Efficient Assembly for High-Order Unstructured FEM Meshes (FPL 2015)

PAVEL BUROVSKIY, Maxeler Technologies
PAUL GRIGORAS, SPENCER SHERWIN, and WAYNE LUK, Imperial College London

The Finite Element Method (FEM) is a common numerical technique used for solving Partial Differential Equations on large and unstructured domain geometries. Numerical methods for FEM typically use algorithms and data structures which exhibit an unstructured memory access pattern. This makes acceleration of FEM on Field-Programmable Gate Arrays using an efficient, deeply pipelined architecture particularly challenging. In this work, we focus on implementing and optimising a vector assembly operation which, in the context of FEM, induces the unstructured memory access. We propose a dataflow architecture, graph-based theoretical model, and design flow for optimising the assembly operation for spectral/hp finite element method on reconfigurable accelerators. We evaluate the proposed approach on two benchmark meshes and show that the graph-theoretic method of generating a static data access schedule results in a significant improvement in resource utilisation compared to prior work. This enables supporting larger FEM meshes on FPGA than previously possible.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; **Data flow architectures**; *Heterogeneous (hybrid) systems*; • **Applied computing** → Physics; • **Hardware** → *Hardware accelerators;*

Additional Key Words and Phrases: Application mapping, FPGA architecture, graph algorithms, high performance computing, reconfigurable computing

ACM Reference Format:
Pavel Burovskiy, Paul Grigoras, Spencer Sherwin, and Wayne Luk. 2017. Efficient assembly for high-order unstructured FEM meshes (FPL 2015). ACM Trans. Reconfigurable Technol. Syst. 10, 2, Article 12 (April 2017), 22 pages.
DOI: http://dx.doi.org/10.1145/3024064

1. INTRODUCTION

Finite Elements Method (FEM) is an ubiquitous tool in many areas of science and engineering, such as geophysics, fluid and structure mechanics, electromagnetics, and biomedicine [Cantwell et al. 2014]. The method solves Partial Differential Equations (PDEs) by discretising space and time and thus reducing the PDE problem to a finite dimensional, linear algebra problem to construct a numerical approximation of the solution.

This work was supported in part by the Maxeler University Programme, Altera, UK EPSRC Projects EP/I012036/1, EP/L00058X/1, and EP/N031768/1, the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, and the HiPEAC NoE.
Authors' addresses: P. Burovskiy, Maxeler Technologies, 3-4 Albion Pl, London, W6 0QT; email: pburovskiy@maxeler.com; P. Grigoras and W. Luk, Department of Computing, Imperial College London, Huxley Building, 180 Queen's Gate, London SW7 2AZ; emails: {paul.grigoras09, w.luk}@imperial.ac.uk; S. Sherwin, Department of Aeronautics, Imperial College London, South Kensington Campus, London SW7 2AZ; email: s.sherwin@imperial.ac.uk.

DOI: http://dx.doi.org/10.1145/3024064

The FEM supports *unstructured* spacial discretisations, which are essential for accurately capturing complex geometrical shapes found in industrial and scientific applications. But numerical methods solving PDEs over these unstructured domains, including the method studied in this work, require the solution of large scale sparse linear systems of equations. These sparse data structures induce an *unstructured memory access pattern*, which cannot be determined and optimised statically, at compile time.

Previous work shows that, in the context of the FEM, the fine-grained control over the massively parallel on-chip memory resources of the Field-Programmable Gate Array (FPGA) may be used to efficiently implement no-stall unstructured memory access [Burovskiy et al. 2015]. The key is to take advantage of the time-invariant nature of the sparse data structures used by the iterative linear solvers, preserved over many timesteps of a PDE solver. Since data do not change during the FEM simulation, the unstructured memory access can be resolved efficiently by employing (1) a deeply pipelined architecture with an efficient custom cache, which maps well to the FPGA fabric, and (2) a software generated execution schedule which maps various FEM simulation inputs to this architecture. Altogether, this enables the hardware acceleration of FEM problems of practical interest at the cost of runtime preprocessing of FEM data.

In this work, we improve the heuristic schedule generation presented in Burovskiy et al. [2015] by proposing a systematic graph-based method of identifying and processing the access constraints. This article has the following contributions:

(1) *A novel approach to FEM vector assembly designed for execution directly on FPGA*. We decompose the problem of supporting vector assembly on FPGA into (1) data storage on chip with scheduled access and (2) several graph-based sub-problems to be solved on CPU as data pre-processing; we propose a graph-theoretic formulation to solve these sub-problems and, therefore, the problem of on-chip vector assembly; by leveraging standard graph-theoretical approaches, we are able to both simplify the implementation and improve the resource utilisation of the design;
(2) *Design flow and prototype implementation of an accelerator for the single-level static condensation linear solver on a Maxeler Maia Dataflow Engine (DFE) acceleration board*. This linear solver is a part of the incompressible Navier-Stokes PDE solver within the Nektar++ spectral/hp FEM framework [Cantwell et al. 2015], which is used as a software reference in this work. The linear solver is based on the diagonally preconditioned Conjugate Gradient (CG) method with reduced communication reordering [Demmel et al. 1993].
(3) *Experimental evaluation of the proposed approach on two unstructured 3D benchmark meshes with tetrahedral and prismatic elements.* We provide the resource utilisation for both problems and compare performance for both accelerated and non-accelerated Nektar++ runs on the same Intel Xeon server. We demonstrate significant improvement in quality of results compared to our previous work [Burovskiy et al. 2015] due to improvement in schedule. This leads to increased supported problem size: both benchmark FEM problems can fit on a single FPGA device, leaving resources for further architectural improvements.

Some parts of this article have appeared in Burovskiy et al. [2015]. New material includes the graph-theoretic approach for translating assembly mapping into data access schedule (Section 5), evaluating its impact on resource utilisation (Section 6), and discussion of processing rate balance in the proposed architecture (Section 4.2).

As in previous work [Burovskiy et al. 2015], we focus on the higher-order spectral/hp FEM methods, which result in a more regular data access pattern outside of the assembly operation. This leads to realising sparse matrix-vector multiply in the CG as a collection of dense block-matrix-vector multiplies [Grigoras et al. 2016a, 2016b] and on-chip vector assembly. We also assume the FEM matrices remain static throughout the

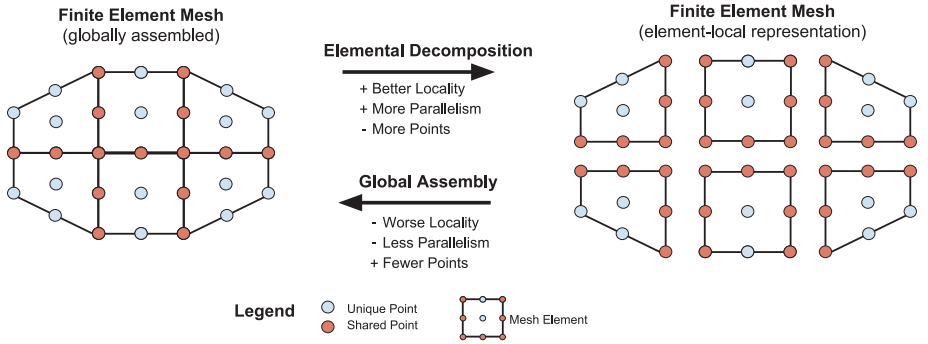

Fig. 1. Left: FEM mesh. Right: domain decomposed into separate elements with their local discretisations. Elemental decomposition and global assembly can be denoted using the incidence matrix A and its transpose A^T of a mapping from the set of element-local discretisation points onto the set of mesh-global points.

computation, which makes the static overhead of data pre-processing and restructuring to optimise on-chip execution of vector assembly tolerable: the overhead is amortised across the large total execution time of the entire FEM simulation.

The proposed implementation strategy of vector assembly might be applicable to other types of hardware accelerators (e.g., GPUs), although that discussion is beyond the scope of this article.

2. BACKGROUND AND RELATED WORK

The FEM [Galerkin 1915; Strang and Fix 1973; Karniadakis and Sherwin 2013] is a canonical high performance computing problem used to solve PDEs on large-scale geometric domains, which are used in many fields of engineering. Given the importance of the method, the use of custom accelerators such as GPUs [Ikushima et al. 2015; Markall et al. 2013] or FPGAs [Elkurdi et al. 2008; van der Veen 2007; Lienhart et al. 2005; Wu et al. 2013; Hu et al. 2008; Piechotka 2013] has long been anticipated to reduce the substantial execution times observed on traditional CPU clusters. So far, a substantial limitation of prior work, particularly on FPGAs, has been the small problem size supported. This makes previous work, although exciting and showing great potential for reconfigurable acceleration, limited to problems so small they are of no practical interest to FEM users in important domains such as computational fluid dynamics. This limitation is precisely what we propose to address in this work. A key factor in enabling support for larger problem sizes is the focus on *higher-order* FEM, as explained in the following section.

2.1. The FEM

The FEM operates on meshes which represent the geometrical domains by splitting them into *elements*: simple geometrical shapes such as prisms or tetrahedra. The adjacency graph on these elements is unstructured. A solution to the PDE is approximated with a piecewise polynomial, defined at every element separately, with an additional continuity constraint at element boundaries. Boundary points of adjacent elements overlap as shown in Figure 1. Hence, to enforce continuity, the element-local numerical approximations are accumulated at the points of overlap. This is the procedure of *global assembly*, the key operation of the FEM method [Markall et al. 2013; Karniadakis and Sherwin 2013].

For each element e of the domain, the FEM discretises a differential operator into an element-local $n \times n$ matrix M^e and n-component vector v^e, where n is the number of discretisation points for the element. This produces a block-diagonal *local matrix* M_l of dense local matrix blocks M^e and a *local vector* v_l. M_l represents the mesh-global

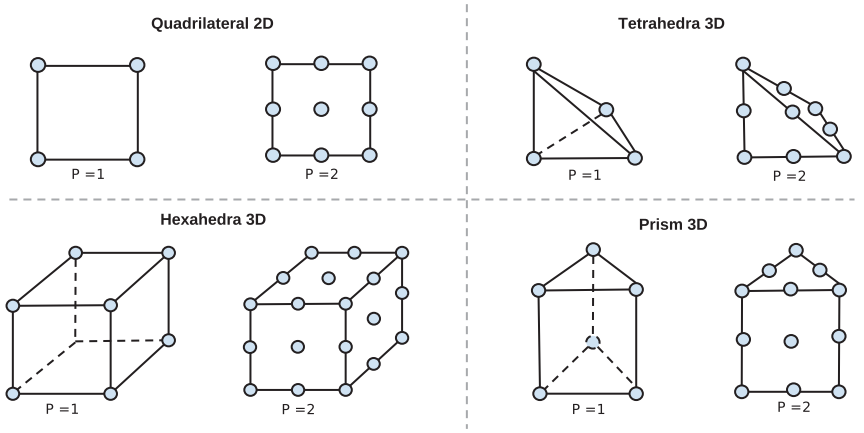

Fig. 2. Example of discretisation for various elemental shapes at $P = 1$ and $P = 2$.

discretisation with some redundancies, but with a locally regular structure of dense blocks. To eliminate redundant data, a global assembly operation can be performed, leading to a *global matrix* M_g. This is done by multiplying the block-diagonal matrix M_l from left and right with matrices A^T and A, accordingly: $M_g = (A^T M_l A)$. The resulting matrix M_g is a sparse matrix of a smaller rank; its rank, which normally equals the number of matrix rows, also equals the number of unique mesh-global discretisation points.

Similarly, the *global vector* v_g can be assembled from a local vector v_l by right-multiplying it by A^T matrix: $v_g = A^T v_l$. In this work, we refer to vectors v_l and v_g as the same vector represented in a *local coefficient space* and a *global coefficient space*, respectively. It is also possible to construct the vectors in a local coefficient space from their global coefficient space counterparts by changing matrix A^T with A: $\tilde{v_l} = A v_g$. However, v_l and $\tilde{v_l}$ are different vectors.

Every mesh element is discretised into a regular grid of points, which form the support of a polynomial approximation function of order P. Higher values of P lead to more internal element points, and, therefore, more regular local mesh structure. Figure 2 illustrates this in the case of 2D and 3D elements: in 2D, the number n of internal points is $O(P^2)$; in 3D, it is $O(P^3)$. Exact formulas depend on element shape and dimension; e.g., for a quadrilateral, the number n of internal points equals $(P+1)^2$.

The geometrical discretisation of each element can be automatically refined by increasing P, without the need to modify the geometrical description of the mesh. The special case of $P = 1$ corresponds to the classical FEM, where basis functions are piecewise linear. In this article, we assume every element has the same polynomial order, although it is possible to have meshes with spacially-local variations of P.

2.2. Evaluation Strategies for Matrix-Vector Multiplication in Spectral/hp FEM

The structure of M_g can be exploited to optimise the evaluation of matrix-vector products $s_g := M_g v_g$. The *global matrix approach* [Karniadakis and Sherwin 2013] explicitly forms M_g and stores it in a sparse storage format. This operation can be performed only once as a pre-processing step, as long as the mesh remains static throughout the computation. The alternative *local matrix approach* [Karniadakis and Sherwin 2013] uses the structure of M_g at every evaluation of the matrix-vector multiplication:

$$s_g = \underbrace{M_g v_g}_{\text{global matrix approach}} = (A^T M_l A) v_g = \underbrace{A^T M_l (A v_g) = A^T (M_l v_l) = A^T s_l}_{\text{local matrix approach}}. \quad (1)$$

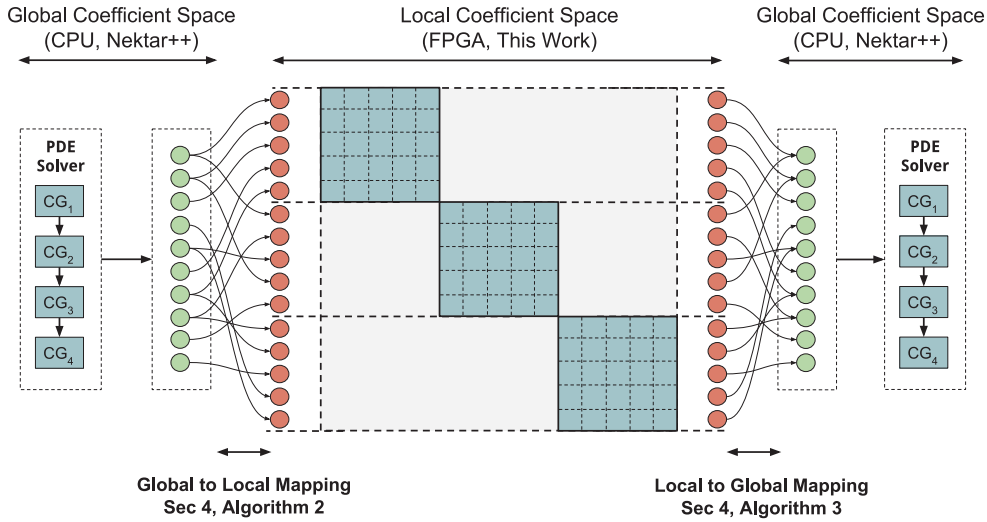

Fig. 3. Transformations between coefficient spaces within a compute pipeline.

Equation (1) shows that the two approaches are equivalent: the same result can be achieved by either multiplying a global vector v_g by a sparse matrix M_g (which implies using SpMV kernels) or by a three-stage procedure, as show in Figure 3:

(1) Vector v_g in a global coefficient space is first *scattered* into a local coefficient space by multiplying it by matrix A, to form vector v_l;
(2) Operation done in a local coefficient space: vector v_l is multiplied by a block-diagonal dense matrix M_l to produce vector s_l;
(3) Vector s_l is *gathered* from local into the global coefficient space by multiplying it with matrix A^T, to form vector s_g.

In the local matrix approach, the sparse matrix M_g is not explicitly formed. The only unstructured computations are the vector gather and scatter operations. Vos et al. [2010] presents the CPU performance of matrix evaluation strategies as a function of the polynomial order and mesh structure. For lower polynomial orders, most mesh points are shared across multiple elements, thus, the redundancy in local data representations is substantial. This makes a dramatic impact on performance: for lower polynomial orders the sparse matrix representation provides sufficient compression ratios to compensate for the performance loss stemming from the unstructured data access. However, for higher polynomial orders, the local matrix approach becomes more computationally efficient due to the more regular memory access pattern and a diminished effect of compression.

An additional factor of computational efficiency of the local matrix approach stems from higher computational complexity of block diagonal dense matrix multiplication compared to the computational complexity of gather and scatter operations: the workload for a local matrix multiply scales as $O(P^{2d})$ where d is space dimension, while the workloads of gather and scatter scale as $O(P^d)$. This makes the regular part of computational flow dominate for higher polynomial orders. In Vos et al. [2010], the optimal implementation strategy as a function of P is studied for various finite element operators with focus on CPUs.

It is these two effects that we propose to exploit in our work in order to improve the applicability of FPGA-based accelerators to larger FEM problems.

2.3. Acceleration of FEM

Due to its long running times, there has been a significant interest in accelerating the FEM using GPUs or FPGAs. Some previous work avoids implicit numerical schemes to accelerate large-scale FEM computations on GPUs [Ikushima et al. 2015]. A comprehensive review of available GPU papers on FEM matrix assembly in the context of implicit numerical schemes can be found in Markall et al. [2013]. This work also provides a good introduction to the operator implementation options for the spectral/hp method. Finally, it concludes that higher-order methods are more suitable for GPU implementation.

In the FPGA community, a number of past publications and student works (e.g., Elkurdi et al. [2008] and van der Veen [2007]) focused on the Sparse Matrix Vector multiply (SpMV) architectures, specialised for FEM matrices of particular connectivity. Lienhart et al. [2005] focuses on the LU decomposition of a global matrix for the low-order FEM method, where the matrix is generated by FEMLAB software for 1D or 2D meshes. Wu et al. [2013] presents a general purpose CG solver architecture with SpMV kernel, evaluated on sparse matrices with dimensions up to 66,127. Chow et al. [2014] is another general purpose SpMV-based CG solver on FPGA, evaluated on sparse matrices of rank up to 63,838.

Piechotka [2013] (master's thesis) presents a completely different approach to accelerating FEM on FPGAs. It prototypes a matrix-free Flux Reconstruction method [Vincent et al. 2011], which is still quite unconventional in the FEM community, evaluating a 2D linear advection on a 100×100 regular grid using single precision arithmetic a fourth-order polynomial basis.

Only Hu et al. [2008] can be directly compared to this work: they implement the local matrix approach in the context of higher-order FEM methods and evaluate it on regular geometric domains with up to 48,000 tetrahedra using single precision arithmetic. They mention the benefit of using a higher-order method for the increase in regularity of data access, but do not mention the approximation orders used in their study. Assuming first-order approximations, the number of local degrees of freedom (local coefficients) for this test mesh equals 192,000. Our benchmark meshes presented in the Table II are unstructured and, for the *third-order approximations* used in this work, have 3.6 and 14.3 times more local degrees of freedom, accordingly.

In summary, acceleration of FEM problems using custom accelerators poses significant challenges due to the unstructured nature of memory accesses. However, exploiting the locally regular data structure induced by higher-order FEM problems enables and simplifies custom accelerator implementations. As we show in the remainder of this work, this can lead to an efficient, scalable architecture which maps well to the FPGA fabric, enabling support for FEM problems of practical importance.

3. APPROACH OVERVIEW

We propose to take advantage of the runtime reconfiguration capabilities of the FPGA to generate an efficient, problem-specific custom accelerator for the high-order spectral/hp FEM problems. Our approach consists of three components:

(1) A parametric, reconfigurable streaming dataflow architecture, presented in Section 4; this enables a stall-free implementation of the assembly operation and makes use of an efficient design to perform the block matrix vector multiplication required in high-order spectral/hp FEM;
(2) A software component interfacing the spectral/hp FEM package (in our case, Nektar++) with an instance of our dataflow architecture on FPGA;
(3) A runtime pre-processing software component for translating vector assembly mapping data from the spectral/hp FEM package into the execution schedule

required for the dataflow architecture; this is presented in more detail in Section 5 and builds on our graph-based approach to generate the execution schedule for a particular mesh.

Together these components enable the generation of a problem-specific custom acceleration architecture. Taking advantage of runtime reconfiguration capabilities of the FPGA, an appropriate architecture can be loaded at runtime for a specific mesh instance. The operation of the proposed flow is summarised below:

(1) An *offline phase*, where a repository of accelerated architectures is created:
    —a set of target architecture parameters is identified from the problem domain;
    —all required FPGA bitstreams are synthesised;
(2) A *runtime phase*, where the accelerator executes for a particular input mesh:
    (a) A numerical simulation with the spectral/hp FEM package is started for a particular input mesh problem;
    (b) Based on the vector assembly mapping received from the spectral/hp FEM software package and a set of architecture parameters for available FPGA bitstreams, the on-chip data access schedule is generated as described in Section 5;
    (c) For a schedule generated, an architecture is selected from the repository and the accelerator is reconfigured;
    (d) The accelerator begins execution and all CG calls are redirected from the spectral/hp FEM software package to the reconfigurable accelerator.

Steps 2b and 2c are executed only once in the beginning of FEM simulation. This two-phase approach is required because some architecture parameters must be fixed at compile time: the number of on-chip buffers to use during assembly, the number of parallel pipelines, and the vector width of arithmetic processing units are static parameters. These are discussed in more detail in Section 4 and Section 5.

A given instance of a dataflow architecture may support many mesh problems, as long as on-chip buffer configuration and capacity match the schedule generated. The use of a systematic graph-based approach to capture the data access constraints enables generating less resource demanding schedules compared to prior work, which leads to substantial on-chip resource savings for a given mesh problem; this forms the main contribution of this work.

There is a possibility that no available bitstream has sufficient on-chip buffers to meet the scheduling constraints for a given mesh on a single DFE. In this case, the only viable option is to decompose the geometric domain and perform multi-FPGA/CPU computation, where every device executes the same computation on a subdomain it owns. In this work, we do not cover this case.

4. ARCHITECTURE

To demonstrate the applicability of FPGA-based acceleration to higher order FEMs we implement the most time-consuming stage of the implicit numerical scheme for the incompressible Navier-Stokes equation: solving a set of linear equations using the CG method. We interface with the spectral/hp FEM framework Nektar++ [Cantwell et al. 2015], which provides all input data to our prototype dataflow accelerator of a preconditioned CG solver (presented in Section 4.2). In our dataflow architecture, we focus on a local matrix approach and higher polynomial orders, mapping the three-stage local matrix evaluation procedure described in Section 2.2 to a hardware pipeline.

4.1. System Overview

We use the Velocity Correction Scheme for the incompressible Navier-Stokes problem, available in Nektar++, which discretises a pressure field and three components of the

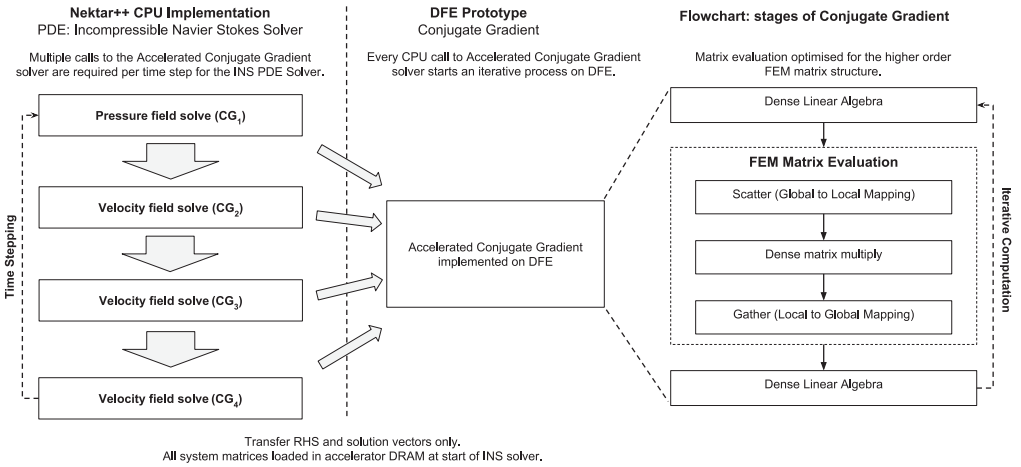

Fig. 4. System level overview of the heterogeneous execution model: Nektar++ software runs on CPU and the most computationally intense tasks are accelerated with FPGA-based acceleration boards (Maxeler DFEs).

Table I. Boundary Elemental Matrix Sizes for Different Polynomial Orders and Geometrical Shapes

Element type	**P = 1**	**P = 2**	**P = 3**	**P = 4**	**P = 5**	**P = 6**
Tetrahedron	4	10	20	34	52	74
Prism	6	18	38	66	102	146
Hexahedron	8	26	56	98	152	218

velocity field into four systems of linear equations

$$M_g\, x_g = b \tag{2}$$

to be solved one strictly after another, as shown in Figure 4, due to the sequential data dependency. Although every set of equations mentioned above has its own matrix M_g, the right hand side vector b, and the left hand side vector x_g, for convenience, in this article, we do not introduce separate notations. For our problem, among alternatives provided by Nektar++, we choose the built-in single level static condensation linear solver. This solver splits the solution into three phases: (1) constructing the boundary linear problem; (2) solving the boundary linear problem using CG method; (3) prolongating the solution to the remaining unknowns.

The matrix of the boundary linear problem has the same structure as the full system, but smaller matrix blocks; this reduces solution time. For example, the full local matrix block for hexahedral element at $P = 6$ has rank $343 = (6 + 1)^3$, while the rank of its boundary matrix block is 218. The larger the P, the higher the impact of extracting the boundary problem. Table I examples the dimensions of matrix ranks per element shape and polynomial order.

Nektar++ and our prototype accelerator implement the reduced communication reordering variant [Demmel et al. 1993] of the preconditioned CG method (Algorithm 1). This is an iterative method for solving large and unstructured systems of linear equations. It consists of repeated evaluations of the matrix-vector product of the matrix M_g present in Equation ($\mathbf{M}$ in Algorithm 1) to an auxiliary vector, in order to build a successively better approximation of the solution vector x_g ($\vec{x}$ in Algorithm 1) within a reasonable amount of iterations N_{max}. The sequence of iterations is called convergent, if the correction terms $\alpha\,\vec{p}$ become smaller and smaller starting from some iteration.

ALGORITHM 1: Preconditioned CG Method

1: **function** CG(M, P, b, y, tol, N_{max})
2: $\vec{x} \leftarrow 0,\ \ \vec{r} \leftarrow b$
3: $\vec{v} \leftarrow \mathbf{P}\vec{r};\ \ \vec{s} \leftarrow \mathbf{M}\vec{v}$
4: $\varepsilon \leftarrow (\vec{r},\vec{r}),\ \mu \leftarrow (\vec{v},\vec{s}),\ \rho \leftarrow (\vec{v},\vec{r});\ \alpha \leftarrow \rho/\mu,\ \beta \leftarrow 0$
5: **while** $(N_{step} \leq N_{max})$ & $(\varepsilon < tol^2)$ **do**
6: $\vec{p} \leftarrow \vec{v} + \beta\vec{p},\ \vec{q} \leftarrow \vec{s} + \beta\vec{q}$ ▷ Vector arithmetic
7: $\vec{x} \leftarrow \vec{x} + \alpha\vec{p},\ \vec{r} \leftarrow \vec{r} - \alpha\vec{q}$
8: $\vec{v} \leftarrow \mathbf{P}\vec{r}$ ▷ Applying preconditioner
9: $\vec{s} \leftarrow \mathbf{M}\vec{v}$ ▷ Matrix-vector multiply
10: $\varepsilon \leftarrow (\vec{r},\vec{r});\ \mu \leftarrow (\vec{v},\vec{s});\ \rho_{new} \leftarrow (\vec{v},\vec{r})$ ▷ Dot products
11: $\beta \leftarrow \rho_{new}/\rho;\ \alpha \leftarrow \rho_{new}/(\mu - \rho_{new}\beta/\alpha);\ \rho \leftarrow \rho_{new}$
12: **end while**
13: **end function**

Many real-life problems need a *preconditioner*, the matrix **P** approximating the inverse of problem matrix M_g, in order to improve the convergence behaviour and reduce the number of iterations. Since choosing a good preconditioner is problem specific, we use a basic diagonal preconditioner and leave the exploration of more interesting and complex preconditioning techniques as future work.

The main CG iteration loop consists of: four vector arithmetic operations, a multiplication by a preconditioner matrix (which, in our case of a diagonal preconditioner, is equivalent to componentwise multiplication of two data streams), and evaluating FEM matrix-vector multiply, followed by computation of dot products and scalar factors. This version of the CG method rotates the loop of a classical CG so that the dot products are co-located toward the end of its iteration. Dot product acts as a synchronisation barrier both for inter- and intra-device operation. On FPGAs, it prevents pipelining the compute stages before and after dot product; on the system scale, it prevents a device proceeding further until all devices complete evaluation of a dot product and communicate their results.

For the local matrix approach, all vectors and the preconditioner **P** are present in the global coefficient space while **M** is stored in the local coefficient space. Hence, $\vec{v}$ must be converted from the global to the local coefficient space representation v_l, in order to compute the matrix-vector product; the resulting vector s_l must then be converted from the local to the global space $\vec{s}$ to continue CG evaluation: computing the dot product at the same iteration and using vector $\vec{s}$ at the consecutive iteration. This coefficient space transformation is the major challenge for hardware acceleration of FEM simulations, especially for larger problems, where the data does not fit entirely on chip. Due to the unstructured data access patterns, this challenge is frequently addressed by sending vector data from the hardware accelerator to the CPU. However, sending vector streams to the CPUs and back via slow interconnect creates a communication bottleneck diminishing the acceleration potential. The reconfigurable architecture presented in Section 4.2 aims to avoid external communication: it performs this conversion directly on chip as part of the hardware pipeline.

For the incompressible Navier-Stokes solver run on CPU using Nektar++, the linear solver may account for more than 95% of total execution time. For our benchmark mesh problems at $P = 3$ (see Table II), CG requires ≈50–100 iterations for velocity field solves and ≈1.5k iterations for the pressure field solve. The most compute-expensive stage is the matrix-vector product at line 9 of Algorithm 1. In general, the second and third time consuming operations for MPI-parallel run of Nektar++ are inter-device communication and evaluating advanced preconditioner. In our study, we use basic preconditioner and evaluate execution on a single FPGA board orchestrated by a single

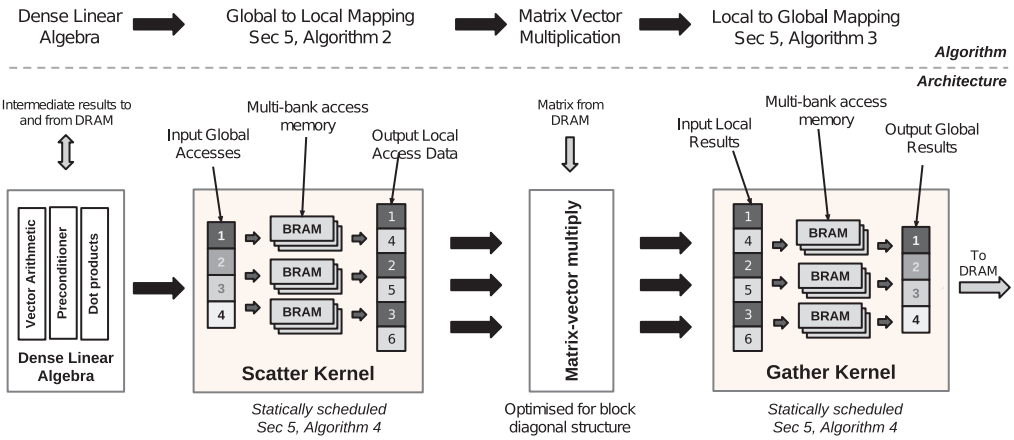

Fig. 5. Overview of the proposed dataflow CG architecture.

thread on the CPU node; the only impact of inter-device communication is the latency of CPU control.

4.2. Proposed Dataflow Architecture

The proposed dataflow architecture comprises multiple pipelined processing elements (PEs), which altogether implement one iteration of the CG Algorithm as shown in Figure 5. The implementation of the Vector Arithmetic, Diagonal Preconditioner, and Dot Product kernels is straightforward and therefore omitted for brevity. The Matrix-vector multiplier unit (MVMU) performs a large-scale block diagonal dense matrix-vector multiplication. The Scatter and Gather kernels implement the transformation from global-to-local coefficient space (and back) of input and output streams of the MVMU, respectively.

Figure 5 shows the architecture of the Scatter and Gather kernels in more detail: (1) input and output streams are processed at different rates because of the difference in the coefficient space lengths; (2) to facilitate the unstructured on-chip data access, input data for each kernel is stored in its own internal multi-bank Block RAM (BRAM) storage; (3) schedule controls which banks to read and write at which address; (4) schedule also controls when the data read from BRAM storage should form the output of the kernel; (5) additionally, the Gather kernel has adders to concurrently accumulate new local coefficient values with their partial sums stored in BRAM. The schedule is prepared on the CPU and stored in acceleration board's DRAM as a preparation step. We generate the same schedule to control both Gather and Scatter kernels, as detailed in Section 5.

The architecture maintains a number of DRAM streams for the matrix, preconditioner, and vector data, as well as two streams representing the schedule in Gather and Scatter kernels. The schedules are split into `global_schedule` and `local_schedule` since these two streams are being read at different rates. Each of the two streams coalesce gather and scatter schedules to save on the total number of DRAM streams. Similarly, vector and preconditioner data are all coalesced into a single input and single output streams.

In order to minimise the impact of CPU control latency overhead, the proposed hardware architecture performs several iterations of the loop in Algorithm 1 directly on the FPGA board as part of a single action. A state machine on chip keeps track of the current phase of execution and checks the stopping criteria.

The overall throughput of the proposed architecture is determined by the processing rates of all kernels forming the CG pipeline. To achieve maximum performance, the throughputs of all kernels must be carefully balanced. The following analysis links the balance of the design throughput with mesh parameters.

For a given d-dimensional FEM mesh, the rank of every local matrix is proportional to $O(P^d)$. Therefore, the number of arithmetic operations required for processing the full block-diagonal matrix in the MVMU is $O(kP^{2d})$, where k is the number of mesh elements; all other compute kernels perform $O(\alpha kP^d)$ operations, where $\alpha < 1$ is the mesh connectivity dependent data reduction factor, measuring how many components of a local coefficient space are mapped to a single component of a global coefficient space on average (the lower the P, the lower the α). Hence, on average, the MVMU performs $O(P/\alpha)$ more work than other kernels. In other words, the throughput of the whole design is bounded by the throughput of the MVMU. This implies that the optimal number of pipes in all other kernels is fully determined by the number of matrix rows processed per cycle by MVMU. The more cycles it takes for MVMU to process a matrix block, the less parallel all other kernels need to be.

The balance in processing rates in the proposed architecture depends on three factors: the average size of a matrix block the MVMU needs to process (which is a function of a polynomial order and proportion of various element shapes in the mesh), the throughput of DRAM interconnect, and compute parallelism in MVMU. Since the latter two factors are bound by available hardware, the polynomial order is a critical design parameter for the whole architecture that use may control.

For small values of P, the number of matrix elements fetched from DRAM is less than the available DRAM bandwidth. Therefore, to fully utilise all available DRAM bandwidth, the design must include replicated pipes, which process independent matrix blocks in a task-parallel fashion. Each pipe includes its own memory channel with independent command and data queues and requires its own on-chip buffers, thus, replication significantly complicates the architecture. Additionally, the rest of the design also needs to be more parallel, which makes it more architecturally challenging. In this work, we use $P = 3$, which is too small for MVMU to completely dominate: other kernels need to process three vector components per cycle to match the MVMU processing rate. This leads to the challenge of processing data in *bursts* in Gather and Scatter kernels: preparing the data access schedule so that reading or writing several coefficients per cycle from or to the buffers yields no access conflicts and no kernel stalls.

However, the proposed architecture scales well with the polynomial order P: for larger values of P, fewer independent pipes are required to saturate DRAM bandwidth and, hence, maximise throughput in the MVMU. This improves resource utilisation, particularly for memory controllers in the MVMU by avoiding the use of independent memory channel, commands, and data queues for each pipe. Also, the rest of the design needs to compute at most one global coefficient per cycle, which removes the need for SIMD style parallelism and removes access conflicts, therefore, simplifying both the schedule constraints and the architecture of all kernels.

In summary, the proposed architecture can efficiently implement the CG method used in the inner loop of the FEM. However, variability in P determines different choices in balancing the hardware design components to maximise performance and improve applicability to larger meshes.

5. ON-CHIP VECTOR ASSEMBLY STRATEGY

The key to achieving good performance and supporting useful problem sizes is an efficient implementation of vector assembly. Large FEM problems typically require the distribution of the computational workload across multiple cluster nodes. Therefore, the slow inter-device communication can make the vector assembly operation a

ALGORITHM 2: Transforming Vector from Global into the Local Coefficient Space

```
for local_idx from 1 to num_local_coefficients do
    global_idx, sign ← local_to_global_map[local_idx]
    v_local[local_idx] = sign * v_global[global_idx]
end for
```

bottleneck of the entire computation, which results in reduced overall performance. An efficient FPGA implementation can reduce the number of devices required, thus addressing the communication bottleneck and resulting in increased performance. To achieve the level of efficiency required to support large meshes on a single device, we propose to use a fully streaming, deeply pipelined architecture with minimal control logic. This architecture reduces computational inefficiencies and removes the need for complex circuitry for cache management, enabling more resources to be dedicated to useful computation.

In order to support the unstructured nature of memory accesses without using a cache, in this section, we present a novel method to generate the full data access schedule on the CPU, prior to accelerator execution. Given that the mesh does not change for the duration of the computation, the overhead of generating the schedule is amortised across the long runtime of the entire application. To be functionally correct and efficient, the generated schedule must:

(1) *avoid cache misses*, by completely removing data hazards associated with concurrent writing or reading of vector elements from on-chip banked memory. In Section 5.2, we formulate and solve this problem by finding a graph coloring for the data constraint graph induced by the read and write operations;
(2) *reduce on-chip storage requirements*, by minimising the number of global vector coefficients stored on chip at any given time. In Section 5.3, we formulate and solve this problem by minimising the bandwidth of the data dependency graph induced by the local to global mapping.

The algorithms proposed in Section 5.2 and in Section 5.3 are designed specifically to preserve the block diagonal structure of the local matrix M_l. As explained in Section 3, this is a vital aspect of the proposed approach, leading to increased performance, scalability, and applicability to larger problems.

By satisfying these constraints, a data access schedule can be generated which allocates every global coefficient to a corresponding memory bank for the entire runtime of the FEM accelerator. In Section 5.4, we introduce an algorithm which produces such a schedule for the proposed dataflow architecture, to control the Gather and Scatter kernels. This data access schedule induces a reordering of both the local and the global coefficient spaces which satisfies all the above constraints. While the reordering of local coefficient space is constrained to preserve block diagonal matrix structure, no constraints are placed on the global coefficient space reordering by the dataflow architecture: all computational kernels which process data in the global coefficient space implement component-wise commutative and associative arithmetic operations and are, therefore, not sensitive to data ordering.

5.1. Coefficient Space Transformation

Algorithm 2 and Algorithm 3 present the only unstructured memory accesses in the whole dataflow design. Matrix A provides the mapping from the global to the local coefficient space, which represents the *elemental decomposition* step in Figure 1. This rectangular matrix contains only 0 and ± 1, and only one nonzero entry per row, so it

ALGORITHM 3: Transforming Vector from Local into the Global Coefficient Space

```
for local_idx from 1 to num_local_coefficients do
    global_idx, sign ← local_to_global_map[local_idx]
    v_global[global_idx]  += sign * v_local[local_idx]
end for
```

is practical to represent the assembly mapping on CPU with an indexing array, and implement the global-to-local assembly as in Algorithm 2.

In the proposed architecture, the action of a global-to-local mapping is implemented in the Scatter kernel. This kernel stores the required components of the global vector in an internal buffer and produces the corresponding local coefficients by indexing the buffer using the generated data access schedule.

The transformation from local to global coefficient space may be implemented with the same indexing array, as in Algorithm 3. Since matrix A^T has several nonzero entries per row, we need to accumulate several components of a local vector contributing to the same global coefficient:

In the proposed architecture, the local to global mapping is implemented in the Gather kernel, which (1) reads data from its internal buffer, (2) accumulates new local coefficient into it and (3) either writes the result back to the buffer, (4) or outputs the result to the downstream kernel as a stream in a global coefficient space. In our architecture, these two kernels share the same data access schedule but have their own internal buffers. Sections 5.2 to 5.4 describe the procedure of converting the local to global mapping into the data access schedule described above.

5.2. Resolving Access Conflicts

In the proposed dataflow architecture, (1) all unstructured data access is performed only via on-chip buffers, (2) any data request from the local coefficient space can be resolved by a look-up into these buffers, and (3) requested data is always available in the buffers at the cycle the request has been made. Both Gather and Scatter kernels process data in *bursts*, processing k elements of both input and output streams per cycle, where k is the design parameter balancing the throughput of Gather/Scatter kernels with a matrix-vector multiplier. The locally unstructured access patterns require a multi-bank buffer with minimum k banks, satisfying the following two conditions, exemplified for the Scatter kernel:

(1) it must be possible to write k global coefficients concurrently into buffer banks with no access conflicts;
(2) it must be possible to concurrently read k global coefficients from the buffer both at the cycle of their first reference by the local-to-global mapping array and at every cycle after (to avoid future access conflicts).

Allocation of global coefficients to banks subject to the above constraints is challenging since a naive allocation strategy results in data access conflicts. Since buffer banks are shared by both reading and writing operations, the access constraints for both procedures need to be resolved *simultaneously* to yield a correct coefficient-to-bank allocation policy.

For a given local-to-global mapping, we define its access constraints graph as follows. For every i-th global coefficient of a vector v, we add a vertex g_i to the set of vertices $\mathcal{G}$ of this graph. Edges represent access constraints: when two vertices g_i and g_j are connected via an edge, the global coefficients v_i and v_j cannot be placed to the same bank. We add edges as follows:

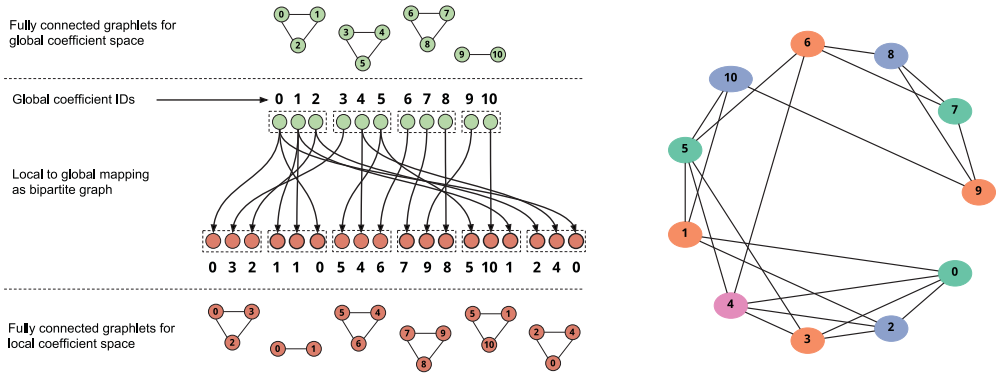

Fig. 6. Left: building the access constraints graph from the global-to-local mapping for $k = 3$. Every group of up to three coefficients contributes with its complete graphlet. These graphlets then merge to form the resulting constraint graph. This global-to-local mapping corresponds to a mesh consisting of one tetrahedral element, one prismatic element, and one hexahedral element, $P = 1$. Right: the resulting constraint graph with vertex coloring resolving all data access constraints.

(1) For every consecutive group of k local coefficients to be accessed at the same cycle, we perform k look-ups into the local-to-global mapping to resolve them to a group of k global coefficients $v_{i_1}, \ldots, v_{i_k}$. All vertices $g_{i_1}, \ldots, g_{i_k}$ within this group become pair-wise connected. This represents, for example, the read burst in Scatter kernel.
(2) Every consecutive group of k global coefficients $g_{i_1}, \ldots, g_{i_k}$ is also fully connected. This represents, for example, write burst in Scatter kernel.

These two types of fully connected vertex subgroups form connected components of the access constraints graph (see Figure 6). Disconnected vertices in this graph, if any, represent non-conflicting global coefficients which may be assigned to an arbitrary buffer bank.

We propose to employ vertex coloring of the access constraints graph to satisfy the conditions above. A vertex coloring guarantees no bank access conflicts at runtime, while *minimal* vertex coloring yields provident use of on-chip memory resource. Every color denotes its own BRAM bank, thus minimising the number of colors leads to resource optimisation. In our prototype, we use `boost::sequential_vertex_coloring` algorithm [Coleman and Mor 1983], which attempts to minimise the number of colors, but does not guarantee an optimal solution. The number of colors generated by the algorithm is the lower bound for the number of buffer banks an architecture must have to resolve all access conflicts for a given local-to-global mapping with no stalls. The number of buffer banks is a compile time parameter for our architecture, although the required number of buffer banks can only be determined at runtime (at the initialisation stage of the FEM simulation on the CPU). Therefore, the result of vertex coloring of an access constraints graph for a given local-to-global mapping determines whether a given mesh at a given P can be supported with available sets of hardware builds.

5.3. Bandwidth Reduction and Data Reordering

Our architecture requires that all data requested from the buffers is always available in the buffers at the cycle the request has been made. Satisfying this condition may result in a higher than optimal utilisation of on-chip resources. We propose to formulate the problem of minimising the resources required for correctly handling the unstructured accesses of Algorithm 2 and Algorithm 3 as a graph bandwidth minimisation problem. This approach leads to a more efficient solution than previously proposed in Burovskiy et al. 2015].

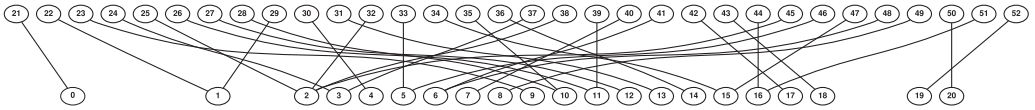

Fig. 7. Local-to-global mapping for the regular 2×2 quadrilateral mesh with four elements at $P = 2$, represented with a bipartite graph.

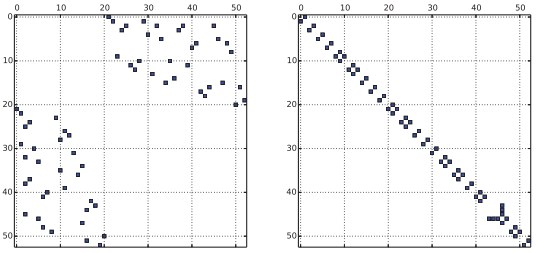

Fig. 8. Incidence matrix of a bipartite graph of the local-to-global mapping before and after bandwidth minimising vertex reordering.

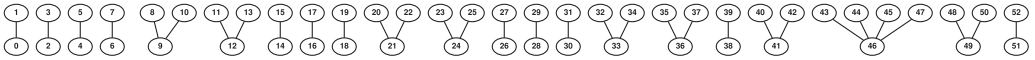

Fig. 9. Result of bandwidth minimisation reordering of a bipartite graph above.

The local-to-global mapping can be represented with a bipartite graph as shown in Figure 7. On this graph, vertices are split into two groups: a group $\mathcal{L}$ represents local coefficients (the top row on Figure 7) and another group $\mathcal{G}$ represents global coefficients (the bottom row on Figure 7). No two vertices from the same group are connected; a vertex $l_i \in \mathcal{L}$ is connected by an edge to a vertex $g_j \in \mathcal{G}$ if and only if the i-th component of a local-to-global mapping array references the j-th global coefficient. Figure 7 illustrates such a bipartite graph for the regular quadrilateral mesh with four elements arranged into a 2×2 grid. This problem has 21 global coefficients and 32 local coefficients, which gives 53 vertices of the bipartite graph in total.

The bandwidth of this graph, or, equivalently, the bandwidth of its incidence matrix, defines (1) the maximum offset of indirect access to vector streams, and (2) the lifetime interval in the buffer. The lifetime interval is the duration in cycles between the first and last reference of the global coefficient by the local-to-global mapping.

Since we need to store all data on chip throughout its lifetime, the buffer size required for correct execution should be greater than or equal to the largest lifetime interval.

To minimise the required buffer size for a particular mesh, we can apply a bandwidth reduction algorithm such as Cuthill and McKee [1969]. The result of Cuthill-McKee reordering is shown in Figure 8. The bandwidth reduction induces a reordering of graph vertices whose result is shown in Figure 9.

The newly generated ordering on vertices of this graph defines the new order of visits to both local and global coefficients such that

(1) the global coefficient is required to be present in the on-chip buffer for the minimum number of cycles;
(2) the global coefficients are enumerated and visited strictly in order of their first reference by the local-to-global mapping.

However, directly applying the Cuthill-McKee reordering may destroy the block-diagonal dense structure of the local FEM matrix. This is because reordering local coefficients is equivalent to reordering rows and columns of the local FEM matrix,

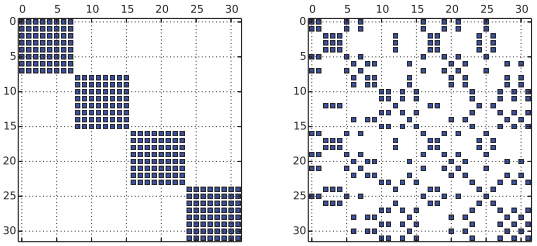

Fig. 10. Structure of a local matrix before (on the left) and after (on the right) reordering of a local coefficient space induced by the bandwidth minimisation of a bipartite graph shown on Figure 7.

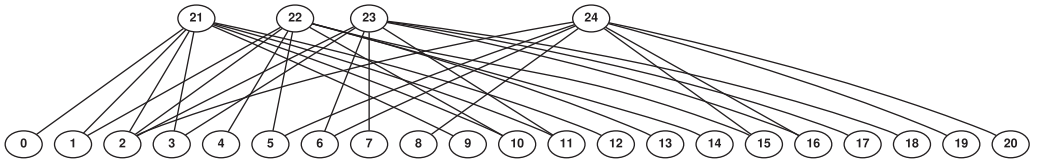

Fig. 11. Block-bipartite graph for the same mesh as above. Local coefficients are grouped together with respect to the matrix block to which they correspond.

without taking into account the local block structure formed by its nonzero entries. Figure 10 shows the effect of such a reordering on the structure of a local FEM matrix.

To preserve the block structure of the FEM local matrix, which is crucial for the performance and scalability of the proposed architecture, we propose to construct a *block-bipartite graph* and minimise its bandwidth instead. First, we add a set of vertices $\mathcal{G}$, as above, and a set of vertices $\mathcal{M}$, which contains one vertex m_j for each local matrix block M^e_j. We then partition local coefficients with respect to their matrix blocks (as on Figure 3); there is an edge between every $g_i \in \mathcal{G}$ and every $m_j \in \mathcal{M}$ if and only if the local coefficient mapped to g_i (by the local to global mapping) belongs to the group of local coefficients for the matrix block M^e_j. The block-bipartite graph for the above example is shown on the Figure 11.

The bandwidth minimisation of a block-bipartite graph induces a block-reordering on the local coefficient space; this formulation captures two constraints: (1) all local vector components must be on chip when MVMU is processing a corresponding local matrix block; (2) rows corresponding to exactly one matrix block must be processed in a contiguous range of cycles, to avoid destroying the block diagonal structure of the matrix.

The algorithm to devise a streaming schedule which minimises the required on-chip resources then becomes:

(1) build the bipartite graph, connecting global coefficients with local matrix blocks;
(2) minimise its bandwidth;
(3) reorder local matrix blocks, which induces block-reordering of local coefficients;
(4) reorder global coefficients accordingly;
(5) update the local-to-global mapping array to match the reorderings.

The gather kernel adds one constraint to the local data reordering: due to the latency of a floating point accumulator, two local coefficients contributing to the same global vector component should come to the accumulator at least L_{fp} cycles one after another, where L_{fp} is the latency of both accumulator and BRAM buffer access. This allows to complete previous accumulation before the next summand arrives. The alternative is to use C-slowed accumulation, store partial results in the buffer at separate addresses,

ALGORITHM 4: Generating Access Schedules

```
for local_idx from 1 to num_local_coefficients do
    global_idx, sign ← local_to_global_map[local_idx]
    bank_idx ← vertex_coloring[global_idx]
    if first_referenced[global_idx] then
        if available_address_list[bank_idx] is empty then
            re-run algorithm with more colors or larger buffers
        else
            address[global_idx] ← available_address_list[bank_idx].pop()
            global_schedule.push({address[global_idx], bank_idx})
        end if
    end if
    if last_referenced[global_idx] then
        available_address_list[bank_idx].push(address[global_idx])
    end if
    local_schedule.push({address[global_idx], bank_idx, sign})
end for
```

and build an additional reduction pipeline stage. We avoid this problem by further aggregating the local coefficient space: when two local coefficients $l_i \in m_j$ and $l_k \in m_l$ are mapped to the same global coefficient g_s, and the number of cycles between reads of l_i and l_k from the local-to-global mapping is less than L_{fp}, we merge vertices m_j and m_l of a block-bipartite graph. This facilitates three goals:

(1) bandwidth minimisation problem is completely decoupled from the adder latency data hazard problem;
(2) enforcing appropriate intervals between consecutive references of same global coefficient is done via local post-processing of the $\mathcal{M}$ vertex set of the block-bipartite graph after bandwidth minimisation.
(3) such decoupling guarantees both Gather and Scatter kernels use the same ordering on local and global coefficients.

To summarise, the proposed bandwidth minimisation of the block-bipartite graph reduces the lifetime intervals for the data stored in the internal buffers of the Gather and Scatter kernels. The reduced lifetime intervals enable the processing of the unstructured mesh with smaller on-chip buffers or, equivalently, enables the processing of larger mesh problems with a given hardware architecture. Also, since both Gather and Scatter kernels use the same ordering on coefficient spaces, these two kernels do not need separate data access schedules: their data access patterns to their (separate) on-chip memory buffers are identical. The resulting reorderings on the coefficient spaces (1) enforce the linear streaming of global vector data at the input of the Scatter kernel and the output of the Gather kernel, and (2) guarantee no stalls of the Scatter kernel when the local coefficient is out of the buffer.

5.4. Generating Schedules

We translate the local-to-global mapping into two separate control streams for the Gather and Scatter Kernels. The schedule generation procedure consists of the following steps:

(1) (optional) bandwidth minimisation: for a local to global mapping array generated by the FEM package, produce a new local-to-global mapping to minimise the on-chip buffer size requirements;
(2) generate an access constraints graph from a given local-to-global mapping array and compute its vertex coloring;

Table II. Characteristics of Benchmark Problems

Mesh name **Work**	**DBTV** [Rocco 2014]	**Naca 1L** [Chow et al. 1997]	**N/A** [Hu et al. 2008]
Mesh type	unstructured	unstructured	structured
P used in this article and Hu et al. [2008]	3	3	1
P normally being used	7	6	
Num. tetrahedra, block size for $P = 3$	34,887; 20×20	58,728; 20×20	up to 48,000; 4×4
Num. prisms, block size for $P = 3$	0	11549, 38×38	0
Num. local coefficients	697,740	2,758,986	192,000
Num. global coefficients	158,905	1,077,373	?
Floating point precision	double	double	single
Dense matrix size	106.7MB	820MB	up to 2.92MB
One CG vector size	1.21MB	8.21MB	?

(3) perform "register allocation:" allocate every global coefficient its buffer bank ID and the address in that buffer (the access to every bank is unstructured).

For every global coefficient, the vertex coloring determines its buffer bank, but we still need to allocate a particular address within a buffer bank and check there is enough space in all banks for storing all simultaneously live vector components. We suggest a one-pass procedure shown in Algorithm 4, which simulates both reads and writes to the buffer. Algorithm 4 assumes all data access conflicts have been resolved and the coefficient space has been reordered. Also, it assumes every buffer bank has a fixed capacity, defined at design compile time.

This algorithm reads the local-to-global mapping sequentially and tries to allocate every new global coefficient an available slot in a buffer, as determined by its color. The algorithm maintains the pool of available slots in each buffer. Initially, the whole buffer bank is available. Once the global coefficient is referenced for the last time (its lifetime interval ends), its bank address joins the pool of available bank storage and may be overwritten in the same cycle with a new coefficient.

6. EVALUATION

In this section, we evaluate our approach to FEM vector assembly to demonstrate that this enables a large-scale FEM simulation using a single FPGA-based acceleration board, connected to a CPU server. The goal of this work is to show the maximum scale of the FEM problems supported by a single, state of the art FPGA-based acceleration board. This indicates how large FEM problems could be solved by a cluster of FPGAs. We leave additional performance optimisation opportunities and multi-FPGA evaluations to the future work.

The spectral/hp framework Nektar++ [Cantwell et al. 2015] is written in C++ and parallelised with MPI. It implements the Incompressible Navier-Stokes solver (INS) we use in this study. This solver has been recently used on the ARCHER supercomputer to study the NACA 0012 problem with 243,000 elements at $P = 5$ and $P = 6$, which accounts for 16.7 and 25.3 million degrees of freedom (global coefficients) respectively [Lombard et al. 2015].

In this article, we use smaller mesh problems whose parameters are shown in Table II. Nevertheless, these meshes are substantially larger than FEM meshes evaluated on FPGAs previously [Hu et al. 2008]. The Naca 1L mesh listed in Table II represents a coarser tessellation into 70,277 elements of the same NACA 0012 geometry as above; we also use a lower polynomial order: $P = 3$, which accounts for 1 million global coefficients. Even these two benchmark problems provide a substantial challenge for FEM acceleration using FPGAs.

Table III. Schedule Parameters of the Hardware Architecture, Per Scatter/Gather Kernel

	DBTV mesh, 64bit	**Naca 1L, 64bit**
	this work / Burovskiy et al. [2015]	this work / Burovskiy et al. [2015]
Number of buffer banks	10 / 48	10 / 90
Size of a buffer bank	2,048 / 2,048	4,608 / 1,280
Num iterations to converge	deterministic / 1	deterministic / 4,411
Estimate utilisation of M20K	80 / 336	180 / 630

In our previous work [Burovskiy et al. 2015], we implemented the proposed architecture using MaxCompiler 2014.1, targeting Maxeler MAX4 Maia acceleration board with an Altera Stratix V D8 chip, 48GB on–board DRAM, and Infiniband interconnect to the CPU server. In this article, we use the same implementation of the dataflow architecture, however, we recompile it for different schedule parameters using MaxCompiler 2014.1. In both works, we built our designs for 150Mhz on-chip clock frequency and 666Mhz memory controller frequency, with three MPEs in MVMU and three compute pipelines for the rest of the design; each MPE performs 24 floating point multiplications per cycle.

The reference CPU implementation of the Nektar++ framework uses double precision floating point arithmetic for all compute operations and data representation. Our design maintains double precision floating point vectors and performs all arithmetic operations in double precision; however, the matrix data is stored in single precision in on-board DRAM and cast to double precision on the fly.

In Burovskiy et al. [2015], we conclude that the design supporting DBTV mesh fits the chip and leaves enough resources for architectural optimisations, while the corresponding design for the Naca 1L mesh overfits the available resources of the Stratix V D8 FPGA by 239 M20Ks. The schedule generated by our heuristic algorithm requires more BRAMs than available on chip after other PEs and general infrastructure. In this work, we evaluate a more efficient schedule generation procedure. It produces a more resource efficient schedule, which enables the architecture for the Naca 1L mesh problem to fit on a single chip.

The proposed scheduling algorithm determines the number of BRAM buffers and their sizes necessary for supporting on-chip assembly for both test meshes. Our scheduling algorithm takes the local-to-global mapping data, as well as the list of matrix block sizes directly from Nektar++, running against the mesh of interest. These numbers become compile time parameters to our hardware architecture. Table III presents the results generated by our graph-based schedule (marked as “this work”) and our heuristic algorithm described in previous work [Burovskiy et al. 2015]. In both cases, the schedule is generated for the architecture supporting read/write burst size three in both Gather and Scatter kernels, and vector data in double floating point precision. The scheduling algorithm takes the number of buffer banks and their depth as parameters and aborts if the local-to-global mapping is too demanding for these parameters. Table III presents the parameters for the graph-based scheduling procedure that minimise spare buffer space. Note, both buffer bank sizes 2,048 and 4,608 generated by the scheduling procedure are multiples of 512, which accounts for two physical M20K blocks in configuration 512×32.

Table IV and Table V present the place and route results for the Scatter and Gather kernels, as well as the whole design compiled against parameters shown in Table III. We present only BRAM utilisation for the Scatter and Gather kernels since this is the most critical resource for these kernels. In our previous work [Burovskiy et al. 2015] we show more detailed resource utilisation for these kernels, which shows negligible amounts of LUTs and FFs used in these kernels. We conclude that an optimised schedule improves resource utilisation for both hardware builds, even without any resource optimisations

Table IV. Does Not Change with Problem Size: Area Utilisation for the MVMU and Vector Arithmetic Units

Kernel	BRAMs	LUTs	FFs	DSPs
Stratix V DDR	149	46,327	52,094	
Memory Controller (MVMU)	652	17,165	64,581	
PCIe (MVMU)	100	6,713	7,828	
FIFOs (MVMU)	184	568	709	
Vector arithmetic, dot products, preconditioner	127	35,886	43,290	36
MVMU, 3 x MPEs	201	105,129	145,147	288

Table V. Varies with Problem Size: Area Utilisation for Gather/Scatter Kernels and the Whole Design: This Work

Resource Usage	DBTV	Naca 1L
Scatter Kernel (BRAM)	85 / 341	186 / 638
Gather Kernel (BRAM)	101 / 358	198 / 652
Total (including MVMU)		
BRAM	1,703 / 2,215	**1,835** / **2,806**
LUT	128K / 133K	144K
FF	189K / 270K	341K
Available	2,567 BRAMs, 524K LUTs, 1,024K FFs, 1,963 DSPs	

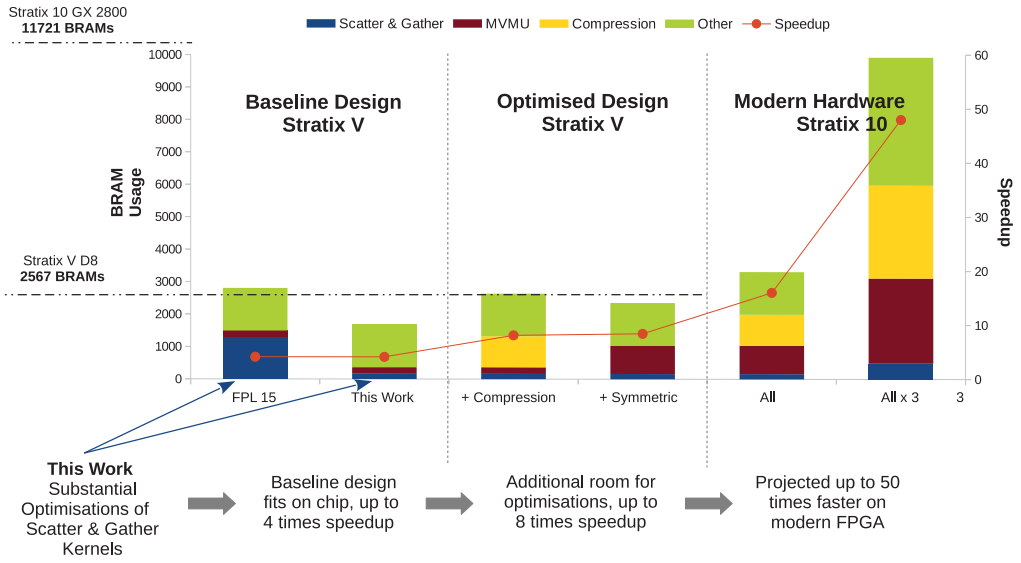

Fig. 12. Projected speed-up opportunities over multicore CPU software Nektar++.

of the remaining design. With this improvement, the Naca 1L problem fits the chip, and the remaining spare 732 M20K enable future performance optimisations.

Finally, Figure 12 shows the benefits in terms of acceleration potential of the proposed solution. First, we note that the approach presented in this work achieves a substantial resource saving for the Scatter and Gather kernels compared to previous work: the baseline design fits on chip for the much larger NACA 1L mesh, thus enabling a potential four times speedup over an optimised implementation of Nektar++ running with 6 MPI ranks on an Intel Xeon E5-2640 [Burovskiy et al. 2015]. Furthermore, the additional spare resources could be used for important optimisations such as a simple compression algorithm, which could deliver up to two times memory throughput for

matrix data at the cost of 12 additional BRAMs per MVMU input [Grigoras et al. 2015]. Alternatively, the spare resources could be used to implement a symmetric matrix multiplier to exploit the symmetry of the element-local blocks which arise in the FEM. A naive implementation which loads the entire block (up to 60×60 elements, at $P = 3$) and supports an MVMU with 72 inputs as the one used in this work may require approximately 800 BRAMs: an additional 7 BRAMs per input since 512 single precision entries can be stored per BRAM plus the cost of the baseline design. This number can be reduced substantially by applying additional optimisations such as fused floating point adder trees and additional exploration of vendor tool configurations, particularly level of pipelining, mapping of computations to DSPs, increased clock-frequency, and so on. Both the compression and symmetric multiplier optimisations fit *independently* on the Stratix V, due to the resource savings achieved by applying the method proposed in this work. While they may not fit on the Stratix V chip *simultaneously*, the current generation of high-end chips such as the Stratix 10 GX 2800, could not only accomodate both optimisations, but the design could even be replicated three times to provide increased throughput, assuming sufficient memory bandwidth, such as through the use of a hard memory controller, which is supported in Stratix 10. Overall, the proposed implementation could deliver up to eight times acceleration over the aforementioned Nektar++ CPU implementation on a Stratix 5 and up to 50 times acceleration on a high-end Stratix 10.

7. CONCLUSION

We have proposed a high performance FPGA design for the CG Solver specialised to spectral/hp FEM problems on arbitrarily unstructured meshes, a novel approach to performing FEM assembly directly on chip and the graph-theoretic method of building the schedule of access to on-chip buffers. This improves the resource efficiency of the proposed architecture and enables the simulation of a large-scale spectral/hp FEM problems with up to 70,277 mesh elements and ≈1M global degrees of freedom to a single FPGA acceleration board. This is the largest FEM problem proposed for FPGA acceleration so far. Our study shows a good potential in accelerating FEM on FPGA. A number of performance and further resource utilisation improvements is left for future work. This includes exploiting the symmetric structure of FEM matrix, advanced preconditioners, multi-FPGA design, data compression, and use of reduced precision.

REFERENCES

Pavel Burovskiy, Paul Grigoras, Spencer Sherwin, and Wayne Luk. 2015. Efficient assembly for high order unstructured FEM meshes. In *Proceedings of the 2015 25th International Conference on Field Programmable Logic and Applications (FPL'15)*. IEEE, 1–6.

C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, et al. 2015. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications* 192 (July 2015), 205–219.

C. D. Cantwell, S. Yakovlev, R. M. Kirby, N. S. Peters, and S. J. Sherwin. 2014. High-order spectral/hp element discretisation for reaction-diffusion problems on surfaces: Application to cardiac electrophysiology. *Journal Of Computational Physics* 257 (2014), 813–829.

Gary C. T. Chow, Paul Grigoras, Pavel Burovskiy, and Wayne Luk. 2014. An efficient sparse conjugate gradient solver using a Beneš permutation network. In *Proceedings of the 2014 24th Internatinal Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–7.

Jim S. Chow, Gregory G. Zilliac, and Peter Bradshaw. 1997. Mean and turbulence measurements in the near field of a wingtip vortex. *AIAA Journal* 35, 10 (1997), 1561–1567.

Thomas F. Coleman and Jorge J. Mor. 1983. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.* 20, 1 (1983), 187–209. DOI:http://dx.doi.org/10.1137/0720013

Elizabeth Cuthill and James McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*. ACM, 157–172.

James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. 1993. Parallel numerical linear algebra. *Acta Numerica* 2 (1 1993), 111–197.

Yousef Elkurdi, David Fernández, Evgueni Souleimanov, Dennis Giannacopoulos, and Warren J. Gross. 2008. FPGA architecture and implementation of sparse matrix–vector multiplication for the finite element method. *Computer Physics Communications* 178, 8 (2008), 558–570.

Boris G. Galerkin. 1915. Rods and plates. Series solution of some problems in elastic equilibrium of rods and plates., Vol. 19. Vestnik Inzhenerov i Tekhnikov, 897–908.

Paul Grigoras, Pavel Burovskiy, and Wayne Luk. 2016a. CASK: Open-source custom architectures for sparse kernels. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. ACM, 179–184.

Paul Grigoras, Pavel Burovskiy, Wayne Luk, and Spencer Sherwin. 2016b. Optimising sparse matrix vector multiplication for large scale high order FEM problems on FPGAs. In *Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL'16)*.

Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. 2015. Accelerating SpMV on FPGAs by compressing nonzero values. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 64–67.

Jing Hu, Steven F. Quigley, and Andrew Chan. 2008. An element-by-element preconditioned conjugate gradient solver of 3D tetrahedral finite elements on an FPGA coprocessor. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'08). International Conference on*. IEEE, 575–578.

Kazuki Ikushima, Shinsuke Itoh, and Masakazu Shibahara. 2015. Development of idealized explicit FEM using GPU parallelization and its application to large-scale analysis of residual stress of multi-pass welded pipe joint. *Welding in the World* 59, 4 (2015), 589–595.

George Karniadakis and Spencer Sherwin. 2013. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford University Press.

Gerhard Lienhart, Daniel Gembris, and Reinhard Männer. 2005. Perspectives for the use of field programmable gate arrays for finite element computations. In *Proceedings of the COMSOL Multiphysics User's Conference*.

Jean-Eloi W. Lombard, David Moxey, Spencer J. Sherwin, Julien F. A. Hoessler, Sridar Dhandapani, and Mark J. Taylor. 2015. Implicit large-eddy simulation of a wingtip vortex. *AIAA Journal* (2015), 1–13.

G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. 2013. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids* 71, 1 (2013), 80–97.

Maciej Piechotka. 2013. *Unstructured Mesh Fluid Dynamics Using the Flux Reconstruction Method on an FPGA Dataflow Engine*. Master's thesis. Imperial College London.

Gabriele Rocco. 2014. *Advanced Instability Methods using Spectral/hp Discretisations and their Applications to Complex Geometries*. Ph.D. Dissertation. Imperial College London.

Gilbert Strang and George J. Fix. 1973. *An Analysis of the Finite Element Method*. Vol. 212. Prentice-Hall Englewood Cliffs, NJ.

Marcel van der Veen. 2007. *Sparse Matrix Vector Multiplication on a Field Programmable Gate Array*. Master's thesis. University of Twente.

Peter E. Vincent, Patrice Castonguay, and Antony Jameson. 2011. A new class of high-order energy stable flux reconstruction schemes. *Journal of Scientific Computing* 47, 1 (2011), 50–72. DOI:http://dx.doi.org/10.1007/s10915-010-9420-z

Peter E. J. Vos, Spencer J. Sherwin, and Robert M. Kirby. 2010. From H to P efficiently: Implementing finite and spectral/Hp element methods to achieve optimal performance for low- and high-order discretisations. *Journal of Computational Physics* 229, 13 (July 2010), 5161–5181.

Guiming Wu, Xianghui Xie, Yong Dou, and Miao Wang. 2013. High-performance architecture for the conjugate gradient solver on FPGAs. *IEEE Transactions on Circuits and Systems II: Express Briefs* 60, 11 (2013), 791–795.

Received April 2016; revised August 2016; accepted November 2016