nVidia CUDA Toolkit Libraries

ECE 6397
Electrical and Computer Engineering
University of Houston
Dr. David Mayerich
Standard Libraries (CUDA 10.0)

- CUDA math library – C99 mathematical functions
- cuBLAS – basic linear algebra subroutines
- cuFFT – fast Fourier transform
- cuSPARSE – sparse matrix subroutines
- cuSOLVER – dense and sparse direct solvers
- cuRAND – GPU random number generator
- NPP – image and video processing primitives
- nvGRAPH – graph analytics library
- Thrust – templated parallel algorithms and data structures
CUDA Math Library

• IEEE 754 compliance
  • IEEE 754-1985 Binary Floating-Point Arithmetic
  • standardizes floating point approximation

• Importance of compliance
  • heterogeneous systems rely on different hardware platforms
  • variations in compliance can generate different results across multiple executions

• Supported operations
  • arithmetic: addition (+), subtraction (-), multiplication (*), division (/)
    • sign changes (-)
    • fused multiply add (fma)
  • comparisons (<, >, ==, <=, >=)
  • functions: modulus (fmod), square root (sqrt)
  • conversions:
    
    ```
    double pi_d = 3.141592653589793;
    float pi_f = (double) pi_d;
    ```
Floating Point

- Floating point format: 
  \[ (-1)^s \times m \times b^x \]
  - \( s \) controls the sign of the number
  - \( b \) is the number base (2 in binary systems)
  - \( m \) is the mantissa – controls the numerical value of the floating point number
    - \( \frac{1}{b} \leq m < 1 \) (the mantissa is normalized)
  - \( x \) is the exponent – controls the magnitude of the floating point number

- All floating point number have a finite number of potential values
  - indicated by the number of bits (binary digits) in the mantissa and exponent

- precision – the accuracy of a floating point number system
  - usually specified by the number of bits (binary digits) used to represent the number
Precision (base 10, decimal)

Examples

```plaintext
float pi = 3.14159;
  (-1)^0 \times 3.14159 \times 10^0

float f = -224.3377;
  (-1)^1 \times 2.243377 \times 10^2
```

• The number of digits in the mantissa indicate the number of discrete digit sequences that can be represented
  • 3 digits can be used to represent:
    3.11
    31.1
    3100000000
    0.0000000311

• The number of digits in the exponent indicate the largest and smallest magnitude values
  • 3 digits in the exponent with $m = 311$
    smallest magnitude value: $3.11 \times 10^{000} = 3.11$
    largest magnitude value: $3.11 \times 10^{999} = \text{a super massive number}$
Bias

• Note that the smallest value for the exponent is 1
  • we can include a sign bit in the exponent to represent values < 1
  • instead we introduce a bias
    \((-1)^s \times m \times 10^{x-500}\)

• Including a bias term allows small numbers without a sign bit
  • 3-digit exponent with \(m = 3.11\) and a bias of 500
    
    smallest magnitude value: \(3.11 \times 10^{-500}\)
    
    largest magnitude value: \(3.11 \times 10^{999-500} = 3.11 \times 10^{499}\)
IEEE 754 Floating Point

• Half precision (16-bit)
  • 1 sign bit (0 = positive, 1 = negative)
  • 10 binary digits in the mantissa
  • 5 binary digits in the exponent
  • 15 (decimal) exponent bias

• Single precision (32-bit)
  • 1 sign bit
  • 23 bit mantissa
  • 8 bit exponent
  • 127 bias

• Double precision (64-bit)
  • 1 sign bit
  • 53 bit mantissa
  • 11 bit exponent
  • 1023 bias
Compute capability and floating point

• compute capability < 1.2
  • single precision (32-bit) IEEE 754 support

• compute capability 1.3
  • double precision (64-bit) IEEE 754 support
  • fused multiply-add (FMA) in 64-bit:
    calculation of $a * b + c$ with only one rounding step

• compute capability 2.0
  • FMA in 32 and 64-bit

• compute capability 5.3
  • half precision (16-bit)
Testing Floating Point Attributes

• `bool isfinite(float x)` – returns true if \( x \) is finite (not Inf or NaN)
• `bool isinf(float x)` – returns true if \( x \) is Inf
  • Inf occurs in some circumstances: \( \log(0) \)
• `bool isnan(float x)` – returns true if \( x \) is NaN
  • NaN values occur for invalid operations: \( \sqrt{x} \) where \( x < 0; \frac{x}{0} \)
• `__RETURN_TYPE signbit(float x)` – returns the sign bit of \( x \)
  • `__RETURN_TYPE` is `bool` for MSVS, `int` otherwise
CUDA Math Library Functions

• trigonometric functions
  • \texttt{cospi( float x )} – calculate $\pi \cdot \cos x$
  • \texttt{sinpi( float x )} – calculate $\pi \cdot \sin x$
  • \texttt{sincos( float x, float* s, float* c)} – calculate $\sin x$ and $\cos x$

• mathematical functions
  • \texttt{erf( float x )} – calculates $\frac{1}{\sqrt{\pi}} \int_{-x}^{x} e^{-t^2} dt$
  • \texttt{erfinv( float y )} – calculates $x = \text{erf}^{-1} y$
  • \texttt{rsqrt( float x )} – calculates $\frac{1}{\sqrt{x}}$
  • \texttt{rcbrt( float x )} – calculates $\frac{1}{\sqrt[3]{x}}$

• Bessel functions
  • \texttt{j0( float x )}, \texttt{j1( float x )}, \texttt{jn( float x )}
  • \texttt{y0( float x )}, \texttt{y1( float x )}, \texttt{yn( float x )}
CUDA Basic Linear Algebra Subprograms

• BLAS – FORTRAN specification for low-level linear algebra subroutines
• Libraries used in most high-performance packages
  • LAPACK, GNU Octave, MATLAB

• Level 1
  • vector subroutines that operate on a linear array
  • dot products ($\mathbf{x} \cdot \mathbf{y}$); vector norms $||\mathbf{x}||$
  • generalized vector addition $\text{axpy}()$: $\mathbf{y} = \alpha \mathbf{x} + \mathbf{y}$

• Level 2
  • matrix-vector operations
  • generalized matrix-vector multiplication $\text{gemv}()$: $\mathbf{y} = \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}$
  • solver for $\mathbf{y} = \mathbf{T} \mathbf{x}$ where $\mathbf{T}$ is triangular
CUDA Basic Linear Algebra Subprograms

• Level 3
  • matrix-matrix operations
  • general matrix multiplication gemm(): $C = \alpha AB + \beta C$
  • solver for $B = \alpha T^{-1}B$ where $T$ is triangular

• nVidia provides a CUDA implementation through cuBLAS
  • available with the CUDA toolkit
Initializing cuBLAS

- `cudaError_t cublasCreate(cublasHandle_t* h)`
  - initialize cuBLAS and returns a handle containing cuBLAS device information
  - use the `cublasHandle_t` value to execute cuBLAS functions

- `cudaError_t cublasDestroy(cublasHandle_t h)`
  - destroy a cuBLAS handle and free any required resources

```c
float* x = (float*) malloc(N * sizeof(float)); //allocate host arrays
...
cublasHandle_t handle;
cublasCreate(&handle); //initialize cuBLAS
float* gpu_x;
cudaMalloc(&gpu_x, N * sizeof(float));
cudaMemcpy(gpu_x, x, N * sizeof(float)); //copy array to the device
... //call cuBLAS functions
```
Calling cuBLAS functions

• function calls use device pointers – copy data to the GPU first
• functions that return scalar values (min, max, sum, etc.) can accept a host pointer for the return value
  • basically, you don’t have to copy a scalar value from the device
• cuBLAS calls follow a predictable format
  • a character identifier in the function signature indicates data type
    • s – single precision (32-bit) real
    • d – double precision (64-bit) real
    • c – single precision complex (2 x 32-bit values per element)
    • z – double precision complex
Level 1: Index Functions

- Functions that return indices to array elements
  \[ \text{cublasI}<t>a------() \]

  \[ \text{cublasIsamin(cublasHandle_t h, int n, const float* x, int incx, int* result)} \]
  - returns the index to the smallest \( i \) such that \( |x_i| \leq \forall x \in x \)
  - \( \text{incx} \) specifies the stride, or spacing between array elements (usually 1)

- Indices are 1-based, following FORTRAN convention!!!
CUDA defines `cuComplex` and `cuDoubleComplex` for simplicity
  - both are two-element structures

Find the largest value $|x_i| \in \mathbf{x}$ where $\mathbf{x} \in \mathbb{C}^N$

```c
cuComplex* x = (cuComplex *) malloc(N * sizeof(cuComplex));
...
//load array elements
float* gpu_x;
cudaMalloc(&gpu_x, N * sizeof(cuComplex)); //allocate device memory
cudaMemcpy(gpu_x, x, N * sizeof(cuComplex), cudaMemcpyHostToDevice);
int i;
cublasIcamin(handle, N, gpu_x, 1, i); //call the cuBLAS function
```
Level 1: Scalar Functions

cublas<t>a----() 

• For <t>: 
  • S – single precision real 
  • D – double precision real 
  • Sc – single precision complex 
  • Dz – double precision complex 

cublasSasum(cublasHandle_t h, int n, const float* x, int incx, float* y)  
cublasDzasum(cublasHandle_t h, int n, const cuDoubleComplex* x, int incx, double* y) 

\[ y = \sum_{i=1}^{N} |x_i| \] 

• Note that y can be a host or device pointer 
  • cuBLAS will automatically detect the location
Level 3: General Matrix Multiplication

cublas<t>gemm()

• \(<t> = 'H' offers support for half-precision (16-bit) floating point

\[ C = \alpha \cdot p_a(A) \cdot p_b(B) + \beta C \]

cublasSgemm(  cublasHandle_t h,
               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

);
Level 3: General Matrix Multiplication

• gemm allows an operation to be applied to each input matrix:
  • CUBLAS_OP_N  do nothing
  • CUBLAS_OP_T  transpose the matrix

• Calculate $\mathbf{C} = \mathbf{A} \mathbf{B}$

```c
float* A = (float*) malloc(M*K*sizeof(float));
float* B = (float*) malloc(K*N*sizeof(float));
float* C = (float*) malloc(M*N*sizeof(float));
...
    //load arrays, generate handle, copy to device
float alpha = 1.0;    //no scalar factors are applied
float beta = 1.0;
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
            M, N, K, &alpha,
            gpu_A, m, gpu_B, k, &beta, gpu_C, m);
cudaMemcpy(C, gpu_C, M*N*sizeof(float), cudaMemcpyDeviceToHost);
```
Fourier Transform

- Fourier transform of a function \( f(t) = \hat{f}(\xi) \):

\[
\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} \, dx
\]

\[
f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{2\pi i x \xi} \, d\xi
\]
Fourier Transform as a Rotation

• See also:

https://www.youtube.com/watch?v=NAsm30MAHlg

source: /u/wave_equation (Reddit)
Discrete Fourier Transform

• DFT definition – $O(n^2)$

$$X_v = \sum_{t=0}^{T-1} x_t e^{-\frac{2\pi i}{T} n t}$$

$v = [0, \cdots, N - 1]$

• Fast Fourier Transform (FFT) – $O(n \log n)$
  • Cooly – Turkey algorithm
  • recursively divide the FFT into blocks of size $N/2$
FFT W

• Fastest Fourier Transform in the West (FFTW)
  • fastest open-source FFT (used in MATLAB)
  • developed by Frigo and Johnson (MIT)
  • uses Cooley – Turkey
  • written in C: http://www.fftw.org/

#include<fftw3.h>

... 

fftw_complex* f = (fftw_complex*) malloc(N * sizeof(fftw_complex));
fftw_complex* g = (fftw_complex*) malloc(N * sizeof(fftw_complex));

  //create an FFTW handle (allocate FFTW data and resources)
  fftw_plan p = fftw_plan_dft_1d(N, f, g, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute(p); //perform the DFT
  fftw_destroy_plan(p); //free memory for the fftw handle
...

  //use result, free memory...
cuFFT

• CUDA DFT implementation

```
#include<cufft.h>
...
cufftComplex* f = (cufftComplex*) malloc(N * sizeof(cufftComplex));
... //load data
cufftHandle h; //create a cuFFT handle
cufftComplex* gpu_f; //create a device pointer
cudaMalloc(&gpu_f, N * sizeof(cufftComplex)); //allocate device space memory
cudaMemcpy(gpu_f, f, N * sizeof(cufftComplex), cudaMemcpyHostToDevice);
cufftPlan1d(&h, N, CUFFT_C2C, 1); //allocate cuFFT resources
cufftExecC2C(h, gpu_f, gpu_f, CUFFT_FORWARD); //perform the DFT
... //use the data, free memory...
```