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

Joe Wingerter

University of Chicago

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, λ_{cu}, which constitutes the primary contribution of this work. λ_{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 naïvete 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 (programmers must manage allocations and refernce lifetime manually) and code reuse (as general-purpose libraries cannot be reused on the GPU), which result in low programmer productivity. 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; this work presents improvements to the translation of NESL programs into CUDA, the low-level programming model for NVIDIA GPUs.

1.2 Background

There is a long history of research into compiling for data parallelism. Because of the many existing programs written in languages oriented toward serial processing of scalar values, much effort has been devoted to automatic parallelization, which seeks to automatically identify opportunities for task- or data-parallel execution of programs specified in a serial fashion. 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.

Research into compiling for parallel processors has explored different avenues depending on the organization of the prevailing hardware platform of the day. Without any specialized hardware support, a serial program can be adapted for parallel execution by adopting the *single program, multiple data* (SPMD) paradigm, which runs multiple copies of the same program on independent machines or processors. SPMD supports both task- and data-parallelism, but can be less efficient than other approaches to parallelization due to unnecessary duplication of control state and other data which may be identical on different processors.

The *single instruction multiple data* (SIMD) approach relies on hardware support for vector instructions that operate on multiple input items at once, organized into bounded- or fixed-size vectors. This enables data parallelism up to the vector width of the processor, but often requires programs to be reorganized to process input items in vector-sized batches. *Autovectorization* attempts to automate this reorganization by performing compile-time analyses to mechanically translate loops in scalar programs from using scalar operations to vector ones, but analyses for autovectorization can be both fragile and costly. In many cases programmers cannot rely on autovectorization to provide consistent speedups and instead rely on hand-optimized vectorized implementations of critical routines, such as those in the BLAS [LHKK79] library.

The advent of GPUs has led to the popularization of the *single instruction multiple thread* (SIMT) paradigm of parallel execution. This model, though executing a single program across many data elements like SIMD, differs by featuring a large number of processors operating on data items of small width (as opposed to wide vector registers). These processors share an instruction pointer and control logic, but have their own sets of registers and cache memory, and perform scalar memory accesses as opposed to loading entire contiguous vectors at once as is most common in SIMD.

In addition, 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 hide memory latency by switching to execute an unrelated thread while the original thread awaits completion of memory operations.

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 *nested data parallelism* (NDP), where parallel operations can be nested arbitrarily and vectors may contain rich variety of data types including other vectors. Composable parallel constructs as found in NDP (such as arrays-of-arrays, sequence comprehensions, sums of sums, *etc.*) avoid specifying the sequence of individual scalar operations; as a result, compilers have more flexibility in scheduling subcomputations without requiring complex analysis to justify reordering or fusion, and the task of programmers becomes simpler as programs using these constructs can more closely resemble declarative specifications. 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.

1.3 The flattening transform

The *flattening transform* converts NDP into flat data parallelism, which is straightforward to execute efficiently on GPUs and vector processors. Introduced by Blelloch [BS90] [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. Unlike using the parallel operations at the leaves of nesting as the unit of flat parallelism, processing flattened arrays and segment descriptors produces large uniformly workloads that do not require runtime load balancing to prevent computational capacity sitting idle. In addition, because GPU architectures exhibit high memory latency relative to their computational capacity, pointer-based implementations of nested arrays are undesirable: workloads heavy on pointer chasing can exceed the latency-hiding capability of "barrel threading"-style pipelining of thread execution during memory accesses. In other cases, access patterns may force serialization of accesses (as when traversing a linked list, or when address patterns force bank conflicts) and lead to decreased throughput. As such, it is desirable to rely on flat data layout and carefully-planned access patterns to achieve maximal throughput.

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

Low-level general-purpose programming on GPUs is supported by major GPU vendors through two systems, CUDA and OpenCL, which offer similar semantics but target different sets of underlying hardware. Using these systems a programmer constructs a program in a C-like language using pointers, arrays, and mutable updates, which is executed in lockstep on 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 primitive datatypes and operations built into CUDA and OpenCL.

1.4 A language for NDP

The NESL language is a full-featured strict, first-order, array-oriented functional programming language with a number of features for nested data parallelism [Ble95]. The foremost parallel primitive provided by NESL is a parallel "apply to each" construct which operates on sequences of values of the same type, along with an optional predicate to select only a subset of elements. 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 component of a program, 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 to provide a language offering greater productivity when programming for supercomputers of the era. The initial implementation of NESL was compiled via translation to VCODE [BCH+94], a stack-based, array-oriented intermediate language whose opcodes were dispatched by an interpreter loop to optimized hand-written parallel implementations [BC90]. The vast majority of computation in data-parallel programs occurs in vector operations, so interpretive overhead did not impose an unacceptable performance cost. For a modern, GPU-targetted implementation of NESL, this design is no longer appropriate because of the growing disparity between memory latency and the speed of arithmetic operations; synchronous communication between the CPU and GPU after each array operation would dominate the time consumption of program execution.

```
function dotprod(a,b) =
  sum({x * y: x in a ; y in b});
```

The dot-product of two arrays is expressed in NESL simply as the sum of their pairwise products. Pairwise multiplication uses a sequence comprehension, selecting corresponding elements from each array.

```
function quicksort(a) =
if (#a < 2) then a
else
  let pivot   = a[#a/2];
      lesser  = {e in a| e < pivot};
      equal   = {e in a| e == pivot};
      greater = {e in a| e > pivot};
      result  = {quicksort(v): v in [lesser,greater]};
  in result[0] ++ equal ++ result[1];
```

An implementation of quicksort demonstrates other sequence operations supported by the language and the usefulness of filtering to refer to subsets of sequences.

Fig. 1: Examples of NESL programs, demonstrating the feel of programming in NESL and how the language's features can be used for practical computations.

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.

2.1 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 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. Warps of threads are grouped into blocks, and all threads in a block can communicate across shared memory. A variable number of warps can be executed on a single Streaming Multiprocessor (SM), depending on how much memory is required for each thread; this bounds the size and count of the blocks used for execution of a kernel. 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].

```
int max_block_size = 1024; // max size of one dimension of grid
int shmem_bytes = 0; // how much extra share memory to allocate
int stream = 0; // which cpu-gpu synchronization queue to use

#define div_round_up(x, by) (((x) + (by) - 1)/(by))

__device__ void mul_kernel(size_t len, int* a, int* b, int* out) {
    // compute the index to operate on from the id of this thread
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    // the last block may have threads past the end of the array,
    // so compare the index against len
    if(i < len)
        out[i] = a[i] * b[i];
}

void pairwise_mul(size_t len, int* a, int* b, int* out) {
    // specify one-dimensional grid and thread block
    dim3 grid_dim(div_round_up(len, max_block_size), 1, 1);
    dim3 block_dim(min(len, max_block_size), 1, 1);

    mul_kernel<<<grid_dim, block_dim, shmem_bytes, stream>>>(N, a, b, out);
}
```

Fig. 2: Declaration and invocation of a CUDA kernel mapping pointwise multiplication over a pair of arrays. `grid_dim` and `block_dim` are 3-vectors specifying a grid of blocks to execute and how many threads are in each block in the grid. `shmem_bytes` specifies how much extra shared memory to allocate, and `stream` specifies which CPU-GPU synchronization queue to use.

2.2 Memory regions and hierarchy

The CUDA model presents several memories of differing capacities, levels of sharing, and access latencies. All CUDA memory spaces have significantly higher maximum throughput compared to main memory in desktop and server-class machines.

The fastest memory available to CUDA programs serves 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 the number of registers required by currently-executed kernels [NVI15].

Shared memory is slower than registers, but can be used for cross-thread communication or to parallelize loads from global memory through collaboration from multiple threads.

Global memory is shared across the entire GPU, but has much higher access latency than shared memory or registers. In addition to backing global memory, the same device hardware also acts as the underlying storage for local and texture memory:

Each SM is assigned *local memory*, a portion of device memory used to hold per-thread values spilled from registers or which require pointer semantics (such as support for indirect addressing).

Constant memory is a portion of device memory to which SMs do not perform writes, and which benefits from additional caching as a result of invalidation being impossible.

Texture memory is a portion of device memory that allows 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 with a number of behaviors useful for texture data. Nessie does not presently use texture memory because its specialized feature set (read-only interpolated sampling) does not overlap with the functionality required by most general compute workloads. The bandwidth and latency of texture memory are similar to those of global memory.

In hardware supporting NVIDIA's Compute Capability 2.0, accesses to local, global, and texture memory can benefit from two layers of caching: a globally-coherent L2 cache, and per-SM L1 caches (with identical performance to shared memory). All accesses are cached by default, but instructions exist to perform uncached loads and stores to local and global memory as well as uncached loads from texture memory, and to access particular levels of cache explicitly. Aside from the caches, values are only placed into the different memory spaces through explicit loads into and stores from registers.

2.3 Thread, block, and kernel synchronization

CUDA exposes synchronization and communication primitives at warp, block, and kernel levels of granularity. Within a warp, all threads are inherently synchronized, sharing a program counter and running in lockstep; divergent control flow of threads within a warp is achieved through predicated execution. 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 straighforwardly by introducing kernel boundaries and serializing execution of kernels. Because blocks are multiplexed across SMs, it is not, in general, possible for two arbitrary blocks to synchronize, since one may only be scheduled after the other on the same SM. Following [XF10] and [SDB$^+$16], it is possible to achieve this form of synchronization 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. In this configuration atomic operations can be used to synchronize execution of blocks in the same kernel without requiring the CPU-GPU communication of a kernel launch. Alternatively (at higher latency than using atomic operations), blocks can synchronize using thread-level synchronization primitives and communication through global memory.

Because of the implementation complexity and inter-block concurrency constraints imposed by block synchronization techniques, Nessie relies on simple kernel-based synchronization, running a single kernel at a time.

2.4 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 a different memory bank, and broadcast access, where multiple threads in a warp request the same word in each bank accessed. Each word in shared memory is stored in one of 32 banks based on the low bits of its address [NVI15].

On hardware supporting NVIDIA's Compute Capability 2.0, 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.

3 Mitigating GPU performance hazards

Given the architecture described above, there are several rules of thumb that guide the optimizations implemented in Nessie. Each optimization attempts to avoid 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 steps has a performance cost; rewriting computations that use temporary array variables into ones without temporaries can eliminate all three costs at once. In Nessie, this optimization is achieved through fusion of vector computations.
2. *Bank conflicts* – When reading data in global memory, concurrent reads of different addresses in the same bank (*i.e.*, two words whose locations differ by a multiple of 32 words) will incur access serialization [NVI15]. Serialization can be avoided by duplicating the data, changing the timing or ordering of accesses, or via higher-level program transformations that find a way to avoid the read entirely. Nessie's parallel primitives implement efficient patterns for distributing data: permutations are required to be 1:1, and scans and reductions are lowered to CUDA code that avoids bank conflicts. Data required by multiple threads is broadcast, 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. Rewriting computations to read less data (*e.g.*, by using smaller scalar datatypes, arrays of smaller dimensions, or rematerialization) will put less pressure on available memory bandwidth, and will not require as much parallel arithmetic 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 that do not take a branch will 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 §2.3). 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.

4 The λ_{cu} IR

$$
\begin{array}{lcl}
\mathit{prog} & ::= & \mathit{kern}^* \mathit{func}^* \mathit{cexp} \\
\mathit{soac} & ::= & \texttt{ONCE} \\
 & | & \texttt{MAP} \\
 & | & \texttt{PERMUTE} \\
 & | & \texttt{REDUCE} \\
 & | & \texttt{SCAN} \\
 & | & \texttt{FILTER} \\
 & | & \texttt{PARTITION} \\
\mathit{arg} & ::= & \Lambda \\
 & | & \mathit{rop}^* \\
\Lambda & ::= & x^* \texttt{=>}\ \mathit{gexp}\ \texttt{using}\ x^* \\
\mathit{gexp} & ::= & \texttt{let}\ x^* \texttt{=}\ \mathit{gexp}\ \texttt{in}\ \mathit{gexp} \\
 & | & \mathit{prim}\ \mathit{gatm}^* \\
 & | & \texttt{if}\ \mathit{gatm}\ \texttt{then}\ \mathit{gexp}\ \texttt{else}\ \mathit{gexp} \\
 & | & \texttt{ret}\ \mathit{gatm}^* \\
 & | & x\texttt{[}\mathit{gatm}\texttt{]} \\
 & | & \mathit{gatm} \\
\mathit{gatm} & ::= & x \\
 & | & v
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{kern} & ::= & \texttt{kernel}\ k\ x^* \mathit{kexp} \\
\mathit{kexp} & ::= & \texttt{let}\ x^* \texttt{=}\ \mathit{soac}\ \mathit{arg}^* \mathit{kexp} \\
 & | & \texttt{ret}\ x^* \\
\mathit{func} & ::= & \texttt{function}\ f\ x^* \mathit{cexp} \\
\mathit{cexp} & ::= & \texttt{let}\ x^* \texttt{=}\ \mathit{cexp}\ \texttt{in}\ \mathit{cexp} \\
 & | & \texttt{if}\ \mathit{catm}\ \texttt{then}\ \mathit{cexp}\ \texttt{else}\ \mathit{cexp} \\
 & | & f\ x^* \\
 & | & \texttt{run}\ k\ \mathit{catm}^*\ \mathit{shape}^* \\
 & | & \texttt{tuple}\ \mathit{catm}^* \\
 & | & \mathit{prim}\ \mathit{catm}^* \\
 & | & \mathit{catm} \\
\mathit{catm} & ::= & x \\
 & | & v \\
 & | & \mathit{vs} \\
v & ::= & \mathit{bool} \\
 & | & \mathit{char} \\
 & | & \mathit{int} \\
 & | & \mathit{float} \\
\mathit{vs} & ::= & \mathit{bool}^* \\
 & | & \mathit{char}^* \\
 & | & \mathit{int}^* \\
 & | & \mathit{float}^*
\end{array}
$$

Fig. 3: Abstract syntax of λ_{cu}.

The λ_{cu} IR is designed to capture the relevant constraints of efficient GPU execution and enable optimizations within those constraints. To achieve this goal, λ_{cu} syntactically separates (1) scalar-producing computations that run in data-parallel fashion from (2) the kernels that 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 structure determined by the list of array combinators it applies. These combinators are called *Second-Order Array Combinators* (SOACs) because they apply first-order *thunks* (which compute scalar elements of array contents) to an index sequence corresponding to a particular array access pattern; the distinction between SOACs and thunks separates the description of an array traversal from computations of array element values. This separation makes enables simple detection of when array contents influence traversal patterns, simplifying high-level optimizations on λ_{cu} that depend on static knowledge of array traversals.

$$
\begin{array}{lll}
\textit{base} & ::= & \mathtt{bool} \\
 & \mid & \mathtt{char} \\
 & \mid & \mathtt{float} \\
 & \mid & \mathtt{int} \\
 & \mid & \mathtt{idx}
\end{array}
$$

$$
\begin{array}{lll}
\textit{bases} & ::= & \textit{base}^{+} \\
 & \mid & \mathtt{()}
\end{array}
$$

$$
\begin{array}{lll}
\textit{arr} & ::= & \mathtt{[}\textit{base}\mathtt{]}
\end{array}
$$

$$
\begin{array}{lll}
\textit{arrs} & ::= & \textit{arr}^{+} \\
 & \mid & \mathtt{()}
\end{array}
$$

$$
\begin{array}{lll}
\textit{shape} & ::= & \mathtt{*} \\
 & \mid & n \\
 & \mid & \mathtt{\#}v \\
 & \mid & \textit{shape}+\textit{shape}
\end{array}
$$

$$
\begin{array}{lll}
\textit{type} & ::= & \mathtt{rops} \\
 & \mid & \{\ \textit{bases} \Rightarrow \textit{bases}\ \} \\
 & \mid & \textit{shape} \\
 & \mid & \textit{arrs} \\
 & \mid & \textit{bases}
\end{array}
$$

Fig. 4: λ_{cu} type grammar.

A program is represented in λ_{cu} by a list of kernel declarations, a list of function definitions, and an expression serving as the program entrypoint. Function bodies may refer to any kernel or any function in the program. Function definitions contain code that runs on the CPU and does not itself 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.

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

4.1 Second-order array combinators (SOACs)

An SOAC specifies a pattern of data dependencies and parallelism within an array operation. Each SOAC accepts one or two thunk arguments, followed by a tuple of reduction

$$
\begin{aligned}
\textbf{ONCE} &: \{\ ()\ \Rightarrow bs\ \} \rightarrow bs \\
\textbf{MAP} &: \{\ \texttt{idx} \Rightarrow bs\ \}\ shape \rightarrow [bs]^{\uparrow} \\
\textbf{PERMUTE} &: \{\ \texttt{idx} \Rightarrow bs\ \}\ \{\ \texttt{idx} \Rightarrow \texttt{idx}\ \}\ shape \rightarrow [bs]^{\uparrow} \\
\textbf{REDUCE} &: \{\ \texttt{idx} \Rightarrow bs\ \}\ \texttt{rops}\ shape \rightarrow [bs]^{\uparrow} \\
\textbf{SCAN} &: \{\ \texttt{idx} \Rightarrow bs\ \}\ \texttt{rops}\ shape \rightarrow [bs]^{\uparrow} \\
\textbf{FILTER} &: \{\ \texttt{idx} \Rightarrow bs\ \}\ \{\ bs \Rightarrow \texttt{bool}\ \}\ shape \rightarrow [bs]^{\uparrow} \\
\textbf{PARTITION} &: \{\ \texttt{idx} \Rightarrow bs\ \}\ \{\ bs \Rightarrow \texttt{bool}\ \}\ shape \rightarrow [bs]^{\uparrow}\ [bs]^{\uparrow}
\end{aligned}
$$

Fig. 5: SOAC type signatures. Each SOAC's type is indexed over a tuple *bs* of base types (*i.e.*, $\forall bs \in bases, \ldots$). Lifting base-type tuples into array-type tuples is denoted by $[t]^{\uparrow}$:

$$[\,()\,]^{\uparrow} = () \qquad [base_1, \ldots, base_n]^{\uparrow} = [base_1], \ldots, [base_n]$$

operators (rop) for those which merge thunk results, and a shape which determines the iteration space. (**ONCE** does not accept a shape because it operates over a trivial index space.)

$$
\begin{aligned}
rop \quad ::= &\ \textbf{iadd} \mid \textbf{imul} \mid \textbf{imax} \mid \textbf{imin} \mid \textbf{iadd} \mid \textbf{ior} \mid \textbf{ixor} \\
\mid &\ \textbf{fadd} \mid \textbf{fmul} \mid \textbf{fmax} \mid \textbf{fmin} \mid \textbf{band} \mid \textbf{bor} \mid \textbf{bxor} \\
\mid &\ \textbf{id}
\end{aligned}
$$

Fig. 6: Reduction operators. A tuple of reduction operators, one for each array being processed, is used with the **REDUCE** and **SCAN** SOACs to specify how to combine elements of the corresponding array.

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

The **id** reduction operator is unique in that it forms a monoid operating on arrays, not scalar values; it is handled specially by **REDUCE** and **SCAN** and indicates to return an array as a whole rather than performing a reduction on it.

Thunks

Thunks express the per-element work performed by some arrays. Each thunk maps from an index to a scalar or tuple result.

Within a scalar expression, thunks can reference literal scalar values, combine their parameters with arithmetic primitive operations, bind variables, evaluate expressions conditionally, read the thread index, and index into captured array variables. When applied by an 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 §8.

5 Semantics of the IR

5.1 Semantics

The syntactic sets used to give the semantics of λ_{cu} consist of several base sets containing the classes of primitive scalar values from which compound values are constructed, as well as the compound sets: homogeneous arrays, tuples of scalars, and tuples of equal-length arrays.

The sets giving scalars are:

- Integers $iv \in \mathbb{Z}$
- Truth values $bv \in \mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$
- ASCII characters $cv \in \mathbf{A}$
- Real numbers $rv \in \mathbb{R}$

Arrays can be constructed from any number of elements taken from a single one of these scalar syntactic sets. Given a scalar syntactic set $\mathbf{X}$, the set of arrays of length n of elements taken from $\mathbf{X}$ is denoted as $\mathbf{X}^n$, defined as the n-ary cartesian product of $\mathbf{X}$:

$$\mathbf{X}^n = \prod_{i=0}^{n} \mathbf{X}$$

The set of all arrays whose elements are selected from the scalar syntactic set $\mathbf{X}$ is denoted as $\mathbf{X}^*$, and defined as:

$$\mathbf{X}^* = \bigcup_{n=0}^{\infty} \mathbf{X}^n$$

To specify elements of these array sets, some notation is necessary. The notation $[x := y]z$ denotes substitution, replacing all occurrences of the symbol x in a formula z with the formula y. The notation $[expr \mid 0 \leq i < n]$ denotes the array value of length n whose element at index i is given by $[i := 0]expr$. Metavariables for array values follow the convention of appending s to the variable that would be used with the scalar set from which the array's elements are drawn.

Values may also be tuples of scalar values. A tuple of n scalar elements is written as $\langle x_1, ..., x_n \rangle$, and a tuple of n array values of equal length is written as $\langle xs_1, ..., xs_n \rangle$. Tuples containing only one element are identified with their contents. Projection of the i^{th} element of a tuple t is denoted $t \,.\, i$.

In addition, the semantics refers to the various nonterminal productions of the λ_{cu} grammar, as given in Figure 3.

5.2 Environment

The set $\mathbb{F}$ is defined as the smallest set generated by the two rules below:

$$\frac{\mathit{func} \subset \mathit{prog}}{\mathit{func} \in \mathbb{F}}\ \textsc{func-def} \qquad\qquad \frac{\mathit{kern} \subset \mathit{prog}}{\mathit{kern} \in \mathbb{F}}\ \textsc{kern-def}$$

5.3 Evaluation

λ_{cu} programs are executed by evaluating their body in the environment created from their kernel and function definitions:

$$\frac{\mathit{prog} = \mathit{funcs}\ \mathit{kerns}\ \mathit{cexp} \qquad \mathit{prog}\ \mathbf{def}\ \mathbb{F} \qquad \mathbb{F} \vdash \emptyset; \mathit{cexp} \Downarrow_{\mathsf{c}} v}{\mathit{prog} \Downarrow v}\ \textsc{eval-main}$$

The $\Downarrow_{\mathsf{c}}$ relation is a big-step semantic relation capturing CPU-level reduction, which involves pairs of CPU-level λ_{cu} expressions and value environments $\mathbb{V} = \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}$ which are finite maps from variables to values. Lookup of the variable x in a variable environment $\mathbb{V}$ is denoted $\mathbb{V}(x)$.

Given a function environment $\mathbb{F}$, $\Downarrow_{\mathsf{c}}$ relates a pair of variable environment and an expression to the value resulting from the expression's evaluation within that environment:

$$\mathbb{F} \vdash \mathbb{V}; \mathit{cexp} \Downarrow_{\mathsf{c}} v$$

This relation is defined inductively by the following rules:

$$\frac{\mathbb{F} \vdash \mathbb{V}; \mathit{cexp}_{rhs} \Downarrow_{\mathsf{c}} v_{rhs} \qquad \mathbb{F} \vdash \mathbb{V} \cup \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}; \mathit{cexp}_{body} \Downarrow_{\mathsf{c}} v_{body}}{\mathbb{F} \vdash \mathbb{V}; \mathbf{let}\ x_1\ ...\ x_n = \mathit{cexp}_{rhs}\ \mathbf{in}\ \mathit{cexp}_{body} \Downarrow_{\mathsf{c}} v_{body}}\ \textsc{cpu-let-exp}$$

$$\frac{\mathbb{V} \vdash \mathit{catm}_{cond} \longrightarrow_{\mathsf{a}} \mathbf{true} \qquad \mathbb{F} \vdash \mathbb{V}; \mathit{cexp}_t \Downarrow_{\mathsf{c}} v}{\mathbb{F} \vdash \mathbb{V}; \mathbf{if}\ \mathit{catm}_{cond}\ \mathbf{then}\ \mathit{cexp}_t\ \mathbf{else}\ \mathit{cexp}_e \Downarrow_{\mathsf{c}} v}\ \textsc{cpu-if-true}$$

$$\frac{\mathbb{V} \vdash \mathit{catm}_{\mathit{cond}} \longrightarrow_{\mathsf{a}} \mathbf{false} \qquad \mathbb{F} \vdash \mathbb{V}; \mathit{cexp}_e \Downarrow_{\mathsf{c}} v}{\mathbb{F} \vdash \mathbb{V}; \mathbf{if}\ \mathit{catm}_{\mathit{cond}}\ \mathbf{then}\ \mathit{cexp}_t\ \mathbf{else}\ \mathit{cexp}_e \Downarrow_{\mathsf{c}} v} \ \text{CPU-IF-FALSE}$$

$$\frac{\begin{array}{c} \mathbf{function}\ f\ x_1\ ...\ x_n\ \mathit{cexp}_{\mathit{body}} \in \mathbb{F} \\ \mathbb{V} \vdash \mathit{catm}_1 \longrightarrow_{\mathsf{a}} v_1 \qquad ... \qquad \mathbb{V} \vdash \mathit{catm}_n \longrightarrow_{\mathsf{a}} v_n \\ \mathbb{F} \vdash \mathbb{V} \cup \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\}; \mathit{cexp}_{\mathit{body}} \Downarrow_{\mathsf{c}} v \end{array}}{\mathbb{F} \vdash \mathbb{V}; f\ \mathit{catm}_1\ ...\ \mathit{catm}_n \Downarrow_{\mathsf{c}} v} \ \text{CPU-FUNC-APP}$$

$$\frac{\begin{array}{c} \delta(\mathbf{prim}_p, v_1, ..., v_n) = v_{\mathit{out}} \\ \mathbb{V} \vdash \mathit{catm}_1 \longrightarrow_{\mathsf{a}} v_1 \qquad ... \qquad \mathbb{V} \vdash \mathit{catm}_n \longrightarrow_{\mathsf{a}} v_n \end{array}}{\mathbb{F} \vdash \mathbb{V}; \mathbf{prim}_p\ \mathit{catm}_1\ ...\ \mathit{catm}_n \Downarrow_{\mathsf{c}} v_{\mathit{out}}} \ \text{CPU-PRIM}$$

For brevity, the definition of the δ function, which performs evaluation of each basic arithmetic operations such as addition ($\mathbf{prim}_+$), subtraction, and so on, is omitted as it is uninteresting from the perspective of language semantics and corresponds directly to operators built into the CUDA language.

$$\frac{\mathbb{V} \vdash \mathit{catm}_1 \longrightarrow_{\mathsf{a}} v_1 \qquad ... \qquad \mathbb{V} \vdash \mathit{catm}_n \longrightarrow_{\mathsf{a}} v_n}{\mathbb{F} \vdash \mathbb{V}; \mathbf{tuple}\ \mathit{catm}_1\ ...\ \mathit{catm}_n \Downarrow_{\mathsf{c}} \langle v_1, ..., v_n \rangle} \ \text{CPU-TUPLE}$$

$$\frac{\begin{array}{c} \mathbf{kernel}\ k\ x_1\ ...\ x_n\ \mathit{kexp}_{\mathit{body}} \in \mathbb{F} \\ \mathbb{V} \vdash \mathit{catm}_1 \longrightarrow_{\mathsf{a}} v_1 \qquad ... \qquad \mathbb{V} \vdash \mathit{catm}_n \longrightarrow_{\mathsf{a}} v_n \\ \mathbb{V} \vdash [x_1, ..., x_n := v_1, ..., x_n]\mathit{kexp}_{\mathit{body}} \Downarrow_{\mathsf{k}} v \end{array}}{\mathbb{F} \vdash \mathbb{V}; \mathbf{run}\ k\ \mathit{catm}_1\ ...\ \mathit{catm}_n \Downarrow_{\mathsf{c}} v} \ \text{CPU-RUN}$$

$$\frac{\mathbb{V}(x) = v}{\mathbb{V} \vdash x \longrightarrow_{\mathsf{a}} v} \ \text{CPU-ATM-VAR} \qquad \frac{}{\mathbb{V} \vdash v \longrightarrow_{\mathsf{a}} v} \ \text{CPU-ATM-VAL}$$

Kernel expressions:

$$\frac{\mathbb{V} \vdash \mathit{soac}\ \mathit{arg}_1\ ...\ \mathit{arg}_m \Downarrow_{\mathsf{s}} \langle v_1, ..., v_n \rangle \qquad \mathbb{V} \cup \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\} \vdash \mathit{kexp} \Downarrow_{\mathsf{k}} v}{\mathbb{V} \vdash \mathbf{let}\ x_1\ ...\ x_n = \mathit{soac}\ \mathit{arg}_1\ ...\ \mathit{arg}_m\ \mathit{kexp} \Downarrow_{\mathsf{k}} v} \ \text{KERN-LET}$$

$$\frac{\mathbb{V}(x_1) = v_1 \qquad ... \qquad \mathbb{V}(x_n) = v_n}{\mathbb{V} \vdash \mathbf{ret}\ x_1\ ...\ x_n \Downarrow_{\mathsf{k}} \langle v_1, ..., v_n \rangle} \ \text{KERN-RET}$$

SOAC applications:

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

$$\frac{\mathbb{V} \vdash [x := i] gexp \Downarrow_g \langle v_1, ..., v_n \rangle}{\mathbb{V} \vdash \{x \Rightarrow gexp \textbf{ using } xs_1, ..., xs_m\}(i) \Downarrow_{\mathsf{t}} \langle v_1, ..., v_n \rangle} \text{ IDX-THUNK}$$

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

$$\frac{\mathbb{V} \vdash \Lambda(0) \Downarrow_{\mathsf{t}} \langle v_1, ..., v_n \rangle}{\mathbb{V} \vdash \textbf{ONCE } \Lambda \Downarrow_{\mathsf{s}} \langle v_1, ..., v_n \rangle} \text{ SOAC-ONCE}$$

MAP applies its thunk to each index in its index space, computing a tuple of arrays.

$$\frac{\begin{array}{c}(\forall_{0 \le i < s})(\mathbb{V} \vdash \Lambda(i) \Downarrow_{\mathsf{t}} \langle v^i_1, ..., v^i_n \rangle) \\ vs_k \equiv \left[v^i_k \;\middle|\; 0 \le i < s \right]\end{array}}{\mathbb{V} \vdash \textbf{MAP } \Lambda \; s \Downarrow_{\mathsf{s}} \langle vs_1, ..., vs_n \rangle_s} \text{ SOAC-MAP}$$

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

$$\frac{\begin{array}{c}(\forall_{0 \le i < s})(\mathbb{V} \vdash \Lambda_1(i) \Downarrow_{\mathsf{t}} \langle v^i_1, ..., v^i_n \rangle) \\ (\forall_{0 \le i < s})(\mathbb{V} \vdash \Lambda_2(i) \Downarrow_{\mathsf{t}} \langle idx^i \rangle) \\ vs_k \equiv \left[v^{idx^i}_k \;\middle|\; 0 \le i < s \right]\end{array}}{\mathbb{V} \vdash \textbf{PERMUTE } \Lambda_1 \; \Lambda_2 \; s \Downarrow_{\mathsf{s}} \langle vs_1, ..., vs_n \rangle_s} \text{ SOAC-PERMUTE}$$

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/or base values, where each element in the tuple is computed by folding the reduction operator in corresponding position over the array, or simply computing the array itself for the **id** reduction operator.

$$\frac{\begin{array}{ll}(\forall_{0\le i<s}) & \\ (\mathbb{V} \vdash \varLambda_1(i) \Downarrow_{\mathsf{t}} \langle v_1^i, ..., v_n^i \rangle) & vs_k \equiv \begin{cases} \left[v_k^i \;\middle|\; 0 \le i < s\right] & \text{if } \mathit{rop}_k = \mathbf{id} \\ \displaystyle\mathop{\mathcal{R}_{\mathit{rop}_k}}_{i=0}^{s-1} v_k^i & \text{otherwise} \end{cases}\end{array}}{\mathbb{V} \vdash \mathbf{REDUCE}\ \varLambda_1\ (\mathit{rop}_1, ..., \mathit{rop}_n)\ s \Downarrow_{\mathsf{s}} \langle vs_1, ..., vs_n \rangle_s}\ \text{SOAC-REDUCE}$$

The application of a reduction operator to a sequence is denoted with the $\mathcal{R}$ symbol, which behaves similarly to summation or product notation. For example,

$$\mathop{\mathcal{R}_{\mathit{rop}}}_{i=a}^{b} f(i) = f(a)\ \mathit{rop}\ f(a+1)\ \mathit{rop}\ ...\ \mathit{rop}\ f(b-1)\ \mathit{rop}\ f(b)$$

$$\mathop{\mathcal{R}_{\mathbf{iadd}}}_{i=a}^{b} f(i) = \sum_{i=a}^{b} f(i) = f(a) + f(a+1) + ... + f(b-1) + f(b)$$

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.

$$\frac{\begin{array}{ll}(\forall_{0\le i<s}) & \\ (\mathbb{V} \vdash \varLambda_1(i) \Downarrow_{\mathsf{t}} \langle v_1^i, ..., v_n^i \rangle) & vs_k \equiv \begin{cases} \left[v_k^i \;\middle|\; 0 \le i < s\right] & \text{if } \mathit{rop}_k = \mathbf{id} \\ \left[\displaystyle\mathop{\mathcal{R}_{\mathit{rop}_k}}_{i=0}^{j} v_k^i \;\middle|\; 0 \le j < s\right] & \text{otherwise} \end{cases}\end{array}}{\mathbb{V} \vdash \mathbf{SCAN}\ \varLambda_1\ (\mathit{rop}_1, ..., \mathit{rop}_n)\ s \Downarrow_{\mathsf{s}} \langle vs_1, ..., vs_n \rangle_s}\ \text{SOAC-SCAN}$$

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

$$\frac{\begin{array}{c}(\forall_{0\le i<s}) \left(\mathbb{V} \vdash \varLambda_1(i) \Downarrow_{\mathsf{t}} \langle v_1^i, ..., v_n^i \rangle \qquad \mathbb{V} \vdash \varLambda_2(\langle v_1^i, ..., v_n^i \rangle) \Downarrow_{\mathsf{t}} \langle p^i \rangle\right) \\ u_i \equiv \displaystyle\sum_{j=0}^{i} [p^j] \qquad o_i \equiv \min_{u_j = i} j \qquad vs_k \equiv \left[v_k^{o_i} \;\middle|\; 0 \le i < u_s\right]\end{array}}{\mathbb{V} \vdash \mathbf{FILTER}\ \varLambda_1\ \varLambda_2\ s \Downarrow_{\mathsf{s}} \langle vs_1, ..., vs_n \rangle_{u_s}}\ \text{SOAC-FILTER}$$

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.

$$\frac{\begin{array}{c}(\forall_{0\le i<s}) \left(\mathbb{V} \vdash \varLambda_1(i) \Downarrow_{\mathsf{t}} \langle v_1^i, ..., v_n^i \rangle \wedge \mathbb{V} \vdash \varLambda_2(\langle v_1^i, ..., v_n^i \rangle) \Downarrow_{\mathsf{t}} \langle p^i \rangle\right) \\ u_i \equiv \displaystyle\sum_{j=0}^{i} [p^j] \qquad o_i \equiv \min_{u_j = i} j \\ vs_{\mathsf{t},k} \equiv \left[v_k^{o_i} \;\middle|\; 0 \le i < u_s\right] \qquad vs_{\mathsf{f},k} \equiv \left[v_k^{i-o_i} \;\middle|\; 0 \le i < s - u_s\right]\end{array}}{\mathbb{V} \vdash \mathbf{PARTITION}\ \varLambda_1\ \varLambda_2\ s \Downarrow_{\mathsf{s}} \langle vs_{\mathsf{t},1}, ..., vs_{\mathsf{t},n} \rangle_{u_s} \langle vs_{\mathsf{f},1}, ..., vs_{\mathsf{f},n} \rangle_{s-u_s}}\ \text{SOAC-PARTITION}$$

GPU scalar expressions:

$\Downarrow_g$ is a big-step semantic relation capturing reduction on a single GPU thread to produce a tuple of scalars:

$$\mathbb{V} \vdash gexp \Downarrow_g v$$

$$\frac{\mathbb{V} \vdash gexp_{rhs} \Downarrow_g \langle v_1, ..., v_n \rangle \qquad \mathbb{V} \cup \{x_1 \mapsto v_1, ..., x_n \mapsto v_n\} \vdash gexp_{body} \Downarrow_g v}{\mathbb{V} \vdash \mathbf{let}\ x_1, ..., x_n = gexp_{rhs}\ \mathbf{in}\ gexp_{body} \Downarrow_g v}\ \text{GPU-LET}$$

$$\frac{\mathbb{V} \vdash gatm_{cond} \longrightarrow_{\mathsf{a}} \mathbf{true} \qquad gexp_t \Downarrow_g v}{\mathbb{V} \vdash \mathbf{if}\ gatm_{cond}\ \mathbf{then}\ gexp_t\ \mathbf{else}\ gexp_e \longrightarrow_g v}\ \text{GPU-IF-TRUE}$$

$$\frac{\mathbb{V} \vdash gatm_{cond} \longrightarrow_{\mathsf{a}} \mathbf{false} \qquad gexp_e \Downarrow_g v}{\mathbb{V} \vdash \mathbf{if}\ gatm_{cond}\ \mathbf{then}\ gexp_t\ \mathbf{else}\ gexp_e \longrightarrow_g v}\ \text{GPU-IF-TRUE}$$

$$\frac{\begin{array}{c}\delta(\mathbf{prim}_p, v_1, ..., v_n) = v_{out} \\ \mathbb{V} \vdash gatm_1 \longrightarrow_{\mathsf{a}} v_1 \qquad ... \qquad \mathbb{V} \vdash gatm_n \longrightarrow_{\mathsf{a}} v_n\end{array}}{\mathbb{V} \vdash \mathbf{prim}_p\ gatm_1\ ...\ gatm_n \Downarrow_g v_{out}}\ \text{GPU-PRIM}$$

$$\frac{\mathbb{V} \vdash gatm \longrightarrow_{\mathsf{a}} idx \qquad \mathbb{V}(xs) = vs \qquad vs = [v_i \mid 0 \leq i < |vs|]}{\mathbb{V} \vdash xs[gatm] \Downarrow_g v_{idx}}\ \text{GPU-SUBSCRIPT}$$

6 Translation from VCODE

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

6.1 Elementwise operations

The elementwise operations are implemented directly as parallel maps.

Binary operators The binary operator $\odot : X \times Y \to Z$, is implemented with the kernel:

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

Unary operators The unary operator $\wr : X \to Y$ is implemented with the kernel:

```
kernel K≀ (xs : [X]) -> [Y] {
  let res = MAP { i => ≀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] using flgs, xs, ys } (#xs)
  return res
}
```

RAND

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

6.2 Vector instructions

Scans

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

⊙_SCAN

```
kernel ⊙_SCAN (xs : [S]) -> [S] {
  let res = SCAN { i => xs[i] using xs } (⊙) (#xs)
  return res
}
```

Reductions

Like scans, reductions are defined in terms of a monoid $\langle S, \odot, \mathbf{I}\rangle$.

⊙_REDUCE

```
kernel ⊙_REDUCE (xs : [S]) -> S {
  let res = REDUCE { i => xs[i] using xs } (⊙) (#xs)
  return res
}
```

Permutations

VCODE defines a number of permutation operations, including ones that mask elements of their input by flags before applying the permutation and which permute into an array of default values. Some of these operators require convoluted translations into λ_{cu}, which indicates a possible need for more expressiveness in the representation. For example, it should be possible to efficiently take flags into account while permuting without running a separate filter operation.

PERMUTE

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

DPERMUTE

```
kernel DPERMUTE (xs : [T], ix : [int], default: [T]) -> [T] {
  let (vals, wrotes) = PERMUTE
    { i => (if i < #xs then xs[i] else xs[0], i < #xs) using xs }
    { i => if i < #ys then ys[i] else 0 using ys } (#default)
  let res = MAP
    { i => if wrotes[i] then vals[i] else default[i] using wrotes, vals, default }
    (#default)
  return res
}
```

FPERMUTE

```
kernel FPERMUTE (xs : [T], ix : [int], flgs: [bool]) -> [T] {
  let (filt_xs, filt_ix) = FILTER { i => (xs[i], ix[i]) using xs, ix }
    { i => flgs[i] using flgs } (#xs)
  let res = PERMUTE { i => filt_xs[i] using filt_xs }
    { i => filt_ix[i] using filt_ix }
    (#filt_ix)
  return res
}
```

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

```
kernel DFPERMUTE (xs : [T], ix : [int], flgs: [bool], default: [T]) -> [T] {
  let (filt_xs, filt_ix) = FILTER { i => (xs[i], ix[i]) using xs, ix }
    { i => flgs[i] using flgs }
    (#xs)
  let (vals, wrotes) = PERMUTE
    { i => (if i < #filt_xs then filt_xs[i] else filt_xs[0], i < #filt_xs)
      using filt_xs }
    { i => if i < #ys then ys[i] else 0 using ys } (#default)
  let res = MAP { i => if wrotes[i] then vals[i] else default[i] } (#default)
  return res
}
```

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
}
```

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 λ_{cu}, it is not necessary to produce the PACK operation for conditional operations because the elements can be separated with **PARTITION** over the predicate, but **FILTER** 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 => tmp[i].2 using tmp } (#tmp)
  return res
}
```

7 Loop fusion

Fusion of array-processing kernels is the primary optimization performed on λ_{cu}. Combining iterations over arrays results in performance benefits as described in §3, motivating the fusion of these operations wherever possible. In general, programs may contain many array operations, but these operations do not all occur in a single sequence and on arrays of the same size. To the contrary, array operations are frequently separated by control flow and often operate on arrays of differing sizes. Nessie simplifies the problem of selecting operations to fuse by focusing on sections of code without intervening control flow, referred to as *control regions*. In many cases, however, even within a single control region, not all array operations can be fused into a single efficient CUDA kernel. For example, two array operations may have incompatible memory access patterns or have their execution serialized if one requires the result of the other to first be processed by a third, unfusible operation. In addition, it may be possible to fuse non-disjoint 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:

7.1 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 top-down on a dataflow graph: start with a node that is a producer of some data consumed below, but not itself a consumer of any other computation. Fuse consumers into producers at each possible opportunity, moving down the dataflow graph in a breadth-first fashion. This strategy is the approach taken in a previous incarnation of this work [BR12], but misses some fusions that would be beneficial. This traversal order can "shoot itself in the foot" by choosing a weakly beneficial fusion opportunity if it is evaluated before one that this choice precludes, but that would be more beneficial if selected.

In the opposite direction, operations can be fused starting from leaves of the graph, which in some cases finds more opportunities to fuse than the downward approach. Operations further down the dataflow graph have fewer descendant nodes which might consume their outputs, lessening potential interference (precluding fewer other fusions) when fusing them with nearby nodes.

Both approaches succumb to the classic pitfall of greediness– sensitivity to traversal order over ultimate benefit. By examining only small neighborhoods of the graph at once, relatively insignificant fusion opportunities can be selected even if they conflict with significant ones, if the latter are not evaluated first.

7.2 ILP-based scheduling

A straightforward strategy that prioritizes the most beneficial fusions can be achieved through integer linear programming (ILP). The relative benefits of fusion opportunities can be encoded into weights and conflicts between fusions into constraints, giving an ILP instance. This approach allows expressing preferences on fusion plans but outsources the difficult combinatorial search 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 of the top-level program 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 based on the dataflow graph of that control region.

7.3 Instance construction

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

$$\texttt{let}\ (d_1, d_2, ...) = \texttt{run}\ \mathrm{k}\ u_1\ u_2\ ...$$

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), the list of variables it reads as arguments, and the list of variables it defines. This permits the notation $I = \langle k, \overline{u}, \overline{d} \rangle$ for an invocation I of kernel k, using variables $\overline{u}$, and defining variables $\overline{d}$. For convenience, $d(\cdot)$, $u(\cdot)$ and $k(\cdot)$ denote projections of the corresponding component of a kernel invocation, *e.g.*, $k(\langle k, \overline{u}, \overline{d} \rangle) = k$.

Each ILP instance describes the fusion problem for a given sequence of kernel invocations by defining variables $o_i \in \mathbb{N}$ specifying the sequential ordering of kernel invocation I_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: $\forall_{i,j}\ I_i \overset{a}{\multimap} \implies c_a \geq f_{i,j} \times [\xrightarrow{a} I_j]$.

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

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 consuming a given array can be fused into the kernel invocation that produces the array, then the array computation may be fused 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 that 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 multipled 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, to enforce the priorities of optimization goals (minimizing allocation, memory traffic, and number of invocations, in descending order). Weighting is achieved by computing an integer n that is guaranteed to be larger than any of the summations, then multiplying the more important summations by increasing powers of n:

$$n^2 \sum_{i,j} f_{i,j} + n \sum_{i,j} f_{i,j} sr(i,j) + \sum_a c_a$$

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.

The $\multimapinv$ relation holds between a kernel invocation $I = \langle K, \overline{u}, \overline{d} \rangle$ and the variables it binds:

$$I \overset{v}{\multimapinv} \iff v \in d(I)$$

Similarly, the $\rightarrow$ relation specifies the variables read by a particular kernel invocation:

$$\xrightarrow{v} I \iff v \in u(I)$$

The $\multimapinv\!\!\rightarrow$ relation and the family of per-variable $\overset{v}{\multimapinv\!\!\rightarrow}$ relations capture direct data dependencies; given kernel invocations I_1 and I_2,

$$\frac{I_1 \overset{v}{\multimapinv} \qquad \xrightarrow{v} I_2}{I_1 \overset{v}{\multimapinv\!\!\rightarrow} I_2}$$

$$I_1 \multimapinv\!\!\rightarrow I_2 \iff \exists v. I_1 \overset{v}{\multimapinv\!\!\rightarrow} I_2$$

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 $\multimapinv\!\!\rightarrow$ relation) are considered to determine whether edges may be fused.

The $\cong_{\downarrow}$ and $\cong_{\leftrightarrow}$ relations capture horizontal and vertical compatibility between kernels and are defined based on the SOAC applications inside the kernels. The complement of these relations are written $\ncong_{\downarrow}$ and $\ncong_{\leftrightarrow}$, respectively. The symmetric $\cong_{\leftrightarrow}$ relation describes the potential for horizontal fusion between all pairs of matching SOACs, and forbids horizontal fusion of distinct SOACs. Vertical fusion may occur in more interesting pairings of SOAC, and compatibility for vertical fusion is asymmetric: $\cong_{\downarrow}$ holds between `MAP` flowing into any other SOAC (excluding `ONCE`), between `FILTER`s, between `REDUCE` and subsequent `ONCE`, and between `FILTER` or `PERMUTE` flowing into `REDUCE`.

Vertical fusion constraints

The constraints on o and f for vertical fusion simply describe the impossibility of fusing vertically incompatible SOACs and the necessity of avoiding dataflow-incompatible reordering of kernel invocations. These can be written as follows:

$$\frac{I_1 \multimapinv\!\!\rightarrow I_2 \qquad k(I_1) \ncong_{\downarrow} k(I_2)}{f_{1,2} = 1} \text{ VFUSE-COMPAT}$$

$$\frac{I_1 \mathrel{\circ\!\!\longrightarrow} I_2}{o_1 \leq o_2 + f_{1,2}} \ \textsc{vfuse-cluster}$$

Horizontal fusion constraints

A notion of dataflow independence of kernel invocations (denoted $\downdownarrows$) is necessary to define the conditions for horizontal fusion. This symmetric relation holds for two kernel invocations if there is no (even indirect) dataflow in either direction between them; neither reads data influenced by the other. Dataflow paths may involve scalar code which is not expressed as run statements, so this relation considers more possible avenues for dataflow than simply relying on the negation of the transitive closure of the $\mathrel{\circ\!\!\longrightarrow}$ relation would. It can nonetheless be computed easily from a program dependence graph.

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

$$\frac{I_1 \downdownarrows I_2 \qquad k(I_1) \ncong_{\leftrightarrow} k(I_2)}{f_{1,2} = 1} \ \textsc{hfuse-compat}$$

When two dataflow-independent nodes I_1, I_2 are "siblings" (share an input), they may be fused if their kernels are compatible, denoted $I_1 \curlywedge I_2$. alone is not sufficient to conclude any particular value of o or f for these kernel invocations. Fusion compatibility is also dependent on the size of the index space. Ideally sizes would be directly represented in λ_{cu}, but the current implementation uses shared inputs as a conservative proxy for size information. It is important to note that not all inputs to a kernel invocation are guaranteed to be the size of the iteration space, so only the subset of kernel inputs which are iterated over is considered when using inputs to determine size compatibility.

$$\frac{I_1 \downdownarrows I_2 \qquad \xrightarrow{v} I_1 \qquad \xrightarrow{v} I_2}{I_1 \curlywedge I_2} \ \textsc{hfuse-sibs}$$

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.

$$\frac{\begin{array}{c} I_1 \downdownarrows I_2 \qquad I_1 \not\curlywedge I_2 \qquad k(I_1) \cong_{\leftrightarrow} k(I_2) \\ allPairs = |\{I_{p1} : I_{p1} \mathrel{\circ\!\!\longrightarrow} I_1\}| * |\{I_{p2} : I_{p2} \mathrel{\circ\!\!\longrightarrow} I_2\}| \\ unfusedPairs = \displaystyle\sum_{\substack{I_{p1} \mathrel{\circ\!\!\longrightarrow} I_1 \\ I_{p2} \mathrel{\circ\!\!\longrightarrow} I_2}} f_{p1,p2} \end{array}}{1 - f_{1,2} \leq allPairs - unfusedPairs} \ \textsc{hfuse-preds}$$

7.4 Recovering clusters

Once the ILP instance is constructed, it is passed off to an off-the-shelf solver, which computes an assignment of values to the indexed sets of o, f, and c variables. Kernel invocations are grouped into equivalence classes based on equality of the corresponding values of o. These equivalence classes give a partition of the kernel invocations into clusters, where each cluster will be rewritten into a single kernel in the fused program.

8 Fusion rewriting

Given a partition of the kernel invocations in a dataflow region, it then remains to rewrite the kernel invocations in each cluster into 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 the kernel invocations grouped into each cluster do not exhibit fusion incompatibilities.

8.1 Rewrite existence

It is not obvious without substantiation 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. The problem of how to fuse kernel invocation pairs which are separated in the dataflow graph by scalar computations can be overcome by moving the scalar computations onto the GPU. This could be achieved by copying the scalar computation into the thunk of the SOAC, and accordingly running a copy of it on each GPU thread (increasing the total amount of work performed), or by modifying code generation to perform these operations in a single thread and broadcast the results, as would occur if the scalar computation were simply lifted into the thunk of an application of the **ONCE** SOAC. Because it simplifies the definition of fusion to consider a dataflow graph consisting of only kernel invocations, scalar computations are assumed to be handled by lifting into the thunk of a **ONCE** SOAC application.

Now that a program region can be viewed as strictly a dataflow graph between kernel invocations, an approach for fusing these invocations is needed. 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.

8.2 Fusing kernel invocation pairs

Since a kernel invocation is simply a λ_{cu} `let` statement applying a kernel, the meaning of a kernel invocation is simply binding $vars$ to the results of passing $args$ to $kern$. From the perspective of the rest of the program, 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 combine the SOAC applications within kernels and remove intermediate array allocations.

Operationally, a kernel invocation can be seen as allocating space its results and then passing these allocations to the kernel. To support removal of allocations, fused kernel invocations must be rewritten to a kernel invocation which binds a subset of the union of the bound sets of variables from the invocations fused into it.

To optimize out exactly the arrays that a given fusion makes unnecessary, fusing kernel invocations I_1 and I_2 (where data may flow from I_1 to I_2 but not vice-versa) should result in a kernel invocation binding the union of I_1's and I_2's bound variables, minus any variables referenced only by I_2:

$$\frac{\nexists v. I_2 \overset{v}{\hookrightarrow} I_1 \qquad vs = (d(I_1) \cap u(I_2)) \setminus \bigcup_{i \neq 2} u(I_i)}{d(I_1 \oplus I_2) = (d(I_1) \cup d(I_2)) \setminus vs}$$

SOAC fusion

The kernels produced by lowering from NESL are constructed to apply a single SOAC each, and the fusion compatibility relation constrains clusters to containing SOACs with compatible cross-thread communication patterns and index spaces. Thus fusing two kernels involves combining their SOAC applications into a single one.

The resulting SOAC application must traverse the same index space as its constituent applications and compute the combined results of the two at each point. However, different pairs of SOACs combine in different ways. For example, the thunk of a **MAP** can be folded into the pull thunk of another SOAC, but horizontal fusion of two **REDUCE** kernel invocations needs to combine their reduction operators as well as build a pull thunk that will produce the arrays produced by both kernels invocations' thunks. The process of combining two thunks into a single one accepting the parameters of both original thunks and producing merged outputs is referred to as *thunk combination*.

Thunk combination

Thunk combination is denoted with the ‡ 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 as a subscript to the operator (*e.g.*, $\ddagger_A$). Thunk combination operates syntactically given two thunks and A, the list of array variables bound by the first thunk:

Let e_1 and e_2 be GPU-level expressions, xs and ys be sets of variables, where $A_j \in ys$. Let $a_1, a_2, ...$ be fresh variables that do not appear in e_1 or e_2, of the same number as the variables in the tail **ret** expression of e_1, and similarly for $b_1, b_2, ...$ and e_2. Then,

$$\{\ i \texttt{ => } e_1 \texttt{ using } xs\ \} \ddagger_A \{\ i \texttt{ => } e_2 \texttt{ using } ys\ \} =$$

$$\{\ i \texttt{ => let } a_1, a_2, ... \texttt{ = } e_1 \texttt{ in let } b_1, b_2, ... \texttt{ = } e_2{}' \texttt{ in ret } a_1, a_2, ..., b_1, b_2, ... \texttt{ using } xys\ \}$$

where $e_2{}'$ is e_2 with all occurrences of $A_j[i]$ replaced by a_j, and $xys = xs \ \cup \ ys \setminus A$.

This has the effect of substituting the scalar components of array variables computed by the left-hand thunk into the right-hand one, and allows arrays which are no longer referenced after fusion to be removed by simply performing dead-variable elimination.

8.3 SOAC fusion rules

The rules to combine each fusible pair of SOACs applications are presented below, using the notation $vars_1 = soac_1\ args_1 \oplus vars_2 = soac_2\ args_2$. For all rules, the variables in A are assumed to be unused outside of the SOAC which binds the list of variables B, so that unused variable elimination removes them.

$$\frac{A \texttt{ = MAP } \Lambda_1\ n \qquad \oplus \qquad B \texttt{ = MAP } \Lambda_2\ n}{B \texttt{ = MAP } (\Lambda_1 \ddagger_A \Lambda_2)\ n}\ \textsc{map-map}$$

Fusion of two **MAP**s simply combines their pull thunks.

$$\frac{A \texttt{ = MAP } \Lambda_1\ n \qquad \oplus \qquad B \texttt{ = PERMUTE } \Lambda_2\ \Lambda_3\ n}{B \texttt{ = PERMUTE } (\Lambda_1 \ddagger_A \Lambda_2)\ \Lambda_3\ n}\ \textsc{map-permute}$$

$$\frac{A \texttt{ = MAP } \Lambda_1\ n \qquad \oplus \qquad B \texttt{ = REDUCE } \Lambda_2\ id_r\ \Lambda\ n}{B \texttt{ = REDUCE } (\Lambda_1 \ddagger_A \Lambda_2)\ id_r\ \Lambda\ n}\ \textsc{map-reduce}$$

$$\frac{A \texttt{ = MAP } \Lambda_1\ n \qquad \oplus \qquad B \texttt{ = SCAN } \Lambda_2\ id_r\ n}{B \texttt{ = SCAN } (\Lambda_1 \ddagger_A \Lambda_2)\ id_r\ n}\ \textsc{map-scan}$$

$$\frac{A \texttt{ = MAP } \Lambda_1\ n \qquad \oplus \qquad B \texttt{ = FILTER } \Lambda_2\ \Lambda_3\ n}{B \texttt{ = FILTER } (\Lambda_1 \ddagger_A \Lambda_2)\ \Lambda_3\ n}\ \textsc{map-filter}$$

$$\frac{A \texttt{ = MAP } \Lambda_1\ n \qquad \oplus \qquad B \texttt{ = PARTITION } \Lambda_2\ \Lambda_3\ n}{B \texttt{ = PARTITION } (\Lambda_1 \ddagger_A \Lambda_2)\ \Lambda_3\ n}\ \textsc{map-partition}$$

Fusion of **MAP** with another SOAC (**PERMUTE**, **REDUCE**, **FILTER**, **SCAN**, or **PARTITION**) merges the former into the latter's pull thunk.

$$\frac{\begin{array}{c} A \texttt{ = FILTER } \Lambda_1\ \{\ v \texttt{ => } p_1 \texttt{ using } ys\ \}\ n \\ \oplus \\ B \texttt{ = FILTER } \{\ i \texttt{ => } A[i] \texttt{ using } A\ \}\ \{\ v \texttt{ => } p_2 \texttt{ using } ws\ \}\ (\#A) \end{array}}{\begin{array}{c} B \texttt{ = FILTER } \Lambda_1 \\ \{\ i \texttt{ => let } a_1, a_2, ... \texttt{ = } body(\Lambda_1) \texttt{ in } p_1\ \&\&\ p_2{}' \texttt{ using } ys \cup (ws \setminus A)\ \}\ n \end{array}}\ \textsc{Filter-Filter}$$

where $p_2{}'$ is p_2 with all occurrences of $A_j[i]$ replaced by a_j.

Fusion of two **FILTER**s, where the result of the first is only consumed by the second, results in a single predicate composed of the conjunction of the **FILTER**s' predicates.

$$\frac{\begin{array}{c} A \texttt{ = } \mathbf{FILTER}\ \{\ i \texttt{ => } e_1\ \mathbf{using}\ xs\ \}\ \{\ w_1, w_2, \ldots \texttt{ => } p\ \mathbf{using}\ ys\ \}\ n \\ \oplus \\ B \texttt{ = } \mathbf{REDUCE}\ \{\ i \texttt{ => } e_2\ \mathbf{using}\ zs\ \}\ id_r\ (\#A) \end{array}}{\begin{array}{c} B \texttt{ = } \mathbf{REDUCE}\ \{\ i \texttt{ => } \mathbf{let}\ a_1, a_2, \ldots \texttt{ = } e_1\ \mathbf{in}\ \mathbf{if}\ p'\ \mathbf{then}\ e_2{}'\ \mathbf{else}\ id_r \} \\ xs \cup (ys \cup zs) \setminus A\ id_r\ n \end{array}}\ \textsc{Filter-Reduce}$$

where $e_2{}'$ is e_2 with all occurrences of $A_j[i]$ replaced by a_j, and p' is p with all occurrences of w_j replaced by a_j.

A **FILTER** into a **REDUCE** is fused by substituting the identity element of the **REDUCE**'s reduction operators for elements which would be filtered out.

9 Code generation

After fusion, λ_{cu} programs are lowered to CUDA C++ and passed to the NVIDIA `nvcc` compiler.

The top-level code generator accepts a λ_{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 CUDA kernel function for each of the kernels in the λ_{cu} program, as well as a top-level C++ function definition for each function definition in the λ_{cu} program. Lowering for code other than kernels and **run** statements is unremarkable: λ_{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. The dimension constraints imply using block sizes no larger than those expected by the lowerings of scans and reductions, which rely on block-level synchronization.

9.1 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 λ_{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.

10 Solver performance

A drawback of relying on ILP to decide which kernels to fuse is the computational difficulty of ILP itself: integer programming is NP-hard. As such, the worst-case runtime of fusion scheduling scales exponentially with dataflow region size, potentially leading to impractical compilation times.

To evaluate the practical impact of solver time complexity, a number of sample programs in λ_{cu} were generated to assess the time taken to perform fusion scheduling on them via ILP. The ILP instances generated for a given NESL program depend on the dataflow and SOACs applied in individual control-flow regions, but disparate control-flow regions do not influence each other. Therefore it is sufficient to generate programs with a single control-flow region, and only vary the dataflow within that region.

Programs were generated with varying numbers of kernel invocations and with varying lengths of dataflow dependency chains, providing opportunities for horizontal and vertical fusion, as well as dataflow constraints.

The time taken by the solver is shown in Figure 7. For control-flow regions with fewer than 30 kernels, solver runtime was consistently below one second, with runtime growing exponentially in the number of kernels as expected. Notably, although there is variation in runtime as the dependency structure of the control-flow region varies, there is not a strong relationship between the average length of dataflow chains and solver runtime. The strong dependence of solver runtime on program size implies that it should be possible to predict which programs will be difficult to schedule. These programs may admit alternate approaches to employing ILP, such as computating kernel schedules offline to reuse across compiler runs, artificially splitting of large control-flow regions, or allowing programmers to suggest simplifying assumptions for the generated ILP instances.

If the runtime of the ILP solver becomes a constraint on the practical compilation of real-world programs, there are a number of additional potential avenues for speeding up ILP solving: it may be possible to eliminate redundant constraints prior to running the solver, or it may help to express both separateness and ordering of kernels with a single set of variables, rather than having both sets and implicitly tying the two together via ordering constraints. In addition, the current solver does not take full advantage of multi-core systems or GPUs, so Nessie itself could potentially be used to write a faster, parallel ILP solver.

It is also possible to perform some fusion optimistically in the compiler frontend, for example in simple single-producer, single-consumer situations where fusion is always permitted and advantageous. The resulting reduction in number of kernels will improve solver performance when possible.

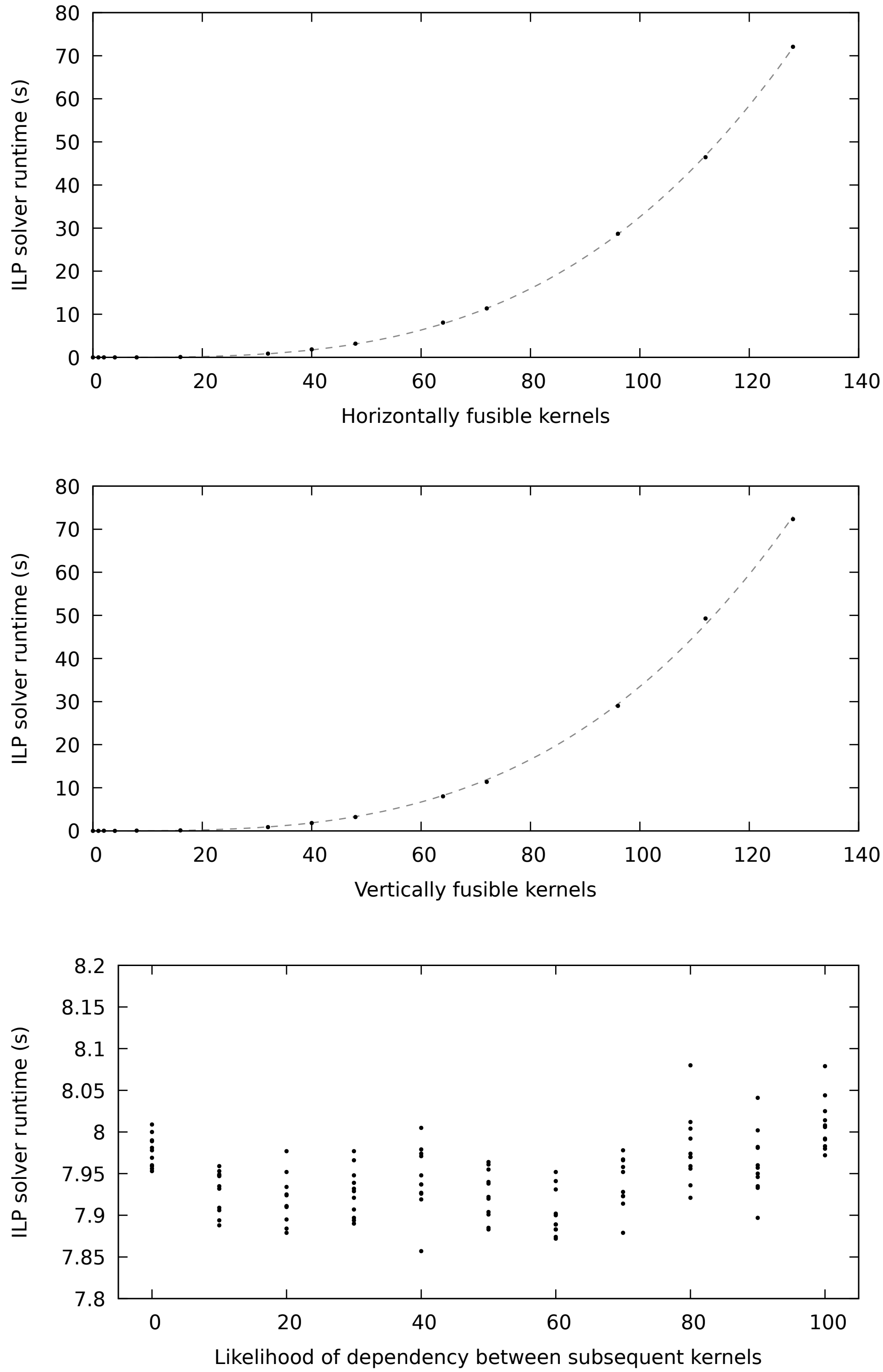

Fig. 7: Solver runtimes for single control-flow regions invoking single-result kernels. Exponential scaling is observed as the number of kernels increases, but the length of data dependency chains is not strongly related to solver runtime.

11 Results

Nessie was used to compile a number of NESL programs, which were benchmarked against implementations in hand-written CUDA from NVIDIA's CUDA Samples. Where possible, the performance of Nessie's predecessor NESL/GPU was also measured to verify that Nessie is an improvement over previous efforts in compiling NESL to CUDA.

The first benchmark program computes call and put prices for stock options via the Black-Scholes equation. It exhibits flat parallelism as each stock is processed independently; computing prices for a stock involves sequential arithmetic computation and branches.

The output from Nessie is compared against the implementation from NVIDIA's CUDA Samples, modified to use precise math instead of the approximate operations from the original code. Nessie does not use output CUDA intrinsics for approximate floating-point operations because because NESL does not include them in the source language and replacing indiscriminately would change program semantics; therefore, the NVIDIA implementation was altered so both programs compute the same results.

The hand-rolled CUDA implementation of Black-Scholes is slightly faster than Nessie's output, but the two scale similarly. Performance differences between the two are due to memory access coalescing, which Nessie does not yet perform, as well as overhead from the iteration structure used by Nessie's lowering of the **MAP** SOAC. The NESL/GPU implementation of the Black-Scholes simulation did not compile successfully, so it was not compared.

The second program is a dot-product computation over two randomly-generated vectors. Nessie's output is compared against a program derived from the `dotProd` program from the NVIDIA CUDA SDK, which uses the cuBLAS linear algebra library.

Again, Nessie runs faster for small array sizes but is slower by a constant multiple as input sizes grow. Nonetheless, both implementations scale similarly with input size and have similar throughput.

The third benchmark is a simulation of gravitational interaction between point bodies in 3-dimensional space.

At each simulation step, each body's gravitational influence on the acceleration of each other body is computed, and their positions and velocities are updated accordingly. This is an $O(n^2)$ algorithm exhibiting nested data parallelism in the nested loops over the set of bodies. In addition, the NESL code for the simulation lowers to a segmented reduction when summing other bodies' contributions to each body's acceleration, testing segmented operations.

Nessie's performance in this benchmark is again faster than NESL/GPU, but is hindered by a failure to fuse away the construction of an n^2-sized array of forces, owing to segment-descriptor operations performed between the initialization and summation of the array. This unfortunate interleaving of segment-descriptor and array computations is owed to the current implementation relying on the NESL/GPU frontend to perform lowering from NESL to its own intermediate representation before conversion to λ_{cu}. An in-progress frontend that

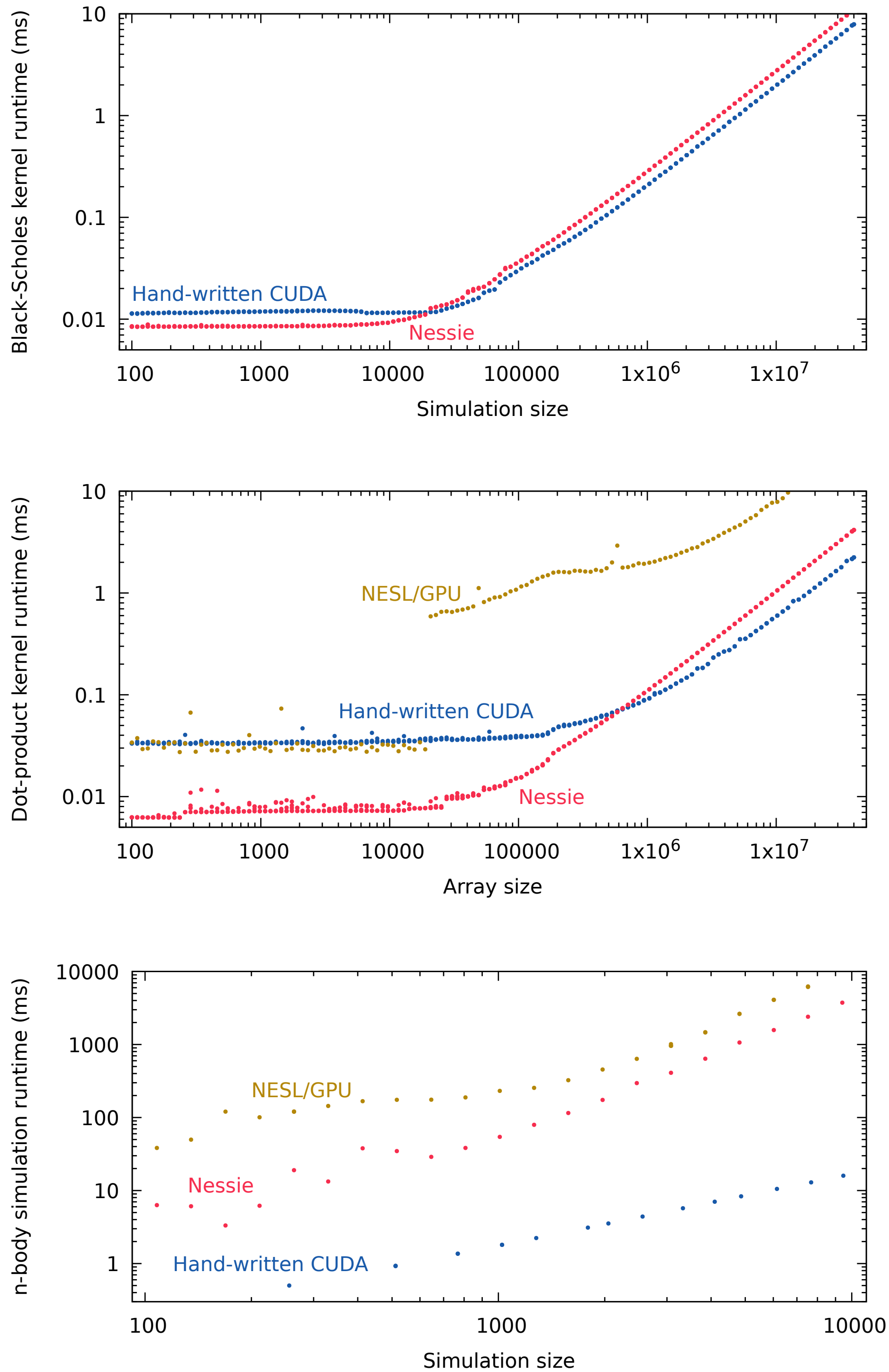

Fig. 8: Kernel execution timings. Each point represents the average of 512 executions of the GPU-side code.

performs flattening differently will avoid this interleaving, bringing Nessie's performance closer to that of hand-written CUDA.

Although they are still preliminary and do not exercise all of Nessie's features, these benchmarks demonstrate the ability to compile high-level NESL programs to be competitive with hand-written CUDA.

12 Related work

λ_{cu} occupies only a single point of the rich design space available to systems for nested data parallelism. Some systems target CPUs or GPUs specifically while others avoid making considerations for particular hardware in order to be more portable. A number of systems support only flat data parallelism, while others handle nested data parallelism; among NDP systems, not all are capable of expressing irregular nesting.

12.1 Conceptual tools for optimizing array operations

A number of disparate approaches have demonstrated their utility for reasoning about computations involving arrays. Nessie employs some of these (such as pull arrays) and could likely benefit from adopting ideas from others.

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 that offer first-class support for arrays often introduce mutability as an optimization. Array SSA [KS98] extends the framework of Single Static Assignment with an analogue of the ϕ function that facilitates reasoning about updates to subindices of arrays, which can help optimize patterns like functional update of array values.

Push and pull arrays

Representing arrays as generators or acceptors of values, termed *pull arrays* and push arrays, respectively [SS14], can enable additional optimizations compared to simply considering them as compound values. λ_{cu} benefits from a pull-based representation of arrays, where thunks are applied to index spaces to generate array elements. Push arrays, which are dual to pull arrays, indicate where to place generated elements in memory, but Nessie instead represents these semantics with SOACs like **`PERMUTE`** and indexing in thunks for **`MAP`**.

Uniqueness types

Reasoning about the usage of variables across control flow can allow for greater memory reuse; uniqueness types [dPA07] are one such static analysis. Typesystems supporting uniqueness types distinguish between values which have multiple references and those which to which only a single reference exists; the latter values can be transparently updated in-place, avoiding reallocation and copying.

12.2 Other systems for NDP

Numerous other systems support nested data parallelism, and a number of these served as inspiration and guidance when implementing Nessie.

ILP-based fusion

Megiddo and Sarkar [MS97] first applied integer linear programming to find fusion clusterings for array programs. Robinson's work on fusing filters [RLK14] applied ILP-based reasoning about fusion in a context where operations on arrays could produce outputs of different sizes than their inputs. Nessie applies similar ILP-based fusion in a setting with nested data parallelism using a flattening transform.

Parallel combinators

Nova [CGG+14] is a functional progamming language exposing parallelism through a set of built-in parallel operations including map, reduce, scan, permute, and filter, which can be used in a nested fashion. Nested arrays are converted to flat arrays along with segment descriptors, as in Nessie. Nova fuses sequences of maps, sequences of filters, and maps inside reductions, but has fewer fusion rules than Nessie and does not do whole-program analysis to decide between alternative fusion schedules.

Explicitly hierarchical parallelism

Instead of applying a fully general flattening transform, Obsidian [CSS12] implements nested data parallelism for GPUs by providing a language that represents the hierarchical structure of parallelism on GPUs in its typesystem. Obsidian exposes push and pull arrays and gives users control over when arrays become manifest in memory. In contrast, Nessie implements the NESL language where semantics treat all arrays as manifest values and works to optimize as many arrays as possible out of compiler output. Push arrays as implemented by Obsidian play an analogous rôle to that of SOAC applications in Nessie: they specify array contents along with iteration structures, and can be fused together into larger iteration structures. Pull arrays are most similar to the application of the **`MAP`** SOAC to a thunk in Nessie: they represent computations of values from indices, and can be fused together with other pull arrays, or be fused into a push array (other SOAC).

HIDP [ZM13] is a language compiling NDP to CUDA that exposes parallelism through map syntax and parallel builtins which may occur standalone or as suffices to map clauses. It performs shape inference and flattening and optimizes segmented descriptors for *regular* arrays by representing them as pairs of scalars tracking segment size and count. HIDP allows users to hint to which level of the GPU hierarchy a construct should map and performs automatic tuning of its GPU code. It represents data as n-dimensional arrays with reference semantics and mutable update. Fusion in HIDP can combine maps for which one operates over a prefix of the other's index space.

Copperhead [CGK11] is an EDSL which converts a subset of python sequence operations into CUDA or parallel C++. Copperhead permits nesting, but relies on explicitly mapping onto the hierarchical organization of the GPU rather than using a flattening transform. Fusion in Copperhead relies on a rough classification of sequence operations into those iterating over input space or output space, with further division of output-space operations into those where ouput elements depend on none, the corresponding, or all elements of their inputs.

Low-level libraries

There are a number of libraries providing parallel primitives, supporting composition into larger parallel programs, or acting as a compilation target for other languages and runtimes.

Thrust [HB11] is a C++ library providing a high-level interface to various parallel algorithms. Thrust has multiple backends including other parallel libraries such as Intel TBB (relying on multithreading on the CPU), CUDA, and OpenMP; it abstracts over the details of the memory and parallelism hierarchies of its backends, but does require user programs to handle the distinction between the main memory space and possibly-separate device memory. Thrust does not provide its own compiler or do extensive transformation, but its "functors" allow explicit fusion of elementwise scalar operations.

Running NESL on GPUs

CuNesl [ZM12] was an effort to compile NESL to CUDA, focusing on rewriting recursive functions including non-tail calls into loops and supporting a variety of mappings between CUDA thread blocks and array segments. The barriers between these execution modes required manual annotation, and CuNesl did not perform significant fusion of array operations.

NESL/GPU [BR12], the predecessor to Nessie, compiled the VCODE language used by the NESL implementation to CUDA, and performed map-map fusion optimizations, but did not make use of pull arrays or other forms of array operation fusion. Programs were still executed by interpreting VCODE on the CPU, with CUDA implementations accelerating certain subroutines. Nessie improves over the previous approach by fully compiling NESL programs to CUDA executables and performing more fusion to minimize a performance heuristic modelling GPU architectures.

Work-efficient flattening

Systems for nested data parallelism have implemented varied approaches to flattening to distribute computations across available computing resources without inducing asymptotic increases in total computational work performed.

Proteus [PPW95] was a language for NDP supporting nested and irregular data parallelism, compiled to a runtime based on CVL, a library for vector operations also used by Blelloch's original NESL implementation. Its authors present a flattening transformation for NDP which avoids increasing work complexity and uses randomized accesses among multiple copies of data to accelerate reads.

Data-Parallel Haskell [CLP+07] implements nested data parallelism atop the GHC compiler for Haskell, based on a set of rewrite rules that perform local fusion. Lippmeier *et al.* [LCK+12] investigated approaches to flattening that avoid asymptotic work increases caused by duplication of arrays.

Data-parallel DSLs

In addition to Data-Parallel Haskell, there are a number of libraries and domain-specific languages for data-parallel computations on the GPU, implemented as extensions to Haskell, other functional languages, or standalone. Each of these exposes different capabilities and tradeoffs compared to Nessie.

Nepal [CKLP01] extends Haskell to support NDP by adding a parallel array datatype with similar operations to NESL. It employs a generalized form of Blelloch-style flattening which supports parallel arrays of user-defined element types [CK00]. Nepal uses the distinction between parallel and sequential types to reason statically about data distribution in a distributed-computation setting, and has fusion rewriting rules on both array operations and commmunication operations.

Repa [LCKJ12] is a Haskell library providing parallel, multi-dimensional arrays for flat parallelism with shape polymorphism. It has notions of delayed and manifest arrays, cursored delayed arrays, and partitioning of arrays, as well as rewrite rules for optimization.

Accelerate [CKL+11] [LGCK09] is a library eDSL which introduces a type-level distinction between CPU and GPU datatypes, employing a caching scheme similar to Nessie's. Accelerate focuses on flat data parallelism under the *scan-vector* model. Like Repa, Accelerate supports shape or rank polymorphism. Accelerate compiles to CUDA by way of a skeleton-based code generator.

Futhark is a domain-specific language designed to support efficient nested data-parallel computations on a variety of backends including GPUs. Futhark employs an algebraic array operation fusion strategy [HO13], and performs shape analysis to fuse across operations with different sizes, as well as additional rewrite rules. Futhark's internal representation includes a novel `redomap` combinator that captures `reduce ∘ map ∘ filter`, whereas Nessie has a specific fusion rule for filter-reduce fusion and maps are fused into other combinators via their own fusion rules. Futhark uses uniqueness types to permit safe in-place updates and performs a flattening transformation similarly to Nessie.

Delite [SBL+14] is an eDSL embedded in Scala which builds ASTs, then performs simplifications based on a model of linear algebra, followed by traditional optimizations, liveness anaylysis, and code generation for C++, Scala, CUDA, and OpenCL. Delite performs local fusion of operations in loops and performs array-of-struct to struct-of-array transformation, but does not have a full flattening transform.

13 Future work

There are a number of improvements planned for future addition to the Nessie compiler.

13.1 Memory management

One goals of λ_{cu} 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].

13.2 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* [JK11] or *traversal splicing* [JK12] .

13.3 Decomposing filters

Nessie currently lowers filter operations into a sequence of three parallel operations: a map of 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.

13.4 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.

13.5 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

[BC90] Blelloch, G. and S. Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, 1990, pp. 471–480.

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

[Ble90] Blelloch, G. E. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

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

[Ble96] Blelloch, G. E. Programming parallel algorithms. *Communications of the ACM*, **39**(3), March 1996, pp. 85–97.

[BR12] Bergstrom, L. and J. Reppy. Nested data-parallelism on the GPU. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functonal Programming*, Copenhagen, Denmark, September 2012. ACM, pp. 247–258.

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

[CGG+14] Collins, A., D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY '14)*, Edinburgh, UK, June 2014. ACM, pp. 8:8–8:13.

[CGK11] Catanzaro, B., M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '11)*, San Antonio, TX, February 2011. ACM, pp. 47–56.

[CK00] Chakravarty, M. M. T. and G. Keller. More types for nested data parallel programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2000, pp. 94–105.

[CKL+11] Chakravarty, M. M., G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the 6th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, Austin, Texas, USA, January 2011. ACM, pp. 3–14.

[CKLP01] Chakravarty, M. M. T., G. Keller, R. Leshchinskiy, and W. Pfannenstiel. Nepal – nested data parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference on Parallel Computing*, vol. 2150 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2001, pp. 524–534.

[CLP+07] Chakravarty, M. M. T., R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. ACM, January 2007, pp. 10–18.

[CSS12] Claessen, K., M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP '12)*, Philadelphia, PA, January 2012. ACM, pp. 21–30.

[dPA07] de Vries, E., R. Plasmeijer, and D. M. Abrahamson. Uniqueness typing simplified. In O. Chitil, Z. Horváth, and V. Zsók (eds.), *19th International Workshop on the Implementation of Functional Languages (IFL '07)*, vol. 5083 of *Lecture Notes in Computer Science*, Freiburg, Germany, September 2007. Springer-Verlag, pp. 201–218.

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

[HEO14] Henriksen, T., M. Elsman, and C. E. Oancea. Size slicing: A hybrid approach to size inference in futhark. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '14)*, Gothenburg, Sweden, September 2014. ACM, pp. 31–42.

[HO13] Henriksen, T. and C. E. Oancea. A T2 graph-reduction approach to fusion. In C. Grelck, F. Henglein, U. A. Acar, and J. Berthold (eds.), *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, Boston, MA, USA, FHPC@ICFP 2013, September 25-27, 2013*. ACM, 2013, pp. 47–58.

[JK11] Jo, Y. and M. Kulkarni. Enhancing locality for recursive traversals of recursive structures. In C. V. Lopes and K. Fisher (eds.), *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. ACM, 2011, pp. 463–482.

[JK12] Jo, Y. and M. Kulkarni. Automatically enhancing locality for tree traversals with traversal splicing. In G. T. Leavens and M. B. Dwyer (eds.), *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. ACM, 2012, pp. 355–374.

[KS98] Knobe, K. and V. Sarkar. Array SSA form and its use in parallelization. In D. B. MacQueen and L. Cardelli (eds.), *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. ACM, 1998, pp. 107–120.

[LCK+12] Lippmeier, B., M. M. Chakravarty, G. Keller, R. Leshchinskiy, and S. Peyton Jones. Work efficient higher-order vectorisation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functonal Programming*, Copenhagen, Denmark, September 2012. ACM, pp. 259–270.

[LCKJ12] Lippmeier, B., M. M. T. Chakravarty, G. Keller, and S. L. P. Jones. Guiding parallel array fusion with indexed types. In J. Voigtländer (ed.), *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*. ACM, 2012, pp. 25–36.

[LGCK09] Lee, S., V. Grover, M. M. T. Chakravarty, and G. Keller. Gpu kernels as data-parallel array computations in haskell, 2009.

[LHKK79] Lawson, C. L., R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, **5**(3), September 1979, pp. 308–323.

[MS97] Megiddo, N. and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of the 9th ACM Annual Symposium on Parallel Algorithms and Architectures*. ACM, 1997, pp. 282–291.

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

[OH05] Olukotun, K. and L. Hammond. The future of microprocessors. *ACM Queue*, **3**(7), September 2005. Available from `http://www.acmqueue.org`.

[PPW95] Palmer, D. W., J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Processing*. IEEE Computer Society Press, 1995, pp. 186–193.

[RLK14] Robinson, A., B. Lippmeier, and G. Keller. Fusing filters with integer linear programming. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '14)*, Gothenburg, Sweden, September 2014. ACM, pp. 53–62.

[SBL+14] Sujeeth, A. K., K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embedded Comput. Syst.*, **13**(4s), 2014, pp. 134:1–134:25.

[SDB+16] Sorensen, T., A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamaric. Portable inter-workgroup barrier synchronisation for gpus. In E. Visser and Y. Smaragdakis (eds.), *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. ACM, 2016, pp. 39–58.

[SRA94] Shao, Z., J. Reppy, and A. Appel. Unrolling lists. In *Conference record of the 1994 ACM Conference on Lisp and Functional Programming*. ACM, June 1994, pp. 185–195.

[SS14] Svensson, B. J. and J. Svenningsson. Defunctionalizing push arrays. In J. Berthold, M. Sheeran, and R. Newton (eds.), *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing, FHPC@ICFP 2014, Gothenburg, Sweden, September 4, 2014*. ACM, 2014, pp. 43–52.

[Wil13] Wilt, N. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, Reading, MA, 2013.

[XF10] Xiao, S. and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, Atlanta, GA, April 2010.

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

[ZM13] Zhang, Y. and F. Mueller. Hidp: A hierarchical data parallel language. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2013)*, Shenzhen, China, February 2013. IEEE Computer Society Press, pp. 171–181.