For instance, here is an excerpt of the CUDA matrix-multiply code presented in a talk given by Prof. Richard Edgar:[2]

```
int a = aBegin, b = bBegin; a <= aEnd; a += aStep,
      b+= bStep) {
   __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
   // Load matrices from global memory into shared memory
   // Each thread loads one element of each sub-matrix
   As[ty][tx] = A[a + (dc_wA * ty) + tx];
   Bs[ty][tx] = B[b + (dc_wB * ty) + tx];
```

Here **A** and **B** are in the GPU global memory, and we copy chunks of them into shared-memory arrays **As** and **Bs**. Since the code has been designed so that **As** and **Bs** are accessed frequently during a certain period of the execution, it is worth incurring the copying delay to exploit the fast shared memory.

Fortunately, the authors of the CUBLAS library have already done all the worrying about such matters for you. They have written very finely hand-tuned code for good performance, making good use of GPU shared memory and so on.

12.4 BLAS Libraries

In any discussion of high-performance matrix operations, the first question that arises is, “Which BLAS are you using?”

12.4.1 Overview

BLAS is an acronym for the Basic Linear Algebra Subprograms, a library of functions that perform very low-level operations such as matrix addition and multiplication. As will be discussed below, there are various BLAS implementations, all of them tailored (to varying degrees) to having good cache behavior and to otherwise have good performance.

R uses “vanilla” BLAS, CBLAS, which for instance comes standard with Ubuntu Linux. However, one may build R from source to include one’s own

[2]The same or similar code is available in full on the NVIDIA CUDA Samples Web site.

favorite BLAS, or to have the choice of BLAS done dynamically each time one runs R.

Thus even in ordinary serial computation, e.g. R's **%*%** operator, the speed of matrix operations may vary according to the version of BLAS that your implementation of R was built with.

In our context here of parallel computation, one version of BLAS of special interest is OpenBLAS, a multithreaded version that can bring performance gains on multicore machines, and also includes various efficiencies that greatly improve performance even in the serial case. We'll take a closer look at it in Section 12.5.

As noted, there is also CUBLAS, a version of BLAS for NVIDIA GPUs, highly tailored to that platform. A number of R packages, such as **gputools** and **gmatrix**, make use of this library, and of course you can write your own special-purpose R interfaces to it, say using **Rcpp** to interface it to R.

For message-passing systems (clusters or multicore), there is PBLAS, designed to run on top of MPI. The MAGMA library, with the R interface **magma**, aims to obtain good performance on hybrid platforms, meaning multicore systems that also have GPUs.

BLAS libraries also form the basis of libraries that perform more advanced matrix operations, such as matix inversion and eigenvalue computation. LAPACK is a widely-used package for this. as is ScaLAPACK for PBLAS.

12.5 Example: Performance of OpenBLAS

OpenBLAS is relatively new, having taken over a discontinued project, GotoBLAS. It is not yet clear what its long-term prospects are, but it seems very promising indeed.

For the timing experiments below, I simply switched from the default BLAS to OpenBLAS. This required setting some file permissions, and since I was running on a machine on which I did not have root access, I installed my own copy of R, in a directory **/home/matloff/MyR311**. (I had to build R with the option **–with-shared-blas**.) I also downloaded and built OpenBLAS, installing it in **/home/matloff/MyOpenBLAS**. I then needed to replace the R standard BLAS library via a symbolic link, as follows.

I entered the directory **/home/matloffMyR311/lib/R/lib** and did the operations

```
$ mv libRblas.so libRblas.so.SAVE
$ ln -s /home/matloff/MyOpenBLAS/lib/libopenblas.so \\
    libRblas.so
```

So, now whenever I run R, it loads OpenBLAS instead of the default BLAS.

OpenBLAS is a threaded application. It doesn't use OpenMP to manage threads, but it does allow one to set the number of threads using the OpenMP environment variable, e.g. in the **bash** shell,

```
$ export OMP_NUM_THREADS=2
```

If the number of threads is not set, OpenBLAS will use all available cores.[3] Note, though, that this may not be optimal, as we will see later.

It is important to note that that is all I had to do. From that point on, if I wanted to compute the matrix product AB in *parallel*, I just used ordinary R syntax:

```
> c <- a %*% b
```

I ran a squaring operation of random matrices of size 5000×5000 for 1, 2, 4, 6, 8, 10, 12, 14 and 16 cores on the 16-core machine described in this book's Preface. Let's look at the timing, in Figure 12.1. Though there is quite a bit of sampling variability and we would need to do multiple runs for a smoother graph, the results are clear: We are achieving linear speedup (e.g. doubling the number of cores cuts the run time approximately in half) up to about six cores, after which the returns are diminishing, if positive.

Note that not all multicore systems are alike, in terms of how resources are shared, say within groups of cores. More than one core may share a cache, for instance. Thus it's hard to predict at what point the "diminishing returns" effect will occur for any given application and and given hardware platform.

Performance of numerical algorithms is not just about speed; we must also consider accuracy. OpenBLAS derives its speed not just from making use of multiple cores, but also from various tweaks of the code, yielding a very fine degree of optimization. One can thus envision a development team so obsessed with speed that they might cut some corners regarding numerical accuracy. Thus the latter is a subject of legitimate concern.

I conducted a brief experiment to investigate this. I generated $p \times p$ correlation matrices, with all pairs of variables having correlation ρ, using the

[3] If the machine has hyperthreading (Section 1.4.5.2), the number of "cores" will be the product of the number of cores and the degree of hyperthreading.

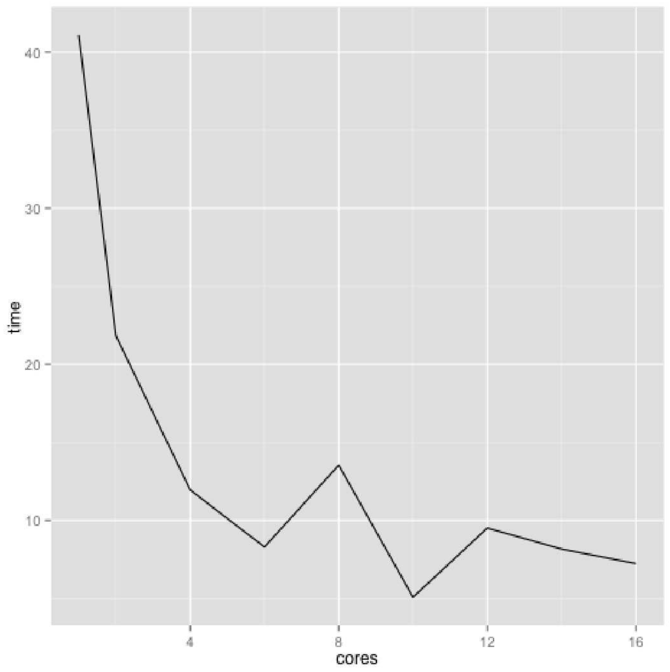

Figure 12.1: OpenBLAS Run Times

code

```
covrho <- function(p,rho) {
   m <- diag(p)
   m[row(m) != col(m)] <- rho
   m
}
```

The chosen application was eigenvalue computation, using R's **eigen()** function. I ran this on the 16-core machine, which has a hyperthreading degree of 2, and did not specify the number of threads, so OpenBLAS likely used 32 threads.

First, one more timing: With $p = 2500$ and $\rho = 0.95$, OpenBLAS handily beat R's stock BLAS, with a time of 12.101 seconds, versus 57.407 for stock BLAS.

Now, for the accuracy test: On the advice from a reader of my blog, Mad (Data) Scientist, I first set an R parameter to display 20 decimal digits in output:

```
options(digits = 20)
```

For plain R, the main eigenvalue was reported to be 2375.0500000000147338, while OpenBLAS gave it as 2375.049999999991087. The proportional difference, about 10^{-14}, seems pretty good for such an ill-conditioned matrix.[4] And of course, we don't know which reported eigenvalue is closer to the "true" one.

Needless to say, if you are using OpenBLAS and you are already using all the cores in your machine, it probably would not be profitable to use OpenMP and the like at the same time.

12.6 Example: Graph Connectedness

Let n denote the number of vertices in the graph. As before, define the graph's *adjacency matrix* A to be the $n \times n$ matrix whose element (i, j) is equal to 1 if there is an edge connecting vertices i and j (i.e. i and j are "adjacent"), and 0 otherwise. Let $R^{(k)}$ denote the matrix consisting of 1s and 0s, with a 1 in element (i, j) signifying that we can reach j from i in k steps. (Note that k is a superscript, not an exponent.)

Also, our main goal will be to compute the corresponding *reachability matrix* R, whose (i, j) element is 1 or 0, depending on whether one can reach j in some multistep path from i. In particular, we are interested in determining whether the graph is **connected**, meaning that every vertex eventually leads to every other vertex, which will be true if and only if R consists of all 1s. Let us consider the relationship between the $R^{(k)}$ and R.

12.6.1 Analysis

First, note that

$$R = b(\sum_{k=1}^{n-1} R^{(k)}) \qquad (12.15)$$

[4]The term means that slight changes in the matrix may result in large changes in output, in this case in the principal eigenvalue. In such a setting, roundoff error could be quite serious.