Automatic Hierarchical Parallelization of Linear Recurrences

Sepideh Maleki
Department of Computer Science
Texas State University[1]
San Marcos, TX 78666, USA
s_m390@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX 78666, USA
burtscher@txstate.edu

Abstract

Linear recurrences encompass many fundamental computations including prefix sums and digital filters. Later result values depend on earlier result values in recurrences, making it a challenge to compute them in parallel. We present a new work- and space-efficient algorithm to compute linear recurrences that is amenable to automatic parallelization and suitable for hierarchical massively-parallel architectures such as GPUs. We implemented our approach in a domain-specific code generator that emits optimized CUDA code. Our evaluation shows that, for standard prefix sums and single-stage IIR filters, the generated code reaches the throughput of memory copy for large inputs, which cannot be surpassed. On higher-order prefix sums, it performs nearly as well as the fastest handwritten code from the literature. On tuple-based prefix sums and digital filters, our automatically parallelized code outperforms the fastest prior implementations.

CCS Concepts • **Computing methodologies** → Concurrent algorithms; Massively parallel algorithms

Keywords Linear recurrences; prefix sums; recursive filters; automatic parallelization; code optimization

ACM Reference format:
Sepideh Maleki and Martin Burtscher. 2018. Automatic Hierarchical Parallelization of Linear Recurrences. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18), March 24–28,* Williamsburg, VA, USA. ACM, New York, NY. 11 pages.
DOI: https://doi.org/10.1145/3173162.3173168

1 Introduction

Many important algorithms are instances of linear recurrences. Prominent examples include prefix sums and digital filters. Prefix sums are a key primitive that can be used to parallelize computations such as sorting, stream compaction, polynomial evaluation, histograms, and lexical analysis [3]. Infinite Impulse Response (IIR) filters, also known as recursive filters, are fundamental algorithms in digital signal processing. They are, for example, used for DC removal, noise suppression, wave shaping, and smoothing of discrete-time signals in telecommunication and audio applications [22]. Moreover, linear recurrences are important in economics, data compression, biology, pseudo random-number generation, mathematics, complexity analysis, and finance.

A recurrence transforms a sequence of input values $x_0, \ldots, x_{n-1}$ into an output sequence $y_0, \ldots, y_{n-1}$ of the same length. This paper focuses on order-k homogeneous linear recurrences with constant coefficients of the form

$$y_i = a_0x_i+a_{-1}x_{i-1}+\ldots+a_{-p}x_{i-p} + b_{-1}y_{i-1}+b_{-2}y_{i-2}+\ldots+b_{-k}y_{i-k}, \quad (1)$$

where $x_j = 0$, $y_j = 0$, $\forall j < 0$. Equation (1) is called the recursion equation. We refer to the a_j constants as the non-recursion (feed-forward) and the b_j constants as the recursion (feedback) coefficients. To simplify the notation and improve the readability, we express recurrences in the following signature format, where the a_j and b_j coefficients are separated by a colon.

$$(a_0, a_{-1}, \ldots, a_{-p}: b_{-1}, b_{-2}, \ldots, b_{-k})$$

If all the a_j are zero, the output sequence is all zeros and independent of the input values. Hence, we only consider cases where $a_{-p} \neq 0$ for some $p \geq 0$. If all the b_j are zero, the recurrence becomes a map operation that can be computed in an embarrassingly parallel fashion. Thus, we are only interested in cases where $b_{-k} \neq 0$ for some $k \geq 1$. The largest k for which $b_{-k} \neq 0$ determines the order of the recurrence. We use k and the term "order" interchangeably.

Table 1. Signatures of a Few Linear Recurrences

Signature	Computation
(1: 1)	prefix sum
(1: 0, 1)	2-tuple prefix sum
(1: 0, 0, 1)	3-tuple prefix sum
(1: 2, -1)	2nd-order prefix sum
(1: 3, -3, 1)	3rd-order prefix sum
(0.2: 0.8)	a 1-stage low-pass filter
(0.04: 1.6, -0.64)	a 2-stage low-pass filter
(0.008: 2.4, -1.92, 0.512)	a 3-stage low-pass filter
(0.9, -0.9: 0.8)	a 1-stage high-pass filter
(0.81, -1.62, 0.81: 1.6, -0.64)	a 2-stage high-pass filter
(0.73, -2.19, 2.19, -0.73: 2.4, -1.9, 0.5)	a 3-stage high-pass filter

Table 1 lists a few linear recurrences expressed using our signature notation. The coefficients of some of the digital filters are truncated for improved readability. The signature (1: 1) represents the standard prefix-sum computation. Tuple-based prefix sums,

[1] Now at the University of Texas at Austin.

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173168

i.e., prefix sums over tuples rather than scalar values, have b_{-s} set to one and the remaining b_j coefficients to zero, where s is the tuple size. In higher-order prefix sums, i.e., prefix sums of prefix sums, the signature follows the binomial coefficients except b_j is negative for all even j. These signature coefficients can be obtained using the z-transform [16]. The formulas for calculating the filter coefficients listed in Table 1 are given by Smith [22].

Recursion equation (1) can always trivially be separated into the following two recurrences.

$$t_i = a_0x_i + a_{-1}x_{i-1} + \ldots + a_{-p}x_{i-p} \quad (2)$$

$$y_i = t_i + b_{-1}y_{i-1} + b_{-2}y_{i-2} + \ldots + b_{-k}y_{i-k} \quad (3)$$

The signature of the map operation (2) producing the intermediate sequence t_i is $(a_0, a_{-1}, \ldots, a_{-p}: 0)$, and the signature of the recurrence (3) producing the final sequence y_i, assuming t_i as input, is $(1: b_{-1}, b_{-2}, \ldots, b_{-k})$. The map operation is a non-recursive Finite Impulse Response (FIR) filter. This two-stage formulation is important because it allows to compute the overall recurrence by first computing the intermediate sequence t_i in an embarrassingly parallel manner, which eliminates the a_j coefficients and makes the remaining recurrence (3) easier to handle [15]. The rest of this paper focuses on how to automatically parallelize these remaining recurrences, particularly for hierarchical architectures with multiple levels of hardware parallelism. For example, GPUs expose a fine granularity (warps) with very fast communication, a medium granularity (thread blocks) with relatively fast communication, and a coarse granularity (grid) with slow communication. By "recurrence", we henceforth mean recurrences of type (3).

This paper makes the following main contributions.

- It presents a new algorithm for computing linear homogeneous 1D recurrences with constant coefficients that is not similar to any prior recurrence algorithm.
- It describes a hierarchical approach to automatically parallelize such recurrences based on n-nacci numbers.
- It introduces domain-specific optimizations that result in the generated code outperforming the fastest alternatives from the literature in many cases.

The paper makes the following additional contributions.

- It proposes a domain-specific language (our signatures) for compactly expressing recurrences of type (1).
- It presents a simple proof-of-concept compiler called PLR that translates these signatures into CUDA code.

PLR is freely available to the research and education community at http://cs.txstate.edu/~burtscher/research/PLR/.

The rest of this paper is organized as follows. Section 2 explains our algorithm and the hierarchical parallelization approach. Section 3 describes the domain-specific code generator and the optimizations it performs. Section 4 reviews related work. Section 5 presents the evaluation methodology. Section 6 analyzes and discusses the measurement results. Section 7 concludes with a summary and future work.

2 Algorithm

The serial algorithm and code for computing a linear recurrence of the form $(1: b_{-1}, b_{-2}, \ldots, b_{-k})$ are straightforward.

```
for (i = 0; i < n; i++) {
  y[i] = t[i];
  for (j = 1; j <= min(i, k); j++)
    y[i] += b[j] * y[i - j];
}
```

Here, the array elements b[j] hold the coefficients b_{-j} and k denotes the order. This code performs $O(nk)$ computations and $O(nk)$ memory accesses on sequences with n elements. Its space complexity is $O(n+k)$. We assume k to be much smaller than n, which is generally the case in practice.

The reason why recurrences are difficult to parallelize is that each value in the result sequence depends on earlier values from the result sequence, i.e., there is a loop-carried dependency. Hence, the prior values must have been computed before the current value can be computed, making the above implementation inherently sequential.

Since all linear recurrences can be expressed as prefix scans [3], they can be computed in parallel in $O(\log n)$ steps. The total work performed by a prefix-scan-based parallel implementation is close to optimal, i.e., only somewhat worse than the serial algorithm if k is much smaller than n.

Our algorithm is based on a different approach and comprises two phases. The first phase iteratively *merges* pairs of adjacent chunks by correcting the values in the second chunk of each pair using the up to last k values from the first chunk of the pair. The second phase processes the resulting chunks in a pipelined fashion to compute the final solution.

2.1 Phase 1

Phase 1 computes recurrences of type (3) by first subdividing the input sequence into chunks of size one. This yields the correct solution within each chunk as the first output element is always identical to the first input element and each chunk only has one element. Then Phase 1 iteratively merges pairs of adjacent chunks into chunks of twice the size. It does this with the help of precomputed correction factors by which the up to last k elements of the first chunk in a pair have to be multiplied to form the correction terms that need to be added to the elements of the second chunk. We illustrate this process for the arbitrary first-order recurrence (1: d) on the following two contiguous chunks

$$w_0, w_1, \ldots, w_{m-1} \mid w_m, w_{m+1}, \ldots, w_{2m-1},$$

where m is the chunk size and the vertical bar denotes the border between the chunks. To correct the w_m element, we have to add d times the prior element w_{m-1} per the recurrence. Thus, the first correction term is dw_{m-1}. Similarly, to correct the w_{m+1} element, we have to add d times the corrected prior element. However, in the preceding iterations that brought the chunk size up to m, we already added dw_m, so now we only need to add d times the correction term of the prior element. Thus, the second correction term is d^2w_{m-1}. Continuing in this manner, we find the third correction term to be d^3w_{m-1}, and so on. Importantly, they are all the product of a correction *factor* and the *carry* value w_{m-1} from the previous chunk. The carry depends on the input values, but the correction factors do not and can be precomputed for a given recurrence. In this example, the m correction factors are

$$d, d^2, d^3, \ldots, d^m.$$

For reasons that will become clear soon, we extend this sequence of factors by one element on the left, giving

$$1 \mid d, d^2, d^3, \ldots, d^m.$$

So, for any first-order recurrence, the correction factors can be computed by starting with a 1 and applying the recurrence (0: d) to it. Note that this is identical to the original recurrence except the non-recursive term is zero.

Next, we show how to compute the correction factors for the arbitrary second-order recurrence (1: d, e), from which we will then derive the general solution. We start with the following two chunks of m elements from prior iterations:

$$w_0, w_1, \ldots, w_{m-2}, w_{m-1} \mid w_m, w_{m+1}, \ldots, w_{2m-1}.$$

To correct w_m, we have to add d times the previous element w_{m-1} and e times the second-to-previous element w_{m-2}. Hence, the first correction term is $dw_{m-1}+ew_{m-2}$. (Recall that all missing terms are zero. For example, in the first iteration, where $m = 1$, there is no term w_{m-2}.) To correct w_{m+1}, we have to add d times the corrected previous element plus e times the second-to-prior element w_{m-1}. In the earlier iterations, we already added dw_m, so we only need to add d times the correction term of the previous element. Thus, the second correction term is $d(dw_{m-1}+ew_{m-2})+ew_{m-1}$, that is, $(d^2+e)w_{m-1}+ dew_{m-2}$. At this point, we need to add d times the previous correction term plus e times the second-to-previous correction term for all remaining elements. Hence, all correction terms consist of a first correction factor multiplied by w_{m-1} plus a second correction factor multiplied by w_{m-2}. The values w_{m-1} and w_{m-2} are the two carries from the prior chunk. There are two carries because it is a second-order recurrence. The first few correction terms come out to be

$$dw_{m-1}+ew_{m-2}, (d^2+e)w_{m-1}+dew_{m-2}, (d^3+2de)w_{m-1}+ (d^2e+e^2)w_{m-2}, (d^4+3d^2e+e^2)w_{m-1}+(d^3e+2de^2)w_{m-2}, \ldots.$$

Listing just the correction factors for w_{m-1}, we obtain

$$d, d^2+e, d^3+2de, d^4+3d^2e+e^2, \ldots.$$

Similarly, the correction factors for the carry w_{m-2} are

$$e, de, d^2e+e^2, d^3e+2de^2, \ldots.$$

As in the first-order example above, these correction-factor sequences can be computed by changing the non-recursive 1 into a 0 in the underlying recurrence, meaning they can be produced by the recurrence (0: d, e). To correctly compute the first elements of these two sequences, we need to extend both sequences by two elements on the left. In particular, the extended sequences of correction factors turn out to be

$$0, 1 \mid d, d^2+e, d^3+2de, d^4+3d^2e+e^2, \ldots \text{ and}$$

$$1, 0 \mid e, de, d^2e+e^2, d^3e+2de^2, \ldots.$$

Note that there is a 1 in the location of the corresponding carry in the prior chunk and a 0 in the other position.

With this in mind, we can now describe the general approach for computing the correction factors of the arbitrary k^{th}-order recurrence (1: c_{-1}, c_{-2}, ..., c_{-k}). We begin with k elements that are all zero except for a single element that is one. The location of this element is determined by the position of the corresponding carry in the previous chunk. Starting with these k elements, we apply the recurrence (0: c_{-1}, c_{-2}, ..., c_{-k}) to generate the correction factors for that carry. We repeat this procedure for the remaining carries.

The resulting sequences are known as n-nacci numbers [14]. For example, the correction factors of the recurrence (1: 1, 1) are the Fibonacci numbers. Interestingly, there are two Fibonacci sequences, one that is started with "0, 1" and the other with "1, 0". They are not typically distinguished because both sequences are identical except they are shifted by one position relative to the other. The correction factors of the recurrence (1: 1, 1, 1) are the Tribonacci numbers, of which there are three sequences that are started with "0, 0, 1", "0, 1, 0", and "1, 0, 0". Again, the first and the last of these three sequences are shifted by one position relative to each other, but the middle sequence is entirely different (cf. OEIS sequence A001590 vs. A000073). The recurrence (1: 1, 2) results in the so called (1, 2)-Fibonacci sequence. In general, the correction factors of the recurrence (1: c_{-1}, c_{-2}, ..., c_{-k}) are the (c_{-1}, c_{-2}, ..., c_{-k})-nacci numbers, which are the Fibonacci numbers generalized to factors other than one, to more than two terms, and to real numbers. It is the signature notation of the n-nacci numbers that gave us the idea of using a similar notation for compactly expressing linear recurrences.

Since the correction factors can be precomputed, the amount of work at runtime is k multiplications and k additions to correct an element, which is $O(k)$ work. In each iteration, half of the elements need to be corrected (the elements in the second chunk of each pair). If there are n elements in the input, this is $O(n)$ elements. Hence, the amount of work per iteration is $O(nk)$. Note that each element can be corrected independently and in parallel. As the chunk size doubles in each iteration, we need $O(\log(n))$ iterations to reach a size of n. So the total work is $O(nk \log(n))$.

Phase 1 requires $O(\log(n))$ more work than the serial algorithm. The scan-based approach by Blelloch [3] for parallelizing recurrences, which we call "Scan" in this paper, also takes $O(\log n)$ parallel steps but requires a factor of $O(T_{kk}/k)$ more work than the serial algorithm, where T_{kk} is the time to perform a k by k matrix multiplication. Assuming Strassen's algorithm, $T_{kk} \approx k^{2.8}$ and, thus, $O(T_{kk}/k) \approx O(k^{1.8})$. Hence, Phase 1 is more efficient than Scan when $O(\log(n)) < O(k^{1.8})$. This is the case for sufficiently small n or sufficiently large k.

As neither approach is work efficient, we switch to Phase 2 beyond a constant chunk size of m. Since m is constant, we perform $O(nk \log(m)) = O(nk)$ work using Phase 1 to build chunks of fixed size m, which is work efficient.

The iterative doubling of the chunk size make this phase suitable for architectures with different levels of parallelism as will be explained in Section 3. In particular, we chose m to exploit the hardware parallelism within warps and thread blocks. Phase 2 takes advantage of the remaining hardware parallelism.

2.2 Phase 2

There are two main reasons for using a second phase. First, the Phase 2 algorithm is work efficient. Second, the larger the chunk size is, the more correction factors need to be loaded, which incurs overhead. To avoid this overhead, Phase 2 processes fixed-size chunks and operates in a pipelined manner, i.e., the processing of the chunks is partially overlapped.

This pipelined approach is also used by other GPU codes such as in CUB's and SAM's prefix-sum implementations [11, 13]. Once Phase 1 is complete, the last k values of each chunk are written out to make these local carries available to later chunks. Phase 2 reads in the carries from the previous chunk, corrects the values of the current chunk, and emits the now globally correct k carries.

Our implementation uses Merrill and Garland's variable look-back strategy to minimize the waiting for carries [13]. In particular, it does not wait for the global carry values from the prior chunk. Instead, it takes the global carries from the most recent chunk for which they are available as well as the local carries from all chunks that follow, which become available sooner. Based on these values, it computes the global carries for the current chunk using precomputed correction factors akin to Phase 1. CUB and SAM only directly support recurrences whose correction factors are all 1, so they do not need to explicitly precompute them.

Disregarding any waiting for carries, Phase 2 performs $O(mk)$ work per chunk to correct the values and $O(k)$ work to handle the carries. Moreover, it reads m input values, writes m output values, reads $O(k)$ carries, and writes $O(k)$ carries. Since there are n/m chunks, Phases 1 and 2 perform a total of $O(nk)$ work, i.e., they are work efficient. The two phases together read and write $O(n+nk/m)$ words in main memory, which is $O(nk)$ as m is a constant. More precisely, every input value is read once, every output value is written once, and each chunk writes $2k$ carries plus two flags to

indicate when the carries are ready. Our algorithm's space complexity is O(n+k) since it requires storage for n input values, n output values, c flags, and 2ck carries, where c is the constant maximum pipeline depth. We use c = 32 in our implementation so that the carries can be handled by the 32 threads of a single warp. This space complexity is the same as that of the serial algorithm, meaning our algorithm is space efficient. In contrast, the Scan algorithm is not space efficient. It encodes each value in the sequence by a k by k matrix and a k-element vector, meaning it requires O(nk^2) memory.

2.3 Putting It All Together

The following example illustrates how our algorithm works on the second-order recurrence (1: 2, -1), i.e., the second-order prefix sum, with m = 8 and n = 20. Assume the input is:

3	-4	5	-6	7	-8	9	-10	11	-12	13	-14	15	-16	17	-18	19	-20	21	-22

Based on the serial code from the beginning of Section 2, we can compute the expected output for later verification:

3	2	6	4	9	6	12	8	15	10	18	12	21	14	24	16	27	18	30	20

Before our algorithm can start, k lists of m correction factors need to be precomputed (offline) based on the recurrence (0: 2, -1) and the starting values described in Section 2.1. For example the first list below is started with "0, 1". Each entry in the list is generated by multiplying the prior element by 2, the second to prior element by -1, and adding the two products. This yields the following two lists of eight bold-printed correction factors.

		1	2	3	4	5	6	7	8	
0	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	correction-factor list 1
1	0	**-1**	**-2**	**-3**	**-4**	**-5**	**-6**	**-7**	**-8**	correction-factor list 2

At this point, Phase 1 commences by breaking up the input into chunks of one element each before merging pairs of adjacent chunks into successively larger chunks.

The first iteration corrects every second value (aka chunk) by adding 2 times the value of the previous chunk. The "2" is the first correction factor from the first list above. Since there is only one element per chunk, no further work is needed. This yields the following sequence, in which the shaded values are the values that have been corrected.

3	2	5	4	7	6	9	8	11	10	13	12	15	14	17	16	19	18	21	20

In the second iteration, Phase 1 merges every second pair (shaded below) with the preceding pair. The first value is corrected by adding 2 times the last value of the previous pair and -1 times the second-to-last value (the first factors from the two lists above). The second value is corrected by adding 3 times the last value of the previous pair and -2 times the second-to-last value (the second factors from the two lists).

3	2	6	4	7	6	14	12	11	10	22	20	15	14	30	28	19	18	38	36

In the third iteration, every second chunk of four values (shaded below) is merged with the preceding chunk using the first four correction factors from the two lists, i.e., again the i^{th} value in the chunk is corrected with the i^{th} factor in each list. This yields chunks of size m = 8, at which point Phase 1 terminates with the following partial result.

3	2	6	4	9	6	12	8	11	10	22	20	33	30	44	40	19	18	38	36

Note that each iteration not only doubles the chunk size but also the correct and final values at the beginning of the sequence. In other words, after iteration s, the first 2^s elements are correct.

Phase 2 corrects later chunks of size m based on the last two carry values from the previous chunk. It does this in the same manner as Phase 1, except it does not iteratively double the chunk size, which is why it never needs more than m correction factors. Instead, it sequentially proceeds through the chunks. For example, the last element of the second chunk is 40 plus 9 (the last correction factor in list 1) times 8 (the last element of the prior chunk) plus -8 (the last correction factor in list 2) times 12 (the second-to-last element of the prior chunk), which is 40 + 9 · 8 − 8 · 12 = 16. Once the (carry) elements of a chunk have been corrected, the correction of the next chunk can begin. This yields the final result.

3	2	6	4	9	6	12	8	15	10	18	12	21	14	24	16	27	18	30	20

To speed up Phase 2, we pipeline its operation. This is possible because the local carries, which are the last k values of each chunk after Phase 1, are produced earlier than the global carries, which are the last k values of each chunk after Phase 2. Any missing global carries can be precomputed based on the global carries from several chunks ago and the intervening chunks' local carries. For instance, we can precompute the global carries of the third chunk in the above example, which are 24 and 16, based on the global carries from the first chunk (12 and 8) and the local carries from the second chunk (44 and 40) even before the global carries of the second chunk become available. In particular, the global carries of the third chunk are the local carries of the second chunk corrected using the global carries of the first chunk and the correction factors from the two lists above. In our example, 24 = 44 + **8** · 8 + **-7** · 12 and 16 = 40 + **9** · 8 + **-8** · 12, where the bold-printed values are the correction factors and the remaining values are the various carries. This calculation requires O(ck^2) operations, where c is the distance in chunks to the most recent chunk for which global carries are available, i.e., the degree of pipelining. Since the full processing of a chunk takes O(mk) operations, this carry correction is faster as long as O(ck^2) is smaller than O(mk). In other words, the product ck must be sufficiently smaller than m, which is the case in all examples studied in this paper, where k < 4, $c \le 32$, and $m \ge 1024$. In particular, c is typically much smaller than 32 as our code automatically looks for the most recent available global carries, thus dynamically minimizing c. As a consequence, the carry precomputation can be performed while Phase 2 is working on the second chunk but in a fraction of the time, thus enabling Phase 2 to start processing the third chunk before the second chunk is done, which results in a pipelined operation of Phase 2.

3 Implementation

To validate our algorithm and demonstrate the feasibility of our automatic parallelization approach, we wrote a domain-specific compiler, which we call PLR for Parallelized Linear Recurrences. It reads in a recurrence of type (1) in signature format, i.e., two lists of coefficients separated by a colon, which can be thought of as a domain-specific language. It then translates this recurrence into CUDA code. Upon execution, the code computes the recurrence on the given input sequence of values.

CUDA exposes three levels of hardware parallelism. The first level is the warps, which are sets of 32 threads that execute in lockstep. The threads within a warp can exchange data using shuffle instructions without explicit synchronization. The second level is the thread blocks, which hold up to 1024 threads. The threads in a block can exchange data via a software-controlled cache called "shared memory". The third level is the device, which contains a number of streaming multiprocessors (SMs). Thread blocks

are automatically assigned to the SMs until all blocks have executed. Threads from different blocks can only exchange data via global/main memory, which is backed by a shared L2 cache.

PLR supports sequences of any length up to 4 GB. It supports integer and floating-point signatures, including all forms of prefix sums and 1D digital filters. The code it generates requires at least compute capability 3.0, i.e., it works on the several most recent GPU generations from NVIDIA. Such GPUs typically have 65,536 registers per SM and support a maximum thread-block size of 1024 threads. Based on these hardware parameters, PLR sets the chunk size m for each thread block to $1024x$, where x is the number of values each thread has to process. x is the smallest integer for which $x \cdot 1024 \cdot T > n$, where n is the input size and T denotes the number of thread blocks the GPU can simultaneously process. Moreover, $x \leq 9$ for floating-point signatures and $x \leq 11$ for integer signatures. PLR allocates 32 registers per thread for floating-point signatures as well as for integer signatures that only contain ones and zeros (such as those of normal and tuple-based prefix sums). For more complex integer signatures, it allocates 64 registers per thread. These crude heuristics can obviously be improved. For example, most of the recurrences we tested yield higher performance for other values of m and/or x. SAM uses an auto-tuner to find the best value of x for different input sizes [11]. Optimizing these parameters in PLR is left for future work.

The code-generation process starts by parsing the signature and performing some simple checks. For instance, the last non-recursive and the last recursive coefficients must not be zero. Then PLR begins emitting the CUDA program, which consists of the following code sections.

1) The first section includes k constant arrays of size m that are initialized with the correction factors, where k is the order of the recurrence and m is the chunk size at which Phase 1 terminates. The correction factors are precomputed based on the n-nacci sequences described above. They are needed in Phase 1, which iteratively merges chunks until they contain m elements. This procedure requires progressively longer lists of correction factors. However, the longest list contains all needed shorter lists. Moreover, it also contains the corrections factors for Phase 2. Thus, only a single array holding m factors is emitted per carry.

2) The second code section is the beginning of the recurrence kernel. Each thread block first atomically increments a counter to determine for which chunk it is responsible. Then it reads the m input values of that chunk.

3) The following code section performs the map operation (2) to eliminate the non-recursive coefficients.

4) The next section contains the first few unrolled iterations of Phase 1. They are implemented with shuffle instructions to bring the chunk size to the warp size. The rest of Phase 1 is done within a thread block but across warps, so shared memory is used to exchange data. This is an example of how our hierarchical approach makes it possible to exploit the features available at different levels of hardware parallelism. This code extensively uses the correction-factor arrays.

5) As soon as the local carries (the last k values of the chunk) are ready, they are written to global memory. Then a memory fence is executed before a flag is set to indicate that these carries are available. Unless this is the first chunk of the input, the Phase 2 code starts at this point by reading the prior ready flags. It busy waits until there is at least one global set of carries ready within a distance of 32 and all following local carries are ready as well.

6) The next code section reads the k global carries and all later local carries, if any. It then corrects those local carries using the global carries and k correction factors (cf. Section 2.3). This yields the carries needed to correct the m values of the current chunk to produce the global result. The last k values are written to global memory and the corresponding flag is set to indicate that the global carries are ready.

7) Lastly, the m result values are written out.

8) Multiple kernels are generated in the above manner for various values of x. For testing, PLR also emits a main function that calls the appropriate kernel, measures its performance, and verifies the output by comparing it to the result of the serial code.

The entire code generation, which runs serially on one CPU thread, takes only roughly 10 ms on our system (cf. Section 5). It is very fast because the m correction factors are computed using the n-nacci approach rather than by solving the equations we initially used to derive them.

3.1 Optimizations

PLR performs several optimizations to boost the performance of the emitted code. It uses shuffle instructions whenever possible to quickly exchange data. If that does not work, it resorts to shared memory to exchange data. It merely uses main memory for exchanging the local and global carries. PLR only emits correction code for existing terms and suppresses the remaining terms. This is useful in the first few iterations in higher-order recurrences. However, the most important optimizations pertain to the correction factors and are explained next. As far as we know, these optimizations are new and have never before been described.

For each array of correction factors, PLR allocates a buffer in shared memory to cache up to the first 1024 elements. Any correction code that needs one of these elements obtains it from the shared memory. Only the remaining elements are read from main memory. This optimization is particularly useful as the merging in Phase 1 starts with small chunks that progressively get larger. Thus, the first factors are accessed the most often, and all corrections factors can be obtained from the shared memory until the chunk size exceeds 1024.

PLR analyzes the correction factors and emits specialized code when possible. If it finds that all elements are identical within a correction-factor array, the array is suppressed and its accesses are replaced by the appropriate constant. This is helpful for the standard prefix sum. If all array elements are either zero or one, the code generator emits code to conditionally add the correction terms rather than multiplying them by the factors. This helps the tuple-based prefix sums. If the correction factors repeat, only the first "repetition" is emitted.

The most effective optimization applies to the floating-point correction factors of the recursive filters, which typically approach zero quickly for stable filters. In theory, the impulse response of IIR filters is infinitely long, but it is well known that it tends to decay below the arithmetic precision after a few hundred elements [22]. To speed up this effect, we flush denormal values to zero. PLR then changes the code such that only the first few warps of a thread block run Phase 1 since later warps whose correction factors are all zero do not need to execute.

Other optimizations are possible. For example, we could execute the code that leads to the production of the carry values first so that they can be emitted even sooner. For low orders, we could buffer more than 1024 elements of each correction-factor array in the shared memory. Since the first and last correction-factor arrays always contain the same values except shifted by one position (for $k > 1$), one of these two arrays could be suppressed. These and other optimizations are left for future work.

4 Related Work

Karp et al. [9] published the first work on parallelism in recurrence equations in 1967. Six years later, Stone [23] describes a recursive doubling strategy to parallelize first-order recursive filters, which Kogge and Stone [10] extend to higher-order recursive filters. Over a decade later, Sung and Mitra [24] as well as Blelloch [2] present general techniques for exposing parallelism within 1D recursive filters.

StreamIt is an architecture-independent language and compiler for high-performance streaming applications and the first framework to automate the implementation of recursive filters and the optimization of the code [25]. However, it does not address the computation and parallelization of linear recurrences in general, and it does not support GPUs.

Some of the earliest GPU implementations of recurrences are presented by Hensley et al. [7], Sengupta et al. [20], and Dotsenko et al. [6]. Sengupta et al. [19, 21] are the first to publish a work-efficient GPU implementation of scan operations, which can be used to implement prefix sums and recursive filters [3]. Merry [12] studied several recent GPU implementations of prefix sums. Of the investigated codes, he found the CUB library to provide the best performance [13].

CUB implements a work-efficient, single-pass method with $2n$ data movement, meaning that it reads each input value once and writes each output value once. It performs very well across different GPU types in part because it comprises multiple algorithms. For example, it employs different kernel specializations, grain sizes, local scans, and strategies for rearranging data between threads for each GPU architecture. PLR adopts CUB's variable look-back strategy for propagating the carries to hide the communication latency and pipeline the Phase 2 computation.

For higher-order and tuple-based prefix sums, SAM provides the fastest GPU implementation [11]. It is also a work-efficient, single-pass approach with $2n$ data movement. It runs an autotuner upon installation that determines the optimal number of elements to assign to each thread for different problem sizes and different types of prefix sums.

We compare PLR to CUB and SAM in the result section. Since PLR's code is automatically generated, it can be optimized for each given recurrence. This is difficult to do in the handwritten CUB and SAM implementations, which use a single code base to handle multiple types of recurrences.

The best-performing GPU implementations of linear recursive filters are probably by Nehab et al. [15] and Chaurasia et al. [4]. However, those codes are not communication efficient as they read the input values multiple times. Moreover, their work focuses on 2D image processing rather than 1D signal processing (which is our focus). Nevertheless, we compare to both of these codes in the result section.

Nehab et al. present an algorithmic framework to facilitate the overlapping of the causal, anticausal, row, and column filter processing. This reduces the memory bandwidth as the output of one filter is directly piped into the next filter. Moreover, they exploit the fact that a higher-order filter can be decomposed into an equivalent set of several lower-order filters. They found that applying multiple lower-order filters sometimes results in faster processing than using the single, corresponding higher-order filter.

Chaurasia et al.'s work also targets 2D image processing. They are the first to provide a code generator for automatically synthesizing GPU recurrence code. In this sense, their work is the most closely related to ours. Their domain-specific code generator, which is based on Halide, supports a set of program transformations that can be composed to implement many different filters on CPUs and GPUs. It then performs locality optimizations on the composed filters to minimize memory accesses through interleaving and tiling. Moreover, their code generator supports several heuristics to schedule the tiled code. The programmer specifies the recursive filters and selects the heuristics in form of a short program written in a domain-specific language.

PLR's input is simpler, just a signature, and does not require the user to specify or experiment with tile sizes and heuristics. However, PLR does not support the automatic combination of filters, which has to be done offline using, for example, the z-transform. PLR parallelizes every stage of the computation whereas Chaurasia et al.'s code serially combines the local carries to produce the global carries.

Except for StreamIt and Chaurasia et al.'s work, the implementations described in the above work are all handwritten, which is a difficult, error-prone, and time consuming process, and none of these papers propose an automatic parallelization approach, which is one of the key contributions of our work. Moreover, our algorithm is based on merging and correction factors, making it not similar to any of the previously proposed algorithms.

5 Experimental Methodology

We compare the performance of the code emitted by our PLR code generator to the fastest published codes. On the integer side, these are CUB 1.5.1 [5] and SAM 1.1 [18] for various types of prefix sums. On the floating-point side, these are "Alg3" by Nehab et al. [1] and "Rec" by Chaurasia et al. [17] for recursive filters. Since Alg3 and Rec are designed for 2D image processing, we ran them with square 2D inputs (of a similar total size as our 1D inputs) where the width and height are multiples of 32 to match the warp size. To make the comparison as fair as possible, we disabled the vertical filtering in these codes. Note, however, that Alg3 still filters in both the positive and the negative horizontal direction. We were only able to limit the filtering to one horizontal direction for Rec to make it similar to PLR, which also only filters in one direction. Finally, we compare to the scan-based approach (Scan) described by Blelloch [3], which, like PLR, only requires the signature to specify the desired parallel recurrence computation. He derived a general way to express recurrences in form of k by k matrices and k-element vectors as well as appropriate associative operators that are based on matrix multiplication and vector addition, which allows the computation of arbitrary recurrences in parallel using a standard prefix scan. A prefix scan is similar to a prefix sum except the operator is not addition. For CUB, SAM, Alg3, and Rec, we used publicly available code. For Scan, we implemented the operator ourselves but used CUB to run the actual scan. We compiled all codes with nvcc 7.5 using the "-O3 -arch=sm_52" flags. We measured the GPU memory usage with the NVIDIA Management Library (NVML). To obtain the cache miss results, we used the nvprof profiler.

We present results for a GeForce GTX Titan X GPU, which is based on the Maxwell architecture. It has 3072 processing elements in 24 multiprocessors that can hold the contexts of up to 49,152 threads. Each multiprocessor has 96 kB of shared memory, up to 48 kB of which are accessible from a single thread block. The 24 multiprocessors share a 2 MB L2 cache as well as 12 GB of global memory with a peak bandwidth of 336 GB/s. We use the default clock frequencies of 1.1 GHz for the processing elements and 3.5 GHz for the GDDR5 memory. The GPU is plugged into a 16x PCIe 3.0 slot in a system with dual 10-core Xeon E5-2687W v3 CPUs running at 3.1 GHz. Its memory size is 128 GB, the operating system is Fedora 22, and the CUDA driver is 361.42.

When measuring the performance, we only consider the computation time, excluding the time it takes to transfer the input sequence to the GPU or the result to the CPU. In other words, we assume the data to already be on the GPU from a prior processing step and the result of the recurrence to be needed on the GPU for a later processing step. After each run, we validate the result by comparing it to the serial CPU result. We check the integer results for exact matches. Since floating-point addition and multiplication are not truly associative, the parallel codes produce slightly different results than the serial code for floating-point recurrences. In this case, we make sure the discrepancy is within 10^{-3}.

We evaluate the six codes on the recurrences listed in Table 1 using 32-bit integer and floating-point values. In all cases, we varied the input size from 2^{14} to 2^{30} words in powers of two. None of the tested codes support sizes above 4 GB, that is, inputs with more than 2^{30} words. Since the codes' control-flow and memory-access behavior are independent of the values in the input sequence, any input of the same length and data type will result in the same performance for a given recurrence. Note that PLR supports input sizes that are not powers of two and that powers of two do not result in the highest performance since m is generally not a power of two. We repeated each experiment six times, measured the average runtime of the last five runs, and computed the throughput from it (in words processed per second).

6 Results and Analysis

Our PLR approach supports arbitrary integer and floating-point recurrences. However, we can only show results for a few examples. We selected the recurrences listed in Table 1, which we believe to be representative of important real-world computations. In the following, we first study the performance on integer recurrences using various prefix sums as examples. Then we separately study floating-point recurrences on several digital filters.

6.1 Integer Recurrences

The results in this section are based on 32-bit integer sequences and signatures with integer coefficients. We compare the generated PLR code to CUB and SAM, the two best-performing codes from the literature for prefix sums [11], as well as the general Scan approach. For reference, the memory-copy throughput is also given, which represents an upper bound on the achievable throughput since it just copies the input sequence to the output without any computation.

6.1.1 Standard Prefix Sum

The standard prefix sum is perhaps the most frequently used recurrence in parallel computing. Figure 1 shows the throughput in billions of 32-bit integers processed per second on the Titan X GPU for different sequence lengths.

The performance of CUB, SAM, and PLR is quite similar. PLR is a little slower than the other two codes in the mid-range. SAM is somewhat faster in the low range due to its use of auto-tuning. On long sequences, PLR's throughput is on par with the other two codes and, in some cases, even a little higher than that of SAM. All three codes reach the throughput of memory copy, which means their performance cannot be exceeded by any other code that reads each input value and writes each output value. In this sense, PLR delivers optimal performance for large sequences on prefix sums. The three codes transfer up to 264 GB/s, which is beyond the theoretical memory bandwidth of most CPU systems, implying that the serial code running on a CPU has to be slower.

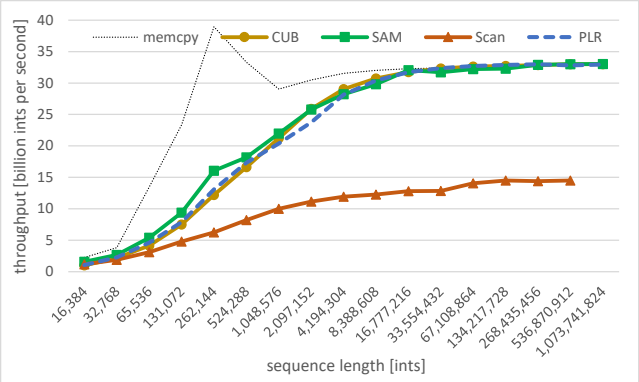

Figure 1. Prefix-sum throughput

The Scan code delivers about half the throughput of the other three approaches because it accesses twice as much memory (a 1x1 matrix and a 1-element vector per sequence element), which is also why it only supports problem sizes up to 2^{29}. Prefix sums of 32-bit floating-point values yield the same performance for all four codes (not shown).

6.1.2 Tuple-based Prefix Sums

Figures 2 and 3 show the 2- and 3-tuple prefix-sum throughput in billions of 32-bit integers processed per second for different sequence lengths.

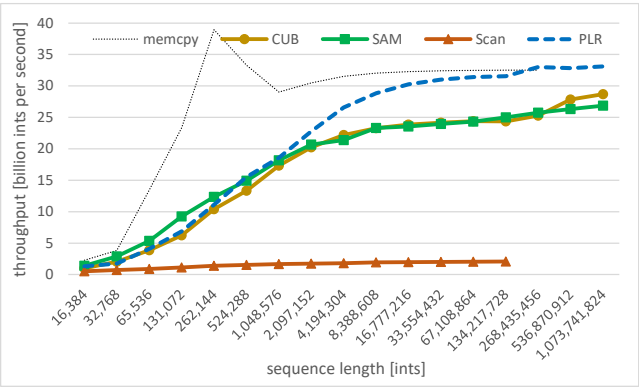

Figure 2. Two-tuple prefix-sum throughput

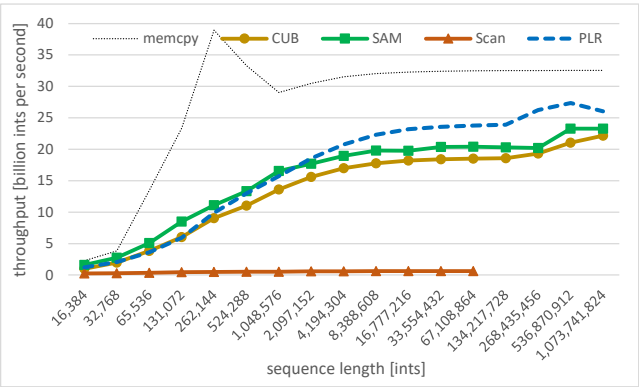

Figure 3. Three-tuple prefix-sum throughput

For tuple-based prefix sums, there is a clear difference in performance between the codes. On small inputs, SAM is again the fastest due to the auto-tuning. In the mid-range, PLR outperforms

CUB and starts to outperform SAM. For long sequences, PLR substantially outperforms the other two codes. On 2-tuples, it is 30% and on 3-tuples 17% faster. The reason for this performance difference is that the three codes implement different approaches. In case of 2-tuples, SAM computes two independent interleaved scalar prefix sums, CUB computes a prefix sum on 2-element vectors, and PLR computes a single scalar second-order recurrence.

The performance advantage of PLR is higher on tuple sizes that are powers of two because they allow for additional code optimizations. In fact, PLR's 4-tuple throughput (not shown) is slightly higher than its 3-tuple throughput. In contrast, CUB's and SAM's throughputs consistently decrease with larger tuple sizes as they use the same code base for different tuple sizes. The Scan throughput is quite low because it requires six and twelve times more memory accesses (a 2x2 matrix with a 2-element vector and a 3x3 matrix with a 3-element vector), respectively, and suffers from correspondingly higher register pressure.

In summary, PLR performs as well as CUB on tuple-based prefix sums on all tested inputs and yields superior performance over CUB and SAM for medium and long sequences. Tuning the m and x parameters might boost PLR's performance on short inputs.

6.1.3 Higher-order prefix sums

Figures 4 and 5 show the 2nd- and 3rd-order prefix-sum throughput, respectively, in billions of 32-bit integers processed per second for different sequence lengths.

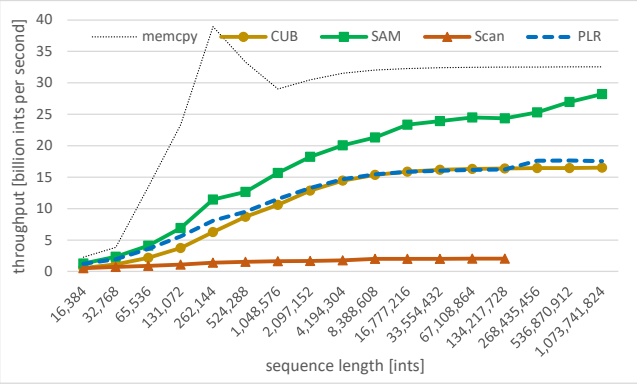

Figure 4. Second-order prefix-sum throughput

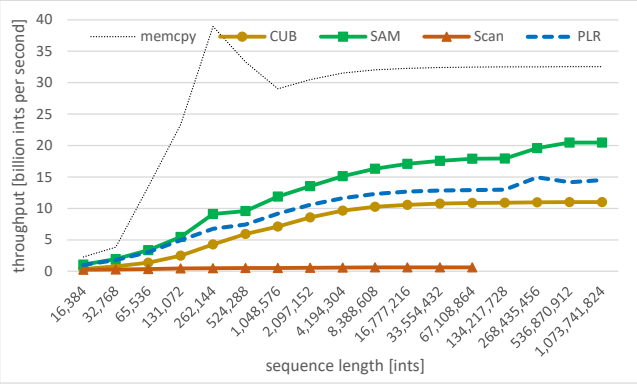

Figure 5. Third-order prefix-sum throughput

On higher-order prefix sums, ignoring Scan, CUB yields the lowest throughput, PLR is in the middle, and SAM the highest, except at the smallest tested problem sizes, where PLR performs on par with SAM. PLR's performance is relatively low because it cannot optimize the resulting correction factors (cf. Section 6.3). On second-order prefix sums, PLR barely outperforms CUB, on third-order prefix sums it significantly outperforms it, and on fourth-order prefix sums (not shown) it outperforms it even more. At the same time, SAM's performance advantage over PLR decreases with increasing order. For order 2, it is 50% faster, for order 3 about 38%, and for order 4 about 33%. Hence, PLR's performance increases relative to both CUB and SAM with higher orders. This is again due to the different approaches used to compute the recurrences. PLR computes a single scalar recurrence whereas CUB and SAM iteratively compute prefix sums of prefix sums. CUB repeats the entire code whereas SAM only repeats the computation but not the reading in and writing out of the values, which is why it outperforms CUB.

For a given order, the performance of Scan on the higher-order prefix sums is identical to that on the tuple-based prefix sums as only the matrix and vector values change but not the number of memory accesses or the operations executed.

In summary, PLR outperforms CUB on higher-order prefix sums but underperforms SAM, at least for small k. Since the shared memory is not fully utilized, buffering more than just the first 1024 correction factors of each array might boost PLR's performance on these and similar recurrences.

6.2 Floating-Point Recurrences

The results in this section are based on 32-bit floating-point sequences and signatures with floating-point coefficients. As mentioned above, prefix sums of 32-bit floating-point values yield the same performance on all tested codes as the integer prefix sums. Hence, we do not show prefix-sum results for floating-point data. Instead, we compare PLR to Alg3 by Nehab et al. [15] and to Rec by Chaurasia et al. [4], the two best-performing codes from the literature for floating-point recurrences, as well as to Scan by Blelloch [3]. Again, the memory-copy throughput is included as an upper bound on the achievable throughput. Note that, unlike Rec and PLR, Alg3 filters in both the positive and negative direction.

6.2.1 Low-pass Recursive Filters

Figures 6, 7, and 8 show the 1-stage, 2-stage, and 3-stage low-pass filter throughputs, respectively, in billions of 32-bit floats processed per second for different sequence lengths. Alg3 only supports inputs up to 2 GB and Rec up to 1 GB. Since Scan requires $O(k^2)$ memory per item, its maximum supported problem size decreases quickly with increasing order.

On the evaluated low-pass filters and on all input sizes, PLR is again much faster than Scan. It is also faster than Alg3, but Alg3 performs two filter operations whereas PLR only performs one. We were unable to turn off the extra filter operation in Alg3 to make the comparison fairer. (When filtering 2D images, Alg3 and Rec perform similarly.) For inputs up to a million elements, Rec performs on par or is faster than PLR. This may be because Rec executes many small filter operations on a square input whereas PLR executes a single long filter operation on a linear input, which likely incurs a higher carry-propagation delay. Nevertheless, PLR is the fastest of the tested codes on the larger inputs.

On the single-stage filter, PLR reaches the throughput of memory copy for large problem sizes, i.e., optimal performance. As we go to higher orders, the throughput of all four codes decreases. PLR's throughput decreases faster than that of Rec and Alg3. For 1 GB inputs, it is 1.90, 1.88, and 1.58 times faster than Rec on the 1-stage, 2-stage, and 3-stage filters, respectively. We surmise this is because a high-order recurrence can be slower to compute than multiple (simpler) low-order recurrences that, together, compute the same result [15]. Since recursive digital filters

above about order ten tend to be unstable [22], low-order filters dominate in practice, where PLR provides the highest throughput.

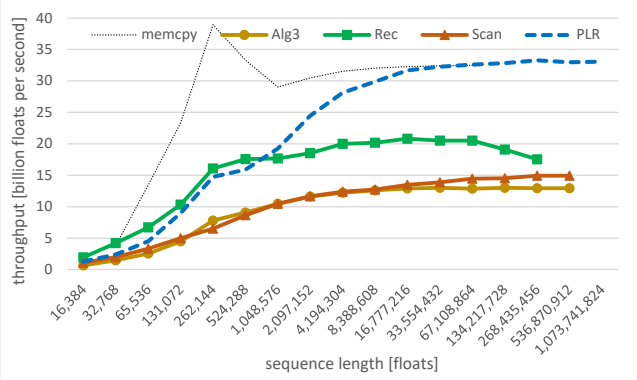

Figure 6. 1-stage low-pass filter throughput

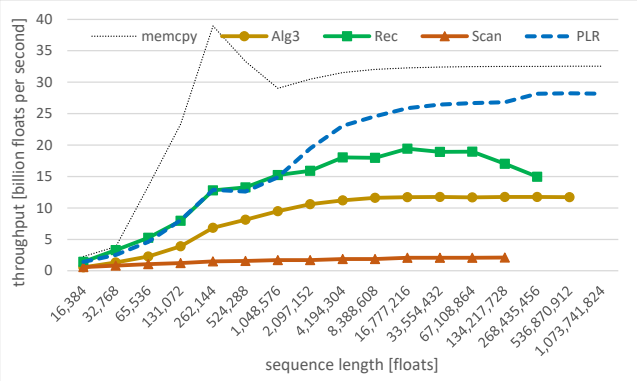

Figure 7. 2-stage low-pass filter throughput

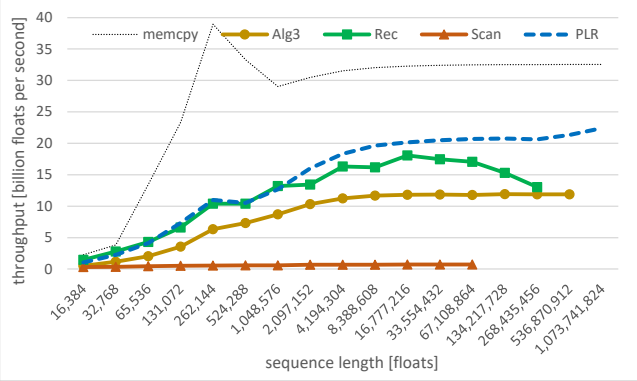

Figure 8. 3-stage low-pass filter throughput

6.2.2 High-pass Recursive Filters

Neither Alg3 nor Rec currently support recursive filters with more than one non-recursive coefficient, which is why they cannot implement the high-pass filters from Table 1. Hence, Figure 9 only shows the PLR 1-stage, 2-stage, and 3-stage high-pass filter throughput in billions of 32-bit floats processed per second for different sequence lengths. We included Scan's 1-stage recursive high-pass filter throughput but not that for more stages since it is already the slowest. Note that our Scan implementation uses the same code as PLR for computing the map operation (2), that is, for processing the non-recursive FIR coefficients.

As expected, the throughputs decrease with increasing order. Moreover, they are lower than the corresponding low-pass filter throughputs because there are additional non-recursion coefficients to handle (cf. Table 1). Since the recursion coefficients are the same, the drop in performance must be due to the additional non-recursion coefficients. Interestingly, this decrease is quite consistent and around 17% for medium to large problem sizes, irrespective of the order. This shows that the map operation (2) to handle the non-recursion coefficients is quite fast compared to the code that handles the recursion coefficients.

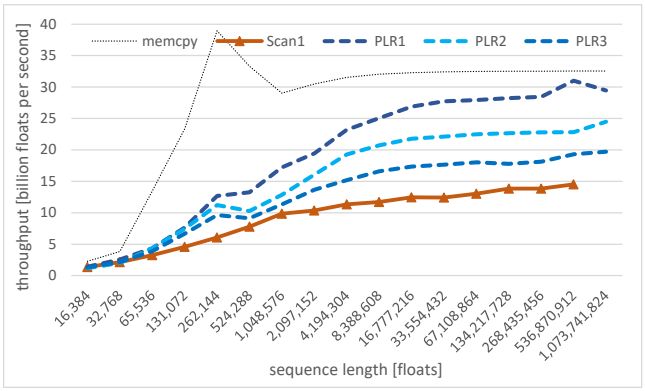

Figure 9. High-pass filter throughput

6.3 Performance Optimizations

Figure 10 combines the PLR throughputs on the largest input of the eleven studied recurrences in a single chart. The left half shows the results for integers and the right half for floats. For each recurrence, the figure includes the throughput when turning off the optimizations pertaining to the correction factors, i.e., when they are always loaded from global memory and no special code is emitted for factors that are constants, only zero or one, repeat, or decay to zero after a certain point.

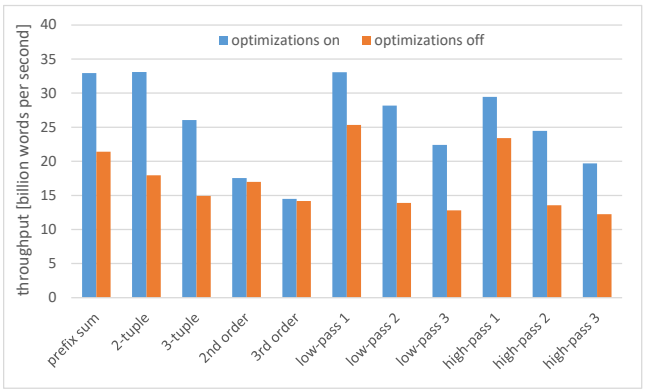

Figure 10. PLR throughput with and without optimizations

The optimizations help in all cases. On the higher-order prefix sums, they improve performance by only 3%, whereas on the two-stage low-pass filter, they more than double the throughput. For the prefix sum and the tuple-based prefix sums, the performance difference is primarily due to treating correction factors of zero and one specially. On the digital filters, the difference is mostly due to suppressing zero factors at the end of the array and buffering factors in shared memory. Except for the buffering, none of these optimizations apply to the higher-order prefix sums, which is probably why PLR is outperformed on this type of recurrence

by pre-existing code. In the remaining cases, the optimization are important as they boost the performance substantially.

6.4 GPU Memory Usage

Table 2 lists the total memory usage on the GPU when processing the largest input that all six recurrence codes support, i.e., 67,108,864 words. The usage only depends on the order of the recurrence but not the coefficients or the data type (int or float), which is why the table only shows three results for each code. The memory consumption on the CPU is not included. For reference, the memory usage of the trivial memory-copy code is also listed. It contains no user-written kernel code but only CUDA library calls to allocate and free the input and output buffers, calls to copy the input data to the GPU and the output data back to the CPU, and the timed GPU-to-GPU memory-copy call.

Table 2. Total GPU Memory Usage in Megabytes

	PLR	CUB	SAM	Scan	Alg3	Rec	memcpy
order 1	623.5	623.5	622.5	1,135.5	895.8	638.5	621.5
order 2	623.5	623.5	622.5	3,188.8	911.8	654.5	
order 3	624.5	623.5	622.5	6,278.9	927.8	670.5	

In addition to the 512 MB of combined storage for the input and output arrays, even the memory-copy code allocates an extra 109.5 MB. Interestingly, SAM requires only one more megabyte, CUB two more megabytes, and PLR between two and three more megabytes, i.e., less than half a percent. This additional memory holds the code and the auxiliary arrays for the carries and flags. Rec requires between 17 MB and 49 MB more memory and Alg3 between 274 MB and 306 MB. It appears that both of these codes allocate significant extra memory whose amount increases for higher orders. Scan needs by far the most memory because it encodes each value using a k by k matrix and a k-element vector. Just to store the input and output, it requires 1024 MB for first-order, 3072 MB for second-order, and 6144 MB for third-order recurrences. In summary, PLR (like CUB and SAM) is nearly as memory efficient as the memory-copy code.

6.5 L2 Cache Misses

Table 3 lists the L2-cache read misses incurred on the GPU multiplied by the block size of 32 bytes when processing the 67,108,864-word input. We combined the miss counts from both slices of the L2 cache, which contributed nearly equal counts. Again, the measurements are largely unaffected by the coefficients and the data type, so we only show results for three different orders for each code. The cache misses on the CPU are not included. We cannot show cache misses for the memory-copy code because it does not incur any, i.e., it does not appear to use the L2 cache.

Table 3. L2 Cache Read Misses Converted into Megabytes

	PLR	CUB	SAM	Scan	Alg3	Rec
order 1	256.1	256.5	256.2	512.3	550.6	528.3
order 2	256.2	256.1	256.6	1,537.1	591.3	545.3
order 3	256.4	256.2	256.8	3,074.1	632.0	562.5

Reading the input array results in a transfer of 256 MB of data due to cold misses. As the table shows, PLR, CUB, and SAM only incur a tiny amount of additional L2-cache read misses (less than one megabyte or 0.3%). In other words, these codes either have good locality or do not read much additional data. In either case, they are not memory bound aside from reading the input once. Scan accesses many more memory locations because of its inefficient data representation. When accounting for the two, six, and twelve times higher cold misses, we find that it only accesses an additional 0.3 to 2.1 megabytes. Finally, the results in Table 3 make it evident that Alg3 and Rec are not communication efficient as they read the input data twice. For the given problem size, which greatly exceeds the capacity of the 2 MB L2 cache, the second access to the input again has to fetch the data from main memory. This explains why PLR outperforms these codes and why it starts outperforming Rec at a size of one million entries, which is the smallest problem size that exceeds the L2 capacity. Moreover, both Alg3 and Rec incur cache misses beyond reading the input twice, which is consistent with the additional memory they allocate as mentioned in the previous subsection. In summary, PLR (as well as CUB, SAM, and even Scan) are communication efficient in that they essentially only incur cold read misses in the L2 for accessing the input data.

7 Summary and Conclusion

Recurrences convert a sequence of input values into a sequence of output values. They are non-trivial to parallelize because later output values are data dependent on earlier output values. This paper presents a new algorithm for computing linear recurrences that is work- and space-efficient as well as general. The algorithm is based on iteratively merging partial solutions using n-nacci numbers. Moreover, the paper describes a hierarchical parallelization approach that is a good fit for architectures like GPUs with multiple levels of hardware parallelism. It also introduces several domain-specific optimizations that result in some of the fastest recurrence computations on a GPU.

We implemented our algorithm, parallelization approach, and code optimizations in a proof-of-concept compiler called PLR that translates the signature notation of a recurrence into CUDA code. It supports both integer and floating-point recurrences. The performance of the emitted code is optimal for large inputs on low orders in the sense that it reaches the throughput of memory copy, which cannot be exceeded. On higher-order tuple-based prefix sums and 1D recursive digital filters, our automatically generated code outperforms the fastest codes from the literature, many of which are handwritten and support fewer recurrence types.

Whereas we evaluate our approach on a GPU, the presented algorithm, parallelization technique, and even most of the code optimizations are not GPU specific but apply equally to CPUs, DSPs, FPGAs, and other parallel computing devices. Moreover, whereas we demonstrate our approach in the context of a standalone tool, it could equally be part of a full-fledged (C/C++) compiler that is invoked either via an intrinsic or to augment an existing loop-nest transformation engine that automatically parallelizes code (such as Graphite in gcc).

There are several avenues for future work, especially regarding PLR's code optimizer. For instance, we could add better heuristics to boost the performance on small inputs. To improve the performance on higher-order prefix sums, we could compute and emit the carries sooner, support inputs that consist of multiple signatures, and buffer more correction factors in the shared memory. We could also support operators other than addition, multiple dimensions, and non-GPU systems.

In summary, our approach makes it possible to automatically parallelize arbitrary-order homogeneous linear recurrences with constant coefficients and delivers heretofore unreached GPU performance for many important instances, especially compared to Scan, the only parallel implementation we tested that supports all the recurrences that PLR does. Since recursive filters are widely

used in telecommunication and prefix sums are widely used in parallel programming, being able to automatically parallelize and optimize them, including variations for which no efficient parallel code exists to date, may prove useful in practice. Looking forward, we believe our hierarchical approach is well suited for future systems that will likely be even more parallel.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, which greatly helped improve our paper. We thank Diego Nehab and André Maximo for their help with the Alg3 code as well as Gaurav Chaurasia for his help with the Rec code. We thank Sahar Azimi for her help with writing and running the Scan code. This work was supported in part by the National Science Foundation under award #1406304 and by equipment donations from Nvidia.

References

[1] Alg3: https://github.com/andmax/gpufilter/, accessed 8/8/2017.
[2] G.E. Blelloch. "Scans as Primitive Parallel Operations." *IEEE Transactions on Computers*, 38(11):1526-1538. 1989.
[3] G.E. Blelloch. "Prefix Sums and Their Applications." In John H. Reif (Ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1990.
[4] G. Chaurasia, J. Ragan-Kelley, S. Paris, G. Drettakis, and F. Durand. "Compiling High Performance Recursive Filters." In Proceedings of the *7th Conference on High-Performance Graphics*, pp. 85-94. 2015.
[5] CUB: https://nvlabs.github.io/cub/, accessed 8/8/2017.
[6] Y. Dotsenko, N.K. Govindaraju, P.P. Sloan, C. Boyd, and J. Manferdelli. "Fast Scan Algorithms on Graphics Processors." In Proceedings of the *22nd Annual International Conference on Supercomputing*, pp. 205-213. 2008.
[7] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. "Fast Summed-Area Table Generation and its Applications." *Computer Graphics Forum*, 24(3):547-555. 2005.
[8] W.D. Hillis and G.L. Steele. "Data Parallel Algorithms." *Communications of the ACM*, 29(12): 1170-1183. 1986.
[9] R.M. Karp, R.E. Miller, and S. Winograd. "The Organization of Computations for Uniform Recurrence equations." *Journal of the ACM*, 14:3, pp. 563-590. 1967.
[10] P.M. Kogge and H.S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations." *IEEE Transactions on Computers*, 22(8):786-793. 1973.
[11] S. Maleki, A. Yang, and M. Burtscher. "Higher-Order and Tuple-Based Massively-Parallel Prefix Sums." In Proceedings of the *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 539-552. 2016.
[12] B. Merry. "A Performance Comparison of Sort and Scan Libraries for GPUs." *World Scientific Publishing Company*. 2014.
[13] D. Merrill and M. Garland. "Single-Pass Parallel Prefix Scan with Decoupled Look-back." *NVIDIA Technical Report* NVR-2016-002. 2016.
[14] n-nacci numbers: https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers, accessed 8/8/2017.
[15] D. Nehab, A. Maximo, R.S. Lima, and H. Hoppe. "GPU-Efficient Recursive Filtering and Summed-Area Tables." In Proceedings of the *SIGGRAPH Asia Conference*, pp. 176:1-176:12. 2011.
[16] A.V. Oppenheim and R.W. Schafer. "Discrete-Time Signal Processing." 3rd Edition. Prentice Hall. 2009.
[17] Rec: https://github.com/mit-gfx/recfilter, accessed 8/8/2017.
[18] SAM: http://cs.txstate.edu/~burtscher/research/SAM/, accessed 8/8/2017.
[19] S. Sengupta, A.E. Lefohn, and J.D. Owens. "A Work-Efficient Step-Efficient Prefix Sum Algorithm." In Proceedings of the *Workshop on Edge Computing Using New Commodity Architectures*, pp. 26-27. 2006.
[20] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. "Scan Primitives for GPU Computing." In Proceedings of *Graphics Hardware*, pp. 97-106. 2007.
[21] S. Sengupta, M. Harris, and M. Garland. "Efficient Parallel Scan Algorithms for GPUs." NVIDIA. 2008 - gpucomputing.net.
[22] S.W. Smith. "Digital Signal Processing: A Practical Guide for Engineers and Scientists." Newnes, 2002. ISBN 0-7506-7444-X.
[23] H.S. Stone. "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations." *Journal of the ACM*, 20(1):27-38. 1973.
[24] W. Sung and S. Mitra. "Efficient Multi-Processor Implementation of Recursive Digital Filters. In Proceedings of the *IEEE Conference on Acoustics, Speech and Signal Processing*, 11:257-260. 1986.
[25] W. Thies, M. Karczmarek, and S.P. Amarasinghe. "StreamIt: A Language for Streaming Applications." In Proceedings of the *11th International Conference on Compiler Construction*, pp. 179-196. 2002.