Design Patterns for Scientific Computations on Sparse Matrices

Davide Barbieri[1], Valeria Cardellini[1],
Salvatore Filippone[1], and Damian Rouson[2]

[1] University of Rome "Tor Vergata", Italy
salvatore.filippone@uniroma2.it, cardellini@ing.uniroma2.it
[2] Sandia National Laboratories
rouson@sandia.gov

Abstract. We discuss object-oriented software design patterns in the context of scientific computations on sparse matrices. Design patterns arise when multiple independent development efforts produce very similar designs, yielding an evolutionary convergence onto a good solution: a flexible, maintainable, high-performance design. We demonstrate how to engender these traits by implementing an interface for sparse matrix computations on NVIDIA GPUs starting from an existing sparse matrix library. We also present initial performance results.

1 Introduction

Computational scientists concern themselves chiefly with producing science, even when a significant percentage of their time goes to engineering software. The majority of professional software engineers, by contrast, concern themselves with non-scientific software. In this paper, we demonstrate the fruitful results of bringing these two fields together by applying a branch of modern software engineering design to the development of scientific programs.

Our meeting ground is the field of sparse matrices and related computations, one of the centerpieces of scientific computing. This paper covers how to handle certain kinds of design requirements, and illustrates what can be done by consciously applying certain design techniques. Specifically, we discuss the benefits accrued by the application of the widely used software engineering concept of *design patterns* in the context of scientific computation on sparse matrices. We choose as a case study the implementation of an interface for sparse matrix computations on NVIDIA GPUs starting from an existing sparse library.

A number of related projects provide libraries for constructing and using sparse (and dense) matrices and vectors, as well as provide solver libraries for linear, nonlinear, time-dependent, and eigenvalue problems. These projects include Trilinos [11], PETSc [1], and PSBLAS [8,7].

Trilinos focuses on the development of algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems. PETSc is the Portable,

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 367–376, 2012.

Extensible Toolkit for Scientific Computation. Many of the algorithms in PETSc and Trilinos can be interchanged via abstract interfaces without impacting application code. Both projects employ MPI to exploit distributed-memory, parallel computers and provide sparse matrix solvers for linear, nonlinear, and eigenvalue problems. They differ in implementation language: PETSc is written in C whereas Trilinos is written in C++. Language differences ultimately influence the programming paradigm and architectural style, with C supporting procedural programming and C++ explicitly enabling an object-oriented programming (OOP) style that facilitates the adoption of the architectural design patterns that comprise the focus of the current paper.

Parallel Sparse BLAS (PSBLAS) is a library of Basic Linear Algebra Subroutines for parallel sparse applications that facilitates the porting of complex computations on multicomputers. PSBLAS includes routines for multiplying sparse matrices by dense matrices, solving sparse triangular systems, and preprocessing sparse matrices; the library is mostly implemented in Fortran 95, with some additions of Fortran 77 and C. A Fortran 2003 version is currently under development, and forms the basis for the examples in this paper, because of the language support that we are going to describe.

Sparse matrices are widely used in scientific computations; most physical problems modeled by partial differential equations (PDEs) are solved via discretizations that transform the original equations into a linear system and/or an eigenvalue problem with a sparse coefficient matrix. A matrix is sparse when most of its elements are zero; this fact is exploited in devising a representation that does not store explicitly the null coefficients. This means abandoning the language's native array type along with the underlying assumption that one can infer the indices (i, j) associated with an element a_{ij} from the element's position in memory and vice versa. Any viable replacement for these assumptions must involve storing the indices explicitly. Despite the resulting overhead, in the vast majority of applications the scheme pays off nicely due to the small number of nonzero elements per row.

Variations on this concept abound in the COOrdinate, Compressed Sparse Rows, Compressed Sparse Columns, ELLpack, JAgged Diagonals, and other formats. Each storage format offers different efficiencies with respect to the mathematical operator or data transformation to be implemented (both typically map into an object "method"), and the underlying platform (including the hardware architecture and the compiler).

Sect. 2 of the current paper presents three design patterns: State, Builder, and Prototype, leveraging the newly available OOP constructs of Fortran 2003 in scientific applications. Sect. 3 presents interfaces for sparse matrix computations on GPUs starting from PSBLAS. Sect. 4 concludes.

2 Design Patterns

Many professionals will confirm that, when confronted with design patterns, their colleagues will often have a "recognition" moment in which they declare they have been doing things "the right way" all along, without knowing their fancy names.

Applying design patterns in a conscious way can be highly beneficial. Evidence from the literature suggests that these benefits have been reaped in the context of scientific applications only recently [10,14], the timing being due in part to the arrival of compilers that support the OOP constructs in Fortran 2003, the only language for which the international standards body has scientific programmers as its target audience. With this paper, we discuss implementations of design patterns not previously demonstrated in Fortran 2003.

2.1 "STATE" Is Your Friend

The State pattern allows the encapsulation of object state behind an interface that allows the object type to vary at runtime. Figure 1 shows a Unified Modeling Language (UML) class diagram of the State pattern, including the class relationships and the public methods. The diagram hides the private attributes.

Let us consider the problem of switching among different storage formats for a given object. An old-fashioned but feasible solution would be to have a data structure containing integer values driving the interpretation and dispatching of the various operations; however, this route is not very maintainable and scalable. Using an object-oriented design and language per se is not a solution either; indeed, while it seems that all variations in storage formats could be derived from a base class, switching at runtime would require the same object to change its class dynamically. This is generally not supported by object-oriented languages (with very rare exceptions [9]); the solution is to add a layer of indirection, encapsulating the "dynamic" object inside a normal one. The application to the sparse matrix case is shown here:

```
type :: psb_d_base_sparse_mat
contains
  procedure, pass(a) :: foo
end type psb_d_base_sparse_mat

type :: psb_dspmat_type
  class(psb_d_base_sparse_mat), allocatable  :: a
contains
  procedure, pass(a) :: mat_foo
end type psb_dspmat_type

subroutine mat_foo(a)
    call a%a%foo()
end subroutine mat_foo
```

The methods of the outer class are always thrown onto the inner class, which is the actual workhorse. To enable a runtime class switch it is necessary to devise a conversion strategy; a viable choice we employed is to have one reference storage class, and to have all other classes provide methods to convert to/from it.

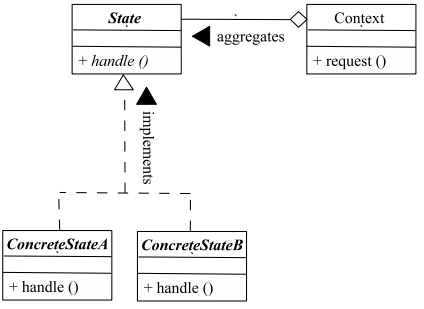


Fig. 1. UML class diagram for the State pattern: classes (boxes) and relationships (connecting lines), including abstract classes (bold italics) and relationship adornments (line labels) with arrows indicating the relationship direction. Line endings indicate relationship type: inheritance (open triangle) and aggregation (open diamond). Class boxes include: name (top), attributes (middle), and methods (bottom). Leading signs indicate visibility: public (+) or private (−). Italics denote an abstract method.

2.2 "PROTOTYPE" and "BUILDER" Are Good Ideas

This section is concerned with maintaining and extending a body of software, and how certain patterns can help. Suppose you are designing a library for sparse matrix computations; you spend a long time in thinking about the capabilities you have to implement, and how to combine them in a way that is both efficient and flexible. You have also spent a significant amount of effort in properly segregating the specifics of any given storage format to its class, and in optimizing the implementation of its methods. This is a success, everything works properly, and publication ensues; so far, so good.

However, at this point two issues arise: (1) your software has to be used on the latest BNE Tour-de-France processor; (2) Professor Hook in the University of Neverland absolutely wants to fit her favourite storage method into your framework, since she thinks it is so good (both her method and the framework). If your software is really successful, these requests might be coming in with an alarming frequency. Each time you have to derive a new class for the inner storage object (remember, we are systematically using the STATE pattern), and this is the (relatively) easy part, but you also have to adapt the library to handle its existence. You have to add constructors, converters, and what not; potentially, you have to touch multiple places, and break multiple things. How do you get out of this? The strategy that was devised in PSBLAS can be interpreted in terms of two design patterns: Builder and Prototype.

BUILDER. The Builder pattern in OO design allows for an abstract specification of the steps required to construct an object. Figure 2 shows a UML class diagram of the Builder pattern, including the class relationships and the public methods of the abstract parent. Child classes must provide concrete implementations of these methods (not shown). The diagram hides the private attributes.

The strategy to build a sparse matrix is: (1) initialize to some default; (2) add sets of coefficients by calling buildup methods in a loop; (3) assemble the results and bring the object to the desired final storage status. Most sparse matrix libraries (including Trilinos, PETSc, PSBLAS) are organized around these concepts; this is an example of "convergent evolution" towards a reasonable solution that is more or less forced by the constraints of the application domain.

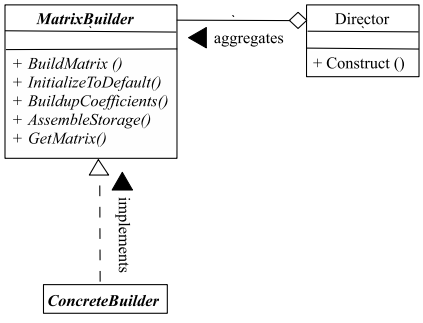


Fig. 2. UML class diagram for the Builder pattern

It should be clear that the only place where the desired output storage format has to be enforced explicitly is at the assembly step; in PSBLAS software, this is handled at the inner level, by allocating a new object of the desired class and converting to it from the existing inner object. It may appear that it is necessary to have an exact knowledge of the derived class of the new object at the time the assembly code is written; however this is not quite true. All that is needed is to know that it is derived from a given base class, and that it is capable of converting to/from a reference storage format. With this scheme, a conversion between arbitrary derived classes can always be implemented by at most one intermediate object of the reference class, even if the outer code invoking the conversion methods does not know the exact dynamic type involved. This scheme works fine, provided that the library code can allocate a new object with the correct dynamic type, which is only known at runtime: to this end we call the next design pattern to our rescue.

PROTOTYPE. This design pattern might also be defined as "copy by example": when a method needs to instantiate an object, it does it by referring to another object which is a "source" or a "mold". The class for copied object will include a cloning or molding method by which the desired copy can be obtained; the two variations refer to whether a full copy of the source object is created, or just an empty copy with the correct dynamic type.

Returning to our example of assembling a sparse matrix, you (or rather, your library code) gets a reference object for the inner storage; you do not need to know the details of its contents, as long as it is an extension of the base storage class, and it implements the necessary conversion methods to/from the reference format.

This idea is so good that in Fortran 2003/2008 it has become part of the language itself. To call into existence a polymorphic object the language provides a specification of dynamic type in the `ALLOCATE` statement; the most common way is shown in the following example:

```
class(base_sparse_mat), allocatable : mat_object
allocate(my_storage_format : mat_object)
```

where `my_storage_format` is the name of the desired dynamic type. However it is also possible to use the following alternatives:

```
class(base_sparse_mat) :: sourcemat;
allocate(mat_object, source=sourcemat)
```

or alternatively

```
allocate(mat_object, mold=sourcemat)
```

depending on whether the original contents of `sourcemat` have to be copied or not. The `MOLD=` variant is extensively used in PSBLAS to implement the PROTOTYPE pattern. In this way, a new storage format can be added by (1) defining a derived class from the base class, providing the necessary implementations of the methods; (2) using the new class in the main application, declaring a variable of the desired new class; (3) passing the "mold" variable to the assembly routine. This is it; no changes are necessary to the library code, not even a recompilation, and the existing computational methods will happily use the new storage format.

3 An Example: Adding Support for NVIDIA GPUs

Graphics Processing Units (GPUs) have 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). We discuss here how our desing techniques help in interfacing sparse matrix computations kernels for the GPU into the existing sparse library PSBLAS.

The NVIDIA GPU architectural model is based on a scalable array of multithreaded 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. CUDA is the programming model provided by NVIDIA for its GPUs; a CUDA program consists of a host program that runs on the CPU host, and a kernel program that executes on the GPU device. 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. A key component of CUDA is the GPU memory hierarchy.

Memory on the GPU includes a global memory area in a shared address space accessible by the CPU and by all threads, a low-latency memory called the shared memory, which is local to each multiprocessor, and a per-thread private local memory, not directly available to the programmer.

3.1 Sparse Matrix Computation on GPU

The considerable interest in GPUs for General Purpose computation (GPGPU) is due to the significant performance benefits possible with its usage; for example, the works in [2,3,16] demonstrated how to achieve significant percentages of peak single-precision and double-precision throughput in dense linear algebra kernels. It is thus natural that GPUs (and their SIMD architecture) are considered for implementing sparse matrix computations; sparse matrix-vector multiplication has been the subject of intense research on every generation of high performance computing architectures.

Sparse matrix computations on the GPU introduce additional challenges with respect to their dense counterparts, because operations on them are typically much less regular in their data access patterns; recent efforts on sparse GPU codes include [4,5,6], and NVIDIA's CUSPARSE library [13].

Let us consider the matrix-vector multiplication $y \leftarrow \alpha Ax + \beta y$ where A is large and sparse and x and y are column vectors; we will need to devise a specific storage format for the matrix A to implement the sparse matrix computations of interest. Our starting point is a GPU-friendly format we developed; we will concentrate on how the design patterns discussed in Sect. 2 can be used to plug in the new formats and the GPU support code in the PSBLAS library.

Our storage format is a variation of the standard ELLpack (or ELL)format; an M-by-N sparse matrix with at most K nonzeros per row is stored as a dense M-by-K array `data` of nonzeros and array `indices` of column indices; all rows are zero-padded to length K; this format is efficient when the maximum number of nonzeros per row is close to the average. ELL fits a sparse matrix in a regular data structure; thus it is a good candidate to implement sparse matrix operations on SIMT architectures. The usage of ELL format and its variants on GPUs have been previously analyzed in [15,16].

Our storage format ELL-G takes into account the memory access patterns of the NVIDIA Tesla architecture [12], as well as other *many-threads* performance optimization patterns of the CUDA programming model.

A critical issue in interfacing with existing codes is the need to take care of the data movements from the main memory to the GPU RAM and vice versa; unfortunately data movement is very slow compared to the high bandwidth internal to the GPU, and this is one of the major problems in GPU programming. To add support for NVIDIA GPUs in the PSBLAS library we had to derive from ELL a new GPU class requiring the following modifications:

- At assembly time, copy the matrix to the GPU memory;
- At matrix-vector time, invoke the code on the GPU side;
- At deallocation time, release memory on both the host and device sides.

On the library side of things, a set of wrappers handles the communication with the application and sorts out the needed inter-language call details; attached to this layer there is the CUDA layer which performs the actual work. Thus, given the preparatory work discussed above, we can have the code

```
call psb_spmm(-done,a,x,dzero,y,desc_a,info,'n')
```

which performs the matrix-vector product; according to the dynamic type and state of the inner component(s) of `a`, `x` and `y` the code will run on the CPU or on the GPU, possibly including copying the vector data to the GPU side.

3.2 Performance Results

First of all, since this paper is dedicated to the design technique, let us state that after writing the CUDA kernel code, embedding the new format in the existing library required a development effort of just about a couple of days, including debugging. Our computational experiments were run on an NVIDIA GeForce GTX 285 graphics card, which has a maximum throughput of 94.8 Gflop/s in double precision. The computation rates are reported in Gflop/s; the number of arithmetic operations per matrix-vector is assumed to be $2NZ$ where NZ is the number of nonzeros in the matrix, and the rate is averaged over multiple runs. For the experiments we used a collection of sparse matrices arising from

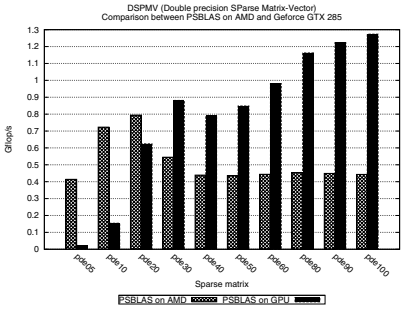
Table 1. Sparse matrices used in the performance experiments

matrix name	N	NZ
pde05	125	725
pde10	1000	6400
pde20	8000	53600
pde30	27000	183600
pde40	64000	438400

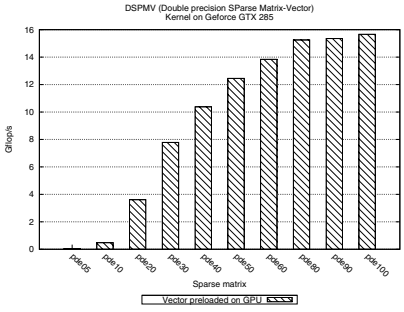
matrix name	N	NZ
pde50	125000	860000
pde60	216000	1490400
pde80	512000	3545600
pde90	729000	5054400
pde100	1000000	6940000

a test three-dimensional partial differential equation (PDE) problem; the PDE is discretized with finite differences on a cubic domain. Table 1 summarizes the matrix characteristics, where N is the matrix size and NZ is the number of nonzeros. For the matrix-vector multiplication, in the experiments we set $\beta = 0$, i.e., we consider $y \leftarrow Ax$, and report results only for double precision computations.

In our experiments we compare the throughput of the sparse matrix-vector multiplication in PSBLAS exploiting the GPU and using our ELL-G storage format with that obtained by the standard PSBLAS library on CPU. For the experiments on CPU we used an AMD Athlon 64 processor running at 2.7 GHz; this is a dual-core processor, but we only run in serial mode for the purposes of this comparison. Figure 3(a) shows the performance improvement that we obtain implementing the PSBLAS interface for sparse matrix computations on GPUs even when we include the overhead of transferring the vector data from main memory; in this case the GTX 285 vs AMD matrix-vector multiplication gives a speedup between 2 and 3 depending on the sparse matrix.



(a) Throughput comparison of PSBLAS on GPU (including vector copy-in overhead) and on CPU (double precision).



(b) Throughput of the sparse matrix-vector multiplication kernel on GPU (without vector copy-in overhead).

Fig. 3.

With the measurements shown in Figure 3(b) we report the same operations on the same data, but with the vectors prearranged in the GPU memory; this is more representative of usage of the kernels in a sparse iterative solver. Comparing these results with those in Figure 3(a), we see that data transfer overhead is very significant; having the vectors on the GPU enables a performance level that is essentially identical to that of the inner kernels. Arranging the vectors to be loaded on the GPU device memory is possible because the vectors undergo the same build cycle as the matrices; therefore by employing the State pattern for vectors we can have the data loaded on the device side "on demand". The high-level solver code looks exactly the same for GPU and CPU execution, but during the solution process only scalars are transferred between the CPU and the GPU; the solution vector itself is recovered upon exit from the iterative process.

4 Conclusions

In this paper, we have discussed how the well-known software engineering concept of Design Patterns can be applied with benefits in the context of sparse matrix computation. We have demonstrated how these patterns can be used to implement an interface for sparse matrix computations on GPUs starting from the existing PSBLAS library. Our experience shows that this solution provides good flexibility and maintainability and allows to exploit the GPU computation with its related performance benefits. While the ideas discussed have been tested in the PSBLAS framework, future work will include extension to multilevel preconditioners as well as interfacing with ForTrilinos. The techniques described in this paper can also be employed to encapsulate and use other storage formats, including the format used in the CUSPARSE library; a detailed performance analysis and comparison is the subject of currently ongoing work.

Acknowledgments. This research has been partially supported by the Italian Ministry of Instruction, University, and Research within the project FIRB 2007 "Studio, progettazione e sviluppo e sperimentazione di una nuova generazione competitiva di motori innovativi a basso consumo ed a basso impatto ambientale nellarco dellintero ciclo di vita."

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

1. Balay, S., Gropp, W., McInnes, L.C., Smith, B.: PETSc 2.0 user manual. Tech. Rep. ANL-95/11 - Revision 2.0.22, Argonne National Laboratory (1995)
2. Barbieri, D., Cardellini, V., Filippone, S.: Generalized GEMM applications on GPGPUs: Experiments and applications. In: ParCo 2009. IOS Press (2009)
3. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S., Quintana-Ortí, G.: Exploiting the capabilities of modern gpus for dense matrix computations. Concurr. Comput.: Pract. Exper. 21, 2457–2477 (2009)
4. Baskaran, M.M., Bordawekar, R.: Optimizing sparse matrix-vector multiplication on GPUs. Tech. Rep. RC24704, IBM Research (April 2009)
5. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Supercomputing 2009. ACM (2009)
6. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. SIGPLAN Not. 45, 115–126 (2010)
7. D'Ambra, P., di Serafino, D., Filippone, S.: MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. ACM Trans. Math. Softw. 37(3) (2010)
8. Filippone, S., Colajanni, M.: PSBLAS: a library for parallel linear algebra computations on sparse matrices. ACM Trans. on Math Software 26, 527–550 (2000)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
10. Gardner, H., Manduchi, G.: Design Patterns for e-Science. Springer (2007)
11. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An overview of the Trilinos project. ACM Trans. Math. Softw. 31(3), 397–423 (2005)
12. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro. 28, 39–55 (2008)
13. NVIDIA Corp.: CUDA CUSPARSE library version 4.0 (2011)
14. Rouson, D.W.I., Xia, J., Xu, X.: Scientific Software Design: The Object-Oriented Way. Cambridge University Press (2011)
15. Vazquez, F., Ortega, G., Fernández, J.J., Garzon, E.M.: Improving the performance of the sparse matrix vector product with GPUs. In: CIT 2010, pp. 1146–1151 (2010)
16. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Supercomputing 2008 (2008)