Rocblas Users Guide

Greg Mackey, Xiangmin Jiao and Gengbin Zheng

Updated: 12/6/2005

1 Overview

Rocblas provides some basic algebraic operations on data attributes registered with Roccom. Written in C++ using Roccom's developers interface, Rocblas's interface functions take pointers to COM::Attribute objects or scalar numbers as arguments, and should in general be invoked through Roccom using COM_call_function. In the implementation, each process computes on the datasets of its local panes. Most operations implemented in Rocblas are embarrassingly parallel requiring no inter-process or inter-pane communications.

1.1 Requirements and Conventions

All operations of Rocblas share similar requirements and conventions on the operands, listed as follows.

- All operands must have the same base data type.
- All nodal or elemental attributes of an operation must be associated with the same set of nodes or elements.
- At least one of the input operands must be a nodal or elemental attribute.
- For functions taking two inputs and without the **_scalar** suffix (including **add**(), **sub**(), **mul**(), and **div**()), all input operands must be Attribute objects, and one is allowed to be a panel or windowed attribute. In the corresponding functions with the **_scalar** suffix, the second input argument must be a scalar number.
- Each attribute operand has two versions: a scalar version (with only one entry per entity) and a vector version (with only one entries per entity). An attribute operand can be either one, except that for swap and copy, the two attribute operands must have the same number of components.
- Attribute operands with more than one data entry per entity (where the entity is a window, pane, a node, or an element for windowed, panel, nodal or elemental, respectively) must have the same number of entries per entity.
- It is legal to use an attribute with only one entry per entity in place of an attribute with more than one entry per entity. Rocblas will use the value of that single entry to compute against with all entries of another operand.
- In most cases, it is legal to mix contiguous and staggered layouts of nodal and elemental attributes.
- Output arguments are usually the last arguments, except for functions with optional input arguments.

Most of these requirements are checked at runtime by Rocblas if possible. A violation will cause a run to abort. A user, however, can disable error checking at compile time by passing -DNDEBUG to the C++ compiler.

2 Rocblas Interface

2.1 Supported Operations

The following table lists the operations supported by Rocblas, in which we use S, W, P, N, and E to abbreviate different types of operands.

S = scalar pointer
W = windowed attribute
P = panel attribute
N = nodal attribute
E = elemental attribute

$+, -, \times, \div$	$N = N \diamond W$,$N = N \diamond P$,$N = N \diamond N$; $E = E \diamond W$, $E = E \diamond P$,$E = E \diamond E$
scalar $+, -, \times, \div$	$N = N \diamond S$, $E = E \diamond S$
dot	$W/P/N = \langle N,\ N \rangle$, $W/P/E = \langle E,\ E \rangle$
dot-scalar	$S = \langle N,\ N \rangle$, $S = \langle E,\ E \rangle$
2-norm	$W/P/N = \parallel 2 \parallel_2 N$, $W/P/E = \parallel 2 \parallel_2 E$
2-norm-scalar	$S = \parallel 2 \parallel_2 N$, $S = \parallel 2 \parallel_2 E$
swap	$N \leftrightarrow N$, $E \leftrightarrow E$
neg	$N = -N$, $E = -E$
copy	$N = W$, $N = P$, $N = N$, $E = W$,$E = P$, $E = E$
copy-scalar	$N = S$, $E = S$
axpy	$N = WN + N$, $N = PN + N$, $N = NN + N$ $E = WE + E$, $E = PE + E$, $E = EE + E$
axpy-scalar	$N = SN + N$, $E = SE + E$

2.2 Rocblas API

- void **Rocblas_load_mudule**(const char name)
  void **Rocblas_unload_mudule**(const char *name)

  Loads/unloads Rocblas to/from Roccom by creating a window of the given name and register its functions.

- void **add**(const Attribute *x, const Attribute *y, Attribute *z)
  void **sub**(const Attribute *x, const Attribute *y, Attribute *z)
  void **mul**(const Attribute *x, const Attribute *y, Attribute *z)
  void **div**(const Attribute *x, const Attribute *y, Attribute *z)

  Performs the operation $\mathtt{z} = \mathtt{x}\ op\ \mathtt{y}$, where op is $+, -, \times$, or $\div$. The output argument z must be nodal or elemental; one of x and y must have the same type as z, and the other can have the same type or be a windowed or panel attribute. If all operands are nodal or elemental, then the operation is performed node-wise or element-wise, respectively. If one of the operand is windowed, then its value is used in the operation of every node/element. If one of the operand is panel, then its value in a pane will be uses in the operations on the nodes/elements of this pane. It is legal to have the same attribute to appear multiple times in the operands.

- void **add_scalar**(const Attribute *x, const void *y, Attribute *z, const char *swap = NULL)
  void **sub_scalar**(const Attribute *x, const void *y, Attribute *z, const char *swap = NULL)
  void **mul_scalar**(const Attribute *x, const void *y, Attribute *z, const char *swap = NULL)
  void **div_scalar**(const Attribute *x, const void *y, Attribute *z, const char *swap = NULL)
  Performs the operation $\mathtt{z} = \mathtt{x}\ op\ \mathtt{y}$ or $\mathtt{z} = \mathtt{y}\ op\ \mathtt{x}$, if swap is null or not, respectively, where op is $+, -, \times$, or $\div$. Here, the scalar behaves similar to a window attribute in their corresponding functions without **_scalar**. It is a user's responsibility to ensure that the data type of the scalar is the same as the base data types of the attributes. Again, it is legal to have the same attribute to appear multiple times in the operands.

- void **maxof_scalar**(const Attribute *x, const void *y, Attribute *z)
  Performs the operation z = max(x, y). It is a user's responsibility to ensure that the data type of the scalar is the same as the base data types of the attributes. Again, it is legal to have the same attribute to appear multiple times in the operands.

- void **dot**(const Attribute *x, const Attribute *y, Attribute *z, const Attribute *mults = NULL)

  Performs the operation $\mathtt{z} = \langle \mathtt{x}, \mathtt{y} \rangle$. The inputs x and y must be nodal or elemental. The output z can be windowed, panel, nodal, or elemental. If z is windowed, then the result is the dot product for x and y over the whole window. If z is panel, the result is over the each pane. If z is nodal or elemental, then the results are over each node or element (i.e., the value associated with a node is treated as vectors).

  The optional argument `mults` specifies the multiplicity of the nodes or elements, i.e., the number of times a node or pane appears in the window. It is useful only when z is a windowed attribute. The product of of the values associated with a node or element will be divided by its multiplicity before being summed. When no value is passed for `mults`, then a multiplicity of 1 is assumed for each node or element.

  Note: An MPI all-reduce is needed for this operation if the solution is a scalar.

- void **nrm2**(const Attribute *x, const Attribute *y, const Attribute *mults = NULL)

  Performs the operation $\mathtt{y} = \| \mathtt{x} \|_2$. This function works the same as **dot**(), except that it takes only 1 required input argument instead of 2.

- void **swap**(Attribute *x, Attribute *y)

  Swaps the contents of x and y. x and y must be nodal or elemental and must have the same number of entries per entity.

- void **neg**(Attribute *x)

  Negate the signs of the values of x, where x must be nodal or elemental and must have the same number of entries per entity.

- void **copy**(const Attribute *x, Attribute *y)

  Assigns the value of x to y. The argument y must be nodal or elemental. x can be windowed, panel, nodal, or elemental. If x is windowed, then its value is assigned to every node or element in

y. If x is panel, then each node or element of y receives the value of x associated with its pane. If x is nodal or elemental, then each node or element of y receives the value of its corresponding node or element of x.

- void **axpy**(const Attribute *a, const Attribute *x, const Attribute *y, Attribute *z)

  Performs the saxpy operation z = ax + y. x, y, and z must be nodal or elemental. The attribute a can be windowed, panel, nodal, or elemental.

- void **dot_scalar**(const Attribute *x, const Attribute *y, void *z, const Attribute *mults = NULL)
  void **nrm2_scalar**(const Attribute *x, void *z, const Attribute *mults = NULL)
  void **copy_scalar**(const void *x, Attribute *y)
  void **axpy_scalar**(const void *a, const Attribute *x, const Attribute *y, Attribute *z)

  Has the same semantics as their corresponding version without **_scalar**, except that the scalar argument acts in place of a window attribute.

3 Building and Testing Rocblas

The makefile of Rocblas uses Roccom's common makefile for implicit rules and automatic generation of dependencies. To build the library, invoke the makefile using the command "gmake", which will create a file genx/lib/libRocblas.so for dynamic linking or libRocblas.a for static linking.

Rocblas comes with a test program named blastest.C in the test subdirectory. Build the program using the command "gmake blastest". The program takes no command-line arguments. Instead, it will prompt for a user to choose interactively the types and layout of operands and the operations to be tested.