PIR: A Domain Specific Language for Multimedia Retrieval

Xiaobing Huang and Tian Zhao
Department of Computer Science
University of Wisconsin – Milwaukee
Milwaukee, USA
{*xiaobing, tzhao*}*@uwm.edu*

Yu Cao
Department of Computer Science
The University of Massachusetts Lowell
Lowell, USA
ycao@cs.uml.edu

***Abstract*—Multimedia retrieval is a problem domain involving salient features extraction, machine learning, indexing, and retrieval. There are a variety of implementations for these tasks, which are difficult to compose and reuse due to the interface and language incompatibility. Because of this low reusability, researchers often have to implement their experiments from scratch and the resulting programs are not optimized for efficiency and cannot be easily adapted for parallelization. In this paper, we present PIR (Pipeline Information Retrieval), a domain specific language (DSL) for multimedia feature manipulation. The goal is to unify the programming tasks for feature-related programming in multimedia retrieval experiments by hiding the programming details under a flexible layer of domain specific interface. This DSL enables us to optimize the feature-related tasks by compiling the DSL programs into pipeline graphs, which can be executed using a variety of strategies to eliminate redundant computation and enable parallelization and change propagation.**

***Keywords*-DSL; pipeline; multimedia information retrieval; parallel programming; Scala**

I. Introduction

Multimedia Information Retrieval (MIR) refers to the research endeavor that centers on searching knowledge from multimedia data. In the last decades, substantial progress has been made in different MIR research areas, such as multimedia feature extraction learning and semantics high performance indexing and query. As a result of substantial progress of MIR research and applications, many related software packages, libraries, and systems have been developed and evaluated using a wide range of multimedia data. Some prominent examples include the GIFT, FIRE, Caliph & Emir, LIRE, ImageTerrier, and OpenIMAJ. While substantial progress in both MIR research and software development have been made, in practice, we have witnessed that code reuse and system composition for MIR research are still very difficult and new systems developed based on existing MIR implementation are not optimized for efficiency and cannot be easily adapted for parallelization, which is essential for handling large multimedia data sets.

This work was supported in part by U.S. National Science Foundation (NSF) under Grant Number 1156639 and 1229213

To address these problems, we introduce a domain specific language: PIR [1] for developing MIR applications. PIR is an embedded DSL using Scala as the host language. Programs written in PIR are plain Scala programs and the DSL implementation is a Scala library. Users can construct MIR applications with simple DSL constructs that focus more on the high-level logic of the MIR application and less on the implementation details for resource management, optimization, and parallelization. PIR provides an abstraction layer over the concrete implementations of various MIR algorithms for feature extraction, machine learning, indexing, and query. This layer separates the construction of a MIR workflow from its execution so that users can choose optimal execution strategies with minimal changes to the program source. Since PIR is embedded in Scala, it is able to utilize the type system of Scala to ensure that PIR programs are well-typed. That is, if a DSL program compiles in Scala, then it will execute according to the DSL semantics. The execution of a PIR program includes two stages. In the first stage, the program is "compiled" into a pipeline graph through a Scala library. In the second stage, the pipeline graph is executed according to a specified optimization strategy.

A. Examples

We demonstrate the utility of our DSL with two examples.

Listing 1: Image Query Example

```
1    img = load("index_image")
2    qImg = load("query_image")
3
4    idx = index(f_luceneIdx,
5                    img.connect(f_cedd),
6                    img.connect(f_fcth))
7    q =  query(f_weightedQuery, idx,
8                    qImg.connect(f_cedd),
9                    qImg.connect(f_fcth))
10   q.out(4)
```

Listing 1 is an example for image retrieval using two global image features: Color and Edge Directivity

[1] A preliminary implementation is at https://github.com/pir-dsl/pir

Descriptor (CEDD) and Fuzzy Color and Texture Histogram (FCTH). This example includes predefined operators and functions for MIR operations. For example, `load("index_image")` loads images from the file path `index_image`. In `img.connect(f_cedd)`, the loaded images are connect to a projection function `f_cedd`, which extracts CEDD features from images. Also, `index` operator creates a Lucene index using `f_luceneIdx` and the extracted features, while `query` operator applies the function `f_weightedQuery` to an index and query features.

The first 4 statements in Listing 1 construct a pipeline graph with each vertex of the graph corresponds to an operation such as image loading, feature extraction, indexing, and querying. The actual computation is delayed until the last statement, which executes the pipeline graph using 4 parallel threads to answer the query `q`. Delayed execution allows the DSL runtime to manage how to execute the function of each pipeline vertex and how the intermediate results of the vertex are stored and reused. In this example, the image loading, feature extraction, and indexing vertices are executed in parallel. The caching of the intermediate results is automatic so that they can be stored in memory if they fit and stored in file system otherwise. Note that the functions such as `f_cedd`, `f_fcth`, and `f_luceneIdx` are wrappers for implementations adapted from the other MIR libraries. Users can expand the capability of the DSL by defining similar functions with appropriate type signatures.

Listing 2: Transmedia Query Example

```
1   txt = load("training_text")
2   img = load("training_image")
3   qImg = load("query_image")
4
5   siftImg = img.connect(f_sift)
6   lModel = train(f_ldaTrain, txt)
7   kModel = train(f_kMeansTrain, siftImg)
8
9   ccaModel = train(f_cca,
10        txt.connect(f_ldaPrj, lModel),
11        siftImg.connect(f_cluster, kModel))
12  p = qImg.connect(f_sift)
13          .connect(f_cluster, kModel)
14          .connect(f_transmedia, ccaModel)
15  p.out(4)
```

The second example (Listing 2) illustrates a transmedia multi-modal retrieval experiment, which uses a `train` operator to apply training algorithms such as LDA, K-Means, and Cross Correlation Analysis (CCA) algorithm to text and image data. The CCA model is trained on clustered image features and topic distributions of text. The example finally applies query image to the trained CCA model to obtain the related text documents. Similar to the first example, the actual computation does not start until the last statement. Note that users can apply method chaining for consecutive feature transformations. For example, line 12–14 in Listing 2 transforms a raw image file to sift features, to a distribution on k-means centroids, and to correlated text documents.

s	::=		Statement
		$x = e$	assignment
	|	$x.f = f$	update
	|	$e.out(n)$	execution
	|	$s; s'$	sequence
e	::=	$src \mid drain \mid pipe$	Expression
src	::=	$load(f)$	load source
$pipe$	::=	$e.connect(f)$	projection
	|	$e.connect(f, e')$	projection with model
$drain$	::=	$train(f, e)$	model
	|	$index(f, e)$	index
	|	$query(f, e_i, e)$	query
f			functions and parameters

Figure 1: Pipeline DSL Language Syntax

We can make changes to the pipeline graph and then run the query again. For example, the following statements changes the file path for the text files and then run the projection `p` again.

```
txt.f = "training_text2"
p.out(4)
```

The second run of `p.out(4)` uses a runtime semantics with change propagation such that the intermediate results of pipeline vertices that depend on `txt`, such as `lModel`, `ccaModel`, and `p`, are recomputed while the results of other vertices, such as `siftImg` and `kModel`, are reused.

II. Syntax

The formal DSL syntax is shown in Figure 1, where a program consists of a sequence of statements and each statement is an assignment, an update, or an execution. Each assignment associates an expression with a local variable. The update statement changes the function (and parameters) associated with a pipeline vertex. The execution statement in $s; e.out(n)$ triggers the execution of the pipeline graph compiled from s and e and outputs the results of the vertex compiled from e with n threads. The expressions include source, drain, and pipe. A source expression loads the raw data files from a directory and we use a function f to represent the load operation with path to the data file embedded in f. A pipe expression projects input data using a function and it may take a model expression as a second parameter. The pipe expression represents the extraction of features such as global/local features of images and term frequency histograms of text documents. The second type of pipe expression is suitable for extracting features such as the LDA topic distribution, which requires a LDA topic model obtained from training. The drain expressions include model, index, and query. The model expression represents the

training of a model. Index and query expressions represent the index and query operations. Each of the drain expression takes a function as parameter. The model expression uses its second parameter as training data. The index expression creates an index for its second parameter. The query expression takes an index e_i and a query input e to compare against the index. Note that it is straightforward to expand the syntax to include cases such as $index(f, e_1, \ldots, e_n)$.

A. Pipeline Graph

The pipeline that we are concerned with can be described with a graph of the edges E, where the vertices are instances of load type V_l, projection type V_p, projection with model type V_{pm}, indexing type V_i, training type V_t, and query type V_q. Each vertex type has a field f to store the function/-parameter that serves the purpose of file loading, feature extraction (projection), training, indexing, or querying. Each vertex type has a field $data$ to store the intermediate results and a flag $isDirty$ to indicate whether the data field of the vertex needs to be updated. Note that not all types of connections between vertices are legal. For instance, it does not make sense to connect V_i to V_p. The DSL relies on Scala's type system to statically rule out incorrect DSL programs that can be compiled into incorrect pipeline graph.

Next, we define the semantics for the DSL programs.

III. From DSL to pipeline graph

We first describe the "compilation" of a DSL program into a pipeline graph. The formal details are omitted due to lack of space. The compilation uses a set of transformation rules, one for each type of expression e and statement s. Each expression rule, $[\![e]\!]\sigma = (v, E)$, transforms an expression e to a new vertex v and a (possibly empty) set of edges E given a runtime state σ that maps variables to vertices. Each statement rule, $[\![s]\!](\sigma, E) = (\sigma', E')$, processes a statement s and changes the state σ to another state σ' while producing a new set of edges E'. A new vertex is created in $v = \mathsf{new}\ V(f)$, where V is the type of the vertex v and f is the function assigned to $v.f$. In the constructor of V, we initialize $v.isDirty$ to true and $v.data$ to empty.

The compilation of a program s can be written as $[\![s]\!](\emptyset, \emptyset) = (\sigma, E)$ with an empty initial state and empty set of edges, where E is the set of pipeline edges compiled from s. The execution statement $e.out(n)$ triggers actual computation by calling $\textsc{execute}(E \cup E', v, n)$, which is a call that starts the execution of the vertex v reduced from e using the edges $E \cup E'$ compiled from the previous statements and e. The call of the form $\textsc{execute}(E, v, n)$ is executed using various algorithms explained in the next section.

IV. Execution of the pipeline graph

After transforming a PIR program into a pipeline graph, we execute the pipeline graph using several variations of execution semantics.

A. Runtime semantics with sequential execution

The algorithm for sequential execution of the pipeline graph computes the results for each vertex after all its predecessors' computation has completed. The algorithm does not run a vertex v's function more than once since its computation results are stored in $v.data$ once the run is completed. The main part of the algorithm is a loop that uses a stack $jobS$ to hold all the vertices that are waiting for results. Initially, we put the starting vertex v in the stack. In the loop, we examine the vertex v on top of the stack to see if it is ready to run. If v's predecessors' data is ready, then we apply the function f of vertex v to the data of all v's predecessors. If the data is not ready, then we push v's predecessor vertices that need to compute results onto the stack and continue. We repeat the loop until $jobS$ becomes empty.

B. Runtime semantics with data and pipeline parallelism

We can improve runtime performance with data parallelism. The algorithm for parallel execution computes the results of a projection or indexing vertex v in parallel by dividing up the input data into n portions. Then it uses n number of threads, where each thread i applies the vertex function f to a portion of the input data in parallel and stores the results in the i element of an array r. After all threads have terminated, the result array r is merged into the final results to store in v. While data parallel execution can speed up computation of projection or indexing vertices, the intermediate data needs to be stored in its entirety, which may result in exhaustion of main memory if $v.data$ is stored in memory or a large number of I/O operations if $v.data$ is stored in file system.

To mitigate the storage problem, we can also implement an execution semantics using pipeline parallelism where we use one thread per vertex. The data of a projection and indexing vertex v is divided into N portions but not all of them needs to be stored. The computation of vertex v proceeds in a loop from $i = 0$ to $N - 1$ such that at step $i = k$, the thread of v checks if the kth portion of v's predecessors is ready: if so, it will compute the kth portion of v's results. If the threads are synchronized by the portion index, then only $1/N$th portion of each vertex's results need to be stored in memory.

C. Runtime semantics with change propagation

In multimedia retrieval research, experiments are often executed many times with various changes to the algorithms, source data, and parameters. Since most experiments take a long time to complete, it is useful to avoid recomputing results that are not affected by the changes in experimental setup. In the DSL syntax, programmers can apply changes to a vertex using the statement $x.f = f$, which may be changes to the source data or changes to the algorithms or parameters for feature extraction, machine learning, indexing, or

querying. If in an update statement $x.f = f$, x is compiled to the vertex v, then we set the flag $v.isDirty$ to true. We define the runtime semantics with change propagation by first inferring the vertices that are affected by the changes based on the dependency relation in the pipeline graph and then recomputing the results of the affected vertices.

V. Implementation with Scala

In general, we can choose any host language to embed our DSL. Practically, Scala is a good choice as it provides advanced features such as lazy evaluation, implicit conversion, closures, mixins, and pattern matching, which greatly simplify our DSL design and are flexible enough for us to plug in and plug out artifacts as we need. For instance, we create vertices (or nodes) in different stages (source, pipe, and drain) with the help of implicit conversion. Moreover, Scala is 100% binary compatible with Java code, which allows use to easily reuse a large number of the existing MIR Java libraries. In our experiments, we used the LIRE image retrieval framework to extract global and local image features, MALLET for LDA (Latent Dirichlet Allocation) modeling, Apache Lucene for indexing and query, and a Java library for Canonical Correlation Analysis for transmedia query in experiments in Section VI.

A. Implicit conversion

Let's consider the example in Listing 1. As explained in Section III, the PIR program is "compiled" into a pipeline graph. In our preliminary implementation, the "compilation" involves a Scala feature called implicit conversion and demonstrated in Listing 3. The first line in Listing 1 will cause a load class `Load` to be instantiated. When line 5 is encountered by the Scala compiler, since the `Load` class does not have a connect method, the compiler will check the global implicit conversion definition and found it in line 1 of Listing 3. Thus, `Load` is first converted to a `LoadNode` v_l and then a new `ProjNode` v_p is created with the edge (v_l, v_p) between the `LoadNode` and the `ProjNode` established (`new ProjNode (this,` f_p`)`).

Listing 3: Implicit Conversion

```
1  implicit def loadToLoadNode(load: Load)
2                        = new LoadNode(load)
3  class LoadNode (load: Load)
4        extends SourceNode with ITask {...}
5  trait SourceNode {
6       def connect(f_p: ProjFunction)
7                   = new ProjNode (this, f_p)
8  ... }
```

VI. Experiments

To show the expressiveness and efficiency of PIR, we ran experiments using a public Wikipedia dataset [1] consisting of 2866 Wikipedia articles (image + text) that spread over 10 categories. Each article comes in a pair, i.e. every image has its corresponding text annotation. This one-to-one mapping is necessary for CCA calculation. We ran tests on a system with Intel Core I5 CPU with 2 cores and 2 hyperthreads per core so that we expect maximal parallel performance using a thread pool of 4 threads. In [2], MIR tasks that involve the combination of two or more different modality data are summarized in three categories, early, late, and transmedia fusion. Our experiments cover two topics – early and transmedia fusion. We skip the late fusion experiment as this is similar to transmedia fusion from execution perspective.

A. Image query with Lucene Index

The first experiment we performed is image query against index (see Listing 1). In this experiment, we used all the 2866 images for execution. The CEDD and FTCH features were extracted from these images. The features were then fed into Lucene engine to generate index. Finally, a query image was supplied to query against the index to retrieve similar images. We showed results using the sequential vs. parallel strategy in Table I.

	image query (unit:sec)	
run	Sequential	Parallel
1	482	304
2	476	303
3	478	303
4	475	299
5	479	305

	transmedia (unit:sec)	
run	Sequential	Parallel
1	240	186
2	232	186
3	233	178
4	233	182
5	234	172

Table I: Runtime of image query and transmedia query

B. Transmedia query with CCA model

In this experiment we performed a transmedia query multimedia information retrieval task (see Listing 2 that is illustrated in Figure 2). We used 1433 (half of the entire

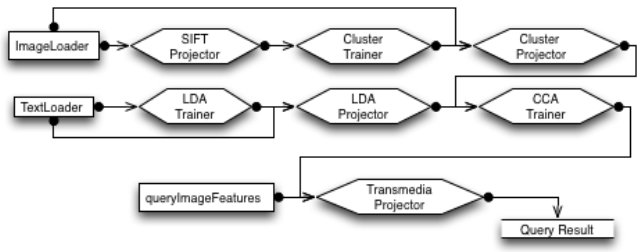

Figure 2: Transmedia Query Graph

dataset) image and text files respectively from all the 10 categories. SIFT features were extracted from the image files. Then a clustering step was applied to all the SIFT features to obtain histogram features (or bag of visual words in some literature). The LDA process was applied to the text files to obtain a LDA model. This model was then applied to all the text files to obtain a probability distribution across the 10 categories for each image. Both the histogram and

distribution data from image and text were used for the CCA computation and ended up in a CCA model. Finally, a query image against the CCA model will result in a list of similar text files (with descending similarity scores) while a query text against the same CCA model will result in a list of similar image files. This is also why this is called transmedia query in the literature. The results are summarized in Table I.

Our results show that our parallel strategy outperforms the sequential counterpart consistently across multiple executions. Note for both of the experiments we only parallelized portion of the pipeline when we applied the parallel strategy. Namely, only source loaders (Image and Text), the projectors (CEDD, FTCH, SIFT and LDA) were parallelized. The training processes and index were still sequentially executed. We argue we will see further performance gain if we apply parallel algorithms for training and indexing in the future.

VII. Related Work

The benefits of DSL are summarized in [3], which also described some common DSL design patterns. A DSL is a special purpose language and thus needs to be designed/tailored for the relevant domain(s). Special domain knowledge/characteristics must be reflected in the DSL and in turn special operations can be adopted. PIR can be associated with several patterns mentioned in [3]. Since PIR is embedded in the host language Scala, it falls under the "piggyback" pattern and naturally inherits all the merits from Scala "for free".

OptiML [4] is designed to provide a parallel DSL for the machine learning (ML) community to bridge the gap between ML algorithms and heterogeneous hardware. Built on top of Delite [5], OptiML performs static domain-specific optimization based on the ML algorithm features such as iterativeness and probabilistic reasoning. Instead of looking for deep optimization of vector/matrix computation that ML algorithms often desire, our pipeline DSL tries to help user concentrate on building the main pipeline structure of the whole algorithm, which forms the backbone of most of the MIR tasks.

Diderot [6] DSL is deigned for image analysis and visualization domain and supports a high-level model of computation that is based on continuous tensor fields. Diderot abstracted out the mathematical details from image processing domain and make that optimizable and parallelizable. Our DSL emphasis on capturing the control flow (the pipeline) in a typical multimedia information retrieval task. Thus, it is a lot easier to locate the critical path, get better parallelization and optimization scheme with pipeline DSL program than with Diderot.

[7] presents a logic-based scripting pipeline language, BANpipe, to model compositions of time consuming analysis. BANpipe creates a pipeline to represent the relayed processing of computation in the biological sequence analysis domain. While BANpipe shares the pipeline concept with PIR, PIR is different from BANpipe in various ways. First, BANpipe works with Prolog for the biological sequence processing while PIR works with Scala in the multimedia information retrieval domain. Second, BANpipe implementation applies a list of chaining processors on files and thus essentially works like a piped batch job processor. In comparison, our DSL implementation follows two steps, translating code into a pipeline graph and implements the graph with different strategies. With such high level pipeline graph abstraction, different optimization, scheduling, and analyzing can be performed on the graph level independent of the detailed operations and/or computation within each node.

VIII. Conclusion

In this paper, we presented a domain specific language, PIR, for implementing MIR applications. PIR provides a high level structure for constructing MIR workflow and hides the details of resource management, optimization, and parallelization. Programs written in our DSL are easy to read and reusable. It is also extensible since PIR is based on a Java compatible host language Scala that allows us to incorporate algorithms implemented in Java-based MIR libraries. As future work, we plan to investigate the applicability of our DSL to a wider range of MIR applications.

References

[1] N. Rasiwasia, J. Costa Pereira, E. Coviello, G. Doyle, G. Lanckriet, R. Levy, and N. Vasconcelos, "A New Approach to Cross-Modal Multimedia Retrieval," in *ACM International Conference on Multimedia*, 2010, pp. 251–260.

[2] S. Clinchant, J. Ah-Pine, and G. Csurka, "Semantic combination of textual and visual information in multimedia retrieval," in *International Conference on Multimedia Retrieval*, 2011.

[3] D. Spinellis, "Notable design patterns for domain-specific languages," *J. Syst. Softw.*, vol. 56, no. 1, pp. 91–99, 2001.

[4] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky, "Optiml: an implicitly parallel domainspecific language for machine learning," in *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[5] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 89–100.

[6] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer, "Diderot: a parallel dsl for image analysis and visualization," in *Proceedings of the 33rd ACM conference on Programming Language Design and Implementation*, 2012, pp. 111–120.

[7] H. Christiansen, C. Theil Have, O. Torp Lassen, and M. Petit, "A declarative pipeline language for complex data analysis," in *Logic-Based Program Synthesis and Transformation*, 2013, pp. 17–34.