*September 19, 2008*

# CUDA, Supercomputing for the Masses: Part 8

*Rob Farber*

## Using libraries with CUDA

*Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.*

---

In CUDA, Supercomputing for the Masses: Part 7 of this article series on CUDA (short for "Compute Unified Device Architecture"), I took a look at some next-generation CUDA hardware. In this installment, I shift gears a bit, moving from hardware to software, focusing on using libraries with CUDA.

Optimized libraries often provide an easy way to improve performance of applications. When porting large legacy projects, libraries may be the only real way to optimize for a new platform because code changes would require extensive validation efforts. Essentially libraries are convenient and can greatly accelerate code development as well as application performance, but they cannot be blindly utilized. GPU-based libraries in particular require you to think carefully about data placement and how the library is used. Otherwise poor performance will be the result.

The Basic Linear Algebra Subprograms (BLAS) package is the de facto programming interface for basic linear algebra operations such as vector and matrix multiplication. NVIDIA supports this interface with its own library for the GPU called CUBLAS. CUFFT is another NVIDIA-supported library that provides a GPU-based implementation of the Fast Fourier Transform (FFT), a commonly used method in scientific and signal processing applications. It is modeled after FFTW, a very highly-optimized and popular FFT package for general-purpose processors.

BLAS is structured according to three different "Levels":

- Examples of level-1 algorithms include taking an inner product of two vectors, or scaling a vector by a constant multiplier.
- Level-2 algorithms are matrix-vector multiplication or a single right-hand-side triangular solve.
- Level-3 algorithms include dense matrix-matrix multiplication.

If we assume a vector is length **N** or a matrix is order **N*N**, then the number of floating-point operations for a level-1, Level-2, and level-3 algorithm are **O(N)**, **O(N2)**, and **O(N3)**, respectively. (Big-O notation is a convenient way to describe how the size of an input affects an algorithm's consumption of a computational resource such as time or memory.)

The basic model by which applications use the CUBLAS library is to create matrix and vector objects in GPU memory space, fill them with data, call a sequence of CUBLAS functions, and, finally, move the results from

GPU memory space back to the host. To accomplish this, CUBLAS provides helper functions for creating and destroying objects in GPU space, and for writing data to and retrieving data from these objects. CUBLAS uses column-major storage and 1-based indexing for maximum FORTRAN compatibility. C and C++ applications need to use macros or inline functions to facilitate access to CUBLAS-created objects.

Data movement is a central consideration when using CUBLAS (and BLAS routines in general). BLAS data movement according to level is **O(N)**, **O(N2)**, and **O(N2)**, which makes the number of floating-point operations per data item moved **O(1)**, **O(1)**, and **O(N)**, respectively. This last fact indicates how important it is to locate the data on the GPU, as I describe in greater detail shortly.

Since the thread-processors of the GPU operate only on data local to the GPU, any BLAS operation on a vector or matrix located in the memory space of the host computer requires a data transfer operation. These data transfers are expensive relative to the floating-point capability of the graphics processors (GPUs can perform floating-point far faster than they can move data) and are to be avoided whenever possible or they will bottleneck performance.

To illustrate this behavior, consider the costs involved in moving a vector from the host system to the GPU, multiplying it by a constant, and then returning the result to the host. This "application" requires moving **4N** bytes of data (where **N** is the number of **float**s in the vector) to and from the GPU to perform **N** multiplications (the constant times the vector). The best possible performance this "application" could achieve is the transfer bandwidth divided by 8 (the number of bytes required to move a 32-bit **float** to and from the GPU). Such an application would be lucky to achieve a = GFLOP (billion floating-point operations per second) floating-point performance assuming a 4 GB/s transfer rate between the host and GPU. This is well below the floating-performance capability of even the lowest-cost low-end NVIDIA GPU. Batching multiple operations could potentially this performance by exploiting the full-duplex capability of the PCIe bus.

Bottom line: Getting good performance for level-1 and level-2 BLAS applications requires keeping as much data as possible on the GPU. When that is not possible, at least try to batch as many library calls as possible.

Level-3 BLAS applications can achieve extremely efficient performance. Table 1 describes a SGEMM benchmark on various architectures (e.g., an HP xw8600 with a Xeon E5440 at 2.83 GHz, a C870 and a C1060 both running CUDA 2.0). It is worth noting that benchmark performs the same sequence of operations used by computational finance applications to perform fast matrix exponentiation -- so it does reflect a real-world performance capability. All results are reported in GFLOP. "Thunking" in the CUBLAS nomenclature is an interface that automatically allocates memory, copies data to the GPU, runs the computation, copies the results back, and deallocates memory. This makes it a drop in replacement for BLAS but with obvious performance implications due to all of the data movement. The columns denoted "GPU only" are where data allocation/freeing and copies are managed by the application developer. In this case the memory is allocated once, the data is copied over, and a large sequence of SGEMM calls are made, and then the results are copied back. The resulting cost of moving data is trivially small. Included are results for several **N*N** matrix sizes on the 8- and 10-series Tesla cards and a quad-core CPU. Please notice that the CPU is faster for small matrix sizes than the thunking interface, but the GPU can perform much better if the application developer more closely manages data movement.

| N | CPU single threaded | CPU 2 cores | CPU 4 cores | C870 Thunking | C870 GPU only | C1060 Thunking | C1060 GPU only |
|---|---|---|---|---|---|---|---|
| 512 | 14.92 | 29.84 | 51.14 | 44.75 | 171.29 | 67.13 | 245.12 |
| 768 | 14.62 | 28.32 | 53.31 | 75.51 | 172.86 | 90.62 | 332.18 |
| 1024 | 14.62 | 28.08 | 54.73 | 85.92 | 173.7 | 113.04 | 317.68 |
| 2048 | 20.48 | 40.06 | 76.71 | 121.01 | 166.72 | 197.5 | 354.96 |

**Table 1**

Improvements over the current generation of stock NVIDIA libraries can be achieved. For example, in their paper LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs, Vasily Volkov and James Demmel claim to approach the peak performance of the G80 series GPUs with their matrix-matrix multiply routine. Their paper includes a detailed benchmarking study of the GPU memory system, which they used to devise their particular improvements. They claim their method is approximately 60 percent faster than the CUBLAS 1.1 implementation. These improvements have already been incorporated into the CUBLAS 2.0 release which was used to generate the results in the previous table as NVIDIA is continually working to improve both the hardware and library code.

Excellent performance can be achieved with BLAS on the GPU. For example, FORTRAN programmers should look to FLAGON, an open source library project that implements a middle-ware call layer that adds GPU "device descriptors" to help optimize performance. In their Supercomputing 2007 poster, Nail A. Gumerov, Ramani Duraiswami, and William D. Dorland claim a 25x speedup over an Intel QX6700 serial CPU code using CUFFT and some exceptional performance results compared to serial code when using an iterative solver to fit radial basis functions to scattered data.

There are also many examples in the literature where GPU libraries fill a convenience role and do not provide any performance benefits (or even a performance disadvantage). As discussed previously, the cost of data movement can be the deciding factor. For example, see CUFFT vs. FFTW for an excellent characterization.

In summary, the NVIDIA CUFFT and CUBLAS libraries provide a convenient and generally accepted interface for linear algebra and the Fast Fourier Transform. These libraries can enable large projects to use graphics processors. Excellent performance can be achieved when care is taken to minimize or eliminate data movement. Increased performance can be achieved by batching multiple library calls. Significant improvements can also be achieved by paying close attention to the CUDA-enabled device characteristics. Since NVIDIA is continually improving both the hardware and library codes, expect continued performance improvements.

## For More Information

- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8

- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Click here for more information on CUDA and here for more information on NVIDIA.

Courtesy of: www.ddj.com