An Exploration of Using the Intel AVX2 Gather Load Instructions for Vectorised Image Processing

Michael J. Cree
School of Engineering
University of Waikato
Hamilton, New Zealand
michael.cree@waikato.ac.nz

***Abstract*—Processing image data with single-instruction multiple-data (SIMD) CPU instructions provides a means of vectorising, thus speeding up execution, of standard image processing operators. SIMD register loads normally load from consecutive locations in memory, that is, consecutive pixels in a row of the image. For some algorithms, however, data dependencies between pixels along rows render SIMD vectorisation useless. If one could efficiently load pixels from columns of images this problem would be fixed.**

The Intel AVX2 CPU extension introduces an instruction for the gather loading of data from multiple memory locations into a single CPU SIMD register. We explore using these instructions for column loads of image data in two common image operations, transposing images and mean filtering, to test 1) whether they provide useful speed-ups when other vectorised approaches exist (and find that they do not), and 2) whether they provide means of implementing operations that otherwise would be difficult or extremely inefficient to achieve without a column load (they can provide speed-ups over scalar code).

***Index Terms*—SIMD, vectorisation, image processing**

I. Introduction

Simultaneous instruction multiple data (SIMD) CPU instructions process multiple data in one CPU instruction and are a means of processing simultaneous calculations on many data. They are very useful for accelerating image processing operators by parallelising (or vectorising) computation [1].

SIMD CPU load instructions read multiple data from consecutive locations in memory into a CPU register. When operating on an image that is stored in a C compliant array format the SIMD load instruction efficiently loads the CPU register with consecutive pixels from a row of the image. In contrast, directly loading a register with consecutive pixels from columns of the image is usually not possible with a single (or even a few) CPU instructions and can be extremely inefficient to perform.

Many standard image processing operations are decomposable into a sequence of simpler operations that are faster to execute. For example, 2D-linear filtering operations can sometimes be decomposed [2] into processing along rows of the image followed by processing along columns of the image, likewise some useful morphology structuring elements can be decomposed in a similar manner [3]–[5]. In a scalar implementation (i.e. no use of SIMD instructions) this decomposition of a 2D operation into two 1D operations can provide very significant speed-ups in execution. Vectorising the sub-filter operations with SIMD vectorisation can provide further substantial speed ups, provided that a means to vectorise all sub-filtering operations exists [6].

That SIMD CPU registers can only be efficiently loaded from rows of the image often means that vectorising processing in one direction of the image is only possible, particularly if there are data-dependencies between pixels such that subsequent pixels can only be calculated once the result on a prior pixel is known. For example, the well-known mean filter can be decomposed into two 1-D filtering operations, one operating along columns and one along rows. An efficient algorithm (that is independent of the size of the kernel) results by calculating the kernel for a pixel from the kernel of the preceding pixel, but this introduces a data dependency from a pixel to its neighbour in the direction of calculation. Processing down columns with SIMD is efficient because the pixel data dependency is between pixels in different CPU registers, but processing along rows with SIMD is inefficient because the data dependency is between pixels in the same CPU register. To calculate the result on all pixels in the register requires the result of processing another pixel in the register which breaks a necessary assumption for vectorising with SIMD instructions.

In the Haswell generation of Intel CPUs, Intel introduced a new CPU instruction that can load a SIMD register from multiple independent locations in memory, that is, each 32-bit or 64-bit element of the register can be loaded from random locations in memory and constituted into a single SIMD register. It appears that this instruction was primarily introduced to enable larger table lookups for evaluating standard mathematical functions [7], which then enables correct rounding to the least significant bit of the result to a greater portion of the domain in vectorised mathematical libraries (which is the state of art in scalar mathematical function evaluation).

To the machine vision scientist, this CPU instruction suggests another application: the loading of a SIMD register from the column of an image. We explore in this paper the use of the Intel gather load instruction to load image data from columns (see Fig. 1(a)) to implement transpose operators and to implement a 1D horizontal (i.e. along rows) mean filter that

978-1-7281-0125-5/18/$31.00 ©2018 Crown

is otherwise much more difficult to implement.

We explore image transposes because the loading of image pixels from a column of the image followed by a normal SIMD write along the row effects the transpose without any further calculation. Without a column load this operation is normally performed by loading the rows of the block, performing the transpose by permutation operations on the CPU registers, then writing out the rows of the block with the transposed data. We investigate on images with 32-bit pixels because that is the smallest sized element that can be loaded with the gather-load instruction, and ask whether a column load using a gather-load instruction is faster. We also investigate the transpose of images with 8-bit and 16-bit pixels using the gather-load approach because other methods to implement the transpose result in excessive spill of CPU registers to the stack, which negates the advantage gained by vectorisation.

We then explore the use of the column load in implementing the horizontal 1D mean filter on 32-bit data. We do this not only because the column load has a minimum resolution of loading 32-bit data, but because on an Intel processor with both the streaming SIMD extension (SSE) and advanced vector extensions (AVX, AVX2) we can load a 'half-vector' which provides an alternative means of efficiently loading the same basic block of data with half-length row loads with a transpose of the data in the CPU registers. This gives a method to perform the loads by column, or by rows with transposition, and compare which is the best approach.

II. THEORY

Recent Intel CPUs contain both the SSE and AVX/AVX2 extensions. The SSE extensions provides for SIMD calculations on CPU registers of 128-bit width. SSE can perform calculations, as examples, on 16 bytes simultaneously, or on four 32-bit integers simultaneously. The AVX extension lengthened the register to 256-bit width thus can simultaneously perform calculations on 32 bytes, or eight 32-bit integers, in CPU registers. The AVX2 extension provided more CPU instructions to the AVX registers including the gather load instruction.

Note that there is *no* accompanying scatter write instruction that does the equivalent of the gather read for writing data. That is quite a limitation that can be argued to largely negate the advantage of a gather read for reading in columns, because there is no means to write out data to the same columns! We work around this problem where necessary by use of SSE row writes: by loading four neighbouring columns of 32-bit integers into four AVX2 registers we load a 4×8 block of pixels[1] which can be transposed with CPU instructions operating on the CPU registers to give eight SSE registers that can be used to perform row writes (see Fig. 1(b)).

A. Transpose of images with 32-bit pixels

Consider the transpose of an image of size $M \times N$ 32-bit pixels. To improve cache locality we divide the image into tiles of size $T \times T$ pixels where T is chosen to ensure the tile

[1]We use the $x \times y$ convention when specifying image and block sizes.

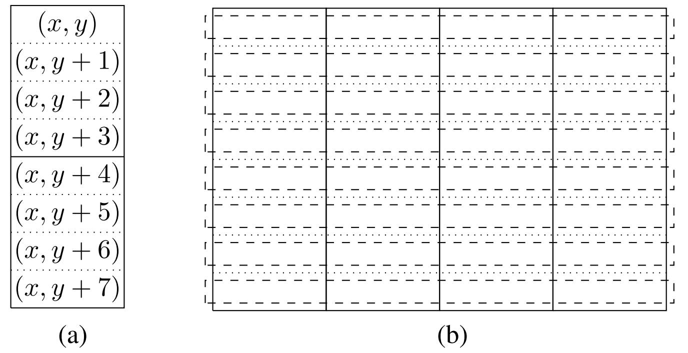

Figure 1. Illustration of column loads: (a) A single column load of 32-bit data from position (x, y) of the image into an AVX2 register formed from two gather load CPU instructions. (b) Four adjacent column loads used to load a 4×8 block of 32-bit pixels into four AVX2 registers that can also be written/read to/from memory with eight SSE writes/reads (dashed boxes along rows).

can easily fit into cache. A tile read from position (x, y) in the source image is written in transposed form to location (y, x) in the destination image. The image is processed in tiles by sequentially processing each tile in row major order from the source image.

We implement a number of versions of the transpose:

1) Scalar implementation in which the tile transpose operator reads and writes individuals pixels (32-bit data) one at a time. In the figures this is labelled 'Scalar-Transpose'.
2) A vectorised implementation in which the gather load instruction is used to read into four AVX2 vectors being the columns of a 4×8 block of pixels. These are written out directly as AVX2 vector rows writes, writing the 8×4 block of pixels to the destination image. In the figures this is labelled 'Transpose: Gather'.
3) A vectorised implementation in which SSE row reads are used to read from consecutive rows a 4×8 block of pixels. These are permutated in CPU registers to produce four AVX2 registers which are the columns of the block of pixels. Note that this operation produces exactly the same result in the four AVX2 registers as the Tranpose:Gather operation described above. The transpose is therefore effected in the same manner: simply writing the AVX2 registers out as rows to the required location in the destination image. In the figures this is labelled 'Transpose: SSErow'.
4) A typical vectorised implementation which loads eight AVX2 registers from consecutive rows of the same x-location giving a block of 8×8 pixels that is transposed by permutation CPU instructions operating on the registers, and then written out as AVX2 row writes to the required locations in the destination images. In the figures this is labelled 'Vector-Transpose'.

B. Transpose of Images with 16-bit and 8-bit pixels

An AVX2 CPU register contains sixteen 16-bit elements and thirty-two 8-bit elements. Note that the transpose of

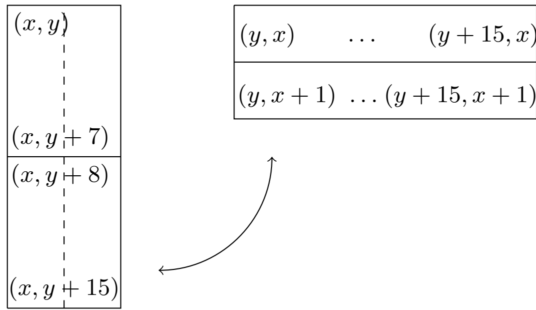

Figure 2. Two column loads performed (left) to load two AVX2 registers with a 2×16 block of 16-bit pixels. With minimal permutation these can be rearranged into two AVX2 registers with a row of sixteen 16-bit pixels each (right) that can be written out to the destination image with normal row writes.

SIMD registers containing N elements requires greater than N registers to perform the transpose without spilling CPU registers to the stack. The Intel CPU has 16 AVX2 CPU registers thus to transpose a 16×16 block of 16-bit pixels will result in register spill to the stack and is likely to be inefficient. The transpose of a 32×32 block of 8-bit pixels involves repeated substantial spill of CPU registers to the stack. These spills involve extra memory reads and writes which negate the advantage of working with SIMD registers.

But note if we use the gather load instruction to load a column of 1×16 32-bit data into two AVX2 registers, the first register being the top half of the column and the second register the bottom half of the column, we get a block of 2×16 16-bit data (see Fig. 2). With a simplified transpose operation using only six permutation instructions on CPU registers those two AVX2 registers can be rearranged to be the transposed 16×2 data in two AVX2 registers one for each row. Two straightforward row writes of the AVX2 registers to the correct location in the destination effects the transpose of images of 16-bit data.

Likewise if we use four column loads to load a column of 1×32 32-bit data into four AVX2 registers, each subsequent one continuing the load from the rows below the last one, we get a block of 4×32 8-bit data. With eighteen permutation operations on the four AVX2 registers this can be transposed to a 32×4 block of data, and with four AVX2 register writes to consecutive rows the transpose of the block of pixels to the destination image is effected.

Note that a column load enables us to transpose a non-square block of image pixels without risk of register spill to the stack even though the register may hold many image pixels. The above two scenarios therefore presents the opportunity to implement image operations that are otherwise very inefficient when implemented without column loads.

C. Horizontal 1-d mean filter on unsigned 32-bit pixels

The horizontal 1-D mean filter with a kernel of k-pixels can be calculated by summing up the k-pixels contributing to the kernel at the start of the row (i.e. the first pixel in the row). The accumulator is divided by k and the result is stored back to the pixel. For 'in-place' image operation the original pixel is saved into a temporary row buffer as it is needed again. Shifting to the next pixel the new pixel added to the right-edge of the kernel is added in to the accumulator and the old pixel lost from the left-edge of the kernel (and previously saved into temporary row buffer) is subtracted off. The accumulator now has the new value needed for calculating the result with only an addition and subtraction to the previous kernel's accumulator. This is the case no matter the value of k and presents a very efficient means of calculating the horizontal 1-D mean filter with run-time that is largely independent of the size of the kernel. Provided there is no under or overflow of the accumulator this scheme is exact for integer pixel data, but can suffer a serious loss of significance for floating-point data if the image data consists of large variations (orders of magnitude) in value.

The horizontal 1-D mean filter just described is difficult to implement in SIMD for general k because there is a data dependency between pixels along rows. (Relatively straightforward solutions exist if k is very small, namely 2 or 3.) The usual load of SIMD data from a row of the image results in the data dependency being intra-register. Breaking such a dependency usually takes as many operations as there are elements in the register (when k is large) thus negating any advantage of vectorising. But if one could load from columns of images into the CPU SIMD register then the data dependency is transformed from intra- to inter-register, a condition needed for efficient vectorisation.

We use the AVX2 gather load instruction to load an AVX2 register worth of pixels from a column of the image. It is an advantage to load four such neighbouring columns of pixels (i.e. a 4×8 block of 32-bit pixels) for two reasons: 1) we can use SSE row writes to the write the resultant block back to the image in the absence of a column write, and 2) we can compare against a series of SSE row reads with a transpose to implement the same column load. (There is a third advantage: it is easier to port to other architectures that do not have gather load instructions.)

We should make comment on the calculation of the division of the kernel accumulator to effect the calculation of the mean. Because a constant divisor is used (the same divisor for every pixel) the integer reciprocal can be calculated once for the image then integer multiplication of the kernel accumulator by the reciprocal can be performed at each pixel. Integer multiplication on most architectures is much faster than integer division: this proves to be true on Intel CPUs even though there is an integer division CPU instruction.

In the scalar code we use the method of Robison [8] to calculate the integer reciprocal and the division by multiplication. This method exploits the fact that we have more than the 32-bits width of the pixel in a CPU register to do the calculation (indeed 64-bits are available). But this is not the case for the vectorised SIMD implementation: it is much more efficient in SIMD to calculate at the width of the SIMD vector elements (in our case 32-bits) but an unsigned integer division of N-bits by multiplication with the integer reciprocal can

Table I
SYSTEMS TESTS WERE PERFORMED ON. CACHE (L1, L2 AND L3) IS SPECIFIED AS SIZE/WAYS OF ASSOCIATIVITY.

System	CPU	Memory	L1	L2	L3
Haswell	i5-4590	8 GB	32kB/8	256kB/8	6MB/12
Haswell	i7-4790	16 GB	32kB/8	256kB/8	8MB/16
Haswell	E5-2640	128 GB	32kB/8	256kB/8	20MB/20
Coffee-Lake	i5-8600K	16 GB	32kB/8	256kB/4	9MB/12

require $N+1$ bits in the calculation to get correct results, thus we use the method of Granlund and Montgomery [9] which simulates a N+1-bit multiplication with only N-bit arithmetic. The reciprocal is calculated with scalar code as it is only done once for the image. The multiplication by the integer reciprocal is vectorised for efficient division at each pixel.

III. METHODOLOGY

We test on four Intel platforms: Intel Core i5, i7 and Xeon (Haswell generation CPUs) running Debian Linux 'Stretch' version 9.5 with code compiled with the GCC 6.3 compiler, and on an Intel Core i5 (Coffee Lake generation) running Ubuntu Linux 'Xenial Xerus' version 16.04.5 (LTS) with code compiled with the GCC 5.4 compiler. Further details on these platforms are in Table I.

Code was profiled with the Linux Performance Events subsystem to exploit hardware event counters in the CPU to measure elapsed time and count number of CPU cycles, instructions retired and cache misses incurred while the timed process is scheduled on the CPU. Images were initialised with uniformly distributed random data and the image processing operator was first run once to test for correct operation of the operator and prime the cache. This operation makes copies of original images to test that the operator does not modify the source image, and checks that the result image is correct. The operator then was executed a further ten times in succession solely to time the operation and count hardware events. Elapsed time, CPU cycles incurred, instructions retired and number of cache misses incurred on average for one execution of the operation were stored for further analysis.

Tests were performed on a square image for sizes ranging from 128×128 pixels (0.016 megapixel) up to 8192×8192 pixels (67 megapixel). For sizes below 16 megapixel the tests were re-run 50 times and for sizes above 16 megapixels the tests were re-run 30 times. Results for the fastest 25 execution times were used of the 50 (or 30) test runs to eliminate disproportionately large results due to latencies incurred by other processes running on the system. The mean value calculated for the chosen 25 test runs are used for analysis.

IV. RESULTS

We first report results (Fig. 3) for copying and transposing an image of 32-bit pixel data. The scalar implementation copies images with the C-standard memcpy() library routine applied per row of the image. The glibc implementation of memcpy() detects the CPU and may use CPU specific optimised code including copying via SIMD registers. Then we

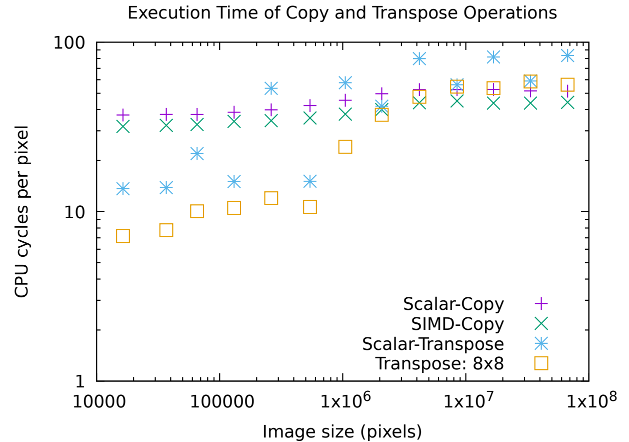

Figure 3. Execution time (in CPU cycles per pixel) of Scalar and SIMD implementations of image copy and transpose. The scalar copy operation may well be vectorised as it uses the standard memcpy() library function.

tested image copy with our own SIMD optimised copy routine (Fig. 3) which proved slightly faster than using memcpy(), probably because our routine could exploit the fact that every row of the image is allocated to ensure correct alignment for SIMD operation and has row padding to avoid any special casing code at the start and end of rows.

In Fig. 3 we also report on transposing the image with a pixel-by-pixel scalar implementation and by the 8×8 block transpose vector implementation that uses only typical SIMD row loads from memory. The image transpose for images smaller than about 1 megapixel proved faster than image copy on this system. Considering that there are two images (the original and the result copy/transpose) of 4 MB each, and that on this system (Haswell, i7) the L3 cache is 8 MB is suggestive that the transpose out-performs a straight image copy when the images can be held entirely in L3 cache. Tests on the other Haswell systems (not shown here) produced similar results with the merging of the various implemented operations to similar speeds occurring at about L3 cache size.

Now we consider the implementation of the transpose using the column load operation. The use of the column load eliminates the need for an 8×8 block transpose of the image data in SIMD registers. Timing results for the four implemented transpose operators (scalar and the three vectorised implementations) are shown in Fig. 4. The 'SIMD-Copy' and 'Transpose: 8×8' curves are the same as presented in Fig. 3. One can see that the column load (labelled 'Gather') did not perform as well as the baseline 'Transpose: 8×8' implementation or the SSErow variant, showing that the computer architecture is (not unsurprisingly) much more efficient at loading row data.

To better see these differences in speed we plot the speed-up of the vectorised copy versus scalar copy, and the speed-up of the three vectorised transpose operations versus the scalar operation in Fig. 5. It is clear that the SIMD column load based transpose operator ('Transpose: Gather') is actually slower in many cases than the scalar implementation! The tests on the

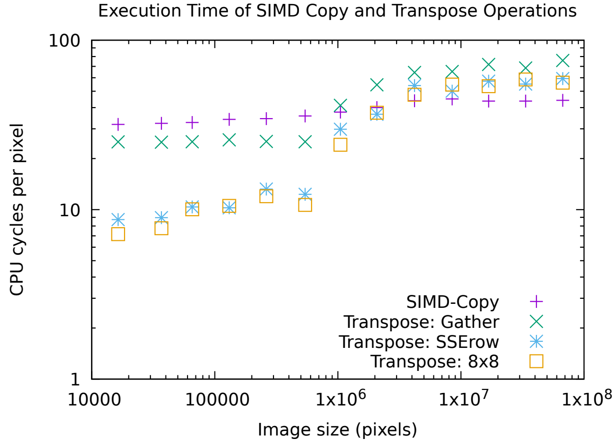

Figure 4. Execution time (in CPU cycles per pixel) of SIMD implementations, including the column load implementations, of image transpose.

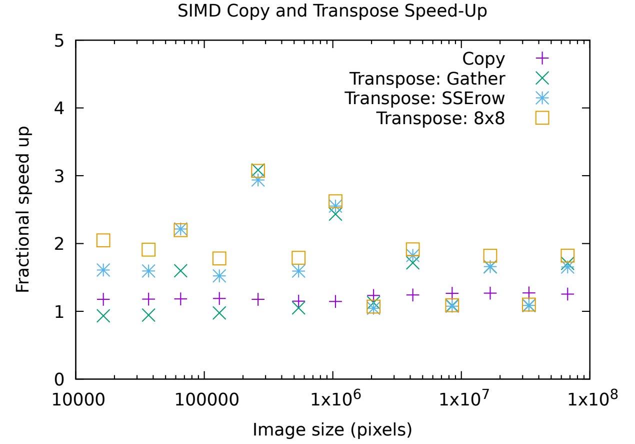

Figure 6. Speed-up in execution of image copy versus the scalar copy implementation, and of image transposes versus the scalar transpose implementation on the Coffee Lake system.

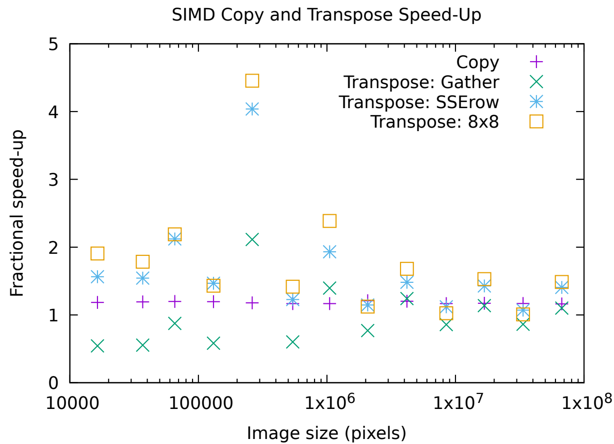

Figure 5. Speed-up in execution of image copy versus the scalar copy implementation, and of image transposes versus the scalar transpose implementation.

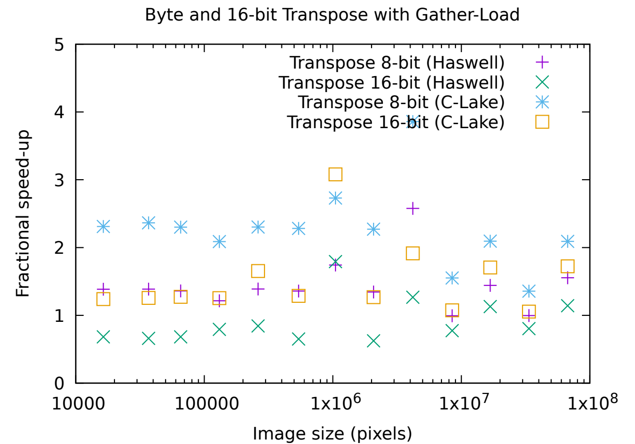

Figure 7. Speed-up of the vectorised 8-bit and 16-bit image transpose when implemented with column load operations on the Haswell system and the Coffee-Lake (C-Lake) system compared against their respective scalar implementations.

Coffee Lake system (Fig. 6) showed an improved performance on the gather load method, but they are still considerably slower than loading data by rows.

The speed-up in execution over scalar code for transposes of 8-bit and 16-bit images using the column load approach described in Sect. II-B is shown in Fig. 7. On the Haswell systems the vectorised transpose of 16-bit images is slower than the scalar implementation, and the vectorised transpose of 8-bit images provides only a small speed advantage. The Coffee-Lake system shows better performance with now a small advantage for transposing 16-bit images and a more significant and worthwhile advantage for transposing 8-bit images.

We now turn attention to the mean filtering. The horizontal 1-D mean filter algorithms (scalar and the two vectorised implementations) are shown in Fig. 8 for the Haswell i7 system. The improvement in performance over equivalent scalar implementations is better seen in Fig. 9. The column load (gather) gives a small, nevertheless disappointing, improvement in speed, and the use SSE row loads gives a better speed-up of just over 2 times over the scalar implementation. For comparison the speed-up of the vectorised vertical 1-D mean filter is also shown, and speed-ups of between 4.5 and 6 times the scalar implementation is evident. This illustrates the penalty of loading data from columns compared to loading data from rows.

V. DISCUSSION

The results show that the slower speed of the gather load instruction has limited utility in loading image data from columns of the image. It is always faster, when it is possible and there is no register spill, to reorganise the algorithm to read from rows and transpose within CPU registers to give the columns in the CPU registers, and transpose again before writing data out as rows. It seems that Intel has made

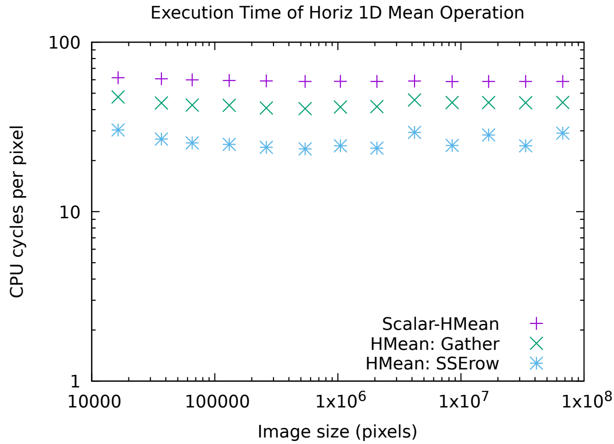

Figure 8. Execution time (in CPU cycles per pixel) of the three implementations of the Horizontal Mean 1-D filter.

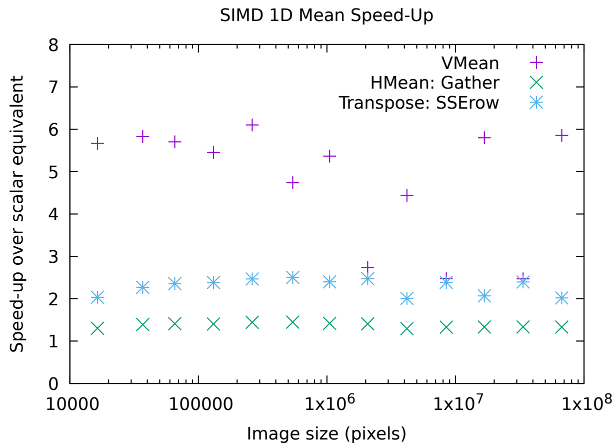

Figure 9. Speed-up in execution of the two SIMD implementations of the horizontal mean 1-D filter ('HMean') with the speed-up of the vectorised vertical mean 1-D filter ('VMean') for comparison.

some attempt to speed up the gather load instructions in the newest generation of Intel CPUs (as evidenced by our results on the Coffee Lake generation CPU) and it appears that Intel acknowledge that the gather load instructions are not as successful as one might have anticipated, as they have introduced extra permutation instructions in the AVX-512 extension to provide larger table lookups by operating on CPU registers only [7], [10]. They state that using these new permutation instructions in mathematical functions, although it incurs a larger polynomial to interpolate between tabulated points, can be as much as 2× faster than using the gather load instruction [7]. Note, however, the AVX-512 extension is only available on recent high-end server CPUs and it does not solve the column load problem.

ARM is also introducing gather-load and scatter-write instructions to its CPUs in its new Scalable Vector Extension [11]. They note that these instructions enable vectorisation of loops accessing discontiguous data, but also note that these instructions in their tests did not scale well with increasing size of the SIMD register, and blamed the compiler for poor scheduling of instructions. To help to break the dependencies of data intra-register ARM also propose 'horizontal operations' that operate on data within a register.

Despite these limitations we find that the gather load instruction does have utility when reading from rows of the image into CPU registers and transposing incurs excessive spill of registers to the stack, as occurs, for example, when working with 8-bit data in the AVX2 256-bit registers. We found a 2× speed-up in operation over scalar code when transposing images with 8-bit pixels when using the gather load instruction on the newest generation Intel CPUs.

VI. Conclusion

Mixed results were obtained with the gather load instruction for loading CPU SIMD registers from the columns of images. We found that column loads of data never outperformed loading data from rows and transposing entirely within CPU registers to get the columns. While the more recent Coffee Lake CPUs have a faster gather load instruction this conclusion nevertheless remains unchanged. The only situation where loading data from columns with the gather load gives an improvement over other methods is when the only alternative is transposing data in CPU registers and that would lead to excessive register spill to the stack. Such a situation occurs when transposing images consisting of byte (8-bit) pixels. On Haswell systems only a small (approximately 1.3×) speed-up was achieved using the gather-load instruction, but this increased to a more useful 2× speed-up on Coffee Lake systems.

References

[1] R. Cypher and J. L. C. Sanz, "SIMD architectures and algorithms for image processing and computer vision," *IEEE Trans. Acoust. Speech Sig. Proc.*, vol. 37, no. 12, pp. 2158–2174, 1989.

[2] C.-S. Bouganis, G. A. Constantinides, and P. Y. K. Cheung, "A novel 2-d filter design methodology for heterogeneous devices," in *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, Los Alamitos, 2005, pp. 13–22.

[3] X. Zhuang and R. Haralick, "Morphological structuring element decomposition," *Comput. Vis. Graphics Image Process.*, vol. 35, pp. 370–382, 1986.

[4] X. Zhuang, "Decomposition of morphological structuring elements," *J. Math. Imaging Vis.*, vol. 4, pp. 5–18, 1994.

[5] E. Urbach and M. Willkinson, "Efficient 2-d grayscale morphological transformations with arbitrary flat structuring elements," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 12, pp. 1606–1617, 2008.

[6] M. J. Cree, "Vectorised SIMD implementations of morphology algorithms," in *International Conference on Image and Vision Computing New Zealand (IVCNZ2015)*, Auckland, New Zealand, 2015.

[7] M. Cornea, "Intel AVX-512 instructions and their use in the implementation of math functions," Intel Corporation, 2015.

[8] A. D. Robison, "N-bit unsigned division via n-bit multiply-add," in *IEEE Symposium on Computer Arithmetic*, 2005, pp. 131–139.

[9] T. Granlund and P. L. Montgomery, "Division by invariant integers using multiplication," in *Proceedings ACM SIGPLAN 1994 Conference on Programming Language Design and Implmentation*, 1994, pp. 61–72.

[10] C. S. Anderson, J. Zhang, and M. Cornea, "Enhanced vector math support on the Intel AVX-512 architecture," in *25th IEEE Symposium on Computer Arithmetic (ARITH 25)*, Amherst, MA, 2018, pp. 116–120.

[11] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, and et al., "The ARM scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2005.