Dynamic GPU Acceleration of Linear Algebra Computations

Princeton Ferro
University of Rochester
pferro@u.rochester.edu

December 18, 2017

Abstract

We investigate a way of running on the GPU software that performs CPU linear algebra computations, without modifying program code. We use a strategy called library interposition to affect how the dynamic linker does symbol resolution on Basic Linear Algebra Subprograms (BLAS), allowing us to supply a replacement that runs the linear algebra computation on the GPU. This strategy is used by NVBLAS to provide "drop-in" acceleration of BLAS. This strategy works mainly for Level 3 BLAS because the overhead in transferring data to the GPU is outweighed by the speedup in performing the matrix-matrix computations. It falls short for Level 1 (vector-vector) and Level 2 (matrix-vector) routines where the execution is dominated by the data transfer overhead. We consider a supplement to this strategy by interposing `malloc`, `free`, and friends, allowing us to track use of memory objects in BLAS routines. By collecting this information, we could preemptively allocate shared memory between the CPU and GPU and eliminate explicit data transfers, allowing for a greater throughput in linear algebra computations. We evaluate the performance impact of our approach on micro-benchmarks, as well as scientific computing software such as GNU Octave.

Contents

1 Introduction

Many researchers use scientific packages like NWChem to simulate physical interactions, or scripting environments like Octave and NumPy to process data. Linear algebra computations are employed for these tasks. These computations are encapsulated into optimized subroutines in a library like Intel MKL, OpenBLAS, or LAPACK, which all run on the central processor. However, the graphics processor is better-suited to performing many linear algebra computations with a high degree of parallelism. Improving performance in these intermediate calculations has the potential to significantly increase efficiency in research that relies on this scientific software.

NVidia's CUDA runtime allows programmers to run code written in the CUDA language on the GPU. There also exist runtimes for the C, C++, Fortran, and other languages. Perhaps one reason why software hasn't taken advantage of this runtime is the effort needed to write a GPU-based layer for existing code, while maintaining another layer for devices that don't have NVidia GPUs. Another factor is the desire to be portable to other GPUs, which is not supported by CUDA. A runtime solution would avoid these drawbacks.

2 Related Work

NVidia's NVBLAS [nvb17] tackles the same problem domain, using library interposition to accelerate BLAS Level 3 routines, but not Levels 1 and 2. The reason is that Level 3 routines are computation-intensive matrix-matrix operations, so for large matrices the execution time is dominated by computation on the CPU. Hence, moving this computation to the GPU reduces computation time more than it increases data transfer overhead. This is not the case for vector-vector and vector-matrix operations. This is not an ideal solution for a few reasons:

1. Level 1 and 2 routines are not GPU-accelerated due to the data transfer overhead. However, it may be possible for certain computations to see a speedup if the data transfer overhead could be overcome.

2. Even though Level 3 routines are accelerated, the implementation must still perform manual memory transfers before and after kernel execution. This incurs overhead.

3 Methods

3.1 Setup

We developed software to run on Linux-based operating systems. Our approach to GPU-acceleration uses NVidia's CUDA environment. Our tests are on Centos 7 with an NVIDIA Quadro M4000 with 8GB DDR5 and an Intel Xeon E7 v4 with 32 threads at 2.80 GHz.

3.2 Library Interposition

Linux's dynamic linker, `ld.so`, recognizes the `LD_PRELOAD` environment variable, which specifies a shared object to load first, causing all subsequent symbol resolution to go through this library. We use this approach to override cBLAS (`cblas_*`) and BLAS routines.

3.3 Tracking Objects

We also use library interposition to override some of the memory management routines in the GNU C Library (`malloc`, `realloc`, and `free`). We define an *allocation* as any call to `malloc` or a similar memory management routine, and a *deallocation* as any call to `free` on a valid object. On an allocation, we record the address of the object along with requested size and the return address. This is the call information, which is stored as the object's *metadata*.

There are two phases to object tracking:

Phase 1 We generate a trace of the desired program. The trace file contains the information identifying each allocation by the requested size and the saved return address in `malloc`'s stack frame. We select those allocations that are for matrices and vectors that may later be passed as arguments to a BLAS routine. This produces the trace in its final form.

Phase 2(a) Once we have the trace, we run the program again. At any time `malloc` and `calloc` are called:

1. Compare the call information with the trace.
2. If the call information matches, allocate the object using the custom memory manager and record the object's metadata in a global structure. Otherwise, allocate using the standard memory management routines from the C library, and do not track the object.

For the custom memory manager, we use `cudaMallocManaged`, which allocates a buffer that is shared between the CPU and GPU. Instead of using CUDA APIs to explicitly move data, this is done automatically by the graphics driver on a page fault.[cud17]

Phase 2(b) When an overridden BLAS routine is invoked, we lookup the pointer arguments in our storage of object metadata. Lookup takes $O(\log n)$, where n is the number of tracked memory objects.[1] If there is an entry, we need not explicitly copy the argument to the GPU. Then we invoke the equivalent CUDA API call for the BLAS routine.

4 Limitations

Because object tracking requires knowledge of the program counter offset, which we use as an invariant between each program invocation, to obtain this information requires unwinding the stack. However, there is more involved. Recently, compilers have decided to omit using the frame pointer (%ebp) register by default as an optimization.[Fou17] This complicates a stack unwinding implementation. Furthermore, it is not enough to get the saved instruction pointers of every frame, which change under ASLR. What is invariant is the offset of the program counter within the current function. But this information must be parsed from the debug information in other sections of the program. This is what libunwind does.

Unfortunately, there is a significant performance cost associated with debug-level stack unwinding, and this contributes majorly to the overhead present in object tracking. Consequently, it was necessary to come up with a more efficient approximation.

The approximate method was to take the differences between saved instruction pointers of successive calls in the call chain. Unfortunately, this method does not work for ASLR. The data collected below was done under ASLR and therefore does not reflect any potential benefit from the object-tracking strategy.

5 Results

We compare our implementation, blas2cuda[Fer17], against NVBLAS + Intel MKL, and just Intel MKL.

5.1 Micro-benchmarks

We use Linux's `clock` API to measure execution time for BLAS operations. We timed matrix multiplication on square matrices of dimension up to $n = 2^{14}$. Below are the times in seconds for each computation.

[1]This metadata is stored using `tsearch` from the GNU C library, which is implemented with a red-black tree.[gli17]

	blas2cuda	Intel MKL	NVBLAS
General MM			
$n = 2$	0.00001	0.00000	0.00000
$n = 4$	0.00001	0.00000	0.00000
$n = 8$	0.00001	0.00000	0.00000
$n = 16$	0.00001	0.00000	0.00000
$n = 32$	0.00005	0.00000	0.00000
$n = 64$	0.00005	0.00000	0.00002
$n = 128$	0.00002	0.00001	0.00008
$n = 256$	0.00008	0.00004	0.00029
$n = 512$	0.00020	0.00018	0.00226
$n = 1024$	0.00072	0.00098	0.00256
$n = 2048$	0.00291	0.00583	0.00479
$n = 4096$	0.00339	0.00451	0.00358
$n = 8192$	0.00598	1.01000	0.00783

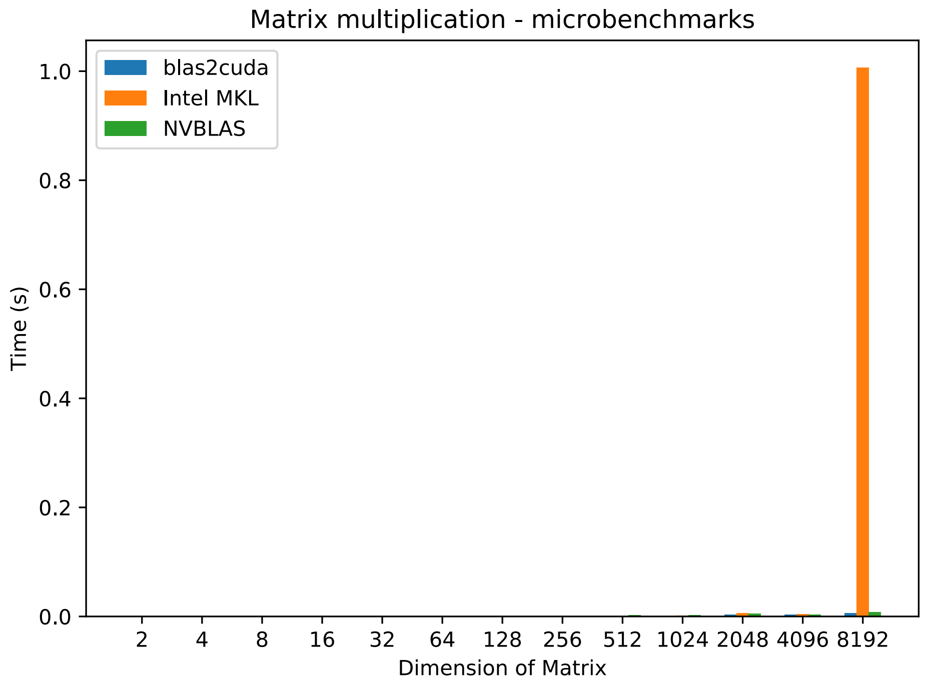

Figure 1: General Matrix Multiplication

	blas2cuda	Intel MKL	NVBLAS
MM with Complex			
$n = 2$	0.00001	0.00000	0.00000
$n = 4$	0.00001	0.00000	0.00000
$n = 8$	0.00001	0.00001	0.00000
$n = 16$	0.00006	0.00004	0.00000
$n = 32$	0.00006	0.00006	0.00001
$n = 64$	0.00003	0.00006	0.00010
$n = 128$	0.00008	0.00005	0.00028
$n = 256$	0.00014	0.00012	0.00111
$n = 512$	0.00034	0.00045	0.00402
$n = 1024$	0.00135	0.00317	0.00509
$n = 2048$	0.00713	0.00394	0.00002
$n = 4096$	0.00018	0.00965	0.00932
$n = 8192$	2.01000	4.00000	2.01000

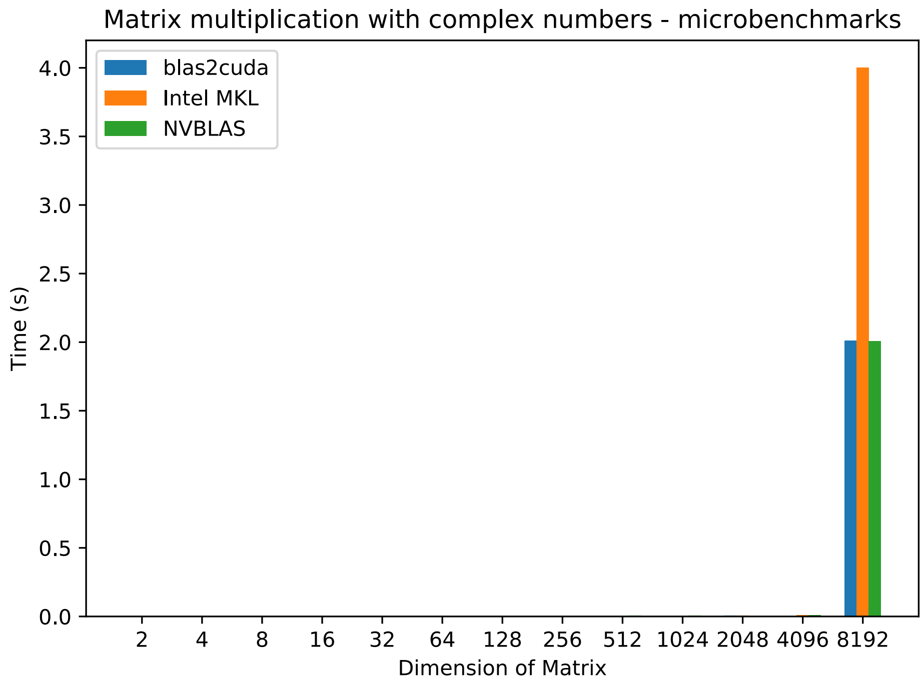

Figure 2: Matrix Multiplication with Complex Numbers

5.2 Octave

Octave only uses the `{s,c,d,z}gemm` Level 3 BLAS routine for matrix-matrix multiplication. Below are the times in seconds for each computation.

	blas2cuda	Intel MKL	NVBLAS
General MM			
$n = 2$	0.00473	0.00010	0.00047
$n = 4$	0.00380	0.00010	0.00036
$n = 8$	0.00351	0.00011	0.00009
$n = 16$	0.00370	0.00010	0.00009
$n = 32$	0.01080	0.00012	0.00011
$n = 64$	0.01070	0.00027	0.00020
$n = 128$	0.00450	0.00075	0.00064
$n = 256$	0.01170	0.00172	0.00278
$n = 512$	0.02640	0.00839	0.07640
$n = 1024$	0.03370	0.05930	0.10500
$n = 2048$	0.07500	0.31300	0.13400
$n = 4096$	0.22600	1.86000	0.28700
$n = 8192$	1.09000	13.50000	0.84900

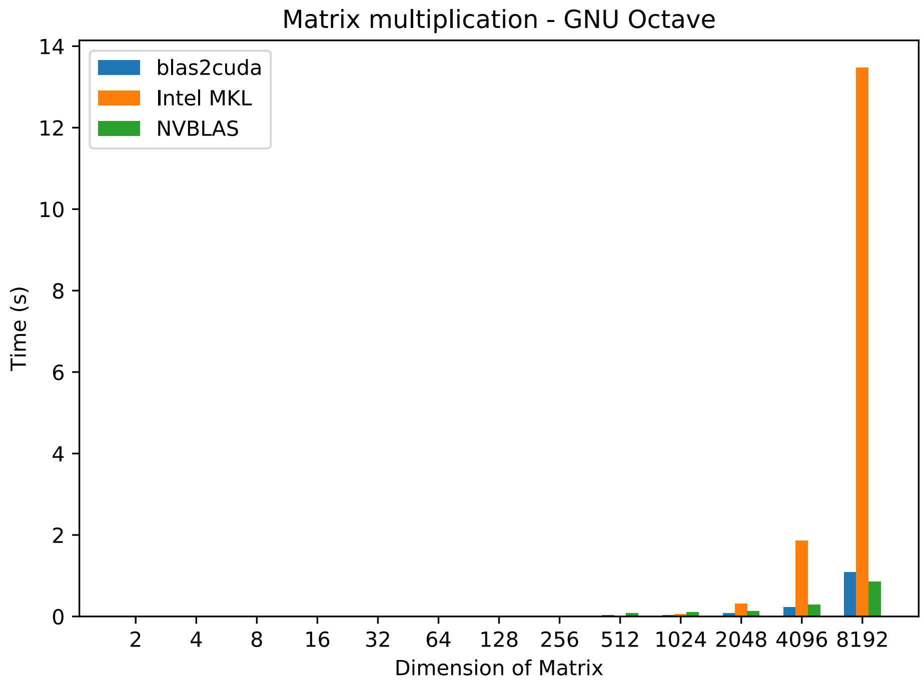

Figure 3: Matrix Multiplication - GNU Octave

5.3 Conclusions

The data show that a naive approach to GPU acceleration without object tracking, even still, yields considerable performance gains of 2x to 10x. The challenge of future work is to improve performance of stack unwinding and stack walking, as well as to conduct more tests with other scientific software.

6 Acknowledgments

Thanks to Dr. Sreepathi Pai, who supervised this research. Thanks to the Center for Integrated Research Computing at the University of Rochester for providing resources. Thanks to the University of Rochester CS Lab Staff, and James Roche in particular, for installing software needed for this research.

References

[cud17] Data migration and coherency. https://web.archive.org/web/20171120171620/http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-data-migration, November 2017.

[Fer17] Princeton Ferro. blas2cuda - translation layer from blas to cuda blas. https://github.com/Prince781/blas2cuda/, December 2017.

[Fou17] GNU Foundation. Options that control optimization. https://web.archive.org/web/20170916171321/https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, 2017.

[gli17] sourceware.org git - glibc.git/blob - misc/tsearch.c. https://sourceware.org/git/?p=glibc.git;a=blob;f=misc/tsearch.c;hb=bfff8b1becd7d01c074177df7196ab327cd8c844, January 2017.

[nvb17] Nvblas :: Cuda toolkit documentation. https://web.archive.org/web/20170709180114/http://docs.nvidia.com/cuda/nvblas/index.html, July 2017.