Program Synthesis of Parallel Scans

Sagar Jain

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-141
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-141.html

December 16, 2011

Copyright © 2011, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Program Synthesis of Parallel Scans

by Sagar Jain

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

____

Professor Rastislav Bodik
Research Advisor

____

(Date)

* * * * * * *

____

Professor Sanjit A. Seshia
Second Reader

____

(Date)

Abstract

We consider the problem of synthesis of data parallel and task parallel implementations of the scan also known as prefix sum. The user provides an example scan network as an input along with a partial program with missing array index expressions. We then manually extract the constraints for index expressions from the partial program such that the given example network is satisfied. Finally, the generated constraints are solved using linear programming solver. We found that our method is general enough to synthesize data parallel implementations of previously described parallel scan networks.

1 Introduction

Parallel scan is an important primitive operation for developing parallel algorithms. Blelloch [2] showed that many parallel algorithms for radixsort, quicksort, solving recurrence relations, polynomial evaluation etc use prefix sum as one of the building blocks. Given the importance of this operation, parallel scan is provided in standard libraries for many modern parallel platforms. As platforms evolve the programming libraries needs to be updated as well. Many times a different algorithm works better for the different versions of the platform. Thus, it would be very useful to even partially automate the generation of implementations of parallel scan algorithms.

In this report, we looked at the problem of coming up with implementations of parallel scans more efficiently than manual programming, with the help of user's domain knowledge. In the case of parallel scans, we used two pieces of domain information from user – first, the structure of the implementation in the form of a partial program and second, the search space of index expressions that need to be synthesized in the partial program. We successfully demonstrated that implementations of various parallel scan algorithms can be generated from an example network. The programmer can now quickly experiment with various algorithms and choose the best one depending on the architecture. We hope that the user's domain knowledge can be harnessed for applications other than parallel scans as well.

Section 2 gives background on scans and `SKETCH` tool. Section 3 describes the synthesis of data parallel implementation of scans using example networks. In section 4, synthesis of task level parallel implementation for scans is described. The report is then concluded with Results, Related Work, and Conclusion.

2 Background

2.1 Scan Definition

In this section, we formally define the scan operation. The scan function [3] takes as inputs a sequence of numbers x and an associative operator $\oplus$ and outputs another sequence of numbers x' such that

$$x = [x_0, x_1, x_2, \ldots, x_n]$$
$$x' = [x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \ldots, x_0 \oplus x_1 \oplus \ldots x_{n-1} \oplus x_n]$$

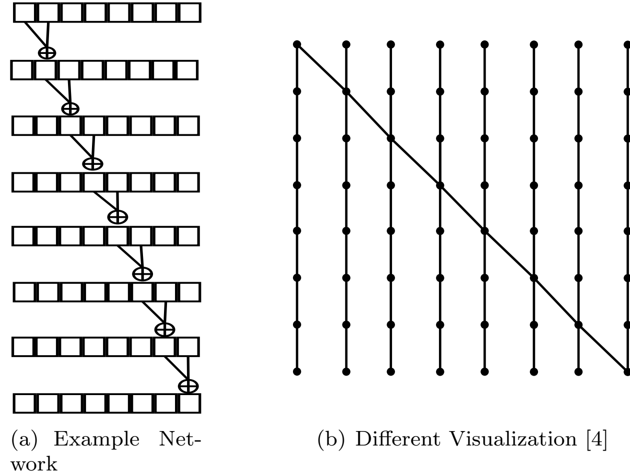
In other words, scan function replaces each element in the input sequence by the fold operation of the associative operator on the prefix sequence (including itself). There are several variants of the scans, which are described below. Let x'_i denotes the ith element in the output sequence and x_i denotes the ith element in the input sequence of length n.

Forward and Backward Scans In Forward Scan, associative operator is applied to all the elements before itself inclusively ,*i.e.* $x'_i = x_0 \oplus x_1 \oplus \ldots x_i$. On the other hand in Backward Scan, associative operator is applied to all the elements following itself, inclusively, *i.e.* $x'_i = x_i \oplus x_{i+1} \oplus \ldots x_n$.

Exclusive and Inclusive Scans In Inclusive Scan, the output element includes itself and its predecessors or successors (depending on forward or backward direction) for the associative operation. In Exclusive Scan, the output element does not include itself and only its predecessors or successors for the associative operation. The first element is normally defined as a zero element but for operators like max and min, it is left to the users for its proper initialization.

In this report, we will focus our discussion only on Forward-Inclusive Scans but the same methodology could be applied for the other three kinds of scans. Also, we fix associative operator as the sum operator throughout the discussion. We will argue later that if synthesis of scans works with the sum operator then it would work with any associative operator.

Figure 1(a) illustrates a way to perform scan on an input sequence of length 8. It describes a sequential way to compute the prefix sum, where in the ith step, element x_i is updated by $x_{i-1} \oplus x_i$. Figure 1(b) describes the same computation but with a different diagram (this notation is taken from [4]). In the later



(a) Example Network

(b) Different Visualization [4]

Figure 1: Sequential Scan Network

diagram, the nodes in the first row denote the input sequence. The values of the nodes flow downwards through the edges and each row signifies one step of computation. The node with exactly one incoming edge means that the value of element is copied, unmodified, from the previous step. The node with more than one incoming edge takes the value given by application of the associative operation on the source nodes of the incoming edges. The outgoing edges distribute the node's value to the target nodes of these edges. It is easy to see for this *example network* [1], that in each step the current element updates its next element with the sum of their values.

We are interested in the synthesis of data parallel algorithms for the scan function. We will consider parallel computations of two kinds:

Single Program Multiple Data (SPMD) where multiple threads execute a single program simultaneously.

Divide and Conquer where a large input is first divided into blocks of smaller inputs and each block is then handled in SPMD style which are later merged in an appropriate way.

We call the former data parallel computation as `Leaf Computation` and the later computation as `Hierarchical Computation`. In modern GPUs, `Leaf Computation` corresponds to the blocks of threads running in a Stream Processor and `Hierarchical Computation` corresponds to the entire kernel function which involves distributing the work among the thread blocks followed by a merging computation.

An example of `Leaf Synthesis` would be the generalization of an example parallel scan network as shown in Figure 3 and producing its correct implementation for the input sequences of sizes other than that used in the concrete example. In `Hierarchical Synthesis`, we are interested in the synthesis of functions f, g and h with respect to the schema shown in Figure 9 and in the generation of the network connections for the blocks with functions f, g and h.

2.2 Sketching

`SKETCH` [5] is a program synthesis system. It takes as input a partial program, which has some missing fragments of code and the desired functional specification. The system then completes the partial program such that the given specification is met. Figure 2 shows a sketch for implementing a function which doubles the input value. In this example, the synthesizer will set '??' to the constant 2. The working of `SKETCH` is described in more detail in Section 6.2.

[1] Example network in our discussion means computation diagram for fixed length input sequence

```
int spec(int x) {
  return x + x;
}

int sketch(int x) {
  return x * ??;
}
```

Figure 2: Sketch for double function

3 Leaf Synthesis

Our synthesis system for SPMD style parallel implementation of scan takes as input an example scan network of a fixed sized input. The output is an implementation satisfying the example scan network for the given and other network sizes. The example scan network illustrates steps of a scan algorithm on a small input sequence just like a trace of an algorithm on an input. The goal is to generalize these sequence of steps so that the resultant program performs exactly the same steps as in the example network for the same sequence size. Moreover, the generated implementation should also work for the inputs of sizes different than that of the given example network size. We do not check the functional correctness of the generated solution during synthesis. We will describe the approach in more detail using some examples.

3.1 Kogge Stone Network

First, we will introduce our encoding of the example scan network. Let us consider the scan network from Figure 3. Whenever two incoming edges meet at a node, it means that the two elements are being get added and the result is stored in the incident node. Such incoming edges are represented by a tuple (i, j) where edges from the nodes i and j meet at the node j. Thus, the representation for the first step in Figure 3 is the following sequence of tuples:

$$\{(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7)\}.$$

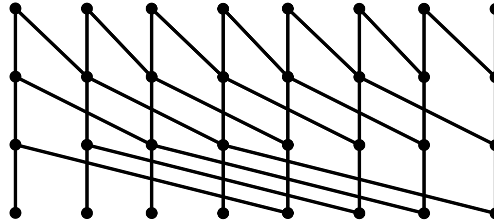


Figure 3: Kogge Stone Network

The complete Kogge-Stone scan network of input size 8 is given below by the nested sequence of pairs E. A sequence of pairs denote a single step of computation in the example network, and to represent all the sequence of computation steps we need a nested sequence of pairs. Let's call the pairs occurring in the representation of the example network as *example pairs*.

$$\begin{aligned} E = \{&\{(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7)\},\\ &\{(0,2),(1,3),(2,4),(3,5),(4,6),(5,7)\},\\ &\{(0,4),(1,5),(2,6),(3,7)\}\} \end{aligned}$$

Let us map the above sequence of tuples in a x-y space with origin at top-left corner, x direction (row) from top to down and y direction (column) from left to right. Thus, x denotes the step index and y denotes the thread index. If we can find functions f and g such that each pair at co-ordinate (i, j) is given by $(f(i, j), g(i, j))$, then our sketch as shown in Figure 4 produces the desired program. The underlined expressions in the sketch, namely $f(i, j)$, $g(i, j)$, and $h(N)$ needs to be synthesized.

```
1  for(int i = 0; i < h(N); i++) {
2    for(int j = 0; j < N; j++) InParallel {
3      if (g(i, j) < N) {
4        a[g(i, j)] = a[f(i, j)] + a[g(i, j)];
5        if (N == 8) {
6          assert(g(i, j) == E[i, j].first);
7          assert(f(i, j) == E[i, j].second);
8          }
9        }
10     }
11 }
```

Figure 4: Sketch for Kogge Stone Network

In the sketch (Figure 4) , the inner loop (lines 2-10) is executed in parallel (SPMD style) by threads. Let us assume that $E[i, j]$ denotes the example pair in the ith row sequence of E and at the jth position in that row sequence. Further, $E[i, j].first$ accesses the first element of that pair and similarly, $E[i, j].second$ accesses the second element of that pair. In addition to meeting the functional specification of the scan, the sketch in Figure 4 also puts constraints that when the size of the input is 8 (lines 5-8), *i.e.* same as in the example network, then the program should exactly follow the steps described in the example network.

Thus, our problem is to find the array index functions (f and g) and loop bound (h). For Kogge-Stone network,

$$
\begin{aligned}
h(N) &= log(N) \\
f(i, j) &= j \\
g(i, j) &= 2^i + j
\end{aligned}
$$

We will now define the search space during synthesis for the functions f, g, and h and our synthesis methodology. We define the program space for the functions f, g and h below as a linear expression with some constraints on coefficients. For our purpose, $term(i, j)$ denotes the set of basic terms, f and g are then defined as a polynomial over all these basic terms and as well as over all the pairs of these basic terms, *i.e.* $term(i, j)_1 \times term(i, j)_2$. The space for the function h is defined as a linear expression over $log(N)$ only, as described below. We discuss in Section 3.4 on how we came up with this space of expressions.

$$
\begin{aligned}
term(i, j) &= \{i, j, 2^i, \lfloor \frac{j}{2^i} \rfloor\} \\
f(i, j) &= \sum_k a_k \times term(i, j) + \sum_{k'} a_{k'} \times term(i, j)_1 \times term(i, j)_2 \\
g(i, j) &= \sum_k b_k \times term(i, j) + \sum_{k'} b_{k'} \times term(i, j)_1 \times term(i, j)_2 \\
h(N) &= \{c_0 \times log(N) \pm c_1 | c_0 \in \{0, 1\}, c_1 \in \{0, 1, 2\}\}
\end{aligned}
$$

To solve for the functions f and g, we need to find the appropriate values for the coefficients of the polynomial. From the example network, we already know the values of the functions f and g at various data points. So, using the known function values at the given points, we will get a system of linear equations. This system of linear equations is then solved using a linear programming solver for the coefficient values to obtain the desired function. Here, we are ignoring the functional correctness and only focusing on solutions

that would satisfy the given example network. Later, for the solution found we validate the solved program on other sequences with different lengths. Currently, we manually extract the constraints from the sketch using the example network and find coefficients in the above expressions. For this example, we will generate constraints for the functions f and g as shown in Figure 5(a) and 5(b) respectively.

```
f(0, 0) = 0, f(0, 1) = 1, f(0, 2) = 2, ... , f(0, 6) = 6
f(1, 0) = 0, f(1, 1) = 1, f(1, 2) = 2, ... , f(1, 5) = 5
f(2, 0) = 0, f(2, 1) = 1, f(2, 2) = 2, f(2, 3) = 3
```

(a) Constraints for 'f'

```
g(0, 0) = 1, g(0, 1) = 2, g(0, 2) = 3, ... , g(0, 6) = 7
g(1, 0) = 2, g(1, 1) = 3, g(1, 3) = 5, ... , g(1, 5) = 7
g(2, 0) = 4, g(2, 1) = 5, g(2, 2) = 6, g(2, 3) = 7
```

(b) Constraints for 'g'

Figure 5: Constraints for Kogge Stone Network

In this particular case of synthesis, we are avoiding generation of constraints as done normally in `SKETCH` system. The reason is that `SKETCH` would simultaneously solve for correctness of the implementation for all sequences with size N and for the resultant implementation to satisfy the example network. In our approach, we are decomposing correctness and satisfaction of example network by first cheaply getting a solution that satisfies the example network, which is later validated for other sequence sizes. Moreover in the case of loops, the constraint generation in `SKETCH` would involve loop unrolling and consist of constraints sequenced for all the iterations. We found that synthesis of array index expressions f and g is cheaper using linear programming solver (Section 5.1). However, this optimization was possible in this case of scan synthesis because firstly, the user has simplified the problem by giving us the trace of the algorithm. Secondly, the only missing pieces are index expressions (*i.e.* we are not looking into synthesis of general program statements).

3.2 Brent Kung Network

Figure 6 shows the Brent Kung network of size 16. The sketch for Kogge Stone network would not satisfy this example network. This is because all the steps in the Brent Kung network do not follow one regular pattern. Though if we consider the first half steps in isolation, the sketch for Kogge Stone network would work for the implementation of the first half of the Brent Kung network. We term contiguous steps which have a regular pattern and can be expressed in the sketch of Kogge Stone network as a *phase*. Following this notion, Kogge Stone is a one-phase algorithm where as Brent Kung is a two-phase algorithm.

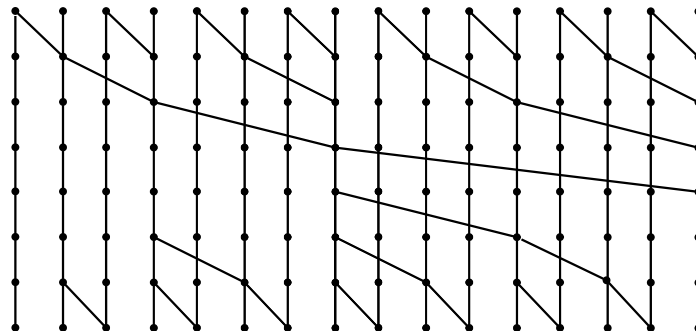


Figure 6: Brent Kung Network

In Brent Kung algorithm, we need two phases for the first and second half of the steps. The complete sketch for the Brent Kung network is shown in Figure 7. Note that for the second phase, value of the variable i decreases in the loop instead of increasing like in the first phase. The reason for that is we would like to

have functions f and g to be monotonically increasing; in principle it could be handled by enriching the space of functions f and g to take monotonically decreasing functions into account as well.

```
for(int i = 0; i < h1(N); i++) {
  for(int j = 0; j < N; j++) InParallel {
  if (g1(i, j) < N) {
    a[g1(i, j)] = a[f1(i, j)] + a[g1(i, j)];
    if (N == 16) {
      assert(g1(i, j) == E1[i, j].first);
      assert(f1(i, j) == E1[i, j].second);
      }
    }
  }
}
for(int i = h2(N) - 1; i >= 0; i--) {
  for(int j = 0; j < N; j++) InParallel {
  if (g2(i, j) < N) {
    a[g2(i, j)] = a[f2(i, j)] + a[g2(i, j)];
    if (N == 16) {
      assert(g2(i, j) == E2[i, j].first);
      assert(f2(i, j) == E2[i, j].second);
      }
    }
  }
}
```

Figure 7: Sketch for Brent Kung Network

We will employ the same strategy as we used for Kogge Stone network. The only difference from the previous case is that now the user has to specify the different phases as well and we assume that the first and the second phase of example network are stored in sets $E1$ and $E2$ respectively. Each of the two phases is then synthesized separately to generate the full Brent Kung algorithm. The synthesized program in the case of Brent Kung is below:

$$h1(N) = h2(N) = log(N)$$
$$f1(i, j) = -1 + 2^i + 2.j.2^i$$
$$g1(i, j) = -1 + 2.2^i + 2.j.2^i$$
$$f2(i, j) = -1 + 2.2^i + 2.j.2^i$$
$$g2(i, j) = -1 + 3.2^i + 2.j.2^i$$

3.3 General Networks

As described in Section 3.2, a phase means a contiguous sequence of steps in the network which can be expressed by a function in the space of polynomials considered by us. We assumed so far that the user is identifying the regular phases in the input example network. We will now relax this constraint for the user to identify the phases in the example network.

We will use a greedy scheme for identification of the phases. At first, we consider only the first step and would try to come up with satisfying functions f and g for the first step. Next, we will consider both the first and second step together to come up with a satisfying f and g for both steps. If there exist satisfiable functions, then it means that the first and second step could be in one phase. If not, then we will start a new phase from the second step. Suppose, we are able to find f and g for the first two steps, we will try to find satisfying functions for the first three steps. If successful, then we will try to see if the fourth step could be included with the first three steps otherwise, we will start a new phase from the third step. We

will continue this process until we reach the end of the steps in the example network and introduce phases as necessary.

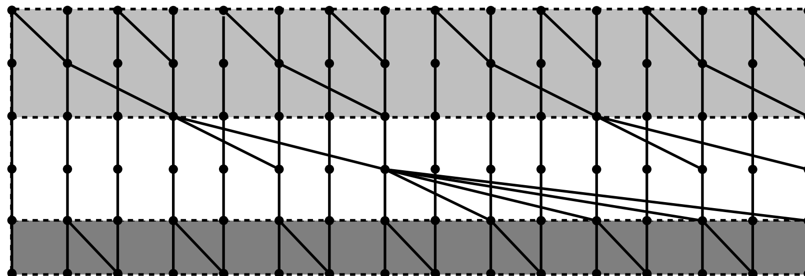


Figure 8: Ladner Fischer Network

Applying this approach on the Brent Kung network would correctly give us two phases. Let us consider now the Ladner–Fischer network in Figure 8. For this example of size 16, the tool identifies three phases – the first phase with size 2 (shaded in light grey), second phase with size 2 (white middle region), and third phase with size 1 (dark grey region). But there is another way the three phases could have been grouped – first and last phase with a single step and the middle phase with all the remaining steps. While both solutions are correct, the greedy approach only identifies the former one. One possible way to get other implementations would be to make additional passes on the phases we identified and try to move down steps from the previous phase to the next phase and see if it can be accommodated. In practice, we did not find this to be a problem and thus, our implementation consists of only the greedy approach. In the worst case, user can always identify the phases if he/she is looking for another implementation from that given by the tool.

3.4 Expression Language

Language of index expressions is important because we want it to be general enough to express many scan networks but not too general so that the search space gets huge. We studied existing prefix adder networks [1] and included expressions in our language to cover characteristics from various networks. For example we observed that stepping through exponentially with step size 2^i is quite common. One other interesting observation was the distribution pattern in which a single value flows to multiple nodes. For expressing a distributor node, we actually needed some sort of periodic function because for all receiving nodes, *i.e.* the second value in the pair, the first value remains same. To handle such cases, we included the function $\lfloor \frac{j}{2^i} \rfloor$. The middle phase of Ladner–Fishcer network (Figure 8) is an instance which involves the distribution pattern.

In the work [1] , author parameterized all the discovered prefix adder networks along the three dimensions namely, logical levels, fanout, and wiring tracks. The three parameters are described below:

Logical Level (l): This parameter characterizes the number of steps in the network. If there are total L number of steps, then l is defined as $l = L - log(N)$.

Fanout (f) : This parameter corresponds to the maximum number of outgoing edges from a node at any step. If there is maximum of F fanout, then $F = 2^f + 1$.

Wiring Track (t): It corresponds to the maximum number of horizontal wirings tracks needed at any step for the lay-out of the nodes. This parameter is for characterizing the lay-out of prefix circuits. If there are maximum of T tracks needed in a step, then $T = 2^t$.

In [1], the taxonomy of the prefix sum networks is characterized by tuple (l, f, t) such that $l + f + t = log(N) - 1$. It is an interesting observation to characterize the prefix sum networks with the meaningful parameters for the case of circuits. Kogge Stone network satisfies $(0, 0, log(N) - 1)$, Brent Kung $(log(N) - 1, 0, 0)$, and Ladner Fischer lies on $(log(N) - 2, 1, 0)$. We considered all families of networks described in this work for size 16 and we were successful in synthesizing SPMD implementation for them.

4 Hierarchical Synthesis

In hierarchical scans, the input sequence is partitioned into blocks of smaller sequences. Figure 9 illustrates the schema we selected for decomposing scans of large sized inputs. The schema has three stages, in the first stage, each of the smaller sized blocks is operated with some function f. In the middle stage, some elements from the result of the second stage are selected and is operated with another function g. Lastly, the results from the second stages along with the original input are merged appropriately with function h to produce the desired scan result. The goal of synthesis in this case is to come up with the functional specifications for f, g, and h and the network connections between the various stages. One example satisfying this schema would be to take function f as reduce, g as scan, and h as scan. For this example, the network connections between the second and the last stage are made by adding elements from the result sequence of the second stage to the first element of the blocks in the third stage.

Once we obtain the functional specifications for f, g, and h. The user can generate an example network and generalize it using the technique described in Section 3. Thus, after synthesizing functional specifications we can get the generalized implementation with some more work, specifically, by developing manually the example networks.

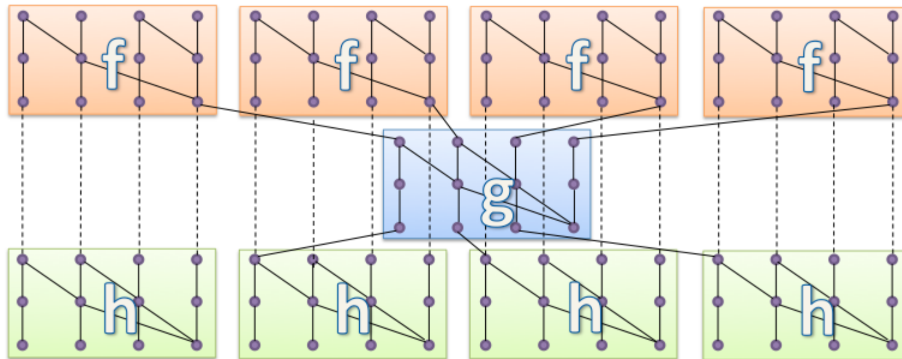


Figure 9: Hierarchical Scan Schema

We parameterize hierarchical scans with three parameters – N = Size of input sequence, K = Number of partitions, and T = Number of elements transferred between the first and second stage per block. For scan in Figure 9, $N = 16, K = 4, T = 1$. We synthesize a task parallel implementation for a particular configuration, *i.e.* parameters N, K, T; using `SKETCH` tool. The partial program for functions f, g, and h corresponding to the three stages as shown in Figure 10. We define the program space of functions f, g, and h by enumerating all the possible partial scans. The sketch in Figure 10 selects for each element a sequence of elements from its predecessors to add to it. The `choose` array in the sketch defines the function. The `choose` array is a large bit array which can be set to either 0 or 1, in turn, deciding which elements to be included.

The higher level sketch (not shown here) first calls function f on each of the input blocks. Then, it is followed by a sketch for choosing T elements from the same position indices of the output blocks of the first stage. Next, there is a sketch for selecting the block inputs from the middle output array and the original array for the final stage. Finally, the function h is called on the blocks in third stage and the end result is

```
for(int i = 1; i< N/K; i++) {
  for(int j = 0; j < i; j++) {
    if(choose[curr_idx]) {
      out[i] += inp[j];
    }
    curr_idx += 1
  }
}
```

Figure 10: Sketch for Block Functions

matched with the final specification. We use SKETCH tool to solve this problem completely , we are not doing any optimization like in previous case of Leaf Synthesis here. One can model the underlying architecture and experiment with parameter N, K and T to find an efficient implementation minimizing a user-defined cost function like the communication cost. In our experiments, we found different specifications of f, g, and h for $N = 16$ while varying the other two parameters. In principle, this could lead to a better implementation previously not explored. Also, such technique can be used for auto-tuning purposes if we could make the synthesis more scalable or could generate the full implementation automatically after obtaining functional specifications.

One last point we would like to make here is that for synthesis of scans with the addition operation, we considered only a single input sequence of fixed length N instead of considering all possible sequences of size N. The input sequence we considered is given by $a_i = 2^i$ for $i = 0 \dots N-1$ and the output sequence should be $a'_i = 2^{i+1} - 1$ for $i = 0 \dots N-1$. Since the sketch has only the addition operation, the output sequence will be correct only when each element is updated with the sum of all the previous elements to it. Thus, if the synthesizer is successful in satisfying this input sequence then it will be correct for any sequence of the same size. Also, the generated implementation will work for any other associative and commutative operator if we replace the sum operator with the desired operator.

5 Results

5.1 Leaf Synthesis

In this section, we will describe our tool's implementation and report evaluation of our synthesis approach for SPMD implementations. We are also interested in the finding if our linear programming based approach performs better than SKETCH which uses SAT solver as a backend to solve the synthesis constraints.

Our tool is implemented in Python and takes as inputs a text file which describes the example scan network and an integer representing the size of the example network. The format of the example scan network is same as the sequence of example pairs notation described in Section 3.1. We used the sympy package for generating the symbolic linear expressions for array index functions. Evaluation of the linear expressions at the known points from the example network is also done using sympy and corresponding symbolic constrants are generated. These constraints are then translated into an Integer Linear Program, which is then passed as input to the glpk library which we use as our linear progam solver.

All the experiments were run on an Intel dual-core 2.8 GHz processor with 3 GB Ram. The tool takes around 2 sec for synthesizing array index expressions using the example scan networks of size 16 as described in [1]. The reported time does not include validation time for the generated implementation. For validation we run the generated implementation on the input with sizes 32, 64, 128, 256, and 512. It takes less than 1 sec to finish all the tests for a single implementation. We found that the synthesized implementations were functionally correct for all of the networks considered by us. However, we found that some of the generated implementations were not functionally correct when we considered the input example network with size 8. Thus, if we use bigger sized input network we are more likely to get the functionally correct implementation.

We also used SKETCH tool directly to synthesize Kogge Stone algorithm from the example network of size 8. We represented the sketch for the index expressions (idxExpr) as:

```
#define idxExpr {|??*offset (+|-) ??*j (+|-) ??*i (+|-) ??*offset*j (+|-)
                   ??*(j/offset)  (+|-) ??*i*(j/offset) (+|-) ??|}
```

In the above definition, `offset` $= 2^i$ and each of the missing values, *i.e.* '??', can take values in the range of $[0, 3]$. The complete input sketch is exactly as shown in Figure 4 but additionally `SKETCH` will verify that the final result array is the scan of the input array with size 8. Note that the sketch for `idxExpr` does not contain complete search space as described in Section 3.1. We found that using the full linear expression search space for array index expressions, `SKETCH` did not terminate within 6 min. The reason being `SKETCH` expands the variable `idxExpr` by enumerating its regular expression, which would be exponential in the number of terms separated by `(+|-)`. When we considered the search space for the array index expressions exactly as described earlier, `SKETCH` still timed out in 6 min. We believe that it is due to the verification of the output array. To validate our hypothesis, we ignored the functional correctness and only solved for the example scan correctness, for this case `SKETCH` took around 7 sec. The size of the input example network in `SKETCH` is smaller compared to the size considered in our approach. Using the same input network size 16 and solving only for example network correctness, `SKETCH` takes around 22 sec. We can say that our approach works better but the main point is we avoided the functional correctness and satisfied the example constraints. It would be interesting to come up with a general way of isolating constraints which can be solved efficiently through a dedicated solver.

5.2 Hierarchical Synthesis

In this section, we will describe our experience in terms of discovering decomposition strategies with the hierarchical synthesis. As discussed in Section 4, the input sketch for hierarchical scan has three parameters N, K, and T. We are interested in decompositions which can be generalized into algorithms and work for sequences with sizes other than that of the input sequence.

Table 1 shows the results of various configurations run on an Intel dual-core 2.8 GHz processor with 3 GB Ram. We did not put any additional constraints on the sketch like minimization of the writes to the memory etc. We found that when we increased value of N from 16 to 32, the synthesis did not finish because of insufficient memory. We would like to handle input sequences with larger N because the decomposition algorithm will become more apparent than performing synthesis on smaller values of N. In our experiments, we did discover various networks but their generalizations was not clear using N as 16. Thus, it would be very interesting if the hierarchical synthesis could be made more scalable which could potentially lead to discovery of better algorithms.

N	K	T	Time (sec)
16	2	1	7
16	2	2	11
16	4	1	5
16	4	2	7

Table 1: Hierarchical Scan Results

6 Related Work

6.1 Parallel Scan

Parallel scans have been extensively studied both in the contexts of prefix sum circuits and programs for parallel architectures. The prefix adder circuits are well summarized in [1]. In [6], the authors provided an efficient Brent Kung data parallel implementation for GPUs including both inclusive and exclusive scans. They also gave a segmented scan implementation, in segmented scan the input is a sequence with partitions

(segments). The scan operation is confined to each of the segments in the input array instead of the entire array as in normal scan operation. Our methodology can synthesize Brent Kung network from the example network (Section 3.2). However, our generated implementation as opposed to given implementation in [6] does not include any low-level optimizations like avoiding memory bank conflicts.

Recent work [7] experimented with the task level parallel implementations of scans for GPUs. The authors systematically validated their various hypotheses on performance of scans on GPUs. They then pruned the search space of the optimal programs by applying their hypotheses and performed experiments manually in the pruned space to find the best program. Our synthesis of hierarchical scans would be a good fit for such scenarios like auto-tuning where the users have enough idea of the search space of the interesting programs. Synthesis could accelerate the process of finding the optimal programs by using additional constraints like minimizing user-defined cost functions.

6.2 Sketching

`SKETCH` is a program synthesis tool and has been demonstrated to be successful in various domains like bit-vector programs, sequential and concurrent programs [5]. The input to the tool is a partial program with missing pieces of code and the desired functional specification. The key idea is the use of Counter Example Guided Iterative Synthesis (CEGIS). There is a synthesizer and a verifier and they complement each other. First, the synthesizer completes the partial program such that it satisfies any random input and the verifier then verifies if this solution holds for all other inputs. If not, then the verifier passes that counter example to the synthesizer. The synthesizer then tries to find the solution which will satisfy previous inputs as well as the new counter example and passes its solution to the verifier again. The cycle stops when either the verifier finds no counter example, in this case `SKETCH` is successful in synthesizing the program. If the synthesizer fails to return a solution satisfying the latest counter example as well as the previous inputs meaning that there is no possible correct program in the search space given by the user.

We used the idea of partial programs introduced in `SKETCH` to describe the user's domain knowledge about the structure of the parallel scan implementations. We used `SKETCH` directly in the synthesis of hierarchical scans. It can be used in the synthesis of SPMD implementations as well but we found it better to use a linear programming solver.

6.3 Angelic Programming

Angelic Programming [8] is a programming methodology to help the programmers to discover algorithms. Similar to `SKETCH`, it takes as inputs a sketch and a functional specification. The key idea is the introduction of a '`choose`' (!!) construct. The `choose` construct can select different values for different instances of control reaching the same program point. For example if inside a loop, we have a '!!' operator, then it can take different values in different iterations of the loop. This is helpful in case when the user is in the process of discovering an algorithm. Thus, he can start with a sketch with many '!!' operators in it and gradually refine the sketch to a correct program. The exploration of various options in '!!' operator is implemented through backtracking mechanism as described in [8]. In the context of scan, discovering an example network can be done through refinement using Angelic Programming and later, its implementation can be derived using our methodology.

7 Conclusion

We looked into the problem of synthesis of data parallel and task level parallel implementation of scans also known as prefix sum. We identified that in the case of a given example scan network, synthesis of index expressions just involved solving of a linear program. In other words, we demonstrated that with the user's domain knowledge, SPMD programs can be synthesized in an efficient manner. It would be interesting to see if one can describe a more general way of identifying parts of the problem suited for different kinds of constraint solvers rather than using a more general constraint solver for the entire problem. Our hope is that

this approach could be useful in other contexts, for example many parallel implementations involve tricky index expressions for distributing work among the threads and loading and storing of results.

Our greedy approach exploits regularity in the example instance and is very brittle with respect to any irregularity. For example if every alternate instruction is generalizable but not any two consecutive instructions then our strategy would fail. One fix to handle irregularity is to introduce conditional statements in the language of the target programs but the greedy strategy was able to handle all the networks used in our study.

The main limitation is the manual extraction of constraints for array index expressions from the sketch of parallel scan. We found that `SKETCH` tool did not perform well with linear expression constraints but if the extraction of such constraints can be automated and solved with a dedicated solver, we can hope for overall improved performance. This could make `SKETCH` more scalable and more useful as it can synthesize general program statements.

Our hierarchical synthesis sketch could handle input sequences with size 16 but does not scale for bigger sized input like 32. In our sketch, we used a simple integer encoding but we believe that the input sketch could be encoded in a better way like with bit-vector encoding to make the approach more scalable. Specifically the input sequence of size N considered by us is given by $[2^0, 2^1, \ldots, 2^{N-1}]$, for this sequence the addition operation between any two elements is an `OR` operation. Moreover, for the correct scan no single element can be ever added twice, so the `AND` of the two elements to be added should always be 0. This optimization in encoding makes use of the properties of the scan specification as well as the single input sequence.

8 References

[1] D. Harris, "A taxonomy of parallel prefix networks," in *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, vol. 2, pp. 2213–2217, IEEE, 2003.

[2] G. E. Blelloch, "Prefix sums and their applications," *Computer*, no. 7597, pp. 35–60, 1990.

[3] "Prefix sum," *http://en.wikipedia.org/wiki/Prefix_sum*.

[4] R. Hinze, "An algebra of scans," in *In Mathematics of Program Construction*, pp. 186–210, Springer, 2004.

[5] A. Solar Lezama, *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[6] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for GPUs," *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, no. 1, pp. 1–17, 2008.

[7] D. Merrill and A. Grimshaw, "Parallel scan for stream architectures," *University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14*, no. December 2009, pp. 1–54, 2009.

[8] R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor, "Programming with angelic nondeterminism," *ACM Sigplan Notices*, vol. 45, no. 1, p. 339, 2010.