One size does not fit all: Implementation trade-offs for iterative stencil computations on FPGAs

Gaël Deest, Tomofumi Yuki, Sanjay Rajopadhye, Steven Derrien

One Size Does Not Fit All: Implementation Trade-Offs for Iterative Stencil Computations on FPGAs

Gaël Deest, Univ. Rennes 1 / IRISA
Tomofumi Yuki, INRIA / IRISA
Sanjay Rajopadhye, Colorado State University
Steven Derrien, Univ. Rennes 1 / IRISA

***Abstract*— Iterative stencils are kernels in various application domains such as numerical simulations and medical imaging, that merit FPGA acceleration. The best architecture depends on many factors such as the target platform, off-chip memory bandwidth, problem size, and performance requirements.**

We generate a family of FPGA stencil accelerators targeting emerging System on Chip platforms, (e.g., Xilinx Zynq or Intel SoC). Our designs come with design knobs to explore trade-offs. We also propose performance models to hone in on the most interesting design points, and show how they accurately lead to optimal designs. The optimal choice depends on problem sizes and performance goals.

I. Introduction

Iterative stencil computations arise in many application domains, ranging from medical imaging to numerical simulation. Since they are computationally demanding, a large body of work addressed the problem of parallelizing and optimizing stencils for multi-cores, GPUs, and FPGAs.

Earlier attempts targeting FPGAs showed that the performance of such accelerators is a complex interplay between the raw FPGA computing power, the amount of on-chip memory, and the performance of the external memory system [1]–[8]. They also illustrate different application requirements. For example, in the context of embedded vision, designers often seek the cheapest design achieving real-time performance constraints (e.g., 4K@60fps). In an exascale context, they may want to maximize performance (measured in ops-per-second) for a given FPGA board, while maintaining power dissipation to a minimum. Therefore, we explore a family of design options that can accommodate a large set of constraints, by exposing trade-offs between computing power, bandwidth requirements, and FPGA resource usage.

We focus on system-level issues. Our aim is not to provide hand-optimized FPGA implementations. We have developed a code generator that produces HLS-optimized C/C++ descriptions of accelerator instances, leaving low-level decisions to the HLS back-end.Our designs build upon the *tiling* transformation, that we use to balance on-chip memory cost and off-chip bandwidth. The design space we explore can be characterized by the following design knobs.

- Unrolling Factor: Our accelerators are based on a heavily pipelined datapath derived from HLS tools. The amount of fine-grain parallelism in the datapath is configured through unrolling of the innermost loops.
- Tile Shape: The choice of tile shapes, characterized by the sizes of a tile in each dimension (possibly not tiling some of them), enables trade-offs between on-chip memory usage and bandwidth consumption.
- We propose simple analytical models for both performance and area cost to guide the design space exploration.

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

The rest of the paper is organized as follows. We introduce the necessary background in Section II, and describe our accelerator architecture in Section III. We discuss the performance models we use to guide the exploration in Section IV. Section V describes the result of our extensive design space exploration on a Zynq board. We discuss related work in Section VI and conclude in Section VII.

II. Background

We introduce the notion of stencil computations, our target platform, and the core of our approach: *loop tiling*.

A. Stencil Computations

Iterative stencil kernels are computations that iteratively update 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 are of the form $f_x(\vec{i}) = \vec{i}+\vec{c}_x, \vec{c}_x \in \mathbb{Z}^d$, i.e., add some constant offset to $\vec{i}$, and the function `update` defines some 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.

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. This leads to large workloads that are excellent candidates for hardware acceleration.

B. Programming FPGA Accelerators

In both the embedded and high performance computing domains, FPGA platforms are evolving toward hybrid hardware/software platforms, where the FPGA fabric is tightly coupled to the processor. Examples of such platforms include the Xilinx Zynq and the Intel SoC, in which the FPGA has access to the last level cache of the ARM processor.

For such platforms, memory bandwidth and/or latency constraints must be considered early in the design flow. Because of the complexity of the platform, it is often difficult for designers to determine the best architecture, and

manual Design Space Exploration is not realistic. This makes generative approaches (i.e., automatic generation of domain specific hardware accelerators) attractive. However, the design space offered by such approaches is extremely large, and the use of performance models to drive the exploration stage becomes necessary.

Since our work focus on system-level issues, we utilize HLS to derive our hardware accelerator. Specifically, the state-of-the-art HLS tools provide the ability to (i) synthesize pipelined datapaths, (ii) expose complex multi-banked on-chip memory organization in the code, and (iii) abstract away complex I/O interfaces through the use of a simple API.

C. Parallelizing Stencils on FPGAs

We outline the challenges of implementing stencil on FPGAs using a running example. This example consists of a (simplified) Jacobi-style stencil over 2D data, and can be described using the loop nest shown below.

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

Figure 1 depicts the same kernel, visualizing the iteration domain, and the inter-iteration dependencies.

Given that the two innermost loops are completely parallel, accelerating the algorithm on FPGA may seem trivial. However, stencils can operate on large spatial grids that are too large to fit in on-chip memory (for example, when computing the optical flow of a 4K image). It is therefore necessary to store this data-set in external memory, making the naive parallelization severely I/O bound.

It is thus necessary to partition the computations into atomic blocks that are amenable to efficient hardware accelerations on an FPGA. We achieve this decomposition using loop tiling that provides two important benefits:

- It improves memory access locality, and eases the use of on-chip (scratchpad) memory.
- It exposes tile-level *wavefront* parallelism, which is well suited for parallelization on multi-core (through OpenMP) or GPUs. We use this parallelism for overlapping computation with accesses to memory.

Tiling consists of partitioning the iteration domain into regular-shaped (e.g., hyper-parallelepipedic) blocks, called *tiles*, so that the computation can be performed by executing those tiles "atomically," either sequentially or in parallel.

In contrast to multi-core/GPU targets, we are not interested in executing tiles in parallel. Our goal is to accelerate the execution of a single tile by (i) utilizing fine grain parallelism through unrolling and pipelining, (ii) operating exclusively from on-chip memory, and (iii) using burst transfers to fill/flush this on-chip memory with appropriate data from external memory without causing I/O stalls.

D. Loop Tiling Strategies

Since tiling changes the execution order of operations in the loop, not every loop tiling is legal. In particular a legal tiling must guarantee the absence of cyclic dependencies at the tile level. Figure 1 illustrates this constraint in the case of our running example, where it can be observed that an orthogonal (i.e., rectangular) tiling leads to cyclic dependencies between adjacent tiles.

For stencils, these cycles are caused by the dependencies (data-flow) flowing both forward and backward on x and y axes, and this can be resolved using additional transformations. The idea consists of "skewing" the iteration space along one (or both) axis to guarantee unidirectional data-flow.

Skewing only along the x or y axis enables tiling along that dimension; we refer to this strategy as **partial oblique tiling**. We illustrate this approach in Figure 2, where it is applied along y and t axis. This results in incomplete tiles, which we must pad with fake iterations to reuse the same accelerator as full tiles. Another consequence is that the on-chip memory requirement is not fully controlled, since one of the tile dimensions (along x) is the full domain size. This approach may not be suited to stencils with large spatial domain, on target platforms with limited on-chip memory.

Another strategy is to use skewing along both x and y axes. Then, it becomes possible to tile along all three dimensions x, y, and t, and we will refer to this strategy as **fully oblique tiling** (see Figure 3). The use of combined skewing results in significantly more incomplete tiles compared to partial oblique tiling. On the other hand, it is now possible to have

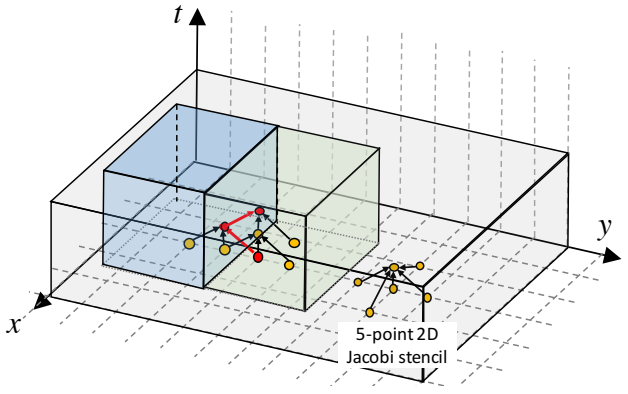


Fig. 1: Example of illegal loop tiling shapes for 2D stencils, with a cyclic tile-level dependency along y axis.

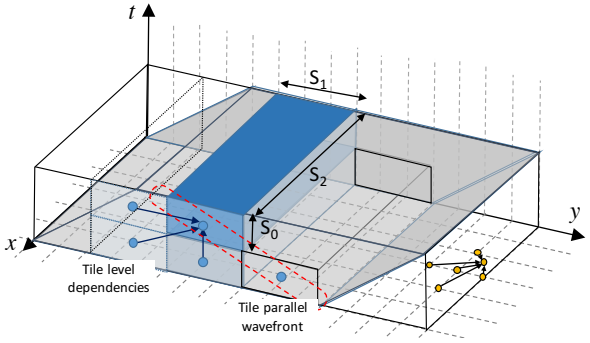


Fig. 2: Illustration of partial tiling for a 2D jacobi

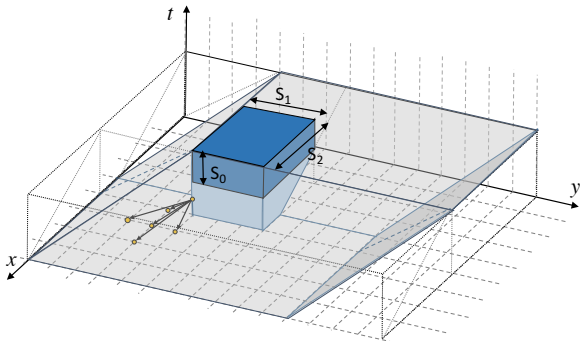


Fig. 3: Illustration of full oblique tiling for a 2D Jacobi

```
void top(T* arr_in, T* arr_out) {
#pragma HLS DATAFLOW
    fifo fin, fout;

    // Read Actor
    for (int i=0; i<N; i++)
#pragma HLS PIPELINE
        fin.write(arr_in[i]);

    // Compute Actor
    fout.write(fin.read() * 2);

    // Write Actor
    for (int i=0; i<N; i++)
#pragma HLS PIPELINE
        arr_out[i] = fout.read();
}
```

Fig. 4: Use of the `DATAFLOW` directive to implement computation / communication overlapping.

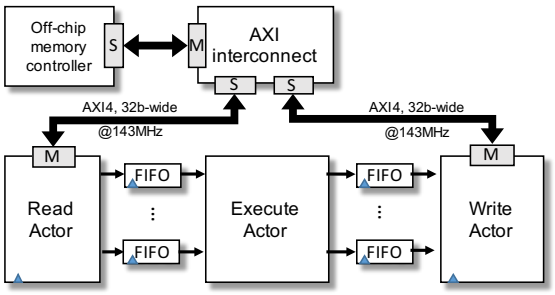


Fig. 5: Diagram of the architecture.

full control on the tile memory footprint (and therefore on on-chip memory requirements).

The two aforementioned tiling strategies are based on hyperplane partitioning and result in parallelepipedic tiles, however it is also possible to use more complex shapes. For example, the **overlapped** tiling approach uses trapezoid/pyramid shaped tiles to avoid dependence violations in exchange for redundantly computing some values in each tile. However it also enables concurrent start of tiles (i.e., tiles within a same time step can start concurrently, which is not the case when using oblique tiling). This is valuable when taking advantage of tile level parallelism, but is often not required for acceleration on a single FPGA.

We summarize the characteristics of the various tiling strategies (discussed more in Sections V and VI) in Table I.

III. Architectural Design Space

We now present our accelerators, and show how their design parameters address the concerns raised in Section II.

A. Overview of the Architecture

Our proposed accelerator is implemented as a bus master device on the AXI4 bus, using HP ports to access external memory. Although we mostly discuss Zynq in this paper, our design is portable to other platforms with AXI. Our implementation decouples memory accesses from execution through macro pipelining at the tile level. This is achieved through Vivado HLS `DATAFLOW` pragma: inputs to the next tile are fetched while the current one is being processed, as shown in Figure 4, where the hardware accelerator is decomposed into a three stage macro-pipeline.

We obtain a family of accelerators (see Figure 5) that operate on a series of tile coordinates, corresponding to a wavefront of independent tiles, computed as a single hardware call. The host program running on the ARM processor is in charge of sequencing the execution of wavefronts, but can also be used as processing element.

B. Overview of the Execution Datapath

Each tile is computed in a single sweep using a deeply pipelined datapath. The depth, Δ, of the pipeline depends on the target operating frequency provided by the user. We use 143 Mhz as the target frequency for both the IP and the AXI4 bus, as this is the highest frequency supported by both SDSoC and the AXI interconnect (limited to 150 MHz).

The entire set of operations in a tile is pipelined with Initiation Interval of one. In terms of input C code to HLS, this pipelining is realized by coalescing the entire loop nest that iterate over computations in a tile, and pipelining the resulting loop. The updates within a single time step are independent of each other and can be fed to the datapath every cycle, provided that the data is available. Pipelining across time steps require that the results from the previous time step have exited the pipeline before its use. The constraints over tile shapes are later discussed in Section III-D.

Our datapath can be further configured to perform an arbitrary number of stencil updates per cycle, simply by unrolling the innermost loop by a fixed factor before coalescing. Adjusting this factor allows us to control the computational intensity of our IP.

The execute actor takes advantage of reuse of input data and intermediate results within a tile. We apply a technique similar to the one by Cong et al. [9] to minimize local memory usage and to avoid memory bank conflicts, but using HLS arrays instead of explicit FIFOs, which necessitates dealing with the initialization of these arrays for each tile.

This can be achieved in two ways (i) by increasing the number of memory ports in the on-chip memory to parallelize the initialization (i.e., without performance overhead), or (ii) by inserting wait-states to serialize the initialization phase. We use the latter as on-chip memory is a scarce resource.

This overhead comes from what we call the *halo* regions.

TABLE I: Qualitative analysis of existing approach to accelerate stencil on FPGAs. Non-tiled refers to versions where the data-set either fully fit in on-chip memory, or is communicated to/from external memory at every sweep. However, these versions may still benefit from sophisticated data-reuse along the spatial dimensions.

Approach vs. characteritics	non-tiled on-chip [6]	non-tiled off-chip [5], [9]	partial oblique tiling [10]	full oblique tiling (ours)	overlapped tiling [7]
Domain size scalability	✗✗	✓	✗	✓✓	✓✓
Off-chip memory bandwidth constraints	✓	✗✗	✓✓	✓✓	✓✓
Computational overhead (redundant or useless computations)	✓	✓	✗	✗✗	✗✗✗

Some values used by a tile come from neighboring tiles. For stencils, these values are always adjacent to the tile. The compute actor scans these regions for initialization, adding cycles where the datapath is only propagating data. Because the halos are d dimensional (tiles are $d+1$), the overhead diminishes as tile size increases.

C. Overview of the Read/Write Actors

The read actor streams in input data to the execute actor, and the write actor streams out results from the execution actor. These actors perform burst accesses to external memory through the AXI4 interface. We use a custom data layout, which we omit for space reasons, to ensure that most of the memory accesses are contiguous[1].

We take special care to minimize idle time and maximize bus occupation to get as close as possible to the maximum achievable bandwidth (600 MB/s with 32 bit bus width). Up to four HP ports on the Zynq can be used concurrently.

D. Design Parameters

Our approach aims at exposing relevant design knobs to drive the design space exploration. These knobs are:

- The choice of the Unrolling Factor (UF). The datapath performs UF updates, and hence the value of UF determines the amount of parallelism. Increasing this factor will boost the maximum throughput that can be attained, but will also raise bandwidth requirement to keep feeding the datapath.
  The choice of this parameter is mostly driven by the throughput requirement. Larger values give higher throughput, but increase area cost.
- The choice of the tile sizes (S_0, S_1, S_2) is also critical, as tile shape determines data locality. Tile sizes control the trade-off between off-chip bandwidth requirement and on-chip memory usage. Larger sizes reduce bandwidth requirement, but increase on-chip data storage.
  Larger tile sizes also reduce the overhead due to halo regions, further improving throughput.
- The use of partial oblique tiling can be beneficial since it has fewer partial tiles, but at the cost of increased on-chip memory requirements.

[1]Our approach is based on tile-face projections along canonical axes. It comes at a small overhead cost, as some values are mapped to multiple faces, but reduces all tile inputs/outputs to $d+1$ contiguous segments.

We require that S_2 is evenly divisible by UF to avoid complex controls arising from cases where only a subset of the unrolled iterations are valid computations. The tile sizes in the spatial dimensions are constrained to have more iterations than the pipeline depth: $S_1 \times \frac{S_2}{\mathrm{UF}} > \Delta$.

IV. Performance Modeling

The parameters above expose a huge design space to be explored. In this section we present a performance model to guide the exploration of this space.

A. Asymptotic Performance

The important metric to model is the number of stencil updates per cycle[2], computed as follows:

$$\mathrm{UpdatesPerCycle} = \frac{\mathrm{TileVolume}(S_0 \times S_1 \times S_2)}{\mathrm{TileCycles}}$$

where $\mathrm{TileCycles}$ denote the number of cycles it takes to execute a tile. Assuming that the communication is overlapped with computation, this is the slower of the number of cycles spent for computing, $\mathrm{CompCycles}$, and spent for communicating, $\mathrm{CommCycles}$:

$$\mathrm{TileCycles} = \max(\mathrm{CompCycles}, \mathrm{CommCycles})$$

This is the asymptotic performance of our design that is reached when the problem size is large enough to make the overhead at the boundaries (where the computation and communication are not fully overlapped) negligible.

1) Performance of the Compute Actor: The compute actor is centered around a pipelined datapath that computes, in steady-state, UF updates per cycle. In addition to the tile volume, the compute actor scans the boundary *halo* regions to fetch input data. Representing the extend of the halo in the d-th data dimension as h_d, the number of times the compute actor datapath is invoked per tile is:

$$\mathrm{CAVolume} = S_0 \times (S_1 + h_1) \times \left\lceil \frac{S_2 + h_2}{\mathrm{UF}} \right\rceil$$

Since the initiation interval is always 1 for our design, the total number of cycles that it takes to execute the compute actor, assuming all inputs are ready, is given by:

$$\mathrm{CompCycles} = \mathrm{CAVolume} + \mathrm{Depth} - 1$$

[2]Note that $\mathrm{UpdatesPerCycle}$ is a direct proxy to throughput, which is $\mathrm{UpdatesPerCycle} \times \mathrm{FlopsPerUpdate} \times \mathrm{Frequency}$.

where Depth is the pipeline depth of the compute actor datapath. The pipeline depth, determined by the HLS tool during RTL generation, is a function of the update formula and synthesis frequency, but not tile size or unrolling factor.

2) Communication Modeling: It is critical to make use of burst communication to maximize bandwidth utilization. We ensure that almost all memory transfers permit burst accesses using a custom memory layout. Furthermore, the concurrent use of four HP ports completely hides the latency of burst transfers. Hence, modeling the communication cost can be simplified to modeling the data volume.

The data volume to be communicated is exactly the halo regions of a tile. This can be computed as:

$$\mathrm{CommVolume} = \prod_{i=0}^{d} (S_i + h_i) - \prod_{i=0}^{d} S_i$$

When the data element is one word, $\mathrm{CommVolume}$ directly translates to the number of transfer cycles: $\mathrm{CommCycles}$.

B. Modeling the Area Cost

Precise modeling of the area cost can be extremely challenging, and is heavily influenced by the HLS tool. However, it is not difficult to make a relative comparison among design points in our parameter space.

We expect that the unrolling factor and the tile face volumes both have linear relationships with area: UF with LUTs/DSPs and tile faces with on-chip buffer requirement.

We use the sum of the utilization rates of Slice/BRAM/DSP as area metric (see Section V). In order to capture the interaction between UF and tile sizes, and to relate these values to area metrics, we used linear regression to compute, respectively, two functions:

$$C_{\mathrm{dp}} = a_{\mathrm{dp}} \times \mathrm{UnrollFactor} + b_{\mathrm{dp}}$$
$$C_{\mathrm{mem}} = a_{\mathrm{mem}} \times \mathrm{CommVolume} + b_{\mathrm{mem}}$$

C_{dp} models datapath cost and C_{mem} on-chip memory cost. Parameters $a_{\ldots}, b_{\ldots}$ are inferred by the regression tool. The sum of utilization rates for Slice and DSP is a function of UF, and BRAM usage is a function of parameters affecting communication volume. We only needed a few samples (three or four, one per power-of-two unroll factors up to the largest factor that fits on the board) to learn the area model. The RMSE was between 4% and 7% for the response variable (sum of utilization rates) that range from 30 to 130.

V. Validation and Discussion

As mentioned in Section I, we have developed a tool that take a stencil kernel specification as input. Our code generator builds on polyhedral compilation tools (namely the Integer Set Library [11]) to generate the tiled loops, and python scripts for generating the communication actors.

We emphasize that our goal is not necessarily to present a design with the highest throughput, but rather to show that we are able to select the "right size" for a given context. We have generated a series of design using different tile sizes, unrolling factors and tiling modes (full and partial).

TABLE II: Number of floating-point operations, and pipeline depth for one update of the kernels.

Kernel	flops	Pipeline depth
Jacobi 2D	$1\times$, $4+$	43
Anisotropic Diffusion	$9\times$, $17+$, $2/$, $9\exp$	87

Each of these designs were automatically generated from the design parameters and synthesized using Xilinx SDSoC 2016.3, targeting the ZC706 board with an XC7Z045 Zynq chip. The target frequency for all designs was 142.86 MHz.

Unlike many prior work, our performance numbers are obtained from actual accelerators instances running on the target FPGA platform. Hence, our results account for all performance degradation issues related to bus interconnect and/or external memory.

We validate our work on two different stencil kernels: Jacobi 2D and Anisotropic diffusion. Jacobi 2D is a standard example for stencils that have relatively few number of operations, and is strongly bandwidth constrained. Anisotropic diffusion is an iterative smoothing filter, which is much more compute-intensive. The characteristics of their update operations are summarized in Table II.

In this section, we abbreviate a design as S_0xS_1xS_2_UF. The area cost is the sum of utilization rates for Slices/BRAMs/DSPs, and takes a value between 0 and 300. The target board has 54650 slices, 545 BRAM tiles, and 900 DSPs.

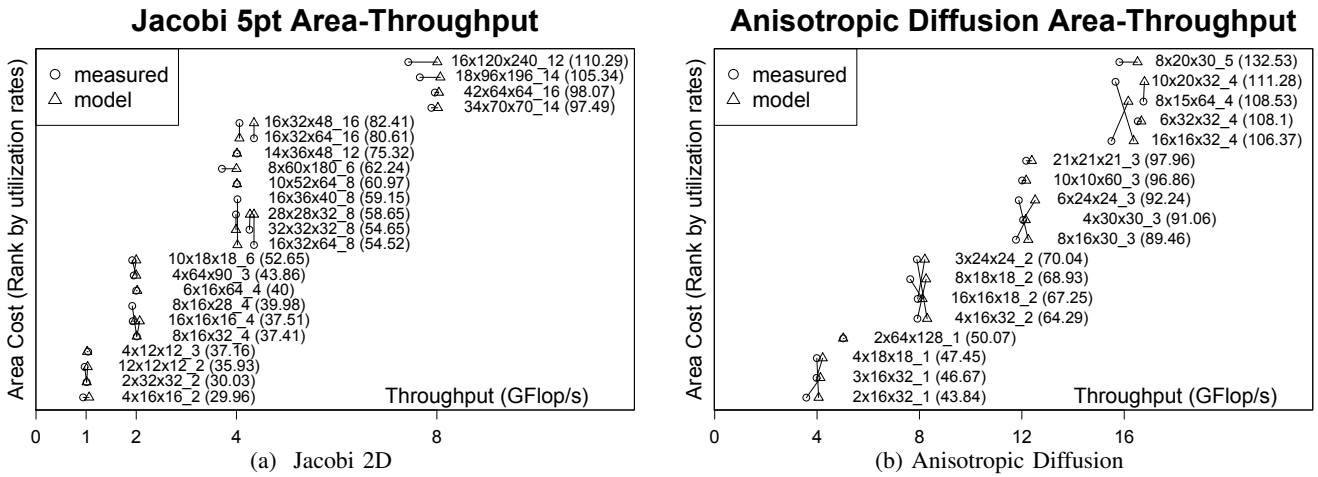
A. Jacobi 2D

We use four target performances; 1, 2, 4, and 8GFlop/s; to illustrate the trade-offs exposed by our design knobs. A number of design points that have the desired performance with different tile shapes were synthesized. Linear regression for the area model used the following four design points: 4x16x16_2, 8x16x32_4, 32x32x32_8, and 64x64x64_16.

Figure 6a summarizes the area and throughput of the resulting designs, as well as those predicted by the model. One thing that is clearly visible in the figure is that the performance model is quite accurate. Almost all points are on the target GFlop/s based on the model. The largest divergence from model is 7% (16x120x240_12).

The area result is also in agreement with the model. There are some interchanges when compared to the predicted ranks, which is due to powers-of-two tile sizes. When the tile size in a dimension is a power of two, the control logic can be significantly simplified. This favors powers-of-two tile sizes over slightly smaller tile sizes with less buffer usage.

In most cases, using the smallest UF is also area efficient. However, there are some cases where using higher UF can be beneficial. Design points such as 8x60x180_6 and 16x120x240_12 are examples of these cases. This is explained by the diminishing returns from increasing the tile sizes. Increasing the tile sizes improves performance in two ways: by improving locality, and by reducing the overhead of the halo regions. Once tiles are large enough that data locality is sufficient to keep the datapath busy (i.e., the accelerator is no longer I/O-bound), further performance comes only from



(a) Jacobi 2D (b) Anisotropic Diffusion

Fig. 6: Predicted and measured area/throughput of the two benchmarks. Area metric is the sum of utilization rates for LUTs, BRAMs and DSPs. The labels are placed next to measured values (circles), with a line segment connecting each point to its modeled counterpart. Y-axis is the rank of the area cost in ascending order to avoid overlapping of points. The actual area utilization is shown next to its label.

overhead reduction. The above designs are in such situations, where the performance target is at the limit of what can be achieved by the given UF: larger tile sizes have to be used to meet the goal. Our performance model can identify these situations and point to better designs.

We report the resource usage for the best performing designs for each target performance in Table IIIa.

B. Anisotropic Diffusion

We use four target performance levels: 4, 8, 12, and 16 GFlop/s. A number of design points that have the desired performance with different tile shapes were synthesized. The area model is learnt with the following points: 2x16x32_1, 4x16x32_2, and 16x16x32_4. The area-throughput trade-off is summarized in Figure 6b, and Table IIIb reports the detailed resource usage for the best performing designs.

We do not repeat the same discussion as in Jacobi 2D case; all of them applies to anisotropic diffusion as well. One key difference is that the importance of BRAM is much less significant compared to Jacobi 2D. This is because the arithmetic intensity of this kernel is high (37 floating-point operations, including 9 exponentiation), and not much data locality is needed to keep the accelerator busy.

C. Comparison with Earlier Work

It would be interesting to directly compare our design with earlier work. However, this is not practical due to the availability of the implementation and the FPGA platform used in prior work. We are still able to compare the system-level characteristics of a large subset of prior work that are within the family of designs covered in our work.

The non-tiled variants [5], [6], [9] (recall Table I) may be viewed as designs with tile size in the time dimension set to 1, and the remaining dimensions equal to the problem size. These designs do not exploit temporal locality, and are not suited for iterative stencils. They can give similar performance to other designs only for small problem sizes, where the entire data fit on chip. We have implemented a few non-tiled designs to highlight the scalability issue of non-tiled designs. Attaining 1GF/s with Jacobi 2D kernel for 256×256 image uses 25% of the available BRAM, and the limit is reached with 512×512 using 95% of the BRAM.

Partial tiling [10] is an attractive alternative for small problem sizes. We implemented this strategy and compared to fully tiled cases. Figure 7 illustrates the trade-offs between the two approaches. Partial tiling is beneficial for relatively small problem sizes with moderate performance requirements. For anisotropic diffusion, it scales to larger problem sizes because its operations are much more compute-intensive, and the tile sizes in the remaining dimensions can be kept small and still have sufficient data locality.

We did not implement overlapped tiling, because the overhead due to redundant computation is too significant (e.g., 367% with $12 \times 12 \times 12$ tiles). For 2D data stencils, it is a cubic function of tile size, and so tile shapes must be thin and flat tiles to limit the overhead. This approach may be more attractive for platforms with higher bandwidth.

D. Additional Considerations

We have extensively discussed the trade-offs of different design choices in this section. There are other factors that can also influence the trade-off, including the frequency of the compute actor, and the overhead of different tiling strategies.

In this work, all the designs were synthesized at the same frequency (143 MHz) on a single clock domain. The communication and computation parts could use independent clocks, allowing the compute actor to reach higher frequencies than the 150 MHz limit of the AXI bus.

Padding the domain with dummy computations to execute incomplete tiles in hardware has an impact on overall performance that depends on tile size, problem size, and tiling strategy. This impact can be relatively important with full tiling (e.g., 10.09% dummy iterations with $4 \times 16 \times 16\_2$ tiles on a $50 \times 512 \times 512$ domain) and is smaller with partial tiling

TABLE III: Resource usage of the kernels, selecting the best design for each performance target in Figure 6.

(a) Jacobi 2D

Design	Slices	BRAMs	DSP48E
$2 \times 32 \times 32$, UF=2	9663	27	66
$8 \times 16 \times 32$, UF=4	11123	30	104
$16 \times 32 \times 64$, UF=8	13148	57	180
$34 \times 70 \times 70$, UF=14	18103	126	372

(b) Anisotropic Diffusion

Design	Slices	BRAMs	DSP48E
$2 \times 16 \times 32$, UF=1	12675	29	138
$4 \times 16 \times 32$, UF=2	17119	30	248
$8 \times 16 \times 30$, UF=3	22961	32	375
$16 \times 16 \times 32$, UF=4	26000	37	468

for the same performance target (2.63% with $2 \times 16 \times 512\_2$ with the same domain size).

Finally, we only considered single-field stencils with Jacobi-style dependences operating on 32-bit floating-point data. Our generator could be easily extended to Gauss-Seidel dependence patterns and multi-field stencils such as FDTD. The use of fixed-point or custom floating-point arithmetic opens further trade-offs involving accuracy.

All these factors influence throughput and/or area, and will impact the specific trade-offs and performance modeling. The scalability illustrated in Figure 7 is mostly unaffected, and the key message—**that there is no single design that works best in all cases**—is expected to hold. We make the case with an important subset of the design space.

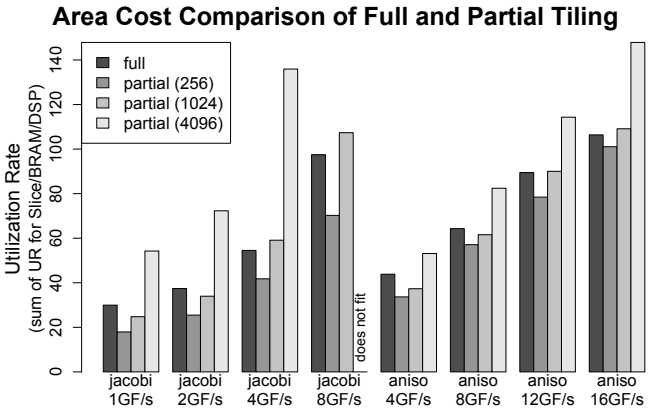


Fig. 7: Area cost comparison between full and partial tiling for each performance target. The different bars for partial tiling corresponds to the size of the untiled dimension, which is the size of the problem that can be executed. Fully tiled designs scale to any problem size. Jacobi 2D with partial tiling for 4GF/s target (UF=8) used 99% of the BRAMs, and 8GF/s target (UF=16) could not fit when $S_2 = 4096$.

VI. Related Work

Two bodies of related work are relevant: (i) tiling in compilers and (ii) FPGA stencil accelerators.

Tiling in Compilers: Tiling is a classical loop transformation used in various contexts [12], [13]. 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., [14]) also use tiling based parallelization as its core strategy.

Tiling is known to be one of the most important transformations for many classes of programs, including stencils. Due to the importance of stencil computations, and its regular dependence pattern, other variants of tiling have been proposed [15], [16]. 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 2). 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 [17]–[19]. These work also employ variations of tiling combined with additional optimizations.

At the high-level, we use the parallelism identical to those utilized by tiling-based parallelizers. 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. This is yet another direction that enriches the design space, and is expected to further complicate the exploration.

Stencil Computations on FPGAs: There have been several ad-hoc implementations of FPGA hardware accelerators targeting stencil-like algorithms, such as optical flow estimation [5] and/or FDTD [1], [2]. 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 [7], [20], although they do not use the name tiling. These work use a variant of tiling (known as overlapped tiling [16]) 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. [6] 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 [8], [9], [21]. 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.

VII. Conclusions and Perspectives

We showed how to synthesize a family of FPGA accelerators for stencil computations (our tools are available as open source[3]). Our approach builds upon loop transformations (tiling, skewing) and covers a large design space.

Our work opens up an interesting perspective. In the exascale era, energy is the dominant performance metric, and an exponentially increasing fraction of future generation chips will remain "dark" (powered off) or "dim" (in low-power mode). Therefore, *accelerators* will dominate exascale processors, and compiling to such a fabric is in important challenge. Our work can be generalized towards this.

Programs mapped to accelerators will be *asymptotically compute bound*: the total number of *operations* performed is at least one polynomial degree larger than the *volume of data* accessed by the program. Also, the accelerator hardware is *resource constrained* in computation and storage: it can only perform/store a small fraction of the operations/data of the program. Our design methodology, and associated design space exploration are well suited here.

References

[1] 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*, 2004, pp. 213–222.

[2] 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*, April 2005, pp. 127–136.

[3] 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*, Sept 2011, pp. 50–57.

[4] 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*, vol. 29, no. 2–3, pp. 51–62, 2005, special Issue on FPGA Tools and Techniques.

[5] 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*, Sept 2014, pp. 1–4.

[6] 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*, 2012, pp. 26–39.

[7] 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*, 2013, pp. 52:1–52:6.

[8] 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*, Oct 2014.

[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*, 2014, pp. 77:1–77:6.

[10] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "A polyhedral model-based framework for dataflow implementation on FPGA devices of iterative stencil loops," in *Proceedings of the 35th International Conference on Computer-Aided Design*, 2016, pp. 77:1–77:8.

[11] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Proceedings of the 3rd International Congress on Mathematical Software*, Sep. 2010, pp. 299–302.

[12] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th Symposium on Principles of Programming Languages*, 1988, pp. 319–329.

[13] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the 12th Conference on Programming Language Design and Implementation*, 1991, pp. 30–44.

[14] 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*, 2008, pp. 101–113.

[15] 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*, 2014, pp. 66:66–66:75.

[16] 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*, 2007, pp. 235–244.

[17] 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*, May 2011, pp. 676–687.

[18] 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*, 2011, pp. 117–128.

[19] 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*, 2013, pp. 13–24.

[20] 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*, 2011, pp. 65–74.

[21] 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*, 2014.

[3] https://github.com/gdeest/hls-stencil