Librerias y Python

cuBLAS

- BLAS: Basic Linear Algebra Subroutines
- Se puede usar operaciones de algebra lineal optimizadas con el GPU (un GPU o varios)
  - Nivel 1: vector-vector
  - Nivel 2: matriz-vector
  - Nivel 3: matriz-matriz

cuBLAS

- Ejemplo: Single-Precision General Matrix Multiplication (SGEMM), nivel 3
- cuBLAS es una parte del toolkit y es como usar una libreria externa de C/C++ (no requiere CUDA)

cuBLAS

```
cublasHandle_t handle;
cublasCreate(&handle);

.. { data operation } ..

cublasSgemm(...);

.. { data operation } ..

cublasDestroy(handle);
```

cuBLAS

- Implementa la operación:

$$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$$

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa,
                            cublasOperation_t transb,
                            int m, int n, int k,
                            const float *alpha,
                            const float *A, int lda,
                            const float *B, int ldb,
                            const float *beta,
                            float *C, int ldc);
```

cuBLAS

- transa, transb
  - determina si usamos las transpuestas de A y/o B
- m, n, k
  - número de elementos en cada dimensión de las matrices
- alpha, beta
  - parámetros de la operación SGEMM
- *A, *B, *C
  - *linear buffer* para los datos de la matriz (*column-major order)*
- lda
  - *leading column dimension* matriz A (cuBLAS alinea los elementos de las matrices con este valor)
- ldb
  - *leading column dimension* matriz B

cuBLAS

- cublasSetMatrix()
- cublasGetMatrix()
  - *wrapper* de cudaMemcpy()
- Ejemplo: cublasSgemm.cpp
  - las matrices están transpuestas

cuBLAS

- cuBLAS-XT: para varios GPUs
- cuBLAS permite computación de precisión mixta
- Si hay *tensor cores* en el GPU (Volta+), cuBLAS (v. 11.0) usará esos cores para los cálculos
  - para versiones anteriores es posible especificar el uso de los *tensor cores*

cuRAND

- Para la creación de números aleatorios en una forma paralela

```
curandGenerator_t curand_gen;
curandCreateGenerator(&curand_gen, CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(curand_gen, 2019UL);
```

- Para generar números distribuidos uniformemente:

```
curandGenerateUniform(curand_gen, random_numbers_on_device_memory, length);
```

cuRAND

- Hay varias funciones para generar números:
  - curandGenerateUniform()
  - curandGenerateNormal()
  - curandGenerateLogNormal()
  - curandGeneratePoisson()
  - curandGenerateUniformDouble()
  - curandGenerateNormalDouble()
  - curandGenerateLogNormalDouble()

cuRAND

- Ejemplo:
  - curand_host.cpp
    - números aleatorios en el host
  - curand_device.cu
    - números aleatrios en el *device* (en los kernels)

cuFFT

- FFT: *Fast Fourier Transform*
  - algoritmo para calcular la transformada de Fourier (discreta) numericamente
- Funciones de cuFFT corresponden a las de FFTW, una libreria estandar

cuFFT

- Hay que crear un “plan”: una definición de toda la información necesaria para el problema
  - cufftPlan1D(), cufftPlan2D(), cufftPlan3D()
  - cufftPlanMany() → varias transformadas en una llamada
  - Para datos de más de 4GB hay que poner “64” al final del nombre de la función
  - También se puede usar multi-GPU

cuFFT

- La operación del FFT está realizada por:
  - cufftExecC2C() → complejo a complejo
  - cufftExecR2C() → real a complejo
  - cufftExecC2R() → complejo a real
- C → Z, R → D para precisión doble

cuFFT

- La operación es *forward* (FFT) o *inverse* (IFFT)
  - en el programa tenemos un par de operaciones
- Para las transformadas C2R, R2C, hay que crear 2 planes para cada dirección
- Para C2C solo se requiere un plan
- Ejemplo: cufft.1d.cpp

cuDNN

- Libreria para *deep neural nets*
- Integrado en:
  - PyTorch
  - TensorFlow
  - Keras

Numba

- Módulo de Python para compilación del código
  - *Just-in-time* (JIT) compilador
  - Compatible con NumPy
  - Se puede compilar para el GPU (ocupa CUDA, pero no es necesario programar en CUDA)

Numba

```
from numba import vectorize
@vectorize([“float32(float32, float32, float32)”], target=’cuda’)
def saxpy(scala, a, b):
      return scala * a + b
```

- “@vectorize” es un *decorator*
  - especifica los tipos de los parámetros y el valor de retorno
  - hay 3 *targets*:
    - cuda → GPU
    - parallel → multi-core CPU
    - cpu → un thread

Numba

- Otra opción es usar @cuda.jit

```
from numba import cuda

@cuda.jit
def matmul(d_c, d_a, d_b):
    x, y = cuda.grid(2)
    if (x < d_c.shape[0] and y < d_c.shape[1]):
        sum = 0
        for k in range(d_a.shape[1]):
            sum += d_a[x, k] * d_b[k, y]
        d_c[x, y] = sum
```

- cuda.grid() da los índices de los threads al nivel del *grid*
  - se puede llamar con:
    - matmul[dimGrid, dimBlock](d_c, d_a, d_b)

Numba

- Ejemplo: numba_saxpy.py
  - @vectorize
- Ejemplo: numba_matmul.py
  - @cuda.jit

CuPy

- Aceleración de algebra lineal con el GPU (a través de CUDA)
- Compatible con NumPy
- Se puede aplicar 3 tipos de *kernel* con CuPy:
  - *Elementwise*: aplica una operación a cada elemento de un vector/matriz
  - *Reduction*: operación de reducción
  - *Raw*: se puede definir un *kernel* de CUDA directamente en Python
- Ejemplo: cupy_op.py

PyCUDA

- Acceso al API de CUDA en Python
- Para escribir CUDA C/C++ en Python
- Ejemplo: pycuda_matmul.py