Parallelization of Linear Recurrences on GPUs

This project refers to implementing a work-efficient parallel algorithm for linear recurrences, a.k.a.,infinite-impulse response filters (IIRs) or recursive filters.

1. Preliminaries

To keep it simple, a linear recurrence of length `n` has the form:

```
y[0] = a[0], …, y[k-1] = a[k-1]
y[i] = a[i] + b[1]*y[i-1] + … + b[k]*y[i-k], for i = k .. n
```

and can be implemented in **`O(k*n)`** time with the sequential implementation below:

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

In “Prefix Sums and Their Applications”, Blelloch presents an elegant data-parallel implementation based on scan with k-by-k matrix multiplication as an operator, but this is not work-efficient in the general case, i.e., it is work efficient only if you consider `k` to be a constant. The simple idea behind the scan-based implementation is to trivially fill the system of k equations:

```
y[i]   = a[i] + b[1]*y[i-1] + … + b[k-1]*y[i-k+1] + b[k]*y[i-k]
y[i-1] = 0    +    1*y[i-1] + … +      0*y[i-k+1] +    0*y[i-k]
…
y[i-k+1]=0    +    0*y[i-1] + … +      1*y[i-k+1] +    0*y[i-k]
```

which essentially means that the k-vector Y[i] = `(y[i],…,y[i-k+1])` can be obtained from vector Y[i-1] = `(y[i-1],…,y[i-k])` by multiplying the latter with the constant matrix `B`:

```
    b[1], …, b[k-1], b[k]
    1,    …,      0,    0
B = …
    0,    …,      1,    0
```

and adding the vector A[i] = `(a[i], 0, …, 0)` to it, i.e., Y[i] = A[i] + B * Y[i-1]

It follows that we can compute the recurrence in parallel by means of a scan with linear-function composition as operator: denoting by `f_i(x) = a[i] + B[i]*x`, we then have that `f_1 o f2 (x) = f_1(f_2(x)) = a_1 + B_1*a_2 + B_1 * B_2 * x`

which essentially means that the operator for scan is:
`(a1, B1) o (a2, B2) = (a1 + B1*a2, B1 * B2)`
where `B1*a2` and `B1*B2` correspond to matrix-vector and matrix-matrix multiplications, and + denotes vector addition. However, this is not work efficient because matrix-vector multiplication is **`O(k^2)`** and matrix-matrix multiplication is **`O(k^3)`**, which means that the computation for the whole recurrence would be **`O(n * k^3)`** instead of **`O(n * k)`**.

2. Project's taks:

This project is about reproducing the implementation and, as much as possible, the experimental evaluation, of ***Maleki and Burtscher's paper "Automatic Hierarchical Parallelization of Linear Recurrences", published in ASPLOS'18***, which is available in the project folder.

This is a new project that has not been tried before, hence we are going to be lenient in the evaluation of your work. The focus should be on providing a parallel implementation that is provably work efficient (you should argue why it is so). It is acceptable if the constants are suboptimial: for example the paper uses the idea from the single-pass scan, in that it combines parallel execution with pipelining so that everything is executed inside one kernel, hence each input/output element is read/written from/to global memory exactly once.

Since the single-pass scan is the subject of another project, you do not have to implement this optimization: it is fine to have several Cuda kernels implementing the paper's algorithm.

The report is expected to

1. present the algorithm at a high-level in an understandable manner;
2. detail on your low-level Cuda implementation, including snippets of essential code which are also explained in text;
3. reason about the work-depth efficiency of your implementation. Then move on and discuss constants, i.e., performance bottlenecks, and avenues for improving them (e.g., pipelining in the fashion of single-pass scan)
4. provide an experimental evaluation that demonstrates that your implementation is work efficient, and that compares with other baseline implementations. Baselines can be, for example, a scan-based implementation in Futhark that you write yourselves, or, if Cub supports computation of linear recurrences, you may use that, or you can submit code to be compiled with the approach of the paper (they seem to have an online compiler that can be accessed on the web to compile such recurrences to Cuda code). Essentially, in the remaining time, try to reproduce as much as possible the experimental results of the paper. It would be deeply appreciated if you also increase the magnitude of "k", i.e., the maximal evaluated k is 3 in the paper; it would be interested how the runtime changes for bigger k, such as 7.