Evaluating the Performance of Parallel Linear Algebra Libraries for Level-1 BLAS

Douglas A. Augusto
DMA – IM – UFRJ
daa@labma.ufrj.br

Luiz Mariano Carvalho
DMA – IME – UERJ
luizmc@ime.uerj.br

Daniel Estrela
PPGEM -FEN – UERJ
destrela@gmail.com

Brunno F. Goldstein
PESC – COPPE – UFRJ
bfgoldstein@cos.ufrj.br

Paulo Goldfeld
DMA – IM – UFRJ
goldfeld@ufrj.br

Michael Souza
DEMA – UFC
michael@ufc.br

Abstract: We present a performance evaluation of the scalar-vector product (axpy) operation on four widespread linear algebra libraries. Benchmarks are performed for multi-cores and many-cores architectures and the results are compared.

Keywords: *Scalar-vector product, Linear Algebra libraries, Intel MKL, CUDA, ViennaCL*

1 Introduction

This document aims at presenting a performance comparison of different parallel linear algebra libraries, Intel MKL, Cusp, CuBLAS and ViennaCL, on the fundamental axpy operation (BLAS Level 1). More precisely, we are interested in measuring how fast they compute $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$, where α is a scalar and $\mathbf{x}$ and $\mathbf{y}$ n-dimensional dense vectors.

Intel *Math Kernel Library* (MKL)[1] is a proprietary package of mathematical routines, including a rich set of linear algebra, that are optimized for Intel processors. *cuBLAS*[2] is the NVIDIA CUDA version of the complete standard BLAS library. *Cusp*[3] is an open-source library of generic parallel algorithms for sparse matrix and graph computations on CUDA. ViennaCL[4] is also an open-source linear algebra library, but takes advantage of the portability offered by OpenCL to seamlessly support multi-core and many-core processors from different vendors and architectures.

2 Experiments

2.1 Computational environment

The experiments were performed on a 64-bit GNU/Linux machine equipped with two 16-core Intel Xeon E5-2650v2 processor at 2.6GHz (with hyperthreading) and an NVIDIA Titan GPU (driver version 319.82). All the experiments were compiled with the optimization flag enabled. The following versions of the libraries were used: Intel MKL 11.1, NVIDIA cuBLAS 5.5, Cusp 3.0 and ViennaCL 1.5.1.

The results were obtained using double-precision floating point arithmetic and reported in GFlop/s, measured as $GFlop/s = \frac{fp_{operations}}{execution_{time} \times 10^9}$, where $fp_{operations}$ is given by $2n$, accounting for the scalar-vector multiplication and sum of two vectors, and $execution_{time}$ is the wall-clock which measures the time taken by the axpy operation [1].

[1] http://software.intel.com/en-us/intel-mkl
[2] http://developer.nvidia.com/cuBLAS
[3] http://cusplibrary.github.io/
[4] http://viennacl.sf.net

2.2 Results

The benchmark results in our experiments represent the average performance of one thousand executions of the `axpy` operation using two groups of vectors sizes: small vectors (2^{10}, 2^{11}, 2^{12} and 2^{13} dimensions) and large vectors (2^{20}, 2^{21}, 2^{22} and 2^{23} dimensions). In this work we adopt the methodology utilized and justified by Bell *et al.* [1], in which the authors say *"our measurements do not include time spent transferring data between host (CPU) and device (GPU) memory, since we are trying to measure the performance of the kernels"*.

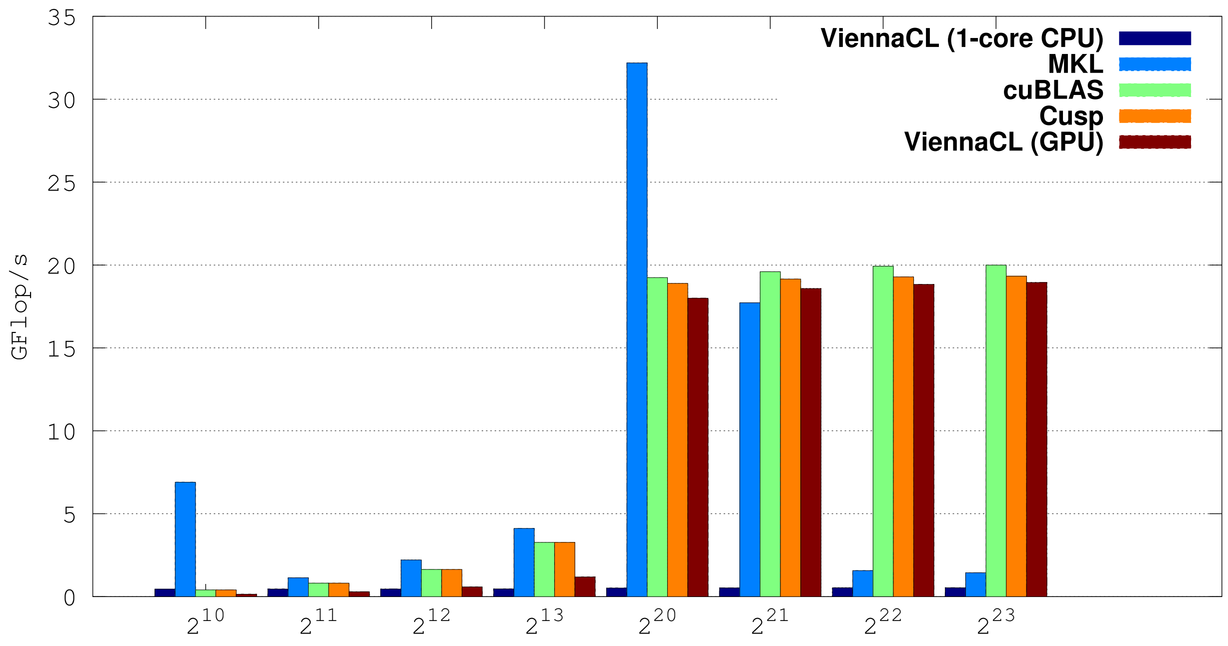

Figure 1: Performance of Intel MKL, cuBLAS and Cusp in GFlop/s.

Figure 1 summarizes the performances achieved by each library on all considered instances, including a baseline measured using ViennaCL on only one CPU core. One can see that although Intel MKL attains the highest throughput on the 2^{20} instance, its performances consistently degrade on the largest vector dimensions (2^{21}, 2^{22} and 2^{23}), while the GPU is able to sustain the same performance level after saturation. Without a more detailed analysis it is hard to tell why exactly Intel MKL behaves strangely with respect to the 2^{10} and 2^{20} instances, but it might have to do with how well the storage requirements of such vectors fit the cache hierarchy. With respect to cuBLAS and Cusp, they obtained the same results on the small vectors, but the former performed sightly better than the latter on the large workloads. Although being competing on the large vectors, it is very noticeable the difficult of ViennaCL in getting high performance out of the small ones on the GPU; this suggests, though, there is great room for improvement in this library.

Given the results of our set of benchmarks, the next logical step as a follow-up study is to investigate (i) why MKL peaks at 2^{10} and 2^{20} and rapidly degrades from 2^{22} on; and (ii) what is preventing ViennaCL from performing well on the small vector dimensions.

References

[1] Nathan Bell and Michael Garland. "Efficient sparse matrix-vector multiplication on CUDA". NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.