GPU Teaching Kit
Accelerated Computing

Module 25 – CUDA libraries
Lecture 25.1 – cuBLAS

Objective

- To learn how to utilize the cuBLAS library
  - Learn about BLAS
  - Learn about dense matrix storage schemes
  - Learn how to initialize and terminate the cuBLAS environment

BLAS and cuBLAS

- BLAS (Basic Linear Algebra Subprograms) is a low-level linear algebra library originally written in Fortran and standardized by the BLAS Technical Forum
- It provides three levels of routines:
  - Level 1: Scalar and Vector-Vector operations
    - Example: Dot product and SAXPY
  - Level 2: Matrix-Vector operations
    - Example: Matrix vector multiplication, solving a triangular system
  - Level 3: Matrix-Matrix operations
    - Example: GEMM
- cuBLAS is a BLAS implementation for CUDA devices

Data layout in cuBLAS

- There are two natural data layouts to linearly store dense matrices:
  - Row major order:
    - Each row is stored contiguously
    - Used by C, C++ and derivatives
  - Column mayor order:
    - Each column is stored contiguously
    - Used by Fortran and derivatives
- cuBLAS uses column major order with 1 indexing for compatibility with Fortran numeric libraries

$$\begin{array}{cc} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{array}$$

Matrix A

$A_{1,1}$	$A_{1,2}$	$A_{2,1}$	$A_{2,2}$
$A[0]$	$A[1]$	$A[2]$	$A[3]$

Row major order with 0 indexing

$$A_{i,j} = A[i * cols + j]$$

Addressing in row major order

$$\begin{array}{cc} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{array}$$

Matrix A

$A_{1,1}$	$A_{2,1}$	$A_{1,2}$	$A_{2,2}$
$A[1]$	$A[2]$	$A[3]$	$A[4]$

Column major order with 1 indexing

$$A_{i,j} = A[j * rows + i]$$

Addressing in column major order

Creation and destruction of a cuBLAS environment

- cuBLAS needs an execution context to store internal resources. This context needs to be created before executing any cuBLAS routine

- After all cuBLAS executions are finished the context needs to be destroyed to free resources

- Creating and destroying contexts should be considered an expensive operation. Recommended that each thread and each device have its own context

- To create a context in a specific device call cudaSetDevice before the creation

- cublasHandle_t
  - Type used by cuBLAS to store contexts

- cublasCreate(cublasHandle_t* handle)
  - Creates a cuBLAS context
  - Parameters:
    - Pointer to cuBLAS handle to create

- cublasDestroy(cublasHandle_t handle)
  - Destroys a cuBLAS context
  - Parameters:
    - cuBLAS handle with the context to destroy

- cublasStatus_t
  - Type used by cuBLAS for reporting errors
  - Every cuBLAS returns an error status

cuBLAS streams API and thread safety

- cuBLAS permits the use of cuda streams (cudaStream_t) for increasing resource usage and introduce other levels of parallelism
- cuBLAS is a thread safe library, meaning that the cuBLAS host functions can be called from multiple threads safely

- cublasSetStream():
  - Sets the stream to be used by cuBLAS for subsequent computations
  - Parameters:
    - cuBLAS handle to set the stream
    - cuda stream to use
- cublasGetStream():
  - Gets the stream being used by cuBLAS
  - Parameters:
    - cuBLAS handle to get the stream
    - pointer to cuda stream

cuBLAS API naming convention for BLAS routines

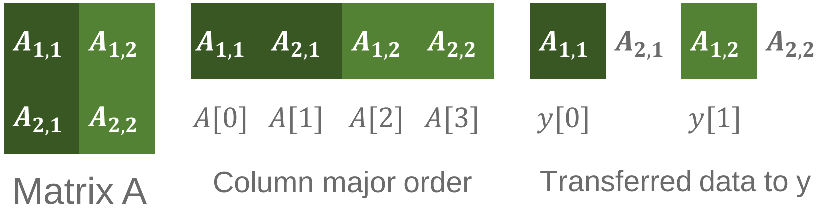
- Each of the three levels of BLAS routines in cuBLAS have multiple interfaces for the same operation, having the naming convention:
- cublas<t>operation where <t> is one of:
  - S for float parameters
  - D for double parameters
  - C for complex float parameters
  - Z for complex double parameters
- Example: For the axpy operation ($\mathrm{y[i]} = \alpha x[i] + y[i]$), the available functions are:
  - cublasSaxpy, cublasDaxpy, cublasCaxpy, cublasZaxpy

cuBLAS memory API

- cuBLAS offers specialized data migration and copy functions for strided matrix and vector transfers
- Available functions:
  - cublasGetVector & cudaGetMatrix for device to host transfers
  - cublasSetVector & cudaSetMatrix for host to device transfers
  - cublas<t>copy for device to device transfers
- Useful for obtaining a row in a matrix with column major order

cuBLAS memory API

– cublasGetVector():
  – Parameters:
    – Number of elements to transfer in bytes
    – Element size in bytes
    – Source device pointer
    – Stride to use for the source vector
    – Destination host pointer
    – Stride to use for the destination vector



– Example

– cublasGetVector(2*sizeof(float), sizeof(float), A, 2, y, 1);

cuBLAS example: Conjugate Gradient

Conjugate Gradient

- The Conjugate Gradient method is an iterative method to compute an approximation for the solution of the linear algebraic system $\mathrm{Ax} = b$
- Assumptions:
  - Let A be a $n * n$ symmetric and positive definite matrix (all the eigenvalues are positive)
  - Let b be a n-dimensional vector

Algorithm:

1. $r_0 = b - A * x_0$
2. $p_0 = r_0$
3. $k = 0$
4. loop
   1. $\alpha_k = \frac{r_k^T * r_k}{p_k^T * A * p_k}$
   2. $x_{k+1} = x_k + \alpha_k p_k$
   3. $r_{k+1} = r_k - \alpha_k A * p_k$
   4. if $\|r_{k+1}\|_2 < \varepsilon$, break the loop
   5. $\beta_k = \frac{r_{k+1}^T * r_{k+1}}{r_k^T * r_k}$
   6. $p_{k+1} = r_{k+1} + \beta_k p_k$
   7. $k = k + 1$
5. return x_{k+1}

cuBLAS implementation of the Conjugate Gradient method

```
double zero = 0, minusOne = -1, one = 1, alpha = 0, beta = 0, rxr = 0, tmp;
cublasDcopy(handle, n, b, 1, r, 1);                                          // $r_0 = b$
cublasDgemv(handle, CUBLAS_OP_N, n, n, &minusOne, A, rows, x, 1, &one, r, 1);  // $r_0 = b - A * x_0$
cublasDcopy(handle, n, r, 1, p, 1);                                          // $p_0 = r_0$
cublasDdot(handle, n, r, 1, r, 1, &rxr);                                     // $r_k^T * r_k$
while(k < maxit) {
  cublasDgemv(handle, CUBLAS_OP_N, n, n, &minusOne, A, rows, p, 1, &zero, Axp, 1);  // $A * p_k$
  cublasDdot(handle, n, p, 1, Axp, 1, &tmp);                                 // $p_k^T * A * p_k$
  alpha = rxr / tmp;
  cublasDaxpy(handle, n, &alpha, p, 1, x, 1);                                // $x_{k+1} = x_k + \alpha_k p_k$
  tmp = -alpha;
  cublasDaxpy(handle, n, &tmp, Axp, 1, r, 1);                                // $r_{k+1} = r_k - \alpha_k A * p_k$
  cublasDdot(handle, n, r, 1, r, 1, &tmp);                                   // $r_k^T * r_k$
  if (sqrt(tmp) < epsilon) break;
  beta = tmp / rxr;
  rxr = tmp;
  cublasDscal(handle, n, beta, p, 1);                                        // $\beta_k p_k$
  cublasDaxpy(handle, n, &one, r, 1, p, 1);                                  // $p_{k+1} = r_{k+1} + \beta_k p_k$
  k += k;
}
```

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 25 – CUDA libraries

Lecture 25.2 – cuSOLVER

Objective

- To learn how to utilize the cuSOLVER library
  - Learn about direct factorization methods
  - Learn how to initialize and terminate the cuSOLVER environment
  - Learn about available factorization methods in cuSOLVER
  - Learn about eigenvalue and related problems in cuSOLVER

cuSOLVER

- cuSOLVER is linear algebra library with LAPACK-like features focused on:
  - Direct factorization methods for dense matrices
  - Linear systems solving methods for dense and sparse matrices
  - Eigenvalue problems for dense and sparse matrices
  - Refactorization techniques for sparse matrices
  - Least square problems for sparse matrices
- In cases where the sparsity pattern produces low GPU utilization cuSOLVER provides CPU routines to handle those sparse matrices

cuSOLVER dense API I

- cuSOLVER dense matrices routines are available under the cuSolverDN API
- cuSolverDN provides two different APIs:
  - Legacy naming convention:
    - cusolverDn<T><operation>
  - Generic naming convention:
    - cusolverDn<operation>

- <T> is the data type used by the matrices:

<T>	Type
S	float
D	double
C	cuComplex
Z	cuDoubleComplex
X	Generic type

- <operation> is the operation to be performed, some examples are:
  - portf for Cholesky factorization
  - gesvd for the SVD decomposition
- Example:
  - cusolverDnDportf calls the routine performing the Cholesky decomposition for matrices of type double

cuSolverDN dense API II

- cuSolverDN assumes that matrices are stored in column-major format
- The cuSolverDN generic API provides 64bit support for integer parameters
- The cuSolverDN generic API uses the type cudaDataType in the routines arguments to specify the type of the input matrix

cudaDataType	Type
CUDA_R_32F	Single precision real number
CUDA_R_64F	Double precision real number
CUDA_C_32F	Single precision complex number
CUDA_C_64F	Double precision complex number

cuSOLVER dense environment

- cuSolverDN needs an execution context to store internal resources. This context needs to be created before executing any cuSolverDN routine

- After all cuSolverDN executions are finished the context needs to be destroyed to free resources

- To create a context in a specific device call cudaSetDevice before the creation

- cusolverDnHandle_t
  - Type used by cuSolverDN to store contexts handles

- cusolverDnCreate(cusolverDnHandle_t* handle)
  - Creates a cuSolverDN context
  - Parameters:
    - Pointer to cuSolverDN handle to create

- cusolverDnDestroy(cusolverDnHandle_t handle)
  - Destroys a cuSolverDN context
  - Parameters:
    - cuSolverDN handle with the context to destroy

- cusolverStatus_t
  - Type used by cuSOLVER for reporting errors

cuSOLVER dense environment

- cuSolverDN routines don’t allocate workspace memory by themselves, the user needs to allocate the device workspace
- To find the size in bytes needed by a cuSolverDN routine, the user must call:
  - <routine>_bufferSize()
    - The arguments of the function varies according to the routine
    - <routine> is the name of the cuSolverDN routine
- Once the user allocates the workspace region the user can call the routine
- cuSolverDN routines accept an info parameter, if info is less than 0 this tells the user that the i-th parameter (not counting the handle) is invalid

cuSolverDN streams API and thread safety

- cuSolverDN permits the use of cuda streams (cudaStream_t) for increasing resource usage and introduce other levels of parallelism

- cuSolverDN is a thread safe library, meaning that the cuSolverDN host functions can be called from multiple threads safely

- cusolverDnSetStream():
  - Sets the stream to be used by cuSolverDN for subsequent computations
  - Parameters:
    - cuSolverDN handle to set the stream
    - cuda stream to use
- cusolverDnGetStream():
  - Gets the stream being used by cuSolverDN
  - Parameters:
    - cuSolverDN handle to get the stream
    - pointer to cuda stream

cuSolverDN LU factorization with generic API

– Given a matrix A, the LU factorization is given by:

$$P * A = L * U$$

where P is a permutation matrix produced by the algorithm with the row pivots. L is a lower triangular matrix with unit diagonal and U is an upper triangular matrix.

– The matrix A is assumed to be in column-major order

– If info=i and is positive, it means the factorization failed and U(i,i) = 0

```
cusolverDnHandle_t handle;    // cuSolverDN handle
cusolverDnParams_t params; // Routine options
void* A;                      // Matrix to factorize
int64_t m, n;                 // Rows and columns respectively
int64_t* ipiv;                // Row pivots, vector of m elements
size_t workSize;              // Workspace size in bytes
void* buffer;                 // Workspace buffer
cudaDataType typeA;           // Type of matrix A
int info = 0;                 // Info parameter

// Create and initialize options struct:
cusolverDnCreateParams(&params);
cusolverDnSetAdvOptions(params, CUSOLVERDN_GETRF,
CUSOLVER_ALG_0);
// Get the buffer size and allocate it:
cusolverDnGetrf_bufferSize(handle, params, m, n, typeA, A, n,
typeA, &workSize);
cudaMalloc((void**) &buffer, workSize);
// Compute the factorization
cusolverDnGetrf(handle, params, m, n, typeA, A, n, ipiv, typeA,
buffer, workSize, &info);
// Destroy the options struct
cusolverDnDestroyParams(params);
```

cuSolverDN Cholesky factorization with legacy API

– Given a matrix A, the Cholesky factorization is given by either of:

$$A = L * L^H$$

$$A = U^H * U$$

Where L is a lower triangular matrix and U is an upper triangular matrix.

– cuSolverDN uses a fill mode parameter to determine which decomposition to compute

– If info=i and is positive, it means the factorization failed and U(i,i) = 0

```
cusolverDnHandle_t handle;    // cuSolverDN handle
double* A;                    // Matrix to factorize
cublasFillMode_t uplo;        // Type indicating filling mode
int n;                        // Number of rows and columns
int workSize;                 // Workspace size in bytes
void* buffer;                 // Workspace buffer
int info = 0;                 // Info parameter

// Get the buffer size and allocate it:
cusolverDnSpotrf_bufferSize(handle, uplo, n, A, n,
&workSize);
cudaMalloc((void**) &buffer, workSize);
// Compute the factorization
cusolverDnDpotrf(handle, uplo n, A, n, buffer, workSize,
&info);
```

List of some available routines in cuSolverDN

- Factorization:
  - potrf, potrs for Cholesky factorization and linear system solving, see potrfBatched and potrsBatched for batched versions
  - getrf, getrs for LU factorization and linear system solving
  - geqrf for QR factorization
  - sytrf for LDL factorization
- Eigenvalue problems:
  - gesvd for SVD decomposition
  - syevd for Eigenvalue decomposition
  - sygvd for general Eigenvalue decomposition

Compiling and linking against cuSOLVER

- In order to compile and link against cuSOLVER, the user needs to:
  - Include the appropriate header in the required files
    - #include <cusolverDn.h> for cuSOLVER dense functionality
    - #include <cusolverSp.h> for cuSOLVER sparse functionality
    - #include <cusolverRf.h> for cuSOLVER refactorization functionality
  - Link against the cuSOLVER library:
    - For dynamic linking use the flag -lcusolver
    - For static linking use the flags -lcusolver_static -llapack_static

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 25 – CUDA libraries

Lecture 25.3 – cuFFT

Objective

- To learn how to utilize the cuFFT library
  - Learn about the Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT)
  - Learn about the data layouts used by cuFFT
  - Learn how to interact with the cuFFT environment
  - Learn about available cuFFT plans

Discrete Fourier Transform (DFT)

– Widely used in signal processing from astronomical radio signals to image processing and many other areas

– The forward Discrete Fourier Transform is given by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i\omega kn}$$

where $(x_0, \cdots, x_{N-1})$ is the input signal, $(X_0, \cdots, X_{N-1})$ the transformed signal, $\omega = \frac{2\pi}{N}$ and i the imaginary unit

– Given the transformed signal it’s possible to recover the input signal, this process is known as the inverse transform

Fast Fourier Transform (FFT) & cuFFT

- A fast method to compute the DFT of a signal, with computational complexity of $O(N \log_2 N)$ whereas the DFT has $O(N^2)$ complexity
- A cornerstone of numeric algorithms for its many applications and speed
- cuFFT is CUDA implementation of several FFT algorithms
- cuFFT is especially fast for input sizes of $2^a 3^b 5^c 7^d$ elements
- cuFFT provides 1D, 2D and 3D transforms, as well as:
  - C2C complex input to complex output
  - R2C real input to complex output
  - C2R complex input to real output
- cuFFT provides Multi-GPU support trough the cuFFT Xt API

cuFFT plans & environment

- Plans contain necessary information to compute a transform, such as:
  - The best algorithm to use for the specified size
  - Memory resources needed by cuFFT to compute the transform
  - Plans need to be created before executing the transform and destroyed after the user is done using them

- There exists 2 mains way to create plans:
  - Through the basic plan API:
    - cufftPlan1D, cufftPlan2D, cufftPlan3D, cufftPlanMany
    - Aimed for easy setup
  - Through the extensible plan API:
    - cufftCreate, cufftMakePlan1D, cufftMakePlan2D, cufftMakePlan3D, cufftMakePlanMany, cufftMakePlanMany64
    - Aimed for customization and extensibility

cuFFT numeric data types

- cuFFT offers the following numeric types:
  - cufftReal for single precision real numbers
  - cufftDouble for double precision real numbers
  - cufftComplex for single precision complex numbers
  - cufftDoubleComplex for double precision complex numbers
- For simplicity, the rest of the slides will use real and complex for referring to cufftReal/cufftDouble and cufftComplex/cufftDoubleComplex respectively
- For memory size considerations is important to note that:

$$sizeof(complex) = 2 * sizeof(real)$$

Data layout for 1D out-of-place transforms

- Out of place transforms have different memory regions for the input and output signal
- Input and output sizes for each transform:

Type	Input size in bytes	Output size in bytes
C2C	$N * sizeof(complex)$	$N * sizeof(complex)$
C2R	$\left(\lfloor N/2 \rfloor + 1\right) * sizeof(complex)$	$N * sizeof(\mathrm{real})$
R2C	$N * sizeof(\mathrm{real})$	$\left(\lfloor N/2 \rfloor + 1\right) * sizeof(complex)$

- Custom strides in the layout causes cuFFT to overwrite the input array in the C2R transform
- The size of the output array in R2C consists of $\lfloor N/2 \rfloor + 1$ complex numbers due to "Hermitan" redundancy property of the transform

Data layout for 1D in-place transforms

- In place transforms stores the output result in the input array
- Array sizes for each transform:

Type	Array size in bytes
C2C	$N * sizeof(complex)$
C2R	$\left(\lfloor {}^{N}/_{2} \rfloor + 1\right) * sizeof(complex)$
R2C	$\left(\lfloor {}^{N}/_{2} \rfloor + 1\right) * sizeof(complex)$

- The sizes of C2R and R2C are due to the transform needs to hold at most $(\lfloor {}^{N}/_{2} \rfloor + 1)$ complex numbers
- In R2C only the first N **real** entries are filled with input data, the rest of the array is allocated for the output

cuFFT extensible API

- Provided for customization and extensibility compared to the basic API

- Typical usage of the extensible API:
  - Initialize cufftHandle
  - Make a plan
  - Compute transforms
  - Destroy the plan in cufftHandle

- Every function returns an error status with type cufftResult

- cufftHandle
  - Type used by cuFFT to store plans

- cufftCreate(cufftHandle* handle)
  - Creates a cuFFT handle
  - Parameters:
    - Pointer to cuFFT handle to create

- cufftDestroy(cufftHandle handle)
  - Destroys resources of a cuFFT plan
  - Parameters:
    - cuFFT handle with the context to destroy

- cufftResult
  - Type used by cuFFT for reporting errors
  - Every cuFFT returns an error status

cuFFT workspace memory

- cuFFT by default auto allocates on plan creation the workspace memory needed for computations
- cuFFT allows the user to provide the memory region to be used for the workspace
- To disable auto allocation, the user needs to call cufftSetAutoAllocation() after cufftCreate() and before cufftMakeplan*()

- cufftSetAutoAllocation():
  - Sets if cuFFT should use auto workspace allocation
  - Parameters:
    - cufftHandle plan handle
    - integer indicating whether to allocate workspace area, 0:false & 1:true

cuFFT workspace memory

- If auto allocation has been disabled, the user needs to:
  - Get the size of the workspace area needed by cuFFT
  - Allocate device memory with the size specified by cuFFT
  - Set the memory region to be used by the plan

- Allocating and setting the workspace memory needs to be done after plan creation and before plan execution

- cufftGetSize():
  - Gets the size in bytes needed by the cuFFT plan
  - Parameters:
    - cufftHandle plan handle
    - pointer to size_t variable to store the size
- cufftSetWorkArea():
  - Sets the memory region to be used by the cuFFT plan
  - Parameters:
    - cufftHandle plan handle
    - Pointer to device memory region

Creating cuFFT plans

- cuFFT allows 1, 2 and 3 dimensional transforms
- Available plaining routines:
  - cufftMakePlan1d: for 1d transforms
  - cufftMakePlan2d: for 2d transforms
  - cufftMakePlan3d: for 3d transforms
  - cufftMakePlanMany: for 1, 2 and 3D transforms with support for strided input and output memory layouts
  - cufftMakePlanMany64: same as cufftMakePlanMany but with 64-bit support for sizes and strides

- cufftMakePlan1d():
  - Creates a plan for a 1D transform
  - Parameters:
    - cufftHandle plan handle
    - Size of the transform
    - cufftType type of the transform
    - Number of transforms to perform in batch
    - size_t pointer to the size in bytes of the workspace area

- cufftType types:
  - CUFFT_R2C, CUFFT_C2R, CUFFT_C2C for single precision transforms
  - CUFFT_D2Z, CUFFT_Z2D, CUFFT_Z2Z for double precision transforms

cuFFT plan and transform execution

- After plan creation the user can execute the transform
- The execution function depends on the type of the transform:
  - cufftExecC2C and cufftExecZ2Z for complex to complex transforms
  - cufftExecR2C and cufftExecD2Z for real to complex transforms
  - cufftExecC2R and cufftExecZ2D for complex to compreallex transforms

- cufftExec<t>():
  - Executes the transform specified by the plan:
  - Parameters:
    - cufftHandle plan handle
    - pointer to input array
    - pointer to output array
    - (*) Transform direction:
      - CUFFT_FORWARD for forward transform
      - CUFFT_INVERSE for the inverse transform
      - (*) This argument only exists when <t> is either C2C or Z2Z
  - If the input and output pointers are the same cuFFT will perform an in-place transform

cuFFT streams API and thread safety

- cuFFT permits the use of cuda streams (cudaStream_t) for increasing resource usage and introduce other levels of parallelism

- cuFFT is a thread safe library if functions host threads execute computations on different plans and output memory regions

- cufftSetStream():
  - Sets the stream to be used by cuFFT for subsequent computations
  - Parameters:
    - cuFFT handle to set the stream
    - cuda stream to use

Example: Cross correlation of two signals

```
cufftComplex *x, *y, *z;
// Allocate and initialize the signals
cufftHandle plan;
cufftCreate(&plan);                              // Create the handle
cufftMakePlan1d(plan, n, CUFFT_C2C , 1);         // Create the plan
cufftExecC2C(plan, x, x, CUFFT_FORWARD);         // Compute the forward x transform
cufftExecC2C(plan, y, y, CUFFT_FORWARD);         // Compute the forward y transform
// Launch a kernel executing z[i]=x[i] * y[i]    // Compute the cross correlation
cufftExecC2C(plan, z, z, CUFFT_INVERSE);
cufftDestroy(&plan);                             // Destroy the plan
```

Compiling and linking against cuFFT

- In order to compile and link against cuFFT, the user needs to:
  - Include the header “#include <cufft.h>” in the appropriate files
  - Link against the cuFFT library:
    - For dynamic linking use the flag -lcufft
    - For static linking and a version of cuda 9.0 or later use the flags -lcufft_static -lculibos

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 25 – CUDA libraries

Lecture 25.4 – Thrust

Objective

- To learn how to utilize the Thrust library
  - Learn about the C++ STL library and its relationship with Thrust
  - Learn about Thrust vector objects
  - Learn about common Thrust functions
  - Learn about Thrust functional programming capabilities

C++ STL

- C++ Standard Template Library provides a set of type agnostic features as a way of simplifying C++ programming, some examples of these features are:
  - Type agnostic data structures: lists, map, vectors, etc.
  - Type agnostic algorithms: sort, fill, max, etc.
  - Iterators for data structures
- C++ STL features reside inside the C++ namespace std

- Example:

```
#include <algorithm>
#include <vector>
…
// Creates int vector with 10 elements
std::vector<int> a(10);

// Fill vector with 10 9 8 … 1
for(int i = 0 ; i < a.size(); ++i)
  a[i] = a.size() - i;

// Sort the vector
std::sort(a.begin(), a.end());
// a vector now contains 1 2 3 … 10
```

Thrust

- Thrust is a C++ template library written for CUDA devices and based on the C++ STL with the intention of simplifying certain aspects of the CUDA programming model
- Thrust provides users with three main functionalities:
  - The host and device vector containers
  - A collection of parallel primitives such as, sort, reduce and transformations
  - Fancy iterators
- All Thrust functions and containers reside inside the thrust namespace
- Thrust allows either host or device execution
- Since thrust is a template library it doesn't need to be linked against a library

Thrust vector containers

– Thrust provides 2 vector containers:
  – thrust::host_vector<T> for a vector stored in host memory
  – thrust::device_vector<T> for a vector stored in device memory
– Containers features:
  – Interoperability between the host_vector<T> and device_vector<T>, specifically:
    – Memory transfers using the assign operator and constructors
  – Interoperability with STL containers through iterators
  – API similar to the STL std::vector
  – Dynamic resizing of the container
– To use them include:
  – #<include> “thrust/host_vector.h”
  – #<include> “thrust/device_vector.h”

STL and Thrust Iterators

- Iterators provide a high-level abstraction for data access to containers, specifically:
  - Contains information of a specific element in the container
  - Contains information on how to access other elements in the container
- There are several kinds of iterators for example:
  - Random access iterators, allows access to any element in the container
  - Bidirectional iterators, allows access to the previous and next element
  - Forward iterators, allows access to the next element

– Example:

```
thrust::host_vector<int> a(10);
// Fill a with 0 1 2 … 9
auto it = a.begin(); // Iterator pointing to the first element
int tmp = *it; // Dereference the iterator, tmp holds 0
it = a.begin() + 5; // Iterator pointing to a[5]
++it; // Iterator pointing to a[6]
*it = tmp; // Equivalent to performing  a[6] = tmp;
for( it = a.begin(); it != a.end(); ++it)
  std::cout << *it << std::endl; // Print a contents
```

*** It's important to observe that a.end() doesn't point to a[9], it refers to a position past the final element

Example: Thrust vector containers interoperability

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <list>
…
// Declare and initialize STL list container
std::list<float> stl_list;
stl_list.push_back(3.14);
stl_list.push_back(2.71);
stl_list.push_back(0.);
// Initialize thrust device vector with the STL list containing {3.14, 2.71, 0.}
thrust::device_vector<float> vector_D(stl_list.begin(), stl_list.end());
// Perform computations on vector_D
// Create host vector and copy back results to host memory
thrust::host_vector<float> vector_H = vector_D;
```

Thrust pointers and memory

- Thrust provides a pointer interface thrust::device_ptr<T> to be used when data was allocated using cudaMalloc or similar mechanisms
- The thrust::device_ptr<T> interface is compatible with all Thrust algorithms and allows similar semantics as iterators
- To use Thrust device pointer the user needs to include:
  - #include <thrust/device_ptr.h>

– Example:

```
#include <thrust/device_ptr.h>
…
float *a;
// Allocate device memory
cudaMalloc((void**)&a, 1024 * sizeof(float));
// Initialize thrust pointer with a
thrust::device_ptr<float> a_thrust(a);
// Assign to a_thrust 0, 0 + 2, 2 + 2, …, 2046
thrust::sequence(a_thrust, a_thrust + 1024, 0, 2);
// Extract the pointer being used by a_thrust
float *b = thrust::raw_pointer_cast(a_thrust);
```

Thrust algorithms

- Provides common parallel algorithms like:
  - Reductions
  - Sorting
  - Reorderings
  - Prefix-sums
  - Transformations
- All Thrust algorithms have host and device implementations
- If an algorithm is invoked with an iterator, thrust will execute the operation in the host or device according to the iterator memory region
- Except for thrust::copy which can copy data between host and device all other routines must have all its arguments reside in the same place

Thrust transformations

- Transformations apply a function to each element in the input range and stores the result in the output range
- Exists a diverse set of available transformations such as:
  - thrust::transform applies out-of-place a function to each element in a data range
  - thrust::transform_if applies out-of-place a function if a condition is met
  - thrust::for_each applies in-place a unary function
- To use Thrust transormation algorithms the user needs to include:
  - #include <thrust/transform.h>

```
#include <thrust/transform.h>
…
struct saxpy_functional {
  float alpha;
  float operator()(float &x, float &y) {
    return alpha * x + y;
  }
};
…
thrust::device_vector<float> a(1024);
thrust::device_vector<float> b(1024);
thrust::device_vector<float> c(1024);
// Initialize a and b
saxpy_functional saxpy;
saxpy.alpha = 2;

thrust::transform(a.begin(), a.end(), b.begin(),
c.begin(), saxpy); // Perform c[i] = 2 * a[i] + b[i]
```

Thrust function objects

- Thrust provides a collection of predefined operators usable by Thrust algorithms like transform and reduce
- Available operator categories are:
  - [Arithmetic operators]
  - [Comparison operators]
  - [Logical operators]
  - [Bitwise operators]
  - [Generalized identity operators]

– Example:

```
#include <thrust/functional.h>
#include <thrust/transform_reduce.h>
…
thrust::device_vector<double> x;
double zero = 0;
double norm;
// Initialize data
// Compute x norm using the thrust::square (x * x) and
        thrust::plus (x + y) operators
norm = std::sqrt(thrust::transform_reduce(x.begin(),
        x.end(), thrust::square <double>(), zero,
        thrust::plus<double>()));
```

Thrust sorting routines

- Thrust provides sorting algorithms for GPU namely:
  - thrust::sort for sorting an array
  - thrust::sort_by_key for sorting an array by a key array
- Stable sorting routines are also available:
  - thrust::stable_sort
  - thrust::stable_sort_by_key
- To use Thrust sorting algorithms the user needs to include
  - #include <thrust/sort.h>

– Example:

```
#include <thrust/sort.h>
…
thrust::device_vector<int> keys(4);
thrust::device_vector<double> values(4);
// Initialize data with:
//      keys = {3, 1, 2, 7}
//      values = {1., 2., 3., 4. }
// Launch thrust sorting by key algorithm
thrust::sort_by_key(keys.begin(), keys.end(),
values.begin());
// Values after sorting:
//      keys = {1, 2, 3, 7}
//      values = {2., 3., 1., 4. }
```

Thrust fancy iterators

- Thrust provides a collection of special iterators to be used by Thrust algorithms enhancing language programmability:
  - thrust::constant_iterator<T> is an iterator with constant value
  - thrust::counting_iterator<T> is an iterator addressing a range of numbers
  - thrust::zip_iterator is an iterator zipping two memory regions together into a single object of pairs

- Example:

```
#include <thrust/iterator/zip_iterator.h>

…

thrust::device_vector<int> A;

thrust::device_vector<float> B;

…

auto begin = thrust::make_zip_iterator(thrust::make_tuple(A.begin(), B.begin()));

auto end  = thrust::make_zip_iterator(thrust::make_tuple(A.end(), B.end()));

thrust::maximum< thrust::tuple<int,float>> max_op;

thrust::reduce(begin, lendast, init, max _op);
```

Thrust execution policies and streams

- Execution policies gives users control over certain runtime execution decisions
- Common execution policies are:
  - thrust::seq for sequential execution
  - thrust::omp::par for parallel execution using the OpenMP backend
  - thrust::cuda::par for CUDA execution
  - thrust::host for host execution
  - thrust::device for device execution

- Thrust allows the use of CUDA streams through the execution policy:
  - thrust::cuda::par.on(stream):
    - Sets the stream to use by the Thrust algorithm
    - Parameters:
      - Stream to use
- To use this policy the user needs to include:
  - #include <thrust/system/cuda/execution_policy.h>
- Example:

```
cudaStream_t stream;
…
thrust::sort(thrust::cuda::par.on(stream),
dev_vector.begin(), dev_vector.end())
```

GPU Teaching Kit

Accelerated Computing