Scalable Multimedia Content Analysis on Parallel Platforms Using Python

EKATERINA GONINA, University of California, Berkeley
GERALD FRIEDLAND, International Computer Science Institute and University of California, Berkeley
ERIC BATTENBERG, PENPORN KOANANTAKOOL, MICHAEL DRISCOLL, EVANGELOS GEORGANAS, and KURT KEUTZER, University of California, Berkeley

In this new era dominated by consumer-produced media there is a high demand for web-scalable solutions to multimedia content analysis. A compelling approach to making applications scalable is to explicitly map their computation onto parallel platforms. However, developing efficient parallel implementations and fully utilizing the available resources remains a challenge due to the increased code complexity, limited portability and required low-level knowledge of the underlying hardware. In this article, we present PyCASP, a Python-based framework that automatically maps computation onto parallel platforms from Python application code to a variety of parallel platforms. PyCASP is designed using a systematic, pattern-oriented approach to offer a single software development environment for multimedia content analysis applications. Using PyCASP, applications can be prototyped in a couple hundred lines of Python code and automatically scale to modern parallel processors. Applications written with PyCASP are portable to a variety of parallel platforms and efficiently scale from a single desktop Graphics Processing Unit (GPU) to an entire cluster with a small change to application code. To illustrate our approach, we present three multimedia content analysis applications that use our framework: a state-of-the-art speaker diarization application, a content-based music recommendation system based on the Million Song Dataset, and a video event detection system for consumer-produced videos. We show that across this wide range of applications, our approach achieves the goal of automatic portability and scalability while at the same time allowing easy prototyping in a high-level language and efficient performance of low-level optimized code.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.m [**Software Engineering**]: Miscellaneous—*Rapid prototyping*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms: Experimentation, Performance

This research was supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. It was also partially supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract number D11PC20066. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of IARPA, DOI/NBC, or the U.S. Government.
Authors' addresses: E. Gonina, Soda Hall, 573D, Berkeley, CA, 94720; email: egonina@eecs.berkeley.edu; G. Friedland, 1947 Center Street, Suite 600, Room 505, Berkeley, CA, 94704; email: fractor@icsi.berkleey.edu; E. Battenberg, Soda Hall, ParLab, Berkeley, CA, 94720; email: ericb@eecs.berkeley.edu; P. Koanantakool, Soda Hall, ParLab, Berkeley, CA, 94720; email: penpornk@eecs.berkeley.edu; M. Driscoll, Soda Hall, ParLab, Berkeley, CA, 94720; email: mbdriscoll@eecs.berkeley.edu; E. Georganas, Soda Hall, ParLab, Berkeley, CA, 94720; email: egeor@eecs.berkeley.edu; K. Keutzer, Soda Hall, 566, Berkeley, CA, 94720.

DOI: http://dx.doi.org/10.1145/2517151

Additional Key Words and Phrases: Multimedia content analysis, parallelism, rapid prototyping, GPU

ACM Reference Format:

Ekaterina Gonina, Gerald Friedland, Eric Battenberg, Penporn Koanantakool, Michael Driscoll, Evangelos Georganas, and Kurt Keutzer. 2014. Scalable multimedia content analysis on parallel platforms using Python. ACM Trans. Multimedia Comput. Commun. Appl. 10, 2, Article 18 (February 2014), 22 pages.
DOI: http://dx.doi.org/10.1145/2517151

1. INTRODUCTION

Consumer-produced multimedia content is growing exponentially on the Web [Kosner 2012]. With hundreds of videos and images being uploaded to the web every minute, there is a high demand for scalable solutions to large-scale multimedia content analysis. Content analysis applications typically rely on machine learning techniques to automatically classify and predict useful content. For example, appealing multimedia content analysis applications include automatic video and audio transcription and search, recommendation of new content, and tools for geo-location and privacy analyses. Accurate and robust applications require training on hundreds of thousands of learning examples requiring hours or days of processing time. In addition, many multimedia content analysis applications often require real-time processing when integrated into interactive environments such as home entertainment systems and mobile applications, which require fast, low-latency solutions to multimedia processing. With such intensive application demands we are faced with the problem: how do we efficiently process growing multimedia data and create scalable appealing applications that meet user requirements?

Commoditization of parallel platforms such as multicore CPUs, Graphics Processing Units (GPUs) and multi-node clusters suggests one compelling solution to this problem: mapping the content analysis applications to parallel platforms. When efficiently mapped onto parallel platforms, computationally-demanding, large-scale and low-latency applications can achieve several orders of magnitude in performance improvements allowing for real-time processing and scaling to much larger datasets [Asanovic et al. 2006]. In fact, with single-node processor clock speeds tapering due to power constraints, in order for applications to scale to new processor architectures, they *have* to go parallel.

While the benefits of parallelizing applications are appealing, in order to utilize parallel hardware current practice is to rewrite the applications in low-level languages such as OpenMP [OpenMP 2008], Pthreads [Mueller 1995], CUDA [NVIDIA Corporation 2010] and OpenCL [Khronos Group 2010], resulting in cumbersome, hard-to-maintain code. Writing efficiency-level code takes significant amount of effort and expertise, not available to every application programmer. In addition, application code developed and tuned for one particular hardware platform is not portable to other platforms, requiring at least partial application code rewrite. Finally, with the exponential growth in multimedia data, application scalability becomes a highly important factor as well, further impeding application development. Thus, when developing applications, programmers are faced with the following dilemma: on one hand, they would like to stay productive and develop applications in a high-level language, on the other hand, their implementations need to be efficient, scalable and portable.

Libraries such as OpenCV [Gregory 2000] for computer vision and BLAS [Blackford et al. 2001] for numerical computations, present one solution. They package up complex, efficient implementations of particular computations allowing for reuse and productivity. However, libraries typically implement a specific computation for a specific platform, and thus are not flexible, limit application portability, presenting a brittle approach. Libraries with aggressive compiler optimization and auto-tuning such as ATLAS [Whaley and Petitet 2005] and OSKI [Vuduc et al. 2005] can be used to enhance both the portability and efficiency of library code by tuning the library code to a particular platform. However, with a fixed interface and implementation, libraries do not provide guidance on how to design an application *as a whole* to allow for most efficient and scalable implementation. Application frameworks on the other hand provide a flexible environment for application development while still allowing flexibility, (see for example Cactus [Goodale et al. 2002] and CHARMS [Grinspun et al. 2002]). Programmers can develop their applications using the framework that guides them in the composition of various computations in a predefined way.

In this work, we present a *software framework* solution that aims to address the following research questions.

(1) *Productivity & Efficiency*. How can one programming environment provide productivity of high-level languages (such as Python or MATLAB) and at the same time allow for efficient performance of low-level implementations?
(2) *Portability*. How can we ensure that the same application code is portable to new generations of processors and across a variety of hardware platforms?
(3) *Scalability*. How can one programming environment allow programmers to go from experimentation on a single node to a cluster of processors, from processing a sample subset to the entire dataset of content?
(4) *Flexibility*. Even when achieving the above requirements, how flexible is such a programming environment when designing and prototyping different content analysis applications?

Specifically, in this article we present PyCASP: a Python-based framework that automatically generates optimized parallel implementations of many multimedia content analysis algorithms from high-level Python code. PyCASP targets a variety of hardware platforms and allows for *automatic scalability* to clusters of processors. When designing PyCASP, we used a systematic pattern-oriented approach originally described in Keutzer and Mattson [2010] with the goal of creating a comprehensive software environment for multimedia content analysis. By identifying the set of core application, computational and structural patterns in multimedia content analysis applications, PyCASP is designed to be modular, have a tractable scope and yet to be flexible and applicable to a wide variety of applications. We describe in detail how PyCASP addresses the research questions to allow for productive, efficient and portable applications that scale to large datasets.

To illustrate how PyCASP addresses the application development concerns, we describe three multimedia content applications: a state-of-the-art speaker diarization system, a content-based music recommendation system, and a video event detection system. The speaker diarization system segments an audio recording into speaker-homogeneous regions detecting who spoke when in the recording. Using our framework, the application is captured in under 50 lines of Python code and achieves up to 115× faster-than-real-time performance on NVIDIA GPUs. The content-based music recommendation system finds a list of songs most similar to a given song or artist based on similarity between audio features in a database of one million songs. It is implemented in under 400 lines of Python code and is able to recommend songs based on queries from one million songs in under 1 second. The video event detection system automatically identifies "events" in video collections from the audio soundtrack using the TRECVid MED dataset. With a two-line change in application code, the system is able to achieve nearly optimal speedup of 15.5× on a 16-GPU cluster, processing 1000 video files in parallel compared to processing them sequentially on one machine. We show that PyCASP allows for rapid application prototyping, abstracting away the low-level details of parallel programming and automatically allows for performance and portability. Quantifying productivity is a difficult task, but from our assessments of previous implementations, the same application written in C/C++ would require at least 10–60× lines of code and 7–30× time to develop the application.

This article is structured as follows. Section 2 describes the conceptual design of the framework. Section 3 describes the pattern-oriented design of PyCASP. Section 4 discusses the internal details of PyCASP and its components. Section 5 outlines the three multimedia processing applications and Section 6 shows how PyCASP allows for rapid prototyping and automatic performance gains for each of the three applications. Section 7 shows performance and portability results. Section 8 discusses related work in parallelization technology as well as alternative solutions and Section 9 concludes.

2. CONCEPTUAL BACKGROUND

The goal of PyCASP is to address the following application development concerns:

Productivity & Efficiency. When analyzing programmer productivity, case studies have found that high-level scripting languages allow programmers to express the same programs in 3-10 fewer lines of code and in one fifth to one third the development time [Chaves et al. 2006; Hudak and Jones 1994; Prechelt 2000]. However, when we look at the landscape of parallel processing to compare application efficiency, reimplementing algorithms in C/C++ from Python code alone can give at least one to two orders of magnitude performance improvement [Catanzaro et al. 2009b]. Furthermore, when porting the application code from sequential C/C++ to multicore CPUs using OpenMP or Pthreads, application typically gain 2–10× in performance [You et al. 2009]. When porting sequential C/C++ to GPUs using CUDA or OpenCL we can see anywhere from 10× to 200× performance improvement [Chong et al. 2009; Catanzaro et al. 2008]. Finally, distributing the computation across a cluster of parallel machines can give further one or two orders of magnitude in performance improvement [Gonina et al. 2011]. Thus, starting from a Python-based application implementation and combining several parallelization techniques can give several orders of magnitude performance improvement, at the expense of significantly decreased productivity.

Portability. In order to extract the most performance out of a hardware platform, application code needs to be heavily optimized and tuned for that specific hardware architecture. Thus, source code developed for a single desktop GPU platform is not readily portable to a multicore CPU platform or even to a previous generation of a GPU. In order to run the application on a different platform, application source code needs to be rewritten for every platform, significantly impeding productivity.

Scalability. In order to enable application scalability to the vast amount of multimedia data, application code needs to scale to clusters of processors. To achieve scalability of their applications, programmers need to rewrite their applications using distributed programming frameworks such as Hadoop [White 2009] and MapReduce [Dean and Ghemawat 2008]. Furthermore, commodity clusters and datacenters typically consist of a large set of multicore nodes, thus requiring composition of code using the distributed frameworks and the low-level parallel implementations for single node, further impeding productivity.

Flexibility. When using a constrained programming environment, programmers can typically see significant performance improvements due to the constrained nature of the problem. On the other hand, unconstrained general programming frameworks allow for most flexibility at the expense of performance.

In this work, we present a software framework that aims to address the concerns above by combining the benefit or rapid prototyping in high-level languages *and* the orders of magnitude performance improvement from parallelization of application code as well as portability and scaling.

3. PRODUCTIVITY WITH PYTHON AND DESIGN PATTERNS

We allow for *individual programmer productivity* by embedding PyCASP in a high-level language, in our case Python. We chose Python due to its growing popularity among scientists and researchers but the methodology can be applied to any high-level language that supports higher-order functions and

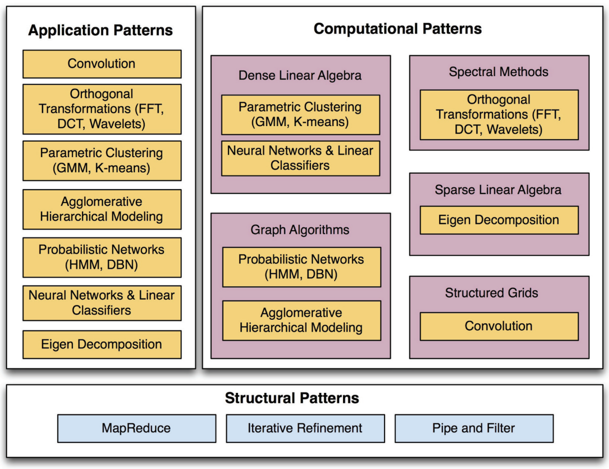
Application Patterns
Convolution
Orthogonal Transformations (FFT, DCT, Wavelets)
Parametric Clustering (GMM, K-means)
Agglomerative Hierarchical Modeling
Probabilistic Networks (HMM, DBN)
Neural Networks & Linear Classifiers
Eigen Decomposition
Computational Patterns
Dense Linear Algebra
Parametric Clustering (GMM, K-means)
Neural Networks & Linear Classifiers
Spectral Methods
Orthogonal Transformations (FFT, DCT, Wavelets)
Sparse Linear Algebra
Eigen Decomposition
Graph Algorithms
Probabilistic Networks (HMM, DBN)
Agglomerative Hierarchical Modeling
Structured Grids
Convolution
Structural Patterns
MapReduce
Iterative Refinement
Pipe and Filter

Fig. 1. Application patterns, computational patterns and structural patterns for conceptual design of PyCASP.

introspection. Further, we aim to bring modularity to PyCASP to aide in large application development and debugging by using a *pattern-oriented approach*.

The design of PyCASP is based on a systematic, pattern-oriented approach that uses the methodology and taxonomy of parallel patterns originally defined in Keutzer and Mattson [2010]. In order to create a modular framework that will comprehensively span a variety of multimedia content analysis applications, we extensively studied applications in the multimedia content analysis domain by implementing applications on a variety of parallel platforms [Chong et al. 2010; You et al. 2009; Chong et al. 2009; Anguera et al. 2012; Elizalde et al. 2012] as well as discussing the applications with domain experts. By mining the applications, we discovered a set of application patterns common to multimedia content analysis, shown in the left panel of Figure 1.

Using the pattern-based methodology, we can then distill the underlying computation in the application patterns to five distinct computational patterns, as shown in the right panel of Figure 1. Identifying and distilling the applications to the core computational patterns allows for a more tractable framework scope, as we can focus on specific computational building blocks that are present in the application patters. Moreover, we can identify specific parallel platforms for each computational pattern that allow for its most efficient implementation (for example GPUs are efficient at executing data-parallel dense linear algebra algorithms). Finally, this modular approach allows us to identify software already available in the community that may implement a specific computation on a given parallel hardware, allowing us to integrate it into the backend of our framework instead of reimplementing it from scratch. This allows us to develop a comprehensive framework for building parallel content analysis applications leveraging our knowledge of the application domain and existing tools and software.

At the final step in the design process, we identify the three structural patterns shown at the bottom of Figure 1. Structural patterns define ways in which the computational components can be *composed* together. For example, the MapReduce pattern is a common structural pattern used in large-scale data processing; it maps relatively independent computations onto separate nodes of a compute cluster to achieve higher scalability. Each computation can be an application component that runs on single parallel processor. This allows for nested parallelism, as demonstrated by the video event detection application in Section 6.3.

Identifying the core application, computational and structural patterns allows us to have a systematic way of developing modular, comprehensive software environments. By supporting the primary application and structural patterns present in the multimedia content analysis, we provide the basic building blocks and mechanisms for their composition to the application writer. This pattern-oriented design allows PyCASP to be applicable to a wide variety of multimedia content analysis applications, yet to be modular and have a tractable scope. By implementing all patterns and using customizations (discussed in the next section), we hope to make PyCASP flexible enough to develop a wide variety of multimedia content analysis applications.

After identifying the set of core design patterns used in PyCASP, we use the Selective Embedded Just-in-Time Specialization (SEJITS) approach for automatic parallelization of PyCASP's components to allow for efficiency, portability and scaling. In the next section we describe the SEJITS mechanism as well as present two components of PyCASP: a Gaussian Mixture Model (GMM) parametric clustering component that targets GPUs and multicore CPU platforms and a MapReduce component that is based on the MapReduce structural pattern and targets commodity clusters. PyCASP also includes a Support Vector Machine (SVM) linear classification application component as well as a Pipe-And-Filter structural component that are not discussed in this article.

4. EFFICIENCY AND PORTABILITY WITH SELECTIVE SPECIALIZATION

We use Selective Embedded Just-in-Time Specialization (SEJITS) [Catanzaro et al. 2009a] as the approach to implement automatic parallelization in our PyCASP framework to allow for *efficiency*, *portability* and *productivity*.

4.1 SEJITS

To allow for productivity, efficiency and portability in SEJITS-based frameworks, the approach focuses on the *separation of concerns*: the application programmer can focus on developing and innovating the application in Python and the efficiency programmer can focus on developing fast parallel code for the identified patterns using low-level languages such as C, OpenMP or CUDA. Efficiency programmers encapsulate the patterns in SEJIT *specializers* that automatically parallelize specific computations on parallel hardware. The separation of concerns allows SEJIT specializers to be efficient and target multiple parallel platforms in the back-end of the framework without changing the application code and thus allows the applications written using SEJITS-based frameworks to be performance *portable*.

When working with SEJITS-based frameworks, scientists express their applications entirely in Python using Python libraries and tools. They also import Python modules containing the framework's specializers. When calling the specialized functions from the application code, SEJITS specializer automatically generates low-level parallel code for a given back-end platform and executes the computation on the parallel platform. From the Python programmer's view, this experience is like calling a pure Python library, except that performance is potentially orders of magnitude faster.

As a SEJITS-based framework, PyCASP contains a list of components that are implemented as SEJIT specializers corresponding to the design patterns described in Section 3. In the next section we describe the details of creating SEJIT specializers for PyCASP and then discuss the implementation of the two PyCASP components: GMM parametric clustering component and MapReduce structural component.

4.2 Creating a Specializer

PyCASP is implemented using a framework for developing SEJIT specializers called Asp [Kamil et al. 2011]. Asp provides a framework for creating specializers as Python classes. The specializers can combine two specialization mechanisms: pre-specified low-level code (templates) and manipulation code of the Python abstract syntax tree (AST) to *generate* low-level source code from Python. The specializer logic is implemented in Python. Thus, Asp provides a general productive framework for embedding "mini-compilers" into Python that specialize specific computations to low-level parallel hardware.

Currently PyCASP uses the template-based mechanisms of Asp to implement its component as Asp specializers. To create a PyCASP component, the efficiency programmers typically start with an efficient version of a particular application pattern and a target hardware platform (for example Gaussian Mixture Model EM training written in CUDA code). They identify tuning/specialization parameters for each function that need to be adjusted for a particular instance of a hardware platform, for example number of thread blocks and number of threads per block in a CUDA implementation and define the data movement logic, if needed (for example moving training data to and from GPU memory). They can also have multiple code variants of the same computation (for example different blocking strategies or loop reordering) whose efficiency differs depending on particular hardware and input data parameters.

Once the efficient implementation(s) for a particular component exist, they are translated into Asp templates and placed into Asp code modules. Templates contain the original low-level code as well as place-holder variables for the tuning/specialization parameters for each function and modules to plug in different code variants of the same computation. The specializer logic specifies *how* to populate the place-holder values and select the code variants (for example, by querying the hardware platform for the specs or the shape of the input data) and what compiler toolchain to call on the generated code. All of the specializer logic is implemented in Python, only the template code is implemented in a low-level efficiency language, this allows for *specializer writer productivity*. When the specialized function gets called, Asp populates the place-holder variables, selects appropriate code variants according to specializer logic, and calls the appropriate compile toolchain on the resulting code. Once the code is compiled, it is automatically executed and the results are brought back to Python. The compiled object code is cached to avoid redundant recompilation.

To allow for *portability*, the specializer writer has to target multiple low-level backends; however, they can typically be grouped together into a few "general" backends, that is, one CUDA specializer backend can target all CUDA-programmable GPUs, while another Cilk+ specializer can target all Cilk-programmable Intel x86 hardware. Thus, while it does take significant amount of effort to implement a specializer, it is intended to be done by an efficiency programmer who is familiar with low-level intrinsics of the hardware and typically implementing the same computation for multiple backends is not as difficult once the programmer is familiar with the algorithm. In addition, this effort has to be done once, and can then be reused by all application programmers who use PyCASP without knowing any details of the specializer implementation. At the cost of increased specializer developer coding time, this approach significantly reduces application developer coding time.

Finally, given a multilayered structure of the specializers, special debugging techniques need to be employed to guarantee specializer correctness, such as Xia et al. [2012]; this is generally ongoing research work. Debugging applications that use specializers can be facilitated by ensuring PyCASP's specializes have high code coverage and undergo thorough regression testing. Since specializer creation is an isolated process, this process is fairly self-contained. The application code is then written in compact Python using the specializers, this facilitates isolation of user errors.

As an example of two concrete specializers, in the next section we present the details of two PyCASP components: the GMM parametric clustering specializer and the MapReduce specializer.

4.3 Parametric Clustering

The parametric clustering GMM component implements the Expectation - Maximization (EM) algorithm [Bishop 1995] for training GMMs as well as computing the likelihood given an already trained

```
import numpy as np
from gmm import *

training_data = get_training_data() # get numpy array of training data
testing_data = get_testing_data() # get numpy array of testing data

M = 16  # set number of GMM components
D = training_data.shape[1]   # get dimensionality of the data

gmm = GMM(M, D, cvtype='diag')    # initialize the GMM, use diagonal covariance
gmm.train(training_data)    # train the GMM

log_lkld = gmm.score(testing_data)    # evaluate the GMM to get log likelihoods on test data
```

Fig. 2. Example usage of the GMM component.

GMM. Given an initial estimate of the parameters, the EM algorithm iterates between two phases: the E-step and the M-step. The E-step computes the expectation of the log-likelihood of the observations given parameter estimates. The M-step in turn computes the parameter estimates that maximize the expected log-likelihood of the observation data.

The GMM component performs training on NVIDIA GPUs by adopting the parallelized EM algorithm (originally presented in [Pangborn 2010]) written in CUDA. The specializer also targets multi-core CPUs using the Cilk+ [Intel] threading framework. To select which parallel platform the application will be run on, the programmer selects a backend in a config file specifying either “cuda” or “cilk” as the backend target.

4.3.1 *CUDA Programming Model.* A CUDA application is organized into sequential host code written in C running on the CPU and many parallel device kernels running on the GPU [NVIDIA Corporation 2010]. The kernel executes a set of scalar sequential programs across a set of parallel threads. The programmer can organize these threads into thread blocks, which are mapped onto the processor cores at runtime. Each core has a small software-managed fast local memory. To run an application on the GPU, data structures must be explicitly transferred from host to device. Task scheduling and load balancing are handled by the device driver automatically. Figure 2 shows example usage of the GMM component in Python code.

4.3.2 *GMM Training on the GPU.* The parallel training code consists of a set of CUDA kernel functions for the expectation and a maximization steps of the EM algorithm. Depending on the selected type of model, there are several parameters that are passed to the algorithms. For example, in GMM training, there are three parameters: M the number of Gaussian components in each GMM D, the dimensionality of the Gaussian components; and N, the number of feature vectors.

PyCASP’s GMM component implements several versions of the GMM training computation for the GPU and handles all data movement to and from the device. The efficiency of each version depends on values of M, D and N as well as the hardware characteristics of the GPU. For details of the implementation see Cook et al. [2011].

On average we saw a 30% performance improvement in covariance matrix computation time during EM training by using the optimal code version compared to always using the original implementation described in Pangborn [2010]. PyCASP automatically selects the best version given M, D and N values and the platform specifications. The GMM training component is able to run on any CUDA-programmable NVIDIA GPU allowing for portability across generations of NVIDIA GPUs.

4.3.3 *Cilk+ Programming Model.* Intel Cilk+ language [Intel] is a set of C/C++ extensions for programming multicore processors. The programmer exposes parallelism in a C/C++ program by identifying elements that can be executed concurrently and marking it with a Cilk-specific keyword. The programming environment decides exactly how the work is split among threads and how tasks are scheduled, thus the same program can run on one as well as many processors without code change.

4.3.4 *GMM Training on a Multicore CPU.* The component's EM training algorithm is implemented using Cilk+, extended from sequential C code. The component uses `cilk_for` keywords around loops whose elements can be computed in parallel in the E and the M stages. We also use the reducer operators to perform cumulative computations (reductions) in our training code. The number of processors that the training utilizes is controlled by `CILK_NWORKERS` variable without change to the specializer code.

4.4 MapReduce Specialization

MapReduce enables the automatic parallelization of programs that implement specific `map` and `reduce` operations on key-value pairs, or records. Specifically, the `map` operation takes one record as input and generates any number of intermediate records. Intermediate records with the same key are grouped by the runtime system and passed to the `reduce` function, which takes a list of records with the same key and emits any number of output records. Using these two simple data operators, many loosely coupled tasks can be readily mapped to clusters [Ramakrishnan et al. 2011].

PyCASP's MapReduce component provides both `map` and `reduce` functions for the application writer to use as well as high-level functions that compose one or more calls to `map` and `reduce` and perform common operations for multimedia content analysis. From the programmer's point of view, it replaces for-loops where each iteration is conceptually independent but must be executed in order because of language semantics.

Our implementation targets the popular Hadoop MapReduce framework [White 2009] via the `mrjob` abstraction for writing MapReduce programs in Python.[1] Our choice of MapReduce and Hadoop as the means of parallelization has several added benefits: Hadoop has automatic load balancing and fault tolerance, and it provides an array of tuning parameters that PyCASP can use to optimize performance. Furthermore, it comes bundled with a distributed filesystem that provides high read bandwidth, data durability, and the ability to schedule computation close to its data.

GMM training can be performed on a cluster of GPU nodes by using the MapReduce component and composing it with the single-GPU GMM training component. This composition enables scaling by executing separate, independent training instances simultaneously on many machines in the cluster.

5. EXAMPLE APPLICATIONS

We now give an overview of the three multimedia content analysis applications that we implemented in Python using PyCASP. All three applications use the GMM parametric clustering application pattern. In addition, the video event detection system uses the MapReduce structural pattern to scale to a large set of videos. Section 6 then describes how each application uses the framework and Section 7 describes the performance and portability results achieved.

5.1 Speaker Diarization

Speaker diarization is the process of segmenting an audio recording into speaker-homogeneous regions, addressing the question "who spoke when" without any prior knowledge of the recording. One popular

[1]http://packages.python.org/mrjob/.

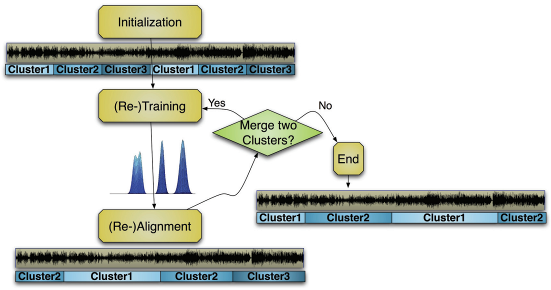
Initialization
Cluster1 Cluster2 Cluster3 Cluster1 Cluster2 Cluster3
(Re-)Training
Yes
No
Merge two Clusters?
End
Cluster1 Cluster2 Cluster1 Cluster2
(Re-)Alignment
Cluster2 Cluster1 Cluster2 Cluster3

Fig. 3. llustration of the segmentation and clustering algorithm used for speaker diarization.

diarization method is the Bayesian Information Criterion (BIC) with GMMs trained with frame-based cepstral features [Anguera et al. 2012]. This method combines the speech segmentation and segment clustering tasks into a single stage using agglomerative hierarchical clustering, a process by which many simple candidate models are iteratively merged into more complex, accurate models. Figure 3 shows the general organization of such a diarization system.

The diarization is based on 19-dimensional, Gaussianized, Mel-Frequency Cepstral Coefficients (MFCCs). We use a frame period of 10ms with an analysis window of 30ms in the feature extraction, as well as the speech/nonspeech segmentation used in Wooters and Huijbregts [2007]. In the segmentation and clustering stage of speaker diarization, an initial segmentation is generated by uniformly partitioning the audio track into K segments of the same length. K is chosen to be much larger than the assumed number of speakers in the audio track. For meeting recordings of about 30 minute length, previous work [Imseng and Friedland 2009] experimentally determined $K = 16$ to be a good value.

The procedure for diarization is shown in Figure 3 and takes the following steps (more details can be found in Wooters and Huijbregts [2007]).

(1) *Initialize*: Train a set of GMMs, one per initial segment, using the expectation-maximization (EM) algorithm.
(2) *Resegment*: Resegment the audio track using majority vote over the GMMs' likelihoods.
(3) *Retrain*: Retrain the GMMs on the new segmentation.
(4) *Agglomerate*: Select the most similar GMMs and merge them. At each iteration, the algorithm checks all possible pairs of GMMs, looking to obtain an improvement in BIC scores by merging the pair and retraining it on the pair's combined audio segments. The GMM clusters of the pair with the largest improvement in BIC scores are permanently merged. The algorithm then repeats from the resegmentation step until there are no remaining pairs whose merging would lead to an improved BIC score.

The result of the algorithm consists of a segmentation of the audio track with n segment subsets and with one GMM for each subset, where n is assumed to be the number of speakers. This system was proven to be highly effective, but the computational burden was such that the processing took about real-time. In Section 6 we discuss how PyCASP can provide performance gains to mitigate such overheads, while still allowing innovation at the application level.

5.2 Content-Based Music Recommendation

Music recommendation is one of the most challenging applications in Music Information Retrieval (MIR). The goal of a music recommendation system is to recommend a set of songs that are most similar to a given song or artist. Most current recommendation systems such as Pandora (www.pandora.com)

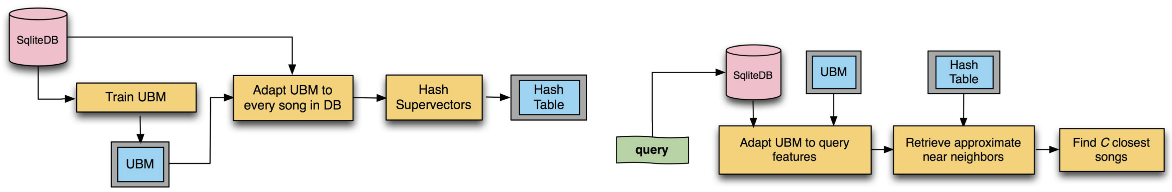

Fig. 4. Left: Offline data preparation phase of the content-based music recommendation system. RIght: Online song recommendation phase of the content-based music recommendation system.

and Last.fm (www.last.fm) use collaborative filtering [Takács et al. 2009] or manually label songs with tags. These approaches require tedious, manual labeling of high-level audio features thereby severely limiting scalability of the system.

Our second example application is a content-based music recommendation (CBMR) system that produces a set of songs similar to a given query song based on the audio features of the songs. We combine several existing approaches to create a music recommendation system in order to illustrate the fast prototyping environment of PyCASP.

Our system uses the UBM[2]-GMM supervector approach and Locality Sensitive Hashing (LSH) described in Charbuillet et al. [2011] and Michael Casey and Slaney [2008], respectively, to retrieve a set of most similar songs given a query song title or artist name. Figure 4 (left and right) shows the offline and online phases of the recommendation system. In the offline phase, the system prepares the data for online querying. In the online phase, the system uses the data structures pre-built during the offline phase to retrieve a list of most similar songs to a given query. We use a Python interface to SQLite database to store and retrieve song meta-data and features (shown as “SqliteDB” in the Figures).

5.2.1 *Dataset and Audio Features.* We use the Million Song Dataset [Bertin-Mahieux et al. 2011] assembled by Columbia University using data provided by Echo Nest. The Million Song Dataset contains audio features (onsets, timbre) as well as metadata (title, artist name, duration etc.) for 1 million popular songs. Our system uses the timbre features to recommend a set of similar songs out of the 1 million songs based on the features of the songs that matched the query.

5.2.2 *Offline Data Preparation Phase.* In the offline phase (shown on the left in Figure 4), we prepare data to be used during recommendation in the following way.

(1) *Train UBM*. We first train the UBM on a random subset of all timbre features of all songs in the Million Song Dataset to obtain UBM parameters (weights, means and covariance of the GMM). The UBM is a GMM of 64 components.

(2) *Adapt the UBM to all songs*. After we compute the UBM parameters from the previous step, we use UBM MAP adaptation (described in [Charbuillet et al. 2011]) to compute supervectors (mean vectors of the trained GMM) for each of the songs in the Million Song Dataset and normalize them using the MCS-norm [Charbuillet et al. 2011].

(3) *Hash song supervectors*. After computing and normalizing the supervectors, we use a Locality Sensitive Hashing (LSH) technique to hash the supervectors to a hash table. LSH is a general technique for computing approximate nearest-neighbors in a high-dimensional space originally described by Andoni and Indyk [2006]. We base our implementation on the one described by Casey in Michael Casey and Slaney [2008]. We use $K = 8$ projections and $L = 11$ hash tables with the

[2]UBM=Universal Background Model.

quantization bin size of $w = 1.291$ to retrieve (on average) 90% of songs within $R = 0.3$ radius of the query point and reject 89% of songs farther than $c * R = 0.72$ radius from the query point.

5.2.3 *Online Recommendation Phase.* In the online recommendation phase of our CBMR system (shown on the right in Figure 4), we use the data structures constructed during the offline phase and compute the set of songs similar to a given query.

(1) *Get the query from the user*. Our system can recommend songs based on song title, an artist name or a list of song titles and/or artist names. After receiving a query, we retrieve the features of the songs that match the query (i.e., songs with the given artist name) from the SQLite database and concatenate the set of timbre features.
(2) *Adapt the UBM on the query*. After obtaining the timbre features for all the songs that matched the user query, we then adapt the UBM on the query song features to obtain the query timbre supervector.
(3) *Get approximate nearest-neighbors for the query supervector*. We use our LSH hash functions to retrieve the set of nearest neighbors to the query supervectors.
(4) *Compute the closest C songs*. We use p-norm distance (described in [Charbuillet et al. 2011]) to compute the C closest songs out of the nearest neighbors returned by our LSH table.

In Section 6.2 we describe how we use PyCASP to implement the CBMR system and in Section 7 show the performance results for different query sizes.

5.3 Video Concept Detection

The third application illustrating the use of PyCASP is a data-driven video event detection system originally described in Elizalde et al. [2012]. Most state-of-the-art approaches rely on manual definition of predefined sound concepts such as "engine sounds" or "outdoor/indoor sounds". These approaches require manual event definitions and are very domain specific. The goal of our third application is to detect events in "wild" videos found on the web. The definition of "event" goes beyond simple object recognition to more abstract concepts such as "feeding an animal," "wedding ceremony," or "attempting a board trick." This system is designed for large scale retrieval and tested on the TRECVid MED 2011 development data set. It performs learning based on arbitrary low level audio features and can be used to detect high level concepts like those given in the dataset. Here we don't aim to present a new event detection technique but to show how an approach like this can be productively implemented using PyCASP and scale to large sets of videos.

The event detection system is generalized from speaker diarization (described in Section 5.1) for indexing audio contents. We use the detection system to automatically identify low-level sound concepts similar to annotator defined concepts and then use these concepts for indexing. The applicability of speaker diarization to video indexing and event detection is most similar to the approaches described in Lu and Hanjalic [2008] and Chaudhuri et al. [2011].

As in the speaker diarization system, we use GMMs to represent the audio concepts. In order to match low level audio concepts across training videos and also to classify low level feature models found in testing videos, the system reduces the per-event GMM to a single vector that consists of the sums of the weighted means and the sums of the weighted variances of each Gaussian (we call this vector a simplified supervector). A K-means method is then used to cluster the simplified supervectors that were generated from all of the low level acoustic concepts, resulting in clusters that represent audio event abstractions. These can then be mapped back to the concepts in each video by calculating the distance between the video's speaker models and the abstract simplified supervectors.

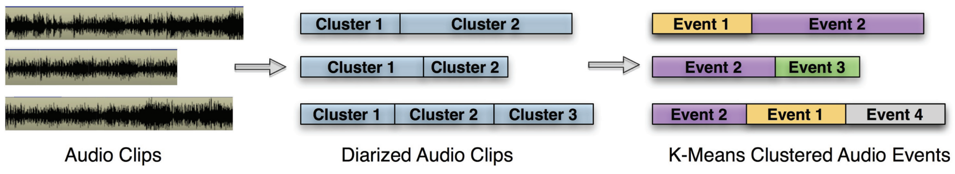

Fig. 5. Overview of the video event detection system. Each video soundtrack file gets diarized, that is, clustered based on the audio event content. Then all clusters across all audio files get clustered into a global set of audio events using k-means clustering.

Figure 5 shows the overall structure of the video event detection system. Starting with the features of audio files, each file is diarized using the speaker diarization algorithm. Then K-means clustering is performed on the simplified supervectors. This results in each audio file being segmented into events, and all the same events clustered together. Thus, we obtain a global view of the different audio events present in a video collection and can then perform indexing and search based on the events.

Section 6.3 describes how we use PyCASP's MapReduce component to accelerate and scale the video event detection system by performing diarization on videos in parallel and Section 7 then discusses performance results on a cluster of GPUs.

6. APPLICATIONS & SPECIALIZATION

We now describe how each of the three multimedia content analysis applications utilize PyCASP to automatically target GPUs and clusters of GPUs. We describe the speaker diarization first and in most detail to illustrate the separation of concerns in our approach and then outline the use of PyCASP in the other two content analysis applications.

PyCASP itself is written in about 800 lines of Python, the C/CUDA/Cilk+ code for GMM training is written in about 3600 lines and the MapReduce component is written in about 80 lines of Python. Both the specialization framework, the low-level GMM training code and the MapReduce specializer code are written once and can be reused by all applications on recent CUDA-programmable GPUs, multicore Intel CPUs or Hadoop clusters.

6.1 Speaker Diarization Implementation

Using PyCASP, the implementation of our diarization system is captured in less than 50 lines of Python code and is shown in Figure 6 with the components that are executed on the parallel platform (either GPU or multicore CPU) highlighted in light-gray.

Based on the algorithm description from Section 5.1 we now step through the Python code.

(1) *Initialize*. First we import our specialization framework (line 1) and uniformly initialize a list of K GMMs (in our case 16 5-component GMMs) on line 5. After creating the list of GMMs, we perform initial training on equal subsets of feature vectors (lines 6–9). The training computation is executed on the parallel platform. Next, we implement the agglomerative clustering loop based on the Bayesian Information Criterion (BIC) score [Reynolds and Torres-Carrasquillo 2005] (line 12–13).
(2) *Resegment*. In each iteration of the agglomeration, we resegment the feature vectors into subsets using majority vote segmentation (lines 16–18). We use the parallel platform to compute the log-likelihoods (`gmm.score()` method), which calls the E-step of the GMM training algorithm.
(3) *Retrain*. After resegmentation, we retrain the Gaussian Mixtures on the parallel platform on the corresponding subsets of frames (lines 22–23).
(4) *Agglomerate*. After retraining, we decide which GMMs to merge by first computing the unscented-transform-based KL-divergence of all GMMs (line 31). We then compute the BIC score of the top k

```
import numpy as np
from gmm import *

# Main diarization function. Parameters:
#    M: number of GMM components, D: data dimensionality, K: starting number of segments

def diarize(self, M, D, K, data):
  gmm_list = new_gmm_list(M,D,K)
  N = data.shape[0]
  per_cluster = N/K
  init = uniform_init(gmm_list, data, per_cluster, N)
  for gmm, data in init:
    gmm.train(data)

  # Perform hierarchical agglomeration
  best_BIC_score = 1.0
  while (best_BIC_score > 0 and len(gmm_list) > 1):

    # Resegment data based on likelihood scoring
    L = gmm_list[0].score(data)
    for gmm in gmm_list[1:]:
      L = np.column_stack((L, gmm.score(data) ))
    most_likely = L.argmax()
    split_data = split_obs_data_L(most_likely, data)

    for gmm, data in split_data:        # retrain on new segments
      gmm.train(data)

    # Score all pairs of GMMs using BIC
    best_merged_gmm = None
    best_BIC_score = 0.0
    m_pair = None

    #find most likely merge candidates using KL
    gmm_pairs = get_top_K_GMMs(gmm_list, 3)

    for pair in gmm_pairs:
        gmm1, d1 = pair[0] #get gmm1 and its data
        gmm2, d2 = pair[1] # get gmm2 and its data
        new_gmm, score =
            compute_BIC(gmm1, gmm2, concat((d1, d2)))
        if score > best_BIC_score:
          best_merged_gmm = new_gmm
          m_pair = (gmm1, gmm2)
          best_BIC_score = score

    # Merge the winning candidate pair
    if best_BIC_score > 0.0:
      merge_gmms(gmm_list, m_pair[0], m_pair[1])
```

Fig. 6. Speaker diarization in Python. Components that are executed on the parallel platform are highlighted in light-gray.

pairs of GMMs (in our case $k = 3$) by retraining merged GMMs on the GPU as described in Section 5.1 (lines 33–37) and keeping track of the highest BIC score. Finally we merge two GMMs with the highest BIC score (lines 43–45) and repeat the iteration until no more GMMs can be merged.

6.2 CBMR Implementation

The Content-Based Music Recommendation (CBMR) system uses the GMM training component of PyCASP to train and adapt the UBM on the GPU. Based on the algorithm described in Section 5.2, we now describe the usage of PyCASP in the application:

6.2.1 *Offline Data Preparation Phase*

(1) *Train UBM*. We use the GMM training component of PyCASP to train the timbre UBM. We randomly sample the features of all songs in the database and train the UBM on 7 million timbre vectors. This constraint on the number of features is based on the size of DRAM on the GPU platform. To train the UBM we setup a GMM object and invoke the `train()` method on it, as shown in the speaker diarization example in Figure 6 on line 9.

(2) *Adapt the UBM to all songs*. To adapt the UBM we also use the GMM training component, but set the number of EM iterations to 1. For every song in the database, we retrieve the timbre features and call the GMM training component to adapt the UBM to each song using the features.

(3) *Hash song supervectors*. This step does not use specialization but is implemented in Python using the NumPy [Ascher et al. 1999] library.

6.2.2 *Online Recommendation Phase*

(1) *Get the query from the user*. This step requires parsing the user query and obtaining the song features from the SQLite database using Python tools. No specialization is used in this step.

(2) *Adapt the UBM to the query*. After parsing the query, we retrieve the timbre vectors of all the songs that match the query and store them in a matrix. We then use the PyCASP GMM training component to adapt the UBMs to the features by doing 1 EM iteration of the `train()` function.

(3) *Get approximate nearest-neighbors for the query supervector*. We retrieve the approximate nearest neighbors from the hash tables. This is currently done in Python using the NumPy [Ascher et al. 1999] library.

(4) *Compute the closest C songs*. Computing the distance between the query and all the nearest neighbors returned in the previous step is done in Python using the NumPy [Ascher et al. 1999] library.

We are able to rapidly prototype the system in about 400 lines of Python code (excluding the SQLite database and LSH setup) and use PyCASP to remove the UBM training bottleneck by automatically offloading the computation onto the GPU.

6.3 Video Event Detection Implementation

The video event detection system uses the MapReduce component of PyCASP to execute separate instances of the speaker diarization clustering algorithm on a cluster of GPU-equipped machines. The clustering algorithm is described in Section 5.3 and uses the GMM training component. We compose the MapReduce component and GMM training component to map the event detection system to a cluster of GPUs using a two-line code change in the application code.

As mentioned in Section 4, the MapReduce component of PyCASP provides a high-level `map` function for applying the same diarization operation to a set of data segments in parallel. We use this function to perform the diarization computation on each video in parallel. Figures 7 and 8 show a portion of the

```
import sys
from diarizer import diarize

input_filenames = parse_input_filenames( sys.argv ):
for filename in input_filenames:
    diarize(components=5, dim=60, clusters=16, data=filename )
```

Fig. 7. A for-loop that applies the "diarize" operation to every filename. Python runs each iteration sequentially.

```
input_filenames = parse_input_filenames( sys.argv ):
map(diarize, components=5, dim=60, clusters=16, data=input_filenames )
```

Fig. 8. The same operation (in light-gray) expressed without an implied order, allowing each iteration to be executed in parallel.

FF DER	FF ×RT	NF DER	NF ×RT
35.49 %	71.02×	24.76 %	115.40×

Fig. 9. Average Diarization Error Rate (DER) of the diarization system and the faster than real-time performance factor (×RT) for far-field (FF) and the near-field (NF) microphone array setup for the AMI corpus.

video event detection code before and after it was refactored to use the MapReduce component. From the application point-of-view, this is the entire diarization phase of the video event detection system, consisting of 4 lines of code for the MapReduce component in addition to the diarization code shown in Section 6.1. With this modification, the application is now able to scale to a cluster of GPUs.

When invoked, the MapReduce component generates an input record for each content file to be processed. The record is passed to a worker node, which diarizes the file efficiently using the aforementioned diarization algorithm. The reduction step is skipped, and the call returns when all input records (video soundtrack files) have been successfully processed. Hadoop handles load balancing by assigning records to under-utilized nodes and fault tolerance by reassigning records from nodes that have failed to healthy ones.

Thus, when using PyCASP, a two line change scales the application from a single GPU desktop to a cluster of GPUs within the same software environment.

7. RESULTS

7.1 Speaker Diarization

We evaluated the performance of our speaker diarization system using a popular subset of 12 meetings (5.4 hours) from the Augmented Multiparty Interaction (AMI) corpus [Carletta 2007] and comparing to an equivalent system written in C++. The AMI corpus consists of audio-visual data captured from four to six participants in a natural meeting scenario. For the experiments described here, the beamformed far-field (FF) and near-field (NF) array microphone signals were used.

Figure 9 shows the accuracy and speed performance of the speaker diarization system averaged across the 12 meetings. Columns "FF DER" and "NF DER" show the average accuracy (Diarization Error Rate (% DER)[3]). The accuracy of our system is consistent with a state-of-the art system described in Friedland et al. [2010]. Columns "FF ×RT" and "NF ×RT" in Figure 9 show corresponding performance for far-field and near-field microphone setup in terms of faster than real-time factor (×RT) using PyCASP on NVIDIA GTX480 GPU using CUDA 3.2. The ×RT factor is computed by dividing the

[3]National institute of standards and technologies: Rich transcription spring 2004 evaluation.

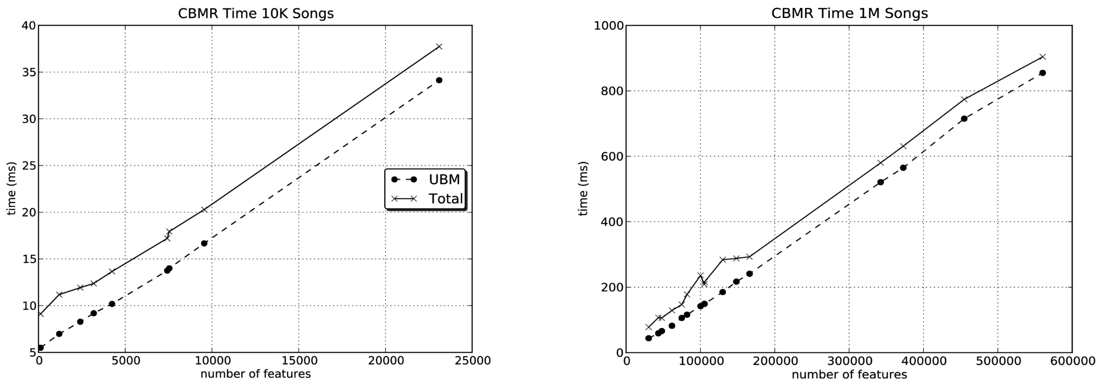

Fig. 10. Scaling of the CBMR system on the 10,000 song subset of the Million Song Dataset (left) and on the full dataset (right).

meeting time by the processing time. For example, a 100× real-time factor means we can process a ten minute meeting in six seconds.

Figure 9 shows the average performance numbers. The real-time performance varies by each meeting from about 50–250×RT, depending on the length of the audio as well as the number of clustering iterations computed before convergence [Gonina et al. 2011a]. Our reference application achieves about real-time performance.

7.2 Content-Based Music Recommendation

We evaluate our CBMR system performance using two datasets: the 10,000 song subset of the Million Song Dataset and the entire one million set of songs in the Million Song Dataset on a one NVIDIA GTX480 GPU desktop. Figure 10 shows the time it takes to adapt the UBM and the total recommendation time (excluding the database accesses) for various query sizes for the two datasets respectively. The total time contains UBM adaptation time, approximate nearest-neighbor computation using LSH and computing the C closes songs returned by LSH (Steps 2–4 shown in Section 5.2.3). We used artist names to query our CBMR system for recommendations; for example, we ran queries like “Radiohead” and “Elton John or Eric Clapton”. In the 10K system, typical queries returned 1–17 songs, up to 23,000 feature vectors. The number of songs returned by our queries using the 1M song recommendation system ranged from 30 to 500, up to 560,000 feature vectors total. Figure 10 shows the scalability of PyCASP’s GMM training component as we increase the number of features returned by a query. For the largest query our system is able complete the online recommendation phase from one million songs in under 1 second. The GMM training component in PyCASP can process up to 7 million features to train 64-component GMMs on NVIDIA GTX480 GPU. Thus, we believe that the UBM adaptation phase will scale to even larger queries.

Since the Sqlite database query time is quite different depending on whether the data is in memory or on disk, ranging anywhere from 1 to 200 seconds, we do not include the time it in the analysis. Future work will include improving the database to improve query execution time.

In the CBMR system, the offline phase takes a significantly longer time to execute than the online phase. The most significant component of the offline phase of the CBMR system is the UBM feature gather, the Sqlite database setup and the LSH hash table setup. Since these operations require disk access, the offline phase of our CBMR system is I/O bound and takes about 24–36 hours to complete on a 120GB database. This step needs to be done once for the entire system, and thus we don’t include it in the detailed analysis of performance.

Backend	SEJITS(CUDA)	SEJITS(Cilk+)	Native(CUDA)	Native(Pthreads)
M = 2	**0.07**	0.09	**0.07**	0.37
M = 10	**0.11**	0.28	0.12	1.82
M = 20	**0.15**	0.30	0.18	3.68

Fig. 11. GMM training time (in seconds) given number of mixture-model components M using the CUDA backend and a native CUDA version (both on NVIDIA GTX480), and the Cilk+ backend and a C++/pthreads version (both on dual-socket Intel X5680 Westmere 3.33GHz). Best time in bold font.

Mic Array	Orig. C++/pthreads	Py+Cilk+	Py+CUDA
	Westmere	Westmere	GTX285/GTX480
Near field	20×	56×	101 × / 115×
Far field	11×	32×	68 × / 71×

Fig. 12. The Python application using CUDA and Cilk+ outperforms the native C++/pthreads implementation by a factor of 3–6.

7.3 Video Concept Detection

To evaluate the performance of the diarization part of the video event detection system on a cluster of GPUs, we use the TRECVid Med 2011 dataset. This dataset is comprised of consumer-produced videos collected from social networking sites, or “found videos.” The data is broken down into 15 categories or “event-kits,” with 5 of those categories available in the test set. The event categories available in the test set are “attempting a board trick,” “feeding an animal,” “landing a fish,” “wedding ceremony,” and “working on a woodworking project.” Of the test set, 496 videos are from these 5 categories, and the remaining 3755 videos are random videos not belonging to any event category. The system uses 60-dimensional features: C0-C19 Linear Frequency Cepstral Coefficients (LFCC) features with 25 ms windows and 10 ms step size, along with deltas and double-deltas.

We performed the experiments on a cluster of 8 nodes with two NVIDIA Tesla M1060 GPUs each, total of 16 GPUs, using CUDA 3.2. We analyzed the speedup of the video event detection system using the MapReduce component of PyCASP compared to running the diarization on each video sequentially on one machine with one GPU. We map an increasing number of videos from different event categories to nodes in our cluster to investigate the scalability of our implementation. Once we process enough video files, we obtain nearly perfect speedup - 15.5 on 500 and 1000 videos, showing that our MapReduce component achieves nearly optimal scaling to the GPU cluster. We performed the same experiment on Amazon EC2 cloud compute platform, yielding the same results.

7.4 Portability

Figure 11 shows that the GMM training performance of PyCASP’s component can beat even the hand-coded CUDA [Pangborn 2010] implementation by selecting the best-performing algorithmic variant at runtime based on training problem size [Cook et al. 2011]. The specializer can emit CUDA and Cilk+ code, making it performance-portable both within and across architecture families with no changes to client Python application code.

Figure 12 shows the speaker diarization performance on Intel Westmere CPU and on two generations of NVIDIA GPUs, GTX285 and GTX480. On all platforms, the implementation that uses PyCASP achieves higher performance than even the reference sequential C++ implementation.

8. RELATED WORK

8.1 Speedups of Applications

Many multimedia content analysis applications have seen significant performance improvements when mapped onto multicore CPUs and GPUs [Chong et al. 2009; Battenberg and Wessel 2009; Ferraro et al. 2009; Sundaram et al. 2010; Catanzaro et al. 2009b]. While these efforts demonstrate significant speedups (e.g., Chong et al. achieve 10.5× faster than real-time performance with a parallel speech recognition inference engine [Chong et al. 2009]), their parallel implementations are written in complex low-level code, are not readily reusable in other applications, and are typically not portable to other platforms. In contrast, PyCASP allows for code reuse by abstracting the computation into pattern-based specialized components which can be used by any application on multiple parallel platforms, all transparent to the high-level application writer.

As discussed by Slaney [2010], recent advances in cloud computing technology have allowed researchers to rewrite several multimedia content analysis algorithms to minimize dependencies among tasks and thereby make them more amenable to parallelism [Chang et al. 2009; Liu et al. 2011]. When mapped to a cluster of compute nodes, these algorithms can see substantial improvement in scalability compared to their sequential implementations. However, implementing these algorithms using MapReduce frameworks is a nontrivial task in itself. PyCASP aims to give researchers a software environment that enables productive implementation of such algorithms on clusters of machines.

8.2 Frameworks Targeting the GPU

In our work, we focus on domain experts and aim to bring the benefits of raising the levels of abstraction and autotuning to high-level languages, in our case Python, to improve *domain expert* productivity and application performance. Other Python-based GPU frameworks allow for similar improvements in performance and productivity, for example Theano [Bergstra et al. 2010], a CPU/GPU math-compiler for Python, Copperhead language [Catanzaro et al. 2010] that provides a set of data-parallel abstractions expressed as a restricted subset of Python and the Delite framework [Chafi et al. 2011] a Domain Specific Language (DSL) creation framework and runtime. While these frameworks provide automatic mapping of data-parallel computations onto GPUs, they are not application-specific and therefore lack specific constructs for creating multimedia content analysis applications.

8.3 Libraries and Frameworks for Multimedia Analysis

There is an extensive list of frameworks for audio and visual media analysis, here we name a few important ones. OpenCV [Gregory 2000] is a framework for developing computer vision applications, that includes Python, C/C++ and Java interfaces and targets CPU platforms. The Hidden Markov Model Toolkit (HTK) [HMM] is a portable toolkit for building and manipulating hidden Markov models as well as performing other processing on audio, used for speech recognition. Marsyas [Tzanetakis 2007] is a popular music information retrieval (MIR) software framework for rapid prototyping, design and experimentation with audio analysis and synthesis with specific emphasis on processing music signals in audio format. CLAM is another C++ framework, for development and research in audio and music signal processing applications [Amatriain et al. 2002]. These frameworks provide an extensive API for processing audio and image data, however most of them are presented in a low-level language such as C++ and do not target parallel platforms. While some frameworks, such as OpenCV have some modules that are ported to parallel backends, parallelism is not the framework's primary target. PyCASP's goal is to present programmers with a single software environment putting *parallelism and portability* (from CPUs to GPUs to clusters) with auto-tuning and specialization as its first class citizens. With significant amount of effort and collaboration, PyCASP aims to become as comprehensive as the above mentioned frameworks in terms of API support.

9. CONCLUSION

In this article, we presented PyCASP, a Python-based content analysis parallelization framework. PyCASP is designed using a systematic, pattern-oriented approach with the goal of making it modular, comprehensive and applicable to a wide range of multimedia content analysis applications. With PyCASP we aim to provide the programmers with a software environment allowing for *both* productivity in application development and performance gains of several orders of magnitude using automatic parallelization. Using the selective specialization approach and its separation of concerns, PyCASP aims to also provide application portability across a variety of parallel platform targets without application code change. Finally, applications written using PyCASP can be scalable to large sets of multimedia data enabling the programmer to easily go from single-desktop, small subset experimentation to running the application on a cluster of parallel nodes using a large comprehensive dataset.

To illustrate the broad applicability of PyCASP we have shown three diverse multimedia content analysis applications that use our framework: a speaker diarization application, a content-based music recommendation system, and a video event detection system. We showed that across this wide range of applications and parallel platforms, PyCASP is able to give high productivity to application writers when prototyping applications by reducing the number of lines of code needed for an implementation by 10–60$\times$. Our example applications using PyCASP are able to automatically achieve 50–100$\times$ faster-than-real-time performance on both multicore CPU and GPU platforms and 15.5$\times$ speedup on 16-node cluster of GPUs showing near-optimal scaling.

PyCASP illustrates an initial effort in a larger project of using selected embedded specialization to create productive and efficient software environments for a variety of application areas. PyCASP illustrates the use of key elements such as pattern-oriented design, SEJITS for component specialization and customization, that can be reused in designing productive programming environments for other application areas. Future work for PyCASP includes completing the PyCASP framework with more components, especially machine learning components such as neural-network training and classification. Furthermore, we are interested in other researchers to try out our software to develop new content analysis applications and contribute to the open source approach. PyCASP source and documentation is available at www.eecs.berkeley.edu/egonina/pycasp.html. Continuation of this work is supported by a full NSF grant and thus we expect a significant increase in the magnitude of this project.

REFERENCES

X. Amatriain, M. D. Boer, and E. Robledo. 2002. Clam: An OO framework for developing audio and music applications. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*.

A. Andoni 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the Annual Symposium on Foundations of Computer Science*. IEEE, 459–468.

X. Anguera, S. Bozonnet, N. W. D. Evans, C. Fredouille, G. Friedland, and O. Vinyals. 2012. Speaker diarization: A review of recent research. *IEEE Trans. Acoust. Speech Signal Process.* 20, 356–370.

K. Asanovic, R. Bodik, et al. 2006. The landscape of parallel computing research: A view from Berkeley. Tech. rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. 1999. *Numerical Python* UCRL-MA-128569. Lawrence Livermore National Laboratory, Livermore, CA.

E. Battenberg and D. Wessel. 2009. Accelerating non-negative matrix factorization for audio source separation on multi-core and many-core architectures. In *Proceedings of the International Symposium on Music Information Retrieval*. K. Hirata, G. Tzanetakis, and K. Yoshii, Eds., International Society for Music Information Retrieval, 501–506.

J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. 2010. Theano: A CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference*.

T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. 2011. The million song dataset. In *Proceedings of the 12th International Symposium on Music Information Retrieval (ISMIR'11)*.

C. M. Bishop. 1995. *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford, UK.

L. S. Blackford, J. Demmel, et al. 2001. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.* 28, 135–151.

J. Carletta. 2007. Unleashing the killer corpus: experiences in creating the multi-everything ami meeting corpus. *Language Resources Eval.* 41, 2, 181–190.

B. Catanzaro, M. Garland, and K. Keutzer. 2010. Copperhead: Compiling an embedded data parallel language. Tech. rep. UCB/EECS-2010-124, EECS Department, University of California, Berkeley.

B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. 2009a. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA'09)*.

B. Catanzaro, B.-Y. Su, N. Sundaram, Y. Lee, M. Murphy, and K. Keutzer. 2009b. Efficient, high-quality image contour detection. In *Proceedings of the IEEE 12th International Conference on Computer Vision*. 2381–2388.

B. Catanzaro, N. Sundaram, and K. Keutzer. 2008. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning (ICML'08)*. ACM, New York, 104–111.

H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. 2011. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, 35–46.

E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui. 2009. Psvm: Parallelizing support vector machines on distributed computers. In *Foundations of Large-Scale Multimedia Information Management and Retrieval*, Springer, 213–220.

C. Charbuillet, D. Tardieu, and G. Peeters. 2011. Gmm supervector for content based music similarity. In *Proceedings of the 14th International Conference on Digital Audio Effects*.

S. Chaudhuri, M. Harvilla, and B. Raj. 2011. Unsupervised learning of acoustic unit descriptors for audio content representation and classification. In *Proceedings of the 11th Proceedings of the Annual Conference of the International Speech Communication Association*.

J. Chaves, J. Nehrbass, B. Guilfoos, J. Gardiner, S. Ahalt, A. Krishnamurthy, J. Unpingco, A. Chalker, A. Warnock, and S. Samsi. 2006. Octave and Python: High-level scripting languages productivity and performance evaluation. In *Proceedings of the HPCMP Users Group Conference*. 429–434.

J. Chong, G. Friedland, A. Janin, N. Morgan, and C. Oei. 2010. Opportunities and challenges of parallelizing speech recognition. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HotPar'10)*. USENIX Association, Berkeley, CA, 2–2.

J. Chong, E. Gonina, Y. Yi, and K. Keutzer. 2009. A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit. In *Proceedings of the 10th Annual Conference of the International Speech Communication Association*.

H. Cook, E. Gonina, S. Kamil, G. Friedland, D. Patterson, and A. Fox. 2011. Cuda-level performance with python-level productivity for Gaussian mixture model applications. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*.

J. Dean and S. Ghemawat. 2008. Mapreduce: simplified data processing on large clusters. *Commun. ACM* 51, 1, 107–113.

B. Elizalde, G. Friedland, H. Lei, and A. Divakaran. 2012. There is no data like less data: Percepts for video concept detection on consumer-produced media. In *Proceedings of the 1st ACM Workshop on Audio and Multimedia Methods for Large-Scale Video Analysis*.

P. Ferraro, P. Hanna, L. Imbert, and T. Izard. 2009. Accelerating query-by-humming on gpu. In *Proceedings of the International Symposium on Music Information Retrieval*. K. Hirata, G. Tzanetakis, and K. Yoshii, Eds., International Society for Music Information Retrieval, 279–284.

G. Friedland, C. Yeo, and H. Hung. 2010. Dialocalization: Acoustic speaker diarization and visual localization as joint optimization problem. *ACM Trans. Multimedia Comput. Commun. Appl.* 6, 27:1–27:18.

E. Gonina, G. Friedland, H. Cook, and K. Keutzer. 2011. Fast speaker diarization using a high-level scripting language. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding*. 553–558.

E. Gonina, A. Kannan, J. Shafer, and M. Budiu. 2011. Parallelizing large-scale data processing applications with data skew: A case study in product-offer matching. In *Proceedings of the 2nd International Workshop on MapReduce and Its Applications (MapReduce'11)*. ACM, New York, 35–42.

T. Goodale, G. Allen, G. Lanfermann, J. Mass, E. Seidel, and J. Shalf. The cactus framework and toolkit: Design and applications. In *Proceedings of the 5th International Conference on High Performance Computing for Computational Science (VECPAR'02)*. Springer, 26–28.

V. W. Gregory. 2000. Programmers tool chest: The OpenCV library. Dr. Dobbs Journal.

E. Grinspun, P. Krysl, and P. Schröder. 2002. Charms: A simple framework for adaptive simulation. *ACM Trans. Graphics* 281–290.

HMM Toolkit web page.

P. Hudak and M. Jones. 1994. Haskell vs. ada vs. c++ vs. awk vs. . . . an experiment in software prototyping productivity. Research Report YALEU/DCS/RR-1049, Department of Computer Science, Yale University, New Haven, CT. Oct.

D. Imseng and G. Friedland. 2009. Robust speaker diarization for short speech recordings. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding*. 432–437.

Intel. Cilk 5.4.6 Reference Manual. Intel. Version 5.4.6.

S. Kamil, D. Coetzee, and A. Fox. 2011. Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization. In *Proceedings of the Python for Scientific Computing Conference*.

K. Keutzer and T. G. Mattson. 2010. A design pattern language for engineering (parallel) software. *Intel Tech. J. 4*.

Khronos Group 2010. OpenCL 1.1 Specification. Khronos Group. Version 1.1.

A. Kosner. 2012. Youtube turns seven today, now uploads 72 hours of video per minute. Forbes.

Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun. 2011. Plda+: Parallel latent dirichlet allocation with data placement and pipeline processing. *ACM Trans. Intell. Syst. Technol.* 2, 3, 26:1–26:18.

L. Lu and A. Hanjalic. 2008. Audio keywords discovery for text-like audio content analysis and retrieval. *IEEE Trans. Multimedia* 10, 1, 74–85.

C. R. Michael Casey and M. Slaney. 2008. Analysis of minimum distances in high-dimensional musical spaces. *IEEE Trans. Audio Speech Lang. Process* 16, 10151028.

F. Mueller. 1995. Pthreads library interface. Florida State University.

NVIDIA Corporation 2010. NVIDIA CUDA Programming Guide. NVIDIA Corporation. Version 3.2.

OpenMP 2008. OpenMP Application Programming Interface. OpenMP. Version 3.0.

A. D. Pangborn. 2010. Scalable data clustering using gpus. M.S. thesis, Rochester Institute of Technology.

L. Prechelt. 2000. An empirical comparison of seven programming languages. *Computer* 33, 10, 23–29.

L. Ramakrishnan, P. T. Zbiegel, et al. 2011. Magellan: experiences from a science cloud. In *Proceedings of the 2nd International Workshop on Scientific Cloud Computing (ScienceCloud'11)*. ACM, New York, 49–58.

D. Reynolds and P. Torres-Carrasquillo. 2005. Approaches and applications of audio diarization. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'05)*. Vol. 5. v/953–v/956.

M. Slaney. 2010. Processing web-scale multimedia data. In *Proceedings of the International Conference on Multimedia*.

N. Sundaram, T. Brox, and K. Keutzer. 2010. Dense point trajectories by gpu-accelerated large displacement optical flow. In *Proceedings of the 11th European Conference on Computer Vision (ECCV'10)*. Springer, 438–451.

G. Takács, I. Pilászy, B. Németh, and D. Tikk. 2009. Scalable collaborative filtering approaches for large recommender systems. *J. Mach. Learn. Res.* 10, 623–656.

G. Tzanetakis, Marsyas submissions to MIREX 2007. In *Proceedings of the 8th International Conference on Music Information Retrieval*.

R. Vuduc, J. W. Demmel, and K. A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *J. Phys. Conf. Ser.* 16, 1, 521.

R. C. Whaley and A. Petitet. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice Experi.* 35, 2, 101–121. http://www.cs.utsa.edu/˜whaley/papers/spercw04.ps.

T. White. 2009. *Hadoop: The Definitive Guide* Ist Ed. O'Reilly.

C. Wooters and M. Huijbregts. 2007. The ICSI RT07s Speaker Diarization System. In *Proceedings of the 2nd International Workshop on Classification of Events, Activities, and Relationships (CLEAR'07) and the 5th Rich Transcription Meeting Recognition (RT'07)*. 509–519.

R. Xia, T. Elmas, S. A. Kamil, A. Fox, and K. Sen. 2012. Multi-level debugging for multi-stage, parallelizing compilers. Tech. rep. UCB/EECS-2012-227, EECS Department, University of California, Berkeley.

K. You, J. Chong, Y. Yi, E. Gonina, C. Hughes, Y. Chen, W. Sung, and K. Keutzer. 2009. Parallel scalability in speech recognition: Inference engine in large vocabulary continuous speech recognition. *IEEE Signal Process Mag.* 6, 124–135.

Received January 2013; revised June 2013; accepted August 2013