Abstract. GPUs provide more raw processing power than has been accessible in the past, but making effective use of this processing power is a forbidding programming task. With high memory bandwidth and vast capacity for arithmetic operations, GPUs are well-equipped for executing data-parallel programs, but the programming patterns required to hide memory latency impose strict requirements on the design of GPU programs, and using the low-level hardware programming interfaces remains very difficult. The Nessie compiler facilitates programming for data-parallel GPGPU applications by implementing a translation from NESL—a high-level, strict, functional programming language—to efficient low-level C++ code invoking nVidia CUDA kernels on the GPU. The compiler achieves efficiency in generated code through optimizations on a specialized intermediate representation, $\lambda_{cu}$, which constitutes the primary contribution of this work. $\lambda_{cu}$ represents arrays as the application of per-element generator functions to index spaces, separates serial and parallel portions of input programs, and makes explicit a set of building blocks for parallel array operations. The most significant optimization pass in the Nessie compiler fuses together parallel array operations. Alternative choices of which operations to fuse are evaluated by constructing integer linear programming instances using a heuristic model of performance benefit associated with each optimization opportunity, constrained by the compatibility relation between different array operations. This approach allows Nessie to avoid exhaustively searching the space of potential optimized programs, which is combinatorially large due to incompatibilities between fusion choices.

1 Introduction

1.1 Motivation

Serial execution of instructions is the most obvious way to run programs, but trends in computing hardware increasingly betray the naiveté of this approach [OH05]. In order to minimize the time taken to execute programs, parallel execution is desirable: different computing resources should work on evaluating independent subcomputations at the same time whenever possible. Outside of algorithmic changes, the identification and exploitation of parallelism is now one of the most important avenues for improving software performance.

A program can be decomposed into parallel subprograms by identifying subprograms that may run independently (task parallelism) or subsets of input data that may be processed independently (data parallelism); these two branches of parallelism can also be employed together. The number of independent tasks present in a program bounds the speedup achievable through task parallelism but the ideal speedup obtained by data parallelism scales with the size of input data. While the typical program continues to increase in size and complexity, trends in program sizes are much flatter than trends in the size of the data they process.
As such, this work focuses on strategies for data parallelism, which will only become more relevant to program performance as data sizes increase.

The modern computer hardware ecosystem includes a wide variety of substrates for computation. Among these, graphics processing units (GPUs) offer the best potential throughput for executing data-parallel code. Modern GPU hardware was developed by adding flexibility to a “fixed-function” pipeline which was originally designed to perform a number of 3D transforms and polygon rasterization operations for 3D graphics, taking advantage of the data-parallelism inherent in those tasks. Current GPU hardware is capable of performing arbitrary computations, with many independent data streams being transformed according to the same set of instructions.

Nonetheless, GPUs cannot achieve high performance with a naïve translation of existing programming languages into their machine code. A programmer wishing to achieve maximum GPU throughput must be careful to avoid many forms of conditional branches, obey strict disciplines of data layout and access, and carefully manage synchronization and data movement between the host CPU and the GPU. These limitations on program structure can be met by disciplined programmers writing programs in low-level languages where the language semantics mirror those of GPU hardware architectures, but only at the cost of memory safety (dealing with pointers and manual memory management), code reuse (no affordance for generic programming), and static analysis (weak, assembler-like type systems).

Instead of writing low-level GPU languages by hand, it makes sense to write a compiler to transform high-level programs into low-level languages for GPU execution. As long as the semantics of the high-level program are compatible with an efficient translation, it is possible to construct an equivalent, fast, low-level program, without entangling the programmer in low-level, architecture-specific hardware concerns.

The NESL language is one such high-level language and this work presents a translation from NESL programs into CUDA, a low-level GPU programming model.

1.2 Background

There is a long history of research into compiling data parallelism. Because programs are more frequently written in languages with serial semantics and scalar variables, much effort has been devoted to automatic parallelization, which seeks to automatically identify opportunities for task- or data-parallel execution lying undiscovered in programs. However, this is an intractable analysis, and even after opportunities for parallelism have been identified, there still remains the challenge of producing efficient code for highly parallel processors such as GPUs.

Past research into compiling for parallel processors focused on the hardware available at the time, which was typically organized in single instruction multiple data (SIMD) fashion, with hardware vector instructions which act on bounded- or fixed-size vectors. Translating loops in scalar programs into vector operations is the domain of autovectorization, which enables significant data parallelism, but is suited to CPU architectures which feature wide registers, and analyses for autovectorization can be both fragile and costly. With the advent
of GPUs, the *single instruction multiple thread* (SIMT) paradigm has become more common. This model differs from SIMD by featuring a large number of processors operating on data items of small width (as opposed to wide vector registers) and which share an instruction pointer and control logic, but have their own sets of registers and cache memory. In both cases, data parallelism is achieved by performing regular operations on homogeneous data vectors. Unfortunately, storing all data in homogeneous vectors is not an obvious match for NESL’s model of computation, where parallel operations can be nested arbitrarily and vectors may contain rich variety of data types including other vectors. These composable parallel constructs such as arrays-of-arrays, list comprehensions, sums of sums, etc., allow representing nested loops as data-parallel operations, improving expressiveness while permitting very flexible scheduling of subcomputations. Being able to express multiple levels of parallelism in this way is known as *nested data parallelism* (NDP).

### 1.3 The flattening transform

Given the ease of specifying parallel algorithms with NDP, it is desirable to be able to map this mode of specification onto the available parallel hardware so that algorithms written in this style can be executed quickly. The *flattening transform* maps NDP onto flat data parallelism, enabling execution of high-level code on GPUs and vector processors. Introduced by Blelloch [Ble90], the flattening transform rewrites code operating on nested data structures into code operating on flat arrays along with *segment descriptors* which track the nesting structure of the data. Flattening is necessary not merely because of the need to express computations in terms of the primitives presented by low-level languages like CUDA which serve as compilation targets, but also because GPU architectures exhibit high memory latency, making pointer-based implementations of nested arrays undesirable. Although GPUs employ “barrel”-style execution which masks memory latency by pipelining accesses from different threads, serialization of accesses often occurs, impacting throughput.

Via flattening, programs become significantly more amenable to GPU execution, since bulk data in nested structures may then be processed without additional indirections.

Low-level general-purpose programming on GPUs is supported by major GPU vendors through two libraries, CUDA and OpenCL, which offer similar semantics but target different sets of underlying hardware. These libraries provide a language with pointers, arrays, and mutable updates; programmers specify a single piece of code which is executed in lock-step in many different processing elements on the GPU. The programmer must manually move data between host and GPU memory, ensure that all array accesses are in bounds, avoid races in reading and writing data, and manually map high-level specifications of algorithms down to the datatypes and operations available from GPU programming libraries.

### 1.4 A language for NDP

The NESL language is a full-featured strict, nested data-parallel, array-oriented functional programming language [Ble95]. The language provides a number of parallel primitives, the most important of which is a parallel “apply to each” operation on sequences of values of the
same type. By nesting “apply to each” and other parallel sequence operations such as sums, partitioning, and permutations, NESL can express complex data-parallel computations.

The language specifies the asymptotic running times of these parallel primitives in terms of both *work* and *depth*. The notion of work measures the runtime a computation would have when executed on a single processor, which is equivalent to the sum of all time spent by each processor in a multiprocessor system running the program in parallel, neglecting interprocessor communication costs. The depth of a computation measures the minimum time the program would take to execute if run on a machine with an infinite number of processors; it captures the inherently sequential behaviors of programs, which cannot be sped up by further parallelization of execution. By comparing work and depth for a program, one can see (asymptotically) how much of the program’s workload is parallelizable [Ble96].

NESL was designed by Guy Blelloch in the early 1990s and originally implemented for a number of supercomputers in terms of translation to VCODE [BCH+94], a stack-based, array-oriented intermediate language executed by an interpreter that dispatched each VCODE operation to an optimized hand-written parallel implementation. [BC90] The vast majority of computation in data-parallel programs occurs in vector operations, so interpretive overhead did not impose an unacceptable performance cost. However, this design is no longer appropriate because of the growing disparity between memory latency and the speed of arithmetic operations; for a GPU implementation of NESL, synchronous communication between the CPU and GPU between every array operation would dominate the time consumption of program execution.

For simplicity of explanation, this work considers a demonstrative toy version of NESL called μNESL, which retains the essential attributes of NESL while being smaller and admitting more concise specifications of the transformations involved in running the language on the GPU. In particular, μNESL omits a number of arithmetic and logical operations as well as some syntactic sugar available in the full NESL, since these do not require any additional considerations for translation. The salient features of NESL which cannot be omitted are the core calculus along with a representative subset of the built-in parallel sequence operations.

Specifically, μNESL augments a typed lambda calculus with primitive machine types `float`, `int`, and `bool`, nestable sequences, and a set of parallel sequence operations:
<table>
<thead>
<tr>
<th>Basic sequence operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dist(val, len)</code></td>
</tr>
<tr>
<td><code>elt(seq, idx)</code></td>
</tr>
<tr>
<td><code>replace(seq, val, idx)</code></td>
</tr>
<tr>
<td><code>count(seq)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence pair operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>zip(seq1, seq2)</code></td>
</tr>
<tr>
<td><code>unzip(seq)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scans</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{plus, min, max, or, and}.scan(seq)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unfolds</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iseq(start, inc, end)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{sum(seq), min, max, any, all, max_index, min_index}(seq)</code></td>
</tr>
<tr>
<td>Reordering</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>seq -&gt; idxs</td>
</tr>
<tr>
<td>permute(vals, idxs)</td>
</tr>
<tr>
<td>seq &lt;- ivpairs</td>
</tr>
<tr>
<td>rotate(seq, by)</td>
</tr>
</tbody>
</table>

get values from a sequence at a sequence of indices
reorder a sequence by a sequence of new indices
overwrite elements in a sequence at a sequence of (index, new value) pairs
create a sequence by rotating its elements by an offset
reverse a sequence
given a sequence of (value, flag) pairs, produce a sequence of the values for which the flag is true
concatenate two sequences
extract the half-open subsequence \([\text{start}, \text{end})\)
create a sequence by alternatively taking an element from each sequence until reaching the end of one
compute the sequence of lengths between true entries in a boolean sequence
split a sequence into a sequence of disjoint sequences
given a sequence of lengths for the subsequences
concatenate the inner sequences of a sequence into a single flat sequence
split a sequence into a pair of sequences containing elements whose corresponding flags are uniform
splits a sequence into a pair of sequences containing the elements less and greater than its median respectively
sort the contents of a sequence
compute the sorted positions of the elements of a sequence
create a sequence of \((key, value sequence)\) pairs from a sequence of \((key, value)\) pairs
given a regular (rectangular) nested sequence, produce a nested sequence with swapped inner and outer dimension
2 Nessie

Nessie is a compiler for NESL that targets the CUDA API for nVidia GPUs. It leverages a novel high-level intermediate representation to reason about fusion of array operations, and produces as output CUDA C++ which is lowered to machine code and specialized to the GPU’s instruction set by nVidia’s nvcc compiler. Nessie’s IR and the optimization passes in its compilation process have been designed to take advantage of the flexibility of functional specifications of program behavior and map them onto efficient usage of GPU hardware resources.

3 Efficient GPU execution

Making efficient use of the computational resources on GPUs presents a unique set of challenges compared to targeting other common processors. In addition to the obvious difference in core count, GPU architectures share instruction fetch/dispatch logic between large numbers of individual processing elements, each with individual register sets; this is known as “single-instruction multiple thread” (SIMT) organization. Whereas CPUs rely on techniques such as branch prediction, prefetching, reordering of memory accesses, and out-of-order execution of instructions to minimize the effects of memory latency, GPU architectures keep a queue of waiting threads and when performing a memory access for one thread switch in another which is not blocked on memory.

3.1 The CUDA model

The CUDA API presents a hierarchical model of the architecture of nVidia GPUs: Programers use the C++ language to write kernel functions, denoted by the __global__ declaration specifier, which perform scalar or small-width SIMD computations, and access large vectors through pointers. Kernels are executed on the GPU by multiple threads operating on different offsets in the input data vector. Threads are executed in lockstep in groups of 32 called warps, which share an instruction pointer. A variable number of warps can be executed on a single Streaming Multiprocessor (SM), depending on how much memory is required for each thread. Code outside of kernels is executed on the CPU as usual, and invokes kernels via special triple-angle-bracket function call syntax which specifies resource allocation for the function including threads and memory [Wil13].

\[ \text{func} \lll \text{gridDim}, \text{blockDim}, \text{sharedMemBytes}, \text{execStream}\rrr (\text{args}) \]

Fig. 1. CUDA kernel invocation syntax. gridDim and blockDim are 3-vectors specifying a grid of blocks and the size, measured in threads, of the blocks within the grid. sharedMemBytes specifies extra shared memory to allocate, and execStream specifies which CPU-GPU communication queue to use.
3.2 Memory regions and hierarchy

The CUDA model presents several memories of differing capacities, levels of sharing, and access latencies.

The fastest of these is employed as registers for kernel execution. Each SM contains on the order of hundreds of kilobytes of register memory which is portioned out to threads based on register requirements of the kernels being executed.

Each SM is also assigned local memory, which is generally smaller in capacity than register memory, exhibits higher access latencies, and is employed to hold thread-specific values spilled from registers or which require pointer semantics due to indirect addressing.

Similar in performance but shared across all threads in a block, shared memory can be used for cross-thread communication or to parallelize loads from global memory by collaboration from multiple threads.

Global memory is shared across the entire GPU, but has much higher access latency than shared memory or registers.

Texture memory allows cached and read-only sampling of floating-point or integer data in one, two, or three dimensions using floating-point coordinates. Samples of floating-point data may be linearly interpolated, and out-of-bounds accesses may be configured in a number of behaviors. Nessie does not presently use texture memory because it would only provide a minor speedup for a small subset of workloads for which performance is limited by floating-point arithmetic.

Programs explicitly reference the different memory regions exposed by the CUDA programming model.

3.3 Thread, block, and kernel synchronization

CUDA exposes synchronization and communication primitives at block, kernel, and GPU levels of granularity. Within a warp, all threads are inherently synchronized, as they run in lockstep. Communication within a warp may be achieved through “shuffle” and “vote” intrinsic functions: shuffles permute data across the instances of a variable in different threads, and votes summarize the values of a boolean variable across all threads in a warp.

Synchronization between threads in a block is performed by the __syncthreads intrinsic function, which pauses execution of a warp until a quorum of threads across all blocks in a kernel reach the same call to __syncthreads. Variants of this intrinsic also permit limited intra-block communication, returning the total number of nonzero values passed by all threads. Communication within a block may also be achieved through volatile accesses to shared memory.

Synchronization between blocks in a kernel can be achieved most straightforwardly by introducing kernel boundaries and serializing execution of kernels. It is also possible, following [XcF10], to synchronize between blocks in a kernel by leveraging memory limits to prevent sharing of SMs by multiple blocks and fixing the count of blocks per kernel to the number of SMs. This allows leveraging atomic operations to synchronize execution of blocks in the same kernel without requiring CPU-GPU communication. Alternatively,
it is possible to synchronize blocks without atomic operations by using thread-level synchronization to synchronize a group of threads which each monitors a locations in global memory signalling that the corresponding blocks has entered the critical section.

Because of the implementation complexity and inter-block concurrency constraints imposed by GPU-based block synchronization techniques, Nessie relies on simple kernel-based synchronization via `cudaThreadSynchronize`.

### 3.4 Memory banking and access patterns

While texture and constant memory are read-only from kernels, and benefit from aggressive caching, accesses to global and shared memory have more complex performance attributes.

Shared memory is optimized for two read patterns: strided access, where each thread in a warp loads a 4-byte word from an address whose lower bits match those of the thread index, and broadcast access, where all threads in a warp request the same word.

On hardware which supports nVidia’s Compute Capability 3.0, reads from global memory are cached, but different layers of the memory hierarchy continue to differ in coherency, and programmers must be aware of how caching alters the performance model.

### 4 Mitigating GPU performance hazards

Given the architecture described above, there are several rules of thumb that guide the optimizations implemented in Nessie. All of them are related to avoiding particular performance pitfalls; in rough order of performance impact, the fastest programs will minimize:

1. **Intermediate array usage** – every array manifest in the execution of a CUDA program must first be allocated, then written into, and subsequently read from. Each of these incurs significant costs, and rewriting computations with temporary array variables into computations which have fewer temporaries eliminates all three costs at once. In Nessie, this is achieved through fusion of vector computations.

2. **Bank conflicts** – When reading data in global memory, multiple concurrent reads of the same address will incur access serialization. This can be avoided by duplicating the data, changing the timing or ordering of accesses, or via higher-level program transformations, which may find a way to avoid the read entirely. Nessie’s implementations of parallel primitives implement efficient patterns for distributing data: permutations are required to be 1:1, and scans and reductions generate CUDA code which avoids bank conflicts. Data required by multiple threads is broadcasted, while kernels writing to memory are ensured to use a stride that avoids contention.

3. **Memory traffic** – despite the GPU’s high memory bandwidth, there are many cases where memory latency cannot be hidden completely. Computations that can be rewritten to read less data, *e.g.* by using smaller scalar datatypes, arrays of smaller dimensions, or algorithms that compute data from locally-available predecessors rather than storing it globally, will put less pressure on available memory bandwidth, and will not require as much parallel work to hide memory latency. Nessie’s fusion system reasons about
memory traffic as an incentive to fuse operations, and successful fusion of multiple
vector operations with the same input will combine their reads. Fusion that combines a
producer and consumer avoids both reads and writes, and is doubly beneficial.

4. Synchronization and divergence – Within a warp, threads which do not take branches sit
idle until any threads which took the branch complete it; this wastes potential to perform
useful computation. Idleness at larger scales can result from synchronizations between
warps or across the GPU (see §3.3). Nessie performs synchronization necessary for the
correctness of reductions and scans, but fusion can combine multiple scans or multiple
reductions into a single one, reducing total synchronization. All optimizations which
reduce the number of kernels invoked in a program also decrease the number of GPU-
wide synchronization events, because kernels are executed serially.

5 The \( \lambda_{cu} \) IR

The \( \lambda_{cu} \) IR is designed to capture the relevant constraints of efficient GPU execution and
enable optimizations within those constraints. To achieve this, \( \lambda_{cu} \) syntactically separates
(1) scalar-producing computations which are run in data-parallel fashion from (2) the ker-
nels which control their iteration spaces and (3) the serial code that orchestrates overall
program structure and the execution of kernels. Each kernel has a given iteration struc-
ture determined by the list of array combinators it applies. These combinators are called
Second-Order Array Combinators (SOACs) because they accept first-order thunks (which
compute scalar elements of array contents) as parameters, and apply them to an index se-
quence corresponding to a particular array access pattern. The distinction between SOACs
and thunks separates the description of a traversal from that of array element values. Sep-
arating the specification of array destinations from the code that computes scalars avoids
implicit mixing of the flow of index transformations with the flow of array contents, sim-
plifying high-level optimizations on \( \lambda_{cu} \).

5.1 The \( \lambda_{cu} \) representation

A program is represented in \( \lambda_{cu} \) by a list of kernel declarations followed by function defi-
nitions, one of which is designated as the program entrypoint. Function bodies may refer to
any kernel or any function in the program.

\[
\begin{align*}
\text{prog} & ::= \text{kern}_1 \ldots \text{kern}_n \text{funcs} \quad \text{bind } \text{binders}(\text{kern}_1 \ldots \text{kern}_n) \text{ in } \text{funcs} \\
\text{_funcs} & ::= \text{func}_1 \ldots \text{func}_n \quad \text{bind } \text{fbinders}(\text{funcs}) \text{ in } \text{funcs} \\
\text{binders} & = \text{fbinders}(\text{func}_1 \ldots \text{func}_n)
\end{align*}
\]
5.2 Kernels

Kernel declarations define routines that run on the GPU; kernels may transform any number of scalar or array values into new scalars and arrays.

\[
\text{kern} ::= \text{kernel} \ k \ (x_1, \ldots, x_n) \ \{ kexp \} \\
\text{binders} = k \\
\text{bind} \ x_1 \ldots x_n \ \text{in} \ kexp
\]

The body of a kernel performs parallel array operations by applying a series of SOACs, then returns a specified subset of the computed values.

Note that kernel bodies cannot run kernels or call CPU-side functions.

\[
kexp ::= \text{let} \ x_1, \ldots, x_n = \text{soac arg}_i \ kexp \\
| \ \text{ret} \ x_i \\
\text{bind} \ x_1 \ldots x_n \ \text{in} \ kexp
\]

5.3 Second-order array combinators (SOACs)

Each SOAC specifies a different pattern of data dependencies and parallelism within an array operation:

\[
\text{soac} ::= \text{ONCE} \quad \text{perform a scalar operation} \\
| \ \text{MAP} \quad \text{apply a computation independently to each element of an array} \\
| \ \text{PERMUTE} \quad \text{compute a value and a destination index to write it to} \\
| \ \text{REDUCE} \quad \text{summarize all values in an array with a given operator} \\
| \ \text{SCAN} \quad \text{generate an array of partial reductions} \\
| \ \text{FILTER} \quad \text{produce an output only at elements where a predicate holds} \\
| \ \text{PARTITION} \quad \text{produce two arrays within which a predicate holds/fails}
\]

Each SOAC accepts a particular number of arguments, which are array thunks (Λ), followed by a tuple of reduction operators (rop) for those which merge thunk results. The form of valid applications for each SOAC is given precisely in §6.2.

A tuple of reduction operators may be thought of as a predefined thunk with no captures that computes the monoid operation of a \text{SCAN} or \text{REDUCE}, and also carries information about the monoid’s identity element. A single reduction operator, such as \text{iadd}, has the identity element 0, and forms a monoid over integers via addition. A tuple such as (\text{iadd}, \text{fmul}) has the identity element (0, 1.0) and its monoid operation combines inputs \((a_1, b_1)\) and \((a_2, b_2)\) into \((a_1 + a_2, b_1 \times b_2)\).

The \text{id} reduction operator is unique in that it does not form a monoid operating on scalar values; instead it is handled specially by \text{REDUCE} and \text{SCAN} and indicates to return an array as a whole rather than performing a reduction on it.

\[
\text{arg} ::= \Lambda
\]
Thunks express the per-element work performed across flattened arrays. Each thunk maps from some index space to a scalar output expression, which may be a tuple.

Within a scalar expression, thunks can reference literal scalar values, combine their parameters with arithmetic primitive operations, bind variables, perform conditional dispatch, read the thread index, and index into captured array variables. When applied by a SOAC, thunks are executed zero or more times, with each execution operating on one point in the SOAC’s index space. The communication required by the SOAC determines the parallelism of thunk evaluation, because thunk bodies are pure and therefore free of antidependencies. SOACs express the communication and iteration context of a parallel operation, while thunks specify per-element computation. SOACs which produce array results evaluate their thunk arguments multiple times, passing different indices and storing the results at the corresponding locations in output arrays. Array-producing SOACs deconstruct tuples returned by thunks and write their primitive components into separate homogeneous arrays.

A thunk explicitly lists its captured variables after the `using` keyword; this syntactic convention is useful to simplify the presentation of thunk fusion in §9.

\[ \Lambda ::= \{ x_1, \ldots, x_n \Rightarrow gexp \textbf{ using } x_{s1}, \ldots, x_{sm} \} \]

\[ \text{bind } x_1 \ldots x_n \text{ in } gexp \]

\[ gatm ::= x | b \]

\[ gexp ::= \text{let } x_1, \ldots, x_n = gexp_1 \text{ in } gexp_2 \]
\[ \mid \text{prim } x_1 \ldots x_n \]
\[ \mid \text{if } x \ gexp_1 \ gexp_2 \]
\[ \mid \text{ret } x_1 \ldots x_n \]
\[ \mid x[s][x] \]
\[ \mid gatm \]

5.5 CPU expressions

The coarse-scale structure of a $\lambda_{cu}$ program is a list of definitions of functions which run on the CPU and do not themselves operate on data in parallel. Functions on the CPU can perform a similar set of scalar operations to per-thread code in thunk bodies, but importantly CPU-side code can run kernels, make recursive function calls, and invoke input/output commands.
\[ func ::= function f (x_1, \ldots, x_n) \{ cexp \} \text{ func} \]
\[ catm ::= x \mid b \mid bs \]
\[ cexp ::= \text{ let } (x_1, \ldots, x_n) = cexp_1 \text{ in } cexp_2 \]
\[ \mid \text{ if } catm \text{ then } cexp_1 \text{ else } cexp_2 \]
\[ \mid f (catm_1, \ldots, catm_n) \]
\[ \mid \text{ prim } catm_1 \ldots catm_n \]
\[ \mid (catm_1, \ldots, catm_n) \]
\[ \mid \text{ run } k (catm_1, \ldots, catm_n) \]
\[ \mid \text{ cmd } (catm_1, \ldots, catm_n) \]

6 Semantics of the IR

6.1 Values

The base types of \( \lambda_{cu} \) are booleans, ASCII characters, 32-bit integers, and 32-bit IEEE floating-point numbers; for each of these, there also exists a corresponding array type:

\[ b ::= \text{ bool } \mid \text{ char } \mid \text{ int } \mid \text{ float } \]

\[ bs ::= [\text{ bool}_1, \ldots, \text{ bool}_n] \]
\[ \mid [\text{ char}_1, \ldots, \text{ char}_n] \]
\[ \mid [\text{ int}_1, \ldots, \text{ int}_n] \]
\[ \mid [\text{ float}_1, \ldots, \text{ float}_n] \]

The notation \([expr \mid 0 \leq i < n]\) denotes the array value whose element sequence is given by \([i := 0]expr, \ldots, [i := n - 1]expr\).

Scalar values are tuples \((b_1, \ldots, b_n)\) of base values:

\[ \tau ::= b^* \]

The tuple with no elements is denoted () and 1-tuples are identified with their single element.

At the CPU level, both base values and arrays of base values are present; the metavariable \( a \) denotes values which may be either: \( a ::= b \mid bs \). The most general class of values in \( \lambda_{cu} \) consists of tuples which may contain both base values and array values \((a_1, \ldots, a_n)\):

\[ \gamma ::= a^* \]
6.2 Evaluation

First, the \( \rightarrow_p \) relation specifies how top-level function and kernel definitions are collected from the top-level scope of a \( \lambda_{cu} \) program into an environment \( F \).

\[
F; \text{prog} \rightarrow_p F'; \text{prog}' \cup \epsilon
\]

\( F; \text{kern prog} \rightarrow_p F \cup \{ \text{kern} \}; \text{prog} \)

\( F; \text{func}_1 \ldots \text{func}_n \rightarrow_p F \cup \{ \text{func}_1, \ldots, \text{func}_n \}; \epsilon \)

The \( \downarrow \) relation specifies that \( \lambda_{cu} \) programs evaluate to final values by preparing an environment with \( \rightarrow_p \) and then evaluating the \text{main} function:

\[
\emptyset; \text{prog} \rightarrow^* \emptyset; F; \epsilon
\]

\[
\text{function main} \; () \{ \text{cexp} \} \in F \quad \emptyset; \text{cexp} \rightarrow^* \emptyset; \bar{\nabla}; v
\]

\[
\text{prog} \downarrow v
\]

The \( F \vdash \rightarrow_c \) relation is a small-step semantic relation capturing CPU-level reduction, which involves pairs of CPU-level \( \lambda_{cu} \) expressions and value environments \( \bar{V} = \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \} \) which are finite maps from variables to values. Lookup of the variable \( x \) in a stack of variable environments \( \bar{V} \) is denoted \( \bar{V}(x) \), and is defined to return the value to which the variable is bound in the leftmost environment to contain an entry for that variable.

Given a function environment \( F \), \( \rightarrow_c \) maps a pair of a stack of variable environments \( \bar{V} = \bar{V}_1, \bar{V}_2, \ldots \) and an expression to a possibly changed stack of variable environments and a different expression.

\[
F \vdash \bar{V}; \text{cexp} \rightarrow_c \bar{V}'; \text{cexp}'
\]

\[
F \vdash \bar{V}; \text{cexp}_{\text{rhs}} \rightarrow_c \bar{V}', \bar{V}; \text{cexp}'_{\text{rhs}}
\]

\[
F \vdash \bar{V}; \text{let} \; (x_1, \ldots, x_n) = \text{cexp}_{\text{rhs}} \; \text{in} \; \text{cexp}_{\text{body}} \rightarrow_c \bar{V}'; \text{let} \; (x_1, \ldots, x_n) = \text{cexp}'_{\text{rhs}} \; \text{in} \; \text{cexp}_{\text{body}}
\]

\[
\bar{V} = \bar{V}_1, \ldots, \bar{V}_m \quad F \vdash \bar{V}; \text{cexp}_{\text{rhs}} \rightarrow_c \bar{V}'(v_1, \ldots, v_n)
\]

\[
F \vdash \bar{V}; \text{let} \; (x_1, \ldots, x_n) = \text{cexp}_{\text{rhs}} \; \text{in} \; \text{cexp}_{\text{body}} \rightarrow_c \bar{V}_1 \cup \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \}, \ldots, \bar{V}_m; \text{cexp}_{\text{body}}
\]
\[ \forall; \text{catm}_{\text{cond}} \rightarrow a \text{ true} \]

\[ \forall; \text{if catm}_{\text{cond}} \text{ then } cexp_t \text{ else } cexp_e \rightarrow_c \forall; cexp_t \]

\[ \forall; \text{catm}_{\text{cond}} \rightarrow a \text{ false} \]

\[ \forall; \text{if catm}_{\text{cond}} \text{ then } cexp_t \text{ else } cexp_e \rightarrow_c \forall; cexp_e \]

\[ \text{function } f(x_1, ..., x_n) \{ cexp_{\text{body}} \} \in F \]
\[ \forall; \text{catm}_1 \rightarrow a v_1 \ldots \forall; \text{catm}_n \rightarrow a v_n \]

\[ \forall; f(\text{catm}_1, ..., \text{catm}_n) \rightarrow_c \forall; [x_1, ..., x_n := v_1, ..., v_n] \text{cexp}_{\text{body}} \]

\[ \text{primitive rules} \vdash \text{prim}_p v_1, ..., v_n = v_{\text{out}} \]
\[ \forall; \text{catm}_1 \rightarrow a v_1 \ldots \forall; \text{catm}_n \rightarrow a v_n \]

\[ \text{CPU-PRIM} \]

For brevity, the \text{primitive rules}, which specify evaluation of each basic arithmetic operations such as addition (\text{prim}_+), subtraction, and so on, are omitted as they are uninteresting from the perspective of language semantics and correspond directly to operators built into the C++ language.

\[ \forall; \text{catm}_1 \rightarrow a v_1 \ldots \forall; \text{catm}_n \rightarrow a v_n \]
\[ \forall; \text{run } k(\text{catm}_1, ..., \text{catm}_n) \rightarrow_c \emptyset; (v_1, ..., v_n) \]

\[ \text{CPU-TUPLE} \]

\[ \forall; x \rightarrow a v \]
\[ \forall; v \rightarrow a v \]

\[ \text{CPU-ATM-VAL} \]

\[ \text{Kernel semantics:} \]
\[ \forall; \text{soac}_{\text{app}} \downarrow_S (v_1, ..., v_n) \]
\[ \forall; \text{let } a_1, ..., a_n = \text{soac}_{\text{app}} kexp_2 \rightarrow_k \forall \cup \{ a_1 \mapsto v_1, ..., a_n \mapsto v_n \}; kexp_2 \]

\[ \{ x_1 \mapsto v_1, ..., x_m \mapsto v_m \} \subset \forall \]
\[ \forall; \text{ret } a_1, ..., a_n \rightarrow_k \forall; (v_1, ..., v_n) \]

\[ \text{KERN-LET} \]

\[ \text{KERN-RET} \]
Semantics of SOAC applications:

Each SOAC evaluates the thunks it is passed at the index arguments necessary to compute its output. The $\text{IDX-THUNK}$ rule describes the evaluation of a single thunk at a given index argument $i$, in the context of an environment $\mathbb{V}$:

$$\mathbb{V};[x := i]gexp \rightarrow^*_g \mathbb{V}'; \text{ret } x_1, \ldots, x_n \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \subset \mathbb{V}'$$

$\text{IDX-THUNK}$

$\text{ONCE}$ executes its thunk argument a single time. In the implementation, instead of passing a single argument with the value zero, thunks for $\text{ONCE}$ are defined with no arguments, but considering all to take a single index argument simplifies the presentation of semantics. $\text{ONCE}$ evaluates to a tuple of base values.

$$\mathbb{V};A(0) \downarrow_t (v_1, \ldots, v_n) \quad \text{SOAC-ONCE}$$

$\text{MAP}$ applies its thunk to each index in its index space, computing a tuple of arrays.

$$\forall 0 \leq i < s(\mathbb{V};A(i) \downarrow_t (v^1_i, \ldots, v^n_i)) \quad \forall 0 \leq i < s(v^k_s \equiv [v^i_k \mid 0 \leq i < s]) \quad \text{SOAC-MAP}$$

$\text{PERMUTE}$ applies its thunk to each index in its index space, but unlike $\text{MAP}$ also evaluates its destination index thunk to compute the position to store the computed base values in its output arrays.

$$\forall 0 \leq i < s(\mathbb{V};A_1(i) \downarrow_t (v^1_i, \ldots, v^n_i)) \quad \forall 0 \leq i < s(\mathbb{V};A_2(i) \downarrow_t (idx^i)) \quad \forall 0 \leq i < s(v^idx_s \equiv [v^{idx^i}_k \mid 0 \leq i < s]) \quad \text{SOAC-PERMUTE}$$

$\text{REDUCE}$ performs a parallel reduction using its reduction-operator arguments over the arrays generated by applying its thunk argument to its index space. It produces a tuple of arrays and base values, where each element in the tuple is computed by folding (denoted $\mathcal{R}$).
and used like summation) the reduction operator in corresponding position over the array, or simply computing the array itself for the id reduction operator.

\[(\forall 0 \leq i < s)(V;A_1(i) \downarrow t (v_1^i, ..., v_n^i)) \quad vs_k \equiv \begin{cases} [v_k^i] & 0 \leq i < s \\ \mathcal{R}_k v_k^i & \text{otherwise} \end{cases} \text{ if } rop_k = \text{id} \]

SOAC-REDUCE

**SCAN** produces the partial results of applying its reduction operators to prefixes of increasing length across its input arrays. Like with REDUCE, the id reduction operator returns an array as-is rather than computing the sequence of outputs of the reduction operator over its prefixes.

\[(\forall 0 \leq i < s)(V;A_1(i) \downarrow t (v_1^i, ..., v_n^i)) \quad vs_k \equiv \begin{cases} [v_k^i] & 0 \leq i < s \\ \mathcal{R}_k v_k^i & 0 \leq j < s \end{cases} \text{ if } rop_k = \text{id} \]

SOAC-SCAN

**FILTER** acts to remove elements not fulfilling a predicate from the arrays it generates.

\[(\forall 0 \leq i < s)(V;A_1(i) \downarrow t (v_1^i, ..., v_n^i)) \quad u_i \equiv \sum_{j=0}^{p_j} \quad a_i \equiv \min_{u_j = i} j \quad vs_k \equiv [v_k^{o_i}] \quad 0 \leq i < s \]

SOAC-FILTER

**PARTITION** computes both the arrays which would be produced by **FILTER** on the same arguments and the arrays which **FILTER** would produce with the opposite predicate.

\[(\forall 0 \leq i < s)(V;A_1(i) \downarrow t (v_1^i, ..., v_n^i)) \quad u_i \equiv \sum_{j=0}^{p_j} \quad a_i \equiv \min_{u_j = i} j \quad vs_{t,k} \equiv [v_k^{o_i}] \quad 0 \leq i < u_s \]
\[
\quad vs_{f,k} \equiv [v_k^{i-a_i}] \quad 0 \leq i < s - u_s \]

SOAC-PARTITION

**GPU semantics:**

\(\rightarrow_g\) is a small-step semantic relation capturing reduction on a single GPU thread:

\(\forall;gexp \rightarrow_g \forall';gexp'\)
7 Translation from VCODE

To give a notion of the completeness of the $\lambda_{cu}$ intermediate representation, this section describes how the VCODE operations used in the original Nesl compiler [BC90] may be represented in $\lambda_{cu}$.

7.1 Elementwise operations

The elementwise operations (both flat and segmented) are implemented directly as parallel maps.

Maps

**Binary operators** For any binary operator $\odot$ with type $X \times Y \to Z$, the flat and segmented versions are the same.

```
kernel $K_{\odot}$ (xs : [X], ys : [Y]) -> [Z] {
  let res = MAP (i => xs[i] $\odot$ ys[i] using xs, ys ) (#xs)
  return res
}
```

**Unary operators** For any unary operator $\circ$ with type $X \to Y$, the flat and segmented versions are the same.

```
kernel $K_{\circ}$ (xs : [X]) -> [Y] {
  let res = MAP (i => $\circ$ xs[i] using xs, ys ) (#xs)
  return res
}
```
**SELECT**

kernel SELECT (flgs : [bool], xs : [T], ys : [T]) -> [T] {
  let res = MAP { i => if flgs[i] then xs[i] else ys[i] } (flgs, xs, ys) (#xs)
  return res
}

**RAND**

kernel RAND (ns : [int]) -> [int] {
  let res = MAP { i => rand(n[i]) } (ns) (#ns)
  return res
}

### 7.2 Vector instructions

**Scans** VCODE defines prefix scan operations for a number of different operations. For a monoid \( \langle S, \odot, I \rangle \), where \( S \) is a set, \( \odot \) is an associative binary operation over \( S \), and \( I \) is the identity (i.e., \( a \odot I = I \odot a = a \)), this can be implemented with the following kernel.

\( \odot \text{\_SCAN} \)

kernel \( \odot \text{\_SCAN} \) (xs : [S]) -> [S] {
  let res = SCAN { i => xs[i] } (I) { (x, y) => x \odot y } (#xs)
  return res
}

**Reductions** Like scans, reductions are defined in terms of a monoid \( \langle S, \odot, I \rangle \).

\( \odot \text{\_REDUCE} \)

kernel \( \odot \text{\_REDUCE} \) (xs : [S]) -> S {
  let res = REDUCE { i => xs[i] } (I) { (x, y) => x \odot y } (#xs)
  return res
}

**Permutations**

**PERMUTE**

kernel PERMUTE (xs : [T], ix : [int]) -> [T] {
  let res = PERMUTE { i => xs[i] } (ix) { i => ix[i] } (#xs)
  return res
}

**DPERMUTE**

kernel DPERMUTE (xs : [T], ix : [int]) -> [T] {
  let (vals, wrotes) = PERMUTE { i => (xs[i], true) } (#xs)
  let out = MAP { i => if wrotes[i] then vals[i] else default[i] } (#vals)
  return out
}
FPERMUTE

BPERMUTE

kernel BPERMUTE (xs : [T], ix : [int]) -> [T] {
    let res = MAP { i => xs[ix[i]] using xs, ix } (#xs)
    return res
}

BFPERMUTE

kernel BFPERMUTE (xs : [T], ix : [int], flgs : [bool]) -> [T] {
    let res = MAP { i => if flgs[i] then xs[ix[i]] else Z using flgs, xs, ix } (#xs)
    return res
}

DFPERMUTE

Other operations

EXTRACT returns a single element of a sequence.

kernel EXTRACT (xs : [T], ix : int) -> T {
    let res = ONCE { () => xs[ix] using xs, ix }
    return res
}

Segmented EXTRACT

kernel SEG_EXTRACT (xs : [T], ix : [int], sd : segdes) -> T {
    let res = MAP { i => xs[sd.start[i] + ix[i]] using xs, ix, sd } (#ix)
    return res
}

REPLACE modifies a single element of a sequence.

kernel REPLACE (xs : [T], ix : int, v : T) -> [T] {
    let res = MAP { i => if (ix == i) then v else xs[i] using xs, ix, v } (#xs)
    return res
}

PACK When translating NESL to λcu, it is not necessary to produce the PACK operation for conditional operations, but FILTER and PARTITION can be used to encode VCODE’s PACK operation for completeness.

kernel PACK (xs : [T], flgs : [bool]) -> [T] {
    let tmp = FILTER { i => (flgs[i], xs[i]) } (flg, x) => flg (#xs)
    let res = MAP { i => #2(tmp[i]) } (#tmp)
    return res
}
8 Loop fusion

The primary optimization enabled by $\lambda_{cu}$ is fusion of array operations. Fusion of loops over arrays results in performance benefits as described in §4, motivating the fusion of array operations in GPU programs wherever possible. However, in many cases the array operations in a control region of a program cannot all be fused into a single efficient CUDA kernel. Two array operations may have different memory access patterns or may be prevented from fusing together if an unfused loop separates them along a chain of data dependencies. In addition, it may be possible to fuse non-disjoin subsets of a set of loops, but still impossible to fuse the set as a whole into a single loop.

These fusion incompatibilities introduce nontrivial choices into the partitioning of the kernel dataflow graph into fusion clusters, and raises the question of which pairs of kernels should be fused (and which must be left unfused) to produce an output program with maximal performance. There are a number of strategies to consider for making these choices:

8.1 Greedy scheduling

The simplest approach is a greedy strategy: impose an ordering on fusion opportunities, and take them in that order, skipping any which become impossible as a result of previous choices.

One such ordering arises from working from the top downward on a dataflow graph: start with the topmost node in a dataflow graph, which must necessarily be a producer of some data consumed below. Fuse consumers into producers at each possible opportunity, moving down the dataflow graph in a breadth-first fashion. This strategy is analogous to the approach taken in a previous incarnation of this work [BR12], but misses many fusions which would be beneficial; this traversal order will “shoot itself in the foot” by choosing a weakly beneficial fusion opportunity before an incompatible one with more potential benefit.

Robinson applied a bottom-up greedy fusion strategy [RLK14], which in some cases finds more opportunities to fuse than the downward approach: because operations further down the dataflow graph are less likely to be shared as inputs to many other operations, there tends to be less interference between fusing them with nearby nodes.

Both approaches succumb to the classic pitfall of greediness—because they only examine small neighborhoods of the input at once, they pick the most immediate fusions over the most beneficial ones if the latter more contrived or require considering distant parts of the program at once.

8.2 ILP-based scheduling

A relatively direct strategy of prioritizing the most beneficial fusions can be achieved through integer linear programming (ILP). The relative predicted speedup of a fusion can be encoded into weights, and conflicts between fusions into constraints in an ILP program. This
allows to look for good fusion plans while pushing the combinatorial problem to an optimized solver.

Weights can be assigned to impose an ordering on fusions based on their precise effect on resource consumption of the resulting program: in order, fusions are prioritized which reduce the number of (1) kernels executed, (2) intermediate arrays allocated, and (3) array traversals performed.

Because control divergence is not handled by kernels, all fusion occurs within regions of the program which are not separated by conditional control flow. Instances of the ILP problem are constructed for each control region of the program. Each ILP instance is computed from the dataflow graph of a given control region.

8.3 Instance construction

Within a control region, kernels are invoked in a sequence of statements of the form

\[
\text{let } (r_1, r_2, \ldots) = \text{run } K v_1 v_2 \ldots
\]

which are known as kernel invocations. Each kernel invocation is fully described by the 3-tuple of the kernel itself (a pure function that can be viewed as a dataflow graph between SOACs), its list of argument variables, and the list of result variables it binds.

Each instance describes the fusion problem on a given sequence of kernel invocations by defining variables \( o_i \in \mathbb{N} \) specifying the sequential ordering of kernel invocation \( i \) after fusion; kernel invocations \( I_i \) and \( I_j \) are fused if and only if \( o_i = o_j \). These variables are used to define another set of variables, \( f_{i,j} \in \{0, 1\} = [o_i \neq o_j] \) (following Iverson bracket notation) which indicate whether two kernel invocations remain unfused. In addition, a set of variables \( c_a \in \{0, 1\} \) are defined indicating whether any fused kernel invocation reads from the array; in particular, \( c_a \) may be zero if fusion is able to fuse the array’s producer and consumer together. The constraint that a consumer kernel invocation which is not fused with the producer forces the array’s allocation is written as: \( \forall i, j \ I_i \text{def} a \implies c_a \geq f_{i,j} \times \lfloor I_j \text{use} a \rfloor \).

Instance construction consists of computing the benefit (decrease in objective function) and the applicability constraints for every possible fusion of two kernel invocations.

8.4 Objective function

First, all pairs of kernel invocations are evaluated to determine the potential benefit of fusing them together by counting the savings in counts of array allocations, array traversals, and kernel invocations.

If all kernel invocations which read a given array can be fused into the kernel invocation which produces that array, then the array element computation may be inlined into the consumers, saving a memory allocation and all the memory traffic involved in reads and writes. This is expressed for an array \( a \) as \( c_a = 0 \). The total cost of array allocations and initialization in a program is modelled as \( \sum_a c_a \).
Fusing kernel invocations which iterate over the same input array (or one that defines an array with one that subsequently iterates over it) results in a single iteration rather than multiple ones, saving memory traffic. Each pair of kernel invocations \( I_i \) and \( I_j \) is checked for these conditions, and the number of array reads across the two invocations is multiplied by \( f_{i,j} \) to cancel the cost of reads eliminated by fusion. This is the second component of the objective function, and, counting shared reads with \( sr \), can be written as \( \sum_{i,j} f_{i,j} sr(i, j) \).

Furthermore, any pair of kernel invocations being fused results in one fewer CPU-side kernel launch, so in general fusion is preferred over leaving kernels unfused. The number of unfused kernel invocations can be modelled with \( \sum_{i,j} f_{i,j} \).

The objective function for the ILP optimization as a whole is simply the sum of the component costs weighted by powers of \( n \), a number chosen to be greater than any of the summations, to enforce goal priorities (minimizing allocation, memory traffic, and number of invocations, in descending order):

\[
N^2 \sum_{i,j} f_{i,j} + n \sum_{i,j} f_{i,j} sr(i, j) + \sum_{a} c_a
\]

### 8.5 ILP scheduling constraints

In the absence of any constraints, simply setting \( f_{i,j} = 0 \) would minimize the objective function, but \( f \) is constrained by which fusions are valid and which are mutually exclusive.

Prior to fusion, all kernel declarations contain only a single SOAC application, so fusion compatibility of kernel invocations can be based on SOAC behavior once it is established that the other conditions for fusing invocations are satisfied.

A kernel invocation \( I = \langle K, \overline{u}, \overline{d} \rangle \) consists of its kernel \( K \), its set of used variables \( \overline{u} \), and its set of defined variables \( \overline{d} \). For convenience, we define a projection operator \( K(\langle K, \overline{u}, \overline{d} \rangle) = K \), which gives the kernel invoked by a kernel invocation.

The def relation holds between a kernel invocation \( I = \langle S, \overline{u}, \overline{d} \rangle \) and the variables it binds:

\[
I \text{ def } v \iff v \in \overline{d}
\]

Similarly, the use relation specifies the variables read by a particular kernel invocation:

\[
I \text{ use } v \iff v \in \overline{u}
\]

The into relation captures direct data dependencies; given kernel invocations \( I_1 \) and \( I_2 \),

\[
\frac{I_1 \text{ def } v \land I_2 \text{ use } v}{I_1 \text{ into } v, I_2}
\]

Because of combinatorial explosion in the number of possible fusion schedules, it is not feasible to generate the set of all partitions of the set of kernel invocations and evaluate conditions on each. Instead, dataflow edges between invocations (i.e. pairs in the into relation), are considered to determine whether edges may be fused.
The $incompat_v$ and $incompat_h$ relations capture horizontal and vertical compatibility between SOACs. Horizontal fusion may occur between all pairs of identical SOACs, and vertical fusion may occur between MAP flowing into any SOAC but ONCE, between FILTERs, between REDUCE and subsequent ONCE, and between FILTER or PERMUTE flowing into REDUCE.

The constraints on $o$ and $f$ for vertical fusion are as follows:

$$
\begin{align*}
&I_1 \text{ into } I_2 & \text{incompat}_v(K(I_1), K(I_2)) \\
&f(I_1, I_2) = 1 & \text{VFUSE-COMPAT}
\end{align*}
$$

$$
\begin{align*}
&I_1 \text{ into } I_2 \\
&o(I_1) \leq o(I_2) + f(I_1, I_2) & \text{VFUSE-CLUSTER}
\end{align*}
$$

To define the conditions for horizontal fusion, we must also have a notion of dataflow independence ($\text{indep}$) of kernel invocations: it holds for two kernels if there is no (even indirect) dataflow in either direction between a pair of kernel invocations; neither reads data influenced by the other. Dataflow paths may involve scalar code which is not expressed as run statements, so it considers more possible avenues for dataflow than would simply relying on the negation of the transitive closure of the $\text{into}$ relation. It can nonetheless be easily computed from a program dependence graph.

If two nodes’ kernels are incompatible, they cannot participate in horizontal fusion:

$$
\begin{align*}
&I_1 \text{ indep } I_2 & \text{incompat}_h(K(I_1), K(I_2)) \\
&f(I_1, I_2) = 1 & \text{HFUSE-COMPAT}
\end{align*}
$$

When two dataflow-independent nodes $I_2, I_2$ are “siblings” (share an input), they may be fused if their kernels are compatible, and we say $I_1 \parallel I_2$. However, this does not permit concluding anything about $o$ or $f$.

Ideally we would reason about size directly, but for now we use shared inputs as a conservative proxy for size information. However, not all inputs to a kernel invocation are guaranteed to be the size of the iteration space (e.g., BPERMUTE iterates over the index array, not the source array), so we should take this into account when using inputs to determine size compatibility.

$$
\begin{align*}
&I_1 \text{ indep } I_2 & I_2 \text{ use } v & I_1 \text{ use } v \\
&I_1 \parallel I_2 & \text{HFUSE-SIBS}
\end{align*}
$$

If two nodes are not siblings, but are still dataflow-independent, it is impossible to fuse them unless some other fusion leads to their iteration spaces becoming equal. This could occur either when the two nodes have a dataflow parent which becomes common after fusion, or via one node fusing with the parent of the other. In this case the constraint expressed is that if none of these conditions occur, then the two nodes must not be fused.
\[ I_1 \text{ indep } I_2 \quad I_1 \parallel I_2 \quad \neg \text{incompat}_h(\mathbb{K}(I_1), \mathbb{K}(I_2)) \]
\[
\text{allPairs} = |\{I_{p1} : I_{p1} \text{ into } I_1\}| \ast |\{I_{p2} : I_{p2} \text{ into } I_2\}|
\]
\[
\text{unfusedPairs} = \sum_{I_{p1} \text{ into } I_1 \atop I_{p2} \text{ into } I_2} f(I_{p1}, I_{p2})
\]
\[
1 - f(I_1, I_2) \leq \text{allPairs} - \text{unfusedPairs}
\]

9 Fusion rewriting

Given a partition of the kernel invocations in a dataflow region, it then remains to rewrite the kernel invocations in each cluster in the partition as a single fused kernel invocation computing the same result. This rewriting is essentially independent of the scheduling algorithm that produces the partitioning, as long as fusion is feasible.

9.1 Rewrite existence

It is not obvious \textit{a priori} that it is always possible to rewrite many kernel invocations and any control-flow-free scalar code between them as the invocation of as a single kernel. When fusing pairs of kernel invocations which are separated in the dataflow graph by scalar computations, kernel fusion is nonetheless possible by duplicating the scalar operations into each thread or performing them in a single thread and broadcasting the results. To allow defining fusion as acting on a dataflow graph of only kernel invocations, any scalar computation can be moved into the thunk of a new kernel applying the \texttt{ONCE} SOAC, and invoking that kernel in place of the scalar computation.

Fusion between kernel invocations is performed iteratively: pairs of kernel invocations are combined into a single invocation calling a fused kernel, which is defined using both the definitions of the two called kernels and the contextual information about dataflow between them. This process is repeated until all kernels in a cluster have been merged.

Since a kernel invocation is simply a \texttt{$\lambda$} let statement applying a kernel, the meaning of a kernel invocation is simply binding \texttt{vars} to the results of passing \texttt{args} to \texttt{kern}. Two kernel invocations in sequence are indistinguishable from a single kernel invocation that binds the same set of variables to the same values. However, fusion must do more than merely combine kernels; it must also remove intermediate array allocations.

Operationally, a kernel invocation can be seen as allocating space for the variables it binds, then passing these allocations to the kernel. To remove allocations, the kernel invocations must be rewritten to a kernel invocation which binds a potentially smaller set of variables than those bound the invocations fused to create it.

The output of the fusion of kernel invocations \( I_1 \) and \( I_2 \) (where data flows from \( I_1 \) to \( I_2 \) but not vice-versa) binds the union of the sets of variables bound by \( I_1 \) and \( I_2 \), minus any variables referenced only by \( I_1 \).
9.2 SOAC fusion rewrite rules

Thunk combination is denoted with the \( \hat{+}_A \) operator. The thunk on the right-hand side of the operator may read data in a variable computed by the left-hand side, which is denoted by the presence of that variable in the operator’s subscript. Thunk combination operates syntactically on two thunks and a variable \( A \) as follows.

Let \( i : \text{Var}, e_1 : \text{Expr}, e_2 : \text{Expr}, xs : \text{Var}^*, ys : \text{Var}^*, \) and \( A : \text{Var}; \) let \( a \) be a fresh variable that does not appear in \( e_1 \) or \( e_2 \).

\[
\{ i \Rightarrow e_1 \text{ using } xs \} \hat{+}_A \{ i \Rightarrow e_2 \text{ using } ys \} = \{ i \Rightarrow \text{let } a = e_1 \text{ in } [A [i]/a]e_2 \text{ using } xs \cup (ys \setminus \{A\}) \}
\]

\[
\begin{align*}
\text{MAP-MAP} & \quad A = \text{MAP } A_1 n \
& \quad \oplus \
& \quad B = \text{MAP } A_2 n \
& \quad B = \text{MAP } (A_1 \hat{+}_A A_2) n
\end{align*}
\]

\[
\begin{align*}
\text{MAP-PERMUTE} & \quad A = \text{MAP } A_1 n \
& \quad \oplus \
& \quad B = \text{PERMUTE } A_2 A_3 n \
& \quad B = \text{PERMUTE } (A_1 \hat{+}_A A_2) A_3 n
\end{align*}
\]

\[
\begin{align*}
\text{MAP-REDUCE} & \quad A = \text{MAP } A_1 n \
& \quad \oplus \
& \quad b = \text{REDUCE } A_2 id_r A n \
& \quad b = \text{REDUCE } (A_1 \hat{+}_A A_2) id_r A n
\end{align*}
\]

\[
\begin{align*}
\text{FILTER-FILTER} & \quad A = \text{FILTER } A_1 \{ v \Rightarrow p_1 \text{ using } ys \} n \
& \quad \oplus \
& \quad B = \text{FILTER } A_1 \{ v \Rightarrow p_2 \text{ using } ws \} (#A) \\
& \quad \{ i \Rightarrow \text{let } a = \text{body}(A_1) \text{ in } p_1 \& \& [A [i]/a]p_2 \text{ using } ys \cup (ws \setminus \{A\}) \} n
\end{align*}
\]

\[
\begin{align*}
\text{FILTER-REDUCE} & \quad A = \text{FILTER } \{ i \Rightarrow e_1 \text{ using } xs \} \{ w \Rightarrow p \text{ using } ys \} n \
& \quad \oplus \
& \quad b = \text{REDUCE } \{ i \Rightarrow e_2 \text{ using } zs \} id_r A (#A) \\
& \quad b = \text{REDUCE } \{ i \Rightarrow \text{let } a = e_1 \text{ in } \text{if } [a/w]p \text{ then } [A [i]/a]e_2 \text{ else } id_r \text{ using } xs \cup (ys \cup zs \setminus \{A\}) \} \}
\end{align*}
\]

10 Code generation

After fusion, \( \lambda_{cu} \) programs are lowered to CUDA C++ and passed into the nVidia \texttt{nvcc} compiler.

The top-level code generator accepts a \( \lambda_{cu} \) program and a specification of CUDA target device properties (such as CUDA hardware generation), and generates C++ source code with CUDA kernels and CUDA kernel invocations.

The lowering produces a kernel function marked with the \texttt{global} CUDA keyword for each of the kernels in the \( \lambda_{cu} \) program, as well as a top-level C++ function definition for each function definition in the \( \lambda_{cu} \) program. Lowering for code other than kernels and \texttt{run} statements is unremarkable: \( \lambda_{cu} \) already resembles static single assignment, which can be translated into initialized C++ declarations as the CPU expression language is as expressive as a subset of C++.

Each run statement is translated to a CUDA kernel invocation using grid and block dimensions compatible with the lowered kernels. In particular, this implies using block
sizes no larger than those expected by the lowerings of scans and reductions, which rely on block-level synchronization.

10.1 CPU-GPU synchronization and copying
The CUDA API only allows allocating memory for arrays through a CPU-side call to cudaMalloc, and in the C++ API kernel invocations are required to return void. Because of these constraints, the lowering of $\lambda_{cu}$ kernel invocations first allocates space on the CPU side and then writes into it space from the GPU, even for scalar return values. This is encapsulated in the Scalar<T>, Sequence<T>, and SegDes C++ classes included in the CUDA output, which wrap the logic of allocation, initialization, segment descriptor creation, and cudaMemcpy. These classes track whether the value has been fetched from the GPU yet, so that generated code does not need to enforce this invariant through its own control flow. Arrays which are simply written from the GPU do not need the CPU to initialize them, but scalar values used as the target of reductions must be initialized to the identity value of the reduction operator used, and arrays read from files need to be copied from CPU-side storage to GPU memory.

11 Related work
11.1 Common threads
A number of systems have explored a wide gamut of points in the design space of compilers for nested data parallelism. In addition to the general-purpose compilers which accept existing high-level programming languages, there are many specialized languages for expressing nested data-parallel operations. These span the gamut of:

- General-purpose vs DSL compilers
- Multi-target vs GPU-specific
- Nested vs flat data parallelism
- Regular vs irregular NDP

11.2 Conceptual tools for optimizing array operations
- Array SSA - Single Static Assignment (SSA) lies at the heart of many optimizing compilers for languages with imperative features. Because the cost of copying arrays is so high, even functional languages which offer first-class support for arrays often introduce mutability as an optimization. Array SSA extends the framework of Single Static Assignment with an analogue of the $\phi$ function that facilitates reasoning about updates to subindices of arrays.
- Push/pull arrays (in Obsidian and other systems) - $\lambda_{cu}$ benefits from a pull-based representation of arrays, where thunks are applied to index spaces to generate array elements; push arrays are dual in a sense, indicating where to place generated elements in memory. There are interesting and worthwhile fusion opportunities within and between these two views of arrays, in addition to the pull-array fusion performed by Nessie.
– Uniqueness types [dPA07] - Reasoning about the usage of variables across control flow can allow for greater memory reuse; uniqueness types are one such static analysis.

11.3 Other NDP compilers

– NESL [Ble95]
– Robinson’s work on fusing filters [RLK14]
– Obsidian [CSS12]
– Thrust [HB11]
– Nova [CGG+14]
– NESL/GPU [BR12]
– Proteus [PPW95]
– Data-Parallel Haskell [LCK+12] [CLP+07]
– Nepal [CKLP01]
– Repa
– Accelerate [CKL+11]
– Futhark
– CuNesl [ZM12]
– Delite
– HIIDP [ZM13]

12 Future work

12.1 Memory management

One goal of λcu is to enable optimizing away the expensive allocations of temporary arrays. In addition to fusion, which can eliminate temporary variables entirely if all consumers are fused into computation that produces the value, allocations of temporary arrays can be avoided by reusing the storage of arrays at their last use site to store the results of the final operation. This problem is similar to that of register allocation, where storage locations should be selected to minimize the size of the working set, but with the added complication that array sizes must be compatible (so that a larger allocation can be reused to store any smaller array) and that sizes of some arrays are not fully known until runtime.

Memory reuse would therefore benefit from additional information about array sizes, which can be inferred by propagating size constraints across programs according to dataflow. This may be able to take inspiration from prior work on size inference in Futhark [HEO14].

12.2 User-defined datatypes

User-defined recursive datatypes provide a significant expressiveness benefits for programmers, and it would be ideal for the Nessie compiler to support them. Other work has investigated compilation strategies for recursive datatypes in a nested data-parallel setting, but the optimal strategy is not yet obvious. Possible avenues of investigation include datatype unrolling [SRA94] and point blocking or traversal splicing.
12.3 Decomposing filters

Nessie currently lowers filter operations into a sequence of three parallel operations: mapping the filter predicate, a prefix sum to compute destination indices, and a permutation to place output data. This lowering occurs after fusion has been performed, so these lowered parallel operations do not see the full benefit of Nessie’s reasoning about array operation fusion. Performing subsequent iterations of fusion or integrating knowledge of filter lowering into the optimization problem used in the single fusion pass may provide substantial speedup for some filter-heavy programs.

12.4 CPU-GPU concurrency

Nessie separates data-parallel GPU array operations from CPU control flow, but from the CPU perspective, execution of data-parallel operations is synchronous. Taking advantage of parallelism between kernels could have a significant impact on GPU occupancy and provide performance improvements. Removing CPU-GPU synchronization points by making use of CUDA “dynamic parallelism” to launch kernels from other kernels could also contribute to increased parallelism.

12.5 Low-level optimizations

There are unrealized opportunities for optimization of data representations, including segment descriptors for nested arrays. Projects such as HiDP [ZM13] report performance gains by specializing segment descriptors for regular (identically-sized) segments; condensed representations using only segment lengths can also be used when random access to the segment for a given array index is not necessary. It may be possible to statically select the most efficient segment descriptor format for each array variable based on its usage patterns. Costs of conversions between segment descriptor representations could be integrated into Nessie’s ILP-based optimization selection.

References


