Accelerating unstructured finite volume computations on field-programmable gate arrays

Zoltán Nagy [*]   Csaba Nemes [†]   Antal Hiba[†]   Árpád Csík [‡]   András Kiss[*†]

Miklós Ruszinkó [§]   Péter Szolgay[*†]

Abstract

Accurate simulations of various physical processes on digital computers requires huge computing performance, therefore accelerating these scientific and engineering applications has a great importance. Density of programmable logic devices doubles in every 18 months according to Moore's Law. On the recent devices around one hundred double precision floating-point adders and multipliers can be implemented. In the paper an FPGA based framework is described to efficiently utilize this huge computing power to accelerate simulation of complex physical spatio-temporal phenomena. Simulating complicated geometries requires unstructured spatial discretization which results in irregular memory access patterns severely limiting computing performance. Data locality is improved by mesh node renumbering technique which results in predictable memory access pattern. Additionally storing a small window of node data in the on-chip memory of the FPGA can increase data reuse and decrease memory bandwidth requirements. Generation of the floating-point data path and control structure of the arithmetic unit containing dozens of operators is a very challenging task when the goal is high operating frequency. Long and high fanout control lines and improper placement can severely affect computing performance. In the paper an automatic data path generation and partitioning algorithm is presented to eliminate long delays and aid placement of the circuit. Efficiency and use of the framework is described by a case study solving the Euler equations on an unstructured mesh using finite volume technique. On the currently available largest FPGA the generated architecture contains three processing elements

[*]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, Péter Pázmány Catholic University

[‡]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

working in parallel providing 90 times speedup compared to a high performance microprocessor core.

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 requires expensive supercomputing facilities. Several previous studies proved the efficiency of programmable logic devices in numerical simulation of various physical phenomena such as electromagnetic [1], transient wave [2] [3] and computational fluid dynamics [4] [5] simulations.

A complex spatio-temporal problem can be approximated in a regular mesh structure. To get a more accurate solution the the resolution of the mesh should be increased, which increases the number of grid points and the running time as well. To overcome the outlined problem an unstructured mesh will be used. The resolution of the mesh will be increased where it is required by rapid change in the dynamics or shape of the problem. Conventional microprocessors has around 10% utilization during unstructured mesh computations due to the irregular memory access pattern of grid data.

To accelerate the computation irregular memory access patterns should be hidden by temporally storing the relevant grid points on the FPGA. Due to the finite on-chip memory size mesh points have to be ordered and stored according to the requirements of computation. A very efficient new mesh point ordering method was developed.

To obtain a high performance accelerator the operating frequency of the arithmetic unit is critical. By using local controls to clusters of arithmetic operators the clock speed can be significantly increased paying with some area increase for it. The data transfer between the clusters are synchronized by FIFOs. It was proofed that a good tradeoff can be achieved if a cluster has about 10 I/O FIFOs. To define the clusters several partitioning algorithms [6] [7] were carefully tested and compared. The placement and the routing of the clusters however were not as good as it should be. Especially due to the long interconnections the clock frequency could not increased further. To eliminate this kind of problem a new combined the clustering and placement method will be shown and tested in a complex fluid dynamical problem.

In Section 2. an overview of recent publications on accelerators working on unstructured meshes are given. A new architecture for efficient unstructured mesh computations is proposed in Section 3. A new node reordering algorithm is presented in Section 4. and a new partitioning algorithm is

descried in Section 5. The proposed methods are tested on a complex case study described in Section 6. Finally performance of the proposed architecture is compared to a high performance microprocessor in Section 7.

2   Related work

Several papers are published in the early 2000s dealing with the acceleration of PDEs on unstructured meshes. Most of them are focused on accelerating Finite Element Methods (FEM) where the global stiffness matrix is given and the resulting large linear system of equations are 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 papers 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 arising in our case such as 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 [8] 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 is computed by the CPU and only the matrix vector multiplication is performed on the FPGA and achieved $22.4 - 35.7$MFLOPs computing performance.

Another approach is the architecture proposed by deLorimier and DeHon [9] where all elements of the matrix are stored on the FPGA to avoid bandwidth limitations but this solution severely limited the size of the matrix. Performance of the architecture is depended on the structure of the matrix and usually 66% of the peak performance can be achieved which results in 2-10 times acceleration compared to the common microprocessors at that time.

Elkurdi et al. [10] first reorganize the finite element sparse matrix into a banded form then calculate the matrix vector multiplication along special pipelineable diagonal stripes, where two successive elements can be processed by a pipelined architecture. Performance of the architecture is determined by the available memory bandwidth and the sparsity of the matrix however utilization of the processing elements is varying in a very wide range between $17.74 - 86.24\%$.

duBois et al. [11] 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 suffer from memory bandwidth limitation.

Recently Nagar et al. [12] proposed an architecture using an optimized Streaming Multiply- Accu-

mulator 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 PEs is around 50% similarly to the previous architectures and increasing the number of PEs 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 small communication to computation ratio of sparse matrix vector multiplication. On the other hand utilization of the execution units depends on the structure of the sparse matrix.

In the case of finite volume discretization irregular memory access pattern and high memory bandwidth requirement can be eliminated by storing a small part of the grid on-chip and reordering its nodes. Right hand side of the discretized equations should be computed for each node in every timestep, which requires several floating-point operations, resulting in better communication to computation ratio and higher utilization of FPGA resources.

3   Architecture

Time evolution of dynamical systems can be easily solved numerically by explicit finite volume discretization and the solution is computed on a discrete set of points in space. The solution is advanced in time by approximating the derivative of each node using linear combination of state values from its small local neighborhood. Computation of subsequent grid points is independent however some parts of its input data sets are overlapped. For practical applications the number of grid points is far exceeding the available on-chip memory of recent FPGAs therefore state and constant values must be stored and loaded from an off-chip memory. Overlapping input data sets can be utilized to reduce the number of memory accesses and save memory bandwidth by saving all nodes which required during the following computations into a local memory on the FPGA. Simple example for the values to be stored is shown in Figure 1.

Each node is numbered and the nodes are updated in an increasing order, bold numbers indicate the already processed nodes, encircled node 8 is the currently processed vertex, while squared nodes are stored on the FPGA. In this case all elements required for the computation of node 8 can be read from the fast on-chip memory of the FPGA. To process the next node a new node 15 should be loaded and node 2 can be discarded. When updating node 9 all the required data will be available in the on-chip memory. It is possible that multiple new nodes are required for the update of a node
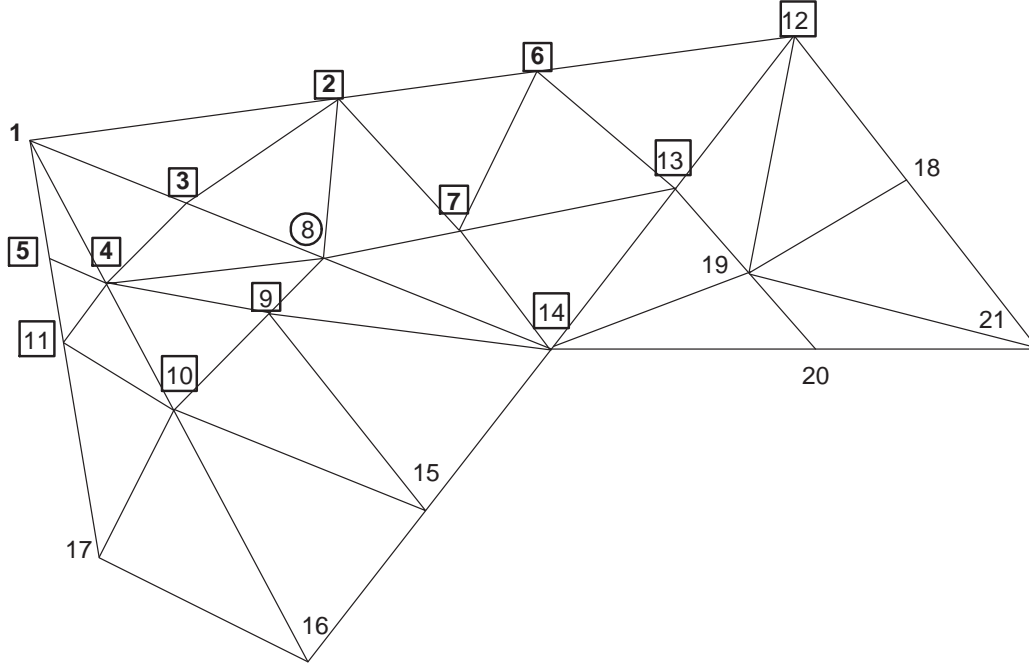
Figure 1: Local node data stored on the FPGA

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 nodes.

Node data is treated as a 1D array where each element contains two separate parts one for storing time dependent state values and one to store constant data such as physical constants and coordinates of the grid points. In case of structured grids neighbors of each grid point can be determined by its array index. For unstructured grids connected grid points are described by a sparse adjacency matrix which is usually stored in a Compressed Row Storage (CRS) format [13]. In our case the matrix contains only 0 and 1 elements therefore only column indices are stored. Additionally the elements are read in a serial sequence hence row pointers can be replaced by a single bit to indicate the start of a new row. It is possible that the discretization stencil is defined on triangle or tetrahedron instead of a line between two grid points. In this case an additional element descriptor is required where node indices of the vertexes of the element are stored. Data structures used by the system are shown in Figure 2. Additionally an example data structure is filled with connectivity and element data for node 8 from Figure 1. Width of the index field can be set according to the requirements of the application. More than 8 million nodes can be handled by using 24 bit wide indices which can be sufficient in many applications.

Node data:

| ... | (state var, ...), (const, ...) | ... |

Connectivity descriptor: next node bit

| ... | 1 | 14 | 0 | 2 | 0 | 3 | 0 | 4 | 0 | 7 | 0 | 9 | 1 | 14 | 0 | 4 | ... |

Element descriptor:

| ... | 1 | 8 | 14 | 0 | 2 | 3 | 0 | 3 | 4 | 0 | 2 | 7 | 0 | 4 | 9 |

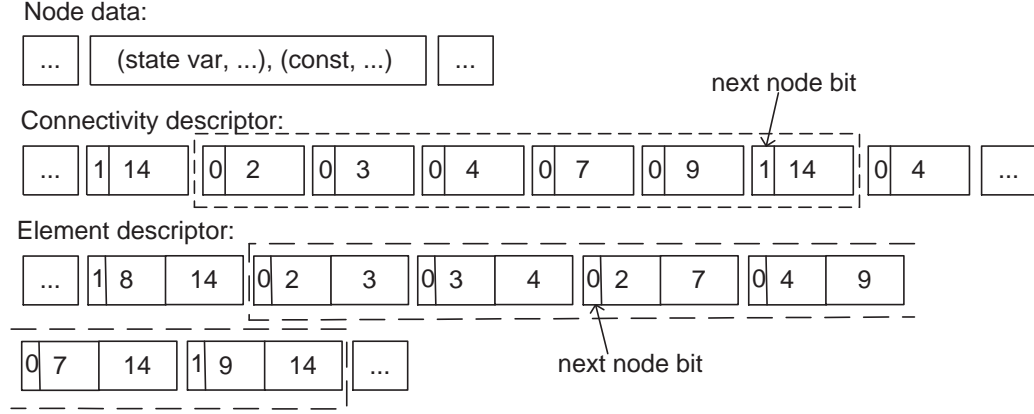| 0 | 7 | 14 | 1 | 9 | 14 | ... | next node bit

Figure 2: Data structures

Main parts of the proposed architecture are the Memory unit, the Neighborhood memory unit and the Arithmetic unit as shown in Figure 3. The Memory unit is build from dual ported on-chip BRAM memories and store a small part of grid data. Size of the on-chip memory is determined by the bandwidth of the adjacency matrix of the mesh. Bandwidth of a matrix is defined as the maximum distance of nonzero element from the main diagonal. (See Section 4.1 for a more formal definition.) Neighbors of the currently processed nodes are stored in the Neighborhood memory unit. Its structure is depending on the particular discretization stencil. In the simplest case when the stencil is defined on a line it can be a simple register. When the stencil is defined on a triangle or tetrahedron a one write two or three read multi ported memory is required respectively. Size of this memory is depending on the largest node rank in the mesh, usually a 64 element memory is sufficient, which can be efficiently implemented using the Distributed RAMs on FPGAs. Computation of the updated node value is carried out in an element by element order by the arithmetic unit.

Computation is started by loading node values into the Memory unit until it is half filled. In this phase value of the first node is loaded to the Current node register and the Neighborhood memory is filled by its neighbors using the incoming connectivity descriptors. Global indices of the neighboring nodes are translated into addresses in the Memory unit by the Local address generator unit. When all neighbors are loaded valid stencil data can be send to the arithmetic unit in each clock cycle. Neighborhood of the second node can be loaded into the free space of the Neighborhood memory while processing the first node. When all parts of the first stencil are sent to the arithmetic unit Node addressA register is incremented and the second node is loaded into the Current node register. During the next clock cycle new node data can be written into the Memory unit and computation of the
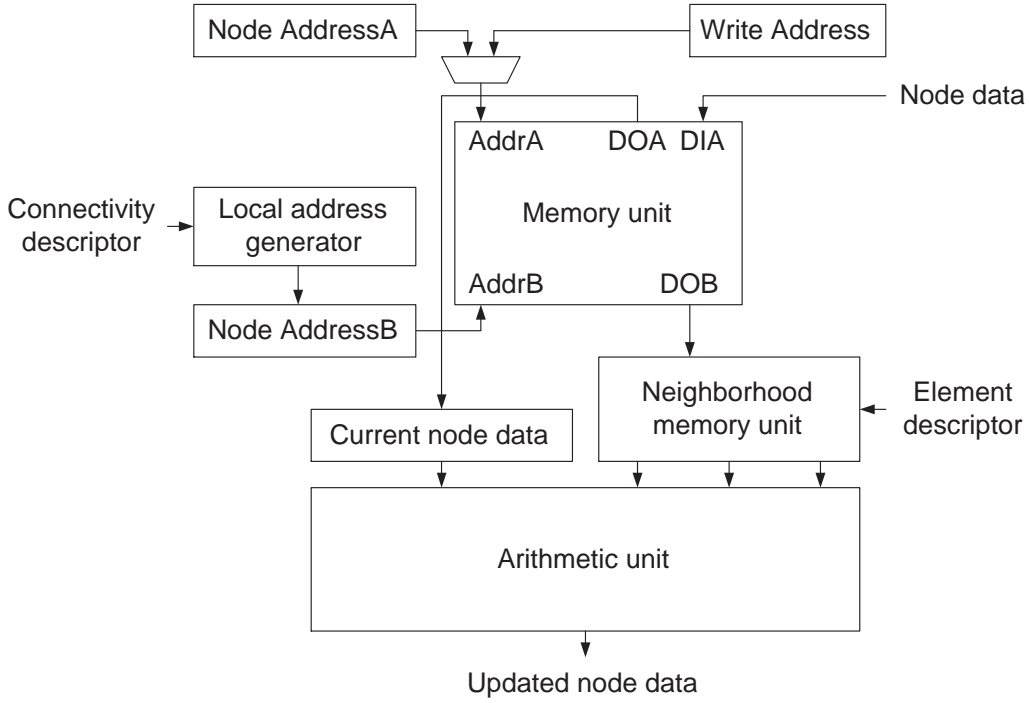
Figure 3: Block diagram of the proposed architecture

second node is started along with the loading of the neighborhood of the third node. The Memory unit is operating as a circular buffer when it is filled the oldest node data is overwritten which can be done safely because the size of the memory is set to be at least twice the bandwidth of the adjacency matrix and one can be sure that these old values are never required during the update of the remaining nodes.

Connectivity and element descriptors add an overhead for the off-chip memory requirements and increase memory bandwidth requirement of the processor. The width of the index fields can be optimized according to the depth of the Memory and Neighborhood memory units because all memory accesses are hidden by these two units. Therefore the global index of the nodes are never required only the translated memory addresses for the on-chip memories which can be much shorter. The order of node updates are statically scheduled hence memory address translation can be done offline by the host CPU.

The main advantage of the proposed architecture is the serial off-chip memory access pattern on the node data and descriptor arrays. Each array is read into a FIFO buffer by using an optimal burst length and penalties of random memory access patterns can be eliminated. Maximum size of the mesh

is limited by the bandwidth of the adjacency matrix. Using today large, high performance FPGAs even 10,000-40,000 nodes can be stored on-chip and depending on the structure of the mesh its size can be in the 100,000-400,000 node range.

4 Memory Bandwidth Optimization

Efficient use of on-chip memory resources of the architecture described in the previous section is depending on the bandwidth of the adjacency matrix of the mesh. By reordering the nodes of the mesh the bandwidth of the adjacency matrix can be significantly reduced. In this section we show a fast constructive method for reordering, which is comparable to the classical algorithms, and we present a solution for generating memory access patterns which have lower bandwidth than a given bound.

4.1 Description of the problems and related works

Matrix Bandwidth Minimization: Let G(V,E) a graph with vertex set V, $|V| = n$ and edge set E. Labeling is a function f(v) which assigns integers[1..n] to vertices. f(v)=f(u) if and only if u=v $u, v \in V$. N(v) is the set of vertices which adjacent to v. The bandwidth of a vertex v is $B_f(v) = Max\{|f(v) - f(u)| : u \in N(v)\}$, and the bandwidth corresponds to G with labeling f() is $B_f(G) = Max\{B_f(v) : v \in G\}$.

The problem of bandwidth reduction of a sparse matrix was shown to be NP-complete by Papadimitriou[14], so exact methods can not be applied for large problems. Many heuristic algorithm was developed from the well-known Cuthill-McKee (CM) algorithm[15] to recent metaheuristic approaches. One of the most promising metaheuristic results in solution quality is the GRASP (Greedy Randomized Adaptive Search Procedure) with Path Relinking, which was presented in[16]. The most practical solution is the GPS(Gibbs, Poole and Stockmeyer) method[17], which is one of the fastest heuristics, and also provides good solution quality. Metaheuristic methods gives better solutions, but their time consumption many times higher than GPS. GPS algorithm was born in 1976, afterwards many attempts shown to improve the original method, the most interesting from them is the method created by Luo[18]. All of the improved GPS heuristics have higher time complexity, which is an important parameter in case of large meshes.

Serial-Bandwidth: Using the classical definition of bandwidth, the size of the on-chip memory can be given as $(B_f(G) * 2 + 1) * sizeof(node)$, where $(B_f(G) * 2 + 1)$ called C_BW. If we assume the

8

architecture described in previous section, a more proper definition than C_BW can be given. S_BW means the number of nodes which have to be stored in on-chip memory, if we want 0% cache-miss.

$$s(i) = MIN\{f(v) : v \in N(u), f(u) = i\}$$

$$e(i) = MAX\{f(v) : v \in N(u), f(u) = i\}$$

$$S(i) = MIN\{s(i), s(i+1), ..., s(n)\}$$

$$E(i) = MAX\{e(1), e(2), ..., e(i)\}$$

$$S\_BW = MAX_i\{E(i) - S(i)\}$$

4.2 Algorithm for bandwidth reduction

Several methods have been shown in the literature for minimizing the classical bandwidth of a system. In this section we define Amoeba1(AM1) algorithm for serial-bandwidth minimization. Our goal is to create a fast, effective constructive method which has proper, easy to calculate S_BW bounds in each construction step(details in next subsection).

4.2.1 Notations and definitions

Amoeba1 is a constructive method, in which a solution element is chosen and labeled in each step. Solution elements are the vertices of the input mesh, and the method grows a part till all of the vertices are covered.
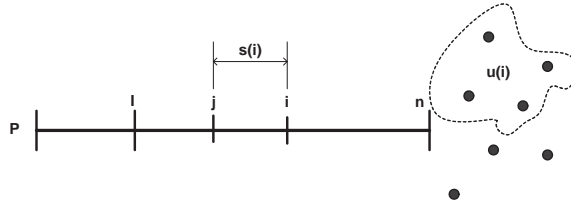

Figure 4: Structure of solution part P.

Figure 4. shows the structure of a solution part P with n elements. Every node(i) have three base parameters: local_index=i, s(i), u(i).

s(i): is the distance between node(i) and its lowest indexed neighbor in the part: $s(i) = MAX\{i - j : j \in N(i)\}$.

9

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

$u(i) = \{v : v \in N(i) \; AND \; v \notin P\}$.

I: is the index of the first elemet which has not empty u() set, so for every node(i) where $i < I$ all neighbors covered by P.

With these parameters we can give bounds on the serial bandwidth. In AM1 method we use a simple lower bound for describing the importance of node(i):

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

This is obviously a lower bound, because if we add node $v \notin u(i)$ to part P we still have to add all elements of u(i) to the part. For every node(i) $i < I$ imp(i)=0, because these nodes have all of their neighbors involved, so their effect on bandwidth do not depend on the later decisions.

4.2.2 Description of Amoeba1

Amoeba1 algorithm has two base steps: finding a starting vertex, and the labeling loop. The result is an ordering of the vertices.

Finding a starting vertex The quality of the result of constructive bandwidth-reduction heuristics depends on which is the starting vertex. In GPS method, the authors presented a simple and effective solution for this problem. They gave an algorithm which returns the two endpoints of a pseudo-diameter. AM1 algorithm uses this subroutine for finding the starting vertex.

Choosing a solution element AM1 selects a node from u(I), which has a neighbor in P with maximal importance. Because all nodes in u(I) has node(I) as its neighbor, only $l \neq node(I)$ neighbors take part in the search.

// selecting candidate from u(I)

1   $candidate = random\ element\ of\ u(I)$

2   $global\_max = 0$

3   **for** $\forall k \in u(I)$

4        $local\_max = 0$

5   **for** $\forall l \in N(k) :\ l \in P\ and\ l \neq node(I)$

6   **if** $l.imp() > local\_max$

7                  $local\_max = l.imp()$

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 Results and conclusions

AM1 is a simple constructive algorithm for large problems, we compare its results to the classical fast and effective GPS method. As mentioned earlier, better quality algorithms exist for bandwidth reduction, but these methods can not be applied to large meshes(¿100.000 vertex) because of their complexity. The cases showed on Table 1 comes from 2-dimmensional meshes, with assigning a vertex to each triangle and an edge between vertices which are represent adjacent triangles, so we get a mesh with maximal_degree = 3. This meshes appears when we use finite volume solver during the solution of a partial differential equation. In these low-degree cases AM1 provides similar solution quality to GPS, in 4% less time. The running time of both method depends on the number of vertices, and the structure of the mesh(finding a starting vertex).

The results for high-degree(20-30) cases can be found on Table 2. These cases generated from the same complex 3D geometry, by increasing the density of the mesh. We found that GPS is 29% superior on these general instances, but 13% slower than AM1. These results shows, that the difference is not increasing with the complexity of the problems. In our further work, we want to increase solution quality by guiding the labeling depending on the knowledge of the geometry.

Table 1: Results of Amoeba1 method compared to GPS.

Case	N	S_BW GPS	S_BW AM1	GPS time(s)	AM1 time(s)
step_2d_bc_cl30	7063	122	122	0,078	0,052
step_2d_bc_cl40	12297	176	175	0,154	0,109
step_2d_bc_cl50	20807	253	227	0,175	0,1
step_2d_bc_cl70	42449	359	341	0,633	0,49
step_2d_bc_cl90	68271	481	506	0,998	0,785
step_2d_bc_cl110	112093	569	591	2,144	1,955
step_2d_bc_cl130	157099	740	738	1,59	1,316
step_2d_bc_cl150	201069	794	805	3,239	3,094
step_2d_bc_cl170	252869	972	923	4,316	3,92
step_2d_bc_cl190	316715	1030	1082	5,913	5,707
step_2d_bc_cl200	394277	1093	1155	5,855	5,532
step_2d_bc_cl320	930071	1923	1809	17,035	18,687

S_BW: serial-bandwidth of solutions, N: number of vertices.

Algorithms tested on one core of an Intel P8400 processor

Table 2: Results for 3D high-degree cases

Case	N	S_BW GPS	S_BW AM1	GPS time(s)	AM1 time(s)
3d_075	3652	381	391	0,279	0,27
3d_065	5185	500	763	0,144	0,107
3d_055	8668	712	764	0,655	0,587
3d_045	15861	1066	1478	0,468	0,42
3d_035	33730	1880	1863	2,209	2,22
3d_025	88307	3384	3443	7,569	6,42
3d_018	244756	6582	10110	39,509	27,598
3d_015	417573	9066	14930	85,797	59,958
3d_012	519983	20561	23554	413,72	383,075

4.3   Memory Access Optimization for Bounded Bandwidth

In case of large problems it is possible, that the renumbered mesh has larger bandwidth than the available on-chip memory, these cases should be handled too. In this section we show an AM1 based method, which generate an input order which has at most a pre-specified serial-bandwidth. In the input order every vertex executed once, but can be loaded many times, so we need a flag **Ex** to store it is only a ghost node(false) or have to be calculated(true). Serial bandwidth in this job means the following: for every Ex=true vertex all of their neighbors surely be in the on-chip memory when the execution reaches them. If the defined bound is less than the serial-bandwidth for whole mesh provided by the AM1 method, the input will be k-times longer, where $1 \leq k$. The bound on bandwidth obviously have to be more than the maximal_degree of the graph.

4.3.1 AM1 based bounded S_BW method

The main concept of handling the bounded bandwidth is the usage of a proper serial bandwidth estimation, which is available in AM1 method. When a Part reaches the S_BW bound, the process calculates which vertices can be executed, and call the AM1 method for the rest of vertices where Ex is not true. The main process starts new AM1 parts until all vertices are Ex=true. The output of the method is an access pattern(list of {index,Ex} pairs), where all vertex has true execute flag once, and can be appear many times as a ghost node.

Estimation of serial-bandwith: Given an AM1 Part, the task is to estimate its serial bandwidth. AM1 estimates the part's bandwidth in each construction step. If $i < I$ for node(i) in part P, it has all of its neighbors inside the part, so if the bandwidth is less than the bound when I become larger than k, node(k) can not increases S_BW anymore. As shown earlier imp(i) is a lower bound on serial-bandwidth, but in the estimation a proper upper bound is required for the stopping condition. Because in all step AM1 adds a node from u(I) to the part, we can calculate more than a proper upper bound for node(i), we can give the exact value.

$$S\_BW(i) = (n - i) + |\bigcup_{I \leq k \leq i} u(k)| + s(i) \quad I \leq i \tag{1}$$

Eq. 1 is equivalent to the definition of serial bandwidth[1] Equation 1 could be a stopping condition, but the proposed method has a less complex and useful upper bound for stopping decision defined in Eq. 2.

$$S\_BW\,Bound \geq \underbrace{MAX}_{I \leq k \leq n} imp(k) \tag{2}$$

If Eq. 2 holds, AM1 continue to add nodes to part P, stop otherwise. This condition is not an upper bound for the whole part, but it provides in every step that node(I) has lower serial-bandwidth than the given bound. If I jumps to I' when AM1 adds node(n) to P, we can be sure that for all node(i) $I \leq i \leq I'$ serial bandwidth is under the bound, because $\bigcup_{I \leq k \leq I'} u(k) = \{node(n)\}$, so $imp(i) = S\_BW(i)$ inside the range [I ,I'].

Finalizing a Part: When Eq. 2 not holds, the proposed algorithm finalize the part, and starts a new instance of AM1 on the rest of the not executed nodes. Finalization have two tasks: it has to label vertices which executed in part P, and have to label vertices which all neighbors executed too, because these nodes can be cut out of the mesh(we call them perfect nodes Pr=true). Ex=true and

---

[1]the authors have a proof

Pr=false vertices have to be loaded again, because they have at least one Ex=false neighbor. In AM1 imp(i)=0 and u(i)={} for all node(i) for which Ex=true.

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

// setting Ex and Pr flags

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

2   **if** $k.local\_index < s^*$ and $k.Ex \, ! = \, true$

3               $k.Pr = true$

4   **if** $k.local\_index < I$

5               $k.Ex = true$

4.3.2   Results and conclusions

It is obvious that the proposed algorithm generates access patterns which has lower S_BW than a given bound. The input length multiplier **k** is a good parameter for measuring the solution quality. (k-1)*100% of the vertices have to be reloaded from the main memory, but the processing still has 0% cache-miss[2].

Measurements on three meshes with different S_BW bounds can be found on Table 3. The results shows that the solution quality mainly depends on the distance of the S_BW bound and the maximal_degree of the mesh. This is a really good news, because maximal_degree is around 20-30 for a typical 3D mesh, while the S_BW bound is around 10-40k[3] nowadays and increasing with each new generation of FPGA-s. The number of generated parts determine the time consumption of the proposed method, because in each restart of AM1 the algorithm calculates the pseudo diameter for the rest nodes. The results on 3d_015 shows that we can go below 25% of the original S_BW with 15-30% reload. This method gives the opportunity of deciding the size of the on-chip memory synthesized to the FPGA, so the designers can have more free area with sacrificing computational time.

---

[2]assuming the handling of multiplicity problems

[3]the bound is depending on sizeof(node)

Table 3: Results of AM1 bounded bandwidth optimization

Case	AM1_BW	S_BW Bound	num. of parts	N	overall length	k	time(s)
3d_075	391	392	1	3562	3562	1	0,255
3d_075	391	380	4	3562	4288	1,203	0,392
3d_075	391	300	9	3562	4945	1,388	0,7
3d_075	391	200	20	3562	5929	1,664	1,148
3d_035	1863	1864	1	33730	33730	1	2,317
3d_035	1863	1800	6	33730	38053	1,128	7,78
3d_035	1863	1500	5	33730	38702	1,147	4,439
3d_035	1863	500	88	33730	58171	1,724	36,095
3d_015	14930	14931	1	417573	417573	1	70,108
3d_015	14930	14000	2	417573	431081	1,032	77,004
3d_015	14930	10000	2	417573	427211	1,023	71,278
3d_015	14930	7500	8	417573	449441	1,076	91,058
3d_015	14930	5000	34	417573	476170	1,140	53,247
3d_015	14930	2500	130	417573	557190	1,334	687,385

AM1_BW: the bandwidth provided by AM1 for the whole mesh

overall length: length of the generated access pattern

N: number of vertices, Algorithm tested on one core of an Intel P8400 processor

5 Arithmetic unit generation

The VHDL representation of the arithmetic unit optimized for speed and area was automatically generated via the framework presented at [7]. To reach high operating frequency the floating-point units shall be partitioned and a local control unit shall be assigned to every cluster. The objective is to minimize the number of extra FIFOs required for data synchronization between the clusters while guaranteeing that the signals of the control unit have tolerable fanout and do not decrease the operating frequency of the rest of the circuit. In [7] it was demonstrated that the placement of a partition generated by traditional partitioniers which minimize the previous objective is very challenging and the maximum operating frequency of the slowest floating-point unit cannot be reached with the standard Xilinx place and route tools. A new partitioning strategy was demonstrated which produces more cut nets than common strategies however the resulting partition can be easily mapped to the FPGA and the resulting circuit can operate near to the maximum operating frequency of the slowest floating-point units.

The main idea of the algorithm is to draw the graph into the plane before the partitioning starts. If a representation of the graph which minimizes the distance between the connected edges is given a simple greedy clustering algorithm can provide a partitioning without long interconnections between the clusters.

The input of the procedure is a text file containing the discretized state equations of the state variables. In the first step the text file is parsed and a data-flow graph is created where every global variable and mathematical operator is represented with a vertex. As there are enough resources on the FPGA and our goal is to reach the maximum operating frequency the data-flow graph is not optimized to reduce the number of vertices however the same effect can be reached manually if local variables are defined in the input file.

The next step of the initialization of the algorithm is to create a bipartite graph from the original data-flow graph. As every vertex representing an operator is associated with a delay time a bipartite graph can be easily created via a breadth-first search which visits every vertex of the graph and computes the level of the vertex and the time required for the input to reach the given vertex. If the levels of its ancestors are different the algorithm inserts the proper number of extra vertices (delays) after the problematic ancestor. In physical implementation these delays will be shift registers which hold the data for the proper number of clock cycles and the computed levels determines the vertical coordinates of the vertices.

The proposed algorithm consists of two greedy phases. The first phase collects global information about the graph by positioning the vertices horizontally, while clusters are created in the second phase using the information encoded in the spatial position of the vertices.

The first greedy phase tries to minimize the distance between the connected vertices described by the following equation:

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

where x_A and x_B are the horizontal coordinates of vertex A and B respectively. A simple swap based algorithm like Kernighan-Lin [19] have been designed which minimize this objective for all vertices together. This algorithm can be easily trapped in the local minimum therefore the initial placement of the vertices is critical. In our experiments the fast Barycenter heuristic [20] was used to create an initial solution however more complex techniques can be used as well. The Barycenter heuristic was initialized with several random placements and the best solution was seeded to the swap based iterative algorithm.

The second phase is another simple greedy algorithm which search the result of the first phase for rectangular clusters. Height of the rectangular domains can be chosen arbitrary however in our examples it is set to two. The algorithm is started from the top left corner and the largest possible rectangular cluster is searched which still meets the I/O constraint. Next the algorithm moves right

and search for the largest possible rectangular cluster from the rest of the unclustered vertices. If there are no more unclustered vertices in the selected layers the algorithm moves down and continues with the lower layers.

Position of specific parts of the circuit can be constrained by the Xilinx PlanAhead software. It enables the user to create rectangular placement constraints also called pblocks. The Xilinx P&R tools are likely to disperse the registers of the FIFOs which can limit the operating frequency of the circuit therefore separate pblocks were created for each FIFO buffer. As the Xilinx P&R tools were able to place and route the floating-point units inside the clusters no pblock were created for the floating-point units just for the clusters. The pblocks was placed manually using the positions of the placed vertices however in a future work this can be fully automatized if the area requirements of the clusters are also calculated.

6 Case study: Finite volume solver for the Euler equations

In this section we describe the computational modelling of compressible fluid and gas flows by some of the basic tools available in the field of Computational Fluid Dynamics (CFD). The art of CFD is heavily exploiting the enormous processing power available by recent computer technology. Indeed, its central concept is the approximation of the continuous model problem by a discrete one, requiring the processing and the manipulation of a huge amount of data.

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 modelled 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 [21, 22], describing the dynamics of dissipation-free, inviscid, compressible fluids. They are a coupled set of nonlinear hyperbolic partial differential equations, in conservative form expressed as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{3a}$$

$$\frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot \left(\rho \mathbf{v} \mathbf{v} + \hat{I} p \right) = 0 \tag{3b}$$

17

$$\frac{\partial E}{\partial t} + \nabla \cdot ((E + p)\mathbf{v}) = 0 \tag{3c}$$

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

$$E = \frac{p}{\gamma - 1} + \frac{1}{2}\rho\mathbf{v} \cdot \mathbf{v}. \tag{3d}$$

In equation (3d) 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 (3a), (3b) and (3c) into hyperbolic conservation law form in terms of U and the flux tensor

$$\mathbf{F} = \begin{pmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\mathbf{v} + Ip \\ (E + p)\mathbf{v} \end{pmatrix} \tag{4}$$

as:

$$\frac{\partial U}{\partial t} + \nabla \cdot \mathbf{F} = 0. \tag{5}$$

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.
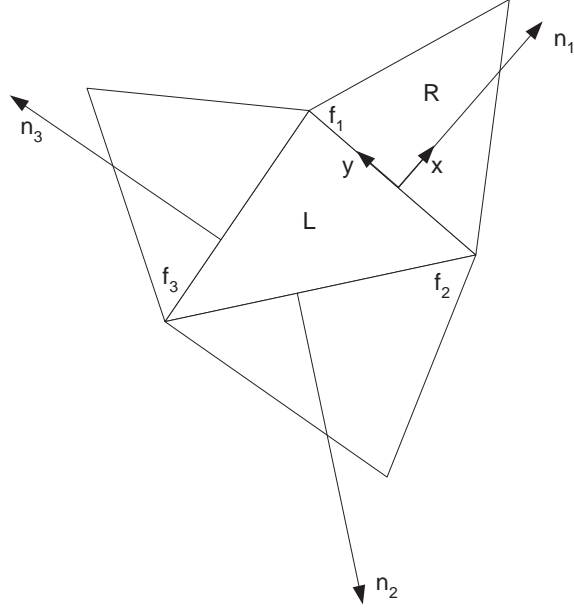
Figure 5: Interface with the normal vector and the cells required in the computation

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 (5)

$$\frac{dU_\mathcal{T}}{dt} = -\frac{1}{V_\mathcal{T}} \sum_f \mathbf{F}_f \cdot \mathbf{n}_f, \tag{6}$$

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, such, that its x-axes is normal to f (see Fig. 5). 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 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

$$F^N = \frac{F_L + F_R}{2} - (|\bar{u}| + \bar{c}) \frac{U_R - U_L}{2}. \tag{7}$$

In the last equation $\mathrm{F}_L = \mathrm{F}_x(\mathrm{U}_L)$ and $\mathrm{F}_R = \mathrm{F}_x(\mathrm{U}_R)$ 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

19

derivative is dicretized by the first-order forward Euler method:

$$\frac{dU_{\mathcal{T}}}{dt} = \frac{U_{\mathcal{T}}^{n+1} - U_{\mathcal{T}}^{n}}{\Delta t}, \tag{8}$$

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 and the dissipation term D.

$$U_{\mathcal{T}}^{n+1} = U_{\mathcal{T}}^{n} - \frac{\Delta t}{V_{\mathcal{T}}} \sum_{f} \hat{\mathcal{R}}_{\mathbf{n_f}} F_f |\mathbf{n_f}|, \tag{9}$$

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

$$F_f^{\rho} = \frac{\rho_L u_L + \rho_R u_R}{2} + (|\bar{u}| + \bar{c}) \frac{\rho_R - \rho_L}{2} \tag{10a}$$

$$F_f^{\rho u} = \frac{\left(\rho_L u_L^2 + p_L\right) + \left(\rho_R u_R^2 + p_R\right)}{2} + (|\bar{u}| + \bar{c}) \frac{\rho_R u_R - \rho_L u_L}{2} \tag{10b}$$

$$F_f^{\rho v} = \frac{\rho_L u_L v_L + \rho_R u_R v_R}{2} + (|\bar{u}| + \bar{c}) \frac{\rho_R v_R - \rho_L v_L}{2} \tag{10c}$$

$$F_f^{E} = \frac{(E_L + p_L) u_L + (E_R + p_R) u_R}{2} + (|\bar{u}| + \bar{c}) \frac{E_R - E_L}{2} \tag{10d}$$

7   Results, performance

The generated arithmetic unit will be implemented on our AlphaData ADM-XRC-6T1 reconfigurable development system [23] equipped with a Xilinx Virtex-6 XC6VSX475T FPGA [24] and 2Gbyte on-board DRAM in four 32bit wide banks running on 8000MHz.

In our test case the cell centered approach is used therefore each triangle is represented by a node and each node has 3 neighbors except at the boundaries where ghost nodes are used to implement inflow, outflow or wall boundary conditions. For simplicity size and normal vector of all faces are precomputed which results in higher memory bandwidth requirement, but the arithmetic unit will be simpler. The architecture is implemented using both single and double precision floating point numbers. The node data structure contains four time dependent variables $[\rho, \rho u, \rho v, E]$ and the area of
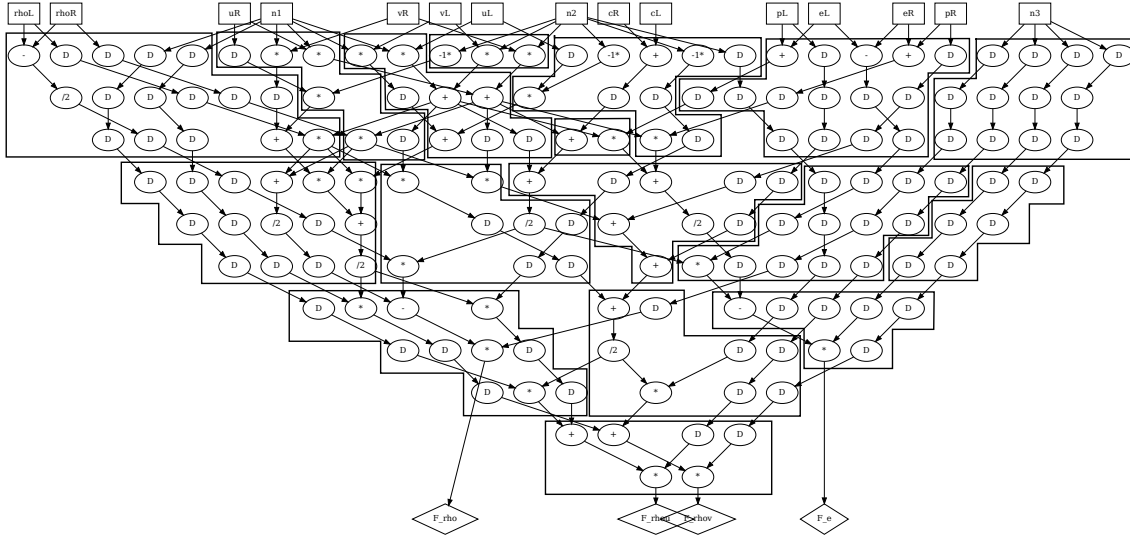
Figure 6: The partitioned arithmetic unit

the triangle as a constant additionally pressure p and local speed of sound c which is computed on the FPGA for each node also should be stored. Therefore width of data bus of the Memory unit is 28 and 56 byte in the single and double precision cases. To provide this wide memory bus 14 BRAMs should be used in a 32×512 configuration. When all BRAMs are allocated for the Memory unit 38,912 nodes can be stored on the FPGA which is usually more than enough for practical 2D meshes.

Depending on the width of the values the input data bus is 20 or 80 byte wide while 16 or 32 byte wide results should be loaded and saved in every 3 clock cycles. Additionally 3 face descriptors should be loaded which contains 9 floating point values and 3×2 byte wide indices altogether. Therefore 10.3Gbyte/s or 19.7Gbyte/s memory bandwidth is required to feed the arithmetic unit with valid data in each clock cycle. In the double precision case this bandwidth cannot be provided on our prototyping board and the system will be memory bandwidth limited. It should be noted that computing face normals on the FPGA using the coordinates of the vertices results in about 30% smaller memory bandwidth requirement at a price of more complicated on-chip memory structure and arithmetic unit. Results from the output of one processor can be feed directly to another processor linearly increasing the performance of the architecture without additional bandwidth requirement.

7.1 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 has
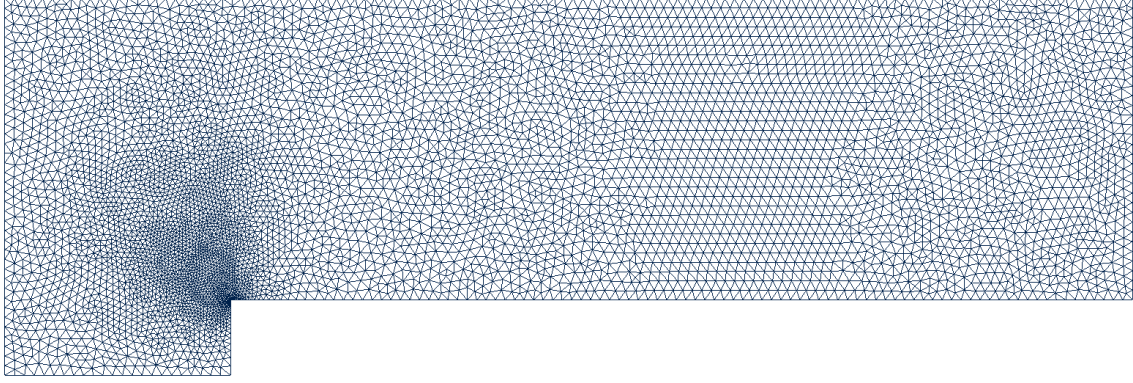
Figure 7: Coarse resolution mesh for the forward facing step test case

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-time the speed of sound constantly. The solution contains shock waves reflecting from the closed boundaries.

Unstructured mesh for the domain are generated by the freely available Gmsh mesh generator [25]. Several meshes are generated using different characteristic length between $1/30.0 - 1/200.0$ while 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 $1/40.0$ characteristic length and containing 12,297 triangles is shown in Figure 7.

7.2   Performance comparison

During a comparison various mesh sizes are used with 7,063 to 394,277 triangles. Performance of our architecture is estimated using the result of static timing analysis which is indicated 390MHz operating frequency in the double precision case. Three clock cycles are required to update the state of one triangle therefore performance of one processor is 130million triangle update/s. Computation of one triangle requires 213 floating point operations therefore performance of our architecture is 27.69GFLOPs. On the Virtex-6 XC6VSX475T FPGA three arithmetic units can be implemented resulting in 83.07GFLOPs cumulative computing performance.

Performance of our architecture is compared to a high performance Intel Xeon E5620 microprocessor running on 2.4GHz clock frequency. On the microprocessor a single core is used and the simulation is carried out with and without renumbering the nodes. Performance of the simulations is shown in Figure 8. As we expected without renumbering the performance of the simulation is steadily decreasing as the mesh size is increased while renumbering the nodes improved the performance of the
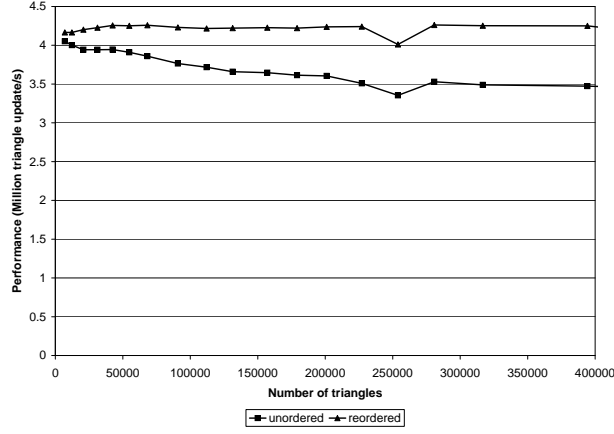
22

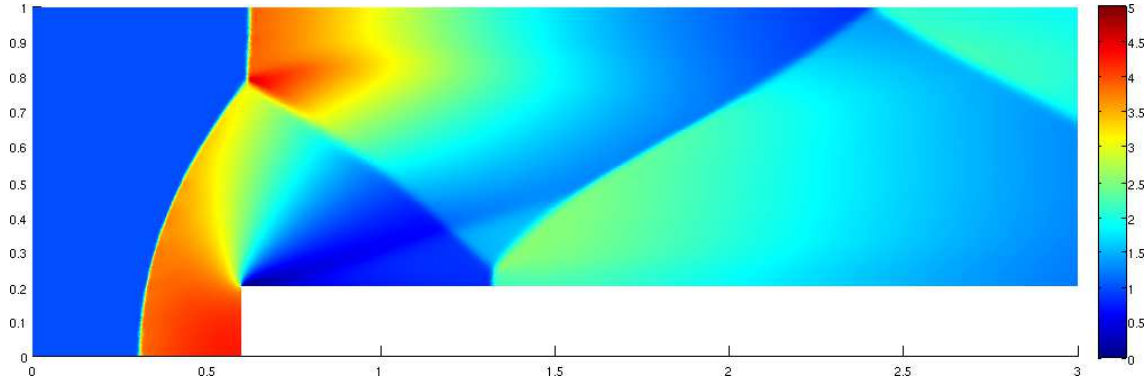Figure 8: Measured performance of Intel Xeon E5620 microprocessor

Figure 9: First-order solution of the Mach 3 flow after 4s simulation time on a 394,277 triangle mesh

microprocessor by 15% to 4.22million triangle update/s or equivalently 898.86MFLOPs.

Comparison of the performance of the two architectures show that a single processor implemented on the FPGA can outperform the Intel Xeon processor computing 30 times faster. By connecting three processors operating in parallel on a single FPGA achieve linear speedup and provide 90 times more computing power than a microprocessor.

Result of the computation on the largest simulated grid after 4s of simulation time is shown in Figure 9. Reference solution for the previous problem computed by the more accurate residual distribution upwind scheme can be found in [26].

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 the 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. Nodes of the arithmetic unit is rearranged to minimize edge crossing which helped placement of the partitions and improved clock frequency of the design.

Performance comparison of the architecture using a single processor running on 390MHz showed that 30 times speedup can be achieved compared to a high performance microprocessor core. Computing performance can be further improved by implementing three processors on one FPGA reaching 90 times speedup.

Currently size of the mesh is limited by the bandwidth of its adjacency matrix which must be smaller than 40,000. The architecture should be improved to efficiently handle multiple partitions and extended to use multiple FPGAs during computation.

Acknowledgments

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)*, page 99–102, 2010.

[7] C. 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[20] 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.

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

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

[23] Alpha Data Inc. http://www.alpha-data.com/, 2011.

[24] Xilinx Inc. http://www.xilinx.com/, 2011.

[25] 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.

[26] 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.