Exploring Parallel T-distributed Stochastic Neighbor Embedding

15-418/618 Project Final Report

Yue Yin (Andrew ID yyin5)
Zhe Chen (Andrew ID zhechen2)

Thursday 5th May, 2022

1 Summary

We parallelized t-distributed stochastic neighbor embedding (t-SNE), a dimensionality reduction technique commonly used for high-dimensional data visualization. We parallelized the program with CUDA on GPU, with OpenMP and ISPC on CPU, and evaluated the performance of the programs on PSC. We are able to achieve significant speedups as compared to the baseline through a heterogeneous computing approch involving all 3 parallelization techniques.

2 Background

2.1 t-SNE Algorithm

From a D-dimensional data set $\mathbf{X} \in \mathbb{R}^D$, t-SNE produces a d-dimensional embedding $\mathbf{Y} \in \mathbb{R}^d$ ($d < D$), such that if two points $\mathbf{x}_i$ and $\mathbf{x}_j$ are close to one another in the space $\mathbf{X}$, their corresponding points $\mathbf{y}_i$ and $\mathbf{y}_j$ are also close in the lower dimensional space $\mathbf{Y}$.

t-SNE models similarities between points as probability densities. In the input space, the similarity between $\mathbf{x}_i$ and $\mathbf{x}_j$, modelled by a Gaussian distribution and denoted as p_{ij}, is defined as

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2}, \text{ where } p_{j|i} = \frac{\exp\left(-||\mathbf{x}_i - \mathbf{x}_j||_2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-||\mathbf{x}_i - \mathbf{x}_k||_2 / 2\sigma_i^2\right)}$$

where the bandwidth of the Gaussian kernels σ_i is set such that the perplexity of the conditional distribution equals a predefined perplexity. Perplexity can be considered as a continuous analogue to the k in k-NN, to which t-SNE will try to preserve distances.

In the embedding space, the similarity between $\mathbf{y}_i$ and $\mathbf{y}_j$, denoted as q_{ij}, is modelled by a Student's t-distribution

$$q_{ij} = \frac{\left(1 + ||\mathbf{y}_i - \mathbf{y}_j||_2\right)^{-1}}{\sum_{k \neq l} \left(1 + ||\mathbf{y}_k - \mathbf{y}_l||_2\right)^{-1}}$$

Values of $\mathbf{y}$'s are learned by minimizing the Kullback-Leibler divergence (KL divergence) between the input distribution $\mathbf{P}$ and the embedding distribution $\mathbf{Q}$. We use gradient descent to optimize the differentiable cost function.

$$C = KL(\mathbf{P} \mid\mid \mathbf{Q}) = \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}}, \text{ with gradient } \frac{\partial C}{\partial \mathbf{y}_i} = 4 \sum_{j \neq i} \frac{\left(p_{ij} - q_{ij}\right)\left(\mathbf{y}_i - \mathbf{y}_j\right)}{\left(1 + ||\mathbf{y}_i - \mathbf{y}_j||_2\right)}$$

In summary, there are two major steps for t-SNE. First, compute input similarities p_{ij}. Second, compute embedding similarities q_{ij}, KL divergence, and optimize it using gradient descent.

2.2 Sequential Implementation

We developed our code base on the Multicore t-SNE implementation [1], which is in turn based on the original Barnes-Hut t-SNE implementation [2]. We chose this version mainly because the original implementation is unable to generate correct results.

2.2.1 Input Similarities

We use a vantage-point tree (VP tree) to compute input similarities, which helps to search for the k nearest neighbors of each point efficiently.

2.2.2 Output Similarities and Gradients

We can rewrite the gradient as

$$\frac{\partial C}{\partial \mathbf{y}_i} = 4\left(\sum_{j\neq i} p_{ij}q_{ij}Z\left(\mathbf{y}_i - \mathbf{y}_j\right) - \sum_{j\neq i} q_{ij}^2 Z\left(\mathbf{y}_i - \mathbf{y}_j\right)\right), \text{where } Z = \sum_{k\neq l}\left(1 + ||\mathbf{y}_k - \mathbf{y}_l||_2\right)^{-1}$$

, which splits the gradient into two parts. The first part can be interpreted as the attractive forces between points in the embedding, and the second part can be interpreted as the repulsive forces.

Note that p_{ij} can be pre-computed and reused in gradient descent, because the input data never changes. However, q_{ij} and the normalization factor $\sum_{k\neq l}\left(1 + ||\mathbf{y}_k - \mathbf{y}_l||_2\right)$ require re-computation in each iteration of the gradient descent.

Attractive forces. As mentioned earlier, the attractive force is computed between each point and its k nearest neighbors, where k equals to the perplexity. As perplexity is a constant much smaller than N (the number of points), computing attractive forces is relatively cheap ($O(Nk)$ time).

Repulsive forces. The computation of repulsive forces is expensive, as it requires computing all q_{ij}'s. Obviously, a direct implementation of t-SNE is slow, because computing q_{ij} takes $O(N^2)$ time. This can be approximated, and thus accelerated, using Barnes-Hut tree. The Barnes-Hut tree divides the embedding points into a quadtree, which group nearby points into cells (we also refer to cells as nodes, as they are implemented as nodes in the Barnes-Hut tree). The intuition is that when computing the interactions between a point and other points in a distant node, points in the node can be approximated by a single large point at the center of mass of the node. Precisely, a cell can serve as the summary/approximation for points inside if

$$\frac{r_{\text{cell}}}{||\mathbf{y}_i - \mathbf{y}_{\text{cell}}||^2} < \theta$$

where r_{cell} is the length of the diagonal of the cell, and $\mathbf{y}_{\text{cell}}$ is the center of mass of points in the cell. θ is a parameter that makes the trade off between accuracy and speed. Constructing the Barnes-Hut tree takes $O(N)$ time, and computing q_{ij} now takes $O(NlogN)$ on average.

2.3 Pseudo Code for Sequential Implementation

```
def tSNE(X, perplexity, num_iters):
    vp_tree = VPTree(X)
    kNNs = inputSimilarities(X, vp_tree, perplexity)

    Y = initY()
    for i = 1..num_iters:
        pos_f = computePositiveForces(kNNs, X, Y)
        neg_f = computeNegativeForces(Y, BHTree(Y))
        Y -= computeGradient(pos_f, neg_f)

    return Y

def inputSimilarities(X, vp_tree, perplexity):
    for x in X:
        knns = vp_tree.search(x)
        while not ok(perplexity):
            perplexity = computeNextPerplexity()

def computePositiveForces(kNNs, X, Y):
    pos_f = float[N]
    for x in X:
        for nn in kNNs[x]:
            D = p[x, nn] / (1 + euclidean_distance(Y[x], Y[nn]))
            pos_f[x] += D * (Y[x] - Y[nn])

    return pos_f

def computeNegativeForces(Y, bh_tree):
    def computeNegativeForces(Y, idx, bh_tree, neg_f):
        D = euclidean_distance(Y[idx], bh_tree.center_of_mass)
        r_cell = max(bh_tree.cell_length)

        if bh_tree.leaf() or m / sqrt(D) < theta:
            neg_f[idx] = ... // use center of mass as summary
        else:
            for c in bh_tree.children:
                computeNegativeForces(Y, idx, c, neg_f)

    neg_f = float[N]
    for y in Y:
        neg_f[y] = computeNegativeForces(Y, y, bh_tree, neg_f)

    return neg_f
```

2.4 Workload Analysis

As explained earlier, the workload can be divided into two major steps. **Step 1** builds a VP tree and computes p_{ij} between inputs. **Step 2** executes in iterations, where each iterations computes positive forces, negative forces, gradient, and update the embedding result.

We use `perf` to analyze the sequential program (running gradient descent for 1000 iterations) on the medium input. The program takes around 92 seconds. Step 1 takes 47 seconds and Step 2 takes 45 seconds.

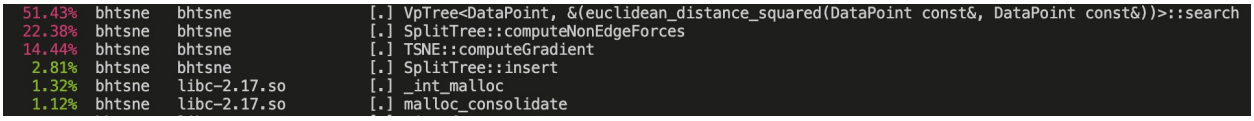


Figure 1: perf result of sequential program on medium input

As shown in Figure 1, we observe the following computation expensive components: searching on VP tree in Step 1, traversing BH tree (to compute negative forces) in Step 2, and computing gradient in Step 2.

Obviously, the sequential program can benefit from parallel execution. Step 1 can benefit from parallelism, because the loop in `inputSimilarities` are independent from each other. Step 2 can benefit from parallelism, because the loops in `computePositiveForces` are independent, and the loops in `ComputeNegativeForces` are independent as well.

3 Approaches

3.1 Parallel Loops with OpenMP

We use OpenMP to parallelize the independent loops in `inputSimilarities`, `computePositiveForces` and `computeNegativeForces`. Note `inputSimilarities` performs read-only operations on the VP tree, and `computeNegativeForces` performs read-only operations on the BH tree.

With 8 cores, the time spent on the medium input drops from 92 seconds to 23 seconds. Compared with sequential version, Step 1 time drops from 47 to 7 seconds, and Step 2 time drops from 45 to 16 seconds.

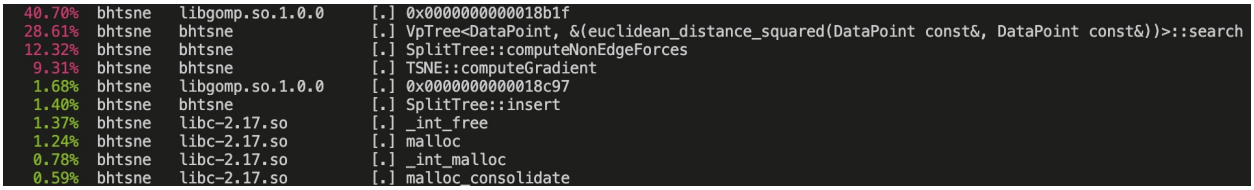


Figure 2: perf result of OpenMP program on medium input

Observe that `libgomp` takes over 40% of total execution time when parallelizing both Step 1 loops and Step 2 loops using OpenMP. Also, we make the two following observations: (1) On entering Step 2, the CPU utilization immediately drops from 100% to 40%, and (2) `libgomp` takes less than 1% of total time if we disable OpenMP for Step 2. This indicates that the suboptimal speedup is not caused by workload imbalance among worker threads or OpenMP overhead. We are unable to identify the root cause of this phenomenon, but we speculate that this is due to memory latency caused by Barnes-Hut Tree traversals, which are pointer chasing in nature and involve irregular memory access patterns.

3.2 Vectorizing Computations with ISPC

A further analysis of the perf report reveals that more than 14% of total time is spent on euclidean distance computation executed by `VpTree::search`. Consider the high dimensionality of input data ($>= 64$) as well as how frequently this function is executed, vectorizing the Euclidean distance computation using ISPC would further increases the parallelism of the program.

We also used ISPC to implement attractive force computation and gradient descent, since these also takes up 9.31% (under `TSNE::computeGradient`) of the total run time.

The Intel i7-9700 CPU on GHC machines supports AVX2, and we found that setting the ISPC compilation

target to `avx2-i32x16` gives the best performance. With 8 cores, the OpenMP+ISPC version reduces the time spent on the medium input from 23 seconds to 19 seconds. Step 1 time drops from 7 to 4 seconds, and Step 2 time drops from 16 to 15 seconds.

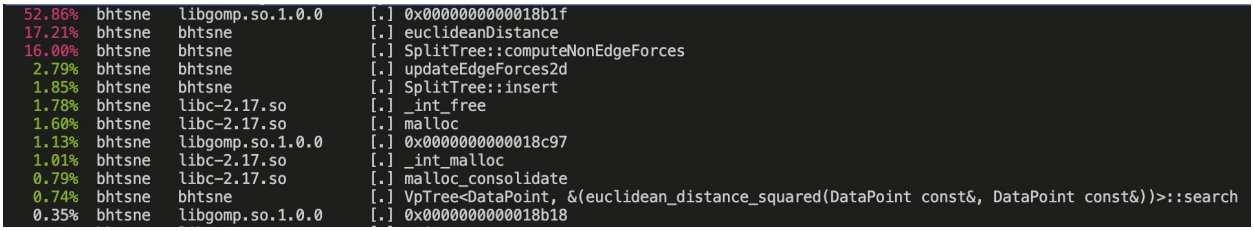


Figure 3: perf result of OpenMP+ISPC program on medium input

3.3 Parallelizing BH Tree Traversal with OpenMP task

With parallelized loops and vectorized euclidean distance computation, Step 2 takes over 75% of the total execution time and thus becomes the focus of our optimization efforts. Perf result shows that the major proportion of time spent on Step 2 is in function `SplitTree::computeNonEdgeForces` (the `computeNegativeForces` function in the pseudo code), which recursively traverse the BH tree to compute negative forces.

As BH tree traversal is a read-only operation, we implemented parallel tree traversal with OpenMP. Different from operations on regular/linear data structures like array, operations on irregular data structure like a tree cannot be easily parallelized with `#pragma parallel for`. We parallelized tree traversal with OpenMP task, which is suitable for parallelizing recursive executions.

However, this parallelization approach performed very poorly (over 20x slowdown), and perf result shows that most of time is spent on OpenMP runtime. We hypothesize this approach incurs significant OpenMP overhead because the amount of work executed at each node is small, and thus it is not worthy the overhead of OpenMP tasks. If the amount of computation required at each node is larger, this approach will probably achieve better result.

3.4 Parallelizing BH Tree Traversal with CUDA

We parallelized BH tree traversal for negative force computation, based on the idea of parallelizing the N-body problem with CUDA [3]. High-level steps for computing negative forces with CUDA are implemented as separate kernels, and listed as follows:

```
Kernel 1: Compute bounding box around all embedding points;
Kernel 2: Build BH tree (as a quadtree);
Kernel 3: Compute center of masses for tree nodes;
Kernel 4: Approximately sort the embedding points;
Kernel 5: Compute negative forces by traversing the BH tree.
```

Here are several optimizations that make this implementation efficient [3].

1. On CPU, irregular data structures like trees are typically implemented using dynamically-allocated heap objects, containing child-pointers to other objects. However, because dynamic memory allocation is expensive, we represent the tree using an pre-allocated array. Embedding points (leaf nodes) are allocated at the beginning of the array, and internal tree nodes are allocated at the end of the array. In particular, the root node is stored in the last slot of the array. Because we implement BH tree as a quadtree (each node has at most 4 children), it is easy to traverse the tree using index. We use -1 as the ”null pointer”.

2. Kernel 2 implements an iterative tree-building algorithm with light-weight locks (implemented with atomic operations). A fixed number of GPU threads are launched and embedding points are assigned to threads in round-robin fashion. When trying to insert a new embedding point, a thread traverses the tree from the root to the last-level internal node, and attempts to lock the appropriate child pointer (an array index) by atomically writing a special value -2 to it. If the locking succeeds (the atomic write succeeds), the threads inserts the embedding point by overwriting -2 with the corresponding index, thereby releasing the lock. If an embedding point is already stored at the location, the thread executes the following: create a new cell by atomically requesting the next unused index, insert the original and the new embedding point into the new cell, execute `__threadfence` to make the new cell visible to the other cores, and attach the new cell to the tree, which releases the lock. Threads that fail to acquire a lock retry until they succeed.
3. Kernel 4 sorts embedding points according to an in-order traversal, which essentially places spatially close points close to each other in the array (note that originally nodes/points are stored in allocation order). This is crucial for the performance of Kernel 5.
4. Kernel 5 is the most time-consuming kernel, which computes the negative forces by traversing the BH tree from the root. It launches a fixed number of GPU threads and assign embedding points to threads in roung-robin fashion. For each embedding point, the corresponding thread traverses a prefix of the BH tree to compute the forces. Theses prefixes are similar for spatially close embedding points, but different for spatially distant ones. Because threads in the same warp execute in lockstep, every warp effectively traverses a union of the tree prefixes of all threads in the warp. To avoid thread divergence, it is important to minimize the union of prefixes. This is the reason for having Kernel 4 sort the array and put spatially close nodes close to each other.

The force normalization factor is computed using `thrust::reduce`.

We replaced the CPU version of negative force computation with the GPU version, while keeping the positive force computation on CPU. Step 2 time spent on medium input drops from 15 seconds (with 8 CPU cores) to 3.2 seconds.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   67.65%  409.40ms      1000  409.40us  371.87us  584.03us  ForceCalculationKernel(float volatile *)
                   15.59%  94.368ms      1000  94.367us  80.128us  117.02us  TreeBuildingKernel(void)
                    5.90%  35.695ms      1000  35.695us  31.999us  40.576us  SummarizationKernel(void)
                    3.73%  22.549ms      6000  3.7580us     832ns  5.4080us  [CUDA memcpy DtoH]
                    3.57%  21.602ms      1000  21.602us  19.424us  27.904us  SortKernel(void)
                    1.73%  10.498ms      3019  3.4770us  1.1200us  3.7760us  [CUDA memcpy HtoD]
                    0.85%  5.1330ms      1000  5.1320us  4.9910us  5.4400us  BoundingBoxKernel(void)
                    0.34%  2.0607ms      1000  2.0600us  2.0160us  2.2080us  void thrust::cuda_cub::cub::DeviceReduceKernel<thrust::cuda_
cub::cub::DeviceReducePolicy<float, int, thrust::plus<float>>::Policy600, thrust::detail::normal_iterator<thrust::device_ptr<float>>, flo
at*, int, thrust::plus<float>>(int, float, thrust::plus<float>, thrust::cuda_cub::cub::GridEvenShare<float>, thrust::cuda_cub::cub::Devic
eReducePolicy<float, int, thrust::plus<float>>::Policy600)
                    0.27%  1.6562ms      1000  1.6560us  1.6000us  1.9200us  void thrust::cuda_cub::cub::DeviceReduceSingleTileKernel<thr
ust::cuda_cub::cub::DeviceReducePolicy<float, int, thrust::plus<float>>::Policy600, float*, float*, int, thrust::plus<float>, float>(int,
 float, thrust::plus<float>, thrust::cuda_cub::cub::DeviceReducePolicy<float, int, thrust::plus<float>>::Policy600, float*)
                    0.19%  1.1692ms      1000  1.1690us  1.1200us  1.2800us  void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub:
:__parallel_for::ParallelForAgent<thrust::cuda_cub::__uninitialized_fill::functor<thrust::device_ptr<float>, float>, unsigned long>, thru
st::cuda_cub::__uninitialized_fill::functor<thrust::device_ptr<float>, float>, unsigned long>(thrust::device_ptr<float>, float)
                    0.17%  1.0583ms      1000  1.0580us  1.0230us  1.4400us  InitializationKernel(void)
      API calls:   52.46%  537.56ms      3000  179.19us     885ns  661.73us  cudaDeviceSynchronize
                   28.52%  292.24ms      2011  145.32us  1.6580us  283.66ms  cudaMalloc
                   12.42%  127.29ms      8000  15.911us  6.3900us  98.097us  cudaMemcpy
                    3.61%  37.038ms      9000  4.1150us  2.9250us  234.25us  cudaLaunchKernel
                    1.18%  12.129ms      1000  12.129us  10.974us  16.741us  cudaMemcpyAsync
                    0.63%  6.4535ms      2011  3.2090us  1.7830us  231.61us  cudaFree
                    0.62%  6.3226ms      3000  2.1070us  1.8530us  45.269us  cudaFuncGetAttributes
                    0.11%  1.1633ms      2000     581ns     392ns  46.240us  cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
                    0.10%  1.0386ms      1000  1.0380us     921ns  3.6750us  cudaStreamSynchronize
```

Figure 4: nvprof result of using CUDA for negative force computation on medium input

Nvprof shows that a fair amount of time is spent on `cudaMemcpy`, because we interleaved computation on GPU and CPU, and thus must copy data to and from GPU. To reduce memory time, we implemented positive forces and gradient descent on CUDA, and thus moved the entire Step 2 to GPU. Step 2 time spent on medium input drops from 3.2 seconds (CPU+GPU) to 1.7 seconds. The proportion of time spent on

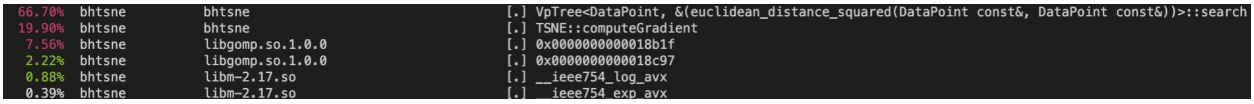


Figure 5: perf result of using CUDA for negative force computation on medium input

memory transfer drops from 12.42% to 0.08%.

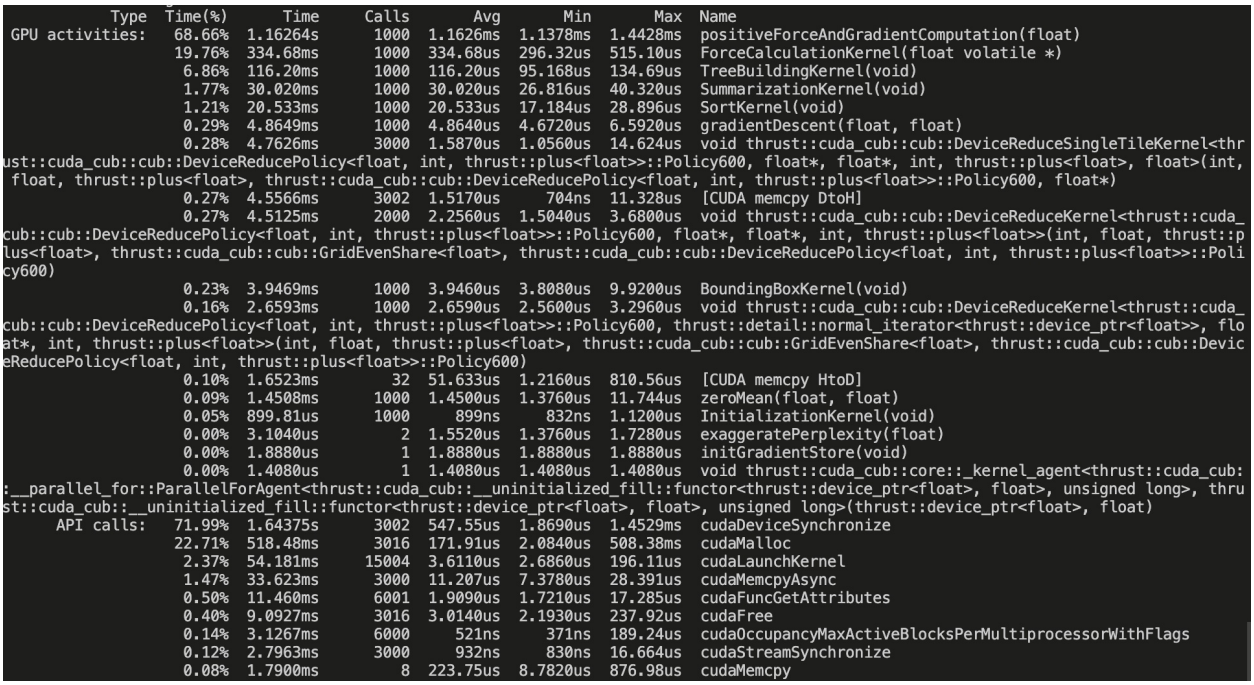


Figure 6: nvprof result of using CUDA for Step 2 on medium input

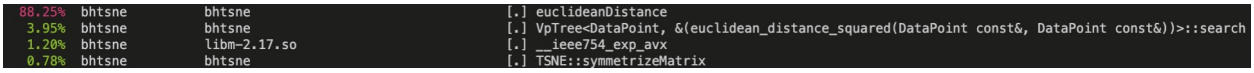


Figure 7: perf result of using CUDA for Step 2 on medium input

We further parallelized Step 2 by executing positive forces and negative forces computation in parallel on CPU and GPU in a heterogeneous manner, wait for both to finish before computing gradients. This reduces Step 2 time spent on medium input from 1.7 seconds to 0.9 seconds.

At this stage, Step 1 becomes the bottleneck of the algorithm and takes over 80% of total time. This concludes our parallelization exploration on t-SNE algorithm.

3.5 GPU-only Step 2

We further attempt to optimize Step 2 by moving positive force computation and gradient descent algorithms to GPU. As such, the entire Step 2 is executed on GPU and eliminates the need of communicating negative forces from GPU to CPU in each iteration. The resulting point positions are only communicated back to CPU after all iterations have completed.

4 Evaluations

4.1 Test Setup

We timed our code using `std::chrono::high_resolution_clock` for high-precision timing. We timed the two steps of the t-SNE algorithm separately.

All our experiments were run on the *GHC 73* machine. Due to time constraints, we did not run experiments on the Pittsburgh Supercomputer.

We used the following parameters throughout all experiments: random seed = 15618, reduced dimension = 2, max step 2 iterations = 1000, perplexity = 50, theta = 0.5.

Table 1 lists the 4 different test cases that we created from various sources to measure the performance of our code. These test cases cover the majority of use cases of t-SNE, where the source data has high dimensionality. We have one more simple test case for debugging only.

We note that all digit-based test cases contain only positive integers, while the CLIP-based test case contains real numbers.

Test case name	Source	Number of points	Point dimensions
easy_1797x64	Scikit-Learn digit data	1797	64
medium_10000x768	MNIST[4]	10000	768
hard_60000x768		60000	768
hard_62656x512	CLIP[5] image embeddings of a custom dataset	62656	512

Table 1: Test Data for Benchmarking Performance

We designed a simple binary data interface for 2-dimensional 32-bit floating-point array inputs and outputs, as defined by the following C++ struct:

```
struct DataInferface {
    int numPoints;
    int pointDimension;
    float data[]; // in row-major order
}
```

4.2 Results

Figure 8 compares the wall-clock time (run time) in seconds of our implementations using various parallelization approaches. It demonstrated that we have achieved significant performance improvements as compared to the sequential baseline.

Sequential Baseline Since OpenMP-based parallelization do not require changes to the sequential algorithm, we treat its single-core run time as the sequential baseline and calculated all subsequent speedups using this baseline.

The run time for the sequential baseline is shown in table 2. We observe that the run time of step 1 correlates super-linearly (likely $O(N \log N)$) to the number of points in the test case. It also correlates to the complexity of the test case input itself, as evident by the fact that the real-numbered test case takes significantly more time than the integer-numbered test case of similar size. The run time of step 2 is mostly correlated linearly or super-linearly with the the number of points in the test cases.

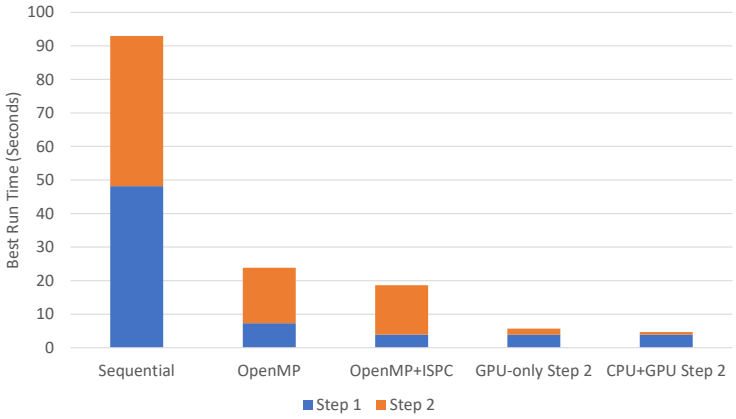


Figure 8: Program Run Time on Medium Test Case

Test Case	Step 1 Time (s)	Step 2 Time (s)	Total Time (s)
easy_1797x64	0.2518	6.2224	6.4742
medium_10000x768	48.1983	44.7489	92.9472
hard_60000x768	916.2615	392.0656	1308.3271
hard_62656x512	1482.9269	363.2149	1846.1418

Table 2: Sequential Baseline Run Time (Seconds)

OpenMP-only approach (section 3.1) Figure 9 shows the speedup achieved by this approach. We observe that the speedup achieved by Step 1 is mostly linear to the number of CPU cores. However, Step 2 is unable to achieve linear speedup to the number of CPU cores, even though both steps are fully parallellized by points in the test cases and do not involve critical sections beyond diagnostic information, which is disabled during benchmarking. We believe the sub-optimal speedup in Step 2 is caused by memory operations, as discussed in 3.1.

Due to suboptimal speedup of Step 2, the total speedup of the OpenMP-only approach is sub-linear.

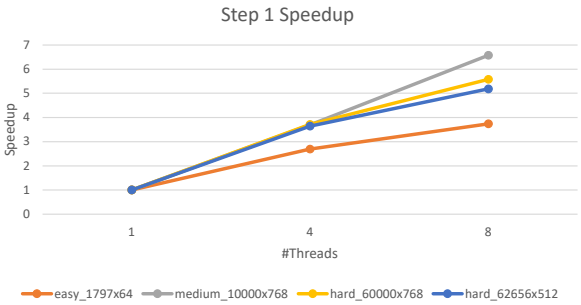
OpenMP+ISPC approach (section 3.2) Figure 10 shows the speedup achieved by this approach.

Utilizing SIMD for Euclidean distance computation gave Step 1 an immediate 3x 4x speedup in test cases larger than medium on a single thread. This is expected as the larger test cases have higher dimensionality, hence they would benefit more from this approach. However, as we increase the number of threads, the speedup provided by ISPC starts to plateau faster than the OpenMP-only approach. This is likely caused by other bottlenecks in the VP-tree-based nearest neighbor search and may be related to memory latency in tree node access as well.

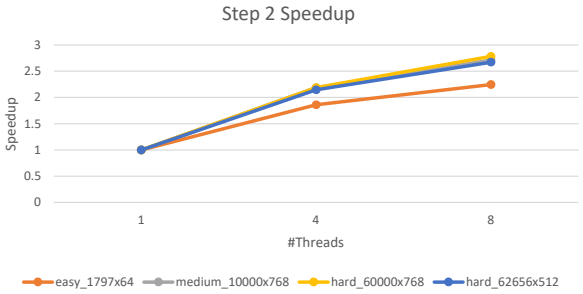
Since attractive force computation is not the main time-consuming component of Step 2, using SIMD did not significantly improve its overall performance. Nonetheless, we got a small performance improvement of 0.5x speedup. Similar to the OpenMP-only approach, The achieved speedup is sub-linear.

Overall, we are able to achieve super-linear total speedup with SIMD on the large test cases.

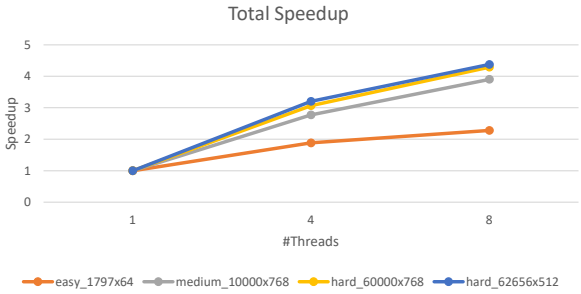
CPU+GPU Heterogeneous Computing approach (section 3.4) Figure 11 shows the speedup achieved by this approach. We are focused on optimizing Step 2 in this approach, and we immediately observed that utilizing CUDA for Barnes-Hut-based repulsive force computation gave a huge performance boost over all CPU-based versions. Although the CUDA-based Barnes-Hut implementation that we adapted is highly optimized, we initially expected that frequent memory data transfer between CPU and GPU required by



(a) Step 1



(b) Step 2



(c) Total

Figure 9: OpenMP-only Speedups

this approach would slow down its execution. The results proved that the communication overhead between CPU and GPU is not a bottleneck in most cases.

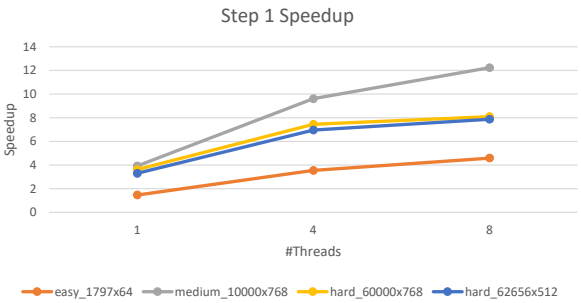
As expected, the smallest `easy` test case had a smaller and constant performance improvement throughout the number of CPU threads. We speculate that this is due to low arithmetic intensity associated with smaller number of data points.

The larger test cases benefited greatly from this heterogeneous approach, by effectively splitting the workload between CPU and GPU. We see a close-to-linear speedup improvement going from 1 thread to 4 threads, indicating that the bottleneck was CPU on a single thread. But going from 4 threads to 8 threads, we see that speedup improvement plateaus. This indicates that run time on CPU and GPU is more balanced with more CPU threads. With the heterogeneous approach, we are able to achieve 66x speedup for the 2 hard test cases using 8 threads.

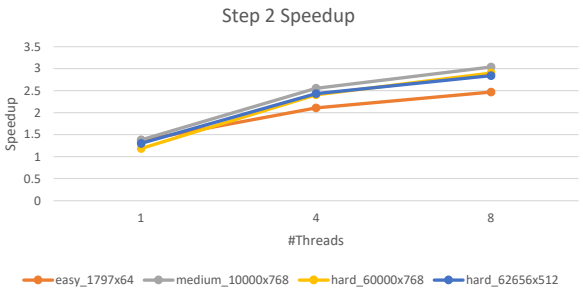
The total speedup, though super-linear, is affected by the lack of further optimizations of Step 1.

GPU-only approach (section 3.5) Figure 12 shows the speedup achieved by this approach. As expected, the number of CPU threads have no impact on the performance because we are merely using a single thread to synchronize with GPU during reduce operations. We observed that different test cases attained significantly different speedups on GPU. This is likely caused by the characteristics of test cases. A more complex tree in the case of the real-valued CLIP-based test case would cause more thread divergence and hence run slower than the integer-valued MNIST-based test case.

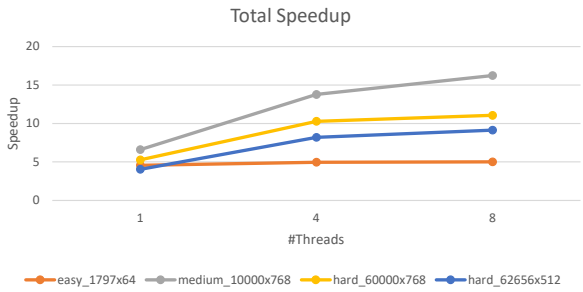
Comparing Figure 12(a) with Figure 11(a) from the previous experiment, we can see that when we use only one CPU thread, the GPU-only approach performs better than the heterogeneous approach. However, when 4 or more CPU threads are used, the heterogeneous approach performs significantly better. This indicates that Step 2's workload is sufficiently high such that using more compute resources, even heterogeneous ones, would result in better performance.



(a) Step 1

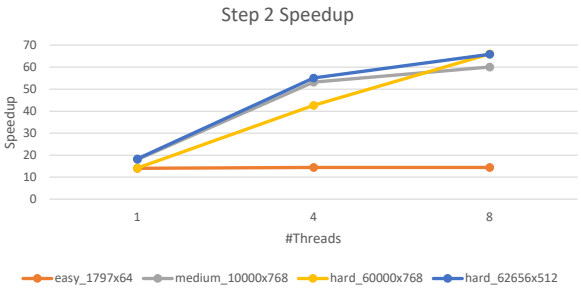


(b) Step 2

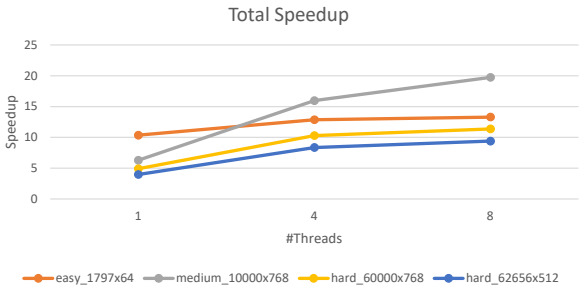


(c) Total

Figure 10: ISPC+OpenMP Speedups



(a) Step 2



(b) Total

Figure 11: CPU+GPU Heterogeneous Computing Step 2 Speedups

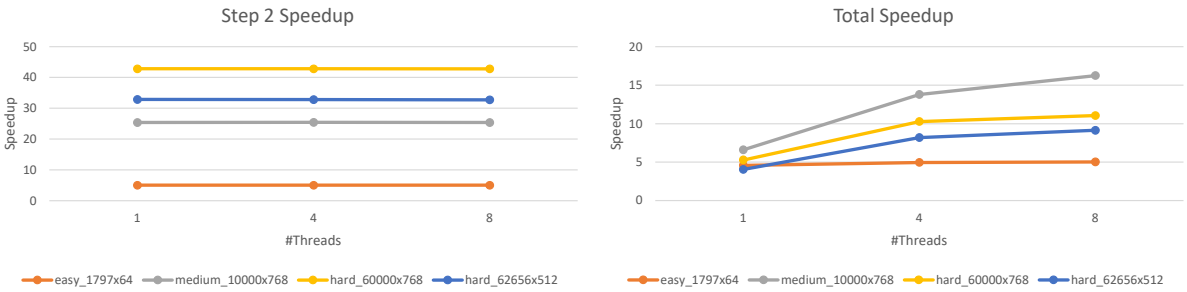
Overall, the total speedup is still affected by Step 1, which is now the main bottleneck of our implementation.

5 Conclusions and Future Work

In this project, we have successfully parallelized the Barnes-Hut t-SNE algorithm using 3 different parallelization approaches: multi-threading with OpenMP, Single Program Multiple Data with ISPC and massive parallelism with CUDA. We achieved significant speedups as compared to the baseline and analyzed the results thoroughly.

Overall, we have found that the optimized Barnes-Hut algorithm for CUDA can significantly outperform a regular CPU-based implementation. We also found that even with a GPU, harnessing the computing power of CPU heterogeneously can still lead to performance improvements.

We believe we have attempted to optimize Step 2 of the algorithm thoroughly. To further speed up the t-SNE algorithm, we need to optimize Step 1 as well. The Faiss [6] algorithm could be a good candidate for



(a) Step 2

(b) Total

Figure 12: Full GPU Step 2 Speedups

speeding up kNN search with GPU in Step 1.

Both team members, Yue Yin and Zhe Chen, contributed equally to this project.

References

[1] D. Ulyanov, "Multicore TSNE." https://github.com/DmitryUlyanov/Multicore-TSNE. Accessed: 2022-04-12.

[2] L. van der Maaten, "Accelerating t-SNE using Tree-Based Algorithms," *Journal of Machine Learning Research*, vol. 15, no. 93, pp. 3221–3245, 2014.

[3] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based barnes hut n-body algorithm," in *GPU computing Gems Emerald edition*, pp. 75–92, Elsevier, 2011.

[4] "MNIST in CSV." https://www.kaggle.com/datasets/oddrationale/mnist-in-csv. Accessed: 2022-04-10.

[5] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning Transferable Visual Models From Natural Language Supervision," *CoRR*, vol. abs/2103.00020, 2021.

[6] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.