Implementing Domain-Specific Languages for Heterogeneous Parallel Computing

Domain-specific languages offer a solution to the performance and the productivity issues in heterogeneous computing systems. The Delite compiler framework simplifies the process of building embedded parallel DSLs. DSL developers can implement domain-specific operations by extending the DSL framework, which provides static optimizations and code generation for heterogeneous hardware. The Delite runtime automatically schedules and executes DSL operations on heterogeneous hardware.

HyoukJoong Lee
Kevin J. Brown
Arvind K. Sujeeth
Hassan Chafi
Kunle Olukotun
Stanford University

Tiark Rompf
Martin Odersky
École Polytechnique Fédérale de Lausanne

Power constraints have limited the ability of microprocessor vendors to scale single-core performance with each new generation. Instead, vendors are increasing the number of processor cores and incorporating specialized hardware to improve performance.[1] For example, GPUs have become essential components of modern systems because of their massively parallel computing capability.[2,3] However, the advent of heterogeneous systems creates the problem of having too many programming models. Common examples are Pthreads or OpenMP for multicore CPU, OpenCL or CUDA for GPU, and message passing interface (MPI) for clusters. As a result, application programmers pursuing higher performance must have expertise not only in the application domain but also in disparate parallel programming models and hardware implementations. In addition, the relative performance improvement is difficult to predict until the program is written and executed in different models with elaborate optimization phases that might also depend on the input data. Even worse, the optimized code for one system is neither portable nor guarantees high performance on another system. Ideally, application programmers shouldn't have to manage low-level implementation details, so they can focus on algorithmic description and still obtain high performance.

A successful parallel programming model should have three broad characteristics (known as the three Ps): productivity, performance, and portability. To provide productivity, a parallel programming model must raise the level of abstraction above that of current low-level programming models such as Pthreads and CUDA. Ideally, such a programming model would also be general, allowing the application developer to express arbitrary semantics.

0272-1732/11/$26.00 © 2011 IEEE

However, despite decades of research, no such programming model exists. Instead, it seems that generality, productivity, and performance are usually at odds with one another, and successful programming languages must carefully trade them off. For example, C++ is general and high performance, but usually not considered as productive as higher-level languages such as Python and Ruby. On the other hand, Python and Ruby can't compete with C++ in terms of performance. Another approach is to focus on performance and productivity while trading off generality—for example, by focusing on a particular domain using domain-specific languages (DSLs).[4,5] The ability to exploit domain knowledge to pursue high performance and productivity make DSLs an ideal platform for attacking the heterogeneous parallel programming problem. However, making the DSL approach useful on a large scale requires lowering the barrier for DSL development.

To facilitate development of embedded parallel DSLs, we designed the Delite compiler framework. DSL developers can implement domain-specific operations by extending the DSL framework, which provides static optimizations and code generation for heterogeneous hardware. The Delite runtime automatically schedules and executes DSL operations on heterogeneous hardware. We evaluated this approach with OptiML, a machine-learning DSL, and found significant performance benefits when running OptiML applications on a system using multicore CPUs and GPU.

Domain-specific languages

DSLs provide carefully designed APIs that are easy for developers in the domain to use. DSL code often more closely resembles pseudocode than C code, deferring most if not all of the implementation details to the language. This deferral of responsibility lets the DSL developer use the most efficient parallel implementation and target different devices transparently to the application. This is feasible only because the DSL doesn't try to do everything; instead, it tries to do a few things very well. DSLs provide a structured foundation to identify and exploit parallel execution patterns specific to particular domains. They aren't a silver bullet, however. They can't parallelize existing sequential code, and they don't on their own eliminate the parallel programming burden for the DSL developer.

For parallel DSLs to be tractable, they must be easy to create for many domains. Traditionally, there are two types of DSLs.

Internal DSLs are embedded in a host language, and are sometimes called the "just-a-library" approach.[6] These DSLs typically use a flexible host language to provide syntactic sugar over library calls. Although this is the easiest approach (no additional compiler is necessary), it constrains the DSL's capabilities. A purely embedded, or library-based, DSL can't build or analyze an intermediate representation (IR) of user programs. Thus, such DSLs can only perform dynamic analyses and optimizations, and these handicaps can severely impact achievable parallel performance. More importantly, without an IR, DSLs can't do their own code generation, which prevents retargeting DSL code to heterogeneous devices.

External DSLs are implemented as a stand-alone language.[7] Although these DSLs obviously don't have the limitations of internal DSLs, they're extremely difficult to build. The DSL developer must define a grammar and implement the entire compiler framework as well as tooling to make it useful (for example, IDE support). This clearly isn't a scalable approach.

Our hybrid approach addresses this dilemma. We use the concept of *language virtualization*[8] to characterize a host language that lets us implement embedded DSLs that are virtually indistinguishable from stand-alone DSLs. A virtualizable host language provides an expressive, flexible front end that the DSL can borrow, while letting the DSL leverage metaprogramming facilities to build and optimize an IR. Figure 1 demonstrates this separation.

Language virtualization is an effective way to define and implement DSLs inside a sufficiently flexible host language. However, building parallel DSLs adds new challenges, such as implementing parallel patterns, launching and scheduling parallel tasks, synchronizing

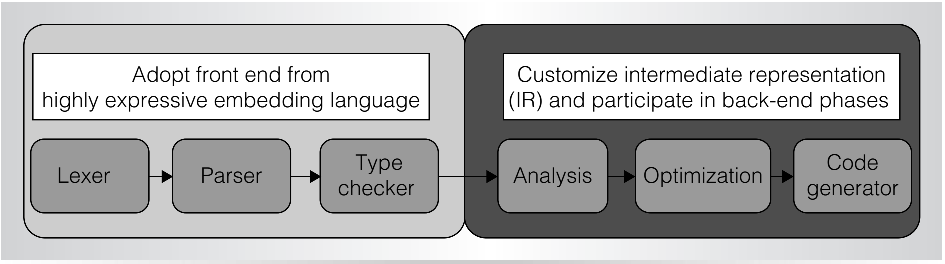

Adopt front end from
highly expressive embedding language
Customize intermediate representation
(IR) and participate in back-end phases
Lexer
Parser
Type checker
Analysis
Optimization
Code generator

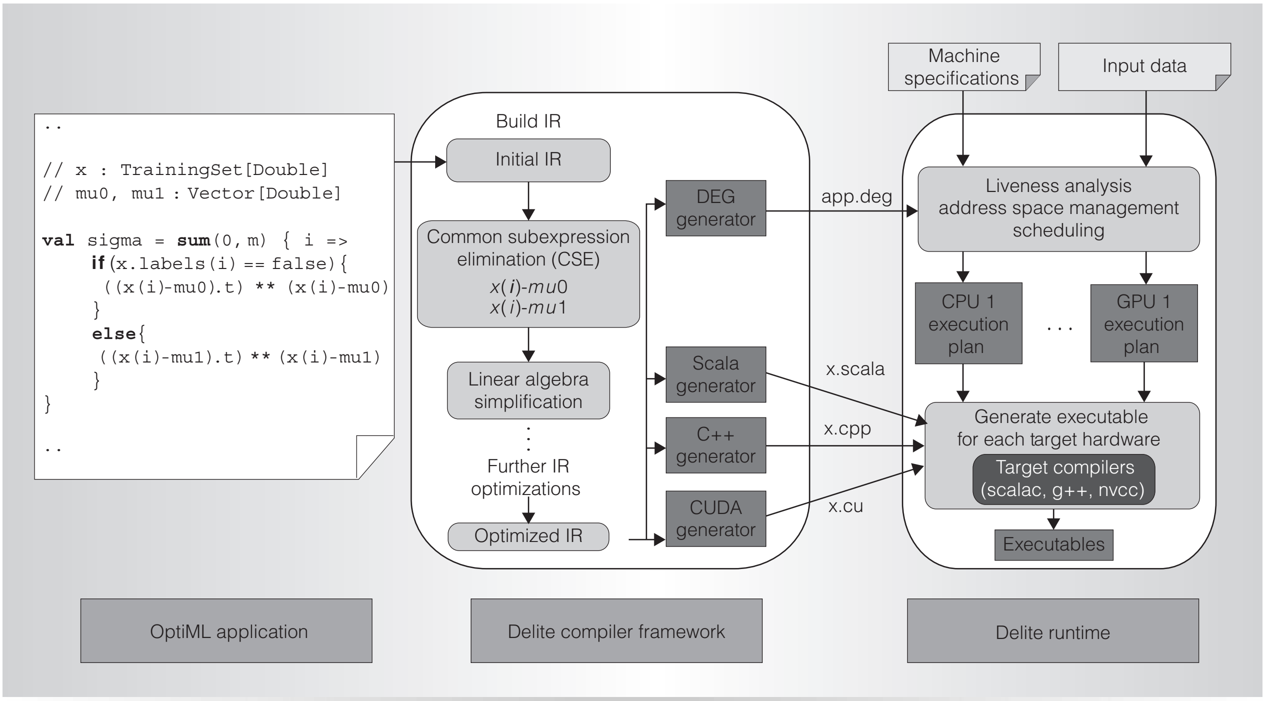
Figure 1. Typical compilation phases and the language virtualization approach to embedded domain-specific languages. This approach lets the DSL adopt the front end from a highly expressive embedding language, while customizing its intermediate representation and participating in the back-end phases of its compilation.

communication, and managing multiple address spaces. Therefore, we implemented a reusable compiler infrastructure and runtime (the Delite compiler framework and runtime) to make developing parallel DSLs even easier. The framework provides common parallel execution patterns, and DSL developers can easily implement a DSL operation by mapping it to one of these patterns. This mapping process requires minimum effort because most of the behavior is already encoded in the pattern. For example, to implement an operation that iterates over a collection and updates each element by applying a given function, the DSL developer needs only to specify the function behavior. The framework and runtime manage the boilerplate code (for example, iterating over the loop, updates, and parallelization). The framework also provides code generators for those patterns, so the DSL can target heterogeneous hardware (multicore CPU, GPU, and so on) without developing code generators for each target. In addition, the runtime manages efficient scheduling, exploiting both task and data parallelism with proper synchronization—tasks that used to be additional burdens on the DSL developer. Therefore, DSL developers can focus on the language design rather than the implementation details and can exploit heterogeneous parallel performance without writing any low-level parallel code.

OptiML and Delite

Figure 2 shows the overall operation of the Delite compiler framework and runtime. It gives an example of OptiML,[9] a machine-learning DSL developed with our framework. OptiML provides matrix, vector, and graph data structures; domain-specific operations, including linear algebra; and domain-specific control structures such as *sum*, which is used in the code snippet in Figure 2. The *sum* construct accumulates the result of the given block for every iteration (0 to *m*) and is implemented by extending the framework's `DeliteOpMapReduce` parallel pattern.

To generate optimized executables from the high-level representation of the DSL operations, the Delite compiler builds an IR of the application and applies various optimizations on the IR nodes. For example, because the two vector minus operations $(x(i) - mu0)$ within the *sum* are redundant, common subexpression elimination removes the latter operation by reusing the former result. After building the optimized IR, the framework's code generators automatically emit computation kernels for both the CPU and the GPU. When all of the application kernels have been generated along with the Delite execution graph (DEG), which encodes the data and control dependencies of the kernels, the Delite runtime starts analyzing the DEG to schedule the kernels and generate an execution plan for each target. The execution plan consists of the kernel calls scheduled on the target with necessary memory transfers and synchronization. Finally, target language compilers (such as Scala and CUDA) compile and link the kernels from the framework with the execution plans from the runtime to generate executable files that run on the system.


```
..

// x : TrainingSet[Double]
// mu0, mu1 : Vector[Double]

val sigma = sum(0, m) { i =>
     if(x.labels(i) == false){
      ((x(i)-mu0).t) ** (x(i)-mu0)
     }
     else{
      ((x(i)-mu1).t) ** (x(i)-mu1)
     }
}

..
```

Figure 2. Operation of the Delite compiler framework and runtime. The application written in OptiML is transformed into an IR and optimized by the framework to generate the kernels for heterogeneous targets along with the Delite execution graph. The runtime schedules the kernels on heterogeneous hardware for parallel execution. (DEG: Delite execution graph.)

Building embedded parallel DSLs

For a high-performance DSL to target heterogeneous parallel systems, its IR should have at least three major characteristics:

- It should be able to accommodate traditional compiler optimizations on DSL operations and data types.
- It should expose common parallel patterns for structured parallelism.
- It should encode enough domain information to allow implementation flexibility and domain-specific optimizations.

The Delite compiler framework, a reusable compiler infrastructure for developing DSLs, structures the intermediate representation of DSLs in a way that meets these requirements.

Intermediate representation

A single IR node can be viewed from three perspectives, which provide different optimizations and code-generation strategies. We built the Delite compiler framework using the concept of a multiview IR, as Figure 3 illustrates.

The most basic view of an IR node is a symbol and its definition (the *generic IR* view), which is similar to a node in a traditional compiler framework's flow graph. We can therefore apply all the well-known static optimizations at this level. The primary difference is that our representation has a coarser granularity because each node is a DSL operation rather than an individual instruction, and this often leads to better optimization results. For example, we can apply the common subexpression elimination (CSE) to the vector operations ($x(i) - mu0$, $x(i) - mu1$), as Figure 2 shows, instead of just to scalar operations. Currently applied optimizations include CSE, constant propagation, dead code elimination, and code motion.

A generic IR node can be characterized by its parallel execution pattern (the *parallel IR* view). Therefore, on top of the generic IR view, the Delite compiler framework

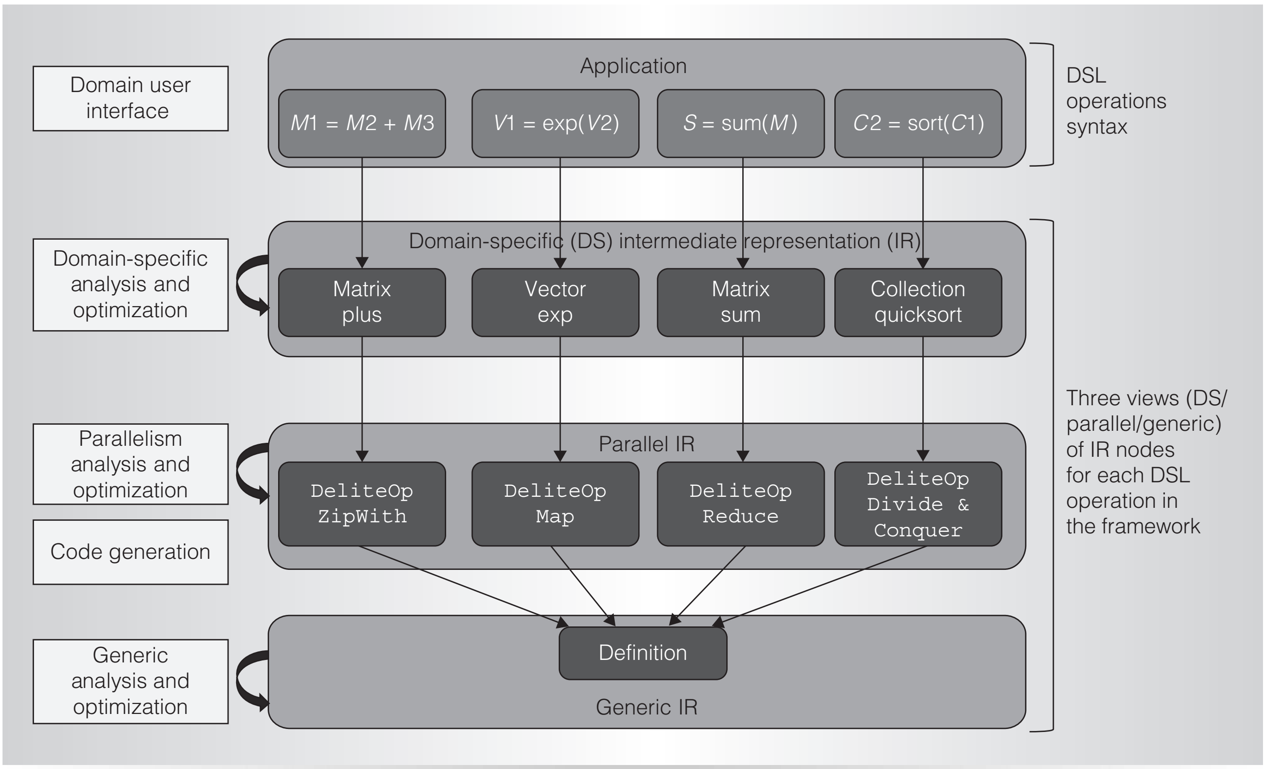

Domain user interface
Application
M1 = M2 + M3
V1 = exp(V2)
S = sum(M)
C2 = sort(C1)
DSL operations syntax
Domain-specific analysis and optimization
Domain-specific (DS) intermediate representation (IR)
Matrix plus
Vector exp
Matrix sum
Collection quicksort
Three views (DS/parallel/generic) of IR nodes for each DSL operation in the framework
Parallelism analysis and optimization
Parallel IR
DeliteOp ZipWith
DeliteOp Map
DeliteOp Reduce
DeliteOp Divide & Conquer
Code generation
Generic analysis and optimization
Definition
Generic IR

Figure 3. Multiview of IR nodes in the Delite compilation framework. Here, the matrix addition operation ($M1 = M2 + M3$) is represented as a MatrixPlus IR node, which extends the `DeliteOpZipWith` IR node, which again extends the Definition IR node. The framework uses the generic IR view for traditional compiler optimizations, the parallel IR view for exposing parallel patterns and loop-fusing optimizations, and the domain-specific IR view for domain-specific optimizations.

provides a finite number of common structured parallel execution patterns in the form of `DeliteOp` IR nodes. Examples include `DeliteOpMap`, which encodes disjoint element access patterns without ordering constraints, and `DeliteOpForeach`, which allows a DSL-defined consistency model for overlapping elements. The `DeliteOpSequential` IR node is for patterns that aren't parallelizable. Because certain parallel patterns share a common notion of loops, we can fuse multiple loop patterns into a single loop. The parallel IR optimizer iterates over all the IR nodes of the various loop types (`DeliteOpMap`, `DeliteOpZipwith`, and so on), and fuses those with the same number of iterations into a single loop. This optimization removes unnecessary memory allocations and improves cache behavior by eliminating multiple passes over data, which is especially useful for memory-bound applications.

In addition to the parallel execution pattern, each domain operation has its own semantic information encoded in the corresponding domain-specific IR node (the *domain-specific IR* view). This view enables the framework to apply domain-specific optimizations, such as linear algebra simplifications, through IR transformations. The transformation rules are simply described by pattern matching on domain-specific IR nodes, and the optimizer replaces the matched nodes with a simpler set of nodes. Examples of the transformation on matrix operations include $(A^t)^t = A$ and $A \cdot B + A \cdot C = A \cdot (B + C)$. The separation of the domain-specific IR from the parallel IR makes it easy to design implicitly parallel DSLs, abstracting parallel execution patterns away from DSL users.

This multiview IR greatly simplifies the process of developing a new DSL because all DSLs can reuse the generic IR and parallel IR in the framework, so DSL developers only need to design a domain-specific IR for each domain operation as an extension. Because DSL developers have expertise in the parallel execution pattern of each domain operation in the DSL, they extend the appropriate Delite parallel IR node to create a domain-specific IR for each operation. In other words, DSL developers are only exposed to a high-level parallel instruction set (the parallel IR nodes), and the Delite compiler framework automatically manages the implementation details of each pattern on multiple targets.

To build the IR from a DSL application, the Delite compiler framework uses the *lightweight modular staging* (LMS) technique.[10] As the application starts executing within the framework, the framework translates each operation into a symbolic representation to form an IR node rather than actually executing the operation. The IR nodes track all dependencies among one another, and the framework applies the various optimizations on the IR as mentioned earlier. After building the machine-independent optimized IR, the Delite compiler framework starts the code-generation phase to target heterogeneous parallel hardware.

Heterogeneous target code generation

Generating a single binary executable for the application at compile time limits the application's portability and requires runtime and hardware systems to rediscover dependency information to make machine-specific scheduling decisions. The Delite compiler framework defers such decisions by generating kernels for each IR node in multiple target programming models as well as the DEG describing the dependencies among kernels. Currently supported targets are Scala, C++, and CUDA.

DEG and kernel generation. The Delite generator controls multiple target generators. It first schedules IR nodes to form kernels in the DEG, and iterates over the list of available target generators to generate corresponding target code for the kernel. Because of the restrictions of the target hardware and programming model, it might not generate the kernel for all targets, but the kernel generation will succeed as long as at least one target succeeds. As each IR node has multiple viewpoints, kernels can be generated differently for each view. For example, a matrix addition kernel could be generated in the domain-specific view code generator written by a DSL developer, but also in the parallel view because the operation is implemented by extending the `DeliteOpZipWith` parallel IR. Because the Delite compiler framework provides parallel implementations for the parallel IR nodes `(DeliteOps)`, DSL developers don't have to provide code generators for the DSL operations that extend one of them. When DSL developers already have an efficient implementation of the kernel (such as basic linear algebra subroutine [BLAS] libraries for matrix multiplication), they can generate calls to the external library using `DeliteOpExternal`.

GPU code generation. GPU code generation requires additional work because the programming model has more constraints than the Scala and C++ targets. Memory allocation, for example, is a major issue. Because dynamic memory allocation within the kernel is either impossible or impractical for performance in GPU programming models, the Delite runtime preallocates all device memory allocations within the kernel before launching it. To achieve this, the CUDA generator collects the memory requirement information and passes it to the runtime through a metadata field in the DEG. In addition, because the GPU resides in a separate address space, input and output data transfer functions are generated so that the Delite runtime can manage data communication. The CUDA generator also produces kernel configuration information (the dimensionality of the thread blocks and the size of each dimension).

Variants. When multiple data-parallel operations are nested, various parallelization strategies exist. In a simple case, a `DeliteOpMap` operation within a `DeliteOpMap` can parallelize the outer loop, the inner loop, or both. Therefore, the Delite compiler framework generates a data-parallel operation in both a sequential version and a parallel version to provide flexible parallelization options when they're nested. This feature is especially useful for the CUDA target generator to improve the coverage of GPU kernels, because parallelizing the outer loop isn't always possible for GPUs due to the kernel's memory-allocation requirements. In those cases, the outer loop is serialized and only the inner loop is parallelized as a GPU kernel.

Target-specific optimizations. Whereas machine-independent optimizations are applied when building the IR, machine-specific optimizations are applied during code generation. For example, the memory-access patterns that allow better bandwidth utilization might not always be the same on the CPU and the GPU. Consider a data-parallel operation on each row of a matrix stored in a row-major format. For the CPU, where each core has a private cache, assigning each row to each core naturally exploits spatial cache locality and prevents false sharing. However, the GPU prefers the opposite access pattern, where each thread accesses each column, because the memory controller can coalesce requests from adjacent threads into a single transfer. Therefore, the CUDA generator emits code that uses a transposed matrix with inverted indices for efficient GPU execution. In addition, to exploit single-instruction, multiple-data (SIMD) units for data-parallel operations on the CPU, we generate source code that the target compiler can vectorize. It would also be straightforward to generate explicit SIMD instructions (such as streaming SIMD extensions and advanced vector extensions).

Executing embedded parallel DSLs

DSLs targeting heterogeneous parallelism require a runtime to manage application execution. This phase of execution includes generating a great deal of "plumbing" code focused on managing parallel execution on a specific parallel architecture. The implementation can be difficult to get right, both in terms of correctness and efficiency, but is common across DSLs. We therefore built a heterogeneous parallel runtime to provide these shared services for all Delite DSLs.

Scheduling the DEG

The Delite runtime combines the machine-agnostic DEG with the current machine's specifications (for example, the number of CPUs or GPUs) to schedule the application across the available hardware resources. It schedules the application before beginning execution using the static knowledge provided in the DEG. Because branch directions are still unknown, the Delite runtime generates a partial schedule for every straight-line path in the application and resolves how to execute those schedules during execution. The scheduling algorithm attempts to minimize communication among kernels by scheduling dependent kernels on the same hardware resource and bases device decisions on kernel and hardware availability. It schedules sequential kernels on a single resource while splitting data-parallel kernels selected for CPU execution to execute on multiple hardware resources simultaneously. Because the best strategy for parallelizing and synchronizing these data-parallel chunks isn't known until after scheduling, the runtime rather than the compiler framework is responsible for generating the decomposition. In the case of a *reduce* kernel, for example, the framework's code generator emits the reduction function and the runtime generates a tree-reduction implementation that's specialized to the number of processors selected to perform the reduction.

Generating execution plans for hardware resources

Dynamically dispatching kernels into a thread pool can have high overhead. However, the knowledge provided by the DEG and the application's static schedule is sufficient to generate an execution plan for each hardware resource and compile them to create executable files. Each executable file launches the kernels and performs the necessary synchronization for its resource according to the partial schedules. The combination of generating custom executable files for the chosen schedule and delaying the injection of synchronization code until after scheduling allows for multiple optimizations in the compiled schedule that minimize runtime overhead. For example, data that doesn't escape a given resource doesn't require any synchronization. In addition, the synchronization implementations are customized to the underlying memory model between the communicating resources. When shared memory is available, the implementation simply passes the necessary pointers, and when the resources reside in separate address spaces, it performs the necessary data transfers. Minimizing runtime overhead by eliminating unnecessary synchronization and removing the central kernel dispatch bottleneck lets applications scale with much less work per kernel.

Managing execution on heterogeneous parallel hardware

Executing on heterogeneous hardware is more challenging than executing on traditional uniprocessor or even multicore systems. The introduction of multiple address spaces requires expensive data transfers that should be minimized. The Delite runtime minimizes data transfers through detailed kernel dependency information provided by the DEG. The graph specifies which inputs a kernel will simply read and which it will mutate. This information combined with the schedule lets the runtime determine at any time during the execution whether the version of an input data structure in a given address space is nonexistent, valid, or old.

Managing memory allocations in each of these address spaces is also critical. The Delite runtime uses the Java Virtual Machine to automatically manage memory for all CPU kernels, but GPUs have no such facilities. In addition, all memory used by a GPU kernel must be allocated before launching the kernel. To deal with these issues, the Delite runtime preallocates all the data structures for a given GPU kernel by using the allocation information supplied by the framework's GPU code generator. The runtime also performs liveness analysis using the schedule to determine the earliest point at which the GPU no longer needs each kernel's inputs and outputs. By default, the GPU host thread attempts to run ahead asynchronously as much as possible, but when this creates memory pressure it uses the liveness information to wait until enough data becomes dead, free it, and continue executing.

Experiments

We evaluated a set of machine-learning applications written in OptiML. We used two quadcore Xeon 2.67-GHz processors with 24 Gbytes of memory and an Nvidia Tesla C2050 GPU for the performance analysis. We didn't include the initialization phase, including input data reading, in the execution time. We report the average of the last five executions.

For the performance comparison with OptiML, we implemented the applications in three other ways: sequential C++ with library, parallel Matlab for multicore CPU, and Matlab for GPU. Matlab is the most widely used programming model in the machine-learning community. Moreover, its performance is often competitive with C++ for machine-learning kernels due to its efficient implementation of linear algebra operations (such as BLAS). We made a reasonable effort in optimizing and parallelizing the code and selecting the most efficient implementation among the libraries. However, this was more challenging than we expected. We couldn't predict which optimization strategy (for example, vectorization, the parallel construct *parfor* in Matlab) would perform best, because many factors, such as the number of cores on the system, affect performance. In addition, the best optimization strategy depends on the particular use case (for example, the matrix's size or the amount of work in the operation). Requiring the application to specify these low-level implementation details often results in multiple versions of the code and makes porting to new devices difficult. OptiML does not suffer from this issue because the operations are implicitly parallel and the optimizations are applied automatically in the Delite compiler framework and runtime rather than manually in the source code.

We evaluated the C++ implementations using the Armadillo linear algebra library[11] to show that the OptiML single-core baseline performs comparably to the C++ library-based implementation.

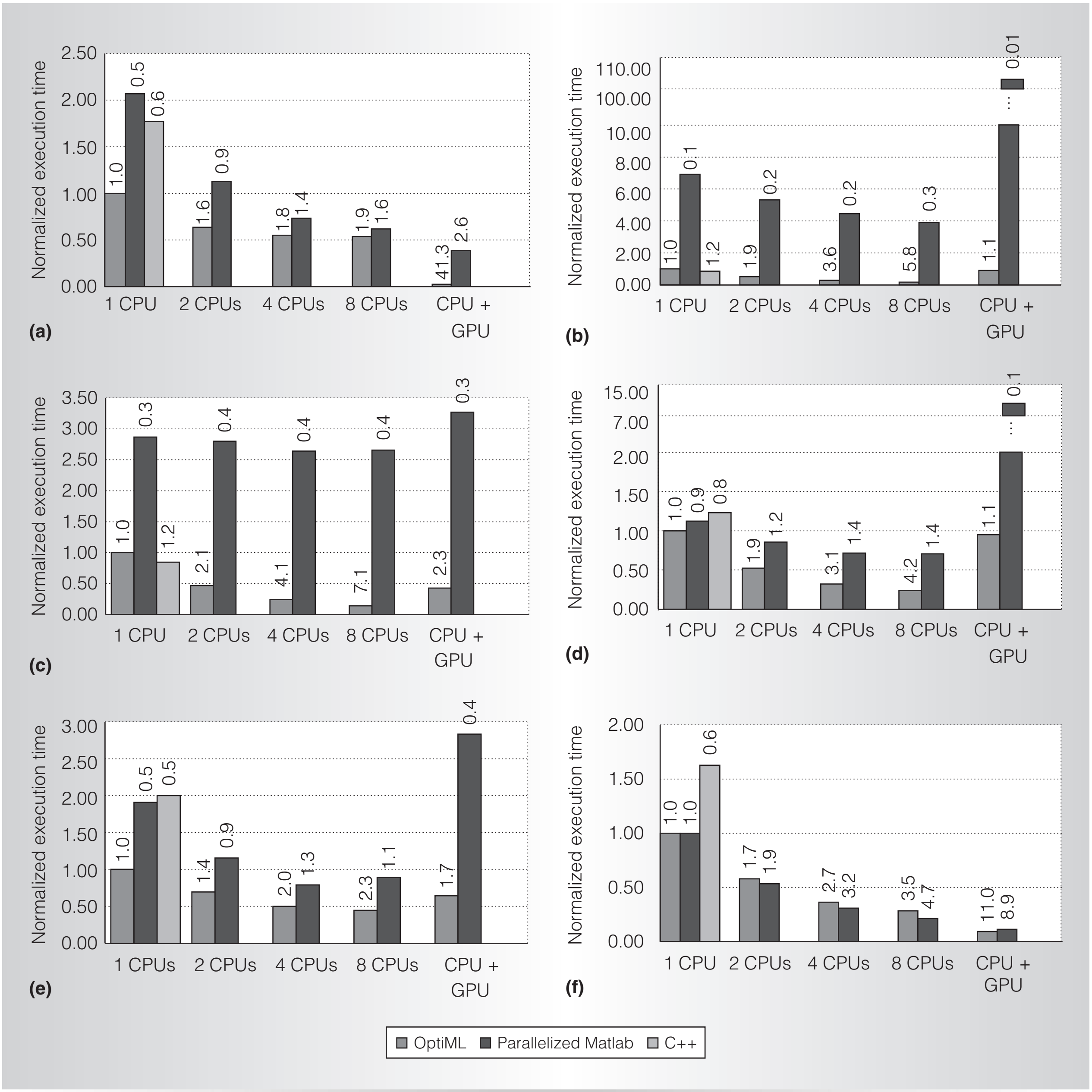


Figure 4. Execution time of the machine-learning applications implemented in OptiML, Matlab CPU and GPU, and C++: Gaussian Discriminant Analysis (GDA) (a), naïve Bayes (b), K-Means (c), support vector machine (d), linear regression (e), and Restricted Boltzmann Machine (RBM) (f). We normalized execution times to the OptiML single-CPU case, and we report speedup numbers at the top of each bar.

Next, we compared the OptiML performance results on multicore CPU and GPU against Matlab implementations. Figure 4 shows that OptiML outperforms Matlab in nearly all applications both in the absolute execution time and in scalability. This is mainly because Delite offers efficient code generation, while Matlab has interpretive overhead. Naïve Bayes, K-Means, and linear regression especially benefit from our ability to generate user-specified functions into specialized GPU kernels, which yields superior performance compared to Matlab's GPU support.

When we compare OptiML performance of multicore CPU and GPU, we see that Gaussian Discriminant Analysis (GDA) and Restricted Boltzmann Machine (RBM) show better performance on the GPU. This is because those applications consist of data-parallel operations with regular memory-access patterns, which is a good fit for GPUs. The other four applications don't perform well on the GPU because the initial memory transfer to the GPU takes too much time (naïve Bayes) or because of frequent communication between the CPU and GPU. From this result, we conclude that neither the CPU nor the GPU is always the optimal solution, and therefore we need a hybrid approach with enough flexibility, as in the Delite runtime. In addition, OptiML applications don't need to be changed at all to run on different targets, whereas other implementations require source code modifications with optimization efforts to run reasonably well on each target.

Figure 5 shows the performance impact of static optimizations on a downsampling application. The C++ implementation is hand-optimized with manual operation fusing. Without the Delite compiler framework's static optimizations, OptiML is 3 times slower than C++. However, with the fusing optimization, which combines multiple iterations into a single pass, and the code motion of hoisting operations out of the loops, OptiML obtains slightly better performance than C++.

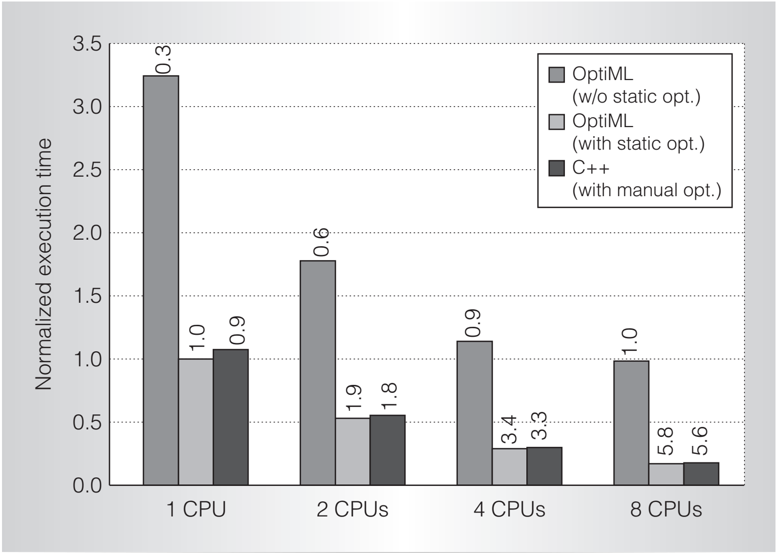


Figure 5. Impact of static optimizations on a downsampling application. Fusing and code motion optimizations of the Delite compiler framework significantly improve OptiML performance, even better than manually optimized C++ code. Speedup numbers are reported on top of each bar.

Using the Delite compiler framework, we're currently implementing DSLs for other domains including graph analysis, database querying, and mesh-based solvers. They all share the framework infrastructure for the IR optimizations and heterogeneous target code generation, demonstrating our system's effectiveness in building implicitly parallel DSLs that target heterogeneous systems. However, the current Delite framework isn't well-suited for expressing all possible DSLs. For example, highly dynamic languages that modify classes or dispatch methods at runtime would be difficult to implement, given that Delite promotes a functional rather than object-oriented design. In addition, Delite doesn't support DSLs whose front ends can't be fully embedded in Scala. This isn't a fundamental limitation, however, and we expect to address it with further development. MICRO

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research was sponsored by US Army contract AHPCRC W911NF-07-2-0027-1; DARPA contract Oracle order US1032821; European Research Council (ERC) under grant 587327 "DOPPLER"; 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

1. J. Held, J. Bautista, and S. Koehl, eds., ''From a Few Cores to Many: A Tera-Scale Computing Research Review,'' white paper, Intel, 2006.
2. ''The Industry-Changing Impact of Accelerated Computing,'' white paper, Advanced Micro Devices, 2008.
3. J. Nickolls and W.J. Dally, ''The GPU Computing Era,'' *IEEE Micro,* vol. 30, no. 2, 2010, pp. 56-69.

4. P. Hudak, ''Building Domain-Specific Embedded Languages,'' *ACM Computing Surveys,* vol. 28, no. 1, 1996, p. 196.
5. A. van Deursen, P. Klint, and J. Visser, ''Domain-Specific Languages: An Annotated Bibliography,'' *ACM SIGPLAN Notices,* vol. 35, no. 6, 2000, pp. 26-36.
6. J. Peterson, P. Hudak, and C. Elliott, ''Lambda in Motion: Controlling Robots with Haskell,'' *Proc. Practical Aspects of Declarative Languages* (PADL 99), Springer-Verlag, 1998, pp. 91-105.
7. D. Bruce, ''What Makes a Good Domain-Specific Language? APOSTLE, and Its Approach to Parallel Discrete Event Simulation,'' *Proc. 1st ACM SIGPLAN Workshop Domain-Specific Languages* (DSL 97), ACM Press, 1997, pp. 17-35.
8. H. Chafi et al., ''Language Virtualization for Heterogeneous Parallel Computing,'' *Proc. ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications,* ACM Press, 2010, pp. 835-847.
9. A.K. Sujeeth et al., ''OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning,'' *Proc. 28th Int'l Conf. Machine Learning* (ICML 11), ACM Press, 2011, pp. 609-616.
10. T. Rompf and M. Odersky, ''Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs,'' *Proc. 9th Int'l Conf. Generative Programming and Component Engineering* (GPCE 10), ACM Press, 2010, pp. 127-136.
11. C. Sanderson, *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments,* tech. report, NICTA, 2006; www.nicta.com.au/pub?id=4314.

HyoukJoong Lee is a PhD candidate in electrical engineering at Stanford University. His research interests include parallel computer architecture and general-purpose GPU computing with their programming models. Lee has an MS in electrical engineering from Stanford University.

Kevin J. Brown is a PhD candidate in electrical engineering at Stanford University. His research interests include parallel computer architecture and heterogeneous runtime systems. Brown has an MS in electrical engineering from Stanford University.

Arvind K. Sujeeth is a PhD candidate in electrical engineering at Stanford University. His research interests include parallel programming models and domain-specific languages for machine learning. Sujeeth has an MS in electrical engineering from Stanford University.

Hassan Chafi is a PhD candidate in electrical engineering at Stanford University. His research interests include parallel programming models, domain-specific languages, and frameworks for simplifying programming heterogeneous systems. Chafi has an MS in electrical engineering from Stanford University.

Kunle Olukotun is a professor of electrical engineering and computer science at Stanford University and director of the Stanford Pervasive Parallelism Lab. His research interests include computer architecture and parallel programming environments. Olukotun has a PhD in computer engineering from the University of Michigan. He's a fellow of IEEE and the ACM.

Tiark Rompf is a PhD candidate at École Polytechnique Fédérale de Lausanne. His research interests include programming languages (in particular Scala), parallelism and compilers, with a focus on lowering the cost of very high level programming abstractions. Rompf has an MS in computer science from the University of Lübeck, Germany.

Martin Odersky is a professor in the school of computer and communication sciences at École Polytechnique Fédérale de Lausanne. His research interests cover fundamental as well as applied aspects of programming languages, including semantics, type systems, programming language design, and compiler construction. Odersky has a PhD in computer science from ETH Zürich. He's a fellow of the ACM.

Direct questions and comments about this article to HyoukJoong Lee, Gates Building, Room 454, Stanford, CA 94305; hyouklee@stanford.edu.