Performance Improvements of ?nrm2

Eleni Vlachopoulou

AMD

Agenda

Introduction to Euclidean Norm

Initial Implementation and Limitations

New Implementation using Old Algorithms

Q&A

Introduction

- The Euclidean norm (or vector-2 norm) of a vector $x \in \mathbb{C}^n$, is the function defined as $\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$

$$\|x\|_2 = \left(\sum_{i=0}^{n-1} |x_i|^2\right)^{\frac{1}{2}} = \sqrt{|x_0|^2 + |x_1|^2 + \cdots + |x_{n-1}|^2}.$$

- For complex numbers, the absolute value of $x = a + bi$ is defined as $|a + bi| = \sqrt{a^2 + b^2}$, so computing the norm of x is equivalent of the norm of computing a real vector y with twice the size, as shown below

$$\|x\|_2 = \left\| \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} a_0 + b_0 i \\ a_1 + b_1 i \\ \vdots \\ a_{n-1} + b_{n-1} i \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} a_0 \\ b_0 \\ a_1 \\ b_1 \\ \vdots \\ a_{n-1} \\ b_{n-1} \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{2n-1} \end{bmatrix} \right\|_2 = \|y\|_2 .$$

- Used in error analysis, Singular Value Decomposition (SVD) and eigenvalue solvers.

Overflow and Underflow in Norm Computation

- Let L be the largest positive floating-point number. Then, for any positive number x, such that $x > \sqrt{L}$, x^2 will overflow.

  For example, for x = 1e+200, x*x = inf on double precision.

- Let S be the smallest positive floating-point number. Then, for any positive number x, such that $x < \sqrt{S}$, x^2 will underflow.

  For example, for x = 1e-200, x*x = 0 on double precision.

- A simple fix to avoid overflow is scaling all elements using the maximum vector element. That is, if $x_{max} = \max_i |x_i|$, then the norm of x can be computed as

$$\|x\|_2 = x_{max} \sqrt{\sum_{i=0}^{n-1} (x_i / x_{max})^2}$$

- Similarly, there is a fix to avoid underflow and the two fixes can be combined to compute the norm accurately.

Initial Implementation and Limitations

- Initial implementation of Euclidean norm is based on ?lassq() and uses the sum of squares in a scaled form.
- Norm is computed as $\|x\|_2 = scl * \sqrt{sumsq}$, where $scl = 0$ and $sumsq = 1$ are set, before iterating through the elements of x.
- For each i, we compare $|x|$ to scl, and update scale to have the maximum value of $|x_i|$.
- Since we compare and update scl at each iteration, there is a dependency between the i-th element and the (i-1)-th element, through scl.
- This algorithm uses many divisions, which are relatively expensive.

Parallelizing this algorithm is not trivial.

Blue's Algorithm

- Iterate through the elements of x and compute the sums as follows

$$sum_L = \sum_i (x_i/s_L)^2 \,, \text{ for } i \text{ s.t. } |x_i| > t_L$$: accumulator for large values, using scaling to avoid overflow

$$sum_S = \sum_i (x_i/s_S)^2 \,, \text{ for } i \text{ s.t. } |x_i| < t_S$$: accumulator for small values, using scaling to avoid underflow

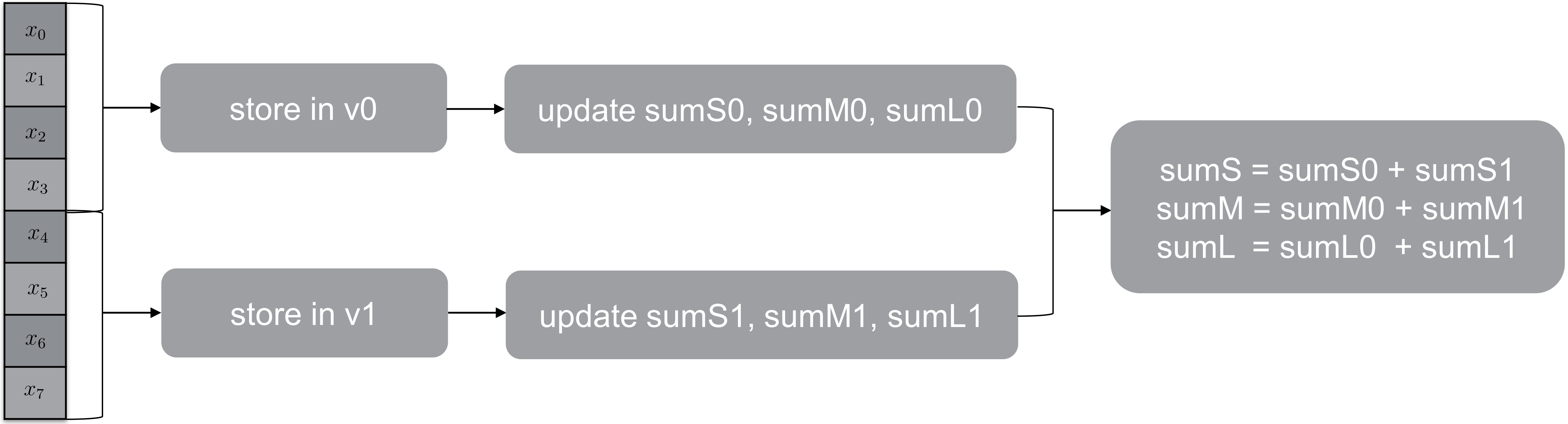
$$sum_M = \sum_i x_i^2, \text{ for } i \text{ s.t. } t_S \leq |x_i| \leq t_L$$: accumulator for medium values

- If there are only medium values, then $norm = \sqrt{sum_M}$.
- If there are large values, scale medium value accumulator and compute the norm.
- Similarly, for small values when large values are not present.
- The scalars $t_L,\ t_S,\ s_L$ and s_S are computed using machine constants.

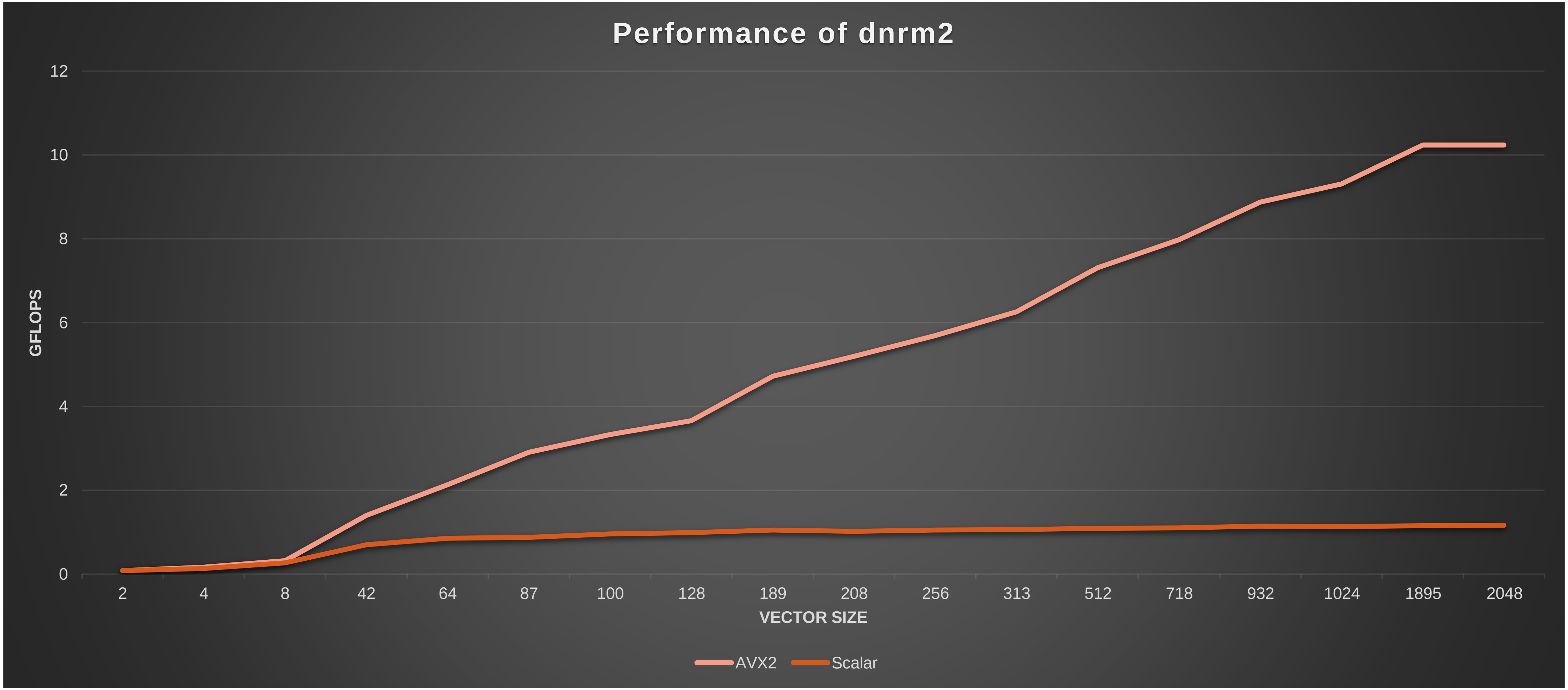
Since the thresholds and the scaling factors are independent of the elements of x, the iterations are independent, and parallelizing this algorithm is easier.

AVX2 Implementation of Blue's Algorithm

- Since the operations for each element of the vector are independent, we can use vectorization.
- Use partial sums for each of the sum_L, sum_M, sum_S to optimize further.
- Use masks and blend operations to compute the partial sums correctly.



Benchmark Results



Final Notes

- For complex numbers, we use the same algorithm on a vector y with double the size of x.
- For simplification, the details of NaN and Inf checks have been omitted, but they are implemented using AVX2 intrinsics.
- Overflow and underflow is being handled correctly through scaling.
- Only AVX2 intrinsics have been used, no OpenMP parallelism.

Reference

J.L. Blue, "A portable Fortran program to find the Euclidean norm of a vector", in *ACM Transactions on Mathematical Software (TOMS)*, *4*(1), pp.15-23, 1978.

Questions?

AMD

DISCLAIMER AND ATTRIBUTIONS

DISCLAIMER

The information contained herein is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD