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

JOHANNES DE FINE LICHT, ETH Zurich, Switzerland
SIMON MEIERHANS, ETH Zurich, Switzerland
TORSTEN HOEFLER, ETH Zurich, Switzerland

Specialized hardware architectures promise a major step in performance and energy efficiency over the traditional load/store devices currently employed in large scale computing systems. The adoption of high-level synthesis (HLS) from languages such as C/C++ and OpenCL has greatly increased programmer productivity when designing for such platforms. While this has enabled a wider audience to target specialized hardware, the optimization principles known from software design are no longer sufficient to implement high-performance codes, due to fundamental differences between software and hardware architectures. In this work, we propose a set of optimizing transformations for HLS, targeting scalable and efficient architectures for high-performance computing (HPC) applications. We show how these can be used to efficiently exploit pipelining, on-chip distributed fast memory, and on-chip streaming dataflow, allowing for massively parallel architectures with little off-chip data movement. To quantify the effect of our transformations, we use them to optimize a set of high-throughput FPGA kernels, demonstrating that they are sufficient to scale up parallelism within the hardware constraints of the target device. With the transformations covered, we hope to establish a common framework for performance engineers, compiler developers, and hardware developers, to tap into the performance potential offered by specialized hardware architectures using HLS.

1 MOTIVATION

Since the recent ending of Dennard scaling, when the power consumption of digital circuits stopped scaling with their size, compute devices become increasingly limited by their power consumption [80]. In fact, the shrinking feature size even increases the loss in the metal layers of modern microchips. The load/store architectures in use-today suffer mostly from the cost of data movement and addressing [30]. Other approaches such as dataflow architectures have not been widely successful due to the varying granularity of applications [22]. However, *application-specific* dataflow can be used to lay out memory, such as registers and buffers, to fit the *specific structure* of the computation and minimize data movement. Reconfigurable architectures, such as FPGAs, can be used to implement application-specific dataflow [10, 65, 71], but they are too hard to program [6]. Traditional hardware design languages, such as VHDL and Verilog, do not benefit from the rich set of software engineering techniques that improve programmer productivity and code reliability. For these reasons, the community is beginning to embrace hardware development techniques based on traditional procedural languages such as C or C++. These tools and languages are commonly called high-level synthesis (HLS) [13, 45]. In this way, HLS bridges the gap between hardware and software development and enables basic performance portability implemented in their compilation systems. For example, HLS programmers do not have to worry how exactly a floating point operation is implemented on the target hardware. For the same source code, a good compiler will generate the necessary circuit when compiled for an Intel Stratix V FPGA, and will transparently use optimized floating point cores when compiled for a Stratix 10. However, compiler optimizations are fundamentally limited. Numerous HLS systems [46, 48] synthesize hardware

Authors' addresses: Johannes de Fine Licht, ETH Zurich, Rämistrasse 101, Zurich, 8092, Switzerland, definelicht@inf.ethz.ch; Simon Meierhans, ETH Zurich, Rämistrasse 101, Zurich, 8092, Switzerland, mesimon@ethz.ch; Torsten Hoefler, ETH Zurich, Rämistrasse 101, Zurich, 8092, Switzerland, htor@inf.ethz.ch.

designs from C/C++ [11, 24, 32, 47, 54, 85], OpenCL [52, 72, 82] and others [3, 4, 23, 28, 49]. All-in-all, HLS provides a viable path-way for the software and hardware communities to meet and address each other's concerns.

For many applications, compute performance is a primary goal which is achieved through careful tuning by highly-specialized performance engineers. To guide these engineers, optimizing transformations for CPU [5] and GPU [61] are well-understood. For HLS, a comparable collection of guidelines and principles for code optimization has yet to be established. Optimizing codes for hardware implementations is drastically different from optimizing codes for a fixed architecture. In fact, the optimization space is larger because it contains known software optimizations, and in addition, programmers can change the microarchitecture and design application-specific circuits in HLS. Thus, the established set of transformations is not sufficient because it does not consider aspects of optimized hardware design, such as pipelining.

In this work, we define a set of optimizing transformations that compilers or performance engineers can apply in order to improve the performance of hardware layouts generated from HLS codes. For this, we discuss how code transformations known from tuning for fixed hardware apply to HLS. Furthermore, we introduce a set of optimizing transformations at the HLS level that generate pipelined hardware layouts with optimized buffer distributions. We show that these key transformations mainly aim at laying out the buffers into an *application-specific* dataflow architecture that efficiently uses the available distributed storage and computation.

1.1 Key transformations for high-level synthesis

We propose a set of optimizing transformations that are fundamental to designing scalable and efficient hardware kernels in HLS. These transformations are often composed of multiple basic source code transformations, such as strip-mining and loop interchange, that achieve the desired patterns, and we will list these when relevant. We divide them into four categories, as given below:

Pipeline-enabling transformations:

(1) **Transposition**: resolve loop-carried dependencies by transposing the iteration space.
(2) **Interleaving**: interleave accumulations of outer loop or use two-phase accumulation.
(3) **Cross-input pipelining**: interleave accumulations across different inputs.
(4) **Inlining**: functions and operators in pipelined sections must be inlined.
(5) **Cyclic buffering**: use FIFO buffers to exploit fast memory in pipelined applications.
(6) **Pipelined loop flattening/coalescing**: merge (perfectly or imperfectly) nested loops to avoid pipeline drains.
(7) **Pipelined loop fusion**: fuse consecutive pipelines to merge their cycle counts.

Scalability transformations:

(1) **Vectorization**: single instruction multiple data (SIMD) parallelization.
(2) **Replication**: increase amount of compute logic to scale up performance without spending bandwidth by exploiting on-chip memory.
(3) **Streaming dataflow**: partition kernel into multiple processing elements to separate scheduling, improve placement and routing results, and optimize memory performance.
(4) **Tiling**: fit large domain sizes into available fast memory.

Secondary transformations:

(1) **Memory access extraction**: extract memory accesses from computations, allowing them to be optimized separately.
(2) **Memory oversubscription**: amortize bandwidth from nondeterministic data sources by accessing memory at a higher rate than required by the kernel.

(3) **Memory striping**: stripe memory onto multiple banks to multiply access bandwidth.
(4) **Type demotion**: demote to cheaper data types when allowed by precision requirements.

Software transformations: traditional software transformations that apply directly to HLS.

We will show how transformations can be applied manually by a performance engineer by directly modifying the source code, by giving examples before and after a transformation is applied, but many are also amenable to automation in an optimizing compiler. Before diving into the transformations, however, we need to establish the metrics for performance in a pipelined design, as a target of optimization in the following.

1.2 Basics of pipelining

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

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

To quantify the importance of pipelining, we consider the number of cycles C it takes to execute a pipeline with latency L (both in [cycles]), taking N inputs, with an initiation interval of I [cycles], assuming a reliable producer and consumer at either end, which is exactly:

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

The time to execute all N iterations with clock rate f [cycles/s] of this pipeline is thus C/f. By formulating our program as a pipeline, optimization can be condensed to three primary goals:

(A) **Perfect pipelining**: achieve $I = 1$ cycle for all essential components, i.e., ensure that all pipelines run at maximum throughput.
(B) **Scaling/folding**: fold N by scaling up the parallelism of the design, thus cutting the total number of pipeline iterations required to execute the program.
(C) **Saturation**: saturate pipelines for the majority of the runtime to avoid stalls.

The rest of this paper is organized as follows. Section 2 will cover transformations that enable (A), and Section 3 covers transformations that achieve (B). Together, these make up the core of hardware optimization, as all these transformations will apply to nearly every HPC program. Section 4 covers transformations that contribute to (C), as well as more situational optimizations. Section 5 covers the relationship between well-known software optimizations and HLS, and accounts for which of these apply directly to HLS code. Finally, Section 7 includes performance results for a selection of kernels optimized using the transformations presented here, and we conclude in Section 8.

2 PIPELINE-ENABLING TRANSFORMATIONS

This category of transformations covers detecting and resolving issues that prevent pipelining of computations. When analyzing a basic block of a program, the HLS tool determines the dependency

graph between computations, and pipelines operations accordingly to achieve the target initiation interval. There are two classes of problems that hinder this process:

(1) **Interface contention** (intra-iteration): a hardware resource with limited ports is accessed multiple times in the same iteration of a loop. This could be a FIFO queue or RAM block that only allows a single read and write per cycle, or an interface to external memory, which only supports sending one request per cycle.
(2) **Loop-carried dependency** (inter-iteration): an iteration of a pipelined loop depends on a result produced by a previous iteration. If the latency of the operations producing this result is L, the minimum initiation interval of the pipeline will be L.

For each of the following transformations we will give examples of programs exhibiting properties that prevent them from being pipelined, and how the given transformation can resolve this.

All examples use C++ syntax, which allows objects (in particular `FIFO` buffers) and templating. We perform pipelining and unrolling using a pragma based syntax, 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).

2.1 Iteration space transposition

For multi-dimensional iteration spaces, loop-carried dependencies arising from accumulation can often be resolved by reordering the loops, adding additional buffers to store intermediate results. This also affects the memory access pattern, which can significantly impact memory performance. We will see these effects by applying the transformation to a concrete example.

Consider the matrix multiplication code in Listing 1a, computing $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C}$, with matrix dimensions N, M, and P. The inner loop $m \in M$ accumulates into a temporary register, which is written back to $\boldsymbol{C}$ at the end of each iteration $p \in P$. The multiplication of elements of $\boldsymbol{A}$ and $\boldsymbol{B}$ can be pipelined, but the addition on Line 8 requires the result of the addition in the previous iteration of the loop, resulting in an initiation interval of L_+, where L_+ is the latency of an addition for the given data type (for integers $L_{+,\texttt{int}} = 1$ cycle, and the loop can be pipelined without further modifications). To avoid this, we can transpose the iteration space, swapping the P-loop with the M-loop, with the following consequences:

- Rather than a single register, we now require an accumulation buffer of depth P and width 1.
- The loop-carried dependency is resolved, as we only update each location every P cycles.
- $\boldsymbol{A}$, $\boldsymbol{B}$, 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 P-loop and N-loop).

```
1 for (int n = 0; n < N; ++n)
2
3   for (int p = 0; p < P; ++p) {
4     auto acc = C[n][p];
5     #pragma PIPELINE
6     for (int m = 0; m < M; ++m)
7       // Loop-carried dependency
8       acc += A[n][m] * B[m][p];
9     C[n][p] = acc;
10  }
```

(a) Naive implementation of GEMM.

```
1 for (int n = 0; n < N; ++n) {
2   float acc[P]; // Uninitialized
3   for (int m = 0; m < M; ++m)
4     auto a = A[n][m];
5     #pragma PIPELINE
6     for (int p = 0; p < P; ++p) {
7       auto prev = (m == 0) ? C[n][p] : acc[p];
8       acc[p] = prev + a * B[m][p]; }
9   for (int p = 0; p < P; ++p)
10    C[n][p] = acc[p]; }
```

(b) Transposed iteration space.

Listing 1. Transposing the iteration space of GEMM removes the loop-carried dependency.

```
1 for (int i = 0; i < N; ++i) {
2   Vec<double, 3> acc;
3   Vec<double, 3> s0 = s[i];
4
5
6
7   #pragma PIPELINE
8   for (int j = 0; j < N; ++j)
9     acc += Force(s0, s[j], m[j]);
10
11  v[i] = v[i] + dt * acc;
12  s[i] = s0   + dt * v[i]; }
```

(a) N-body code with loop-carried dependency.

```
1 for (int i = 0; i < N / K; ++i) {
2   Vec<double, 3> acc[K];
3   Vec<double, 3> s0[K];
4   for (int k = 0; k < K; ++K)
5     s0[k] = s[i*K + k];
6   for (int j = 0; j < N; ++j)
7     #pragma PIPELINE
8     for (int k = 0; k < K; ++j)
9       acc[k] += Force(s0[k], s[j], m[j]);
10  for (int k = 0; k < K; ++K) {
11    v[i*K + k] += dt * acc;
12    s[i*K + k] += dt * v[i*K + k]; }}
```

(b) Strip-mine outer loop to interleave K accumulations.

Listing 2. Interleaving accumulations to eliminate the loop-carried dependency.

- Elements from $\boldsymbol{A}$ are only read once per iteration of the M-loop.

The modified code is shown in Listing 1b. We leave the accumulation buffer defined on Line 2 uninitialized, and implicitly reset it on Line 7, avoiding P extra cycles to reset.

2.2 Accumulation interleaving

For loop-carried dependencies on an accumulation variable where it is undesirable to transpose the full iteration phase, we can interleave accumulations to resolve the dependency by 1) partially folding an outer loop, or by 2) accumulating partial sums, then collapsing them in a separate module. We distinguish between the two cases below.

2.2.1 Nested accumulation interleaving. For accumulations done in a nested loop, we can resolve loop-carried dependencies due to accumulation by pipelining across multiple instances of the outer loop, using a buffer to store intermediate results.

Listing 2 shows this transformation on an N-body simulation code. We strip-mine the outer loop by a factor $K \geq L_{\text{acc}}$, where L_{acc} is the latency of the accumulation operation (in this case double addition), and absorb it into the inner loop. This allows $I = 1$ cycle by interleaving the accumulation of K instances of the outer loop in parallel, at the cost of a saturation and drain phase, and a buffer of depth K. This has the additional benefit of reducing memory bandwidth usage, as every external particle loaded is reused K times, cutting the total memory transferred by a factor of K.

2.2.2 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 depth K. On the last pass over the partial results, values will be streamed to the second phase (for more on streaming, see Section 3.3). The second phase is responsible for collapsing the partial results, and must be pipelined with an initiation interval less than or equal to the total number of iterations of the first phase to avoid pipeline stalls. For large input sizes, a single additional reduction unit thus suffices.

It is important to note that native accumulation units, if available, should be favored over either method due to higher resource efficiency (e.g., a single-adder floating point accumulator [9]).

```
1 Vec IterSolver(Vec state, int T) {
2
3
4   for (int t = 0; t < T; ++t) {
5     #pragma PIPELINE // I=L_Step
6     state = Step(state);
7   }
8
9   return state;
10 }
```

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

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

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

Listing 3. Pipeline across multiple inputs to maximize throughput despite loop-carried dependency.

2.3 Cross-input accumulation interleaving

For algorithms with loop-carried dependencies (e.g., due to a non-commutative reduction), we can still maintain high throughput by pipelining across multiple inputs to the algorithm. This procedure is similar to the interleaving done in Section 2.2, but requires altering the behavior of the program to accept multiple elements that can be interleaved.

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

2.4 Inlining

In order to successfully pipeline a code section, all function calls within must be absorbed into the pipeline. The simplest way to achieve this is *inlining*, which instantiates the called function as dedicated hardware as part of the pipeline. As a preprocessing step, this transformation is no different from the software equivalent and is handled transparently by most compilers when possible, but results in additional hardware being generated for every inlined function call. Inlining is thus desirable in all contexts that don't otherwise allow significant reuse of hardware resources. We implicitly assumed inlining in Listing 2, for example when assigning vectors on Line 5, when performing vector addition on Line 9, or when calling the `Force` function, also on Line 9. Both the member functions and the free function call must thus be inlinable, as well as pipelineable in the inlined context.

2.5 Cyclic buffering

When iterating over regular domains in a pipelined fashion, it is often sufficient to express buffering patterns using cyclic FIFO buffers. A common set of applications that adhere to this pattern are stencil applications such as partial differential equation solvers [19, 66, 70], image processing pipelines [29, 59], and convolutions in deep neural networks [7, 38], all of which are typically traversed using a *sliding window* buffer. These applications have been shown to be a good fit to

```
1 for (int n = 0; n < N; ++n) {
2   FIFO<float> acc(P);
3   for (int m = 0; m < M; ++m)
4     auto a = A[n][m];
5     #pragma PIPELINE
6     for (int p = 0; p < P; ++p) {
7       auto prev = (m == 0) ? C[n][p]
8                            : acc.Pop();
9       acc.Push(prev + a * B[m][p]); }
10  for (int p = 0; p < P; ++p)
11    C[n][p] = acc.Pop(); }
```

Listing 4. Accumulation array reduced to a FIFO buffer.

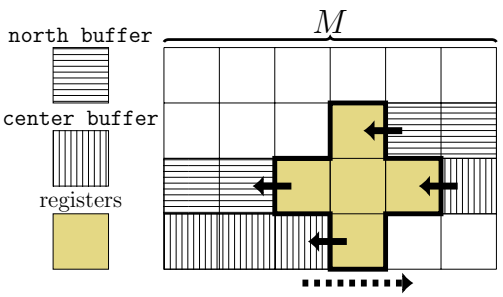

Fig. 1. Sliding window buffering for 2D stencil.

FPGA architectures [20, 21, 33, 50, 51, 76, 87], as FIFO buffers (also referred to as just "FIFOs") are natively supported, either as shift-registers or RAM blocks configured as FIFOs.

Opportunities for cyclic buffering often arise naturally from transforming programs to a pipelineable state. If we consider the transposed matrix multiplication code in Listing 1b, we notice that the read from acc on Line 7 and the write on Line 8 are both sequential, and cyclical with a period of P cycles. We could therefore substitute the array with a FIFO buffer of depth P, replace the read and write with FIFO queue operations Pop and Push, respectively. The resulting code is shown in Listing 4. The same transformation can be applied to the accumulation codes in Listings 2b and 3b.

Listing 5 shows two examples of applying cyclic buffering to simple sliding window stencil code, namely a 2D Jacobi stencil, which updates each point on a 2D grid to the average of its four neighbors: north, west, east and south. 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 processing two rows (illustrated in Figure 1), when the same value has been used as all four neighbors.

In Listing 5a we explicitly instantiate two FIFO line buffers on lines 1-2. We only read the south element from memory in each iteration of the stencil (Line 8), which we store in a FIFO buffer (Line 13). This element is then reused after M cycles, when it is used as the east value (Line 10), shifted in registers for two cycles until it is used as the west value (Line 14), after which it is pushed to the north buffer (Line 13), and reused for the last time after M cycles on Line 9. This scheme is illustrated in Figure 1. For more detail we refer to other works on the subject [15, 76].

Listing 5b includes an alternative pattern to express a sliding window buffering scheme in HLS. Rather than explicitly creating the FIFOs and registers required to propagate the values, a single array is used, which is shifted by one element every cycle using unrolling (Line 14). The compute elements access elements of this array directly, relying on the tool to infer the partitioning into FIFOs and registers (loop idiom recognition [5]) that we did explicitly in Listing 5a. While this method is less verbose, its implicit nature makes it more tool-dependent, as it can compile to inefficient hardware if the pattern is not recognized.

2.6 Pipelined loop flattening/coalescing

To minimize the number of cycles spent in saturating and draining pipelines (i.e., not streaming at full throughput), we can flatten nested loops. A pipelined loop has a saturation, streaming and drain phase, with the total number of cycles as given by Equation 1. Listing 6a shows a code with two nested loops, along with the total number of cycles to execute the program. The drain phase of the inner loop must be paid every iteration of the outer loop, or in terms of Equation 1, becomes the initiation interval of the outer loop. For large values of N_0 and N_1, the cycle count is just $I_1 N_0 N_1$,

```
1 FIFO<float> nb(M); // North buffer
2 FIFO<float> cb(M); // Center buffer
3 float west, center;
4 // ...initialization omitted...
5 for (int i = 0; i < N; ++i) // We assume
6   #pragma PIPELINE          // padding
7   for (int j = 0; j < M; ++j) {
8     auto south = in[i][j+1]; // Wavefront
9     auto north = nb.Pop(); // Read line
10    auto east  = cn.Pop(); // buffers
11    out[i][j] = 0.25*(north + west +
12                      south + east);
13    nb.Push(e); cb.Push(rd);
14    west = center; center = east; // Shift
15  }
```

(a) Buffering using streams and registers.

```
1 float b[2*M]; // Sliding window buffer
2
3 // ...initialization omitted...
4
5 for (int i = 0; i < N; ++i)
6   #pragma PIPELINE
7   for (int j = 0; j < M; ++j) {
8     auto rd = in[i+1][j]; // Wavefront
9     out[i][j] = 0.25*(b[M-1] + b[0] +
10                      b[M+1] + rd);
11    #pragma UNROLL
12    for (k = 0; k < 2*M-1; ++k)
13      b[k] = b[k+1]; // Shift the window
14    b[2*M-1] = rd; // Append wavefront
15  }
```

(b) Buffering using a sliding window buffer.

Listing 5. Two ways of reducing memory accesses in a stencil code from 4 to 1 using explicit buffering.

but for applications where N_1 is comparable to L_1, even if N_0 is large, this means that the drain of the inner pipeline can significantly impact performance. By *coalescing* the two loops into a single loop (shown in Listing 6b), the next iteration of the outer loop can be executed immediately after the previous finishes, leaving only a combined draining phase of $L_0 + L_1$ cycles at the end of the program.

To perform the transformation in Listing 6, we had to absorb any code present after each execution of the inner loop (Line 5 in Listing 6a) into the coalesced loop, adding a loop guard (Line 4 in Listing 6b). 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., by avoiding deep conditional logic in a pipeline, small inner loops can see a significant performance improvement by eliminating the draining phase. It should additionally be noted that the modulo used in the loop guard is amenable to strength reduction, i.e., for values of N_0 that are a power of two, where this operation reduces to a binary AND, or the more intrusive transformation of re-introducing individual loop-counters (an example of such code is given in Section 4.1) for each iteration variable present before the flattening, which will preserve the desired pipeline properties.

2.7 Pipelined loop fusion

We can exploit fine-grained dependencies between consecutive loops to fuse them into a single pipeline using loop guards. This transformation is closely related to loop fusion [36] from software optimization. For two consecutive loops with latencies/bounds $\{L_0, N_0\}$ and $\{L_1, N_1\}$, respectively, that are both pipelined with initiation interval I, the total runtime according to Equation 1 is $(L_0 + IN_0) + (L_1 + IN_1)$. If we can fuse the two loops without breaking dependencies between them, this can be reduced to $L_0 + L_1 + I \cdot \max(N_0, N_1)$.

Listing 7 shows an example of pipeline fusion applied to the GEMM code from Listing 9, fusing both the buffering of $\boldsymbol{A}$ and the write back to $\boldsymbol{C}$ into the inner loop, using loop guards and exploiting the fine-grained dependencies between the three loops. In addition to saving clock cycles, the code now constitutes a perfect loop nest, and can be coalesced similarly to Listing 6.

```
for (int i = 0; i < N1; ++i) {
  #pragma PIPELINE
  for (int j = 0; j < N0; ++i)
    // Code with latency L0
  // Code with latency L1
}
```

(a) Before coalescing: $\{L_1 + N_1 \cdot (L_0 + N_0)$ cycles$\}$

```
#pragma PIPELINE // Single loop
for (int i = 0; i < N0*N1; ++i) {
  // Code with latency L0
  if (i % N0 == 0)
    // Code with latency L1
}
```

(b) After coalescing: $\{L_0 + L_1 + N_0 N_1$ cycles$\}$.

Listing 6. Coalescing a perfect loop nest to avoid pipeline drains for the inner loop.

```
for (int nk = 0; nk < N / K; ++nk) {
  float acc[K][P];
  for (int m = 0; m < M; ++m) {
    float a_buffer[K];
    #pragma PIPELINE
    for (int p = 0; p < P; ++p)
      #pragma UNROLL
      for (int k = 0; k < K; ++k) {
        if (m == 0) // Buffer A
          a_buffer[k] = A[nk*K + k][m];
        auto prev = (m == 0) ? 0 : acc[k][p];
        auto res = prev + a_buffer[k]*B[m][p];
        if (m == M - 1) // Write back
          C[nk*K + k][p] = res;
        else
          acc[k][p] = res; }}}
```

Listing 7. Fusing the three pipelines in GEMM collapses them to a single pipeline with P iterations.

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

3 SCALABILITY TRANSFORMATIONS

Parallelism in HLS revolves around the *folding* of loop nests, which is achieved through *unrolling*. In Section 2.1 and 2.2, 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 an 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 program, as every expression in an unrolled statement is required to exist as hardware. We can thus move nested loop iterations from the sequential schedule into the parallel schedule. This corresponds to *folding* the sequential iteration space, as the number of cycles taken to execute the program are effectively reduced by the inverse of the unrolling factor.

3.1 Vectorization

We implement SIMD parallelism with HLS by *partially* unrolling loop nests in pipelined sections. This is the most straightforward way of folding our iteration space to obtain parallelism, as it can often be applied directly to the inner loop, without further reordering.

Listing 8 shows two functionally equivalent ways of vectorizing a loop over N elements by a factor of W: Listing 8a strip-mines a loop into chunks of the vector size and unrolls the chunk, while Listing 8b uses partial unrolling by specifying the unroll factor. OpenCL additionally includes built-in vector types, such as `float4`, `float8`, and `int16`, which similarly replicate registers and compute logic by the specified factor, but with less flexibility in choosing the vector type and length.

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

(a) Vectorization by strip-mining.

```
#pragma UNROLL W // By factor W
for (int i = 0; i < N; ++i)
  C[i] = A[i] * B[i];
```

(b) Vectorization by partial unrolling.

Listing 8. Two flavors of SIMD-style vectorization using loop unrolling.

The vectorization factor W [operand/cycle] is constrained by the available bandwidth B [Byte/s] to external memory according to

$$W_{\max} = \left\lfloor \frac{B}{fS} \right\rfloor, \tag{2}$$

where f [cycle/s] is the clock frequency of the vectorized logic and S [Byte/operand] is the operand size. While vectorization is a straightforward way of parallelization, it is bottlenecked by external memory bandwidth, and is thus not sufficient to achieve a scalable design. 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 Sections 3.2 and 3.3).

3.2 Replication

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

To demonstrate this process, we will look at how this can be done for the GEMM code from Listing 1. In Section 2.1, we saw that reordering loops allowed us to move reads from matrix $\boldsymbol{A}$ out of the inner loop, re-using the loaded value P times for P streamed columns of matrix $\boldsymbol{B}$. To obtain the final result, every column of $\boldsymbol{A}$ is combined with every row of $\boldsymbol{B}$. If we consider that every loaded value of $\boldsymbol{B}$ will contribute to *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. By buffering K elements of $\boldsymbol{A}$ prior to streaming the full $\boldsymbol{B}$-matrix, we can *fold* the outer loop over rows by a factor of K, using unrolling to multiply the amount of compute (as well as buffer space required for the partial sums), by a factor of K. The result of this transformation is shown in Listing 9.

3.3 Streaming dataflow

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

- Different functionality runs at different schedules. For example, issuing memory requests, performing memory requests, and servicing memory requests require different pipelines, state machines, and even clock rates.

```
1 for (int nk = 0; nk < N / K; ++nk) {
2   float acc[K][P]; // Is now 2D
3   // ...initialize acc from C...
4   for (int m = 0; m < M; ++m) {
5     float a_buffer[K];
6     #pragma PIPELINE
7     for (int k = 0; k < K; ++k) // Buffer A
8       a_buffer[k] = A[nk * K + k][m];
9     #pragma PIPELINE
10    for (int p = 0; p < P; ++p) // Stream B
11      #pragma UNROLL // K-fold replication
12      for (int k = 0; k < K; ++k)
13        acc[k][p] += a_buffer[k] * B[m][p];
14    // ...write back C...
15 } }
```

Listing 9. K-fold replication of compute units for GEMM. Saturation and drain phases marked in gray.

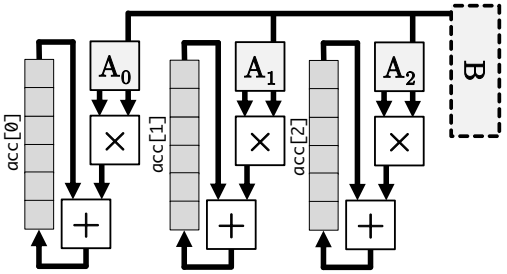

Fig. 2. Distribute elements streamed in to multiple buffered values.

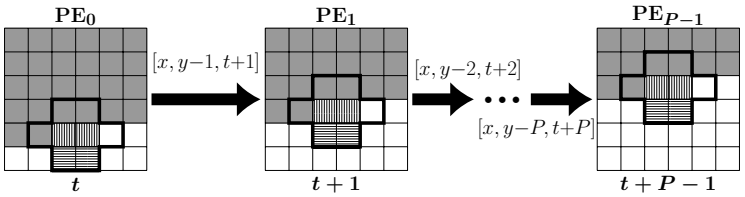

Fig. 3. Fold the time dimension of an iterative stencil by streaming across replicated processing elements.

- Modularity and testing: smaller components are easier to reuse, debug and verify.
- Synchronization, such as pipeline stalls, only need to propagate within the PE.
- Large *fanout*/*fanin* is challenging to translate into hardware (1-to-N/N-to-1 connections for large N). This can be resolved by partitioning components into smaller subparts, thus adding more pipeline stages to the design.
- The effort to schedule loops increases with the number of statements that need to be considered for the dependency and pipelining analysis. Scheduling logic in smaller chunks can be beneficial for both runtime and result.

To move data between PEs, *channels* with a handshaking mechanism are used. These data channels double as synchronization points, as they imply a consensus on the program state. In practice, channels are (with the exception of I/O) always FIFO interfaces, and support standard queue operations `Push`, `Pop`, and optionally `Empty`/`Full`, and `Size` operations. For higher depth requirements, channels can occupy the same resources as regular FIFO buffers (see Section 2.5).

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

```
1 void PE(FIFO<float> &in,
2         FIFO<float> &out) {
3   // ..initialization...
4   #pragma PIPELINE
5   while(streaming) {
6     auto prev = in.Pop(); // From t-1
7     // ...load values from buffers...
8     auto next = 0.25*(prev + /*...*/);
9     out.Push(next); }} // To t+1
```

(a) Processing element for a single timestep.

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

(b) Stream between processing elements.

Listing 10. Streaming between replicated processing elements to compute P stencil timesteps in parallel.

To see how streaming can be an important tool to express scalable hardware, we apply it in conjunction with replication (Section 3.2) to implement an iterative version of the stencil example from Listing 5. Unlike the GEMM code, the stencil code has no scalable source of parallelism in the spatial dimension. Instead, we can fold the outer time-loop to treat B_T timesteps in parallel, each computed by distinct PEs connected via channels [21, 62], as illustrated in Figure 3. We replace the memory interfaces to the PE with channels, such that the memory accesses on lines 8 and 11 become Pop and Push operations, respectively. The resulting code is shown in Listing 10a. We then use unrolling to make B_T replications of the PE, effectively increasing the throughput of the kernel by a factor of B_T, and consequently the runtime by folding the outermost loop by a factor of B_T, shown in Listing 5a. Such architectures ‘ares sometimes referred to as a *systolic arrays* [37, 44].

For platforms/HLS tools where large fanout is an issue, the principle of streaming between replicated PEs can also be applied to the GEMM example from Listing 9. We can move the K-fold unroll out of the PE code and replicate the entire PE instead, again replacing reads and writes with channel accesses. $\boldsymbol{B}$ is then streamed into the first PE, and passed downstream every cycle. $\boldsymbol{A}$ and $\boldsymbol{C}$ should no longer be accessed by every PE, but rather be handed downstream similar to $\boldsymbol{B}$, requiring a careful implementation of the drain and saturation phases, where the behavior of each PE will vary with its depth in the sequence.

3.4 Tiling

Loop tiling in HLS is commonly used to fold arbitrarily large problem sizes into chunks that fit into fast on-chip memory, in an *already pipelined program*. This contrasts loop tiling on CPU and GPU, where tiling is used to make a working program faster, rather than making a fast program work for large domains. Common for both paradigms is that they ultimately aim to meet fast memory constraints. As with vectorization and replication, tiling relies on strip-mining loops to gain useful properties by altering the iteration space.

As an example, consider the GEMM code from Listing 9. The buffer on Line 8 is required to pipeline the inner loop, but increases in size with P (columns of $\boldsymbol{B}$). Because of this, the code cannot support arbitrarily large matrices. Similar to the loop on Line 1, we can strip-mine the P-loop on Line 6 by a factor B_P and move it outside the M-loop, reducing the buffer size to $K \cdot B_P$, which is independent of the matrix dimensions. B_P can be as small as the latency L_+ of the addition used to accumulate without re-introducing a loop-carried dependency.

4 OTHER TRANSFORMATIONS

Once a design has been pipelined and scaled up to the desired degree of parallelism and hardware resource consumption, we can perform a number of additional optimizations to tune the design further. The transformations covered in this section are more situational and/or more amenable to compiler automation than the previous two classes, but are important to consider for maximizing pipeline, bandwidth and clock frequency results.

4.1 Condition flattening

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

Listing 11a shows an example of computing nested indices in a two dimensional iteration space, similar to how a loop is executed in software: the iterator of the inner loop is incremented until it exceeds the loop bounds, at which point the loop is terminated, and the iterator is incremented for the outer loop. This requires two integer additions and two comparisons to be executed before the final value of `j` is propagated to a register, where it will be read the following clock cycle to compute the next index. Because we know that `i` and `j` will always exceed their loop bounds in the final iteration, we can remove the additions from the critical path by bounds-checking the iterators before incrementing them, shown in Listing 11b. Note that these semantics differ from software loop at termination, as the iterator is not incremented to the out-of-bounds value before terminating.

```
int i = 0, j = 0;
for (int ij = 0; ij < i_max * j_max) {
  Foo(i, j); // Use indices in body
  if (++i == i_max) {
    i = 0;
    if (++j == j_max)
      j = 0;
  } }
```

(a) Two adds and two compares on critical path.

```
int i = 0, j = 0;
for (int ij = 0; ij < i_max * j_max) {
  Foo(i, j); // Use indices in body
  if (i == i_max - 1) {
    i = 0;
    if (j == j_max - 1) j = 0;
    else ++j;
  } else ++i; }
```

(b) One add and one compare on critical path.

Listing 11. Flattening conditional logic can significantly reduce the critical path to each branch.

4.2 Memory access extraction

By extracting accesses to external memory from the computational logic, we enable the two aspects to be pipelined and optimized separately. Accessing the same interface multiple times within the same pipelined section is a common cause for increased initiation interval due to interface contention, since the interface can only service a single request per cycle. In many cases, such as for independent reads, this is not an intrinsic memory bandwidth or latency constraint, but arises from the tool scheduling iterations according to program order. This can be relaxed when allowed by inter-iteration dependencies (this can in many cases be determined automatically, e.g., using polyhedral analysis [25]).

In Listing 12a, the same memory is accessed twice in the inner loop, preventing pipelining due to interface contention on A. By inserting buffered streams A_0 and A_1 of depth M, we can alternate between reading each section of A, allowing the HLS tool to infer bursts accesses to A of length M,

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

(a) Multiple accesses to A cause interface contention.

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

(b) Move memory accesses out of computational code.

```
1 void ReadA(const int A[2N],
2             FIFO<int> &A0,
3             FIFO<int> &A1) {
4   for (int i = 0; i < N/2; ++i) {
5     #pragma PIPELINE
6     for (int j = 0; j < M; ++i)
7       A0.Push(A[i][j]);
8     // Allows bursts of size M
9     #pragma PIPELINE
10    for (int j = 0; j < M; ++i)
11      A1.Push(A[N/2 + i][j]);
12  }
13 }
```

(c) Read bursts of A into buffered streams.

Listing 12. Separating memory reads from computational elements to allow burst access.

shown in Listing 12c. Since the schedules of memory and computational modules are independent, `ReadA` can run ahead of `PE` by up to $2M$ iterations, ensuring that memory is always read at the maximum bandwidth of the interface. From the point of view of the computational PE, both A_0 and A_1 are read in parallel, as shown on Line 6 in Listing 12b, hiding initialization time and inconsistent memory producers in the synchronization implied by the data streams.

A second use case for memory access extraction is to perform in-fast memory data layout transformations, such as transposing column-wise burst reads to a row-wise stream. Such a transformation could be applied after tiling the GEMM code in Listing 9, reading in a full tile of A and streaming it to the kernel in column-major order.

4.3 Memory oversubscription

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

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

4.4 Memory striping

When multiple memory banks with dedicated channels (e.g., multiple DRAM modules) are available, the bandwidth at which a single array is accessed can be increased by a factor corresponding the the number of available interfaces by striping it across the banks. This optimization is commonly known from RAID configurations.

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 asynchronous memory modules requesting memory from a mapped interface, then pushing to FIFO buffers that are read in parallel and combined by a third module, or vice versa for writing, exposing a single data stream to the computational kernel.

4.5 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. In particular, this can lead to significant improvements on architectures that are specialized for certain data types, such as FPGAs, which have traditionally been optimized for integer and fixed point computations. Because 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 be instantiated on the device.

While reduced energy consumption from using lower precision operations or integer operations over floating point operations is a benefit in general, other benefits of type demotion, namely area usage, bandwidth requirement and operational latency, vary greatly in effectiveness depending on the target architecture and the application bottleneck. The largest benefits are seen in the following three scenarios:

- In a compute bound scenario, the data type can be changed to a type that occupies less of *the same resources*. This in particular applies to FPGAs, that traditionally implement floating point operations using general purpose resources such as LUTs, FFs and DSPs.
- In a compute bound scenario, the data type can be moved to a type that is natively supported by the target architecture, such as 16 bit integers on Xilinx' 7 series DSP blocks [31], or single-precision floating point on Intel's Arria 10 and Stratix 10 devices [64].
- In a bandwidth bound scenario, performance can be improved by up to the same factor that the size of the data type can be reduced by.
- In a latency bound application, the data type can be reduced to a lower latency operation, such as from floating point, which requires multiple pipeline stages, to an integer type, which can typically be evaluated in a single cycle.

In the most extreme case, it has been shown that collapsing the data type of weights and activations in deep neural networks to binary [7, 14, 74] can provide sufficient speedup for inference that the loss of precision can me made up for with the increase in number of weights.

5 SOFTWARE TRANSFORMATIONS IN HLS

In addition to the transformations described in the sections above, we include a comprehensive overview of well-known software transformations and how they apply to HLS. We base this on the compiler transformations compiled by Bacon et al. [5]. The transformations are split into the following tables:

- Table 1 describes transformations that are essential components of the transformations presented in this paper, and notes how they relate.
- Table 2 lists transformations that apply to HLS in the same way that they apply to software.
- Additional transformations that we deemed to have little or no relevance to HLS, due to fundamental difference in software and hardware paradigms, are included in Appendix A.

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

CPU transformation	In HLS
Loop interchange [2, 36]	Used to resolve loop carried dependencies throughout Section 2.
Strip-mining [77]	Central component of many HLS transformations, including accumulation interleaving (Section 2.2), vectorization (Section 3.1), replication (Section 3.2), and tiling (Section 3.4).
Loop tiling [36, 40]	
Cycle shrinking [56]	
Loop distribution/fission [35, 36]	Useful for separating differently scheduled computations to allow pipelining (see Section 3.3).
Loop fusion [36, 79, 83]	Used for merging pipelines (see Section 2.7).
Loop unrolling [18]	Essential tool for scaling up performance by generating more computational hardware (Section 3.1 and 3.2).
Software pipelining [39]	Used by the HLS tool to schedule loop bodies according to the interdependencies of operations.
Loop coalescing/flattening [55]	Used to save pipeline drains in nested loops (Section 2.6).
Loop collapsing	
Reduction recognition	Prevent loop-carried dependencies in accumulation codes (Section 2.1 and 2.3).
Loop idiom recognition	Relevant for HLS backends, for example to recognize sliding-window buffers (Section 2.5) in Intel OpenCL [72].
Procedure inlining	Required to pipeline code sections with function calls (Section 2.4).
Procedure cloning	Every occurrence of a function is always specialized to all variables that can be statically inferred.
Loop unswitching [17]	Often the **opposite** is beneficial (see Section 2.6 and 2.7).
Loop peeling	Often the **opposite** is beneficial to allow coalescing (Section 2.6).
Graph partitioning	Streaming is central to hardware algorithms (Section 3.3).
SIMD transformations	Covered in Section 3.1.

Table 1. Software transformations that relate directly to the proposed HLS transformations.

directly (i.e., without further adaptation to HLS) to the imperative source code or intermediate representation before synthesizing for hardware (in particular transformations *loop-based strength reduction* through *scalar replacement* in Table 2). Despite not receiving much attention in this paper, 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 RELATED WORK

Much work has been done in optimizing C/C++/OpenCL HLS codes for FPGA, such as stencils [33, 75, 76, 78, 87], deep neural networks [69, 74, 84], matrix multiplication [16, 75], and Smith Waterman protein sequencing [60, 63]. These works optimize the respective applications using cyclic buffering, vectorization, replication, and streaming, which we describe as general transformations here.

Zohouri et al. [86] use the Rodinia benchmark to evaluate the performance on OpenCL codes on FPGA, employing optimizations such as SIMD vectorization, sliding-window buffering, accumulation interleaving, and compute unit replication across multiple kernels. We present a general description of a superset of these transformations, along with concrete code examples that show they are applied in practice. Kastner et al. [34] go through the implementation of many HLS codes in Vivado HLS, focusing on algorithmic optimizations for FPGA, and apply some of the transformations found here. Lloyd et al. [43] 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.

Software transformation	**Notes**
Loop-based strength reduction [8, 12, 68]	Benefits from eliminating code are larger, as this results in less generated hardware.
Induction variable elimination [1]	
Unreachable code elimination [1]	
Useless-code elimination [1]	
Dead-variable elimination [1]	
Common-subexpression elimination [1]	
Constant propagation, constant folding [1]	
Copy propagation, forwarding substitution [1]	
Reassociation	
Algebraic simplification, strength reduction	
Bounds reduction	
Redundant guard elimination	
Loop-invariant code motion (hoisting) [1]	Hoisting code from loops does not save hardware in itself, but can save memory operations.
Loop normalization	Used as an intermediate transformations.
Loop reversal [1]	Same arguments apply to HLS.
Array padding, array contraction	
Scalar expansion, scalar replacement	
Loop skewing [1]	Used in multi-dimensional wavefront codes.
Function memoization	Requires explicitly instantiating fast memory.
Tail recursion elimination	Eliminating dynamic recursion can enable a code to be implemented in hardware.
Regular array decomposition	Applies to partitioning of fast memory in addition to partitioning of external memory.
Message vectorization	We do not consider implications of distributed settings and message passing in this paper, but these optimizations should be implemented in dedicated message passing hardware when relevant.
Message coalescing	
Message aggregation	
Collective communication	
Message pipelining	
Guard introduction	
Redundant communication	

Table 2. Software transformations that have equivalent or similar meaning in HLS.

High-level, directive-based frameworks such as OpenMP and OpenACC have been proposed as alternative abstractions for generating FPGA kernels. Leow et al. [42] implement an FPGA code generator from OpenMP pragmas, primarily focusing on correctness in implementing a range of OpenMP pragmas. Lee et al. [41] present an OpenACC to OpenCL compiler, using Intel OpenCL as a backend. The authors implement vectorization, replication, pipelining and streaming by introducing new OpenACC clauses. As an alternative to OpenCL, Papakonstantinou et al. [53] generate HLS code for FPGA from directive-annotated CUDA code.

Mainstream HLS compilers automatically apply many of the transformations in Table 2 [3, 26, 27], but can also employ more advanced FPGA transformations. Intel OpenCL [72] performs memory access extraction into load store units (LSUs), does memory striping between DRAM banks, and detects and auto-resolves some cyclic buffering and accumulation patterns.

Polyhedral compilation is a popular framework for optimizing CPU and GPU programs [25], and has also been applied to HLS for FPGA for optimizing data reuse [57]. Such techniques may prove valuable in automating, e.g., the tiling transformation.

Implementing programs in domain specific languages (DSLs) can make it easier to detect and exploit opportunities for advanced transformations. Darkroom [29] generates optimized HDL for image processing codes, and the popular image processing framework Halide [59] has been extended to support FPGAs [58]. Additionally, Luzhou et al. [44] propose a framework for generating stencil codes for FPGAs. These frameworks rely on optimizations such as cyclic buffering, streaming and replication, which we cover here. Using DSLs to compile to structured HLS code can be a viable approach to automating a wide range of transformations, as proposed in the FROST [67] DSL framework.

7 EXPERIMENTS

To demonstrate the effects of the set of optimizing transformations proposed here, we apply them to a set of HLS kernels and report the resulting performance when targeting an FPGA platform. These kernels are written in C++ for the Xilinx Vivado HLS [81, 85] tool. We target the TUL KU115 [73] board, which hosts a Xilinx Kintex UltraScale XCKU115-2FLVB2104E FPGA and four 2400 MT/s DDR4 banks, although we only use two banks for these experiments. The chip hosts two smaller dies with limited interconnect between them, where each die is connected to two of the DDR4 pinouts. This multi-die design is used in all of Xilinx' larger UltraScale and UltraScale+ devices, and while it allows multiplying the amount of available logic resources ($2 \times 331{,}680$ LUTs and 2×2760 DSPs for the TUL KU115) with the number of connected dies, crossing between them is challenging for the routing process, which impedes the achievable clock rate and resource utilization for a monolithic kernel attempting to span the full chip. To interface with the host computer we use version 4.0 of the board firmware provided with the SDx 2017.2 [82] Development Environment, which provides memory and PCIe controllers on the device, and allows access to device memory and execution of the kernel through an OpenCL interface on the host side (this interface is compatible with kernels written in C++). For each example, we will describe the sequence of transformations applied, and give the resulting performance at each major stage.

7.1 Stencil code

As one of the most popular target applications for FPGAs in HPC, we will optimize a stencil code using the proposed transformations to optimize and scale up the design within the hardware constraints set by the FPGA platform. We implement the Jacobi 2D 4-point stencil from Listing 5. The experiments use single precision floating point types, and iterate over a 8192×8192 domain, and avoid memory conflicts by using a double-buffering scheme. We begin from a naive implementation with all explicit memory accesses, which has heavy interface contention on the input array, then perform the following optimization steps:

(1) To get rid of the interface contention, we implement cyclic buffers [§2.5] to store two rows of the domain, according to Listing 5a.
(2) We exploit spatial locality by introducing vectorization [§3.1], using memory extraction [§4.2], oversubscription [§4.3], and striping [§4.4] to stream reads and writes from and to two DRAM banks for consistent bandwidth.
(3) To exploit temporal locality we introduce the replication [§3.2] and streaming [§3.3] scheme shown in Listing 10. Furthermore, the domain is tiled [§3.4] to limit fast memory usage.

The effect of each stage above is quantified in Table 3. Enabling pipelining with cyclic buffers allows the kernel to throughput ~1 cell per cycle. Improving the memory performance to add

vectorization (using $W = 8$ operands/cycle for the kernel) exploits spatial locality through additional bandwidth usage. The replication and streaming step scales the design to only be limited by placement and routing due to high resource utilization.

	Perf. [GOp/s]	Speedup Relative	Speedup Cumulative
Naive	0.02	1×	–
Buffered [§2.5]	0.8	40×	–
Vectorized [§3.1, §4.2, §4.3, §4.4]	6.4	8×	320×
Replicated [§3.2, §3.3, §3.4]	227.8	36×	11,400×

Table 3. Performance progression of applying transformations to stencil kernel.

7.2 GEMM code

We implement a scalable GEMM kernel based on Listing 7. For experiments, we build for single precision floating point types, and benchmark for 8192×8192 matrices. The optimization stages performed are given below, starting from the naive code in Listing 1a:

(1) We transpose the iteration space [§2.1], removing the loop-carried dependency on the accumulation register, and extract the memory accesses [§4.2], vastly improving spatial locality. The buffering, streaming and writing phases [§2.7] are fused, allowing us to coalesce the three 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, and replicate computations by buffering multiple values of $\boldsymbol{A}$ and applying them all to the streamed in values of $\boldsymbol{B}$ in parallel [§3.2]. To avoid the issue of high fanout, we furthermore partition each buffered element of $\boldsymbol{A}$ into processing elements [§3.3], arranged in a systolic array architecture. Finally, the horizontal domain is tiled to accommodate arbitrarily large matrices with finite buffer space.

The result of each optimization stage is shown in Table 4. Allowing pipelining and regularizing the memory access pattern brings a dramatic improvement of 40×, throughputting ~1 cell per cycle. Vectorizing multiplies the performance by W, set to 8 in the benchmarked kernel. The replicated and streaming kernel is only limited by placement and routing due to high resource usage on the chip. Compute utilization is lower than for the stencil code, due to 1) different distribution of floating point multiplications to additions, and 2) more control logic overhead from the multiple data streams between the processing elements with respect to the computational logic.

	Perf. [GOp/s]	Speedup Relative	Speedup Cumulative
Naive	0.01	1×	–
Fused [§2.1, §2.6, §2.7, §4.2]	0.4	40×	–
Vectorized [§3.1]	3.2	8×	320×
Replicated [§3.2, §3.3, §3.4]	184.1	58×	18,410×

Table 4. Performance progression of applying transformations to a matrix multiplication kernel.

7.3 N-body code

Finally, we show the optimization process of an N-body based on the implementation in Listing 2. We use single precision floating point types and iterate over 16,128 bodies. Since Vivado HLS does not allow memory accesses of a width that is not a power of two, it was necessary to include memory extraction in the first stage. The steps taken were as follows:

(1) We extract the memory accesses [§4.2] and read wide 512-bit vectors [§4.3], converting these into the appropriate vector sizes (96 bit for velocities, 128 bit for combined position and mass).
(2) The loop-carried dependency on the accumulation if the acceleration is solved by applying nested accumulation interleaving [§2.2.1], pipelining across L different resident particles.
(3) To scale up the performance, we further multiply the number of resident particles, this time replicating [§3.2] compute through unrolling of an outer loop into K parallel processing element. Each element holds L resident particles, and interacting particles are streamed [§3.3] through them in a systolic array architecture.

The impact of steps 1-3 are shown in Table 5. The second stage gains a factor of 7× corresponding to the latency of the interleaved accumulation, then by a factor of 39× from replicated units across the chip. The memory bandwidth requirement is regulated by L. In fact, we can further reduce the bandwidth requirements by storing more resident particles on the chip, scaling up to the full fast memory usage of the FPGA. In this case, the accumulation interleaving transformation thus enables not just pipelining the compute, but also minimization of bandwidth consumption, and thus energy consumption due to I/O.

With these examples, we have demonstrated the effect of our transformations on a reconfigurable hardware platform, showing that we can scale up kernels until constrained by high resource utilization on the device. In particular, enabling pipelining, regularizing memory accesses and replicating were shown to be central components of scalable hardware architectures. Using these principles, we can continue to exploit new platforms as the hardware landscape evolves, adapting the transformation parameters to accommodate available resources.

	Perf.	**Speedup**	
	[GOp/s]	**Relative**	**Cumulative**
Initial [§4.2, §4.3]	0.9	1×	–
Interleaved [§2.2.1]	6.0	7×	–
Replicated [§3.2, §3.3]	231.9	39×	258×

Table 5. Performance progression of applying transformations to an N-body simulation kernel.

8 CONCLUSION

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

ACKNOWLEDGMENTS

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

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. Compilers, Principles, Techniques. *Addison Wesley* 7, 8 (1986), 9.

[2] John R. Allen and Ken Kennedy. 1984. Automatic Loop Interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*. ACM, New York, NY, USA, 233–246. https://doi.org/10.1145/502874.502897

[3] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. 2012. A Compiler and Runtime for Heterogeneous Computing. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 271–276. https://doi.org/10.1145/2228360.2228411

[4] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 89–108. https://doi.org/10.1145/1869459.1869469

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420. https://doi.org/10.1145/197405.197406

[6] David F. Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA programming for the masses. *Commun. ACM* 56, 4 (2013), 56–63.

[7] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *arXiv preprint arXiv:1802.09941* (2018).

[8] R. Bernstein. 1986. Multiplication by Integer Constants. *Softw. Pract. Exper.* 16, 7 (July 1986), 641–652. https://doi.org/10.1002/spe.4380160704

[9] M. R. Bodnar, J. R. Humphrey, P. F. Curt, J. P. Durbano, and D. W. Prather. 2006. Floating-Point Accumulation Circuit for Matrix Applications. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 303–304. https://doi.org/10.1109/FCCM.2006.41

[10] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. 2010. State-of-the-art in heterogeneous computing. *Scientific Programming* 18, 1 (2010), 1–33.

[11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[12] John Cocke and Ken Kennedy. 1977. An Algorithm for Reduction of Operator Strength. *Commun. ACM* 20, 11 (Nov. 1977), 850–856. https://doi.org/10.1145/359863.359888

[13] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[14] Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR* abs/1602.02830 (2016). arXiv:1602.02830 http://arxiv.org/abs/1602.02830

[15] Johannes de Fine Licht, Michaela Blott, and Torsten Hoefler. 2018. Designing scalable FPGA architectures using high-level synthesis. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 403–404.

[16] Erik H. D'Hollander. 2017. High-Level Synthesis Optimization for Blocked Floating-Point Matrix Multiplication. *SIGARCH Comput. Archit. News* 44, 4 (Jan. 2017), 74–79. https://doi.org/10.1145/3039902.3039916

[17] Thomas J. Watson IBM Research Center. Research Division, FE Allen, and J Cocke. 1971. *A catalogue of optimizing transformations*.

[18] Jack J. Dongarra and A. R. Hinds. 1979. Unrolling loops in Fortran. *Software: Practice and Experience* 9, 3 (1979), 219–226.

[19] C. A. Fletcher. 1988. *Computational Techniques for Fluid Dynamics 2*. Springer-Verlag New York, Inc., New York, NY, USA.

[20] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*. ACM, New York, NY, USA, 47–56. https://doi.org/10.1145/2145694.2145704

[21] Haohuan Fu and Robert G. Clapp. 2011. Eliminating the Memory Bottleneck: An FPGA-based Solution for 3D Reverse Time Migration. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/1950413.1950429

[22] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. 1982. A Second Opinion on Data Flow Machines and Languages. *Computer* 15, 2 (Feb. 1982), 58–69. https://doi.org/10.1109/MC.1982.1653942

[23] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. 2000. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*. IEEE Computer Society, Washington, DC, USA, 49–. http://dl.acm.org/citation.cfm?id=795659.

795916
[24] Mentor Graphics. 2004. Catapult High-Level Synthesis. (2004). https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls Accessed May 13, 2018.
[25] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.
[26] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. 2003. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *16th International Conference on VLSI Design, 2003. Proceedings.* 461–466. https://doi.org/10.1109/ICVD.2003.1183177
[27] Sumit Gupta, Rajesh Kumar Gupta, Nikil D. Dutt, and Alexandru Nicolau. 2004. Coordinated Parallelizing Compiler Optimizations and High-level Synthesis. *ACM Trans. Des. Autom. Electron. Syst.* 9, 4 (Oct. 2004), 441–470. https://doi.org/10.1145/1027084.1027087
[28] Jerker Hammarberg and Simin Nadjm-Tehrani. 2003. Development of Safety-Critical Reconfigurable Hardware with Esterel. *Electronic Notes in Theoretical Computer Science* 80 (2003), 219 – 234. https://doi.org/10.1016/S1571-0661(04)80820-X Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03).
[29] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4 (2014), 144–1.
[30] Mark Horowitz. 2014. Computing's Energy Problem (and What We Can Do About It). In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, 10–14.
[31] Xilinx Inc. 2018. User Guide 479 - 7 Series DSP48E1 Slice. (2018). https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf Accessed on May 10, 2018.
[32] Intel. 2017. Intel High-Level Synthesis (HLS) Compiler. (2017). https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html Accessed on May 11, 2018.
[33] Q. Jia and H. Zhou. 2016. Tuning Stencil codes in OpenCL for FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 249–256. https://doi.org/10.1109/ICCD.2016.7753287
[34] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel Programming for FPGAs. (May 2018).
[35] David J. Kuck. 1977. A Survey of Parallel Machine Organization and Programming. *ACM Comput. Surv.* 9, 1 (March 1977), 29–59. https://doi.org/10.1145/356683.356686
[36] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/567532.567555
[37] HT Kung and Charles E Leiserson. 1979. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings 1978*, Vol. 1. SIAM, 256–282.
[38] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. 2016. Deep learning on FPGAs: Past, present, and future. *arXiv preprint arXiv:1602.04283* (2016).
[39] M. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 318–328. https://doi.org/10.1145/53990.54022
[40] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. *SIGPLAN Not.* 26, 4 (April 1991), 63–74. https://doi.org/10.1145/106973.106981
[41] S. Lee, J. Kim, and J. S. Vetter. 2016. OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 544–554. https://doi.org/10.1109/IPDPS.2016.28
[42] Y. Y. Leow, C. y. Ng, and W. f. Wong. 2006. Generating hardware from OpenMP programs. In *2006 IEEE International Conference on Field Programmable Technology*. 73–80. https://doi.org/10.1109/FPT.2006.270297
[43] Taylor Lloyd, Artem Chikin, Erick Ochoa, Karim Ali, and Jose Nelson Amaral. 2017. A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers; Proceedings of*. VDE, 1–9.
[44] Wang Luzhou, Kentaro Sano, and Satoru Yamamoto. 2012. Domain-Specific Language and Compiler for Stencil Computation on FPGA-Based Systolic Computational-memory Array. In *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC'12)*. Springer-Verlag, Berlin, Heidelberg, 26–39. https://doi.org/10.1007/978-3-642-28365-9_3
[45] G. Martin and G. Smith. 2009. High-Level Synthesis: Past, Present, and Future. *IEEE Design Test of Computers* 26, 4 (July 2009), 18–25. https://doi.org/10.1109/MDT.2009.83
[46] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. 2012. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems* 16, 3 (2012), 31–51.

[47] Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. 2012. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 619–622.

[48] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 35, 10 (Oct. 2016), 1591–1604. https://doi.org/10.1109/TCAD.2015.2513673

[49] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 69–70.

[50] X. Niu, J. G. F. Coutinho, Y. Wang, and W. Luk. 2013. Dynamic Stencil: Effective exploitation of run-time resources in reconfigurable clusters. In *2013 International Conference on Field-Programmable Technology (FPT)*. 214–221. https://doi.org/10.1109/FPT.2013.6718356

[51] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell. 2012. Exploiting run-time reconfiguration in stencil computation. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 173–180. https://doi.org/10.1109/FPL.2012.6339257

[52] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D. Antonopoulos. 2011. Synthesis of platform architectures from OpenCL programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 186–193.

[53] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *2009 IEEE 7th Symposium on Application Specific Processors*. 35–42. https://doi.org/10.1109/SASP.2009.5226333

[54] C. Pilato and F. Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–4. https://doi.org/10.1109/FPL.2013.6645550

[55] Constantine D. Polychronopoulos. 1987. *Loop coalescing: A compiler transformation for parallel machines*. Technical Report. Illinois Univ., Urbana (USA).

[56] Constantine D. Polychronopoulos. 1988. Advanced loop optimizations for parallel computers. In *Supercomputing*. Springer, 255–277.

[57] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

[58] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 26.

[59] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[60] E. Rucci, C. García, G. Botella, A. D. Giusti, M. Naiouf, and M. Prieto-Matias. 2015. Smith-Waterman Protein Search with OpenCL on an FPGA. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 3. 208–213. https://doi.org/10.1109/Trustcom.2015.634

[61] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/1345206.1345220

[62] K. Sano, Y. Hatsuda, and S. Yamamoto. 2014. Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth. *IEEE Transactions on Parallel and Distributed Systems* 25, 3 (March 2014), 695–705. https://doi.org/10.1109/TPDS.2013.51

[63] Sean O. Settle. 2013. High-performance dynamic programming on FPGAs with OpenCL. In *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*. 1–6.

[64] Udayan Sinha. 2014. Enabling Impactful DSP Designs on FPGAs with Hardened Floating-Point Implementation. *Altera White Paper, WP-01227-1.0 (Aug. 2014)* (2014).

[65] Scott Sirowy and Alessandro Forin. 2008. Where's the beef? why FPGAs are so fast. *Microsoft Research, Microsoft Corp., Redmond, WA* 98052 (2008).

[66] Smith, Gordon D. 1985. *Numerical solution of partial differential equations: finite difference methods*. Oxford University Press.

[67] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. 2017. A Common Backend for Hardware Acceleration on FPGA. In *2017 IEEE International Conference on Computer Design (ICCD)*. 427–430. https://doi.org/10.1109/ICCD.2017.75

[68] Guy L. Steele. 1977. Arithmetic shifting considered harmful. *ACM SIGPLAN Notices* 12, 11 (1977), 61–69.

[69] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 16–25. https://doi.org/10.1145/2847263.2847276

[70] Taflove, Allen and Hagness, Susan C. 1995. Computational electrodynamics: The finite-difference time-domain method. *Norwood, 2nd Edition, MA: Artech House, 1995* (1995).

[71] David Barrie Thomas, Lee Howes, and Wayne Luk. 2009. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '09)*. ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/1508128.1508139

[72] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, Michael Kinsner John Freeman, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. 2012. From OpenCL to High-Performance Hardware on FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 531–534. https://doi.org/10.1109/FPL.2012.6339272

[73] TUL. 2017. TUL KU115 PCIe Accelerator Card. (2017). http://www.tul.com.tw/ProductsFPGA.html Accessed on May 13, 2018.

[74] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/3020078.3021744

[75] Anshuman Verma, Ahmed E. Helal, Konstantinos Krommydas, and Wu-Chun Feng. 2016. *Accelerating Workloads on FPGAs via OpenCL: A Case Study with OpenDwarfs*. Technical Report. Department of Computer Science, Virginia Polytechnic Institute & State University.

[76] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama. 2017. OpenCL-Based FPGA-Platform for Stencil Computation and Its Optimization Methodology. *IEEE Trans. Parallel Distrib. Syst.* 28, 5 (May 2017), 1390–1402. https://doi.org/10.1109/TPDS.2016.2614981

[77] Michael Weiss. 1991. Strip mining on SIMD architectures. In *Proceedings of the 5th international conference on Supercomputing*. ACM, 234–243.

[78] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. 2017. Energy Efficient Scientific Computing on FPGAs Using OpenCL. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 247–256. https://doi.org/10.1145/3020078.3021730

[79] Michael Joseph Wolfe. 1982. *Optimizing Supercompilers for Supercomputers*. Ph.D. Dissertation. Champaign, IL, USA.

[80] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[81] Xilinx. 2013. Vivado HLS. (2013). https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html Accessed on May 13, 2018.

[82] Xilinx. 2015. SDAccel Development Environment. (2015). http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html Accessed on May 11, 2018.

[83] A. P. Yershov. 1966. ALPHA – An Automatic Programming System of High Efficiency. *J. ACM* 13, 1 (Jan. 1966), 17–24. https://doi.org/10.1145/321312.321314

[84] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 25–34. https://doi.org/10.1145/3020078.3021698

[85] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*. Springer, 99–112.

[86] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. 2016. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 35, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014951

[87] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. ACM, New York, NY, USA, 153–162. https://doi.org/10.1145/3174243.3174248

A ADDITIONAL SOFTWARE TRANSFORMATIONS

Software transformation	**Applicability**
Loop spreading	No use-case found.
Parameter promotion	
Array statement scalarization	No built-in vector notation in C/C++/OpenCL.
Code colocation	Not relevant for HLS, as there are no function calls at runtime.
Displacement minimization	
Leaf procedure optimization	
Cross-call register allocation	
I/O format compilation	No I/O in HLS.
Supercompiling	Likely to be infeasible.
Short-circuiting	Meaningless in HLS, as all boolean logic exists in hardware regardless.
Loop pushing/embedding	Inlining completely is favored to allow pipelining.
Automatic decomposition and alignment	There is no (implicit) cache coherency protocol in hardware..
Scalar privatization	
Array privatization	
Cache alignment	
False sharing	
Procedure call parallelization	No forks in hardware.
Split	
VLIW transformations	No instruction sets in hardware.

Table A. Software transformations that have little or no relevance to HLS.