Parallel Programming with Inductive Synthesis

Shaon Barman, Rastislav Bodik, Sagar Jain, Yewen Pu, Saurabh Srivastava, and Nicholas Tung
University of California, Berkeley

Abstract

We show that program synthesis can generate GPU algorithms as well as their optimized implementations. Using the scan kernel as a case study, we describe our evolving synthesis techniques. Relying on our synthesizer, we can parallelize a serial problem by transforming it into a scan operation, synthesize a SIMD scan algorithm, and optimize it to reduce memory conflicts.

1 The Problem

Parallel codes are almost exclusively hand-written. Their coding is time-consuming in part for these reasons:

- *Need for new algorithms.* To map a problem to hardware, we may need a new algorithm. For example, by reformulating a serial problem as a prefix sum, we can solve it on data-parallel architectures.
- *Low-level coordination.* High-level languages are rarely used because programmers explicitly coordinate low-level details, such as data placement.
- *Trial and error.* Parallel code is short but it is produced by experimenting with many code variants. These are often written and debugged only to discover that their optimization yielded no benefit. Furthermore, the optimization is never reused because it cannot be abstracted into a library function.

A good programming system should thus help develop new algorithms, allow productivity programming with the option to orchestrate low-level details, and facilitate reuse of code from code variants that would otherwise be thrown away.

In this paper, we describe a step in this direction. We designed an automatic program generator based on inductive synthesis. Our synthesizer's usage is similar to a code generator in that the programmer gives the *template* of the desired code. Unlike when programming with a code generator, the programmer need not write infrastructure that correctly parameterizes the template; the synthesizer discovers the parameters automatically.

2 Inductive Synthesis

Example. We start with a simple example. Assume that the programmer wants to transform the polynomial $x^2 - 2x - 3$ into its factored form $(x-3)(x+1)$. To formulate this task as a synthesis problem, she views $x^2 - 2x - 3$ as the *specification* and the factored form as an *implementation* of the specification. The implementation is a program that computes the same value as the specification. In the SKETCH synthesizer [1], her synthesis problem is written roughly as follows:

```
def main(x) =
  assert x*x-2*x-3 == factoredForm(x)
```

The function factoredForm gives the template of the implementation, i.e., it defines the factored form:

```
def factoredForm(x) = (x-??)*(x-??)
```

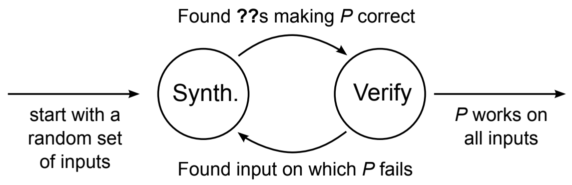
Throughout the paper, we underline names of template functions. The template's parameters are denoted with ??. The synthesizer replaces these with constants so that program passes its assertions on all inputs; this ensures that factoredForm computes the same values as the specification, x*x-2*x-3. The result of synthesis is the template instantiated with the inferred values of ??:

```
def factoredForm(x) = (x-3)*(x+1)
```

Discussion. How much programmer effort went into obtaining a system that can factor a polynomial? She only had to write the template (x-??)*(x-??) which defines how the factored form *looks like*. Contrast that with a code generator: she would have to implement an algorithm that *computes* the roots of the polynomial.

The Synthesis Algorithm. How does the synthesizer resolve each ?? into an constant so that a template instantiated with these values is a correct program, i.e., it passes all assertions?

We start with a random set of inputs, and try to *synthesize* (find) values for ??s that make the instantiated template, called P, correct on these inputs. We then *verify* whether P is correct on *all* inputs. If it is not, we have a failing input, and try to re-synthesize a new program that works on this input (and all previous ones). Otherwise, P is correct on all inputs [15]. The process is shown below.


Found **??**s making *P* correct
Synth.
Verify
start with a random set of inputs
P works on all inputs
Found input on which *P* fails

When a template cannot implement a specification, the synthesizer tells the programmer that no values of the ?? parameters yield a correct program.

Our synthesizer is inefficient for well-understood tasks such as factoring polynomials but it is more general and hence applicable to domains where deductive synthesizers [16, 9] or optimizing compilers have not been developed. We describe such applications in Section 3.

3 Programming Workflow

This section uses implementations of parallel scans on a GPU [10, 13, 14] as a case study. We propose the following synthesis workflow, and elaborate in the following subsections:

- Parallelize a serial problem by synthesizing an associative operator that reformulates the original problem as a scan, which is naively executable in parallel (§3.1).
- Synthesize a SIMD version of a parallel scan algorithm from a programmer-produced example of the parallel algorithm's execution, i.e., from an instance of the parallel scan network (§3.2).
- Optimize a SIMD scan algorithm, e.g., permute array elements to reduce bank conflicts (§3.3).
- Synthesize a SIMD algorithm for a backward scan from a forward scan. In the process, reuse optimizations developed for the forward problem (§3.4).

This workflow is not complete. Preceding these steps, we want to discover an algorithm: we are working on synthesis of high-level algorithms that may decompose the scan problem differently. Proceeding these steps, we want to autotune across multiple code variants; we need to pass along multiple solutions from each step (in an automated fashion), and generate GPU code good enough to merit autotuning.

3.1 Parallelization by Scan-Reduction

Background. *Scan* is a function that takes an array $[x_0,\ldots,x_{n-1}]$, an associative operator $\oplus$, and produces the array $[x_0, x_0 \oplus x_1, \ldots, x_0 \oplus \ldots \oplus x_{n-1}]$. A scan can be computed in parallel because $\oplus$ is associative: Given an input $[a,b,c,d]$, a serial algorithm might compute $((a \oplus b) \oplus c) \oplus d$ for the last output element of the scan, while a parallel one might compute $(a \oplus b) \oplus (c \oplus d)$, evaluating $a \oplus b$ and $c \oplus d$ simultaneously.

The Problem. Some serial $O(n)$-time problems can be reduced to scans and thus solved in parallel in $O(\log n)$ time. The key to this reduction is devising an associative operator that can be applied to the problem.

The Challenge. The steps taken by the serial algorithm may not suggest what the associative operator is. We describe below the *segmented scan* problem whose segmented nature makes the problem serial. In this problem, the associative operator cannot be trivially derived from the serial specification.

Furthermore, in order to devise an associative operator, we may need to compute suitable auxiliary values. This enriches the domain of the operator: it may operate on triples while the original one worked on single values.

Our Approach. The suitable associative operator will be found by the synthesizer. We view the serial problem as a specification and we will write a template of an implementation that must behave like the specification. The synthesizer will determine the associative operator F that is missing in the template. We also ask the synthesizer to invent the auxiliary values that enrich the input by adding extra slots into each input array element.

```
def templateOfImplementation(inputArray) =
    // enrich the input array
    def enrArray = enrich(inputArray)
    // perform a scan with operator F
    enrArray = parallelScan(enrArray, F)
    return disEnrich(enrArray)
```

This template shows that communicating to the synthesizer does not require much expertise. We are simply asking it to synthesize an operator F that parameterizes a parallel scan into a correct algorithm for the original problem. Below, we show how to write the template for segmented scans.

Example. **(Segmented Prefix Sum)** The segmented prefix sum function takes a list of lists, e.g., $[4],[5,2,3],[1,4,5,1]$ and performs the prefix sum (scan) on each list, outputting $[4],[5,7,10],[1,5,10,11]$.

When the lists are small, computing a scan individually for each list is costly. Instead, we would like to perform a single scan over all the lists at once, in parallel. We start by packing the lists into a single array. We represent the list of lists as a contiguous array $[4,5,2,3,1,4,5,1]$ and we use an array of bits to mark list boundaries, $[1,1,0,0,1,0,0,0]$. We then zip the two arrays, obtaining $[(4,1),(5,1),(2,0),...]$. Now we have a single array, which we can execute a scan on. The problem is that we do not posses a suitable associative operator. As we will see below, the problem has a serial specification.

```
def SerialSegmentedScan(tupledArray) =
  def output = [0,0,...,0]
  output[0] = tupledArray[0][0]
  for i in [1, n-1]:
    (v, bit) = tupledArray[i]
    if bit == 1: output[i] = v
    else: output[i] = output[i-1] + v
  return output
```

Note that this function does not define an associative operator. Although $+$ is associative, the `bit`-dependent case splitting breaks associativity.

We now construct a template for the associative operator `F`. In our system, the user provides hints from which the template is automatically constructed. First, the user hints that, like the specification, `F` should behave differently depending on the value of `bit`. The synthesizer translates this hint into a template for `F`, which takes in two $(value, bit)$ pairs and outputs a single such pair:

```
def F(($v_1$, $b_1$),($v_2$, $b_2$)) =
    if ($b_1$==0 and $b_2$==0) return ($f_1(v_1,v_2)$, ??)
    if ($b_1$==0 and $b_2$==1) return ($f_2(v_1,v_2)$, ??)
    if ($b_1$==1 and $b_2$==0) return ($f_3(v_1,v_2)$, ??)
    if ($b_1$==1 and $b_2$==1) return ($f_4(v_1,v_2)$, ??)
```

She also hints that f_i can use addition. The synthesizer thus generates this template for f_is:

```
def $f_i(x,y)$ =
    switch ??:
        case 0: return 0  // default cases
        case 1: return x
        case 2: return y
        case 3: return x+y // user's hint
```

The synthesizer then resolves these templates, choosing correct values for bits ??, and synthesizing functions f_i. The synthesized `F` is:

```
    def F(($v_1$, $b_1$),($v_2$, $b_2$)) =
        if ($b_1$==0 and $b_2$==0) return ($v_1+v_2$, 0)
        if ($b_1$==0 and $b_2$==1) return ($v_2$, 1)
        if ($b_1$==1 and $b_2$==0) return ($v_1+v_2$, 1)
        if ($b_1$==1 and $b_2$==1) return ($v_2$, 1)
```

Given only the serial algorithm and hints, the synthesizer was able to find the correct parallel algorithm.

Synthesizing the enriched domain. As it turned out, no enrichment was nessesary for the segmented scan problem. When needed, the programmer can give possibilities for what the enriched domain might be. The enrichment is a straightforward process. It takes in an input array, and outputs an enriched array whose elements are tuples of values:

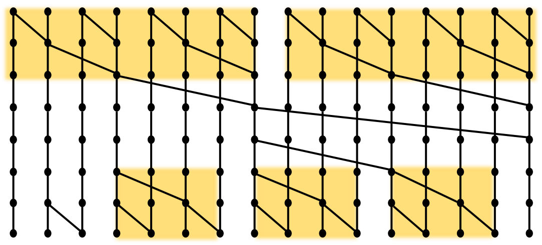
```
def enrichment(input):
    output = []
    for i in [0,n]:
        output[i] = g(input[i])
    return output
```

The function g takes in an element, and returns a tuple containing the auxiliary values. Function g is also written in some template, similar to the templates for functions f_i, and the synthesizer will find the correct enrichment function.

Discussion. Blelloch showed that a subclass of first-order recurrences can be automatically reformulated into scans [3]. However, to apply his automatic transformation, we need to first verify properties of the subclass, which is difficult and must be done manually. Our synthesis is not restricted to first-order recurrences but can currently handle any serial $O(n)$-time problems, and if the problem is indeed scan-reducible, we obtain a $O(\log n)$-time parallel scan algorithm. An example of such a $O(n)$-time scan-reducible problem is maximal string subsequence (MSS). We are currently applying this strategy to other dynamic programming algorithms.

3.2 From Examples to Algorithms

The problem. We need to encode one or more parallel scan algorithms in SIMD form. Here is an *example* of the Brent-Kung scan algorithm executed for input size $N = 16$. Each row corresponds to one SIMD step; each dot with two incoming edges correspond to one $\oplus$ operation.



We need to obtain the following SIMD code, which executes the Brent-Kung algorithm for arbitrary input size N. This is not code that anyone wishes to write by hand.

```
offset = 1
for step = 1 .. log(N)
  for i from 0 to N-1 in parallel
    if((i+1)%2*offset) a[i] = a[i]+a[i-offset]
  offset = offset * 2
offset = offset / 2
for step = 1 .. log(N)
  for i from 0 to N-1 in parallel
    if((i+1)%2*offset && (i + offset) < N)
      a[i + offset] = a[i] + a[i + offset]
  offset = offset / 2
```

The challenge. Translating the algorithmic description into iterative SIMD form is typically non-trivial because the formal algorithm is described recursively. That is, the description combines rectangular subnetworks indicated in yellow in the figure; in contrast, the SIMD formulation needs to combine SIMD steps.

Our approach. We ask the programmer for an example such as the above figure, and the synthesizer both generalizes the example from a particular value of N to any N and produces the SIMD code. We achieve this by providing the synthesizer with (i) a skeleton of the SIMD code, (ii) a functional scan specification; and (iii) assertions that insist that the completed skeleton needs to behave like the example when executed on size $N = 16$. The skeleton looks like the above SIMD code with the complicated expressions replaced with templates such as `Expr(I,offset)`. We synthesized from examples all scan algorithms in [7].

3.3 Tuning algorithms for GPUs

The Problem. The algorithms synthesized in the previous subsection are not tuned for real GPU architectures. Real GPU architectures are inefficient when the code produces bank conflicts and thread divergence [12].

The Challenge. The algorithm designer would like to explore many transformations that remove these inefficiencies, but they often involve tricky array placement. Furthermore, an optimal implementation may need to accept imperfect array placement in exchange for easy index computations. Arriving at such optimization may require exploring many code variants.

Our Approach. Inefficiencies are addressed by designing an entirely new algorithm; synthesizing a variant of an algorithm, e.g., bk_3 (base-3 Brent-Kung); applying index remapping; or placing array elements in extra storage [14, 10].

The algorithm designer provides templates for different alternative implementations, and our solver will select one that minimizes bank conflicts. We abstract the count of bank conflicts as a programmable function that takes a sequence of array access and synchronization points.

Example. In the bk_2 algorithm, bank conflicts arise at deeper levels of recursion when the algorithm simultaneously accesses array elements at 2^k offsets (as seen in the figure in Section 3.2). As an example, consider accessing elements 1 and 9. We hypothesize that remapping array indices with a function ϕ may reduce the number of bank conflicts. The scan will now access the physical locations $\phi(1)$ and $\phi(9)$, allowing them to be in different banks. ϕ maps logical indices $1 \ldots n$ to physical indices $1 \ldots m$, for $m \geq n$. We translate the original algorithm by replacing index expressions $\mathsf{array}[e]$ with $\mathsf{array}[\phi(e)]$.

The user provides a template for ϕ, which could be a table lookup permitting arbitrary permutations or a sequence of arithmetic operations. For the latter case, our synthesizer (re)invents the common padding technique:

$$\phi(i) = i + \frac{i}{\text{number of banks}}$$

where the division refers to integer division.

Discussion. In practice, this implementation does not improve performance as the expense of index computation offsets the savings from avoiding conflicts. Fortunately, our synthesizer synthesizes bk_3 because it has fewer bank conflicts than bk_2, even without index remapping. On a nVidia GTX 260 we observe a 14% speedup. We are currently applying the index remapping technique to other algorithms with more compute intensive conflicting steps, and synthesizing more efficient ϕ.

3.4 Synthesizing Behavioral Variants

The last step in our workflow transforms a parallel algorithm for one problem to algorithm for a similar problem. From Brent-Kung for forward scans we can synthesize Brent-Kung for backward scans, for both regular and segmented scans. The naive approach for backwards scans of reversing the array involves expensive copying, and therefore a dedicated backwards in-place scan is more desirable. We turn the forward scan into a template by *relaxing* the index expressions and loop bounds by making them template expressions. This relaxed template is then used to synthesize a backward scan.

4 Algorithm Discovery

In Section 3.2, we synthesized code from examples networks but did not explain where these examples came from. To generate these examples, workflow also includes interative algorithm discovery for algorithms with properties beneficial for GPUs. These techniques are not template-based so we only outline them here. This approach enumerates all scan networks of a given size with desired properties such as bank conflict avoidance. We then mine in these networks, finding modules which can

Synthesis problem	N	S	Synth. time
example $\rightarrow$ scan	16	$\sim 10^{18}$	$\sim$ 1 min
Seg. $\sum$ operator $\otimes$	8	$\sim 10^{3}$	$\sim$ 1 sec
Bank confl. **min**()	8	$\sim 10^{9}$	$\sim$ 1 min
bk_3	8	$\sim 10^{3}$	$\sim$ 1 min
fw $\rightarrow$ backward	8	$\sim 10^{11}$	$\sim$ 1 sec
bkw. $\rightarrow$ *seg.* bkw.	8	$\sim 10^{21}$	$\sim$ 1 min
fw $\rightarrow$ *seg.* fw	8	$\sim 10^{21}$	$\sim$ 1 min
MSS operator	8	$\sim 10^{30}$	$\sim$ 1 hour

Table 1: Synthesis times for various problems with input bounds N and search space sizes S.

be used as recursive subcomputations, facilitating divide-and-conquer generation of potentially new algorithms.

5 Experiments

Table 1 summarizes our synthesis results. We present a brief description of the problem, the size of the input array N on which the synthesized program was verified, the size of the candidate space S defined by the template, and the synthesis time. We want to point out that these problems all work with the two-phase work-efficient Brent-Kung algorithm, which is the most advanced scan implemented in software. The low synthesis times suggest that we can potentially synthesize more advanced implementations.

6 Related Work and Conclusion

Synthesizers of various kinds have been used to generate high-performance code. Rewriting synthesizers such as FFTW and AutoBayes rewrite the specification using their databases of problem decomposition rules [5, 11, 4]. Deductive synthesizers obtain desired code by computing it from a specification using a domain algebra [8, 9, 16]. We have shown that high-performance code can be generated also with *inductive synthesis* [2, 6, 17, 16].

We believe that our synthesizer has the potential to simplify code generation because the programmer need not develop an algorithm that completes a code template. The completion is done automatically by the synthesizer. We believe templates that guide our synthesis can be modified by synthesis non-experts who can generalize them to derive new algorithms and optimizations.

The programs that we currently synthesize run at half the speed of distributed nVidia libraries. This limitation is mainly because our optimizations have not yet been combined. We believe that combining them is a matter of engineering.

References

[1] Sketch: a synthesizer language and compiler. `http://bitbucket.org/gatoatigrado/sketch-frontend`.

[2] Angluin, D., and Smith, C. H. Inductive inference: Theory and methods. *ACM Comput. Surv. 15*, 3 (1983), 237–269.

[3] Blelloch, G. E. Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[4] Fischer, B., and Schumann, J. Autobayes: a system for generating data analysis programs from statistical models. *J. Funct. Program. 13*, 3 (2003), 483–508.

[5] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[6] Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *POPL* (2011), pp. 317–330.

[7] Harris, D. A taxonomy of parallel prefix networks. In *In The 10th Text Retrieval Conference (TREC-2001* (2002), pp. 2213–2217.

[8] Mateev, N., Pingali, K., Stodghill, P., and Kotlyar, V. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of ICS'00*, pp. 88–99.

[9] McDonald, J., and Anton, J. SPECWARE - producing software correct by construction. Tech. Rep. KES.U.01.3., 2001.

[10] Merrill, D., and Grimshaw, A. Parallel scan for stream architectures. Tech. Rep. CS2009-14, Department of Computer Science, University of Virginia, Dec. 2009.

[11] Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93*, 2 (2005), 232– 275.

[12] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 73–82.

[13] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan primitives for gpu computing. In *Graphics Hardware* (2007), M. Segal and T. Aila, Eds., Eurographics Association, pp. 97–106.

[14] Shubhabrata Sengupta, Mark Harris, M. G. Efficient parallel scan algorithms for gpus. Tech. Rep. NVR-2008-003, UC Davis, NVIDIA, Dec. 2008.

[15] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. Combinatorial sketching for finite programs. In *ASPLOS-XII* (2006), ACM, pp. 404–415.

[16] Srivastava, S., Gulwani, S., and Foster, J. S. From program verification to program synthesis. In *POPL* (2010).

[17] Vechev, M. T., Yahav, E., and Bacon, D. F. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI* (2006), pp. 341–353.