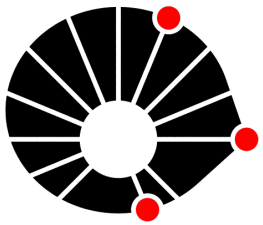
Support for Parallel Scan in OpenMP

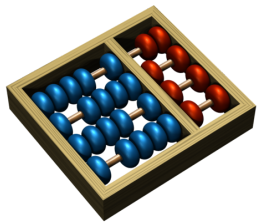
E. Maicol G. Zegarra

maicol.zegarra@students.ic.unicamp.br
University of Campinas
Institute of Computing

Advisor
Prof. Dr. Guido Costa Souza de Araújo

Co-advisor
Prof. Dr. Marcio Machado Pereira



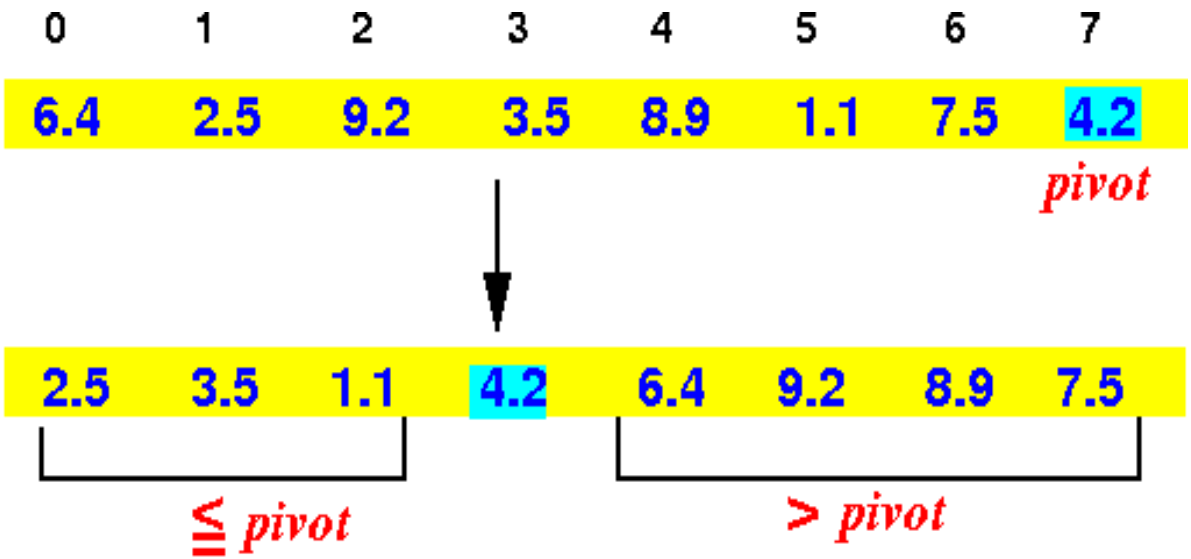


24 April 2018

Table of Contents

Table of Contents

Parallel Quicksort Algorithm



How can we do this in parallel?

Prefix Sums Problem

In computer science, the prefix sum, cumulative sum, inclusive scan or simply scan is defined as:

- Given a **binary associative operator** $\oplus$;
- An ordered set of n elements $[a_0, a_1, \ldots, a_{n-1}]$;
- Return the ordered set $[a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}]$.

Prefix Sums Problem

There are two versions of Prefix Scan: inclusive and exclusive.

	[3 1 7 0 4 1 6 3]
Inclusive	[3 4 11 11 15 16 22 25]
Exclusive	[0 3 4 11 11 15 16 22]

Prefix Sums Problem

There are two versions of Prefix Scan: inclusive and exclusive.

	[3 1 7 0 4 1 6 3]
Inclusive	[3 4 11 11 15 16 22 25]
Exclusive	[0 3 4 11 11 15 16 22]

Table of Contents

Algorithm (Proposed by Horn in 2005)

Based in Hillis and Steele Work (1986).

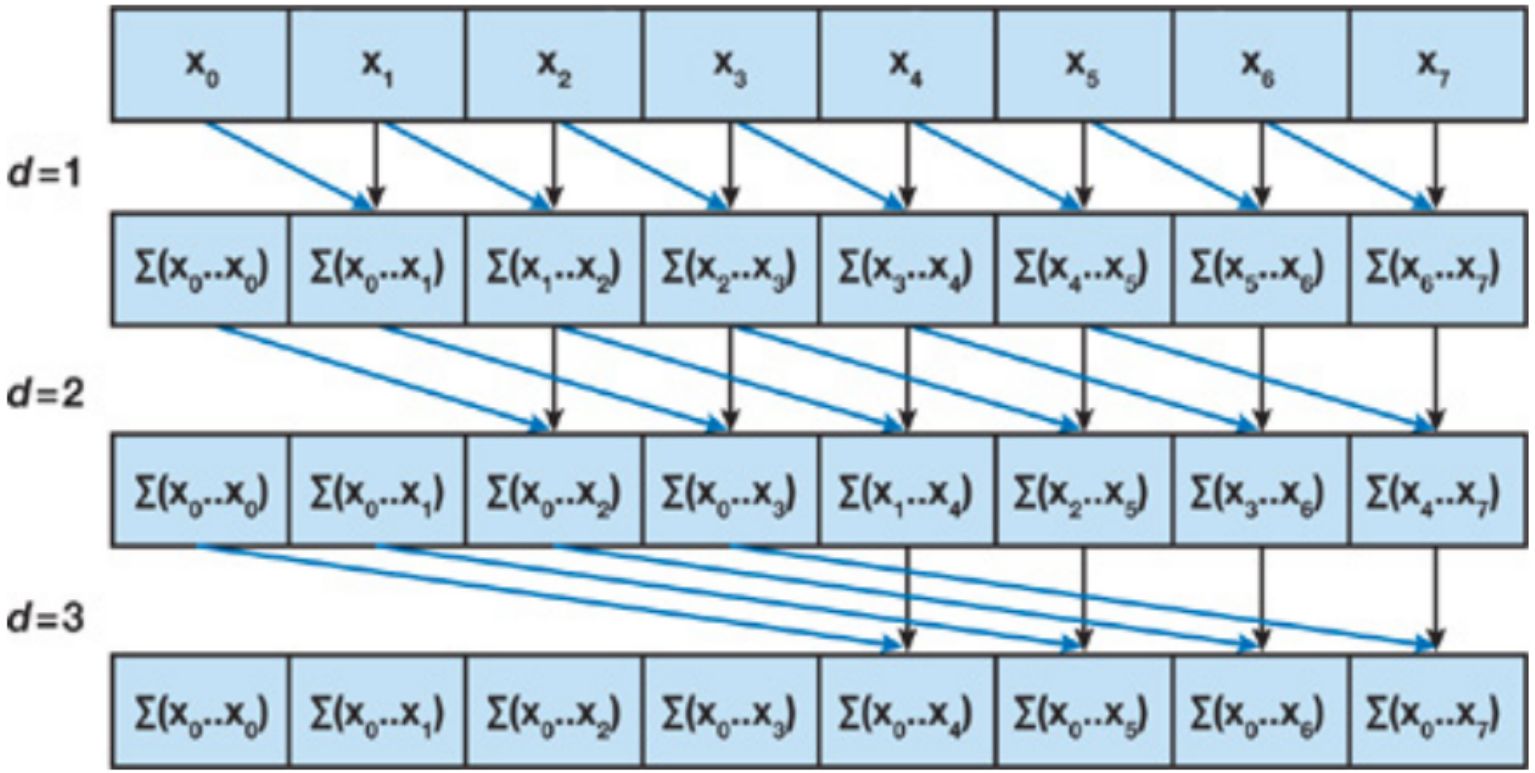


Figure 1: An ilustration of the algorithm proposed by Horn

Complexity Analysis

The algorithm performs $O(n\ log_2 n)$ addition operations.

- The tree has $log_2 n$ levels;
- It is performed $O(n)$ adds for each level;

Remember that a sequential scan performs $O(n)$ adds. Therefore, this naive implementation is not work-efficient. The factor of $log_2 n$ can have a large effect on performance.

Algorithm (Proposed by Mark Harris in 2007)

Based in Blelloch's Work (1990);
Two phases (Up-Sweep & Down-Sweep):

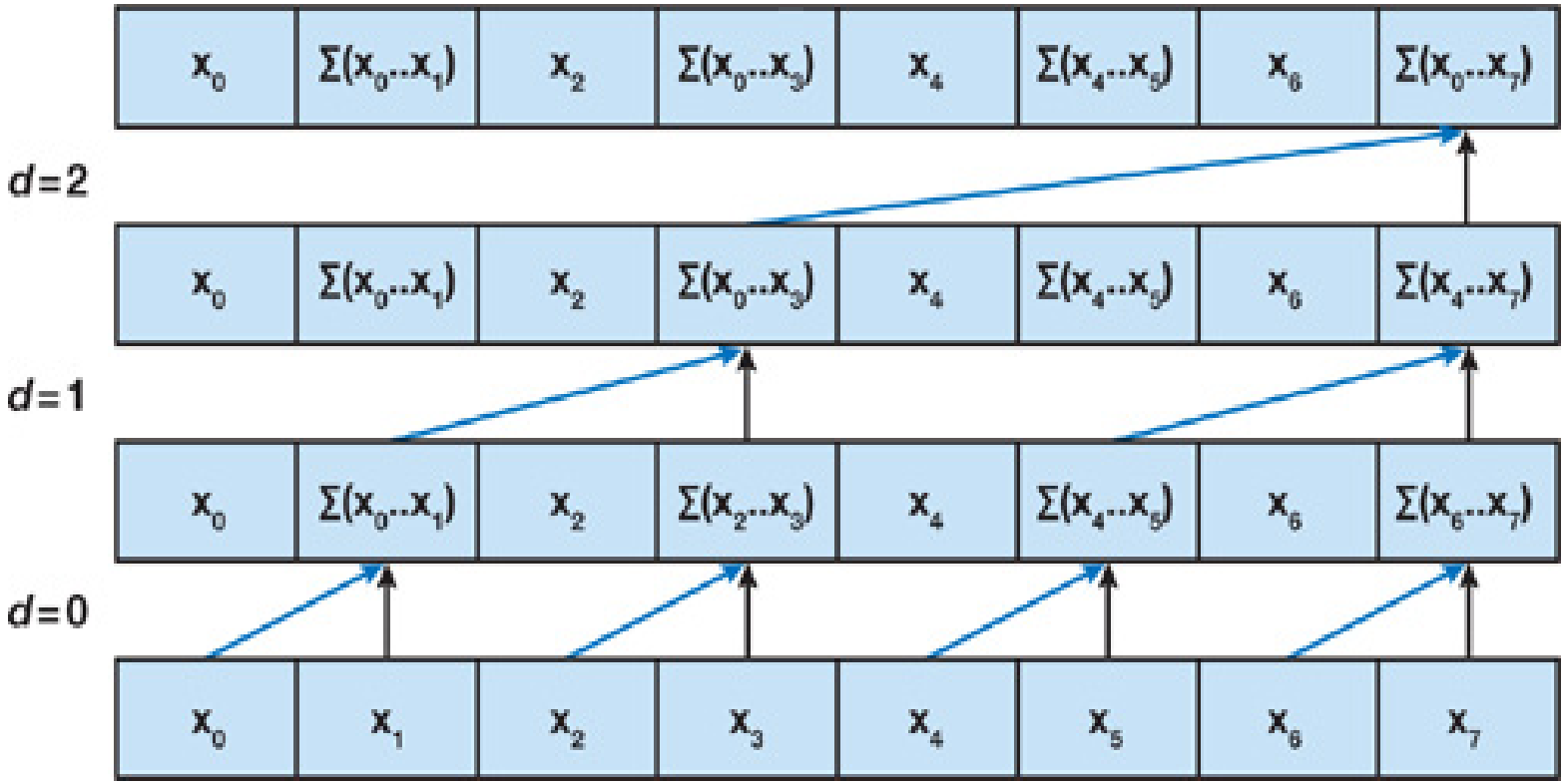
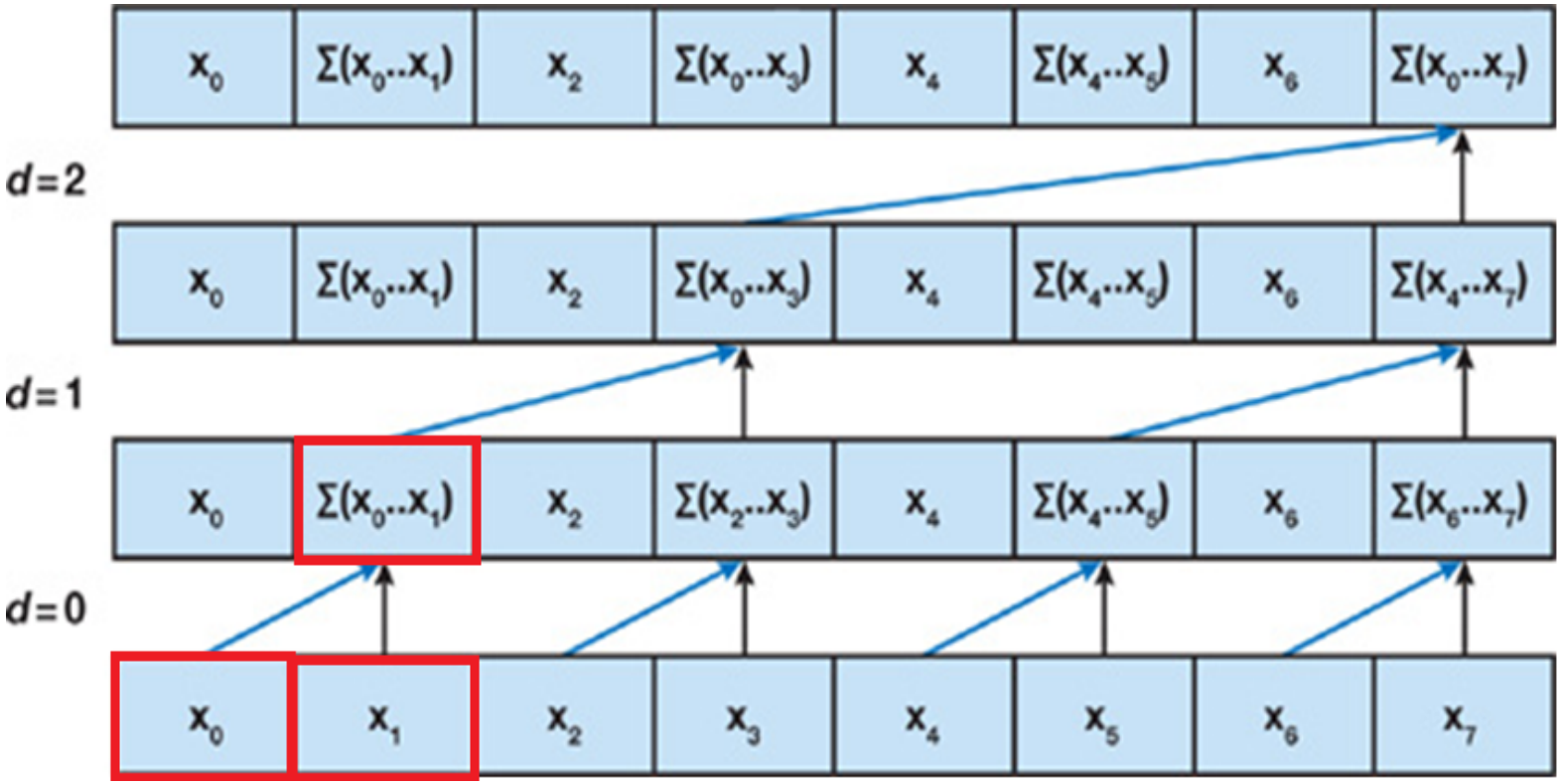
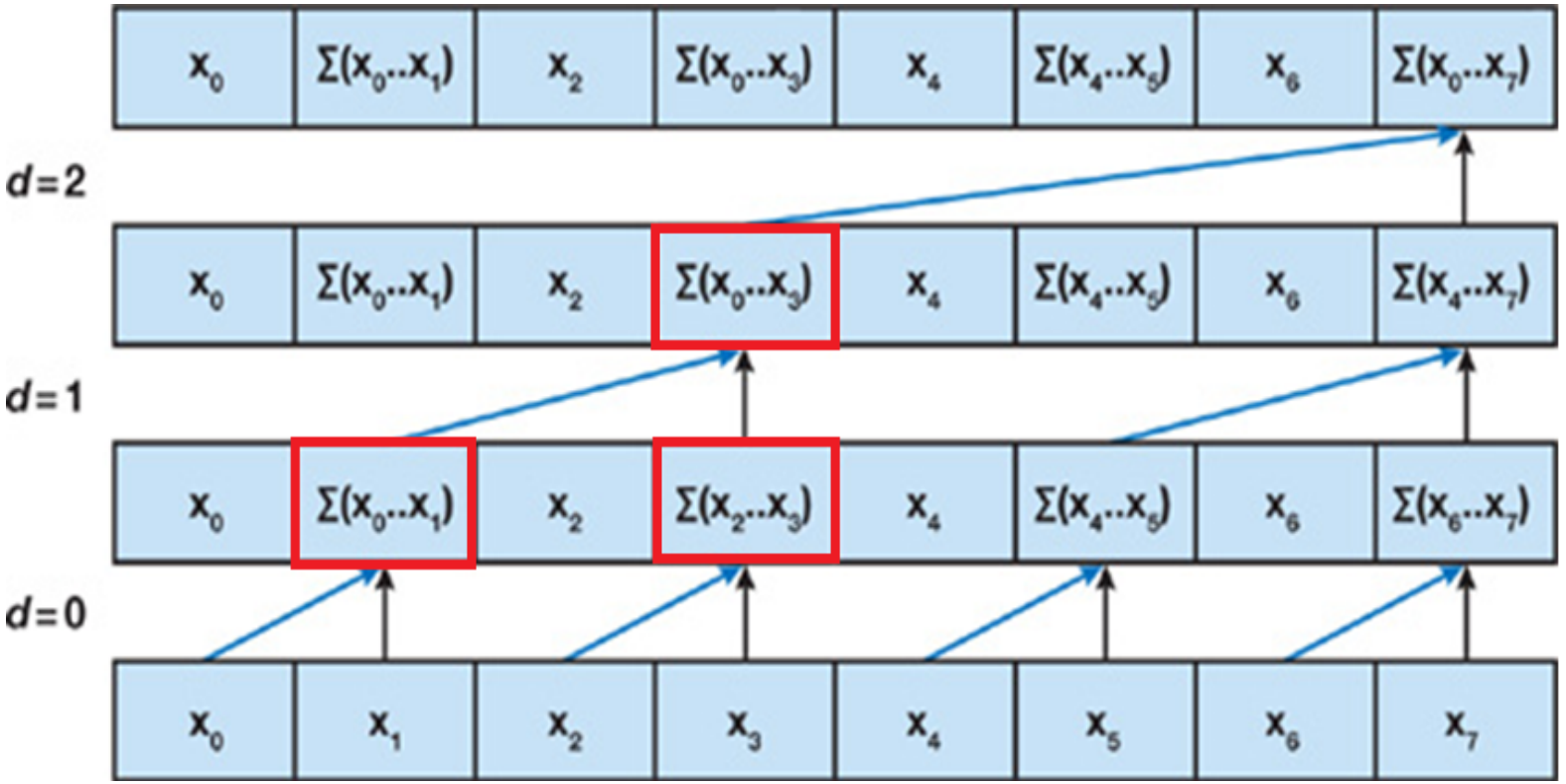


Figure 2: An illustration of the **Up-Sweep** phase

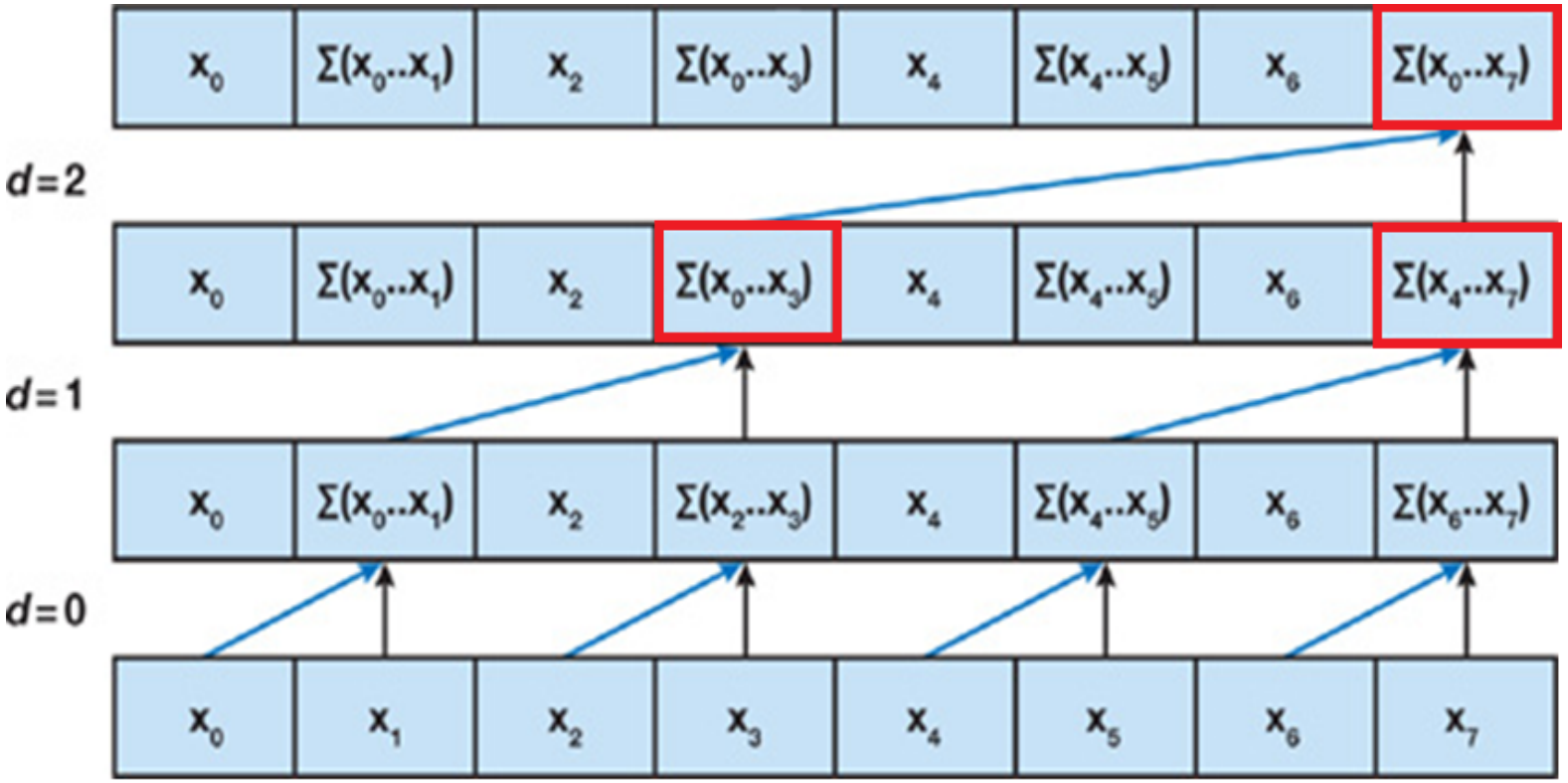
An illustration of the Up-Sweep phase



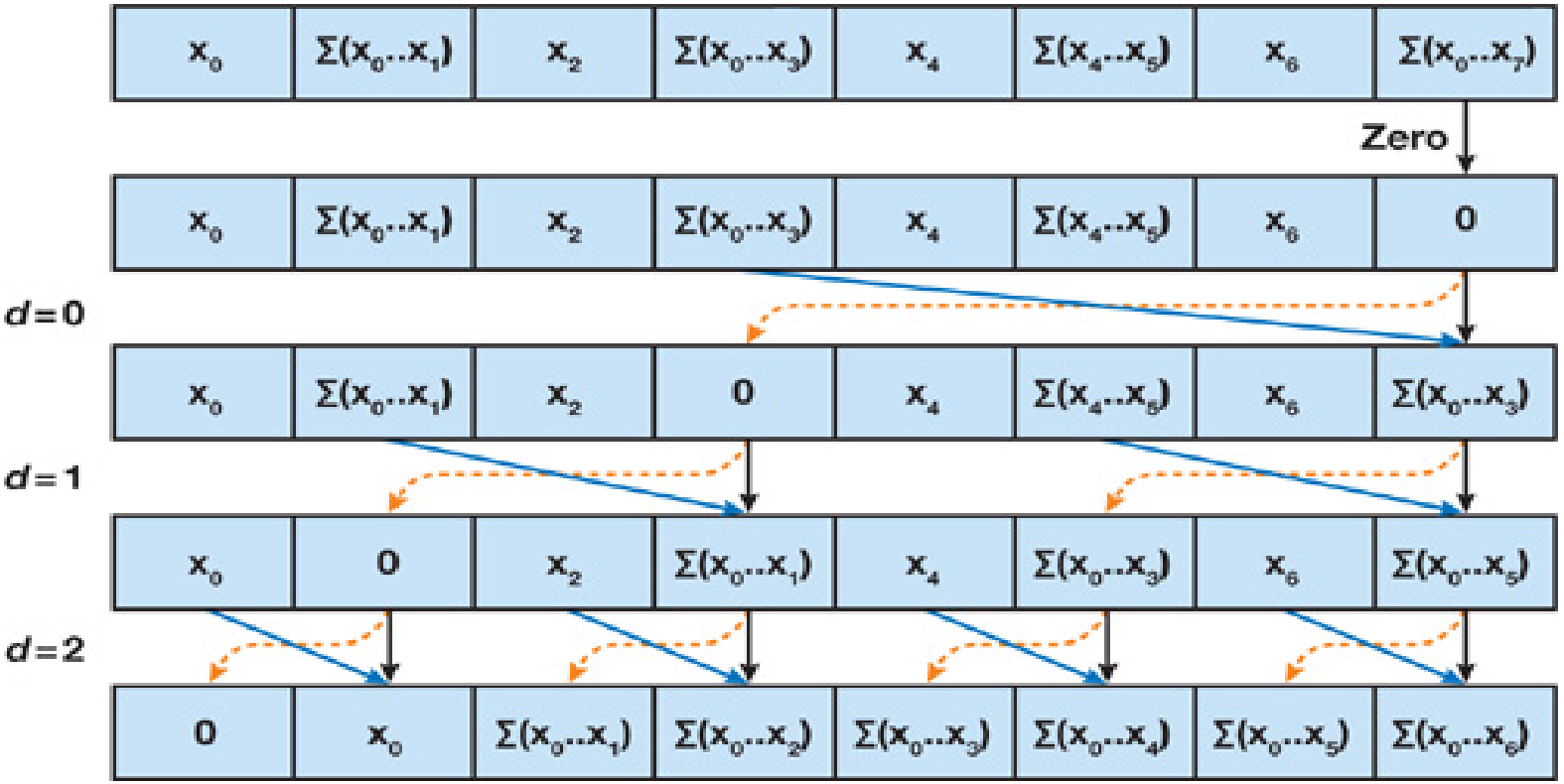
An illustration of the Up-Sweep phase



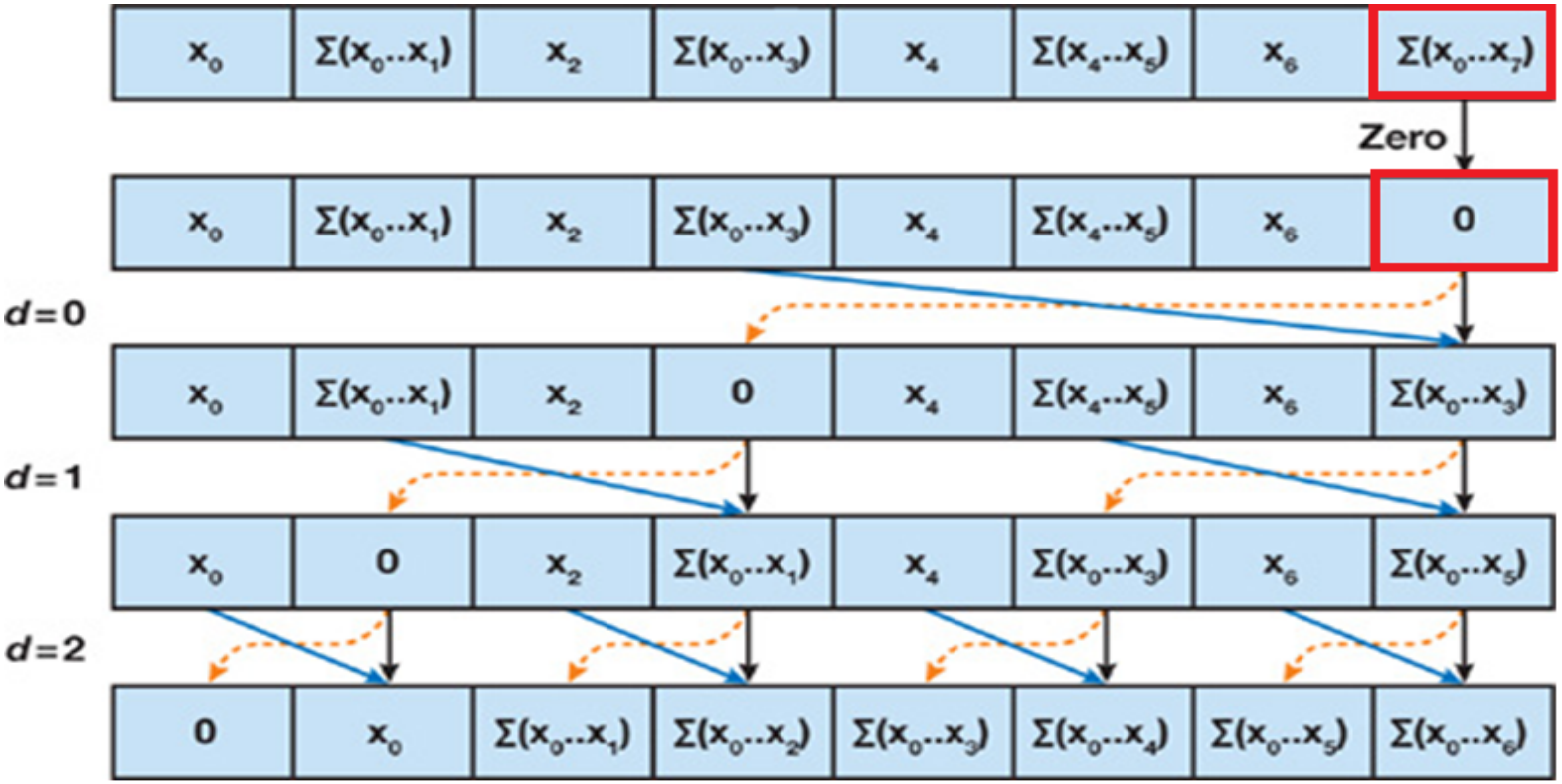
An illustration of the Up-Sweep phase



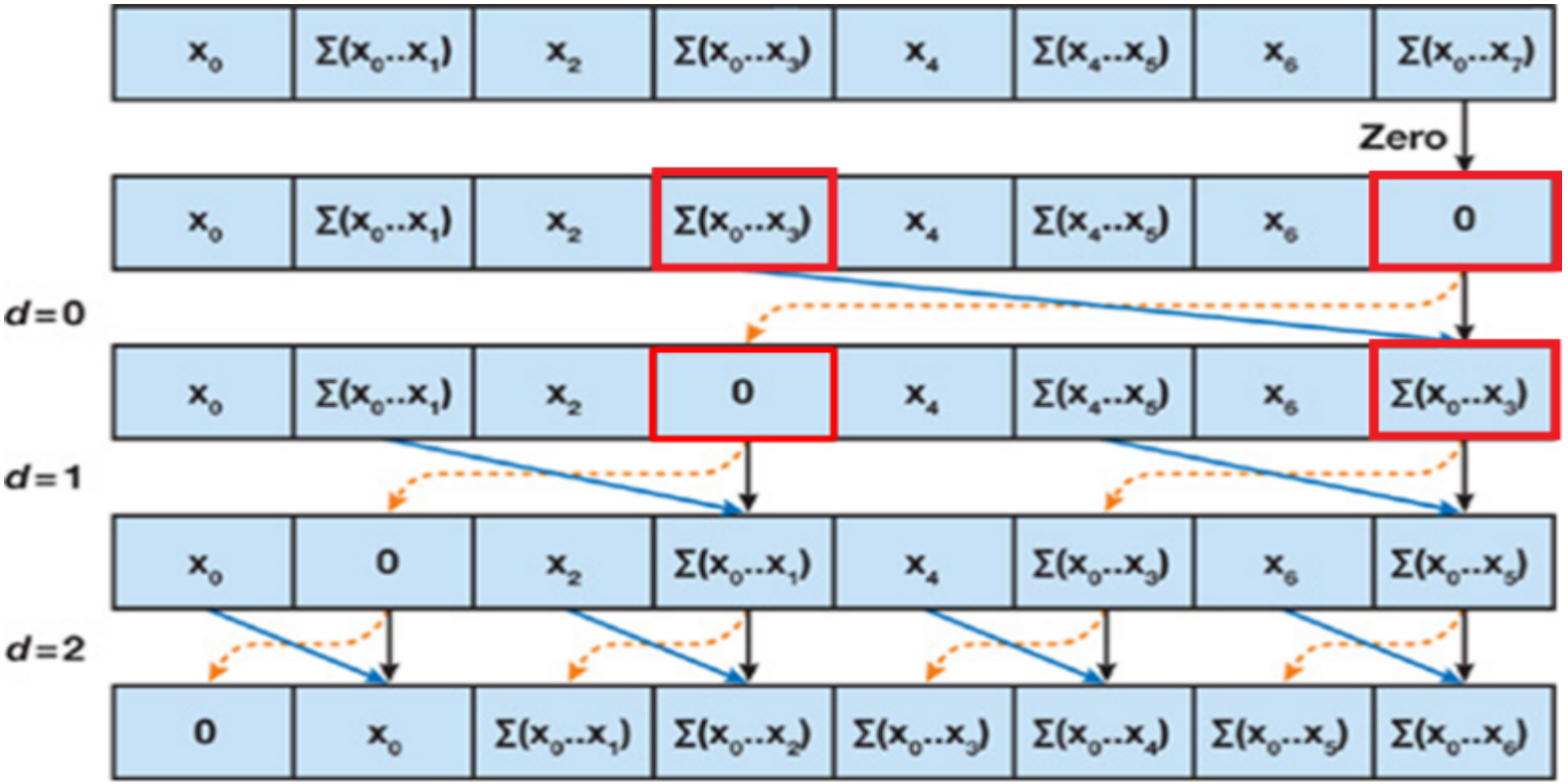
An illustration of the Down-Sweep phase



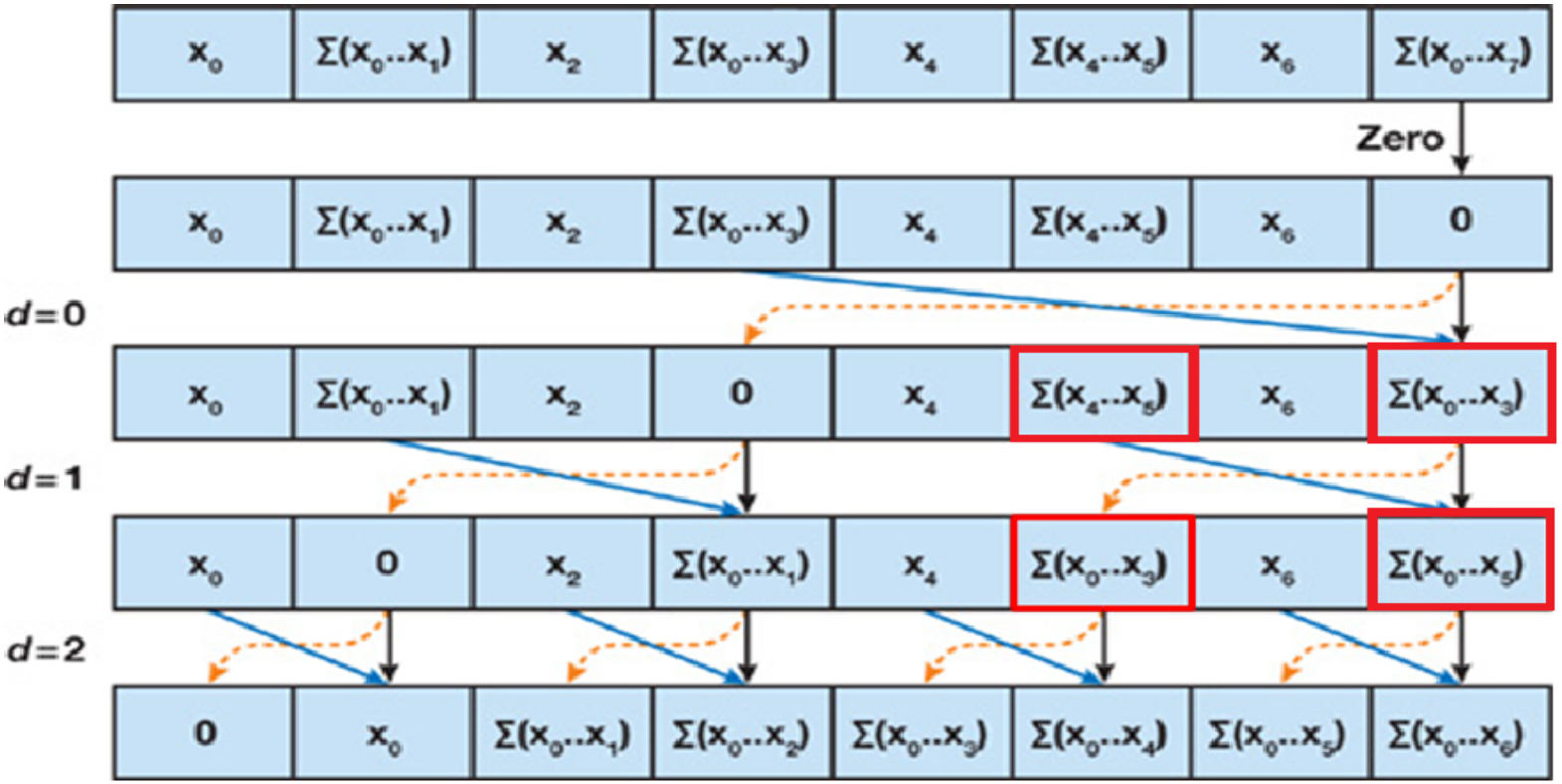
An illustration of the Down-Sweep phase



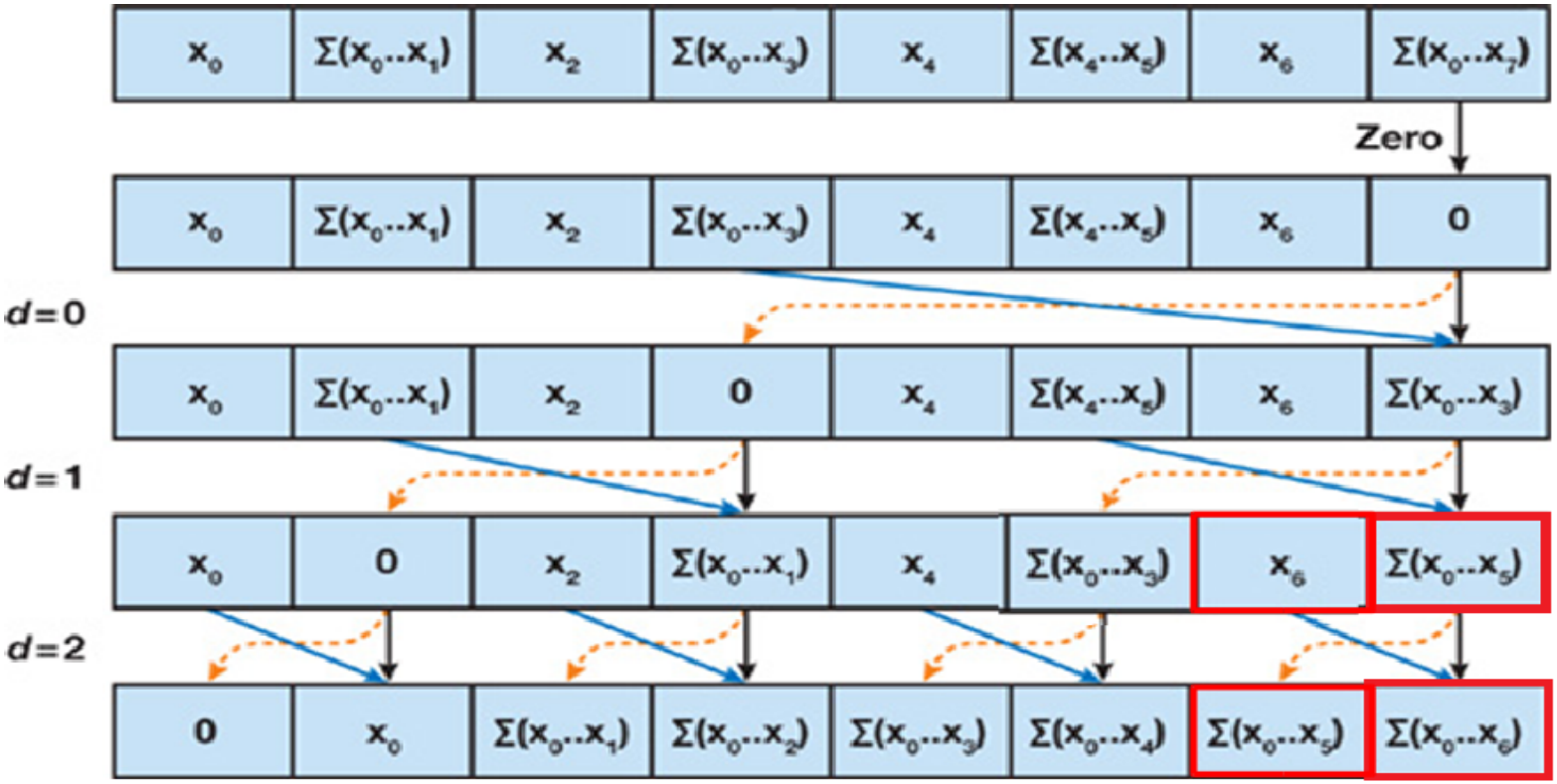
An illustration of the Down-Sweep phase



An illustration of the Down-Sweep phase



An illustration of the Down-Sweep phase



Complexity Analysis

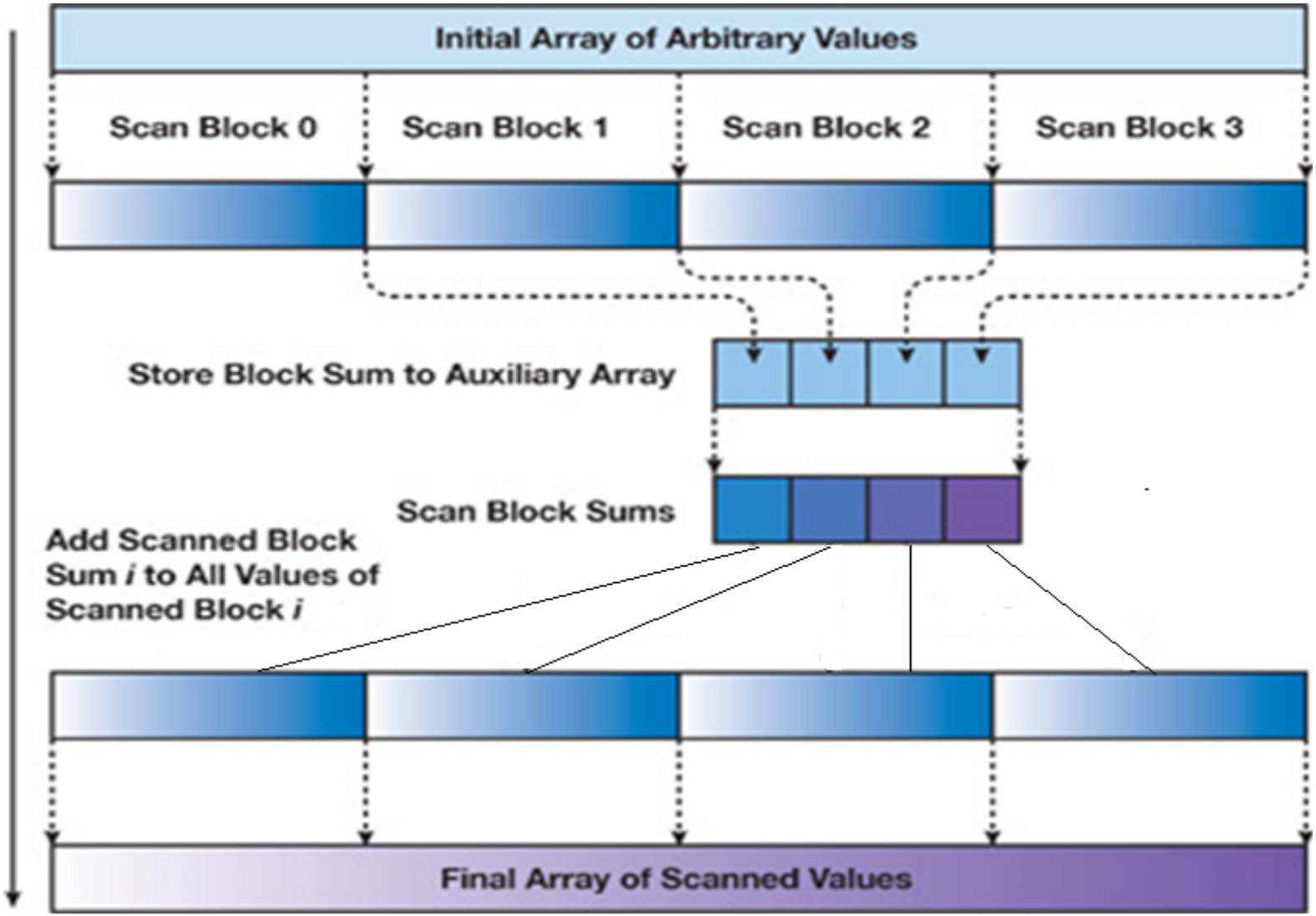
Up-Sweep Complexity:

- A binary tree with n leaves and $d = log_2 n$ levels, and each level d has 2^d nodes. If It is performed one add per node, It is computed O(n) adds on a single traversal of the tree.

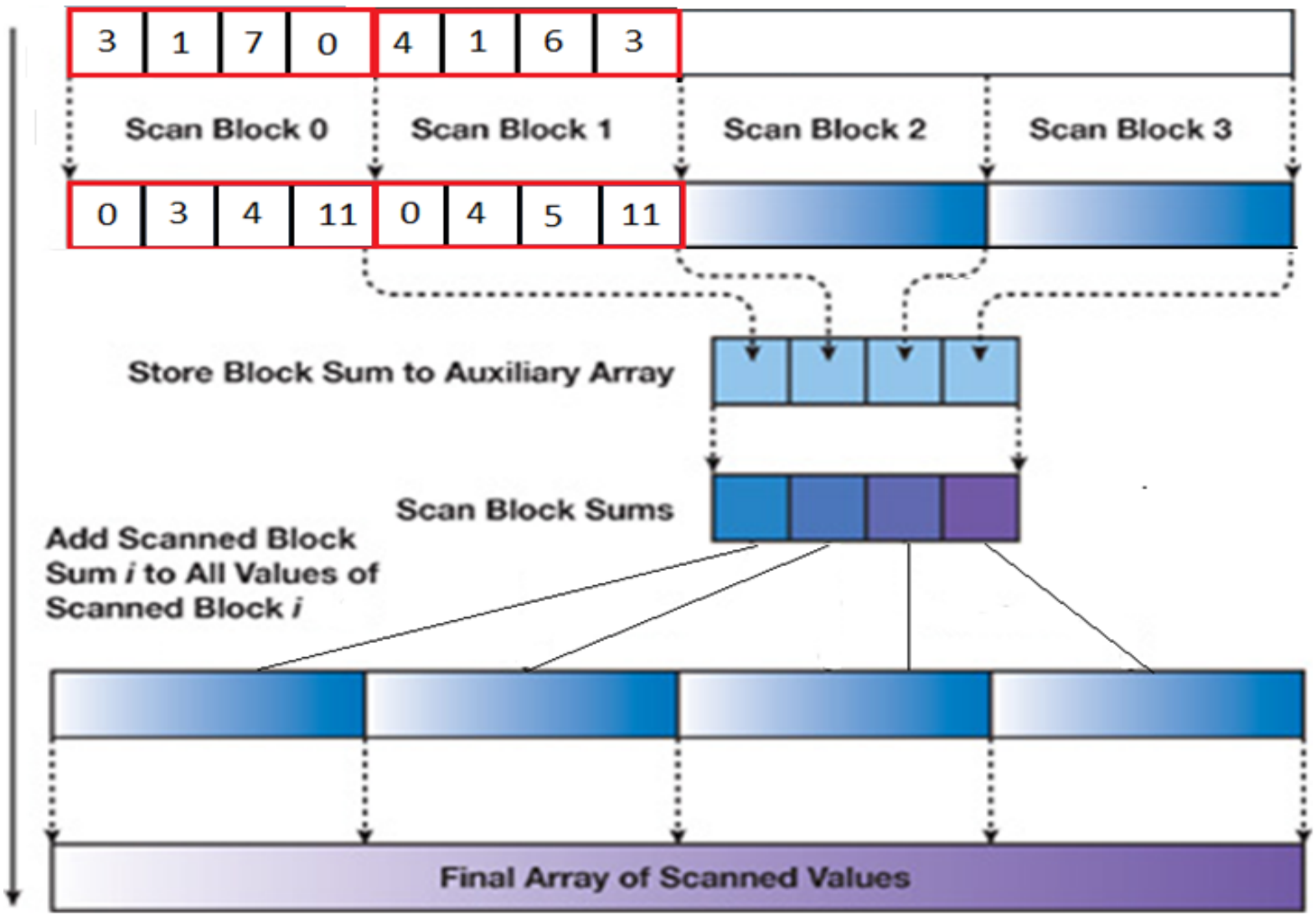
Down-Sweep Complexity:

- Similar as Up-Sweep phase. It is performed O(n) adds and O(n) swaps.

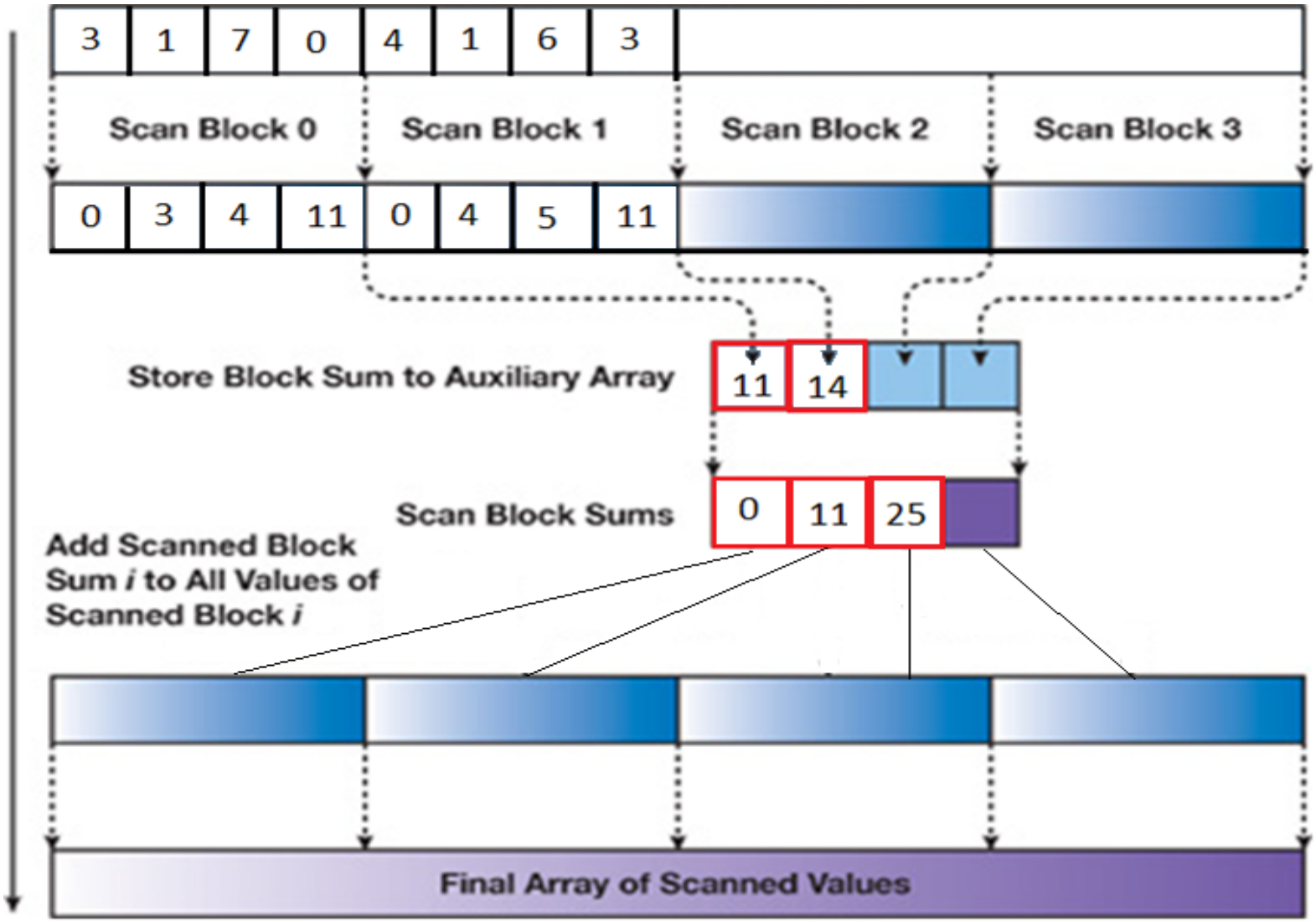
Algorithm for large arrays



Algorithm for large arrays



Algorithm for large arrays



Algorithm for large arrays

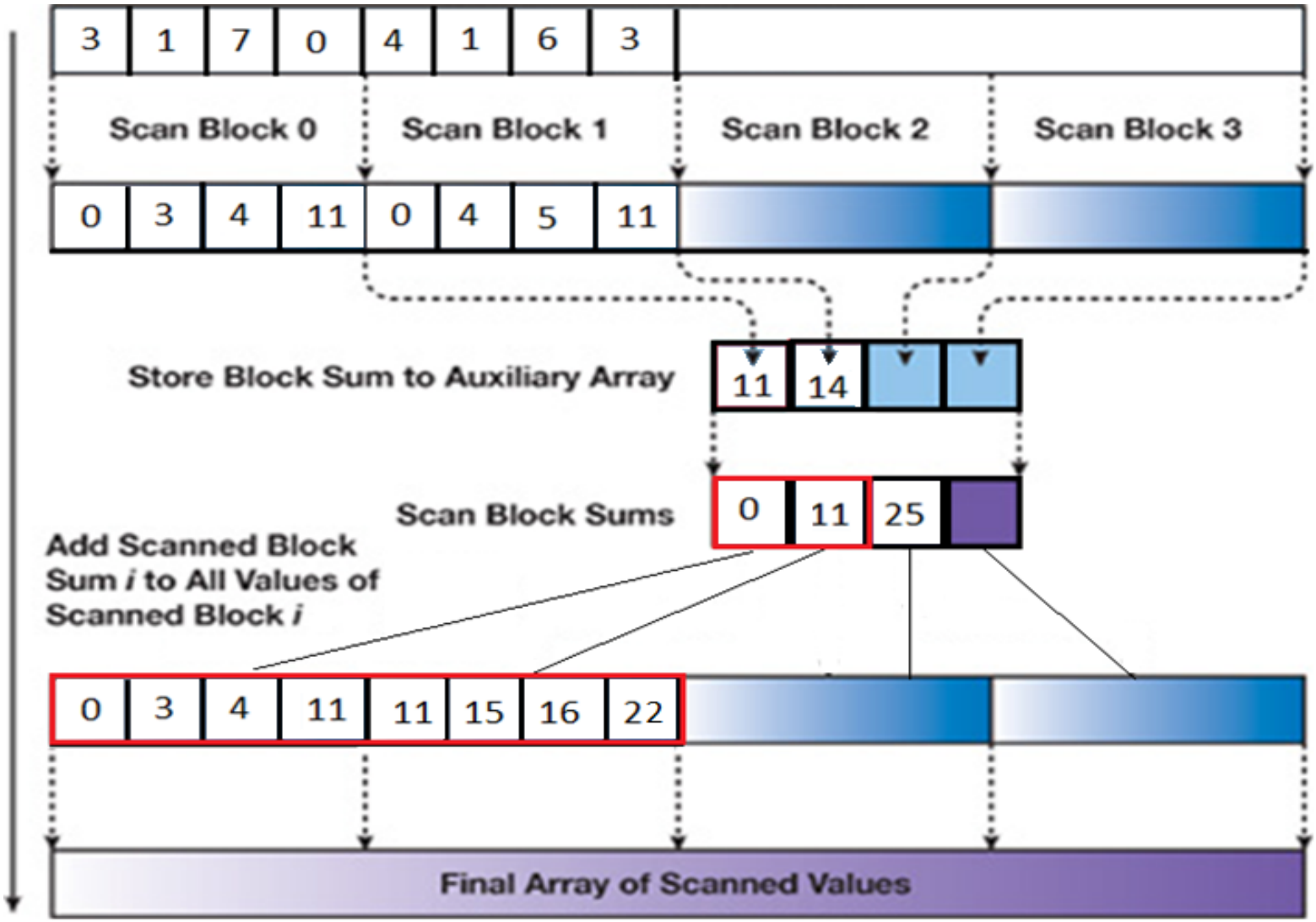
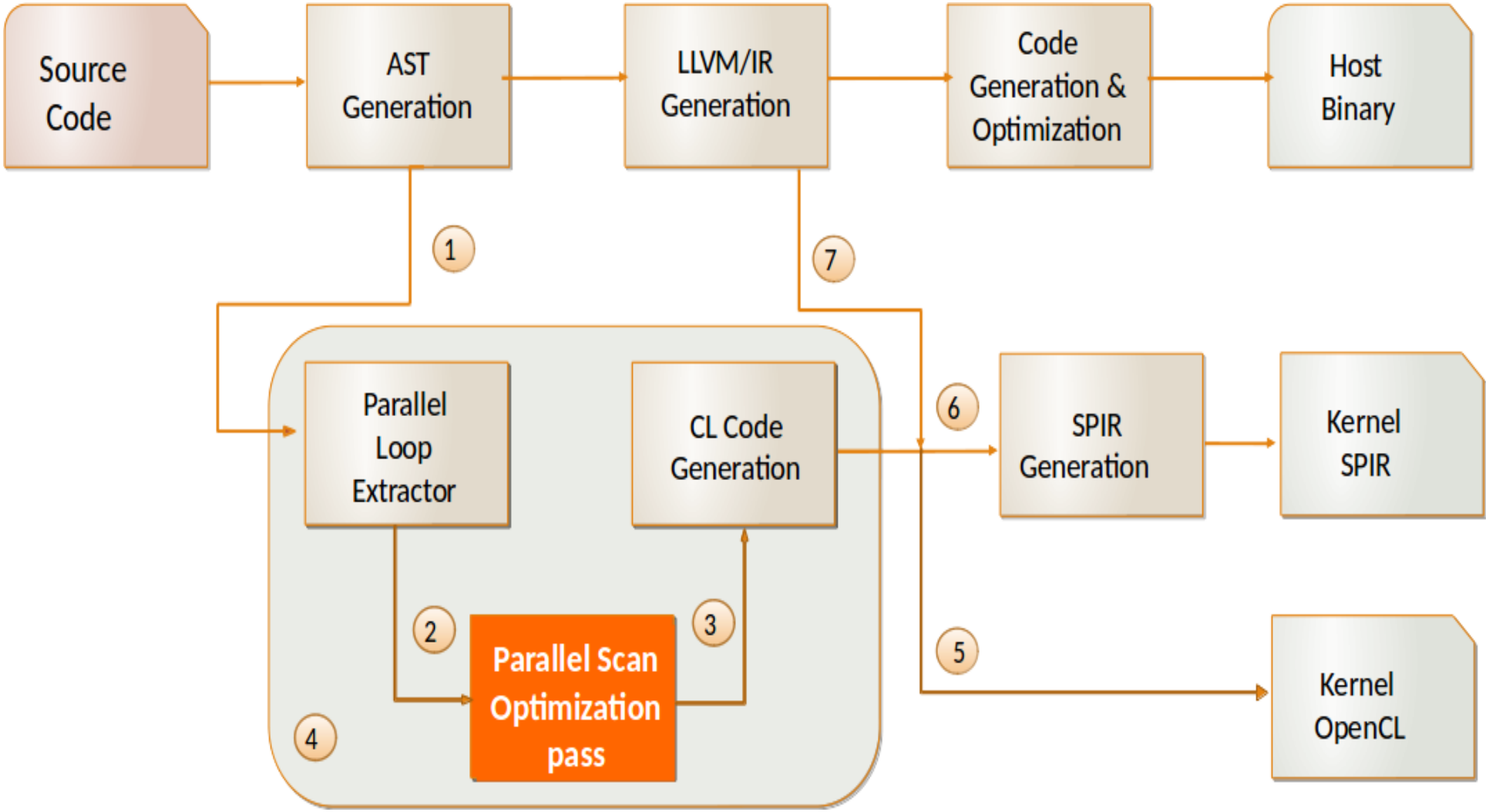


Table of Contents

What is AClang?

- AClang compiler is based in LLVM Clang 3.5;
- Designed by our group at UNICAMP;
- Supports Offloading to accelerators;
- From OMP to OpenCL or SPIR code.

AClang Compiler Pipeline



AClang Compiler

Listing 1: Sequential Implementation of Scan

```
void scan(int *x,int *y,int n) {
  y[0] = 0;
  for(i = 1; i < n; i++)
    y[i] = y[i-1] + x[i-1];
}
```

Listing 2: Parallel implementation of Scan using the new clause

```
void scan(int *x,int *y,int n) {
  y[0] = 0;
  #pragma omp target device (device)\
              map(to:x[:N]) map(from:y[:N])
  #pragma omp parallel for scan (+:y)
  for(i = 1; i < n; i++)
    y[i] = y[i-1] + x[i-1];
}
```

Table of Contents

Implmentation of Scan in AClang

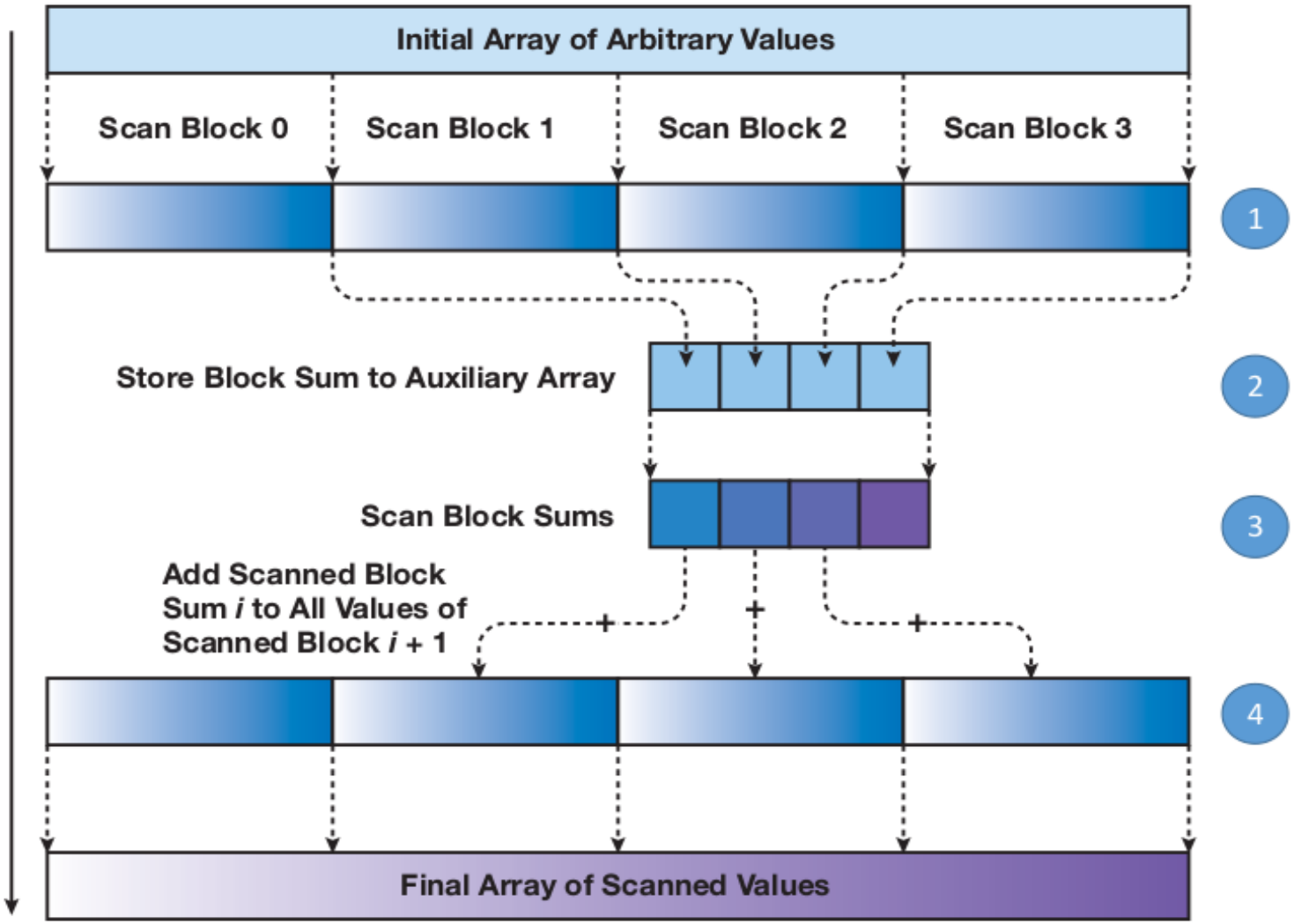
The implementation is based on the best parallel scan algorithm known today (Mark Harris).
Main steps:

- The algorithm obtains the pieces of information of the omp scan clause (variable type, operator, vector size);
- Compute the number of blocks and threads per block to use (Scan algorithm just work for 2^k vector sizes);
- Generate code for the runtime library to call the OpenCL driver to compile the kernels and dispatch them for execution;
- Send information from omp declare target section (new variables types and operator overload).

Table of Contents

The Template

The template represents the algorithm explained in the following figure:



Template's Structure

Listing 3: Template's Structure

```
//OMP_DECLARE_TARGET REGION
//END OMP_DECLARE_TARGET
#define Neutral I //Neutral value

__kernel void kernel_0 (datatype *input,\
        datatype *S, int n){
  //Performs (1) and (2)
}

__kernel void kernel_1(dataType *input, int n){
  //Performs (3)
}

__kernel void kernel_2(dataType *input,\
         dataType *output, dataType *S){
  //Performs (4)
}
```

The Template

The algorithm has only three parameters that could be different according to the applications.

- Variable type (int, float, double and so on);
- Operator used (+, $*$, &, $|$, *max*, *min* and so on);
- OMP declare target region (New variable type and operator).

The Template

Ouput and input vector are the same. In this case the last kernel does not have the output vector, the result is overwritten in input.

Listing 4: Return the result into the input vector

```
int main(){
  ...
  int aux1 = input[0], aux2;
  input[0] = 0;
  #pragma omp target device (GPU) map(tofrom:input[:N])
  input[0] = 0;
  #pragma omp parallel for scan(+:input)
  for(i = 0 ; i < n ; i++){
    aux2 = input[i];
    input[i] = input[i-1] + aux;
    aux = aux2;
  }
  ...
}

__kernel void kernel_2(dataType *input, dataType *S)
```

Table of Contents

Example of Applications

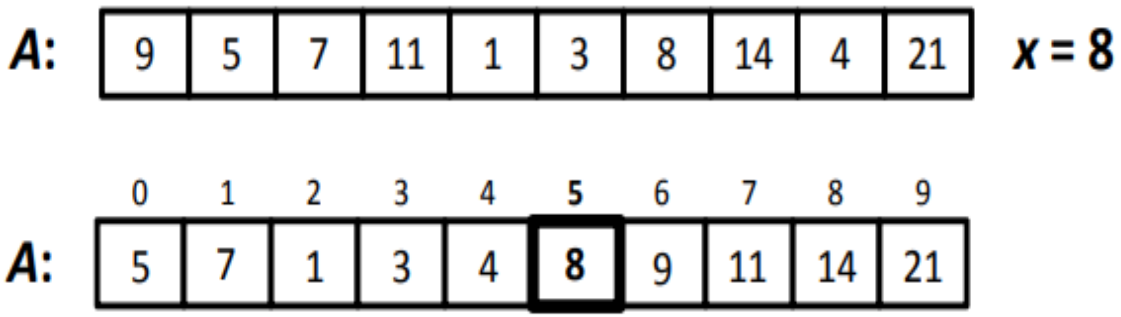
- Random number generation
- Sequence alignment
- N-body problem
- To perform lexical analysis
- Polynomial Evaluation
- To implement Parallel Quick Sort version
- Array Filter
- Solving linear recurrences

Example of Applications

- Random number generation
- Sequence alignment
- N-body problem
- To perform lexical analysis
- Polynomial Evaluation
- To implement Parallel Quick Sort version
- Array Filter
- Solving linear recurrences

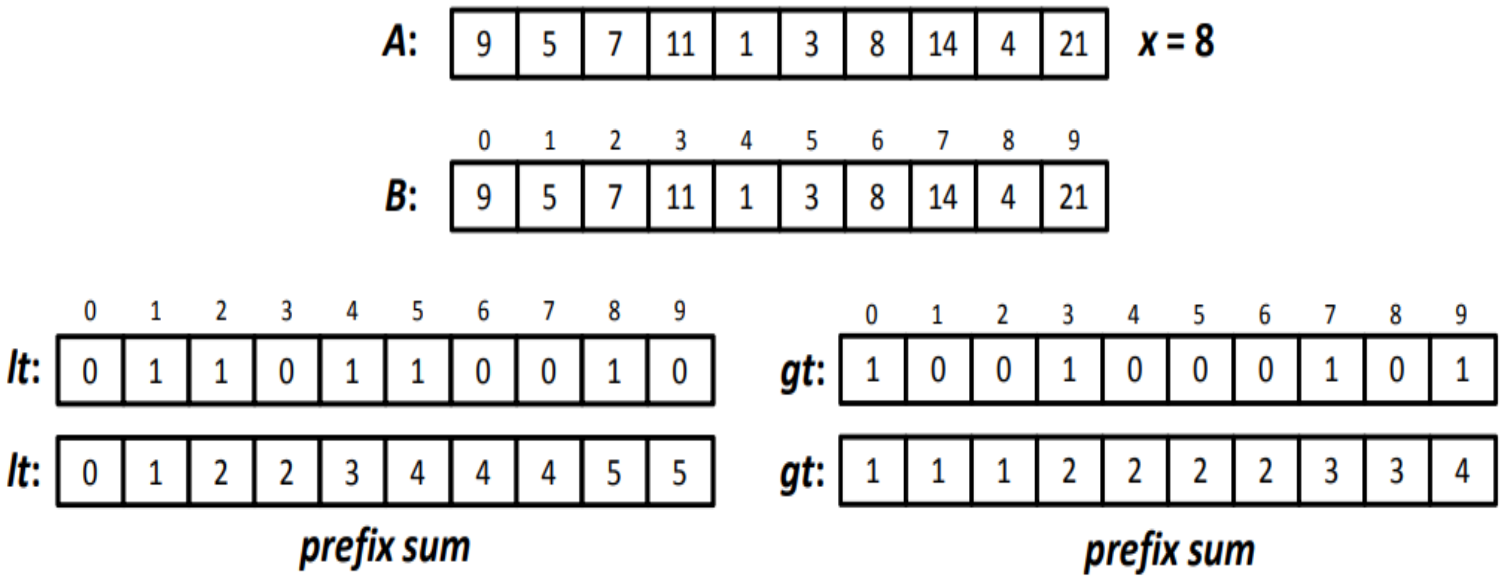
Example: Array Filter

Input: An array A[l:r] of integers elements, and an element **x** from A[l:r].
Output: Rearrange the elements of A[l:r], such that for some index $k \in [l,r]$, all elements in A[l:$k-1$] are smaller than **x** and all elements in A[$k+1$:r] are larger than **x**.

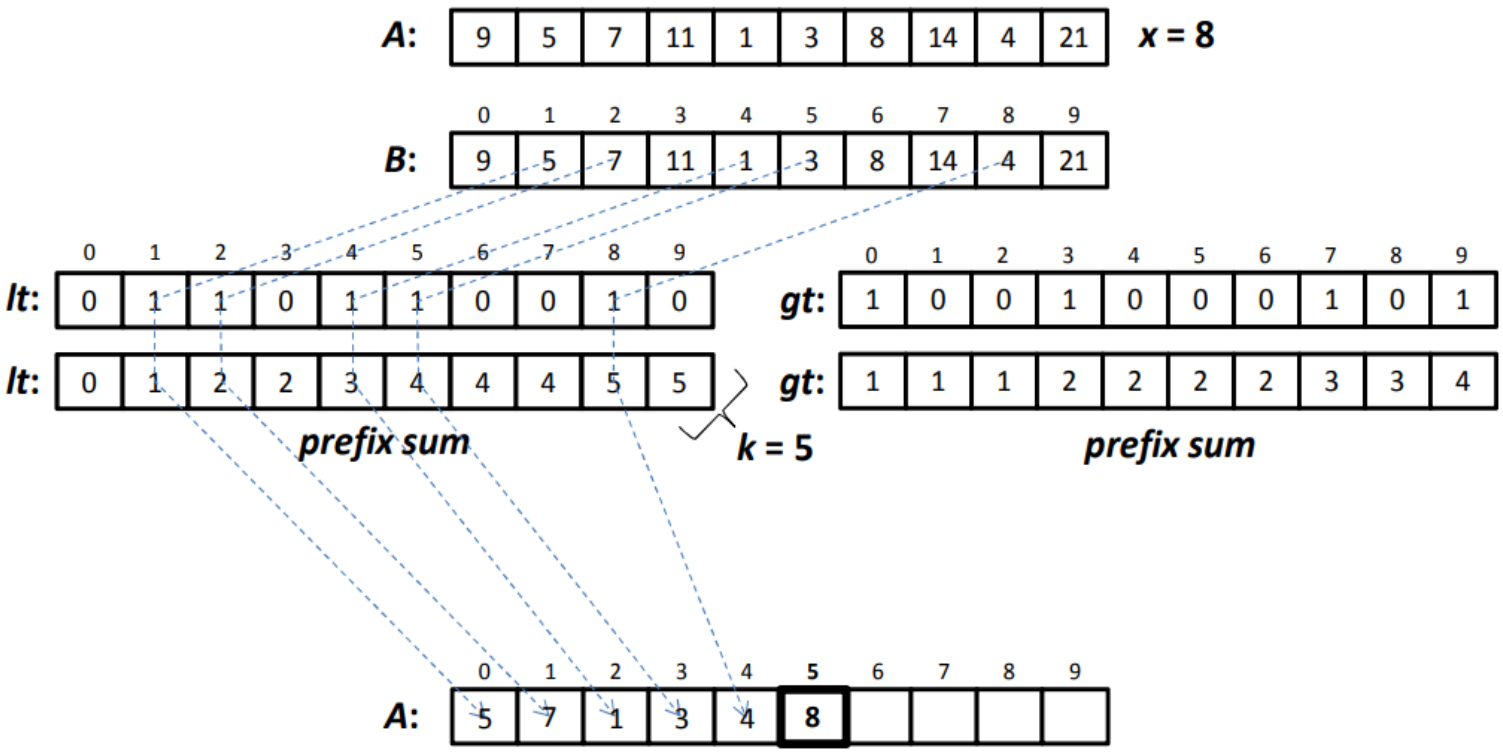


Example: Array Filter

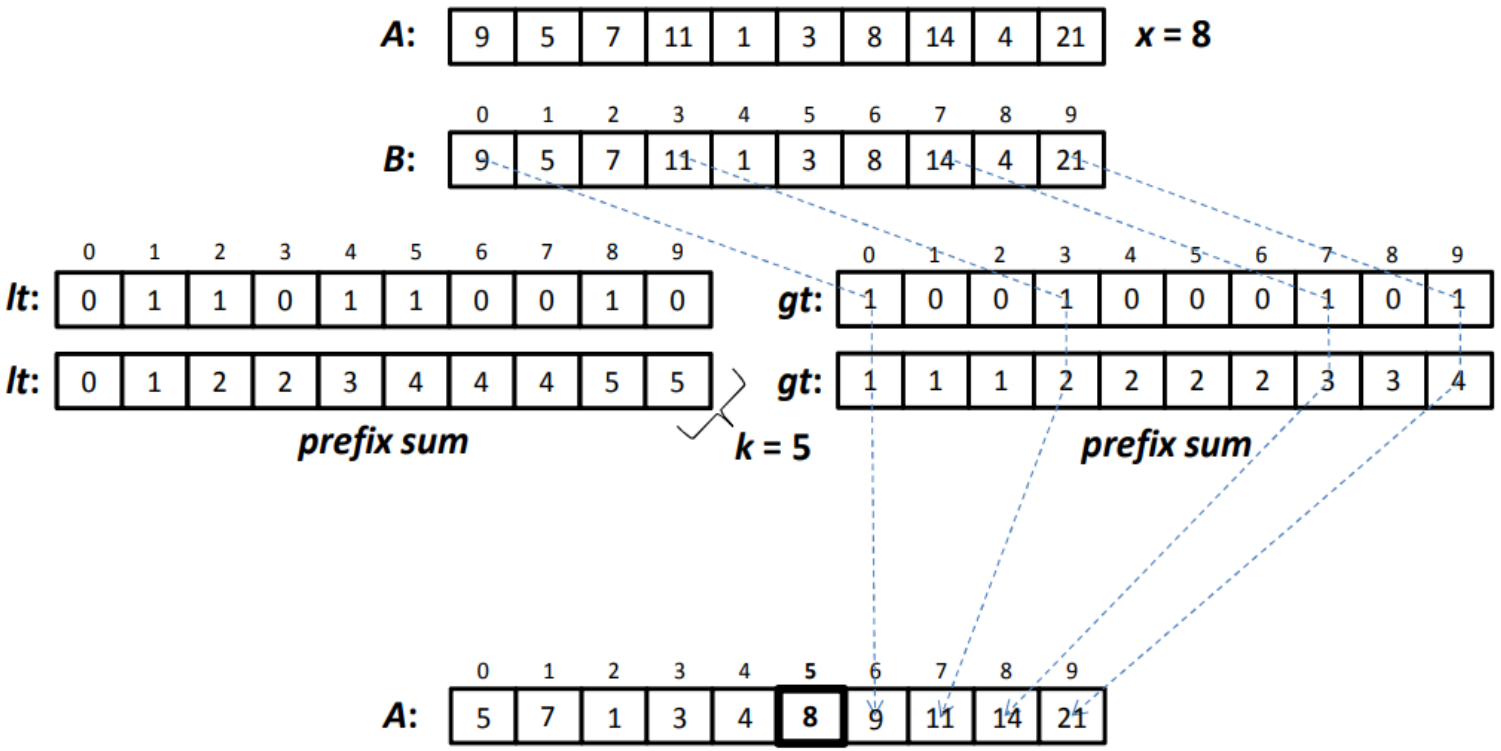
Example: Array Filter



Example: Array Filter



Example: Array Filter



Source Code: Array Filter

Listing 5: Array Filter Implementation

```
int main(){
  int *A, *lt , *gt, ltScan, gtScan, x, N;
  Reading and allocating variables

  compute(lt, x, lower_equal); //In parallel
  compute(gt, x, greater);     //In parallel

  #pragma omp target device (GPU) map(to:lt[:N]) map(from:ltScan[:N])
  ltScan[0] = 0;
  #pragma omp parallel for scan(+:ltScan)
  for(i = 1 ; i < N ; i++)
    ltScan[i] = ltScan[i-1] + lt[i-1];

  #pragma omp target device (GPU) map(to:gt[:N]) map(from:gtScan[:N])
  gtScan[0] = 0;
  #pragma omp parallel for scan(+:gtScan)
  for(i = 1 ; i < N ; i++)
    gtScan[i] = gtScan[i-1] + gt[i-1];

  Rearrange the elements from A, using lt, gt, ltScan and gtScan //In parallel
}
```

Source Code: Array Filter

Listing 6: Array Filter Template Generated

```
#define Neutral 0

__kernel void kernel_0 (__global int *input,\
                        __global int *S, const int n)

__kernel void kernel_1 ( ... )

__kernel void kernel_2 ( ... )
```

Example of Applications

- Random number generation
- Sequence alignment
- N-body problem
- To perform lexical analysis
- Polynomial Evaluation
- To implement Parallel Quick Sort version
- Array Filter
- Solving linear recurrences

A realistic Example: Fibonacci Series

Let

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

And the Fibonnacci numbers, defined by

$$F_0 = 0$$
$$F_1 = 1$$
$$F_{n+1} = F_n + F_{n-1}$$

Then by induction

$$A^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix}$$

And if for n the formula is true, then:

$$A^{n+1} = A.A^n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

$$A^{n+1} = \begin{bmatrix} F_{n+1} + F_n & F_n + F_{n-1} \\ F_{n+1} & F_n \end{bmatrix} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix}$$

Listing 7: Fibonacci Implementation

```
#pragma omp declare target
struct Matrix{
        long x00. x01, x10, x11;
        //Default constructor
        Matrix(){
                x00 = 1; x01 = 1;
                x10 = 1; x11 = 1;
        }
};

Matrix Operator *(Matrix A, Matrix C){
        return A*C;
}

#pragma omp declare target
```

Source Code : Fibonacci Series

Listing 8: Fibonacci Implementation

```
int main ( ) {
  Matrix *x = new Matrix[N];
  Matrix *y = new Matrix[N];
  ...
  #pragma omp target device (GPU) \
              map (to: x[:N]) map (from: y[:N])
  y[0] = Identity_Matrix;
  #pragma omp parallel for scan(*:y)
  for (i = 1; i < N; i++)
    y[i] = y[i - 1] * x[i - 1];

  ...
}
```

Source Code : Fibonacci Series

Listing 9: Fibonacci Template Generated

```
struct Matrix {
  long x00, x01, x10, x11;
};
Matrix *(Matrix A, Matrix C) {
  Mat X;
  X.x00 = A.x00 * C.x00 + A.x01 * C.x10;
  X.x01 = A.x00 * C.x01 + A.x01 * C.x11;
  X.x10 = A.x10 * C.x00 + A.x11 * C.x10;
  X.x11 = A.x10 * C.x01 + A.x11 * C.x11;
  return X;
}
#define Neutral (Matrix){ 1, 0, 0, 1 }

__kernel void kernel_0 (__global Matrix *input,
    __global Matrix *S, const int n)
__kernel void kernel_1 ( ... )
__kernel void kernel_2 ( ... )
```

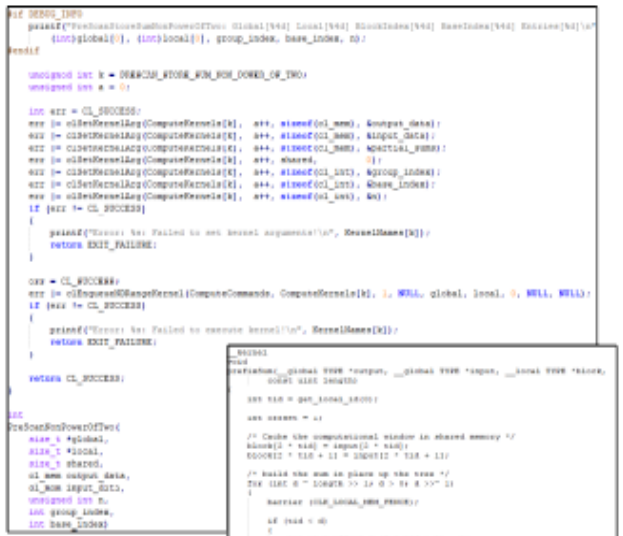
Source Code : Fibonacci Series

Using Scan Clause

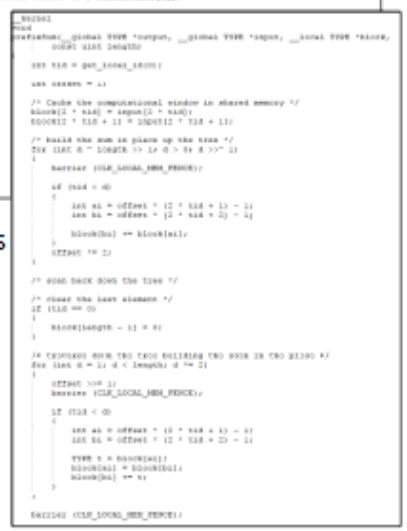
```
int main ( ) {
  Matrix *x = new Matrix[N];
  Matrix *y = new Matrix[N];
  ...
  #pragma omp target device (GPU) \
   map (tofrom: y[:N]) map (to: x[:N])
  #pragma omp parallel for scan ( * : y)
  for (int i = 1; i < N; i++)
    y[i] = y[i - 1] * x[i - 1];
  ...
}
```

main.c ~ 30 lines

OpenCL direct Programming



main.c ~ 92 lines



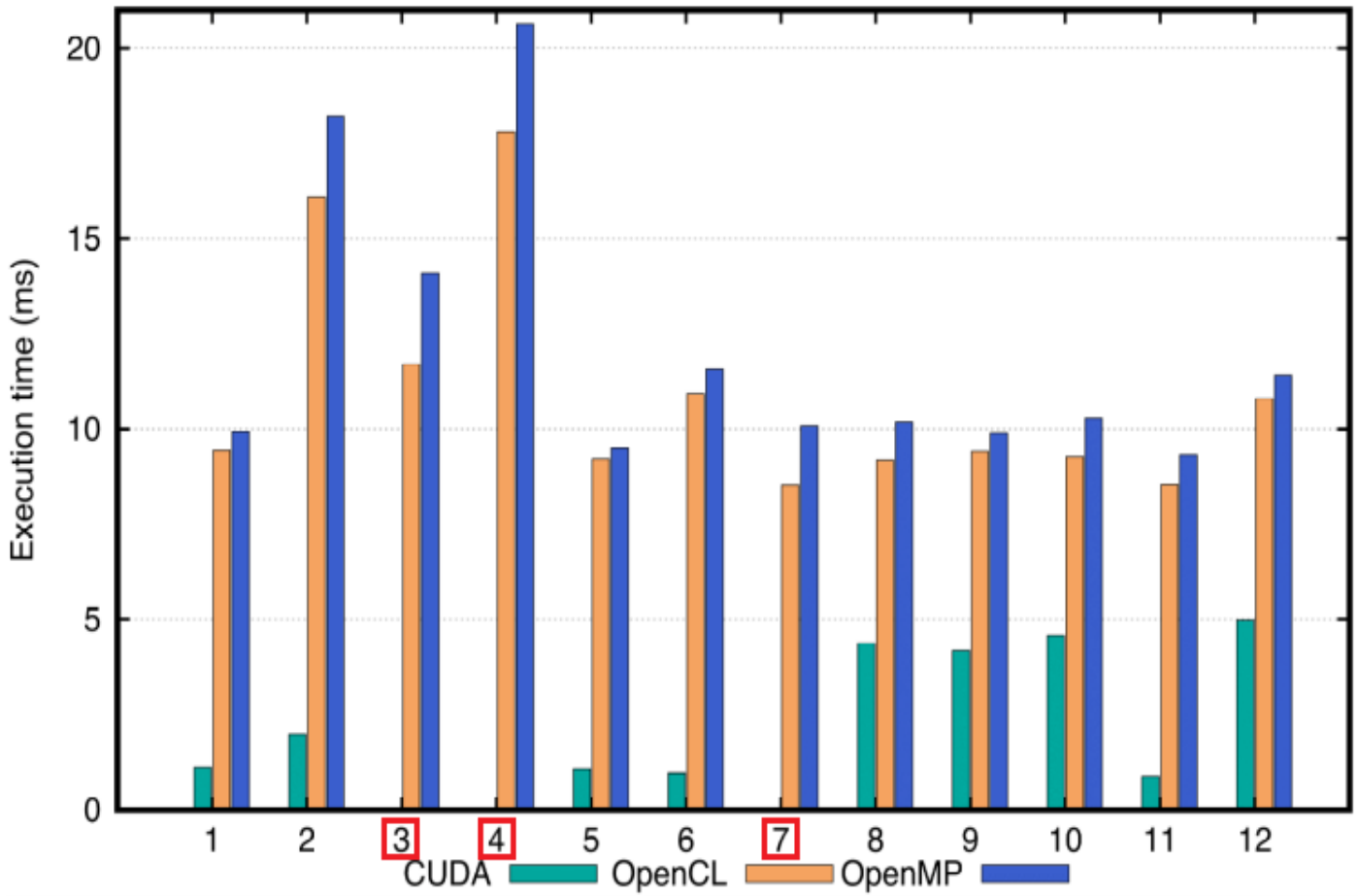
kernel.cl ~ 140 lines

cl_functions.c ~ 100 lines

Total ~ 332 lines

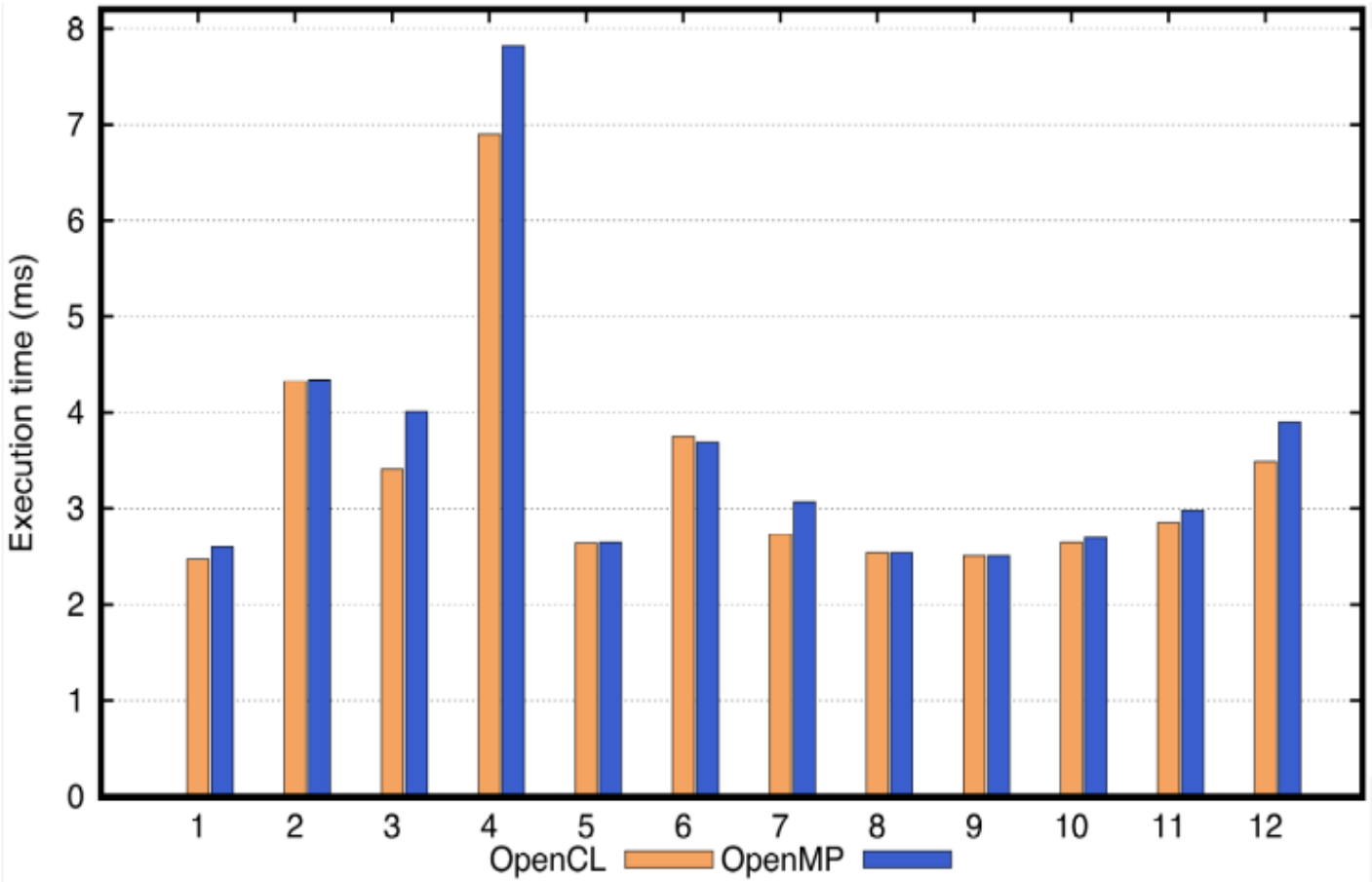
Table of Contents

a) The execution on Intel Xeon CPU E5-2620 with NVIDIA Tesla K40c



Index	Name
1	Stream Compaction
2	Longest Span
3	Polynomial Evaluation
4	Linear Recurrences
5	Random Num Generator
6	Upward Accumulation
7	Adding Big Integers
8	Divide an array in three contiguous parts having equal sum
9	Maximum sum of two non-overlapping subarray
10	Maximum Subarray sum modulo m
11	Maximum occurred integer in n ranges
12	Prime numbers as sum of consecutive primes

b) The execution on Intel Core i5 with Intel Iris GPU



Index	Name
1	Stream Compaction
2	Longest Span
3	Polynomial Evaluation
4	Linear Recurrences
5	Random Num Generator
6	Upward Accumulation
7	Adding Big Integers
8	Divide an array in three contiguous parts having equal sum
9	Maximum sum of two non-overlapping subarray
10	Maximum Subarray sum modulo m
11	Maximum occurred integer in n ranges
12	Prime numbers as sum of consecutive primes

Table of Contents

Conclusion and Future Works

Conclusion:

- The scan operation is a simple and powerful parallel primitive with a broad range of applications;
- The new scan clause in OpenMP exhibits a similar performance as direct programming in OpenCL at a much smaller design effort.

Conclusion and Future Works

Future Works:

- Implement a template of scan algorithm for NVIDIA;
- Extend the limits of scan;
- Implement a template of reduction algorithm for AClang compiler;
- Improve the scan clause the performance by providing specific routines to handle scan operations into the AClang runtime library.

Thanks