Speeding Up Computation

Tips Geared Towards R

Adam J. Suarez

Departments of Statistics
North Carolina State University

Arpil 10, 2015

Advantages of R

- As an interpretive and interactive language, developing an algorithm in R can be done very quickly.
- The main sacrifice is speed.
- As-is, R is better suited for prototyping, where the final program will eventually be run in a lower-level language like C or Fortran.
- However, the potential exists to be able to speed up much of the computation.
- R code should be seen as modular, where individual components can eventually be swapped out for faster versions when it is time for final runs or producing packages.
- This approach allows the best of both worlds, where R's excellent graphical abilities and user-contributed packages can still be used.

Hardware Considerations

- Faster hardware is the most straightforward way to speed up your computation.
- Statistical/scientific computation can have some special considerations compared to general computing.
- Of special importance to statistical computing is floating point performance, both single and double precision.
- When choosing hardware, one important factor to consider is theoretical peak FLOPS/sec (FLOPS = FLoating Point OperationS).
- Theoretical FLOPS/sec = (FLOPS/Processor cycle) * (Processor cycles/sec) * (# of Processors).

Hardware Considerations - CPU

- Intel and AMD use very different CPU architectures.
- AMD Bulldozer/Piledriver/Steamroller cores are paired and share some resources.
- The important fact for us is that they share the FPU (floating point unit).
- So, when performing floating point operations, an 8-core AMD processor acts like a 4-core processor.
- Integer operations are more independent.
- Intel's "cores" are completely independent.

Hardware Considerations - CPU

- Intel and AMD differ also in FLOPS/cycle:
  - Intel Haswell: 16 DP FLOPS/cycle, 32 SP FLOPS/cycle (per core).
  - AMD Bulldozer/Piledriver/Steamroller: 8 DP FLOPS/cycle, 16 SP FLOPS/cycle (per module).
- **Example**: Intel i7-5820k 6-core Haswell @ 4.0GHz has a theoretical peak of 384 DP GFLOPS/sec.
- **Example**: AMD FX-8150 8-core Bulldozer @ 4.0GHz has a theoretical peak of 128 DP GFLOPS/sec.
- Theoretical peak has the potential to be much higher than actual peak performance, depending on the problem and implementation.

Hardware Considerations - GPU

- The two main GPU competitors are AMD and NVIDIA.
- In general, GPUs do not have the 2:1 ratio of single:double precision performance.
- GPUs are often purposely crippled for DP at the consumer level to encourage purchases of workstation grade parts.
- The top of DP performance for their respective companies (consumer lines, single chip):
  - NVIDIA: GTX Titan Black $\sim$ 1700 DP GFLOPS/sec
  - AMD: Radeon R9 280x $\sim$ 870 DP GFLOPS/sec
- The main selling point for the workstation cards is ECC RAM.

Hardware Considerations - GPU

- NVIDIA is much more popular in the super-computing world, and the libraries for their platform are more developed.
- Computing on an AMD GPU is typically done using OpenCL, which aims to be more general than NVIDIA's CUDA libraries.
- NVIDIA GPUs tend to be more expensive, but much more user-friendly and more widely supported.
- For large problems, GFLOPS come much less expensively with GPUs than with CPUs, and libraries can now take advantage of multiple GPUs in a system (e.g., cuBLAS-XT).

BLAS Libraries

- By default, R comes with a basic version of BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage).
- It is a good idea to never use these shared libraries!
- There are many optimized versions available that can easily be interfaced with R.
    - OpenBLAS (Free)
    - Intel MKL (Free for Students)
    - AMD ACML (Free, GPU accelerated)
    - Many others...
- These all have BLAS and LAPACK libraries built in.

BLAS Libraries

- R can be compiled from source and told to build with an external library (recommended for MKL).
- You can also build R with a shared BLAS library and then "drop in" another library.
  - Shared libraries are:
    `/R/lib/libRblas.so`
    `/R/lib/libRlapack.so`
  - Either replace them (backing up the original) by copying, or use `update-alternatives` to easily change between them.
  - If using the alternatives option, make sure that the new library is in the run-time library load path.

BLAS Libraries

- OpenBLAS is the most user-friendly of the shared libraries.
- Intel MKL seems to need the Intel C compiler (icc) to function well.
- ACML 5.3.1 did not have GPU acceleration, and is much faster than ACML 6.1 for non-accelerated calls.
  - ACML 6.1 uses .lua scripts to determine whether to offload to the GPU.
  - This seems to significantly slow the non-offloaded calls.
  - ACML 6.1 also produced errors for me when calling `svd` in R for large matrices.

BLAS Libraries - NVBLAS

- NVIDIA distributes NVBLAS as part of their CUDA Toolkit.
- It works in a qualitatively different way than the other BLAS libraries.
- It “intercepts” certain level-3 BLAS calls and runs them on the GPU using cuBLAS.
    - GEMM, SYRK, HERK, SYR2K, HER2K, TRSM, TRMM, SYMM, HEMM
- Utilizes a unified memory approach, which means the data is never fully offloaded to the GPU RAM.
    - Don’t use on PCIe 2.0!
- You can use any full BLAS library as a default when it decides not to offload.
- Does not include a LAPACK library (CULA is a separate project).

BLAS Libraries - NVBLAS

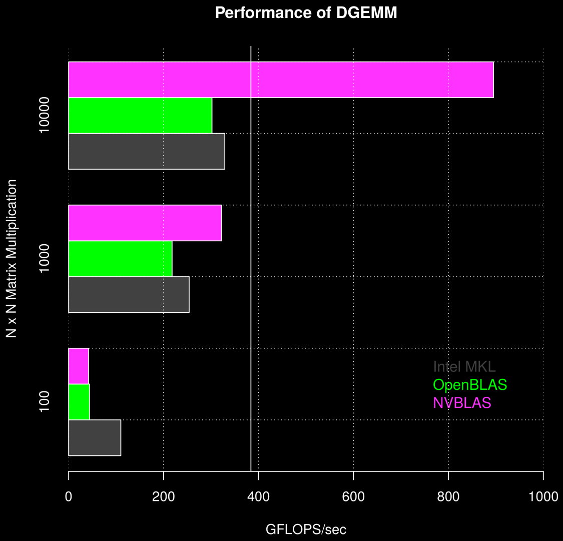
- Example NVBLAS configuration file (`nvblas.conf`):

```
NVBLAS_LOGFILE nvblas.log
NVBLAS_CPU_BLAS_LIB libmkl_rt.so
NVBLAS_GPU_LIST ALL
NVBLAS_AUTOPIN_MEM_ENABLED
# NVBLAS_CPU_RATIO_SGEMM 0.10
```

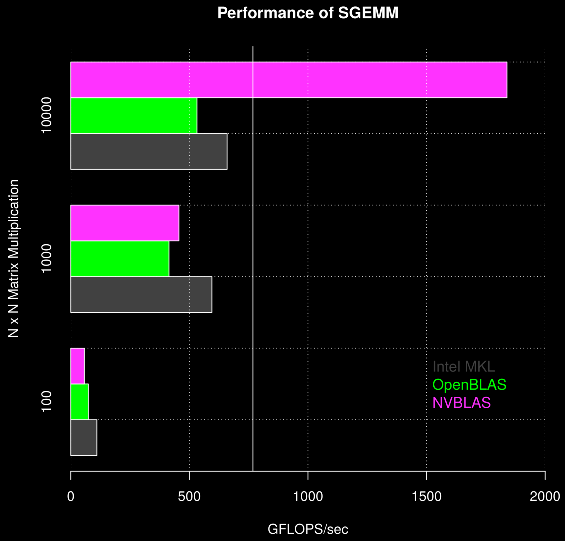
- To start R using NVBLAS, you could use the following:

```
env LD_PRELOAD=libnvblas.so R
```

BLAS Libraries



BLAS Libraries



Calling C from R

- Two ways to call C functions from R: `.C` and `.Call`/`.External`
- `.C` requires void functions with pointer arguments. Example:

```c
void my_function(int * a, double * b)
```

- `.Call` requires functions that both take and return SEXP values (S EXpression Pointer). Example:

```c
SEXP my_function(SEXP a, SEXP b)
```

- `.Call` is much faster than `.C` and is more in the style of "hacking" R.
- It allows you to create and use R objects directly.

Calling C from R - .Call

- Any SEXP values that you create must be protected from R's garbage collector using `PROTECT`.
- At the end, you need to use `UNPROTECT(N);`, where N is the number of previous protecting statements.
- For a single number, to convert from an SEXP value, use the functions `asInteger`, `asReal`, etc.
- To get the pointer to the numeric part of an SEXP value, use the functions `REAL`, `INTEGER`, etc.
- If you don't want to return a value, return `R_NilValue`.
- Don't alter SEXP objects passed as arguments! Use `duplicate` first, then alter the copy.

GSL/OpenMP Example

- The goal of this function is to take advantage of a multicore processor when generating random variables, in this case, standard normals.
- For random number generation, we will use the GSL (GNU Scientific Library), and its Ziggurat implementation.
- We will also use OpenMP directives to easily parallelize our method.
- The main issue that needs care is creating separate RNGs for each possible thread to ensure that the sequences still appear random.

GSL/OpenMP Example - Initializing RNG

```
gsl_rng_type *GSL_rng_t;
gsl_rng **GSL_rng;
int GSL_nt;
SEXP INIT_GSL_RNG(SEXP SEED){
  int j,seed=asInteger(SEED),i;
  GSL_nt=omp_get_max_threads();
  gsl_rng_env_setup();
  GSL_rng_t = gsl_rng_mt19937;
  GSL_rng = (gsl_rng **) malloc(GSL_nt * sizeof(gsl_rng *)
      );
  omp_set_num_threads(GSL_nt);
  ...
```

GSL/OpenMP Example Initializing RNG

```
...
#pragma omp parallel for private(i) shared(GSL_rng,
    GSL_rng_t) schedule(static,1)
  for(j=0;j<GSL_nt;j++){
    i=omp_get_thread_num();
    GSL_rng[i] = gsl_rng_alloc (GSL_rng_t);
    gsl_rng_set(GSL_rng[i],seed+i);
  }
  return R_NilValue;
}
```

GSL/OpenMP Example - Core Function

```
void generate_normal(double *out_v, int n, int nt){
  int j;
  #pragma omp parallel for shared(out_v,GSL_rng)
      num_threads(nt)
  for(j=0;j<n;j++){
    out_v[j] = gsl_ran_gaussian_ziggurat(GSL_rng[
        omp_get_thread_num()],1.0);
  }
}
```

GSL/OpenMP Example - Wrapper for .Call

```c
SEXP rnorm_gsl(SEXP N, SEXP NT)
{
  int n=asInteger(N),nt=asInteger(NT);
  SEXP result = PROTECT(allocVector(REALSXP,n));
  double * out_v = REAL(result);

  generate_normal(out_v,n,nt);

  UNPROTECT(1);
  return result;
}
```

GSL/OpenMP Example - Typical R Call

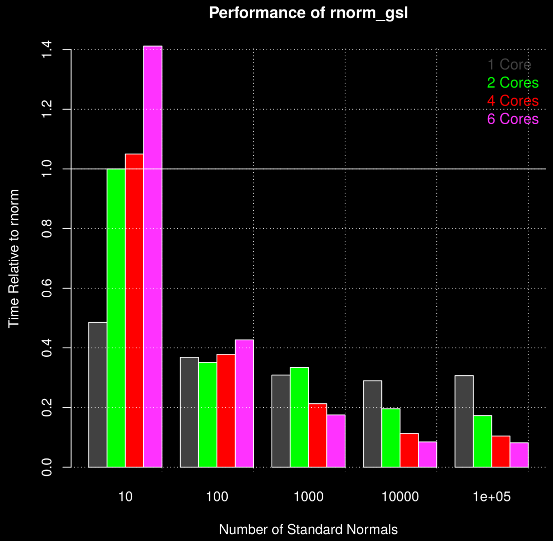
- The shared library can be compiled using a command like:

```
gcc -fPIC -shared -fopenmp -O3 -march=native gslrand.c
    -o gslrand.so -lgsl
```

- The function can now be used in R:

```
dyn.load("gslrand.so")
.Call("INIT_GSL_RNG",1)
a <- .Call("rnorm_gsl",20,1)
```

GSL/OpenMP Example - Performance



CUDA Example

- The goal of this function is to use the GPU to generate multivariate normal random variables.
- While simply generating univariate normals on the GPU is not nearly as efficient as the previous example, if we also Cholesky and matrix-multiply on the GPU, we see benefits in lower dimensions more quickly.
- NVIDIA includes cuRAND in the CUDA Toolkit for random number generation.
- CULA is a LAPACK library built using CUDA that can be obtained freely for academic use (requires registration).

CUDA Example - Initialization

```
curandGenerator_t CURAND_gen;
cublasHandle_t handle;

SEXP INIT_CURAND_RNG(SEXP SEED){
  curandCreateGenerator(&CURAND_gen,
      CURAND_RNG_PSEUDO_MTGP32);
  curandSetPseudoRandomGeneratorSeed(CURAND_gen,asInteger
      (SEED));

  culaInitialize();
  cublasCreate_v2(&handle);

  return R_NilValue;
}
```

CUDA Example - Core Function

```
SEXP rmvnorm_cuda(SEXP N, SEXP M, SEXP SIGMA)
{
  size_t n=asInteger(N), m=asInteger(M);
  double * devData, *dev_sigma;
  cudaMalloc((void **)&devData, n*m*sizeof(double));
  cudaMalloc((void **)&dev_sigma, m*m*sizeof(double));
  SEXP result = PROTECT(allocMatrix(REALSXP,n,m)),SIGMA2
     = PROTECT(duplicate(SIGMA));
  double * hostData = REAL(result), * sigma = REAL(SIGMA2
     ),alpha=1.0;
  cudaMemcpy(dev_sigma, sigma, m * m*sizeof(double),
     cudaMemcpyHostToDevice);
  ...
```

CUDA Example - Core Function

```
    ...
    culaDeviceDpotrf('L',m,dev_sigma,m);
    curandGenerateNormalDouble(CURAND_gen, devData, n*m,
        0.0, 1.0);
    cublasDtrmm_v2(handle, CUBLAS_SIDE_RIGHT,
        CUBLAS_FILL_MODE_LOWER, CUBLAS_OP_T,
        CUBLAS_DIAG_NON_UNIT,n,m,&alpha,dev_sigma,m,devData,
        n,devData,n);
    cudaMemcpy(hostData, devData, n * m*sizeof(double),
        cudaMemcpyDeviceToHost);
    cudaFree(devData);
    cudaFree(dev_sigma);
    UNPROTECT(2);
    return result;
  }
```

CUDA Example - Typical R Call

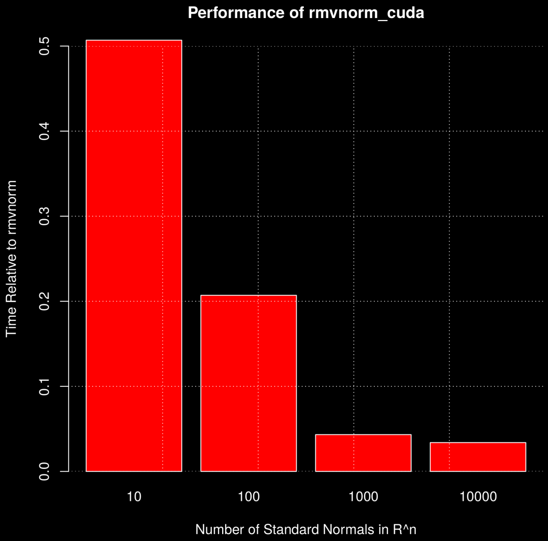
- The shared library can be compiled using a command like:

```
gcc -fPIC -shared -O3 -march=native curand.c -o
    cudanorm.so -lcudart -lcublas -lcurand -
    lcula_lapack
```

- The function can now be used in R:

```
dyn.load("cudanorm.so")
.Call("INIT_CURAND_RNG",1)
a <- .Call("rmvnorm_cuda",N,nrow(Sigma),Sigma)
```

CUDA Example



Thank You for Listening!

I will try to make my C code available on the SLG website, along with this presentation.