Streaming Nested Data Parallelism on Multicores *

Frederik M. Madsen Andrzej Filinski
Department of Computer Science (DIKU)
University of Copenhagen, Denmark
{fmma,andrzej}@di.ku.dk

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.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming; D.3.2 [*Programming Languages*]: Language Classifications—applicative (functional) languages, dataflow languages; D.3.4 [*Programming Languages*]: Processors—compilers, interpreters

General Terms Languages, Performance

Keywords Nested data parallelism, streaming, dataflow

1. Introduction

Many common data processing tasks are highly repetitive, yet not easily parallelizable by traditional techniques. Such tasks are often characterized by nested loops in which the number of inner iterations is data dependent and non-uniform. This irregularity presents an obstacle to simple parallelization approaches, such as OpenMP annotations for multicore execution, or effective use of SIMD instructions, since any static classification of individual loops as sequential or parallelizable is likely to lead to substantially suboptimal performance for particular data patterns. In particular, if the parallelization granularity is a poor fit for the data, performance may suffer considerably from poor load balancing and/or excessive synchronization and bookkeeping overheads.

An often useful alternative approach to parallelization in such cases is the paradigm of *flattening nested data parallelism*, pioneered by the NESL language (Blelloch 1992). As in other divide-and-conquer parallelization strategies, the idea is to partition the problem instance into a number of independent subproblems, such that the solutions to the subproblems can be combined into a solution of the whole, and such that each subproblem of non-trivial size can itself be effectively parallelized – either by already being naturally data parallel, or by a logarithmic-depth recursive subdivision.

Crucially, however, in the flattening approach, there is no implied requirement that the subproblems be of uniform, or even roughly similar sizes, nor that the subproblem boundaries are aligned with processor allocations – either as several subproblems per processor, or as several processors per subproblem. Instead, all the available work (which, possibly after further subdivisions, we may assume to be of uniform nature), is evenly divided among all processors, ensuring that none go idle.

As noted by Blelloch (Blelloch 1991), the class of problems amenable to such an approach is remarkably large. In particular, suppose the task can be effectively expressed in terms of nested combinations of independent element-wise operations (maps), including random-access CREW load/store operations (gather/scatter), and efficiently parallelizable combining operations in the form of *reduces* (associative folds) and *scans* (prefix sums). Then the whole computation can be flattened into a sequence of global per-element operations and *segmented scans*, where a segmented scan also takes a vector of segment-boundary flags, and resets the summation at each boundary. Somewhat surprisingly, the tree-structured parallel algorithms for scans can be refined to take segment flags into account with very little extra cost. And crucially, the resulting algorithm exhibits no *control* or *communication-pattern* dependence on the segment flags, so that it is still highly amenable to SIMD style execution; and its performance remains largely uniform and predictable from a high-level work/step cost model.

A fundamental drawback, however, is that this approach a priori requires the whole problem (including in its intermediate stages) to fit in memory at once, which is particularly problematic if the available parallelism far exceeds the computational resources. For example, the direct data-parallel rendering of $\sum_{i=1}^{n} f(i)$, where f is some simple function, nominally first computes a vector of all the n values of $f(i)$ in parallel, and addition-reduces it (typically in a tree-like fashion) afterwards. Even when this is feasible (possibly through paging to secondary storage), it incurs a considerable overhead due to poor cache utilization and extra memory traffic. More fundamentally, it goes against the programmer's intuition that such a summation should not require more than $O(p)$ memory, where p is the number of processors.

* This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center HIPERFIT: Functional High Performance Computing for Financial Information Technology (`hiperfit.dk`) under contract number 10-092299.

FHPC'16, September 22, 2016, Nara, Japan
ACM. 978-1-4503-4433-3/16/09...$15.00
http://dx.doi.org/10.1145/2975991.2975998

A natural refinement, therefore, consists of not materializing vector values in their entirety, but sequentially in $O(p)$-sized *chunks*, in a streaming fashion. Like for processor allocations, chunk and segment boundaries are in general not aligned. Maps, reduces, and scans (including segmented ones) are all naturally streamable, and even random-access scatter/gather operations do not inherently require more space than the length of the longest vector in a sequential implementation.

Depending on the platform, a chunk would typically contain 10^4–10^6 elements: large enough that scheduling and synchronization overhead becomes relatively insignificant, but still small enough to fit into cache or other fast memory, so that multiple sequential passes over a single chunk of data are not substantially slower than a single, fused pass.

While the semantics of original NESL language does not preclude a streaming SIMD implementation (Palmer et al. 1995a), or even a MIMD-oriented approach (Blelloch and Greiner 1996), the language is not particularly well suited for such an implementation 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 (Madsen and Filinski 2013) 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 (Bergstrom and Reppy 2012). Previously (Madsen et al. 2015), 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.

2. Streaming NESL Language and Compiler

In the following, we sketch the source language and the main phases in the compilation process. More details about SNESL and its execution model can can be found in (Madsen and Filinski 2013).

Name	Type	Examples
$+$, chr, ...	$(\delta_1, ..., \delta_k) \to \delta_0$	$2 + 3 = 5$, chr$(65) =$ 'A'
sel $(? :)$	$(\mathsf{bool}, \delta, \delta) \to \delta$	sel$(2 \le 2, 3, 4) = 3$
length $(\#)$	$[\tau] \to \mathsf{int}$	length$([2, 3, 5, 7]) = 4$
elt $(!)$	$([\tau], \mathsf{int}) \to \tau$	$[2, 3, 5, 7] \,!\, 2 = 5$
iota $(\&)$	$\mathsf{int} \to \{\mathsf{int}\}$	&4 = {0,1,2,3}
$++$	$(\{\sigma\}, \{\sigma\}) \to \{\sigma\}$	$\{2, 3\} ++ \{5\} = \{2, 3, 5\}$
concat	$\{\{\sigma\}\} \to \{\sigma\}$	concat$(\{\{2, 3\}, \{\}, \{5\}\}$ $= \{2, 3, 5\}$
part	$(\{\sigma\}, \{\mathsf{bool}\})$ $\to \{\{\sigma\}\}$	part$(\{2, 3, 5\}, \{\mathsf{f}, \mathsf{f}, \mathsf{t}, \mathsf{t}, \mathsf{f}, \mathsf{t}\})$ $= \{\{2, 3\}, \{\}, \{5\}\}$
sep	$\{(\sigma, \mathsf{bool})\}$ $\to \{\{\sigma\}\}$	sep$((2, \mathsf{f}), (3, \mathsf{t}), (4, \mathsf{f}))$ $= \{\{2, 3\}, \{4\}\}$
zip	$(\{\sigma_1\}, \{\sigma_2\})$ $\to \{(\sigma_1, \sigma_2)\}$	zip$(\{2, 3\}, \{4, 5\})$ $= \{(2, 4), (3, 5)\}$
the	$\{\sigma\} \to \sigma$	the$(\{3\}) = 3$
seq	$[\tau] \to \{\tau\}$	seq$([2, 3, 5]) = \{2, 3, 5\}$
tab	$\{\tau\} \to [\tau]$	tab$(\{2, 3, 5\}) = [2, 3, 5]$
reduce$_\odot$	$\{\delta\} \to \delta$ where $\odot :: (\delta, \delta) \to \delta$	reduce$_+(\{2, 3, 5, 7\}) = 17$
scan$_\odot$	$\{\delta\} \to \{\delta\}$ where $\odot :: (\delta, \delta) \to \delta$	scan$_+(2, 3, 5, 7)$ $= \{0, 2, 5, 10\}$

Figure 1. Selected builtin functions

2.1 Front-End Language (SNESL)

SNESL appears as a fairly traditional, statically typed functional language with a few specialized types and operations. Its central concepts are laid down in the type structure (presented in Haskell-style notation):

$$\begin{array}{lll} \delta & ::= & \mathsf{int} \mid \mathsf{real} \mid \mathsf{bool} \mid \mathsf{char} \quad \text{primitive types} \\ \tau & ::= & \delta \mid (\tau_1, , .., \tau_k) \mid [\tau] \quad \text{concrete types} \\ \sigma & ::= & \tau \mid (\sigma_1, ..., \sigma_k) \mid \{\sigma\} \quad \text{general types} \end{array}$$

(Here, and in the following, $k \ge 0$.) Values of concrete types (including vectors $[\tau]$) are fully materialized, while those of general types (including sequences $\{\sigma\}$) need not be. While it is possible to have a sequence of integer vectors, the opposite is not possible.

Apart from insignificant syntactic conveniences (e.g., infix operators), the syntax of expressions is as follows:

$$\begin{array}{lll} e & ::= & x \mid n \mid n.n \mid \mathsf{true} \mid \mathsf{false} \mid \texttt{'}c\texttt{'} \mid \texttt{"}c_1 ... c_k\texttt{"} \mid (e_1, ..., e_k) \\ & & \mid [e_1, ..., e_k] \mid \{e_1, ..., e_k\} \mid \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \mid f(e_1, ..., e_k) \\ & & \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \{e_0 : p_1\ \mathsf{in}\ e_1; ...; p_k\ \mathsf{in}\ e_k \mid e_0'\} \\ p & ::= & \_ \mid x \mid (p_1, ..., p_k) \end{array}$$

The syntax is made useful by a collection of built-in functions f, with the main ones listed in Figure 1. Their meanings should hopefully be clear from name, type, and example. We also allow a program to include a number of global function definitions,

$$\mathsf{fun}\ f(p_1, ...p_k) = e$$

Function definitions are cumulative (later functions may call earlier-defined ones), but not recursive. (Supporting recursion is a planned extension, as discussed in the Conclusion.)

The semantics and typing rules for all language constructs are what one would expect, with the possible exception of sequence comprehensions, in which the bindings are zip-like, not sequential like in Haskell; for example,

$$\{x + y : x\ \mathsf{in}\ \{2, 3, 5\}; y\ \mathsf{in}\ \{9, 4, 1\} \mid x < y\}$$

evaluates to $\{11, 7\}$. The boolean guard e_0' may be omitted when constantly true.

More unusually, the *intensional* semantics of sequence comprehensions 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.

In support of this exactly-once semantics, the typing rule for comprehensions also stipulates that any free variables in e_0 and e'_0 that are not bound by $p_1, ..., p_k$ (i.e., that represent values constant throughout the computation) must be of concrete type, and hence already fully evaluated. For example,

$$\mathsf{let}\ s = \{10 * x : x\ \mathsf{in}\ \&3\}\ \mathsf{in}\ \{s +\!\!+ \{y\} : y\ \mathsf{in}\ \{40, 50\}\}$$

is considered ill-typed, because it would require s to either be memoized in its entirety or recomputed. The latter behavior can be achieved simply by syntactically unfolding the let (we may offer some explicit notation for that). If the former semantics is desired, we must instead explicitly materialize the sequence into a concrete value, and then re-stream it every time it's needed:

$$\mathsf{let}\ v = \mathsf{tab}(\{10{*}x : x\,\mathsf{in}\,\&3\})\ \mathsf{in}\ \{\mathsf{seq}(v) +\!\!+ \{y\} : y\,\mathsf{in}\,\{40, 50\}\}$$

Note, however, that if a comprehension contains no bindings, only a guard, there are no restrictions on the free variables:

$$\mathsf{let}\ s = \{10 * x : x\ \mathsf{in}\ \&3\}\ \mathsf{in}\ \{s \mid y > 0\}$$

will evaluate to either {{0,10,20}} or {}, depending on y, but in either case, no recomputation of s is necessary.

The type system does not guard against all errors. For example, like in NESL, a zip-like comprehension with sequences of unequal length will explicitly fail at runtime. (A Haskell-like, longest-common-prefix semantics might be more useful if sequences could potentially be infinite, but that is not possible here.) More subtly, if the sequences to be zipped cannot be produced at the same *rate*, the computation may deadlock. To guard statically against this, we would need a more refined type system, e.g., with clocks; for now, this is left as future work. In practice, though, in most cases where sequences are manifestly of the same length, they will also be derived from the same source, and can hence also be produced synchronously.

Built-in functions with multiple stream-typed arguments may also have synchronization constraints, more involved than for zip For example, $s +\!\!+ \{\mathsf{reduce}_+(s)\}$ is OK, but $\{\mathsf{reduce}(s)\} +\!\!+ s$ is not. Intuitively, computations that could in principle be expressed in constant space using loops in a sequential language will normally be unproblematic, but others might not be.

2.2 Core SNESL

As the first proper step in the compilation process, programs are desugared into a subset of the language that we call Core SNESL. Key transformations include:

- Type inference and monomorphization (i.e., selection of specific type instances of polymorphic functions).
- Unfolding of programmer-provided function definitions.
- Desugaring of conditionals into guarded comprehensions:

  $\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \equiv \mathsf{let}\ b = e_0\ \mathsf{in}\ \mathsf{the}(\{e_1 \mid b\} +\!\!+ \{e_2 \mid \mathsf{not}(b)\}$

  (If both branches are already simple values, sel can be used instead.)
- Desuagaring of parallel bindings:

  $$\{e : x_1\ \mathsf{in}\ e_1; x_2\ \mathsf{in}\ e_2\} \equiv \{e : (x_1, x_2)\ \mathsf{in}\ \mathsf{zip}(e_1, e_2)\}$$

- Decomposition of comprehensions containing both bindings and guards:

  $$\{e_0 : x\ \mathsf{in}\ e_1 \mid e'_0\} \equiv \mathsf{concat}(\{\{e_0 \mid e'_0\} : x\ \mathsf{in}\ e_1\})$$

- Desugaring of patterns into explicit projections.

There is a reference interpreter for Core SNESL, including a cost semantics covering both time aspects (work, step) and space usage (parallel, sequential). However, the main implementation strategy is compilation, using flattening, into a lower-level intermediate language.

2.3 Streaming VCODE (SVCODE)

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

$$\mathsf{s}_i := p_i(\mathsf{s}_{i_1}, ..., \mathsf{s}_{i_k})$$

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

It is possible to execute SVCODE in an essentially VCODE-like, non-streaming fashion, by simply performing the instructions in sequence, fully materializing the results (and possibly garbage-collecting those that will not be subsequently used, which can be done by simple reference-counting). The intended implementation model, however, is data-flow execution, described in the next section.

The key to the flattening transform from Core SNESL to SVCODE is the treatment of nested sequences. Like in NESL, we represent a nested sequence, such as $\{\{2, 3\}, \{\}, \{5\}\}$, as a separate stream of the underlying data values, $\langle 2, 3, 5\rangle$, and a segment-descriptor stream. The latter, however, is based on boundary flags, rather than on segment lengths, i.e., $\langle \mathsf{f}, \mathsf{f}, \mathsf{t}, \mathsf{t}, \mathsf{f}, \mathsf{t}\rangle$, rather than $\langle 2, 0, 1\rangle$. In fact, the builtin part function conceptually just glues the data and descriptor streams together, while concat discards the descriptor stream.

We are thus effectively representing segment lengths in *unary*, which in practice does not impose a significant space overhead, while allowing us to express various stream transformations in constant space, without needing to know the current segment length. On the other hand, for a sequence of vectors, such as $\{[2, 3], [], [5]\}$, the vector lengths are represented compactly as a stream of non-negative integers, making length a constant-time operation as expected. Thus, the main task of the seq and tab functions actually becomes to convert between the two forms of segment descriptors, rather than the data values.

3. DPFlow: A Multicore Interpreter for SVCODE

The key piece that separates this paper from our previous paper (Madsen and Filinski 2013) 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 pre-compiled 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 3.4).

3.1 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 2.

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 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 2.

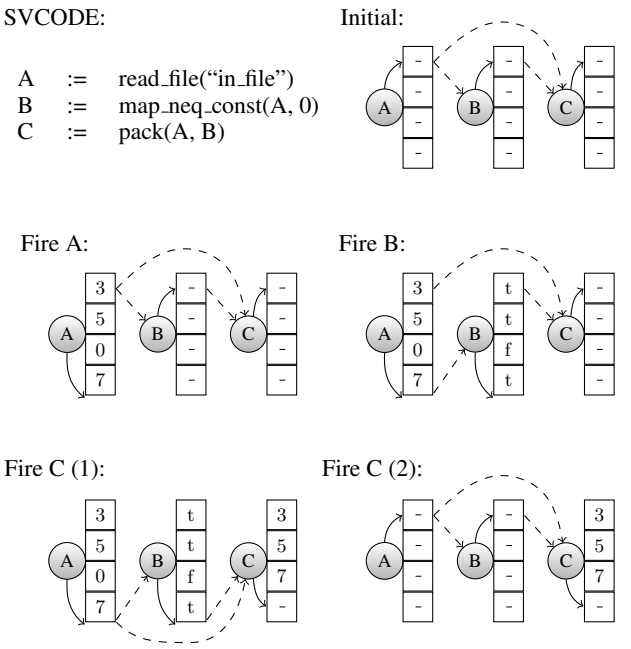


Figure 2. An illustration of a stream transducer network that removes all null characters (0) from a file named "in_file". The figure shows three stream transducers firing one at a time and the contents of their buffers (which are limited to a chunk size of 4). Write cursors are illustrated with solid 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.

3.2 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.

3.3 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 2, 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.

3.4 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.

Based on the Kogge-Stone scan circuit (Kogge and Stone 1973), 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 1.3 and 2.5 depending of the average length of the segments and whether or not they are regular.

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.

3.5 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 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. 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.

4.1 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 3. 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.

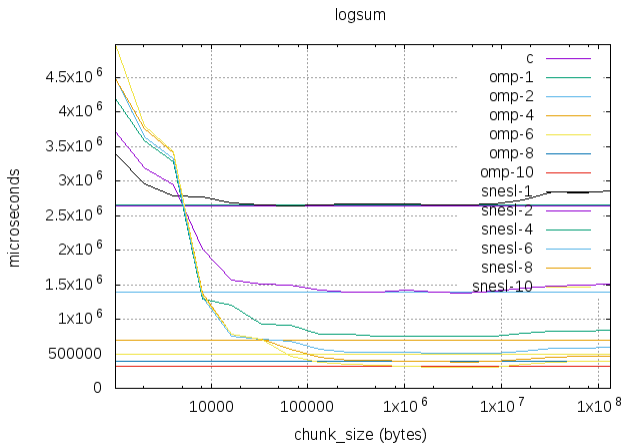
From the graph, we also see that the choice of chunk size does not matter as long as it is between 100 KB and 10 MB. Below 100 KB, the overhead of scheduling and managing buffers and cursor becomes too significant. Above 10 MB we start to exceed the L3 cache. All in all, these results are very promising. It remains to be seen how we fare on more complicated examples.

4.2 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.



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 3. Logsum execution times and speedups.

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.

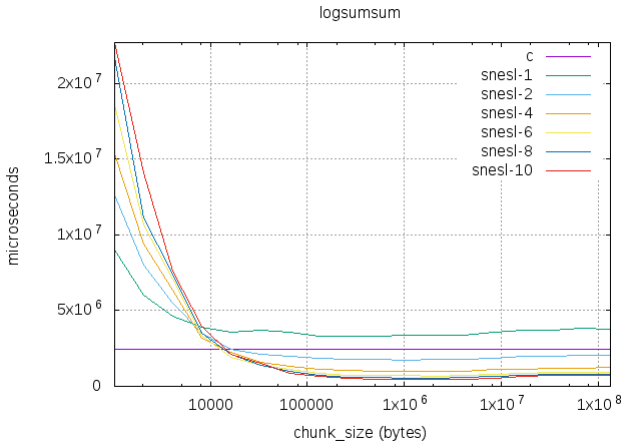
We express logsumsum in SNESL as:

```
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. 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.

4.3 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



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. 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 5. Word count speedups

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:

```
-- 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 5. 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.

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 6. Max line length speedups

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 7. Line reverse speedups

4.4 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
```

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 6. Once again, we are faster than the Linux tool in single-threaded performance, and we scale well with the number of threads.

4.5 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 express 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 7. 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.

4.6 Cut

The cut benchmark explores filtering. Here, we select the ith 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. Otherwise, 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 8. 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.

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 8. Cut speedups

5. 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 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.

Currently, two major aspects of the SNESL implementation require further work. First, as noted in Section 2.1, the current language front-end does not support recursion, or even general iteration. This is not as crippling a restriction as it may seem: since many practical parallel algorithms only require a worst-case logarithmic recursion depth, it is actually feasible (and, with just a little extra front-end support, quite practical) to *statically* unfold the recursion to some maximal depth; for example, 30 levels of unfolding of a binary-tree algorithm, such as a summation tree, would allow for problem sizes up to 10^9, and 50 levels would probably be sufficient for anything within the range of a desktop system. However, to also support algorithms like quicksort, where the recursion, depth bound is only probabilistic, a general-purpose SNESL implementation should have support for dynamically adding nodes to the dataflow graph, as and when additional depth is needed.

Another 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 SVCODE, 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.

References

F. M. Madsen, R. Clifton-Everest, M. M. T. Chakravarty, and G. Keller. Functional Array Streams. In *Fourth ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC'15*, pages 23–34, Vancouver, British Columbia, Sept. 2015.

F. M. Madsen and A. Filinski. Towards a Streaming Model for Nested Data Parallelism. In *Second ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC'13*, pages 13–24, Boston, Massachusetts, Sept. 2013.

L. Bergstrom and J. H. Reppy. Nested data-parallelism on the GPU. In *International Conference on Functional Programming, ICFP'12*, pages 247–258, Copenhagen, Denmark, Sept. 2012.

G. E. Blelloch. Prefix sums and their applications. In *Synthesis of Parallel Algorithms*, Edited by John H. Reif, Morgan Kaufmann, 1991.

G. 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.

G. E. Blelloch and J. 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.

D. W. Palmer, J. F. Prins, S. Chatterjee, and R. 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, Aug. 1995a.

P. M. Kogge,and H. S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence. In *IEEE Transactions on Computers Volume C-22 Issue 8*, pages 786-793, Aug. 1973.