Towards Scalable and Efficient FPGA Stencil Accelerators

Work-In-Progress

Gaël Deest
IRISA/Université de Rennes 1
gdeest@irisa.fr

Nicolas Estibals
IRISA/Université de Rennes 1
niestiba@irisa.fr

Tomofumi Yuki
INRIA / LIP / ENS Lyon
tomofumi.yuki@inria.fr

Steven Derrien
IRISA/Université de Rennes 1
sderrien@irisa.fr

Sanjay Rajopadhye
Colorado State University
sanjay.rajopadhye@colostate.edu

ABSTRACT

In this paper we propose a design template for stencil computations targeting FPGA-based accelerators. The goal for our design is to provide scalable high throughput designs that can efficiently process iterative stencil programs with *large* size parameters, i.e., those whose data footprint is too large to fit on-chip. Our context is when we seek to use FPGAs as accelerators attached to CPUs. Minimizing the area is not our primary goal.

We propose a family of architectures based on hierarchical tiling, where the inner tiling is used to build coarse-grain data-path operators, increasing computational throughput, and the outer tiling is used to control the memory requirement, specifically data transfers to/from the accelerator. We present preliminary results for Jacobi-style stencils on 1D and 2D data, and are working on fully automating the flow.

1. INTRODUCTION

As we approach exa-scale computing, providing high performance within reasonable power budget is becoming more and more important. Application Specific Integrated Circuits (ASICs) are known to be much more energy efficient, at the cost of having fixed functionality dedicated to its use case. Reconfigurable architectures, such as Field Programmable Gate Arrays (FPGAs) provide a compromise between programmability and efficiency.

This has led to the development of architectures such as Xilinx Zynq,[1] a combination of general purpose processor and FPGA on a chip. Intel has recently announced Xeon+FPGA [17], which also aims at close integration of CPU and FPGA. However, reconfigurable architectures are much harder to "program," and require careful design to deliver high throughput.

In this paper, we address the problem of finding the best architecture for a class of programs called stencils. More specifically, we are interested in iterative stencil computations that updates some grid of data points over time. This class of programs can be found in image/video processing and in scientific simulations (such as solving PDEs). The importance of its applications, e.g., real-time processing for medical imaging, has led to many designs to accelerate this class of computations [5, 8, 13, 19, 25, 26, 29, 31, 33].

Even though computations that mainly manipulate data in floating-point is typically considered challenging for FPGAs, existing work have used fixed-point arithmetic to overcome this challenge [5, 8]. For some applications, the full dynamic range covered by floating-point encondings is not necessary, making them better suited for hardware accelerators. In particular, the work by Chen et al. [5] has presented a FPGA implementation of the 2D Finite Difference Time Domain method. Others have compared the performance of FPGAs with GPU implementations [8,33], reporting FPGAs to have better energy-delay products.

There are many different choices to be made when designing an architecture for stencil computations that have complex interplay. How to maximize compute-I/O ratio? How to exchange data between the host and the accelerator, especially when the accelerator is not large enough to fit the entire data? How to control the area-performance trade-off? How to distribute memory across banks to enable concurrent accesses? Existing approaches only address a subset of these challenges for stencil architectures.

Furthermore, the goal of existing approaches is often having the cheapest design that is "fast enough," a choice that makes sense in the context of embedded system design. For FPGAs as on-chip accelerators, our goal is to maximize the throughput of the given FPGA.

This paper presents a design methodology for stencil computations using High-Level Synthesis that considers all of these challenges. We borrow a well-known technique called *tiling* as an important building block of our design. Tiling is a reordering transformation, originally proposed for data locality, that groups operations as "tiles." In addition to improving compute-I/O ratio through better locality, it exposes coarser grained parallelism. Furthermore, the size of these blocks of operations are tunable parameters called "tile sizes" that give us convenient knobs to explore the trade-offs. Specifically, our design employs:

- a coarse-grain data-path operator with configurable amount of work,
- aggressive pipelining of the above operator to maximize throughput and frequency,

[1] `http://www.xilinx.com/products/silicon-devices/soc.html`

IMPACT 2016
Sixth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2016, Prague, Czech Republic
In conjunction with HiPEAC 2016.

`http://impact.gforge.inria.fr/impact2016`

- mapping of values to FIFOs for on-chip communication with minimal control overhead, and
- partitioning of the problem such that each partition fits the available memory on the accelerator.

This paper presents work-in-progress, explaining our strategy in detail, and providing preliminary results with simple examples that are semi-automatically generated.

The rest of the paper is organized as follows. We introduce the necessary background in Section 2, and describe our approach in Section 3. We present preliminary results for simple examples in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. BACKGROUND

In this section, we introduce stencil computations, tiling and related loop transformations, and high-level synthesis.

2.1 Stencil Computations

We define the stencil computations in our work. We are interested in a computation that updates a d-dimensional rectilinear array of size $N_1 \times \cdots \times N_d$ (d is typically 2 or 3).

Given the initial state of the array A^0, the successive states are computed as:

$$A^{t+1}(\vec{i}) = \texttt{update}\left(A^t\left(f_1(\vec{i})\right), \cdots, A^t\left(f_n(\vec{i})\right)\right)$$

where the functions $f_x(\vec{i}) = \vec{i} + \vec{c}_x, \vec{c}_x \in \mathbb{Z}^d$, i.e., some constant offset of $\vec{i}$, and the function `update` defines arbitrary operation to be performed using the input values of A^t. Note that the update uses the state of the array from *strictly the previous* iteration. There are more general definitions of stencil computations, but we use the one defined above, as it matches those used in prior work [9,29]. Our work can be extended to more general stencils.

A^0 is assumed to be an input, and $1 \leq t \leq T$, where T is the number of iterations, and is application dependent. Simple image filters may only apply one iteration ($T = 1$), but more complex filters are iteratively applied for larger values of T and/or until convergence.

2.2 Tiling

Tiling is an important loop transformation for improving data locality [22,37]. The idea is to partition the computation domain into regular-shaped (e.g., hyper-parallelepipedic) blocks, called *tiles*, so that the computation can be performed by executing those tiles as "atomic" computations, either sequentially or in parallel.

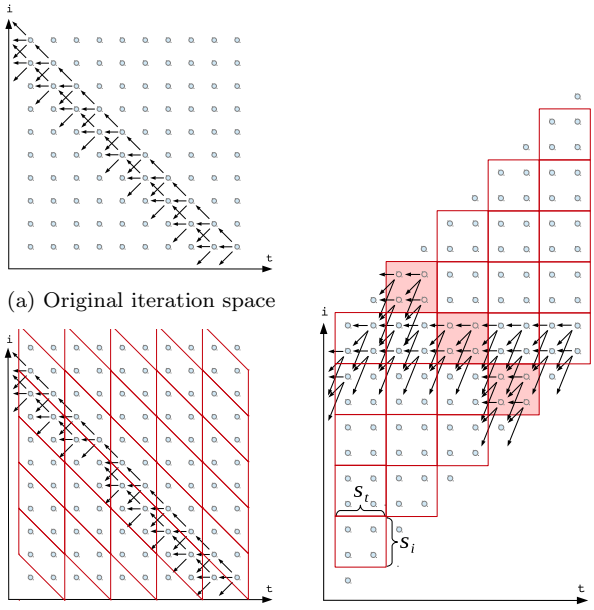
For example, the following loop nest can be readily split into rectangular tiles of shape `St`×`Si`:

```
for (t = 1; t <= T; t++)
  for (i = 1; i < N; i++)
    A[t,i] =  foo(A[t-1,i],
                  A[t-1,i-1]);
```

Listing 1: Original Loop

The result of the transformation is the following:

```
for (tt = 1; tt <= T; tt += St)
  for (ii = 1; ii < N; ii += Si)
    for (t = tt; t < min(T+1, tt+St); t++)
      for (i=ii; i < min(N, ii+Si); i++)
        A[t,i] =  foo(A[t-1,i],
                      A[t-1,i-1]);
```



(a) Original iteration space

(b) Tiled iteration space

(c) Tiled after skewing

Figure 1: Illustration of tiling for 1D Jacobi-style stencil. Only a subset of the dependences are drawn to avoid clutter. Figure 1b illustrates a possible tiling defined by two families of hyper-planes. The tiling hyper-planes are chosen such that there are no cyclic dependences among the tiles. Figure 1c is an equivalent tiling applied to a transformed iteration space such that the hyper-planes are along the canonic axes.

The outermost two loops iterate over tile origins in lexicographic order, while the inner loops visit all instances within a tile. An obvious benefit is that intermediate results that are only required by instances within the same tile are likely to be in cache.

2.3 Loop Skewing

The previous example could be directly tiled because the distance vectors $[1,0]$, $[1,1]$ are positive in each dimension, ensuring no mutual dependences between tiles. This is not the case in general. Take for example the 3-point Jacobi stencil in Figure 1a. The distance vectors are:

$$[1,1], [1,0], [1,-1]$$

This stencil cannot be tiled with rectangular tiles because adjacent tiles would be mutually dependent.

For such loops, we need to find non-rectangular tiles that partitions the iteration space without mutual dependences among tiles. This can be viewed as finding a set of hyper-planes (called tiling hyper-planes) such that the dependences do not cross the hyper-planes bi-directionally. Algorithms to find such hyper-planes have been developed [3], and stencils as defined in this paper are always tilable.

An alternative view to non-rectangular tiling is a pre-processing transformation called *skewing*. The iteration space can be transformed such that the hyper-planes along the canonic axes become valid tiling hyper-planes. For our example, applying the transformation $(t, i \rightarrow t, i+t)$ enables rectangular tiling as illustrated in Figure 1c.

Another important application of loop skewing is to ex-

tract parallelism. Recall the loop program in Listing 1. Its distance vectors are $[1,0]$ and $[0,1]$. Obviously, the inner loop is *not* parallel, as each iteration depends on the previous one. Once again, the solution is to skew the domain so that the inner loop carries no dependency. By applying the following transformation: $(t, i \rightarrow t+i, i)$, the dependence vectors become: $[1,0], [1,1]$. Each hyperplane of equation $t+i=d$ (in the original domain) hence represents a set of independent computations.

2.4 High-Level Synthesis

High-Level Synthesis [10] is a collection of techniques that produce hardware descriptions from higher level inputs, such as C, MatLab, and/or other languages. The primary objective is to increase the productivity of hardware designers by allowing (part of) the design to be written as algorithmic specifications, instead of logical circuits. In this work, we produce C programs that will be synthesized by the HLS tools, such as Xilinx Vivado[2] or Calypto Catapult[3]. Indeed, much of the motivation for our design approach is the increasing sophistication of these tools.

Current HLS tools are capable of synthesizing pipelined datapaths from loops by using a form of modulo scheduling (software pipelining). From a compiler writer's perspective, one may view FPGAs as VLIW processors with unlimited number of functional units.

3. APPROACH

In this section, we explain our approach in detail using a running example. The example we use is a simple Jacobi-style stencil over 1D data, which corresponds to the following loop nest:

```
for (t=1; t<=T; t++)
  for (i=1; i<N-1; i++)
     A[t][i] = update(A[t-1][i],
                      A[t-1][i-1],
                      A[t-1][i+1]);
```

The above computation can also be expressed as the following equation (or single assignment code):

$$A^t[i] = \text{update}\left(A^{t-1}[i], A^{t-1}[i-1], A^{t-1}[i+1]\right);$$

Although 1D data stencils are rather contrived, 2D iteration spaces are better suited for visualizing our approach. We discuss the implications to our approach when higher dimensional data grids are used at the end of this section.

3.1 High-Level Strategy

In terms of loop transformations, our approach can be succinctly described as two-levels of tiling.

- The inner/smaller tiling separates the iteration space into what we call *micro-tiles*. The computations in (i.e., the unrolled graph of) these tiles are input to the HLS tool, and it produces a deeply pipelined coarse grain *data-path operator* capable of executing the body of these tiles on the FPGA at a high throughput.
- The outer/larger tiling creates tiles of micro-tiles, which we name *macro-tiles*. A macro-tile is a set of independent micro-tiles in the parallel wave-front. Macro tiles represent a unit of computation that can fit on the accelerator.

Each level of the hierarchical tiling targets different levels of the memory hierarchy. For general purpose processors, it is used to target different levels of cache, TLBs, and disks [4]. In our approach, micro-tiles target on-chip memory, and macro-tiles target off-chip memory.

In contrast to approaches that use large numbers of small datapaths (or processing elements) to increase the throughput, the primary parallelism in our design comes from the pipelining of the datapath. Thus, we synthesize a *single*, large, processing element that is deeply pipelined.

3.2 Micro Tiling

The purpose of micro-tiling is to increase the granularity of the data-path. Most existing approaches and architectures [5, 19, 25] use one iteration of the loop as the granularity of computation, and pipeline across multiple iterations of the innermost loop. This limits the maximum achievable throughput, and we may want larger data-paths if we seek more performance.

More importantly, applying tiling increases the ratio of computations to I/O. With the tiled execution order, a subset of the data array is updated for multiple time iterations while the values are kept in registers. Using this temporal reuse allows more computations to be performed per data element fetched into registers. The ratio of communication to computation is controlled by the sizes of the micro-tiles s_x, and the goal is to use a large enough tile to fully utilize the available bandwidth.

Using the tiling visualized in Figure 1c, as an example, our "data-path" is now the four instances of the stencil update in a tile. There is some parallelism within a micro-tile, which is left to the High-Level Synthesis tools. For our example, the entire column within a micro-tile can be executed in parallel, and the synthesized data-path will make use of this parallelism.

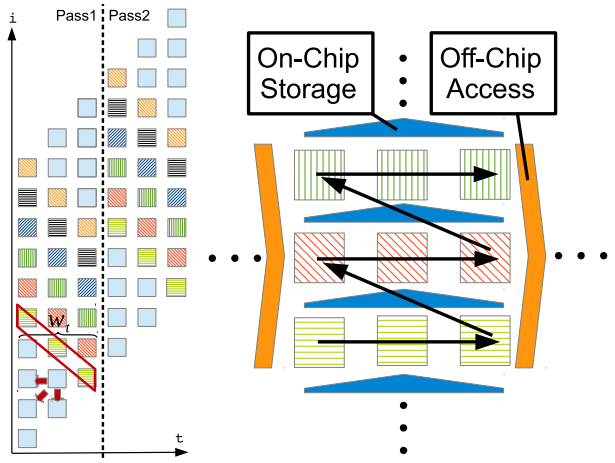
Another important property of tiling is that there exists a one-dimensional schedule among the tiles, i.e., there is $d-1$ dimensional, wave-front parallelism. For our example in Figure 1c, each diagonal strip of the tile (such as the shaded tiles) are independent of each other and can be executed in parallel. However, we *do not* exploit all this parallelism at the outer level. Because of the accelerator's resource constraints, the entire memory footprint of a single wave-front of micro-tiles is expected to be too large to fit on the accelerator. We therefore partition the computation into multiple "passes of micro-tile executions," and execute these passes sequentially on the accelerator. Similar strategies lead to energy-efficient parallelization of stencils on CPUs [39] and also on GPUs [32].

3.3 Macro Tiling

Within a pass, we define a macro-tile as the *set of micro-tiles in the parallel wave-front*, as illustrated in Figure 2. To simplify the control-flow, our design will be specialized for "completely full" macro-tiles, where all instances within any micro-tile in the macro-tile are valid iteration points. At the boundaries where some of the tiles (be they micro- or macro-) are empty or partial, we use the host processor to perform the computation in software.

A macro-tile is defined by $d+1$ size parameters $w_0 \ldots w_d$, and the macro-tile volume, V is the number of micro-tiles

[2] http://www.xilinx.com/products/design-tools/vivado.html
[3] http://calypto.com/en/products/catapult/overview



(a) Macro Tiling (b) Three macro-tiles on one data-path

Figure 2: Figure 2a shows the iteration space where each square corresponds to a micro-tile in Figure 1. The bold red arrows show inter-tile dependences. Each such tile is executed on a specialized, pipelined datapath generated by the HLS tool. Because of the pipelining, multiple micro-tiles need to be initiated on the datapath in successive clock cycles, and the sequence of these micro-tiles constitute a *macro-tile*, shown by different colors/shadings. Also note that only full tiles are executed on the accelerator.
Figure 2b is an collection of three consecutive instances of the macro-tile, where the micro-tiles are skewed to simplify the visualization. There are dependences across the macro-tiles where the values are communicated by on-chip storage (registers and/or BRAMs). The arrows show the order in which the micro-tiles are fed into the pipeline.

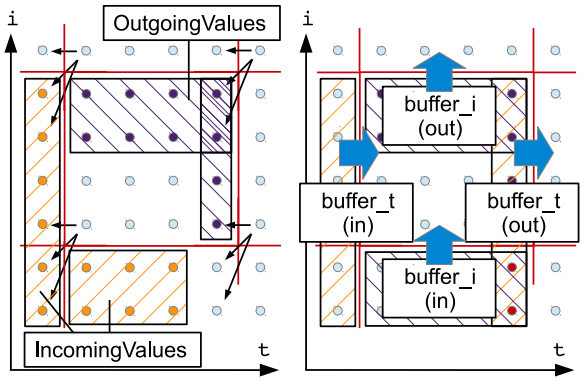
in a macro-tile, $V = \prod_x w_x$. We use the convention that w_0 is along the direction of the pass, and we will see later that it can be assumed to be 1 without loss of generality. If the pipeline depth of the datapath is δ, a necessary condition to achieve a sustained throughput of one micro-tile per clock cycle is $V \geq \delta$. If this condition is not met, some of the values used by the next wave-front of the micro-tiles (i.e., next horizontal strip of micro-tiles in Figure 2b) may not have exited the pipeline.

Within a macro-tile, the micro-tiles are executed following the wave-fronts, and lexicographically within the wave-fronts. Note that in higher dimensions, the parallel wave-fronts are d dimensional families of hyper-planes, and hence a macro-tile consists of a d dimensional "slice" of micro-tiles, since $w_0 = 1$.

3.4 Inter-Tile Communications

One of the important questions we need to address in our design is the communications between the tiles (both micro, and macro). Each tile depends on several preceding tiles, and the values must be communicated. In this section, we discuss the communication among the micro-tiles in detail. The communication among the macro-tiles are essentially the same as the micro-tiles, except that the communications across certain dimensions are performed using off-chip memory instead of on-chip memory.

Figure 3 illustrate the communications for our running example. The main requirement is that the incoming values



(a) Communication Pattern (b) Buffers

Figure 3: Figure 3a shows the communication pattern between micro-tiles for our running example for micro-tile size 4×4. Because stencil computations only consist of uniform dependences, only faces of a tile (of some constant thickness) are communicated. The top-right two values are used both by the tile to the immediate right and also by the tile to the upper-right diagonal. (The parallel wave-front—the pipelined tiles—is the diagonal in this figure.)
Figure 3b illustrates the buffers used for communication. Note that the two values in the right most column of the input `buffer_i` are copied to the output `buffer_t`. This is to satisfy the diagonal dependences across micro-tiles, so that all dependences are to the immediately previous wave-front of micro-tiles.

must be all read simultaneously in a cycle so that a micro-tile can enter the pipeline each cycle. Thus, 12 incoming values must be stored in a separate bank (or FIFO), and be all read into registers at the first stage of the pipeline.

Although the problem shares some similarities with existing work on bank mapping [7, 9, 36], we also require that all reads/writes by different micro-tiles, concurrently executed in the pipelined data-path, to not conflict with each other. This can be achieved by ensuring that the read and the write of a value is separated by a *constant distance*, and that the distance is larger than the pipeline depth. In another words, we want a FIFO for each incoming values (thus 12 FIFOs for our example).

One source of complication are values that are used by multiple later tiles. In our running example, the upper-right values are used both by the right and the upper-right tile. Having two tiles reading a write obviously violates the concept of FIFO.

3.4.1 Mapping Values to Buffers

We have developed a systematic approach to map produced values to FIFOs, or buffers, such that efficient communications across tiles are achieved. The proposed approach utilizes ideas from a technique for memory allocation [38] targeting distributed memory architectures, which has strong links to communications across tiles, called *canonic multi projection*. The key ideas are to:

- Store values into *multiple* buffers, each of them being a *canonic projection* of the iteration space. This avoids complicated access patterns to the buffers, simplifying the control.

- Some of the values are redundantly stored, and are appropriately copied from one buffer to another. This can be viewed as breaking diagonal dependences across tiles into a composition of dependences along canonical axes.

For an n-dimensional iteration space, we store the values into n separate buffers, where each element of these buffers becomes a FIFO. The buffers (B^x) are represented as a combination of a projection p^x, and a thickness factor τ^x. The projection p^x is the projection along the x-th dimension, and the thickness is computed based on the dependences: the magnitude of the largest dependence vector in that dimension. In addition, the projection is applied to an expanded region of the tiles to account for values produced by iterations of another tile, but used in the current tile. The way that a tile is expanded is different for each projection, and the manner in which the thickness and expansion factors are computed is presented in Algorithm 1. The reads from the resulting buffer are the lexicographically minimum values of the expanded tile, and the writes are the lexicographically maximum values.

Input:
I_x : set of dependences to carried by dimension x as distance vectors
δ^{x-1} : expansion factor of the previous dimension (0 vector when $x = 0$)
Output:
δ^x : expansion factor for p^x
τ^x : thickness factor for p^x
begin
  $\delta^x = \delta^{x-1}$
  `// No expansion in the outer dimensions`
  $\delta^x_{x-1} = 0$
  foreach $v \in I_x$ **do**
    $\delta^x = \max(v, \delta^x)$ `// element-wise max`
  end
  $\tau^x = \delta^x_x$
end

Algorithm 1: Find expansion and thickness factor for p^x. Note that for Jacobi stencils, all dependences are carried by the outermost dimension. Thus the `foreach` loop is only active for the first dimension. However, for Gauss-Seidel patterns, the loop may also be active in inner dimensions.

For our running example, the set of distance vectors are $[1, 0]$, $[1, 1]$, and $[1, 2]$. We have two dimensions to project and the algorithms gives the following:

- $\delta^t = [1, 2]$, $\tau^t = 1$
- $\delta^i = [0, 2]$, $\tau^i = 2$

The buffer B^t is computed by the projection along t, p^t, with expansion factor $[1, 2]$ and thickness of 1. This means that we take the 4×4 tile, expand in the lexicographically negative direction by $[1, 2]$, and then take the resulting projection along the t axis.[4] Since the thickness is 1, we obtain the 1×6 line as the B^t.

[4]Note that expanding along the projected dimension does not affect the size of the buffer, but is relevant to which values the incoming buffers correspond to.

Similarly for B^i, we expand by $[0, 2]$, and then take the projection along i. With thickness factor 2, we obtain the 4×2 strip as the B^i.

3.4.2 Propagating Values Across Buffers

Now the values are mapped to buffers, we discuss the propagation of values across buffers. The propagation is rather straightforward; when an input buffer and an output buffer overlap, the values are copied from the input to the output.

One important property is that the propagation is always from a buffer $B^{x'}$ to another buffer B^x where $x < x'$ (i.e., from an inner dimension buffer to an outer dimension one).

Observe that in Algorithm 1, for each projection, the expansion factors for the outer dimensions are set to 0, and that the thickness is equal to the expansion factor. This means that all input buffers are isolated from other input buffers, and hence they do not overlap. In other words, an input buffer B^x corresponds to the values at the expanded region along x. Thus, for another buffer $B^{x'}$, where $x < x'$, the tile is not extended along the x dimension, and thus it is not possible to overlap. Using the same argument, input buffers from projections in inner dimensions can overlap with output buffers in outer dimensions, which is the property stated above.

One exception to the above is when the micro-tile sizes are so small such that the dependences crosses multiple tiles. For our example in Figure 3, using $s_i = 1$ would cause the long dependence to "skip over" a tile. In such cases, copies may happen between input and output buffer based on the same projection. In the example, values from `buffer_i` may remain in `buffer_i` at output (although moved to another element).

3.5 Handling Higher Dimensional Data

Our design is applicable to computations over higher dimensional data. Recall that we used 1D data stencil to simplify the visualization. Although some dynamic programming algorithms, such as Smith-Waterman, can be viewed as stencil computations with 2D iteration space, most stencil applications operate over higher dimensional data.

The most complicated part of our design is the communication between tiles, which has been presented for arbitrary dimensions in the above discussion. For stencils over d dimensional data, each buffer is a d dimensional facet of constant thickness. Values produced by a tile may take at most d "hops" from the buffer corresponding to the innermost dimension to the outermost. Assuming that the tile sizes are large enough so that tiles only depend on immediate neighboring tiles (including diagonal neighbors), the longest dependence (diagonal in all directions) is apart by d in the parallel wave-fronts, and thus d hops is exactly what is needed.

One important remark is that for a stencil over d dimensional data ($d + 1$ dimensional iteration space), the parallel wave-front spans a d dimensional space. Thus, for higher dimensional stencils, the macro-tile sizes need to be large enough so that there are enough micro-tiles in the d dimensional hyper-plane to fill the pipeline.

3.6 Selecting Tile Sizes

Our design have two sets of configurable parameters, the micro-tile sizes s_x, and the macro-tile sizes w_x. The best tile sizes depend on many factors, including the desired perfor-

mance, size of the FPGA, size of the iteration space, stencil patterns, and so on.

In this paper, we do not address the problem of finding the best tile sizes. However, changing the tile size for different dimensions have different implications on performance, and area requirements. In this section, we discuss these effects, as well as constraints on the tile sizes posed by our design.

3.6.1 Macro Tile Shape

Before we discuss the tile sizes, we first explain a subtlety related to the choice of tiling hyper-planes for macro-tiles. After tiling for micro-tiles, we apply the second level of tiling, and the tiling hyper-planes are chosen such that the subset of them correspond to the parallel wave-front. For instance, for our example in Figure 2a, the tiling hyper-planes are i and $t+i$, where $t+i$ is the parallel wave-front. This can also be viewed as skewing the micro-tiles by $(t, i \rightarrow t, i+t)$ such that the horizontal dimension becomes parallel (as in Figure 2b). This ensures that the amount of parallelism is constant at all phases of the macro-tile.

It is also possible to use the hyper-planes t and $t+i$, i.e., skew by $(t, i \rightarrow t+i, i)$, in our example, which makes the vertical dimension parallel. In general, $d-1$ dimensions of the macro-tile correspond to the parallel dimensions, and the remaining dimension is for the wave-front time. Let us denote the wave-front time dimension as *time* and the parallel dimensions as $p1, \cdots, pd$.

The choice of the hyper-planes determine which buffers are communicated on-chip. The buffer B^{time} is the buffer that will remain on-chip across macro-tiles, except at the iteration space boundaries. All other buffers are communicated off-chip at the boundaries of each macro-tile. Thus, we want to select the hyper-planes such that the wave-front time crosses the dimension with most communication volume. One approximation, ignoring the micro-tile sizes, is to select the dimension with the largest thickness factor.

3.6.2 Macro Tile Sizes

One important constraint on the macro-tile sizes is that the number of tiles in a wave-front must be at least equal to the pipeline depth of the micro-tiles. Otherwise, the micro-tiles in the next wave-front cannot start immediately, causing the micro-tile data-path to be under-utilized.

The number of tiles in the wave-front is $w_{p1} \times \cdots \times w_{pd}$. This implies that increasing the macro-tile size in the *time* dimension does not help in satisfying the depth constraint.

The macro-tile size in the *time* dimension does not have significant influence to the throughput of our design. Extending the tile size in *time* increases the amount of off-chip memory access per macro-tile, but the compute-I/O ratio remains the same. If the off-chip memory access can be streamed, then the macro-tiling for this dimension is not necessary. This is why we claimed that the w_0 (which is w_{time}) can be set to 1 without loss of generality in Section 3.3. Our example in Figure 2 has the macro-tile size in the vertical dimension (w_i) set to one for the same reason.

However, the tile size in the *time* dimension can be interpreted as the granularity of synchronization. When there are other tasks running concurrently on the host that requires synchronization, and/or when off-chip accesses is not streamed, the tile size in the *time* dimension can be used to control the frequency of synchronization.

Increasing the other tile sizes in the wave-front parallel dimensions increases the amount of computation per micro-tile without increasing I/O requirements. Thus, we would like to maximize the macro-tiles in these dimensions, up to the limitations by resource (on-chip memory) and the problem size (having macro-tiles to be too large creates many partial tiles to be executed on the host).

3.6.3 Micro Tile Sizes

In this paper, we primarily treat Jacobi-style stencils that have d dimensional parallelism over the data grids. Therefore, increasing the micro-tile size in these dimensions increase the parallelism within a micro-tile. Although the low-level details of the design is left to the architecture, we can expect the pipeline depth to be only proportional to the diameter of the dependency graph of a micro-tile, i.e., the micro-tile size along the wave-front time.

Increasing the tile sizes in all dimensions improves locality. In the parallel dimensions, spatial locality is improved, where the time dimension improves temporal locality. In stencil computations, values are typically reused in all dimension, and hence we would like to increase the tile sizes in all directions as well.

For instance, if we consider our example in Figure 3a, having a thin vertical tile of 1×8 creates a tile that performs 8 updates with total of 9 inputs. Tiling with a rectangular tile of 2×4 creates a tile with the same amount of updates, but only uses 8 inputs.

In general, inputs to a micro-tile is $O(n^d)$ where the computation is $O(n^{d+1})$. Thus, increasing the tile sizes will improve the ratio of computation to I/O. The best tile size depends on the characteristic of the stencil. For example, the thickness of the buffers in each dimensions, which is computed based on the dependences, influence the additional inputs required by extending the tile in each dimension.

4. STATUS AND PRELIMINARY RESULTS

We now present the current state of our tools and some preliminary results. Our target platform is a programmable System on Chip, which is a combination of general purpose processors and FPGAs. The Xilinx Zynq is an example of such an architecture currently available.

The full flow that we envision is as follows:

1. We take as an input the stencil specification in `C` or possibly other languages.
2. We use polyhedral techniques to analyze the dependences and to find the tiling hyper-planes.
3. We employ the algorithm presented in this paper to generate the necessary communications. Skewing of the micro-tiles for parallel execution does not even require polyhedral machinery [18, 23].
4. We are working on automating the generation of the host code to compute boundary tiles, and to manage the off-chip communications.

At the time of writing we do not have a fully automated flow. In this section, we present preliminary results focused on the programmable logic of our target platform. We have developed a simple code generator to generate the declarations of buffers and the communication. This code was manually integrated into a larger piece of code that has auxiliary functions that define interfaces to off-chip communications.

We generated the accelerator codes for 1D, 3-point, stencil and for 2D, 5-point, stencil where the `update` function simply takes the average of the input values. We used Vivado HLS 2014.4 to synthesize and to perform place and route for the Zynq-7000.[5]

4.1 Metrics for Evaluation

Since our design is not fully integrated with the software code that will be executing the partial tiles, we use two metrics computed from the tile sizes, number of operations per update, and the achieved frequency. These metrics are defined for the *steady-state*, i.e., when a sequence of macro-tiles are sweeping the interior region of the iteration space.

The first metric is the throughput captured as the number of Giga OPerations per second (GOP/s). Since we use floating-point arithmetic in this paper, this directly translates to GFLOP/s. At stead-state, a micro-tile is initiated every cycle. Thus, the number of operations performed per cycle is the product of the micro-tile volume, and the number of operations per stencil update ($\#\texttt{OPs} = 4$ for 1D and 6 for 2D.) The resulting formula is as follows:

$$\texttt{GOP/s} = \prod_{i=0}^{d} s_i \times [\#\texttt{OPs}] \times [\texttt{frequency(MHz)}] \times 10^{-6}$$

The other metric we use is the off-chip bandwidth required during the steady-state execution. This is the bandwidth required to keep feeding a micro-tile each cycle, once the sweep has started. This does not capture the bandwidth requirement for the full execution, since at the steady-sate, some of the values are already in memory from the previous macro-tile, allowing for reuse.

We recall Figure 2a to illustrate an example. After executing a macro-tile depicted as a parallelogram in the figure, starting the next macro-tile (wave-front of micro-tiles directly above) only require one face of the micro-tile at the left boundary. This ignores the additional off-chip accesses required to transfer the inputs to the macro-tile at the boundary.

Each macro-tile needs d faces of inputs to start in the steady-state. The face that is communicated on-chip from the previous macro-tile depends on the skewing used at the macro-tile level. Let us assume that the first dimension is the dimension that corresponds to the *time* as defined in Section 3.6.1, which is the dimension that stays on-chip. Then the macro-tile faces $\texttt{MacroFace}_1$ through $\texttt{MacroFace}_d$ are the faces that needs to be read from off-chip. We assume that the latency is hidden by a form of double buffering (or streaming.) Thus, we need enough bandwidth to support transferring the inputs needed for a macro-tile, while executing another macro-tile. Since a micro-tile is fed to the pipeline each cycle, we would like to have the next macro-tile ready after V cycles where V is the number of micro-tiles in a macro-tile. Thus, the per-cycle bandwidth requirement is the sum of macro-faces read from off-chip, divided by the number of micro-tiles in a macro-tile. This gives us the following formula for computing the stead-state bandwidth requirement:

$$\texttt{GB/s} = \sum_{i=1}^{d} \texttt{MacroFace}_i \times \frac{1}{\prod_{i=0}^{d} w_i} \times [\texttt{frequency(MHz)}] \times 10^{-6}$$

[5]XC7Z100-FFG900-1, largest Zynq available (the complete board is around $1000.)

We ignore the output bandwidth, since the volume of data transferred is the same as the input, and the interconnect in our target platform supports fully asymmetric transfers.

The micro-tile faces for 1D Jacobi are:

- $\texttt{MicroFace}_0 = s_1 + 2$
- $\texttt{MicroFace}_1 = s_2 \times 2$

and those for 2D Jacobi are:

- $\texttt{MicroFace}_0 = (s_1 + 2) \times (s_2 + 2)$
- $\texttt{MicroFace}_1 = s_0 \times 2 \times (s_2 + 2)$
- $\texttt{MicroFace}_2 = s_0 \times s_1 \times 2$

The macro-tile faces are the size of the corresponding plane multiplied by the size of the micro-tile face. For example, $\texttt{MacroFace}_0 = w_1 \times w_2 \times \texttt{MicroFace}_0$ for the 2D case.

As we mentioned in Section 3.6.2, the value of w_0 does not influence the compute-I/O ratio. Note that increases in inputs values with larger w_0 are cancelled[6] by the division by the macro-tile volume.

4.2 Preliminary Results

The above metrics applied to our design are summarized in Table 1.

As expected, increasing the micro-tile sizes gives higher throughput. Although larger micro-tiles decrease the achievable frequency, the increased granularity provide much higher throughput. We think this frequency drop is mostly due to the actual implementation and not to the architecture; thus, we expect it to be less important in the future implementation. Even with this degradation, the 2D case with 4^3 tile size performs 8 times more iterations per cycle compared to the one with 2^3 tiling, but is only 33% slower.

On Zynq architectures, the main memory can be accessed at the rate of 3.8 GB/s. Note that the bandwidth requirement shown in the table is the upper bound using the smallest possible macro-tile sizes. Since only a small fraction of the BRAMs are used, it is possible to increase the macro-tile sizes to reduce the bandwidth requirement, with negligible cost to other resources. Moreover, other platforms and next-generation FPGA SoCs support much higher bandwidth.

We believe that the result we present here is promising, but is still too preliminary to directly compare with other designs. We are working on fully integrating the accelerator to the host, so that we have a more complete picture of the performance that include the computation of the tiles at the boundaries on the host.

4.3 Implications of Our Design

As we have discussed in Section 3.6.2 there are many factors that influence the efficiency of our design. In this section, we discuss some of the factors that have strong influences on our design.

We have used floating-point arithmetic in this paper. This is a more challenging case for FPGAs, and fixed-point implementations are likely to deliver more performance. We do not present a full design space exploration in this paper.

If fixed-point arithmetic can be used or not depends on the application and its intended use case. We are likely to get higher throughput with fixed-point, since it has much lower

[6]All $\texttt{MacroFace}_i, i > 0$ includes a multiplication by w_0.

Table 1: Evaluation with the metrics described in Section 4.1 for the synthesis results of our design for 1D and 2D Jacobi stencils on Zynq-7000. Increasing the wave-front size will proportionally decrease the bandwidth requirement with negligible cost until the capacity of the BRAM is reached. When the number of micro-tiles in a wave-front exceeds the BRAM capacity (512 floats in this board), the number of BRAMs required is doubled.

Micro Tile Size	Macro Tile Size	Pipeline Depth	Frequency [MHz]	Area [slice/BRAM/DSP]	Synthesis + P&R time [s]	Steady-State Throughput [GFLOP/s]	Steady-State Bandwidth [GB/s]
2×2	1×62	61	210	2001(2%)/ 4(0%)/ 40(1%)	354	3.4	0.055
2×4	1×62	61	180	3483(5%)/ 5(0%)/ 80(3%)	428	5.8	0.047
4×2	1×118	117	180	6820(9%)/ 6(0%)/ 80(3%)	703	5.8	0.049
4×4	1×118	117	180	6050(8%)/ 7(0%)/160(7%)	843	11.5	0.049
8×8	1×230	229	110	24258(34%)/ 13(1%)/640(31%)	2005	28.2	0.031
$2 \times 2 \times 2$	$1 \times 11 \times 10$	100	150	9175(13%)/ 16(2%)/112(5%)	1055	7.2	1.4
$3 \times 3 \times 3$	$1 \times 13 \times 12$	148	125	15206(21%)/ 32(4%)/378(18%)	2664	20.3	1.9
$4 \times 4 \times 4$	$1 \times 15 \times 14$	196	100	31169(44%)/ 52(6%)/896(44%)	12308	38.4	2.2

cost in terms of hardware resources compared to floating-points. However, we are also likely to used less than 32 bits to encode the values (otherwise, fixed-point arithmetic itself is not beneficial,) which also influences the compute-I/O ratio. We will be exploring these trade-offs in the future.

Our design also constraints the problem sizes that are likely to be beneficial. For example, the 1D case with the largest micro-tile size assumes 230 instances of 8×8 micro-tiles in the wave-front, i.e., 230 independent instances of 8×8 micro-tiles. This requires the problem size to be at lest 1840×1840—even more due to skewing and partial tiles.

The evaluation in this paper assumes large enough problem sizes so that the large numbers of such macro-tiles can be executed. The applicability of our technique does depend on the problem sizes expected for specific use cases.

Although the 1D case may look over-constraining, this is due to the limited parallelism in 1D date stencils. For Jacobi 2D, the pipeline depth wit the largest micro-tile ($4 \times 4 \times 4$) is 196. Since we need a 2D plane of micro-tiles to keep the pipeline busy for 196 cycles, a possible macro-tile size is $1 \times 15 \times 14$. Since we are looking at a two-dimensional plane of tiles, there are smaller implications on each dimension.

We cannot fully characterize the specific instances of stencil applications where our technique is applicable without a full implementation. We do not expect our design to be efficient for image processing filters that perform little or no iterations over the given image.

5. RELATED WORK

In this section, we discuss three bodies of related work: (i) tiling in compilers, (ii) loop transformations for HLS, and (iii) stencil computations on FPGAs.

5.1 Tiling in Compilers

Tiling is a classical loop transformation used in various contexts [22,37]. Since maximizing the parallelism in a program often does not equal to best performance, tiling is also used for extracting coarse-grained parallelism. The state-of-the-art automatic parallelizers (e.g., [3]) also use tiling based parallelization as its core strategy.

Tiling has been known to be one of the most important optimizations for many classes of programs, including stencil computations. Due to the importance of stencil computations, and its regular dependence pattern, there are different variations of tiling than the one used in this work [16, 24]. One of the main issues addressed in these work is the problem of load imbalance. Parallelization of standard tiling combined with loop skewing have different degrees of parallelism at each parallel wave-front (which is visible in Figure 2a). Allowing concurrent start by more complex tiling and/or by performing redundant computations is one of the main goals of the other variants.

Another body of work around stencil computations have developed domain specific languages and compilers specialized for stencils [6,21,35]. These work also employ variations of tiling combined with additional optimizations.

At the high-level, parallelism we use is identical to those utilized by automatic parallelizers based on tiling. However, targeting FPGAs poses different challenges at the lower levels, such as pipelining and on-chip communications through FIFOs.

We have not explored other variations of tiling in this work. We expect the general strategy of tiles as a coarse-grained, pipelined, operator to hold for other shapes of tiling. The communications across tiles must be revisited for more complex shapes of tiling.

5.2 Loop Transformations for HLS

Recently, with the increased maturity of HLS tools, source-to-source transformations for HLS is gaining interest. In particular, a number of transformations based on polyhedral techniques have been proposed [1, 2, 7, 9, 13, 28, 31, 36].

The work by Alias et al. [1, 2], `PolyOpt/HLS` by Pouchet et al. [31], and the `DEFACTO` [13] tool use tiling to manage off-chip memory accesses. Tiling is used to partition the computation so that each block of computation fits the on-chip memory. The key difference to our work is that we use hierarchical tiling, and pipeline the micro-tiles. To the best of our knowledge these tools do not consider making a larger "operator" through tiling to be later pipelined, which is the primary purpose of our micro-tiles.

Work on memory bank mapping have some similarities to the buffer mapping in our work [7, 36]. These work find mapping from array elements to memory banks to enable concurrent access to the values required by an iteration of the innermost parallel loop. This allows iterations of the innermost loop to be pipelined with no delays between iterations.

Although the objective—concurrent access to allow the pipeline to be fed every cycle—is the same, we address the

problem at the tile level instead of individual iterations. Our mapping cannot be described as affine projections [36] or lattices [7]. Moreover, the redundant storage through multi-projection is essential to view the buffers as FIFO, which simplifies control.

5.3 Stencil Computations on FPGAs

There have been several ad-hoc implementations of FPGA hardware accelerators targeting stencil-like algorithms, such as optical flow estimation [25] and/or FDTD [5, 19]. All these approaches focused on exposing parallelism within the innermost loop, while exploiting data reuse within a stencil sweep through pipeline. None of these earlier attempts did try to take advantage of data reuse along the time dimension, although several work acknowledge that off-chip memory traffic ends up being the main performance bottleneck when the whole data-set does not fit in FPGA on-chip memory.

Some of the FPGA implementations do perform a form of tiling [15, 29], although they do not use the name tiling. These work use a variant of tiling (known as overlapped tiling [24]) that redundantly computes the boundaries of the tiles to avoid frequent communications with the neighboring tiles. However, overlapped tiling over 2D data significantly increases both the amount of redundant computation and extra I/O.

Luzhou et al. [26] also use a form of tiling to implement stencil computations on FPGA. However, the tiling applied in this work is only for the spatial dimensions. Their approach is only applicable when the problem size is small and fits on-chip memory.

There is also another body of work that focus on a different class of stencil applications where the number of time iterations are small [9, 20, 33]. Many image processing applications only make one pass over the image, but multiple of these filters may be composed to form an image processing pipeline. Our work is not directly applicable to this type of stencils, because our design relies on the temporal reuse present in time-iterated stencils to manage off-chip accesses.

5.4 Systolic Architectures

It is also important to note that tiling is not new to hardware design. Earlier work on systolic architectures dealt with similar problems of partitioning the iteration space since it is not realistic to have systolic cells (processing elements) for each iteration point. The macro-tiling in our work, resembles the Locally Parallel Globally Sequential partitioning [27, 30].

The use of hierarchical tiling/partitioning has also been proposed in the context of custom hardware accelerators. Eckhardt and Merker [14] proposed a two-level partitioning, called co-partitioning, to efficiently support multiple levels of memory hierarchy when synthesizing systolic arrays. Similar approach was used in the PICO NPA tool [11, 34] to limit the on-chip memory usage.

Tiling/partitioning has also been used as a mean to enable aggressive datapath pipelining. For example, in systolic arrays, Derrien et al. [12] have shown that the combined use of LSGP (Locally Sequential Globally Parallel) partitioning and fine grain gate-level retiming could significantly improve the datapath throughput using pipelining. Alias et al. [2] proposed a similar approach in the context of High-Level-Synthesis, where they choose the tile sizes to match the latency of a datapath built out of heavily pipelined operators.

Our work draws ideas from many earlier work in this domain. We use the macro-tiles to manage bandwidth usage, and aggressively pipeline the micro-tiles for throughput. Our micro-tiles may be viewed as a FloPoCo operator used in the work by Alias et al. [2] that have configurable size parameters (micro-tile sizes.)

In contrast to the earlier systolic architectures, we pipeline the micro-tiles to increase the compute power. The approach in PICO [11, 34] uses $n-1$ dimensional grid of processors (given n dimensional iteration space) to parallelize a tile where a physical Processing Element computes one loop iteration. If more throughput is desired, the number of physical PEs are increased.

Our work may be casted to the PICO framework as follows. We first tile the input loop nest to create micro-tiles. These micro-tiles are then fully unrolled to form a "single" loop iteration that will lead to a much larger PE, and we always use a single PE. Since we only have a single PE, mapping to the virtual PEs is simple; we use the parallel wavefront of the micro-tiles. However, we skew the micro-tiles before applying the outer level of tiling so that the parallel wave-front is along the tiling hyper-planes. Increase in parallelism comes in two forms through the increase in micro-tile sizes: (i) the parallelism within the micro-tile, and (ii) the depth of the pipeline. In other words, we are exploring a different design space where we synthesize a single datapath with high degrees of parallelism, where other work use a number of smaller PEs with a lower degree of parallelism in each PE.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a design methodology for stencil computations, targeting programmable hardware. Our strategy has its roots in the automatic synthesis of systolic architectures, which can be viewed as one of the earlier work on High-Level Synthesis.

There have been much recent work on synthesizing efficient architectures for stencil computations using HLS. These work only address some partial picture of the problem, and are lacking connections to the long history of developments in the compiler community on stencil computations. This work attempts to provide a solution that incorporates techniques from systolic architectures, loop transformations, and HLS to achieve a scalable and customizable design template for stencil computations.

We are currently in the process of developing an automated design flow that will generate codes for both the processing system and programmable logic. This paper mostly presents our current status on programmable logic side.

The stencil computations targeted in this paper exhibit local communications by nature. However, localization of communications was extensively studied during the days of systolic architectures. Locality of communication is an essential property in distributed computing in general, not limited to hardware accelerators. We believe that our work can also be generalized beyond stencil computations to handle more complex dependences.

Acknowledgement

The work was funded in part by AFOSR, under grant FA9550-13-1-0064 and by DoE under grant DE-SC0014495.

7. REFERENCES

[1] C. Alias, A. Darte, and A. Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *Proceedings of the 2013 Conference on Design, Automation and Test in Europe*, DATE '13, pages 575–580, 2013.

[2] C. Alias, B. Pasca, and A. Plesco. FPGA-specific synthesis of loop-nests with pipelined computational cores. *Microprocessors and Microsystems*, 36(8):606–619, Nov. 2012.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, 2008.

[4] L. Carter, J. Ferrante, and S. Hummel. Hierarchical tiling for improved superscalar performance. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 239–245, Apr. 1995.

[5] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *Proceedings of the 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pages 213–222, 2004.

[6] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 25th International Parallel Distributed Processing Symposium*, IPDPS '11, pages 676–687, May 2011.

[7] A. Cilardo and L. Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Transactions on Architecture and Code Optimization*, 11(4):45:1–45:25, Jan. 2015.

[8] J. Cong, M. Huang, and Y. Zou. Accelerating fluid registration algorithm on multi-FPGA platforms. In *Proceedings of the 21st International Conference on Field Programmable Logic and Applications*, FPL '11, pages 50–57, Sept 2011.

[9] J. Cong, P. Li, B. Xiao, and P. Zhang. An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 77:1–77:6, 2014.

[10] P. Coussy and A. Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*, volume 1. Springer, 2010.

[11] A. Darte, R. Schreiber, B. Rau, and F. Vivien. A constructive solution to the juggling problem in processor array synthesis. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, IPDPS '00, pages 815–821, may 2000.

[12] S. Derrien, S. Rajopadhye, and S. S. Kolay. Combined instruction and loop parallelism in array synthesis for fpgas. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 165–170, 2001.

[13] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler. Automatic mapping of c to FPGAs with the DEFACTO compilation and synthesis system. *Microprocessors and Microsystems*, 29(2–3):51–62, 2005. Special Issue on FPGA Tools and Techniques.

[14] U. Eckhardt and R. Merker. Hierarchical algorithm partitioning at system level for an improved utilization of memory structures. *IEEE Transaction on CAD of Integrated Circuits and Systems*, 18(1):14–24, 1999.

[15] H. Fu and R. G. Clapp. Eliminating the memory bottleneck: An FPGA-based solution for 3d reverse time migration. In *Proceedings of the 19th International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 65–74, 2011.

[16] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, 2014.

[17] P. K. Gupta. Xeon+FPGA: Platform for the data center, June 2015. Talk at the Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL '15).

[18] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Dyntile: Parametric tiled loop generation for parallel execution on multicore processors. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, IPDPS '10, pages 1–12, 2010.

[19] C. He, W. Zhao, and M. Lu. Time domain numerical simulation for transient waves on reconfigurable coprocessor platform. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '05, pages 127–136, April 2005.

[20] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. In *Proceedings of the 41st International Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '14, 2014.

[21] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, 2013.

[22] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, 1988.

[23] D. Kim and S. Rajopadhye. Efficient tiled loop generation: D-tiling. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, LCPC '09, 2009.

[24] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th Conference*

on Programming Language Design and Implementation, PLDI '07, pages 235–244, 2007.

[25] M. Kunz, A. Ostrowski, and P. Zipf. An FPGA-optimized architecture of horn and schunck optical flow algorithm for real-time applications. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications*, FPL '14, pages 1–4, Sept 2014.

[26] W. Luzhou, K. Sano, and S. Yamamoto. Domain-specific language and compiler for stencil computation on FPGA-based systolic computational-memory array. In *Proceedings of the 8th International Symposium on Applied Reconfigurable Computing*, ARC '12, pages 26–39, 2012.

[27] D. Moldovan and J. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35(1):1–12, Jan 1986.

[28] A. Morvan, S. Derrien, and P. Quinton. Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(3):339–352, 2013.

[29] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza. A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 52:1–52:6, 2013.

[30] J. Navarro, J. Llaberia, and M. Valero. Partitioning: An essential step in mapping algorithms into systolic array processors. *Computer*, 20(7):77–89, July 1987.

[31] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the 21st International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, 2013.

[32] W. Ranasinghe. Reducing off-chip memory accesses of wavefront parallel programs in graphics processing units. Master's thesis, Colorado State University, Computer Science Department, 2014.

[33] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *Proceedings of the 12th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '14, Oct 2014.

[34] R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. In *Proceedings of the 12th International Conference on Application-Specific Systems, Architectures, and Processors*, ASAP '00, pages 113–124, Jul 2000.

[35] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, 2011.

[36] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 12:1–12:8, 2013.

[37] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the 12th Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, 1991.

[38] T. Yuki and S. Rajopadhye. Canonic multi-projection: Memory allocation for distributed memory parallelization. Technical report, CS-11-106, Colorado State University, 2011.

[39] Y. Zou and S. Rajopadhye. Automatic energy efficient parallelization of uniform dependence computations. In *Proceedings of the 29th International Conference on Supercomputing*, ICS '15, pages 373–382, 2015.