bandwidth is 25.60 GB/s. Thus, more than 50% of that peak is obtained when MxPA compiler is used. Additionally, in order to compare to a CPU baseline, we run sequential versions of padding and unpadding on the Intel CPU. With MxPA, our DS Padding is 2.80× faster, and our DS Unpadding is 2.45× faster.

B. Select

Given a source array, the *select* primitive filters elements according to the evaluation of a certain predicate. One version removes the elements satisfying the predicate, while another one keeps them. Figure 11 shows a *select* operation with a predicate *element value is even*.

This is an irregular Data Sliding algorithm, because the number of positions that each element is shifted depends on the number of elements that satisfy (or do not satisfy) the predicate before it. Thus, it follows Algorithm 2. A particular

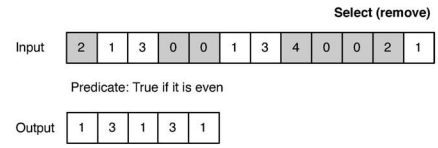


Figure 11. *Select* operator on an array. This version removes elements satisfying the predicate *element value is even*. Thus, the output array contains only the elements that do not satisfy it.

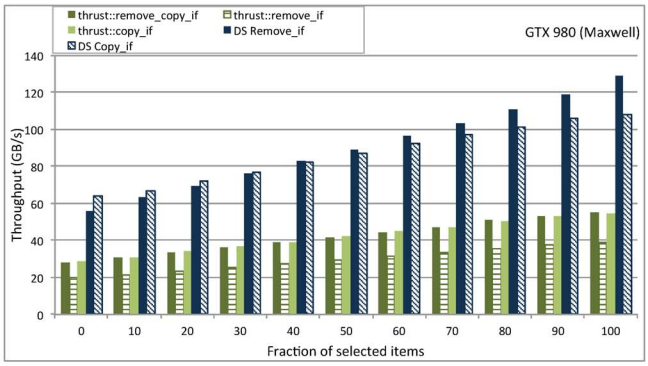


Figure 12. Throughput of *select* primitives on Maxwell. Our DS Remove_if (in-place) and DS Copy_if (out-of-place) primitives are compared to Thrust's in-place (`thrust::remove_if`) and out-of-place (`thrust::remove_copy_if` and `thrust::copy_if`). The size of the input array is 16M elements (single precision).

case of *select* is stream compaction, which eliminates those elements of the array that are equal to a certain value.

Experimental Evaluation. Two select-like primitives are tested in Figure 12. One of them removes array elements that satisfy a certain predicate. This can operate in-place or out-of-place. The other one copies elements that satisfy a predicate to another array (out-of-place). The percentage of elements that satisfy the predicate is between 0 and 100 in steps of 10. The CUDA versions of our DS select primitives are compared to NVIDIA Thrust implementations [12] on Maxwell. Our DS select primitives use shuffle instructions to improve the performance of reduction and binary prefix sum, as explained in Section III-B. They outperform Thrust by factors up to 2.15-3.50×.

Figure 13 shows the performance difference for the stream compaction operator using multiple existing optimizations on Maxwell. We compare to Thrust's out-of-place (`thrust::remove_copy`) and in-place (`thrust::remove`) implementations. As a reference, three out-of-place, *unstable* methods using atomic operations [22] are also tested. Two of them use aggregated atomic operations on shared memory or global memory, in order to reduce atomic contention. These methods are only useful in applications where stability is not required, that is, it is not necessary to maintain the relative order of input elements. Our DS Stream Compaction is more than 3.2× faster than the corresponding Thrust's in-place method, and achieves up to 68% the throughput of the fastest out-of-place, unstable method.

Figure 14 shows the throughput of the OpenCL version of

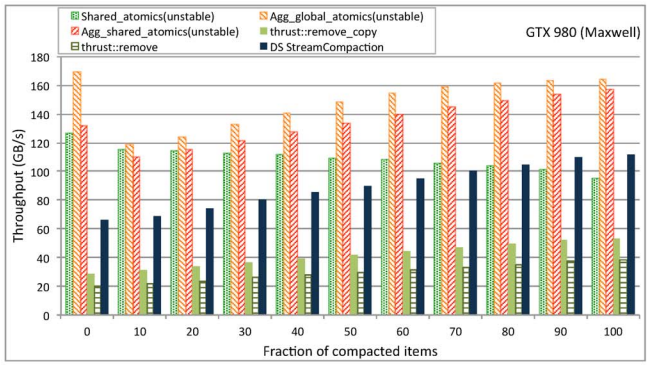


Figure 13. Throughput of stream compaction on Maxwell. Our DS Stream Compaction (in-place) is compared to Thrust's in-place (`thrust::remove`) and out-of-place (`thrust::remove_copy`) methods. Out-of-place, unstable methods using atomic operations are also presented for reference. The size of the input array is 16M elements (single precision).

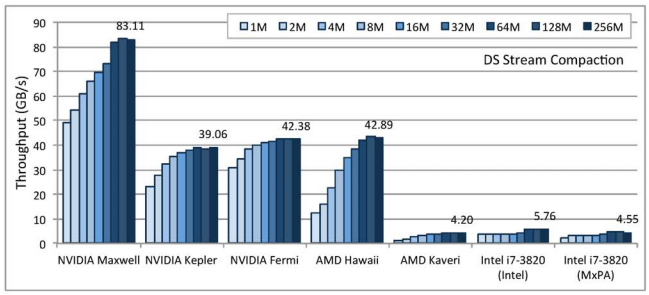


Figure 14. Throughput (GB/s) of OpenCL versions of DS Stream Compaction on Fermi, Kepler, Maxwell, Hawaii, Kaveri, and Intel CPU compiled with MxPA and Intel compilers. The legend presents array sizes (M elements, single precision). The percentage of compacted elements is 50%.

our DS Stream Compaction on the NVIDIA, AMD GPUs, and Intel CPU. By using optimized reduction and binary prefix sum (see Section III-B), it is possible to achieve a further 7% to 40% improvement on the GPUs. The intra-warp shuffle instructions can be emulated through local memory for devices lacking of them, such as NVIDIA Fermi and AMD GPUs. We observe Kepler delivers less throughput than Fermi in OpenCL Stream Compaction, which does not happen in the corresponding CUDA results (not shown in the figure, but summarized in Table I in Section VI). A possible reason is that Kepler, unlike Fermi, does not support L1 cache for global memory accesses, which is critical for irregular memory accesses. Another possible reason is that shuffle instructions are supported in CUDA but not in OpenCL.

C. Unique

The *unique* operator works as explained in Figure 15. For each group of consecutive elements that are equal in the input array, it removes all but the first of these consecutive elements.

While a work-item loads an array element, it compares

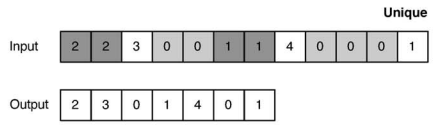


Figure 15. *Unique* operator on an array. For each group of consecutive elements with the same value, only the first of them is kept.

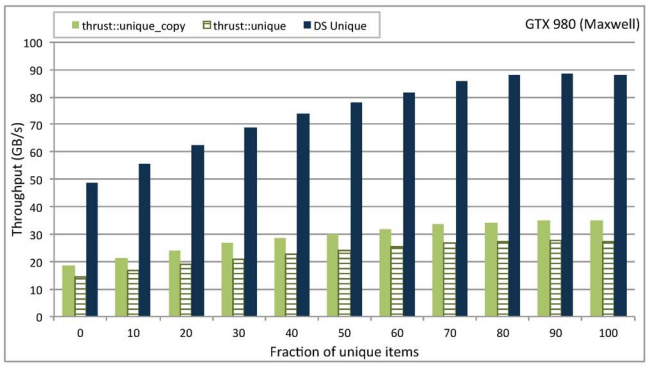


Figure 16. Throughput of *unique* primitives on Maxwell. Our DS Unique (in-place) primitive is compared to Thrust's in-place (`thrust::unique`) and out-of-place (`thrust::unique_copy`) methods. The size of the input array is 16M elements (single precision).

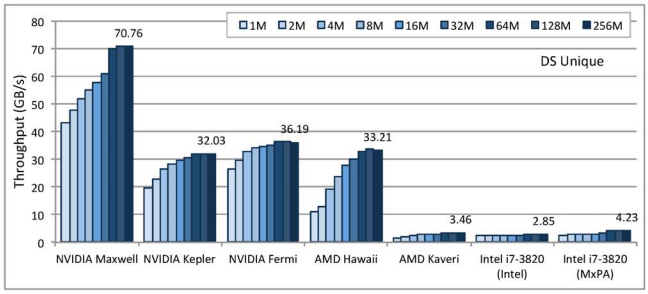


Figure 17. Throughput (GB/s) of OpenCL versions of DS Unique on Fermi, Kepler, Maxwell, Hawaii, Kaveri, and Intel CPU compiled with MxPA and Intel compilers. The legend presents array sizes (M elements, single precision). The percentage of unique elements is 50%.

the current element with the previous adjacent one. To implement this stencil, we apply `__shfl_up` instruction can be used for NVIDIA devices of c.c. 3.0 or higher (otherwise, it can be emulated through local memory), while on the boundaries we directly access global memory.

Experimental Evaluation. Figure 16 shows the experimental results for our CUDA DS Unique operator on Maxwell. We compare to Thrust's out-of-place (`thrust::unique_copy`) and in-place (`thrust::unique`) implementations. The speedup of our in-place DS Unique to Thrust's out-of-place version is more than 2.70×, and more than 3.47× to Thrust's in-place version. Meanwhile, Figure 17 shows the throughput of our OpenCL DS Unique on the NVIDIA, AMD GPUs, and Intel CPU. An additional 6% to 28% of the throughput can be obtained with the optimized reduction and binary prefix sum (Section III-B). Similarly, Kepler still has less throughput than Fermi in OpenCL.

D. Partition

The partition operator classifies the elements of a source array according to the evaluation of a predicate. The elements that satisfy the predicate are placed in the first part of the array, preceding the elements that do not satisfy the predicate. An example is shown in Figure 18.

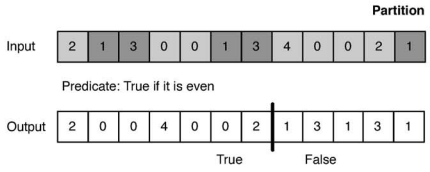


Figure 18. Partition operator on an array. The elements that satisfy the predicate are placed in the beginning of the array, while the rest of elements are moved to the tail of the array.

Due to the fact that two classes of elements are handled, two local counters per thread are used, in order to count the number of $true$ and $false$ elements. Two output arrays are needed. $Array\_true$ can be the same input array $Array$. $Array\_false$ should be an auxiliary buffer. For an in-place partitioning operation, it will be necessary to move the $false$ elements from the auxiliary buffer to the tail of $Array$, once the $true$ elements have been shifted. Although the in-place partition operation itself does not explicitly satisfy the condition of unidirectional data movement, considering the intermediate operations of $Array\_true$ and $Array\_false$, we still recognize it as a benchmark of DS algorithms.

Experimental Evaluation. Figure 19 shows the experimental results for our DS Partition operator. Our DS Partition has an out-of-place and an in-place version. The throughput of the in-place version increases with the number of true elements, since the amount of false elements to move from the auxiliary buffer is smaller.

We compare to Thrust's out-of-place (`thrust::stable_partition_copy`) and in-place (`thrust::stable_partition`) implementations. Thrust also include *unstable* versions (`thrust::partition` and `thrust::partition_copy`), that actually give very similar results to the stable versions. Our out-of-place version is 3.02× faster than Thrust's out-of-place codes. Our in-place version is at least 2.16× faster than Thrust's out-of-place, and 3.15× faster than Thrust's in-place.

Figure 20 shows the throughput of our OpenCL DS Partition on the NVIDIA, AMD GPUs, and Intel CPU. As the previous primitives, the throughput can be further increased by 10% to 45% using shuffle-based reduction and binary prefix sum. As in previous sections, Kepler has less throughput than Fermi in OpenCL.

V. Related Work

DS algorithms on many-core architectures have not been tackled as a whole class of algorithms, but there are several attempts to deal with some of them. Sung [11] developed an implementation of padding that can concurrently run several

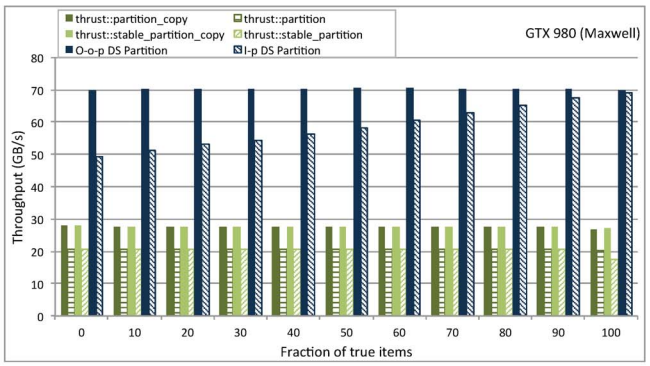


Figure 19. Throughput of partition primitives on Maxwell. Our DS Partition (in-place and out-of-place) primitives are compared to Thrust's stable in-place (`thrust::stable_partitition`) and out-of-place (`thrust::stable_partition_copy`), and unstable in-place (`thrust::partitition`) and out-of-place (`thrust::partition_copy`) methods. The size of the input array is 16M elements (single precision). The percentage of true elements is 50%.

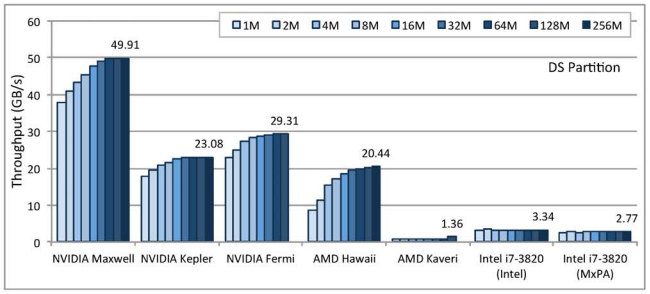


Figure 20. Throughput (GB/s) of OpenCL versions of DS Partition on NVIDIA GPUs, AMD GPUs and Intel CPU with two compilers (MxPA and Intel). Figure (a) presents the results on Fermi, Kepler, Maxwell, Hawaii and Kaveri, and Intel CPU compiled with MxPA and Intel compilers. The legend presents array sizes (M elements, single precision).

work-groups in the initial iterations, while there is enough empty space. Then, it necessarily runs only one work-group that moves rows sequentially. The opposite approach could be used for unpadding: sequential operation in the initial iterations, and some concurrent work-groups when some space appears after moving lower-indexed rows. We have shown our DS approaches can clearly outperform these implementations.

NVIDIA Thrust library [12] includes some of the DS primitives, such as stream compaction, select, partition, and unique. Same as other works [4], the implementation of these primitives requires the use of several kernels. This entails a high cost due to repeated global memory loads and stores.

Regarding work-group synchronization, there have been very few attempts to design mechanisms that allow work-groups to coordinate within a kernel [23]. The adjacent work-group synchronization we use in this work is similar to the one presented in [14][15].

VI. Conclusions

We present a generic parallel in-place DS algorithm for many-core architectures. We demonstrate our OpenCL DS

Table I
SUMMARY OF IN-PLACE, SINGLE-PRECISION EXPERIMENTAL RESULTS: 12000×11999 MATRIX SIZES AND 1 PADDED COLUMNS ON *OpenCL* PADDING AND UNPADDING AND 16M ARRAY SIZES AND 50% FRACTION OF ELEMENTS ON *CUDA* SELECT, UNIQUE, AND PARTITION, WITH SHUFFLE-OPTIMIZED REDUCTION AND BINARY PREFIX SUM.

Primitive	Device	DS(GB/s)	Thrust(GB/s)	Sung's(GB/s) [11]	Speedups
Padding	Maxwell	131.53	–	16.23	8.10
	Hawaii	168.58	–	2.66	63.31
Unpadding	Maxwell	137.13	–	15.05	9.11
	Hawaii	146.79	–	2.00	73.25
Select	Maxwell	87.34-89.21	42.15-29.27	–	2.07-3.05
	Kepler	47.66-52.26	18.76-18.69	–	2.54-2.80
	Fermi	42.68-42.69	23.99-24.31	–	1.76-1.78
Unique	Maxwell	78.10	24.04	–	3.24
	Kepler	38.88	14.26	–	2.73
	Fermi	29.93	18.01	–	1.66
Partition	Maxwell	58.34	20.56	–	2.84
	Kepler	37.41	13.01	–	2.88
	Fermi	27.21	16.57	–	1.64

implementations for *padding* and *unpadding* outperform the existing work in both NVIDIA and AMD GPUs by a factor of up to 9.11× and 73.25×, respectively. Our CUDA implementations of `remove`, `copy`, `unique`, and `partition` using the irregular DS algorithm outperform the state-of-the-art NVIDIA Thrust library on multiple NVIDIA GPUs by a factor of up to 3.24×. Most interesting throughput results on GPUs are summarized in Table I. We also show our algorithm can perform well even in multicore CPU environments using the state-of-the-art OpenCL stacks.

ACKNOWLEDGEMENT

We thank the Starnet Center for Future Architecture Research (C-FAR), the UIUC CUDA Center of Excellence, the University of Málaga CUDA Research Center, the Junta de Andalucía of Spain (TIC-1692), and the Government of Spain (TIN2013-42253P).

REFERENCES

[1] A. Cano, A. Zafra, and S. Ventura, "Speeding up the evaluation phase of GP classification algorithms on GPUs," *Soft Computing*, vol. 16, no. 2, pp. 187–202, 2012.

[2] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems," in *Proceedings of the 23rd International Conference on Supercomputing*, 2009, pp. 244–255.

[3] G. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili, "Relational algorithms for multi-bulk-synchronous processors," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, pp. 301–302.

[4] I. Saeed, J. Young, and S. Yalamanchili, "A portable benchmark suite for highly parallel data intensive query processing," in *Proceedings of the 2nd Workshop on Parallel Programming for Analytics Applications*, 2015, pp. 31–38.

[5] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," *Computer Graphics Forum*, vol. 28, no. 2, 2009.

[6] D. Roger, U. Assarsson, and N. Holzschuch, "Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the GPU," in *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, 2007, pp. 99–110.

[7] J. Gómez-Luna, J. M. González-Linares, J. Ignacio Benavides, E. L. Zapata, and N. Guil, "Load balancing versus occupancy maximization on graphics processing units: The generalized hough transform as a case study," *International Journal of High Performance Computing Applications*, vol. 25, no. 2, pp. 205–222, 2011.

[8] J. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-M. Hwu, and N. Obeid, "Algorithm and data optimization techniques for scaling to massively threaded systems," *Computer*, vol. 45, no. 8, pp. 26–32, 2012.

[9] B. Catanzaro, A. Keller, and M. Garland, "A decomposition for in-place matrix transposition," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2014, pp. 193–206.

[10] J. Gómez-Luna, I. Sung, L.-W. Chang, J. González-Linares, N. Guil, and W.-M. W. Hwu, "In-place matrix transposition on gpus," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[11] I.-J. Sung, "Data layout transformation through in-place transposition," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 2013.

[12] N. Bell and J. Hoberock, "Thrust: a productivity-oriented library for CUDA," *GPU Computing Gems: Jade Edition*, 2012.

[13] M. Dow, "Transposing a matrix on a vector computer," *Parallel Computing*, vol. 21, no. 12, pp. 1997 – 2005, 1995.

[14] S. Yan, G. Long, and Y. Zhang, "StreamScan: Fast scan algorithms for GPUs without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, pp. 229–238.

[15] L.-W. Chang, "Scalable parallel tridiagonal algorithms with diagonal pivoting and their optimization for many-core architectures," Master's thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 2014.

[16] 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, pp. 205–213.

[17] M. Harris, "Optimizing parallel reduction in CUDA," in *NVIDIA CUDA SDK*. NVIDIA, 2007.

[18] G. E. Blelloch, "Prefix sums and their applications," Carnegie Mellon University, Technical Report CMU-CS-90-190, 1990.

[19] M. Harris and M. Garland, "Optimizing parallel prefix operations for the Fermi architecture," *GPU Computing Gems: Jade Edition*, 2012.

[20] Julien Demouth, "Kepler shuffle: Tips and tricks," in *GPU Technology Conference*, 2013.

[21] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015, pp. 257–268.

[22] Andrew Adinetz, "CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics," 2014, http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.

[23] S. Xiao and W.-c. Feng, "Inter-block GPU communication via fast barrier synchronization," in *Parallel Distributed Processing, 2010 IEEE International Symposium on*, 2010, pp. 1–12.