GPU libraries — Get More for Less

Jan H. Meinke | 18.02.2020

Abstract: GPU libraries come in many shapes and sizes. They make GPU programming easier, more portable, and less error prone. In this talk, I start with very general libraries such as CUB and Thrust, present some of the numerical libraries like cuBLAS and cuRand, and give examples of domain specific libraries such as OpenMM and QUDA. Short code snippets illustrate how to use GPU libraries from C++, Fortran, and Python.

We have seen in Ben Cummings talk how CSCS uses libraries to provide common abstractions for porting applications. In this talks, I will introduce some libraries that make working with GPUs easier. Several of these libraries deserve a talk (or a course) of their own, but I hope I can give you a taste.

CUDA unbound (CUB)

When CUDA was first released you had to write your own kernel. If you needed to perform a reduction in your code, you wrote your own reduction algorithm. But cooperative parallel programming is hard and an optimal reduction algorithm for a Kepler GPU like the K80 on JURECA looks different from an optimal implementation on a Volta GPU on JUWELS.

CUB is a template library for parallel algorithms that we can use from within our CUDA kernels that provides optimal implementations of various algorithms for all available Nvidia GPU architectures. It provides warp, block, and system level algorithms. CUB uses some additional abstractions for flexibility and is very powerful.

C++ Template library with warp-, block, and device-level implementations.

Optimized for *all* Nvidia GPU architectures

Includes

- Reduction
- Scan
- Histograms
- and more

If we are using hip, we can use hipCUB. It will use rocPRIM on an AMD platform and CUB on a CUDA platform.

CUB

Block-level reduction

Let's take a look at a block-level reduction:

```
__global__ void BlockReduce(double* in, double* out){
    // Use a block with 256 threads
    using BlockReduceT = BlockReduce<double, 256>;
    __shared__ typename BlockReduceT::TempStorage temp;
    // Load 4 items per thread
    double data[4];
    LoadDirectStriped<256>(threadIdx.x, in, data);

    double mySum = BlockReduceT(temp.Sum(data));

    if(threadIdx.x == 0){
        *out = mySum;
    }
}
```

We first define the type of block reduce that we want to perform. Here we'll sum up doubles using 256 threads per block. The block reduce uses shared memory to optimize the reduction. It defines it's own type TempStorage to do that. To load the data efficiently, we use LoadDirectStriped and pass the block size, our thread ID and the storage spaces. Finally, we perform the sum. Thread 0 accumulates the result and returns it. If we were using multiple blocks, each would write its result to a result array instead of a single value, e.g.,

```
if(threadIdx.x == 0){
    out[blockIdx.x] = mySum;
}
```

CUB

Template parameters

There are a couple of hardcoded parameters in the code. The most notable may be the block size. What do I do if I need a smaller or bigger block size? I could write a new kernel, but there is a more convenient way: I can use template paramters.

```
template <int ThreadsPerBlock>
__global__ void BlockReduce(double* in, double* out){
    // Use a block with ThreadsPerBlock threads
    using BlockReduceT = BlockReduce <double, ThreadsPerBlock>;
    __shared__ typename BlockReduceT::TempStorage temp;
    // Load 4 items per thread
    double data[4];
    LoadDirectStriped<ThreadsPerBlock>(threadIdx.x, in, data);

    double mySum = BlockReduceT(temp);

    if(threadIdx.x == 0){
        *out = mySum;
    }
}

// Call the kernel from the host
BlockReduce<256><<<1, 256>>>(in, out);
```

We can adjust other parameters, too, for example, the number of data items to load per thread or the algorithm used for the reduction. Each would be passed as a template parameter. The advantage of using template parameters is that they are resolved at compile time and don't entail any run time overhead.

CUB

Device functions

The device level functions can be called directly from the host program or from a kernel (dynamic parallelism). If you call them from a kernel make sure that only *1* thread makes the call. In this example we call DeviceReduce::Sum from the host. Note the memory management.

```
CachingDeviceAllocator  g_allocator(true);  // Caching allocator for device memory
...
int *d_in = NULL;
g_allocator.DeviceAllocate((void**)&d_in, sizeof(int) * num_items);
// Initialize device input
cudaMemcpy(d_in, h_in, sizeof(int) * num_items, cudaMemcpyHostToDevice);
// Allocate device output array
int *d_out = NULL;
g_allocator.DeviceAllocate((void**)&d_out, sizeof(int) * 1));

// Request and allocate temporary storage
void            *d_temp_storage = NULL;
size_t          temp_storage_bytes = 0;
DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_out, num_items);
g_allocator.DeviceAllocate(&d_temp_storage, temp_storage_bytes);
// Run
DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in, d_out, num_items);
```

Here is the complete example for the device reduction.

Thrust

CUB is mostly used from within CUDA kernels. Thrust on the other hand provides a set of parallel C++ template algorithms for the GPU. Here is an example that fills a vector with random number, transfers the data to the GPU, sorts them there and then returns the smallest value:

```
#include <algorithm>
#include <iostream>
#include <random>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

int main(){
    size_t N = 10'000'000;
    std::mt19937 re;
    std::uniform_real_distribution<double> uniform_dist(0.0, 1.0);
    auto uniform = [&](){return uniform_dist(re);};
    thrust::host_vector<double> a_h(N);
    std::for_each(a_h.begin(), a_h.end(), [=](auto& e){e = uniform();});
    thrust::device_vector<double> a = a_h;
    thrust::sort(a.begin(), a.end());
    std::cout << "The smallest random number generated is " << a[0]
              << " and  the largest is " << a[N-1] << "\n";
 }
```

If you want to learn more, take a look at An Introduction to Thrust.

Thrust

Random number generation on the GPU

Why generate random numbers on the CPU, if all we want to do is sort them on the GPU. It would be better to generate the random numbers on the GPU in the first place. With CUDA we can define functions that can be called from the host *and* the device. Many Thrust functions are defined as host/device functions. The following function uses Thrust's random number generator:

```
#include <thrust/random.h>

struct thrust_uniform_dist {

    double a, b;

    __host__ __device__
    thrust_uniform_dist(double a = 0.0, double b = 0.0) : a(a), b(b) {};

    __host__ __device__ double operator()(const unsigned int n) const {
        thrust::default_random_engine rng;
        thrust::uniform_real_distribution<double> dist(a, b);
        rng.discard(n);
        return dist(rng);
    }
};
```

We can call this function to fill a host or a device vector. To fill the host vector, the code runs on the CPU and to fill the device vector is runs on the GPU.

Thrust

Random number generation on the GPU

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/iterator/counting_iterator.h>

/** Generate N random numbers on the GPU and sort them */
int main(int argc, char **argv) {
    const size_t N = 10'000'000;
    thrust::device_vector<double> a(N);
    // This would also work with a host_vector:
    thrust::counting_iterator<unsigned int> index_sequence_begin(0);
    thrust::transform(index_sequence_begin, index_sequence_begin + N,
                      a.begin(), thrust_uniform_dist(0.0, 1.0));
    ...
}
```

Thrust

Reduction

Let's calculate the first few moments of our distribution.

```
...
double mean = thrust::reduce(a.begin(), a.end(), 0.0) / N;
double variance = thrust::transform_reduce(a.begin(), a.end(),
                                   [mean] __host__ __device__ (double e){
                                       return (e - mean) * (e -mean);
                                   }, 0.0, thrust::plus<double>()) / N;
double kurtosis = thrust::transform_reduce(a.begin(), a.end(),
                                   [mean] __host__ __device__ (double e){
                                       return pow(e - mean, 4);
                                   }, 0.0, thrust::plus<double>()) / (variance * variance * N);
std::cout << "The first three moments of the distribution are "
          << std::setprecision(3)
          << mean << ", " << variance << ", and " << kurtosis << ".\n";

...
```

Thrust

Reduction called from CUDA (kernel)

Thanks to dynamic parallelism thrust::reduce can also be called from a device function:

```
 #include <iostream>
 #include <numeric>
 #include <thrust/device_vector.h>
 #include <thrust/host_vector.h>

__global__ void square_reduce(double* v, double* result, int N){
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N){
        v[i] *= v[i];
    }
    if (i == 0){
        *result = thrust::reduce(thrust::cuda::par, v, v + N, 0.0);
    }
}
```

The first argument of thrust::reduce is an execution policy. Here, we are telling it to launch a parallel reduction kernel. Note, that it's called from a single thread!

Thrust

Reduction called from CUDA kernel (main)

The main function converts the pointer to the data of the device_vector into a CUDA raw pointer that can be passed to a CUDA kernel function. This is also useful for combining thrust with the cu* libraries, which we will discuss next.

```
int main(){
    int N = 10'000;
    thrust::host_vector<double> v(N);
    std::iota(v.begin(), v.end(), 0.0);

    double host_result = thrust::transform_reduce(v.begin(), v.end(), [](double e){
        return e * e;}, 0.0, thrust::plus<double>());

    thrust::device_vector<double> v_dev = v;
    double* result = nullptr;
    cudaMallocManaged(&result, sizeof(double));
    // Cast to raw pointer that can be passed to kernel
    double* v_dev_raw = thrust::raw_pointer_cast(v_dev.data());

    square_reduce<<<(N + 255) / 256, 256>>>(v_dev_raw, result, N);
    cudaDeviceSynchronize();
    std::cout << "Difference in Result: " << *result - host_result << "\n";
}
```

cu* Libraries

Nvidia provides a number of GPU accelerated libraries that cover common computational tasks such as calculating matrix multiplication, solving graph problems, or calculating Fourier transforms using fft. These libraries expect pointers to device memory as input parameters and are meant to be called from a program that is compiled with a CUDA capable compiler.

- cuBLAS — linear algebra routines
- cuSparse — linear algebra routines for sparse matrices
- cuGraph — not to be confused with CUDA Graphs.
- cuRand — random number generators
- cuFFT — fast Fourier transforms
- cuSolver — solver for systems of equations
- cuTensor — tensor library

cuRand

Good random number generators are needed for many application including statistical thermostats and (Markov chain) Monte Carlo methods. cuRand offers several random number generators, including Mersenne Twister. Based on these random number generators it can generate many distributions of random numbers.

Generators:

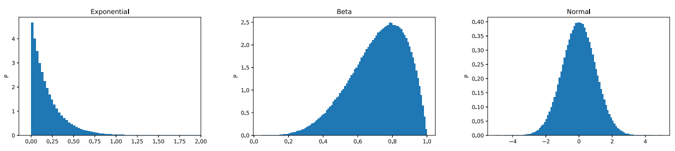
Routines to generate uniform random integers:

- Mersenne Twister
- XORWOW
- Philox
- …

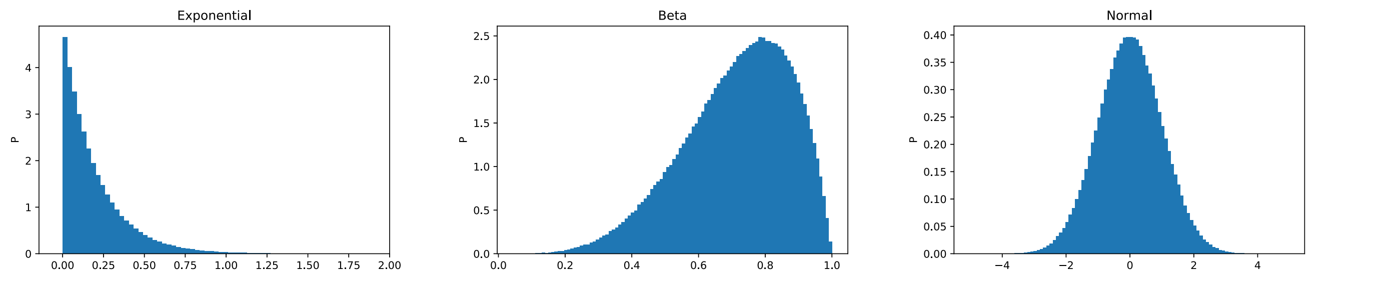
Distributions:

Generators are used to obtain samples from random distributions including

- uniform
- normal
- exponential
- beta
- …



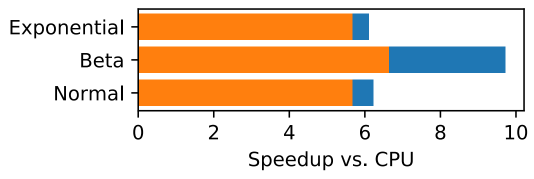
Nvidia cuRand vs. Intel MKL



The following script measures the time it takes to generate random number using cuRand and the MKL. Both are called from Python, but you can expect similar performance difference when you call them from other languages. The CPU would use a single thread. The %timeit magic command is not part of Python but of IPython.

```
t = []
for rs in [random_intel.random_sample, cupy.random.random]:
    for i in range(3, 8):
        tt = %timeit -o rs(10 ** i)
        t.append(tt)
```

The script to generate the data for the following plot was more complicated since it ran the random number generator on the CPU using multiple threads. I used a single JUWELS GPU node with 2 20-core Intel Xeon Gold 6148 CPUs and 4 Nvidia V100 GPUs. The speedup for 1 GPU is measured compared to one CPU (blue) and both CPUs (orange).



cuBLAS

Linear algebra routines

BLAS routines are the basics for many things including solving systems of linear equations. Since the linpack benchmark that determines the order of supercomputers in the Top 500 list is basically a solver for linear equations, BLAS routines tend to be highly optimized for many different architecture.

BLAS routines are split into different levels:

BLAS Level 1:
Functions that act on scalars and vectors. They include

- sums
- dot product
- ax+y (axyp)

BLAS Level 2:
Functions that perform matrix-vector operations, e.g.,

- gemv ($y = \alpha Ax + \beta y$, where A is a matrix)

BLAS Level 3:
Functions that perform matrix-matrix operations, e.g.,

- gemm ($C = \alpha AB + \beta C$, where A, B, and C are matrices)

BLAS Level 3 routines have the highest compute intensity. The routine DGEMM performs the above calculation for double precision numbers (thus the D).

Nvidia cuBLAS vs. Intel MKL

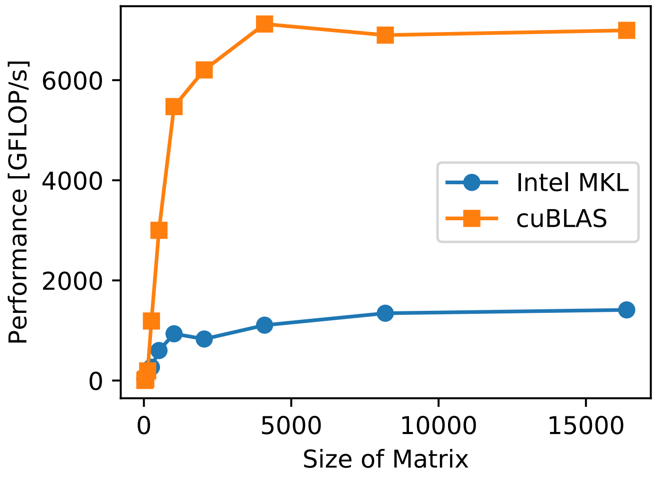
DGEMM

The following plot shows the performance of a matrix-matrix multiplication using cuBLAS and the MKL. Both are called from Python, but you can expect similar performance differences when you call them from other languages. The measurements were done on a single JUWELS GPU node. Intel's MKL automatically uses all available cores. Nvidia's cuBLAS uses only a single GPU.

Note

Timing cuBLAS routines can be tricky. They are called asynchronously, i.e., they may return right away. To make sure you include the entire time, you can synchronize with the GPU using cupy.cuda.Device(0).synchronize().

And here is a plot of the (properly synchronized) results.



We are getting to peak performance on the GPU already with 4096 by 4096 matrices. Since we generated the data using cuRand, the data always stayed on the GPU.

The CPU only achieves about 2/3 of its peak performance and is still slowly increasing as we go to larger and larger matrices.

How to Use Them From Python

Nvidia supports cupy a Python library that provides a numpy-like interface for GPUs. Just like numpy takes advantage of fast BLAS and FFT libraries, so does cupy use the cu* libraries:

```
import cupy
N = 4096
# A and B are generated on the GPU
A = cupy.random.random((N, N)) # uniform random numbers using cuRand
B = cupy.identity((N, N))
C = A@B # Python matrix operator uses cuBLAS
```

How to Use Them From C++

C/C++ are CUDA's first languages and all the libraries can easily be called from C/C++. For documentation see the CUDA API Reference.

```
 #include <cublas_v2.h>

int main(){
    double *A, *B, *C;
    double alpha=1.0, beta=1.0;
    cublasStatus_t status;
    cublasHandle_t handle;

    cudaMallocManaged(&A, width * width * sizeof(double));
    cudaMallocManaged(&B, width * width * sizeof(double));
    cudaMallocManaged(&C, width * width * sizeof(double));
    initialize(A, B, C); // initialize A, B, and C with some values;

    status = cublasCreate(&handle);
    status = cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, width, width, width, &alpha, A,
                         width, B, width, &beta, C, width);
    cudaDeviceSynchronize();

    doSomethingwC(C);
}
```

Make sure to check the return value of cudaMallocManaged and the status values returned from the cuBLAS calls. Otherwise, you won't know if your program actually did anything.

How to Use Them From Fortran 2003

Using CUDA Fortran

If you have a CUDA Fortran compiler available, calling the libraries is fairly easy. As mentioned above, you can allocate memory on the GPU using

```
use cudafor
use curand

real*8, managed :: A(1024, 1024)
real*8, managed, dimensions(:,:), allocatable :: B, C
```

You can pass these arrays to the appropriate library:

```
allocate(B(1024, 1024))
allocate(C(1024, 1024))

type(curandGenerator):: g
istat = curandCreateGenerator(g, CURAND_RNG_PSEUDO_MT19937)
istat = curandGenerateUniformDouble(g, A, 1014 * 1024)
istat = curandGenerateUniformDouble(g, B, 1014 * 1024)
C = 0.0
istat = cublasInit()
call dgemm('n', 'n', size(A, 1), size(B, 2), size(A, 2), alpha, A, size(A, 1), B, size(B, 1), &
           beta, C, size(C, 1))
istat = cudaDeviceSynchronize()
```

Note

There are couple of things missing in the above code, to make it complete. You should of course use implicit none and define alpha, beta, and istat. In production code, you should also check the value of istat to make sure that no errors are returned. For more information take a look at PGI Fortran CUDA Library Interfaces.

How to Use Them From Fortran

Writing your own interface

If you want to call a library function that has not been wrapped by CUDA Fortran (or you are using another Fortran compiler), you have to write your own interface. Fortran 2003 provides the iso_c_binding module to help with this PGI provides some additional utilities. The following example is taken from that section:

```fortran
! cufftExecC2C
interface cufftExecC2C
    integer function cufftExecC2C( plan, idata, odata, direction ) bind(C,name='cufftExecC2C')
        integer, value :: plan
        complex, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecC2C
end interface cufftExecC2C
```

Note

bind(C, name='cufftExecC2C') ensures that the correct capitalization is used for the call to the C interface. Btw., this function already exists. In the documentation, you'll also find an example for calling a Thrust library routine using a !$cuf kernel do for the data initialization. You'll learn more about cuf kernels later.

nv* Libraries

Nvidia's nv* libraries take care of memory transfers to and from the GPU.

- NVBLAS
- nvGraph (will be dropped from future CUDA releases, use cuGraph instead)

NVBLAS

NVBLAS is special. It can be linked in addition to a CPU library and intercepts BLAS calls. If it considers it worth it to transfer data to the GPU, it will perform matrix-matrix operations on the GPU and deal with the data transfer. Otherwise, the call will be passed on to a CPU BLAS library.

- Drop-in replacement
- Interecepts standard BLAS calls
- Heuristic to decide if it's worth it to use the GPU
- Only BLAS3 routines (matrix-matrix multiplication)
- Multi-GPU capable

Going Beyond a Single Node

The libraries I talked about so far are all limited to a single node, but there are some attempts to build libraries that take advantage of distributed resources:

- SLATE — modern replacement of ScaLAPACK
- AccFFT — distributed FFT library

The single node performance on Summit is about as 43.6 TF. The 72 nodes used for the figure have a peak performance of about 3.1 PF.

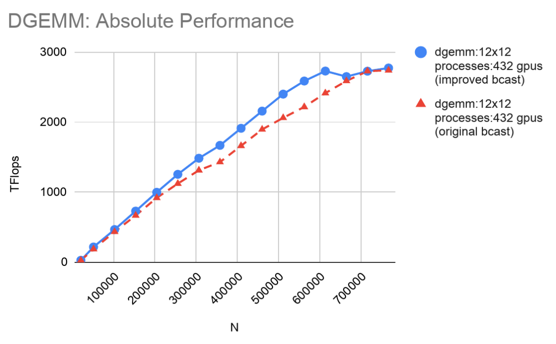


Figure 2.1 from [Gates2020] 72 nodes on Summit.

[Gates2020] Gates, M., Charara, A., & Kurzak, J. (1936). SLATE} Working Note 13. Psychometrika, 1(3), 211-218.

Machine Learning

Machine learning and in particular deep neural networks has been a driver for the development of GPUs in the last few years. The tensor cores introduced by Nvidia with the Volta architecture were the first elements on a GPU that had no immediate application to graphics and gaming. Since ther's a lot of money in providing the best hardware for machine learning, AMD, Intel and Nvidia put a lot of effort in their machine learning software stacks. The dominant language interface in this case is Python, so I'll stick to Python examples.

- Mxnet (Apache)
- PyTorch (Facebook)
- Tensorflow (Google)
- …

Here's an example taken from https://www.tensorflow.org/overview/

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(input_shape=(28, 28)),
tf.keras.layers.Dense(128, activation='relu'),
tf.keras.layers.Dropout(0.2),
tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Rapids

Rapids is similar in spirit to thrust. It provides a set of objects and algorithms that is GPU- enabled with a focus on data science and Python.

- (Dask-)cuDF, a (distributed) pandas-like dataframe
- cuML, machine learning algorithms on the GPU. The API is usually identical to scikit-learn's API
- cuGraph, a library for graph analytics that provides some basic graph algorithm, e.g, single-source shortest path and page rank.
- Support for dask

It takes advantage of CuPy, which as mentioned above provides an interface to the various cu* libraries including cuDNN. Also, it tries to stick to the API known from standard packages such as Pandas and scikit-learn.

Rapids

Linear regression (CPU)

Let's take a look at a simple linear regression (linear fit). I use a simple linear relation with some added random noise. This example is taken from the Introduction to RAPIDS notebook.

```
import cudf
import cuml
import numpy
import sklearn
import matplotlib.pyplot as plt

n_rows = 100000
w = 2.0
x=numpy.random.normal(loc=0, scale=1, size=(n_rows,))
b = 1.0
y = w * x + b

noise = numpy.random.normal(0, 2, (n_rows,))
y_noisy = y + noise
```

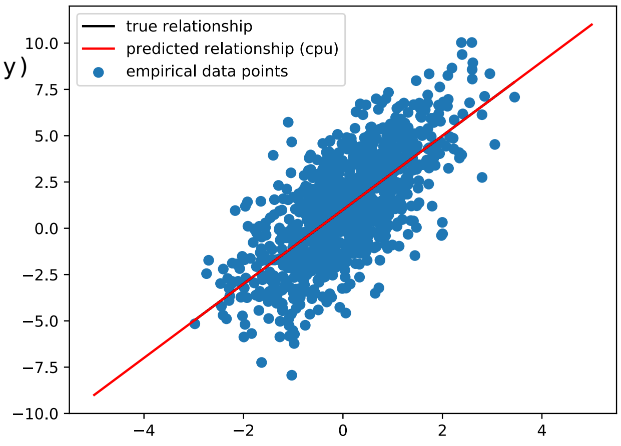
Now, we first perform the linear regression using scikit-learn:

```
from sklearn.linear_model import LinearRegression


linear_regression = LinearRegression()
linear_regression.fit(numpy.expand_dims(x, 1), y)

inputs = numpy.linspace(-5, 5, 1000)
outputs = linear_regression.predict(
              numpy.expand_dims(inputs, 1))

 plt.scatter(x, y_noisy,
             label='empirical data points')
 plt.plot(x, y, color='black',
          label="true relationship")
 plt.plot(inputs, outputs, color='red',
          label='predicted relationship (cpu)')
 plt.legend()
 plt.savefig("linear_regression_cpu.svg")
```



Rapids

Linear regression (GPU)

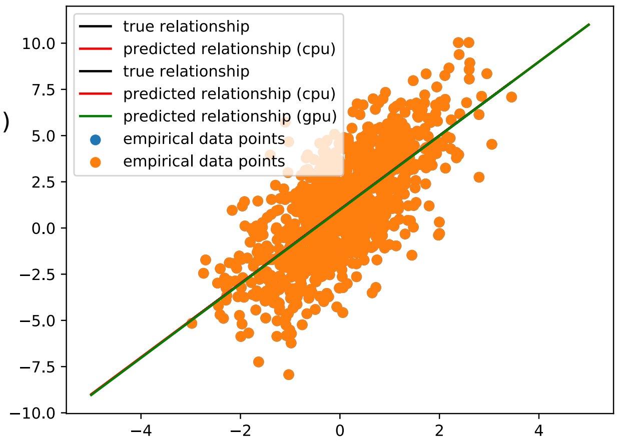
To run the linear regression on the GPU, we use a cudf DataFrame. Other than that the code is very similar:

```
from cuml.linear_model import LinearRegression

df = cudf.DataFrame({'x': x, 'y': y_noisy})
linear_regression = LinearRegression()
linear_regression.fit(df['x'], df['y'])

new_data_df = cudf.DataFrame({'inputs': inputs})
outputs_gpu = linear_regression.predict(
                  new_data_df[['inputs']])

 plt.scatter(x, y_noisy,
             label='empirical data points')
 plt.plot(x, y, color='black',
          label='true relationship')
 plt.plot(inputs, outputs, color='red',
          label='predicted relationship (cpu)')
 plt.plot(inputs, outputs_gpu.to_array(),
          color='green',
          label='predicted relationship (gpu)')
 plt.legend()
 plt.savefig("linear_regression_gpu.svg")
```



Other Domain Specific Libraries

- GridTools — used to accelerate the weather model COSMO
- QUDA — a library to build LQCD applications
- OpenMM — molecular simulations on AMD, Intel, and Nvidia
- …