Scientific Programming 22 (2014) 1–19
DOI 10.3233/SPR-130363
IOS Press

Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms [1]

Valeria Cardellini [a], Salvatore Filippone [a,*] and Damian W.I. Rouson [b]
[a] *Università di Roma "Tor Vergata", Roma, Italy*
E-mails: cardellini@ing.uniroma2.it, salvatore.filippone@uniroma2.it
[b] *Stanford University, Stanford, CA, USA*
E-mail: rouson@stanford.edu

Abstract. We apply object-oriented software design patterns to develop code for scientific software involving sparse matrices. Design patterns arise when multiple independent developments produce similar designs which converge onto a generic solution. We demonstrate how to use design patterns to implement an interface for sparse matrix computations on NVIDIA GPUs starting from PSBLAS, an existing sparse matrix library, and from existing sets of GPU kernels for sparse matrices. We also compare the throughput of the PSBLAS sparse matrix–vector multiplication on two platforms exploiting the GPU with that obtained by a CPU-only PSBLAS implementation. Our experiments exhibit encouraging results regarding the comparison between CPU and GPU executions in double precision, obtaining a speedup of up to 35.35 on NVIDIA GTX 285 with respect to AMD Athlon 7750, and up to 10.15 on NVIDIA Tesla C2050 with respect to Intel Xeon X5650.

Keywords: Design patterns, sparse matrices, GPGPU computing

AUTHOR COPY

Glossary

aggregation A "has a" or "whole/part" relationship between classes wherein an instance of the class representing the "whole" encapsulates (has) one or more instances of the class representing the part(s). Fortran supports aggregation via the components of derived types.

class An extensible type that encapsulates data and procedures for operating on those data. Fortran supports classes via extensible derived types. An instance of an extensible derived type can be declared with the keyword "class" if it is a subprogram dummy argument or if it has either allocatable or pointer attribute.

composition A special case of aggregation wherein the lifetimes (from construction to destruction) of the whole and part coincide.

derived type A user-defined Fortran type. Fortran user-defined types are by default extensible.

design pattern A common solution to a recurring problem in software design.

dynamic polymorphism A run-time technology in which the compiler resolves the invocation of a type-bound procedure name into one of many actual procedure names based on the dynamic type of the object on which the procedure is invoked. The dynamic type may be either the type named in the declaration of the object or any type that extends the named type.

encapsulation Bundling data of various intrinsic or programmer-defined types together with procedures for operating on those data. Fortran supports encapsulation via the definition of derived-type components and type-bound procedure.

information hiding Limiting access to data or procedures. Fortran supports information hiding through the "private" attribute, the application of which limits the accessibility of data and procedures to code in the same module where the data or procedures are defined.

[1] A preliminary version of this paper has been presented at the HPSS workshop in the EuroPar 2011 conference [4].

[*] Corresponding author: Salvatore Filippone, Dipartimento di Ingegneria Industriale, Università di Roma "Tor Vergata", Viale del Politecnico 1, 00133 Roma, Italy. Tel.: +39 06 72597558; Fax: +39 06 72597158; E-mail: salvatore.filippone@uniroma2.it.

inheritance An special case of composition which a child class supports procedure invocations as its parent class either by delegating calls to the parent's implementations of those procedures or by overriding the parents' procedures. Inheritance often referred to as an "is a" relationship or as subclassing. OOP languages support inheritance by automating the insertion of the parent into the child and by automating the procedural delegations. Fortran supports inheritance via type extension.

instantiate To construct an object as an instance of a class. This typically involves allocating any resources the object requires (e.g., memory) and initializing the object's data. A mechanism for designating that one derived type extends another and thereby inherits the components and type-bound procedures of the latter type.

object-oriented design Schematic descriptions of software structure and behavior stressing the relationships between data structures and between data structures and the procedures for operating on the data structures.

object-oriented programming A computer program construction philosophy wherein the programmer couples discrete packages of data with a set of procedures for manipulating those data, emphasizing four basic principles: encapsulation, information-hiding, polymorphism and inheritance.

polymorphism The ability for one named class or one named procedure to reference many different classes or procedures at runtime.

software design A schematic description of the organization and behavior of a computer program.

static polymorphism A compile-time technology in which the compiler resolves the invocation of a generic procedure name into one of many actual procedure names based on the type, kind, and rank of the procedure's arguments at the point of invocation.

type extension A mechanism for designating that one derived type extends another and thereby inherits the components and type-bound procedures of the latter type.

warp A group of threads in a CUDA-enabled device, treated as a unit by the scheduler.

Acronyms

2D	Two-Dimensional.
ADSM	Asymmetric Distributed Shared Memory.
COO	COOrdinate.
CPU	Central Processing Unit.
CSC	Compressed Sparse Columns.
CSR	Compressed Storage by Rows.
CUDA	Compute Unified Device Architecture.
ECC	Error-Correcting Code.
ELL	ELLpack.
FLAME	the Formal Linear Algebra Methods Environment.
GFLOPS	Giga (billions) of Floating-point Operations Per Second, or GigaFLOPS.
GPGPU	General-Purpose Graphics Processing Unit.
GPU	Graphics Processing Unit.
JAD	JAgged Diagonals.
LOC	Lines Of Code.
OOD	Object-Oriented Design.
OOP	Object-Oriented Programming.
PDE	Partial Differential Equation.
PETSc	the Portable Extensible Toolkit for Scientific Computation.
PSBLAS	Parallel Sparse Basic Linear Algebra Subroutines.
RAM	Random-Access Memory.
SIMD	Single-Instruction, Multiple-Data.
SIMT	Single-Instruction, Multiple-Threads.
SpMV	Sparse Matrix–Vector multiplication.
UML	Unified Modeling Language.

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. We cover how to handle certain kinds of design requirements, and illustrate 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* [18] in the context of scientific computation on sparse matrices. In the Object-Oriented Design (OOD) of software, the term "design pattern" denotes an accepted best practice in the form of a common solution to a software design problem that recurs in a given context.

AUTHOR COPY

Sparse matrices and related computations are one of the centerpieces of scientific computing: most mathematical models based on the discretization of Partial Differential Equations (PDEs) require the solution of linear systems with a coefficient matrix that is large and sparse. An immense amount of research has been devoted over the years to the efficient implementation of sparse matrix computations on high performance computers. A number of projects provide frameworks and 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. Among these projects are Trilinos [23], the Portable Extensible Toolkit for Scientific Computation (PETSc) [2], the Formal Linear Algebra Methods Environment (FLAME) [9] and Parallel Sparse Basic Linear Algebra Subroutines (PSBLAS) [12,17].

Trilinos from Sandia National Laboratories 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. Trilinos is a collection of packages and allows package developers to focus only on things that are unique to their package. Many of the algorithms in PETSc and Trilinos can be interchanged via abstract interfaces without impacting the application code. Both projects employ the Message Passing Interface (MPI) to exploit parallel, distributed-memory computers, and both 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.

The PSBLAS library supports parallel, sparse applications and 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 describe below (see also [16]).

The Formal Linear Algebra Matrix Environment (FLAME) project encompasses the `libflame` library for developing parallel, dense linear algebra codes; a notation for expressing algorithms; a methodology for systematic derivation of algorithms; application programming interfaces (APIs) for representing the algorithms in code; and tools for mechanical derivation, implementation and analysis of algorithms and implementations.

Sparse matrices are widely used in scientific computations; most physical problems modeled by PDEs are solved via discretizations that transform the original equations into a linear system and/or an eigenvalue problem with a sparse coefficient matrix. The traditional definition of a sparse matrix, usually attributed to Wilkinson [13], is that there are so many zeros that it pays off to devise a representation that does not store them explicitly. This means abandoning the language's native array type along with the underlying assumption that one can use the default language rules to infer the indices (i, j) associated with an element a_{ij} from the element's position in memory and vice versa. Most viable replacements for these assumptions involve storing the indices explicitly;[2] 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 (COO), Compressed Storage by Rows (CSR), Compressed Sparse Columns (CSC), ELLpack (ELL), JAgged Diagonals (JAD), and other formats (a historical perspective of various sparse storage formats can be found in [6,14]). 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).

Most matrices coming from discretized PDEs are indeed sparse, because the nonzeros in each matrix row will depend on the local topology of the discretization stencil, involving a very small number of neighbors, independently of the overall size and shape of the computational domain.

This paper demonstrates how well-known software engineering design patterns can be used to implement an interface for sparse matrix computations on Graphics Processing Units (GPUs) starting from an existing, non-GPU-enabled library. Specifically, we focus on some known design patterns (State, Mediator, Builder

[2]In some very specific cases, it is possible to have a reduced explicit storage of indices; an example is the storage by diagonals. These storage formats are usually not generally applicable, but they are easily supported in our framework.

AUTHOR COPY

and Prototype), leveraging the newly available OOP constructs of Fortran 2003 in scientific applications.

We present what we believe to be the first published demonstration of these patterns in Fortran 2003, and we discuss the application of these patterns to sparse matrix computations within the PSBLAS library. We then discuss how to employ the considered techniques to interface the existing PSBLAS library with a plug-in in the Compute Unified Device Architecture (CUDA) C language that implements the computational kernels on the NVIDIA GPUs. Our reported experience demonstrates that the application of design patterns facilitated a significant reduction in the development effort in the presented context; we also present some experimental performance results on different NVIDIA platforms demonstrating the throughput improvement achieved by implementing the PSBLAS interface for sparse matrix computations on GPUs. The software described in this paper is available at http://www.ce.uniroma2.it/psblas.

The PSBLAS library provides by default three storage formats, namely COO, CSR and CSC, for real and complex data, in both single and double precision. The work described in this paper revolves around the ELG storage format for the GPU, which is itself an extension of the ELLpack storage format, bringing the number to five; to these we also add interfaces to the CSR and HYB storage format provided in the NVIDIA cuSPARSE [31] library, for a total of seven storage formats. Note however that in this paper we are focusing on the interfacing of the storage formats, not on a detailed comparison of their relative efficiencies, which will be examined in future work.

Extending numerical libraries to integrate GPUs is a growing trend. The current paper describes the extension of PSBLAS to integrate GPUs. Likewise, the other aforementioned numerical libraries and frameworks have recently been extended to harness the GPU power. For example, PETSc [29] recently added GPU support. Specifically, PETSc recently introduced a new subclass of the vector class. This new subclass performs its operations on NVIDIA GPUs. PETSc also now contains a new sparse-matrix subclass that performs matrix–vector products on the GPU. Developers use these new subclasses transparently from existing PETSc application codes, whereas the implementation of linear algebra is done with the Thrust [32] and CUSP [30] libraries from NVIDIA. The core idea driving the PETSc implementation is the separation of the PETSc control logic from the computational kernels.

Trilinos is also on the path towards executing on scalable, manycore systems by providing basic computational capabilities on NVIDIA GPUs. The new Trilinos linear algebra package Tpetra uses the Thrust [32] and cuSPARSE [31] libraries from NVIDIA to support execution on GPUs.

The FLAME project automatically generates a large set of possible implementations. When a new architecture becomes available, an appropriate implementation is selected. For this purpose, FLAME establishes a separation of concerns between the code and the target architecture by coding the dense linear algebra library at a higher level of abstraction and leaving the computations and data movement in the hands of the runtime system [25]. However, FLAME addresses the issue of how to write the inner kernels on the new architecture (e.g., the GPU), whereas the current paper focuses more on creating a convenient and practical interface that plugs the inner kernels into the main framework after they have been developed.

The rest of the paper is organized as follows: Section 2 describes several design patterns; Section 3 provides some background on GPUs and presents the interfaces for sparse-matrix computations on GPUs starting with the PSBLAS library and focusing on matrix–vector multiplication with code examples; Section 4 demonstrates the utility and performance benefits accrued by use of the presented patterns; and Section 5 concludes the paper and gives hints for future work.

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.

OOD patterns have gained popularity in software design since the 1995 publication of the highly influential text *Design Patterns: Elements of Reusable Object-Oriented Software* [18] by Gamma et al., who are often referred to as the “Gang of Four” (GoF). Applying design patterns in a conscious way can be highly beneficial. Benefits stemming from an appropriate use of patterns include the improvement of the structure of the software and the readability of the code, the reduction of the programming costs, resulting in applications with better extensibility, flexibility and maintainability. Evidence from the literature suggests that these benefits have been reaped in the context of scientific applications only in the past decade, e.g., [10,19, 27] and most of the works on design patterns for scien-

AUTHOR COPY

tific software employed C++, Java or Fortran 95, the last of which requires emulating object-oriented programming (OOP) because of the lack of some object-oriented features, such as inheritance and polymorphism. The recent surge of interest [28,34,35] is due in part to the arrival of compilers that support the OOP constructs in Fortran 2003, the only programming language with an international standards body for whom scientific programmers comprise the target audience. With this paper, we discuss implementations of design patterns that were first described in [18] and have been not previously demonstrated in Fortran 2003; the related paper [16] also discusses the usage of design patterns, but is focused on the software library itself.

Given the sizeable number of Fortran programs developed before widespread compiler support for Fortran 2003, most Fortran programmers do not yet employ OOP. Likewise, most programmers who employ OOP do not write in Fortran. For the reader unfamiliar with OOP in Fortran, we summarize here how the main concepts of OOP map to Fortran language features. OOP rests on four pillars: *encapsulation*, *information hiding*, *inheritance* and *polymorphism*. Additionally, Fortran supports two types of data-structure relationships that frequently arise in OOP: *aggregation* and *composition*.

Fortran's primary mechanism for data encapsulation involves defining components inside *derived types* and type-bound procedures that manipulate those components. Components in turn may be of derived type or of language-provided, intrinsic type. When a component is of derived type, the relationship between the containing type and the contained type is aggregation. Composition is the specific case of aggregation in which the aggregating object and the aggregated object have coincident lifetimes: constructing an object of the aggregating type implies simultaneously constructing the corresponding aggregated object; likewise, destroying an object of the aggregating type implies the immediate destruction of its aggregated component(s).

Inheritance is the specific case of aggregation in which the aggregating object (the "child") supports the type-bound procedures defined in the aggregated object (the "parent"). Fortran supports inheritance via the language's type extension mechanism. A Fortran compiler automatically places a component of the parent type inside the child type. The compiler further facilitates invocations of the parents' type-bound procedures on the child by automatically delegating such invocations to the type-bound procedures defined in the parent (unless the programmer overrides this automatic behavior).

Type extension also implies one form of dynamic polymorphism. When one invokes a type-bound procedure on an object in Fortran, the compiler passes that object to the procedure as the "passed-object dummy argument" by default. All passed-object dummy arguments must be declared with the keyword `class` instead of the usual `type`. The `class` declaration allows for the actual argument passed to be an object of the declared type or of any type that extends the declared type. Thus, the type of the declared object can vary from one invocation to the next. In this article, we treat the common OOP term "class" as synonymous with "extensible derived type". Most Fortran derived types, including all derived types discussed in this article, are extensible. Other forms of dynamic polymorphism exist in the language as well.

For OOP purposes, the Fortran `module` feature provides the most important mechanism for hiding information. A `module` determines the scope within which an entity is visible to other parts of the program. Giving an entity the `private` attribute hides it from program lines outside the `module` that contains the entity. Giving an entity the `public` attribute makes it visible to program lines outside the given `module`. Both data and procedures can be declared `public` or `private`.

Lastly, many forms of static polymorphism in Fortran predate Fortran 2003. For example, a programmer may overload function names, enabling the compiler to determine which specific function to invoke based on the type, kind and rank of the actual arguments passed at the point of invocation.

A final OOP concept plays a central role in many design patterns: interfaces. Although Fortran lacks an interface feature in the sense defined by OOP, a programmer can model interfaces via abstract classes. The Fortran keyword `abstract` employed in a class definition indicates that no instance of the class may be constructed. However, the name of the abstract class may be used in declaring an object whose dynamic type may be any non-abstract class that extends the abstract class. In that sense, the abstract class serves as an interface to the non-abstract classes that extend the abstract class. Program lines may therefore be written that depend only on the interface (the abstract class) with no direct reference to the implementation (the non-abstract class).

Figure 1 demonstrates modern-Fortran OOP. In that figure, a non-abstract, `cartesian_vector` class extends an abstract `vector` class and aggregates a `units` class. The `vector` class demonstrates one more OOP principle: an abstract method, or "de-

AUTHOR COPY

```fortran
module vector_interface ! "Module" defines scope and ensures type safety of procedure arguments.
  implicit none         ! Prevent implicit typing of undeclared variables and procedures.
  private               ! Hide everything by default.
  public :: vector      ! Expose vector class and public type-bound procedures (methods).
  type, abstract :: vector ! Abstract class
  contains
    procedure(length_interface), deferred :: length ! Deferred binding.
  end type
  abstract interface ! Specify requirements for child class implementations of deferred binding.
     function length_interface(this) result(v)
       import vector ! Bring the "vector" name into the abstract interface scope.
       class(vector), intent(in) :: this ! The "intent(in)" attribute prevents modifying
     end function                        ! the actual argument passed to the procedure.
  end interface
end module
module cartesian_vector_class
  use vector_interface, only : vector ! Import only the vector class and its public methods.
  implicit none
  private
  public :: cartesian_vector
  type units ! Non-abstract class.
    integer :: MeterExponent,KilogramExponent,SecondExponent
  end type
  type, extends(vector) :: cartesian_vector ! Child inherits from parent vector class.
    private                                 ! Hide child class components.
    real, dimension(:), allocatable :: x    ! Encapsulation.
    type(units) :: mks                      ! Aggregation (composition).
  contains
    procedure :: length        ! Method.
  end type
  interface cartesian_vector    ! Generic name for referencing one or more procedures that
    module procedure new_vector ! must be distinguishable by their arguments' type, kind,
  end interface                 ! and rank. (Static polymorphism.)
contains
  real function length(this)
    class(cartesian_vector), intent(in) :: this ! Dynamic Polymorphism.
    length = norm2(this%x)
  end function
  type(cartesian_vector) function new_vector(array,m,k,s) ! Instantiate and return new object.
    real, dimension(:), intent(in) :: array
    integer, intent(in) :: m,k,s
    new_vector%x = array              ! Initialize new object's components.
    new_vector%mks%MeterExponent = m
    new_vector%mks%KilogramExponent = k
    new_vector%mks%SecondExponent = s
    end function
end module
program main
  use cartesian_vector_class, only : cartesian_vector ! Import class into main program scope.
  implicit none
  type(cartesian_vector) :: velocity   ! Declare object class.
  velocity = cartesian_vector([3.,4.,0.],m=1,k=0,s=-1) ! Instantiation.
  print*,"Speed=",velocity%length() ! Invoke length method on cartesian_vector object (velocity).
end program
```

Fig. 1. Demonstration of OOP in modern Fortran.

AUTHOR COPY

ferred binding" to use Fortran terminology (`length` at line 7 in Fig. 1). A deferred binding lacks an implementation but imposes a requirement that any non-abstract child class must implement a procedure corresponding to a specified `abstract interface` (`length_interface` at lines 9–14 in Fig. 1). The comments in Fig. 1 mark the Fortran implementation of various OOP terms. The Glossary at the beginning of this paper summarizes several of the important concepts and acronyms at the intersection of OOP and modern Fortran. For a more in-depth presentation of OOP in modern Fortran with more extensive coding examples, see [35].

2.1. "State"

The State pattern in OOD is a behavioral pattern that involves encapsulating an object's state behind an interface in order to facilitate varying the object's type at runtime. Figure 2 shows a UML class diagram of the State pattern, including the class relationships and the public *methods*. The methods described in an OOD typically map to the type-bound procedures in Fortran OOP.

Let us consider the problem of switching among different storage formats for a given object. Before the dawn of OOP, a common solution involved defining a data structure containing integer values that drive the interpretation and dispatching of the various operations on the data. This older route complicates software maintenance by requiring the rewriting and recompiling of previously working code every time one incorporates a new storage format. A more modern, object-oriented strategy builds the dispatching information into a type system, thereby enabling the compiler to perform the switching. However, most OOP languages do not allow for a given object to change its type dynamically (except for the less common, dynamically typed languages). This poses the dilemma of how to reference the object and yet allow for the type being referenced to vary.

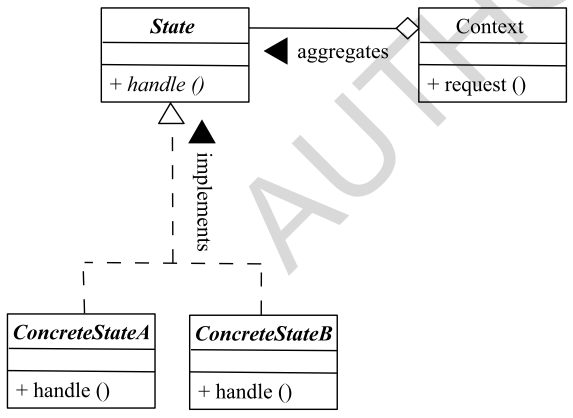

Fig. 2. Unified Modeling Language (UML) class diagram for the State pattern: classes (*boxes*) and relationships (*connecting lines*), including abstract classes (*bold italics*) and relationship adornments (*line labels*) with solid arrowheads indicating the relationship direction. Line endings indicate the 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 (−). Non-bold italics denotes an abstract method. We omit private attributes.

The solution lies in adding a layer of indirection by encapsulating the object inside another object serving only as an interface that provides a handle to the object in a given context. All code in the desired context references the handle but never directly references the actual object. This solution enables the compiler to delay until runtime the determination of the actual object type (what Fortran calls the "dynamic type"). The sample code in Fig. 3 demonstrates the State pattern in a sparse-matrix context, wherein a `base_sparse_mat` type plays the role of "State" from Fig. 2 and `spmat_type` serves as the "Context" also depicted in Fig. 2. The methods of the outer class delegate all operations to the inner-class methods. The inner class serves as the actual workhorse.

The State pattern allows easy handling of heterogeneous computing platforms: the application program making use of the computational kernels will see a uniform outer data type, but the inner data type can be easily adjusted according to the specific features of the processing element that the current process is running on.

The representation of an index map provides another example wherein the State pattern offers a natural solution. In parallel programming, we often have the following problem: *There exists an index set scanning a problem space, and this index set is partitioned among multiple processors.* Given this data distribution, we need to answer questions that hinge upon relating the "global" indices to their "local" counterparts:

(1) Which processor owns a certain global index, and to what local index does it correspond?
(2) Which global index is the image of a certain local one?

Answers to these questions depend on the strategy by which the index space has been partitioned. Example strategies include:

AUTHOR COPY

```
module base_mod
  ! The base class for STATE objects
  type :: base_sparse_mat
    ! data components here
  contains
    procedure, pass(a) :: foo => base_foo
  end type base_sparse_mat
contains
  subroutine base_foo(a)
    class(base_sparse_mat) :: a
    ! Actual implementation
    write(*,*) 'This_the_FOOing_of_a_base_sparse_matrix'
  end subroutine base_foo
end module base_mod

module coo_mod
  ! A derived class for STATE objects in COO
  use base_mod
  type, extends(base_sparse_mat) :: coo_sparse_mat
    integer :: nnz=0                 !> Number of nonzeros.
    integer, allocatable :: ia(:)    !> Row indices.
    integer, allocatable :: ja(:)    !> Column indices.
    real, allocatable :: val(:)      !> Coefficient values.

  contains
    procedure, pass(a) :: foo => coo_foo
  end type coo_sparse_mat
contains
  subroutine coo_foo(a)
    class(coo_sparse_mat) :: a
    ! Actual implementation
    write(*,*) 'This_the_FOOing_of_a_coo__sparse_matrix_with',&
         & a%nnz,'_nonzero_entries'
  end subroutine coo_foo
end module coo_mod
```

```
module spmat_mod
  ! The class for CONTEXT objects
  use base_mod
  type :: spmat_type
    class(base_sparse_mat), allocatable :: a
  contains
    procedure, pass(a) :: foo => spmat_foo
  end type spmat_type
contains
  subroutine spmat_foo(a)
    class(spmat_type) :: a
    call a%a%foo()
  end subroutine spmat_foo
end module spmat_mod

! Simple example
program try
  use spmat_mod
  use coo_mod
  type(spmat_type) :: a
  ! Start with the base STATE
  allocate(a%a)
  call foobar(a)
  ! Switch to COO
  deallocate(a%a)
  allocate(coo_sparse_mat :: a%a)
  call foobar(a)

contains
  ! Workhorse
  subroutine foobar(a)
    type(spmat_type) :: a
    call a%foo()
  end subroutine foobar
end program try
```

Fig. 3. Sample code using the State pattern.

AUTHOR COPY

- serial/replicated distribution, where each process owns a full copy and no communication is needed;
- block distribution, where each process owns a subrange of the indices;
- arbitrary assignment, where we have (at least conceptually) a list specifying for each index its owner process.

Encapsulating these variations under a uniform outer shell allows for having a single entry point with no need for conditional compilation in the case in which the user is actually running in serial mode. The pattern also allows for delegating to the internal setup the choice of the most appropriate representation in a given environment (see Section 2.2).

The usefulness of having an object with the ability to switch among different types was recognized long ago; as early as 1983 we find the following statement in [1], Section 4.12:

> Often a seemingly simple representation problem for a set or mapping presents a difficult problem of data structure choice. Picking one data structure for the set makes certain operations easy, but others take too much time and it seems that there is no one data structure that makes all the operations easy. In that case the solution often turns out to be the use of two or more different structures for the same set or mapping.

The State pattern handles such situations.

Another advantage of the State pattern becomes apparent in a message passing parallel context: the various processes can have both a uniform outer type (the Context) as well as an inner type (State) which needs not be the same everywhere. Hence it becomes very easy to handle a heterogeneous computing environment.

2.1.1. *“Mediator”*

To enable a runtime class switch, it is necessary to devise a conversion strategy among all of the inner object’s allowable dynamic types. The obvious strategy of writing conversion methods for each possible pair of dynamic data types scales poorly with growth in the type system for two reasons:

(1) The number of conversion routines needed grows as the *square* of the number of legal data types;
(2) A predefined set of converters cannot cope with the addition of a new data type variation without code modifications.

Another behavioral pattern, the Mediator pattern, alleviates these problems. Adapted to our particular situation, the pattern takes on the following meaning:

> Route all conversions through a predefined storage format for which all classes must provide *to* and *from* conversion methods.

This is equivalent to replacing a network having a fully connected topology with one having a star topology: the shortest path between any two nodes is now longer, but the number of arcs only grows linearly. With this scheme, the conversion between two different storage formats can be done in two steps, as shown in the following code example:

```
subroutine cp_fmt_to_fmt(a,b)
    class(base_sparse_mat), intent(in)  :: a
    class(base_sparse_mat), intent(out) :: b
    type(coo_sparse_mat)                :: tmp

    call a%copy_to(tmp)
    call b%copy_from(tmp)
end subroutine cp_fmt_to_fmt
```

Note that the copy method `cp_fmt_to_fmt` does *not* need to know the precise dynamic type of `a` and `b`, and therefore can be used untouched, not even recompiled, as new derived (child) classes that extend the `base_sparse_mat` base (parent) class.

In our production code this conversion is usually reimplemented in all classes, i.e., the first argument `a` has a more restricted type; this adds a bit of complexity, but the resulting code base still grows linearly, while allowing the developer to add “shortcuts” based on the dynamic type of `b` for those cases where this is worth the effort; one common such case is when `b` has the same dynamic type as `a`, and in this situation it obviously pays off to use a direct copy of each component.

2.2. *“Builder” and “Prototype”*

This section concerns how certain design patterns reduce the cost of maintaining and extending a software collection.

The development of a software library for sparse-matrix computations harbors considerable complexity, requiring many tradeoffs among multiple factors. Upon exerting substantial effort to properly design the data structures and algorithms according to fit various target applications and computing architectures, the reusability and maintainability of the infrastructure become obvious priorities. These priorities buttress the investments against changes in space – ensuring the ease-of-use by researchers working on different, existing platforms or applications – and in time – accommodating the evolution of the computing landscape.

It would thus be desirable to achieve the following goals:

- Adding a new storage format to the existing framework should require minimal effort;
- The existing software framework should require no change in response to the new storage variations;
- Existing applications should require no substantial changes in their source code.

Reviewing these requirements from a software designer’s viewpoint, one sees that the State design pattern plays a key role:

- The programmer who adds a new storage format needs only to add an inner class interfacing to the new codes;
- The existing solver software framework remains essentially untouched because it only references the outer container class;
- The application code needs only to link to the implementation of the new class.

And yet, this is not a complete solution because it leaves unaddressed a fundamental question: how does the application instantiate a sparse matrix of a given storage format?

One can easily discern the inadvisability of having the application code instantiate directly an instance of one storage class:

AUTHOR COPY

(1) Direct instantiation in application code would violate the encapsulation principle by requiring the user to deal directly with the class components;
(2) Hardwiring the often complex steps of assembling a sparse matrix limits the code's ability to adapt to the many various use cases;
(3) In particular, it is impossible to have a "one-shot" constructor for all but the most trivial cases.

One can interpret the strategy employed in PSBLAS in terms of two design patterns: Builder and Prototype.

2.2.1. Builder

The Builder pattern in OOD is a creational pattern that allows for an abstract specification of the steps required to construct an object. Figure 4 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 Builder pattern decouples the process of instantiating a storage class from the knowledge of the precise storage details by specifying a set of steps to be performed to build such an object relying on methods that exchange data to/from the internal representation. A proper design of the data exchange methods has the additional advantage of allowing the build of the matrix entries to be performed in any order is most convenient to the application. Considering for instance a PDE discretized by finite differences, it is relatively easy to build a whole row of the matrix at the same time; by contrast, when employing finite elements, it is natural to proceed element by element, but each element contributes to entries in multiple rows, so that

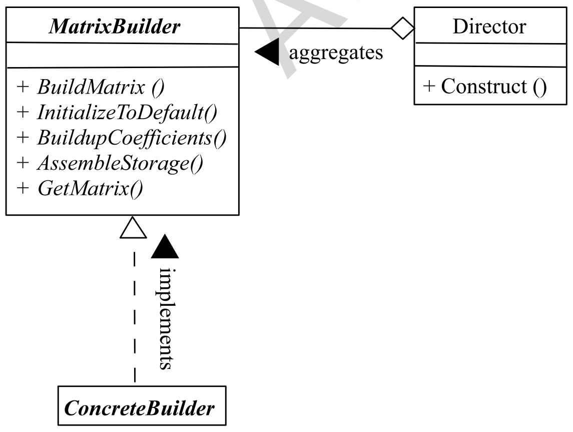

Fig. 4. UML class diagram for the Builder pattern.

asking the user to generate a given row all at once is almost the same as asking to build the whole matrix.

The strategy devised to build a sparse matrix is thus:

(1) Initialize the elements to some default values;
(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 and 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.

The application code exchanges data with the library with simple lists of triples: row index, column index, and coefficient. This allows for an arbitrary order in the matrix build phase. During the buildup loop, the library is free to store the incoming data in whatever internal format is most convenient. PSBLAS uses COOrdinate format for this phase, but we could change this default choice without any impact on the application code.

The only point at which the ultimate, desired output storage format must be enforced explicitly is during the assembly step. In PSBLAS, this enforcement occurs at the inner level by allocating a new object of the desired class and converting to it from the existing inner object. The conversion happens via the aforementioned Mediator pattern.

A subtle point now arises: the assembly method is part of the library framework; thus, we would like to write it just once. However, implementing the Mediator necessitates instantiating an object of the target inner class to convert the data from the existing storage. Hence, the question arises: how can the library code allocate a new object with a dynamic type known at runtime but not at compile-time? The Prototype pattern answers this question.

2.2.2. Prototype

The Prototype design pattern might also be defined as "copy by example": when a method needs to instantiate an object whose dynamic type is unknown at compile time, it does so by referring to another object as a "source" or a "mold" for the desired object. The class for the copied object must include a cloning or molding method by which the desired copy can be obtained: cloning creates a full copy of the source object, whereas molding creates an empty copy with only the correct dynamic type but none of the object's component values. This is essentially the idea of C++ "virtual constructors" (see Section 15.6.2 of [37]), for

AUTHOR COPY

which the user must write a function that calls a constructor because actual constructors must know the class of the object they construct.

What represents a pattern in one language might be an intrinsic feature of another language. Whereas the programmer explicitly writes molding methods in C++, the Fortran `allocate` statement includes an optional `mold` argument that can be used to copy only the dynamic type of the argument into the object being allocated without copying any of the mold object's component data. This argument finds use when instantiating what Fortran terms "polymorphic objects", which are objects that either have the `allocatable` attribute, the `pointer` attribute, or are procedure dummy arguments. The dynamic type of pointer or allocatable objects must be specified at the time of instantiation by one of several forms of an `allocate` statement. The following example demonstrates the most common way:

```
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, the following alternatives also suffice:

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

or

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

where the `source=` variant clones `sourcemat` and `mold=` copies only its dynamic type into `mat_object`. The source or mold argument can be any variable with a compatible dynamic type. Compatibility requires that the dynamic type either matches the declared type (`base_sparse_mat`) or extends the declared type. In particular, the source or mold argument can be a dummy argument passed by the application code to the library. Fortran thus embeds the Prototype pattern into the language via the `source` and `mold` arguments to the `allocate` statement.

In summary, adding a new storage format involves the following steps:

(1) Define a child class, e.g., `new_inner_matrix`, from the parent class and implement the required methods;
(2) Declare in the application code a variable with `type(new_inner_matrix)`;
(3) Pass this variable as the "mold" dummy argument to the assembly routine.

These steps necessitate neither modification nor recompilation of the library code.

3. Interfacing sparse-matrix computational kernels on GPUs with PSBLAS

GPUs have recently gained widespread adoption in the scientific computing community and are now routinely used to accelerate a broad range of science and engineering applications, delivering dramatically improved performance in many cases. They are used to build the core of the most advanced supercomputers (according to the June 2012 list of the TOP500 supercomputer sites,[3] three out of the ten top supercomputers are GPU-based). Furthermore, cloud-computing providers are deploying clusters with multiple GPUs per node and high-speed network interconnections (e.g., Cluster GPU instances in Amazon EC2) in order to make them a feasible option for HPC as a Service (HPCaaS) [15].

In this section, we discuss how our design techniques help in interfacing sparse-matrix computational kernels on the GPU with the existing PSBLAS library. Section 3.1 presents some background on the NVIDIA GPU architecture and its CUDA programming model. Section 3.2 deals with sparse-matrix computations on GPUs; specifically, it presents how to plug a GPU-friendly, sparse-matrix storage format and its associated computational kernels into PSBLAS using the design patterns discussed in Section 2. Section 3.3 analyzes how to employ the considered design patterns to manage vectors in the GPU version of PSBLAS. Finally, Section 3.4 describes how the design patterns can also be used to encapsulate the sparse matrix format used in the NVIDIA's cuSPARSE library.

We focus on the usage of design patterns for interfacing new kernels and maintaining an existing body of software in response to architectural and programming-model changes. Hence, we do not analyze in-depth the kernels' implementations on GPUs, which is the subject of ongoing, related work to be published elsewhere.

[3] http://www.top500.org/lists/2012/06.

AUTHOR COPY

3.1. *Overview of NVIDIA GPU architecture and programming model*

The NVIDIA GPU architectural model is based on a scalable array of multi-threaded, streaming multiprocessors, each composed of a fixed number of scalar processors, a dual-issue instruction fetch unit, an on-chip fast memory with a configurable partitioning of shared memory, an L1 cache plus additional special-function hardware. The computation is carried on by threads grouped into blocks. More than one block can execute on the same multiprocessor, and each block executes concurrently. Each multiprocessor employs a Single-Instruction, Multiple-Threads (SIMT) architecture akin to the Single-Instruction, Multiple-Data (SIMD) architecture of traditional vector supercomputers. The multiprocessor creates, schedules, and executes threads in groups called warps; threads in a warp start together at the same program address but can execute their own instructions and are free to branch independently. A warp executes one common instruction at a time; therefore maximum performance is achieved when all threads in a warp follow the same path.

CUDA is the programming model provided by NVIDIA for its GPUs; a CUDA program consists of a host program that runs on the Central Processing Unit (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 containing various levels that differ by speed, size, and scope. The 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.

3.2. *Sparse matrix computations on a GPU*

The considerable interest in General-Purpose Graphics Processing Unit (GPGPU) computation arises from the significant performance benefits possible. For example, the chapters of [24] present various case studies that leverage the GPU power for many computationally-intense applications belonging to diverse application domains, not just to graphics; the work in [3,5,39] 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 SIMT architecture) are considered for implementing sparse-matrix computations. In particular, Sparse Matrix–Vector multiplication (SpMV) has been the subject of intense research on every generation of high-performance computing platforms because of its role in iterative methods for solving sparse, linear systems and eigenvalue problems.

Sparse-matrix computations on GPUs 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 Refs [7,8,11] and the CUDA Sparse Matrix library (cuSPARSE) [31] included in CUDA 4.0 and 4.1. Here we will discuss briefly only the basic issues involved because our focus is on encapsulation and interfacing rather than on the inner kernels themselves.

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. Starting from a GPU-friendly format we developed, we will concentrate on how to use the design patterns discussed in Section 2 to plug the new format and the GPU support code into PSBLAS.

Our GPU-friendly storage format, named ELL-G, is a variation of the standard ELLpack (or ELL) format [22]: an M-by-N sparse matrix with at most K nonzeros per row is stored as dense M-by-K arrays `val` and `ja` holding the nonzero matrix values and the corresponding column indices, respectively. Also, the rows in `val` and `ja` with fewer than K nonzero elements are padded with zeros (or some other sentinel, padding value). ELL thus fits a sparse matrix into a regular data structure, making it a good candidate for implementing sparse-matrix operations on SIMT architectures. In the computational kernel, each row of the sparse matrix is assigned to a thread that performs the associated (sparse) dot product with the input vector x. Threads are organized in warps, according to CUDA programming guidelines, so that each multiprocessor element on the GPU card handles a group of consecutive rows of the sparse matrix. The storage of the matrix data in a regular, Two-Dimensional (2D) array allows each thread to compute easily where its data (i.e., the relevant matrix row) reside in memory given only

AUTHOR COPY

the row index and the size of the 2D array. This contrasts with the CSR storage format mentioned in Section 3.4, where finding the starting point of a given row requires that a thread travels through the previous rows in the data structure. The ELL format makes efficient use of memory when the maximum number of nonzeros per row is close to the average. The usage of ELL format and its variants on GPUs has been analyzed in, for example, [38,39].

Our storage format ELL-G takes into account the memory access patterns of the NVIDIA Tesla architecture [26], 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 move data from the main memory to the GPU Random-Access Memory (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 challenges in GPU programming [21]. In particular, having the matrix data moved from main memory to the device memory each time a matrix–vector product is performed would totally destroy any computational advantage of the GPU platform. Therefore we have to prearrange the matrix data to reside on the GPU. To add support for NVIDIA GPUs in PSBLAS, 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 multiplication time, invoke the code on the GPU side;
- At deallocation time, release memory on both the host and device sides.

The class interface for ELL-G is defined as follows (in the code, we refer to ELL-G as `elg` because PSBLAS matrix format names are 3 characters long):

```
type, extends(base_sparse_mat) :: ell_sparse_mat
  !
  ! ITPACK/ELL format, extended with IRN and IDIAG.
  !

  integer, allocatable :: irn(:), ja(:,:), idiag(:)
  real, allocatable :: val(:,:)
contains
      .....
end type ell_sparse_mat

type, extends(ell_sparse_mat) :: elg_sparse_mat
  type(c_ptr) :: deviceMat = c_null_ptr
contains
      .....
end type elg_sparse_mat
```

where the Fortran standard provides the `c_ptr` type to facilitate handling a C-language pointer in the form of a native Fortran derived type, and where Fortran also provides the constant `c_null_ptr` for assigning the companion C compiler's `null` value to a `c_ptr`. Fortran's C-interoperability feature set requires that compilers make `c_ptr` and `c_null_ptr` available via Fortran's intrinsic `iso_c_binding` module. Finally, the `irn` component of `ell_sparse_mat` contains the actual row size – that is the number of relevant entries in that row (the ELLPACK format has padding) – and the `idiag` component contains the position of the entry holding each row's diagonal element.

When converting from the COO format supported by PSBLAS to the new ELL-G format it would be possible in principle to reuse the parent type conversion and then simply copy the relevant data to the GPU. However, some adjustments may be necessary to the leading dimensions of the `val` and `ja` components, padding to a size that agrees with the specific features of the available GPU card. Once the ELL (parent) data structure is ready, the ELL-G (child) data structure can invoke its `to_gpu` method that clones the host memory layout into the device memory. The implementation of the `to_gpu` method involves the invocation of appropriate library routines in the CUDA environment, e.g., `cudaMalloc()`, as well as the computation of appropriate parameters for the allocation sizes; these routines, as well as the computational routines, are written in the CUDA extension of the C language and are accessed via modern Fortran's C-interoperability features.

An `elg` sparse matrix has an associated control flag that signals which of the memory copies (host, device or both) contains an updated version of the matrix data. This scheme is also used for dense vectors as Section 3.3 discusses.

3.3. Vectors on the GPU

Vectors in the library follow the same State design pattern as matrices. At first sight, this would seem wasteful: after all, the Fortran language already provides very good native vector support, in particular `allocatable` arrays that are very well-protected against memory leaks.

The additional layer of indirection provided by State is nonetheless quite useful: it provides a uniform interface to memory that has been allocated elsewhere, outside of the control of the language runtime environment. The GPU device memory is an example of

AUTHOR COPY

such memory areas:[4] the memory management is performed by specialized routines in the CUDA runtime library, and the areas are interfaced via pointers handled through the `iso_c_binding` facility.

The vectors in the GPU version of PSBLAS have a dual-allocation scheme similar to that of the sparse matrices; a flag keeps track of whether it is the host copy or the GPU copy that has been most recently updated.

Keeping track of the updates to vector entries is essential: we want to perform the computations on the GPU, but we cannot afford the time needed to move data between the main memory and the GPU memory because the bandwidth of the interconnection bus would become the main bottleneck of the computation. Thus, each and every computational routine in the library is built according to the following principles:

- If the data type being handled is GPU-enabled, make sure that its GPU copy is up to date, perform any arithmetic operation on the GPU, and if the vector has been altered as a result, mark the main-memory copy as outdated.
- The main-memory copy is never updated unless this is requested by the user either
  - *explicitly* by invoking a synchronization method;
  - *implicitly* by invoking a method that involves other data items that are not GPU-enabled, e.g., by assignment to a normal array.

In this way, data items are put on the GPU memory "on demand" and remain there as long as "normal" computations are carried out. As an example, the following call to a matrix–vector product

```
call psb_spmm(alpha,a,x,beta,y,desc_a,info)
```

will transparently and automatically be performed on the GPU whenever all three data inputs `a`, `x` and `y` are GPU-enabled. If a program makes many such calls sequentially, then

- The first kernel invocation will find the data in main memory, and will copy it to the GPU memory, thus incurring a significant overhead; the result is however *not* copied back, and therefore:
- Subsequent kernel invocations involving the same vector will find the data on the GPU side so that they will run at full speed.

For all invocations after the first the only data that will have to be transferred to/from the main memory will be the scalars `alpha` and `beta`, and the return code `info`.

To manage the data transfers between the main memory and the GPU memory, an alternative would have been to use an automatic CPU–GPU memory management model such as the Asymmetric Distributed Shared Memory (ADSM) model for heterogeneous computing architectures proposed in [20]. ADSM allows a CPU direct access to objects in the GPU physical memory and provides two types of memory updates that determine when to move data on and off the GPU. However, ADSM does not allow the GPU access to main memory. Therefore, although our solution is PSBLAS-specific, we believe it provides an efficient manner to transfer data between the main memory and the GPU memory, avoiding redundant transfers that could be introduced by proposals to automate the CPU–GPU memory management [33].

We conclude this section by adding some statistics about the amount of code needed to achieve our solution. The basic framework for the State pattern for double-precision matrices totals 5238 Lines Of Code (LOC), of which 661 are shared with single-precision and complex variants. The total line count includes the base sparse-matrix type definition, the container class, and all the shared methods. To these we have to add the support on the CPU side for the sparse matrix formats themselves. The corresponding total for the ELLPACK format is 5816 LOC. The code for the ELL-G innermost kernels in our in-house GPU library totals just 230 LOC, but this is purely for the matrix–vector product kernel: a C/C++ code would need to provide some infrastructure around it.

This paper chiefly concerns the incremental new code needed to bind CUDA into the main framework, given all the other pieces: on the Fortran side, the `elg` format is a class that extends `ell`. To make `elg` fully functional, we need about 2300 LOC. Moreover, many of the needed files can be cloned and adapted from the files for the CPU-side `ell` format.

3.4. Interfacing to the NVIDIA cuSPARSE library

The techniques described in the previous sections can also be employed to encapsulate the format used in

[4]At least when the compiler does not directly address the GPU memory as would be possible with the implementation of the OpenACC standard but is not widely available in compilers at the time of this writing.

AUTHOR COPY

the cuSPARSE library [31], which provides a collection of basic linear algebra subroutines used for sparse matrices. One of the cuSPARSE formats for sparse matrices released with CUDA 4.0 is a CSR representation.[5] Because CSR is one of the default data formats of the PSBLAS library, interfacing PSBLAS with cuSPARSE can be achieved in a very simple fashion:

- Extend the CSR storage format by adding a copy of the matrix data on the GPU memory;
- Override the matrix–vector product methods to invoke the cuSPARSE library routines.

The interface itself is thus straightforward and comprises 1030 LOC, many lines of which were "cloned" and extended from the basic CSR implementation, which itself needs 2923 LOC in addition to the base classes. A complete total would include CUDA kernel lines, but NVIDIA does not release the corresponding code publicly. A similar exercise for the CuSPARSE 4.1 hybrid (HYB) format yields a Fortran interface comprising about 1030 LOC.

4. Performance results

This section presents performance results gathered on our GPU-enabled software. We remind the reader that this papers central focus falls on achieving code reusability inside an existing framework while keeping any related performance penalty as small as possible. Such an effort differs in aim from attempts to obtain the absolute best performance for our kernels on the GPU.

4.1. Human performance results

After writing the CUDA kernel code, embedding the new ELL-G format in the existing PSBLAS library required a very limited development effort on the order of a couple of days, including debugging. Indeed, a simplified version of the matrix class has been used as the basis for a programming exercise in tutorials on OOP in modern Fortran that two of the authors have offered at various institutions. In that exercise, one hour of a student's work suffices for writing the minimum code necessary to compute a single matrix–vector product and to convert the COO storage format to the ever-useful "rank-2 array" format.

[5] Plus some additional data only used for triangular matrices.

The overall software design helps keep the in-class exercise short by encouraging the reuse of existing implementations as templates for deriving new data structures. If the GPU-side code is a variation on an existing storage format, students can implement it starting from the CPU version, which in turn can reuse the basic storage formats PSBLAS provides by default.

In the same vein, interfacing with the CUDA version 4.0 programming environment took only a few hours. As explained in Section 3.4, this was facilitated by the cuSPARSE format being essentially the same as the already existing CSR, making it feasible to extend the existing class and reuse most of its supporting code, overriding only the computational kernels.

4.2. Machine performance results

Our computational experiments were run on two NVIDIA platforms:

- *Platform 1:* a GeForce GTX 285, with a maximum throughput of 94.8 Giga (billions) of Floating-point Operations Per Second, or GigaFLOPS (GFLOPS) in double precision, attached to an AMD Athlon 7750 Dual-Core CPU running at 2.7 GHz;
- *Platform 2:* a Tesla C2050, with a maximum throughput of 515 GFLOPS in double precision, attached to an Intel Xeon X5650 CPU.

We report the measured computation rates in Giga (billions) of Floating-point Operations Per Second, or GigaFLOPS (GFLOPS), assuming the number of arithmetic operations per matrix–vector product is to be approximately $2N_{nz}$, given N_{nz} nonzero matrix elements,[6] and the computation rate is averaged over multiple runs.

For the experiments, we used a collection of sparse matrices arising from a three-dimensional convection-diffusion PDE discretized with centered finite differences on the unit cube. This scheme gives rise to a matrix with at most 7 nonzero elements per row: the matrix size is expressed in terms of the length of the cube edge, so that the case *pde10* corresponds to a 1000×1000 matrix. Table 1 summarizes the matrix characteristics used in our experiments along with the matrix size (N rows) and the number of nonzero elements (N_{nz}). For the experiments with the SpMV ker-

[6] Some storage formats may employ padding with zeros and thus would perform more floating-point operations than strictly necessary.

AUTHOR COPY

Table 1

Sparse matrices used in the performance experiments

Matrix name	N	N_{nz}
pde05	125	725
pde10	1000	6400
pde20	8000	53,600
pde30	27,000	183,600
pde40	64,000	438,400
pde50	125,000	860,000
pde60	216,000	1,490,400
pde80	512,000	3,545,600
pde90	729,000	5,054,400
pde100	1,000,000	6,940,000

nel, we set $\beta = 0$ so that $y \leftarrow Ax$. We report the results only for double-precision calculations.

Our experiments compare the throughput of SpMV in PSBLAS exploiting the GPU using our ELL-G storage format with the performance obtained by the standard PSBLAS library on CPU, considering Platforms 1 and 2. The employed CPUs have multiple cores, but we only ran in serial mode for the purposes of this comparison.

Figure 5 shows the performance improvement on Platform 1 implementing the PSBLAS interface for sparse-matrix computations on GPUs when we include the overhead of transferring the vector data from the main memory to the GPU device memory. As explained in Section 3.3, this would correspond to the first invocation in a sequence, when the vector data is flagged to indicate that the valid copy is in the host memory, and thus it constitutes the worst case scenario. Even in this worst case, the GTX 285 vs AMD matrix–vector multiplication gives a speedup of up to 3. For small matrices, the code runs faster on the CPU than on the GPU due to multiple factors:

- The GPU architecture and programming model requires a sufficient number of threads and a sufficient workload to fully exploit its hardware capabilities;
- The overhead of data transfers between main memory and GPU is more noticeable;
- The CPU computations benefit from the cache memory, whereas the GPU (at least, the GTX 285) does not show a comparably large effect.

Optimizing for cache reuse on the CPU would also be interesting but beyond the scope of this paper.

Figure 6 reports the computation rates for the same operations and data as in Fig. 5 but with the vectors already in the GPU memory; this is a more realistic situ-

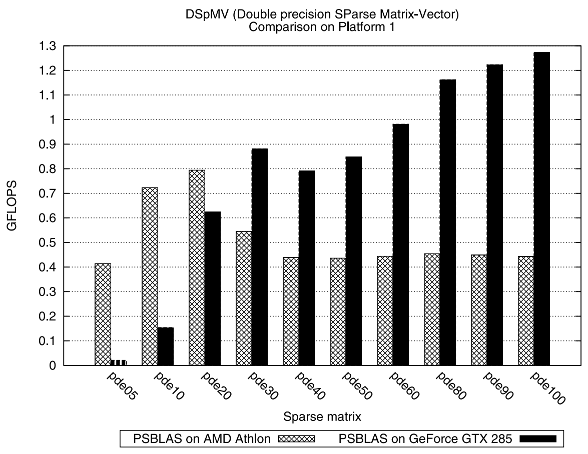

Fig. 5. Throughput comparison of PSBLAS on GPU (including vector copy-in overhead) and on CPU (double precision) for Platform 1.

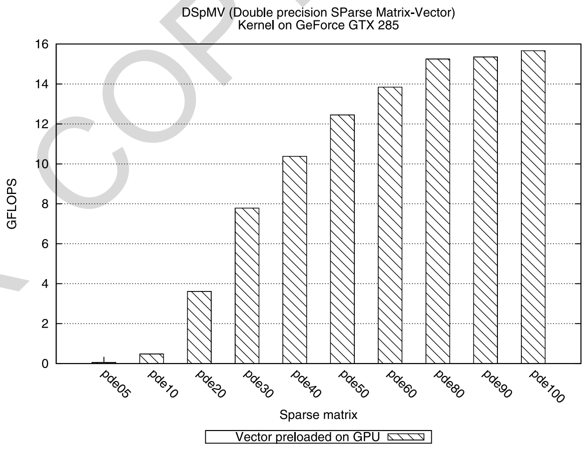

Fig. 6. Throughput of the SpMV kernel on GPU (without vector copy-in overhead) for Platform 1.

ation, since in the course of the solution of a linear system by an iterative solver, the vector data for all kernel invocations after the first one is already contained in the device memory, and there are typically many iterations of the algorithm's main loop.

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". According to the discussion in Section 3.3, during normal operations, all arithmetic operations are executed on the GPU, and the updated data is left there; during the execution of an iterative solution method, we have multiple iterations combining matrix–vector products with dot products and other vector operations. The high-

AUTHOR COPY

Table 2

Throughput and speedup on Platform 1 (AMD Athlon 7750, GeForce GTX 285)

Matrix name	GFLOPS		Speedup
	CPU	GPU	
pde05	0.414	0.055	0.13
pde10	0.723	0.481	0.66
pde20	0.794	3.61	4.54
pde30	0.545	7.784	14.28
pde40	0.439	10.380	23.63
pde50	0.436	12.452	28.54
pde60	0.444	13.842	31.18
pde80	0.454	15.250	33.63
pde90	0.449	15.354	34.20
pde100	0.443	15.664	35.35

Table 3

Throughput and speedup on Platform 1 (AMD Athlon 7750, GeForce GTX 285)

Matrix name	GFLOPS	
	Fortran	C/CUDA
pde05	0.05	0.06
pde10	0.46	0.52
pde20	3.51	3.88
pde30	7.74	7.99
pde40	10.29	10.58
pde50	12.44	12.60
pde60	13.86	13.79
pde80	15.23	15.03
pde90	15.33	15.30
pde100	15.64	15.46

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 and copied to host memory only upon exiting the iterative process.

Comparing these results with those in Fig. 5, we can see just how heavy the data transfer overhead is; having the vectors on the GPU enables a performance level that is essentially identical to that of the inner kernels. We still observe relatively low performance at small matrix sizes, but the ratio to the CPU is much more favourable. Table 2 reports the throughput measured in GFLOPS and the speedup we achieved on Platform 1 when the vectors are prearranged in the GPU memory (the throughput data coincide with those in Figs 5 and 6 for the CPU and GPU cases, respectively). The speedup of the GeForce GTX 285 with respect to the AMD Athlon 7750 ranges from 0.13 for the smallest matrix size to 35.35 for the largest matrix size.

Table 3 compares our code's performance with the measurements obtained by writing a simple C test program that calls directly into the CUDA kernels. Two observations become apparent:

(1) The Fortran performance rates differ slightly from Table 2 as they are purposedly obtained by a different test run, showing the unavoidable noise introduced by elapsed time measurements.
(2) Although the Fortran layer adds a performance penalty, as shown in the measurements at small sizes, this penalty quickly reduces to noise level for medium to large sizes, the cases for which one would be more likely to use a GPU.

Figures 7 and 8 show the performance improvement in terms of computation rates that we obtain on Plat-

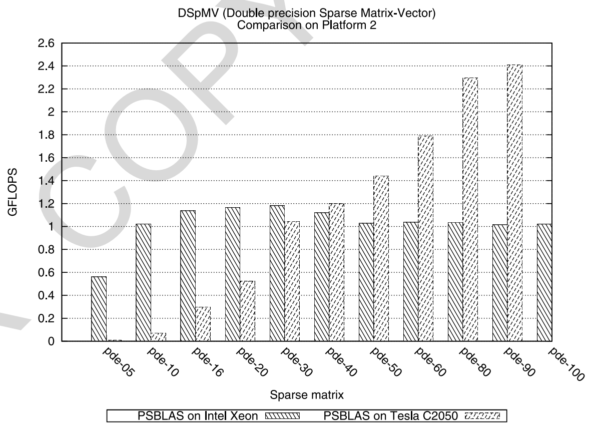

Fig. 7. Throughput comparison of PSBLAS on GPU (including vector copy-in overhead) and on CPU (double precision) for Platform 2.

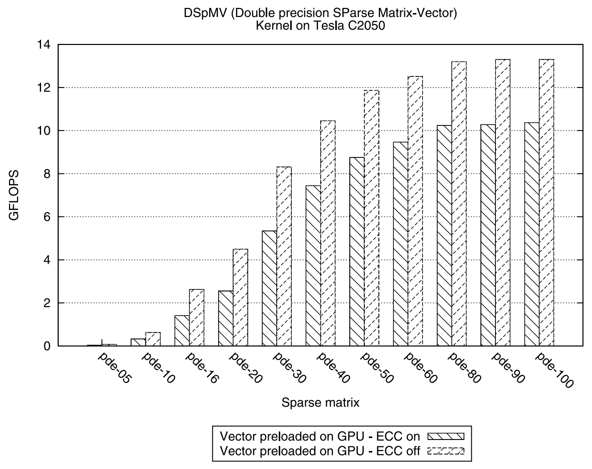

Fig. 8. Throughput of the SpMV kernel on GPU (without vector copy-in overhead) for Platform 2 when Error-Correcting Code (ECC) is either turned on or off.

AUTHOR COPY

Table 4
Throughput and speedup on Platform 2 (Intel Xeon CPU X5650, Tesla C2050) with ECC turned on

Matrix name	GFLOPS		Speedup
	CPU	GPU	
pde05	0.561	0.038	0.07
pde10	1.022	0.329	0.32
pde16	1.137	1.412	1.24
pde20	1.165	2.550	2.19
pde30	1.182	5.341	4.52
pde40	1.120	7.439	6.64
pde50	1.028	8.753	8.52
pde60	1.037	9.464	9.13
pde80	1.036	10.238	9.88
pde90	1.015	10.277	10.13
pde100	1.021	10.365	10.15

form 2. We observe that the performance of the GPU card on Platform 2 suffers considerably from having ECC turned on. ECC can fix single-bit errors and report double-bit errors. ECC protection on the memory was not available on the older GPU card of Platform 1. With ECC turned on, the GPU's available user memory is reduced by 12.5% because ECC bits occupy a portion of the dedicated memory (e.g., 3 GB total memory yields 2.625 GB of user-available memory on the Tesla C2050). To test the hypothesis of performance degradation due to ECC, we run the same experiments having previously disabled the ECC support on the C2050 card (this can be achieved through the `nvidia-smi` program but a reboot is needed to get the setting active). We observe that the speedup of the Tesla C2050 card with ECC off compared to that with ECC on ranges from 1.94 for the smallest matrix size to 1.28 for the largest matrix size.

Table 4 reports the throughput measured in GFLOPS and the speedup we achieved on Platform 2, always when the vectors are prearranged in the GPU memory. The speedup of the Tesla C2050 GPU (having ECC turned on, which is the default card configuration) with respect to the Intel Xeon CPU ranges from 0.07 for the smallest matrix size to 10.15 for the largest matrix size.

5. Conclusions

In this paper, we have discussed how several object-oriented software design patterns benefit sparse-matrix computation. We demonstrated how to use these patterns in writing interfaces for sparse-matrix computations on GPUs starting from the existing, non-GPU-enabled PSBLAS library [16]. Our experience shows that this solution provides flexibility and maintainability and facilitates the exploitation of GPU computation with resultant performance benefits. In our ongoing work, we recently added to PSBLAS support for the hybrid storage format of the cuSPARSE library. Thanks to the design patterns described in this paper, we have been able to obtain performance results for that storage format in less than one working day.

The ideas discussed in this paper were tested in PSBLAS. Future work will extend the discussed techniques to a multilevel preconditioner package that builds atop PSBLAS [12]. Additionally, we plan interfaces for ForTrilinos [36], which provides a set of standards-conforming, portable Fortran interfaces to Trilinos packages.

Acknowledgements

We gratefully acknowledge the support coming from "HPC Grant 2011 on GPU clusters" provided by the "Consorzio interuniversitario per le applicazioni di supercalcolo per università e ricerca" (CASPUR), Rome, Italy.

We also thank the anonymous referees whose comment helped us to improve the presentation of this paper.

References

[1] A. Aho, J. Hopcroft and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.
[2] S. Balay, W. Gropp, L.C. McInnes and B. Smith, PETSc 2.0 user manual, Technical Report ANL-95/11 – Revision 2.0.22, Argonne National Laboratory, 1995.
[3] D. Barbieri, V. Cardellini and S. Filippone, Generalized GEMM applications on GPGPUs: experiments and applications, in: *Parallel Computing: from Multicores and GPU's to Petascale, ParCo'09*, IOS Press, 2010, pp. 307–314.
[4] D. Barbieri, V. Cardellini, S. Filippone and D. Rouson, Design patterns for scientific computations on sparse matrices, in: *Euro-Par 2011: Parallel Processing Workshops*, LNCS, Vol. 7155, Springer, 2012, pp. 367–376.
[5] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, E.S. Quintana-Ortí and G. Quintana-Ortí, Exploiting the capabilities of modern GPUs for dense matrix computations, *Concurrency and Computation: Practice & Experience* **21** (2009), 2457–2477.
[6] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donat, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst, *Templates for the Solution of Linear Systems*, SIAM, 1993.
[7] M.M. Baskaran and R. Bordawekar, Optimizing sparse matrix–vector multiplication on GPUs, Technical Report RC24704, IBM Research, April 2009.

AUTHOR COPY

[8] N. Bell and M. Garland, Implementing sparse matrix–vector multiplication on throughput-oriented processors, in: *Proceedings of International Conference on High Performance Computing Networking, Storage and Analysis, SC'09*, ACM, 2009.

[9] P. Bientinesi, J. Gunnels, M. Myers, E. Quintana-Ortí, T. Rhodes, R. Geijn and F. Van Zee, Deriving dense linear algebra libraries, *Formal Aspects of Computing* **25** (2013), 933–945.

[10] C. Blilie, Patterns in scientific software: An introduction, *Computing in Science and Engineering* **4** (2002), 48–53.

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

[12] 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) (2010), 7–23.

[13] T.A. Davis and Y. Hu, The university of Florida sparse matrix collection, *ACM Trans. Math. Softw.* **38**(1) (2011), 1:1–1:25.

[14] I.S. Duff, A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices*, Oxford Univ. Press, 1986.

[15] R.R. Expósito, G.L. Taboada, S. Ramos, J. Touriño and R. Doallo, General-purpose computation on GPUs for high performance cloud computing, *Concurrency and Computation: Practice and Experience* **25**(12) (2013), 1628–1642.

[16] S. Filippone and A. Buttari, Object-oriented techniques for sparse matrix computations in Fortran 2003, *ACM Trans. Math. Softw.* **38**(4) (2012).

[17] S. Filippone and M. Colajanni, PSBLAS: a library for parallel linear algebra computations on sparse matrices, *ACM Trans. Math. Softw.* **26**(4) (2000), 527–550.

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

[19] H. Gardner and G. Manduchi, *Design Patterns for e-Science*, Springer, 2007.

[20] I. Gelado, J.E. Stone, J. Cabezas, S. Patel, N. Navarro and W.-M.W. Hwu, An asymmetric distributed shared memory model for heterogeneous parallel systems, *SIGARCH Comput. Archit. News* **38**(1) (2010), 347–358.

[21] C. Gregg and K. Hazelwood, Where is the data? why you cannot debate CPU vs. GPU performance without the answer, in: *Proceedings of 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2011, pp. 134–144.

[22] R. Grimes, D. Kincaid and D. Young, ITPACK 2.0 user's guide, CNA-150, Center for Numerical Analysis, University of Texas, 1979.

[23] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams and K.S. Stanley, An overview of the Trilinos project, *ACM Trans. Math. Softw.* **31**(3) (2005), 397–423.

[24] W.W. Hwu (ed.), *GPU Computing Gems: Emerald Edition*, Morgan Kaufmann, 2011.

[25] F.D. Igual, E. Chan, E.S. Quintana-Ortí, G. Quintana-Ortí, R.A. van de Geijn and F.G.V. Zee, The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations, *J. Parallel Distrib. Comput.* **72**(9) (2012), 1134–1143.

[26] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, NVIDIA Tesla: a unified graphics and computing architecture, *IEEE Micro* **28** (2008), 39–55.

[27] A. Markus, Design patterns and Fortran 90/95, *SIGPLAN Fortran Forum* **25**(1) (2006), 13–29.

[28] A. Markus, Design patterns and Fortran 2003, *SIGPLAN Fortran Forum* **27**(3) (2008), 2–15.

[29] V. Minden, B. Smith and M. Knepley, Preliminary implementation of PETSc using GPUs, in: *Proceedings of 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering*, Springer, 2010.

[30] NVIDIA Corp., CUDA CUSP library, 2012, available at: http://developer.nvidia.com/cusp.

[31] NVIDIA Corp., CUDA cuSPARSE library, 2012, available at: http://developer.nvidia.com/cusparse.

[32] NVIDIA Corp. CUDA Thrust library, 2012, available at: http://developer.nvidia.com/thrust.

[33] S. Pai, R. Govindarajan and M.J. Thazhuthaveetil, Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme, in: *Proceedings of 21st International Conference on Parallel Architectures and Compilation Techniques, PACT'12*, ACM, New York, NY, USA, 2012, pp. 33–42.

[34] D.W.I. Rouson, J. Xia and X. Xu, Object construction and destruction design patterns in Fortran 2003, *Procedia Computer Science* **1**(1) (2010), 1495–1504.

[35] D.W.I. Rouson, J. Xia and X. Xu, *Scientific Software Design: The Object-Oriented Way*, Cambridge Univ. Press, 2011.

[36] Sandia National Laboratories, ForTrilinos, 2012, available at: http://trilinos.sandia.gov/packages/fortrilinos.

[37] B. Stroustrup, *The C++ Programming Language*, Special edn, Addison-Wesley, Boston, MA, 2000.

[38] F. Vazquez, G. Ortega, J.J. Fernández and E.M. Garzon, Improving the performance of the sparse matrix vector product with GPUs, in: *Proceedings of 10th IEEE International Conference on Computer and Information Technology, CIT'10*, pp. 1146–1151, 2010.

[39] V. Volkov and J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: *Proceedings of 2008 ACM/IEEE Conference on Supercomputing, SC'08*, 2008.

AUTHOR COPY