λ_{cu}— An Intermediate Representation for Compiling Nested Data Parallelism

John Reppy and Joe Wingerter

University of Chicago

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 [**?**], 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 [**?**]. 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 challenge, we have been implementing a new NESL compiler that provides a platform for exploring optimization techniques for NDP on GPUs.

A major part of this new compiler is the design of an intermediate representation for the FDP code, which is the topic of this paper. Our IR, which is called λ_{cu}, has a number of useful properties:

- λ_{cu} explicitly distinguishes between CPU and GPU data and computations.
- λ_{cu} supports easy analysis to enable kernel fusion.
- λ_{cu} provides a uniform framework for applying implementing a rich set of different flavors of fusion
- λ_{cu} provides a modular implementation of code generation that avoids the combinatorial complexity of supporting many different specializations of kernels.
- λ_{cu} is designed to support future improvements to the compiler.

In this paper, we present the design of λ_{cu}, describe the techniques that we use to support fusion, and how the design of λ_{cu} enables modular code generation.

2 Background

In this section, we describe the context in which we are operating, including a quick introduction to NESL, a brief description of the CUDA programming model for GPUs, and an overview of our compiler's architecture.

2.1 NESL

NESL is a first-order data-parallel functional language that supports data parallelism in two ways: through a parallel array comprehension (*apply-to-each*) and through a set of parallel primitive operators. Apply-to-each allows the programmer to map an arbitrary computation over a vector. For example, the following function squares each element of the vector `xs`:

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

It is also possible to map a computation over multiple vectors of the same length, as in this example that computes the element-wise product of `xs` and `ys`.

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

An apply-to-each may include an optional filter to specify which elements to apply the computation to; for example, we could compute the product of the positive elements of `xs` and `ys` as follows:

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

The computations that are mapped by apply-to-each may themselves be parallel computations, which is called *nested-data parallelism* (NDP). 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:

```
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. Within each call to `dotp`, the elements of the row are all multiplied in parallel, then added up with the parallel `sum` operation.

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

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

This function has the same nested structure as before, but the amount of work per row varies.

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 [**?**].

2.2 GPU hardware and programming model

Graphics processing units (GPUs) are high-performance parallel processors that were originally designed for computer graphics applications, but are now often used for other computational tasks. GPU applications are typically written in either CUDA [**?**] or OpenCL [**?**], which been developed for the general-purpose programming of GPUs. While these languages provide a C-like expression and statement syntax, they expose many of the peculiarities of the GPU hardware model.

A typical GPU consists of multiple streaming multiprocessors (SM), each of which contains multiple computational cores. An SM executes a group of threads, called a *warp*, in parallel, with one thread per computational core. Execution is *Single-Instruction-Multiple-Thread* (SIMT), which means that each thread in the warp executes that same instruction. To handle divergent conditionals, GPUs execute each branch of the conditional in series using a bit mask to disable those threads that took the other branch. An SM can efficiently switch between different warps, which allows it to hide memory latency. GPUs have some hardware support for synchronization, such as per-thread-group barrier synchronization and atomic memory operations. One of the major differences between GPUs and CPUs is that the memory hierarchy on a GPU is explicit, consisting of a global memory that is shared by all of the SMPs, a per-SM local memory, and a per-core private memory.

A function that runs on the GPU is called a *kernel*. The host CPU invokes the kernel and specifies the configuration of the execution, which is a 1, 2, or 3D grid structure onto which the parallel threads are mapped. This grid structure is divided into blocks, with each block of threads being mapped to the same SM. The explicit memory hierarchy is also part of the CUDA programming model, with pointer types being annotated with their address space (*e.g.*, global vs. local).

The various features of the GPU hardware and programming models discussed above pose a number of challenges to the effective use of GPU hardware. These challenges include:

Memory access Within a GPU kernel, access to the global memory is significantly slower than access to local or private memory. Furthermore, GPU global memory performance is very sensitive to the patterns of memory accesses across the warp [**?**].

Control divergence Threads within a warp must execute the same instruction each cycle. When execution encounters divergent control flow, the SM is forced to execute both control-flow paths to the point where they join back together. Thus care must be taken to reduce the occurrence of divergent conditionals.

No recursive calls Recursion is not, in general, permitted in GPU kernels.[1] This limitation means that any program with recursive calls must be transformed into a non-recursive one [**?**,**?**]. The flattening transformation in our compiler results in a program where the recursion is confined to the CPU.

2.3 Compiling NESL

Our NESL compiler, called Nessie, is organized as a series of transformations between intermediate languages. Included in these is a monomorphization pass that instantiates all polymorphic functions to monomorphic types, producing a monomorphically-typed AST. We apply the flattening transformation to the monomorphic code. Flattening is a global program transformation that converts nested data parallelism into flat data parallelism [**?**,**?**,**?**]. 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. Details about these aspects of the Nessie compiler can be found in Sandler's Honors Thesis [**?**]. The output of the flattening transformation is a sequential program that controls the execution of data-parallel vector operations. This program is annotated with *shape* information that allows the compiler to determine when computations over different vectors are compatible. Producing efficient code from this representation requires aggressive fusion of the data-parallel operations, as well as optimizations of memory access patterns and memory management.

The focus of this paper is on the representation of the program after flattening and the stages needed to transform that representation to efficient CUDA code.

3 The λ_{cu} IR

λ_{cu} is an explicitly-typed, monomorphic, three-level language consisting of a top-level representation for the CPU-level control flow, a mid-level language for representing the iteration structure of GPU kernels, and a low-level language for representing the computations performed on the GPU. Kernels in the mid-level language are represented as a sequence of *Second-Order Array Combinators* (SOAC), which define the iteration structure of the various stages of the kernel. SOACs have been used in a number of previous implementations of data-parallelism [**?**,**?**]; our approach differs from this past work in a couple of aspects. First, every SOAC has a *pull-thunk* argument that abstracts the way that data is acquired. The presence of the pull-thunk allows us to fuse any SOAC with a proceeding map combinator. The other difference is that unlike previous uses of SOACs, the pull-thunk iterates over the index space of the computation instead of the value space. Abstracting over indices enables greater expressiveness in the design and more opportunities for kernel fusion. In this section, we present the abstract syntax of λ_{cu} and give a brief introduction to its static and dynamic semantics.

3.1 Abstract syntax

We assume a countable set of variables (VAR), which we denote by x, y, and z, with subscripts and decorations as necessary. We use the notation $\overline{x_i}$ to denote a sequence of xs indexed by i, and we write xs for $\overline{x_i}$ when we do not need the index.

Figure 1 gives the abstract syntax for the three levels of λ_{cu}. The CPU language (Figure 1a) defines a

$$
\begin{aligned}
\mathit{prog} &::= \overline{\mathit{kern}}\ \mathit{dcl} \\
\mathit{dcl} &::= \texttt{function}\ \mathrm{f}\ \texttt{(}\ \mathit{params}\ \texttt{)}\ \mathit{blk}\ \mathit{dcl} \\
&::= \texttt{let}\ \mathit{params}\ \texttt{=}\ \mathit{exp}\ \mathit{dcl} \\
&::= \mathit{exp} \\
\mathit{params} &::= \overline{x_i : \chi_i} \\
\mathit{blk} &::= \{\ \overline{\mathit{bind}}\ \mathit{exp}\ \} \\
\mathit{bind} &::= \texttt{let}\ \mathit{params}\ \texttt{=}\ \mathit{exp} \\
\mathit{exp} &::= \mathit{blk} \\
&\quad \mid \texttt{run}\ \mathrm{K}\ \mathit{args} \\
&\quad \mid \mathrm{f}\ \mathit{args} \\
&\quad \mid \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{blk}\ \texttt{else}\ \mathit{blk} \\
&\quad \mid \mathit{exp} \odot \mathit{exp} \\
&\quad \mid \cdots
\end{aligned}
$$

(a) CPU-expression syntax

$$
\begin{aligned}
\mathit{kern} &::= \texttt{kernel}\ \mathrm{K}\ \mathit{xs}\ \{\ \overline{\mathit{bind}}\ \texttt{return}\ \mathit{ys}\ \} \\
\mathit{bind} &::= \texttt{let}\ \mathit{xs}\ \texttt{=}\ \mathit{SOAC}\ \overline{\mathit{arg}} \\
\mathit{SOAC} &::= \texttt{MAP} \mid \texttt{PERMUTE} \mid \texttt{REDUCE} \mid \texttt{SCAN} \\
&\quad \mid \texttt{FILTER} \mid \texttt{PARTITION} \mid \texttt{ONCE} \\
\mathit{arg} &::= \Lambda \\
&\quad \mid \texttt{(}\ \mathit{xs}\ \texttt{)}
\end{aligned}
$$

(b) Kernel-expression syntax

$$
\begin{aligned}
\Lambda &::= \{\ \mathit{xs}\ \texttt{=>}\ \mathit{exp}\ \texttt{using}\ \mathit{ys}\ \} \\
\mathit{exp} &::= \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \\
&\quad \mid \texttt{let}\ \mathrm{x}\ \texttt{=}\ \mathit{exp}\ \texttt{in}\ \mathit{exp} \\
&\quad \mid \mathit{exp} \odot \mathit{exp} \\
&\quad \mid \mathrm{x}\texttt{[}\mathrm{i}\texttt{]} \\
&\quad \mid \mathrm{x} \\
&\quad \mid \cdots
\end{aligned}
$$

(c) GPU-expression syntax

Fig. 1. The abstract syntax of λ_{cu}

$$
\begin{array}{lll}
b ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{unit} & & \text{base types} \\
\tau ::= b & & \text{GPU-level types} \\
\quad \mid \tau \times \cdots \times \tau & & \\
\theta ::= b & & \text{Kernel-level types} \\
\quad \mid \theta \times \cdots \times \theta & & \\
\quad \mid \texttt{[}b\texttt{]} & & \\
\chi ::= b & & \text{CPU-level types} \\
\quad \mid \chi \times \cdots \times \chi & & \\
\quad \mid \texttt{[}b\texttt{]} & & \\
\quad \mid \&\texttt{[}b\texttt{]} & &
\end{array}
$$

Fig. 2. The λ_{cu} types

program as a collection of kernel definitions, followed by declarations. The CPU level language is fairly standard, except for the **run** K *args* expression, which is used to dispatch a kernel on the GPU. GPU kernels (Figure 1b) consist of a series of *Second-Order Array Combinator* (SOAC) applications. The SOACs provide an abstraction of data-parallel computations and are described below. A GPU thunk { *xs* **=>** *exp* **using** *ys* } is a first-order function with parameters xs, body exp, and free variables ys.[2] The only operation that works on vectors in the GPU expressions is subscripting; otherwise expressions are limited to base values and tuples.

3.2 SOACs

We define seven basic SOACs that provide support for most of NESL's features.[3] The thunk arguments to SOACs serve a number of distinct rôles in λ_{cu}. We use thunks to *pull* and transform values from vectors, we use them to implement predicates, and we use thunks to implement the monoid operators for the **SCAN** and **REDUCE** combinators. Because thunks can produce tuples of results, these SOACs can produce multiple result vectors. In the description below, however, we assume single results, since it simplifies the description and it is straightforward to generalize to multiple results. λ_{cu} provides the following SOACs:

MAP $\Lambda\ n$
: applies the pull thunk Λ to the range $0..n-1$ producing an n-element vector result. Note that by abstracting over the index space, instead of the value space, **MAP** subsumes the traditional vector-map operation, as well as generators like *iota* and the back-permute (or gather) operation.

PERMUTE $\Lambda_1\ \Lambda_2\ n$
: applies Λ_1 to the range $0..n-1$ to produce an n-element vector X, applies Λ_2 to the range to produce an n-element vector I of permutation indices, and then returns an n-element vector Y, such that $Y[I[i]] = X_k[i]$.

REDUCE $\Lambda_1\ v\ \Lambda_2\ n$
: applies the pull thunk Λ to the range $0..n-1$ producing an n-element vector result X and then reduces X to a scalar result using the monoid operator Λ_2.

SCAN $\Lambda_1\ v\ \Lambda_2\ n$
: is similar to **REDUCE**, but produces the exclusive prefix-scan using the monoid operator Λ_2.

FILTER $\Lambda_1\ \Lambda_2\ n$
: applies the pull thunk Λ to the range $0..n-1$ producing an n-element vector X. It then filters this vector by applying the predicate Λ_2 to produce a residual vector of unknown size.

[1] Recursion is supported on some newer cards, but only for kernels that may be called from other kernel functions — not the host.

[2] We borrow the idea of explicitly listing the free variables of a thunk from Madsen and Filinski [**?**], since it simplifies both the semantics and the fusion rules.

[3] There are some library functions, such as sorting, that get mapped to library kernels in our implementation. Segmented versions of the SOACs are discussed in Section 3.4.

PARTITION $\Lambda_1\ \Lambda_2\ n$
is similar to filter, except that it returns two vectors — one of true values and one of false values.

ONCE Λ
is used to evaluate sequential code on a single thread (strictly speaking, **ONCE** is not an array combinator).

Not that every SOAC (except **ONCE**) has a pull thunk that can represent mapping a computation over the inputs; this feature plays an important rôle in fusion (Section 4).

3.3 Types

The syntax of λ_{cu} types is given in Figure 2; as with the program syntax, types are stratified. Values of base type (b) are used on both the CPU and GPU. We use τ to denote the type of simple GPU values, which are used in thunks, and which include the base values and tuples. The tuple type operator $\times$ is associative (*i.e.*, tuples are not nested). Kernel-level types are denoted by τ and include both GPU-level types and vectors of base values. Lastly, CPU-level types include base values, CPU-side vectors, and references to GPU-side vectors.

Because NESL is first-order, we do not include function types in the type syntax. We do, however, use the following syntactic forms to describe the signatures of thunks and SOACs:

$$\begin{array}{lr} \vartheta ::= \tau \Rightarrow \tau & \text{thunk type} \\ \varsigma ::= \alpha \cdots \alpha \rightarrow \theta & \text{SOAC signature} \\ \alpha ::= \tau \mid \vartheta & \text{SOAC parameter type} \end{array}$$

The types of the SOACs are given in Figure 3. The notation $\tau^{\uparrow}$ denotes the lifting of a simple type τ into a sequence (or tuple of sequences):

$$\begin{aligned} b^{\uparrow} &= \texttt{[}b\texttt{]} \\ (\tau_1 \times \tau_2)^{\uparrow} &= \tau_1^{\uparrow} \times \tau_2^{\uparrow} \end{aligned}$$

The typing rules for λ_{cu} are not surprising; perhaps the two most interesting rules are the rule for typing SOAC applications,

$$\frac{\begin{array}{cc} \text{TypeOf}(\mathit{SOAC}) = \overline{\alpha_i} \rightarrow \theta & \overline{\Gamma \vdash \mathit{arg}_i : \alpha_i} \\ \theta = \theta_1 \times \cdots \times \theta_n & \Gamma' = \Gamma[x_j \mapsto \theta_j] \end{array}}{\Gamma \vdash \texttt{let}\ \overline{x_j}\ \texttt{=}\ \mathit{SOAC}\ \overline{\mathit{arg}_i} : \Gamma'}$$

and the rule for typing thunk arguments to an SOAC,

$$\frac{(\Gamma|_{ys})[x_i \mapsto \tau_i] \vdash e : \tau}{\Gamma \vdash \{\ \overline{x_i : \tau_i}\ \texttt{=>}\ e\ \texttt{using}\ ys\ \} : \tau_1 \times \cdots \tau_n \Rightarrow \tau}$$

3.4 Segmented operations

The previous discussion describes the flat version of λ_{cu} without the segmented arrays needed to implement NDP. Although we are still working out the details, we believe that the extension to segmented arrays is straightforward. The key observation is that for an operation f on vectors, the segmented version of f is just the piecewise application of f to each vector segment; *i.e.*,

$$\mathit{concat} \circ (\mathit{map}\ f) \circ \mathit{split}$$

To support segmented operations, we extend λ_{cu} with segmented versions of some of the SOACs (some of them, such as **MAP** do not have segmented versions). Figure 4 gives the type signatures of these segmented SOACs, where $\mathbf{sd}$ is the type of segment descriptors.

$$
\begin{aligned}
\texttt{MAP} &: (\mathbf{int} \Rightarrow \tau)\ \mathbf{int} \to \tau^{\uparrow} \\
\texttt{PERMUTE} &: (\mathbf{int} \Rightarrow \tau)\ (\mathbf{int} \Rightarrow \mathbf{int})\ \mathbf{int} \to \tau^{\uparrow} \\
\texttt{REDUCE} &: (\mathbf{int} \Rightarrow \tau)\ \tau\ (\tau \times \tau \Rightarrow \tau)\ \mathbf{int} \to \tau \\
\texttt{SCAN} &: (\mathbf{int} \Rightarrow \tau)\ \tau\ (\tau \times \tau \Rightarrow \tau)\ \mathbf{int} \to \tau^{\uparrow} \\
\texttt{FILTER} &: (\mathbf{int} \Rightarrow \tau)\ (\tau \Rightarrow \mathbf{bool})\ \mathbf{int} \to \tau^{\uparrow} \\
\texttt{PARTITION} &: (\mathbf{int} \Rightarrow \tau)\ (\tau \Rightarrow \mathbf{bool})\ \mathbf{int} \to \tau^{\uparrow} \times \tau^{\uparrow} \\
\texttt{ONCE} &: (\mathbf{unit} \Rightarrow \tau) \to \tau
\end{aligned}
$$

Fig. 3. SOAC type signatures

$$
\begin{aligned}
\texttt{SEG\_PERMUTE} &: (\mathbf{int} \Rightarrow \tau)\ (\mathbf{int} \Rightarrow \mathbf{int})\ \mathbf{sd} \to \tau^{\uparrow} \\
\texttt{SEG\_REDUCE} &: (\mathbf{int} \Rightarrow \tau)\ \tau\ (\tau \times \tau \Rightarrow \tau)\ \mathbf{sd} \to \tau^{\uparrow} \\
\texttt{SEG\_SCAN} &: (\mathbf{int} \Rightarrow \tau)\ \tau\ (\tau \times \tau \Rightarrow \tau)\ \mathbf{sd} \to \tau^{\uparrow} \\
\texttt{SEG\_FILTER} &: (\mathbf{int} \Rightarrow \tau)\ (\tau \Rightarrow \mathbf{bool})\ \mathbf{sd} \to \tau^{\uparrow} \times \mathbf{sd} \\
\texttt{SEG\_PARTITION} &: (\mathbf{int} \Rightarrow \tau)\ (\tau \Rightarrow \mathbf{bool})\ \mathbf{sd} \to \tau^{\uparrow} \times \mathbf{sd} \times \tau^{\uparrow} \times \mathbf{sd}
\end{aligned}
$$

Fig. 4. Segmented SOAC operators

4 Fusion

Much in the way that inlining is a key optimization for functional-language compilers, fusion is critical to performance for data-parallel language implementations — it increases the amount of computation per kernel and, most importantly, decreases the memory bandwidth required by the program. The design of λ_{cu} has been motivated by the goal of making it easy to reason about when fusion is legal and providing a uniform framework for all of the many different possible fusion operations.

Our compiler's flattening phase produces a program with many trivial kernels, each consisting of a single simple SOAC application. The fusion phase proceeds by first identifying data dependencies between kernels, then applying fusion rules to determine the set of possible kernel fusions. Using the technique of Robinson *et al.* [**?**], we pick an optimal strategy for fusing kernels, which is then realized by applying rewrite rules. We describe each of these steps in the sequel.

4.1 The PDG

A *Program-Dependence Graph* (PDG) is a graph that combines a control-flow graph for the conditional and iterative structure of a program with dataflow graphs that reflect the data-dependencies between statements in control regions [**?**]. In the CPU-level of λ_{cu}, the control regions correspond to the *blk* terms and the data dependencies are explicit in the result bindings and arguments to the **run** expressions. Computations that are not inside a kernel can be moved to the GPU by encapsulating them in a **ONCE** kernel.

4.2 Fusion rules

The core of the λ_{cu} fusion process consists of a set of rules which determines how to combine SOACs within a kernel. Our compiler supports two kinds of fusion: producers and consumers can be *vertically* fused and kernels that have the same iteration structure, but no data dependencies, can be *horizontally* fused. For a given control region in the PDG, the dataflow graph can be examined to determine the candidate fusion pairs; vertical fusion when there is a data dependency, and horizontal fusion when there is no dependency but

$$\textsc{Map-Map}\ \frac{A = \texttt{MAP}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ n \quad \oplus \quad B = \texttt{MAP}\ \{\ i \texttt{ => } e_2 \texttt{ using } ys\ \}\ n}{B = \texttt{MAP}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ using } xs \cup (ys \setminus \{A\})\ \}\ n}$$

$$\textsc{Map-Permute}\ \frac{A = \texttt{MAP}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ n \quad \oplus \quad B = \texttt{PERMUTE}\ \{\ i \texttt{ => } e_2 \texttt{ using } ys\ \}\ \Lambda\ n}{B = \texttt{PERMUTE}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ using } xs \cup (ys \setminus \{A\})\ \}\ \Lambda\ n}$$

$$\textsc{Map-Reduce}\ \frac{A = \texttt{MAP}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ n \quad \oplus \quad b = \texttt{REDUCE}\ \{\ i \texttt{ => } e_2 \texttt{ using } ys\ \}\ id_r\ \Lambda\ n}{b = \texttt{REDUCE}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ using } xs \cup (ys \setminus \{A\})\ \}\ id_r\ \Lambda\ n}$$

$$\textsc{Filter-Filter}\ \frac{\begin{array}{c} A = \texttt{FILTER}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ \{\ v \texttt{ => } p_1 \texttt{ using } ys\ \}\ n \\ \oplus \\ B = \texttt{FILTER}\ \{\ i \texttt{ => } e_2 \texttt{ using } zs\ \}\ \{\ v \texttt{ => } p_2 \texttt{ using } ws\ \}\ (\#A) \end{array}}{\begin{array}{l} B = \texttt{FILTER}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ using } xs \cup (zs \setminus \{A\})\ \} \\ \quad \{\ i \texttt{ => let } a = e_1 \texttt{ in } p_1\ \&\&\ [A\ \texttt{[}\,i\,\texttt{]}/a]p_2 \texttt{ using } ys \cup (ws \setminus \{A\})\ \}\ n \end{array}}$$

$$\textsc{Filter-Reduce}\ \frac{\begin{array}{c} A = \texttt{FILTER}\ \{\ i \texttt{ => } e_1 \texttt{ using } xs\ \}\ \{\ w \texttt{ => } p \texttt{ using } ys\ \}\ n \\ \oplus \quad b = \texttt{REDUCE}\ \{\ i \texttt{ => } e_2 \texttt{ using } zs\ \}\ id_r\ \Lambda\ (\#A) \end{array}}{b = \texttt{REDUCE}\ \{\ i \texttt{ => let } a = e_1 \texttt{ in if } [a/w]p \texttt{ then } [A\ \texttt{[}\,i\,\texttt{]}/a]e_2 \texttt{ else } id_r \texttt{ using } xs \cup (ys \cup zs) \setminus \{A\})\ \}\ id_r\ \Lambda\ n}$$

Fig. 5. Selected fusion rewriting rules

iteration structures are nonetheless compatible; the most frequent cases of horizontal fusion occur when two SOACs operate on the same input. For example, two **REDUCE** operations that take arrays of the same shape can be horizontally fused.

The following are the legal producer-consumer fusion pairs, where we use the body of a trivial kernel (*i.e.* a single SOAC) to identify its kind:

- A **MAP** whose output is consumed by any SOAC ***S*** can be fused with ***S***.
- Two **PERMUTE**s in series can be fused in a way that is similar to the fusion of two **MAP**s in series. The index transformations can be composed, and the value generator of the consuming combinator must be modified to take into account the producer's transformation of index space.
- A **FILTER** whose result is consumed by **REDUCE** can be fused by using the reduction operator's identity for those values that are filtered out. rewriting the **FILTER** as a **MAP** that replaces filtered-out values with the reducing operator's identity input; the **MAP** can then be fused into the array's thunk representation.
- Two **FILTER**s in series can be fused by forming the conjunction of their predicates.
- A **REDUCE** whose scalar result is consumed by a **ONCE** may be fused with the **ONCE**.

Two kernels that are not related by data dependency may be fused horizontally if they have compatible iteration structure, which means that the sizes (or shapes) of the vectors being operated on are the same and that the SOACs being used are compatible. The latter property always holds when the two kernels have the same SOAC, but it also holds for some other combinations, such as **MAP** and **REDUCE**.

4.3 Scheduling

Once all the candidate pairs have been identified, we need to determine which pairs of operations to fuse. It is not possible to simply perform *all* fusions, because of mutual incompatibility between some choices of fusion candidates. For example, consider the following NESL expression:

```
let total = sum(as) in { a*a / total : a in as }
```

The initial λ_{cu} translation of this code has three kernel invocations, which separately compute the sum, array of squares, and array of quotients.

The sum and array of squares are both computed by traversing `as`, so they could be computed together in a single fused kernel, producing both results at once. A second kernel would then divide each square by

the sum. Alternatively, if the sum is computed in its own kernel, the multiplication and division can both be performed in a second kernel. In both cases the sum of all elements must be computed before performing the division on any element, so there is no way to reduce the computation to a single kernel invocation.

We follow the approach of Robinson *et al.* and use an ILP solver to determine the set of fusion operations that minimized execution cost [**?**]. The objective function accounts for global memory traffic (*i.e.* for kernel arguments and results), the need for intermediate storage, along with constraints derived from the incompatibilities between alternative choices for fusing particular SOAC applications.

4.4 Rewriting

The result of solving the ILP problem is a mapping from kernels to slot indices in the execution schedule for the control region. All of the kernels in the same slot get fused into a single kernel. This operation is done in two steps: first the SOACs from the individual kernels are collected together and then rewriting rules are applied to reduce them to a minimal number of SOAC invocations. The first step requires some bookkeeping to remove redundant arguments and to make sure that all of the results used outside the fused kernel are returned as results. The second step involves merging the code of thunks.

Figure 5 presents several representative examples of the rewriting used to fuse SOAC applications (the $\oplus$ operator denotes fusion). To keep the presentation simple, we limit ourselves to the case where the incoming SOACs and thunks produce a single result. The notation "$[A\ \texttt{[}\ i\ \texttt{]}/a]e$" means substitute the variable "a" for the subscript expression "$A\ \texttt{[}\ i\ \texttt{]}$" in the expression "$e$."

5 An example of fusion

To illustrate how our fusion phase works, we consider the following small program, which is a NESL version of the example from Robinson *et al.* [**?**]. The code computes two scaled versions of its input array using different scaling factors, each of which depends on the entire array contents.

```
function norm2 (xs) : [int] -> ([int], [int]) =
  let sum1 = sum(xs);
      gts = { x : x in xs | (x > 0) };
      sum2 = sum(gts);
  in
    ({ x / sum1 : x in xs }, { x / sum2 : x in xs })
```

The flattening transform converts this NESL code into the following λ_{cu} fragment. Note that while there are five kernel invocations, only three unique kernels are present, because the original program contains only three different data-parallel operations: sum (`K1`), filtering with `(x > 0)` (`K2`), and element-wise division by a constant (`K3`).

```
kernel K1 (xs : [float]) -> float {
  let z = REDUCE { i => xs[i] using xs } (0) { (x, y) => x + y } (#xs)
  return z
}
kernel K2 (xs : [float]) -> [float] {
  let ys = FILTER { i => xs[i] using xs } { x => x > 0 } (#xs)
  return ys
}
kernel K3 (xs : [float], s : float) -> [float] {
  let zs = MAP { i => xs[i] / s using xs, s } (#xs)
  return zs
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let sum1 : float = run K1 (xs)
  let gts : [float] = run K2 (xs)
  let sum2 : float = run K1 (gts)
  let res1 : [float] = run K3 (xs, sum1)
  let res2 : [float] = run K3 (xs, sum2)
  return (res1, res2)
}
```

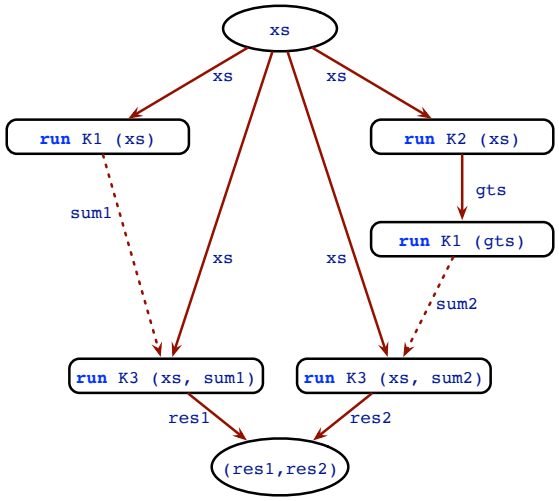

Fig. 6. The PDG for the `norm2` example.

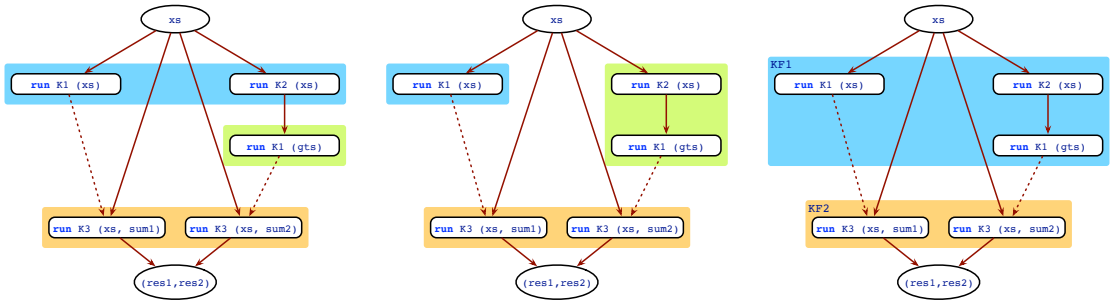

Fig. 7. Three possible schedules for the `norm2` example. The third illustrates filter-reduce fusion and is the choice of the ILP solver.

The next step is to compute a schedule for the program. The control flow of this program at the CPU level is strictly linear, *i.e.*, the kernel invocations all occur in the same single control region, so there is a single PDG for the program, which is shown in Figure 6.

The fusion rules described in Section 4.2 are used to determine the set of available fusion opportunities and their dependencies and conflicts. For this example, there are three possible ways to fuse the kernels, which are illustrated in Figure 7. Following the approach of Robinson, the dependencies and conflicts are encoded into constraints that must be satisfied while attempting to minimize the expected computational and communication cost of the fused program. This system is given to an ILP solver (**lp_solve** in our case [**?**]), which returns the right-most clustering into two kernels.

The next step is to merge kernels according to the schedule. For the example, this process produces the following λ_{cu} code, which has two kernels: `KF1`, which corresponds to the blue cluster, and `KF2`, which corresponds to the orange cluster.

```
kernel KF1 (xs : [float]) -> ([float], [float]) {
  let gts = FILTER { i => xs[i] using xs } { x => x > 0 } #(xs)
  let sum1 = REDUCE { i => xs[i] using xs } (0) { (x, y) => x + y } #(xs)
  let sum2 = REDUCE { i => gts[i] using gts } (0) { (x, y) => x + y } #(gts)
```

```
  return (sum1, sum2)
}

kernel KF2 (xs : [float], sum1 : float, sum2 : float) -> ([float], [float]) {
  let res1 = MAP { i => xs[i] / sum1 using xs, sum1 } #(xs)
  let res2 = MAP { i => xs[i] / sum2 using xs, sum2 } #(xs)
  return (res1, res2)
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let (sum1 : float, sum2 : float) = run KF1 (xs)
  let (res1 : [float], res2 : [float]) = run KF2 (xs, sum1, sum2)
  return (res1, res2)
}
```

The final step is to fuse the SOACs inside the merged kernels using the rewrite rules described in Section 4.4. For the example, this step results in the following λ_{cu} code:

```
kernel KF1 (xs : [float]) -> ([float], [float]) {
  let (sum1, sum2) =
        REDUCE { i => (xs[i], if x > 0 then xs[i] else 0) using xs } (0, 0) { (x, y) => x + y } (#xs)
  return (sum1, sum2)
}

kernel KF2 (xs : [float], sum1 : float, sum2 : float) -> ([float], [float]) {
  let (res1, res2) = MAP { i => (xs[i] / sum1, xs[i] / sum2) using xs, sum1, sum2 } (#xs)
  return (res1, res2)
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let (sum1 : float, sum2 : float) = run KF1 (xs)
  let (res1 : [float], res2 : [float]) = run KF2 (xs, sum1, sum2)
  return (res1, res2)
}
```

Note that the merging step can produce ill-formed kernels, such as `KF1`, which has two reductions in series. This situation is temporary, however, as the SOAC fusion step merges these into a single reduction.

6 Code generation

One of the goals in designing λ_{cu} is to enable easy generation of CUDA kernels while supporting the rich set of fusion optimizations described above. Each SOAC effectively defines an algorithmic skeleton [**?**] that coordinates the data access, iteration, and synchronization of the kernel. The thunks then define the computational kernels inside this iteration structure.

Consider the simple case of a kernel that computes the product of two vectors.

```
kernel K (u : [int], v : [int]) {
  let w = MAP { i => u[i] * v[i] using u, v } (# u)
  return w
}
```

This might produce the following simple CUDA kernel, where we have highlighted the computational kernel in blue and the target of the computation in green:[4]

```
__global__ void K (__global__ int *u,__global__ int *v, __global__ int *w, int n)
{
    int blockId = blockIdx.y * gridDim.x + blockIdx.x;
    int idx = blockId * blockDim.x + threadIdx.x;
    if (idx < n) {
        w[idx] = u[idx] * v[idx];
    }
}
```

The rest of the code is essentially the same, no matter what the actual thunk being mapped is.

Our code generator exploits this decomposition of kernels by defining a higher-order function for each SOAC that takes arguments for generating the computational kernels and generates the CUDA code for

[4] In practice, the code generator produces more efficient kernels that process multiple elements per thread.

the SOAC. Figure 8 gives a partial Standard ML signature for the code generator. The `gen_thunk` type is a functional representation of the code generator for a λ_{cu} thunk; it takes an environment for resolving λ_{cu} variables to CUDA expressions and a function for generating code to save the results. The `genThunk` function is used to generate code for a λ_{cu} thunk. It takes the AST representing the thunk as an argument and returns a function that can then be passed to one of the SOAC code generation functions.

```
(* representation of λcu code *)
structure LCu = struct ... end
(* representation of CUDA code *)
structure Cuda = struct ... end

signature CODEGEN = sig
(* environment for mapping λcu vars to CUDA expression *)
  type env

  type gen_thunk = env * (LCu.exp list -> Cuda.stm list) -> Cuda.stm list

(* generate code for the MAP SOAC *)
  val genMap : gen_thunk * LCu.var -> Cuda.stm list

(* generate code for the PERMUTE SOAC *)
  val genPermute : gen_thunk * gen_thunk * LCu.var -> Cuda.stm list

(* generate code for the REDUCE SOAC *)
  val genReduce : gen_thunk * gen_thunk * LCu.var -> Cuda.stm list

  ...

(* translate a λcu thunk to CUDA *)
  val genThunk : LCu.thunk -> gen_thunk
end
```

Fig. 8. CUDA code generation interface

There are some additional details that we must address when generating the code for running a kernel, one of which is the need to allocate storage for the result array on the GPU. We pass the address of this storage to the kernel as an extra argument (*e.g.*, the `w` parameter in the above example).

7 Related work

The use of SOACs is common in languages and compiler representations for data-parallel programming. Examples include Nova [**?**], Futhark [**?**], and Delite [**?**]. Out combinators are different from these in that we abstract over the index space, instead of the value space, and that we include a pull thunk that allows us to fuse map operations into the data-fetch part of other operations. Delite also includes map operations in some of its combinators (which they call parallel patterns), but their approach is more *ad hoc* and does not support segmented operations. The data-fetch operations are reminiscent of the pull arrays used in some embedded DSLs for array processing [**?**].

8 Future work

We are planning a number of future improvements to our compiler, which we believe will be easily supported by λ_{cu}. One of these is *vectorization avoidance*, which is a pre-flattening analysis that identifies sequential expressions that appear in parallel apply-for-each contexts. These expressions are encapsulated as lambda abstractions, which are *not* transformed by the flattening phase [**?**]. This technique reduces the need for producer-consumer fusion, but it also allows the compiler to avoid encoding conditionals using **PARTITION**.

A second improvement is the use of uniqueness types [**?**] at the CPU-level of λ_{cu} to analyze memory and optimize memory usage. To fully take advantage of this information, we also need to support update-in-place in the GPU-level of λ_{cu}. Supporting mutation will also improve the implementation of some of the permutation operators produced by flattening and will enable more kinds of horizontal fusion.

Acknowledgments

The original NESL/GPU implementation was primarily the work of Lars Bergstrom, who has also provided helpful advice about CUDA hacking. Portions of this research were supported by the National Science Foundation under award 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. We also thank the NVIDIA Corporation for their generous donation of both hardware and financial support.