OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning

Arvind K. Sujeeth ASUJEETH@STANFORD.EDU
HyoukJoong Lee HYOUKLEE@STANFORD.EDU
Kevin J. Brown KJBROWN@STANFORD.EDU
Hassan Chafi HCHAFI@STANFORD.EDU
Michael Wu MIKEMWU@STANFORD.EDU
Anand R. Atreya AATREYA@STANFORD.EDU
Kunle Olukotun KUNLE@STANFORD.EDU
Stanford University, 353 Serra St., Stanford, CA 94305 USA

Tiark Rompf TIARK.ROMPF@EPFL.CH
Martin Odersky MARTIN.ODERSKY@EPFL.CH
École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland

Abstract

As the size of datasets continues to grow, machine learning applications are becoming increasingly limited by the amount of available computational power. Taking advantage of modern hardware requires using multiple parallel programming models targeted at different devices (e.g. CPUs and GPUs). However, programming these devices to run efficiently and correctly is difficult, error-prone, and results in software that is harder to read and maintain. We present OptiML, a domain-specific language (DSL) for machine learning. OptiML is an implicitly parallel, expressive and high performance alternative to MATLAB and C++. OptiML performs domain-specific analyses and optimizations and automatically generates CUDA code for GPUs. We show that OptiML outperforms explicitly parallelized MATLAB code in nearly all cases.

1. Introduction

Modern machine learning (ML) applications are characterized by large datasets with time-bound computations which require significant amounts of computational power. Recent computer systems featuring a combination of general and specialized processors, so-called "heterogeneous hardware" (AMD, 2008), offer unprecedented computational power and so are well-suited to benefit ML applications. However, taking advantage of these heterogeneous systems is difficult (Sutter, 2005).

To take advantage of all the capabilities of a modern heterogeneous parallel system, machine learning researchers must have expert knowledge in different programming models each aimed at a specific component of modern computing systems: a message passing library for clusters (e.g. MPI), a threaded library to take advantage of parallelism available in a single compute node (e.g. OpenMP) and a data parallel programming model (e.g. CUDA and OpenCL) to take advantage of the GPU. Multiple programming models are needed because no single model is the right choice for all situations. Furthermore, a significant analysis effort is required to match the various parts of the application to the different programming models, and a mix of programming models is required to achieve peak performance (Lee, 2010).

Ideally, ML researchers could leverage these heterogeneous parallel machines with a programming language that is general, productive and results in high-performance execution. However, no such language exists, as generality, productivity and performance are goals that are often at odds. A way to achieve performance and productivity is to give up generality and focus on a particular domain (Chafi et al., 2010). In this paper, we present OptiML, a DSL for developing ma-

Appearing in *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, USA, 2011.

chine learning algorithms and applications. The goal of OptiML is to bridge the gap between ML algorithms and heterogeneous hardware to provide a productive and high performance programming environment.

The main contributions of this paper are:

- OptiML, a new DSL designed to enable machine learning algorithms to easily take advantage of heterogeneous parallelism.
- We show that applications written using domain-specific abstractions can be highly expressive without sacrificing performance.
- We demonstrate ML-specific analyses, optimizations and code generation that are not feasible with a general purpose compiler.
- We show that OptiML without any explicit parallelization outperforms parallel MATLAB and AccelerEyes' Jacket by an average of 3.52x on 8 cores and 3.98x with a GPU.

2. Design

The OptiML language focuses on describing *what* an operation should do, rather than *how* it should do it, deferring the *how* to the language implementation and runtime. OptiML describes ML operations using restricted semantics and data structures that generate efficient parallel and heterogeneous code. In this section, we describe the design and the key features of OptiML.

2.1. Domain Model

OptiML is designed to handle iterative statistical inference problems, in particular those that can be expressed by the Statistical Query Model (Kearns, 1998) which has been shown to cover a large subset of ML algorithms (Chu et al., 2007). These algorithms usually exhibit a combination of regular and irregular data parallelism at varying granularities. OptiML allows these problems to be expressed as dense or sparse linear algebra operations, or as first-class operations on Graph-based data structures. The majority of operations in this model are summation-based (e.g. dot product) and can be parallelized using composable map-reduce operators. However, because ML algorithms typically have many fine-grained operations with low arithmetic intensity, efficiency is more important than in other domains where map-reduce has been traditionally used.

Table 1. Example domain-specific data structures

	Sub-type	Semantics
Matrix	Image	Iteration can access pixels within a window
	Training Set	Only streaming (next, prev) access. Can be file-backed and larger than memory
Vector	Indices	Can be used to index vectors and matrices
	Vertices	Iteration can access neighboring vertices
	Edges	Iteration can access connected vertices
	View	A view of a contiguous section of a matrix Updates propagate to the underlying matrix

2.2. Language Overview

All OptiML programs use data types derived from three fundamental base types: Vector, Matrix, and Graph. These data types are polymorphic and flexible. If they are used with scalar values they will be efficient and leverage BLAS and GPU support for applicable operations. However, they can also be used with other types (e.g. a vector of vectors). Vector, in particular, can be thought of as an array-like container that takes on its mathematical meaning when used with data types that support arithmetic operations. Vector and Matrix support all of the standard linear algebra operations used in most ML algorithms. They also provide a wide range of convenient collection operators, such as *map*, *count*, and *filter*. Subtypes support even richer type-specific operations (e.g. *histogram* on Image). Finally, the Graph type allows machine learning algorithms based around networks and graphical models (such as belief propagation) to be naturally expressed through iteration over vertices and edges.

For convenience, OptiML data types can be used in a normal imperative way (e.g. using a 'while' loop and assigning each index to a value), but this will result in suboptimal parallel performance. Instead, OptiML encourages the use of the domain-specific control and data structures listed in Listing 1 and Table 1. These structures provide the OptiML compiler with additional semantic information while also restricting the operations they support (illegal use of operations will cause a compiler error). By supporting domain-specific access patterns, OptiML can efficiently encode common operations without the performance sacrifices associated with rare and expensive cases. For example, OptiML only allows neighboring vertices in a Graph to be accessed in a bulk operation (e.g. foreach) and automatically synchronizes accesses to these elements while running the operation in parallel. In contrast, a general purpose compiler has to be conservative, because it is unaware of the data structures being used and allows arbitrary memory access patterns to be expressed.

```
/* the following structures are restricted to
   accessing elements with provided index i only: */

// sum: implemented as a parallel tree-reduce
val ans = sum(begin, end){ i =>
  <ith value to sum>  }

// aggregate: returns a concatenated list of results
// implemented as a parallel tree-reduce
val ans = aggregate(v) { i =>
  <ith value to append to buffer> }

// vector construction: implemented as a parallel map
val my_vector = (0::end) { i =>
  <ith value of my_vector>  }

// matrix construction: implemented as a parallel map
val my_matrix = (0::endRow, 0::endCol) { (i,j) =>
  <(i,j)th value of my_matrix> }

/* this structure has type-specific restrictions: */

// unordered iteration over elements:
// implemented as a parallel foreach
for (e <- object) { .. }

/* these structures have no indexing restrictions: */

// untilconverged: implemented sequentially, but can
// be parallelized dynamically using optimizations
untilconverged(x, threshold) { x =>
  <new value of x>  }

// sequential
while(condition) { .. }
```

Listing 1. Pseudocode snippets demonstrating OptiML control structures.

2.3. Using OptiML

We demonstrate OptiML syntax and control structures by showing how the k-means clustering algorithm is written in OptiML:

```
untilconverged(mu, tol){ mu =>

  // calculate distances to current centroids
  val c = (0::m){i =>
    val allDistances = mu mapRows { centroid =>
      // distance from sample x(i) to centroid
      ((x(i)-centroid)*(x(i)-centroid)).sum
    }
    allDistances.minIndex
  }

  // move each cluster centroid to the
  // mean of the points assigned to it
  val newMu = (0::k,*) { i =>
    val (weightedpoints, points) = sum(0,m) { j =>
      if (c(i) == j){
        (x(i),1)
      }
    }
    if (points == 0) Vector.zeros(n)
    else weightedpoints / points
  }

  newMu
}
```

This example highlights the usage of four OptiML control structures: *untilconverged{..}*, *vector constructor (0::m){..}*, *matrix constructor (0::k,*){..}*, and *sum{..}*. Unlike MATLAB functions, each of these control structures accepts any user-defined function that meets the requirements listed in Listing 1. The syntax "x => y" represents a function that takes a value x and returns a value y. *untilconverged* iterates until the difference between `mu` and `newMu` falls below a provided tolerance `tol`. *vector constructor* computes each value of the new `c` vector, which represents the closest cluster for each training sample. *matrix constructor* computes a new $k \times n$ Matrix by computing a new vector for each new cluster location.

These abstractions represent commonly occuring operations in machine learning; in future sections, we will show how we use them to generate high-performance parallel code. This example also shows that vectors (e.g. *x(i)*) and matrices can be used with normal arithmetic syntax. The *mapRows* function is used to perform an operation on every row of a matrix in a concise way. Together, these features allow for programs that resemble pseudocode or scripts. This can be extremely useful during algorithm prototyping, when one wishes to focus on algorithm description rather than implementation details. In the following section, we explore in more detail how OptiML can be used to write more productive code.

3. Productivity

To demonstrate how OptiML's machine learning abstractions can increase programmer productivity and application readability, we compare an OptiML application to a corresponding C++ version. The application we will explore is used for visual object detection on the Willow Garage PR2 robot. The algorithm searches across an image for matches against a database of binary gradient templates and produces a list of object detections and their locations in the image (Bradski & Muja, 2010). The snippet below is used to filter image gradients via non-max suppression:

C++:

```
void gradMorphology(Mat &gradient, Mat &clnGradient) {
  int rows = gradient.rows;
  int cols = gradient.cols;
  // zero out borders
```

```
  uchar *bptrTop = gradient.ptr<uchar>(0);
  uchar *bptrBot = gradient.ptr<uchar>(rows - 1);
  for(int x = 0; x < cols; ++x, ++bptrTop, ++bptrBot)
    *bptrTop = 0; *bptrBot = 0;
  for(int y = 1; y < rows - 1; ++y) {
    uchar *bptr = gradient.ptr<uchar>(y);
    *bptr = 0; *(bptr + cols - 1) = 0;
  }
  // ... 23 lines omitted
  for(int y = 0; y < rows - 2; ++y) {
    // ... 5 lines omitted
    uchar *c = clnGradient.ptr<uchar>(y+1);
    // ... 6 lines omitted
    for(int x = 0; x < cols - 2; ++x, ++c) {
      // ... 3 lines omitted
      int maxindx = 1; int maxcnt = counts[1];
      for(int j = 2; j < 9; ++j) {
        if(counts[j] > maxcnt)
          maxindx = j; maxcnt = counts[j];
      }
      if(maxcnt > 1) { *c = maxindx; }
      // ... 3 lines omitted
    }
  }
}
```

OptiML:

```
def gradMorphology(gradient: GrayscaleImage) = {
  // zero out borders
  gradient.top(1) = 0; gradient.bottom(1) = 0
  gradient.left(1) = 0; gradient.right(1) = 0

  // returns a new Matrix of the same dimensions
  gradient filter(3, 3) { window =>
    val (max,maxIdx) = window.histogram.maxWithIndex
    if (max > 1) maxIdx else 0
  }
}
```

This part of the algorithm does a sliding window computation (filter) over the image, computing a histogram for each window. The output of the filter is the index of the largest histogram value within the window, if the value is greater than 1 (otherwise the output is 0). The C++ code consists of several nested loops that compute the window efficiently, but the resulting code is difficult to read and even harder to parallelize because it contains writes to shared data structures. In contrast, the OptiML code is succinct and expresses only algorithmic intent. The *filter* operation accepts any user-defined function that computes a result without writing to shared data structures. Thus, it is more expressive than MATLAB's fixed function convolution operators, but still restrictive enough to allow OptiML to generate efficient parallel or CUDA code. This example shows how OptiML's restricted semantics lead programmers towards patterns that can be naturally expressed and easily parallelized.

While both the C++ and OptiML versions of the previous example are written sequentially, the OptiML code executes in parallel. It is important to note that the OptiML code does not include implementation details for the target parallel architectures. The same source code currently targets systems with CPUs and/or GPUs, and will also compile to new parallel hardware as support is added to the OptiML compiler. For example, we are currently working on adding support for clusters, where each compute node contains a combination of CMPs and GPUs; OptiML programs will automatically inherit this support. In contrast, with MATLAB a developer must choose up front whether to write a loop to be run in parallel (using *parfor*), on a GPU (using *gfor* with Jacket), or in its most efficient sequential form (using *vectorization*). This decision is difficult or impossible to make statically, because the right choice depends on factors such as data size, parallelization overhead, and the availability and characteristics of the target hardware.

4. Performance

OptiML uses domain-specific knowledge in order to reason about programs at a higher level than a library or a general purpose language, which enables it to provide superior parallel performance. At the heart of OptiML's ability to deliver high performance is its ability to build, analyze, and optimize an intermediate representation (IR) of the user program.

4.1. Building an IR

OptiML is a domain-specific language *embedded* in Scala (Odersky, 2011), (Hudak, 1996), (Hofer et al., 2008). This means that OptiML implements its compiler in Scala, and OptiML programs are also valid Scala programs. Scala is a JVM-based language with advanced features for DSL embedding. OptiML uses a metaprogramming technique known as *lightweight modular staging* (Rompf & Odersky, 2010) to build an intermediate representation of a program. Instead of directly executing the operations expressed by the program, OptiML records a node representing the operation and its dependencies (data and control). These nodes encode both the domain-specific nature of the operation (e.g. outer product, matrix transpose) and multiple mappings to efficient parallel execution patterns. Once the application is in this form, it becomes amenable to analysis and optimization.

4.2. Analyses and Optimizations

OptiML performs several static and dynamic optimizations. Static optimizations are applied as transformations on the OptiML IR before code generation, while

dynamic optimizations are implemented as part of OptiML data types or control structures.

Static domain-specific optimizations:

OptiML implements general and well-known static optimizations such as common subexpression elimination, dead code elimination, and loop hoisting. All of these optimizations occur at the granularity of DSL operations (e.g. vector plus). It also provides the following domain-specific optimizations:

Pattern rewriting: OptiML uses pattern matching on the IR to optimize sequences of operations according to standard linear algebra simplification rules. For example, a simplification that can be exploited in Gaussian Discriminant Analysis (GDA) is:

$$\sum_{i=0}^{n} \vec{x}_i * \vec{y}_i \rightarrow \sum_{i=0}^{n} X(:,i) * Y(i,:) = X * Y$$

This optimization converts a summation of outer products into a single matrix multiplication. We also use pattern rewriting to identify sequences of operations that should be generated differently depending on target device, consolidating them into a single IR node that can be generated accordingly.

Op fusing: OptiML classifies domain operations according to their implementation semantics. For example, it knows that a vector + vector operation is a ZipWith operation, i.e. a simultaneous loop over the two vectors that will build a new vector as a result. This knowledge is then used to fuse adjacent operations; for example, k loops that iterate over data structures of the same size can be transformed into a single loop that computes k results, thereby reducing the number of main memory accesses. As a practical example, consider the following line from the SMO algorithm (Platt, 1998) for SVM (`*:*` is a dot product):

```
val eta = (X(i)*:*X(j)*2) - (X(i)*:*X(i))
          - (X(j)*:*X(j))
```

Here, OptiML automatically fuses all of the dot product calculations into a single loop instead of 4 (3 for each dot product plus 1 for the scalar multiplication). For the entire SMO algorithm, op fusing reduces 35 loops to 11. More importantly, fusing an operation can eliminate allocations of intermediate data structures. The `*:*` operator can be implemented as `(X(i) * X(j)).sum` and op fusing will ensure that no intermediate vector will be allocated to hold the result of `X(i) * X(j)`.

Dynamic domain-specific optimizations:

Best-effort computing: because many ML algorithms are iterative and probabilistic, they are often robust to minor variations in computation (Meng et al., 2009). OptiML allows users to trade-off accuracy, if they choose, for better performance, by using best effort data structures. These data structures drop computations according to a policy, which can improve single-threaded execution time and reduce sequential bottlenecks, improving parallel scalability.

Relaxed dependencies: for the same reasons as above, it is sometimes useful to allow ML algorithms to intentionally race, which again can improve parallel performance at the expense of strict consistency for some operations. OptiML provides a version of the *untilconverged* construct that allows some number of iterations to be run in parallel. Recent work (Zinkevich et al., 2010) has shown the potential for this optimization.

4.3. Code Generation

OptiML generates code by translating IR nodes to their corresponding implementation in the target language. Currently, OptiML generates Scala, C++ and CUDA code, although not all targets are generated for every operation. There are some operations that do not fit the CUDA programming model (e.g. operations that dynamically allocate memory with unknown sizes or use complex reference-based data structures); OptiML will only generate Scala or C++ code for these. Along with generated code for each node, we emit an execution graph describing the program's operations and dependencies. OptiML is built on top of a common runtime, Delite (Chafi et al., 2011), that is designed specifically for heterogeneous parallel DSLs. An OptiML program is executed by invoking Delite with the execution graph and generated code. Delite schedules the OptiML operations on the underlying hardware and provides synchronization and communication between kernels.

The restricted semantics of OptiML control structures and the compiler's knowledge of the possible data structures being used are the key factors that enable the OptiML compiler to target the same application code to both CMPs and GPUs. Since OptiML knows what data types an operation uses, it can transparently generate the required CUDA data structures and memory transfers. The compiler is able to generate code without extensive analysis because the restrictions of the constructs guarantee the transformation is safe. Furthermore, OptiML is able to generate code that is optimized for each device; for example, GPUs prefer to stride column-wise through a row-major Matrix as this maximizes the global memory bandwidth utilization by coalescing the memory requests from

multiple threads, whereas CPUs prefer to stride row-wise to maximize single-threaded locality. OptiML generates the appropriate stride for both cases, maximizing performance per device.

OptiML's heterogeneous code generation is essential to providing a portable, productive programming model with the best possible performance. Some operations either do not fit the CUDA model or actually perform worse on a GPU, and it is necessary to be able to run these on a CPU. Since application programmers do not specify any device-specific details, OptiML can generate multiple versions and select the best one at runtime. In contrast, with both MATLAB and AccelerEyes' GPU support, the programmer must specify which data structures should be resident on the GPU device memory. Choosing which data structures should go to the GPU and deciding when to bring the data back to obtain the best performance is difficult.

4.4. Implications for OptiML Users

The fact that OptiML is embedded in Scala and executed on heterogeneous systems is ideally completely transparent to the end user. From a syntactic and semantic point of view, this illusion is maintained; OptiML programs require no knowledge of the underlying embedding implementation, no explicit parallelization, and no explicit code for the lower level programming models (e.g. CUDA). However, there are still some issues that can make the underlying implementation visible to OptiML programmers. The most important issue is debugging; errors in generated code at runtime can be tricky to capture and propagate back to the user in a meaningful way. Similarly, Scala compilation errors should be trapped and reported to the OptiML user in a simplified way. We believe these issues are technical rather than fundamental, and we are addressing them in our current and future research.

5. Evaluation

This section presents performance results for a set of machine learning applications written in OptiML and compares them to reference implementations written using existing alternative systems, including MATLAB, GraphLab (Low et al., 2010), and C++. In addition, we analyze the performance improvements achievable due to OptiML's static and dynamic optimizations described in Section 4.2.

5.1. Methodology

For the first set of experiments, we compare our applications to multiple MATLAB implementations. We used MATLAB 7.11 with its CPU parallelization and GPU support, as well as GPU support from AccelerEyes's Jacket (AccelerEyes, 2010). Each application is algorithmically identical, but for the MATLAB versions we made a reasonable effort to vectorize and parallelize the CPU code, and make the best data locality choices for the GPU. In cases where both vectorization or parallelization was possible, we report the results for the version that had the best performance at 8 CPUs.

We also present two ML applications that are not well suited to MATLAB, and therefore chose an alternative language to provide a performance baseline. We implemented a version of loopy belief propagation (LBP) in OptiML and compare it to a baseline implementation in GraphLab, which is a C++ library for ML graph applications. We also compare the binarized gradient template matching (TM) algorithm described in section 3 to a hand-optimized C++ baseline.

For each of the experiments we timed the computation component of the application, without initialization. We ran each application (with initialization) 10 times in order to warm up the JIT and smooth out fluctuations due to garbage collection and other variables. We present here the averaged time of the last five executions. We used a Dell Precision T7500n with two quad-core Intel Xeon X5550 2.67 GHz processors, 24GB of RAM, and an NVidia Tesla C2050.

5.2. Performance Comparison

In Figure 1 we compare the performance of our applications to the MATLAB implementations. Execution time for each application is normalized to the single-threaded OptiML version. The MATLAB parallel constructs use MPI, which adds significant overhead, while the single-threaded OptiML version is generated without any parallelization overhead. We also ran purely sequential MATLAB versions and found OptiML to have equivalent or better performance for each application. In most cases, OptiML performs better because it is statically compiled and generates optimized code instead of being interpreted. In some cases, such as deep belief learning with a Restricted Boltzmann Machine (RBM), both OptiML and MATLAB are predominantly calling native BLAS libraries (e.g. matrix multiplication), and so achieve roughly the same performance.

For some applications, offloading operations to the GPU results in a significant performance improvement, while for others it merely adds overhead, leading to performance equivalent to the CPU (or sometimes worse in MATLAB's case). For GDA, OptiML's substantially better GPU performance is due to its ability

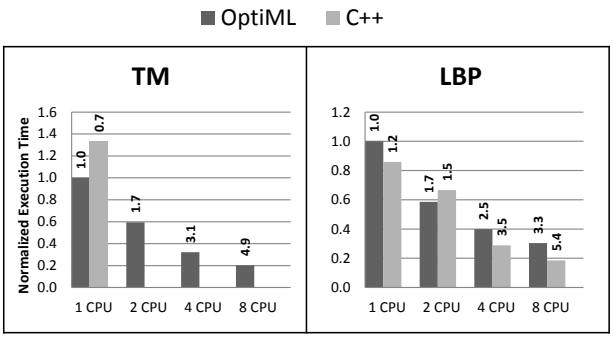

Figure 2. Execution time of our applications compared to C++, and normalized to single-threaded OptiML. Speedup numbers are reported on top of each bar.

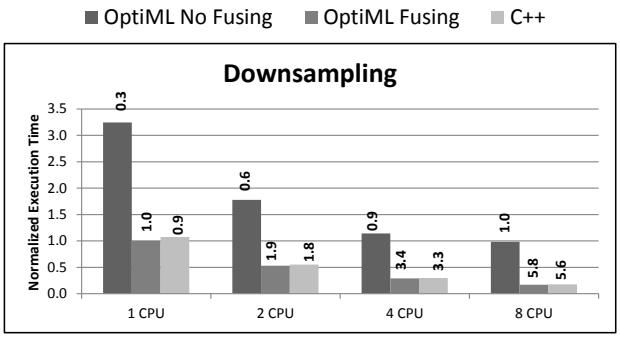

Figure 3. Normalized execution time of Downsampling in C++ and OptiML with and without op-fusing optimizations. Speedup numbers are reported on top of each bar.

to analyze the application IR and generate a CUDA kernel that minimizes memory accesses by exploiting local GPU registers.

Figure 2 compares the performance of two applications written in OptiML to a baseline written in an alternative environment. LBP is compared against an equivalent implementation in GraphLab. The results show that OptiML achieves performance and scaling close to GraphLab, which is written in C++ and designed specifically for Graph-based algorithms. TM is compared to a baseline C++ implementation which is single-threaded and designed for high performance robotics. The OptiML version outperforms the C++ version while being significantly shorter and easier to read. It also scales to 8 cores, while the C++ baseline would require non-trivial manual parallelization to achieve any parallel performance.

5.3. Impact of Optimizations

Section 4.2 described multiple static and dynamic domain-specific optimizations. The performance results presented in Figures 1 and 2 included the static optimizations of common subexpression elimination, dead code elimination, code motion, and linear algebra rewrites. We studied the benefits of op fusing on the downsampling part of a bioinformatics application. This part of the application streams over a large dataset, performing multiple operations on each sample. Without optimization, the OptiML version is 3x slower than a hand-optimized, manually-parallelized C++ version, as shown in Figure 3. After fusing, the OptiML version is approximately as fast as the C++ version.

We next look at the additional improvement from applying relaxed dependencies to SVM and best-effort computation to k-means. The SMO implementation of SVM contains inter-loop dependencies that prevent parallelization across iterations. In previous work we have shown that relaxing dependencies between the outer loop iterations, allowing iterations to sometimes run in parallel, can increase performance by 1.8x with less than 1% loss in classification accuracy (Chafi et al., 2011). For k-means we demonstrated that a best-effort convergence policy that drops distance calculations that have remained unchanged in the previous n iterations creates a unique tradeoff between accuracy and performance for different values of n. Specifically, we observed speedups of 1.8x with a 1.2% loss in accuracy, 4.9x with a 4.2% loss, and 12.7x with a 7.4% loss in accuracy (Chafi et al., 2011).

6. Conclusion

Many useful machine learning algorithms and datasets are facing computational challenges and will require the use of state-of-the-art hardware. As the size of datasets continues to grow and hardware becomes even more parallel and heterogeneous, being able to exploit these hardware features will become essential. OptiML provides the link between ML applications and heterogeneous parallel hardware. We demonstrated the productivity of OptiML code with an ML application for robotics. We have shown that the OptiML compiler can perform domain-specific optimizations and generate efficient parallel code for heterogeneous devices without exposing any parallelism or device details to OptiML users. Finally, we presented experimental results showing that OptiML code outperformed explicitly parallelized MATLAB code on a heterogeneous system consisting of multicore CPUs and a GPU.

7. Acknowledgements

We would like to thank Andrew Saxe and Andrew Ng for valuable discussions and comments about this

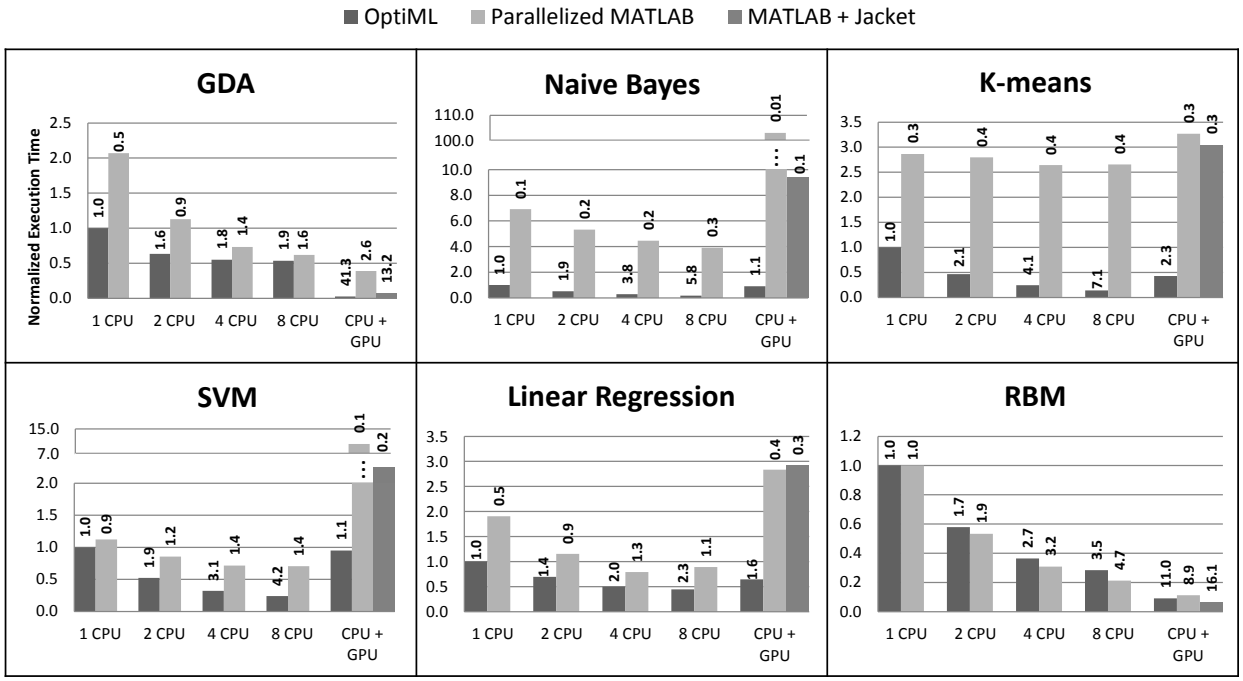

Figure 1. Execution time of our applications compared to MATLAB, and normalized to single-threaded OptiML. Speedup numbers are reported on top of each bar.

work. This research was sponsored by Army contract AHPCRC W911NF-07-2-0027-1; DARPA contract, Oracle order US1032821; the Stanford PPL affiliates program, Pervasive Parallelism Lab: NVIDIA, Oracle/Sun, AMD, NEC, and Intel; and the following Fellowship programs: SGF, SOE, and KFAS.

References

AccelerEyes. Jacket. http://www.accelereyes.com, 2010.

AMD. The Industry-Changing Impact of Accelerated Computing. White Paper, 2008.

Bradski, G. and Muja, M. BiGG Detector. http://www.ros.org/wiki/bigg_detector, 2010.

Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A. K., Hanrahan, P., Odersky, M., and Olukotun, K. Language Virtualization for Heterogeneous Parallel Computing. Onward!, 2010.

Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R., and Olukotun, K. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, 2011.

Chu, C., Kim, S., Lin, Y., Yu, Y., Bradski, G., Ng, A. Y., and Olukotun, K. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19*. 2007.

Hofer, C., Ostermann, K., Rendel, T., and Moors, A. Polymorphic embedding of DSLs. GPCE, 2008.

Hudak, P. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996. ISSN 0360-0300.

Kearns, M. Efficient noise-tolerant learning from statistical queries. *Joural of the ACM*, 45:983–1006, 1998.

Lee, V. W. et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proc. ISCA*, 2010.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. GraphLab: A New Parallel Framework for Machine Learning. In *UAI*, 2010.

Meng, J., Chakradhar, S., and Raghunathan, A. Best-effort parallel execution framework for recognition and mining applications. In *Proc. of IPDPS*, 2009.

Odersky, M. Scala. http://www.scala-lang.org, 2011.

Platt, J. C. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998.

Rompf, T. and Odersky, M. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. GPCE, 2010.

Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30 (3):202–210, 2005.

Zinkevich, M. A., Weimer, M., Smola, A., and Li, L. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, December 2010.