Efficient Assembly for High Order Unstructured FEM Meshes

Pavel Burovskiy*, Paul Grigoras*, Spencer Sherwin‡, Wayne Luk*
*Department of Computing, Imperial College London
‡Department of Aeronautics, Imperial College London
{p.burovskiy, paul.grigoras09, s.sherwin, w.luk}@imperial.ac.uk

***Abstract*—The Finite Element Method (FEM) is a common numerical technique used for solving Partial Differential Equations (PDEs) on complex domain geometries. Since FEM meshes are large and unstructured, an efficient mapping of the computation onto FPGAs is particularly challenging. The focus of this paper is on assembly mapping, a key kernel of FEM, which induces the sparse and unstructured nature of the problem. We translate FEM vector assembly mapping into data access scheduling, thus performing vector assembly directly on the FPGA, as part of the hardware pipeline. We show how to efficiently partition the problem into dense and sparse sub-problems which map well onto FPGAs. The proposed approach, implemented on a single FPGA could outperform highly optimised FEM software running on two Xeon E5-2640 processors.**

Keywords*—*Finite Element Methods, assembly, sparsity, dataflow

I. Introduction

Finite Elements Method (FEM) is an ubiquitous tool in many areas of science and engineering, such as geophysics, fluid and structure mechanics, electromagnetics, biomedicine [1]. 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 a solution.

The FEM supports unstructured spacial discretisations which is essential for accurately capturing complex geometrical shapes found in industrial and scientific applications. The FEM meshes represent the computational domain by splitting it into *elements*: simple geometrical shapes (e.g. tetrahedra) which form the support of a piece-wise polynomial approximation of a PDE solution, local to each element. However, the ways of enforcing the continuity of a global solution are substantially different for Continuous and Discontinuous Galerkin methods [2]. In this paper we concentrate on the Continuous Galerkin method, which implicitly couples the mesh-elemental solutions globally by discretising differential operator on the whole computational domain with a large sparse system of linear equations. The sparse problem matrix is *assembled* from mesh-elemental dense matrix discretisations by means of an *assembly mapping*. This assembly mapping is defined based on the element adjacency graph, which is generally very sparse and frequently unstructured. However, when *higher order polynomials* are used, the data structures become more regular.

Solving time-dependent PDEs using FEM is a classical high performance computing (HPC) problem, where similarly structured computation is repeated for many time steps, which may last for hours. Traditionally, FEM codes execute on distributed CPU-based HPC clusters. Although using hardware accelerators for FEM problems is highly anticipated, limited progress has been made so far for the Continuous Galerkin method. Previous FPGA efforts [3], [4], [5] accelerate classical FEM and constrain either problem size or sparsity pattern/mesh topology.

In this paper we discuss the efficient implementation strategy of assembly for high order FEM meshes directly on FPGA, to enable FPGA acceleration of larger FEM problems with an arbitrary connectivity structure. Our contributions are:

1) A novel approach to high order FEM vector assembly designed for execution directly on FPGA as part of the iterative linear solver: the preconditioned Conjugate Gradient Method (Section III).
2) Prototype implementation of the solver with on-chip assembly on Maxeler Maia FPGA acceleration board, as custom hardware accelerator to Nektar++ spectral/hp FEM framework software (Section IV).
3) Experimental evaluation of the proposed approach on benchmark meshes. We provide a performance comparison for both accelerated and non-accelerated Nektar++ runs on the same Intel Xeon server (Section V).

II. Background

For the Continuous Galerkin method solving PDE is reduced a solution to a large distributed sparse linear system of equations

$$M_g \, x_g = b. \tag{1}$$

The unknowns x_g, also referred to as *degrees of freedom* define a piece-wise approximation to the PDE solution. Frequently, the run time of a linear solver, such as preconditioned Conjugate Gradient (CGM; sketched in Algorithm 1), dominates the whole computation and thus forms a performance bottleneck. In the example of incompressible Navier Stokes solve on a substantially unstructured mesh, problem (1) may take more than 95% of total execution time. Typically, for a time-dependent PDE both problem geometry (mesh) and system matrix M_g remain unchanged for all time steps, so it is affordable to pre-process the matrix representation to improve time stepping performance.

Matrix (operator) evaluation strategies. The structure of the matrix M_g can be exploited to optimise compute node local evaluation of matrix-vector products $s_g := M_g \, w_g$ in the CGM while-loop body. The *global matrix approach* [2] suggests to explicitly form the *global matrix* M_g once and store it in a sparse storage format. The alternative *local matrix approach*

Algorithm 1 Preconditioned Conjugate Gradient Method [6]

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

fuses matrix construction operations with matrix-vector multiply into a single *operator*. We use index g as in M_g to denote the choice of a global matrix approach and index l to denote local matrix approach accordingly.

Figure 1 from [7] illustrates this concept. The splitting of mesh into elements is denoted as "elemental decomposition" and mathematically represented with an incidence matrix A, while taking the adjacency information into account is named "global assembly" and is mathematically represented with A^T (so that $w_g = A^T w_l$). For the global matrix approach the matrix-vector multiply takes the form

$$s_g = M_g w_g = \left(A^T M_l A\right) w_g, \tag{2}$$

where sparse matrix M_g (Figure 1(a)) is assembled from a dense block-diagonal matrix M_l (the center of Figure 1(b)) by multiplying it from left and right with matrices A^T and A accordingly; every dense matrix block is an element-local discretisation of a differential operator. Such assembly of a problem matrix is done only once as a preparation step. This sparse matrix is then repeatedly multiplied to a vector w_g in a *global coefficient space*.

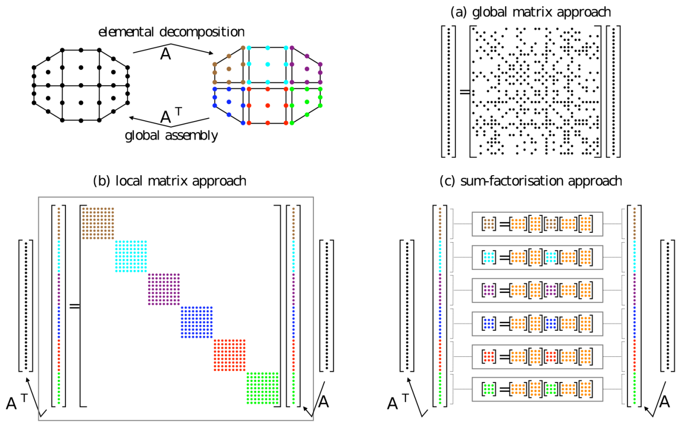
elemental decomposition
A
A T
global assembly
(a) global matrix approach
=
(b) local matrix approach
=
A T
A
(c) sum-factorisation approach
A T
A

Fig. 1. Three different matrix evaluation strategies based on elemental decomposition of a mesh. Cases (a) and (b) are relevant to the context of CGM solver.

Figure 1(b) shows the same computation restructured to represent the *local matrix approach*,

$$s_g = A^T M_l \left(A w_g\right) = A^T \left(M_l w_l\right), \tag{3}$$

which implies first scattering vector w_g to a vector w_l in *local coefficient space* ($w_l = A w_g$), then performing dense matrix vector multiply $s_l = M_l w_l$ and finally gathering resulting vector s_l back to its global coefficient counterpart s_g: $s_g = A^T s_l$. Note this ordering of computes refrains from explicit formation of a sparse matrix. Instead, the only sparse computations are vector gather and scatter. [7] presents the performance of matrix evaluation strategies as a function of a polynomial order and mesh structure. For the higher polynomial orders the local matrix approach becomes more computationally efficient due to more regular access to the memory, since the complexity of dense matrix multiply scale as $O(P^2)$, while the complexity of gather/scatter scale as $O(P)$.

In this paper, we focus on a local matrix approach: our goal is FEM evaluation for higher polynomial orders on FPGA. The case (b) on Figure 1, read from the right to the left, might be viewed as diagrammatic representation of a hardware pipeline: initial vector is mapped by matrix A to a larger size vector, which is then multiplied to a block-diagonal dense matrix, then the resulting vector is reduced back to the original size using A^T. We map this execution flow to a dataflow design, present in the Section IV.

Related work. On CPUs and GPUs the Finite Element Methods assembly is an area of active research [8], [7] where the focus is on an optimal matrix evaluation *operator* strategy, trading the compute redundancy to a degree of more regular memory access pattern.

In the FPGA community a number of past publications [3], [4], focused on the Sparse Matrix Vector multiply (SpMV) architectures, specialised to FEM matrices of particular connectivity. [9] is the Conjugate Gradient solver architectures with SpMV kernel, evaluated on the sparse matrices with dimensions up to 66,127. [10] is another SpMV-based Conjugate Gradient solver on FPGA, evaluated on sparse matrices of rank up to 63,838. Only [5] can be directly compared to this work: they implement the local matrix approach evaluated on regular geometric domains with up to 48,000 tetrahedra. Assuming the first order approximations, the number of local degrees of freedom for this test mesh equals 192,000. Our test meshes presented in the Table I have 3.6 and 14.3 times more local degrees of freedom.

III. On-chip Assembly Strategy

In this section we describe a novel implementation strategy of an assembly mapping, whose primary objective is to perform assembly directly on the hardware accelerator. As we show in the next section, this allows us to coalesce all stages of local matrix evaluation described above (Figure 1 (b)) to form a hardware pipeline.

Action of global to local assembly mapping to a vector: scatter. Matrix A provides the mapping from global coefficient space to the local coefficient space ("elemental decomposition" on Figure 1). This matrix has only values 0 and ± 1, and only one nonzero entry per row, so it is practical to represent the assembly mapping on CPU with an indexing array(s), and

Algorithm 2 Action of global to local mapping to a vector

```
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
```

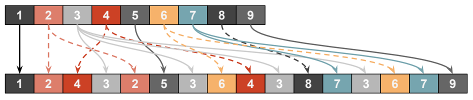

Fig. 2. Global to local mapping for 2x2 quadrilateral mesh, P=1, applied to a vector in global coefficient space, i-th component of which equals i. For this use case, the components of sign array are all positive.

implement the global to local assembly as in Algorithm 2. The data flow in Algorithm 2 is presented on Figure 2, where the arrow directions are defined by a local_to_global_map array.

Action of local to global assembly mapping to a vector: gather. The action of A^T matrix to a vector can be implemented on the CPUs similarly. Since matrix A^T has several nonzero entries per row, its action on a vector in local coefficient space is add-reduce (gather), as in Algorithm 3. The data flow for the Algorithm 3, applied to the output vector of Algorithm 2, is shown on Figure 3. In the context of local matrix operator evaluation, a vector scattered to the local coefficient space is an input to a block-diagonal dense matrix multiplier, while the gather algorithm is run on the matrix multiplier output (Figure 1, case (b)).

Custom hardware execution in absence of on-chip caches motivates us to avoid (completely unstructured) indirect access to the vector components. We re-structure the local_to_global_map array so that the whole local matrix multiply operator maps to the hardware pipeline with a linear access to both its input- and output vectors. This implies implementing an equivalent of Algorithms 2 and 3 as two hardware kernels providing the conversion between coefficient spaces on the fly by reading and writing data at different rates, ideally with no stalls and (on-chip) memory access conflicts, and with only a static latency overhead. As presented in Section IV, we implement a custom cache for each of two assembly kernels as an array of BRAM buffers, storing global degrees of freedom on-chip from the point of their first reference by the local_to_global_map and until their last reference. Once a global coefficient is read from the input stream, it stays in BRAM until its last reference, and then is evicted from the buffer to save on space. In order to avoid data hazards and BRAM bank conflicts, we translate local_to_global_map to the scheduling arrays on the CPU, and then stream these scheduling arrays along with other

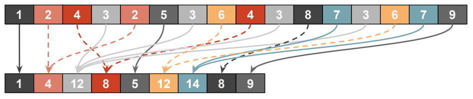

Fig. 3. Local to global mapping for 2x2 quadrilateral mesh, P=1, applied to a vector in local coefficient space.

Algorithm 3 Action of local to global mapping to a vector

```
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
```

CGM vectors from DRAM. The size of on-chip buffer memory should be enough to keep all global degrees of freedom already referenced, but for which assembly is incomplete. Nevertheless, it can be substantially smaller than the size of a input vector(s) since there are a large number of degrees of freedom fully assembled long before streaming of a vector is complete.

Data reordering. Figure 2 shows that the assembly mapping may reference entries of a vector in global coefficient space not in the order of their streaming from e.g. DRAM. By re-ordering the global coefficient space and updating scheduling arrays we enforce linear access to the input vectors. Note the other parts of the CGM algorithm such as vector arithmetic operations and computing dot products are element-wise operations on CGM vectors in the global coefficient space; this implies the re-ordering of the global coefficient space preserves the correctness of these operations. When the preconditioner is diagonal, arbitrary re-ordering of the global coefficient space is also possible.

Scheduling arrays help to completely resolve BRAM bank conflicts during the gather/scatter execution, as well as avoid data hazards associated with floating point accumulation. We aim to read/write exactly k local coefficients every cycle (*local scheduling*) and k global coefficients at some enabled cycles (*global scheduling*), where k is the supported concurrency level (number of pipes in the hardware). We first allocate every coefficient its BRAM buffer ID and the address in that buffer for its "lifetime" interval between the first and the last reference by the assembly mapping. Every BRAM buffer implements a random access cache; as long as some coefficient's "lifetime" interval ends, its BRAM space joins the pool of available BRAM storage. We then produce two co-related schedules instructing the hardware when to access a coefficient at a BRAM location for reading and when to access for writing.

To prepare such scheduling, we suggest to symbolically execute vector assembly on the CPU: by reading the local_to_global_map array entries in batches of the size k one can build a list of coefficient-to-BRAM access constraints. The unstructuredness of original local_to_global_map array creates a number of per-cycle access constraints to coefficients in their buffers. Since the same array of BRAM buffers with same coefficients is shared by two data access procedures (reading and writing), the access constraints for both procedures need to be resolved *simultaneously* to yield a correct coefficient-to-BRAM space allocation policy. The access schedules also differ for the gather and scatter operations. Scatter needs only to resolve access/assignment BRAM conflicts. Gather also resolves the access problem to the partial sums stored in the BRAM: due to the latency of BRAM access and a floating point accumulator one must not touch a coefficient until previous accumulation is complete. Since we do not re-order local coefficient space (e.g. the entries of local_to_global_map), it is possible that two or more partial contributions to a coefficient come before accumulation is complete; in this case scheduling procedure should determine the number of partial sums to be stored at different BRAM locations, and these partial sums are to be reduced at the output. Also, reading these partial sums for the output scheduling should account for latency of accumulation to BRAM, thus reading coefficients few cycles after they are referenced by the original local_to_global_map.

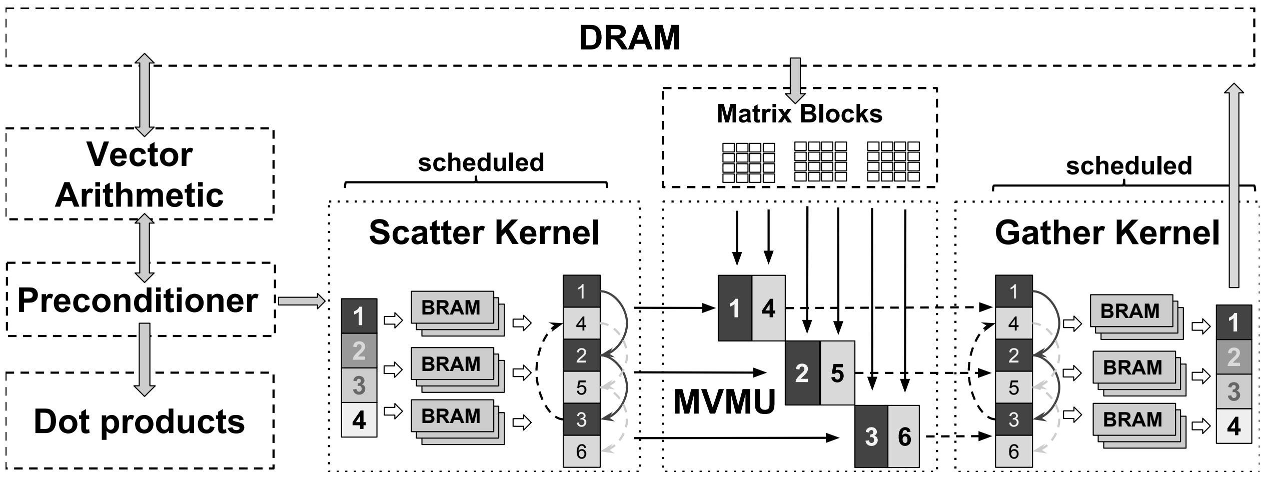

Fig. 4. Overview of the proposed dataflow CGM architecture.

IV. Architecture

In this section we describe the prototype hardware architecture of a preconditioned Conjugate Gradient Method, designed as a plug-in accelerator to the spectral/hp FEM code running on a CPU node. As a prototype, it uses a diagonal preconditioner and executes on a single FPGA: in this paper our primary objective is to evaluate the assembly strategy proposed in the previous section.

The proposed hardware architecture performs several iterations of the loop in Algorithm 1 directly on FPGA, without a control from the CPU host or sending data for the halo exchange to the CPU. Instead, a state machine on FPGA keeps track of the current phase of execution and checks the stopping criteria. In our experience, it is important to perform more than one iteration without returning control to the CPU host to minimise the impact of both static and dynamic execution overheads. For this architecture we employ a local matrix evaluation strategy and represent CGM vectors in global coefficient space, as presented in the Section III. As a result, we implement a block-diagonal dense matrix-vector multiply, at the expense of on-chip assembly done at the vector level. We assume all necessary data structures such as matrix storage, assembly mappings and other auxiliary arrays are partitioned, reordered and packed to the expected bit widths by a host CPU code.

The proposed architecture comprises of a number of processing elements (PEs), arranged into a pipeline, which processes input matrix and vectors with no stalls as long as their processing rates are well-balanced. Figure 4 presents the scheme of the design workflow. The Vector Arithmetic, (diagonal) Preconditioner and Dot Product Kernels are trivial compute kernels, thus we omit their implementation details. Matrix-vector multiplier unit (MVMU) is used to perform the large scale block diagonal dense matrix-vector multiplication. Each block is a dense matrix, stored column-wise, one after another. The workload of MVMU is evenly split among multiple Matrix Processing Elements (MPEs) as shown on Figure 4 Each MPE multiplies one matrix block with the corresponding elements of the vector. The MPE processes a block in a column major order. In our implementation, MPE supports matrix blocks of size varying from 20×20 and 100×100. The architecture maintains a number of DRAM streams for the matrix, preconditioner and vector data, as well as two streams representing the scheduling, generated on the CPU. Most of CGM vectors are read/written from/to the DRAM at the same time by Vector Arithmetic Kernel and thus coalesced into two input/output streams. The output of the Preconditioner Kernel is also forwarded to the Scatter Kernel; the Scatter Kernel, Matrix Vector Multiply Unit (MVMU) and the Gather Kernel form the pipeline, mathematically equivalent to the sparse matrix-vector multiply $M_g\, v_g$. The schedulings are split into `global_dof_schedule` and `local_dof_schedule` since these two streams are being read at different rates. Each of the two latter streams coalesce gather and scatter schedulings on the bit levels to save on the total number of DRAM streams.

Scatter Kernel implements the assembly strategy presented in the Section III. It consists of *1)* an array of BRAMs, each with one read and one write port, *2)* two meshes of comparators and muxes to select which BRAM buffer(s) to write to and read the entries of an input vector from, *3)* a tree of comparators and muxes to select the values read from BRAMs, chosen by the scheduling to form the output, *4)* a floating point negator, to support the sign of a local to global mapping (see Figure 3). Since all BRAMs should have been written each cycle, first mesh or comparators and muxes ensures the correct buffers (chosen by the `global_dof_schedule` stream) are written at the appropriate addresses, and sets the write disable flags to all other BRAMs. Similarly, the other mesh of comparators and muxes ensures the correct BRAMs are read at the correct addresses, chosen by the `local_dof_schedule` stream, while the other BRAMs are read at random. A selection tree is used to choose only scheduled reads from an array of BRAMs, to form the PE's output. Finally, one bit of stream determines whether the PE's output requires a negation.

Gather Kernel has similar structure to the Scatter Kernel. Inputs from the matrix-vector multiplier are scheduled to BRAMs via `local_dof_schedule`: the appropriate BRAMs are read at scheduled addresses using one comparator mesh, and a comparator tree is used to select the correct read results. The value read from the BRAM represents a partial sum of some previous elements of the input stream, mapped to the same global degree of freedom and accumulated together. New input from the MVMU is added to this partial sum and the result is written to the same location of the same BRAM as before, which requires an adder and the second comparator mesh. Then, `global_dof_schedule` determines when the appropriate degree of freedom is completely assembled and should form the PE's output. This global degree of freedom needs to be selected from the array of values read from BRAMs as part of accumulation loop, for which another comparator mesh is used. Finally, this PE provides an adder tree to reduce partial sums of a global coefficient, in case they are stored in separate BRAM locations.

V. Evaluation

In this section we compare our initial FPGA implementation with a reference, highly optimised CPU implementation to study the applicability of our approach to realistic problem sizes.

The CPU Reference is the incompressible Navier Stokes (INS) Solver from the Nektar++ spectral/hp framework [11] written in C++ and parallelised with MPI. We compile Nektar++ using ICC 12.1.4 with `-O3` flag, OpenMPI (1.6.5) and MKL (10.3 Update 10) support, and run it on a server with 2 Xeon E5-2640 CPUs and 64GB of RAM, the same server we use for benchmarking FPGA prototype. INS is run with a single level static condensation linear solver, which, in turn, uses preconditioned CGM for the boundary problem solve. The CGM solver is run with diagonal preconditioner and local matrix evaluation strategy. The INS solver makes 4 linear solves per time step; we provide the run times for the pressure solve, the most time consuming linear solve per time step, at a fixed time step $T_{\text{num}} > 2$ to allow for initialisation and warm-up. Our two benchmark 3D meshes are presented in the Table I. We also compare them to the setting used in the previous work [5]: we stress it is substantially smaller then both our meshes. Normally, our benchmark meshes are run at higher polynomial orders on high-end compute clusters. Accordingly, the meshes of this size should be benchmarked on a cluster of FPGA accelerators, thus reducing data set size per FPGA. However, since for this work we focus on the performance of a single compute node with and without FPGA accelerator, we reduce polynomial order P from 6–7 down to 3 in order to fit the Nektar++ PDE solver to available DRAM budget.

Figure 5 presents the scalability of the CPU implementation with number of MPI ranks for the Naca One Layer mesh. The CPU performance profile on Figure 5 is significantly impacted by the MPI communication overheads, therefore linear scaling with the number of threads is not likely. The performance scaling of DBTV mesh is inconclusive since it takes more time to run as the number of MPI threads increase. A detailed analysis show that the run time of a CGM solver increases with number of MPI threads due to increased communication volumes. We conclude we need to increase the polynomial order for this problem to shift the compute/communication

TABLE I. Characteristics of mesh problems

Mesh name	DBTV [12]	Naca 1L [13]	Past work [5]
Mesh type	unstructured	unstructured	structured
P used in this paper and [5]	3	3	1
P normally being used	7	6	
Num. tetrahedra, block size	34887, 20x20	58728, 20x20	up to 48000, 4x4
Num. prisms, block size	0	11549, 38x38	0
Num. local dofs	697740	2758986	192000
Num. global dofs	158905	1077373	?
Fl. point precision	double	double	single
Dense matrix size	106.7MB	820MB	up to 2.92MB
Once CGM vector size	1.21MB	8.21MB	?

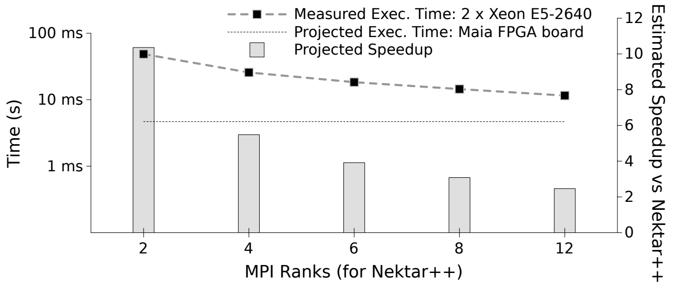

Fig. 5. Strong scaling of performance of Nektar++ run on Naca One Layer mesh as function of number of MPI ranks

balance and thus yield a quality CPU benchmark: this problem, in a sense, is *too small* at $P = 3$.

Scheduling Our scheduling heuristic on the CPU determines the number of BRAM buffers and their sizes necessary for supporting on-chip assembly for both test meshes. These numbers become compile time parameters to our hardware architecture. The results are projected in the Table II for the 3-pipe architecture. The estimate to the number of BRAMs are based on the BRAM configurations available to the Altera Stratix V D8 chip in our Maxeler board (see below). Additionally, for Naca model we re-run scheduling algorithm for the 40-bit buffer entries. For this case, the scheduling requires nearly twice less BRAMs as for the 64-bit case. Note scheduling procedure produces *the upper bound* to the scheduling complexity the appropriate design can support. Once the design with a given scheduling parameters is built in hardware, it may support other meshes with less demanding scheduling.

TABLE II. Scheduling parameters of the hardware architecture, per scatter/gather kernel

	DBTV mesh	Naca One Layer	
Bit width of buffer entries	**64 bit**	**64 bit**	**40 bit**
Number of BRAM buffers	48	90	96
Size of a buffer	2048	1280	2048
Estimated number of BRAMs	336	630	384
Num iterations to converge	1	4411	2625

FPGA Implementation We implemented the proposed architecture using the MaxCompiler 2014.1 on a Maxeler Max4 Maia acceleration board with an Altera Stratix V D8 chip, 48 GB DRAM and Infiniband connection to the CPU server.

We built our designs for 150Mhz stream frequency and 666 Mhz memory controller frequency, with 3 MPEs in MVMU, 3 pipes for the rest of the design; each MPE performs 24 floating point multiplies per cycle. These choices avoid excessive FIFO usage by MaxCompiler: at 800Mhz the usage of the memory controller and additional FIFOs is almost double that of the resource usage of the MVMU. Also, the DRAM throughput available to application kernels at memory frequency 666 Mhz (2791 bits made available per cycle) already exceeds required threshold for our design: the total effective rate of all streams for the DBTV mesh is 2566.32 bits per cycle and

TABLE III. Does not change with problem size: area utilisation for the MVMU

Kernel	BRAMs	LUTs	FFs	DSPs
Stratix V DDR	149	46327	52094	
Memory Controller (MVMU)	652	17165	64581	
PCIe (MVMU)	100	6713	7828	
FIFOs (MVMU)	184	568	709	
Vector arithmetic	127	35886	43290	36
MVMU (3 x MPEs)	201	105129	145147	288

TABLE IV. VARIES WITH PROBLEM SIZE: ESTIMATED AND ACTUAL AREA UTILISATION FOR GATHER/SCATTER KERNELS

Resource Usage	DBTV	Naca One Layer
Scatter Kernel		
BRAM (Est/Act)	336 / 341	630 / 638
LUT(Act)	4191	8197
FF (Act)	33720	61294
Gather Kernel		
BRAM (Est/Act)	342 / 358	636 / 652
LUT (Act)	14079	21537
FF (Act)	57674	101856
Total (including MVMU)		
BRAM (Est/Act)	2194 / 2215	**2782 / 2806**
LUT(Act)	224K	237K
FF (Act)	397K	467K
Available	2567 BRAMs, 524K LUTs, 1024K FFs, 1963 DSPs	

2753.84 bits per cycle for the Naca One Layer. Thus, the design is compute bound. We validate this by building the MVMU part of our design with 666 Mhz and 800 Mhz DRAM rates and benchmarking it on a local matrix of a Naca One Layer mesh: the run times are almost identical, up to a measurement error. We are also building our design for both 64-bit scheduling parameter sets suggested by the scheduling runs on CPU. The results are presented in Table III and Table IV. We conclude the design supporting the scheduling for DBTV mesh fits the chip, while the hardware build supporting the Naca One Layer mesh is just 239 BRAMs over-fit. Note our estimates to the BRAM utilisation based on the scheduling results are very accurate, thus providing the tool for predicting the scalability for a given user mesh to available FPGA. Also note that even with a small number of optimisations, such as using a hard memory controller — which is already available in current generation FPGAs, our design could even fit on one chip. Additionally the proposed design could benefit greatly from more on-chip BRAMs or higher flexibility of BRAM configurations (e.g. M20K does not support 64-bit configuration). Given that our scheduling guarantees each pipe in the Scatter Unit to produce an output and each pipe in the Gather Unit to process an input each cycle, the processing rates between our kernels are balanced and thus design yields no stalls. Therefore, the raw matrix multiplier performance can be used to estimate the final run time, per iteration, of the proposed design. Figure 5 shows that the proposed approach can achieve a speedup of up to 3.92 times when compared with the highly optimised Nektar++ implementation running on 6 threads and up to 2.46 times compared with 12 threads (running on two physical CPUs).

VI. CONCLUSION

We have proposed a high performance FPGA design for the Conjugate Gradient Solver specialised to FEM problems, as well as the novel approach to FEM assembly done directly on chip. It is important to note that the architecture supports an arbitrary sparsity structures. The problem sizes considered in this paper are also substantially larger than in previous FPGA works. We note that the implementation is impacted by constraints imposed by the available FPGA technology despite of the high flexibility of the original architecture. Our experiments show there is a good potential in accelerating Finite Element Methods on FPGA based on an accurate choice of the problem size to fit available FPGA technology. A number of improvements are left for future work: exploiting symmetric structure of FEM matrix, advanced preconditioners, multi-FPGA design and reduced precision.

ACKNOWLEDGMENT

This work was supported in part by Altera, the Maxeler University Programme, the UK EPSRC grants EP/I012036/1 and EP/L00058X/1, the European Union Seventh Framework Programme under grant agreement number 318521, and the HiPEAC NoE.

REFERENCES

[1] C. Cantwell *et al.*, "High-order spectral/hp element discretisation for reaction-diffusion problems on surfaces: Application to cardiac electrophysiology," *Journal Of Computational Physics*, vol. 257, pp. 813–829, 2014.

[2] G. Karniadakis and S. Sherwin, *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2013.

[3] Y. Elkurdi *et al.*, "FPGA architecture and implementation of sparse matrix–vector multiplication for the finite element method," *Computer Physics Communications*, vol. 178, no. 8, pp. 558–570, 2008.

[4] M. van der Veen, "Sparse matrix vector multiplication on a field programmable gate array," Master's thesis, University of Twente, 2007.

[5] J. Hu, S. F. Quigley, and A. Chan, "An element-by-element preconditioned conjugate gradient solver of 3d tetrahedral finite elements on an fpga coprocessor," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 2008, pp. 575–578.

[6] J. W. Demmel, M. T. Heath, and H. A. van der Vorst, "Parallel numerical linear algebra," *Acta Numerica*, vol. 2, pp. 111–197, 1 1993.

[7] P. E. J. Vos, S. J. Sherwin, and R. M. Kirby, "From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations," *J. Comput. Phys.*, vol. 229, no. 13, pp. 5161–5181, Jul. 2010.

[8] G. Markall *et al.*, "Finite element assembly strategies on multi-core and many-core architectures," *International Journal for Numerical Methods in Fluids*, vol. 71, no. 1, pp. 80–97, 2013.

[9] G. Wu *et al.*, "High-performance architecture for the conjugate gradient solver on fpgas," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 60, no. 11, pp. 791–795, 2013.

[10] G. C. Chow *et al.*, "An efficient sparse conjugate gradient solver using a beneš permutation network," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE, 2014, pp. 1–7.

[11] C. Cantwell *et al.*, "Nektar++: An open-source spectral/hp element framework," *Computer Physics Communications*, 2015.

[12] G. Rocco, "Advanced instability methods using spectral/hp discretisations and their applications to complex geometries," Ph.D. dissertation, Imperial College London, 2014.

[13] J. S. Chow, G. G. Zilliac, and P. Bradshaw, "Mean and turbulence measurements in the near field of a wingtip vortex," *AIAA journal*, vol. 35, no. 10, pp. 1561–1567, 1997.