Fast Linear Algebra on GPU

Lukas Polok and Pavel Smrz
Brno University of Technology
Faculty of Information Technology
IT4Innovations Centre of Excellence
Bozetechova 2, 61266 Brno, Czech Republic
e-mail: {ipolok,smrz}@fit.vutbr.cz

Abstract—**GPUs have been successfully used for acceleration of many mathematical functions and libraries. A common limitation of those libraries is the minimal size of primitives being handled, in order to achieve a significant speedup compared to their CPU versions. The minimal size requirement can prove prohibitive for many applications. It can be loosened by batching operations in order to have sufficient amount of data to perform the calculation maximally efficiently on the GPU. A fast OpenCL implementation of two basic vector functions – vector reduction and vector scaling – is described in this paper. Its performance is analyzed by running benchmarks on two of the most common GPUs in use – Tesla and Fermi GPUs from NVIDIA. Reported experimental results show that our implementation significantly outperforms the current state-of-the-art GPU-based basic linear algebra library CUBLAS.**

GPU; parallel reduction; linear algebra; BLAS; OpenCL; CUDA

I. Introduction

Although programmable GPUs have been available for almost two decades [1], the quest for the most efficient implementation of various algorithms on GPU still forms a hot topic of the current research. This is partially caused by the complex architecture of today's GPUs which makes optimization difficult, and partially by the ever evolving nature of GPUs that can hinder general applicability of new methods.

Scientific computing is one of the most active areas where GPUs have been successfully used. Core problems often benefit from their formulations in terms of linear algebra performing various operations on vectors and matrices. There are several available libraries that enable developers and scientists to accelerate their code using high-level functions running on GPU. The most popular library – NVIDIA CUBLAS [2] became a de-facto standard in this area.

GPU is a highly efficient architecture capable of executing hundreds of threads in parallel [3]. While this is its main strength, it also imposes some limitations for an efficient usage. To deliver its maximal performance, the GPU needs many threads running and even more threads scheduled. For some applications operating on large structures such as image processing, it is typically very easy to achieve. However, there are other types of tasks that require special procedures to keep the GPU busy as they operate on many smaller units (e.g., vectors of lower dimensions). That is why batching of operations is required in order to fully leverage the computing power of the GPU. As simple as it seems, efficient batching is not easy to implement on today's GPUs. One proof of this may be the CUBLAS library itself. It implements batching for only one type of complex matrix operations in its newest version (Q1 2012).

The work reported in this paper focuses on measuring basic machine properties which govern the performance of batched algorithms on GPU. Initial experimental data is employed in the design of simple and fast batched vector operations. For the sake of simplicity, this paper deals with vector operations only. However, the results can be applied to other operations as well. This is outlined in the conclusion section.

There are two classes of operations discussed in the paper. Operations of the first type are limited by the shared memory to run on a single streaming multiprocessor (SM), e.g. vector reduction. The other class involves operations that use the shared memory to cache data but are not limited to a single SM, e.g. vector scaling (a vector-scalar product). Note that there is also a third class of operations which does not require the use of the shared memory at all, but that is not interesting as it runs efficiently on the GPU, and does not require any optimizations.

II. Previous Work

Krüger and Westermann [4] implemented a linear algebra framework using programmable shading, present in the hardware at that time. The disadvantage of their approach is that it is limited by possibilities of programmable shading pipeline itself. The authors had to resort to complicated matrix and vector representations in memory and had to break some of the operations (e.g.

matrix – matrix multiplication or vector reduction) down to several rendering passes, effectively loosing performance.

Harris, Sengupta and Owens [5] described an efficient implementation of parallel prefix sum on GPU, using the CUDA language. This is considered an efficient approach, as it leverages the access to the specialized GPU features such as shared memory for intra-thread communication. That was not possible using programmable shading, as it does not expose this functionality. But it has some shortcomings, as it requires two passes to fully reduce a vector. The second pass can be efficiently implemented on the CPU (it is too small to be executed efficiently by the GPU).

Volkov and Demmel [6] present a hybrid CPU – GPU solution of dense matrix-matrix multiplication and LU, QR and Cholesky decomposition. They easily outperform the standard libraries by implementing several hardware optimizations, notably using a register file instead of shared memory, as opposed to the general GPU programming paradigm. Their work is orthogonal to the focus of this paper.

Nath, Tomov, Dong, and Dongarra [7], [8] show a number of different techniques for optimizing dense linear algebra on the GPUs. These techniques seek include blocking (the use of shared memory to cache small blocks of data), optimizing memory access patterns, optimal storage design, autotuning (employing heuristic search to generate optimal code) and finally some optimizations for generic primitive size. Their work is also orthogonal to the focus of this paper.

There are several high-level libraries oriented on acceleration of linear algebra. These include the above mentioned CUBLAS [2], along with CUFFT [9], CULA [10], CUDPP [11] and possibly more. As of today, none of those libraries provide batched operations with the exception of CUBLAS' function cublas<t>gemmBatched() which calculates the standard BLAS GEMM function, a multiplication and addition of three matrices:

$$\text{gemm}(\alpha, \beta, op_a, op_b, A, B, C) = \alpha op_a(A) op_b(B) + \beta C. \quad (1)$$

where the operations on matrices include no-op, transpose, or conjugate transpose (for complex matrices). The absence of batched versions of simpler functions [2] may indicate that efficient batching of simple operations is difficult to implement.

On modern GPUs which are capable of executing several kernels in parallel, simple batching is possible using several command queues. This technique is, however, limited to maximal amount of simultaneously running kernels (which is 16 on current NVIDIA hardware [3]), and may not therefore offer maximal performance, which is also confirmed by experimental results provided in section 5.

In this paper, two simple operations on vectors will be considered and performance analyzed with and without batching. The first is vector reduction, an operation that produces a scalar from a vector. One example of such operation may be calculation of Euclidean vector length. Implementation of vector reduction on GPUs is complex because of its parallel nature. A solution described in [5] was used as a baseline solution. It makes an efficient use of shared memory, delivering very good performance. To show that even tasks of lesser complexity can be batched efficiently, the second operation, vector scaling is implemented. Vector scaling is embarrassingly parallel and that is why naive implementation is sufficient to run efficiently on the GPU.

Note that this paper is based on our previous work, briefly described in [12]. This paper brings detailed performance analysis of the algorithms outlined in its predecessor, along with the explanation of their behavior. It also describes the devised algorithms in more detail and verifies their function on the newer, more modern hardware.

III. BENCHMARKS

Some simple benchmarks were run before investigating various implementation paths to limit the number of options to the best ones. For the implementation of the batched vector reduction kernel, it is logical to assume that each GPU SM will work on a single vector (because shared memory is required in order to be able to perform parallel reduction and the threads running on one SM can only access it’s shared memory space). Next, it was assumed that there will be potentially many vectors, ideally more than the number of SMs (Fermi architecture GPUs have 16 SMs [3], Tesla GPUs have up to 32 SMs [13]).

The question in implementing efficient batching is how to schedule the individual vectors to be processed. There are several aspects that need to be considered. Most importantly, as vector reduction is not an arithmetically intensive operation, memory accesses need to be properly aligned. Then, the GPU needs to have enough threads to occupy all of the SMs (that is not the same as having the same amount of thread blocks as SMs as the threads are suspended at the first blocking instruction, yielding to the sleeping threads until the outstanding operation is finished). And finally, the workload needs to be well distributed in order to prevent slack.

The first set of benchmarks involved a simple read-modify-write operation carried out on a set of vectors, while each vector is processed in a single thread block, one thread per vector element (while the number of elements matches SIMD width). There are several options for implementing this operation: a single thread block per vector in 1D grid, a single thread block per vector in 2D grid, and a single thread block per several vectors in 1D grid. While this is a simple test, the results are already useful. It turns out that scheduling a 2D kernel is unsurprisingly somewhat slower than scheduling a 1D kernel on both Tesla and Fermi GPUs (the driver needs to create more instances of register files and fill them with the appropriate kernel arguments). That is also the reason why 3D and higher-dimension kernels were not considered. However, using a loop in a 1D kernel to process several vectors is actually much faster than having a single block per vector on Fermi (the difference is negligible on Tesla). This makes sense as Fermi thread scheduler is more complicated than Tesla, and launching a new thread is slower than looping, even though it reduces the total number of running threads. This way, the bandwidth could be boosted from 100 GB/sec to almost 160 GB/sec, much closer to the theoretical 192.4 GB/sec [3] on Fermi. The complete results can be seen in table 1. It contains bandwidths for different numbers of vectors for different types of kernels, speedup of a 1D kernel with loop compared to a 1D kernel, number of iterations where the peak occurred and the corresponding number of thread blocks each SM needs to process. The results for Tesla were omitted in order to save space since there was no significant speedup. Note that the bandwidth is also dependent on the complexity of operations performed on each of the vectors, in this case a linear one. But as previously stated, linear algebra operations are mostly bandwidth-limited and that's why the results are still valid.

TABLE I. SM-BOUND THREAD SCHEDULING STRATEGIES (FERMI)

	Peak bandwidth [GB/sec]					
Vectors	***1D***	***2D***	***1D/loop***	**Speedup**	**Iters**	**Blocks**
102400	108.805	106.097	153.370	40.96 %	32	200
89600	108.727	106.270	153.014	40.73 %	32	175
76800	108.531	106.035	152.655	40.66 %	25	192
64000	108.418	105.919	152.275	40.45 %	20	200
51200	108.231	105.769	152.512	40.91 %	20	160
38400	108.293	105.085	152.128	40.48 %	25	96
25600	108.063	105.448	152.677	41.29 %	64	25
12800	107.212	104.375	150.159	40.06 %	400	2
2560	99.863	97.396	138.661	38.85 %	160	1
1280	92.727	89.524	123.475	33.16 %	80	1
256	56.735	55.512	64.771	14.16 %	16	1

*Iters denotes number of loop iterations, Blocks is number of thread blocks scheduled per SM

The second set of benchmarks was aimed at the memory subsystem. Memory operations on the GPU are fast only if the memory accesses are coalesced [8], [14] (each thread reads from memory location which corresponds to its thread index, with offset that needs to be an integer multiple of 32). While on Fermi the cache should take care of most of the problems, matters are complicated on Tesla. But it is possible to use either the constant memory which is cached on Tesla as well as on Fermi, or to use the shared memory.

Constant memory has the advantage of being transparent to the kernel. No code modifications are required, compared to the global memory approach. The data are automatically copied from global memory by the driver upon kernel launch. Both architectures are equipped with 64 kB of constant memory [3], [13].

Shared memory, on the other hand, has the advantage that there is more shared memory (up to 16 × 48 kB on Fermi [3] and 32 × 16 kB on Tesla [13]) than there is of constant memory. The disadvantage of shared memory is that it needs to be filled explicitly by the kernel, requiring changes in the code. On the other hand, it enables the choice of size of the blocks in which the scales are copied from global memory to shared memory.

The second benchmark was a simple one, it involved iterating over a list of vectors and multiplying each vector by a corresponding scale (again, a read-modify-write operation). This time, vectors were not bound to SMs which makes thread scheduling a lot easier. A constant number of threads per thread block was used (1024 for Fermi and 512 for Tesla) and each thread processed a single vector element. The total amount of data to be processed was 800 MB, vector length was 128. The choice of the large amount of relatively short vectors is to avoid being limited by having too little vectors, hence penalizing the use of larger interleave sizes. Then the performance of the kernel was measured, while using different interleave sizes (the amount of data read into the shared memory before processing any vector elements). The results can be found in table 2.

TABLE II. VECTOR SCALING BANDWIDTH AS A FUNCTION OF INTERLEAVE SIZE

	Bandwidth [GB/sec]	
Interleave size [B]	***Fermi***	***Tesla***
8	12.291	10.859
16	34.824	27.390
256	144.019	74.087
512	146.723	86.804
1024	149.956	89.668
2048	150.039	91.415
4096	151.620	91.990
16384	153.125	92.437
32768	155.167	
49152	156.538	

It is clear that the greater the interleave size, the greater performance. Also, the scheduling parameters were set reasonably, as the peak bandwidths on either platform reach their maximum theoretical values. The results from the two above benchmarks will affect the design of the proposed vector operation algorithms described in the next chapter.

IV. ALGORITHM DESIGN

With the benchmark results at hand, it is now straightforward to implement efficient batched operations. OpenCL was chosen as the implementation language in order to make the resulting code portable and not limited to NVIDIA GPUs. Only simple functions on vectors of real values were implemented, complex values are not supported at the moment.

The parallel reduction is now pretty straightforward, it is sufficient to take the code from [5] and wrap it in looped 1D block code from chapter 3. The only changes required in order to make the implementation practical involve adding vector range specification as the pass size might exceed maximum pass size limitation of the GPU, and the kernel would fail to launch in that case. That already gives good results. This is summed up in algorithm 1. Other variants were also implemented to verify that this variant really gives the best results. The other variants included vector per thread and vector per kernel - different levels of granularity.

The parallel scaling was also practically described in the benchmarks chapter. But again, variants of the algorithm employing the global memory and the constant memory instead of the shared memory were implemented as well in order to show that these really give suboptimal results. The best solution (shared memory) is described in algorithm 2. Like the reduction algorithm, additional parameters for multipass processing were added.

V. Results

Runtimes of a simple baseline "C" implementation, of the proposed GPU implementation and of CUBLAS were recorded. Times needed for generating the test data or storing the results were omitted. GPU times include copying data to GPU memory as well as copying the results back. To ensure all the GPU operations have finished, clFinish() was called. The results of all the operations were cross-checked for correctness and the algorithms operated on pseudorandom data, excluding the possibility of filling the GPU memory with the correct results in one instance and failing to actually generate any results in the next (as often happens with GPU debugging).

The implementations presented here were built using the Microsoft Visual Studio 2008 compiler. The CPU time of all the algorithms was measured on a pair of unloaded AMD Opteron 2360 SE processors (8 cores running at 2.6 GHz in total), running on Windows XP x64. The GPU times were measured on the NVIDIA GeForce GTX 590 (Fermi) and the GTX 260 (Tesla), while using the most recent drivers. The GPUs were present in the computer one at a time, and were replaced between the tests. GPU times were a bit noisy so each test was run four times and the results were averaged.

The results for the three vector reduction approaches described in chapter 4 and for CUBLAS function cublasSnrm2() are plotted on figure 1.a and 1.c and are denoted VAT (vector at a time), VPT (vector per thread), VTB (vector per thread block), and CUDA (function cublasSnrm2()).

All the tests were run with an equal amount of data, only the vector dimensions (and hence the number of vectors) changed. CPU time was therefore almost constant. It can be seen that the last strategy (vector per thread-block) is the most beneficial and even surpasses cublasSnrm2() by more than 50 GB/sec on Tesla and by 80 GB/sec on Fermi, for the very large vector sizes.

The whole situation for the vector scaling benchmarks is depicted on figure 1.b and 1.d. These are denote global (scales are stored in global memory), const (scales are stored in constant memory), local (scales are stored in shared memory) and CUDA (function cublasSscal()). Note the spiking is caused by element misalignment of the non-multiple-of-32 dimensional vectors. As expected, CUBLAS function cublasSscal() turns out to perform sub optimally, although it is better at the unaligned vectors. It is caused by the fact that the first vector is always aligned, and may only result in a small amount of idle threads. And since it only processes a single vector at a time, it never has uncoalesced reads and its performance exceeds performance of the proposed batched version for cases with a few large vectors.

VI. Conclusions and Future Work

A simple, yet novel approach for implementing two different classes of batched algorithms was suggested. It was also demonstrated on two simple vector functions that belong in either class. The results confirm that the described approach is really effective, outperforming the standard GPU BLAS library, CUBLAS, by a factor of three.

There are a lot of algorithms that would profit from this improvement. The list may include matrix operations, sparse vector and matrix operations, and some other. It would be interesting to remove the current limitation that all the vectors in the batch need to have the same dimension. That would enable even more of the existing code to profit from the accelerated algorithms. Another interesting aim is to focus on vectors with lengths that are not a multiple of 32, a feature that seems to be missing in most of the GPU libraries at the time.

Acknowledgment

This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and by the Advanced Recognition and Presentation of Multimedia Data project (FIT-S-11-2).

References

[1] Simon Green, "History of Programmability in OpenGL", online http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_OpenGL_exts.pdf, 2004

[2] NVIDIA, "CUDA CUBLAS Library", online http://developer.nvidia.com/cublas, 2012

[3] NVIDIA, "Fermi Compute Architecture White Paper", online http://www.nvidia.com/object/fermi_architecture.html, 2009

[4] Krüger J., Westermann R., "Linear algebra operators for GPU implementation of numerical algorithms", ACM Transactions on Graphics 22, 2003, 908–916.

[5] Harris M., Sengupta S., Owens J. D.: "Parallel prefix sum (scan) with CUDA", GPU Gems 3, Nguyen H., (Ed.). Addison Wesley, 2007, ch. 31.

[6] Volkov V. and Demmel J., "Benchmarking gpus to tune dense linear algebra", In Proceedings of SC '08, pages 1–11, Piscataway, NJ, USA, 2008.

[7] R. Nath, S. Tomov and J. Dongarra: "Accelerating GPU Kernels for Dense Linear Algebra", VECPAR 2010

[8] R. Nath, S. Tomov, T. Dong, and J. Dongarra, “Optimizing Symmetric Dense Matrix-Vector Multiplication on GPUs”, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11), 2011

[9] NVIDIA, “CUDA CUFFT Library”, online http://developer.nvidia.com/cufft, 2012

[10] EM Photonics, “CUDA CULA Library”, online http://www.culatools.com/, 2010

[11] “CUDA Data Parallel Primitives Library. CUDPP”, online http://code.google.com/p/cudpp/, 2012

[12] Polok, L., Smrz, P., “Implementing Random Indexing on GPU”, In: High Performance Computing Symposium 2011, Boston, US, SCS, 2011, s. 9, ISBN 978-1-61782-840-9

[13] NVIDIA, “Tesla M2090 Board Specification”, online http://www.nvidia.com/object/tesla_product_literature.html, 2011

[14] NVIDIA, “CUDA C Best Practices Guide”, online http://developer.nvidia.com/nvidia-gpu-computing-documentation, 2012

Algorithm 1. Batched vector reduction

```
FUNCTION reduce(vectors, dest, first_vec, last_vec, vec_dimensions)
    FOR v := first_vec + get_group_id(0) TO last_vec - 1 STEP get_num_groups(0) DO
        accum := 0
        FOR i := get_local_id(0) TO vec_dimensions - 1 STEP get_local_size(0) DO
            accum := accum + OP(vectors[i + v * vec_dimensions])
        shared[get_local_id(0)] := accum
        barrier(CLK_LOCAL_MEM_FENCE)
        { reduce in shared memory as described in [5] }
        IF get_local_id(0) = 0 THEN
            dest[v] := shared[0]
```

Algorithm 2. Batched vector scaling

```
FUNCTION scale(vectors, scales, dest, first_vec, last_vec, vec_dimensions, interleave)
    first_elem := first_vec * vec_dimensions
    last_elem := last_vec * vec_dimensions
    elem_step := (interleave - 1) * vec_dimensions
    e0 := get_group_id(0) * elem_step + first_elem
    IF e0 >= last_elem THEN
        RETURN
    WHILE TRUE DO
        v0 := e0 / vec_dimensions
        FOR i := get_local_id(0) TO min(interleave, last_vec - v0) - 1 STEP get_local_size(0) DO
            shared[i] := scales[i + v0]
        barrier(CLK_LOCAL_MEM_FENCE)
        FOR e := get_local_id(0) + e0 TO min(e0 + elem_step, last_elem) - 1 STEP get_local_size(0) DO
            dest[e] := vectors[e] * shared[e / vec_dimensions - v0]
        e0 := e0 + get_num_groups(0) * elem_step
        IF e0 >= last_elem THEN
            RETURN
        barrier(CLK_LOCAL_MEM_FENCE)
```

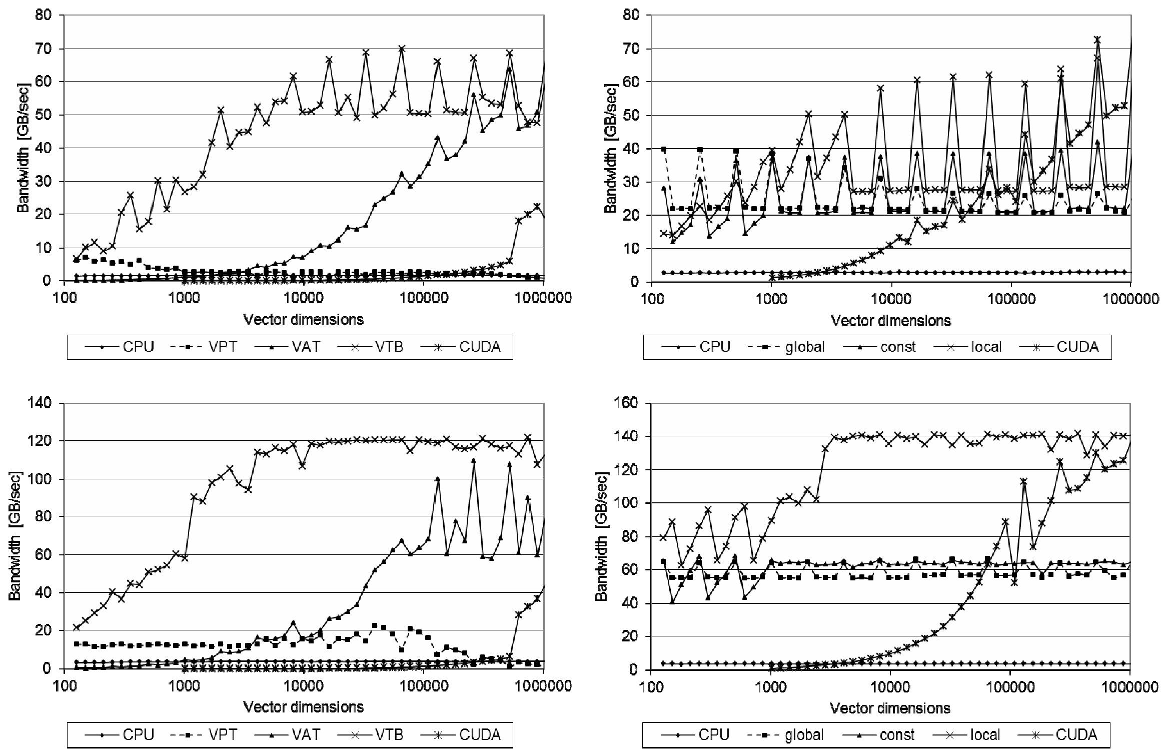

Figure 1. (a) comparison of vector reduction kernels (Tesla), (b) comparison of vector scaling kernels (Tesla), (c) comparison of vector reduction kernels (Fermi), (d) comparison of vector scaling kernels (Fermi).