Parallel Programming with Inductive Synthesis

Shaon Barman, Ras Bodik,
Sagar Jain, Yewen Pu,
Saurabh Srivastava,
Nicholas Tung

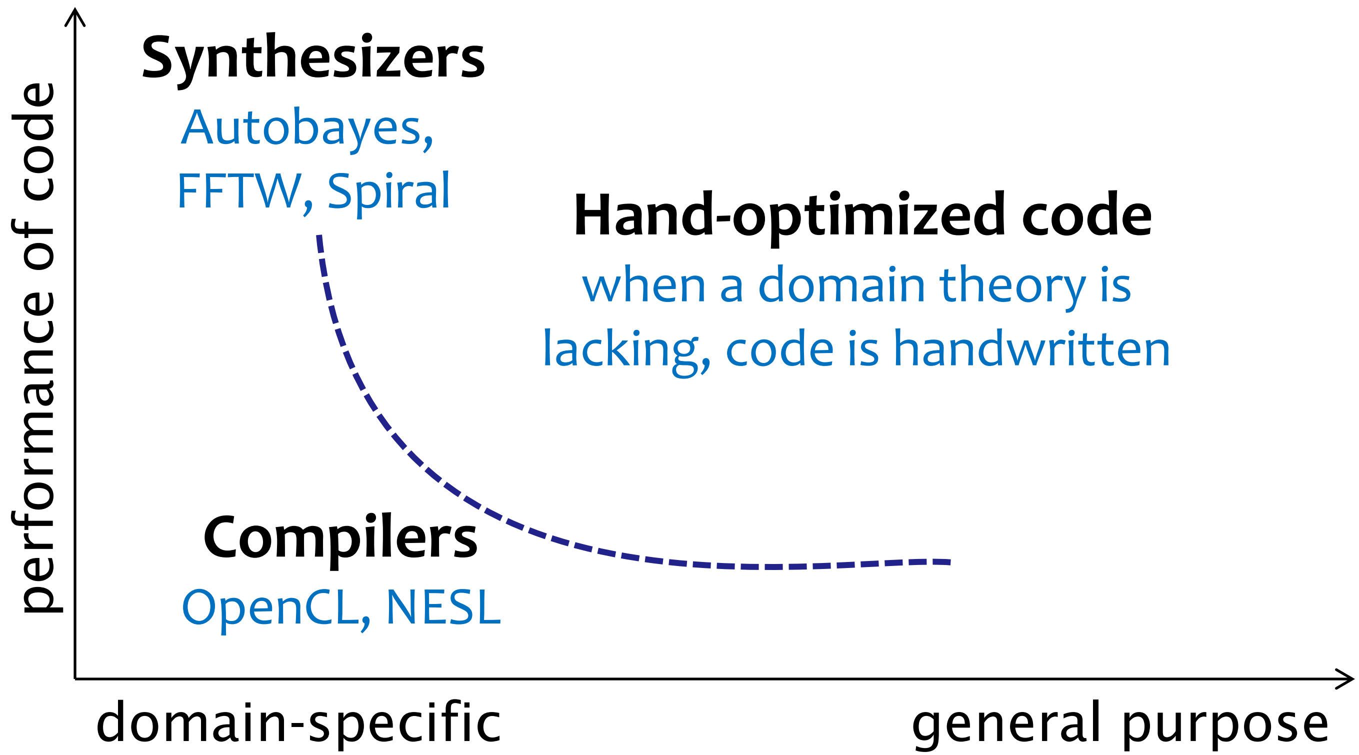
UC Berkeley
ParLab

with help from
Armando Solar-Lezama, MIT

**Once you understand how to write a program,
get someone else to write it.**

Alan Perlis, Epigram #27

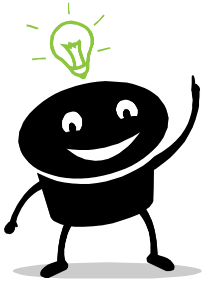
What's between compilers and synthesizers?



Our approach: *help programmers auto-write code without (us or them) having to invent a domain theory*

The HPC Programming ~~Challenge~~ Opportunity

Automating code writing



SKETCH

The SKETCH Language

try it at bit.ly/sketch-language

SKETCH: just two constructs

spec:

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

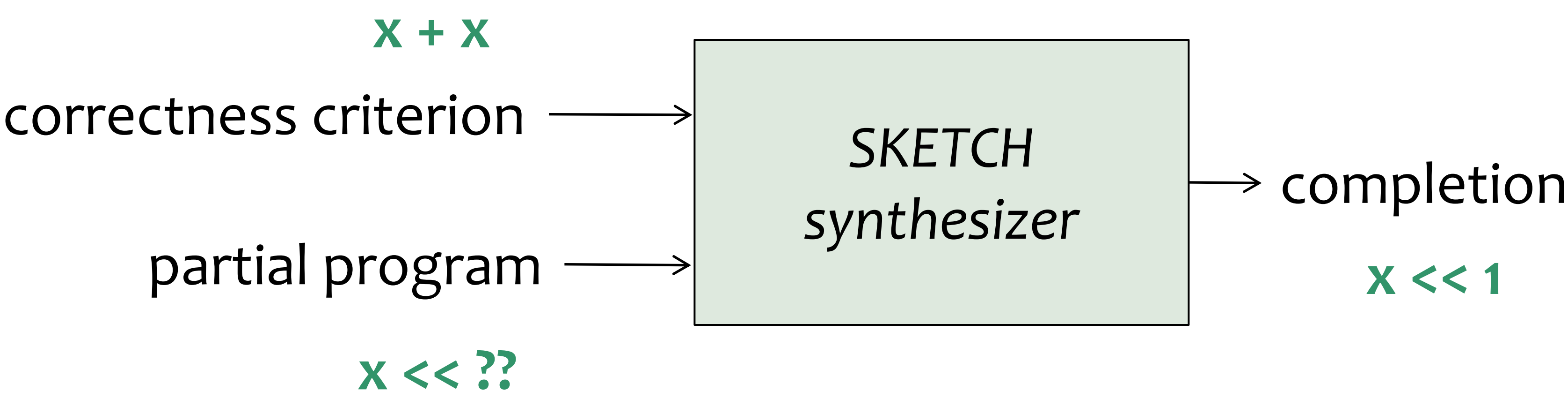
sketch:

```
int bar (int x) implements foo {
    return x << ??;
}
```

result:

```
int bar (int x) implements foo {
    return x << 1;
}
```

SKETCH is synthesis from partial programs



No need for a domain theory. No rules needed to rewrite
x+x into **2*x** into **x<<1**

Example: Silver Medal in a SKETCH contest

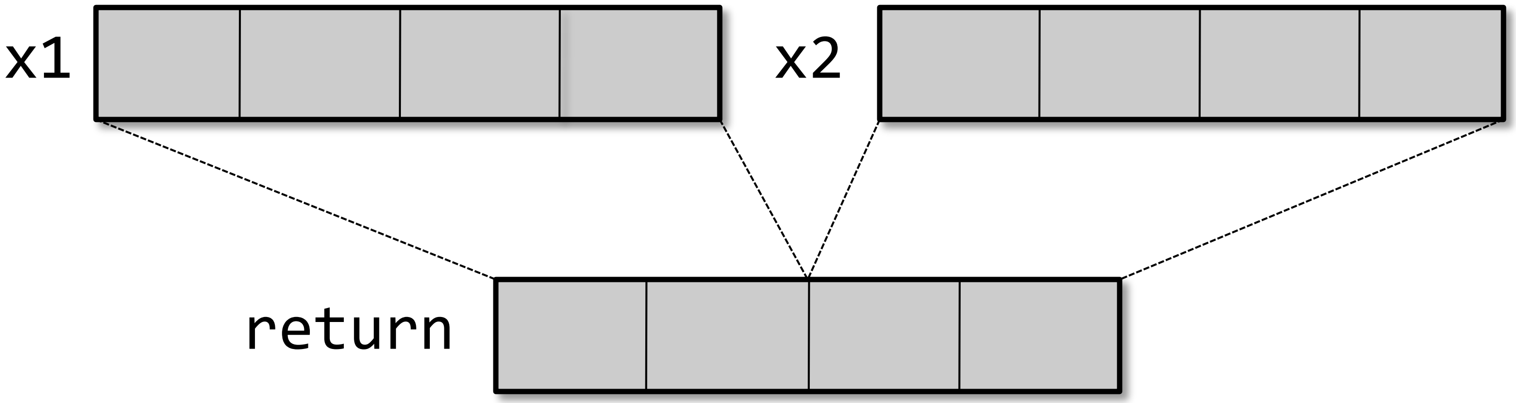
4x4-matrix transpose, the specification:

```
int[16] trans(int[16] M) {
  int[16] T = 0;
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      T[4 * i + j] = M[4 * j + i];
  return T;
}
```

Implementation idea: parallelize with SIMD

Intel shufps SIMD instruction

SHUFP (shuffle parallel scalars) instruction



The SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
  repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
int[16] trans_sse(int[16] M) implements trans { // synthesized code
  S[4::4]   = shufps(M[6::4],   M[2::4],  11001000b);
  S[0::4]   = shufps(M[11::4],  M[6::4],  10010110b);
  S[12::4]  = shufps(M[0::4],   M[2::4],  10001101b);
  S[8::4]   = shufps(M[8::4],   M[12::4], 11010111b);
  T[4::4]   = shufps(S[11::4],  S[1::4],  10111100b);
  T[12::4]  = shufps(S[3
  T[8::4]   = shufps(S[4
  T[0::4]   = shufps(S[1
}
```

From the contestant email:

`Over the summer, I spent about 1/2 a day manually figuring it out.`

Synthesis time: 30 minutes.

Beyond synthesis of constants

Holes can be completed with more than just constants:

Array index expressions:	`A[ ??*i+??*j+?? ]`
Polynomial of degree 2:	`??*x*x + ??*x + ??`
Initialize a lookup table:	`table[N] = {??,…,??}`

The price SKETCH pays for generality

What are the limitations behind the magic?

Sketch doesn't produce a proof of correctness:

SKETCH checks correctness of the synthesized program on all inputs of up to certain size. The program could be incorrect on larger inputs. This check is up to programmer.

Scalability:

Some programs are too hard to synthesize. We propose to use refinement, which provides modularity and breaks the synthesis task into smaller problems.

Interactive synthesis with refinement

Automatic functional equivalence checking

enabled by recent advances in program analysis, testing

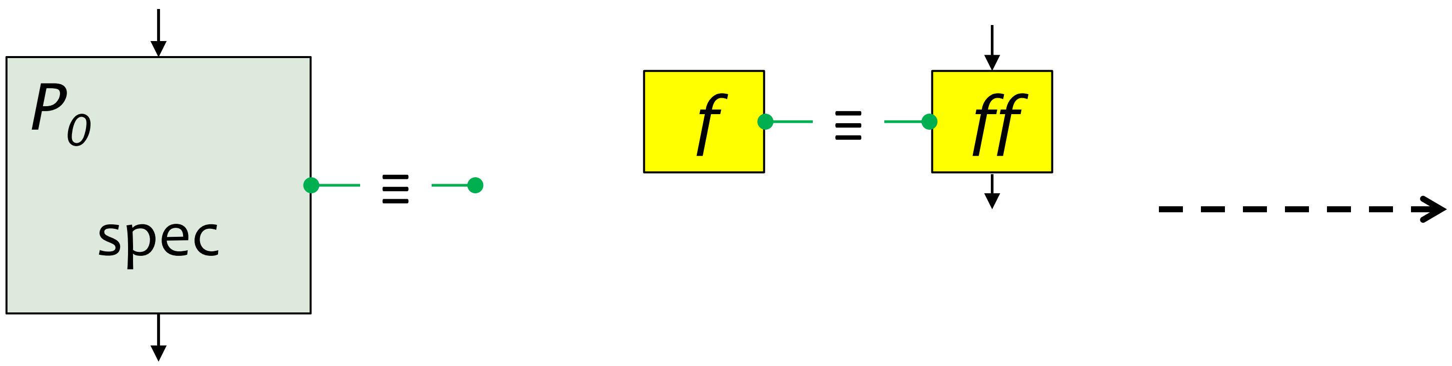
Sketch-based synthesis

automatically generate details of tricky algorithms

Autotuning and algorithm design space exploration

search design spaces you could never consider by hand

These ingredients allow Refinement



Refinement is already a popular form of development

automation and language support can make it better

Refinement

Sequential code

↓

Parallel code using naïve shared memory

↓

Two-level parallel code using naïve shared memory

↓

Two-level parallel code with shared memory within blocks and MPI across

- How do I break the task into parallel units of work?
- How do I synchronize them?

- How do I group threads into blocks?
- How do I reduce interaction among different blocks?

- How do I partition my data into independent pieces?
- How do the pieces communicate?

Refinement

Sequential code

↓

Parallel code using naïve shared memory

↓

Two-level parallel code using naïve shared memory

↓

Two-level parallel code with shared memory within blocks and MPI across

Automatic Validation prevents bugs from being introduced after each refinement

Refinement

Sequential code

Parallel code using naïve shared memory

Two-level parallel code using naïve shared memory

Two-level parallel code with shared memory within blocks and MPI across

Synthesis helps derive the details of each refinement

Refinement

Sequential code

Parallel code using naïve shared memory

Two-level parallel code using naïve shared memory

Two-level parallel code with shared memory within blocks and MPI across

Each refinement step produces a space of possibilities that autotuning can explore

HPC Scenarios

Domain scientist:

problem spec --> dynamic programming --> parallel scan

Parallel algorithm expert:

example of parallel scan network --> SIMD algorithm

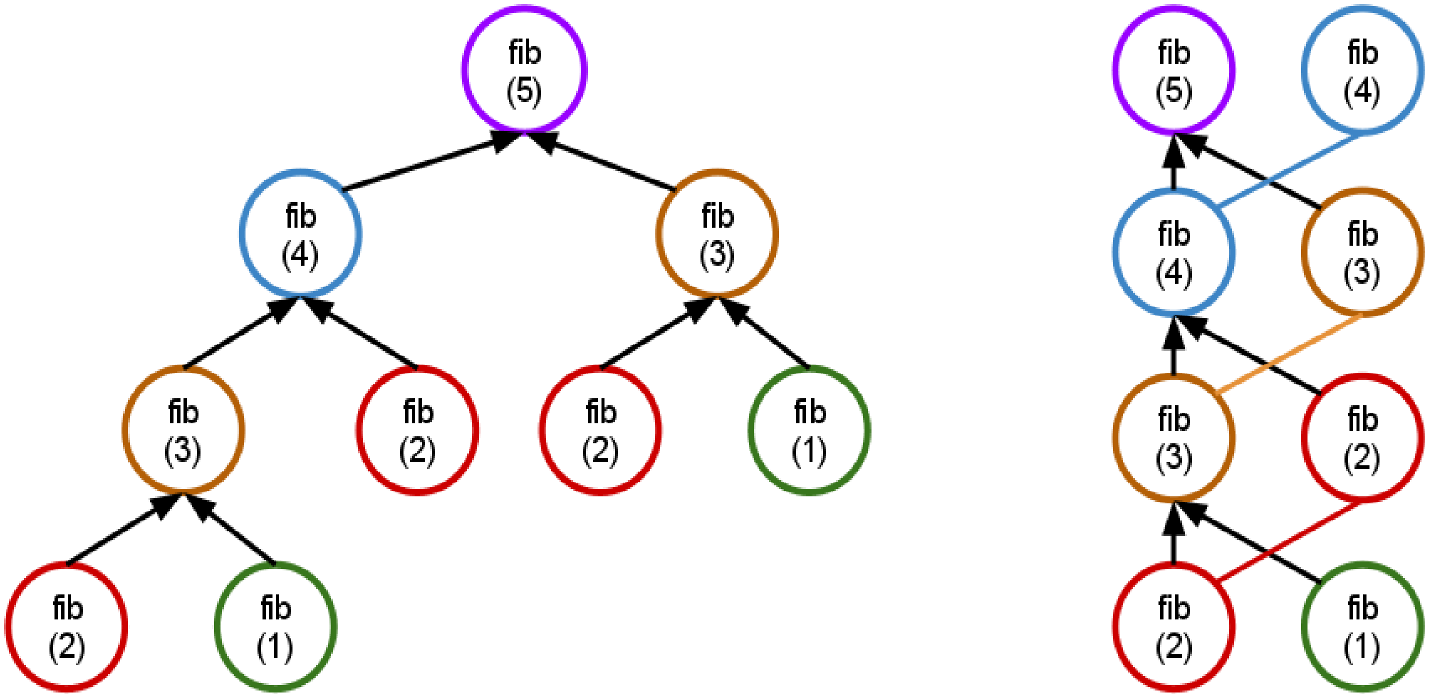
GPU tuning expert:

SIMD algorithm --> bank conflicts --> index expressions

Dynamic Programming

Compute $O(2^n)$ algorithms in $O(n^k)$ time

Example: *fib(n)*



Challenges in DP algorithm design

The divide problem: Suitable sub-problems often not stated in the original problem. We may need to invent different subproblems.

The conquer problem: Solve the problem from subproblems by formulate new recurrences over discovered subproblems.

Maximal Independent Sum (MIS)

Given an array of positive integers, find a non-consecutive selection that returns the best sum and return the best sum.

Examples:

mis([4,2,1,4]) = 8
mis([1,3,2,4]) = 7

Exponential Specification for MIS

The user can define a specification as an clean exponential algorithm:

```
mis(A):
   best = 0
   forall selections:
      if legal(selection):
         best = max(best, eval(selection, A))
   return best
```

Sketch = "shape" of the algorithm

```
def linear_mis(A):
  tmp1 = array()
  tmp2 = array()
  tmp1[0] = initialize1()
  tmp2[0] = initialize2()
  for i from 1 to n:
    tmp1 = prop1(tmp1[i-1],tmp2[i-1],A[i-1])
    tmp2 = prop2(tmp1[i-1],tmp2[i-1],A[i-1])
  return term(tmp1[n],tmp2[n])
```

Synthesize propagation functions

```
def prop (x,y,z) :=
    switch (??)
    case 0: return x
    case 1: return y
    case 2: return z
    case 3: return unary(prop(x,y,z))
    ...
    case r: return binary(prop(x,y,z),
                          prop(x,y,z))
```

MIS: The synthesized algorithm

```
linear_mis(A):
  tmp1 = array()
  tmp2 = array()
  tmp1[0] = 0
  tmp2[0] = 0
  for i from 1 to n:
    tmp1[i] = tmp2[i-1] + A[i-1]
    tmp2[i] = max(tmp1[i-1],tmp2[i-1])
  return max(tmp1[n],tmp2[n])
```

A guy walks into a Google Interview …

Given an array of integers $A=[a_1, a_2, \ldots, a_n]$,
return $B=[b_1, b_2, \ldots , b_n]$
such that: $b_i = a_1 + \ldots + a_n - a_i$

Time complexity must be $O(n)$

Can't use subtraction

Google Interview Problem: Solution

```
puzzle(A):
  B = template1(A)
  C = template2(A,B)
  D = template3(A,B,C)
  return D

template1(A):
  tmp1 = array()
  tmp1[0] = 0
  for i from 1 to n-1:
    tmp1[i] = tmp[i-1]+A[n-1]
  return tmp1
```

```
template2(A,B):
  tmp2 = array()
  tmp2[n-1] = 0
  for i from 1 to n-1:
    tmp2[n-i-1]
            = tmp2[n-i]+A[n-i]

template3(A,B,C):
  tmp3 = array()
  for i from 0 to n-1:
    tmp3[i] = B[i] + C[i]
  return tmp3
```

HPC Scenarios

Domain expert:

problem spec --> dynamic programming --> parallel scan

Parallel algorithm expert:

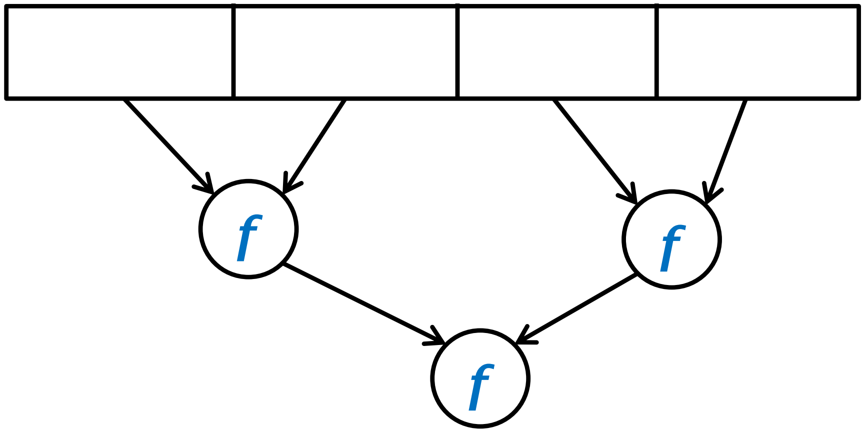
example of parallel scan network --> SIMD algorithm

GPU tuning expert:

SIMD algorithm --> bank conflicts --> index expressions

Parallelizing with Synthesis

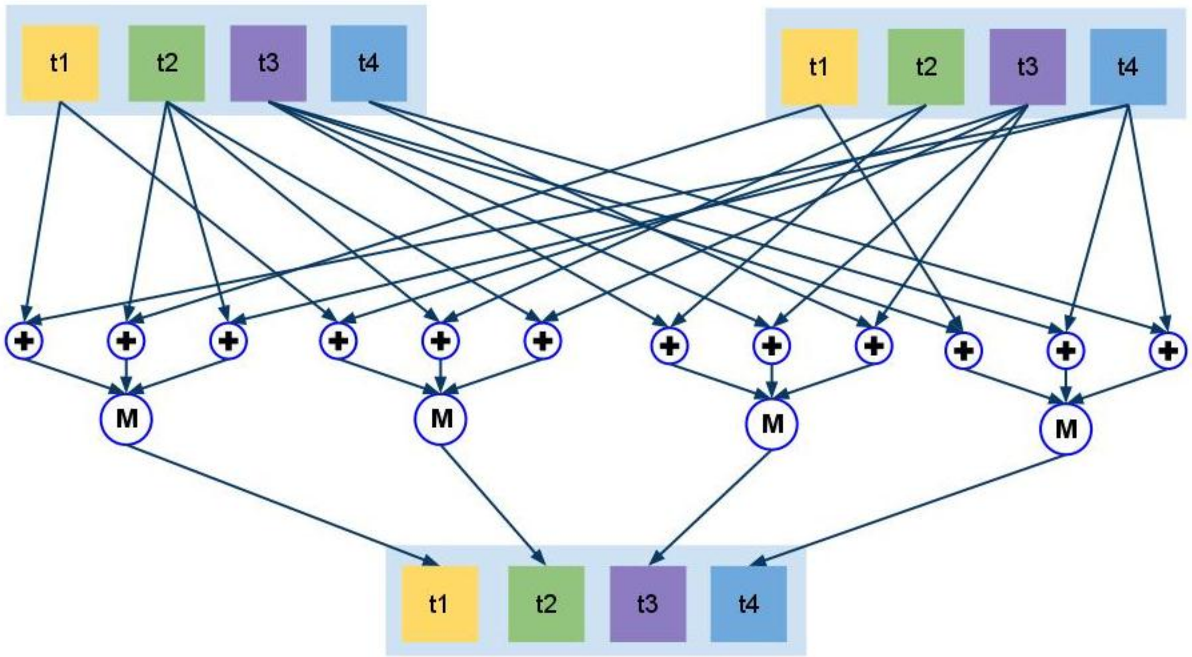
Goal: synthesize an *associative* function that allows solving the problem in parallel, as a prefix sum.



The sketch: force the function to work on a tree:

```
result = prop(prop(A[0],A[1]),
              prop(A[2],A[3]))
```

Synthesizes associative operator for MIS



Evan: "It is quite exciting that I do NOT have a good idea of what the synthesizer returned (kind of magic!)"

HPC Scenarios

Domain expert:

problem spec --> dynamic programming --> parallel scan

Parallel algorithm expert:

example of parallel scan network --> SIMD algorithm

GPU tuning expert:

SIMD algorithm --> bank conflicts --> index expressions

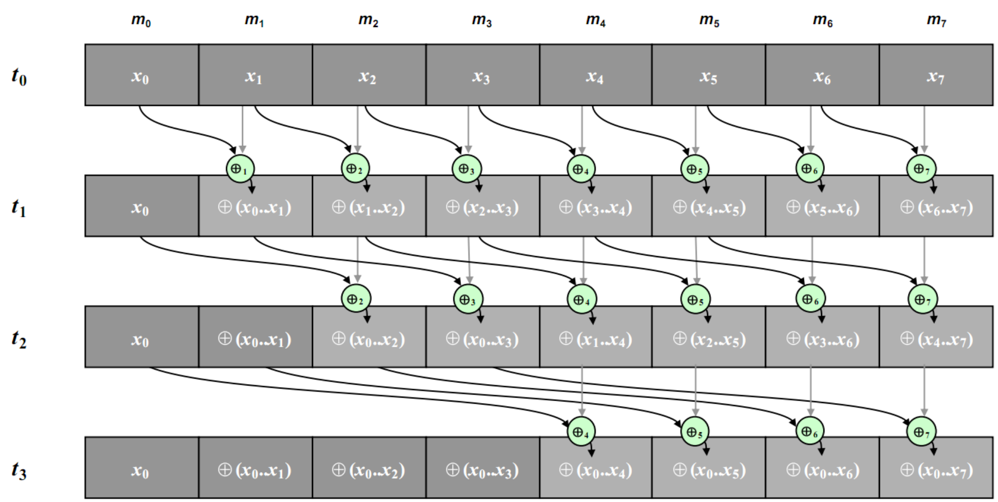
Why scans?

Many practical algorithms use scans [Blelloch '90]

- lexically compare string of characters; lexical analysis
- evaluate polynomials
- radix sort, quicksort
- solving tridiagonal linear systems
- delete marked elements from an array
- search for regular expressions
- tree operations
- label components in two dimensional images
- dynamic programming (see Evan Pu's poster)

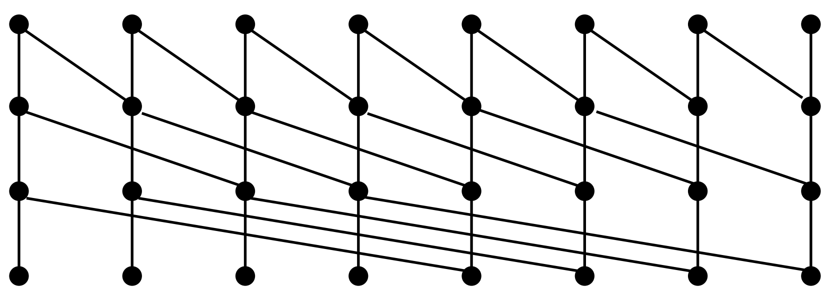
Many problems are sums with some assoc operator

Implementing scans



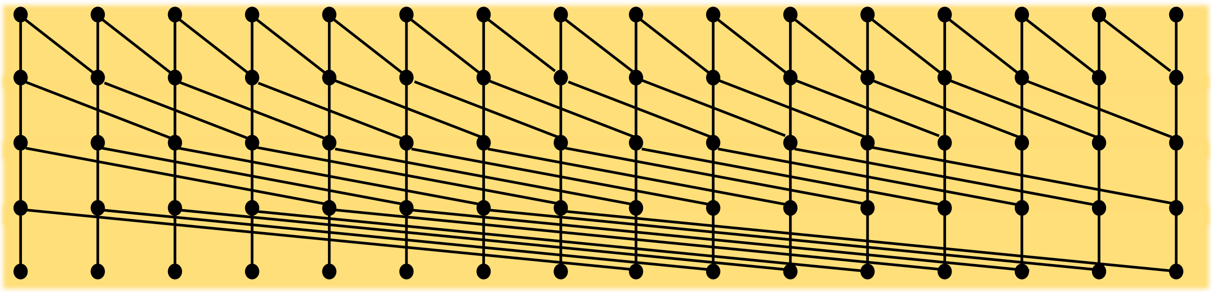
N = 8

instance of parallel scan algorithm



its abstract visualization

SIMD execution of scan algorithms



HPC Scenarios

Domain expert:

problem spec --> dynamic programming --> parallel scan

Parallel algorithm expert:

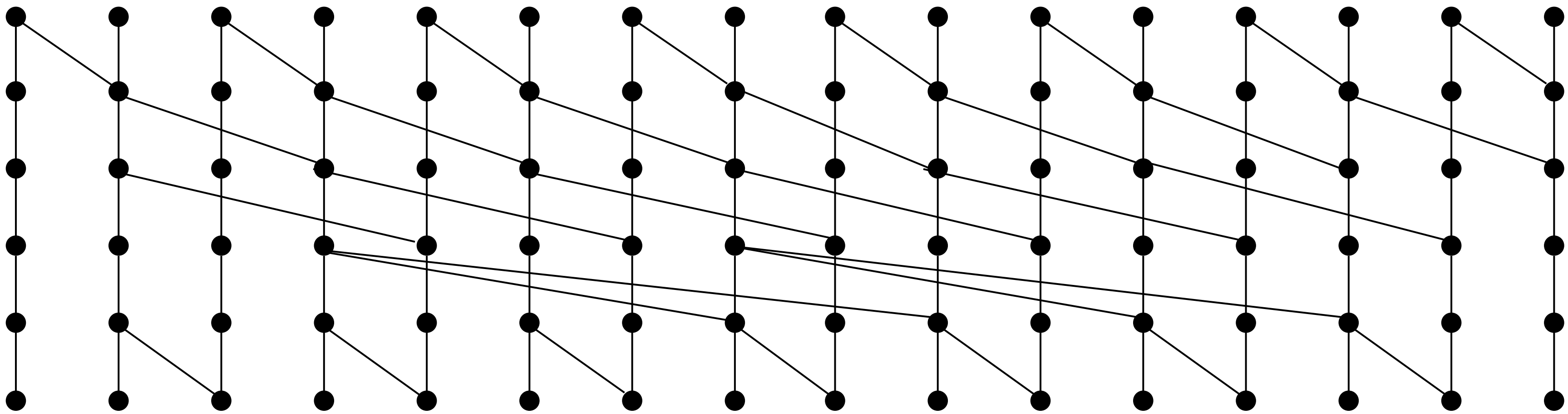
example of parallel scan network --> SIMD algorithm

GPU tuning expert:

SIMD algorithm --> bank conflicts --> index expressions

Example Scan Network

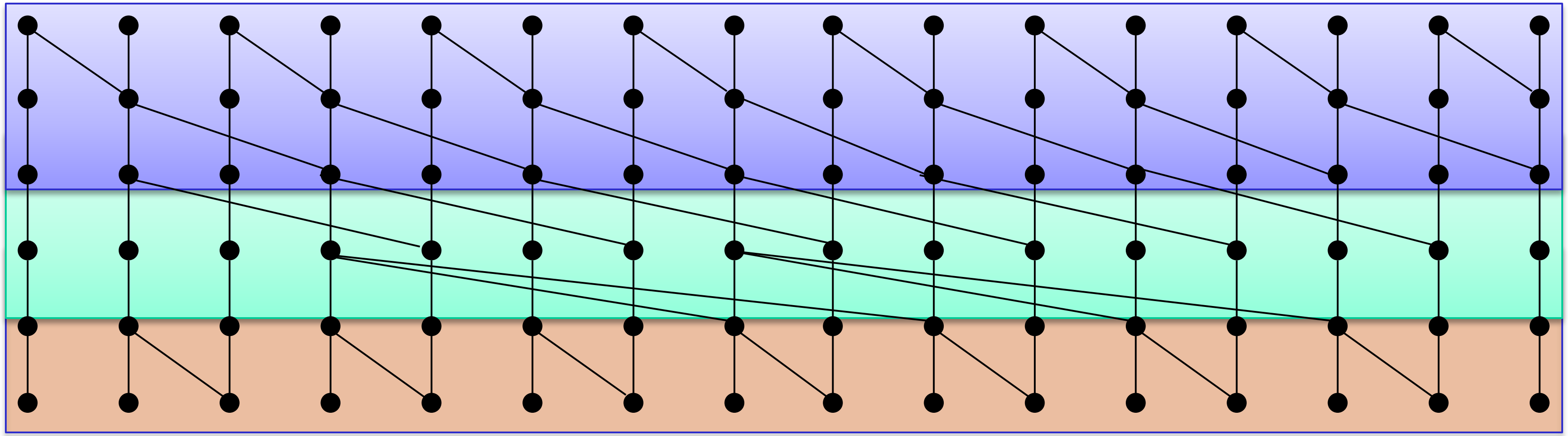
Programmer provides an example network for N=16.



[D. Harris, A taxonomy of parallel prefix networks]

Synthesis: generalize example into an algo

The algorithm must work for any N.



Synthesizer greedily identifies stages in the example and necessary expressions for each stage.

Synthesized Code

Sketch for each stage:

```
for i = 0 .. h(N)
    for j = 0 to N-1 in parallel
        if (g(i, j) < N)
            a[g(i, j)] = x[f(i, j)] + x[g(i, j)]
```

$$f = -1 + 2j + 2^i$$
$$g = -1 + 2j + 2.2^i$$

$$f = 1 + 2j$$
$$g = 2 + 2j$$

$$f = -1 + 2.2^i.floor\left(\frac{j}{2^i}\right) + 2.2^i$$
$$g = -1 + (2 - 2^i).floor\left(\frac{j}{2^i}\right) + (4 + \mathrm{j}).2^i$$

HPC Scenarios

Domain expert:

problem spec --> dynamic programming --> parallel scan

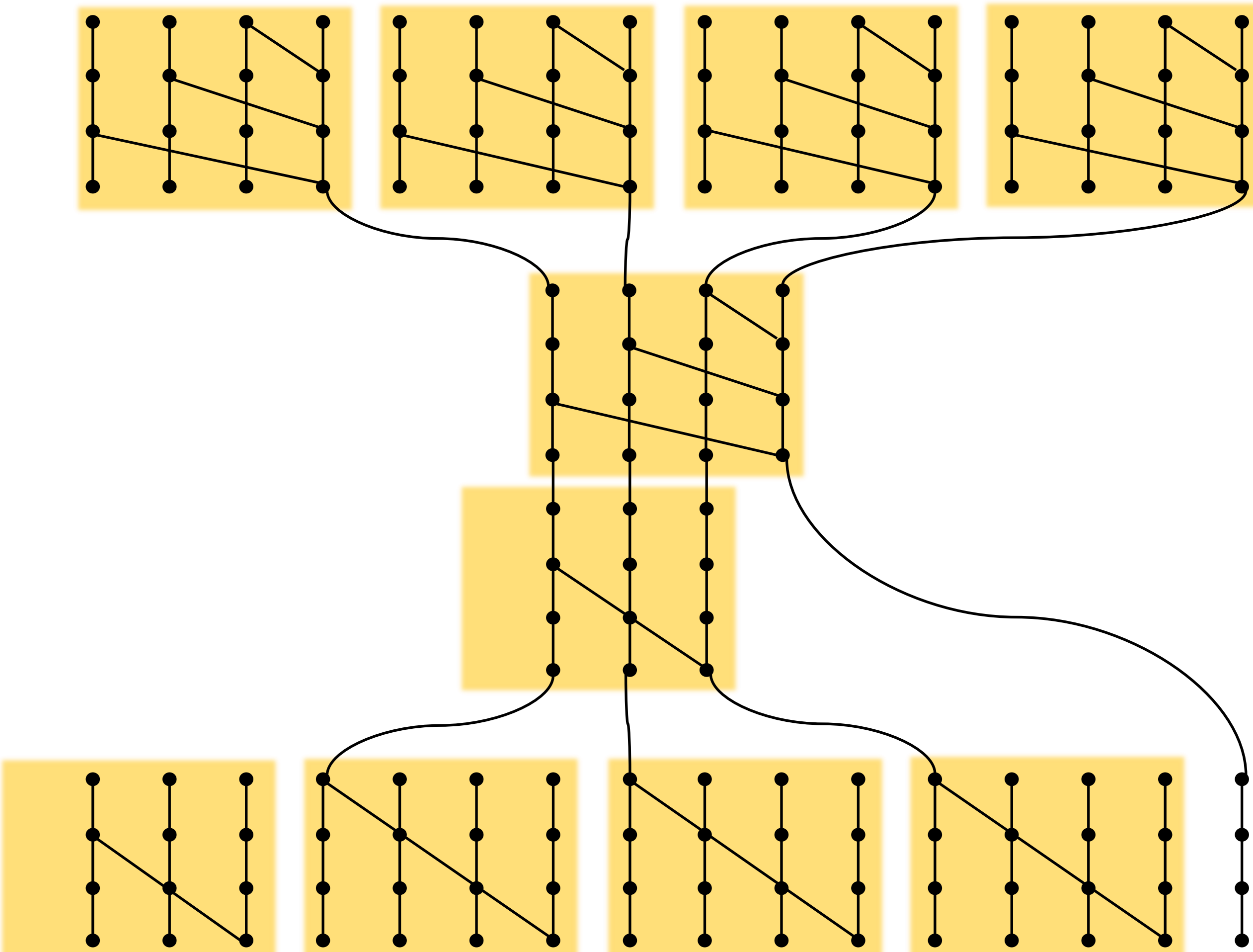
Parallel algorithm expert:

example of parallel scan network --> SIMD algorithm

GPU tuning expert:

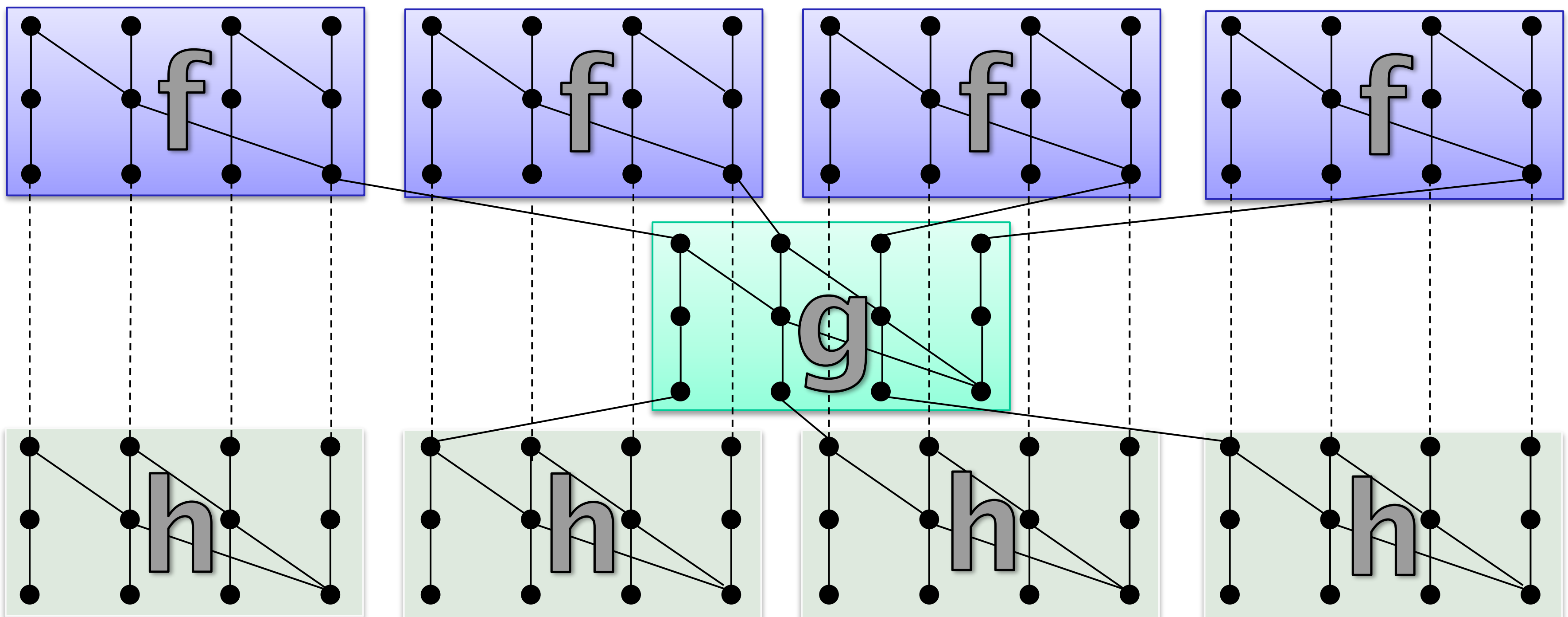
SIMD algorithm --> bank conflicts --> index expressions

Hierarchical execution of scans algorithms



Hierarchical Scan Synthesis

Holes in the sketch are functions f, g, h and number of elements transferred between 1st and 2nd stage.



HPC Scenarios

Domain expert:

problem spec --> dynamic programming --> parallel scan

Parallel algorithm expert:

example of parallel scan network --> SIMD algorithm

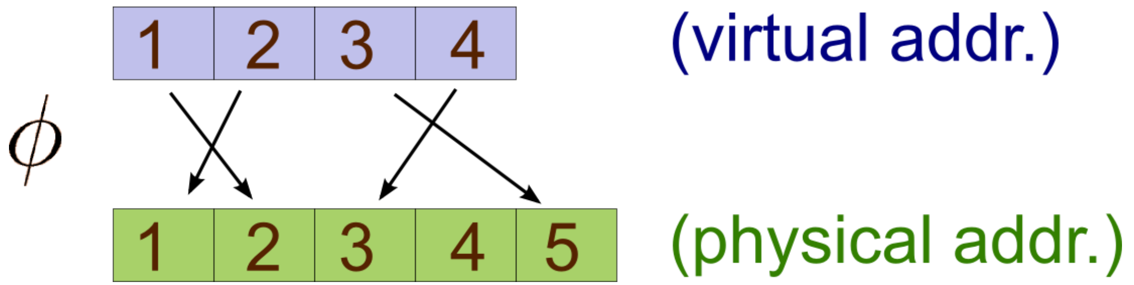
GPU tuning expert:

SIMD algorithm --> bank conflicts --> index expressions

Bank conflict avoidance

Goal: map logical array elements to a physical array.

Result: we have synthesized [injective] reordering functions as shown below. Synthesis takes approximately 2 minutes.



What does the programmer do: rewrites program to map indices to a synthesized function *phi*: `A[e]` → `A[phi(e)]`

How does the synthesizer understand bank conflicts: it simulates array accesses and synthesizes a program that minimizes bank conflicts.

HPC Scenarios

Domain expert:

problem spec --> dynamic programming --> parallel scan

Parallel algorithm expert:

example of parallel scan network --> SIMD algorithm

GPU tuning expert:

SIMD algorithm --> bank conflicts --> index expressions

Optimize index expressions

Base version:

```
for d := 1 to log2n do
    for k from 0 to n/2 in parallel do
        block := 2 * (k – (k mod 2^d))
        me := block + (k mod 2^d) + 2^d
        spine := block + 2^d – 1;
        m[me] := m[me] + m[spine];
```

[Merrell, Grimshaw, 2009]

Optimized version (a refinement)

Produced from a sketch (work in progress):

```
for (i := 1; i < 64; i := i * 2)
    rightmask := i – 1;
    leftmask := ¬rightmask;
    block := (k & leftmask) << 1;
    me := block | rightmask;
    spine := block | (k & rightmask) | i;
    m[me] := m[me] + m[spine];
```

[Merrell, Grimshaw, 2009]

Conclusion

Automatic functional equivalence checking

enabled by recent advances in program analysis, testing

Sketch-based synthesis

automatically generate details of tricky algorithms

Autotuning and algorithm design space exploration

search design spaces you could never consider by hand

Acknowledgements

Berkeley

Shaon Barman
Ras Bodik
Sagar Jain
Evan Pu
Saurabh Srivastava
Nicholas Tung

MIT

Armando Solar-Lezama
Rishabh Singh
Kuat Yesenov
Jean Yung
Zhiley Xu