Autotuning dense linear algebra libraries on GPUs

Rajib Nath, Stanimire Tomov, and Jack Dongarra
University of Tennessee

Emmanuel Agullo
INRIA Bordeaux Sud Ouest / LaBRI

***Abstract*—As GPUs are quickly evolving in complexity, tuning numerical libraries for them is becoming more challenging. We present an autotuning approach in the area of dense linear algebra (DLA) libraries for GPUs. The MAGMA library is used to demonstrate the techniques and their effect on performance and portability across hardware systems. We show that, figuratively speaking, our autotuning approach for GPUs can be regarded as both *the beauty* and *the beast* behind hybrid DLA libraries as it is an elegant and very practical solution for easy maintenance and performance portability, while often being a brute force, empirically based exhaustive search that would find and set automatically the best performing algorithms/kernels for a specific hardware configuration. The exhaustive search is often relaxed by applying various performance models.**

Keywords: GPU BLAS, autotuning, hybrid computing, dense linear algebra, multicore processors

Automatic performance tuning (optimization), or autotuning in short, is a technique that has been used intensively on CPUs to automatically generate near-optimal numerical libraries. For example, ATLAS [1], [2] and PhiPAC [3] are used to generate highly optimized BLAS. With the success of autotuning techniques on generating highly optimized DLA kernels on CPUs, it is interesting to see how the idea can be used to generate near-optimal DLA kernels on modern GPUs. Indeed, work in the area [4] has already shown that autotuning for GPUs is very practical solution to easily port existing algorithmic solutions on quickly evolving GPU architectures and to substantially speed up even highly hand-tuned kernels.

There are two core components in a complete autotuning system: **Code Generator** and **Heuristic Search Engine**. The code generator produces code variants according to a set of pre-defined, parametrized templates/algorithms. The code generator also applies certain state of the art optimization techniques. The code generator is very application and algorithm specific. For example in case of a GEMM ($C = \alpha AB + \beta C$) kernel autotuner, five parameters can critically impact performance, and therefore are incorporated in a GEMM code generator. This choice though can be extended and enhanced with various optimization techniques: tuning number of threads per thread block, tuning the choice whether to put matrix A or B in shared memory, tuning the size and layout of the shared memory to avoid bank conflicts, prefetching into registers, applying other state of the art loop optimization techniques such as circular loop skewing to avoid partition camping in GPU global memory, etc. The code generator takes precision as a parameter as well, in order to apply the same mechanism for various precisions. The heuristic search engine runs the variants produced by the code generator and finds out the best one using a feedback loop, e.g., the performance results of previously evaluated variants are used as a guidance for the search on currently unevaluated variants.

One way to implement autotuning is to generate a small number of variants for some matrices with typical sizes during installation time, and choose the best during run time, depending on the input matrix size and high level DLA algorithm.

There are several success stories of the autotuner. For a variation of DGEMM on square matrices, the autotuner was able to find a better kernel which runs 15% faster than the CUBLAS 2.3 DGEMM. For another variation of SGEMM, CUBLAS has performance deteriorations for certain problem size (e.g., up to 50% of the total performance). Interestingly, the autotuner was successful in finding a better kernel by applying circular loop skew optimization. Also in case of a Level 2 BLAS, in particular SSYMV, with a new recursive block based algorithm, the autotuner was able to attain slightly above 100 GFlops/s performance in GTX280 as compared to 2 GFlops/s in CUBLAS.

Moreover, in the area of DLA it is very important to have high performance GEMMs involving rectangular matrices. This is dictated by the fact that the algorithms are blocked, and the blocking size is in general small, resulting in GEMMs involving on rectangular matrices. In this case the autotuner found kernels to significantly outperforms (up to 27%) the DGEMM from CUBLAS 2.3.

These results support experiences and observations by others on "how sensitive the performance of GPU is to the formulations of your kernel" [5] and that an enormous amount of well thought experimentation and benchmarking [6], [5] is needed in order to optimize performance.

References

[1] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. Parallel Computing **27** (2001), no. 1-2, 3–35.

[2] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. Proc. IEEE 93 (2005), no. 2, special issue on "Program Generation, Optimization, and Adaptation".

[3] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. International Conference on Supercomputing, 1997, pp. 340–347.

[4] Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In ICCS'09, pages ,884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Michael Wolfe, Compilers and More: Optimizing GPU Kernels. HPC Wire, http://www.hpcwire.com/features/33607434.html, 10/2008.

[6] V. Volkov and J. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.