J Supercomput (2013) 63:710–721
DOI 10.1007/s11227-011-0626-0

Performance modeling and optimization of sparse matrix-vector multiplication on NVIDIA CUDA platform

Shiming Xu · Wei Xue · Hai Xiang Lin

Published online: 7 June 2011

Abstract In this article, we discuss the performance modeling and optimization of Sparse Matrix-Vector Multiplication (SpMV) on NVIDIA GPUs using CUDA. SpMV has a very low computation-data ratio and its performance is mainly bound by the memory bandwidth. We propose optimization of SpMV based on ELLPACK from two aspects: (1) enhanced performance for the dense vector by reducing cache misses, and (2) reduce accessed matrix data by index reduction. With matrix bandwidth reduction techniques, both cache usage enhancement and index compression can be enabled. For GPU with better cache support, we propose differentiated memory access scheme to avoid contamination of caches by matrix data. Performance evaluation shows that the combined speedups of proposed optimizations for GT-200 are 16% (single-precision) and 12.6% (double-precision) for GT-200 GPU, and 19% (single-precision) and 15% (double-precision) for GF-100 GPU.

Keywords Sparse matrices-vector multiplication · GPU · CUDA · Matrix permutation · Cache optimization

S. Xu (✉)
Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: auhgnist@gmail.com

W. Xue
Tsinghua University, RM. 8-210, East Main Bldg., 100084 Beijing, China
e-mail: xuewei@tsinghua.edu.cn

H.X. Lin
Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: h.x.lin@tudelft.nl

1 Introduction

Sparse Matrix-Vector Multiplication (SpMV) is an important computational kernel for many numerical applications such as Krylov subspace solvers [14]. With the wide adoption of Graphics Processing Units (GPUs) in high performance computing systems [3, 6], it is crucially important to optimize SpMV kernel on GPU based systems. In this article, we discuss the performance modeling and optimization of SpMV using NVIDIA CUDA [1]. With CUDA, the GPU is abstracted as a chip with multiple Stream Multiprocessors (SMs), with each SM containing several Stream Processors (SPs). The execution model of CUDA form a massively parallel scenario, with threads being executed in an SIMD manner. The hierarchy of threads includes grid, thread blocks, and threads. Threads are scheduled to SP's at the granularity of Thread Blocks. The actual scheduling of execution of threads onto SPs are in the granularity of warps, i.e., 32 threads. The NVIDIA GPUs used in this article are compared in Table 1, including cache hierarchy differences.

We propose the optimization of SpMV operations based on ELLPACK format mainly from two aspects: (1) access to the matrix, and (2) access to the dense vector. We decouple these two aspects and construct a performance model for the accesses to matrix. By permutations the matrix can be transformed into a bandwidth-reduced form, which enhances locality in accessing the dense vector, and reduces the memory amount for the matrix. On GF-100 with better cache support, we propose the technique to differentiate the memory accesses so that the dense vector data can fully utilize the cache hierarchy on new NVIDIA GPUs.

The following part of the article is organized as follows. Section 2 gives a brief introduction to the state-of-the-art SpMV optimizations on GPUs, and the performance modeling based on ELLPACK format. Section 3 includes cache-oriented optimization on GT-200 and GF-100. In Sect. 4, we propose SpMV optimization based on matrix bandwidth reduction. Section 5 evaluates the performance enhancements. Section 6 concludes the article.

Table 1 Quantitative comparison between GPU architectures

GPU	Series	GT-200		GF-100	
	name	C1060		GTX-480	
Memory bandwidth		$\sim$80 GB/s		$\sim$115 GB/s	
Concurrent thread count		30720		23040	
Concurrent warp count		960		720	
		Size	Latency	Size	Latency
L1 DCache		N/A	N/A	16/48 KB	80
L2 DCache		N/A	N/A	768 KB	212
L1 TCache		$\sim$5 KB	258	12 KB	22
L2 TCache		256 KB	366	None	427
Global memory		4 GB	506	1.5 GB	319

"DCache" denotes Data Cache. "TCache" denotes Texture Cache. L1 caches are per-SM resource. All timing are in cycles

2 Performance analysis and modeling of SpMV with ELLPACK

SpMV involves operation as follows:

$$y = y + A \times x \tag{1}$$

where y and x are both dense vectors, and A is a sparse matrix. The values of the elements in A, x and y are either Single-Precision (FLOAT) or Double-Precision (DOUBLE) numbers. Suppose that A has size of $n \times n$, and the total nonzero element count in A is nnz. Then for each SpMV operation, the elements in the sparse matrix A are accessed only once. In total, SpMV involves $2nnz$ floating point operations. SpMV has a very low computation/data ratio (close to 1). The performance of SpMV is bound by the memory bandwidth.

2.1 SpMV on GPU—state of the art

Performance analysis and optimizations of SpMV on CPU have been discussed in many works. Most of the works focus on the reduction of the amount of data involved in accessing the matrix. These techniques include: (1) Register Blocking and similar ones [5, 11, 13], (2) sparsity pattern-based compression [4, 12], and (3) data-based compression [9]. With Register Blocking, small dense blocks of the matrix are recorded and accessed, rather than single elements. This reduces the access to the row/column index information of the nonzero elements. Another important optimization with Register Blocking is SIMDization, enabled by unrolling of the inner-block iterations. For CPUs, due to the readily available cache support, the reuse in the dense vector x is taken care of implicitly by the hardware.

Due to the growing adoption and popularity of GPUs, there have been recent works on porting SpMV to these platforms [7, 10, 13]. In [5] and various other works, SpMV is used to construct CG solvers using GPU platforms. In [7], the authors applied Register Blocking to SpMV on GPU. In [7, 10], the authors show that effective SpMV on GPUs rely on memory-bandwidth formats, and among them ELLPACK shows best overall performance.

In this article, we focus on ELLPACK format based SpMV operations on GPU. When stored in ELLPACK format, the ith nonzero element of adjacent rows are stored in adjacent locations in memory. In SpMV, each CUDA thread is assigned to one matrix row, hence that the access to the ith nonzero element of each row can be coalesced to avoid waste of memory bandwidth. Hence, a total $n \times m$ size of data array should be allocated for both column index information and value information, where m is the maximal nonzero element count of each row. In ELLPACK, padding is necessary when the nonzero element count per row varies. When the amount of padding is large, it may negatively affect the performance. To alleviate this, in [10] the authors introduces HYB (Hybrid) format to contain the major parts of the matrix in ELLPACK format. The extra elements are contained in a separate part recorded in Coordinate format (COO).

2.2 Performance analysis for SpMV

To quantitatively study the performance profile of SpMV, we breakdown the computation of SpMV into two parts: (1) reading of data in A, reading and writing of y, computations (i.e., Multiplication-and-Add operations), and (2) reading of elements in x. Part (1) includes all deterministic memory accesses: reading to indices and values of nonzero elements are coalesced; reading and writing of y vector can be easily made into coalesced accesses. Part (2) includes nondeterministic memory accesses: offset into x is decided by the column index of the nonzero elements, which is subjected to the matrix sparsity pattern.

To measure the timing of both parts, we use a pseudo x vector with each element as a pre-defined constant value. This value is hard-wired into codes hence avoiding access to the actual memory for x. Then the time dedicated to the access of dense vectors (denote as t_x) can be calculated as: $t_x = t_{\text{all}} - t_{\text{pseudo\_}x}$, where t_{all} denotes the time required to perform SpMV with a normal, nonpseudo x vector, and $t_{\text{pseudo\_x}}$ for that with a pseudo one. For $t_{\text{pseudo\_}x}$, we build a performance model for it by dividing the execution of SpMV kernel into conceptual passes. Each pass corresponds to n_{th} rows/threads, where n_{th} is the maximal number of concurrent threads on GPU. In the ith pass, the computation of the rows with indices between $(i-1)\cdot n_{\text{th}}+1$ and $i\cdot n_{\text{th}}$ are carried out. We also consider the kernel launching and scheduling overhead. We model the execution time of pass i as follows (denoted T_i):

$$T_i = \begin{cases} \Delta_0 + m \times \Delta_2 & \text{if } i = 1 \\ \Delta_1 + m \times \Delta_2 & \text{if } 1 < i < p \\ \Delta_1 + \alpha \times m \times \Delta_2 & \text{if } i = p \end{cases} \tag{2}$$

Δ_0 and Δ_1 models the startup overhead for the first pass and the rest passes, respectively. Δ_2 models the access to matrix data. Note that the multiplier m is the number of nonzero elements per row. The linear relationship between T_i and Δ_i reflects that the execution time of SpMV is dominated by global memory accesses to matrix data. Value α is a proportion of the actual warp count of the last pass in the maximal allowed concurrent warp in a pass. It is used to describe the situation in which the last pass is not complete due to n values. With $t_{\text{pseudo\_}x} = \sum_{i=1}^{p} T_i$, we measure $t_{\text{pseudo\_}x}$ for artificial matrices which satisfy: (1) the size of the matrix is an integer multiple of n_{th}, and (2) the matrix has exact same number of nonzero elements per row. We compute the values of Δ_0, Δ_1, and Δ_2 by varying n and m and measurements over $t_{\text{pseudo\_}x}$. They are shown in Table 2 for C1060 and GTX-480.

Table 2 Δ_0, Δ_1 and Δ_2 values for SpMV on C1060 and GTX-480

GPU	FLOAT			DOUBLE		
	Δ_0	Δ_1	Δ_2	Δ_0	Δ_1	Δ_2
C1060	7.753e-3	3.530e-3	2.807e-3	1.077e-2	6.292e-3	4.312e-3
GTX-480	3.713e-3	1.180e-3	1.250e-3	4.430e-3	2.444e-3	1.775e-3

Table 3 Performance results of SpMV when a pseudo vector of x is used for GT-200

Matrix	n	nnz	$t_{\text{pseudo\_}x}$ (ms)	t_{all} (ms)	t_x in t_{all}
FEM/Cantiliver	62451	4007383	0.493	0.530	7.1%
FEM/Sphere	83334	6010480	0.604	0.658	8.2%
FEM/Accelerator	121192	2624331	0.470	0.624	25.7%
Economics	206500	1273389	0.421	0.508	17.1%
Epidemiology	525825	2100225	0.275	0.324	15.1%
Protein	36417	4344765	0.652	0.703	7.2%
WindTunnel	217918	11634424	1.203	1.246	3.4%
QCD	49152	1916928	0.190	0.207	8.4%
FEM/Harbor	46835	2374001	0.428	0.448	4.5%
Circuit	170998	958936	0.231	0.310	25.4%
Web	1000005	3105536	0.719	0.956	24.8%
Geo-Mean					13.1%

Table 3 shows the measurement of t_x for matrix test suite used in [10, 13] on C1060 using FLOAT operations. Dimension and nonzero element count of these matrices are also shown. This test suite is also used for the evaluation of SpMV optimizations in this article. On average about 13% of the time of SpMV is spent in accessing x, which contributes a significant portion to the total SpMV kernel execution time.

3 Caching of x—analysis and optimization

3.1 Caching of x on GT-200

There are 3 hardware mechanisms that can be used for caching dense vector x on GT-200 architecture: (1) use texture cache and treat x as a 1-D texture; (2) use constant cache, by decorating x as a constant; and (3) use Shared Memory as a software-managed cache. The first approach is used in [10]. The second one is virtually limited by the small size of the constant memory space, which across all CUDA devices is 64 KB. Due to the large size of n, constant cache is not sufficiently large to contain x, hence cannot be used for caching. The third one requires software management of Shared Memory. According to both our experiments and [7], it results in too much overhead for the management of the cache. Hence, we do not consider Approach 2 and Approach 3 in this article. Due to the large access time of TCache (shown in Table 7), it is beneficial to improve the cache hit ratio and avoid both long access latency and the overhead of extra memory accesses in the case of TCache misses.

3.2 Caching optimization on GF-100

As noted in Table 1, with the new GPU architecture (i.e., GF-100), dramatic improvements are introduced in the memory subsystem. This reflects the trend of proving more friendly programmability of GPUs, especially in cache subsystem: (1) larger cache size, (2) lower access latencies, and (3) support for cache coherency. For SpMV, there is only data reuse of accesses to x which is read-only. Hence, the larger size and lower latency in caches will generally reduce the time spent in accessing x.

Like on modern CPUs, on GF-100 all data accesses are filtered through cache by default. This in effect results in conflicts between data in A and data in x when using a popular LRU (least-Recently-Used) mechanism for cache management. Potentially, it is possible that data in x which would be used again be evicted from the cache by the data in A which is never used again. While on the API level, current CUDA implementations (for up to date version of 3.1) does not expose different cache behavior for different data, we use inline PTX assembly to achieve differentiated cache behavior for accesses to A and those to x:

1. Let accesses to A be marked as un-cached, or data in A to have lowest priority in cache, so that they are never cached or evicted first when capacity conflict happens.
2. Let accesses to x be fully cached, so that data in x always have higher priority for staying in cache than data in A.

The two lines of codes below are the example of inlined content to a CUDA kernel code. Differentiated access patterns are used for accesses to data at `addr1` and `addr2`. The syntax is in PTX assembly [2].

```
__asm("ld.ca.f32 %0, [%1];" : "=f" (a) : "l" (addr1));
__asm("ld.cv.f32 %0, [%1];" : "=f" (b) : "l" (addr2));
```

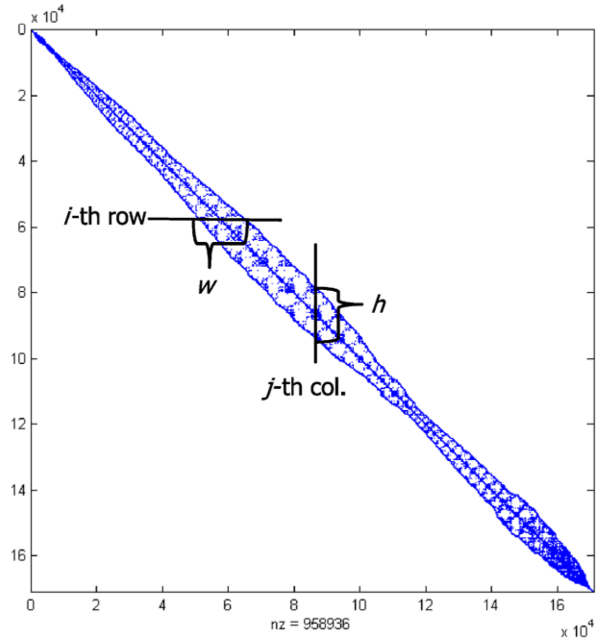
By the first line of the code, we load a FLOAT number to value `a` in C language space from `addr1`, and by the second line, we load a FLOAT number to value `b` from `addr2`. The loading of `a` is cached, by the PTX instruction `ld.ca` where `ca` means "cache all." The loading of `b` is marked as volatile by `ld.cv`, which means "load-with-cached-as-volatile" and implies that the value at address `addr2` is volatile and access to it will skip caches and be direct from main memory.

In SpMV, we use `ld.ca` for the accesses to data in x and `ld.cv` for accesses to data in A. This allows maximal dedication of cache resource to data in x. It is well recognized that future GPUs will evolve to include full cache support, bearing more similarity to conventional CPUs. Differentiation in data access pattern in terms of cache behavior provides potential for better performance of applications, given the knowledge of the specific data access pattern for the specific application.

4 SpMV optimization with matrix bandwidth reduction

In this section, we carry out SpMV optimization based on Matrix Bandwidth/Profile Reduction. There are two benefits of matrix bandwidth reduction: (1) improved locality in accessing x vector, and (2) index compression which is enabled by a reduced matrix bandwidth. We use Reverse Cuthill–McKee (RCM) [8] for the bandwidth reduction in the following part of the chapter, mainly due to its simplicity and popularity. Other algorithms are possible and contained in future research.

Fig. 1 SpMV of Matrices in a Reduced Bandwidth form



4.1 Enhanced locality in accessing x

Shown in Fig. 1, there are two aspects of locality in accessing x: (1) for the ith row, the accessed part of the ith CUDA thread lies within a range of w elements, with $w < BW_A$; (2) for the jth element of x, the threads accessing this element will be adjacent, within a range of h and $h < BW_A$. The first implies that with a smaller value of w, there will be denser distribution of nonzero elements in a row, and better temporal locality. The second implies that for the CUDA threads within the same thread block will access a smaller set of values in x, improving the hit ratio of the cache. If the bandwidth of a matrix can be reduced significantly by algorithms such as RCM, potentially the cache access to x can be improved.

4.2 Column index compression

For any nonzero element in a matrix with row index r and column index, we have: $BW_L < (c - r) < BW_R$, where BW_L and BW_R are the left and right bandwidth of the matrix, respectively. The smaller values of BW_L and BW_R imply a smaller range of the values of $(c - r)$, and hence the potential of compressing the column index information based on the row index. Since each row is mapped to a CUDA thread, a thread knows the row index, i.e., value of c, explicitly. Practically r have to be recorded at least in 32-bit integer format due to large size of the matrix. Value range of c usually require the same integer format for storage in ELLPACK. By recording $(c - r)$ instead of c, we can regenerate values of c by the value of r and $(c - r)$. If it is applicable that we record $(c - r)$ in a shorter format, e.g., `short` (2-byte) or `byte` (1-byte), the accessed memory amount would be reduced, at the overhead of

Table 4 Δ_0, Δ_1 and Δ_2 values for SpMV w/index reduction on C1060 and GTX-480

GPU	FLOAT			DOUBLE		
	Δ_0	Δ_1	Δ_2	Δ_0	Δ_1	Δ_2
C1060	8.135e–3	3.156e–3	2.189e–3	1.137e–2	5.972e–3	4.036e–3
GTX-480	3.290e–3	1.614e–3	1.114e–3	4.812e–3	2.412e–3	1.526e–3

generating values of c on the fly. In this article, we only consider using `short` for column index compression.

Column index compression is only applicable to some matrix, when the values of $(c - r)$ fall within the boundary of $[-32768, 32767]$. Out of the 11 test matrices, 4 matrices does not accept column index compression. But with RCM permutations, only "Web" does not allow compression. The ideal speedup of SpMV by means of index compression is the reduction ratio of the accessed data amount with shorter storage formats. The reduction ratio is 25% for matrix recorded in FLOAT, and 16.7% for that in DOUBLE. This proportion serves as the upper bound for the speedup by column index compression. Due to factors such as runtime overhead, use of HYB format, the actual speedups would be lower. Table 4 shows the measured Δ_0, Δ_1, and Δ_2 for SpMV with index compression. Compared with those in Table 2, Δ_2 is reduced with index reduction but to a lesser amount of the upperbound mentioned above, while Δ_0 and Δ_1 are generally unchanged.

5 Performance evaluation

In this section, we evaluate the performance enhancement of the SpMV optimizations proposed in previous sections. With GT-200 GPU (i.e., C1060), we evaluate the effect of reduced matrix bandwidth including both the effect on accessing x vector and index compression. Afterward, the cache-oriented optimization based on GF-100 architecture is evaluated, with index compression and reduced matrix bandwidth included.

5.1 Effect of bandwidth reduction with C1060

We first evaluate the effect of RCM permutation on t_{all} and t_x. We record the values of t_x for matrices before and after RCM permutation. Out of the 11 matrices, 8 have shown reduction in bandwidth by RCM, and 5 show speedup in terms of t_x over 5%. For these 5 matrices, the geometric mean of the speedup in t_x is 25% and 34%, for FLOAT and DOUBLE, respectively. For the 3 matrices with reduced bandwidth but no significant speedups in t_x, it is mainly due to several reasons: (1) there is substructures in some matrices which are small, dense blocks, such as "Protein", or (2) the access pattern into x is already very regular, such as "QCD." For these matrices, accesses to x is already not a performance issue and there is little effect in reducing t_x.

Table 5 summarizes the speedups on the $t_{\text{pseudo\_}x}$ and on t_{all}. Note that 10 out of 11 matrices now accepts index compression with RCM. The speedups for FLOAT and

Table 5 Performance evaluation of matrix bandwidth reduction on $t_{\text{pseudo\_}x}$ and t_{all}

Matrix	Use RCM?	Use index compression?	Speedup in $t_{\text{pseudo\_}x}$		Speedup in t_{all}	
			FLOAT	DOUBLE	FLOAT	DOUBLE
FEM/Cantiliver	No	Yes	18.7%	4.6%	11.1%	5.0%
FEM/Sphere	Yes	Yes	20.8%	15.1%	23.0%	10.9%
FEM/Accelerator	Yes	Yes	12.2%	10.3%	17.1%	32.5%
Economics	No	Yes	20.6%	17.3%	13.0%	10.6%
Epidemiology	No	Yes	23.1%	11.2%	23.1%	9.7%
Protein	No	Yes	12.0%	9.3%	9.3%	9.3%
WindTunnel	Yes	Yes	23.5%	14.7%	22.0%	14.4%
QCD	No	Yes	27.3%	6.6%	19.0%	10.1%
FEM/Harbor	Yes	Yes	9.9%	10.1%	7.7%	9.7%
Circuit	Yes	Yes	6.5%	6.7%	14.3%	17.8%
Web	Yes	No	N/A	N/A	18.0%	3.0%
GeoMean			16.2%	10.5%	16.0%	12.6%

DOUBLE are close to the theoretical speedup of 25% and 16.7%. There are several matrices that have lower speedups, such as "Circuit," "FEM/Harbor" and "Protein." The performance are actually compromised by several factors such as the use of HYB format, rather than ELLPACK format. In effect, extra COO part in HYB format does not produce performance enhancements but also introduces overhead. These factors adds up to the negative part of speedups. Although lower than the ideal speedups, the reduced amount of accessed memory yields solid speedup in $t_{\text{pseudo\_}x}$ on average: 16.2% for FLOAT and 10.5% for DOUBLE. For t_{all}, speedup is achieved for all the matrices in the test suite. The geometric mean for FLOAT and DOUBLE is 16% and 12.6%, respectively.

5.2 GF-100 based optimization

Here, we outline the three strategies for caching x on GF-100 architecture. Strategy-1: use texture fetching mechanism for x, as used in [10]. Strategy-2: use texture fetching mechanism for x, but mark accesses to A and y as volatile to avoid contamination to cache. Strategy-3: access in x through data cache, and mark access to A and y as volatile and avoid contamination to cache.

Strategy-1 is the caching strategy for x in [10], which is used as a baseline for comparison. With Strategy-1, both matrix data and vector data will occupy L2 cache, while L1 Texture Cache is still dedicated to x. Strategy-2 avoids contamination of L2 cache caused by data in A, hence vector data will consume both L1 Texture Cache and L2 cache. But due to the texture fetching mechanism is used, the latency is high. Strategy-3 is the caching strategy proposed in Sect. 3, which fully uses fast L1 data cache (configured to be 48 KB) and L2 cache for x.

We define Speedup-I as the speedup of Strategy-3 over Strategy-2, Speedup-II as that of Strategy-3 over Strategy-1, and Speedup-III as that of Strategy-3 over Strategy-1 with the combined effect of using RCM. Table 6 lists the results. Note

Table 6 Performance Enhancement for GF-100 (GTX-480) GPU

Matrix	Use RCM?	FLOAT Speedup-I	FLOAT Speedup-II	FLOAT Speedup-III	DOUBLE Speedup-I	DOUBLE Speedup-II	DOUBLE Speedup-III
FEM/Sphere	No	8.4%	14.1%		10.6%	14.7%	
FEM/Accelerator	Yes	21.2%	23.6%	29.8%	9.8%	9.6%	15.0%
Economics	No	22.3%	23.4%		9.1%	10.2%	
Epidemiology	No	10.2%	8.4%		4.7%	5.3%	
Protein	No	7.8%	13.7%		9.8%	14.0%	
WindTunnel	No	12.1%	18.7%		9.9%	21.3%	
QCD	No	16.1%	21.9%		15.7%	21.9%	
FEM/Harbor	No	7.7%	13.9%		7.3%	11.4%	
Circuit	Yes	28.8%	30.6%	31.8%	20.4%	21.0%	22.9%
Web	Yes	10.6%	10.7%	16.0%	5.9%	8.8%	14.9%
Geo-Mean		14.3%	17.7%	19.0%	10.2%	13.7%	15.0%

that Strategy-2 usually performs better than Strategy-1 (20 out of 22 cases). The caching strategy proposed in previous section outperforms that in [10] by 17% and 14% for FLOAT and DOUBLE, respectively. Speedup-I shows the comparison between the effect of caching through texture fetching mechanism and through ordinary data loads. The results show 13.6% and 10.3% speedups, which are due to 2 reasons. First, L1 data cache is configured to 48 KB (4 times size that of L1 Texture Cache); this reduces capacity misses and enhances hit ratio at the SM level. Second, a hit/miss in data cache incurs much lower latency than a hit/miss caused by Texture fetches on both L1 and L2, and this reduces the chance that the data fetch latency is not well hidden by accesses to the matrix data.

In Table 7, we compare the SpMV performance on GTX-480 and C1060. Overall, GTX-480 is faster than C1060 in SpMV performance by 50% and 75% for FLOAT and DOUBLE, respectively. Note that these two cards differ in the peak memory bandwidth: C1060 is lower than GTX-480 in memory bandwidth by about 30%, lower in frequency, and higher in global memory latency, as shown in Table 1. These hardware differences, together with the firmware differences, contribute to the performance gain shown in the table. Also, the cache subsystem difference contributes to the difference in that accesses to vector x are now faster and incurs fewer cache misses.

6 Conclusions and future work

In this article, we discuss the performance modeling and optimization of SpMV operation using ELLPACK format and NVIDIA CUDA. By separating the execution time of accesses to matrix data and dense vector data, we build performance model for the SpMV and propose various optimizations. The first categories of optimizations rely on matrix bandwidth reduction for: (1) enhanced locality in cache access for dense vector, and (2) column index compression for matrix data. The performance model

Table 7 Performance Comparison—GT-200 and GF-100 GPUs

Matrix	FLOAT			DOUBLE		
	C1060	GTX-480	Speedup	C1060	GTX-480	Speedup
FEM/Sphere	0.6661	0.4203	58.5%	1.0835	0.5579	94.2%
FEM/Accelerator	0.5665	0.3982	42.3%	0.8141	0.5270	54.5%
Economics	0.5133	0.2972	72.7%	0.6658	0.4035	65.0%
Epidemiology	0.2959	0.1733	70.8%	0.4724	0.2561	84.5%
Protein	0.7753	0.4545	70.6%	1.1226	0.5844	92.1%
WindTunnel	1.2649	0.8217	54.0%	2.0217	1.0825	86.8%
QCD	0.2105	0.1429	47.3%	0.3182	0.1863	70.9%
FEM/Harbor	0.5466	0.3802	43.8%	0.8896	0.4663	90.8%
Circuit	0.2918	0.2381	22.6%	0.4183	0.2912	43.7%
Web	0.9564	0.6203	54.2%	1.3850	0.7630	81.5%
Geo-Mean			52.9%			75.6%

reflects the enhancement in matrix-data accesses. The combined speedups achieved by these optimizations are 16% and 12.6% for single-precision and double-precision on GT-200 GPU, respectively. For GF-100 architecture with better cache support, we propose differentiated cache accesses to further enhance cache utilization with inline PTX codes. The speedups on GF-100 GPU are 19% and 16%, respectively.

In the future, we plan to combine index compression can be combined with Register Blocking in [11] for further reduction of accessed data amount in SpMV. Evaluating the performance and energy-efficiency of SpMV operations in a large scale iterative solvers, is also a future research direction. Due to the high efficiency of GPUs for SpMV, quantitative study of the optimizations on performance and total running cost could serve as valuable information for system builders for large scale computation based on these iterative solvers.

References

1. Zone CUDA. http://www.nvidia.com/cuda
2. decuda. http://wiki.github.com/laanwj/decuda
3. GPGPU.org. http://www.gpgpu.org
4. Belgin M, Back G, Ribbens C (2011) A library for pattern-based sparse matrix vector multiply. Intl J Parallel Program 39(1):62–67
5. Buatois L, Caumon G, Levy B (2009) Concurrent number cruncher—a GPU implementation of a general sparse linear solver. Intl J of Parallel, Emergent and Distributed Systems 24(3):205–223
6. Chen D, Li D, Xiong M, Bao H, Li X (2010) GPGPU-aided ensemble empirical mode decomposition for EEG analysis during anaesthesia. IEEE Trans Inf Technol BioMed 14(6):1417–1427
7. Choi JW, Singh A, Vuduc RW (2010) Model-driven autotuning of sparse matrix-vector multiply on CPUs. ACM SIGPLAN Not 45(5):115–126

8. Cuthill E, McKee J (1969) Reducing the bandwidth of sparse symmetric matrices. In: Proc 24th nat conf ACM, pp 157–172
9. Kourtis K, Goumas G, Koziris N (2008) Optimizing sparse matrix-vector multiplication using index and value compression, pp 87–96
10. Bell N, Garland M (2009) Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proc SC'09
11. Vuduc RW (2002) Automatic performance tuning of sparse matrix kernels. PhD thesis, University of California, Berkeley, 2002
12. Willcock J, Lumsdaine A (2006) Accelerating sparse matrix computations via data compression. In: Proc of the 20th annual intl conf on supercomputing, ICS '06. ACM, New York, pp 307–316
13. Williams S, Oliker L, Vuduc R, Shalf J, Yelick K, Demmel JW (2007) Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proc 2007 ACM/IEEE conference on supercomputing, SC '07. ACM, New York, pp 38:1–38:12
14. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. SIAM, Philadelphia