ACCELERATE R APPLICATIONS WITH CUDA

PATRIC ZHAO, SR. GPU ARCHITECT, NVIDIA
patricz@nvidia.com
AGENDA

- Background
- Deploy CUDA Libraries
- Apply DIRECTIVES
- Combine CUDA C/C++/Fortran
- Case study: kNN

Appendix: Build R with CUDA by Visual Studio on Windows
1. BACKGROUND

➢ Advantages of R:
  - Help to think with statistical methods
  - Design for data orientation
  - Interactive with other databases
  - Integrate with other languages
  - Provide high quality graphics

➢ Drawbacks of R:
  - speed: sometimes is very slow
  - memory: requires all data to be loaded into major memory (RAM)
R SOFTWARE STACK WITH CUDA

- R GPU Packages: easy to use
- CUDA Libraries: high quality, usability, portability
- DIRECTIVES: both CPU and GPU
- CUDA C/C++/Fortran: high performance & flexibility
2. DEPLOY CUDA LIBRARIES TO R

- Excellent usability, portability and performance
- Less development efforts and risks

Two examples:

- Accelerate Basic Linear Algebra Subprograms (BLAS)
  - how to use drop in library with R (S5355, S5232)

- Accelerate Fast Fourier Transform (FFT)
  - how to deploy CUDA APIs
  - how to build, link and use CUDA shared objects (.so)
**CASE 1. ACCELERATE BASIC LINEAR ALGEBRA SUBPROGRAMS (BLAS)**

- **Target**: speedup R BLAS computation, such as \%*\%

---

- R applications
- R standard interface Rblas.so

---

Various CPU BLAS implementations
- Intel MKL CPUs
- OpenBLAS

cuBLAS/NVBLAS
- Fermi GPU
- Kepler GPU
- Maxwell GPU
Drop-in NVBLAS Library on Linux

- Wrapper of cuBLAS
- Includes Standard BLAS3 routines, such as SGEMM
- Supports Multiple-GPUs
- ZERO programming effort

Q: How to use it with R?
A: Simple PRE-LOAD nvblas.so on Linux

Normally: 
R CMD BATCH <code>.R

NVBLAS: 
env LD_PRELOAD=libnvblas.so 
R CMD BATCH <code>.R
BENCHMARK RESULTS

- **revolution-benchmark** & **R-benchmark-2.5**

![Graphs showing benchmark results](image)

**CPU**: Intel, Sandy Bridge E5-2670, Dual socket 8-cores, @ 2.60GHz, 128 GB

**GPU**: NVIDIA, Tesla, K40m, 6GB memory
CASE 2. ACCELERATE FAST FOURIER TRANSFORM (FFT)

How to link CUDA libraries to R, including
- Determine R target function
- Write an interface function
- Compile and link to shared object
- Load shared object in R wrapper
- Execute in R
- Test Performance
Target Function in R

Basic compute pattern in finance, image processing, ...

such as stats:convolve() function in R is implemented by fft()

Fast Discrete Fourier Transform
Description
Performs the Fast Fourier Transform of an array.
Usage
fft(z, inverse = FALSE)
Arguments
z : a real or complex array containing the values to be transformed.
inverse : if TRUE, the unnormalized inverse transform is computed (the inverse has a + in the exponent of e, but here, we do not divide by 1/length(x))

CUDA library: cuFFT
Writing an interface function

Standard workflow for interface function

```c
#include <cufft.h>

void cufft(int *n, int *inverse, double *h_idata_re, double *h_idata_im, double *h_odata_re, double *h_odata_im) {
    cufftHandle plan;
    cufftDoubleComplex *d_data, *h_data;
    cudaMalloc((void**)&d_data, sizeof(cufftDoubleComplex)*(*n));
    h_data = (cufftDoubleComplex*)malloc(sizeof(cufftDoubleComplex) * (*n));
    // Covert data to cufftDoubleComplex type
    for(int i=0; i< *n; i++) {
        h_data[i].x = h_idata_re[i];
        h_data[i].y = h_idata_im[i];
    }
    cudaMemcpy(d_data, h_data, sizeof(cufftDoubleComplex) * (*n), cudaMemcpyHostToDevice);
    /* Use the CUFFT plan to transform the signal in place. */
    if(!*inverse) {
        cufftExecZ2Z(plan, d_data, d_data, CUFFT_FORWARD);
    } else {
        cufftExecZ2Z(plan, d_data, d_data, CUFFT_INVERSE);
    }
    cudaMemcpy(h_data, d_data, sizeof(cufftDoubleComplex) * (*n), cudaMemcpyDeviceToHost);
    // split cufftDoubleComplex to double array
    for(int i=0; i< *n; i++) {
        h_odata_re[i] = h_data[i].x;
        h_odata_im[i] = h_data[i].y;
    }
    /* Destroy the CUFFT plan. */
    cufftDestroy(plan);
    cudaFree(d_data);
    free(h_data);
} //main
```
Compile and link to Shared Object (.so)

```bash
nvcc -O3 -arch=sm_35 -G -l/usr/local/cuda/r65/include \ 
   -l/home/patricz/tools/R-3.0.2/include/ \ 
   -L/home/patricz/tools/R/lib64/R/lib -lR \ 
   -L/usr/local/cuda/r65/lib64 -lcufft \ 
   --shared -Xcompiler -fPIC -o cufft.so cufft-R.cu
```

Load Shared Object (.so) in Wrapper

```r
cufft1D <- function(x, inverse=FALSE) {
  dyn.load("cufft.so")
  n <- length(x)
  rst <- .C("cufft",
             as.integer(n),
             as.integer(inverse),
             as.double(Re(z)),
             as.double(Im(z)),
             re=double(length=n),
             im=double(length=n))
  rst <- complex(real = rst["re"], imaginary = rst["im"])
  return(rst)
}
```
Execute and Testing

```r
> source("wrap.R")
> num <- 4
> z <- complex(real = stats::rnorm(num), imaginary = stats::rnorm(num))
> cpu <- fft(z)
[1] 1.140821-1.352756i -3.782445-5.243686i 1.315927+1.712350i -0.249490+1.470354i
> gpu <- cufft1D(z)
[1] 1.140821-1.352756i -3.782445-5.243686i 1.315927+1.712350i -0.249490+1.470354i
> cpu <- fft(z, inverse=T)
[1] 1.140821-1.352756i -0.249490+1.470354i 1.315927+1.712350i -3.782445-5.243686i
> gpu <- cufft1D(z, inverse=T)
[1] 1.140821-1.352756i -0.249490+1.470354i 1.315927+1.712350i -3.782445-5.243686i
```
Intel Xeon CPU 8-cores (E5-2609 @ 2.40GHz / 64GB RAM)
NVIDIA GPU (Tesla K20Xm with 6GB device memory)
3. APPLY DIRECTIVES

- Directives is a common programming model now
  - Easy Programming: add several ‘#pragma’ statements
  - Portability: compiler, devices, performance
  - Works for legacy code: less effort

- Implementations in C/C++/Fortran level
  - CPU: Coarse granularity, task/data parallel w/ OpenMP
  - GPU: Finer granularity, data parallel w/ OpenACC
Example: speedup legacy code in `dist()`

- Compute the distances between the rows of a data matrix
- Implemented by C function

```r
> dist
function (x, method = "euclidean", diag = FALSE, upper = FALSE, p = 2)
{
  if (!is.na(pmatch(method, "euclidian")))
    method <- "euclidean"
  METHODS <- c("euclidean", "maximum", "manhattan", "canberra",
               "binary", "minkowski")
  method <- pmatch(method, METHODS)
  if (is.na(method))
    stop("Invalid distance method")
  if (method == -1)
    stop("Ambiguous distance method")
  x <- as.matrix(x)
  N <- nrow(x)
  attr(x, "dimnames") = list(NULL, NULL)
  contents <- x + x[1, ]
  D = diag(contents)
  if (method == 1)
    D = diag(contents)
  call = match.call()
  attributes(call) = list(method = method, call = match.call())
  .Call(C_Cdist, x, method, call, N, attr(x, "dimnames"))

<bytecode: 0x0000000014060a90>
<environment: namespace:stats>
```
Tips:
1. Reorganize code structure for GPU friendly
2. Avoid much logical checks, such as isnan()
3. Notice data copy method/size between CPU and GPU
4. Use `'-Mlarge_arrays'` compiler option for big data

source code: `<R source code path>/src/library/stats/src/distance.c`

```c
static double R_euclidean(double *x, int nr, int nc, int i1, int i2) {
    double dev, dist;
    int count, j;
    count = 0;
    dist = 0;
    for(j = 0; j < nc; j++) {
        if(both_non_NA(x[i1], x[i2])) {
            dev = (x[i1] - x[i2]);
            if(!ISNAN(dev)) {
                dist += dev * dev;
                count++;
            }
        }
    }
    i1 += nr;
    i2 += nr;
}
```

```c
// Patric: Fine granularity parallel by openACC
#include <cmath>
static double R_euclidean(double *x, int nr, int nc, int i1, int i2) {
    double dev, dist;
    int count, j;
    dist = 0;
    dev = 0;
    count = 0;
    #pragma acc routine(std::isnan) seq
    #pragma acc data copyin(x[0:nc*nr-1]) copy(dist)
    #pragma acc parallel for
    firstprivate(nc, nr) 
    private(j, dev, dist) 
    reduction(+:dist)
    for(j = 0; j < nc; j++) {
        dev = (x[i1 + j*nr] - x[i2 + j*nr]);
        dist += dev * dev;
    }
    // if(count == 0) return NA_REAL;
    // if(count != nc) dist /= ((double)count/nc);
    return sqrt(dist);
}
```
Compile with PGI

1. Do `make VERBOSE=1` in stats/src
   this step will generate detail information for build
2. Compile distance.c by PGI
   original: gcc -std=gnu99 ... -c distance.c -o distance.o
   changed: **pgcc** -acc -ta=nvidia -Minfo ... -c distance.c -o distance.o
3. Link all .o file to .so by PGI
   original: gcc -std=gnu99 -shared -o stats.so init.o <all.o> ....
   changed: **pgcc** -acc -ta=nvidia -shared -o stats.so init.o <all.o> ...
4. Update stats.so
   cp stats.so <R-path>/lib64/R/library/stats/libs/
5. Launch R and Execution as normally
   use nvprof to confirm : nvprof R ....
Compile with PGI

1. Do ‘make VERBOSE=1’ in stats/src
   this step will generate detail information for

2. Compile distance.c by PGI
   original: gcc -std=gnu99 ... -c distance.c -o distance.o
   changed: pgcc -acc -ta=nvidia -Minfo ... -c distance.c -o distance.o

3. Link all .o file to .so by PGI
   original: gcc -std=gnu99 -shared -o stats.so init.o <all.o> ....
   changed: pgcc -acc -ta=nvidia -shared -o stats.so init.o <all.o> ...

4. Update stats.so
   cp stats.so <R-path>/lib64/R/library/stats/libs/

5. Launch R and Execution as normally
   use nvprof to confirm : nvprof R ....

R_euclidean:
  53, Generating copyin(x[:nr*nc])
  Generating copy(dist)
  54, Accelerator kernel generated
  54, Sum reduction generated for dist
  55, #pragma acc loop gang, vector(256)
    /* blockIdx.x threadIdx.x */
  54, Generating Tesla code
Compile with PGI

1. Do `make VERBOSE=1` in stats/src
   this step will generate detail information for build

2. Compile distance.c by PGI
   original: gcc -std=gnu99 ... -c distance.c -o distance.o
   changed: pgcc -acc ...

3. Link all .o file to .so by PGI
   original: gcc -std=gnu99 -shared ...
   changed: pgcc -acc ...

4. Update stats.so
   cp stats.so <R-path>/lib64/R/library/stats/libs/

5. Launch R and Execution as normally
   use nvprof to confirm: nvprof R ....
RESULTS

Testing code from R:

```r
a <- runif(2^24, 1, 5)
b <- runif(2^24, 1, 5)
x <- rbind(a,b)

system.time( dist(x) )
```

<table>
<thead>
<tr>
<th>Vector (2^24)</th>
<th>Runtime (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>R built-in dist()</td>
<td>0.207</td>
<td></td>
</tr>
<tr>
<td>OpenACC</td>
<td>0.093</td>
<td>2.23X</td>
</tr>
</tbody>
</table>

CPU Intel Xeon E5-2609 @ 2.40GHz / 64 GB RAM
GPU Tesla K20Xm with 6GB device memory
3. COMBINE CUDA LANGUAGES TO R

- Existing libraries can't meet up function/performance target
- Write up your own functions by CUDA
- Same flow with calling CUDA library
  - Just change the CUDA API to your own kernel
Step 1: write GPU kernel function for your algorithm

```c
__global__ void vectorAdd(const double *A,
 const double *B,
 double *C,
 int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```
extern "C" void gvectorAdd(double *A, double *B, double *C, int *n)
{
    // Device Memory
    double *d_A, *d_B, *d_C;
    // Define the execution configuration
    dim3 blockSize(256,1,1);
    dim3 gridSize(1,1,1);
    gridSize.x = (*n + blockSize.x - 1) / blockSize.x;

    // Allocate output array
    cudaMalloc((void**)&d_A, *n * sizeof(double));
    cudaMemcpy(d_A, A, *n * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, *n * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, C, *n * sizeof(double));

    // Copy data to device
    cudaMemcpy((d_A, A, *n * sizeof(double)), cudaMemcpyHostToDevice);
    cudaMemcpy((d_B, B, *n * sizeof(double)), cudaMemcpyHostToDevice);
    // GPU vector add
    vectorAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, *n);

    // Copy output
    cudaMemcpy(C, d_C, *n * sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
4. CASE STUDY: K NEAREST NEIGHBORS

- Common classify algorithm
- Find K nearest neighbors from the training data by distance
- \( O(MNP) \) time complexity for direct implementation
- Benchmark: handwritten digits data of MNIST
  
  Kaggle data size: test(~30k, ~2k), train(~40k, ~2k)

5-NN Classifier Map from Wikipedia

Image from ~athitsos
Parallel Strategies

CRAN packages
- class:kNN
- FNN :kNN

directives
- openACC
- openMP

algorithm
KNN

R implementation
Custom Function

CUDA libraries
- nvBLAS

CUDA directives
- openACC
- openMP

represent by pattern
matrix solver

isolate computationally intensive task
rewrite by C/C++/Fortran

Parallel Algorithm
Basic Algorithm and Performance Baseline

Steps for kNN:
- Query a record: compute distance, sort, return most frequent labels

\[ \text{distance}(j) = \sum_k (\text{test}_{jk} - \text{train}_{jk})^2 \]

Implementations:
- Most common package
  class:KNN (C)
- Fast package
  FNN:KNN (C++, fast algorithm kd-tree)
- R implementation
  BenchR (R with 1 loop)

CPU: Ivy Bridge E5-2690 v2 @ 3.00GHz, dual socket 10-core, 128G
GPU: Nvidia Kepler, K40, 6G
Parallel Strategies

- **algorithm**: KNN
- **CRAN packages**: class:kNN, FNN :kNN
- **directives**: openACC, openMP
- **CUDA libraries**: nvBLAS
- **CUDA**: Parallel Algorithm
- **R implementation**: Custom Function
- **represent by pattern**: matrix solver
- **isolate computationally intensive task**: rewrite by C/C++/Fortran
- **encode by pattern**: OpenACC, openMP
- **CUDA directives**: OpenACC, openMP
Rewrite R implementation by pattern

\[
\text{distance} = \sum_j^n \sum_i^p (test_i - train_i)^2 \\
= \sum_j^n \sum_i^p (test_i^2 - 2*test_i*train_i + test_i^2)_j \\
= \sum_j^n \sum_i^p test_{ij}^2 - 2*\sum_j^n \sum_i^p (test_i * train_i)_j + \sum_j^n \sum_i^p train_{ij}^2
\]

Now, we have represented KNN algorithm by matrix operations, and we can easily accelerate it by CUDA libraries as we mentioned previously.
Rewrite KNN by matrix pattern and vectorization

# Rewrite BenchR knn by matrix operations and vectorization
knn.customer.vectorization <- function(traindata, testdata, cl, k)
{
    n <- nrow(testdata)
pred <- rep(NA_character_, n)

    # (traindata[i,] - testdata[i,])^2 --> (a^2 - 2ab + b^2)
    traindata2 <- rowSums(traindata*traindata)
testdata2 <- rowSums(testdata*testdata)
    # nvBLAS can speedup this step
    testXtrain <- as.matrix(testdata) %*% t(traindata)

    # compute distance
    dist <- sweep(testdata2 - 2 * testXtrain, 2, traindata2, '+')

    # get the k smallest neighbor
    nn <- t(apply(dist, 1, order))[,1:k]

    # get the most frequent labels in nearest K
    class.frequency <- apply(nn, 1, FUN=function(i) table(factor(cl[i], levels=unique(cl))))
    # find the max label and break ties
    pred <- apply(class.frequency, 2, FUN=function(i) sample(names(i)[i == max(i)], 1))

    unname(factor(pred, levels=unique(cl)))
}

- Matrix version is as fast as FNN:knn
- Run with nvBLAS we got:
  - 15X faster than class:knn
  - 3.8X faster than FNN:knn
Parallel Strategies

- CRAN packages
  - class: kNN
  - FNN : kNN
- directives
  - openACC
  - openMP
- algorithm
  - KNN
- R implementation
  - Custom Function
- represent by pattern
  - matrix solver
- CUDA libraries
  - nvBLAS
- isolate computationally intensive task
  - rewrite by C/C++/Fortran
- directives
  - openACC
  - openMP
- CUDA
  - Parallel Algorithm
Isolated computational task and rewrite by C

**rewrite kNN by matrix operations and vectorization**

```r
knn.customer.vectorization <- function(traindata, testdata, cl, k) {
  n <- nrow(testdata)
pred <- rep(NA_character_, n)

  # (traindata[i,] - testdata[i,])^2 --> (a^2 - 2ab + b^2)
  traindata2 <- rowSums(traindata*traindata)
testdata2  <- rowSums(testdata*testdata)
testXtrain <- as.matrix(testdata) %*% t(traindata)

  # compute distance
  dist <- sweep(testdata2 - 2 * testXtrain, 2, traindata2, '+')

  # get the k smallest neighbor
  nn <- t(apply(dist, 1, order))[,1:k]

  # get the most frequent labels in nearest K
  class.frequency <- apply(nn, 1, FUN=function(i) table(factor(cl[i], levels=unique(cl))))

  # find the max label and break ties
  pred <- apply(class.frequency, 2, FUN=function(i) sample(names(i)[i == max(i)],1))

  unname(factor(pred, levels=unique(cl)))
}
```

**dist.C**

```r
dist.C <- function(tndata, ttdata) {
  m <- nrow(ttdata)
n <- nrow(tndata)
p <- ncol(ttdata)
  rst <- .C("compute_dist", as.integer(n), as.integer(m), as.integer(p), as.double(ttdata), as.double(t(tndata)), mm = double(length=n*m))
  return(matrix(rst["mm"], nrow=m, ncol=n))
}
```
Write a C function
- don’t need to transfer R to C line by line (use C style!)
- rethink KNN computations, which is really like GEMM

\[ \text{GEMM}(i, j) = \sum_{k}^{P} (A_{ijk} \times B_{ijk}) \]

\[ \text{distance matrix}(i, j) = \sum_{k}^{P} (\text{test}_{ijk} - \text{train}_{ijk})^2 \]
So, we write C code by GEMM style for KNN

```c
void compute_dist(int *m, int *n, int *p, double *traindata, double *testdata, double *result);

void compute_dist(int *m, int *n, int *p, double *traindata, double *testdata, double *result)
{
    int i = 0, j = 0, k = 0 ;

    // Compute Distance Matrix
    for(i = 0; i < (*m); i++)
        for(k = 0; k < (*p); k++)
            for(j = 0; j < (*n); j++)
            {
                // GEMM
                //  result[i* (*n) +j] += testdata[i* (*p) +k] *  traindata[k * (*n) +j];

                // KNN
                double dist = testdata[i* (*p) +k] - traindata[k * (*n) +j];
                result[i* (*n) +j] += dist * dist ;
            }
}
```
And then, accelerate by openACC

```c
void compute_dist(int* m, int* n, int* p, double* restrict traindata, double* restrict testdata, double* restrict result);

void compute_dist(int* m, int* n, int* p, double* restrict traindata, double* restrict testdata, double* restrict result) {
    int i = 0, j = 0, k = 0;
    int mm = *m, nn = *n, pp = *p;

    // Compute Distance Matrix
    #pragma acc data copyout(result[0 : (mm * nn) -1]), copyin(testdata[0 : (mm * pp) -1], traindata[0 : (pp * nn) -1])
    {
        #pragma acc region for parallel, private(i), vector(8)
            for(i = 0; i < mm; i++) {
                #pragma acc for parallel,private(j,k), vector(8)
                    for(j = 0; j < nn; j++) {
                        #pragma acc for seq
                            for(k = 0; k < pp; k++) {
                                double tmp = testdata[i* pp +k] - traindata[k * nn +j];
                                result[i* nn +j] += tmp * tmp;
                            }
                        }
                    }
            }
        } // end openACC data region
    }
```
- C version is as fast as FNN:knn
- Compile with PGI (-Mlarge_arrays), we got:
  \textbf{13X} faster than class:knn
  \textbf{3.2X} faster than FNN:knn
Parallel Strategies

CRAN packages
- class:kNN
- FNN :kNN

CRAN packages directives
- openACC
- openMP

CUDA libraries
- nvBLAS

CUDA directives
- openACC
- openMP

R implementation
- Custom Function

algorithm
- KNN

R implementation represent by pattern
- matrix solver

parallel computationally intensive task
- rewrite by C/C++/Fortran

CUDA
- Parallel Algorithm
Accelerate CRAN packages by directive
- May be not easy since the package structure will be complex
- Need to fully understand algorithms and their implementations
- Select proper data decomposition method
  coarse granularity - openMP
  finer granularity - openACC

Class:KNN : source code is under:
<R source code path>/src/library/Recommended/class/src/class.c
knn function: VR_knn(…)

Coarse Granularity Decomposition

void VR_knn(Sint *kin, Sint *lin, Sint *pntr, Sint *pnte, Sint *p,
    double *train, Sint *class, double *test, Sint *res, double *pr,
    Sint *votes, Sint *nc, Sint *cv, Sint *use_all)
{
    ......
    
    // Patric: Coarse Granularity Parallel by openMP
    #pragma omp parallel for \
    private(npat, i, index, j, k, k1, kn, mm, ntie, extras, pos, nclass, j1, j2, needed, t, dist, tmp, ndist) \
    shared(pr, res, test, train, class, nte, ntr, nc)
    for (npat = 0; npat < nte; npat++) {
        ......

        // Patric: each thread malloc new buffer to resolve memory conflict of votes
        // change all votes to __votes in below source code.
        // Calloc is thread-safe function located in memory.c.
        Sint *__votes = Calloc(nc+1, Sint);

        ......

        Free(__votes);
    } // Patric: Top iteration and end of openMP
    RANDOUT;
}
Finer Granularity Decomposition

```c
void VR_knn(Sint *kin, Sint *lin, Sint *pntr, Sint *pnte, Sint *p,
          double *train, Sint *class, double *test, Sint *res, double *pr,
          Sint *votes, Sint *nc, Sint *cv, Sint *use_all)
{
    // Patric: Finer Granularity Parallel by openACC
    #pragma acc data copyin(test[0:nn*nte], train[0: nn*ntr])
    for (npat = 0; npat < nte; npat++) {
       ......
    }
    // Only parallelize this loop for Least Squares Model
    #pragma acc parallel loop private(k), reduction(+:dist)
    for (k = 0; k < *p; k++) {
        tmp = test[npat + k * nte] - train[j + k * ntr];
        dist += tmp * tmp;
    }
    ......
    RANDOUT;
}
```
- OpenACC version is not fast than original (only 2k features)
- OpenMP (1 CPU, 10 threads) is faster, we got:
  - **8.3X** faster than class:knn
  - **2.3X** faster than FNN:knn
Our post includes more details:

http://devblogs.nvidia.com/parallelforall/author/patricz/

Learn more on GTC 2015

CUDA General (tools, libraries)
S5820 - CUDA 7 and Beyond

CUDA Programming
S5651 - Hands-on Lab: Getting Started with CUDA C/C++
S5661, S5662, S5663, S5664, CUDA Programming Series

Directives
S5192 - Introduction to Compiler Directives with OpenACC

Handwritten Digit Recognition
S5674 - Hands-on Lab: Introduction to Machine Learning with GPUs: Handwritten Digit Classification
APPENDIX:

BUILD R WITH CUDA BY VISUAL STUDIO 2013 ON WINDOWS

1. Download and install Visual Studio 2013
   http://www.visualstudio.com/downloads/download-visual-studio-vs

2. Download and install CUDA toolkit
3. Open VS2013, and create ‘New Project’ then you will see NVIDIA/CUDA item.
4. Select ‘Visual C++’ → ‘Win32 Console Application’
5. Select ‘DLL’ for Application type to create a ‘Empty project’ in Wizard platform
6. Changes Project type to CUDA

‘Solution Explorer’ →
right click project name →
‘Build Dependencies’ →
‘Build Customizations...’ →
‘CUDA 6.5’
7. Add cuda and cuda accelerated libraries into Visual Studio

Right project name in ‘Solution Explorer’ →
‘Properties’ → ‘Linker’ → ‘Input’ → ’Additional Dependencies’
Add “cufft.lib” and “cudart.lib”
8. Add CUDA source code file with .cu suffix

Right click “Source Files” in “Solution Explorer”

→ ‘Add’
→ ‘New Item’
→ ‘C++ File(.cpp)’
→ type cuFFT.cu
- Check the ‘Item type’ of `cuFFT.cu` by right clicking filename (`cuFFT.cu`) and selecting ‘Properties’.
- The type should be ‘CUDA C/C++’; otherwise, change to CUDA type.
9. Change to 64bit in case you are using 64bit R and CUDA

→ ‘Build’
→ ‘Configuration Manager’
→ ‘Active solution platform:’
→ ‘New’
→ select ‘x64’
10. Select 64bit CUDA and shared runtime
   → Right project name in ‘Solution Explorer’
   → ‘Properties’ → ‘CUDA C/C++’ → ‘Common’
   Select:
   ‘Shared/dynamic CUDA runtime library’ in CUDA Runtime
   ’64-bit (--machine 64)’ in Target Machine Platform
11. Copy your CUDA code into this file

- Add necessary header files for CUDA
  
  ```
  /* Basic API header files*/
  #include <stdlib.h>
  
  /* CUDA API header files*/
  #include <cufft.h>
  #include <cuda_runtime.h>
  ```

- Declare routines which need to call from R with 
  extern “c” __declspec(dllexport)

  ```
  extern "C" __declspec(dllexport)
  void cufft(int *n, int *inverse, double *h_idata_re, double *h_idata_im, double *h_odata_re, double *h_odata_im)
  ```
12. Build Project and get cuFFT.dll

13. Load cuFFT.dll in R and check the dll path
14. Run cuFFT in R on Windows

```r
> z <- complex(real = stats::rnorm(num), imaginary = stats::rnorm(num))
> cufft1D(z)
[1] -3.375226-0.617570i  1.128137+3.148557i  -0.781643+2.983633i  -6.233749-0.037744i
> fft(z)
[1] -3.375226-0.617570i  1.128137+3.148557i  -0.781643+2.983633i  -6.233749-0.037744i
```
Multi-GPUs Case: General Matrix Multiplication

- Just add more GPU index in nvblas.conf file
  \textit{NVBLAS\_GPU\_LIST 0 1}
- GPU solution gains
  - higher speedup than multi-threads solutions