Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

RCUDA: General programming facilities for GPUs in R

Paul Baines
University of California at Davis
Duncan Temple Lang
University of California at Davis

Abstract

General Purpose Graphical Programming Units (GPGPUs) provide the ability to perform computations in a massively parallel manner. Their potential to significantly speed computations for certain classes of problems has been clearly demonstrated. We describe an R package that provides high-level functionality that allows R programmers to experiment with and exploit NVIDIA GPUs. The package provides an interface to the routines and data structures in the CUDA (Compute Unified Device Architecture) software development kit (SDK), and also provides higher-level R interfaces to these. Currently, programmers write the code that runs on the GPU in C code but call that code and manage the inputs and outputs in R code. We describe experimental approaches to compile R directly so that all computations can be expressed in R.

Keywords: GPUs, parallel computing, R, **RCUDA** package.

1. Introduction

In recent years Graphics Processing Units (GPUs) have emerged as the dominant computational platform for massively parallel computation. Fortunately, unlike CPUs, the performance of GPUs continues to improve dramatically from year to year. While the high-throughput, memory-light paradigm of GPU programming is not well-suited to all problems, it is well-suited to particular classes of computations, some quite common in statistics. As the size of both modern data sets and modern computational requirements continue to grow it is increasingly important for statisticians to be able to leverage the power of GPUs to conduct these computations in an efficient manner.

In presenting the **RCUDA** package we assume the reader has a basic familiarity with GPUs

and the NVIDIA-specific Compute Unified Device Architecture (CUDA) programming model. For an introduction to GPU programming and CUDA see Kirk and Hwu (2012). As we will see in later sections **RCUDA** provides a high-level interface to the CUDA SDK and lessens the burden for R programmers seeking to utilize the power of GPUs. Nonetheless, GPUs are tailored for massively parallel computation and perform best when the programmer understands the fundamentally different computational model required by GPUs. Most general purpose programming on GPUs (GPGPU) is built upon a slightly extended version of the C programming language (e.g., CUDA, OpenCL) and requires the programmer to think about two processing units - the host CPU and the GPU device. When coding in CUDA C the programmer must explicitly allocate and move data from the host to the device and back. This memory management and data transfer introduces a programming burden as well as a performance cost. As we will see in section 2.1, **RCUDA** has the ability to automatically handle much of the programming burden. The performance costs of data transfer to/from the device are problem specific and are an important consideration when designing and optimizing GPU code.

The **RCUDA** package is quite different in intent and functionality than other GPU-related R packages such as **gputools** (Buckner, Seligman, and Wilson 2011) and **rgpu** (Kempenaar and Dijkstra 2010). **gputools** implements several commonly used statistical algorithms in C code that runs on a GPU and provides R functions to invoke those with data in R. The set of functions is fixed and R programmers wanting to implement a different algorithm or approach for one of these algorithms must program in C.

The **rgpu** package also provides implementation of a few algorithms written in C that run on a GPU. However, it also provides an "interpreter" for R scalar mathematical expressions. This does not appear to handle arbitrary code and also has to map each expression from R to a different representation for each call from R to the GPU. It also interprets this on the GPU rather than using native code. We discuss in section 5 how might be able to compile a larger subset of the R language to native GPU code.

The **RCUDA** package is similar in nature to **OpenCL** (Urbanek 2012). Whereas **OpenCL** can in principal be used with all GPUs, **RCUDA** specifically targets the CUDA SDK and NVIDIA GPUs. This is based on both the belief that the dedicated CUDA SDK can outperform the more general OpenCL programming model (Khronos OpenCL Working Group 2008) and the wider popularity of CUDA in the GPGPU community. In addition, **RCUDA** aims to expose the entire SDK to R programmers. The **OpenCL** package provides the essential functionality to invoke kernels on the GPU, passing data from R to the GPU and back. This difference illustrates one of the primary motivations of the **RCUDA** package. We want R programmers to experiment with different features of the SDK and to explore the performance characteristics of various programming strategies for GPUs. Different GPUs exhibit quite different performance characteristics, and different programming paradigms and even tuning parameters also can have significant impact. For this reason, we want to be able to be able to control these aspects dynamically in a high-level language rather than fix them statically in a language such as C.

2. The Basics of the RCUDA package

In this section, we describe both the essential concepts of the CUDA SDK and the R interface to it provided by the **RCUDA** package.

At its simplest, computing with GPUs involves

1. writing (and compiling) a kernel that performs the computations,
2. allocating memory on the GPU device,
3. copying data from the host to the device
4. invoking the kernel
5. copying the results back from the device to the host.

We now discuss each of these steps and how they are handled by **RCUDA**.

The most fundamantal component of any GPGPU code is the kernel. At present we recommend that the kernel be compiled outside of R. This means that an R programmer can write and compile a kernel or perhaps that it be compiled by somebody else and made available to the R programmer. The **RCUDA** package can compile a kernel for the R user but it is typically more convenient to use the command-line to compile the code directly. We now use the *dnorm()* kernel as an example. The code is reasonably simple for a CUDA kernel and is shown in figure 1. Again, we assume the reader has some familiarity with CUDA kernels, and include the code in figure 1 for illustration only. For readers who are less familiar with CUDA C, the key parts of the code are the `__global__` qualifier (which indicates that this is indeed a kernel), the thread/block indexing (handled using standard CUDA conventions) and the actual calculation of the normal density. As written, the kernel is designed to be executed over a specfied number of threads as chosen by the programmer. The indexing portion of the code ensures that each thread operates on a unique element of the input vector (or does nothing if the index is out of bounds).

Before we can invoke a kernel, we have to load it into the host process, i.e. R. We do this by loading the compiled code as a *Module*. We use the R function *loadModule()* to do this. It can read the code in various formats - the PTX text format and the two binary formats cubin and fatbin.

The code can be compiled at the command line using

```
nvcc --ptx -o dnorm.ptx dnorm.cu
```

Alternatively, we can compile the two binary formats with

```
nvcc -cubin -m64 -gencode arch=compute_10,code=sm_10 -o dnorm.cubin dnorm.cu
nvcc -fatbin -m64 -gencode arch=compute_10,code=sm_10 -o dnorm.fatbin dnorm.cu
```

respectively. Note that the *arch* flag specifies the GPU compatability level. Once we have compiled the code, we can load it with

```
filename = system.file("sampleKernels", "dnorm.ptx", package = "RCUDA")
mod = loadModule(filename)
```

Next, we can obtain a reference to the particular routine in the module that we want to invoke. The R interface makes the module appear like a list and so we can use

```
extern "C"
__global__ void dnorm_kernel(float *vals, int N, float mu, float sigma)
{
       // Taken from geco.mines.edu/workshop/aug2010/slides/fri/cuda1.pd
    int myblock = blockIdx.x + blockIdx.y * gridDim.x;
           /* how big is each block within a grid */
    int blocksize = blockDim.x * blockDim.y * blockDim.z;
            /* get thread within a block */
    int subthread = threadIdx.z*(blockDim.x * blockDim.y) +
                         threadIdx.y*blockDim.x + threadIdx.x;

    int idx = myblock * blocksize + subthread;

        if(idx < N) {
            float std = (vals[idx] - mu)/sigma;
            float e = exp( - 0.5 * std * std);
            vals[idx] = e / ( sigma * sqrt(2 * 3.141592653589793));
        }
}
```

Figure 1: The C code defining a GPU kernel to compute the Normal density. This kernel writes the results back into the input vector, overwriting its values.

```
kernel = mod$dnorm_kernel
```

to get this reference.

We now have the kernel, so before executing it we need the data to pass to it. Here for simplicity we just simulate the data in R via the *rnorm()* function.

```
N = 1e6
mean = 2.3
sd =  2.1
x = rnorm(N, mean, sd)
```

As outlined at the beginning of section 2, memory needs to be allocated on the device, and the data copied from host to device before invoking the kernel. However, using the R interface to the GPU provided by **RCUDA** we can simply call the kernel with host data using the *.cuda()* (or the synonomous *.gpu()*) and the R interface will allocate memory and copy the R vector argument to the device without user input. In short, we can invoke the kernel with

```
ans = .cuda(kernel, x, N, mean, sd, gridDim = c(62, 32), blockDim = 512)
```

Note that we must explicitly specify the dimension for the grid and block to be used when launching the kernel. Both arguments can take positive integer inputs of up to three-dimensions with any omitted dimensions defaulting to 1. We chose 62, 32 and 512 for this example as the product of these exceeds the number of elements N. Since the number of threads exceeds the number of elements to be operated on, there will be some redundant

threads on the GPU (as determined by the `if` statement in figure 1). Since thread launches are exceptionally cheap on GPUs, this over-saturation strategy is standard practice for GPU programming.

The *.cuda()* function recognizes the inputs and determines which to copy to the device and which can be passed directly (i.e. the scalar values *N*, *mean* and *sd*). As we will see later we can pass arguments that refer to data or memory already on the GPU and *.cuda()* recognizes that it doesn't need to transfer this, but merely pass the reference as-is. The option of using either host or device resident arguments in the call to *.cuda()* provides flexibility and simplicity while retaining the ability to produce highly efficient code that minimizes data transfers between the host and device.

When calling the *.cuda()* function with an R vector as an input argument, *.cuda()* recognizes that since the data was passed from R, it may be modified by the kernel and thus returns the vector. If there are multiple vector inputs, *.cuda()* returns all of these as a list in the same style as the *.C()* function. However, *.cuda()* also recognizes if only one input argument is a vector, and thus *ans* in our example also contains the actual vector of normal density values. More generally, we can specify which inputs are to be copied back to R from the device via the **outputs** parameter of the *.cuda()* function.

We conclude this simple example by noting that *.cuda()* executes the kernel calls using the current context which we describe in section 2.2. This same context must be used to both load the module and call any of its kernels.

2.1. Manually Allocating Memory on the Device

While the *.cuda()* function processes (most) R vectors for us, we may want to explicitly control how values are passed from the host (CPU) to the device (GPU). The **RCUDA** package provides functions to control this and also some short-hand, covenient mechanisms for implementing the transfers. The two fundamental functions are *copyToDevice()* and *copyFromDevice()*. These are reasonably flexible functions. *copyToDevice()* takes an R object and copies its contents to memory on the GPU device. By default, it allocates the space on the device, using information about the number of elements and the type of each element in the R object to determine the space needed. For example, we can copy an R vector to the GPU in the following manner:

```
dev.ptr = copyToDevice(x)
```

We can also explicitly allocate space on the device and then copy values to that space. The function *cudaMalloc()* (and its alias *cudaAlloc()*) allocates space on the device. We pass it the number of elements and either the size of each element (in bytes) or the name of the element type which it looks up to determine the number of bytes for each element. We can explicitly allocate memory and pass this as the destination target for *copyToDevice()*, e.g.

```
ptr = cudaAlloc(N, elType = "numeric")
copyToDevice(x, ptr)
```

Why would we want to explicitly allocate space on the device rather then letting R copy vectors for us? The most common situation in which this is desirable is when we want to allocate space on the device once and then reuse it to store different values at different times.

For example, consider a simple non-parametric bootstrap in which we generate many samples of the same length by sampling with replacement and apply some kernel to each generated dataset. In this context we can reuse the same memory for each bootstrap dataset. In the code below we demonstrate how to allocate the space once, copy the boostrap data to it each iteration and then compute the necessary summaries via a call to a GPU kernel, e.g.,

```
ptr = cudaMalloc(length(x), elType = class(x))
replicate(B, {
      copyToDevice(sample(x), ptr)
      .cuda(kernel, ptr, N, mean, sd, gridDim = c(62, 32), blockDim = 512))
})
```

By default, *copyToDevice()* uses *cudaMalloc()* to allocate the space on the device. *cudaMalloc()* returns an object that points to the allocated memory, but also contains the number and the type of the elements, if specified. This allows us to retrieve the contents of the memory on the device. The *copyFromDevice()* function allows for explicitly copying data from the device to host. It takes the pointer, the number of elements and the type of each element. While we can specify these arguments explicitly, in many cases it is more convenient to use the subset operator

```
p[]
```

This utilizes the information stored when we allocated the space and is equivalent to

```
copyFromDevice(p, p@nels, p@elTypeName)
```

We can also use the subset syntax to assign values to an existing memory location on the device. For example,

```
p[] <- rnorm(N)
```

copies the the values on the right-hand side to the device memory specified by `p[]`.

In addition to this standard usage it is also worth noting that we can use *cudaMalloc()* to allocate space for arbitrary data types since we only need the size of each element. For known data types such as `float` or `int` values, **RCUDA** knows how to copy data both to and from the device. For more general data types this can be done by the R programmer using the functionality provided by **RCUDA**

When we no longer have an R reference memory on the device (i.e. in an R variable), R releases the memory using a finalizer routine we register when allocating the space. This allows R programmers to ignore memory management issues and treat the memory on the device as a regular R object.

2.2. Contexts

We can call many of the functions in the **RCUDA** package without having to know about CUDA context objects. However, they are important for some computations and so we address them briefly here.

Most **RCUDA** functions require an active context. We can explicitly create a default context using *createContext()* or we can use *cuGetContext()* to query the current context and create one if there is none. When creating a context, we can specify the device with which it is associated and also specify different flags or options to control how it behaves. These options are a combination of individual options which we can specify in different ways. We can use R variables representing the different options, e.g. *CU_CTX_SCHED_AUTO* and *CU_CTX_MAP_HOST*, and then we can combine them with the | operator. Alternatively, we can use a vector of names that identify the different options. The following are equivalent

```
c("SCHED_AUTO", "BLOCKING_SYNC", "MAP_HOST")
c("CU_CTX_SCHED_AUTO", "CU_CTX_BLOCKING_SYNC", "CU_CTX_MAP_HOST")
CU_CTX_SCHED_AUTO | CU_CTX_BLOCKING_SYNC | CU_CTX_MAP_HOST
```

Note that the first variant avoids the `"CU_CTX"` prefix. Similarly, there are also functions that can query and set attributes of a context such as the stack and heap size, shared memory configuration and cache configuration.

CUDA maintains a stack of contexts (per host thread, of which there is only one in R). When we create a context, it becomes the active one and is used in subsequent computations. We can also explicitly push an existing context on to the top of the stack with *cuCtxPushCurrent()*. We can pop the current context off the top of the stack with *cuCtxPopCurrent()*. At present, R users must explicitly release a context with *cuCtxtDestroy()*. It would be preferable to release the memory using R's finalizer mechanism. However, this is, at best, complex because it is difficult to determine if CUDA is still using the context.

2.3. Querying Devices

The function *getNumDevices()* tells us the number of devices on the local machine. We can get the name of a device with *cuDeviceGetName()*, e.g.

```
cuDeviceGetName(1L)
```

```
[1] "GeForce GT 330M"
```

We can query the characteristics of a device using the *getDeviceProperties()* function or *cuDeviceGetAttributes()*. The former uses a deprecated routine in the CUDA API, while the second queries all of the individual attributes. The queryable attributes include the maximum dimensions of a grid and a block, the maximum number of threads per block and per processor, the warp size, the number of multi-processors and the shared memory per block. The names of the entire set of attributes are available via the *CUdevice_attributeValues* variable in the package. Rather than querying all of the attributes in one call, we can retrieve a single attribute with *cuDeviceGetAttribute()*. We specify the attribute using the name or value of one of the elements in *CUdevice_attributeValues*.

In addition to querying information about a device, we can query the version of the CUDA SDK with *cudaVersion()* e.g.,

```
cudaVersion()
```

```
 driver runtime
   5000    5000
```

We can determine the amount of memory a device has with the function *cuMemInfo()*. Unlike the other functions for querying a device, *cuMemInfo()* does not take a device as an argument. Instead, it uses the device associated with the current context. Thus we must have created a context before calling this function, either explicitly or implicitly. The function reports the total memory on the device, the amount free and the proportion free.

2.4. Profiling

The primary reason to use GPUs for scientific computing is usually to improve performance. Key considerations in the performance of GPU code include the overhead of copying data between the host and the device, the efficiency of the kernel code and even the dimensions of the grid and block for controlling the threads. In light of this, it is useful to understand where the entire code spends time in order to improve the performance by reducing these bottlenecks. As in R itself, we can profile computations involving the GPU with several routines in the CUDA API.

We can profile an entire sequence of R expressions with the R function *profileCUDA()*. This takes one or more R expressions (enclosed within `{}` for more than one expression) and returns the profiling information as a data frame. The following bootstrap example allocates space and reuses it for each bootstrapped data set, with GPU code used to compute summary statistics for each sample:

```
B = 100
prof = profileCUDA( {
  p = copyToDevice(x)
  replicate(B, {
      p[] = sample(x, N, replace = TRUE)
      .cuda(kernel, p, N, mean, sd, outputs = 1,
                   gridDim = c(62, 32), blockDim = 512)
  })
})
```

This returns a data frame with one row for each line of output generated by the CUDA profiler. Each row corresponds to a particular CUDA routine, kernel routine or routine called by the kernel. There are typically multiple rows for the same routine corresponding to when the routine was observed by the profiler. The columns of the data frame correspond to the "counters" we specified for the profiler to record. We specify this via a configuration file the profiler reads when it is instantiated. The **RCUDA** package provides a default configuration file and uses that if the caller does not specify one. We can use a different configuration file either by explicitly passing a file name in the call to *profileCUDA()* via the **config** parameter, or globally via the R option "*CUDAProfilerConfig*".

Rather than profiling an entire collection of R expressions as a single unit, we can create a profiler and start and stop it at different times to collect information about particular R expressions. The *cudaProfiler()* function creates the profiler, and optionally starts it. We specify the name of a file to which the profiler writes the information, and we can select

either of two formats - CSV or name=value pairs. We start and stop the profiler with *cudaStartProfiler()* and *cudaStopProfiler()*. We can read the CSV output from the profiler with *readCUDAProfile()*. Note that *profileCUDA()* is merely a wrapper that uses all of these functions.

The *summary()* method for the result of *profileCUDA()* and *readCUDAProfile()* aggregates the rows in the data frame for each routine and gives the total counts for each as a new data frame.

2.5. Memory Management

As discussed in section 2, the **RCUDA** package provides both high- and low-level functionality to transfer data between the host and the GPU device. In this section we describe some of the richer memory management functionality offered by the package. As may be expected, the low-level functionality mirrors the CUDA API with the ability to allocate memory on the device in several different ways, copy memory between the host and device, and to release the memory. These include *cudaMalloc()* and *cudaMallocPitch()* for allocating memory and *cudaMemcpy()* for copying the contents of memory between the two devices. *cuMemFree()* releases the memory. We can use these primitives directly or alternatively we can use higher-level functionality provided by the package.

In a call to *.gpu()* (or *.cuda()*), any R vector with more than one element is automatically transferred from R to the GPU. By default, after the kernel execution has completed, the contents of this memory on the GPU are then transferred back to R and returned by the *.gpu()* call. The memory is then automatically released. This is done via a finalizer on the external pointer. Typically if a vector argument does not contain any useful output data from the kernel then we would not want *.gpu()* to spend time transfering its contents back from the GPU. Fortunately we can avoid this by specifying which arguments are to be considered outputs from the kernel via the **outputs** paramater of *.gpu()*.

As discussed in section 2, rather than have *.gpu()* implicitly transfer data to the device, we can also explicitly control the transfer of data from R to the GPU with *copyToDevice()*. Manually copying data from host to device is often preferable if we are applying one or more kernels to different R vectors or matrices with the same number of elements. The *copyToDevice()* function returns an object that is an instance of the class *cudaPtrWithLength()*. These objects have the address of the memory on the GPU in which the data are stored. However, they also contain the number of elements and the size of each element on the GPU. This allows us to retrieve all of the values in the array from the GPU as the object is fully self-describing (unlike a simple pointer). We can use the simple empty-subsetting operator to copy the contents from the GPU back to R. For example, the following shows how to copy an R vector to the GPU and retrieve it:

```
x = rnorm(1e6)
ptr = copyToDevice(x)
ptr[]
```

The final expression retrieves the collections values from the GPU as into a new R vector, separate from the original vector in *x*. We can also a subset of the elements in the usual manner, e.g.

```
ptr[1:10]
ptr[which(x) > 0 ]
ptr[ - c(11:1e6)]
```

We can explicitly free memory we allocate on the GPU with *cudaMalloc()* via the function *cuMemFree()*. However, R will also do this for us via the usual garbage collection. When an R object referencing memory on the GPU is deleted, the finalizer routine calls *cuMemFree()* for us. We have to remember to remove such variables when we no longer need them as otherwise the memory will not be released. We also have to wait until the R garbage collector is invoked for the GPU memory to actually be freed. This is why the *.gpu()* function explicitly calls *gc()*, by default. If it did not, it is possible that memory on the GPU used from a previous call to *.gpu()* would still appear to be in use and so not released. This could have lead to situations where there was insufficient memory on the GPU to perform this new call to *.gpu()*, when in practice we could have used the memory from the previous call.

In addition to the *cudaMalloc()* function, there are other functions available for memory allocation and data transfer to/from the GPU such as *cudaMallocPitch()*, *cudaMemcpy2D()* and *cudaMemcpy3D()* which allow for the transfer of 2-dimensional and 3-dimensional arrays. Further discussion of some aspects of this functionality is provided in 3.3.

2.6. Asynchronous execution

A typical mode of using the GPU is to launch a kernel and wait for it to complete and then return the results to the R calling command. We can, however, launch the kernel asynchronously. This allows us to dispatch one or more tasks to the GPU while we continue to perform tasks in the R session. This allows us to truly use the GPU as a co-processor.

There are several ways to excute tasks asynchronously. The simplest is, perhaps, to use the **.async** parameter in the *.gpu()* function. If this is **TRUE**, the *.gpu()* function launches the kernel and does not wait for it it to complete. It returns the references to memory on the GPU device for any objects it copied to the GPU. At this point, we can evaluate other R expressions while the GPU is processing the kernel threads. To obtain the results from the kernel, we can simply copy the values from the GPU memory. Copying the memory from the device will cause CUDA to synchronize with the GPU. Alternatively, we can explicitly synchronize with the GPU by calling *cudaDeviceSynchronize()* or *cuCtxSynchronize()*.

A different approach than using *.async()* is to use streams. We can think of a stream as managing a collection of tasks. A task can be an invocation of a kernel or copying data to or from the GPU. We can add a kernel invocation to a stream by passing the stream object to the *.gpu()* function. We first create the stream with

```
stream = cudaStreamCreate()
```

We then launch a kernel with

```
out = .gpu(mod$kernel, ..., stream = stream)
```

By default, the presence of the **stream** argument will cause **.async** to be **TRUE**and *.gpu()* will return without waiting for the kernel to complete.

We can queue multiple kernel invocations on the same stream via repeated calls to *.gpu()*. The GPU can interleave and schedule the threads across these kernels in a way that improves the overall performance relative to serial scheduling.

We can also add requests to a stream to copy memory to or from the GPU. For example, the *cudaMemcpyAsync()* accepts a **stream** argument, as do any of the Async variants of the memory copying functions.

While we can synchronize on the device or context, we can also use more fine-grained synchronization using a stream. The *cudaStreamSynchronize()* function will wait in R until the entire collection of tasks in the stream have completed. Rather than blocking until the tasks are complete, we can use *cudaStreamQuery()* to query whether the stream's task are all finished. This allows us to periodically "check in" with stream without actually waiting for it to complete.

2.7. Other Routines in the API

For the most part, there is a corresponding R function for each routine in the CUDA SDK that can run on the host. At present we have not created bindings to the routines associated with textures and surfaces. However, should these prove useful for the community, these bindings can be generated in a similar manner.

3. Examples/case-Studies and performance

In this section, we look at several examples and see how they perform on various GPUs. The first example is a simple scalar version of the normal density function and we compare this with the highly efficient *dnorm()* function in base R. We also look at a similar example involving importance sampling. This is similar as it maps an element-wise vectorized operation in R to element-wise threads on a GPU. We then look at computing distances on a GPU by addapting code in the **gputools** package. This illustrates how we can use an existing kernel and invoke it from R without the supporting C code to interface from R.

3.1. Computing the Normal density

Performance in R puts a great premium on vectorized operations. Since most of the primitives are vectorized, expressions that use these primitives are also vectorized. Since these primitives are vectorized in C code, they are fast. Some of them also use multiple processors when the vectors are sufficiently large and there are additional available CPUs.

Element-wise vector operations in R map naturally to GPUs. We perform the same computation on each element of the vector in a separate thread on the GPU. We define a kernel to operate on an individual element and then we can apply that to each element of the vector.

Consider the function *dnorm()* in R. It computes the value for a Normal density a vector of values, e.g.

```
N = 1e6L
x = rnorm(N)
d = dnorm(x, mu, sigma)
```

We can define a similar kernel routine to compute the density for a single observation. The code will look similar to the following:

```
extern "C"
__global__ void
dnorm_kernel(float *vals, int N, float mu, float sigma)
{
    int myblock = blockIdx.x + blockIdx.y * gridDim.x;
    int blocksize = blockDim.x * blockDim.y * blockDim.z;
    int subthread = threadIdx.z*(blockDim.x * blockDim.y) +
                      threadIdx.y*blockDim.x + threadIdx.x;

    int idx = myblock * blocksize + subthread;
    float pi = 3.141592653589793;

    if(idx < N) {
       float std = (vals[idx] - mu)/sigma;
       float e = exp( - 0.5 * std * std);
       vals[idx] = e / ( sigma * sqrt(2 * pi));
    }
}
```

The initial expressions determine on which element of the vector this particular thread should operate. It does this by computing its unique identifier in the grid of blocks of threads. Once it has computed the index of the element, it verifies that this is not a redundant thread but is operating on a value within the extent of the vector. Then it computes the result.

In our kernel, we have passed the vector of values via the first parameter. We have arranged for the kernels to write their results back into this vector since we don't expect to reuse this veector on the device. This removes the need to have two related vectors for the inputs and outputs in memory on the device simultaneously.

We compile this code, either on the command line outside of R or with

```
ptx = nvcc('dnorm.cu')
```

We can then load the resulting PTX file with

```
mod = loadModule(ptx)
```

Now that we have the kernel, we can obtain a reference to the kernel with

```
k = mod$dnorm_kernel
```

We can now apply this kernel to our vector above with

```
ans = .gpu(k, x, N, mu, sigma, gridBy = N)
```

The *.gpu()* function recognizes which arguments are local vectors (not scalars) and, by default, returns the updated contents of these vector arguments. Therefore, this call to *.gpu()* returns the new contents of *x*.

Note that N has to be an integer. If it is not, we must coerce it to an integer in the call or before. The types of each argument must match the corresponding type of the kernel.

We also created a kernel that does not overwrite its inputs. This has the signature

```
void dnorm_kernel(float *vals, int N, float mu, float sigma, float *out)
```

where the *out* is the vector into which each kernel instance stores its result. We would invoke this in a similar manner as the previous kernel, but here we have to pass this extra argument. We can just allocate this output vector with a call to *numeric()* and pass it in the call to `.gpu`. The command

```
ans = .gpu(k, x, N, mu, sigma, out = numeric(N), gridBy = N)
```

returns the update values of both vectors in the call, i.e. x and **out**. This involves more computation and memory than is necessary. We don't need the contents of x. We use the **outputs** parameter to specify which vectors to explicitly transfer. We can use the name of the argument or its position (in the collection of kernel arguments). For example, to return only the vector associated with the variable named **out**, we use

```
ans = .gpu(k, x, N, mu, sigma, out = numeric(N),
           outputs = 'out', gridBy = N)
```

Note that this is different from

```
ans = .gpu(k, x, N, mu, sigma, out = numeric(N),
           gridBy = N)$out
```

as we might use in R's *.C()* interface. The reason this is different is that *.gpu()* will transfer and return both x and *out* and then we are extracting only the *out* element. By explicitly specifying which objects to transfer in the call to *.gpu()*, we avoid the overhead.

TODO: Do we want timings here? If we don't then I think we need to say we don't provide them...

3.2. Importance Sampling

Importance sampling is a standard approach that allows for the numerical approximation of expected values that cannot be computed directly. Suppose we want to compute $E_f[h(X)]$ where f denotes the probability density (or mass) function of the random variable X i.e.,

$$E_f[h(X)] = \int h(x)f(x)dx. \tag{1}$$

If we are able to sample independent random variates $x_1, \ldots, x_n$ from f then we can approximate the expectation in (1) in a straightforward manner using:

$$E_f[h(X)] = \frac{1}{n}\sum_{i=1}^{n} h(x_i). \tag{2}$$

Now suppose it is difficult or computationally intensive to sample from the density f. Instead, we can sample $x_1, \ldots, x_n$ from another density g with the same support as f. By a simple importance weighting scheme we can modify the approximation in (2) to obtain

$$E_f[h(X)] = \frac{1}{n}\sum_{i=1}^{n} w(x_i)h(x_i), \tag{3}$$

where $w(x_i) = f(x_i)/g(x_i)$ is the importance weight given to x_i.

We now apply this technique to a simple example from Mathew Shum (www.hss.caltech.edu/~mshum/gradio/simulation.pdf). Suppose we want to compute the expectation of a truncated standard Normal with support $[0, 1]$. Note that the density of the truncated Normal is

$$f(x) = \frac{\phi(x)}{\int_0^1 \phi(x)}dx$$

where $\phi(x)$ is the standard normal density. Instead of sampling from $f(x)$, we instead sample N variates from a standard Uniform i.e., we select $g(x) = 1_{\{0<x<1\}}$. Since $g(x) = 1$ the importance weights in (3) can just be seen to be $w(x_i) = \phi(x_i)/.34134$. This can be easily implemented in R using

```
N = 1e6
x = runif(N)
mean(x * dnorm(x)/.34134)
```

Like the *dnorm()* example, we could also implement this vectorized computation in R as a computation on a GPU with each thread calculating $x_i w(x_i)$.

```
extern "C"
__global__ void truncNorm(float *out, float *unifVals, int N)
{
    int myblock = blockIdx.x + blockIdx.y * gridDim.x;
           /* how big is each block within a grid */
    int blocksize = blockDim.x * blockDim.y * blockDim.z;
            /* get thread within a block */
    int subthread = threadIdx.z*(blockDim.x * blockDim.y) +
                    threadIdx.y*blockDim.x + threadIdx.x;

    float phi0_1 = 0.3413447460685;
    int idx = myblock * blocksize + subthread;
    if(idx < N) {
       out[idx] = unifVals[idx] * dnorm(unifVals[idx], 0, 1)/phi_0_1;
    }
}
```

As is typical with a GPU kernel, the code determines which part of the data this particular instance of the kernel is to work on. It does this using its threadIdx and blockIdx and the grid and block dimensions. The actual computations are very simple

```
out[idx] = unifVals[idx] * dnorm(unifVals[idx], 0, 1)/phi_0_1;
```

N	Quadro 600	Tesla K20
1e6	1.16	
1e7	1.61	

Table 1: **Speedup for simple importance sample.** These are relative speedups of the GPU implementation relative to the vectorized R implementation. The Quadro 600 GPU has 1 gigabyte of RAM and is more of a graphics card than a GPGPU. The Tesla K20 has 5 gigabytes of RAM and is one of the top-of-the-line GPUs at present.

We omit the code for the *dnorm()* routine since it is very similar to that in the previous example. Full code is available at http://www.omegahat.org/RCUDA/importanceSample.cu, along with some additional kernels.

We compile this code and load it with

```
nvcc('importanceSampling.cu')
mod = loadModule('importanceSampling.ptx')
u = runif(N)
z = .gpu(mod$truncNorm, numeric(N), u, as.integer(N),
         gridDim = c(64, 32), blockDim = 512, outputs = 1L)
```

Here we have determined that we can run 512 threads in each block and so determine that we need `ceiling(N/512)` blocks to compute all N values.

Do we gain much from using the GPU in this circumstance? We have had to write the kernel in C and compile it. Undoubtedly, this is more complicated than the R code above and involves debugging, etc. What we do get is the parallelism. Table 1 shows timings on different GPUs for different length vectors.

We note that we have elected to compute the random values from the uniform density in R and explicitly pass these to the kernel. Alternatively, we could use the CUDA random number generators. This avoids transferring data from R to the GPU memory. It also allows us to compute these values in parallel. Examples involving the generation of random numbers on the GPU using the CURAND library (NVIDIA corp) are also available in the `examples` directory of **RCUDA**.

3.3. Computing distances

In this final example, we will explore how to compute distances on the GPU. Since the **gputools** package already does this, we here consider how to use the kernels provided by that package within the **RCUDA** framework. Our aim is to contrast how interaction with the GPU is done with R bindings to CUDA and how it is done with C code.

The *gpuDist()* function in the **gputools** package is similar to R's own *dist()* function. It takes a matrix of observations and the name of a distance metric. For ease of computation in the C code, the function transposes the matrix so that the elements of each observation are contiguous in memory (i.e. in row order rather than column order). *gpuDist()* calls a C routine *Rdistances()*. This is a simple wrapper that calls the routine *distance()*. This routine copies the data from the host to the device, allocating the space for the input and the output arrays. It then calls *distance_device()* which launches the kernel. Which kernel is used depends on which distance metric is to be used. So *distance_device()* contains the same code for each

kernel. It is here that a kernel thread is launched for each pair of observations and the GPU is actually involved in the computations.

We can invoke any of the kernels in the **gputools** packages directly from R and so remove the need for the C routines described above. This not only greatly simplifies the understanding and development of the code, but also leads to more flexible, reusable code. We now illustrate how to call the *euclidean_kernel_same()* kernel. We start by extracting it from the C code and compiling it directly into PTX code (or cubin or fatbin format). We then load this into the R session with *loadModule()*. This kernel expects a single matrix, say stored in *AB*. Mimicing the C code in **gputools**, we pass the same matrix as two different parameters. (This is related to a different kernel we will discuss below.) Since we are passing it twice, we can transfer it just once to the GPU memory and then pass a reference to that one instance of the data in two places. This avoids making two copies of the data. As a result, we copy the data to the device before calling *.gpu()* with

```
ABref = copyToDevice(t(AB))
```

Note that we have transposed the matrix as the kernel expects the elements of each observation to be contiguous. This contrasts with R's column-oriented representation of a matrix.

Having loaded the module and extracted the function, we can now launch the kernel. We pass the reference to the matrix values (*ABref*) and also the stride/pitch between observations (the number of columns in the matrix) and the number of observations. Note that we pass these twice, even though the second set of parameters is not used in the kernel. We also pass space to insert the distances. This is an $n \times n$ matrix (where n is the number of rows in AB) and we can pass a simple numeric vector with the correct number of elements for this. We'll convert it to a matrix after the kernel completes. We also need to specify the stride/pitch for this matrix which is `nrow(AB)`, i.e. the number of columns in the distance matrix.

```
d = .gpu(kernel,
         ABref, ncol(AB), nrow(AB),            # inputs & dimensions
         ABref, ncol(AB), nrow(AB),            # inputs again
         ncol(AB),                             # dimension of matrix
         dist = numeric(nrow(AB)^2), nrow(AB), # results
         2.0,                                  # ignored
         outputs = "dist",                # which arguments to copy back
         gridDim = c(nrow(AB), nrow(AB)), # square grid
         blockDim = 32L)                  # threads within block
```

The final three arguments in this call to *.gpu()* are for controlling *.gpu()* rather than being passed to the kernel. **outputs** identifies which inputs(s) are to be copied back as part of the result. These are the out variables of the kernel. In our case, we only want the distances which we have explicitly named **dist**. We run a separate block of threads for each pair of observations. To do this, we specify a grid dimension of `c(nrow(AB), nrow(AB))`. The kernel uses 32 threads to compute the squared differences of the elements in the pair of observations and add them together. This number is hard-coded in the kernel in order to use shared memory between the cooperating threads. Therefore, we use a block dimension of 32.

Once the call to *.gpu()* returns, the variable d contains the vector of all pairwise distances. These distances are converted to a matrix to produce the desired result. Once completed,

we should ideally remove the variable *ABref* so that the garbage collector can release the associated memory on the GPU device.

Note that since the second set of inputs are not actually used in this kerne, we could also specify `NULL` for the matrix of values and any integer values. *.gpu()* transfers the R value to the kernel as the C value *NULL*. Ideally, the code should be changed to remove the parameters that are not used. However, this happens when the code itself is complex to develop and maintain and we don't want to modify it once it is working. This is an argument for writing in a high-level language such as R and **RCUDA**.

3.4. Different approaches to allocating memory

One of the goals of writing the **RCUDA** package is to allow R programmers to explore and experiment with the entire CUDA API. Writing C code for all aspects of copying data between the host and device and launching kernels is time-consuming and error-prone. It makes one less likely to experiment with different approaches to enhance performance. However, if we can perform experiments directly in R code, we are more likely to take the time to understand what idioms work best in different circumstances.

One example of using a different idiom arises in the **gputools** package. Rather than just directly copying the matrix of values from R to the GPU's memory, the C code uses *cudaMallocPitch()* and *cudaMemcpy2D()* to transfer the data. This has a potential benefit of padding out the rows of the matrix so that they are aligned in a way that makes the GPU more efficient. The GPU can then load different blocks of memory in a more efficient manner.

We can change our implementation to see if this does yield improved performance. We can replace the call

```
ABref = copyToDevice(t(AB))
```

in our initial example with an explict call to *cudaMallocPitch()* and *cudaMemcpy2D()*. We do this with

```
mem = cudaMallocPitch( ncol(AB) * 4L,  nrow(AB))
cudaMemcpy2D(mem$devPtr, mem$pitch,
             convertToPtr(t(AB), 'float'),
             ncol(AB)*4L, ncol(AB)*4L, nrow(AB),
             RCUDA:::cudaMemcpyHostToDevice)
```

Note that we explicitly convert the (transposed) matrix to a C-level array of `float` elements.

We had to explicitly create this array as the memory we allocated has no information about the type. The **RCUDA** package provides a higher-level interface for this, built on these SDK bindings. We can use *mallocPitch()* (without the cuda prefix) and specify the type of element with

```
mem = mallocPitch( ncol(AB), nrow(AB), "float")
```

With this function, we specify the number of elements and not the number of bytes in the first argument. Furthermore, the resulting object returned by *mallocPitch()*, of class *PitchMemory2D*, contains information about the location of the allocated memory, its size and type of

element. This allows us to simply assign an R object to it and have the assignment actually convert the R object and copy it to the GPU's memory. So we can use

```
mem[] = t(AB)
```

to copy the matrix to the GPU.

3.5. Distances between two matrices

For certain statistical applications it is useful to compute the distance between each observation of two matrices, say A and B. Given the existing distance function code one way to do this is to combine the two matrices into a single matrix and compute all pairwise-distances between the rows. While simple to implement, such an approach is clearly sub-optimal. First, this approach uses much more memory by creating a new matrix. It also performs a large number of redundant computations that will be ignored, namely the distances between each pair of rows in A and also in B where we only want those for the observations between A and B.

To avoid these inefficiencies, the **gputools** package contains a kernel that can handle two different matrices as inputs. This is called *euclidean_kernel()* (i.e. without the *_same()* suffix.) We can invoke this with

```
out = .gpu(mod$euclidean_kernel,
           t(A), ncol(A), nrow(A),
           t(B), ncol(B), nrow(B),
           ncol(A),
           ans = numeric(nrow(A) * nrow(B)), nrow(A),
           outputs = "ans",
           gridDim = c(nrow(A), nrow(B)),
           blockDim = 32L)
```

The arguments are very similar to the previous kernel call, but there are no ignored parameters.

Note that here we pass the two matrices directly in the call to *.gpu()*, rather than than copying them to the device ahead of the call. This is because we are not passing the same matrix as two separate inputs. Instead, we can leave it to the *.gpu()* function to copy the vectors to the device, pass the references to each of them to the kernel, and then to release the memory after it is no longer used.

Passing the two matrices separately rather than stacking them into one single matrix avoids unnecessary overhead in R before invoking the GPU kernel. Ignoring this overhead, this form of the computation that avoids the redundant computations of within-matrix distances runs 3 times faster for a pair of $10,000$ and 5000 matrices than the GPU version of the computation involving stacking.

3.6. A different metric

With the flexibility provided by the **RCUDA** package it is straightforward to use different distance metrics. To do so, we can simply load a different kernel and pass that to our

function. For example, we can use the *canberra_kernel()* in the **gputools** code. We can access it with

```
mod$canberra_kernel
```

and then pass that to the call to *.gpu()* function above without changing any of the other arguments. In this way, the kernel acts like a function or a pointer to a routine in C and allows us to parameterize the same distance calculations. This makes the code simpler, as we would expect of a high-level implementation. It also makes it more flexible. If we want to use an entirely different kernel, we can write just the code for that kernel, compile and load it. Our distance function does not need to know where the kernel came from or have any a priori knowledge about it.

3.7. Reusing the computed distances

As we have seen before, it is sometimes useful to be able to leave the results computed by one kernel on the device and have those be used as inputs to a second kernel. The **gputools** package does this when performing hierarhical clustering. Starting with a collection of observations (i.e. a data frame or matrix), we calculate the pair-wise distances between all observations. Then we pass these to the *gpuHclust()* function to compute the clusters. If we just use *gpuDist()* and then *gpuHclust()*, we wil have moved the distances from the GPU device back to R, and then copy them back to the device. **gputools** provides *gpuDistClust()* to avoid this unnecessary overhead. This is implemented with a separate, but very similar, routine to *distance()*. Again, this allocates and copies the inputs on the GPU. So there is a separate R function and a separate C routine to implement this, adding to the complexity.

We can also avoid the overhead in **RCUDA** using code of the form:

```
distances = cudaMalloc(N["A"] * N["B"], elType = "numeric")
.gpu(mod$euclidean_kernel_same, .., distances, outputs = FALSE)
.gpu(mod$centroid_kernel, distances, ...)
```

Here, we allocate the memory on the GPU for holding the computed distances. We tell *.gpu()* not to return it to R. At this point, the distances will be in this memory. We then pass the reference to that memory on the device as the input to the next kernel. This reduces the complexity and also allows the R programmer to combine the kernels in different ways than the original developers planned.

4. Low-level interface and its implementation

We initially implemented the bindings to copy data between the host (CPU) and device (GPU) and the ability to launch a kernel to verify that this approach would work and to deal with the marshalling of R objects to the kernel calls and vice-versa. We also added some additional bindings to query the available GPUs and their properties. Subsequently, however, we programmatically generated the binding (R and C) code and the enumerated constants and information about the data structures (e.g. the size of each so we can use that when allocating memory on the device).

Next, we describe a reasonably simple example of a programmatically generated binding to illustrate the nature of the interface and to help mapping from the CUDA documentation to the R functions. One function is *cuDeviceGetAttribute()* and corresponds to the routine with the same name in the CUDA API. The routine is defined in the header file **cuda.h** as

```
CUresult CUDAAPI
cuDeviceGetAttribute(int *pi, CUdevice_attribute attrib, CUdevice dev);
```

The second parameter identifies which attribute we are querying and the third parameter identifies the device. The latter is an integer, with 0 corresponding to the first device, 1 for the second and so on. The first parameter is a pointer to a single integer. This is how the CUDA API returns the result of the query. This is an output variable. Accordingly, we do not need to include in the R function that calls the C routine.

The CUDA routine returns a status value of type `CUresult`. We check this to see if the call was successful. If it was, we return the result stored in *pi*; otherwise, we raise an error in R, using the particular value of the status type and the CUDA error message.

The R function we generate to interface to the CUDA routine is defined as

```
cuDeviceGetAttribute <-
function( attrib , dev  )
{
   ans = .Call('R_auto_cuDeviceGetAttribute',
                  as(attrib, 'CUdevice_attribute'),
                  as(dev, 'CUdevice'))
   if(is(ans, 'CUresult') && ans != 0)
       raiseError(ans, 'R_auto_cuDeviceGetAttribute')
   ans
}
```

This makes the call to the wrapper routine *R_auto_cuDeviceGetAttribute()*, passing it the two R values identifying the attribute and device of interest. An important step here is that this R function coerces the arguments it receives to the correct types. Here we have an opportunity to use 1-based counting familiar to R users for specifying the device. The coercion from the device number to a *CUdevice* object in R performs the subtraction to map to the 0-based couting used by CUDA. We manually defined the *CUdevice* class and the coercion method.

The function checks the status of the result to see if it is an R object of class *CUresult*. If it is and not 0, the call produced an error and we raise this in R. By doing this in R rather than in C code, it is easier to raise exceptions/conditions with classes corresponding to the type of error. This allows programmers to react to particular types of errors, e.g. out of memory or invalid device, in different ways.

The final piece of the bindings is the C wrapper routine *R_auto_cuDeviceGetAttribute()*. This is defined as

```
SEXP
R_auto_cuDeviceGetAttribute(SEXP r_attrib, SEXP r_dev)
{
    SEXP r_ans = R_NilValue;
```

```
    int pi;
    CUdevice_attribute attrib = (CUdevice_attribute) INTEGER(r_attrib)[0];
    CUdevice dev = INTEGER(r_dev)[0];
    CUresult ans;
    ans = cuDeviceGetAttribute(& pi,  attrib,  dev);
    if(ans)
       return(R_cudaErrorInfo(ans));
    r_ans = ScalarInteger(pi) ;
    return(r_ans);
}
```

The logic is quite straightforward and very similar to how we would write this manually. We convert the R-level arguments to their equivalent C types. We create a local variable (*pi*) used to retrieve the actual result. We call the CUDA routine and check the status value. If there is an error, we call a manually written routine *R_cudaErrorInfo()* which collects information about the CUDA error in the form of an R object. If the CUDA call was successful, we return the value of the local variable *pi* converted to the approriate R type.

We used **RCIndex** Temple Lang (2010-) to both read the declarations in the CUDA header files and to generate the bindings. Programmatically generating the bindings reduces the number of simple programming errors and also allows us to update the bindings when new versions of the API and SDK are released.

The CUDA API uses an idiom for most of its routines. A routine returns an error status value, and if this indicates success, the actual results are accessible via the pointers we passed to the routine. In other words, the routines return their values via input pointers. We specialized the general binding generation mechanism in **RCIndex** to exploit this idiom and be able to understand what was essentially an output parameter passed as a pointer input and what were actual pointer input parameters. This code to generate the CUDA-specific bindings is included in the package.

We also generated some of the documentation for the R functions programmatically. The CUDA header files have documentation in comments directly before each routine and data structure. We used **RCIndex** to extract these comments and then process the different parts describing the purpose of the routine, its parameters and return value.

5. Compiling R code to native GPU code

In this section we describe experimental work designed to allow programmers to utilize the power of the GPU without ever needing to leave R i.e., to allow the creation of CUDA kernels directly from R code. The approach to achieve this is to use LLVM (Low Level Virtual Machine), a compiler toolkit that can be embedded within R. This allows us to dynamically generate machine code within an R session and invoke those newly created routines, passing them inputs from R and obtaining the results in R. This simple approach is very powerful and allows us to "compile around the R interpreter" and gain significant speedups for some common R idioms Temple Lang (2013).

LLVM can generate machine code for the machine's CPU, but can also create PTX code to run on an NVIDIA GPU. We can use our R compiler code (**RLLVMCompile**), or a specialized version of it for targetting GPUs, and have it generate PTX code for kernel routines. We

can then load these routines onto the GPU and invoke them as a regular kernels. This was done using R as an example by researchers in NVIDIA Grover and Lin (2012) investigating dynamic, interpreted languages and GPUs.

6. Future Work

Now that we have the general-purpose bindings and access to the CUDA API, we, and hopefully others, will explore how to map commonly used statistical algorithms onto the GPU to take advantage of their potential. We plan to explore different examples and approaches and understand their characteristics in the context of GPU applications.

We also plan to explore the potential benefits of aspects of the API not addressed in this paper such as pinned memory, streams for interleaving computations and peer-to-peer communication between two or more GPUs.

As the need arises, we may generate bindings to other GPU-related libraries and APIs. It is useful to note though that there is no need to have R bindings to interface with kernel-level libraries such as Thrust Hoberock and Bell (2010) and parts of curand NVIDIA corp as these can be accessed in the usual manner when implementing the kernel. While this handles many such uses, however, we may need to determine the sizes of the different data structures they use that should be allocated by the host on the device (see `curand` examples in the **RCUDA** documentation for more details).

References

Buckner J, Seligman M, Wilson J (2011). ***gputools**: A few GPU enabled functions*. R package version 0.26, URL http://CRAN.R-project.org/package=gputools.

Grover V, Lin Y (2012). "Compiling CUDA and Other Languages for GPUs." http://on-demand.gputechconf.com/gtc/2012/presentations/S0235-GTC2012-CUDA-Compiling-Languages.pdf.

Hoberock J, Bell N (2010). "Thrust: A Parallel Template Library." Version 1.7.0, URL http://thrust.github.io/.

Kempenaar M, Dijkstra M (2010). ***rgpu**: Using the Graphics Processing Unit to speedup bioinformatics analysis with R*. R package version 0.8-1, URL https://gforge.nbic.nl/projects/rgpu/.

Khronos OpenCL Working Group (2008). *The OpenCL Specification, version 1.0.29*. URL http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

Kirk DB, Hwu WmW (2012). *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 0124159923, 9780124159921.

NVIDIA corp (????). *CURAND guide*. NVIDIA corp. See http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf.

Temple Lang D (2010-). "**RCIndex**: An R interface to libclang." http://www.omegahat.org/RCIndex. Version 0.2-0.

Temple Lang D (2013). "Compiling Code in R with LLVM." *Statistical Science*.

Urbanek S (2012). "**OpenCL**: Interface allowing R to use OpenCL." http://www.rforge.net/OpenCL/.

Affiliation:

Paul Baines
4210 Math Sciences Building,
University of California at Davis
One Shields Avenue
Davis, CA 95616
E-mail: pbaines@ucdavis.edu
URL: http://www.stat.ucdavis.edu/~pdbaines/

Duncan Temple Lang
4210 Math Sciences Building,
University of California at Davis
One Shields Avenue
Davis, CA 95616
E-mail: duncan@r-project.org
URL: http://www.omegahat.org