Sparse Computations on GPGPUs[*]

Davide Barbieri
Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Roma, Italy
samuelmurdoch@gmail.com

Valeria Cardellini
Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Roma, Italy
cardellini@ing.uniroma2.it

Salvatore Filippone
Dipartimento di Ingegneria Industriale
Università di Roma "Tor Vergata", Roma, Italy
salvatore.filippone@uniroma2.it

Università di Roma "Tor Vergata"
Dipartimento di Informatica, Sistemi e Produzione
Technical Report RR-12.90

January 12, 2012

Abstract

Sparse matrix computations are ubiquitous in scientific computing; General-Purpose computing on Graphics Processing Units (GPGPU) is fast becoming a key component of high performance computing systems. It is therefore natural that a substantial amount of effort has been devoted to implementing sparse matrix computations on GPUs.

In this paper, we discuss our work in this field, starting with the data structures we have employed to implement common operations, together with the software architecture we have devised to allow interoperability with existing software packages. To test the effectiveness of our approach we have run experiments with it on two platforms; the experimental results show that our data structures allow us to achieve very good performance results, significantly better than what can be obtained with the most recent version of the CUSPARSE library.

1 Introduction

Graphics Processing Units (GPUs) have steadily entered as an attractive choice the world of scientific computing, building the core of the most advanced supercomputers and even being offered as an infrastructure service in Cloud computing (e.g., Amazon EC2). The GPU cards produced by NVIDIA are today among the most popular computing platforms; their architectural model is based on a scalable array of multi-threaded streaming multi-processors, each composed by a fixed number of scalar processors, one dual-issue instruction fetch unit, one on-chip fast memory with a configurable partitioning of shared memory and L1 cache plus additional special-function hardware. Each multiprocessor is capable of creating and executing concurrent threads in a completely autonomous way, with no scheduling overhead, thanks to the hardware support for thread synchronization.

NVIDIA's CUDA is a programming model designed for the data-parallel processing capabilities of NVIDIA GPUs [14, 19]. A CUDA program consists of a host program that runs on the CPU host, and a kernel program that executes on the GPU itself. The host program typically sets up the data and transfers it to and from the GPU, while the kernel program processes that data. The CUDA programming environment specifies a set of facilities to create, identify, and synchronize the various threads involved in the computation.

In addition to the primary use of GPUs in accelerating graphics rendering operations, there has been considerable interest in exploiting GPUs for General Purpose computation (GPGPU) [12]. A large variety of complex algorithms

[*]This Technical Report has been issued as a Research Report for early dissemination of its contents. No part of its text nor any illustration can be reproduced without written permission of the Authors.

can indeed exploit the GPU platform and gain tremendous performance benefits (e.g., [10, 1, 21]). With their advanced SIMD architectures, GPUs appear also ideal candidates for performing fast operations on sparse matrices, such as the matrix-vector multiplication; this is an essential kernel in engineering and scientific computing and hence over the last two decades its efficient implementation on general-purpose architectures has been addressed in a significant amount of work. Its importance stems from the fact that a matrix-vector product is the central computational step in the execution of the most commonly used iterative solvers for linear equations, i.e., solvers based on the so called Krylov subspace projection methods [18].

Matrix-vector multiplication on GPUs presents some new challenges, because optimization techniques applied in general-purpose architectures cannot be directly applied on them. Moreover, sparse matrix structures introduce additional challenges with respect to their dense counterparts, because operations on them are typically much less regular in their access patterns. Therefore, in the last two years there has been a surge of interest about the efficient implementation of sparse matrix-vector multiplication, e.g., [3, 4, 5, 13, 17, 20], NVIDIA's CUDPP [15].

In this paper, we consider the *sparse matrix-vector multiply* (SpMV) operation $y \leftarrow \alpha Ax + \beta y$ where A is large and sparse and x and y are column vectors. The storage format used for the matrix A is a crucial factor in determining the performance of sparse matrix computations.

To address this issue, we have designed and developed a GPU-friendly storage format for sparse matrices and we have integrated it in our software libraries [8, 6] to extend the support provided by the GPU platform. The GPU-friendly storage format we propose is a variant of the standard ELLPACK (or ELL) format, which was previously designed for vector architectures, and is therefore named as ELLG. To test the effectiveness of our approach, we have run experiments with it on two different platforms, a desktop machine and a cluster, and we have compared the performance attained by our ELLG format to that of the CSR storage format of the CUDA CUSPARSE library, wrapping both the GPU side kernels in an existing library for parallel sparse applications. Our results show that the proposed ELLG format outperforms the CSR storage format of CUSPARSE.

The remaining of this paper is organized as follows. In Section 2 we present our GPU-friendly sparse matrix format. In Section 3 we describe the software architecture we have devised to allow interoperability with existing software packages. In Section 4 we discuss our experimental results. Finally, we draw some conclusions and give hints for future work in Section 5.

2 A GPU friendly sparse matrix format

According to the definition given by Wilkinson:

> A matrix is sparse when there are so many non-zeros that it pays off to take advantage of them in the computer representation.

It should therefore not be surprising that the performance of sparse matrix computations depends critically on the specific representation chosen. There are multiple factors that contribute in determining the overall performance:

- the match between the data structure and the underlying computing architecture, including the possibility of exploiting special hardware instructions;
- the amount of overhead due to the explicit storage of indices;
- the amount of padding with explicit zeros that may be necessary.

Many storage formats have been invented over the years; a number of attempts have also been directed at standardizing the interface to these data formats for convenient usage (see e.g., [7]).

Three widely-used data formats for representing sparse matrices are COOrdinate (COO), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC).

The COO format is a particularly simple storage scheme. As shown in Figure 2, three arrays (`Elements`, `Col idx`, and `Row idx`) store the values, column indices, and row indices, respectively, of the non-zero entries.

The CSR format is perhaps the most popular general-purpose sparse matrix representation. It explicitly stores column indices and non-zero values in two arrays (`Elements` and `Col idx` in Figure 3). A third array of row pointers (`Row Pointer` in Figure 3) allows the CSR format to represent rows of varying length. The name is based on the fact that the row index information is compressed with respect to the COO format. Figure 3 illustrates the CSR representation of the example matrix shown in Figure 1.

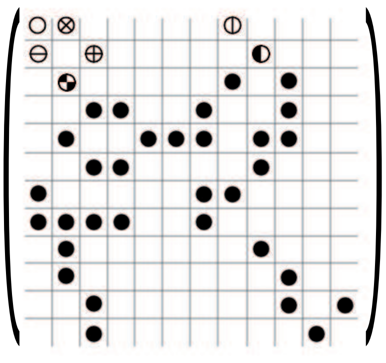

Figure 1: Example of sparse matrix

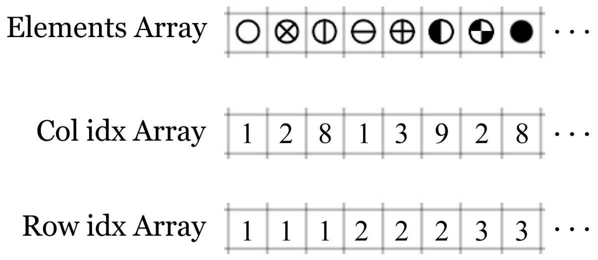

Figure 2: COO compression of matrix in Figure 1

Finally, the CSC format is similar to CSR except that the matrix values are read first by column, a row index is stored for each value, and column pointers are stored.

The previous data formats can be thought of as “general-purpose”, at least to some extent, in that they can be used on most computing platforms with little changes. Additional (and somewhat esoteric) formats become necessary when moving onto special computing architectures. Many examples of these formats are available from the vector computing era, including the ELLPACK (or ELL) and Jagged Diagonals (JAD) formats. For example, the latter represents the (compressed) diagonals occurring in the matrix obtained by sorting the CSR index arrays according to their population. The main issue with vector computers was to find a good compromise between the amount of overhead and the introduction of a certain amount of “regularity” in the data structure allowing the use of vector instructions.

The situation in the case of GPUs bears both resemblances and differences: we have to accommodate for the coordinated action of multiple threads in an essentially SIMD fashion; however, we have to make sure that we have many independent threads available, the number of threads in action at any given time being much larger than the typical vector lengths of vector computers.

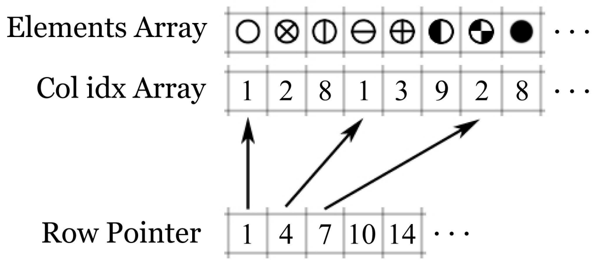

Figure 3: CSR compression of matrix in Figure 1

The ELLPACK/ITPACK format (shown in Figure 4) in its original conception comprises two 2-dimensional arrays (`Elements` and `Col idx`) containing $n * k_{max}$ elements, where n is the number of rows and k_{max} is the maximum number of non-zeros on the same row [11]. The rows of these arrays contain:

- the non-zero elements (in `Elements`) ;
- their respective column indices (in `Col idx`);
- padding with zeros, where necessary.

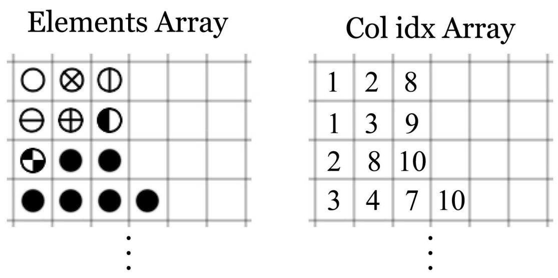

Figure 4: ELLPACK compression of matrix in Figure 1

For our CUDA implementation, we searched for the matrix format that allows to reach the maximum performance of one of the most frequent building blocks in linear algebra's applications, that is the sparse matrix-vector multiply routine. Sparse matrix-vector multiply is a data-parallel routine of course, but it is memory-bounded anyway, since we have a little amount (in CUDA scale) of floating point operations per memory access, and we cannot provide an efficient memory access pattern on GPU to both sparse matrix and input vectors. Therefore, our solution was to adapt the ELLPACK format to a GPU implementation, since it provides a straightforward way to read the sparse matrix values and one of the input vectors in a efficient manner (i.e., using coalesced accesses on NVIDIA GPUs), provided that we choose a pitch for arrays that makes them aligned to the appropriate boundaries in memory (128B is a good choice for all compute capabilities). This is possible if every thread computes one of the resulting elements, and thus reads a whole compressed row (assuming a column-major order). To deal with the different number of non-zero elements per row, a simple solution is to introduce padding (elements equal to zero) to fill unused locations of the elements array. An even better solution is to create an additional array of row lengths, at the cost of just one more access per row; this is similar to CSR storage, except that we are keeping the regular memory occupancy per row implied by the usage of 2-dimensional arrays. We found that other groups have used a similar solution, e.g., in the ELLPACK-R format described in [20].

The code that uses the format is listed below:

```
{ y = A*x }
procedure spMV(ValuesA, IndicesA ,RowsDim, x, y)
  idx = threadIdx + blockIdx*blockSize
  res = 0;
  for i=0 to RowsDim(idx)
  begin
    ind = IndicesA(i, idx)
    val = ValuesA(i, idx)
    res = res + val*x[ind]
  end
  y(idx) = res
end
```

For this code, memory accesses to array x were implemented in order to use the full L1+L2 caches hierarchy on the NVIDIA's Fermi GPU architecture, whereas the *texture cache* on GPU devices with previous compute capabilities. Such implementation still approaches to inefficiency when the non-zero elements are spread to the whole row, so that we cannot take advantage from the caching values from array x (because we have highly varying indices). Another main advantage of the integration of a non-zero elements counter array is that we know from the beginning of the main loop the length of the processing row, so we can operate programming techniques such as loop unrolling and prefetching. Listed below we report the complete kernel for the single precision *spMV* routine:

```
#define THREAD_BLOCK 128

// Operation is y = alpha*Ax+beta*y
// x and y are vectors
// cM is the elements array
// rP is the column pointer array
// rS is the non-zero count array
// n is the number of rows
__global__ void
Sspmvm_gpu_unroll_2_krn(float *y, float alpha,
        float* cM, int* rP, int* rS, int n,
        int pitch, float *x, float beta)
{
 int i=threadIdx.x+blockIdx.x*THREAD_BLOCK;
 if (i >= n) return;

 float y_prod = 0.0f;
 int row_size = rS[i];

 rP += i;
 cM += i;

 for (int j = 0; j < row_size / 2; j++)
 {
  int pointers[2];
  float values[2];
  float fetches[2];

  // Prefetching pointers and vector values
  pointers[0] = rP[0];
  fetches[0] = tex1Dfetch(x_tex, pointers[0]);
  pointers[1] = rP[pitch];
  fetches[1] = tex1Dfetch(x_tex, pointers[1]);
  values[0] = cM[0];
  values[1] = cM[pitch];

  cM += 2*pitch;
  rP += 2*pitch;

  y_prod += __fmul_rn(values[0], fetches[0]);
  y_prod += __fmul_rn(values[1], fetches[1]);
 }

 // odd row size
 if (row_size % 2) {
  int pointer = rP[0];
  float fetch = tex1Dfetch(x_tex, pointer);
  float value = cM[0];
  y_prod += __fmul_rn(value, fetch);
 }

 if (beta == 0.0f)
  y[i] = (alpha * y_prod);
 else
```

```
  y[i] = __fmul_rn(beta,y[i])
        +__fmul_rn(alpha, y_prod);
}
```

3 Interfacing with library code

The usage of sparse matrix packages has always been a major issue confronting both the user community and the software designers.

Ideally, each new data storage should be simply pluggable into an existing software framework without any undue effort. This has been obtained by embedding the GPU side data structures in an existing framework based on the Fortran 2003 language.

The basic ideas of the embedding are as follows:

- each sparse matrix has a dual representation, in main memory as well as in the GPU device memory;
- the sparse matrix is usually built in an incremental manner on the CPU side; it is only at the finalization step that the resulting data structure is copied on the GPU device memory;
- the sparse matrix is layered according to the “STATE” design pattern [9], with an outer shell and an inner object whose dynamic type can change to accommodate the needs of the application.

A more detailed discussion of the interfacing issues may be found in [2]. This scheme has also been replicated by devising an interface to the sparse matrix structure available in the CUSPARSE library from the CUDA development toolkit version 4.0 [16]; the principle is the same, but the details are slightly different since the CUDA matrix is based on the CSR storage format.

The major issue in interfacing the inner data structures is the need to avoid as much as possible data movement between the host main memory and the GPU device memory (e.g., see [22]); each data movement entails a substantial overhead that, for the kind of operations we are considering, is capable of overcoming any performance advantage accrued by using the GPU device. This is achieved for the sparse matrix data structures by copying them from main memory to the GPU memory at the time of assembly (i.e., when the storage representation is fixed at the end of the coefficient build loop); the appropriate data structure is passed to the assembly code via a “mold” variable, according to the “prototype” design pattern [9].

```
do i=1, n
   if ( <this index belongs to me> ) then
      nz = <number of entries in equation  i>
      ia(1:nz) = i
      ja(1:nz) = <list of neighbours of i>
      val(1:nz) = <coefficients Aij >
      call spins(nz,ia,ja,val,a,desc_a,info)
   endif
enddo
call spasb(a,desc_a,info, mold=aellg)
```

A similar strategy has been also applied for the vectors involved in the matrix-vector product and in the other operations typical of the sparse linear system solvers commonly in use. Each vector object has two images, one in main memory and one in the GPU memory. Whenever an operation is invoked that involves these objects, the code keeps track of whether the GPU memory holds an up-to-date copy of the data; operations on the vectors are executed preferentially on the GPU side, and the GPU acts as a sort of “magnet” in that the data remains there until the user explicitly requires a synchronization with the CPU main memory (e.g., by requesting a copy to a normal array). Thus, we have a kind of “on demand” placement of data that acts in a completely transparent manner, without changing the linear solver code. The relevant methods are the `is_host()` method which signals that the up-to-date copy of the vector is in the host memory, the `set_dev()` method which signals that an operation occurred such that the up-to-date copy is the one on the device side, and the `sync()` method which copies the host image onto the device, or viceversa, as necessary to make sure the two copies of the data are synchronized.

As an example, the code for the matrix-vector product is (slightly simplified) as follows:

```
subroutine d_elg_vect_mv(alpha,a,x,beta, &
  & y,info,trans)
  use iso_c_binding
  implicit none
  class(d_elg_sparse_mat), intent(in) :: a
  real(dpk_), intent(in)    :: alpha, beta
  class(d_vect), intent(inout) :: x
  class(d_vect), intent(inout) :: y
  integer, intent(out)         :: info
  character, optional, intent(in) :: trans

  if (x%is_host()) call x%sync()
  if (beta /= dzero) then
    if (y%is_host()) call y%sync()
  end if
  info = spmvEllDevice(a%deviceMat,&
         & alpha,x%deviceVect,&
         & beta,y%deviceVect)
  call y%set_dev()
```

With the appropriate support we can then have a sparse linear solver such as Conjugate Gradient (CG) to run unchanged on the GPU, by simply feeding the appropriate data objects:

```
call psb_geaxpby(one,b,zero,r,desc_a)
call psb_spmm(-one,A,x,one,r,desc_a)
rho = zero
iterate: do  it = 1, itmax
   call prec%apply(r,w,desc_a)
   rho_old = rho
   rho = psb_gedot(r,z,desc_a)
   if (it == 1) then
      call psb_geaxpby(one,z,zero,p,desc_a)
   else
      beta = rho/rho_old
      call psb_geaxpby(one,z,beta,p,desc_a)
   endif
   call psb_spmm(one,A,p,zero,q,desc_a)
   sigma = psb_gedot(p,q,desc_a)
   alpha = rho/sigma
   call psb_geaxpby(alpha,p,one,x,desc_a)
   call psb_geaxpby(-alpha,q,one,r,desc_a)
   rn2 = psb_genrm2(r,desc_a)
   bn2 = psb_genrm2(b,desc_a)
   err = rn2/bn2
   if (err.lt.eps) exit iterate
enddo iterate
call x%sync()
```

4 Experimental results

To test the effectiveness of our approach we have selected a set of test cases and we have run experiments with it on two platforms. The first platform is a desktop machine installed at our university; it is equipped with an AMD Athlon^{TM}7750 Dual-Core Processor, and an NVIDA GTX-285 graphics card. The second platform is a cluster of the CASPUR supercomputing center, each node of which hosts an Intel Xeon X5650, with NVIDIA Tesla C2050 graphics card. In the following, the two platforms are referred to as GTX 285 and C2050, respectively.

Card model	Clock	Bus speed	Bus width	Peak DP
GTX 285	648 MHz	159 GB/s	512 b	90 GFLOPS
C2050	575 MHz	144 GB/s	348 b	515 GFLOPS

Table 1: The GPU models employed

matrix	N	NZ	MFLOPS			Speedup
			CPU	xGPU	GPU	
pde05	125	725	414	21	55	0.13
pde10	1000	6400	723	154	481	0.67
pde20	8000	53600	794	624	3610	4.55
pde30	27000	183600	545	881	7784	14.27
pde40	64000	438400	439	791	10380	23.63
pde50	125000	860000	436	848	12452	28.54
pde60	216000	1490400	444	981	13842	31.18
pde80	512000	3545600	454	1162	15250	33.63
pde90	729000	5054400	449	1223	15354	34.20
pde100	1000000	6940000	443	1273	15664	35.35

Table 2: ELLG matrix-vector product performance on platform 1 — GTX–285

On both platforms we have compiled the inner GPU computational kernels with the CUDA toolkit version 4.0, whereas the outer layers were compiled with the GNU Compiler Collection 4.6.1, employing the Fortran, C and C++ drivers for various parts of the system. To position our work we have also considered the CUSPARSE library that is part of the CUDA toolkit version 4.0 [16]; it proposes a storage format that is essentially identical to the CSR format (in the following, referred to as CSRG). We have wrapped the GPU side kernels in a manner similar to our own ELLG kernels, so as to be able to use exactly the same test programs; therefore all overhead factors are the same for both sets of kernels. All experiments have been run in double precision, as this is usually necessary for the problems that are typically encountered in contemporary scientific applications.

The main features of the two graphics cards are shown in Table 1; for the interpretation of our subsequent results it is very important to look carefully at the various performance metrics. In particular, the Tesla C2050 has a much better execution rate for double precision floating point arithmetic; however, and somewhat counterintuitively, it also shows a lower clock rate and a lower bus speed. As we shall see, since our sparse computations are latency and memory bound, the peak performance indicator is not necessarily accurate, and indeed the older GTX-285 card actually does outperform the newer C2050.

Table 2 shows the performance of the matrix-vector product in MFLOPS for the first platform. The matrices have been generated from the finite difference discretization of Partial Differential Equation (PDE) on a cubic domain with a uniform step size; the name of the matrix in the first column refers to the length of the cube edge in the discrete units. The second column reports the number of rows and columns in the matrix, whereas the third column the number of non-zero elements; most rows in each matrix contain 7 non-zero elements. The column labeled CPU shows the performance of the matrix-vector product as implemented on the CPU with the CSR storage format; the column labeled xGPU is the performance with the product executed on the GPU but including the overhead of transferring the x and y vectors from main memory to the device; the GPU column shows the performance data when the x and y vectors are in the GPU memory from the beginning, and finally we also list a speedup of the GPU vs the CPU computation. The measurements were obtained by performing 2000 products in sequence, and computing the overall execution rate. Table 3 is built in exactly the same way, but the data have been gathered on the second platform.

As we can see, in both cases for the CPU column there is a surge in performance at low to medium sizes, then followed by a levelling off for large matrix cases; this is due to the memory occupation of the matrix overflowing the cache memory, which is larger for platform 2. Performing the matrix-vector products in a sequence, with a warm cache, is consistent with the usage within the iterative linear system solvers which are our target applications, where one or more products are performed within each iteration. The CPU performance is better for platform 2; however exactly the opposite is true for the GPU performance; since the sparse matrix-vector computation is bound by the accesses to

matrix	N	NZ	MFLOPS			Speedup
			CPU	xGPU	GPU	
pde05	125	725	542	6	28	0.05
pde10	1000	6400	959	61	187	0.20
pde20	8000	53600	1091	370	2147	1.97
pde30	27000	183600	1018	791	4319	4.25
pde40	64000	438400	1127	907	6394	5.67
pde50	125000	860000	1059	1098	8237	7.78
pde60	216000	1490400	875	1404	9442	10.80
pde80	512000	3545600	861	1746	10396	12.08
pde90	729000	5054400	844	1821	10812	12.81
pde100	1000000	6940000	867	1924	10859	12.52

Table 3: ELLG matrix-vector product performance on platform 2 — C2050

matrix	N	NZ	MFLOPS			Speedup
			CPU	xGPU	GPU	
pde05	125	725	402	20	40	0.10
pde10	1000	6400	733	149	434	0.59
pde20	8000	53600	782	454	1807	2.31
pde30	27000	183600	564	665	2562	4.55
pde40	64000	438400	446	601	2898	6.49
pde50	125000	860000	436	697	3035	6.97
pde60	216000	1490400	454	827	3135	6.91
pde80	512000	3545600	451	900	3220	7.14
pde90	729000	5054400	454	939	3232	7.12
pde100	1000000	6940000	445	957	3175	7.14

Table 4: CSRG Matrix-vector product performance on platform 1 — GTX–285

the memory subsystem, the superior floating point performance of the C2050 platform does not compensate for the higher clock rate and bus speed of the GTX-285 platform. In both cases the sparse kernel execution rate is much lower than the peak execution rate, which is known to be achievable with a good approximation when running kernels for dense matrices, such as the matrix-matrix product DGEMM.

Tables 4 and 5 report the results of the same measurements, with the same program but using the data structures and inner kernels from the CUSPARSE library. As we can see, we have a very different behaviour, in that:

1. the attained GPU performance is significantly lower in both cases;
2. platform 2 (the Tesla C2050) is slightly better than platform 1 (the GTX-285).

Especially in the GTX-285 case, the regularity of memory accesses in our data structure as opposed to the more irregular accesses in memory that are intrinsic to CSR makes a large difference in the attained performance; this is still true on the C2050, albeit to a much lesser extent, since the underlying Fermi architecture is substantially different from that of the older card, and in particular it suffers much less from uncoalesced accesses to memory.

5 Conclusions

In this paper, we have discussed the implementation issues of the sparse matrix by dense vector product kernel on GPU devices, specifically on the NVIDIA GPUs. We have presented the ELLG format, which is derived from the ELLPACK format and properly adapted to a GPU implementation, and described how the proposed data storage can be simply plugged into existing software frameworks for parallel sparse applications. We have shown that how our

matrix	N	NZ	MFLOPS			Speedup
			CPU	xGPU	GPU	
pde05	125	725	556	7	30	0.05
pde10	1000	6400	933	58	178	0.19
pde20	8000	53600	1102	340	1588	1.44
pde30	27000	183600	1020	702	2923	2.87
pde40	64000	438400	1033	794	3833	3.71
pde50	125000	860000	967	953	4094	4.23
pde60	216000	1490400	863	1175	4431	5.14
pde80	512000	3545600	851	1408	4628	5.44
pde90	729000	5054400	842	1470	4618	5.48
pde100	1000000	6940000	858	1518	4640	5.41

Table 5: CSRG Matrix-vector product performance on platform 2 — C2050

data structures allow us to achieve very good performance results, significantly better than what can be obtained with the CUSPARSE library version 4.0.

The overall software infrastructure in which this work is embedded has been described in other papers [X,Y], and is an ongoing effort. Since the final objective is to enable applications to run on the GPU, future work will include:

- Further tuning of the kernels;
- Testing in the context of a complete iterative solver;
- Studying the various possibilities for implementing suitable preconditioners for the Krylov methods;
- Exploring the possibility of interfacing our software from high-level environment such as MATLAB or Octave;
- Extending our GPU-enablement to the full MPI version of our software.

Much work lies still ahead of us; moreover, a lot of work will be needed to keep pace with the changes in the architectural landscape. This is very reminiscent of the situation during the various development cycles in High Performance Computing platforms, first with vector computers from the early 70s into the 90s, then with RISC machines and clusters from the late 80s onwards.

Acknowledgments

We wish to acknowledge the organization of the CASPUR supercomputing center for awarding us with a grant for access to the experimental platform 2 under the HPC-Grant 2011 program.

References

[1] D. Barbieri, V. Cardellini, and S. Filippone. Generalized GEMM applications on GPGPUs: Experiments and applications. In *Proc. of 2009 Int'l Conf. on Parallel Computing*, ParCo '09. IOS Press, 2009.

[2] D. Barbieri, V. Cardellini, S. Filippone, and D. Rouson. Design Patterns for Scientific Computations. In *Europar 2011*, Bordeaux, France, 2011.

[3] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. Technical Report RC24704, IBM Research, Apr. 2009.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. of Int'l Conf. on High Performance Computing Networking, Storage and Analysis*, SC '09. ACM, 2009.

[5] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *SIGPLAN Not.*, 45:115–126, Jan. 2010.

[6] P. D'Ambra, D. di Serafino, and S. Filippone. MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. *ACM Trans. Math. Softw.*, 37(3):7–23, Sept. 2010.

[7] I. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: a user level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, Sept. 1997.

[8] S. Filippone and M. Colajanni. PSBLAS: a library for parallel linear algebra computations on sparse matrices. *ACM Trans. on Math Software*, 26:527–550, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] W.-M. Hwu. *GPU computing gems emerald edition*. Morgan Kaufmann, first edition, 2011.

[11] D. R. Kincaid, T. C. Oppe, and D. M. Young. ITPACKV 2D Users Guide. `http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/`.

[12] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. GPGPU: general-purpose computation on graphics hardware. In *Proc. of 2006 ACM/IEEE Conf. on Supercomputing*, SC '06. ACM, 2006.

[13] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers*, volume 5952 of *LNCS*, pages 111–125. Springer, 2010.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6:40–53, March 2008.

[15] NVIDIA Corp. CUDPP: CUDA data parallel primitives library. `http://gpgpu.org/developer/cudpp/`.

[16] NVIDIA Corp. CUDA CUSPARSE library version 4.0, 2011. `http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUSPARSE_Library.pdf`.

[17] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez. Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs. *Microprocessors and Microsystems*, In Press, 2011.

[18] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

[19] J. Sanders and E. Kandrot. *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley, first edition, 2010.

[20] F. Vazquez, G. Ortega, J. J. Fernández, and E. M. Garzon. Improving the performance of the sparse matrix vector product with GPUs. In *Proc. of 10th IEEE Int'l Conf. on Computer and Information Technology*, pages 1146–1151, 2010.

[21] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proc. of 2008 ACM/IEEE Conf. on Supercomputing*, SC '08. IEEE Press, 2008.

[22] R. Vuduc, A. Chandramowlishwaran, J. W. Choi, M. E. Guney, and A. Shringarpure. On the limits of GPU acceleration. In *Proc. of 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar '10)*, June 2010.