

# Fast Morphological Image Processing Open-Source Extensions for GPU Processing With CUDA

Matthew J. Thurley and Victor Danell

**Abstract**—GPU architectures offer a significant opportunity for faster morphological image processing, and the NVIDIA CUDA architecture offers a relatively inexpensive and powerful framework for performing these operations. However, the generic morphological erosion and dilation operation in the CUDA NPP library is relatively naive, and performance scales expensively with increasing structuring element size. The objective of this work is to produce a freely available GPU capability for morphological operations so that fast GPU processing can be readily available to those in the morphological image processing community. Open-source extensions to CUDA (hereafter referred to as LTU-CUDA) have been produced for erosion and dilation using a number of structuring elements for both 8 bit and 32 bit images. Support for 32 bit image data is a specific objective of the work in order to facilitate fast processing of image data from 3D range sensors with high depth precision. Furthermore, the implementation specifically allows scalability of image size and structuring element size for processing of large image sets. Images up to 4096 by 4096 pixels with 32 bit precision were tested. This scalability has been achieved by forgoing the use of shared memory in CUDA multiprocessors. The vHGW algorithm for erosion and dilation independent of structuring element size has been implemented for horizontal, vertical, and 45 degree line structuring elements with significant performance improvements over NPP. However, memory handling limitations hinder performance in the vertical line case providing results not independent of structuring element size and posing an interesting challenge for further optimisation. This performance limitation is mitigated for larger structuring elements using an optimised transpose function, which is not default in NPP, and applying the horizontal structuring element. LTU-CUDA is an ongoing project and the code is freely available at <https://github.com/VictorD/LTU-CUDA>.

**Index Terms**—Morphological image processing, erosion, dilation, GPU, NVIDIA, CUDA.

## I. INTRODUCTION

### A. Background

MORPHOLOGICAL image processing is powerful non-linear image analysis tool for the analysis of spatial structure based on pre-defined spatial structures known as structuring elements. The fundamental operations of morphological image processing, erosion and dilation, have in

the most general case, time complexity  $O(npq)$  for  $n$  image pixels and rectangular structuring elements of size  $p * q$ . Faster computation can be achieved using both structuring element decomposition [1, p. 12–13] (or refer to any textbook on morphological image processing) if an approximation of the structuring element is satisfactory, and a number of advanced algorithms such as the van Herk/Gil-Werman (vHGW) algorithm [2], [3] for constant time (with respect to structuring element size) erosion and dilation of rectangular and octagonal structuring elements, or Gil and Kimmel [4].

Application to parallel architectures such as a GPU offers further potential for faster processing, however GPU implementations based on traditional  $O(npq)$  or other sub-optimal time complexity algorithms can still perform worse than efficient CPU implementations as image size and structuring element size increases [5]. Domanski *et al.* [6] provide a short description of a GPU implementation of the vHGW algorithm [2], [3] demonstrating constant time computation at a significant gain over CPU implementation. However, Domanski *et al.* [6] use shared memory limiting the size of structuring elements, and the code was not available due to restrictions from their employer.

The objective of this work is to produce a freely available GPU capability for fast morphological operations so that fast GPU processing can be readily available to those in the morphological image processing community.

A non-exhaustive survey of image processing libraries identified the following existing libraries;

CUDA is a device architecture developed by NVIDIA that allows general parallel computation to be performed on their CUDA enabled GPU graphics cards. NVIDIA provides the NPP library at no cost that contains fundamental morphological operations. However, at the time of writing NPP supports erosion and dilation using a flat, rectangular shaped ( $p * q$ ) structuring element,  $O(npq)$  complexity, for images with 8 bit depth. NPP provides a solid generic implementation and is maintained by NVIDIA. Therefore it offers a good platform upon which to provide more optimised and open-source extensions. On request NVIDIA provided part of the source code for their generic erosion and dilation.

Alternate open-source libraries, OpenCV\_GPU [7] for CUDA, GPUCV [8], and etothepe-CUDA-Image-Processing [9] were inadequate. OpenCV appeared no faster when running in GPU mode and became unstable when eroding with non-square structuring elements. GPUCV stated only support for square structuring elements on the GPU with all others processed using the CPU, and etothepe-CUDA-Image-Processing

Manuscript received February 06, 2012; revised May 15, 2012; accepted May 29, 2012. Date of publication June 14, 2012; date of current version October 12, 2012. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Daya Sagar Behara.

The authors are with the Department of Computer Science, Electrical, and Space Engineering, Luleå University of Technology, Luleå 97187, Sweden (e-mail: matthew.thurley@ltu.se; victor.danell@gmail.com).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/JSTSP.2012.2204857

appeared to be in an unreliable, and unmaintained state when tested.

The MATLAB [10] image processing toolbox is not a parallel implementation but it forms a useful comparison for this work as it provides detailed morphological image processing capabilities and implements optimised algorithms. MATLAB uses the van Herk algorithm [2], [11] for fast erosion and dilation, and implements structuring element decomposition for faster computation [12].

Halcon [13] is a commercial machine vision and software development library. Halcon was easy to use and was significantly faster than MATLAB for a  $4096 \times 4096$  image using an  $11 \times 11$  structuring element. Larger structuring elements caused Halcon's performance to significantly decrease revealing a non-constant time implementation with respect to structuring element size. So while faster than MATLAB in the tested cases, Halcon did not exhibit the same constant time optimised algorithms from MATLAB that we wish to compare against so Halcon is not considered in our assessment.

For our research, a high performance parallel implementation (in this case GPU based) requires the following capabilities;

- free software as in freedom to further develop and extend the code, and free price,
- optimised erosion and dilation algorithms that can provide performance equivalent to vHGW,
- capacity to processes floating point images and provide floating point structuring elements,
- capacity to process large images with between one and four million pixels, and larger in the future.

The last two requirements stem from the need to process 3D surface profile data from single or multiple range sensors [14], [15] with 16 bit depth or larger resolution per data set.

### B. Contribution

This paper presents a freely available implementation of fast morphological erosion and dilation based on CUDA/NPP using the vHGW algorithm [2], [3], [6] implemented in a way that allows scalability of image size and structuring element size, and 8 and 32 bit depth data handling. Testing highlights an important limitation and performance bound based not on the number of cores, but on CUDA global memory handling for vertically oriented structuring elements. This vertical structuring element limitation can be mitigated for larger structuring elements using an optimised transpose [16], which is not the default implementation in NPP, and then applying the horizontal structuring element.

Erosion and dilation are implemented for a base set of structuring elements from which larger structuring elements can be constructed via structuring element decomposition. Comparison is provided against CUDA's NPP library generic structuring element implementation  $O(npq)$  and against MATLAB's image processing toolbox [10]. A test framework using MATLAB is provided in LTU-CUDA.

Although MATLAB is not known as a particularly fast implementation environment for programming it demonstrates vHGW constant time performance as the structuring element size increases. Therefore performance results from MATLAB provide a useful relative comparison.

TABLE I  
vHGW ALGORITHM

- 
- 1 Image rows are partitioned into segments of length  $p$  with  $(p-1)/2$  columns of overlap on each side to form a window of size  $2p-1$ , centered at  $p-1, 2p-1, 3p-1, \dots$
  - 2 For each pixel  $k = 0..(p-1)$  in a given window  $w$ , a suffix max array  $R$  is created for the pixels left of center  
 $R[k] = \max(w[j]) : j = k..(p-1)$   
and a prefix max array  $S$  is created for the pixels right of center  $(p-1)..(2p-2)$ ,  
 $S[k] = \max(w[p-1+j]) : j = 0..k$
  - 3 For each pixel  $\frac{(p-1)}{2} \leq j < p + \frac{(p-1)}{2}$  in  $w$  (the segment of length  $p$ ) the dilation result is  
 $result[j] = \max(R[m], S[m])$ , where  $m = j - \frac{p-1}{2}$
- 

## II. IMPLEMENTATION

### A. CUDA Parallel vHGW

The implementation begins with the vHGW algorithm for a dilation with a 1D horizontal structuring elements of size  $p = 2N + 1$  implemented for the GPU [2], [3], [6]. Erosion follows the same process using minimum arrays. Table I outlines the vHGW algorithm.

Domanski *et al.* [6] implement this with shared memory arrays for the max arrays  $R[i]$ ,  $S[i]$ , and 2 threads per window  $w$ , one for each max array. They show the result of a dilation using a square structuring element of various sizes up to  $63 \times 63$ , showing constant time computation regardless of structuring element size on an image of  $1024 \times 1024$ . It is not exactly clear how the vertical structuring element component of this dilation was implemented, but Domanski *et al.* describe two alternatives. Firstly, a modified horizontal dilation that writes the results out into a transposed result image so that a subsequent horizontal dilation will generate the square dilation result, and secondly that individual horizontal and vertical structuring element kernels would provide better potential to performed coalesced (grouped) memory accesses. It is not clear which method was implemented in their result but it seems the former was likely.

LTU-CUDA uses the following implementation details for vHGW;

- Separate code for horizontal, vertical, and diagonal  $\pm 45$  degree structuring elements
- A single thread calculates the max values for the two max arrays  $S[k]$ ,  $R[k]$
- No shared memory is used as this limits the utilization of CUDA's multiprocessors as image and structuring element size increases.

Shared memory usage is an important issue in the implementation. If one uses multiple threads for a single result as Domanski *et al.* [6] it is necessary to have inter-thread communication such as via shared memory to store the max arrays for the two threads. Shared memory is however limited in size making it only viable for smaller images and structuring elements. CUDA enabled GPUs have shared memory per multi-

processor of 48 KB at CUDA compute capability version 2.0, and 16 KB prior ([17], Appendix F).

A few CUDA definitions are necessary at this point;

- A CUDA “multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps” ([17], Section 4.1)
- A CUDA multiprocessor contains a number of blocks of threads, with a maximum of 8 blocks of threads in CUDA compute version 2.x and below ([17], Appendix F).
- Block size should be a multiple of 32 threads (a warp) with a recommended minimum of 64 [18] (2 warps).

Assuming it is possible and efficient to use only 32 threads (one warp) we can therefore perform the horizontal Domanski dilation on an image with 1100 columns per row and a structuring element of size 11 using greater than 32 KB of shared memory as follows;

- structuring element is a horizontal line of 11 pixels, therefore  $p = 11$
- each image row is 1100 pixels therefore  $1100/p = 100$  windows
- 2 arrays per window using  $2p$  values & 2 threads
- each block using 32 threads
- therefore each block requires  $100 * 2p * 32 / 2 \approx 34$  K values of shared memory, which equals 34 KB for 8 bit integer images, or 136 KB for 32 bit floating point images.
- each multiprocessor has 48 KB of shared memory for a card supporting CUDA compute capability of 2.0 or better.

Shared memory on the multiprocessor is 71% consumed for the 8 bit case (with only an 11 pixel structuring element), and exceeded by a factor of 1.8 for the 32 bit floats. In addition, a resource allocation that allocates all 48 KB of shared memory to a single block with 32 threads per block results in only 1 active warp for that multiprocessor. With only 1 active warps out of a possible 48 warps (CUDA computer version 2.0) ([17], Appendix F) an occupancy rate (thread usage rate) of the only 2% is achieved for that multiprocessor ([19], CUDA Occupancy Calculator) significantly limiting potential performance.

Given the stated goal to support large images between one and four million pixels total, with significantly larger than 8 bit precision, shared memory usage for the vHGW max arrays is not viable with this strategy. Further investigation into how to gain the benefits of shared memory without limiting scalability is ongoing.

### B. Supported Operations

LTU-CUDA development thus far assumes that given a sufficient base set of structuring elements, structuring element decomposition can be applied to create other required structuring elements. The focus is presently on a set of simple structuring elements, with more complex shapes left to a generic NPP  $O(npq)$  erosion and dilation. LTU-CUDA provides the following operations and structuring elements primitives from which a variety of other shapes can be constructed including squares, rectangles, diamonds, octagons, and 8-sided approximations to a circular disc;

- erosion and dilation using 8 bit or 32 bit images
- flat horizontal, vertical, and diagonal  $\pm 45$  degree line structuring elements (of odd length) using GPU vHGW.

- two 3 by 3 mask structuring elements implemented using loop free code providing a 10–20% speed improvement over NPP.
  - 5 pixel cross or ‘+’ symbol,
  - 4 pixel hollow cross,
- other structuring elements (non flat, non-filled) are currently implemented using a generic structuring element kernel equivalent to the NPP generic algorithm.

### C. Development and Testing Conditions

LTU-CUDA has been tested and runs under linux 32 bit (ubuntu 10.04 LTS) and windows 7 64 bit. MATLAB and the image processing toolbox are not necessary but if present can be used to generate comparisons with LTU-CUDA. LTU-CUDA works with CUDA toolkit 4.0, 4.1 and 4.2. It is important to use the NPP version that corresponds to the installed CUDA toolkit or strange behavior will appear without any compiler warnings or errors. There were a few issues with the default installation of the toolkit, where a number of inline functions defined in npps.h had to be prefixed with ‘static’ for compilation to complete successfully.

MATLAB uses structuring element decomposition for many area based structuring elements, including the square and disc structuring elements used in this work. The following MATLAB functions and structuring element decompositions were used with LTU-CUDA being tested with the equivalent decomposition;

- `strel('line', angle, length)` produces linear structuring elements with angle equal to 0, 90,  $\pm 45$  to create horizontal, vertical and 45 degree diagonal lines.
- `strel('square', side_length)` produces a square structuring element, which is decomposed into two structuring elements, horizontal and vertical lines of length `side_length`.
- `strel('disc', radius)` produces an 8-sided approximation of a circular disc decomposed into a combination of horizontal, vertical and  $\pm 45$  degree linear structuring elements.

As we use structuring element decomposition for MATLAB and LTU-CUDA it is only reasonable to apply structuring element decomposition for NPP where this would improve performance. NPP uses a generic arbitrary rectangular grid structuring element mask. Therefore we expect that this will be efficient in terms of the mask size for a horizontal line and vertical line structuring elements. However for a diagonal line of length  $p$  this would require a square structuring element of size  $p * p$ . Using structuring element decomposition it is possible to perform repeated operations with a 3 by 3 structuring element that contains a 3 pixel diagonal line. Iterating  $m$  times would produce a line of length  $3 + 2 * (m - 1)$ . The following structuring element decompositions were used to achieve the fastest results for NPP;

- 1) Diagonal line of length  $p$  represented by  $(p - 3)/2 + 1$  iterations of a 3 by 3 structuring element mask containing a 3 pixel diagonal line. Only odd length diagonal lines were created.
- 2) Disc structuring element approximated using  $\pm 45$  degree diagonal lines and NPP’s generic structuring element for

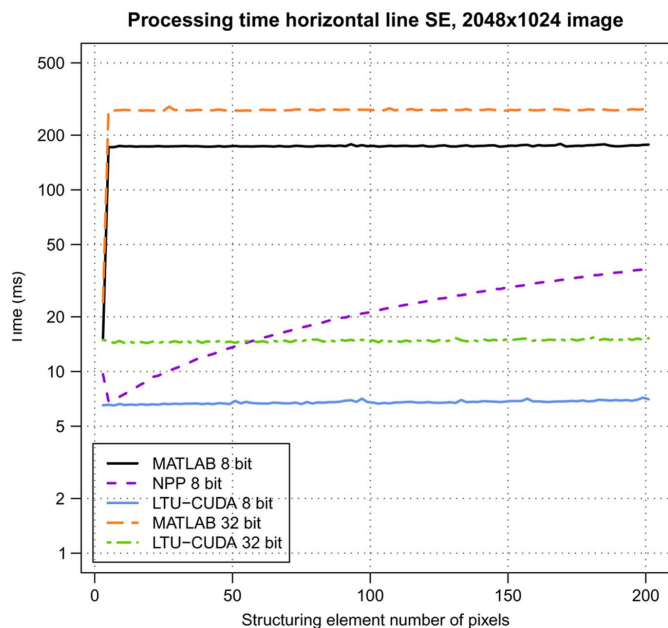


Fig. 1. Timing results for a horizontal line structuring element on a  $2048 \times 1024$  image. LTU-CUDA uses the GPU vHGW horizontal line SE. Logarithmic vertical axis.

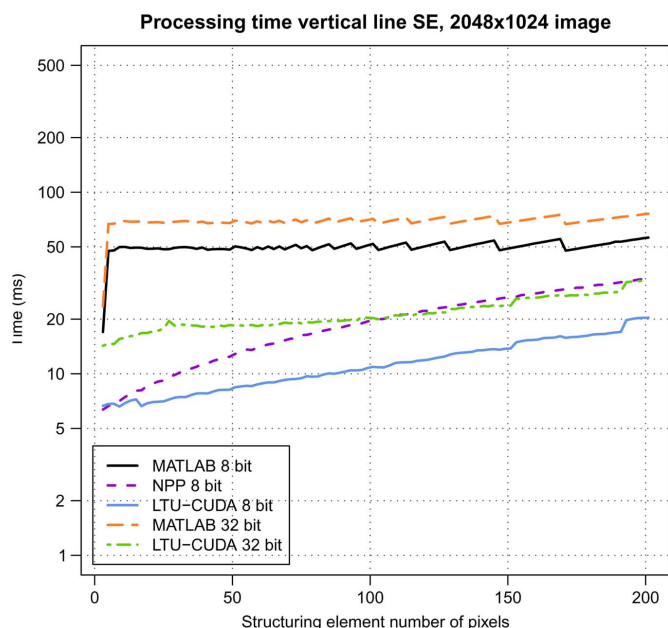


Fig. 2. Timing results for a vertical line structuring element on a  $2048 \times 1024$  image. LTU-CUDA uses the GPU vHGW vertical line SE. Logarithmic vertical axis.

horizontal and vertical structuring elements. The diagonal lines are further decomposed as described here in item 1).

### III. RESULTS

The presented results were produced using morphological erosion on a NVIDIA GTX 470 with 448 CUDA cores running NVIDIA driver 270.41.19, AMD Athlon II X4 620 CPU running at 3067 MHz, and with MATLAB version 7.10.0 R2010a. Performance results were generated under linux 32 bit (ubuntu

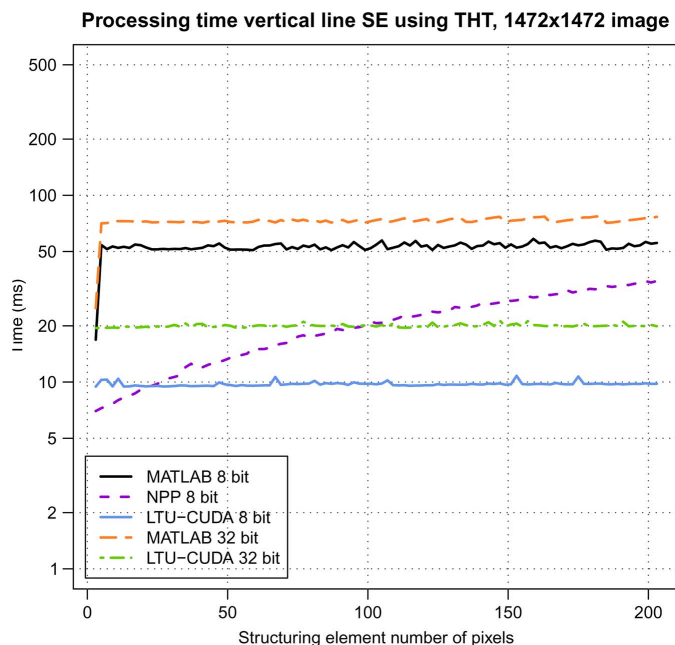


Fig. 3. Timing results for a vertical line structuring element on a  $1472 \times 1472$  image. LTU-CUDA uses the GPU vHGW vertical line SE calculated using an optimised Transpose [16], then a Horizontal line, and finally the same Transpose (the combination denoted THT). Logarithmic vertical axis.

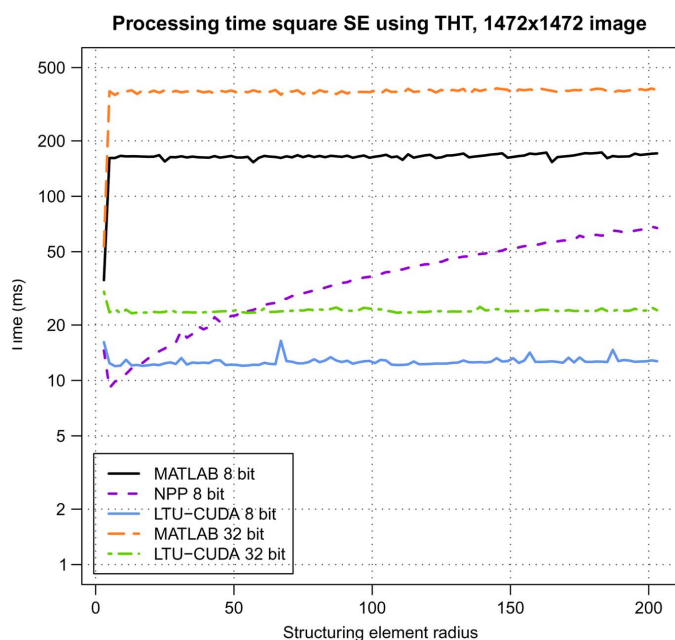


Fig. 4. Timing results for a square structuring element on a  $1472 \times 1472$  image. LTU-CUDA uses the GPU vHGW square SE (vHGW horizontal line dilated by THT-vertical line). Logarithmic vertical axis.

10.04 LTS). Performance results for NPP and LTU-CUDA include not only the morphological image processing time, but also all operations required to replace the functionality provided by MATLAB including memory allocations and transfers to the GPU and back.

Table II shows the mean and standard deviation of the increased performance of LTU-CUDA as a multiple of MATLAB and NPP calculated from the results shown in Figs. 1 to 6.

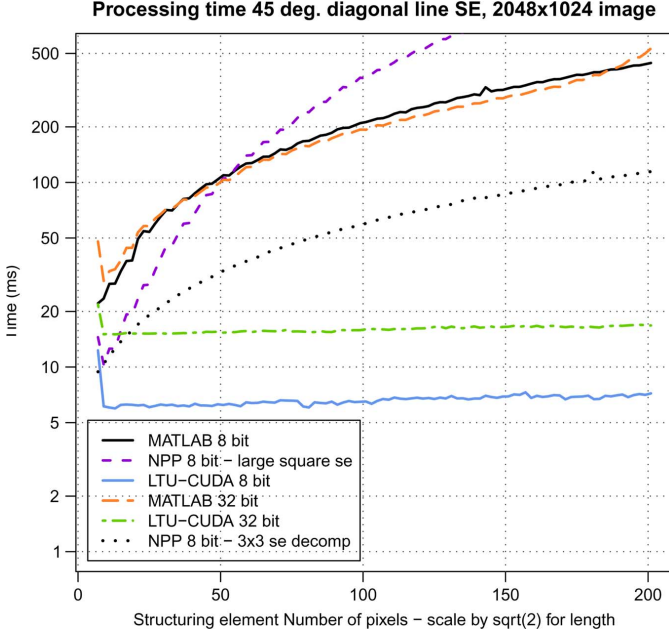


Fig. 5. Timing results for a diagonal line SE ( $\pm 45^\circ$ ) on a  $2048 \times 1024$  image. LTU-CUDA uses a vHGW diagonal line. NPP is demonstrated in two cases; firstly using a large square structuring element containing the entire diagonal line, and secondly using structuring element decomposition with a repeated series of 3 by 3 masks with a 3 pixel diagonal line to improve NPP's performance. Logarithmic vertical axis.

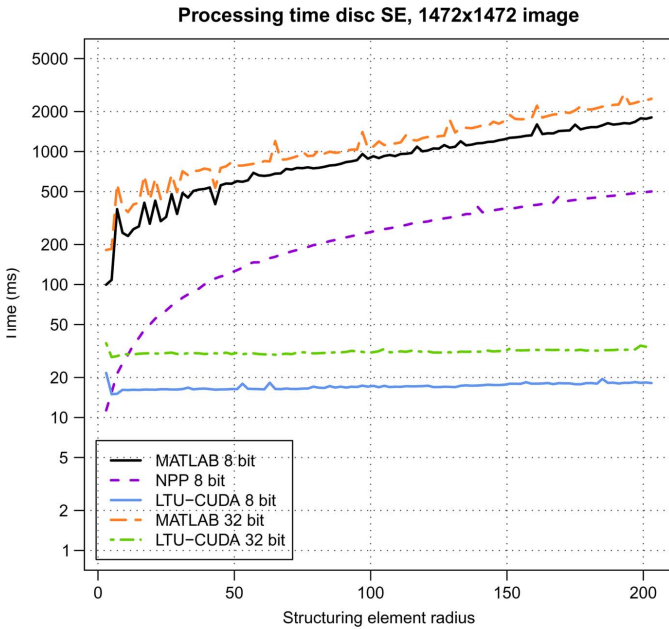


Fig. 6. Timing results for an 8-sided approximation of a circular disc structuring element on a  $1472 \times 1472$  image. The disc is based on structuring element decomposition using horizontal, vertical and  $\pm 45$  degree diagonal line structuring elements following the strategy employed by MATLAB ([10], `strel('disc', radius)`).

Example results are graphed for 2 M pixel images. In most cases a  $2048 \times 1024$  image size has been used. However, due to resulting non-constant time performance of the vHGW vertical line structuring element in CUDA, a THT-vertical line structuring element has also been implemented based on an

TABLE II  
PERFORMANCE INCREASE OF LTU-CUDA (MEAN AND STANDARD DEVIATION) AS A MULTIPLE OF NPP AND MATLAB, INCLUDING CUDA OVERHEADS FOR MEMORY TRANSFER AND ALLOCATION

Performance Increase : LTU-CUDA as a multiple of NPP and MATLAB						
structuring element	MATLAB 8		NPP 8bit		MATLAB 32	
	mean	stdev	mean	stdev	mean	stdev
horizontal line	25.5	2.38	3.18	1.26	18.5	1.73
vertical line	4.73	1.38	1.66	0.25	3.29	0.58
vertical-THT line $\phi$	5.45	0.42	2.13	0.83	3.64	0.25
square THT $\dagger$	13.0	1.21	3.01	1.34	15.5	1.43
diagonal line	33.1	17.3	9.22*	4.32*	13.1	7.21
disc $\ddagger$	54.1	23.2	14.7	7.84	40.3	18.3

$\phi$  THT-vertical line comprises the operations: optimised transpose [16], erosion with a vHGW horizontal line, and optimised transpose

$\dagger$  square uses a vHGW horizontal line followed by a vHGW THT-vertical line

\* Comparison against NPP for the diagonal line is based on the faster structuring element decomposition with a series of 3 by 3 mask structuring elements containing a 3 pixel line

$\ddagger$  disc uses a series of 4 structuring elements comprising vHGW horizontal, vertical, and  $\pm 45$  degree diagonal lines. The NPP comparison further decomposes the diagonal lines into a series of 3 by 3 masks.

optimised transpose [16] and the vHGW horizontal line structuring element. Square  $1472 \times 1472$  images have been used for all operations that use this THT-vertical structuring element as this optimised transpose is currently restricted to square images “whose dimensions are integral multiples of 32. However, modifications of (the) code required to accommodate matrices of arbitrary size are straightforward” [16].

#### IV. DISCUSSION

The results for vHGW horizontal line structuring element in Fig. 1 show the performance of the vHGW algorithm both in LTU-CUDA and MATLAB independent of structuring element size.

The results for the vHGW vertical line structuring element in Fig. 2 are not as desired and show interesting results for LTU-CUDA and MATLAB. LTU-CUDA shows a significant dependency on structuring element size although still significantly better than NPP, and MATLAB shows a substantially faster result (nearly 4 times faster) than MATLAB processing horizontal lines. Both results are related to memory handling.

The LTU-CUDA results in Fig. 2 are memory bound rather than computationally bound as a result of how the image is laid out in CUDA global memory.

Further investigation and optimisation to overcome the CUDA memory bottleneck for vertically oriented operations is of interest. Transposing the image using the default NPP transpose was attempted but the overhead of this operation made the result even slower. However, using an optimised transpose function a THT-vertical line structuring element was implemented based on transpose, vHGW horizontal line structuring element, and transpose. This optimised transpose implementation is provided by NVIDIA [16] but it is not the default transpose operation in NPP. The optimised transpose currently only works with square images that are a multiple of

32 but as noted earlier, this limit can be overcome and further development will allow more flexible image dimensions. The default transpose exchanges horizontal row data with vertically oriented column data. The optimised transpose [16] views the image matrix as a series of  $\pm 45$  degree diagonal rows of data, instead of horizontal rows and vertical columns. Indexing the data this way allows CUDA to avoid continuous column ordered memory access in the CUDA global memory which typically creates a bottleneck.

From Ruetsch and Micikevicius [16] “global memory is divided into a number of partitions of 256-byte width. To use global memory efficiently, concurrent access to global memory should be divided evenly amongst partitions. The term partition camping is used to describe the case when global memory accesses are directed through a subset of partitions, causing requests to queue up at some partitions while other partitions go unused”. This is the case with the vHGW vertical line structuring element as memory access “column-wise . . . will typically access global memory through just a few partitions” [16].

For the MATLAB results in Fig. 2 the results are significantly faster than the horizontal line case in Fig. 1. Vertical lines are more optimally arranged for memory handling and faster reads. This is due to MATLAB’s column-major-order for the layout of multi-dimensional arrays in linear memory. This means the elements of each column are laid out in a contiguous block of memory making it fast to read a column in a single read, but slower to read a row, as it is a sequence of memory elements periodically spaced through the memory.

From Figs. 2 and 3 we can observe that it is still faster to use the memory bounded vertical line vHGW structuring element for sizes smaller than approximately 90 pixels, and the THT-vertical line for larger structuring elements.

The square structuring element is decomposed into the vHGW horizontal and the vHGW THT-vertical line structuring elements, thus mitigating the memory handling limitation in the vHGW vertical line case. Fig. 4 shows the performance results for the square structuring element and we observe the desired constant time results.

For diagonal lines as shown in Fig. 5, NPP results are shown both using a large square structuring element containing the entire diagonal line, and the decomposition of the diagonal line into a series of 3 by 3 masks containing a 3 pixel diagonal line. The large square structuring element is extremely slow, being slower than MATLAB once the line exceeds 50 pixels. The decomposition into multiple 3 by 3 masks is significantly faster, but still far slower than LTU-CUDA.

Fig. 6 shows the results for the disc structuring element with an average 14 times speed improvement over NPP.

## V. CONCLUSION AND FUTURE WORK

LTU-CUDA is a project to deliver open-source extensions for GPU enabled morphological image processing based on CUDA. It allows 32 bit processing of large scale images and structuring elements, unlimited by shared memory constraints in the CUDA architecture. It provides erosion and dilation for a base set of

structuring elements with the horizontal, vertical and  $\pm 45$  degree line structuring element operations based on the vHGW algorithm. The implementation highlights memory handling limitations of CUDA for vertical structuring elements. LTU-CUDA is available for download at <https://github.com/VictorD/LTU-CUDA>.

Further work includes but is not limited to;

- improvements to global memory handling to mitigate performance limitations for vertical structuring elements
- optimised algorithms for non-flat structuring elements
- structuring element discrete lines at arbitrary angles [20]
- implement automatic structuring element decomposition calculation for disc and octagon
- investigate shared memory usage for further improvements
- implement periodic line structuring elements

## ACKNOWLEDGMENT

The authors would like to thank the INTERREG IVA Nord programme of the European Union.

## REFERENCES

- [1] E. R. Dougherty and R. A. Lotufo, “Hands-on morphological image processing,” *Proc. SPIE—Int. Soc. for Opt. Eng.*, vol. TT59, 2003.
- [2] M. Vanherk, “A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels,” *Pattern Recogn. Lett.*, vol. 13, no. 7, pp. 517–521, Jul. 1992.
- [3] J. Gil and M. Werman, “Computing 2-D min, median, and max filters,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 15, no. 5, pp. 504–507, May 1993.
- [4] J. Gil and R. Kimmel, “Efficient dilation, erosion, opening, and closing algorithms,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 12, pp. 1606–1617, Dec. 2002.
- [5] P. Karas, “Efficient computation of morphological greyscale reconstruction,” in *Proc. 6th Doctoral Workshop Math. Eng. Methods Comput. Sci. (MEMICS’10)*, 2010, pp. 54–61.
- [6] L. Domanski, P. Vallotton, and D. Wang, “Parallel van herk/gil-werman image morphology on gpus using CUDA,” *NVIDIA GPU Computing Poster Showcase 2009* [Online]. Available: [http://www.nvidia.com/object/SC09\\_posters.html](http://www.nvidia.com/object/SC09_posters.html)
- [7] OpenCV Opencv gpu. [Online]. Available [Online]. Available: [http://opencv.willowgarage.com/wiki/OpenCV\\_GPU](http://opencv.willowgarage.com/wiki/OpenCV_GPU)
- [8] GpuCV, “Gpucv,” [Online]. Available: <http://picoforge.int-evry.fr/cgi-bin/twiki/view/Gpucv/Web/GpuCVDoc>
- [9] A. Reiner, Etotheipi CUDA-Image-Processing Sep. 2010 [Online]. Available: <https://github.com/etotheipi/CUDA-Image-Processing>
- [10] MATLAB version r2010a. Natick, MA, 2010 [Online]. Available: <http://www.mathworks.com>
- [11] Steve on Image Processing, Dilation With Linear Structuring Elements MATLAB Central, Dec. 2008 [Online]. Available: <http://blogs.mathworks.com/steve/2008/12/31/dilation-with-linear-structuring-elements/>
- [12] R. Adams, “Radial decomposition of disks and spheres,” *CVGIP: Graphical Models and Image Process.* vol. 55, no. 5, pp. 325–332, 1993 [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1049965283710242>
- [13] MVTEC-Software-GmbH Halcon [Online]. Available: <http://www.halcon.de>
- [14] M. Thurley and K. Ng, “Identifying, visualizing, and comparing regions in irregularly spaced 3D surface data,” *Comput. Vis. Image Understand.*, vol. 98, no. 2, pp. 239–270, Feb. 2005.
- [15] M. Thurley, “Automated online measurement of limestone particle size distributions using 3D range data,” *J. Process Control*, vol. 21, no. 2, pp. 254–262, 2011.
- [16] G. Ruetsch and P. Micikevicius, “Optimising matrix transpose in CUDA,” *NVIDIA*, Jan. 2009.
- [17] NVIDIA NVIDIA CUDA C Programming Guide—Version 4.2 NVIDIA developer website, Apr. 2012 [Online]. Available: <http://developer.download.nvidia.com>
- [18] Optimising CUDA—Part II NVIDIA developer website, 2009 [Online]. Available: <http://developer.download.nvidia.com>



- [19] NVIDIA developer website, version 2.1., 2010 [Online]. Available: <http://developer.download.nvidia.com>
- [20] P. Soille, E. Breen, and R. Jones, "Recursive implementation of erosions and dilations along discrete lines at arbitrary angles," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 18, no. 5, pp. 562–567, May 1996.



**Matthew J. Thurley** is an associate professor in industrial image analysis and founder of the industrial image analysis group at Luleå university of technology. He has a strong focus on machine vision research applied to fully automated on-line particle size measurement of overlapped particulate material. Matthew's focus is on industrial measurement projects and research in image analysis, morphological image processing, algorithms, and analysis of geometric structure.



**Victor Danell** is a student in computer science, electrical and space engineering. He has a strong interest in economics starting his university studies in this area but soon discovered a natural ability and passion for computer science.