Accelerating Delaunay Mesh Generation Using Tensor Cores

Calvin Weaver
dept. Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
cweaver1@stevens.edu

Hang Liu
dept. Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
hliu77@stevens.edu

***Abstract*—Two-dimensional triangular mesh generation is a computationally intensive task that stands to benefit from GPU parallelization. Tensor Cores are a relatively new specialized GPU hardware feature that accelerate low-level matrix multiply and accumulate operations. This project proposes a method for applying Tensor Core operations to mesh generation by simplifying the triangle circumcircle test and performing many of these tests in parallel via Tensor Core executions. The metrics gathered through this investigation show signs that this approach is viable, though much work is still required to navigate several issues including: data precision, matrix post-processing, vertex batch size, launch overhead, and GPU memory transfer limits.**

Keywords—Tensor Core, mesh generation, graphics processing unit, GPU, Delaunay, triangulation, hardware acceleration

I. Introduction

Within the last five years, significant advances have been made in the realm of high performance, widely available graphics processing unit (GPU) hardware. Modern NVIDIA RTX-series GPUs now feature dedicated hardware components for both extremely high-throughput matrix operations (Tensor Cores) and fast handling of operations related to light bounce tracing through a geometrically complex scene (Ray Tracing or RT Cores) [1]. The applications of this specialized hardware are still being recognized, and there are many fields that are only just beginning to employ its strengths to tackle historically difficult tasks.

One such problem that could potentially benefit from the specific operations provided by this hardware is that of mesh generation. Two-dimensional meshes (sometimes called ‘surface meshes’ or ‘polygon meshes’) are widely used to model the surface properties of objects by subdividing surfaces into large numbers of similarly-sized cells. By simulating each cell independently, computer algorithms can accurately estimate how geometrically complex objects will react to things like light bounces, structural forces, and temperature changes.

Typically, meshes are formed by linking raw point cloud vertices into a set of similar polygons (usually triangles). The limitations of modern data processing and simulation algorithms often impose strict requirements on the way that these vertices are connected. Mesh cells are almost never allowed to overlap: each cell must represent a unique collection of vertices such that the all parts of the surface are uniquely mapped to only a single cell. In simulation applications, this is important so that the entire surface can be simulated exactly once. When possible, cells must also be consistently sized and avoid excessively sharp angles. Irregular cell distributions or cell deformities are often linked to inconsistent outputs produced by the algorithms that process the surface. An example of this issue would be visual rendering: if a square graphic is to be mapped to the surface of an irregular cell, the per-pixel color sampling of the graphic will be drastically skewed, which will lead to obvious visual artifacts on the final mesh render.

Algorithms that link vertices into surface meshes must do so in a way that adheres to these rules. This problem is compounded by the fact that, in general, meshes are generated from extremely large datasets (think millions of unique vertices). Vertex maps of this scale are necessary so that the surface can be modeled in as much detail as possible. These criteria are the foundation for the highly complex problem that is regular surface mesh generation.

Many studies have investigated ways to improve the efficiency of mesh generation algorithms. Most fast mesh generation algorithms include steps that partition the mesh into digestible chunks that can be built in parallel. This

process is made difficult by the fact that the sections of a mesh are often interdependent and require that information be shared between partitions during triangulation. Methods have been developed to circumvent this by optimally constructing mesh divisions first [2] or using an intricate alternating grid strategy to synchronize partition accesses [3] or to track low-level dependencies in real time [4]. Additional investigations have explored using advanced algorithms such as K-means clustering [5] or star splaying [6] to cleverly divide the mesh so that partitions more accurately reflect the local mesh properties.

The rest of this paper will cover the following topics, in order: 1) the problem description, 2) the algorithm design process and tradeoffs, 3) an analysis of the measured algorithm performance, and 4) a brief conclusion.

II. Problem Statement

A common goal for mesh generation is to build a triangular mesh so that all triangles satisfy a criterion known as the Delaunay condition, which states that for all triangles on the mesh, no other vertex may reside inside the circumcircle formed by that triangle's vertices. This simple rule is widely accepted as the primary quality standard that defines a surface mesh's usability in most common applications. As such, this project aimed to develop an algorithm that produces a Delaunay-complaint triangular mesh.

The applications of cutting-edge dedicated GPU hardware in surface mesh generation is an area that is largely unexplored, particularly when it comes to the features introduced by NVIDIA's Volta and Turing microarchitectures [1]. This is likely due to the incredibly specific set of operations supported by these features, and their recent introduction into the consumer GPU market. These features have been made easily accessible, not just in datacenters and laboratories but also common household gaming computers and personal devices. This greatly increases the potential benefit brought by the intelligent integration of such features; and it exemplifies why it is important to take steps to fully understand their applications.

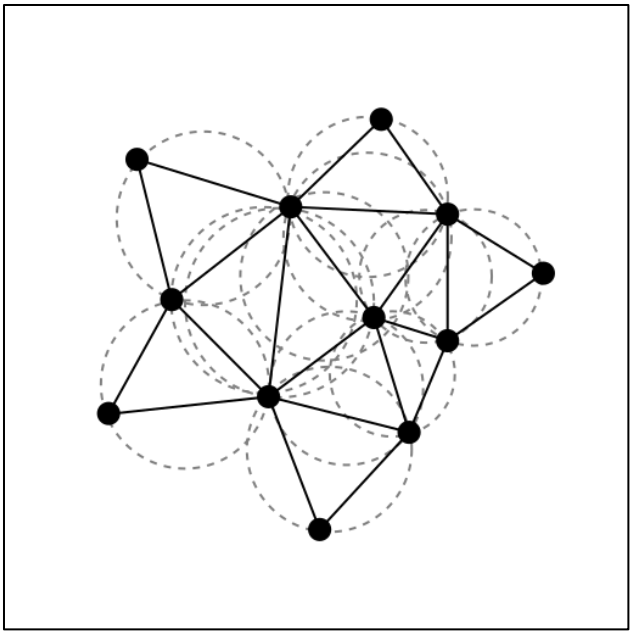


Fig. 1. An illustration of a mesh that satisfies the Delaunay condition.

Tensor Cores excel at extremely fast matrix multiply and accumulate operations (referred to as a general matrix multiply, or GEMM), and are capable of executing both of these operations across an entire matrix in a single clock cycle, in a variety of precisions [7]. These qualities could theoretically be applied to the complex and numerous geometry calculations required by modern mesh generation algorithms. This project investigated some of these applications in hopes of proving their potential efficacy.

III. Algorithm Design Process

To gain a better understanding of what parts of mesh generation stand to benefit from Tensor Core acceleration, a well-known triangulation algorithm was implemented [2] (nicknamed "DeWall")[1]. A cursory code inspection reveals one calculation in particular that seemed likely to be a significant performance bottleneck. This calculation, which will be referred to as the "circumcircle test", evaluates whether or not a newly introduced vertex (referred to as the "subject vertex") resides in the circumcircle formed by the vertices of an existing triangle. Outlined in (1), this function requires the evaluation of the determinant of a 3x3 matrix and consists of 20 addition, 24 multiplication, and one comparison operation. In order for a mesh to satisfy the Delaunay condition, this function must evaluate to *false* for every triangle on the mesh paired with every vertex on the mesh that is not one of the vertices

[1] A second algorithm was also implemented [4], though this algorithm was far too dependent on a highly dynamic map data structure and was not pursued due to consistent problems and low performance.

of the triangle. Consequently, this function can be expected to run many times for every vertex in the input data set.

$$\begin{vmatrix} a_x - d_x & a_y - d_y & (a_x^2 - d_x^2) + (a_y^2 - d_y^2) \\ b_x - d_x & b_y - d_y & (b_x^2 - d_x^2) + (b_y^2 - d_y^2) \\ c_x - d_x & c_y - d_y & (c_x^2 - d_x^2) + (c_y^2 - d_y^2) \end{vmatrix} < 0 \quad (1)$$

To test this hypothesis, a single-threaded version of the DeWall algorithm was profiled. The algorithm was implemented in a serial manner to simplify debugging and to more clearly highlight the least performant sections. Unsurprisingly, the circumcircle test represented 67.99% of the total algorithm runtime[2]. It was for this reason that the test was selected as the project's primary target for optimization.

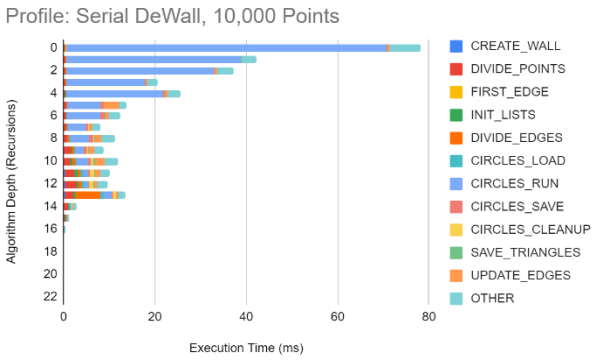


Fig. 2. Runtime profile of a single-threaded implementation of the DeWall algorithm. The "CIRCLES_RUN" section represents the circumcircle test of (1). Also see footnote 2.

$$\alpha AB + \beta C = D \quad (2)$$

[8]

The most obvious approach to optimizing this calculation would be to attempt to evaluate as much of the matrix of (1) as possible using Tensor Core GEMM executions. Unfortunately, this is extremely inefficient. Tensor Cores specialize in accelerating the operation given in (2) [8], where A, B, C, and D are matrices and α and β are scalar constants (all supplied at runtime). The product of A and B is a true matrix multiplication (and not elementwise). This means that while it might be possible to simplify the values of the 3x3 matrix (and subsequently find the determinant) over a few Tensor Core cycles, there is no easy way to perform many circumcircle tests simultaneously on the same GEMM matrix using this approach. This issue is compounded by the fact that Tensor Cores do not activate for GEMM operations with matrix dimensions of less than 64 [8]. This project proposes an alternative approach that reduces the work required to perform each circumcircle test such that Tensor Cores becomes a more viable option for acceleration, in particular for tests with a large number of subject vertices.

$$a = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \quad (3.1)$$

$$b_x = -\begin{vmatrix} x_1^2 + y_1^2 & y_1 & 1 \\ x_2^2 + y_2^2 & y_2 & 1 \\ x_3^2 + y_3^2 & y_3 & 1 \end{vmatrix} \quad (3.2)$$

$$b_y = \begin{vmatrix} x_1^2 + y_1^2 & x_1 & 1 \\ x_2^2 + y_2^2 & x_2 & 1 \\ x_3^2 + y_3^2 & x_3 & 1 \end{vmatrix} \quad (3.3)$$

$$c = -\begin{vmatrix} x_1^2 + y_1^2 & x_1 & y_1 \\ x_2^2 + y_2^2 & x_2 & y_2 \\ x_3^2 + y_3^2 & x_3 & y_3 \end{vmatrix} \quad (3.4)$$

$$x_0 = -\frac{b_x}{2a} \quad (3.5)$$

$$y_0 = -\frac{b_y}{2a} \quad (3.6)$$

$$r_0 = \frac{\sqrt{b_x^2 + b_y^2 - 4ac}}{2|a|} \quad (3.7)$$

[9]

Using the equations in (3), the position (x_0 and y_0) and radius (r_0) of the circumcircle formed by the vertices of a triangle can be obtained. These calculations require a total of 31 addition, 61 multiplication, one square root, and one absolute value operation when simplified. Once the position and radius of a circumcircle are calculated, the circumcircle test can be performed on a subject vertex using a simple Euclidean distance formula. This dramatically reduces the number of operations required to test each subject vertex while adding a fixed overhead for the initial calculation. Using the circumcircle position and radius to directly perform the circumcircle test is an approach that seems to be rarely mentioned in other works, though using this data to support other algorithm functions is definitely not new. In the case of DeWall, this

[2] Note that this metric isn't entirely accurate. Halfway through the project, the entire code base was refactored to remove a serious (unrelated) inefficiency. As part of this refactor, spatial partitions (a common DeWall optimization aspect) were also removed, which would inflate this number. The actual value is estimated to be between 40% and 60%. There was not enough allotted time to add spatial partitions again solely to collect a more accurate version of this metric.

information serves an important optimization function. DeWall employs a grid-based spatial partitioning system to avoid running the circumcircle test on vertices that are not relatively close to the active edge. In this case, the circumcircle data is used to determine which partition cells need to be queried. This removes the need to evaluate the majority of the input vertices when validating a new triangle.

$$(x - x_0)^2 + (y - y_0)^2 < r_0^2 \quad (4)$$

Running (3) before evaluating a large set of subject vertices against one circumcircle effectively reduces the work required per vertex to (4) (plus the overhead calculation). This assumes that the radius is also squared, so the square root operation can be avoided on the left-hand side of the equation. This reduces the operation count per vertex to five addition, two multiplication, and one comparison operation. This is far more practical to execute using a GEMM, however it can be further reduced using the technique outlined in (5).

$$(x - x_0)^2 + (y - y_0)^2 < r_0^2 \quad (5.1)$$

$$x^2 - 2xx_0 + x_0^2 + y^2 - 2yy_0 + y_0^2 < r_0^2 \quad (5.2)$$

$$(-2x_0)x + (-2y_0)y + x^2 + y^2 < r_0^2 - x_0^2 - y_0^2 \quad (5.3)$$

$$c_1 x + c_2 y + p < c_3 \quad (5.4)$$

where

$$c_1 = -2x_0 \quad (5.5)$$

$$c_2 = -2y_0 \quad (5.6)$$

$$c_3 = r_0^2 - x_0^2 - y_0^2 \quad (5.7)$$

$$p = x^2 + y^2 \quad (5.8)$$

This technique was discovered as part of this project in an effort to maximize circumcircle test compatibility with Tensor Core GEMM operations. The constants c_1, c_2, and c_3 can be determined as part of the initial calculations with (3) and are valid for any subject vertex tested against the circumcircle. p is a useful constant that assists with any Euclidean distance calculation involving the vertex (x, y) and can be pre-calculated for all vertices at the beginning of the mesh generation algorithm. This leaves the per-test operation count at two addition and two multiplication operations, again ignoring the initial overhead of (3), (5.5), (5.6), and (5.7) for each unique circumcircle.

By substituting B from (2) with the identity matrix, every cell of A and C can be made to function as an independent operation, with the catch being that the scalars α and β are the same for all cells. Conveniently, this aligns almost perfectly with (5.4), where α is replaced by c_1 and β is replaced by c_2. This substitution works because c1 and c2 remain constant per circumcircle. This means that every cell in a single GEMM execution could potentially complete the majority of the calculations for one subject vertex. With a second GEMM execution (and α and β being set to one), all but the comparison operation can be completed. With this approach, the bandwidth of two GEMM executions is M x N subject vertices tested against one circumcircle (where M and N are the dimensions of the GEMM matrices A and C), minus the comparison operation for each subject vertex. In this case, the comparison would have to be completed in the main thread or in a separate kernel. This approach is relatively efficient and makes Tensor Cores a viable acceleration option for a circumcircle test with a large number of subject vertices.

It should be noted that this simplification method likely has practical benefits for standard mesh generation implementations as well as Tensor Core implementations. While the approach incurs additional overhead per circumcircle, the extra initial instructions should be cancelled out by reduced number of total instructions as the subject vertex count increases. In order to investigate this, a simple test was conducted. The outcome of this test is presented in the results section.

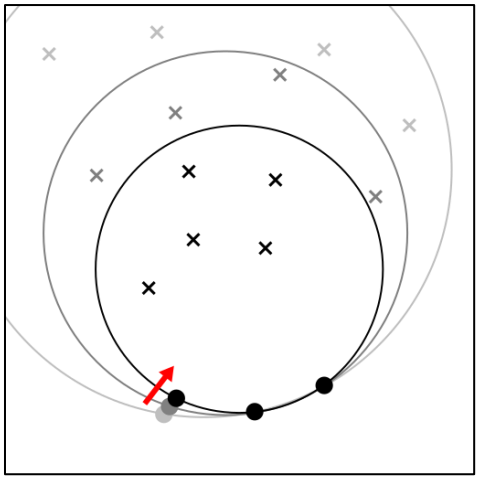


Fig. 3. Illustration of how small changes in the position of a vertex can drastically impact the radius of the circle that circumscribes the parent triangle, and consequently the set of vertices contained within the circumcircle.

The circumcircle position and radius calculation (simplified or otherwise) has another convenient benefit for Tensor Core applications. The DeWall algorithm implementation revealed that high-precision floating point values are required in some instances where the mesh vertices are incredibly dense. Unfortunately, this is due to

the nature of the problem itself; this is illustrated in Fig. 3. The radius of a circle that circumscribes a trio of vertices can be incredibly sensitive to the position of those vertices. Most available Tensor Cores employ 16-bit floating point (half) precision instead of 64-bit (double) precision, which is not ideal in this situation. GPUs built with NVIDIA's Ampere microarchitecture (or newer) do actually have double precision Tensor Cores, though such hardware was not available at the time of this investigation. Luckily, separating the circumcircle position calculation allows for the values to be evaluated in double precision on the main thread. The Euclidean distance formula is also far less sensitive to precision errors than (1) due to its simplicity. So even if (5.4) is conducted in half precision, the process of generating double precision dependency constants and pre-evaluating the circumcircle location data significantly reduces the equation's sensitivity to precision-related errors. If deemed necessary, a precision error check could be added to the post-GEMM comparison kernel to further identify and eliminate inconsistencies resulting from the remaining operations.

IV. Numerical Results and Analysis

All tests performed in this section were conducted on an MSI GE66 Raider laptop with an Intel i7-10750H CPU, Windows 10 Home operating system, 32 GB RAM, and an NVIDIA RTX 2070 Super GPU.

A simple test was constructed to evaluate algorithm performance across different circumcircle test scenarios. As the circumcircle test was the primary optimization focus of this project, the test was incredibly useful for comparing algorithms to each other in a sterile environment. This test effectively removed any problems that could result from failures to fully optimize the specific implementation of the DeWall algorithm[3]. The test selects 100 random triangles, then pairs those triangles with a batch of random subject vertices and asks the algorithm to determine whether each vertex resides in the circumcircle formed by the vertices of the triangle. The total execution time is measured, and the output values are verified using the standard determinant circumcircle test equation from (1). For GPU-based algorithms, memory allocation, transfer, and deallocation instructions are ignored and only the true execution time is recorded (after all threads are fully synchronized). The test is repeated for subject vertex batch sizes per triangle ranging from one to ten million.

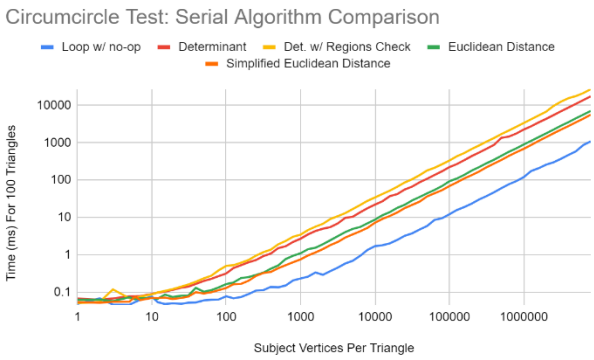


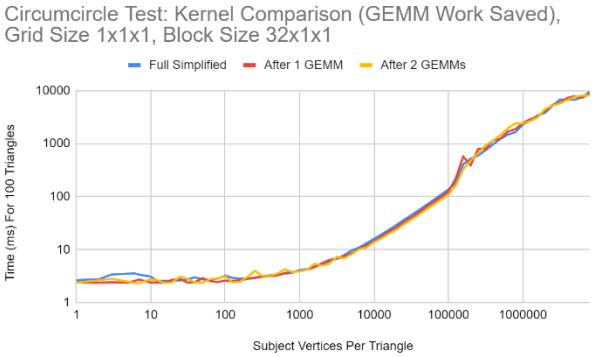
Fig. 4. Circumcircle test results comparison for several single-threaded algorithms.

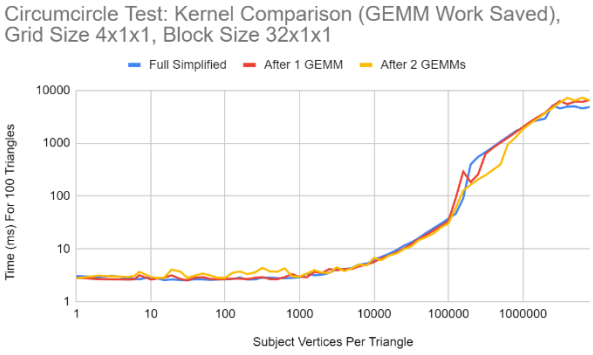
To identify the most effective approach to use as a foundation for GPU-based algorithms, several different serial algorithms were compared using the aforementioned testing function (see Fig. 4). A simple loop over the entire subject vertex test batch with a no-op instruction was recorded as a frame of reference. All measured algorithms achieved roughly the same performance for batches of less than five subject vertices, however after this margin performance clearly separated and a very obvious hierarchy appears. The least performant algorithm (the determinant evaluation with a "regions" check) is a variation of the pure determinant evaluation of (1) that attempts to use lines connecting the triangle vertices to preemptively eliminate possible test outcomes for each vertex. The second least performant algorithm is, predictably, the pure determinant evaluation of (1). This algorithm contains far more instructions than the distance-based algorithms, and the test results demonstrate the instructions' weight. The pure distance and simplified distance algorithms both achieve very high speeds, outpacing the determinant algorithms by almost a factor of 5x. Consequently, the simplified Euclidean distance approach introduced in (5.4) is used as the primary algorithm for both serial and parallel applications for all other parts of this project.

The most important aspect of this investigation is the question of how much processing power can be indirectly gained by performing work on Tensor Cores. A test was conducted to show the potential benefit of offloading parts of the circumcircle test calculation to Tensor Cores. In other words, these tests did not invoke Tensor Core GEMMs and instead demonstrated the kernel work required to handle the output of said GEMM executions. This information is critical, because if the offloaded

[3] While the speed might be useful, optimizing DeWall in its entirety was not within the scope of this project.

calculations amount to an insignificant part of the overall mesh generation runtime, there is no point in assuming the overhead of launching the GEMM. This test was conducted using the aforementioned isolated circumcircle test conditions and the simplified algorithm with precalculated constants introduced in (5.4). For this test, three kernels were compared across a range of kernel grid and block dimensions. The dimensions were selected based on the number of streaming multiprocessors (SM) and warps per SM available in the RTX 2070 Super. The first kernel acted as the control and simply performed the entire circumcircle test. The second kernel performed only the operations that would remain if a single GEMM execution completed the first addition and associated multiplication steps in (5.4). The third kernel performed only the comparison operation, which is all the work that would remain if two GEMM executions completed both additions and all multiplication steps of (5.4). The results of this test can be viewed in Fig. 5.





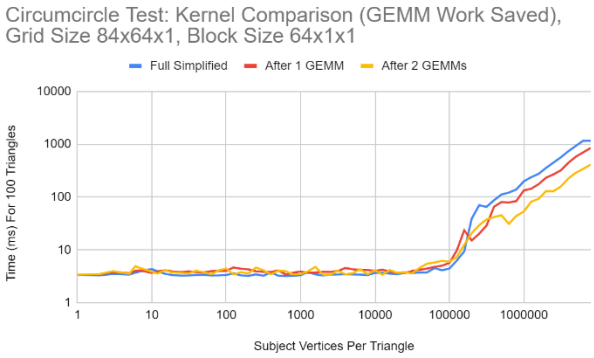


Fig. 5. (Three plots) Circumcircle test results comparison for three different kernels across a range of grid and block dimensions. Differences between the performance of the kernels demonstrates the CUDA kernel compute time that could potentially be saved by running calculations on Tensor Core GEMMs instead.

The first GEMM step of this approach saves two multiplication and one addition operation per element. The second GEMM step saves another addition operation. Theoretically, one would expect this extremely small kernel instruction count per element to favor smaller launch dimensions, however surprisingly the opposite is true. The smallest kernel (two GEMM executions) is the fastest by almost half an order of magnitude in the most parallelized test (5,376 blocks with 64 threads each). In the least parallelized test (one block with 32 threads) the kernel performance was almost identical for all batch sizes. Unfortunately, in the case where the smallest kernel was the most beneficial, the kernel performance only separated after about 600,000 subject vertices per triangle. This roughly coincides with the point at which each thread begins to test more than one element. Practically, this means that (for the given test system) Tensor Core acceleration will not be able to give a meaningful performance boost over parallel CUDA kernel evaluation because kernels are still required to do the final comparison operation of (5.4). The results show that the only exception to this is instances where kernels are parallelized to a very high degree and each kernel is responsible for at least two elements. It's also worth noting that even if the combination of Tensor Cores and kernel-based parallel comparison operations is slower than the pure-kernel approach, the fact that there is a case where the reduced kernels actually complete faster shows that there is potential for work to be effectively offloaded to Tensor Cores. In these specific circumstances, employing Tensor Core processing would free up some remaining GPU power for other tasks and slightly increase the productivity of the system as a whole.

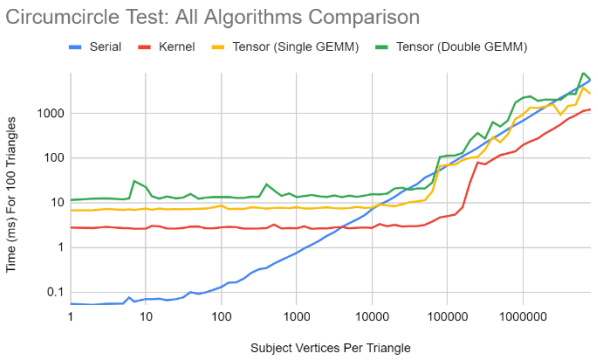


Fig. 6. Circumcircle test results comparison for single and double GEMM Tensor Core algorithms, as well as the best serial and parallel algorithms.

To contextualize the performance of the full Tensor Core algorithms, circumcircle test results were compared to the results of the best serial and parallel algorithms (Fig. 6). Both the GEMM post-processing kernel and the pure kernel algorithms shared grid and block dimensions of 84x64x1 and 64x1x1, respectively. The results show that both Tensor Core algorithms struggle to maintain a runtime advantage over the standard serial algorithm and are easily outpaced by the pure kernel algorithm. Despite the much slower runtime of the two-step GEMM algorithm, it can be inferred that more work is being transferred to the Tensor Cores and more processing power is available in standard CUDA cores as a result. This is somewhat confirmed by the results of the previous test. It should be noted that these results could also be system dependent. So, while the overall runtime is worse, this still has the potential to be viewed as a more economical approach due to the freed GPU resources that are less domain specific.

A very obvious downside to the Tensor Core algorithms is the extreme initial launch overhead. The minimum GEMM matrix dimension must be fully populated, otherwise Tensor Cores are not activated and/or matrix space is wasted. By looking at Fig. 6, it is clear that this overhead eliminates all Tensor Core algorithm viability for subject vertex arrays smaller than 10,000 elements. In the case of DeWall, the optimal spatial partitioning grid cell vertex population was observed to be somewhere between two and ten vertices. This means that each circumcircle test in a DeWall implementation with a spatial partitioning grid would likely only evaluate several dozen vertices, depending on the number of grid cells examined. This completely eliminates all Tensor Core viability for the fastest DeWall implementations.

Despite this, the project was still able to test these algorithms in real mesh generation applications. In order to give Tensor Cores the best possible chance at providing a genuine performance boost, spatial partitions were removed from a later DeWall implementation. This gave the Tensor Core approach two distinct advantages. Firstly, it increased the batch size significantly for every circumcircle test. Instead of being limited to only the vertices enclosed in the partition grids immediately surrounding the active triangle, the algorithm had to test the entire active section of the mesh. The size of this section varies based on the recursion depth, however it is approximated by the equation $n * 0.5^x$, where x is the current recursion depth and n is the total number of vertices in the input data set. The second advantage is that the list of vertices contained in the section remained constant for a large number of circumcircle tests. This helps Tensor Core approaches because the same three GEMM input matrices can be reused many times, with only GEMM invocation scalars α and β being changed. Though this implementation isn't realistic (and is far slower than could be achieved with spatial partitions, even single-threaded), it allows for a demonstration of the potential strengths of Tensor Cores. Without these artificial advantages, Tensor Cores would obviously be far slower due to the launch overhead observed in Fig. 6.

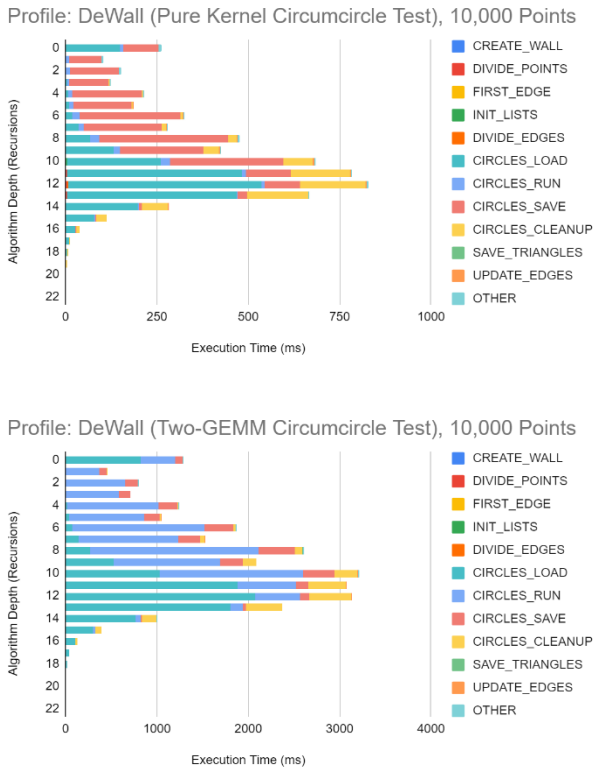


Fig. 7. (Two plots) Runtime profiles of a single-threaded implementation of the DeWall algorithm with CUDA core and Tensor Core circumcircle test functions. The "CIRCLES_LOAD", "CIRCLES_SAVE" and "CIRCLES_CLEANUP" sections all contain signficiant GPU memory

management operations. In the Tensor Core profile, the "CIRCLES_RUN" section also has some GPU memory allocation calls.

Unfortunately, despite these advantages, both kernel and Tensor Core approaches were much slower than serial functions when substituted in the DeWall implementation. This was a problem for which a solution could not be adequately developed within the allotted project time, and the cause of the problem is quite obvious (Fig. 7). The clumsy and inefficient use of GPU memory transfers almost certainly prevented the GPU-based circumcircle algorithms from providing any real contribution the overall algorithm runtime, and the attempts were actually severely detrimental to the algorithm's performance. A GPU-based implementation of the full DeWall algorithm was tested, though this had many issues and ran several orders of magnitude slower than the CPU implementation. This is likely due to the high complexity of the algorithm and much slower GPU clock speeds per-thread when compared to the CPU. DeWall is a recursive algorithm, and a substantial time investment would be required to develop an effective parallelization strategy to eliminate the GPU communication overhead for kernel and Tensor Core circumcircle test approaches.

V. Conclusions

To accelerate two-dimensional Delaunay-compliant mesh generation, this project introduced a method for simplifying the triangle circumcircle test and performing many tests in parallel via Tensor Core GEMMs. This approach has the drawback of limited precision (for some GPU architectures) and a comparison operation must still be completed outside the GEMMs, meaning a follow-up kernel launch is likely required to efficiently complete the tests. A successful implementation would also have to find a way to retain algorithm functionality while navigating GPU memory transfer restrictions. While the Tensor Core implementations were found to be slower than pure kernel approaches in isolated circumcircle tests, the Tensor Cores meaningfully reduced the work required to complete the tests, which suggests that Tensor Cores can be used as an efficiency option to improve system performance as a whole. Unfortunately, fast DeWall mesh generation algorithms rarely require the evaluation of subject vertex batch sizes that are large enough to offset the launch overhead of the Tensor Core algorithms used in this project, which means that Tensor Cores are likely not the optimal solution to this problem. Overall, the concept of performing mesh generation function in the form of rapid circumcircle tests using Tensor Cores shows promise, both as a higher-speed alternative to highly parallel kernel execution and as a means of offloading work to an otherwise idle hardware feature, though much work is still required to optimize the technique and eliminate barriers, with the former strategy requiring more effort than the latter.

The source code for this project has been made available in the following GitHub repository: https://github.com/osreboot/Research-Mesh-Generation

References

[1] N. Oh, "NVIDIA Announces RTX Technology: Real Time Ray Tracing Acceleration for Volta GPUs and Later," 19 March 2018. [Online]. Available: https://www.anandtech.com/show/12546/nvidia-unveils-rtx-technology-real-time-ray-tracing-acceleration-for-volta-gpus-and-later. [Accessed 24 October 2022].

[2] P. Cignoni, C. Montani and R. Scopigno, "DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed," Computer-Aided Design, vol. 30, no. 5, pp. 333-341, 1998.

[3] C. M. Nguyen and P. J. Rhodes, "TIPP: parallel Delaunay triangulation for large-scale datasets," in SSDBM '18: Proceedings of the 30th International Conference on Scientific and Statistical Database Management, Bozen-Bolzano, 2018.

[4] G. E. Blelloch, Y. Gu, J. Shun and Y. Sun, "Parallelism in Randomized Incremental Algorithms," in SPAA '16: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, Pacific Grove, 2016.

[5] T. Zahida, K. Bouhadja, O. Azouaoui and N. Ghoualmi-Zine, "Enhanced unstructured points cloud subdivision applied for parallel Delaunay triangulation," Cluster Computing, 2022.

[6] L. Caraffa, P. Memari, M. Yirci and M. Brédif, "Tile & Merge: Distributed Delaunay Triangulations for Cloud Computing," in IEEE International Conference on Big Data, Los Angeles, 2019.

[7] NVIDIA Corporation, "NVIDIA Tensor Cores," NVIDIA Corporation, [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/. [Accessed 24 October 2022].

[8] NVIDIA Corporation, "Matrix Multiplication Background User's Guide," 12 December 2022. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[9] E. W. Weisstein, "Circumcircle," [Online]. Available: https://mathworld.wolfram.com/Circumcircle.html. [Accessed 24 October 2022].

[10] Y. Zhu, "RTNN: accelerating neighbor search using hardware ray tracing," in PPoPP '22: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul Republic of Korea, 2022.