CS 179: Lecture 10

Introduction to cuBLAS

Table of contents, you are here.

- Welcome to week 4, this is new material from here on out so please ask questions and help the TAs to improve the lectures and course material.
- We will be discussing how to accelerate many common Linear Algebra operations for use in lab4 through lab6.

Shared Memory note

- When allocating shared memory dynamically by passing in the third parameter to the kernel call mind how much you are allocating
- <<<blockNum, threadsPerBlock, sharedMemSize>>>
- If you allocate way too much CUDA won't properly error out on the kernel call and your kernel will not execute any instructions inside it.
- cuda-GDB will continue stepping correctly but not even a printf will work.

Goals for this week

- How do we use cuBLAS to accelerate linear algebra computations with already optimized implementations of the Basic Linear Algebra Subroutines (BLAS).
- How can we use cuBLAS to perform multiple computations in parallel.
- Learn about the cuBLAS API and why it sucks to read.
- Learn to use cuBLAS to write optimized cuda kernels for graphics, which we will also use later for machine learning.

What is BLAS?

- https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms
- BLAS defines a set of common functions we would want to apply to scalars, vectors, and matrices.
- Libraries that implement it exist in almost all major languages.
- The names of these functions are opaque and hard to follow, so keeping an index nearby is useful. There are different functions for different number types

Some cuBLAS functions

- cublasIsamax() : Is - a- max. finds the smallest (first) index in a vector that is a maximum for that vector
- cublasSgemm() : generalized matrix matrix multiplication with single precision floats. Also how you do Vector Vector multiplication.
- cublasDtrmm() : triangular matrix matrix multiplication with double precision floats. See what I mean?

How to Follow This Lecture

- Some familiarity with linear algebra is necessary, but you don't need to know that much theory.
- The symbols used throughout these slides will be consistent to the following:
  - Scalars: α, β
  - Vectors: χ, γ
  - Matrices: **A**, **B**, **C**
- BLAS (Basic Linear Algebra Subprograms) was written for FORTRAN and cuBLAS follows its conventions.
  - Matrices are indexed column major
    - The relevant dimension is the number of columns
  - Arrays start at 1 (or 0, depending on if it was written for FORTRAN or C)
  - We will use a macro to handle our matrix indexing as our cuBLAS arrays will be one dimensional

How to Follow This Lecture

- There are 3 “levels” of functionality
  - Level 1: Scalar and Vector, Vector and Vector operations, $\gamma \rightarrow \alpha\chi + \gamma$
  - Level 2: Vector and Matrix operations, $\gamma \rightarrow \alpha\mathbf{A}\chi + \beta\gamma$
  - Level 3: Matrix and Matrix operations, $\mathbf{C} \rightarrow \alpha\mathbf{AB} + \beta\mathbf{C}$
- Some desired functionality like Vector Vector complex multiplication, like the kind done in lab 3, can be implemented using the generalized matrix matrix multiplication (gemm.)
  - Vectors are just 1xN matrices right?
  - No need to have even more functions for these cases.
    - Makes finding the function you want more of a pain sometimes
    - Writing a proper “templated” C++ wrapper around cuBLAS is an interesting idea.

What is cuBLAS good for?

- Anything that uses heavy linear algebra computations (on dense matrices) can benefit from GPU acceleration
  - Graphics
  - Machine learning (this will be covered next week)
  - Computer vision
  - Physical simulations
  - Finance
  - etc.....
- cuBLAS excels in situations where you want to maximize your performance by batching multiple kernels using streams.
  - Like making many small matrix-matrix multiplications on dense matrices

The various cuBLAS types

- All of the functions defined in cuBLAS have four versions which correspond to the four types of numbers in CUDA C
  - S, s : single precision (32 bit) real float
  - D, d : double precision (64 bit) real float
  - C, c : single precision (32 bit) complex float (implemented as a float2)
  - Z, z : double precision (64 bit) complex float
  - H, h : half precision (16 bit) real float

cuBLAS function types

- cublasIsamax -> cublas I s amax
  - cublas : the cuBLAS prefix since the library doesn't implement a namespaced API
  - I : stands for index. Cuda naming is dumb. Sorry.
  - s : this is the single precision float variant of the isamax operation
  - amax : finds a maximum
- cublasSgemm → cublas S gemm
  - cublas : the prefix
  - S : single precision real float
  - gemm : general matrix-matrix multiplication
- cublasHgemm
  - Same as before except half precision
- cublasDgemv → cublas D gemv
  - D : double precision real float
  - gemv : general matrix vector multiplication

How to actually use cuBLAS

- https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf
  - This pdf contains many various cuBLAS examples in different implementation. Referencing this as well as the official NVIDIA docs are highly recommended.
- http://docs.nvidia.com/cuda/cublas/index.html the official NVIDIA docs.
  - As usual the actual NVIDIA documentation is terrible, but it is the canonical index of all the various functions and types as well as what they “do”, the explanations leave something to be desired.
- Include the header “cublas_v2.h” and link the library with “-lcublas”, this is done in the Makefile given.
- Cublas uses handles just like cuFFT in lab 3 and most of the current CUDA libraries.

Numpy vs math vs cuBLAS

numpy	math	cuBLAS (<T> is one of S, D, C, Z)
numpy.dot(α, χ)	$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}\mathbf{b}^T$	cublas<T>dot(α, χ)
numpy.dot(χ, γ)	$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}\mathbf{b}^T$	cublas<T>dot(χ, γ)
numpy.dot(χ, **A**)		cublas<T>dot(χ, **A**)
numpy.dot(**A, B**)		cublas<T>dot(**A, B**)
numpy.dot(χ, **A**)		

Numpy vs math vs cuBLAS

numpy	math	cuBLAS (<T> is one of S, D, C, Z, H)
numpy.multiply(α, χ)	$(\lambda \mathbf{A})_{ij} = \lambda(\mathbf{A})_{ij}$	cublas<T>gemm(α, χ)
numpy.multiply(χ, γ)	$(A \circ B)_{i,j} = (A)_{i,j}(B)_{i,j}.$	cublas<T>gemm(χ, γ)
numpy.multiply(χ, **A**)	$A\chi = C$	cublas<T>gemm(χ, **A**)
numpy.multiply(**A**, **B**)	$C \leftarrow \alpha AB + \beta C$	cublas<T>gemm(**A**, **B**)

Array Indexing

- Since our arrays are linearized into one dimension we need to use an indexing macro.

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

Where “i” is the row, “j” is the column, and “ld” is the leading dimension.

In column major storage “ld” is the number of rows.

Error Checking

- Like CUDA and cuFFT, cuBLAS has a similar but slightly different status return type.
- cublasStatus_t
- We will use a similar macro to gpuErrchk and the cuFFT version to check for cuBLAS errors.

Streaming Parallelism

- Used to overlap computations when we need the results of several linear algebra operations to continue our program.
- Makes “more efficient” use of the GPU hardware than a naively, or even moderately optimized handwritten written kernel.
  - If you just need to multiply two vectors element wise and normalize (scale) you can use the GEMM operation.

Streaming Parallelism

- Use cudaStreamCreate() to create a stream for computation.
- Assign the stream to a cublas library routine using cublasSetStream()
- When you call the routine it will operate in parallel (asynchronously) with other streaming cublas calls to maximize parallelism.
- Should pass your constant scalars by reference to help maximize this benefit.

Streaming Gotcha

- In general you won't be able to have more than 16 kernels running simultaneously on a GPU
  - Even though we are using a library there is still a kernel being run on the GPU, you just aren't writing it yourself.