Nessie: A New NESL Compiler

Nora Sandler
University of Chicago
nlsandler@cs.uchicago.edu

Spring 2014

Abstract

Graphics Processing Units (GPUs) have hundreds to thousands of cores, promising extremely high performance for data-parallel applications. However, most general-purpose GPU programming today is done in low-level languages like CUDA and OpenCL, which require programmers to painstakingly hand-tune their code in order to achieve good performance.

NESL, a functional data-parallel programming language, offers several advantages over these low-level languages. It is more concise, and it supports nested-data parallelism, in which parallel primitives may themselves be called in parallel. This allows NESL to cleanly express irregular parallel computations. A program transformation called flattening converts nested parallel programs into single-instruction multiple-data (SIMD) programs suitable for execution on GPU.

However, flattening alone does not ensure good performance; the flattened code must be further optimized to handle the specific challenges of GPU architecture. We are currently developing Nessie, a new compiler that translates NESL into efficient CUDA C. The Nessie compiler is structured as a series of whole-program transformations; by relying on different intermediate representations to reflect different aspects of the GPU programming model, we can implement a wide range of optimizations that must be hand-coded in GPU programs today. This paper presents the design and current state of the Nessie compiler.

1 Introduction

Graphics processing units (GPUs) offer abundant, highly parallel computing resources. However, developing a usable programming model for GPU that exposes the hardware's full capacity has proven to be a challenge. The most popular languages for GPU programming, like CUDA C [NVI15], require the programmer to manage hardware resources at a low level in order to get good performance. For instance, a programmer might need to rewrite programs in unintuitive ways to produce faster memory access patterns, or reorder kernel calls and data transfers to hide the latency of communication between host and device.

NESL, a first-order functional programming language, can express data parallel computations clearly and concisely. NESL also supports *nested data parallelism* (NDP), meaning that parallel computations themselves may be applied in parallel [Ble95, BCH$^+$94].

This makes NESL a natural fit for irregular parallel algorithms that cannot be easily expressed in lower-level data-parallel languages. Although NDP cannot be executed on GPUs directly, the *flattening* transformation eliminates all nested parallel expressions; the resulting program expresses all parallel computations through flat and segmented primitives that can be implemented efficiently for GPU [BS90, Kel99]. By relying on libraries like Thrust [HB11] or CUDPP [cud] to provide these primitives, it is fairly easy to implement NESL on GPU. This approach led to promising preliminary results in [BR12]. However, a straightforward mapping from flattened NESL to CUDA leads to large performance overheads; the flattened code must be further optimized to achieve comparable performance to hand-crafted CUDA.

To address this problem, we are working on Nessie, a new NESL-to-CUDA compiler. Nessie relies on several different intermediate representations, each of which lends itself to a different set of optimizations. We discuss the design of the compiler, focusing on our implementation of the flattening transformation, and the additional analysis and optimization passes used to produce efficient CUDA code.

The rest of the paper is organized as follows. In the next section, we describe the NESL language. In Section 3, we give an overview of the Nessie compiler. In Section 4, we discuss the flattening transformation in detail. Section 5 discusses optimization and analysis of the flattened code, particularly our approach to shape analysis and kernel fusion. Section 6 describes the code generation phase of the compiler. comparing it to other implementations of NESL for GPU and to hand-crafted CUDA. Section 7 examines related work, and Section 8 discusses the current state of the compiler and future work. Complete source code for the Nessie compiler is available at `https://smlnj-gforge.cs.uchicago.edu/projects/neslgpu`.

2 NESL

NESL supports data parallelism in two ways: through an *apply-to-each construct* 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:

```
function square_sequence(xs) =
    {x * x : x in xs};
```

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

```
function multiply_sequences(xs, ys) =
    {x * 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:

```
function multiply_positive(xs, ys) =
   {x * y : x in xs; y in ys | x >= 0 and y >= 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:

$$\begin{array}{rcll} b & ::= & \texttt{INT}|\texttt{BOOL}|\texttt{FLOAT}|\texttt{CHAR} & \text{(base types)} \\ \tau & ::= & b & \\ & | & (\tau_1, \tau_2) & \text{(pairs)} \\ & | & [\tau] & \text{(sequences)} \end{array}$$

Figure 1: Subset of the NESL type system.

```
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; this is the inner layer of parallelism.

2.1 NESL Types

The NESL type system includes base types (e.g. chars, floats, integers), pairs, and sequences. NESL also permits user-defined record types, which cannot be recursive; records are desugared into nested pairs in the early stages of the compiler. All parallel operations in NESL are oriented around sequences, and all elements of a sequence must be of the same type. Elements of a sequence or pair may be of any type, so sequences and pairs may be arbitrarily nested. For instance, a sequence might have type [([INT], [BOOL])]; this would be a sequence of pairs of sequences. A subset of the NESL type system is given in Figure 1; we refer to the notation introduced here in Section 4.

2.2 The NESL Basis

In addition to the apply-to-each, NESL provides several primitive operations on sequences. Parallel NESL primitives can be divided into several different categories:

- *Generators.* Operations that take scalar arguments and generate sequences. Two generators are available in NESL. The `dist` primitive takes a value v and length l, and distributes v over a sequence of length l:

  `dist(2, 5)` $\equiv$ `[2, 2, 2, 2, 2]`

  The `index` primitive generates a sequence of integers between specified start and end points, increasing by some stride:

  `index(1, 2, 10)` $\equiv$ `[1, 3, 5, 7, 9]`

- *Reductions.* Operations that fold an associative binary operator over a sequence and return a scalar. For example, `sum` is a reduction using the + operator, and the `product` reduction uses the `*` operator:

`product([2, 3, 4])` $\equiv$ `24`

Likewise, `any` uses the boolean `OR` operator:

`any([F, F, T, F, F])` $\equiv$ `T`

- *Scans.* Operations that apply an associative binary operator over a sequence to compute a generalized prefix sum. In other words, given some operator $\otimes$, $\otimes\_scan$ will return a sequence, where the i^{th} element e_i of the sequence is the $\otimes$ reduction of elements e_1 through $ei - 1$. (The first element of the result will always be the identity.) The `mul_scan` and `or_scan` operations correspond to `product` and `any` in the above example.

  `mul_scan([2, 3, 4])` $\equiv$ `[1, 2, 6]`

  `or_scan([F, F, T, F, F])` $\equiv$ `[F, F, F, T, T]`

- *Reordering operations.* This category includes operations that may change the relative order of the elements in a sequence, such as `permute` and `reverse`, and operations whose result is a different length from the inputs, including `concat`, which concatenates two sequences, and `pack`, which takes a sequence of values and a sequence of flags, and returns the values corresponding to the true flags.

  `pack([1, 2, 3], [T, F, T])` $\equiv$ `[1, 3]`

- *Operations on nesting structure.* The `partition` operation adds a nesting layer to a sequence according to a given sequence of segment lengths, and `flatten` eliminates a nesting layer.

  `partition([4, 5, 6], [2, 1])` $\equiv$ `[[4, 5], [6]]`

  The `zip` operation converts a pair of sequences into a sequence of pairs, and `unzip` does the reverse.

  `zip([1, 2, 3], [4, 5, 6])` $\equiv$ `[(1, 4), (2, 5), (3, 6)]`

- *Element extraction and update.* The `get` operation extracts a sequence element at a given position, and `put` updates a sequence element at a given position.

3 The Nessie Compiler

The Nessie compiler translates NESL source code into CUDA through a series of program transformations. This section provides an overview of these transformations, which are outlined in Figure 2.

Basis Library The basis library defines all primitives available in the surface language. Basis library functions are written in NESL, but can access certain constructs that are not available in the surface language. First of all, many basis functions are defined in terms of *pure* operators, a more restricted set of primitives based on Guy Blelloch's VCODE operators [BC90]. Second, certain basis functions need to be able to directly manipulate *segment descriptors*, which are used

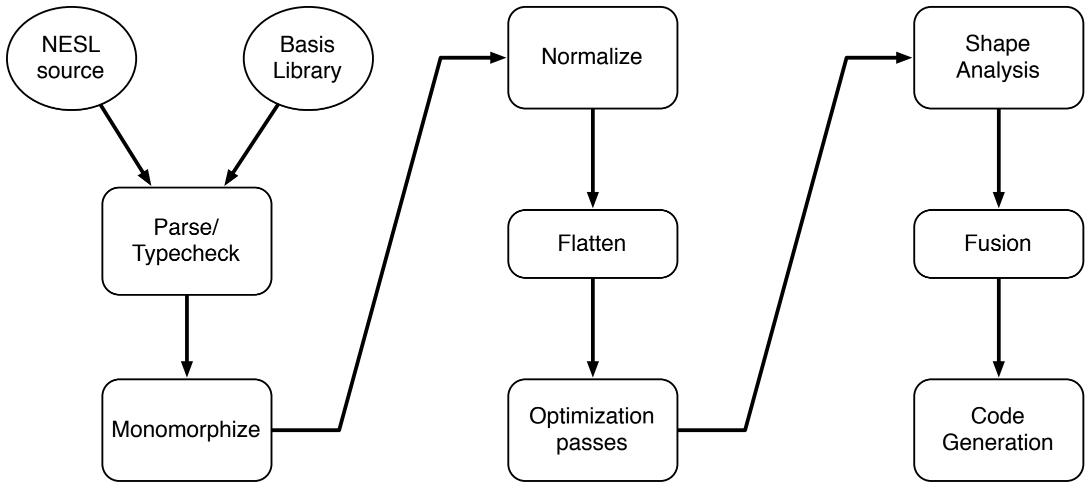

Figure 2: The stages of the Nessie compiler.

internally to describe the nesting structure of sequences. The `__vector` construct allows basis functions to decompose a sequence into its segment descriptor and data. Finally, basis functions may use `base-typecase` and `poly-typecase` constructs to define their behavior on different types. This mechanism provides a simple way to express the instantiation rules for operations on nested sequences and sequences of pairs, as outlined in [Kel99] Our basis library code is ported with extensive modifications from Blelloch's NESL compiler.

Parsing/Typechecking The parser reads in the source code and basis library and generates an abstract syntax tree. The typechecking pass then ensures type correctness.

Monomorphization The monomorphizaton pass instantiates all polymorphic functions as needed; this eliminates all `base-typecase` and `poly-typecase` constructs from library functions, and produces a monomorphic IR, monoAST.

Normalization The normalization pass simplifies the program; we require all non-atomic expressions to be either let-bound or tail expressions. We also eliminate tuple unpacking, instead introducing an explicit projection primitive.

Flattening The flattening transformation eliminates apply-to-each constructs. This transformation ensures that the program contains no nested parallelism, and can therefore be executed on SIMT architectures. Flattening produces an intermediate representation in a flattened internal language called Flan. The flattening transformation is discussed in more detail in Section 4.

Fusion After shape analysis, the program is converted into another internal representation, Fuse-AST. Using the information gathered during shape analysis, we eliminate unnecessary segment

$$
\begin{array}{lcll}
b & ::= & \texttt{INT}|\texttt{BOOL}|\texttt{FLOAT}|\texttt{CHAR} & \text{(base types)} \\
s & ::= & < i_1, \ldots, i_n > & \text{(segment descriptors)} \\
\tau & ::= & b & \\
 & | & s & \\
 & | & [\!: b :\!] & \text{(base sequences)} \\
 & | & (t_1, t_2) & \text{(pairs)}
\end{array}
$$

Figure 3: The Flan type system.

$$
\begin{array}{lcl}
\mathbf{F}[\![b]\!] & = & b \\
\mathbf{F}[\![(t_1, t_2)]\!] & = & (\mathbf{F}[\![t_1]\!], \mathbf{F}[\![t_2]\!]) \\
\mathbf{F}[\![[t]]\!] & = & \mathbf{S}[\![[t]]\!]
\end{array}
$$

$\mathbf{F}[\![.]\!]$ is the translation from NESL to Flan types.

$$
\begin{array}{lcl}
\mathbf{S}[\![b]\!] & = & [\!: b :\!] \\
\mathbf{S}[\![(t_1, t2)]\!] & = & (\mathbf{S}[\![t_1]\!], \mathbf{S}[\![t_2]\!]) \\
\mathbf{S}[\![[t]]\!] & = & (s, \mathbf{S}[\![t]\!])
\end{array}
$$

$\mathbf{S}[\![.]\!]$ is the translation of NESL sequence types into Flan.

Figure 4: Translation from NESL to Flan types.

descriptors and fuse parallel primitives into larger parallel map expressions. The fusion phase is described in more detail in Section 5.3.

Code Generation The program is translated into one more intermediate representation, λ_{cu}. In λ_{cu}, we express GPU operations using task graphs, which allows us to fuse more complex operations, including generators, scans and reductions. Finally, we generate CUDA kernels and C host functions to call them. Code generation is discussed in more detail in Section 6.

4 Flattening

Because GPUs cannot execute nested parallel operations directly, we use the flattening transformation to map the NESL program to a SIMT execution model. Flattening systematically eliminates all apply-to-each constructs, instead relying on an extended set of parallel primitives to express the necessary operations. Guy Blelloch introduced the flattening transformation in his NESL compiler for wide-vector machines. Blelloch's implementation of flattening produced VCODE, a virtual stack-machine language that operates on vectors [BCH$^+$94]. However, stack-machine code does not lend itself to the program analyses or transformations necessary to generate efficient CUDA. Our implementation of flattening targets a new intermediate representation, called Flan. We discuss Flan and our approach to flattening below. We do not describe the flattening transformation rules themselves, as Keller lays them out very clearly in [Kel99].

4.1 Flan Types

In Flan, a sequence is decomposed into two parts: a flat *base sequence*, which contains elements of some base type, and one or more *segment descriptors* encoding the nesting structure of the sequence. For the purpose of this section we can assume that a segment descriptor is an array of integers, where each integer corresponds to the length of one segment. Every sequence needs at least one segment descriptor, with an additional segment descriptor for each level of nesting. For example, the NESL array

```
[1, 2, 3]
```

would be represented in Flan as a single segment descriptor paired with a base sequence of integers:

```
(<3>, [: 1, 2, 3 :])
```

For every additional level of nesting, a new segment descriptor is added:

```
[[1, 2, 3], [4, 5]]
```

would become:

```
(<2>, (<3, 2>, [: 1, 2, 3, 4, 5 :]))
```

At the outermost layer, the sequence is a single segment of length 2, because it contains two subsequences. At the inner layer, we have two segments of lengths 3 and 2, respectively. This accounts for the inner segment descriptor, which is paired with the actual base sequence.

Sequences of Pairs Although a sequence may have elements of pair type, allowing base sequences to contain pairs is a bad idea. First of all, we cannot directly perform any parallel operations on sequences of pairs. Secondly, the elements of a pair may themselves be sequences. We wish to make the sequence's nesting structure explicit through its segment descriptors, but we cannot do this if the base sequence may contain additional layers of nesting. We therefore desirable to represent a sequence of pairs as a pair of sequences, along with their common segment descriptor. For instance, the following NESL sequence:

```
[(1, a), (2, b), (3, c)]
```

would become:

```
(<3>, ([:1, 2, 3:], [:a, b, c:]))
```

This allows us to restrict base sequences to elements of base type, while maintaining distinct types for pairs of sequences and sequences of pairs. The Flan type system is defined in Figure 3, and the translation from NESL to Flan types is defined in Figure 4.

4.2 Flan Primitives

The set of Flan primitives is based on Blelloch's VCODE operators, and is generally similar to the NESL basis: it includes arithmetic operations, scans, reductions, permutations, and so on. Furthermore, Flan has a base and lifted version of each primitive. For every Flan primitive p, the equivalent lifted primitive, $p^{\uparrow}$, applies p to every element of a sequence in parallel. (Where a primitive takes multiple arguments, the lifted primitive takes multiple sequences, which must be of the same length.) These additional primitives allow us to preserve the meaning of the original program without the apply-to-each construct.

```
$\mathbf{C}[\![ [b_1, \ldots, b_n] ]\!]$   $\rightarrow [b_1, \ldots, b_n]$                     $b_1, \ldots, b_n$ are elements of base type
$\mathbf{C}[\![ [s_1, \ldots, s_n] ]\!]$   $\rightarrow$                                        $s_1, \ldots, s_n$ are elements of sequence type
        let  $len_1 = length(s_1)$;
        ...
        let  $len_n = length(s_n)$;
        let  $lengths = [len_1, \ldots, len_n]$;
        let  $ss = s_1 ++ \ldots ++ s_n$;                    (sequence concatenation)
        in
          $partition(ss, lengths)$;
$\mathbf{C}[\![ [t_1, \ldots, t_n] ]\!]$   $\rightarrow$                                        $t_1, \ldots, t_n$ are elements of pair type
        let  $t_{1,1} = \#1\ t_1$;
        ...
        let  $t_{n,1} = \#1\ t_n$;
        let  $t_{1,2} = \#2\ t_1$;
        ...
        let  $t_{n,2} = \#2\ t_n$;
        let  $seq_1 = \mathbf{C}[\![ [t_{1,1}, \ldots, t_{n,1}] ]\!]$
        let  $seq_2 = \mathbf{C}[\![ [t_{1,2}, \ldots, t_{n,2}] ]\!]$
        in
          $zip(seq_1, seq_2)$;
```

Figure 5: Flattening transformation on sequence constructors. Sequence construction using base type elements is unchanged. To construct a sequence of sequences, we must concatenate all of the elements, then partition the result. To construct a sequence of pairs, we recursively construct a sequence of the first element of each pair and another of the second element of each pair, then zip them together.

Just as the set of base Flan primitives contains scalar and parallel operations, we can divide the corresponding lifted primitives into elementwise and segmented operations. Lifted scalar operations are elementwise; they are applied to flat sequences. Lifted sequence operations are segmented; they are applied to segmented sequences, which have exactly one level of nesting. For instance, the lifted `+_REDUCE` primitive computes a segmented sum:

`+_REDUCE`$^{\uparrow}$`(<2>, (<3, 2>, [:1, 2, 3, 4, 5:]))` $\equiv$ `(<2>, [:6, 9:])`

In the above example, the inner segment descriptor is clearly necessary; it tells us to sum the first three elements and last two elements separately. But we don't actually need the outer segment descriptor. In fact, none of the outermost segment descriptors are necessary. We retain them in Flan because they simplify the flattening transformation and are helpful during shape analysis, but they are eliminated in the conversion to FuseAST.

4.3 Sequence construction

NESL includes a sequence construction operation which allows the programmer to explicitly list the elements of a sequence:

```
v = [1, 2, 3];
```

When the sequence elements are of base type (as in the example above), this can be expressed in Flan directly; ultimately it will just be a CPU-side array declaration. But when the sequence elements are themselves sequences, this becomes a parallel operation, which should be expressed using parallel primitives. Likewise, when the sequence elements are pairs, we need to ensure that the result follows the representation of sequences of pairs discussed in Section 4.1. Keller's flattening transformation rules all transform apply-to-each expressions; to correctly handle sequence construction in Flan, we extend the flattening transformation with a set of rules to transform sequence constructors. These rules, which are outlined in Figure 5, ensure that sequence constructors in Flan will only take elements of base type.

4.4 Operations that Cannot Be Flattened

A few basis library functions–`partition`, `flatten`, `length`, `zip`, and `unzip`–must operate directly on base sequences and segment descriptors. Because segment descriptors do not appear in the surface language, and it would not be correct to introduce sequences of segment descriptors, we cannot apply the flattening transformation to these functions. Instead, we have added handwritten lifted versions of these functions to the basis library. When our flattening pass encounters one of these functions, it pulls in its lifted equivalent instead of flattening it directly.

5 Optimizations

The Flan programs produced by the flattening transformation could be translated into CUDA without much additional modification, but the result would be inefficient. The most significant issue is that every parallel Flan primitive would become a single CUDA kernel, resulting in frequent data transfers between host and device. The program would also contain many unnecessary segment descriptors, since operations on flat vectors don't need segment descriptors at all, and many segment descriptors are identical. These useless segment descriptors lead to even more data transfers and increased memory pressure on the GPU. To address these issues, we introduce an additional IR, FuseAST, that facilitates kernel fusion and redundant segment descriptor elimination. We also apply a suite of optimizations to the Flan representation of a program, whose primary purpose is to expose more opportunities for optimization in FuseAST. This section discusses optimizations to both IRs.

5.1 Flan Optimizations

We apply several standard optimizations to the Flan IR. Although these optimizations are not specific to parallel languages, they are especially important here. In particular, aggressive inlining exposes more opportunities for kernel fusion and segment descriptor elimination. Contraction and arity-raising, in turn, provide more opportunities to inline.

Contraction The contraction pass combines several optimizations, including dead code elimination; copy propagation; and inlining of functions that are called only once. We also perform constant folding over tuple projections. For instance, the expression

$$
\begin{array}{rcll}
\mathbb{S} & ::= & \texttt{SCALAR}(p) & \text{(parameter is value of scalar)} \\
 & | & \texttt{VEC}(p_1, p_2) & \text{(first parameter is length, second is value)} \\
 & | & \texttt{UNIFORM\_VEC}(p_1, p_2) & \text{(first parameter is length, second is value of each element)} \\
 & | & \texttt{PAIR}(s_1, s_2) & \text{(recursively define another shape for each pair element.)}
\end{array}
$$

Figure 6: Encoding of Flan shapes

```
tup = (1, 2, 3);
y = #3 tup;
```

will be simplified to

```
y = 3;
```

Inlining In addition to the conservative inlining during contraction, we run a separate pass to aggressively inline every function below a certain size threshold. It is especially important to inline lifted primitives, so we can fuse them later. We also want to inline `zip` and `unzip`. Once sequences of pairs have been converted to their Flan representation (discussed in Section 4.1), these functions just perform pair selection and construction. once they are inlined, we can eliminate most or all of their bodies in a subsequent contraction pass.

Arity-raising Initially, every function in the Flan IR has a single argument; where the surface language function takes multiple arguments, the Flan function takes a tuple of arguments (represented as a nested pair). This simplifies the flattening transformation, but results in unnecessary tuple construction before function calls, and unnecessary tuple projection in function bodies. In the arity-raising pass, we expand a single tuple of arguments back into a list of arguments.

5.2 Shape Analysis

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

- 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 5.3).
- If we know that two segment descriptors are identical, we can eliminate one of them.
- If we know that a segment descriptor is *uniform*–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 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, `DIST` and `INDEX` take scalar arguments that specify the length of the result.

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 has a unique ID and an optional concrete value. Our full system for encoding shape information is given in Figure 6.

Roughly speaking, a shape parameter may determine a variable's length or its 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, but provides no further information about it. In this 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 base sequence, 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 `VEC(1, 2)`, because it has one element, and that element's value is 2; the inner segment descriptor would have shape `UNIFORM_VEC(2, 3)`, because it has two elements and both of them have value 3; and the base sequence would have shape `VEC(6, UNKNOWN)`, since it has six elements and it is not uniform.

The shape analysis pass has three basic phases. In the first phase, we traverse the program's AST, assigning shapes to variables and setting constraints between those shapes. In the second phase, we solve these constraints, partitioning our set of shape parameters into equivalence classes, and updating the values of the parameters accordingly. (In particular, if we know some shape parameter has constant value C, we associate C with each parameter in the same equivalence class.) Finally, we use this solution to update the shape of each variable, replacing each shape parameter with a representative member of its equivalence class. Thus, two identical sequences or segment descriptors will have identical shapes; two sequences that are the same length but may not have the same values might have shapes `VEC(a, b)` and `VEC(a, c)`, where `a`, `b` and `c` are shape parameters.

5.3 Fusion

After optimizing and analyzing our Flan program, we convert it to a new IR, FuseAST. In FuseAST, we no longer use lifted primitives to express elementwise operations. Instead, we introduce kernels–functions that contain only scalar operations. A kernel is applied in parallel to each element of a sequence using the *map* construct. For example, the Flan expression

`v = a` $+^{\uparrow} b$`;`

would become:

```
kernel PLUS(x, y) = x + y;
v = map PLUS (a, b);
```

In FuseAST, it is possible to perform *kernel fusion*, merging two kernels into a larger one. Whenever an argument to a kernel is itself the result of a previous kernel invocation, we can merge the current and previous kernel. For example:

```
kernel PLUS(x, y) = x + y;
kernel TIMES(a, b) = a * b;

tmp = map PLUS (xs, ys);
result = map TIMES(tmp, zs)
```

can be fused into:

```
kernel PLUS_TIMES(y', a', b') =
  let x = a' + b'
  in x + y';

result = map PLUS_TIMES(xs, ys, zs);
```

We avoid creating temporary sequence `tmp`, and passing `tmp` between the host and device. We also avoid the additional overhead of the second kernel invocation.

Redundant argument elimination We also specialize kernels to eliminate redundant arguments. For example:

```
kernel k (a, b) = a + b;
result = map k (as, as);
```

is specialized to:

```
kernel k' (a) = a + a;
result = map k' as;
```

We can now copy `a` from host to device just once, instead of twice.

Contraction and segment descriptor elimination Contraction in FuseAST is similar to contraction in Flan. However, we do not attempt to inline, and we use the information from shape analysis to replace redundant segment descriptors. If we find that a segment descriptor s is identical to a previously defined segment descriptor t, we can eliminate s, and replace all references to s with references to t.

6 Code Generation

The last phase of the compiler is code generation. This phase is not fully implemented yet, but we provide an overview of our current design. Code generation proceeds in two steps: we first convert our program to its final IR, λ_{cu}, and then from λ_{cu} to CUDA. λ_{cu} is a two-level representation; it represents CPU-side computation with an AST, similar to those used in earlier phases, and GPU-side computation with a task graph.

Constructing the task graphs A task graph represents a set of computations over one or more sequences of the same shape. The sources of the graph may be generators (`DIST` or `INDEX`) or sequences. Similarly, the sinks of the graph may be reduction operators or sequences. Internal nodes may be parallel maps, or flat or segmented scans. To construct the task graphs, we create a node for each invocation of a generator, map, reduce or scan operation, and add edges from each node to each of its consumers. This gives us a single graph with several connected components; we treat each connected component as a separate task graph in the λ_{cu} representation.

Variable reuse Sometimes the memory allocated for the inputs to a task graph can also be used to store its results. We perform a liveness analysis of the λ_{cu} program, which allows us to determine which variables are still live after any expression is executed. Whenever some variable v is passed as input to a task graph, we check whether it is still live after the task graph is run. If not, v is available for reuse. If any result of the task graph is the same length as v, we can bind the result to v instead of introducing a new variable.

Generating the CUDA kernels Our CUDA generation pass currently supports task graphs containing only flat generators and maps. Each of these task graphs can be transformed into a single CUDA kernel, because no communication between threads is required. A task graph containing scans and reductions, on the other hand, requires synchronization. Unless the inputs are small enough to be processed within a single thread block, scans and reductions require multiple kernel invocations to achieve global synchronization. If the source of a reduction is another node in the same task graph (i.e. a generator or map), that node can be fused with the first kernel invocation of the reduction. The same is true for scans. Because reordering operations cannot be fused with other parallel operations, they are not included in the task graph. Instead, they are implemented using custom library functions.

7 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]. However, DPH targets multicore CPUs, rather than GPUs.

NOVA is another high-level, functional language for GPU programming [CGG+13]. Like Nessie, the NOVA compiler takes advantage of a simple, functional intermediate language (although in this case the intermediate language is NOVA itself) in order to implement a range of optimization passes. NOVA 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. NOVA targets sequential, multi-threaded, and GPU architectures.

8 Status and future work

We have implemented the Nessie front end, the flattening transformation on apply-to-each expressions, and the optimizations discussed in Section 5. We are currently working on the code generation phase, and expect to be able to compile simple examples (without scans, reductions, reordering or segmented operations) within the week. In order to compile and run our full suite of benchmarks, we still need to add several functions to the NESL basis library, implement the flattening transformation on sequence construction (Figure 5), and extend code generation to support more complex task graphs. In addition, we hope to add support for the following optimizations:

- *Vectorization Avoidance.* If an apply-to-each construct contains only scalar subexpressions, it does not need to be flattened; it can essentially be translated directly into a "pre-fused" FuseAST kernel. Vectorization avoidance is a technique that allows us to identify such apply-to-each expressions, and avoid flattening them [KCL$^+$12]. This makes it possible to produce kernels with conditional branches, whereas the full flattening transformation eliminates all conditionals within parallel operations. Because flattening conditionals results in multiple expensive `PACK` operations, it is often more efficient to allow divergent control flow within kernels.

- *Fusing independent maps.* When two independent FuseAST kernels operate on sequences of equal width, they can be fused into a single kernel that returns multiple results. This optimization provides additional opportunities for redundant argument elimination, as described in Section 5.3, particularly when two kernels consume the same sequence.

- *Piecewise execution.* Palmer et. al. observe that flattening can introduce extremely large intermediate sequences, leading to excessive memory requirements [PPCF95]. To solve this problem, they propose partially serializing flattened programs through a technique they call piecewise execution. Madsen and Filinski apply this method on the GPU [MF13]. The design of λ_{cu} would make it relatively easy to add support for piecewise execution.

- *Concurrent GPU tasks.* CUDA supports limited task parallelism, allowing several kernels, one or two data transfers, and CPU computation to be executed simultaneously on recent architectures [NVI15]. During code generation, we would like to identify opportunities for task parallelism, and schedule data transfers and kernel invocations to take advantage of these opportunities.

On the whole, we have found that the compiler design presented in this paper makes it quite easy to incorporate additional optimizations and program analyses as the need for them arises. We are optimistic that this will allow the Nessie compiler to generate efficient CUDA and provide a solid foundation for future research into compilation techniques for nested parallelism.

Acknowledgments

I would like to thank John Reppy for his extensive guidance and support in the process of writing this thesis. I would also like to thank Lars Bergstrom for his helpful advice on code generation.

References

[BC90] Blelloch, G. and S. Chatterjee. VCODE: A data-parallel intermediate language. In *FOMPC3*, 1990, pp. 471–480.

[BCH+94] Blelloch, G. E., S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *JPDC*, **21**(1), 1994, pp. 4–14.

[Ble95] Blelloch, G. E. NESL: A nested data-parallel language (version 3.1). *Technical Report CMU-CS-95-170*, School of C.S., CMU, Pittsburgh, PA, September 1995.

[BR12] Bergstrom, L. and J. Reppy. Nested data-parallelism on the GPU. In *ICFP '12*, Copenhagen, Denmark, September 2012. ACM, pp. 247–258.

[BS90] Blelloch, G. E. and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *JPDC*, **8**(2), 1990, pp. 119–134.

[CGG+13] Collins, A., D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A functional language for data parallelism. *Technical Report NVR-2013-002*, NVIDIA, July 2013.

[CGK11] Catanzaro, B., M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *PPoPP '11*, San Antonio, TX, February 2011. ACM, pp. 47–56.

[CLPK08] Chakravarty, M. M. T., R. Leshchinskiy, S. Peyton Jones, and G. Keller. Partial vectorisation of Haskell programs. In *DAMP '08*. ACM, January 2008, pp. 2–16. Available from `http://clip.dia.fi.upm.es/Conferences/DAMP08/`.

[cud] Cudpp. Available from `http://cudpp.github.io`.

[HB11] Hoberock, J. and N. Bell. Thrust: A productivity-oriented library for CUDA. In W. W. Hwu (ed.), *GPU Computing Gems, Jade Edition*, chapter 26, pp. 359–372. Morgan Kaufmann Publishers, October 2011.

[KCL+12] Keller, G., M. M. T. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. Peyton Jones. Vectorisation Avoidance. In *HASKELL '12*, Copenhagen, Denmark, September 2012. ACM. Forthcoming.

[Kel99] Keller, G. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. Ph.D. dissertation, Technische Universität Berlin, Berlin, Germany, 1999.

[LCK06] Leshchinskiy, R., M. M. T. Chakravarty, and G. Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra (eds.), *ICCS '06*, number 3992 in LNCS. Springer-Verlag, May 2006, pp. 920–928.

[MF13] Madsen, F. M. and A. Filinski. Towards a streaming model for nested data parallelism. In *FHPC '13*, Boston, MA, September 2013. ACM, pp. 13–24.

[NVI15] NVIDIA. NVIDIA CUDA C Programming Guide, September 2015. Available from `http://docs.nvidia.com/cuda/index.html`.

[PPCF95] Palmer, D. W., J. F. Prins, S. Chatterjee, and R. E. Faith. Piecewise execution of nested data-parallel programs. In *Languages and Compilers for Parallel Computing*, vol. 1033 of *LNCS*. Springer-Verlag, 1995, pp. 346–361.

[ZM12] Zhang, Y. and F. Mueller. CuNesl: Compiling nested data-parallel languages for SIMT architectures. In *ICPP '12*, Pittsburgh, PA, September 2012. IEEE Computer Society Press, pp. 340–349.