Towards a Streaming Model for Nested Data Parallelism

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

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.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.1.1 [*Applicative (Functional) Programming*]; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

Keywords cost semantics; space efficiency; dataflow networks

FHPC '13, September 23, 2013, Boston, MA, USA.

http://dx.doi.org/10.1145/2502323.2502330

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 [2] 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 [4]. 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 expressing 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. Con-

sequently, 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 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.

[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 $O(\log n)$.

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 [4] (and apparently inherited in both direct derivatives such as NESL-GPU [1], and reimplementations such as CuNesl [13]), 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 [9], or in recent versions of Data Parallel Haskell [6]. 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 [3] 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. [12], 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 [8], 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. 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).

2.1 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 *values* are as follows:

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

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:

$$\begin{aligned}e &::= x \mid a \mid (x_1, ..., x_k) \mid x.i \mid \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 \mid \phi(x) \\ &\quad \mid \{e_0 : x\ \mathbf{in}\ x_0\ \mathbf{using}\ x_1, ..., x_k\} \mid \{e_0 \mid x_0\ \mathbf{using}\ x_1, ..., x_k\} \\ \phi &::= \text{(See Figure 2)}\end{aligned}$$

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

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

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

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

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

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

Figure 1. Typing rules

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 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.)

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. (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 into subsequences, e.g.,

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

(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-

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

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

Figure 2. Primitive operations

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 shorthands. 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.,

$$
\begin{array}{l}
\mathbf{let}\ (x, y) = e\ \mathbf{in}\ e' \equiv \\
\quad \mathbf{let}\ p = e\ \mathbf{in}\ \mathbf{let}\ x = p.1\ \mathbf{in}\ \mathbf{let}\ y = p.2\ \mathbf{in}\ e' \,,
\end{array}
$$

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

$$
\begin{array}{l}
\{e : x_1\ \mathbf{in}\ e_1; \ldots; x_k\ \mathbf{in}\ e_k\} \equiv \\
\quad \{e : (x_1, ..., x_k)\ \mathbf{in}\ zip(e_1, ..., e_k)\} \,.
\end{array}
$$

And we may combine general and predicated comprehensions:

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

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:

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

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

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

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*:

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

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,

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

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

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

(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 function-less 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 nontermination. 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 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 numbers, representing, respectively, the space required to process the value sequentially and in parallel.

More specifically, for any value v, we define $\|v\|$ as a pair of natural numbers (M, N), as follows:

$$
\begin{array}{rcl}
\|a\| & = & (1, 1) \\
\|(v_1, ..., v_k)\| & = & (\Sigma^k_{i=1} M_i, \Sigma^k_{i=1} N_i) \\
 & & \quad where\ \forall i.\ \|v_i\| = (M_i, N_i) \\
\|[v_1, ..., v_l]\| & = & (1 + \Sigma^l_{i=1} M_i, 1 + \Sigma^l_{i=1} N_i) \\
 & & \quad where\ \forall i.\ \|v_i\| = (M_i, N_i) \\
\|\{v_1, ..., v_l\}\| & = & (\max^l_{i=1}(M_i + 1), \Sigma^l_{i=1}(N_i + 1)) \\
 & & \quad where\ \forall i.\ \|v_i\| = (M_i, N_i)
\end{array}
$$

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. Since vectors are always fully materialized, their sequential and parallel sizes are the same.

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. Consequently, for a value v of any concrete type τ, we always have $\|v\| = (M, M)$ for some M, but for general v's, $\|v\| = (M, N)$ with $M \leq N$.

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 4-tuple of natural numbers, $\omega = (W, D; M, N)$, 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_∞). In contrast to time complexity, here we have $S_1 \leq S_\infty$; that is, in order to take advantage of more processors, we will often need more temporary space.

The evaluation rules are given in Figure 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.

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

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

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

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

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

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

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

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

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

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

Figure 3. Evaluation semantics with costs

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 space (sequential or parallel) 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.

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

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

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 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.

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* [8], 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.

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* [6], introduced to avoid the performance anomaly in code like $\{v!i : i\ \mathbf{in}\ a\}$ 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:

$$\begin{aligned} &\{\{3,1,4\},\{\},\{1\},\{5,9\}\} \triangleleft \\ &\quad((\langle 3,1,4,1,5,9\rangle, \langle \mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T},\mathsf{T},\mathsf{F},\mathsf{T},\mathsf{F},\mathsf{F},\mathsf{T}\rangle), \langle \mathsf{F},\mathsf{F},\mathsf{F},\mathsf{F},\mathsf{T}\rangle) \end{aligned}$$

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.

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 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 $[\![\Gamma(x)]\!]$.

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

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:

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

(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:

$$
\begin{aligned}
dist_\pi\ s_0\ s &= pdist\ s_0\ s \\
dist_{(\tau_1,...,\tau_k)}\ (t_1, ..., t_k)\ s &= (dist_{\tau_1}\ t_1\ s, ..., dist_{\tau_k}\ t_k\ s) \\
dist_{[\tau]}\ (t_0, s_\mathrm{s}, s_1)\ s &= (t_0, pdist\ s_\mathrm{s}\ s, pdist\ s_1\ s)\,,
\end{aligned}
$$

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

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

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:

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

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:

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

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:

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

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

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

similar in principle to the control-free fragment of VCODE [4] (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 some-

what 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.

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 [5] version 0.13.0.1, a GPU-enabled language embedded in Haskell.
- An implementation in Single Assignment C (SaC) [10] 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 [1], 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.

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 straightforward mapping of the primitives:

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

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 4 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.

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.

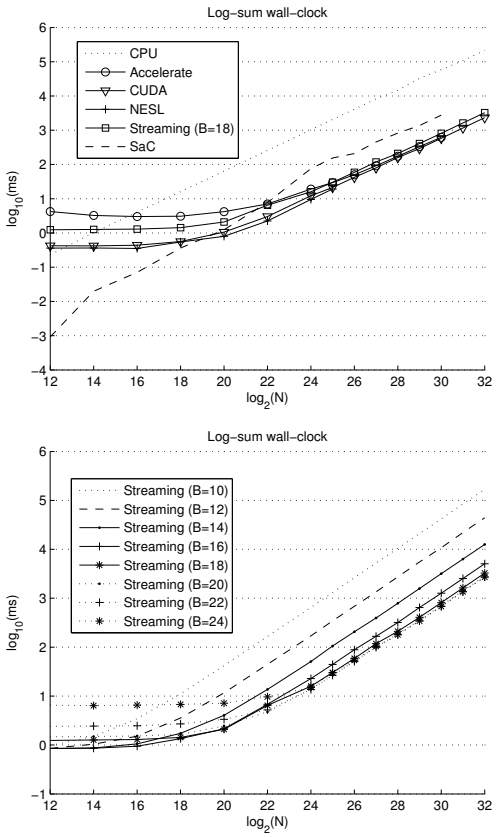

Figure 4. Benchmark results of the log-sum problem. The x-axis is the problem size in base-2 logarithm, and the y-axis is the running time in milliseconds in base-10 logarithm. The upper plot shows the running time of different implementations measured in wall-clock time. The lower plot shows the running time of the streaming implementation for different choices of block sizes.

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:

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

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 5 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.

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 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.

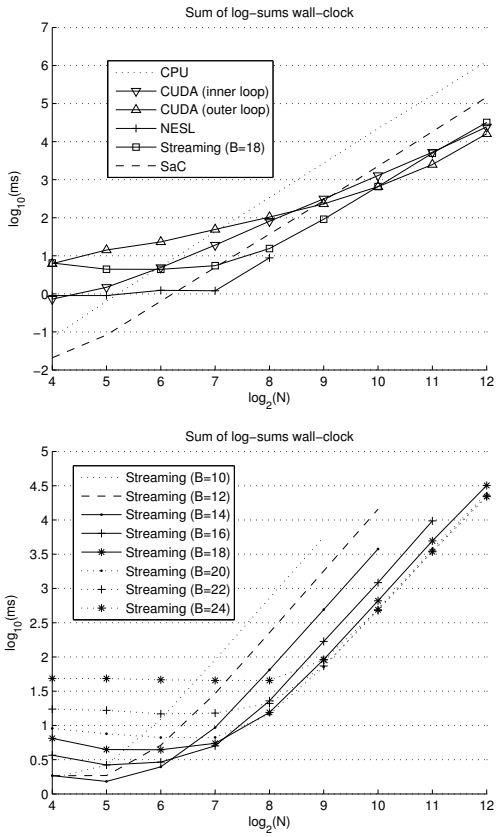

Figure 5. Benchmark results of the sum of log-sums problem, with the same conventions as in Figure 4.

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 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

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

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

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

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* [7] 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 6 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.

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

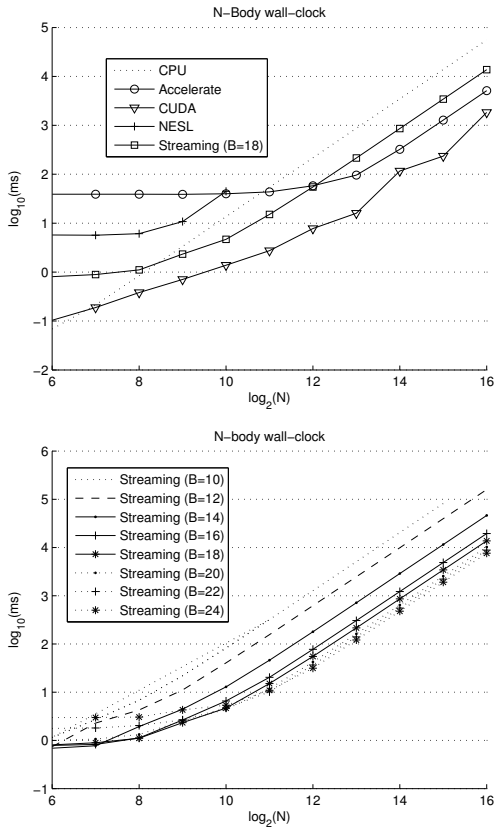

Figure 6. Benchmark results of the sum of N-Body problem, with the same conventions as in Figure 4.

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 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.

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 [11]. 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.

Acknowledgments

The authors want to thank the FHPC'13 reviewers for their helpful and insightful comments. 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.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Sixth Workshop on Declarative Aspects of Multicore Programming, DAMP'11*, pages 3–14, Austin, Texas, Jan. 2011.

[6] B. Lippmeier, M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and S. L. Peyton Jones. Work efficient higher-order vectorisation. In *International Conference on Functional Programming, ICFP'12*, pages 259–270, Copenhagen, Denmark, Sept. 2012.

[7] L. Nyland, M. Harris, and J. Prins. Chapter 31. Fast N-Body Simulation with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison-Wesley Professional, 2007.

[8] 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. 1995.

[9] D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Fifth IEEE Symposium on Frontiers of Massively Parallel Processing, FRONTIERS'95*, pages 186–193, 1995.

[10] S.-B. Scholz. Single Assignment C: Efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[11] T. 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.

[12] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming, ICFP'08*, pages 253–264, Victoria, BC, Canada, Sept. 2008.

[13] Y. Zhang and F. Mueller. CuNesl: Compiling nested data-parallel languages for SIMT architectures. In *41st International Conference on Parallel Processing, ICPP 2012*, pages 340–349, 2012.