Implementing 2x2 SVD

Chaz Gwennap, Jimmy Wu

12/24/12

Summary: The goal of this project was to create a program which could calculate the singular value decomposition of a 2x2 matrix on a custom MIPS-like processor. We coded a high-level program which did SVD and implemented in assembly some advanced math functions required for coding SVD in assembly.

Background:

Singular value decomposition is a method of breaking up the transforming function of a matrix into simpler actions. In general, a matrix has a unique pair of orthogonal vector spaces where if you apply the matrix to one, it will be mapped to the other. SVD decomposes a matrix into a product of three other matrices which are a rotation to the right, a scaling, and another rotation to the left. In short, the SVD of a matrix A can be expressed as A = U∑V^T, where U and V are rotations and the ∑ provides the information for scaling. The columns of U are the basis vectors for one of the aforementioned vector spaces. Likewise goes for V. The third matrix, ∑, is a diagonal matrix, where the values on the diagonal are the scaling factors. SVD has a number of applications in computer vision, image processing and compression, and data processing (for example, it is used in the Netflix algorithm which suggests movies for you based on the ones you've seen already).

The purpose of this lab was to create a CPU in hardware language that can compute the SVD of any 2x2 matrix. We were inspired by a paper published by Kishore Kota, which outlines the design of a highly optimized SVD processor which can compute the SVD of a pxp matrix in O(p log p) time, using O(n^1.5) area complexity. Conventional hardware methods would require O(p^3) time and O(n^2) area. Kota's approach involves the use of a central math processor capable of performing CORDIC calculations, and a grid of small processors to store and manipulate the entries of the matrix – this grid is known as a systolic array. Each processor in the grid can compute the SVD of a 2x2 matrix, and work together to solve the pxp matrix as a whole.

Our initial approach to this problem started by building off of an incomplete CPU that was developed prior to the project. We needed to complete this CPU in order to have the central CORDIC processor. The CPU was only capable of register manipulations by basic math functions, passed to the instruction fetcher as binary instructions.

We proceeded to implement the majority of a reduced instruction set based on the MIPS assembly specifications, including the loading and moving of immediate values into the register file. A compiler was built (in Python) so that assembly code could be written in a text file and translated into machine instructions. The next step would have been to develop the CORDIC operations into an assembly library. After that, we could have written the SVD operations in assembly, referencing the CORDIC functions.

Design:

SVD:

For a 2x2 matrix A with values [a , b; c, d], calculating SVD is accomplished through the following steps:

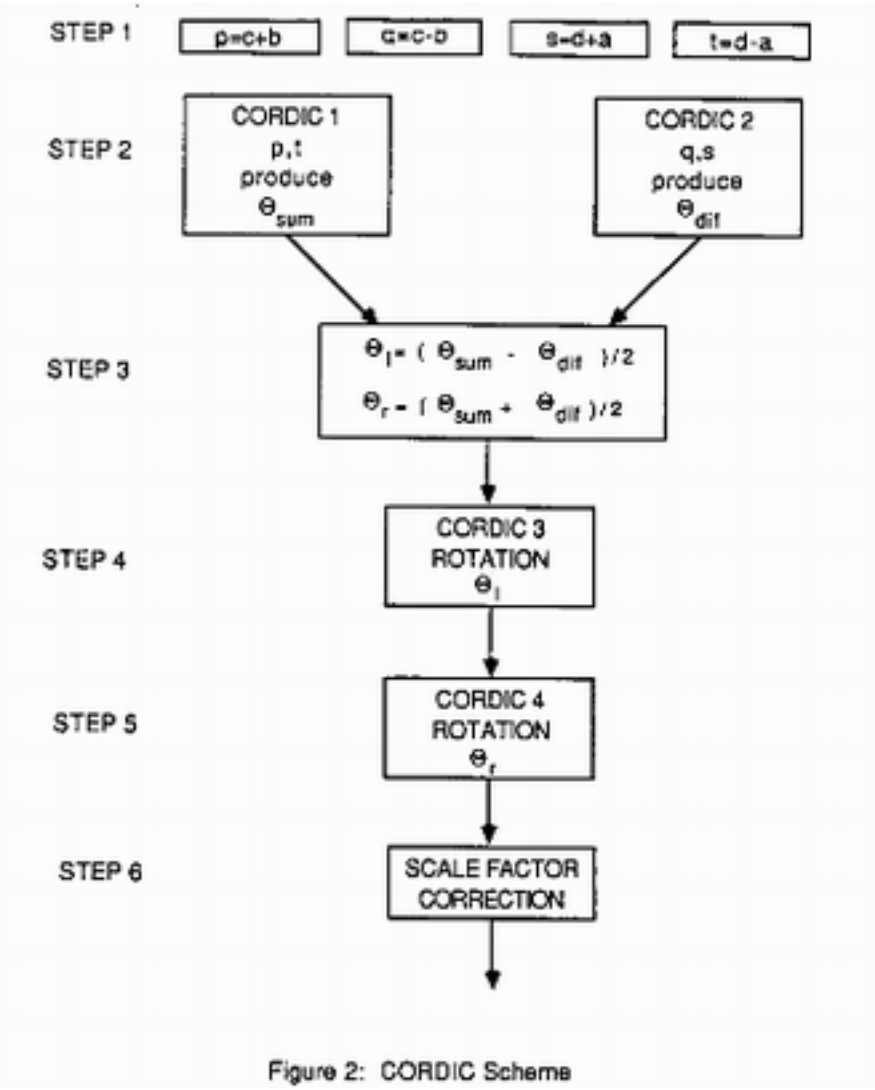

Figure 2: CORDIC Scheme

(Source: http://ftp.cs.ucla.edu/tech-report/198_-reports/870043.pdf)

First, the sums p, t, q, and s are calculated, then CORDIC is then used to calculate the inverse tangent of vectors (p,t) and (q,s). Once Θ_l and Θ_r, the amount required to rotate, have been calculated, the rotation matrix can also be calculated, with the style:

$$R(\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

(Source: http://ftp.cs.ucla.edu/tech-report/198_-reports/870043.pdf)

$\sum$, the diagonal matrix and scaling factor, can then be calculated through multiplying the transpose of $R(\Theta_l)$ by A and then multiplying the product by $R(\Theta_r)$. Thus, all the components of SVD have been calculated. Multiplying $R(\Theta_l)$ by $\sum$ and then multiplying the product by the transpose of $R(\Theta_r)$ should return the original matrix A.

We implemented and tested this SVD design using MATLAB, a high level programming language with matrix math and trigonometry already implemented. We then worked to implement SVD using MIPS assembly code.

Fixed Point Math:

Because our MIPS-like processor doesn't have fixed-point math built in, we had to code it ourselves in MIPS. We chose to implement objects with signed I16Q16 precision, 16 bits before and after the decimal place, as a precursor to implementing floating point addition/multiplication. As a proof of concept, I16Q16 is good enough to demonstrate the validity of our algorithms.

Addition: Adding in fixed point can be done before using standard addition, but with careful attention paid to overflow correction, cases in which 3+5 = -8 because of summing into a negative number. Thus, careful attention must be paid to the input signs and the output signs. For adding any two same-sized fixed point numbers, the following table check applies:

Sign (args)	Sign (result)	Overflow?
mixed (one positive, one negative)	any	no
both positive	positive (0...)	no
both positive	negative (1...)	yes
both negative	negative (1...)	no
both negative	positive (0...)	yes

When adding two negative numbers, overflow is guaranteed (two 32 bit negative numbers will add to a 33 bit number), but when the most significant value is truncated to recreate the original fixed-point notation, this logic is applicable.

Multiplication: Multiplying is significantly more complex because multiplying two I16Q16 numbers together nets a 64-bit product with I32Q32 fixed-point notation. In this case, to get back to a 32-bit number, you must truncate the 16 most significant digits and truncate/roundoff the 16 least significant digits. For overflow, you must check the most significant digits before truncating. In general, the following table applies:

Sign (args)	top 16 bits (result)	Overflow?
mixed (one positive, one negative)	???	no
mixed (one positive, one negative)	???	yes
both same sign	zero	no
both same sign	nonzero	yes

Due to time constraints, we were unable to find a satisfactory generalization for overflow detection for the product of mixed numbers.

Correctness:

SVD: Our SVD function in MATLAB correctly does SVD for matrix [4, 3; 2, 1], but was non-functional with most other matrixes, such as those with negative numbers.

Fixed Point Addition: We coded a I2Q2 fixed point adder and tested for correct handling of any combination of positive and negative numbers, and for correct handling of positive and negative overflow situations. We then generalized it to make a I16Q16 adder which should be just as effective, because it uses the exact same concepts as the I2Q2 adder.

Fixed Point Multiplication: We coded a I16Q16 fixed point multiplier and tested for correct handling of any combination of positive and negative numbers, but did not implement handling of overflows.

Time Analysis:

Fixed point addition takes typically 8 instruction cycles to complete, with a maximum of 10 instruction cycles.

Fixed point multiplication takes 8 instruction cycles to complete, without taking overflow detection into account.

References/Further Reading:

Ercegovac, Milos D., and Tomas Lang. "On-line Schemes for Computing Rotation Angles for SVDs." UCLA CS Department, n.d. Web. 25 Dec. 2012. http://ftp.cs.ucla.edu/tech-report/198_-reports/870043.pdf

Kota, Kishore. "Architectural, Numerical and Implementation Issues in the VLSI Design of an Integrated CORDIC-SVD Processor." N.p., n.d. Web. 25 Dec. 2012. http://scholarship.rice.edu/bitstream/handle/1911/20042/Kot1999Aug2Architectu.PDF?sequence=1

http://en.wikipedia.org/wiki/Singular_value_decomposition: Wikipedia article on SVD.

http://en.wikipedia.org/wiki/CORDIC: Wikipedia article on CORDIC.