A Multicore Path to Connectomics-on-Demand

Alexander Matveev * Yaron Meirovitch * Hayk Saribekyan
Wiktor Jakubiuk Tim Kaler Gergely Odor
David Budden Aleksandar Zlateski Nir Shavit
Massachusetts Institute of Technology

Abstract

The current design trend in large scale machine learning is to use distributed clusters of CPUs and GPUs with MapReduce-style programming. Some have been led to believe that this type of horizontal scaling can reduce or even eliminate the need for traditional algorithm development, careful parallelization, and performance engineering. This paper is a case study showing the contrary: that the benefits of algorithms, parallelization, and performance engineering, can sometimes be so vast that it is possible to solve "cluster-scale" problems on a single commodity multicore machine.

Connectomics is an emerging area of neurobiology that uses cutting edge machine learning and image processing to extract brain connectivity graphs from electron microscopy images. It has long been assumed that the processing of connectomics data will require mass storage, farms of CPU/GPUs, and will take months (if not years) of processing time. We present a high-throughput connectomics-on-demand system that runs on a multicore machine with less than 100 cores and extracts connectomes at the terabyte per hour pace of modern electron microscopes.

CCS Concepts **• Computing methodologies → Concurrent algorithms; • Computer systems organization → Multicore architectures; • Computing methodologies → Neural networks; • Applied computing → Computational biology**

Keywords Multicore Programming, Machine Learning, Big-Data, Connectomics

*These authors contributed equally to this work

PPoPP '17, February 04-08, 2017, Austin, TX, USA
© 2017 ACM. ISBN 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3018743.3018766

1. Introduction

The conventional wisdom in machine learning is that large scale "big-data" problems should be addressed using distributed clusters of CPUs, GPUs or specialized tensor processing hardware in the cloud [6, 11, 20–22, 36, 63]. This paper presents a solution to one of the most demanding big-data machine learning areas: connectomics. It provides a case study in how novel algorithms combined with proper parallelization and performance engineering, can reduce the problem from a large cluster to a single commodity multicore server (that can be placed in any neurobiology lab), eliminating the crucial bottleneck of transferring data to the cloud at terabyte-an-hour rates.

1.1 High-Throughput Connectomics

Perhaps neuroscience's greatest challenge is to provide a theory that accommodates the highly complicated structure and function of neural circuits. These circuits, found in flies, in mammals, and ultimately, in humans, are conjectured to be the substrate that enables the complex behavior we call "thought." However, to this day, our ability to provide such a theory has been hampered by our limited ability to view even small fragments of these circuits in their entirety. Neurobiologists have had sparse circuit maps of mammalian brains for over a century [57]. However, as surprising as this may seem, no one has seen the dense connectivity of even a single neuron in cortex, that is, all of its input and output connections (called synapses) to other neurons. No one has been able to map the full connectivity among even a small group of neighboring neurons, not to mention the tens of thousands of neurons that constitute a "cortical column." Without such maps, it would seem hard to understand how a brain consisting of billions of interconnected neurons actually works. Would you believe that someone understood how a modern microprocessor worked if they could tell you in great detail how individual transistors operated but were unable to describe how these transistors are interconnected?

Mapping brain networks at the level of synaptic connections, a field called *connectomics*, began in the 1970s with a study of the 302-neuron nervous system of a worm [67].

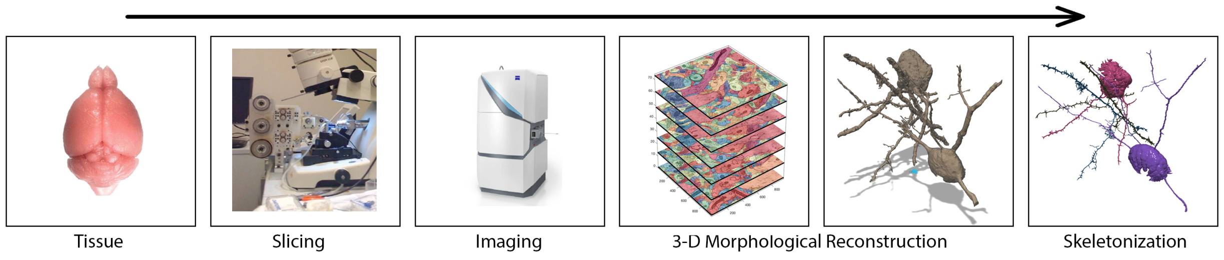

Figure 1: The Stages of a high-throughput connectomics pipeline.

This study, which required capturing and combining hundreds of electron microscopy images, was done by hand and took 8 years. Manual mapping of minute volumes of neural tissue have recently produced breakthrough results in neuroscience [28, 34, 49]. However, extending the approach to the millions and billions of neurons in higher animals seems an unattainable goal without a fast and fully automated pipeline.

Recent advances in the design of multi-beam electron microscopes now allow researchers to collect the nanometer resolution images, necessary to view neurons and the synaptic connections between them, at unprecedented rates. A cubic millimeter of brain tissue, enough to contain a mouse cortical column, is within reach. This is only a tiny sliver of brain, about the size of a grain of sand, but it will contain about 100 thousand neurons and a billion synapses. The cubic millimeter will constitute about two petabytes of imagery that will be collected in about 6 months using a 61-beam electron microscope that generates half a terabyte of imagery per hour [15, 38].

A modern connectomics pipeline [40], as depicted in Figure 1, consists of a physical part and a computational part. The physical part takes a piece of stained brain tissue embedded in a resin, slices it thousands of times using a special microtome device, and feeds these minute slices into an electron microscope that scans them and produces separate images of the slices [15, 61]. The computational part of the pipeline, the focus of this paper, then takes the thousands of separate 2-dimensional images, reconstructs the 3-dimensional neurons within them, and produces skeletonizations or graphs that capture their morphological and connectivity properties.

However, a viable solution to the computational problem of extracting the skeletonizations and connectivity maps from the image data still seems far away. Using existing algorithms and at the present computing rates, the common assumption is that extracting the connectivity of the circuits within two petabytes of data may take years and require supercomputing clusters [38, 66]. It is even unclear how to efficiently move the vast amounts of data from the microscope to a large scale storage and compute facility where it will be processed [40]. The prospect of connectomics-on-demand, where neurobiology labs around the world each run their own microscopes and extract connectomes "as needed" seems far far away.

This paper takes a first step towards proving the feasibility of designing a high-throughput connectomics-on-demand system that runs on a multicore machine with less than 100 cores and extracts connectomes at the terabyte-per-hour pace of modern microscopes. Such a system, once achieved, will eliminate the need to transfer and store petabytes of data in special warehouses. It could be readily deployed in labs across the world, allowing scientists to view connectomes as they come off the scope. Down the road, such efficient connectomics systems could make it feasible for neurobiologists to extract the complete connectome of a mouse cortex of about 12 million neurons and 120 billion synapses from about 100 petabytes of data, and eventually make it possible to analyze exabyte-scale parts of the connectome of both healthy and diseased human brains (Human brain has roughly 100 billion neurons and a quadrillion synapses).[*]

1.2 Towards an Automated Terabyte-Per-Hour Connectomics Pipeline

The "proof of concept" connectomics system we present here processes a terabyte of data, from image-stack to detailed skeletons, in less than **4 hours** on a single 72 core Haswell-based multicore machine with 500GB of memory. The upcoming generation of both GPU and CPU chips, with some further optimizations (see our performance section), easily place it within the target terabyte-an-hour performance envelope of todays fastest electron microscopes. One should contrast this with the recent connectomics system of Roncal *et al.* [58] that uses a cluster of 100 AMD Opteron cores, 1 terabyte of memory, and 27 GeForce GTX Titan cards to process a terabyte in **4.5 weeks**, and the state-of-the-art distributed MapReduce based system of Plaza and Berg [56] that uses 512 cores and 2.9 terabytes of memory to process a terabyte of data in **140 hours** (not including skeletonization). Importantly, the speed of our pipeline does not

[*]These numbers may sound like science fiction, yet as Lichtman and Sanes note by analogy [39], sequencing the first human genome took multiple labs around the world a concerted effort over 15 years, while today a single lab can sequence a human genome within hours.

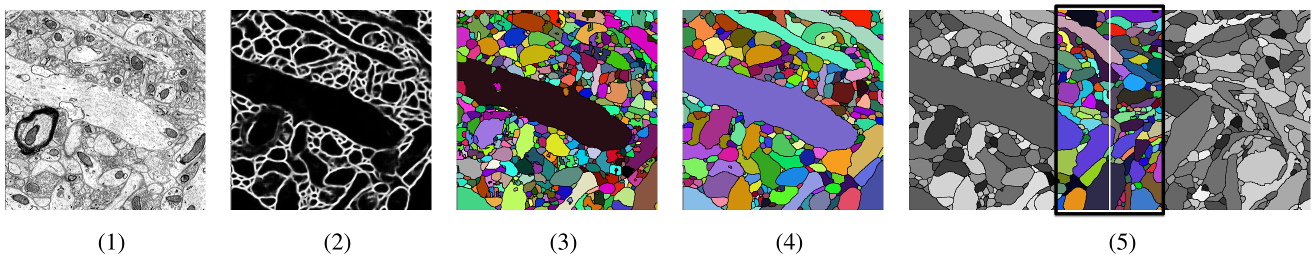

Figure 2: 2D visualization of connectomics pipeline stages: (1) Electron microscope (EM) image. (2) Membrane probabilities generated by CNN. (3) Over-segmentation generated by Watershed. (4) Neuron reconstruction generated by agglomeration. (5) Pipeline inter-block merging: for each two adjacent blocks, the pipeline slices a boundary sub-block and executes agglomeration.

come at the expense of accuracy, which is on par or better than existing systems in the literature [29, 56, 58] (using the accepted *variation of information* (VI) measure [44]).

Our high-level pipeline design builds on prior work [29, 42, 51, 52, 55, 56]. It passes the data through several stages as seen in Figure 2. The image data is split into blocks. The pipeline first runs a convolutional neural network (CNN) on the separate image blocks to detect the boundaries of neurons. The neuronal objects defined by the boundaries are then segmented into objects by a watershed algorithm. At this point the image is over-segmented, that is, neuronal objects are fragmented. The next step in the reconstruction is to run an agglomeration phase that merges all the fragments in a block into complete objects. Then the blocks are merged, so that objects now span the entire volume. Finally, as depicted in Figure 8, the objects are skeletonized.

1.3 Our Contributions

- A new CPU execution engine for CNNs that scales well on multicore systems, processing 1024x1024 images 70x faster than previous state-of-the-art [72] on a single 18-core Haswell CPU. Our code applies GCC-Cilk (a state-of-the-art work-stealing scheduler [3]), is optimized to leverage Intel AVX2 instructions [68] and maximizes memory reuse in L1, L2 and L3 cache subsystems. Analysis shows that this code achieves 80% utilization of the peak theoretical FLOPS of the system on a single-thread and scales almost linearly, compared to previous approaches (Caffe CNN framework with Intel MKL [10, 65]) that exhibit only 20% utilization and no scalability beyond 4 threads. The remaining performance gap is obtained by implementing minimal “dense” (fully-convolutional) CNN inference, which involves 284x fewer computations than naive implementations.
- A new GPU execution engine for CNNs that leverages custom fast Fourier transforms (FFTs) and the latest cuDNN primitives [53]. The system performs $\sim$2x faster for our CNN benchmarks than previous state-of-the-art [50] on a Titan X GPU.
- Parallelized and performance engineered version of NeuroProof, a system originally developed in Janelia Research Labs by Parag, Chakrobarty and Plaza [54]. This system implements an agglomeration procedure that reconstructs 3D neurons from the watershed oversegmentation. Specifically, we redesigned the Regional Adjacency Graph (RAG) algorithm as an augmented merge-sort, efficiently defer expensive operations that occur during constructions and traversals of RAGs, and batch computations for lazy execution. Our implementation is 2.3x faster on a single-thread, and has 85x scalability factor on our 72-core server (super-linear factor due to Hyper-Threading).
- Two new algorithms for the connectomics domain: (1) a new inter-block merging algorithm that, unlike prior approaches, applies parallelized NeuroProof to optimize object-pair merges, and (2) a parallel skeletonization algorithm that uses novel techniques and GCC-Cilk based chromatic scheduling [25, 26] to execute efficiently on multicores. We describe the algorithmic side in more detail in [46].

We believe that our work departs from existing systems by showing that a combination of new algorithms, proper parallelization of existing algorithms, modern scheduling using languages like Cilk, and performance engineering of code to fully utilize CPU resources, can move the connectomics problem from the large scale distributed systems space to run on a commodity shared-memory multicore system. This might be true for other big-data machine-learning based problems in medicine and the life sciences, and even for problems where big systems are necessary, our approach may provide insights on how to make individual system elements much faster and scalable.

1.4 Related Work

The image segmentation approach we use was developed by [1] and suggested for automatic electron microscopy segmentation by [51]. It is used by most of todays systems [29, 42, 51, 52, 55, 56]. Increasingly accurate membrane predictors based on convolutional neural networks have recently been proposed [8, 59]. Our system implements a solution that is inspired by these algorithms, is on-par with their current accuracy levels, and yet is simpler and faster to execute. Other studies have focused on new approaches to segmentation of supervoxels (e.g., [42]). Further research however is required to benchmark their accuracy on even gigabyte size datasets, so at this time their scalability cannot be addressed.

We are aware of three previous connectomics pipelines: the original RhoAna pipeline by Kaynig *et al.* [29], the system by Roncal *et al.* that extends RhoAna [58], and state-of-the-art Plaza and Berg's Spark-based system [56] that uses NeuroProof.

2. System Overview

This is the story of how one can replace the use of a large distributed system with a single multicore machine. Although a large distributed system may have tremendous computational resources at its disposal, its raw computing power comes with a cost of additional overheads and system complexity, e.g. data must be moved over the network, and the system must support a degree of fault tolerance [6, 12, 37, 56, 70, 71]. These overheads tend to be small for problems that are embarrassingly parallel, but can come to dominate the execution time of more complex computations that need to operate on shared data. In addition, the opportunities to obtain performance within a single multicore are abundant, and can result in performance improvements that rival or even dwarf those obtained through horizontal scaling. As we will see, the connectomics problem requirements can be satisfied using a single multicore machine to execute a pipeline of carefully designed multicore algorithms that efficiently utilize a machine's available computing cycles, take advantage of the low shared-memory communication overheads, and parallelize across cores using efficient task scheduling tools.

2.1 Pipeline Structure

The input to our pipeline is a sequence of aligned 2D images each of which represents a single slice of a 3D volume of brain that was captured at high resolution ($\sim$3-4nm) by an electron microscope. These images are broken down into smaller sub-images with a standardized size of 1024x1024 pixels, which are then grouped into blocks of 1024x1024x100 pixels. The pipeline executes a segmentation procedure that extracts the neuronal objects within each block, and then executes an inter-block merging procedure that "combines" the per-block segmentations to obtain a complete segmentation of the original input volume [46]. These stages of the connectomics pipeline are illustrated in 2D within Figure 2.

First, as seen in Parts 1 and 2 of Figure 2, a convolutional neural network (CNN) is executed to detect membranes (or cell borders). This network was trained using "ground truth" annotation by human experts. The training process is time-consuming, but since it only needs to be performed once on an annotated subset of the dataset it does not impact the pipeline's throughput. The result of applying the CNN to a subvolume is a new 1024x1024x100 block where each pixel indicates a membrane probability between 0 and 1. Since the CNN may run on blocks independently, this stage of the pipeline may be executed in a distributed manner. It turns out, however, that careful performance engineering enabled this stage of the pipeline to execute sufficiently fast on a single multicore machine. In fact, we found that our optimized code for executing the CNN was able to significantly outperform existing state-of-the-art CPU [10, 65, 72] and GPU [50] implementations. In Section 3 we describe the design of this CNN, and the steps taken to ensure that the our implementations made efficient use of the machine's compute cycles, L1/L2/L3 caches, and disk.

Next, as seen in Part 3 of Figure 2, a 3D watershed algorithm executes on the membrane probability map. The watershed algorithm performs a BFS-style flood from "seed" pixels that have 0 probability of being a membrane, to pixels with higher probability. The result of the watershed execution is a new block in which segments (3D objects) have been formed around seeds, each with a segment identifier. Methods of parallelizing watershed have been described in the literature (e.g. [48]), but it turned out that our pipeline was able to achieve better performance by engineering an efficient sequential algorithm with low-memory consumption. This strategy allowed us to obtain a watershed algorithm that is an order of magnitude faster than the code provided in the popular OpenCV [23] library, and whose low-memory requirements permit us to run many independent instances of the algorithm on a shared memory machine. Section 4 describes the design of this serial watershed algorithm in greater detail.

The segments produced by the watershed algorithm represent an over-segmentation of the true neuronal objects; i.e. each neuron might be fragmented into many smaller parts, as shown in Part 3 of Figure 2. To obtain segments that represent whole neurons an agglomeration algorithm is employed that merges segments that lie within the same 3D object. The result of the agglomeration procedure is a collection of larger segments that each represent a whole neuronal object, as is illustrated (in 2D) in Part 4 of Figure 2. The agglomeration procedure used by the pipeline is based upon the serial NeuroProof agglomeration algorithm of Parag *et al.* [54]. We reformulated the Neuroproof procedure to use parallel algorithms, and performed a variety of performance optimizations to reduce the total work and memory usage. These optimizations obtained a 2.3x improvement in serial runtime, and a 6.5x improvement when executing on 4-cores. Moreover, we achieve near-linear scalability (when running multiple 4-core instances simultaneously) which provides an additional 82x speedup over the optimized serial code on our 72-core server with HyperThreading. Section 5 describes, in greater detail, the parallelization techniques and performance optimizations that were employed to optimize the agglomeration stage of the pipeline.

After a segmentation has been obtained for each block, the pipeline executes an efficient inter-block merging procedure on the reconstructed blocks [46]. Here we utilize the shared memory properties of the machine in which I/O operations are automatically cached by the OS kernel. Thus, for every two adjacent blocks, our block merging algorithm carves out a thin sub-block near the shared boundary and employs a variant of our parallel agglomeration procedure to identify and merge objects across this boundary. Part 5 of Figure 2 is a two dimensional rendering of this merging process. The result of each merging is a small file containing pairs of segment identifiers that should be combined, and it is straightforward (and inexpensive) to combine all of them into a full segmentation. This contrasts with the approach of Plaza and Berg [56] where a sophisticated merge algorithm must be executed across many machines in the network. In our case, there is no need to perform expensive data-transfers over the network and to support a complex failure detection and recovery mechanisms, since the whole system executes on a single server. The design of the inter-block merging procedure is discussed further in Section 6.

The final step of the pipeline skeletonizes the volume segmentation (see Figure 8) on a per-block basis. A skeleton provides a space-efficient one dimensional representation of a 3D volume that runs a long the volume's medial axis, allowing for faster and easier analysis of the biological structures. It is also an intermediate step on the way to creating a graph representation of the neuronal objects. We experimented with various skeletonization algorithms and found out that a subfield "thinning" algorithm [2] fits our purposes best. Thinning starts with points on the object boundary and repeatedly removes ones that do not affect overall topological connectivity. We devised a simple and efficient parallel algorithm for extracting volume skeletons using chromatic scheduling [25, 26] to efficiently schedule the parallel order of which points are considered for deletion doing the thinning process. The details of the skeletonization algorithm are discussed further in Section 7.

3. CNN-based Membrane Detection

The first stage of the pipeline employs a *convolutional neural network* (CNN) [31] to identify cell membranes within the *electron microscopy* (EM) images obtained from the microscope. The output of this stage is a new set of images whose pixels (ranging from 0 to 1) denote the probability that a particular pixel is part of a membrane. A CNN system is typically composed of two components: the *network architecture* which defines layers of perceptrons and their connectivity, and the *computational framework* that executes an architecture's forward propagation over a trained network. In Section 3.1 we describe the *MaxoutNet* CNN architecture that is used in our pipeline to compute membrane probabilities [46]. In Sections 3.2 and 3.3 we describe CPU and GPU-based implementations of our computational framework for executing CNNs.

There are a variety of methods for devising efficient computational frameworks for fully-convolutional neural networks: *e.g.* patch-based sliding windows [8], dilated convolution [69], strided kernels [64], max filtering [32, 72] and filter rarefaction [41]. In our framework, we utilize "maxpooling fragments," which apply the traditional maxpooling operation of CNNs to different offsets of the input image [19, 43]. The resultant images are then recombined to produce the final dense result. The advantage of this approach is that maxpool fragments generate independent matricies that are contiguous in memory, which is leveraged for improved parallelization and memory-locality in both of our CPU and GPU-based implementations.

3.1 Our Network Architecture

The pipeline uses a CNN architecture called *MaxoutNet* that we recently proposed in [46] based on prior work in connectomics [28, 30]. The *MaxoutNet* architecture consists of 4 *ConvPool* layers of alternating convolution/maxpool pairs aggregated with a maxout function, and a final convolution that implements an inner-product. Convolutional layers use 4×4 kernels with stride 1 and either 8 or 32 channels, which combined with stride 2 max-pooling yields a 105×105 field-of-view for each output pixel. For our benchmarks, described in Sections 3.2.4 and 3.3.1, we simplify this network and execute only the last 3 layers of *ConvPool*.

In our pipeline execution, *MaxoutNet* leverages the fact that our input EM images are 3nm resolution (2048×2048) by performing subsampling of the input by executing standard pooling in the first ConvPool layer, while executing "dense" poolings in the next layers. This produces a 2-fold subsampled 1024×1024 output image, and accelerates the network by a factor of 4 without a significant loss of accuracy.

3.2 A Fast CPU framework for CNNs

We introduce *XNN*, a CNN framework implemented in C and inline assembly for Haswell CPUs. The CPU architecture provides support for AVX2 instructions and includes two FMA units, allowing the chip to execute 32 floating point operations per cycle [68]. As a result, a single 2.5 GHz 18-core Haswell chip can theoretically produce 1.44 TFLOPS (compared to state-of-the-art GPU, like NVIDIA's Titan X, that theoretically produces 6TFLOPS). It is, of course, quite challenging to achieve these peak floating point utilizations on either CPU or GPU architectures, and approaching this ideal requires careful engineering techniques that take into account the hardware specifics.

The remainder of this section describes the techniques we applied to optimize *XNN* for the Haswell CPU architecture which enable *XNN* to achieve $70 - 80\%$ FLOP utilization for direct convolutions. Note that FFT-style convolutions can provide additional speedups, as we describe in [4], however, here we focus on low-level performance-engineering of direct convolutions.

3.2.1 Haswell AVX2 SIMD Parallelization

We initially adopted Intel's MLK convolution primitives [10]. However, our results showed that the CPU FLOPS utilization for MKL was only ∼20%, so we reimplemented this critical component with hand-crafted AVX2 assembly.

One of the hurdles that must be overcome to use AVX2 instructions is that one must ensure memory operands have properly aligned addresses. This constraint complicates or even precludes the use of AVX2 instructions in standard convolutional windows that slide over a 2D matrix since the window will contain unaligned memory locations. This led us to an alternate strategy that is based on the observation that a CNN's internal matrices are typically 3D, where the first two dimensions are spatial and the third dimension denotes the channels of the CNN. Instead of vectorizing along the spatial dimension, we vectorize along this channel dimension. This strategy assumes that the number of channels is a multiple of the AVX width, but this constraint is not problematic for CNNs that already use many channels.

Our first implementation was a simple loop over the channel dimension, which did not provide a high CPU floating point utilization, since the GCC compiler could not unroll it optimally. To improve this, we manually unrolled the loops with AVX2 C-inline assembly so that the most inner loop computes 6 convolutions. We interleaved the inner loop's AVX register load, FMA compute, and store vector operations of the 6 convolutions, in a way that maximizes the usage of the two FMA units of the Haswell CPU. In addition, we made the inner and outer loop bounds and counter increments constant at compile-time, to allow maximal usage of GCC's automatic code vectorization and loop unrolling for both inner and outer loops that wrap the manually unrolled convolutions.

3.2.2 Cilk-based Concurrency and Caching

We used Cilk [18, 35], a work-stealing scheduler that is supported by GCC 4.8, to dynamically generate multi-core fine-grained tasks. The key advantage of Cilk is that it provides a "fork-join" primitive that scales well to many cores and retains the serial semantics of the program (*i.e.* the removal of parallel control constructs does not change the behavior of the code on 1-core). Cilk uses a sophisticated combination of per-thread double-ended queues, and a clever work-stealing

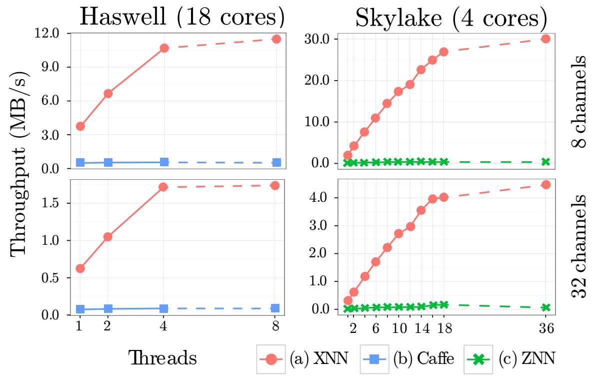

Figure 3: Throughput of the three CPU-based CNN implementations evaluated using 8 and 32-channel MaxoutNet.

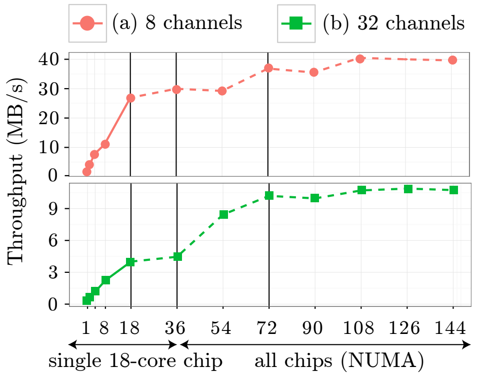

Figure 4: Throughput of the XNN up to 144 hardware threads on our 4 socket system.

algorithm that is provably efficient, and has low memory-contention on shared memory systems.

Our implementation applies the Cilk "fork-join" primitive to all "for-loops" that have no loop iteration dependencies, simplifying our parallelization effort. However, we still require that memory accessed by executing Cilk threads fits into the L3 cache. The L3 cache is much faster than main memory, and having threads working on data that resides in L3 is key to ensuring a high ratio of compute to memory access, which is important for scalability. For example, large matrix operations (*e.g.* convolution) are executed over sub-matrices that each fit into the L3 cache, with a set of threads operating over this sub-matrix before proceeding to the next. We observed that this approach improves scalability by a factor of 3-4x.

3.2.3 Memory Usage

Another important aspect of the implementation is the amount of memory used to compute the forward propagation pass of the CNN network. Since we are only concerned with the performance of forward propagation, we optimized the memory usage significantly by allocating only two large matrix buffers, one for input and one for output, and then reuse (swap) these buffers between layers. As a result, the memory usage is bounded by the largest input/output size of a single CNN layer, which reduces cache trashing and OS paging effects. Note that memory is only allocated at the start, *i.e.* there are no dynamic memory allocations during the computation itself that could introduce contention and bottlenecks.

3.2.4 CPU Benchmarking

Benchmarking was performed on the same shared memory server we use to run our pipeline: a 4-socket machine with Intel *Haswell* 18 core chips and 512 GB of RAM. We also benchmarked a machine equipped with a single 4-core Intel *Skylake* CPU with 64 GB RAM.

We benchmarked our XNN framework against the most efficient known multicore CNN frameworks: the Caffe framework of Yangqing *et al.* [24] and the ZNN framework of Zlateski *et al.* [72]. Throughput was evaluated using 1024×1024 (6 nm) images. We benchmarked all three using a lightweight version of MaxoutNet with 3 fully convolutional ConvPool layers (field-of-view = 53) because the subsampling required for the fourth layer (3 nm data) is available only in our XNN framework. We benchmark for both 8 and 32 channels. To ensure best performance, we compiled with the Intel C++ Compiler (ICC) version 16.0.0, optimized for maximum speed (`-O3`), linked against Intel Threading Building Blocks (TBB) version 4.4 (`libtbbmalloc`, `libtbbmalloc_proxy`), and Intel Math Kernel Library (MKL) version 11.3 with enabled FFT caching, and single-precision floating-point arithmetic.

The results of this benchmarking are presented in Figure 3. In this execution, we perform a scalability test on a single 18-core chip: an input is set to a fixed size and we increase the number of threads. By binding multi-threading to one chip we disable the NUMA-effects for this test (expensive inter-chip communication). It is evident that our XNN CPU framework exhibits substantial throughput improvement over Caffe and ZNN. First, one can see that there is a significant difference in single-threaded performance, which is the effect of the hand-crafted assembly of XNN that utilizes the two FMA AVX2 units of Haswell CPU. Second, the scalability of XNN is almost linear, 15x over a single thread on 18 cores. This is achieved by constraining active threads to operate on memory sets that fit into the L3 cache while using Cilk to manage short-living jobs.

Figure 4 presents the results of XNN executed as a single instance (fixed input) on all 72 cores across 4 sockets, running up to 144 threads to utilize hyperthreading. As can be seen, for the 32 channel execution (green) linear scalability is evident up to 18 cores and plateaus all the way to 36 cores, as hyperthreading provides only 10-20% boost in performance. Increasing threads from 36 to 72 utilizes additional cores to again provide pseudo-linear scalability. Beyond 72 cores, additional threads leverage hyperthreading for a minimal further improvement. For 8 channels (red), the smaller network size generates low latency Cilk tasks that introduce more pressure on the software implementation. There is still linear scalability up to 18 threads, but only small improvement is evident beyond 36 cores. This is because the overhead of threading (NUMA side effects) is substantial compared to the performance benefit from adding additional cores.

In our pipeline execution, we overcome these NUMA overheads by combining multi-processing with multi-threading. Specifically, we execute multiple instances of XNN (each bound to a specific chip) with a empirically optimized number of threads, resulting in a 2.8x speedup. Generating these instances increases the RAM requirements (not problematic for our 512 GB RAM system).

3.3 A Fast GPU Framework for CNNs

In this section we present *gpuZNN*, a GPU-based CNN framework that improves upon the previously published ZNN [72] to optimize forward propagation throughput.

The SIMT programming model and limited amount of memory available on a GPU require a different implementation than our XNN CPU framework. To saturate the GPU, we process multiple input images simultaneously. We also leverage the fact that all sub-samplings of the “dense” computation have equal size, which is equivalent to processing multiple inputs of a layer at the same time.

Most of our convolutional primitives use cuDNN’s low level implementations [53]. To address the large memory overhead of some primitives, we allow the computation to be split into multiple stages, computing a subset of results at the time. This is accomplished by computing a subset of input batches and/or subset of output images (feature maps). This can reduce the parallelization potential, but reducing the memory overhead can allow for using a computationally cheaper primitive. We also implement optimized FFT-based convolutional primitives with very low memory overhead. Batched 1D FFTs and memory reshuffling was used to efficiently utilize the GPU.

Sub-sampling layers (maxpooling and maxout) were implemented using cuDNN’s maxpooling primitive. We minimized the number of calls to the primitives such that each primitive performs more computation and can thus saturate the GPU. To allow this, we implicitly gathered the inputs and scattered the results for each call. Implicit gather/scatter is performed by providing the shapes and strides for all inputs/outputs from which the memory location of each element can be computed.

3.3.1 GPU CNN Benchmarking

We benchmarked our gpuZNN implementation relative to the fastest available GPU frameworks [7] on an NVIDIA GeForce GTX Titan X (3072 cores, 1.0 GHz, 12 GB memory). Specifically, we benchmarked the MaxoutNet network with both 8 and 32 channels for three GPU frameworks: (a)

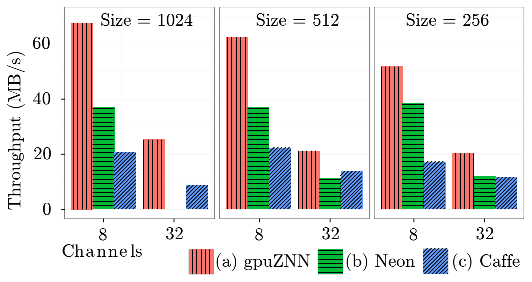

Figure 5: Throughput of GPU-based CNNs using 8 and 32-channel MaxoutNet architectures.

Method	**Type**	**8-channel MaxoutNet Throughput (MB/s)**	**32-channel MaxoutNet Throughput (MB/s)**
XNN	CPU (72-core)	111.1	16.67
gpuZNN	GPU (Titan X)	67.61	25.28
Neon	GPU (Titan X)	37.06	(exceeds memory)
XNN	CPU (4-core)	11.49	1.74

Figure 6: A comparison of CNN throughput for the best-performing CPU and GPU-based implementations using the MaxoutNet architecture.

gpuZNN, our modified version of ZNN [72]; Caffe [24], compiled with cuDNN v4; and Neon by Nervana [50], which outperformed Caffe and other alternatives on several recently published benchmarks [7]. Throughput was evaluated using 6 nm images presented with different sizes $(1024^2, 512^2, 256^2)$ in batches of size 32 (a constraint of Neon's GPU backend), while using max-pool layers with a stride of 1. This eliminates "sub-sampling" effects of max-pooling and simulates the same number of floating point computations as a "dense" inference (not supported by Caffe or Neon).

The results of this benchmarking are presented in Figure 5. It is evident that gpuZNN provides the best throughput on both 8 and 32-channels for all image sizes: 1.8x and 2.9x faster than Neon and Caffe for 8 channels respectively, and 1.8x faster than Caffe for 32 channels. The 32-channel MaxoutNet could not be benchmarked on Neon as the memory usage exceeded the 12 GB available on the Titan X. The main reason for gpuZNN's improvement is its low-memory overhead FFTs that allows it to aggregate large sub-computation units and minimize the amount of calls to primitives of the GPU (each such call involves host-device memory transfers on the PCI-E bus). We note that the actual speed difference between 32 and 8 channels for gpuZNN is 2.6x (and not 32/8=4), which is an effect of the overheads of PCI-E host-device memory transfers that become more significant as the network becomes smaller.

3.3.2 A Comparison of CPUs and GPUs

Figure 6 shows the maximum throughput that we could achieve for XNN, gpuZNN and Neon for 8 and 32 channel CNNs. For 72-cores, we found that the best combination of multi-threading and multi-processing is 8 instances (2 per chip), where each is using 18 threads. One can see that for a network with 32 channels, the GPU is faster than the CPU. However, this is not the case for 8 channels, where the CPU is twice as fast as the GPU. This is because an 8 channel CNN implies less compute for the GPU and makes the PCI-E bus memory transfers more expensive.

4. Watershed

The next stage in the connectomics pipeline involves producing an over-segmentation of neuron candidates from the CNN membrane probability output. Over-segmentation ensures that no supervoxel straddles more than one true segment and is later resolved by agglomeration. Specifically, we apply a custom 3D implementation of the popular linear-time watershed algorithm that yields an 11x speed-up on previous implementations [23]. In general, the key idea is to take into account that probability maps are allowed to take only 8-bit values, which allows us to implement the priority queue as a set of 2^8 FIFO queues. The FIFO queues are implemented using simple arrays with an amortized cost of $O(1)$ for both *push* and *pop* operations. As a result, queue access are cache and pre–fetcher friendly. In addition, we reduce the memory overhead by an order of magnitude compared to OpenCV, which is an important factor for the watershed algorithm that exhibits low ratio of compute-to-memory.

5. Agglomeration

The agglomeration stage refines the over-segmentation generated by the watershed algorithm by merging regions it identifies as belonging to the same neuronal volume. Our pipeline's agglomeration stage is based on Neuroproof [54], a state of the art tool for graph-based image segmentation. Serial optimizations and shared memory parallelization techniques were employed to enhance the performance of Neuroproof on our shared memory multicore system, with the results summarized in Figure 7.

5.1 Regional Adjacency Graphs

Agglomeration is performed on a higher-level representation of the labeled volume called a *regional adjacency graph (RAG)*. A RAG is a graph with a vertex for each distinctly labeled region, and an edge connecting each pair of adjacent regions. Each vertex within the RAG maintains several regional features such as histograms of membrane-probabilities and multiple image moments. These features are typically associative with respect to the agglomeration's merge operation, and thus enable the agglomeration procedure to operate directly on the RAG instead of the much larger input volume.

The initial construction of the RAG requires analyzing the entire input volume, and is thus the most expensive component of this stage. Prior to our performance engineering efforts, RAG construction required 124 seconds on a 1024x1024x100 input stack and was responsible for 70% of the total time spent in the stage.

5.1.1 Serial Optimizations

Several optimizations were performed to reduce the total work performed during RAG construction. These optimizations fell into three categories: optimizations to feature computation, eliminating layers of indirection, and improving memory locality.

Feature computation was optimized by providing a batched version the feature computation function, so that it could process multiple pixels at once. In addition, a significant opportunity for optimization was uncovered in the portion of the program that computed image moment features. The ith image moment feature computes the sum of p^i for all pixels p in a region. The standard library `pow` function was used to compute these features for 4 moments, and was to blame for over 50% of the runtime in RAG construction. We modified the function computing moment features to perform iterative multiplication, and reduce the per-pixel-cost to 3 floating point multiplications. These modifications to feature computation resulted in a 2.5x improvement in serial runtime for RAG construction, resulting in a total runtime of 50 seconds.

Another class of changes involved eliminating unnecessary layers of indirection in matrix and hashtable access. The labeled input volume was loaded into an OpenCV `Mat` object and was accessed through a provided interface. We modified all accesses to operate directly on matrices underlying arrays to eliminate unnecessary indirection to the OpenCV library. This resulted in a further 1.6x improvement in RAG construction time, reducing the total runtime of RAG construction to 30 seconds.

The last major optimization was the design of a divide-and-conquer RAG construction algorithm that takes advantage of the associativity of region features. Given a 3D volume, the largest dimension of the volume was divided in half and the RAG was computed for each half recursively. These RAGs were then merged with their regional features combined according to each of their associative update rules. The base-case of the recursion was coarsened so that the total volume would fit into an L2 cache of approximately 256 KB.

5.1.2 Parallelization

The divide-and-conquer implementation of RAG construction was especially amenable to efficient parallelization. In particular, the method of merging RAGs can be performed

	Baseline (s)	**Fast 1-core** (s)	**Fast 4-core** (s)
Input/Output	11.2	10.5	6.7
RAG Construction	123.2	24.2	6.8
RAG Agglomeration	42.5	40.2	13.4
Total	176.9	75.0	27.0

Figure 7: Performance improvements to agglomeration stage.

efficiently by thinking of the algorithm as a parallel merge sort with an augmented merge operation [9, Ch 27.3]. In other words, the RAG construction algorithm can be parallelized in the same fashion as merge sort by considering a RAG to be a sorted edge-list. We represent regions with a self-edge and store the region's computed features as meta data. After performing a merge of two RAGs' edge lists, the edge list is scanned in parallel to identify and merge duplicated edges. These steps can all be performed in parallel using logarithmic-depth algorithms. On 4 cores, the parallel RAG construction algorithm achieves speedup of 3.5x, reducing the time of RAG construction to 6.8s.

After RAG construction, RAG agglomeration analyzes edges to determine which node pairs to merge. We observed that most of the work was performed by a random-forest classification library within OpenCV that computes edge weights. To parallelize RAG agglomeration, we defer the computation of edge weights, and mark effected nodes and edges as *dirty* [54]. A dequeued edge is ignored if it is dirty or incident to a dirty node. When the priority queue is empty, the edge weights are all computed in parallel as a batch, and reinserted into the queue. This strategy resulted in 3x speedup for agglomeration, reducing its runtime from 40.2 to 13.4 seconds.

6. Merging

In this section we describe how our pipeline merges all of the per-block segmentations produced by the earlier stages to obtain a segmentation for the entire volume. Prior work by Plaza and Berg [56] describe a method of performing such a merge operation by generating overlaps between adjacent blocks and identifying pairs of segments to merge based on a set of complex hand-crafted heuristics. As reported in [56], this approach is made difficult by the substantial number of edge cases that must be handled correctly to obtain a high quality segmentation. Our approach is of similar spirit, but simplifies the problem of identifying merge-pairs by treating it as a special case of agglomeration.

Our inter-block merge procedure operates in two phases that each perform an agglomeration over the block boundaries. These phases reuse the optimized agglomeration algorithms described in Section 5, but use a random forest classifier that is trained specifically on inter-block segmentations. Segments identified as merge-pairs are then recorded on-disk, and processed into equivalence classes using a disjoint-set data structure. The performance of the merge stage was further optimized by tuning the ordering and parallelism degree to ensure that the machine's caches and disks were used efficiently.

6.1 Merge Pair Decisions.

Merge pairs are determined for all adjacent blocks in the volume in two phases. First, the segments in the block boundary are agglomerated using our parallel agglomeration algorithm. This phase is conservative, to avoid incorrect mergers, but since neuronal objects are large (spanning many blocks) most merge-pairs are found in this phase. In the second phase, the algorithm generates a synthetic ground-truth by re-executing the watershed and agglomeration stages on the block boundaries. This synthetic ground-truth allows for the remaining merge-pairs to be identified via a straightforward optimization procedure. This procedure merges a pair and computes the associated VI score relative to the synthetic ground-truth. If accuracy decreases, it aborts the merge and tries a different pair.

6.2 Combining and Relabelling.

After merge-pairs are identified for pairs of adjacent blocks, the next step is to actually relabel the volume so that each distinct object has a distinct label.

To identify equivalent labels, we operate on the lists of merge-pairs generated for pairs of adjacent blocks and use a disjoint-set data structure to form equivalence classes of segment identifiers (i.e. where two segments are equivalent if they appear as a merge-pair). This process is straightforward since all merge-pairs are present on the machine's local disk, and the total size of all merge-pairs is miniscule in-comparison to other data sets in the pipeline. Finally, the algorithm executes a parallel pass over all segmented blocks and relabels them based on their equivalence class. This relabeling step is I/O bound, but this is mitigated by ordering blocks to be relabeled such that all of the machine's disks are well utilized.

7. Skeletonization

A skeleton is a one dimensional space-efficient representation of a 3D volume that runs along the segments's medial axis. There are a vast number of algorithms to compute skeletons for a variety of purposes in computer graphics [60]. We found that for connectomics a thinning algorithm would be most fitting because it is the fastest algorithm that still preserves the connectivity of the objects. Their algorithms remove all so-called *simple points*, voxels whose removal does not change the topology. Checking if a point is simple or not is non-trivial, especially when multiple voxels are deleted in parallel.

Our parallel thinning algorithm is implemented as a chromatically scheduled dynamic data-graph computation [25, 26]. A grid graph represents the labeled 3D volume and a statically-computed distance-2 coloring is used to identify sets of voxels that may be processed in parallel.

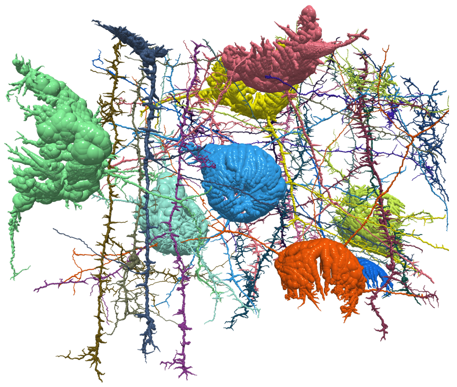
Figure 8: Skeletonization of 20 objects from the Kasthuri *et al.* dataset (as one large block) [28] (473 GB $\approx$ 100,000 cubic microns of cortex).

Initially updates are scheduled on all surface points, which check if a given point satisfies the *simple point* criterion based on their 26-neighborhood. An update that deletes a point schedules an update dynamically for all non-deleted neighbors. Since there are only 2^{26} possible 26-neighborhoods, the simple point criterion can be precomputed. Our simple point criterion is based upon the 38 templates of [62] and ensures local connectedness. Since our algorithm uses chromatic scheduling, it is equivalent to a standard 8-subfield thinning algorithm and thus provably preserves the topology of the objects [2].

After skeletonization, we transform the 3D segments into tree graphs based on 26-connectivity. If cycles exist in the graph they are broken up arbitrarily, however, large cycles are detected and are reported for further analysis and error detection. After performing a one-pass pruning step the trees are outputted as .swc files and visualized using the neuTube software [17]. Our pipeline currently generates skeletons per-block, but the same algorithm can be applied (with additional engineering effort) to the entire volume (Figure 8).

8. Pipeline Performance

The earlier portions of the paper focused on improving the performance and scalability of individual machine learning algorithms that consume a large fraction of the computation in existing connectomics systems [56, 58]. In this section, we focus on the combined performance of our pipeline elements. The main workhorse for our experiments is the previously described 4-socket shared memory machine equipped with 4 18-core Intel Xeon CPUs with 512 GB of RAM running Ubuntu 14.04.

We tested our pipeline on a 473 GB (3x3x30 nm resolution) electron microscopy dataset of mouse somatosensory cortex [27], containing 1850 pre-aligned 16,384x16,384 (256 MB) 2D EM brain scans.

The total time to process the entire 473 GB EM dataset was **1.7 hours**, implying a throughput of **3.6 hours / TB**.

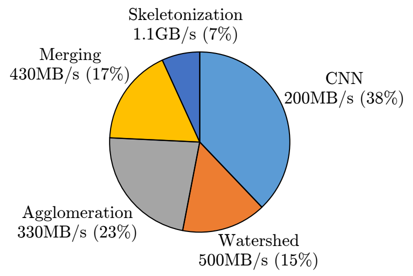

Figure 9: Proportion of the execution time spent on each stage.

Figure 9 provides a breakdown illustrating each stage's contribution to the total execution time. Notably, we have managed to improve the machine-learning-based segmentation component to be on par with the other stages, which alleviates a previously constraining bottleneck in the throughput of connectomics pipelines [56].

To analyze the efficiency of using a 72-core system, we compared the throughput of our pipeline on a single core against our throughput on all 72 cores. On a single core, our pipeline processes data at a speed of 1.11s / MB, compared to 0.013s / MB on 72 cores. This implies that our pipeline achieves an 85x factor speedup, which is greater than 72 since hyper-threading provides our machine with 144 hardware threads.

The number of concurrent instances and the degree-of-parallelism used within each instance varies depending on the pipeline stage. To achieve maximum throughput in the CNN phase, we execute 8 instances of XNN (2 per chip), where each instance uses 18 threads. This eliminates any NUMA side-effects that could introduce contention and bottlenecks. In the watershed phase, we simply spawn 144 instances since the implementation we have is a fast sequential code with low memory consumption (each instance <1 GB, and total memory is 512 GB). In the agglomeration phase, we execute 36 instances (9 per chip) of parallel NeuroProof, where each instance uses 4 threads. A similar scheme is applied to merging and skeletonization.

8.1 Reconstruction Accuracy

To evaluate the reconstruction accuracy of our pipeline we followed the benchmark described by Kaynig *et al.* [29]. We partitioned a total of 150 images (a region of the data called AC3 in [29]) at 2048x2048 and 3 nm resolution into three sets; 10 for training, 65 for validation and 75 for testing. Ground truth neuron segmentation was provided by expert neurobiologist annotators.

Our measure of segmentation accuracy is *variation of information* (VI) [45], that captures the statistical difference between two segmentations. In a typical segmentation there are split errors (neurons split erroneously) and merge errors (neurons merged erroneously), and the VI is an indicator to the extent of such errors in the data set. We benchmarked our 8-channel MaxoutNet CNN architecture (described in Section 3.1) and the state-of-the-art reconstruction of RhoAna described by Kaynig *et al.* [29] using the same evaluation benchmark on our test dataset described above. We note that our comparisons with the RhoAna-based pipeline described in [29] are only with respect to accuracy, since it was not extensively engineered for performance (it requires several weeks and thousands of machines to process a terabyte).

On the AC3 dataset, our pipeline achieved an accuracy score of VI = 1.6483 which is, surprisingly, an improvement over the VI = 1.99 score reported for RhoAna (lower VI means improved reconstruction). This comparison indicates that our pipeline's efficiency does not inherently come at the expense of reconstruction quality, albeit we caution that reconstruction accuracy is a complex metric and that accuracy comparisons performed on a single data set should be taken with a grain of salt.

9. Lessons Learned

In this section we discuss some of the general lessons learnt from our effort to reduce a high-throughput, big-data problem to the domain of a single shared-memory multicore system. Some of these may be known to the reader, but others were counter-intuitive, at least to us, and therefore worth explicit mention.

9.1 Scalable software saves memory

Writing multicore code is generally hard – it is far simpler to write a single-threaded program that avoids concurrency "nightmares," and scale horizontally by launching one instance per core. While this may work for simple tasks, we observed that this approach can be seriously flawed for complex software. In the case of both the agglomeration and membrane-detection stages of our pipeline, the large memory footprint of each instance caused the OS to frequently swap between memory and disk, thus increasing I/O bandwidth requirements. Moreover, performance degraded due to L3 cache pollution caused by multiple instances populating the shared L3 cache with disjoint memory accesses. By engineering our software to scale to multiple cores, we obtained a degree of freedom in our pipeline design that allowed us to run a smaller number of instances, where each uses multiple cores to operate on data fitting into L3 cache. For example, our pipeline runs 8 instances (two per socket) of the CNN code for membrane detection, where each instance uses 9 cores. This enabled efficient use of the caches on each socket and eliminated the need to handle complex NUMA overheads [5, 13, 14, 16, 47].

9.2 Disk I/O is scalable on a single machine

Once computation is sufficiently optimized, disk-to-memory I/O for a single machine can become a bottleneck. This is often part of the motivation to migrate computation to a cluster of machines, and is a problem we encountered in the watershed and agglomeration pipeline phases (where the disk I/O 100 MB/s read and 200 MB/s write were reached). Instead, we resolved this bottleneck by horizontally scaling our disk drives – data was sharded across a set of 5 drives that were installed on the same machine, yielding 500 MB/s read and 1000 MB/s write for the system. Adding more disks to improve I/O bandwidth in this manner is far cheaper and simpler than migrating to a distributed cluster.

9.3 Dynamic multithreading simplifies cache-aware parallelization

A potential criticism of a multicore-centric approach is that it simply shifts from one set of programming complexities (networking, failure detection and recovery schemes on a distributed system) to the unique challenges posed by shared-memory concurrency. Our experience is that writing cache-efficient multicore code can be simple, if using the right tools. Our multicore programs are written using GCC-Cilk which employs an efficient work-stealing scheduler, and allows parallelism to be expressed while retaining the semantics and many performance properties of serial code. As a result, our techniques to make our programs cache and pre-fetcher friendly largely mirror those used in serial code. This allowed us to achieve good cache efficiency without needing to write complex cache-aware scheduling logic.

9.4 A GPU is not 100X faster than a CPU

It is widely believed in the machine learning community that a single GPU can perform orders of magnitude faster than a single CPU. This is supported empirically by popular machine learning packages – if one compares Caffe (bound to Intel MKL and Nvidia cuDNN libraries) execution times on GPU versus CPU, the speed-up is approximately 50 to 100-fold. Following our multicore performance engineering efforts, we observe substantially different results. On a single 18-core 2.4 GHz Haswell chip, our XNN execution is only 2-3x slower than both gpuZNN and the previous fastest CNN framework (Neon [50]) executed on a Titan X GPU. Moreover, XNN execution on a commodity 4-core Skylake chip (a standard desktop/laptop processor) was only 4-6x slower than this top-end GPU. These observations are consistent with the findings reported in [33], but here we support the claim on a modern CPU.

9.5 Multicores enable new efficient algorithms that are expensive on clusters

A common belief of Hadoop/Spark programmers is that coding for a distributed cluster is simpler than for a multi-core, and therefore, shifting software to multicores is not worth the effort. However, our experience shows that this not true. A key point that needs to be emphasized is that coding for clusters is simple as long as the parallelization of the problem is ”trivial”. In other words, as long as one can break the problem into small parts that can be processed without communicating, then it is easy to do map-reduce style programming. However, if this is simple for the cluster, then it will also be simple for the multicore. Moreover, cluster-based algorithms exhibit high network latencies, which forces them to avoid communication between machines. As a result, programmers are constrained to coding patterns that are ”trivially parallelizable” in the context of clusters, since it is the only way to avoid high network costs and get scalability. However, this is not the case for multicores: inter-core communication is orders of magnitude less expensive than network cluster communication, which allows programmers to design more sophisticated approaches that involve complex communication patterns.

10. Conclusion

We presented a “proof of concept” connectomics pipeline that can extract a full skeletonization from an EM image stack in less than 4 hours on a commodity multicore machine and with a VI accuracy on par or better than any existing system. This has the potential to move the connectomics problem, a big-data research problem in the natural sciences, from the realm of distributed data and warehouse storage, to that of on-demand processing in labs across the world. Given current trends in multicore CPU and GPU architectures, we venture to predict that a single socket machine, perhaps with a single attached GPU card, will be able to provide a solution for many connectomics pipelines around the world.

The domain-specific results we report on Kashturi dataset [28], constitute a case-study on the role of multicore performance engineering in large-scale image processing pipelines. The lessons learnt from our experiences designing this system are of great import to those in the multicore and cluster computing communities. Through careful performance engineering, we show that a single commodity multicore machine can not only compete, but significantly outperform, existing CPU- and GPU- based clusters solving the same problem. This, of course, is not intended to advocate that all “big data” problems are best solved on a single multicore, but rather to serve as a reminder of the importance and dramatic benefits that can be obtained through multicore performance engineering.

Acknowledgments

Support is gratefully acknowledged from the National Science Foundation (NSF) under grants IIS-1447786 and CCF-1563880, and the Intelligence Advanced Research Projects Activity (IARPA) under grant 138076-5093555. In addition, we would like to thank the MIT SuperTech group for helpful discussions. In particular, Quan Nguyen, William Hasenplaugh, Bradley Kuszmaul, and Charles Leiserson. Finally, we would like to thank the Harvard connectomics group for reviewing our work and providing useful feedback. Specifically, Daniel Berger, Adi (Suissa) Peleg, Thouis R. Jones, Hanspeter Pfister, David Cox, and Jeff Lichtman.

References

[1] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik. Contour detection and hierarchical image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(5): 898–916, 2011.

[2] G. Bertrand and Z. Aktouf. Three-dimensional thinning algorithm using subfields. volume 2356, pages 113–124, 1995. doi: 10.1117/12.198601.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi: 10.1145/209936.209958. URL http://doi.acm.org/10.1145/209936.209958.

[4] D. Budden, A. Matveev, S. Santurkar, S. R. Chaudhuri, and N. Shavit. Deep tensor convolution on multicores. *CoRR*, abs/1611.06565, 2016. URL http://arxiv.org/abs/1611.06565.

[5] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442532. URL http://doi.acm.org/10.1145/2442516.2442532.

[6] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 571–582, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685094.

[7] S. Chintala. Convnet benchmarks. https://github.com/soumith/convnet-benchmarks.

[8] D. Ciresan, A. Giusti, L. M. Gambardella, and J. Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In *Advances in neural information processing systems*, pages 2843–2851, 2012.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[10] I. Corporation. Intel math kernel library, 2016. URL https://en.wikipedia.org/wiki/Math_Kernel_Library.

[11] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 4:1–4:16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901323. URL http://doi.acm.org/10.1145/2901318.2901323.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1): 107–113, Jan. 2008.

[13] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining numa locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989502. URL http://doi.acm.org/10.1145/1989493.1989502.

[14] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145848. URL http://doi.acm.org/10.1145/2145816.2145848.

[15] A. Eberle, S. Mikula, R. Schalek, J. Lichtman, M. K. TATE, and D. Zeidler. High-resolution, high-throughput imaging with a multibeam scanning electron microscope. *Journal of microscopy*, 259(2):114–120, 2015.

[16] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. Snzi: Scalable nonzero indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 13–22, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5. doi: 10.1145/1281100.1281106. URL http://doi.acm.org/10.1145/1281100.1281106.

[17] L. Feng, T. Zhao, and J. Kim. neuTube 1.0: a New Design for Efficient Neuron Reconstruction Software Based on the SWC Format. *eneuro*, Jan. 2015. ISSN 2373-2822.

[18] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90. ACM, 2009.

[19] A. Giusti, D. C. Ciresan, J. Masci, L. M. Gambardella, and J. Schmidhuber. Fast image scanning with deep max-pooling convolutional neural networks. In *ICIP*, page in press, 2013.

[20] Google. Google cloud platform blog: Google supercharges machine learning tasks with tpu custom chip, 2016. URL https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[21] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 27–40, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2749472. URL http://doi.acm.org/10.1145/2749469.2749472.

[22] IBM. Introducing a brain-inspired computer, 2016. URL http://www.research.ibm.com/articles/brain-chip.shtml.

[23] itseez. Open source computer vision library, 2016. URL http://opencv.org/.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[25] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms*

and Architectures, SPAA '14, pages 154–165, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2821-0. doi: 10.1145/2612669.2612673. URL http://doi.acm.org/10.1145/2612669.2612673.

[26] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. *ACM Trans. Parallel Comput.*, 3(1):2:1–2:31, July 2016. ISSN 2329-4949. doi: 10.1145/2896850. URL http://doi.acm.org/10.1145/2896850.

[27] N. Kasthuri, K. Hayworth, J. C. Tapia, R. Schalek, S. Nundy, and J. W. Lichtman. The brain on tape: Imaging an ultra-thin section library (utsl). In *Soc. Neurosci. Abstr*, 2009.

[28] N. Kasthuri, K. J. Hayworth, D. R. Berger, R. L. Schalek, J. A. Conchello, S. Knowles-Barley, D. Lee, A. Vázquez-Reina, V. Kaynig, T. R. Jones, et al. Saturated reconstruction of a volume of neocortex. *Cell*, 162(3):648–661, 2015.

[29] V. Kaynig, A. Vazquez-Reina, S. Knowles-Barley, M. Roberts, T. R. Jones, N. Kasthuri, E. Miller, J. Lichtman, and H. Pfister. Large-scale automatic reconstruction of neuronal processes from electron microscopy images. *Medical image analysis*, 22(1):77–88, 2015.

[30] S. Knowles-Barley. Rhoana git. https://github.com/Rhoana/membrane_cnn/tree/master/maxout.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[32] K. Lee, A. Zlateski, V. Ashwin, and H. S. Seung. Recursive Training of 2D-3D Convolutional Networks for Neuronal Boundary Prediction. In *Advances in Neural Information Processing Systems*, pages 3559–3567, 2015.

[33] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1816021. URL http://doi.acm.org/10.1145/1815961.1816021.

[34] W.-C. A. Lee, V. Bonin, M. Reed, B. J. Graham, G. Hood, K. Glattfelder, and R. C. Reid. Anatomy and function of an excitatory network in the visual cortex. *Nature*, 532(7599): 370–374, 2016.

[35] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[36] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. Malt: Distributed data-parallelism for existing ML applications. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 3:1–3:16, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741965. URL http://doi.acm.org/10.1145/2741948.2741965.

[37] Y. Li and Z. Lan. Exploit failure prediction for adaptive fault-tolerance in cluster computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 8–pp. IEEE, 2006.

[38] J. W. Lichtman and W. Denk. The big and the small: challenges of imaging the brains circuits. *Science*, 334(6056): 618–623, 2011.

[39] J. W. Lichtman and J. R. Sanes. Ome sweet ome: what can the genome tell us about the connectome? *Current Opinion in Neurobiology*, 18(3):346–353, June 2008. ISSN 0959-4388. doi: 10.1016/j.conb.2008.08.010. URL http://www.sciencedirect.com/science/article/pii/S0959438808000834.

[40] J. W. Lichtman, H. Pfister, and N. Shavit. The big data challenges of connectomics. *Nature neuroscience*, 17(11): 1448–1454, 2014.

[41] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[42] J. Maitin-Shepard, V. Jain, M. Januszewski, P. Li, J. Kornfeld, J. Buhmann, and P. Abbeel. Combinatorial energy learning for image segmentation. *arXiv preprint arXiv:1506.04304*, 2015.

[43] J. Masci, A. Giusti, D. Ciresan, G. Fricout, and J. Schmidhuber. A fast learning algorithm for image segmentation with max-pooling convolutional networks. In *Image Processing (ICIP), 2013 20th IEEE International Conference on*, pages 2713–2717. IEEE, 2013.

[44] M. Meilă. Comparing clusteringsan information based distance. *Journal of multivariate analysis*, 98(5):873–895, 2007.

[45] M. Meilă. Comparing clusteringsan information based distance. *Journal of multivariate analysis*, 98(5):873–895, 2007.

[46] Y. Meirovitch, A. Matveev, H. Saribekyan, D. Budden, D. Rolnick, G. Odor, S. K.-B. T. R. Jones, H. Pfister, J. W. Lichtman, and N. Shavit. A Multi-Pass Approach to Large-Scale Connectomics. *ArXiv e-prints*, Dec. 2016.

[47] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. URL http://doi.acm.org/10.1145/103727.103729.

[48] A. N. Moga, B. Cramariuc, and M. Gabbouj. Parallel watershed transformation algorithms for image segmentation. *Parallel Comput.*, 24(14):1981–2001, Dec. 1998. ISSN 0167-8191. doi: 10.1016/S0167-8191(98)00085-4. URL http://dx.doi.org/10.1016/S0167-8191(98)00085-4.

[49] J. L. Morgan, D. R. Berger, A. W. Wetzel, and J. W. Lichtman. The fuzzy logic of network connectivity in mouse visual thalamus. *Cell*, 165(1):192–206, 2016.

[50] Nervana. Neon. https://github.com/NervanaSystems/neon.

[51] J. Nunez-Iglesias, R. Kennedy, T. Parag, J. Shi, and D. B. Chklovskii. Machine learning of hierarchical clustering to segment 2D and 3D images. *PloS one*, 8(8):e71715, 2013.

[52] J. Nunez-Iglesias, R. Kennedy, S. M. Plaza, A. Chakraborty, and W. T. Katz. Graph-based active learning of agglomeration (gala): a python library to segment 2d and 3d neuroimages. *Frontiers in neuroinformatics*, 8, 2014.

[53] NVIDIA. Nvidia cudnn - gpu accelerated deep learning, 2016. URL https://developer.nvidia.com/cudnn.

[54] T. Parag, A. Chakrobarty, and S. Plaza. A context-aware delayed agglomeration framework for em segmentation. *CoRR*, 2014.

[55] T. Parag, A. Chakraborty, S. Plaza, and L. Scheffer. A context-aware delayed agglomeration framework for electron microscopy segmentation. *PloS one*, 10(5):e0125825, 2015.

[56] S. M. Plaza and S. E. Berg. Large-scale electron microscopy image segmentation in spark. *arXiv preprint arXiv:1604.00385*, 2016.

[57] S. Ramón and S. Cajal. *Textura del Sistema Nervioso del Hombre y de los Vertebrados*, volume 2. Madrid Nicolas Moya, 1904.

[58] W. R. G. Roncal, D. M. Kleissas, J. T. Vogelstein, P. Manavalan, K. Lillaney, M. Pekala, R. Burns, R. J. Vogelstein, C. E. Priebe, M. A. Chevillet, et al. An automated images-to-graphs framework for high resolution connectomics. *Frontiers in neuroinformatics*, 9, 2015.

[59] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015*, pages 234–241. Springer, 2015.

[60] P. K. Saha, G. Borgefors, and G. Sanniti di Baja. A survey on skeletonization algorithms and their applications. *Pattern Recognition Letters*, 76:3–12, June 2016. ISSN 0167-8655. doi: 10.1016/j.patrec.2015.04.006.

[61] S. Seung. *Connectome: How the brain's wiring makes us who we are*. Houghton Mifflin Harcourt, 2012.

[62] F. She, R. Chen, W. Gao, P. Hodgson, L. Kong, and H. Hong. Improved 3d Thinning Algorithms for Skeleton Extraction. In *Digital Image Computing: Techniques and Applications, 2009. DICTA '09.*, pages 14–18, Dec. 2009. doi: 10.1109/DICTA.2009.13.

[63] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[64] F. Tschopp. Efficient convolutional neural networks for pixelwise classification on heterogeneous hardware systems. *arXiv preprint arXiv:1509.03371*, 2015.

[65] B. Vision and L. Center. Caffe deep learning framework. http://caffe.berkeleyvision.org/.

[66] J. Vogelstein. Machine intelligence from cortical networks (microns), 2016. URL https://www.iarpa.gov/index.php/research-programs/microns.

[67] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 314(1165):1–340, 1986. ISSN 0080-4622. doi: 10.1098/rstb.1986.0056. URL http://rstb.royalsocietypublishing.org/content/314/1165/1.

[68] Wiki. Advanced vector extensions, 2016. URL https://en.wikipedia.org/wiki/Advanced_Vector_Extensions.

[69] F. Yu and V. Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[70] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[71] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[72] A. Zlateski, K. Lee, and H. S. Seung. ZNN-A fast and scalable algorithm for training 3D convolutional networks on multicore and many-core shared memory machines. *arXiv preprint arXiv:1510.06706*, 2015.