Accelerating Unstructured Finite Volume Computations on FPGAs

Zoltán Nagy [*] Csaba Nemes [†] Antal Hiba[†] Árpád Csík [‡] András Kiss[*†]
Miklós Ruszinkó [§] Péter Szolgay[*†]

November 7, 2012

Abstract

In the paper an FPGA based framework is described to efficiently accelerate unstructured finite volume computations where the same mathematical expression has to be evaluated at every point of the mesh. The irregular memory access patterns caused by the unstructured spatial discretization are eliminated by a novel mesh node reordering technique, and a special architecture is designed to fully utilize the benefits of the predictable memory access patterns. In the proposed architecture a fixed size moving window of the input stream of the reordered state variables is cached into the on-chip memory and a pipelined chain of processing elements, which gets input only from the fast on-chip memory, is used to carry out the computations. The arithmetic unit of the processing elements is generated from the data-flow graph extracted from the given mathematical expression. The data-flow graph is partitioned with a novel graph partitioning algorithm to break up the arithmetic unit into smaller locally controlled parts, which can be more efficiently implemented in FPGA than the globally controlled arithmetic unit. The proposed architecture and algorithms are presented via a case study solving the Euler equations on an unstructured mesh. On the currently available largest FPGA the generated architecture contains three processing elements working in a pipelined fashion to provide one order of magnitude speedup compared to a high performance microprocessor and 3 times speedup compared to a high performance GPU.

[*]Cellular Sensory and Wave Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences, 1518 Budapest, Pf. 63., Hungary, Email: nagyz@sztaki.hu

[†]Faculty of Information Technology, Pázmány Péter Catholic University, Budapest, Hungary

[‡]Széchenyi István University, Department of Mathematics and Computational Sciences, Győr, Hungary

[§]Applied Mathematics Research Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, Hungary

1 Introduction

Numerical simulation of complex problems evolving in time plays an important role in scientific and engineering applications. Accurate behavior of dynamical systems can be understood using large scale simulations, which traditionally require expensive supercomputing facilities. Several previous studies proved the efficiency of programmable logic devices in numerical simulation of various physical phenomena such as electromagnetism [1], transient waves [2] [3] and computational fluid dynamics (**CFD**) [4] [5] .

The most obvious way to solve complex spatio-temporal problems is numerical approximation over a *regular mesh* structure, however, practical applications usually contain complex boundaries which can be handled by *unstructured meshes* more efficiently. Unfortunately, conventional microprocessors have around 10% utilization during unstructured mesh computations due to the irregular memory access patterns of grid data.

To accelerate the computation irregular memory access patterns should be hidden by temporally storing the relevant grid points in the on-chip memory of the FPGA or in the CPU cache. The new state values of the grid points have to be computed in a proper order to fit into the on-chip memory, that is, once the state variables of a grid point are loaded into the on-chip memory they can be kept there as long as they are needed.

Reordering of the grid points can improve the performance of both CPU and FPGA implementations, however in case of FPGA it plays a more important part. In modern CPUs there is a fixed size cache, therefore, if the necessary grid points fit into the cache, it is not worth to further optimize the memory access patterns. On the contrary, in case of FPGA the size of the on-chip memory is user-definable and shall be minimized to decrease the area requirements of the circuit. The second difference is that in FPGA it is inevitable to fit into the cache because in this case a simple and fast memory handling can be designed, in which data can be loaded into the on-chip memory in bursts. This feature makes the FPGA very attractive for the acceleration, while without this feature the FPGA cannot compete with the CPU. Consequently, the mesh has to be divided into smaller parts, if the number of the necessary grid points which should be stored is still too large for the available on-chip memory after reordering.

In the paper both the reordering and the mesh partitioning problems are addressed and novel algorithms are given. The optimization task is handled as a Matrix Bandwidth Minimization problem, in which the mesh is treated as an undirected graph and the bandwidth of the adjacency matrix of the graph is minimized. In the new algorithm the mesh points are reordered to improve data locality

and an upper bound can be chosen by the user for the bandwidth of the resulting mesh parts.

If the problem of the efficient transfer of the input data to the chip is solved, the next question to be addressed is how to make computations on this data flow. In practice this means that at every mesh point the same mathematical expression has to be evaluated, which describes the new state values of the mesh point based on the current state values of the point and its neighbors. As there are plenty of mesh points to be processed, long pipelines are tolerable and a pipelined arithmetic unit(**AU**) is suitable which can operate at high frequency.

As our primary focus is on the operating frequency of the AU, each operator of the expression is implemented by a separate floating-point unit (**FPU**) selected from Xilinx IP Core library. Each floating point unit can operate at the desired operating frequency; however, the question to be answered is how to connect and control them to reach the desired operating frequency for the whole AU. In a previous work [6] it has been shown that higher operating frequency can be reached if the FPUs of the AU are partitioned into locally controlled groups. The partitioning problem can be interpreted as the partitioning of the data-flow graph generated from the given mathematical expression. A partition which results in a fast AU has to fulfill several requirements, which cannot be addressed by traditional graph partitioning [7]. One of the main disadvantages is that they produce clusters which cannot be placed and routed in FPGA without long interconnections that limit the overall operating frequency of the AU. Furthermore, they blindly group FPUs which are in totally different position in the pipeline and produce awkward AUs where the overall pipeline length and the area requirements are over-sized.

In the paper a novel partitioning representation and data-flow graph partitioning algorithm are given in which both the length of the interconnections and the length of the overall pipeline can be addressed to design a fast and locally distributed control to the AU. The presented framework can automatically convert a mathematical expression described in a text file to a locally controlled AU represented in VHLD, which can be synthesized and implemented with the standard Xilinx tools [8].

If efficient usage of the external memory access and the control of the AU are solved, the FPGA architecture is a perfect candidate to accelerate unstructured finite volume computations. The proposed algorithms for these problems are demonstrated in case of a complex CFD problem.

In Section 2 an overview of recent publications on accelerating unstructured mesh computations are given. The proposed accelerator architecture including memory data structures and local control unit is presented in Section 3. The mesh point reordering algorithm and the mesh partitioning are presented in Section 4, while the data-flow graph partitioning and AU generation are described in Section 5. The proposed methods are tested on a complex case study described in Section 6. Finally,

results and performance analysis are given in Section 7.

2 Related work

Several papers were published in the early 2000s dealing with the acceleration of Partial Differential Equations (PDE) on unstructured meshes. Most of them were focused on accelerating Finite Element Methods (FEM), where the global stiffness matrix is given and the resulting large linear system of equations is solved usually by the iterative Conjugate Gradient (CG) method. The most time-consuming operation of this method is a sparse matrix vector multiplication, therefore, most of the works try to accelerate this part. Though our architecture is designed for explicit unstructured finite volume calculations, examination of these architectures is helpful because similar problems arise in our case. For example the adjacency matrix of the mesh is sparse and elements of the state vector are accessed in a random order.

In 2000 Jones and Ramachandran [9] examined several aspects of accelerating unstructured mesh computations on FPGAs. They proposed a hybrid architecture to accelerate the CG algorithm, where the local stiffness matrix and the bulk of the CG algorithm was computed by the CPU and only the matrix vector multiplication was performed on the FPGA and achieved $22.4-35.7$MFLOPs computing performance.

Another approach was the architecture proposed by deLorimier and DeHon [10], where all elements of the matrix were stored on the FPGA to avoid bandwidth limitations but this solution severely limited the size of the matrix. Performance of the architecture depended on the structure of the matrix and usually 66% of the peak performance could be achieved, which resulted in 2-10 times acceleration compared to the common microprocessors at that time.

Generally, if the size of the input data is too large to be stored fully on the chip (or in a relatively fast memory) a streaming architecture is needed, and prior preparation of the input stream can highly improve the performance [11]. Elkurdi et al. [12] proposed a process where the finite element sparse matrix was reorganized into a banded form in the first step. Then the matrix vector multiplication was calculated along special pipelineable diagonal stripes, where two successive elements could be processed by a pipelined architecture. Performance of the architecture was determined by the available memory bandwidth and the sparsity of the matrix, however utilization of the processing elements was varying in a very wide range between $17.74-86.24\%$.

duBois et al. [13] presented an architecture where nonzero elements from each row of the sparse matrix are processed in 7 element wide vectors. They also proposed to use a reordered banded matrix

to improve data locality, but the architecture still suffered from memory bandwidth limitation.

Recently Nagar et al. [14] proposed an architecture using an optimized Streaming Multiply- Accumulator with separate cache memories for matrix and vector data. The implementation platform they used has special memory architecture providing high 20GB/s peak memory bandwidth. Performance of the system with four FPGAs is in the $1.17 - 3.94$GFLOPs range outperforming a Tesla 1070 GPU. However, utilization of the processing elements is around 50%, similarly to the previous architectures, and increasing the number of processing elements to occupy the entire FPGA still runs into a memory bandwidth limit.

The surveyed architectures provide general solutions to accelerate FEM calculations on FPGAs but suffer from the inherent high memory bandwidth requirement and the small FLOPs per loaded bytes ratio of sparse matrix vector multiplication. On the contrary, the bandwidth requirement of our architecture can be adapted to the memory bandwidth limitations of the given prototyping board, and the processing elements of the proposed architecture are implemented in a pipelined chain to fully utilize the available FPGA resources. As the processing elements can be continuously fed with input data and the number of processing elements is only limited by the available resources of the given FPGA, high FLOPs per loaded bytes ratio can be achieved.

3 Architecture

3.1 Data structures and memory access patterns

The data structures used in the presented architecture were designed to efficiently use the available memory bandwidth during transmission of the unstructured mesh data to the FPGA. In numerical simulations data is discretized over space and can be represented at the vertices of the mesh (*vertex centered*) or the spatial domain can be partitioned into discrete cells (e.g. triangles) and the data is represented at the center of the cells (*cell centered*). From the aspect of accelerator design both vertex and cell centered discretization can be handled with a very similar memory data structure and accelerator architecture. In both cases the data can be divided into a time dependent (state variables) and a time independent part which contains mesh related descriptors (e.g. connectivity descriptor) and physical constants.

In the solution of the CFD problem which is presented in the paper the cell centered approach is used, therefore the proposed data structures are explained for this case. In Figure 1 an example of an unstructured mesh is shown, in which the cells are ordered and the computation of the new state values

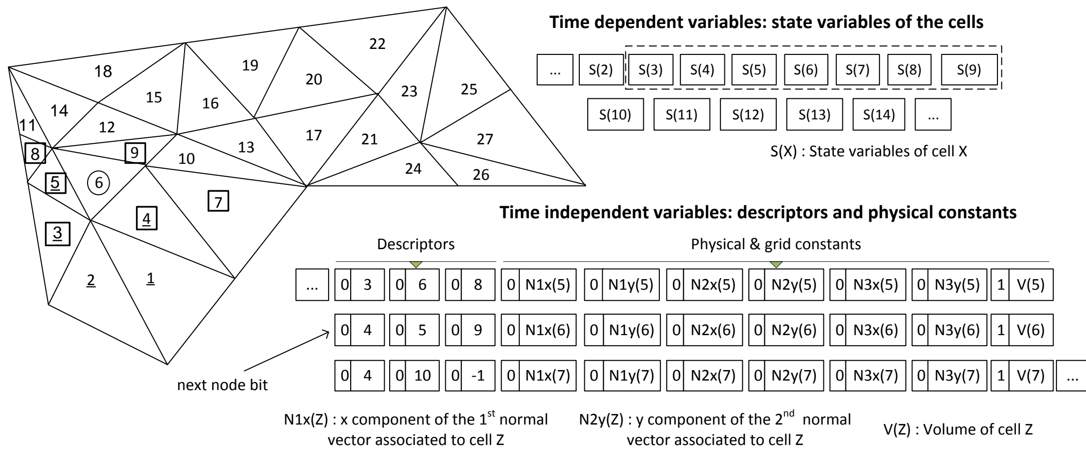

Figure 1: On the left an unstructured mesh is shown to illustrate which cells are stored in the on-chip memory. Indices of the cells indicate the order of computation. Already processed cells are indicated by underlined numbers, the currently processed cell is encircled and squared cells are currently stored in the on-chip memory. On the right fragments of the appropriate data structures are presented.

is carried out in an increasing order. Already processed cells are indicated by underlined numbers, the currently processed cell is encircled and squared cells are currently stored in the on-chip memory. On the right side of the figure fragments of the appropriate data structures are illustrated.

Time dependent variables are composed of the state variables associated to the cells. When a cell is processed the state variables are updated based on the state variables of the given cell and its neighborhood (*discretization stencil*). State variables are transmitted to FPGA in processing order and can be stored in a fixed-size shift register (on-chip memory). When a cell is processed all the state variables of the cells from the neighborhood stencil must be loaded into the on-chip memory, however cells which have no unprocessed neighbors can be flushed out. In the presented example when cell 6 is processed all the necessary state variables are in the fast on-chip memory of the FPGA. To process the next cell (7) a new cell (10) should be loaded and cell 3 can be discarded from the on-chip memory. It is possible that multiple new cells are required for the update of a cell, indicating that the on-chip memory is undersized. The size of the required on-chip memory depends on the structure of the grid and the numbering of the cells, consequently in the paper great attention is paid for the ordering of the mesh points. In the presented cell centered example the neighborhood stencil is relatively simple, however, more complicated patterns can be handled in the same way.

Time independent variables are composed of mesh related descriptors and other physical constants which are only used for the computation of the currently processed cell. They mainly differ from the time dependent variables because they are stored only for the currently processed cell and they are not written back to the off-chip memory. Consequently, time dependent and time independent data must be stored separately in the off-chip memory, otherwise the updated state variables cannot be transferred back to the off-chip memory in bursts.

Mesh related descriptors describe the local neighborhood of the cell (vertex) which is currently processed. In unstructured grids the topology of the cells (vertices) can be described by a sparse adjacency matrix, which is usually stored in a Compressed Row Storage (CRS) format [15]. As the matrix is sparse and the vertices are read in a serial sequence (row-wise) the nonzero elements can be indicated by column indices and a *next node bit*, which indicates the start of a new row (see Figure 1). In the 2D cell centered case the descriptor of a cell is a list, which contains the indices of the neighboring cells, however in others cases (e.g. vertex centered) additional element descriptor may be required. Boundary conditions can be encoded by negative indices in the connectivity descriptor as illustrated in case of cell 7. If the size of the descriptor list is constant, the next node bit can be neglected.

Time independent data also contain physical constants which are needed for the computation of the new state values. These constants can be appended after the descriptors as shown in Figure 1. In case of the demonstrated CFD problem these constants are the normal vectors indicating the edges of the cells (triangle) and the volume of the cells.

3.2 Structure of the proposed processor

The processor were designed to efficiently operate on the input stream of the data of the ordered mesh points. The two main parts of the processor are the Memory unit and the Arithmetic unit (**AU**) as shown in Figure 2. The job of the Memory unit is to prepare the input stream and generate the necessary inputs to the AU, which should continuously operate and get new inputs in each clock cycle. The Memory unit could be reused with little adjustments if an other PDE needed to be solved, however the AU would have to be completely reimplemented according to the new state equations. Consequently, the presented automatic generation and optimization of the AU can drastically decrease the implementation costs of a new problem.

The Memory unit is built from dual ported on-chip BRAM memories and stores the state variables of the relevant mesh points. The minimal size of the on-chip memory is determined by the bandwidth of the adjacency matrix of the mesh. The bandwidth of a matrix is defined as the maximum distance

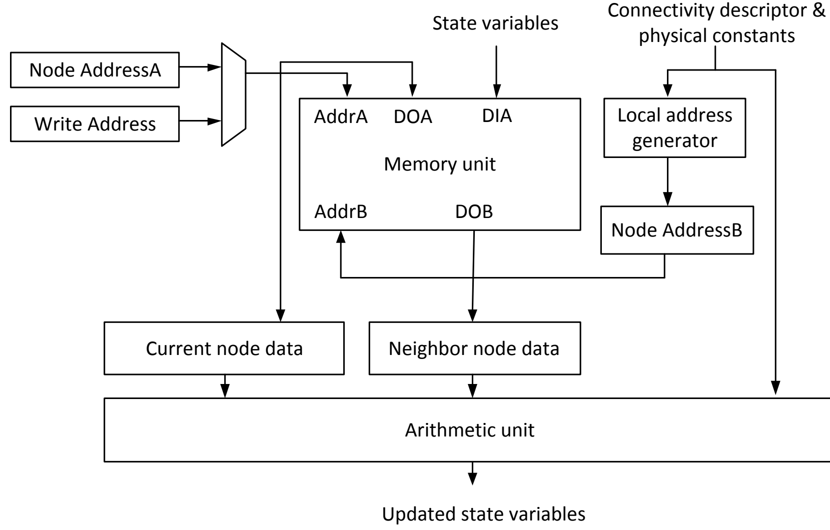

Figure 2: Block diagram of the proposed processor

of a nonzero element from the main diagonal (see Definition 1).

In the presented CFD problem each cell (triangle) has three interfaces and the state variables are updated based on a flux function computed at the three interfaces. (For mathematical formulation see Section 6.2.) In each clock cycle one interface of the given cell is evaluated by the AU, therefore, the new state variables can be computed in 3 or 4 cycles in case of 2D or 3D cell centered discretization, respectively. In the vertex centered discretization the length of the computation is determined by the degree of the given vertex.

Computation is started by loading serial sequence of state variables into the Memory unit until it is half filled. Global indices of the neighboring cells are translated into addresses in the Memory unit by the Local address generator. In this phase the state variables of the first cell are loaded into the Current node register and the Neighborhood node register is filled by the state variables of the first neighbor using the incoming connectivity descriptor. When all neighbors of the first cell are sent to the AU *Node AddressA* register is incremented and the state variables of the second cell are loaded into the Current node register. During the next clock cycle the state variables of a new cell data can be written into the Memory unit and computation of the second cell can be started. After an initial latency of the arithmetic unit the updated state variables are written back to the off-chip memory in the same sequential order as they were loaded. The Memory unit is operating as a circular buffer; when it is filled the oldest cell data is overwritten. This can be safely done because the size of the memory is set

to twice the bandwidth of the adjacency matrix, and the oldest values will not be required during the update of the remaining cells.

Descriptors add an overhead for the off-chip memory requirements and increase the memory bandwidth requirement of the processor. As the global index of the cells are never required and the order of cell data are statically scheduled the memory address translation can be done offline. In this case the shorter local addresses can be stored and transferred, which significantly decreases the memory bandwidth requirement of the processor.

3.3 Locally distributed control of the arithmetic unit

In the arithmetic unit the mathematical expression is implemented which describes the flux crossing the interface between two adjacent cells. All the input and output variables of the AU are stored in separate FIFO buffers. The AU is designed to operate independently from the rest of the processor and start the computation of a new interface in each clock cycle if all the inputs are available. In FPGA to design an efficient control unit (**CU**) to the AU the fanout of the control signals and the LUT depth of the control logic shall be minimized, otherwise the slow CU will hold down the operating frequency of the whole AU.

One straightforward way to control the AU is to implement a global CU which checks the states of the input FIFOs and schedules the operation of the FPUs. In this case every FPU is connected to the CU with a global enable signal which can start or stop the operation of the given FPU. Unfortunately, in this case the fanout of the control signal and the complexity of the control logic are too high to reach the desired operating frequency.

If the AU do not have feedbacks and accumulators, the enable signal can be neglected to decrease the complexity of the CU. Instead of halting the FPUs they are let to operate full time and the valid results are filtered out at the outputs of the AU. Filtering can be achieved by implementing an extra shift register in which the pipeline stages which hold valid data are distinguished. As the FPUs cannot be halted and the data goes through the AU once it has been read, the output FIFOs have to be checked whether they will be ready to store the results of the AU before the AU reads the inputs. This can be solved by adding extra virtual FIFOs (one per each output FIFO) whose lengths are set to the length of the corresponding output FIFO. The usage of extra shift register and the virtual FIFOs are demonstrated in Figure 3.

Before the operation starts the virtual FIFOs are empty indicating all the corresponding output FIFOs and the pipeline are empty. In every clock cycle if the inputs are ready to be read and the

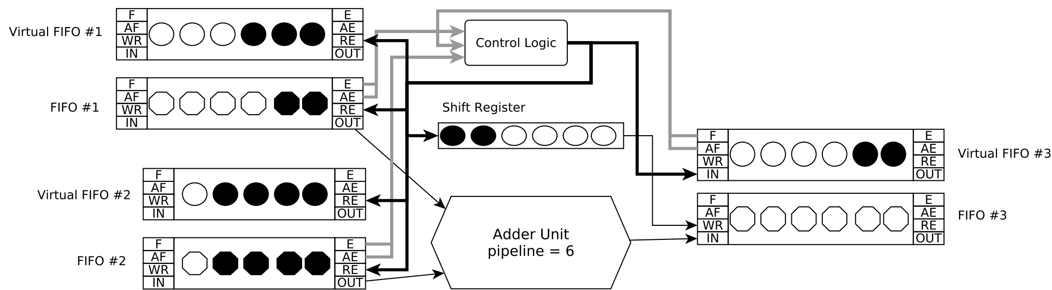

Figure 3: Usage of the shift register and the virtual FIFOs in case of a simple AU which contains only one adder FPU. Shift register is used to mark pipeline stages which hold valid data. In the example the first and second stages hold valid data. After 5 and 6 clock cycles the output of the shift register will write the results of the FPU to the output FIFO. Virtual FIFO #3 contains two elements indicating the two valid pipeline stages and will allow input data to enter the pipeline four more times if the output FIFOs is not read.

virtual FIFOs are ready to be written, new input is read from the input FIFOs and a bit is written to each virtual FIFO indicating the number of occupied elements in the pipeline and the corresponding output FIFO has been increased by one. The bit remains in the virtual FIFO as long as the data is in the pipeline or the corresponding result is in the corresponding output FIFO. This mechanism guarantees that every data which has entered the pipeline can be safely written out to the output FIFOs. To be able to read input into the pipeline in every clock cycle the size of the output and virtual FIFOs have to be at least the length of the pipeline.

Without enable signal the complexity of the CU can be significantly decreased and the fanout depends only on the number of I/O FIFOs of the AU. In case of 32 bit FPUs and simpler mathematical expressions the FIFOs can be placed close to each other in the FPGA and can be controlled at the desired frequency. However, in case of 64 bit FPUs or more complex expressions the area requirement of the AU significantly increases and the fanout of control signals and the placement of the I/O FIFOs become critical. To reach the desired operating frequency the FPUs shall be partitioned into separately controlled clusters which have smaller number of I/Os than the original AU. The partitioning problem can be described as the partitioning of the data-flow graph generated from the mathematical expression. If the arcs cut by the partitioning are replaced by FIFO buffers, the previously presented CU can be used for controlling each cluster. Unfortunately, the partitioning of the FPUs introduces other problems which have effects on both the area and the operating frequency of the circuit.

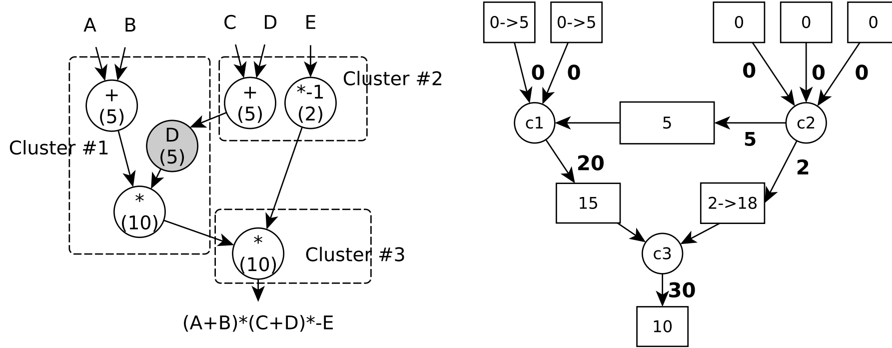

Figure 4: On the left a partitioned data-flow graph, while on the right the corresponding cluster adjacency graph is shown. In the data-flow graph pipeline length of the FPUs is displayed in brackets. In the cluster adjacency graph a FIFO (indicated by rectangle) is added to every cut arc. In cluster 1 an extra delay shift register has to be added to keep the proper data timing inside the cluster. The length of the FIFO between cluster 2 and 3 has to be set to 18 instead of 2, otherwise cluster 2 cannot operate continuously.

First of all, the added synchronizing FIFOs explicitly increase the area requirement of the circuit which makes efficient placements of the AU more challenging. To minimize the area requirements the number of cut arcs shall be minimized.

To explain the second problem a partitioned data-flow graph and the corresponding *cluster adjacency graph* are shown in Figure 4 where clusters are represented by vertices and the connections between the clusters (cut arcs) are represented by arcs. The position (level) of an arc in the pipeline is defined as the first clock cycle when partial results computed from the first inputs reach the given arc. In the cluster adjacency graph the levels of the arcs and the size of the added FIFOs are also displayed. If a cluster (e.g. cluster 3) has two inputs (two inward cut arcs) which have different levels, that is, the partial results reach the given cluster in two different routes, the data arriving at the shorter route has to be stored until other parts of the input arrive. In the presented example if the length of the FIFO between cluster 2 and 3 was only 2, the operation of cluster 2 would be paused after every 2 successful reads for 18 clock cycles, which is the time needed for the data to reach cluster 3 via the other route. The size of a FIFO at a given input arc shall be set at least to the level of the highest level input arc minus the level of the given arc to guarantee continuous operation. The size of the introduced FIFOs and the overall pipeline length of the AU heavily depend on the partitioning, therefore, an ideal data-flow graph partitioner shall avoid clusters which have big differences in the levels of the incoming

arcs. Unfortunately, common partitioning algorithms, which minimize the number of cut arcs, cannot target this objective. In the illustration the minimum size of the FIFOs is given, however in practice these values are rounded up to the nearest number which is integer power of 2 because they can be implemented more efficiently in FPGA.

The third problem is that the partitioning can create directed cycles in the cluster adjacency graph (mutually dependent clusters) even if the data-flow graph is acyclic. As a cluster reads new input only if all of its input FIFOs are ready to be read mutually dependent clusters will never start reading and cause a deadlock in the AU. Unfortunately, common partitioning algorithms do not have mechanism to avoid mutually dependent clusters.

In the paper a novel partitioning algorithm is presented, which addresses all of these problems to create a partition which can be converted to a fast locally controlled AU.

3.4 Outline of the multi-core architecture

High-level block diagram of the proposed architecture is shown in Figure 5. The memory interface provides the physical interface for the off-chip memory and the arbitration between the DMA engines competing for the memory. The sequential off-chip memory access pattern is a great advantage of the architecture because the off-chip memory can be accessed with optimal burst length and the penalties of random access patterns can be eliminated. The DMA engines load the time dependent (states) and time independent (descriptors and constants) data into the corresponding input FIFOs of the processor in long sequential bursts, and the computed new state values are also written back to the off-chip memory in long sequential bursts.

State of the system is usually saved after computing hundreds of explicit timesteps during the computation, therefore the results of the first iteration can be fed directly into a second processor which computes the second iteration. The second processor must wait until its Memory unit is half filled to start computation. The results of the second iteration can be either saved into the off-chip memory or fed into another processor. As the time independent data do not change between the iterations an additional FIFO is enough to store the previously loaded descriptors and constants between the two computation stages. The depth of the FIFO is determined by the size of the Memory unit and the length of descriptors. The number of implemented processors is only limited by the available logic and memory resources on the FPGA. After a short initial startup latency, which is negligible compared to the number of mesh cells, the pipelined chain of processors can work in parallel providing linear speedup without increasing memory bandwidth requirements.

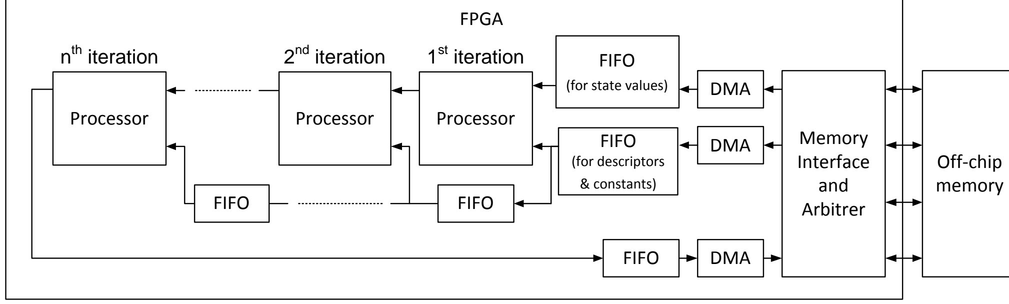

Figure 5: Outline of the proposed architecture. The processors are connected to each other in a chain to provide linear speedup without increasing memory bandwidth requirements. The number of processors is only limited by the available resources on the given FPGA.

4 Memory bandwidth optimization

Feasibility of the architecture described in the previous section mainly depends on the proper updating order of the nodes. Initial numbering of the nodes depends on the mesh generation algorithm, however the node data should be reordered to utilize data locality. In this section the relation between the Matrix Bandwidth Minimization (**MBM**) and the minimal size of the on-chip memory is established. After a brief summary on the field of MBM an algorithm is presented, which reorders the node data to decrease the on-chip memory requirements. In case of large meshes, when the algorithm cannot decrease the size of the required on-chip memory to fit the resources available at the given FPGA, an extension of the algorithm can be used to partition the mesh into smaller parts which can be handled using the available on-chip memory.

4.1 Description of the problem and related work

Definition 1 *Let $G(V, E)$ be a graph with vertex set V ($|V| = n$) and edge set E. **Labeling** is a function f which exclusively assigns an integer from interval $[1, n]$ to each vertex, that is, $f(v) = f(u)$ if and only if $u = v$ and $u, v \in V$. Let N(v) denote the set of vertices which are adjacent to v. The **bandwidth of a vertex** v respect to labeling f is $B_f(v) = Max\{|f(v) - f(u)| : u \in N(v)\}$, while the **bandwidth of a graph** G respect to labeling f is $B_f(G) = Max\{B_f(v) : v \in V\}$. The **bandwidth of a matrix** is the maximum distance of a nonzero element from the main diagonal. The bandwidth of a graph G respect to labeling f equals to the bandwidth of the adjacency matrix of G, if the vertices in the adjacency matrix are indexed according to labeling f.*

The problem of bandwidth reduction of a sparse matrix was shown to be NP-complete by Papadimitriou [16], so exact methods can not be applied for large problems. Many heuristic algorithms were developed from the well-known Cuthill-McKee (CM) algorithm [17] to recent metaheuristic approaches. One of the most promising metaheuristic regarding solution quality is the GRASP (Greedy Randomized Adaptive Search Procedure) with Path Relinking [18]. The most practical solution is the GPS(Gibbs, Poole and Stockmeyer) method [19], which is one of the fastest heuristics, and also provides good solution quality. Metaheuristic methods give better solutions, but their runtime is many times higher than runtime of the GPS. The GPS algorithm was born in 1976, afterwards many attempts were made to improve the original method, e.g. Luo et al. [20]. Most of the improved GPS heuristics have higher time complexity, which can be an important parameter in case of large meshes which arise in practical applications.

To design a Memory unit which can store the data of the neighboring nodes of each node when the given node is processed, the minimal size of the on-chip memory (**LM**) shall be determined as

$$LM = BW * size\ of\ a\ node\ data$$

where **BW** is the minimum number of nodes to be stored in the on-chip memory:

$$BW = (B_f(G) * 2 + 1)$$

To decrease the on-chip memory requirements the bandwidth of the adjacency matrix of the mesh shall be minimized.

4.2 Algorithm for bandwidth reduction

Several methods have been shown in the literature for MBM, in this section we define Amoeba1 (AM1) algorithm for bandwidth minimization. Our goal is to create a fast, constructive method where BW bound can be easily and accurately estimated in each step of the algorithm.

4.2.1 Notations and definitions

Given an unstructured mesh which contains nodes (points in case of vertex centered discretization, triangles or tetrahedrons in case of cell centered discretization), the task of AM1 is to reorder the nodes into a different sequence to minimize BW. Before the description of the algorithm a snapshot of its operation and the necessary notations are illustrated in Figure 6. Solution part **P** denotes the set of the already ordered nodes and is illustrated by a line segment to express the order of the nodes.

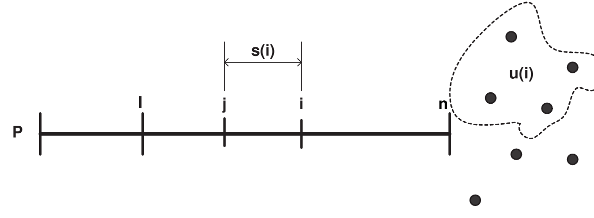

Figure 6: Structure of solution part P.

s(i): is the difference between the indices of node **i** and of its lowest indexed neighbor in P:

$$s(i) = MAX\{i.index - j.index : j \in N(i)\}$$

u(i): is the set of nodes which are uncovered by **P**, but must be added in later steps before node **i** is processed:

$$u(i) = \{v : v \in N(i) \text{ and } v \notin P\}$$

I: is the index of the first element having a nonempty u set:

$$I = MIN\{X \in \mathbb{N} : \forall k \in P,\ k.index < X \text{ and } u(k) = \emptyset\}$$

imp(i): is the importance of node **i** defined as:

$$imp(i) = (n - i.index) + |u(i)| + s(i)$$

where n is the index of the last node in P. This is the minimum BW which is required to process node i; it shows how important is to select a node from the neighborhood of **i**.

4.2.2 Description of Amoeba1

Amoeba1 algorithm has two main steps: in the first stage a starting vertex is selected, and the nodes are relabeled in the second stage. The result is a reordered array of nodes and adjacency list.

Finding a starting vertex The quality of the result of constructive bandwidth-reduction heuristics depends on the starting vertex. In the GPS method, the authors present a simple and effective solution for this problem. They give an algorithm which returns the two endpoints of a pseudo-diameter [19]. During the first step of the AM1 algorithm the same subroutine is used to find the starting vertex.

Choosing a solution element AM1 selects a node from u(node(I)), which has a neighbor in P with maximal importance. Only $l \neq node(I)$ neighbors take part in the inner search loop, because each node in u(node(I)) has node(I) as a neighbor.

```
   // selecting candidate from u(node(I))
1  candidate = random element of u(node(I))
2  global_max = 0
3  for ∀k ∈ u(node(I))
4      local_max = 0
5      for ∀l ∈ N(k) : l ∈ P and l ≠ node(I)
6          if imp(l) > local_max
7              local_max = imp(l)
8      if local_max > global_max
9          candidate = k
10         global_max = local_max
```

AM1 adds the candidate to the part with index=n+1, and chooses the next element till the whole mesh is indexed. AM1 performs a kind of breadth-first indexing.

4.2.3 Optimization results

Efficiency of the constructive AM1 algorithm is tested on relatively large meshes and compared to the classical fast and efficient GPS method. More sophisticated algorithms are also presented in the literature for bandwidth reduction [18][20], however quality of results is usually not proportional to their time complexity when they are applied to large meshes ($\geq 100,000$ *node*). Optimization results on 2-dimensional meshes are shown on Table 1. Cell centered discretization is used and the degree of the nodes is 3. For these low-degree cases solution quality of AM1 is similar to GPS, while runtime is 4% shorter. BW value of the 2-dimensional meshes can be significantly reduced by AM1 algorithm. Only 1% of the node data have to be stored on-chip, to eliminate random read and write accesses to the external memory.

The AM1 algorithm was also tested on a mesh generated around a complex 3D geometry using vertex centered discretization. Average degree of the nodes is around 14, while the highest degree is 30. Results of the optimization are shown in Table 2. The GPS algorithm is found to be 29% superior on 3D meshes, but 13% slower than AM1. The difference between the two algorithms is not increasing with the complexity of the mesh.

Table 1: Results of Amoeba1 method compared to GPS.

Case	N	BW GPS	BW AM1	GPS time(s)	AM1 time(s)
step_2d_bc_cl30	7063	125	127	0,078	0,052
step_2d_bc_cl40	12297	183	175	0,154	0,109
step_2d_bc_cl50	20807	257	231	0,175	0,1
step_2d_bc_cl70	42449	363	345	0,633	0,49
step_2d_bc_cl90	68271	487	517	0,998	0,785
step_2d_bc_cl110	112093	575	595	2,144	1,955
step_2d_bc_cl130	157099	747	747	1,59	1,316
step_2d_bc_cl150	201069	805	817	3,239	3,094
step_2d_bc_cl170	252869	985	935	4,316	3,92
step_2d_bc_cl190	316715	1037	1089	5,913	5,707
step_2d_bc_cl200	394277	1101	1159	5,855	5,532
step_2d_bc_cl320	930071	1931	1821	17,035	18,687

BW: min. number of nodes to be stored in local memory

N: number of nodes in problem.

Algorithms tested on one core of an Intel P8400 2.26GHz processor

In case of 3D meshes, 5-10% of the nodes should be stored in the on-chip memory, but this ratio is decreasing when the number of nodes is increased. Depending on the number of state and constant values 10,000-40,000 nodes can be stored on the FPGA, therefore the size of the largest mesh which can be handled by our architecture is in the 100,000-400,000 node range. Practical applications are computed on a mesh containing several millions of nodes, therefore the AM1 method is extended to generate a partitioned mesh where the bandwidth of each partition is limited.

Table 2: Results for 3D high-degree cases

Case	N	BW-GPS	BW-AM1	GPS time(s)	AM1 time(s)
3d_075	3652	385	411	0,279	0,27
3d_065	5185	505	781	0,144	0,107
3d_055	8668	769	813	0,655	0,587
3d_045	15861	1097	1521	0,468	0,42
3d_035	33730	1891	1893	2,209	2,22
3d_025	88307	3473	3529	7,569	6,42
3d_018	244756	6661	10245	39,509	27,598
3d_015	417573	9163	14985	85,797	59,958
3d_012	519983	20663	23675	413,72	383,075

4.3 Memory Access Optimization for Bounded Bandwidth

In case of large problems($\geq 500,000$ *node*) it is possible that the bandwidth of the renumbered mesh is larger than the available on-chip memory; these cases should be handled, too. In this section we show an AM1 based method, which generates an input order which has at most a pre-specified BW. In the access pattern every node is updated and written once but can be loaded many times when it has a neighbor in another partition too. These vertices are called ghost nodes and indicated by a flag called **Ex**, which is false for ghost nodes and true for nodes inside the partition. Node data are still stored in the order of execution, and the access pattern appears in the descriptors. Additionally, a ghost node descriptor is required to help gathering of the ghost nodes and hide read latency by prefetching them. Results can be still written in sequential order, however, data dependency exists between the ghost nodes. Therefore old state values are preserved and new state values are saved into a different memory location and the role of each memory part is changed after every iteration.

4.3.1 AM1 based bounded BW method

The main concept of handling bounded bandwidth is to use the BW estimation of the AM1 method. When a partition reaches the BW bound, the process calculates which vertices can be updated and call the AM1 method for the rest of vertices where Ex is not true. The main process starts new partition until Ex is true for all vertices. The output of the method is an access pattern (list of {index,Ex} pairs), where every vertex can appear as a ghost node several times, but only once with a true execute flag. By using this list the nodes can be reordered and the ghost node descriptors can be filled.

Estimation of BW: Given an AM1 Part, the task is to estimate its BW value. In each step a node is added to the part from u(node(I)), where I is the lowest index for which u(node(I)) is not an empty set. Every time when index I is changed, the following equation have to be evaluated:

$$BWBound \geq MAX\{s(node(I)), (n-I) + |u(node(I))|\} * 2 + 1 \qquad (1)$$

New nodes are added to the part P when Eq. 1 holds, otherwise the part is finalized and a new instance of AM1 is started on the rest of the not updated nodes.

Finalizing a Part: During finalization the vertices which have been updated in part P are labeled. Furthermore, the vertices of which all neighbors have been updated are also labeled, because these nodes can be cut out of the mesh (called perfect nodes, Pr=true). Perfect nodes will not appear in later parts, but Ex=true and Pr=false vertices have to be loaded again, because they have at least

one Ex=false neighbor. s^* is the index of the first node, which has a neighbor which will be updated in the following parts:

$$s^* = \underbrace{MIN}_{I \leq k.index \leq n} \{k.index - s(k)\}$$

// setting Ex and Pr flags

1 **for** $\forall k \in P$

2 **if** $k.index < s^* \quad and \quad k.Ex \,! = \; true$

3 $k.Pr = true$

4 **if** $k.index < I$

5 $k.Ex = true$

In AM1 imp(i) is set to zero for each node(i) which has true Ex flag.

4.3.2 Bandwidth limited results

It is obvious that the proposed algorithm generates access patterns which has lower BW than a given bound. The quality of the solution can be measured by the node reload factor **k**, which is defined by the ratio of the length of the access pattern and the number of nodes.

Table 3: Results of AM1 bounded bandwidth optimization

Case	AM1_BW	BW Bound	num. of parts	N	overall length	k	time(s)
3d_075	411	412	1	3562	3562	1	0,264
3d_075	411	400	3	3562	4489	1,26	0,485
3d_075	411	300	8	3562	5241	1,471	0,932
3d_075	411	200	15	3562	5899	1,656	1,247
3d_035	1893	1894	1	33730	33730	1	2,367
3d_035	1893	1800	3	33730	37952	1,125	6,979
3d_035	1893	1500	5	33730	39987	1,185	7,14
3d_035	1893	1000	15	33730	44439	1,317	9,523
3d_015	14985	14986	1	417573	417573	1	76,88
3d_015	14985	14000	2	417573	430693	1,031	79,869
3d_015	14985	10000	3	417573	439136	1,052	154,14
3d_015	14985	7500	7	417573	452510	1,084	68,43
3d_015	14985	5000	21	417573	483391	1,158	59,01
3d_015	14985	2500	97	417573	577474	1,383	1170,8

AM1_BW: the bandwidth provided by AM1 for the whole mesh
overall length: length of the generated access pattern
N: number of vertices

Measurements on three meshes with different BW bounds can be found in Table 3. The results

show that the BW value of large meshes can be reduced more effectively (with better k factors), with BW_Bound=0.5*AM1_BW, we get k={1.65, 1.31, 1.08}. BW_Bound is determined by the on-chip memory capabilities of the FPGA, which increases with every new generation of the technology; furthermore, the ratio becomes better for larger problems.

5 Data-flow graph partitioning

5.1 Properties of a good partition

The aim of the partitioning of the data-flow graph is to find a good partition of the FPUs, in which each cluster can be controlled by a simple CU and the clusters can be connected together to form a fast AU in the FPGA. In Section 3.3 the main motivation behind the partitioning and the possible side-effects were presented. Before the description of the proposed algorithm we review the necessary properties of a good partition in terms of graph partitioning. Furthermore, an extra property is presented which is needed to efficiently place and route the AU in the FPGA.

1. **The number of I/O connections of each clusters is bounded by a user defined constant**
   To implement fast CU the complexity of the CU has to be decreased which depends on the number of I/O connections.

2. **The number of cut arcs is minimal**
   Every cut arc is replaced by a synchronization FIFO, which increases the area requirement of the circuit.

3. **The input arcs of each cluster are roughly on the same pipeline level**
   Clusters which have input arcs on different pipeline levels require larger synchronization FIFOs to guarantee the continuous operation and can increase the overall pipeline length of the AU.

4. **There is no directed cycle in the cluster adjacency graph**
   Mutually dependent clusters never start to read input data and cause a deadlock in the AU.

5. **The clusters can be mapped to the FPGA without long interconnection between the clusters**
   Mapping elements of a circuit description to FPGA is a 2D placement problem, where the routing resources are limited. In the implemented circuit high fanout and long interconnections should be avoided, otherwise they limit the operating frequency of the whole circuit. In a previous

work [21] it was demonstrated that common graph partitioning algorithms blindly create partitions in which the clusters cannot be mapped to the FPGA without long interconnections. To reach significant speedup in the operating frequency of the AU both the partitioning problem and the placement of the clusters should be solved. The operating frequency could be further increased if the placement of the clusters were explicitly set by using the Xilinx physical constraints (pblocks), however, the presented algorithm provides significant speedup even without the physical constraints.

In the paper both steps (placement and partitioning) of the previous algorithm have been improved and replaced by simulated annealing. In case of partitioning a novel representation is presented, which is more convenient to define objective functions for targeting the properties of a good partition. Instead of maximizing the speedup by using physical constraints, our motivation is to investigate how the described properties of the circuit and the free parameters of our algorithm affect the performance of the AU.

5.2 Algorithm

The main idea of the algorithm is to combine the partitioning and the placement objectives in a two-step procedure. In the first step an initial and simplified floorplan of the FPUs is created with simulated annealing to minimize the distance between the connected FPUs. In the second step the floorplanned FPUs are partitioned by another simulated annealing to find a good partition with the previously described properties. The resulting clusters can be easily placed on the FPGA and the lack of long interconnections results in a high operating frequency.

5.2.1 Preprocessing and Layering

The mathematical expression to be implemented is described in a text file, where inputs, outputs and internal variables are also defined. In the first step the input file is parsed and a data-flow graph representation of the mathematical expression is created. Every mathematical operator is represented by a vertex and has an associated delay which will be the pipeline latency of the corresponding IP core in the implemented circuit.

In the next step a *layering* [22] is performed, in which the data-flow graph is converted to a special bipartite graph. In this bipartite graph every vertex is associated to a layer and each arc directs immediately to the next layer.

Definition 2 $L = \{l_1, l_2, ...\}$ *layering is a partition of the* V *vertices of the* $G(V, E)$ *directed graph*

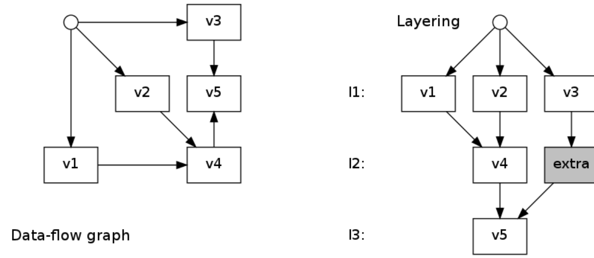

Figure 7: A simple data-flow garph and its layered version

such that

$$\forall(u, v) \in E \quad and \quad u \in l_i \rightarrow v \in l_{i+1}.$$

A layering can be generated via a breadth-first-search in linear time [22] by splitting up the arcs, which span more than one layer, with extra delay vertices. An example layering is shown in Figure 7. The layering of the graph is an artificial restriction on the placement and the partitioning of the graph. Vertices can only be moved horizontally during the placement, and the representation of the clusters also depends on the structure of the layers. Evidently, this restriction can leave out the optimal solution from the search space, however, it significantly decreases the representation costs of the placement and the partitioning and makes the application of simulated annealing possible. Unfortunately, the complexity of the original problem requires some expandable restrictions, otherwise the problem cannot be handled.

Fortunately, the layering has several other benefits beside the simplification. First of all, if the vertices had the same pipeline lengths, the horizontal cutting would guarantee that the incoming arcs of a cluster have the same pipeline level. In spite of that the pipeline lengths are different in practice, horizontal cutting combined with the cut arcs minimization produces acceptable results and does not increase the overall pipeline length drastically (see Table 4). The second benefit of the layering is that a simple and sufficient criterion can be formulated to check the existence of deadlocks during the simulated annealing. (see Theorem 1)

Finally, in physical implementation extra delay vertices are implemented as shift registers (extra vertices inside one cluster are joined), which hold the data for the proper number of clock cycles. From the aspect of performance it is advantageous because smaller interconnections help to keep the timing requirements.

5.2.2 Floorplan with simulated annealing

During the floorplan vertices are horizontally positioned to minimize the length of the interconnections and to prepare the partitioning phase.

In the framework a simplified homogeneous floorplan is used where every vertex has a unit width, however, the principles used during partitioning can be adapted to floorplans where the size of the different resource types are distinguished. The blocks in one layer are represented by their sequence which is the 1D version of the famous sequence pair representation [22]. This representation is appropriate for ASIC floorplanning, however, in our case empty spaces can be favorable inside the design, therefore place holder (*bubble*) vertices have been introduced. To distinguish the bubble vertices they are indicated by negative indices. To limit the complexity of the problem the number of the bubble vertices on a layer is limited.

The floorplan is initialized by a random sequence of the vertices of the first layer. Vertices of the rest of the layers are computed by the Barycentre heuristic [23], which is a frequently used method to decrease the number of the edge crossings in layered digraphs.

During each move of the simulated annealing the sequence of the vertices of a layer is perturbed. Layers are selected with probability proportional to the number of vertices(N_l) they contain. In the selected layer(l) a vertex is selected with probability $(1/(N_l+1))$ or a bubble is created with probability $(1/(N_l+1))$. If a normal vertex is selected, it will be swapped with one of its neighbor. If a bubble vertex is selected, its size will be increased or decreased by one. If a bubble with size one is selected for decreasing, it will be deleted.

The linear combination of the following three objective functions is minimized during the simulated annealing:

1. **Total squared distance (TSD) of the connected vertices**

   This objective minimizes the distance between the connected vertices. As the clusters are determined based on the position of the vertices, this objective automatically avoids long interconnections between the clusters. Distance between two vertices are determined according to their horizontal coordinates:

   $$distance(A,B) := \begin{cases} (x_A - x_B)^2 & \text{if A and B are connected} \\ 0 & \text{otherwise} \end{cases}$$

   where x_A and x_B are the horizontal coordinates of vertex A and B respectively. Vertical coordinates can be neglected because the distance is always one as the graph is layered. Manhattan distance cannot be applied: if vertex A is connected to two other vertices which are relatively

far from each other it only guarantees that vertex A will be somewhere between the two vertices but not at the middle of the interval.

2. **Maximum distance (MDV) between connected vertices**

   This objective is used beside the TSD to put an extra pressure on the longest interconnection because usually the longest interconnection has the largest delay limiting the operating frequency.

3. **Maximum distance (MDI) between vertices which get input from a common vertex**

   If the fanout of the output data signal of a vertex is larger because the vertex supplies data to several other vertices it is practical to put the target vertices close to each other. In this case the fanout of the data signal can be tolerable and the partitioning phase can put the target vertices into the same cluster to decrease the number of the cut arcs.

The result of the simulated annealing in case of the presented CFD problem is shown in Figure 10. In our experiments the following coefficients were used in the global objective function: TSD=0.2, MDV=3, MDI=6. To get a practical solution usually 1K-10K iterations have been executed.

5.2.3 Novel representation for graph partitioning

The input of the partitioning is a layered digraph where the horizontal coordinates of the vertices are already set. To force horizontal cutting the user can group the layers into *belts* before the partitioning starts and partitions on each belt are represented separately. The height of the layers should be set according to the complexity and the pipeline length of the given mathematical expression. In our experiments the belts have been 2 or 3 layers high.

The main idea of the proposed representation is that vertices inherit their affiliation (cluster ID) from a neighboring vertex or are assigned to a new cluster. In this case clusters of the represented partitions can be covered by continuous non-overlapping regions. If the placed vertices are connected with short interconnections another simulated annealing can be used to find a partition in which each cluster forms a continuous non-overlapping region in the 2D plane and is only connected to the neighboring clusters (*locally connected*). The length of the connections between the vertices should be shorter than the width of the clusters.

In our case the vertices have uniform size and the direction of the inheritance can be described with a *spin* associated to every vertex. In case of variable size vertices spins cannot describe all the possible partitions which have continuous clusters, therefore other descriptors should be used beside the spins (e.g. visiting order of the vertices). The possible spin values are the following:

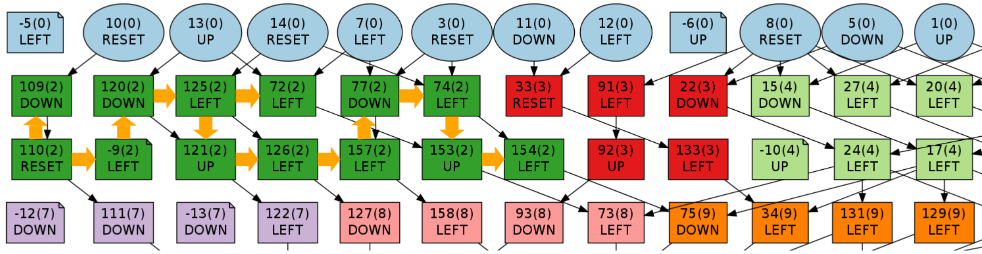

Figure 8: A fragment of the first belt of Figure 10 is shown to demonstrate how inheritance works.

- LEFT : Vertex will inherit cluster ID from the left neighbor.
- DOWN : Vertex will inherit cluster ID from the bottom neighbor.
- UP : Vertex will inherit cluster ID from the upper neighbor.
- RESET : Vertex will be assigned to a new cluster.

For an example how inheritance works see thick arrows and cluster 2 in Figure 8.

The vertices of a belt are assigned to *columns* based on their horizontal position. Two vertices are assigned to the same column if and only if they have the same horizontal position. In a worst-case situation the number of the columns in a belt is equal to the number of the vertices of the belt.

When a partition is built up from a representation the columns of the given belt are visited from left to right. In each step all the vertices of a column are assigned to clusters. First, the vertices of the given column are visited from top to down and each vertex inherits its cluster ID according to its spin. If the inheritance is ambiguous, the vertex is not clustered. If unclustered vertices remain in the column, the vertices are revisited in bottom-up order. During the second visit of a vertex the IDs are also associated based on the spins, however, if the inheritance is ambiguous, the vertex is associated to a new cluster. In the worst case all the vertices are visited twice, therefore the partition can be built up in $O(N)$ steps, where N is the number of the vertices on the given belt.

One of the main benefits of the layering and the belts is that directed cycles induced by partitioning can only occur inside the belts, which can be easily controlled.

Theorem 1 *Assuming a layered data-flow graph in which the belts are partitioned separately, if there is any directed cycle in the cluster adjacency graph all the vertices of the directed cycle must belong to the same belt.*

For a proof it shall be observed that if an arc connects two clusters which are in different belts, it is always directed toward the lower belt.

5.2.4 Partitioning

The initial partition is built up based on random spins associated to every vertex. To avoid *meaningless spin directions* the spins of the vertices which are at the belt boundaries are not allowed to direct outward the belt.

In each iteration of the simulated annealing one belt is selected with a probability proportional to the number of vertices they contain. In the belt one vertex is selected randomly and its spin is perturbed, however meaningless spin directions are not allowed. In the next step the partition on the selected belt is rebuilt and the linear combination of the following objective functions is used to compute the energy function of the simulated annealing.

1. **Total number of cut arcs (TNC)**

   According to the properties of a good partition the number of cut arcs should be minimized.

2. **Total I/O penalty of the clusters (TIOP)**

   The I/O penalty of cluster C is defined as

   $$f_{IOP}(C) = \begin{cases} f_{IO}(C) - T_{user} & \text{if } f_{IO}(C) > T_{user} \\ 0 & \text{otherwise} \end{cases}$$

   where $f_{IO}(C)$ denotes the number of the I/O connections of cluster C and T_{user} is the user defined threshold to limit the I/O connections of each cluster.

3. **Number of clusters (NC)**

4. **Number of connections between non-neighboring clusters which are on the same belt (NCNN)**

   To find a partition in which clusters are only connected to their neighbors NCNN should be zero.

5. **Number of mutual dependencies between neighboring clusters which are on the same belt (NMD)**

   If both NCNN and NMD are zero then no directed cycle can be present in the belts. According to Theorem 1 this means that the partition is deadlock free.

If the new partition is not accepted, the perturbed spin is reverted and the partition on the selected belt is rebuilt.

The result of the partitioning is shown in Figure 10. In our experiments the following coefficients were used to compute the energy function: $C_{TNC} = 1$, $C_{TIOP} \in \{8..13\}$, $C_{NC} = 2$, $C_{NCNN} = 18$, $C_{MDI} = 10$. To get a practical solution usually 10K-100K iterations have been executed.

6 Case study: Finite volume solver for the Euler equations

In this section we describe the numerical method which we have selected as a test case from the field of Computational Fluid Dynamics (CFD) to demonstrate the previously presented algorithms and our accelerator architecture. The general concept of simulation of the CFD problems is the approximation of a continuous model by a discrete one, in which a huge amount of data have to be processed and manipulated. The art of CFD is how to exploit the enormous processing power available by recent computer technology for these computations.

6.1 Fluid Flows

A wide range of industrial processes and scientific phenomena involve gas or fluids flows over complex obstacles, e.g. air flow around vehicles and buildings, the flow of water in the oceans or liquid in BioMEMS. In such applications the temporal evolution of non-ideal, compressible fluids is quite often modeled by the system of Navier-Stokes equations. The model is based on the fundamental laws of mass-, momentum- and energy conservation, including the dissipative effects of viscosity, diffusion and heat conduction. By neglecting all non-ideal processes and assuming adiabatic variations, we obtain the Euler equations [24, 25], which describe the dynamics of dissipation-free, inviscid, compressible fluids. They are a coupled set of nonlinear hyperbolic partial differential equations, in conservative form expressed as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{2a}$$

$$\frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot \left(\rho \mathbf{v}\mathbf{v} + \hat{I} p \right) = 0 \tag{2b}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot ((E + p)\mathbf{v}) = 0 \tag{2c}$$

where t denotes time, ∇ is the Nabla operator, ρ is the density, u, v are the x- and y-component of the velocity vector $\mathbf{v}$, respectively, p is the thermal pressure of the fluid, $\hat{I}$ is the identity matrix, and E is the total energy density defined by

$$E = \frac{p}{\gamma - 1} + \frac{1}{2}\rho \mathbf{v} \cdot \mathbf{v}. \tag{2d}$$

In equation (2d) the value of the ratio of specific heats is taken to be $\gamma = 1.4$. For later use we introduce the conservative state vector $\mathbf{U} = [\rho, \rho u, \rho v, E]^T$, the set of primitive variables $\mathbf{P} = [\rho, u, v, p]^T$ and the speed of sound $c = \sqrt{\gamma p/\rho}$. It is also convenient to merge (2a), (2b) and (2c) into hyperbolic

conservation law form in terms of U and the flux tensor

$$\mathbf{F} = \begin{pmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\mathbf{v} + Ip \\ (E+p)\mathbf{v} \end{pmatrix} \tag{3}$$

as:

$$\frac{\partial U}{\partial t} + \nabla \cdot \mathbf{F} = 0. \tag{4}$$

6.2 Discretization of the governing equations

Logically structured arrangement of data is a convenient choice for the efficient operation of the FPGA based implementations [5]. However, structured data representation is not flexible for the spatial discretization of complex geometries. As one of the main innovative contributions of this paper, here we consider an unstructured, cell-centered representation of physical quantities. In the following paragraphs we describe the mesh geometry, the governing equations, and the main features of the numerical algorithm.

6.2.1 The geometry of the mesh

The computational domain Ω is composed of non-overlapping triangles. The i-th face of triangle $\mathcal{T}$ is labelled by f_i. The normal vector of f_i pointing outward $\mathcal{T}$ that is scaled by the length of the face is $\mathbf{n_i}$. The volume of triangle $\mathcal{T}$ is $V_{\mathcal{T}}$. Following the finite volume methodology, all components of the volume averaged quantities are stored at the mass center of the triangles.

6.2.2 The Discretization Scheme

Application of the cell centered finite volume discretization method leads to the following semi-discrete form of governing equations (4)

$$\frac{dU_{\mathcal{T}}}{dt} = -\frac{1}{V_{\mathcal{T}}} \sum_f \mathbf{F}_f \cdot \mathbf{n}_f, \tag{5}$$

where the summation is meant for all three faces of cell $\mathcal{T}$, and $\mathbf{F}_f$ is the flux tensor evaluated at face f. Let us consider face f in a coordinate frame attached to the face, in such a way that its x-axe is normal to f (see Fig. 9). Face f separates triangle L (left) and triangle R (right). In this case the $\mathbf{F}_f \cdot \mathbf{n}_f$ scalar product equals to the x-component of $\mathbf{F}$ (i.e. F_x) multiplied by the area of the face. In order to stabilize the solution procedure, artificial dissipation has to be introduced into the scheme. According to the standard procedure, this is achieved by replacing the physical flux tensor by the numerical flux function F^N containing the dissipative stabilization term. A finite volume scheme is

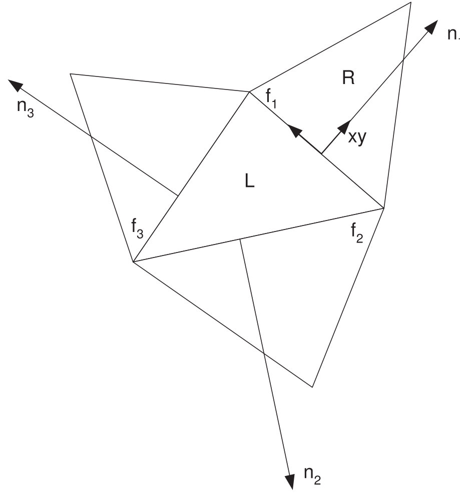

Figure 9: Interface with the normal vector and the cells required in the computation

characterized by the evaluation of F^N, which is the function of both U_L and U_R. In this paper we employ the simple and robust Lax-Friedrichs numerical flux function defined as

$$F^N = \frac{F_L + F_R}{2} - (|\bar{u}| + \bar{c})\,\frac{U_R - U_L}{2}. \tag{6}$$

In the last equation F_L=$\mathrm{F}_x(\mathrm{U}_L)$ and F_R=$\mathrm{F}_x(\mathrm{U}_R)$ and notations $|\bar{u}|$ and $|\bar{c}|$ represent the average value of the u velocity component and the speed of sound at an interface, respectively. The temporal derivative is discretized by the first-order forward Euler method:

$$\frac{dU_{\mathcal{T}}}{dt} = \frac{U_{\mathcal{T}}^{n+1} - U_{\mathcal{T}}^{n}}{\Delta t}, \tag{7}$$

where $U_{\mathcal{T}}^{n}$ is the known value of the state vector at time level n, $U_{\mathcal{T}}^{n+1}$ is the unknown value of the state vector at time level $n+1$, and Δt is the time step.

By working out the algebra described so far, leads to the discrete form of the governing equations to compute the numerical flux term F.

$$U_{\mathcal{T}}^{n+1} = U_{\mathcal{T}}^{n} - \frac{\Delta t}{V_{\mathcal{T}}} \sum_{f} \hat{\mathcal{R}}_{\mathbf{n_f}} F_f |\mathbf{n_f}|, \tag{8}$$

where $\hat{\mathcal{R}}_{\mathbf{n_f}}$ is the rotation tensor describing the transformation from the normal-parallel coordinate frame of face f to the $x-y$ frame. Quantity F_f is defined in a coordinate frame attached to face f, with such an orientation that state left is identical to the state of the update triangle $\mathcal{T}$ while state right corresponds to the state of the triangle situated at the opposite side of the face. With these conventions, the normal component of the numerical flux function is given by

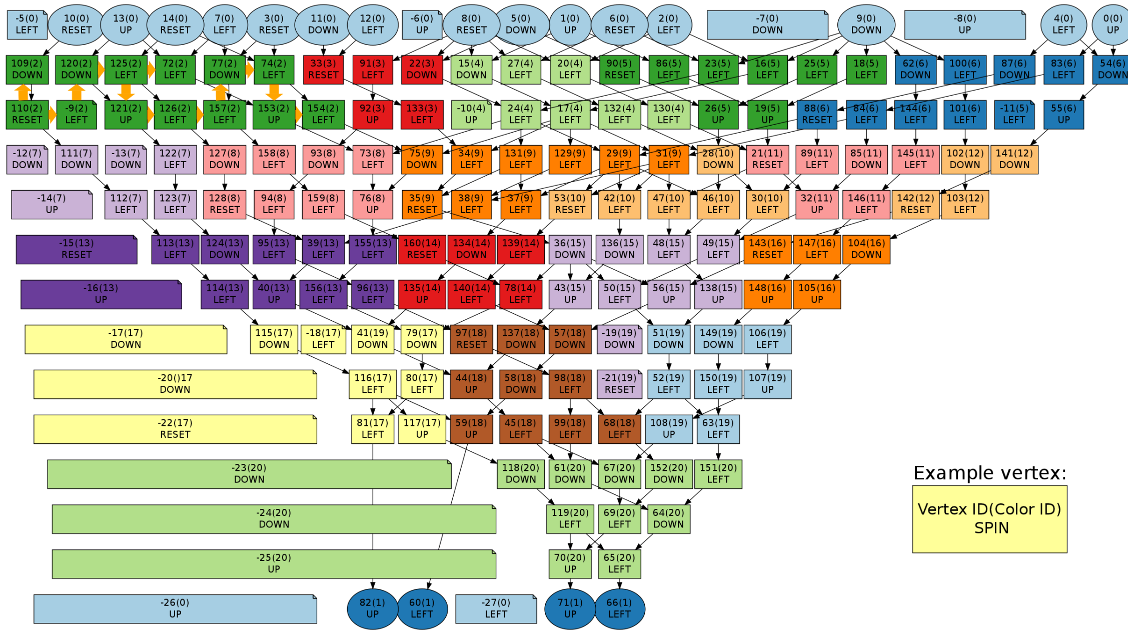

Figure 10: The partitioned data-flow graph generated from equations (9a) to (9d). Bubble vertices are indicated by negative indices.

$$F_f^{\rho} = \frac{\rho_L u_L + \rho_R u_R}{2} + (|\bar{u}| + \bar{c}) \frac{\rho_R - \rho_L}{2} \tag{9a}$$

$$F_f^{\rho u} = \frac{\left(\rho_L u_L^2 + p_L\right) + \left(\rho_R u_R^2 + p_R\right)}{2} + (|\bar{u}| + \bar{c}) \frac{\rho_R u_R - \rho_L u_L}{2} \tag{9b}$$

$$F_f^{\rho v} = \frac{\rho_L u_L v_L + \rho_R u_R v_R}{2} + (|\bar{u}| + \bar{c}) \frac{\rho_R v_R - \rho_L v_L}{2} \tag{9c}$$

$$F_f^{E} = \frac{(E_L + p_L)\, u_L + (E_R + p_R)\, u_R}{2} + (|\bar{u}| + \bar{c}) \frac{E_R - E_L}{2} \tag{9d}$$

The AU is generated from these equations (9a to 9d) and is designed to execute the computation of the normal component of the numerical flux function for a new interface in each clock cycle. Beside the AU an additional simple arithmetic unit is required to update conservative state variables $(\rho, \rho u, \rho v, E)$ of the cells using the flux vectors. As a cell has three interfaces three clock cycles are required for a complete cell update.

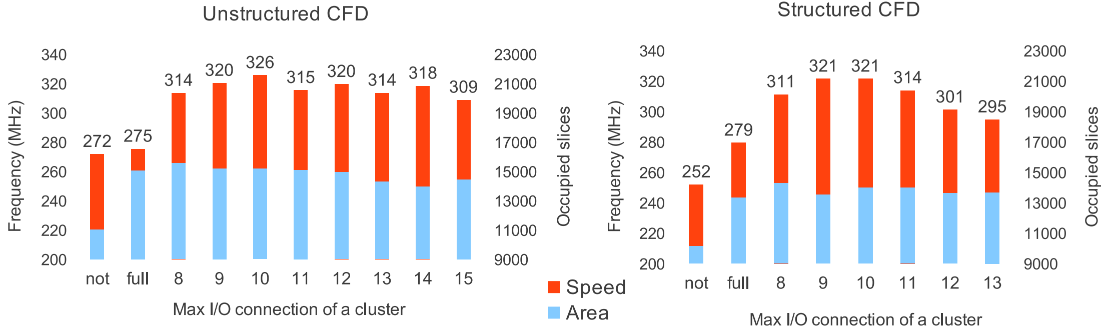

Figure 11: Operating frequency and area requirements of the AU as the maximum number of I/O connections of a cluster is changing.

7 Results and performance

7.1 Partitioning results

Using the automatic VHDL generation the effects of the T_{user} parameter (which limits the number of I/Os of a cluster) of the partitioning algorithm have been investigated in case of two examples. The first example is an AU related to a structured CFD problem [5], while the second example is the unstructured CFD problem presented in the previous section. In both cases the unpartitioned version and the fully partitioned version of the AU have been implemented as references. In the unpartitioned version one CU is assigned to the whole circuit, while in the fully partitioned version each FPU is controlled by a separate CU. The resulting operating frequencies and area requirements are summarized in Figure 11, while partitioning and implementation details are in Table 4 and 5.

In the structured CFD example two interfaces can be computed in the AU requiring 23 input variables, 4 output variables and 44 FPUs (14 multipliers and 30 adders). The large number of I/Os of the AU results in slow operating frequency in the unpartitioned case, however, if the I/Os of the clusters are limited to 9 or 10, the operating frequency can be improved by 15-27%.

In the presented unstructured CFD example only one interface is computed in the AU requiring 15 inputs, 4 outputs and 64 FPUs (30 multipliers, 25 adders, 9 specials). Special FPUs include FPUs which is used for negation or division by 2. The latter one is implemented by modifying the exponent of the floating point numbers. Similarly to the previous example, the largest speedup (18-19%) can be reached if the number of I/O connections of the clusters is limited to 10, although clusters with slightly larger I/O connections can also run at high operating frequency. In the following paragraphs

Table 4: Partitioning and implementation results of the structured CFD graph.

				Cluster I/O threshold (T_{user})					
		Unpartitioned	Fully-partitioned	8	9	10	11	12	13
Num. of clusters		1	44	19	100	128	143	100	103
Extra delay vertex		26	0	33	32	32	37	28	31
Pipeline length		100	100	128	100	128	143	100	103
Max cluster cut		27	6	8	9	10	11	12	13
Cut arcs	Outside	27	46	39	38	38	35	35	35
	Inside	0	46	56	56	52	52	49	49
	Total	27	92	95	94	90	87	84	84
FIFO	32	23	68	63	72	60	47	65	60
	64	0	18	26	22	27	36	19	24
	128	4	6	6	0	3	4	0	0
	TOTAL	27	92	95	94	90	87	84	84
Area	FF	41,664	48,591	51,544	51,137	50,775	50,787	49,641	49,783
	LUT	31,384	37,297	40,022	38,337	38,869	39,387	36,583	37,397
	DSP	244							
Frequency (MHz)		251,889	279,33	311,139	321,44	321,44	313,775	301,114	294,638
Improvement		100%	110,89%	123,52%	127,61%	127,61%	124,57%	119,54%	116,97%
			100%	111,39%	115,08%	115,08%	112,33%	107,80%	105,48%

the fastest AU (325.627MHz) is used for performance evaluation.

7.2 Implementation results

The generated arithmetic unit is implemented on our AlphaData ADM-XRC-6T1 reconfigurable development system [26] equipped with a Xilinx Virtex-6 XC6VSX475T FPGA [8] and 2Gbyte on-board DRAM in four 32bit wide banks running on 800MHz providing 12.8Gbyte/s peak theoretical bandwidth. During implementation memory interface and Open Core Protocol (OCP) infrastructure IP cores, which are available in the AlphaData SDK, are used to connect our architecture to the host system and the on-board memory. On-board memories can be accessed via four 128bit wide OCP channels running on 200MHz clock frequency to match the speed of the memories. The user design is running at a much higher 325MHz clock frequency. Synchronization between the two clock domains is performed by the FIFOs between the processors and the DMA engines. On the user side the four memory OCP channels are merged into a 512 bit wide datapath which is shared between three DMA engines (two for reading and writing state values and one for reading descriptors and constants) by using adb3_ocp_mux_nb IP cores which use round-robin arbitration.

The architecture is implemented using the standard Xilinx floating-point cores, therefore the width of the mantissa can be selected from the 4-64 bit range. Using nonstandard mantissa width an optimal

Table 5: Partitioning and implementation results of the unstructured CFD graph.

				Cluster I/O threshold (T_{user})							
		Unpartitioned	Fully-partitioned	8	9	10	11	12	13	14	15
Num. of clusters		1	64	32	26	21	20	19	17	14	13
Extra delay vertex		37	0	43	46	50	49	48	48	50	51
Pipeline length		165	165	179	198	201	199	199	183	183	197
Cut arcs	Outside	19	44	26	25	25	24	25	21	22	20
	Inside	0	79	88	79	76	74	72	68	63	64
	Total	19	123	114	104	101	98	97	89	85	84
Max cluster cut		27	6	8	9	10	11	12	13	14	15
FIFO	32	15	101	82	71	67	60	66	66	63	55
	64	0	12	29	30	30	31	27	23	22	26
	128	2	6	3	3	4	7	4	0	0	3
	256	2	4	0	0	0	0	0	0	0	0
	TOTAL	19	123	114	104	101	98	97	89	85	84
Area	FF	43,966	55,606	58,052	56,980	56,830	56,393	56,139	54,977	54,477	54,566
	LUT	33,043	43,035	42,950	42,419	42,254	42,615	41,723	39,846	39,563	40,126
	DSP	405									
Frequency (MHz)		271,655	275,331	313,513	320,307	325,627	315,457	319,591	313,578	318,407	308,833
Improvement		100%	101,35%	115,41%	117,91%	119,87%	116,12%	117,65%	115,43%	117,21%	113,69%
			100%	113,87%	116,34%	118,27%	114,57%	116,08%	113,89%	115,65%	112,17%

architecture can be found in the space of implementation area, computing performance and solution accuracy. Automatic generation of the arithmetic unit brings up the possibility to use different mantissa width for each operator in the arithmetic unit or to implement a fused-datapath where floating-point numbers are not normalized between successive operations [27] [28]. Examination of these possibilities is beyond the scope of the recent paper because its application requires detailed roundoff error analysis. Therefore in the following paragraphs the most accurate double precision case is examined.

The time independent data structure consists of a 8 byte constant containing the area of the cell and three 26 byte wide descriptors for the three interfaces. The descriptor of an interface consists of a 2 byte address of the neighboring cell and three 8 byte variables representing the x and y coordinates of the normal vector of the face and its length. The size of the address of the neighboring cell allows to address 65K elements, which is more than the size of the memory unit which we can implement in a multi-processor case. The size and normal vector of each face are precomputed, which slightly increases the memory bandwidth requirement, but a simpler arithmetic unit can be implemented.

The time dependent data structure consists of four 8 byte state variables $[\rho, \rho u, \rho v, E]$. Beside the state variables pressure p and local speed of sound c are also stored in the Memory unit and computed

only once for each cell. To provide the 48 byte wide memory bus of the Memory unit 11 Xilinx Block_RAMs (BRAMs) should be used in a 36×512 configuration. When all BRAMs are allocated for the Memory unit 98,816 nodes can be stored on the FPGA, which is usually more than enough for practical 2D meshes.

When multiple processors are used in a pipeline to speed up computation, connectivity descriptors and constants should be saved into a FIFO as shown in Figure 5. Three 26 byte wide descriptors and the 8 byte wide constant must be saved for each node stored in the Memory unit of the processors. For a 512 element deep FIFO 20 additional BRAMs should be allocated. Therefore the number of nodes stored in the Memory units altogether is reduced to 34,816 in a multiprocessor configuration.

Computation of each new state value requires loading and storing of one state variable vector, loading of the area of the triangle and loading of three connectivity descriptors which are 150 byte altogether. Therefore a 16.3Gbyte/s memory bandwidth is required to feed the processor with valid data in every third clock cycle. This bandwidth cannot be provided on our prototyping board and the system will be memory bandwidth limited. This limitation can be removed by slightly modifying the architecture shown in Figure 5 and connecting two Memory units to one Arithmetic unit creating two virtual processors. One Memory unit is enabled in even clock cycles while the other is enabled in odd clock cycles and input data is provided to the arithmetic unit alternately. As a result, three additional clock cycles are available to load and store node data to the off-chip memory because the second memory unit uses values computed previously on the FPGA. Therefore memory bandwidth requirement of the processors is effectively halved to a manageable 8.2Gbyte/s level.

Area requirements of the implemented system and the utilization of the Virtex-6 SX475T FPGA on our prototyping board are summarized in Table 6. Utilization of the FPGA shows that the most

Table 6: Area requirements of the whole architecture

	DSP	LUT	FF
Number of elements	525	43754	61936
XC6VSX475T utilization	26%	14.7%	10.4%

limiting factor of the implementation is the number of DSP48E slices and the number of implementable processors is three. In case of three processors maximum bandwidth of the adjacency matrix of the mesh is 14,848 nodes, however, to avoid memory bandwidth bottleneck on our prototyping board three AUs and six memory units have been implemented reducing the maximum bandwidth to 6,144 nodes.

Performance of our architecture is determined using the result of the post place and route static timing analysis which is indicated 325MHz operating frequency. Three clock cycles are required to

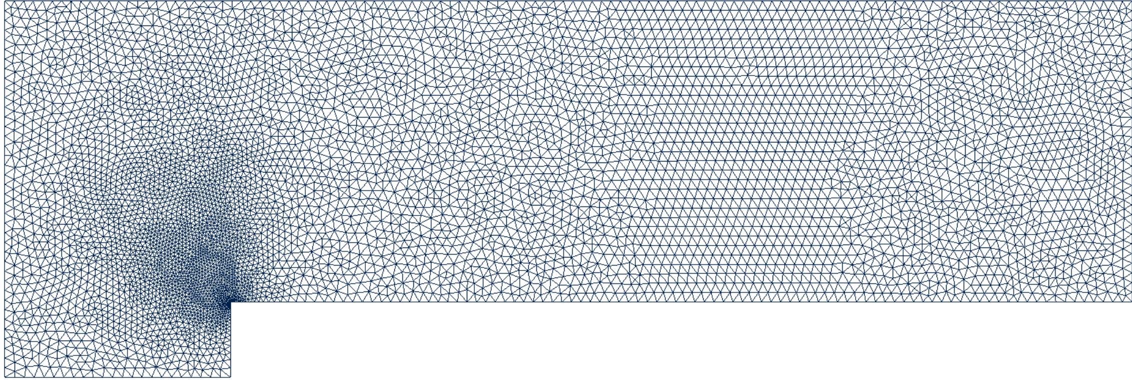

Figure 12: Coarse resolution mesh for the forward facing step test case

update the state of one triangle, therefore the performance of one processor is 108.3million triangle update/s. Computation of one triangle requires 213 floating point operations, therefore the performance of our architecture is 23.08GFLOPs. On the Virtex-6 XC6VSX475T FPGA three arithmetic units can be implemented and connected in a pipeline resulting in 69.22GFLOPs cumulative computing performance.

7.3 Test setup

To show the efficiency of our solution a complex test case was used, in which a Mach 3 flow over a forward facing step was computed. The simulated region is a two dimensional cut of a pipe which is closed at the upper and lower boundaries, while the left and right boundaries are open. The direction of the flow is from left to right and the speed of the flow at the left boundary is 3 times the speed of sound constantly. The solution contains shock waves reflecting from the closed boundaries.

Unstructured mesh for the domain is generated by the freely available Gmsh mesh generator [29]. Several meshes are generated using different characteristic lengths between $1/30 - 1/200$ and the characteristic length of the elements near the corner of the step is divided by 12.5. The number of triangles in the resulting meshes is ranging in the 7,063-394,277 interval. An example mesh generated by using 1/40 characteristic length and containing 12,297 triangles is shown in Figure 12.

The result of the computation on the largest simulated grid after 4s of simulation time is shown in Figure 13. The required timestep is 56.81ms and 70,400 iterations are computed to get the result. Reference solution for the previous problem computed by the more accurate residual distribution upwind scheme can be found in [30]. Using double precision numbers 12.6Mbyte memory should be allocated for the initial state of the simulation and 30.7Mbyte for the connectivity descriptors.

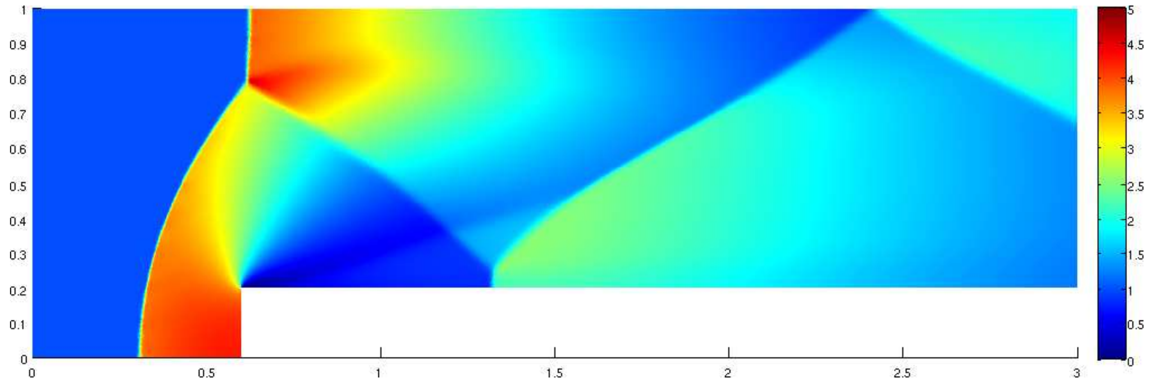

Figure 13: First-order solution of the Mach 3 flow after 4s of simulation time on a 394,277 triangle mesh

Communication time to download initial state and connectivity descriptors into the on-board memory from the host memory is 69.61ms and the result can be read in 20.25ms via the x8 PCI-Express interface of the board. Computation of the result using three arithmetic units is performed in 71.1s therefore communication between the host and the FPGA can be neglected.

7.4 Performance comparison

The performance of our architecture is compared to both a high performance Intel Xeon E5620 microprocessor and an Nvidia GeForce GTX 570 graphics card. During comparison various mesh sizes are used with 7,063 to 394,277 triangles and the simulation is carried out with and without reordering the triangles.

In case of the microprocessor a two-socket server board was used which had two Xeon E5620 processors, each having 4 cores running on 2.4GHz clock frequency. Via Intel hyper-threading technology 16 threads were available for the simulation but one last thread was reserved for the OS. The simulator was implemented in C language using the OpenMP application program interface. The performance of the simulator using 1, 2, 4 and 8 threads is shown in Figure 14. As we expected, without reordering the performance is decreasing as the mesh size is increased, while reordering preserves the performance. In case of the largest mesh and 8 threads the reordered case outperforms the original one by 28.2% and reaches 33.22 million triangle update/s or equivalently 6.86GFLOPs.

The average performance of the simulator over various mesh sizes is shown in Figure 15 to investigate the speedup caused by the increasing number of threads. The average performance scales well with the number of threads in both the reordered and the original case, however in both cases the speedup

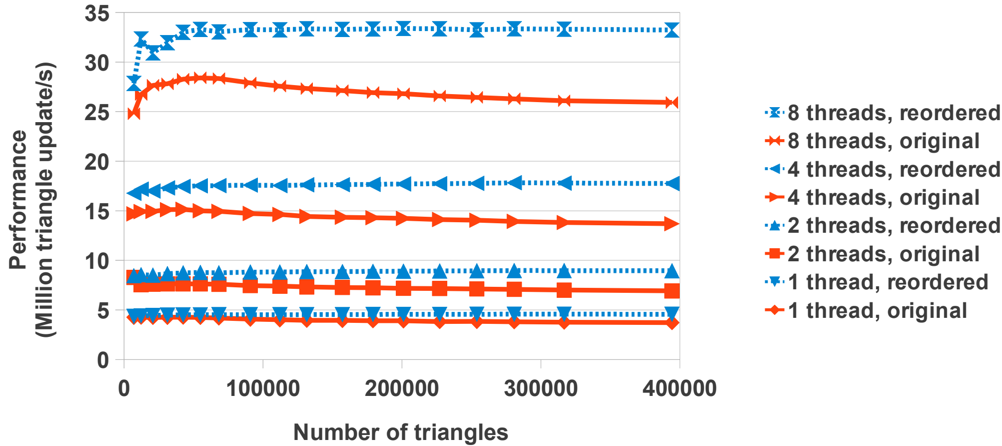

Figure 14: Measured performance of Intel Xeon E5620 microprocessor using 1, 2, 4 and 8 threads

compared to a single thread remains below the number of threads. Using 8 threads the speedup is approximately 6.7 and 7.2 in case of the reordered and the original input, respectively. Scaling breaks after 8 threads which is in agreement with the fact that there are only 8 physical cores in the system, however, hyper-threading technology can slightly further increase the performance. Using 15 threads the speedup is 8.6 and 8.5 in case of the reordered and the original input, respectively.

The code used for testing the performance of a GPU architecture is implemented in OpenCL and is a customized version of an already published GPU application [31] solving a similar numerical problem. The code was kindly given by the author and the measurements were performed on a Nvidia GeForce GTX 570 graphics card, which has 480 cores running on 1464 MHz frequency, and 1280 MB GDDR5 memory with 152 GB/s bandwidth. The GPU program consists of a simple framework and a kernel, which computes a full triangle update. The performance of the GPU architecture is shown in Figure 16. In case of the largest mesh the application with reordered input outperforms the original input by 48% and reaches 108.12 million triangle update/s or equivalently 23.02GFLOPs.

In case of both reference architectures (CPU, GPU) using sufficiently large meshes (> 0.2 million triangles) the applications with reordered input produced a stable performance independent of the actual mesh size. Comparison of the performance of the FPGA with the two reference cases shows that the FPGA can compute 71.28 times faster than a single core of an Intel Xeon processor, 10.09 times faster than 8 cores of two Intel Xeon processors, and approximately 3 times faster than a Nvidia GTX 570 graphics card.

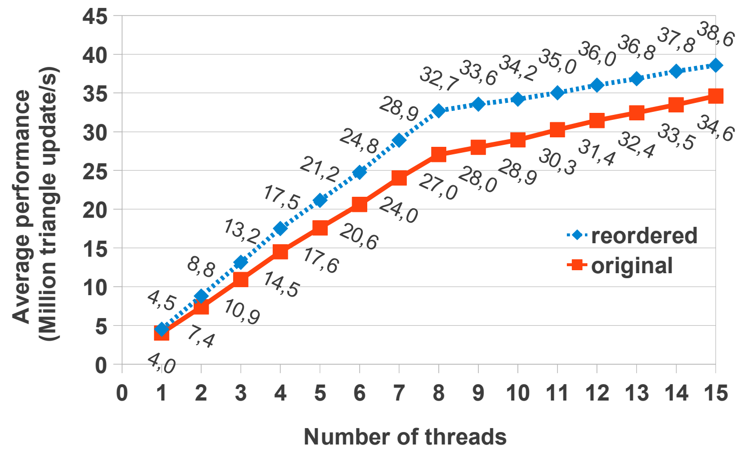

Figure 15: Measured average performance of Intel Xeon E5620 microprocessor over various mesh sizes using different number of threads

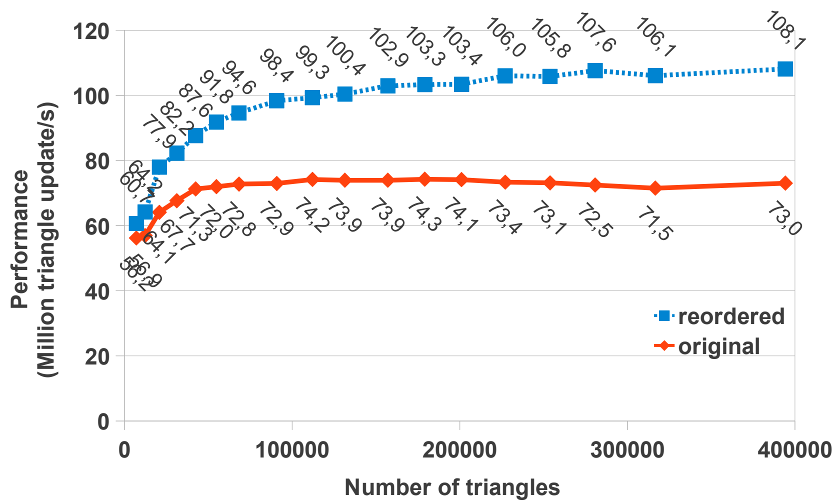

Figure 16: Measured performance of NVidia GTX570 GPU

To make a fair comparison between the architectures not only the performance but the development time has to be compared. Both the CPU and the GPU application can be developed approximately in one week, while the development of FPGA applications usually takes significantly longer. The great advantage of the architecture proposed in the paper is the flexibility. To develop the VHDL code of the architecture took approximately 3 months, however, using the automatic AU generation framework presented in the paper the architecture can be easily adopted to a different CFD problem. The VHDL code of the AU of the new CFD problem can be generated in a couple of hours, and a couple of days are needed to adjust the memory interface and to place and route the new design.

8 Conclusions

A framework for accelerating the solution of PDEs using explicit unstructured finite volume discretization is presented. Irregular memory access patterns can be eliminated by using the proposed memory structure which results in higher available memory bandwidth and full utilization of the arithmetic unit. Efficient use of the on-chip memory is provided by a new node reordering algorithm which can be extended to generate fixed bandwidth partitions. The new algorithm is comparable to the well known GPS algorithm in both runtime and quality of results.

Generation of the application specific arithmetic unit is described by using a complex numerical problem solving the Euler equations. The discretized state equations are automatically translated to a synthesizable VHDL description using Xilinx floating-point IP cores. Performance of the arithmetic unit is improved by using partitioning and local control units. Partitioning is based on a preliminary placement of the vertices of the data-flow graph which helps placement of the partitions and clock frequency of the design is improved.

The final architecture contains three AUs connected into a pipeline and operates at 325MHz resulting in 69.22GFLOPs performance. Performance comparison showed 10 times speedup compared to a high performance microprocessor and 3 times speedup compared to a high performance GPU.

Currently the size of the mesh is limited by the bandwidth of its adjacency matrix, which must be smaller than 100,000 in case of a Virtex-6 FPGA. The architecture should be improved to efficiently handle multiple partitions and extended to use multiple FPGAs during computation.

Acknowledgments

This research project supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences, TAMOP-4.2.1./B-10, TAMOP-4.2.1./B-11, OTKA Grant No. K84267 and in part by OTKA Grant No. K68322. Furthermore, we are very grateful for helpful discussion and the GPU implementation to Endre László.

References

[1] James P. Durbano, Fernando E. Ortiz, John R. Humphrey, Petersen F. Curt, and Dennis W. Prather. FPGA-Based Acceleration of the 3D Finite-Difference Time-Domain Method. In *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, volume 0, pages 156–163, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[2] Chuan He, Wei Zhao, and Mi Lu. Time Domain Numerical Simulation for Transient Waves on Reconfigurable Coprocessor Platform. In *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, volume 0, pages 127–136, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[3] P. Sonkoly, I. Noé, J. M. Carcione, Z. Nagy, and P. Szolgay. CNN-UM based transversely isotropic elastic wave propagation simulation. In *Proc. of 18th European Conference on Circuit Theory and Design, 2007 (ECCTD 2007)*, pages 284–287, 2007.

[4] K. Sano, T. Iizuka, and S. Yamamoto. Systolic Architecture for Computational Fluid Dynamics on FPGAs. In *Proc. of the 15th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, volume 0, pages 107–116, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[5] S. Kocsárdi, Z. Nagy, Á. Csík, and P. Szolgay. Simulation of 2D inviscid, adiabatic, compressible flows on emulated digital CNN-UM. *International Journal on Circuit Theory and Applications*, 37(4):569–585, 2009.

[6] Cs. Nemes, Z. Nagy, M. Ruszinkó, A. Kiss, and P. Szolgay. Mapping of high performance data-flow graphs into programmable logic devices. In *Proceedings of International Symposium on Nonlinear Theory and its Applications, (NOLTA 2010)*, pages 99–102, 2010.

[7] G. Karypis and V. Kumar. HMETIS 1.5: A Hypergraph Partitioning Package. Technical report, Department of Computer Science, 1998. http://www-users.cs.umn.edu/~karypis/metis.

[8] Xilinx Inc. http://www.xilinx.com/, 2012.

[9] M.T. Jones and K. Ramachandran. Unstructured mesh computations on CCMs. *Advances in Engineering Software*, 31:571–580, 2000.

[10] M. deLorimier and A. DeHon. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 75–85, 2005.

[11] Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk, and Jack Snoeyink. Streaming computation of delaunay triangulations. *ACM Trans. Graph.*, 25(3):1049–1056, July 2006.

[12] Y. Elkurdi, D. Fernández, E. Souleimanov, D. Giannacopoulos, and W. J. Gross. FPGA architecture and implementation of sparse matrix-vector multiplication for the finite element method. *Computer Physics Communications*, 178:558–570, 2008.

[13] D. Dubois, A. Dubois, T. Boorman, C. Connor, and S. Poole. Sparse Matrix-Vector Multiplication on a Reconfigurable Supercomputer with Application. *ACM Transactions on Reconfigurable Technology and Systems*, 3(1):2:1–2:31, 2010.

[14] K.K. Nagar and J.D. Bakos. A Sparse Matrix Personality for the Convey HC-1. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 1–8, may 2011.

[15] I. S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15:1–14, March 1989.

[16] C.H. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[17] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the ACM National Conference, Association for Computing Machinery, New York*, pages 157–172, 1969.

[18] E. Pinana, I. Plana, V. Campos, and R. Marti. GRASP and path relinking for the matrix bandwidth minimization. *European Journal of Operational Research*, 153(1):200–210, 2004.

[19] N.E. Gibbs, W.G. Poole, and P.K. Stockmeyer. An algorithm for reducing the bandwidth and profile of sparse matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, 1976.

[20] J.C. Luo. Algorithms for reducing the bandwidth and profile of a sparse matrix. *Computers and Structures*, 44:535–548, 1992.

[21] Cs. Nemes, Z. Nagy, and P. Szolgay. Efficient Mapping of Mathematical Expressions to FPGAs: Exploring Different Design Methodologies. In *Proceedings of the 20th European Conference on Circuit Theory and Design, (ECCTD 2011)*, pages 750–753, 2011.

[22] A.B. Kahng, J. Lienig, I.L. Markov, and J. Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer, Jul. 2011.

[23] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for Visual Understanding of Hierarchical System Structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109 –125, Feb. 1981.

[24] J. D. Anderson. *Computational Fluid Dynamics - The Basics with Applications*. McGraw Hill, 1995.

[25] T. J. Chung. *Computational Fluid Dynamics*. Cambridge University Press, 2002.

[26] Alpha Data Inc. http://www.alpha-data.com/, 2012.

[27] M. Langhammer and T. VanCourt. FPGA Floating Point Datapath Compiler. In *17th IEEE Symposium on Field Programmable Custom Computing Machines, 2009. FCCM'09.*, pages 259–262, 2009.

[28] F. de Dinechin and B. Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design and Test of Computers*, 28(4):18–27, 2011.

[29] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79:1309–1331, 2009.

[30] Á. Csík and H. Deconinck. Space-time residual distribution schemes for hyperbolic conservation laws on unstructured linear finite elements. *International Journal for Numerical Methods in Fluids*, 40:573–581, 2002.

[31] E. Laszlo, P. Szolgay, and Z. Nagy. Analysis of a gpu based cnn implementation. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on*, pages 1–5, 2012.