Fast Radix Sort for Sparse Linear Algebra on GPU

Lukas Polok, Viorela Ila, Pavel Smrz
{ipolok,ila,smrz}@fit.vutbr.cz
Brno University of Technology, Faculty of Information Technology.
Bozetechova 2, 612 66 Brno, Czech Republic

Keywords: Parallel sorting, radix sort, sparse matrix, matrix-matrix multiplication

Abstract

Fast sorting is an important step in many parallel algorithms, which require data ranking, ordering or partitioning. Parallel sorting is a widely researched subject, and many algorithms were developed in the past. In this paper, the focus is on implementing highly efficient sorting routines for the sparse linear algebra operations, such as parallel sparse matrix - matrix multiplication, or factorization. We propose a fast and simple to implement variant of parallel radix sort algorithm, suitable for GPU architecture.

Extensive testing on both synthetic and real-world data shows, that our method outperforms other similar state-of-the-art implementations. Our implementation is bandwidth-efficient, as it achieves sorting rates comparable to the theoretical upper bound of memory bandwidth. We also present several interesting code optimizations relevant to GPU programming.

1. INTRODUCTION

Efficient parallel sorting is an important building stone of many algorithms. Although parallel sorting algorithms have been researched extensively in the past, implementing the same algorithms on GPU presents a significant challenge, due to the necessary amount of communication and synchronization, not to mention high irregularity of memory accesses. In this paper, a highly efficient implementation of efficient radix sort is discussed. The ultimate goal is to support sparse linear algebra calculations, where sorting is often employed as a preprocessing step of matrix compression [1] in order to improve load balancing and to increase utilization [2] of parallel processors. On Fig. 1 you can see that sorting takes a substantial portion of execution time of the current sparse matrix multiplication algorithms, running on GPU.

Sparse matrix multiplication is characteristic by *scattering* the elementwise products in not easily predicted pattern. In order to be efficient, it must calculate products in the order in which the matrices are stored (such as compressed sparse column). When implemeted in parallel, this scattering would cause a lot of conflicts where different threads would require access to the same element of the output matrix. To resolve this, the current implementations calculate the product as a

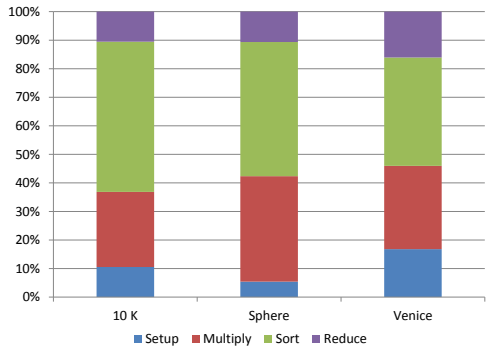

Figure 1. Relative amount of time spent in different phases of sparse matrix multiplication on GPU.

set of destination coordinates and associated values, which are then sorted and compacted.

This puts the problem in a different perspective: the data to be sorted is produced by the GPU (e.g. by a matrix multiplication routine), and the sorted results are consumed by the GPU. Therefore we are not burdened by having to transfer the data between CPU and GPU, much to the contrary: the conventional approach would be to only use GPU for large - enough problems and to process small problems on the CPU. In our perspective, such processing would involve the prohibitive cost of data transfers and CPU - GPU synchronization. On the other hand, there is some prior knowledge about the range and distribution of the sorted data. Our algorithm is able to use such knowledge to significantly accelerate the sorting, but still remains general.

When comparing the state-of-the-art GPU accelerated libraries that provide sorting functionality, there is a significant performance gap: implementations based on CUDA achieve about twice the sorting rates of the OpenCL-based ones. The proposed implementation is intended to show that efficient sorting can be implemented even without advanced features exposed by CUDA, such as dynamic parallelism or thread voting. The proposed approach outperforms all of the compared implementations.

2. RELATED WORK

Some of the first attempts on efficient sorting on GPU [3], [4], [5] were implemented using the programmable shading pipeline, and were based on sorting network [6] approach. Govindaraju et. al. [7] extended the idea to fully utilize the

vector pipeline of the shading units and implemented a large-scale out-of-core sorting. The obvious disadvantage is a considerable overhead of using a graphics API, but general-purpose computing API did not exist yet. Sorting networks furthermore require relatively large number of passes, which grows with the size of the sorted sequence. These passes required communication through global (texture) memory, and the upper bound of performance was relatively low.

One of the first influential sorting implementations in CUDA, Satish et. al. [8] proposed to use the radix sort algorithm. Their method processed data in four passes that included local block sorting using 1-bit split operations [9], local histogram calculation, global prefix sum over histograms and finally reordering the data. Although this method is similar to the Algorithm 1 from this paper, it is not optimal. The first local block sorting step was intended in order to improve memory access patterns in the last scattering step, which can have detrimental effects on performance if not properly handled. However, such sorting is not work efficient.

Sintorn et. al. [10] was able to develop a method based on a combination of merge sort and bucket sort. The bucket sort is used to improve parallelism at the later stages of sorting, where the number of lists to be merged becomes lower than the number of parallel processors. Their implementation, although based on comparison sorting algorithms, outperformed the work of Satish [8] for arrays of 8 M elements, or more. One disadvantage of this method is the use of atomic counters to perform the bucket sort, and as such it depended on the distribution of the sorted data, as atomic operations on the same counter are subject to serialization in many parallel architecture, including GPUs.

The efficiency of radix sorting was improved by Ha et. al. [11] by focusing on the arithmetic intensity of the sorting. To reduce the number of arithmetic operations in sorting, several optimizations such as accumulating three 10-bit histogram bins in a single 32-bit integer or use of mixed-data structure are applied. It is based on the observation that bigger value types suffer less from irregular memory access pattern at the scatter phase. Therefore, array of key-value structures is preferred for this step, rather than the usually used structure of arrays. As a result, about 30 % greater sorting rate is achieved, compared to the Satish [8] implementation.

Currently the fastest state-of-the-art implementation is that of Merrill [12] and [13], which was greatly influential also to our method. They build on work of Satish [8] and also use the idea of accumulating four 8-bit histogram bins in a single 32-bit integer. Several novel ideas are introduced in these works, one of the most important ones being the reduction of number of steps per radix to three, as in Algorithm 1 where lines 3, 4 and $5-9$ can run each as one step that only requires global communication at the beginning or at the end. This reduction in global communication effectively increases the upper bound on sorting throughput. It is made possible by performing local sorting at the end of the scattering step, where it can be done in work-efficient manner.

The remainder of the paper is structured as follows. The next section introduces nonlinear least squares problem as the motivation of this work. Section 4. details the proposed implementation and optimizations used. Section 5. shows the performance of the proposed solution through benchmarks and time comparisons with the exiting implementations. Conclusions and future work are given in Section 6.

3. MOTIVATION

Many sparse numerical applications ranging from physics, computer graphics, computer vision to robotics rely on efficiently solving large systems of equations. In case of nonlinear systems, the solution can be approximated by incrementally solving a series of linearized problems. In some applications, the size of the system considerably affects performance. The most computationally demanding part is to solve the linearized system at each iteration.

A matrix is called *sparse* if many of its entries are zero. Considering the sparsity of the matrices can bring important advantages in terms of storage and operations. Some of the existing implementations of nonlinear solvers rely on fast sparse linear algebra packages for solving the linearized system. Here is where CSparse [14] or CHOLMOD [15] libraries are used to perform the matrix factorization. Similar libraries have also been developed for GPUs.

In our previous work [16], we proposed a fast and cache efficient data structure for sparse *block* matrix representation, which takes advantage of the block structure naturally occuring in many of the nonlinear least squares problems, and showed its advantages in nonlinear least square applications [17], [18]. The data structure enables simple matrix modification, be it structural or numerical, while also maintaining, and often even exceeding the speed of element-wise operations schemes.

In order to accelerate the same scheme on GPU, a fast linear solver needs to be implemented. Cholesky decomposition is suitable for parallelization, however it involves highly irregular memory access patterns and sequential dependences, and as such would not scale well [19] on a GPU. In order to get arround this problem, we propose the use of Schur complement [20]. Let us consider the following system of linear equations:

$$\begin{bmatrix} A & U \\ U^T & D \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{1}$$

Supposing that D is invertible, the Schur complement of the block D of the system matrix is:

$$A - UD^{-1}U^T \tag{2}$$

This can be used to solve the system of equations in the following manner:

$$(A - UD^{-1}U^T)x = a - UD^{-1}b \quad (3)$$

$$y = D^{-1}(b - U^T x) \quad (4)$$

Note that solving for x is done using Cholesky decomposition of the Schur Complement, followed by backsubstitution. Solving for y only involves sparse matrix-vector multiplication. This divides the problem of factorization of the whole system matrix to inversion of D and factorization of A-sized Schur Complement. It is possible to use *maximum independent set* to reorder the original system matrix, in order to make D a diagonal matrix. Inverting D is then reduced to inverting its elements, and is embarrasingly parallel. The rest of the computation is then involved in matrix multiplication for calculating the Schur Complement, and in factorization of a smaller, but possibly more dense matrix.

Since sparse matrix multiplication scales reasonably well on GPUs [1], [2], we argue that performing Schur Complement will improve scaling of linear solving on GPU as well. Fast sorting operations are required in both calculating the approximate maximum independent set and matrix multiplication. As shown in section 5., there is a performance gap between OpenCL and CUDA sorting implementations. This paper proposes a sollution that closes this gap.

4. PROPOSED IMPLEMENTATION

The next section contains a brief description and performance analysis of the radix sort algorithm, followed by a detailed description of our implementation and the methods used for optimizing it. Our algorithm consists of the same three steps as [13], but they are executed on GPU in just two steps, for reasons described below. Although our implementation is written in OpenCL, the design considerations are with respect to NVIDIA hardware, and when referring to some particular hardware specifics, it is that of the NVIDIA platform, unless specified otherwise.

In the proposed algorithm design, the emphasis is on low arithmetic density, taking advantage of OpenCL just-in-time compilation model for flexible scheduling, and parallel programming with minimal synchronization using *warp-synchronous* programming where possible.

4.1. The Radix Sort Algorithm

Radix sort [21] is a stable sorting algorithm, which is suitable for sorting keys that map to integral values, such as integers or to certain extent the floating-point values. Note that this is converse to the widely used sorting paradigm that uses a comparison predicate, and is implemented in e.g. C++ Standard Template Library. It works by grouping the given integer keys by their corresponding digits. This is done in successive fashion, starting with the least significant digits. Once grouped, the keys are then read out, starting with the group corresponding to the lowest value and maintining relative order of the keys in the same group. After going through all of the digits, the sequence is sorted. The parallel version of this algorithm, called split radix sort [9], relies on parallel prefix sum primitive extensively, to facilitate grouping of the sorted elements. Parallel prefix sum, or *scan*, can be implemented efficiently on GPU [22]. In order to extend radix sort algorithm to run efficiently on multi-processor machines such as GPUs, a notion of segments [9] is introduced. The sort can be broken down to local operations on the individual segments of the input sequence, which can be performed with reduced amount of communication between processors, working on different segments. The final sorting algorithm is described in Algorithm 1. A similar algorithm was used in [13].

Algorithm 1 Segmented Parallel Radix Sort.

```
1:  function RADIXSORT(input)
2:      for each digit_place in {LSB, ..., MSB} do
3:          Calculate segment histogram of digits at digit_place
4:          Inplace global scan of all the histograms
5:          begin
6:              Segment scan of counts of digits at digit_place
7:              Add histogram scan to get global offsets
8:              Scattter temp ← input
9:          end
10:         Swap input ↔ temp
11:     end for
12:     return input
13: end function
```

In the first step inside the loop, counts of digit values in each segment of the input sequence are calculated. Prefix sum of those counts gives the global position of the first occurrence of each digit in the output sequence, for each segment. Finally, the last step will calculate prefix sums of each digit, determining output position of each key in terms of the segment and by using the histogram prefix sum also the global output position. The output sequence of one loop iteration becomes input to the next one, output of the last iteration is the sorted sequence. In order to sort k-bit numbers, one needs to perform k/d iterations of the loop above, where d is size of a digit, in bits. Each segment histogram will therefore contain 2^d bins. An example of a single step of the loop is depicted on Fig. 2.

Since sorting is certainly a bandwidth-limited operation, let us analyze the cost in terms of memory accesses. Given that the length of input sequence is n, and the hardware architecture dictates us to use m segments (where each segment corresponds to an individual parallel processor), the required bandwidth can be found in Table 1.

Since m is quite limited by the hardware (up to tens on GPUs, or hundreds on Intel MIC), and d is limited by regis-

Line of algorithm	Memory reads	Memory writes
3	n	$2^d m$
4	$2^d m$	$2^d m$
$5-9$	$n+2^d m$	n

Table 1. Memory Complexities of Algorithm 1, the Segmented Radix Sort

ter pressure, the memory complexity is roughly $3nk/d$. This can give us an idea about the upper bound of the sorting rates achievable on the current hardware. For example, NVIDIA GeForce GTX 780 has maximum bandwidth of 288.4 GB/sec, which can yield peak sorting rates up to 3.0 GKeys/sec for the common case of $k = 32$, $d = 4$. The proposed implementation is efficient, in the sense of achieving performance, comparable to this upper bound. Note that in the following text, the convention of binary units is used, where 1 M equals 1024^2, 1 G equals 1024^3, and so on.

4.2. Segmented histogram calculation

Histogram calculation is fairly straightforward algorithm if implemented on a serial processor. On a parallel processor, two common approaches prevail. Sintorn [10] used atomic operations for incrementing the histogram bin counters, but despite recent architectural improvements, atomic operations still serialize if working on the same variable (the same histogram bin). The efficiency of histogram accumulation is then heavily dependent on the data, and is reduced up to $32\times$ on NVIDIA platforms in the worst case (since threads execute in groups of 32, called *warps*), or even slower on AMD platforms (similarly, threads execute in *waveforms* of 64 threads).

The other solution, which our implementation uses, is to trade time for space, having each thread accumulate in its private histogram, and have the threads reduce the histograms at the end. Segmented histogram is highly advantageous for GPU implementation, as there is no communication between the segments, and the reduction can take place entirely in the fast shared memory. The size of the segments is of great importance, as it affects performance greatly. If the segments are too small, the costs of each thread initializing its private histogram with zeros and of the final reduction will easily outweigh the time, spent in the actual accumulation of values, rendering the calculation inefficient. If, on the other hand, the segments are too large, there may not be enough segments to occupy all the streaming multiprocessors of the GPU. Many of the previous implementations restrict the size of the segment to a constant, implementation of Satish [8] is an example, it uses *tiles* of 1024 items. Instead, our implementation, similarly to that of Merrill [13] uses variable length segments. The number of segments is chosen as a minimum that can keep the GPU fully utilized.

A distinguishing feature of our algorithm is the choice of memory space for thread histogram storage. On GPU, there are several memory spaces with varying suitability. Global memory is mostly unsuitable for histogram accumulation, due to its latency. Shared memory is roughly two orders of magnitude faster, but it is accessed through a small amount of banks (16, or 32 on newer Fermi GPUs). If bank conflicts occur, the I/O operations are serialized. Therefore, even though not using atomic instructions, the accumulation would still be dependent on the data. Local memory [23] (not to be confused with local memory in OpenCL) is a memory space, specific to GPUs. It is a memory space, which is private to each thread. The values written to local memory space are stored in L1 cache, but can be evicted to L2 and eventually to *global* memory (highly likely for bandwidth-intensive applications). This memory space is used only for register spills and *addressable arrays*. This is due to the absence of register addressing. In vertex program specification, there is the ARL instruction, but its use is limited. That means that code like in Algorithm 2 will actually store values in global memory, and will be dependent on the data.

Instead, the proposed histogram algorithm accumulates the histogram in registers. Due to the nature of GPU execution model, to use branching to decide which histogram bin should be incremented would result in thread divergence, serialization and again dependence of execution speed on the data. On GPU, it is better to compare data at the input to all histogram bins, and use the results of the comparison to increment all the histogram bins, for every item of data. This approach, however, yields high arithmetic intensity and is only efficient if there are enough threads running to cover up the latency. Instead, bit masking operations are employed to calculate the comparison. That enables accumulation of several different values at once by simply or-ing their masks together. Special care needs to be taken for accumulating duplicate values. The final accumulation part is summarized in Algorithm 3.

Note that the $>>$ and $<<$ operators represent bitwise shift to the left and to the right, respectively, while $\cup$ and $\cap$ represents logical *or* and logical *and*. Also, the algorithm accumulates two symbols at once, and for the sake of simplicity does not handle the situation of odd-sized input. The code can be further optimized by sacrificing several bits of accumulator precision, and instead of performing 2^b shifts of *bin* (16 in Algorithm 3), only one shift (by 0 and by 8 bits) is used and the (constant) binary masks are shifted instead. That reduces

Algorithm 2 Naïve Histogram Calculation.

```
1: function HISTOGRAM(input)
2:     histogram[16] = {0, 0, ..., 0}
3:     for each i in input do
4:         histogram[i] ← histogram[i] + 1
5:     end for
6:     return histogram
7: end function
```

Algorithm 3 Register Histogram Accumulation.

1: **function** THREADPRIVATEHISTOGRAM(*input*)
2: $\{ha, hb, \ldots, hp\} = \{0, 0, \ldots, 0\}$
3: **for each** (i, j) in *input* **do**
4: $bin \leftarrow 1 << i$
5: $bin \leftarrow bin \cup (1 << j)$
6: $multiplicity_{log_2} \leftarrow i = j$
7: $ha \leftarrow ha + ((bin >> 0) \cap 1) << multiplicity_{log_2}$
8: $hb \leftarrow hb + ((bin >> 1) \cap 1) << multiplicity_{log_2}$
$\vdots$
9: $hp \leftarrow hp + ((bin >> 15) \cap 1) << multiplicity_{log_2}$
10: **end for**
11: **return** $\{ha, hb, \ldots, hp\}$
12: **end function**

the work to 26 simple instructions per accumulated value. The accumulators need to be shifted at the end but that is a small, constant overhead. Note that the maximum size of the input is not reduced, thanks to parallelism.

After thread private histograms have been calculated, the values need to be reduced. The first part of the reduction is dome in *warp-synchronous* manner, where each warp cooperatively reduces all its thread private histograms to a single histogram in shared memory. In order to completely avoid synchronization, each thread rotates its histogram bins by its id modulo 2^b. Afterwards, standard tree-based reduction is applied in shared memory. As a result, to reduce 512 histograms of 16 bins each, only four barrier synchronizations are required.

4.3. Fast Scan & Scatter

After accumulating the segment histograms, their prefix sum is calculated much like in [8], which will be used as a global destination offset for the sorted elements. Since the number of segments required to occupy the GPU is small, this step is not large enough to be efficiently issued as a separate kernel, and is fused with the last scattering step. Note that although this saves kernel execution, it does not save significant amount of communication and Table 1 still applies.

In order to perform scattering of the sorted sequence, global indices need to be calculated for each of the elements. Segmented prefix sum of histogram bin affiliation flags yields local ranks of the sorted elements. By adding value of histogram prefix sum for the corresponding segment and histogram bin, global position in the output sequence is obtained, as illustrated on Fig. 2. This requires us to calculate 2^b prefix sums, each of the size of the segment, or alternately more shorter prefix sums with carry.

Several interesting observations can be made. The prefix sums are of binary flags, and sum up to segment length. This gives us knowledge of how many bits are needed for the accumulators and it is possible to employ data-level parallelism.

Figure 2. An example of segmented radix split operation for $b = 1$, best viewed in color.

Ha [11] performs accumulation of three 10-bit accumulators, similarly Merrill [13] accumulates four 8-bit numbers in a single 32-bit variable.

It is possible to perform dynamic scheduling of accumulator precision in case of prior knowledge of the final sums, such as the histograms calculated in the first step of the algorithm. For example, a tile of 1024 element flags summed in 16 bins requires up to 128 bits, in 8 bins up to 64 bits. It is therefore possible to scan 1024×16 flags in two 64-bit numbers (always 8 in each), and if the distribution is favorable, a single 64-bit number suffices. This can be verified by solving unordered partition problem for 1024, 16. This however relies on calculating segment histograms for relatively small segments which reduces performance on the current GPUs, and we choose to use Merrill's method. This technique is, however, relevant to the future GPUs that would have more multiprocessors or more registers.

In the proposed implementation, each thread calculates local scan of two flags. Warp-synchronous prefix sums with carry are used to calculate segment scan. Threads working on a single segment exchange sorted elements in shared memory as in [13] and write them out to the temporary array.

4.4. Register usage optimization

One of the disadvantages of register histogram accumulation described in section 4.2. is the number of registers it uses (34 in our case). That directly affects possible number of workgroups, running on a single multiprocessor, and affects the capability to hide computational latency. In order to reduce the number of registers, a simple novel technique called *volatile stripping* is proposed. It is based on an observation that the OpenCL compiler allocates registers in a manner that will yield high processing speed, while the programmer has very little control over it. Declaring variables as *register* has no effect, and the compiler (NVIDIA 331.82) seems to ignore the '`-cl-nv-maxrregcount`' option.

In the histogram kernel, accumulation of the bins can be done in-place, but the compiler does not do that, possibly to improve pipelining. In our implementation, the histogram bin

variables are declared as *volatile*. That makes the compiler generate code for storing the value of the variables in local memory. A post-processing step is applied to the generated assembly code, which uniquely identifies each variable based on its address in local memory, strips all the volatile load and store instructions, and instead assigns a single register where the variable is stored. Using this technique, we were able to reduce register use from 34 down to 27, significantly improving occupancy.

Since this technique is rather low-level, and while general, currently only implemented for NVIDIA PTX assembly format thus creating platform dependence, it was disabled in the performance evaluation in order to make fair comparison to the other OpenCL implementations, which are platform independent. Although volatile stripping possibly damages software pipelining, increased occupancy results in roughly 10 % speedup for inputs of sufficient size to saturate the GPU memory subsystem.

5. PERFORMANCE ANALYSIS

In this section we compare the timing results of radix sorting performed using the proposed implementation with similar state of the art implementations such as CUDPP 2.1, Thrust 1.6.0, CLOGS 1.2.0, CLpp v1 beta 3 and libCL 1.2.1. All of those libraries use the radix sort algorithm. Some of them also implement predicate-based sorting, but it is slower than radix sorting, and therefore of no interest in our application. The evaluation was performed by sorting vectors of random numbers of varying lengths (the same sequences were used for all the implementations). We also performed evaluation on sequences, produced by multiplying sparse matrices from The University of Florida Sparse Matrix Collection [24]. This collection was chosen because it contains sparse matrices corresponding to a diverse set of problems, and as such it is suitable for testing of general purpose linear algebra implementations.

CUDA Data Parallel Primitives Library (CUDPP) is feature rich library with functions like parallel reduction, prefix sum, radix sorting, sparse matrix operations, random number generation and hashing. It supports comparison sorting, sorting optimized for strings, and radix sorting.

Back 40 computing (B40C) is another reusable parallel primitive library, developed in CUDA. It contains fast scalable radix sorting routines, designed arroung the allocation paradigm [25]. The B40C is now deprecated, the radix sorting code was reused in CUB and Thrust [26] libraries. We will focus on Thrust in our evaluation, as it is included in CUDA releases and is widely used. Thrust provides many functions, including predicate-based and radix sorting, with interface similar to the one of C++ Standard Template Libary.

CLOGS is a mature OpenCL implementation, providing scan and sort primitives. Sorting of any combination of scalar or vector key and value type is supported, as well as sorting only keys. The implementation is "loosely based" on Merrill's Back40Computing [13] radix sort implementation. CLOGS feature auto-tuning ability, which chooses the best parameters for target platform by exhaustively trying possible launch options, which are cached.

CLpp implements several sorting algorithms. Simple implementation of Radix Sort, as described by Blelloch [9], as described by Satish [8], and a generic version due to the authors of the library. It offers functions for sorting keys or key-value pairs. The size of the value can be configured, the keys are expected to be 32-bit unsigned integers. The default sort implementation, which is used in the benchmarks is based on the paper of Satish.

libCL only offers limited sorting capability: it can only sort key-value pairs, and only up to $4M - 1$ of them. Also, both key and value must be 32-bit types, and the key is compared as 32-bit unsigned integer, reducing usability for sorting floating-point numbers. There is no support for sorted type specification.

It is apparent that the CUDA implementations are of better quality, and are influential to the mostly inferior OpenCL implementations. This is in part given by the supported hardware features: CUDA naturally supports advanced NVIDIA hardware functions, such as dynamic parallelism or warp voting functions, which are unavailable in OpenCL. These features are used in the CUDA implementations, giving them a certain advantage. The one disadvantage of CUDA is that it is compiled for certain hardware profiles, and when a new platform emerges, the binary must be updated. This is not the case with OpenCL, where the programs are compiled at runtime and can therefore adapt to new hardware immediately. This adaptation is only limited to number of registers, size of memory and similar device parameters.

All the tests were performed on a computer with NVIDIA GeForce GTX 680 and GTX 780, a pair of AMD Opteron 2360 SE CPUs running at 2.5 GHz and 16 GB of RAM. Latest GPU drivers (version 331.82) were used. CUDA implementations were linked against CUDA 5.5 SDK libraries. During the tests, the computer was not running any time-consuming processes in the background. Each test was run

	GPU Type			
	GTX 680		**GTX 780**	
Library	**Key**	**Key-value**	**Key**	**Key-value**
CUDPP	689.752	538.849	804.798	590.816
Thrust	696.706	540.675	792.496	621.417
CLOGS	451.049	276.837	503.756	366.238
CLpp	134.716	94.245	154.076	122.487
libCL	N/A	85.106	N/A	98.655
proposed	**805.605**	**641.969**	**1119.422**	**892.055**

Table 2. Saturated sorting performance in MKeys/sec.

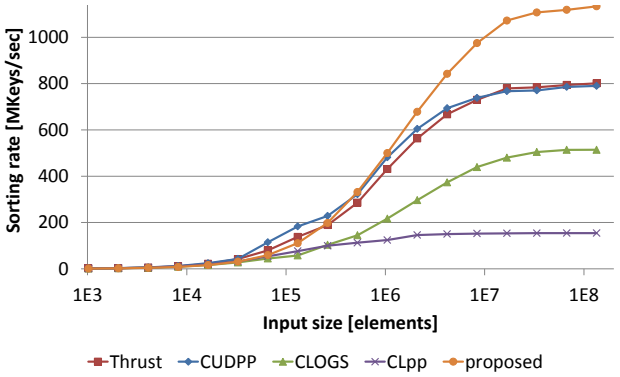

Figure 3. Sorting rates on 32-bit keys (higher is better).

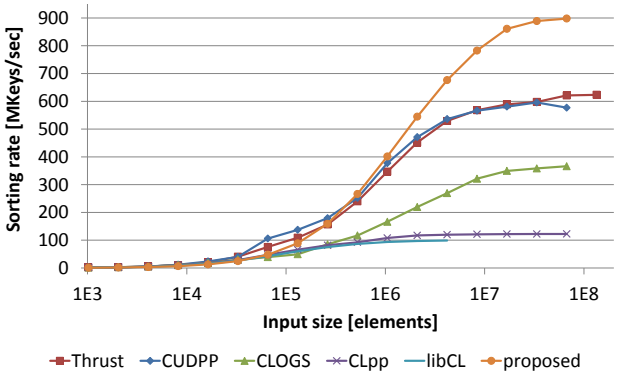

Figure 4. Sorting rates on 32-bit key-value pairs (higher is better).

at least ten times until cumulative time of at least 5 seconds was reached, and the average time was calculated in order to avoid measurement errors, especially on smaller sequences. Explicit CPU - GPU synchronization was always performed, using `cuCtxSynchronize()` or `clFinish()`, respectively. Recorded times do not include any data transfers. The computer was running Windows 7 (64 bit) and all the tested libraries were compiled using Visual Studio 2008 SP1.

Summative results can be found in Table 2. These were measured on random unsigned 32-bit numbers (care was taken so that the random numbers are not banded, but indeed span the whole 32 bits) and optionally 32-bit values. More detailed benchmarks are seen at Fig. 3 (keys only) and Fig. 4 (key-value pairs).

Since different implementations might react differently on the distribution of the sorted numbers, we also performed benchmarks by sorting element indices, obtained by performing sparse matrix multiplication, and recording destination row and column indices of results of every scalar product (see [1] for more details). Row and column indices are combined to a single key by multiplying column index by the number of rows and adding row index. Average runtime results on data generated by multiplying 160 of randomly chosen matrices from University of Florida Sparse Matrix Collection with

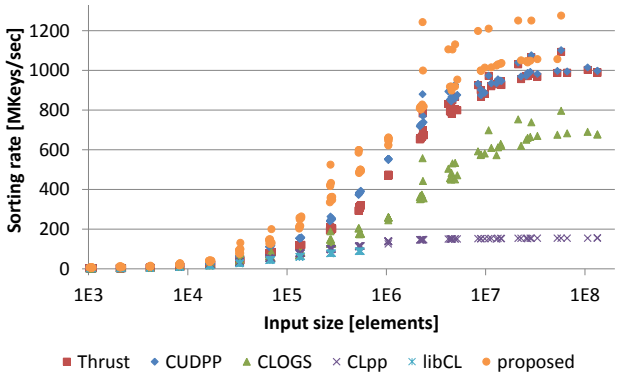

Figure 5. Sorting rates on 32-bit key-value pairs, keys were generated in sparse matrix multiplication (higher is better).

their respective transposes are plotted in Fig. 5. Note that the proposed implementation consistently gains the fastest saturated sorting rates, only outperformed by CUDPP for very short sequences.

Also note that authors of Thrust and CUDPP report greater sorting rates than measured, comparable with the proposed implementation. This is most likely due to the behavior on the particular GPU models, where our implementation is better optimized.

6. CONCLUSIONS AND FUTURE WORK

In this paper a simple portable radix sort implementation suitable for GPU was proposed. Although the achieved sorting rates are not much higher than the ones of the CUDA implementations, it improves over the fastest state-of-the-art OpenCL implementations by nearly 50 %. We achieved it by implementing fast histogram accumulation in registers, using warp-synchronous synchronization-free operation. We proposed a novel technique of *volatile stripping*. Another proposed technique of dynamic allocation of accumulator precision is currently less efficient than state-of-the-art, but will be applicable on bigger future GPUs.

We will focus on development of fast sparse linear algebra kernels using the proposed sorting implementation. The implementation is available as a part of our block matrix library, at `http://sourceforge.net/p/blockmatrix`.

7. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union, 7th Framework Programme grants 316564-IMPART and the IT4Innovations Centre of Excellence, grant n. CZ.1.05/1.1.00/02.0070, supported by Operational Programme Research and Development for Innovations funded by Structural Funds of the European Union and the state budget of the Czech Republic.

REFERENCES

[1] S. Dalton, N. Bell, and L. N. Olson, "Optimizing sparse matrix-matrix multiplication for the gpu," *Matrix*, vol. 3, p. 3c.

[2] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[3] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 41–50, Eurographics Association, 2003.

[4] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: a gpu-based particle engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 115–122, ACM, 2004.

[5] P. Kipfer and R. Westermann, "Improved gpu sorting," *GPU gems*, vol. 2, pp. 733–746, 2005.

[6] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 307–314, ACM, 1968.

[7] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 325–336, ACM, 2006.

[8] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–10, IEEE, 2009.

[9] G. E. Blelloch, *Vector models for data-parallel computing*, vol. 75. MIT press Cambridge, 1990.

[10] E. Sintorn and U. Assarsson, "Fast parallel gpu-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1381–1388, 2008.

[11] J. Ha, J. Krüger, and C. T. Silva, "Implicit radix sorting on gpus," *GPU GEMS*, vol. 2, 2010.

[12] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for gpgpu stream architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 545–546, ACM, 2010.

[13] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[14] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, 2006.

[15] T. A. Davis and W. W. Hager, "Modifying a sparse cholesky factorization," 1997.

[16] L. Polok, V. Ila, and P. Smrz, "Cache efficient implementation for block matrix operations," in *Proceedings of the 21st High Performance Computing Symposia*, pp. 698–706, Association for Computing Machinery, 2013.

[17] L. Polok, M. Solony, V. Ila, P. Zemcik, and P. Smrz, "Efficient implementation for block matrix operations for nonlinear least squares problems in robotic applications," in *Proceedings of the IEEE International Conference on Robotics and Automation*, IEEE, 2013.

[18] L. Polok, V. Ila, M. Solony, P. Smrz, and P. Zemcik, "Incremental block cholesky factorization for nonlinear least squares in robotics," in *Proceedings of the Robotics: Science and Systems 2013*, 2013.

[19] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of gpu acceleration," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pp. 13–13, USENIX Association, 2010.

[20] F. Zhang, *The Schur complement and its applications*, vol. 4. Springer, 2005.

[21] D. E. Knuth, *The art of computer programming. 1, (1973). Fundamental algorithms*. Addison-Wesley, 1973.

[22] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Graphics Hardware*, vol. 2007, pp. 97–106, 2007.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.

[24] T. Davis, "The university of florida sparse matrix collection," in *NA digest*, Citeseer, 1994.

[25] D. G. Merrill III and A. Adviser-Grimshaw, *Allocation-oriented algorithm design with application to gpu computing*. University of Virginia, 2011.

[26] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. Version 1.7.0.