Vectorised SIMD Implementations of Morphology Algorithms

Michael J. Cree
School of Engineering
University of Waikato
Hamilton 3240
New Zealand
email: cree@waikato.ac.nz

***Abstract*—We explore vectorised implementations, exploiting single instruction multiple data (SIMD) CPU instructions on commonly used architectures, of three efficient algorithms for morphological dilation and erosion. We discuss issues specific to SIMD implementation and describe how they guide algorithm choice. We compare our implementations to a commonly used opensource SIMD accelerated machine vision library and find orders of magnitude speed-ups can be achieved for erosions using two-dimensional structuring elements.**

I. Introduction

Mathematical morphology is a core part of image processing, with most morphological operators defined in terms of the two primitive operations, the dilation and the erosion [1]. The dilation and erosion can be specified in a straightforward manner however naïve implementation of the formula describing the operators leads to an efficient algorithm, indeed, one that grows as $O(k)$ for processing each pixel in an image when using a linear structuring element of length k elements and $O(k^2)$ for a two-dimensional structuring element. Larger structuring element sizes (particularly if using a two-dimensional structuring element) can quickly become computationally inefficient.

There are three ways to deal to the computational load. Firstly, decompose the structuring element into sequential processing of smaller and, ideally, of fewer dimensions, structuring elements. For example, morphology with a square structuring element can be rendered $O(k)$ by processing with two structuring elements: a horizontal line followed by a vertical line. Each step has $O(k)$ operations thus the result is $O(k)$.

Secondly, by implementing a cleverer and more efficient algorithm for morphology. For example, line structuring elements can be broken down into a sequence of approximately $\log k$ structuring elements consisting of two points appropriately spaced apart. The processing time is $O(\log k)$ rather than $O(k)$, a significant speed-up for larger line structuring elements. But even better, an algorithm exists since the mid 1990s for the processing of a line structuring element in $O(1)$ time with two memory buffers equivalent to two lines of the image [2]–[4]. We refer to this as the HGW algorithm following the nomenclature of Gil and Kimmel [5]. An image, thus, can be dilated or eroded with a square structuring element and a small amount of extra memory in time almost nearly completely independent of the structuring element size.

Other algorithms exist for the acceleration of general arbitrary structuring elements. An example is the decomposition of the structuring element into a summation of horizontal (or vertical) chords and efficiently computing the contribution of the various chords by use of the line recursion algorithm and avoiding recomputation of results for subsequent chords of the same length [6]. Van Droogenbroeck and Talbot also propose [7] an efficient algorithm for arbitrary structuring elements that works by updating a histogram while shifting the structuring element from pixel to pixel. Because the histogram size is determined by the number of grey-scale levels of the pixel data type this method is typically only suitable for pixel data types of 16-bits or fewer.

The third method to improve computational efficiency is to exploit the simultaneous instruction multiple data (SIMD) CPU instructions for vectorised processing of multimedia like data that are present on modern CPUs. Indeed, OpenCV[1], a popular opensource C++ library for computer vision has for over a decade provided SIMD accelerated machine vision operations, at least on Intel hardware [8], [9]. As Urbach and Wilkinson noted in 2008 [6] the OpenCV implementation, while it exploits Intel SIMD CPU instructions, it is rather slow for larger structuring element sizes as it implements the inefficient direct algorithm!

We explore implementing standard algorithms for accelerating erosions and dilations with SIMD techniques on the Intel architecture.

II. Erosion Algorithms Under Consideration

Only flat structuring elements used for erosions and dilations on two-dimensional images are under consideration herein. We assume all images f have their origin at the top-left. The morphological erosion is defined by

$$\mathcal{B}f = (f \ominus B)(x,y) = \min_{(u,v)\in B} f(x+u, y+v), \qquad (1)$$

for structuring element B. The dilation $f \oplus B$ is similarly defined but with the maximum rather than the minimum. A

[1] http://opencv.org

978-1-5090-0357-0/15/$31.00

direct implementation of (1) results in $O(k^2)$ operations per image pixel if k characterises the size of B in each dimension.

An erosion $\mathcal{R}_{UV}$ with a rectangular structuring element of U pixels width and V-pixels height, can be decomposed into two independent line structuring erosions, namely

$$\mathcal{R}_{UV} f = \min_{v=0}^{V-1} \min_{u=0}^{U-1} f(x+u, y+v) \tag{2}$$

$$= \mathcal{L}_V \mathcal{L}_U^T f \tag{3}$$

where $\mathcal{L}_V$ is the erosion with a vertical line structuring element of length V and the transpose results in an erosion with a horizontal line structuring element. Note that $\mathcal{R}_{11}$ and $\mathcal{L}_1$ are identities under erosion and dilation.

Let $\mathcal{P}_{2,j}$ be the erosion using a vertical linear structuring element consisting of two points only, one at the origin, and the other at $j-1$ pixels below the origin. A line erosion $\mathcal{L}_V$ can be decomposed into a sequence of the $\mathcal{P}_{2,j}$, each doubling in size, namely,

$$\mathcal{L}_V = \mathcal{P}_{2,V-2^n} \mathcal{P}_{2,2^n} \cdots \mathcal{P}_{2,8} \mathcal{P}_{2,4} \mathcal{P}_{2,2}. \tag{4}$$

where $n = \text{floor}(\log_2 V)$ (and we omit $\mathcal{P}_{2,1}$ from the end of the sequence as it is the identity). This provides a means of calculating line erosions in $O(\log k)$ operations as each application of $\mathcal{P}_{2,j}$ incurs only one comparison to calculate the minimum.

Line erosions can be calculated by the HGW algorithm with three comparisons per pixel, thus independently of the line structuring element size. Two memory buffers with as many elements of the column of the image are required. In the first buffer a down-ordered sequence of k minima are calculated from the image column with each k-th entry copied from the respective image pixel. The second buffer is similarly constructed but with an up-ordered sequence of k-minima. The erosion of the image column is then the minimum of the corresponding pixel in the first buffer with the second buffer displaced by k pixels.

The upshot is that all erosions (and dilations) involving line and rectangular structuring elements can be calculated in $O(1)$ time. Calculating erosions efficiently on arbitrary structuring elements, that is, bettering the $O(k^2)$ efficiency of the direct implementation of (1) is a non-trivial problem and the author is not aware of a single solution that improves efficiency for all possible structuring elements. Nevertheless, Urbach and Wilkinson [6] describe a good general purpose algorithm that provides significant efficiency gains on almost all structuring elements that are likely to be of interest.

The structuring element is broken down into individual connected horizontal chords. The chord lengths are sorted to provide a sorted list of unique chords lengths which is then padded with extra power of two lengths to ensure that the sequence of chord lengths required to compute any chord in the list via (4) is also present in the list. It is also noted that if there are two or more chords of the same length in the structuring element then the calculations of the erosion of the first chord on the image rows is identical to the calculations for subsequent chords of the same length but applies at some other row displacement in the image. These calculations are therefore stored to save on future calculation. The algorithm requires an extra 2D memory array of size of the number of rows in the structuring element by the number of columns in the image.

It is not necessary to break the structuring element down into horizontal chords; one can instead use vertical chords and process the image by columns. Urbach and Wilkinson choose between horizontal or vertical chords based on which one uses the fewest chords to represent the structuring element. As we point out below this is unlikely to provide the best performance in all cases due to the streaming nature of modern computer memory architecture.

III. Vectorised SIMD implementation

The SIMD CPU instructions in modern CPUs (SSE and AVX on Intel X86, Neon on Arm and Altivec on PowerPC) are a means of vectorised hardware acceleration particularly intended for multimedia data. Extra CPU registers of large bit size (128-bit on SSE, Neon and Altivec, 256-bit on AVX[2]) coupled with new CPU instructions permit a single operation to be performed simultaneously on multiple data. A 128-bit register, for instance, holds 16 unsigned 8-bit bytes, and addition between two such registers produces the results of 16 byte additions, usually in only one CPU cycle. Since calculations are repeated from pixel to pixel in an image, SIMD CPU instructions are very suited for accelerating image processing operators.

There are some constraints in implementing SIMD computer code. Since the same operation is applied to all data in the SIMD register, and because branching is costly on modern CPUs, it is usually more efficient to calculate both paths of a conditional statement then use special and efficient SIMD instructions that select the results between the two paths on an element by element basis. This is particularly true if the two paths are both short.

Most operations apply between SIMD registers, not along the elements of a SIMD register. Algorithms that operate on pixels between two rows of an image (or between two images) can be made very efficient, but operations that proceed along the row (such as the rotating sequence of minima along the line needed in the HGW algorithm) do not lead themselves to SIMD implementation along the row, but can be implemented with a transposed vector calculating multiple rows at once. But to do that is not necessarily efficient on modern computer hardware due to the way the memory subsystem is structured.

We would like to implement portable algorithms that can be easily compiled for various SIMD implementations (e.g. adjustable to the various stages of SSE and AVX implementations on the Intel platform, and to other CPU architectures), but that is made more difficult by the nature of the Intel implementation compared to that of the RISC CPUs (Arm and PowerPC). While some compilers (e.g. gcc and clang) can now

[2]The AVX/AVX2 SIMD implementation is dual 128-bit thus does not provide full 256-bit permutations or shifts.

automatically vectorised certain programming constructs those facilities are still primitive and insufficient for implementation of more complex operations such as image morphology.

The Intel SSE and AVX implementations provide SIMD instructions that operate between two (or more) SIMD registers, between SIMD registers and a single memory access, or on a SIMD register with one parameter provided as an immediate. There are 16 SIMD registers (we are considering 64-bit Intel hardware only), but the fact that one operand in a SIMD instruction can be a memory access reduces the risk of register spill to the stack. Most Intel instructions expect memory accesses to vectors to be aligned (i.e. to 128 or 256-bit boundaries), nevertheless provide some direct memory access instructions that can load misaligned vectors. The Intel implementation, particularly in the earlier incantations of SSE, is weak in not providing useful instructions for permutation and selection of vector elements, and provides certain operators only for limited data types (e.g. min and max operators only on unsigned byte and signed 16-bit integer data). Furthermore, some useful operators when implemented (e.g. the alignr operator for pulling out a misaligned vector from two neighbouring aligned vectors) only accept an immediate operand, which must be known at compile time rendering the instruction useless for many general purpose image processing applications.

In contrast the RISC implementations (Neon and Altivec) tend to have a much more orthogonal and complete implementation in that if an operator is provided for one data type, then it is provided for all data types that it makes sense with. They have a richer implementation of selection and permutation operators (byte table lookup in Neon), but that is necessary as they can only perform memory operations on vectors aligned in memory[3]. Converting between SIMD CPU registers and integer CPU registers and vice versa can be very costly, for example, on PowerPC the transfer can only be done via a memory store and load, and for certain Arm archv7 CPUs an extraction of a scalar element from a vector register can result in a substantial CPU stall.

For implementation of erosion along lines with arbitrary shifts we need to be able to construct vectors at any byte position along an image row. Let us consider two possible approaches. The first is to load two aligned neighbouring vectors from memory and construct the misaligned vector from the low part of one and the high part of the other. This approach must be taken on Neon and Altivec as the direct memory access of vectors at misaligned addresses is not supported by hardware, but this approach can be difficult on Intel SSE/AVX because the operators for constructing a misaligned vector from two neighbouring aligned vectors only take an immediate operand for the amount of misalignment, thus the misalignment must be known at compile time. On Intel SSE/AVX one therefore is forced into using the instructions for loading misaligned vectors direct from memory.

[3]Some Arm architectures can do misaligned Neon vector loads under certain circumstances, usually with a performance penalty.

To further illustrate the problem let us define the function `simd_cvu(l,r,o)` that constructs a misaligned (or 'unaligned') vector from the two neighbouring aligned vectors l and r where l would lie at the lower memory address 'left' of r (which is 'right' of l), and o is an integer specifying the byte offset into l where the misaligned vector starts.[4] Erosion with $\mathcal{P}_{2,4}$ (for an example) along an image row (with the start of the image row aligned to an SIMD vector) would therefore proceed something like:[5]

```
r = row[0];
for (int k = 0; k<row_length; k++) {
  l = r;
  r = row[k+1];
  u = simd_cvu(l, r, 4);
  result[k] = simd_min(l, u);
}
```

Note that we only load each vector from memory in the image row once, and that if we are processing unsigned bytes then the erosion of sixteen pixels (assuming a 128-bit SIMD vector) are computed in each iteration, for a potential speed-up of 16 times over scalar code. This code is easily compileable on all architectures because the offset in `simd_cvu()` is a constant integer known at compile time.[6]

Using such code in the computation of the chord erosions of the method of Urbach and Wilkinson requires the third parameter (the offset of the misaligned vector) of `simd_cvu` to be calculated on the fly according to the list of unique chord lengths. Such code using `simd_cvu` is not compileable on the Intel architecture because the third parameter is no longer a constant known at compilation. We therefore consider the alternative using the function `simd_ldu(a)` which takes an address to memory (a) which may be misaligned and returns the SIMD vector located at that address in memory. Intel CPUs have a single CPU instruction capable of performing this function (it is slower than an aligned memory access). On Arm Neon and PowerPC altivec the function would be implemented by zeroing the lowest bits in a to get an aligned address, loading the aligned vector l from that address, loading the aligned vector r from the next aligned address, calculating the amount of offset o from a, and calling `simd_cvu(l,r,o)` to return the misaligned vector. Note that on Neon and Altivec that two memory accesses are incurred for each invocation of `simd_ldu()`.

The implementation of the row erosion with $\mathcal{P}_{2,j}$, where the offset j is not known until runtime, is

```
for (int k=0; k<row_length; k++) {
  l = row[k];
  a = address_N_elements_right_of(row[k], j);
  u = simd_ldu(a);
```

[4]The exact implementation of `simd_cvu` need not concern us.

[5]We skip detail regarding image edges and avoiding reading/writing past row array ends in these code examples.

[6]On Intel SSE one requires the SSSE3 CPU extension to implement `simd_cvu()` efficiently. It is only older CPUs that do not have this feature.

```
  result[k] = simd_min(l, u);
}
```

(Here the function `address_N_elements_right_of()` calculates the required misaligned address.) This works well on Intel SSE/AVX, but is potentially less efficient on Arm and PowerPC due to the two memory accesses in `simd_ldu()`, i.e., the re-loading of data from memory that is already available in CPU registers.

For the purposes of this paper we use the `simd_cvu()` alternative wherever possible (i.e. when the offset is a known constant at compilation) but resort to the `simd_ldu()` alternative otherwise despite a potential performance impact on certain architectures.

Finally we should mention the streaming nature of modern memory architecture [10]. Memory hardware is optimised for (thus very efficient for) reading (streaming) data from sequential memory locations. Random access to memory is, however, inefficient and can result when data are not in cache, in stalls of the CPU for hundreds of clock cycles. Assuming C style arrays (memory increases sequential in address along a row of an array or image) then it is most efficient to process images along rows. Processing down columns can have a very noticeable detrimental impact (sometimes by many factors) on performance. Drepper gives a nice tutorial description of these issues and solutions with the example of computing a matrix multiplication [10].

IV. Implementation Detail

We implement in C in a portable manner the erosion and dilation with arbitrary flat structuring elements using the chord based method of Urbach and Wilkinson. The implementation is in two versions: scalar code that operates by pixel pumping, and an SIMD implementation that vectorises operations wherever possible. Both processing by row (horizontal) chords and column (vertical) chords is implemented. Note that the distinction between row and column chords is only relevant when reading or writing to the image being processed. The processing to construct the chord erosion table, and update it when each row (or column) of the image is read, is independent of the direction data are processed in the image, thus can be calculated very efficiently with an SIMD implementation of a sequence of $\mathcal{P}_{2,j}$ operators. The code uses the `simd_ldu()` function because the offsets are calculated at run time.

The operation to calculate the minimum (or maximum for dilation) across the results of the chords and write back to the image can be very effectively vectorised, because the vector data can be read from each chord in the table, the minimum calculated via the vector min instruction, and the vector written back to the image row with the vector write to memory. This is not the case if vertical chords are used. The resultant vector from the chord table needs to be written down the image column, that is, each element of the vector is to be written to a memory address that is offset by the size of each image row. The only why to achieve this is to extract each element from the CPU SIMD register, and use scalar code to write down the image column. Given that extraction of a vector element is quite inefficient on some CPU architectures we have not implemented this. The vertical chord code remains scalar at this stage.

For line erosions in the vertical direction the two line buffers of the HGW algorithm is extended to be an array of the size of a SIMD vector in one direction, and of the image column in the other. A full SIMD vector of image data (i.e. sixteen columns of the image) is processed simultaneously. To better exploit cache spatial locality we actually implement the two line buffers to be 64-bytes wide (the width of a line in the CPU cache) rather than 16, and process the image in blocks of N rows by 64 columns where N is the number of rows in the image.

Implementing the HGW algorithm for horizontal line erosions is difficult and might well be impossible to implement efficiently. The problem is that no architecture provides SIMD min and max CPU instructions that operate along the vectors. We could construct misaligned vectors via `simd_cvu()` but the requirement that the offset be a constant known at compile time on the Intel architecture provides a significant obstacle. In any case, we would be using only one element from the vector for each displacement calculated via `simd_cvu()` thus wasting most of the efficiency of the vector calculations, so much so that the efficiency unlikely to be better than scalar code!

We therefore process horizontal line erosions with the recursive line algorithm using $\mathcal{P}_{2,j}$. Because we must iterate through a number of offsets in the erosions we use the `simd_ldu()` function.

We present results for compilation using Debian gcc version 4.9.2-10 at optimisation level -O2 targeted at Intel SSE up to and including the SSE4.1 extensions. We compare against OpenCV as compiled by Debian for Debian Stable 8.2 (also known as 'Jessie') on the x86_64 architecture. Debian compiles software to work on generic x86_64 thus limits compiled packages to use the features of SSE2 only, excepting that the package detects the specific CPU at run time and verifies that it can use newer CPU extensions.

We run the morphological operators on unsigned byte monochromatic images of size 1055×1025 pixels initialised with random data. The performance events subsystem of the Linux kernel is exploited to provide hardware counts of the number of cycles and instructions executed in the morphological operation, and the elapsed time of the operation while it is scheduled on the CPU. The operation is run once on the image to warm up the caches, and then timing is started, the operation is performed 10 times, and then timing is stopped. The elapsed time is provided in microseconds, and the number of CPU cycles executed in kilocycles. We find the number of CPU cycles to be the most consistent measure across multiple invocations of the test program.

V. Results and Discussion

We present the timing results for the vertical and horizontal line erosions in Fig. 1. The solid line type is duplicated: the

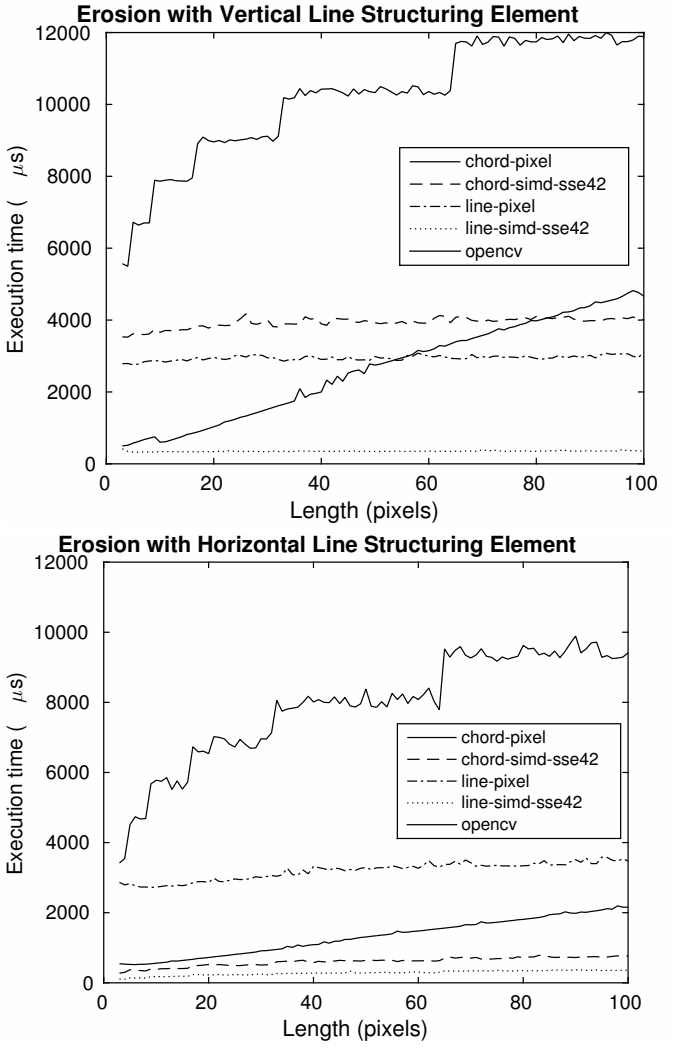

Fig. 1. Time to computer erosions with vertical (top) and horizontal (bottom) lines of increasing length.

top solid line in both graphs is the chord-pixel result and the bottom solid line is OpenCV result. The OpenCV result has a reasonable speed because of the acceleration using the SSE instructions of the Intel processor, nevertheless the time take to compute the erosion increases in proportion to the length of the structuring element because (1) is implemented.

The implementation of the chords method without SIMD acceleration (chord-pixel) is, as expected, the slowest, and increases logarithmically with structuring element length due to the use of the recursive line erosion algorithm. The horizontal line erosion is faster than the vertical erosion due to the better memory access pattern, hence a greater likelihood of cache hits. The SIMD implementation (chord-simd-sse42) is faster in both the vertical and horizontal line cases, but the biggest speed up is achieved for the horizontal situation. Indeed it beats the accelerated OpenCV implementation in this case. The significant difference in processing horizontal chords versus vertical chords suggests that it may be better to implement a preference for processing horizontal chords for arbitrary

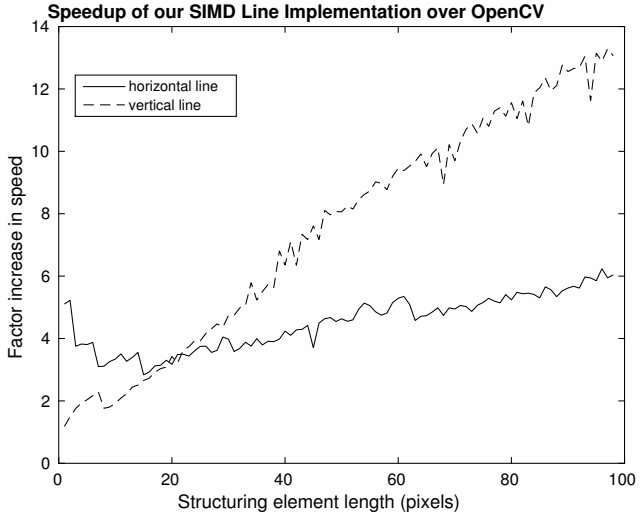

Fig. 2. Speedup of our SIMD line erosion implementations over that provided by OpenCV.

structuring elements even if the number of horizontal chords greatly exceeds the number of vertical chords. We comment further below.

The execution time of the unaccelerated HGW algorithm (line-pixel) barely increases with line structuring element length and in the case of the vertical line erosions is faster than the OpenCV implementation for structuring elements of length of about 58 pixels and longer. Note that our SIMD implementations of line erosions (line-simd-sse42) uses the fast line processing algorithm only for vertical line erosions and uses the recursive algorithm for the horizontal direction. Nevertheless, the erosions in the horizontal direction are faster than the corresponding vertical one (particularly for structuring element lengths below about 50 pixels) despite the less efficient algorithm. This comes about because of the better use of cache when processing image data along rows of the image. The SIMD implementations are very much faster than OpenCV. A speed comparison of our fastest algorithm versus OpenCV is shown in Fig. 2. The nature of the two algorithms (recursive versus HGW) is very evident with the vertical line erosion (using the HGW algorithm) increasing to an order of magnitude speed-up over OpenCV for long structuring element lengths.

This speedup in using an efficient line erosion algorithm becomes very much more pronounced for two-dimensional rectangular erosions. Since it can be decomposed into two line erosions the rectangular erosion is as efficient as the line algorithms, however OpenCV implements the direct algorithm (1) without exploiting the decomposition into line erosions. Because of this the OpenCV implementation shoots off the top of the graph shown in Fig. 3 even before we get to 10×10 pixel erosions Even our scalar implementations (chord-pixel and line-pixel) beat OpenCV at square erosions greater than 3×3 pixels in size! A SIMD implementation of a bad algorithm, no matter how cleverly implemented it is, can perform worse than

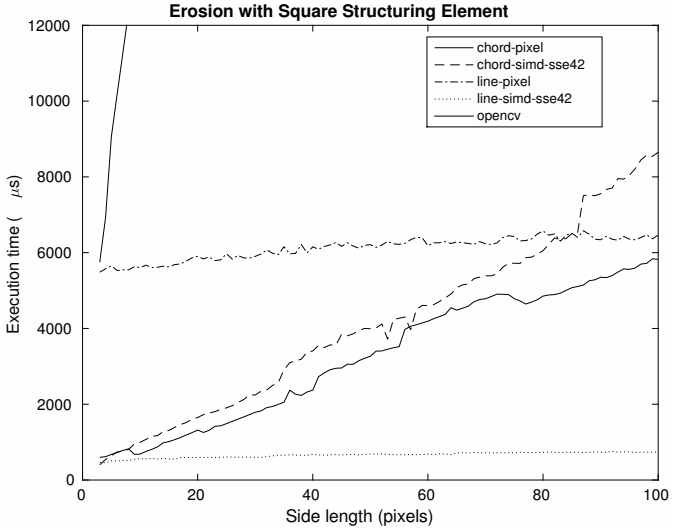

Fig. 3. Time to compute erosions with a square structuring element. The solid line shooting off the top of the graph is OpenCV.

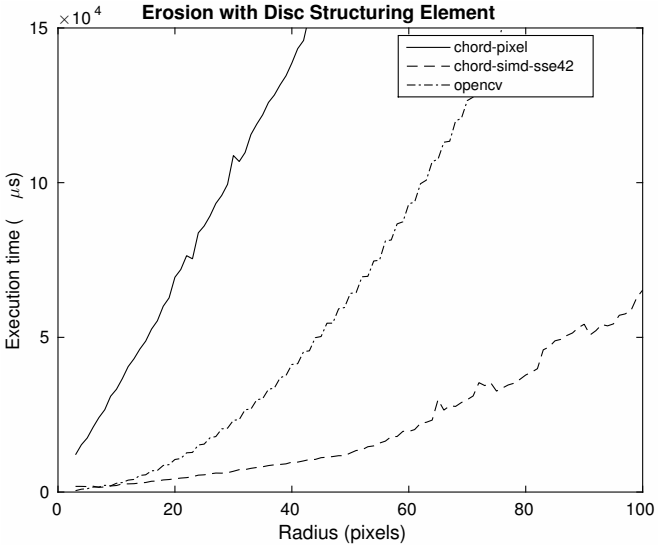

Fig. 4. Time to computer erosions with a disc structuring element.

straight-forward scalar implementations of a better algorithm.

At this point it is worth making a comparison to Matlab, a commonly used commercial package. In 2008, Urback and Wilkinson reported that Matlab implemented the naïve direct equation for erosion [6], but our tests with Matlab R2015a reveal that the morphology implementation for vertical line erosions execute in a constant (approximately) 20 000 μs for increasing line structuring element size, and in about 27 000 μs for horizontal line erosions. The vertical erosion is quicker because Matlab stores arrays in column priority order, thus processing in columns better utilises cache. While a much better algorithm is now implemented, the time to process an image is still rather slow on Matlab: our SIMD code is nearly two orders of magnitude faster.

As a final result we show the erosion using a disc-like structure. We define the disc to be those pixels contained within a fixed radius of the origin. We admit that such a structuring element is not suitable for granulometry, but it is useful for our purposes as a structuring element that is not easily decomposed into a series of simpler and more efficient erosions, nevertheless the SIMD chord implementation gives a significant speed-up over the OpenCV implementation. While not shown in the graph, the Matlab implementation returns to the direct implementation of the erosion (1) and gets extremely slow for large discs (approximately 20 s per image erosion for discs of radius 100 pixels). Our SIMD implementation is taking approximately 30 ms for the same.

VI. Conclusion

We demonstrated SIMD implementations on Intel architecture of line erosions that are faster than OpenCV by many factors, and of square erosions by orders of magnitude, particularly for larger structuring element sizes. SIMD implementations must take account of the streaming nature of memory and the organisation of data in the image matrix. For example, we found that that implementing the HGW line algorithm in SIMD for horizontal line erosions was not feasible, and resorted to the slightly less efficient recursive line algorithm. Nevertheless impressive speed gains were found for commonly used structuring element sizes.

References

[1] J. Serra, *Image Analysis and Mathematical Morphology*, 2nd ed. New York: Academi, 1982.

[2] A. van Herk, "A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels," *Patt. Recog. Lett.*, vol. 13, pp. 517–521, 1992.

[3] J. Gil and M.Werman, "Computing 2-D min, median and max filters," *IEEE Trans. Pat. Anal. Mach. Intell.*, vol. 15, no. 5, pp. 504–507, 1993.

[4] E. B. P. Soille and R. Jones, "Recursive implementation of erosions and dilations along discrete lines at arbitrary angles," *IEEE Trans. Pat. Anal. Mach. Intell.*, vol. 18, no. 5, pp. 562–567, 1996.

[5] J. Gil and R. Kimmel, "Efficient dilation, erosion, opening and closing algorithms," *IEEE Trans. Pat. Anal. Mach. Intell.*, vol. 24, no. 12, pp. 1606–1617, 2002.

[6] E. R. Urbach and M. H. F. WIlkinson, "Efficient 2-d grayscale morphological transformations with arbitrary flat structuring elements," *IEEE Trans. Im. Proc.*, vol. 17, no. 1, pp. 1–8, 2008.

[7] M. van Droogenbroeck and H. Talbot, "Fast computation of morphological operations with arbitrary structuring elements," *Pattern Recognot. Lett.*, vol. 17, pp. 1451–1460, 1996.

[8] G. R. Bradski and V. Pisarevsky, "Intel's computer vision library: Applications in calibration, stereo, segmentation, tracking, gestere, face and object recognition," in *Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR2000)*, Hilton Head Island, SC, Jun 2000, pp. 2796–2797.

[9] G. Bradski, "The OpenCV library," *Dr Dobbs Journal*, no. Nov 01, 2000.

[10] U. Drepper, "What every programmer should know about memory, part 5: what programmers can do," *Linux Weekly News*, October 2007, https://lwn.net/Articles/255364.