Abstract

The heterogeneous parallel computing era has been accompanied by an ever-increasing number of disparate programming models. As a result, improving performance via heterogeneous computing is currently very challenging for application programmers. Domain-specific languages (DSLs) are a potential solution to this problem, as they can provide productivity, performance, and portability within the confines of a specific domain. However, making the DSL approach useful on a large scale requires lowering the barrier for DSL development. We describe a reusable compiler infrastructure called the Delite Compiler Framework that drastically simplifies the process of building embedded parallel DSLs. DSL developers can easily implement domain-specific operations by extending this framework, which provides static optimizations and code generation for heterogeneous hardware. We also describe the Delite Runtime, which automatically schedules and executes DSL operations on heterogeneous hardware. We demonstrate the potential of the DSL approach by showing the performance of applications written in OptiML, a machine learning DSL developed with the framework, on a system with multi-core CPUs and GPU.

Keywords  Parallel Programming, Domain-Specific Languages, Heterogeneous Computing
productivity, and performance are usually at odds with one another, and must be carefully traded off in successful programming languages. For example, C++ is general and high performance, but usually not considered as productive as higher level languages like Python and Ruby. On the other hand, Python and Ruby cannot compete with C++ in terms of performance. Another approach is to focus on performance and productivity while trading off generality. This can be done by focusing on a particular domain using domain-specific languages (DSLs) [4, 15]. The ability to exploit domain knowledge to pursue high performance and productivity make DSLs an ideal platform for attacking the heterogeneous parallel programming problem.

Domain-specific languages can deliver performance and productivity by providing carefully designed APIs that are easy for developers in the domain to use. DSL code often more closely resembles pseudo-code than C code, deferring most if not all of the implementation details to the language. This deferral of responsibility is extremely important. It allows the DSL developer to use the most efficient parallel implementation and target different devices transparently to the application. This is feasible only because the DSL does not try to do everything; instead, it tries to do a few specific things very well. DSLs provide a structured foundation to identify and exploit parallel execution patterns specific to particular domains. They are not a silver bullet, however. They cannot parallelize existing sequential code, and they do not on their own eliminate the parallel programming burden for the DSL developer.

For parallel DSLs to be a tractable approach, they must be easy enough to create for many domains. Traditionally, there are two types of DSLs: internal and external. Internal DSLs are embedded in a host language, and are sometimes called the "just-a-library" approach [10]. These DSLs typically use a flexible host language to provide nice syntactic sugar over library calls. While this is the easiest possible approach (no compiler necessary), it fundamentally constrains the capabilities of the DSL. A purely embedded, or library-based, DSL cannot build or analyze an intermediate representation (IR) of user programs. This means that they can only perform dynamic analyses and optimizations and these handicaps can severely impact achievable parallel performance. More importantly, without an IR, DSLs cannot do their own code generation, which prevents retargeting DSL code to heterogeneous devices.

The other class of DSLs is external. These are the DSLs that are implemented as a standalone language [2]. While these DSLs obviously do not have the limitations of internal DSLs, they are 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 (e.g., IDE support). This is clearly not a scalable approach. To solve this dilemma, we propose a hybrid approach. We use the concept of language virtualization [3] to characterize a host language that allows implementing embedded DSLs that are virtually indistinguishable from standalone DSLs. A virtualizable host language provides an expressive, flexible front-end that the DSL can borrow, while allowing the DSL to 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 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 the 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, only the function behavior needs to be specified by the DSL developer. The boilerplate code (e.g., iterating over the loop, updates, parallelization) is managed by the framework and the runtime. The framework also provides code generators for those patterns, and therefore the DSL can target heterogeneous hardware (multi-core CPU, GPU, etc) without developing code generators for each target. In addition, efficient scheduling and exploiting the task & data parallelism with proper synchronization are managed by the runtime, which used to be another burden on the DSL developer. Therefore, our system allows DSL developers to focus on the language design rather than on the implementation details and enables exploiting heterogeneous parallel performance without writing any low-level parallel code.

3. OptiML and Delite

The overall operation of the Delite Compiler Framework and Runtime is shown in Figure 2. The figure shows an example of OptiML [13], a machine learning DSL developed with our framework. OptiML provides Matrix / Vector / 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 DeliteOpMapReduce parallel pattern of the framework.

To generate optimized executables from the high level representation of the DSL operations, the Delite compiler builds an intermediate representation (IR) of the application and applies various optimizations on the IR nodes. For example, since the two vector minus operations \((x(i) - \mu_0)\) and \((x(i) - \mu_1)\) within the sum are redundant, common subexpression elimination removes the latter operation by reusing the former result. After building the IR, the code generators in the framework automatically emit computation kernels for both the CPU and the GPU.
When all of the kernels of the application 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 make scheduling decisions to generate execution plans. Necessary memory transfers and synchronization are added as the execution plan for each target is generated. Finally, the kernels from the compiler and the execution plan from the runtime are compiled and linked together by target language compilers (e.g., Scala, Cuda) to generate an executable that runs on the system.

### 4. Building Embedded Parallel DSLs

For a high performance DSL to target heterogeneous parallel systems, its IR should have at least the following 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

#### 4.1 Building an Intermediate Representation (IR)

To incorporate all the aforementioned requirements, we propose a multi-view representation of the IR as depicted in Figure 3. A single IR node can be viewed from three different perspectives which provide different optimizations and code generation strategies. We built the Delite Compiler Framework, which is a reusable compiler infrastructure for developing DSLs, using the concept of a multi-view IR.

**Generic IR**: The most basic view of an IR node is a symbol and its definition, which is similar to a node in the flow graph of a traditional compiler framework. Therefore, we can 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, the common subexpression elimination (CSE) can be applied to the vector operations \((x(i) - \mu_0, x(i) - \mu_1)\) as shown in Figure 2 instead of just to scalar operations. Currently applied optimizations include CSE, constant propagation, dead code elimination, and code motion.

**Parallel IR:** A generic IR node can be characterized by its parallel execution pattern. At this level of view, the Delite Compiler Framework 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 the pattern that is not parallelizable. Since certain parallel patterns share a common notion of loops, multiple loop patterns can be fused into a single loop. The parallel IR optimizer iterates over all of the IR nodes of loop types (e.g., DeliteOpMap, DeliteOpZipwith, etc.), and fuses those with the same number of iterations into a single loop. This optimization removes unnecessary memory allocations and also improves cache behavior by eliminating multiple passes over data, which is especially useful for memory-bound applications.

**Domain-specific (DS) IR:** Since the parallel IR does not encode domain-specific information, there is another viewpoint for semantic information of the operation. This enables domain-specific optimizations such as linear algebra simplification. The transformation rules are simply described by pattern matching on DS IR nodes, and the optimizer replaces the matched nodes with a simpler set of nodes. Examples on matrix operations are \((A^T)^t = A\), and \(A \times B + A \times C = A \times (B + C)\). Since the DSL developer has expertise in the execution patterns of each DS IR node, the DSL developer extends the appropriate Delite parallel IR node. In this way parallel execution patterns are abstracted away from DSL users.

This multi-view IR greatly simplifies the process of developing a new DSL since the generic IR and the parallel IR can be re-used by all DSLs, and therefore DSL developers only need to design a DS IR for each operation as an extension. In other words, DSL developers are only exposed to a high-level parallel instruction set (the parallel IR nodes) and the implementation details of each pattern on multiple targets are automatically managed by the Delite Compiler Framework.

To build the IR from a DSL application, the Delite Compiler Framework uses a technique called Lightweight Modular Staging (LMS) [11]. As the application starts executing within the framework, each operation is lifted to a symbolic representation to form an IR node rather than actually being executed. The IR nodes track all dependencies among one another and the various optimizations mentioned above are applied. After building the machine-independent IR, the Delite Compiler Framework starts the code generation phase to target heterogeneous parallel hardware.

### 4.2 Heterogeneous Target Code Generation

Generating a single binary executable for the application at compile time limits the portability of the application and requires runtime and hardware systems to rediscover dependency information in order 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 Delite Execution Graph describing the dependencies among kernels. Currently supported targets are Scala [9], C++, and CUDA.

#### 4.2.1 Delite Execution Graph (DEG)

The Delite generator is the main code generator that controls multiple target generators. It first schedules IR nodes to form kernels in the execution graph, and iterates over the list of available target generators to generate corresponding target code for the kernel. It may not be possible to generate the kernel for all targets, but the kernel generation will succeed as long as at least one target succeeds. The fact that each IR node has multiple viewpoints means that they can also be generated in different ways for each view. For example, a matrix addition kernel could be generated in the domain-specific view written by the DSL developer, but it also can be generated in the parallel view since the operation is of type DeliteOpZipWith. Since the Delite Compiler Framework provides parallel implementations for DeliteOps, DSL developers do not have to provide code generators when they extend one of the parallel IR nodes. When DSL developers already have an efficient implementation of
the kernel (e.g., BLAS libraries for matrix multiplication), they can generate calls to the external library using `DeliteOpExternal`.

### 4.2.2 GPU code generation

GPU code generation requires additional work since the programming model has more constraints compared to the Scala and C++ targets. One major issue is memory allocation. Since dynamic memory allocation within the kernel is either not possible or not practical for performance in GPU programming models, all device memory allocations within the kernel are pre-allocated by the Delite Runtime before launching the kernel. This is enabled by the CUDA generator collecting the memory requirement information and passing it to the runtime through a metadata field in the DEG. In addition, since the GPU resides in a separate address space, input/output transfer functions are also generated so that the Delite Runtime can manage data communication. Kernel configuration information (the dimensionality and the size of each dimension) also needs to be generated by the CUDA generator.

### 4.2.3 Variants

When multiple data parallel operations are nested, various parallelization strategies exist. In a simple case, a `DeliteOpMap` op within a `DeliteOpMap` can parallelize the outer loop or 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 are nested. This feature is especially useful for the CUDA target generator to improve the coverage of GPU kernels, since parallelizing the outer loop is not always possible for GPU due to the memory allocation requirements of the kernel. In those cases, the outer loop is serialized and only the inner loop is parallelized as a GPU kernel.

### 4.2.4 Target-specific Optimizations

While machine-independent optimizations are applied when building the IR, machine-specific optimizations are applied during the code generation phase. For example, the memory access patterns that enable better bandwidth utilization may 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. In the case of the CPU where each core has its own 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 SIMD units for data-parallel operations on the CPU, we currently generate source code that can be vectorized by the target compiler. It would also be straightforward to generate explicit SIMD instructions (e.g., SSE, AVX).

### 5. Executing Embedded Parallel DSLs

DSLs targeting heterogeneous parallelism require a runtime to manage application execution. The work done at 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.

#### 5.1 Scheduling the Delite Execution Graph (DEG)

The Delite Runtime combines the machine-agnostic DEG generated by the framework with the specification of the current machine (e.g., number of CPUs, number of GPUs, etc.) to schedule the application across the available hardware resources. The Delite Runtime schedules the application before beginning execution using the static knowledge provided in the DEG. Since 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 makes device decisions based on kernel and hardware availability. Sequential kernels are scheduled on a single resource while data-parallel kernels selected for CPU execution are split by the scheduler to execute on multiple hardware resources simultaneously. Since the best strategy for parallelizing and synchronizing these data-parallel chunks is not known until after scheduling, the runtime 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 is specialized to the number of processors selected to perform the reduction.

5.2 Generating Execution Plans for Each Hardware Resource
Dynamically dispatching kernels into a thread pool can have very high overheads. However, the knowledge provided by the DEG and static schedule of the application is sufficient to generate and compile an executable (execution plan) for each hardware resource. Each executable launches the kernels and performs the necessary synchronization for its resource according to the partial schedules. The combination of generating custom executables 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 does not escape a given resource does not require any synchronization. This synchronization code is 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 enables applications to scale with much less work per kernel.

5.3 Managing Execution on Heterogeneous Parallel Hardware
Executing on heterogeneous hardware introduces new and difficult challenges compared to traditional uniprocessor or even multi-core systems. The introduction of multiple address spaces requires expensive data transfers that should be minimized. The Delite Runtime achieves this 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 allows the runtime to determine at any given time during the execution if the version of an input data structure in a given address space is currently nonexistent, valid, or old.

Managing the memory in each of these address spaces is also critical. The Delite Runtime currently utilizes the JVM to perform memory management for all CPU kernels, but GPUs have no such facilities. In addition, all memory used by a GPU kernel must be allocated prior to launching the kernel. In order to deal with these issues, the Delite Runtime pre-allocates 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 each kernel’s inputs and outputs are no longer needed by the GPU. By default the GPU host thread attempts to run ahead 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.

6. Experiments
We evaluated a set of machine learning (ML) applications written in OptiML. The system for the performance analysis consists of two quad-core Xeon 2.67 GHz processors with 24 GB of memory and an nVIDIA Tesla C2050 GPU. The initialization phase including input data reading is not taken into account for the execution time, and the average of the last 5 executions are reported.

For the performance comparison with OptiML, we implemented the applications in three other ways: sequential C++ with library, parallel MATLAB [6] for multi-core CPU, and MATLAB for GPU. MATLAB is the most widely used programming model in the ML community and the performance is often competitive with C++ for ML kernels due to the efficient implementation of linear algebra operations (e.g., BLAS) in
MATLAB. We made a reasonable effort in optimizing and parallelizing the code and selecting the most efficient implementation among the libraries. However, this turned out to be more challenging than we expected. First of all, it is not obvious to predict which of the optimization strategies (e.g., vectorization, the parallel construct ‘parfor’ in MATLAB) would perform the best, since it depends on many factors such as the number of cores. In addition, the best optimization strategy depends on the particular use-case (e.g. the size of the matrix, the amount of work in the operation). Requiring the application to specify these low-level implementation details results in multiple versions of the code and makes porting code to new devices difficult.

The C++ implementations with Armadillo linear algebra library [12] are evaluated to show that OptiML single-core baseline performs comparably to the library-based implementation. We next compare the OptiML performance results on multi-core CPU and GPU against MATLAB implementations. Figure 4 shows that OptiML outperforms MATLAB in nearly all applications not only in the absolute execution time but also in terms of scalability. This is mainly due to efficient code generation of Delite, in contrast to MATLAB that has interpretive overhead. Naive Bayes, Kmeans, and Linear Regression are the applications that especially benefit from our ability to generate user-specified functions into specialized GPU kernels, which yields superior performance compared to MATLAB’s GPU support.

Comparing OptiML implementations of multi-core CPU and GPU, GDA and 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 do not perform well on the GPU either because the initial memory transfer to the GPU takes too much time (Naive Bayes) or because of frequent communication between the CPU and GPU. From this result, we can conclude that neither the CPU nor the GPU is always the optimal solution and therefore a hybrid approach with enough flexibility as in the Delite Runtime is necessary. In addition, OptiML applications do not need to be changed at all to run on different targets, whereas other implementations require source code modifications with optimizations to run reasonably well on each target.
Figure 5: Impact of static optimizations on a downsampling application. Speedup numbers are reported on top of each bar.

Figure 5 shows the performance impact of static optimizations on a downsampling application. The C++ implementation is hand-optimized with manual op-fusing. Without the static optimizations of the Delite Compiler Framework, OptiML is 3x 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++.

7. Future Work

With the Delite Compiler Framework, we are 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 the effectiveness of using our system to easily build implicitly parallel DSLs that target heterogeneous systems. However, the current Delite framework is not 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, as Delite promotes a functional rather than object-oriented design. Delite also does not support DSLs whose front-end cannot be fully embedded in Scala. This is however not a fundamental limitation and can be addressed with further development.

8. Conclusion

In the era of heterogeneous computing, most programmers have difficulty taking advantage of the full capabilities of the system due to too many disparate parallel programming models. To address this important problem, higher-level parallel programming models are necessary, and we propose domain-specific languages (DSLs) as a potential solution. Since building a new DSL must be easy for the DSL approach to be successful, we designed the Delite Framework that allows DSL developers to easily develop their own DSLs that can target heterogeneous parallel systems. We demonstrated the potential of the DSL approach with OptiML, a machine learning DSL, by showing the performance benefits of OptiML applications on a system with multi-core CPUs and GPU. Future work will focus on supporting other DSLs with the Delite Framework.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. 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


