High-Throughput Image Alignment for Connectomics using Frugal Snap Judgments

Tim Kaler
MIT CSAIL
tfk@mit.edu

Brian Wheatman
Johns Hopkins University
wheatman@cs.jhu.edu

Sarah Wooders
UC Berkeley
wooders@berkeley.edu

***Abstract*—The accuracy and computational efficiency of image alignment directly affects the advancement of connectomics, a field which seeks to understand the structure of the brain through electron microscopy.**

We introduce the algorithms Quilter and Stacker that are designed to perform 2D and 3D alignment respectively on petabyte-scale data sets from connectomics. Quilter and Stacker are efficient, scalable, and can run on hardware ranging from a researcher's laptop to a large computing cluster. On a single 18-core cloud machine each algorithm achieves throughputs of more than 1 TB/hr; when combined the algorithms produce an end-to-end alignment pipeline that processes data at a rate of 0.82 TB/hr — an over 10x improvement from previous systems. This efficiency comes from both traditional optimizations and from the use of "Frugal Snap Judgments" to judiciously exploit performance–accuracy trade-offs.

A high-throughput image-alignment pipeline was implemented using the Quilter and Stacker algorithms and its performance was evaluated using three datasets whose size ranged from 550 GB to 38 TB. The full alignment pipeline achieved a throughput of 0.6–0.8 TB/hr and 1.4–1.5 TB/hr on an 18-core and 112-core shared-memory multicore, respectively. On a supercomputing cluster with 200 nodes and 1600 total cores, the pipeline achieved a throughput of 21.4 TB/hr.

I. Introduction

Accurate and computationally efficient image alignment is vital within the field of connectomics [1]–[5], which seeks to study comprehensive maps of connections in the brain through the analysis of high-resolution imagery obtained from electron microscopes. Advances in electron microscopy have enabled the acquisition of image datasets that capture both the small and large features present in neural tissue. The resultant datasets are quite large with a relatively small $1mm^3$ volume producing petabytes of data when imaged at $(3 \times 3 \times 30)$nm resolution. The scale of the acquired data necessitates the development of image processing algorithms that are both scalable and efficient.

The images obtained from the electron microscope must be aligned before they can be used in later stages of a connectomics pipeline. The image alignment algorithms assemble the imagery produced by the microscope and correct errors that were introduced during the sample preparation and imaging processes. The process of imaging a large physical volume at such high resolution does not immediately produce a single representative three-dimensional image. Instead, the volume to be imaged is physically sliced into very thin sections (approximately 30nm thick) which are then each imaged separately. Imaging a single two-dimensional section in its entirety is, unfortunately, not possible since the size of each section vastly exceeds the limited field of view of the microscope. In order to image each section, therefore, the head of the microscope must physically move to scan over the entirety of a section, each time capturing a single image tile. These image tiles must be aligned using 2D and 3D alignment algorithms to correct for the physical movements of the microscope and the deformations introduced during the slicing process [1].

Existing systems for performing image alignment in connectomics distribute fine-grained tasks over large compute clusters that have specialized hardware (GPUs) for accelerating performance-intensive tasks. Although there are many, often lab-specific, alignment systems, these all tend to closely follow the methodology used in the FijiBento library [6] and the HHMI/Janelia Aligner project [7]. Each of these systems distributes work in stages; each stage is composed of one or more tasks for each image tile (approximately 8MB) in the dataset. The intermediate results needed to combine the results of these tasks are communicated either directly over the network or, more commonly, through a distributed filesystem such as Lustre [8] or pNFS [9]. In Section II, we review the algorithms used in connectomics to perform 2D and 3D alignment and discuss the performance challenges that previous systems have encountered.

This paper presents a case study on the design of a multicore-centric image-alignment pipeline for connectomics that is scalable and efficient. Along these ends, our discussion focuses on three principal topics: (a) fast and memory-efficient multicore algorithms for alignment that can scale to future dataset sizes on the horizon in connectomics; (b) pipeline designs that allow efficient horizontal scaling, using coarse-grained tasks, over a computing cluster where each node is a multicore; and, (c) strategies to exploit application-specific performance–accuracy trade-offs to reduce the work required to perform alignment.

Designing an efficient alignment pipeline for multicores

This paper presents two shared-memory multicore algorithms called Quilter and Stacker that perform 2D and 3D alignment, respectively.

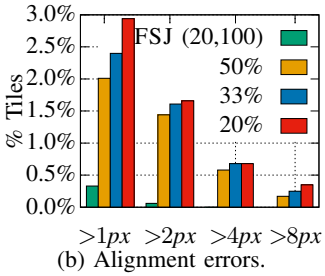
In Section III, we introduce Quilter, which is an efficient multicore algorithm for 2D alignment that avoids costly serialization and communication of intermediate results. Key to the design of Quilter is its efficient task ordering, which allows it to handle intermediate results efficiently while retaining the ability to scale to datasets that cannot be stored entirely in memory on a single multicore.

In Section IV, we introduce the Stacker algorithm, which performs 3D alignment of sections. Stacker enables the alignment of a stack of sections in parallel by computing the pairwise elastic alignment of all adjacent sections concurrently. These pairwise alignments can be combined associatively to obtain a global 3D alignment for the entire volume. Although

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

Algorithm	Runtime	Throughput
100%	180 min	0.18 TB/hr
50%	86 min	0.38 TB/hr
33%	49 min	0.67 TB/hr
25%	30 min	1.1 TB/hr
20%	27 min	1.2 TB/hr
FSJ(20,100)	32 min	1.03 TB/hr

(a) 2D alignment throughput.



(b) Alignment errors.

Fig. 1: Impact of downsampling on throughput and accuracy on the *Common Multicore* platform and the *mouse50* dataset. Figure 1a illustrates the performance of 2D alignment when downsampling images. Figure 1b illustrates the amount of alignment error introduced when downsampling relative to the algorithm running on full-resolution images. FSJ$(20\%, 100\%)$ uses frugal snap judgments with a fast path that downsamples images to 20% resolution and recomputes at full-resolution when detecting a likely error.

this general approach is often employed when aligning sections using affine transformations, it has not been used for more complex elastic transformations that are required to align large datasets in connectomics. The ability to 3D align adjacent sections independently facilitates the design of a horizontally scalable pipeline, as we discuss in Section IV.

Exploiting performance–accuracy trade-offs

A simple way to improve throughput of an image processing pipeline is to downsample the images, but this has the potential to impact the quality, or correctness, of the computed alignment. In Section V we introduce a technique called *frugal snap judgments* to safely exploit performance–accuracy trade-offs in Quilter. Using frugal snap judgments, our system learns to identify when the result of the fast alignment algorithm is likely to match the result of the slower, more-accurate code. The use of frugal snap judgments facilitates a 3–5x performance improvement in Quilter without introducing significant alignment error.

Figure 1 illustrates the trade off between performance and accuracy when using downsampled images in Quilter. On full resolution images, Quilter achieves a throughput of 0.18 TB/hr. When using images downsampled to 20% resolution, however, Quilter achieves a much higher throughput of 1.2 TB/hr. Unfortunately, the use of downsampled images causes some tiles to be misaligned[1], as is shown in Figure 1b. The FSJ$(20, 100)$ algorithm performs alignment using images downsampled to 20% resolution and identifies when a tile's alignment is unreliable so that it can be recomputed using full resolution data. Using frugal snap judgments, Quilter achieves a throughput of 1.03 TB/hr while producing alignments that very closely match those produced with full resolution data.

Performance evaluation of Quilter and Stacker

Section VI presents an end-to-end performance evaluation of the image alignment pipeline we developed using Quilter and Stacker. We investigate the performance of the pipeline on a set of four computing platforms that range from a modestly sized desktop machine to a computing cluster with thousands of cores. The efficient design of Quilter and Stacker combined with the judicious exploitation of performance–accuracy trade-offs using frugal snap judgments allows us to obtain state-of-the-art performance, even while using only commodity multicore hardware. Specifically, we show that the alignment pipeline achieves 0.6–0.8 TB/hr and 1.4–1.5 TB/hr throughput on an 18-core and 112-core shared-memory multicore, respectively. Additionally, we demonstrate that our pipeline can scale horizontally over multiple multicores in a cluster: the alignment pipeline achieves a throughput of 21.4 TB/hr on a 200 node cluster with a total of 1600 cores.

Contributions

A summary of the contributions are as follows:

- A shared-memory multicore algorithm for 2D alignment called Quilter that has low memory requirements and does not perform redundant file I/O or recomputation.
- A horizontally scalable 3D alignment algorithm called Stacker that supports affine and elastic transformations.
- An optimization technique called *frugal snap judgments* which improves the performance of Quilter by 3–5x, with no significant accuracy loss.
- System evaluation of Quilter and Stacker on three datasets, ranging in size from 550GB to 38TB, and on four computing platforms, including a single 18-core workstation and a computing cluster with thousands of cores. The evaluation illustrates end-to-end performance, vertical/horizontal scalability, and strong/weak scaling.

II. Alignment Algorithms used in Connectomics

In this section, we provide necessary background on the general structure of algorithms used to align large datasets in connectomics. This discussion provides a basic understanding of the key operations that are performed during alignment when using FijiBento (and related) systems. Additionally, we will introduce some performance challenges related to memory-limitations and overheads due to data serialization and file/network I/O.

Image data format

Let us briefly describe the format of the images provided by the microscope to make our later, back-of-the-envelope calculations concrete. The microscope provides a set of image tiles, where each image is 2724×3128 pixels. Each pixel in an image represents a $(3 \times 3 \times 30)$nm physical volume. We assume that the set of tiles that compose a 2D section fall within a bounding rectangle whose length and width differ by a constant factor, and we assume that the average tile overlaps with ≈ 8 neighbors. These assumptions are realistic guarantees that are ensured during the image acquisition process.

2D alignment algorithms

The first step of the alignment pipeline constructs a 2D *section* from the set of image tiles obtained by imaging a single, physical slice of tissue. During the imaging process each acquired image tile is associated with an approximate location in the plane, and adjacent tiles are imaged such that there is a small amount of overlap. This overlap allows

[1]Measured relative to the alignment produced on full resolution data.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

Algorithm 1: 2D Image Alignment

```
Result: locations of each tile in a single global space
read in metadata file;
create list P of all pairs of possibly overlapping tiles;
all_local_offsets = ∅;
foreach p ∈ P do
    load_image(p.first);
    load_image(p.second);
    kp_1 = getKeyPoints(p.first);
    kp_2 = getKeyPoints(p.second);
    matched_kp = matchKeyPoints(kp_1, kp_2);
    local_offset = find_best_offset(matched_kp);
    all_local_offsets[p] = local_offset;
end
global_offset = combine_offsets(all_local_offsets)
```

the 2D alignment algorithm to identify common landmarks in adjacent tiles that are then used to precisely determine the tiles' relative positions. After the relative alignments are computed, optimization is performed to find the tile locations that minimize the total energy of a system in which each adjacent tile pair is connected by a spring with rest position determined by the tile pair's relative alignment.

Algorithm 1 illustrates the structure of the 2D alignment step. The input to Algorithm 1 is a metadata file that includes the location of each image tile on disk and the approximate coordinates of each tile in the plane. These approximate coordinates define a poor alignment but are sufficient to determine the pairs of tiles that overlap. The image data for each tile is often compressed to reduce file and network I/O. A common format is JPEG2000 [10], which obtains up to 10x compression ratios on connectomics datasets. Generating keypoints is commonly performed using the SIFT algorithm [11]. Other algorithms such as SURF [12] and ORB [13] are sometimes used in place of SIFT, but their use is not as common. Keypoints in the two images are matched by finding the nearest neighbors in the SIFT feature space and the matches are filtered using the ratio of difference heuristic described in [11]. Next, a random sampling and consensus algorithm (RANSAC) [14] is used to find a coordinate transformation (in the case of 2D a rigid translation) that is consistent with the maximum number of matched keypoints.

Design considerations for 2D alignment

We considered a variety of designs for the 2D alignment algorithm that vary in terms of their memory and communication requirements. Our goal in the design of Quilter, which we describe in Section III, was to build a system that would: (a) read each image tile only once to avoid unnecessary I/O; (b) avoid performing redundant computation; (c) avoid the costs associated with serializing intermediate results to disk; and, (d) have projected memory requirements that enable a single multicore to align a section of human brain.

To illuminate the design challenges addressed by Quilter, we will discuss a few natural alternative designs and explain their shortcomings.

All-I/O and All-Mem

First, let us consider two extreme approaches: All-Mem, which stores all images and intermediate results in memory, and All-I/O, which keeps only a single pair of tiles in memory and writes intermediate results to disk.

An All-Mem algorithm loads all images from disk and then computes the 2D alignment, while keeping all images and intermediate results in memory. All-Mem requires memory proportional to the size of the area being aligned, which precludes its use for large sections. The advantages of All-Mem are that it only reads each image once and it does not need to perform recomputation. Thus, All-Mem satisfies (a)–(c), but fails to satisfy (d).

An All-I/O algorithm maintains a constant memory footprint by performing redundant computation and file I/O. For each pair of overlapping image tiles, All-I/O reads both images from disk, computes the SIFT keypoints, and finds the tile pair's relative alignment. The images and keypoints are then discarded, and the relative alignment is written to disk. All-I/O must read and compute keypoints for each image approximately 4 times — increasing compute and I/O costs by 4x. Thus, All-I/O satisfies (c) and (d), but fails to satisfy (a) and (b).

Inter-Mem and Inter-I/O

Let us now consider two variations, Inter-Mem and Inter-I/O, that avoid recomputation by caching the keypoints computed for each image. The Inter-Mem algorithm reads each image and generates keypoints that are then cached in memory. Then, the relative alignments of all adjacent tiles are computed using the previously computed keypoints. The Inter-I/O algorithm operates similarly, but caches keypoints on disk instead of in memory. For a typical tile of size 8MB, the size of the cached keypoints is between 1.2 and 3.1MB[2]. As such, the Inter-Mem algorithm is generally more memory-efficient than All-Mem. Inter-I/O, however, is generally less efficient than All-I/O for two reasons: (1) images are often read from disk in a compressed format that can be 10x smaller than the uncompressed data; and (2) the Inter-I/O algorithm must read a tile's keypoints an average of 4 times to align it to all of its neighbors. As such, Inter-I/O performs less computation than All-I/O by avoiding recomputation, but it does so at the expense of extra I/O to access cached keypoints.

3D alignment algorithms

After 2D alignment, sections are aligned to form a 3D image that represents the imaged volume. Many processes during the imaging process can distort the 2D sections. The physical cutting of the volume is the likely cause of most distortions, but other processes such as the transport of the very-thin sections via water-bed and non-uniform expansion/compression of the tissue due to environmental conditions may play a role as well. Regardless of the cause, these distortions must be corrected to obtain a representative 3D image for the sample.

The 3D alignment algorithm approximates the function mapping the coordinates of one section into another using a collection of locally-affine transformations. Specifically, a triangle mesh overlays each section and a barycentric coordinate transformation [15] maps the points within each triangle into the coordinate space of adjacent sections.

Typical systems, such as those in FijiBento, for performing 3D alignment operate by dividing the volume into smaller 3D blocks and perform iterative optimization of the triangle mesh.

[2]A data table is provided in Figure A2 that contains the information needed for this back-of-the-envelope calculation

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

Corresponding points between adjacent sections are identified (using a procedure similar to that used in 2D). Then, an iterative procedure adjusts the triangle meshes in the volume to minimize the distance between corresponding points. A disadvantage of this approach is that the entire volume must be aligned more-or-less simultaneously. The addition or removal of a single, new section can require recomputing the alignment for the entire volume. This can be especially problematic when one fails to identify a single, bad section prior to investing the computing resources to optimize a large volume. Furthermore, this approach precludes the interleaving of 2D and 3D alignment in a principled manner — since the computed 3D alignment for a single section depends on all other sections.

III. Quilter Algorithm

This section describes Quilter: a 2D alignment algorithm for stitching very large mosaics in memory. Quilter employs careful task ordering to bound its memory use while avoiding unnecessary file-I/O and recomputation. This enables Quilter to reap the performance advantages of in-memory computing, even for very large mosaics. Quilter can process a 2D cross-section of an entire mouse brain with under 50 GB of memory and process a 2D cross-section of a human brain with less than a 1 TB memory.

Line-sweep ordering of tasks

The essential idea of Quilter is to order tasks based upon a line-sweep through the section being 2D aligned. At any given point in the execution, the image data and intermediate results for tiles that are touched by the line-sweep and those tiles' neighbors are kept in memory. The data retained for a tile is released once the line sweep has advanced beyond all of a tile's potential neighbors.

Figure 2 illustrates an in-progress execution of Quilter using this line-sweep task-ordering. Quilter begins by computing, for each tile, a set of overlapping neighbors and sorts all tiles by the y-coordinate of their bounding box's bottom-left corner. The initial set of tiles processed by Quilter consists of tiles whose bounding box overlaps with a horizontal slab extending along the bottom of the section. For each tile being processed, all of its neighbors that are not already in memory are read from disk. Quilter then computes pairwise alignments between selected tiles and all of their overlapping neighbors. Quilter then progresses by increasing the position of its horizontal slab to select a new set of tiles to process. This new slab contains the neighbors of the previous slab, which are already in memory. Before local alignment is computed on the new slab, the new slab's neighbors are brought into memory. Once the local alignment is finished, the first slab's data can be released, the slab is moved up, and the process is repeated.

After computing the pairwise alignments of all tiles in the section, Quilter solves the optimization problem described in Section II to position tiles in the plane using a loss function determined by the local relative alignments between tile pairs. This optimization problem is formulated and solved in parallel by treating it as a data-graph computation and employing the techniques from [16].

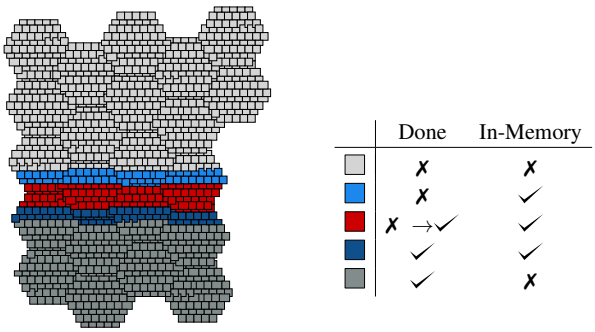


Fig. 2: An illustration of a Quilter execution in-progress. Tiles are shaded red if being processed, blue if its keypoints are in memory, and light/dark gray if it is untouched/done.

Method	Memory	Total I/O	I/O Ops
All-Mem	1000 TB	100 TB	130 million
All-I/O	0.5 TB	400 TB	520 million
Inter-Mem	160 – 400 TB	100 TB	130 million
Inter-I/O	0.5 TB	640 – 1600 TB	780 million
Quilter	0.8 TB	100 TB	130 million

Fig. 3: Memory and I/O Characteristics of 2D alignment algorithms on a $10cm^2$ section imaged at $(3 \times 3 \times 30)$ nm resolution with tile size 2724×3128 pixels.

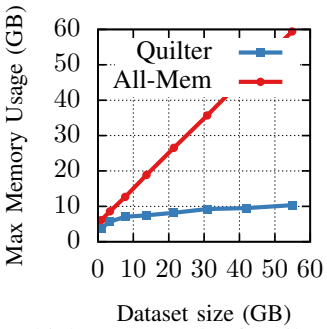


Fig. 4: Memory high-water mark of Quilter compared to All-Mem as the input section size grows.

Memory requirements of Quilter

We shall now analyze the memory and I/O required by Quilter to align a section using the data from Figure A2.

Quilter performs I/O to read compressed image data and write its final result to disk. For a 10cm^2 section, Quilter reads 100TB of compressed image data from disk. Quilter writes a list of tile IDs and 2D offsets to disk, which are negligible (less than 64 bytes per tile). The total memory required by Quilter depends upon the maximum size of its working set. For each tile in the working set, we store the tile's uncompressed image data and all of its keypoints. The maximum size of the working set is approximately 3 rows of tiles. As such, the total memory required to process a 10cm^2 section using Quilter is approximately 800GB. A comparison of Quilter to the methods described in Section II can be found in Figure 3.

The memory requirements of Quilter relative to All-Mem were measured empirically and plotted as a function of dataset size in Figure 4. For this experiment, we ran All-Mem and Quilter on progressively larger subsets of a large 2D section of $40,000$ tiles. These subsets were circular regions centered in the middle of the 2D section, with varied radius. As expected,

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

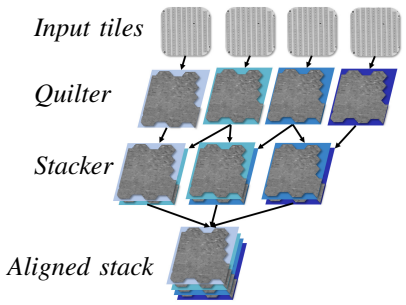


Fig. 5: Illustration of the alignment pipeline design using Quilter and Stacker to perform 2D and 3D alignment, respectively.

the memory requirements of Quilter scale proportionately to the diameter of the region being aligned, and All-Mem scales proportionately to the area.

IV. The Stacker Algorithm

This section describes the Stacker 3D alignment algorithm. Stacker operates on pairs of adjacent sections that have been 2D-aligned using Quilter. Stacker supports affine transformations and non-affine "elastic" transformations, which enable Stacker to compute quality 3D alignments that correct the distortions introduced during sample preparation and imaging.

Our discussion will focus on two aspects of Stacker that are especially relevant to the alignment pipeline described in this paper: (a) its ability to align adjacent sections independently in parallel, and (b) its memory requirements when operating on large sections.

Independent alignment of sections

Stacker is designed to align pairs of adjacent sections independently using composable transformations. Given a pair of sections (S_{i-1}, S_i), Stacker computes a coordinate transformation mapping points in S_i to points in S_{i-1}. This coordinate transform is represented using a hexagonal triangle mesh that overlays section S_i. Each triangle vertex in S_i has a corresponding *transformed vertex* in S_{i-1}, whose position is computed by Stacker during 3D alignment. After 3D aligning S_i to S_{i-1}, a point in S_i can be mapped to a coordinate in S_{i-1} by finding the triangle containing that point and performing a barycentric coordinate transformation [15] to obtain that point's location in the corresponding triangle within S_{i-1}.

Figure 5 illustrates the end-to-end alignment pipeline that uses Quilter and Stacker. Given a stack of sections $S_1, S_2, \ldots, S_k$ that have been 2D aligned by Quilter, Stacker computes the pairwise alignment of $(S_1, S_2), (S_2, S_3), \ldots, (S_{k-1}, S_k)$ independently. To map all sections into the global coordinate space of a single 3D volume, the relative alignments are combined using function composition, which can be accomplished efficiently by applying the barycentric coordinate transformations upon the vertices of the triangle mesh itself.

Memory requirements of Stacker

Stacker is designed to align a pair of sections on a single multicore. As such, a natural concern is that Stacker will be unable to align very-large sections efficiently due to the limited available memory of a single machine. Fortunately, the memory requirements of Stacker are actually quite modest

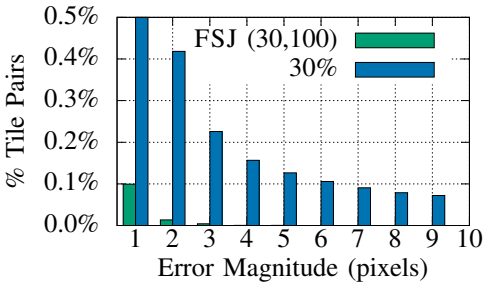
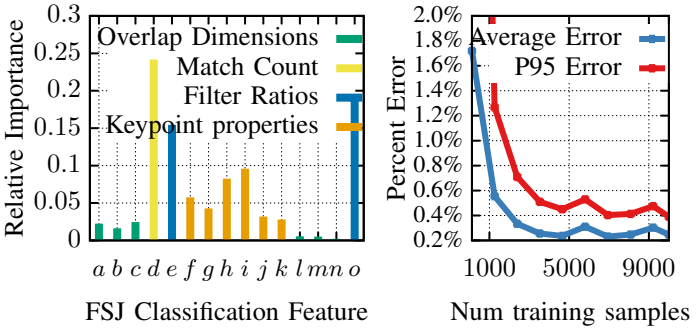


Fig. 6: Percentage of overlapping tile pairs misaligned when downsampling to 30% resolution on *human100*.



(a) FSJ Feature Importance (b) FSJ Training

Fig. 7: Figure 7a shows the relative importance of variables used by FSJ on *human100* dataset. Figure 7b illustrates FSJ classification error during training.

and allow it to scale all the way to the human brain, while only requiring about 4TB of memory.

A good estimate of the memory requirements of Stacker can be obtained via a back-of-the-envelope calculation. For each image tile of size $\approx$ 8MB, Stacker requires memory for approximately 100 corresponding points and 12 triangles. Each corresponding point, along with its SIFT feature vector, is represented using 156 bytes, and each triangle requires 128 bytes. Each section of a human brain ($10cm^2$) is composed of approximately 125 million tiles, and so the total memory requirements of Stacker per-section is approximately 2TB. Since Stacker stores data for 2 sections while computing a pairwise alignment, Stacker requires about 4TB of memory to perform 3D alignment on sections of a human brain.

Presently, the volumes being aligned in connectomics are much smaller than the human brain. For the "grand challenge" of reconstructing an entire mouse brain, which is 100x smaller by volume than the human brain, the same calculation shows that Stacker requires only about 50GB of memory.

V. Frugal Snap Judgments

This section describes a technique called *frugal snap judgments* (FSJ) that is used to accelerate the performance of Quilter by a factor of 3–5x, without a significant loss of alignment accuracy.

Design of FSJ in Quilter

Frugal snap judgments are used in Quilter to obtain more advantageous performance–accuracy trade-offs when computing a tile pair's relative alignment.

Consider, for example, a "fast path" algorithm for computing pairwise tile alignments that operates on images downsampled to 30% resolution. As shown in Figure 6, downsampling

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

results in an alignment error for approximately 0.4% of all tile pairs (and, as a consequence, approximately 2%–3% of tiles have a misaligned neighbor). Downsampling results in a less accurate algorithm relative to the original code path that operated on full-resolution images. Yet, for the vast majority of tile pairs, the fast path computes a result that matches the slow path within a tolerance of 1-2 pixels.

To detect when the result of the fast path is unreliable, we employ random-forest classification over feature vectors that summarize the intermediate results of the fast-path algorithm. The feature vectors include the following information: the area, width, and height of the overlapping region between a pair of tiles, the fraction of keypoints matched, the fraction of matched keypoints that are filtered by RANSAC, the total number of filtered keypoints, and aggregate statistics from the filtered keypoints. Additional details are provided in Figure 7a. These feature vectors are small and inexpensive to compute since they are obtained from the intermediate results of the fast path and not from the raw pixel data in image tiles.

Training the fast-path error detector

Data to train the FSJ error detector is obtained through random sampling of overlapping tile pairs from the stack and executing the fast and slow path codes on the same input. The relative offsets between the tiles computed by the fast and slow path are compared and considered matching if they differ by less than 1px. We extract a feature vector from the fast-path result and add it to either the training or testing sets.

During training, a *false positive* occurs when the detector incorrectly predicts that the fast/slow codes will match, and a *false negative* occurs when it incorrectly predicts that they will disagree. A well-trained detector will minimize the false negative rate, while strictly constraining the false positive rate.

Figure 7b illustrates the evolution of the false-positive rate, while training the FSJ classifier. A 95th percentile confidence bound on the false-positive rate is estimated, using the testing set, during training and used as a stopping criterion. For our datasets, we typically stopped training once the average false-positive rate fell to 0.2% with a 95th percentile confidence bound below 0.4%. The false-negative rate of our FSJ classifiers varied between $\approx 4 - 10\%$ depending on the resolution of the fast path and the dataset. Training time does not depend on the size of the dataset and took between 10-20 minutes on an 18-core Intel Xeon CPU (E5-2666 v3, 2.9GHz).

After training on the *human100* dataset using 30% resolution images in the fast path, we achieved an out-of-bag error of $6.7e^{-2}$, an average false-positive rate of 0.2%, and a false negative rate of 6%. The relative variable importance scores for the random forest classifier are provided in Figure 7a.

Figure 6 shows the 2D alignment errors of Quilter when using FSJ on a set of 4 sections not used during training. When using FSJ, there are nearly no errors greater than one pixel, and no errors greater than five pixels.

VI. System Evaluation

This section provides end-to-end performance results for the alignment pipeline built using the Quilter and Stacker algorithms on three datasets across four computing platforms.

Experimental setup

A summary of the software, datasets and hardware used in our evaluation is as follows:

Software. The alignment pipeline composed of Quilter and Stacker was implemented as a C++ software library parallelized using Cilk Plus [17], [18] and the Tapir [19] branch of the LLVM [20], [21] compiler (version 6)[3]. The following software libraries were used: OpenCV v3.2.0 [22], OpenJPEG v3.2.0 [10], and Google protocol buffers [23].

Datasets. Three datasets were used in our evaluation: *Mouse50*, *Mouse200*, and *Human100*. *Mouse50* is a 550GB dataset composed of 50 sections and 65,000 image tiles. *Mouse200* is a 2TB dataset composed of 200 sections and 200,000 image tiles. *Human100* is a 100-section 38TB dataset.

Hardware. Our evaluations employ four different computing platforms to evaluate runtime performance: *Common Multicore*, *Large Multicore*, *LLSC Cluster*, and *AWS Cluster*. Figure 8 details the hardware for each platform, and additional details are provided in Section A.

Multicore performance of Quilter and Stacker

Let us first analyze the efficiency of Quilter on the *Common Multicore* system relative to FijiBento. For this experiment, we used the *mouse50* dataset. Both Quilter and FijiBento compute equivalent 2D alignments, but FijiBento has additional overheads due to the serialization of intermediate results to disk. In Figure 9, we see that Quilter out-performs FijiBento by a factor of 2x, when operating on full-resolution data. The use of frugal snap judgments in Quilter (Quilter FSJ(20,100) in Figure 9) provides a further performance boost of 5.6x and causes Quilter to outperform FijiBento by a factor of 11.3x.

The performance of Stacker on the *Common Multicore* platform is similarly efficient. To 3D align the *mouse50* dataset Stacker required only 8 minutes of compute time on *Common Multicore*. The total runtime to 2D and 3D align *mouse50* was 40 minutes, as shown in Figure 8.

Next, let us analyze the performance of Quilter and Stacker on the *Large Multicore* platform to evaluate its scalability on a single machine with a larger number of processing cores.

Figure 10 illustrates the speedup achieved on the *Large Multicore* platform when aligning 4 sections of the *mouse200* and *human100* dataset. The left y-axis shows the speedup achieved on the *mouse200* dataset relative to a serial execution. On *mouse200* approximately 10x speedup is achieved on a 28-core (1 full socket) execution relative to a serial execution. We believe the sublinear speedup is largely due to aggressive downclocking on the socket when using AVX instructions [24]. On 112-cores (4 sockets), the speedup achieved on *mouse200* relative to a serial execution is approximately 39x. For the *human100* dataset, we report the scalability relative to a 1-socket execution using the right y-axis.[4] Similar to *mouse200*, each additional socket provides near-linear improvements in performance when aligning the *human100* dataset.

Scaling across multiple machines in a cluster.

Now, we evaluate the scalability of Quilter and Stacker when executing the pipeline across many multicores in a computing cluster. For these experiments, we use the *AWS Cluster* platform to align the *mouse200* and *human100* datasets.

[3]Available from `http://cilk.mit.edu`

[4]A serial execution of a section of *human100* is unduly time-consuming.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

Dataset	FSJ	Platform	Hardware	Wall-clock runtime	Throughput
mouse200	FSJ30	*AWS Cluster*	200 8-core AWS C4 Instances (1600 cores)	5.6 minutes	21.4 TB/hr
mouse200	FSJ30	*LLSC Cluster*	170 Opteron nodes (5440 cores)	16 minutes	7.5 TB/hr
mouse200	FSJ30	*Large Multicore*	112-Core Intel Xeon Platinum 8180	80 minutes	1.5 TB/hr
human100	FSJ30	*Large Multicore*	112-Core Intel Xeon Platinum 8180	26.7 hours	1.4 TB/hr
mouse50	FSJ30	*Common Multicore*	18-Core AWS C4 Instance	49 minutes	0.67 TB/hr
mouse50	FSJ20	*Common Multicore*	18-Core AWS C4 Instance	40 minutes	0.82 TB/hr

Fig. 8: End-to-end performance results for whole alignment pipeline executing both Quilter and Stacker. In the FSJ column, FSJ30 and FSJ20 indicate that the fast path employed by FSJ used 30% and 20% resolution images, respectively.

2D Alignment Method	Runtime	Throughput
FijiBento	362 minutes	0.091 TB/hr
Quilter Full Resolution	180 minutes	0.18 TB/hr
Quilter FSJ(20,100)	32 minutes	1.03 TB/hr

Fig. 9: Performance comparison of FijiBento and Quilter performing 2D alignment on the *Common Multicore* platform.

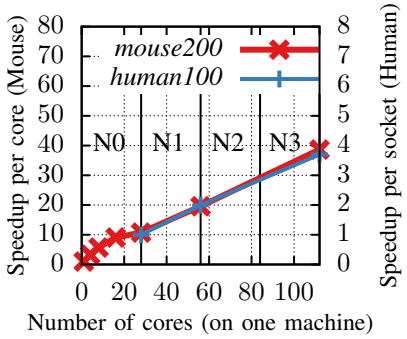


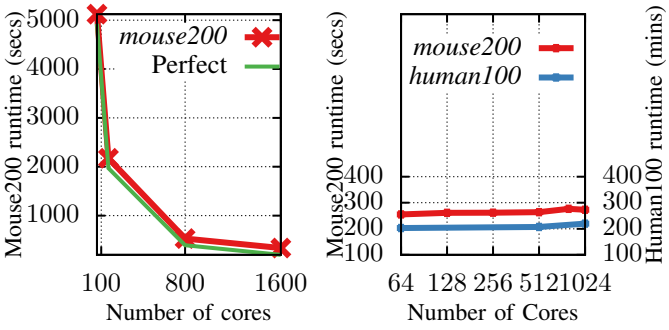
Fig. 10: Vertical scalability. Speedup obtained when executing 4 sections of *mouse200* and 4 sections of *human100* on the *LargeMulticore* platform. The *mouse200* scalability is relative to a 1-core executing using the left-axis of the plot. The *human100* scalability is relative to a 1-socket execution using the right-axis. The mapping of cores to sockets is provided via the $N0, N1, N2$, and $N3$ labels.

The strong scaling of the pipeline is shown in Figure 11a. For this experiment, when running the pipeline on N nodes, we divide the *mouse200* stack into $200/N$ blocks. A block is first processed by a single task that runs Quilter and Stacker back-to-back as a single, shared-memory process on the sections within a block. Next, $N-1$ tasks are created that run Stacker on pairs of sections that border adjacent blocks. On the *mouse200* dataset and the *AWS Cluster* platform, we observe near-linear strong scaling.

The weak scaling of the alignment pipeline is shown in Figure 11b. For the weak scaling experiment, we ensured that the work-per-node was constant by scaling the number of sections being aligned proportionately to the number of machines used. We observe no appreciable decrease in efficiency-per-node when scaling from 1 to 128 machines where each machine is an 8-core Intel E5 node.

End-to-end system experiments

Figure 8 illustrates the absolute runtime for the *mouse200*, *human100*, and *mouse50* datasets on the *Common Multicore*, *Large Multicore*, *LLSC Cluster*, and *AWS Cluster* systems. Due to the difficulty of moving the full *human100* dataset to different platforms, we only ran a full end-to-end test on *human100* on the *Large Multicore* platform. The highest



(a) Strong scaling. (b) Weak scaling.

Fig. 11: Strong and weak scalability on the *AWS Cluster* platform on the *mouse200* and *human100* datasets.

throughput was achieved on the *AWS Cluster* platform which was able to align data at a rate of 21.4 terabytes per hour, using 1600 cores.

VII. Conclusion

This paper presented an image alignment pipeline for connectomics that makes efficient use of commodity multicore hardware. The design of Quilter and Stacker was guided by an understanding of future data sizes on the horizon in connectomics. Through careful management of memory, both Quilter and Stacker can align future datasets in connectomics on a single multicore. By designing Stacker to perform elastic 3D alignment by composing pairwise alignments, our pipeline is able to efficiently scale horizontally over multiple multicore nodes in a cluster. Finally, the use of frugal snap judgments enabled the exploitation of application-specific performance–accuracy trade-offs in Quilter that improved the performance of 2D alignment by 3–5x.

Connectomics is a rapidly evolving field, and every stage of the software pipeline, including alignment, is subject to continuous innovation. The alignment pipeline presented in this paper demonstrates efficient methods for performing "standard" image alignment in connectomics, but it does not yet address all important edge cases. For example, when a large volume is sliced into sections it is necessary to periodically replace the knife, and the replaced knife will slice the volume at a slightly different angle. The next several sections produced after a knife replacement will contain very little data, and there is some sophistication required to correctly handle these sections. As a second example, the thin slices of tissue can fold in on themselves, and there have been efforts to detect and identify these folds during the alignment process. Although we have observed good 2D and 3D alignment when applying our pipeline to large real world data, the resolution of edge cases such as these are vital, in practice, to having a highly effective alignment system for connectomics.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

VIII. Acknowledgements

We thank Nir Shavit, Alexander Matveev, Yaron Meirovitch and the other members of the Computational Connectomics Group for their support and helpful discussions. We thank Charles Leiserson, Tao Schardl, and the members of the Supertech research group for support and helpful discussions. We thank Jeff Lichtman and the Lichtman Lab for providing access to datasets and computational resources. We thank Adi (Suissa) Peleg for his early assistance familiarizing us with the existing image alignment pipelines in connectomics. We thank the organizers and attendees of the Max Planck / HHMI Connectomics Conference for helpful discussions related to image alignment and other computational problems in connectomics. We thank Lincoln Laboratory Supercomputing Center for providing HPC resources that contributed to the results in this paper. We thank Chansup Byun for providing assistance running experiments on the LLSC system.

Support is gratefully acknowledged from the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000; the National Science Foundation (NSF) under grants IIS-1447786, CNS-1017058, CCF-1162148, CCF-1314547, CCF-1563880; the Intelligence Advanced Research Projects Activity (IARPA) under grant 138076-5093555; and, the MIT-IBM Watson AI Lab under grant 027397-00059. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Appendix A
Computing Platforms and Datasets

This section provides additional details on the computing platforms and datasets employed to evaluate Quilter and Stacker in Section VI.

Common Multicore is an 18-core, 2-way hyperthreaded Intel Xeon CPU (E5-2666 v3, 2.9GHz) available as a 4th-generation compute-optimized machine from Amazon Web Services which has 64GB of memory and runs Ubuntu 14.04 on Linux Kernel 3.13.0-106. Amazon EFS [25] (elastic filesystem) is used to store image data files. Amazon EFS provides performance tiered to the size of the mounted volume. Based on these tiers and our mounted volume size of 2.8TiB our maximum burst I/O throughput was 100 MiB/s during our experiments on this platform.

Large Multicore is a 112-core, 2-way hyperthreaded Intel Xeon Platinum 8180 CPU 2.5GHz with 1.5TB of memory running Centos7 on Linux Kernel 3.10.0-862. This machine was part of the Odyssey Cluster and retrieves stored image data from Lustre mounted storage connected via 56 Gb/s FDR InfiniBand network.

LLSC Cluster is a shared supercomputing center LLSC (Lincoln Lab Supercomputing Center) [26] using the AMD Opteron partition of the TX-Green system, consisting of 274 compute nodes, each containing two, 16-core AMD Opteron(TM) Processor 6274, running at 2.2GHz, for a total of 32 cores per node, with 128 GB memory per node. The nodes employ a shared Lustre filesystem and are connected with a 10 GigE Arista switch. A total of 170 Opteron compute nodes and 64 Intel E5 nodes were available for our experiments on this platform. Experiments run on more than 4 nodes were run on LLSC's private cluster by a third party with guidance from the authors.

AWS Cluster is a 200-node cluster, where each node is an 8-core version of the *Common Multicore* platform. AWS ParallelCluster software is used to manage the nodes as a SLURM cluster and EFS is used as the cluster's shared filesystem.

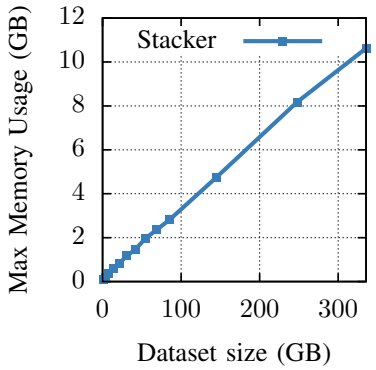


Fig. A1: Memory high-water mark of Stacker versus the size of the sections being aligned. Stacker computes the relative alignment of pairs of sections. The total dataset size in this plot refers to the total size of all tiles in the two sections being aligned by one Stacker task.

Appendix B
Empirical analysis of Stacker's Memory Usage

This section provides details on the empirical memory usage of Stacker.

We ran an experiment to empirically measure the memory requirements of Stacker. We followed a similar methodology to that used in Section III to analyze Quilter's memory requirements. Regions of varied size were extracted from a section of the *human100* dataset, and then this section was duplicated to construct a stack of two identical sections. Stacker then executed to align this dataset. Figure A1 shows the results of the empirical experiments measuring Stacker's memory requirements. Stacker uses 300 KB of data per tile, which is $\approx$ 20x more than analytic estimates. There are two reasons for the gap: (a) presently our implementation of Stacker uses 4-byte floats to represent keypoint descriptors where 1-byte is sufficient (since values are discretized into 255 bins); and, (b) aligning a section to its identical copy results in an abnormally large number of inter-section keypoint matches. As such, this experiment illustrates a worst-case scenario for Stacker's memory usage. Regardless, even in this scenario all datasets but the human brain remain a feasible task on commodity multicore hardware.

References

[1] J. W. Lichtman, H. Pfister, and N. Shavit, "The big data challenges of connectomics," *Nature neuroscience*, vol. 17, no. 11, p. 1448, 2014.

[2] J. W. Lichtman and W. Denk, "The big and the small: challenges of imaging the brains circuits," *Science*, vol. 334, no. 6056, pp. 618–623, 2011.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE

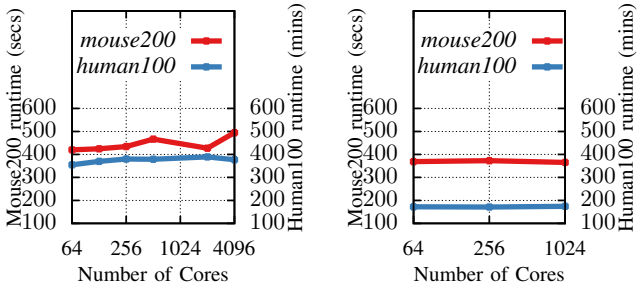
constant	value
pixel resolution	3×3nm
tile height	2724 pixels
tile width	3128 pixels
raw tile	8.5 MB
JP2 tile	832 KB
tile metadata	4 KB
keypoint size	156 bytes
keypoints per image	8,000–20,000

region size	# tiles
1mm row	107
1cm row	1,066
10cm row	10,656
$1mm^2$ *section*	13,040
$1cm^2$ *section*	1,304,000
$10cm^2$ *section*	130,400,000

Fig. A2: Values for the different constants needed to calculate memory usage of a 2D alignment algorithm.

Dataset	Z	Size	Description
mouse200	200	2 TB	Subset of 100umSept2017 dataset from $100\mu^3$ volume of mouse brain, stored in J2K compressed format.
mouse50	50	0.55 TB	Subset of 100umIARPASep14 dataset from $100\mu^3$ volume of mouse brain, stored in JPEG compressed format.
human100	100	38 TB	ROI2w05 which is a subset of a 600 TB dataset obtained from human brain biopsy, stored in J2K compressed format.

Fig. A3: Dataset descriptions. The Z column provides the number of $2D$ sections in the dataset. Datasets were obtained from the Lichtman Lab using the Zeiss multiSEM microscope.



(a) *LLSC Cluster* running on Opteron nodes with 32-cores.

(b) *LLSC Cluster* running on Intel E5 nodes with 16-cores.

Fig. A4: Weak scalability experiments on the LLSC cluster. Illustrates reported runtimes on the *LLSC Cluster* platform on the *mouse200* and *human100* datasets when the stack input-size scales with the number of nodes used to execute Quilter and Stacker. The left y-axis provides the runtime in seconds for *mouse200* and the right y-axis provides the runtime in minutes for *human100*.

[3] N. Shavit, "A multicore path to connectomics-on-demand," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2016, pp. 211–211.

[4] S. Saalfeld, A. Cardona, V. Hartenstein, and P. Tomančák, "As-rigid-as-possible mosaicking and serial section registration of large sstem datasets," *Bioinformatics*, vol. 26, no. 12, pp. i57–i63, 2010.

[5] T. Tasdizen, P. Koshevoy, B. C. Grimm, J. R. Anderson, B. W. Jones, C. B. Watt, R. T. Whitaker, and R. E. Marc, "Automatic mosaicking and volume assembly for high-throughput serial-section transmission electron microscopy," *Journal of neuroscience methods*, vol. 193, no. 1, pp. 132–144, 2010.

[6] Rhoana, "Fijibento," 2018. [Online]. Available: https://github.com/Rhoana/FijiBento

[7] L. Scheffer, B. Karsh, and S. Vitaladevun, "Automated alignment of imperfect em images for neural reconstruction," vol. abs/1304.6034, 04 2013.

[8] P. J. Braam and R. Zahir, "Lustre: A scalable, high performance file system," *Cluster File Systems, Inc*, 2002.

[9] D. Hildebrand and P. Honeyman, "Exporting storage systems in a scalable manner with pnfs," in *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*. IEEE, 2005, pp. 18–27.

[10] A. Descampe, F. Devaux, H. Drolon, D. Janssens, and Y. Verschueren, "Openjpeg 2.0. 0," 2012.

[11] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[12] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *European conference on computer vision*. Springer, 2006, pp. 404–417.

[13] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*. Ieee, 2011, pp. 2564–2571.

[14] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[15] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Mesh optimization," in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '93. New York, NY, USA: ACM, 1993, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/166117.166119

[16] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson, "Executing dynamic data-graph computations deterministically using chromatic scheduling," in *SPAA*, 2014.

[17] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "An analysis of dag-consistent distributed shared-memory algorithms," in *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, Jun. 1996, pp. 297–308.

[18] C. E. Leiserson, "The Cilk++ concurrency platform," *Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, March 2010.

[19] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into llvm's intermediate representation," in *ACM SIGPLAN Notices*, vol. 52, no. 8. ACM, 2017, pp. 249–265.

[20] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec. 2002, *See* `http://llvm.cs.uiuc.edu`.

[21] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004, p. 75.

[22] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[23] Google, "Protocol buffers," https://developers.google.com/protocol-buffers/, 2019.

[24] WikiChip, "Xeon platinum 8180 - intel," 2020. [Online]. Available: https://en.wikichip.org/wiki/intel/xeon˙platinum/8180

[25] Amazon, "Amazon elastic file system," https://aws.amazon.com/efs/, 2019.

[26] A. Reuther, J. Kepner, C. Byun, S. Samsi, W. Arcand, D. Bestor, B. Bergeron, V. Gadepally, M. Houle, M. Hubbell *et al.*, "Interactive supercomputing on 40,000 cores for machine learning and data analysis," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–6.

978-1-7281-9219-2/20/$31.00 ©2020 IEEE