HPC (HTTPS://DEVBLOGS.NVIDIA.COM/CATEGORY/HPC/)

Drop-in Acceleration of GNU Octave

By Nikolay Markovskiy (https://devblogs.nvidia.com/author/nmarkovskiy/) | June 5, 2014 (https://devblogs.nvidia.com/drop-in-acceleration-gnu-octave/)

Tags: CUBLAS (https://devblogs.nvidia.com/tag/cublas/), CUDA (https://devblogs.nvidia.com/tag/cuda/), Libraries (https://devblogs.nvidia.com/tag/libraries/), Parallel Programming (https://devblogs.nvidia.com/tag/parallel-programming/)

cuBLAS (http://docs.nvidia.com/cuda/cublas/) is an implementation of the BLAS (http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms) library that leverages the teraflops of performance provided by NVIDIA GPUs. However, cuBLAS can not be used as a direct BLAS replacement for applications originally intended to run on the CPU. In order to use the cuBLAS API:

- a CUDA context first needs to be created
- a cuBLAS handle needs to be initialized
- all relevant data needs to be copied to preallocated GPU memory, followed by deallocation after the computation

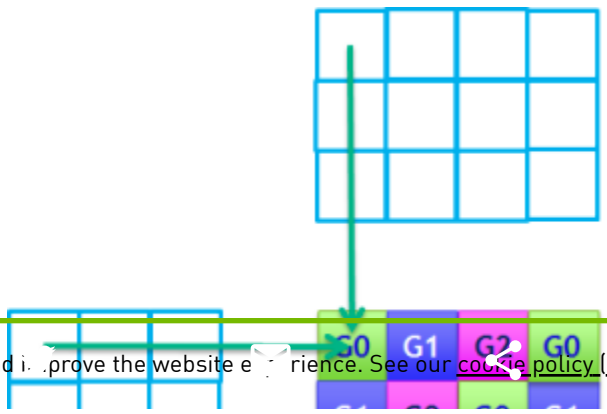
Such an API permits the fine tuning required to minimize redundant data copies to and from the GPU in arbitrarily complicated scenarios such that maximum performance is achieved. But it is less convenient when just a few BLAS routines need to be accelerated (simple data copy) or when vast amounts of code need to be modified (large programmer effort). In these cases it would be useful to have an API which managed the data transfer to and from the GPU automatically and could be used as a direct replacement for CPU BLAS libraries.

Additionally, there is the common case where the input matrices to the BLAS operations are too large to fit on the GPU. While using the cuBLAS API to write a tiled BLAS implementation (which achieves even higher performance) is straightforward, a GPU BLAS library which implemented and managed such tiling in a near optimal way would certainly facilitate access to the computing power of the GPU.

To address these issues, CUDA 6 adds new Multi-GPU extensions, implemented for the most compute intensive BLAS Level 3 routines. They are called cuBLAS-XT (https://developer.nvidia.com/cublasxt) and can work directly with host data, removing the need to manually allocate and copy data to the GPU's memory. NVBLAS is a dynamic library built on top of these extensions which offers a transparent BLAS Level 3 acceleration with zero coding effort. That is, CPU BLAS libraries can be directly replaced with NVBLAS. As such, NVBLAS can be used to easily accelerate any application which uses level-3 BLAS routines.

cuBLAS-XT and NVBLAS supports unlimited number of devices as soon as they are connected to a single shared memory system.

The figure below demonstrates how cuBLAS-XT distributes work among Multiple GPUs in (x)GEMM by splitting matrices into tiles. Such tiling allows not only overlapping PCI transfers with computations, but also permits the solution of problems that exceed the size of GPU memory available.



(https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2014/04/cublasxt-mgpu.png)

Example of cublasXt[t]gemm() tiling for 3 Gpus

To demonstrate how to use NVBLAS, let's start by accelerating matrix-matrix multiplication in GNU Octave (https://www.gnu.org/software/octave/) on the GPU and compare these results with the CPU. Our benchmark system is dual socket Ivy Bridge (dual E5-2690v2 (http://ark.intel.com/products/64596/intel-xeon-processor-e5-2690-20m-cache-2_90-ghz-8_00-gts-intel-qpi), 20 cores total) with a single K10 (http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf) board (dual GK104) with ECC (http://en.wikipedia.org/wiki/ECC_memory) ON. We choose GNU Octave, since it supports single precision which the GK104 we'll be running performs best on.

```
N = 8192;
A = single(rand(N,N));
B = single(rand(N,N));
start = clock();
C = A * B;
elapsedTime = etime(clock(), start);
disp(elapsedTime);
gFlops = 2*N*N*N/(elapsedTime * 1e+9);
disp(gFlops);
```

Running the script ' `octave ./sgemm.m` ' yields 2.5 GFLOPs

This number is far from theoretical peak of E5-2690 (16 SP FLOPs/cycle * 3.0GHz = 48 GFLOPs per core). Thus, the bundled-in BLAS version, which came with GNU Octave distribution, is not efficient.

It is a common situation for a binary distribution of a freely available package, such as NumPy, GNU Octave, R, etc, to not be linked against the highest performance BLAS implementation available. This can happen because of compatibility or licensing reasons. Efficient implementation of BLAS is highly hardware dependent and major hardware vendors ship their versions of the library, but not all of them are free. There is a CPU optimized open source solution and OpenBLAS (http://xianyi.github.com/OpenBLAS) demonstrates high performance in compute heavy Level 3 routines (https://github.com/xianyi/OpenBLAS/wiki/faq#sandybridge_perf).

A well-known trick to skip the time consuming rebuilding step is to dynamically intercept and substitute relevant library symbols with high performing analogs. On Linux systems LD_PRELOAD environment variable allows us to do exactly that.

Now we will try OpenBLAS, built with OpenMP and Advanced Vector Extensions (AVX) support. Assuming the library is in LD_LIBRARY_PATH,

' `OMP_NUM_THREADS=20 LD_PRELOAD=libopenblas.so octave ./sgemm.m` ' yields 765 GFLOPs.

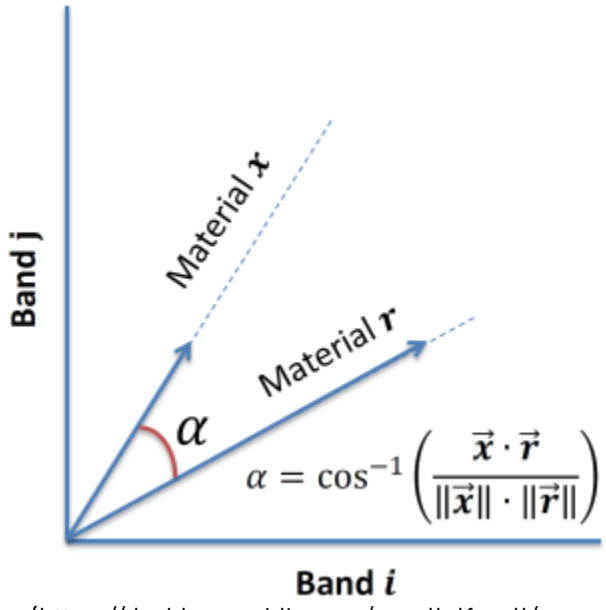
Since NVBLAS supports BLAS Level 3 API only, one should provide a reference to CPU implementation of remaining BLAS routines. This is done through a configuration file nvblas.conf with default location in current directory. There are a couple parameters, relevant to the GPU worth mentioning

```
# Put here the CPU BLAS fallback Library of your choice
NVBLAS_CPU_BLAS_LIB libopenblas.so

# Specify which output log file (default is stderr)
NVBLAS_LOGFILE nvblas.log

# List of GPU devices Id to participate to the computation
# By default if no GPU are listed, only device 0 will be used
NVBLAS_GPU_LIST 0 1
NVBLAS_AUTOPIN_MEM_ENABLED
```

'`NVBLAS_GPU_LIST ALL`' would mean all compute-capable GPUs detected on the system will be used by NVBLAS. Use of NVBLAS_AUTOPIN_MEM_ENABLED flag can be essential for good performance. See NVBLAS documentation (http://docs.nvidia.com/cuda/nvblas/index.html) for complete list of options.

'`LD_PRELOAD=libnvblas.so octave ./sgemm.m`' yields 940 GFLOPs and 1580 GFLOPs on 1 and 2 GPUs of K10 board, respectively. So, we get an immediate 2X speedup versus 2 socket IVB system just by substituting the library names – there were no code changes! Speedup can be even higher (https://developer.nvidia.com/cublasxt) for larger problem sizes.

Spectral Angle Mapper (SAM) (http://www.harrisgeospatial.com/docs/SpectralAngleMapper.html) is a well-known technique for classification of spectrum data. The data can be visualized as a 2D image or series of pixels. Each pixel is described by a 1D array, where the length is a number of bands and corresponds to spectral resolution of the spectrometer used to measure the data. One can think of that 1D array as a vector in n-dimensional space. Two vectors are alike if their mutual angle is close to zero or, in other words, their dot product is close to one. In SAM, each pixel is classified by calculating the dot product (http://www.harrisgeospatial.com/portals/0/pdfs/envi/Mapping_Methods.pdf) against all the samples from a library of reference spectra



(https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2014/04/nvblas_sam.png)

Example of SAM classification in case of 2 spectral bands. Scalar product between unknown material *x* and library sample *r*

Given a series of pixels and reference library, the procedure can be vectorized to process all the data in one (x)GEMM call. For example, 512×512 pixel image with spectral resolution of 2048 pixels and 2048 library samples, leads to (512·512)×2048 by 2048×2048 matrix-matrix product.

Below is toy implementation of SAM with random input data

```
M = 512*512; N = 2048; K = 2048;
A = rand(K,M,"single"); B = rand(K,N,"single");

# Count total elapsed time
tic

#Assume matrices are stored columnwise
A = A / diag(norm(A,"columns")); B = B / diag(norm(B,"columns"));

# Time sgemm only
start = clock();
C = transpose(B)*A;
elapsedTime = etime(clock(), start);

# Find maximum value in each column and store in a vector
SAM=max(C);
toc
disp(elapsedTime);
gFlops = 2*M*N*K/(elapsedTime * 1e+9);
disp(gFlops);
```

Here we assume the data does not come pre-normalized. Since NVBLAS is Level 3 BLAS only, norm computation and maxima's search is performed on CPU. GNU Octave stores matrices column-wise, so as an additional optimization to save on fraction of CPU time, we perform norm computation on columns and do additional transpose and change of order of multiplication in the matrix-matrix product:

	2 CPUs (20 cores)	1 GPU	2 GPUs
Measured SGEMM GFLOPS	438.8	771	1115
Total elapsed time, sec	8.6	6.4	5.5

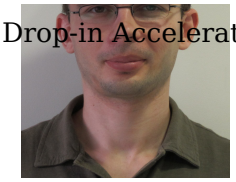
We observe 2.5X speedup in SGEMM versus dual socket Ivy Bridge. But overall speedup is 1.56X. The CPU fraction does not scale, since it is executed in single thread, and NVBLAS always uses one CPU thread per GPU. The acceleration is still significant, but we start to be limited by Amdahl's law (http://en.wikipedia.org/wiki/Amdahl%27s_law).

NVBLAS is a great way to try GPU acceleration if your application is bottle-necked by compute intensive dense matrix algebra and it is not feasible to modify the source code. cuBLAS-XT offers host C-API and a greater control of the features, if some changes of the source code are acceptable.

About the Authors

About Nikolay Markovskiy

Nikolay is HPC DevTech engineer at NVIDIA. He has PhD in Physical Chemistry from University of Southern California. He has experience in scientific research and software development focusing on

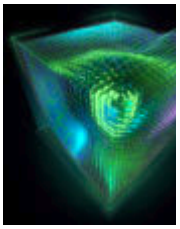


computational techniques related to physics, chemistry, and biology.
View all posts by Nikolay Markovskiy » (https://devblogs.nvidia.com/author/nmarkovskiy/)

Related posts

CUDA 9 Features Revealed: Volta, Cooperative Groups and More (https://devblogs.nvidia.com/cuda-9-features-revealed/)

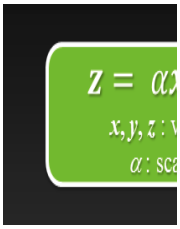
By Mark Harris (https://devblogs.nvidia.com/author/mharris/) | May 11, 2017 (https://devblogs.nvidia.com/cuda-9-features-revealed/)



(https://devblo /cuda-9-featur

Six Ways to SAXPY (https://devblogs.nvidia.com/six-ways-saxpy/)

By Mark Harris (https://devblogs.nvidia.com/author/mharris/) | July 2, 2012 (https://devblogs.nvidia.com/six-ways-saxpy/)



(https://devblo /six-ways-saxp

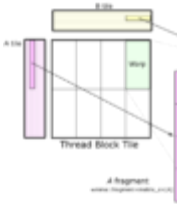
CUDA Pro Tip: How to Call Batched cuBLAS routines from CUDA Fortran (https://devblogs.nvidia.com/cuda-pro-tip-how-call-batched-cublas-routines-cuda-fortran/)

By Greg Ruetsch (https://devblogs.nvidia.com/author/gruetsch/) | March 5, 2014 (https://devblogs.nvidia.com/cuda-pro-tip-how-call-batched-cublas-routines-cuda-fortran/)

(https://devblo /cuda-pro-tip- batched-cubla cuda-fortran/)

CUTLASS: Fast Linear Algebra in CUDA C++ (https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/)

By Andrew Kerr (https://devblogs.nvidia.com/author/akerr/), Duane Merrill (https://devblogs.nvidia.com/author/dumerrill/), Julien Demouth (https://devblogs.nvidia.com/author/jdemouth/) and John Tran (https://devblogs.nvidia.com/author/jotran/) | December 5, 2017 (https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/)



(https://devblo /cutlass-linear cuda/)

Comments