18-645 Final Report

Christopher Stange
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
cjstange@andrew.cmu.edu

Bharathi Sridhar
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
bsridha2@andrew.cmu.edu

Wei-Che Huang
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
weichehu@andrew.cmu.edu

I. Introduction

For our project we propose implementing Delaunay triangulation and Voronoi diagram calculation. These algorithms have multiple applications in image processing, computer graphics, and computational geometry. The algorithms begin with a set of points in a 2D plane. Delaunay triangulation connects these points with edges ensuring that we only get "nice" triangles. From here the Voronoi Diagram can be calculated as it is the dual mesh of the Delaunay triangulation. The most important kernel for Delaunay triangulation is a point in circumcircle check and the most important kernel for Voronoi diagram generation is the calculation of a triangles circumcenter. We implement high performance kernels for these operations and compare their performance with OpenCV. Our targeted architecture is the ECE clusters.

II. Kernel Design and Implementation

A. Point in Circumcircle Check

Kernel Overview: The Point in Circumcircle Check kernel determines whether a given point lies inside the circumcircle of a triangle defined by three vertices. The inputs to this kernel are the coordinates of the three triangle vertices, (A_x, A_y), (B_x, B_y), and (C_x, C_y), and the coordinates of the point to be checked, (D_x, D_y). The output is a boolean value indicating if the point is inside the circumcircle. The kernel performs a series of arithmetic operations, including subtractions, multiplications, and additions, to compute a determinant that dictates the result.

Original Kernel Design: The Point in Circumcircle Check kernel is designed to determine if a point $D = (D_x, D_y)$ lies within the circumcircle of a triangle defined by vertices $A = (A_x, A_y)$, $B = (B_x, B_y)$, and $C = (C_x, C_y)$. The kernel computes the determinant of a 3×3 matrix formed using the coordinates and squared distances of these points:

$$\begin{aligned} a &= A_x - D_x, \quad b = A_y - D_y, \quad d = B_x - D_x \\ e &= B_y - D_y, \quad g = C_x - D_x, \quad h = C_y - D_y \end{aligned} \tag{1}$$

We calculate the squared distances:

$$c = a^2 + b^2, \quad f = d^2 + e^2, \quad i = g^2 + h^2 \tag{2}$$

The determinant is computed as:

$$\text{out} = a(ei - fh) + b(fg - di) + c(dh - eg) \tag{3}$$

In order to implement this we use the SIMD add, subtract, and multiply units. There is one add/subtract unit with a latency 3 cycles which means we need 3 independent instructions minimum. There are two multiply units with a latency of 3 cycles which means we need 6 independent instructions minimum. There are two fma units with a latency of 5 cycles which means we need 10 independent instructions minimum. The kernel has enough independent instructions to prevent a significant number of stalls using the add, subtract, and multiply units. Using fma results in stalling and resource under utilization.

Final Kernel Design: We decided to use floats because the extra precision was not needed. The main challenge with this kernel was register limitation in between the multiply, subtract and add step towards the end.

We also decided to refactor our kernel into 3 sub-kernels for implementation ease and to experiment with intermediate loads and stores between the sub-kernels. However, this implementation is essentially the same as one fused kernel because we decided to `inline` all the sub-kernels which effectively combined them into one. Additionally, we decided to keep intermediate results in registers since the trade-off in intermediate loads and stores was not worth the performance penalty.

1) `kernel_sub`: This kernel calculates the differences

$$\begin{aligned} a &= A_x - D_x, \quad b = A_y - D_y, \quad d = B_x - D_x \\ e &= B_y - D_y, \quad g = C_x - D_x, \quad h = C_y - D_y \end{aligned} \tag{4}$$

in parallel using 6 independent vector subtractions. The results are stored across 12 SIMD registers, each holding 8 float values (output: 96 floats total).

2) `kernel_square_add`: Using the results from `kernel_sub`, this kernel computes the squared distances:

$$c = a^2 + b^2, \quad f = d^2 + e^2, \quad i = g^2 + h^2 \tag{5}$$

It performs 6 squaring (multiplication) operations and 3 pairwise additions, resulting in 3 sets of values distributed across 6 SIMD registers (output: 48 floats total).

3) `kernel_det2`: This kernel calculates the determinant:

$$\text{out} = a(ei - fh) + b(fg - di) + c(dh - eg) \tag{6}$$

It executes 9 multiplications and 6 subtractions, computing cross-products and combining terms to form the determinant's components. The output is stored in 3 sets of values across 6 SIMD registers (output: 48 floats total).

The main kernel calls the smaller kernels mentioned above. It iterates through the input arrays, calling `kernel_sub`, `kernel_square_add`, and `kernel_det2` in sequence. As mentioned earlier, we refactored our kernel into three sub-kernels for implementation ease and to explore using intermediate loads and stores to optimize register use/reuse, but in the end, we inlined them, effectively merging them into a single fused kernel.

Memory Layout: There are 8 inputs (A_x, A_y, B_x, B_y, C_x, C_y, D_x, D_y) and 1 output ($det3\_out$). This means that there would not be enough ways for each of them to occupy. Therefore, a packing routine is required for inputs to make sure consecutive accesses are near each other.

Assume that 1 way is used to hold kernel outputs, the remaining 7 should be used to hold inputs in this fashion: $A_{x0:x15}, A_{y0:y15}, B_{x0:x15}, B_{y0:y15}, C_{x0:x15}, C_{y0:y15}, D_x, D_y$. For example, when the kernel's first load (A_x) occurs, $A_{x0:x7}$ would be loaded into the SIMD register, therefore these points should be packed sequentially, same applies to the rest of the inputs. This means that for each kernel iteration, the inputs would be in a struct of size $6 \times 16 + 2$ floats or 392 bytes. With 7 ways available for inputs, we can pack 73 such input structs in L1 cache and 585 in L2 cache.

B. Circumcenter Calculation

Kernel Overview: The Circumcircle Center Point kernel computes the center coordinates, (U_x, U_y), of the circumcircle of a triangle defined by three vertices, (A_x, A_y), (B_x, B_y), and (C_x, C_y). The inputs are the coordinates of the three triangle vertices, and the outputs are the coordinates of the circumcircle's center.

Original Kernel Design: The Circumcircle Center Point Calculation kernel computes the circumcenter (U_x, U_y) for a triangle with vertices $A = (A_x, A_y)$, $B = (B_x, B_y)$, and $C = (C_x, C_y)$. The determinant D is first calculated as:

$$D = 2\left[A_x(B_y - C_y) + B_x(C_y - A_y) + C_x(A_y - B_y)\right] \quad (7)$$

Using this, we compute:

$$U_x = \frac{1}{D}\big[(A_x^2 + A_y^2)(B_y - C_y) + (B_x^2 + B_y^2)(C_y - A_y) + (C_x^2 + C_y^2)(A_y - B_y)\big] \quad (8)$$

$$U_y = \frac{1}{D}\big[(A_x^2 + A_y^2)(C_x - B_x) + (B_x^2 + B_y^2)(A_x - C_x) + (C_x^2 + C_y^2)(B_x - A_x)\big] \quad (9)$$

The independent operation is the computation of the circumcircle's center point. In order to implement this we use the SIMD add, subtract, multiply, and divide units.

Final Kernel Design: Our original kernel ran into problems with the limited number of available registers which prevented us from hiding the latency of the division. In order to address this our updated kernel splits the computation into two sections with a small intermediate buffer between them. The first section computes:

$$\text{part}U_x = (A_x^2 + A_y^2)(B_y - C_y) + (B_x^2 + B_y^2)(C_y - A_y) + (C_x^2 + C_y^2)(A_y - B_y) \quad (10)$$

$$\text{part}U_y = (A_x^2 + A_y^2)(C_x - B_x) + (B_x^2 + B_y^2)(A_x - C_x) + (C_x^2 + C_y^2)(B_x - A_x) \quad (11)$$

$$\text{part}D = \left[A_x(B_y - C_y) + B_x(C_y - A_y) + C_x(A_y - B_y)\right] \quad (12)$$

The latency of the division is 11 cycles so we run the above six times. Then we do the following computation to get our final result:

$$D = 2 \times \text{part}D \quad (13)$$

$$U_x = \frac{\text{part}U_x}{D} \quad (14)$$

$$U_y = \frac{\text{part}U_y}{D} \quad (15)$$

The splitting of the computation allows us to hide the latency of all the intermediate operations and proved more performant than the original approach during testing. The original kernel output one set of SIMD U_x and U_y. The updated kernel outputs six sets of SIMD U_x and U_y.

Implementation Details: We decided to use floats because the extra precision was not needed and the latency of the division is significantly less than that for doubles. We found that it is possible to hide the latency of the loads when computing the first half of the kernel, we have enough free registers to load in the inputs to the next iteration while still executing the current iteration. We considered expanding the size of the kernel to more than six but this becomes excessive overhead depending on the problem size and is already close to the register limit. We tested using a reciprocal kernel that computes

$$U_x = \text{part}U_x \times \frac{1}{D} \quad (16)$$

$$U_y = \text{part}U_y \times \frac{1}{D} \quad (17)$$

but found that this degrades performance. Twelve inputs are needed to hide the latency of the division and there are not enough registers available.

Memory Layout: There are 6 inputs (A_x, A_y, B_x, B_y, C_x, C_y), 2 outputs (U_x, U_y), and only 8 ways in the caches. This means it would be extremely inefficient to give each their own way and we must pack them together. In order to get the highest cache utilization we pack all the inputs and outputs needed for a kernel using a nested struct. A pointer to a single struct is then passed to the kernel as input. We can use an array of structs to represent a sequence of inputs for the kernel. There are 8 ways in the L1 and L2: 1 is reserved for the intermediate buffer (packed in a similar fashion) and the other 7 can be used to hold kernel inputs and outputs. We can fit 18 of these structs in the L1 and 149 in the L2. Using a unified struct gets better utilization than splitting into input and output structs or other smaller partitions. Using an entire way for the intermediate buffer is suboptimal from a cache utilization perspective and and splitting the kernels entirely is left to future work.

C. Why are these designs high performance?

These designs are high performance because they attempts to use all the available resources of the processor we are working with. We have identified the independent operations and used them to ensure that the SIMD units are running as close to maximum throughput as possible without having any stall cycles. We have designed a memory layout that will maximize cache utilization.

III. Parallelization

We decided to use the OpenMP parallelization scheme for its ease of use and lower overhead compared to MPI. For Delaunay triangulation, the parallelization is done on the packing and kernel loops. Through experiments (see performance plots), we found that two threads provide the best performance for all problem sizes tested. However, we found that for some other machines (Xeon 4208), larger problem sizes scale better with 4 threads. We decided to leave the option for 4 or more threads in the code (commented out). Additionally, when triangle counts are low at the start of the algorithm, the program only uses one thread until the number of triangles reach a certain threshold (512 triangles).

For Voronoi Diagram calculations, parallelization is done on the packing, kernel, and unpacking loops. Through benchmarking we found that the amount of time spent in the circumcenter kernel is minimal compared to the total time it takes to do the rest of Voronoi diagram generation (nanoseconds versus milliseconds). Therefore our attempts to use OpenMP resulted in a minimal change in runtime (Amdahls Law). The majority of the time for Voronoi diagram generation is spent rearranging C++ data structures. We leave the investigation of more efficient and performant data structures to future work.

IV. Performance Plots

A. Baseline

We implemented an OpenCV-based baseline for Delaunay triangulation and Voronoi diagram generation, leveraging OpenCV's Subdiv2D class. We wrote a pure C++ implementation of Delaunay triangulation and Voronoi diagram generation using only the C++ Standard Library (STL).

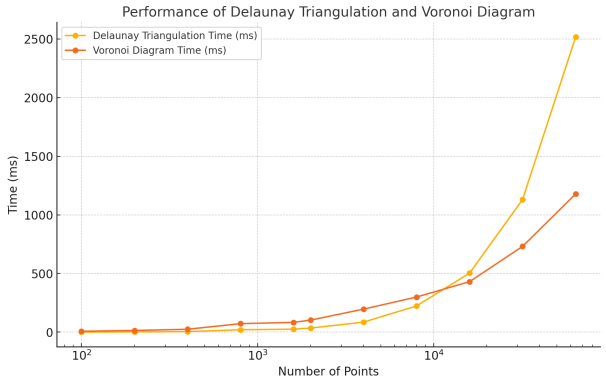

Fig. 1. OpenCV Baseline Performance for Delaunay Triangulation and Voronoi Diagram Generation

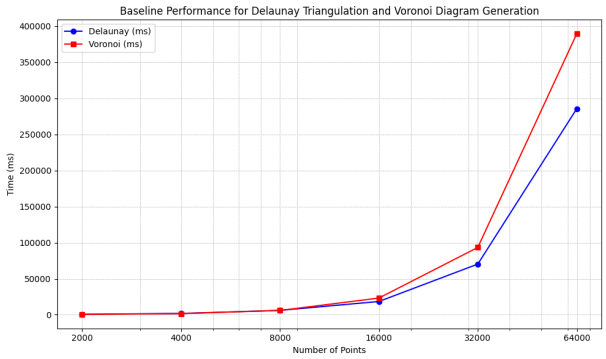

Fig. 2. Baseline Performance for Delaunay Triangulation and Voronoi Diagram Generation

B. Point in Circumcircle Check

1) Theoretical Peak: For this kernel, SIMD multiply has the highest throughput of 16 (8x single precision floats $*$ 2x functional units). Therefore, we use 16 FLOPS/clk as an upper bound on this kernel. The bottleneck for this kernel is the number of available registers, which somewhat limits the number of parallel operations that can be performed.

2) Kernel Performance: We measured the performance of this kernel using input size from 128 to 65536 SIMD elements (1024 to 524288 floats in respective). Note that the performance drop-off after the input size of 1024 SIMD elements likely indicate running out of L2 cache space.

The performance plots for the Det3 kernel show that it somewhat lags behind the compiler-vectorized baseline for smaller input sizes. This is likely due to the overhead introduced by explicit SIMD instruction handling and memory management, which outweighs the benefits of manual optimization at smaller scales. In contrast, modern compilers effectively vectorize scalar code, allowing the baseline to perform near peak throughput for a smaller number of points.

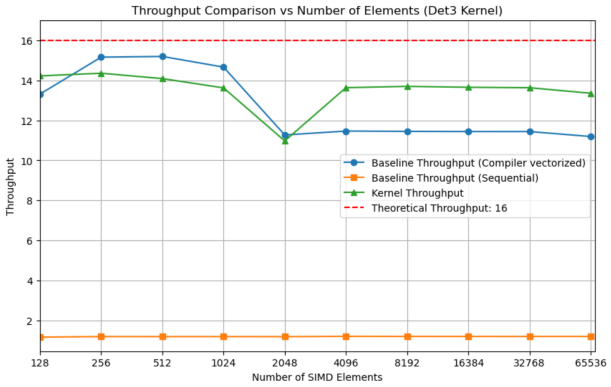

Fig. 3. Throughput (flops/clk) comparison between scalar code (orange line), kernel (green line), and compiler vectorized scalar code (blue line).

However, as the input size grows, our kernel demonstrates better scalability and performs much better than the baseline.

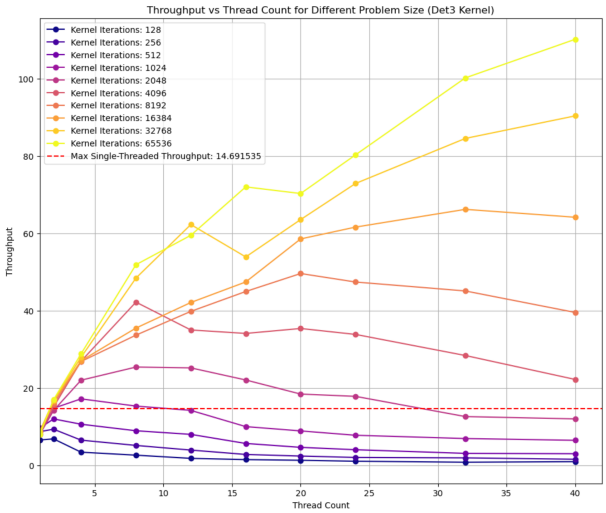

Fig. 4. Multi-threaded throughput (flops/clk) given different problem sizes and thread count, showing that larger prolems scale better with more threads.

The multi-threaded throughput plot shows significant improvements as thread count increases, particularly for larger problem sizes. Inputs with higher iterations (e.g., 32,768 and 65,536) use parallelism effectively, whereas smaller inputs experience diminishing returns likely due to thread management overhead and limited workload distribution.

C. Circumcenter Calculation

1) Theoretical Peak: SIMD multiply is the highest throughput instruction in this kernel and when working with floats this instruction can process 16 elements per clock. Therefore we use 16 FLOPS/clk as an upper bound on performance. Division is by far the slowest operation in the kernel and is the bottleneck.

2) Kernel Performance: In order to measure performance we ran our kernel using different input sizes and counted the number of cycles. The first section of our kernel does

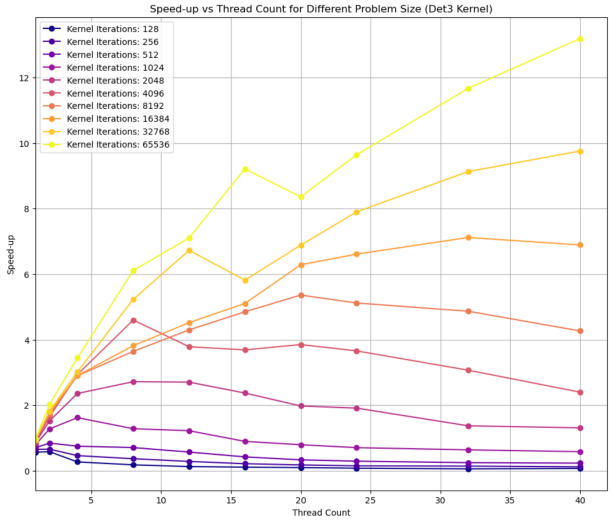

Fig. 5. Multi-threaded throughput (flops/clk) normalized against the single-threaded throughput at given problem sizes showing the same trend as absolute throughput, however it gives a better picture as single-threaded throughput drops as problem size increases.

30 SIMD operations and the second section of our kernel does 16 SIMD operations. Using these numbers we are able to get the performance of our kernel using the standard method from homework. The measure performance of our kernel depends heavily on where the cycles measurement occurs. When the measurement is done inside of the loop the performance sustained across the range of problem sizes. When the measurement is done outside of the loop performance drops significantly when going from one iteration of the kernel to two iterations of the kernel. Our baseline kernel implementation does not suffer this sensitivity and its performance is invariant to the change in timing. Both the baseline kernel and our designed kernel use the same memory layout. All implementations see a drop off in performance when the problem size exceeds L2 capacity. We include a plot highlighting how the kernels performance scales when parallelized used OpenMP. Performance scales sublinearly as we increase the number of threads but a significant decrease in runtime can be achieved if the problem size provides enough work for each thread to do and fits in cache. Note that 48 triangles are processed per kernel iteration.

D. Full Program

We first profiled the performance of our program by sweeping different multi-threaded settings across different problem sizes (Number of points from 2000 to 64000). Here we can see that 2 threads are the best choice for all problems sizes tested (Fig. 9); therefore it is the default setting for our program. Then we use that default setting and compared the performance of our implementation against the C++ STL baseline where we performed favorably (Fig. 11). It is a different story when we tried comparing our results with the OpenCV baseline (Fig. 12), as OpenCV scaled way better than our implementation with increasing number of points despite our

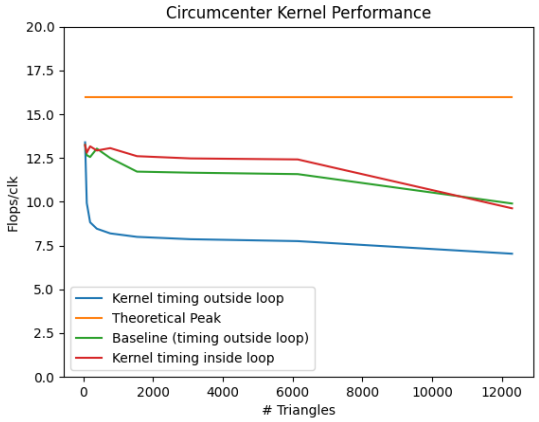

Fig. 6. Throughput (flops/clk) comparison between Circumcenter kernel and baseline.

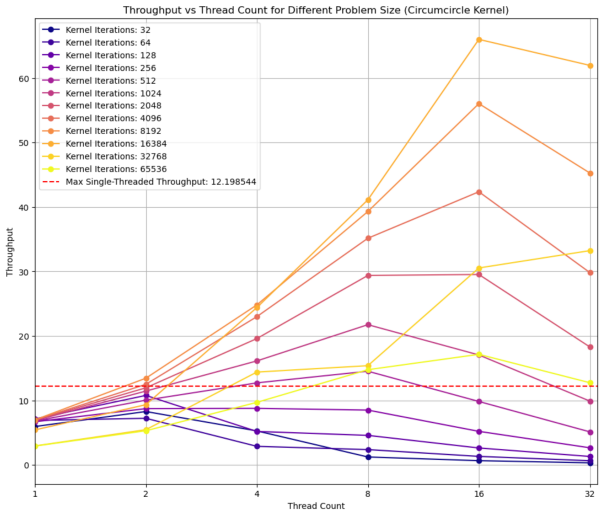

Fig. 7. Throughput (flops/clk) versus thread count for the circumcircle kernel at different problem sizes (shown as kernel iterations). Although this still shows that larger problem sizes scale better with thread count, significant performance dropoff can be seen as number of kernel iterations grow beyond 32768 possibly due to running out of L3 cache space.

implementation being marginally faster with smaller number of points.

V. Future Directions

While our custom kernels for the Point in Circumcircle Check and Circumcenter Calculation demonstrate strong performance, the overall performance of the Delaunay triangulation and Voronoi diagram algorithms is still limited by other factors. If given more time, our team could explore alternative algorithmic approaches to improve overall performance. Specifically, we would investigate other incremental Delaunay triangulation algorithms and updating the triangulation locally. Spatial data structures like quadtrees or k-d trees could also help significantly reduce the computational complexity by allowing faster point location/mapping and more efficient geometric operations. We could also explore alternative trian-

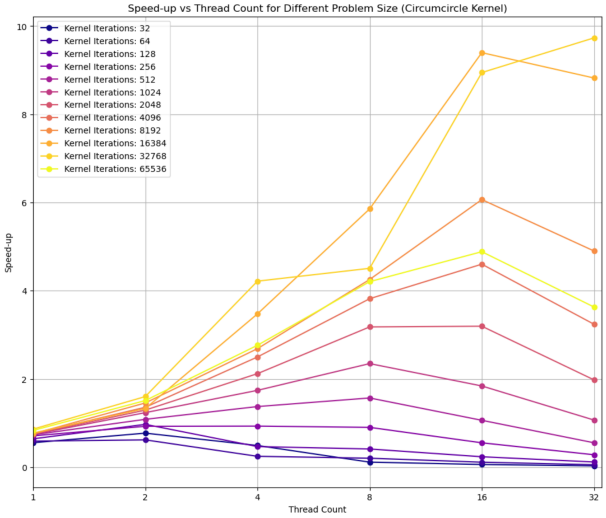

Fig. 8. The relative speed-up shows roughly the same trend, but the scalability at the largest problem sizes is not as bad as shown in the absolute throughput plot.

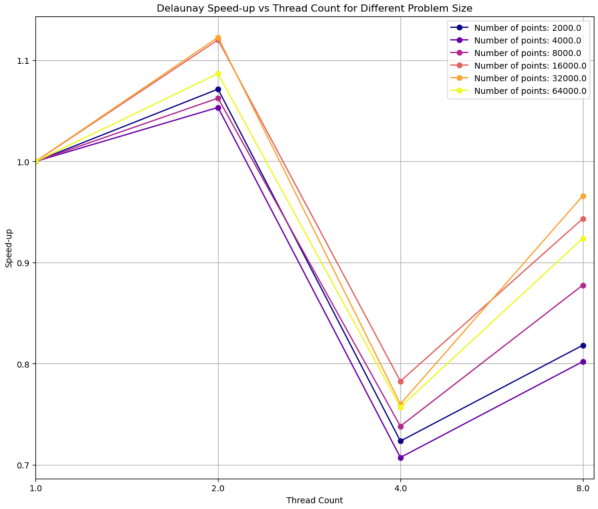

Fig. 9. The Delaunay triangulation portion of the program scales best with 2 threads for all the problem sizes tested on the Xeon 2640 machines.

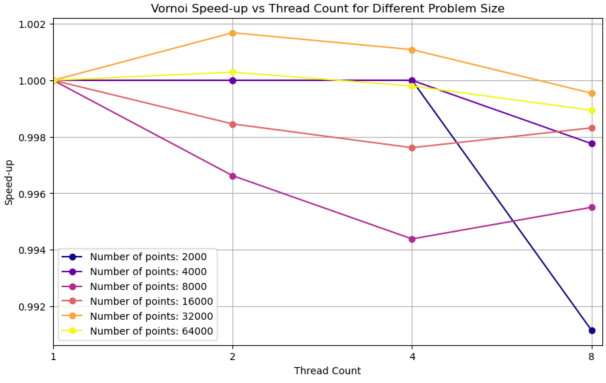

Fig. 10. The parallel region of the Voronoi portion does not contribute much to the total program time; therefore, the performance remains consistent no matter the problem size and thread count.

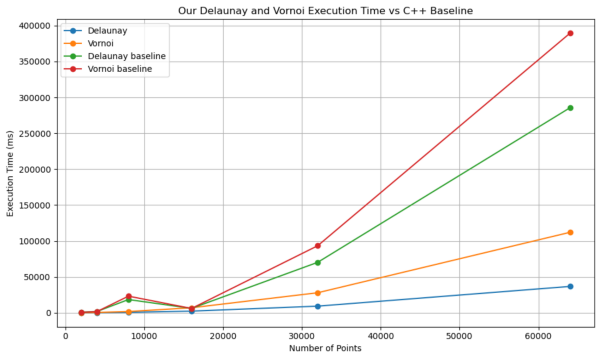

Fig. 11. Our Delaunay and Voronoi implementation outperforming the C++ standard library baseline execution time.

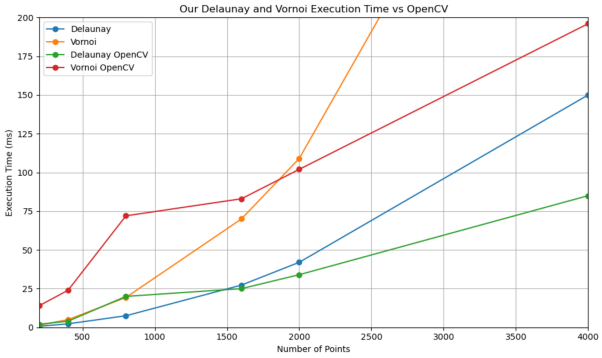

Fig. 12. Our Delaunay and Voronoi implementation versus the OpenCV baseline execution time.

gulation algorithms beyond the super triangle method, such as divide-and-conquer approaches that recursively subdivide the point set, potentially reducing the computational overhead of the current implementation. These algorithmic optimizations could address the performance bottlenecks in the non-kernel portions of the code, potentially offering more substantial speedups in addition to the SIMD kernels implemented.