Supplementary material for Paper 1175

1 Implementation for Double Filtering

The same kernel logic for FilterKernel2D can be invoked to accomplish the double filtering as shown in Algorithm 1. The additional kernel like ZeroingKernel and FilterKernel2DCol though trivial, are briefly explained here for more clarity. After the initial row filtering, the ZeroingKernel (Line 2) is used to update all the entries other than filtered entries as 0. This is followed by FilterKernel2DCol (Line 3) which extracts the extreme entries across columns and updates the lists F and L accordingly.

Algorithm 1 Double Filtering

1: FilterKernel2D(G, F, L, n)
2: ZeroingKernel(G, F, L, n)
3: FilterKernel2DCol(G, F, L, n)

2 Filtering parallel implementation 2D and 3D- Visualization

Figure 1: parallel filtering process in 2D- Competing entries for last locations marked in brown squares

Figure 1 represents the filtering process for *lastLocations*. In the matrix shown in Figure 1, the 3rd row has many entries with value 1, but only those with zero in the right cell compete for the minimum index calculation for the last location. Here, the entry in index $(2, 6)$ only qualifies, and therefore its column index 6 is updated into the *lastLocations* array, as shown in the right side. In the 2nd row, the entry in the index $(1, 7)$ is chosen, as it is the entry in the extreme right cell in that row. If there are no entries in a particular row, for example, in the 4th row of the Figure

1, the initialized value of -1 (initialization is done in Lines 1-2) will remain in *lastLocations*. FILTERKERNEL2D performs these computations along all the rows parallelly.

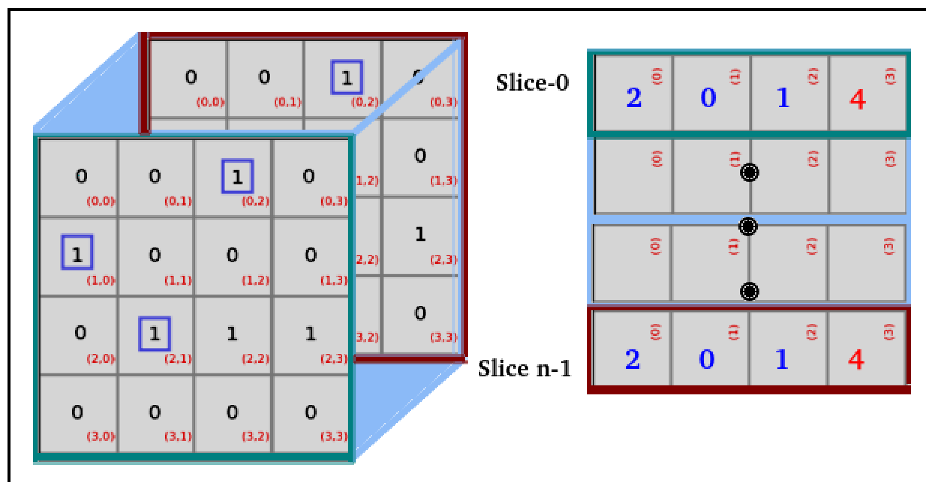

Figure 2: parallel filtering process in 3D

Parallel filtering process in 3D in a $(4 \times 4 \times 4)$ grid is depicted in figure 2. The corresponding first locations matrix, F of size (4×4) is shown in the right side,slice 0 and slice n-1 are only represented. The first location values in slice 0 along the rows are 2,0,1,–(no ON-entry in last row). Here slice n-1 is identical to slice 0. So the values in row-0 and row n-1 of F matrix are updated to 2,0,1,4. Here 4 is the initialized value in F, which remains as there is no ON-entry in the last row of slice-0 and slice n-1.

3 Parallel implementation of Kovalevsky's algorithm in 3D

Figure 3 depicts an exploded view of a $(3 \times 3 \times 3)$ voxel cube to list the 13 pairs of collinear neighbors of an interior input pixel. The interior input pixel is shown in yellow, and the neighbors in blue. It can be noted that there are 3 pairs of collinear neighbors along diagonally opposite faces, 6 pairs along edges and 4 pairs along vertices as depicted in 3. If a given input pixel, say P in a voxel representation, has at least one pair of collinear input voxels in its neighborhood, then P won't be a part of the CH. In this way, we eliminate such input pixels and apply a 3D incremental algorithm on the filtered input.

The compact version of the parallel algorithm for eliminating such input voxels is shown in algorithm 2. Note that here, there is an overhead of transferring two matrices to the kernel, both representing the input. The first one is used for collinearity check of an input voxel along all possible directions as shown in line 6 of algorithm 2.

This involves checking along all directions(up to 13 directions) for all input pixels in parallel. Now if the input pixel is found to be collinear, it wouldn't be a candidate for the CH and therefore, it will be updated as 0. This filtering method works well only if the input contains large number of connected voxels or in other words the input voxel matrix is highly dense. The filtered voxels

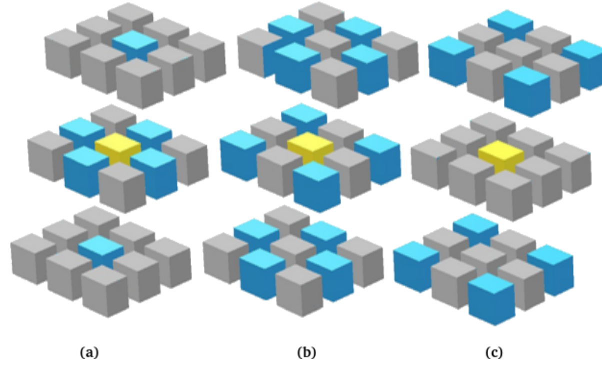

Figure 3: Exploded view of a $(3\times3\times3)$ voxel cube to depict 26 neighbors of an interior input voxel. Input voxel is shown in yellow and its neighbors in blue (a) 6 neighbors along 3 pairs of opposite faces (b) 12 neighbors along 6 pairs of diagonally opposite edges (c) 8 neighbors along 4 pairs of diagonally opposite vertices

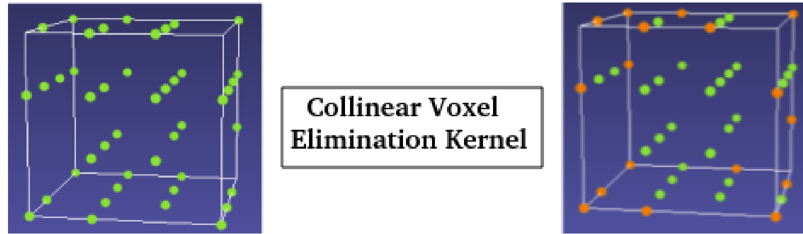

Figure 4: Illustration of Kovelevsky's filtering method, Input voxels are shown in green and the filtered voxels are shown in orange.

Algorithm 2 Collinear Voxel EliminationKernel(GPU)

Input: 3D input boolean matrix $gridmat_in$ of size $(n \times n \times n)$ and a copy of the same in $gridmat_out$
Output: Boolean matrix $gridmat_out$ of $(n \times n \times n)$ representing the filtered points.

1: **function** NNFilterKernel3D($gridmat_in$, $gridmat_out$, n)
2:     **for** each element of $gridmat_in$ **do** ▷ Parallel
3:         Let row index be row and column index be col
4:         $index \leftarrow row \times n + col$
5:         **if** $gridmat[index] = 1$ **then**
6:             **if** IsCollinear ($gridmat_in$,index) **then**
7:                 $gridmat_out[index] = 0$
8:             **end if**
9:         **end if**
10:     **end for**
11: **end function**

are then fed to a 3D incremental CH algorithm to obtain the final hull. The filtering method is illustrated in figure 4. The input voxels are shown in green and the filtered voxels are shown in orange.

4 Additional results on running time, speedup and filtering

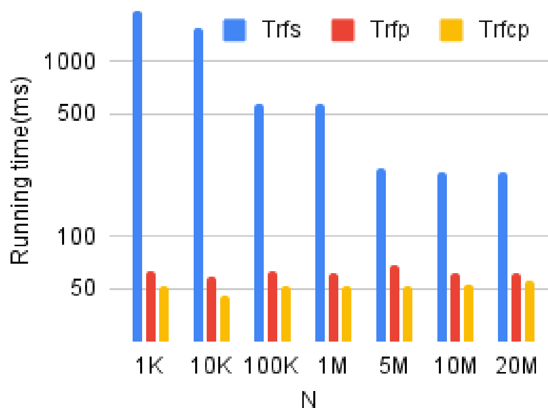

Figure 5: Running time comparison among row filtering serial version(RFS), row filtering parallel version(RFP), row filtering- cell-wise parallel version(RFCP) on $20K \times 20K$ space.

Running time comparison among our methods, namely row filtering serial version (RFS), row filtering parallel version (RFP), and row filtering- cell-wise parallel version (RFCP) for circle shape sampled from $20K \times 20K$ grid space is shown in Figure 5. A similar trend is observed for shell shape as well. Note that the running time of RFS decreases as the number of input points increases, infact RFS converges faster for the square shape. This is because as the grid becomes denser, extracting the extreme entries becomes faster. Additional results for speedup in 2D and 3D is provided in Table 1 and Table 2

Table 1: Timing and speedup comparison of CGAL-Akl and Matlab Bwconvhull against RFS and RFCP on input points sampled from a $20K \times 20K$ matrix. Notations used here are N: Total number of input points, Takl: Running of CGAL-Akl algorithm (on input points directly provided as (x, y) coordinates), Tpre: Pre-processing time elapsed in reading coordinates from the boolean matrix, Ttot:= $Takl + Tpre$, i.e., the running time of akl inclusive of pre-processing time, T_{bw}: Running time for bwconvhull, T_{RFCP}: Running time for RFCP, T_{RFS}: Running time for RFS, S_{rfs-bw}: Speedup of RFS against bwconvhull, $S_{rfs-akl}$: Speedup of RFS against CGAL-Akl $S_{rfcp-akl}$: Speedup of RFCP against CGAL-Akl, (All running times are in milliseconds.)

Shape	**N**	**CGAL-Akl**			T_{bw}	T_{RFS}	T_{RFCP}	S_{rfs-bw}	$S_{rfs-akl}$	$S_{rfcp-akl}$
		Tpre	**Tch**	**Ttot**						
Square	1M	970	144.32	1114.32	2380	54	52	44	21	21
	5M	1149	815.9	1964.9	3735	19	52	197	103	38
	10M	1332	1696.6	3028.6	5402	11	54	491	275	56
	20M	2099	3455	5554	8766	9	59	974	617	94
Circle	1M	1008	67.57	1075.57	2554	572	51	4	2	21
	5M	1140	350	1490	18808	242	52	78	6	29
	10M	1377	757	2134	67660	231	53	293	9	40
	20M	1475	1510	2985	270979	232	55	1168	13	54
Ring	1M	1070	51.73	1121.73	560224	228	50	2457	5	22
	5M	1194	327.59	1521.59	XX	231	50	XX	7	30
	10M	1355	795.27	2150.27	XX	226	52	XX	10	41
	20M	1559	1369.19	2928.19	XX	230	51	XX	13	57

Table 2: Timing and speed up a comparison for voxelized point cloud representation of benchmark shapes: Tqtot: total time taken for CGAL Quickhull(3D), $T_{sl(CPU)}$ and $T_{sl(GPU)}$: total time taken for SLICER3D(CPU) and SLICER3D(GPU) respectively, T_{kov}: total time taken for KOV3D GPU implementation $S_{sl(CPU)}$: speedup of SLICER3D(CPU), $S_{sl(GPU)}$: speedup of SLICER3D(GPU), S_kov: speedup of KOV3D.(All timings are in milliseconds)

Shape	ρ	**N**	$\mathbf{T_{qhull}}$	$\mathbf{T_{sl(CPU)}}$	$\mathbf{T_{sl(GPU)}}$	$\mathbf{T_{kov}}$	$\mathbf{S_{sl(CPU)}}$	$\mathbf{S_{sl(GPU)}}$	$\mathbf{S_{kov}}$
Cube (300)	0.1	2.7M	1579	420	231	962	3.76	6.84	1.34
	0.5	13.5M	6015	1012	276	164	5.94	21.79	14.05
	0.8	21.6M	8712	1195	280	25	7.29	31.11	29.84
	1	27M	9962	1308	272	12	7.62	36.63	36.63
Sphere (401)	0.1	3.35M	4465	969	358	4001	4.61	12.47	1.06
	0.5	16.75M	24335	1512	518	1280	16.09	46.98	15.82
	0.8	28.8M	31003	1773	530	324	17.49	58.5	53.09
	1	33.5M	39287	1965	582	278	19.99	67.5	70.03
Shell (441)	0.1	1.44M	1391	1123	393	1605	1.24	3.54	0.76
	0.5	7.24M	8229	1555	480	557	5.29	17.14	10.25
	0.8	10.13M	11113	1645	475	389	6.76	23.4	17.81
	1	14.48M	15374	1779	508	325	8.64	30.26	27.07

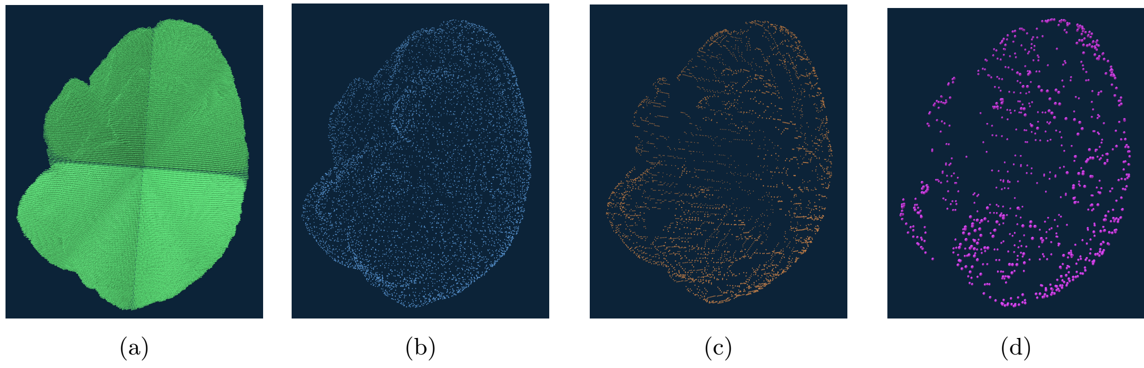

Figure 6: Filtering of a Brain Tumor Diagnosis CT Scan acquired from an open source DICOM library (a) input voxel grid (1.3M) (b) After Kovalevsky's filtering (6.69KM) (c) After slice-wise CH filtering(4.38K) (d) Voxels in convex hull(836)