On How to Accelerate Iterative Stencil Loops: A Scalable Streaming-Based Approach

RICCARDO CATTANEO, GIUSEPPE NATALE, CARLO SICIGNANO, DONATELLA SCIUTO, and MARCO DOMENICO SANTAMBROGIO, Politecnico di Milano

Received June 2015; revised September 2015; accepted October 2015

Authors' addresses: R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio, Politecnico di Milano, c/o Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Via Ponzio 34/5, Milano (MI) 20123, Italy.

ALGORITHM 1: Generic ISL Algorithm

```
foreach $t \leqslant TimeSteps$ do
    foreach points $p$ in matrix $P$ do
        $p \leftarrow f_{transition}(stencil(p))$
    end
end
```

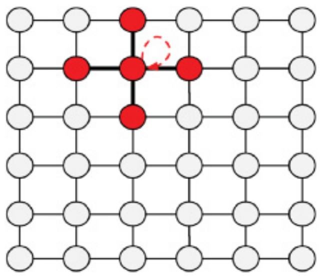


Fig. 1. On the left, a generic ISL pseudocode. On the right, an illustration of a generic 5-point 2-dimensional ISL.

1. INTRODUCTION

Numerical methods for partial differential equation solving employed in weather and ocean modeling [Marshall et al. 1997; Bleck et al. 1992], fluid dynamics [Ding and Shu 2006; Satofuka 2001], quantum dynamics simulations [Nakano et al. 1994], heat diffusion [Frigo and Strumpen 2007], geometric modeling [Kim and Shimada 2006], nonequilibrium statistical mechanics [Cao and Dong 2012], seismic simulations [Robertsson 2012], cellular automata [Neumann 1966], multimedia/image-processing applications such as Haralick and Shapiro [1992], Gaussian smoothing [Shapiro and Stockman 2001], and Sobel edge detection [Arandiga et al. 2010] represent only a fraction of a wide class of applications sharing the same computational nature, in which a series of sweeps are performed over a regular grid, whose points are updated using a fixed nearest-neighbor pattern. Algorithms and codes structured this way are called *Iterative Stencil Loops (ISLs)*.

These algorithms update values associated with points on a multidimensional grid using weighted contributions from a subset of its neighbors in both time and space (see Figure 1). The fixed pattern of neighbors is called a *stencil*, and the function that uses those elements to update an array cell is called a *transition function*. An ISL can be generically represented by the pseudocode of algorithm reported in Figure 1 on the left. As already mentioned, the number of algorithms whose computational kernels fall into this category is large and relevant to both industry and science, which is why efficiently implementing them is of great concern.

As they are characterized by a regular computation structure, they are ideal candidates for automatic compile-time analysis and transformation aimed at improving their runtime performance. While many state-of-the-art works precisely explore this opportunity (particularly in the multicore/multiprocessor domains), ISLs' *memory boundedness* and *spatial dependencies* limit the scope of current transformation, analysis, or optimization techniques [Meng and Skadron 2009; Li and Song 2004; Christen et al. 2011].

With *ISLs' memory boundedness,* we refer to ISLs' requirement for potentially large transfers of data blocks in order to compute the next state of each element of the problem grid. Additionally, depending on the size of the grid and the stencil, and the degree of parallelization, processor-to-processor bandwidth might be an issue, as *halos* (i.e., portions of data processed by a processing unit and required by another to progress the computation) must be transferred too.

ISLs' spatial dependencies, on the other hand, refer to the presence of true data dependencies in ISLs' codes between updated points of the grid within the same time frame. The Gauss-Seidel method [Salkuyeh 2007] is a perfect example of an ISL with spatial dependencies. Depending on whether further true data dependencies are present inside each time frame or not, automatic parallelization and further aggressive optimizations might or might not be made, making the overall automatic optimization process heuristic and more complex.

Some important consequences induced by these properties that must be taken into account when designing custom accelerators are as follows:

—The performance of parallelization-oriented techniques—specifically, the ubiquitous tiling—is bounded by the available off-chip and on-chip bandwidth and the need for synchronization of the parallel units.Hence, in most cases, the achieved performance is far below the predicted one.
—Techniques designed to take advantage of data locality are not really effective when designing custom accelerators, mainly due to the nature of the computing architectures on which they are applied; a well-designed pipelined architecture, given the regular structure of ISLs' codes, is superior in structure and performance to the tiled one.
—Although the employment of custom logic explicitly designed to accelerate a specific ISL is promising, it is in general a hard task, specifically when the design methodology is required to address the acceleration of problems with realistic or growing problem sizes.

Within the scope of this work, we aim at alleviating the aforementioned issues by introducing a novel code analysis and synthesis methodology that allows designers to specify an ISL using plain C/C++ codes and obtain a highly pipelined, scalable, and high-performance hardware accelerator.

Contributions

To the best of our knowledge, no work addresses all the presented issues *at once*, as it either focuses on a subset of them or fails to properly consider bandwidth and memory as *the* limiting factor of a system designed to scale. The aim of our work is to *simultaneously* address all these issues at once with the definition of a proper ISL-centric methodology.

Therefore, our contributions are as follows:

(1) A streaming architecture implementing a single stencil time-step able to realize full data reuse within the constraint of a single time-step, with the minimum on-chip memory requirements, while allowing for multiple concurrent accesses, the Streaming Stencil Time-step (SST); we realize it as a distributed architecture exploiting the inherent parallelism of a FPGA.
(2) A technique specifically aimed at scaling the throughput of the optimal *SST*-based system, which we called *SST queuing*, characterized by constant off-chip bandwidth and progressive improvements of the power efficiency with the occupied area
(3) An architecture and a methodology to exploit it—a *design automation flow*—to automatically implement ISLs with the proposed hardware accelerator.

The remainder of this article is organized as follows. Section 2 thoroughly reviews related works and their limitations. Section 3 is an introduction to the overall methodology. The SST, the core computational element of the proposed system, is described in Section 4. Section 5 focuses on the scalability issues of the proposed SST technique. Section 6 provides experimental results, both of the single SST component and the queued ones. Finally, Section 7 concludes the article and elaborates on future works.

2. STATE OF THE ART

Iterative Stencil Loops (ISLs) have been extensively studied in state-of-the-art works. We divide this section into three parts to introduce the most relevant techniques and compare them with our solution.

2.1. Tiling-Based Optimizations

Tiling, also known as *blocking*, is a code optimization technique employed to both enhance data locality and expose parallelism. This technique has been exploited in a number of different ways and performed in both spatial—when possible—and temporal dimensions [Rahman et al. 2011], as we show later.

Single iteration tiling. The earliest and simplest tiling-based technique usually employed both in CPU and GPGPU code optimization consists of applying conventional loop blocking to improve cache reuse. In this case, a single time frame (i.e., a single iteration) is partitioned into smaller blocks, allowing points that are close in space to remain in cache when used, thus allowing them to update them together, improving locality [Kamil et al. 2005]. This technique has also been exploited to distribute the computation to multiple processing elements in order to parallelize the computation of next-state points within a single iteration [Frigo and Strumpen 2005], also leveraging specific APIs such as OpenMP. However, tiling across multiple processing elements potentially incurs heavy off-chip and on-chip bandwidth requirements, as stencils along a tile's boundary require values that were previously computed by other processing elements, increasing communication and synchronization between them. An effective technique to overcome this issue is the one known as *ghost zone optimization* or *overlapped tiling* [Meng and Skadron 2009; Krishnamoorthy et al. 2007], which consists of the enlargement of the tiles with ghost zones, that is, the overlapping regions between tiles, replicating some computations but nevertheless reducing communication and synchronization. Although this technique might mitigate communication issues, an improper selection of the ghost zone size may result in even worse performance with respect to no optimization at all [Holewinski et al. 2012]. Additionally, the execution of ISLs on general-purpose processors usually translates into poor power efficiency figures compared to custom logic accelerators, as most of their area is dedicated to coping with irregular codes (which is an inefficiency when the executed code is indeed regular). As a final remark, dealing with ISLs with spatial dependencies between grid points forbids the application of this optimization scheme for parallelization purposes, and performance is usually degraded even with cache optimizations as outlined in Weiß et al. [1999].

Time skewing. In this tiling scheme, multiple iterations are *collectively* partitioned into blocks: with respect to single-iteration tiling, multiple iterations are computed as part of each tile. The reason beyond the application of such a strategy (specifically in CPUs and GPGPUs) is to exploit temporal locality too, thus increasing the overall data reuse factor. However, in order to make tiling legal, loop skewing [Wolf and Lam 1991] along the time dimension is required. In fact, as point updates occur in both spatial and temporal dimensions in each block, they must shift their collection of points backward on the time dimension to respect temporal dependencies induced by the ISL, that is, transform dependency distances into nonnegative values [Renganarayana et al. 2007], resulting in a loss of intertile concurrency (skewing introduces intertile dependencies in the spatial direction). While it might seem that this scheme always delivers better performance than the simple single-iteration version, it really depends on a careful selection of the skewing factor [Rahman et al. 2011], as well as on the form of the tile [Strzodka et al. 2011; Renganarayana et al. 2007], which can be a major concern especially on FPGAs [Zuo et al. 2013]. With respect to the previously mentioned strategy, time skewing can provide better cache hit rates and effectively reduce processor idle time caused by the ISLs' memory boundedness [Wonnacott 2000] in CPUs and GPGPUs.

As for the previous tiling strategy, even in this case it is possible to distribute the computation among different processing elements [Alias et al. 2012], but likewise with single-iteration tiling, this requires explicit synchronization between them, since a block must wait for its neighbors to complete in order to have enough data to start. As a consequence, rather than a purely parallel execution, time-skewing blocks are executed in a pipeline fashion.

A possible solution to mitigate time-skewing adverse effects when tiling along multiple iterations is proposed in Nacci et al. [2013], where *code transformation* is performed to fuse the stencil loops together in order to reduce the number of reads and writes and increase both the computational intensity and data reuse via local buffers. While this is an effective approach to cope with ISLs' memory boundedness, scaling their approach is not straightforward, as they focus only on data reuse efficiency rather than scaling out the system. A very similar technique has been developed in Chipeperekwa [2013], in which a domain-specific compiler is proposed, namely, *Caracal*, that is able to perform unrolling of the time loop and fuse accordingly the stencils, with comparable effects as of Nacci et al. [2013].

Wavefront parallelization. Instead of pipelining the execution of time-skewed blocks, wavefront parallelization dictates that blocks must be scheduled collectively in a *wavefront* fashion [Wellein 2009; Schäfer and Fey 2011; Treibig et al. 2011]. Blocks are arranged in a way that on the time dimension the computation blocks are independent from each other, thus not requiring synchronization. Although in Rahman et al. [2011] this scheme has been explicitly defined as the one in which *multiple* blocks are *scheduled together*, this class can be easily extended to the case in which only *one* block implements a single iteration. Indeed, this is exactly the behavior exhibited when tiling is *only* applied on the time dimension [Sano et al. 2011, 2014; Niu et al. 2013]; this approach is promising as it has been proven to scale [Sano et al. 2011, 2014; Niu et al. 2013] with low communication overhead. We draw inspiration from this work and largely extend it to be able to build a dataflow architecture customized around the specific ISL workload.

2.2. DSLs-Based Optimizations

The exploitation of Domain-Specific Languages (DSLs) and ad hoc frameworks has been explored as an interesting way to drive compile-time optimizations by exposing specific, custom problem semantics to compilers. While General-Purpose Languages (GPLs) are the dominating software development tools in High-Performance Computing (HPC), the lack of specialized language constructs and semantics to serve domains such as ISLs is a limitation, since most of the time they do not allow one to express a problem in a manner that gives compilers the ability to explicitly and directly manipulate the code.

DSLs are certainly interesting, as they allow designers to define a problem in a way that some features are explicitly signaled to the compiler, enabling whole kinds of aggressive manipulations, optimizations, and specific code generation techniques. However, this comes at the cost of losing broad applicability, as they usually can be used to express only a fairly limited set of algorithms.

Among ISL-oriented DSLs, PATUS [Christen et al. 2011] is able to achieve high performance by means of *auto-tuning*, targeting different hardware architecture, while Pochoir [Tang et al. 2011] provides a C++ template library based on a divide-and-conquer skeleton that is then translated into Cilk [Blumofe et al. 1995], a C/C++ extension designed for multithreaded parallel computing. ExaStencils [Lengauer et al. 2014] employs a direct mathematical formulation (ExaSlang) of the problem and, through a series of steps of transformations, including a wide range of polyhedral model-based optimizations, generates target code in a specific language, which by now is C/C++ but in the future could be extended to other languages. DeLite [Sujeeth et al. 2014] abstracts from Scala with the aim of making stencil programming easier and uses meta-programming to construct an Intermediate Representation (IR) of the problem and compile to a large number of languages so that it can easily target heterogeneous hardware. In Zhang and Mueller [2012], a single mathematical formula is used to implement 3D stencil codes on GPGPUs via auto-tuning and automatic target code generation, and GPGPUs are also the target device of Holewinski et al. [2012], in which low-level code is generated, starting from an abstract representation, by trading an increase in the computational workload for a decrease in the required global memory bandwidth. In Wester and Kuper [2014], a single high-order function specified in Haskell, and specifically in CλaSH [Baaij et al. 2010], a functional HDL able to translate plain Haskell (with some restrictions) into VHDL, is used in combination with a series of transformations to generate hardware accelerators.

As a final consideration, although using DSLs can lead to good performance, in HPC and industrial-grade high-performance embedded systems, this is not common at all, as GPLs are preferred due to their versatility, ease of use, and vast availability of optimized libraries and components. Additionally, designers are usually familiar with imperative languages like C or C++, which is a major limitation in the adoption of DSLs, which employ custom (and not necessarily imperative) semantics; indeed, most available High-Level Synthesis (HLS) tools ship with support for C and C++. For these reasons, we adopt a subset of C as our input language, and specifically a fragment of it analyzable by means of the polyhedral model, in order to both support an industrial-grade language and be able to heavily analyze, restructure, and manipulate input codes to optimize the resulting implementations, as we thoroughly describe in Section 3.

2.3. Custom Architectures

When designing custom hardware, Field-Programmable Gate Arrays (FPGAs) can offer both high flexibility and sustained performance with high energy efficiency, often orders of magnitude better than other hardware platforms, depending on the target workload. In fact, an increasing number of works are focusing on exploiting FPGAs to implement ISLs with the development of custom hardware, finely designed to efficiently leverage the regular structure of this class of algorithms. Specifically, the design of custom hardware has been proven useful in mitigating the memory boundedness issue of stencil computations.

In Ritcher et al. [2012], for instance, a generic tunable VHDL template has been proposed to parallelize 3D stencil computations. Their work uses the so-called Full Buffering [Liang et al. 2001] instead of Partial Buffering (which is a strategy where solely the data needed by the current computation is stored to minimize memory consumption), a technique in which data is stored in the on-chip memory until all the computations depending on it have completed, showing that the increasing number of available resources in modern FPGAs allows one to obtain very good performance. However, the contributions of this work do not include either an explicitly streaming mechanism or a scalable solution (i.e., capable of targeting multiple processing elements with adequate memory and bandwidth considerations), which we do in our work.

In Cong et al. [2014], the polyhedral model is employed to take advantage of the stencil access pattern and perform nonuniform memory partitioning in order to generate a custom microarchitecture, streaming oriented, which is proven to be optimal with respect to memory usage, since it allows Full Buffering with the minimum number of reuse buffer banks and minimum buffer size. This architecture has only been simulated, without actual bandwidth and memory consumption figures, the main bottleneck when scaling an architecture to multiple computational elements. Additionally, the case in which the computation has more than one input channel (a common situation in ISL computation) is not covered at all. Cong et al. [2014] lack two aspects that are instead properly addressed by this work: (1) their architecture is not able to deal with ISLs with spatial dependencies among points updates, and (2) they do not validate the proposed architecture on a real system. Our work provides more insight regarding the effective capability of the streaming approach when scaling considerations (bandwidth, resource consumption, power efficiency) are taken into account.

The work of Sano et al. [2011] consists of replicating the architecture that executes one time-step a number of times putting them in cascade; that is, the output of one architecture is the input of the next. The hardware accelerator is a composition of soft processors that must be explicitly programmed; hence, it is a different approach with respect to the one proposed in this work, where the core computational element (Streaming Stencil Timestep (SST)) is a synthesized accelerator. This work is the first proposing a methodology to construct the queue using an algorithmic methodology and some concepts, such as the *queue looping condition*, which are completely novel. In Kobayashi [2013], 2D stencils are addressed using *ScalableCore*, a system composed of multiple, low-end FPGAs, connected in a 2D mesh. To efficiently exploit such an architecture, the stencil computation is tiled and each computational block is assigned to an FPGA. Their work proves how an FPGA custom architecture delivers an improved power efficiency over traditional computing devices. However, the isolation of computation on multiple computing elements effectively reduces data reuse by a factor proportional to the number of elements, which is a disappointing property when the goal is to scale out the system. Moreover, *ghost zones* must be properly synchronized among processing elements, leading to an unnecessary (as we show in Section 3) increase in on-chip bandwidth consumption.

In Shafiq et al. [2009], a memory architecture is developed to implement symmetric 3D stencils, that is, of the form of $n \times (n + 1) \times n$, which features First In First Out (FIFO) queues for input and output streams (one for each dimension); a *data engine* (the *front-end*), which prefetches data; a *compute engine* (the *back-end*), which consists of multiple instances of the computation unit; and a *control engine* responsible for synchronizing the flow of data in the whole architecture. While this work is interesting for the customized approach to 3D stencil computation, the applicability is limited to this class only; additionally, no explicit scaling mechanism is reported. In our work, we extend the applicability of our methodology to N-dimensional stencils and explicitly define a scaling methodology.

Maxeler [Pell and Huggett 2012] proposes an FPGA-based heterogeneous system, where the accelerator has to be implemented with a dataflow specification, that is, a Dataflow Engine (DFE). Maxeler's computing system includes Central Processing Units (CPUs) and DFEs, and DFEs' configurations are created using Maxeler's MaxCompiler. To create applications exploiting DFE configurations, an application must be explicitly split into three parts: the *kernel*, which implements the computational components of the application in hardware; the *manager configuration*, which connects kernels to the CPU, engine RAM, other kernels, and other DFEs via a custom interconnection (MaxRing); and the *CPU application*, which interacts with the dataflow engines to read and write data to the kernels and engine RAM. From an architectural point of view, the structure of our accelerator resembles the one obtainable with a Maxeler's DFE on the specific application domain of ISLs. However, a Maxeler's software designer must have profound knowledge of the semantic of Maxeler MaxJ, a DSL based on Java, and nevertheless deep knowledge of the general structure of a dataflow architecture, as it must specify the accelerator structure in an explicit way. Hence, there is a steep learning curve and a required expertise that is but easy to attain: being able to implement a complex program can be a hard and time-consuming task. On the other hand, our methodology is able to automatically extract the accelerator from the original ISL C code.

3. OVERVIEW OF THE PROPOSED METHODOLOGY

We propose a methodology to derive an hardware accelerator designed to target a specific Iterative Stencil Loop (ISL). The hardware accelerator can be viewed, at every level of granularity, as a composition of *independent* modules that communicate over

First In First Out (FIFO) channels and employ *blocking reads* and *writes* to manage the dataflow and ensure its correctness. In particular, we developed a **streaming architecture** that performs a single ISL time-step, which we called Streaming Stencil Time-step (SST) (the entire accelerator is represented by the composition of multiple SSTs in a *queue fashion*). We propose a *design automation flow* to **automate** the SST derivation and the queuing process.

This design automation flow is a two-step process. While it can be automated—as we show in the successive sections of the article—the process described in this article is still manual. The first step consists of deriving the microarchitecture that implements a single iteration—that is, a *time-step*—the SST. This can be viewed as the *basic building block* of the accelerator. The second part addresses how to actually build the complete accelerator, in which SSTs are arranged in a *queue* fashion, realizing an effective and explicit streaming, scalable mechanism. This step takes as input the ISL's Static Control Part (SCoP), written in an imperative form (e.g., in C/C++), and produces the corresponding accelerator. The proposed flow prepends to the aforementioned steps a *preprocessing* phase, in which the Reduced Static Control Parts (rSCoPs) are extracted. Both the preprocessing phase and the first macroblock heavily rely on the *polyhedral framework*, as most ISLs, due to their staticness and regular structure, can be viewed as a subset of the class of programs known as Static Affine Nested Loop Programs (SANLPs). We restrict ourselves to treat only the ISLs enjoying this property. An overview of the proposed flow can be seen in Figure 2.

Let us briefly describe the two macroblocks of the proposed design automation flow. The first macroblock is the *SST microarchitecture derivation* and is composed of the following parts:

—The first part performs the polyhedral analysis in order to extract a polyhedral Intermediate Representation (IR) of the input code, and also the corresponding Data Dependency Graph, which is crucial for the entire SST derivation process.
—The second part consists of an ad hoc manipulation of the obtained Data Dependency Graph in order to obtain the skeleton of the SST, which we call the *streaming-oriented graph*.
—After that, two concurrent phases take place. Both phases employ the polyhedral IR along with the streaming-oriented graph, and their function is to further characterize respectively the *memory system* and the *computing system* of the SST.

The result of this process is an IR of the derived SST, which is used to generate the code of the modules that will be synthesized via High-Level Synthesis (HLS).

The second macroblock is the *SSTs queuing*, which employs the estimated resource usage of an SST and the total amount of available resources in order to derive the maximum achievable queue length and generate the final Register-Transfer Level (RTL) of the resulting hardware accelerator.

It can happen that the ISL's SCoP contains conditionals that are *affine functions* of the outermost loop, which is indeed the time dimension. In this case, the code must be transformed beforehand, as a conditional on the time dimension means that only certain code parts are executed within a time-step; that is, code parts execute in a mutually exclusive manner, and when deriving the SST, this situation is unacceptable. To deal with this case, a solution is to apply the *index-set splitting* transformation along the outermost loop *only*. The affine conditionals can be used to effectively drive the splitting on the original loop nest, and after the transformation is performed it can be safely assumed that the conditionals can be removed from the obtained code. The result of this process is a series of loop nests, each one iterating over a subset of the original iteration vector of the time dimension. We call them rSCoP, since they actually still belong to a single SCoP, but despite this we treat each of them individually.

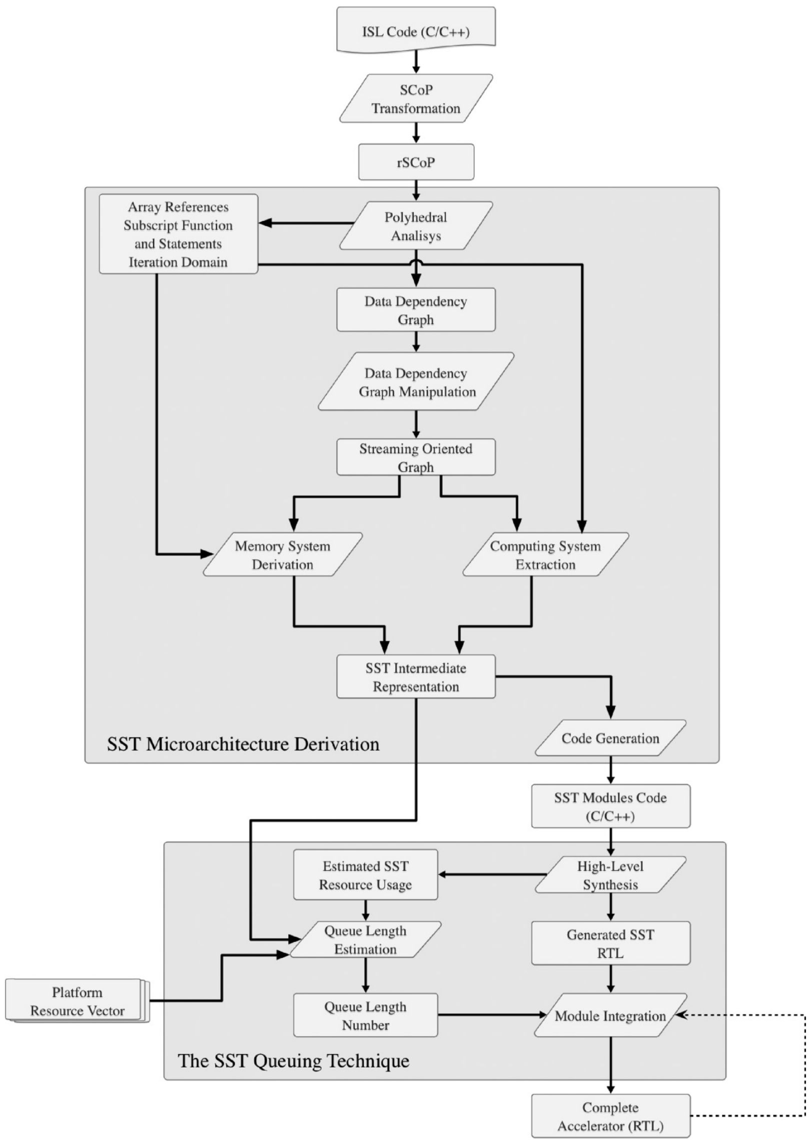
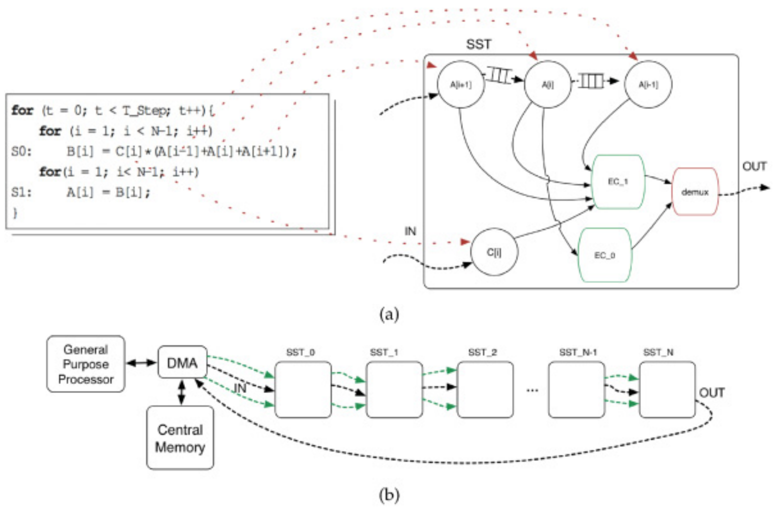


Fig. 2. The proposed design automation flow. All steps are described in Sections 4 and 5.

4. PROPOSED ARCHITECTURE: THE STREAMING STENCIL TIME-STEP

An SST is required to execute a single ISL time-step and has *in general* one input stream and one output stream. In the case in which the ISL updates grid points employing constants or other arrays, the input streams are more than one. The components within an SST can be divided into two main categories, the first being the *memory system*, the second being the *computing system*.

Memory System. The *Memory System* consists of a series of *chains* of modules connected via FIFO channels, one chain for every distinct input array, all responsible for feeding the computing architecture with the needed data. Each chain receives a single data stream—which is the array itself—and the modules within the chain represent



```
for (t = 0; t < T_Step; t++){
    for (i = 1; i < N-1; i++)
S0:      B[i] = C[i]*(A[i-1]+A[i]+A[i+1]);
    for(i = 1; i< N-1; i++)
S1:      A[i] = B[i];
}
```

Fig. 3. (a) An example of SST resulting from the sample code on the left. Each array relates and belongs to a chain, and each array reference corresponds to a new module inside this chain, as shown by the red arrow in figure. Each SST computes the code associated to a single, specific time-step. (b) The architectural template of the accelerator for an ISL with multiple inputs. The complete hardware accelerator is the composition of multiple SSTs as in (a) arranged in a queue fashion. The green arrows represent the additional streams. As described in the text, the last SST has only the actual output stream.

the different read array references. These modules are the ones actually responsible for sending the data, as in fact they read any existing data element from their preceding FIFO and send the data element to the following FIFO as well as to the computing system. From a high-level perspective, this arrangement can still be viewed as a single stream, from where each module filters data only when needed, which is why we called them *filters*. The chain-like organization of filters ensures that the data is read only once and at the same time allows more concurrent accesses, realizing also the Full Buffering.

Computing System. It is composed of a series of modules that perform the actual computation taking data from the memory system. Let us now provide some further details to better understand how they are arranged. Given the fact that in the case of ISLs there is always the presence of the boundary to take into account, there is the possibility of performance loss when hardware accelerators are employed, as the host processor could be forced to waste time to reconstruct the array from the output. For this reason, our SST considers the boundary in an explicit way. This is also a prerogative for the SSTs' queuing. In fact, since the accelerator consists of a chain of replicas of a single SST, within a single SST both the output and the input must be of the same form. To ensure that the output stream is rearranged in the exact same form of the input, we inserted a module—in addition to the modules that actually perform the computation—called *demux*, whose function is precisely to rearrange the output.

An SST has *in general* a single output stream. This, however, is not true when performing queuing with an ISL that takes multiple arrays as input (Figure 3). In fact, to provide the required data to all the SSTs within the queue, the SSTs are equipped with additional output streams, that is, additional communication channels from the involved *filters chains* toward the next SST in the queue, other than the one that drives the proper accelerator's output. This is not true for the last SST, which does not have the added output streams as they are not needed. As shown in Figure 2, the first part of the proposed design automation flow is the derivation of the SST from the input rSCoP. Let us now describe the different phases that take place within this macroblock.

4.1. Streaming-Oriented Graph Construction

The purpose of this first phase is to construct a graph, namely, a *streaming-oriented graph*, which will be used as a skeleton for the SST. Also, streaming architectures can be effectively represented as a graph; therefore, a graph-like representation of the SST microarchitecture fits perfectly with its streaming nature—and allows an easier manipulation and processing of the information during the SST derivation. The first task that must be performed is the *data dependency analysis*, in order to produce, for a given rSCoP, the corresponding Data Dependency Graph. The only dependencies that must be taken into account are the RAW dependencies, as an SST can be viewed as an *in-order* microarchitecture. After that, the Data Dependency Graph must be further pruned: the dependencies carried by the time dimension—that is, the outermost loop—*must* be discarded, since, as stated before, an SST is demanded to implement the execution of a *single* time-step. There is, however, a case in which an edge on the time dimension can remain after this pruning task. This situation can occur whenever the rSCoP loop nest is *imperfect*, because flow dependencies within the *same* time-step could be carried exactly along the time dimension. As a result, we can define the following conditions for an edge to be removed during the pruning task:

Definition 4.1 (*Data Dependency Graph Edge Removal Conditions*). Let us consider a Data Dependency Graph $G = (N, E)$, with only RAW dependencies, in which each node N is marked with an increasing number given by the execution order of each statement, which, by the way, in the case of an rSCoP corresponds also to the *syntactic order*. For the process of the *streaming-oriented graph* construction, an edge E must be removed *if it represents a dependence carried along the outermost loop (time dimension)*, and E is a self-loop or is directed from N_i to N_j and $i > j$.

After that, *each* node (i.e., a statement) of the Data Dependency Graph is *expanded* in the following way:

(1) Each array reference becomes a new node. Since for each array assignment—that is, statement—the polyhedral analysis (indeed, the parsing step taking place at the beginning) is able to identify the read and write operations, we employ this information to connect these nodes. Specifically, each *read* node of the given assignment will have an outgoing edge connected to the corresponding *write* node, as the write operation trivially depends on these data. Furthermore, read nodes with the same array reference are merged. Note that the write nodes symbolically represent the execution statement; hence, they are associated with the statement's assignment—that is, formula—and its iteration domain.
(2) The original edges of the Data Dependency Graph are now connected, rather than with the entire statement, to the specific node—that is, array reference—involved in the corresponding dependence.

The last step toward the construction of a *streaming-oriented graph* consists of an iterative removal of the "copy" dependencies, that is, an outgoing edge from the write node W of a statement entering another node N, such that:

—N has no outgoing edges entering back W, causing a cycle;
—if N is a read node, the corresponding write node must not refer to the same array as W;

—if N is a read node, its *data domain*—that is, the *image* of the corresponding statement iteration domain on the reference *subscript function*—matches the iteration domain of W;
—if N is a write node, its iteration domain is the same as that of W.

Notice that this reduction can only be made if the aforementioned write node W does not have other outgoing edges. This operation is described by the pseudocode of Algorithm 1.

ALGORITHM 1: Iterative Reduction of the *Streaming-Oriented* Graph

```
Input: the streaming-oriented graph $G = (N, E)$
Output: the reduced version of $G$
$R = 0$; foreach write node $N \in G$ do
    if $N$ has only one outgoing edge $E \wedge E$ is directed to a node $N'$ in a “copy” dependence relation then
        $R \leftarrow R \cup (N, E, N')$;
    end
end
while $R \neq 0$ do
    remove $r = (N, E, N')$ from $R$; substitute the reference to $N$ in $N'$ with $N$ assignment’s formula; create a new write node $W$ with the same formula of $N'$; substitute $r$ in $G$ with the single node $W$;
    if $W$ has only one outgoing edge $E \wedge E$ is directed to a node $W'$ in a “copy” dependence relation then
        $R \leftarrow R \cup (W, E, W')$
    end
end
```

The obtained graph is the skeleton of the SST microarchitecture, and we call it the *streaming-oriented graph*. In order to have the IR of an SST, from this graph both the *computing system* and the *memory system* must be properly characterized. In the following two subsections, we will explain the procedures to achieve this goal.

4.2. Computing System Extraction

The *streaming-oriented* graph provides us the write nodes that effectively become computation modules of the *computing system*. Since the streams entering the *filter chains* of the *memory system* must be in the form of the entire array, the boundaries must be explicitly managed. To do so:

(1) The array updated within the statement with the highest index—that is, the last in the syntactic order – must be the array updated by the ISL. We employ this information to explicitly identify the output array (stream) of an SST, as the one updated from the *last write node*.
(2) A *demux* D is instantiated and associated to the *last write node* L such that $L \rightarrow D$. How the boundary is actually managed is explained in Section 4.3.
(3) The streaming-oriented graph is traversed, and each node N whose statement updates the same array of L is also associated to D, that is, $N \rightarrow D$. If more than one write node is associated to the same demux, this means that there are different portions of the array that will be updated with different formulae. We call these portions *equivalence groups*. Since the write nodes are indeed responsible for the update of these regions of the array space, we will indistinctly refer to both regions and corresponding write nodes as *equivalence groups*. Therefore, we can introduce the following definition of equivalence group.

Definition 4.2 (*Equivalence Group*). An equivalence group is the maximal set P of points of a given array, updated within the ISL computation, such that each point of P has the same *update formula* and is dependent on the same set of *filters*.

(4) The process is repeated with all the remaining write nodes, whose array updates won't be forwarded as output of the SST. They will be only needed to ensure the correctness of the computation of an SST.

After the pruning task of the Data Dependency Graph previously described, the only kind of cyclic dependencies that can be present are the statement's self-dependencies. When deriving the streaming-oriented graph, this translates into cycles between a given write node and some of its input read nodes, indicating the presence of spatial dependencies between grid points. In this case, the write nodes involved in these cyclic dependencies further require manipulation. From now on we will refer to the write nodes that enjoy this characteristic as *cyclic-write* nodes, and the read nodes that concur in the cyclic dependency as *cyclic-read* nodes. First of all, we claim that these cyclic dependencies can only involve read nodes whose *subscript function* is not of the form $f(\vec{x}) = I\vec{x}$ (being $f(\vec{x}) = F\vec{x} + \vec{a}$, where the *subscript matrix* F is the *identity matrix* I, and $\vec{a} = 0$). This condition is trivially enforced by the fact that these dependencies are between subsequent integral points of the iteration domain. Therefore, a cyclic-read node *data space* (also called *data domain*)—that is, the *image* of the cyclic-write node iteration domain on the reference *subscript function*—will always partially overlap with the boundary. Hence, since part of the data space of the cyclic-read node will come from the boundary and part from the cyclic-write node's output—that is, two distinct data streams—the cyclic-read node will be actually implemented as two distinct filters, each belonging to a different chain. The result is that the iteration domain of each cyclic-write node is partitioned into subsets that are dependent on a different set of filters, even if the formula is actually the same. Hence, they are indeed different *equivalence groups*, as previously defined, even though we will refer to them as *sd-equivalence groups* (the prepended "sd" stands for *spatial dependence*) to differentiate them from the previously derived one. To partition the original cyclic-write node iteration domain, we need two basic pieces of information, namely, the iteration domain of the sd-equivalence groups—each of them being a partition of the original iteration domain—and the set of input filters of each sd-equivalence group. The extraction of this information requires a specific algorithm, whose pseudocode is presented in Algorithm 2. For the applicability of the algorithm, we need to be able to manipulate the polyhedra associated to the ISL under analysis. We do so in this work by employing the state-of-the-art library *isl* [Verdoolaege 2010], a library for manipulating sets and relations of integer points bounded by linear constraints. The result is the set of sd-equivalence groups with the associated input filters and iteration domains. Note that the read nodes that are not cyclic-read nodes are implicit inputs of each sd-equivalence group, as they are indeed implemented as a single filter. The complexity of the proposed algorithm is $O(n^2)$ in the worst case, where n is the number of cyclic-read nodes. However, in real-world cases, n is always small enough for the algorithm to terminate in a reasonable amount of time (e.g., in our benchmarks n has been no more than 4). It is also important to point out that this is, as far as we know, the first algorithm that enables us to automatically implement with an hardware accelerator ISLs with spatial dependencies between grid points.

4.3. Memory System Derivation

In order to characterize the memory system, the following steps have to be performed *for each write node*:

ALGORITHM 2: sd-Equivalence Group Extraction

```
Input: $I$: the iteration domain of the cyclic-write node $w$.
Input: $A$: set of cyclic-read nodes $a_i = (f_w, f_{nw})$, with a subscript function $f_{a_i}$.
$f_w$ represents the part of the data domain that overlaps with $I$, while $f_{nw}$ represents the part
that overlaps with the boundary.
Output: $E$: set of equivalence groups $e = (i_e, r_e)$, where $i_e$ is the iteration domain and $r_e$ the set
of input $f_w$.
$E \leftarrow 0$; $P \leftarrow 0$; {$P$ is the set of the preimage portion of each $a_i$ that overlaps with $I$}
foreach $a_i \in A$ do
    $D_{a_i} \leftarrow f_{a_i}$; $S \leftarrow D_{a,i} \cap I$; $p_{a_i} \leftarrow (f_{a_i}^{-1}(S),\ a_i)$ {the first element of the tuple is the preimage, the
    second is the identifier}; $P \leftarrow P \cup p_{a_i}$;
end
$i_0 \leftarrow \cap_i preImage(p_{a_i}),\ p_{a_i} \in P$; $E \leftarrow E \cup e_0 = (i_0, F = \{f_w(a_i) \mid \forall a_i \in A\})$;
$i_1 \leftarrow I - (\cup_i p_i,\ p_i \in P)$;
if $i_1 \neq 0$ then
    $E \leftarrow E \cup e_1 = (i_1, 0)$;
end
while $P \neq 0$ do
    $p_{a_j} \leftarrow firstElement(P)$; $Temp \leftarrow P - p_{a_j}$; $i_{new} \leftarrow preImage(p_{a_j})$; $r_{new} \leftarrow 0$;
    $r_{new} \leftarrow r_{new} \cup f_w(identifier(P_{a_i}))$;
    while $Temp \neq 0$ do
        $t_{a_k} \leftarrow firstElem(Temp)$; $Temp \leftarrow Temp - t_{a_k}$; $i_{old} \leftarrow i_{new}$; $i_{new} \leftarrow i_{new} \cap preImage(t_{a_k})$;
        if $i_{new} = 0$ then
            $i_{new} \leftarrow i_{old}$;
        else
            $r_{new} \leftarrow r_{new} \cup f_w(identifier(t_{a_k}))$;
        end
    end
    foreach $p_{a_i} \in P$ do
        $preImage(p_{a_i}) \leftarrow preImage(p_{a_i}) - i_{new}$;
        if $preImage(p_{a_i}) = 0$ then
            $P \leftarrow P - p_{a_i}$;
        end
    end
    $E \leftarrow E \cup e = (i_{new}, r_{new})$;
end
```

(1) The cyclic-read nodes—whenever present—are implemented as two different filters, for the previously described reasons in Section 4.2. The remaining read nodes will be implemented as a single filter. Filters are then clustered according to both the corresponding *array name* and the input stream—for example, whether it is the output of the write node or not—to obtain the so-called *chains*.

(2) Whenever a chain contains more than one filter, those filters are ordered from the lexicographic maximum to the lexicographic minimum. The input stream will enter the maximum and flow following the *reverse lexicographic order*, down to the minimum, which means that those filters are linked together by the input stream. The size of the communication channels is computed as the modulus of the *data distance vector* of a *fixed* and *common*—but nevertheless arbitrary—iteration between the subscript functions of the two filters. Following the principles of Cong et al. [2014], we can define the conditions for the proper structuring of a chain as follows:

Definition 4.3 (*Chain Structuring Conditions*). A chain of ordered filters $\{f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_n\}$ related to an array A must be compliant to the following two rules in order to have Full Buffering (FB) being also be deadlock-free:

—For every couple f_i and f_j such that $i < j$, then

$$f_i \succ_l f_j.$$

—The size W of a communication channel between a filter f_i with subscript function f_A^i and a filter f_j with subscript function f_A^j must be

$$W \geq \mid \delta(\nu, \nu)_{f_A^i f_A^j} \mid .$$

If W is minimal ($=$), the Full Buffering is also *optimal*.

(3) As stated before, there is also the boundary of the output array to consider. Within the chain referring to the same array updated by the write node (in the case in which the ISL has spatial dependencies, the chain considered is the one *that does not take as input stream the write node's output*), the filter—whenever present—whose data domain perfectly overlaps with the iteration domain, that is, for which the subscript function is $f(\vec{x}) = I\vec{x}$ (intuitively, this is the "central" node of the chain), will be the one demanded to route the boundary toward the *demux*. If this node is not present (i.e., the update of a grid point is performed without reading its previous value), it will be added and its functionality will only be to route the boundary.

(4) Each remaining communication channel, indeed, every channel within an SST except the one inside the chains, will be of size 1.

4.4. SST IR and Code Generation

The purpose of the previous phases was to extract from the input rSCoP all the information needed to enable the actual implementation of an SST as a hardware microarchitecture. At the end of these phases, the information is encoded in the form of an IR. The IR contains the following:

—For each filter: an identifier; its data domain, which is the filtering condition; the input and output streams, that is, the input and output communication channels
—For each equivalence group: an identifier; its iteration domain; the array update formula; the input communication channels, as well as the output one
—For each communication channel: an identifier; its minimum size

Information about the demux is not needed as its structure is inferred from the iteration domain of the associated equivalence groups. From the SST's IR, the hardware equivalent is generated employing HLS; hence, it is required to generate the code for each module of the microarchitecture: demuxes, equivalence groups, and filters.The communication channels will be implemented as FIFO queues. The module's code can be generated using state-of-the-art polyhedral model tools, integrated with the additional information we need in our case, such as read and write instructions on, respectively, input and output ports of the communication channels. Note how the boundary transfer is actually made with a single channel from the involved filter to the demux. No additional modules will be inserted in between, as they are indeed unnecessary.

4.5. Pipelining the SST

Pipelining the equivalence groups can be an effective optimization to increase the overall throughput. However, since an SST is a composition of independent modules, this optimization could lead to situations of deadlock. To avoid this situation, we propose two solutions:

—The communication channels between the memory system and any equivalence group could be oversized, thus allowing the memory system to proceed even if the equivalence groups are stalled. However, to compute the communication channel's size, the pipeline depth of each equivalence group must be known, something that in general is not easy to do. Also, this solution will cause the SST to enjoy no more the Full Buffering property.
—Only the innermost loop of each equivalence group is pipelined, which results in the flushing of the pipeline right when the memory system starts to send data to another equivalence group, completely avoiding the possibility of a deadlock within an SST.

There is a last, important, remark to be made. In the case of the presence of spatial dependencies, the related equivalence groups *cannot* be pipelined.

4.6. Scaling the Problem Size

Whenever the available on-chip memory is not large enough to allow the instantiation of all the communication channels, there is always the possibility of tackling the problem by trading bandwidth requirements for on-chip memory usage [Cong et al. 2014]. In practice, this means that there is always the possibility to remove the *largest* communication channel and replace it with an additional input data stream from the off-chip memory. The process could be repeated iteratively until the overall memory requirements are compatible with the available resources. Note that this tradeoff possibility is, however, limited by the available bandwidth.

4.7. Final Remarks

We introduced the notion of SST, the basic computation and memory block of the computing system, characterized by the full buffering (within a given time-step) property. Additionally, we designed the SSTs to allow their *chaining*, thus realizing a mechanism to explicitly scale the performance of the overall system by simply adding more SSTs to the chain itself. We do this taking as input C, a popular imperative language whose semantic is well known to most designers. Finally, we show in the following section how this approach realizes an energy-proportional computing system. To the best of our knowledge, this is the first work that defines an algorithmic methodology to design a custom accelerator for ISLs with an explicit focus on both scaling and bandwidth considerations.

5. THE SSTS QUEUING TECHNIQUE

The functionality of a Streaming Stencil Time-step (SST) is to perform the stencil computation associated to a single time-step. Hence, having a hardware accelerator built up of a single SST would mean that, in order to perform more time-steps, the same SST should be employed over and over again, transferring back and forth data from the off-chip memory to the accelerator itself. These frequent off-chip memory transfers can effectively bound the achievable performance, as an off-chip memory access is definitely much more expensive in terms of latency compared to data transfers within the accelerator. This is an issue already known in Iterative Stencil Loops (ISLs) literature, as their inherent memory boundedness is a major reason that obtaining high-performance codes and accelerators is in general an involved task. A possible solution to this problem could be to have a technique to limit as much as possible the off-chip memory transfers, exploiting the available hardware resources to offload not only the computation within a single time-step but also the data transfers across time-steps. Our SSTs *queuing* techniques go exactly in this direction.

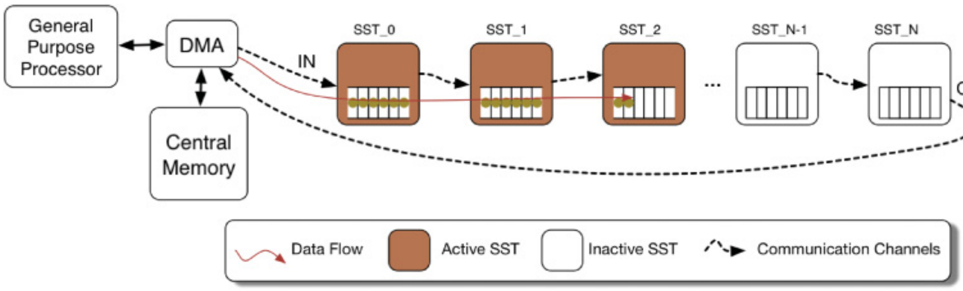


Fig. 4. A visualization of the pipelined execution within the queue.

The key point of the queuing technique is that multiple SSTs are arranged in a queue fashion, which means that, within the queue, the output of one SST is input to the next. Off-chip memory transfers occur *only* at the *beginning* and at the *end* of the queue. This implies that having a queue of arbitrary length or a single SST will involve the *same volume* of off-chip/on-chip memory transfers, which in turn means that the bandwidth requirements *will remain constant*, or more precisely, they are independent from the queue length. Therefore, the off-chip memory latency bottleneck is progressively alleviated as the queue length increases, since a greater volume of computation will be performed with the same off-chip bandwidth requirements.

The SSTs' dataflow can be managed by an additional module, which is always aware of the progress of the computation as well as the total number of time-steps to be performed, which we called *mux*. This happens in two cases (which can also occur together):

—The number of SSTs—and hence the corresponding number of time-steps—of the hardware accelerator is *not* an exact divider of the total number of iterations. In this case, the *mux* is responsible for breaking the computational flow when the total number of time-steps of the ISLs is performed and redirecting the output to the off-chip memory.

—The queue length is large enough to be able to cycle the dataflow, redirecting the output stream of the queue back to the queue itself instead of transferring it back to the off-chip memory. We call this condition the *queue looping condition*, which will be detailed in Section 5.1.1. In this case, the *mux* is responsible for breaking this loop when the total number of time-steps has been executed.

It is important to note that the streaming nature of a single SST allows one to have, at a certain point, all the SSTs concurrently processing on a "portion" of the stream, having therefore a *pipelined* computation within the queue. Also, the system is fully scalable, as the only constraint is the total amount of available resources. The fact that the achievable queue length is completely independent from the available bandwidth allows linear scalability of the size of the accelerator: the queue length can be enlarged until the available resources are saturated.

Furthermore, we claim that employing the SSTs' queuing will *speed up the Reduced Static Control Part (rSCoP) computation, and hence the throughput, by a pseudolinear factor*, dependent on the number of SSTs instantiated within the hardware accelerator, that is, the *queue length*. We provide a simple proof of this claim.

PROOF. Let us model the completion time C of a given ISL when using an SST as hardware accelerator. We take as reference the state of a single grid point between two subsequent time-steps, thus:

$$C = T \times (N \times (m_{in} + sst + m_{out})),$$

where $T \in \mathbb{N}$ is the total number of time-steps, $N \in \mathbb{N}$ is the total number of points to be updated, $m_{in} \in \mathbb{N}$ is the number of clock cycles a given point takes to be transferred from the off-chip memory to the hardware accelerator, $sst \in \mathbb{N}$ is the number of clock cycles spent from an SST to actually update it, and $m_{out} \in \mathbb{N}$ is the number of clock cycles it takes to be transferred back to the off-chip memory.

Then, if we employ *queuing*, with a queue of length $q \in \mathbb{N}$, the completion time C_q is

$$C_q = \frac{T}{q} \times (N \times (m_{in} + q \times sst + m_{out})).$$

Whenever T or N are large numbers,

$$T \times N \gg q \text{ and } T \times N \gg m_{in} + sst + m_{out},$$

it would mean that

$$T \times N \times (m_{in} + sst + m_{out}) \approx T \times N \times (m_{in} + q \times sst + m_{out});$$

hence:

$$C_q \approx \frac{C}{q}.$$

Remark 5.1. The speedup is pseudolinear because of the approximation $T \times N \times (m_{in} + sst + m_{out}) \approx T \times N \times (m_{in} + q \times sst + m_{out})$, which is, however, a reliable approximation given the fact that N and T are very large in real ISLs.

5.1.1. Queue Length Estimation. Within the proposed *design automation flow*, the queue length estimation is a process that takes as input:

—the estimated resource usage of an SST given from the high-performance systems (HLS), but also the resource usage of the communication channels, both platform dependent;
—the resource vector R_{max}, which represents all the available resources; and
—the total number t of time-steps of the rSCoP.

Given this information, the estimation process is represented by a simple division between R_{max} and the sum of the needed resources for both the communication channels and the SST, however limited in that the queue length q must be $q \leq t$. This limitation is nevertheless virtually nonexistent as, in general, ISLs are characterized by a very large number of time-steps. Interestingly, it should be noted that there is an analytical bound to the queue length, and therefore a maximum number Q_{max} of iterations to be queued. When this analytical bound is reached, the stream can then flow back again in the queue instead of being transferred back to the off-chip memory, thus reaching the *maximum* achievable speedup. We call the condition for which the hardware accelerator is able to perform all the iterations of the ISL *queue looping condition*.

Definition 5.2. *Q_{max} estimation*. An SST holds a fraction f of the sum of all the array involved in the computation, whose total size is S_A, hence:

$$f = \frac{S_A}{k}.$$

Therefore, the number of SSTs to be queued in order to perform the entire computation is

$$Q_{max} = \min\{q \mid \sum_{q} f > S_A\,,\ q \in \mathbb{N}\}.$$

Lastly, we recall, as shown in Figure 2, that the actual implementation of the hardware accelerator *could be* an iterative process, since the estimated queue length may be too high to be able to instantiate the accelerator whenever the available resources are not enough. This is indeed a platform-related situation, as it depends on the accuracy of the resource estimation provided by the specific HLS tool. A simple solution could be to iteratively decrement the queue length (Figure 2) until the accelerator fits into the available resources.

5.1.2. Handling More Than One Input. The solution to handle more than one input array when performing the SSTs' queuing has already been described in Section 4.1. The only detail that must be added is that in such a case, the HLS must produce two different versions of the SST: one with the added streams and one with only the actual output. The filter within a chain demanded to forward the data will be the first. The way in which they are arranged is determined within the module integration phase.

6. EXPERIMENTAL RESULTS

In order to test the proposed methodology and resulting hardware accelerator, we select a number of significant Iterative Stencil Loop (ISL) benchmarks, manipulate the code as described in Section 3, and generate the resulting system using Xilinx Vivado Design Suite. The Streaming Stencil Time-step (SST) architecture derivation has been aided by state-of-the-art polyhedral analysis tools. Specifically, we employ the following components:

—*Clan (Chunky Loop ANalyzer)*, to extract a polyhedral Intermediate Representation (IR) from the source code
—*Candl (Chunky ANalyzer for Dependencies in Loops)*, to compute polyhedral dependencies, and thus the corresponding Data Dependency Graph, from the polyhedral IR

The SST's modules have been implemented using *Vivado HLS* (v2014.3.1). Both SST modules' integration and queuing have been performed using the *Vivado* (v2014.3.1) toolchain, which is also employed to synthesize and implement the resulting RTL. Synthesis and implementation have been performed with an Intel Core i7-3630QM, featuring an 8GB DDR3 RAM. These specifications allowed us to push queuing only to a fraction of the total available resources, as we systematically ran out of memory during placement and routing with larger designs (i.e., with more SSTs enqueued). All the tests have been performed on a single VC707 board, which comes with a *Virtex-7* XC7VX485T Xilinx Field-programmable Gate Arrays (FPGA) chip. Along with other resources, the board features 1GB of DDR3 DRAM, which we also employed in our tests as reference off-chip memory.

6.1. Test Cases

Let us now focus on the benchmarks selected to validate both the methodology and the accelerator.

—*jacobi2D*: the fundamental computational kernel of 2D PDE solvers, and due to its nature it is well suited to perform SST queuing. As previously stated, our limited computational resources forced us to limit the queue length in order to successfully complete the accelerator synthesis and implementation. However, with *jacobi2D*, we were still able to push queuing up to a considerable number of SSTs without running out of memory during the synthesis process.
—*jacobi3D*: a 3-dimensional version of the previous benchmark.
—*seidel2D*: this ISL contains spatial dependencies between grid point updates; hence, it has no trivial and/or explicit parallelization opportunities. This is usually the most

complex kernel for automatic tools to analyze and is relevant to test how well our methodology performs in the worst case (from the standpoint of data dependencies).
—*3D31pt*: a 31-point 3D, compute- and memory-intensive ISL with variable coefficients, and thus multiple input arrays. This kernel is employed in different applications and relates to 3D field solvers.
—*heat3D*: a discretized 3D heat equation stencil with nonperiodic boundary conditions.
—*3D7pt*: a 3D 7-point stencil [Datta 2009] from the Berkeley autotuner framework.

All the benchmarks have been implemented using single-precision *floating-point* data types, both in hardware and Central Processing Unit (CPU).

6.2. Experimental Settings and Goals

We explored multiple operating frequencies during synthesis of the hardware accelerator. We could systematically synthesize without running into timing closure issues and/or out of memory exceptions at *200MHz*, which is our target frequency for the subsequent experiments. The data path toward the off-chip memory is 32 bits wide, and the frequency 200MHz, so that the available bandwidth is 800MB/s.

The experiments on the CPU side were conducted on an *Intel Xeon E5-1410*, a quad-core processor running at 2.8GHz, with a peak performance of 179.2 GFLOPS. The benchmarks were compiled using Pluto [Bondhugula et al. 2008] with diamond tiling [Bandishti et al. 2012] activated, a state-of-the-art technique for stencil optimization. For each of them we compiled both the original version and a diamond-tiled version running on eight threads, where the optimal tile sizes have been determined empirically with a limited amount of search. Notice that for *seidel2D*, Pluto was not able to compile a diamond-tiled version, as its inherent sequentiality makes it not suitable parallelization-oriented optimizations.

Our goal is to demonstrate different aspects of our design:

—The efficient usage of the on-chip memory resources realized by an SST allows one to treat problem sizes whose implementation would otherwise not be possible via direct synthesis or with trivial manipulation of the original source code via High-Level Synthesis (HLS).
—The scalability given by the SSTs' queuing ensures a pseudolinear increase in throughput while keeping the off-chip bandwidth constant.
—Improved power efficiency with respect to a general-purpose processor, specifically when scaling out the design by queuing multiple SSTs.

6.3. Resource Usage

In this section, we show the resource usage for all the benchmarks, expressed as a percentage of the total available resources (Figure 5). First of all, note how none of the benchmarks could be either directly synthesized via HLS or synthesized without heavy hardware-oriented code restructuring (by also applying nontrivial codesign considerations).

We have been able to synthesize successfully—that is, without running out of memory during synthesis—48 SSTs for *jacobi2D*. For *3D31pt*, we could not synthesize more than four SSTs without running out of memory during the synthesis process on our machine, while for *3D7pt*, *jacobi3D,* and *heat3D,* we stopped at eight SSTs for analogous reasons. *Seidel2D* has spatial dependencies within point updates, and thus a pipelined version cannot be obtained. Hence, only a no-pipeline version has been tested. We remark that in this case we were able to achieve, without running out of memory during synthesis, a queue length of 10 SSTs. For *3D31pt,* we provide in Figure 6 the floorplans of the 1 and 4 SSTs' designs to visualize resource consumption and energy proportionality.

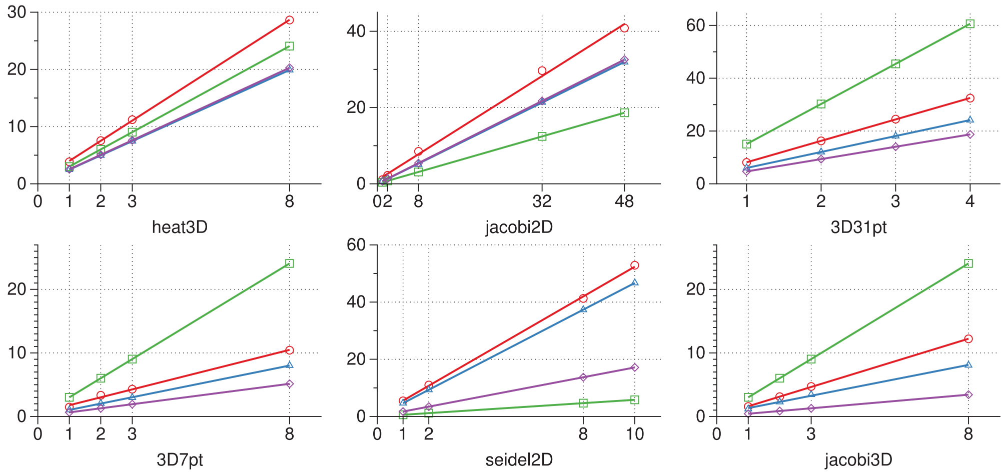


Fig. 5. Resource usage of the accelerator for all the benchmarks. Legend: Flip Flops (FFs) (circles), Look Up Tables (LUTs) (triangles), Block RAMs (BRAMs) (squares), Digital Signal Processing Blocks (DSPs) (diamonds). Linear fits exhibit an R^2 coefficient always greater than 0.99. We recall that R^2 is the *coefficient of determination*, a coefficient ranging from 0 to 1 that indicates how well data fit the prediction model, which in this case is linear.

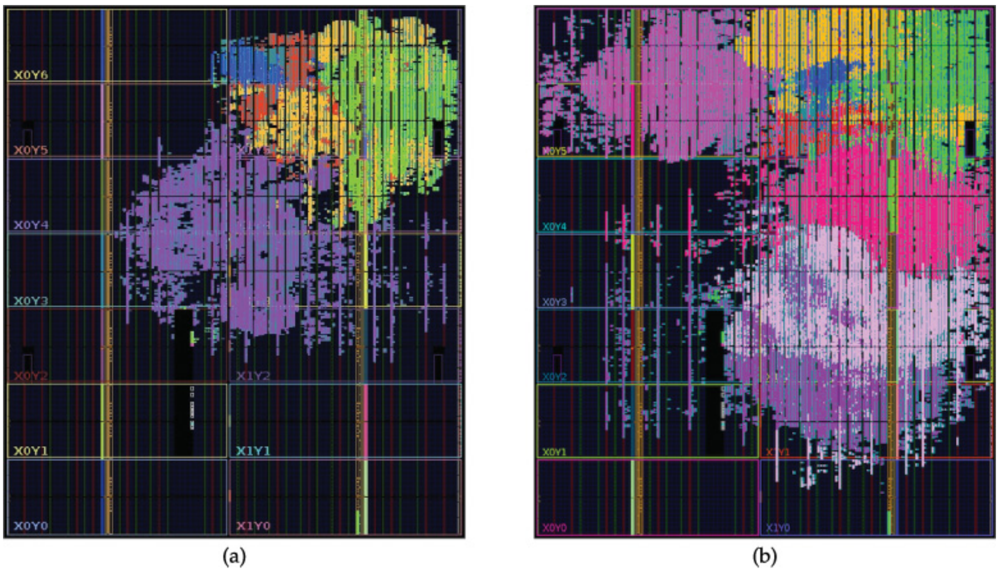


Fig. 6. Floorplan of (a) the 1 SST and (b) 4 SSTs queued designs implementing *3D31pt*. In blue, the Microblaze used as a host CPU to validate the accelerator. In green, the DDR3 controller. In red, the DMA. In ocher, the AXI interconnect of the design. In lily, violet, purple, and pink, the various SSTs. Observe how the relatively compactness of each accelerator allows the design to scale in throughput.

As we show in Figure 5, our methodology allows one to consume an amount of resources proportional to the number of SSTs enqueued, with a correspondingly proportional increase in throughput. Thus, data clearly shows that our methodology is *energy proportional*, a very desirable feature of systems in general and *heterogeneous systems* in particular.

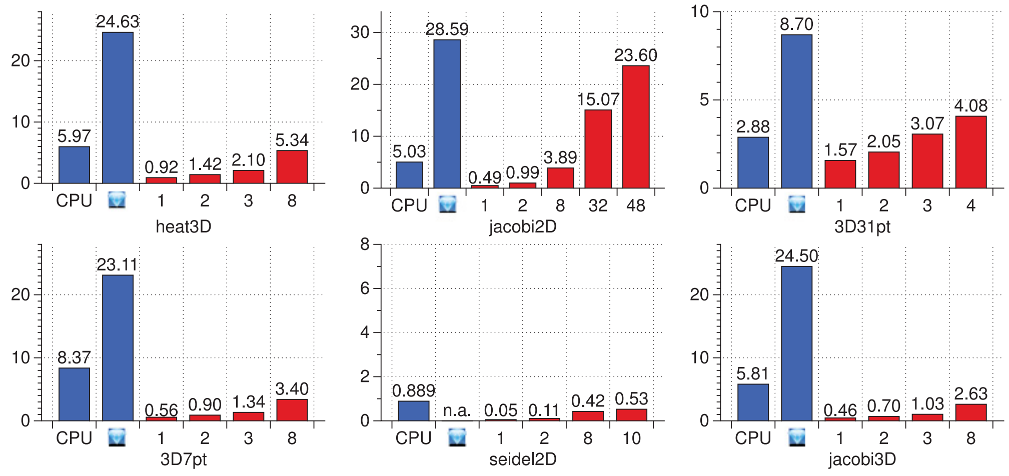


Fig. 7. Throughput, in GFLOPS. CPU refers to the Intel Xeon E5-1410. “Diamond” still refers to the throughput obtained on the Xeon but running an eight-threaded version of the benchmarks, compiled using Pluto with diamond tiling activated. Finally, the other categories indicate the number of enqueued SSTs.

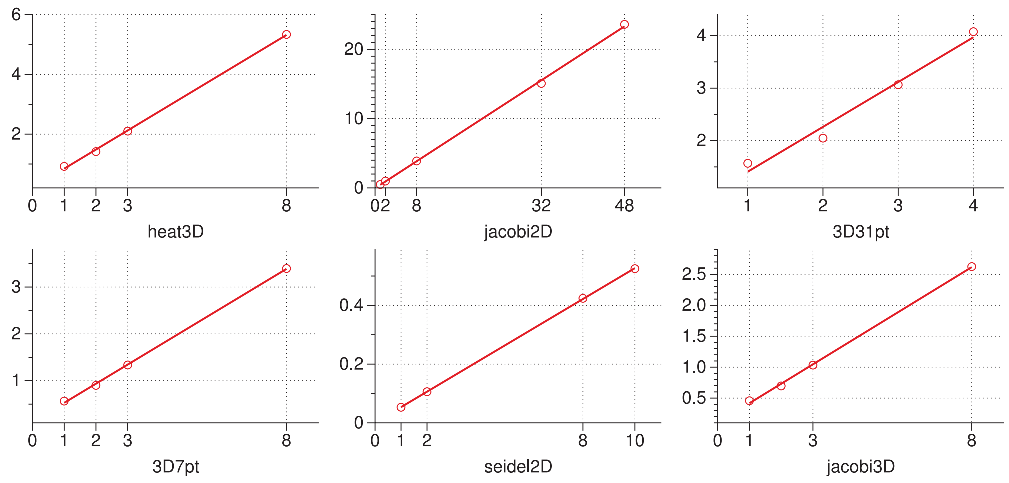


Fig. 8. Throughput, in GFLOPS; hardware accelerators only. The number on the x-axis represents the number of enqueued SSTs. Also in this case, linear fits exhibit an R^2 coefficient always greater than 0.99.

6.4. Performance

The performance of all the benchmarks is reported in Figure 7. Notice how, in all cases, the enlargement of the queue of SSTs corresponds to a pseudolinear throughput speedup (as in Figure 8). Hence, data shows how our approach scales *by design*. It is also important to remark that the performance comparison between the proposed solution and the CPU must only be intended to provide at a glance an insight of the potential of our accelerator. As previously stated, the embedded system employed for validation was able to deliver a maximum bandwidth of only 800MB/s, which is much less than the available bandwidth on the CPU. Future work will address this issue focusing on the design of a much more efficient underlying system, able to deliver sustained bandwidth and interface our hardware accelerator with a proper host processor—probably employing PCI-Express as communication bus.

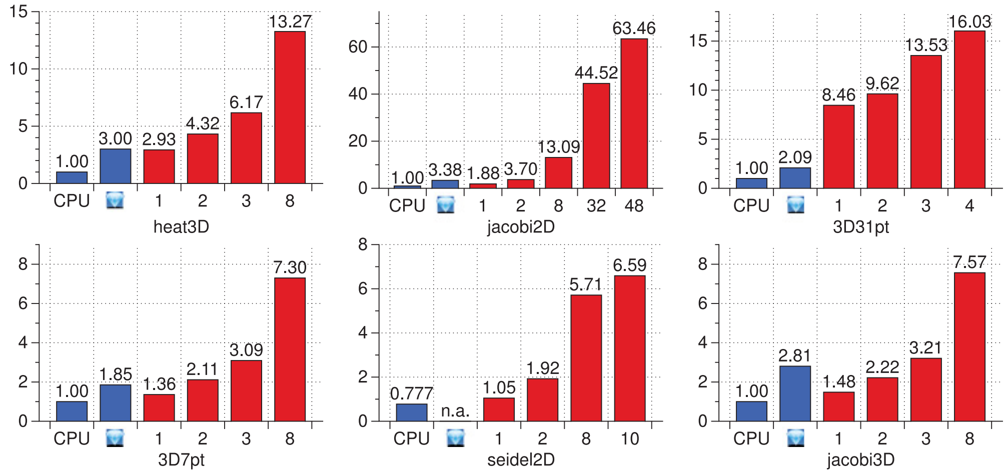


Fig. 9. Power efficiency ratio, normalized to the CPU. CPU refers to the Intel Xeon E5-1410. "Diamond" still refers to the Xeon but running an eight-threaded version of the benchmarks, compiled using Pluto with diamond tiling activated. Finally, the other categories indicate the number of enqueued SSTs.

6.5. Power Efficiency

Figure 9 shows the power efficiency normalized to the CPU. We measure power consumption at the wall, using a watt meter. We claim that this methodology is sufficiently accurate for this experiment as the CPU spends most of the power on a compute-intensive workload—CPU and RAM accesses, specifically. For all benchmarks, our final accelerator has a better power efficiency than the CPUs. This is in line with state-of-the-art works; however, our methodology allows us to push the utilization of the target device, and thus to push the power efficiency to the limit imposed by the algorithm, device, and synthesis process (which all contribute to setting a hard limit to the power efficiency of a device). Data show that our methodology not only is *energy proportional* but also allows the system to push the limits of the system's power efficiency, again a desirable feature when thinking about larger, scaled out systems.

7. CONCLUSIONS AND FUTURE WORK

In this work, we proposed a design automation flow to accelerate a target Iterative Stencil Loop (ISL) ona Field-Programmable Gate Array (FPGA), consisting of a queue of architectures demanded to perform a single ISL time-step, the Streaming Stencil Timestep (SST). We efficiently exploit the available resources realizing a Full Buffering and ensure a pseudolinear throughput speedup when queuing multiple SSTs, taking into account both memory and bandwidth considerations. We automatically derive the accelerator from the original source code, employing the polyhedral model in combination with High-Level Synthesis (HLS). Experimental results show an efficient usage of the on-chip memory resources realized by an SST, allowing one to deal with problem sizes that would otherwise be intractable with a direct synthesis of the original code via HLS. We also show that the SSTs' queuing technique ensures a pseudolinear increase in throughput obtained with constant bandwidth requirements. Also, the comparison shows that the proposed accelerator has the potential to outperform all comparable solutions thanks to its inherent scalability and specialization of the algorithm with respect to ISLs; overall power efficiency rivals the currently available top power-efficient systems and is expected to grow with the increase of the number of SSTs within the queue (i.e., with the increase of logic utilization).

As future work, we envision the implementation of an automatic tool chain realizing this design flow, as we demonstrated its feasibility. Additionally, we envision further validation of the scaling claims of the SST architecture and methodology in a multi-FPGA environment.

ACKNOWLEDGMENTS

The authors would like to thank Xilinx Inc. and Cathal McCabe, specifically, for the precious collaboration and for the donation of the VC707 boards employed to collect the experiments.

REFERENCES

C. Alias, B. Pasca, and A. Plesco. 2012. FPGA-specific synthesis of loop-nests with pipelined computational cores. *Microprocess. Microsyst.* 36, 8 (Nov. 2012), 606–619. DOI:http://dx.doi.org/10.1016/j.micpro.2012.06.009

F. Arandiga, A. Cohen, R. Donat, and B. Matei. 2010. Edge detection insensitive to changes of illumination in the image. *Image Vision Comput.* 28, 4 (2010), 553–562. DOI:http://dx.doi.org/10.1016/j.imavis.2009.09.002

C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. 2010. CλaSH: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*. 714–721. DOI:http://dx.doi.org/10.1109/DSD.2010.21

V. Bandishti, I. Pananilath, and U. Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 40. DOI:http://dx.doi.org/10.1109/sc.2012.107

R. Bleck, C. Rooth, D. Hu, and L. T. Smith. 1992. Salinity-driven thermocline transients in a wind- and thermohaline-forced isopycnic coordinate model of the north atlantic. *J. Phys. Oceanogr.* 22, 12 (Dec. 1992), 1486–1505. DOI:http://dx.doi.org/10.1175/1520-0485(1992)022\%3C1486:sdttia\%3E2.0.co;2

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. 1995. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.* 30, 8 (Aug. 1995), 207–216. DOI:http://dx.doi.org/10.1145/209937.209958

U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 101–113. DOI:http://dx.doi.org/10.1145/1375581.1375595

B.-Y. Cao and R.-Y. Dong. 2012. Nonequilibrium molecular dynamics simulation of shear viscosity by a uniform momentum source-and-sink scheme. *J. Comput. Phys.* 231, 16 (June 2012), 5306–5316. DOI:http://dx.doi.org/10.1016/j.jcp.2012.04.017

T. M. Chipeperekwa. 2013. *Caracal: Unrolling Memory Bound Stencils*. Technical Report. Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093 USA.

M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 676–687. DOI:http://dx.doi.org/10.1109/IPDPS.2011.70

J. Cong, P. Li, B. Xiao, and P. Zhang. 2014. 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)*. ACM, New York, NY, Article 77, 6 pages. DOI:http://dx.doi.org/10.1145/2593069.2593090

K. Datta. 2009. *Auto-Tuning Stencil Codes for Cache-Based Multicore Platforms*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

H. Ding and C. Shu. 2006. A stencil adaptive algorithm for finite difference solution of incompressible viscous flows. *J. Comput. Phys.* 214, 1 (May 2006), 397–420. DOI:http://dx.doi.org/10.1016/j.jcp.2005.09.021

M. Frigo and V. Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, New York, NY, 361–366. DOI:http://dx.doi.org/10.1145/1088149.1088197

M. Frigo and V. Strumpen. 2007. The memory behavior of cache oblivious stencil computations. *J. Supercomput.* 39, 2 (2007), 93–112. DOI:http://dx.doi.org/10.1007/s11227-007-0111-y

R. M. Haralick and L. G. Shapiro. 1992. *Computer and Robot Vision*. Addison-Wesley Longman, Boston, MA.

J. Holewinski, L.-N. Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, NY, 311–320. DOI:http://dx.doi.org/10.1145/2304576.2304619

S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. 2005. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 Workshop on Memory System Performance (MSP'05)*. ACM, New York, NY, 36–43. DOI:http://dx.doi.org/10.1145/1111583.1111589

M. S. Kim and K. Shimada. 2006. *Geometric Modeling and Processing - GMP 2006: Proceedings of the 4th International Conference (GMP'06)*. Springer.

R. Kobayashi. 2013. *The 100-FPGA Stencil Computation Accelerator*. Master's thesis. Tokyo Institute of Technology, Department of Computer Science, Graduate School of Information Science and Engineering.

S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 235–244. DOI:http://dx.doi.org/10.1145/1250734.1250761

C. Lengauer, S. Apel, M. Bolten, A. Großlinger, F. Hanning, H. Kölster, U. Rude, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt. 2014. ExaStencils: Advanced stencil-code engineering. In *Euro-Par 2014: Parallel Processing Workshops*. Lecture Notes in Computer Science. Springer Science, 553–564. DOI:http://dx.doi.org/10.1007/978-3-319-14313-2_47

Z. Li and Y. Song. 2004. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.* 26, 6 (Nov. 2004), 975–1028. DOI:http://dx.doi.org/10.1145/1034774.1034777

X. Liang, J. Jean, and K. Tomko. 2001. Data buffering and allocation in mapping generalized template matching on reconfigurable systems. *J. Supercomput.* 19, 1 (May 2001), 77–91. DOI:http://dx.doi.org/10.1023/A:1011196613858

J. Marshall, A. Adcroft, C. Hill, L. Perelman, and C. Heisey. 1997. A finite-volume, incompressible navier-stokes model for studies of the ocean on parallel computers. *J. Geophys. Res* 102 (1997), 5733–5752. DOI:http://dx.doi.org/10.1029/96jc02775

J. Meng and K. Skadron. 2009. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM, New York, NY, 256–265. DOI:http://dx.doi.org/10.1145/1542275.1542313

A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza. 2013. 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)*. ACM, New York, NY, Article 52, 6 pages. DOI:http://dx.doi.org/10.1145/2463209.2488797

A. Nakano, R. K. Kalia, and P. Vashishta. 1994. Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Comput. Phys. Commun.* 83, 2 (1994), 197–214.

J. V. Neumann. 1966. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL.

X. Niu, J. G. F. Coutinho, and W. Luk. 2013. A scalable design approach for stencil computation on reconfigurable clusters. In *Field Programmable Logic and Applications (FPL'13)*. 1–4.

O. Pell and J. Huggett. 2012. Method and apparatus for designing and generating a stream processor. (Dec. 27 2012). http://www.google.com/patents/US20120330638 US Patent App. 13/166,565.

S. M. F. Rahman, Q. Yi, and A. Qasem. 2011. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF'11)*. ACM, New York, NY, Article 30, 10 pages. DOI:http://dx.doi.org/10.1145/2016604.2016641

L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye. 2007. Towards optimal multi-level tiling for stencil computations. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. Institute of Electrical & Electronics Engineers (IEEE). DOI:http://dx.doi.org/10.1109/ipdps.2007.370291

F. Ritcher, M. Schmidt, and D. Fey. 2012. A configurable VHDL template for parallelization of 3D stencil codes on FPGAs. In *Proceedings of the European Regional Science Association Conference 2012 (ERSA'12)*.

J. O. A. Robertsson. 2012. *Numerical Modeling of Seismic Wave Propagation: Gridded Two-Way Wave-Equation Methods*. Society of Exploration Geophysicists, the International Society of Applied Geophysics.

D. K. Salkuyeh. 2007. Generalized Jacobi and Gauss-Seidel methods for solving linear system of equations. *Numer. Math., a Journal of Chinese Universities* 16, 2 (2007), 164–170.

K. Sano, Y. Hatsuda, and S. Yamamoto. 2011. Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In *Field-Programmable Custom Computing Machines (FCCM'11)*. 234–241.

K. Sano, Y. Hatsuda, and S. Yamamoto. 2014. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems,* 25, 3 (March 2014), 695–705. DOI:http://dx.doi.org/10.1109/TPDS.2013.51

K. Sano, S. Yamamoto, and Y. Hatsuda. 2011. Domain-specific programmable design of scalable streaming-array for power-efficient stencil computation. *SIGARCH Comput. Archit. News* 39, 4 (Dec. 2011), 44–49. DOI:http://dx.doi.org/10.1145/2082156.2082168

N. Satofuka. 2001. *Computational Fluid Dynamics 2000: Proceedings of the 1st International Conference on Computational Fluid Dynamics, (ICCFD'01), Kioto, Japan, July 10-14, 2000, Edited by Nobuyuki Satofuka*. Springer, Berlin. https://books.google.it/books?id=Iohv7GR3cIEC.

A. Schäfer and D. Fey. 2011. High performance stencil code algorithms for GPGPUs. *Procedia Comput. Sci.* 4 (2011), 2027–2036. DOI:http://dx.doi.org/10.1016/j.procs.2011.04.221

M. Shafiq, M. Pericàs, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguadè. 2009. Exploiting memory customization in FPGA for 3D stencil computations. In *Field-Programmable Technology (FTP'09)*. 38–45. DOI:http://dx.doi.org/10.1109/fpt.2009.5377644

L. G. Shapiro and G. C. Stockman. 2001. *Computer Vision*. Prentice Hall.

R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. 2011. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. IEEE Computer Society, 571–581. DOI:http://dx.doi.org/10.1109/icpp.2011.47

A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. 2014. DeLite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134 (April 2014), 25 pages. DOI:http://dx.doi.org/10.1145/2584665

Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. ACM, New York, NY, 117–128. DOI:http://dx.doi.org/10.1145/1989493.1989508

J. Treibig, G. Wellein, and G. Hager. 2011. Efficient multicore-aware parallelization strategies for iterative stencil computations. *J. Comput. Sci.* 2, 2 (may 2011), 130–137. DOI:http://dx.doi.org/10.1016/j.jocs.2011.01.010

S. Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *Proceedings of the 3rd International Congress Conference on Mathematical Software (ICMS'10)*. Springer-Verlag, Berlin, 299–302. http://dl.acm.org/citation.cfm?id=1888390.1888455.

C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. 1999. Memory characteristics of iterative methods. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC'99)*. ACM, New York, NY, Article 31. DOI:http://dx.doi.org/10.1145/331532.331563

G. Wellein. 2009. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'09)*. 579–586. DOI:http://dx.doi.org/10.1109/compsac.2009.82

R. Wester and J. Kuper. 2014. Deriving stencil hardware accelerators from a single higher-order function. In *Communicating Process Architectures 2014 (CPA'14)*.

M. E. Wolf and M. S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.* 2, 4 (Oct. 1991), 452–471. DOI:http://dx.doi.org/10.1109/71.97902

D. Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS'00)*. IEEE Computer Society, 171. DOI:http://dx.doi.org/10.1109/ipdps.2000.845979

Y. Zhang and F. Mueller. 2012. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 155–164. DOI:http://dx.doi.org/10.1145/2259016.2259037

W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. 2013. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*. ACM, New York, NY, 9–18. DOI:http://dx.doi.org/10.1145/2435264.2435271