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

Department of Computer Science, ETH Zurich

Abstract—Spatial computing architectures promise a major stride 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++ and OpenCL has greatly increased programmer productivity when designing for such platforms. While this has enabled a wider audience to target spatial computing architectures, 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. To alleviate this, we present a collection of optimizing transformations for HLS, targeting scalable and efficient architectures for high-performance computing (HPC) applications. We systematically identify *classes* of transformations (pipelining, scalability, and memory), the *characteristics* of their effect on the HLS code and the resulting hardware (e.g., increasing 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 dataflow, allowing for massively parallel architectures. To quantify the effect of various transformations, we cover the optimization process of a sample set of HPC kernels, provided as open source reference codes. We aim to establish a common toolbox to guide both performance engineers and compiler engineers in tapping into the performance potential offered by spatial computing 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 as registers and on-chip memory 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 about 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] and other high-level languages [20], [21], [22], [23], [24], providing a viable path for software and hardware communities to meet and address each other's concerns.

For many applications, computational performance is a primary goal, which is achieved through careful tuning by specialized performance engineers using well-understood optimizing transformations when targeting CPU [25] and GPU [26] architectures. **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 be transformed considerably before the advantages of specialized hardware can be 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 survey and 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. This set unions transformations extracted from previous work, where they were applied either explicitly or implicitly, with additional techniques that fill in gaps to maximize completeness. We characterize and categorize transformations, allowing performance engineers to easily look up those relevant to improving their HLS code, based on the problems and bottlenecks currently present. The transformations have been verified to apply to both the Intel OpenCL and Xilinx Vivado HLS toolflows, but are expected to translate to any pragma-based imperative HLS tool.

	Transformations		Characteristics								Objectives						
			PL	RE	PA	ME	RS	RT	SC	CC	LD	RE	CU	BW	PL	RT	RS
Pipelining	Accumulation interleaving	§2.1	👍	–	–	∼	👎	–	👎	∼	👈	–	–	–	–	–	–
	Delay buffering	§2.2	👍	👍	(👍)	👍	👎	(👎)	👎	👎	👈	👈	–	–	–	–	–
	Random access buffering	§2.3	👍	👍	(👍)	👍	👎	👎	👎	👎	👈	👈	–	👈	–	–	–
	Pipelined loop fusion	§2.4	👍	(👍)	–	∼	∼	(👎)	–	👎	–	–	–	–	👈	–	–
	Pipelined loop switching	§2.5	👍	(👍)	–	∼	∼	(👎)	–	∼	–	–	–	–	👈	–	👈
	Pipelined loop flattening	§2.6	👍	–	–	👍	∼	∼	–	👎	–	–	–	–	👈	–	–
	Inlining	§2.7	👍	–	(👍)	–	(👎)	–	–	👍	👈	–	–	–	–	–	–
Scaling	Horizontal unrolling	§3.1	–	(👍)	👍	👍	👎	👎	👎	(👎)	–	–	👈	–	–	–	–
	Vertical unrolling	§3.2	–	👍!	👍!	–	👎!	👎!	👎	👎	–	👈	👈	–	–	–	–
	Dataflow	§3.3	–	–	(👍)	–	(👎)	👍!	–	👍	–	👈	–	–	👈	👈	–
	Tiling	§3.4	–	👍	–	∼	👎	∼	👎	👎	👈	👈	–	–	–	👈	👈
Memory	Mem. access extraction	§4.1	(👍)	–	–	👍	👎	👍	–	👎	👈	–	–	👈	–	–	–
	Mem. buffering	§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; (**PA**) *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 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*, due to inter-iteration dependencies or resource contention; (**RE**) *increase data reuse*; (**CU**) *increase parallelism*; (**BW**) *increase memory bandwidth utilization*; (**PL**) *reduce pipelining overhead*; (**RT**) *improve routing results*; (**RS**) *reduce resource utilization*.

In addition to identifying previous work that apply one or more of the transformations defined here, we describe and publish a set of end-to-end "hands-on" examples, optimized from naive HLS codes into high performance implementations. This includes a stencil code, matrix multiplication, and the N-body problem, all available on github. The optimized codes exhibit dramatic cumulative speedups of up to $29{,}950\times$ relative to their respective naive starting points, showing the crucial necessity of hardware-aware transformations, which are not performed automatically by today's HLS compilers. As FPGAs are currently the only platforms commonly targeted by HLS tools in the HPC domain, transformations are discussed and evaluated in this context. Evaluating FPGA performance in comparison to other platforms is out of scope of this work. **Our work provides a set of guidelines and a cheat sheet for optimizing high-performance codes for reconfigurable architectures, guiding both performance engineers and compiler developers to efficiently exploit these devices.**

1.1 From Imperative Code to Hardware

Before diving into transformations, it is useful to form an intuition of the major stages of the source-to-hardware stack, to understand how they are influenced by the HLS code:

❶ **High-level synthesis** converts a pragma-assisted procedural description (C++, OpenCL) to a functionally equivalent behavioral description (Verilog, VHDL). This requires mapping variables and operations to corresponding constructs, then scheduling operations according to their interdependencies. The dependency analysis is concerned with creating a hardware mapping such that the throughput requirements are satisfied, which for pipelined sections require the circuit to accept a new input every cycle. Coarse-grained control flow is implemented with state machines, while computations and fine-grained control flow are organized in (predicated) pipelines. ❷ **Hardware synthesis** maps the register-level circuit description to components and wires present on the specific target *architecture*. At this stage and onwards, the procedure is both vendor and architecture specific. ❸ **Place and route** maps the logical circuit description to concrete locations on the target *device*, by performing a lengthy heuristic-based optimization that attempts to minimize the length of the longest wire and the total wire length. The longest propagation time between two registers including the logic between them (i.e., the critical path of the circuit), will determine the maximum obtainable frequency. ❹ **Bitstream generation** translates the final circuit description to a binary format used to configure the device.

Most effort invested by an HLS programmer lies in guiding the scheduling process in ❶ to implement deep, efficient pipelines, but ❷ is considered when choosing data types and buffer sizes, and ❸ can ultimately bottleneck applications once the desired parallelism has been achieved, requiring the developer to adapt their code to aid this process.

1.2 Key Transformations for High-Level Synthesis

This work identifies a set of optimizing transformations that are essential to designing scalable and efficient hardware kernels in HLS. 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 increase memory utilization and efficiency. Each transformation is further classified according to a number of characteristic effects on the HLS source code, and on the resulting hardware architecture (central columns). To serve as a cheat sheet, the table furthermore lists common *objectives* targeted by HLS programmers, and maps them to relevant HLS transformations (rightmost columns). Characteristics and objectives are discussed in detail in relevant transformation sections.

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 it is applied. However, many transformations are also amenable to automation in an optimizing compiler.

1.3 The Importance of Pipelining

Pipelining is essential to 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 attribute two primary characteristics to pipelines:

- **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**.
- **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 $I{=}1\,\mathrm{cycle}$, 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., $I{=}2\,\mathrm{cycles}$ 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 in HLS, we consider the number of cycles C it takes to execute a pipeline with latency L (both in [cycles]), taking N inputs, with an initiation interval of I [cycles]. Assuming a reliable producer and consumer at either end, we have:

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

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

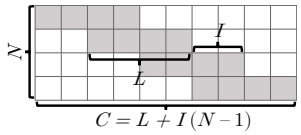


Fig. 1: Pipeline characteristics.

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 this deeper pipeline increases pipeline parallelism by adding more computations *without increasing the runtime*; and without introducing additional off-chip memory traffic. **We are thus interested in building deep, perfect pipelines to maximize performance and minimize off-chip data movement.**

1.4 Optimization Goals

We organize the remainder of this work 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. Sec. 6 shows the effect of transformations on a selection of kernels, Sec. 7 presents related work, and we conclude in Sec. 9.

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).

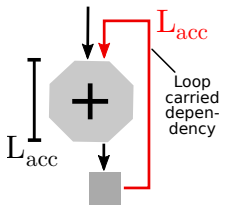


Fig. 2: Loop-carried dependency.

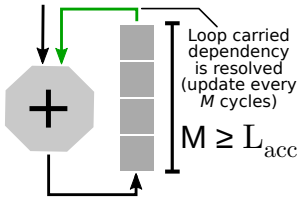


Fig. 3: Buffered accumulation.

2.1 Accumulation Interleaving

For multi-dimensional iteration spaces, *loop-carried dependencies* can 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.

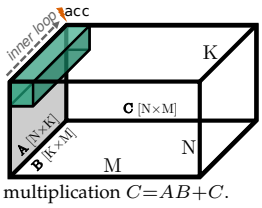
2.1.1 Full Transposition

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.

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 $64\,\text{bit}$ floating point addition (for integers $L_{+,\text{int}}{=}1\,\text{cycle}$, 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:

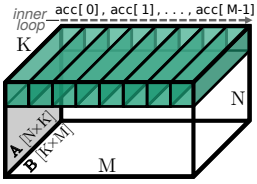
- Rather than a single register, we now implement an accumulation buffer of depth M and width 1 (line 2).

```
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; }
```



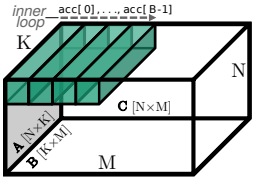
(a) Naive implementation of general matrix multiplication $C{=}AB{+}C$.

```
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]; // Tiles of size 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: Interleave accumulations to remove loop-carried dependency.

- The loop-carried dependency is resolved: each location is only updated every M cycles (with $M{\geq}L_{\text{acc}}$ 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

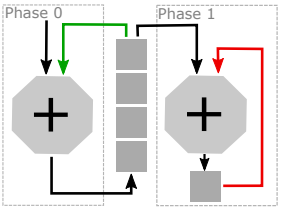
For accumulations done in a nested loop, it can be sufficient to interleave across a tile of an outer loop to resolve a loop-carried dependency, using a limited size buffer to store intermediate results. This tile only needs to be of size $\geq L_{\text{acc}}$, where L_{acc} is the latency of the accumulation operation.

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 $\geq L_{\text{acc}}$, 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

```
1 double Acc(double arr[], int N) {
2   double t[16];
3   #pragma PIPELINE
4   for (int i = 0; i < N; ++i) { // P0
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)  // P1
9     res += t[i]; // Not pipelined
10  return res; }
```



Listing 2: Two stages required for single loop accumulation.

result buffers, the second phase collapses the partial results into the final output. This is shown in Lst. 2 for $K{=}16$. 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 dataflow between modules, see Sec. 3.3).

2.1.4 Batched Accumulation Interleaving

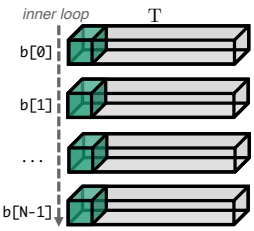
For algorithms with loop-carried dependencies that cannot be solved by either method above (e.g., due to a non-commutative accumulation operator), we can still pipeline the design by processing *batches* of inputs, introducing an additional loop nested in the accumulation loop. This procedure is similar to Sec. 2.1.2, but only applies to programs where it is relevant to compute the accumulation for multiple data streams, and requires altering the interface and data movement of the program to interleave inputs in batches.

The code in Lst. 3a shows an iterative solver code with an inherent 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). We effectively inject another loop over inputs, then perform transposition or tiled accumulation interleaving with this loop. The result of this transformation is shown in Lst. 3b, for a variable number of interleaved inputs `N`.

```
1 Vec<double> IterSolver(Vec<double> state, int T) {
2   #pragma PIPELINE // Will fail to pipeline with I=1
3   for (int t = 0; t < T; ++t)
4     state = Step(state);
5   return state; }
```

(a) Solver executed for T steps with 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 $N{\geq}L_{\text{step}}$ inputs to achieve $I{=}1$ cycle.

Listing 3: Pipeline across multiple inputs to avoid loop-carried dependency.

2.2 Delay Buffering

When iterating over regular domains in a pipelined fashion, it is often sufficient to express buffering using delay buffers, expressed either with cyclically indexed arrays, or with

constant offset delay buffers, also known from the Intel ecosystem as *shift registers*. These buffers are only accessed in a FIFO manner, with the additional constraint that elements are 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 when large capacity is needed, where values are not *physically* shifted.

A common set of applications that adhere to the delay buffer pattern are stencil applications such as partial differential equation solvers [27], [28], [29], image processing pipelines [30], [31], and convolutions in deep neural networks [32], [33], [34], [35], [36], 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 spatial computing architectures [37], [38], [39], [40], [41], [42], [43], as delay buffering is cheap to implement in hardware, either as shift registers in general purpose logic, or in RAM blocks.

Lst. 4 shows two ways of applying delay buffering to a stencil code, namely a 4-point stencil in 2D, 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 – after two rows, when the same value has been used as all four neighbors.

In Lst. 4a we use cyclically indexed line buffers to implement the delay buffering pattern, instantiated as arrays on lines 1-2. We only read the south element from memory each iteration (line 7), which we store in the center line buffer (line 13). This element is then reused after M cycles (i.e., "delayed" for M cycles), when it is used as the east value (line 9), propagated to the north buffer (line 12), shifted in registers for two cycles until it is used as the west value (line 14), and reused for the last time after M cycles on line 8. The resulting circuit is illustrated in Fig. 4.

```
1 float north_buffer[M];  // Line
2 float center_buffer[M]; // buffers
3 float west, center; // Registers
4 for (int i = 0; i < N; ++i) {
5   #pragma PIPELINE
6   for (int j = 0; j < M; ++j) {
7     auto south = memory[i][j]; // Single memory read
8     auto north = north_buffer[j];         // Read line buffers
9     auto east = center_buffer[(j + 1)%M]; // (with wrap around)
10    if (i > 1 && j > 0 && j < M - 1) // Assume padding of 1
11      result[i - 1][j] = 0.25*(north + west + south + east);
12    north_buffer[j] = center; // Update both
13    center_buffer[j] = south; // line buffers
14    west = center; center = east; } } // Propagate registers
```

(a) Delay buffering using cyclically indexed line buffers.

```
1 float sr[2*M + 1]; // Shift register buffer
2 for (int i = 0; i < N; ++i) {
3   #pragma PIPELINE
4   for (int j = 0; j < M; ++j) {
5     #pragma UNROLL
6     for (int k = 0; k < 2*M; ++k)
7       sr[k] = sr[k + 1]; // Shift the array left
8     sr[2*M] = memory[i][j]; // Append to the front
9     if (i > 1 && j > 0 && j < M - 1) // Initialize/drain
10      result[i-1][j] = 0.25*(sr[0] + sr[M-1] + sr[M+1] + sr[2*M]); } }
```

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

Listing 4: Two ways of implementing delay buffering on an $N \times M$ grid.

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 (lines 6-7). The computation accesses elements of this array using *constant indices only* (line 10), relying on the tool to infer the partitioning into individual buffers (akin to loop idiom recognition [25]) 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 [44], [39].

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 M cycles. We could therefore also use the shift register abstraction for this array. The same is true for the accumulation code in Lst. 3b.

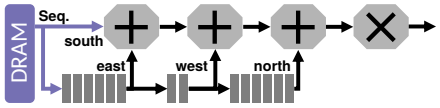


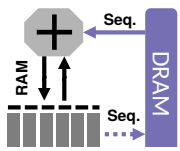
Fig. 4: A delay buffer for a 4-point stencil with three *taps*.

2.3 Random Access Buffering

When a program unavoidably needs to perform random accesses, we can buffer data in on-chip memory and perform random access to this fast memory instead of to slow off-chip memory. A random access buffer implemented with a general purpose replacement strategy will emulate a CPU-style cache; but to benefit from targeting a spatial system, it is usually more desirable to *specialize* the buffering strategy to the target application [45], [46]. This can enable off-chip memory accesses to be made contiguous by loading and storing data in stages (i.e., tiles), then exclusively performing random accesses to fast 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 hist[256] = {0}; // Array of bins
2 #pragma PIPELINE // Will have II=2
3 for (int i = 0; i < N; ++i) {
4   int bin = CalculateBin(memory[i]);
5   hist[bin] += 1; // Single cycle access
6 } // ...write result out to memory...
```

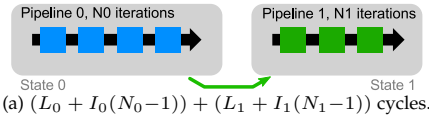


Listing 6: Random access to on-chip histogram buffer.

2.4 Pipelined Loop Fusion

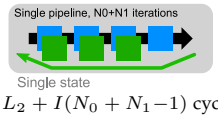
When two pipelined loops appear sequentially, we can fuse them into a single pipeline, while using loop guards to enforce any dependencies that might exist between them. This can result in a significant reduction in runtime, at little to no resource overhead. This transformation is closely related to loop fusion [47] from traditional software optimization.

```
// Pipelined loops executed sequentially
for (int i = 0; i < N0; ++i) Foo(i, /*...*/);
for (int i = 0; i < N1; ++i) Bar(i, /*...*/);
```



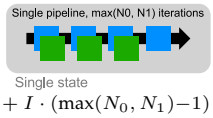
(a) $(L_0 + I_0(N_0-1)) + (L_1 + I_1(N_1-1))$ cycles.

```
for (int i = 0; i < N0+N1; ++i) {
  if (i < N0) Foo(i,      /*...*/);
        else Bar(i - N0, /*...*/); }
```



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

```
for (int i = 0; i < max(N0, N1); ++i) {
  if (i < N0) Foo(i, /*...*/);   // Omit ifs
  if (i < N1) Bar(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).

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 most cases): Loops can be fused by summing the loop bounds, using loop guards 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 can be fused by iterating to the maximum loop bound, and loop guards are placed as necessary to predicate each section (Lst. 5c).
3) Conditions 1 and 2 are met, **and** $N{=}N_0{=}N_1$ (same loop bounds): Loops bodies can be 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 Switching

The benefits of pipelined loop fusion can be extended to coarse-grained control flow by using *loop switching* (as opposed to loop *un*switching, which is a common transformation [25] on load/store architectures). Whereas instruction-based architectures attempt to only execute one branch of a conditional jump (via branch prediction on out-of-order processors), a conditional in a pipelined scenario will result in *both* branches being instantiated in hardware, regardless of whether/how often it is executed. The transformation of coarse-grained control flow into fine-grained control flow is implemented by the HLS tool by introducing *predication* to the pipeline, at no significant runtime penalty.

Lst. 7 shows a simple example of how the transformation fuses two pipelined loops in different branches into a single loop switching pipeline. The transformation applies to *any* pipelined code in either branch, following the principles described for pipelined loop fusion (§2.4 and Lst. 5).

The implications of pipelined loop switching are more subtle than the pure fusion examples in Lst. 5, as the total number of loop iterations is not affected (assuming the fused loop bound is set according to the condition, see line 1 in Lst. 7b). There *can* be a (tool-dependent) benefit from saving overhead logic by only implementing the orchestration and interfaces of a single pipeline, at the (typically minor) cost of the corresponding predication logic. More importantly, eliminating the coarse-grained control can enable other transformations that significantly benefit performance, such as fusion [§2.4] with adjacent pipelined loops, flattening nested loops [§2.6], and on-chip dataflow [§3.3].

```
if (condition)
  #pragma HLS PIPELINE
  for (int i = 0; i < N0; ++i)
    y[i] = Foo(x[i]);
else
  #pragma HLS PIPELINE
  for (int i = 0; i < N1; ++i)
    y[i] = Bar(x[i]);
```

(a) Coarse-grained control flow.

```
auto N = condition ? N0 : N1;
#pragma HLS PIPELINE
for (int i = 0; i < N; ++i) {
  if (condition)
    y[i] = Foo(x[i]);
  else
    y[i] = Bar(x[i]);
}
```

(b) Control flow absorbed into pipeline.

Listing 7: Pipelined loop switching absorbs coarse-grained control flow.

2.6 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. 8a 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 `Bar` 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 $L_1 + N_0N_1$, but for applications where N_0 is comparable to L_0, draining the inner pipeline can significantly impact the runtime (even if N_1 is large). By transforming the code such that all loops are *perfectly nested* (see Lst. 8b), the HLS tool can effectively *coalesce* the loops into a single pipeline, where next iteration of the *outer* loop can be executed immediately after the previous finishes.

```
for (int i = 0; i < N1; ++i) {
  #pragma PIPELINE
  for (int j = 0; j < N0; ++i)
    Foo(i, j);
  Bar(i); }
```

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

```
for (int i = 0; i < N1; ++i) {
  #pragma PIPELINE
  for (int j = 0; j < N0; ++i)
    Foo(i, j);
    if (j == N0 - 1) Bar(i); }
```

(b) $L_2 + N_0N_1-1$ cycles.

Listing 8: Before and after coalescing loop nest to avoid inner pipeline drains.

To perform the transformation in Lst. 8, we had to absorb `Bar` into the inner loop, adding a loop guard (line 5 in Lst. 8b), analogous 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.

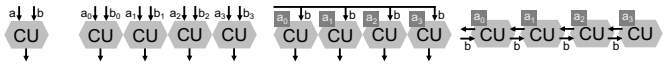
2.7 Inlining

In order to successfully pipeline a scope, all function calls within the code section must be pipelineable. This typically

requires "inlining" functions into each call site, creating dedicated hardware for each invocation, resulting in additional resources consumed for every additional callsite after the first. This replication is done automatically by HLS compilers on demand, but an additional `inline` pragma can be specified to directly "paste" the function body into the callsite during preprocessing, removing the function boundary during optimization and scheduling.

3 Scalability Transformations

Parallelism in HLS revolves around the *folding* of loops, achieved through *unrolling*. In Sec. 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.



(a) Before. (b) Horizontal unroll. (c) Vertical unroll. (d) Dataflow.

Fig. 5: Horizontal unrolling, vertical unrolling, and dataflow, as means to increase parallelism. Rectangles represent buffer space, such as registers or on-chip RAM. **Horizontal:** four independent inputs processed in parallel. **Vertical:** one input is combined with multiple buffered values. **Dataflow:** similar to vertical, but input or partial results are streamed through a pipeline rather than broadcast.

3.1 Horizontal Unrolling (Vectorization)

We implement vectorization-style parallelism with HLS by "horizontally" unrolling loops in pipelined sections, or by 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 an inner loop without further reordering or drastic changes to the nested loop structure. Vectorization is more powerful in HLS than SIMD operations on load/store architectures, as the unrolled compute units are not required to be homogeneous, and the number of units are not constrained to fixed sizes. Horizontal unrolling increases bandwidth utilization by explicitly exploiting spatial locality, allowing more efficient accesses to off-chip memory such as DRAM.

Lst. 9 shows two functionally equivalent ways of vectorizing a loop over N elements by a horizontal unrolling factor of W. Lst. 9a strip-mines a loop into chunks of W and unrolls the inner loop fully, while Lst. 9b uses partial unrolling by specifying an unroll factor in the pragma. As a third option, explicit vector types can be used, such as those built into OpenCL (e.g., `float4` or `int16`), or custom vector classes [48]. These provide less flexibility, but are more concise and are sufficient for most applications.

In practice, the unrolling factor W [operand/cycle] is constrained by the bandwidth B [Byte/s] available to the

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

(a) Using strip-mining.

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

(b) Using partial unrolling.

Listing 9: Two variants of vectorization by factor W using loop unrolling.

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 unrolled logic, and S [Byte/operand] is the operand size in bytes. Horizontal unrolling is 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).

3.2 Vertical Unrolling

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 "vertically" through unrolling [49], [38], [50]. *This is the most potent source of parallelism on hardware architectures*, as it can conceptually scale indefinitely with available silicon when enough reuse is possible. Viewed from the paradigm of cached architectures, the opportunity for this transformation arises from temporal locality in loops. Vertical unrolling 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. In comparison, horizontal unrolling 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 vertical unrolling is essential to whether the design 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 delay buffering [§2.2] or random-access [§2.3] buffering scheme. Then, if additional reuse is possible, vertically unroll 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

```
1  for (int n = 0; n < N / P; ++n) { // Folded by unrolling factor P
2    for (int m = 0; m < M / T; ++m) { // Tiling
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 tile of B
12         #pragma UNROLL
13         for (int p = 0; p < P; ++p) // P-fold vertical unrolling
14           acc[t][p] += a_buffer[p] * B[k][m*T+t];
15     } /* ...write back 2D tile of C... */ } }
```

Listing 10: P-fold vertical unrolling of matrix multiplication.

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 parallelism (as well as buffer space required for the partial sums) by a factor of P (lines 12-14).

3.3 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 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 *fan-out/fan-in* 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 parts and adding more pipeline stages.
- The fan-in and fan-out of control signals (i.e., stall, reset) *within* each module is reduced, reducing the risk of these signals constraining the maximum achievable frequency.

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 always FIFO interfaces, and support standard queue operations `Push`, `Pop`, and sometimes `Empty`, `Full`, and `Size` operations. They occupy the same register or block memory resources as other buffers (Sec. 2.2/Sec. 2.3).

The mapping from source code to PEs differs between HLS tools, but is manifested when functions are connected using channels. In the following example, 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. Note that this **functionally diverges from C++ semantics** without additional abstraction [48], as each function in the dataflow scope is executed in parallel in hardware, rather than in the sequence specified in the imperative code. In Intel OpenCL, dataflow 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 vertical unrolling (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 a distinct PE, connected in a chain via channels [37], [51], [38]. 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 vertically unroll to generate P instances of the PE (shown in Lst. 11b), effectively increasing the throughput of the kernel by a factor of P, and consequently reducing 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* [52], [53].

For architectures/HLS tools where large fan-out is an issue for compilation or routing, an already replicated design can be transformed to a dataflow architecture. 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 [50]. $\boldsymbol{B}$ 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 [38]. Rather than making the program faster, this lets the already fast architecture support arbitrarily large problem sizes. This is in contrast to loop tiling on CPU and GPU, where tiling is used to increase performance. Common to both paradigms is that they fundamentally aim to meet fast memory constraints. As with horizontal and vertical unrolling, 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

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

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

```
1 #pragma DATAFLOW // Schedule nested functions as parallel modules
2 void SystolicStencil(const float in[], float out[], int T) {
3   FIFO<float> pipes[P + 1]; // Assume P is given at compile time
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], T); // Forms a chain
8   WriteMemory(pipes[P], out); } // Tail
```

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

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

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 line buffers or shift register, so they a no longer proportional to the problem size.

4 Memory Access Transformations

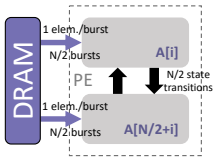
When an HLS design has been pipelined, scheduled, and unrolled as desired, the memory access pattern has been established. In the following, we describe transformations that optimize the efficiency of off-chip memory accesses in the HLS code. For memory bound codes in particular, this is critical for 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 [54]. In many cases, such as for independent reads, this is not an inherent 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 [55]).

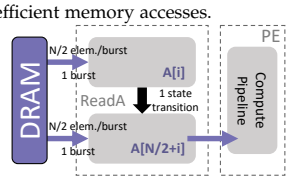
In Lst. 12a, the same memory (i.e., hardware memory interface) is accessed twice in the inner loop. In the worst case, the program will issue two $4\,\text{Byte}$ memory requests every iteration, resulting in poor memory performance, and preventing pipelining of the loop. In software, this problem is typically mitigated by caches, always fetching at least one cache line. If we instead read the two sections of A sequentially (or in larger 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/2 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 Buffering

When dealing with memory interfaces with an inconsistent data rate, such as DRAM, it can be beneficial to request and buffer accesses earlier and/or at a more aggressive pace than what is consumed or produced by the computational elements. For memory reads, this can be done by reading ahead of the kernel 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 or from computational elements, respectively, or 2) increasing the clock rate of modules accessing memory with respect to the computational elements.

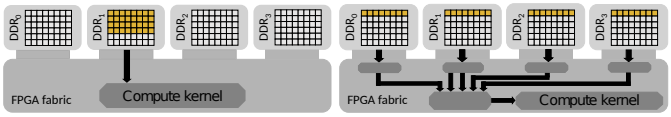
The memory access function Lst. 12c allows long bursts to the interface of A, but receives the data on a narrow bus at $W \cdot S_{\text{int}} = (1 \cdot 4)\ \text{Byte/cycle}$. In general, this limits the bandwidth consumption to $f \cdot W S_{\text{int}}$ at frequency f, which is likely to be less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

4.3 Memory Striping

When multiple memory banks with dedicated channels (e.g., multiple DRAM modules or HBM lanes) 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 memory modules requesting memory from their

respective interfaces, pushing to FIFO buffers that are read in parallel and combined by another module (for writing: in reverse), 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 four DRAM modules. The Intel OpenCL compiler [19] applies this transformation by default.



(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 $64\,\text{bit}$ integers to $48\,\text{bit}$ 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 [56].
- 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 [34] can provide sufficient speedup for inference that the increased number of weights makes up for the loss of precision per weight.

5 Software Transformations in HLS

In addition to the transformations described in the sections above, we include an overview of how well-known CPU-oriented transformations apply to HLS, based on the compiler transformations compiled by Bacon et al. [25]. 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.

	CPU-Oriented Transformations and how they apply to HLS codes.
Directly related to HLS transformations	**Loop interchange** [57], [47] is used to resolve loop-carried dependencies [§2]. **Strip-mining** [58], **loop tiling** [59], [47], and **cycle shrinking** [60] are central components of many HLS transformations [§2.1, §3.1, §3.2, §2.1.2]. **Loop distribution** and **loop fission** [61], [47] are used to separate differently scheduled computations to allow pipelining [§3.3]. **Loop fusion** [62], [47], [63] is used for merging pipelines [§2.4]. **Loop unrolling** [64] is used to generate parallel hardware [§3.1, §3.2]. **Software pipelining** [65] is used by HLS tools to schedule code sections according to operation interdependencies to form *hardware* pipelines. **Loop coalescing/flattening/collapsing** [66] saves pipeline drains in nested loops [§2.6]. **Reduction recognition** prevents loop-carried dependencies when accumulating [§2.1]. **Loop idiom recognition** is relevant for HLS backends, for example to recognize shift registers [§2.2] in the Intel OpenCL compiler [19]. **Procedure inlining** is used to remove function call boundaries [§2.7]. **Procedure cloning** is frequently used by HLS tools when inlining [§2.7] to specialize each function "call" with values that are known at compile-time. **Loop unswitching** [67] is rarely advantageous; its *opposite* is beneficial [§2.6, §2.4]. **Loop peeling** is rarely advantageous; its **opposite** is beneficial to allow coalescing [§2.6]. **SIMD transformations** is done in HLS via horizontal unrolling [§3.1]. **Short-circuiting**: while the logic for both boolean operands must always be instantiated in hardware, dynamically scheduling branches [68] can effectively "short-circuit" otherwise deep, static pipelines.
Same or similar in HLS	**Loop-based strength reduction** [69], [70], [71], **Induction variable elimination** [72], **Unreachable code elimination** [72], **Useless-code elimination** [72], **Dead-variable elimination** [72], **Common-subexpression elimination** [72], **Constant propagation** [72], **Constant folding** [72], **Copy propagation** [72], **Forwarding substitution** [72], **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)** [72] does not save hardware in itself, but can save memory operations. **Loop normalization** can be useful as an intermediate transformation. **Loop reversal** [72], **array padding and contraction**, **scalar expansion**, and **scalar replacement** yield the same benefits as in software. **Loop skewing** [72] 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/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 [73].
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. **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.

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 End-to-End Examples

To showcase the transformations presented in this work and provide a "hands-on" opportunity for seeing HLS optimizations applied in practice, we will describe the optimization process on a sample set of classical HPC kernels, available as open source repositories on github[1]. These kernels are

1. https://github.com/spcl?q=hls

written in C++ for Xilinx Vivado HLS [12] with hlslib [48] extensions, and are built and run using the Xilinx Vitis environment. For each example, we will describe the sequence of transformations applied, and give the resulting performance at each major stage.

The included benchmarks were run on an Alveo U250 board, which houses a Xilinx UltraScale+ XCU250-FIGD2104-2L-E FPGA and four $2400\,\mathrm{MT/s}$ DDR4 banks (we utilize 1-2 banks for the examples here). The chip consists of four almost identical chiplets with limited interconnect between them, where each chiplet is connected to one of the DDR4 pinouts. This multi-chiplet design allows more resources (1728K LUTs and 12,288 DSPs), 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. Kernels were compiled for the xilinx_u250_xdma_201830_2 shell with Vitis 2019.2 and executed with version 2.3.1301 of the Xilinx Runtime (XRT). All benchmarks are included in Fig. 7, and the resource utilization of each kernel is shown in Fig. 8.

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 [38]. We show the optimization of a 4-point 2D stencil based on Lst. 4. Benchmarks are shown in Fig. 7, and use single precision floating point, iterating over a 8192×8192 domain. We first measure a naive implementation, where all neighboring cells are accessed directly from the input array, which results in no data reuse and heavy interface contention on the input array. We then apply the following optimization steps:

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) Spatial locality is exploited by introducing vectorization [§3.1]. To efficiently use memory bandwidth, we use memory extraction [§4.1], buffering [§4.2], and striping [§4.3] from two DDR banks.
3) To exploit temporal locality, we replicate the vectorized PE by vertical unrolling [§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 $W = 16\,\mathrm{operands/cycle}$ for the kernel) exploits spatial locality through additional bandwidth usage. The vertical unrolling and dataflow step scales the design to exploit available hardware resources on the chip, until limited by placement and routing. The final implementation is available on github[2].

6.2 Matrix Multiplication Code

We consider the optimization process of a matrix multiplication kernel using transformations presented here. Benchmark for 8192×8192 matrices across stages of optimization are shown in Fig. 7. Starting from a naive implementation (Lst. 1a), the following optimization stages were applied:

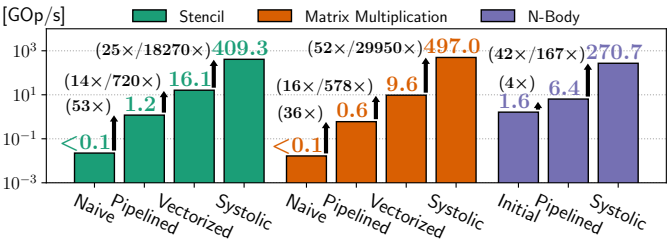


Fig. 7: Performance progression of kernels when applying transformations. Parentheses show speedup over previous version, and cumulative speedup.

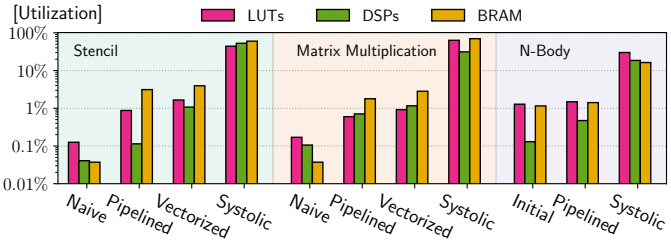


Fig. 8: Resource usage of kernels from Fig. 7 as fractions of available resources. The maxima are taken as 1728K LUTs, 12,288 DSPs, and 2688 BRAM.

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 are fused [§2.4], allowing us to coalesce the three nested loops [§2.6].
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 vertically unroll by buffering multiple values of $\boldsymbol{A}$, applying them to streamed in values of $\boldsymbol{B}$ in parallel [§3.2]. To avoid high fan-out, we partition buffered elements 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 results in a throughput of $\sim$1 cell per cycle. Vectorization multiplies the performance by W, set to 16 in the benchmarked kernel. The performance of the vertically unrolled dataflow kernel is only limited by placement and routing due to high resource usage on the chip. The final implementation achieves state-of-the-art performance on the target architecture [50], and is available on github[3].

6.3 N-Body Code

Finally, we show an N-body code in 3 dimensions, using single precision floating point types and iterating over 16,128 bodies. Since Vivado HLS does not allow memory accesses of a width that is not a power of two, memory extraction [§4.1] and buffering [§4.2] was included in the first stage, to support 3-vectors of velocity. We then performed the following transformations:

1) The loop-carried dependency on the acceleration accumulation is resolved by applying tiled accumulation interleaving [§2.1.2], pipelining across $T \geq L_+$ different resident particles applied to particles streamed in.
2) To scale up the performance, we further multiply the number of resident particles, this time replicating compute through vertical unrolling [§3.2] of the outer loop

2. https://github.com/spcl/stencil_hls/

3. https://github.com/spcl/gemm_hls

into P parallel processing element arranged in a systolic array. Each element holds T resident particles, and particles are streamed [§3.3] through the PEs.

The second stage gains a factor of $4\times$ corresponding to the latency of the interleaved accumulation, followed by a factor of $42\times$ from unrolling units across the chip. $T \geq L_+$ can be used to regulate the arithmetic intensity of the kernel. The bandwidth requirements can be reduced further by storing more resident particles on the chip, scaling up to the full fast memory usage of the FPGA. The tiled accumulation interleaving transformation thus enables not just pipelining of the compute, but also minimization of I/O. The optimized implementation is available on github[4].

These examples demonstrate the impact of different transformations on a reconfigurable hardware platform. In particular, enabling pipelining, regularizing memory accesses, and vertical unrolling are shown to be central components of scalable hardware architectures. The dramatic speedups over naive codes also emphasize that HLS tools do not yield competitive performance out of the box, making it critical to perform further transformations. For additional examples of optimizing HLS codes, we refer to the numerous works applying HLS optimizations listed below.

7 Related Work

Optimized applications. Much work has been done in optimizing C/C++/OpenCL HLS codes for FPGA, such as stencils [38], [39], [40], [74], [75], deep neural networks [76], [77], [35], [36], [34], matrix multiplication [78], [75], [50], [79], graph processing [80], [81], networking [82], light propagation for cancer treatment [46], and protein sequencing [49], [83]. These works optimize the respective applications using transformations described here, such as delay buffering, random access buffering, vectorization, vertical unrolling, tiling for on-chip memory, and dataflow.

Transformations. Zohouri et al. [84] use the Rodinia benchmark to evaluate the performance of OpenCL codes targeting FPGAs, employing optimizations such as SIMD vectorization, sliding-window buffering, accumulation interleaving, and compute unit replication across multiple kernels. We present a generalized description of a superset of these transformations, along with concrete code examples that show how they are applied in practice. The DaCe framework [85] exploits information on explicit dataflow and control flow to perform a wide range of transformations, and code generates efficient HLS code using vendor-specific pragmas and primitives. Kastner et al. [86] go through the implementation of many HLS codes in Vivado HLS, focusing on algorithmic optimizations. da Silva et al. [87] explore using modern C++ features to capture HLS concepts in a high-level fashion. Lloyd et al. [88] 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.

Directive-based frameworks. High-level, directive-based frameworks such as OpenMP and OpenACC have been proposed as alternative abstractions for generating FPGA kernels. Leow et al. [89] implement an FPGA code generator from OpenMP pragmas, primarily focusing on correctness in implementing a range of OpenMP pragmas. Lee et al. [90] present an OpenACC to OpenCL compiler, using Intel OpenCL as a backend. The authors implement horizontal and vertical unrolling, pipelining and dataflow by introducing new OpenACC clauses. Papakonstantinou et al. [91] generate HLS code for FPGA from directive-annotated CUDA code.

Optimizing HLS compilers. Mainstream HLS compilers automatically apply many of the well-known software transformations in Tab. 2 [22], [92], [93], 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 [94] uses high-level acceleration directives to automatically perform some of the transformations described here, as source-to-source transformations to underlying HLS code. Polyhedral compilation is a popular framework for optimizing CPU and GPU loop nests [55], and has also been applied to HLS for FPGA for optimizing data reuse [95]. Such techniques may prove valuable in automating, e.g., memory extraction and tiling transformations. While most HLS compilers rely strictly on static scheduling, Dynamatic [68] considers dynamically scheduling state machines and pipelines to allow reducing the number of stages executed at runtime.

Domain-specific frameworks. Implementing programs in domain specific languages (DSLs) can make it easier to detect and exploit opportunities for advanced transformations. Darkroom [30] generates optimized HDL for image processing codes, and the popular image processing framework Halide [31] has been extended to support FPGAs [96], [97]. Luzhou et al. [53] and StencilFlow [44] propose frameworks for generating stencil codes for FPGAs. These frameworks rely on optimizations such as delay buffering, dataflow, and vertical unrolling, 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. [98], and the FROST [99] DSL framework.

Other approaches. There are other approaches than C/C++/OpenCL-based HLS languages to addressing the productivity issues of hardware design: Chisel/FIRRTL [100], [101] maintains the paradigm of behavioral programming known from RTL, but provides modern language and compiler features. This caters to developers who are already familiar with hardware design, but wish to use a more expressive language. In the Maxeler ecosystem [102], kernels are described using a Java-based language, but rather than transforming imperative code into a behavioral equivalent, the language provides a DSL of hardware concepts that are instantiated using object-oriented interfaces. By constraining the input, this encourages developers to write code that maps well to hardware, but requires learning a new language exclusive to the Maxeler ecosystem.

8 Toolflow of Xilinx vs. Intel

When choosing a toolflow to start designing hardware with HLS, it is useful to understand two distinct approaches by the two major vendors: Intel OpenCL wishes to enable *writing accelerators using software*, making an effort to abstract away low-level details about the hardware, and

4. https://github.com/spcl/nbody_hls

present a high-level view to the programmer; whereas Xilinx' Vivado HLS provides *a more productive way of writing hardware*, by means of a familiar software language. Xilinx uses OpenCL as a vehicle to interface between FPGA and host, but implements the OpenCL compiler itself as a thin wrapper around the C++ compiler, whereas Intel embraces the OpenCL paradigm as their frontend (although they encourage writing single work item kernels [103], effectively preventing reuse of OpenCL kernels written for GPU).

Vivado HLS has a stronger coupling between the HLS source code and the generated hardware. This requires the programmer to write more annotations and boilerplate code, but can also give them stronger feeling of control. Conversely, the Intel OpenCL compiler presents convenient abstracted views, saves boilerplate code (e.g., by automatically pipelining sections), and implements efficient substitutions by detecting common patterns in the source code (e.g., to automatically perform memory extraction [§4.1]). The downside is that developers end up struggling to write or generate code in a way that is recognized by the tool's "black magic", in order to achieve the desired result. Finally, Xilinx' choice to allow C++ gives Vivado HLS an edge in expressibility, as (non-virtual) objects and templating turns out to be a useful tool for abstracting and extending the language [48]. Intel offers a C++-based HLS compiler, but does not (as of writing) support direct interoperability with the OpenCL-driven accelerator flow.

9 Conclusion

The transformations known from software are insufficient to optimize HPC kernels targeting spatial computing systems. We have proposed a new 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. Performance and compiler engineers can benefit from these guidelines, transformations, and the presented cheat sheet as a common toolbox for developing high performance hardware using HLS.

Acknowledgements

This work was supported by the European Research Council under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880). The authors wish to thank Xilinx and Intel for helpful discussions; Xilinx for generous donations of software, hardware, and access to the Xilinx Adaptive Compute Cluster (XACC) at ETH Zurich; the Swiss National Supercomputing Center (CSCS) for providing computing infrastructure; and Tal Ben-Nun for valuable feedback on iterations of this manuscript.

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," *Scientific Programming*, 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. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. Accessed May 15, 2020.
[14] A. Canis *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *FPGA*, 2011.
[15] Mentor Graphics. Catapult high-level synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls. Accessed May 15, 2020.
[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] R. Nikhil, "Bluespec system Verilog: efficient, correct RTL from high level specifications," in *MEMOCODE*, 2004.
[21] J. Auerbach *et al.*, "Lime: A Java-compatible and synthesizable language for heterogeneous architectures," in *OOPSLA*, 2010.
[22] ——, "A compiler and runtime for heterogeneous computing," in *DAC*, 2012.
[23] J. Hammarberg and S. Nadjm-Tehrani, "Development of safety-critical reconfigurable hardware with Esterel," *FMICS*, 2003.
[24] M. B. Gokhale *et al.*, "Stream-oriented FPGA computing in the Streams-C high level language," in *FCCM*, 2000.
[25] D. F. Bacon *et al.*, "Compiler transformations for high-performance computing," *CSUR*, 1994.
[26] S. Ryoo *et al.*, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *PPoPP*, 2008.
[27] G. D. Smith, *Numerical solution of partial differential equations: finite difference methods*, 1985.
[28] A. Taflove and S. C. Hagness, "Computational electrodynamics: The finite-difference time-domain method," 1995.
[29] C. A. Fletcher, *Computational Techniques for Fluid Dynamics 2*, 1988.
[30] J. Hegarty *et al.*, "Darkroom: compiling high-level image processing code into hardware pipelines." *TOG*, 2014.
[31] J. Ragan-Kelley *et al.*, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *PLDI*, 2013.
[32] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CSUR*, 2019.
[33] G. Lacey *et al.*, "Deep learning on FPGAs: Past, present, and future," *arXiv:1602.04283*, 2016.
[34] M. Courbariaux *et al.*, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv:1602.02830*, 2016.
[35] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *FPGA*, 2017.
[36] M. Blott *et al.*, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *TRETS*, 2018.
[37] H. Fu and R. G. Clapp, "Eliminating the memory bottleneck: An FPGA-based solution for 3d reverse time migration," in *FPGA*, 2011.
[38] H. R. Zohouri *et al.*, "Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL," in *FPGA*, 2018.
[39] H. M. Waidyasooriya *et al.*, "OpenCL-based FPGA-platform for stencil computation and its optimization methodology," *TPDS*, May 2017.

[40] Q. Jia and H. Zhou, "Tuning stencil codes in OpenCL for FPGAs," in *ICCD*, 2016.

[41] X. Niu *et al.*, "Exploiting run-time reconfiguration in stencil computation," in *FPL*, 2012.

[42] ——, "Dynamic stencil: Effective exploitation of run-time resources in reconfigurable clusters," in *FPT*, 2013.

[43] J. Fowers *et al.*, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *FPGA*, 2012.

[44] J. de Fine Licht *et al.*, "StencilFlow: Mapping large stencil programs to distributed spatial computing systems," in *CGO*, 2021.

[45] X. Chen *et al.*, "On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs," in *FPL*, 2019.

[46] T. Young-Schultz *et al.*, "Using OpenCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator," in *FPGA*, 2020.

[47] D. J. Kuck *et al.*, "Dependence graphs and compiler optimizations," in *POPL*, 1981.

[48] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design," *arXiv:1910.04436*, 2019.

[49] S. O. Settle, "High-performance dynamic programming on FPGAs with OpenCL," in *HPEC*, 2013.

[50] J. de Fine Licht *et al.*, "Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis," in *FPGA*, 2020.

[51] K. Sano *et al.*, "Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth," *TPDS*, 2014.

[52] H. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings*, 1978.

[53] W. Luzhou *et al.*, "Domain-specific language and compiler for stencil computation on fpga-based systolic computational-memory array," in *ARC*, 2012.

[54] T. Kenter *et al.*, "OpenCL-based FPGA design to accelerate the nodal discontinuous Galerkin method for unstructured meshes," in *FCCM*, 2018.

[55] T. Grosser *et al.*, "Polly – performing polyhedral optimizations on a low-level intermediate representation," *PPL*, 2012.

[56] U. Sinha, "Enabling impactful DSP designs on FPGAs with hardened floating-point implementation," *Altera White Paper*, 2014.

[57] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *SIGPLAN*, 1984.

[58] M. Weiss, "Strip mining on SIMD architectures," in *ICS*, 1991.

[59] M. D. Lam *et al.*, "The cache performance and optimizations of blocked algorithms," 1991.

[60] C. D. Polychronopoulos, "Advanced loop optimizations for parallel computers," in *ICS*, 1988.

[61] D. J. Kuck, "A survey of parallel machine organization and programming," *CSUR*, Mar. 1977.

[62] A. P. Yershov, "ALPHA – an automatic programming system of high efficiency," *J. ACM*, 1966.

[63] M. J. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertation, 1982.

[64] J. J. Dongarra and A. R. Hinds, "Unrolling loops in Fortran," *Software: Practice and Experience*, 1979.

[65] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *PLDI*, 1988.

[66] C. D. Polychronopoulos, "Loop coalescing: A compiler transformation for parallel machines," Tech. Rep., 1987.

[67] F. E. Allen and J. Cocke, *A catalogue of optimizing transformations*, 1971.

[68] L. Josipović *et al.*, "Dynamically scheduled high-level synthesis," in *FPGA*, 2018.

[69] J. Cocke and K. Kennedy, "An algorithm for reduction of operator strength," *CACM*, 1977.

[70] R. Bernstein, "Multiplication by integer constants," *Softw. Pract. Exper.*, 1986.

[71] G. L. Steele, "Arithmetic shifting considered harmful," *ACM SIGPLAN Notices*, 1977.

[72] A. V. Aho *et al.*, "Compilers, principles, techniques," *Addison Wesley*, 1986.

[73] T. De Matteis *et al.*, "Streaming message interface: High-performance distributed memory programming on reconfigurable hardware," in *SC*, 2019.

[74] D. Weller *et al.*, "Energy efficient scientific computing on FPGAs using OpenCL," in *FPGA*, 2017.

[75] A. Verma *et al.*, "Accelerating workloads on FPGAs via OpenCL: A case study with opendwarfs," Tech. Rep., 2016.

[76] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *FPGA*, 2016.

[77] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *FPGA*, 2017.

[78] E. H. D'Hollander, "High-level synthesis optimization for blocked floating-point matrix multiplication," *SIGARCH*, 2017.

[79] P. Gorlani *et al.*, "OpenCL implementation of Cannon's matrix multiplication algorithm on Intel Stratix 10 FPGAs," in *ICFPT*, 2019.

[80] M. Besta *et al.*, "Graph processing on FPGAs: Taxonomy, survey, challenges," *arXiv:1903.06697*, 2019.

[81] ——, "Substream-centric maximum matchings on FPGA," in *FPGA*, 2019.

[82] H. Eran *et al.*, "Design patterns for code reuse in HLS packet processing pipelines," in *FCCM*, 2019.

[83] E. Rucci *et al.*, "Smith-Waterman protein search with OpenCL on an FPGA," in *Trustcom/BigDataSE/ISPA*, 2015.

[84] H. R. Zohouri *et al.*, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *SC*, 2016.

[85] T. Ben-Nun *et al.*, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *SC*, 2019.

[86] R. Kastner *et al.*, "Parallel programming for FPGAs," *arXiv:1805.03648*, 2018.

[87] J. S. da Silva *et al.*, "Module-per-Object: a human-driven methodology for C++-based high-level synthesis design," in *FCCM*, 2019.

[88] T. Lloyd *et al.*, "A case for better integration of host and target compilation when using OpenCL for FPGAs," in *FSP*, 2017.

[89] Y. Y. Leow *et al.*, "Generating hardware from OpenMP programs," in *FPT*, 2006.

[90] S. Lee *et al.*, "OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing," in *IPDPS*, 2016.

[91] A. Papakonstantinou *et al.*, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *SASP*, 2009.

[92] S. Gupta *et al.*, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *VLSID*, 2003.

[93] ——, "Coordinated parallelizing compiler optimizations and high-level synthesis," *TODAES*, 2004.

[94] J. Cong *et al.*, "Source-to-source optimization for HLS," in *FPGAs for Software Programmers*, 2016.

[95] L.-N. Pouchet *et al.*, "Polyhedral-based data reuse optimization for configurable computing," in *FPGA*, 2013.

[96] J. Pu *et al.*, "Programming heterogeneous systems from an image processing DSL," *TACO*, 2017.

[97] J. Li *et al.*, "HeteroHalide: From image processing DSL to efficient FPGA acceleration," in *FPGA*, 2020.

[98] D. Koeplinger *et al.*, "Automatic generation of efficient accelerators for reconfigurable hardware," in *ISCA*, 2016.

[99] E. D. Sozzo *et al.*, "A common backend for hardware acceleration on FPGA," in *ICCD*, 2017.

[100] J. Bachrach *et al.*, "Chisel: constructing hardware in a scala embedded language," in *DAC*, 2012.

[101] A. Izraelevitz *et al.*, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *ICCAD*, 2017.

[102] Maxeler Technologies, "Programming MPC systems (white paper)," 2013.

[103] Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide, UG-OCL003, revision 2020.04.1. Accessed May 15, 2020.

Johannes de Fine Licht is a PhD student at ETH Zurich. His research topics revolve around spatial computing systems in HPC, and include programming models, applications, libraries, and enhancing programmer productivity.

Maciej Besta is a PhD student at ETH Zurich. His research focuses on understanding and accelerating large-scale irregular graph processing in any type of setting and workload.

Simon Meierhans is studying for his MSc degree at ETH Zurich. His interests include randomized and deterministic algorithm and data structure design.

Torsten Hoefler is a professor at ETH Zurich, where he leads the Scalable Parallel Computing Lab. His research aims at understanding performance of parallel computing systems ranging from parallel computer architecture through parallel programming to parallel algorithms.