High-Level Synthesis with Linear Types

Izumi Tanaka, Ken Sakayori, Shinya Takamaeda-Yamazaki, Naoki Kobayashi
The University of Tokyo
Japan

Abstract

A specialized memory system with sufficient on-chip data reuse and off-chip memory bandwidth utilization is crucial for efficient hardware accelerators. High-level synthesis (HLS) is powerful in developing such accelerators, but it still requires careful manual optimization via directives in designing high-performance specialized memory systems. In this paper, we propose a novel HLS compiler with a dedicated linear type system for recognizing memory access patterns in an input program. The compiler automatically translates naïve programs without any directives into optimized HLS code with directives for a specialized memory system. Based on analyzed memory access patterns through the proposed linear type, the compiler inserts on-chip buffers to maximize data reuse and coalesces multiple off-chip memory accesses into long off-chip burst transfers. Experiments using the prototype compiler and a real FPGA board confirmed certain performance improvements.

1 Introduction

In high-level synthesis (HLS) development, optimizing data handling is crucial to improve hardware efficiency [4]. The program in Figure 1 is a filtering function using a naïve algorithm, whereas the program in Figure 2 achieves the same result but uses *buffering* and *stream processing*. The program in Figure 2 (i) stores a value read from the input in the buffer `buf`, and reuses it to reduce external memory access, and (ii) reads inputs from a stream instead of an array to avoid costly random memory access. When synthesizing hardware from these two functions, the function in Figure 2 performs about 10 times faster than the naïve implementation when the data size is large. This significant performance improvement can be attributed to two factors: First, buffering reduces the number of accesses to external memories by the kernel functions. Second, stream processing allows data to be prepared in advance without waiting for the computation to complete.

We propose a method for automatically translating naïve programs such as the one in Figure 1 to more efficient programs such as the one in Figure 2, which uses buffering and stream processing. This approach enables efficient hardware design through HLS without requiring a deep understanding of hardware specifics. We formalize the method as type-based two-step program translations: *buffer translation* and *stream translation*.

The buffer translation inserts buffering commands to avoid repeated access to the same memory index. Applying buffer translation to the program in Figure 1, we obtain the program in Figure 3, which accesses each memory index just once. The stream translation, on the other hand, transforms arrays into streams, where possible. Using a novel linear type system, the program in Figure 3 is translated into the stream-based version shown in Figure 2. We have implemented a prototype tool for automatic program translation and confirmed performance improvements in several programs through experiments.

```
1   filter(input, output) {
2     for (i = 0; i < N-1; i++) {
3       output[i] := (input[i] + input[i+1]) / 2; } }
```

Figure 1: A Naïve Filtering Program

```
1   filter(input, output) {
2     buf := input.read();
3     for (i = 0; i < N-1; i++) {
4       buf' := input.read();
5       output.write( (buf + buf') / 2 );
6       buf := buf'; } }
```

Figure 2: A Fast Filtering Program

```
1   filter(input, output) {
2   // input: rarray[0,N-1,1]
3     buf := input[0];
4     // input: rarray[1,N-1,1]
5     for (i = 0; i < N-1; i++) {
6       buf' := input[i+1];
7       // input: rarray[i+2,N-1,1]
8       output[i] := (buf + buf') / 2;
9       buf := buf'; } }
```

Figure 3: An Intermediate Program after Buffer Translation

Related Work. We are unaware of any prior work on a type-based approach to optimize HLS. Nigam et al. [4] apply affine types to restrict HLS to well-performing programs, eliminating inefficient programs but requiring users to write efficient ones manually. Seto et al. [6, 7] used scalar replacement and the polyhedral model to optimize C programs for HLS, improving hardware area and performance. We expect that our type-based approach offers more flexibility for program translation, making it better suited for handling non-array data structures and control structures such as recursion.

2 Buffer Translation

To achieve buffer translation, we introduce linear types $\mathbf{rarray}[S]$ and $\mathbf{warray}[S]$, which respectively describe read/write-only arrays where S is the set of array indices that can be accessed. The translation is expressed by the type-based translation relation $\Delta \mid \Gamma \vdash e \dashv \Delta' \mid \Gamma' \Longrightarrow e'$, where (i) e and e' are the source and target programs, (ii) Γ and Γ' are type environments that respectively describe the type of each variable before and after the execution of e', and (iii) Δ and Δ', called buffer environments, are of the form $b_1 : a[x_1], \ldots, b_k : a[x_k]$, which describe that the variables $b_1, \ldots, b_k$ hold the values of $a[x_1], \ldots, a[x_k]$. The key translation rules are as follows:

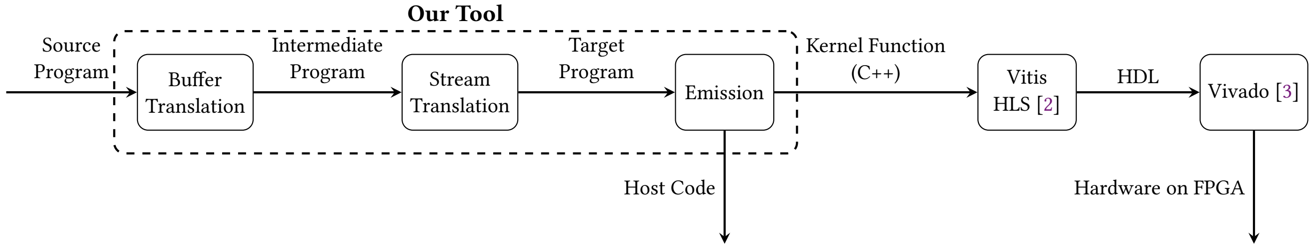

Figure 4: A High-Level Synthesis Toolchain with Our Tool

$$\frac{x \in S}{\Delta \mid \Gamma, a : \mathbf{rarray}[S] \vdash a[x] \dashv \Delta \mid \Gamma, a : \mathbf{rarray}[S \setminus \{x\}] \Longrightarrow a[x]}$$

$$\frac{}{\Delta, b : a[x] \mid \Gamma \vdash a[x] \dashv \Delta, b : a[x] \mid \Gamma \Longrightarrow b}$$

The first rule is for the case where $a[x]$ is read for the first time in the source program. In this case, $a[x]$ is read also in the target program, and x is removed from the set of indices, so that $a[x]$ will not be read again. The second rule is for the case where the value of $a[x]$ is already stored in a buffer, as indicated by the buffer environment $b : a[x]$. In this case, the read access $a[x]$ is replaced by the read from buffer b. Given $a[x]$, which rule should be applied can be determined by a type inference algorithm, whose description is omitted in this paper due to lack of space.

3 Stream Translation

For stream translation, we introduce new linear types for arrays: $\mathbf{rarray}[x, y, n]$ and $\mathbf{warray}[x, y, n]$. The type $\mathbf{rarray}[x, y, n]$ describes a read-only array that must be accessed from index x to y with a stride of n, while the type $\mathbf{warray}[x, y, n]$ describes a write-only array that has been accessed from index x to y with a stride of n. The stream translation relation is of the form $\Gamma \vdash e \dashv \Gamma' \rightsquigarrow e'$, where e and e' are the source and target programs, and Γ and Γ' describe type environments before and after the execution of e. The key translation rules are as follows:

$$\frac{}{\Gamma, a : \mathbf{rarray}[x, y, n] \vdash a[x] \dashv \Gamma, a : \mathbf{rarray}[x + n, y, n] \rightsquigarrow a_{\mathbf{strm}}.read()}$$

$$\frac{}{\Gamma, a : \mathbf{warray}[x, y - n, n] \vdash a[y] := z \dashv \Gamma, a : \mathbf{warray}[x, y, n] \rightsquigarrow a_{\mathbf{strm}}.write(z)}$$

The first rule replaces an array access $a[x]$ with a stream read $a_{\mathbf{strm}}.read()$. The type $\mathbf{rarray}[x, y, n]$ of a ensures that x is the index that should be read first (so that in the target program, $a[x]$ is available at the stream head). The type of a in the type environment is updated to $\mathbf{rarray}[x + n, y, n]$, which ensures that the subsequent read access to the array a occurs at the index $x + n$. In Figure 3, the type of `input` is updated according to this rule before and after the third and sixth lines. The second rule replaces an array assignment $a[y] := z$ with a stream write operation $a_{\mathbf{strm}}.write(z)$, provided that a has type $\mathbf{warray}[x, y - n, n]$. The type of a in the type environment is updated to $\mathbf{warray}[x, y, n]$, which expresses that $a[x], a[x + n], \ldots, a[y]$ have now been written in the source program, and the values of $a[x], a[x + n], \ldots, a[y]$ are stored in the stream a in this order in the target program.

Table 1: Execution time of the kernel functions before and after translation, along with the performance ratio. The *Buf* column reports the execution time after applying only buffer translation. The performance of Merge was measured using hand-optimized programs. (Automatic translation of such programs is left for future work.)

Name	*Src*[ms]	*Buf*[ms]	*Str*[ms]	*Src*/*Buf*	*Src*/*Str*
Filter	384	35.6	30.5	10.8	12.6
Filter-Dilated	422	35.6	30.5	11.9	13.8
Filter-2D	1150	36.6	31.1	31.4	37.0
Simple	35.6	35.6	30.8	1.00	1.16
Simple-Skip	71.8	71.8	30.6	1.00	2.35
MatVec-Mul	29.2	29.2	31.1	1.00	0.94
Merge	2660	2440	61.4	1.09	43.3

4 Experiments

We have implemented a prototype tool, whose overall architecture is shown in Figure 4. We used AMD Kria KV260 Vision AI Starter Kit [1] as the target FPGA platform. The host code, executed in Jupyter Lab [5], corresponds to the non-kernel portion of the translated program and was used to evaluate the accelerator's performance. Table 1 shows the execution times of the kernel functions of the benchmark programs we prepared. The program **Filter** is similar to the program shown in Figure 1, while **Filter-Dilated** and **Filter-2D** are its variants. **Simple** doubles the value of each element of the input array and writes the result to the output array, while **Simple-Skip** processes only the even-indexed elements. **Merge** combines two sorted arrays into one. The translation time was less than a second for all programs.

Programs with multiple memory accesses and simple access patterns, such as **Filter**, experienced substantial speedup through buffer translation alone. Stream translation, however, proves particularly effective for programs with more complex memory access patterns, such as **Merge**, where Vitis HLS's "burst access" inference does not apply.

References

[1] AMD. 2025. Kria K26 and KV260 Vision Starter Kit. https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html Accessed: 2025-01-06.
[2] AMD. 2025. Vitis Unified Software Platform. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html Accessed: 2025-01-06.

[3] AMD. 2025. Vivado Design Suite - Overview. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html Accessed: 2025-01-06.

[4] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 393–407.

[5] Project Jupyter. 2020. JupyterLab: An Interactive Development Environment. https://jupyter.org Accessed: 2025-01-15.

[6] Kenshu Seto. 2018. Scalar replacement with polyhedral model. *IPSJ Transactions on System and LSI Design Methodology* 11 (2018), 46–56.

[7] Kenshu Seto. 2019. Scalar Replacement with Circular Buffers. *IPSJ Transactions on System and LSI Design Methodology* 12 (2019), 13–21.