Streaming for Functional Data-Parallel Languages

Frederik M. Madsen
University of Copenhagen
fmma@diku.dk

September, 2016

This thesis has been submitted to the
PhD School of The Faculty of Science,
University of Copenhagen

Abstract

In this thesis, we investigate streaming as a general solution to the space inefficiency commonly found in functional data-parallel programming languages. The data-parallel paradigm maps well to parallel SIMD-style hardware. However, the traditional fully materializing execution strategy, and the limited memory in these architectures, severely constrains the data sets that can be processed. Moreover, the language-integrated cost semantics for nested data parallelism pioneered by NESL depends on a parallelism-flattening execution strategy that only exacerbates the problem. This is because flattening necessitates all sub-computations to materialize at the same time. For example, naive n by n matrix multiplication requires n^3 space in NESL because the algorithm contains n^3 independent scalar multiplications. For large values of n, this is completely unacceptable.

We address the problem by extending two existing data-parallel languages: NESL and Accelerate. In the extensions we map bulk operations to data-parallel streams that can evaluate fully sequential, fully parallel or anything in between. By a dataflow, piecewise parallel execution strategy, the runtime system can adjust to any target machine without any changes in the specification. We expose streams as sequences in the frontend languages to provide the programmer with high-level information and control over streamable and non-streamable computations. In particular, we can extend NESL's intuitive and high-level work–depth model for time complexity with similarly intuitive and high-level model for space complexity that guarantees streamability.

Our implementations are backed by empirical evidence. For Streaming Accelerate we demonstrate performance on par with Accelerate without streams for a series of benchmark including the PageRank algorithm and a MD5 dictionary attack algorithm. For Streaming NESL we show that for several examples of simple, but not trivially parallelizable, text-processing tasks, we obtain single-core performance on par with off-the-shelf GNU Coreutils code, and near-linear speedups for multiple cores.

Resumé

I denne afhandling undersøger vi *streaming* som en general løsning på den plads-ineffektivitet, der er at finde blandt mange funktionelle data-parallelle sprog. Det data-parallelle paradigme har en god oversættelse til SIMD-hardware, men den traditionelle fuldt materialiserende kørselsstrategi, og den begrænsede mængde hukommelse på disse arkitekturer, begrænser dog de datasæt der kan arbejdes på. Den sprog-integrerede omkostningsmodel for *nested* data-parallelisme, pioneret af NESL, afhænger af en parallelisme-udglattende kørselsstrategi, der kun forværrer problemet. Dette er fordi udglatning nødvendiggør at alle delberegninger er materialiseret på samme tid. For eksempel kræver naiv n gange n matrix-multiplikation n^3 plads i NESL, fordi algoritmen indeholder n^3 uafhængige skalar-multiplikationer. For store værdier af n er dette helt uacceptabelt.

Vi adresserer problemet ved at udvide to eksisterende data-parallelle sprog: NESL og Accelerate. I udvidelserne oversætter vi parallelle operationer til data-parallelle strømmer der kan evalueres helt sekventielt, helt parallelt eller alt indimellem. Ved brug af en *dataflow*, stykvis parallel kørselsstrategi, kan runtime-systemet tilpasse sig enhver målmaskine uden nogen ændring i specifikationen. We eksponerer strømme som sekvenser i frontend-sproget for at give programmøren et højt niveau af information og kontrol over strømbare og ikke-strømbare beregninger. Navnligt udvider vi NESL's intuitive og højniveau arbejde–skrid omkostningsmodel med en ligeledes intuitiv og højniveau omkostningsmodel for plads der garanterer strømbarhed.

Vores implementationer er opbakket af empirisk evidens. For Streaming Accelerate demonstrerer vi ydelse på linje med Accelerate uden streaming for en række benchmarks heriblandt PageRank-algoritmen og en MD5 opslagsangrebs-algoritme. For Streaming NESL viser vi for adskillige simple, men ikke trivielt paralleliserbare, tekst-processerings-opgaver, at vi opnår ydelse på linje med GNU Coreutils værktøjer, og nær lineær speedup på flere kerner.

Contents

Preface

This is my PhD dissertation, which I completed at the Computer Science Department at the University of Copenhagen (DIKU) in September 2016 (expected) under the HIPERFIT research center. It is the culmination of my work as PhD student under Andrzej Filinski.

The dissertation is structured around three papers, on all of which I am a main author. The papers were published and presented at different installments of the ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC) in the years 2013, 2015 and 2016. The papers are presented each in a separate chapter, and in chronological order.

The first paper is presented as it was published with a couple of revisions: The original space cost model was too pessimistic and has consequently been updated. The second paper is presented as it was published. The third paper has been heavily revised in order to better fit in the dissertation. The introduction has been removed to avoid repetition, and the description of the target language and its interpreter has been expanded to give a more detailed description. Finally, we have added a section in the end that reflects on the potential of a formal proof of validity of the cost model.

The bulk of the work has been carried out by myself at DIKU under supervision of, and in collaboration with, Andrzej. This work includes Chapter 2 and Chapter 4, and parts of the introduction that are adapted from my Master's Thesis [Mad13], which I completed during my PhD (my PhD was an integrated 4+4 program).

Chapter 3 was developed during my change of scientific environment at the University of New South Wales in Sydney (UNSW), in collaboration with Rob Clifton-Everest under the supervision of Gabriele Keller and Manuel Chakravarty.

Acknowledgments

Andrzej Filinski is an incredibly intelligent, insightful and discerning man, and I hold him in the highest regard. He has been a great help throughout my time at DIKU, and I thank him most sincerely.

Thanks to my dear friends and colleagues at HIPERFIT. Thanks to Martin Elsman for employing me in the research center and for arranging many enjoyable meetings and retreats. Thanks to Fritz Henglein for setting the agenda while allowing me to focus on my own work. Thanks to Martin Dybdal, Troels Henriksen and Cosmin Oancea who also work on data-parallel languages and whom have been a great source of discussions and inspiration.

A warm thank you to all the people I got to know in Sydney during my stay at UNSW. You gave me a great time both professionally and personally. Thanks to Amos Robinson, Michael Schröder, Abdallah Saffidine, George Roldugin, Li Lee and Timo von Holtz. Special thanks to Robert Clifton-Everest for working with me on Streaming Accelerate. Thanks to Manuel Chakravarty and Gabriele Keller for hosting me and making me feel welcome.

An infinite stream of thanks to my beloved Nikoline. Thanks for the surprise party, thanks for accompanying me to Sydney and thanks for enduring my devotion to my work.

Frederik M. Madsen, Copenhagen 2016

Chapter 1

Introduction

1.1 Background and Motivation

Making computations run fast is perhaps the single biggest agenda in contemporary computer science. Today's successful scientific papers often demonstrate impressive improvements in the execution time for some problem in some context. Multiple areas of research are working towards this common goal: Algorithms, programming languages, systems and architectures. This dissertation studies the problem from a programming language perspective.

Programming languages abstract the underlying hardware – the machine. They create a formal language in which programmers and algorithms researchers can talk to each other and to the machine. If the language is sufficiently abstract (high level) the same language may be used for many different machines, including, hopefully, the machines of tomorrow.

This is an attractive property for many reasons. For instance, programs written in a high-level language are less platform-dependent, and therefore also more future proof. Furthermore, a good abstraction hides many low-level details of the machine from the programmer, allowing the programmer to focus on the essence of the problem. Together, these qualities may drastically increase the value of the enormous amount of man hours spend on programming every day, world wide.

However, it is absolutely crucial that the high-level programs execute efficiently on the target machine(s). Execution time and power consumption are both important considerations when considering the quality of a piece of software. For many application, response time is critical, and power consumption adversely affects the environment and one's electricity bill. Energy consumption is proportional to execution time, and consequently, lowering the execution time should be the foremost concern.

1.1.1 Time, Space and the Speed of Light

Computers are not exempt from the laws of physics, but we like to think they are. Even though physics tells us that information cannot travel faster than the speed of light, many machine models in computer science erroneously assumes that it can. The random-access machine model assumes that a piece of information can be retrieved from an arbitrarily large storage space in constant time, which is clearly a violation of the principles of physics.

In reality, the time it takes to access a cell must be proportional to the physical distance between the cell and the observer – here, the computational unit. In an n-sized collection, the time for random access therefore cannot be any faster than $O(\sqrt[3]{n})$. This assumes that the storage occupies all three dimensions of space. In practice, storage is primarily arranged in two-dimensional grids (or small number of layers of two-dimensional grids), which gives an even worse bound of $O(\sqrt{n})$.

A realistic value of n is given by the address space of modern 64-bit systems, which allows up to 2^{64} locations. Constant-time random-access can be justified in a machine model by considering this number to be a constant. However, the hidden constant factors are not insignificant. Most models that do account for the size of the storage, charge $O(\log n)$ for random-access. This is still overly optimistic if one considers the concrete factors at $n = 2^{64}$:

$$\begin{aligned}
\text{Random-access:} &= 1 \\
\text{Log-access:} &= \log_2(2^{64}) = 64 \\
\text{Theoretical limit:} &= \sqrt[3]{2^{64}} \approx 2,600,000 \\
\text{Theoretical limit (2D):} &= \sqrt{2^{64}} \approx 4,300,000,000
\end{aligned}$$

Over the years, we have witnessed a general increase in the amount of data being processed and the sheer scale of computations. This makes the problem increasingly more pronounced, and the model becomes increasingly inaccurate. Reality shows that locality matters, and caching mechanisms have proven to be a serious necessity. Accessing information not in cache is almost always orders of magnitude slower than accessing in-cache information.

Caches are by no means a silver bullet. They only work when the same piece of information is retrieved multiple times in a short period of time. Whether or not they create the illusion of true random-access, therefore depends on the nature and the right formulation of the problem. Caches are implemented by hardware designers, and are usually completely transparent to the programmer. If the programmer wishes to tweak the program to get better cache behavior, it usually has to be done at a very low level of

abstraction; the programmer must know the details of the cache hierarchy of the machine.

In this dissertation, we attack the problem from a programming language standpoint. A part of our hypothesis is, that by confining random-access to only where it is absolutely needed, at the language level, *streaming*, as a programming language paradigm, can aid programmers design, implement and reason about random-access-friendly algorithms at a high level of abstraction, without even knowing what a cache is.

However, by simply abolishing random-access everywhere, we end up with a terribly restricted language. Even though many problems can be expressed without the use of random-access, other problems inherently require the ability to perform the indexing operation. The challenge is therefore to design an abstraction for streams that clearly encapsulates the extend of which a computation requires random-access withing a context of streams and manifest data.

Furthermore, we want a language that affords high-level reasoning about time and space, and that can be implemented efficiently on any given machine, especially including, since we care about performance, parallel machines.

1.1.2 Parallelism

After the breakdown of Moore's law, one of the key factors in modern hardware performance is parallelism. Since a single core hardly gets any faster, modern machines are made faster by employing many parallel cores. As long as the problem to be solved can be hopefully broken down into more pieces, more cores can work on the problem at the same time. Almost all problems exhibit parallelism in one way or another.

Parallel machines come in many flavors, and are generally programmed in wildly different ways. Exploiting the full potential parallelism of a given parallel machine, requires careful attention to the low-level details. If a programming language is too abstract, it becomes very difficult to exploit the parallelism fully, not only from the perspective of the programmer, but also from the perspective of the compiler that has to infer these low-level details. Thus, there is a trade-off between abstraction and performance.

A high-level language can never perform as well as a low-level language, but we try to get as close as possible; close enough that the benefits of abstraction out-weights the loss in performance. One of the main hurdles, and the focus of this dissertation, is the advent and the added complexity of parallelism.

1.1.3 Data Parallelism

Data parallelism is one approach to dealing with parallelism from a language point-of-view. Here, parallelism is expressed as parallel operations on sequences of values. The operations on sequences are specified uniformly, and the language then translates this to parallel operations on the target machine however it sees fit. Thus, the programmer can expect the values of a sequence to be computed simultaneously if the machine has enough parallel resources.

This is very natural and high-level way of thinking about parallelism. There are other ways of thinking about parallelism, for instance task parallelism. Here, parallelism must be explicitly stated by the programmer in the form of tasks to run, potentially on different physical processors. Although more natural for some applications (including applications in systems, communication and concurrency), this approach is a lot more explicit, and therefore a lot less abstract, than the data-parallel approach. Moreover, it precludes homogeneously parallel machines, that are unable to exploit parallelism in completely unrelated tasks. In data parallelism, task are inherently related, since operation are specified uniformly.

It is important to stress, that data-parallel programming languages almost always express work that *can* be done in parallel, not work that *must* be done in parallel. This is because the *available parallel resources*, the number of available processing units, can vary greatly from back-end to back-end, and it is usually easy for the compiler to "chunk up" too parallel steps into several steps, whereas it would be tedious, if not impossible, for the programmer to do so manually for each imaginable back-end. The amount of work in a single step can therefore be used as a measure of *potential parallel degree* of a program, and once a program is executed on a specific back-end, we can measure the *actual parallel degree* of the program, which should be the same as the minimum of the available parallel resources and potential parallel degree.

Data parallelism in a programming language can be achieved simply by having a library of parallel algorithms that operate on sequences. A common way to introduce more expressive data parallelism, is to introduce a new construct that essentially is a parallel *foreach loop*, or a parallel sequence comprehension:

$$\{e_0 : x \textbf{ in } e_1\},$$

where x may be free en e_0. We call this parallel comprehension an *apply-to-each*, and the semantics is, first e_1 must evaluate to a sequence of values $[v_1, ..., v_l]$. Then, the body expression e_0 is evaluated l times, once for each v_i substituted for x, the result of which forms the resulting sequence of the

whole expression. In functional languages, this construction is commonly referred to as a *map* of $\lambda x.e_0$ over e_1.

A key point is, that each evaluation of the body must be independent in order to claim that they can be executed in parallel or out of order without unexpected results, and it is therefore necessary that the body expression has no side-effects, that can effect the parallel evaluations. This makes functional programming very attractive for this kind of data parallelism.

1.1.4 Nested Data Parallelism (NDP)

Some data-parallel programming languages allows data-parallel to be nested. Those languages are referred to as nested data-parallel languages. The unrestricted apply-to-each construct affords nested data-parallelism in the form of nested maps and maps of other data-parallel operations. An example of a nested data-parallel expression is:

$$\{\{e_0 : x \textbf{ in } e_1\} : y \textbf{ in } e_2\}$$

Conversely, a flat data-parallel programming language is a programming language that can express data parallelism, but does not allow nesting. This is usually imposed as restrictions in the type system, or by simply not having an apply-to-each construct in the language.

Nestedness is an inherent property of a data-parallel language with the parallel apply-to-each construct. Consequently, nested data-parallel languages are generally more expressive than flat parallel languages, and many common parallel algorithms are indeed more concise and/or more potentially parallel, when written in a nested data-parallel language [Ble96]. At the source level, nested data parallelism is clearly more desirable than flat data parallelism, but at the compiler level the story is the opposite.

As mentioned in the previous section, it is usually up to the compiler to perform "chunking" as necessary in a data-parallel language. "Chunking" is trivial if the data-parallel operation is flat since it is just a matter of doing some work in one step, and then proceed to do the rest. In the case of nested data parallelism the process of "chunking" becomes less obvious as each sub-computation may have a different potential parallel degree. A common approach for nested data parallelism languages is to *flatten* nested data parallelism expressions into equivalent flat data-parallel expressions first, and then compile it or interpret it using a simple flat data-parallel compiler or interpreter.

1.1.5 Flattening

Flattening was first proposed by Guy Blelloch and realized in the programming language NESL [BS90, Ble90b, Ble95], and has later been studied and

refined by others such as Jan Prins and Daniel Palmer with the language Proteus [PP93, RPI95, PPW95], and Gabriele Keller, Simon Peyton Jones and Manuel Chakravarty with the language Data Parallel Haskell [KS96, PJ08, CLJ+07].

The main idea of flattening is to lift every primitive operation in the language. In the lifted version, the input and output type are lifted to sequences. E.g if we have an addition primitive with the type

$$+ : (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int},$$

then lifted addition has the type (where square brackets denote both lifting and sequence type):

$$\mathcal{L}(+) : [(\mathsf{Int}, \mathsf{Int})] \to [\mathsf{Int}]\,.$$

Furthermore, it is common to convert sequences of tuples into tuples of sequences. In that case, lifted addition has the type:

$$\mathcal{L}(+) : ([\mathsf{Int}]\,, [\mathsf{Int}]) \to [\mathsf{Int}]\,.$$

In this representation, barring length checking, zipping (and unzipping) becomes free, which is very convenient for when dealing with lifted contexts containing many conceptually zipped vectors.

Lifting can be extended to expressions. In a standard type system, for any well-typed expression e with type τ in type context Γ (written as $\Gamma \vdash e : \tau$), the lifting of e has type:

$$[\Gamma] \vdash \mathcal{L}(e) : [\tau]$$

where the length of the context (which is a sequence of contexts) is supposed to be the same as the length of the result.

If we apply the sequence-of-tuples to tuple-of-sequences transformation on the context $\Gamma = [x_1 \mapsto \tau_1, ..., x_k \mapsto \tau_k]$, the lifting of e has the type:

$$[x_1 \mapsto [\tau_1], ..., x_k \mapsto [\tau_k]] \vdash \mathcal{L}(e) : [\tau]$$

which gives a single context (rather than a sequence of contexts) that can be used in a standard type system. Furthermore, lifting variables and let-bindings becomes trivial and can be implemented simply as:

$$\mathcal{L}(x) = x$$

$$\mathcal{L}(\textbf{let } x = e_0 \textbf{ in } e_1) = \textbf{let } x = \mathcal{L}(e_0) \textbf{ in } \mathcal{L}(e_1)$$

At the basis, primitive operation application is then lifted by using the lifted version of the operation:

$$\mathcal{L}(p(x)) = \mathcal{L}(p)(x)$$

However, constants become a bit more complicated. For $\Gamma \vdash n : \mathsf{Int}$ where n ranges over constant integers, we need the lifted version $[\Gamma] \vdash \mathcal{L}(n) : [\mathsf{Int}]$ to evaluate to a sequence of n's where the length of the sequence is the same as the length of the context (which is a sequence of contexts). In order to do so, the language must support the primitive (or something equivalent):

$$distribute_\tau :: (\tau, \mathsf{Int}) \to [\tau]$$

In the tuple-of-sequences representation, we do not always have the length argument at hand. In particular, the context might be empty, and so, we cannot provide a lifting for constants in that case:

$$\mathcal{L}(n) = distribute(n, ?)$$

If the context is non-empty, i.e. if there exists an x in the (lifted) domain, we can define

$$\mathcal{L}(n) = distribute(n, length(x))$$

where

$$length_\tau :: [\tau] \to \mathsf{Int}$$

must be provided as a primitive also. One solution is to let lifting be indexed by and expression that evaluates the needed length. Another solution is to ensure that the context is always non-empty by let-binding a dummy control value at the top level, e.g.:

$$\textbf{let } ctrl = () \textbf{ in } e$$

We will use the latter approach.

As we shall see in a moment, it is not necessary to define lifting of apply-to-each. The ultimate goal of lifting is to eliminate nested data parallelism. Lifting allows us to eliminate apply-to-each constructs, thus eliminating nested data parallelism. The required transformation is called flattening. We denote it here by $\mathcal{F}(-)$. It operates on typed expressions, and it preserves the types. If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \mathcal{F}(e) : \tau$ and $\mathcal{F}(e)$ is guaranteed to be free of apply-to-each constructs.

It works by transforming apply-to-each constructs to let-bindings. The body expression is lifted to operate on the entire sequence at once instead of on each element. This lifting requires the surrounding context (the variables in the context other than x) to be distributed over the length of x. In a context where the domain (without x) is $x_1, ..., x_k$, the flattening of the apply-to-each

construct is:

$$\begin{aligned}\mathcal{F}(\{e_0 : x \textbf{ in } e_1\}) = \textbf{let}\quad & x = \mathcal{F}(e_1)\\ & x_1 = distribute(x_1, length(x))\\ & \vdots\\ & x_k = distribute(x_k, length(x))\\ \textbf{in}\quad & \mathcal{L}(\mathcal{F}(e_0))\end{aligned}$$

Note that let-bindings here are not recursive, and $x_1 = distribute(x_1, length(x))$ defines a new value for x_1 that hides the old value. This is the central case in the flattening transformation. In all other cases, $\mathcal{F}(-)$ is simply applied to the sub-expressions. By flattening e_0 *before* lifting it, we eliminate apply-to-each constructs before lifting and do therefore not have to consider lifting apply-to-each.

As a simple example, consider flattening the expression $\{x+1 : x \textbf{ in } range(1,10)\}$ in the empty context. Here $range(1,10)$ is a primitive operation that computes the sequence $[1,2,...,10]$. We will use a top-level control value to lift constants, so we want to calculate:

$$\mathcal{F}(\textbf{let } ctrl = () \textbf{ in } \{x+1 : x \textbf{ in } range(1,10)\})$$

The first step is to flatten the let-binding which simply flattens the let-bound expression and the body expression. Flattening the let-bound expression does nothing as $\mathcal{F}(()) = ()$, so

$$\begin{aligned}&\mathcal{F}(\textbf{let } ctrl = () \textbf{ in } \{x+1 : x \textbf{ in } range(1,10)\})\\ &\quad = \textbf{let } ctrl = () \textbf{ in } \mathcal{F}(\{x+1 : x \textbf{ in } range(1,10)\})\end{aligned}$$

We now arrive at the interesting case where we must flatten an apply-to-each. Following the rule, we get:

$$\begin{aligned}&\textbf{let } ctrl = () \textbf{ in } \mathcal{F}(\{x+1 : x \textbf{ in } range(1,10)\})\\ &\quad = \textbf{let } ctrl = () \textbf{ in let}\quad x = \mathcal{F}(range(1,10))\\ &\qquad\qquad\qquad\qquad\qquad\quad ctrl = distribute(ctrl, length(x))\\ &\qquad\qquad\qquad\qquad\quad \textbf{in}\quad \mathcal{L}(\mathcal{F}(x+1))\end{aligned}$$

We can then apply $\mathcal{F}(range(1,10)) = range(1,10)$ and $\mathcal{F}(x+1) = x+1$ (since these expression do not contain apply-to-each), and lift the body by $\mathcal{L}(x+1) = \mathcal{L}(+)(x, distribute(1, length(ctrl)))$ to get the final result:

$$\begin{aligned}&\mathcal{F}(\textbf{let } ctrl = () \textbf{ in } \{x+1 : x \textbf{ in } range(1,10)\})\\ &\quad = \textbf{let } ctrl = () \textbf{ in let}\quad x = range(1,10)\\ &\qquad\qquad\qquad\qquad\qquad\quad ctrl = distribute(ctrl, length(x))\\ &\qquad\qquad\qquad\qquad\quad \textbf{in}\quad \mathcal{L}(+)(x, distribute(1, length(ctrl)))\end{aligned}$$

Further simple program analysis allows us to reduce the whole expression to:

$$\textbf{let } x = range(1, 10) \textbf{ in } \mathcal{L}(+)(x, distribute(1, length(x)))$$

In the case of nested data parallelism, the operation in the inner-most body will be lifted twice, as can be seen by the following example: In the context $[xss \mapsto [[\mathsf{Real}]]]$, we have:

$$\begin{aligned}&\mathcal{F}(\{\{sqrt(x) : x \textbf{ in } xs\} : xs \textbf{ in } xss\})\\ &\quad = \textbf{let } xs = xss \textbf{ in let } x = xs \textbf{ in } \mathcal{L}(\mathcal{L}(sqrt)(x))\end{aligned}$$

Then, how do we proceed with $\mathcal{L}(\mathcal{L}(sqrt)(x))$? If we use the same rule for lifting for lifted operation as we do for non-lifted operations, we would get $\mathcal{L}(\mathcal{L}(p)(x)) = \mathcal{L}(\mathcal{L}(p))(x)$. However, that would require us to have to define lifted versions of every operation up to an arbitrary high level of lifting. In order to avoid that, the lifting transformation can treat lifting lifted operations differently from lifting non-lifted operations. Essentially, it is possible to transform doubly-lifted operations into singly-lifted operations by concatenating the input, applying the singly-lifted operation and partitioning the output according to the shape of the original input. For that we need two additional and fairly standard primitive operations:

$$concat_\tau :: [[\tau]] \to [\tau]$$

$$partition_\tau :: ([\tau], [\mathsf{Int}]) \to [[\tau]]$$

We can then describe the lifting of lifted operations as follows:

$$\mathcal{L}(\mathcal{L}(p)(x)) = partition(\mathcal{L}(p)(concat(x)), \mathcal{L}(length)(x))$$

Here, $\mathcal{L}(length)$ is the lifted version of *length* that computes the top-most structure of a nested sequence. The following is one way to define (by example) the three operations:

$$\begin{aligned}concat([[a, b], [], [c]]) &= [a, b, c]\\ \mathcal{L}(length)([[a, b], [], [c]]) &= [2, 0, 1]\\ partition([a, b, c], [2, 0, 1]) &= [[a, b], [], [c]]\end{aligned}$$

The whole approach is made viable by a clever representation of nested sequences that allows these three operations to be cheap. A nested sequence is represented by a flat data sequence with an accompanying segment descriptor that holds the nesting structure of the nested sequence. As an example the nested sequence:

$$v = \big[[1, 2, 3],\ [4],\ [],\ [5, 6]\big],$$

will be represented by a flat data sequence:

$$d = [1, 2, 3, 4, 5, 6],$$

and a segment descriptor describing the length of each sub-sequence:

$$s_0 = [3, 1, 0, 2].$$

This representation can be generalized to sequences of any nesting depth. For example:

$$v = \Big[[] \ , \ \ [[1, 2, 3], [4], [], [5, 6]] \ , \ \ [[7], [], [8, 9, 10]] \Big]$$

will be represented by

$$d = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

and

$$s_0 = [3, 1, 0, 2, 1, 0, 3]$$
$$s_1 = [0, 4, 3]$$

Concatenation is then achieved by simply removing the top-most segment descriptor, and similarly partitioning by the shape of the original input is achieved by attaching the top segment descriptor of the original input on top of the result. A general property of the segment descriptors is that the sum of the elements in a segment descriptor equals the length of the lower segment descriptor (or data sequence). Depending on the backend, flattening is a necessary step in order to execute nested data-parallel programs. Flattening also gives work balancing for map-like operations for free, since we concatenate all sub-sequences in nested data parallelism expressions. Reduction and scans can also be made work balanced by providing efficient lifted versions of these operations. These essentially become segmented reduce and segmented scan, which have well-known efficient parallel implementations.

1.1.6 Ideal Cost Model

The usual way to reason about time in an abstract way is to consider the two extreme cases: A sequential machine with one processor (available parallel resources = 1), and a machine with an unbounded number of processors (available parallel resources = ∞). The time complexity on a sequential machine is often called work and the time complexity on an unbounded parallel machine is often called steps or depth. Steps is also a measure of the longest chain of sequential dependencies in a program.

By assigning an idealized work and step complexity to the primitive operations, the ideal work and step complexities of any expression can be quantified in a natural language-based cost model alongside the value semantics. This provides a simple compositional cost model, that is easily understood by the programmer.

The reasons why this approach works, as Blelloch demonstrates using Brent's Lemma in [Ble90b], is that the asymptotic time complexity T on a given machine with P processors can then be derived as a function of work and steps:

$$T = O(work/P + steps)$$

I.e. the total work can be cleanly distributed over P processors with the exception of the longest sequential path. Unfortunately, this result assumes the random-access model, which does not sufficiently penalize random-access. Consequently, large vectors are not discouraged in the considered implementation models. On the contrary, bigger is better, as large vectors exhibit a larger degree of potential parallelism. In practice, this causes a space cost proportional to the degree of potential parallelism, when, in many cases, a space cost proportional to the degree of realized parallelism is sufficient.

For example, $n \times n$ matrix multiplication contains n^3 scalar multiplications that can all be computed in parallel. Whether it is a good idea to compute them all in parallel or not depends on n and the available parallel resources of the backend. Both are parameters that the programmer should not be aware of when expressing the algorithm in a high level language.

For an even simpler example, consider the expression

$$sum(range(1, n)),$$

that simply sums the numbers from 1 to n. If we assign the following realistic costs to the basic operations:

Operation	Work	Steps
sum	n	$\log n$
range	n	1,

the cost model will charge the whole expression with $2n$ work and $1 + \log n$ steps.

This can be trivially realized by performing the two steps one at a time and storing the intermediate result in memory in its full length. In other words, the cost model suggests a fully eager semantics. For large values of n, not only will this result in bad cache performance, we might run out of space altogether.

An immediate fix, is to manually divide the computation at the expression level, and transform the expression into the equivalent expression (with

respect to the result):

$$sum(range(1, n/2)) + sum(range(n/2 + 1, n))$$

where the two operands to $(+)$ are computed sequentially. That is, first $sum(range(1, n/2))$ fully evaluates to a number, and only then will $sum(range(n/2 + 1, n))$ evaluate.

Even though work is still $2n$, the steps is increased from $1 + \log n$ to $2 + 2\log\frac{n}{2} = 2 + 2(\log n - 1) = 2\log n$. $2\log n$ is almost always words than $1 + \log n$. This means the execution time probably gets worse on machines that have sufficient available parallel resources. On the other hand, a less parallel machine might benefit from this transformation. We do not want to punish the programmer for exposing too much parallelism. The actual space cost should not depend on the potential degree of parallelism, but rather the degree of realized parallelism. What is needed is a cost model for space, that prohibits the implementation model from being fully eager – unless there is enough available parallel resources.

There is no standard way to reason about space in a data-parallel program; the subject is not covered as much as time complexity. Most existing data-parallel languages do not distinguish the space complexity on a sequential machine from the space complexity on a parallel machine[1], i.e. there is no space-related equivalent of work and steps giving a space complexity cost model in the two extreme case of available parallel resources = 1 and available parallel resources = ∞. One of the assertions of this dissertation is that we need to make this distinction between sequential and parallel space, and one of the contributions is to provide a natural, language-based cost model, that sufficiently quantifies the space complexity, and derives an ideal space cost given a concrete machine.

Continuing with our simple example, our approach is to assign sequential and parallel space cost to the primitive operations:

Operation	Work	Steps	Sequential space	Parallel space
sum	n	$\log n$	1	1
range	n	1	1	n,

and augmenting the semantics with a language-based cost model for space. By doing so, in addition to having values for work and steps, our examples has a sequential space of 2 and a parallel space of $1 + n$.

Ideally, on a concrete machine with P processors, the evaluation of an expression with sequential space S_1 and parallel space S_∞ must satisfy an

[1]There are some notable exception to this statement, Guy Blelloch et. al. has given a provably space efficient scheduling for nested parallelism[BGM99], but their work only applies to fine-grained control parallelism.

asymptotic space cost of

$$S = O(\min(P \cdot S_1, S_\infty)).$$

Our example must then evaluate in $O(\min(P, n))$ space. This means that the implementation model is no longer allowed to manifest the entire range before commencing the reduction. Keep in mind that it still has to obey the time cost model which stipulates an asymptotic execution time in $O(n/P)$. The only way this can be realized, is if the execution model executes the operations in *chunks* where the size of each chunk is $O(P)$. Instead of a fully eager semantics, the execution model needs to dynamically perform the transformation we did earlier, and instead of splitting the expression in two, it may have to split the expression into multiple steps.

It is, however, not always possible to evaluate each step of a program in sequential fixed-sized chunks. There are a number of situations in which our simple approach breaks down. For instance, as mentioned earlier, we cannot allow unrestricted indexing, as the chunk that contains the indexed element may not be available at the time of the index operation. Another problem arises if we attempt to use a sequence from the outer context in the body of an apply-to-each. For example the expression

$$\textbf{let } xs = range(1, 1000) \textbf{ in } \{xs : \_ \textbf{ in } range(1, 100)\}$$

requires traversing xs 100 times in the apply to each. We will explore the problematic situations and how we can deal with them in this dissertation.

1.2 Hypothesis and Method

Data-parallel programming languages – in particular, *nested* data-parallel programming languages – can become more space-efficient – without degrading performance – by using streaming semantics instead of being fully eager. Execution times may even be faster due to better cache utilization. Mainly due to random-access, the class of programs amendable for streaming execution is not clearly marked out in most languages, and in order to provide predictable performance and/or formal guarantees about performance, the frontend language must be extended with a restricted syntax for streamable expressions. It is possible to define such an extension such that:

1. It contains a large subset of the original language, and therefore allows a large class of data-parallel problems to be expressed as streams.

2. It integrates well with the original language in the sense that streams may be converted to manifest vectors and vice versa, and streams of non-streamable computations are expressible.

3. Streams are data-parallel and have at least as good time performance as their manifest counterparts in the original language.

4. It allows high-level platform-independent predictions and/or guarantees about space performance, with excellent space performance for streams compared to fully manifest data structures.

The hypothesis is tested by theorizing, implementation and experimentation on different platforms, with most of the work devoted to implementation. We implement streaming extensions to two existing languages: NESL and Accelerate. Streaming NESL targets multicores with vector units and Streaming Accelerate targets GPUs. The two implementations are validated by empirical experiments.

1.3 μNESL

The primary focus in this dissertation centers around the programming language NESL [Ble95]. NESL is a first-order functional nested data-parallel language with ML-like syntax. The standard implementation model for NESL is VCODE, which is a stack-based vector bytecode language. It suffers from the space problem of fully eager evaluation outlined in the previous sections.

NESL has a formal cost model for work and steps, which makes it suitable for testing our hypothesis. In this section we give a brief description of μNESL, a language we have designed as a simplified version of NESL that contains only the core functionality. This is the language we attempt to improve by extending it with stream syntax and semantics, and a cost model for space usage.

The values of μNESL are:

$$\begin{aligned} a &::= \mathsf{T} \mid \mathsf{F} \mid n\ {\scriptstyle (n \in \mathbf{Z})} \mid r\ {\scriptstyle (r \in \mathbf{R})} \mid \cdots \\ v &::= a \mid (v_1, ..., v_k) \mid [v_1, ..., v_l] \end{aligned}$$

where $[v_1, ..., v_l]$ is a sequence of values of length l. As part of our notation, l and k both range over natural numbers: k are small, statically known numbers, such as the number of elements in a tuple, while l are potentially very large runtime-dependent numbers, such as the length of a sequence.

The types are:

$$\begin{aligned} \pi &::= \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Real} \mid \cdots \\ \tau &::= \pi \mid (\tau_1, ..., \tau_k) \mid [\tau] \end{aligned}$$

The syntax of μNESL expressions is defined as:

$$
\begin{array}{rcl}
e & ::= & x \mid a \mid (e_1, ..., e_k) \mid e.i \mid \textbf{let } x = e_0 \textbf{ in } e_1 \mid \phi(e) \mid \{e_0 : x \textbf{ in } e_1\} \\
\phi & ::= & \oplus \mid mkseq^k_\tau \mid zip^k_{\tau_1,...,\tau_k} \mid partition_\tau \mid concat_\tau \mid \& \mid \#_\tau \mid !_\tau \\
 & & \mid reduce_R \mid scan_R \\
\oplus & ::= & + \mid - \mid log \mid \cdots \\
R & ::= & + \mid * \mid max \mid min
\end{array}
$$

μNESL has let-bindings, products and nested data parallelism in the form of the apply-to-each construct. The builtin operations ϕ (see Figure 1.2) contains only the minimal set of operations necessary to express all of NESL's operations. The &-operation reads as *iota* and $\&(n)$ computes the sequence $[0, ..., n-1]$. The #-operation is the length operation and the !-operation is indexing.

For ease of reading, we introduce a couple of syntactic short-hands:

$$[e_1, ..., e_k] \equiv mkseq^k(e_1, ..., e_k)$$

$$e_1 +\!\!+ e_2 \equiv concat([e_1, e_2])$$

Conditionals are not primitive as they are expressible through the other constructs of the language. As demonstrated in the following, we can represent a nullable value with an empty sequence for null and a unit sequence for a non-null value, and we can then use apply-to-each to evaluate an expression conditionally:

$$\textbf{if } e_1 \textbf{ then } e_2^\tau \textbf{ else } e_3^\tau \equiv$$

$$\begin{array}{l}
\textbf{let } b = bool2int(e_1) \textbf{ in} \\
\quad (\{e_2 : _ \textbf{ in } \&b\} +\!\!+ \{e_3 : _ \textbf{ in } \&(1-b)\})\ !_\tau\ 0
\end{array}$$

Likewise, we can define *guarded* apply-to-each constructs:

$$\{e_0 : x \textbf{ in } e_1 \mid e_2\} \equiv$$

$$concat(\{\{e_0 : \_ \textbf{ in } \&(bool2int(e_2))\} : x \textbf{ in } e_1\})$$

Here, the body expression e_0 is only evaluated in the cases where e_2 evaluates to T. Notice that crucially, x is available to be used in e_2.

The type system (Figure 1.1) and big-step semantics (Figure 1.4) are quite standard, with the exception that the semantics incorporates a cost model for work and steps. Primitive operations are given types and semantics separately in Figures 1.2 and 1.3.

All primitive operation are defined to take unit steps, which means we abstract away the logarithmic depth that is often associated with a divide-and-conquer parallel implementation of primitive operations. Instead, we

$\boxed{\Gamma \vdash e :: \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau} \qquad \frac{}{\Gamma \vdash \mathsf{T} :: \mathsf{Bool}} \quad \frac{}{\Gamma \vdash n :: \mathsf{Int}} \quad \frac{}{\Gamma \vdash r :: \mathsf{Real}} \quad \cdots$$

$$\frac{\left(\Gamma \vdash e_i :: \tau_i\right)_{i=1}^{k}}{\Gamma \vdash (e_1, ..., e_k) :: (\tau_1, ..., \tau_k)} \qquad \frac{\Gamma \vdash e :: (\tau_1, ..., \tau_k)}{\Gamma \vdash e.i :: \tau_i}\ (1 \le i \le k)$$

$$\frac{\Gamma \vdash e_0 :: \tau_0 \quad \Gamma[x \mapsto \tau_0] \vdash e_1 :: \tau_1}{\Gamma \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 :: \tau_1} \qquad \frac{\Gamma \vdash e :: \tau_1 \quad \phi :: \tau_1 \to \tau_2}{\Gamma \vdash \phi(e) :: \tau_2}$$

$$\frac{\Gamma \vdash e_0 :: [\tau_0] \qquad \Gamma[x \mapsto \tau_0] \vdash e :: \tau}{\Gamma \vdash \{e : x\ \mathbf{in}\ e_0\} :: [\tau]}$$

Figure 1.1: Typing rules for μNESL.

$\boxed{\phi :: \tau_1 \to \tau_2}$

$$\begin{array}{rcl}
+ & :: & (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int} \\
 & \vdots & \\
mkseq_{\tau}^{k} & :: & (\overbrace{\tau, ..., \tau}^{k}) \to [\tau] \quad k \ge 0 \\
zip_{\tau_1,...,\tau_k}^{k} & :: & ([\tau_1], ..., [\tau_k]) \to [(\tau_1, ..., \tau_k)] \quad k \ge 1 \\
partition_{\tau} & :: & ([\tau], [\mathsf{Int}]) \to [[\tau]] \\
concat_{\tau} & :: & [[\tau]] \to [\tau] \\
\& & :: & \mathsf{Int} \to [\mathsf{Int}] \\
\#_{\tau} & :: & [\tau] \to \mathsf{Int} \\
!_{\tau} & :: & ([\tau], \mathsf{Int}) \to \tau \\
reduce_R & :: & [\pi] \to \pi \quad R::(\pi,\pi)\to\pi \\
scan_R & :: & [\pi] \to [\pi] \quad R::(\pi,\pi)\to\pi
\end{array}$$

Figure 1.2: Primitive operations types.

$\boxed{F_\phi(v_0) = (v, W)}$

$$
\begin{array}{rcl}
+(n_0, n_1) & = & (n_0 + n_1, 1) \\
 & \vdots & \\
mkseq^k_\tau(v_1, ..., v_k) & = & ([v_1, ..., v_k], \Sigma^k_{i=1}|v_i|) \quad k{\geq}0 \\
zip^k_{\tau_1,...,\tau_k}([v^1_1, ..., v^l_1], ..., [v^1_k, ..., v^l_k]) & = & ([(v^1_1, ..., v^1_k), ..., (v^l_1, ..., v^l_k)], \\
 & & \Sigma^l_{i=1}\Sigma^k_{j=1}|v^i_j|) \quad k{\geq}1 \\
partition_\tau([v_1, ..., v_l], [n_1, ..., n_{l'}]) & = & (\overbrace{[v_1, ...]}^{n_1}, ..., \overbrace{[..., v_l]}^{n_{l'}}], \Sigma^l_{i=1}|v_i|) \quad \Sigma^{l'}_{i=1} n_i = l \\
concat_\tau([[v_1, ...], ..., [..., v_l]]) & = & ([v_1, ..., v_l], \Sigma^l_{i=1}|v_i|) \\
\&(n) & = & ([0, ..., n-1], n) \quad 0{\leq}n \\
\#_\tau([v_1, ..., v_l]) & = & (l, 1) \\
!_\tau([v_1, ..., v_l], n) & = & (v_n, |v_n|) \quad 0{\leq}n{<}l \\
reduce_R([v_1, ..., v_l]) & = & ([\![R]\!]^l_{i=1} v_i, l) \\
scan_R([v_1, ..., v_l]) & = & ([0, [\![R]\!]^1_{i=1} v_i, ..., [\![R]\!]^{l-1}_{i=1} v_i], l)
\end{array}
$$

Figure 1.3: Primitive operations semantics. $[\![R]\!]$ is the reduction denotation of the reduction operator R. E.g. summation for $R = +$.

add a logarithmic term to the derived asymptotic time complexity. That is, given an evaluation of $\rho \vdash e \Downarrow v \,\$\, (W, D)$, we can expect an asymptotic actual running time on P processors in

$$O(W/P + D \log P),$$

which has $D \log P$ instead of D. As derived by Blelloch [Ble90b] among others, in the PRAM model, the logarithmic factor shows up anyway due to the cost of allocating tasks to processors, so there is not much gained in accounting for the logarithmic depth in the cost model, and abstracting it away makes depth costing simpler.

The work cost of operations are defined along side their semantics in Figure 1.3. Work is often related to the size of the computed value. The size of a value v is denoted by $|v|$, and is defined as:

$$
\begin{array}{rcl}
|a| & = & 1 \\
|(v_1, ..., v_k)| & = & \Sigma^k_{i=1}|v_i| \\
|[v_1, ..., v_k]| & = & \Sigma^k_{i=1}|v_i|
\end{array}
$$

$\boxed{\rho \vdash e \Downarrow v \,\$\, (W, D)}$

$$\frac{\rho(x) = v}{\rho \vdash x \Downarrow v \,\$\, (1,1)}$$

$$\frac{}{\rho \vdash a \Downarrow a \,\$\, (1,1)}$$

$$\frac{\left(\rho \vdash e_i \Downarrow v_i \,\$\, (W_i, D_i)\right)_{i=1}^{k}}{\rho \vdash (e_1, ..., e_k) \Downarrow (v_1, ..., v_k) \,\$\, (\Sigma_{i=1}^{k} W_i, \Sigma_{i=1}^{k} D_i)}$$

$$\frac{\rho \vdash e \Downarrow (v_1, ..., v_k) \,\$\, (W, D)}{\rho \vdash e.i \Downarrow v_i \,\$\, (W, D)}$$

$$\frac{\rho \vdash e_0 \Downarrow v_0 \,\$\, (W_0, D_0) \quad \rho[x \mapsto v_0] \vdash e_1 \Downarrow v_1 \,\$\, (W_1, D_1)}{\rho \vdash \textbf{let } x = e_0 \textbf{ in } e_1 \Downarrow v_1 \,\$\, (W_0 + W_1, D_0 + D_1)}$$

$$\frac{\rho \vdash e_0 \Downarrow v_0 \,\$\, (W_0, D_0) \quad F_\phi(v_0) = (v, W)}{\rho \vdash \phi(e_0) \Downarrow v \,\$\, (W_0 + W, D_0 + 1)}$$

$$\frac{\rho \vdash e_0 \Downarrow \{v_1, ..., v_l\} \,\$\, (W_0, D_0) \quad \left(\rho[x \mapsto v_i] \vdash e \Downarrow v_i' \,\$\, (W_i, D_i)\right)_{i=1}^{l}}{\rho \vdash \{e : x \textbf{ in } x_0\} \Downarrow \{v_1', ..., v_l'\} \,\$\, (W_0 + \Sigma_{i=1}^{l} W_i, D_0 + \max_{i=1}^{l} D_i)}$$

Figure 1.4: Evaluation semantics with costs.

Unlike NESL, μNESL does not support recursion. If recursion is desired, it must be manually unfolded. Parallel algorithms are almost always limited to a logarithmic depth recursion, so static unfolding is not as bad as it may sound. The biggest problem is unfolding a recursion with a statically unknown depth. In this case, the recursion must be unfolded a number of times that represents the maximum possible recursion depth. A recursion that splits an array of unknown length in two equal sizes can be unfolded 32 times and can then support arrays up to a length of 2^{32}.

Figure 1.5 lists a classical example of a program written in NESL (apart from the recursive function definition and some type annotations, this is actually a μNESL program). The program is a parallel implementation of the quicksort algorithm that showcases the use of nested data parallelism in a recursive divide-and-conquer strategy.

In the recursive case, the function `qsort` is called inside an apply-to-each on the two recursive cases: On all the elements smaller than the pivot and on all the elements greater than the pivot. In order to obey the cost model, the evaluation must take advantage of parallelism across blocks in the call tree on the same horizontal level in the figure. This is exactly what flattening achieves.

The same program can be written in μNESL by unfolding the definition of `qsort` a small number of times, with the deepest occurrence replaced by some error-generating expression (e.g. a division by zero) in the hope that this case will never be reached. Since the recursion depth is expected to be logarithmic in the size of the input sequence, a modest number of unfolds is almost always sufficient for all realistic input sizes. Importantly, since the time cost model is defined in terms of the evaluation semantics, the time cost of the statically unfolded program is no different from the dynamically unfolded program. I.e. unvisited depths in the recursion do not count towards the time cost.

The cost model is almost realized by fist compiling μNESL programs to flat vector code. The compilation employs flattening as described previously in Section 1.1.5. The vector code is then evaluated eagerly.

The reason why the cost model is not fully realized, is that the cost model essentially assumes MIMD execution, which vector code does not provide. Consider the costing of apply-to-each in Figure 1.4. It stipulates that the depth of the evaluation of an apply-to-each is the maximum depth of each sub-evaluation. If the body expression contains conditionals, the compilation will generate vector instructions for each branch that are executed in separate steps and then combined in the end. In other words, the total number of steps is the sum of the steps in each branch and not the maximum. Turning the max into a sum in the cost model for apply-to-each, would be

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

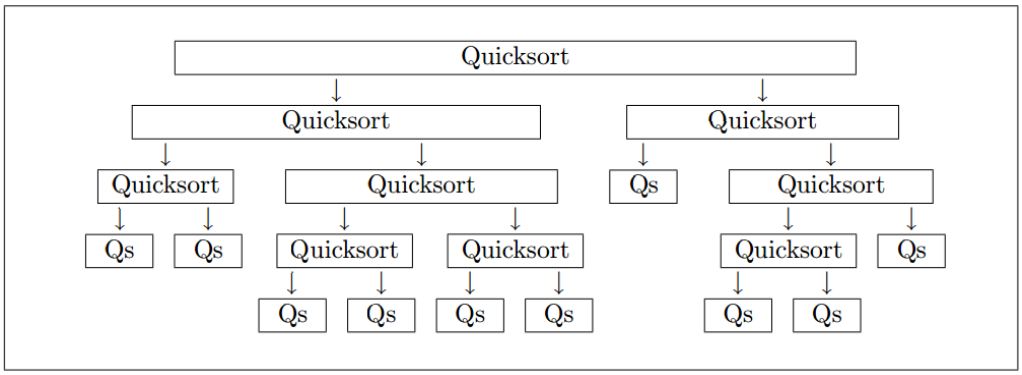


Figure 1.5: The quicksort algorithm implementation and call tree. The implementation and illustration is from [Ble95]. Just using parallelism within each block yields a parallel running time at least as great as the number of blocks (O(n)). Just using parallelism from running the blocks in parallel yields a parallel running time at least as great as the largest block (O(n)). By using both forms of parallelism the parallel running time can be reduced to the depth of the tree (expected O(lg n)).

overly pessimistic as that would stipulate that there is no parallelism at all in an apply-to-each, which is not the case. A general fix that allows proper SIMD execution costing, would require keeping track of all execution paths. Such a cost model quickly becomes complicated for larger programs, and is consequently not suitable as an intuitive cost model for the programmer.

There is one redeeming point however. Like the if-then-else in the quicksort example (Figure 1.5), conditionals in parallel algorithms are often the distinction between a base case and a recursive case. The base case is often of constant depth. If at most one branch has non-constant depth, the problem goes away as maximum and summation becomes asymptotically the same operation. Such a program is called *contained* by Blelloch in [Ble95]. In this dissertation, we keep in mind (i.e. completely ignore) that the cost model only works for the flattening transformation in the case of contained programs.

1.3.1 Virtual Segment Descriptors

Another issue with the cost model is the cost of distributing constants over a comprehension. Variable lookup is assigned a work-cost and a step-cost of 1. This correctly accounts for the distribution of variables of primitive type. For example, in the flattened version of

$$\textbf{let } c = 42 \textbf{ in } \{c + x : x \textbf{ in } \&(100)\},$$

c is distributed over the 100 instances of x, which is necessary before the lifted version of $+$ can be performed. This action should take 1 step and 100 work, which the cost model accounts for through the costing rules of variable lookup and apply-to-each. The problem arises when a non-primitive value is distributed. For example, consider the expression

$$\textbf{let } c = \&1000 \textbf{ in } \{c \,!\, x : x \textbf{ in } \&(100)\}.$$

Here, similarly, c must be distributed over the 100 instances of x in order to perform the lifted version of indexing. Using segment lengths as segments descriptors for nested sequences as described in Section 1.1.5, the only way to represent the distribution of c, would be to copy all 1000 elements a hundred times. To see why, consider the representation of c:

$$\begin{array}{l}([1000], \\ \ [0, ..., 999]).\end{array}$$

This value distributed a thousand times (named c') would then be represented by:

$$\begin{array}{l}([100],\\ (\overbrace{[1000,...,1000]}^{100},\\ \ [0,...,999,...,0,...,999])).\end{array}$$

Here, the length of the data vector (the bottom-most vector) is $100 \cdot 1000$, which is much greater than what can be computed using 100 units of work. Clearly, either the cost model is significantly flawed, or the chosen representation of nested sequences is insufficient.

A solution is to use a different segment descriptor representation known as *virtual* segment descriptors. This representation – and *scattered* segment descriptors, which we shall see in a moment – was first presented by Ben Lippmeier et al. [LCK+12]. Apart from segment lengths, virtual segment descriptors also has an offset into the underlying segmented vector. In this way, segments can overlap and the same segment can be repeated without copying the underlying data. This allows us to represent c' as:

$$\begin{array}{l}([(0,100)],\\ (\overbrace{[(0,1000),...,(0,1000)]}^{100},\\ \ [0,...,999]))\end{array}$$

Here, all the segments draw from the same segment in the data vector (offset 0) and the length of the data vector is 1000. More importantly, there is no work involved in computing it; it is identical to the data vector in c. The only work required is computing the vector

$$\overbrace{[(0,1000),...,(0,1000)]}^{100},$$

which can be done in 100 work assuming the size of an element in a segment descriptor is 1 even though it is a pair. This assumption is no different than assuming booleans and reals have the same size.

Virtual segment descriptors complicate some of the lifted operations on nested sequences. In particular, the lifted sequence constructor (or append if that was the primitive) becomes impossible to implement in the cost-prescribed way. Lifted append takes two sequences of the same length where the elements are sequences themselves and the same type. It the merges the two sequences by appending the element of the two sequences

pointwise. In the virtual segment descriptor representation, this translates to a situation where we have two values represented by (sd_1, a) and (sd_2, b), and we must combine sd_1 and sd_2 by pointwise addition and interleave the elements of a and b (which may themselves be trees of segment descriptors and data vectors).

We could implement the operation by appending a and b appending sd_1 and sd_2, and then adjust the segment offsets from sd_2 by the length of a. However, due to virtual segmentation, a and b may be much larger than the values the represent, so this could be much more costly than the cost model prescribes because the cost model only operates on the represented high-level values. Low-level values that are much larger than the values they represent arise in expressions like indexing sequences of sequences, where we cannot afford to copy the whole indexed sequence but simply index the segment descriptor and keep the data vector as it is.

Another approach could be to do a normalization step that converts virtual segment description into an equivalent non-virtual length-based segment description. This approach fails because it can cause the distribution problem (which virtual segment descriptors were originally designed to solve) to manifest [Mad12].

To achieve a full work-efficient representation, it is necessary to employ a more elaborate segment descriptor representation known as scattered segment descriptors. Scattered segment descriptors are virtual segment descriptors that furthermore allows the segments to be located in different data vectors. In this dissertation, we leave scattered segment descriptors for future work, and employ virtual segment descriptors and consider the sequence constructor to be more expensive than it ideally could be. We note that the actual work cost semantics is non-trivial to define as it depends on the representation and not on the high-level value.

1.4 Contributions

The main contributions of this dissertation are:

- The formal design of a high-level programming language for (nested) data-parallel streaming:
  - The formal distinction between streamed and manifest collections, and the classification of their relationship.
  - A cost model for space usage (a cost model for time also exists, but it is not a contribution of this dissertation).
  - An implementation strategy:

    - A low-level dataflow language based on stream chunking.
    - A (presumably) cost-preserving translation from the high-level language to the low-level language, which includes a well-known flattening transformation adopted to streams.

- Two concrete implementations:

    1. Streaming Accelerate, which implements regular streaming of multi-dimensional arrays on GPUs.
    2. Streaming NESL, which implements general streaming of nested data parallelism on multicores with vector instructions.

- A number of experimental benchmarks.

1.5 Terminology

The terms sequences, arrays, vectors and streams all refer to an enumerated collection of objects that allows repetitions. In all cases, the objects of a collection must all belong to the same *type*. As a general rule, arrays and vectors emphasize a spatial collection, streams emphasize a temporal collection, and for sequences, it depends on the context.

Sequence Sequence is the high-level mathematical term. It is the term we expose to the programmer in the front-end languages.

The word sequence is used to describe the primary collection type in NESL, which is actually implemented as a spatial collection.

In SNESL and Streaming Accelerate, sequences are characterized by being elementwise fully *streamable*.

Array In Accelerate, array refers to the primary collection type: a spatial finite multi-dimensional grid of primitive values (i.e. scalars). In the one-dimensional case, an array is also referred to as a vector. n-dimensional arrays, where $n > 1$, may be thought of as being nested arrays of $n - 1$ dimensions. However, the nesting is always *regular*, meaning that sub-arrays must have the same length.

In other contexts, arrays refer to a fixed-sized contiguous region of memory (e.g. an array in C).

Vector In NESL terminology, vectors refers to the low-level implementation model where programs are compiled to VCODE (short for vector code). As such, vectors are flat implementation-specific data structures, usually realized as low-level arrays.

In SNESL terminology, both sequences and vectors are high-level concepts. Whereas sequences are streamable, vectors are not. This means that vectors supports random-access, but have worse space cost semantics.

Stream In this dissertation, streams are the low-level implementation-specific data structure for sequences. They can be thought of as a *buffer* in memory that holds some of the value of the stream. Over time, the buffer is incrementally updated, so that in the end, all values will have been in the buffer.

Both sequences and vectors in SNESL are implemented with streams. Sequences use bounded buffering, while vectors use unbounded (more precisely, runtime-bounded) buffering.

1.6 Road Map

Chapter 2: Towards a Streaming Model for Nested Data Parallelism [MF13] Here, we introduce the syntax and semantics of SNESL, as well as the indented implementation model based on chunked streaming. We also give hand-written GPU timings. However, we do not pursue an actual GPU implementation.

Chapter 3: Functional Array Streams [MCECK15] Here, we explore an implementation of a streaming model on GPUs for the programming language Accelerate; a data-parallel language, similar to NESL, but based on regularly shaped multi-dimensional arrays instead of nested vectors. Consequently, Accelerate does not offer true nested data parallelism.

Chapter 4: Streaming Nested Data Parallelism on Multicores This chapter is a based on [MF16]. Here, we return to SNESL to give a full working implementation of streaming nested data parallelism. We implement a backend based on multicores with vector instructions, and measure actual performance numbers.

Chapter 2

Towards a Streaming Model for Nested Data Parallelism

This chapter is based on [MF13].
Sections with non-trivial changes or additions are marked with a ⋆.

Abstract

The language-integrated cost semantics for nested data parallelism pioneered by NESL provides an intuitive, high-level model for predicting performance and scalability of parallel algorithms with reasonable accuracy. However, this predictability, obtained through a uniform, parallelism-flattening execution strategy, comes at the price of potentially prohibitive space usage in the common case of computations with an excess of available parallelism, such as dense-matrix multiplication.

We present a simple nested data-parallel functional language and associated cost semantics that retains NESL's intuitive work–depth model for time complexity, but also allows highly parallel computations to be expressed in a space-efficient way, in the sense that memory usage on a single (or a few) processors is of the same order as for a sequential formulation of the algorithm, and in general scales smoothly with the actually realized degree of parallelism, not the potential parallelism.

The refined semantics is based on distinguishing formally between fully materialized (i.e., explicitly allocated in memory all at once) *vectors* and potentially ephemeral *sequences* of values, with the latter being bulk-processable in a streaming fashion. This semantics is directly compatible with previously proposed piecewise execution models for nested data parallelism, but allows the expected space usage to be reasoned about directly at the source-language level.

The language definition and implementation are still very much work in progress, but we do present some preliminary examples and timings, suggesting that the streaming model has practical potential.

2.1 Introduction

A long-standing goal in high-performance computing has been to develop a programming notation in which the inherent parallelism in regular data-processing tasks can be naturally expressed (also by domain specialists, not only trained computer scientists), and gainfully exploited on today's and tomorrows hardware. The functional paradigm has shown particular promise in that respect, being close to mathematical notation, and focusing on *what* is to be computed, rather than *how*. In particular, computations expressed purely functionally are naturally deterministic.

However, a good programming notation should also enable the programmer to predict, with reasonable accuracy, what kind of performance to expect from a particular way of expressing a calculation. For sequential languages, even (eager) functional ones, it is usually fairly easy to deduce the asymptotic time and space behavior of an algorithm, at least for the purpose of choosing between different alternatives; indeed, elementary complexity analysis is routinely taught in undergraduate classes. However, for parallel computations, the programmer has often been at the mercy of the compiler: sometimes an innocuous-looking change in the concrete expression of an algorithm may have drastic performance implications (in either direction).

The NESL language [Ble92] was a breakthrough not only in offering a concise, platform-independent notation for expressing complex, multi-level parallel algorithms in functional style, but perhaps even more so for offering an intuitive, language-integrated cost model to the programmer. The model allows one to derive expected work and depth complexities of a high-level parallel algorithm in a structural way, with effort comparable to that for a purely sequential language.

The NESL compilation model is centered around a relatively simple and predictable "flattening" translation to a uniform, low-level implementation language based on segmented prefix sums (scans) of flat vectors [BCH+94]. This means that, from the derived high-level parallel costs assigned by the model, one can immediately obtain a fairly reliable prediction of the expected concrete performance of the program, and especially how it will scale with increasing number of processors.

However, a substantial weakness in the NESL model is that, while time complexities of most algorithms are usually close to what would be intuitively expected, having flat vector operations as the only vehicle for express-

ing parallelism means that many "embarrassingly parallel" computations (say, matrix multiplication), when naturally expressed in the language, will uniformly allocate space proportional to the available parallelism (for instance, allowing for up to n^3 independent scalar multiplications when multiplying two n-by-n matrices), even if the available computation resources are nowhere near sufficient to exploit this parallelism. Consequently, programmers are often forced to explicitly sequentialize their code, to avoid prohibitive – or at least *embarrassing* – space usage. In other words, the plain NESL model effectively penalizes code that exposes "too much" parallelism.

For an even simpler example, consider the problem of computing $\Sigma_{i=1}^{n} \log i$ ($= \log n!$), where n is on the order of 10^9. In NESL, this computation would be naturally expressed as

$$logsum(n) = sum(\{log(float(i)) : i \textbf{ in } [1:n]\}),$$

with work $O(n)$, and depth $O(1)$.[1] Since the depth is negligible in comparison to the work, for all realistic numbers of processors p, we expect the computation time to be $O(n/p)$, which is as good as could be hoped for. But conversely, the computation will conceptually allocate and traverse $O(n)$ space, even when $p = 1$.

Of course, the NESL cost model does not *force* the compiler to naively allocate gigabytes of space for the above computation. For example, a 4-core back-end is perfectly allowed to divide the range into 4 equal parts, let each core compute the corresponding subrange sum, and then sequentially add up the 4 final results. This still achieves very close to a 4-times speedup over the sequential code, with negligible memory use. But relying on the compiler to be clever in such cases means that the programmer effectively has no reliable mental model of how much memory a conceptually low-space algorithm can be expected to use under any given circumstances. Worse, the space usage may be subtly context dependent: maybe the obvious optimization will be performed at the top level, but not inside another, already parallel computation with varying subproblem sizes, such as

$$sum(\{logsum(n * n) : n \textbf{ in } [1:1E3]\}).$$

We aim to refine the NESL language cost model so that, in addition to determining meaningful depth and work complexities, the space usage will also reflect what is intuitively truly required for execution – without sacrificing platform independence and the efficient, vector-based implementation

[1] In the NESL cost model, the logarithmic depth of the summation tree is accounted for in the mapping to a PRAM model, not in the source-level depth. This way, many hidden administrative tasks, such as data distribution, can also be given depth 1, simplifying the calculations considerably. But even if *sum* were computed with an explicit parallel algorithm, it would only have depth $O(\log n)$.

model. We do this by explicitly introducing the notion of *streaming* at the language level.

Streaming A key feature of NESL and similar languages is that the linked-list datatype commonly used to express bulk operations, such as maps or folds in functional settings, is replaced by a type of immutable arrays with constant-time access to arbitrary elements. This is done not so much to to accommodate algorithms that do need truly random access to individual elements (though those are important too), but mainly for two reasons:

1. To allow all processors to immediately get to work on pieces of large problems. For example, adding two billion-element vectors elementwise has no inherent inter-element dependencies; but if the vectors were represented as linked lists that each had to be traversed, this traversal would represent a major sequential bottleneck.

2. To ensure spatial locality and compactness, in particular to fully utilize cache lines, and allow meaningful prefetching of data from main memory. While (1) above could largely be achieved by some kind of indexing superstructure (e.g., a balanced binary tree with pointers to equal-length segments of the lists), gathering each processor's assigned work from all over memory would still represent a significant overhead.

However, full random-access vectors are actually overkill for many applications, such as vector addition. In principle, one could achieve most of goals (1) and (2) by segmenting the vectors into individually allocated *chunks* (of size anywhere from a few hundreds to a few millions elements), with the additions within a pair of chunks performed in parallel, but with the chunks themselves still processed sequentially. (Indeed, if the vectors are so large that they do not fit into main memory at all, but must be read in from auxiliary storage, such a chunked implementation is what the programmer has to code explicitly.)

Of course, an appropriate chunk size depends heavily on the platform, and we do not want to force programmers to commit to any particular size in the code: they should merely express the computational task in a way that is conductive to streaming, and the compiler should take care of the rest.

Returning to the sum-of-logsums example (and ignoring that some of the computations could obviously be shared), if the chunk size is, for instance, 10^3, then the early chunks will cover the computations of $logsum(n^2)$ for multiple n's (1 through 13 plus most of 14 for the first one), while the late chunks will each just cover part of $logsum(n^2)$ for a single n (the last n takes

10 chunks). This would be considerably more awkward to express if one were doing the problem partitioning manually.

Related work The space usage of flattening-based implementations of nested data-parallel algorithms has long been recognized as a problem. In the standard implementation of the NESL front end [BCH+94] (and apparently inherited in both direct derivatives such as NESL-GPU [BR12], and reimplementations such as CuNesl [ZM12]), the most immediately apparent problem arises from the excessive distribution of large vectors across parallel computations. It is ameliorated by an explicit parallel fetch, such that $\{v[i] : i \textbf{ in } a\}$ can be considerably more efficiently expressed as $v \rightarrow a$. This performance anomaly is also relatively easy to fix by a refined flattening translation, such as the one in Proteus [PPW95], or in recent versions of Data Parallel Haskell [LCK+12]. However, neither of these approaches addresses the more general problem of sequences always being fully represented in memory at once.

In particular, Blelloch and Greiner's space-efficient model implementation of NESL [BG96] takes a materializing semantics of sequences as the sequential baseline, and establishes that a parallel implementation does not need that much *additional* space to achieve speedups. (This is reasonable, since the available NESL operations on sequences, such as random-access indexing, in general force them to be materialized, in order to achieve the work complexity predicted by the model.) However, it does not flatten *nested* sequence constructions, keeping space usage reasonable in, e.g., the sum-of-logsums problem or the naive n-body algorithm. The downside is that the execution model requires more general task-level parallelism, not immediately realizable on a SIMD machine, or even on a vector-oriented GPU. It also relies on a fairly sophisticated garbage collector, working efficiently at low granularities. In contrast, we propose a language model that identifies streamable computations already at the source level, assigning them much lower sequential space costs. With this refinement, the uniform parallelism-flattening approach can still be employed, with all computations and allocations/deallocations performable in bulk.

Subsequent work on space costs of parallel functional programs has also tended to focus less on data parallelism, and more on general task parallelism. In particular Spoonhower et al. [SBHG08], building on the work by Blelloch and Greiner, extend the deterministic parallelism model and cost semantics to *futures*, but further deemphasize SIMD-like execution models. Futures allow streaming computations (which fall outside the strictly nested parallelism model) to be expressed, along with much more general computation structures. In contrast, we use a rather modest generalization

of nested parallelism by modeling streams of unbounded length as conceptually existing all at once, but only being materialized a fragment at a time.

Finally, our back-end execution model is similar to *piecewise* execution of flattened data-parallel programs [PPCF95], which also focuses on reducing the space usage in a data-parallel setting. The main difference is that we expose the streamability potential also in the source cost model. Our initial timing experiments suggest that piecewise execution is still relevant as an execution model is on modern platforms (GPGPUs), perhaps even more so than on the hardware of the mid-1990s.

2.2 A Simple Language with Streamed Vectors

In this section we present a minimalistic, expression-oriented core language for expressing nested data-parallel computations (only). For the purpose of defining the semantics, the language is slightly more explicit than one would expect from a practically usable notation. Programs written in an end-user language (such as NESL) would be desugared and elaborated into our notation, possibly with the default being the fully materializing elaboration, but allowing the programmer to express others by suitable syntax extensions.

Throughout this section we will use the convention that the metavariable k, when used as a length, ranges over "small" natural numbers (typically related to static program sizes), while l ranges over "potentially large" numbers (related to runtime data sizes).

2.2.1 Syntax and Informal Semantics

Types and values The language is first-order and explicitly typed, with a grammar of types (in Haskell-style notation):

$$\begin{aligned} \pi &::= \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Real} \mid \cdots \\ \tau &::= \pi \mid (\tau_1, ..., \tau_k) \mid [\tau] \\ \sigma &::= \tau \mid (\sigma_1, ..., \sigma_k) \mid \{\sigma\} \end{aligned}$$

Here π represents some fixed collection of primitive types. τ is the grammar of *concrete* types, the values of which are always fully materialized in memory. In particular, *vectors* $[\tau]$ provide constant-time read access to arbitrary elements. (Vectors of vectors may be jagged; there is no requirement that they represent proper matrices.)

More unconventionally, σ is the grammar of *general*, or *streamable*, types, which adds *sequence* types $\{\sigma\}$. Unlike vectors, sequences do not have to be fully represented in memory at the same time, and do not provide random access to elements. However, just like vectors, they have a strict semantics,

and every sequence will always be fully computed (exactly once) in a program execution; this is essential to allow chunked processing of sequences while presenting a chunk-size indifferent cost model to the programmer. Note that sequences may contain vectors, but not the other way around.

The *intensional* semantics of sequences could be summarized as "strict but lazy". Just as in NESL, sequences are always fully evaluated, whether their values are needed or not. However, in SNESL this evaluation may happen incrementally, with the sequence being produced and consumed in chunks. As the chunking is completely transparent to the programmer, we impose an exactly-once semantics in order to maintain a deterministic and predictable value and cost model: with a purely demand-driven, Haskell-stream-like semantics of sequences, we would simply discard a failing or very expensive computation in a part of the sequence that was never requested. But since each individual chunk is always fully evaluated for uniformity, the observable behavior in such cases would ultimately depend on the chunk size.

The *values* are as follows:

$$\begin{aligned} a &::= \mathsf{T} \mid \mathsf{F} \mid n\ {\scriptstyle (n\in\mathbf{Z})} \mid r\ {\scriptstyle (r\in\mathbf{R})} \mid \cdots \\ v &::= a \mid (v_1, ..., v_k) \mid [v_1, ..., v_l] \mid \{v_1, ..., v_l\} \end{aligned}$$

Here, a are the atomic values of the relevant primitive types. Values are typed in the obvious way.

Expressions The expression language is syntactically very similar to a NESL subset; the main difference is in the refined typing of the constructs and built-in operations. The raw grammar is quite minimal, as follows:

$$\begin{aligned} e &::= x \mid a \mid (x_1, ..., x_k) \mid x.i \mid \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 \mid \phi(x) \\ &\quad \mid \{e_0 : x\ \mathbf{in}\ x_0\ \mathbf{using}\ x_1, ..., x_k\} \mid \{e_0 \mid x_0\ \mathbf{using}\ x_1, ..., x_k\} \\ \phi &::= \text{(See Figure 2.2)} \end{aligned}$$

For simplicity, we require many subexpressions to be variables; more general expressions can be brought into the required form by adding **let**-bindings. (In larger examples we may assume that this let-insertion has been done automatically by a desugaring phase.)

The typing rules are given in Figure 2.1. They should be quite straightforward, except possibly the rules for comprehensions $\{\cdots\}$. In particular, in the explicit syntax, we require that all the auxiliary variables occurring free in the comprehension body (and representing values constant across all iterations) be explicitly listed. (Again, the list can be mechanically constructed by the desugarer, by simply enumerating the variables occurring free in e; the order is not significant.)

$\boxed{\Gamma \vdash e :: \sigma}$

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x :: \sigma} \qquad \frac{}{\Gamma \vdash \mathsf{T} :: \mathsf{Bool}} \quad \frac{}{\Gamma \vdash n :: \mathsf{Int}} \quad \frac{}{\Gamma \vdash r :: \mathsf{Real}} \quad \ldots$$

$$\frac{\left(\Gamma(x_i) = \sigma_i\right)_{i=1}^{k}}{\Gamma \vdash (x_1, ..., x_k) :: (\sigma_1, ..., \sigma_k)} \qquad \frac{\Gamma(x) = (\sigma_1, ..., \sigma_k)}{\Gamma \vdash x.i :: \sigma_i}\ (1 \le i \le k)$$

$$\frac{\Gamma \vdash e_0 :: \sigma_0 \quad \Gamma[x \mapsto \sigma_0] \vdash e_1 :: \sigma_1}{\Gamma \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 :: \sigma_1} \qquad \frac{\Gamma(x) = \sigma_1 \quad \phi :: \sigma_1 \to \sigma_2}{\Gamma \vdash \phi(x) :: \sigma_2}$$

$$\frac{\Gamma(x_0) = \{\sigma_0\} \quad \left(\Gamma(x_i) = \tau_i\right)_{i=1}^{k} \quad [x \mapsto \sigma_0, x_1 \mapsto \tau_1, ..., x_k \mapsto \tau_k] \vdash e :: \sigma}{\Gamma \vdash \{e : x\ \mathbf{in}\ x_0\ \mathbf{using}\ x_1, ..., x_k\} :: \{\sigma\}}\ (k \ge 0)$$

$$\frac{\Gamma(x_0) = \mathsf{Bool} \qquad \left(\Gamma(x_i) = \sigma_i\right)_{i=1}^{k} \qquad [x_1 \mapsto \sigma_1, ..., x_k \mapsto \sigma_k] \vdash e :: \sigma}{\Gamma \vdash \{e \mid x_0\ \mathbf{using}\ x_1, ..., x_k\} :: \{\sigma\}}\ (k \ge 0)$$

Figure 2.1: Typing rules

In the general form of comprehensions (with the "**in**" syntax), to preserve the invariant that sequences are only traversed once, any auxiliary variables must be of concrete type, i.e., materialized throughout the evaluation of the comprehension. The restricted form (with the "|" syntax) could be seen as abbreviating a general comprehension,

$$\{e \mid x_0\ \mathbf{using}\ \vec{x}\}\ \text{“}\equiv\text{”}\ \{e : \_\ \mathbf{in}\ iota(b2i(x_0))\ \mathbf{using}\ \vec{x}\}$$

where $b2i(\mathsf{F}) = 0$, $b2i(\mathsf{T}) = 1$, and $iota(n) = \{0, ..., n-1\}$. However, since e here will only be evaluated at most once, there are no restrictions on the types of the auxiliary variables.

Complementing the base syntax are the primitive operations in Figure 2.2. (We will usually write binary operators infix in concrete examples.) Most of these should be self-explanatory, with the following notes. The ellipses after "+" represents a collection of further basic arithmetic and logical operations, all with types of the form $(\pi_1, ..., \pi_k) \to \pi_0$. $mkseq^k$ constructs a length-k sequence; *empty* tests whether a sequence has zero length (but without traversing it otherwise); and *the* returns the sole element of a singleton sequence. $+\!+$ appends two sequences, and zip^k tuples up corresponding elements of k equal-length sequences. $flagpart$ chops a sequence

$\boxed{\phi :: \sigma_1 \to \sigma_2}$

$$
\begin{array}{rcll}
+ & :: & (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int} & \\
 & \vdots & & \\
mkseq^k_\sigma & :: & (\overbrace{\sigma, ..., \sigma}^{k}) \to \{\sigma\} & k{\geq}0 \\
empty_\sigma & :: & \{\sigma\} \to \mathsf{Bool} & \\
the_\sigma & :: & \{\sigma\} \to \sigma & \\
{+\!\!+}_\sigma & :: & (\{\sigma\}, \{\sigma\}) \to \sigma & \\
zip^k_{\sigma_1,...,\sigma_k} & :: & (\{\sigma_1\}, ..., \{\sigma_k\}) \to \{(\sigma_1, ..., \sigma_k)\} & k{\geq}1 \\
flagpart_\sigma & :: & (\{\sigma\}, \{\mathsf{Bool}\}) \to \{\{\sigma\}\} & \\
concat_\sigma & :: & \{\{\sigma\}\} \to \{\sigma\} & \\
iota & :: & \mathsf{Int} \to \{\mathsf{Int}\} & \\
tab_\tau & :: & \{\tau\} \to [\tau] & \\
seq_\tau & :: & [\tau] \to \{\tau\} & \\
length_\tau & :: & [\tau] \to \mathsf{Int} & \\
!_\tau & :: & ([\tau], \mathsf{Int}) \to \tau & \\
reduce_R & :: & \{\mathsf{Int}\} \to \mathsf{Int} & R{\in}\{+,\times,\mathrm{max},...\} \\
scan_R & :: & \{\mathsf{Int}\} \to \{\mathsf{Int}\} & R{\in}\{+,\times,\mathrm{max},...\}
\end{array}
$$

Figure 2.2: Primitive operations

into subsequences, e.g.,

$$
\begin{array}{l}
flagpart(\{3,1,4,1,5,9\}, \{\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{T},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T}\}) = \\
\quad \{\{3,1,4\}, \{\}, \{1\}, \{5,9\}\}\,.
\end{array}
$$

(The flag sequence must end in a T, and the number of F's sequence must match the number of elements in the data sequence.) Conversely, *concat* appends all subsequenes into one.

Finally, *tab* tabulates and materializes a sequence into a vector, while *seq* streams the elements of a vector as a sequence. *length* returns the length of a vector; and element indexing, !, is zero-based. $reduce_R$ computes the R-reduction of sequence elements (where R ranges over a fixed collection of basic monoids R), while $scan_R$ computes the *exclusive* scan (all proper-prefix reductions), e.g., $scan_+(\{3,5,4,2\}) = \{0,3,8,12\}$.

In the actual implementation, we make available a number of short-hands. First, as already mentioned, the front-end automatically performs **let**-insertions where general expressions are used instead of variables, and computes the auxiliary-variable lists in comprehensions. It also infers the

type subscripts on primitive operations. Further, we allow pattern-matching bindings on the left-hand side of $=$ and **in**, so that, e.g.,

$$\begin{aligned}&\textbf{let } (x, y) = e \textbf{ in } e' \equiv \\ &\quad \textbf{let } p = e \textbf{ in let } x = p.1 \textbf{ in let } y = p.2 \textbf{ in } e'\,,\end{aligned}$$

where p is a fresh variable. Likewise, we allow comprehensions to traverse several sequences of the same length simultaneously,

$$\begin{aligned}&\{e : x_1 \textbf{ in } e_1; \ldots; x_k \textbf{ in } e_k\} \equiv \\ &\quad \{e : (x_1, ..., x_k) \textbf{ in } zip(e_1, ..., e_k)\}\,.\end{aligned}$$

And we may combine general and predicated comprehensions:

$$\{e : x \textbf{ in } e_0 \mid e_1\} \equiv concat(\{\{e \mid e_1\} : x \textbf{ in } e_0\})\,,$$

where, naturally, any variable occurring free in e or e_1 must be of concrete type. Moreover, we allow sequence and vector constructions as abbreviations:

$$\begin{aligned}\{e_1, ..., e_k\} &\equiv mkseq^k(e_1, ..., e_k) \\ [e_1, ..., e_k] &\equiv tab(\{e_1, ..., e_k\})\,.\end{aligned}$$

Finally, note that the base language does not include an explicit conditional form. Instead, we can define it as:

$$\begin{aligned}&\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \equiv \\ &\quad \textbf{let } b = e_0 \textbf{ in } the(\{e_1 \mid b\} +\!\!+ \{e_2 \mid \neg b\})\,,\end{aligned}$$

This decomposition mirrors the data-parallel NESL computation model for conditionals occurring inside comprehensions: rather than alternating between evaluating e_1 and e_2 on a per-element basis, we first evaluate e_1 for the subsequence of elements where e_0 evaluates to T, then e_2 for those where e_0 evaluates to F, and finally merge the results.

Likewise, other useful functions can be efficiently (at least in an asymptotic sense) defined in terms of the given primitive ones. For example, we can compute the length of a *sequence*:

$$\begin{aligned}slength &:: \{\sigma\} \to \mathsf{Int} \\ slength(s) &= reduce_+(\{1 : \_ \textbf{ in } s\})\,.\end{aligned}$$

Some operations require a little more thought to express in a streamable way. For example, to tag each element of a sequence with its serial number, we cannot simply say,

$$\begin{aligned}number &:: \{\sigma\} \to \{(\sigma, \mathsf{Int})\} \\ number(s) &= zip(s, iota(slength(s)))\,,\end{aligned}$$

because that would require traversing s twice. Instead, we must say,

$$number(s) \quad = \quad zip(s, scan_{+}(\{1 : \_ \textbf{ in } s\}))\,.$$

(In fact, *iota* is nominally implemented in terms of a +-scan anyway, so the above solution is arguably more direct than explicitly computing the sequence length first.)

The language as presented does not provide for programmer-defined functions, so the definitions above must be thought of as notational abbreviations. True functions, possibly recursive, add another layer of complication – not so much in the high-level semantics, but more in the compilation and low-level execution model. For now, we have concentrated on the functionless fragment, since it already highlights most of the significant issues related to streaming.

Likewise, there is no notion of unbounded iteration (whether in the form of tail recursion or more explicitly), and hence potential divergence; but given the eager nature of the language, there should be no semantic problem with introducing potential non-termination. However, just like in Haskell, we are forced to – at least formally – identify all run-time errors (division by zero, indexing out of bounds, etc.) with divergence; if we distinguish between them, the language becomes formally nondeterministic: if it aborts with an error in one run, another run might diverge, or abort with a different error, depending on low-level scheduling decisions. We still guarantee, however, that if a run terminates with a non-error answer, all other runs will also terminate with that answer.

2.2.2 Value Size Model ⋆

The actual data representation is invisible to the programmer, and has no influence on the value semantics. However, in order to provide a reasonable model of the program's execution and resulting space-usage behavior, we do need to have a formal, asymptotically accurate, definition of the *size* of any particular value. In the streaming setting, we characterize the size as a pair of metrics, representing, respectively, the space required to process the value sequentially and in parallel.

More specifically, for any value v, we define $\mathcal{P}\|v\|$ as its parallel size, and $\mathcal{S}\|v\|$ as its sequential size, as follows:

$$\begin{aligned}
\mathcal{P}\|a\| &= 1 \\
\mathcal{P}\|(v_1, ..., v_k)\| &= \Sigma_{i=1}^{k} \mathcal{P}\|v_i\| \\
\mathcal{P}\|[v_1, ..., v_l]\| &= 1 + \Sigma_{i=1}^{l} \mathcal{P}\|v_i\| \\
\mathcal{P}\|\{v_1, ..., v_l\}\| &= 1 + \Sigma_{i=1}^{l} (1 + \mathcal{P}\|v_i\|)
\end{aligned}$$

$$
\begin{aligned}
\mathcal{S}\|a\| &= (1,0) \\
\mathcal{S}\|(v_1,...,v_k)\| &= (\Sigma_{i=1}^{k} M_i, \Sigma_{i=1}^{k} N_i) \\
&\quad \textit{where } \forall i.\, \mathcal{S}\|v_i\| = (M_i, N_i) \\
\mathcal{S}\|[v_1,...,v_l]\| &= (1 + \max_{i=1}^{l} M_i, \Sigma_{i=1}^{k} \mathcal{P}\|v_i\|) \\
&\quad \textit{where } \forall i.\, \mathcal{S}\|v_i\| = (M_i, \_) \\
\mathcal{S}\|\{v_1,...,v_l\}\| &= (1 + \max_{i=1}^{l} M_i, \max_{i=1}^{l} N_i) \\
&\quad \textit{where } \forall i.\, \mathcal{S}\|v_i\| = (M_i, N_i)
\end{aligned}
$$

The sequential size is divided in two natural numbers (M, N), where M represents *scalable* space that scales with the chunk size and N represents *vectorial* space; space that must manifest regardless of the chunk size. The sequential size without this distinction is simply $M + N$. On P processors, the actual size of v where $\mathcal{P}\|v\| = L$ and $\mathcal{S}\|v\| = (M, N)$ is supposed to be $O(\min(P \cdot M + N, L))$.

For simplicity, since we are mainly interested in asymptotic behavior, we consider all atomic values to require the same amount of space, though there wouldn't be any problem with accounting more precisely for space usage, so that, e.g., a Real would have a constantly larger size than a Bool.

For tuples, the arity k is statically known, and doesn't need to be explicitly represented at runtime at all, so the size of a tuple is simply the sum of sizes of the elements. In particular, empty tuples take truly zero space.

On the other hand, for vectors, the extra $1+$ represents the need to store the length of the vector somewhere, in addition to the element values. (This cost may be non-negligible for a nested vector type like [[Int]], especially if many of the inner vectors may be empty.) This cost mirrors the eventual concrete representation, where a nested vector is represented as a separate vector of subvector lengths and a vector of the underlying values.

The vectorial size of a vector is the sum of the parallel sizes of all its elements. This means that all elements must be allowed to exist at once, regardless of the number of processors available. The length of the vector is recorded in the scalable size (the 1+ term), which indicates that the top-most layer of vector lengths are streamed. If the vectors are small and the chunk size is large, it is necessary to manifest more than one vector at a time. This is reflected in the definition of the the scalable size as the largest scalable size of all the elements. Although we have already accounted for all the elements in the vectorial size, we must also include a scalable term to account for this case. In effect, the space an implementation is allowed to allocate for a stream of vectors, is the chunk size plus the length of the longest vector in the stream. A slightly more restricted implementation is possible that only allows the maximum of the chunk size and the longest vector to be allocated

(i.e. max instead of plus). However, that would require the chunk size to appear as a parameter to the cost model.

Finally, for sequences, the conceptual representation model is that segment boundaries are represented as flags marking the end of each subsequence. The reason for this difference from vectors is that, when streaming a sequence of subsequences, we do not know the length of each subsequence until *after* it has been generated. Also, since we want a faithful representation of consecutive empty sequences, we effectively represent a value of type $\{\{\pi\}\}$ as if it were $\{\pi + ()\}$, i.e., every element is either a data element or a subsequence terminator.

More fundamentally, sequences differ from vectors in that they are in general not materialized in memory all at once; in fact, for purely sequential execution, the are processed strictly one element at a time. Therefore, the sequential size of a sequence value is simply the size of its largest element, while the parallel size – where all elements are simultaneously available for processing – is the sum of the element sizes, just like for vectors.

2.2.3 Evaluation and Cost Model ⋆

We will now consider a big-step semantics of the language and primitive operations. As far as the computed result is concerned, one could simply erase the distinction between vectors and sequences, and even identify them both with simple ML-style lists. The parallel nature of the language, and the role of streaming and random-access indexing, is only made apparent through the cost semantics.

Since sequence values are not directly expressible as literals in the language (syntactic sugar notwithstanding), precluding a simple substitution-based semantics, we use a semantics in which open expressions are evaluated with respect to an environment ρ, mapping variables to their values. The form of the judgment is thus $\boxed{\rho \vdash e \Downarrow v \,\$\, \omega}$, where the *cost metric* ω is built as follows.

A metric is a 5-tuple of natural numbers, $\omega = (W, D; M, N; L)$, where the first two capture the standard *work* and *depth* cost of the computation. The former represents the total number of atomic (constant-cost) operations performed during the evaluation; it corresponds to the execution time on a single processor, T_1. The latter (also called the *span*, or *step* complexity) represents the the longest chain of sequential dependencies in the computation, thus representing how fast the evaluation could proceed with an unlimited number of processors, T_∞. Note that we will always have $W \geq D$, with the inequality being strict precisely when parallel evaluation is possible.

Like in NESL, the components of a tuple constructor – though nominally independent – are *not* considered to be evaluated in parallel (as far as the

cost model is concerned; an opportunistic compiler of course has the option of doing so anyway). This reflects our focus on data parallelism, where sequences are the only source of speedups. In particular, in an expression like $f(x_1) + f(x_2)$, the two f-computations would not be considered independent, but will be performed in sequence, and in particular with the depths summed. (In fact, in our restricted language, the addition will have to be explicitly let-sequenced anyway.)

If the programmer intends to actively exploit the parallelism in evaluating the summands independently, he can write instead,

$$\textbf{let } r = tab(\{f(x) : x \textbf{ in } \{x_1, x_2\}\}) \textbf{ in } r!0 + r!1\,.$$

This would most likely only be appropriate in the context of a recursive definition of f, so that the total available parallelism would increase drastically at each level of recursion.

The last components of the cost, dubbed *sequential* and *parallel space*, represent the maximal space usage during the computation, respectively corresponding to a fully sequential execution (i.e., S_1), and one exploiting the maximal number of processors (S_∞).

The parallel space L represents the space used in a fully eager evaluation of streams. This corresponds to the traditional VCODE implementation. Fully eager evaluation is necessary for step-efficiency in the case where the available parallel resources exceeds the potential parallelism in the expression.

Just like for the size model for values, the sequential space for computations is captured by two numbers N and M. N represent the total number of *active* streams. By active we mean non-empty. Non-empty streams are not charged a space cost, and should therefore not be allocated in an actual implementation. The reason why it is necessary to make this distinction is that branches in the computation that are never visited, should not count towards the space cost. This is particularly important in recursive definitions where the branches deeper than deepest path in the dynamic call tree are never visited. Although our language does not support recursion at this point, we do want to create a good cost model for statically unfolded recursion in the hope that the cost model will scale to dynamic unfolding. In a statically unfolded recursion, it would be inefficient to charge a space cost for each recursive step, when in practice, the evaluation might stop earlier than that. Just like for values, N is dubbed the scalable space, and each active stream are allowed to use up to the chunk size amount of space for its buffer.

On top of that, N represents the *vectorial space*, and can be thought of as the maximum size that all the unbounded buffers in the computation may

take. This is the amount of space that a computation may require besides the scalable space. Crucially, this metric is invariant to the chunk size as we do not want to charge P times the length of a very long vector. Not only would that be overly pessimistic, it may even cause the sequential space to vastly exceed the parallel space. Unbounded buffers arise in operations on vectors, and the vectorial space of a computation is a metric that reports the sum of the largest vectors in all its sub-computations. Sub-computations that do not involve vectors, simply have a vectorial space of zero. In practice, it is not necessarily the case that the longest vector of all sub-computation are manifest at the same time, so our space cost model is an over-approximation. However, since our model is not parameterized by the actual chunk size, it would be impossible to give a more precise measure.

The evaluation rules are given in Figure 2.3. Note that variable accesses are themselves considered free wrt. time (the cost is assigned to the computations using the variable's value). Tuple construction and component selection costs are also considered negligible (since they don't actually perform any extra data movement at runtime in our implementation model), but literals do have unit cost.

More interestingly, in let-bindings, both work and depth costs of the subexpression evaluations are summed, reflecting strict sequential evaluation of e_0 and e_1. But for space usage, the parallel space used to evaluate the let-expression is the maximum of two numbers: the space used to evaluate e_0, and the sum of the size of e_0's value and the space needed to evaluate e_1. This choice reflects that the lifetime of a variable is limited to the scope of its let-binding. The reason for using summation in the sequential space in let-bindings, is that in a dataflow execution model, operations must be allowed to execute partially and then be suspended in order to allow other operations to execute. For let-bindings, this means that values may still be needed in future execution when executing parts outside the scope of the variable. It may happen that e_1 must take a step before e_0 can continue, even though some of the temporary values in e_0 are not fully used. It would therefore be incorrect to require all temporary storage to be deallocated in e_0 before we allow e_1 to execute, which is precisely what the parallel space stipulates.

Note that let-bindings, though commutative wrt. value and time costs are not so wrt. space costs. That is, in an expression,

$$\textbf{let } x_1 = e_1 \textbf{ in let } x_2 = e_2 \textbf{ in } (x_1, x_2)\,,$$

as long as x_1 does not occur in e_2 and vice versa, the order of the bindings does not matter for the result value, or work and depth. However, if e_1 returns a small result but uses much temporary space, while e_2 requires little

$\boxed{\rho \vdash e \Downarrow v \,\$\, (W, D; M, N; L)}$

$$\frac{\rho(x) = v}{\rho \vdash x \Downarrow v \,\$\, (0,0;0,0;0)}$$

$$\frac{}{\rho \vdash a \Downarrow a \,\$\, (1,1;1,0;1)}$$

$$\frac{\left(\rho(x_i) = v_i\right)_{i=1}^{k}}{\rho \vdash (x_1, ..., x_k) \Downarrow (v_1, ..., v_k) \,\$\, (0,0;0,0;0)}$$

$$\frac{\rho(x) = (v_1, ..., v_k)}{\rho \vdash x.i \Downarrow v_i \,\$\, (0,0;0,0;0)}$$

$$\frac{\begin{array}{l}\rho \vdash e_0 \Downarrow v_0 \,\$\, (W_0, D_0; M_0, N_0; L_0) \\ \rho[x \mapsto v_0] \vdash e_1 \Downarrow v_1 \,\$\, (W_1, D_1; M_1, N_1; L_1)\end{array}}{\begin{array}{l}\rho \vdash \textbf{let } x = e_0 \textbf{ in } e_1 \Downarrow v_1 \,\$ \\ \quad (W_0 + W_1, D_0 + D_1; M_0 + M_1, N_0 + N_1; \max(L_0, L_1 + \mathcal{P}\|v_0\|))\end{array}}$$

$$\frac{F_\phi(\rho(x)) = (v, W)}{\rho \vdash \phi(x) \Downarrow v \,\$\, (W, 1; \mathcal{S}\|v\|; \mathcal{P}\|v\|)}$$

$$\frac{\rho(x_0) = \{v_1, ..., v_l\} \quad \left(\rho[x \mapsto v_i] \vdash e \Downarrow v_i' \,\$\, (W_i, D_i; M_i, N_i; L_i)\right)_{i=1}^{l}}{\begin{array}{l}\rho \vdash \{e : x \textbf{ in } x_0 \textbf{ using } x_1^{\tau_1}, ..., x_k^{\tau_k}\} \Downarrow \{v_1', ..., v_l'\} \,\$ \\ \quad ((l+1) \cdot \Sigma_{i=1}^{k}|\tau_i| + \Sigma_{i=1}^{l} W_i,\ \Sigma_{i=1}^{k}|\tau_i| + \max_{i=1}^{l} D_i; \\ \quad\ 1 + \Sigma_{i=1}^{k}|\tau_i| + \max_{i=1}^{l} M_i, \max_{i=1}^{l} N_i; l + l \cdot \Sigma_{i=1}^{k}|\tau_i| + \Sigma_{i=1}^{l} L_i)\end{array}}$$

$$\frac{\rho(x_0) = \mathsf{F}}{\rho \vdash \{e \mid x_0 \textbf{ using } x_1^{\sigma_1}, ..., x_k{}^{\sigma_k}\} \Downarrow \{\} \,\$\, (\Sigma_{i=1}^{k}|\sigma_i|, \Sigma_{i=1}^{k}|\sigma_i|; 1, 0; 1)}$$

$$\frac{\rho(x_0) = \mathsf{T} \qquad \rho \vdash e \Downarrow v \,\$\, (W, D; M, N; L)}{\begin{array}{l}\rho \vdash \{e \mid x_0 \textbf{ using } x_1^{\sigma_1}, ..., x_k{}^{\sigma_k}\} \Downarrow \{v\} \,\$ \\ \quad (\Sigma_{i=1}^{k}|\sigma_i| + W,\ \Sigma_{i=1}^{k}|\sigma_i| + D; 1 + \Sigma_{i=1}^{k}|\sigma_i| + M, N; 1 + \Sigma_{i=1}^{k}|\tau_i| + L)\end{array}}$$

Figure 2.3: Evaluation semantics with costs

space beyond the large result it allocates, the above sequencing is preferable to the one with the bindings of x_1 and x_2 swapped.

The value and cost of primitive operations ϕ are given by an auxiliary function F_ϕ. The value returned should be immediate from the informal semantics of the operations. As previously mentioned, we consider the depth to always be 1, even for operations like *reduce*. The work can be taken to be simply the (parallel) size of the result in all cases except for *the* and *zip*, which perform no work; ! which has unit cost; and *concat* and *reduce* whose work is proportional to the length of the input sequence.

Finally, for sequence comprehensions (general or restricted), work and depth costs of the body computations are combined in the expected way, but with the addition of explicit distribution or packing costs for the auxiliary variables. (For notational simplicity, we have assumed that all such variables have been annotated by their types in the **using**-clause.) Also, the space costs exhibit a difference between the sequential and parallel cases analogous to the one for value sizes. For the space costs, the per-element size of a type, $|\sigma|$, is given by:

$$\begin{aligned} |a| &= 1 \\ |(\sigma_1, ..., \sigma_k)| &= \Sigma_{i=1}^{k} |\sigma_i| \\ |\{\sigma\}| &= |\sigma| + 1 \\ |[\tau]| &= 1 \end{aligned}$$

Note that this is different from the sizes of values of that type: since sequences are never copied, and vectors in the implementation are copied as pointers, their actual lengths don't matter.

2.3 Implementation Model

Much like the source language refines NESL, the implementation model is also an extension of NESL's parallelism-flatting approach, in that the two effectively coincide in the case of fully materialized vectors, but we have a more space-efficient model for implementing sequences, including sequences of vectors.

For sequences, our model is conceptually similar to that of *piecewise execution* [PPCF95], in which long sequences are broken up into fixed-sized chunks (which may cross segment boundaries). Each chunk is then processed using all available computation units, and the chunks are processed sequentially using a dataflow model.

The main difference in our model is that the chunking (but not the chunk size!) is exposed at the source level in the type system and cost model, rather

than as an optimized implementation strategy, whose applicability in any particular situation remains hidden to the programmer – except through sometimes drastic effects on performance or memory use. In particular, unlike transparent piecewise execution of NESL or Proteus programs, the compiler will never silently recompute a sequence if it needs to be traversed more than once; instead, the programmer must explicitly make the choice between materialization and recomputation based on the overall asymptotic-complexity requirements for time and space usage.

2.3.1 Data Representation

In a bit more detail, all values are represented as trees of low-level, flat *streams* of primitive values. Writing SA for the set of finite streams of A-elements, we interpret source-language types as follows:

$$\begin{aligned}
[\![\mathsf{Bool}]\!] &= S\mathbf{B} \\
[\![\mathsf{Int}]\!] &= S\mathbf{Z} \\
[\![\mathsf{Real}]\!] &= S\mathbf{R} \\
[\![(\sigma_1, ..., \sigma_k)]\!] &= [\![\sigma_1]\!] \times \cdots \times [\![\sigma_k]\!] \\
[\![[\tau]]\!] &= [\![\tau]\!] \times (S\mathbf{N} \times S\mathbf{N}) \\
[\![\{\sigma\}]\!] &= [\![\sigma]\!] \times S\mathbf{B}
\end{aligned}$$

Tuples are just cartesian products. For vectors, we augment the interpretation of the base type with a *generalized segment descriptor* describing starts and lengths of the vectors. In the *canonical* representation, the segments are allocated contiguously, and so the starting positions are simply given as the +-scan of the lengths. For example, writing streams between $\langle \cdots \rangle$, and using $\triangleleft$ for the "is represented as" relation, we have:

$$\begin{aligned}
&[[3,1,4],[],[1],[5,9]] \triangleleft \\
&\quad ((\langle 3,1,4,1,5,9\rangle, (\langle 0,3,3,4\rangle, \langle 3,0,1,2\rangle), (\langle 0\rangle, \langle 4\rangle))
\end{aligned}$$

However, we also allow the subvectors to be permuted, allocated non-contiguously, or share data – even across segment boundaries. For example, the above nested vector could also be represented non-canonically as

$$((\langle 7,5,9,3,1,4\rangle, (\langle 3,0,4,1\rangle, \langle 3,0,1,2\rangle), (\langle 0\rangle, \langle 4\rangle))$$

(Note that the length stream is always the same as in the canonical representation.) More usefully, we can represent the vector of all prefixes or suffixes of another vector in linear, rather than quadratic space. The only well-formedness constraint is that each "slice" (determined by a corresponding (start,length) pair) has to fit entirely within the base vector.

This representation corresponds to Lippmeier et al.'s *virtual segment descriptors* [LCK+12], introduced to avoid the performance anomaly in code like $\{v!i : i \textbf{ in } a\}$ where the entire vector v is first distributed to all parallel computations, each one of which selects only a single element. By instead keeping track of segment starts and lengths separately (rather than uniquely determining the former by a +-scan of the latter), we can avoid duplicating the full data, but only the pointers. The price, of course, is the potential for read–read memory contention, but that will normally be a second-order effect compared to the performance impact on both time and space of proactive massive duplication.

(We do not presently use *scattered* segment descriptors, where different segments may also come from different base vectors, because the need for copying in appends is significantly reduced in our setting: it is only needed in the case where the concatenated sequence must ultimately be materialized.

For sequences, as previously mentioned, we represent subsequence boundaries as flags:

$$\{\{3,1,4\},\{\},\{1\},\{5,9\}\} \triangleleft \\ ((\langle 3,1,4,1,5,9\rangle, \langle \mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{T},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T}\rangle), \langle \mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T}\rangle)$$

Here, the representation is actually unique. It can be seen as a unary counterpart of the canonical vector representation (where the segment starts are redundant).

The explicit flag representation is for intended for interfacing between operations. When a bulk-processing a chunk, as in a segmented scan, we can coalesce consecutive T's in the flag vector to a simple count; then the segment-flag vector vector has exactly as many elements as data vector, and so the corresponding elements of both can be accessed in constant time. For example, the above sequence without the top-most segment descriptor can be represented in two forms:

$$\begin{array}{ll} \text{General form:} & (\langle 3,\ 1,\ 4,\ 1,\ 5,\ 9\rangle, \langle \mathsf{F},\ \mathsf{F},\ \mathsf{F},\ \mathsf{T},\ \mathsf{T},\ \mathsf{F},\ \mathsf{T},\ \mathsf{F},\ \mathsf{F},\ \mathsf{T}\ \rangle) \\ \text{Contracted form:} & (\langle 3,\ 1,\ 4,\ 1,\ 5,\ 9\rangle, \langle 0,\ 0,\ 2,\quad 1,\quad 0,\ 1\quad \rangle) \end{array}$$

The two forms are uniquely determined from each other, except in the case where the segment descriptor has at least one leading T. It is therefore also necessary to keep a separate count of the number of leading T's in the contracted form.

$$
\begin{array}{rcl}
[\![x]\!]\zeta\ s & = & \zeta\ x \\
[\![a]\!]\zeta\ s & = & rep\ s\ a \\
[\![(x_1,...,x_k)]\!]\zeta\ s & = & (\zeta x_1,...,\zeta x_k) \\
[\![x.i]\!]\zeta\ s & = & let\ (t_1,...,t_k) = \zeta\ x\ in\ t_i \\
[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]\zeta\ s & = & let\ t = [\![e_1]\!]\zeta\ s\ in\ [\![e_2]\!]\zeta[x \mapsto t]\ s \\
[\![\phi(x)]\!]\zeta\ s & = & [\![F]\!]_\phi(\zeta\ x)\ s \\
\multicolumn{3}{l}{[\![\{e_0 : x\ \textbf{in}\ x_0\ \textbf{using}\ x_1^{\tau_1},...,x_k^{\tau_k}\}]\!]\zeta\ s\ =} \\
\multicolumn{3}{l}{\quad let\ (t,s') = \zeta\ x_0\ in} \\
\multicolumn{3}{l}{\quad\quad ([\![e_0]\!][x \mapsto t, (x_i \mapsto dist_{\tau_i}\ (\zeta x_i)\ s')_{i=1}^k]\ (usum\ s'), s')} \\
\multicolumn{3}{l}{[\![\{e_0 \mid x_0\ \textbf{using}\ x_1^{\sigma_1},...,x_k^{\sigma_k}\}]\!]\zeta\ s\ =} \\
\multicolumn{3}{l}{\quad let\ s_0 = \zeta\ x_0, s' = b2u(s_0)\ in} \\
\multicolumn{3}{l}{\quad\quad ([\![e_0]\!][(x_i \mapsto pack_{\sigma_i}\ (\zeta x_i)\ s_0)_{i=1}^k]\ (usum\ s'), s')}
\end{array}
$$

Figure 2.4: Translation

2.3.2 Translation

In the actual implementation, we translate a nested data-parallel source program to a stream-manipulating target-language program in a low-level language. Here, for conciseness, we present the essence of the translation by directly interpreting each source-language term as the mathematical stream it denotes. The translation is given in Figure 2.4.

The semantics is compositional and type directed: For every $\Gamma \vdash e :: \sigma$, we have $[\![e]\!] : [\![\Gamma]\!] \to S\mathbf{1} \to [\![\sigma]\!]$, where $\zeta \in [\![\Gamma]\!]$ is a run-time environment mapping each variable x in $\mathrm{dom}(\Gamma)$ to a low-level stream tree in $[\![\Gamma(x)]\!]$. The meaning of a closed top-level expression e is then given by $[\![e]\!]\ []\ \langle * \rangle$. In general, the stream of dummy input values represents the parallelism degree of the computation, represented in unary because the length of a sequence is in general not known a priori.

In the translation, the auxiliary function $rep : S\mathbf{1} \to A \to SA$ produces a stream with every $*$ in s replaced by a. The function $usum : S\mathbf{B} \to S\mathbf{1}$ counts, in unary, the F's in a segment-boundary stream; formally, we can define it by the equations:

$$
\begin{array}{rcl}
usum\ \langle\rangle & = & \langle\rangle \\
usum\ \langle \mathsf{F} \mid s \rangle & = & \langle * \mid usum\ s \rangle \\
usum\ \langle \mathsf{T} \mid s \rangle & = & usum\ s
\end{array}
$$

(We write stream heads and tails between $\langle \cdot \mid \cdot \rangle$.) For any concrete τ, the

distribution function $dist_\tau : [\![\tau]\!] \to S\mathbf{B} \to [\![\tau]\!]$ is given by:

$$\begin{aligned} dist_\pi\ s_0\ s &= pdist\ s_0\ s \\ dist_{(\tau_1,...,\tau_k)}\ (t_1,...,t_k)\ s &= (dist_{\tau_1}\ t_1\ s, ..., dist_{\tau_k}\ t_k\ s) \\ dist_{[\tau]}\ (t_0, s_\mathrm{s}, s_\mathrm{l})\ s &= (t_0, pdist\ s_\mathrm{s}\ s, pdist\ s_\mathrm{l}\ s)\,, \end{aligned}$$

where $pdist : SA \to S\mathbf{B} \to SA$ is a segmented distribute for atomic values:

$$\begin{aligned} pdist\ \langle\rangle\ \langle\rangle &= \langle\rangle \\ pdist\ \langle a\,|\,s\rangle\ \langle \mathsf{F}\,|\,s'\rangle &= \langle a\,|\,pdist\ \langle a\,|\,s\rangle\ s'\rangle \\ pdist\ \langle a\,|\,s\rangle\ \langle \mathsf{T}\,|\,s'\rangle &= pdist\ s\ s'\,. \end{aligned}$$

Note that each iteration consumes exactly one element of the flag stream, but zero or one element of the data stream. (For actual execution, as described in the next section, streams are processed chunkwise, and the element-wise specification would be implemented efficiently in parallel using segmented scans, like in NESL.)

The restricted comprehension is handled similarly. $b2u : S\mathbf{B} \to S\mathbf{B}$ maps truth values to segment flags:

$$\begin{aligned} b2u\ \langle\rangle &= \langle\rangle \\ b2u\ \langle \mathsf{F}\,|\,f\rangle &= \langle \mathsf{T}\,|\,b2u\ f\rangle \\ b2u\ \langle \mathsf{T}\,|\,f\rangle &= \langle \mathsf{F}\,|\,\langle \mathsf{T}\,|\,b2u\ f\rangle\rangle \end{aligned}$$

The function $pack_\sigma : [\![\sigma]\!] \to S\mathbf{B} \to [\![\sigma]\!]$ is defined analogously to $dist_\tau$, in terms of a primitive $ppack : SA \to S\mathbf{B} \to SA$ given by:

$$\begin{aligned} ppack\ \langle\rangle\ \langle\rangle &= \langle\rangle \\ ppack\ \langle a\,|\,as\rangle\ \langle \mathsf{F}\,|\,bs\rangle &= ppack\ as\ bs \\ ppack\ \langle a\,|\,as\rangle\ \langle \mathsf{T}\,|\,bs\rangle &= \langle a\,|\,ppack\ as\ bs\rangle\,, \end{aligned}$$

but *pack* also has an additional clause for packing sequence types:

$$pack_{\{\sigma\}}(t, s)\ b = (pack_\sigma\ t\ (pdist\ b\ s), upack\ s\ b)\,.$$

That is, we first distribute the pack flags b according to stream's segment flags, and use them to pack the underlying stream elements. $upack : S\mathbf{B} \to S\mathbf{B} \to S\mathbf{B}$ is like *ppack* but packs unary numbers (subsequences of the form $\langle \mathsf{F}, ..., \mathsf{F}, \mathsf{T}\rangle$, rather than atomic values.

The other primitive functions in $[\![F]\!]$ are defined similarly, many in a type-directed fashion. For instance, $mkstr^k_{\mathsf{Int}}$ is ultimately defined in terms of a k-way primitive merge:

$$\begin{aligned} pmerge\ \langle\rangle \cdots \langle\rangle &= \langle\rangle \\ pmerge\ \langle a_1\,|\,s_1\rangle \cdots \langle a_k\,|\,s_k\rangle &= \langle a_1\,|\,\cdots \langle a_k\,|\,pmerge\ s_1 \cdots s_k\rangle\rangle\,. \end{aligned}$$

2.3.3 Execution Model

The low-level streaming language is effectively a *dag* of stream definitions, represented as a linear list of "instructions" such as

$$\texttt{s1} := \texttt{lit}\langle 5\rangle;\ \texttt{s2} := \texttt{iota}(\texttt{s1});\ \texttt{s3} := \texttt{reduce\_plus}(\texttt{s2}),$$

similar in principle to the control-free fragment of VCODE [BCH+94] (though we use named variables rather than a stack model). However, while it would be correct (wrt. the value computed and work/depth complexity) to execute such a sequence from top to bottom, it would entirely defeat the point of streamability, and the space usage would always be on the order of the "parallel space" from the cost model, even on a completely sequential machine.

Instead, we compute the stream definitions incrementally and chunk-wise, in a dataflow fashion. We repeatedly "fire" the definitions to transform some elements in the input stream(s) into elements of the output stream. Each stream definition has an associated *buffer*, which represents a moving window on the underlying stream of values. For streams representing vector-free values, the buffer is always of a fixed size, related to the number of processors; but for streams of vectors, the buffer may expand dynamically to contain at least each subvector at once. (The buffer never shrinks below the chunk size, so that, for example, the buffer for a stream of length-2 vectors would normally contain many such vectors at once.) Note that vectors only represent data storage, not active computations; it is only when they are explicitly turned into sequences (by *seq*) that they are either divided or coalesced into chunks.

Each stream window can only move forwards; once it passes past a part of the stream, those stream elements become inaccessible. To ensure that all consumers of a stream have accessed the stream elements they need before the window advances, the implementation maintains *read-cursors* for each stream, keeping track of the progress of each reader, to make sure that all of the consumer firings have happened before the next producer firing is enabled.

In addition to the buffer, each stream may have a fixed-size *accumulator*, which keeps tracks of the computation state across chunks. For example, when computing the sum or +-scan of a stream, the accumulator represents the sum of the elements so far, and is used to "seed" the computation of the next chunk, rather than restarting from zero each time. (This is how sums or scans of vectors larger than the maximal block size must be implemented in CUDA anyway; the difference is that we allow the processing of consecutive sum/scan chunks to be interleaved with chunks from unrelated computations.)

To keep the scheduling overhead small compared to the work performed in each chunk, their size must generally be chosen somewhat larger than the number of available processors. For example, on a fairly large GPU, a suitable chunk size seems to be 64k–256k elements; see next section for details. Currently, for simplicity, the chunk size is fixed for all streams and throughout the computation, but in principle, it could vary dynamically, depending on memory pressure, or even adaptively based on on-going performance measurements.

Streamability To actually be executable in a streaming fashion, source programs must respect the inherent *temporal* dependencies between subcomputations. Most notably, no auxiliary variable in a general comprehension may depend on a computation that requires a prior traversal of the sequence currently being traversed. For example,

$$\begin{array}{l}\textbf{let } s = \{log(real(x+1)) : x \textbf{ in } iota(n)\} \textbf{ in}\\ \quad \textbf{let } m = reduce_+(s) \textbf{ in}\\ \quad\quad reduce_+(\{x \times x + m : x \textbf{ in } s \textbf{ using } m\})\end{array}$$

cannot be executed in constant space (i.e., independent of n), without duplicating the computation of s, because m is only known after all of s has been traversed. On the other hand, the following, mathematically equivalent, expression is fine:

$$\begin{array}{l}\textbf{let } s = \{log(real(x+1)) : x \textbf{ in } iota(n)\} \textbf{ in}\\ \quad \textbf{let } m = reduce_+(s) \textbf{ in}\\ \quad\quad reduce_+(\{x \times x : x \textbf{ in } s\}) + slength(s) \times m\end{array}$$

because all three traversals of s can be performed in the same pass. An alternative approach would be to materialize s, and traverse the stored copy twice:

$$\begin{array}{l}\textbf{let } sv = tab(\{log(real(x+1)) : x \textbf{ in } iota(n)\}) \textbf{ in}\\ \quad \textbf{let } m = sum(seq(sv)) \textbf{ in}\\ \quad\quad reduce_+(\{x \times x + m : x \textbf{ in } seq(sv) \textbf{ using } m\})\end{array}$$

A related situation arises with $+\!\!+$ (or *mkstr*): while transducing s to $s +\!\!+ scan_+(s)$ is obviously infeasible in constant space, $s +\!\!+ \{sum(s)\}$ or $scan_+(s) +\!\!+ \{sum(s)\}$ are fine – but $\{sum(s)\} +\!\!+ s$ or $\{sum(s)\} +\!\!+ scan_+(s)$ are not.

In our current implementation, such illegal dependencies are only detected at runtime, but they should be conservatively preventable already at the source level by a suitable analysis.

In most functional (or imperative) languages, a programmer who wants to compute, say, both the sum of a number sequence and whether it contains any zero elements, without unnecessarily materializing it, must explicitly merge both reductions into a single `foldl` (or loop). Even though a lazy language, like Haskell, could in principle compute both consumers of s in

```
... sum s ... not (all (/= 0) s) ...
```

in lockstep, garbage-collecting s incrementally, most likely it would memoize all of s during the computation of `sum`, and only deallocate it again after the `all` had been computed. In any case, the programmer would not be able to count on the optimization.

It remains to be seen if working in a nominally eager language, but with additional temporal constraints between variables (and getting an error instead of a silent space explosion when those constraints are violated), is desirable in practice. We suspect that for performance-sensitive applications, it may be; otherwise, the obvious easy fix is for the compiler to insert (possibly with a warning) seq/tab-pairs and/or duplicate computations, in those places where it cannot guarantee streamability.

2.4 Empirical Validation

The practical applicability of our model is investigated through a number of experiments over three semi-realistic parallel problems. The GPU used for the benchmarks is an NVIDIA GeForce GTX 690 (2 GB memory, 1536 cores, 915 MHz), and the CPU is a dual AMD Opteron 6274 (2×16 cores, 2200 MHz). Due to significant numerical sensitivity, all tests are performed using double-precision floating points for real numbers when possible. The problems we consider are:

- The sum of logarithms from the Introduction. From now on referred to as *log-sum*.
- A total sum of several sum of logarithms, also presented in the Introduction. From now on referred to as *sum of log-sums*.
- An N-body simulation, where the force interaction for all pairs of bodies is computed, without using any special data structures.

For all problems, we compare the running time on a number of implementations:

- A single-threaded C implementation running on the CPU serving as a sanity check for the rest of the implementations.

- A hand-optimized CUDA implementation.
- An implementation in Accelerate [CKL+11] version 0.13.0.1, a GPU-enabled language embedded in Haskell.
- An implementation in Single Assignment C (SaC) [Sch03] version 1.00_17229 using a multicore backend. SaC also supports a GPU target, but for the experiments that we consider, the SaC compiler does not emit GPU code. Namely, with-loops with reductions are not executed on the GPU in the version of SaC we have tested.
- An implementation in NESL-GPU [BR12], both with and without kernel fusion. NESL-GPU is NESL with a VCODE interpreter implemented in CUDA as back-end. Real numbers are only implemented with single-precision in the NESL-GPU backend, so the NESL benchmarks suffer from numerical imprecision and an unfair advantage. Nonetheless, NESL-GPU uses the double-precision version of the logarithm instruction, so in comparison to the sum-log problem, the advantage is negligible as the calculation of the logarithm dominates the performance.
- A streaming implementation written in CUDA that reflects the streaming model of execution presented in this paper.

The comparison to Accelerate, SaC and NESL-GPU is done to measure the performance of the streaming model against other high-level data parallel languages without streaming execution. NESL-GPU and SaC support irregular nested data parallelism, while Accelerate only supports flat parallelism, and consequently NESL-GPU and SaC are are similar to the source language for the streaming model presented in this paper and therefore the most interesting languages to compare with. Both Accelerate and NESL-GPU support a GPU backend and perform kernel fusion, but NESL-GPU requires the programmer to manually run a separate fusion phase and compile and link the fused kernels. Using kernel fusion in NESL-GPU gives a marginal speedup for all our experiments, and therefore, only the timings using kernel fusion for NESL-GPU are presented here. The streaming implementation is based on the streaming model presented in this paper, implemented manually. However, there is nothing to suggest that similar code could not have been generated automatically by a compiler. The streaming implementations uses the CUDA Parallel Primitive library (CUDPP) for performing reduction and scan primitives as well as stream compaction.

We measure the running time of each experiment by using the wall-clock time averaged over an appropriate number of executions. Note that the time it takes to load the CUDA driver and initialize the GPU is not included in

the benchmark, since it varies greatly from platform to platform. Memory allocation and de-allocation on the GPU and data transfer between device and host is, however, included in the timings for the CUDA and streaming implementations.

2.4.1 Log-sum

The log-sum problem can be categorized as flat data-parallelism, and it can easily be expressed in all languages included in the experiments. In the streaming source language it can be implemented using only sequence types, so we can expect to compute the problem in constant space. The total work is proportional to N – the problem size.

Without going into details, the problem can be compiled from its source form to the following data-flow network using a straight-forward mapping of the primitives:

$$\begin{aligned} s_0 &:= \texttt{range}(1, N); \\ s_1 &:= \texttt{log}(s_0); \\ s_2 &:= \texttt{sum}(s_1); \end{aligned}$$

Each stream definition is implemented by a separate kernel in CUDA, and scheduling is simply implemented as a for-loop, scheduling each of the three definitions in sequence in each iteration.

Figure 2.5 shows the running times of the log-sum problem for a problem size N varying from 2^{12} to 2^{32}. We can see that all the GPU implementations outperform C and SaC for large enough problem sizes as expected. Furthermore, the running time of all the GPU implementations converge as the input size increases. Note that NESL-GPU runs out of memory when $\log_2(N) > 25$. Accelerate and SaC fail when $\log_2(N) > 30$ due to the number of bits used to represent the size of a single dimension is limited to 32. In both cases, the problem could probably be mapped to a 2-dimensional matrix without significant performance loss, but such a mapping stands in contrast to the high-level of abstraction that the languages have been selected for comparison because of.

From the second plot we can see that the choice of chunk size greatly affects the running time: the running time grows rapidly as the chunk size decreases for small chunks sizes ($B < 2^{18}$), but for sufficiently large chunk sizes ($B \geq 2^{18}$), the running time stays more or less the same. Furthermore, a larger block size incurs a larger overhead, which leads to significant performance degradation for small problem sizes. This is an indication that on our particular hardware, the chunk size $B = 2^{18}$ is a good choice, keeping in mind that a larger chunk size requires more memory.

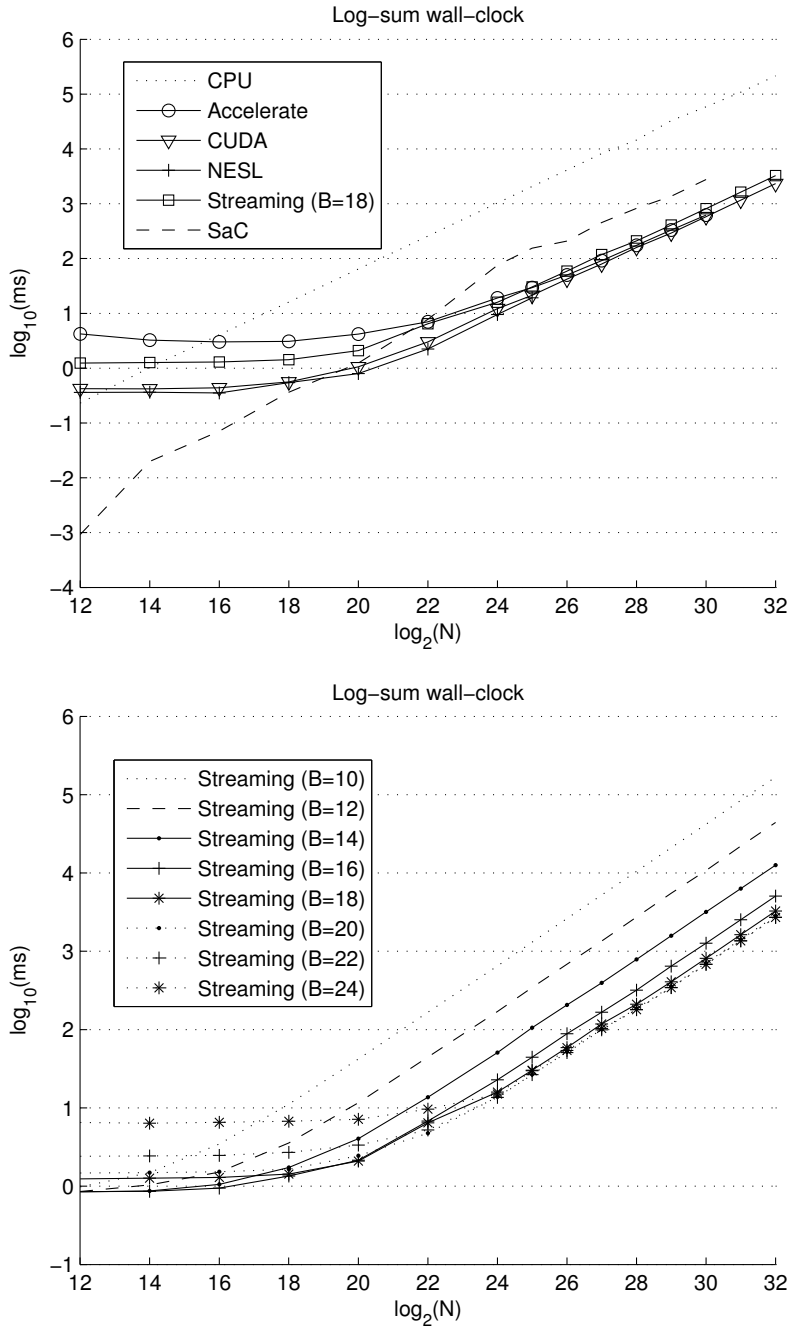


Figure 2.5: Benchmark results of the log-sum problem. The x-axis is the problem size in base-2 logarithm, and the y-axis is the running time in milliseconds in base-10 logarithm. The upper plot shows the running time of different implementations measured in wall-clock time. The lower plot shows the running time of the streaming implementation for different choices of block sizes.

2.4.2 Sum of Log-sums

The sum of log-sums problem can be categorized as irregular nested data-parallelism because the sub-sums varies in size. The total work is proportional to N^3. Just like log-sum, sum of log-sums can be implemented using only sequence types in the streaming language. It is not at all obvious how to implement this problem efficiently in Accelerate or CUDA as these languages do not facilitate automatic parallelization of nested data parallelism, and since the parallelism is irregular, there is no straight-forward way to sequentialize the programs by hand. We leave out an Accelerate implementation for this problem and implement two CUDA versions. The two versions are manually sequentialized on two different levels to make the problem flat:

- Inner loop: Using N threads, each sub-sum is computed sequentially in a single thread. The results are then summed in parallel.
- Outer loop: In a top-level sequential loop, compute log-sum for $i = 1^2, \cdots, N^2$ with i threads using the CUDA implementation from the log-sum experiment.

Both sequentialization strategies are easy to implement, but yield uneven work distribution.

The compilation of sum-log-sum in the streaming model is similar to the compilation of log-sum, but with parallel versions of range computation and summation, leading to segmented streams. Without going into detail, the compilation will produce the following data-flow network:

$$
\begin{aligned}
s_0 &:= \texttt{range}(1, N) \\
s_1 &:= \texttt{mult}(s_0, s_0) \\
s_2 &:= \texttt{segment-head-flags}(s_1) \\
s_3 &:= \texttt{ranges}(s_2); \\
s_4 &:= \texttt{log}(s_2); \\
s_5 &:= \texttt{segmented\_sum}(s_2, s_3); \\
s_6 &:= \texttt{sum}(s_4);
\end{aligned}
$$

Here follows and explanation of the newly introduced instructions:

- `segment-head-flags`: Converts segment lengths to head flags. E.g.

$$\langle 2, 3 \rangle \mapsto \langle \mathsf{T}, \mathsf{F}, \mathsf{T}, \mathsf{F}, \mathsf{F} \rangle.$$

- `ranges`: Produces a range $1..n$ for each segment. E.g.

$$\langle \mathsf{T}, \mathsf{F}, \mathsf{T}, \mathsf{F}, \mathsf{F} \rangle \mapsto \langle 1, 2, 1, 2, 3 \rangle.$$

It is implemented as a segmented scan of 1's followed by adding 1 to each element.

- `segmented_sum`: Takes a stream of segment head flags and a stream of values and outputs a sub-sum for each segment. E.g.

$$\begin{matrix}\langle \mathsf{T}, & \mathsf{F}, & \mathsf{T}, & \mathsf{F}, & \mathsf{F}\rangle \\ \langle 2, & 3, & 1, & 0, & 7\rangle\end{matrix} \mapsto \langle 5, 8\rangle.$$

Scheduling is an outer loop over all the instructions with an inner loop over instructions s_2, s_3, s_4 and s_5.

Figure 2.6 shows the running times of the sum of log-sums problem for an problem size N varying from 2^4 to 2^{12}. Just like for the log-sum problem, the GPU implementations will only outperform C and SaC for large enough problem sizes. NESL-GPU has good performance, but runs out of memory at $N = 2^9$. If the implementation was able to continue beyond this point, the performance seems to coincide with the streaming implementation suggesting that the two have equivalent performance, except the streaming implementation has some initial overhead that is significant for small problem sizes. The two CUDA versions are outperformed by the streaming implementation for medium problem sizes ($7 \leq \log(N) < 10$), which is likely due to uneven work distribution. The inner loop implementation is apparently asymptotically superior to the other implementations, but this is likely due to the total running time being bounded by the most work-heavy thread, which computes exactly N^2 logarithms, suggesting that any work done up until this thread is started, is negligible. The curve will likely converge to a cubic slope for even larger problem sizes. The outer loop implementation seems to reach the point of cubic slope at around $\log(N) = 11$, where it already outperforms the streaming model. With this problem size, the work is dominated by a few very large computations of log-sum which can utilize the entire GPU, so this result is not surprising.

A chunk size of $B = 2^{18}$, appears to be a good choice again. The gap between the CPU implementation and the remaining implementations is significantly smaller for sum of log-sums than for log-sum, leading to the conclusion that none of the implementations handle irregular nested parallelism particularly well.

2.4.3 N-Body

The N-body problem can be categorized as regular nested data-parallelism (i.e. all sub-computations have the same size). For simplicity we assume that all bodies have unit mass, and we simulate each body with the unit

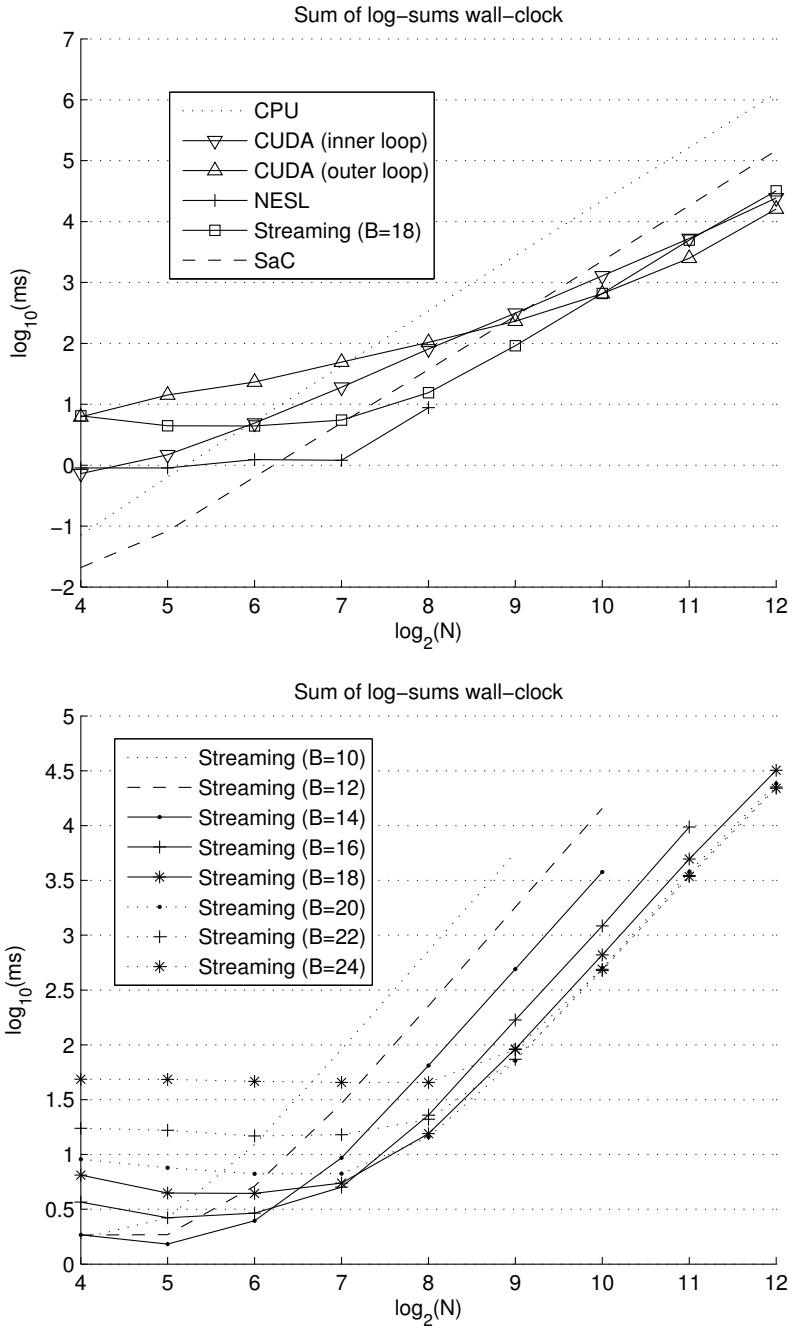


Figure 2.6: Benchmark results of the sum of log-sums problem, with the same conventions as in Figure 2.5.

time-step. To avoid the problem of singularities, we use the formula

$$f(\vec{x},\vec{y}) = \frac{\vec{x}-\vec{y}}{((\vec{x}-\vec{y})\cdot(\vec{x}-\vec{y})+\epsilon)^{3/2}},$$

to compute the directional force between body x and y, where the ϵ term ensures that no two bodies will ever have zero distance sending them off to infinity. We define the problem as, given a system of N bodies, for each body, given an initial position and velocity, compute the acceleration subject to the force interaction from all other bodies in the system, and compute a new position and velocity, in three dimensions using the formula

$$\begin{aligned}\vec{x}_\ell &= \vec{x}_{\ell-1} + dt\cdot\vec{v}_{\ell-1} + 1/2\cdot dt^2\cdot\vec{a}_{\ell-1}\\ \vec{v}_\ell &= \vec{v}_{\ell-1} + dt\cdot\vec{a}_{\ell-1}\end{aligned}$$

The total work in one iteration is proportional to N^2. We measure the average execution time of an iteration over a long simulation.

Although Accelerate contains no support for nested data-parallelism, the regularity of the problem enables easy manual flattening by replication of the bodies to form a matrix of all body-pairs. A scalar function is mapped over each element of the matrix computing the force between a pair of bodies, and each row is subsequently reduced to find the sum of all forces acting on each body.

The implementation in NESL-GPU is much more intuitive due to the support for nested data-parallelism[2]:

$$\begin{aligned}\texttt{sum\_3d}(X) &= \textbf{let}\ (X,Y,Z) = \texttt{unzip3}(X)\\ &\quad\ \textbf{in}\ (\texttt{sum}(X),\texttt{sum}(Y),\texttt{sum}(Z))\\ \texttt{g}(x,X) &= \texttt{sum\_3d}(\{f(x,y) : y\ \textbf{in}\ X\})\\ \texttt{nbody}(X) &= \{\texttt{g}(x,X) : x\ \textbf{in}\ X\}\end{aligned}$$

The matrix of all body-pairs is implicitly computed in this expression since X must be distributed over itself in order to use X in the inner-most apply-to-each.

We cannot implement the problem in the streaming language without using concrete types. More precisely, if we use the NESL expression as a starting point, the variables that are used in the body of both apply-to-each constructs, must be explicitly stated. The outer apply-to-each uses X, and consequently from the type rule of apply-to-each, X cannot have sequence type and must be fully materialized in memory. The solution is to make

[2]The code for updating positions and velocities is now shown here.

sure X is tabulated, which is also what one would assume, and use $\texttt{seq}(X)$ to traverse it multiple times as a sequence.

$$\begin{aligned} \texttt{g}(x, X) &= \texttt{sum\_3d}(\{f(x,y) : y \textbf{ in } \texttt{seq}(X) \textbf{ using } x\}) \\ \texttt{nbody}(X) &= \{\texttt{g}(x, X) : x \textbf{ in } \texttt{seq}(X) \textbf{ using } X\} \end{aligned}$$

A compiler can utilize the regularity of the problem to generate more efficient code. More specifically, we can compute the index ranges using modulo arithmetic, and the segmented sum using a single unsegmented scan, a gather and a subtraction. If we assume that the streaming compiler can infer and exploit this regularity, we can generate the following code for the streaming model:

$$\begin{aligned} X &:= < \texttt{input} > \\ s_0 &:= \texttt{2d\_range\_x}(N, N); \\ s_1 &:= \texttt{2d\_range\_y}(N, N); \\ s_2 &:= \texttt{gather}(X, s_0); \\ s_3 &:= \texttt{gather}(X, s_1); \\ s_4 &:= \texttt{force}(s_2, s_3); \\ s_5 &:= \texttt{2d\_segmented\_sum}(N, N, s_4); \end{aligned}$$

Here $\texttt{2d\_range\_x}(N, M)$ produces the stream

$$\langle \overbrace{\overbrace{0, \ldots, N-1}^{}, \cdots, \overbrace{0, \ldots, N-1}^{}}^{M} \rangle,$$

and $\texttt{2d\_range\_y}(N, M)$ produces the stream

$$\langle \overbrace{0, \ldots, 0}^{N}, \cdots, \overbrace{M-1, \ldots, M-1}^{N} \rangle.$$

$\texttt{2d\_segmented\_sum}(N, M, s)$ produces a regular segmented sum of s segmented in N segments, each of length M. `force` is the force calculation between two bodies, fused into a single instruction. The force calculation consists solely of scalar operations, so fusion is straight-forward, and it is fair in comparison since both Accelerate and NESL-GPU uses fusion.

The hand-optimized CUDA implementation is based on the algorithm presented by Nyland, Harris, and Prins in *GPU Gems 3* [NHP07] and uses explicit cache management and tiling.

The implementations in NESL-GPU, Accelerate and Sac do not contain any explicit sequentialization except for the simulation iterations. This is important because such a sequentialization would be a platform-specific

optimization, and we are comparing with these languages because they are platform-agnostic. We were not able to produce an implementation in SaC that performs better than the CPU implementation for N-Body.

Figure 2.7 shows the running time of N-Body. Here the NESL-GPU implementation runs out of memory for all but the smallest problem sizes (2^{10} bodies) and performs horribly, likely due to explicit replication. Accelerate on the other hand is able to handle all tested input sizes indicating that it handles replication symbolically.

Once the input size is large enough, the streaming version is a constant factor faster than the CPU version, Accelerate is a constant factor faster than the streaming version, and the CUDA version is a constant faster than Accelerate. Compared to the previous problems, the CUDA implementations is now significantly faster than the other GPU implementations, and the streaming implementation is painfully close to the CPU implementation in performance. From the lower plot we can see that the optimal chunk size is the same as for the previous problems.

2.4.4 Discussion

Considering the experimental results of the streaming implementation in isolation, it is evident that the running time of a given problem converges as the chunk size increases, and furthermore, it converges long before the chunk size reaches the problem size for large enough problem sizes. As stated previously, when the chunk size is big enough, the streaming execution model is largely equivalent to that of NESL. In conclusion, choosing a reasonable chunk size, the streaming model will not be slower than a traditional execution model. The three experiments all suggested the same optimal chunk size of 2^{18}, which is important since it is an indication that the optimal chunk size is independent from the algorithm and problem size, meaning that for a given concrete machine, a specific chunk size can be selected once and for all programs. Given a chunk size of 2^{18} and depending on the type, a single buffer requires roughly 8–16 MB worth of memory on the device, enabling several hundreds of buffers to be allocated at any given time - more than enough for most algorithms. It should be possible to estimate the number of required buffers before execution begins, at least for our somewhat restricted source language, and if more buffers are needed than the GPU capacity enables, the block size can be lowered. In extreme cases, buffers can be swapped in and out of device memory dynamically.

Comparing the results of the streaming implementation with the other GPU implementations, the experiments shows that a streaming execution of nested data-parallel programs on GPGPUs is on-par with existing GPU-enabled high-level languages both for flat, regular nested and irregular

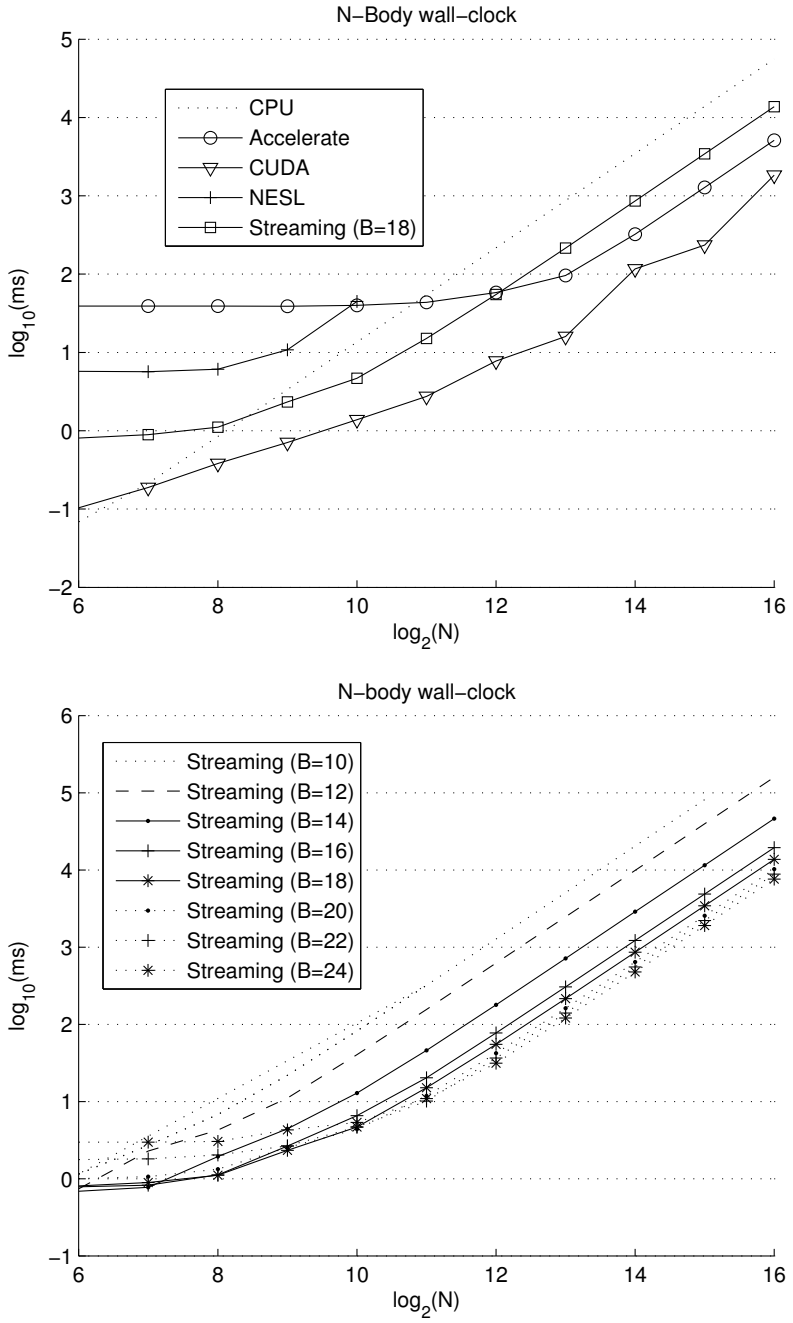


Figure 2.7: Benchmark results of the sum of N-Body problem, with the same conventions as in Figure 2.5.

nested data-parallelism. We also conclude that much larger problem sizes are supported when using streaming than what is available in NESL. This is a critical feature for data-parallel languages, because the benefit of parallel execution increases as the problem size increases. For small problem sizes, the difference between 1 second and 10 milliseconds is quickly shadowed by the overhead of loading and unloading CUDA's drivers, but the difference between 1 hour and 100 hours for huge problem sizes is significant. By allowing a data-parallel program to work on such problem sizes is therefore highly valuable, and that is exactly what the streaming execution model provides that both Accelerate and NESL-GPU does not.

On the other hand, the running time for streaming execution is still considerably higher than we had hoped and what a hand-optimized CUDA implementation offers. This can partly be attributed to lack of tiling and explicit cache management. Another concern is that dividing a parallel instruction into several kernel invocations, as is required by data-flow execution, precludes the use of registers to store intermediate results; In CUDA, it is not possible to carry values stored in registers or shared memory from one kernel invocations to the next, even if it is the same kernel that is invoked. Instead, the global memory must be used, which is much slower. This indicates that kernel fusion is still beneficial in the streaming model.

The experiments do not provide a clear validation of the streaming model, but they do not reject it either. The results suggest that implementing a GPU backend for a NESL-like language that scales to extremely large problem sizes is possible using the streaming model presented in this paper, without incurring too severe performance degradation for small and medium problem sizes.

2.5 Preliminary Conclusions and Future Work

We have outlined a high-level cost model and associated implementation strategy for irregular nested data parallelism, which retains the performance characteristics of parallelism-flattening approaches, while drastically lowering the space requirements in several common cases. In particular, many highly parallelizable problems that also admit constant-space sequential algorithms, when expressed in the language, have space usage proportional to the number of processors – not to the problem size.

The language and implementation are still under development, and many details are incomplete or preliminary. Particular on-going and future work, not already mentioned at length, includes:

- Extending the language and cost model with recursion, to allow expression of more complex algorithms. The main challenge here is to

determine to what extent common parallel-algorithm skeletons admit streaming formulations. For example, to explicitly code a logarithmic-depth *reduce*, a divide-and-conquer approach (split vector in halves, reduce each half in parallel, and add the partial results) will obviously not work for sequences, when not even the sequence length is known a priori. On the other hand, a unite-and-conquer reduction (add pairs of adjacent elements in parallel, then recursively reduce the resulting half-length sequence) can be implemented in a streaming fashion, and probably exhibits better space locality as well.

- Extending the model to account for bulk random-access vector *writes* (permutes, or more generally, combining-scatter operations). A significant class of algorithms that nominally involve random-access vector updates, such as histogramming or bucket sorting, can still be expressed in a parallel, streaming fashion by generalizing (segmented) scans to *multiprefix* operations [She93]. Making multiprefix primitives conveniently utilizable by the programmer should minimize, or maybe even eliminate, the need for explicitly distinguishing between copying and in-place implementations of vector updates in the cost model.
- Formally establishing the time and space efficiency of the implementation model, in the sense that the work and depth complexity, and parallel and sequential space usage, predicted by the high-level model are in fact realized, up to a constant factor, by the low-level language with chunk-based streaming.
- A full language implementation with a representative collection of back-ends (including at least sequential, multicore/SIMD, and GPGPU) to gather more practical experience with the model, and in particular determine whether the hand-coded implementations of particular streaming algorithms can also be realistically generated by a fixed compiler.

Finally, though we believe that the main value of the streaming model is its explicit visibility to the programmer, some of the ideas and concepts presented in this paper might be adaptable for transparent incorporation in other data-parallel language implementations (APL, SaC, Data Parallel Haskell, etc.), to achieve drastic reduction in memory consumption in many common cases, without requiring explicit programmer awareness of the streaming infrastructure.

Chapter 3

Functional Array Streams

This chapter is a reprint of [MCECK15] without any non-trivial changes.

Abstract

Regular array languages for high performance computing based on aggregate operations provide a convenient parallel programming model, which enables the generation of efficient code for SIMD architectures, such as GPUs. However, the data sets that can be processed with current implementations are severely constrained by the limited amount of main memory available in these architectures.

In this paper, we propose an extension of the embedded array language Accelerate with a notion of sequences, resulting in a two level hierarchy which allows the programmer to specify a partitioning strategy which facilitates automatic resource allocation. Depending on the available memory, the runtime system processes the overall data set in streams of chunks appropriate to the hardware parameters.

In this paper, we present the language design for the sequence operations, as well as the compilation and runtime support, and demonstrate with a set of benchmarks the feasibility of this approach.

3.1 Introduction

Functional array languages facilitate high-performance computing on several levels. The programmer can express data-parallel algorithms declaratively, and the compiler can exploit valuable domain specific information to generate code for specialised parallel hardware. A standard array language separates itself from traditional languages by offering data-parallel collection oriented constructs as primitives such as map, fold, scan and permuta-

tion. Without these primitives, the same logic would have to be encoded as sequential for-loops or recursive definitions, obfuscating data-dependency and access patterns. Due to the artificial data-dependencies introduced by the loop counter or the recursion stack, these encodings prevent natural parallelisation, and the compiler must resort to program analysis to detect and exploit implicit parallelism. Such automatic parallelisation strategies are fragile, and small changes in the code may cause them to fail, significantly degrading the performance for reasons not obvious to the programmer.

Array languages present a complementary problem. The explicit data-parallelism exposed in an array program may vastly exceed the actual parallel capabilities of the target hardware. Data-parallel programs often require working memory in order of the degree of parallelism. Therefore, it is not always desirable or even possible to execute an array program in its full parallel form. To conserve space, we would like the compiler to make the program "less parallel" prior to execution. However, the absence of explicit sequential data-dependencies prevents natural *sequentialisation*.

If we would execute each parallel combinator in isolation, we could simply sequentialise the combinator by partitioning the index-space and scheduling the different parts in a tight loop. Evidently, this is how CUDA schedules a kernel in blocks on a large grid.

In practice, however, it is essential to fuse sequences of parallel combinators together to form complex computations, thereby reducing the number of array traversals and intermediate structures. As soon as such a sequence includes more than simple maps, the combinators may not traverse the index-space in a uniform way. Consequently, loop fusion can be very complex. Compiler-controlled sequentialisation affects the fusion-transformation, and complicates it further. Finding the optimal sequentialisation strategy in this context is not decidable, so we would have to resort to using heuristics, leaving the programmer at the mercy of the compiler again.

Therefore, we propose to give control over this step to the programmer, who has more knowledge about the nature of the application and size of the processed data set. We achieve this by including a set of *sequence* combinators for array languages, so sequential data-dependency over data-parallel computations can be specified and the amount of parallelism exposed be controlled.

This paper presents these new sequence combinators, using the language Accelerate as starting point and discusses the extensions to the runtime system with the required streaming and scheduling mechanism. In summary, the contributions of this paper are as follows:

- We present a new set of sequence combinators, which, together with

the usual combinators like maps, folds and scans, can be used to express a two-level hierarchy sequentially combining a sequence of parallel operations over chunks of data.

- We present a runtime system extension which implements the necessary scheduling and streaming mechanisms.
- We present an evaluation of the approach presented in the paper.

While we are currently only targeting single-GPU architectures, the programming model we propose in this paper also allows the programmer to expose pipeline-parallelism in a program, which we could exploit in an implementation for multi-GPU architectures. Although outside the scope of this paper, other data-parallel architectures, such as multi-processors and distributed systems, would also benefit from the model presented here.

3.2 Accelerate

Accelerate is a domain specific functional language for high-performance, multi-dimensional array computations, implemented as deep embedding in Haskell. In addition to the collection oriented operations similar to those on lists, like maps, scans, reductions, it also offers array-oriented operations, such as stencil convolutions, as well as forward- and backward permutations, conditionals and loops. Indeed, apart from the type annotation (to which we will get back shortly), many Accelerate programs – like this dot-product example – look almost like the corresponding list operation in Haskell:

```
dotp :: Acc (Vector Float)
     → Acc (Vector Float)
     → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

In contrast to Haskell, though, Accelerate is strict and fully normalising.

The language has two non-obvious restrictions: (1) Arrays must be *regular*. By regular, we mean arrays cannot contain other arrays as elements. Instead, arrays are multidimensional. Scalars, vectors, matrices, tensors, and so on, are all regular arrays, but a vector of arbitrary-length vectors is not. (2) Accelerate does not permit nested data-parallelism. For example, even though one could imagine using a two dimensional array as a vector of vectors, mapping a map over each sub-vector is not allowed. These restriction enables a smooth compilation to SIMD hardware. Accelerate comes with a number of backends, among them a GPU implemen-

tation generating CUDA [NVI12] code, that demonstrate excellent performance [CKL+11, MCKL13].

The restriction to regular computations and arrays is enforced statically via the type system, which serves to separate Accelerate-expression into two distinct categories.

- `Elt a => Exp a`: Expressions which evaluate to values of type *a*, where *a* has to be a member of type class `Elt`, which includes basic types such as integers, floats, bools, as well a *n*-tuples of these. Accelerate generates valid CUDA C from `Exp` expressions.

- `(Shape sh, Elt a) => Acc (Array sh a)`: Expressions which evaluate to *n*-dimensional arrays of type with element type `a` and shape `sh`. Accelerate generates CUDA GPU kernels from `Acc` expressions.

Shapes are sequences of integer dimensions separated by `:.` (e.g. `Z :. 2 :. 3`), and `Scalar` is a type synonym for a zero-dimensional array, `Vector` for a one-dimensional array. The type annotation of the `dotp` example therefore states that the function accepts two floating point vectors as arguments, and returns a scalar floating point value as result. More precisely, since Accelerate is a deep embedding, `dotp` takes in accelerate expressions specifying computations which produce results of these types, and returns a new computation.

Functions like `map`, `fold` and so on are *rank polymorphic*. For example, the type of `zipWith` is

```
zipWith :: (Shape sh, Elt a, Elt b, Elt c)
    ⇒ (Exp a → Exp b → Exp c)
    → Acc (Array sh a)
    → Acc (Array sh b)
    → Acc (Array sh c)
```

Several things are happening here: the type of the function passed to `zip` is, by its type, restricted to sequential computations over values of basic type. The type class constraint `Shape sh` essentially restricts `sh` to *n*-tuples of integers, where *n* determines the rank of the array. Both array arguments have to have the same rank, which is also the rank of the result. The actual sizes, however, are not tracked statically and may be different.

3.2.1 Fusion

It is well known that the collection oriented style of programming which Accelerate relies on has a serious potential drawback: if implemented naively,

by executing each aggregate operation separately, it can result in an excessive number of array traversals, intermediate structures, and poor locality. For example, it would clearly be inefficient if the code for the `dotp` example would first produce an intermediate array of the pairwise products, and then, in a second traversal, add all these sums to the final result. Therefore, Accelerate aggressively employs *fusion*, merging the operations to more complex computations, trying to minimise the number of traversals.

Fusion cannot, in general, guarantee that its results are optimal. Consider, for example, a fusible computation whose result is consumed by two different operations. If we would fuse into both consumers, we would avoid creating the intermediate structure, but duplicate the work involved to compute the array element, which can, in theory, slow down the performance considerably. In practice, most computations are fairly cheap compared to creating and accessing an array, so fusion would result in a significant speedup nevertheless. Without sophisticated cost-analysis, the compiler cannot decide which alternative results in the best performance, so we err on the side of caution and never fuse computations whose result is used more than once.

3.2.2 Handling Large Data Sets

We have shown previously [CKL+11, MCKL13] that the Accelerate approach of expressing parallel computations enables the generation of highly efficient code. The dot-product in Accelerate, for example, is only slightly slower than CUBLAS, a hand-written implementation of the Basic Linear Algebra Subprograms in CUDA by NVIDIA. However, on GPU architectures, we can only achieve peak performance if the data set we are processing in one parallel step is large enough to utilise all processing elements, yet small enough to still fit in the GPU memory, which is, at currently around 4GB, for the majority of hardware, much more restricted than CPU memory.

If programmers want to develop GPU programs which process larger set of data, they have to explicitly stage the computation into a sequence of parallel computations on smaller data chunks, and combine the subresults. While this is possible, it adds a significant layer of complexity to an already difficult task, and would lead to code whose relative performance is architecture dependent.

The other extreme option would be to try and let the compiler shoulder all the complexity of solving this problem. However, sophisticated optimisations like these have the downside that they usually cannot guarantee optimality, and behave in a way hard to predict by the programmer. Therefore, we choose an intermediate route: we allow the programmer to explicitly distinguish between parallel, random access structures, and streamed ones,

which allow only for a more limited set of operations. This gives the programmer the opportunity to design an algorithm tailored for this model, instead of hoping the compiler optimisations will work out.

In the following section, we describe the stream extension to Accelerate's program model, before we discuss its implementation and performance.

3.3 Programming Model

3.3.1 Examples

Let us go back to our `dotp` example. If we know that the input vectors most likely will not fit into memory, or we wish to ensure it minimises its space usage, we want to tell the compiler to split the input into chunks of appropriate size, calculate the product and sum for each chunk, and add the subresults as they are produced. Our stream extension makes this possible:

```
dotpSeq :: Acc (Vector Float)
        → Acc (Vector Float)
        → Acc (Scalar Float)
dotpSeq xs ys =
    collect
  $ foldSeqE (+) 0
  $ zipWithSeqE (*) (toSeqE xs) (toSeqE ys)
```

Here, `toSeqE` turns a normal `Vector` into a sequence of `Scalars`, `zipWithSeqE` performs element-wise multiplication of the two input sequences, `foldSeqE` calculates the sum, and `collect` takes the conclusion of the sequence computation and turns it into an `Acc` expression. The rest of this section will explain these primitives in more detail.

As Accelerate is rank-polymorphic, sequence operations can be parametrised by shape information. By convention, we denote specialised versions of these operations for sequences of scalars by the suffix `E`, as for example `toSeqE` above.

It is not just sequences of scalars that are supported, however. Our extension supports sequences of arbitrary rank. If for example we wanted to perform a matrix vector multiplication:

```
mvmSeq :: Acc (Matrix Float)
       → Acc (Vector Float)
       → Acc (Vector Float)
mvmSeq mat vec
  = let rows = toSeq (Z:.Split:.All) mat
    in collect
```

```
$ fromSeqE
$ mapSeq (dotp vec) rows
```

In this case, we first split the vector up into rows with `toSeq`, then apply `dotp vec` over every row, turn what is now a sequence of scalars into a `Vector` with `fromSeqE`, before finally `collect`ing the result.

In addition to not requiring the entire matrix be made manifest, this example also highlights how our extension enables an extra degree of nesting, in this case, defining matrix-vector multiplication in terms of the parallel dot-product, something not previously possible.

3.3.2 Streams

As we have seen in the previous examples, an Accelerate array is a collection where are all elements are simultaneously available, whereas a sequence value corresponds to a loop, where each iteration computes an element of the sequence. Sequences are ordered temporally, and are traversed from first to last element. Once an element has been computed, all previous elements are out of scope, and may not be accessed again. The arrays of a sequence are restricted to having the same rank, but not necessarily the same shape. If the shapes happen to be the same, we call the sequence *regular*. Using A to range over array values, and square brackets to denote sequences,

$$[A_1, A_2, ..., A_n]$$

denotes the sequence that computes A_1 first, computes A_2 second and so forth until the final array A_n is computed. Here, n is the length of the sequence (possibly zero).

Sequences model the missing high-level connection between the parallel notation of array languages and sequential notation of traditional for-loops. The basic sequence combinators are carefully selected such that the arrays of a sequence can be evaluated entirely sequentially, entirely parallel, or anything in between as long as the strategy respects the sequence order of arrays; even on SIMD hardware. The runtime system then selects a strategy that fits the parallel capabilities of the target hardware. The programmer may assume full parallel execution with respect to what the hardware can handle, while maintaining a limit on memory usage. A purely sequential CPU would evaluate one array at a time with a minimal amount of working memory. A GPU would evaluate perhaps the first 100 arrays in one go, and then evaluate the next 100 arrays, and so on. The working memory would be larger, but not as large as the cost of manifesting the entire sequence at once. Ideally, the runtime performance, in terms of execution time, should correspond to a fully parallel specification, and in terms of working memory,

should be in the order of a constant factor related to the parallel capabilities of the hardware - Unless any one array of the sequence exceeds this amount.

Of course, not all array algorithms can be expressed as sequences. As sequences can only be accessed linearly, any algorithm which relies on permuting or reducing an array in a non-linear way, cannot be expressed as a sequence. It is the responsibility of the programmer, not the compiler, to expose inherent sequentialism.

3.3.3 From Arrays to Sequences and Back

As we discussed previously, the type constructors `Exp` and `Acc` represent nodes in the AST from which Accelerate generates CUDA C code and CUDA GPU kernels, respectively. Sequence computations are represented by the type constructor `Seq`. Accelerate will generate CUDA kernels together with a schedule for executing the kernels over and over until completion.

While the type constructor `Seq` represents sequence AST nodes, we use the Haskell list syntax to represent the actual sequence type. That is, the type `[a]` represents sequences of `a`'s, and the type `Seq [a]` represents sequence computations that produce sequences of `a`'s when executed. The type `Seq a`, where `a` is *not* a sequence type, represents sequence computations that produce a single result of type `a`. We will see an example of such a type when we explain `foldSeqE`.

Sequences are introduced in Accelerate either by slicing an existing array, as we did in our examples, or by streaming an ordinary Haskell list into Accelerate, which we will discuss in detail in Section 3.3.4.

In our examples, we used the combinator `toSeqE` to convert one dimensional array into a sequence of values of the same element type:

```
toSeqE :: (Elt a)
    ⇒ Acc (Vector a)
    → Seq [Scalar a]
```

However, `toSeqE` is just a special case of the more general combinator `toSeq`, which operates on multi-dimensional arrays and is parametrised with a specific slicing strategy `div`:

```
toSeq :: (Division div, Elt a)
    ⇒ div
    → Acc (Array (FullShape  div) a)
    → Seq [Array (SliceShape div) a]
```

Values of types belonging to the `Division` type-class define how an array is divided into sub-arrays along one or more dimensions, where `Split`

at a given position tells the compiler to divide the elements along the corresponding dimension into a sequence, `All` to leave it intact.

Divisions are generated by the following grammar.

$$\texttt{Div} \ni \mathit{div} ::= \texttt{Z} \mid \mathit{div} \texttt{ :. All} \mid \mathit{div} \texttt{ :. Split}$$

`FullShape` and `SliceShape` are type functions that, for a given division, yield the shape of the full array and the shape of every slice. Let us have a look at an example to see how divisions can be used to slice a two dimensional array in different ways. Let A be the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 10 & 11 & 12 \end{pmatrix}$$

then we can either leave the matrix intact and create a sequence containing one element (somewhat pointless), slice it column-wise, row-wise, or element-wise:

$$\begin{aligned}
\texttt{toSeq}\ (\texttt{Z :. All :. All})\ A &= [\begin{pmatrix} 1 & 2 & 3 \\ 10 & 11 & 12 \end{pmatrix}] \\
\texttt{toSeq}\ (\texttt{Z :. All :. Split})\ A &= [\begin{pmatrix} 1 \\ 10 \end{pmatrix}, \begin{pmatrix} 2 \\ 11 \end{pmatrix}, \begin{pmatrix} 3 \\ 12 \end{pmatrix}] \\
\texttt{toSeq}\ (\texttt{Z :. Split :. All})\ A &= [\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}, \begin{pmatrix} 10 & 11 & 12 \end{pmatrix}] \\
\texttt{toSeq}\ (\texttt{Z :. Split :. Split})\ A &= [1, 2, 3, 10, 11, 12]
\end{aligned}$$

Fusion, as described in Section 3.2.1, is applied across sequentialisation. This means that, if matrix A is the result of a computation, we leverage the existing fusion transformation of Accelerate to combine it with any operation on A_i. In this way, we avoid the full manifistation of A.

If A is the result of an operation that prevents subsequent fusion, such as a reduction or a scan, we have no choice but to materialise the entire input array prior to slicing. This undesirable effect can sometimes be avoided by a simple transformation that moves the fusion-preventing operation into the sequence computation. As an example, consider the following program that converts a matrix into a sequence of row sums:

```
rowSums :: Acc (Array DIM2 Int)
        → Seq [Scalar Int]
rowSums mat =
  toSeqE (fold (+) 0 mat)
```

Since `fold` prevents further fusion in Accelerate, the result of `(fold (+) 0 mat)` will be a fully materialised vector, that happens to hold the entire sequence at once. If `mat` has a huge number of rows, full manifestation is catastrophic. However, it is entirely unnecessary and can be avoided in this case by first constructing a sequence of rows, and then mapping a sum over that sequence:

```
rowSums' mat =
  mapSeq
    (fold (+) 0)
    (toSeq (Z:.Split:.All) mat)
```

Assuming the user of this function provides an array expression `mat` that does not prevent further fusion, `mat` will be fused row-wise into the first operation of the sequence. Therefore, no initial manifestation is required and this definition works for arbitrary many rows. As a subject for future work, the compiler could potentially perform array-to-sequence expression transformations like this one. For now, as a rule of thumb when working with sequences, it is advisable to slice early and put as much of the program logic in `Seq` rather then in `Acc`. Finally, as a specific optimization for the GPU backend, if A is a host-side array constant, it will be transferred to the device in parts.

Sequences of arrays can be converted into flat data vectors and a vector containing the shape of each array, or to the data vector only if we are not interested in the shapes:

```
fromSeq :: (Shape ix, Elt a)
     ⇒ Seq [Array ix a]
     → Seq (Vector ix, Vector a)
```

Ordinary Accelerate array function can be lifted from working on arrays to working on sequences using *mapSeq* and *zipWithSeq*. These sequence combinators are parametrised by the to-be-lifted array function, and the denotation is simply to apply the function to each array of the input sequence(s).

```
mapSeq :: (Arrays a, Arrays b)
       ⇒ (Acc a → Acc b)
       → Seq [a]
       → Seq [b]

zipWithSeq :: (Arrays a, Arrays b, Arrays c)
           ⇒ (Acc a → Acc b → Acc c)
           → Seq [a]
           → Seq [b]
           → Seq [c]
```

The type class `Arrays` contains n-tuples of `Array` type, expressing the fact that the arguments of both operations can be multiple arrays.

In addition to mapping operations over sequences, we can fold a sequence of scalars with an associative binary array operator. Unlike with

map and zipWith, foldSeqE is not implemented in terms of a more general foldSeq. The reason why is explained in Section 3.4.

```
foldSeqE :: Elt a
         ⇒ (Exp a → Exp a → Exp a)
         → Exp a
         → Seq [Scalar a]
         → Seq (Scalar a)
```

Note that foldSeq still returns a sequence computation, but the result of that computation is a scalar array, not a sequence. This allows multiple reductions to be expressed and contained in the same sequence computation, ensuring a single traversal. For example, here we have two reductions, one summing the elements of an array, the other calculating the maximum. We can combine this into a single traversal with lift.

```
maxSum :: Seq [Scalar Float]
       → Seq (Scalar Float, Scalar Float)
maxSum xs = lift ( foldSeqE (+) 0 xs
                 , foldSeqE max 0 xs)
```

If we want to convert this back into an Acc value, we need to use collect:

```
collect :: Arrays arrs
        ⇒ Seq arrs → Acc arrs
```

Note the Arrays constraint on arrs in collect. As sequences are not members of the Arrays class this ensures that we cannot embed a whole sequence into an array computation without first reducing it to an array.

3.3.4 Lazy Lists to Sequences

Our language extension allows interfacing with ordinary Haskell lists. We define two convenient operations for converting sequence expressions to Haskell list and vice versa.

```
streamIn  :: Arrays a ⇒ [a] → Seq [a]
streamOut :: Arrays a ⇒ Seq [a] → [a]
```

streamIn is a language construct that takes a constant sequence and embeds it in Accelerate. It is the sequence-equivalent of an array constant in ordinary Accelerate. streamOut on the other hand, is an interpretation that runs a sequence expression and produces a Haskell list as output. Therefore, it must be defined on a per-backend basis, just like the ordinary Accelerate interpretation function run :: Arrays a ⇒Acc a →a. Using streamOut is

the only way to interpret a sequence expression that is not embedded in an array expression.

Accelerate is a strict language, and has not been equipped to deal with infinite sequences until now. Arrays are naturally finite, and the result sequence of `toSeq` is no longer than the size of the input array. However, there is nothing that prevents the programmer from passing an infinite list to `streamIn`. Being a strict language, Accelerate will go into an infinite loop if the programmer attempts to reduce an infinite sequence. It is however possible to productively stream out an infinite sequence to an infinite Haskell list. The elements will then be forced according to the evaluation strategy, which is hidden from the programmer. For example, if the programmer tries to print the third element of a streamed out sequence in Haskell, Accelerate may internally evaluate the first ten elements.

3.4 Execution Model

After discussing the language extensions for sequences, we are now looking into how we can generate efficient code from these sequence expressions. For a sequential CPU architecture, a sequential, element-by-element evaluation would be feasible, but would clearly lead to unacceptable performance on our main target architecture, GPUs. Instead, we want to process just enough data to saturate the GPU to achieve optimal performance. Therefore, before code generation, we group multiple elements of the sequence together in vectors to form *chunks*. Each chunk can then be streamed to the GPU and processed in parallel. The actual size of the chunk is chosen by the runtime, as the best choice depends on the concrete architecture the program is executed on.

We define a *chunk* to be a vector of arrays (or n-tuple of arrays) written with angular brackets $\langle A_1, ..., A_k \rangle$. Each array is required to have the same rank, but not necessarily the same shape. k is referred to as the length of the chunk, and the total size of the chunk elements $\sum_{i \in \{1..k\}} size(A_i)$ is referred to as the chunk size. If all the arrays have the same shape, we say that the chunk is regular. Note that a regular chunk is essentially just an array with rank $r + 1$ where r is the rank of each element.

The execution model presented here implements the programming model by translating sequence expressions to stream-manipulating acyclic dataflow graphs, where the nodes consume and/or produce chunks that flow along the edges. There are two key challenges in this approach: Lifting and scheduling. Sequence operations must be lifted at compile time to operate on chunks instead of just arrays. At run time, appropriate chunk lengths must be selected as small as possible while still keeping the backend satu-

rated in each step, and the sequence operations must be scheduled accordingly. We solve these challenges for regular chunks by means of *vectorisation* together with an analysis phase that yields a static schedule. We proceed to explain the vectorisation strategy of each primitive sequence operation.

- Array slicing is trivial to vectorise. `toSeq` is easily extended to produce chunks of slices, and the chunks will always be regular with known sizes.
- For `streamIn`, since Accelerate cannot track shapes, there is no guarantee that the list supplied by the programmer contains same-shape arrays, and consequently, we consider the resulting sequence to be irregular in all cases. One could imagine the addition of a `streamInReg` operation that takes the shape of elements as an additional argument. The programmer then promises that all arrays in the supplied list have this shape. Such an operation would be beneficial for applications streaming large amounts of regular data such as video processing.
- Sequence maps (and zipWith's) are vectorised by applying a lifting transformation on the argument array function as described in Section 3.4.2. Sequence maps are the main source of irregularity since we can map any array functions. As Accelerate cannot handle irregular arrays, we analyse the mapped array functions to detect irregularity and avoid chunking in that case. The analysis is described in Section 3.4.4.
- For sequence reduction with an array function as the combining operator, we need to turn an array-fold into a chunk-fold. Vectorizing the combining operator gives a function that combines chunks. We could fold the chunks of a sequence with this function and then use the unlifted function in the end to fold the final chunk. However, there are a number of problems with this approach:
  - The combining operator would have to be commutative since elements are combined, not with the next element in the sequence, but with the element in the next chunk at the same position.
  - It is not always desirable to keep a chunk of accumulated values. For example, `fromSeq` is a fold using array append as the combining operator, and the accumulated value is an array containing all the elements in the sequence seen so far. A chunk of accumulated values would be unreasonably large.

  A better solution is to fold each chunk with the unlifted function immediately and then combine the resulting folded value with the accumulator, again using the unlifted function. However, Accelerate does

not support a general parallel array fold. Instead, as described in the following paragraphs, we opt to provide a less general sequence fold primitive more suitable for chunking.

The sequence fold primitive is named `foldSeqFlatten` and has the type signature:

```
foldSeqFlatten :: (Arrays a, Shape sh, Elt b)
   ⇒ (  Acc a → Acc (Vector sh)
      → Acc (Vector b) → Acc a)
   → Acc a
   → Seq [Array sh b]
   → Seq a
```

This operation works by applying the given function to each chunk in a sequence. In each step, the function is applied to an accumulated value, a vector of shapes and a vector of elements, and it produces a new accumulated value. The vector of shapes is the shapes of the arrays in the input chunk, and the vector of elements is all the elements of the chunk concatenated to a flat vector. Our representation of chunks enables extracting the shape vector and element vector of a chunk in constant time, also for irregular chunks. This means that we can execute `foldSeqFlatten` on chunks of any length without having to vectorise. The operation unfortunately exposes the chunk size in the surface language as the size of the shape vector. This is not something the programmer should rely on since it is backend specific parameter. However, the programmer is obligated to obey the following rule for the folding function:

```
f (f a sh1 e1) sh2 e2 = f a (sh1 ++ sh2) (e1 ++ e2)
```

That is, applying the folding function twice on two shape and element vectors must be the same as applying it once on the appended vectors. This severely limits how the programmer can exploit knowing the chunk size.

Scalar fold `foldSeqE` is then defined as a prelude function using `foldSeqFlatten` and the standard `fold` operator of accelerate:

```
foldSeqE :: Elt a
         ⇒ (Exp a → Exp a → Exp a)
         → Exp a
         → Seq [Scalar a]
         → Seq (Scalar a)
foldSeqE f z =
  foldSeqFlatten
   (λ acc _ → fold f (the acc))
```

```
    (unit z)
```

Here `the` is convenience function for indexing a scalar array.

Likewise, `fromSeq` is also currently also a prelude function. Folding with append is not very efficient, so we plan to specialise this operation in the near future.

```
fromSeq :: (Shape ix, Elt a)
        ⇒ Seq [Array ix a]
        → Seq (Vector ix, Vector a)
fromSeq = foldSeqFlatten f (lift (empty, empty))
  where
    f x sh1 a1 =
      let (sh0, a0) = unlift x
      in lift (sh0 ++ sh1, a0 ++ a1)
```

Here `empty` produces the empty vector and (++) is vector append.

3.4.1 Translation

Sequence expression are first converted to A-normal form where producing sequences are let-bound, and the sequence arguments in a sequence operation must be variables. For example, the sequence dot-product example is converted into the form:

```
let s1 = toSeqE xs
    s2 = toSeqE ys
    s3 = zipWithSeqE (*) s1 s2
in foldSeqE (+) 0 s3
```

If a sequence expression contains more than one consumer, they are grouped together in a n-tuple. The A-normal expression is then traversed from top to bottom and translated into the following continuation-passing-style executable representation:

```
data StreamDAG senv res where
  Transduce
    :: (senv → s → (a, s))
    → s
    → (s → Bool)
    → StreamDAG (senv, a) res
    → StreamDAG senv res

  Consume
```

```
      :: (senv → s → s)
      → s
      → StreamDAG senv s

    Reify
      :: (senv → [a])
      → StreamDAG senv [a]
```

The type variable `senv` refers to the surrounding sequence context that holds the current live values in each step, and `res` is the final result of the entire stream execution.

`Transduce` is a stream transformer with a local state `s` that produces values of type `a` from the surrounding context `senv`. Sequence maps, `toSeq` and `streamIn` translates to transducers. In each step of the stream, the termination condition of type

```
(s -> Bool)
```

is checked. It is only `toSeq` and `streamIn` that can terminate a sequence, and they do so once there are no more chunks to produce. If the stream is not ready to terminate, the stepping function of type

```
(senv -> s -> (a, s))
```

will be applied to the current context and state to produce a value of type `a` that is made available to the subsequent nodes. The functional also produces a new state to be used in the next iteration. The subsequent nodes are then stepped by recursively stepping the argument of type

```
StreamDAG (senv, a) res.
```

`Consume` also carries a local state that is updated in each step. The final state in a consumer will be the result of the sequence. Once termination is reached, the result can be read from the value of the consumer. `foldSeqFlatten` is currently the only operation that translates to a consume. Multiple folds are combined in a single consume node.

`Reify` is a special kind of consumer that produces a list of values in each step. This constructor is only used when a sequence is streamed out. In this case, instead of a final result, the sequence produces a list of intermediate results. The function argument converts a chunk from the surrounding sequence context into a list of arrays. These lists are appended together to form the result of `streamOut`. When an element belonging to a chunk is forced in the host language, the whole chunk is forced along with all preceding chunks that have not been forced already.

The stepping functions in the stream DAG are mostly obtained by evaluating the array-typed arguments of the operations in the sequence using

the existing backend. After vectorisation is applied, the translation becomes straight-forward. The only new backend-specific operation we had to define are related to slicing and streaming in and out.

3.4.2 Vectorization

Operations applied to the elements of a sequence in the source program have to be applied in parallel to all the elements of a chunk when we execute the code on a GPU. This process of lifting element-wise operations to a parallel operation on a collection, referred to as the *lifting transform* [BS90], was popularised by NESL [Ble92].

Lifting, for a fully featured, higher-order functional language is a complicated process and can easily introduce inefficiencies [KCL+12, LCK06]. However, the constrained nature of the Accelerate array language works in our favour here, and we can get the job done with a much simpler version of the transformation. While the user-facing surface language may appear to be higher order, it is strictly first-order. That is to say, the only higher order functions are primitive operations (e.g. *map*, *zipWith*, *fold*) and functions cannot be let bound.

The transformation $\mathcal{L}[\![f]\!]$, which lifts a (potentially already parallel) function $f :: \alpha \to \beta$ into vector space has type:

$$\mathcal{L}[\![f]\!]_{xs} :: \texttt{Vector}\ \alpha \to \texttt{Vector}\ \beta$$

Conceptually, it is just a parallel map. However, we cannot implement it as such, since f may already be a parallel operation, and feeding it to a parallel map would result in nested parallelism, precisely what we want to avoid. In order to maintain flat parallelism, the lifting transform must recurse over the term applying the transformation to all subterms. Formally, this is defined as follows.

$$\begin{aligned}
\mathcal{L}[\![\mathbf{C}]\!]_{v:_} &= \mathit{replicate}\ (\mathit{length}\ v)\ \mathbf{C} \\
\mathcal{L}[\![x]\!]_{v:vs} &= \begin{cases} x & x \in (v : vs) \\ \mathit{replicate}\ (\mathit{length}\ v)\ x & x \notin (v : vs) \end{cases} \\
\mathcal{L}[\![\lambda v.\ e]\!]_{vs} &= \lambda v.\ \mathcal{L}[\![e]\!]_{v:vs} \\
\mathcal{L}[\![e_1\ e_2]\!]_{vs} &= \mathcal{L}[\![e_1]\!]_{vs}\ \mathcal{L}[\![e_2]\!]_{vs} \\
\mathcal{L}[\![\mathbf{let}\ v = e_1 \mathbf{in}\ e_2]\!]_{vs} &= \mathbf{let}\ v = \mathcal{L}[\![e_1]\!]_{vs} \mathbf{in}\ \mathcal{L}[\![e_2]\!]_{v:vs} \\
\mathcal{L}[\![\mathbf{P}]\!]_{_} &= \mathbf{P}^{\uparrow}
\end{aligned}$$

Here, the lifting transform, $\mathcal{L}[\![\cdot]\!].$, takes a term along with a subset of its environment, expressed as a list of variables. The subset corresponds to the variables that have been lifted as part of the transform. For example, given `mapSeq` f *seq*, we want to vectorise f, but f may contain variables that were bound outside of the sequence computation, in which case we have to replicate them at their use sites. Similarly, constants, **C**, are also replicated at their use sites.

In the case of primitive operations, **P**, we use a lifted version of that operation, $\mathbf{P}^{\uparrow}$ defined in terms of existing primitives. Fortunately, the range of combinators Accelerate provides is rich enough that lifted versions of all of them are able to be implemented, with our chosen nested array representation.

Nested Array Representation

A consequence of this flattening transform is that it produces nested vectors. For example lifting

$$g :: \texttt{Vector Float} \to \texttt{Vector Int}$$

is going to yield

$$\mathcal{L}[\![g]\!] :: \texttt{Vector}(\texttt{Vector Float}) \to \texttt{Vector}(\texttt{Vector Int}).$$

Indeed, with the multidimensional arrays accelerate supports, arrays of type `Vector (Array (Z:.Int:.Int) Float)` could occur. One possible, and elsewhere very popular, flattened array representations is this: A vector of arrays is represented as a vector of segment descriptors (the shape of the sub-array) along with a vector containing all the values of the sub-arrays flattened and concatenated. For example:

$$\left[\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}, \begin{pmatrix}10 & 11 & 12\\ 13 & 14 & 15\end{pmatrix}\right]$$

$$([2 :. 2, 3 :. 2)], [1, 2, 3, 4, 10, 11, 12, 13, 14])$$

However, this representation is unnecessarily general for our model. As our execution model only allows chunked execution for *regular* chunks, with this representation, all the segment descriptors would be the same. Instead, we use a much simpler representation: A regular vector of arrays of rank *sh* can be represented as an array of rank $sh :. Int$. Of course, this affects the lifting transform. If we only allow regular vectors then not all array functions can be lifted. For example, this function cannot be lifted.

```
enumFromTo :: Int → Int → Vector Int
```

The reason why is its lifted form does not produce a regular nested vector. The size of each sub-vector of the output is determined by the contents of the input vectors.

```
ℒ⟦enumFromTo⟧. :: Vector Int
                → Vector Int
                → Vector (Vector Int)
```

We classify an array function as regular if the shape of the output, as well as the shape of any intermediate arrays, can be determined from the shape of the input and, assuming the function is open, its environment. We describe in Section 3.4.4 how we identify these functions and compute their parallel degree.

This regular nested array representation is also of benefit when it comes to defining the lifted version of the built in primitives. Typically, the lifting transform relies heavily on prefix-sums (scans) to perform these *segmented* operations [Ble90a]. Forgoing the segment descriptors means that, for example, our lifted `fold`, can be implemented purely in terms of regular fold. This is because it is already rank polymorphic.

```
fold :: (Exp e → Exp e)
     → Exp e
     → Array (sh:.Int) e
     → Array sh e
```

That is to say, it folds along the inner dimension, reducing the rank by one.

While most of Accelerate's primitives are rank polymorphic, there are some notable exceptions. Primarily, prefix-sums (scans) are not rank polymorphic. However, we can use segmented scans [CBZ90] and pass in a vector of segment descriptors that are all the same length.

Implementation

Accelerate is realised as a type preserving embedded language where closed terms are indexed by their type, and open terms are indexed by their type and their environment. Here is a simplified version of the core AST.

```
data OpenAcc aenv t where
  Avar :: Idx aenv t
       → OpenAcc aenv t
  Alet :: OpenAcc aenv bnd
       → OpenAcc (aenv,bnd) body
  Map  :: Exp aenv (a → b)
```

```
        → OpenAcc aenv (Array sh a)
        → OpenAcc aenv (Array sh b)
```

The `Idx aenv t` supplied to `Avar` represents a DeBruijn index of type `t` into the environment `aenv`.

```
data Idx aenv t where
  ZeroIdx :: Idx (aenv,t) t
  SuccIdx :: Idx aenv     t
          → Idx (aenv,s) t
```

A closed term can then be represented with the empty environment.

```
type Acc = OpenAcc ()
```

In order to represent the nested regular arrays that result from the lifting transform, we introduce a type family, `Regular`, that encodes the non-parametric representation. [1]

```
type family Regular t where
  Regular (Array sh e) = Array (sh:.Int)
  Regular (a,b)        = (Regular a, Regular b)
  Regular (a,b,c)
    = (Regular a, Regular b, Regular c)
  ...
```

Of course, our lifting transform will, by necessity change a terms environment. We can capture this change in environment by encoding the lifting context as follows.

```
data Context aenv aenv' where
  Base       :: Context aenv aenv
  Push       :: Context aenv aenv'
             → Context (aenv, t) (aenv', t)
  PushLifted :: Context aenv aenv'
             → Context (aenv,t) (aenv', Regular t)
```

Here, `Push` represents an unlifted variable and `PushLifted` one that was lifted into regular vector space. Our actual lifting transform then takes this form.

```
liftAcc :: Context aenv aenv'
        → OpenAcc aenv t
        → OpenAcc aenv' (Regular t)
```

[1] In our implementation `Regular` is encoded with some slight differences, due to the use of *representation* types, but that is orthogonal to the transform we describe.

This concludes the vectorisation. With this function, we can lift any Accelerate expression, including array functions, into a vectorised equivalent that works on regular chunks. We use this to achieve chunked stream transducers as described previously. It remains to show how streams are scheduled.

3.4.3 Scheduling

If a sequence is regular with element sizes n, a chunk of length k will have size $n * k$. Assuming this size corresponds to the parallel degree required to compute the chunk (we have to be a bit more careful here which we will return to shortly), if n is known, k can be fixed for all chunks in a sequence, consistently saturating the hardware. If on the other hand, a sequence is not regular, the size of elements, and thereby the optimal value of k, varies for each chunk, and the scheduler must re-calculate k in each step.

Right before executing a stream, we perform regularity analysis on the corresponding sequence expression that computes the element size n or reports that the sequence is potentially irregular. If n is found, we set $k = k_{\text{opt}}/n$ where k_{opt} is a constant that defines the optimal chunk size for the given hardware. For SIMD backends, k_{opt} is related to the number of processors and how many scalar elements each processor operates on in each step. We also multiply by a constant factor to reduce scheduling overhead. If n cannot be determined, we currently fall back to purely sequential (static) scheduling with $k = 1$, which is always possible. We plan to improve on this in the future.

As mentioned earlier, size does not always correspond to parallel degree. For example, summing a n-length vector in $\log(n)$ steps will have a parallel degree of at most n in each step and a result size of 1. If we sequence-map a vector-sum over a sequence of vectors s, regardless of s, the resulting sequence is trivially regular with element size 1 (the size of scalars). By fixing the chunk length to $k_{\text{opt}}/1 = k_{\text{opt}}$, one chunk depends on k_{opt} vectors in s. This means that the chunk length of s would have to be at least k_{opt}. However, the vectors of s may be arbitrarily large, and as a result, each step of the stream requires arbitrarily many computational resources, including working memory, which is what we are trying to avoid. The right approach here is to check that s is regular first, and if so, use that elements size to select the chunk length. Furthermore, since we allow mapping any array function over a sequence, the function may internally create large arrays that we have to keep track of as well.

In principal, each producer and consumer in a stream DAG may have its own optimal chunk length. Ideally, streams should be processed at different rates and communicate with each other using buffers that are either filling

or draining at different points in time. Static scheduling in a multi-rate context has been covered in the signal processing community [LM87], but here, the rates are known a priori, and the schedule is then constructed. Our problem is slightly different. We have a trivial schedule (the purely sequential one), but we would like to pick the rates such that each step runs with optimal parallel degree. We take a simpler approach for now: By fixing the chunk length globally across all stream nodes to be the smallest of the optimal chunk lengths of each node, the schedule becomes a single simple loop. The downside is that some nodes may not saturate the hardware, however, the node with the smallest optimal chunk length (the one that will run optimally), is almost always the most costly node to execute.

3.4.4 Parallel Degree and Regularity Analysis

We perform the regularity analysis by traversing the sequence expression at runtime, prior to executing the corresponding stream DAG. At this point, we have access to the arrays in the surrounding context, that we may use to refine the analysis. The traversal is essentially interpreting array functions using partial arrays as defined by the following type interpretation (encoded as a GADT in the actual implementation):

$$\mathcal{C}[\![\texttt{Array}\ sh\ e]\!] = (\texttt{Maybe}\ sh, \texttt{Maybe}\ (sh \to e))$$
$$\mathcal{C}[\![(\alpha_1, ..., \alpha_n)]\!] = (\mathcal{C}[\![\alpha_1]\!], ..., \mathcal{C}[\![\alpha_n]\!])$$

We allow an element lookup function in the type of a partial array, but we will only use it if it is a constant time operation, such as if the array is already manifest. The cost interpretation of a an array expression *acc* of type α in a surrounding array context *aenv* is a function:

$$\mathcal{C}[\![aenv \vdash acc : \alpha]\!] : \mathcal{C}[\![aenv]\!] \to (\mathcal{C}[\![\alpha]\!], \texttt{Maybe Int})$$

Here, the result type $(\alpha, \texttt{Maybe Int})$ is a writer monad where the accumulator type `Maybe Int` represents the parallel degree or `Nothing` if it cannot be determined. Two parallel degree are combined by maximum. If even a single intermediate result has an unknown shape, we cannot predict the parallel degree of the expression. Array functions are distinguished syntactically from base array expressions, and the above interpretation does not apply to functions. Instead array functions are interpreted by extending the surrounding context:

$$\mathcal{C}[\![aenv \vdash \lambda acc : \alpha \to \beta]\!] = \mathcal{C}[\![(aenv, \alpha) \vdash acc : \beta]\!]$$

The definition of $\mathcal{C}[\![-]\!]$ is:

$$
\begin{aligned}
&\mathcal{C}[\![c]\!]v && = (\texttt{Just}\ (\mathit{shape}\ c), \texttt{Just}\ (\lambda x.c!x)) \\
&\mathcal{C}[\![\textbf{let}\ \mathit{acc}_1\ \textbf{in}\ \mathit{acc}_2]\!]v && = \mathcal{C}[\![\mathit{acc}_1]\!]v \gg\!= (\lambda x.\mathcal{C}[\![\mathit{acc}_2]\!](v, x)) \\
&\mathcal{C}[\![x]\!]v && = v(x) \\
&\mathcal{C}[\![\lambda \mathit{acc}]\!]v && = \lambda x.\mathcal{C}[\![\mathit{acc}]\!](v, x) \\
&\mathcal{C}[\![\mathit{acc}_1\ \mathit{acc}_2]\!]v && = \mathcal{C}[\![\mathit{acc}_2]\!]v \gg\!= \mathcal{C}[\![\mathit{acc}_1]\!]v \\
&\mathcal{C}[\![\texttt{Map}\ \_\ \mathit{acc}]\!]v && = \mathcal{C}[\![\mathit{acc}]\!]v \gg\!= \\
& && \quad (\lambda(\mathit{sh}, \_). \\
& && \quad\ ((\mathit{sh}, \texttt{Nothing})), \mathit{fmap}\ \mathit{size}\ \mathit{sh}) \\
& && \quad \vdots
\end{aligned}
$$

Here v ranges over partial array contexts $\mathcal{C}[\![\mathit{aenv}]\!]$, *fmap* is the standard functor map over `Maybe`, and $(\gg\!=)$ (bind) is standard bind operator for the writer monad. *size* computes the size of a shape.

The cost analysis of a sequence expressions requires shape information for the sequence in the surrounding sequence context. For the purpose of detecting regularity, we do not need to track additional information besides the element type of a sequence type $[\alpha]$:

$$\mathcal{C}[\![[\alpha]]\!] = \mathcal{C}[\![\alpha]\!]$$

We assume that the sequence expression is in A-normal form. Note that we are analysing the unlifted array functions before vectorisation. The analysis for a sequence expression *seq* in surrounding sequence context *senv* and array context *aenv* has the following signature:

$$
\begin{aligned}
\mathcal{C}[\![\mathit{aenv}, \mathit{senv} \vdash \mathit{seq} : \alpha]\!] :\ & \mathcal{C}[\![\mathit{aenv}]\!] \\
\to\ & \mathcal{C}[\![\mathit{senv}]\!] \\
\to\ & (\mathcal{C}[\![\alpha]\!], \texttt{Maybe Int})
\end{aligned}
$$

It is defined as follows:

$$
\begin{aligned}
&\mathcal{C}[\![\texttt{ToSeq}\ \mathit{div}\ \mathit{acc}]\!]v\ \_ \\
&\quad = (\mathit{sl}, \mathit{fmap}\ \mathit{size}\ \mathit{sl}) \\
&\quad \text{where} \\
&\qquad ((\mathit{sh}, \_), \_) = \mathcal{C}[\![\mathit{acc}]\!]v \\
&\qquad \mathit{sl} = \mathit{fmap}\ (\mathit{sliceShape}\ \mathit{div})\ \mathit{sh} \\
&\mathcal{C}[\![\texttt{StreamIn}\ \mathit{xs}]\!]\_\ \_ = \texttt{Nothing} \\
&\mathcal{C}[\![\texttt{MapSeq}\ f\ x]\!]v\ w = \mathcal{C}[\![f]\!](v, w(x)) \\
&\mathcal{C}[\![\texttt{ZipWith}\ f\ x\ y]\!]v\ w = \mathcal{C}[\![f]\!](v, w(x), w(y)) \\
&\mathcal{C}[\![\texttt{FoldSeqFlatten}\ f\ \mathit{acc}\ x]\!]v\ w \\
&\quad = \mathcal{C}[\![f]\!](v, (\mathit{sh}_a, \texttt{Nothing}), \mathit{shs}, \mathit{elts}) \\
&\quad \text{where} \\
&\qquad ((\mathit{sh}_a, \texttt{Nothing}), \_) = \mathcal{C}[\![\mathit{acc}]\!]v \\
&\qquad (\mathit{sh}_x, \_) = w(x) \\
&\qquad \mathit{shs} = (\texttt{Just}\ (\texttt{Z}\ \texttt{:.}\ 1), \mathit{fmap}\ \mathit{unit}\ \mathit{sh}_x) \\
&\qquad \mathit{elts} = (\mathit{fmap}\ ((\texttt{Z}\ \texttt{:.}) \circ \mathit{size})\ \mathit{sh}_x, \texttt{Nothing})
\end{aligned}
$$

Here w ranges over partial sequence context $\mathcal{C}[\![\mathit{senv}]\!]$ tracking sequence element sizes. *sliceShape* computes the slice shape from a given source shape and division strategy. The array argument *acc* in both the `ToSeq` case and the `FoldSeqFlatten` case are not evaluated in each step of the sequence. We therefore discard the parallel degree reported by the size analysis on these - We are only interested in the shapes.

The current definition of the `FoldSeqFlatten` case is admittedly faulty. It performs the analysis using the shape of the initial accumulator. If the accumulator grows, such as when folding with vector append, the actual parallel degree may very well be much larger than the parallel degree we report here. A correct definition should check that the shape of the initial accumulator is equal to the shape of the new accumulator, and only report a parallel degree if that is the case. However, that would cause the analysis to fail for our current definition of `FromSeq`. We therefore omit this check until we have a better implementation of `FromSeq` or until we have better support for dynamic scheduling.

3.5 Evaluation

In order to evaluate the performance of our implementation, we give the results from two sets of benchmarks. Firstly, three smaller benchmarks where

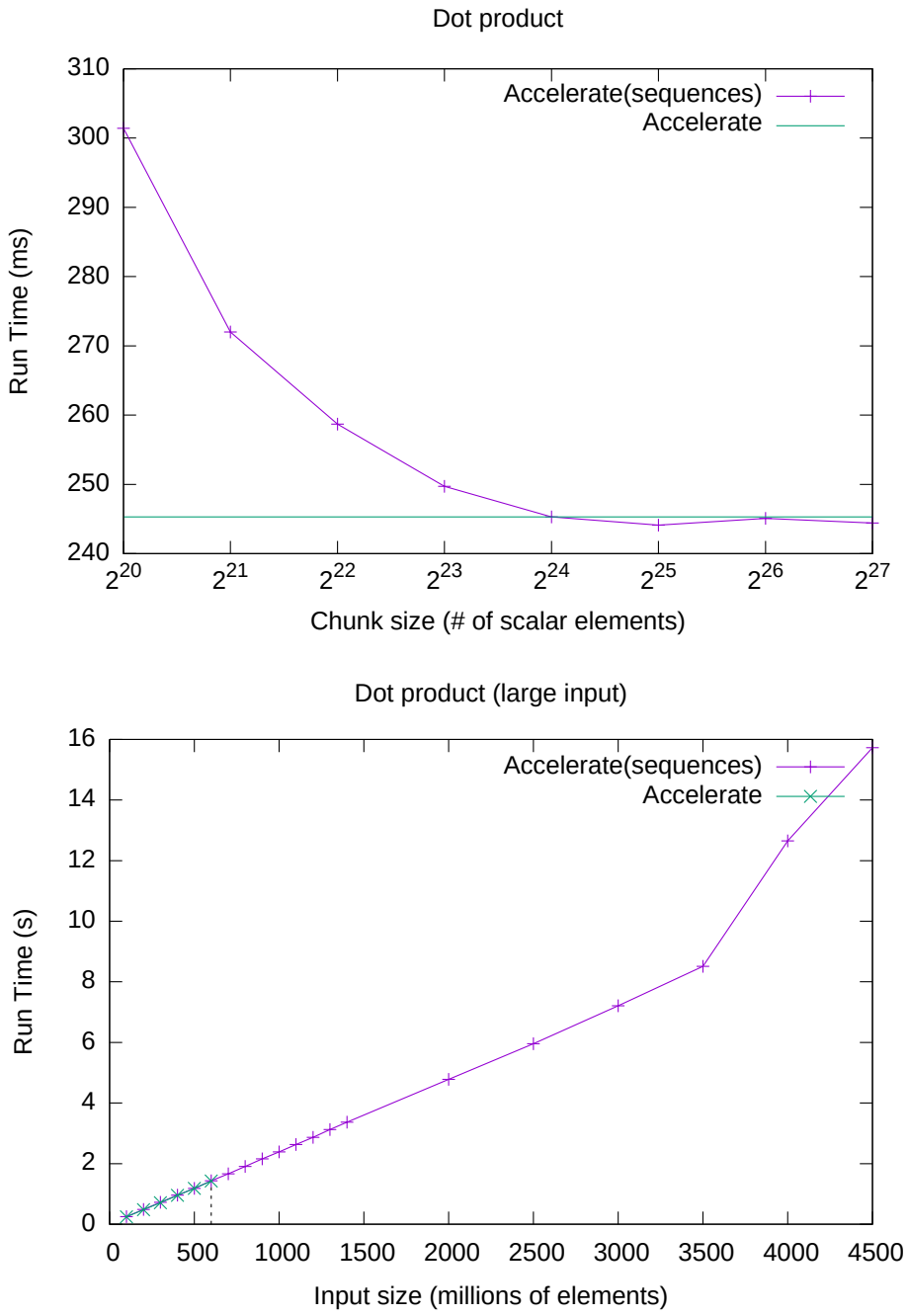


Figure 3.1: Dot product benchmark results

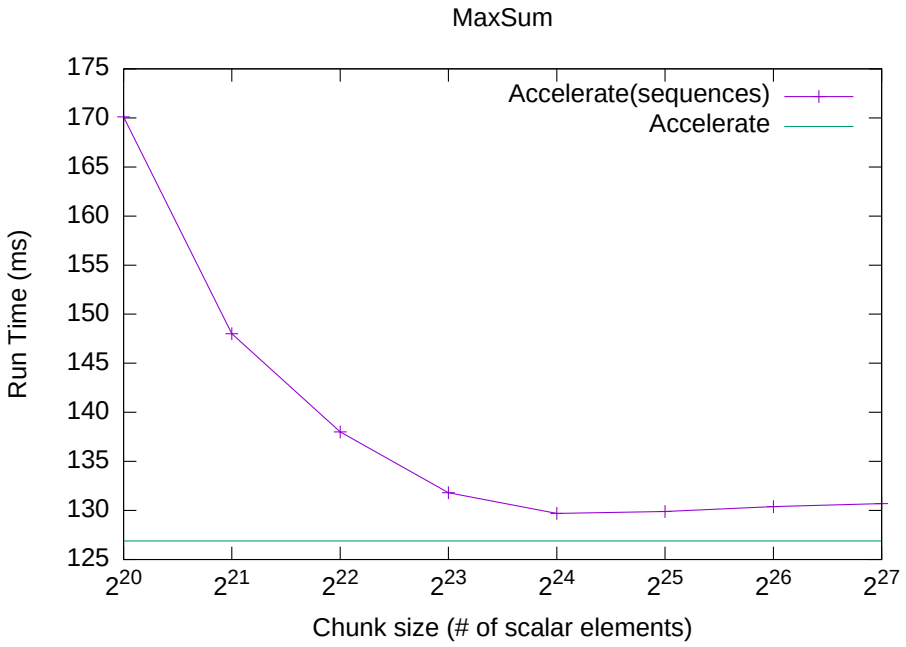


Figure 3.2: MaxSum benchmark results

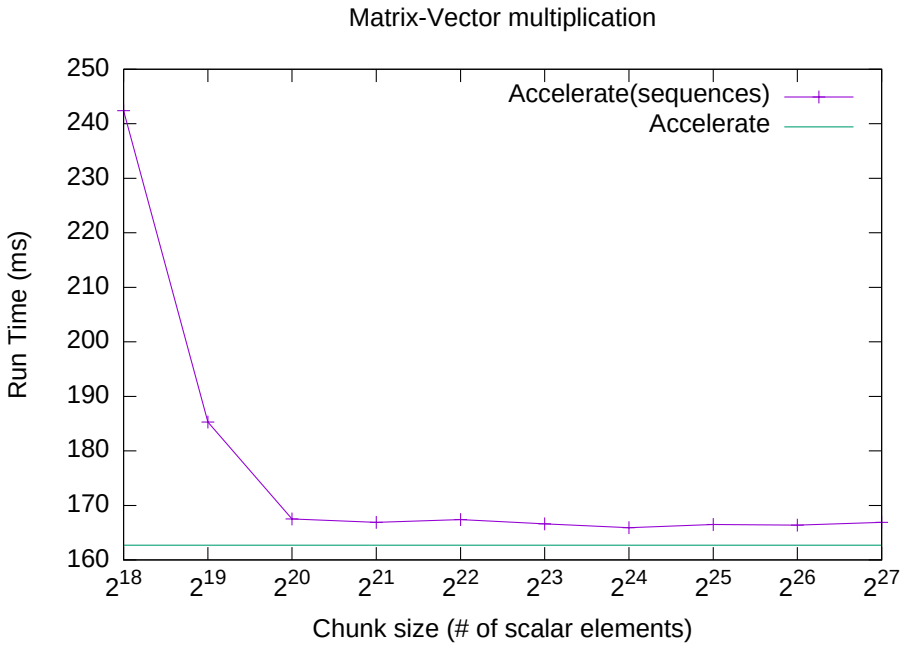


Figure 3.3: MVM benchmark results

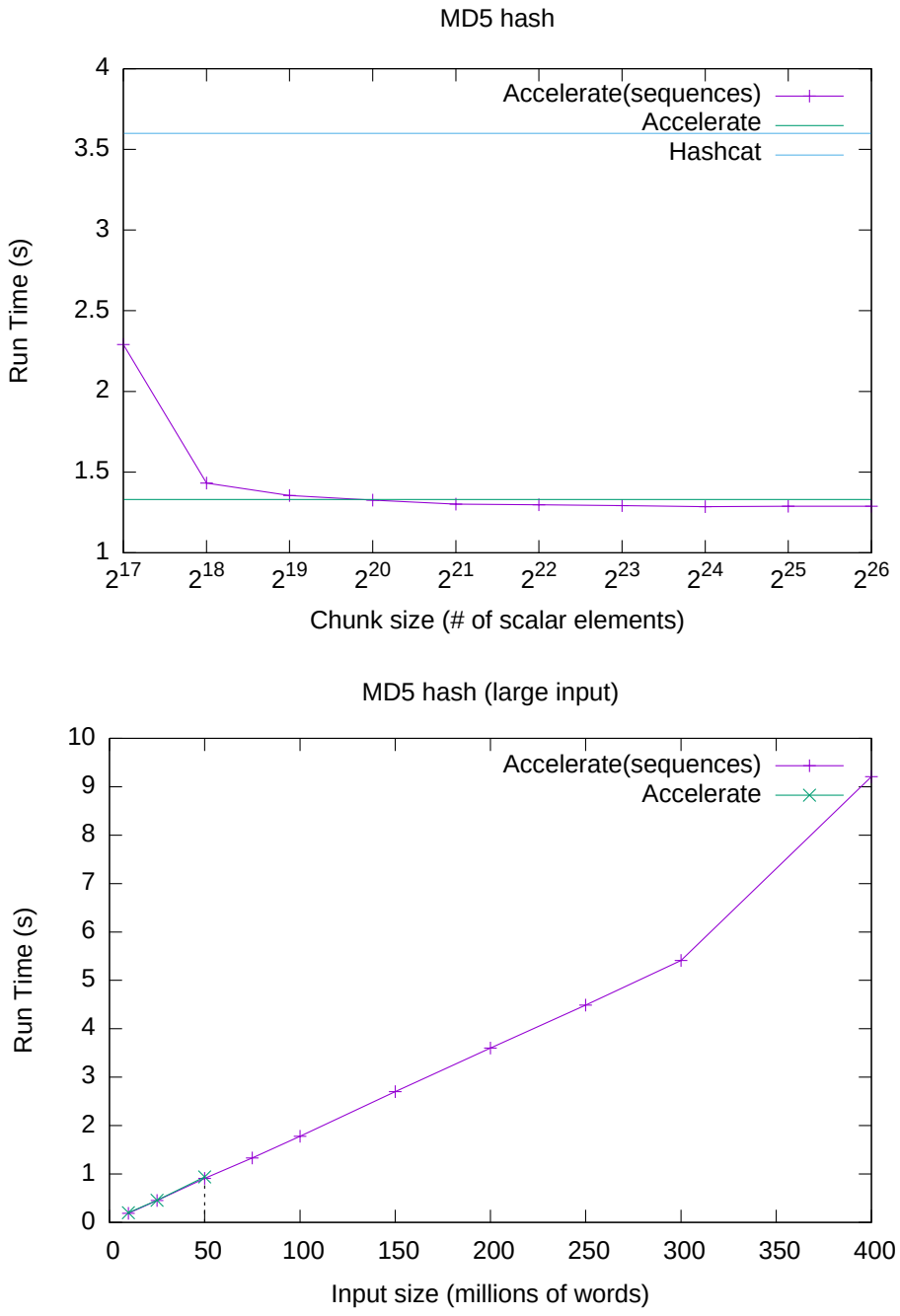


Figure 3.4: MD5 Hash benchmark results

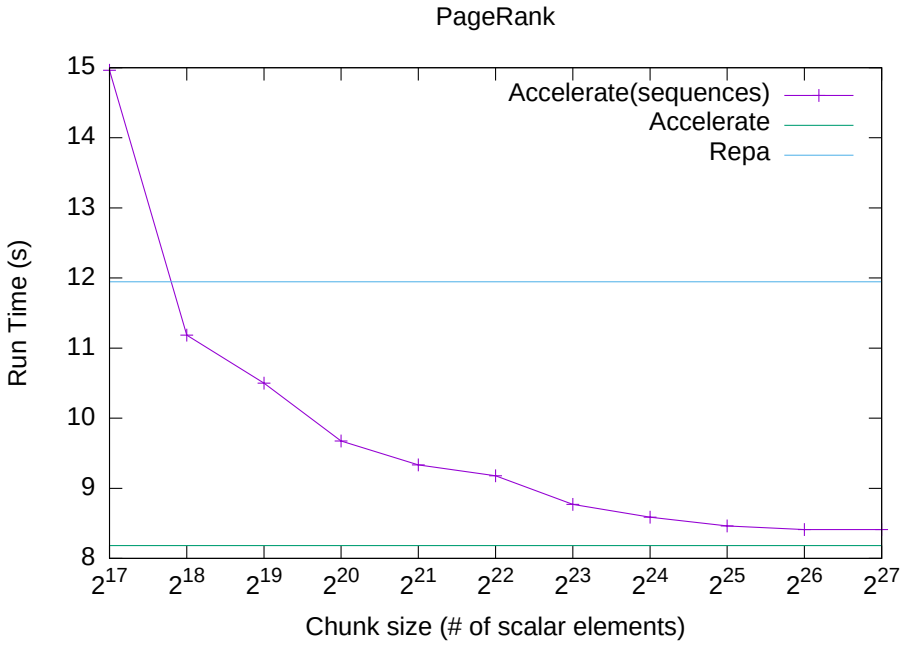


Figure 3.5: PageRank benchmark results

we compare the performance of accelerate sequences against arrays in order to show that no performance is lost by using sequences. Secondly, we demonstrate two larger scale applications, comparing them both with Accelerate array implementations and other CPU implementations. All benchmarks were run on a single Tesla K20c (compute capability 3.5, 13 multiprocessors = 2496 cores at 705.50 MHz, 5 GB RAM). The host machine has 4 8-core Xeon E5-2650 CPUs (64-bit, 2.00GHz, 64GB ram, with hyperthreading). Our benchmark times include time to transfer the data onto the GPU, the execution time, and the time to transfer data back to the host, but not compilation time. The fact that Accelerate is online compiled is orthogonal to what we present here, and compiled kernels are cached to prevent unnecessary recompilation.

3.5.1 Dot Product

This is simply the example we show in Section 3.3 against the version that does not use sequences. While this example is very simple, it helps highlight that there is no significant loss of performance due to the overheads associated with scheduling sequence computations. At the ideal chunk-size, 2^{24} scalar elements, performance is essentially equal. It is worth noting that this is significantly less than the total size of the data which is 100 million

floating point values in each input vector. It is not till 2^{27}, that the chunk size covers all input.

In the second graph, where the chunk size is fixed at 2^{24}, we see that normal Accelerate runs out of memory with input vectors of 700 million floating point numbers. However, with sequences we are able to process much larger inputs with no noticeable overhead. That is until it exceeds 3.5 billion in which case it runs out of physical memory on the host and pages start getting swapped to disk.

3.5.2 MaxSum

Here we demonstrate how, even when applying two separate reductions over a sequence, only one pass is ever made over the input. The program is simply:

```
maxSumSeq :: Vector Float
          → Acc (Scalar Float, Scalar Float)
maxSumSeq xs = collect
             $ lift ( foldSeqE (+) 0 xs'
                    , foldSeqE max 0 xs')
  where xs' = toSeqE xs
```

We compare it against a version written without sequences.

```
maxSumSeq :: Vector Float
          → Acc (Scalar Float, Scalar Float)
maxSumSeq xs = lift (fold (+) 0 xs, fold max 0 xs)
```

Here, `lift` converts `(Seq a, Seq b)` into `Seq (a,b)`, or `(Acc a, Acc b)` into `Acc (a,b)`, depending on the context.

Even though only one pass is made over the input as a whole, this example does not outperform normal Accelerate due to limitations in Accelerate's fusion system. Because `foldSeqE` applies `fold` over each chunk, two traversals of each chunk is made. The sort of *horizontal* fusion that would resolve this would be advantageous in this instance.

Once again, we apply this function over 100 million floating point values.

3.5.3 MVM

This is the matrix-vector multiplication example shown earlier. This is not just a simple reduction, but uses a sequence map, so requires vectorisation. Once again, we compare it to a version written without sequences.

```
mvm :: Acc (Matrix Float) → Acc (Vector Float)
```

```
      → Acc (Vector Float)
mvm mat vec
  = let h = fst (unindex2 (shape mat))
    in fold (+) 0
    $ zipWith (*) mat (replicate (lift (Z:.h:.All)) vec)
```

The net result in this case is that, after subsequent optimisation passes, both examples produce almost identical code. The only difference being index manipulation. From the graph it can be observed that, even with very small chunk sizes, the version with sequences runs in close to the same time as the version without. The input is a matrix of 10000×10000 floating point values.

3.5.4 MD5 Hash

The MD5 message-digest algorithm [Riv92] produces a 128-bit cryptographic hash. For the purposes of establishing the ideal chunk size we apply this algorithm to each word in a dictionary of 64 million words, attempting simple password recovery. Like MVM above, we generate very similar code to a version written without sequences. However, in this case, slight differences in index manipulation actually work in our favour, giving us minor improved performance over the contender. We also compare our results to that of Hashcat, a CPU-based password recovery tool.

To demonstrate the effectiveness of Accelerate sequences or out-of-core algorithms, we also run this benchmark against dictionaries of varying size. As can be seen, if sequences are not used, only a dictionary of up to 75 million words will fit in the GPUs memory. If they are used, however, we can use dictionaries up to an almost arbitrary size. Although, like the dot product example, if the dictionary is larger than can fit in physical host memory then there is slowdown due to paging. This occurs at around 400 million words.

3.5.5 PageRank

While our current restriction on sequences being regular is not ideally suited to graph processing, we can still implement graph algorithms, like PageRank [PBMW99] by changing the way we represent a graph. In this case, we represent it as a sequence of links (edges). This is a much heavier, in terms of space, than a representation using a sparse matrix in compressed sparse row (CSR) format, however it is still performant. Comparing our results to that of a CPU-based implementation written in Repa [KCL+10, LCKPJ12], we see that while it can use a more space-efficient representation, we still outperform it in terms of overall speed.

For the benchmark, we use a dump of the Wikipedia link graph that contains approximately 130 million links.

3.6 Related Work

There are a number of languages which aim at facilitating GPU programming in the presence of data streams. For example, the Brook[Buc03] stream programming language is an extension of C with support for data parallel computations. BrookGPU [BFH+04] is an implementation of a subset of Brook which targets GPUs.

Brook uses only streams to express parallelism.

Sponge[HSW+11] is a compilation framework for GPUs using the synchronous data flow streaming language StreamIt[TKA02].

Both Sponge and BrookGPU are based on a very different programming model. Starting from a stream programming model, the compiler and runtime system tries to execute the code efficiently on a GPU architectures. We provide the programmer both with parallel arrays, which support a much wider range of parallel operations, and more restricted streams. This distinction enables the programmer to design algorithmic solutions for a problem better tailored towards the actual architecture, while still providing a high level of abstraction.

Though it does not target GPUs, the Proteus[PPW95] language uses an idea similar to chunking to restrict the memory requirements of vectorisation based implementations of nested data-parallel programs.

While there are a number of domain specific languages to support GPU programming, some of them also embedded in Haskell, [Ell04, MM10, CSS08, Lar11, RSA+13, BR12] to name some, none of them currently, to the best of our knowledge, support streams.

A conceptual stream model for NESL has been presented in previous work [MF13]. The model is motivated by the lack of a good realizable space cost model, and it is conceptually similar to ours; Sequences are exposed at the source level and execute in streams of bounded chunks. The model is more general than ours in the sense that it allows nested sequences. However, it has yet to be implemented with a high-performance backend. Furthermore, the work presented here features more operations on multi-dimensional arrays.

3.7 Future Work

The work presented in this paper is a first step towards full streaming support for collection-oriented parallel languages targeting GPUs. Our next

steps will be to lift some of the current restrictions of the model, as well as further improving the performance of the generated code and runtime system. In particular, we are planning to address the following issues in the near future:

- Chunked execution of irregular sequences: This would allow us to express more algorithms over irregular data, like graph processing, but will require more sophisticated scheduling strategies than in the regular case.
- More combinators: We currently only support a minimal set of combinators in Accelerate, and there are a number of obvious additions, as for example scans on non-scalar sequences which we will add.
- Automatic sequentialisation: As mentioned in the introduction, the whole point of exposing sequences to the programmer is that optimal automatic sequentialisation is generally undecidable. That does not mean that the compiler should not attempt it however. Guided by a size analysis, the compiler could opportunistically transform array expressions into equivalent sequence traversals in the presence of very large intermediate arrays.

With respect to performance improvements:

- We do not yet attempt to transfer the data for the next chunk while we process the current one. This overlapping of communication and computation could potentially give significant improvements to overall runtime [GLGLBG12].
- As we have already mentioned, our model lends itself very well to multi-gpu support via pipeline-parallelism. This is our ultimate aim, but it does carry with it challenges in regards to fusion of sequence computations– i.e. that sequence operations should not be always fused, but rather split up amongst devices where possible.

Chapter 4

Streaming Nested Data Parallelism on Multicores

This chapter is adapted from [MF16].
There are significant changes in all section except Section 4.5: Experiments and Section 4.6: Conclusion and Future Work.

Abstract

The paradigm of nested data parallelism (NDP) allows a variety of semi-regular computation tasks to be mapped onto SIMD-style hardware, including GPUs and vector units. However, some care is needed to keep down space consumption in situations where the available parallelism may vastly exceed the available computation resources. To allow for an accurate space-cost model in such cases, we have previously proposed the Streaming NESL language, a refinement of NESL with a high-level notion of streamable sequences.

In this paper, we report on experience with a prototype implementation of Streaming NESL on a 2-level parallel platform, namely a multicore system in which we also aggressively utilize vector instructions on each core. We show that for several examples of simple, but not trivially parallelizable, text-processing tasks, we obtain single-core performance on par with off-the-shelf GNU Coreutils code, and near-linear speedups for multiple cores.

4.1 Introduction

While the semantics of original NESL language does not preclude a streaming SIMD implementation [PPCF95], or even a MIMD-oriented approach [BG96], the language is not particularly well suited for such an implementa-

tion strategy. The reason is that the cost model makes no formal distinction between operations that are streamable, and ones that require random access to vector elements, and thus the programmer may be encouraged to use idioms (such as accessing the last element of a vector as $v[\#v - 1]$) that needlessly force full materialization.

Therefore, in order to investigate the potential for streaming execution, we have previously [MF13] (Chapter 2) proposed a refinement of NESL, tentatively called Streaming NESL (SNESL), that makes streamability apparent in the programming and cost models. The main difference to NESL is that the language syntax and type systems makes a clear distinction between *sequences*, which are conceptually always traversed in order (including by reduces and scans), and *vectors* that afford random access and multiple traversals, but must be materialized, with a corresponding space cost.

In the above-mentioned previous work, we presented very preliminary timings on GPUs for hand-transformed code. While the numbers were not inherently discouraging, it became clear that the CUDA API is not well suited as a backend for SNESL, since kernel launches are relatively expensive, requiring chunks to be big in order to achieve good performance, which means that each chunk can no longer fit in on-chip cache. Even more problematically, as of CUDA 7.5, "shared" (on-chip, per-thread-block) memory does not persist across kernel invocations, leading to considerable extra memory traffic. While it might be possible to utilize the hardware more efficiently through a lower-level interface, this is would require substantial further development.

NESL *without* streaming on GPUs has been explored by [RS]. Previously [MCECK15] (Chapter 3), we implemented streaming execution on GPUs, albeit for a different language: Accelerate with a streaming extension similar to the one we propose for NESL. Accelerate only supports limited nesting of data parallelism in the form of regular multi-dimensional arrays. Streaming Accelerate performs on par with ordinary Accelerate, but it relies heavily on the regularity of the language and aggressive fusion.

In this paper, we consider the prospect of streaming on multicore CPUs while exploiting SIMD instructions. We show that, for several representative tasks, even on a single core, the SNESL execution time is comparable to that of a sequential C program, while performance on multiple cores handily exceeds the sequential code.

The section on SNESL from the original publication has been left out here. We instead jump straight to the backend language SVCODE and refer the reader to Chapter 2 for a description of SNESL.

$$
\begin{array}{rcl}
\texttt{ctrl} & :: & \mathbf{1} \to S\mathbf{1} \\
\texttt{map\_}\oplus & :: & S\pi_1 \times \cdots \times S\pi_k \to S\pi \qquad \oplus :: \pi_1 \times \cdots \pi_k \to \pi \\
\texttt{segscan\_}R & :: & S\mathsf{Bool} \times S\pi \to S\pi \qquad R :: \pi \times \pi \to \pi \\
\texttt{to\_flags} & :: & S\mathsf{Int} \to S\mathsf{Bool} \\
\texttt{dist} & :: & \forall\pi. S\pi \times S\mathsf{Bool} \to S\pi \\
\texttt{pack} & :: & \forall\pi. S\pi \times S\mathsf{Bool} \to S\pi \\
\texttt{unpack} & :: & \forall\pi. S\pi \times S\pi \times S\mathsf{Bool} \to S\pi \\
\texttt{segtab} & :: & \forall\pi. S\pi \times S\mathsf{Bool} \to S^{\mathsf{u}}\pi \\
\texttt{so\_gather} & :: & \forall\pi. S\mathsf{Int} \times S\mathsf{Int} \times S^{\mathsf{u}}\pi \to S\pi \\
\texttt{usum} & :: & S\mathsf{Bool} \to S\mathbf{1} \\
\texttt{usegsum} & :: & S\mathsf{Bool} \times S\mathsf{Bool} \to S\mathsf{Bool} \\
\texttt{cons}_a & :: & S\pi \to S\pi \qquad a :: \pi \\
\texttt{segmerge}_k & :: & \forall\pi. \overbrace{(S\mathsf{Bool} \times S\pi) \times \cdots \times (S\mathsf{Bool} \times S\pi)}^{k} \to S\pi \quad k>0 \\
\texttt{usegmerge}_k & :: & \overbrace{(S\mathsf{Bool} \times S\mathsf{Bool}) \times \cdots \times (S\mathsf{Bool} \times S\mathsf{Bool})}^{k} \to S\mathsf{Bool} \quad k>0 \\
\texttt{check} & :: & S\mathsf{Bool} \to \mathbf{1} \\
\texttt{write\_file}_{\mathit{filepath}} & :: & S\mathsf{Char} \to \mathbf{1} \\
\texttt{read\_file}_{\mathit{filepath}} & :: & \mathbf{1} \to S\mathsf{Char}
\end{array}
$$

Figure 4.1: Core stream instruction with types.

4.2 Streaming VCODE (SVCODE)

As described in Section 2.3, recall that the backend language for Streaming NESL is a flat list of stream-defining instructions. To emphasize the analogy to the NESL compiler, we adapt its name for the flat-sequence language. However, our SVCODE bears little syntactic resemblance to VCODE, mainly because the former is currently just a flat list of instructions of the form

$$\mathsf{s}_i := p_i\ \mathsf{s}_{i_1}\ \ldots\ \mathsf{s}_{i_k}$$

where all $i_1, \ldots i_k$ are less than i, representing previously defined primitive-value streams. On the other hand, VCODE is explicitly stack-structured, to support recursive functions.

Let p range over primitive instruction names, s range over stream identifiers (natural numbers), t (for "transducer") range over stream instructions

and c (for "commands") range over SVCODE programs. The formal grammar for SVCODE programs is quite simple and is given by:

$$\begin{array}{rcll} p & ::= & \texttt{map\_}\oplus \mid \texttt{segscan\_}R \mid \cdots & \text{(see Figure 4.1)} \\ \textbf{INSTR} \ni t & ::= & p_i\ \mathsf{s}_{i_1}\ \ldots\ \mathsf{s}_{i_k} & \\ \textbf{SVCODE} \ni c & ::= & \epsilon \mid c_1; c_2 \mid \mathsf{s}_i := t & \end{array}$$

The full list of instructions in our implementation is bigger, as it includes specialized instructions that we generate through various SVCODE-to-SVCODE optimization. Although we present the types for instructions as polymorphic here, SVCODE is explicitly typed. Every stream in SVCODE has primitive type of the form $S\pi$ for some primitive SNESL type π. Streams of tuples are represented as tuples of streams. SVCODE programs can be type checked by simply keeping track of the defined streams' types while going through the list of instructions from top to bottom and check that the argument types match.

We will now demonstrate the semantics of some of the instructions through examples. `map_`$\oplus$, `segscan_`R, `dist` and `pack` should be fairly self-explanatory from their types.

The `ctrl` instruction simply creates the initial control stream:

$$[\![\texttt{ctrl}]\!] = \langle * \rangle$$

`to_flags` is used to convert a length-based segment descriptor to a flag-based one. This need arises when converting vectors to sequences or in the translation of &:

$$[\![\texttt{to\_flags}]\!]\ \langle 3,4,0,1\rangle = \langle \mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{T},\mathsf{F},\mathsf{T}\rangle$$

`segtab` is an atypical instruction. It defines an operational segmentation or grouping of elements in a stream. It is used when tabulating sequences. Essentially, the instruction takes a flag-segmented stream and returns the data stream, but where the segment information is recorded in the runtime system for the purpose of defining the buffer size. In the following example

$$[\![\texttt{segtab}]\!]\ \langle \texttt{'a'},\texttt{'b'},\texttt{'c'},\texttt{'d'},\texttt{'e'}\rangle\ \langle \mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T}\rangle = \langle \overbrace{\texttt{'a'},\texttt{'b'}}, \overbrace{\texttt{'c'},\texttt{'d'},\texttt{'e'}}\rangle,$$

the runtime system will allow `'a','b'` to be manifest at the same time and also `'c','d','e'`. This means, if the chunk size is 1, the runtime system will first grow the buffer to size 2, then to size 3. If the chunk size is 10, `segtab` will have no effect in this example as both groupings can fit in the buffer at the same time. These streams that have an operational grouping, are marked by superscripting an u on the stream type constructor, the u

stands for "unbounded" as these streams are statically unbounded in their buffer size. They may be used in place of any ordinary ungrouped (or bounded) streams. However, there is one instruction that explicitly requires an unbounded stream as input: `so_gather`. `so_gather` or "semi-ordered gather" is basically a gather operation, but where the gathering indices are divided into an offset and a positive relative index. The offsets are then required to be monotonically increasing. Consider the following example:

$$\begin{aligned} [\![\texttt{so\_gather}]\!] \quad & \langle 0,0,0,3,3,3\rangle \\ & \langle 2,1,0,1,1,1\rangle \\ & \langle \overbrace{\texttt{'a'},\texttt{'b'},\texttt{'c'}}, \overbrace{\texttt{'d'},\texttt{'e'}}\rangle \\ = \quad & \langle \texttt{'c'},\texttt{'b'},\texttt{'a'},\texttt{'e'},\texttt{'e'},\texttt{'e'}\rangle \end{aligned}$$

Here, he value streams has two groupings: `'a','b','c'` and `'d','e'`. The offsets for these grouping are 0 and 3. We see that the first three indices refer to the first group while the last 3 indices refers to the second group. Once the runtime system reaches the point where the offset becomes 3, it may safely free the group `'a','b','c'` by removing the values from the buffer, since monotonicity implies that they will never be indexed again.

The `usum` instruction conceptually performs a sum of a stream of unary numbers:

$$[\![\texttt{usum}]\!]\ \langle \mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{T}\rangle = \langle *,*,*\rangle$$

Here, the input stream represents the numbers $0,2,1$ in unary. The sum of these numbers is 3, so `usum` outputs 3 unit elements. `usum` is used to define control streams that defines the current parallel degree.

`usegsum` performs a unary segmented sum, and it is used in the translation $concat_\sigma$ to construct the new segment descriptor:

$$\begin{aligned} [\![\texttt{usegsum}]\!] \quad & \langle \mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{T}\rangle \\ & \langle \mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T}\rangle \\ = \quad & \langle \mathsf{T},\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T}\rangle \end{aligned}$$

In this example, the first stream is a segment descriptor defining 3 segments of the second stream. The first segment is empty, the next segment contains 2 unary values $\overbrace{\mathsf{F},\mathsf{F},\mathsf{T}}, \overbrace{\mathsf{F},\mathsf{F},\mathsf{T}}$ (representing 2, 2), and the last segment contains a single unary value $\mathsf{F},\mathsf{F},\mathsf{T}$ (representing 2). We then "sum" each segment by appending the unary values. So, since it is empty, the first segment sums to 0, which is just T. The second segment sums to 4 which is $\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T}$ and the last segment sums to 2 which is $\mathsf{F},\mathsf{F},\mathsf{T}$.

The `cons` instruction pushes an element in the front of a stream. The last element in the stream is discarded. Cons can be used to right-shift the

elements of a stream by one, i.e. creating a delay of 1 in the stream:

$$[\![\texttt{cons}]\!]_1\ \langle 2,3,4\rangle = \langle 1,2,3\rangle$$

It can be used to convert an end-flag representation to a head-flag representation. To see how, consider cons'ing T onto the segment descriptor representation of (0,2,1), T, F, F, T, F, T:

$$[\![\texttt{cons}]\!]_{\mathsf{T}}\ \langle \mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{T}\rangle = \langle \mathsf{T},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{F}\rangle.$$

Here, the the result represents the same segmentation, but where the T-flags are given in the start position instead of the end position.

$\texttt{segmerge}_k$ merges k segmented streams in a round-robin fashion. For example:

$$\begin{aligned}
[\![\texttt{segmerge}_2]\!] \quad & \langle \mathsf{F},\mathsf{F},\mathsf{T},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{FT}\rangle \\
& \langle \texttt{'a'},\texttt{'a'},\texttt{'a'},\texttt{'a'},\texttt{'a'}\rangle \\
& \langle \mathsf{F},\mathsf{T},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{T}\rangle \\
& \langle \texttt{'b'},\texttt{'b'},\texttt{'b'}\rangle \\
= \quad & \langle \texttt{'a'},\texttt{'a'},\texttt{'b'},\texttt{'b'},\texttt{'a'},\texttt{'a'},\texttt{'a'},\texttt{'b'}\rangle
\end{aligned}$$

Here, we merge a stream of `'a'`-characters with a stream of `'b'`-characters. We start by selecting a segment from the first stream: the `'a'`-stream. The first segment is F, F, T, or 2, which means we first select two `'a'`-characters for the result stream. We hen select a segment from the `'b'`-stream. The first segment here is F, T, so we simply select a single `'b'` for the next element of the result. The result, so far, is then `'a'`, `'a'`, `'b'`. The next segment in the `'a'`-stream is empty (T), so we select no elements an proceed to select the next segment in the `'b'`-stream, which gives us `'a'`, `'a'`, `'b'`, `'b'`. We continue this process until both streams are exhausted.

$\texttt{usegmerge}_k$, or unary segmented merge, is similar to $\texttt{segmerge}_k$, except that the value streams are unary numbers, and we merge segments of unary numbers instead of segments of single values.

The `check` instruction is the way we implement runtime failure in SVCODE. It gives a unit result if all values in the input stream are T:

$$[\![\texttt{check}]\!]\ \langle \mathsf{T},\mathsf{T},\mathsf{T},...,\mathsf{T},\mathsf{T}\rangle = *$$

If at least one value is F, the semantics are left undefined, and the runtime system aborts with an error.

$\texttt{read\_file}_{filepath}$ and $\texttt{write\_file}_{filepath}$ allows interfacing with external files and pipes. A file may be used as a stream of characters, and a stream of characters may be written to a file. The runtime system performs reads and writes incrementally as the buffers are filled and emptied.

4.3 SNESL to SVCODE

The translation from SNESL to SVCODE is based on the interpretation described in Chapter 2 Section 2.3.2. Instead of interpreting the primitive stream operations as the mathematical stream they denote, we generate SVCODE instructions. Thus, the definition is almost identical but with different interpretations for the primitive operations and where we use a monad to capture fresh identifiers and instruction generation.

The translation runs in a monad that allows us to generate fresh stream identifies and write SVCODE fragments:

$$\mathbf{Gen}\ a = \mathbf{N} \to (a, \mathbf{SVCODE}, \mathbf{N})$$

In Haskell terms, **Gen** is a state monad where the state is **N** combined with a writer monad that writes **SVCODE**. Note that **SVCODE** is a monoid where ϵ is the identity and append is given by:

$$\begin{aligned} \epsilon +\!\!+ c &= c \\ c +\!\!+ \epsilon &= c \\ c_1 +\!\!+ c_2 &= c_1; c_2 \end{aligned}$$

We define the standard bind and return as:

$$return\ x\ i = (x, \epsilon, i)$$

$$\begin{aligned} bind\ f\ g\ i = let \quad & (x, c_1, i') = f\ i \\ & (y, c_2, i'') = g\ x\ i' \\ in \quad & (y, c_1 +\!\!+ c_2, i'') \end{aligned}$$

For better readability, we will make use of the standard Haskell do-notation for complex usage of *bind*:

$$do\{x \leftarrow f; g\ x\} \equiv bind\ f\ g$$

Gen defines a special function that allows us to emit instructions. The type of *emit* is

$$emit : \mathbf{INSTR} \to \mathbf{Gen\ SId},$$

where **SId** are stream identifiers. The argument is an instruction of the form $p\ \mathsf{s}_{i_1}\ ...\ \mathsf{s}_{i_k}$ where the s_{i_j}'s will be previously emitted stream identifiers. This is only enforced after translation by a type check on the generated SVCODE. The definition of *emit* is:

$$emit(p\ \mathsf{s}_{i_1}\ ...\ \mathsf{s}_{i_k})\ i = (i, \mathsf{s}_i := p\ \mathsf{s}_{i_1}\ ...\ \mathsf{s}_{i_k}, i+1)$$

$\boxed{\mathcal{S}\,[\![e]\!]\,\zeta : \mathbf{SId} \to \mathbf{Gen}\ (\mathbf{Tree\ SId})}$

$$
\begin{aligned}
\mathcal{S}\,[\![x]\!]\,\zeta\ s &= return\ (\zeta\ x) \\
\mathcal{S}\,[\![a]\!]\,\zeta\ s &= rep\ s\ a \\
\mathcal{S}\,[\![(x_1,...,x_k)]\!]\,\zeta\ s &= return\ (\zeta x_1,...,\zeta x_k) \\
\mathcal{S}\,[\![x.i]\!]\,\zeta\ s &= let\ (t_1,...,t_k) = \zeta\ x\ in\ return\ t_i \\
\mathcal{S}\,[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]\,\zeta\ s &= do\ \{st \leftarrow \mathcal{S}\,[\![e_1]\!]\,\zeta\ s;\ \mathcal{S}\,[\![e_2]\!]\,\zeta[x \mapsto st]\ s\} \\
\mathcal{S}\,[\![\phi(x)]\!]\,\zeta\ s &= \mathcal{S}\,[\![F]\!]_\phi\ (\zeta\ x)\ s \\
\mathcal{S}\,[\![\{e_0 : x\ \mathbf{in}\ x_0\ \mathbf{using}\ x_1^{\tau_1},...,x_k^{\tau_k}\}]\!]\,\zeta\ s\ = & \\
\quad let\ (t,s') = \zeta\ x_0\ in & \\
\quad\ do\ s'' \leftarrow usum\ s' & \\
\quad\quad st_1 \leftarrow dist_{\tau_1}\ (\zeta x_1)\ s' & \\
\quad\quad \vdots & \\
\quad\quad st_k \leftarrow dist_{\tau_k}\ (\zeta x_k)\ s' & \\
\quad\quad st \leftarrow \mathcal{S}\,[\![e_0]\!]\,[x \mapsto t, (x_i \mapsto st_i)_{i=1}^k]\ s'' & \\
\quad\quad return\ (st,s') & \\
\mathcal{S}\,[\![\{e_0 \mid x_0\ \mathbf{using}\ x_1^{\sigma_1},...,x_k^{\sigma_k}\}]\!]\,\zeta\ s\ = & \\
\quad let\ s_0 = \zeta\ x_0\ in & \\
\quad\ do\ s' \leftarrow b2u\ s_0 & \\
\quad\quad s'' \leftarrow usum\ s' & \\
\quad\quad st_1 \leftarrow pack_{\sigma_1}\ (\zeta x_1)\ s_0 & \\
\quad\quad \vdots & \\
\quad\quad st_k \leftarrow pack_{\sigma_k}\ (\zeta x_k)\ s_0 & \\
\quad\quad st \leftarrow \mathcal{S}\,[\![e_0]\!]\,[(x_i \mapsto st_i)_{i=1}^k]\ s'' & \\
\quad\quad return\ (st,s') &
\end{aligned}
$$

Figure 4.2: Translating SNESL to SVCODE.

When *emit* is used in conjunction with *bind*, an SVCODE program is constructed incrementally.

We can now define our primitives as simple instruction emitters:

$$rep\ s\ a = emit(\texttt{map\_const}\ a\ s)$$

$$usum\ s = emit(\texttt{usum}\ s)$$

$$pdist\ s_1\ s_2 = emit(\texttt{pdist}\ s_1\ s_2)$$

And so on. Note that `map_const` is a map instruction that maps the units in the control stream to the constant a.

For the purpose of encoding high-level structure in low-level representations, we use a tree algebraic data structure:

$$\textbf{Tree}\ a ::= a \mid (\textbf{Tree}\ a, ..., \textbf{Tree}\ a)$$

We adopt the notation of a postfix t on meta-variables that ranges over trees. For example, while s ranges over **SId**, st ranges over **Tree SId**.

Translation of expressions is given in Figure 4.2. This definition is almost identical to the translation given in Chapter 2. The calligraphic $\mathcal{S}$ refers to SVCODE and $\mathcal{S}[\![e]\!]$ can be read as "the SVCODE of e". The translation of e takes the form

$$\mathcal{S}[\![e]\!]\,\zeta\ i = (st, c, i').$$

Here, i is the next fresh identifier, st is the result (a tree of stream identifiers representing a high-level value), c is the generated program and i' is the next fresh identifier after code generation of e. ζ is a mapping from SNESL variables to trees of stream identifiers:

$$\zeta ::= [x_1 \mapsto st_1, ..., x_k \mapsto st_k].$$

In ζ, each x_i of type σ_i is mapped to an st_i of type $\mathcal{S}[\![\sigma_i]\!]$, where $\mathcal{S}[\![\sigma]\!] = \pi t$ maps the general type σ to a tree of primitive types πt, defined by (this definition is identical to the definition in Section 2.3.1):

$$\begin{aligned}
\mathcal{S}[\![\mathsf{Bool}]\!] &= \mathit{SBool} \\
\mathcal{S}[\![\mathsf{Int}]\!] &= \mathit{SInt} \\
\mathcal{S}[\![\mathsf{Real}]\!] &= \mathit{SReal} \\
\mathcal{S}[\![(\sigma_1, ..., \sigma_k)]\!] &= (\mathcal{S}[\![\sigma_1]\!], \cdots, \mathcal{S}[\![\sigma_k]\!]) \\
\mathcal{S}[\![[\tau]]\!] &= (\mathcal{S}[\![\tau]\!], (\mathit{SInt}, \mathit{SInt})) \\
\mathcal{S}[\![\{\sigma\}]\!] &= (\mathcal{S}[\![\sigma]\!], \mathit{SBool})
\end{aligned}$$

An important point is the translation of individual builtin primitives. The translation of a primitive ϕ is given by $\mathcal{S}[\![F]\!]_\phi$ which takes a stream tree and the control stream, and produces a new stream tree in the **Gen** monad. We will now give examples of the translation of $scan_R$, $reduce_R$, &, tab_τ and seq_τ.

Scan Scan is straight-forward to translate using the builtin segmented scan.

$$\begin{aligned}
\mathcal{S}[\![F]\!]_{scan_R}\ (s_0, s_1)\ _ = do\quad & s_1' \leftarrow \mathit{short_form}\ s_1 \\
& s_0' \leftarrow \mathit{emit}(\texttt{segscan}_R\ s_0\ s_1') \\
& \mathit{return}\ (s_0', s_1)
\end{aligned}$$

Since scan in SNESL has the type $\{\pi\} \to \{\pi\}$ for some reduction operator $R : \pi \times \pi \to \pi$, the translation of scan accepts two stream identifiers (s_0, s_1) where s_0 is a stream of values of type π and s_1 is an end-flag segment descriptor. Due to the possibility of empty segments, segment descriptors use a representation where there is one additional flag for each segment. On the other hand, in order to have an efficient implementation, segmented scan is defined for segment descriptors where the length of the segment descriptor is the same as the length of the value stream. This is why we have a *short_form* function. This function gives the short form of a general form. It is implemented using `pack` and `cons`. To give an example, the two representation forms of $\{3,1,4\}, \{\}, \{1\}, \{5,9\}$ are

$$\begin{aligned}&\text{General form: } (\langle 3,\ 1,\ 4,\ 1,\ 5,\ 9\rangle, \langle \mathsf{F},\ \mathsf{F},\ \mathsf{F},\ \mathsf{T},\ \mathsf{T},\ \mathsf{F},\ \mathsf{T},\ \mathsf{F},\ \mathsf{F},\ \mathsf{T}\ \rangle)\\ &\text{Short form:}\quad (\langle 3,\ 1,\ 4,\ 1,\ 5,\ 9\rangle, \langle \mathsf{F},\ \mathsf{F},\ \mathsf{T},\quad \mathsf{T},\quad \mathsf{F},\ \mathsf{T}\quad \rangle)\end{aligned}$$

Notice that we lost the information of the presence of $\{\}$ in the short form. In our actual implementation, it is not always necessary to compute the short form as some operations provide it for free. We therefore track what segment descriptors already have a short form defined, and we only compute it once if we have to compute it at all.

Reduce Although it would be an optimization, we do not currently provide segmented reduction as a primitive, and we do not have to. We can simply perform a segmented scan and pack the last values into a resulting stream. However, some care must be taken. An empty sequence does not have any values for us to pack. There are a couple of ways to get around this. One of the least efficient ways, but an asymptotically correct and a simple way to do this is to unpack the identity element in the original value stream. By adding the identity element to the end of each segment in the value stream, the segment descriptor becomes the short form for the extended value stream. This translation is given by:

$$\begin{aligned}\mathit{segreduce}\ s_0\ s_1 = \mathit{do}\quad & s_2 \leftarrow \mathit{rep}\ s_1\ R_{id}\\ & s_3 \leftarrow \mathit{emit}(\texttt{map\_not}\ s_1)\\ & s_4 \leftarrow \mathit{emit}(\texttt{unpack}\ s_2\ s_0\ s_3)\\ & s_5 \leftarrow \mathit{emit}(\texttt{segscan\_}R\ s_4\ s_1)\\ & \mathit{emit}(\texttt{pack}\ s_5\ s_1)\end{aligned}$$

$$\mathcal{S}\,[\![F]\!]_{\mathit{reduce}_R}\ (s_0, s_1)\ \_ = \mathit{segreduce}\ s_0\ s_1$$

We have extracted *segreduce* as a separate function since we will use it later.

There is amble room for optimizations in this definition. If there are many empty or almost empty segments, unpacking the identity element is

costly. Here, it would be better to unpack the identity element after the pack. Another thing we do in our implementation, is to special case the unlifted version to perform a simple reduction, using a non-segmented reduction primitive. We also allow the SNESL programmer to use a special reduction $reduce1_R$ that has undefined behavior on empty sequences. This allows us to ignore the empty case and provide a much simpler definition:

$$\begin{aligned}\mathcal{S}\,[\![F]\!]_{reduce1_R}\ (s_0, s_1)\ \_ = do\quad & s_1' \leftarrow short\_form\ s_1 \\ & s_0' \leftarrow emit(\texttt{segscan}_R\ s_0\ s_1') \\ & emit(\texttt{pack}\ s_0'\ s_1')\end{aligned}$$

Iota The basic translation of (lifted) iota (&) is to perform a segmented scan of 1's. First, we convert the input stream, which defines the lengths of each segment, to flag representation using the builtin operation `to_flags`. We then generate a constant stream of 1's with the same length as the number of elements we need, and finally we perform a segmented scan. Iota can cause runtime failure if the argument is negative. We therefore check it using the `check` instruction: Finally, we attach the segment descriptor and we are done:

$$\begin{aligned}iotas\ s = do\quad & s_1 \leftarrow usum\ s \\ & s_2 \leftarrow rep\ s\ 1 \\ & s' \leftarrow short\_form\ s \\ & emit(\texttt{segscan}_+\ s_2\ s')\end{aligned}$$

$$\begin{aligned}\mathcal{S}\,[\![F]\!]_{\&}\ s_0\ s = do\quad & s_{\text{zeros}} \leftarrow rep\ s\ 0 \\ & s_{\text{check}} \leftarrow emit(\texttt{map\_geq}\ s_0\ s_{\text{zeros}}) \\ & emit(\texttt{check}\ s_{\text{check}}) \\ & s_2 \leftarrow emit(\texttt{to\_flags}\ s_0) \\ & s_3 \leftarrow iotas\ s_2 \\ & return\ (s_3, s_2)\end{aligned}$$

In our actual implementation, we consider unlifted iota as a special case and have a special `range` instruction that is much more efficient. Another possible optimization that we do not currently implement is to have a specialized version of segmented scan that specializes the scanned values (not the segment descriptor) to a constant. In this way, we can avoid having to generate the constant stream of ones.

Tab In SNESL, tab_τ converts a sequence of τ-typed values to a vector. In SNESL, this corresponds to changing the segment descriptor representation and `segtab`'ing the values in order to specify that vectors are fully manifest. In the base case, a sequence of primitive type π is tabulated by:

$$psegtab\ s_{\text{v}}\ s_{\text{sd}} = emit(\texttt{segtab}\ s_{\text{v}}\ s_{\text{sd}})$$

To support the full polymorphic version of tab_τ, we define a function inductively over the concrete type τ:

$$\begin{aligned}
segtab_\pi\ s_0\ s &= psegtab\ s_0\ s \\
segtab_{(\tau_1,...,\tau_k)}\ (st_1,...,st_k)\ s &= do\ st_1' \leftarrow segtab_{\tau_1}\ st_1\ s \\
&\quad\ \vdots \\
&\quad\ st_k' \leftarrow segtab_{\tau_k}\ st_k\ s \\
&\quad\ return\ (st_1',...,st_k') \\
segtab_{[\tau]}\ (st_0,(s_\mathrm{s},s_\mathrm{l}))\ s &= do\ s_\mathrm{s}' \leftarrow psegtab\ s_\mathrm{s}\ s \\
&\quad\ s_\mathrm{l}' \leftarrow psegtab\ s_\mathrm{l}\ s \\
&\quad\ return\ (st_0,(s_\mathrm{s}',s_\mathrm{l}'))
\end{aligned}$$

We also need to convert a flag segment descriptor to a virtual segment descriptor. Luckily, sequences do not use virtual segmentation, so the virtual segment descriptor that we generate, is really a normal length-based segment descriptor where the offsets are a simple scan of the lengths.

$$\begin{aligned}
to_lens\ s_0 = do\quad & s_1 \leftarrow usum\ s_0 \\
& s_2 \leftarrow rep\ s_1\ 1 \\
& segreduce_+\ s_2\ s_0
\end{aligned}$$

We define a scan using segmented scan:

$$\begin{aligned}
scan_R\ s_0 = do\quad & s_1 \leftarrow rep\ s_0\ \mathsf{F} \\
& emit(\texttt{segscan}_R\ s_0\ s_1)
\end{aligned}$$

The final translation is then:

$$\begin{aligned}
\mathcal{S}\,[\![F]\!]_{tab_\tau}\ (st_0,s_\mathrm{sd})\ s = do\quad & s_\mathrm{l} \leftarrow to_lens\ s_\mathrm{sd} \\
& s_\mathrm{s} \leftarrow scan_+\ s_\mathrm{l} \\
& s' \leftarrow short_form\ s \\
& st_1 \leftarrow segtab_\tau\ st\ s' \\
& return\ (st_1,(s_\mathrm{s},s_\mathrm{l}))
\end{aligned}$$

Seq In SNESL, seq_τ converts a vector of τ-typed values to a sequence. In SNESL, this corresponds to changing the segment descriptor representation from virtual segmentation (offsets and lengths) to segment end flags. We therefore also have to sequence the segments by gathering them, since we can no longer represent virtual segmentation. For this purpose, SVCODE provides the `so_gather` operation.

$$so\_gather\ s_\mathrm{s}\ s_\mathrm{i}\ s_\mathrm{v} = emit(\texttt{so\_gather}\ s_\mathrm{s}\ s_\mathrm{i}\ s_\mathrm{v})$$

This allows us to define a segmented gather operation for primitive types:

$$\begin{aligned} psegGather\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ s_{\mathrm{v}} = do \quad & s_{\mathrm{sd}} \leftarrow to\_flags\ s_{\mathrm{l}} \\ & s_{\mathrm{i}} \leftarrow iotas\ s_{\mathrm{sd}} \\ & s_{\mathrm{s}'} \leftarrow pdist\ s_{\mathrm{s}}\ s_{\mathrm{sd}} \\ & so\_gather\ s_{\mathrm{s}'}\ s_{\mathrm{i}}\ s_{\mathrm{v}} \end{aligned}$$

We can generalize segmented gather to any stream tree representing a value of concrete high-level type τ:

$$\begin{aligned} segGather_{\pi}\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ s &= psegGather\ s_0\ s \\ segGather_{(\tau_1,\ldots,\tau_k)}\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ (st_1,\ldots,st_k) &= do\ st_1' \leftarrow segGather_{\tau_1}\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ st_1 \\ &\quad \vdots \\ &\quad st_k' \leftarrow segGather_{\tau_k}\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ st_k \\ &\quad return\ (st_1',\ldots,st_k') \\ segGather_{[\tau]}\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ (st_0,(s_{\mathrm{s}},s_{\mathrm{l}})) &= do\ s_{\mathrm{s}}' \leftarrow psegGather\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ s_{\mathrm{s}} \\ &\quad s_{\mathrm{l}}' \leftarrow psegGather\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ s_{\mathrm{l}} \\ &\quad return\ (st_0,(s_{\mathrm{s}}',s_{\mathrm{l}}')) \end{aligned}$$

Notice how we use virtual segment description in the vector-type case and simply performs the segmented gather on the segment descriptor and leave the data streams untouched. We are now ready to give the translation of seq_τ:

$$\begin{aligned} \mathcal{S}\,[\![F]\!]_{seq_\tau}\ (st_0,(s_{\mathrm{s}},s_{\mathrm{l}}))\ s = do \quad & s_{\mathrm{sd}} \leftarrow to\_flags\ s_{\mathrm{l}} \\ & st_1 \leftarrow segGather_{\tau}\ s_{\mathrm{s}}\ s_{\mathrm{l}}\ st \\ & return\ (st_1, s_{\mathrm{sd}}) \end{aligned}$$

One optimization we perform in the actual implementation, is that we keep track of whether or not a virtual segment descriptor is actually a normal length-based segment descriptor. In that case, there is no need to perform the gather operation since the values are already guaranteed to be in sequence. We can then return the data streams st_0 as they are.

4.3.1 Optimization

In order to optimize SVCODE, we first perform static analysis. The properties we try to establish are: Stream length, stream element value (if it is constant) and, for segment flag streams, the number of segments (the number of T-flags). The analysis is based on constraint generation and resolution. It is similar to the shape analysis found in [RS], where we also generate constraints for element value. Additionally, for boolean streams, we generate

constraints for the number of T-flags. This is useful information because in segment descriptor stream, the number of T-flags corresponds directly to the number of segments. We can therefore track the resulting length of a segmented reduction and similar operations. Although it would be possible, we do not currently generate constraints for the uniformity of segments (i.e. whether or not all segments in a segment descriptor have the same length). Uniform segments are found in regular multi-dimensional arrays, which permits a whole class of useful optimizations. It could be an area of future investigation.

If a stream has a known length and value, we can simply replace the stream's definition by a special instruction: `constant` with the type

$$\texttt{constant}.\forall\pi.\pi \times \mathsf{Int} \to S\pi$$

The semantics are simply to replicate the first argument n times where n is the second argument. For example:

$$[\![\texttt{constant}]\!]\ \mathsf{T}\ 5 = \langle \mathsf{T}, \mathsf{T}, \mathsf{T}, \mathsf{T}, \mathsf{T} \rangle$$

For an example that uses this optimization, consider the pathological SNESL program:

$$\{3 + 4 : \_\ \textbf{in}\ \&100\}$$

It translates to the SVCODE below. For brevity we have temporarily introduced and `iotas` instruction that computes the iota values from the segment descriptor. We have also omitted a check of the argument to iotas.

```
s  := ctrl;
s0 := rep s 100;
s1 := to_flags s0;
s2 := iotas s1;
s3 := usum s1;
s4 := rep s3 3;
s5 := rep s3 4;
s6 := map_add s4 s5
```

Here, (s_6, s_1) is the handle to the result and can therefore not be eliminated. s_6 is the data stream and s_1 is the segment descriptor. Our size-analysis infers the following information where `l`=n means the length of the stream is n, `v`=a, means that all the values of the stream are equal to a and `t`=n means the stream is a boolean stream and it contains n T-flags. We annotate the instructions with comments containing the information we get from our analysis:

```
s  := ctrl;             -- l=1, v=*
s0 := rep s 100;        -- l=1, v=100
s1 := to_flags s0;      -- l=101, t=1
s2 := iotas s1;         -- l=100
s3 := usum s1;          -- l=100, v=*
s4 := rep s3 3;         -- l=100, v=3
s5 := rep s3 4;         -- l=100, v=4
s6 := map_add s4 s5     -- l=100, v=7
```

Note that we have used the information about the number of T flags (and the length) in s_1 to infer the length of s_2. This is because the constraint for the length of the output stream of usum is given by the equation $\mathtt{l}_{\text{out}} = \mathtt{l}_{\text{in}} - \mathtt{t}_{\text{in}}$. Also note that the analysis infers the value of the last stream. This is done by allowing the analysis to perform scalar computations for map instructions when all arguments have known constant values.

The streams where both the length and the value is known can be replaced by constant stream definitions. The program then turns into the equivalent program:

```
s  := constant *   1;
s0 := constant 100 1;
s1 := to_flags s0;
s2 := iotas s1;
s3 := constant * 100;
s4 := constant 3 100;
s5 := constant 4 100;
s6 := constant 7 100
```

Dead-code elimination can then reduce the program to:

```
s0 := constant 100 1;
s1 := to_flags s0;
s6 := constant 7 100
```

Similarly, we also use specialized instructions for some of the core instructions where one stream argument is specialized to a constant. For instance, if we make a slight change to our running example:

$$\{x + 4 : x \textbf{ in } \&100\},$$

then, after the optimization we have seen so far, we will end up with the program:

```
s0 := constant 100 1;
s1 := to_flags s0;      -- l=101, t=1
```

```
s2 := iotas s1;          -- l=100
s5 := constant 4 100;
s6 := map_add s2 s5    -- l=100
```

But, we know that s_5 is constantly 4, so we can replace `map_add` by a new specialized instruction which we call `map_add_const` with the type

$$\texttt{map\_add\_const} : S\mathsf{Int} \times \mathsf{Int} \to S\mathsf{Int}$$

Now, we can avoid generating the constant stream of 4's and the final program becomes:

```
s0 := constant 100 1;
s1 := to_flags s0;
s2 := iotas s1;
s6 := map_add_const s2 4
```

We also attempt to eliminate runtime checks by checking if the argument to `check` is constantly true. We may also find that the stream is constantly false, in which case we can reduce the whole program to an error message. All these optimizations can lead to dead-code elimination opportunities, which we then exploit.

If two instruction in an SVCODE program are the same and have identical arguments, we can almost always eliminate the last occurrence, and substitute any subsequent occurrences of the last stream identifier for the first. The exception is instructions that take no stream arguments (e.g. `constant` and `read_file`). Two such identical instructions have no interrelated constraint on the rate of which elements are produced. By replacing one with the other, we effectively constraint the consumers to consume the stream at the same rate. It might be the case that the consumers of the last instruction *must* operate at a different rate than the first. Hence, we cannot eliminate one and still ensure that streamability is preserved. If, on the other hand, the two identical instructions take a stream (the same stream) as argument, they are already constrained to operate at the same rate, and replacing one with the other will not affect the streamability of the network in any way.

The generated code for many SNESL operation can be improved if we know that the input does not contain any empty sequences. As we saw in the translation for reduce_R in the previous section, segmented sum, where none of the segments are empty, can be implemented as a segmented scan followed by a pack. If just one of the segment is empty, we must also perform an unpack to write the zero element in the result. In our current implementation, this can be hinted in the source code. However, in many cases, it should be possible to discover this information automatically in the static

analysis phase. That would involve generating constraints on the length of the segments in segment flag streams.

4.4 DPFlow: A Multicore Interpreter for SVCODE

The key piece that separates this paper from our previous paper [MF13] is a fully implemented multicore interpreter for SVCODE. We call this interpreter DPFlow; a contraction of data parallelism and dataflow. In essence, DPFlow is a low-level dataflow-based virtual machine, that executes SVCODE instructions using chunked streams and highly optimized data-parallel kernels written in C.

The kernels exploit both threaded execution and vector instructions, which we realize using pthreads and the automatic vectorization found in gcc 5.3. Crucially, the kernels are not generated per program, but are precompiled only once. This allows very fine-tuned optimization of each kernel, since we do not have to incorporate that into a code generator. For instance, we know exactly what kernels are automatically vectorized. When this is not the case, we may opt to vectorize the kernels by hand using Intel's SSE intrinsics (see Section 4.4.4).

4.4.1 Execution

DPFlow starts by setting up a network of stream transducers in memory based on a given SVCODE program. Each definition becomes one transducer. A transducer holds a fixed-sized buffer where the bytes of the output stream are stored, and a local state. The local state contains accumulators, an *end-of-stream* flag, a *write cursor* and multiple *read cursor*s; one for every input stream. After the initial phase, the transducers are fired repeatedly by the scheduler until all transducers have reached end-of-stream. Firing a transducer is the act of calling the corresponding kernel and updating all involved cursors.

A cursor is a relative offset in a buffer. A read cursor allows a reader to consume only part of the current bytes in the buffer and remember that. This is important to support different data rates. For example, the `map_char_to_int` transducer reads a stream of characters and converts them to a stream of (32-bit) integers. Since four bytes are output for every input byte, the transducer can consume at most a quarter of the chunk size bytes at a time. If the input buffer holds more than that, the transducer must perform a partial consumption of the input buffer. An example network and execution is illustrated in Figure 4.3.

In more detail, a transducer fires by first computing the number of available elements from the input buffers and the number of elements that there

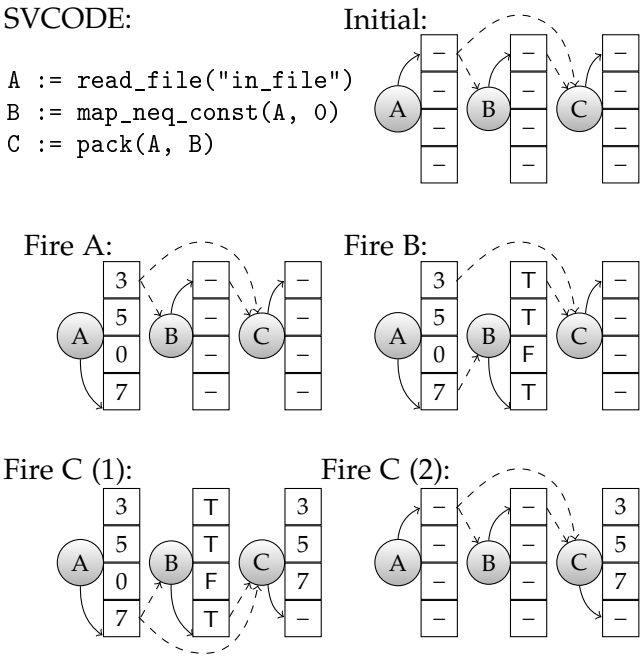


Figure 4.3: An illustration of a stream transducer network that removes all null characters (0) from a file named "in_file". The figure shows three stream transducers firing one at a time and the contents of their buffers (which are limited to a chunk size of 4). Write cursors are illustrated with solid arrows, and read cursors are illustrated with dashed arrows. Fire A: The reader transducer fills its buffer with bytes from the input file and moves its write cursor to the end of the buffer. Fire B: Each byte is compared to 0 in the transducer for map_neq_const. The read and the write cursors are moved. Fire C (1): The pack transducer filters the bytes from the read transducer using the booleans from the map transducer. Fire C (2): Since all read cursors on the buffers for A and B are at the end, the buffers are emptied by resetting the cursors and are hereby ready for the next chunk of data.

are room for in the output buffer. If the output buffer is uninitialized, it requests an empty buffer from the *nursery*. The transducer then calls a kernel function that performs the actual computation. The kernels functions are simple function on arrays. When the kernel is done, the transducer advances the read cursors and the write cursor, and updates the end-of-stream indicator. Elements that are located before the smallest read cursor of a buffer will never be used again. When a read cursor is advanced, a buffer may therefore become completely used, in which case that buffer is returned to the nursery, as is the case in step Fire C(2) of Figure 4.3.

4.4.2 Nursery

By using a nursery, we are able to reuse memory that is already in cache. Without a nursery, each transducer would have its own pre-allocated buffer, and all the buffers may not be able to fit in the cache at the same time. This means that each transducer may have to bring its input buffers into cache each time it fires, resulting in bad cache utilization.

With a nursery on the other hand, in the ideal scenario, the network consists of a long string of unary map transducers that fully consumes their input in each step. Here, only two buffers are needed: A read buffer and a write buffer. After each transducer is executed, the read buffer becomes the new write buffer and vice versa. In practice however, transducers of greater arity require more than one read buffer, and buffers are not fully consumed due to different data rates. In our experiments, the number of buffers required is approximately one third of the total number of transducers in the network.

4.4.3 Scheduling

An important part of executing a network is finding out what transducer to execute next. Scheduling is important for performance, because we want to minimize the number of active buffers and maximize the work in the kernels. We want to avoid executing a transducer with almost empty input buffers, as that would magnify the overhead of scheduling, and violate the high-level work/step cost model.

As an example, in Figure 4.3, notice that there are two strategies for further execution: Either fire the consumer(s) of (C) even though its buffer is less than full, or repeat the steps in the figure in an attempt to fill the buffer completely. The latter strategy requires more buffers to be active at the same time, reducing the effectiveness of the nursery, while the former strategy causes the consumers to fire with less-than-full input buffers. In this example, the buffer is almost full, but it could as well have been almost

empty. There is definitely opportunity for future work investigating the cost and benefit of different strategies. In this paper however, we focus on the simplest possible scheduling strategy called *loop scheduling*.

Loop scheduling executes the transducers in succession from the first to the last, starting over unless the network has reached completion. The transducers fire regardless of the fullness of the buffers. Consequently, a transducer may fire on almost empty input, which is sub-optimal in theory. In particular, we break the step part of the work/step cost model. However, in practice, loop scheduling performs well. This is due to a number of reasons. First, its simplicity makes the scheduling overhead small. Second, when executing with a chunk size much greater than the available parallel resources (as we do), a step in the cost model is actually many steps in the execution. Therefore, a non-full step likely saturates all available resources, and since we do obey the work part of the cost model, we achieve good performance anyway. Third, loop scheduling exhibits good nursery usage. Since consumers are usually located close to producers, and since we consume buffers as soon as possible, buffers are returned to the nursery and recycled relatively quickly, resulting in fewer total number of buffer allocations and, in turn, better cache utilization. We have not encountered any examples where loop scheduling performs significantly worse than a hard-coded, supposedly optimal, schedule.

4.4.4 SIMD Vectorization

Most kernels consist of a simple for-loop doing a single operation. One would therefore expect a high-quality C compiler to generate vectorized instructions for the pre-compiled kernel. Maps and reductions are indeed generally automatically vectorized, but this is not the case for scans and scan-like operations, such as packing. We have checked with the latest version (as of the time of writing) of gcc (5.3), clang (3.8) and icc (16.0), and none of them vectorize a simple scan with addition. We therefore hand-vectorize scan and segmented-scan kernels using Intel's SSE intrinsics.

An advantage of DPFlow, as opposed to writing a program in a traditional language like C, is that the programmer does not have to worry about whether or not the vectorizer succeeds. Adding a single scan-like dependency in a loop in C, will most likely cause the vectorization of the entire loop to fail. In effect, every operation in the loop becomes slow, not just the scanned operation. On top of that, if the compiler performs loop fusion, it may introduce scan-like dependency in an otherwise vectorizable loop, causing the vectorizer to fail in a non-obvious way.

In order to explore the possibility of improving the performance of scans, we have investigated different scan patterns: The Kogge-Stone scan cir-

Benchmark	Speedup	Elements scanned per vector addition
sse_blelloch_1x4	2.95	4 / 3
sse_blelloch_2x4	2.32	8 / 6
sse_blelloch_4x4	2.21	16 / 13
sse_kogge_stone_1x4	2.95	4 / 3
sse_kogge_stone_2x4	3.35	8 / 6
sse_kogge_stone_4x4	3.02	16 / 14
sse_sklansky_2x4	2.17	8 / 4

Figure 4.4: Vectorized scan speedups. The table show the speedups of a normal scan with floating point addition.

```
void scan_baseline(float *xs, float *ys, int n) {
  float acc = 0.0f;
  int i;
  for (i = 0; i < n; ++i) {
    acc = acc + xs[i];
    ys[i] = acc;
  }
}
```

Figure 4.5: Scan baseline implementation.

cuit [KS73], the Sklansky scan circuit [Skl60] and Blelloch's scan algorithm [Ble90a]. Our best implementation is the Kogge-Stone scan circuit [KS73], with which we have been able to boost the performance of float-add scan by a factor of 3. For the segmented version we have boosted the performance by a factor between 1.13 and 2.5 depending of the average length of the segments and whether or not they are regular.

SSE instructions pose a number of limitation on how a circuit can be implemented. The operators of the circuit must be arranged for small fixed-sized SIMD execution, which may require shuffling elements around. Additionally, there is only a limited number of vector registers, so our circuits must be small and used iteratively to scan a whole chunk. This means we have to account for an accumulator, which could simply be added to the first element at the start of each iteration. However, we have found that it is more efficient to broadcast the accumulator to a vector, and add it to all the elements in the end of each iteration.

We compare ourselves against baseline implementations in C, which are given in Figure 4.5.

Kogge Stone Scan

Kogge Stone scan is not work efficient. The asymptotic complexity is $O(n \log n)$. However, Kogge Stone circuits are a good fit for SIMD execution, since the operations are arranged in a very regular way, and thus require little shuffling.

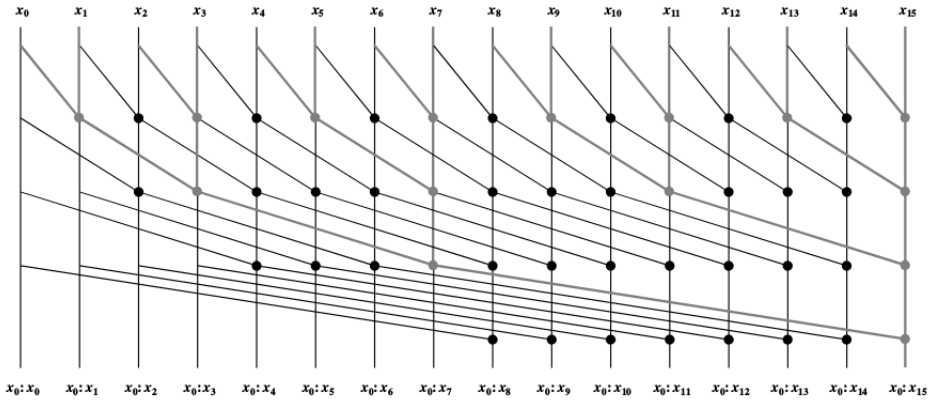
Figure 4.6 show the smallest Kogge Stone Scan circuit we have implemented. It scans one vector containing 4 floats in each iteration, which corresponds to the first 4 vertical lines of the illustration. The vector is added to a left shifted version of itself two times with an increasing shift. The first addition performs the operation $\lambda[a,b,c,d] \rightarrow [a, a+b, b+c, c+d]$, and the second addition performs the operation $\lambda[a,b,c,d] \rightarrow [a, b, a+c, a+d]$. Finally, the accumulator is added to every element and then updated. The result is that we need 3 vector additions to scan every 4 floats. We also implemented a version that scans two vectors in each iteration, and one that scans four vectors in each iteration. The benchmarks in Figure 4.4 are labeled with a post-fix to indicate the number of 4-element vectors scanned in each iteration. In the case of Kogge Stone: `1x4`, `2x4` and `4x4`. The best implementation of vectorized scan according to our experiment, is a version of this algorithm where we scan 2 vectors.

Blelloch's Scan

Blelloch's scan algorithm, is the only work-efficient algorithm (i.e. with asymptotic work of $O(n)$) we explore. However, asymptotic complexity is not a interesting here, because we operate on very small circuits. The property only manifests itself when we compare the number of elements scanned per vector addition for Blelloch's scan and Kogge Stone Scan in Figure 4.4. For 1 and 2 vectors per iteration, they require exactly the same number of vector additions. When using 4 vectors per iteration, Blelloch's scan requires one less than Kogge Stone scan. At this point however, the benchmarks already starts to degrade in performance – most likely due to vector register pressure.

Blelloch's algorithm is commonly used to implement scans on large arrays on GPUs, where work-efficiency is much more important. It is divided in an upsweep phase that recursively computes the sum of the array while keeping the partial sums, and a downsweep phase that computes the remaining elements with the previously computed partial sums.

Figure 4.7 shows the smallest circuit based on Blelloch's scan we have implemented. The upsweep phase is the upper gray operations in the circuit and the downsweep phase is the lower black operations. The implementation scans one vector containing 4 floats in each iteration, which corresponds

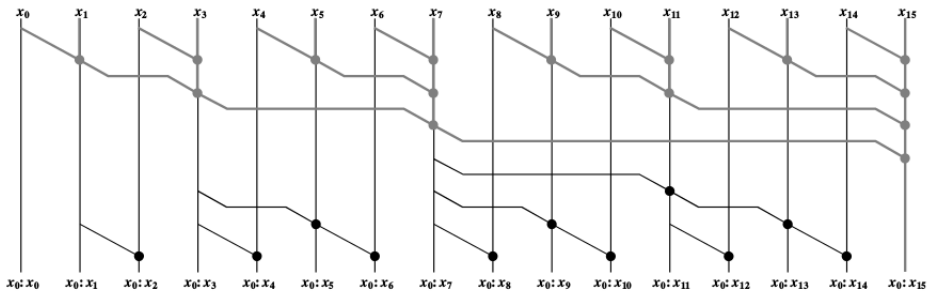



```
// Left shift a 4 element vector by i elements, shifting in zeroes.
inline __m128 lshift(__m128 v, int i) {
  return _mm_castsi128_ps(_mm_slli_si128(_mm_castps_si128(v), 4*i));
}

// 4 elements, 3 vector additions.
void sse_steele_1x4(float *xs, float *ys, int n) {
  __m128 v, acc;
  acc = _mm_set1_ps(0.0f);
  int i;
  for (i = 0; i+3 < n; i += 4) {
    v = _mm_load_ps(xs + i);
    v = _mm_add_ps(lshift(v, 1), v);
    v = _mm_add_ps(lshift(v, 2), v);
    v = _mm_add_ps(acc, v);
    _mm_store_ps(ys + i, v);

    // Broadcast the last element to the accumulator:
    acc = _mm_shuffle_ps(v, v, _MM_SHUFFLE(3, 3, 3, 3));
  }
}
```

Figure 4.6: Float addition scan based on the Kogge Stone circuit using Intel's SSE intrinsics. At the top there is an illustration of the circuit. The illustration is from [MG09]. Below the illustration, an implementation of the circuit is given.




```
// 4 elements, 3 vector additions
void sse_blelloch_1x4(float *xs, float*ys, int n) {
  __m128 v, acc128, zero;
  acc128 = _mm_set1_ps(0.0f);
  zero = _mm_setzero_ps();
  int i;
  for (i = 0; i+3 < n; i += 4) {
    v = _mm_load_ps(xs + i);
    v = _mm_add_ps(_mm_blend_ps(zero, _mm_moveldup_ps(v), 10), v);
    v = _mm_add_ps(_mm_shuffle_ps(zero,v, _MM_SHUFFLE(1,1,0,0)), v);
    v = _mm_add_ps(v, acc128);
    acc128 = _mm_shuffle_ps(v, v, _MM_SHUFFLE(3, 3, 3, 3));

    _mm_store_ps(ys + i, v);
  }
}
```

Figure 4.7: Float addition scan based on Blelloch's scan algorithm using Intel's SSE intrinsics. The circuit is a Brent-Kung circuit upon which Blelloch's scan algorithm is based. The circuit illustration is from [MG09].

to the first 4 vertical lines in the illustration.

In the implementation there are two steps. The first step performs the operation $\lambda[a,b,c,d] \to [a,a+b,c,c+d]$, and the second step performs the operation $\lambda[a,b,c,d] \to [a,b,b+c,b+d]$. To be more efficient, we the downsweep is done at the same time as the last step in the upsweep. The upsweep phase corresponds to the first step and $b+c$ in the second step. The downsweep phase is simply $b+d$ in the second step.

Like in the case of Kogge Stone scan, we have implemented instances of the circuits up to 4 vectors in each iteration. The execution times are generally worse than for Kogge Stone scan except in the simplest case where the circuits are almost identical. We believe this is due to an increased amount of shuffling.

Sklansky scan

Sklansky's circuit has the attractive property, from a vectorization perspective, that the number of additions in each step is constant. Choosing an appropriately-sized network therefore yields a situation, in which we must perform exactly 4 additions in each step. If we ignore the problem of arranging elements for vector execution, this circuit then maps directly to SIMD execution using 4-element vectors.

Figure 4.7 shows the only Sklansky scan circuit we have implemented. If all other operations were free and only the additions and memory operations had a time cost, this implementation would almost certainly be the fastest as it has the best vector additions to number of elements ratio; Only 4 vector additions are required to scan every 8 elements. Unfortunately, shuffling is not free, and the benchmark performs the worst of all the vectorized scans we have tested (though still more than twice as fast as the baseline).

Segmented Scan

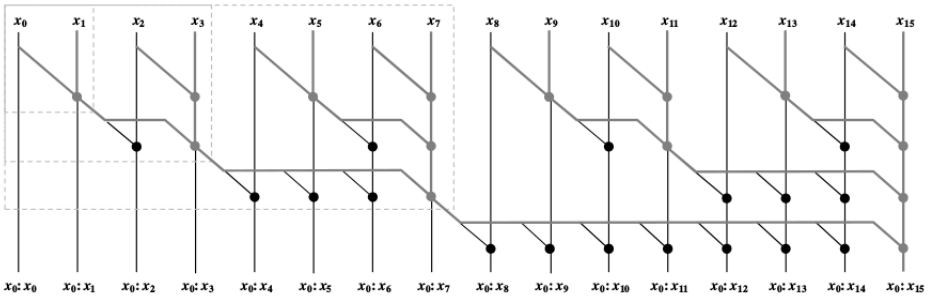
For the flag-segmented versions, we have only considered the Kogge Stone scan. It is implemented in the same way as non-segmented scan by using Blelloch's well-known operator transformation [Ble90a]:

$$(f_1, a) \oplus (f_2, b) = (f_1 \vee f_2, (\overline{f_2} \cdot a) + b),$$

where $\overline{f_2} \cdot a$ is the operation that resets the accumulator if $f_2 = \mathsf{T}$ and leaves a unchanged otherwise. We represent flags with bytes, so a vector of flags in SSE contains 16 flag values. We have to pay an additional cost of unpacking the flags into 4 element vectors in order to perform the transformed addition operation with vector instructions.

Our baseline implementations of segmented scan are given in Figure 4.9.

Figure 4.10 shows our speedups for different types of segments, long and short, regular and irregular segments. The vectorized versions are branchless, and do therefore not benefit from branch prediction. For short regular segments, we see that the branch predictor works perfectly. The branchful baseline therefore performs very well, and the speedup from vectorization is almost non-existent. We also see that the branchless baseline is quite slow in comparison. The same trend is noticeable – although to a lesser extend – for long segments, irrespective of whether or not the segments are regular. Here, the branch predictor correctly guesses when a segment does not end, but it almost surely fails to correctly guess when an end of segment occurs. The best scenario for our implementation is short irregular segments, where the branch predictor fails often. Here we achieve an even greater speedup than what was the case for non-segmented scan. In any case, since we out-




```
// 8 elements, 4 vector additions
void sse_sklansky_2x4(float *xs, float *ys, int n) {
  __m128 x1, x2, _x1, _x2, acc;
  acc = _mm_set1_ps(0.0f);
  int i;
  for (i = 0; i+7 < n; i += 8) {
    x1 = _mm_load_ps(xs + i);
    x2 = _mm_load_ps(xs + i + 4);

    _x1 = _mm_shuffle_ps(x1,x2, _MM_SHUFFLE(2,0,2,0));
    _x2 = _mm_shuffle_ps(x1,x2, _MM_SHUFFLE(3,1,3,1));
    _x2 = _mm_add_ps(_x1, _x2);

    _x1 = _mm_shuffle_ps(_x1, _x2, _MM_SHUFFLE(3,1,3,1));
    _x2 = _mm_shuffle_ps(_x2, _x2, _MM_SHUFFLE(2,0,2,0));
    _x1 = _mm_add_ps(_x2, _x1);

    x1 = _mm_shuffle_ps(x1, _x1, _MM_SHUFFLE(2,0,1,0));
    x1 = _mm_insert_ps(x1, _x2, _MM_SHUFFLE(0,1,0,0));
    x2 = _mm_shuffle_ps(x2, _x1, _MM_SHUFFLE(3,1,1,0));
    x2 = _mm_insert_ps(x2, _x2, _MM_SHUFFLE(1,1,0,0));

    x1 = _mm_add_ps(acc, x1);
    acc = _mm_shuffle_ps(x1, x1, _MM_SHUFFLE(3, 3, 3, 3));
    x2 = _mm_add_ps(acc, x2);
    acc = _mm_shuffle_ps(x2, x2, _MM_SHUFFLE(3, 3, 3, 3));

    _mm_store_ps(ys + i, x1);
    _mm_store_ps(ys + i + 4, x2);
  }
}
```

Figure 4.8: The implementation scans two vectors containing 4 floats each in each iteration using Intel's SSE intrinsics. The scan is based on the Sklansky scan circuit. The circuit illustration is from [MG09].

```
void seg_scan_baseline(bool *fs, float *xs, float *ys, int n) {
  float acc = 0.0f;
  int i;
  for (i = 0; i < n; ++i) {
    acc = fs[i] ? 0 : acc;
    acc = acc + xs[i];
    ys[i] = acc;
  }
}

void seg_scan_branchless(bool *fs, float *xs, float *ys, int n) {
  float acc = 0.0f;
  int i;
  union {
    uint32_t i;
    float f;
  } a;
  int f;
  a.f = acc;
  for (i = 0; i < n; ++i) {
    f = fs[i];
    a.i = (f - 1) & a.i;
    a.f = a.f + xs[i];
    ys[i] = a.f;
  }
}
```

Figure 4.9: Segmented scan baseline implementations.

perform the baseline on all benchmarks, it is safe to say that segmented scan is worth to vectorize.

4.4.5 Multi-Threading

One could hope that making DPFlow multi-threaded would require little more than placing OpenMP pragmas on top of each kernel loop. The reality is not so simple, however. It turns out that kernels are called very frequently and do not contain enough work, so the overhead of creating and joining threads in each kernel, as OpenMP does, becomes too expensive. We cannot simply increase the chunk size to accommodate the overhead, since that overflows the cache. Instead, we start a pool of worker threads at the beginning, and keep them alive for the entire duration of the execution. The worker threads busy-wait until the main thread passes work to them by using low-level synchronization primitives. When calling a kernel function, the main thread first places the kernel arguments in a globally accessible

(short)

Benchmark	Speedup
baseline_branchless	0.44
sse_kogge_stone_2x4	1.13
sse_kogge_stone_4x4	1.16

(short-ireg)

Benchmark	Speedup
baseline_branchless	1.29
sse_kogge_stone_2x4	3.27
sse_kogge_stone_4x4	3.48

(long)

Benchmark	Speedup
baseline_branchless	0.60
sse_kogge_stone_2x4	1.50
sse_kogge_stone_4x4	1.58

(long-ireg)

Benchmark	Speedup
baseline_branchless	0.59
sse_kogge_stone_2x4	1.49
sse_kogge_stone_4x4	1.57

Figure 4.10: Vectorized segmented scan speedups. The tables show the speedups of a flag-segmented scan with floating point addition for different types of segments. *(short)* and *(short-ireg)* benchmarks short segments of length 4. *(long)* and *(long-ireg)* benchmarks long segments of length 100. For *(short-ireg)* and *(long-ireg)* the segment lengths is an average value as the lengths are chosen at random to benchmark irregular segments.

array. It then signals the worker threads to call the same kernel function. Each thread runs the kernel on part of the index space, and signals the main thread, again using low-level synchronization, when they are done.

Multi-threaded scan and other scan-like kernels work in two passes. In the first pass, each thread performs a reduction on its part of the input array. The reduced results are then passed to the main thread, which scans them to compute a start accumulator for each thread. The second pass uses the hand-vectorized scan kernels. Each thread performs a scan using the given start accumulator.

One could expect a doubling of single-threaded execution time, because we effectively multiple the work by 2 by doing 2 passes. However, experiments show that the additional work is negligible. Partly because the first pass is a reduction which is relatively cheap as it does not require a memory write operation, and partly because it ensures that the relevant (to each thread) part of the input array is in cache for the second pass. Furthermore, the reduction pass is automatically vectorized by the C compiler and is generally more efficient then the scan pass.

4.5 Experiments

Text processing algorithms often contain sequential dependencies and irregular groupings (lines, words, etc.). This makes them difficult to parallelize

in languages with explicit task parallelism, and difficult to express in data-parallel languages without support for nested data parallelism. We evaluate four text-processing benchmarks that showcase SNESL's ability to express irregular nested data parallelism with scalable performance.

The benchmarks are selected from common Linux command line tools: word count, max line length, line reverse, and cut. We also revisit two basic benchmarks from our previous paper: logsum and logsumsum. Logsum sums the logarithms of i where i ranges from 1 to n, which a way to compute $\log(n!)$. Logsumsum computes the grand sum of multiple logsums.

To deal with file I/O, we introduce two functions in SNESL, read_file and write_file, that allows the programmer to work with files as sequences of characters. In order to avoid large overhead in measurements during the benchmarks, we use a RAM-based filesystem (tmpfs).

For the experiments, we use a 2.50 GHz Intel Xeon E5-2670 v2 with 10 cores, 256 KB L2 cache and 25 MB shared L3 cache. We perform the text processing benchmarks on a 3.5 GB file: the ASCII encoding of *Pride and Prejudice* by Jane Austen concatenated 5000 times, downloaded from Project Gutenberg (https://www.gutenberg.org/files/1342/1342.txt). Line reverse is from the standard package util-linux, version 2.27, and the other text utilities are from GNU Coreutils, version 8.25.

4.5.1 Logsum

We define logsum as $logsum(N) = \sum_{i=1}^{N} \log i$, computed in double precision. We compare ourselves to a straight-forward C implementation using a loop and an accumulator. We also test the program with OpenMP annotations on the loop.

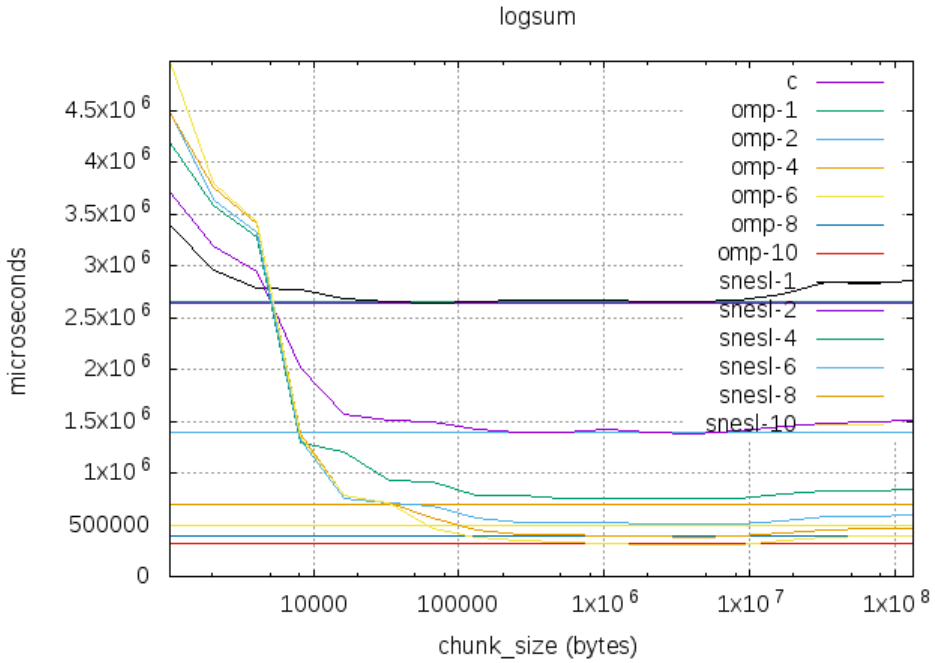
In SNESL, logsum can be expressed as follows:

```
fun logsum(N) =
  sum({log(real(i+1)) : i in &N})
```

(sum is an alias for reduce_plus.)

The benchmark numbers for $N = 10^8$ are given in Figure 4.11. They show that SNESL performs on par with C and OpenMP. Furthermore, the performance scales with the number of threads, matching the execution time of OpenMP.

From the graph, we also see that the choice of chunk size does not matter as long as it is between 100 KB and 10 MB. Below 100 KB, the overhead of scheduling and managing buffers and cursor becomes too significant. Above 10 MB we start to exceed the L3 cache. All in all, these results are very promising. It remains to be seen how we fare on more complicated examples.



Benchmark	Speedup	Chunk size	Milliseconds
c	1.00	N/A	2646
omp-1	0.99	N/A	2658
omp-2	1.90	N/A	1393
omp-4	3.76	N/A	701
omp-6	5.30	N/A	498
omp-8	6.70	N/A	394
omp-10	8.43	N/A	313
snesl-1	1.00	65536	2640
snesl-2	1.92	4194304	1379
snesl-4	3.62	4194304	750
snesl-6	5.23	4194304	510
snesl-8	7.00	4194304	383
snesl-10	8.70	4194304	308

Figure 4.11: Logsum execution times and speedups.

4.5.2 Logsumsum

Logsumsum computes the grand sum of multiple logsums. For some function $f : \mathbb{N} \to \mathbb{N}$, we define

$$logsumsum_f(M) = \sum_{N=1}^{M} \sum_{i=1}^{f(N)} \log(i)$$

This example is admittedly a bit contrived, but it serves as a simple example that highlights the challenges of irregular nested data parallelism. For the sake of simplicity, we ignore the possibility of memoization if f is non-injective, and we do not compute larger logsums from smaller, already computed, logsums.

Logsumsum is difficult to parallelize without flattening as the best parallelization strategy depends on f. If the image of f contains very large numbers, computing the logsum for those numbers will dominate the performance, and parallelization of the inner summation would be sufficient. If, on the other hand, f only produces small numbers, parallelizing the inner summation would expose very little parallelism. Here, it would be better to parallelize the outer summation, and let each thread compute M/p small logsums. However, the distribution of work may become skewed if the loop is parallelized naively. For example, the function $f(x) = 10x/M$ yields small skewed numbers.

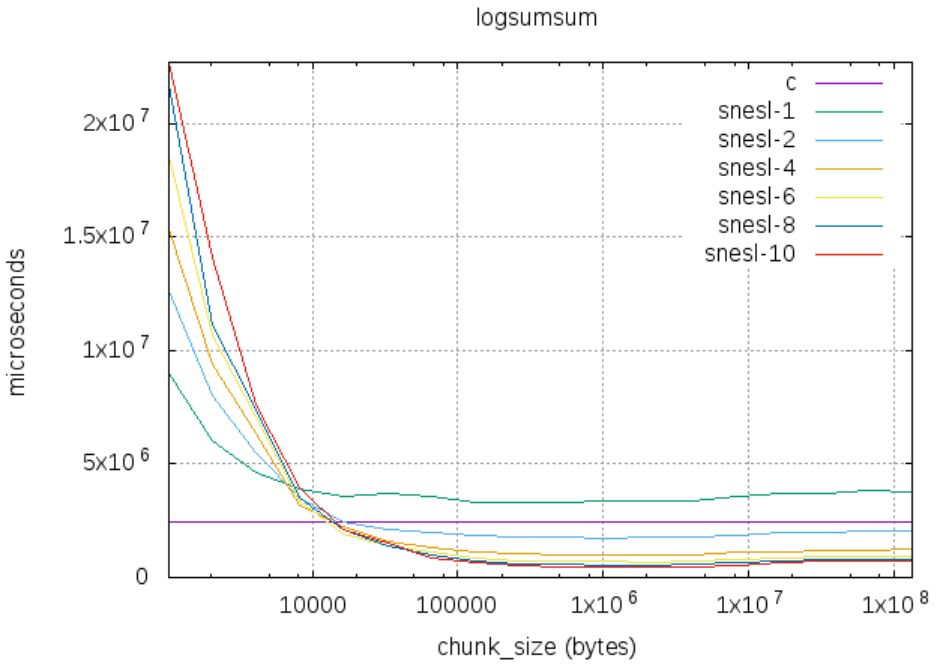
We express logsumsum in SNESL as:

```
fun logsumsum(M) =
  sum({logsum(10 * (N+1) / M) : N in &M})
```

The result of logsumsum for $M = 2 \times 10^7$ are given in Figure 4.12. As we see, SNESL performs approximately 20% slower than C using a single thread, and the performance scales decently. We were unable to obtain any speedup by placing OpenMP pragmas on either the inner loop, the outer loop or both. In light of this, the 20% performance drop seems a small price to pay.

4.5.3 Word Count

Word counting is a bit trickier than simply counting whitespace characters. Words, as defined by "wc -w", may be separated by more than one whitespace characters, and a word only counts towards the total word count if it contains at least one "printable" character. The numeric value of the whitespace characters are 9 to 13 and 32, and the printable characters are 32 to 126. The following is a SNESL program that produces exactly the same result as "wc -w" in the POSIX locale, even on binary files:



Benchmark	Speedup	Chunk size	Milliseconds
c	1.21	N/A	2461
snesl-1	1.00	1310720	2974
snesl-2	1.90	1310720	1562
snesl-4	3.40	1310720	874
snesl-6	4.73	1310720	628
snesl-8	6.09	1310720	488
snesl-10	7.33	1310720	405

Figure 4.12: Logsumsum execution times and speedups

Benchmark	Speedup	Chunk size	Milliseconds
wc -w	0.65	N/A	19760
snesl-1	1.00	163840	12770
snesl-2	2.06	163840	6214
snesl-4	3.54	327680	3607
snesl-6	4.73	655360	2700
snesl-8	5.84	655360	2188
snesl-10	6.74	1310720	1894

Figure 4.13: Word count speedups

```
-- Whitespace?
fun ws(c) = c == ' ' || c >= '\t' && c <= '\r'

-- Printable character?
fun pc(c) = c <= '~' && c > ' '

-- Is word printable?
fun pw(w) = reduce-or({pc(c): c in w})

fun wc(file) =
  let cs = read_file(file)
  in sum({ int(pw(w))
         : w in sep({(c, ws(c)): c in cs})})
```

The results are given in Figure 4.13. Surprisingly, we out-perform "wc -w". Even on a single thread, we out-perform the Linux tool by more than 50%. On top of that, we scale well with the number of threads, even though the problem is irregular.

4.5.4 Max Line Length

The problem of finding the maximum length of the lines in a file resembles word count. The challenge here, is that tab characters expand to count for 1–8 characters in the Linux tool we compare against ("wc -L"). To mimic this behavior, we separate each line by tabulation characters in SNESL and then compute the length of each "tab word" and round up to the nearest multiple of 8. Care must be taken to treat the last tab word in each line differently as it should not be expanded.

Rounding up can be done using bitwise operations:

```
fun round_to_8(n) = (n | 7) + 1
```

Benchmark	Speedup	Chunk size	Milliseconds
wc -L	0.74	N/A	19760
snesl-1	1.00	655360	14420
snesl-2	1.86	1310720	7764
snesl-4	3.17	1310720	4554
snesl-6	4.31	1310720	3343
snesl-8	5.24	1310720	2750
snesl-10	6.04	1310720	2385

Figure 4.14: Max line length speedups

The tricky part is to treat the last tab word differently. We do this by using the builtin `tail_flags` operation, which gives us a boolean for each word, that we can use to distinguish the last tab word. The following function computes the length of a line in SNESL:

```
fun line_len(l) =
  let ws = sep({(c, c == '\t') : c in l})
  in
    sum({ let n = #w - 1
          in (is_last ? n : round_to_8(n))
        : w in ws;
          is_last in tail_flags(ws)
        })
```

We are now ready to compute the maximum line length of a file:

```
fun lines(cs) =
  sep({(c, c == '\n') : c in cs})

fun max_ll(file) =
  let cs = read_file(file)
  in maximum({line_len(l) : l in lines(cs)})
```

The results are given in Figure 4.14. Once again, we are faster than the Linux tool in single-threaded performance, and we scale well with the number of threads.

4.5.5 Line Reverse

Reversing each line in a file is accomplished using the Linux tool "rev". The equivalent program in SNESL is interesting, because it is impossible to ex-

Benchmark	Speedup	Chunk size	Milliseconds
rev	1.88	N/A	21942
snesl-1	1.00	327680	40864
snesl-2	1.86	655360	22694
snesl-4	3.02	655360	13534
snesl-6	4.01	1310720	10188
snesl-8	4.90	1310720	8343
snesl-10	5.61	1310720	7286

Figure 4.15: Line reverse speedups

press without using vectors. Reversing a line requires unbounded buffering, which we express in SNESL as a sequence of vectors.

We first define a function to reverse a line (a sequence of characters). The function assumes that the line ends in a newline character. That character is kept in the end as we do not want to move newline characters to the beginning of each line.

```
fun rev_line(w) =
  let v = tab(w);
      n = #v-1
  in {v!(i == n ? i : n-1-i) : i in &n}
```

We then reverse each line, like so:

```
fun rev(file_in, file_out) =
  let cs = read_file(file_in);
      res = {rev_line(l) : l in lines(cs)}
  in write_file(file_out, concat(res))
```

The results are given in Figure 4.15. Here, our single-threaded performance is not particularly impressive, being nearly half as fast as "rev". However, once we increase the number of threads, we easily out-perform the Linux tool, which does not have multi-threaded support.

4.5.6 Cut

The cut benchmark explores filtering. Here, we select the *i*th column of a space delimited file (i.e., `cut -d" " -f`*i*). Cut is interesting because it requires random-access on a vector of words, i.e. a vector of vector of characters. Cut still works even when a line does not have enough columns. In this case, there are two possible outcomes. If there is only one column (i.e. the line contains no space characters), then the whole line is returned. Oth-

erwise, the empty string is returned. The benchmark therefore also explores true conditionals where we do not evaluate the false branch (as opposed to sel (? :)), which may cause out of bounds error in this case.

We first define two functions for converting lines to tabulated rows and indexing fields of a row:

```
fun row(l) =
  tab({ tab(f)
      : f in sep({(x, x == ' ') : x in l})})

fun index_field(r, ix) =
  let n = #r;
      i = n == 1 ? 0 : ix
  in if i < n
     then r ! i
     else [' ']
```

Then, we bring the functions together to define "cut" in SNESL:

```
fun cut(in, out, ix) =
  let cs = read_file(in);
      res =
        { let f = index_field(row(l), ix)
          in { (c == ' ' ? '\n' : c)
               : c in seq(f)}
        : l in lines(cs)
        }
  in write_file(out, concat(res))
```

The results with $i = 2$ can be seen in Figure 4.16. The performance of SNESL is, again, not as impressive as our previous experiments. We only beat the Linux tool if we use 4 threads or more. A part of the reason is conditionals with non-scalar type. The if-then-else becomes more than 60 instructions in SVCODE that accounts for approximately 20% of the execution time.

4.6 Conclusions and Future Work

We have shown that a chunked-dataflow execution model for streaming nested data parallelism – despite a number of apparent inefficiencies due to scheduling and data-transfer overheads – actually gives single-core performance similar to sequential C code on a selection of simple text-processing tasks. We attribute this parity mainly to the much better utilization of SIMD instructions by hand-written kernels, than what is achieved by current

Benchmark	Speedup	Chunk size	Milliseconds
cut -d" " -f2	1.98	N/A	16192
snesl-1	1.00	655360	32120
snesl-2	1.78	655360	18012
snesl-4	2.92	655360	11003
snesl-6	3.79	655360	8480
snesl-8	4.42	655360	7262
snesl-10	4.95	655360	6491

Figure 4.16: Cut speedups

industrial-strength compilers. Crucially, however, the dataflow code is also directly suitable for further speedup by subdividing the per-chunk work equally among multiple cores. Taking into account that parallel algorithms for scans at least double the number of fundamental operations performed, in addition to increased bookkeeping overheads, we observe speedups on moderate numbers of cores that are probably close to what can be reasonably achieved.

An immediate area for future work is static checking of schedulability, to outlaw inherently non-streamable computations such as

$$\mathsf{let}\ m = \mathsf{reduce}_{\mathsf{min}}(s)\ \mathsf{in}\ \{x - m : x\ \mathsf{in}\ s\}$$

(where we can only know m after having traversed all of s, and so cannot use it to process s from the beginning). Intuitively, a SNESL program should only be considered correct if it cannot deadlock when executed with a chunk size of one element, corresponding to a purely sequential program with coroutines. While simple sufficient criteria for this property exist (or can be derived from classical work on synchronous dataflow), the abstraction of *nested* sequences brings some challenges in turning them into a compositional (and programmer-comprehensible) analysis or type system. However, we expect to establish formally that if a system does not deadlock with a chunk size of one element, it will not deadlock for any larger (not necessarily uniform throughout the network) size either; thus, stress-testing the network with minimal buffer sizes should still uncover most concurrency bugs.

Finally, there are still ample opportunities for performance tuning. Though the compiler already performs a simple shape analysis before the flattening transformation, and a range of peephole optimizations on the generated SV-CODE, we expect that further improvements such as selective transducer fusion (especially map–map, which shouldn't interfere significantly with vectorization, analysis-guided specialized representations of data and/or

descriptor streams (e.g., run-length compression), and similar tweaks would improve both single-core performance and scalability even further.

Chapter 5

Towards a Formal Validation of the SNESL Cost Model

It is one thing to have a nice theoretical characterization of time and space usage and another to know whether it actually works in practice or not. We deemed it more important to have early indications for the latter. Consequently, the majority of the work undertaken in this dissertation has been devoted to implementation and experimentation, and we have not pursued a theoretical result. Nevertheless, we have reflected on in what would be necessary in order to prove a theorem that formally validates our cost model.

5.1 Translation soundness

The first result we would like to have is semantic preservation by the translation defined in Section 2.3.2. Recall the "is represented as" relation ($\triangleleft$) that relates the low-level representation back to frontend values. Using this relation, we can state a theorem that validates that the translation is sound with respect to the computed values:

Theorem: Translation Soundness
For all $[] \vdash e : \sigma$, if

$$[] \vdash e \Downarrow v \,\$\, \omega$$

for some v and ω, then

$$v \triangleleft [\![e]\!] [] \ \langle * \rangle.$$

The proof of translation soundness rests on a lemma that does not assume empty contexts and does not necessarily give a translation using the unit control stream $\langle * \rangle$.

Before we can state the lemma, we need some definitions. Define the "are represented as" relation:

$$(v_1, ..., v_l) \lhd^l st \Leftrightarrow \{v_1, ..., v_l\} \lhd (st, \langle \overbrace{\mathsf{F}, ..., \mathsf{F}}^{l}, \mathsf{T} \rangle) \quad l{>}0$$

This relation relates multiple high-level values of the same type to a single low-level representation. Some examples:

$$\begin{array}{rcl} (1,2,3) & \lhd^3 & \langle 1,2,3 \rangle \\ (\{1,2\},\{\},\{3\}) & \lhd^3 & (\langle 1,2,3 \rangle, \langle \mathsf{T},\mathsf{T},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{T} \rangle) \\ ((1,\mathsf{T}),(2,\mathsf{T}),(3,\mathsf{F}),(4,\mathsf{F})) & \lhd^4 & (\langle 1,2,3,4 \rangle, \langle \mathsf{T},\mathsf{T},\mathsf{F},\mathsf{F} \rangle) \end{array}$$

As can be seen, the relation essentially looks under the top-most segment descriptor and allows us to relate low-level values with more than one T in the top-most segment descriptor. Note that for $l = 1$, $(\lhd)$ is the same as $(\lhd^1)$ as we have

$$(v_1) \lhd^1 s \Leftrightarrow \{v_1\} \lhd (s, \langle \mathsf{F}, \mathsf{T} \rangle) \Leftrightarrow v_1 \lhd s.$$

We then lift this definition to value environments. For high-level environments $\rho_1, ..., \rho_l$ and low-level environment ζ, define

$$(\rho_1, ..., \rho_l) \lhd^l \zeta$$

if and only if, for all $i \in 1..l$, $dom(\rho_i) \supseteq dom(\zeta)$ and for all $x \in dom(\zeta)$, $(\rho_1(x), ..., \rho_l(x)) \lhd^l \zeta(x)$.

Translation Soundness Lemma For all $\Gamma \vdash e : \sigma$, $l > 0$, $(\rho_1, ..., \rho_l) \lhd^l \zeta$ (all ρ_i well-typed in Γ). If

$$\rho_1 \vdash e \Downarrow v_1 \,\$\, \omega_1 \qquad \cdots \qquad \rho_l \vdash e \Downarrow v_l \,\$\, \omega_l$$

for some $v_1, ..., v_l$ and $\omega_1, ..., \omega_l$, then

$$(v_1, ..., v_l) \lhd^l [\![e]\!]\zeta \; \langle \overbrace{*, ..., *}^{l} \rangle.$$

The proof is by induction on the syntax of e. The most important case is apply-to-each, which we will demonstrate here. We assume a weakening lemma in the type system that allows us to add assumptions to the typing context.

Case $e = \{e_0 : x \textbf{ in } x_0 \textbf{ using } x_1, ..., x_k\}$. Since there is only one typing rule for apply-to-each, the typing derivation must be:

$$\frac{\Gamma(x_0) = \{\sigma_0\} \quad \left(\Gamma(x_i) = \tau_i\right)_{i=1}^{k} \quad [x \mapsto \sigma_0, x_1 \mapsto \tau_1, ..., x_k \mapsto \tau_k] \vdash e_0 :: \sigma}{\Gamma \vdash \{e_0 : x \textbf{ in } x_0 \textbf{ using } x_1, ..., x_k\} :: \{\sigma\}} \; (k{\geq}0)$$

There is also only one big-step evaluation rule that applies, so each of the big-step derivations must be:

$$\frac{\rho_i(x_0) = \{v_{i,1}, ..., v_{i,l'_i}\} \quad \left(\rho_i[x \mapsto v_{i,j}] \vdash e_0 \Downarrow v'_{i,j} \,\$\, \omega_{i,j}\right)_{j=1}^{l'_i}}{\rho_i \vdash \{e_0 : x \textbf{ in } x_0 \textbf{ using } x_1^{\tau_1}, ..., x_k^{\tau_k}\} \Downarrow \{v'_{i,1}, ..., v'_{i,l'_i}\} \,\$\, \omega_i}$$

It remains to show that

$$(\{v'_{1,1}, ..., v'_{1,l'_1}\}, ..., \{v'_{l,1}, ..., v'_{l,l'_l}\}) \lhd^l [\![\{e_0 : x \textbf{ in } x_0 \textbf{ using } x_1, ..., x_k\}]\!]\zeta \ \langle \overbrace{*, ..., *}^{l} \rangle$$

From the definition of the translation we have

$$[\![\{e_0 : x \textbf{ in } x_0 \textbf{ using } x_1^{\tau_1}, ..., x_k^{\tau_k}\}]\!]\zeta \ s =$$

$$\begin{array}{l} let \ (t, s') = \zeta \, x_0 \ in \\ \quad ([\![e_0]\!][x \mapsto t, (x_i \mapsto dist_{\tau_i} \ (\zeta x_i) \ s')_{i=1}^k] \ (usum \ s'), s') \end{array}$$

From $\rho_i(x_0) = \{v_{i,1}, ..., v_{i,l'_i}\}$ for $i = 1..l$ and from $(\rho_1, ..., \rho_l) \lhd^l \zeta$, we have that

$$(\{v_{1,1}, ..., v_{1,l'_1}\}, ..., \{v_{l,1}, ..., v_{l,l'_l}\}) \lhd^l \zeta \, x_0$$

Furthermore, from the definition of representation of sequences and $(t, s') = \zeta \, x_0$, we have

$$(\overbrace{v_{1,1}, ..., v_{1,l'_1}}, ..., \overbrace{v_{l,1}, ..., v_{l,l'_l}}) \lhd^{l''} t,$$

where $l'' = \sum_{i=1}^{l} l'_i$ and

$$s' = \langle \overbrace{\mathsf{F}, ..., \mathsf{F}}^{l'_1}, \mathsf{T}, ..., \overbrace{\mathsf{F}, ..., \mathsf{F}}^{l'_l}, \mathsf{T} \rangle.$$

Now, there are two sub-cases: $l'' = 0$ and $l'' > 0$. The first case $l'' = 0$ is a base case that does not use the inductive hypothesis. We will proceed to demonstrate the case where $l'' > 0$.

For the using variables x_j in $x_1, ..., x_k$, from $(\rho_1(x_j), ..., \rho_l(x_j)) \lhd^l \zeta x_j$, we have $(v''_{1,j}, ..., v''_{l,j}) \lhd^l \zeta x_j$ for some $v''_{i,j}$. Since s' defines l segments, we get (from the definition of *dist*, and by pulling the result back to high-level values) that

$$(\overbrace{v''_{1,j}, ..., v''_{1,j}}^{l'_1}, ..., \overbrace{v''_{l,j}, ..., v''_{l,j}}^{l'_l}) \lhd^{l''} dist_{\tau_j} \ (\zeta x_j) \ s'.$$

By combining this result with $(v_{1,1}, ..., v_{1,l'_1}, ..., v_{l,1}, ..., v_{l,l'_l}) \lhd^{l''} t$, we get that

$$(\overbrace{\rho_1[x \mapsto v_{1,1}], ..., \rho_1[x \mapsto v_{1,l'_1}]} ..., \overbrace{\rho_l[x \mapsto v_{l,1}], ..., \rho_l[x \mapsto v_{l,l'_l}]}) \lhd^{l''}$$

$$[x \mapsto t, (x_i \mapsto dist_{\tau_i}\ (\zeta x_i)\ s')_{i=1}^{k}]$$

By induction on e_0 with this and with $l'' > 0$ and with all the big-step evaluations of e_0 and the typing derivation of e_0 (after weakening the context), we get

$$(\overbrace{v'_{1,1}, ..., v'_{1,l'_1}}, ..., \overbrace{v'_{l,1}, ..., v'_{l,l'_l}}) \lhd^{l''} [\![e_0]\!][x \mapsto t, (x_i \mapsto dist_{\tau_i}\ (\zeta x_i)\ s')_{i=1}^{k}]\ \langle \overbrace{*, ..., *}^{l''} \rangle$$

But since $usum\ s' = \langle \overbrace{*, ..., *}^{l''} \rangle$, the right-hand side is really

$$[\![e_0]\!][x \mapsto t, (x_i \mapsto dist_{\tau_i}\ (\zeta x_i)\ s')_{i=1}^{k}]\ (usum\ s'),$$

which is the definition of the whole translation without the segment descriptor s'. We can attach s', and since the segments add up to the high-level lengths $l'_1, ..., l'_l$, the representation relation gives us the desired result:

$$(\{v'_{1,1}, ..., v'_{1,l'_1}\}, ..., \{v'_{l,1}, ..., v'_{l,l'_l}\}) \lhd^{l} [\![\{e_0 : x\ \mathbf{in}\ x_0\ \mathbf{using}\ x_1, ..., x_k\}]\!]\zeta\ \langle \overbrace{*, ..., *}^{l} \rangle.$$

□

5.2 Translation completeness

The other direction of the translation soundness theorem also holds as we do not turn a program that results in an error in the big-step semantics into a error-free program that gives a result in the translated semantics.

Theorem: Translation Completeness
For all $[] \vdash e : \sigma$, if

$$[\![e]\!][]\ \langle * \rangle = st$$

for some stream value st, then

$$[] \vdash e \Downarrow v\ \$\ \omega$$

for some v and ω and $v \lhd st$.

Since we do not have recursion, the only things that can go wrong in the big-step semantics are runtime errors. The only operations that can cause runtime errors are:

- $!_\tau$: Out-of-bounds error when indexing.
- $zip^k_{\sigma_1,...,\sigma_k}$: Zipping two sequences of different lengths.

- &: Iota on negative arguments.
- the_σ: If the argument sequence does not have length 1.
- $flagpart_\sigma$: Flag-partitioning a sequence where the number of T-flags does not match the number of elements in the sequence.
- Many scalar operations cause an error on certain inputs such as $log(0)$, $x/0$ and $sqrt(-1)$.

We ensure that the generated code inserts checks for all these operations, and thus, translation completeness holds.

5.3 Space Cost Model

The ultimate goal of a formal treatment is to prove the correctness of our space cost model. The time cost model is more or less identical to the work–step cost model for NESL, and although it would be necessary to validate the time cost model as well in an exhaustive validation, we focus on the space cost model here, since that is the novel part. We refer to our empirical validation for an assertion of the correctness of the time cost model.

The denotational semantics of translated SNESL expressions defined in Chapter 2 does not say anything about buffers and the maximum size of buffers. Consequently, translation soundness and completeness has essentially only been proved for the most lenient evaluation strategy possible: fully eager semantics. In order to say anything about chunked evaluation, we must define a notion of buffers and the chunk size formally. For that, we will use the translation given in Chapter 4, Figure 4.2, and define an operational semantics for SVCODE.

5.3.1 Operational Semantics for SVCODE

Suppose we have two operational semantics for SVCODE. One that is fully eager (like the denotational translation) and one that is fully sequential. To describe them, we will use two to-be-defined judgments:

$$c \vdash \Phi_0^{\mathrm{p}} \Rightarrow \Phi_1^{\mathrm{p}}$$

$$c \vdash \Phi_0^{\mathrm{s}} \rightarrow \Phi_1^{\mathrm{s}}$$

Here c ranges over SVCODE programs, $\Rightarrow$ is a single step in the parallel operational semantics, and $\rightarrow$ is a single step in the sequential operational semantics. $\Phi^{[\mathrm{p}|\mathrm{s}]}$ ranges over the runtime state of the dataflow DAG. It is a mapping from stream identifiers to stream values. In the parallel semantics, Φ^{p} maps stream identifiers to fully manifest streams. In the sequential

semantics, Φ^{s} maps bounded stream identifiers to single-valued cells that may only hold one value at a time, and unbounded stream identifiers to buffers whose sizes are determined by the stepping rules of the judgment. They should reflect the intended semantics of `segtab` where one argument defines the segments on the buffer, and the semantics must allow one full segment to occupy the buffer at a time.

The induced multi-step judgments $\Rightarrow^*$ and $\rightarrow^*$ then tracks the additional space (= the number of primitive values) stored in $\Phi^{[\mathrm{p}|\mathrm{s}]}$ in each step. We annotate multi-step sequences with this information as a postfix ($\$\ S$):

$$c \vdash \Phi_0^{\mathrm{p}} \Rightarrow^* \Phi_1^{\mathrm{p}} \,\$\, S$$

$$c \vdash \Phi_0^{\mathrm{s}} \rightarrow^* \Phi_1^{\mathrm{s}} \,\$\, S$$

Since the parallel semantics is essentially the same as the denotational semantics given in Chapter 2, translation soundness and completeness holds for the parallel semantics.

The two semantics should produce the same observable result in the final state. Note that, in the parallel semantics, the instructions are evaluated one at a time, from top to bottom, possibly by keeping a program counter in Φ^{p}, whereas, in the sequential semantics, any instruction that has sufficient input data and room in its output cell may step. To allow comparing the results, we allow observation on Φ^{s} that can to look at past values. Thus $\Phi^{\mathrm{s}}_{\mathrm{obs}}$ is a mapping from stream identifiers to all values of that stream seen so far. We can then state the desired partial equivalence of the two semantics as:

$$\text{If } c \vdash \mathit{init}(c) \rightarrow^* \Omega^{\mathrm{s}} \,\$\, S, \text{ and}$$

$$c \vdash \mathit{init}(c) \Rightarrow^* \Omega^{\mathrm{p}} \,\$\, S \text{ then } \Omega^{\mathrm{s}}_{\mathrm{obs}} = \Omega^{\mathrm{p}}.$$

Where $\mathit{init}(c)$ is the initial runtime state for the program c and Ω ranges over runtime states that are in a completed state. That is, if the sequential semantics terminate and the parallel semantics also terminate then the resulting states are equal under full observation of the sequential state. Starting from a high-level SNESL expression e, we know from translation soundness when $\Rightarrow^*$ terminates. Whether or not $\rightarrow^*$ terminates is less certain and requires a schedulability analysis. Strictly speaking, for the purpose of proving the upcoming desired theorem about space cost preservation, one could cheat by defining the sequential semantics so that no program ever terminates. However, our implementation certainly suggests that it is possible to define the sequential semantics so that almost all programs that terminate in the parallel semantics, also terminate in the sequential semantics, and we expect an actual definition to behave similarly.

5.3.2 Space Cost Preservation

Desired Theorem: Space Preservation For all $[] \vdash e : \sigma$, $l > 0$, let

$$(st, c, \_) = \mathcal{S}\,[\![e]\!]\;\;[]\;\mathtt{s}_0\;1.$$

If $[] \vdash e \Downarrow v \,\$\, (W, D; M, N; L)$ then 2 things:

1. Parallel space preservation: If

$$\mathtt{s}_0 := \mathtt{ctrl}; c \vdash \mathit{init}(\mathtt{s}_0 := \mathtt{ctrl}; c) \Rightarrow^* \Omega^{\mathrm{p}} \,\$\, S$$

   then $S = O(1 + L)$.

2. Sequential space preservation: If

$$\mathtt{s}_0 := \mathtt{ctrl}; c \vdash \mathit{init}(\mathtt{s}_0 := \mathtt{ctrl}; c) \rightarrow^* \Omega^{\mathrm{s}} \,\$\, S$$

   then $S = O(1 + M + N)$.

In the translation, the $\mathtt{s}_0$ argument is the control stream, and 1 is the next fresh identifier. If we can prove this desired theorem, we can combine the two results and extrapolate to P processors by scaling the number of active buffers (recorded in M) by P. This is sufficient because the unbounded buffers are also accounted for in M and N only refers to the additional space requirements for these buffers. In other words, scaling M scales all active buffers. If, for any buffer, we overflow the parallel size by scaling with P, we assume that the operational semantics does not allocate more than the parallel size. We can then derive the expected cost on P processors as

$$S = O(\min(P \cdot M + N, L)).$$

The lemma that the theorem rests upon is quite complicated. The following is a sketch of how it could look:

Desired Lemma: Space Preservation Lemma For all $\Gamma \vdash e : \sigma$, $l > 0$, c_0, ζ, ρ_1,...,ρ_l, i, s_{ctrl}, such that ρ_1,...,ρ_l are well-typed in Γ and ζ maps the variables of Γ to stream identifiers defined in c_0, and $i > j$ for all s_j defined in c_0, and s_{ctrl} defines a stream of units in c of length l, let

$$(st, c, \_) = \mathcal{S}\,[\![e]\!]\ \zeta\ s_{\text{ctrl}}\ i.$$

If $\rho_1 \vdash e \Downarrow v_1\ \$\ (W_1, D_1; M_1, N_1; L_1) \quad \cdots \quad \rho_l \vdash e \Downarrow v_l\ \$\ (W_l, D_l; M_l, N_l; L_l)$ then 2 things:

1. Parallel space preservation: If

   $$c_0 \vdash init(c_0) \Rightarrow^* \Omega_1^{\text{p}}\ \$\ S'$$

   and

   $$c_0; c \vdash init(c_0; c) \Rightarrow^* \Omega_2^{\text{p}}\ \$\ S$$

   then $S - S' = O(\Sigma_{i=1}^{l} L_i)$.

2. Sequential space preservation: If

   $$c_0 \vdash init(c_0) \rightarrow^* \Omega_1^{\text{s}}\ \$\ S'$$

   and

   $$c_0; c \vdash init(c_0; c) \rightarrow^* \Omega_2^{\text{s}}\ \$\ S$$

   then $S - S' = O(\max_{i=1}^{l} M_i + \max_{i=1}^{l} N_i)$.

The lemma states how much *additional* space is required in the parallel and sequential evaluation of a translated program in a not necessarily empty context. c_0 represents any SVCODE program that computes the contexts ρ_1,..., ρ_l. We then consider an extended program $c_0; c$ where c is the instructions generated from e. If c_0 terminates using S' space and the extended program terminates using S space, then the difference is at most the amount of additional space required by the instructions in c. The additional space in the two operational semantics is then related to the high-level cost model. Just like the lemma for translation soundness, this lemma also relates multiple high-level evaluations with a single low-level evaluations.

We will now present a sketch of the required proof. The proof proceeds by induction on the syntax of e:

- Case $e = a$: There is also only one big-step evaluation rule that applies, so each of the big-step derivations must be:

$$\frac{}{\rho_i \vdash a \Downarrow a \,\$\, (1,1;1,0;1)}$$

We start by showing parallel space preservation (1.). We have

$$\mathcal{S}\,[\![e]\!]\;\zeta\; s_{\text{ctrl}}\; n = (n, n := \texttt{rep}\; s_{\text{ctrl}}\; a, n+1)$$

Assume

$$c_0 \vdash \mathit{init}(c_0) \Rightarrow^* \Omega_1^{\text{p}} \,\$\, S'$$

$$c_0; n := \texttt{rep}\; s_{\text{ctrl}}\; a \vdash \mathit{init}(c_0; n := \texttt{rep}\; s_{\text{ctrl}}\; a) \Rightarrow^* \Omega_2^{\text{p}} \,\$\, S$$

then we must show

$$S - S' = O(\Sigma_{i=1}^{l} 1) = O(l).$$

Since s_{ctrl} defines a stream of length l by assumption, $\texttt{rep}\; s_{\text{ctrl}}\; a$ defines a stream of length l as well. In the parallel semantics, this should require exactly l space to evaluate, and so adding this instruction can only introduce l more space, so $S - S' = O(l)$ as required.

The case of sequential space is analogous, only here, we must show $S - S' = O(1)$. Since `rep` uses bounded buffers, the sequential semantics cannot define more than 1 cell in Ω. Clearly, this requires 1 space and we are done.

- Variables, tuples and projection: These operations generate no additional instructions. Therefore, $c_0; c = c_0$ and assuming determinism with respect to space $S = S'$. But then $S - S' = 0$ which trivially satisfies what we must show.

- Case $e = \textbf{let}\; x = e_0\; \textbf{in}\; e_1$: We must have:

$$\frac{\begin{array}{l}\rho_i \vdash e_0 \Downarrow v_{0,i} \,\$\, (W_{0,i}, D_{0,i}; M_{0,i}, N_{0,i}; L_{0,i}) \\ \rho_i[x \mapsto v_{0,i}] \vdash e_1 \Downarrow v_{1,i} \,\$\, (W_{1,i}, D_{1,i}; M_{1,i}, N_{1,i}; L_{1,i})\end{array}}{\begin{array}{l}\rho_i \vdash \textbf{let}\; x = e_0\; \textbf{in}\; e_1 \Downarrow v_{1,i} \,\$ \\ \quad (W_{0,i} + W_{1,i}, D_{0,i} + D_{1,i}; M_{0,i} + M_{1,i}, N_{0,i} + N_{1,i}; \max(L_{0,i}, L_{1,i} + \mathcal{P}\|v_{0,i}\|))\end{array}}$$

By inspecting the translation rule for let-bindings. We see that the generated SVCODE takes the form

$$c_0; (c_0'; c_1')$$

where c_0' is the code generated from e_0 and c_1' is the code generated from e_1. We will proceed by sketching the proof for parallel space and sequential space preservation separately:

1. Parallel space: Assume (A)

$$c_0 \vdash init(c_0) \Rightarrow^* \Omega_1^{\mathrm{p}} \,\$\, S'$$

and:

$$c_0; (c_0'; c_1') \vdash init(c_0; (c_0'; c_1')) \Rightarrow^* \Omega_2^{\mathrm{p}} \,\$\, S.$$

which we assume is the same as (B):

$$(c_0; c_0'); c_1' \vdash init((c_0; c_0'); c_1') \Rightarrow^* \Omega_2^{\mathrm{p}} \,\$\, S,$$

where we have associated the SVCODE fragments differently. It then remains to show that

$$S - S' = O(\Sigma_{i=1}^{l}(\max(L_{0,i}, L_{1,i} + \mathcal{P}\|v_{0,i}\|))).$$

Since we assumed (B), we can conclude (C):

$$c_0; c_0' \vdash (c_0; c_0') \Rightarrow^* \Omega_3^{\mathrm{p}} \,\$\, S_0.$$

For some intermediate result-state Ω_3^{p} and some space cost S_0. That is, we can remove instruction from the end of an SVCODE program without introducing non-termination. This is obviously true but formally requires a separate lemma. Now, by using the induction hypothesis on e_0 on (A) and (C) with all the other assumption unchanged we get:

$$S_0 - S' = O(\Sigma_{i=1}^{l} L_{0,i}).$$

Using the induction hypothesis on e_1 (with all the assumption corrected to account for the new binding) on (C) and (B) we get

$$S - S_0 = O(\Sigma_{i=1}^{l} L_{1,i}).$$

Rearranging the two equations for space we get:

$$S - S' = O(\Sigma_{i=1}^{l} L_{0,i}) + O(\Sigma_{i=1}^{l} L_{1,i}) = O(\Sigma_{i=1}^{l}(L_{0,i} + L_{1,i})).$$

Unfortunately, this is greater (or at least different) than what we must show, i.e.

$$S - S' = O(\Sigma_{i=1}^{l}(\max(L_{0,i}, L_{1,i} + \mathcal{P}\|v_{0,i}\|))).$$

The reason is that we have not accounted for deallocation in the operational semantics. In order to complete this case, the parallel semantics must be able to capture that streams that are no longer

referred to are freed. For let-bindings, that means that the streams in e_0 that do not constitute the stream values bound to x are freed after evaluating e_0. For example, in the parallel evaluation of

$$\textbf{let } x = e_0 \textbf{ in } \phi(x),$$

the value v_0 of e_0 can be safely discarded once it is consumed by $\phi(x)$. If a stream escapes the scope, e.g. x is visible in e_1 in

$$\textbf{let } y = \textbf{let } x = e_0 \textbf{ in } (\phi(x), x) \textbf{ in } e_1,$$

its size is accounted for in the cost model by the term $\mathcal{P}\|(v_0', v_0)\|$ that arises when costing the outer let-binding. v_0' is the value of $\phi(v_0)$.

This has the side-effect that if the value occurs multiple time, such as in

$$\textbf{let } x = e_0 \textbf{ in } (\phi(x), x, x),$$

then we charge the size of x twice. In our actual implementation we do not copy x twice. Since our cost model is conservative, this is not a problem.

2. Sequential space: This case is very much the same as the case for parallel space, except that we actually arrive at the correct result. The cost model for sequential space in let-bindings combines the cost of e_0 and e_1 by summation. This is because, unlike for parallel space, we cannot assume that it is safe to deallocate streams defined in e_0 after the scope of x. The dataflow nature of the sequential semantics might require us to step e_1 before e_0 is completed.

- Case $e = \{e : x \textbf{ in } x_0 \textbf{ using } x_1^{\tau_1}, ..., x_k^{\tau_k}\}$: There is also only one big-step evaluation rule that applies, so each of the big-step derivations must be:

$$\frac{\rho_i(x_0) = \{v_{i,1}, ..., v_{i,l_i'}\} \quad \left(\rho_i[x \mapsto v_{i,j}] \vdash e_0 \Downarrow v_{i,j}' \,\$\, \omega_{i,j}'\right)_{j=1}^{l_i'}}{\rho_i \vdash \{e_0 : x \textbf{ in } x_0 \textbf{ using } x_1^{\tau_1}, ..., x_k^{\tau_k}\} \Downarrow \{v_{i,1}', ..., v_{i,l_i'}'\} \,\$\, \omega_i}$$

Here, for each i in $1..l$, each $\omega_{i,j}'$ (for j in $1..l_i'$) is related to ω_i as defined by the cost model. We will give more precise definitions in the subcases to follow. The translation of e takes the form

$$c_0; c = c_0; \mathtt{s}_i := \mathtt{usum}\ \mathtt{s}_{\text{ctrl}}; c_{\text{dist}}; c_0'$$

where c_{dist} is the SVCODE generated from distributing the using variables and c_0' is the SVCODE generated from the body expression e_0.

1. Parallel space: According to the cost model, the parallel space for evaluating e in ρ_i (for $i = 1..l$) is

$$\omega_{i,L} = l'_i + l'_i \cdot \Sigma_{i=1}^{k} |\tau_i| + \Sigma_{j=1}^{l'_i} \omega'_{i,j,L}.$$

By assuming evaluation of c_0 and evaluation of $c_0; c$, we arrive to a point where it remains to show that

$$\begin{aligned} S - S' &= O\left(\sum_{i=1}^{l} (l'_i + l'_i \cdot \Sigma_{i=1}^{k} |\tau_i| + \Sigma_{j=1}^{l'_i} \omega'_{i,j,L})\right) \\ &= O\left(\sum_{i=1}^{l} l'_i + \sum_{i=1}^{l} (l'_i \cdot \Sigma_{i=1}^{k} |\tau_i|) + \sum_{i=1}^{l} (\Sigma_{j=1}^{l'_i} \omega'_{i,j,L})\right) \\ &= O\left(l'' + l'' \cdot \Sigma_{i=1}^{k} |\tau_i| + \sum_{i=1}^{l} (\Sigma_{j=1}^{l'_i} \omega'_{i,j,L})\right) \text{ where } l'' = \sum_{i=1}^{l} l'_i \end{aligned}$$

We must account for the space cost of `usum`, the distributions, and c. l'' is the parallel degree of the body, and consequently the length of the streams defined by `usum` and `dist`. The space cost of `usum` is accounted for by the first term l''. By a simple inductive proof of the concrete types, the space cost of the distributions are accounted for by the second term $l'' \cdot \Sigma_{i=1}^{k} |\tau_i|$ (both also work when $l'' = 0$, in which case `usum` and `dist` does not allocate anything), the space cost of c requires two subcases: $l'' = 0$ and $l'' > 0$.

   - Subcase $l'' = 0$: If $l'' = 0$, then each of $l'_i = 0$ for $i = 1..l$ and $\sum_{i=1}^{l} (\Sigma_{j=1}^{l'_i} \omega'_{i,j,L}) = 0$. Therefore, the execution of the translation of e_0 is required to use no space in the parallel semantics. One way of showing this, could be by a separate lemma that shows that when the control stream is empty, no space is used. This should go through as all instructions results in the empty stream on empty stream arguments.
   - Subcase $l'' > 0$: Induction hypothesis on e_0 gives a space cost of

$$S - S_0 = O(\sum_{i=1}^{l} (\Sigma_{j=1}^{l'_i} \omega'_{i,j,L})),$$

     where S_0 is the parallel space cost that includes everything so far as well as `usum` and the distributions. This cost is accounted for by the last term as they are identical.

2. Sequential space: According to the cost model, the seqeuntial space for evaluating e in the different environments are

$$\omega_{i,M} = 1 + \Sigma_{i=1}^{k}|\tau_i| + \max_{j=1}^{l'_i} \omega'_{i,j,M}$$

and

$$\omega_{i,N} = \max_{j=1}^{l'_i} \omega'_{i,j,N}.$$

The core equation we have to show here is then:

$$\begin{aligned} & S - S' \\ = \; & O\left(\max_{i=1}^{l}\left(1 + \Sigma_{i=1}^{k}|\tau_i| + \max_{j=1}^{l'_i}\omega'_{i,j,M}\right) + \max_{i=1}^{l}\left(\max_{j=1}^{l'_i}\omega'_{i,j,N}\right)\right) \\ = \; & O\left(1 + \Sigma_{i=1}^{k}|\tau_i| + \max_{i=1}^{l}\left(\max_{j=1}^{l'_i}\omega'_{i,j,M}\right) + \max_{i=1}^{l}\left(\max_{j=1}^{l'_i}\omega'_{i,j,N}\right)\right) \end{aligned}$$

`usum` and the distributions are accounted for by the terms $1 + \Sigma_{i=1}^{k}|\tau_i|$ because both `usum` and `dist` define a single bounded buffers. For the translation and evaluation of e_0 we consider two sub-cases:

- Subcase $l'_i = 0$ for all $i \in 1..l$: So

$$\max_{i=1}^{l}\left(\max_{j=1}^{l'_i}\omega'_{i,j,M}\right) + \max_{i=1}^{l}\left(\max_{j=1}^{l'_i}\omega'_{i,j,N}\right) = 0.$$

  This subcase is similar to the subcase for parallel space. If e_0 requires no space, then it cannot have any active buffers, and there is nothing to account for.
- Subcase $l'_i > 0$ for some $i \in 1..l$: Induction on e_0 gives a space cost of

$$S - S_0 = O\left(\max_{i=1}^{l}(\max_{j=1}^{l'_i}\omega'_{i,j,M}) + \max_{i=1}^{l}(\max_{j=1}^{l'_i}\omega'_{i,j,N})\right),$$

  where S_0 is the sequential space cost that includes everything so far as well as `usum` and the distributions. This cost is accounted for by the last term.

- Case $\{e \mid x_0$ **using** $x_1^{\sigma_1}, ..., x_k^{\sigma_k}\}$: The case for conditional comprehensions is similar to apply-to-each. `dist` and `pack` are basically the same operation with respect to space cost.

Chapter 6

Conclusion

The conclusions of the previous chapters indicates a positive assessment of our hypothesis: Data-parallel functional programming languages *can* become more space-efficient by using streaming semantics instead of being fully eager. We have demonstrated so for both GPUs and CPUs. As a general technique, we have demonstrated how data-parallel languages can be extended with sequences:

1. Many fully manifesting vector computations can be re-expressed as sequence computation allowing a large class of problems to be formulated statically as streamable.

2. Sequences integrate well with the existing languages. We support manifest vector-to-sequence conversion through sequencing and sequence-to-vector conversion through tabulation. Lifting (and flattening in the case of SNESL) allows efficient support chunking – including sequences of vectors where each element is generated by a vector computation.

3. Sequences offer the same time-performance characteristics as vectors, and offer much better space performance.

4. For SNESL, we have been able to formulate an intuitive, high-level, language-integrated cost model for space that clearly describes the improved space-performance characteristics of sequences over vectors.

We have been able to demonstrate high performance for data-parallel streaming, by extending two existing languages and backends: Streaming NESL with our own streaming multicore and vector instruction CPU backend, and Streaming Accelerate with an extended streaming GPU backend.

For SNESL, we have shown significant performance improvements on CPUs over GNU coreutils text processing tools. Here, execution times are

sometimes faster due to better cache utilization. Moreover, on a single thread, execution times may even be faster than sequential C code in situations where the C code is written in such a way that the compiler cannot apply automated vectorization whereas we can.

For Streaming Accelerate, we have shown that much larger data sets can be processed with virtually no additional effort from the programmer on a number of representative benchmarks. Unlike CPUs, GPUs have no means of recovering when running out of memory, so a streaming implementation for GPUs is quite valuable, especially considering that GPUs are very attractive candidates for big data and big compute.

For both NESL and Accelerate we have integrated a large subset of the language as part of the syntax for sequences. Streaming Accelerate is a bit more restricted in the sense that only regular sequences (sequences of arrays, where the arrays have the same size) are executed predictably efficient. This seems to reflect the general regularity of the language in a natural way.

In the case of Streaming NESL, we have provided an optimistic cost model for space usage, that guarantees streaming, and that is completely independent of hardware parameters – all while leaving the time cost model intact, guaranteeing parallel execution. Although we have not formally shown it, our experiments certainly indicate that the cost models are respected by our DPFlow backend for a representative collection of benchmarks.

6.1 Further Work Summary

Streaming Accelerate As of this writing, the language is under active development, and some of the points are being addressed by the people working on Accelerate at the University of New South Wales. The points may be summarized as:

- Chunked execution of irregular sequences.
- Low-level optimizations such as automatic sequentialisation, overlapping of data transfer and computation and multi-GPU support.

SNESL The future work mentioned in the two SNESL papers remains mostly unsolved. The one exception is that the second paper implements a multicore backend for SNESL, which was a point of future work in the first paper. The other points may be summarized as:

- Extending the language and cost model with recursion.
- Static checking of schedulability.

- Formally establishing the time and space efficiency of the implementation model.
- Extending the model to account for bulk random-access vector *writes* (permutes, or more generally, combining-scatter operations) in order to support histogramming and bucket sorting.

Even without these points, Streaming Accelerate and SNESL are both already useful languages for high-level functional data-parallel streaming. The underlying principle of sequence expressions and a chunked dataflow execution model could certainly have interest and bring value to other functional data-parallel languages. In a world where storage, computations and data sets are getting larger, but the speed of light remains the same, streaming will eventually become an unavoidable concern for any data-parallel language. This thesis breaks new ground towards platform-independent, cost-model guided streaming.

Bibliography

[BCH+94] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[BG96] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *International Conference on Functional Programming, ICFP'96*, pages 213–225, Philadelphia, Pennsylvania, May 1996.

[BGM99] Guy E. Blelloch, Phillip B Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of ACM*, 46(2):281–321, March 1999.

[Ble90a] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, November 1990.

[Ble90b] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[Ble92] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103; updated version: CMU-CS-05-170, School of Computer Science, Carnegie Mellon University, 1992.

[Ble95] Guy E. Blelloch. NESL: A nested data-parallel language (3.1). Technical report, 1995. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-95-170.ps.gz`.

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

[BR12] Lars Bergstrom and John Reppy. Nested data-parallelism on the GPU. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, ICFP '12, pages 247–258, Copenhagen, Denmark, 2012. ACM.

[BS90] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8(2):119–134, February 1990.

[Buc03] Ian Buck. Brook language specification, October 2003. `http://merrimac.stanford.edu/brook`.

[CBZ90] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 666–675, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[CKL+11] Manuel M. Chakravarty, Gabriele Keller, Sean Lee, Trevor McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, Austin, Texas, USA, 2011. ACM.

[CLJ+07] Manuel M. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, Nice, France, 2007. ACM.

[CSS08] Koen Claessen, Mary Sheeran, and Joel Svensson. Obsidian: GPU programming in Haskell. In *IFL: Implementation and Application of Functional Languages*, 2008.

[Ell04] Conal Elliott. Programming graphics processors functionally. In *Haskell Workshop*. ACM Press, 2004.

[GLGLBG12] Juan GóMez-Luna, José MaríA GonzáLez-Linares, José Ignacio Benavides, and NicoláS Guil. Performance models for asynchronous data transfers on consumer graphics processing units. *J. Parallel Distrib. Comput.*, 2012.

[HSW+11] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable stream programming on graphics engines. *SIGARCH Comput. Archit. News*, 39(1):381–392, March 2011.

[KCL+10] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *ICFP: International Conference on Functional Programming*. ACM, 2010.

[KCL+12] Gabriele Keller, Manuel M. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. Vectorisation avoidance. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 37–48, Copenhagen, Denmark, 2012. ACM.

[KS73] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, 100(8):786–793, 1973.

[KS96] Gabriele Keller and Martin Simons. A calculational approach to flattening nested data parallelism in functional languages. In *Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security*, ASIAN '96, pages 234–243, Singapore, 1996. Springer-Verlag.

[Lar11] Bradford Larsen. Simple optimizations for an applicative array language for graphics processors. In *DAMP: Declarative Aspects of Multicore Programming*. ACM, 2011.

[LCK06] Roman Leshchinskiy, Manuel MT Chakravarty, and Gabriele Keller. Higher order flattening. In *Computational Science–ICCS 2006*, pages 920–928. Springer, 2006.

[LCK+12] Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon L. Peyton Jones. Work efficient higher-order vectorisation. In *International Conference on Functional Programming, ICFP'12*, pages 259–270, Copenhagen, Denmark, September 2012.

[LCKPJ12] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Guiding parallel array fusion with indexed types. In *Haskell Symposium*. ACM, 2012.

[LM87] Edward Ashford Lee and David G Messerschmi'tt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 2(36), 1987.

[Mad12] Frederik M. Madsen. Flattening nested data parallelism. Master project. 2012. `http://www.diku.dk/~fmma/publications/nested.pdf`.

[Mad13] Frederik M. Madsen. A streaming model for nested data parallelism. Master's thesis, DIKU, University of Copenhagen, March 2013. `http://www.diku.dk/~fmma/publications/thesis-report%20(handin).pdf`.

[MCECK15] Frederik M. Madsen, Robert Clifton-Everest, Manual M. T. Chakravarty, and Gabriele Keller. Functional array streams. In *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '15, pages 23–34, New York, NY, USA, 2015. ACM.

[MCKL13] Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *ICFP: International Conference on Functional Programming*, September 2013.

[MF13] Frederik M. Madsen and Andrzej Filinski. Towards a streaming model for nested data parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 13–24, New York, NY, USA, 2013. ACM.

[MF16] Frederik M. Madsen and Andrzej Filinski. Streaming nested data parallelism on multicores. In *Proceedings of the 5th ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '16, New York, NY, USA, 2016. ACM.

[MG09] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. *University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14*, 2009.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Haskell Symposium*. ACM, 2010.

[NHP07] Lars Nyland, Mark Harris, and Jan Prins. Chapter 31. Fast N-Body Simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison-Wesley Professional, 2007.

[NVI12] NVIDIA. CUDA C Programming Guide, 2012.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[PJ08] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the Sixth Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–150, Bangalore, India, 2008. Springer-Verlag.

[PP93] Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 119–128, San Diego, California, USA, 1993. ACM.

[PPCF95] Daniel W. Palmer, Jan F. Prins, Siddhartha Chatterjee, and Rickard E. Faith. Piecewise execution of nested data-parallel programs. In *Languages and Compilers for Parallel Computing, 8th International Workshop, LCPC'95*, volume 1033 of *Lecture Notes in Computer Science*, Columbus, Ohio, August 1995.

[PPW95] Daniel W. Palmer, Jan F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, FRONTIERS '95, pages 186–193, McLean, Virginia, 1995. IEEE Computer Society.

[Riv92] Ronald Rivest. The MD5 message-digest algorithm. 1992.

[RPI95] J. W. Riely, Jan F. Prins, and S. P. Iyer. Provably correct vectorization of nested-parallel programs. In *Proceedings of the Conference on Programming Models for Massively Parallel Computers*, PMMP '95, pages 213–222, Washington, DC, USA, 1995. IEEE Computer Society.

[RS] John Reppy and Nora Sandler. Nessie: A NESL to CUDA compiler. Paper presented at the 18th International Workshop on Compilers for Parallel Computing, CPC '15, London.

[RSA+13] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Martin Odersky, and Kunle Olukotun. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL'13*. ACM, 2013.

[SBHG08] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming, ICFP'08*, pages 253–264, Victoria, BC, Canada, September 2008.

[Sch03] Sven-Bodo Scholz. Single Assignment C: Efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[She93] Thomas J. Sheffler. Implementing the multiprefix operation on parallel and vector computers. In *Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 377–386, 1993.

[Skl60] Jack Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic computers*, (2):226–231, 1960.

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, 2002.

[ZM12] Yongpeng Zhang and Frank Mueller. CuNesl: Compiling nested data-parallel languages for SIMT architectures. In *41st International Conference on Parallel Processing, ICPP 2012*, pages 340–349, 2012.