Automatic Generation of 1D Recursive Filter Code for GPUs

Sepideh Maleki and Martin Burtscher

TEXAS STATE UNIVERSITY
The rising STAR of Texas

ECL Efficient Computing Laboratory
Based on Fibonacci Sequences

- **Fibonacci numbers:** 0, 1, 2, 3, 5, 8, 13, 21, ...
  - Sum of previous two values \( F_i = F_{i-1} + F_{i-2} \)

- **Tribonacci numbers:** 0, 0, 1, 1, 2, 4, 7, 13, 24, ...
  - Sum of prior three values \( F_i = F_{i-1} + F_{i-2} + F_{i-3} \)

- **(2, -3, 1)-Fibonacci numbers:** 0, 0, 1, 2, 1, -3, -7, -4, ...
  - Weighted sum of prior values \( F_i = 2F_{i-1} - 3F_{i-2} + F_{i-3} \)

- **\((w_1,\ldots,w_k)\)-Fibonacci numbers:** 0, ..., 0, 1, \( w_1, w_1^2+w_2, \ldots \)
  - Weighted sum of prior \( k \) values with \( w_j \in \mathbb{R} \) \( F_i = w_1F_{i-1} + w_2F_{i-2} + \ldots + w_kF_{i-k} \), called \( k \)-nacci numbers
Linear Recurrences

- Transform input sequence into output sequence

\[ x_0, \ldots, x_{n-1} \rightarrow y_0, \ldots, y_{n-1} \]

- Our focus is on order-\( k \) homogeneous linear recurrences with constant coefficients

\[
y_i = a_0x_i+a_{-1}x_{i-1}+\ldots+a_{-p}x_{i-p} + b_{-1}y_{i-1}+b_{-2}y_{i-2}+\ldots+b_{-k}y_{i-k}
\]
Importance of Linear Recurrences

- Linear recurrences appear in many domains
  - Mathematics
  - Data compression
  - Biology
- Parallel programming
  - Prefix sums
- Telecommunication
  - Digital filters
- Random-number gen.
- Finance and economics
- Complexity analysis
Prefix Sums

- Prefix sums are fundamental building blocks
  - Help parallelize many seemingly serial algorithms
- Given a sequence of values (integer or real)

\[
y_i = x_i + y_{i-1}
\]

- Compute the sequence whose values are the sum of all previous values from the original sequence

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>-1</th>
<th>8</th>
<th>-6</th>
<th>1</th>
<th>-9</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>-2</td>
<td>3</td>
</tr>
</tbody>
</table>
Digital (Recursive) Filters

- IIR filters are fundamental DSP algorithms
  - Used in telecommunication and audio DSP codes

- Digital equivalent to analog RC circuits

Illustration
- High-pass filter

\[ y_i = 0.93 x_i - 0.93 x_{i-1} + 0.86 y_{i-1} \]
Parallelization Difficulty

- Recurrence equation \((x_j = 0, \ y_j = 0, \ \forall j < 0)\)

\[ y_i = a_0 x_i + a_{-1} x_{i-1} + \ldots + a_{-p} x_{i-p} + b_{-1} y_{i-1} + b_{-2} y_{i-2} + \ldots + b_{-k} y_{i-k} \]

- Computation of element \(y_i\)

\[ \text{Input sequence: given, read-only} \]

\[ \text{Output sequence: written and read} \]

\[ \sum \]

The \(a_j\) are the non-recursion (feed-forward) coefficients.

The \(b_j\) are the recursion (feed-back) coefficients.

\(k\) denotes the order of the recurrence.

Data dependency!
Simplified Notation

- Recurrence equation
  \[ y_i = a_0 x_i + a_{-1} x_{i-1} + \ldots + a_{-p} x_{i-p} + b_{-1} y_{i-1} + b_{-2} y_{i-2} + \ldots + b_{-k} y_{i-k} \]

- Signature
  - Lists only non-recursion and recursion coefficients in parentheses and separated by a colon

Automatic Generation of 1D Recursive Filter Code for GPUs
Signature Examples

- **Standard prefix sum**
  - Prefix sum over scalar values
  - $(1 : 1)$

- **Low-pass digital filters**
  - Retain low frequencies but dampen high frequencies
  - 1-stage $(0.2 : 0.8)$, 2-stage $(0.04 : 1.6, -0.64)$, etc.

- **High-pass digital filters**
  - Retain high frequencies
  - 1-stage $(0.9, -0.9 : 0.8)$
  - 2-stage $(0.81, -1.62, 0.81 : 1.6, -0.64)$
Separation into Map + Recurrence

- Original recurrence

\[ y_i = a_0 x_i + a_{-1} x_{i-1} + \ldots + a_{-p} x_{i-p} + b_{-1} y_{i-1} + b_{-2} y_{i-2} + \ldots + b_{-k} y_{i-k} \]

- Equivalent map and simpler recurrence

  - Map operation
  \[ t_i = a_0 x_i + a_{-1} x_{i-1} + \ldots + a_{-p} x_{i-p} \] (\( a_0, \ldots, a_{-p} : 0 \))

  - Recurrence
  \[ y_i = t_i + b_{-1} y_{i-1} + b_{-2} y_{i-2} + \ldots + b_{-k} y_{i-k} \] (\( 1 : b_{-1}, \ldots, b_{-k} \))

- Benefit: easier to parallelize

  - Recurrence always has (\( 1 : \ldots \)) format; map is trivial
Our PLR Approach

- High-level idea
  - Break input into chunks of size 1 (trivial)
  - Iteratively combine adjacent chunks into larger chunks

- Two phases
  1. Merging
  2. Pipelining*

Automatic Generation of 1D Recursive Filter Code for GPUs
PLR Merging (1 : \(d\))

- Merging two adjacent chunks
  - \(v_0, v_1, \ldots, v_{m-1} \mid v_m, v_{m+1}, \ldots, v_{2m-1}\)

- Correcting element \(v_m\)
  - Per (1 : \(d\)), need to add \(d\) times prior element \(v_{m-1}\)
  - The correction term is \(d \cdot v_{m-1}\)

- Correcting element \(v_{m+1}\)
  - Need to add \(d\) times the corrected prior element
    - Already added \(d\) times \(v_m\) in an earlier iteration
    - Only need to add \(d\) times the prior correction term
  - The correction term is \(d \cdot d \cdot v_{m-1}\)
PLR Merging (1 : d) cont.

- Correcting elements $v_{m+2}, v_{m+3},$ etc.
  - The correction terms are $d^3 \cdot v_{m-1}, d^4 \cdot v_{m-1},$ etc.
  - Correction factor times carry $v_{m-1}$ from prior chunk

- Key observation
  - Carry value depends on input sequence
  - Correction factors only depend on recurrence
    - Can be precomputed as they are the same for all inputs

- Just the correction factors
  - $d, d^2, d^3, \ldots, d^m$ → $1 \mid d, d^2, d^3, \ldots, d^m$

Start with 1, apply recurrence (0 : d)

All factors are 1 for $d = 1$; prefix sum is trivial base case
PLR Merging (1 : d, e)

- Merging two adjacent chunks
  - \( v_0, v_1, \ldots, v_{m-2}, v_{m-1} | v_m, v_{m+1}, \ldots, v_{2m-1} \)

- Correcting element \( v_m \)
  - Per \( (1 : d, e) \), need to add \( d \) times \( v_{m-1} \) plus \( e \) times \( v_{m-2} \)
  - The correction term is \( d \cdot v_{m-1} + e \cdot v_{m-2} \)

- Correcting element \( v_{m+1} \)
  - Need to add \( d \) times \( (d \cdot v_{m-1} + e \cdot v_{m-2}) \) plus \( e \) times \( v_{m-1} \)
  - The correction term is \( d \cdot (d \cdot v_{m-1} + e \cdot v_{m-2}) + e \cdot v_{m-1} \), which is \( (d^2 + e) \cdot v_{m-1} + (d \cdot e) \cdot v_{m-2} \) after rearranging the terms
PLR Merging (1 : d, e) cont.

- Correcting elements $v_{m+2}$, $v_{m+3}$, etc.
  - The correction terms are $(d^3 + 2de) \cdot v_{m-1} + (d^2e + e^2) \cdot v_{m-2}$,
    
    $$
    (d^4 + 3d^2e + e^2) \cdot v_{m-1} + (d^3e + 2de^2) \cdot v_{m-2},
    $$
  - There are two carries $v_{m-1}$ and $v_{m-2}$ from prior chunk
    - Because the recurrence (1 : d, e) has order 2
  - Just the correction factors for $v_{m-1}$
    - $d$, $d^2 + e$, $d^3 + 2de$, $d^4 + 3d^2e + e^2$, ...
  - Just the correction factors for $v_{m-2}$
    - $e$, $de$, $d^2e + e^2$, $d^3e + 2de^2$, ...
PLR Merging (1 : d, e) cont.

- Correction factors for $v_{m-1}$
  - $d$, $d^2+e$, $d^3+2de$, $d^4+3d^2e+e^2$, ...
- Correction factors for $v_{m-2}$
  - $e$, $de$, $d^2e+e^2$, $d^3e+2de^2$, ...
- Both sequences can be generated by (0 : d, e)
  - 0, 1 | $d$, $d^2+e$, $d^3+2de$, $d^4+3d^2e+e^2$, ...
  - 1, 0 | $e$, $de$, $d^2e+e^2$, $d^3e+2de^2$, ...

The “1” indicates the location of the carry in the prior chunk
PLR Merging (1 : $b_{-1}, b_{-2}, ..., b_{-k}$)

- Correction-factor computation
  - Recurrence has order $k$, so $k$ lists of factors needed
  - Start with $k-1$ zeros and a one: $0, ..., 0, 1, 0, ..., 0$
    - “1” is in location of corresponding carry
  - Compute factors using $(0 : b_{-1}, b_{-2}, ..., b_{-k})$

Correction factors are $k$-nacci sequences
(generalized Fibonacci sequences)
PLR: Proof of Concept Tool

- PLR code generator
  - Compiles signature into CUDA code for GPUs
  - Performs domain-specific code optimizations

- Generated code
  - Performs map operation \((a_0, a_{-1}, ..., a_{-p} : 0)\)
  - Computes recurrence \((1 : b_{-1}, b_{-2}, ..., b_{-k})\)
    - First five merge steps are done at warp level
    - Remaining merge steps are done at thread-block level
    - Pipelining is performed at grid level*
  - Uses \(m \leq 9 \cdot 1024\) for floats and \(m \leq 11 \cdot 1024\) for ints
Experimental Methodology

- **GPU**
  - **GeForce GTX Titan X** (1.1 GHz cores, 3.5 GHz memory)
  - 3072 cores, 24 SMs, up to 49,152 active threads
  - 2 MB L2 cache, 12 GB of global memory (336 GB/s)

- **Compiler and flags**
  - `nvcc 7.5 with "-O3 -arch=sm_52"`

- **Comparison codes**
  - Prefix sums: **CUB** (Nvidia), **SAM** (us), **Scan** (CMU)
  - Digital filters: **Alg3** (IMPA), **Rec** (Halide/MIT), **Scan**
  - All downloaded except Scan (uses CUB’s scan)
Different Approaches

- **CUB**: prefix scan on templated objects/operators
  - Warp/block/grid prefix scans with carry pipelining
- **SAM**: scalar/higher-order/tuple-based prefix sums
  - Auto-tuned specialized prefix sums (warp/block/grid)
- **Scan**: arbitrary 1D linear recurrences
  - Elem = vector, recur = matrix, operator = vec/mat mult
- **Alg3**: 2D recursive filters (for image processing)
  - Framework to facilitate overlap of multi-dim filters
- **Rec**: 2D recursive filters (for image processing)
  - Specify filters and heuristics, Halide compiles/optimizes

We use scaled-down versions with 1D filters on 2D inputs
1-Stage Low-Pass Filter Throughput

PLR outperforms other tools on large inputs

PLR reaches memory-copy throughput; cannot be exceeded

Rec runs 2D input (many short carry chains)

Alg3 computes forward and backward recurrence

Automatic Generation of 1D Recursive Filter Code for GPUs
2-Stage Low-Pass Filter Throughput

L2 cache capacity exceeded

Alg3 and Rec read data twice
High-Pass Filter Throughput

Alg3 and Rec do not support multiple $a$ coefficients

Scan yields 50% of throughput b/c it accesses 2x as much data
Prefix-Sum Throughput

CUB, SAM, and PLR perform roughly equally.
2-Tuple Prefix-Sum Throughput

PLR outperforms CUB and SAM on large inputs

Scan accesses six times as much data
2nd-Order Prefix-Sum Throughput

SAM is much faster on medium and large inputs

PLR performs on par with CUB
Domain-Specific Code Optimizations

Int: specializing 0 and 1 factors is important
No specialization implemented
Float: suppressing 0 factors is important

Automatic Generation of 1D Recursive Filter Code for GPUs
Try PLR for Yourself

http://cs.txstate.edu/~burtscher/research/PLR/

PLR v1.0

PLR is an automatic parallelizer and CUDA code generator for linear recurrences, including prefix sums and 1D digital IIR filters, as described in this ASPLOS paper.

To generate PLR code for a specific recurrence, enter the signature of the recurrence in the text box below and click the submit button. Note that the (generated) PLR code is protected by this license and that by downloading the code you agree to the terms and conditions set forth in this license.

For example, to generate scalar, first-order, integer prefix-sum code, enter the signature "1 : 1" without any quotes or parentheses (note the spaces before and after the colon). Table 1 and the surrounding text in the ASPLOS paper explain how to create the signatures for other recurrences.

SIGNATURE: $1 : 2, -1$

Assuming the generated code has been stored in a file called plr.cu, it can be compiled as follow:

```
nvcc -O3 -arch=sm_35 plr.cu -o plr
```

To execute the included test code and have it compute a recurrence over 100,000,000 elements, enter:

```
./plr 100000000
```

The PLR code has been tested on Pascal-, Maxwell- and Kepler-based GPUs with integer and floating-point recurrences. It requires at least compute capability 3.0. If an unstable recurrence is entered, the code may not compile because of correction factors that are ±∞ or NaNs.
Summary

- Linear recurrences are widely used computations
  - Difficult to parallelize due to data dependencies
- PLR: new general parallelization approach
  - Based on iterative merging (and pipelining)
  - Input-independent correction factors (k-nacci)
    - Precomputed and optimized for a given recurrence
  - Work, space, and communication efficient
- Highest GPU performance in many cases
  - Automatically compiled, parallelized, and optimized
Thank you!

▪ Acknowledgments
  ▪ Diego Nehab, André Maximo, Gaurav Chaurasia, Sahar Azimi, NSF, Nvidia, paper reviewers

▪ Contact information
  ▪ burtscher@txstate.edu

▪ PLR web page
  ▪ http://cs.txstate.edu/~burtscher/research/PLR/
  ▪ Includes link to paper