Title:
Fast and Cross-vendor OpenCL-based Implementation for Voxelization of Triangular Mesh Models

Authors:
Mohammadreza Faieghi, mfaieghi@uwo.ca, Western University, London, Canada
O. Remus Tutunea-Fatan, rtutunea@eng.uwo.ca, Western University, London, Canada
Roy Eagleson, eagleson@uwo.ca, Western University, London, Canada

Keywords:
voxelization, OpenCL, general purpose graphical processing unit (GPGPU), parallel computing

DOI: 10.14733/cadconfP.2017.410-414

Introduction:
Voxel based modelling of geometrical objects is receiving a lot of interest over the recent years. The simple topology of voxel-based models makes them an effective alternative to triangular meshes in many applications, including simulation of material removal [4], 3D printing [7], generation of porous surfaces [2], etc. However, many applications continue to rely solely on triangular mesh models and therefore voxelization is often required to convert them into alternate voxel-based representations.

As shown in Fig.1, the core of the voxelization process essentially consists of the calculation of the intersection between the triangular facets of the mesh and a 3D grid obtained through the discretization of the bounding box of the tessellated geometry. This process involves numerous iterative procedures and thereby is computationally demanding, particularly for small-sized meshes and/or high-resolution voxel grids. To address this challenge, earlier studies made efforts to use the graphics pipeline – that is available in virtually all commercial graphics cards – in order to achieve fast voxelization during rendering passes [3, 8]. However, the fixed functions of the graphics pipeline led to inaccurate results in some applications [5] such that it was only after the relatively recent involvement of the general-purpose computing performed on the graphics processing unit (GPGPU) when accurate yet fast voxelization became possible. To exemplify, NVIDIA's compute unified device architecture (CUDA) platform [1] was used in [5] to achieve precise voxelization runtime with one order of magnitude faster than the conventional pipeline method [8]. However, since the applicability of CUDA is restricted to NVIDIA products, the development of fast, accurate and vendor-independent voxelization tools continue to remain a valid investigational objective.

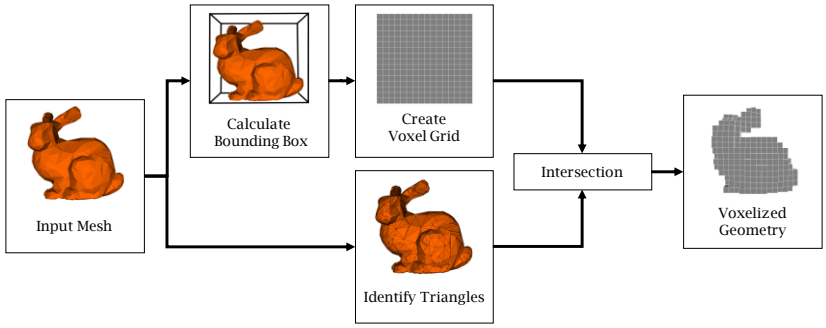

Fig. 1: Phases of the voxelization process.

Along this line of thoughts, OpenCL [6] represents an alternate GPGPU platform that can be used by a broad range of graphics hardware manufacturers. However – unlike CUDA – OpenCL has an inherent verbose nature and probably this made it somewhat less attractive for parallelization tasks. Building on this, the present study represents one of the first attempts made to test the viability of an OpenCL-based voxelization engine in the context of non-NVIDIA hardware. It is also important to note here that the broader context of the current study is the development of a real-time graphic rendering module for a prototypical orthopedic virtual simulator to be used for training purposes.

OpenCL Framework:
A typical OpenCL program primarily consists of host and device applications (Fig. 2). In most common implementations, the *device* is a GPU that conducts the parallel computing task – termed as kernel – and is programmed in the OpenCL platform. By contrast, CPU represents the *host* – controlling the device and data flow by means of command queue – that is developed on the common C language. The kernel input data flows from host memory (RAM) to device global memory (*i.e.*, the video memory of graphics card). The kernel divides the computing task at hand into multiple work-groups, each assigned to a processing core of the GPU. The work groups are further separated into multiple work-items that can be computed concurrently. The device relies on different memory levels for work-items and work-groups (i.e., private vs. local memory). Both types are much faster, but limited in size when compared to the global memory. Therefore, an efficient OpenCL code has to minimize the number of accesses to the global memory. In an ideal situation, this should happen only twice: when reading the input data and when writing the output. However, this goal may be hindered by limited local and private resources.

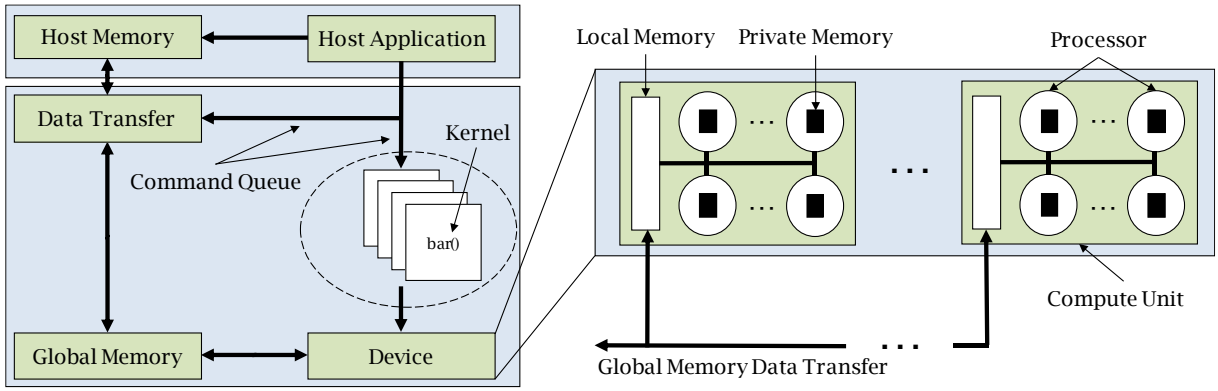

Fig. 2: OpenCL device architecture.

Implementation:
Upon importing a model $\mathcal{M}$ with n number of triangles, vertex coordinates are stored in an 1D float array of length $9n$. The elements $9i$ to $9i+8$ correspond to the vertices of the i-th triangle. This enables a grouped storage of the facet coordinates that in turn guarantees a coalesced memory access constituting one of the prerequisites of high-performance. The voxel grid $\mathcal{G}$ is constructed by means of the axis-aligned bounding box (AABB) technique applied on $\mathcal{M}$, followed by a further subdivision into identical cubes with a preset size. The resulting voxels are indexed by means of integer triplets (x,y,z) ranging from $(0,0,0)$ to (d_x-1, d_y-1, d_z-1) and this enables an easy access to the spatial information within the grid. The grid itself is uniquely identified by three parameters including: the minimum corner of the grid $\mathbf{p}_0 \in \mathbb{R}^3$, the number of voxels in each direction $\mathbf{d} \in \mathbb{N}^3$ and the space diagonal of the voxels $\Delta\mathbf{p} \in \mathbb{R}^3$. The mapping between the voxel grid and a contiguous chunk of memory is used to store the voxel material information in an array of length $d_x d_y d_z$ where the voxel $\mathcal{V}(x,y,z) \in \mathcal{G}(\mathbf{p}_0, \mathbf{d}, \Delta\mathbf{p})$

occupies location $i = x + yd_x + zd_x d_y$ of the array. Conversely, for a given i, voxel indices are determined by sequentially solving $z = i / d_x d_y$, $y = \left(i - zd_x d_y \right) / d_x$ and $x = i - zd_x d_y - yd_x$.

The voxelization kernel launches one work-item per triangle of $\mathcal{M}$. Each work-item starts with the coordinates of an assigned triangle, calculates the corresponding AABB lying within $\mathcal{G}$ and then inspects the intersection of each AABB voxel with the corresponding triangular facet of the mesh. Since the voxels located outside of AABB are – by default – in a disjoint condition with the triangular facets, the number of intersection tests will be reduced significantly when compared to the case of launching one work-item per voxel of $\mathcal{G}$ (Tab. 1, Stanford bunny case study). Furthermore, each work item requires only the coordinates of a single triangle from the whole $\mathcal{M}$ data available in the global memory. This permits the transfer of the entire data associated with a work-item to local and private memory and in turn, this limits the number of accesses of the global memory to two. The implemented intersection test detects overlaps through a four-step process, one of which checks if triangle's face overlays with the voxel and the rest evaluate if the projections of the triangle and the voxel are coincident with XY, XZ and YZ planes [5]. As soon as one of these steps fails, the assessed triangle/voxel pair is deemed as being in a non-intersecting condition.

Voxel Size (mm)	*Number of Intersection Tests*		
	Triangle-Based (TB)	*Voxel-Based (VB)*	*Order of Magnitude Ratio (VB/TB)*
3	93	1,767,431	4.2789
2	224	4,615,507	4.3140
1	1,107	44,406,021	4.6033
0.5	5,874	368,832,950	4.7979

Tab. 1: Comparison between triangle- and voxel-based parallelization schemes.

The technique described above was implemented into a high-performance OpenCL-based voxelization algorithm. As shown in Fig. 3, the OpenCL program starts with the identification of a device to be followed by kernel compilation in order to build a device-specific executable application. Meantime, the computing capabilities of the device are identified since they are required during the parallelization process. After OpenCL buffers and pointers have completed the host to global memory data transfer, the kernel is queued and the results are transferred to the host memory through the memory mapping technique. This data can be then further used in export operations or for graphical rendering purposes.

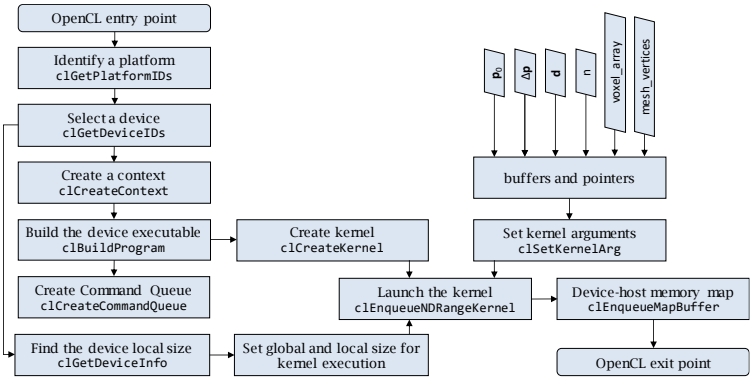

Fig. 3: Core structure of an OpenCL algorithm to implement parallelization in the context of a voxelization task.

Results:
Numerical experiments have been carried out to evaluate the efficiency of the developed OpenCL-based routine. The benchmarked models include both fine and coarse meshes with different domain sizes (Fig. 4). To assess hardware performance, four different graphics hardware configurations were tested: NVIDIA GeForce 970 GTX (desktop), GeForce 960 GTXM (laptop), AMD Radeon R7 240 (desktop) and Intel HD Graphics 530 (integrated laptop). For this purpose, the execution time of the voxelization kernel was assessed – by means of the built-in profiling features of OpenCL, essentially allowing a CPU/RAM decoupled-evaluation – for various voxel sizes of each of the geometric models. These results were further compared with a baseline yielded by a single-thread implementation ran on an i7-6700K processor with 16GB DDR4 RAM.

894 triangles (63.17, 39.31, 29.48) (a) | 16301 triangles (15.55, 15.33, 12.06) (b) | 100000 triangles (56.37, 25.21, 39.76) (c) | 4006 triangles (21, 36.47, 36.50) (d) | 470340 triangles (144.76, 137.24, 73.28) (e)

Fig. 4: Benchmarked models (including mesh and domain sizes): (a) Utah teapot, (b) Stanford bunny, (c) Stanford dragon, (d) surgical tool/reamer, and (e) scapula.

Model	Voxel Size (mm)	Voxel Grid Dimension	Voxelization Time (ms)				
			GeForce 970 GTX	GeForce 960 GTXM	Intel HD Graphics 530	Radeon R7 240	Single-thread
Utah Teapot	0.5	127, 79, 59	2.12	2.07	2.4	6.49	21
	0.25	253, 158, 118	13.57	13.1	14.66	25.9	108
	0.1	632, 394, 295	166.52	164.68	151.95	315.31	1081
Stanford Bunny	0.5	32, 31, 25	0.05	0.09	0.11	0.21	13
	0.25	63, 62, 49	0.08	0.16	0.19	0.22	18
	0.1	156, 154, 121	0.32	0.66	0.86	1.81	65
Stanford Dragon	0.5	113, 51, 80	0.31	0.72	0.76	1.77	85
	0.25	226, 101, 160	1.09	2.75	2.42	4.04	159
	0.1	564, 253, 398	10.17	25.55	19.44	34.52	757
Surgical Tool	0.5	42, 73, 73	0.77	0.75	0.82	2.46	25
	0.25	84, 146, 146	5.32	5.13	5.56	8.51	61
	0.1	211, 365, 366	71.89	68.84	69.42	109.23	412
Scapula	0.5	290, 275, 147	2.31	5.44	3.8	5.47	420
	0.25	580, 549, 294	11	25.79	13.3	24.14	868
	0.1	1488, 1373, 733	112.66	283.97	82.10	256.34	4913

Tab. 2: Voxelization time associated with different models.

The results presented in Tab.2 suggest that Utah teapot – despite having less facets – requires longer computation time than Stanford models, probably because of its larger domain. Interestingly, the analyzed surgical tool requires a voxelization time of almost six times larger than that required by the Stanford dragon and this is likely a consequence of the geometrical/topological complexity of the reamer. While the results suggest that both the attributes of the geometry and the performance of the graphics hardware play a significant role on the voxelization time, it can also be inferred that the parallelized OpenCL implementation clearly outperforms the traditional single-thread method.

To further this idea, Fig. 5 ranks the average computing time across models and hardware by normalizing it to the time required by the single-thread CPU. The acquired results suggest that while a minimum 70% speed enhancement is observed for Radeon R7 240, the integrated Intel GPU performs slightly better. Even though the speed varies among the benchmarked models and graphics hardware, it

is evident that GPU-based parallelization can speed up voxelization with up to 99% (Stanford bunny on a GeForce 970 GTX).

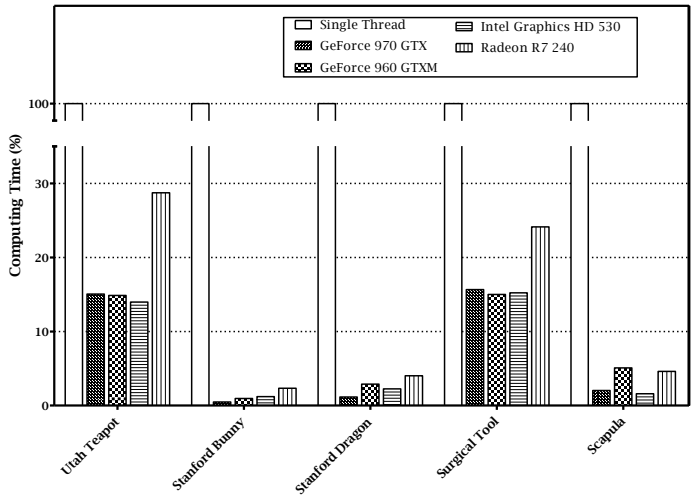

Fig. 5: Normalized representations of the voxelization time for OpenCL and single-thread based implementations.

Conclusion:

The aim of this study was to investigate the feasibility and effectiveness of an OpenCL implementation for high-performance and cross-platform voxelization of tessellated models. The numerical experiments conducted have revealed that while the performance of the developed OpenCL-based algorithm is dependent on the geometry of the model and computer hardware, a significant improvement is obtained in all cases when compared to a single-thread CPU-based implementation.

Acknowledgement:

The work presented in this study was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada and Canadian Institutes of Health Research (CIHR) under the framework of the Collaborative Health Research Program (CHRP).

References:

[1] NVIDIA: CUDA Toolkit Documentation v8.0, http://docs.nvidia.com/cuda
[2] Dinis, J. C.; Moraes, T. F.; Amorim, P. H. J.; Moreno, M. R.; Nunes, A. A., Silva, J. V. L: POMES: An Open-source Software Tool to Generate Porous/Roughness on Surfaces. Procedia CIRP, 49, 2016, 178–182. http://dx.doi.org/10.1016/j.procir.2015.07.085
[3] Eisemann, E.; Décoret, X. Fast Scene Voxelization and Applications. In proceeding of ACM SIGGRAPH 2006, Boston, Massachusett, Article 8 http://dx.doi.org/10.1145/1179849.1179859
[4] Razavi, M.; Talebi, H. A.; Zareinejad, M.; Dehghan, M. R.: A GPU-implemented physics-based haptic simulator of tooth drilling, The International Journal of Medical Robotics + Computer Assisted Surgery: MRCAS, 11(4), 2015, 476–485. http://dx.doi.org/10.1002/rcs.1635
[5] Schwarz, M.; Seidel, H.-P: Fast parallel surface and solid voxelization on GPUs. ACM Transaction on Graphics, 29(6), 2010, Article 179. http://dx.doi.org/10.1145/1882261.1866201
[6] KHRONOS group: The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/
[7] Vidimce, K.; Wang, S.-P.; Ragan-Kelley; J.; Matusik, W: OpenFab: A Programmable Pipeline for Multi-Material Fabrication, ACM Transactions on Graphics, 32(4), 2013, Article 136. http://dx.doi.org/10.1145/2461912.2461993
[8] Zhang, L.; Chen, W.; Ebert, D. S.; Peng, Q: Conservative Voxelization. The Visual Computer, 23(9), 2007, 783–792. http://dx.doi.org/10.1007/s00371-007-0149-0