useful to explicitly register the routines that we can call via any of the .C()/.Call()/.External() interfaces. When the routines are registered, R can help to identify potential errors in calling them. R can detect an incorrect number of arguments for a routine, or that the types are incorrect, e.g., an integer vector when a numeric vector is expected. Registration also allows us to resolve the symbols just once rather than each time we call a routine, and it also allows us to use different symbols/names to refer to the routines. It is convenient to be able to run R code to identify the R-callable routines and to generate the registration information programmatically. As we change these routines, we can programatically update the registration information with little effort and ensure the information is synchronized.

Generating bindings to native libraries: Numerous R packages provide interfaces to existing C/C++ libraries. This typically involves manually creating two pieces of code. The first is an R-callable wrapper routine corresponding to each routine of interest in the third- party library. The second is a corresponding R function that invokes the wrapper routine, having coerced the R arguments to the appropriate form. This is often quite straightforward, but both time-consuming and error-prone. This makes for unnecessarily lengthy write-debug- test cycles. Instead, we'd like to be able to programmatically read the information about the third-party routines and data types and then generate all of the code. We want to minimize human intervention. If we could generate these "bindings" programmatically, the R programmer can spend time creating higher-level functionality using these primitives.

Dynamic calls to native routines: The rdyncall (Adler 2012) and Rffi (Temple Lang 2011) R packages avoid having to explicitly create the wrapper routines and R functions to interface to existing C routines. Instead, they both provide a dynamic mechanism to call arbitrary native routines. However, both approaches require a description of the target routines. Again, we want to obtain this information programmatically and then we can easily generate these descriptions and remove humans from the process.

Understanding third-party libraries interactively: When we interface to third-party libraries, we typically read documentation to identify and understand the important routines and data structures. In some situations, it can be convenient to find this information interac- tively within an environment such as R. Rather than reading static document, we can query the code for information such as how often a particular data type is returned by a routine or passed as an argument? or what idioms does the library use? We can use R's graphics capabilities to visualize the code and which routines call which other routines.

In-line documentation as comments: Often third-party native libraries contain documen- tation in comments adjacent to the corresponding routines and data structures. Accordingly, it is convenient to be able to easily access this documentation and potentially reuse it as R documentation for functions that interface to the routines, as we did for the RCUDA package (http://github.com/duncantl/RCUDA).

Compiling R code: Recently, we have been developing R facilities for compiling R code to native instructions to by-pass the R interpreter. This allows us to rewrite and translate R code so that it is essentially native code and can call other existing

native routines, for example, in the R engine or standard C libraries. This results in significant speedup. However, to make this work, we need to know the signature – parameter types and return type – of these native, external routines. Again, if we can find this information programmatically, we greatly simplify and improve the entire process of generating the code. Determining memory management and mutability of inputs and outputs: When we