GPU Accelerated Libraries

Jan H. Meinke | 03.02.2020

Discrete GPUs Have Their Own Memory

GPUs in HPC systems have their own (often very fast) memory. While some libraries take care of the necessary memory transfers, most of the ones I talk about do not.

Need to

- allocate memory on GPU
- transfer data to GPU
- transfer results back (if necessary)

or let the runtime manage this using cudaMallocManaged() or the managed attribute.

Allocating Memory on the GPU

C/C++

If memory is allocated using cudaMallocManaged data transfers between the host memory and the GPU memory can be managed automatically.

```
size_t N = 1024;
double* data = nullptr;
cudaMallocManaged(&data, N * sizeof(double);
```

Allocating Memory on the GPU

Fortran

In CUDA Fortran managed is an attribute of the array. The code above becomes:

```
integer N = 1024
real*8, managed :: data(1024)
```

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

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

ESM Libraries

- GridTools – used to accelerate COSMO
- …

You'll learn more about GridTools this afternoon.

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:

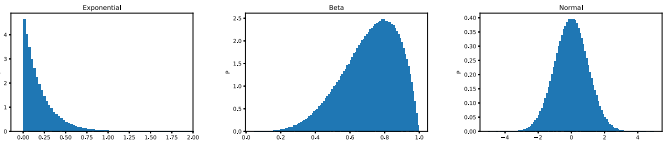
Routines to generate uniform random numbers (often integers) within a given interval.

- Mersenne Twister
- XORWOW
- Philox
- …

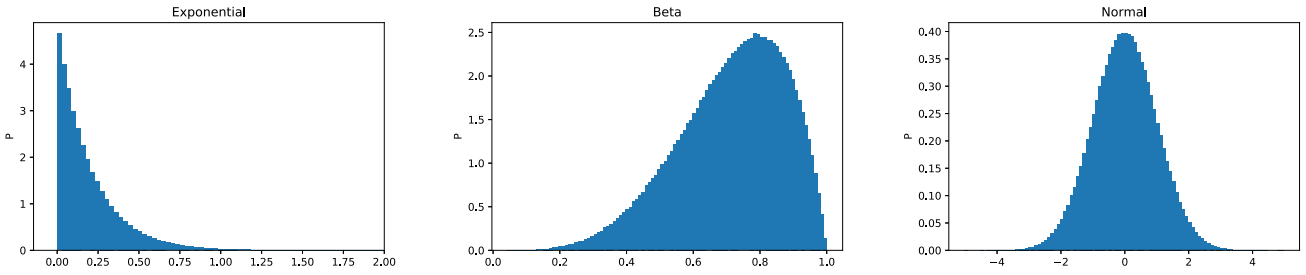
Distributions:

Generators are used to obtain samples from random distributions including

- uniform
- normal
- exponential
- beta
- …



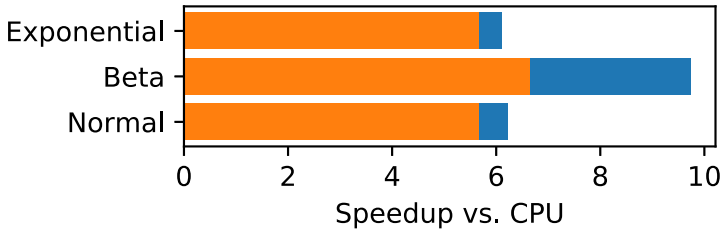
Nvidia cuRand vs. Intel MKL



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.

```
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. The speedup 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)

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

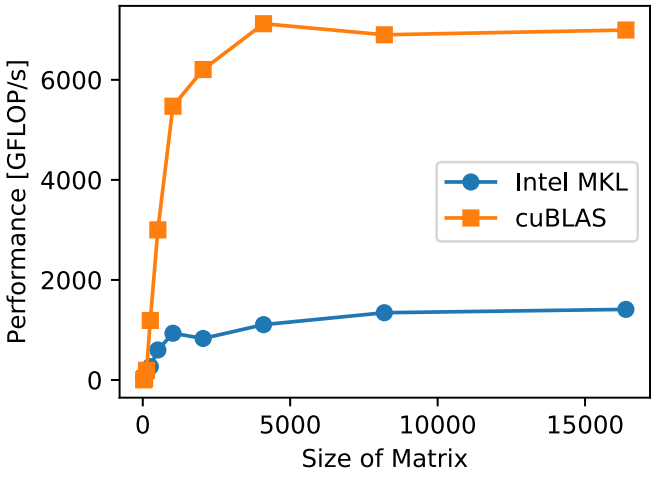
Nvidia cuBLAS vs. Intel MKL

DGEMM

The following script measures the time it takes to performa a matrix-matrix multiplication and using cuBLAS and the MKL. Both are called from Python, but you can expect similar performance difference when you call them from other languages.

> **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:

```
! 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.