Efficient sparse matrix-vector multiplication on cache-based GPUs

István Reguly
Faculty of Information Technology
Pázmány Péter Catholic University
Hungary
reguly.istvan@itk.ppke.hu

Mike Giles
Oxford e-Research Centre
University of Oxford
United Kingdom
mike.giles@maths.ox.ac.uk

ABSTRACT

Sparse matrix-vector multiplication is an integral part of many scientific algorithms. Several studies have shown that it is a bandwidth-limited operation on current hardware. On cache-based architectures the main factors that influence performance are spatial locality in accessing the matrix, and temporal locality in re-using the elements of the vector.

This paper discusses efficient implementations of sparse matrix-vector multiplication on NVIDIA's Fermi architecture, the first to introduce conventional L1 caches to GPUs. We focus on the compressed sparse row (CSR) format for developing general purpose code. We present a parametrised algorithm, show the effects of parameter tuning on performance and introduce a method for determining the near-optimal set of parameters that incurs virtually no overhead. On a set of sparse matrices from the University of Florida Sparse Matrix Collection we show an average speed-up of 2.1 times over NVIDIA's CUSPARSE 4.0 library in single precision and 1.4 times in double precision.

Many algorithms require repeated evaluation of sparse matrix-vector products with the same matrix, so we introduce a dynamic run-time auto-tuning system which improves performance by 10-15% in seven iterations.

The CSR format is compared to alternative ELLPACK and HYB formats and the cost of conversion is assessed using CUSPARSE. Sparse matrix-vector multiplication performance is also analysed when solving a finite element problem with the conjugate gradient method. We show how problem-specific knowledge can be used to improve performance by up to a factor of two.

General Terms

Algorithms, Performance, Many-core, GPU

Keywords

sparse matrix-vector multiplication, cache performance, auto-tuning, finite element method, conjugate gradient method

1. INTRODUCTION

Due to the physical limitations to building faster single core microprocessors, the development and use of multi- and many-core architectures has been the focus of attention for the past few years. Besides the increasing number of processor cores on a single chip, new architectures have emerged that support general purpose computing - the most prominent of which are Graphical Processing Units (GPUs). General Purpose computing on Graphical Processors (GPGPU) has become very popular in the high performance computing community; a great number of papers discuss its viability in accelerating applications ranging from molecular dynamics [1] through dense [12] and sparse linear algebra [4] to medical imaging [17].

The evolution trend of high performance computer architectures shows exponential growth in the number of processing cores, however the increase in bandwidth between on-chip and off-chip memory is slower. Hence, these architectures are becoming increasingly bandwidth-limited, while the cost of computations decreases. This bottleneck can be alleviated by exploiting the multi-level memory hierarchy of these architectures to reuse data that has already been moved to the chip. The absolute amount of L1 cache memory per core in the CPU and per streaming multiprocessor (*SM*) on the GPU is similar, however while the CPU runs 1-2 threads per core, the GPU has hundreds of active threads per SM. On previous-generation NVIDIA GPUs, only 16kBytes of explicitly managed shared memory per SM and limited caching of data stored in texture memory was available. With the introduction of NVIDIA's Fermi architecture [15], this was extended to 16kB/48kB shared memory and 48kB/16kB L1 cache per SM along with a 768kB global L2 cache[1].

The sparse matrix-vector (SpMV) multiplication is commonly known to be a bandwidth-limited operation. Due to the small number of non-zeros in the matrix, sparse matrix formats explicitly store row and column index information. Thus, to perform the multiplication between an element of the matrix and an element of the multiplicand vector, and to write the result to the product vector, the row and column index of the non-zero matrix element is required. The main factor that influences performance is data reuse. The efficiency of index storage can be controlled by the choice of storage format. However, the access pattern to the multiplicand vector is highly dependent on the structure of the matrix and can be extremely irregular.

Due to the high demand for accelerating sparse linear algebra operations several papers have presented different storage formats, algorithms and optimisations to improve the performance of the sparse matrix-vector multiplication. Bell and Garland [3] present a comprehensive study of storage formats like the diagonal format (DIA) for matrices where non-zeros are restricted to a small number of diagonals; the ELLPACK format where the number of non-zeros per row is bounded by a number K and shorter rows are padded with

[1]The size and use of L1 cache can be controlled explicitly by the programmer.

zeros thereby enabling aligned memory access; the coordinate format (COO) which stores the row and column index for each non-zero; and the compressed sparse row (CSR) format stores elements row-by-row thus only a pointer to the first element of each row is stored besides column indices and values. For matrices with a varying number of non-zeros per row they propose a hybrid format, that stores up to K nonzeros per row in the ELLPACK format and the rest in COO format. A special format, the packet format, for storing symmetric mesh-based matrices is also introduced. Their results show the best performance when using the ELLPACK and hybrid formats, they achieve a performance of up to 35 GFlops for structured and up to 25 GFlops for unstructured matrices in single precision using previous generation GTX 280 hardware. They show a performance difference of up to 10 times between storage formats. Vázquez et al. [18] improves upon the performance of aligned formats by introducing a new variant of regular ELLPACK.

Baskaran and Bordawekar [2] improve GPU performance using the CSR storage format by exploiting memory coalescing and synchronisation-free parallelism, and show a speedup of 2 to 8 times over the NVIDIA CUDPP library and segmented scan implementations, and achieve up to 15% improvement over NVIDIA's SpMV library [16]. Several papers [6] [10] [14] discuss special matrix formats like blocked compressed sparse row (BCSR) and blocked ELLPACK to decrease the amount of index data stored and thereby increase data reuse and performance. It has been shown in [8] how problem-specific knowledge can be used to design an efficient sparse matrix storage format that can exploit matrix structure to improve data reuse. Because the structure of different sparse matrices may be very different, any algorithm with a set of fixed parameters can perform well for one matrix and worse for an other. Tuning algorithm and storage format parameters are discussed in [6] for special blocked matrices. A more general library is discussed in [9] to accelerate CPU applications that use sparse matrix-vector products; it presents both a tuning algorithm and a heuristic decision method for choosing multiplication algorithms for different matrices. Their results however include in the total performance the conversion from doubles to floats and the movement of data between the host and the GPU. Especially for small matrices this overhead is significant, and for algorithms implemented entirely on the GPU this overhead would not apply.

This paper aims to demonstrate the design procedure of sparse matrix-vector multiplication algorithms on the latest cache-based NVIDIA Fermi GPUs. Our goal is to implement efficient general purpose code that provides adequate performance for any given sparse matrix. We focus our attention on the most commonly used storage format, the compressed sparse row (CSR) format, and show how different properties of matrices and values of algorithm-specific parameters impact performance, with special attention on caching mechanisms. We demonstrate that with the proper parameters our algorithm can closely match the performance of more "GPU-friendly" formats and algorithms presented in [3], but without the expensive conversion from one format to another. Through a specific example of solving finite element problems, we also compare the performance of ELLPACK and CSR storage formats and demonstrate how problem-specific knowledge enables us to implement a fundamentally different sparse-matrix vector multiplication algorithm.

The key contributions of this work are the following:

1. We highlight the very limited size of the Fermi L1 cache relative to the number of threads that are typically used on each streaming multiprocessor. Based on this observation we present a parametrised algorithm for performing sparse matrix-vector multiplication using the CSR storage format.

2. We analyse the effects of different parameters and matrix structures on performance and propose a constant-time method for deciding algorithm parameters for any given matrix.

3. We propose a dynamic run-time parameter tuning algorithm that improves performance to within 2% of the optimal setting of parameters found by exhaustive search.

4. We compare the performance of the CSR and ELLPACK/HYB formats and demonstrate the cost of format conversion.

5. We show the importance of problem-specific knowledge and analyse SpMV performance during the solution of a finite element problem.

The rest of the paper is organised as follows: section 2 briefly describes the GPU architecture and programming methodology, section 3 discusses the different aspects of the problem of performing sparse matrix-vector multiplication on GPUs. Section 4 describes the parameter space of the multiplication algorithm and section 5 demonstrates the effects of these parameters on performance, and introduces a fixed rule to decide upon the parameters of the algorithm for any given matrix in constant time. Section 6 introduces a dynamic run-time parameter tuning algorithm that aims to improve the performance of the multiplication over repeated evaluations with the same matrix. Section 7 compares the CSR format to the ELLPACK/HYB format, analyses the difference in performance and the cost of conversion. Section 8 discusses the performance of SpMV using different storage formats and algorithms when solving a finite element problem. Finally, section 9 summarises our results.

2. GPU ARCHITECTURE AND PROGRAMMING MODEL

The GPU is a massively parallel architecture with a multilevel hierarchy for both memory and computations. Different manufacturers have slightly different architectures; in this paper we focus on NVIDIA GPUs. As a computational unit, the GPU comprises of a set of streaming multiprocessors (*SMs*) each with a number of streaming processors (*SPs*) which execute instructions in a single instruction multiple data (*SIMD*) fashion. Groups of 32 threads called *warps* are executed in lockstep on the SPs using one instruction counter, thus any branch divergence within a warp results in inactive threads and a loss of parallelism. Groups of warps called *blocks* are assigned to SMs and at the same time each SM can hold up to 8 blocks (depending on the amount of resources they require), but no more than 1536 threads (for compute capability 2.x). If there are more thread blocks than can be processed simultaneously by the SMs, then some blocks will start executing only after others have finished executing.

Each thread uses a number of registers, and each block can use an explicitly managed, low-latency, on-chip shared memory that can be accessed only by the threads within that block. GPUs have a large off-chip global memory (0.5-6 GBytes) that is accessed with a high latency (300-500 cycles). Prior to Fermi, only the caching of texture memory and constant memory was enabled, but Fermi introduced a 16kB/48kB L1 cache for each SM, and a global 768kB L2 cache. To provide the highest possible bandwidth to global memory, a very wide bus is used, and this leads to a cache line of size 128 bytes, which is significantly larger than the 32 byte cache lines in Intel CPUs.

The L1 cache size is comparable to the cache on the CPU (32k in an Intel Core i7), however a CPU core executes only a few threads while the GPU executes up to 1536 threads per SM. This results in a very constrained cache size per thread: since a cache line is 128 bytes long, even when using 48k L1 cache, only 384 lines can be stored. If all threads read or write to different cache lines, at most 384 of them can get cache hits when accessing their cache line; the others get a cache miss. Cache hits can greatly improve data reuse and thus performance, but misses cause new cache lines to be loaded from L2 cache or global memory, resulting in high latency and increased traffic to/from the global memory. It is therefore very important for threads in the same block to work on the same cache lines, to have so called "cache spatial locality". This is the key observation behind the improved performance achieved in this paper.

Any problem to be solved on the GPU has to be decomposed into fairly independent tasks because collaboration is very limited. Threads in a block can communicate via on-chip shared memory. Synchronisation during execution is only possible between threads in the same block. For threads that are not in the same thread block, communication is only possible indirectly via expensive operations through global memory, and synchronization is only feasible by starting a new kernel.

3. PROBLEM STATEMENT

Our test hardware, the NVIDIA Tesla C2070, has a theoretical peak performance of 1.0 TFlops in single precision and off-chip memory bandwidth of up to 144 GBytes/second. Thus, to balance computations and memory movement the number of operations for every floating point number loaded from memory would have to be about 30. SpMV being a heavily bandwidth-limited operation, our goal is to minimise the amount of data transferred to/from the global memory, and maximise the bandwidth achieved by our kernels; computation operations are considered to be "free" as long as there are enough threads to hide instruction and memory latency.

Our main focus is to minimise data transfer by maximising data reuse. When performing the sparse matrix-vector product, each non-zero element of the matrix is only used once, and their column and row indices are used to address the multiplicand and result vectors. Depending on the storage format these indices can be reused to an extent. More importantly, the access pattern to the multiplicand vector depends on the structure of the matrix and, in the general case, very few assumptions can be made about it. This is why caching mechanisms can greatly improve performance - or in some extreme cases decrease it. Due to the relatively small amount of cache per thread it is essential to understand and optimise for these caching mechanisms.

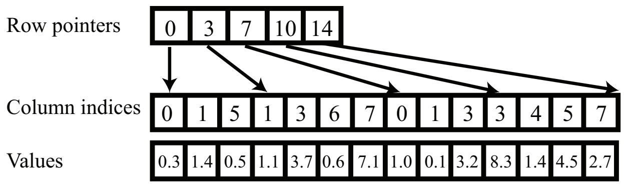

Figure 1: Data layout of the CSR storage format.

In general, sparse matrices can greatly differ in the number of rows and columns, in the average and variance of the length of the rows, and in the distribution of non-zeros within individual rows. Therefore there is no "best" storage format or fixed algorithm for SpMV multiplication. CSR has widespread usage in the scientific community and it is the most commonly used format in CPU based algorithms. It is one of the most efficient formats in terms of storing non-zero values because no padding is required. Blocked matrix formats store small parts of the matrix in a dense format, thereby cutting back on the amount of indexing data required. However, their use is limited to the blocked subclass of sparse matrices, otherwise the fraction of zeros in the dense submatrices would make the storage inefficient. Aligned formats like ELLPACK which are better suited for data access in vector architectures do not require the explicit storage of row indices, because all rows have the same length. This results in having to pad shorter rows with zeros up to the size of the longest row, which means that the memory required for storing both the column indices and non-zero values is higher than absolutely necessary. In case of matrices with a high standard deviation in the length of the rows this would be infeasible; to tackle this problem the hybrid format stores rows up to a certain length in the ELLPACK format, and the rest of the non-zeros are stored along with their row and column indices in the coordinate (COO) format. However, it is difficult to handle matrices in the hybrid format, especially when trying to access structural properties of the matrix. Because of these compromises and the fact that only CSR has widespread support by other libraries we chose the CSR format for our SpMV code.

Many scientific algorithms require repeated execution of sparse matrix-vector products using either the same matrix or matrices with a very similar structure. A well-known example is the iterative solution of a system of linear equations where the number of equations make the use of direct solvers infeasible. During Newton-Raphson iterations, a matrix is constructed then a linear system is solved, but the matrix itself has the same structure in each iteration, just different values. In these cases, a sub-optimal multiplication algorithm can be tuned, and we will see that the overall performance can be greatly improved.

4. SPARSE MATRIX-VECTOR MULTIPLICATION ALGORITHM

During the multiplication of a sparse matrix with a vector, each element of a given row has to be multiplied with the corresponding element from the multiplicand vector, these products are added up and written to the result vector. The CSR format stores matrix data in three vectors as shown in figure 1. The first vector `rowPtrs` points to the first element of each row, and its last element is the total number of non-zeros. The second and third vectors, `colIdxs` and `values`, store the column indices and the values for each non-zero element in the matrix. The total number of rows is `dimRow` and the total number of non-zeros is `nnz`. The simplest serial version of the multiplication $y = Ax$ using the CSR format is described by algorithm 1.

Algorithm 1 Element by element assembly of the stiffness matrix and the load vector.

```
for i = 0 to dimRow-1 do
  $row = rowPtrs[i]$
  $y[i] = 0$
  for j = 0 to rowPtrs[i+1]-row-1 do
    $y[i] += values[row + j]*x[colIdxs[row + j]]$
  end for
end for
```

Based on algorithm 1 a multiplication and addition is performed for each non-zero in the matrix, thus the total number of floating-point operations is:

$$2 * \texttt{nnz}. \tag{1}$$

This formula will be used throughout the paper to calculate the instruction throughput.

Similarly, the number of bytes moved to and from off-chip memory without considering caching, is for each non-zero its value, column index and corresponding value from the multiplicand vector x and for every row the pointer to its first element and the write of the row sum to the result vector y. If there are no empty rows and columns, the amount of data to be moved:

$$\begin{aligned} &\texttt{nnz} * (2 * sizeof(float|double) + sizeof(int)) + \\ &\texttt{dimRow} * (sizeof(float|double) + sizeof(int)). \end{aligned} \tag{2}$$

Dividing this quantity by the execution time gives what is called the *effective* bandwidth, and this is the formula the figures in [3] are based on, thus for the purpose of comparability we use it throughout the paper. On a caching architecture this corresponds to each value on each cache line being read while it is in the cache, but only once.

The worst case scenario occurs when only one value of each cache line is used, thus in fact a lot more data is moved to the chip, but most of it is not read. In single precision a cache line holds 32 floats or integers, thus if each memory access loads an entire cache line, but reads or writes only one value on it before removing it from the cache, the actual amount of data moved to and from global memory:

$$\begin{aligned} 32 * (\ &\texttt{nnz} * (2 * sizeof(float) + sizeof(int)) + \\ &\texttt{dimRow} * (sizeof(float) + sizeof(int))\ ). \end{aligned} \tag{3}$$

In the best case scenario, with 100% data reuse, each element of the multiplicand vector has to be loaded from global memory only once, thus the lower bound on all data transfers considering perfect cache efficiency is:

$$\begin{aligned} &\texttt{nnz} * (sizeof(float|double) + sizeof(int)) + \\ &\texttt{dimRow} * (2 * sizeof(float|double) + sizeof(int)). \end{aligned} \tag{4}$$

4.1 The naive CUDA implementation

The simplest implementation of the CSR SpMV kernel in CUDA assigns one thread to each row, defines the block size (`blockSize`) to be a multiple of the warp size (currently 32), e.g. 128, and calculates the number of blocks (`gridSize`) accordingly. Since the maximum number of blocks is 65535, this particular kernel can only handle matrices with less than $65535 * 128 = 8388480$ rows. The sketch of the CUDA code is described by figure 2.

```
int i = blockIdx.x*blockSize + threadIdx.x;
float rowSum = 0;
int rowPtr = rowPtrs[i];
for (int j = 0; j<rowPtrs[i+1]-rowPtr; j+=1) {
        rowSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
}
y[i] = rowSum;
```

Figure 2: Naive CUDA kernel for performing the SpMV.

If the matrix has several elements per row, this implementation suffers from a high cache miss rate because the cache is not big enough to hold all of the cache lines being used. Moreover, if the number of non-zeros per row has a high variance over adjacent rows, some threads run for more iterations than others in the same warp, thus warp divergence may also become an issue.

4.2 Thread cooperation

To reduce the number of cache lines used when accessing the arrays `values` and `colIdxs`, multiple threads can be assigned to work on the same row [3]. The cooperating threads access adjacent elements of the row, perform the multiplication with the elements of the multiplicand vector x then add up their results in shared memory using parallel reduction. Finally, the first thread of the group writes the result to the vector y. Because threads in the same warp are executed in lockstep, the synchronisation between cooperating threads is implicit - as long as their number is a power of 2, and no more than the warp size. From here on, the number of cooperating threads assigned to each row is indicated by `coop`.

This technique can greatly improve the efficiency of caching; the same cache lines from the `values` and `colIdxs` arrays are used by the cooperating threads, thus there are more cache lines left for storing the values of the multiplicand vector. If the length of the rows is not a factor of the 32 then this algorithm may also suffer from branch divergence, the resulting loss in parallelism decreasing performance.

Because of the assignment of subsequent rows to subsequent groups of threads, the worst case scenario data transfer described by formula (3) can not happen, because either reading non-zeros or writing to the result vector uses the same cache line. With `coop`=1 and long rows, the cache lines storing non-zeros and their column indices may be removed from the cache after just one read, however writing to the elements of the result vector will use the same cache line. On the other hand, with `coop`=32, the cache lines storing non-zeros and column indices are fully utilised (unless row length is not a factor of 32), but writing to the elements of the result vector may force loading a new cache line every time. The effects of this trade-off will be seen in the performance evaluation.

4.3 Granularity

As shown earlier, if a group of cooperating threads is assigned to only one row, then the number of blocks required to process the entire matrix may be more than than 65535. To tackle this problem, a two dimensional grid may be used, which would extend the maximum number of blocks to 65535^2. To complement this approach more than one row per cooperating thread group can be processed. Thus, a thread block processes `repeat*blockSize/coop` contiguous rows.

For a fixed value of `repeat`, `blockSize` and `coop`, the total number of blocks is the following:

$$gridSize = 1 + (dimRow * coop - 1)/(repeat * blockSize). \tag{5}$$

If `repeat` is small, the algorithm is *fine grained* and if `repeat` is big, the algorithm is *coarse grained*. To have sufficient occupancy and load balancing on the GPU, the `gridSize` should be at least a few hundred, which may limit small matrices to execute in a fine grained way. However, for larger matrices, the difference in the number of blocks using the two approaches may be large.

Granularity is closely related to the efficiency of caching, or *cache blocking*; if for example the matrix has a diagonal structure, i.e. the rows access a contiguous block of the multiplicand vector, then the data reuse is improved by coarse grain processing because most of the values used are already in the cache. The optimal granularity depends on the structure of the matrix, which is not known in the general case.

4.4 The fully parametrised algorithm

The full parameter space of the algorithm is described by the following parameters: the number of threads per block (`blockSize`), the number of cooperating threads per row (`coop`) and the number of rows processed by each cooperating thread group (`repeat`). The total number of blocks (`gridSize`) is uniquely defined by these parameters according to equation (5). The complete algorithm is described by figure 3. For the sake of brevity, the parallel reduction is also parametrised with `coop`, whereas in our real implementation there is a different kernel for each different value of `coop`, thus the reduction is unrolled. Note that there is no use of *__syncthreads()* thread synchronisation due to the implicit warp synchronisation discussed in section 4.2.

5. PERFORMANCE EVALUATION

The performance measurements were obtained on a workstation with two Intel Xeon X5650 6-core processors, 24GBytes of system memory running Linux kernel 2.6.35. The system has 2 NVIDIA Tesla C2070 GPUs, both with 6GB global memory clocked at 1.5GHz and 384-bit bus width, 448 CUDA cores in 14 streaming multiprocessors (SMs) clocked at 1.15GHz. The GPU codes were compiled with NVIDIA's nvcc compiler with the CUDA 4.0 framework with the $--use\_fast\_math$ flag. The L1 cache size was set to 48kB, turning it off or reducing its size to 16kB decreased performance in all cases.

The test matrices were taken from the University of Florida Sparse Matrix Collection [7] based on the test cases of the CUSPARSE 4.0 library [16]. 15 of these matrices were used to train and tune our algorithms, and a total of 44 matrices were used to evaluate them and calculate performance figures [2]. A short summary of the training matrices is shown in table 1. These matrices represent a range of structured and unstructured matrices with different sizes, different averages and standard deviations in the length of the rows and different distributions of non-zeros.

In the following sections we describe the performance of the sparse matrix-vector multiplications when adjusting the parameters of the multiplication algorithm. These performance results are interpreted in light of the matrix structure to describe the underlying bottlenecks. For the sake of clarity, these parameters are discussed one by one, however it is shown that they are not independent.

```
__global__ void csrmv(float *values, int *rowPtrs,
                                          int *colIdxs, float *x, float *y,
                                          int dimRow, int repeat, int coop) {
    int i = (repeat*blockIdx.x*blockDim.x + threadIdx.x)/coop;
    int coopIdx = threadIdx.x%coop;
    int tid = threadIdx.x;
    extern __shared__ volatile float sdata[];
    for (int r = 0; r<repeat; r++) {
        float localSum = 0;
        if (i<dimRow) {
            // do multiplication
            int rowPtr = rowPtrs[i];
            for (int j = coopIdx; j<rowPtrs[i+1]-rowPtr; j+=coop) {
                localSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
            }
            // do reduction in shared mem
            sdata[tid] = localSum;
            for(unsigned int s=coop/2; s>0; s>>=1) {
                if (coopIdx < s) sdata[tid] += sdata[tid + s];
            }
            if (coopIdx == 0) y[i] = sdata[tid];

            i += blockDim.x/coop;
        }
    }
}
```

Figure 3: Parametrised algorithm for sparse matrix-vector multiplication.

5.1 Number of cooperating threads

The number of cooperating threads (`coop`) can be a power of 2, up to the warp size. Figure 4 shows the performance on different test matrices for different values of `coop`, with `blockSize = 128` and `repeat = 8`. The performance difference between the best and worst choice can be more than an order of magnitude. The figure also shows the square root of the average row length of the matrices; note how the optimal value of `coop` is close to this value. Table 2 lists the L1 cache hit rates and warp divergence reported by the CUDA Visual Profiler as a function of the number of cooperating threads. It is important to note that the cache hit rate usually increases with the increasing number of cooperating threads because multiple threads access the same cache lines for the matrix data leaving space for more cache lines to store the values of the multiplicand vector. However too high a value of `coop` results in a high fraction of inactive

[2]List of matrices not shown in table 1: amazon0505, cont1_l, filter3D, msdoor, troll, ct20stif, halfb, parabolic_fem, shipsec1, vanbody, BenElechi1, dc1, pkustk12, StocF-1465, bmw3_2, delaunay_n22, largebasis, poisson3Db, stomach, xenon2, boneS01, mc2depi, qcd5_4, tmt_sym, consph, F2, memchip, rma10, torso3

Table 1: Description of test matrices.

Name	`dimRow`	`nnz`	avg	std. dev.	Description
atmosmodd	1270432	8814880	6.9	0.24	diagonal, CFD problem
cage14	1505785	27130349	18	5.36	wide, directed weighted graph
cant	62451	4007383	64.2	14.05	narrow diagonal, FEM
cop20k_A	121192	2624331	21.6	13.8	very wide, 2D problem
F1	343791	26837113	39.5	40.86	very wide, FEM - AUDI engine
mac_econ_fwd500	206500	1273389	6.2	4.43	narrow, macroeconomics problem
pdb1HYS	36417	4344765	60.2	31.93	narrow, sporadic off-diagonals, protein 1HYS
scircuit	170998	958936	5.6	4.39	wide, circuit simulation
shallow_water1	81920	327680	4	0	narrow, some off-diagonals, CFD problem
webbase-1M	1000005	3105536	3.1	25.34	somewhat random, web connectivity
nd24k	72000	28715634	398	76.9	wide, sporadic, ND problem set
crankseg_2	63838	14148858	221	95.8	wide, structural problem
pwtk	217918	11524432	52.8	5.44	narrow, FEM - pressurised wind tunnel
ldoor	952203	42493817	44.6	14.7	very wide, structural problem
2cubes_sphere	101492	1647264	16.2	2.65	wide, sporadic, electromagnetics simulation

Table 2: Cache hit rates and warp divergence for test matrices at different values of `coop`, based on the CUDA Visual Profiler.

`coop`	1	2	4	8	16	32
atmosmodd						
L1 hit rate (%)	54.5	77.5	79.1	75.3	86.0	90.6
Divergence (%)	2.6	13.3	13.79	15.9	16	16
crankseg_2						
L1 hit rate (%)	20.4	48.4	56.1	62.0	60.6	64.4
Divergence (%)	1.8	4.1	5.6	4.8	5.1	5.3
shallow_water1						
L1 hit rate (%)	71.8	60.8	59.0	75.7	87.0	92.1
Divergence (%)	0.0	0.0	0.0	16.0	16.0	16.0
webbase-1M						
L1 hit rate (%)	72.8	81.2	73.6	80.6	86.9	92.1
Divergence (%)	9.5	9.2	16.8	16.55	16.1	15.9
cant						
L1 hit rate (%)	19.56	50.9	60.9	66.9	74.7	78.6
Divergence (%)	8	12.8	7.2	9.9	5.1	6.7

threads: e.g. shallow_water has four non-zeros in every line, thus assigning 8 threads to process each row would result in half of the threads being inactive all the time.

5.2 Level of granularity

The number of rows processed by each cooperating thread group is the most important factor in determining the total number of blocks (`gridSize`). As figure 5 shows, for larger matrices the difference in performance is relatively small (around 10%), however for smaller matrices like *shallow_water1* high values of `repeat` results in low occupancy - in this case at `repeat = 32` there are only 20 blocks. In CUDA, the user is not in control of the execution scheduling of blocks, thus for larger structured matrices higher values of `repeat` offer marginally better data reuse. On the other hand, fine grained processing (more blocks) offers better load balancing across the SMs.

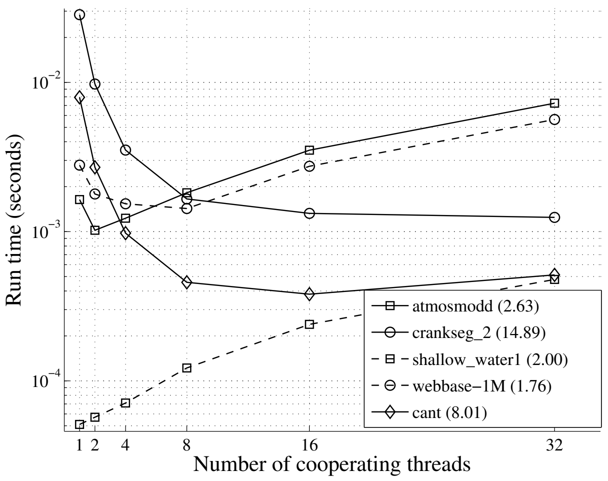

Figure 4: Performance as a function of the number of cooperating threads. `blockSize = 128` and `repeat = 8`. The square root of the average row length is displayed in brackets after the name of the matrix.

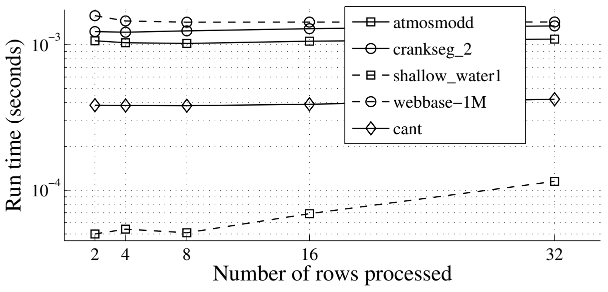

Figure 5: Performance as a function of the number of rows processed per cooperating thread group. `blockSize = 128` and `coop` is fixed at its optimal value according to figure 4.

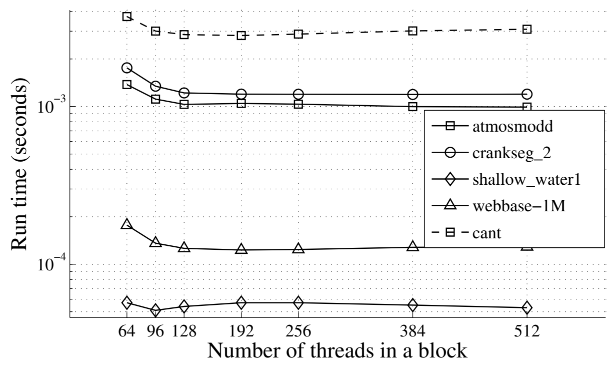

Figure 6: Performance as a function of the number of threads in a block. `repeat = 4` and `coop` is fixed at its optimal value according to figure 4.

5.3 Number of threads in a block

The number of threads in a block (`blockSize`) plays an important role in determining occupancy: the maximum number of threads per SM (in Fermi) is 1536, and the maximum number of blocks per SM is 8. Thus a small block size leads to low occupancy, 33% with `blockSize=64` ($64 * 8 = 512$). Our algorithm also uses shared memory, storing one floating point number for each thread, which does not limit occupancy; since a double precision number is 8 bytes long, with 100% occupancy the total amount of shared memory used is 12.2kB. The optimal block size however is usually smaller than what is required for full occupancy to balance parallelism and cache size per thread. Double precision storage also effectively halves the number of values held in the L1 cache. For this reason the optimal value of the parameter is often even smaller in double precision. Figure 6 shows the impact of `blockSize` on performance.

5.4 The fixed rule

For general-purpose SpMV code, a constant time *fixed rule* is required to decide the input parameters for the multiplication algorithm. For this purpose we first implemented an exhaustive search algorithm that tests a wide range of parameters and ran the training matrices through it. Based on this data, we fine-tuned our fixed rule to find the optimal parameter values that provide the highest average performance. Since this rule is intended for use by a general-purpose library, this average is not a weighted average, performance is equally important for smaller and larger matrices.

The fixed rule defines the parameters as follows:

1. `blockSize = 128`
2. `coop` is the smallest power of two which is larger than the square root of the average row length, up to a maximum limit of the warp size (32).
3. `gridSize` and `repeat` is calculated in a way that ensures there are at least 1500 blocks.

The calculation of these parameters incurs virtually no overhead, since only the number of rows and the number of non-zeros is required.

Figures 7 and 8 describe performance in single precision and figures 9 and 10 in double precision. All bandwidth

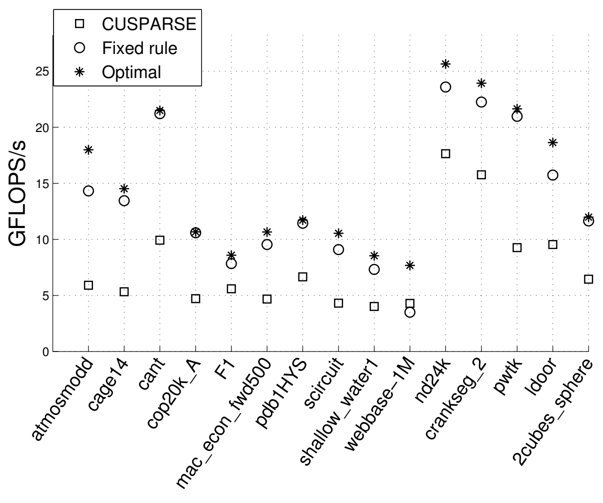

Figure 7: Single precision floating point operation throughput of the sparse matrix-vector multiplication.

numbers show effective bandwidth as described by equation (2). Since instruction throughput is directly proportional to the number of non-zeros processed per second (equation (1)), it accurately describes the relative efficiency of data reuse. Note that the bandwidth of the matrix (the distance of non-zeros from the diagonal) is not directly related to throughput. For example *crankseg_2* has non-zeros everywhere in the matrix but the non-zeros in *pwtk* lie very close to the diagonal, and yet their performance is very similar because they are both structured matrices, thus non-zeros in consecutive rows have similar column indices which enables efficient caching. The average row length and the relative value of the standard deviation of the length of the rows has some effect on the performance, however the most important factor is the "structuredness" of the matrix which is difficult to describe.

5.5 Estimating bandwidth and caching

Block and warp scheduling on GPUs is out of the control of the programmer, and the exact parameters of Fermi's L1 cache, like associativity, are not public knowledge, thus exact modelling of the cache is extremely difficult and out of the scope of this paper. It is however possible to estimate the number of cache lines loaded by each block. We make the following assumptions: (1) cache lines are not shared between blocks, (2) all cache lines loaded by a block stay in the cache until the execution of that block finishes. These assumptions only affect the caching of the multiplicand vector significantly, because the access pattern to other data structures does not depend on matrix structure. Due to the semi-random assignment of thread blocks to SMs and because the L1 cache is probably not fully associative, we argue that the first assumption does not overestimate actual amount of data moved by too much. The second assumption however implies unconstrained cache size per block, thus underestimates the actual amount of data moved. We argue that this error is only significant if the temporal difference between accesses to same cache lines is high; for structural matrices this is usually low, for non-structural matrices data

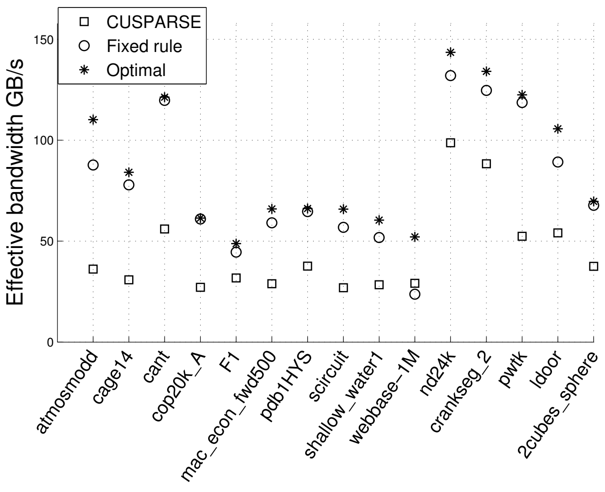

Figure 8: Effective bandwidth of the sparse matrix-vector multiplication in single precision.

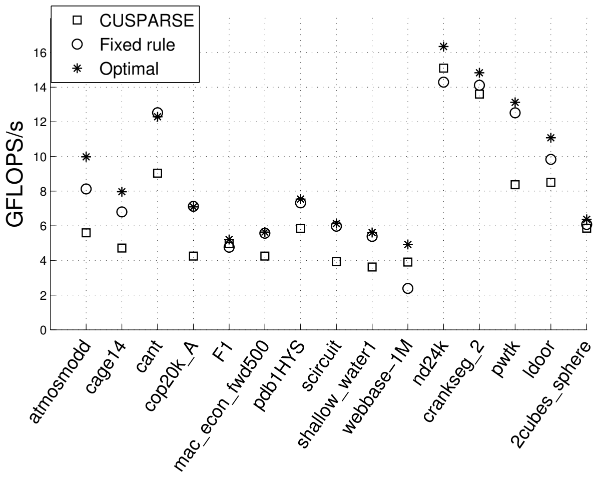

Figure 9: Double precision floating point operation throughput of the sparse matrix-vector multiplication.

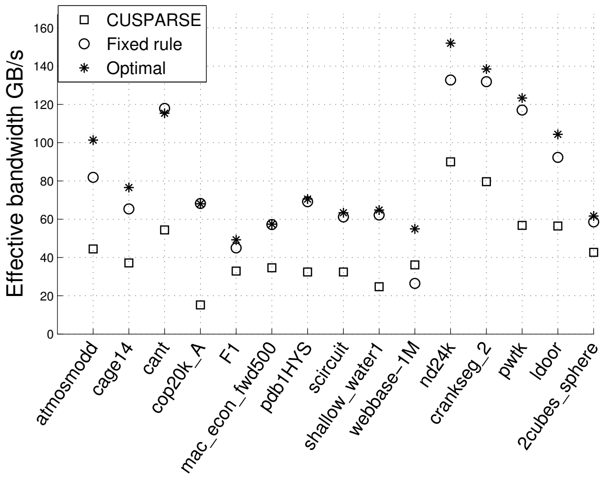

Figure 10: Effective bandwidth of the sparse matrix-vector multiplication in double precision.

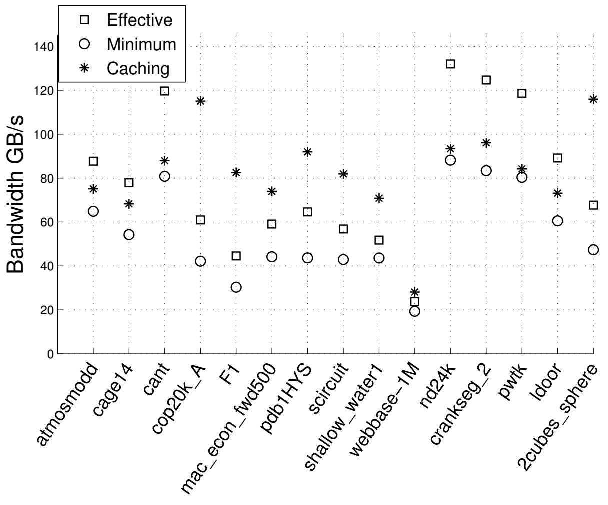

Figure 11: Values of the minimum, the effective and the caching bandwidth in single precision.

reuse is low anyway. The effects of the L2 cache are ignored.

Figure 11 shows calculated bandwidth estimates when using the fixed rule, according to formulas (2) and (4), and based on the number of cache lines touched by each thread block. Note that bandwidth in some cases is over 80 GB/s even according to the minimum formula that assumes perfect data reuse. This is a very high fraction of the theoretical peak (144 GB/s) considering that in practice a bandwidth over 100 GB/s is rarely achieved. It is important to observe that for structured matrices data reuse between subsequent rows is high, thus caching bandwidth falls between the minimum and the effective bandwidth. In the case of non-structural matrices, several values of the cache lines holding elements of the multiplicand vector are not read by that block at all, thus caching bandwidth is even higher than the effective bandwidth, because part of the data moved is not used. Caching bandwidth has a mean of 83 GB/s with a relatively low variation compared to other bandwidth figures, which hints at the average hardware utilisation for the problem of sparse matrix-vector multiplication. In some extreme cases, caching bandwidth exceeds 120 GB/s, probably due to cache lines being in fact shared between blocks.

5.6 Reducing matrix bandwidth

Several matrices in our test collection have very wide bandwidth, which implies that during the SpMV, a wide range of memory locations are accessed in the multiplicand vector. This can potentially decrease performance because cache locality may be low. To evaluate what the performance of SpMV on narrower-band matrices is, we converted the test matrices using Matlab's reverse Cuthill-McKee renumbering algorithm. Timing results show that for most matrices the performance difference was negligible, though for a few test cases (e.g. webbase-1M) performance increased by up to 50%. However, the average speed-up over the 15 test matrices is below 5% for both CUSPARSE and the fixed rule because the performance was negatively affected by the renumbering in some test cases.

6. RUN-TIME PARAMETER TUNING

The performance of the SpMV multiplication using the fixed rule is in many cases close to the optimal, however there is room for improvement in some other cases. An extreme example is *webbase-1M*, where the fixed rule only achieves 50% of the optimal performance, but matrices like *atmosmodd, cage14, F1* and *ldoor* can also benefit from improved parameters. In a general-purpose library it is not feasible to perform off-line parameter tuning because it may have a significant overhead. It is also unknown how many times a sparse matrix-vector multiplication routine is called from the user's code, and whether those calls use the same matrix or different matrices.

We argue that if subsequent calls to the SpMV routine use the same pointers to matrix data and have the same amount of rows and non-zeros then the matrix structure is not changing significantly across multiple executions. This hypothesis enables the library to test slightly different parameters when performing the multiplication in order to find the best set of parameters. This of course may result in having to run a few multiplications with worse performance than the fixed rule, but if the number of iterations is large enough, then this overhead is compensated by the improved overall performance. It is important to note that in such situations there is no statistical data available, thus system noise may affect measured performance.

We propose an empirical algorithm that is based on our experience with different parameter settings and the parameters found by exhaustive search. The main concepts of the algorithm are the following:

1. As the first step, double the number of blocks by halving `repeat`. If the change is higher than 5%, proceed optimising `repeat` (step 3). If the change is less than 5%, double `coop` and increase `blockSize` to 192 (step 2).

2. If doubling `coop` increased performance then try increasing it further, if it did not, start decreasing it.

3. Depending on whether halving `repeat` increased or decreased performance, further decrease or increase it.

4. Finally the value of `blockSize` is changed in increments (or decrements) of 32, but no more than 512 and no less than 96 in single and 64 in double precision.

Figure 12 shows the impact of tuning on performance in the single precision case. The fixed rule achieves 84% of the optimal performance on average over the 15 training matrices, and 73% over all 44 test matrices. Figure 13 shows the improvement of performance over subsequent iterations due to the run-time tuning. The performance surpasses the 95% threshold after just 5 iterations, and 98% after 8 iterations. Similar figures apply in double precision.

6.1 Performance analysis on the full test set

To provide a fair assessment of our algorithm we ran it through a set of 44 test matrices and calculated average instruction throughput, bandwidth and speedup figures. We found one matrix for which the CSR format and the standard multiplication algorithm proved unsuitable (*dc1*): not even the exhaustive search found parameters that would

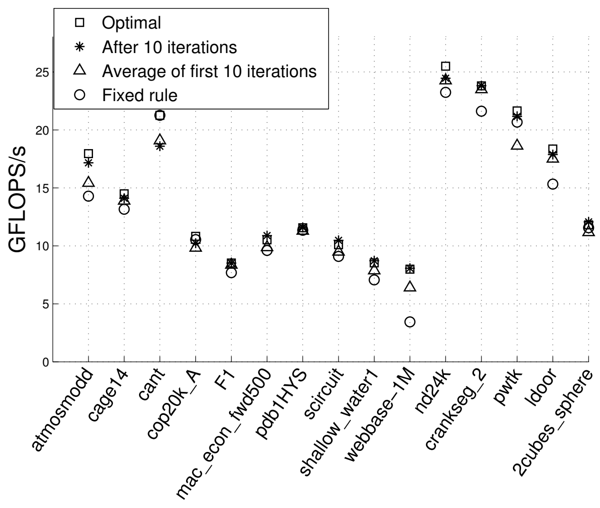

Figure 12: Single precision floating point instruction throughput after 10 iterations of run-time tuning.

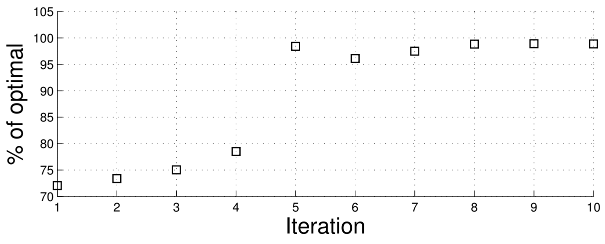

Figure 13: Relative performance during iterations compared to optimal, averaged over the 44 test matrices.

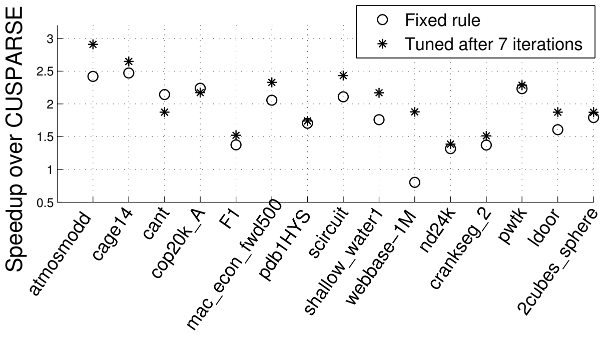

Figure 14: Speedup over CUSPARSE 4.0 CSR in single precision using the fixed rule and run-time tuning.

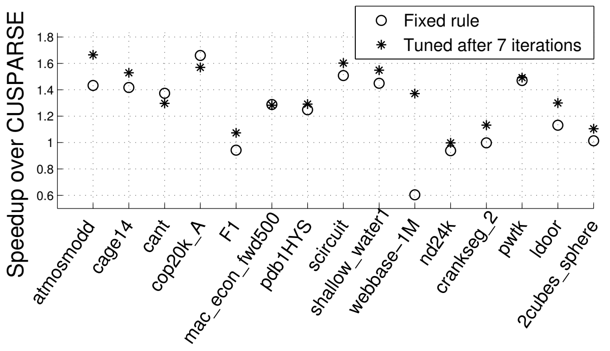

Figure 15: Speedup over CUSPARSE 4.0 CSR in double precision using the fixed rule and run-time tuning.

Table 3: Performance metrics on the test set.

	CUSPARSE	Fixed rule	Tuned
Throughput single GFLOPS/s	7.0	14.5	15.6
Throughput double GFLOPS/s	6.3	8.8	9.2
Min Bandwidth single GB/s	28.4	58.9	63.7
Min Bandwidth double GB/s	38.7	54.0	56.8
Speedup single over CUSPARSE	1.0	2.14	2.33
Speedup double over CUSPARSE	1.0	1.42	1.50

have brought performance above 4 GFLOPS/s. This particular matrix has an average row length of 6.5, however there are two very long adjacent rows (114200 and 47190 nonzeros) which results in severe load imbalance that radically decreases performance. Using the standard SpMV algorithm would be clearly inappropriate in this case; the simplest solution is to process those two rows separately with multiple blocks performing two parallel reductions and the rest is processed by the standard algorithm. Results from the standard algorithm on *dc1* are omitted. Performance metrics in table 3 clearly indicate that there is a wider performance gap between the fixed rule and the tuned version, since the fixed rule itself was trained on the first 15 matrices.

7. COMPARISON WITH THE ELLPACK FORMAT

With the release of CUDA 4.1, NVIDIA enables the users of the CUSPARSE library to convert their matrices to the ELLPACK/HYB format then use it to perform matrix operations. Their library enables conversion from the dense, CSR and COO formats, however the data structure of the new matrix is hidden from the user, thus if a value of the matrix has to be changed the conversion has to be repeated. This not only incurs the overhead of the conversion itself but requires more than twice as much memory since two copies of the same matrix are stored in two different storage formats. Thus, the real question is whether it is worth converting a matrix to the ELLPACK format or not.

To test the performance of the ELLPACK/HYB format, we ran the 44 matrix test set through the conversion process and tested the sparse matrix-vector multiplication itself. The speedup of the multiplication itself over CUSPARSE CSR in single precision on all 43 matrices (without *dc1*) is 1.88 times which is somewhat worse than our fixed rule. In case of *dc1*, the speedup over the standard algorithm was 50 times, which means that the standard ELLPACK algorithm can handle matrices with a few very long rows. Because of the necessity of conversion it is important to see the minimum number of sparse matrix-vector products after which it is worth doing. Since in this case the same matrix has to be used to perform the multiplications we compared it to our tuned multiplication algorithm. In most of the cases it is either not worth the conversion at all (because our algorithm is faster) or thousands of iterations are required. There are 6 matrices in single precision and seven more in double precision where the number of iterations is reasonable (below 100), they are shown on figure 16. Even though the multiplication may be significantly faster, the time the conversion takes may be equivalent to up to a hundred multiplications.

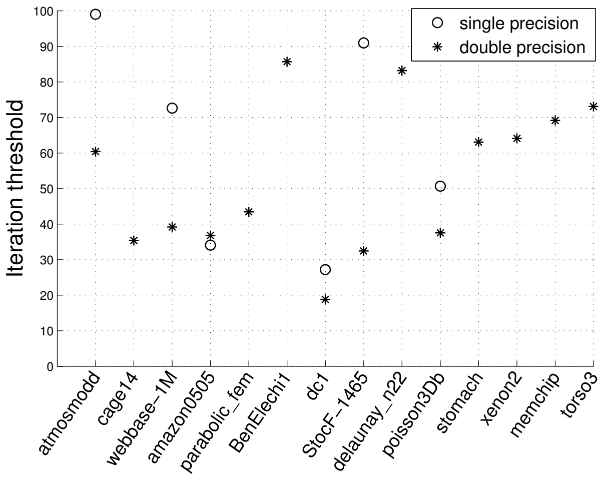

Figure 16: Number of iterations required to be worth converting to ELLPACK/HYB.

8. FINITE ELEMENT PROBLEM

As a final test of the CSR storage format, we present an application - the solution of finite element equations - to compare CSR to ELLPACK and a problem specific storage format.

8.1 Mathematical background

The finite element method (*FEM*) is a powerful numerical method for approximating the solution of partial differential equations (*PDEs*) [11]. The method is based on the polygonal discretisation of the domain Ω over which the PDE is to be solved. Equation (6) describes a simple elliptic problem with a Dirichlet boundary condition:

$$-\nabla \cdot (\kappa \nabla u) = f \text{ in } \Omega, \quad (6)$$
$$u = 0 \text{ on } \partial\Omega. \quad (7)$$

The solution is sought in the form of $u : \Omega \to \Re$, with $u = 0$ on $\partial\Omega$. The standard finite element method constructs a finite dimensional space V_h of functions over Ω, and searches for an approximate solution $u_h \in V_h$. If the number of vertices in the discretisation of Ω is N_e, then let $\{\phi_{1...N_v}\}$ be a basis for V_h, then:

$$u_h = \sum_i \bar{u}_i \phi_i \tag{8}$$

To find the best approximation to u, it is necessary to solve the system:

$$K\bar{u} = \bar{l}, \tag{9}$$

where K is the $N_v \times N_v$ matrix, usually called the *stiffness matrix*, defined by:

$$K_{ij} = \int_\Omega \kappa \nabla\phi_i \cdot \nabla\phi_j \, dV, \ \ \forall i,j = 1,2,\ldots,N_v, \tag{10}$$

and $\bar{l} \in \Re^n$, usually called the *load vector*, is defined by:

$$\bar{l}_i = \int_\Omega f \cdot \phi_i \, dV, \ \ \forall i,j = 1,2,\ldots,N_v. \tag{11}$$

If the underlying discretisation mesh has nodes $\bar{x}_i$, it is possible to choose a finite element space V_h with basis functions such that $\phi_i(\bar{x}_j) = \delta_{i,j}$. In this case u_h is determined by its values at $\bar{x}_i, \ i = 1,2,\ldots N_v$. The mesh is a polygonal partitioning of the domain Ω into a set of disjoint elements $e_i \in E, \ i = 1\ldots N_e$. The basis functions are constructed so that ϕ_i is nonzero only over those elements e which have $\bar{x}_i$ as a vertex. This means that finite element basis functions ϕ_i have their support restricted to neighbouring elements, thus the integral in equation (10) is non-zero only if the two vertices belong to the same element.

8.2 Storage formats

In our GPU implementation the stiffness matrix described by equation (10) is stored in three formats:

1. CSR, as described above, uses three vectors, one for pointers to the first element of each row, one storing the values of the non-zeros and one storing their column indices.

2. ELLPACK stores non-zeros and column indices in a `dimRow` by `K` matrix, where `K` is the maximum row length. This matrix is transposed in GPU memory so that the n^{th} non-zero of each row is in one contiguous block of memory, enabling coalesced memory transfers.

3. LMA stands for Local Matrix Approach which exploits the fact that during the iterative solution of the linear system $K\bar{u} = \bar{l}$ the stiffness matrix K is not required explicitly. During matrix assembly an $m * m$ local matrix is calculated for each element. In the global matrix approach these values are scattered according to the vertex indices, but the local matrix approach stores them as they are, and calculates the matrix-vector product in the following way [5, 13]:

$$\bar{y} = \mathcal{A}^T(K_e(\mathcal{A}\bar{x})), \tag{12}$$

where K_e is the matrix containing the local matrices in its diagonal and $\mathcal{A}$ is the local-to-global mapping from the local matrix indices to the global matrix indices. In a similar way to ELLPACK the values of the local matrices are stored in a way that the n^{th} value of each one is in one contiguous block of memory. This approach stores stiffness data redundantly, however it does not have to store row and column indices explicitly as it is available in the structural description of the underlying mesh.

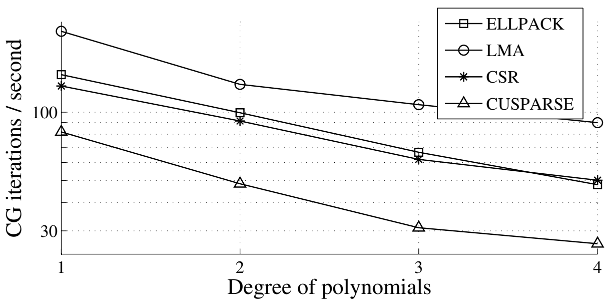

Figure 17: Number of CG iterations per second as a function of the degree of polynomials used, in single precision.

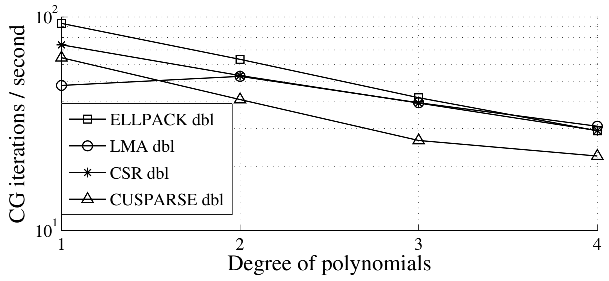

Figure 18: Number of CG iterations per second as a function of the degree of polynomials used, in double precision.

8.3 Performance of the solution

For the solution of the linear system $K\bar{u} = \bar{l}$, the conjugate gradient algorithm was used. The underlying problem to be solved is a simple elliptic Poisson problem. The discretisation uses four million degrees of freedom, thus the matrix K has four million rows. The degree of polynomials used as basis functions ranges from 1 to 4 making the row length 9 in the first degree case and up to 81 in the fourth order case.

Because of its data-scatter nature, LMA has to avoid race conditions when incrementing the result vector. In single precision, atomic operations are used, however in double precision these are not available and so colouring and explicit synchronisation is employed.

Figures 17 and 18 show the advantage of the LMA approach which has to move significantly less index data. It is important to observe, that the performance difference between CSR and ELLPACK is small compared to the average difference presented in [3]. In double precision the LMA performs worse because of expensive synchronisation operations required to avoid write conflicts. However, its relative performance increases with the increasing degree of polynomials.

9. CONCLUSION

In this paper we thoroughly analysed the factors that impact the performance of sparse-matrix vector multiplication algorithms using the CSR format. We demonstrated the effects of parameters on caching efficiency, and showed that proper use of Fermi's caching mechanisms can lead to significant improvements in performance. Based on these experiments, we proposed a fixed rule to determine a set of parameters for any given input matrix. Furthermore, we introduced a dynamic run-time parameter tuning algorithm that detects if a matrix is used repeatedly to calculate matrix-vector products and improves its performance by tuning its parameters with virtually no overhead.

We demonstrate an average speedup of 2.1 times over CUSPARSE 4.0 CSR in single precision and 1.4 times in double precision using the fixed rule. The run-time tuning algorithm further improves this speedup by 10-15%.

We compared the performance of our CSR algorithms to the ELLPACK/HYB format that was introduced with CUSPARSE 4.1. We found that in most cases the difference was negligible but in a very few cases it was significant (for one matrix in our test set). Taking the cost of conversion into account we showed that for about 75% of the matrices it was not worth the conversion and in the rest of the cases 30 to 90 SpMV products with the same matrices are required before it is worth doing.

Through the solution of finite element problems we compared the performance of different sparse storage formats. We demonstrated that for a problem-specific algorithm, the local matrix approach, can provide more than 50% better performance provided atomic operations are available.

Comparing our bandwidth and instruction throughput figures to those in [3] (off-chip bandwidth is the same on both test cards), and based on our own experiments, we think our implementation effectively closes the performance gap between ELLPACK and CSR, more so if we consider the cost of conversion from CSR to ELLPACK. This has important implications for the development of high performance applications, as the use of the traditional CSR format is no longer penalised severely.

10. ACKNOWLEDGMENTS

This research was supported in part by the UK Engineering and Physical Sciences Research Council through project EP/J010553/1 on "Algorithms and Software for Emerging Architectures", and in part by the EU LLP/Erasmus program 10/2010-2011/Erasmus-SMP.

11. REFERENCES

[1] J. Anderson, C. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[2] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 2008.

[3] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, 2003.

[5] C. Cantwell, S. Sherwin, R. Kirby, and P. Kelly. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids*, 43(1):23 – 28, 2011.

[6] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, 2010.

[7] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011.

[8] A. Dziekonski, A. Lamecki, and M. Mrozowski. A memory efficient and fast sparse matrix vector product on a GPU. *Progress In Electromagnetics Research*, 116:49–63, 2011.

[9] A. H. El Zein and A. P. Rendell. Generating optimal CUDA sparse matrix-vector product implementations for evolving GPU hardware. *Concurrency and Computation: Practice and Experience*, 24(1):3–13, 2012.

[10] M. R. Hugues and S. G. Petiton. Optimized sparse matrix formats on GT200 and Fermi GPUs. In *7th International Conference On Preconditioning Techniques For Scientific And Industrial Applications*, 2011.

[11] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Cambridge University Press, 1987.

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

[13] G. R. Markall, D. A. Ham, and P. H. Kelly. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science*, 1(1):1815 – 1823, 2010.

[14] A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5657 of *Lecture Notes in Computer Science*, pages 289–297. Springer Berlin / Heidelberg, 2009.

[15] NVIDIA. *Fermi Compute Architecture Whitepaper*, 2009.

[16] NVIDIA. *CUSPARSE library*, 2011.

[17] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. W. Hwu, Z.-P. Liang, and B. P. Sutton. Accelerating advanced MRI reconstructions on GPUs. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 261–272, 2008.

[18] F. Vázquez, G. Ortega, J. Fernández, and E. Garzón. Improving the performance of the sparse matrix vector product with GPUs. In *Proceeding of the 10th International Conference on Computer and Information Technology (CIT)*, pages 1146 –1151, 2010.