CUDA Libraries

cuBLAS, cuRAND, Thrust

GPU Programming

http://cuda.nik.uni-obuda.hu

Szénási Sándor
szenasi.sandor@nik.uni-obuda.hu

GPU Education Center of Óbuda University

NVIDIA GPU EDUCATION CENTER

GPU Accelerated Libraries

Available libraries

- cuDNN - Deep neural networks
- cuFFT - Fast Fourier Transformation
- nvGRAPH - Graph analytics
- NPP - Image processing primitives
- CULA - Linear algebra
- MAGMA - Next gen. linear algebra
- cuSOLVER - Dense and sparse direct solvers
- cuSPARSE - Sparse matrix library
- cuBLAS - BLAS library implementation
- cuRAND - Random number generation
- Thrust - Parallel algorithms and data structures
- OpenCV - Computer vision
- GPP - Geometry engine
- etc.

More information

- https://developer.nvidia.com/gpu-accelerated-libraries

CUDA Libraries

cuBLAS

cuRAND

Thrust

cuBLAS Library

BLAS

- Basic Linear Algebra Subprograms (BLAS) [28] is a de facto application programming interface standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplication.
- Heavily used in high-performance computing, highly optimized implementations of the BLAS interface have been developed by hardware vendors such as by Intel and Nvidia.

cuBLAS

- CUDA BLAS library
- cuBLAS is an implementation of the BLAS library based on the CUDA driver and framework. It has some easy to use data types and functions. The library is self-contained in the API level, so the CUDA is driver is unnecessary
- Versions
  - Legacy version
  - New BLAS2
- Technical details
  - The interface to the cuBLAS library is the header file cublas.h
  - Applications using cuBLAS need to link against the DSO the DLL cublas.dll (for Windows applications) when building for the device,
  - and against the DSO the DLL cublasemu.dll (for Windows applications) when building for device emulation.

Compiling and linking

Header files

- One of the following headers must be included
  - Legacy: cublas.h
  - New: cublas_v2.h

Compiling and linking

- Additional library file
  - cublas.lib

Execution

- Required DLL file
  - using CUDA capable cards
  - cublas.dll (for Windows applications)
- Emulation
  - without CUDA capable cards
  - cublasemu.dll (for Windows applications)

Developing cuBLAS based applications

Step 1 - Create cuBLAS data structures

- cuBLAS provides functions to create and destroy objects in the GPU space
- There are not any special types (like matrices or vector), the library functions usually needs typed pointers to the data structures

Step 2 - Fill structures with data

- There are some functions to handle data transfers between the system memory and the GPU memory

Step 3 - Call cuBLAS function(s)

- The developer can call a cuBLAS function, or a sequence of these functions

Step 4 - Retrieve results to system memory

- Finally the developer can upload the function results from the GPU memory to system memory

cuBLAS helper functions

Initialize cuBLAS

- Function name: cublasCreate
- Parameters
  - handle - Address of an uninitialized cuBLAS handler
- This function initializes the cuBLAS library and creates a handle to an opaque structure holding the cuBLAS library context. It allocates hardware resources on the host and device and must be called prior to making any other cuBLAS library calls.
- Result: cublasStatus
- Return values:
  - CUBLAS_STATUS_NOT_INITIALIZED: CUDA Runtime init failed
  - CUBLAS_STATUS_ALLOC_FAILED: if resources could not be allocated
  - CUBLAS_STATUS_SUCCESS: if cuBLAS library initialized successfully

Shut down cuBLAS

- Function name: cublasDestroy
- Parameters
  - handle - Address of an initialized cuBLAS handler
- This function releases hardware resources used by the cuBLAS library. This function is usually the last call with a particular handle to the cuBLAS library.
- Result: cublasStatus

cuBLAS function result

Check error codes

- Function name: cublasGetError
- Returns the last error that occurred on invocation of any of the cuBLAS core functions (helper functions return the status directly, the core functions do not)
- The type cublasStatus is used for function status returns
- Currently, the following values are defined:

cuBLAS error codes	
CUBLAS_STATUS_SUCCESS	Operation completed successfully
CUBLAS_STATUS_NOT_INITIALIZED	cuBLAS library not initialized
CUBLAS_STATUS_ALLOC_FAILED	Resource allocation failed
CUBLAS_STATUS_INVALID_VALUE	Unsupported numerical value was passed to function
CUBLAS_STATUS_ARCH_MISMATCH	Function requires an architectural feature absent from the architecture of the device
CUBLAS_STATUS_MAPPING_ERROR	Access to GPU memory space failed
CUBLAS_STATUS_EXECUTION_FAILED	GPU program failed to execute
CUBLAS_STATUS_INTERNAL_ERROR	An internal cuBLAS operation failed

cuBLAS memory handling

Allocate memory

- Function name: cublasAlloc
- Parameters
  - n - Number of items
  - elemSize - Size of items
  - ptr - Void pointer address
- Creates an object in GPU memory space capable of holding an array of n elements, where each element's size is elemSize byte. The result of the function is the common status code, the ptr pointer points to the new allocated memory space
- Result: cublasStatus

Free memory

- Function name: cublasFree
- Parameters
  - ptr - Pointer to memory region to free
- Deallocates the object in the GPU memory referenced by the ptr pointer
- Result: cublasStatus

Transfer vectors

Host to device

- Function name: cublasSetVector
- Parameters
  - n - Item count
  - elemSize - Item size
  - x - Source pointer
  - incx - Source storage space increment
  - y - Destination pointer
  - incy - Destination storage space increment
- The function copies n elements from a vector in the system memory (pointed by x reference) to the y vector in the GPU memory (pointed by the y reference). Storage spacing between elements is incx in the source vector and incy in the destination vector

Device to host

- Function name: cublasGetVector
- Parameters
  - same as above
- Similar to cublasSetVector function. It copies n elements from a vector in the GPU memory (pointed by x reference) to the y vector in the system memory (pointed by the y reference).

Transfer matrices

Host to device

- Function name: cublasSetMatrix
- Parameters
  - rows - Number of rows
  - cols - Number of columns
  - elemSize - Item size
  - A - Source matrix
  - lda - Source leading dimension
  - B - Destination matrix
  - ldb - Destination leading dimension
- Copies a tile of rows x cols elements from a matrix A in host memory space to a matrix B in GPU memory space. Each element requires storage of elemSize bytes and that both matrices are stored in **column-major format**, with the leading dimension of the source matrix A and destination matrix B given in lda and ldb.

Device to host

- Function name: cublasGetMatrix
- Parameters
  - same as above
- Similar to cublasSetMatrix function.

BLAS functions overview

BLAS level 1 functions

- Scalar and vector based operations
- This level contains vector operations of the form as well as scalar dot products and vector norms, among other things
- Functions are grouped into subgroups by the operand types
  - Single-precision BLAS1 functions
  - Single-precision complex BLAS1 functions
  - Double-precision BLAS1 functions
  - Double-precision complex BLAS1 functions

BLAS level 2 functions

- Matrix vector operations
- This level contains matrix –vector operations, solving equals, among other things

BLAS level 3 functions

- Matrix-matrix operations
- This level contains matrix –matrix operations. This level contains the widely used general matrix multiply operation

Some cuBLAS level 1 functions

Index of maximum element (single)

- Function name: cublasIsamax
- Parameters:
  - handle - cuBLAS handler
  - n - Number of elements in input vector
  - x - Single-precision vector with n elements
  - incx - Storage spacing between elements of x
  - result - Address for result (result is 1-based indexing!)
- Result:
  - cuBLAS error code

Variants based on element types

- Double values
  - Function name: cublasIdamax
- Single complex values
  - Function name: cublasIcamax
- Double complex values
  - Function name: cublasIzamax

Some cuBLAS level 1 functions (2)

Vector scalar multiplication and addition

- Function name: cublasSaxpy
- Parameters:
  - handle - cuBLAS handler
  - n - Number of elements in vector x and y
  - alpha - The scalar alpha
  - x - The x vector
  - incx - Stride between consecutive elements of x
  - y - The y vector
  - incy - Stride between consecutive elements of y
- Based on float values
- This function multiplies the vector x by the scalar α and adds it to the vector y overwriting the latest vector with the result:
  y = α * x + y

See the cuBLAS library documentation for full list of available functions

- http://docs.nvidia.com/cuda/cublas

Some cuBLAS level 2 functions

Vector scalar multiplication and addition

- Function name: cublasSgemv
- Parameters
  - handle - cuBLAS handler
  - trans - Operation for A (cublasOperation_t)
  - m - Number of rows of A
  - n - Number of columns of A
  - alpha - The alpha scalar
  - A - The A matrix
  - lda - Leading dimension of A
  - x - The vector x
  - incx - Stride for x
  - beta - The beta scalar
  - y - The vector y
  - incy - Stride for y
- Calculates the following:
  y = α * op(A) * x + β * y

Some cuBLAS level 3 functions

Vector scalar multiplication and addition

- Function name: cublasSgemm
- Parameters
  - handle - cuBLAS handler
  - transa - Operation op(A)
  - transa - Operation op(B)
  - m - Number of rows of matrix op(A) and C.
  - n - Number of columns of matrix op(B) and C.
  - k - Number of columns of op(A) and rows of op(B).
  - alpha – The alpha scalar
  - A – The A matrix
  - lda - Leading dimension of A
  - B – The B matrix
  - ldb - Leading dimension of B
  - beta – The beta scalar
  - C – The C matrix
  - ldc - Leading dimension of C
- Performs the following matrix-matrix operation:
  C = α* op(A) * op(B) + β* C

CUDA Libraries

cuBLAS

cuRAND

Thrust

cuRAND Library

cuRAND

- The cuRAND library provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers
  - A pseudorandom sequence of numbers satisfies most of the statistical properties of a truly random sequence but is generated by a deterministic algorithm
  - A quasirandom sequence of n-dimensional points is generated by a deterministic algorithm designed to fill an n-dimensional space evenly

Two APIs

- Host side
  - Started from CPU call and generate random numbers in GPU memory
  - These numbers are usable from any kernels
  - Header file: curand.h
- Device side
  - Generate random numbers from kernels
  - Header file: curand_kernel.h

Device API Initialisation

State objects

- cuRAND uses a curandState_t type to keep track of the state of the random sequence
- In the case of multiple threads each of them needs its own curandState_t

Initiating curandState

- Function name: curand_init
- Parameters
  - seed - A seed determined the beginning point of the random sequence
  - sequence – Similar to seed
  - offset – Amount to skip ahead in the random sequence
  - state – A pointer to a curandState_t variable

Example

```
__global__ void RandomInit(int seed, curandState *state) {
        int ind = blockIdx.x * blockDim.x + threadIdx.x;
        curand_init(seed, ind, 0, &state[ind]);
}

RandomInit <<<100, 100 >>>(time(NULL), dev_state);
```

Generate random numbers

Basic random

- Function name: curand
- Parameters
  - curandState_t - Reference to the state object

Uniform distribution

- Function name: curand_uniform
- Parameters
  - curandState_t - Reference to the state object

Normal distribution

- Function name: curand_normal
- Parameters
  - curandState_t - Reference to the state object

Poisson distribution

- Function name: curand_poisson
- Parameters
  - curandState_t - Reference to the state object
  - lambda - Lambda of distribution

Generate random numbers

Double precision

- Function names are similar:
  - curand_uniform_double
  - curand_normal_double
  - curand_log_normal_double

Generate multiple floats

- Function names are similar:
  - curand_normal2
  - curand_log_normal2
  - curand_normal2_double
  - curand_log_normal2_double
- The results are float2 value pairs

Another generators

- Different generator algorithms
- Quasirandom sequences
- Discrete distributions

CUDA Libraries

cuBLAS

CUFFT

Thrust

Thrust

Template library

- Thrust is a C++ template library for CUDA based on the Standard Template Library (STL)
- It allows to implement high performance parallel applications through a high-level interface that is based on CUDA C
- It is the part of the standard CUDA Toolkit

Newer versions

- Thrust is available from
  - http://thrust.github.com/
- Documentation
  - https://thrust.github.io/doc

Main modules

- Containers
- Algorithms
- Function objects
- Iterators/Fancy iterators
- Random number generation
- …

Thrust containers

Used memory region

- Device containers
  - thrust::device_vector<T, Alloc >
- Host containers
  - thrust::host_vector<T, Alloc>

Create device vector

- device_vector (void)
  - Creates an empty vector
- device_vector (size_type n)
  - Creates a vector with the given size
- device_vector (size_type n, const value_type &value)
  - Creates a vector with the given size and copy the given value
- device_vector (const device_vector &v)
  - Creates and copies elements from the given device vector
- device_vector (const host_vector< OtherT, OtherAlloc > &v)
  - Creates and copies elements from the given host vector
- operator=
  - Creates and copies from device or host vector
- ...

Access elements in device vector

Global operators

- size( )
  - Returns the number of elements in this vector
- capacity( )
  - Returns the number of elements which have been reserved in this vector
- resize(size_type new_size, const value_type &x=value_type())
  - Resizes the vector to the given size
- clear( )
  - Resizes the vector to 0

Work with elements

- push_back(const value_type &x)
  - This method appends the given element to the end of this vector
- insert(iterator position, const T &x)
  - This method inserts a single copy of a given exemplar value at the specified position in this vector
- erase(iterator pos)
  - This method removes the element at position pos
- operator[]
  - Subscript access to the data contained in this vector

Device iterators

Acquire iterators

- begin( )
  - This method returns an iterator pointing to the beginning of this vector
- end( )
  - This method returns an iterator pointing to one element **past** the last of this vector
- rend( )
  - This method returns a reverse_iterator pointing to one element past the last of this vector's reversed sequence.

```
thrust::device_vector<int> data;
data.push_back(value);

auto it = d_data.begin( );
while (it != d_data.end( )) {
    printf("%d\n", (int)*it);
    it++;
}

for (auto p : d_data)
    printf("%d\n", (int)p);
```

Search in a vector

- Function name: thrust::find
- Parameters:
  - first - An iterator to the first item
  - last - An iterator to the last item
  - value - Value to find
- Find returns the first iterator i in the range [first, last) such that *i == value or last if no such iterator exists.
- The result is an iterator
- Result value == last if it has not found any corresponding items

Sort vector

- Function name: thrust::sort
- Parameters:
  - first - A RandomAccessIterator to the first item
  - last - A RandomAccessIterator to the last item
- Sort sorts the elements in [first, last) into ascending order. It uses the operator< to sort
- Useful variant: thrust::sort_by_key

Copy and fill

Copy a vector

- Function name: thrust::copy
- Parameters:
  - first - An iterator to the first item (source)
  - last - An iterator to the last item (source)
  - result - An iterator for the output (destination)
- Copies elements from the range [first, last) to the range [result, result + (last - first))

Fill a vector

- Function name: thrust::fill
- Parameters:
  - first - An iterator to the first item
  - last - An iterator to the last item
  - value - Value to store
- Assigns the value to every element in the range [first, last)

More algorithms

- Modify all items
- Replace items

Reductions

General reduction

- Function name: thrust::reduce
- Parameters:
  - first - An iterator to the first item
  - last - An iterator to the last item
  - init - Init value
  - oper - Binary operation
- Aa generalization of summation: it computes the sum (or some other binary operation) of all the elements in the range [first, last)

Binary operation can be

- Arithmetic operations
  - thrust::plus<T>
  - thrust::multiplies<T>
  - …
- Comparision operations
- Logical operations
- Bitwise operations
- Custom operations

Special reductions

Counting

- Function name: thrust::count
- Parameters:
  - first - An iterator to the first item
  - last - An iterator to the last item
  - value - Value to count
- Count finds the number of elements in [first,last) that are equal to value
- Variant: thrust::count_if

Min-max

- Function name: thrust::min_element / thrust::max_element
- Parameters:
  - first - An iterator to the first item
  - last - An iterator to the last item
- Finds the smallest element in the range [first, last). It returns an iterator
- Variant: thrust::minmax_element

Logical

- all/any/none

Pointers

Raw pointer to Thrust pointer

- Passing a raw CUDA pointer as an argument to a Thrust function
- It must be wrapped using thrust::device_ptr

```
size_t N = 10;
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

thrust::device_ptr<int> dev_ptr(raw_ptr);

thrust::fill(dev_ptr, dev_ptr + N, (int) 0);
```

Thrust pointer to raw pointer

- Opposite direction

```
size_t N = 10;
thrust::device_ptr<int> dev_ptr = thrust::device_malloc<int>(N);
int * raw_ptr = thrust::raw_pointer_cast(dev_ptr);
```