From Sparse Matrix to Optimal GPU CUDA Sparse Matrix Vector Product Implementation

Ahmed H El Zein
ANU Supercomputing Facility
The Australian National University
Canberra, ACT 0200 Australia
Email: Ahmed.ElZein@anu.edu.au

Alistair P Rendell
School of Computer Science
The Australian National University
Canberra, ACT 0200 Australia
Email: Alistair.Rendell@anu.edu.au

Abstract—The CUDA model for GPUs presents the programmer with a plethora of different programming options. These includes different memory types, different memory access methods, and different data types. Identifying which options to use and when is a non-trivial exercise. This paper explores the effect of these different options on the performance of a routine that evaluates sparse matrix vector products. A process for analysing performance and selecting the subset of implementations that perform best is proposed. The potential for mapping sparse matrix attributes to optimal CUDA sparse matrix vector product implementation is discussed.

I. INTRODUCTION

Graphics Processing Units (GPUs) offer orders of magnitude more floating point performance than conventional processors. Traditionally, accessing this performance for general purpose programming has required the user to cast their problem into a graphical framework of nodes and vertices. This situation changed dramatically in 2007 when NVIDIA released its CUDA programming model for GPUs [1].

CUDA is a software programming model and a programming environment that enables the creation of parallel applications for CUDA enabled GPUs. The CUDA runtime library exposes parts of the GPU and hides others, with the overall effect of presenting the programmer with access to a massively parallel co-processor.

To achieve high performance GPUs use a large number of simple in-order cores that execute in a SIMD fashion. The instructions executed by these cores are associated with an even larger pool of threads, with the hardware being capable of executing instructions from different groups of threads in different cycles. The rationale being that latencies associated with instructions from one group of threads will be masked by operations on other groups of threads. In CUDA threads are grouped into blocks which are in turn organised into grids. Within some limits it is left to the user to determine how to assign threads to blocks and blocks to grids.

The CUDA programming model also has a complex memory hierarchy. Memory categories include register, local, shared, global, constant and texture. These may be associated with different physical locations, may have different scope, and may or may not be cached. Memory accesses can be sequential

or “coalesced”, and may reference standard or wide data types (eg float4, int2 etc).

In short, with CUDA the programmer is presented with a host of different implementation options and it is often not obvious which of these are best, or whether different implementations should be used for different problem sets. This paper outlines some of our work to investigate these issues within the context of a routine to perform sparse matrix-vector products (SpMV).

Our interest in sparse matrices is driven by their widespread appearance in a variety of computational science applications. Generally the structure of a sparse matrix reflects the underlying application, eg non-zero elements representing nearest neighbour interactions. Even on a conventional processor the optimal sparse matrix storage format to use depends heavily on the structure of the sparse matrix and the nature of the hardware platform, eg cache size. This has led to the development of libraries that seek to adapt the sparse matrix format used to the runtime conditions. Three well known examples of such systems are SPARSITY [2], OSKI [3] and AcCLES [4].

The goal of this work is a little different to runtime format selection. As noted above, for CUDA there exists a plethora of different implementation options for any given sparse matrix storage format. Our interest is in determining whether it is possible to design a “blackbox” SpMV routine that for a given storage format is able to identify correctly the best CUDA implementation to use based solely on some attributes of the sparse matrix. Moreover, in light of the fast moving nature of GPU technology, the design methodology developed should be automated or semi-automated, so that it can be applied to new generations of GPU architectures as they appear.

The following section gives some background on sparse matrices, the compressed sparse row (CSR) format used here, and the various possible CUDA implementation options considered. Section III details our test environment while section IV provides performance results for various SpMV implementations on two different CUDA enabled GPUs. Section V outlines our initial attempt to build a “blackbox” SpMV routine, section VI details related work, and section VI contains conclusions and comments on future work.

II. BACKGROUND

A matrix is considered to be sparse if many of its elements are zero and there exists an advantage to exploiting this. This advantage may involve reduced computational effort, reduced memory storage requirements or both. There are a variety of different sparse matrix storage formats, with CSR arguably the most widely used. In CSR format non-zero elements are stored in a dense vector (*val*) while another vector (*ind*) of the same length is used to store the column index of each non-zero element. A third array (*ptr*) is then used to carry offsets into *val* corresponding to the first element of each row. Code for implementing SpMV using CSR format is shown in Figure 1

```
for (i = 0; i < M; i++)
    for (j = ptr[i]; j < ptr[i+1]; j++)
        res[i] = res[i] + val[j]*vec[ind[j]];
```

Fig. 1. Matrix Vector Product using Compressed Sparse Row (CSR) Storage

Executing the code in Figure 1 requires access to five different vectors. As mentioned above CUDA provides a variety of different memory types and memory access modes. Some memory types, such as Constant Memory, are read only, so cannot be used to store the result vector (*res*). Other memory, such as Shared Memory, is extremely limited in size and would be unable to store *val* or *ind* except for trivially small problem cases. In Table I we list the five different vectors and the different memory storage and access types considered viable in this work.

TABLE I
STORAGE OPTIONS FOR THE DIFFERENT CSR SpMV QUANTITIES

Array	Global Coa	Mem Seq	Texture Coa	Mem Seq	Const Mem	Shared Mem	Wide Data Types
val	✓	✓	✓	✓	×	×	✓
ind	✓	✓	✓	✓	×	×	✓
ptr	×	✓	×	✓	✓	✓	×
vec	×	✓	×	✓	✓	×	×
res	✓	✓	×	×	×	×	×

In Table I “Coa” and “Seq” refer to the use of coalesced or sequential memory reads. When using coalesced memory reads all threads in a block cooperate to read an entire row of matrix elements. When using sequential memory reads, each thread within a block of threads reads a unique row of the sparse matrix. Coalesced memory reads are much faster, but are wasteful if a row is not an exact multiple of the number of threads in a block. Wide data types refers to the use of float, float2 or float4 data (hereafter referred to as *f1*, *f2* and *f4* respectively). This has similar performance advantages to coalesced memory reads, and similar penalties due to additional memory traffic when the quantity being accessed is not an exact multiple of the data size.

One final parameter to consider is the assignment of rows to threads or blocks. It is possible to assign multiple rows to each thread or block (depending on whether it is a “Seq” or

“Coa” implementation). Rows can be assigned using either a cyclic or a block distribution pattern. In what follows we will refer to these options as *sr*, *mr_c* and *mr_b* for single row, multi-row cyclic and multi-row block respectively.

Using all these options we can describe a particular SpMV implementation using the following notation:

```
[coa/seq]-[sr,mr_c,mr_b]-[f1,f2,f4]_
[val,ind storage]-[ptr storage]-[vec storage]
```

where the storage options for *val*, *ind*, *ptr* and *vec* will be denoted as *memm*, *cnst* and *text* for global, constant and texture memories respectively.

In total using all available options gives rise to 432 possible SpMV implementations. An automatic code generator was written to generate all possible implementations with the exception of using shared memory for the *ptr* array. This was due to the fact that initial testing showed no performance gain from using shared memory over global memory for the *ptr* storage. The resulting 324 routines were all generated and evaluated for performance. In what follows we aim to determine whether it is possible to focus only on a small subset of these 324 implementations, and then if it is possible to map from sparse matrix attribute to the best performing CUDA implementation for that matrix. First, however, we detail our experimental test environment.

III. TEST ENVIRONMENT

Results presented here were gathered using sparse matrices downloaded from the Florida sparse matrix collection [5] in Nov. 2008. At the time the library contained over 2200 matrices. Of these only those matrices that containing between 10^5 and 10^7 non-zero elements were used. This gave rise to 747 matrices. This was further reduced to 735 matrices by removing those matrices that could not fit in the global memory available on the NVIDIA GeForce 8800 GTX GPU being used at the time.

Two GPU cards will be considered here. The first is the GeForce 8800 GTX. This system was released in late 2006, has 128 CUDA cores, 768MB of GDDR3 memory and a theoretical peak of 388Gflops (single precision using CUDA). The second is the GeForce GTX 295. This system was released in early 2009, combines two GTX 200 GPUs on a single card, where each GTX 200 has 240 CUDA cores, 896MB of GDDR3 memory and a peak performance of 894Gflops. Beyond the obvious differences the two cards also exhibit subtle differences, for example the GTX 295 is able to coalesce a larger variety of memory access patterns than the 8800 GTX. Only a single GTX 200 on the card was used for these results. All results reported here relate to single precision sparse matrix vector product implementations.

IV. PERFORMANCE ASSESSMENT ON 8800 GTX AND GTX 295

For each of the 735 sparse matrices obtained from the Florida sparse matrix library [5] the performance of each of the 324 different SpMV implementations was measured on the 8800 GTX GPU. The recorded time includes the time

to: A) Convert the `vec` array from `double` to `float`, B) Copy the `vec` array to the GPU, C) Execute the SpMV kernel and D) Copy `res` back to the host CPU. One hundred (100) different timing measurements were made in each case, with the results averaged. GFlops are computed using the number of floating point operations that would be performed if the SpMV algorithm presented in Figure 1 were used, i.e. it does not include additional operations that may take place due to the necessity to pad a particular matrix. All experiments were initially performed using 32 threads per block. All multi-row implementations were set to process 16 rows per thread or block (depending on whether it is a "Seq" or "Coa" implementation).

The implementations that gave rise to the highest GFlop rate for at least 2 of the 735 matrices were determined. This identified 45 unique implementations from the 324. While a reduction from 324 possible implementations to 45 represents a significant decrease in complexity, determining how best to map a given sparse matrix to one of 45 possible implementations is still a challenging task. In an attempt to simplify this task further we introduce the concept of a best possible set (BPS) of implementations.

The BPS of size N is defined in terms of the BPS of size N-1, with the BPS of size 1 defined as the implementation that if used for all possible matrices gives rise to the best possible performance. Having defined a BPS of size 1, the BPS of size 2 is defined by adding the one additional implementation that gives rise to the biggest performance enhancement. In this sense the BPS set is defined incrementally using a greedy based approach.

At this point we have not said what metric is used to define "best possible". In the work presented here we will use average performance as the evaluation metric. Thus for the BPS of size 1 we require the implementation that gives the highest average GFlop rate when applied to all 735 test matrices. (Another metric might be the highest minimum performance.)

Using BPS's of increasing size we show in Table II the average, minimum, maximum and standard deviation of the performances obtained across all 735 sparse matrices on the 8800 GTX GPU. Also shown is the actual implementation used for BPS 1, and the routines that are added as the size of the BPS is progressively increased. Results for the BPS of size 45 represent the best possible performance that can be obtained for the 735 matrices using any of the 324 different SpMV implementations.

The results in Table II show that using all 45 SpMV implementations gives an average effective performance of 1.636GFlops, a minimum performance of 0.142GFlops and maximum performance of 8.401GFlops. Meanwhile if only one SpMV implementation is used the average effective performance decreases by about 26% and the maximum effective performance drops by over 55%. Expanding the BPS from 1 raises the effective average performance and immediately enables the maximum effective performance of 8.401GFlops to be reached. The results show that using a BPS of 4 achieves over 96% of the average effective GFlops achieved with the

TABLE II
SPARSE MATRIX VECTOR PERFORMANCE ON THE 8800 GTX GPU AS
FUNCTION OF BEST PERFORMING SET (BPS). SEE TEXT FOR DETAILS.

BPS	Effective GFLOPS				Algorithm Added
	Ave	Min	Max	Stddev	
1	1.199	0.046	3.718	0.878	seq-sr-f4_memmm-text-text
2	1.431	0.142	8.401	1.211	coa-mr_c-f2_memmm-text-text
3	1.556	0.142	8.401	1.274	seq-sr-f4_text-memmm-cnst
4	1.583	0.142	8.401	1.293	coa-sr-f2_memmm-cnst-text
5	1.601	0.142	8.401	1.296	seq-sr-f4_text-memmm-text
6	1.613	0.142	8.401	1.314	coa-sr-f2_memmm-text-cnst
7	1.620	0.142	8.401	1.309	seq-mr_c-f4_text-text-text
8	1.623	0.142	8.401	1.309	seq-mr_c-f4_text-text-cnst
9	1.624	0.142	8.401	1.311	coa-sr-f2_memmm-cnst-cnst
10	1.626	0.142	8.401	1.312	seq-mr_c-f4_memmm-text-text
45	1.636	0.144	8.401	1.314	

BPS of 45. Increasing the BPS beyond 4 converges towards the BPS 45 result relatively slowly. Of the four routines in BPS 4, two use sequential memory accesses while two use coalesced memory accesses. Two use `f4` data types while two use `f2`. Three use global memory for `val` and `ind`, while one uses texture references. Overall there appears to be no obvious pattern to the sort of implementations that are found in BPS 4.

The above process was then repeated using one GPU on the GTX 295 system. This time 47 implementations gave rise to the highest GFlop rate for at least 2 of the 735 matrices. The results are given in Table III.

TABLE III
SPARSE MATRIX VECTOR PERFORMANCE ON THE GTX 295 GPU AS
FUNCTION OF BEST PERFORMING SET (BPS). SEE TEXT FOR DETAILS.

BPS	Effective GFLOPS				Algorithm Added
	Ave	Min	Max	Stddev	
1	1.965	0.053	5.876	1.384	seq-sr-f4_text-memmm-text
2	2.326	0.157	11.080	1.815	coa-mr_c-f4_text-text-text
3	2.447	0.157	11.080	1.858	seq-sr-f4_text-memmm-cnst
4	2.492	0.157	11.602	1.908	coa-sr-f1_memmm-memmm-text
5	2.524	0.157	11.602	1.968	coa-sr-f2_memmm-text-cnst
6	2.548	0.157	11.602	1.984	coa-mr_c-f2_memmm-text-text
7	2.555	0.157	11.602	1.987	coa-sr-f4_text-memmm-text
8	2.561	0.162	11.602	1.984	seq-mr_c-f4_text-memmm-text
9	2.566	0.162	11.602	1.989	coa-mr_c-f2_text-memmm-text
10	2.569	0.162	11.602	1.997	coa-sr-f4_text-text-cnst
47	2.588	0.164	11.602	2.009	

Not surprisingly the effective performance achieved on the GTX 295 is higher than on the 8800 GTX. Again over 96% of the BPS 47 average performance is achieved using BPS 4. It is also interesting to note that the implementations appearing in the various BPS's are different on the GTX 295 GPU compared to the 8800 GTX GPU. For example we now find that three of the routines in BPS 4 use Texture references for `val` and `ind` compared to just one in BPS 4 on the 8800 GTX. This fact highlights the need to develop code optimisation protocols that can be automated and readily applied to new GPU architectures as they appear.

V. TOWARDS A BLACKBOX SPMV ROUTINE

Given a BPS the goal is to identify the optimal implementation within that BPS based on some attributes of the sparse matrix. This is not an easy problem. First the attributes have to be decided and then what decision making approach to use determined. We explore here a minimalistic approach based on a simple two level decision tree and just three matrix attributes; the number of rows, the number of columns, and the average number of non-zero elements per row.

For the GTX 295 and BPS 4 two implementations use sequential memory reads, while two use coalesced memory reads. To investigate whether we can distinguish between these two cases using the three matrix attributes given above we plot in Fig. 2 and for all 735 matrices the memory read type used as a function of matrix attribute. The results suggest that average number of non-zero elements per row is a reasonably good discriminant for determining when to use sequential or coalesced memory reads.

Fig. 2. Use of Coalesced or sequential memory access as function of matrix attribute for BPS 4 and GTX 295

Given the above partitioning the next requirement is to distinguish between the two sequential implementations and the two coalesced implementations. Thus similar plots are given in Figs. 3 and 4. Results indicate that the number of

columns is probably the best discriminant for the sequential implementations while the number of rows is the best for the coalesced implementations.

Fig. 3. Use of the two different sequential implementations as matrix attribute for BPS 4 and GTX 295

Using average number of non-zero elements per row, number of rows and number of columns as discriminants we construct a simple two level decision tree. This is shown in Fig. 5 together with the values for the discriminants that have been obtained by optimising for maximum overall performance across the entire data set. The performance achieved using this decision tree is shown in Table IV where it is compared with the ideal performance, i.e. that which would be achieved if the decision tree correctly mapped every matrix to the best performing SpMV within BPS 4. The blackbox performance is within about 6% of the ideal performance. Or from another perspective, the performance obtained using BPS 4 and the simple decision tree is slightly better than that obtained using BPS 2 and perfect mapping (see the results for BPS 2 given in Table III).

As Stated previously, the number of threads per block does affect the performance of the sparse matrix-vector implementations. In order to gauge this affect, we evaluate each of the 4 implementations with different setting of threads per block.

Fig. 4. Use of the two different coalesced implementations as matrix attribute for BPS 4 and GTX 295

TABLE IV

DIFFERENCE BETWEEN IDEAL PERFORMANCE USING BPS OF SIZE FOUR AND RESULTS OBTAINED USING THE DECISION TREE ON THE GTX 295 GPU.

Method	Effective GFLOPS			
	Ave	Min	Max	Stddev
Ideal	2.492	0.157	11.602	1.908
Decision Tree	2.344	0.053	11.08	1.885

For each of the implementations, the number of times it was selected by the decision tree was noted. The number of times a particular setting of threads per block performed best was noted as well. This information is available in Table V.

TABLE V

OPTIMAL THREADS PER BLOCK FOR DIFFERENT BPS IMPLEMENTATIONS

Implementation	total	Times selected					
		number of threads per block					
		16	32	64	128	256	512
coa-sr-f1_memmm-memmm-text	16	0	0	3	7	4	2
seq-sr-f4_text-memmm-cnst	70	2	13	30	10	8	7
coa-mr_c-f4_text-text-text	141	80	52	4	5	0	0
seq-sr-f4_text-memmm-text	508	123	196	88	40	28	33

Table V suggests that the coa-sr-f1_memmm-memmm-text im-

Fig. 5. Simple Decision Tree for BPS 4 and the GTX 295

plementation be set at 128 threads per block, the seq-sr-f4_text-memmm-cnst implementation at 64 threads per block, the coa-mr_c-f4_text-text-text implementation at 16 threads per block and the seq-sr-f4_text-memmm-text implementation at 32 threads per block.

Incorporating this information into the blackbox SpMV, leads to a very slight increase in the average performance over all the 735 matrices as is show in Table VI.

TABLE VI

BLACKBOX SPMV PERFORMANCE USING THE DECISION TREE AND OPTIMAL THREAD PER BLOCK SETTINGS ON THE GTX 295 GPU.

Method	Effective GFLOPS			
	Ave	Min	Max	Stddev
Blackbox Effective GFLOPS	2.36	0.05	11.00	1.86

VI. RELATED WORK

Two notable contributions in the area of sparse matrix vector products on CUDA enabled GPUs are the works of Bell and Garland [6] and Baskaran and Bordawekar [7]. We briefly summarize these below.

Bell and Garland [6] investigated a variety of sparse matrix formats. Each of these formats requires an SpMV kernel and in the case of CSR format both a sequential and coalesced CSR implementation were created. The authors also investigated the use of texture memory and found a performance gain through its use. Both structured and unstructured matrices were considered. The structured matrices were composed of standard discretizations of Laplacian operations in 1, 2 and 3 dimensions. The Unstructured matrices were represented by a set of 14 matrices taken from previous work by Williams et al [8]. In comparison, the work presented here is focused on a single sparse matrix format type (CSR), exhaustively studies the performance of all possible implementation options, uses a significantly larger number of sparse matrices, and makes

an attempt to map directly from matrix attribute to optimal implementation.

Baskaran and Bordawekar [7] focus solely on the CSR storage format. They identify four optimisations, i) exploiting synchronization free parallelism, ii) optimized thread mapping, iii) optimized off-chip memory access, iv) exploiting data reuse. They evaluate their implementation using 19 sparse matrices taken from the Florida sparse matrix collection [5]. They compare their performance with that of Bell and Garland [6] and the NVIDIA CUDPP library [9] which has an SpMV implementation based on the segmented scan approach of Sengupta et al. [10]. Although more similar to the work presented here in that they focus exclusively on CSR format, it represents a more traditional approach to program optimisation that is less amenable to automation, has many fewer implementations, and uses many fewer test matrices.

VII. CONCLUSIONS AND FUTURE WORK

In this work we have taken a relatively simple piece of code for performing sparse matrix vector products and shown how it can give rise to a large number of different implementations when using CUDA on a GPU. Moreover the variation in performance between these different implementations was found to be quite large. Although the number of implementations was large as was the number of test matrices, both generation of the implementations and performance testing are operations that are amenable to automation.

From the large number of implementations and extensive performance testing we introduced the concept of a BPS. This was defined incrementally using a greedy type algorithm. It is important to note that for anything larger than 1 and less than the maximum BPS size, the BPS may not contain those routines that give the best possible performance. That is BPS 2 must contain the routine in BPS 1, and is not free to chose any two routines even if this may give better overall performance. This approach was taken in order to avoid the factorial blow out in number of permutations that must be evaluated for large BPS sizes.

In this work the performance of one implementation was considered to be better than another as soon as the measured performance was greater. There was no concept of performance being significantly better. Attempting to group implementations that give roughly the same performance for a give matrix may provide a means of reducing the number of implementations that need to be considered.

In addition, if the different settings of threads per block were added as a factor in the selection of the BPS it is possible that the implementations and/or the value of the discriminants in the decision tree would be different.

From the BPS we attempted to construct a simple decision tree that maps matrix attributes to optimal implementation. The approach used here was crude, and based on a minimal number of matrix attributes. The results however show some promise. Use of more elaborate matrix attributes (such as average number of neighbours) may improve this mapping. As might the use of machine learning techniques to provide a more sophisticated mapping function. In the future we also plan to use reinforcement learning techniques that do not rely on pre-learning a set of rules and can adapt to changes in runtime conditions [11].

In terms of performance, the GFlops reported here may seem a little low given the high theoretical peak of the GPU. It should be remembered that the GFlops given are effective GFlop rates that involved converting the `vec` array from `double` to `float` and some transfer of data between the GPU and main memory. They also relate only to useful floating point operations, not to additional operations that might be performed because of padding of the various vectors. It should also be noted that the performance given for the GTX 295 used just one of the two GPUs available on this card.

ACKNOWLEDGEMENT

This work was in part supported by the Australian Research Council through grant DP0987773.

REFERENCES

- [1] NVIDIA, *NVIDIA CUDA Programming Guide*, 2nd ed., July 2008.
- [2] E.-J. Im, K. A. Yelick, and R. W. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *IJHPCA*, vol. 18, no. 1, pp. 135–158, 2004.
- [3] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proceedings of SciDAC 2005*, ser. Journal of Physics: Conference Series. San Francisco, CA, USA: Institute of Physics Publishing, June 2005.
- [4] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone, "Performance optimization and modeling of blocked sparse kernels," *IJHPCA*, vol. 21, no. 4, pp. 467–484, 2007.
- [5] T. A. Davis, "University of florida sparse matrix collection," *NA Digest*, vol. 92, 1994.
- [6] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC*. ACM, 2009.
- [7] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus," *IBM Research Report*, vol. RC24704, 2009.
- [8] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. A. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC*, B. Verastegui, Ed. ACM Press, 2007, p. 38.
- [9] "CUDPP: CUDA Data Parallel Primitives library," <http://www.gpgpu.org/developer/cudpp/>.
- [10] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Graphics Hardware*, M. Segal and T. Aila, Eds. Eurographics Association, 2007, pp. 97–106.
- [11] W. Armstrong and A. P. Rendell, "Reinforcement learning for automated performance tuning: Initial evaluation for sparse matrix format selection," in *CLUSTER*. IEEE, 2008, pp. 411–420.