AN OPTIMIZED IMPLEMENTATION OF THE MARCHING CUBES ALGORITHM

Bo Li, Yifan Yu, Yitian Ma

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We propose an optimized implementation of the Marching Cubes. Marching Cubes is a famous algorithm in computer graphics for extracting isosurface from scalar fields. The standard implementation of the Marching Cubes suffers from redundant computation and other inefficiencies. In this work, we reorganize the structure of the algorithm to eliminate redundancy and improve efficiency. We also integrate SIMD vectorization and other optimization techniques. Experiment results show that our optimization could achieve 2-3x speedup compared to the naive implementation. The optimization proposed is general and should be effective for any resolution and specific values of the input data.

1. INTRODUCTION

The Marching Cubes (MC) algorithm is widely used in computer graphics and other related fields. It solves the problem of extracting an isosurface in the form of a triangle mesh from a 3D (volumetric) scalar field. Due to the generality of this problem, the MC algorithm serves as the foundation of many applications concerning isosurface reconstruction. With the continuing increase of the resolution of input data and the output mesh, the performance of the MC algorithm becomes increasingly critical to related applications.

Contribution. The original MC algorithm[1] and many open-source implementations use a straightforward cube-by cube strategy, which have many redundant computation and impedes vectorization. In this work, we present an optimized implementation of the MC algorithm that marches level-by-level on the grid. It utilizes the feature of the grid structure to store, reuse, and share intermediate results in order to avoid redundant computation. Based on this strategy, SIMD capability of modern processors could be integrated to further accelerate the computation. We test our optimization in experiments to demonstrate the performance gain compared to the baseline implementation. Finally, we use external profiling tools to analyze the performance and locate bottlenecks for further potential optimization. Our optimization is based on the general case of the MC algorithm and does not involve complex data structures. Thus it should provide a performance gain on input data of any resolution and values.

Related work. A lot of improvements and extensions have been proposed since the invention of the vanilla Marching Cubes algorithm. To improve efficiency, many efforts try to eliminate computing cubes that are not intersected by the isosurface, including octree-based [2], interval-based [3, 4] and other methods. However, due to the additional cost of establishing these data structures, whether they could help depends on specifics input values. Only those scalar fields with a sparsely triangulated isosurface may benefit from these methods.

Recently, many researches seek to utilize the SIMD and multi-core compute power of modern processors [5, 6, 7]. Although parallel computing is a trend, we still find it important to optimize the single-core CPU implementation, which could be potentially extended to parallel implementations.

2. BACKGROUND

In this section, we provide some background information about the Marching Cubes algorithm. For more details, variants and extensions of the algorithm, we refer the readers to a comprehensive survey by Newman et al. [8].

Marching Cubes Algorithm. Given a scalar field $f : \mathbb{R}^3 \mapsto \mathbb{R}$ with values discretely evaluated on a 3D grid, and a certain threshold value f_0, the MC algorithm reconstructs the isosurface $f(x, y, z) = f_0$ as a triangle mesh. Fig. 1 gives an illustration of the input and output of the MC algorithm.

The name of the Marching Cubes algorithm comes from the way it produces the triangle mesh: by "marching" through every cube within the grid, and determine for each cube how should the isosurface intersects with it given the values at each of the 8 grid points. Based on the comparison of the isovalues and f_0, there are a total of $2^8 = 256$ possible states of a cube (each vertex could have a larger or smaller isovalue than f_0 hence 2 states per vertex). Each state corresponds to a intersection pattern of the isosurface and the cube, which are pre-defined and directly used to generate the triangles of the mesh. One example is shown in Fig. 2a.

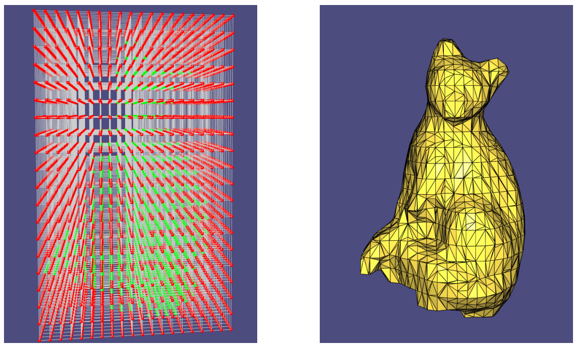


Fig. 1: The MC algorithm extracts the isosurface from a scalar field defined with the signed distance function (SDF) of a cat model. (a) The input: discrete SDF values of thecost-efficientlar field on a grid. Red vertices have positive (outside) closest distance to the surface, while green ones are negative (inside). (b) The output: reconstructed isosurface with isovalue of 0.

Applications. The equation $f(x, y, z) = f_0$ is an implicit representation of a surface. Implicit surface is a commonly used mathematical tool in computer graphics. Therefore the MC algorithm has great importance in applications: it provides a method to map the implicit representation of a surface to its discrete geometric representation. It is widely used in many problems such as fluid simulation [9], shape reconstruction from point clouds [10], medical CT/MR imaging [11], etc.

Cost Analysis. We do not use floating point operations (flops) to measure the cost. In the MC algorithm, flops are only prevalent in the stages of threshold comparison and vertex interpolation (see Section 3.1). There are also a lot of int ops, branching, memory reads and writes so that only counting flops is not reasonable. The actual cost of the algorithm is two-fold: the first stage (see Section 3.1) is performed for all the cubes, but the complexity of the second and third stage depends on specific values of the input data. An extreme example would be when the entire grid is fully inside or outside the isosurface, resulting in no triangle generated at all. Because the bottleneck of our optimized implementation mainly lies in the second and third stage (see Section 5), we choose the number of triangles to measure the cost, and divide it by the runtime as the performance.

3. BASELINE IMPLEMENTATION AND OPTIMIZATIONS

In this section, we first introduce the standard implementation of the MC algorithm in the original paper [1], and describe our optimization methods in detail.

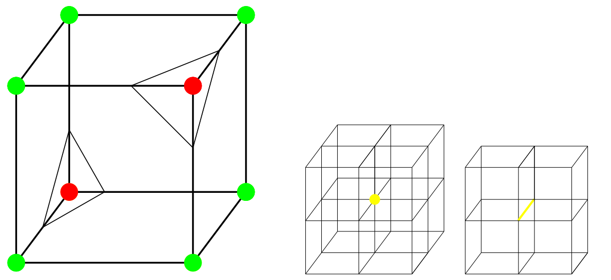


Fig. 2: (a) An illustration of polygonizing the isosurface inside a cube. Each cube of the grid is associated with 8 grid points and 12 edges. In this example, green grid points and red ones are inside and outside the isosurface respectively. The isosurface intersects with an edge if its two points have opposite values compared to the threshold. (b) Redundant computation resulted from the cube-by-cube manner. Each non-boundary vertex is compared with the threshold 8 times in each of its 8 adjacent cubes, while each non-boundary edge is interpolated 4 times in each of its 4 adjacent cubes.

3.1. Baseline Implementation

The baseline implementation uses a straight-forward cube-by-cube iteration approach. In general, the procedure could be expressed in 3 stages as shown in the pseudocode of Algorithm 1. The 3 stages are sequential for each cube, and each stage requires the result from the previous stage.

Algorithm 1 Baseline cube-by-cube implementation

```
Require: $f_0$                                  ▷ The given threshold value
  for each cube $c$ do
      cubeIndex $\leftarrow 0$
      for each vertex $v$ of $c$ do
          compare $f(v)$ with $f_0$                ▷ Stage 1
          update cubeIndex
      end for
      for each edge $e$ of $c$ do
          Interpolate the vertex if needed       ▷ Stage 2
      end for
      for each interpolated triangle $t$ do
          append vertices to buffer              ▷ Stage 3
      end for
  end for
```

The first stage, **threshold comparison** or **thresCmp** in short, compares the values of each of the 8 grid points with the given threshold f_0, and uses the comparison result to produce a `cubeIndex`. `cubeIndex` is a 8-bit integer, with each bit encoding whether the corresponding grid point has a larger value than f_0. Thus, there are 256 possible cases for a cube.

The second stage, **vertex interpolation** or **vertexInterp**, uses the `cubeIndex` to index into a pre-defined 256-element integer array `edgeTable`. An element in `edgeTable` has 12 effective bits, and it tells whether each edge intersects with the isosurface. For the intersected edges, an vertex is linearly interpolated based on the values of the two end points:

$$v = v_a + \frac{f_0 - v_a}{f(v_b) - f(v_a)}(v_b - v_a) \qquad (1)$$

The third stage, **triangle assembling** or **triAssem**, collects the interpolated vertices into triangles. By indexing with `cubeIndex` into a `triTable`, we can get a list of the edge (vertex) indices that form the triangles within the cube. We then fetch the interpolated vertices according to these indices and append the vertices to a global buffer as the final output.

The execution of these three stages in a cube is illustrated in Fig. 2a. The MC algorithm involves information on cubes, grid points and edges. The interaction between these three types is centered around cubes: `cubeIndex` relies on grid points, and decides vertex interpolation. This heterogeneity impedes some optimization such as vectorization.

Another important observation from the baseline implementation is that there are a lot of redundant computation when performing the 3 stages cube-by-cube. As shown in Fig. 2b, **thresCmp** could be redundant up to 8x and **vertexInterp** up to 4x. This inefficiency calls for optimization to avoid these redundant computation by finding a way to reuse the intermediate threshold comparison and vertex interpolation results.

3.2. Level-by-level Optimization

The key idea of our first optimization is to store and reuse the intermediate computation (threshold comparison and vertex interpolation) results. Intuitively, the **thresCmp** results could be stored and reused on grid points, and **vertexInterp** results on edges.

Cube-by-cube iteration is intrinsically not suitable for this reusing, and it is not cost-efficient to allocate a grid of the same size as the input to store intermediate results. Instead, the intermediate result could be shared *within* a block of cubes. To avoid duplicate vertex interpolations, most cubes (except ones on the grid boundaries) only need to interpolate 3 edges and most edges within the block could be covered (except for block boundaries). However, using a block of arbitrary size will lead to complex sharing *between* blocks. For instance, a grid point could belong to 1 (within the block), 2 (on the block boundary), 4 (at block corners that are on the grid boundary), or 8 (at block corners that are within the grid) blocks. Similarly, an edge could belong to 1, 2 or 4 blocks. Enumerating all corner cases is

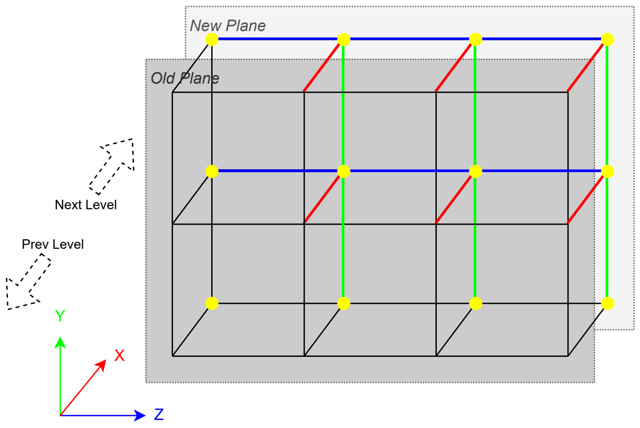


Fig. 3: An illustration of the level-by-level optimization. The current level of cubes contact with its two neighbors on two planes. When iterating over the current level, most cubes only need to do the vertex interpolation on 3 edges (colored with colors corresponding to the axes). Threshold comparison and vertex interpolation (on YZ edges) results are only written on the new plane, while the results on the old plane is inherited from previous level. Two ping-pong buffers are used, each one serves for a plane and alternate for odd and even levels. By marching through all the levels we could process all cubes without any redundancy.

unnecessarily complex and difficult to debug. Instead, we choose levels, a special type of blocks which corresponds to a layer of cubes which contains all cubes with the same X coordinate but differ in YZ coordinates. This level-by-level approach iterates the 3 stages on each level instead of each individual cube. Within each level, the `thresCmp` results are shared on 2 layers of vertices, and the vertexInterp results are shared on 2 layers of edges on YZ directions, and 1 layer of edges on the X direction.

Sharing between levels is straightforward. Grid points of a level are located on two planes. For each level (except for the first), We can fetch threshold comparison results from the plane that is shared with the previous level (old plane), and only compute for the new plane. Moreover, each cube only interpolates one Y and one Z edge on the new plane, as well as one X edge which is not shared between levels. In this way, redundant computation is completely avoided and sharing at level boundaries is elegantly handled. Each stage's homogeneous computation inside all cubes of a level are grouped and performed together, Which paves way for further SIMD vectorization.

3.3. Other Optimization Tricks

There are some noteworthy implementation details in the level-by-level optimization. These tricks are important for maximizing its performance.

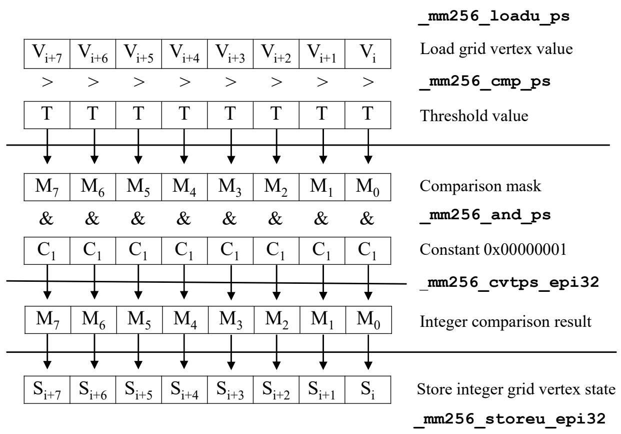
Indexed Mesh. During the vertex interpolation stage, we do not really store the 3D coordinate of an interpolated vertex on an edge; instead, we append the newly generated vertex into a global buffer and then store the index of that vertex on the edge. In this way, we can finally output an indexed mesh: each vertex position will be stored exactly once, and each triangle is identified by 3 indices. In the baseline implementation, since a cube does not know anything about its neighbors, the 3D coordinate of the same vertex are repeated for each triangle. The cube-by-cube implementation results in a much more redundant output data compared to an indexed mesh.

Branching Elimination. At **triAssem** stage, we need to fetch back the vertex (index) from 12 potential edges within a cube. In the cube-by-cube implementation this is not a problem, since we could allocate an 12-size array to accommodate the **vertexInterp** results. However, in the level-by-level implementation, 12 edges of a cube are not continuously stored, since all edges in a level are organized together. If we use this approach in level-by-level implementation, it would require conditionals on the `triTable` entries, as well as the parity (odd or even) of current level, to find the correct offset in the corresponding buffer to retrieve the vertex. Because the order of triangle indices have no temporal pattern, the branch predictions here almost always fail, thus significantly hinder the performance. We first tried use `swtich` statement with 12 cases, which turned out to actually hurt the performance despite the many redundant computation reduced. Our solution is to concatenate the buffers that store the intermediate interpolation results of edges, and pre-calculate a lookup table with 24 entries corresponding to 12 edges of a cube and the parity of level. Doing so allows us to directly lookup the offset from the `triTable` entry and compute using the cube location in the level to retrieve the correct interpolated vertex index. This trick produces approximately 2x speed-up comparing to the "switch" implementation on random inputs.

3.4. SIMD Vectorization

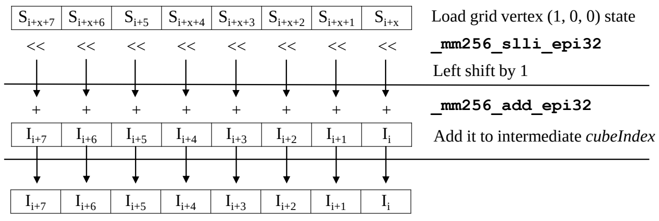
In this section, we introduce vectorization to improve the performance using Intel AVX/AVX2 intrinsics. The vectorization is performed to accelerate homogeneous operations of the same stage for all cubes in a level.

The first part of vectorization happens in **thresCmp** stage. The grid point values and threshold value are stored as single-precision floating point numbers. AVX/AVX2 can process 8 single-precision floating point numbers or 8 32-bit integers simultaneously. During this stage, every 8 grid point values along the Z-axis are loaded, which are stored consecutively in memory. The procedure is shown in Fig. 4. After the floating point comparison, the masks are converted to 0 and 1 integer values to facilitate the computation of the next stage.



_mm256_loadu_ps
V_{i+7} V_{i+6} V_{i+5} V_{i+4} V_{i+3} V_{i+2} V_{i+1} V_i
Load grid vertex value
_mm256_cmp_ps
T T T T T T T T
Threshold value
M_7 M_6 M_5 M_4 M_3 M_2 M_1 M_0
Comparison mask
_mm256_and_ps
C_1 C_1 C_1 C_1 C_1 C_1 C_1 C_1
Constant 0x00000001
_mm256_cvtps_epi32
Integer comparison result
S_{i+7} S_{i+6} S_{i+5} S_{i+4} S_{i+3} S_{i+2} S_{i+1} S_i
Store integer grid vertex state
_mm256_storeu_epi32

Fig. 4: Vectorization of threshold comparison

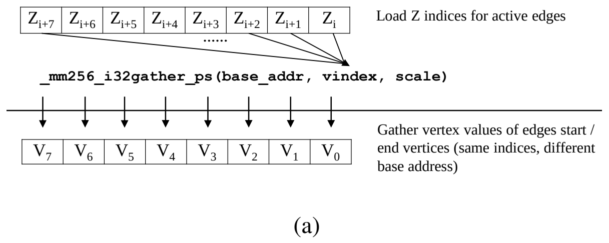


S_{i+x+7} S_{i+x+6} S_{i+5} S_{i+x+4} S_{i+x+3} S_{i+x+2} S_{i+x+1} S_{i+x}
Load grid vertex (1, 0, 0) state
_mm256_slli_epi32
Left shift by 1
_mm256_add_epi32
I_{i+7} I_{i+6} I_{i+5} I_{i+4} I_{i+3} I_{i+2} I_{i+1} I_i
Add it to intermediate *cubeIndex*

Fig. 5: Determine `cubeIndex` based on the cube vertices' `thresCmp` state. Here grid vertex at position (1, 0, 0) is used as example. We repeat this for all 8 cube vertex positions.

The second part is to vectorize the computation of the `cubeIndex` for each cube based on the **thresCmp** results of its 8 vertices. We work on 8 consecutive cubes along Z-axis. For these 8 cubes, we need to load 8 consecutive **thresCmp** results for one cube vertex position. We repeat this for each cube vertex position. Then the 0/1 state value for each cube vertex position is left shifted by a different number of bits according to the pattern of a topology look up table. The process of calculating the contribution of one cube vertex is shown in Fig. 5. Finally for each cube we add up the contribution of each of its vertices to get the `cubeIndex`.

We can also vectorize the vertex interpolation stage. The problem is that not every edge intersects with the isosurface and thus needs interpolation. Therefore, we need to first determine which edges are actually intersecting the isosurface. To this end, We use a two-pass method. We categorize the edges into three different types based on which direction they are in, namely the direction of X, Y or Z axis. In the first pass, for each of the three directions, we generate a data structure telling which of the edges require interpolation. As for the implementation, the format we use to store those in-


Load Z indices for active edges
_mm256_i32gather_ps(base_addr, vindex, scale)
Gather vertex values of edges start / end vertices (same indices, different base address)
(a)

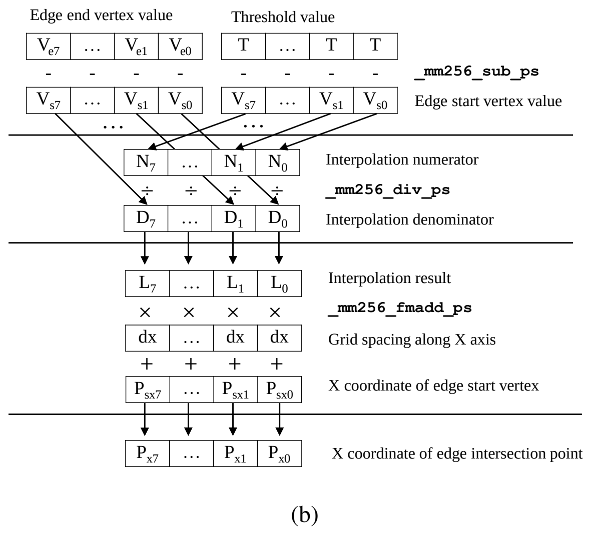


Fig. 6: (a) Gather start and end vertices values for edge interpolation. (b) Vectorized calculation of edge intersection coordinates. Use the edges along X-axis as an example.

formation is similar to the Compressed Sparse Rows (CSR). There is a list of Z indices of active edge start vertices, similar to `col_idx` of CSR; and a list of indices marking the start position in the previous list for each value of Y, similar to `row_start` of CSR. This pass is integrated with the generation of mesh indices. In the second pass, we gather the vertex values as indicated in Fig. 6a. For the 8 edges to be computed, their start vertices and end vertices have the same gathering indices but different based addresses. Then the vectorized computation of intersection point coordinates is done as shown in Fig. 6b. For edges in X direction, we only need to interpolate X coordinate while the other two coordinates stay the same. In the level-by-level method, the X coordinate of edge start vertices are the same for the edges along X-axis direction. For edges along Y or Z direction, overall the calculation is similar but the Y or Z coordinate of edge start vertices may change in different iterations. However, we find that the vectorization of **vertexInterp** does not improve the performance, the reason is explained in Section 5.

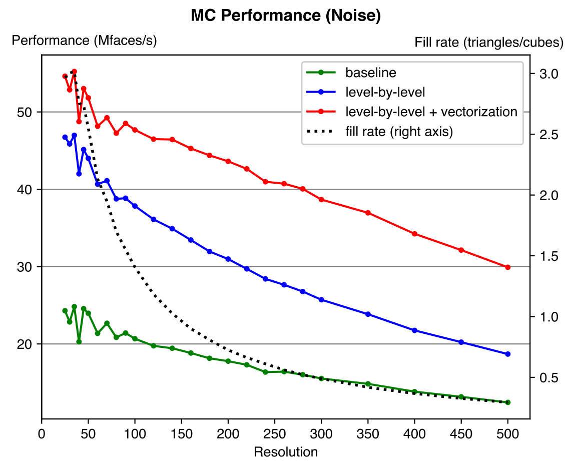


Fig. 7: Performance plot for a random scalar field generated using Perlin Noise as input. It is continuous but changes quickly due to the high frequency, thus the isosurface intersects with many cubes.

4. EXPERIMENTAL RESULTS

In this section, we report the experimental results on performance of our optimization compared with the baseline implementation.

Experimental setup. We test our implementation with an Intel Core i7-7700HQ 2.8GHZ CPU. The code is compiled using MSVC 14.29.30133. The relevant compiler flags are: `/O2` (maximize speed), `/Ob2` (allow inlining), `/D NDEBUG` (disable assertions), `/fp:fast` (enable fast math).

The reported performance are tested with two representative scalar fields as input. The first one is a random input field generated by Perlin's Simplex Noise [12] which results in a dense triangle mesh in the space. The second one is a SDF field of a sphere, and only very few cubes intersect with the isosurface. We also tested on other more realistic inputs such as data from fluid simulation and real medical CT data. The speed-up and performance on these data are comparable to the two representative cases discussed above, so the results are not listed in this report.

Results. The performance for the Noise scene is shown in Fig. 7. The speed-up achieved with level-by-level optimization is around 2x, and the speed-up achieved with level-by-level and vectorization is around 2.5x. The performance plot on sphere distance field is shown in Fig. 8.

As discussed in Section 2, the fixed cost of the algorithm depends on the number of cubes, while the variable cost depends on the number of triangles generated. Thus, we also plot the "fill rate" in the figure (value on right axis), which is defined as the ratio of the number of triangles to the number of cubes (resolution cubed). Since the performance is

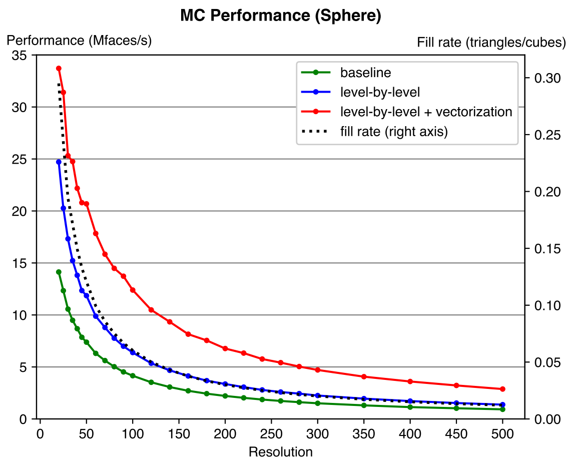


Fig. 8: Performance plot a SDF field to a sphere as input. This is a sparse example and most cubes are empty.

the number of triangle faces processed per second, higher fill rates mean that we are doing more useful computation towards the performance we define. As the plot shows, the trend of the performance with respect to different grid resolution is in accordance with the trend of this "fill rate".

The speed-up achieved with level-by-level optimization is around 1.5x, and the speed-up achieved with level-by-level and vectorization is around 2.5x to 3x. The speed-up of level-by-level alone is lower than the dense Perlin Noise scene, which is expected because the level-by-level optimization could reduce more redundant computation when there are more triangles to be produced. The speed-up of level-by-level and vectorization is higher than the dense case. Vectorization is more useful for the sphere case because the number of intersecting cubes that actually need vertex interpolation is much smaller than the dense case. The threshold comparison and cube topology calculation takes a larger proportion of runtime in the sphere case so when they are fully vectorized, the improvement of runtime is more noticeable.

5. CODE ANALYSIS

In this section, we do code analysis using roofline plot and profiler, which provide insights to the possible bottlenecks in the implementation.

Roofline Analysis. We use Intel Advisor[1] to collect data for roofline analysis. Due to unknown issues, in our test it is only possible to measure total of float and int operations, rather than doing flops and intops measurement separately. Overall, we can see the improvement on operational intensity compared to the baseline. This is mainly because the

[1]www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html

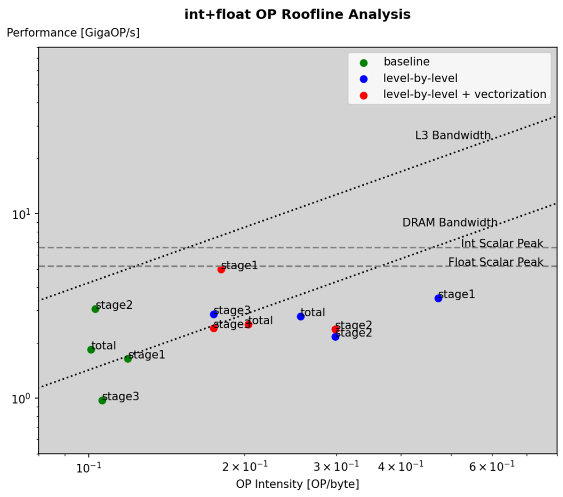


Fig. 9: Roofline analysis of different versions of implementations. Note that the OPs are the sum of integer and single-precision float operations. The performance and operational intensity data for 3 stages are measured, as well as in total.

level-by-level strategy reduces memory access of the redundant computation. The vectorization mainly happens in the first stage (**thresCmp** and `cubeIndex` computation), and its performance gain could be clearly spotted in the plot. The plot also tells us that the third stage **triAssem** is likely to be memory bound.

Bottlenecks. By using Intel VTune Profiler[2], we compare the various implementation variants and evaluate how certain code changes affect the performance and where we can improve the most. It helps primarily in finding hotspots, specific inefficiencies and bad memory utilization. We found that two parts of the code take up almost 80% of the runtime of our final optimized version.

The first bottleneck lies in the **vertexInterp** stage, when computing the intersection vertex index of edges that need interpolation for the indexed mesh. We need to decide which edges are intersecting with the isosurface using "if" statements, increase a global counter of vertex indices which is strictly order-dependent and then store it to the corresponding intermediate array. The "if" branches cannot be predicted, and ILP is difficult to increase with loop unrolling because of the global counter. The cost of actual interpolation is trivial compared to the overhead of branches and the computing and storing of the vertex indices. This explains why the proposed two-pass vectorization of vertex interpolation doesn't work well. In the two-pass method, we further need to store a Z-index of this edge start vertex for the second pass to happen. By doing this, even more

[2]www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

overhead is introduced. Even we can save time by vectorizing interpolation, the bottleneck is still there with the first pass.

The second bottleneck is in **triAssem**, where a triple-level loop over the YZ axis and then over `triTable` entries runs slowly. The access pattern is not consecutive in memory and can't benefit from vectorization. As the roofline plot shows, we think this bottleneck is due to the code is memory bound.

6. CONCLUSIONS

Our optimized implementation of the MC algorithm is settled on two pillars: reorganizing the structure of the computation, and using SIMD intrinsics to integrate vectorization. The proposed optimizations produce 2-3x performance gain. The strategies are general, could be used for any data and input sizes, and are possible to be integrated with other optimization techniques, such as multi-processor parallelism and spatial data structure based methods. In the future, we would like to explore the potential ways for further optimizing the current bottlenecks of our implementation, e.g., ways to increase ILP and branching efficiency in the second stage and improve memory access in the third stage.

7. REFERENCES

[1] William E Lorensen and Harvey E Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM siggraph computer graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[2] Jane Wilhelms and Allen Van Gelder, "Octrees for faster isosurface generation," *ACM Transactions on Graphics (TOG)*, vol. 11, no. 3, pp. 201–227, 1992.

[3] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson, "A near optimal isosurface extraction algorithm using the span space," *IEEE transactions on Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73–84, 1996.

[4] Han-Wei Shen, Charles D Hansen, Yarden Livnat, and Christopher R Johnson, "Isosurfacing in span space with utmost efficiency (issue)," in *Proceedings of Seventh Annual IEEE Visualization'96*. IEEE, 1996, pp. 287–294.

[5] Timothy S Newman, J.Brad Byrd, Pavan Emani, Amit Narayanan, and Abouzar Dastmalchi, "High performance simd marching cubes isosurface extraction on commodity computers," *Computers Graphics*, vol. 28, no. 2, pp. 213–233, 2004.

[6] Gunnar Johansson and Hamish A Carr, "Accelerating marching cubes with graphics hardware.," in *Cascon*. Citeseer, 2006, vol. 6, p. 39.

[7] Christopher Dyken, Gernot Ziegler, Christian Theobalt, and Hans-Peter Seidel, "High-speed marching cubes using histopyramids," in *Computer Graphics Forum*. Wiley Online Library, 2008, vol. 27, pp. 2028–2039.

[8] Timothy S Newman and Hong Yi, "A survey of the marching cubes algorithm," *Computers & Graphics*, vol. 30, no. 5, pp. 854–879, 2006.

[9] Jihun Yu and Greg Turk, "Reconstructing surfaces of particle-based fluids using anisotropic kernels," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 1, pp. 1–12, 2013.

[10] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe, "Poisson surface reconstruction," in *Proceedings of the fourth Eurographics symposium on Geometry processing*, 2006, vol. 7.

[11] Mary L Bouxsein, Stephen K Boyd, Blaine A Christiansen, Robert E Guldberg, Karl J Jepsen, and Ralph Müller, "Guidelines for assessment of bone microstructure in rodents using micro–computed tomography," *Journal of bone and mineral research*, vol. 25, no. 7, pp. 1468–1486, 2010.

[12] Ken Perlin, "Improving noise," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002, pp. 681–682.