Technical University of Munich

Lecture 9: Algorithm building blocks and CUDA Libraries

Informatik elective: GPU Computing

Pratik Nayak

In this session

- Basic data structures, C++ standard library equivalents for the GPU → thrust
- BLAS operations → cuBLAS
- Sparse functionality → cuSPARSE
- Tensors → cuTENSOR
- Solvers → cuSOLVER

C++ STL on GPUs

- The C++ standard template library (STL) contains many useful data structures, containers, and primitive operations on these containers which can be used to develop more complex algorithms.
- Thrust (https://nvidia.github.io/cccl/thrust) is the C++ template library for CUDA. It is based on STL.
- It provides similar containers and primitive operations as STL, with only slight changes in interface, adapting for CUDA GPUs

C++ STL on GPUs

- Similar to `std::vector`, you have `thrust::host_vector` and `thrust::device_vector`
  - Dynamically resizable generic templated containers.
- Operations such as `thrust::fill`, `thrust::copy`, `thrust::sequence` are available.
- Copy data from host to device and back with overloaded “=” operator.
  - `dev_vec = host_vec // Copies data from host to device with cudaMemcpy`
- Generic operations with lambdas using `thrust::transform`, in `<thrust/functional>`.
- For example, a simple axpy looks like

Axpy with thrust

```
struct saxpy_functor{
      const float a;
      saxpy_functor(float _a) : a(_a) {}
      __host__ __device__ float operator()(const float& x, const float& y) const {
                  return a * x + y;
      }
};
void saxpy_fast(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y){
      // Y <- A * X + Y
      thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A));
}
```

Reduction

- Reduction operations are also available: `thrust::reduce`, `thrust::count`, `thrust::count_if` etc.
- Sum of elements in a vector: `auto sum = thrust::reduce(vec.begin(), vec.end())`
- You can better performance and more idiomatic C++ code by fusing kernels with `thrust::transform_reduce`
  - Better optimization from runtime and more favorable cache behaviour.

L2-norm using thrust

```
template<typename T>
struct square{
      __host__ __device__ T operator()(const T& x) const {
                        return x * x;
      }
};
template<typename T>
auto norm(thrust::device_vector<T>& X){
      // return <- sqrt(sum(X * X))
      return std::sqrt(thrust::transform_reduce(X.begin(), X.end(), square<T>{}, T{0},
                        thrust::plus<T>{}));
}
```


Scans, reorderings and sorting on GPU data

- Index operations generally make use of scan operations: inclusive, exclusive scans:
  - {1, 0, 2, 2, 1, 3} → inclusive_scan → {1, 1, 3, 5, 6, 9}
- Reordering operations to select/copy elements satisfying some conditions:
  - copy_if: copy elements that satisfy a condition
  - partition: reorder elements based on a condition
  - remove, remove_if: remove elements
  - unique: remove consecutive duplicates
- Sorting operations
  - sort and stable_sort: equivalent behaviour as in STL.
  - Additionally sort_by_key enables sorting of key-value pairs

Scans, reorderings and sorting on GPU data

```
#include <thrust/sort.h>

...
const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};

thrust::sort_by_key(keys, keys + N, values);

// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

Asynchronous operations

- Thrust also has some support for asynchronous copies and operations.
- Assign operations on streams: `thrust::device.on(stream)`
- Operations in the namespace `thrust::async`
- Capture operation in a `thrust::device_event`, and schedule operations by passing dependencies: `thrust::device.after(event)`.
- Capture results in `thrust::device_future`

Asynchronous operations

```
// Asynchronously transfer to the device.
thrust::device_vector<double> d_vec(h_vec.size());
thrust::device_event e = thrust::async::copy(h_vec.begin(), h_vec.end(),
                                             d_vec.begin());

// After the transfer completes, asynchronously compute the sum on the device.
thrust::device_future<double> f0 = thrust::async::reduce(thrust::device.after(e),
                                                         d_vec.begin(), d_vec.end(),
                                                         0.0, thrust::plus<double>());
```

BLAS: Basic Linear Algebra Subroutines (BLAS)

- Basic functions that provide standard building blocks for matrix and vector operations.
- Classified into three classes:
  - BLAS1: Level 1 BLAS: Basic vector operations
  - BLAS2: Level 2 BLAS: Matrix-vector operations
  - BLAS3: Level 3 BLAS: Matrix-matrix operations
- Documentation and available operations: https://www.netlib.org/blas/

BLAS: Basic Linear Algebra Subroutines (BLAS)

- Basic functions that provide standard building blocks for matrix and vector operations.
- Classified into three classes:
  - BLAS1: Level 1 BLAS operations
    - Copy, scale, dot product, Norm etc
  - Naming: <S: Single precision (32 bit)

    D: Double precision (64 bit)

    C: Single Complex (32 bit)

    Z: Double Complex (64 bit)>

    + operation name

    → `scopy(x, y); // vector copy`

Generate plane rotation
Generate modified plane rotation
Apply plane rotation
Apply modified plane rotation
$x \leftrightarrow y$
$x \leftarrow \alpha x$
$y \leftarrow x$
$y \leftarrow \alpha x + y$
$dot \leftarrow x^T y$
$dot \leftarrow x^T y$
$dot \leftarrow x^H y$
$dot \leftarrow \alpha + x^T y$
$nrm2 \leftarrow ||x||_2$
$asum \leftarrow ||re(x)||_1 + ||im(x)||_1$
$amax \leftarrow 1^{st} k \ni |re(x_k)| + |im(x_k)|$
$= max(|re(x_i)| + |im(x_i)|)$

BLAS: Basic Linear Algebra Subroutines (BLAS)

- BLAS2: Level 2 BLAS operations
  - Matrix vector product, rank operations
  - Naming: <S: Single precision (32 bit)

    D: Double precision (64 bit)

    C: Single Complex (32 bit)

    Z: Double Complex (64 bit)>

    + matrix type (general, symmetric, banded etc) + operation name:

```
→ sgemv(...); // matrix-vector
              // product
```

$y \leftarrow \alpha A x + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$

$y \leftarrow \alpha A x + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$

$y \leftarrow \alpha A x + \beta y$

$y \leftarrow \alpha A x + \beta y$

$y \leftarrow \alpha A x + \beta y$

$y \leftarrow \alpha A x + \beta y$

$y \leftarrow \alpha A x + \beta y$

$y \leftarrow \alpha A x + \beta y$

$x \leftarrow A x, x \leftarrow A^T x, x \leftarrow A^H x$

$x \leftarrow A x, x \leftarrow A^T x, x \leftarrow A^H x$

$x \leftarrow A x, x \leftarrow A^T x, x \leftarrow A^H x$

$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$

$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$

$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$

$A \leftarrow \alpha x y^T + A, A - m \times n$

$A \leftarrow \alpha x y^T + A, A - m \times n$

$A \leftarrow \alpha x y^H + A, A - m \times n$

$A \leftarrow \alpha x x^H + A$

$A \leftarrow \alpha x x^H + A$

$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$

$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$

$A \leftarrow \alpha x x^T + A$

$A \leftarrow \alpha x x^T + A$

$A \leftarrow \alpha x y^T + \alpha y x^T + A$

$A \leftarrow \alpha x y^T + \alpha y x^T + A$

BLAS: Basic Linear Algebra Subroutines (BLAS)

- BLAS3: Level 3 BLAS operations
  - Matrix matrix operations, rank k operations
  - <S: Single precision (32 bit)

    D: Double precision (64 bit)

    C: Single Complex (32 bit)

    Z: Double Complex (64 bit)>

$$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$$
$$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$$
$$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$$
$$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$$
$$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$$
$$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C - n \times n$$
$$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C - n \times n$$
$$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n$$
$$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$$

+ matrix type (general, symmetric, banded etc) + operation name:

```
→ sgemm(...); // matrix-matrix
              // product
```

LAPACK

- Provides routines for solving linear systems, least-squares, eigenvalue problems, SVD and various factorizations (LU, Cholesky, QR etc)
- Leverages BLAS where possible.
- Reorganized algorithms to use block-matrix operations (gemm etc) for better memory accesses and overall higher throughput.
- Vendors (NVIDIA, Intel, AMD) implement highly tuned BLAS for their hardware.
- LAPACK can use the vendor-provided, standardized interface.

cuBLAS → BLAS for NVIDIA GPUs

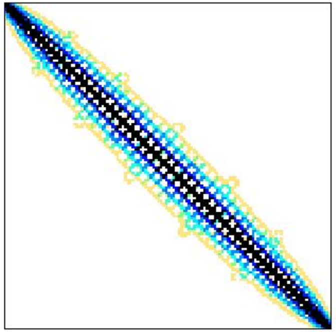
- Provides BLAS operations (Level 1, 2 and 3) for NVIDIA GPUs.
- Default storage is column-major and 1-based indexing.
    - Need to be careful when calling functions from C/C++.
- The general workflow:
    - Create a handle: A context to allow multi-threading and multi-GPU setups.
    - Setup data on the GPU in the required layout.
    - Call the cuBLAS API and assign it to a stream if necessary.
- Use the new API and not the legacy API: `#include <cublas_v2.h>`

cuBLAS → BLAS for NVIDIA GPUs

- DEMO
- https://github.com/NVIDIA/CUDALibrarySamples/tree/master/cuBLAS

Sparsity ?

- Consider a matrix of size $(m \times n)$ most of whose elements are zeros.
- Storing matrices in dense requires $\mathcal{O}(mn)$ elements.
    - Wasteful when most elements are zeros.
    - Sparsity ratio: $\phi = \dfrac{nnz}{mn}$, where nnz denotes the number of nonzeros in the matrix.
- We can do better and store only the nonzero elements.
    - Sparse formats: Specialized formats to store nonzeros and their locations.
    - Examples: COO, CSR, ELL, etc



Sparse formats: COO

- COO: Coordinate format:
    - Store the row indices, column indices and values.

$$\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{pmatrix}$$

Sparse formats: COO

- COO: Coordinate format:
  - Store the row indices, column indices and values.

Row indices: [0 3 1 1 2 0 3]

Column indices: [0 0 1 2 2 3 3]

Values: [1 2 3 4 5 6 7]

$$\begin{array}{cc} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \\ \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{pmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \end{array}$$

- Storage complexity: $\mathcal{S} = 3nnz$

Sparse formats: CSR

- CSR: Compressed sparse row
  - Store the column indices, row pointers and values.

$$\begin{array}{c c}
\begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \\
\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{pmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}
\end{array}$$

Sparse formats: CSR

- CSR: Compressed sparse row
  - Store the column indices, row pointers and values.

Row pointers: $\underbrace{[0\ 2\ 3\ 5\ 7]}$

Total number of nonzeros in matrix

Of size (n+1), with starting element 0

Column indices: [0 0 1 2 2 3 3]

Values: [1 2 3 4 5 6 7]

$$\begin{array}{cc} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \\ \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{pmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \end{array}$$

- Storage complexity: $S = 2nnz + n + 1$

Sparse formats: ELL

- ELL: ELLPack
  - Store a fixed number of nonzeros in each row

$$
\begin{array}{cc}
\begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \\
\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{pmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}
\end{array}
$$

Sparse formats: ELL

- ELL: ELLPack
  - Store a fixed number of nonzeros in each row

Num nnz per row, l: 2

Column indices: [0 1 1 0 3 0 2 3]

Column- major storage

$$
\begin{array}{cc}
\begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \\
\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{pmatrix} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}
\end{array}
$$

Values: [1 3 4 6 2 0 5 7]

Stores an explicit zero

- Storage complexity: $S = 2(n \times l)$

cuSPARSE → Sparse BLAS for NVIDIA GPUs

- Provides BLAS operations for sparse matrices on NVIDIA GPUs.
- Choosing the correct sparse format and ensuring data in that layout is user responsibility.
- The general workflow:
  - Create a handle: A context to allow multi-threading and multi-GPU setups.
  - Setup data on the GPU in the required format and layout.
  - Call the cuSPARSE API and assign it to a stream if necessary.
- `#include <cusparse.h>`

cuSPARSE → Sparse BLAS for NVIDIA GPUs

- DEMO
- https://github.com/NVIDIA/CUDALibrarySamples/tree/master/cuSPARSE

Tensors

- Higher-order arrays. Generalizes the matrix concept in higher dimensions.
  - Scalar → order 0 tensor
  - Vector → order 1 tensor
  - Matrix → order 2 tensor etc.
- An order-n tensor has n ***modes***. Each mode has an ***extent*** (size in that dimension) and a ***stride***.
- Einstein notation: $y = \sum_i x^i z_i$ is represented as $x^i z_i$. With repeated indices, the summation is assumed to be implicit.
- Example of a tensor operation: $C_{a,b,c} = A_{a,k,c} B_{k,b}$

cuTENSOR → Tensor operations on NVIDIA GPUs

- Provides tensor operations for NVIDIA GPUs: Tensor contraction, reduction, and element-wise operations
- Mixed-precision support to utilize tensor cores.
- The general workflow:
  - Create a handle: A context to allow multi-threading and multi-GPU setups.
  - Setup data on the GPU in the required format and layout: More involved than matrix/vectors
  - Setup "Plan" cache to enable efficient memory usage to avoid re-allocations.
  - Call the cuTENSOR API and assign it to a stream if necessary.
- `#include <cutensor.h>`

cuTENSOR → Tensor operations on NVIDIA GPUs

- DEMO
- https://github.com/NVIDIA/CUDALibrarySamples/tree/master/cuTENSOR

Computational physics

Equation describing physics

Non-linear discretization

Linearization of the non-linear iteration

Each step requires a solution of a coupled linear system of the form

$$\mathbf{AX} = \mathbf{B}$$

$$\phi'(t) = \underbrace{R(t, \phi(t))}_{\text{Reaction term}} + \underbrace{F(t, \phi(t))}_{\text{Forcing term}}$$

$$\begin{bmatrix} A_{11} & \cdots & \cdots & A_{1K} \\ \vdots & A_{22} & & \vdots \\ \vdots & & \ddots & \\ A_{K1} & \cdots & & A_{KK} \end{bmatrix} \begin{bmatrix} \Phi_1 \\ \Phi_2 \\ \vdots \\ \Phi_K \end{bmatrix} = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_K \end{bmatrix}$$

Machine learning and optimization

- Principal Component Analysis (PCA): Identify the main components in high-dimensional data $\rightarrow$ Uses an eigenvalue solver.
  - Singular value decomposition (SVD): $M = U\Sigma V^*$
  - Also used for data compression, noise identification etc.
- Regression/fitting/supervised machine learning: $min_{x \in \mathbb{R}^n} ||Ax - b||^2$

cuSolver: LAPACK on GPUs

- Using cuBLAS and cuSPARSE, cuSOLVER provides LAPACK type routines for dense and sparse data structures on GPUs.
- cuSolverDN: Dense LAPACK: Factorization, eigenvalue solvers etc.
- cuSolverSP: Sparse LAPACK: Factorizations and eigenvalue solvers for sparse matrices stored in CSR format.
- DEMO (https://github.com/NVIDIA/CUDALibrarySamples/tree/master/cuSOLVER)

Summary

- cuBLAS: BLAS operations for dense matrices
- cuSPARSE: BLAS operations for Sparse matrices
- cuTENSOR: Tensor operations
- cuSOLVER: Solvers and LAPACK routines for NVIDIA GPUs

Next lecture

- Distributed computing basics.
- Distributed programming models: MPI, OpenSHMEM, NVSHMEM
- Multi-GPU programming with CUDA.