Linear Algebra Libraries and CUDA

Mark Greenstreet & Ian M. Mitchell

CPSC 418 – March 27, 2017

Once Upon a Time. . .

- Numerical calculation has been a key application since the earliest days of computing.
  - The term "computer" has been used for centuries to refer to a person performing mathematical calculations according to a fixed set of rules.
  - One of the earliest electronic general purpose computers (1946) was the Electronic Numerical Integrator and Computer (ENIAC) designed primarily to calculate artillery firing tables for the US Army.
- High level programming language and compiler development was spurred by Fortran ("formula translator") starting in the mid-1950s.
- Turing Award for 1989 went to William Kahan for his work "making the world safe for numerical computations."
  - IEEE standard for floating point arithmetic (IEEE 754) now provides a common, reproducible and robust format across virtually all computing platforms.

Linear Algebra is Everywhere

Many numerical algorithms are designed around linear algebra operations.

- By late 1960s it was common in the numerical computing community to implement these operations as separate "subprograms"
- ACM-SIGNUM project 1973-1977 set out to design what we would now call a common API to these most common routines.
- Design process and outcomes documented in a series of papers in *ACM Trans. Mathematical Software* (*ACM-TOMS*):
  - Lawson et al, "Basic linear algebra subprograms for Fortran usage," *ACM TOMS* 5(3): 308–323 (Sept. 1979).
  - Dongarra et al, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM TOMS* 14(1): 1–17 (March 1988).
  - Dongarra et al, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM TOMS* 16(1): 1–17 (March 1990).
  - Blackford et al, "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM TOMS* 28(2): 135–151 (June 2002).

Basic Linear Algebra Subprograms (BLAS)

Authors and contributors anticipated many benefits:

- Encourages “structured programming”: Modularization of common code sequences.
- Code will be more self-documenting: Other programmers will recognize the subprogram names.
- Subprograms can be coded in assembly to improve efficiency, and if the majority of computational effort is within the subprograms that will significantly benefit the whole application.
- Subprograms can be coded by experts to deal with “algorithmic and implementation subtleties.”
- Code becomes portable while still maintaining efficiency.

While the details may differ, similar benefits still accrue today.

Levels of BLAS

BLAS specification consists of operations at one of three "levels":

- BLAS-1: Vector-vector operations (scalar vector product, vector sum, dot product, etc.).
  - [Lawson et al, 1979].
  - Performs $\mathcal{O}(n)$ operations on $\mathcal{O}(n)$ data.
- BLAS-2: Matrix-vector operations (matrix-vector product, triangular solves)
  - [Dongarra et al, 1988].
  - Performs $\mathcal{O}(n^2)$ operations on $\mathcal{O}(n^2)$ data.
- BLAS-3: Matrix-matrix operations (matrix-matrix product, triangular solves with multiple right-hand sides)
  - [Dongarra et al, 1990].
  - Performs $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ data.

Types of Operands

- Provides for either "single precision" or "double precision" floating point arithmetic.
  - Support for complex variables (real + imaginary components).
  - Note that BLAS does not mandate IEEE FP standard: Definition of precision depends on the platform.
- Initial versions focused on dense or banded matrices.
  - Special cases for symmetric, Hermitian (complex version of symmetry) or triangular form.
- Extended in [Blackford et al, 2002]:
  - Sparse matrices.
  - Extended and mixed precision arithmetic.
  - A number of new routines whose importance was discovered during implementation of LAPACK:
    - ⋆ Commonly used operations, such as matrix norm.
    - ⋆ Slight generalizations of existing routines.
    - ⋆ Perform two existing routines in a single call to reduce memory traffic.
- Many other extensions / implementations have been described.

Fortran? Are You Kidding Me?

- At the time of the initial design of BLAS, Fortran was by far the dominant language of numerical computing.
  - The FORTRAN 77 standard had just been adopted
  - (the first BLAS definition was non-conforming.)
- Many limitations and idiosyncracies can be avoided, such as:
  - Only Fortran bindings.
  - ALL CAPITAL LETTERS for symbols.
  - Static allocation of arrays.
  - 1-based indexing.
- Some Fortran features remain in some implementations, such as:
  - Function names and arguments are incomprehensibly short.
  - Column-major ordering of data in matrices.
  - Arguments are pass by reference (even some scalars).

Decypering BLAS Function Names

Function names in BLAS follow a pattern.

- Often a prefix, such as `BLAS_` or `cblas_`.
- One character to denote data type; for example:
  - `s`: single precision.
  - `d`: double precision.
- Operations involving a matrix add two characters to denote matrix type; for example:
  - `ge`: general dense matrix.
  - `tb`: triangular banded.
- Short mnemonic string to denote operation; for example
  - `axpy`: *ax* plus *y*.
  - `mm`: matrix multiply.
- Put them all together:
  - `BLAS_SAXPY()`: Fortran single precision vector summation.
  - `cblas_dgemm()`: C double precision dense matrix product.

Decyphering BLAS Function Arguments (part 1)

Consider matrix product $C = \alpha A^{op} B^{op} + \beta C$ implemented by

```
cblas_sgemm(enum blas_order_type layout,
            enum blas_trans_type transa,
            enum blas_trans_type transb,
            int m, int n, int k,
            float alpha,
            float *a, int lda,
            float *b, int ldb,
            float beta,
            float *c, int ldc)
```

- `layout` specifies either column-major or row-major.
- `transa` specifies whether to use A, A^T or A^H.
  - Same for `transb` and B.
- `m`, `n`, `k` specify matrix sizes: A is $m \times k$, B is $k \times n$, C is $m \times n$.
- `alpha` and `beta` specify scalar multipliers.
  - Some implementations may require pass by reference.

Decyphering BLAS Function Arguments (part 2)

Consider matrix product $C = \alpha A^{op} B^{op} + \beta C$ implemented by

```
cblas_sgemm(enum blas_order_type layout,
            enum blas_trans_type transa,
            enum blas_trans_type transb,
            int m, int n, int k,
            float alpha,
            float *a, int lda,
            float *b, int ldb,
            float beta,
            float *c, int ldc)
```

- `a` is a pointer to array for A and `lda` is the distance between the start of consecutive columns (for column-major) or rows (for row-major).
  - Same for `b`, `ldb` and B.
  - Same for `c`, `ldc` and C.

What's with the `ld*` Arguments?

- BLAS routines allow for data which is not stored continuously.
- These `ld*` arguments are called the *stride*.
- For vectors, striding allows access to rows or columns of a matrix.
  - Consider the data in an `m` $\times$ `n` column-major matrix.
  - A column has stride `1` and length `m`.
  - A row has stride `m` and length `n`.
- For matrices, striding allows access to submatrices; for example,
  - Consider the data in an `m` $\times$ `n` column-major matrix `a`.
  - We want the `p` $\times$ `q` block starting at row `i` and column `j`.
  - Data starts at `&a[i + j*m]`
  - Data has size `p` by `q`.
  - Data has stride `m`.

CUDA and BLAS

- The cuBLAS library provides an API for running BLAS routines on CUDA GPUs.
- Basic pattern of use:
  - Initialize the cuBLAS library and allocate hardware resources using `cublasCreate()`.
  - Allocate memory using `cudaMalloc()`.
  - Copy data from host to GPU using `cublasSetVector()` or `cublasSetMatrix()`.
  - Perform BLAS operations; for example `cublasSaxpy()` or `cublasSgemm()`.
  - Copy data from GPU to host using `cublasGetVector()` or `cublasGetMatrix()`.
  - Release memory using `cudaFree()`.
  - Release hardware resources using `cublasDestroy()`.
- Example(s).

Notes on cuBLAS

- Always uses column-major ordering
  - So be careful with data layout.
- Always uses 1-based indexing.
  - Usually irrelevant since you do not index into arrays.
- All cuBLAS code is called from the host.
  - You do not write any kernel code.
  - You do not have to worry about grids, blocks, shared memory, ...
- Need to link against cuBLAS library.
  - Check that environment variable `LD_LIBRARY_PATH` includes CUDA library directory.
  - (`/cs/local/lib/pkg/cudatoolkit/lib64` on `linXX` machines.)
  - Add `-lcublas` to compile command.

Efficiency of cuBLAS

Matrix product example from 2017-03-24 lecture (in seconds):

	1024	2048	3072	4096
Brute force `mmult1`	0.079	0.648	2.190	5.152
Tiled `mmult2`†	0.027	0.208	0.724	1.690
`cublas_sgemm`	0.007	0.052	0.176	0.421

- Brute force `mmult1` achieves $\sim$ 13 GFLOPS.
- `cublas_sgemm` achieves $\sim$ 160 GFLOPS.
- (†Ian's tiled implementation `mmult2` is buggy.)

Other Numerical Libraries

- Linear Algebra PACKage (LAPACK).
  - Implements the more complex linear algebra operations.
  - Designed to call BLAS for basic computational steps.
- For your CPU:
  - Intel's Math Kernel Library (MKL) implements core functions from BLAS, LAPACK, FFTs, etc.
  - Automatically Tuned Linear Algebra Software (ATLAS) generates a BLAS library tuned to a machine's memory hierarchy.
- Many other accelerated libraries available for CUDA devices.
  - For example: cuFFT, cuSPARSE, cuRAND, cuDNN, MAGMA (supports LAPACK), ...

You are almost always better off learning to use the library.