Mandelbrot Set Generator

ECE 5900

Michael Bowers
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mail: mkbowers@oakland.edu

***Abstract*—The Mandelbrot Set is a set of complex numbers that do not diverge to infinity when run through the recursive function $Z_{n+1} = Z_n^2 + c$ for a specified number of iterations. Threading Building Blocks (TBB) and Streaming SIMD Extensions (SSE) both greatly improved set generation performance compared to the original sequential implementation , with a combination of both strategies yielding the greatest performance gains.**

I Introduction

This project will cover various implementations of a Mandelbrot set generator using different optimization strategies. A sequential implementation serves as a baseline metric to assess the performance of other optimized implementations, including a multi-threaded implementation using Threading Building Blocks (TBB), a vectored approach using Streaming SIMD Extensions (SSE), and an implementation utilizing both TBB and SSE.

The Mandelbrot Set is a set of complex numbers c, where the function $Z_{n+1} = Z_n^2 + c$ performed over an n number of iterations does not diverge to infinity when Z is initialized to 0. [1] The boundary of the Mandelbrot set is comprised of points that vary in the number of iterations required before escaping the set. This boundary region contains detailed fractals structures at increased magnifications. For image generation, pixels are usually colored according to the number of iterations required to cross the specified threshold in the recursive sequence.

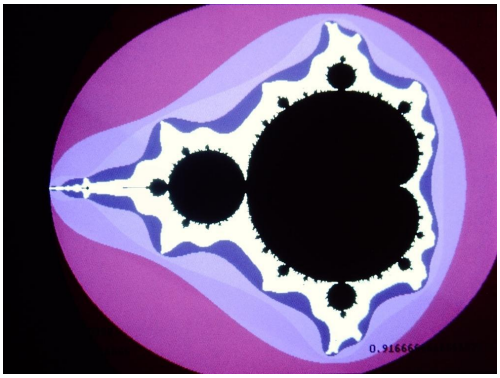


Figure 1: Mandelbrot Set, Keith Shuert (1991)

II Methodology

II.A Image Scaling and Usage

The Mandelbrot Generator requests the following arguments: X_c and Y_c which specify the center coordinates, a radius that specifies our view window, and a maximum number of iterations. The width and height of the image are hard-coded into the main function at a 1:1 aspect ratio. The radius and image dimensions are used to create X and Y scaling factors which are then used to re-scale the X and Y data arrays around the desired center coordinates. A "mode" argument was also used to select between generating all variants, or just the fastest variant (SSE+TBB only).

II.B Sequential Implementation

The basic recursive "escape-time" algorithm can be defined using the following pseudo-code:

```
//xi and yi are starting coordinates
//x and y are initialized to 0

while ((x^2 + y^2) < 4 AND (iterations < maxiterations))
    xtemp = x^2 - y^2 + xi
    y = 2*x*y + yi
    x = xtemp
    iterations++
```

This pseudo-code utilizes two real numbers in place of the original complex-number point representation (one real, one imaginary component) by taking the squared modulus of the complex representation [1]. A point is considered within the set if the squared modulus stays under a threshold of 4 for a maximum specified iteration count. A point escapes the set when the threshold is surpassed (or when a point value reaches beyond a circle with a radius of 2).

The escape-time algorithm forms the basis of the sequential Mandelbrot implementation. For each point, the escape-time algorithm is performed, and the final iteration result associated with a particular point is saved in a 1D raster array. The X and Y data arrays are looped through for-loops to gather and calculate an iteration value for each data point.

II.C ***Parallel (TBB) Implementation***

Threading Building Blocks (TBB) offers *parallel_for* which divides a traditionally serial loop into independent iterations that are executed in parallel. [2] For the parallel Mandelbrot, each for-loop is replaced with a *parallel_for* in a complex-lambda expression format. In this case, each data row is parallelized, and each point within each row is parallelized. Since TBB does not guarantee safety from data race conditions, a function is used to encapsulate each point calculation, and as every function call grabs a distinct iteration index, each task handles a unique point.

II.D ***Vectored (SSE) Implementation***

Streaming SIMD Extensions (SSE) are single-instruction, multiple data instruction set extensions available for most modern processors that allow for the execution of the same operation on multiple data objects, or vectored data.

The Intel Atom N2600 [3] on the Terasic DE2i-150 Board features eight 128-bit registers (XMM) per core, and support for Intel® SSE2, Intel® SSE3, and Intel® SSSE3. The original SSE instructions are also supported. The required header file containing the intrinsics (emmintrin.h) was available via the GCC install on the Terasic board.

This brief guide [4] was used to as an example for proper usage of some SSE intrinsics, the emmintrin.h header, and demonstrates a sample compilation using GCC. This reference [5] contains a complete list and documentation for each SSE intrinsic.

The original escape-time algorithm was used to model the vectored Mandelbrot implementation. As each data row is still processed sequentially, four different X data points are packed into a vector of data type "__m128," (4x32 floats) and a corresponding y-data vector is loaded with the same Y data point associated with the current row. It is imperative that the size of each data row is divisible by four, since the X data pointer is incremented by four for every SSE vector load.

Since it is possible for points on the same vector to result in different final iteration counts, each iteration calculation must finish (the largest count must finish) before a new vector can be loaded and processed. Any iteration count that finishes early will still be processed as a vector component, but will remain unchanged. After the four points are completed, the vector is unpacked and stored on a 1D raster array.

II.E ***Parallel + Vectored (TBB+SSE) Implementation***

TBB was also implemented in concert with SSE. *parallel_for* in compact lambda expression format was used to parallelize each row, and the iteration calculations for each point (points operated on in groups of four) was handled sequentially by an SSE function.

II.F ***Coloring Algorithm***

A simple coloring algorithm for the Mandelbrot set consists of saturating points to one color when the maximum number of iterations was reached, and saturating all other points to a different color. This will highlight the boundaries of the set. Intricate fractal patterns can be better revealed by using a coloring algorithm that assigns a color to a particular iteration count.

A .ppm file was generated from the 1D raster result array, which contains the iteration results for each point. The generation of .ppm files was modeled after a python implementation [6] which uses 24-bit words (split into 3 bytes for RGB information) to color each point. The coloring algorithm utilizes arithmetic, bit manipulation and shifting of the original iteration count to assign values for red, green and blue.

III Experimental Setup

Development was done primarily on a desktop PC installed with Kubuntu and Kate, with final debugging and result gathering done on the Terasic DE2i-150 Board. C++ files were compiled using the GNU Compiler Collection (GCC, using g++ as a compiler driver). A Makefile was used when compiling the Mandelbrot file.

All four Mandelbrot variants were calculated and generated in series, and each had timestamps to record separate run-times. A standard image viewer was used to view the Mandelbrot results. For run-time comparisons, the standard Mandelbrot view (with mode = 0, $X_c = 0.0$, $Y_c = 0.0$, R = 2.0, and MaxIter = 256) was used, and a boundary region was used as well (with mode = 0, $X_c = -0.75$, $Y_c = 0.25$, R = 0.06, and MaxIter = 512).

It was expected that TBB would provide a boost in performance (lower run-time), as each point is calculated independently. SSE should improve run times by up to a factor of four since four points would be simultaneously processed. When combining TBB and SSE, it was expected that the performance increase would be a combination of the performance gains of each individual strategy, since each thread should be able to utilize the XMM registers, and thus be able to perform SSE instructions in parallel.

IV RESULTS

Each optimization strategy at all image sizes tested demonstrated greater performances than their counterpart sequential runs.

IV.A Mandelbrot Run Results at (0, 0, 2, 256)

The run-time results (in ms) for a generations at (0,0,2,256) is shown below:

	Sequential	TBB	SSE	TBB+SSE
1024x1024	1033.54	425.61	185.53	70.82
800x800	620.44	257.63	109.76	42.52
600x600	356.09	150.37	62.35	24.36
480x480	226.92	99.46	41.55	16.37
320x320	102.17	49.47	18.19	7.14

A graphical representation for generation run times at (0, 0, 2, 256) is shown below:

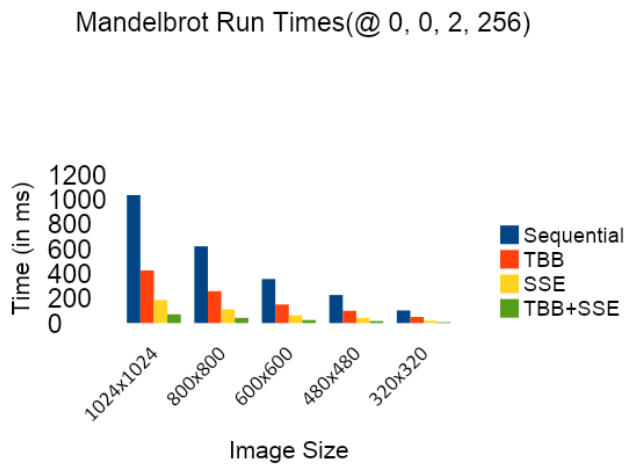


Figure 2: Run times for "Home View"

The relative performance gains for generations at (0, 0, 2, 256) are shown below:

	Sequential	TBB	SSE	TBB+SSE
1024x1024	-	2.428	5.571	14.593
800x800	-	2.408	5.653	14.592
600x600	-	2.368	5.711	14.618
480x480	-	2.281	5.461	13.862
320x320	-	2.065	5.620	14.310

IV.B Mandelbrot Run Results at (-0.75, 0.25, 0.06, 512)

Mandelbrot generations in a zoomed-in boundary region were also tested, and were performed with higher maximum iteration values. Run-time results (in ms) are shown below:

	Sequential	TBB	SSE	TBB+SSE
1024x1024	3791.68	1563.40	733.79	257.23
800x800	2313.76	950.49	423.88	157.37
600x600	1299.89	513.80	246.98	89.87
480x480	833.19	324.14	154.26	57.36
320x320	373.01	148.78	69.14	27.07

A graphical representation for generation run times at (-0.75, 0.25, 0.06, 512) is shown below:

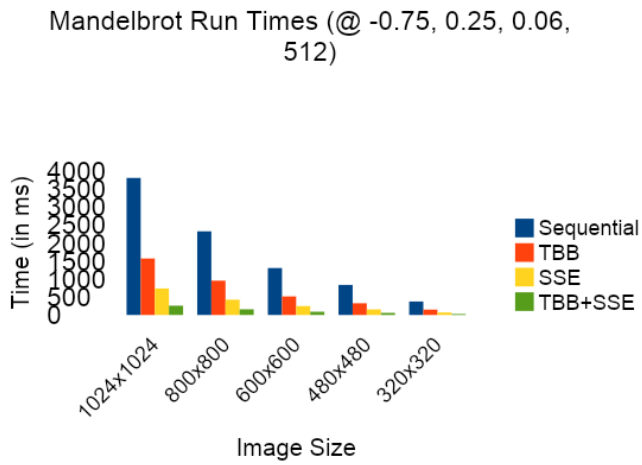


Figure 3: Run times for "Seahorse" View

The relative performance gains for generations at (-0.75, 0.25, 0.06, 512) are shown below:

	Sequential	TBB	SSE	TBB+SSE
1024x1024	-	2.425	5.167	14.741
800x800	-	2.434	5.459	14.703
600x600	-	2.530	5.263	14.464
480x480	-	2.570	5.401	14.526
320x320	-	2.507	5.395	13.789

IV.C ***Interpretation***

TBB consistently improved run-times in all cases. For the (0, 0, 2, 256) runs, the effectiveness of TBB decreased slightly as the image size decreased, but this is expected, as the share of time spent from thread-switching increases. This trend was not seen with the (-0.75, 0.25, 0.06, 512) run, perhaps since the calculation time for each point is longer due to the larger specified maximum iteration value.

For all runs, SSE only improved run-times by factors greater than 5, Since each vector processes four points at a time, but only completes when the point with the largest iteration value completes, the expected increase was under 4 times. The observed results may be due to the SSE code exhibiting better instruction pipe-lining than the original sequential run, or the SSE instructions may simply complete in fewer clock cycles than their single-data counterparts.

The TBB+SSE run performed the best out of all four run variants, which demonstrates that each thread can simultaneously access the SSE architecture located in its respective core. Additionally, for all run scenarios TBB+SSE performed slightly better than the separate performance gains of TBB and SSE multiplied together.

IV.D **Verification**

Verification of results of all four Mandelbrot variants was done primarily by performing checksums on the output .ppm images. Mandelbrot images were generated at region (0.318, 0.5, 0.005, 4096) with a grayscale palette for easier visual checking of the images. Checksums using the md5sum tool showed successful identical checksum results for each Mandelbrot generation.

However, it was discovered by accident that if $X^2 + Y^2$ is changed to X*Y (originally a mistake in the escape-time algorithm that produced a different pattern radiating away from the Mandelbrot set), there are slight visual differences between the SSE generation and non-SSE generation.

IV.E ***Sample Generations***

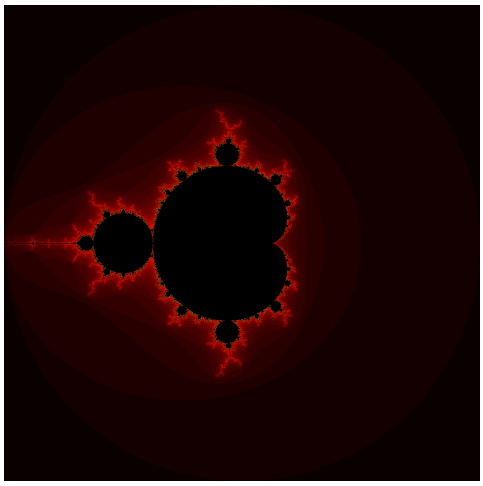


Figure 4: “Home”, Main Mandelbrot
(x = 0, y = 0, r = 2 , I = 512)

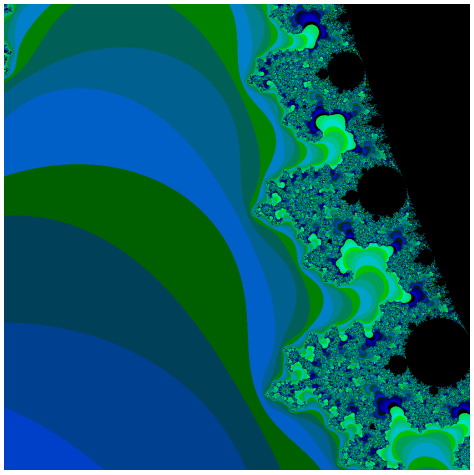


Figure 5: “Seahorse Valley” (x = -0.75, y = 0.25, r = 0.06 , I =256)

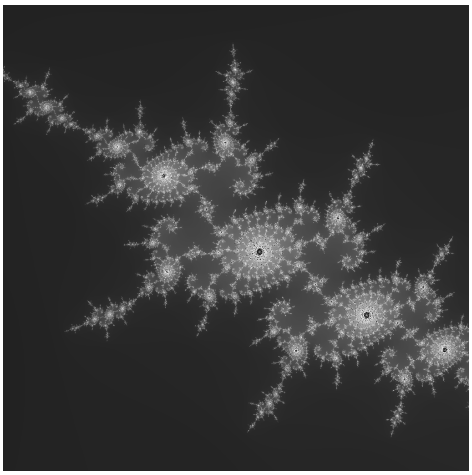


Figure 6: (x = -1.2535, y = -0.0467, r = 0.001, I = 1024)

Conclusions

For applications where the same calculations are performed on a large number of data, both TBB and SIMD instructions appear very suitable, and a combination the two appears especially worthwhile for certain applications. Further work could potentially be done on improving the checking process for points on vectors in the SSE implementation. Newer releases of SSE may contain more useful instructions, including those that allow for easier compares at specific positions on the vector.

Another point potentially worth investigating is why non-SSE and SSE generations differ when using X*Y for threshold checking instead of $X^2 + Y^2$. This was considered an out-of-scope modification to the original Mandelbrot equation and thus was not followed up on. But these results may have implications on the use of SSE. If unexpected results occur from certain calculations with SSE, more caution may be needed to ensure validity of calculated results and if SSE instructions are being used correctly.

Some ease-of-use features and aesthetic changes could potentially be added to enhance user experience. One notable critique is that this Mandelbrot generator is somewhat difficult to navigate. Additionally, the coordinate system is not intuitive; increasing in the negative Y direction moves the Mandelbrot view up. However, this is how a few other Mandelbrot programs set their coordinate system, including the open-source Fraqtive [7]. An aesthetic feature that could be added is a color-smoothing algorithm, which is a feature of many higher end Mandelbrot generators.

References

1 A. Cheritat, Mandelbrot Set, I*nstitut de Mathématiques de* Toulouse. 30 Oct 2016 [Online] Available: https://www.math.univ-toulouse.fr/~cheritat/wiki-draw/index.php/Mandelbrot_set

2 D. Llamocca, Unit 4 – Multi-Core applications, 2021

3 Intel, Intel Atom® Processor N2600, [Online] Available: https://ark.intel.com/content/www/us/en/ark/products/58916/intel-atom-processor-n2600-1m-cache-1-6-ghz.html

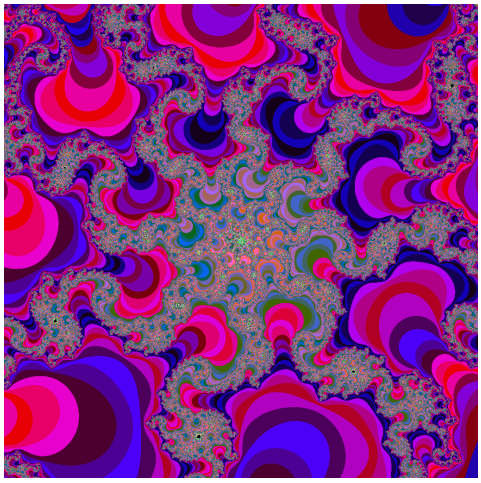
4 C. Woods, SSE Intrinsics, [Online] Available: https://chryswoods.com/vector_c++/emmintrin.html

5 Intel, Intel® Intrinsics Guide, 6 Dec 2021 [Online] Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

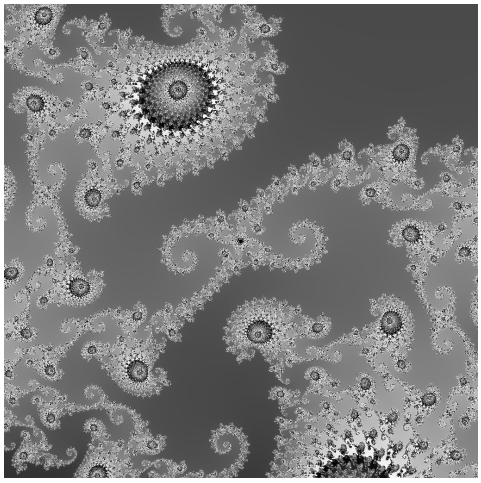
6 Solarian Programmer, PPM image from scratch in Python 3. [Online] Available: https://solarianprogrammer.com/2017/10/25/ppm-image-python-3/

7 Fraqtive, [Online] Available: https://fraqtive.mimec.org/

Appendix

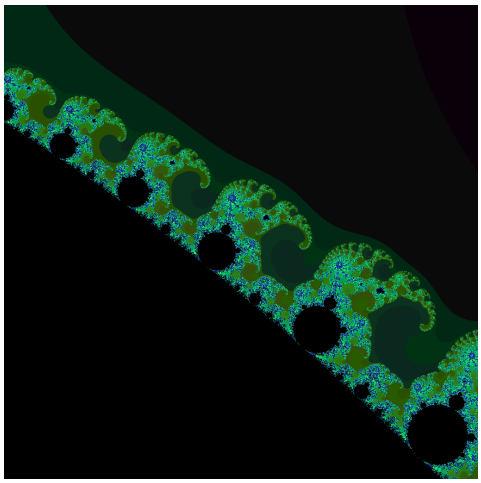


"Octopus" (x = -0.811531, y = 0.201429, r = 0.001, I = 256)

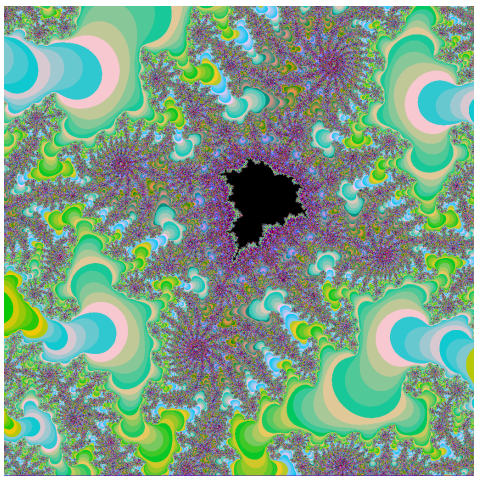


Viral Particles (x = -745428, y = -0.0467, r = 0.001, 1024)

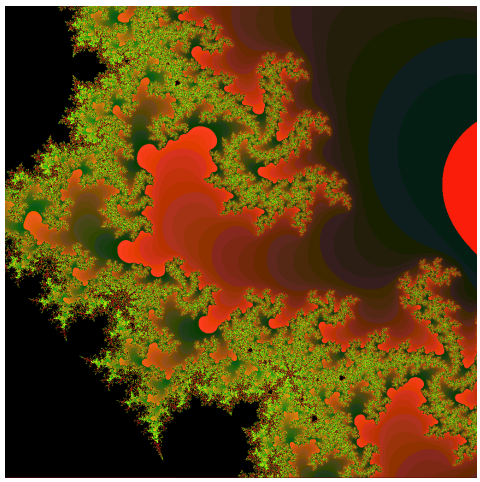
Swirls (x = -0.7568098, y = -0.0668795, r = 0.001, I = 256)



"Elephant Valley " (x= 0.2969, y = 0.020, r = 0.015 , I = 1024)



Hidden Mandelbrot(x = 0.30018, y= 0.4618, 0.0001, I = 2048)



(x = 0.318, y = 0.5, r = 0.005, I = 4096)