Transformations of High-Level Synthesis Codes for High-Performance Computing

Johannes de Fine Licht
definelicht@inf.ethz.ch

Maciej Besta
maciej.besta@inf.ethz.ch

Simon Meierhans
mesimon@ethz.ch

Torsten Hoefler
htor@inf.ethz.ch

ETH Zurich, Department of Computer Science

Abstract—Specialized hardware architectures promise a major step in performance and energy efficiency over the traditional load/store devices currently employed in large scale computing systems. The adoption of high-level synthesis (HLS) from languages such as C/C++ and OpenCL has greatly increased programmer productivity when designing for such platforms. While this has enabled a wider audience to target specialized hardware, the optimization principles known from traditional software design are no longer sufficient to implement high-performance codes, due to fundamentally distinct aspects of hardware design, such as programming for deep pipelines, distributed memory resources, and scalable routing. Fast and efficient codes for reconfigurable platforms are thus still challenging to design. To alleviate this, we present a set of optimizing transformations for HLS, targeting scalable and efficient architectures for high-performance computing (HPC) applications. Our work provides a toolbox for developers, where we systematically identify *classes* of transformations, the *characteristics* of their effect on the HLS code and the resulting hardware (e.g., increases data reuse or resource consumption), and the *objectives* that each transformation can target (e.g., resolve interface contention, or increase parallelism). We show how these can be used to efficiently exploit pipelining, on-chip distributed fast memory, and on-chip streaming dataflow, allowing for massively parallel architectures. To quantify the effect of our transformations, we use them to optimize a set of throughput-oriented FPGA kernels, demonstrating that our enhancements are sufficient to scale up parallelism within the hardware constraints. With the transformations covered, we hope to establish a common framework for performance engineers, compiler developers, and hardware developers, to tap into the performance potential offered by specialized hardware architectures using HLS.

1 Introduction

Since the end of Dennard scaling, when the power consumption of digital circuits stopped scaling with their size, compute devices have become increasingly limited by their power consumption [1]. In fact, shrinking the feature size even *increases* the loss in the metal layers of modern microchips. Today's load/store architectures suffer mostly from the cost of data movement and addressing general purpose registers and cache [2]. Other approaches, such as dataflow architectures, have not been widely successful, due to the varying granularity of applications [3]. However, *application-specific* dataflow can be used to lay out fast memory, such as registers and on-chip RAM, to fit the *specific structure* of the computation, and thereby minimize data movement. Reconfigurable architectures, such as FPGAs, can be used to implement application-specific dataflow [4], [5], [6], but are hard to program [7], as traditional hardware design languages, such as VHDL or Verilog, do not benefit from the rich set of software engineering techniques that improve programmer productivity and code reliability. For these reasons, both hardware and software communities are embracing high-level synthesis (HLS) tools [8], [9], enabling hardware development using procedural languages. HLS bridges the gap between hardware and software development, and enables basic performance portability implemented in the compilation system. For example, HLS programmers do not have to worry how exactly a floating point operation, a bus protocol, or a DRAM controller is implemented on the target hardware. Numerous HLS systems [10], [11] synthesize hardware designs from C/C++ [12], [13], [14], [15], [16], [17], OpenCL [18], [19], [20] and other high-level languages [21], [22], [23], [24], [25]. HLS provides a viable path for software and hardware communities to meet and address each other's concerns.

For many applications, compute performance is a primary goal, which is achieved through careful tuning by specialized performance engineers, who use well-understood optimizing transformations when targeting CPU and GPU architecture [26], [27]. **For HLS, a comparable collection of guidelines and principles for code optimization is yet to be established.** Optimizing codes for hardware is drastically different from optimizing codes for software. In fact, the optimization space is *larger*, as it contains most known software optimizations, in addition to HLS-specific transformations that let programmers manipulate the underlying hardware architecture. To make matters worse, the low clock frequency, lack of cache, and fine-grained configurability, means that *naive* HLS codes typically perform poorly compared to naive software codes, and must go through considerable optimization until the advantages of specialized hardware are sufficiently exploited. **Thus, the established set of traditional transformations is insufficient as it does not consider aspects of optimized hardware design, such as pipelining and decentralized fast memory**.

In this work, we define a set of key transformations that optimizing compilers or performance engineers can apply to improve the performance of hardware layouts generated from HLS codes. These transformations aim at laying out computational and fast memory resources into application-specific dataflow architectures, that efficiently uses the available spatially distributed resources on the target device. In addition to HLS-specific transformations, we discuss how code transformations known from traditional software optimization apply to HLS. We characterize and categorize transformations, allowing performance engineers to easily identify relevant ones for improving an HLS code,

	Transformations		Characteristics								Objectives							
			PL	RE	PR	ME	RS	RT	SC	CC	LD	IC	RE	CU	BW	PL	RT	RS
Pipelining	Accumulation interleaving	§2.1	👍	–	–	∼	👎	–	👎	∼	👍	–	–	–	–	–	–	–
	Delay buffering	§2.2	👍	👍	(👍)	👍	👎	(👎)	–	👎	–	👍	👍	–	–	–	–	–
	Random access buffering	§2.3	👍	👍	(👍)	👍	👎	👎	👎	👎	–	👍	👍	–	–	–	–	–
	Pipelined loop flattening	§2.5	👍	–	–	👍	–	∼	–	👎	–	–	–	–	–	👍	–	–
	Pipelined loop fusion	§2.4	👍	(👍)	–	∼	–	(👎)	–	👎	–	–	–	–	–	👍	–	–
	Inlining	§2.6	👍	–	(👍)	–	(👎)	–	–	👍	–	👍	–	–	–	–	–	–
	Condition flattening	§2.7	(👍)	–	–	–	∼	👍	–	👎	–	–	–	–	–	–	👍	–
Scaling	Vectorization	§3.1	–	–	👍	👍	👎	👎	👎	(👎)	–	–	–	👍	–	–	–	–
	Replication	§3.2	–	👍!	👍!	–	👎!	👎!	👎	👎	–	–	👍	👍	–	–	–	–
	Streaming dataflow	§3.3	–	–	(👍)	–	(👎)	👍!	–	👍	–	–	👍	–	–	–	👍	–
	Tiling	§3.4	–	👍	–	∼	👎	∼	👎	👎	–	–	–	–	–	–	👍	👍
Memory	Mem. access extraction	§4.1	(👍)	–	–	👍	👎	👍	–	👎	–	👍	–	–	👍	–	–	–
	Mem. oversubscription	§4.2	–	–	–	👍	👎	–	–	👎	–	–	–	–	👍	–	–	–
	Mem. striping	§4.3	–	–	–	👍	👎	👎	–	👎	–	–	–	–	👍	–	–	–
	Type demotion	§4.4	–	–	–	👍	👍	👍	–	👍	–	–	–	–	👍	–	–	👍

TABLE 1: Overview of **transformations**, the **characteristics** of their effect on the HLS code and the resulting hardware, and the **objectives** that they can target. The center group of column marks the following transformation characteristics: (**PL**) *enables pipelining*; (**RE**) *increases data reuse*, i.e., increases the arithmetic intensity of the code; (**PR**) *increases or exposes more parallelism*; (**ME**) *optimizes memory accesses*; (**RS**) *does **not** significantly increase resource consumption*; (**RT**) *does **not** significantly impair routing*, i.e., does **not** potentially reduce maximum frequency or prevent the design from being routed altogether; (**SC**) *does **not** change the schedule of loop nests*, e.g., by introducing more nested loops; and (**CC**) *does **not** significantly increase code complexity*. The symbols have the following meaning: "–": no effect, "👍": positive effect, "👍!": very positive effect, "(👍)": small or situational positive effect, "👎": negative effect, "👎!": very negative effect, "(👎)": small or situational negative effect, "∼": positive or negative effect can occur, depending on the context. The right group of columns marks the following **objectives** that can be targeted by transformations: (**LD**) *resolve loop-carried dependencies*; (**IC**) *resolve interface contention*; (**RE**) *increase data reuse*; (**CU**) *increase parallelism*; (**BW**) *increase memory bandwidth available to kernel*; (**PL**) *reduce pipelining overhead*; (**RT**) *improve routing results*; (**RS**) *reduce resource utilization*.

based on the problems and bottlenecks currently present. To illustrate the benefits of our guidelines, we apply them to *naive* HLS implementations of stencils, matrix multiplication, and the N-body problem. This results in dramatic cumulative speedups of $11{,}390\times$, $18{,}410\times$, and $258\times$, respectively, showing the crucial necessity of hardware-aware transformations, which are not performed automatically by today's HLS compilers. **Our work provides a set of guidelines and a reference cheat sheet for developing high-performance codes for reconfigurable architectures, enabling performance engineers, compiler developers, and hardware developers to efficiently exploit these devices.**

1.1 Key Transformations for High-Level Synthesis

We propose a set of optimizing transformations that are essential to designing scalable and efficient hardware kernels in HLS, with an overview given in Tab. 1. We divide the transformations into three major classes: **pipelining** transformations, that enable or improve the potential for pipelining computations; **scaling** transformations that increase or expose additional parallelism; and **memory** enhancing transformations, which improve memory performance. Each transformation is classified according to a number of characteristic effects on the HLS source code, and on the resulting hardware architecture. To serve as a cheat sheet, the table furthermore lists common *objectives* targeted by HLS programmers, and maps them to relevant HLS transformations. The most relevant effects and objectives are discussed in detail in relevant transformation section.

Throughout this work, we will show how each transformation is applied manually by a performance engineer by directly modifying the source code, giving examples before and after a transformation is applied. However, many transformations are also amenable to automation in an optimizing compiler.

1.2 Basics of Pipelining

Pipelining is the essence of efficient hardware architectures, as expensive instruction decoding and data movement between memory, caches and registers can be avoided, by sending data directly from one computational unit to the next. We quantify pipeline performance using two primary characteristics:

- **Latency** (L): the number of cycles it takes for an input to propagate through the pipeline and arrive at the exit, i.e., the number of **pipeline stages**. For a directed acyclic graph of dependencies between computations, this corresponds to the *critical path* of the graph.
- **Initiation interval** or **gap** (I): the number of cycles that must pass before a new input can be accepted to the pipeline. A perfect pipeline has $I{=}1\,\text{cycle}$, as this is required to keep all pipeline stages busy. Consequently, the initiation interval can often be considered the *inverse throughput* of the pipeline; e.g., $I{=}2\,\text{cycles}$ implies that the pipeline stalls every second cycle, reducing the throughput of *all* pipelines stages by a factor of $\frac{1}{2}$.

To quantify the importance of pipelining, we consider the number of cycles C it takes to execute a pipeline with latency L (both in $[\text{cycles}]$), taking N inputs, with an initiation interval of I $[\text{cycles}]$. Assuming a reliable producer and consumer at either end, we have:

$$C = L + I \cdot (N-1)\ [\text{cycles}]. \tag{1}$$

This is shown in Fig. 1. The time to execute all N iterations with clock rate f $[\text{cycles/s}]$ of this pipeline is thus C/f.

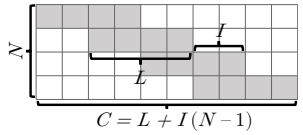


Fig. 1: Pipeline characteristics.

For two pipelines in sequence that both consume and produce N elements, the latency is additive, while the initiation interval is decided by the "slowest" actor:

$$C_0 + C_1 = (L_0 + L_1) + \max(I_0, I_1) \cdot (N-1)$$

When $I_0{=}I_1$ this corresponds to a single, deeper pipeline. For large N, the latencies are negligible, so a deeper pipeline increases pipeline parallelism by adding more computations, *without increasing the runtime*. We are thus interesting building deep, perfect pipelines to maximize performance.

1.3 Optimization Goal

We organize the remainder of this paper according to three overarching optimization goals, corresponding to the three categories marked in Tab. 1:

- **Enable pipelining** (Sec. 2): For compute bound codes, achieve $I{=}1\,\text{cycle}$ for all essential compute components, to ensure that all pipelines run at maximum throughput. For memory bound codes, guarantee that memory is always consumed at line rate.
- **Scaling/folding** (Sec. 3): Fold the total number of iterations N by scaling up the parallelism of the design to consume more elements per cycle, thus cutting the total number of cycles required to execute the program.

- **Memory efficiency** (Sec. 4): Saturate pipelines with data from memory to avoid stalls in compute logic. For memory bound codes, maximize bandwidth utilization.

Sec. 5 covers the relationship between well-known software optimizations and HLS, and accounts for which of these apply directly to HLS code. Finally, Sec. 6 shows the effect of transformations on a selection of kernels, Sec. 7 presents related work, and we conclude in Sec. 8.

2 Pipeline-Enabling Transformations

As a crucial first step for any HLS code, we cover detecting and resolving issues that prevent pipelining of computations. When analyzing a basic block of a program, the HLS tool determines the dependencies between computations, and pipelines operations accordingly to achieve the target initiation interval. There are two classes of problems that hinder pipelining of a given loop:

1) **Loop-carried dependencies** (inter-iteration): an iteration of a pipelined loop depends on a result produced by a previous iteration, which takes multiple cycles to complete (i.e., has multiple internal pipeline stages). If the latency of the operations producing this result is L, the minimum initiation interval of the pipeline will be L. This is a common scenario when accumulating into a single register (see Fig. 2), in cases where the accumulation operation takes $L_{\text{acc}}{>}1$ cycles.
2) **Interface contention** (intra-iteration): a hardware resource with limited ports is accessed multiple times in the same iteration of the loop. This could be a FIFO queue or RAM that only allows a single read and write per cycle, or an interface to external memory, which only supports sending/serving one request per cycle.

For each of the following transformations, we will give examples of programs exhibiting properties that prevent them from being pipelined, and how the transformation can resolve this. All examples use C++ syntax, which allows classes (e.g., "`FIFO`" buffer objects) and templating. We perform pipelining and unrolling using pragma directives, where loop-oriented pragmas always refer to the *following loop/scope*, which is the convention used by Intel/Altera HLS tools (as opposed to applying to *current* scope, which is the convention for Xilinx HLS tools).

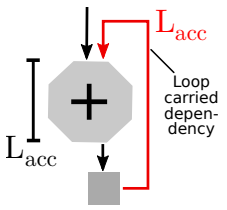


Fig. 2: Loop-carried dependency.

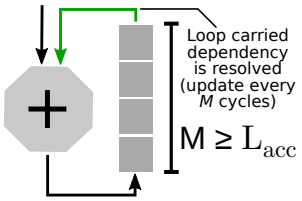


Fig. 3: Buffered accumulation.

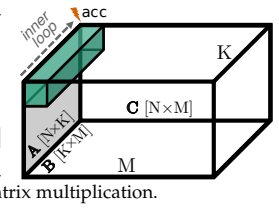
2.1 Accumulation Interleaving

For multi-dimensional iteration spaces, *loop-carried dependencies* are often be resolved by reordering and/or interleaving nested loops, keeping state for multiple concurrent accumulations. We distinguish between four approaches to interleaving accumulation, covered below.

2.1.1 Full Transposition

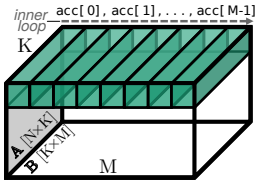
When a loop-carried dependency is encountered in a loop nest, it can be beneficial to reorder the loops, thereby fully transposing the iteration space. This typically also has a significant impact on the program's memory access pattern, which can benefit/impair the program beyond resolving a loop-carried dependency.

```
1 for (int n = 0; n < N; ++n)
2   for (int m = 0; m < M; ++m) {
3     double acc = C[n][m];
4     #pragma PIPELINE
5     for (int k = 0; k < K; ++k)
6       acc += A[n][k] * B[k][m];
7     C[n][m] = acc; }
```



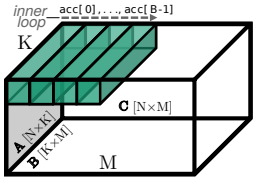
(a) Naive implementation of matrix multiplication.

```
1 for (int n = 0; n < N; ++n) {
2   double acc[M]; // Uninitialized
3   for (int k = 0; k < K; ++k)
4     double a = A[n][k]; // Only read once
5     #pragma PIPELINE
6     for (int m = 0; m < M; ++m) {
7       double prev = (k == 0) ? C[n][m]
8                              : acc[m];
9       acc[m] = prev + a * B[k][m]; }
10  for (int m = 0; m < M; ++m) // Write
11    C[n][m] = acc[m]; }       // out
```



(b) Transposed iteration space, same location written every M cycles.

```
1 for (int n = 0; n < N; ++n)
2   for (int m = 0; m < M/T; ++m) {
3     double acc[T];
4     for (int k = 0; k < K; ++k)
5       double a = A[n][k]; // M/T reads
6       #pragma PIPELINE
7       for (int t = 0; t < T; ++t) {
8         double prev = (k == 0) ?
9             C[n][m*T+t] : acc[t];
10        acc[t] = prev + a * B[k][m*T+t]; }
11    for (int t = 0; t < T; ++t) // Write
12      C[n][m*T+t] = acc[t]; }   // out
```



(c) Tiled iteration space, same location written every T cycles.

Listing 1: Transpose matrix multiplication to remove loop-carried dependency.

Consider the matrix multiplication code in Lst. 1a, computing $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C}$, with matrix dimensions N, K, and M. The inner loop $k \in K$ accumulates into a temporary register, which is written back to $\boldsymbol{C}$ at the end of each iteration $m \in M$. The multiplication of elements of $\boldsymbol{A}$ and $\boldsymbol{B}$ can be pipelined, but the addition on line 6 requires the result of the addition in the previous iteration of the loop. This is a loop-carried dependency, and results in an initiation interval of L_+, where L_+ is the latency of a 64 bit floating point addition (for integers $L_{+,\texttt{int}}{=}1$ cycle, and the loop can be pipelined without further modifications). To avoid this, we can transpose the iteration space, swapping the K-loop with the M-loop, with the following consequences:

- Rather than a single register, we now require an accumulation buffer of depth M and width 1 (line 2).
- The loop-carried dependency is resolved: each location is only updated every M cycles (with $M{\geq}L_+$ in Fig. 3).
- $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are all read in a contiguous fashion, achieving perfect spatial locality (we assume row-major memory layout. For column-major we would interchange the K-loop and N-loop).
- Each element of $\boldsymbol{A}$ is read exactly once.

The modified code is shown in Lst. 1b. We leave the accumulation buffer defined on line 2 uninitialized, and implicitly reset it on line 8, avoiding M extra cycles to reset (this is a form of *pipelined loop fusion*, covered in Sec. 2.4).

2.1.2 Tiled Accumulation Interleaving

For accumulations done in a nested loop, it can be sufficient to interleave across a tile of an outer loop to resolve a

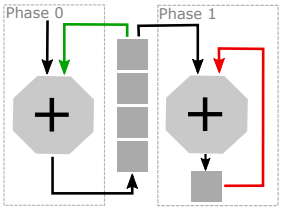
loop-carried dependency, using a limited size buffer to store intermediate results.

This is shown in Lst. 1c, for the transposed matrix multiplication example from Lst. 1b, where the accumulation array has been reduced to tiles of size T (which should be $\geq L_+$, see Fig. 3), by adding an additional inner loop over the tile, and cutting the outer loop by a factor of B.

2.1.3 Single-Loop Accumulation Interleaving

If no outer loop is present, we have to perform the accumulation in two separate stages, at the cost of extra resources. For the first stage, we perform a transformation similar to the nested accumulation interleaving, but strip-mine the inner (and only) loop into blocks of size $K \geq L_{\text{acc}}$, accumulating partial results into a buffer of size K. Once all incoming values have been accumulated into the partial result buffers, the second phase collapses the partial results into the final output. This is shown in Lst. 2.

```
1 double Acc(double arr[], int N) {
2   double t[16];
3   #pragma PIPELINE
4   for (int i = 0; i < N; ++i) { // 0
5     auto prev = (i < 16) ? 0 : t[i%16];
6     t[i%16] = prev + arr[i]; }
7   double res = 0;
8   for (int i = 0; i < 16; ++i)  // 1
9     res += t[i]; // Not pipelined
10  return res; }
```



Listing 2: Two stages required for single loop accumulation.

Optionally, the two stages can be implemented to run in a coarse-grained pipelined fashion, such that the first stage begins computing new partial results while the second stage is collapsing the previous results (by exploiting streaming between modules, see Sec. 3.3).

2.1.4 Cross-Input Accumulation Interleaving

For algorithms with loop-carried dependencies that cannot be solved by either method above (e.g., due to a non-commutative accumulation operator), we can still pipeline the design by interleaving multiple inputs to the algorithm. This procedure is similar to Sec. 2.1.2, but only applies to programs where it's relevant to compute the function for multiple inputs, and requires altering the interface of the program to accept multiple elements that can be interleaved.

The code in Lst. 3a shows an iterative solver code with an intrinsic loop-carried dependency on `state`, with a minimum initiation interval corresponding to the latency L_{Step} of the (inlined) function `Step`. There are no loops to interchange, and we cannot change the order of loop iterations. While there is no way to improve the latency of producing a single result, we can improve the overall throughput by a factor of L_{Step} by pipelining across $N \geq L_{\text{Step}}$ different inputs (e.g., overlap solving for different starting conditions). This effectively corresponds to injecting another loop over inputs, then performing transposition or nested accumulation interleaving with the inner loop. The result of this transformation is shown in Lst. 3b, for a variable number of interleaved inputs `N`.

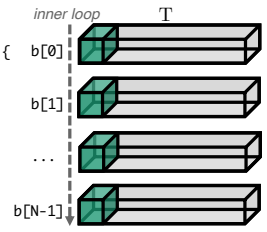
2.2 Delay Buffering

When iterating over regular domains in a pipelined fashion, it is often sufficient to express buffering using constant offset delay buffers, also known from the Intel ecosystem as

```
1 Vec<double> IterSolver(Vec<double> state, int T) {
2   #pragma PIPELINE // Will not pipeline
3   for (int t = 0; t < T; ++t)
4     state = Step(state);
5   return state; }
```

(a) Loop-carried dependency on `state`.

```
1 template <int N>
2 void MultiSolver(Vec<double> *in,
3                  Vec<double> *out, int T) {
4   Vec<double> b[N]; // Partial results
5   for (int t = 0; t < T; ++t)
6     #pragma PIPELINE
7     for (int i = 0; i < N; ++i) {
8       auto read = (t == 0) ? in[i] : b[i];
9       auto next = Step(read);
10      if (t < T-1) b[i] = next;
11      else out[i] = next; }} // Write out
```



(b) Pipeline across $N > L$ inputs to achieve $I = 1$ cycle.

Listing 3: Pipeline across multiple inputs.

shift registers. These buffers adhere to FIFO semantics, with the additional constraint that elements can only be popped once they have fully traversed the depth of the buffer (or when they pass compile-time fixed access points, called "taps", in Intel OpenCL). Despite the "shift register" name, these buffers do not need to be implemented in registers, and are frequently implemented in on-chip RAM for large depth requirements.

A common set of applications that adhere to the delay buffer pattern are stencil applications such as partial differential equation solvers [28], [29], [30], image processing pipelines [31], [32], and convolutions in deep neural networks [33], [34], all of which are typically traversed using a *sliding window* buffer, implemented in terms of multiple delay buffers (or, in Intel terminology, a shift register with multiple *taps*). These applications have been shown to be a good fit to FPGA architectures [35], [36], [37], [38], [39], [40], [41], as FIFO buffering is cheap to implement in hardware, either as shift registers in general purpose logic or RAM blocks configured as FIFOs.

Lst. 4 shows two ways of applying delay buffering to a sliding window stencil code, namely a 4-point stencil in 2D,

```
1 FIFO<float> nb(M); // North buffer
2 FIFO<float> cb(M); // Center buffer
3 float west, center;
4 // ...initialization omitted...
5 for (int i = 0; i < N; ++i) // We assume padding
6   #pragma PIPELINE
7   for (int j = 0; j < M; ++j) {
8     auto south = in[i][j+1]; // Wavefront
9     auto north = nb.Pop(); // Read line
10    auto east  = cn.Pop(); // buffers
11    out[i][j] = 0.25*(north + west + south + east);
12    nb.Push(e); cb.Push(rd);
13    west = center; center = east; } // Shift
```

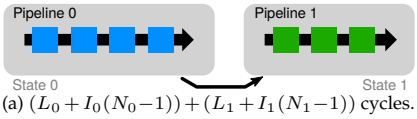
(a) Delay buffering using FIFOs.

```
1 float b[2*M]; // Shift register buffer
2 // ...initialization omitted...
3 for (int i = 0; i < N; ++i)
4   #pragma PIPELINE
5   for (int j = 0; j < M; ++j) {
6     #pragma UNROLL
7     for (int k = 0; k < 2*M-1; ++k)
8       b[k] = b[k+1]; // Shift the array
9     b[2*M-1] = in[i+1][j]; // Append wavefront
10    out[i][j] = 0.25*(b[M-1] + b[0] + b[M+1] + b[2*M-1]); }
```

(b) Delay buffering with an Intel-style shift register.

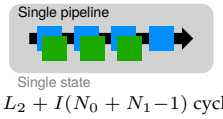
Listing 4: Two ways of implementing delay buffering.

```
1 for (int i = 0; i < N0; ++i)
2   Code0(i);
3 for (int i = 0; i < N1; ++i) Code1(i);
```



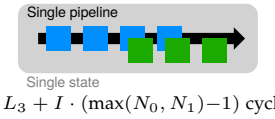
(a) $(L_0 + I_0(N_0-1)) + (L_1 + I_1(N_1-1))$ cycles.

```
1 for (int i = 0; i < N0+N1; ++i) {
2   if (i < N0) Code0(i);
3         else Code1(i - N0); }
```



(b) $L_2 + I(N_0 + N_1 - 1)$ cycles.

```
1 for (int i = 0; i < max(N0, N1); ++i) {
2   if (i < N0) Code0(i); // Can omit ifs
3   if (i < N1) Code1(i); // for N0 == N1 }
```



(c) $L_3 + I \cdot (\max(N_0, N_1) - 1)$ cycles.

Listing 5: Two subsequent pipelined loops fused sequentially (Lst. 5b) or concurrently (Lst. 5c). Assume that all loops are pipelined (pragmas omitted for brevity).

which updates each point on a 2D grid to the average of its north, west, east and south neighbors. To achieve perfect data reuse, we buffer every element read in sequential order from memory until it has been used for the last time, which is after processing two rows, when the same value has been used as all four neighbors.

In Lst. 4a we use FIFOs to implement the delay buffering pattern, instantiated on lines 1-2. We only read the south element from memory each iteration (line 8), which we store in the central delay buffer (line 12). This element is then reused after M cycles (i.e., delayed for M cycles), when it is used as the east value (line 10), shifted in registers for two cycles until it is used as the west value (line 13), after which it is pushed to the north buffer (line 12), and reused for the last time after M cycles on line 9.

Lst. 4b demonstrates the shift register pattern used to express the stencil buffering scheme, which is supported by the Intel OpenCL toolflow. Rather than creating each individual delay buffer required to propagate values, a single array is used, which is "shifted" every cycle using unrolling (line 13). The compute elements access elements of this array directly using *constant indices*, relying on the tool to infer the partitioning into individual buffers (akin to loop idiom recognition [26]) that we did explicitly in Lst. 4a. The implicit nature of this pattern requires the tool to specifically support it. For more detail on buffering stencil codes we refer to other works on the subject [42], [37]. The buffering circuit is illustrated in Fig. 4.

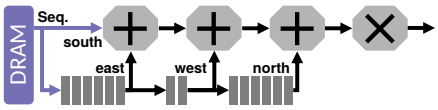


Fig. 4: A delay buffer for a 4-point stencil with three *taps*.

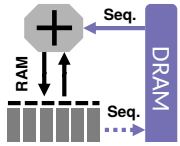
Opportunities for delay buffering often arise naturally in pipelined programs. If we consider the transposed matrix multiplication code in Lst. 1b, we notice that the read from acc on line 8 and the write on line 9 are both sequential, and cyclical with a period of P cycles. We could therefore use the shift register abstraction for this array, or replace it with an explicit FIFO buffer. The same is true for the accumulation code in Lst. 3b.

2.3 Random Access Buffering

When a program unavoidably needs to perform random accesses, we can buffer data in on-chip memory and perform fast random access there. If implemented with a general purpose replacement strategy, this emulates a CPU-style cache, but to benefit from the customizability of the FPGA architecture, it is usually more desirable to specialize the buffering strategy to the target application. Often off-chip memory accesses can be kept contiguous by loading and storing data in stages (i.e., tiles), performing random accesses exclusively to on-chip memory.

Lst. 6 outlines a histogram implementation that uses an on-chip buffer (line 1) to perform fast random accesses reads and writes (line 5) to the bins computed from incoming data, illustrated in Fig. 6. Note that the random access results in a loop-carried dependency on `histogram`, as there is a potential for subsequent iterations to read and write the same bin. This can be solved with one of the interleaving techniques described in Sec. 2.1, by maintaining multiple partial result buffers.

```
1 unsigned histogram[256] = {0};
2 #pragma PIPELINE // Will have II=2
3 for (int i = 0; i < N; ++i) {
4   int bin = CalculateBin(memory[i]);
5   histogram[bin] += 1;
6 } // ...write out result...
```



Listing 6: Random access to on-chip histogram buffer.

2.4 Pipelined Loop Fusion

When two pipelined loops appear sequentially, we can fuse them into a single pipeline, while using loop guards to enforce any dependencies that might exist between them. This transformation is closely related to loop fusion [43] from software optimization.

For two consecutive loops with latencies/bounds/initiation intervals $\{L_0, N_0, I_0\}$ and $\{L_1, N_1, I_1\}$ (Lst. 5a), respectively, the total runtime according to Eq. 1 is $(L_0 + I_0(N_0-1)) + (L_1 + I_1(N_1-1))$. Depending on which condition(s) are met, we can distinguish between three levels of pipelined loop fusion, with increasing performance benefits:

1) $I{=}I_0{=}I_1$ (true in the majority of cases): Loops are fused by summing the loop bounds, and loop guards are used to sequentialize them within the same pipeline (Lst. 5b).
2) Condition 1 is met, **and** *only fine-grained or no dependencies* exist between the two loops: Loops are fused by iterating to the maximum loop bound, and loop guards are placed as necessary to protect each section (Lst. 5c).
3) Conditions 1 and 2 are met, **and** $N{=}N_0{=}N_1$ (same loop bounds): Loops bodies are trivially fused (Lst. 5c, but with no loop guards necessary).

An alternative way of performing pipeline fusion is to instantiate each stage as a separate processing element, and stream fine-grained dependencies between them (Sec. 3.3).

2.5 Pipelined Loop Flattening/Coalescing

To minimize the number of cycles spent in filling/draining pipelines (where the circuit is not streaming at full throughput), we can flatten nested loops to move the fill/drain phases to the outermost loop, fusing/absorbing code that is not in the innermost loop if necessary.

Lst. 7a shows a code with two nested loops, and gives the total number of cycles required to execute the program.

The latency of the drain phase of the inner loop and the latency of `Code1` outside the inner loop must be *paid at every iteration* of the outer loop. If $N_0 \gg L_0$, the cycle count becomes just $L_1 + N_0 N_1 - 1$, but for applications where N_0 is comparable to L_0, even if N_1 is large, this means that the drain of the inner pipeline can significantly impact performance. By *coalescing* the two loops into a single loop (shown in Lst. 7b), the next iteration of the outer loop can be executed immediately after the previous finishes.

To perform the transformation in Lst. 7, we had to absorb `Code1` into the coalesced loop, adding a loop guard (line 4 in Lst. 7b), corresponding to pipelined loop fusion (§2.4), where the second pipelined "loop" consists of a single iteration. This contrasts the loop peeling transformation, which is used by CPU compilers to regularize loops to avoid branch mispredictions and increasing amenability to vectorization. While loop peeling can also be beneficial in hardware, e.g., to avoid deep conditional logic in a pipeline, small inner loops can see a significant performance improvement by eliminating the draining phase.

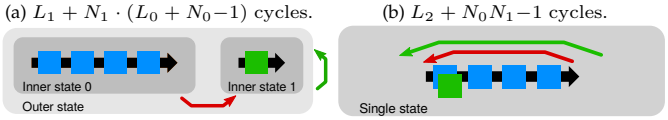
To avoid the modulo in the loop guard, we can often perform strength reduction, e.g., for values of N_0 that are a power of two, reducing the modulo to a binary AND. In the general case, we can re-introduce the individual loop variables manually. An example of this is given in Sec. 2.7.

```
1 for (int i = 0; i < N1; ++i) {
2   #pragma PIPELINE
3   for (int j = 0; j < N0; ++i)
4     Code0(i, j);
5   Code1(i); }
```

(a) $L_1 + N_1 \cdot (L_0 + N_0 - 1)$ cycles.

```
1 #pragma PIPELINE // Single loop
2 for (int ij = 0; ij < N0*N1; ++ij) {
3   Code0(ij / N0, ij % N0);
4   if (ij % N0 == 0) Code1(i / N0);
5 }
```

(b) $L_2 + N_0 N_1 - 1$ cycles.



Listing 7: Before and after coalescing loop nest to avoid inner pipeline drains.

2.6 Inlining

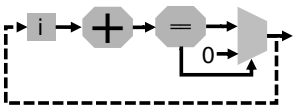
In order to successfully pipeline a code section, all function calls within must be pipelineable. The simplest way to achieve this is *inlining*, which instantiates a called function as dedicated hardware that is integrated into the pipelined circuit whenever it is called. As a preprocessing step, this transformation is no different from the software equivalent and is performed on demand by HLS compilers when possible, but results in additional resources consumed for every inlined function call. Inlining is desirable in all contexts that don't otherwise allow significant reuse of hardware resources. We implicitly assumed inlining in Lst. 5 for `Code0` and `Code1`, and for every call to `Push` and `Pop` in Lst. 4, as these codes will not pipeline otherwise.

2.7 Condition Flattening

Flattening the depth of combinatorial logic due to conditional statements can improve timing results for pipelined sections. Such statements in a pipelined section that *depend on a loop variable* must be evaluated in one cycle (i.e., they cannot be pipelined), and are thus sensitive to the latency of these operations.

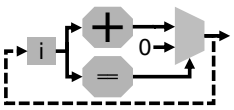
Lst. 8a shows an example of computing nested indices in a two dimensional iteration space, similar to how a loop is executed in software: the iterator of the inner loop is

```
1 int j = 0;
2 #pragma PIPELINE
3 for (int ij = 0; ij < N0*N1;
4      ++ij) {
5   Foo(j);
6   if (++j == N1)
7     j = 0; }
```



(a) Add before compare.

```
1 int j = 0;
2 #pragma PIPELINE
3 for (int ij = 0; ij < N0*N1;
4      ++ij) {
5   Foo(j);
6   if (j == N1 - 1) j = 0;
7   else ++j; }
```



(b) Parallel add and compare.

Listing 8: Flattening conditional logic can reduces the critical path of the circuit.

incremented until it exceeds the loop bounds, at which point the loop is terminated, and the iterator is incremented for the outer loop. This requires two integer additions and two comparisons to be executed before the final value of `j` is propagated to a register, where it will be read the following clock cycle to compute the next index. Because we know that `i` and `j` will always exceed their loop bounds in the final iteration, we can remove the additions from the critical path by bounds-checking the iterators before incrementing them, shown in Lst. 8b, halving the depth of logic that must be computed at each loop iteration. Note that these semantics differ from software loop at termination, as the iterator is not incremented to the out-of-bounds value before terminating.

3 Scalability Transformations

Parallelism in HLS revolves around the *folding* of loops, achieved through *unrolling*. In Sec. 2.1.1 and 2.1, we used strip-mining and reordering to avoid loop-carried dependencies by changing the *schedule* of computations in the pipelined loop nest. In this section, we similarly strip-mine and reorder loops, but with additional unrolling of the strip-mined chunks. Pipelined loops constitute the *iteration space*; the size of which determines the number of cycles it takes to execute the program. Unrolled loops, in a pipelined program, correspond to the degree of *parallelism* in the architecture, as every expression in an unrolled statement is required to exist as hardware. Parallelizing a code thus means turning sequential/pipelined loops fully or partially into parallel/unrolled loops. This corresponds to *folding* the sequential iteration space, as the number of cycles taken to execute the program are effectively reduced by the inverse of the unrolling factor.

3.1 Vectorization

We implement SIMD parallelism with HLS by partially unrolling loop nests in pipelined sections or introducing vector types, folding the sequential iteration space accordingly. This is the most straightforward way of adding parallelism, as it can often be applied directly to the inner loop without further reordering or drastic changes to the loops.

```
1 for (int i = 0; i < N / W; ++i)
2   #pragma UNROLL // Fully unroll inner
3   for (int w = 0; w < W; ++w) // loop
4     C[i*W + w] = A[i*W + w]*B[i*W + w];
```

(a) Using strip-mining.

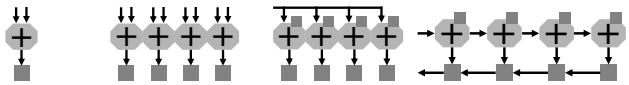
```
1 // Unroll outer loop by W
2 #pragma UNROLL W
3 for (int i = 0; i < N; ++i)
4   C[i] = A[i] * B[i];
```

(b) Using partial unrolling.

Listing 9: Two variants of vectorization using loop unrolling.

Lst. 9 shows two functionally equivalent ways of vectorizing a loop over N elements by a factor of W. Lst. 9a strip-mines a loop into chunks of the vector size and unrolls the chunk, while Lst. 9b uses partial unrolling by specifying the unroll factor in the pragma directive. As a third option, explicit vector types can used, such as the ones built into OpenCL (e.g., `float4` or `int16`), which similarly replicate registers and compute logic by the specified factor, but with less flexibility in choosing the vector type and length.

The vectorization factor W [operand/cycle] is constrained by the bandwidth B [Byte/s] available to the compute logic (e.g., from off-chip memory), according to $W_{\max} = \left\lfloor \frac{B}{fS} \right\rfloor$, where f [cycle/s] is the clock frequency of the vectorized logic, and S [Byte/operand] is the operand size in bytes. While vectorization is a straightforward way of parallelization, it is bottlenecked by available bandwidth, and is thus usually not sufficient to achieve high logic utilization on large chips, where the available memory bandwidth is low compared to the available amount of compute logic. Furthermore, because the energy cost of I/O is orders of magnitude higher than moving data on the chip, it is desirable to exploit on-chip memory and pipeline parallelism instead (this follows in Sec. 3.2 and 3.3).



(a) Single adder. (b) Vectorization. (c) Replication. (d) Streaming dataflow.

Fig. 5: Vectorization, replication, and dataflow as means to increase parallelism. Rectangles represent buffer space, such as registers or on-chip RAM.

3.2 Replication

We can achieve scalable parallelism in HLS without relying on external memory bandwidth by exploiting data reuse, distributing input elements to multiple computational units replicated through unrolling. *This is the most potent source of parallelism on hardware architectures*, as it can conceptually scale indefinitely with available silicon when enough local reuse is possible. Viewed from the paradigm of cached architectures, the opportunity for this transformation arises from temporal locality in loops. Replication draws on bandwidth from on-chip fast memory by storing more elements temporally, combining them with new data streamed in from external memory to increase parallelism, allowing more computational units to run in parallel at the expense of buffer space. This is distinct from vectorization, which requires us to widen the data path that passes through the processing elements (compare Fig. 5b and 5c).

When attempting to parallelize a new algorithm, identifying a source of temporal parallelism to feed replication is essential to whether an architecture will scale. Programmers should consider this carefully before designing the hardware architecture. From a reference software code, the programmer can identify scenarios where reuse occurs, then extract and *explicitly express* the temporal access pattern in hardware, using a constant distance [§2.2] or random-access [§2.3] buffering scheme. Then, if additional reuse is possible, replicate the circuit to scale up performance.

As an example, we return to the matrix multiplication code from Lst. 1c. In Sec. 2.1.2, we saw that strip-mining and reordering loops allowed us to move reads from matrix $\boldsymbol{A}$ out of the inner loop, re-using the loaded value across T different entries of matrix $\boldsymbol{B}$ streamed in while keeping the element of $\boldsymbol{A}$ in a register. Since every loaded value of $\boldsymbol{B}$ *eventually* needs to be combined with all N rows of $\boldsymbol{A}$, we realize that we can perform more computations in parallel by keeping *multiple* values of $\boldsymbol{A}$ in local registers. The result of this transformation is shown in Lst. 10. By buffering P elements (where P was 1 in Lst. 1c) of $\boldsymbol{A}$ prior to streaming in the tile of $\boldsymbol{B}$-matrix (lines 8-9), we can *fold* the outer loop over rows by a factor of P, using unrolling to multiply the amount of compute (as well as buffer space required for the partial sums), by a factor of P (lines 12-14).

```
1  for (int n = 0; n < N / P; ++n) { // Folded by replication factor P
2    for (int m = 0; m < M / T; ++m) {
3      double acc[T][P]; // Is now 2D
4      // ...initialize acc from C...
5      for (int k = 0; k < K; ++k) {
6        double a_buffer[P]; // Buffer multiple elements to combine with
7        #pragma PIPELINE    // incoming values of B in parallel
8        for (int p = 0; p < P; ++p)
9          a_buffer[p] = A[n*P + p][k];
10       #pragma PIPELINE
11       for (int t = 0; t < T; ++t) // Stream B
12         #pragma UNROLL // P-fold replication
13         for (int p = 0; p < P; ++p)
14           acc[t][p] += a_buffer[p] * B[k][m*T+t];
15     } /* ...write back 2D tile of C... */ } }
```

Listing 10: P-fold replication of compute units for matrix multiplication.

3.3 Streaming Dataflow

For complex codes it is common to partition functionality into multiple modules, or *processing elements* (PEs), streaming data between them through explicit interfaces. In contrast to conventional pipelining, PEs arranged in a streaming dataflow architecture are scheduled separately when synthesized by the HLS tool. There are multiple benefits to this:

- *Different functionality runs at different schedules*. For example, *issuing* memory requests, *servicing* memory requests, and *receiving* requested memory can all require different pipelines, state machines, and even clock rates.
- Smaller components are more *modular*, making them easier to reuse, debug and verify.
- The effort required by the HLS tool to schedule code sections increases dramatically with the number of operations that need to be considered for the dependency and pipelining analysis. Scheduling logic in smaller chunks is thus beneficial for compilation time.
- Large *fanout/fanin* is challenging to route on real hardware, (i.e., 1-to-N or N-to-1 connections for large N). This is mitigated by partitioning components into smaller subparts and adding more pipeline stages.
- It is cheaper to *stall* and *reset* smaller PEs, as the signal has to propagate to less logic in a single cycle.

To move data between PEs, communication channels with a handshake mechanism are used. These channels double as synchronization points, as they imply a consensus on the program state. In practice, channels are (with the exception of I/O) always FIFO interfaces, and support standard queue operations `Push`, `Pop`, and sometimes `Empty`, `Full`, and `Size` operations. They thus occupy the same shift register or memory block resources as other buffers (see Sec. 2.2).

```
1 void PE(FIFO<float> &in, FIFO<float> &out) {
2   // ..initialization...
3   for (int t = 0; t < T / P; ++t)
4     #pragma PIPELINE
5     for (/* loops over spatial dimensions */) {
6       auto south = in.Pop(); // From t-1
7       // ...load values from buffers...
8       auto next = 0.25*(north + west + east + south);
9       out.Push(next); }} // To t+1
```

(a) Processing element for a single timestep. Will be replicated P times.

```
1 #pragma PIPELINE DATAFLOW
2 void StreamStencil(const float in[], float out[]) {
3   FIFO<float> pipes[P+1];
4   ReadMemory(in, pipes[0]); // Head
5   #pragma UNROLL // Replicate PEs
6   for (int p = 0; p < P; ++p)
7     PE(pipe[p], pipe[p+1]);
8   WriteMemory(pipes[P], out); } // Tail
```

(b) Instantiate and connect P consecutive and parallel PEs.

Listing 11: Streaming between replicated PEs to compute P timesteps in parallel.

The mapping from source code to PEs differs between HLS tools, but is manifested when functions are connected using channels. In the following, we will use the syntax from Xilinx Vivado HLS to instantiate PEs, where each non-inlined function correspond to a PE, and these are connected by channels that are passed as arguments to the functions from a top-level entry function. In Intel OpenCL, the same semantics are instead expressed with multiple `kernel` functions each defining a PE, which are connected by global channel objects prefixed with the `channel` keyword.

To see how streaming can be an important tool to express scalable hardware, we apply it in conjunction with replication (Sec. 3.2) to implement an iterative version of the stencil example from Lst. 4. Unlike the matrix multiplication code, the stencil code has no scalable source of parallelism in the spatial dimension. Instead, we can achieve reuse by folding the outer time-loop to treat P consecutive timesteps in a pipeline parallel fashion, each computed by distinct PEs connected via channels [35], [44]. We replace the memory interfaces to the PE with channels, such that the memory read and write become `Pop` and `Push` operations, respectively. The resulting code is shown in Lst. 11a. We then use unrolling to make P replications of the PE (shown in Lst. 11b), effectively increasing the throughput of the kernel by a factor of P, and consequently the runtime by folding the outermost loop by a factor of P (line 3 in Lst. 11a). Such architectures are sometimes referred to as *systolic arrays* [45], [46].

For architectures/HLS tools where large fanout is an issue for compilation or routing, streaming dataflow can be applied to an already replicated design. For example, in the matrix multiplication example in Lst. 10, we can move the P-fold unroll out of the inner loop, and replicate the entire PE instead, replacing reads and writes with channel accesses. $\boldsymbol{B}$ is then streamed into the first PE, and passed downstream every cycle. $\boldsymbol{A}$ and $\boldsymbol{C}$ should no longer be accessed by every PE, but rather be handed downstream similar to $\boldsymbol{B}$, requiring a careful implementation of the start and drain phases, where the behavior of each PE will vary slightly according to its depth in the sequence.

3.4 Tiling

Loop tiling in HLS is commonly used to fold large problem sizes into manageable chunks that fit into fast on-chip memory, in an already pipelined program. Rather than making the program faster, this lets the already fast architecture support arbitrarily large problem sizes, in contrast to loop tiling on CPU and GPU, where tiling is used to increase performance. Common for both paradigms is that they ultimately aim to meet fast memory constraints. As with vectorization and replication, tiling relies on strip-mining loops to alter the iteration space.

Tiling was already shown in Sec. 2.1.2, when the accumulation buffer in Lst. 1b was reduced to a tile buffer in Lst. 1c, such that the required buffer space used for partial results became a constant, rather than being dependent on the input size. This transformation is also relevant to the stencil codes in Lst. 4, where it can be used similarly to restrict the size of the FIFOs or shift register.

4 Memory Access Transformations

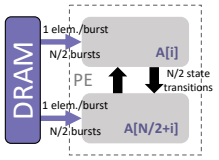
When an HLS design has been pipelined, scheduled, and unrolled as desired, the memory access pattern has been established. In the following, we describe transformations optimize the efficiency of off-chip memory accesses in the HLS code. For memory bound codes, this is crucial for performance after the design has been pipelined.

4.1 Memory Access Extraction

By extracting accesses to external memory from the computational logic, we enable compute and memory accesses to be pipelined and optimized separately. Accessing the same interface multiple times within the same pipelined section is a common cause for poor memory bandwidth utilization and increased initiation interval due to interface contention, since the interface can only service a single request per cycle. In the Intel OpenCL flow, memory extraction is done automatically by the tool, but since this process must be conservative due to limited information, it is often still beneficial to do the extraction explicitly in the code. In many cases, such as for independent reads, this is not an intrinsic memory bandwidth or latency constraint, but arises from the tool scheduling iterations according to program order. This can be relaxed when allowed by inter-iteration dependencies (which can in many cases be determined automatically, e.g., using polyhedral analysis [47]).

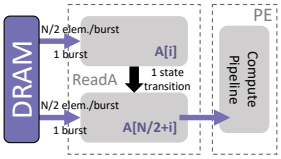
In Lst. 12a, the same memory (i.e., hardware memory interface) is accessed twice in the inner loop. In the worst case, the program will issue two $4\,\mathrm{Byte}$ memory requests every iteration, resulting in poor memory performance, and preventing pipelining of the loop. In software, this problem would be mitigated by the cache, which always fetches at least one cache line. If we instead read the two sections of A sequentially (or in chunks), the HLS tool can infer two bursts accesses to A of length $N/2$, shown in Lst. 12c. Since the schedules of memory and computational modules are independent, `ReadA` can run ahead of `PE`, ensuring that memory is always read at the maximum bandwidth of the interface (Sec. 4.2 and Sec. 4.3 will cover how to increase this bandwidth). From the point of view of the computational PE, both A_0 and A_1 are read in parallel, as shown on line 5 in Lst. 12b, hiding initialization time and inconsistent memory producers in the synchronization implied by the data streams.

```
1 void PE(const int A[N], int B[N/2]) {
2   #pragma PIPELINE // Achieves I=2
3   for (int i = 0; i < N/2; ++i)
4     // Issues N memory requests of size 1
5     B[i] = A[i] + A[N/2 + i];
6 }
```



(a) Multiple accesses to A cause inefficient memory accesses.

```
1 void PE(FIFO<int> &A0, FIFO<int> &A1,
2         int B[N/2]) {
3   #pragma PIPELINE // Achieves I=1
4   for (int i = 0; i < N/2; ++i)
5     B[i] = A0.Pop() + A1.Pop());
6 }
```



(b) Move memory accesses out of computational code.

```
1 void ReadA(const int A[N], FIFO<int> &A0, FIFO<int> &A1) {
2   int buffer[N/2];
3   #pragma PIPELINE
4   for (int i = 0; i < N/2; ++i)
5     buffer[i] = A[i]; // Issues 1 memory request of size N/2
6   #pragma PIPELINE
7   for (int i = 0; i < N/2; ++i) {
8     A0.Push(buffer[i]); // Sends to PE
9     A1.Push(A[N/2 + i]); }} // Issues 1 memory request of size N/2
```

(c) Read A in long bursts and stream them to the PE.

Listing 12: Separate memory accesses from computational logic.

An important use case of memory extraction appears in the stencil code in Lst. 11, where it is necessary to separate the memory accesses such that the PEs are agnostic of whether data is produced/consumed by a neighboring PE or by a memory module. Memory access extraction is also useful for performing data layout transformations in fast on-chip memory. For example, we can change the schedule of reads from $\boldsymbol{A}$ in Lst. 10 to a more efficient scheme by buffering values in on-chip memory, while streaming them to the kernel according to the original schedule.

4.2 Memory Oversubscription

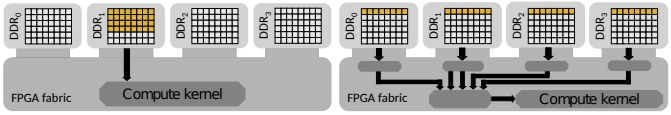
When dealing with memory interfaces with non-deterministic performance such as DRAM, it can be beneficial to request accesses earlier, and at a more aggressive pace than what is consumed or produced by the computational elements. This can be done by reading ahead into a deep buffer instantiated between memory and computations, by either 1) accessing wider vectors from memory than required by the kernel, narrowing or widening data paths (aka. "gearboxing") when piping to and from computational elements, respectively, or 2) increasing the clock rate of modules accessing memory with respect to the computational elements.

The memory access function Lst. 12c allows long bursts to the interface of A, but receives the data on a narrow bus at $W \cdot S_{\text{int}} = (1 \cdot 4)$ Byte/cycle. In general, this limits the bandwidth consumption to $f \cdot W S_{\text{int}}$ at frequency f, which is likely to be less than what the external memory can provide. To better exploit the bandwidth, we can either read wider vectors (increase W) or clock the circuit at a higher rate (increase f).

4.3 Memory Striping

When multiple memory banks with dedicated channels (e.g., multiple DRAM modules) are available, the bandwidth at which a single array is accessed can be increased by a factor corresponding the the number of available interfaces by striping it across memory banks. This optimization is employed by most CPUs transparently by striping across multi-channel memory, and is commonly known from RAID 0 configuration of disks.

We can perform striping explicitly in HLS by inserting modules that join or split data streams from two or more memory interfaces. Reading can be implemented with two or more asynchronous memory modules requesting memory from a mapped interface, then pushing to FIFO buffers that are read in parallel and combined by a third module, or vice versa for writing, exposing a single data stream to the computational kernel. This is illustrated in Fig. 6, where the unlabeled dark boxes in Fig. 6b represent PEs reading and combining data from the different DRAM modules. The Intel OpenCL compiler can in some cases perform this optimization automatically.



(a) Memory stored in a single bank. (b) Memory striped across four banks.

Fig. 6: Striping memory across memory banks increases available bandwidth.

4.4 Type Demotion

We can reduce resource and energy consumption, bandwidth requirements, and operation latency by demoting data types to less expensive alternatives that still meet precision requirements. This can lead to significant improvements on architectures that are specialized for certain types, and perform poorly on others. On traditional FPGAs there is limited native support for floating point units. Since integer/fixed point and floating point computations on these architectures compete for the same reconfigurable logic, using a data type with lower resource requirements increases the total number of arithmetic operations that can potentially be instantiated on the device. The largest benefits of type demotion are seen in the following scenarios:

- Compute bound architectures where the data type can be changed to a type that occupies less of *the same resources* (e.g., from 64 bit integers to 48 bit integers).
- Compute bound architectures where the data type can be moved to a type that is *natively* supported by the target architecture, such as single precision floating point on Intel's Arria 10 and Stratix 10 devices [48].
- Bandwidth bound architectures, where performance can be improved by up to the same factor that the size of the data type can be reduced by.
- Latency bound architectures where the data type can be reduced to a lower latency operation, e.g., from floating point to integer.

In the most extreme case, it has been shown that collapsing the data type of weights and activations in deep neural networks to binary [49], [50], [33] can provide sufficient speedup for inference that the loss of precision can me made up for with the increase in number of weights.

5 Software Transformations in HLS

In addition to the transformations described in the sections above, we include a comprehensive overview of well-known CPU-oriented transformations and how they apply to HLS, based on the compiler transformations compiled by Bacon et al. [26]. These transformations are included in Tab. 2, and are partitioned into three categories:

- Transformations directly relevant to the HLS transformations already presented here.
- Transformations that are the same or similar to their software counterparts.
- Transformations with little or no relevance to HLS.

It is interesting to note that the majority of well-known transformations from software apply to HLS. This implies that we can leverage much of decades of research into high-performance computing transformations to also optimize hardware programs, including many that can be applied *directly* (i.e., without further adaptation to HLS) to the imperative source code or intermediate representation before synthesizing for hardware. We stress the importance of support for these pre-hardware generation transformations in HLS compilers, as they lay the foundation for the hardware-specific transformations proposed here.

6 Application Examples

To show the effects of the toolbox of transformations presented in this work, we will apply them to a set of computational kernels. These kernels are written in C++ for the Xilinx Vivado HLS [66], [12] tool. We target the TUL KU115 board, which houses a Xilinx Kintex UltraScale XCKU115-2FLVB2104E FPGA and four $2400\,\mathrm{MT/s}$ DDR4 banks (we only use two banks for these experiments). The chip consists of two almost identical chiplets with limited interconnect between them, where each die is connected to two of the DDR4 pinouts. This multi-die design allows more resources ($2\times331{,}680$ LUTs and 2×2760 DSPs for the TUL KU115), but poses challenges for the routing process, which impedes the achievable clock rate and resource utilization for a monolithic kernel attempting to span the full chip. To interface with the host computer we use version 4.0 of the board firmware provided with the SDx 2017.2 [20] Development Environment, which provides the shell (e.g., DDR and PCIe controllers), and allows access to device memory and execution of the kernel through an OpenCL interface on the host side (this interface is compatible with kernels written in C++). For each example, we will describe the sequence of transformations applied, and give the resulting performance at each major stage. All results are included in Fig. 7.

6.1 Stencil Code

Stencil codes are a popular target for FPGA acceleration in HPC, due to their regular access pattern, intuitive buffering scheme, and potential for creating large systolic array designs. We implement a 4-point 2D stencil based on Lst. 4. Benchmarks are shown in Fig. 7, and use single precision floating point, and iterate over a 8192×8192 domain. We first measure a naive implementation with all explicit memory accesses, which results in no data reuse and heavy interface contention on the input array, then apply the following optimization steps:

	CPU-Oriented Transformations and how they apply to HLS codes.
Directly related to HLS transformations	☞ **Loop interchange** [51], [43] is used to resolve loop-carried dependencies [§2]. ☞ **Strip-mining** [52], **loop tiling** [53], [43], and **cycle shrinking** [54] are central components of many HLS transformations [§2.1, §3.1, §3.2, §2.1.2]. ☞ **Loop distribution** and **loop fission** [55], [43] are used to separate differently scheduled computations to allow pipelining [§3.3]. ☞ **Loop fusion** [56], [43], [57] is used for merging pipelines [§2.4]. ☞ **Loop unrolling** [58] is used to generate parallel hardware [§3.1, §3.2]. ☞ **Software pipelining** [59] is used by HLS tools to schedule code sections according to operation interdependencies to form *hardware* pipelines. ☞ **Loop coalescing/flattening/collapsing** [60] saves pipeline drains in nested loops [§2.5]. ☞ **Reduction recognition** prevents loop-carried dependencies when accumulating [§2.1]. ☞ **Loop idiom recognition** is relevant for HLS backends, for example to recognize shift buffers [§2.2] in Intel OpenCL [19]. ☞ **Procedure inlining** is required to pipeline code sections with function calls [§2.6]. ☞ **Procedure cloning** is frequently used by HLS tools when inlining [§2.6] to specialize each function "call" with values that are known at compile-time. ☞ **Loop unswitching** [61] is rarely advantageous; its *opposite* is beneficial [§2.5, §2.4]. ☞ **Loop peeling** is rarely advantageous; its **opposite** is beneficial to allow coalescing [§2.5]. ☞ **SIMD transformations** is done in HLS via unrolling [§3.1].
Same or similar in HLS	☞ **Loop-based strength reduction** [62], [63], [64], **Induction variable elimination** [65], **Unreachable code elimination** [65], **Useless-code elimination** [65], **Dead-variable elimination** [65], **Common-subexpression elimination** [65], **Constant propagation** [65], **Constant folding** [65], **Copy propagation** [65], **Forwarding substitution** [65], **Reassociation**, **Algebraic simplification**, **Strength reduction**, **Bounds reduction**, **Redundant guard elimination** are all transformations that eliminate code, which is a useful step for HLS codes to avoid generating unnecessary hardware. ☞ **Loop-invariant code motion (hoisting)** [65] does not save hardware in itself, but can save memory operations. ☞ **Loop normalization** can be useful as an intermediate transformation. ☞ **Loop reversal** [65], **array padding and contraction**, **scalar expansion**, and **scalar replacement** yield the same benefits as in software. ☞ **Loop skewing** [65] can be used in multi-dimensional wavefront codes. ☞ **Function memoization** can be applied to HLS, using explicit fast memory. ☞ **Tail recursion elimination** may be useful if eliminating dynamic recursion can enable a code to be implemented in hardware. ☞ **Regular array decomposition** applies to partitioning of both on-chip and off-chip memory. ☞ We do not consider transformations that apply only in a distributed setting (**message vectorization**, **message coalescing**, **message aggregation**, **collective communication**, **message pipelining**, **guard introduction**, **redundant communication**), but they should be implemented in dedicated message passing hardware when relevant.
Do not apply to HLS	☞ No use case found for **loop spreading** and **parameter promotion**. ☞ **Array statement scalarization**: No built-in vector notation in C/C++/OpenCL. ☞ **Code colocation**, **displacement minimization**, **leaf procedure optimization**, and **cross-call register allocation**, are not relevant for HLS, as there are no runtime function calls. ☞ **I/O format compilation**: No I/O supported directly in HLS. ☞ **Supercompiling**: is infeasible for HLS due to long synthesis times. ☞ **Short-circuiting** is not relevant to HLS, as all boolean logic exists must be instantiated and clocked in hardware regardless. ☞ **Loop pushing/embedding**: Inlining completely is favored to allow pipelining. ☞ **Automatic decomposition and alignment**, **scalar privatization**, **array privatization**, **cache alignment**, and **false sharing** are not relevant for HLS, as there is no (implicit) cache coherency protocol in hardware. ☞ **Procedure call parallelization** and **split** do not apply, as there are no forks in hardware. ☞ **Graph partitioning** only applies to explicit dataflow languages. ☞ There are no instruction sets in hardware, so **VLIW transformations** do not apply.

TABLE 2: The relation of traditional CPU-oriented transformations to HLS codes.

1) Delay buffers [§2.2] are added to store two rows of the domain (see Lst. 4a), removing interface contention on the memory bus and achieving perfect spatial data reuse.
2) We exploit spatial locality by introducing vectorization [§3.1]. To efficiently use memory bandwidth, we use memory extraction [§4.1], oversubscription [§4.2], and striping [§4.3] from two DDR banks.
3) To exploit temporal locality, we replicate the vectorized PE [§3.2] and stream [§3.3] between them (Lst. 11). The domain is tiled [§3.4] to limit fast memory usage.

Enabling pipelining with delay buffers allows the kernel to throughput $\sim$1 cell per cycle. Improving the memory performance to add vectorization (using $W = 8\,\mathrm{operands/cycle}$ for the kernel) exploits spatial locality through additional bandwidth usage. The replication and streaming step scales the design to exploit available hardware resources on the chip, until limited by placement and routing.

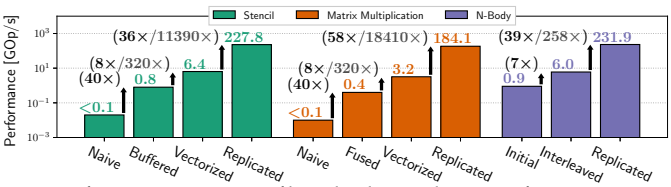


Fig. 7: Performance progression of kernels when applying transformations. Parentheses show speedup over previous version, and cumulative speedup.

6.2 Matrix Multiplication Code

We implement a scalable single precision matrix multiplication kernel using the transformations presented here. Benchmark for 8192×8192 matrices across stages of optimization are shown in Fig. 7. Starting from a naive implementation (Lst. 1a), we perform the following optimization stages:

1) We transpose the iteration space [§2.1.1], removing the loop-carried dependency on the accumulation register, and extract the memory accesses [§4.1], vastly improving spatial locality. The buffering, streaming and writing phases [§2.4] are fused, allowing us to coalesce the three nested loops [§2.5].
2) In order to increase spatial parallelism, we vectorize accesses to $\boldsymbol{B}$ and $\boldsymbol{C}$ [§3.1].
3) To scale up the design, we replicate by buffering multiple values of $\boldsymbol{A}$ and applying them streamed in values of $\boldsymbol{B}$ in parallel [§3.2]. To avoid the issue of high fanout, we partition each buffered element of $\boldsymbol{A}$ into processing elements [§3.3], arranged in a systolic array architecture. Finally, the horizontal domain is tiled to accommodate arbitrarily large matrices with finite buffer space.

Allowing pipelining and regularizing the memory access pattern brings a dramatic improvement of $40\times$, throughputting ~ 1 cell per cycle. Vectorization multiplies the performance by W, set to 8 in the benchmarked kernel. The performance of the replicated and streaming kernel is only limited by placement and routing due to high resource usage on the chip.

6.3 N-Body Code

Finally, we optimize an N-body code in 3 dimensions, using single precision floating point types and iterate over $16{,}128$ bodies. Since Vivado HLS does not allow memory accesses of a width that is not a power of two, memory extraction was included in the first stage, to support 3-vectors of velocity. We performed the following transformations:

1) We extract the memory accesses [§4.1] and read wide 512-bit vectors [§4.2], converting these into the appropriate vector sizes ($96\,\text{bit}$ for velocities, $128\,\text{bit}$ for combined position and mass).
2) The loop-carried dependency on the acceleration accumulation is solved by applying tiled accumulation interleaving [§2.1.2], pipelining across L different resident particles.
3) To scale up the performance, we further multiply the number of resident particles, this time replicating [§3.2] compute through unrolling of an outer loop into P parallel processing element arranged in a systolic array architecture. Each element holds L resident particles, and interacting particles are streamed [§3.3] through the PEs.

The second stage gains a factor of $7\times$ corresponding to the latency of the interleaved accumulation, then by a factor of $39\times$ from replicated units across the chip.

These examples demonstrate the impact of different transformations on a reconfigurable hardware platform. In particular, enabling pipelining, regularizing memory accesses, and replication are shown to be central components of scalable hardware architectures.

7 Related Work

Much work has been done in optimizing C/C++/OpenCL HLS codes for FPGA, such as stencils [36], [37], [38], [67], [68], deep neural networks [69], [70], [50], matrix multiplication [71], [68], graph processing [72], [73], and protein sequencing [74], [75]. These works optimize the respective applications using transformations described here, such as delay buffering, vectorization, replication, and streaming.

Zohouri et al. [76] use the Rodinia benchmark to evaluate the performance of OpenCL codes on FPGA, employing optimizations such as SIMD vectorization, sliding-window buffering, accumulation interleaving, and compute unit replication across multiple kernels. We present a general description of a superset of these transformations, along with concrete code examples that show how they are applied in practice. Kastner et al. [77] go through the implementation of many HLS codes in Vivado HLS, focusing on algorithmic optimizations, and apply some of the transformations found here. Lloyd et al. [78] describe optimizations specific to Intel OpenCL, and include a variant of memory access extraction, as well as the single-loop accumulation variant of accumulation interleaving.

High-level, directive-based frameworks such as OpenMP and OpenACC have been proposed as alternative abstractions for generating FPGA kernels. Leow et al. [79] implement an FPGA code generator from OpenMP pragmas, primarily focusing on correctness in implementing a range of OpenMP pragmas. Lee et al. [80] present an OpenACC to OpenCL compiler, using Intel OpenCL as a backend. The authors implement vectorization, replication, pipelining and streaming by introducing new OpenACC clauses. Papakonstantinou et al. [81] generate HLS code for FPGA from directive-annotated CUDA code.

The Data-Centric Parallel Programming (DaCe) framework optimizes applications expressed as Stateful Dataflow Multigraphs [82], exploiting information of explicit dataflow and control flow to allow a wide range of transformations to the graph-based representation. Many optimizations described in this work can applied transparently in this model, such as vectorization, inlining, and memory optimizations, while others can be implemented as explicit graph transformations. Mainstream HLS compilers automatically apply many of the well-known software transformations in Tab. 2 [23], [83], [84], but can also employ more advanced FPGA transformations. Intel OpenCL [19] performs memory access extraction into load store units (LSUs), does memory striping between DRAM banks, and detects and auto-resolves some buffering and accumulation patterns. The proprietary Merlin Compiler [85] uses high-level acceleration directives to automatically perform some of the transformations described here, as source-to-source transformations on underlying HLS code.

Polyhedral compilation is a popular framework for optimizing CPU and GPU programs [47], and has also been applied to HLS for FPGA for optimizing data reuse [86]. Such techniques may prove valuable in automating, e.g., memory extraction and tiling transformations.

Implementing programs in domain specific languages (DSLs) can make it easier to detect and exploit opportunities for advanced transformations. Darkroom [31] generates optimized HDL for image processing codes, and the popular image processing framework Halide [32] has been extended to support FPGAs [87]. Luzhou et al. [46] propose a framework for generating stencil codes for FPGAs. These frameworks rely on optimizations such as delay buffering, streaming and replication, which we cover here. Using DSLs to compile to structured HLS code can be a viable approach to automating a wide range of transformations, as proposed by Koeplinger et al. [88] and in the FROST [89] DSL framework.

8 Conclusion

Programming specialized hardware architectures has been brought to the mainstream with the adoption of high-level synthesis (HLS) tools. To facilitate the development of HPC kernels using HLS, we have proposed a set of optimizing transformations that enable efficient and scalable hardware architectures, and can be applied directly to the source code by a performance engineer, or automatically by an optimizing compiler. We hope that software and hardware programmers, performance engineers, and compiler developers, will be able to benefit from these guidelines, transformations, and the presented cheat sheet, with the goal of serving as a common toolbox for developing high performance hardware using HLS.

Acknowledgements

We thank Xilinx and Intel for helpful discussions, Xilinx for generous donations of software and hardware, and the Swiss National Supercomputing Center (CSCS) for providing computing infrastructure.

References

[1] W. A. Wulf and S. A. McKee, “Hitting the memory wall: implications of the obvious,” *SIGARCH*, 1995.
[2] M. Horowitz, “Computing’s energy problem (and what we can do about it),” in *ISSCC*, 2014.
[3] D. D. Gajski *et al.*, “A second opinion on data flow machines and languages,” *Computer*, 1982.
[4] S. Sirowy and A. Forin, “Where’s the beef? why FPGAs are so fast,” *MS Research*, 2008.
[5] A. R. Brodtkorb *et al.*, “State-of-the-art in heterogeneous computing,” *Sc. Prog.*, 2010.
[6] D. B. Thomas *et al.*, “A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation,” in *FPGA*, 2009.
[7] D. Bacon *et al.*, “FPGA programming for the masses,” *CACM*, 2013.
[8] G. Martin and G. Smith, “High-level synthesis: Past, present, and future,” *D&T*, 2009.
[9] J. Cong *et al.*, “High-level synthesis for FPGAs: From prototyping to deployment,” *TCAD*, 2011.
[10] R. Nane *et al.*, “A survey and evaluation of FPGA high-level synthesis tools,” *TCAD*, 2016.
[11] W. Meeus *et al.*, “An overview of today’s high-level synthesis tools,” *DAEM*, 2012.
[12] Z. Zhang *et al.*, “AutoPilot: A platform-based ESL synthesis system,” in *High-Level Synthesis*, 2008.
[13] Intel High-Level Synthesis (HLS) Compiler. Accessed June 27, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html
[14] A. Canis *et al.*, “LegUp: High-level synthesis for FPGA-based processor/accelerator systems,” in *FPGA*, 2011.
[15] Mentor Graphics. (2004) Catapult high-level synthesis. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls
[16] C. Pilato *et al.*, “Bambu: A modular framework for the high level synthesis of memory-intensive applications,” in *FPL*, 2013.
[17] R. Nane *et al.*, “DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler,” in *FPL*, 2012.
[18] M. Owaida *et al.*, “Synthesis of platform architectures from OpenCL programs,” in *FCCM*, 2011.
[19] T. Czajkowski *et al.*, “From OpenCL to high-performance hardware on FPGAs,” in *FPL*, 2012.
[20] Xilinx. SDAccel development environment. Accessed June 27, 2019. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
[21] R. Nikhil, “Bluespec system Verilog: efficient, correct RTL from high level specifications,” in *MEMOCODE*, 2004.
[22] J. Auerbach *et al.*, “Lime: A Java-compatible and synthesizable language for heterogeneous architectures,” in *OOPSLA*, 2010.
[23] ——, “A compiler and runtime for heterogeneous computing,” in *DAC*, 2012.
[24] J. Hammarberg and S. Nadjm-Tehrani, “Development of safety-critical reconfigurable hardware with Esterel,” *FMICS*, 2003.
[25] M. B. Gokhale *et al.*, “Stream-oriented FPGA computing in the Streams-C high level language,” in *FCCM*, 2000.
[26] D. F. Bacon *et al.*, “Compiler transformations for high-performance computing,” *CSUR*, 1994.
[27] S. Ryoo *et al.*, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” in *PPoPP*, 2008.
[28] Smith, Gordon D., *Numerical solution of partial differential equations: finite difference methods*, 1985.
[29] A. Taflove and S. C. Hagness, “Computational electrodynamics: The finite-difference time-domain method,” 1995.
[30] C. A. Fletcher, *Computational Techniques for Fluid Dynamics 2*, 1988.
[31] J. Hegarty *et al.*, “Darkroom: compiling high-level image processing code into hardware pipelines.” *TOG*, 2014.
[32] J. Ragan-Kelley *et al.*, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” in *PLDI*, 2013.
[33] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *arXiv:1802.09941*, 2018.
[34] G. Lacey *et al.*, “Deep learning on FPGAs: Past, present, and future,” *arXiv:1602.04283*, 2016.
[35] H. Fu and R. G. Clapp, “Eliminating the memory bottleneck: An FPGA-based solution for 3d reverse time migration,” in *FPGA*, 2011.
[36] H. R. Zohouri *et al.*, “Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL,” in *FPGA*, 2018.
[37] H. M. Waidyasooriya *et al.*, “OpenCL-based FPGA-platform for stencil computation and its optimization methodology,” *TPDS*, May 2017.
[38] Q. Jia and H. Zhou, “Tuning stencil codes in OpenCL for FPGAs,” in *ICCD*, 2016.
[39] X. Niu *et al.*, “Exploiting run-time reconfiguration in stencil computation,” in *FPL*, 2012.
[40] ——, “Dynamic stencil: Effective exploitation of run-time resources in reconfigurable clusters,” in *FPT*, 2013.
[41] J. Fowers *et al.*, “A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications,” in *FPGA*, 2012.
[42] J. de Fine Licht *et al.*, “Designing scalable FPGA architectures using high-level synthesis,” in *PPoPP*, 2018.
[43] D. J. Kuck *et al.*, “Dependence graphs and compiler optimizations,” in *POPL*, 1981.
[44] K. Sano *et al.*, “Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth,” *TPDS*, 2014.
[45] H. Kung and C. E. Leiserson, “Systolic arrays (for VLSI),” in *Sparse Matrix Proceedings*, 1978.
[46] W. Luzhou *et al.*, “Domain-specific language and compiler for stencil computation on fpga-based systolic computational-memory array,” in *ARC*, 2012.
[47] T. Grosser *et al.*, “Polly – performing polyhedral optimizations on a low-level intermediate representation,” *PPL*, 2012.
[48] U. Sinha, “Enabling impactful dsp designs on FPGAs with hardened floating-point implementation,” *Altera White Paper*, 2014.
[49] M. Courbariaux and Y. Bengio, “BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1,” *CoRR*, 2016.
[50] Y. Umuroglu *et al.*, “FINN: A framework for fast, scalable binarized neural network inference,” in *FPGA*, 2017.
[51] J. R. Allen and K. Kennedy, “Automatic loop interchange,” in *SIGPLAN*, 1984.
[52] M. Weiss, “Strip mining on SIMD architectures,” in *ICS*, 1991.
[53] M. D. Lam *et al.*, “The cache performance and optimizations of blocked algorithms,” 1991.
[54] C. D. Polychronopoulos, “Advanced loop optimizations for parallel computers,” in *ICS*, 1988.
[55] D. J. Kuck, “A survey of parallel machine organization and programming,” *CSUR*, Mar. 1977.
[56] A. P. Yershov, “Alpha – an automatic programming system of high efficiency,” *J. ACM*, 1966.
[57] M. J. Wolfe, “Optimizing supercompilers for supercomputers,” Ph.D. dissertation, 1982.
[58] J. J. Dongarra and A. R. Hinds, “Unrolling loops in Fortran,” *Software: Practice and Experience*, 1979.
[59] M. Lam, “Software pipelining: An effective scheduling technique for VLIW machines,” in *PLDI*, 1988.
[60] C. D. Polychronopoulos, “Loop coalescing: A compiler transformation for parallel machines,” Tech. Rep., 1987.
[61] F. E. Allen and J. Cocke, *A catalogue of optimizing transformations*, 1971.
[62] J. Cocke and K. Kennedy, “An algorithm for reduction of operator strength,” *CACM*, 1977.
[63] R. Bernstein, “Multiplication by integer constants,” *Softw. Pract. Exper.*, 1986.
[64] G. L. Steele, “Arithmetic shifting considered harmful,” *ACM SIGPLAN Notices*, 1977.
[65] A. V. Aho *et al.*, “Compilers, principles, techniques,” *Addison Wesley*, 1986.
[66] Xilinx. Vivado HLS. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[67] D. Weller *et al.*, “Energy efficient scientific computing on FPGAs using OpenCL,” in *FPGA*, 2017.
[68] A. Verma *et al.*, “Accelerating workloads on FPGAs via OpenCL: A case study with opendwarfs,” Tech. Rep., 2016.
[69] N. Suda *et al.*, “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks,” in *FPGA*, 2016.
[70] J. Zhang and J. Li, “Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network,” in *FPGA*, 2017.
[71] E. H. D’Hollander, “High-level synthesis optimization for blocked floating-point matrix multiplication,” *SIGARCH*, 2017.
[72] M. Besta *et al.*, “Graph processing on FPGAs: Taxonomy, survey, challenges,” *arXiv preprint arXiv:1903.06697*, 2019.
[73] ——, “Substream-centric maximum matchings on FPGA,” in *ACM/SIGDA FPGA*, 2019.
[74] S. O. Settle, “High-performance dynamic programming on FPGAs with OpenCL,” in *HPEC*, 2013.
[75] E. Rucci *et al.*, “Smith-Waterman protein search with OpenCL on an FPGA,” in *Trustcom/BigDataSE/ISPA*, 2015.
[76] H. R. Zohouri *et al.*, “Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs,” in *SC*, 2016.
[77] R. Kastner *et al.*, “Parallel programming for FPGAs,” 2018.
[78] T. Lloyd *et al.*, “A case for better integration of host and target compilation when using OpenCL for FPGAs,” in *FSP*, 2017.
[79] Y. Y. Leow *et al.*, “Generating hardware from OpenMP programs,” in *FPT*, 2006.
[80] S. Lee *et al.*, “OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing,” in *IPDPS*, 2016.
[81] A. Papakonstantinou *et al.*, “FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs,” in *SASP*, 2009.
[82] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, “Stateful dataflow multigraphs: A data-centric model for high-performance parallel programs,” *CoRR*, 2019.
[83] S. Gupta *et al.*, “SPARK: a high-level synthesis framework for applying parallelizing compiler transformations,” in *VLSID*, 2003.
[84] ——, “Coordinated parallelizing compiler optimizations and high-level synthesis,” *TODAES*, 2004.
[85] J. Cong *et al.*, “Source-to-source optimization for HLS,” in *FPGAs for Software Programmers*, 2016.
[86] L.-N. Pouchet *et al.*, “Polyhedral-based data reuse optimization for configurable computing,” in *FPGA*, 2013.
[87] J. Pu *et al.*, “Programming heterogeneous systems from an image processing DSL,” *TACO*, 2017.
[88] D. Koeplinger *et al.*, “Automatic generation of efficient accelerators for reconfigurable hardware,” in *ISCA*, 2016.
[89] E. D. Sozzo *et al.*, “A common backend for hardware acceleration on fpga,” in *ICCD*, 2017.