Hardware Acceleration of High-Performance Computational Flow Dynamics Using High-Bandwidth Memory-Enabled Field-Programmable Gate Arrays

Hogervorst, Tom; Nane, Razvan ; Marchiori, Giacomo; Qiu, Tong Dong ; Blatt, Markus; Rustad, Alf Birger

DOI
10.1145/3476229

Publication date
2022

Document Version
Final published version

Published in
ACM Transactions on Reconfigurable Technology and Systems

Hardware Acceleration of High-Performance Computational Flow Dynamics Using High-Bandwidth Memory-Enabled Field-Programmable Gate Arrays

TOM HOGERVORST and RĂZVAN NANE, Delft University of Technology, The Netherlands
GIACOMO MARCHIORI and TONG DONG QIU, Big Data Accelerate B.V., The Netherlands
MARKUS BLATT, OPM-OS AS, Norway
ALF BIRGER RUSTAD, Equinor S.A., Norway

Scientific computing is at the core of many High-Performance Computing applications, including computational flow dynamics. Because of the utmost importance to simulate increasingly larger computational models, hardware acceleration is receiving increased attention due to its potential to maximize the performance of scientific computing. Field-Programmable Gate Arrays could accelerate scientific computing because of the possibility to fully customize the memory hierarchy important in irregular applications such as iterative linear solvers. In this article, we study the potential of using Field-Programmable Gate Arrays in High-Performance Computing because of the rapid advances in reconfigurable hardware, such as the increase in on-chip memory size, increasing number of logic cells, and the integration of High-Bandwidth Memories on board. To perform this study, we propose a novel Sparse Matrix-Vector multiplication unit and an ILU0 preconditioner tightly integrated with a BiCGStab solver kernel. We integrate the developed preconditioned iterative solver in *Flow* from the Open Porous Media project, a state-of-the-art open source reservoir simulator. Finally, we perform a thorough evaluation of the FPGA solver kernel in both stand-alone mode and integrated in the reservoir simulator, using the NORNE field, a real-world case reservoir model using a grid with more than 10^5 cells and using three unknowns per cell.

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

Additional Key Words and Phrases: Iterative solvers, FPGA, GPU, ILU0, BiCGStab, CFD, HPC

ACM Reference format:
Tom Hogervorst, Răzvan Nane, Giacomo Marchiori, Tong Dong Qiu, Markus Blatt, and Alf Birger Rustad. 2021. Hardware Acceleration of High-Performance Computational Flow Dynamics Using High-Bandwidth Memory-Enabled Field-Programmable Gate Arrays. *ACM Trans. Reconfigurable Technol. Syst.* 15, 2, Article 20 (December 2021), 35 pages.
https://doi.org/10.1145/3476229

Authors' addresses: T. Hogervorst and R. Nane, Delft University of Technology, Mekelweg 4, 2628CD, Delft, The Netherlands; emails: tom.hogervorst74@gmail.com, razvan.nane@bigdataccelerate.com; G. Marchiori and T. D. Qiu, Big Data Accelerate B.V., Burgemeestersrand 66, 2625NW, Delft, The Netherlands; emails: marchiori.giacomo@gmail.com, tongdong.qiu@bigdataccelerate.com; M. Blatt, OPM-OS AS, Heyerdahlsvei 12b, 0777 Oslo, Norway; email: markus@dr-blatt.de; A. B. Rustad, Equinor S.A., Arkitekt Ebbels veg 10, Rotvoll, Norway; email: abir@equinor.com.

1936-7406/2021/12-ART20 $15.00
https://doi.org/10.1145/3476229

1 INTRODUCTION

The advent of data proliferation and the slowing down of Moore's law is driving many application designers to implement an algorithm's computationally demanding functionality using a hardware accelerator. Although **Graphical Processing Units (GPUs)** benefit from a larger user base because of their massive amount of parallel cores, on-board **High-Bandwidth Memories (HBMs)**, and ease of programming, **Field-Programmable Gate Arrays (FPGAs)** recently have been gaining a lot of attention because of their application-specific customizability. Furthermore, the recent integration of **High-Bandwidth Memory**s (**HBM**s) in FPGA-based systems have touted the **Field-Programmable Gate Array** (**FPGA**) as a viable competitor for the **Graphical Processing Unit** (**GPU**) in emerging machine learning-related domains, but also in **High-Performance Computing (HPC)** applications where the **GPU** was traditionally the sole candidate. The acquisition of Altera by Intel [13] and the recent announcement of AMD acquiring Xilinx [2], two of the biggest **FPGA** providers, can be considered an indication of this trend.

In this context, it is highly relevant to reassess the potential of **FPGA**s for **High-Performance Computing** (**HPC**), when we consider the addition of **HBM** on board, the increase of on-chip memory (e.g., Ultra RAM (URAM) [34]), and the continuing increase of logic cell resources enabling one to implement complex functionality. To perform this evaluation, we look at an ubiquitous scientific computing field used in many **HPC** application domains, namely the solution of systems of non-linear partial differential equations in time and space. It is typically not possible to solve these systems exactly through analytical methods. Hence, the equations are discretized in time and over a domain space using finite difference, finite volume, or finite element methods. The derived system of equations is solved, often iteratively using a combination of preconditioners and iterative linear solvers available in one of the many scientific solver libraries, such as DUNE [4] and PETSc [11]. Due to the complexity of implementing linear solvers on **FPGA**, there are only a few previous works that studied the acceleration of them on **FPGA**. Those works often made assumptions to simplify the design and implementation, such as either focusing solely on single-precision floating point support, or using simple solvers without preconditioners, and all ignored the integration into the whole scientific computing pipeline. However, to realistically assess the potential of **FPGA**s, it is important to eliminate the aforementioned restrictions. Consequently, in this work, we develop a complete solver with a preconditioner on **FPGA** with support for double precision, which is utterly important in a real-world setting because it enables the convergence to a solution. Furthermore, we integrate our work in a state-of-the-art reservoir simulator, *OPM Flow* [25], and validate our work by running with a real-world use case containing a large number of grid cells (10^6).

To accomplish our study goals, we develop a novel BiCGStab [31] solver implementation with ILU0 [5] as preconditioner on **FPGA**. This combination of preconditioner and linear solver was chosen because it is known be a good solver for the large sparse linear systems arising when simulating oil reservoirs [35]. Due to the challenging properties of the linear systems, it is mandatory to use powerful Krylov methods like BiCGStab together with good preconditioners and high precision. Often oil reservoir simulators use even more powerful preconditioners like nested factorization [3, 28]. Otherwise, the iterative solver will not converge due to highly heterogeneous material parameters, which lead to rather ill-conditioned matrices. This choice of solver also leads to an apples-to-apples comparison with the CPU performance on the NORNE field dataset, as this combination was best performing in that case. To maximize the performance, we use manually written **Register-Transfer Level (RTL)**. Please note that there might be other solvers that can be more efficient in terms of time needed per iteration of the linear solver on a parallel architecture such as the **FPGA** (and **GPU**) than an ILU0 preconditioner, but these still might not be suitable to reach convergence or even scale in terms of time needed to reach the solution. However, this is left for future work because it was most important to make certain the results are close to the ones obtained by the already validated software-only simulation. Therefore, we selected the same combination of preconditioner and solver as those used by default in *OPM Flow* and designed the solver to support double precision because high accuracy is required, as otherwise the selected real-world case would not converge. Furthermore, integration is key to benchmark the performance in the overall system. Previous work [6, 16, 32] has only benchmarked the solvers in isolation, whereas we consider the whole system for benchmarking, where including the memory transfers between off-chip and off-board as well as the pre- and post-processing times is crucial. Finally, we make our work available in the **Open Porous Media** (**OPM**) Git repository [19].

The contributions of this work are summarized as follows:

- We propose a novel **Compressed Sparse Row (CSR)**-based encoding to optimize **Sparse Matrix-Vector (SpMV)** multiplication on **FPGA**, which is the core functionality required to implement iterative solvers.
- We develop a hardware model to predict and guide the design of the solver, helping us to understand trade-off between area and performance when scaling the resources.
- We design a novel ILU0-BiCGStab preconditioned solver on **HBM**-enabled **FPGA** using optimized hand-coded **Register-Transfer Level** (**RTL**).
- We integrate the proposed solver in *OPM Flow*, both for **FPGA** and **GPU**. Furthermore, we make all sources available in the **OPM** Git repository [19].
- We provide extensive evaluation results for both stand-alone solver kernel benchmarking and complete reservoir simulation execution on a real-world use case running on three different types of systems: CPU, **FPGA**, and **GPU**.

The rest of the article is organized as follows. In Section 2, we introduce the background required to understand the work, whereas in Section 3, we highlight related works. We then start a bottom-up explanation approach in which we first describe the **Sparse Matrix Vector Multiplication** (**SpMV**) kernel in Section 4, then the solver in Section 5, and discuss the performance hardware model used to guide the design in Section 6. Finally, in Section 7, we present the experimental results.

2 BACKGROUND

2.1 Sparse Matrix Iterative Solvers

The central problem in this work is to solve $A * \vec{x} = \vec{b}$ for vector $\vec{x}$, where A is a known sparse matrix and $\vec{b}$ is a known dense vector. There are different methods for solving such a problem, which can be grouped into direct solvers and iterative solvers. Direct solvers, as the name implies, use linear algebra techniques to solve the problem for x directly and precisely. However, direct methods are not suitable to solve large sparse linear systems, as is usually the case in a real-world **HPC** application, because they will fill in the zero values of the sparse matrix to solve the problem, resulting in a very high computational cost and memory usage. Iterative solvers are a heuristic alternative to direct solvers: they start from an initial guess of the solution and then iteratively improve this guess, until it is deemed close enough to the actual solution. In this article, we use the BiCGStab (bi-conjugate gradient stabilized) solver with the ILU0 (incomplete LU factorization with no fill-in) as the preconditioner because this combination is the most robust and the default software configuration used in *OPM Flow*. Consequently, besides the goal of studying the acceleration potential of FPGAs, another implicit goal required by the reservoir engineers using the simulator was to obtain simulation results close to the software counterpart. Consequently, although it might not

ALGORITHM 1: Pseudo-code BiCGSTAB Solver

```
Input: $\vec{b}, \vec{x}, A$
▷ $\vec{b}$ and $\vec{x}$ are $N$ element arrays, A is an NxN matrix
Output: $\vec{r}, \vec{x}$
▷ Both are $N$ element arrays

$\vec{r} = \vec{b} - spmv(A, \vec{x})$
$\vec{rt} = \vec{r}$
$\rho = 0$
$\vec{p} = \vec{0}$
$norm = sqrt(dot(\vec{r}, \vec{r}))$
$conv\_threshold = norm/desired\_improvement$
while $norm > conv\_treshhold$ do
    $\rho_{new} = dot(rt, r)$
    $\beta = (\rho * \alpha)/(\rho_{new} * \omega)$
    $\vec{p} = axpy(\beta, \vec{r}, axpy(\omega, \vec{p}, \vec{v}))$
    $\vec{y} = preconditioner(\vec{p})$
    $\vec{v} = spmv(A, \vec{y})$
    $\alpha = \rho_{new}/dot(\vec{rt}, \vec{v})$
    $\rho = \rho_{new}$
    $\vec{x} = axpy(\alpha, \vec{x}, \vec{y})$
    $\vec{r} = axpy(-\alpha, \vec{r}, \vec{v})$
    $norm = sqrt(dot(\vec{r}, \vec{r}))$
    if $norm \leq conv\_threshold$ then
        $break$
    end if
    $\vec{y} = preconditioner(\vec{r})$
    $\vec{t} = spmv(A, \vec{y})$
    $\omega = dot(\vec{r}, \vec{t})/dot(\vec{t}, \vec{t})$
    $\vec{x} = axpy(\omega, \vec{x}, \vec{y})$
    $\vec{r} = axpy(-\omega, \vec{r}, \vec{t})$
    $norm = sqrt(dot(\vec{r}, \vec{r}))$
end while
```

be the best choice for acceleration on a parallel architecture, fulfilling the conformance criteria led us to use the same solver used in the software stack.

Algorithm 1 shows the pseudo-code of the BiCGStab solver. This code makes use of several other functions:

- An **SpMV** function, which performs an SpMV multiplication.
- A dot function, which calculates the dot product (or inner product) of two vectors.
- An axpy function, which scales its first input vector by a scalar value and adds it to its second input vector (i.e., $axpy(\alpha, \vec{x}, \vec{y})$ performs: $\alpha * \vec{x} + \vec{y}$).
- A preconditioner function. This is an optional part of the solver and does not have a set behavior. For more information about preconditioners, see Section 2.3.

The result of the solver are two vectors: $\vec{x}$, which is the estimation of the solution of $A*\overrightarrow{x_{actual}} = \vec{b}$, and $\vec{r} = \vec{b} - A\vec{x}$, a residual vector, which is an indication of how close the result $\vec{x}$ vector is to the exact solution $\vec{x_{actual}}$. It is worth noting in Algorithm 1 how the exit condition of the solver is set, or, in other words, how the algorithm determines when to stop iterating because the results are sufficiently close to the actual result. The solver stops iterating when the norm of the residual is lower than a certain *convergence* threshold. There are two ways in which this convergence threshold can be chosen: either it is a constant value, in which case we speak of an absolute exit condition, or it is a value set by the norm of the initial residual by a *desired improvement* factor, in which case we speak of a relative exit condition. An absolute exit condition can be useful when one needs to be certain that the result of a solver is at least within a certain numerical distance from the actual result. However, when solving for multiple different matrices from different domains, a matrix with lower values all across the board might start off at a lower error and also reach the absolute exit condition more quickly. In such a case, a relative exit condition leads to a more fair comparisons between solves, which is why a relative exit condition is used in this work.

Ignoring for now the preconditioner, what remains in the BiCGStab solver are a series of vector operations and **SpMV**s, as well as a few scalar operations. From profiling the solver, it is clear that the majority of its computational time is spent on performing **SpMV** and SpMV-like computations. Therefore, the first goal to design a performant solver kernel would be to perform the **SpMV** efficiently.

2.2 SpMV Multiplication

The **SpMV** is a widely used, rather elementary operation in which a sparse matrix is multiplied by a dense vector. How this operation will be performed algorithmically depends on how the matrix is stored. A sparse matrix consists of many more zeroes than non-zero values, which means that storing all of them is not efficient. There are a variety of different formats in which only the non-zero values of a sparse matrix can be stored. Among these formats, **Compressed Sparse Column (CSC)** and CSR are two of the most commonly used and have inspired many variants (e.g., [23, 24]). The **Compressed Sparse Row** (**CSR**) and CSC formats both store matrices efficiently without putting any restrictions on the contents of the matrix, whereas other formats may require the matrix to have certain sparsity pattern features such as symmetry, values being repeated multiple times, or store additional non-zeroes, such as sliced ELLPACK [15], to be efficient. In this article, we focus only on the **CSR** representation, as the column-major CSC format would be more difficult to use during the application of the ILU0 preconditioner because ILU0 goes through the matrix row by row (see Section 2.3), which would be difficult to implement in the column-major CSC format.

The **CSR** format consists of three arrays: an array with all non-zero values of the matrix stored row-major, an array of the column indices of those non-zero values, and an array of row pointers, which contains an index for each row to identify which non-zero value is the first value of that row, plus a last additional entry containing the total number of non-zeroes stored. The pseudo-code showing how the SpMV is performed on a matrix stored in the CSR format is given in Algorithm 2. It is worth noting that there is a high degree of parallelism available in this SpMV. In theory, the calculations for all rows can be done in parallel because the result of one row is never used during the calculation of another row. However, in practice, the irregular memory accesses in the x vector, which depend on the random column index representing the location of the non-zero value in the system matrix, restrict the amount of parallelism that can be exploited efficiently. The main bottleneck is the requirement to replicate the x vector to enable multiple read units to feed independent row **Processing Units (PUs)**. This replication is impractical for realistic large use cases for two main reasons: (1) limited on-chip (i.e., cache) memory and (2) significant pre-processing computations required to align the read data feed into the multiple PUs, which is needed to exploit the maximum available bandwidth in state-of-the-art accelerators with **HBM**.

ALGORITHM 2: Pseudo-code of a CSR SpMV

```
Input: $nnzs$, $colInds$, $rowPointers$, $x$
▹ $nnz$ is an $M$ element array of floating point values,
▹ $colInds$ is an $M$ element array of integers,
▹ $rowPointers$ is an $N + 1$ element array of integers,
▹ $x$ is an $N$ element array of floating point values
Output: $y$
▹ $y$ is an $N$ element array of floating point values

for $0 \leq i < N$ do
    $y[i] = 0$
    for $rowPointers[i] \leq v < rowPointers[i + 1]$ do
        $y[i] = y[i] + nnzs[v] * x[colInds[v]]$
    end for
end for
```

2.3 The ILU0 Preconditioner

How hard a series of linear equations is to solve depends on various different aspects of those equations, or the matrix that represents them. In other words, how many non-zeroes the matrix has, where they are located in the matrix, and their values in relation to one another, among other things, all play a role. The *condition number* $\kappa(A)$ of a matrix A, which is the ratio of the biggest eigenvalue of the matrix divided by the smallest one, is a measure of how relative changes in the right-hand side b of a linear equation $A * \vec{x} = \vec{b}$ result in relative changes of the solution $\vec{x}$. In addition, it is a measure of how changes in A affect the solution and how numerical errors affect the solution. Often, convergence of iterative methods is worse for matrices with high condition numbers. Therefore, although iterative solvers such as BiCGStab are a powerful method for solving sparse matrix problems, they might still take many iterations before they approach the result within the desired precision when solving matrices with a high condition number.

One technique for reducing the amount of iterations needed for BiCGStab and other Krylov methods is utilizing a preconditioner. Generally, a preconditioner is a transformation or matrix M that approximates A, but for which $M * \vec{x} = \vec{b}$ is less computationally expensive to solve than $A * \vec{x} = \vec{b}$. If M is a matrix, then the preconditioned method solves $M^{-1}A * \vec{x} = M^{-1} * \vec{b}$. Note that if M is a good approximation for A, then this system will have a much better condition number than the original one and hence the iterative solver will converge faster. A simple example of a preconditioner is the Jacobi method with $M = diag(A)$, the diagonal values of A. Many different types of preconditioners exist, from very simple ones that only very weakly approximate the problem's solution to complex ones that almost solve the system by themselves. Unfortunately, how well a preconditioner performs depends not only on the type of preconditioner but also on the matrix being solved. For example, the Jacobi preconditioner is fairly accurate if the original matrix has most of its non-zeroes clustered around the diagonal, and if the values on the diagonal have a higher magnitude than those off the diagonal, but does not work as well for matrices that do not have those characteristics.

In this work, we utilize the zero fill-in incomplete LU factorization (ILU0) as the preconditioner, which falls somewhere in the middle: it approximates the system reasonably while not being too computationally intensive to apply. Using an ILU0 consists of two steps. First, before the solver is started, an incomplete LU decomposition is performed. To understand what this is, we must first introduce the LU decomposition. LU decomposition is a direct solving method that calculates an

Table 1. CPU Solver Time in Milliseconds with 10^{-2} & 10^{-6} Exit Condition

Preconditioner	None	None	Jacobi	Jacobi	ILU0	ILU0
Matrix	10^{-2}	10^{-6}	10^{-2}	10^{-6}	10^{-2}	10^{-6}
Hummocky	5.9	925	**1.2**	125.8	3.1	**32.3**
bcsstk25	5.8	3396	**3.7**	**34.9**	18.9	4331
bodyy6	4.4	224	**1.2**	86.3	3.7	**15.7**
wathen120	8.0	112	**5.2**	25.2	12.8	**18.7**
gridgena	**2.0**	403	5.9	643.3	18.7	**324.7**
qa8fm	**12.7**	114	13.1	79.7	44.8	**60.3**

upper-triangular matrix U and a lower-triangular matrix L so that $A = L*U$. Such a decomposition can be done using Gaussian elimination, among other methods. Once such a decomposition has been found, $A*\vec{x} = \vec{b}$ can be solved for $\vec{x}$ by first solving $L*\vec{y} = \vec{b}$ for $\vec{y}$ and then solving $U*\vec{x} = \vec{y}$ for $\vec{x}$. Since L and U are triangular matrices, these two solves can be done using forward and backward substitution, respectively. However, like we discussed in the previous section, direct solving methods become very computationally intensive for sparse matrices because they start filling in the zero values of the matrix, and LU decomposition is no exception. Incomplete LU(N) decomposition decreases the high computational cost of LU decomposition for sparse matrices by reducing the amount of fill-in that happens, where N is an indicator of how much fill-in is done. The case were $N = 0$ (ILU0) means that for the L and U matrices, the decomposition generates non-zero values only at the same indices where A had non-zero values. Since the result of the decomposition algorithm, L and U, is not unique, it is common practice to restrict the L matrix to be a unit triangular matrix. With the diagonal of the L matrix set to known constant values, it is possible to store both the L and U matrices resulting from ILU0 decomposition in a matrix with the same sparsity pattern as the decomposed matrix. The diagonal of this LU result matrix belongs to the U matrix, and the assumption that the L matrix has a diagonal filled with ones needs to be taken into account by the operations performed using this matrix.

Once the ILU0 decomposition is complete, the solver can start. The solver will then apply the ILU0 preconditioner in every iteration. In this apply_ILU0 step, the solution to the matrix solve is approximated by solving for the LU matrix. Pseudo-code of the ILU0 application is shown in Algorithm 3. The ILU0 application consists of two parts, represented by the two for loops in lines 1 and 11: a forward substitution on the lower-triangular part of the LU matrix, and a backward substitution on the upper-triangular part of that matrix. Both loops are very similar in the operations they perform, with the main differences being that the forward substitution traverses the matrix top to bottom, whereas the backward substitution traverses the matrix bottom to top, and that the backward substitution concludes by dividing the result array by the values on the diagonal of the LU matrix. It is worth noting that although these loops ostensibly have the same basic structure as the SpMV loop, there is one major difference: the results of one row depend on rows before it, which means the operations on the rows cannot be performed all in parallel. See Section 2.4 for more on how this issue can be circumvented somewhat by reordering the matrix.

To quantify the effect of the ILU0 preconditioner in practice for our solver performance, we ran software implementations of the solver without a preconditioner and with the Jacobi and ILU0 preconditioners on several benchmark matrices. The matrices were obtained from the SuiteSparse matrix collection [7].

Table 1 shows the results of these tests, which were obtained by running the preconditioned solver for a single thread on an Intel i7-9700 CPU with 16 GB of RAM. Two exit conditions were used: the more precise exit condition requiring the absolute value of the residue being reduced by

ALGORITHM 3: Pseudo-code of the applying of the ILU0 preconditioner

```
Input: LUmat, $x$
▹ LUmat is an NxN sparse matrix in the CSR format
▹ $x$ is an $N$ element array of floating point values
Output: $p$
▹ $p$ is an $N$ element array of floating point values
for $0 \leq i < N$ do
    $p[i] = x[i]$
    for $\text{LURowPointers}[i] \leq v < \text{LURowPointers}[i+1]$ do
        $pIndex = \text{LUcolInds}[v]$
        if $pIndex \geq i$ then
            $break$
        end if
        $p[i] = p[i] - \text{LUnnzs}[v] * p[pIndex]$
    end for
end for
for $N > i \geq 0$ do
    $diagIndex = \text{LURowPointers}[i+1]$
    for $\text{LURowPointers}[i+1] > v \geq \text{LURowPointers}[i]$ do
        $pIndex = \text{LUcolInds}[v]$
        if $pIndex \leq i$ then
            $diagIndex = v$
            $break$
        end if
        $p[i] = p[i] - \text{LUnnzs}[v] * p[pIndex]$
    end for
    $p[i] = p[i] / \text{LUnnzs}[diagIndex]$
end for
```

10^{-6} of its initial value, and the less precise exit condition requiring a residue norm reduction of 10^{-2}. Not shown in the table are the number of iterations in which the solvers converged, which in all matrices except for *bcsstk25* were lower when ILU0 preconditioning was used over the other two preconditioning options. However, a lower iteration count does not directly result in a higher solver performance. The first reason for this is the time the pre-processing of the preconditioner takes, which accounts for between 60% and 90% of the time taken by the lower-accuracy solver with the ILU0 preconditioner. The Jacobi preconditioner has barely any pre-processing time, and it is powerful enough to be preferred over no preconditioner for all but one matrix, and preferred over ILU0 for all matrices when the solver's precision is low. Only when the solver's precision was set higher did the ILU0 preconditioner's power show: the number of iterations required to reach this accuracy was high enough that the pre-processing time no longer dominated the runtime. For most matrices, the ILU0 preconditioner reduced the iteration count enough that the solve performance was better than that of the other preconditioner options tested, with the bcsstk25 matrix being an exception, as its solve was only accelerated by the Jacobi preconditioner. We suspect the matrix' narrow banded structure and high values on the diagonal make it particularly well estimated by the Jacobi preconditioner. The second reason a lower iteration count does not translate to a higher solver performance directly is not captured in our software tests. In other words, different preconditioners might be easier or harder to parallelize, therefore making them more or less worthy to be accelerated on hardware, respectively. The single vector calculation that is the applying of the Jacobi preconditioner is easy to parallelize and therefore is suitable for hardware acceleration. ILU0 application, however, is not that inherently parallel but can be made more parallel when using **graph coloring (GC)**.

2.4 Matrix Reordering

As covered in Section 2.3, the ILU0 application function does not inherently have a lot of parallelism for a hardware accelerator to exploit. However, by reordering the matrix, some parallelism in this function can be obtained. The goal of such a reordering is to group rows together that do not depend on one another (i.e., that do not have non-zero values in columns that have the same index as the row indices of other rows in the group). During our work, we have focused on two ways of doing such a reordering: **level scheduling (LS)** and GC.

2.4.1 Level Scheduling. LS is a way of reordering a matrix by moving the rows and the columns of the matrix. In other words, the rows are grouped together in colors that can be operated on in parallel, then the columns are reordered in the same way to keep the matrix consistent in the (re)order of the unknowns, which is required during ILU0 decomposition. LS not only groups independent rows together into "levels" but also keeps the order of dependencies of the original matrix intact [26]. In other words, if a certain row j that depends on row i comes after row i in the original order, then after LS, row j_{ls} (the row to which row j was moved during LS) will still come after i_{ls} for all possible values of i and j. An LS algorithm will go through the matrix and gather all rows that do not depend on any other rows into the first level. Then it will go through the matrix again and group all rows that only depend on rows in level 0 into level 1. Then it will fill level 2 with rows only dependent on rows from levels 0 and 1, and so on until every row has been added to a level.

2.4.2 Graph Coloring. GC is a term that we use to describe the reordering of matrices into groups ("colors") of independent rows that do not require the maintaining the order of dependencies in a matrix. To be more specific, we use the Jones-Plassmann algorithm [14] to do GC. This algorithm assigns a random value to every row in the matrix, and then groups all rows that have a higher number than all not-yet-grouped rows it depends on into a single color, and keeps repeating this process until all rows have been assigned a color. In this work, GC was chosen over the other method because it is parallelizable and it allows more control over the maximum amount of rows and columns in a color, which will alleviate some of the size restrictions a hardware design may impose. The disadvantage of GC is that it is not guaranteed to find the minimum number of colors, and thus the highest amount of parallelism because of its random nature.

2.4.3 Reordering Scheme Selection. Selecting one of the two matrix reordering methods depends on the answer to how important it is to obtain a solution that is as close as possible to the one obtained without reordering. If the answer is *very important*, then LS is the preferred reordering scheme. However, when the answer is *not important*, because an iterative solver is already estimating the result instead of solving the system precisely, GC can be used if the result is *close enough* to the actual solution.

2.5 Sparstitioning

To increase the scalability of our design and allow it to work on larger matrices, we will use sparstitioning [27]. Sparstitioning is a method for splitting both the matrix and the vector of a matrix-vector operation into partitions so that no unnecessary data is read during the running of the operation on each partition. However, we make two changes to the sparstitioning algorithm. First, we modify how the partitions are chosen to conform with the matrix reordering scheme chosen. Concretely, every color/level will be one partition. Second, we modify how the vector partitions are stored. When sparstitioning is used for an SpMV, the vector that is operated on is stored in a partitioned format, which introduces some duplication of the vector data. However, storing the vector in this way is not convenient because the non-partitioned vector is needed for the vector operations of the solver. Therefore, instead of storing the vector partitions themselves, our design stores an array that denotes for every partition which indices of the vector are part of that partition. Consequently, only the relevant values are read from the vector when computations start on each partition.

Because of the matrix reordering and sparstitioning optimization required to design an HPC iterative solver, we modify the standard CSR matrix storage format in two ways. First, the number of non-zeroes, rows, and vector partition indices used in each partition of the matrix need to be known. These will be collectively referred to as the sizes of each color. Second, the vector partition indices themselves are needed so that the solver knows which values of the matrix are accessed by each color. To account for these CSR modifications, we introduce our own format, **Compressed Sparse Row-Offsets (CSRO)**, which we will formally introduce in Section 4.2.

3 RELATED WORKS

3.1 SpMV Multiplication on FPGAs

As described in Section 2.1, we focus on **Conjugate Gradient (CG)** solvers in this article, particularly on its general version (i.e., not only for symmetric matrices as CG is), BiCGStab. Previous research into accelerating (parts of) CG solvers using FPGA has focused until now mostly on accelerating the SpMV kernel [30] for one good reason: the SpMV function has a wide range of uses in a broad variety of domains, and because of its irregular memory accesses, it is not efficiently performed by classical von-Neumann architectures like CPUs. This makes SpMV a perfect candidate to be accelerated using customizable dataflow architectures that can fully exploit application-specific configurable memory hierarchies. Furthermore, in the context of linear algebra iterative solvers, SpMV takes up most of the processing time, making it an even more interesting target for acceleration. Consequently, a plethora of previous works addressed the topic of accelerating SpMV on FPGAs. Therefore, we only list a few of the performant designs without trying to be exhaustive:

- The SpMV unit proposed by Zhuo and Prasanna [36] achieves a performance of between 350 MFLOPS and 2.3 GFLOP/s, depending on the number of PUs, the input matrix, the frequency, and the bandwidth. However, to realize their design, the authors use zero-padding and column-wise blocking of the matrix. These design choices make SpMV less efficient due to storage overhead that becomes critical for large matrices and very inefficient to use in combination with an ILU0 preconditioner.
- Fowers et al. [12] implemented an SpMV unit that uses a novel sparse matrix encoding designed with the fact that the matrix will be divided over multiple PUs in mind, and a banked vector buffer that allows many memory accesses to the input vector at the same time. That implementation reaches a performance of up to 3.9 GFLOP/s but does not actually outperform GPU or CPU implementations for large matrices.
- Dorrance et al. [8] obtain a performance of up to 19.2 GFLOP/s for their CSC-based SpMV accelerator, which is as far as we could find the highest performance of any FPGA-based implementation of SpMV. This high performance was thanks to the high computational efficiency of the implementation paired with its 64 PUs and high bandwidth (36 GB/s). However, as can be seen in Algorithm 3, in the applying of the ILU0 preconditioner, the result of one row is needed to calculate the result of ones after it. So, unlike the SpMV, the apply_ILU0

cannot be done as easily with CSC as it can with CSR. In fact, the CSC is completely unsuitable to use in conjunction with ILU0, and as a result we will not use that storage format in this article.

Please note that the scope of the work is not to design the fastest SpMV unit possible, but also to integrate this in a larger and complex application that is a preconditioned linear solver. Consequently, although some of the design choices might be sub-optimal for the design of the *fastest* SpMV (e.g., using a CSR-like format instead of the more parallelizable CSC format), which is thus not a primary goal, we focused on the complete integration and looked for the best SpMV design options from a system-level view and not SpMV in isolation as most of previous work considers. Nevertheless, the SpMV designed and implemented in this work is on par with the best SpMV solution we could find in the literature when we consider normalized resources and bandwidth as comparison parameters. Our SpMV is able to achieve up to 4.5 and 8.5 GFLOP/s in double and single precision, respectively.

3.2 Sparse Matrix CG Solvers on FPGAs

Research into designing a complete CG solver on an FPGA has been done as well. However, probably due to the complexity of iterative solver algorithms coupled with a limited area, small on-chip memory, and no HBMs available on previous generation FPGAs, we could find only a handful of related works of solvers on reconfigurable hardware:

- Morris et al. [16] implement a CG solver on a dual-FPGA using both a HLL-to-HDL compiler and custom floating point units written in VHDL. Despite the fact that using the HLL-to-HDL compiler is probably less optimized than writing VHDL directly, they obtain a speedup over software of 30%, showcasing the potential of FPGA in accelerating sparse CG solvers.
- Wu et al. [32] implemented a CG solver on an FPGA consisting of an efficient SpMV unit, a vector update unit, and a diagonal preconditioner. They obtain a speedup between 4 and 9× over their software implementation.
- Chow et al. [6] introduce a novel way of statically scheduling accesses to the on-chip memory in a **Conjugate Gradient** (**CG**) solver on an **FPGA**. Despite the initial cost of performing this scheduling, an average speedup of 4.4 over a CPU solver and of 3.6 over a GPU solver is achieved.

However, different from the work in this article, all of this previous research focused on smaller (i.e., dimension size up to 66,127) scale matrices that are not enough for a HPC real-world application setting. For example, a small reservoir contains more than 10^5 cells and a medium more than 10^6 cells, whereas a large model contains more than 10^7 grid cells (which translates to the dimension N of the underlying matrix). Please note also that double precision is required to store the model data. In this work, we focus on a small reservoir model due to the limited on-chip FPGA BRAM resources required by our design (i.e., the NORN model used in Section 7 has a dimension of 133,293 cells). Another issue that all of these implementations share is that no or only a very simple preconditioner has been used. Preconditioners can improve the performance of a solver significantly, as they reduce the amount of iterations that the solver needs to run until it reaches its desired precision significantly. Despite costing additional time to perform each iteration, the ILU(0) preconditioner used in this work improves the performance of all kinds of CG solvers over cases without preconditioners or with the diagonal preconditioner [29]. Consequently, not using a preconditioner in an iterative solver for a HPC real-world application that requires very large and sparse matrices is not an efficient approach. Finally, using the CG solver limits the applicability of the design to only problems that use symmetric matrices to describe the physical system.

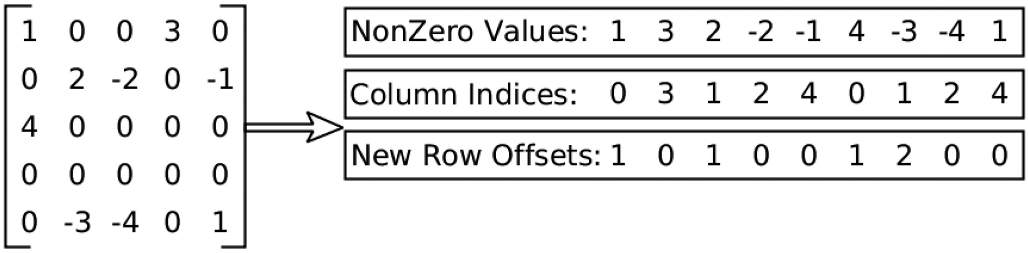

1 0 0 3 0
0 2 -2 0 -1
4 0 0 0 0
0 0 0 0 0
0 -3 -4 0 1
NonZero Values: 1 3 2 -2 -1 4 -3 -4 1
Column Indices: 0 3 1 2 4 0 1 2 4
New Row Offsets: 1 0 1 0 0 1 2 0 0

Fig. 1. An example matrix stored in the novel CSRO format.

4 SPMV MULTIPLICATION

4.1 Design Objective

The main objective of our SpMV kernel on the FPGA is to ensure that the unit has a high computational efficiency, which is required to fully exploit the **HBM** available on the state-of-the-art FPGA generations such as the Xilinx Alveo U280 [33] data center accelerator card. The remainder of this section highlights the design choices of the SpMV implementation for the **HPC** domain when using the latest **HBM**-enhanced **FPGA** accelerators.

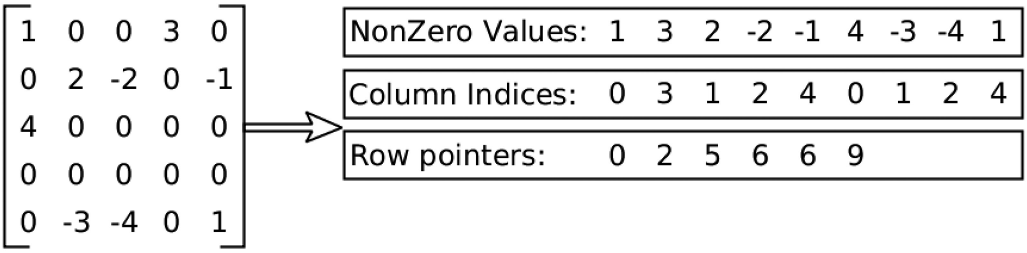
4.2 The Matrix Format

As described in Section 3.1, we have chosen to base our kernel on the row-major CSR matrix storage format. CSR is also the default format used in *Flow*, a real-world HPC software application that we will use in Section 7 to validate our results and benchmark the performance. However, this format poses a challenge for any HPC FPGA-based design because the CSR contains one integer per matrix row to describe how many values are in each row. Since our design objective is to maximize throughput of the SpMV unit, we want to feed as many non-zero values and column indices into the SpMV unit as it can sustain on every clock cycle. Unfortunately, it is not straightforward when using the CSR's *rowPointers* to determine to which row each value belongs to. The number of rows a given number of non-zero values in a matrix belong to can vary wildly based on the matrix's sparsity pattern. Consequently, the number of rowPointers that need to be read every cycle also varies, and also depends on the values of the rowPointers of the cycle before it. A hardware implementation would by necessity need to set a maximum number of row pointers it can read every cycle, thereby putting restrictions on the matrices it can process. To circumvent this, we introduce a novel sparse matrix storage format based on CSR to be used by our accelerator.

Similar to CSR, this new format contains the matrix's non-zero values and their column indices as two arrays, but instead of the rowIndices, it contains a new row offset for every non-zero value in the matrix. Because of the added offset array, this new format is tentatively called *CSRO* format. A value in the newRowOffset array is 0 if the non-zero value at that same index is in the same row as the non-zero value before it. If a non-zero value is not in the same row as the value before it, then the rowOffset at the same index will be equal to 1 + the number of empty rows between that non-zero value's row and its predecessor's row. To help illustrate how a matrix is stored in the CSRO format, Figure 1 shows a matrix as it would be represented in the CSRO format. For comparison, Figure 2 shows how that same matrix would be represented in the CSR format. Note that the NewRowOffset value of the first value in the CSRO format is 1. This is a convention adopted to show that the first row starts with the first value.

4.2.1 *The Blocked CSR.*

The **Open Porous Media (OPM)** platform and its flow simulator do not use regular CSR but rather blocked CSR, in which each non-zero value in the matrix is not one



$$\begin{bmatrix} 1 & 0 & 0 & 3 & 0 \\ 0 & 2 & -2 & 0 & -1 \\ 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -3 & -4 & 0 & 1 \end{bmatrix} \Longrightarrow$$

NonZero Values: 1 3 2 -2 -1 4 -3 -4 1

Column Indices: 0 3 1 2 4 0 1 2 4

Row pointers: 0 2 5 6 6 9

Fig. 2. An example matrix stored in the CSR format.

value but a 3 × 3 matrix of values. This is done to group together all relevant physical information about a single point of the reservoir that the matrix modeled. Unfortunately, the nine values in the 3 × 3 blocks are not convenient for an FPGA. Not only do these blocks often contain zeroes (about half of all values in the blocks of the NORNE testcase are zeroes), but the odd number of values in a block poses a problem when reading them over the ports of the FPGA or storing them in its internal memories, all of which are built to consist of an even number of values. Since the new matrix format that we introduce and the sparstitioning already require extensive reordering of the non-zero values of the matrix, we take this opportunity to also remove the blocking from the matrices in those steps. As a result, the FPGA solver kernel primarily works on non-blocked matrices. The only exception to this is the division at the end of the backward substitution step in the apply_ILU0. Here, a division by the values on the diagonal needs to be done, and this value on the diagonal is a block. These diagonal blocks are the only blocks that could not be removed, and as a result the hardware kernel still uses those blocks as inputs.

4.3 Design Overview

4.3.1 The SpMV Pipeline. The upper limit of the throughput of our SpMV unit design is determined by the number of multipliers in the unit. Thus, to achieve a high computational efficiency, our design aims to stream values into those multipliers every cycle. Figure 3 shows how data is streamed into the multipliers: non-zero values and column indices are read from the memories that hold them, and the column indices are used as addresses at which to read elements from the input vector, whereas the non-zero values are delayed by the delay of the vector memory. This way, the non-zero matrix values and the corresponding vector elements arrive in the multiplier at the same time. There is one memory that holds the vector partition for every two multiplier units, because the BRAM blocks of an FPGA have at most two read ports. Which values of the vector are part of a certain partition is determined by the vector partition indices of that partition. Those values are calculated by the software during pre-processing. Each vector partition memory contains the entire vector partition, so the data is replicated and stored in each one. Although not the most space-efficient solution, this does allow a high number of simultaneous memory accesses at different addresses, which is what the SpMV unit requires.

After the multiplications have finished, the SpMV unit needs to add together the multiplication results that belong to the same matrix row together to obtain the SpMV result of that row. For this, we designed a **selective adder tree (SAT)** and a reduce unit, which receive instructions about which values to add together from a control unit. This control unit in turn obtains this information from the new row offset data. The SAT adds all the values in a single cycle that belong to the same row together, whereas the reduce unit adds together the SAT results from different clock cycles that belong to the same row. SAT results that already represent the final result of a row (i.e., if all the values of that row were in the same clock cycle) do not need to go into the reduce unit. Finally,

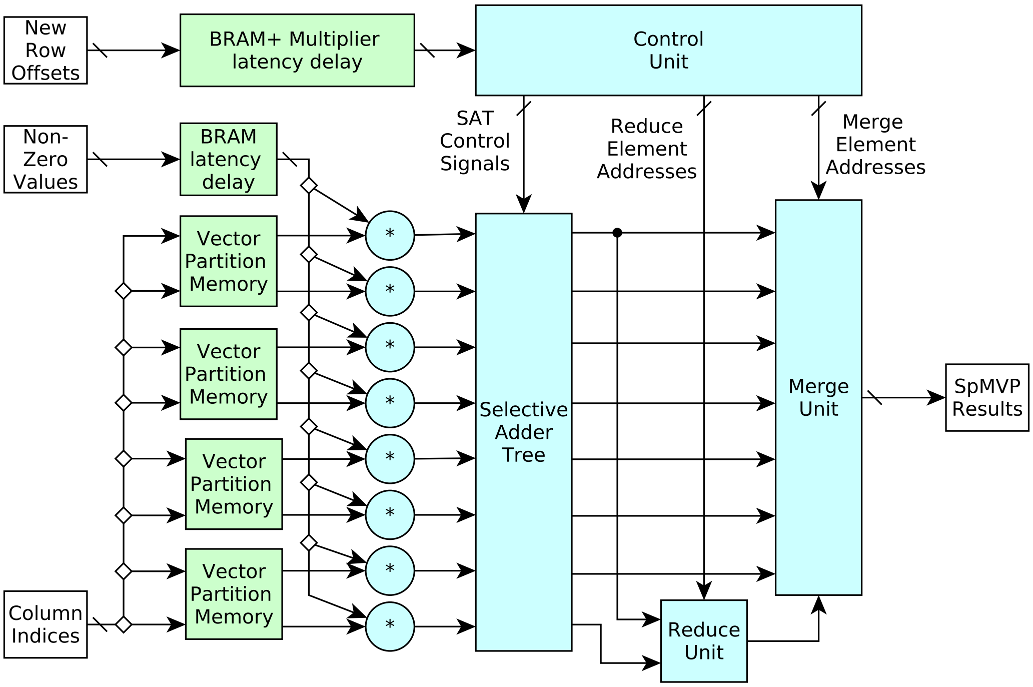


Fig. 3. A high-level schematic of the SpMV unit.

a merge unit merges all of the results of the adder tree and reduce unit into a set number of output ports that will be written to a result memory.

4.3.2 SpMV Top Level. Although the SpMV pipeline is the most important part of the SpMV unit, as it does the calculations, some additional units are needed to keep the pipeline running correctly. These units are as follows:

- The external read unit handles all reads from outside of the FPGA chip. At the start of the SpMV run it reads the sizes of all colors, and it uses these to read all matrix data that is needed during the SpMV.
- For the SpMV unit to be able to use sparstitioning with vector partition indices, the vector values at those indices need to be able to be read quickly. To achieve this, our FPGA design contains a single large URAM memory to store the multiplicant vector once. Due to the current size chosen for this memory in the design, up to 262,144 values can be stored, meaning that a matrix may not have more than 262,144 columns to be able to be multiplied by the kernel. The internal read unit coordinates reads from this memory. It receives the vector partition indices of the current color from the external read unit, reads the vector values at those indices, and sends those values into vector partition memories of the SpMV pipeline.
- The results of the SpMV pipeline can be out of order, as the time some results spend in the reduce and merge units are different from others. The write unit stores the results of the SpMV pipeline in a memory block that has been cyclically partitioned so that it can write multiple results into it at the same time while also reading from it. It also keeps track of up to which index all results have been received. Whenever it has received enough values to fill up a cacheline, it will queue up that line to be sent to the HBM.

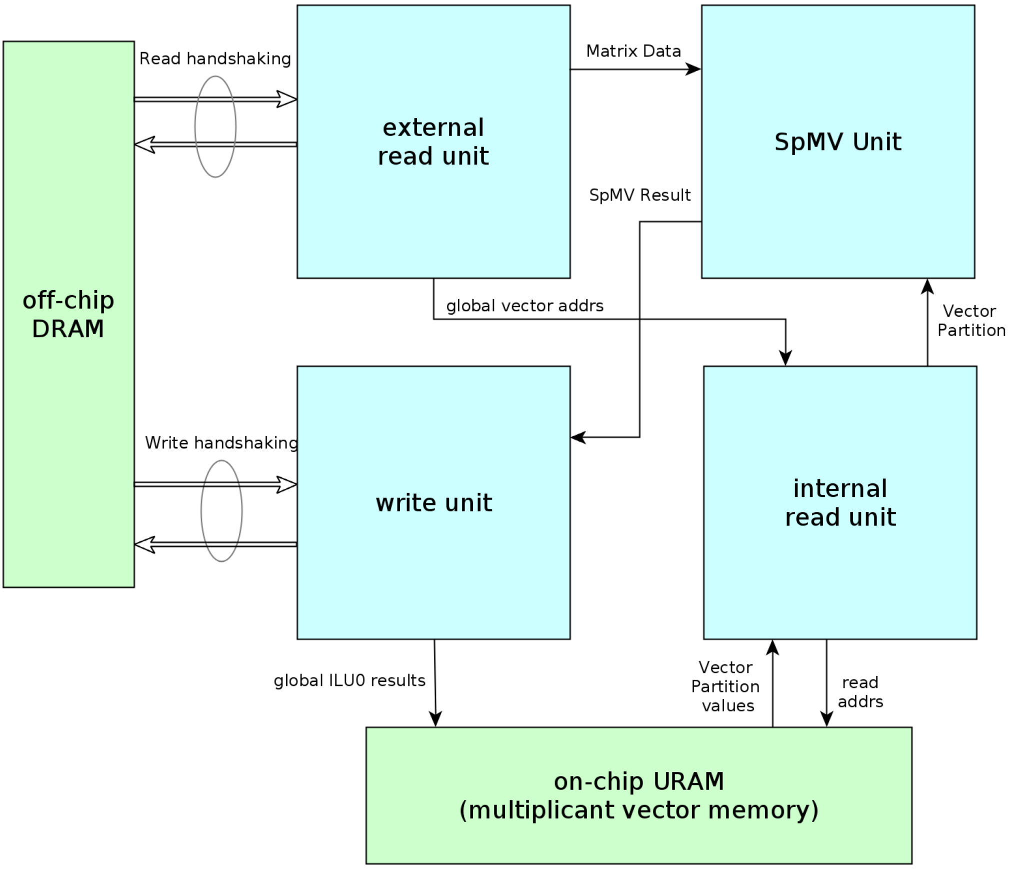
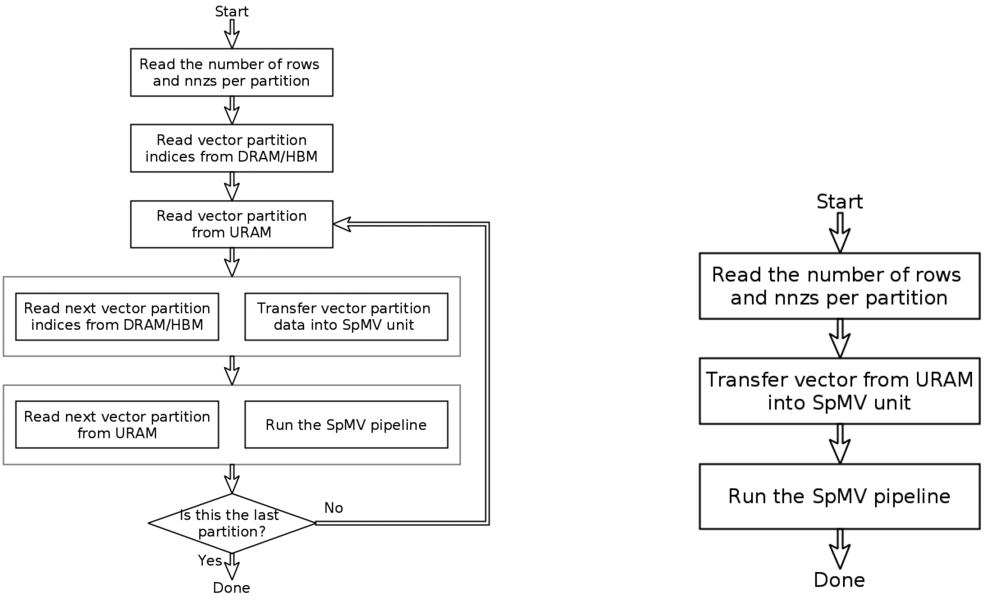


Fig. 4. Schematic of SpMV top level.

An overview of the SpMV top-level unit is shown in Figure 4. Not shown in this figure is the control logic around it that instructs the units how much data to read or write and to/from where. To do this, the control logic keeps track of which color it is currently operating on and how many still remain.

4.3.3 Non-Partitioned SpMV. One of the tasks the SpMV top-level unit must perform is the reading of the vector partition data that the sparstitioning adds to the matrix data. For a large part, this read operation can be done as look-ahead, meaning that while the SpMV pipeline operates on one color, the vector partition data of the next color can already be read from the URAM. This data is then stored in a BRAM block in the internal read unit. However, after the SpMV pipeline completes, this vector partition data needs to be transferred to the vector partition memories of the SpMV pipeline, and the vector partition indices for the next color need to be read from the off-chip memory. This is overhead added by our choice to use sparstitioning. To determine how much this overhead slows down the overall performance, we added the option to the design to not use sparstitioning. This version of the design has as downside that it has a more limited maximum matrix size that it can operate on, as the complete multiplicand vector needs to be stored multiple times in the vector memories of the SpMV pipeline. The advantage is that these additional steps of reading vector partition indices and transferring vector partition data no longer need to occur.


Start
Read the number of rows and nnzs per partition
Read vector partition indices from DRAM/HBM
Read vector partition from URAM
Read next vector partition indices from DRAM/HBM
Transfer vector partition data into SpMV unit
Read next vector partition from URAM
Run the SpMV pipeline
Is this the last partition?
No
Yes
Done
Start
Read the number of rows and nnzs per partition
Transfer vector from URAM into SpMV unit
Run the SpMV pipeline
Done

(a) Sparstitioned SpMV version

(b) Non-Sparstitioned SpMV version

Fig. 5. Overview of the tasks in different **SpMV** unit implemenrtations.

To visualize the difference in way of operating between the sparstitioned and the non-sparstitioned SpMV unit, Figure 5(a) shows the steps the sparstitioned SpMV unit takes, whereas Figure 5(b) shows those the non-sparstitioned SpMV unit takes. Tasks horizontally next to one another in these figures are performed at the same time. Obviously, the flow in Figure 5(b) is much simpler. Also note how in Figure 5(a) the reading of the next vector partition (and the reading of the indices to read from) are already done while the SpMV unit is still working on another partition. This hides the time these reads would normally take but also necessitates an additional memory to read the next vector partition into, and a step in which the data is transferred from this memory into the SpMV unit.

4.4 GPU SpMV Kernels

For the GPU, kernels in CUDA and OpenCL are tested. Only doubles are used for non-zero values. NVIDIA's cuSPARSE [18] features a blocked SpMV function called *cusparseDbsrmv()*. For OpenCL, Algorithm 1 from Eberhardt and Hoemmen [10] is used. In this algorithm, a warp of 32 threads is assigned to a single block row, and threads are assigned to elements in the block row. To cover the entire block row, a warp iterates, handling $floor(32/bs^2)$ blocks at a time, where *bs* is the blocksize. A warp only operates on complete blocks, so some threads might be always idle. For 3×3 blocks, a warp can cover three blocks with 27 threads and has 5 inactive threads. During initialization, a thread calculates the index of the block row, how many blocks the row has, and which block it is supposed to operate on. It also calculates its position within a block (row and column). Since a warp only covers complete blocks, a thread's assigned position in a block never changes. During iterating, it multiplies the target element from A with the value from *x*, adding the product to a running total. Once the whole block row is complete, threads with the same vertical position (row) in the blocks reduce the values in their local registers and write the reduced output to global memory. Reduction can be done with shuffling or via shared memory.

5 SPARSE MATRIX SOLVER DESIGN

To expand the SpMV kernel into a kernel that can perform preconditioned BiCGStab, units to perform both the preconditioning and the vector operations need to be added. In this section, the design of those units is discussed, followed by a description of the complete design that links all units and performs the complete preconditioned solve.

5.1 Applying the ILU0 Preconditioner

As shown in Algorithms 2 and 3, the basic structures of the SpMV and the forward and backward substitutions that the ILU0 application performs are similar. This, combined with the fact that the SpMV and ILU0 application are never active at the same time, since the result of one is needed to start the other, makes reusing the hardware of the SpMV unit for the ILU0 apply a desired feature to save resources. Unlike the SpMV, which goes through the matrix it operates on line by line, the ILU0 application goes through its LU matrix in a non-sequential order. First, it goes through all of the values below the diagonal top to bottom, and then it goes through all values above and including the diagonal bottom to top (last row first). This memory access pattern is not efficient to use on the FPGA, as it needs to read all LU matrix data twice, ignoring about half of it each time. Which data it needs to use and which it does not depends on the current row number and the column index. This memory access pattern is not only inefficient for the FPGA, but in general, so in our design the lower-triangular matrix L, the upper triangular matrix U, and the diagonal values are split from the LU matrix, which are used as three separate data structures. First, L is read to perform the forward substitution, and then U and the diag_vals are read to perform the backward substitution. Since the latter step happens bottom to top, both the U matrix and the diag_vals are stored in reverse order.

The main difference between the substitutions and the SpMV is that the substitutions modify the vector that they are operating on, whereas the SpMV does not. All rows that are grouped into the same color by GC can be in the pipeline at the same time, but between the colors, the results of the previous color need to be integrated into the vector that is currently being operated on. To achieve this, the results of the ILU0 application are not only written to the URAM vector memory but also forwarded directly into the vector partition memories so that they can be used during the next color, without requiring the unit to wait with reading its vector partition until the previous color finishes its computation.

There are additional operations that the forward and backward substitution needs to apply over the SpMV. For the forward substitution, this is a subtraction of the SpMV unit result from the p vector, and for the backward substitution, it is this subtraction followed by a division by the diagonal value. The ILU0 unit is added to the SpMV unit to perform these operations. The reading of the P vector can be done directly from the URAM, since that is where the P vector is stored during the ILU0 application. The diagonal vector is read from off-chip memory by the external read unit. Figure 6 shows how the original SpMV top level was modified to allow the unit to also apply the ILU0 preconditioning step. Note that this unit is not yet the top level of the complete solver.

5.2 The Vector Operations

Over the course of a BiCGSTAB run, a number of vector operations of three different kinds need to be applied:

- A dot product, which calculates the inner product of two vectors.
- An axpy, which performs $\alpha * \vec{x} + \vec{y}$.
- A norm, which calculates the absolute value of a vector (i.e., $\sqrt{\vec{x} * \vec{x}}$).

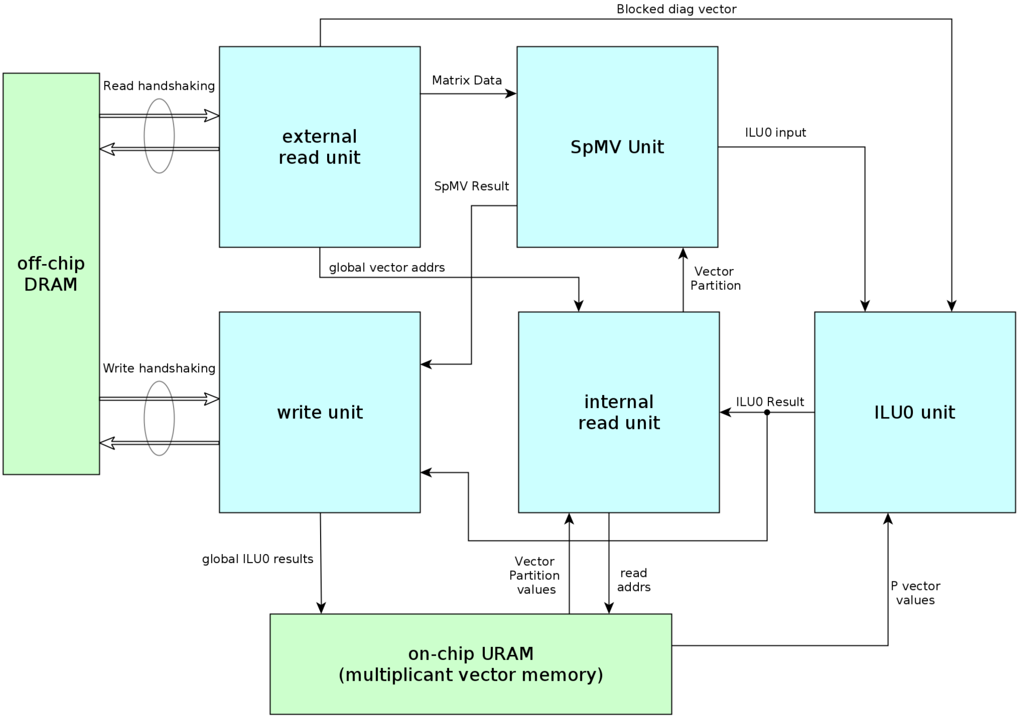


Fig. 6. Schematic of the SpMV/ILU0 unit.

Of these, the dot product and norm are very similar in structure, as a norm is the square root of an inner product of a vector with itself. All three operations are highly parallel and perform the same basic operations (one multiplication and one addition) for every element in their input vectors. For this reason, and to provide more flexibility in exploiting the task-level parallelism in the solver, we designed a unit that can perform either an axpy or a dot product. This dot_axpy unit contains an equal number of multiply and add floating point units, which can be connected in different ways, depending on which function needs to be performed. When an axpy needs to be performed, the adders and multipliers are connected in parallel pipelines that each perform the axpy on one element of the input vectors at a time. For this function, an additional input port is used for the scaling constant α, besides the two input vector and one output vector ports. Figure 7 gives an overview of how the adders and multipliers are arranged in the dot_axpy unit when it performs an axpy operation.

For the dot products, all multipliers work in parallel, the results of which are then added together into a single value by a tree of adders. The result of this adder tree is added together with the result of previous cycles in the final adder, which adds a new input to its most recent output. After the adder tree has processed all values of the input vectors, logic around the final adder continuously feeds two subsequent valid output of that adder back into it, until one singular final result is left. Figure 8 gives an overview of how the adders and multipliers are arranged in the dot_axpy unit when it performs a dot product.

During the vector operations, a degree of task-level parallelism exists in the BiCGStab solver algorithm. This parallelism can be found in the following operations (using line numbers from Algorithm 1):

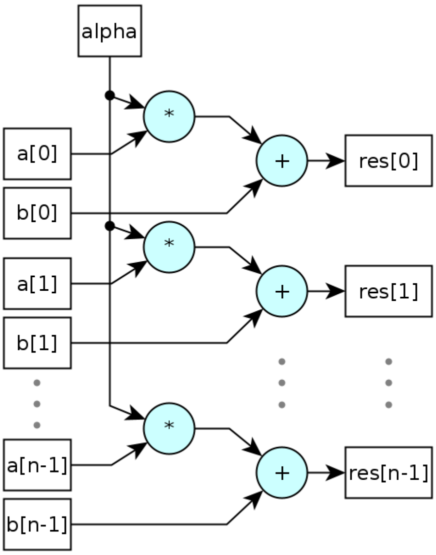


Fig. 7. Schematic overview of the dot_axpy unit in *axpy* mode.

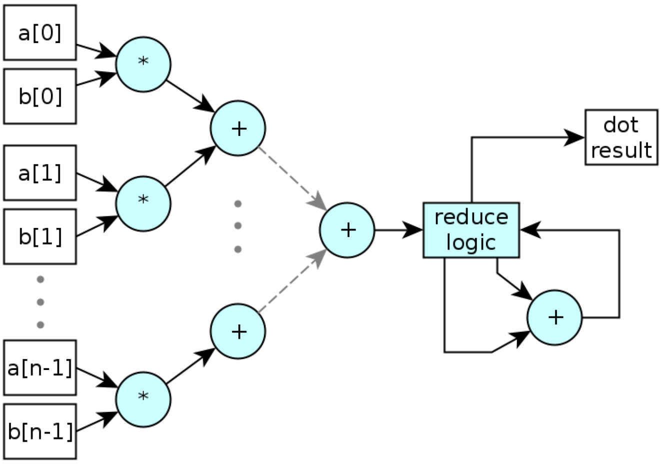


Fig. 8. Schematic overview of the dot_axpy unit in *dot* mode.

- The vector subtraction in line 1 can be pipelined with the SpMV operation in that same line so that the subtraction of each element of $\vec{b}$ happens as soon as the SpMV result in the same position as that element is done. The dot product in line 5 can be pipelined with the subtraction of line 1 in the same way.
- The two axpy operations in line 10 can be pipelined.
- The dot product in line 13 can be pipelined with the SpMV in line 12.
- The axpy operations in lines 15 and 16 can be done in parallel, and the dot product in line 17 can be pipelined with the axpy in line 16.
- The two dot products in line 23 can be done in parallel, and can both be pipelined with the SpMV in line 22.

- The axpy operations in lines 24 and 25 can be done in parallel. The dot product in line 26 can be pipelined with the axpy in line 25, and so can the dot product in line 8 in the following iteration.

Together, these instances of possible task-level parallelism cover all vector operations that the BiCGStab performs. The number of vector operations that can be performed in parallel in each of these instances is, in order: 2, 2, 1, 3, 2, and 4. The complete vector unit contains multiple dot_axpy units to exploit this task-level parallelism; however, to avoid having dot_axpy units being idle often, and to not increase the number of read/write ports too much (see Section 5.3), we choose to implement 2 dot_axpy unit in the *vector operations* unit.

5.3 Read and Write Ports

The Xilinx Alveo U280 data center accelerator card has two different types of large memory that the FPGA chip is connected to: an off-chip DDR memory that consists of two 16-GB DDR4 memory banks, each of which is connected to the FPGA via a single read/write port, and an 8-GB on-chip HBM stack that is connected to the FPGA via 32 read/write ports. Each memory port is accessed at the kernel level either via a 512-bit wide AXI4 interface (for the DDR memory) or via a 256-bit wide AXI3 interface (for the HBM). This memory architecture has led to the following design decisions:

- Reading and writing delays from/to the HBM are slightly shorter than those from/to the DDR memory, so all vectors (which all need to be read and written multiple times during the solver run) are stored in the HBM.
- To reduce the number of memories to which the host CPU needs to transfer its data, and to thereby reduce the initial data transfer time, all matrix data and the initial B vector are stored on the DDR memory.
- Some vector operations read from and write to the same vector—for example, the axpy in line 15 of Algorithm 1. To avoid conflicts and to make data buffering easier, each AXI port is used either to read or write, hence the x, r, and p vectors are stored in two memory locations connected to different ports. During an operation to update one of those vectors, it is read from the port that it was written to most recently, and written to the other port. This essentially makes use of the ping pong buffering technique used to stream data efficiently.
- All HBM ports are located on one of the two narrow sides of the FPGA chip (which has a rectangular shape), where the HBM crossbar and memory stack are located. Moreover, the FPGA is subdivided into three physical regions, each one being a single silicon die and called **Super Logic Region (SLR)**. Signals crossing these regions incur an additional delay compared to signals routed in the same SLR. Each instance of an AXI port (required to use an HBM port) comes with a cost in terms of routing (512 bits of data plus control bits) and BRAMs that are used for data buffering. The main factor for the design's logic to spill over to another SLR, causing the additional delays due to the signals crossing between SLRs to reach the HBM ports, is BRAM usage. To reduce such spilling, and to avoid routing congestion due to the high number of signals used by each AXI port, we choose to limit the number of HBM ports. We choose to use six HBM ports, because this number of read and write ports can be efficiently used during the vector operations.
- Since the width of the read ports is 512 bits, we design our computation pipelines to be able to accept one full cache line every cycle. For the HBM ports, which are natively 256-bit wide, we let the implementation tools to handle the AXI port width conversion. That means each dot_axpy unit has eight double-precision multipliers and adders.

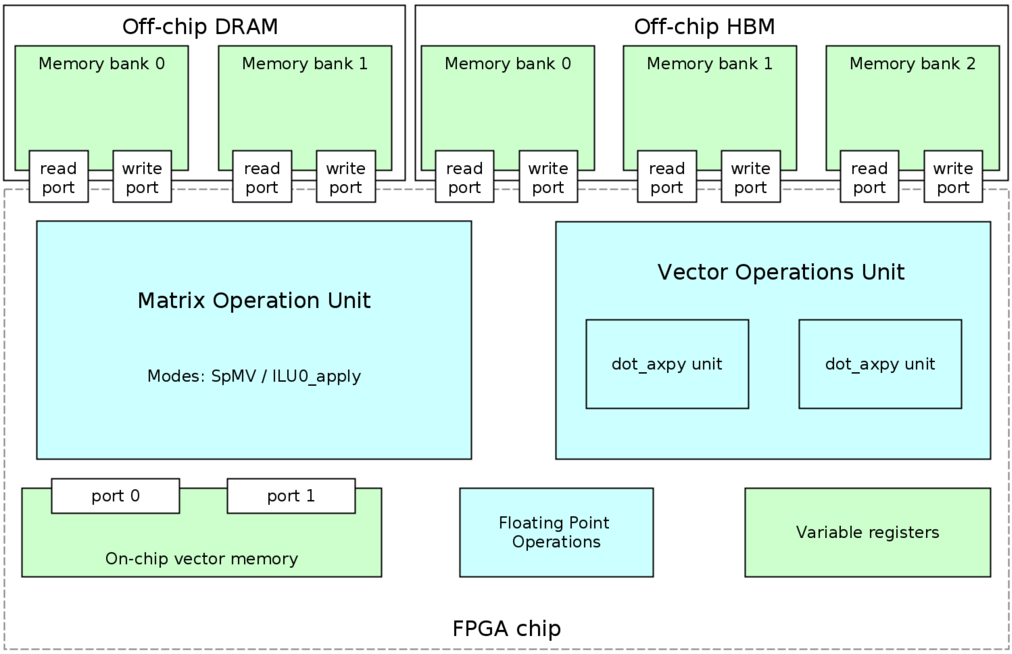


Fig. 9. Schematic overview of the complete solver design.

5.4 The Complete FPGA Solver

A top-level solver unit was designed to encompass both the SpMV/ILU0 unit and vector operations unit. This top-level unit instructs those units which tasks to perform and with which input/output vectors. It also regulates the access of both units to the URAM internal vector memory and handles filling of this memory during the initialization step of the solver. Figure 9 gives a schematic overview of the top-level unit of the BiCGStab solver kernel. To not overly complicate the figure, the signals between the units and the memory are not pictured. Besides the units already covered in the previous sections, the figure also shows a block of floating point operators and a memory for floating point variables. The operators are used to apply the operations that only need to be performed on a single value at a time, such as the square root of the norm calculation or the multiplications and division of the β calculation in line 9 of Algorithm 1. The memory is used to stored the results of these operations, and contains α, β, ω, ρ, ρ_{new} and conv_threshold.

5.5 The GPU Solvers

For completeness of the evaluation and comparison of our work, we also developed a couple of GPU solvers using first CUDA and the NVIDIA's cuSPARSE library [18]. cuSPARSE has functions for vector operations like dot, axpy, and norm, as well as a blocked ILU0 preconditioner. Second, we also implemented the solver using OpenCL [17], where a similar structure to the CUDA is used, defining kernels for vector operations and applying the ILU0 preconditioner. One big difference is that the construction of the ILU0 preconditioner (the decomposition) is done on the CPU. This has not been implemented on GPU yet, whereas CUDA features an ILU0 decomposition on the GPU in cuSPARSE.

For a high-level overview of the complete flow simulator that will be used to integrate the different accelerated solvers, please refer to Figure 10.

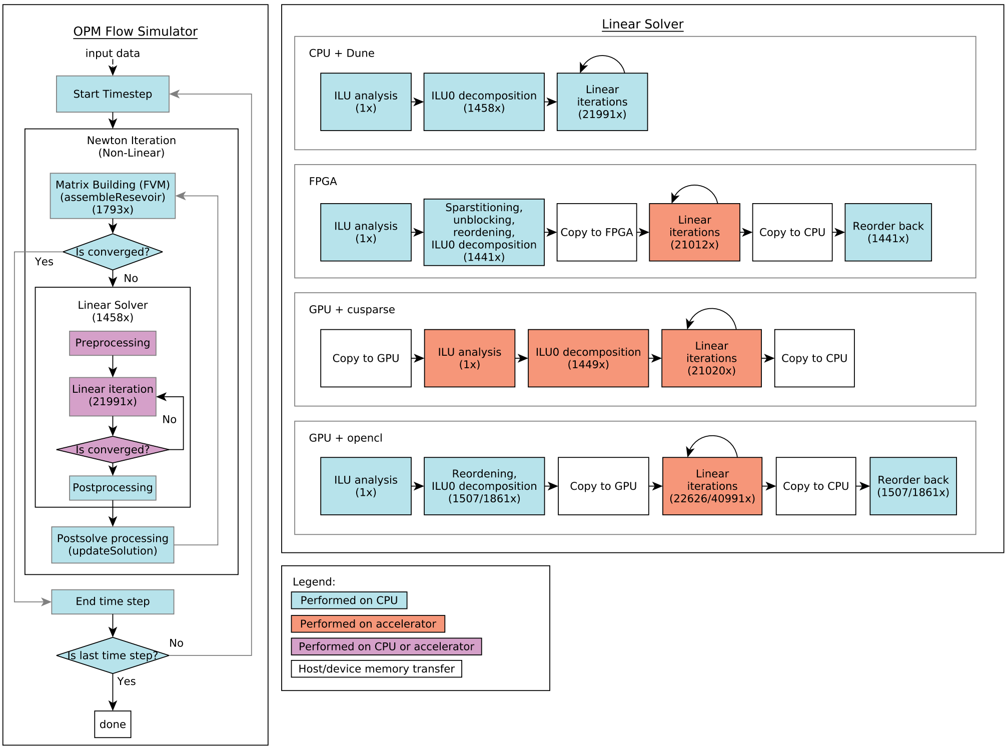


Fig. 10. A block diagram representing the full flow simulator (left) and the different linear solver implementations (right). The numbers between brackets indicate how many times a function is performed for that implementation.

6 PERFORMANCE MODELING

To predict the performance of both the hardware design as it was in development as well as to gauge how well the design scales, we constructed a performance model that we run in Matlab. This model was made to emulate the data flowing through the computation units and to calculate how many clock cycles the execution would take. This section describes first what parts of the design are modeled and which are not, showing second the predicted modeling results.

6.1 Model Details

The performance model is composed of functions that model the individual hardware kernels: it has a function to estimate the performance of the SpMV, one for the ILU0, and one for the vector operations. To make the performance estimations as accurate as possible, the hardware models work on the same data that is sent to the kernel, including the partitioning data. The exception being that since the model does not actually perform the solving calculation, it does not read the non-zero matrix values or any vector data. Nevertheless, the partitioning and sparsity pattern information are needed to calculate the performance. Furthermore, the functions that calculate the performance of the ILU0_apply and SpMV operations share the same structure, in the same way that those operations share the same module in the hardware kernel. These functions do the following for every color:

- First, it calculates the time to transfer the vector partition data into the SpMV unit. It does this by dividing the number of values in the partition by the number of ports allocated to the internal URAM memory from which the partition is read.
- If apply_ILU0 is performed, the function calculates the time the transfer of the P vector into the ILU0 unit memory would take in the manner described previously.
- Then, it calculates the time the SpMV pipeline takes. During this step, the function's emphasis is on the time the reading of the matrix data takes. It assumes a set bandwidth for each of three ports from which the three matrix arrays (non-zero values, column indices, and new row offsets) are read and simulates reading into three FIFOs at the speed set by that bandwidth. Whenever enough data is available in the FIFOs for all three arrays to fill an input line of the SpMV pipeline, that data is removed from the FIFOs.
- After enough data has been read from all three ports for the current color, and the simulated FIFOs are empty, a set delay is added for the SpMV pipeline to write back the result data. Since much less data needs to be written back than needed to be read, no delay beyond the regular write overhead of the final lines is deemed necessary.
- If apply_ILU0 is performed, a set delay for the ILU0 unit is added, and the write delay is calculated with the on-chip URAM instead of an off-chip memory.

The cycle count results of all colors are added together to find the total number of clock cycles that the SpMV or apply_ILU0 takes. The calculations done to estimate the cycle count of the vector operations are comparatively more simple but follow the same method: first the reading of data into FIFOs at a set bandwidth is modeled, data is removed from the FIFO whenever enough is in it to fill up a line of inputs of the vector operation unit, and finally, when all reading is done, the latency of the pipeline of the operation is added, as well as some writing overhead cycles for the axpy operation.

A top-level function adds together the cycle count results of apply_ILU0, SpMV, and vector operations to get to a final estimate. This top-level function has a number of variables that can be changed to do domain space exploration and to assess the scalability of the kernel. These are as follows:

- The number of multipliers and adders in both the vector operation and SpMV/ILU0 units.
- The bandwidth to the off-chip memory in are located.
- The bandwidth to the on-chip URAM in number of ports (how many values of the vector stored on-chip can be read simultaneously).
- The delay of the adder and multiplier units
- The matrix that is modeled for, and how long the solve of that matrix takes, in iterations.

Please note that the largest source of performance uncertainty in the design is the time the memory operations take. Memory accesses do not happen at a constant pace, but in bursts, the size of which and the time between them may vary based on various factors related to the memory. As a result, the weakest part of the model is that it does not accurately capture this memory behavior. Similarly, it does not support modeling reading two arrays of data from the same port, nor the performance penalties from reading and writing on the same memory bank at the same time.

6.2 Modeling Results

We use the model to analyze the scalability of the design and to understand what the critical points are in achieving the best performance. We perform a domain space exploration by varying the model parameters, starting with the number of adders and multipliers set at 2, increasing the bandwidth from 10 GB/s, and increasing the number of internal ports from 2. When a parameter

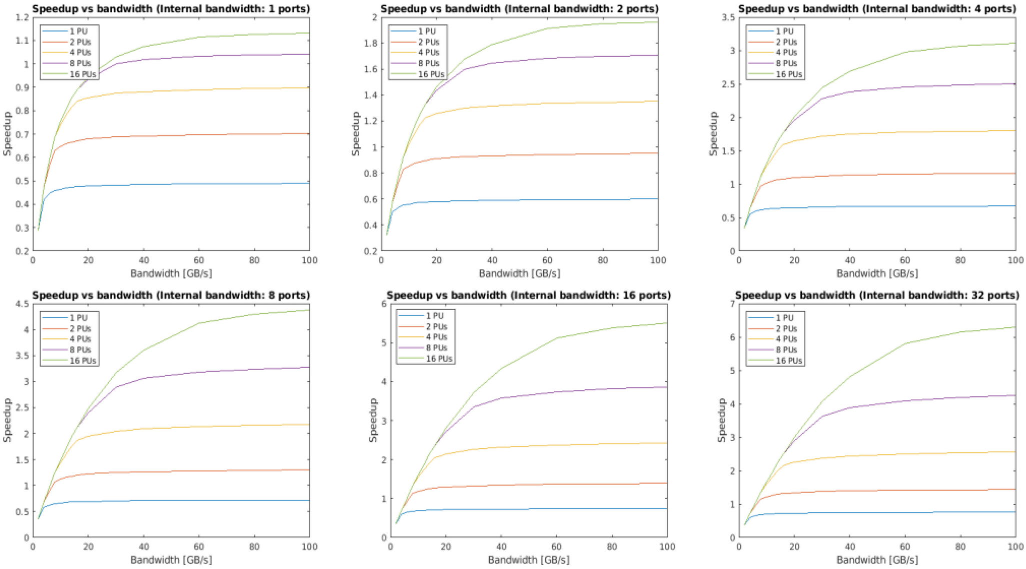


Fig. 11. Domain search exploration for the ILU0-BiCGStab solver. One internal port has a maximum bandwidth of around 18 GB/s at the current kernel frequency of 280 MHz. The configuration achieved in this work for the FPGA solver was two internal ports, eight PUs, and 50-GB/s external bandwidth. This corresponds to the parse line in the top middle figure that estimates a 1.6× speedup, which was confirmed by the experimental results presented next.

does not give a performance increase anymore, we switch to increase the new bottleneck parameter. We perform this exercise by increasing all parameters until a maximum value we select as an upper bound of a maximum practical boundary for our design. The limit for the number of PUs was set to 16 based on the hardware utilization results we obtained, especially the BRAM utilization (see Table 7). A total of 32 PUs might not be realistic as the BRAM utilization would be near 100%, which would make routing the design and meeting timing very challenging. For this number of PUs, 100 GB/s of bandwidth proved enough to approach the maximum performance for any of our models.

Figure 11 shows the modeling results. The first conclusion we can draw from these results is that the number of read ports to the internal vector memory has a major impact on the overall performance of the solver, even if we try to hide the time these memory accesses take. With only one port, barely any speedup can be achieved over the software implementation, whereas a speedup of more than 6× could be achieved with 32 ports, given enough bandwidth. This serves as an indication that the FPGA, with its ability to implement specialized memory infrastructure for multiple simultaneous random accesses, could achieve higher performances than we achieved in this work.

The second conclusion we draw from the modeling results is that doubling the PUs does not double the performance. We can attribute this to the increasing impact of the latency of the SpMV and ILU0 pipelines as the number of PUs increases. With a higher number of PUs, more data can go into the pipeline at the same time, decreasing the number of cycles that new data is fed into the pipeline while the latency of the pipeline stays the same. This effect is especially pronounced for the ILU0 solve steps, which have a longer pipeline, and the latency of which cannot be hidden by already starting operations on the next partition, because that computation depends on the results of the previous partition. Since doubling the number of PUs almost doubles the amount of logic and BRAMs used by the design, one must consider whether adding more PUs is worth the

Table 2. Characteristics of the Platforms Used to Benchmark the Kernels

Short Name	Host			Accelerator	
	CPU (#Total Cores)	Frequency	Memory (#Channels/ Bandwidth)	Name	Memory (Bandwidth)
FPGA	Xeon E5-2640V3 (16)	2.6 GHz	DDR4-2133 (4 / 59 GB/s)	Xilinx Alveo U280	DDR4+HBM (498 GB/s)
GPU^1	Xeon E5-2640V3 (16)	2.6 GHz	DDR4-2133 (4 / 59 GB/s)	Nvidia K40m	GDDR5 (288 GB/s)
GPU^2	Xeon E5-2698V4 (40)	2.2 GHz	DDR4-2400 (4 / 77 GB/s)	Nvidia V100-SXM2-16GB	HBM2 (900 GB/s)
CPU	Xeon E5-2698V4 (40)	2.2 GHz	DDR4-2400 (4 / 77 GB/s)	–	–

Table 3. SpMV Parameters of Different SpMV Implementations

Parameter	double_p	double_np	float_p	float_np
Num mults	8	8	16	16
Max rows	262,144	65,536	262,144	65,536
Read ports	4	4	4	4

additional hardware resources, and the effort of having these resources meet timing. Perhaps more performance could be found more readily by improving the internal memory, as described earlier, or by reducing the apply_ILU0 latency.

7 EXPERIMENTAL RESULTS

Evaluation of the performance of our design is done in three stages. First, the performance of the stand-alone SpMV unit is evaluated, followed by the performance of the complete solver standalone. Finally, the FPGA solver is integrated into the flow simulator, and its performance is compared to that of a CPU and a GPU. All FPGA bitstream implementations were done in Xilinx Vitis 2019.2 and run on a Xilinx Alveo U280 data center accelerator card [33]. In this section, performance for the SpMV unit will be reported in GFLOP/s (Giga FLoating-point OPerations per Second). These are calculated by dividing the total number of floating point operations performed by the time it took to perform them. Therefore, the total number of required operations is 2× the number of non-zero values in the matrix that is being multiplied (since one multiplication and one addition need to be performed on each non-zero value).

Table 2 lists the platforms we used to benchmark the kernels and the flow simulator. Each platform will be then referred to in the rest of the article with its short name (e.g., *FPGA*).

7.1 SpMV Results

The SpMV unit has many configurable parameters that will impact both performance and FPGA resource usage. For this performance evaluation, we choose to compare between single-precision (float) and double-precision (double) implementations, as well as between partitioned (_p) and non-partitioned (_np) implementations. Other parameters were set to the values in Table 3.

The resource utilization of the four chosen SpMV kernel implementations is shown in Table 4. From this table, we can observe that the BRAM utilization sets the limit of how often this kernel could be replicated on this same FPGA device. All implementations are able to reach a frequency of 300 MHz, which is the maximum frequency at which the AXI memory interfaces can run.

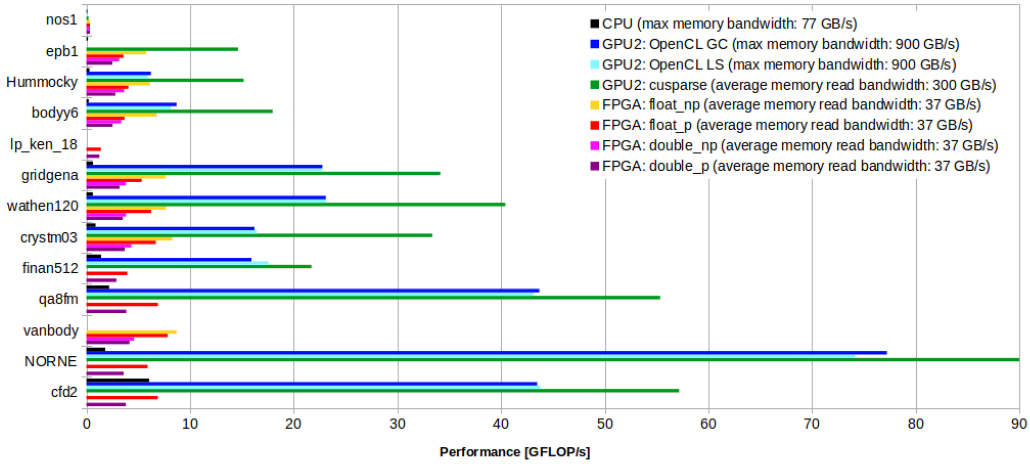


Fig. 12. SpMV performance. Please note that for the FPGA, the maximum aggregated HBM bandwidth is 460 GB/s; however, due to design limitations caused by implementation issues, the maximum bandwidth used by the kernel is around 50 GB/s.

Table 4. FPGA Resource Utilization of Different SpMV Kernel Implementations as a Percentage of the Resources Available in the FPGA's Dynamic Region

Resource	double_p	double_np	float_p	float_np
LUT	4.08%	4.12%	4.92%	5.00%
LUTRAM	0.70%	0.69%	0.82%	0.78%
REG	4.14%	4.45%	4.96%	5.24%
BRAM	15.27%	12.38%	18.97%	12.38%
URAM	4.17%	6.67%	2.50%	6.67%
DSP	0.93%	0.93%	0.74%	0.74%

The sizes and types of matrices that were used to test the SpMV unit are listed in Table 5. All of the listed matrices were obtained from the SuiteSparse matrix collection [7], except for the NORNE matrix, which models a real-life oil field, and is a testcase in the OPM framework. As such, this matrix was not obtained from the SuiteSparse matrix collection but instead by exporting it from the flow simulator. Please note that all but one of the matrices are square, and of the square matrices, three of them are non-symmetric. These matrices were chosen to prove that the SpMV unit works on such non-symmetric and non-square matrices. However, some of these matrices are not used as benchmarks for the BiCGSTAB solver, as it does not work for non-square matrices and has additional limitations as explained in the next section.

In Figure 12, the performance for all matrices used to benchmark the kernel are shown. The corresponding tabulated values can be found in Table 6. For the FPGA and GPU, those performance numbers do not include the data transfer time between the host CPU and the on-board memories. From this figure, we can see that the single-precision versions of the FPGA kernel achieve almost 2× more FLOPS than their double-precision counterparts. This can be explained by the fact that the amount of data that needs to be read for each single-precision operation is half the amount of data that needs to be read for a double-precision one. We also observe, as expected, that the non-partitioned version achieves a slightly higher performance than the corresponding

Table 5. Characteristics of the Matrices Used to Benchmark the SpMV Kernel

Name	Rows	Columns	NNZs	Density%	Symm.
nos1	237	237	2,115	1.811	Yes
epb1	14,734	14,734	95,053	0.044	No
Hummocky	12,380	12,380	121,340	0.079	Yes
bodyy6	19,366	19,366	134,748	0.036	Yes
lp_ken_18	105,127	154,699	358,171	0.002	No
gridgena	48,962	48,962	512,084	0.021	Yes
wathen120	36,441	36,441	565,761	0.043	Yes
finan512	74,752	74,752	596,992	0.011	Yes
qa8fm	66,127	66,127	1,660,579	0.038	Yes
crystm03	24,696	24,696	1,751,310	0.096	Yes
vanbody	47,072	47,072	2,336,898	0.105	Yes
NORNE	133,293	133,293	2,818,879	0.016	No
cfd2	123,440	123,440	3,087,898	0.020	Yes

Table 6. Performance in GFLOP/s for the SpMV Kernel

Matrix	FPGA				GPU[2]			CPU
	double_p	double_np	float_p	float_np	cusparse	OpenCL LS	OpenCL GC	(double)
nos1	0.34	0.34	0.34	0.34	0.20	0.08	0.08	0.00
epb1	2.50	3.17	3.57	5.76	14.62	–	–	0.15
Hummocky	2.79	3.64	4.05	6.11	15.17	5.92	6.22	0.29
bodyy6	2.52	3.36	3.67	6.78	17.97	8.17	8.69	0.22
lp_ken_18	1.25	–	1.40	–	–	–	–	–
gridgena	3.21	3.81	5.33	7.64	34.14	22.76	22.76	0.63
wathen120	3.51	3.82	6.24	7.67	40.41	23.09	23.09	0.62
crystm03	3.69	4.34	6.69	8.29	33.36	16.44	16.22	0.88
finan512	2.89	–	3.95	–	21.71	17.56	15.92	1.41
qa8fm	3.85	–	6.90	–	55.35	43.13	43.70	2.18
vanbody	4.15	4.60	7.84	8.69	–	–	–	–
NORNE	3.59	–	5.90	–	90.93	74.18	77.23	1.81
cfd2	3.80	–	6.89	–	57.18	43.80	43.49	6.04

partitioned version, at the trade-off that the non-partitioned version cannot operate on the larger benchmark matrices.

7.2 Stand-Alone FPGA Solver Results

For the ILU0 BiCGSTAB solver, we are only interested in a version of the solver that can run on larger matrices with high precision, so only the double-precision, partitioned version of the solver is benchmarked. The design parameters as listed for the double_p SpMV implementation in Table 3 hold for this solver implementation as well, except for the number of read ports, which is increased by 1.

In Table 7, the resource utilization of the ILU0 BiCGSTAB solver is compared to that of the SpMV kernel with the same parameters. In both cases, these values refer to the percentage of resources used over the available ones in the dynamic region (which is reconfigurable by the user), and they include all the supporting circuitry to control the solver and access the memory on the FPGA board. Hence, not included in these values are the resources used by the FPGA's static region, which is

Table 7. FPGA Resource Utilization of the Solver Compared to the SpMV Kernel as a Percentage of the Resources Available in the FPGA's Dynamic Region

Resource	SpMV	Solver
LUT	4.08%	6.93%
LUTRAM	0.70%	1.13%
REG	4.14%	6.41%
BRAM	15.27%	24.64%
URAM	4.17%	4.17%
DSP	0.93%	3.18%

Table 8. Characteristics of the Testing Matrices

Name	Dim.	Non-Zeroes			
		FPGA Matrix	FPGA L Matrix	FPGA U Matrix	GPU Matrix
nos1	237	1,017	390	390	1,017
bodyy6	19,366	134,197	121,414	70,762	134,197
gridgena	48,962	513,060	421,068	315,336	513,060
crystm03	24,696	583,770	279,519	279,522	583,770
wathen120	36,441	565,761	468,529	379,446	565,761
qa8fm	66,127	1,660,564	1,431,908	1,251,238	1,660,564
NORNE	133,293	1,314,999	726,369	513,512	2,818,879

provided by Xilinx. Due to the increased complexity of the solver over the SpMV kernel, it could only reach a frequency of 280 MHz.

Some of the matrices used to test the SpMV unit were not usable during the testing of the solver, because they were non-square, had zero values on their diagonals, or exceeded on-board memories of the design. Table 8 lists the matrices that were used to test the BiCGStab solver, along with their original sizes and the size of the derived L/U matrices used by the FPGA solver.

The performance results obtained after running the stand-alone ILU0 BiCGStab solver are shown in Figure 13. For each matrix, we present the best result obtained either by using the GC or the LS reordering algorithms. The detailed timings are listed in Table 9, which contains the runtime as reported by the profiling info (*Solver*) of the runs (only for the FPGA, used in Figure 13), the runtime as measured by the *Host* (used in the figure for the CPU and GPU results), and the time spent transferring (*Transfer*) the input data and the output results between the host CPU's and the accelerators' on-board DRAM/HBM. In addition, the number of iterations (*Iter#*) needed to solve the system are reported. The *Host* time is, in all cases, higher than the *Solver* time, as the former includes overhead of starting and waiting for the end of the solver kernel. The transfer times are not included in the other two times.

In Table 10, we report the bandwidth utilization for each memory port as recorded by the profiling facilities present in the FPGA solver. Each column shows a memory port, along with its mapping (either to DDR4 memory or HBM). The bandwidth utilization is given a percentage of the maximum bandwidth available for each individual port, which is 18.8 GB/s for ports read 0-1 (DDR4 memory, 512-bit AXI bus @ 300 MHz), whereas it is 14.1 GB/s for the remaining ports (HBM, 256-bit AXI bus @ 450 MHz).

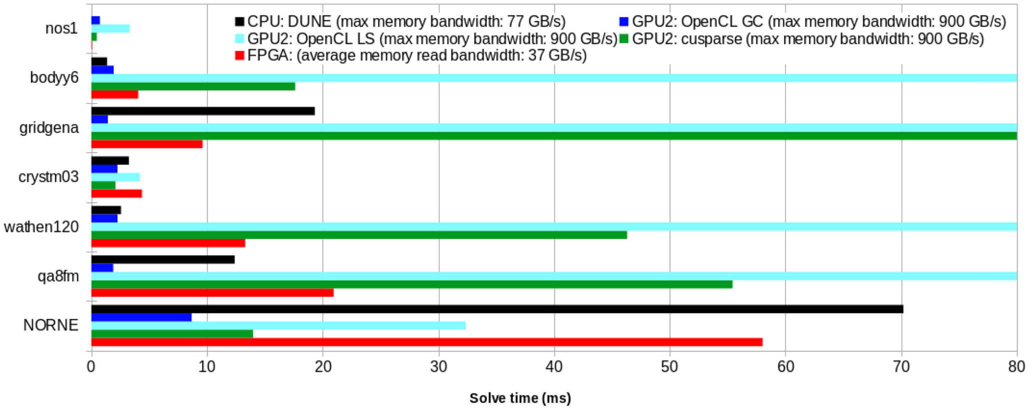


Fig. 13. Solver execution time for selected matrices. For readability of the CPU/FPGA results, some GPU results have been clipped. Refer to Table 9 for the actual values. Please note that for the FPGA, the maximum aggregated HBM is 460 GB/s and the maximum aggregated DDR4 memory bandwidth is around 37 GB/s; however, due to design limitations caused by implementation issues, the maximum bandwidth used by the kernel is around 52 GB/s.

Table 9. Timing Results (in Milliseconds) for Solving Selected Matrices

Matrix	FPGA				CPU DUNE		GPU[2]								
							cusparse			OpenCL LS			OpenCL GC		
	Solver	Host	Copy	Iter#	Host	Iter#	Host	Copy	Iter#	Host	Copy	Iter#	Host	Copy	Iter#
nos1	0.1	0.4	0.4	1.0	0.0	0.5	0.5	0.0	0.5	3.3	0.1	1.0	0.7	0.1	1.5
bodyy6	4.1	4.3	3.2	2.0	1.4	0.5	17.6	0.5	0.5	136.5	0.5	1.0	1.9	0.5	2.0
gridgena	9.6	10.2	12.4	1.5	19.3	4.5	528.7	1.2	4.5	1007.7	1.3	2.5	1.4	1.3	1.5
crystm03	4.4	4.9	9.4	0.5	3.2	0.5	2.1	1.6	0.5	4.2	1.7	0.5	2.3	1.7	1.0
wathen120	13.3	13.5	12.1	2.0	2.6	0.5	46.3	1.1	0.5	334.3	1.3	1.0	2.3	1.3	2.0
qa8fm	20.9	21.2	30.6	1.0	12.4	0.5	55.4	3.4	0.5	179.6	3.6	0.5	1.9	3.7	1.0
NORNE	58.0	58.6	27.3	4.5	70.2	4.5	14.0	2.7	4.5	32.4	3.2	4.5	8.7	3.0	6.5

7.3 Flow Results

The ILU0 BiCGStab solvers were integrated into the flow simulator in OPM [19] by replacing the call to the simulator's own iterative solver with calls to a function that does the required preprocessing, sends the resulting data to the accelerator, and then reads the results from it after it is done. The source code used was from version 2020.10-rc4, with custom modifications to integrate our solvers. The original flow solver is an ILU0 BiCGStab solver based on the DUNE project [9], and in Table 12, the performance results between this software solver and our solvers are compared when running simulation on the NORNE testcase (file *NORNE_ATW2013.DATA*) [22] that is part of the OPM project. For all runs, the parameter "–matrix-add-wellcontributions=true" was added to the command line (to have a fair comparison with the FPGA results, because the FPGA solver currently does not support separated well contributions). For the NORNE case, including well contributions will increase runtime on the CPU by approximately 20% compared to the default configuration. Moreover, for the FPGA run, the parameter "–threads-per-process=8" was also added. This will only impact the assembly part of the simulation, not the preconditioner nor the linear solver, in effect reducing the computational time spent outside the preconditioning and linear solve part. The reordering algorithm has been left to the default for FPGA (LS), because GC produces in this case more colors than the ones supported by the FPGA solver. For the executions

Table 10. Read and Write Ports Bandwidth Utilization of the FPGA Solver, as % of the Available Bandwidth

Matrix	DDR4		HBM						Aggr. read BW	% of max aggr.
	Read 0	Read 1	Read 2	Read 3	Read 4	Write 0	Write 1	Write 2	(GB/s)	read BW
nos1	42.8	14.9	50.6	50.5	49.5	79.4	79.4	79.1	32.0	40.2
bodyy6	82.6	36.8	70.8	64.4	72.4	100	100	100	51.6	64.7
gridgena	81.7	36.2	70.0	65.8	70.2	100	100	100	51.1	64.1
crystm03	79.8	43.2	68.3	69.2	69.3	100	100	100	52.2	65.4
wathen120	80.6	35.6	68.9	67.0	70.0	100	100	100	50.7	63.7
qa8fm	81.4	36.0	68.6	66.5	72.1	100	100	100	51.1	64.2
NORNE	83.0	35.8	66.8	62.5	69.6	100	100	100	50.3	63.1

Table 11. Comparison Between Jacobi and ILU0 as the Preconditioner on the V100

Preconditioner	Tolerance	Maxiter	Total Runtime (s)	Linear Solves	Linear Iterations	No. of Solves Failed
Jacobi	1e-2	200	592	2,036	127,768	125
	1e-3	300	511	1,535	159,537	55
ilu0	1e-2	200	432	1,485	21,167	0
	1e-3	300	415	1,257	29,412	0

Failed linear solves did not reduce the error enough in “maxiter” iterations, and these solves are redone on the CPU (DUNE), with default settings (ilu0, 1e-2, max 200 iterations).

using the FPGA solver, all the application’s threads were pinned to CPU #0 (i.e., the one directly connected to the FPGA) to avoid fluctuations in bandwidth during the transfers between the host and the FPGA memory.

The default reduction of $1e-2$ was used for the results shown in Table 12 and Figure 13, whereas a reduction of $1e-6$ was used for the results shown in Table 13. Please note that the reordering of the matrix for the FPGA solver has been included in the create ILU0 time, inflating the time this pre-processing step costs compared to the other platforms. For an overview of the complete application and how the different computational blocks are mapped to the various accelerators, see Figure 10. For an explanation of the characteristics of each platform used to run flow, please refer to Table 2.

To gain a better understanding of which parts of running the FPGA solver are causing it to be slower than the software version, we timed the different functions applied by this solver during a flow run, the results of which are shown in Table 12. During the running of the flow simulator, the sparsity pattern of the matrix that needs to be solved does not change. This makes it possible for functions that depend only on the sparsity pattern, like the finding of the reordering and partitioning pattern, to be done only once during initialization. From Table 14, we observe that the majority of solver time is actually spent running the FPGA solver, but a significant portion of the runtime is also taken up by the creation of the preconditioner, and the data transfers between the host and FPGA.

7.4 Discussion

In this section, we analyze all results that we presented so far, explaining the discrepancies in the performance between the FPGA and the GPU numbers, and we discuss about the potential of the FPGA in this domain based on the observed results and the potential performance gain when looking at the hardware model. Nevertheless, we would like to note that the GPU results

Table 12. Comparison of Performance for Flow with Reduction = 1e-2(default) Using CPU, FPGA, and GPU (in Seconds)

Function	CPU	FPGA	GPU^1			GPU^2		
			cus	OCL LS	OCL GC	cus	OCL LS	OCL GC
Total time	540.6	751.8	489.7	799.6	749.3	320.1	507.6	458.5
Solver time	540.6	751.7	489.6	799.5	749.2	320.0	507.6	458.5
Assembly time	99.4	94.6	133.7	137.5	163.6	115.6	112.7	114.7
Well assembly time	43.0	47.1	21.15	21.6	26.5	13.9	14.6	16.8
Linear solve time	339.5	559.9	248.9	551.4	460.6	117.6	293.6	207.8
Linear setup time	41.8	45.3	38.1	39.5	47.7	33.6	35.4	43.6
Update time	57.3	54.4	67.3	70.1	84.5	53.6	56.9	71.2
Output write time	3.5	3.5	3.2	3.2	3.2	2.9	2.9	3.0
Number of linear solves	1,458	1,441	1,437	1,507	1,818	1,449	1,507	1,861
Number of linear iterations	21,991	21,012	21,282	22,626	42,160	21,020	22,626	40,991
Speedup total time	1.00	0.72	1.10	0.68	0.72	1.69	1.07	1.18
Speedup linear solver time	1.00	0.61	1.36	0.62	0.74	2.89	1.16	1.63

Table 13. Comparison of Performance for Flow with Reduction = 1e-6 Using CPU, FPGA, and GPU (in Seconds)

Function	CPU	FPGA	GPU^1			GPU^2		
			cus	OCL LS	OCL GC	cus	OCL LS	OCL GC
Total time	1,254.1	1,512.9	900.0	1,893.0	1,333.3	484.5	950.5	579.4
Solver time	1,254.0	1,512.9	900.0	1,893.0	1,333.2	484.5	950.5	579.3
Assembly time	82.2	84.9	112.0	114.7	115.9	101.8	101.7	101.7
Well assembly time	36.1	42.3	17.1	17.5	17.6	11.9	12.1	12.5
Linear solve time	1,081.3	1,333.8	691.4	1,680.1	1,119.5	303.8	768.8	397.0
Linear setup time	35.2	38.9	31.1	31.4	31.3	27.8	28.2	28.7
Update time	46.1	50.9	56.9	57.4	57.3	45.6	45.6	45.8
Output write time	3.5	3.5	3.0	3.2	3.2	2.8	2.9	2.9
Number of linear solves	1,207	1,207	1,202	1,207	1,207	1,207	1,207	1,207
Number of linear iterations	79,163	78,275	78,447	82,673	117,463	78,958	82,673	117,294
Speedup total time	1.00	0.83	1.39	0.66	0.94	2.59	1.32	2.16
Speedup linear solver time	1.00	0.81	1.56	0.64	0.97	3.56	1.41	2.72

in Figures 12 and 13 are shown only as a reference point to give a glimpse of the competitiveness of the current FPGA implementation and to enable benchmarking of follow-up works. However, we refrain from providing an in-depth analysis of the differences between the two competing architectures because the FPGA design was not scaled to its fullest potential, which remains a future work.

7.4.1 GPU Results. To fulfill the high-precision requirements for our target application domain, we performed most of our experiments with double precision, except for the SpMV, for which we present also single-precision results to facilitate easy comparison with future related works. Due to the well-known capability of GPUs to perform double-precision computations, it is not surprising that the GPU performed better, as evidenced by the stand-alone results presented in Figures 12 and 13. Based on those figures, we make two important observations.

First, the V100 GPU packs more resources than the Alveo U280 FPGA, with its nearly 2× increased HBM2 bandwidth, an abundance of hard floating point (double) precision cores, and a superior clock rate, which clearly gives it an edge in this application domain. Therefore, the

Table 14. Breakdown of Linear Solve Time (in Seconds) Spent Running the Solvers in Flow with Reduction = 1e-2 (Default)

Device	Step (accumulated time)	CPU	FPGA	GPU^1			GPU^2		
				cus	OCL LS	OCL GC	cus	OCL LS	OCL GC
Host	Reorder vectors	–	1.10	–	1.08	1.14	–	1.14	1.30
	Create ILU0	38.5	128.75	22.27	60.09	95.35	5.76	58.00	81.00
	BILU0 reorder	–	–	–	20.47	27.72	–	21.30	23.60
	BILU0 decomp.	–	–	–	31.42	57.34	–	30.20	49.60
	BILU0 copy to GPU	–	–	–	5.04	5.97	–	3.79	4.63
	Memory setup	–	10.61	–	–	–	–	–	–
	CPU to accelerator	–	65.95	4.70	5.10	6.18	3.83	4.46	5.59
Accel.	Total solve	274.1	304.90	175.10	448.50	324.10	71.80	197.30	88.10
	ILU apply	112.6	–	134.50	395.10	68.00	63.70	182.30	62.30
	SpMV	127.3	–	28.40	37.30	25.50	2.60	4.40	7.70
	Rest	28.9	–	8.50	12.70	223.80	4.00	8.30	14.20
Host	Accelerator to CPU	–	0.76	0.27	0.35	0.16	0.26	0.34	0.16

Table 15. Some Profiled Metrics of GPU Kernels

Metric	Solve L	Solve U	spmv
SM utilization (%)	31.1	29.1	53.9
Bandwidth utilization (%)	13.4	14.1	46.0
Memory throughput (GB/s)	20.5	19.2	285.2
Eligible warps per scheduler	0.18	0.18	0.48
Achieved occupancy (%)	76.0	77.3	92.6

performance advantage is clearly justified when considering the stand-alone benchmarking of the SpMV in Figure 12 for which the GPU seems to outperform the FPGA by 10× for our benchmark of interest, NORNE. Please note that the same ratio is seen for the solver as well in Figure 13; however, in that case, the run is only for the first NORNE matrix in the complete reservoir, which for the FPGA-based accelerator leads to 1,441 different matrices, as highlighted by the "Number of Linear Solves" calls seen in Table 12.

This leads us to the second observation we make, namely that over the course of running the complete reservoir simulator, the performance gap between FPGA and GPU drops to only 3 to 4×. This can be attributed to the complex and deep pipeline of the solver that over the course of running the complete flow starts to perform better overall than a single solver instance. This is visible in the same Table 12. Therefore, it can be expected that the FPGA can equal and even surpass the performance of the GPU when using the same amount of resources.

7.4.2 Performance Potential. To explain this performance claim, we profiled the GPU using Nsight Compute [1] during a solve on NORNE and obtained the utilization for the three most important kernels for the cusparse version of the solver. This is shown in Table 15. Although the GPU has an abundance of resources, out of those, only a finite number of them are used, which shows the GPU is not utilized efficiently. The aggregated BW utilization for the GPU is 20% (weighted average of the L, U, and spmv kernels, where L and U have each a 40% weight in the complete solver, and spmv only 20%). This would lead to the conclusion that only 180GB/s is used by the GPU. This is roughly the 3 to 4× difference when compared to the FPGA, which deploys efficiently only 50 GB/s of bandwidth in the current design. However, we can see that the FPGA is able to utilize the bandwidth much more efficiently, at 63.1% for the NORNE use case. Therefore, we extrapolate that using the same amount of resources, the FPGA could equal or outperform the performance of the GPU, a fact that can be seen in the modeling figures that predict a 6× improvement versus software.

8 CONCLUSION

In this article, we have evaluated the potential of using **FPGA**s in HPC, which is a highly relevant topic because of the rapid advances in reconfigurable hardware. To perform this study, we began by proposing a novel CSR-based encoding to optimize a new SpMV kernel on **FPGA**, which can be easily integrated with an ILU0 preconditioner. We subsequently developed a hardware model to predict and guide the design of the ILU0 preconditioned BiCGStab solver, which helped us to understand the trade-offs between area and performance when scaling the resources. Next, we implemented the ILU0-BiCGStab preconditioned solver targeted at an HBM-enabled **FPGA** using handwritten RTL to maximize the performance of the double-precision kernel. To validate our work, we integrated the complete solver in the OPM reservoir simulator, both for **FPGA** and GPU. Finally, we provided extensive evaluation results for both the stand-alone SpMV and solver kernels as well as the complete reservoir simulation execution on a real-world use case running on three different platforms: CPU, **FPGA**, and **GPU**.

We find that the **FPGA** is on par with the CPU execution and 3× slower than the **GPU** implementation when comparing only the kernel executions. Although the obtained results show that an **FPGA** is not yet competitive with the **GPU**, we believe that with a better on-chip support for double precision, increased number of computation units and memory ports, decreasing the latency of the ILU0 and **SpMV** units by separating and specializing them further, as well as other optimizations on the host software side (i.e., reduction of preconditioner computation time, increased efficiency of the host-FPGA data exchange and task execution control), the **FPGA** can become an alternative to speed up scientific computing. This has been illustrated by scaling our hardware performance model beyond what it was possible to achieve in this work due to the encountered hardware limitations. The main reason an **FPGA** would be able to outperform a **GPU** is because of its superior utilization of resources that better match irregular applications with long and complex computational pipelines, a topic that was discussed in Section 7.4. In the future, we will work to eliminate these restrictions and provide a more efficient implementation. Alternative research could focus on developing other preconditioned solvers that might be more suitable for an **FPGA** implementation.

Finally, the source codes for the RTL kernels and the integration with *OPM Flow* are made available in the Git repositories of the OPM project [19]. Specifically, the software modifications have been added to the *opm-simulators* repository [20], whereas the FPGA-specific RTL code and implementation scripts have been added to the *FPGA* repository [21]. A *pull request* was created (#2998 in the *opm-simulators* repository [20] that has been merged in the main repository code), which can be used, along with the FPGA code, to reproduce the results of this work. Please note that since the *pull request* was created against the most current version of the code at the time of writing (December 2020), the performance results obtained with that code may not be exactly the same as the results reported in this article, which were gathered with a previous release of OPM *Flow* as stated in Section 7.3.

REFERENCES

[1] NVIDIA. n.d. NVIDIA Nsight Compute Command Line Interface. Retrieved November 3, 2021 from https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[2] AMD. 2020. AMD to Acquire Xilinx. Retrieved November 3, 2021 from https://www.amd.com/en/press-releases/2020-10-27-amd-to-acquire-xilinx-creating-the-industry-s-high-performance-computing.

[3] J. R. Appleyard. 1983. Nested factorization. In *Proceedings of the SPE Reservoir Simulation Symposium.*

[4] Peter Bastian, Markus Blatt, Andreas Dedner, Nils-Arne Dreier, Christian Engwer, Rene Fritze, Carsten Graser, et al. 2021. The Dune framework: Basic concepts and recent developments. *Computers & Mathematics with Applications* 81 (2021), 75–112. https://doi.org/10.1016/j.camwa.2020.06.007

[5] Edmond Chow and Aftab Patel. 2015. Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing* 37, 2 (2015), C169–C193. https://doi.org/10.1137/140968896 arXiv:https://doi.org/10.1137/140968896

[6] 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 International Conference on Field Programmable Logic and Applications (FPL'14).* IEEE, Los Alamitos, CA, 1–7.

[7] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* 38, 1 (Dec. 2011), Article 1, 25 pages. https://doi.org/10.1145/2049662.2049663

[8] Richard Dorrance, Fengbo Ren, and Dejan Marković. 2014. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, New York, NY, 161–170.

[9] Dune. 2020. Dune Project. Retrieved November 3, 2021 from https://dune-project.org/.

[10] R. Eberhardt and M. Hoemmen. 2016. Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'16).* 663–672. https://doi.org/10.1109/IPDPSW.2016.42

[11] Satish Balay. 2020. *PETSc Users Manual.* Technical Report ANL-95/11—Revision 3.14. Argonne National Laboratory. https://www.mcs.anl.gov/petsc.

[12] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. 2014. A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication. In *Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines.* IEEE, Los Alamitos, CA, 36–43.

[13] Intel. 2015. Intel Acquisition of Altera. Retrieved November 3, 2021 from https://newsroom.intel.com/press-kits/intel-acquisition-of-altera.

[14] Mark T. Jones and Paul E. Plassmann. 1993. A parallel graph coloring heuristic. *SIAM Journal of Scientific Computing* 14-3 (1993), 654–669.

[15] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers*, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Springer, Berlin, Germany, 111–125.

[16] Gerald R. Morris, Viktor K. Prasanna, and Richard D. Anderson. 2006. A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer. In *Proceedings of the 2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* IEEE, Los Alamitos, CA, 3–12.

[17] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. 2011. *OpenCL Programming Guide.* Addison-Wesley Professional.

[18] NVIDIA. 2020. The API Reference Guide for cuSPARSE, the CUDA Sparse Matrix Library. Retrieved November 3, 2021 from https://docs.nvidia.com/cuda/cusparse/index.html.

[19] OPM. 2020. Open Porous Media Project. Retrieved November 3, 2021 from https://github.com/OPM.

[20] OPM. 2020. Open Porous Media Reservoir Simulator. Retrieved November 3, 2021 from https://github.com/OPM/opm-simulators.

[21] OPM. 2020. Open Porous Media Reservoir Simulator—FPGA Kernels. Retrieved November 3, 2021 from https://github.com/OPM/FPGA.

[22] OPM. 2020. OPM Tests. Retrieved November 3, 2021 from https://github.com/OPM/opm-tests/tree/master/norne.

[23] Alberto Parravicini, Luca Giuseppe Cellamare, Marco Siracusa, and Marco Domenico Santambrogio. 2021. Scaling up HBM efficiency of top-K SpMV for approximate embedding similarity on FPGAs. *CoRR* abs/2103.04808 (2021). arXiv:2103.04808 https://arxiv.org/abs/2103.04808.

[24] Michail Pligouroudis, Rafael Angel Gutierrez Nuno, and Tom Kazmierski. 2020. Modified compressed sparse row format for accelerated FPGA-based sparse matrix multiplication. In *Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS'20).* 1–5. https://doi.org/10.1109/ISCAS45731.2020.9181266

[25] Atgeirr Flø Rasmussen, Tor Harald Sandve, Kai Bao, Andreas Lauser, Joakim Hove, Bård Skaflestad, Robert Klöfkorn, et al. 2021. The Open Porous Media Flow reservoir simulator. *Computers & Mathematics with Applications* 81 (2021), 159–185. https://doi.org/10.1016/j.camwa.2020.05.014

[26] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics.

[27] Björn Sigurbergsson, Tom Hogervorst, Tong D. Qiu, and Razvan Nane. 2019. Sparstition: A partitioning scheme for large-scale sparse matrix vector multiplication on FPGA. In *Proceedings of the I nternational Conference on Application-Specific Systems, Architectures, and Processors (ASAP'19).* 51–58.

[28] Hamdi Tchelepi and Yifan Zhou. 2013. Multi-GPU parallelization of nested factorization for solving large linear systems. In *Proceedings of the SPE Reservoir Simulation Symposium*.

[29] Erdem Topsakal, Rick Kindt, Kubilay Sertel, and John Volakis. 2001. Evaluation of the BICGSTAB (l) algorithm for the finite-element/boundary-integral method. *IEEE Antennas and Propagation Magazine* 43, 6 (2001), 124–131.

[30] Yaman Umuroğlu. 2018. *Accelerating Sparse Linear Algebra and Deep Neural Networks on Reconfigurable Platforms*. Ph.D. Dissertation. NTNU.

[31] H. A. van der Vorst. 1992. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 13, 2 (1992), 631–644. https://doi.org/10.1137/0913035 arXiv:https://doi.org/10.1137/0913035

[32] 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.

[33] Xilinx. 2020. Alveo U280 Data Center Accelerator Card. Retrieved November 3, 2021 from https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[34] Xilinx. 2020. Ultra RAM. Retrieved November 3, 2021 from https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf.

[35] Song Yu, Hui Liu, Zhangxin John Chen, Ben Hsieh, and Lei Shao 2012. GPU-based parallel reservoir simulation for large-scale simulation problems. In *Proceedings of the SPE Europec/EAGE Annual Conference*.

[36] Ling Zhuo and Viktor K. Prasanna. 2005. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, 63–74.

Received January 2021; revised May 2021; accepted July 2021