Accelerating Householder Bidiagonalization with ARM NEON Technology

Wenjun Yang* and Zhenyu Liu
The Department of Computer Science and Technology, Tsinghua University, Beijing, China
*yangwj10@mails.tsinghua.edu.cn

***Abstract*—Householder bidiagonalization is the first step of Singular Value Decomposition (SVD) – an important algorithm in numerical linear algebra that is widely used in video processing. NEON is a general-purpose Single Instruction Multiple Data (SIMD) engine introduced in ARMv7 architecture, which is targeted to accelerate multimedia and signal processing on mobile platforms. In this paper, we propose a NEON-based implementation and optimization of Householder bidiagonalization, aiming at testifying the potential of NEON to handle with low-dimensional macroblocks if applied to future computing-intensive video codecs. Intrinsics and inline assembly, two most commonly used ways to utilize NEON, are compared in performance. Solutions to the problem of leftover elements in vectorization is also discussed. Our study finally shows that with hand-coded inline assembly and all kinds of optimization, our NEON implementation of Householder bidiagonlization will gain a speedup of 2.3 over the plain C version.**

I. Introduction

Adaptive algorithms are increasingly applied to video codecs nowadays due to the advantage of efficiency. For instance, High Efficiency Video Coding (HEVC) [1], the ongoing next-generation video compression standard, has introduced adaptive algorithms such as adaptive loop filter, adaptive rounding offset and rate distortion optimized quantization (RDOQ), to surpass H.264/MPEG-4 [2] on both video quality and data compression ratio. Moreover, alternatives to existing modules in current standards are proposed to be adaptive, in order to evolve and specialize from input data, eliminating bad cases in traditional static schemes. Mode-dependent Directional Transform (MDDT) [3] and its successors [4] are examples of such alternatives, intended for a replacement of Discrete Cosine Transform (DCT). Good results as they provide, these algorithms also bring in significant increase on computation, making them less attractive in practice. Our goal is to accelerate these procedures.

Singular Value Decomposition (SVD) [5] is one of the most important algorithms in numerical linear algebra because of its broad applications. In fact, MDDT and its successors also use SVD as a toolbox when they try to learn from the input data. So if we can accelerate SVD, the overall performance will be promoted. There are several methods to compute SVD [6]. The most popular one is based on iterative QR algorithm, which reduces the input matrix to a bidiagonal form first before further decomposition. This reduction step is called Householder bidiagonalization, and it could be very computing-intensive. According to some references [7], the Householder bidiagolization will consume over 50 percent of the overall execution time of SVD computation.

Luckily, there exist large quantities of data-level parallelism in this procedure, which is well suited for Single Instruction Multiple Data (SIMD) acceleration. It's quite common for today's CPUs to include an SIMD instruction extension, such as Intel's MMX/SSE, and ARM's NEON [8] [9], etc. Therefore we decide to exploit the parallelism with ARM's NEON, which is a 128-bit SIMD architecture extension to ARM Cortex-A series processors, designed to provide better support for multimedia and signal processing on mobile devices. The reason why we choose NEON is because: 1) the demand of multimedia processing on mobile devices is growing, and ARM is the major instruction set architecture on mobile platforms currently; 2) matrices processed in video codecs are usually low dimensional macroblocks, exactly the main targets that NEON is designed to handle with. Our study tries to testify NEON's potential application in future computing-intensive video codecs.

The rest of this paper is organized as follows. In Section II and III, we introduce some basics of the algorithms of Householder bidiagonailization and NEON architecture briefly. In Sections IV, details about our implementations are discussed. We evaluate our implementations with experiments and give results in Section V and some concluding remarks are given in the final section.

II. Householder Bidiagonalization

Householder bidiagonalization reduces an $m \times n$ input matrix $\mathbf{A}$ into an upper bidiagonal form if $m \geq n$:

$$\mathbf{U}^T\mathbf{A}\mathbf{V} = \mathbf{B} = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ & \alpha_2 & \ddots & \\ & & \ddots & \beta_n \\ & & & \alpha_n \end{bmatrix}. \tag{1}$$

The reduction is realized by applying so-called Householder transformations on columns and rows alternatively.

A. Householder Transformation

Householder transformation is an $n \times n$ orthogonal matrix of the form

$$\mathbf{H} = \mathbf{I} - \frac{2}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T \tag{2}$$

where $\mathbf{I}$ is an identity matrix, and $\mathbf{v}$ is called a Householder vector.

Given an n-length vector $\mathbf{x}$, Algorithm 1 computes the corresponding Householder vector $\mathbf{v}$ and $\beta = 2/\mathbf{v}^T\mathbf{v}$ such

that $\mathbf{H} = \mathbf{I} - \beta\mathbf{v}\mathbf{v}^T$ and $\mathbf{H}\mathbf{x} = \|\mathbf{x}\|_2\mathbf{e}_1$, where $\|\mathbf{x}\|_2$ is the 2-norm of $\mathbf{x}$ and $\mathbf{e}_1$ is a unit vector like $(1, 0, \ldots, 0)^T$.

Algorithm 1 $[\mathbf{v}, \beta] = \mathbf{householder}(\mathbf{x})$

$n = \mathbf{length}(\mathbf{x})$
$\sigma = \mathbf{x}(2:n)^T\mathbf{x}(2:n)$
$\mathbf{v} = \begin{pmatrix}1\\ \mathbf{x}(2:n)\end{pmatrix}$
if $\sigma = 0$ **then**
  $\beta = 0$
else
  $\mu = \sqrt{\mathbf{x}(1)^2 + \sigma}$
  if $\mathbf{x}(1) \leq 0$ **then**
    $\mathbf{v}(1) = \mathbf{x}(1) - \mu$
  else
    $\mathbf{v}(1) = -\sigma/(\mathbf{x}(1) + \mu)$
  end if
  $\beta = 2\mathbf{v}(1)^2/(\sigma + \mathbf{v}(1)^2)$
  $\mathbf{v} = \mathbf{v}/\mathbf{v}(1)$
end if

B. Householder Bidiagonalization

Now that we can eliminate elements except the 1st one in a given vector with Algorithm 1, we can do the same to any specified row or column of $\mathbf{A}$, just by picking up the proper parts of the vector and computing the corresponding $\mathbf{v}$, $\mathbf{H}$ in (2). That's actually the basic principle of Householder bidiagonalization. Algorithm 2 shows how it works:

Algorithm 2 Householder Bidiagonalization

for $j = 1 : n$ **do**
  $[\mathbf{v}, \beta] = \mathbf{householder}(\mathbf{A}(j:m, j))$
  $\mathbf{A}(j:m, j:n) = (\mathbf{I}_{m-j+1} - \beta\mathbf{v}\mathbf{v}^T)\mathbf{A}(j:m, j:n)$
  if $j \leq n-2$ **then**
    $[\mathbf{v}, \beta] = \mathbf{householder}(\mathbf{A}(j, j+1:n)^T)$
    $\mathbf{A}(j:m, j+1:n) = \mathbf{A}(j:m, j+1:n)(\mathbf{I}_{n-j} - \beta\mathbf{v}\mathbf{v}^T)$
  end if
end for

III. ARM NEON Technology

SIMD computing is a classical way to exploit the data-level parallelism in programs. With one single instruction and multiple pieces of data loaded up, the same operation can be finished over all the data simultaneously, saving both time and energy. NEON is a 128-bit SIMD engine introduced in ARM Cortex-A series processors, designed for better multimedia support on mobile platforms. It has 32 vector registers of 64 bits that can also be viewed as 16 vector registers of 128 bits. This dual views enable efficient handling of data and minimizing access to memory, especially when taking care of leftover elements. Multiple data types are supported by NEON, including signed/unsigned 8-bit to 64-bit integer and 32-bit single-precision floating point. That means a maximum of 4 single-precision numbers can be processed by one single NEON instruction. 16-bit half-precision floating point is only used for a storage format and no arithmetic operations supported at this time.

To benefit from NEON features, there exist four levels of ways. The highest and the easiest level is to use libraries optimized for NEON. The second is to rely on compilers to perform automatic vectorization on C or C++ source code. This looks convenient, but the effect is always limited because of challenges for compilers have to "understand" the code first and make sure automatic vectorization will not introduce mistakes.

The next two levels are the most effective and commonly adopted ways to use NEON. The higher one is NEON intrinsics, which include definitions of vector data types and intrinsic functions as an extension to standard C library. Intrinsic function calls are replaced by a sequence of NEON instructions during compilation, which enables programmers to control low-level instructions indirectly. Furthermore, compilers can help with register allocation and other optimizations. However, this method still brings in overheads, comparing to the lowest and the hardest level – writing assembly code directly. In our case, we implement the Householder bidiagonalization with both intrinsics and inline assembly and compare the performance in quantitive way. Our experiments show that inline assembly version is 1.26 times more efficient.

IV. Implementation

Discussion about our implementation includes two parts: 1) basic implementation; 2) handling with leftovers.

A. Basic Implementation

According to Algorithm 2, the bidiagonal reduction can be divided into several basic subroutines, from which we should find data-level parallelism to exploit:

1) the householder function to calculate Householder vector $\mathbf{v}$ and the corresponding β;
2) vector-matrix and scalar-vector multiplications to calculate $\mathbf{u}^T = \beta\mathbf{v}^T\mathbf{A}(j:m, j:n)$;
3) matrix-vector and scalar-vector multiplications to calculate $\mathbf{u} = \beta\mathbf{A}(j:m, j+1:n)\mathbf{v}$;
4) outer-product update to calculate $\mathbf{A}(j:m, j:n) - \mathbf{v}\cdot\mathbf{u}^T$ or $\mathbf{A}(j:m, j+1:n) - \mathbf{u}\cdot\mathbf{v}^T$.

For 1), we notice that in Algorithm 1 there exist two simple loops that we can make use of: one is to calculate σ by adding all the squares of the elements in $\mathbf{x}$ starting from $\mathbf{x}(2)$, and the other is to calculate $\mathbf{v} = \mathbf{v}/\mathbf{v}(1)$. The former can benefit from NEON's vector multiply-accumulate instruction, which accumulates the products of corresponding elements of two vectors into a destination vector. With this instruction, we can calculate a vector of partial sums of products during the loop and add them together finally. Currently NEON has no single instruction to get the sum of all elements in a vector. The only option is to use a sequence of vector pairwise-add instructions, which can add adjacent pairs of elements in a vector. So if the SIMD width is n, it takes $\lceil \lg n \rceil$ of these pairwise-add instructions to get the overall sum of these elements. This pairwise-add way is a little low efficient, because each

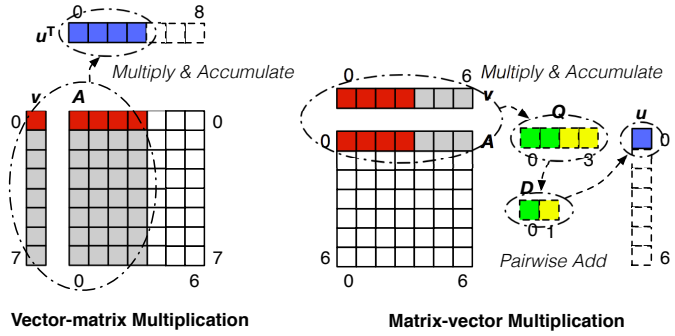

Fig. 1. Vector-matrix and matrix-vector multiplication based on NEON

time the pairwise-add instruction are executed, the number of meaningful values in the vector is halved, resulting in more wasteful operations on meaningless values.

NEON has no vector division instructions, but offers the instruction to calculate estimates of reciprocals instead. So the latter loop in 1) can simply reduce the iteration times through vector multiplication instruction.

For 2) and 3), conditions are a little more complex because of the existence of multi-level loops, which, however, also gives us more choices to maximize the utilization of NEON. Our solution is shown as Fig. 1. In typical vector-matrix multiplication, $u_j = \sum_{i=0}^{m-1} v_i a_{ij},\ j = 0, 1, \ldots, n-1$. If we calculate u_j one by one, then each process has the same structure as the former loop in 1), which needs vector multiply-accumulate instructions and additional low efficient vector pairwise-add instructions. But if we calculate multiple elements of adjacent columns at the same time, we don't need pairwise-add instructions any more, because each lane of the vector is independent and belongs to different u_j calculations. Only multiply-accumulate instructions are used and all the operations are necessary for the final results.

NEON supports vector load/store instructions that can specify multiple continuous elements in memory. For low dimensional matrices in our case, data are totally contained in the cache, so the vector load/store will run faster than multiple normal load/store. Another advantage of the multi-column method is the high utilization of this feature because multi-dimensional arrays in C/C++ are stored in row-major order and elements of adjacent columns are continuous.

At the same time, row-major order also means that elements of adjacent rows are not continuous in memory. So for matrix-vector multiplication in 3), where typically $u_i = \sum_{j=0}^{n-1} a_{ij} v_j$, $i = 0, 1, \ldots, m-1$, if we want to calculate multiple u_i simutaneously, we'll have to access the elements of different rows one at a time to fill them into one single vector register. Moreover, the step and direction of memory accesses are not as regular as those in 2): the base address pointer has to move back and forth, which also introduce overheads to calculate the varying steps.

As a result, our choice is still one single row at a time, showed as Fig. 1, which may need additional pairwise-add instructions but accelerate the procedure anyway by the usage of vector load/store instructions.

For 4), vectorization is easier and more obvious, for there's no need to accumulate any partial results. All we have to do is just going through the elements of $\mathbf{A}$ in row-major order and updating them with the outer product of the two vectors. The vector subtraction and multiplication instructions will help us to do this in parallel.

B. Handling with Leftovers

From Algorithm 2 we can also see that the dimension of vectors and matrices keeps reducing during different iterations of Householder bidiagonalization. So when it is not a multiple of the SIMD width, there will be leftover elements that cannot be filled in the SIMD width. It's not possible to load these elements with one single NEON instruction because the number is varying at runtime. Unlike memory, the position of data in a vector register cannot be specified by a variable. So leftovers cannot be loaded up by a loop, either.

The most direct solution is to deal with them one at a time as scalar processing. Simplicity is its advantage, while NEON is under-utilized and the overall performance is held back. Another solution is to set specialized code blocks corresponding to each number of leftovers. Because the number is definite and constant in each code block, it's possible to determine how many instructions are needed and which position to load each piece of data into. The dual views of NEON register file are helpful here because we're able to access memory with finer granularity. Then with one single SIMD instruction, all the leftovers are processed in one iteration just like normal. Our experiments shows that this solution will enhance the overall performance by 5%, especially when the dimension of matrix is low.

V. Evaluation

We evaluate our different implementations of Householder bidiagonalization on NUFRONT's NuSmart 2816 based development board, which has a dual-core ARM Cortex-A9 processor running at 800 MHz. GCC 4.4.5 on Ubuntu 10.10 is used to compile and optimize our plain C, NEON intrinsics, and inline assembly codes. We also include a CLAPACK version of Householder bidiagonalization as reference in our experiments. CLAPACK is a C interface of the famous FORTRAN77 library LAPACK [10] that implements major numerical linear algebra subroutines.

The dimensions of matrices we choose in our experiments are 4×4, 8×8, 16×16, 32×32, and 64×64, corresponding to the macroblocks (or submacroblocks) in current and ongoing video coding standards. In the following subsections, we will present our performance results over these 5 dimensions.

A. Overall Performance

The overall performance of different implementations is presented as Fig. 2. It shows that our utilization of NEON can accelerate Householder bidiagonalization as expected. The inline assembly way starts to take effect when the matrix is larger than 4×4, and the intrinsics way does so when the matrix is larger than 8×8. The speedups keep growing with

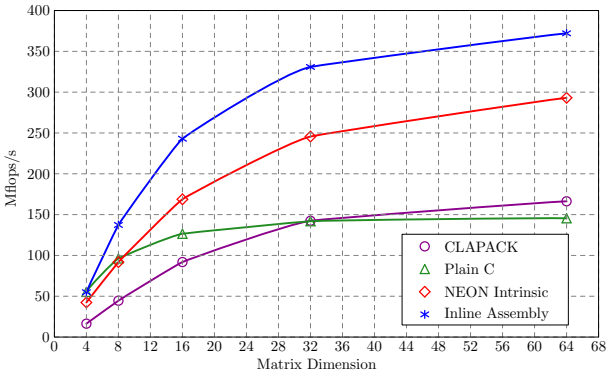

Fig. 2. Speeds of different implementations

the dimension and have a good scalability comparing to the plain C and CLAPACK version.

From Fig. 2 we can also see the difference in efficiency between intrinsics and inline assembly. Although the compiler can do more optimization such as register allocation for intrinsics, the translation of intrinsic functions to the low level still introduces redundant instructions, making the inline assembly codes take the lead over all tests of 5 dimensions.

B. Performance of Different Subroutines

Fig. 3 gives us detailed information about the performance of different subroutines. For each dimension, the subroutines we vectorize are profiled separately and the execution time is divided by the overall execution time of the plain C version, by which we can not only compare the performance, but also gain the proportions of these subroutines.

According to Fig. 3, our vectorization of the two simple loops in the householder function shows no observable promotion and even brings in tiny negative effects when dimension is low, where the limited parallelism cannot counteract the overheads of intrinsics or inline assembly.

However, for the other 3 subroutines, our vectorization is successful, especially for the vector-matrix and matrix-vector multiplication, both of which enhance the performance instantly over all the 5 dimensions and keep enlarging their speedups as the dimension grows. For 64×64 matrices, the

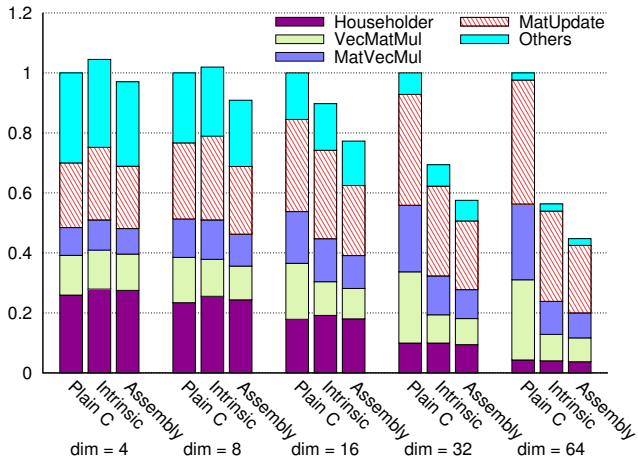

Fig. 3. Speedups of different subroutines

maximal speedups of 3.36 for vector-matrix and 3.04 for matrix-vector multiplication are reached. These values are close to the theoretically optimal taking into consideration the SIMD width and the existence of leftovers.

The vectorization of the outer-product update subroutine does work, but not so notably as the two above. The maximal speedup of 1.84 is achieved when processing 64×64 matrices. The reason for the stagnancy is because this subroutine is memory-bound relatively. Although the numbers of arithmetic operations are almost the same, vector-matrix or matrix-vector multiplication only stores n elements, while outer-product update will stores n^2 elements, assuming that the dimension is n. This higher order of memory accesses conceal NEON's acceleration on arithmetic operations and hold back the overall performance.

Moreover, since the complexity of arithmetic operations is $\mathrm{O}(n^2)$, the same order as memory accesses, the ratio of memory accesses will not drop like vector-matrix and matrix-vector multiplication, which can explain why the overall performance in Fig. 2 improves slower when the dimension is growing.

VI. Conclusions

Our study shows that Householder bidiagonalization can be accelerated by ARM NEON Technology. Operations on macroblocks larger than 8×8 can benefit from NEON inline assembly. For smaller macroblocks, the speedup are limited by the higher proportion of leftovers, while for larger ones, the bound of memory accesses is the bottleneck.

Acknowledgment

This work is funded by TNList cross-discipline foundation, the Nature Science Foundation of China under Grant No. 60833004, 60970002, and the National 863 High-Tech Programs of China (No.2012AA010905, 2012AA012609).

References

[1] G.J. Sullivan and J.R. Ohm. Recent developments in standardization of high efficiency video coding (hevc). In *Proc. SPIE*, volume 7798, pages 7798–30, 2010.

[2] I.E.G. Richardson. *H. 264 and MPEG-4 video compression*, volume 20. Wiley Online Library, 2003.

[3] Y. Ye and M. Karczewicz. Improved h. 264 intra coding based on bi-directional intra prediction, directional transform, and adaptive coefficient scanning. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pages 2116–2119. IEEE, 2008.

[4] O.G. Sezer, R. Cohen, and A. Vetro. Robust learning of 2-d separable transforms for next-generation video coding. In *Data Compression Conference (DCC), 2011*, pages 63–72. IEEE, 2011.

[5] G.H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, 1970.

[6] G.H. Golub and C.F. Van Loan. *Matrix computations*, volume 3. Johns Hopkins Univ Pr, 1996.

[7] H. Ltaief, P. Luszczek, and J. Dongarra. High performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures. *Submitted to ACM Transactions on Mathematical Software*, 2011.

[8] *ARM Architecture Reference Manual*, armv7-a and armv7-r edition, 2010.

[9] *Cortex-A9 NEON Media Processing Engine Technical Reference Manual*, 2011.

[10] E. Anderson, Z. Bai, and C. Bischof. *LAPACK Users' guide*, volume 9. Society for Industrial Mathematics, 1999.