Diana Elisa Noveanu
gr. 935

PDP Lab 7 (1) Documentation

(1) Requirement

Given a sequence of *n* numbers, compute the sums of the first *k* numbers, for each *k* between 1 and *n*. Parallelize the computations, to optimize for low latency on a large number of processors. Use at most 2**n* additions, but no more than 2*log(*n*) additions on each computation path from inputs to an output. Example: if the input sequence is 1 5 2 4, then the output should be 1 6 8 12.

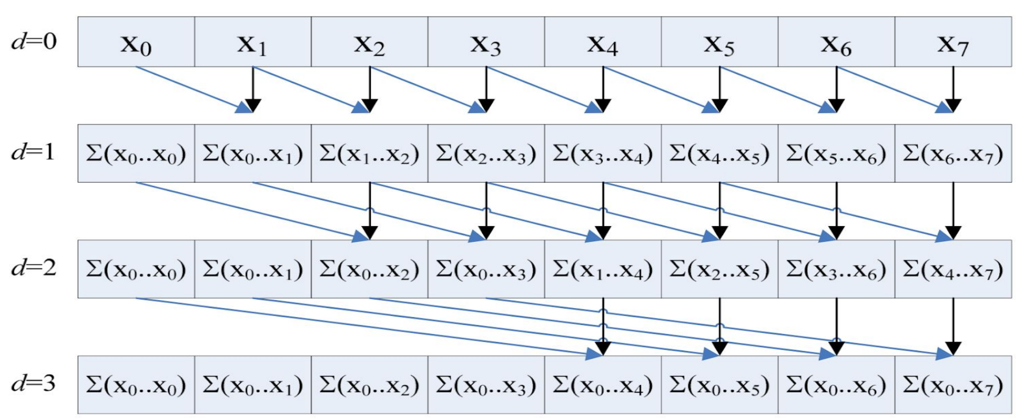
(2) Algorithm

The problem is very famous, usually named Parallel Prefix Scan, which refers to applying a binary associative operator over an array in a manner similar to reduce in high level languages.

For example, having an array A= a1a2..an and a binary associative operator x, we want to obtain another array B = b1b2...bn, where b1 = a1, b2 = a1xa2, b3 = a1xa2xa3, bi = a1xa2...xai.

The algorithm has two phases:

1. upSweep which adds the numbers like in a binary tree
2. downSweep which distributes the results



The implementation is the one described in this article:
https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.htm

(3) Setup

MacBook Pro (Retina, 13-inch, Early 2015)
Processor 2.7 GHz Intel Core i5
Memory 8 GB 1867 MHz DDR3
Graphics Intel Iris Graphics 6100 1536 MB
Language of implementation: C++

(4) Implementation

I Description

For the upSweep phase I've used the following approach:

```
for d = 0 to log2 n – 1 do
    for all k = 0 to n – 1 by 2^(d+1) in parallel do
        x[k + 2^(d+1) – 1] = x[k + 2^d – 1] + x[k + 2^d +1 – 1]
```

As we can see, we can have log2n threads, each thread doing a level of computation, like in a binary tree, where the for having *in parallel do* represents the threadWork for the upSweep phase.

For the downSweep phase the following algorithm was used, again distributing the results in a binary tree manner, it is very similar to the upSweep phase but mirrored.

```
x[n – 1] ← 0
for d = log2 n – 1 down to 0 do
    for all k = 0 to n – 1 by 2^d +1 in parallel do
        t = x[k + 2^d – 1]
        x[k + 2^d – 1] = x[k + 2^d +1 – 1]
        x[k + 2^d +1 – 1] = t + x[k + 2^d +1 – 1]
```

II Parallelization

It's quite simple to deduce the parallelization techniques I've used, the fors with *in parallel do* represents the thread functions for upSweep, and downSweep phases.

III Locking

For synchronization I've used locks on every atomic add operation in thread functions

(5) Performance Measurements

```
1 3 6 10 15 25 45 75 Sequential time: 4.398e-06

1 3 6 10 15 25 45 75 Parallel time: 4.398e-06
```

Because the algorithm is tested on a CPU rather than on a GPU, the performance metrics are not better in comparison to the sequential algorithm implementation, especially not when ran on such small arrays.

```
1 3 6 10 15 25 45 75 119 164 210 257 305 354 404 455 Sequential time: 4.264e-06

1 3 6 10 15 25 45 75 119 164 210 257 305 354 404 455 Parallel time: 0.00027101
```

Plus, in certain cases the sequential algorithm outperforms the parallel algorithm, as it doesn't have to wait for any locks to be released.