Dynamic Stencil: Effective Exploitation of Run-time Resources in Reconfigurable Clusters

Xinyu Niu*, Jose G. F. Coutinho*, Yu Wang†, Wayne Luk*
* Dept. of Computing, School of Engineering, Imperial College London, UK
† Department of Electronic Engineering, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China

***Abstract*—Computing nodes in reconfigurable clusters are occupied and released by applications during their execution. At compile time, application developers are not aware of the amount of resources available at run time. Dynamic Stencil is an approach that optimises stencil applications by constructing scalable designs which can adapt to available run-time resources in a reconfigurable cluster. This approach has three stages: compile-time optimisation, run-time initialisation, and run-time scaling, and can be used in developing effective servers for stencil computation. Reverse-Time Migration, a high-performance stencil application, is developed with the proposed approach. Experimental results show that high throughput and significant resource utilisation can be achieved with Dynamic Stencil designs, which can dynamically scale into nodes becoming available during their execution. When statically optimised and initialised, the Dynamic Stencil design is 1.8 to 88 times faster and 1.7 to 92 times more power efficient than reference CPU, GPU, MaxGenFD, Blue Gene/P, Blue Gene/Q and Cray XK6 designs; when dynamically scaled, resource utilisation of the design reaches 91%, which is 1.8 to 2.3 times higher than their static counterparts.**

I. Introduction

The last few years have given rise to large computer infrastructures, such as clusters and data-centres, that provide ample compute resources. Sharing resources in a cluster where applications can be launched adds complexity to the development process: applications must not only efficiently exploit a given set of compute resources, but also adapt dynamically to available resources at run time. In a reconfigurable cluster with nodes consisting of different FPGAs, the heterogeneous FPGA nodes are used and released by various computational tasks at different points in time. More specifically, for a given design, throughput can be potentially increased if more resources are available to perform its computation. However, the effectiveness of current static design methods is limited by unpredictable run-time conditions. Due to non-deterministic starting points of applications, node availability and the amount of computational resources in available nodes are unknown during compile time.

The basic idea of this paper is illustrated with the following motivating example (Figure 1). In a reconfigurable cluster, 4 FPGA nodes A, B, C and D are released by other applications at time 0, 2, 3 and 4, respectively; node A, B and D possess 1 resource unit and process 1 data unit per second, while node C can process two data units per second; an application with 8 data units to process is launched into the cluster. Linear scalability is assumed for executed tasks, i.e., execution time is halved if the number of utilised resource units doubles. In this scenario, two static designs are illustrated in Figure 1. The OneNode Design will make use of only one node, so would take 8 seconds to complete. The FourNode Design will take all 4 nodes when all of them become available at time 4, and would take 2 seconds to complete. Only half of the computational capacity in node C is utilised, as the FourNode Design pre-defines that one resource unit is used in each run-time node. The Dynamic Design, in contrast, can start at time 0 when node A becomes available; then at time 2, after node A processes two data units, node B becomes available too, so both nodes process another two data units in the next second. At time 3 node A, B and C are available, completing the processing of the 4 remaining data units.

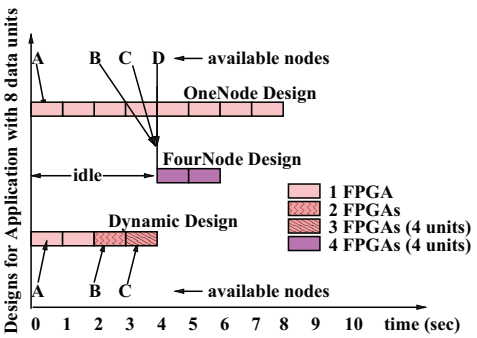

Fig. 1. Execution of various designs in the cluster, when node A, B, C, and D are released. The execution time of three designs (OneNode, FourNode and Dynamic) for the same applications is presented.

In a reconfigurable cluster with various computing nodes, challenges for developing such dynamic designs include: (1) to generate optimised designs that exploit intra-node resources in the cluster; (2) to construct initial designs when applications are mapped into the cluster, which ensures scalable performance for utilised nodes and correct functionality for the applications; (3) to adapt designs to run-time resource variations.

For parallel applications without inter-processor communication such as Monte-Carlo Simulation, the run-time solution is easy to generate as there is no communication operation. For communication intensive applications, challenges (1), (2) and (3) described above become tightly coupled: communication is cycle-accurately scheduled to overlap with computation operations; variations in device processing capacity, device-level parallelism and distributed workload bring deviation in the scheduled timing; if design configurations are not properly updated, incorrect results can be generated after the dynamic design scales into new nodes. In this work, stencil computation, known to be communicationally intensive and difficult to parallelise, is used as a case study for the proposed approach. Contributions of this work include:

- a novel design approach named Dynamic Stencil, which exploits various intra-node resources with compile-time

978-1-4799-2198-0/13/$31.00 ©2013 IEEE

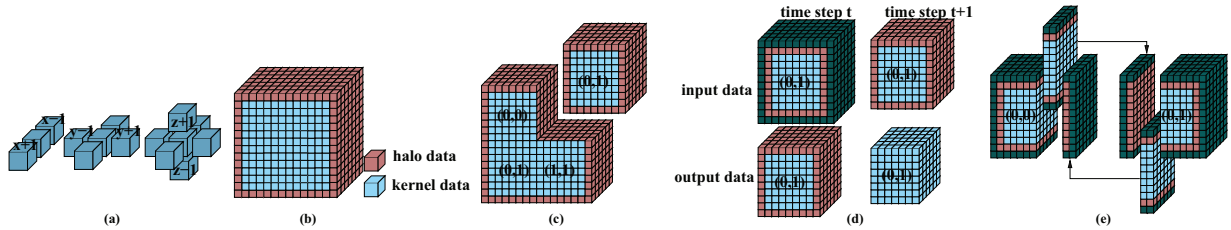

Fig. 2. (a) 1-D, 2-D and 3-D stencil in space; (b) Halo data within space; (c) Stencil data after spatial blocking [1]; (d) Input and output results for time step t and t+1 [2]; (e) Data exchange between neighbouring blocks [3].

optimisation, and utilises run-time resources with run-time initialisation and scaling. These three design stages cooperate with each other to ensure high resource utilisation for stencil applications in reconfigurable clusters;

- an asynchronous communication model that schedules communication operations to eliminate communication overhead based on: intra-node computational capacity, inter-node communication bandwidth, and available nodes. The communication model can be dynamically updated to ensure linear scalability as well as correct functionality when a Dynamic Stencil design scales;
- a run-time performance model that provides rapid evaluation of benefits and overhead for scaling current Dynamic Stencil design into FPGAs provisioned during run time.

II. Background

A. Stencil Computation

Stencil computation refers to a class of iterative operations to update array data with a fixed pattern, named as a stencil. Stencil computations are commonly used in simulating dynamic systems, such as fluid dynamics and heat diffusion, as well as in solving Partial Differential Equations (PDEs). As an example, to capture dynamic properties within target systems, a PDE can be formulated as follows:

$$A\frac{\partial^2 f(s,t)}{\partial t^2} = B\frac{\partial^2 f(s,t)}{\partial s^2} + C\frac{\partial f(s,t)}{\partial s} \qquad (1)$$

where A, B and C are PDE parameters, and $f(s,t)$ denotes simulated properties at space s and time t. Finite difference is a numerical method to approximate derivative expressions. In this example, the target space s includes three dimensions x, y, z and derivatives are replaced with first-order finite difference expressions. The system status can be propagated as shown in Figure 3. α, β and γ are constant coefficients calculated with finite-difference method. The corresponding 3D stencil is shown in Figure 2(a). As neighbouring data are required to support the calculation, as shown in Figure 2(b), boundary data are not updated during computation, named as halo data. In each time step t, the constructed stencil sweeps over kernel data to propagate $f(s,t)$ in time dimension. The number of arithmetic operations in Figure 3 can be calculated as $nt \cdot nz \cdot ny \cdot nx \cdot N_{ar}$, where N_{ar} is the number of arithmetic operations for each data point.

As neighbouring data at multiple dimensions are required for each computation, spatial locality reduces as the dimension size and the number of dimensions increase. If the dimension size is 1024, data accessed in the stencil in Figure 3 span 8 MB data. Limited by the sparse data access patterns, performance of stencil computations is limited to 1.8 GFLOPS [4] on a 4-core Intel i7-870 CPU for a fifth-order stencil. Propagating the stencil for 1000 time steps in $1024*1024*1024$ space requires 63.4 Tera floating-point operations, and takes 10 hours to finish.

```
1: for t ∈ 0 → nt do
2:     for z ∈ 1 → nz − 1 do
3:         for y ∈ 1 → ny − 1 do
4:             for x ∈ 1 → nx − 1 do
5:                 f[t+1][z][y][x] = (f[t][z][y][x−1] + f[t][z][y][x+1]) * α
6:                                 + (f[t][z][y−1][x] + f[t][z][y+1][x]) * β
7:                                 + (f[t][z−1][y][x] + f[t][z+1][y][x]) * γ
8:                                 - f[t−1][z][y][x];
9:             end for
10:        end for
11:    end for
12: end for
```

Fig. 3. An example of a stencil code pattern supported by Dynamic Stencil.

The high-performance requirements limit the usage of stencil computations in scientific research and industrial development.

Parallelism in stencil algorithms can be exploited with optimisation techniques such as design parallelisation, spatial blocking (loop tiling, domain decomposition), temporal blocking and communication scheduling. In general-purpose processors such as CPUs and GPUs, stencil designs are parallelised through spatial blocking, which refers to dividing involved data into multiple blocks to improve temporal data locality. As shown in Figure 2(c), for a 3-D stencil application, blocking the lowest two dimensions (x and y) in half reduces data distance between neighbouring data at the highest dimension (z) by 75%, which allows four parallel cores to process data blocks with improved data locality. When performance of parallelised designs is bounded by memory bandwidth, temporal blocking is used to propagate multiple time steps with one memory pass. As shown in Figure 2(d), propagating stencil data for time steps t and t+1 can be accomplished by either executing the unblocked designs twice, or buffering the intermediate results on-chip to eliminate the redundant memory access operations. Communication operations, as shown in Figure 2(e), are required to exchange halo data between neighbouring devices, when stencil applications are mapped into multi-device clusters. If not properly scheduled, the communication overhead increases with the number of involved devices, which severely limits design scalability.

B. Related Work

Stencil computations have been extensively studied across various platforms including many-core processors, hardware accelerators such as GPUs and FPGAs, and large-scale clusters. In large-scale CPU clusters, communication patterns of stencil computations are customised to fit into communication infrastructures [5], [6]. Data reuse method, workload distribution and communication scheduling are optimised for various GPU architectures [7]–[9] to exploit the massive parallelism and high memory bandwidth. For reconfigurable platforms, streaming architectures with customised memory architectures and datapaths are proposed [10], [11] to exploit available resources. These hardware architectures, efficient as they are, require high-level expertise and manual optimisation. Moreover, the

design effort to apply these approaches to optimise computational resources and enable efficient cooperation between them during run time is very high.

Automatic design frameworks are proposed to enable non-expert developers to utilise the various platforms for stencil computations. Optimisation techniques are often technology dependent, and vary with availability of resources and algorithm properties. Parallel GPU codes are generated in [12] to optimise stencil applications based on properties of GPU architectures. Spatial blocking is optimised to balance workload among parallel threads [13], and auto-tuners are built to search for the optimal blocking strategies for various resources [14] and data structures [15]. Temporal blocking is supported with a blocking algorithm [2], and the design space is searched with various searching algorithms to minimise execution time for CPU and GPU designs. The auto-tuners, which are widely used for general-purpose processors such as CPUs and GPUs, require a long execution time to traverse their search space. Run-time construction and adaptation of designs require rapid update in design configurations, therefore the auto-tuning process is not suitable. MaxGenFD [16] provides a design interface for users to specify design parallelisation and spatial blocking ratios during compile time. This semi-automatic approach requires running a time-consuming synthesis toolchain multiple times to optimise designs, and the optimised designs are statically configured.

When stencil computations are mapped into multiple devices, communication operations are scheduled to provide scalable performance. A programming model is proposed for stencil computations to implicitly translate stencil descriptions into scalable GPU implementations [17], and a multi-FPGA design approach for 1-D stencil computations in FPGAs is proposed in [18]. Communication patterns of these approaches are statically configured, and cannot adapt to run-time variations such as: the design parallelism, the communication bandwidth, the number of involved nodes and the distributed workload.

In this work, we propose a novel approach in which three models: an optimisation model, a communication model and a performance model, are used to systematically derive an optimised design that can automatically scale at both compile time and run time to exploit available resources during the life cycle of the design. Results from the related work are compared with the proposed approach in Section IV.

III. Methodology

The proposed approach, known as Dynamic Stencil, starts with a description for stencil computations, as shown in Figure 3, and ends up as a reconfigurable design that can adapt to available resources at run time. The development process of a Dynamic Stencil design is demonstrated in Figure 4, which includes three steps: compile-time optimisation, run-time initialisation and run-time scaling. The **compile-time optimisation** step first translates a stencil kernel into a data-flow graph. This data-flow graph captures all the kernel operators, the operator dependencies and memory access patterns. This intermediate kernel representation is used with the optimisation model to generate a stream-based architecture supporting multiple inter-connected FPGAs to form a Dynamic Stencil. Design parallelisation, spatial blocking and temporal blocking are integrated into our optimisation model which facilitates evaluation of their impact on the optimised architectures. Differences in FPGA nodes are expressed as variations of available resources. Bounded by available resources, the basic hardware architecture is automatically optimised to achieve maximum throughput, and then synthesised with back-end vendor tools to generate executable bitstreams. **Run-time initialisation** refers to constructing the initial Dynamic Stencil design when the application is launched into clusters. Interconnections between FPGAs are required to support data exchange with neighbouring devices, as shown in Figure 2(e). A topology where FPGA nodes are chained with point-to-point connections or connected host CPUs is referred to as an FPGA path. Synthesised designs for various nodes are loaded and linked to occupy the longest FPGA path among available FPGA nodes. An asynchronous communication model is developed based on properties of mapped designs, resource status and communication bandwidth. Communication operations among utilised nodes are scheduled to run in parallel with communication operations, with data dependency expressed as timing constraints in the model. **Run-time scaling** is triggered if a Dynamic Stencil design finds available nodes to expand. Benefits and overhead for expanding current design into the new nodes are evaluated with a run-time performance model. Once the benefits outweigh the overhead, a run-time scaling algorithm is executed to reconfigure new nodes, switch context into the scaled Dynamic Stencil design, and dynamically update design configurations to ensure correct functionality for the scaled Dynamic Stencil design. Variables and parameters for a Dynamic Stencil design are presented in Table I. Variables for the optimisation model are used as parameters in the communication model and the performance model.

TABLE I. Variables and selected parameters in the Dynamic Stencil approach (indices: i=dimension, n=node, j={front,end})

variables		parameters	
optimisation model			
par_n	design parallelism	A	available resources
sk_i	spatial blocking ratio	BW	available bandwidth
tk	temporal blocking ratio	w_i	stencil size
sl	size of a data slice	D	stencil dimension
communication model			
dc_n	computation delay	F	number of FPGAs
$dm_{n,j}$	memory delay	par_n	design parallelism
$tr_{n,j}$	arrival time for halo data	sk_i,tk	blocking ratios
		dw_n	distributed workload
		w_i	stencil size
performance model			
rtb	run-time benefit	F	number of FPGAs
rov	reconfiguration overhead	par_n	design parallelism
		sk_i,tk	blocking ratios

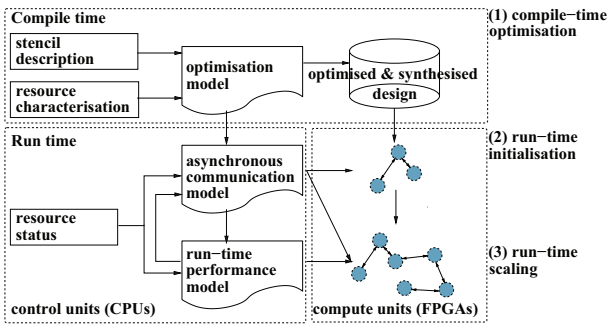

Fig. 4. The Dynamic Stencil design approach has three steps: compile-time optimisation, run-time initialisation and run-time scaling.

A. Compile-Time Optimisation

The basic streaming architecture for stencil computations is shown in Figure 5(a). Arithmetic operations in stencil descriptions are mapped to pipelined data-paths, and a memory

architecture is built based on the extracted data access pattern (stencil shape). At each clock cycle, the stencil moves one step forward in the fastest (x) dimension, one data unit is streamed from off-chip memory, and $\sum_{i=1}^{D} w_i \cdot 2 + 1$ stencil data units are loaded from the on-chip memory, where w_i indicates the number of data units in a stencil at dimension i. In the example in Figure 5(a), w_z is 1. Design parallelisation variable par indicates the number of replicated data-paths. For a streaming architecture with $par = 4$, as shown in Figure 5(b), the replicated stencil moves four steps forward in the x dimension at each clock cycle. The same memory architecture is used to share accessed data, while four data-paths are replicated to process data in parallel. Resources consumed by a pipelined data-path can be estimated by accumulating resources consumed by each arithmetic operator, with $R_{op \in \odot = \{+,-,\bullet,\div\}}$ and S_{op} indicating resource utilisation for operator op and the number of operator op in the stencil description, respectively. Theoretically, n valid results can be generated per clock cycle for a streaming architecture configured as $par = n$ if the design can be accommodated in a single device and if memory bandwidth permits.

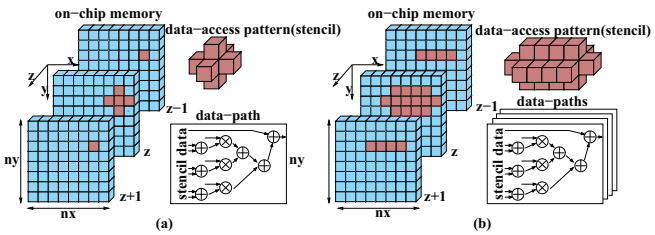

Fig. 5. Data access patterns, memory architectures and data-paths in streaming architectures for stencil computations with (a) a single data-path ($par = 1$) and (b) four replicated data-paths ($par = 4$).

Spatial blocking is applied to reduce memory resource consumption. While the number of buffered data slices is algorithm-specific, the slice size depends on the size of the corresponding dimensions (nx and ny in Figure 5). When the number of dimensions increases, memory resource consumption can easily exceed resource constraints. Blocking dimensions in memory slices regroups streaming patterns in the blocked dimensions, which effectively reduces slice size and memory resource consumption. As shown in Figure 2(c), to protect data dependency of boundary data after blocking, one layer of halo data is distributed to blocked data. In a dimension i with n_i kernel data (as shown in Figure 2(b)) and blocking ratio sk_i, the size of blocked dimension can be expressed as $\frac{n_i}{sk_i} + 2 \cdot w_i$. Since halo data are distributed to each data block, spatial blocking increases the overall data size compared with unblocked designs.

Temporal blocking is applied to reduce memory bandwidth requirements. For a given memory bandwidth, there will be a point where the memory system cannot afford to load and to write par data units per clock cycle. As shown in Figure 2(d), when memory channels are saturated, output data of current time step can be stored as intermediate data accessed as input data for the next step, eliminating redundant memory access and accomplishing multiple time steps in one memory pass. The memory architecture is replicated to accommodate the intermediate data, and the attached data-paths are also replicated to process the intermediate data in parallel. Meanwhile, for the spatially blocked data, accomplishing one more time step on-chip introduces one more layer of halo data for data blocks, to ensure halo data of intermediate results can be properly updated without synchronising with neighbouring blocks. Therefore, the size of blocked dimension i with spatial blocking ratio sk_i and temporal blocking factor tk can be expressed as $\frac{n_i}{sk_i} + 2 \cdot w_i \cdot tk$, where tk layers of halo data are represented as $2 \cdot w_i \cdot tk$. The size of one slice data after spatial and temporal blocking is:

$$sl = \prod_{i=1}^{D-1} \left(\frac{n_i}{sk_i} + 2 \cdot w_i \cdot tk \right) \tag{2}$$

An optimisation model is developed to determine design parallelism par, spatial blocking ratio sk and temporal blocking ratio tk to achieve minimum execution time, i.e., the ratio between overall data size and computational capacity. Overall data size is expressed as $nsteps \cdot sl \cdot \prod_{i=1}^{D-1} sk_i \cdot n_D$, where $nsteps$ is the number of time steps, $\prod_{i=1}^{D-1} sk_i$ indicates the number of data blocks, and n_D is the size of the slowest dimension corresponding to the number of data slices (n_z in Figure 5). Computational capacity $par \cdot tk$ increases with design parallelism and temporal blocking ratio, while slice size and number of data blocks increase with temporal and spatial blocking ratios. Bounded by available on-chip resource A and off-chip bandwidth BW, the model is expressed as:

$$\textbf{minimise:} \quad \frac{nsteps \cdot sl \cdot (\prod_{i=1}^{D-1} sk_i) \cdot n_D}{par \cdot tk} \tag{3}$$

subject to:

$$\sum_{op \in \odot} S_{op} \cdot R_{op} \cdot par \cdot tk + I_{LT/FF/DP} \leq A_{LT/FF/DP} \tag{4}$$

$$ws \cdot tk \cdot sl \cdot 2 \cdot w_D + I_{BR} \leq A_M \tag{5}$$

$$par \cdot ws \cdot \max(IO_{in}, IO_{out}) \cdot fq \leq BW \tag{6}$$

Infrastructure resource consumption $I_{LT/FF/DP/BR}$ indicates the LUTs, FFs, DSP and BRAM resources consumed by communication infrastructures. Eq. 4 expresses the data-path resource consumption which increases linearly with par and tk. The memory resource consumption is estimated in Eq. 5, with tk on-chip memories implemented, and $2w_D$ data slices stored in each on-chip memory. Design parallelism par does not affect memory resource consumption, as the replicated data-paths share the same memory architecture. ws is the width of each data unit (for single floating-point stencils: ws=32 bits). The impact of temporal blocking on memory bandwidth is expressed in Eq. 6, where an increase in tk does not contribute to memory requirements. IO indicates the number of input / output arrays, and fq is the operating frequency.

B. Run-time Initialisation

When a stencil application is mapped into a reconfigurable cluster, optimised designs for available FPGA nodes are loaded and connected to form an initial Dynamic Stencil design. Inter-node communication channels are required between neighbouring nodes to exchange halo data, as shown in Figure 6(a). Among available FPGA nodes, we extract the longest connected path to accommodate the initial Dynamic Stencil design. Design properties of an occupied node i are abstracted with its computational capacity $p_{ar[i]}$, i.e., the number of data processed per clock cycle. As demonstrated in Algorithm 1, after workload distribution (line 2), arrive times of remote halo data can be calculated (line 3), where i indicates the involved FPGA, and j indicates the halo data region. To meet local timing constraints, an initial delay d_{co} can be inserted before computation starts (line 5-10), and a memory delay

d_{me} can be inserted in each iteration to postpone the time to update halo data in local memory (line 11-15), as illustrated in Figure 6(b) and (c). In the following, we elaborate on the workload distribution, the local timing constraints, and the scheduling operations.

For F available FPGA nodes, the slowest dimension with size n_D is decomposed to balance the distributed workload dw_n based on the computational capacity of involved nodes:

$$dw_n = sl \cdot \left(\prod_{i=1}^{D-1} sk_i\right)\left(\frac{n_D}{\sum_{k=1}^{F} par_k} \cdot par_n + 2 \cdot w_D\right) \quad (7)$$

As shown in Figure 6(b), the decomposed data are processed simultaneously in the three involved FPGAs. In an FPGA node, the distributed workload are processed with par data per clock cycle, propagating a time step with $\frac{dw_n}{par_n}$ cycles. Results for kernel data are transmitted to neighbouring nodes to update the corresponding halo data used in next time step. Communication operations between involved FPGAs are performed in parallel with the computation operations to eliminate communication overhead. To satisfy data dependencies in a stencil computation, halo data from remote devices must arrive: (a) after the halo data in current time step are consumed, and (b) before the halo data in next time step are used. In the unscheduled design in Figure 6(b), the end halo data for `FPGA0` and `FPGA1` arrive too early and overwrite the existing end halo data before they are used, rendering all subsequent computations incorrect. An asynchronous communication model is built to facilitate translation of data dependencies into timing constraints, and expression of arrival times in terms of computational capacity and communication bandwidth. Variables and parameters for this model are presented in Table I.

Algorithm 1 Communication scheduling algorithm for a Dynamic Stencil design

input: the number of FPGAs in detected FPGA path: D_{ev}
output: scheduled computation delay d_{co} and memory delay d_{me} for each FPGA.

1: **for** $i \in 0 \rightarrow D_{ev}$, j $\in$ (0,1) **do**
2: $dw_{[i]}$ = workload_distribution()
3: $t_{ar[i,j]}$ = unscheduled_arrive_time()
4: **end for**
5: **for** $i \in 0 \rightarrow D_{ev}$, j $\in$ (0,1) **do**
6: **if** latest_arrive_time() $< t_{ar[i,j]}$ **then**
7: $d_{co,i} = \max(t_{ar[i,j]} -$ latest_arrive_time())
8: $t_{ar[i+1,j]} += d_{co[i+1]}$
9: **end if**
10: **end for**
11: **for** $i \in 0 \rightarrow D_{ev}$, j $\in$ (0,1) **do**
12: **if** earliest_arrive_time() $> t_{ar[i,j]}$ **then**
13: $d_{me[i]} = \max($earliest_arrive_time() - $t_{ar[i,j]})$
14: **end if**
15: **end for**

Timing constraints can be expressed with the consume time of halo data in the current time step and in the next time step, as shown in Figure 6(b). For the halo data j in node n, the arrival times $tr_{n,j}$ are bounded as follows: with $j = front$ indicating the front halo data and $j = end$ indicating the end halo data:

$$\begin{cases} 0 & \leq tr_{n,front} & \leq \frac{dw_n}{par_n} \\ \frac{dw_n - sl \cdot w_D}{par_n} & \leq tr_{n,end} & \leq \frac{2 \cdot dw_n - sl \cdot w_D}{par_n} \end{cases} \quad (8)$$

where $\frac{dw_n}{par_n}$ is the computation time for one time step, and $dw_n - sl \cdot w_D$ indicates the distance between front halo data and end halo data.

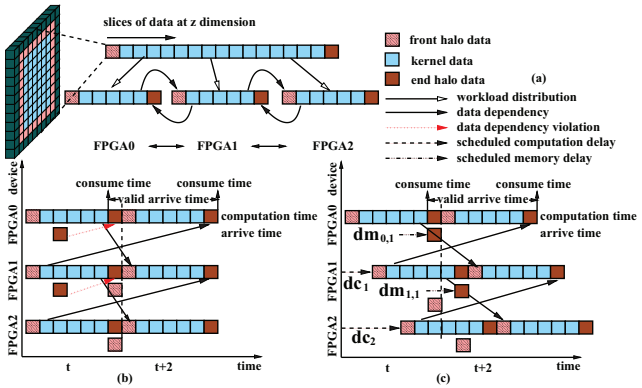

Fig. 6. (a) Decomposed data for an FPGA path with three interconnected FPGAs, and the corresponding communication and computation operations in time dimension if (b) unscheduled and (c) scheduled. Each grid in the figure represents one data slice. Data dependencies, valid times and scheduled delays are labelled in this figure. The three FPGAs process 3 data units per clock cycle, and communication channels can transmit 1 data unit per clock cycle.

Arrival times of halo data are determined by computational capacity in neighbouring nodes and the communication time. The front halo data at node n are derived from the kernel data $(dw_{n-1} - 2 \cdot w_D \cdot sl, dw_{n-1} - w_D \cdot sl)$ at node $n-1$. Similarly, end halo data at node n are generated from the kernel data $(sl \cdot w_D, 2 \cdot w_D \cdot sl)$ at node $n+1$. As there are $\frac{w_D \cdot sl}{par_n}$ cycles delay between when the kernel data are loaded and when the correspondingly results are generated, unscheduled arrival times of halo data j in node n can be expressed as:

$$tr_{n,j} = \begin{cases} \frac{dw_{n-1} - w_D \cdot sl}{par_{n-1}} + \frac{w_D \cdot sl}{bw_n} \cdot m & j = front \\ \frac{2 \cdot w_D \cdot sl}{par_{n+1}} + \frac{w_D \cdot sl}{bw_{n+1}} \cdot m & j = end \end{cases} \quad (9)$$

where bw_n indicates the communication bandwidth between node $n-1$ and node n, m is the margin factor for communication operations, and $\frac{w_D \cdot sl}{bw_n} \cdot m$ is the communication time.

Communication scheduling refers to configuring memory delay dm and computation delay dc to satisfy the timing constraints. If halo data arrive too early, a memory delay dm is inserted to postpone the update time of the halo data in local memory. On the other hand, when halo data arrive too late, since computation operations are initially scheduled to start as early as possible, there is no space to schedule the halo data to arrive earlier. Instead, the starting time of the communication operations is delayed to postpone the latest timing constraints. The actual arrival times after scheduling are $tr_{n,j} + dm_{n,j}$. Since the inserted computation delay postpones the timing constraints, if we keep the timing constraints the same as in Eq. 8, the arrival times can be expressed as $tr_{n,j} + dm_{n,j} - dc_n$.

C. Run-time Scaling

Once initialised, a Dynamic Stencil design investigates run-time status variations to utilise FPGA nodes provisioned during its lifetime. The mapped FPGA path can be expanded if available FPGA nodes can be connected to either the first node or the last node in the current FPGA path. Scaling a Dynamic Stencil design involves run-time evaluation, context switching, run-time reconfiguration and configuration update. Context switching refers to redistributing the intermediate results from the current Dynamic Stencil design into the FPGAs of the new scaled Dynamic Stencil design. Input and output arrays of the current FPGAs are loaded from off-chip memories back to the host memories; appropriate bitstreams are used

to configure the newly-available FPGAs; and the intermediate arrays are redistributed into the FPGAs of the expanded Dynamic Stencil design, thus ensuring that the context of the current stencil computation is preserved in each FPGA. A control unit for the Dynamic Stencil design is implemented on the host CPU, which executes run-time evaluation, design scaling and configuration update.

Run-time benefit rtb refers to the reduction in execution time for the remaining stencil computation when a Dynamic Stencil design expands into more FPGAs. Remaining workload for a stencil application is calculated with its remaining time steps $nsteps$ and distributed workload dw_n. If available nodes are employed, the distributed workload is reduced to dw'_n, and the reduction in execution time is expressed as:

$$rtb = \frac{nsteps}{tk} \cdot \frac{(dw_n - dw'_n)}{par_n \cdot fq} \tag{10}$$

where $dw_n - dw'_n$ indicates the reduction in workload. As distributed workload is proportional to par_n (Eq. 7), rtb is the same for each node.

The scaling overhead refers to time consumed to reconfigure devices and redistribute data, and can be estimated as:

$$rov = \max\left(\frac{R \cdot \phi}{\theta}, \frac{dw_n}{bw_{pci}}\right) + \frac{dw'_n}{bw_{pci}} \tag{11}$$

where $\frac{dw_n}{bw_{pci}}$ and $\frac{dw'_n}{bw_{pci}}$ respectively indicate the time to load and redistribute memory data, through PCI-E channels with bandwidth bw_{pci}. The reconfiguration time can be estimated with bitstream size and throughput of reconfiguration interface θ. The bitstream size is calculated with resource consumption R and bitstream size per resource unit ϕ. Since memory controllers and streaming architectures are configured into the same FPGA in current designs, context data can only be written into new FPGA nodes when run-time reconfiguration is finished. The loading of context data, on the other hand, is executed in parallel with reconfiguration operations.

The scaling algorithm for a Dynamic Stencil design coordinates the communication model and the run-time performance model. Resource status is monitored after a Dynamic Stencil design is initialised at time 0, with run-time benefits and overhead evaluated for detected available resources. If run-time benefit rtb outweighs the scaling overhead rov, the scaling algorithm stalls computation for the next time step in FPGA nodes, reconfigures available nodes and switches context data into the new nodes. Parameters of communication model in Algorithm 1 are updated, and the communication variables are rescheduled to respond to the design variations. Computation operations are then resumed for the scaled Dynamic Stencil design, and the scaling algorithm goes back to the monitoring phase. The algorithm is executed iteratively to adapt a Dynamic Stencil design to run-time resource variations.

IV. Results

Starting from a simple stencil description, the proposed approach generates a run-time scalable design for reconfigurable clusters. A benchmark application, Reverse-Time Migration (RTM), is developed with the proposed approach. The developed design is evaluated with three aspects: resource exploitation, design scalability and run-time adaptivity, which respectively reflect how available resources are exploited for the optimised single-node design, the initially constructed Dynamic Stencil design and the dynamically scaled Dynamic Stencil design. RTM is an advanced seismic imaging technique to detect terrain images of geological structures, based on the Earth's response to injected acoustic waves. The propagation of injected waves in terrains is modelled with the isotropic acoustic wave equation [19], which is solved with a fifth-order stencil in space with three dimensions. Hardware designs are compiled by MaxCompiler version 2012.1. They operate at 100 MHz, and run on an MPC-C500 compute node from Maxeler Technologies. The MPC-C500 has four MAX3 dataflow engines, each of which has a Xilinx Virtex-6 SX475T FPGA.

TABLE II. Single-device and multi-device performance comparison.

single-device performance						
System	freq (GHz)	TH(Gflops)[1]	P(Watt)[1]	E(Gflops/w)[1]		S[1]
CPU	2.93	1.8	183	0.01		72x
GPU [7]	1.15	58.8	369	0.159		2.2x
MaxGenFD [16]	0.1	71.3	137	0.52		1.8x
Dynamic Stencil	0.1	130.67	142	0.92		1x
multi-device performance (throughput (Gflops))						
System/number of devices	2	4	8[2]	16[2]	32[2]	S
Blue Gene/P [5]	2.98	5.96	11.92	23.84	47.68	87.8x
Blue Gene/Q [6]	38.4	76.8	153.6	307.2	614.4	6.8x
Cray XK6 [9]	181	362	524	1048	2096	2.00x
MaxGenFD[3][16]	128.3	196.8	n/a	n/a	n/a	2.66x
Dynamic Stencil	261.3	523	1045	2091	4184	1x

[1] TH, P, E and S respectively stand for throughput, power consumption, power efficiency and speedup.
[2] Limited by available resources, performance for more than 4 FPGAs is simulated. When 1 to 4 FPGAs are involved, measured performance confirms the simulated results.
[3] MaxGenFD supports up to 8 FPGAs, performance cannot be simulated due to lack of optimisation details. Measured scalability for 4 FPGAs is 0.69.

Resource exploitation in an FPGA is evaluated in terms of resource consumption and achieved design throughput. Resource consumption and design throughput of the optimised design are presented in Figure 7. The resource consumption is normalised against available resources, and the resource consumption when design parallelism is 0 indicates the resources consumed by communication infrastructures. Before off-chip memory channels are saturated when $par = 16$, each replicated data-path generates one result per clock, with design throughput and data-path resource consumption scaled linearly. Temporal blocking ratio tk is increased to 2 when the memory bottleneck is hit. One more on-chip memory with 16 attached data-paths are replicated, doubling the performance as well as resource consumption. Design variables par, tk, sk_x and sk_y of the optimised design are respectively configured as 16, 2, 6 and 5. The optimised design consumes 270816 LUTs, 323134 FFs, 952 DSPs and 989 BRAMs, with the optimisation model estimating the design to consume 255936 LUTs, 357120 FFs, 806 DSPs and 947 BRAMs. The optimisation model can capture variation in resource consumption wth more than 90% accuracy, enabling automatic optimisation of stream architectures for FPGAs with various characteristics.

Design performance is listed in Table II. Reference single-device designs include parallelised software designs targeting a 4-core Intel i7-870 CPU, Blue Gene/P [5] and Blue Gene/Q designs [6], a GPU design optimised by NVIDIA [7] and customised for NVIDIA Tesla C2070, and an FPGA design developed with MaxGenFD [16]. Overall performance of the optimised RTM is reduced from 156.8 GFLOPS to 130.67 GFLOPS, due to the additional data introduced by spatial and temporal blocking. Performance of CPU and GPU designs is limited by their general-purpose memory system. Run-time profiling shows that the optimised GPU design can only achieve 35% memory efficiency, i.e., loading one data unit

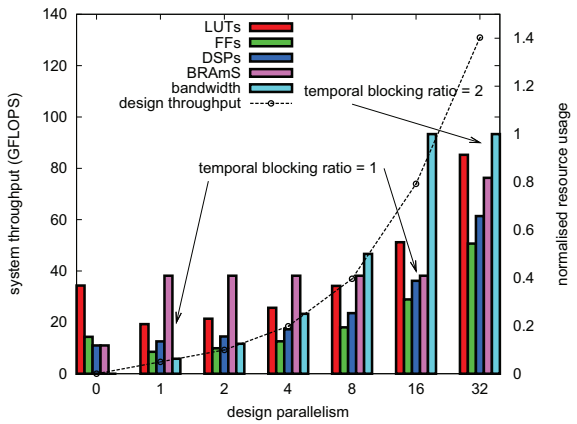

Fig. 7. Design throughput and resource consumption of the RTM design. The optimisation increases design parallelism to 16 until the bandwidth bottleneck is hit, and increase the temporal blocking ratio to utilise left resources.

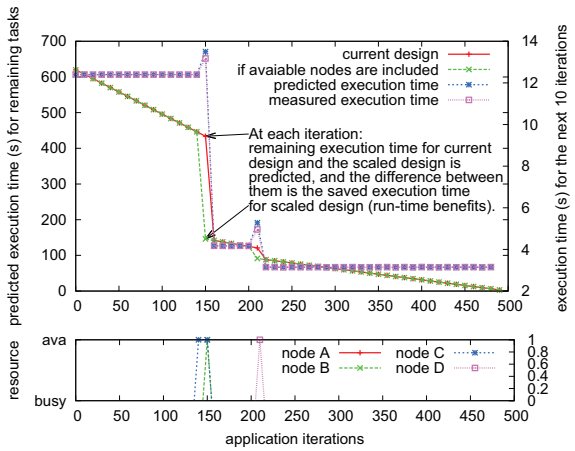

Fig. 8. Evaluation and prediction of the run-time performance model during one of the test cases, at the application iteration (time step) dimension. The resource status is measured from target cluster. 'ava' stands for available.

takes 3 clock cycles. Since temporal blocking is not covered by the automatic optimisation process of MaxGenFD, design parallelism of MaxGenFD designs is limited to 16, when off-chip memory channels are saturated. While temporal blocking can be manually implemented for MaxGenFD designs, to keep the design effort comparable, we use automatically optimised parameters for both Dynamic Stencil designs and MaxGenFD designs. In summary, the optimised design for RTM is up to 1.8 to 72 times faster and 1.7 to 92 times more power efficient than the reference designs.

Design scalability reflects the effectiveness of the asynchronous communication model. For the current platform, inter-FPGA communication operations are either through point-to-point channels with 3.2 GB/s bandwidth or through 10 Gb/s Ethernet connections between CPUs, with data moved between CPUs and FPGAs through 1GB/s PCI-E channels. The lower bound of inter-node bandwidth bw is 1 GB/s, while on-chip results are streamed with 48 bytes per clock cycle. In the asynchronous communication model, the computation delay dc in involved FPGAs is scheduled to be $5w_D = 25$ data slices sl to reduce the bandwidth requirement to 0.8 GB/s, with margin factor $m = 1.25$. Memory delay dm is scheduled to ensure local halo data are consumed before being overwritten. Limited by available FPGAs in our platform, our current design scales up to 4 FPGAs. Based on computation throughput of utilised FPGAs and available bandwidth, performance of the Dynamic Stencil design when more FPGAs are involved is simulated. The simulated and measured results are presented in Table II, which shows that linear scalability has been achieved for the initialised Dynamic Stencil design. Previous large-scale designs on Blue Gene/P [5], Blue Gene/Q [6] and Cray XK6 [9] are also introduced to provide a comparison. As shown in Table II, the measured results confirm the simulated performance, and overall design throughput reaches 4.09 TFLOPS when 32 FPGAs are involved, outperforming the reference designs by 2 to 88 times. Besides throughput, power consumption in large-scale clusters determines the maintenance cost such as cooling infrastructures and electricity, and plays an important role in large-scale designs. Power consumption information is not provided in previous work [5], [6], [9]. If we make a conservative assumption that the Tesla X2090 GPUs in Cray XK6 consume the same power as the Tesla C2070 design in Table II, the Dynamic Stencil design is 5.2 times more efficient than the stencil design running on Cray XK6 when including all host and accelerator power consumption.

Run-time adaptivity of the developed design is evaluated with design performance and device-level resource utilisation ratio, when the RTM design is mapped into the reconfigurable cluster. For the available 4 FPGAs, static designs with 1, 2, 3 and 4 device-level parallelism are developed and executed to provide comparison. Run-time status during 10 separated time periods is measured and used as 10 test cases in this experiment. Evaluation process of the run-time performance model for one of the test cases is demonstrated in Figure 8. The run-time performance model predicts the execution time for the remaining tasks of the current design as well as the scaled design. When new nodes become available, the difference between the two predictions indicates the run-time benefits. FPGA node A is available when the application is launched, and node B, C and D are released by other computational tasks at 150, 142 and 209 iterations, respectively. Although node C becomes available earlier than node B, the detected FPGA path first expands when B is released due to the lack of communication channels between node A and node C. If node B and C are included in the Dynamic Stencil design, execution time for the following tasks is reduced by 357.4s, with 0.71s run-time overhead introduced. As the benefit outweighs the overhead, node B and node C are reconfigured to cooperate with the existing node A. Context data are redistributed, and design variables are rescheduled using Algorithm 1 to ensure linear scalability and correct functionality when the Dynamic Stencil design is expanded. Similarly, node D is employed by the Dynamic Stencil when it becomes available. As shown in Figure 8, the measured performance aligns with predicted execution time for the remaining tasks showing high accuracy of the performance model. Corresponding results in the test case are shown in Figure 9. Device-level parallelism for the static design using one FPGA is limited to 1, while the static designs using multiple FPGAs need to wait for released nodes to start. The Dynamic Stencil design finishes 490 time steps in 297 seconds, outperforming the static designs by 1.67 to 2.72 times.

Resource utilisation ratio is calculated with measured performance and the theoretical performance upper bound. The theoretical performance is calculated as the overall performance if FPGAs are fully utilised once released by other applications. The measured performance and resource utilisation for the 10 test cases are shown in Figure 10. Averaged resource utilisation ratio for the Dynamic Stencil design is 91%. The gap between the achieved resource utilisation ratio and the full

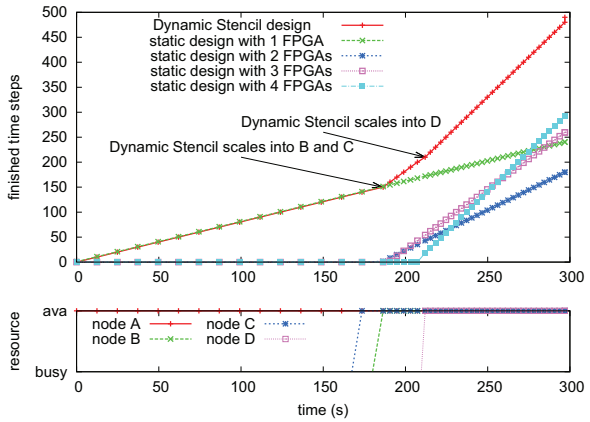

Fig. 9. Performance of a Dynamic Stencil design and a static design with 1, 2, 3 and 4 FPGAs at the time dimension.

utilisation ratio (100%) is introduced by the reconfiguration overhead and communication infrastructure. As shown in the test case in Figure 9, node C remains idle until the Dynamic Stencil design expands into node B, as there are no communication channels between node A and node C. Resource utilisation for static designs is limited between 40% and 49%. In other words, limited by pre-defined communication and computation patterns, 50% of resources in the cluster remain idle. On average, the high resource utilisation of the dynamic designs enables them to run 1.8 to 2.3 times faster than the static designs.

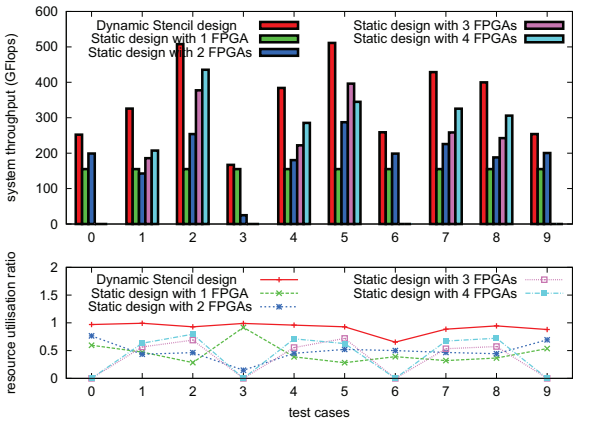

Fig. 10. Design performance and resource utilisation for 10 test cases.

V. Conclusion & Discussion

For large-scale reconfigurable clusters, effectiveness of conventional static design methods which pre-define communication patterns and hardware configurations are limited by unpredictable run-time conditions. In this paper, we propose Dynamic Stencil, a novel approach that statically optimises target applications for various FPGA nodes, and dynamically constructs an executable design that automatically adapts to resources available at run time. In particular, we achieve high resource utilisation ratio and significant speedup over reference designs at each stage of the approach for computationally intensive stencil applications. Limitations of our current design approach mainly come from its single-task considerations: a Dynamic Stencil design tends to occupy all available resources during its execution, which may not be the optimal solution when targeting maximum overall performance of multiple tasks; idle nodes due to lack of communication channels to existing Dynamic stencil design can be occupied by other computational tasks, which can further increase resource utilisation. Future work includes supporting dynamic design methods for multi-task and multi-user environments, which will be built on top of the current Dynamic Stencil approach, to exploit more complex run-time scenarios.

Acknowledgement. This work was supported in part by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 257906, 287804 and 318521, by 973 project 2013CB329000, National Natural Science Foundation of China (No. 61373026), by the HiPEAC NoE, by Maxeler University Program, and by Xilinx.

References

[1] D. A. Reed, L. M. Adams, and M. L. Patrick, "Stencils and problem partitionings: Their influence on the performance of multiple processor systems," *IEEE Trans. Computers*, vol. 36, no. 7, pp. 845–858, 1987.

[2] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-d blocking optimization for stencil computations on modern CPUs and GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–13.

[3] M. Ripeanu, A. Iamnitchi, and I. T. Foster, "Cactus application: Performance predictions in grid environments," in *Euro-Par*, 2001, pp. 807–816.

[4] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell, "Exploiting run-time reconfiguration in stencil computation," in *FPL*, 2012, pp. 173–180.

[5] M. Perrone *et al.*, "Reducing data movement costs: Scalable seismic imaging on blue gene," in *IPDPS*, 2012, pp. 320–329.

[6] L. Lu and K. Magerlein, "Multi-level parallel computing of reverse time migration for seismic imaging on blue Gene/Q," in *PPoPP*, 2013, pp. 291–292.

[7] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *GPGPU*, 2009, pp. 79–84.

[8] E. H. Phillips and M. Fatica, "Implementing the himeno benchmark with CUDA on GPU clusters," in *IPDPS*, 2010, pp. 1–10.

[9] M. Rietmann *et al.*, "Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures," in *SC*, 2012, p. 38.

[10] K. Sano *et al.*, "Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth," in *Proc. FCCM*, 2011.

[11] H. Fu and R. G. Clapp, "Eliminating the memory bottleneck: an FPGA-based solution for 3D reverse time migration," in *Proc. FPGA*, 2011.

[12] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *ICS*. ACM, 2012, pp. 311–320.

[13] S. Krishnamoorthy *et al.*, "Effective automatic parallelization of stencil computations," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 235–244.

[14] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC*. IEEE, 2008, p. 4.

[15] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[16] O. Pell, J. Bower, R. Dimond, O. Mencer, and M. Flynn, "Finite difference wave propagation modeling on special purpose dataflow machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 906–915, 2013.

[17] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011, pp. 1–12.

[18] X. Niu, J. G. F. Coutinho, and W. Luk, "A scalable design approach for stencil computation on reconfigurable clusters," in *FPL*. IEEE, 2013.

[19] M. Araya-Polo *et al.*, "Assessing accelerator-based HPC reverse time migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 147–162, Jan. 2011.