Optimization Techniques for Sparse Matrix-Vector Multiplication on GPUs

Marco Maggioni^{a,*}, Tanya Berger-Wolf^a

^a*Department of Computer Science, University of Illinois at Chicago
851 S. Morgan, Room 1120 SEO, Chicago, IL 60607-7053, USA*

Abstract

Sparse linear algebra is fundamental to numerous areas of applied mathematics, science and engineering. In this paper, we propose an efficient data structure named AdELL+ for optimizing the SpMV kernel on GPUs, focusing on performance bottlenecks of sparse computation. The foundation of our work is an ELL-based adaptive format which copes with matrix irregularity using balanced warps composed using a parametrized warp-balancing heuristic. We also address the intrinsic bandwidth-limited nature of SpMV with warp granularity, blocking, delta compression and nonzero unrolling, targeting both memory footprint and memory hierarchy efficiency. Finally, we introduce a novel online auto-tuning approach that uses a quality metric to predict efficient block factors and that hides preprocessing overhead with useful SpMV computation. Our experimental results show that AdELL+ achieves comparable or better performance over other state-of-the-art SpMV sparse formats proposed in academia (BCCOO) and industry (CSR+ and CSR-Adaptive). Moreover, our auto-tuning approach makes AdELL+ viable for real-world applications.

© 2015 Published by Elsevier Ltd.

Keywords:

SpMV, optimization, GPU, adaptive, AdELL+, blocking, compression, unrolling, auto-tuning

*Corresponding author: 1-(312)-996-8635

Email addresses: mmaggi3@uic.edu (Marco Maggioni), tanyabw@uic.edu (Tanya Berger-Wolf)

1. Introduction

Sparse Matrix-Vector multiplication (SpMV) is a fundamental computational kernel (it is one of the *seven dwarves* [1]) and its efficiency is crucial for numerous applications in science and engineering. The widespread importance of sparse matrix computation has resulted in significant research efforts poured into implementations based on modern-day parallel computer architectures, allowing the solution of increasingly complex computational problems in areas such as linear algebra [2], convex optimization [3], data mining [4] and graph analytics [5]. In general, the research community focused on providing every possible improvements under the valid assumption that so many applications can benefit from better performance, however incremental the approach.

The need of pursuing better implementations based on Graphic Processing Units (GPUs) has been recently questioned by Davis and Chung [6]. Their argument was based on the narrowing memory bandwidth gap between CPUs and GPUs making GPU-based implementations not really worth the additional effort. It might be observed that the bandwidth gap shows a cyclic behavior, where a new fast memory technology is firstly introduced on GPUs and then adopted later by CPUs (indeed, the cycle is about to repeat with Stacked DRAM on the future NVIDIA Pascal architecture). Hence, there will always be an edge in terms of memory bandwidth from which real-world sparse linear algebra computation can benefit. On the other hand, performance-per-watt is the metric that is driving the evolution of large scale scientific computing. Many-core architectures are well-known for their power-efficiency [7]. Therefore, GPU-based implementations will continue to be a relevant research topic with solid practical applications.

Over the last decade, SpMV optimization has been extensively studied on many-core GPUs, with the aim of exploiting the available fine-grain parallelism and memory bandwidth. Despite having a fairly straightforward sequential implementation, the SpMV kernel is a pure memory bound problem due to its extremely low arithmetic intensity and potentially irregular memory patterns. As a consequence, its parallel performance is generally reported to sustain low fractions of computational peak [8]. Several hand-optimized GPU-based sparse matrix formats achieved impressive progresses targeting specific structural features of the underlying problems. However, we conjecture that the SpMV kernel can be further improved by addressing its performance bottlenecks in a more explicit way. In this paper, we propose to integrate several performance optimization techniques (each one targeting a complementary goal) into AdELL+, an efficient data structure for sparse linear algebra computation.

1.1. Related work

The literature on implementing SpMV on throughput-oriented manycore processors is extensive. The main focus has been on proposing and tuning novel formats to efficiently represent the sparse matrix, with the aim of optimizing memory pattern efficiency, memory footprint, load balancing and thread divergency. Bell and Garland presented perhaps the best-known work on GPU-based SpMV [9], including basic sparse matrix formats such as COO, DIA, CSR, ELL, and a new hybrid format HYB (which combines ELL and COO). The vendor-tuned library cuSPARSE [10] provides a well-maintained implementation of the above formats. However, none of them is consistently superior since each has been designed to leverage a specific matrix structure. COO and CSR are suited for unstructured matrices, with COO rarely used in practice due to its space inefficiency. ELL is instead suited for vector architectures and performs well with regular structures. DIA provides a clear advantage for strongly diagonal matrices. Several derivative formats have been subsequently proposed in order to improve the baseline performance. ELL-R [11] uses an auxiliary data structure to store row lengths and avoid wasteful computation. Baskaran and Bordawekar [12] improve the CSR-based SpMV kernel by assuring the memory alignment and by caching values repeatedly accessed. Another format known as SELL [13] (and its generalization for heterogenous vectorized units SELL-C- σ [14]) reduces the memory footprint of the basic ELL format by horizontally partitioning the original matrix into several slices.

Choi *et al.* [15] added the idea of blocking to CSR and ELL, creating two novel sparse formats known as BCSR and BELL. This technique is beneficial for matrices with block substructure and intrinsically reduces the memory footprint (by using indices to entire blocks instead of individual nonzero entries). Row and column reordering is another strategy than can be used to reduce matrix bandwidth and improve cache

locality during SpMV execution [16]. However, original matrices often have inherent locality so reordering may not have any practical benefit (especially considering the preprocessing overhead). There are some works focusing on compression to reduce the matrix footprint and, consequently, improve the bandwidth-limited SpMV kernel. A bit-representation index compression scheme has been proposed by Tang *et al.* [17]. Another work by Xu *et al.* [18] focused on index compression for matrices with diagonal structure. In general, the challenge of compression techniques is the complexity of decompression. This operation becomes embedded into the SpMV kernel and should be as lightweight as possible in order to not cancel out the performance gain (which is also limited by the intrinsic matrix compressibility). Another approach to SpMV optimization is tuning the sparse format parameters (*e.g.* block size or slicing factor) in order to achieve the best possible performance. This is usually done by the means of auto-tuning frameworks that explore the parameter search space (usually a reasonable subset). While auto-tuning is beneficial, it also introduces additional preprocessing and benchmarking costs that should be accounted for. The success of ensemble approaches like cSpMV [19] heavily relies on auto-tuning. Since different matrix sparsity patterns can be related to specific optimizations, the cSpMV framework provides an attractive approach based on analyzing the matrix structure and appropriately selecting the best representation (or a combination of these) among an ensemble of many available GPU-based sparse formats.

The greatest challenge in SpMV optimization is to cope with the irregular matrix structure. Row-based parallelization (*i.e.* assigning one working thread to each row) leads to load imbalance since nonzeros are unevenly distributed across different rows. Roughly speaking, this problem can be solved by distributing threads to rows according to their computational load, a technique known as adaptivity. The first step towards the implementation of this technique was the ability to use collaborative threads to process a single row (as opposed to the row-based parallelization just described). Advanced sparse matrix formats such as ELL-T [20], SIC [21], CMRS [22] or RgCSR [23] use slightly different approaches to map a row to one or more threads in a warp. Unfortunately, this thread mapping is not flexible (*i.e.* each row is mapped to exactly t threads, where t is a parameter) hence unbalanced workloads are still problematic. A more flexible implementation was then proposed by Heller and Oberhuber [24]. However, their ArgCSR composes balanced blocks using a very simple policy that fails in the case of very skewed nonzero distributions (*e.g.* matrices with a single dense row). A complete (and arguably superior) adaptive sparse matrix format is provided by AdELL and its warp-balancing heuristic to compose warps [25]. More recent work on adaptivity has used two-dimensional blocking for composing balanced computation (BRC [26]) and dynamic parallelism for processing heavyweight rows (ACSR [27]).

Focusing on pure performance, the state-of-the-art in SpMV optimization is arguably BCCOO [28]. This advanced sparse matrix format is an evolution of COO based on blocking and row indices compression, where load balancing is achieved by the means of a highly-efficient segmented reduction (which, however, relies on a non-portable synchronization-free mechanism that stalls on modern AMD GPUs). BCCOO, as many other advanced sparse formats, introduces a significant preprocessing overhead for data structure generation and tuning. This set-up cost is acceptable in applications where it can be amortized over repeated SpMV iterations on the same matrix or on matrices with the exact same sparse structure. For this reason, GPU vendors focused their research on performance with lightweight preprocessing over CSR. NVIDIA Research recently released the ModernGPU library [29], a collection of advanced GPU coding approaches including CSR+. Based on segmented reduction for load balancing, CSR+ provides very promising performance with negligible preprocessing overhead. On the same note, AMD Research recently proposed CSR-Adaptive [30]. With a slightly different interpretation of adaptivity, CSR-Adaptive uses an unconventional (compared to previous work) approach to process short/medium rows (segmented reduction over nonzeros buffered in local memory).

Despite the impressive progresses achieved in SpMV optimization, we observe that no previous work has ever explicitly combined different optimization techniques in order to effectively address all the performance bottlenecks of sparse linear algebra computation. This, arguably, provides a promising approach to improve the state-of-the-art, motivating the research work presented in this paper. Our approach combines in an efficient and structured way the existing techniques to achieve maximum performance. In addition, we recognize the need to keep the preprocessing as small as possible in order to amortize the additional overhead.

1.2. Contribution

The main contribution of our research is combining the leading ideas in SpMV optimization on GPUs into a lightweight, general, adaptive, efficient and high-performance method for sparse linear algebra computation. This result has been achieved by proposing AdELL+, an advanced sparse matrix format that explicitly addresses the performance bottlenecks of the SpMV kernel. Building upon our previous work [25, 31], we carefully composed several optimization techniques into a data structure with warp granularity that naturally suits the Single Instruction Multiple Data (SIMD) vectorization associated with many-core GPUs. AdELL+ has several new distinctive advantages that we exploit to reach a substantial performance edge over the state-of-the-art BCCOO. First, it is derived from ELL (a general sparse format well-suited for vectorization and coalesced memory access) but makes no assumption on the sparsity structure. Second, load balancing is achieved by embedding balanced warps directly into the data structure itself rather than solely relying on a (non-portable) global segmented reduction. All the collaborative operations can be then implemented using fast warp intrinsic (plus few atomics in case of heavyweight rows processed using multiple warps), achieving a portable, adaptive and efficient solution. Third, the preprocessing and tuning cost of AdELL+ is more lightweight than BCCOO thanks to the proposed online auto-tuning approach guided by a novel quality metric. In addition, AdELL+ is competitive with CSR+ and CSR-Adaptive for those applications where preprocessing cannot be easily amortized. More specifically our contributions are as follows.

- A transparent integration of complementary memory optimization techniques into AdELL [25]. We address the bandwidth-limited nature of the SpMV kernel by using a new format based on warp granularity, blocking, and delta-based index compression (to reduce the memory footprint) plus nonzero unrolling (to improve the memory hierarchy utilization). We also provide an in-depth analysis of those techniques, devising a theoretical quality metric that can be used for performance evaluation and prediction.
- A parametrized adaptive warp-balancing heuristic to implement the idea of adaptive warp-balancing. Our greedy heuristic distributes workload to hardware-level blocks (warps), with the aim to create a balanced (and efficient) AdELL+ structure. Differently from what was originally proposed [25], we process the matrix incrementally without changing the row ordering. This is beneficial for two reasons. First, we exploit the locality of the original matrix. Second, we avoid the layer of complexity associated with reordering (*i.e.* storing the matrix mapping). Thus, this approach provides comparable results in terms of warp-balancing with lower processing cost.
- A novel online auto-tuning approach based on the aforementioned quality metric. This metric allows a much faster evaluation of performance and a quick search of the parameter space. The underlying idea is to progressively explore the space of optimization parameters with line search while the actual SpMV kernel is doing useful computation, completely subsuming all the preprocessing overhead associated with composing AdELL+.

We evaluated floating-point performances of our approach on a benchmark suite composed of 14 regular and 6 irregular matrices from heterogeneous application domains. Our experimental results on Tesla K40 showed that our proposed approach consistently achieves comparable or better performance over BCCOO (1.22x speedup on regular and 1.05x on irregular benchmarks). Those results were replicated over other state-of-the-art sparse formats (CSR+ and CSR-Adaptive) with even larger margins. The high level of optimization achieved by AdELL+ can be also proved by its effective memory bandwidth (close to the peak achievable on Tesla K40). The widespread importance of SpMV (and the massive effort poured by the research community into its optimization) makes this result greatly relevant in practice. Moreover, the proposed online auto-tuning approach makes AdELL+ directly applicable for real-world applications.

The rest of the paper is organized as follows: Section 2 reviews the existing SpMV formats and related work. In Section 3, we introduce warp-granularity into the ELL format. Section 4 describes the loop unrolling technique. Section 5 and 6 discuss two techniques to improve the memory footprint. Section 7 introduces the adaptivity technique and our warp-balancing heuristic. Section 8 puts all those techniques

together into AdELL+. Section 9 explains how to efficiently perform autotuning. Section 10 provides a comparison with the state-of-the-art in SpMV optimization. Section 11 focuses on bandwidth and cache memory considerations. Last, Section 12 is devoted to conclusions.

2. Background

The SpMV is, arguably, the most important kernel in sparse matrix computations. In this paper, we specifically refer to the operation $\mathbf{y} = \mathbf{y} + \mathbf{A}\mathbf{x}$, where $\mathbf{y}$ and $\mathbf{x}$ are dense vectors of length n and m , respectively, while $\mathbf{A}$ is a sparse matrix of size $n \times m$. Typical matrices in applications have thousands or even millions of rows and columns with only a small fraction of nonzeros entries. Calculation of the SpMV essentially reduces to many independent dot products such as the following

$$y_i = y_i + \sum_{\forall j: a_{ij} \in \mathbf{A}_i} a_{ij}x_j, \quad (1)$$

where $\mathbf{A}_i$ is a sparse row vector of the matrix $\mathbf{A}$ and a_{ij} corresponds to its nonzero entries. While SpMV computation may seem intrinsically parallel, several challenges are faced in achieving high performance. First, an irregular structure may lead to an unbalanced workload with uneven nonzeros across different working threads. Such imbalance problem is even more severe in many-core GPUs since warps operate in SIMD fashion, leading to potential control divergency (*e.g.* a thread processing a heavyweight row that forces adjacent threads idle). Second, the bandwidth-limited nature of the SpMV kernel puts high pressure on the memory hierarchy. Each nonzero a_{ij} in matrix $\mathbf{A}$ is only used once for computing the correspondent dot product, whereas the elements of the dense vector $\mathbf{x}$ may be reused across different dot products (although we expect poor locality in case of irregular structure). Roughly speaking, SpMV can be seen as a streaming computation, where performance is limited by the bandwidth at which matrix data can be streamed from memory. Moreover, modern GPUs can benefit from a cache hierarchy (usually organized on two levels) that performs best in case of aligned and coalesced memory operations (*e.g.* different threads in a warp accessing data within the same cache line).

2.1. Sparse matrix characterization

The sparse structure provides the opportunity to represent a matrix in memory by only its nonzero elements (plus additional information associated with row and column indices) as opposed to using a two-dimensional dense array (which would considerably limit sizes of computed problems). In some cases, the pattern of nonzero elements is quite regular (*e.g.* for problems resulting from discretization on regular grids),

Matrix	Dimension			Nonzeros per Row	
	<i>nmz</i>	<i>n</i>	<i>m</i>	μ	σ
Circuit	170998	170998	958936	5.61	4.39
Dense	4000000	2000	2000	2000.00	0.00
Economics	1273389	206500	206500	6.17	4.44
Epidemiology	2100225	525825	525825	3.99	0.08
FEM/Accelerator	2624331	121192	121192	21.65	13.79
FEM/Cantilever	4007383	62451	62451	64.17	14.06
FEM/Harbor	2374001	46835	46835	50.69	27.78
FEM/Ship	7813404	140874	140874	55.46	11.07
FEM/Spheres	6010480	83334	83334	72.13	19.08
Ga41As41H72	18488476	268096	268096	68.96	105.39
Protein	4344765	36417	36417	119.31	31.86
QCD	1916928	49152	49152	39.00	0.00
Si41Ge41H72	15011265	185639	185639	80.86	126.97
Wind Tunnel	11634424	217918	217918	53.39	4.74

Table 1. Description of the regular benchmark suite from the University of Florida Sparse Matrix Collection [32].

Matrix	Dimension			Nonzeros per Row	
	nnz	n	m	μ	σ
Circuit5M	59524291	5558326	5558326	10.71	1356.62
Eu-2005	19235140	862664	862664	22.30	29.33
In-2004	16917053	1382908	1382908	12.23	37.23
LP	11279748	4284	1092610	2632.99	4209.26
Mip1	10352819	66463	66463	155.77	350.74
Webbase	3105536	1000005	1000005	3.11	25.35

Table 2. Description of the irregular benchmark suite from the University of Florida Sparse Matrix Collection [32]

but the most interesting (and challenging) matrices are those whose distribution of nonzeros per row appears to be highly variable. The matrix sparsity pattern can be roughly evaluated by some simple metrics based on the number of nonzeros per row in $\mathbf{A}$. We can immediately calculate the average nonzeros per row μ and the associated standard deviation σ . Thus, we can get an approximate idea of the nonzero distribution (and, consequently, a sense of the load balancing) by looking at the minimum, median and maximum nonzeros per row along with its binned histogram.

Here we introduce the characterization of the benchmark suite used to provide a comprehensive evaluation of our work. All the sparse matrices (except a synthetically generated dense one) are downloadable from the well-known University of Florida Sparse Matrix Collection [32] as Matrix Market files [33]. As suggested by Langr and Tvrdik [34] in their evaluation criteria for sparse matrix formats, we provide a fair comparison by choosing relatively large matrices that have been widely adopted in previous SpMV optimization research [8, 9, 13–15, 19, 21, 25, 26, 28, 30, 31]. This benchmark suite contains 20 sparse matrices of heterogenous sizes, structures, number of overall nonzeros (*i.e.* nnz) and application domains. We categorize them as regular (14 matrices in Table 1) and irregular (6 matrices in Table 2), based on the coefficient of variation $\frac{\mu}{\sigma}$ as well as on the distribution of the nonzeros per row.

The adopted classification is more evident from Tables 3 and 4. Each irregular matrix has a large range $[min, max]$ and its associated histogram (built with uniform bins within the range $[0, max]$) highlights a skewed distribution of nonzeros per row. Due to the large range, rows falling in the last bins account for a significant percentage of the total nonzeros (*e.g.* 13% of total nonzeros in Circuit5M are associated with rows falling in the last bin). Clearly, this sparsity pattern is hard to parallelize. On the other hand, regular matrices distribute their rows more freely along the range of nonzeros per row. Some benchmarks, such

Matrix	Nonzeros per Row			Binned Histogram									
	min	med	max										
Circuit	1	5	353										
Dense	2000	2000	2000										
Economics	1	5	44										
Epidemiology	2	4	4										
FEM/Accelerator	0	23	81										
FEM/Cantilever	1	74	78										
FEM/Harbor	4	46	145										
FEM/Ship	24	54	102										
FEM/Spheres	1	81	81										
Ga41As41H72	18	37	702										
Protein	18	123	204										
QCD	39	39	39										
Si41Ge41H72	13	37	662										
Wind Tunnel	2	54	180										
Legend (Nonzero Density)				≤10%	≤20%	≤30%	≤40%	≤50%	≤60%	≤70%	≤80%	≤90%	≤100%

Table 3. The distribution of nonzeros per row in the regular matrices. The bins are uniformly spaced in the range $[0, max]$ and highlighted according to their density (zero is equivalent to absence of nonzeros in the bin).

Matrix	Nonzeros per Row			Binned Histogram										
	<i>min</i>	<i>med</i>	<i>max</i>											
Circuit5M	1	5	1290501											
Eu-2005	0	16	6985											
In-2004	0	5	7753											
LP	1	1158	56181											
Mip1	4	40	66395											
Webbase	1	2	4700											
Legend (Nonzero Density)				≤10%	≤20%	≤30%	≤40%	≤50%	≤60%	≤70%	≤80%	≤90%	≤100%	

Table 4. The distribution of nonzeros per row in the irregular matrices.

as QCD, are perfectly balanced since $\sigma = 0$. Some other benchmarks, such as Circuit or Ga41As41H72, still appear skewed but their smaller range (compared with irregular matrices) makes the computing load associated with long row less severe.

2.2. Basic sparse matrix formats

In this subsection, we provide a brief description of the ELL sparse matrix representation (and its GPU-based SpMV kernel) as the necessary foundation to understand the remaining of this paper. ELL is a sparse format particularly well-suited to SIMD vectorized architectures. Its basic idea is to compress the sparse $n \times m$ matrix using a dense $n \times k$ data structure, where k corresponds to *max* (the maximum number of nonzeros per row). As shown by Figure 1, the sparse matrix **A** is stored in memory by the means of two 4×3 dense arrays *Value[]* and *Column[]* (row indices remain implicit as in a dense matrix). Hence, zero-padding is necessary for rows with less than $k = 3$ elements. Indeed, the memory footprint associated with strictly depends by the matrix regularity. Whenever rows approximately have the same number of nonzeros, ELL can be as efficient as CSR. On the other hand, skewed distributions may lead to a unfeasible memory footprint (*i.e.* as large as the dense matrix). Similarly to CSR, ELL works at the granularity of thread per row. However, its memory layout is optimized for coalesced access by adopting column-major ordering. Figure 2 provides a detailed view on how data are stored in memory for efficiency. Due to column-major ordering, a vectorized warp-level instruction (for the sake of simplicity $w = 4$) can process a batch of w distinct nonzeros (*e.g.* [a,d,f,h]) by coalescing the loading of adjacent memory addresses. Given a matrix of arbitrary size, it may be also necessary to add few padding rows at the bottom of the two $n \times k$ arrays in order to guarantee w -alignment (indeed, this optimization is a consequence of how GPU memory hierarchy is designed). In other words, we round n to the next multiple of w (*i.e.* $n' = \lceil \frac{n}{w} \rceil \cdot w$). ELL certainly achieves good performance on regular structures (*i.e.* with an equal number of nonzeros on each row). On the other hand, irregular matrices inevitably lead to memory footprint inefficiency and waste of computation

Fig. 1. Sparse matrix **A** [a] represented as COO [b], CSR [c] and ELL [d].

Fig. 2. ELL sparse format [a] and its column-major memory layout [b].

(i.e. short rows make their thread idle for most of the time). We can quantify efficiency as

$$e^{ELL} = \frac{nnz}{n' \cdot k}, \quad (2)$$

where nnz is the overall number of nonzeros in $\mathbf{A}$ and e^{ELL} is a $[0, 1]$ metric that measures the portion of actual nonzeros in $Value[]$ and $Column[]$ (intuitively, $1 - e^{ELL}$ corresponds to padding).

Using ELL as reference sparse format, we can provide a roofline-type [35] performance model to characterize the SpMV kernel as bandwidth-limited. Assuming no latency effect (due to an effective hardware multithreading) and infinitely fast caches, the arithmetic intensity i (the ratio between performed floating-point operations and bytes moved from/to memory) associated with each nonzero can be calculated as

$$i = \left(\frac{2}{m_A + m_x + m_y} \right) \frac{\text{Flops}}{\text{Byte}}. \quad (3)$$

Here the factor 2 comes from the dot product (i.e. floating-point multiply and add), m_A accounts for reading the nonzero value and column index from $\mathbf{A}$, m_x is the traffic incurred by partially indirect access to vector $\mathbf{x}$, and m_y is the data volume for updating the result $\mathbf{y}$. Assuming single-precision and four-byte integer indices, each nonzero is represented by $m_A = 4 + 4$ bytes. However, we should also take into account the efficiency e^{ELL} itself (which may increase the effective amount of loaded data). The effect of the memory hierarchy on accessing vector $\mathbf{x}$ is characterized by a factor $\alpha \in [0, 1]$. In the worst case scenario, every access causes a cache miss so $\alpha = 1$. In the ideal situation, we only account for cold misses (i.e. those necessary to fetch $\mathbf{x}$ from global memory) so each cache element will be reused a number of times equal to the average number of nonzeros per column ($\alpha = 1/\mu$ for square matrices). The cost of updating $\mathbf{y}$ elements (one read and one write) is instead amortized on μ nonzeros when using a thread to process a row. All those concepts are substituted in formula (3) to achieve the following expression

$$i = \left(\frac{2}{(4 + 4)/e^{ELL} + 4\alpha + 2 \cdot 4/\mu} \right) \leq 0.250 \left[\frac{\text{Flops}}{\text{Byte}} \right]. \quad (4)$$

Let us assume the best case where $e^{ELL} = 1$, $\alpha \approx 0$ and μ is large. We can then derive an upper bound of 0.250 for a single-precision arithmetic intensity. Let us now consider the specific Tesla K40 GPU based on Kepler architecture [36] used in our experimental section. Given a 207.5 GB/s measured peak bandwidth with ECC disabled (as opposed to 288.4 GB/s theoretical), the upper bound on SpMV performance measured in Floating-points Operation Per Second (or FLOPS) is 51.88 GFLOPS. This upper bound is only a small fraction of the 4.29 TFLOPS Tesla K40 theoretical computational peak, proving that SpMV is a bandwidth-limited kernel due to its low arithmetic intensity. We can repeat the same analysis for double-precision computation obtaining

$$i = \left(\frac{2}{(8 + 4)/e^{ELL} + 8\alpha + 2 \cdot 8/\mu} \right) \leq 0.167 \left[\frac{\text{Flops}}{\text{Byte}} \right], \quad (5)$$

which corresponds to an even more capped upper bound of 34.58 GFLOPS, still a small fraction of the double-precision 1.43 TFLOPS Tesla K40 theoretical peak. Until here, we assumed an ideal cache behavior. However, in reality, the irregular memory access to the dense vector $\mathbf{x}$ could hamper performance, especially

in case of scattered accesses. Despite being optimized for throughput, modern GPU architectures provide a cache hierarchy composed by a fully coherent L2 cache and by several local memories, each one associated with a Streaming Multiprocessor (or SMX on Kepler architecture). These latter are further divided into shared memory (*i.e.* a software-managed cache) and texture memory (*i.e.* a hardware-managed read-only cache). Despite being significantly smaller than their CPU counterparts, this cache hierarchy is a fairly efficient (and effortless) way to deal with irregularity. According to our computational experiment, L2 cache has an average hit rate of 63.77%, whereas read-only texture can improve the performance by an average factor of 1.26x. On the other hand, shared memory (and its small capacity) is not useful for efficiently caching the access pattern to the dense vector $\mathbf{x}$.

3. Improving ELL with warp granularity

A common optimization pattern in GPU computing consists of tailoring both the data structure and the computation around the SIMD width (*i.e.* warp size). On the one hand, this provides an easy way to guarantee coalesced and aligned memory accesses. On the other hand, computation that involves only threads in a warp has a better performance due to its ability of exploiting implicit lockstep execution for synchronization rather than explicit primitives [37]. Similarly, collective operations between threads can be efficiently performed by building upon warp intrinsics (*e.g.* shuffle instruction) rather than relying on local/shared or global memory. Inspired by this ideas, we have proposed an optimized sparse format known as WELL (Warp-grained ELL) [38]. WELL partitions the matrix into (warp-sized) slices and represents them with local ELL data structures, leading to a fundamental advantage in terms of the overall memory footprint. Intuitively, the dimension size k_i of each slice depends on the longest row in the warp, rather than a global k . WELL requires two additional vectors $K[]$ and $Offset[]$ of dimension $n_w = \lceil \frac{n}{w} \rceil$ (the number of warps) and $n_w + 1$. $K[]$ is used to keep track of local k_i , whereas $Offset[]$ is used to identify the incremental starting location of each local ELL structure within $Value[]$ and $Column[]$.

Fig. 3. An arbitrary matrix represented as ELL [a] and composed by two warps ($w = 4$). WELL [b] provide a more efficient representation with an obvious memory footprint improvement from 24 (8×3) to 16 entries (in general, we can assume $K[]$ and $Offset[]$ to have negligible contribution since $n_w \ll n \ll nnz$).

Figure 3 shows an example of WELL and depicts the advantage of using WELL over ELL. The memory footprint improvement can be quantified by updating the efficiency formula (2) as

$$e^{WELL} = \frac{nnz}{\sum_{i=1}^{n_w} w \cdot k_i} = \frac{nnz}{Offset[n_w]} \quad (6)$$

where the summation at the denominator takes into account each local ELL contribution to the overall WELL memory footprint. We evaluated e^{WELL} for our regular benchmark suite using both (2) and (6), obtaining the results in Table 5. Using warp granularity, WELL substantially improves over e^{ELL} by a 4.73x factor (calculated using harmonic mean), leading to immediate benefits in terms of memory footprint and, hence, the overall SpMV computation. Table 6 shows the results for the other irregular matrices. The extremely low e^{ELL} associated with ELL is alone sufficient to justify that WELL is the only viable option between the two sparse formats to represent irregular matrices (*i.e.* ELL is so inefficient that some matrices do not even fit in memory).

Matrix	Efficiency	
	ELL	WELL
Circuit	0.016	0.398
Dense	0.992	0.992
Economics	0.140	0.275
Epidemiology	0.998	0.999
FEM/Accelerator	0.267	0.840
FEM/Cantilever	0.823	0.881
FEM/Harbor	0.349	0.663
FEM/Ship	0.544	0.846
FEM/Spheres	0.890	0.930
Ga41As41H72	0.098	0.364
Protein	0.584	0.796
QCD	1.000	1.000
Si41Ge41H72	0.122	0.410
Wind Tunnel	0.297	0.982

Table 5. Efficiency of ELL versus WELL for the regular benchmarks.

Matrix	Efficiency	
	ELL	WELL
Circuit5M	0.000	0.247
Eu-2005	0.003	0.457
In-2004	0.002	0.367
LP	0.047	0.220
Mip1	0.002	0.698
Webbase	0.001	0.316

Table 6. Efficiency of ELL versus WELL for the irregular benchmarks.

WELL has been designed to promote efficient SIMD vectorization at warp level. Intuitively, each local ELL structure is mapped to a warp that executes independently by any other warp in the SpMV kernel. Moreover, there is a conceptual analogy between threads in a warp and SIMD lanes (it is possible to use these two terms interchangeably in the context of warp computation). It might be argued that WELL memory layout intrinsically guarantees coalescing and alignment. In fact, each local ELL structure is still stored using column-major ordering for coalescing. Moreover, each local ELL structure has size $w \times k_i$ such that each and every offset in $Offset[]$ is w -aligned. First, we do not apply local rearranging in order to avoid the extra layer of complexity (benefiting from the intrinsic locality of the original matrix). Second, we use texture (read-only) cache to map the vector $\mathbf{x}$. Specifically, each memory access to $\mathbf{x}$ will pass through a separate 48KB hardware-managed cache. Differently from the coherent L2 cache, this read-only memory is local to each SMX and cannot be polluted by the nonzeros streamed from global memory into the GPU during the SpMV kernel. Third, we let the padding zeros to be processed the same way as the rest of the nonzeros in the matrix. This additional arithmetic performs the same $0 \cdot x_0$ operation over and over, not introducing overhead due to the lockstep execution. Similarly, repeated memory accesses to x_0 are cheap due to caching. In other words, we gain in compile efficiency producing a control divergence free code and maintaining a uniform data structure for the entire matrix.

WELL is very similar to SELL [13] but its implementation of warp granularity goes beyond the simple choice of slices matching the warp size ($w = 32$ for Kepler architecture). GPU programming models such as CUDA [39] or OpenCL [40] logically organize threads into blocks that are mapped to vectorized units (SMXs) and then broken down to warps for the actual execution on the GPU hardware. It is well-known that block size b must be chosen carefully to not undermine performance. The key difference between SELL and WELL is that the former binds slice size s to block size b . Hence, warp granularity $s = w$ (which is the best in terms of e^{WELL}) provides a poor choice in terms of block size (*i.e.* $b = w$ achieves very low utilization of the GPU hardware architecture). On the other hand, WELL implements warp granularity by

decoupling s from b . In other words, we are able to launch SpMV kernels with optimal b while the data structure, as well as the actual computation, can still take advantage of warp granularity. In addition, WELL has a synchronization free execution whereas SELL blocks may need primitives to synchronize their warps.

4. Unrolling nonzeros to optimize memory hierarchy efficiency

Loop unrolling is a well-known technique in compiler optimization used to expose Instruction Level Parallelism (ILP) or to reduce the overhead associated with the loop control structure. Murthy *et al.* [41] provide an in-depth analysis of loop unrolling in the context of GPU programming. In general, the technique is transparently applied to optimize the execution of short loops with a fixed number of iterations (*e.g.* when processing a dense 2×2 matrix subblock). However, loop unrolling can be also useful in the context of bandwidth-limited kernels such as SpMV. Let us assume to have a loop where data (*e.g.* nonzeros) are loaded from memory and processed at each iteration. Memory transactions will be issued in a serialized way leading to potential memory latency inefficiency. On the other hand, unrolling allows to issue multiple independent memory transactions per iteration. This scenario clearly facilitates latency hiding. Indeed, those memory transactions are served simultaneously, improving the overall bandwidth utilization. Hence, loop unrolling is able to optimize the GPU memory hierarchy efficiency by increasing the number of on-the-fly memory transactions.

Fig. 4. Processing nonzeros without unrolling [a] and with 2x unrolling [b]. The example focuses on a single warp taken from an arbitrary sparse format with warp granularity (*e.g.* WELL). In red, we schematized the local register buffering and the dot product operation. As we can observe, 2x unrolling needs only 2 iterations (as opposed to 4) to process lane 0.

Figure 4 shows the unrolling technique applied to the SpMV kernel. Without unrolling, we have a loop that iterates k_i times over the nonzeros. For each iteration, we load both the nonzero value and column from matrix $\mathbf{A}$, we perform an indirect access to vector $\mathbf{x}$ and then we accumulate the dot product (*i.e.* multiply and add). Let us consider lane 0 that processes the first row. During the first iteration, value a and column 0 are loaded into two local registers, x_0 is fetched into another local register and then $a \cdot x_0$ is computed. With unrolling, we aggressively buffer multiple nonzeros (and $\mathbf{x}$ elements) into local registers at each loop iteration (*e.g.* lane 0 buffers $[a, b]$, $[0, 1]$ and then $[x_0, x_1]$). Therefore, this technique (that we call nonzero unrolling) allows to issue multiple memory transactions simultaneously, improving bandwidth utilization and latency hiding. The overall number of iterations k_i may not be a multiple of the loop unrolling factor lu . In that case, buffering will simply pad the buffers with few zeros during the last iteration. This approach clearly promotes code regularity. In fact, the added padding does not change the way in which buffers are processed (similarly, it does not affect the result correctness).

4.1. Empirical evaluation

Here we take time to explain the protocol for all our computational experiments (including those that will be presented in the next sections). Those were performed on a single NVIDIA Tesla K40 (2880@745MHz CUDA cores) equipped with 12GB@3004MHz ECC GDDR5 memory. We implemented all the necessary GPU-based SpMV kernels using CUDA (nvcc compiler version 6.5), and we measured the average performance in GFLOPS over 100 repetitions without considering additional overhead (*e.g.* moving the sparse matrix from GPU to GPU). We also disabled the Tesla K40 ECC feature and we used the 48KB texture(read-only) memory for caching vector $\mathbf{x}$. Our first goal is to provide an empirical analysis of the

Matrix	Performance [GFLOPS]									
	1x	2x	4x	8x	16x	1x	2x	4x	8x	16x
Circuit	4.503	5.752	9.091	11.273	12.007	4.138	5.003	7.849	8.583	9.229
Dense	4.133	5.463	10.283	14.722	16.163	4.020	5.212	12.239	14.060	15.526
Economics	8.783	10.484	12.746	13.333	13.122	7.354	7.996	8.978	9.040	8.786
Epidemiology	31.828	35.049	38.945	38.568	26.441	24.341	24.701	25.991	24.487	17.110
FEM/Accelerator	26.837	32.231	36.246	38.271	37.987	21.827	24.481	24.734	25.389	25.572
FEM/Cantilever	28.763	33.938	41.716	42.804	44.253	23.434	25.754	28.687	30.096	30.475
FEM/Harbor	19.179	22.868	29.796	31.136	31.382	16.606	18.420	21.532	21.746	21.989
FEM/Ship	31.516	36.943	43.662	44.148	44.159	25.967	27.538	29.232	29.721	29.728
FEM/Spheres	35.684	41.001	47.739	47.736	48.005	29.045	30.477	31.298	32.032	32.555
Ga41As41H72	12.015	14.550	18.115	18.684	19.268	10.353	11.425	11.901	12.469	12.898
Protein	21.907	26.030	34.842	37.720	40.783	19.121	21.457	25.349	27.391	26.713
QCD	34.674	40.032	46.969	47.929	47.816	29.046	30.843	31.635	32.661	32.044
Si41Ge41H72	13.491	16.219	20.230	20.670	21.406	11.594	12.672	13.748	13.884	14.341
Wind Tunnel	32.003	36.096	44.783	47.784	48.713	26.777	28.866	32.029	33.089	33.254
Single Precision					Double Precision					

Table 7. Performance results for the nonzero unrolling with WELL on the regular benchmarks.

effectiveness of the proposed nonzero unrolling technique in terms of performance. Table 7 reports the SpMV performance on the regular benchmark suite using WELL kernels with different lu factors. These results can be generalized as long as warp granularity holds. Not surprisingly, higher unrolling factors lead to better performance due to a better utilization of the GPU memory hierarchy. The best result (highlighted in green) is usually associated with 16x unrolling (*i.e.* buffering and processing 16 nonzeros for each iteration). Some of the matrices achieve their best with smaller lu due to their low μ that makes higher order buffering unnecessary (*e.g.* Epidemiology has $\mu = 3.99$ so it achieves its best with $lu = 4$). Over regular matrices, nonzero unrolling can generally provide a performance speedup (calculated using harmonic mean) of 1.98x for single-precision calculation and 1.60x for double-precision calculation. The smaller speedup in double-precision is associated with a larger memory footprint that intrinsically provides a more optimized baseline and, hence, less room for optimization. Last, we also notice that the benchmark Dense obtains relatively low performance despite being perfectly regular. This is due to poor GPU hardware utilization. In fact, the $n = 4000$ rows are mapped to only $n_w = 125$ warps, a small portion of the 960 warps that Tesla K40 can manage. Table 8 reports the analysis on irregular matrices. As expected, the baseline performance without unrolling is generally poor compared to the regular benchmarks (for which WELL is better suited). On the other hand, there is a substantial performance improvement (approximately 4x for both single and double precision) achieved by choosing $lu = 16$. Indeed, a large unrolling factor provides a more efficient processing for skewed warps (*i.e.* those containing very irregular rows).

Matrix	Performance [GFLOPS]									
	1x	2x	4x	8x	16x	1x	2x	4x	8x	16x
Circuit5M	0.095	0.126	0.243	0.357	0.393	0.093	0.122	0.290	0.350	0.383
Eu-2005	4.590	5.911	9.750	12.400	13.094	4.327	5.380	8.939	9.832	10.641
In-2004	4.080	5.251	8.721	11.112	11.988	3.869	4.855	8.000	8.897	10.223
LP	0.407	0.545	1.038	1.499	1.643	0.399	0.526	1.193	1.426	1.568
Mip1	0.322	0.427	0.817	1.197	1.307	0.314	0.412	0.972	1.172	1.282
Webbase	1.322	1.749	3.195	4.378	4.880	1.282	1.662	3.366	3.930	4.529
Single Precision					Double Precision					

Table 8. Performance results for the nonzero unrolling with WELL on the irregular benchmarks.

There are additional architectural factors that should be taken into consideration while predicting the best unrolling factor. As general guideline for performance, SMX occupancy (*i.e.* the fraction of active threads on a SMX) [39] should be maximized by taking into account constraints such as registers per thread

(or shared memory usage) and by choosing an appropriate block size b . Given an arbitrary GPU kernel, we immediately know the amount of registers per thread r . Moreover, we are able to evaluate how r will limit the number of active blocks and, hence, SMX occupancy. For example, $r = 32$ does not impose any register-related limit. On the other hand, $r = 128$ and $b = 96$ (*i.e.* a multiple of warp size) imposes a limit of 5 active blocks (each one using 12288 registers out of the 65536 available on each SMX) achieving a 0.234 occupancy. By selecting block size $b = 64$, the imposed limit raises to 8 with a 0.250 occupancy. Similarly, $b = 32$ corresponds to 16 active blocks with the same 0.250 occupancy. As we can see, the selection of optimal block size b is a convoluted tradeoff. However, we tackle this problem using a simple but effective exhaustive approach. We test all the warp multiples $b = i \cdot w$ such that $i \in \{1, 2, \dots, 32\}$ and we choose the smallest b that leads to highest SMX occupancy (in general, small blocks provide a better turnover). Referring to the example just presented, we choose $b = 32$ over $b = \{64, 128, 256\}$ despite all the options have a 0.250 occupancy. As an aside, this selection approach has been used for tuning b in all our tests.

Unrolling Factor	Registers r	Block b	SMX Occupancy	Unrolling Factor	CUDA r	CUDA b	SMX Occupancy
1x	15	128	1.000	1x	18	128	1.000
2x	19	128	1.000	2x	22	128	1.000
4x	23	128	1.000	4x	32	128	1.000
8x	32	128	1.000	8x	48	128	0.625
16x	47	64	0.625	16x	79	64	0.375
Float Precision				Double Precision			

Table 9. SMX occupancy of SpMV kernels for WELL.

Table 9 reports registers and SMX occupancy for all the SpMV kernels that we presented so far. As we can see, buffering with large unrolling factors involves more complex code and, hence, a higher number of registers. This highlights the fact that register pressure is the limiting factor for loop unrolling, not just for SMX occupancy. The compiler has the ability to minimize the number of registers needed to implement a kernel. However, some very complex codes may cause register spilling (*i.e.* some registers are moved into local memory). In general, this has a very detrimental effect on performance and should be avoided as much as possible. Hence, it is not uncommon for complex kernel to achieve better performance with smaller loop unrolling factors (*e.g.* $lu = \{2, 4\}$) where register spilling is avoided. So far, we presented how to design the data layout in order to promote efficient SIMD vectorization and how to process nonzeros in order to optimize the memory hierarchy efficiency. In the next sections, we will introduce two techniques targeted for mitigating the bandwidth-limited nature of the SpMV kernel.

5. Exploiting dense substructures with blocking

Dense substructures are inherently present in many sparse matrices such as those derived from partial differential equation models, providing an opportunity to further optimize the performance. Blocking is a technique similar to compression that allows to leverage dense substructure to reduce the memory footprint associated with indexing. The idea is to extract a blocked matrix from the original one and store nonzero blocks (rather than single nonzero entries). Blocking can be applied to WELL as described by Figure 5. Given the sparse matrix $\mathbf{A}$ with a dense substructure, its WELL representation needs 32 elements for *Value[]* and *Column[]*. Moreover, the data structure spans two warps ($w = 4$). Given the blocked matrix $\mathbf{B}$ extracted from $\mathbf{A}$, we can build WELL using $[2 \times 2]$ nonzero blocks rather than single nonzeros. We can immediately notice how this new WELL structure has 8 (block) elements rather than 32. As a result, the *Column[]* structure is more compact, leading to an overall space savings. In other words, each $[2 \times 2]$ block needs to store only a single column index corresponding to subelement $[0, 0]$ (everything else can be directly derived from there). We can also notice that now WELL spans a single warp with 4 lanes that process two blocks each. In other words, now WELL works at block granularity, processing multiple rows simultaneously depending on the blocking factor (2 in the given example).

Fig. 5. Sparse matrix [a], WELL [b], sparse blocked matrix [c] and WELL with blocking [d].

In order to guarantee coalesced memory access at warp level, the memory layout of *Value[]* needs to be redesigned. The overall idea is to transform each 2D block as a 1D array (e.g. $\begin{bmatrix} a & b \\ e & f \end{bmatrix}$ becomes $[a, b, e, f]$) and then store those arrays in a interleaved fashion at warp granularity, repeating the process for each batch of nonzeros blocks. Figure 6 clarifies this memory layout with an example. The interleaving placement is highlighted with different shades of gray. The first coalesced memory access will load $[a][h][l][r]$ (one per lane), which is the first element in each block. The second access will load $[b][i][m][s]$, and so on until the entire first batch of blocks is completely loaded. This is then repeated for all the other block batches until the warp completes the processing to its local (blocked) ELL structure. Blocking can naturally take advantage of buffering. Moreover, blocking can be integrated with nonzero unrolling (which will now happen at block granularity). For example, if we apply 2x unrolling to Figure 5, thread lane 0 will buffer two nonzero blocks (i.e. $[a, b, e, f, c, d, g, 0]$ and $[0, 2]$) plus 4 elements from $\mathbf{x}$ (i.e. $[x_0, x_1, x_3, x_4]$). The resulting dot product computation can be aggressively optimized with loop unrolling.

Fig. 6. Sparse blocked matrix [a] and its interleaved memory layout [b].

The blocking technique applied on WELL may be more or less effective, depending on how well an arbitrary blocking factor $[bn \times bm]$ suits the underlying dense substructure in the sparse matrix $\mathbf{A}$. First, we define a $[0, 1]$ metric called block density $d_{[bn \times bm]}$ as

$$d_{[bn \times bm]} = \frac{\text{nnz}}{bn \cdot bm \cdot \text{nnz}_{[bn \times bm]}} \quad (7)$$

where $\text{nnz}_{[bn \times bm]}$ is the number of nonzero blocks in the blocked matrix $\mathbf{B}_{[bn \times bm]}$ extracted from the original

A. Note that $d_{[bn \times bm]} = 1$ when no padding needs to be added. In the example of Figure 5, $d_{[2 \times 2]} = 0.875$ because we added some zeros to densely fill $\begin{bmatrix} c & d \\ g & 0 \end{bmatrix}$, $\begin{bmatrix} l & m \\ 0 & p \end{bmatrix}$ and $\begin{bmatrix} n & o \\ q & 0 \end{bmatrix}$. The next factor to analyze is the impact of blocking on the WELL data structure. Let $n_{[bn \times bm]}^w = \lceil \frac{n}{w} \rceil$ be the number of warps necessary for the blocked matrix $\mathbf{B}_{[bn \times bm]}$ and $k_{[bn \times bm]}^i$ be the longest blocked row in the local (blocked) ELL structure associated with warp i . Hence, we modify the efficiency formula (6) as follows

$$e_{[bn \times bm]} = \frac{\text{nnz}_{[bn \times bm]}}{\sum_{i=1}^{n_{[bn \times bm]}^w} w \cdot k_{[bn \times bm]}^i} = \frac{\text{nnz}_{[bn \times bm]}}{\text{Offset}[n_{[bn \times bm]}^w]}. \quad (8)$$

Considering again the example in Figure 5, $e_{[2 \times 2]} = 0.75$ because 2 blocks out of 8 have been inserted as zero padding. The last factor to analyze is the effect of blocking on the memory footprint. Given a nonzero block, the amount of data necessary for column indexing will be compressed by a $bn \cdot bm$ factor. Assuming single-precision computation, we define the compression $c_{[bn \times bm]}$ associated with blocking as follows

$$c_{[bn \times bm]} = \frac{8}{4 + \frac{4}{bn \cdot bm}} \leq 2, \quad (9)$$

where the upper bound for compression is 2 (*i.e.* the scenario in which we represent the sparse matrix with an unique dense block). It might be argued that compression $c_{[bn \times bm]}$ is only dependent by the given blocking factor $[bn \times bm]$ whereas density $d_{[bn \times bm]}$ and efficiency $e_{[bn \times bm]}$ also depend by the original sparse matrix **A**. We can repeat the same analysis for double-precision computation obtaining

$$c_{[bn \times bm]} = \frac{12}{8 + \frac{4}{bn \cdot bm}} \leq 1.5, \quad (10)$$

where the upper bound is smaller due to the smaller impact of column index compression (and, hence, of blocking) on the overall memory footprint. Finally, we summarize all previous considerations into a quality metric that provides a rough estimate of which blocking factor $[bn \times bm]$ to choose for optimality

$$q_{[bn \times bm]} = d_{[bn \times bm]} \cdot e_{[bn \times bm]} \cdot c_{[bn \times bm]}. \quad (11)$$

In other words, $q_{[bn \times bm]}$ estimates the effect of zero padding and compression on the memory footprint associated with a particular choice of $[bn \times bm]$. The definitions of density $d_{[bn \times bm]}$, efficiency $e_{[bn \times bm]}$ and compression $c_{[bn \times bm]}$ are generalization of the base case $[1 \times 1]$ with no blocking. From (11), $q_{[1 \times 1]} = d_{[1 \times 1]} \cdot e_{[1 \times 1]} \cdot c_{[1 \times 1]} = 1 \cdot e_{\text{WELL}} \cdot 1 = e_{\text{WELL}}$. By comparison, the quality metric $q_{[bn \times bm]}$ may estimate whether applying blocking is a good idea (*i.e.* presence of a dense substructure) and, eventually, which blocking factor provides a more compact memory footprint (and, likely, good SpMV performance).

5.1. Empirical evaluation

Here we analyze the effectiveness of the blocking technique just presented, with the secondary goal to directly correlate the proposed quality metric to measured performance. We considered all the blocking factors with area $bn \cdot bm \in [1, 9]$ (*e.g.* for area $bn \cdot bm = 4$ we consider $[1 \times 4]$, $[2 \times 2]$ and $[4 \times 1]$). Table 10 reports the SpMV performance on the regular matrices focusing on the baseline $[1 \times 1]$ and the best performing blocking factor. Those results are incremental over those in Section 4, in the sense that the implemented SpMV kernels use both blocking and nonzero unrolling (we report the best-tuned result). As we can visually notice (highlighted and in bold), most of the regular matrices have a dense substructure that can be leveraged with blocking to achieve superior performance. Indeed, we have an average of 1.17x (1.09x) speedup for single-precision (double-precision). Not surprisingly, the quality metric can estimate fairly well when it is convenient to apply blocking. Specifically, the best performing blocking factor always has a higher quality metric. For example, matrix QCD has a $[3 \times 3]$ dense substructure on which blocking improves performance from 47.215 to 76.215 GFLOPS (single-precision). This improvement can be

Matrix	Performance [GFLOPS]			Quality Metric		Performance [GFLOPS]			Quality Metric	
	[1x1]	Best		[1x1]	Best	[1x1]	Best		[1x1]	Best
Circuit	12.007	12.007	[1x1]	0.398	0.398	9.229	9.229	[1x1]	0.398	0.398
Dense	16.163	24.180	[1x9]	0.992	1.780	15.526	22.350	[1x9]	0.992	1.405
Economics	13.333	13.333	[1x1]	0.275	0.275	9.040	9.040	[1x1]	0.275	0.275
Epidemiology	38.945	38.945	[1x1]	0.999	0.999	25.991	25.991	[1x1]	0.999	0.999
FEM/Accelerator	38.271	40.032	[2x1]	0.840	0.861	25.572	25.572	[1x1]	0.840	0.840
FEM/Cantilever	44.253	68.577	[3x3]	0.881	1.435	30.475	37.330	[3x3]	0.881	1.133
FEM/Harbor	31.382	37.155	[1x3]	0.663	0.792	21.989	23.444	[1x2]	0.663	0.705
FEM/Ship	44.159	72.493	[1x6]	0.846	1.450	29.728	40.788	[1x6]	0.846	1.172
FEM/Spheres	48.005	79.597	[3x3]	0.930	1.635	32.555	42.509	[3x3]	0.930	1.290
Ga41As41H72	19.268	20.523	[1x2]	0.364	0.390	12.898	12.898	[1x1]	0.364	0.364
Protein	40.783	59.862	[3x3]	0.796	1.325	27.391	34.863	[1x3]	0.796	1.024
QCD	47.929	76.215	[3x3]	1.000	1.800	32.661	43.447	[3x3]	1.000	1.421
Si41Ge41H72	21.406	22.700	[1x2]	0.410	0.441	14.341	14.341	[1x1]	0.410	0.410
Wind Tunnel	48.713	64.080	[1x3]	0.982	1.326	33.254	37.955	[1x3]	0.982	1.137
Harmonic Mean	26.196	1.17x				18.715	1.09x			
	Single Precision					Double Precision				

Table 10. Incremental performance for the blocking technique on the regular benchmarks.

correlated to the quality metric improvement from 1.000 to 1.800. We analyzed the relationship between the measured performance and the quality metric across all the blocking factors more directly. There is a positive correlation of 0.910 (single-precision) and 0.942 (double-precision). For the sake of completeness, we also report performance tests on irregular matrices (Table 11). Even here, despite comparing with a poor baseline (we have not yet addressed the irregularity issue), we can observe that the blocking technique provides a performance edge that we can integrate with other advanced techniques for SpMV optimization. On the other hand, the skewed distribution of some of the irregular matrices makes the proposed quality metric less reliable, although the correlation with performance still remains 0.796 (single-precision) and 0.885 (double-precision).

Matrix	Performance [GFLOPS]			Quality Metric		Performance [GFLOPS]			Quality Metric	
	[1x1]	Best		[1x1]	Best	[1x1]	Best		[1x1]	Best
Circuit5M	0.393	0.589	[1x9]	0.247	0.214	0.383	0.517	[1x4]	0.247	0.237
Eu-2005	13.094	14.286	[1x2]	0.457	0.438	10.641	10.641	[1x1]	0.457	0.457
In-2004	11.988	15.208	[1x3]	0.367	0.367	10.223	10.870	[1x2]	0.367	0.346
LP	1.643	1.964	[1x2]	0.220	0.231	1.568	1.568	[1x1]	0.220	0.220
Mip1	1.307	2.000	[1x9]	0.698	1.039	1.282	1.909	[1x9]	0.698	0.820
Webbase	4.880	4.880	[1x1]	0.316	0.316	4.529	4.529	[1x1]	0.316	0.316
Harmonic Mean	1.401	1.40x				1.351	1.28x			
	Single Precision					Double Precision				

Table 11. Incremental performance for the blocking technique on the irregular benchmarks.

6. Loseless compression for indexing data

In general, compressing the sparse matrix structure (*i.e* indexing data) is a target for optimization since it reduces the memory footprint. For example, the blocking technique just analyzed substantially reduces the column indices associated with each nonzero block, providing a $[bn \times bm]$ implicit compression ratio. There are additional opportunities for compression within the nonzero pattern. However, any technique that we may want to adopt should have a decompression phase which is as simple as possible in order to be efficiently embedded into the SpMV kernel. Following this principle, we proposed a technique based on delta encoding of column indices [31]. Let us consider the way thread lanes process nonzeros within a

Fig. 7. WELL [a] and WELL with delta-based index compression [b]. In this particular example, the column indices in the first warp are close enough to apply differential encoding and reduce the associated memory footprint.

warp. Whenever the column distance between two consecutive elements is small enough, it is convenient to reconstruct the next column using a delta from the current index. This, in turn, allows to simply store a 8-bit or 16-bit delta rather than a 32-bit absolute index. This compression technique can be tailored to SIMD vectorized execution. Specifically, we propose to apply the differential encoding only when all the delta within the warp (*i.e.* local ELL data structure) are representable as 8-bit or 16-bit integers. This provides a regular memory layout as well as a divergency free execution. Figure 7 shows an example of our compression technique integrated into a sparse matrix format with warp granularity. The nonzero structure associated with the first warp is stored as an absolute base column index (from the first nonzero) followed by a sequence of compressed delta values occupying in the case of the example half the space (of 32-bit integers). The correspondent portion of $Column[]$ is now resized down from dimension 4 to 2.5. However, we may decide to introduce some padding in order to w -align the next warp (*i.e.* moving the second warp from location 10 to 12). This delta-based compression technique needs some additional data structures. First, it is necessary to store $ColumnOffset[]$, an incremental offset to locate any local indexing data in $Column[]$. Second, we need a flag array $Flags[]$ that encodes the compression applied to each warp (*e.g.* $b_1b_0 = 00$ is “no compression”, $b_1b_0 = 01$ is “16-bit encoding” and $b_1b_0 = 10$ is “8-bit encoding”). The second warp in Figure 7 is not compressed despite being eligible. In general, it is not profitable to apply delta encoding to warps with small $k_i = \{1, 2\}$ because they do not improve the local memory footprint after the necessary padding for w -alignment. Finally, this compression technique can be easily integrated with blocking due to the independency of column indexing data. Similarly, it is possible to apply nonzero unrolling by buffering deltas rather than absolute column indices.

We can now revise the quality metric $q_{[bn \times bm]}$ defined in Section 5 in order to incorporate our delta-based compression technique. Given an arbitrary warp $w^{[i]}$, we can define its delta compression as

$$\delta_{[bn \times bm]}^i = \begin{cases} 1 & \text{no delta encoding} \\ \frac{4k_i}{4 + 4w \lceil (k_{[bn \times bm]}^i - 1) / 2w \rceil} \leq 2 & \text{if 16-bit delta encoding} \\ \frac{4k_i}{4 + 4w \lceil (k_{[bn \times bm]}^i - 1) / 4w \rceil} \leq 4 & \text{if 8-bit delta encoding} \end{cases} \quad (12)$$

Here the achieved ratio has a different upper bound, depending on which compression it is possible to apply. This definition takes into account w -alignment. Specifically, the ceiling function at the denominator evaluates the number of aligned memory chunks associated with differential encoding. Note also how the delta compression $dc_{[bn \times bm]}^i$ depends on the blocking factor $[bn \times bm]$. For example, doubling the block dimension bm halves all the column deltas of the associated blocked matrix $\mathbf{B}$. As a result, some warp may

become eligible for compression. The overall delta compression $dc_{[bn \times bm]}$ is defined as follows

$$\delta_{[bn \times bm]} = \frac{\sum_{i=1}^{n_{[bn \times bm]}^w} k_{[bn \times bm]}^i \cdot dc_{[bn \times bm]}^i}{\sum_{i=1}^{n_{[bn \times bm]}^w} k_{[bn \times bm]}^i} = \frac{\text{ColumnOffset}[n_{[bn \times bm]}^w]}{\text{Offset}[n_{[bn \times bm]}^w]}. \quad (13)$$

The weighted average across the warps can be quickly obtained by dividing the size of *Value[]* by the size of *Column[]*. Referring to the example in Figure 7, $dc_{[bn \times bm]} = 24/20 = 1.2$. Last, we update definition (9) and (10) in order to integrate the newly defined delta compression $dc_{[bn \times bm]}$

$$c_{[bn \times bm]} = \frac{8}{4 + \frac{4}{bn \cdot bm \cdot \delta_{[bn \times bm]}}} \leq 2, \quad (14)$$

$$c_{[bn \times bm]} = \frac{12}{8 + \frac{4}{bn \cdot bm \cdot \delta_{[bn \times bm]}}} \leq 1.5. \quad (15)$$

Intuitively, the ability to apply delta compression to a sparse matrix increases $c_{[bn \times bm]}$ and, hence, the quality metric $q_{[bn \times bm]}$.

6.1. Empirical evaluation

We performed some computational experiments with the aim to evaluate the incremental benefit of delta-based index compression over the other optimization techniques. Table 12 reports the best-tuned (in terms of blocking and unrolling factors) SpMV performance with and without delta-compression on the regular benchmark suite. In general, we observed an average of 1.09x (1.05x) speedup factor for single-precision (double-precision) SpMV computation over a fairly optimized baseline. As we can see (highlighted and in bold), almost all the matrices have a nonzero pattern that can be successfully exploited to reduce memory footprint and to improve SpMV performance. The only exception is the matrix Dense. As mentioned, the row-based parallelization implemented by the underlying WELL does not generate a sufficient number of warps to properly exploit the GPU hardware architecture for that matrix. Therefore, we can treat that matrix as an outlier. Moreover, the revised quality metric is now able to capture the memory footprint reduction associated with the compression technique. In other words, a higher value of $q_{[bn \times bm]}$ correlates with superior performance.

Table 13 reports the result for the other set of matrices. Observing the negligible increments of the quality metric, we can deduce that there is no real opportunity to leverage the irregular nonzero patterns to improve the memory footprint. As a consequence, there is no substantial effect on the average SpMV performance. On the other hand, we can observe that the additional complexity of integrating the compression technique into a SpMV kernel does not introduce any visible performance overhead. This provides a supporting evidence to always apply the proposed optimization, without the need of introducing an additional variable to the space of tuning parameters.

In the last sections, we presented two complementary optimization techniques explicitly targeted to reducing memory footprint. We could observe tangible performance improvements, especially for matrices with regular structure. We now switch our focus on coping with matrix irregularity.

7. Improving Adaptive ELL

The ability to adapt the nonzero workload of each threads (as opposed to row-based parallelization used so far) provides an effective way to cope with matrix irregularity. We proposed AdELL [25], a warp-grained

Matrix	[GFLOPS]		Quality Metric		[GFLOPS]		Quality Metric	
	Delta Compression				Delta Compression			
	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Circuit	12.007	12.326	0.398	0.419	9.229	9.445	0.398	0.412
Dense	24.180	23.584	1.780	1.924	22.350	21.669	1.405	1.463
Economics	13.333	16.089	0.275	0.356	9.040	10.235	0.275	0.324
Epidemiology	38.945	39.361	0.999	1.152	25.991	26.060	0.999	1.096
FEM/Accelerator	40.032	42.114	0.861	0.972	25.572	27.789	0.840	0.924
FEM/Cantilever	68.577	71.351	1.435	1.510	37.330	38.978	1.133	1.187
FEM/Harbor	37.155	41.839	0.792	0.958	23.444	25.238	0.705	0.791
FEM/Ship	72.493	74.980	1.450	1.551	40.788	41.088	1.172	1.214
FEM/Spheres	79.597	81.399	1.635	1.717	42.509	43.934	1.290	1.274
Ga41As41H72	20.523	24.140	0.390	0.484	12.898	15.060	0.364	0.436
Protein	59.862	62.970	1.325	1.368	34.863	37.111	1.024	1.104
QCD	76.215	78.990	1.800	1.887	43.447	43.910	1.421	1.456
Si41Ge41H72	22.700	26.485	0.441	0.528	14.341	16.774	0.410	0.491
Wind Tunnel	64.080	73.236	1.326	1.588	37.955	41.143	1.137	1.254
Harmonic Mean	30.675	1.09x			20.465	1.04x		
	Single Precision				Double Precision			

Table 12. Incremental performance for delta compression on the regular benchmarks.

Matrix	[GFLOPS]		Quality Metric		[GFLOPS]		Quality Metric	
	Delta Compression				Delta Compression			
	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Circuit5M	0.589	0.579	0.214	0.228	0.517	0.506	0.237	0.238
Eu-2005	14.286	14.302	0.438	0.458	10.641	10.856	0.457	0.476
In-2004	15.208	15.647	0.367	0.409	10.870	11.261	0.346	0.377
LP	1.964	1.963	0.231	0.231	1.568	1.783	0.208	0.208
Mip1	2.000	1.916	1.039	1.091	1.909	1.787	0.820	0.841
Webbase	4.880	4.925	0.316	0.326	4.529	4.553	0.316	0.323
Harmonic Mean	1.969	0.99x			1.713	1.00x		
	Single Precision				Double Precision			

Table 13. Incremental performance for delta compression on the irregular benchmarks.

sparse matrix format implementing the adaptivity technique. AdELL is based on the idea of allocating an adaptive number of warp lanes l_r to each row r , depending on the row's nonzeros nnz_r . In other words, now each warp $w^{[l]}$ provides a more flexible computation strategy where an arbitrary number $\{1, 2, \dots, w\}$ of rows can be used to keep all the processing lanes occupied (*i.e.* $\sum_{r \in w^{[l]}} l_r = w$). In general, thread lanes need to collaborate to aggregate the dot product of the corresponded row r . This can be efficiently implemented as a warp-level segmented reduction by building upon intrinsics (*e.g.* warp shuffle instruction). A careful lane assignment can greatly improve the overall memory footprint efficiency (which is still calculated as e^{WELL}). Specifically, each warp workload k_i is now determined by value of the largest $\lceil \frac{nnz_r}{l_r} \rceil$ among the rows (rather than the absolute nnz_r). Irregular rows (*i.e.* with a large number of nonzeros) can now be processed by up to w lanes or even across multiple warps by the means of atomic operations, providing an effective strategy to implement load balancing.

The example in Figure 8 illustrates the AdELL data structure and its efficiency benefit over WELL. A reduction map array $Map[]$ is used to specify each local segmented reduction pattern (*e.g.* warp $w^{[0]}$ has $Map[0]=1001$ since $l_0 = 3$ and $l_1 = 1$). Moreover, an additional array $RowOffset[]$ is used to store the first row associated with each warp. This is necessary because the implicit assignment of w rows for each warp does not hold anymore. AdELL has the same fundamental warp-grained structure of WELL so all the presented optimization techniques can be directly applied. Considering blocking, the adaptivity technique will just focus on allocating lanes to rows from the blocked matrix. Similarly, delta-based compression can

Fig. 8. An example of the nonzeros distribution unfavorable to row-based parallelization. On the one hand, WELL [a] needs two warps to process 7 rows and introduces a substantial amount of zero padding. On the other hand, AdELL can assign an arbitrary number of lanes to each row (e.g. row 0 has $l_0 = 3$ lanes) obtaining an evident efficiency improvement (e^{WELL} increases from $18/36 = 0.5$ to $18/20 = 0.9$). Indeed, AdELL makes effective use of a total of 12 lanes over 3 warps

be applied on *Column[]* and each lane can unroll the processing of its nonzero blocks. Figure 8 does not show any example of rows distributed across multiple warp. In that case, we may use an addition flag (e.g. b_2 in *Flags[]*) to signal that the final step of segmented reduction is an atomic operation on result vector $\mathbf{y}$.

AdELL is a data structure that supports adaptivity, a technique can be leveraged to optimize the SpMV performance for irregular matrices. However, it is the actual warp assignment that achieves memory footprint efficiency and load-balancing across the GPU hardware architecture. We can define a simplified version of this WARP-BALANCING PROBLEM [25]. Given the set of rows from (blocked) matrix $\mathbf{A}$, we perform the following (constrained) tasks :

Lane Assignment : Assign an adaptive number of lanes $l_r \in \{1, 2, \dots, w\}$ to each row r

Warp Partitioning : Compose each warp $w^{[i]}$ such that $\sum_{r \in w^{[i]}} l_r = w$

The multi-objective function maximizes the memory footprint efficiency e^{WELL} while minimizing the global workload unbalance $u = \frac{\max k_i - k_{median}}{k_{median}}$ across the warps. Here, we do not combine the two objectives into one (e.g. linear combination) since it is more appropriate to generalize rather than focus on a specific formulation. The optimization problem is then to find *a lane assignment and a partition that optimize the given multi-objective function*. Note, the problem just stated is in the class of NP-hard load-balancing problems [42]. We focus on a variation where rows can be distributed across multiple warps and where the original row ordering should be maintained. Here we propose a revised heuristic to efficiently solve the WARP-BALANCING PROBLEM and, consequently, improve the SpMV performance. This $O(n)$ -runtime approach is designed to be faster than the original self-balancing heuristic used by AdELL [25], with the aim to reduce the cost of preprocessing. Our revised heuristic, in turns, achieves both efficient lane assignment (with Algorithm 1) and warp partitioning (with Heuristic 2).

Let first describe a lane assignment approach to compose candidate warps with minimal k_i and, thus, with maximal e^{ELL} . Given a set of $r \leq w$ consecutive rows $\{1, \dots, r\}$, we solve the subproblem of assigning warp lanes to the rows so that the resulting warp $w^{[i]}$ has minimal k_i and, thus, maximal e_i^{ELL} . The greedy approach presented by Algorithm 1 is incremental. In other words, we use the optimal warp lane assignment for $\{1, \dots, r-1\}$ in order to construct the one for $\{1, 2, \dots, r-1, r\}$. Given an optimal warp $w^{[i]}$ with minimal k_i , we calculate the lanes l_r potentially available for row r (lines 1 to 5). l_r may be sufficient to process nnz_r nonzeros. Otherwise, we have to increase k_i by removing one of the lanes assigned to the rows (lines 9 to 10). If no removal is possible (i.e. all rows have a single lane), we update k_i only considering how to accommodate nnz_r nonzeros over l_r lanes (line 12). Finally, we can add the new row r (line 16).

We can prove the optimality of this approach by induction on the rows incrementally added to the warp. The base case is an empty warp $w^{[i]}$ with $k_i = 0$. In that case, k_i is updated to the minimal $\lceil \frac{\text{nnz}_r}{w} \rceil$ to

Algorithm 1 Adaptive Warp Expansion

Input: Adaptive warp $w^{[i]} = \{1, 2, \dots, r-1\}$ with minimal k_i and containing at most $w-1$ rows
Input: Row r to add**Output:** Adaptive warp $w^{[i]} = \{1, 2, \dots, r-1, r\}$ with minimal k_i

```

1:  $l_r \leftarrow w$ 
2: for each row  $j$  in  $w^{[i]}$  do
3:    $l_j \leftarrow \left\lceil \frac{\text{nnz}_j}{k_i} \right\rceil$ 
4:    $l_r \leftarrow l_r - t_j$ 
5: end for
6: while  $\text{nnz}_r > l_r \cdot k_i$  do
7:    $C \leftarrow \{ \text{row } j \text{ with } t_j > 1 \}$ 
8:   Search row  $j \in C$  with  $\min \left\lceil \frac{\text{nnz}_j}{l_{j-1}} \right\rceil$ 
9:    $k_i \leftarrow \min \left( \left\lceil \frac{\text{nnz}_j}{l_{j-1}} \right\rceil, \left\lceil \frac{\text{nnz}_r}{l_r} \right\rceil \right)$ 
10:   Repeat 1 to 5
11: end while
12: Add  $r$  to  $w^{[i]}$ 

```

accommodate nnz_r nonzeros. The induction step can be proved by focusing on the lane assignment. When $l_r \cdot k_i^{r-1}$ is sufficient to process nnz_r , we just keep $k_i' = k_i^{r-1}$. Otherwise, we need to increase k_i^{r-1} either to make the current l_r lanes “longer” (*i.e.* capable of processing more nonzeros) or to move the assignment of one or more lanes from row $j \in \{1, 2, \dots, r-1\}$ to row r (*i.e.* increasing l_r). Note, the assignment between each row j and its l_j lanes follows the modulo arithmetic. Whenever we increase k_i by one, each row j may have either $l_j = \left\lceil \frac{\text{nnz}_j}{k_i+1} \right\rceil = \left\lceil \frac{\text{nnz}_j}{k_i} \right\rceil$ (same assignment) or $l_j = \left\lceil \frac{\text{nnz}_j}{k_i+1} \right\rceil < \left\lceil \frac{\text{nnz}_j}{k_i} \right\rceil$ (reduced number of lanes). On the other hand, the quantity $l_r \cdot k_i$ is monotonically increasing (recall that $l_r = w - \sum_{j=1}^{r-1} l_j$). Given k_i^{r-1} , we can find the optimal k_i^* by unit increments over k_i^{r-1} until $k_i \cdot l_r \geq \text{nnz}_r$. It might be argued that Algorithm 1 goes from the optimal k_i^{r-1} for rows $\{1, 2, \dots, r-1\}$ to k_i^{r-1} for rows $\{1, 2, \dots, r-1, r\}$ with steps that are equivalent to multiple unit increments done at once. Whenever we search for a row j with minimum $k_i' = \left\lceil \frac{\text{nnz}_j}{l_{j-1}} \right\rceil$ (line 8), we aim to move an additional lane from row j to row r . Due to the modulo arithmetic, all the intermediate unit increments between k_i and k_i' do not increase l_r . In other words, only the last step can modify the overall lane assignment and lead to the optimal k_i^* . This reasoning also applies for the case in which it is sufficient expand k_i without increasing l_r (line 9), proving the optimality of the presented greedy approach.

The warp-balancing heuristic itself is described by Heuristic 2. As we can see, we have two additional parameters $k_{\min}$ and $k_{\max}$ that are available to tune the way AdELL is composed. Specifically, $k_{\min}$ gives a suggestion to favor warps with $k_i \geq k_{\min}$. This is motivated by the empirical fact that warps below $k_{\min} = 4$ do not optimize the execution efficiency on the underlying GPU architecture. On the other hand, $k_{\max}$ imposes an upper bound on k_i that can be used to distribute rows across multiple warps, providing a tuning factor for load-balancing. The proposed heuristic incrementally processes the rows in matrix (A) . At each step, it generates a set of w optimal candidate warps using Algorithm 1 (line 3). Then, it greedily selects the best $w^{[i]}$ in terms of efficiency such that $k_{\min} \leq k_i \leq k_{\max}$ (line 4 to 11). If this latter condition does not hold for any candidate, we also consider warps satisfying $k_i > k_{\max}$. Whenever this happen, the upper limit $k_{\max}$ is subsequently enforced by splitting $w^{[i]}$ on multiple atomic warps with $\left\lceil \frac{k_i}{k_{\max}} \right\rceil$ workload (lines 12 to 16). The last part of the heuristic (lines 20 to 22) takes care of the case in which the number of warps n_w is too low to fully utilize the GPU hardware (*e.g.* matrix Dense with WELL). The proposed heuristic clearly has $O(n)$ runtime. In fact, the outer loop over the rows performs constant work (incremental construction of w candidate warps) at each iteration.

7.1. Empirical evaluation

The main target of adaptivity is the irregular benchmark suite. As done for the other optimization techniques, we performed some computational tests with the aim to evaluate the incremental benefit in terms of SpMV performance. We composed the AdELL data structure considering all the following upper limits:

Heuristic 2 Warp-balancing heuristic

Input: Sparse (blocked) matrix **A****Input:** Nonzeros per lane k_{min} (default 4)**Input:** Nonzeros per lane k_{max} (default ∞)**Output:** AdELL data structure

```

1:  $r \leftarrow 0$ 
2: while  $r < n$  do
3:    $C \leftarrow \{ \text{candidate warps from (blocked) rows } \{r, \dots, r + w - 1\} \text{ using Adaptive Warp Expansion} \}$ 
4:    $C_k \leftarrow \{ \text{Only } w^{[i]} \in C \text{ with } k_{min} \leq k_i \leq k_{max} \}$ 
5:   if  $C_k = \emptyset$  then
6:      $C_k \leftarrow \{ \text{Only } w^{[i]} \in C \text{ with } k_i > k_{max} \}$ 
7:     if  $C_k = \emptyset$  then
8:        $C_k \leftarrow C$ 
9:     end if
10:  end if
11:  Search warp  $w^{[i]} \in C_k$  with highest efficiency  $e_i^{ELL}$ 
12:  if  $k_i \leq k_{max}$  then
13:     $W \leftarrow \{ w^{[i]} \}$ 
14:  else
15:     $W \leftarrow \{ \text{Split } w^{[i]} \text{ into } \lceil \frac{k_i}{k_{max}} \rceil \text{ atomic warps} \}$ 
16:  end if
17:  Append  $W$  to  $AdELL$ 
18:   $r \leftarrow r + i$ 
19: end while
20: while  $n_w < occupancy$  do
21:   Split each  $w^{[i]} \in AdELL$  into  $\lceil \frac{k_i}{k_{median}} \rceil$  atomic warps
22: end while

```

$k_{max} = \{4, 8, 16, 32, 64, 128, \infty\}$ (where ∞ corresponds to not providing any k_{max}). Table 14 reports the best-tuned (in terms of blocking, unrolling factors, upper limit k_{max}) SpMV performance with and without adaptivity on the set of irregular matrices. From now on, we will use the term AdELL+ to refer to AdELL with all the optimizations active. Not surprisingly, addressing the irregular matrix structure brings a huge performance improvement on all the benchmark suite (highlighted and in bold). We observe an astonishing 19.93x (14.03x) speedup for single-precision (double-precision) SpMV computation. The simultaneous increment of quality metric $q_{[bn \times bm, k_{max}]}$ (notice that we added k_{max} to the notation) can give us a hint of the effect of adaptivity. The warps selected by the warp-balancing heuristic have by construction a good efficiency e^{ELL} that contributes to improving the overall memory footprint. On the other hand, the imposed upper limit k_{max} is fundamental for load-balancing. Let us take matrix Circuit5M as example. The best-tuned result is obtained when $k_{max} = 16$. This upper limit generates an AdELL+ data structure with 25233 atomic warps out of 216974. This corresponds to a nearly perfect load-balanced execution where heavyweight rows (accounting for 13% of the total nonzeros) are broken down into small pieces of computation. Adaptivity is also useful for regular matrices as shown by Table 14. We observed another substantial 1.57x (1.52x)

Matrix	[GFLOPS]		Quality Metric		[GFLOPS]		Quality Metric	
	Adaptivity				Adaptivity			
	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Circuit5M	0.589	35.870	0.214	0.952	0.517	24.669	0.237	0.985
Eu-2005	14.286	46.140	0.438	0.978	10.641	28.493	0.457	0.951
In-2004	15.208	48.664	0.367	1.095	10.870	28.677	0.346	0.936
LP	1.964	27.059	0.231	1.076	1.568	17.093	0.208	0.967
Mip1	2.000	76.219	1.039	1.560	1.909	44.675	0.820	1.266
Webbase	4.880	28.448	0.316	0.753	4.529	17.267	0.316	0.805
Harmonic Mean	1.938	19.93x			1.716	14.03x		
	Single Precision				Double Precision			

Table 14. Incremental performance for the adaptivity technique on the irregular benchmarks.

Matrix	[GFLOPS]		Quality Metric		[GFLOPS]		Quality Metric	
	Adaptivity				Adaptivity			
	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Circuit	12.326	26.554	0.419	0.864	9.445	18.811	0.412	0.855
Dense	23.584	91.595	1.924	1.938	21.669	50.641	1.463	1.475
Economics	16.089	26.899	0.356	0.934	10.235	19.746	0.324	0.888
Epidemiology	39.361	34.082	1.152	0.891	26.060	24.356	1.096	1.095
FEM/Accelerator	42.114	45.015	0.972	1.090	27.789	27.629	0.924	0.950
FEM/Cantilever	71.351	80.569	1.510	1.748	38.978	44.383	1.187	1.336
FEM/Harbor	41.839	54.889	0.958	1.301	25.238	34.045	0.791	1.092
FEM/Ship	74.980	84.339	1.551	1.775	41.088	47.410	1.214	1.379
FEM/Spheres	81.399	81.361	1.717	1.758	43.934	45.715	1.274	1.357
Ga41As41H72	24.140	56.939	0.484	1.259	15.060	36.630	0.436	1.142
Protein	62.970	77.188	1.368	1.728	37.111	44.554	1.104	1.338
QCD	78.990	77.372	1.887	1.887	43.910	42.342	1.456	1.376
Si41Ge41H72	26.485	56.451	0.528	1.259	16.774	37.165	0.491	1.142
Wind Tunnel	73.236	73.993	1.588	1.603	41.143	42.981	1.254	1.267
Harmonic Mean	33.353	1.57x			21.235	1.52x		
	Single Precision				Double Precision			

Table 15. Incremental performance for the adaptivity technique on the regular benchmarks.

speedup for single-precision (double-precision) SpMV computation. As we can see (highlighted and in bold), most of the regular matrices can benefit from an improved memory footprint efficiency (which is measured as an increase in the quality metric $q_{[bn \times bm, k_{max}]}$). A notable example is the matrix Circuit where the performance improves by more than a factor of 2 over its baseline without adaptivity. The best absolute performance is associated with the matrix Dense. Not surprisingly, AdELL+ provides a sufficient number of warps n_w to achieve full utilization. On the other hand, there are examples in which adaptivity does not add any benefit. Consider the special case of the matrix Epidemiology. Its structure is composed of rows with an average of $\mu = 4$ and almost zero variability. As a result, there is virtually no room for improvement and it is advisable to not apply the adaptivity technique in order to not incur any implementation overhead (e.g. segmented reduction) on the performance. Fortunately, this feature can be easily integrated into AdELL+ by adding a flag b_3 into $Flags[]$ to select between the two options.

8. Putting it all together

To recap, we designed AdELL+, an advanced ELL-based sparse matrix format that addresses the performance bottlenecks of the SpMV kernel. We tailored the underlying data structure on warp-granularity in order to suit the vectorized GPU hardware architecture. We introduced nonzero unrolling to optimize the memory hierarchy utilization. We mitigated the bandwidth-limited nature of the SpMV kernel by reducing the matrix memory footprint with blocking and indexing compression. Finally, we coped with matrix irregularity using adaptivity and our warp-balancing heuristic. We still have not analyzed and addressed the additional complexity associated with preprocessing. Indeed, selecting the best-tuned parameters may represent a practical issue when we use AdELL+ for real-world application. Therefore, we dedicate the next section to efficient (and effective) parameter auto-tuning.

9. Online auto-tuning

Parameter tuning is a key step to achieving the best possible performance from any advanced sparse matrix format. The overhead of evaluating points in the optimization parameter search space is determined by preprocessing and by actual kernel execution on the GPU hardware. In general, it is desirable to keep this one-time cost as small as possible in order for it to be amortizable over a reasonable number of SpMV iterations. This, in turn, means avoiding an exhaustive search by pruning the parameter space with some

Fig. 9. Online auto-tuning CPU/GPU timeline.

clever strategy. Here we propose an online auto-tuning approach for AdELL+ that uses the quality metric $q_{[bn \times bm, k_{max}]}$ to drive the process. The main novelty of this approach is the idea to tune while doing useful SpMV computation (as opposed to an initial offline tuning phase), completely hiding all the preprocessing overhead associated with composing AdELL+. The proposed auto-tuning schema is depicted in Figure 9. We have a timeline with CPU (where each tuning configuration is composed) and GPU (where it can be actually benchmarked). After loading the sparse matrix from the disk, the CPU processes the sparse format data structure and transfers it to the GPU. From that moment, the SpMV execution (e.g. a sparse solver) can begin. We can evaluate the performance by simply timing the SpMV kernel within the application context. In the mean time, the CPU is available to compose the second tuning configuration. Let us assume that the GPU has enough memory to contain two differently-tuned sparse matrices. Let us also assume that computing and data transfer can overlap (as in Kepler architecture [36]). The GPU can keep executing the original SpMV kernel until the entire second configuration has been transferred. Then, it can switch the computation to this latter for few iterations, evaluating the performance. Depending on which configuration is faster, we will keep one and discard the other. This process can be repeated as many times as needed on multiple configurations, providing an incremental online improvement towards the best-tuned configuration.

The proposed auto-tuning schema can be concretely applied to AdELL+ by introducing few additional considerations. We first analyze the preprocessing timing data reported in Table 16. The experimental computing platform on which those data have been generated was a dual-socket system with two 8-cores INTEL Xeon E5-2650@2GHz and 64GB@1.6GHz DDR3 memory. The operating system was 64-bit CentOS 6.6 with kernel 2.6.32-504.16.2. The compiler used was gcc 4.4.7. All the timings have been obtained as average over 10 runs. The first column of the Table 16 reports the time necessary to load the original Matrix Market file [33] from disk. The following columns are instead the preprocessing times to compose and tune CSR, AdELL+ and BCCOO [28]. Note that the measurements for this latter format are available only in single-precision due to the lack of double-precision implementation. Note also that using CSR we can draw a direct comparison with CSR+ [29] and CSR-Adaptive [30]. As we can see, the typical time to compose the AdELL+ data structure from an intermediate representation is 2.84x (2.57x for double-precision) higher than the CSR baseline. Indeed, the design choices associated with the warp-balancing heuristic have led to an overhead (*i.e.* first tuning configuration) that can be reasonably amortized. Second, we define the parameter space including all the $[bn \times bm]$ block factors used in Section 5 and all the k_{max} factors used in Section 7 (plus $k_{max} = 0$ to indicate no adaptivity applied). The nonzero unrolling technique is implemented by the means of differently-coded SpMV kernels applied on the same data structure. We empirically determined that 5 SpMV executions are enough to get a performance evaluation for each unrolling factor lu . Hence, 25 SpMV iterations (within the context of our online auto-tuning approach) are sufficient to find the best lu associated with any new tuning configuration just transferred to GPU. The optimal CUDA block size is instead selected using the strategy outlined in Section 4. The goal is now to use the quality metric $q_{[bn \times bm, k_{max}]}$ to prune the search space. We have already described the strong correlation between performance and quality metric. Hence, we may decide to evaluate $q_{[bn \times bm, k_{max}]}$ for all the search space and pick the top-ranked configurations to be benchmarked on the GPU. Unfortunately, evaluating $q_{[bn \times bm, k_{max}]}$ cannot be done without explicitly building the entire AdELL+ data structure. Note, $q_{[bn \times bm]}$ can approximate $q_{[bn \times bm, k_{max}]}$ due its ability to identify the most suitable blocking factors for a sparse matrix. In addition, $q_{[bn \times bm]}$ can be efficiently calculated without the need of composing AdELL+ (*i.e.* only its accessory data

Matrix	Load	Time [s]						
		CSR	AdELL+		BCCOO	CSR	AdELL+	
		Compose	Autotune	Autotune	Autotune	Compose	Autotune	Autotune
Circuit	2.624	0.071	0.410	0.278	2.098	0.082	0.395	0.328
Dense	7.290	0.455	0.406	0.752	3.914	0.420	0.432	0.637
Economics	3.683	0.098	0.576	0.343	2.397	0.113	0.558	0.222
Epidemiology	5.345	0.104	0.745	0.314	3.448	0.108	0.713	0.316
FEM/Accelerator	4.250	0.199	0.615	0.410	2.505	0.229	0.740	0.419
FEM/Cantilever	6.652	0.209	0.216	0.530	2.335	0.317	0.441	0.531
FEM/Harbor	4.053	0.192	0.465	0.353	1.582	0.261	0.690	0.355
FEM/Ship	11.784	0.441	0.534	0.912	3.483	0.667	0.528	0.966
FEM/Spheres	9.572	0.353	0.278	0.749	3.084	0.476	0.279	0.769
Ga41As41H72	29.918	1.501	4.452	2.675	14.379	1.699	4.436	2.712
Protein	6.929	0.276	0.304	0.571	2.099	0.319	0.548	0.589
QCD	3.107	0.173	0.086	0.278	1.438	0.218	0.111	0.300
Si41Ge41H72	24.747	1.257	3.494	2.306	11.479	1.411	3.473	2.211
Wind Tunnel	18.538	0.522	1.099	1.307	4.750	0.605	1.130	1.228
Circuit5M	171.222	4.527	16.442	9.831	53.024	5.315	16.423	9.207
Eu-2005	32.436	1.635	6.008	2.408	15.946	1.816	5.923	2.226
In-2004	28.160	1.204	5.304	1.998	12.305	1.363	5.191	2.284
LP	21.431	2.026	2.253	2.694	51.349	2.033	3.077	2.779
Mip1	11.436	0.719	0.787	1.221	4.065	0.851	0.992	1.594
Webbase	8.035	0.242	1.607	0.484	4.800	0.257	1.547	0.554
Arithmetic Mean		0.810	2.84x	1.658	6.04x	0.928	2.57x	1.969
		Single Precision				Double Precision		

Table 16. Preprocessing time for CSR and AdELL+.

structure). Table 16 reports the time necessary to autotune $q_{[bn \times bm]}$ for all the considered blocking factors using an OpenMP [43] implementation (indeed, each $q_{[bn \times bm]}$ can be calculated independently). Roughly, this is about 6x faster than tuning BCCOO. We conjecture that $q_{[bn \times bm]}$ can be also calculated while parsing the sparse matrix from disk, completely hiding any additional cost (the loading times, as reported in Table 16, are large enough to implement this strategy). Once we pick a subset of top-ranked blocking factors, we can iteratively benchmark all the combinations with k_{max} . Alternatively, we can exploit the conceptual independency between blocking and adaptivity to reduce the search space, applying the line search strategy outlined in Table 17. First, we optimize k_{max} by testing all the feasible possibilities associated with the best blocking factor. In the given example, $[2 \times 1]$ is the top-ranked factor so we explore its $[2 \times 1, k_{max}]$ combinations with k_{max} from ∞ (no bound) to 64 (we omit $k_{max} = 128$ because FEM/Harbor does not have any blocked row with more than 64 nonzero blocks). Second, we optimize $[bn \times bm, 16]$ by exploring the other top-ranked blocking factors (*i.e.* $[1 \times 2]$, $[3 \times 1]$, $[2 \times 2]$, $[1 \times 3]$). As a result, we identify $[2 \times 1, 16]$, the best-tuned configuration with 54.889 GFLOPS.

k_{max}	Performance [GFLOPS]				
	[2x1]	[1x2]	[3x1]	[2x2]	[1x3]
∞	49.715	46.239	46.012	42.318	43.550
0	36.251	33.138	40.004	35.676	40.004
4	45.780	40.595	44.495	45.864	41.031
8	52.900	47.350	51.961	51.725	44.043
16	54.889	48.218	52.268	50.319	43.411
32	52.810	46.245	49.943	47.428	43.631
64	49.771	46.268	46.029	45.701	43.482
128	49.677	46.209	45.971	42.436	43.633
Single Precision					

Table 17. FEM/Harbor parameter tuning with line search.

Fig. 10. AdELL+ single-precision performance on the regular benchmarks.

10. Comparison with the state-of-the-art

In this section, we provide a performance-focused comparison of auto-tuned AdELL+ with other advanced sparse matrix formats. Specifically, we considered BCCOO [28] (which is arguably the state-of-the-art in SpMV optimization), CSR+ [29] and CSR-Adaptive [30]. Regarding the first two sparse formats, we used the code made available by their respective authors. Regarding CSR-Adaptive, we used the implementation contained in the linear algebra library ViennaCL [44]. In addition, we compared AdELL+ with our previous work AdELL [25] (*i.e.* turning off blocking, index compression and upper limit k_{max} for the warp-balancing heuristic). Our experimental results on Tesla K40 for single-precision computation are illustrated in Figure 10 and 11. As we can observe, AdELL+ consistently achieves comparable or better performance than the state-of-the-art. On the regular benchmark suite, AdELL+ has a 1.22x speedup over BCCOO, a 1.52x over CSR+, a 2.95x over CSR-Adaptive and a 1.38x over AdELL. AdELL+ performs better on each individual matrix but Economics, where its performance is comparable with CSR+. Indeed, these results are directly correlated with the SIMD-oriented layout and the nonzero unrolling that leverage the GPU hardware at its best. As an aside note, the ViennaCL implementation of CSR-Adaptive does not perform too well on the NVIDIA Kepler architecture, at least when compared with the results reported by Greathouse and Daga [30] on a comparable AMD GPU. On the irregular benchmark suite, AdELL+ still has an edge over the competition. Specifically, we measured a 1.05x speedup over BCCOO, a 1.16x over CSR+, a 2.28x over CSR-Adaptive and a 1.89x over AdELL. Perhaps the reduced speedup is due to

Fig. 11. AdELL+ single-precision performance on the irregular benchmarks.

Fig. 12. AdELL+ double-precision performance on the regular benchmarks.

the fact that both BCCOO and CSR+ do a good job already in terms of load balancing. Moreover, there is very little structure in terms of dense subblocks and nonzero patterns to leverage for compression. Last, we can notice that BCCOO clearly obtains the best performance for matrix LP. This particular benchmark has only 4284 rows associated with 54743 atomic warps that create a high level of conflict when reducing the result to vector y . Although BCCOO obtained its advantage using a non-portable synchronization-free segmented reduction, we conjecture that the problem can be mitigated by implementing a reduction strategy more similar to the one in CSR+. In Section 2.2, we estimated a 51.88 GFLOPS theoretical single-precision performance peak associated with ELL-based formats. AdELL+ breaks this limit for the regular benchmark suite (53.323 GFLOPS) whereas it achieves a fairly high peak portion (38.628 GFLOPS) for the irregular matrix suite. The former result (even surpassing the theoretical peak) comes from the use of blocking and delta-based compression. Note also that AdELL+ provides a substantial improvement over AdELL for both regular and irregular benchmarks, justifying the additional optimization techniques.

Figure 12 and 13 reports the same computational tests in double-precision. Again, we could not include BCCOO due to the lack of double-precision implementation. As we can see, AdELL+ still consistently outperforms the other advanced sparse matrix formats. On the regular benchmark suite, AdELL+ is the best for all the matrices with an overall 1.31x speedup over CSR+, a 3.08x over CSR-Adaptive and a 1.18x over AdELL. On the irregular benchmarks, AdELL+ still achieves comparable or better performance with a unique exception of the matrix Webbase (where AdELL+ is only slightly behind CSR+). Again, we conjecture that a little variation in the atomic segmented reduction mechanism may bridge the performance

Fig. 13. AdELL+ double-precision performance on the irregular benchmarks.

Fig. 14. AdELL memory bandwidth.

gap. For double-precision, we estimated a 34.58 GFLOPS theoretical performance peak. This time AdELL+ does not break the limit for regular matrices but it comes very close (33.697 GFLOPS). On the other hand, AdELL+ still achieves a fairly high peak portion (24.067 GFLOPS) for irregular matrices.

11. Memory Considerations

We have seen how SpMV is a bandwidth-bound kernel. In this section, we are interested in analyzing the relationship between performance and GPU memory hierarchy. Table 14 reports the effective memory bandwidth of the best-tuned AdELL+ kernel for all the benchmarks. Intuitively, memory bandwidth results close to the 207.5 GB/s peak indicate that the AdELL+ kernel is efficiently using the underlying GPU memory resources. We observe that some of the benchmarks (*e.g.* FEM/Ship) have an effective bandwidth higher than the measured peak. Indeed, this is an artifact of blocking and compression. On the other hand,

	[GFLOPS]		Hit Rate	
	w/o	w/	Texture	L2
Circuit	14.406	19.486	0.699	0.616
Dense	39.711	43.825	0.970	0.983
Economics	113.558	21.407	0.676	0.908
Epidemiology	24.568	25.018	0.597	0.690
FEM/Accelerator	16.551	28.433	0.623	0.839
FEM/Cantilever	32.887	39.113	0.866	0.945
FEM/Harbor	28.131	33.831	0.861	0.917
FEM/Ship	32.853	38.018	0.857	0.909
FEM/Spheres	28.445	37.417	0.892	0.857
Ga41As41H72	27.109	37.620	0.686	0.890
Protein	30.021	35.964	0.870	0.964
QCD	29.038	35.257	0.760	0.872
Si41Ge41H72	126.986	38.123	0.917	0.691
Wind Tunnel	34.535	38.474	0.541	0.932
Circuit5M	6.648	9.081	0.158	0.786
Eu-2005	21.083	29.594	0.820	0.779
In-2004	23.370	28.135	0.702	0.863
LP	11.252	11.093	0.132	0.310
Mip1	23.203	29.385	0.619	0.940
Webbase	15.230	16.948	0.528	0.712
Harmonic Mean	19.803	1.26x		

Table 18. Cache analysis for AdELL+.

some benchmarks (e.g. LP) underperform due to a very irregular access to the dense vector $\mathbf{x}$. In our computational experiments, we observed that texture cache is currently the best approach to mitigate this irregularity. We considered the AdELL+ kernel with no blocking and upper limit (i.e. $[1 \times 1, \infty]$) and we measured the performance with and without read-only memory. In addition, we collected the hit rate for L2 and texture cache. Those results are reported in Table 18. As we can see, texture cache provides an immediate (and effortless) way to improve performance by an average factor of 1.26x. We can also observe how L2 cache achieves an average hit rate of 63.77%. On the other hand, texture cache has a lower hit rate due to its limited size of only 48KB per SMX (as opposed to 1536KB for the coherent L2 cache). Locality may be improved by slicing the matrix into vertical portions as wide as the number of elements that fit into cache memory. However, this technique has the drawback of requiring separate SpMV kernel launches, one for each slice. Another technique known as vector expansion [45] copes with irregularity by expanding the access pattern over a dense array with mz elements. This, in turn, leads to a fully coalesced access during the SpMV kernel. However, we conjecture that composing the expanded vector is as expensive as the irregular memory pattern itself, making the overall approach equivalent to the original SpMV kernel. Indeed, this technique is limited to the case in which the dense vector $\mathbf{x}$ does not change and, hence, can be reused over and over. Last, we are also interested in analyzing the memory footprint associated with the best-tuned AdELL+. Table 19 provides a comparison with CSR taken as baseline for its well-known compact representation. In general, AdELL+ provides a more compact memory footprint measured as a 0.86x ratio for single-precision representation and as a 0.93x ratio for double-precision representation. This reduction is primarily due to the blocking and delta-based index compression techniques, and secondarily to adaptivity (which arranges nonzero more efficiently across the warps). On the irregular matrices, the compression has in general less room for improvement. Hence, AdELL+ cannot always be more compact than CSR. This also provides an additional explanation regarding the double-precision performance on matrix Webbase. As we can see, AdELL+ has a slightly bigger memory footprint than CSR (44.92MB vs 39.35MB) which can be correlated with the slightly inferior SpMV performance.

	Memory Footprint [MB]			
	CSR	AdELL+	CSR	AdELL+
Circuit	7.97	8.63	11.63	13.00
Dense	30.53	15.76	45.78	31.04
Economics	10.50	10.64	15.36	16.65
Epidemiology	18.03	18.16	26.04	22.27
FEM/Accelerator	20.48	18.48	30.50	31.73
FEM/Cantilever	30.81	17.53	46.10	34.37
FEM/Harbor	18.29	14.03	27.35	24.99
FEM/Ship	60.15	33.68	89.95	64.95
FEM/Spheres	46.17	26.13	69.10	50.73
Ga41As41H72	142.08	112.50	212.61	185.81
Protein	33.29	19.24	49.86	37.21
QCD	14.81	7.76	22.13	15.07
Si41Ge41H72	115.24	91.38	172.50	150.86
Wind Tunnel	89.59	55.55	133.98	105.25
Circuit5Ml	475.34	481.20	702.41	696.75
Eu-2005	150.04	151.20	223.42	233.19
In-2004	134.34	118.92	198.88	207.85
LP	86.07	81.10	129.10	134.62
Mip1	79.24	50.72	118.73	93.69
Webbase	27.51	31.84	39.35	44.92
Arithmetic Mean	79.52	0.86x	118.24	0.93x
		Single Precision		Double Precision

Table 19. Memory footprint.

12. Conclusions

Our research work was aimed to improve the state-of-the-art on SpMV optimization on GPUs. After analyzing the performance bottleneck of sparse linear algebra computation, we proposed to combine several optimization techniques into an efficient ELL-based sparse format named AdELL+. We first provided a foundation for AdELL+ based on a SIMD-oriented memory and computational layout (warp granularity) that suit the underlying GPU hardware architecture. We addressed the bandwidth-limited nature of the SpMV kernel with the blocking and delta-based index compression technique. Moreover, we improved the memory hierarchy efficiency with nonzero unrolling. We coped with matrix irregularity using balanced warps composed using a parametrized warp-balancing heuristic. Finally, we introduced a novel online auto-tuning approach that uses a quality metric to predict efficient block factors and progressively explores the parameter space while useful SpMV computation is done (hiding preprocessing overhead). Our experimental results provided substantial evidence that AdELL+ outperforms other state-of-the-art sparse formats proposed in academia (BCCOO) and industry (CSR+ and CSR-Adaptive). Moreover, our auto-tuning approach makes AdELL+ viable for real-world applications.

Sparse linear algebra is a powerful framework used in numerous areas of science and engineering. Optimizing the SpMV kernel is surely critical to solve today's computation problems faster and to determining which problems we will be able to solve tomorrow. This motivated us (as well as many other researchers) to pour a significant effort into implementations based on modern-day parallel computer architectures. We believe that the performance improvement associated with AdELL+ is a significant and important result since it is an improvement over an already outstanding body of previous works in SpMV optimization. We were also able to ground the presented optimization techniques on higher-level theoretical considerations (*i.e.* quality metric). As a result, the ideas presented in this paper are general enough to be valid for future many-core GPU architectures and produce a long-lasting impact. We are also confident that the adoption of AdELL+ will simplify the life of sparse linear algebra library users, relieving from the burden to explore among different sparse matrix formats. Indeed, AdELL+ (along with its auto-tuning approach) consistently provides a one-size-fits-all SpMV performance that is the best (or, at least, a very good one) over heterogeneous application domains.

13. Acknowledgments

This work was supported by NSF grants IIS-106468. We would also like to thank Nvidia for donating the Testa K40 GPU used for our performance tests.

References

- [1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, Tech. rep., EECS Department, University of California, Berkeley (Dec 2006).
URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/>
- [2] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, Society for Industrial and Applied Mathematics, 2003.
- [3] S. Mehrotra, On the Implementation of a Primal-Dual Interior Point Method, SIAM Journal on Optimization 2 (4) (1992) 575–601.
URL <http://epubs.siam.org/doi/abs/10.1137/0802028>
- [4] E. J. Im, K. Yelick, Optimization of Sparse Matrix Kernels for Data Mining, in: SIAM Conference on Data Mining, 2000.
- [5] J. R. Gilbert, S. Reinhardt, V. B. Shah, High-performance Graph Algorithms from Parallel Sparse Matrices, in: International Conference on Applied Parallel Computing (PARA), 2007, pp. 260–269.
URL <http://dl.acm.org/citation.cfm?id=1775059.1775097>
- [6] J. Davis, E. Chung, SpMV: A Memory-Bound Application on the GPU Stuck between a Rock and a Hard Place, Tech. rep., Microsoft (2012).
- [7] S. Huang, S. Xiao, W. Feng, On the Energy Efficiency of Graphics Processing Units for Scientific Computing, in: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 2009, pp. 1–8.
URL <http://dx.doi.org/10.1109/IPDPS.2009.5160980>
- [8] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, in: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC), 2007, pp. 38:1–38:12.
URL <http://doi.acm.org/10.1145/1362622.1362674>

- [9] N. Bell, M. Garland, Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors, in: *Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009, pp. 18:1–18:11.
URL <http://doi.acm.org/10.1145/1654059.1654078>
- [10] NVIDIA, CUDA Sparse Matrix Library cuSPARSE.
URL <http://docs.nvidia.com/cuda/cusparse/>
- [11] F. Vázquez, J. J. Fernández, E. M. Garzón, A New Approach for Sparse Matrix Vector Product on NVIDIA GPUs, *Concurrency and Computation: Practice and Experience* 23 (8) (2011) 815–826.
URL <http://dx.doi.org/10.1002/cpe.1658>
- [12] M. M. Baskaran, R. Bordawekar, Optimizing Sparse Matrix-Vector Multiplication on GPUs, Tech. rep., IBM (2009).
- [13] A. Monakov, A. Lokhtov, A. Avetisyan, Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures, in: *High Performance Embedded Architectures and Compilers*, Vol. 5952, Springer Berlin Heidelberg, 2010, pp. 111–125.
URL http://dx.doi.org/10.1007/978-3-642-11515-8_10
- [14] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. R. Bishop, A Unified Sparse Matrix Data Format for Modern Processors with Wide SIMD units, *CoRR* 2013.
URL <http://arxiv.org/abs/1307.6209>
- [15] J. W. Choi, A. Singh, R. W. Vuduc, Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs, in: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010, pp. 115–126.
URL <http://doi.acm.org/10.1145/1693453.1693471>
- [16] J. C. Pichel, F. F. Rivera, M. Fernández, A. Rodríguez, Optimization of Sparse Matrix-Vector Multiplication Using Reordering Techniques on GPUs, *Microprocessors and Microsystems* 36 (2) (2012) 65–77.
URL <http://dx.doi.org/10.1016/j.micpro.2011.05.005>
- [17] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S.-h. Kuo, R. S. M. Goh, S. J. Turner, W.-F. Wong, Accelerating Sparse Matrix-vector Multiplication on GPUs Using Bit-Representation-Optimized Schemes, in: *Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2013, pp. 26:1–26:12.
URL <http://doi.acm.org/10.1145/2503210.2503234>
- [18] S. Xu, H. X. Lin, W. Xue, Sparse Matrix-Vector Multiplication Optimizations based on Matrix Bandwidth Reduction using NVIDIA CUDA, in: *Proceedings of International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES)*, 2010, pp. 609–614.
- [19] B.-Y. Su, K. Keutzer, clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs, in: *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2012, pp. 353–364.
URL <http://doi.acm.org/10.1145/2304576.2304624>
- [20] F. Vázquez, G. Ortega, J. Fernández, E. Garzón, Improving the Performance of the Sparse Matrix Vector Product with GPUs, in: *Proceedings of IEEE International Conference on Computer and Information Technology (CIT)*, 2010, pp. 1146–1151.
- [21] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, Z. Shao, Optimization of sparse matrix-vector multiplication with variant csr on gpus, in: *Proceedings of IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2011, pp. 165–172.
- [22] Z. Koza, M. Matyka, S. Szkoda, Ł. Mirosław, Compressed Multirow Storage Format for Sparse Matrices on Graphics Processing Units, *SIAM Journal on Scientific Computing* 36 (2) (2014) C219–C239.
URL <http://dx.doi.org/10.1137/120900216>
- [23] T. Oberhuber, A. Suzuki, J. Vacata, New Row-grouped CSR Format for Storing the Sparse Matrices on GPU with Implementation in CUDA, *CoRR* 2010.
URL <http://arxiv.org/abs/1012.2270>
- [24] M. Heller, T. Oberhuber, Adaptive Row-Grouped CSR Format for Storing of Sparse Matrices on GPU, *CoRR* 2012.
URL <http://arxiv.org/abs/1203.5737>
- [25] M. Maggioni, T. Berger-Wolf, AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs, in: *Proceedings of International Conference on Parallel Processing (ICPP)*, 2013, pp. 11–20.
- [26] A. Ashari, N. Sedaghati, J. Eisenlohr, P. Sadayappan, An Efficient Two-dimensional Blocking Strategy for Sparse Matrix-vector Multiplication on GPUs, in: *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2014, pp. 273–282.
URL <http://doi.acm.org/10.1145/2597652.2597678>
- [27] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, P. Sadayappan, Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications, in: *Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2014, pp. 781–792.
URL <http://dx.doi.org/10.1109/SC.2014.69>
- [28] S. Yan, C. Li, Y. Zhang, H. Zhou, yaSpMV: Yet Another SpMV Framework on GPUs, in: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014, pp. 107–118.
URL <http://doi.acm.org/10.1145/2555243.2555255>
- [29] S. Baxter, Modern GPU, Code Intended to Promote New and Productive Ways of Thinking about GPU Computing.
URL <http://nvlabs.github.io/moderngpu/>
- [30] J. L. Greathouse, M. Daga, Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 769–780.
URL <http://dx.doi.org/10.1109/SC.2014.68>
- [31] M. Maggioni, T. Berger-Wolf, CoAdELL: Adaptivity and Compression for Improving Sparse Matrix-Vector Multiplication on GPUs, in: *Proceedings of the IEEE International Parallel Distributed Processing Symposium Workshops (IPDPSW)*, 2014, pp. 933–940.
- [32] T. A. Davis, Y. Hu, The University of Florida Sparse Matrix Collection, *ACM Transaction on Mathematical Software (TOMS)*

- 38 (1) (2011) 1:1–1:25.
 URL <http://doi.acm.org/10.1145/2049662.2049663>
- [33] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, J. J. Dongarra, Matrix Market: A Web Resource for Test Matrix Collections, in: *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, 1997, pp. 125–137.
 URL <http://dl.acm.org/citation.cfm?id=265834.265854>
- [34] D. Langr, P. Tvrdik, Evaluation Criteria for Sparse Matrix Storage Formats, *IEEE Transactions on Parallel and Distributed Systems* PP (99) (2015) 1–14.
- [35] S. Williams, A. Waterman, D. Patterson, Roofline: An Insightful Visual Performance Model for Multicore Architectures, *Communications of the ACM* 52 (4) (2009) 65–76.
 URL <http://doi.acm.org/10.1145/1498765.1498785>
- [36] NVIDIA, NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110.
 URL <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>
- [37] E. Bardsley, A. Donaldson, Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels, in: *NASA Formal Methods*, Vol. 8430, Springer International Publishing, 2014, pp. 230–245.
 URL http://dx.doi.org/10.1007/978-3-319-06200-6_18
- [38] M. Maggioni, T. Berger-Wolf, J. Liang, GPU-Based Steady-State Solution of the Chemical Master Equation, in: *Proceedings of the IEEE International Parallel Distributed Processing Symposium Workshops (IPDPSW)*, 2013, pp. 579–588.
- [39] NVIDIA, CUDA: Compute Unified Device Architecture.
 URL <http://docs.nvidia.com/cuda/#axzz3apxVEBRJ>
- [40] Khronos, OpenCL: The Open Standard for Parallel Programming of Heterogeneous Systems.
 URL <https://www.khronos.org/opencv1/>
- [41] G. Murthy, M. Ravishankar, M. Baskaran, P. Sadayappan, Optimal Loop Unrolling for GPGPU Programs, in: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010, pp. 1–11.
- [42] J. Kleinberg, Y. Rabani, É. Tardos, Fairness in routing and load balancing, *Journal of Computer and System Sciences* 63 (1) (2001) 2–20. doi:<http://dx.doi.org/10.1006/jcss.2001.1752>.
 URL <http://www.sciencedirect.com/science/article/pii/S0022000001917520>
- [43] L. Dagum, R. Menon, OpenMP: An Industry-Standard API for Shared-Memory Programming, *IEEE Computational Science and Engineering* 5 (1) (1998) 46–55.
 URL <http://dx.doi.org/10.1109/99.660313>
- [44] K. Rupp, ViennaCL: A High Level Linear Algebra Library for GPUs and Multi-Core CPUs.
 URL <http://viennacl.sourceforge.net/>
- [45] Y. Deng, B. Wang, S. Mu, Taming irregular EDA applications on GPUs, in: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2009, pp. 539–546.