Mosaic-Stitching Acceleration for Large Format Image Acquisition

Lucy Camblin Kurt Gu

I. Introduction

Image stitching is often employed where single-shot image captures are insufficiently detailed for a given task, requiring the capture and stitching of multiple smaller image chunks, or *mosaics*, of the same subject. This is often the case for applications such capturing panoramas, digital microscopy, and artwork scanning. Existing methods for performing fast mosaic stitching are predominantly based on feature detection [1] [4] [5] and/or phase correlation [2], from which offset coordinates are derived. However, for applications such as artwork scanning where exceptionally high stitching precision and accuracy are required to preserve data fidelity, existing methods fall short of producing pixel-perfect results. In this work, we propose the use of a straightforward and exhaustive pixel-level stitching scheme, in combination with hardware acceleration, to perform fast and high precision mosaic stitching, which is currently too slow to be practical due to its brute-force computation. We introduce a parallel algorithm to accelerate pixel-level stitching, as well as implementations of the algorithm on two hardware-accelerated platforms: Nvidia Jetson and Xilinx Zynq FPGA, that enable pixel-level stitching to run in real-time, i.e., as fast as the rate of data capture.

II. Background

A. Large Format Image Acquisition

The focus of this work is mosaic stitching for artwork scanning on single board computers (SBCs) such as the Raspberry Pi. Large format artworks, such as paintings, manuscripts, and film prints, often require digitization or ”scanning” for display and archival purposes. This was previously done using purpose-built scanners that are capable of scanning a large area with high resolution. Such scanners have since been gradually discontinued and decommissioned as digital cameras and digital drawings became mainstream in the last decade. However, as analog imaging such as film photography and cinematography enjoys a resurgence in popularity in recent years, technology for digitization remains stagnant and antiquated. The current digitization method of choice for the analog community is camera scanning, where a commercial camera is used in combination with a macro lens to acquire scans of the original. Camera scanning eliminates the need for dedicated scanners and enables a more flexible workflow. However, since the resolution of the scan is limited by the resolution of the camera sensor, its resolution per area decreases as the size of the original increases. This makes camera scanning ill-suited for digitizing large format artwork, and calls for a mosaic and stitching based method that can obtain high resolution over a large area.

In this work, we focus on optimizing mosaic stitching for SBCs because they can be embedded into the digitization workflow and enable a self-contained system, eliminating the need of developing and supporting client-side drivers and software for different OSes and architectures. SBCs are also often performance-limited for the sake of affordabiliy and lower power consumption, which makes them ideal candidates for optimization.

In biology research, digital microscopy faces a similar problem. Due to the rate of magnification, microscopes can only examine a small portion of the sample at any given time. This makes it challenging to acquire an image of the whole sample. In both cases, the final image would need to be reconstructed from many mosaics that contain small and overlapping parts of the subject. We provide a sample of this reconstruction in Figure 1. The main challenge in obtaining good stitching results is misalignment introduced by hardware tolerances and movement of either the camera or the object, which results in a 2D shift in the `xy` plane for the overlap region. A robust and accurate stitching method is then required to offset this alignment error in software.

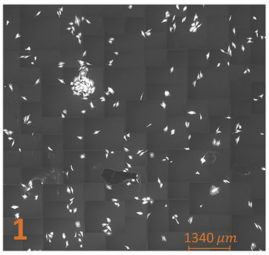

Fig. 1. Example of microscopy stitching.

B. Existing Stitching Methods

Large format image acquisition tasks such as those described above require fast and robust stitching methods to process large amounts of image data efficiently and with acceptable accuracy. Below we give a brief overview of widely-used existing stitching methods.

1) Phase Correlation: Phase correlation works by detecting the relative shift or translation between two images that are being stitched together. Pixel values in both images are

transformed into the frequency domain using the Fast Fourier Transform (FFT). The FFT representations are then cross-multiplied and normalized to obtain a correlation peak. The coordinate of the correlation peak in the resulting cross-multiplied matrix gives the translation (shift in `x` and `y` coordinates) of one iamge compared to the other.

2) Feature Detection & Matching: The first step of feature-based stitching is detecting distinct points (keypoints) in the images. These points are chosen because they are easily identifiable across images. Keypoints can be corners, edges, or blobs that are distinct identifiable. For each keypoint, a descriptor is computed. Descriptors are numerical representations that capture the local structure around the keypoint, enabling recognition and matching of the same feature in different images. Examples of widely-used descriptors include SIFT (Scale-Invariant Feature Transform) [1], SURF (Speeded Up Robust Features) [4], and ORB (Oriented FAST and Rotated BRIEF) [5]. Once features are detected and descriptors given, a coordinate translation can be computed from one image to another such that the corresponding features in both images are lined up.

III. Motivation

While the existing stitching methods described in the previous section are sufficiently robust for most casual use cases, they are still prone to small but visible alignment errors. For stitching mosaics of artwork, and to a lesser degree microscope data, a much higher level of alignment accuracy is required because visual fidelity needs to be maintained. Any visible distortion to the final image introduced by the stitching process would render the image unviable because the data is visual in nature.

Furthermore, empty feature-sparse areas and repetitive patterns are features commonly found in large format images, such as the sky, a body of water, surface of buildings, or empty spaces between microorganisms. Figure 2 sampled from our test dataset demonstrates these two features: the clear sky is feature-sparse, and the sunshade is made up of many repetitive and visually similar thin lines. Neither feature detection nor phase correlation can adequately handle this type of image. Feature detection methods would have trouble detecting enough feature matches or would lower the detection threshold enough to produce false positives. Meanwhile, phase correlation methods cannot robustly handle repetitive patterns because multiple potential phase matches would be detected.

Fig. 2. Feature sparsity and repetitive patterns.

Lastly, widely adopted robust methods such as SIFT [1] and SURF [4] are open sourced but also proprietary, requiring a license for some use cases and thus restricting their applicability.

One observation we make is that the empty areas are not compeletly devoid of information. There are usually film grain or paper texture which are details closer to pixel-scale. Therefore we hypothesize that a matching method looking for pixel-wise value differences instead of features or phases on a larger scale could be more robust to feature-sparse areas and repetitive patterns.

However, such a pixel-wise method requires an exhaustive traversal of all possible cases of overlap and compute all the corresponding residuals (absolute difference in pixel values) for any given pair of mosaics in order to determine the optimal stitching coordinates with minimal residuals. For a pair of 1-megapixel mosaics that share a 20% overlap, that's 200 thousand total possibilities in each of which an average of 100 thousand residual computations are required. Without optimization to reduce the computational overhead, this method would be impractically slow to deploy in real-world tasks that can generate thousands of mosaics [3].

In this work, our aim is to explore methods that can drastically accelerate pixel-level stitching to enable precision mosaic stitching with high visual fidelity.

IV. The Algorithm

A. Naive Approach

To motivate the necessity of parallelization for pixel-level image matching, we first investigate the real-world performance of the naive algorithm, which is simply that, for each possible case of overlap, compute the residual of the overlapping region by taking the average of absolute differences for all pixel pairs within the region. The minimum residual and the corresponding overlap coordinates will be found after all possible cases are computed. Using Python and NumPy, matching a pair of 1-megapixel mosaics takes approximately 20 seconds on a Raspberry Pi 5. In order for the stitching to happen in real-time, the computation needs to execute about 100 times faster, or around 200 milliseconds, to match the readout speed of the camera sensor. In this section we introduce a parallel pixel-level image matching algorithm that provides speedup by enabling more computation to take place simultaneously instead of a naive sequential manner.

B. Parallelization & Minimizing Control Flow

Two insights are key to parallelizing pixel-level image matching. First, sequential control flow such as if statements predicated on value comparisons must be minimized. In the naive approach, a comparison with the current minima is made after each average residual is calculated, and the stitching coordinates may or may not be updated depending on the outcome of the comparison. This could cause slowdown due to branch prediction misses. Maintaining a global memorizer could also lead to data hazards when the algorithm is parallelized, creating more wasted CPU cycles. Second, the size of

each overlap case in the naive approach is not constant. Other than the fact that a varying overlap size is more complicated to express in parallel, each parallel thread could be assigned a different amount of work, bottlenecking the execution time to the time of the thread with most work and potentially lose compute resource to idling. Our proposed parallel algorithm, described below, achieves both objectives.

C. Algorithm Overview

While the naive approach iterates through every possible overlap from all pixels fully overlapping to only one pixel being shared by both images, we make the observation that in practice, the most likely stitching outcome is somewhere in between: since misalignments occur mostly as a result of small hardware tolerances, it is highly unlikely that two overlap regions are misaligned by more than a small percentage. As illustrated in Figure 3, it is almost guaranteed that there exists an inner portion of pixels in all overlaps that will be contained in all optimal stitchings. We call this portion, colored blue in Figure 3, the "core" of the overlaps, and the area outside of the core is the margin of error.

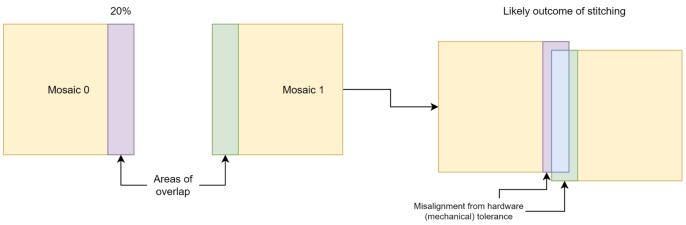

Fig. 3. "Core" of overlapping regions.

If we superimpose the core from one overlap region onto the other, we can observe that for any pixel P in the core, all its potential matches are contained within a square region in the other image, twice as wide as the margin of error, which we call a residual window. As illustrated in Figure 4, each residual window simply describes all the possible movements any pixel P in the core can make and still be contained in the core, and taking the absolute difference between P and its residual windows yields all residuals required for P. In this way we convert the problem from exhaustively trying all stitching alignments with varying overlap dimensions, to computing individual residual windows of a constant size within a core of a constant size, which can be easily parallelized.

In practice, the dimensions of the overlap region, the core and the margin of error should be heuristically defined based on the size of each mosaic and hardware stability to ensure accuracy, i.e., the core should not be too small as to slow down computation, and should not be too large as to miss optimal alignments.

D. Baseline Performance on CPU

We implement this algorithm on the Raspberry Pi 5's quad-core CPU using Numpy as our baseline performance result. The algorithm executed in approximately 0.7 seconds or 700 miliseconds, significantly faster than the naive approach

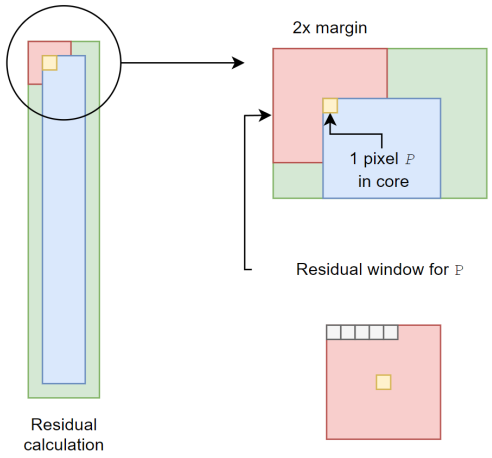

Fig. 4. Computing window of residuals with a core pixel.

and validating our parallelization approach. However, further speedup is still required to meet the 200 ms latency requirement of real-time stitching.

1) Brief Overview of Dataset: Our test dataset is an emulated collection of 847 1-megapixel square mosaics totaling 1.7 GB. We created the dataset by first constructing a single large image consisting of 4 real film scans on a white background. We then partitioned the image into mosaics while introducing random 2D shifts as noise during partition to emulate hardware tolerance errors. We also record the random shifts as ground truths against which the accuracy of each stitching method was evaluated.

V. Implementation on Nvidia Jetson

While Raspberry Pis are ubiquitous in the SBC space, in our preliminary testing its CPU does not provide sufficient hardware parallelism to satisfactorily accelerate our algorithm. Therefore we identify two SBC systems for our workload, an Nvidia Jetson Orin NX with 8 GB VRAM and 1024 CUDA cores, and a Xilinx Pynq-Z2 SoC system with integrated ARM cores as the programming system (PS), and Zynq 7020 FPGA as the programmable logic (PL). In this section and the next, we discuss implementations of our algorithm on these two systems and discuss their respective performances.

A. Parameterizing the Algorithm in CUDA

Parallel programming in CUDA is done through kernels, which are snippets of code that define some computational tasks. Many identical instances of the same kernel is then executed together in a massively parallel manner. These instances are individually initialized as threads, then organized into thread blocks. Blocks are then arranged into grids. Kernels have access to their individual thread and block IDs so they can keep track of their own indices within the thread matrix.

To represent our algorithm in the CUDA programming model, we explored two configurations. In both configurations, the dimension of the overlap region is 1000 by 200 with greyscale pixels.

1) Configuration A: Here we define each pixel-to-pixel residual calculation as one thread, all residual calculations within one residual window as one thread block, and the number of core pixels, which is also the number of residual windows, as the size of the grid.

This configuration is straightforward: every discrete unit of computation is mapped to a discrete unit of execution. One caveat is that, on our Jetson's hardware, the number of threads within a block is limited to 1024. Therefore in this configuration the maximum size of the residual window is 32 by 32, which is suitable for our 1-megapixel mosaics, but could be insufficient for larger mosaics or hardware systems with larger tolerances.

2) Configuration B: Here we define one thread as 64 pixel-to-pixel residual calculations to mitigate the window dimension limitation in configuration A. Each thread block is now defined to have 768 threads to maximize thread occupancy on our specific hardware. The size of the grid is then derived accordingly based on the parameters above and the dimensions of the core and the residual window. This configuration is evaluated with a residual window dimension of 64 by 64 in this paper.

B. Performance Summary

The performances of both configurations are surprisingly close in our testing. Bothing having a one-time setup delay of 260 ms, configuration A computes one stitching pair in 1.9 ms, while configuration B takes 2.2 ms on average, even though its residual window is 4 times the size compared to A. This result suggests that our performance is more bound by memory than compute. Subsequently we tested variations in memory access patterns for configuration B and found no significant improvement. While we hope to try and find a more optimal configuration in the future that better utilizes compute with higher memory reuse, the current performance of our CUDA acceleration is more than acceptable, being more than 300 times faster than the baseline, and is theoretically capable of real-time pixel-level stitching.

VI. Implementation on FPGA

The third implementation explored for parallel pixel-matching was done on the FPGA. Here we consider how the proposed algorithm must be adapted for specific hardware constraints. We aim to maximize parallelism without overconsumption of hardware resources, and therefore both data-flow and memory-management schemes must be carefully considered.

A. Increased Parallelism Approach

Our implementation on FPGA focuses on calculating residuals for the entire 32x32 window associated with each pixel of interest in the assumed overlap region. For simplicity, residual analysis is assumed to take place off-device, and the FPGA implementation solely focuses on producing complete residual results for the potential overlap in a set of images. This system was implemented considering the two regions of interest (1000x200 pixels) in the adjacent images, shown in Figure 5 below.

Fig. 5. Example Images for FPGA Implementation

The blue box in the left image of Figure 5 indicates the overlap core (968x168 pixels) which contains the pixels-of-interest, or POIs, that we're calculating residuals for. Comparison data is obtained from the second image's region of interest, represented by the green box in the right image of Figure 5.

To perform complete pixel matching for one set of adjacent images, 1024 residuals are calculated for each of the 162,624 POIs. This involves simple 8-bit subtraction between the POI's 8-bit intensity value (0-255) and the 1024 intensity values of the residual window data. While computing 1024 residuals in parallel for each POI in a single clock cycle is intuitive, this level of parallelism introduces significant memory management challenges.

Although the computation itself requires minimal hardware resources, increased parallelism drastically raises memory requirements. Larger amounts of data must be accounted for at any given time, and sufficient memory bandwidth is needed to ensure the next POI's computations can begin immediately after the previous one finishes. Pixel data for the comparison window needs to be available in the same clock cycle to perform 1024 parallel residual computations, which requires 1024 simultaneous reads from memory and 1024 simultaneous writes to store the results. Over 2 KB of memory would be required to support the residual computation process for a single POI.

If the process is completely parallelized, the memory requirements increase to support the seamless flow of data through the system, which is done to minimize latency between processing residuals for each POI. The memory requirement to store and access all necessary data for both pixels and residuals when processing even 10,000 POIs would require tens of megabytes, which far exceeds the capacity of on-chip memory of most FPGAs. Since there are 162,624 POIs, it's clear that external memory is necessary to support this level of parallelism.

Unfortunately, relying solely on external memory to support this system introduces new bottlenecks. External memory access is limited by bandwidth that may not be able to support the simultaneous reads and writes required for 1024 parallel computations per POI. Additionally, the latency of external memory access disrupts the smooth progression of

computations, reducing the benefit and practicality of increased parallelism. Therefore, a batched processing approach that maximizes use of on-chip memory, therefore reducing use of external memory, balances parallelism with memory requirements and is a much more viable and feasible system for an FPGA.

B. Batched Processing Approach

In the batched approach, less parallelism is exploited in order to make memory requirements more manageable for the FPGA, but pipelining is utilized to increase throughput. Instead of calculating all 1024 residuals in the 32x32 window for the current POI, the residuals for each POI are calculated one row of the 32x32 window at a time. This data flow allows for the computation of one row of residuals per clock cycle, requiring 32 clock cycles to fully process all 1024 residuals for a single POI. While one row of residuals is being computed, the pipelined system prepares the information for calculating the next row of residuals. Figure 6 shows a four-stage pipeline to describe this system.

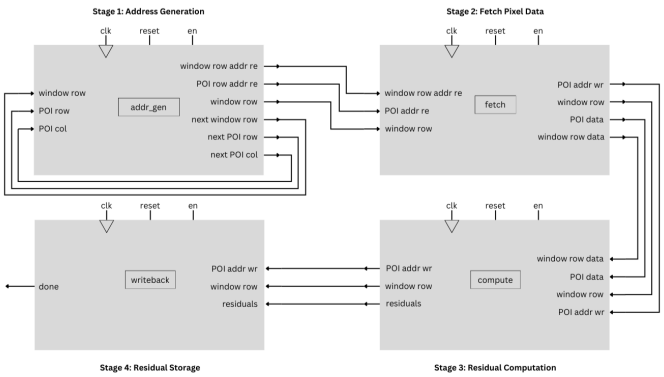

Fig. 6. Four-Stage Pipelined Residual Calculation

Each stage requires a clock signal (clk) and two control signals (reset and en) that are necessary for understanding where the system is in its processing of all POIs in an overlap region. The first stage uses inputs that indicate the position of the current POI and of the current row in its residual window to generate memory addresses to access the associated pixel values. This stage also preemptively determines the positions of the window row and the POI for the next clock cycle. The second stage fetches pixel data for the current POI and window row from on-device memory. This stage also propagates the window row position and POI address to the following stage, because these are used for determining the writeback address in the final stage. The third stage calculates one row of residuals for the current POI, also propagating the window row position and POI address to the next stage. The fourth and final stage writes residuals to memory, using the window row position and POI address to determine the writeback address. When all POIs have been processed and all residuals have been calculated, the writeback stage outputs a done signal. As stated above, each POI requires 32 clock cycles to complete processing, and the pipelined system allows for overlap in processing cycles to improve efficiency.

This pipelined system is also parameterized so that the total number of POIs and the corresponding amount of window data required for each POI can be adjusted. This is useful for scaling the system to available on-device memory. Instead of processing the entirety of the image data and the residual storage in one iteration, the data can be partitioned into smaller, more manageable segments that align with the processing capabilities of realistic FPGA architectures. This batched approach allows on-device memory to be loaded with smaller amounts of data when processing a single segment. Computed residual data is also stored on-device for the current segment. When moving between segments, computed residual data can be streamed to larger external memory while pixel data for the next segment is loaded from external memory to the FPGA. Additional control logic would be required in the pipelined system described above, utilizing signals like en and done to put the system into an idle state while memory is being loaded in and out for each segment. Since external memory is still required in this approach, we expect the idle time between segments to introduce low amounts of latency into the partitioned and pipelined system. However, the data has been partitioned into reasonable batch sizes with the intention of avoiding bandwidth saturation and therefore minimizing latency between segments. By balancing computation, memory, and dataflow, the batched approach provides a practical and efficient method for residual calculations while addressing the challenges of memory availability, bandwidth, and latency. This approach is easily adaptable for different FPGA architectures, making it much more feasible for real hardware.

The optimal batch size that balances available hardware resources with system efficiency must be determined for unique FPGA architectures. The memory management for the partitioned system depends on the segment size of POIs being considered by the system per iteration. Window data required for processing one segment of POIs is defined by a (n x m) segment of the entire comparison data. Window data is accessed in rows, so for the first POI in the segment, pixels at addresses 0 through 31 are accessed first, then pixels at addresses m through m+31 are accessed in the next cycle, and so on for all 32 rows in the POI's residual window. POIs are accessed iteratively, and the starting address for window rows depends on POI position. In a (j x k) segment size of the original overlap core, there are (j * k) total POIs, and they are accessed row by row, then column by column - that is the POI at (0,0) is accessed first, then the POI at (1,0), and this continues until the POI at (j-1, k-1) is accessed, which marks the end of processing for the current segment.

The addresses corresponding to the pixels in the window data for one segment are illustrated in Figure 7, and the addresses corresponding to the POIs in one segment are illustrated in Figure 8.

There are (n * m) total pixels in the segment's corresponding window data, and thus (n * m) 8-bit words need to be allocated

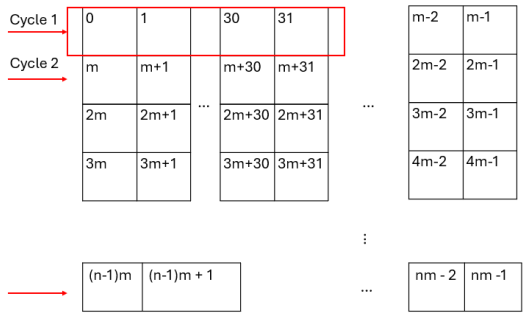

Fig. 7. Window Data Addressing Scheme

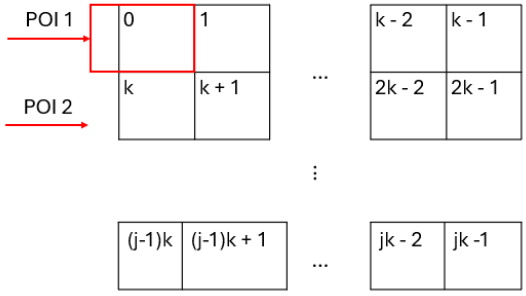

Fig. 8. POI Addressing Scheme

in device memory for one segment's window data. There are (j * k) total POIs in each segment, and thus (j * k) 8-bit words need to be allocated in device memory for one segment's POI data. During residual writeback, separate on-device memory is utilized, where residual data is stored as rows. The writeback address for each residual row combines the current POI address and the current window row index, so the writeback address requires the number of bits needed to describe (j * k) POI addresses plus 5 bits to describe rows 0-31. This is illustrated in Figure 9 for clarity.

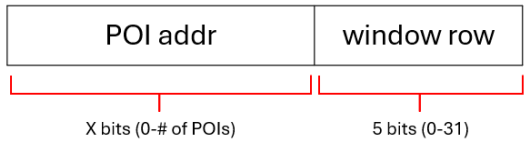

Fig. 9. Residual Writeback Addressing Scheme

Since residuals are stored as rows, the residual data can be stored as 256-bit words. Since there are (j * k) total POIs, each with 32 corresponding residual rows, (32 * j * k) 256-bit words must be allocated in memory to accommodate residual data for all the POIs in the segment.

C. Batch Size Considerations

To further understand optimal batch size and the associated on-device memory required, we considered using segments of sizes 968x168 pixels (the entire overlap core), 32x32 pixels, and 16x16 pixels. These are presented in Figure 10.

Segment Size (pixels)	Window Data Size (pixels)	Required Memory for POIs	Required Memory for Window Data	Required Memory for Residuals	Number of whole segments
968x168	1000x200	163 KB	200 KB	0.1 GB	1
32x32	64x64	1 KB	4 KB	1 MB	158
16x16	64x64	256 B	4 KB	256 KB	635

Fig. 10. Batch Size Comparison for FPGA Implementation

From these results, it's clear that 16x16 is the most feasible segment size to use on real hardware. The FPGA we chose to use for this system is the Xilinx Zynq 7020 (PYNQ-Z2), which contains 630 KB of block RAM [6]. The partitioned pipelined approach described in this paper can be mapped comfortably on this system if 16x16 pixel segments are used, and external memory can be utilized for intermediate data transfer between segments. This specific FPGA also has a built-in Python framework that can be used for off-device memory management, which we aim to implement in future iterations of the pixel-matching algorithm on FPGA. Therefore, this system design is both feasible and optimized for the PYNQ-Z2 board.

To analyze the timing trade-offs when implementing different partition sizes, the processing of one segment of each partition size was simulated in ModelSim using a 1 GHz clock and under the assumption that there is enough memory to contain both image data and residual storage (though this is not applicable on real hardware). These simulations ignored memory setup times and synthesis times and assumed memory was pre-loaded. Following these simulations, the feasible design for residual calculation was implemented on the PYNQ-Z2 board using a 125 MHz clock. Again, only one segment was processed and memory was pre-initialized. These results are summarized in Figure 11 below, with the timing report for the PYNQ-Z2 illustrated in Figure 12.

Segment Size	Execution Location	Clock Speed	Execution Time For Single Segment
968x168	Simulation	1 GHz	0.02 ms
32x32	Simulation	1 GHz	131.087 ns
16x16	Simulation	1 GHz	32.774 ns
16x16	Hardware	125 MHz	135.136 ns

Fig. 11. FPGA Implementation Timing Results

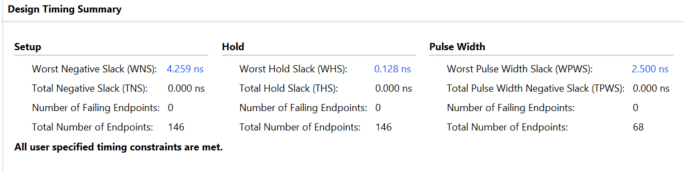

Fig. 12. PYNQ-Z2 Timing Report

D. Performance Summary

The PYNQ-Z2 processed one segment in 135 ns, as shown in Table X, and there are 635 whole segments in the overlap core, so we can project that the system processes the majority of the overlap core in (635 segments) x (135 ns per segment) = 0.08 ms, under the assumption that data transfer to and from external memory between segments takes no additional time (which is an unrealistic assumption). Though no data has been collected for external memory access time between segments, the core processing time is so minimal that we expect high performance to be achieved by the FPGA implementation for complete residual computation of two adjacent images in a dataset. Therefore, the assessment can be made that the FPGA implementation for a parallel pixel-matching algorithm is most constrained by memory availability, memory size, and memory access times (both for on-device and external memory).

VII. Future Work

The work presented thus far represents a significant step towards accelerating image stitching via parallelized pixel-matching. While the current work demonstrates the feasibility and effectiveness of both GPU- and FPGA-based systems, there are several avenues for future research and development to improve performance, scalability, and robustness.

Memory Management for Full Image Processing (FPGA): To extend the current FPGA implementation to process entire images, external memory must be interfaced using robust and optimized management techniques. This would involve optimizing data streaming and storage solutions to handle the large volume of pixel data efficiently.

Caching and Data Management Optimization (FPGA): Introducing caching mechanisms and other data management techniques would reduce the need to re-fetch frequently used data for neighboring pixels of interest that share window data. This optimization would improve processing speed and reduce latency due to memory access.

Parameter Exploration: A detailed parameter search for memory access patterns and algorithm configurations could be conducted to identify optimal settings for the pixel-matching systems, especially as it's used for various image resolutions and dimensions.

Color Matching Extension: Implementing the parallel pixel-matching algorithm on color images would involve handling three times as much data, as each pixel contains RGB values. Though a more complex system design is required, this extension would provide more comprehensive and accurate matching results.

Hardware Scalability: Adjusting the implementation to handle multiple image sets simultaneously would allow more complex stitching to take place, particularly for larger datasets. Creating a flexible and scalable hardware architecture would accommodate various camera and lens combinations with different dimensions and resolutions.

Advancing Imaging Modalities: Implementing the algorithm for RGB or infrared images would expand this system's applicability to a wide array of imaging applications, such as medical imaging or satellite imaging.

VIII. Conclusion

In this work, we presented parallelized pixel-matching for large-scale mosaic stitching workflows on three kinds of hardware: Raspberry Pi 5, Jetson Orin NX, and Xilinx Zynq 7020. We first established a sequential or rather weakly-parallel (as much as its CPU and the programming framework would permit) baseline on the Raspberry Pi. We then created a CUDA implementation on the Jetson with more data parallelism, which achieved significantly higher performance than the baseline. Finally, we implemented a modified version of the parallelized algorithm for the Xilinx FPGA, focusing on complete residual computation, hoping to exploit more hardware parallelism. The FPGA approach also achieved higher performance than the CPU implementation.

Comparative analyses between each implementation revealed the strengths and trade-offs of each approach, with the FPGA and CUDA implementations producing promising results. It should be noted here that since the stitching data originates from an image sensor, there's no additional benefit to be had once the stitching algorithm becomes faster than the sensor's readout speed at full resolution, which is approximately 160 ms / frame in our case. Future efforts will focus on extending the FPGA implementation to process entire images and incorporating other enhancements to accommodate larger datasets containing RGB images, thus broadening the system's applicability across various fields.

Ultimately, the work presented in this paper contributes to the development of a robust, flexible, and high-performance system for near real-time image stitching, paving the way for its integration into a wide range of industrial and research applications. Project progress can be monitored at the following repositories: github.com/therapyKG/Astroscanner, github.com/therapyKG/Astroscanner-Jetson, and github.com/lcambl/FPGA-mosaic-stitching.

References

[1] Lowe, D.G. "Distinctive Image Features from Scale-Invariant Keypoints." International Journal of Computer Vision 60, 91-110 (2004).

[2] A. Alba, J. F. Vigueras-Gomez, E. R. Arce-Santana, R. M. Aguilar-Ponce, "Phase correlation with sub-pixel accuracy: A comparative study in 1D and 2D," Computer Vision and Image Understanding, Volume 137, 2015, Pages 76-87

[3] J. Chalfoun, et al. "MIST: Accurate and Scalable Microscopy Image Stitching Tool with Stage Modeling and Error Minimization". Scientific Reports. 2017;7:4988.

[4] Bay, H., Tuytelaars, T., Van Gool, L. (2006). "SURF: Speeded Up Robust Features." In: Leonardis, A., Bischof, H., Pinz, A. (eds) Computer Vision - ECCV 2006. ECCV 2006. Lecture Notes in Computer Science, vol 3951.

[5] E. Rublee, V. Rabaud, K. Konolige and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," 2011 International Conference on Computer Vision, Barcelona, Spain, 2011, pp. 2564-2571, doi: 10.1109/ICCV.2011.6126544.

[6] Digilent Inc., "PYNQ-Z2 Reference Manual v1.0", May 2018. Available: https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf.