Available online at www.sciencedirect.com

ScienceDirect

Procedia Manufacturing 53 (2021) 310–317

Procedia MANUFACTURING

www.elsevier.com/locate/procedia

49th SME North American Manufacturing Research Conference, NAMRC 49, Ohio, USA

A GPU-based Approach for Path Planning Optimization via Travel Length Reduction

Michael Borish[a]*, Charles Wade[b]

[a]*Oak Ridge National Lab, 2350 Cherahala Blvd., Knoxville 37923, USA*
[b]*University of Toledo, 2801 W. Bancroft St., Toledo 43606, USA*

* Corresponding author. Tel.: 1-850-543-4592; *E-mail address:* borishmc@ornl.gov

Abstract

Typically, before constructing an object with an additive manufacturing system, the 3D object must be sent through a process called slicing. Slicing converts a 3D object commonly in the form of an STL file into a set of layers by horizontally intersecting a plane with the object at various heights. At each height, called a layer, multiple 2D polygons can be generated. Each polygon represents a boundary for solid geometry and is called an island. Each island is then comprised of multiple path types in an attempt to optimally fill the polygon. To move between each island and each islands' paths, travels are inserted. Travels are simply motion by the system to move from one area of construction to another. Travels do not contribute to the construction of the object, and so, are considered wasted motion. In large-scale additive manufacturing, objects can be quite large and the distance between islands can be large as well. As a result, these travels can waste a significant amount of time. Ideally, travels would be as short as possible, however, computing global minimal travel paths is computationally expensive. To combat this problem, researchers at Oak Ridge National Lab developed a GPU-based approach to travel insertion based on a unique factoradic representation. This representation was then utilized by the GPU to solve the Traveling Salesman Problem (TSP). This algorithm was able to compute global minimal travel paths quickly resulting in faster object construction. A general investigation was also carried out to determine when a GPU vs CPU implementation would be beneficial.

© 2021 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Scientific Committee of the NAMRI/SME

Keywords: slicing; single path.

1. Introduction

The standard workflow for constructing an object with an additive manufacturing (AM) machine is to design a 3D object. Then, load that object in some type of slicing software. The slicing software then creates g-code, a standardized set of machine instructions, based on the object and various user settings. Finally, the g-code is uploaded to a specific machine to print the object.

Slicing is a necessary step in this workflow. Slicing is generally a purely geometric approach that itself consists of several stages [1, 2]. Before slicing can begin, an object is typically converted from CAD (computer aided design) to an STL (stereolithography) file. An STL represents all surface geometry using triangles [3]. This STL is then intersected or "sliced" by a horizontal plane at incremental heights. These heights correspond to the preferred layer height of construction on the AM machine. These intersections create multiple 2D polygons called islands that must be filled. This is generally accomplished through the use of several different types of paths including: perimeters, insets, skins, and infill. These paths are

2351-9789 © 2021 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Scientific Committee of the NAMRI/SME
10.1016/j.promfg.2021.06.034

modified by numerous user settings, but commonly begin with perimeters and insets.

Perimeters and insets are closed-loop paths generated by offsetting the boundary of the polygon by the bead width of the machine. Once these paths have been created, skins and infill are used to fill the remaining void and can take numerous forms. A typical example of a perimeter, multiple insets, and a line-style infill is shown in Fig. 1.

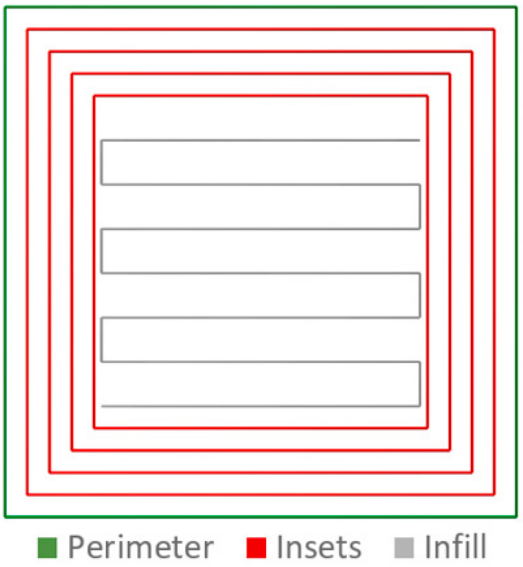


Figure 1: Typical pathing for square shaped object.

Regardless of the type and style of paths, many of those paths are commonly disconnected from one another as well as each island being separate from one another. As a result, travels are added as part of the slicing process.

Travels connect various build paths and one island to another. In terms of a standard FDM (Fused Deposition Modeling) process, a travel involves the machine stopping material flow, lifting the nozzle, moving to the next build path, lowering the nozzle, and resuming material flow. Because no material is being deposited, travels are wasted motions that do not contribute to the construction of an object. Travels also contribute to reducing the aesthetic quality of the object and a reduction in material properties [4]. Ideally then, travels should be minimized in number and as short as possible to reduce the negative impact on object quality and build time.

Further, many slicers compute travels via a simple greedy strategy. That is, move to the next closest island. This strategy can work well in many cases, especially small-scale construction where travel distances are short. However, this strategy does not guarantee the shortest travel times for a particular layer. When considering large-scale additive manufacturing, wasted time is amplified as the object and subsequent travels are much larger. The large size of the object and potential travels make identifying the shortest travel path more significant.

In order to calculate the shortest travels possible, researchers at Oak Ridge National Lab developed a GPU-based algorithm. This algorithm converts the problem of finding the minimal travel distance for a layer into a similar, well-known problem, the Traveling Salesman Problem (TSP). Computing the solution to the TSP is accomplished by utilizing a unique thread representation scheme to allow computation via GPU. To validate this approach, several simulations were run, and data gathered to highlight the algorithm's performance.

2. Related Work

A great deal of work has been conducting on reducing or removing travels from the path planning of objects constructed via AM. Single path algorithms are one example of this work. The idea is to convert all of the pathing in a layer into one single path without travels. Such a process is illustrated in work by Ding et al. [5, 6]. In this work, the authors lay out multiple types of adaptive pathing adjustments for wire arc AM and single path combinations are one of those adaptations.

Many approaches to path planning are purely geometric. However, work done by Michel et al. presented path planning that was feature-based and also attempted to generate a single path [7].

Work has also been conducted on extrusion-based processes and new path designs. In work by Jin, He, & Du, a new type of "wavy" path design was developed to address issues with the build process [8].

Manual approaches based on object design have also been investigated. One example is work done by Roschli et al. [9]. In this work, the authors discuss designing objects with very small cuts in appropriate locations in order to "trick" standard slicing software into creating a single path.

This previous work relates directly to the algorithm presented here. While the ideal goal of all this work is object construction without travels, the fact remains that some geometries will simply require their presence. However, our work builds upon this previous research by attempting to optimize the existing travels as much as possible.

This work utilizes the GPU for optimizing the choice of travels. Most common processes of slicers are CPU-based, however, a great deal of effort has also been spent on investigating the appropriate use of the GPU. For example, in work done by Mao, Chen, & Wang a procedure for utilizing the GPU to calculate adaptive slicing parameters was utilized [10]. In work by Zhang et al., the process of calculating cross-sections was implemented entirely as a GPU-based algorithm as opposed to the traditional CPU-based algorithms [11]. In this work, the segments that comprised the triangular faces of the mesh were pre-processed by sorting and assignment to buckets. Each of these buckets could then be computed on the GPU. A similar approach to Zhang et al. was used by Rebaioli et al. to speed-up cross-sectioning for their system as well [12]. These approaches realized a significant speed-up in one of the primary steps of converting 3D objects to the machine. Our work builds upon this by addressing one of the post-processing steps that typically occurs after cross-sectioning and build path planning: adding travels.

Additionally, approaches have also moved beyond common slicing techniques. In work by Ogayar-Anguita et al., a GPU approach was utilized to convert objects into voxels [13]. These voxels were later used for machine learning training, however, voxel-based construction approaches are becoming more

common. In work by Lefebvre, the entire slicing process was reimagined on the GPU using an A-buffer lookup [14]. In the A-buffer approach, cross-sectioning and hull generation can be seen as steps of the rasterization process. Approaches have also investigated cooperative slicing such as work by McPherson and Zhou [15]. In this work, the slicing process was broken into chunks that were later assigned to multiple printing agents in an attempt to speed-up the construction process.

Lastly, work has also tackled the Traveling Sales Problem (TSP) directly both within and outside of the application area of 3D printing. Details will be provided in the next section, but in general, direct computation of TSP is very difficult. As a result, a great deal of work has focused on a class of algorithms generally described as heuristic solvers. These algorithms compute an approximate solution to TSP while attempting to side-step some of the difficulty of the problem. That is, they compute an almost ideal solution much faster than attempting to compute the true, ideal solution much slower. One of the older, well-known solutions is Christofides [16]. Modern iterations of the heuristic approach are numerous and varied. Work by Fok et al. utilized an any colony approach [17]. Ant colony optimization (ACO) approaches treat the TSP as a graph solved in a manner similar to how ants send out multiple "agents" to identify good routes that are reinforced by additional ants. Similarly, various machine-learning approaches utilizing reinforcement learning via neural nets and genetic algorithms have been proposed [18, 19]. The work presented here builds upon the GPU-based heuristics approaches presented, but in contrast to this work, solves the TSP problem directly.

3. Implementation Theory

As mentioned previously, there are two general locations that travels are inserted during the slicing process. The first location is between each island as the machine must traverse empty space between each solid piece of geometry. The second location is between the multiple paths that comprise a single island. This algorithm will focus on the first case. Commonly in the case of FDM-style build processes, the object is made of multiple closed-loop contours that represent the perimeter and insets as was shown in Fig 1. As a result, a greedy strategy of simply choosing the next closest point to the current location on a path provides very good results and is the "default" setting in many slicers. However, a greedy strategy does not guarantee the shortest distance and significant savings can be recognized by computing the global shortest travel path. Consider the example geometry shown in Fig. 2.

Assuming the origin is located at the bottom of Fig. 2, greedily choosing the next closest island will result in selecting the bottom V shape, then traveling down one of the branches, and finally moving over to the other branch. An expected order would be C-B-A-D-E. However, this is not the shortest set of travels. In this case, the shortest path would be A-B-C-D-E or the inverse.

To compute the shortest path, the problem of adding minimal travels between all islands in a particular layer can be reformulated as the Traveling Salesman Problem (TSP). TSP is

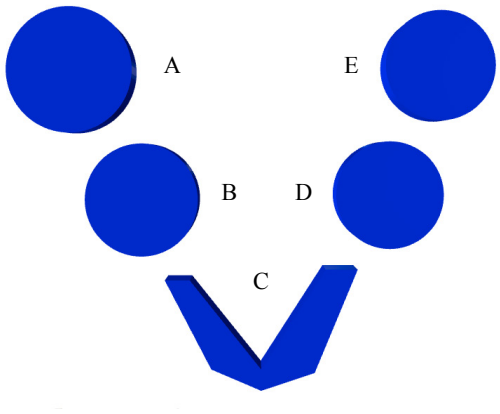


Figure 2: Example geometry. The shortest path between these islands is to begin at a corner and work around the V shape.

a well-known problem in computer science and is generally stated as follows: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?". Alternatively, in terms of graph theory, the problem asks for the shortest path that visits all nodes in the graph. TSP is contained within a family of algorithms classified as NP-hard. In computer science, NP-hard algorithms are problems for which there currently exist no known way to solve in polynomial time. As a result, computation of the optimal solution requires computation of all permutations. For TSP, that means the runtime is bound by O(n!) and grows factorially as the problem size increases. However, there are heuristics and pruning approaches that allow computation to within a reasonable percentage of optimal. For example, Christofides guarantees a solution that is within 1.5 times of optimal with a runtime bounded by $O(n^2 * \log(n))$. All heuristic approaches make similar tradeoffs in that solutions are within some percentage of optimal while requiring less runtime to identify that solution. Such a tradeoff means that heuristic-based approaches will in general be able to deal with larger problem sets than computed the ideal directly.

To combat the problem set size, conversion of the islands in a layer to a vertex representation for TSP is accomplished via a simple transform of center of mass. By finding the center of mass for each island, each island can be represented via a single vertex that drastically reduces the permutation set. However, the permutation set can still easily be quite large so a GPU implementation of TSP was pursued compared to the common CPU implementation.

4. Implementation Details

This algorithm processes the exact solution to TSP utilizing Nvidia's CUDA GPU library [20]. To compute this solution there are two major phases: generation of permutations and reduction. In utilizing a GPU, the approach to generate the permutations is quite different compared to the typical approach for a CPU-based slicer.

To accomplish permutation calculation quickly, each thread on the GPU must be able to calculate a unique permutation

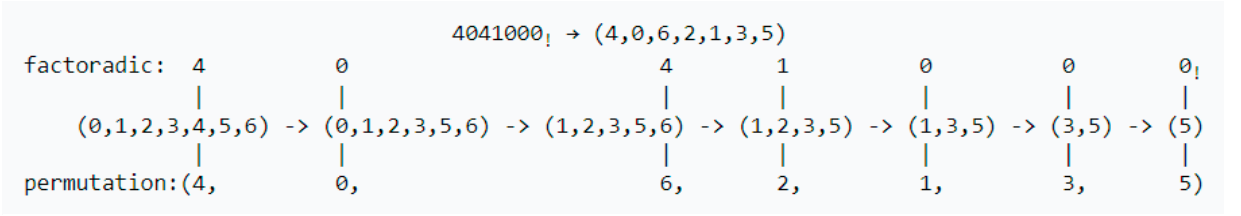


Figure 3: Factoradic conversion to permutation. The factoradic 4041000 is calculated by division by increasing radix. Then, that representation serves to identify indices in a reference list that can be converted to a unique permutation.

given only a thread ID. The architecture provided by CUDA allows queuing a total number of threads that can exceed the maximum number of supported simultaneous threads on the GPU. As a result, each thread must be pre-loaded with all necessary information and be able to compute a permutation's total distance without synchronizing with any other threads. This allows maximal occupation of the CUDA cores on the GPU as each thread calculates the distances between islands in isolation.

In order to generate permutations in isolation and guarantee that each thread is operating on a unique permutation, a conversion to a factorial numbering system was utilized. A factorial number system, or factoradic representation, guarantees that a specific decimal number will correspond to a unique permutation. A full discussion of a factorial numbering system is outside the scope of this paper, but there are several wikis with excellent primers [21, 22]. However, a simple example will be described in the context of the discussed algorithm.

To create a unique permutation, several steps are involved:

1. The bounds of the factorial representation are first calculated since the number of islands and thus number of permutations are known.
2. Both a reference and output row are generated. The reference row includes all island indices while the output row will hold the unique permutation for a thread.
3. The thread ID is converted from the decimal number system into its factoradic representation.
4. The thread consumes the first digit of the factoradic and uses it as an index to pick a value to remove from the refence row.
5. This value is appended to the output vector.
6. The thread repeats steps 4-5 until there are no digits left in the factoradic.
7. The resulting output vector contains the nth permutation.

To put these steps into a concrete example, consider the example in Fig. 3 [22]. To generate the factoradic number 4041000, suppose there are seven islands in a layer leading to 7! = 5,040 permutations. So, step 1 and 2 sets up the appropriately sized and populated reference and output vectors. Now, suppose we are on thread ID 2982. 2982 can be converted to its factoradic representation through successive divisions by increasing radix. 2982 divided by 1, 2982 divided by 2, 2982 divided by 3, and so on. The remainder of these divisions organized right to left convert to the factorial number 4041000. Now as Fig. 3 shows, the thread reads these numbers left to right one digit at a time as per steps 4-6. First is four, so the fourth index from the reference row is removed and placed into the output vector. Then zero is read and the zeroth index is removed from the reference row and placed into the output vector. And so on. Ultimately, the unique permutation (4, 0, 6, 2, 1, 3, 5) is created. This process continues for every thread leading to full coverage of all permutations calculated in isolation.

After the permutations are created, each thread simply iterates over its unique permutation to calculate the total Euclidian distance. This distance is then stored in a global memory array indexed by thread ID. With all distances calculated, the second major phase, reduction, can begin.

The Thrust library is a built-in toolkit provided by CUDA and provides parallel GPU reduction capabilities [23]. Reduction is the simple process of reducing the size of a vector according to some operation. In this case, the algorithm waits for all threads to complete their distance calculation. The vector with thread ID indexed distances is then sent for reduction via a min operation. That is, the vector is successively reduced in size, finding the min at each step. Ultimately, the reduction function of the Thrust library will return the global min. With the minimal distance tour in hand and associated thread ID, the algorithm can determine the permutation of islands that resulted in that particular thread's ID. From that permutation, the shortest set of travels can be added.

5. Experiment

To demonstrate the efficacy of the algorithm, an object was sliced conventionally, that is, using a greedy, next closest island algorithm and with the GPU implementation of the TSP algorithm described in the previous section. Results were then compared for distance and time.

Though the GPU does perform computations quicker than the CPU, there is additional latency involved in transferring and receiving results from the GPU. A general investigation was also carried out to determine the inflection point at which the GPU provides benefit to the pathing computation. Both a CPU and GPU implementation of the TSP algorithm were compared. Islands were generated in random positions within the build volume for slicing and the total number of islands varied from two to eleven. An example of the geometry visualized for one of the trials is shown in Fig. 4.

For each of the island totals from two to eleven, three trials were run. This resulted in a combined thirty trials. The time

taken to compute the pathing was recorded and averaged for both the CPU and GPU.

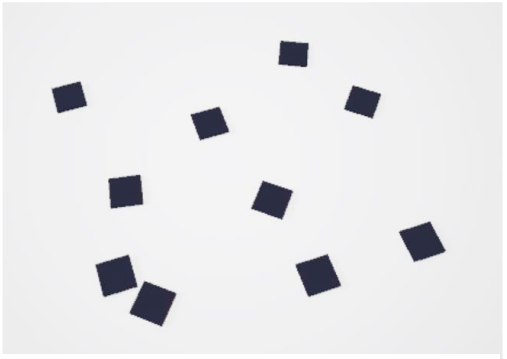


Figure 4: Example geometry for travel calculation.

A preliminary investigation into the use of GPU-based solution for the nearest neighbor problem was also conducted. A simplifying assumption of the previously described algorithm is to evaluate the center of mass of an island for distance purposes. Then, simply connect to that island. A logical extension would be to evaluate all the vertices that comprise the perimeter of the polygon and connect to the closest vertex in a chosen island. However, there are typically many vertices and so a comparison of nearest neighbor calculations was conducted to determine the inflection point for GPU vs CPU use. Multiple totals of vertices were evaluated including 1000, 2000, 3000, 4000, 5000, and 6000. The computation for nearest neighbor was conducted three times for each total, and the time results were averaged.

Both general investigations were carried out on a Dell workstation with an Intel Xeon W-2133 CPU processor, 32GB of RAM, and an Nvidia QUADRO GPU with 5GB of VRAM.

6. Results and Discussion

6.1. Practical Example

A practical example of a V-shaped object was shown in Fig. 2. Naturally, numerous variables contribute to overall build time including physical properties of the system such as velocities and accelerations, interruption by table motion, ratio of travel to build motions, as well as the geometry itself including sparsity and pathing density inside the islands. This geometry was meant to provide a basic example of a typical large-scale object.

In Fig. 5A and 5B, two different travel patterns are shown. The first is the commonly used next closest island algorithm while the second is the global minimum calculation. The distance traveled by the machine was excluded for both cases as the object was in the same location in the build volume when it was sliced. For the first travel algorithm, the total travel distance was found to be 89.1 inches while the second travel algorithm was found to be 80.8 inches. This corresponds to a travel distance that is approximately 9% shorter. Further, the object itself is approximately 12 inches in each axis. Additionally, the time estimate calculated by the slicer shows a savings of approximately three seconds. For this example object, approximately 51% vs 49% of the time was spent traveling for each travel approach respectively. That is, the global minimal path results in spending approximately 2% less time traveling for this example. Crucially, these distances and time savings are for a single layer. Typical large-scale objects of this size contain approximately 100 layers. Thus, the individual layer time saving amounts to several minutes of construction time saved. For comparison, if this object were scaled to two or three feet, approximate layer time savings would amount to 5.7 and 8.2 seconds, respectively. The number of layers would also scale to 200 and 300 respectively, so over the course of construction, time savings amounts to 19 and 41 minutes, respectively. Generally, as the object continues to grow, more savings will be seen.

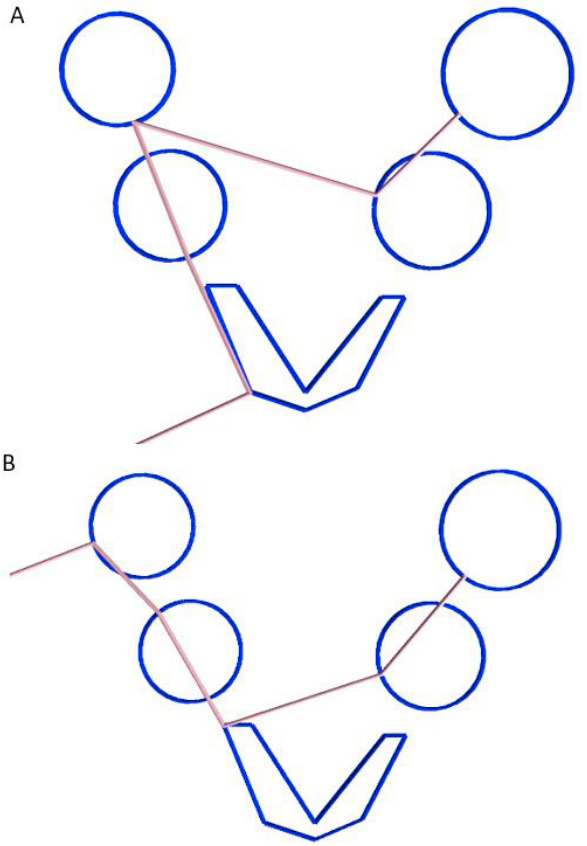


Figure 5: A) Next closest pathing where islands are greedily chosen. B) Global minimum where the true shortest path is calculated

In terms of large-scale AM, objects of two to four feet with a few hundred layers are typical, though much larger objects are constructed. As a result, these minor savings are more significant over larger objects that are typically several feet wide, several hundred layers, and require a full day of printing to complete. Additionally, more extreme geometries, in terms of travel length, would see additional savings. The converse will also generally hold. Objects that are relatively small or dense and spend a significant portion of time building as opposed to traveling will see fewer benefits.

6.2. CPU vs GPU Comparison

The first general comparison performed was to determine the inflection point at which the GPU implementation provides advantages over the CPU implementation. The results are shown in Table 1 and visualized on the next page in Fig. 6A.

Island Total	Average Time Per Layer CPU (ms)	Average Time Per Layer GPU (ms)
2	0.309	14.597
3	0.825	13.688
4	3.257	13.429
5	13.205	15.007
6	74.433	16.961
7	407.913	30.068
8	3005.792	63.789
9	26248.721	503.904
10	215489.581	4033.762
11	2383939.204	32999.112

Table 1: Time taken to compute ideal path on CPU and GPU for various island totals.

Commonly, most slicers implement their algorithms for processing by the CPU. As this investigation shows, even when computing the ideal travel paths, if the layer has a small number of islands, the results in terms of time are perfectly acceptable. The dominating factor for the GPU in these cases is the latency in transferring information. However, with additional islands, the time taken to compute the ideal path grows drastically. When 11 islands are present, the time taken to compute the ideal path for a CPU and GPU are nearly 40 minutes and 33 seconds, respectively.

The inflection point for computation time occurs once a layer contains at least six islands. At this point, the number of permutations is high enough that the additional latency of transferring and receiving information from the GPU is overshadowed by the amount of time necessary to compute the result. Further, as seen from Table 1, once a reasonable number of islands is present, the time taken to compute the solution grows quickly. This time growth is owed to the NP-hardness of the problem, and so, the problem set grows factorially with an increase in size of the input. Overall, with the exception of very small problems, the GPU implementation provides significantly superior performance in terms of time taken when calculating the minimal travel paths and should be the preferred approach.

As mentioned previously, the algorithm presented is a brute-force parallel implementation. While this algorithm will provide the exact solution to the TSP problem, very large data sets such as a lattice geometry with 100 islands will still require significant computation time. For situations such as this, heuristic approaches such as those discussed in the related work will provide better performance at the cost of correctness. Still, recall the complexity of this algorithm as O(n! / p). Because this algorithm is directly affected by the number of GPU cores present to provide computation, no changes are needed in order to take advantage of ever-increasing hardware capabilities. Further, because of the factoradic transformation that allows computation in isolation, this approach lends itself to distributed and cluster-based implementations.

The second general investigation that was carried out was to evaluate the performance of GPU vs CPU implementation of nearest neighbor. This investigation was meant as a practical extension of the algorithm. Recall in the implementation discuss that the algorithm evaluates islands by their center of mass. Extreme edge cases of geometry or concentric geometry could potentially fail. Concentric geometry is easily handled, however, other edge cases may still cause issues. In those cases, evaluation of the entire island's geometry would be necessary. Additionally, once the appropriate order of islands has been established by the algorithm, the specific vertex must be chosen. As already discussed, evaluating every vertex for TSP is not feasible. To facilitate the nearest-neighbor evaluation, the closest vertex in a particular island would need to be identified. The results from this investigation are summarized in Table 2 on the next page and visualized in Fig. 6B.

Vertex Total	Average Time Per Layer CPU (ms)	Average Time Per Layer GPU (ms)
1000	369.891	29.871
2000	1350.514	60.178
3000	2881.400	89.843
4000	5434.914	138.575
5000	8391.984	190.362
6000	11900.454	255.806

Table 2: CPU vs GPU Nearest neighbour calculation for given vertex total.

As seen from these results, for all vertex counts, the GPU outperformed the CPU implementation. Additionally, the vertex counts were not particularly high as many commonly sliced objects will have this many vertices. Similar to the previous case, in all but the simplest of situations, the GPU will outperform the CPU. These results are in line with the commonly accepted knowledge that large problems that can be parallelized are better calculated using a GPU. Thus, future work for dealing with remote edge cases will focus on the use of the GPU.

7. Conclusion

As this work shows, in order to achieve a parallel-brute force GPU implementation of TSP, a factoradic transformation was used. Using a factoradic representation allows calculation of permutations in isolation. This GPU implementation for the TSP problem can be utilized to calculate a global shortest travel path. Calculating the shortest travels for a particular layer when constructing an object via AM methods is attractive for the reduced build time that results. The savings realized from this approach increase as the size of the object increases and can lead to significant time savings. This makes the approach particularly attractive for large-scale AM.

Further, general investigations into the use of the GPU vs CPU for calculations of the minimal travels paths and nearest neighbor show the GPU outperforming the CPU in all cases except for the most simplistic.

This work will be investigated and expanded upon in the future. The complexity of slicing will only increase, and this

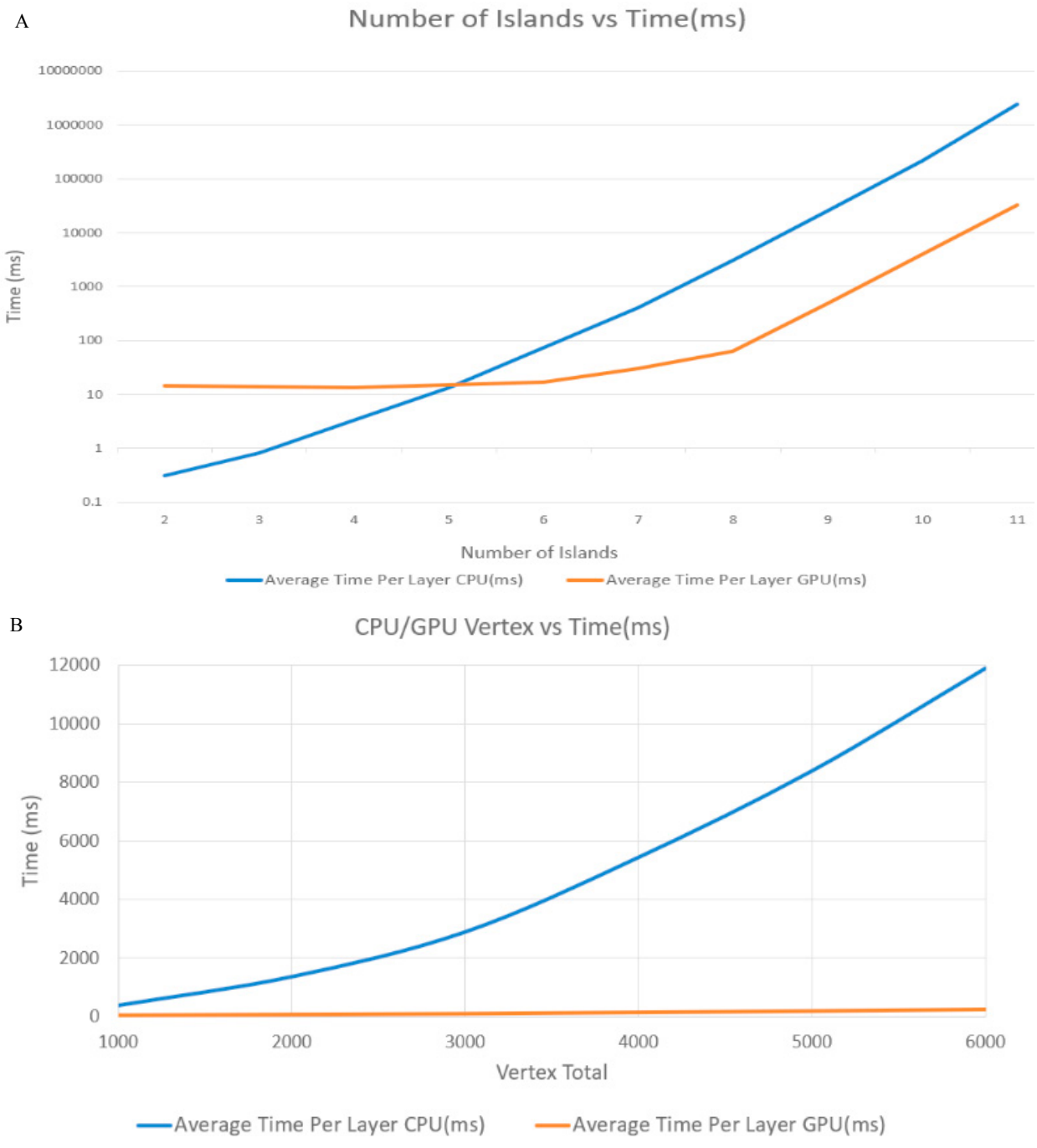


Figure 6: A) Comparison for time taken to calculate global shortest path on the CPU vs GPU for specific number of islands. B) Comparison of CPU vs GPU calculation for nearest neighbour for a given number of vertices.

additional complexity will require the capabilities of the GPU. One clear next step is the evaluation and alterations necessary to deal with edge cases. Additionally, with the added complexity path optimization based on additional factors beyond the simple geometric approach could be investigated. Further, future work will investigate the application of the factoradic representation to other path planning problems for parallel computation.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Energy Efficiency and Renewable Energy, Office of Advanced Manufacturing, under contract number DE-AC05-00OR22725

References

[1] Wang, W., et al. (2014). Saliency-Preserving Slicing Optimization for Effective 3D Printing. Computer Graphics Forum, 33(5), 1 – 12.

[2] Suh, Y., Wozny, M. (1994). Adaptive Slicing of Solid Freeform Fabrication Processes. Solid Freeform Fabrication, 5, 404-411

[3] Chen, Y. H., et al. (1999). Generation of an STL File from 3D Measurement Data with User-Controlled Data Reduction. The International Journal of Advanced Manufacturing Technology, 15(2), 127 – 131.

[4] Chesser, P., et al. (2019). "Extrusion control for high quality printing on big area additive manufacturing (baam) systems." Additive Manufacturing 28: 445-455.

[5] Ding, D., Pan, Z. S., Cuiuri, D., & Li, H. (2014). A tool-path generation strategy for wire and arc additive manufacturing. The international journal of advanced manufacturing technology, 73(1-4), 173-183.

[6] Ding, D., Pan, Z., Cuiuri, D., Li, H., van Duin, S., & Larkin, N. (2016). Bead modelling and implementation of adaptive MAT path in wire and arc additive manufacturing. Robotics and Computer-Integrated Manufacturing, 39, 32-42.

[7] Michel, F., Lockett, H., Ding, J., Martina, F., Marinelli, G., & Williams, S. (2019). A modular path planning solution for Wire+ Arc Additive Manufacturing. Robotics and Computer-Integrated Manufacturing, 60, 1-11.

[8] Jin, Y., He, Y., & Du, J. (2017). A novel path planning methodology for extrusion-based additive manufacturing of thin-walled parts. International Journal of Computer Integrated Manufacturing, 30(12), 1301-1315.

[9] Roschli, A., et al. (2019). Creating Toolpaths without Start and Stops for Extrusion-based Systems. Solid Freeform Fabrication Symosium Proceedings.

[10] Mao, H., Kwok, T. H., Chen, Y., & Wang, C. C. (2019). Adaptive Slicing based on efficient profile analysis. Computer-Aided Design, 107, 89-101.

[11] Zhang, Xipeng, et al. A GPU-based parallel slicer for 3D printing. 2017 13th IEEE Conference on Automation Science and Engineering (CASE). IEEE, 2017.

[12] Rebaioli, Lara, et al. Process parameters tuning and online re-slicing for robotized additive manufacturing of big plastic objects. Robotics and Computer-Integrated Manufacturing 55 (2019): 55-64.

[13] Ogayar-Anguita, C. J., Rueda-Ruiz, A. J., Segura-Sánchez, R. J., Díaz-Medina, M., & García-Fernández, Á. L. (2020). A GPU-Based Framework for Generating Implicit Datasets of Voxelized Polygonal Models for the Training of 3D Convolutional Neural Networks. IEEE Access, 8, 12675-12687.

[14] Lefebvre, Sylvain. IceSL: A GPU accelerated CSG modeler and slicer. 18th European Forum on Additive Manufacturing (AEFA'13). 2013.

[15] Jace McPherson and Wenchao Zhou, (2018). A chunk-based slicer for cooperative 3D printing", Rapid Prototyping Journal.

[16] Christofides, N. (1976). Worst-case analysis of a new heuristic for the travelling salesman problem. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.

[17] Fok, Kai-Yin, et al. Accelerating 3D printing process using an extended ant colony optimization algorithm. 2018 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2018.

[18] D'O Costa, Paulo R., et al. Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning. Asian Conference on Machine Learning. PMLR, 2020.

[19] Alipour, Mir Mohammad, et al. A hybrid algorithm using a genetic algorithm and multiagent reinforcement learning heuristic to solve the traveling salesman problem. Neural Computing and Applications 30.9 (2018): 2935-2951.

[20] NVIDIA, Vingelmann, P., & Fitzek, F. H. P. (2020). CUDA, release: 10.2.89. Retrieved from https://developer.nvidia.com/cuda-toolkit

[21] Factorial numeral system (2020). OEIS Foundation. https://oeis.org/wiki/Factorial_numeral_system

[22] Factorial Number System. Wikipedia, Wikimedia Foundation. 9 October 2020. https://en.wikipedia.org/wiki/Factorial_number_system

[23] Thrust, (2020). CUDA, release; 11.1.1. https://developer.nvidia.com/thrust