Nessie: A NESL to CUDA Compiler

John Reppy and Nora Sandler
University of Chicago

Abstract. Modern GPUs provide supercomputer-level performance at commodity prices, but they are notoriously hard to program. To address this problem, we have been exploring the use of Nested Data Parallelism (NDP), and specifically the first-order functional language NESL, as a way to raise the level of abstraction for programming GPUs. This paper describes a new compiler for NESL language that generated CUDA code. Specifically we describe three aspects of the compiler that address some of the challenges of generating efficient NDP code for GPUs.

1 Introduction

Modern GPUs provide supercomputer-level performance at commodity prices, but they are notoriously hard to program. To address this problem, we have been exploring the use of Nested Data Parallelism (NDP), and specifically the first-order functional language NESL [Ble96], as a way to raise the level of abstraction for programming GPUs. NESL was originally designed by Guy Blelloch as a way to program irregular parallel algorithms on the wide-vector and SIMD architectures of the early to mid 1990s. Blelloch and others developed a global flattening transformation that maps irregular NDP code into regular flat data parallel (FDP) code suitable for SIMD execution. The flat representation uses segment descriptors to represent the original nesting structure and relies on segmented-data-parallel operations (e.g., segmented scans and reductions) in its implementation. Previously, we ported Blelloch’s compiler to generate CUDA, which demonstrated that NESL has the potential to be an effective language for programming GPUs [BR12]. This work also showed, however, that there is a need for better compilation techniques to make NDP competitive with hand-written CUDA code. To address this need, we have been implementing a new NESL compiler that provides a platform for exploring optimization techniques for NDP on GPUs.

The front half of the compiler does the usual parsing and type checking, followed by monomorphization (NESL has parametric polymorphism). We then apply Keller’s flattening transformation [Kel99] to produce a FDP representation of the program. This paper focuses on three techniques that we are using to compile the FDP representation to CUDA.

Shape analysis. We use a constraint-based shape analysis to determine the size and structure of arrays, which is key to enabling optimizations. The analysis identifies arrays of the same size, which enables the compiler to find more opportunities for fusion and memory reuse. We also infer concrete shape information whenever possible, which leads to more efficient memory allocation.
Aggressive fusion. Fusion is the key to improved GPU performance, since it allows us to reduce the number of kernel invocations and reduce global memory traffic. Our compiler supports a broader range of producer-consumer fusion than our previous implementation; for example, it fuses maps and reductions into the same kernel. It also does horizontal fusion, where it fuses independent maps and reductions that have the same shape when possible.

Compile-time memory management. We use Hudak’s compile-time reference counting [Hud86] to statically manage the deallocation of global GPU memory. Coupled with the information from shape analysis, we can detect when a global memory object can be reused, which allows us to reduce memory allocation/deallocation churn.

Our implementation is work in progress, but preliminary results show that Nessie generates significantly better GPU code than our previous effort.

The remainder of the paper is organized as follows. In the next section, we give a quick overview of the NESL language and an overview of our implementation. In the following three sections, we describe how we use shape analysis, fusion, and compile-time memory management in our implementation. We discuss related work in Section 6 and then conclude.

2 Preliminaries

In this section, we give a quick introduction to the NESL language and an overview of the Nessie compiler’s architecture.

2.1 NESL

NESL is a first-order data-parallel functional language designed by Guy Blelloch [Ble96]. NESL supports data parallelism in two ways: through an apply-to-each construct, which is a parallel array comprehension, and a set of parallel primitive operators. The apply-to-each construct allows the programmer to map an arbitrary expression over each element of a sequence. For example, the following function squares each element of a sequence:

\[
\text{function} \quad \text{sqr} \ (xs) = \{x \times x : x \in xs\};
\]

It is also possible to map an expression containing multiple variables over multiple sequences of the same length:

\[
\text{function} \quad \text{prod} \ (xs, ys) = \{x \times y : x \in xs; y \in ys\};
\]

An apply-to-each may also include an optional filter specifying which elements to include in the result. We could modify the above function to only multiply two elements when they are both positive:

\[
\text{function} \quad \text{prod_if_pos} \ (xs, ys) = \{x \times y : x \in xs; y \in ys \mid x \geq 0 \text{ and } y \geq 0\};
\]
More interestingly, NESL supports nested-data parallelism: apply-to-each expressions may themselves contain parallel expressions. NESL thus supports parallel operations on nested sequences of arbitrary depth. This model is well-suited to matrix operations, as well as irregular parallel problems, such as divide-and-conquer algorithms.

For example, we can use nested parallelism to multiply a matrix by a vector:

```plaintext
function dotp (xs, ys) =
    sum({x * y : x in xs; y in ys});

function mxv (m, v) =
    {dotp(row, v) : row in m};
```

The `mxv` function exhibits nested parallelism. The outer apply-to-each applies `dotp` in parallel to each row of the matrix; this is the outer layer of parallelism. Within each call to `dotp`, the elements of the row are all multiplied in parallel, then added up with the parallel `sum` operation, which is the inner layer of parallelism.

Nested parallelism does not have to be regular in NESL. For example, the computation of a sparse matrix (represented by rows of index-element pairs) times a dense vector can be coded as

```plaintext
function sparse_mxv (sm, v) =
    {sum({x * v[i] : (i, x) in sv} : sv in sm)
```

which has the same nested parallel structure as before, but with the difference that the amount of work per row varies.

### 2.2 The NESL Basis Library

In addition to the parallel apply-to-each construct, NESL has a library of parallel vector operations. These include reductions, such as the `sum` function above, prefix-scans, permutations, and various other operations on sequences [Ble95]. In Blelloch’s implementation these operations are implemented using a NESL-like term language that is embedded in Common Lisp (Blelloch’s compiler is written in Common Lisp). Our implementation is based on a mostly mechanical translation of these operations into an extension of NESL that supports typecase and a set of primitive operations based on the VCODE operators [BC90]. The typecases are used to specialize operations to various primitive types (e.g., `sum` works on both integers and floats), as well as to handle the translation of operations of nested arrays to their flat segmented implementation.

### 2.3 The Nessie compiler

The Nessie compiler is organized as a serious of transformations between intermediate languages. We summarize these transformations and representations here, for more details please see the second author’s Honors Thesis [San14].

- Parsing and type checking produces a typed abstract syntax tree (AST) representation. This representation makes instantiation of polymorphic functions explicit.
– Monomorphization instantiates all polymorphic functions to monomorphic types, producing a monomorphically-typed AST. At this stage we also specialize the type-case constructs that are used in the NESL Basis Library to their actual type arguments.
– The normalization phase simplifies the program into a direct-style representation.¹
– Flattening is a global program transformation that converts nested data parallelism into flat data parallelism [BS90, BCH⁺ 94, Kel99]. A key aspect of this representation is that nested arrays are transformed into segmented arrays, which are flat arrays paired with segment descriptors that define the original array’s nesting structure. In addition to affecting the data representation, this transformation also affects the program structure, replacing conditionals inside parallel code with predicated operations. We call the resulting intermediate representation Flan.
– The Nessie compiler performs a number of optimization passes over the Flan representation. These include inlining, arity raising, and multiple contraction passes.
– Flattening is a global program transformation that converts nested data parallelism into flat data parallelism [BS90, BCH⁺ 94, Kel99]. A key aspect of this representation is that nested arrays are transformed into segmented arrays, which are flat arrays paired with segment descriptors that define the original array’s nesting structure. In addition to affecting the data representation, this transformation also affects the program structure, replacing conditionals inside parallel code with predicated operations. We call the resulting intermediate representation Flan.
– The program is then converted into a representation called FuseAST, in which the distinction between host and device computations is made explicit. To this representation we apply a simple optimization of producer-consumer fusion to element-wise operations; i.e., we fuse parallel maps where the output of one map is consumed by another map. This optimization is similar to our previous VCODE optimizer [BR12], except that shape analysis allows us to handle some additional cases.
– We construct a block-structured program-dependence graph [FOW87] from the FuseAST representation. We use the control regions of this graph, combined with shape information, to map the FuseAST representation to $\lambda_{cu}$, which is the Nessie compiler’s final IR before code generation. In $\lambda_{cu}$, we express GPU operations using task graphs, which allows us to represent a richer set of fused kernels. We discuss this representation and our fusion techniques in Section 4.
– We analyze the $\lambda_{cu}$ representation using Hudak’s compile-time reference counting [Hud86] analysis. This analysis allows us to accurately determine the lifetime of GPU memory objects, which we make explicit by adding memory management operations to the $\lambda_{cu}$ representation. We discuss this process in more detail in Section 5.
– Finally, we generate CUDA kernels and C++ host functions to call them.

3 Shape analysis

Shape analysis is the process of statically identifying the nesting structure and sizes of arrays. Shape analysis plays a key role in the compiler as it is the enabler for our fusion optimizations and for smart kernel scheduling.

We use a constraint-based analysis to determine the shapes of sequences and segment descriptors. This analysis enables several different optimizations:

¹ A direct-style representation is similar in spirit to SSA; variables are assigned to once and operators are only applied to variables. The biggest difference is that control flow is represented as function application.
If we know that two separate elementwise primitives operate on sequences of the same length, we can fuse them into a single kernel (see Section 4).

- If we know that two segment descriptors are identical, we can eliminate one of them.
- If we know that a segment descriptor is uniform --- i.e., it describes a sequence whose segments are all the same length --- we can represent it more compactly.

In order to track all of this information, it is not just sufficient to associate each sequence or segment descriptor with a symbolic length. To determine whether two segment descriptors are identical, we must also know something about their contents. To further complicate matters, the value of one variable may dictate the length of another; for instance, the DIST operation takes a scalar argument that specifies the length of its result.\(^2\) We therefore need a more expressive way to encode shape information. Our solution resembles a dependent type system, where each shape depends on one or more shape parameters. A shape parameter \(p\) is essentially a unique variable that denotes an abstract value in the analysis (in some cases it may have a known concrete value). Our full system for encoding shape information is given in Figure 1.

Roughly speaking, a shape parameter may determine a sequence's length or a scalar's value. We only attempt to determine the concrete values of a sequence or segment descriptor in two cases: when we know it contains only one element, or when we know it is uniform. Otherwise, the second parameter in a VEC shape can uniquely identify that shape (i.e., we can test if two shapes are the same), but provides no further information about it. In our analysis, we treat a segment descriptor like a sequence of integers, allowing us to handle segment descriptors and sequences uniformly. For example, consider the following nested sequence:

\[
[[1, 2, 3], [4, 5, 6]]
\]

This sequence is represented in Flan by a flat sequence of the data, an inner segment descriptor, and an outer segment descriptor:

\[
(<2>, (<3, 3>, [ 1, 2, 3, 4, 5, 6 ]))
\]

If we ignore the labels on our shape parameters and just consider their concrete values, the outer segment descriptor would have shape \(\text{VEC}(1, 2)\), because it has one element, and that element's value is 2; the inner segment descriptor would have shape \(\text{UNIFORM}_\text{VEC}(2, 3)\), because it has two elements and both of them have value 3;

\(^2\) The DIST operation creates a vector where every element is the same. It is introduced when flattening for-each expressions such as \{ \text{x + 17 : x in xs} \}. 

---

\( s ::= \text{SCALAR}(p) \quad (\text{parameter is value of scalar}) \)

\( | \text{VEC}(p_1, p_2) \quad (\text{first parameter is length, second is value}) \)

\( | \text{UNIFORM}_\text{VEC}(p_1, p_2) \quad (\text{first parameter is length, second is value of each element}) \)

\( | \text{PAIR}(s_1, s_2) \quad (\text{recursively define another shape for each pair element.}) \)

Fig. 1. Encoding of Flan shapes
and the base sequence would have shape $\text{VEC}(6, \text{UNKNOWN})$, since it has six elements and it is not uniform.

Shape analysis consists of three steps. In the first, we annotate variables with their shapes based on the characteristics of our primitive operations and set constraints between shape parameters. In the second, we solve constraints, partitioning shape parameters into equivalence classes and selecting a representative member for each class. Finally, we normalize our shape annotations by replacing each shape parameter in the program with the representative member of its equivalence class, ensuring that identical shapes are always represented by the same parameter.

4 Fusion

After shape analysis, the program is transformed into the FuseAST representation, which replaces lifted operators with parallel maps, and which makes explicit the distinction between CPU-side and GPU-side operations. The main purpose of this representation is to provide a vehicle for simple map-map fusion and a representation for assigning computation to tasks.

4.1 Simple fusion

The flattening transformation provides a solution to the problem of how to execute irregular NDP code in a regular way, but it breaks the computation down into many individual vector operations that, while they often work on large amounts of data (because of the large width of their inputs), have very small computational kernels. For example, the for-each expression

\[
\{ \sqrt{x^2 + y^2} \; : \; x \; \text{in} \; xs; \; y \; \text{in} \; ys \}
\]

will result in four individual vector operations:

\[
\begin{align*}
\text{let } & \quad t1 = \text{map } (\lambda x \rightarrow x^2) \; xs \\
\text{let } & \quad t2 = \text{map } (\lambda y \rightarrow y^2) \; ys \\
\text{let } & \quad t3 = \text{map } (\lambda (x2, y2) \rightarrow x2 + y2) \\
\text{in } & \quad \text{map } (\lambda t \rightarrow \sqrt{t})
\end{align*}
\]

Fusion lets us merge these operations into a single kernel

\[
\text{map } (\lambda (x, y) \rightarrow \sqrt{x^2 + y^2}) \; (xs, ys)
\]

Our original NESL/GPU implementation [BR12] performed this kind of simple map-map fusion, but, because it did not have accurate shape information, it was unable to perform more aggressive forms of fusion, which resulted in serious performance issues.

4.2 The PDG and aggressive fusion

After simple fusion, the next step is to schedule computations as tasks, where a task will be implemented as a single CUDA kernel function. To do this we first construct a block-structured variant of the classic program-dependence graph (PDG) [FOW87]
representation from the FusedAST representation. For a given control region (e.g., one branch of a conditional), we have a data flow graph where the nodes correspond to value bindings in the FuseAST representation and the graph edges represent data dependencies. We mark some edges as requiring a task-level synchronization (i.e., these edges represent a data dependency between two CUDA kernels), whereas other edges do not. We then partition the graph nodes into stages based on the number of maximum number of synchronization edges between the roots and the node. Nodes that are in the same stage are candidates for fusing into the same task. We fuse such operations if they fit into one of the following cases:

- Array generators are fused with their consumers.
- Map operations that have the same shape can be fused into the same task (even if they are independent).
- A map whose output is consumed by a parallel reduction can be fused into the same task (assuming they are in the same stage).
- Reductions that have the same input shape can be fused.
- Certain other operations, such as prefix scans, that preserve the shape of their inputs can be fused with the producers of their inputs (or consumers of their outputs) that are in the same stage.

Our current implementation does not apply these rules to segmented scans or reductions, but that is an enhancement that we plan to explore.

This approach to fusion produces much bigger kernels than our previous approach. Once we have computed the task schedule, we convert the FuseAST to the $\lambda_{cu}$ representation. This representation separates the GPU-side tasks into their own sub-language and makes explicit the distinction between CPU-side and GPU-side data. It also has explicit representation of the allocation and deallocation of GPU-side data objects.

5 Compile-time memory management

The NESL language does not have explicit memory management, meaning that the implementation must provide some form of automatic garbage collection. Because it also does not have cyclic data structures, reference counting can be an effective technique. Reference counting does not directly support, however, the immediate reuse of argument arrays to hold results. reuse “dead” arrays for results In the example from above

$$\text{map} \ (\lambda \ (x, y) \Rightarrow \sqrt{x*x + y*y}) \ (xs, ys)$$

it might be possible to reuse the storage allocated for $xs$ (or $xy$) to hold the result of the parallel map, but reference counting only determines that fact at runtime. Blelloch’s implementation addresses this problem by providing multiple implementations of the VCODE operators that are customized to different storage scenarios and then dispatching to the appropriate operator based on runtime analysis.

We have decided to take a static analysis approach to solve this problem. Fortunately, it turns out that our $\lambda_{cu}$ representation corresponds very closely to the framework of Hudak’s compile-time reference counting [Hud86] analysis. The results of the analysis are used to both reuse input memory for outputs (when it is semantically safe to
do so for the given task) and to explicitly free GPU memory objects when they become garbage. The static analysis is backed up by a runtime reference-counting scheme, but so far that has not been needed.

6 Related work

Several compilers exist to support nested-data parallel languages on the GPU. CuNESL is a recent compiler framework to translate NESL to CUDA C++ [ZM12]. Instead of fully flattening the program, CuNESL chooses among different levels in the GPU’s hierarchy to operate at different layers of parallelism, depending on the problem size. CuNESL is focused on efficiently implementing recursive calls, so it does not address optimizations, such as kernel fusion, that benefit both recursive and non-recursive functions. Copperhead, a GPU programming language embedded in Python, also takes a hierarchical approach to flattening nested data parallelism [CGK11].

An earlier port of NESL to GPU, described in [BR12], employs Blelloch’s original flattening transformation, which flattens NESL code into a stack-based intermediate language called VCODE. VCODE has proven difficult to optimize, both because it contains far less program information than NESL or Flan, and because the sorts of optimizations described in this paper, which can easily be implemented by traversing a program’s AST, are much more difficult to apply to stack-machine code.

Data Parallel Haskell (DPH) takes a similar approach to ours, relying on the flattening transformation to fully support nested-data parallelism [CLPK08]. DPH employs an extended flattening transformation that supports higher-order functions and recursive datatypes [LCK06]. DPH targets multicore CPUs, however, rather than GPUs.

NOV A is another high-level, functional language for GPU programming [CGG +14]. Like Nessie, the NOV A compiler takes advantage of a simple, functional intermediate language (although in this case the intermediate language is NOV A itself) in order to implement a range of optimization passes. NOV A supports higher-order functions and recursive datatypes, which NESL does not; however, it only permits a more restricted form of nested parallelism, because recursive calls are not allowed inside of map expressions. NOV A targets sequential, multi-threaded, and GPU architectures.

7 Conclusion

We have described the Nessie compiler, which compiles the NESL language to CUDA code. Our compiler relies on shape analysis to enable aggressive fusion and other operations. We also use static analysis to enable static memory management.

The Nessie compiler is a work in progress. Our current focus is completing the implementation described in this paper (for example, we have yet to write CUDA kernels for many of the more complex vector operations used by NESL, such as permutations), but early experiments suggest that our new compiler produces code that is significantly better than our previous effort.
7.1 Future work

We have believe that there are a number of ways to further improve performance and we have designed the structure of our compiler in anticipation of supporting these.

One idea that we plan to explore is Keller et al.’s vectorization avoidance technique [KCL+12]. In this technique, we identify larger scalar expressions that can be abstracted into an opaque operation before flattening. This technique reduces the need for fusion and can prevent the encoding of control flow as data, which could be beneficial in some circumstances.\(^3\)

Our current algorithm for scheduling computations into tasks uses a simple greedy approach that can produce suboptimal results. Robinson et al. have been exploring the use of integer linear programming techniques for making fusion decisions in a similar framework [RLK14] and we believe that we can directly apply their technique for encoding the fusion problem as an ILP system to our PDG representation.

One weakness of the flattening techniques used to implement NDP is that they can greatly increase the amount of intermediate storage required [PPW95,LCK+12]. An approach to reducing the storage requirements is piecewise implementation of data-parallel computations [PPCF95,Pfa00], where the computation is broken into slices. Recent experiments with hand-compiled NDP code have demonstrated that piecewise execution can have significant performance advantages [MF13] and we plan to integrate this approach in our code generator.

Acknowledgments

The original NESL/GPU implementation was primarily the work of Lars Bergstrom, who has also provided helpful advice about CUDA hacking. We also thank the NVIDIA Corporation for their generous donation of both hardware and financial support. Portions of this research were supported by the National Science Foundation under awards CCF-1010568 and CCF-1446412. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References


\(^3\) Doing so breaks the pure SIMD model that is used to execute flattened programs, but it should reduce overhead in many situations at the cost of some divergence.


