Fast Merge Tree Computation via SYCL

Arnur Nigmetov* Dmitriy Morozov†

Lawrence Berkeley National Laboratory

ABSTRACT

A merge tree is a topological descriptor of a real-valued function. Merge trees are used in visualization and topological data analysis, either directly or as a means to another end: computing a 0-dimensional persistence diagram, identifying connected components, performing topological simplification, etc.

Scientific computing relies more and more on GPUs to achieve fast, scalable computation. For efficiency, data analysis should take place at the same location as the main computation, which motivates interest in parallel algorithms and portable software for merge trees that can run not only on a CPU, but also on a GPU, or other types of accelerators. The SYCL standard defines a programming model that allows the same code, written in standard C++, to compile targets for multiple parallel backends (CPUs via OpenMP or TBB, NVIDIA GPUs via CUDA, AMD GPUs via ROCm, Intel GPUs via Level Zero, FPGAs). In this paper, we adapt the triplet merge tree algorithm to SYCL and compare our implementation with the VTK-m implementation, which is the only other implementation of merge trees for GPUs that we know of.

Keywords: triplet merge tree, computations on GPU, SYCL

Index Terms: Computing methodologies—Massively parallel algorithms; Theory of computation—Computational geometry;

1 INTRODUCTION

Let f be a continuous scalar function defined on some domain D in $\mathbb{R}^n$. Let us consider connected components of its sublevel sets $f^{-1}(-\infty, t]$, as t changes from $-\infty$ to ∞. Each time we pass a local minimum, a new connected component appears; existing connected components merge together as we pass a saddle. We say that two points $x, y \in D$ are equivalent, if $f(x) = f(y)$, and both x and y belong to the same connected component of $f^{-1}(-\infty, f(x)]$. Taking a quotient of the domain by this equivalence relation, we obtain a tree, called a *merge tree*. Each branch of the tree corresponds to a pair of critical points (a,b), where a is a local minimum and b is a saddle. Short branches, i.e., branches such that $|f(a) - f(b)|$ is small, can be intuitively interpreted as topological noise, small 'wrinkles' on the graph of f. Longer branches encode more significant features of the function. Merge tree is an example of a topological descriptor.

Another topological descriptor is a *persistence diagram* — in our case specifically, a zero-dimensional persistence diagram. It is a multi-set of points $(f(a), f(b)) \in \mathbb{R}^2$, one point per branch (a,b) of the merge tree. It can be viewed as 'bag of features': compared to the merge tree, we lose the information about how the branches merge together and keep only their endpoints. Persistence diagrams are one of the main tools of topological data analysis. Since they are not the main topic of this article, we refer an interested reader to the textbook [10] for a proper definition of persistence diagrams in all dimensions.

The last topological descriptor that we want to mention is a *contour tree*, which captures the structure of level sets on simply connected domains, i.e., domains where every loop can be contracted to a point. Formally, we declare two points $x, y \in D$ equivalent, if $f(x) = f(y)$, and x and y are in the same connected component of $f^{-1}(f(x))$. We get the contour tree by taking the quotient of D by this equivalence relation. Contour tree is a well-known tool in visualization community, since it allows to graphically present the structure of high-dimensional scalar fields.

In practice, we do not work with continuous functions on continuous domains. Typically, a scalar function is given only on the points of a grid. We can connect neighboring grid points, obtaining a graph with the function defined on its vertices. This is a standard setting, in which all the aforementioned topological descriptors are computed in practice .

We are interested in merge trees for the following reasons:

- There are many applications of merge trees per se, including halo finding in cosmology [11], finding atmospheric rivers in climatology [16], tracking features in combustion simulations [3, 4, 21], among many others.

- One of the standard algorithms for computing contour trees, proposed by Carr et al. [6], computes the contour tree of f from the merge trees of the functions f and $-f$ (the authors call them the *join tree* and the *split tree*).

- The connection between merge trees and persistence diagrams is not just a shortcut for defining the diagram in dimension 0, but it is a practically efficient way to compute it. Persistence diagrams are used, e.g., in neuroscience [25], material science [5], chemistry [19], etc. We can also refer the reader to the database of articles with real-world application of TDA [12] for more applications. Note that for large inputs the zero-dimensional persistence diagram is the only computable option.

Often in these applications the input function is a result of a numerical simulation on a supercomputer. As the supercomputer architectures evolve and most of the computation is offloaded to fast accelerators, most commonly GPUs, data analysis needs to keep up with these changes and adapt the algorithms to run on the same devices that contain the data. There are many reasons for this: computational efficiency, minimizing data movement (both to reduce runtime and for energy efficiency), the need for in situ analysis to provide rapid feedback to the simulation, among many others. All these factors motivate developing efficient, *portable* algorithms that can run on as wide of a range of the devices as the simulations themselves. In addition, the emerging interest in using topological information in machine learning, especially, using persistence diagrams to guide the learning process [8, 20, 23], motivates developing efficient implementations of topological algorithms that can execute on the same devices as the machine learning pipelines, i.e., predominantly on GPUs.

Related work. In the serial setting, merge trees can be computed using a variation of the classical Kruskal's algorithm. There are several papers proposing different shared-memory parallel algorithms [4, 14, 24]. Carr et al. [7] proposed a parallel algorithm for

*e-mail:anigmetov@lbl.gov

†e-mail:dmorozov@lbl.gov

computing merge trees, implemented in VTK-m with OpenMP and Thrust (CUDA). We recall more details about this algorithm in the next section, because we use that implementation as a baseline for comparison. Smirnov and Morozov [26] developed the concept of a triplet merge tree, with a simple path-merging algorithm to compute it. The main advantage of this algorithm is that it is lock-free, using the compare-and-swap idiom. Since this is the algorithm that we implement using SYCL in this paper, we also review it in the next section.

Our contributions are as follows:

- We adapt the triplet merge algorithm [26] to SYCL, a programming model targeting multiple processing units (including CPUs, GPUs, and other accelerators).
- We experimentally evaluate our implementation, comparing it to the VTK-m implementation of merge trees, which is the only implementation that we know of that runs on GPUs (and CPUs).

2 Background

2.1 Merge Trees

Let $f\colon X \to \mathbb{R}$ be a function on a space X. The merge tree of f is the quotient space $\mathscr{T}_f = X/$, where we identify two points x_1 and x_2 if and only if $f(x_1) = f(x_2) =: y$, and x_1 and x_2 belong to the same connected component of the sublevel set $f^{-1}(-\infty, y]$. In our setting, the space is a graph $\mathscr{G} = (V,E)$, and the function is defined only on the vertices, $f\colon V \to \mathbb{R}$. We extend f to edges by linear interpolation and write $f\colon \mathscr{G} \to \mathbb{R}$. For $c \in \mathbb{R}$, we use G_c to denote the sublevel set: $G_c := \{v \in V \mid f(v) \leq c\}$. The *merge tree* of f is a graph $\mathscr{T}_f$ with the same vertex set V. Assuming that $f(v_1) \leq f(v_2)$, there is an edge $v_1 v_2$ in $\mathscr{T}_f$ if and only if there does not exist a vertex u such that $f(v_1) \leq f(u) \leq f(v_2)$ and v_1 and u are in the same component of $G_{f(u)}$.

Merge trees can be computed in $O(m \log n)$ time, on a graph with n vertices and m edges, using the union–find (disjoint sets) data structure. First, one sorts the vertices by values of the function. Then the algorithm goes over the sorted vertices and uses the union–find data structure to determine which of the three alternatives occurs: (1) the current vertex is a local minimum, so it creates a new connected component in the sublevel set; (2) the current vertex belongs to exactly one of the components that were present at the previous vertex; (3) multiple connected components are merged together at the current vertex. The requirement to process the vertices in sorted order makes it difficult to parallelize this algorithm.

Peak pruning. Carr et al. [7] compute the merge tree in parallel avoiding global sorting. For each vertex u that is not a local minimum, they pick an arbitrary edge uv such that $f(v) < f(u)$, a completely independent and thus easily parallelizable operation. They follow the selected edges $f(u) > f(v) > f(w) > \cdots > f(z)$ to build a path from every vertex u to a local minimum z; they say that u is *assigned* to z. Among all vertices assigned to the same local minimum z the authors identify those that can be saddles; such a saddle candidate must have two neighbors below it that are assigned to different local minima. They prove that the lowest saddle candidate assigned to z is exactly the saddle paired with z. That is, the lowest saddle candidate s is the saddle at which the sublevel set component born at z merges into another one. This immediately gives the pairing of the critical points and the branch structure of the merge tree. The remaining part of their algorithm takes care of regular vertices (those vertices that have degree two in the merge tree). Some of those are readily available: all vertices that are assigned to z and are below s must be on that branch. The authors remove all these vertices and repeat the same procedure (note that some of the saddles became minima after removal). This algorithm is implemented in VTK-m [22], and we use it as a baseline for comparison.

Triplet merge trees. A different parallel algorithm is described in [26]. Its main idea is to replace the standard merge tree, described above, by its dual tree of branches. Each branch is a path that tracks when a component is created and when it is merged with an older component. It is represented as a *triplet* of vertices u, s, v, such that $f(v) < f(u) \leq f(s)$ and u and v are in the same connected component of the sublevel set $\mathscr{G}_{f(s)}$. The triplet means that a branch created by vertex u merges with a branch created by vertex v at vertex s. Additionally, if u is a global minimum of f in its connected component of the graph, then, by convention, (u,u,u) is a triplet. It is convenient to interpret each triplet as a directed edge (u,v) with a label s. Vertices of degree two in a standard merge tree (i.e., the inner vertices of a branch) become leaves in the triplet merge tree: they are represented by triplets (u,u,v). To get a unique representation of a merge tree as a collection of triplets, we need two conditions: (1) each vertex u appears exactly once as a first vertex of some triplet; (2) for each (u,s,v), vertex v is the deepest vertex in $f^{-1}(-\infty, f(s)]$. The first condition is called *normalization*, the second one *minimality*.

In [26] the authors explain the details of a lock-free algorithm to compute the triplet representation of $\mathscr{T}$. We reproduce the algorithm here for convenience, see Algorithm 1. Since we want to maintain the normalization condition from the start, we represent the tree as a map T, where the first vertex u of a triplet is the key; the entries are pairs $T[u] = (s,v)$ — the second and the third vertices of the triplet. In the original CPU implementation, $T[u]$ is a pointer to a pair, not a pair itself. We ignore that in the pseudocode, but return to this issue in Section 3.2, where we explain the changes needed for the GPU.

In the first loop, we initialize the tree with triplets (u,u,u) (which corresponds to the case of no edges, $E = \emptyset$). Then the algorithm works in two phases: merge and repair.

The merge phase (lines 4–8) processes the edges E in parallel: for each edge (u,v) it starts with a normalized merge tree on a graph missing this edge and changes it to incorporate (u,v). The proof of correctness can be found in [26]. The updates are synchronized using compare-and-swap operations (Algorithm 3, line 14), which guarantees correctness. Logically, compare-and-swap (CAS, Algorithm 2) checks if the variable v that we want to modify has the value that we expect (usually this means that another thread has not modified it since we read its value), and, only in that case, changes it to the desired value. It returns the result of the comparison. All these operations are performed atomically, as a single transaction. If the comparison failed, we simply start from scratch (line 15).

The repair phase fixes the minimality condition that may be violated in the merge phase. Let u be a vertex, a a real number. We call vertex v with the smallest function value in the connected component of u in $f^{-1}(-\infty, a]$ the *representative* of u at level a. If we view triplets (u,s,v) as directed edges from u to v, then, to find a representative of u, we just need to follow these edges until we reach the deepest vertex. This is exactly the role of Algorithm 4. To ensure minimality, in Algorithm 5 we simply replace triplet (u,s,v) with the triplet (u,s,v'), where v' is the representative of u.

Algorithm 1 Triplet Merge Tree Computation.

```
 1: function ComputeMergeTree(G)
 2:     for all vertex u ∈ G do in parallel
 3:         T[u] ← (u,u)
 4:     for all edge (u,v) ∈ G do in parallel
 5:         if f(v) < f(u) then
 6:             Merge(T, u, u, v)
 7:         else
 8:             Merge(T, v, v, u)
 9:     for all vertex u ∈ G do in parallel
10:         Repair(u)
11:     return T
```

Algorithm 2 Compare-and-Swap.

```
1: function CAS(v, expected, desired)
2:     if v = expected then
3:         v ← desired
4:         return True
5:     else
6:         return False
```

Algorithm 3 Parallel Merge.

```
1: function MERGE(T, u, s, v)
2:     (s_u, u′) ← T[u]
3:     if f(s_u) < f(s) then
4:         return MERGE(T, u′, s, v)
5:     (s_v, v′) ← T[v]
6:     if f(s_v) < f(s) then
7:         return MERGE(T, u, s, v′)
8:     if u = v then
9:         return
10:    if f(v) < f(u) then
11:        SWAP((u, s_u, u′), (v, s_v, v′))
12:    if CAS(T[v], (s_v, v′), (s, u)) then
13:        MERGE(T, u, s_v, v′)
14:    else
15:        MERGE(T, u, s, v)
16:    return T
```

Algorithm 4 Representative in Triplet Merge Tree.

```
1: function REPRESENTATIVE(T, u, a)
2:     (s, v) ← T[u]
3:     while f(s) ≤ a and s ≠ v do
4:         u ← v
5:         (s, v) ← T[u]
6:     return v
```

3 TMT-SYCL

3.1 SYCL

SYCL is a programming model for writing heterogeneous parallel programs. It was originally designed to provide a layer of abstraction over OpenCL, but has since evolved into an independent standard, with several independent implementations, not bound to a particular parallel device. The main feature of SYCL is that it uses a single-source code written in standard C++. The code for parallel execution on device is written in the same file with the host code, and takes advantage of the standard language constructs. This single source is then processed in multiple passes of a compiler (or different compilers for host and device). Different implementations of SYCL target CPUs via OpenMP or TBB, NVIDIA GPUs via CUDA, AMD GPUs via ROCm, Intel GPUs via Level Zero, Intel and Xilinx FPGAs. We use an implementation called hipSYCL [2]. It supports OpenMP, NVIDIA CUDA and AMD (ROCm) backends.

3.2 Changes needed for GPU

One limitation of SYCL is lack of support for dynamic memory allocation. On-device buffers are declared, with specific size, before the execution of the device code that uses them. In the CPU implementation[1] of [26], the tree is represented as a map with vertices as keys and pointers as values. The pointer corresponding to vertex u refers to an object that stores the two vertices s and v. It also stores

[1] Available publicly in Reeber, `github.com/mrzv/reeber`

Algorithm 5 Repair Triplet Merge Tree.

```
1: function REPAIR(T, u)
2:     (s, v) ← T[u]
3:     v′ ← REPRESENTATIVE(T, u, f(s))
4:     if u ≠ v′ then
5:         T[u] ← (s, v′)
6:     return T[u]
```

a vector of degree-2 vertices. Note that in the CPU implementation vertices were represented as double-word variables, which means that we cannot use compare-and-swap to atomically update the pair of values (s,v). Normal CAS operates on single words, and double word CAS, available on `x86-64`, can update 2 contiguous words in memory atomically, but we would need to update 4 words. This is the reason for $T[u]$ being a pointer to a pair.

Although it is possible to implement a dynamic memory allocator on top of the static buffers (and then implement a hash map on top), we choose a simpler route. The triplet (u,s,v) is represented as an array of pairs: $T[u] = (s,v)$. Another limitation of SYCL is that it does not provide a double-word compare-and-swap operation, which the triplet merge algorithm requires to atomically update the pair (s,v). We choose a simple solution: we limit the vertex identifier to a 32-bit integer. This way a pair can be packed into a 64-bit integer and a regular compare-and-swap operation, which is supported by SYCL, suffices. This limitation is minor, given the current memory constraints on GPUs: the data and the tree together for 2^{32} vertices, assuming 32-bit floating point and 32-bit integers for the tree, would require 48 GiB on the device.

4 Experiments

Datasets. We use the following datasets. All of them, except *NYX*, were downloaded from P. Klacansky's database [17]. We used averaging and interpolation to downsample the higher-resolution datasets.

- Cosmology (*NYX*) [1]. This is a snapshot of the dark matter density from a Nyx simulation (for redshift $z = 2$).
- Magnetic reconnection (*MAG*) [15]. This phenomenon is observed in plasma; the lines of magnetic field change their connectivity. The dataset is a single step of a simulation.
- Isotropic pressure (*IP*) [27]. The function is the pressure field of a direct numerical simulation of forced isotropic turbulence.
- Entropy field of Richtmyer–Meshkov instability (*RM*) [9]. This is an event observed when two fluids of different density are intensively mixed with other by the impact of a shock wave.
- Three CT scans. *CHAM* is a CT scan of a chameleon, scanned by DigiMorph. *WOOD* is a CT scan of a wood branch by the Computer-Assisted Paleoanthropology group and the Visualization and MultiMedia Lab at University of Zurich. *PAW* is a scan of a *Pawpawsaurus Campbelli*. These datasets have rich topology, if one considers the evolution of superlevel sets, because many connected components emerge at high density.
- *TRUSS* is a simulated CT scan of an $8 \times 8 \times 8$ octet truss [18]. This dataset has a specifically regular structure.
- *JICF-Q* is a Q criterion of a jet in cross-flow [13]. If $\vec{v}$ is a velocity vector field of fluid, then its gradient ∇v is a 3×3 tensor. It can be decomposed into the symmetric part S and anti-symmetric part Ω: $\nabla \vec{v} = S + \Omega$. Q criterion is defined as $\frac{1}{2}(\|\Omega\|^2 - \|S\|^2)$ and is used to detect vortices. Most of the topological evolution of this dataset is localized in a small part of the domain.

Setup. Experiments were performed on a computer with Intel(R) Xeon(R) Gold 6230 CPU (20 physical cores), NVIDIA GeForce RTX 2080 Ti GPU, running Arch Linux. For comparison, we used VTK-m code for computing the contour tree. Since VTK-m computes contour trees from two merge trees, following the algorithm of Carr et al. [6], we measured the running time of both computations separately. To this end, we modified the code of VTK-m, timing both calls of the merge tree computation routine separately, to make the comparison fair by excluding the overhead to combine them into the contour tree.[2] However, in all these datasets the merge tree of $-f$ carries more information than the tree of f (one can verify that by visually exploring the function for different threshold values using a tool at [17]). For example, for the cosmological dataset, the branches of $\mathscr{T}_{-f}$ are born at the local maxima of f, which capture *halos*, clusters of high density, while $\mathscr{T}_f$ consists of a single path. Therefore, we report the results for $-f$ only. Using the terminology of [7], we are only interested in the *split tree*. Each experiment was run 5 times, and we report the average timing. The running time is stable across the experiments, showing only minor fluctuations of 5% maximum.

Results. On two datasets, *NYX* and *MAG*, our implementation clearly outperforms VTK-m, see Figs. 2 and 3. On CT scans (*CHAM*, *WOOD*, *TRUSS*) our GPU implementation also shows best running times, but here the advantage is less visible, see Figs. 6 to 8. Both algorithms are very data-dependent, and this behavior is not at all universal: for the *IP* dataset, our implementation is just a bit slower, as shown in Fig. 4, but for the *RM* dataset, it performs significantly worse, see Fig. 5. On the other hand, for the largest variant of *PAW*, performance of VTK-m deteriorates, see Fig. 10. One possible explanation for this is that the topology of *NYX* and *MAG* datasets is richer: the number of branches in the merge tree for 512^3 samples from cosmological and magnetic reconnection datasets is between $7 \cdot 10^6$ and $8 \cdot 10^6$, while for *IP* the tree consists of much fewer branches, around $5 \cdot 10^5$. Thus, if the function is expected to be topologically complex, the triplet merge tree algorithm has an advantage.

We note that the topological complexity is not the only factor. The triplet merge tree algorithm's worst-case complexity is quadratic, and it is likely that Richtmyer–Meshkov data set is triggering this behavior. This data set has another distinctive feature. Using the online visualization tool [17], we see that at first the evolution of the superlevel sets is similar to CT scans: there are multiple peaks where the function value is high and, as we decrease the threshold, they gradually merge together, so that the volume of the superlevel set changes slowly. However, then there are two sharp spikes: as we decrease the threshold by a tiny amount, half of the domain is added to the superlevel set. One example of this is shown in Fig. 1.

We summarize all comparisons in Table 1. When we compare the GPU versions of our implementation and VTK-m, we often see a speedup by a factor of 10–14 for the smallest versions of the datasets, 128^3 and similar sizes. This likely has to do with some initial setup computation performed by the VTK-m code rather than with the difference between the algorithms. For larger sizes, we often outperform VTK-m by a factor that ranges between 1.5 and 4.8.

Perhaps the most disappointing result of our experiments is that we never see a significant improvement when switching from OpenMP parallelism to GPU; in fact, sometimes the latter is slightly slower. The highest speed up that we observed was for *TRUSS* dataset, where our GPU implementation performed almost 5 times faster than its OpenMP variant, but this is an exception. Typically, we gain between 10 to 50 percent speed up when we move to GPU. The difference is too small to be seen in the plots with logarithmic scale. As mentioned in the footnote, we cannot fully attribute this to the time needed to transfer data to and from the device. Of course, the ability to compute the tree directly on the GPU is still useful: either to free the CPU for another task, or especially, when the input data already resides on the device (e.g., as part of a numerical simulation or machine learning pipeline).

Finally, we evaluate the strong scaling of the OpenMP versions of the codes, as we increase the number of threads processing the datasets of size 512^3, as shown in Figs. 11 and 12[3]. As we can see, our implementation usually scales better than VTK-m, showing some improvement even as we go from 64 to 80 threads, even though at this number of threads we are using hyperthreading (there are only 40 physical cores available).

5 Conclusion

We presented a GPU implementation of a merge tree computation that often performs better than the previously available implementation in VTK-m for inputs that are topologically rich and do not undergo drastic changes in the volume of the sublevel set. We believe that such inputs are reasonably common, so it makes sense to use the GPU version of the triplet merge algorithm. There are two questions for future work. First, how can we gain more from GPU parallelism? Second, how to adapt our algorithm to a distributed setting? This is not trivial because of the 32 bit limitation that we had to make in order to pack an edge into 64 bits. While innocent for a single GPU, for large-scale simulations that run on the GPUs of multiple nodes, it becomes a problem.

Acknowledgements

This work was supported by Laboratory Directed Research and Development (LDRD) funding from Berkeley Lab, provided by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

References

[1] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel. Nyx: A massively parallel amr code for computational cosmology. *The Astrophysical Journal*, 765(1):39, 2013.

[2] A. Alpay and V. Heuveline. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL*, number Article 8 in IWOCL '20, p. 1. Association for Computing Machinery, New York, NY, USA, Apr. 2020. doi: 10.1145/3388333.3388658

[3] J. C. Bennett, V. Krishnamoorthy, S. Liu, R. W. Grout, E. R. Hawkes, J. H. Chen, J. Shepherd, V. Pascucci, and P.-T. Bremer. Feature-based statistical analysis of combustion simulation data. *IEEE transactions on visualization and computer graphics*, 17(12):1822–1831, 2011.

[4] P.-T. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. Bell. Interactive exploration and analysis of large-scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1324, 2010.

[5] M. Buchet, Y. Hiraoka, and I. Obayashi. Persistent homology and materials informatics. In *Nanoinformatics*, pp. 75–95. Springer, Singapore, 2018.

[6] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry*, 24(2):75–94, 2003.

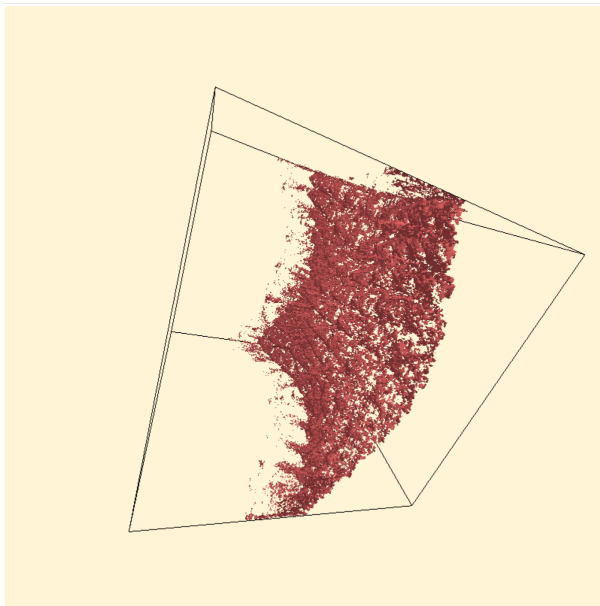
[7] H. A. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens. Parallel peak pruning for scalable SMP contour tree computation. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 75–84. IEEE, 2016.

[2] It is difficult to measure the time it takes to transfer data to GPU and back from it. However, the output of `nvprof` shows that the transfer takes a relatively small time for both VTK-m and tmt-sycl codes, about 10% to 15% of the total execution.

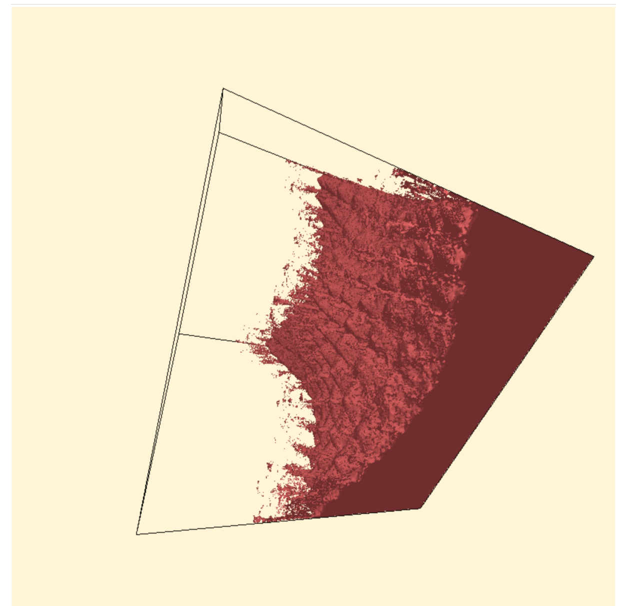
[3] This experiment was performed on a machine that has not one, but two CPUs with 20 physical cores each. Other characteristics of the two machines are the same.

Dataset	Size	Comparison OpenMP	Comparison GPU	TMT-SYCL GPU speedup	VTK-m GPU speedup
WOOD	$128 \times 128 \times 128$	2.177	14.659	1.392	0.207
	$256 \times 256 \times 256$	3.294	4.853	1.459	0.991
	$512 \times 512 \times 512$	4.859	3.404	1.298	1.853
CHAM	$128 \times 128 \times 128$	1.602	13.091	1.526	0.187
	$256 \times 256 \times 256$	0.966	3.068	1.578	0.496
	$512 \times 512 \times 512$	2.468	1.504	1.046	1.717
TRUSS	$128 \times 128 \times 128$	1.155	9.770	1.885	0.223
	$256 \times 256 \times 256$	1.223	2.838	2.423	1.045
	$512 \times 512 \times 512$	0.818	2.710	4.917	1.485
MAG	$128 \times 128 \times 128$	72.489	14.362	0.957	4.833
	$256 \times 256 \times 256$	13.953	4.278	1.070	3.491
	$512 \times 512 \times 512$	6.389	2.997	1.073	2.288
IP	$128 \times 128 \times 128$	1.688	0.905	1.143	2.132
	$256 \times 256 \times 256$	1.694	0.984	1.093	1.881
	$512 \times 512 \times 512$	1.506	0.765	1.060	2.087
PAW	$119 \times 80 \times 136$	2.022	18.417	1.282	0.141
	$238 \times 160 \times 272$	2.053	5.113	1.318	0.529
	$476 \times 320 \times 544$	2.803	2.261	1.411	1.750
	$952 \times 640 \times 1088$	4.791	83.137	1.135	0.065
JICF-Q	$176 \times 135 \times 137$	2.294	11.631	1.386	0.273
	$352 \times 270 \times 275$	2.075	2.745	1.196	0.904
	$704 \times 540 \times 550$	1.476	1.287	1.144	1.312
RM	$128 \times 128 \times 128$	0.772	0.390	0.920	1.821
	$256 \times 256 \times 256$	0.290	0.313	1.878	1.735
	$512 \times 512 \times 512$	0.023	0.218	19.520	2.047
NYX	$128 \times 128 \times 128$	3.068	1.852	0.966	1.600
	$256 \times 256 \times 256$	3.777	2.191	0.986	1.701
	$512 \times 512 \times 512$	4.042	4.412	0.915	0.838

Table 1: Comparison of our implementation to VTK-m. Comparison columns for OpenMP and GPU: running time of VTK-m divided by the running time of TMT-SYCL (greater than 1 means our implementation is faster). TMT-SYCL GPU speed-up: our running time on OpenMP divided by the running time on GPU (speedup factor of switching to GPU). VTK-m GPU speed-up: VTK-m running time on OpenMP divided by VTK-m running time on GPU (speed-up factor of switching to GPU for VTK-m).



(a) Slightly above the threshold



(b) Slightly below the threshold

Figure 1: Screenshots of visualization of *RM* dataset for two isovalues.

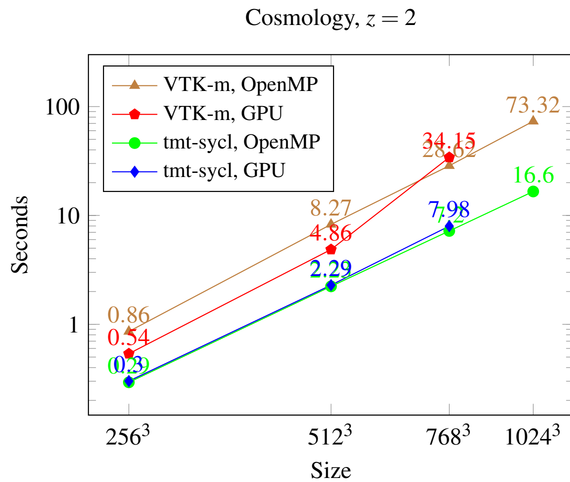


Figure 2: Running time comparison of tmt-sycl and VTK-m, on *NYX* dataset, varying input size.

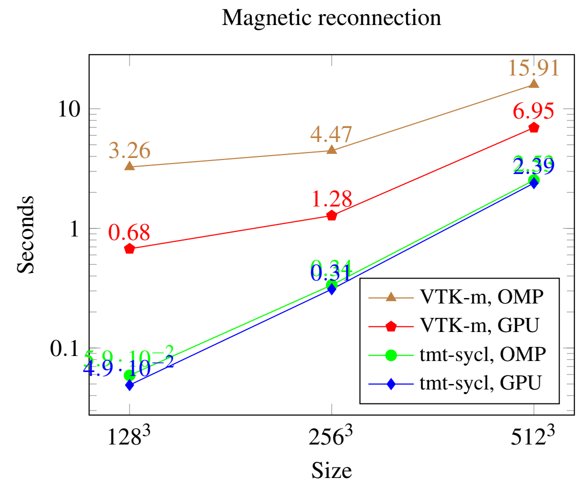


Figure 3: Running time comparison of tmt-sycl and VTK-m, on *MAG* dataset, varying input size.

[8] C. Chen, X. Ni, Q. Bai, and Y. Wang. A topological regularizer for classifiers via persistent homology. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 2573–2582, 2019.

[9] R. H. Cohen, W. P. Dannevik, A. M. Dimits, D. E. Eliason, A. A. Mirin, Y. Zhou, D. H. Porter, and P. R. Woodward. Three-dimensional simulation of a Richtmyer–Meshkov instability with a two-scale initial perturbation. *Physics of Fluids*, 14(10):3692–3709, 2002.

[10] H. Edelsbrunner and J. L. Harer. *Computational topology: an introduction*. American Mathematical Society, 2022.

[11] B. Friesen, A. Almgren, Z. Lukić, G. Weber, D. Morozov, V. Beckner, and M. Day. In situ and in-transit analysis of cosmological simulations. *Computational Astrophysics and Cosmology*, 3(1):4, 2016.

[12] B. Giunti and J. Lazovskis. Tda-applications. khttps://www.zotero.org/groups/2425412/tda-applications, 2020.

[13] R. W. Grout, A. Gruber, H. Kolla, P.-T. Bremer, J. C. Bennett, A. Gyulassy, and J. H. Chen. A direct numerical simulation study of turbulence and flame structure in transverse jets analysed in jet-trajectory based coordinates. *Journal of Fluid Mechanics*, 706:351–383, 2012. doi: 10.1017/jfm.2012.257

[14] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. Task-based augmented merge trees with fibonacci heaps. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 6–15. IEEE, 2017.

[15] F. Guo, H. Li, W. Daughton, and Y.-H. Liu. Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Phys. Rev. Lett.*, 113:155005, Oct. 2014. doi: 10.1103/PhysRevLett.113.155005

[16] M. Inatsu, H. Kato, Y. Katsuyama, Y. Hiraoka, and I. Ohbayashi. A

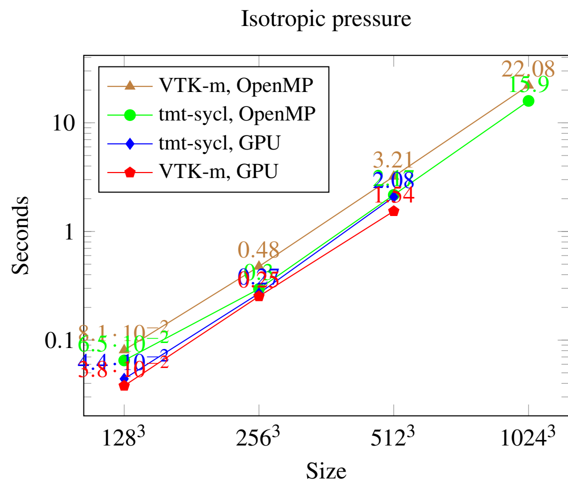


Figure 4: Running time comparison of tmt-sycl and VTK-m, on *IP* dataset, varying input size.

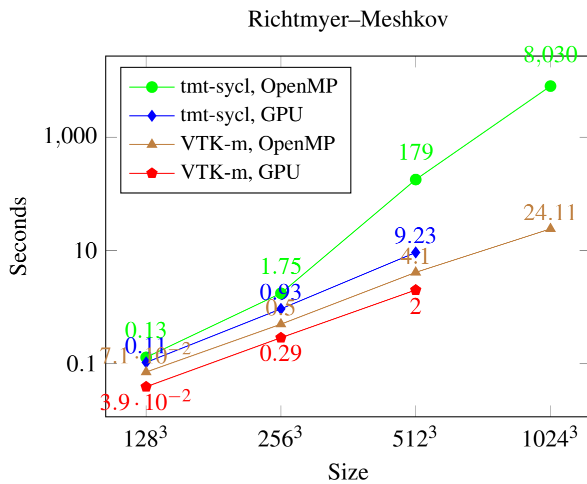


Figure 5: Running time comparison of tmt-sycl and VTK-m, on *RM* dataset, varying input size.

cyclone identification algorithm with persistent homology and merge-tree. *SOLA*, 13:214–218, 2017.

[17] P. Klacansky. Open scientific visualization datasets. klacansky.com/open-scivis-datasets, 2021.

[18] P. Klacansky, H. Miao, A. Gyulassy, A. Townsend, K. Champley, J. Tringe, V. Pascucci, and P.-T. Bremer. Virtual inspection of additively manufactured parts. In *2022 IEEE 15th Pacific Visualization Symposium (PacificVis)*, pp. 81–90, 2022. doi: 10.1109/PacificVis53943.2022.00017

[19] A. S. Krishnapriyan, M. Haranczyk, and D. Morozov. Topological descriptors help predict guest adsorption in nanoporous materials. *The Journal of Physical Chemistry C*, 124(17):9360–9368, 2020.

[20] J. Leygonie, S. Oudot, and U. Tillmann. A framework for differential calculus on persistence barcodes, Oct. 2019.

[21] A. Mascarenhas, R. W. Grout, P.-T. Bremer, E. R. Hawkes, V. Pascucci, and J. H. Chen. Topological feature extraction for comparison of terascale combustion simulation data. In *Topological Methods in Data Analysis and Visualization*, pp. 229–240. Springer, 2011.

[22] K. Moreland, C. Sewell, W. Usher, L.-T. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci. VTK-m: Accelerating the visualization toolkit for massively threaded architectures, 2016. doi: 10.1109/mcg.2016.48

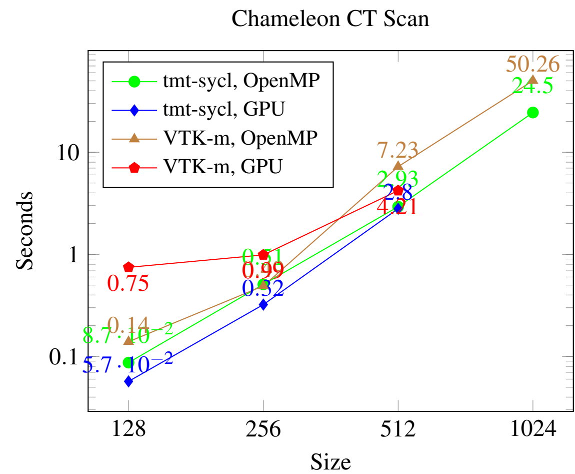


Figure 6: Running time comparison of tmt-sycl and VTK-m, on *CHAM* dataset, varying input size.

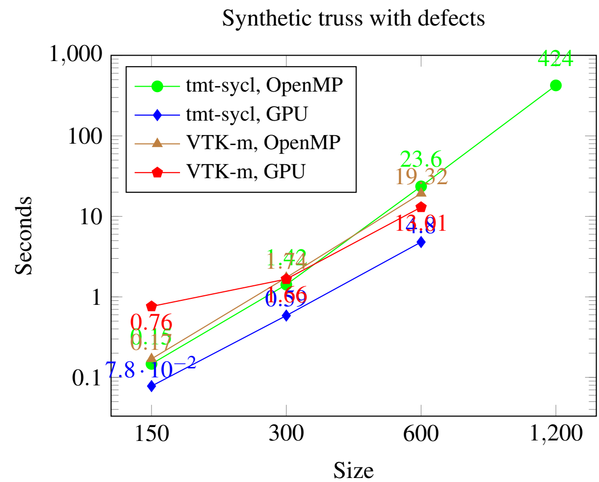


Figure 7: Running time comparison of tmt-sycl and VTK-m, on *TRUSS* dataset, varying input size.

[23] A. Nigmetov, A. S. Krishnapriyan, N. Sanderson, and D. Morozov. Topological regularization via persistence-sensitive optimization, 2020.

[24] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, 2004.

[25] G. Petri, P. Expert, F. Turkheimer, R. Carhart-Harris, D. Nutt, P. J. Hellyer, and F. Vaccarino. Homological scaffolds of brain functional networks. *Journal of The Royal Society Interface*, 11(101):20140873, 2014.

[26] D. Smirnov and D. Morozov. Triplet merge trees. In *Topological Methods in Data Analysis and Visualization V: Theory, Algorithms, and Applications*, 2020.

[27] P. Yeung, D. Donzis, and K. Sreenivasan. Dissipation, enstrophy and pressure statistics in turbulence simulations at high Reynolds numbers. *Journal of Fluid Mechanics*, 700:5, 2012.

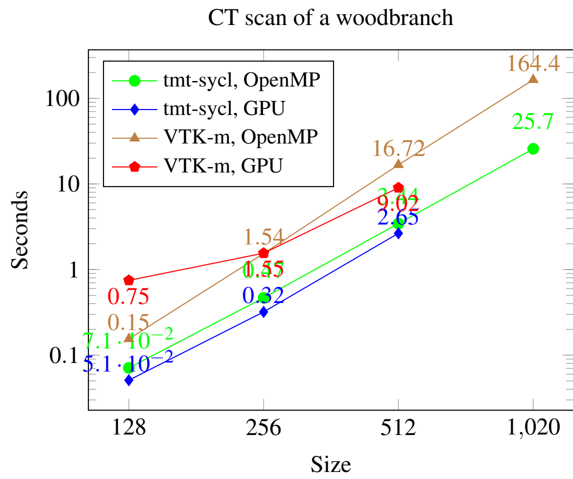


Figure 8: Running time comparison of tmt-sycl and VTK-m, on *WOOD* dataset, varying input size.

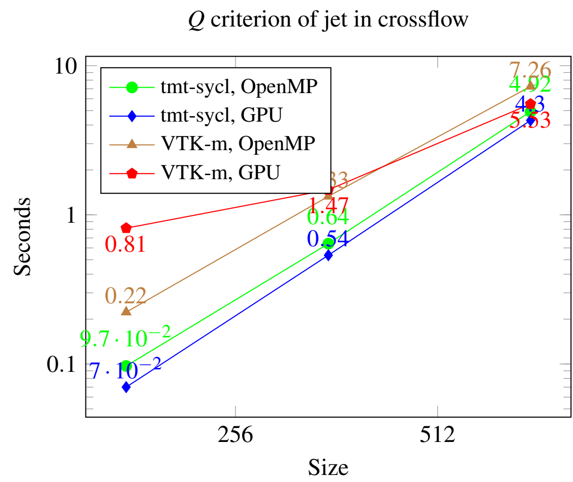


Figure 9: Running time comparison of tmt-sycl and VTK-m, on *JICF-Q* dataset, varying input size.

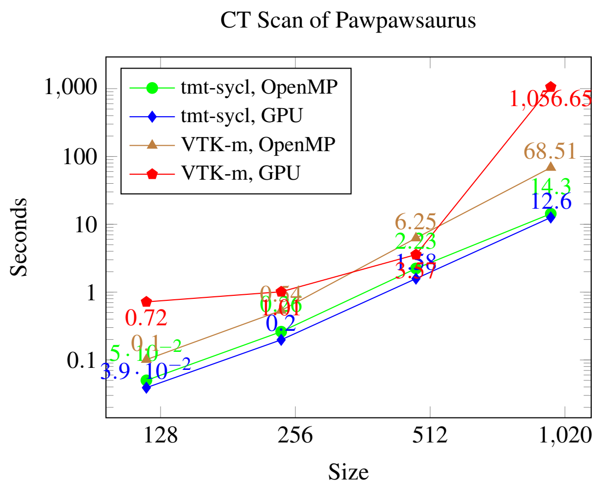


Figure 10: Running time comparison of tmt-sycl and VTK-m, on *PAW* dataset, varying input size.

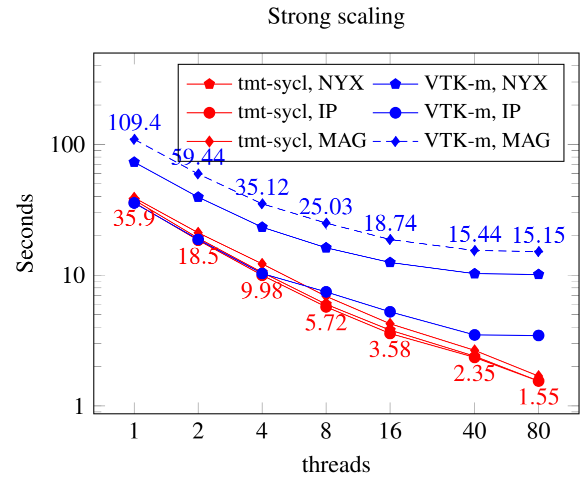


Figure 11: Strong scaling of OpenMP version of tmt-sycl and VTK-m.

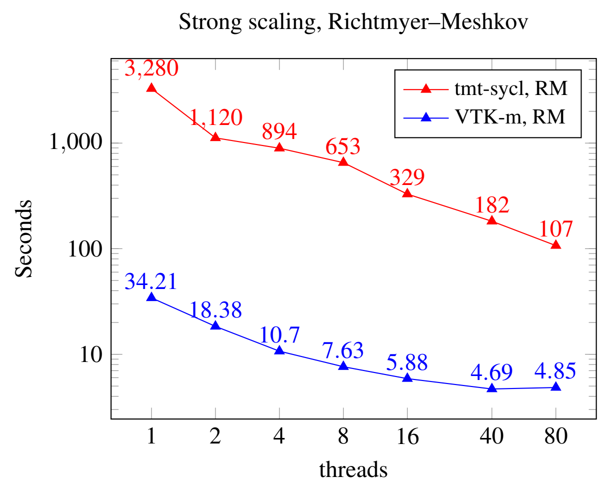


Figure 12: Strong scaling of OpenMP version of tmt-sycl and VTK-m.