Data Management Model to Program Irregular Compute Kernels on FPGA: Application to Heterogeneous Distributed System

Erwan Lenormand[1(✉)], Thierry Goubier[1], Loïc Cudennec[2], and Henri-Pierre Charles[3]

[1] Université Paris-Saclay, CEA, LIST, 91191 Gif-sur-Yvette, France
{erwan.lenormand,thierry.goubier}@cea.fr
[2] DGA Maîtrise de l'Information, BP 7, 35998 Rennes, France
loic.cudennec@intradef.gouv.fr
[3] Univ Grenoble-Alpes, CEA, LIST, 38000 Grenoble, France
henri-pierre.charles@cea.fr

Abstract. This paper presents a data management model targeting heterogeneous distributed systems integrating reconfigurable accelerators. The purpose of this model is to reduce the complexity of developing applications with multidimensional sparse data structures. It relies on a shared memory paradigm, which is convenient for parallel programming of irregular applications. The distributed data, sliced in chunks, are managed by a Software-Distributed Shared Memory (S-DSM). The integration of reconfigurable accelerators in this S-DSM, by breaking the master-slave model, allows devices to initiate access to chunks in order to accept data-dependent accesses. We use chunk partitioning of multidimensional sparse data structures, such as sparse matrices and unstructured meshes, to access them as a continuous data stream. This model enables to regularize memory accesses of irregular applications, to avoid the transfer of unnecessary data by providing fine-grained data access, and to efficiently hide data access latencies by implicitly overlaying the transferred data flow with the processed data flow.

We have used two case studies to validate the proposed data management model: General Sparse Matrix-Matrix Multiplication (SpGEMM) and Shallow Water Equations (SWE) over an unstructured mesh. The results obtained show that the proposed model efficiently hides the data access latencies by reaching computation speeds close to those of an ideal case (i.e. without latency).

Keywords: Distributed shared memory · Field programmable gate array · Irregular application

This work was supported by the LEXIS project, funded by the EU's Horizon 2020 research and innovation programme (2014–2020) under grant agreement no. 825532.

R. Chaves et al. (Eds.): Euro-Par 2021, LNCS 13098, pp. 91–103, 2022.
https://doi.org/10.1007/978-3-031-06156-1_8

1 Introduction

As a response to the power wall problem, High Performance Computing (HPC) systems heterogeneity is gradually increasing. Part of this heterogeneity comes from the association of processors with co-processors, mainly GPUs. These latter allow the execution of computational-intensive portions of applications, called compute kernels, with high FLOP/W efficiency. However, as illustrated by the efficiency on the High Performance Conjugate Gradient (HPCG) benchmark [11], these systems achieve only a fraction of their theoretical peak performance and do not show efficiency gains from their heterogeneity for irregular applications. This poor performance is due on one hand, to the random data access patterns generated by these applications, and on the other hand, to the complexity of porting irregular compute kernels to GPUs. Thanks to their reconfigurable architecture, Field-programmable gate arrays (FPGAs) are particularly suitable for processing irregular compute kernels [7]. The attractiveness of FPGAs for HPC systems is growing by means of their increasing computing power and the improvement of High Level Synthesis (HLS) tools. However, porting irregular compute kernels to FPGA remains a challenging task, because random data access patterns limit the abilities of HLS tools. Thus, designers must deal with low-level kernel design, optimization of data structures for FPGA memory systems and orchestration of distributed data transfers.

To address this issue, we propose a data management model for irregular compute kernels targeting heterogeneous distributed systems with reconfigurable accelerators. The latter is based on a shared-memory provided by a Software-Distributed Shared Memory (S-DSM). The application datasets are sliced in *chunks* managed by the S-DSM. The integration of reconfigurable accelerators in the S-DSM allows devices to initiate accesses to *chunks*. In this way, all the processing units can make fine-grained random data accesses. This unified data access model simplifies programming and meets the needs of irregular applications. By abstracting the data structure, *chunk* partitioning enables to prefetch the data as streams of *chunks*. This prefetching should make it possible to hide high data access latencies by implicitly overlaying the transferred data flow with the processed data flow. The efficiency of the proposed data management model relies on its ability to hide latencies. To assess this efficiency, we have used two case studies: General Sparse Matrix-Matrix Multiplication (SpGEMM) and a tsunami simulation code. These two applications generate a lot of irregular memory accesses, which are complex to optimize because they are data-dependent.

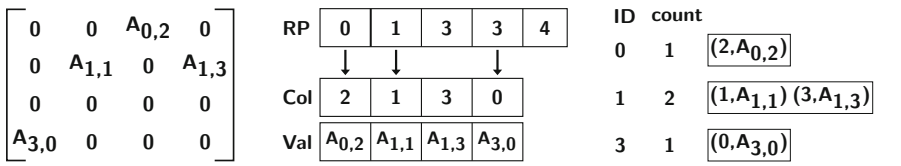
The paper is organized as follows: Sect. 2 presents the data management model, Sect. 3 describes the experiments conducted to validate the model, Sect. 4 gives some references on related work, finally, Sect. 5 concludes this paper.

2 Data Management Model

Shared memory is a convenient programming paradigm to develop multi-threaded applications, which randomly access data. Software-Distributed Shared

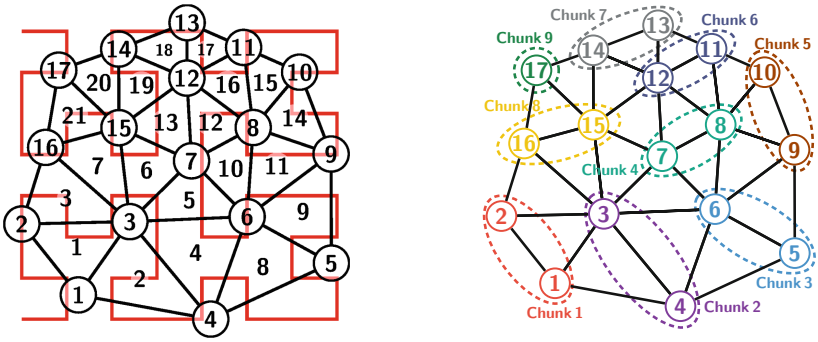
Memory can be used to aggregate distributed physical memories into a shared logical space. In this work, we consider a S-DSM for heterogeneous micro-server that has been proposed in previous work [5]. The latter is organized as a semi-structured super-peer network, where a set of clients are connected to a peer-to-peer network of servers. Clients execute the user code and servers manage the shared data and related metadata. The integration of reconfigurable accelerators in the S-DSM enables compute kernels to initiate access to distributed data. Obviously, this way to access the data can lead to high access latencies. To deal with this problem, the data management model aims to hide data access latencies by overlaying the transferred data flow with the processed data flow. This relies on the ability to access data as continuous streams. To do this we use *chunks*, a common object in computer science, whose concept is to use metadata to describe the data stored in it. *Chunks* are the atomic piece of data managed by the S-DSM. Each one has an unique identifier (*chunk ID*) and their maximum size can be set by the application. We use them to represent irregular data structures, as they are convenient objects for data management in distributed systems and their metadata allow to abstract the stored data. From the point of view of the compute kernels, the role of the S-DSM is to transparently provide the data and metadata corresponding to *chunk ID*. By partitioning the data structures according to the access granularity of the applications, data streams can be generated from sequences of *chunk ID*. Adapting the size of *chunks* to the granularity of access allows to avoid the transfer of unnecessary data. We have chosen two data structures widely used in irregular applications to illustrate the data management model: sparse matrix and unstructured meshes.

Sparse Linear Algebra consists in performing linear operations on matrices (or vectors) for which the majority of the elements are equal to zero. Sparse matrices are compressed to reduce their memory footprint and to accelerate access to their nonzero elements. The compressed sparse row format (CSR), shown in Fig. 1b, is one of the most used sparse matrix representations. The column indices and the values of elements are stored in row-major order in the arrays *Col* and *Val*. $RP[i]$ indicates the position of the first element of row i in the arrays and the operation $RP[i+1]-RP[i]$ is equal to the number of elements in the row. As shown in Fig. 1c, we have adapted the CSR format to the use of *chunks*. We colocalize the value and the column index of an element to form a pair. The set of pairs representing a row is stored in a *chunk*. Then we use *chunks*



(a) Dense format. **(b)** CSR format. **(c)** Chunk-based CSR format

Fig. 1. Matrix representation.



(a) Hilbert space-filling curve over the mesh.

(b) Chunk partitioning of the mesh with chunks of two elements.

Fig. 2. Reordering and partitioning of a 2D unstructured mesh.

metadata to indicate the number of elements per row. This structure reduces the number of memory accesses required to read or write a matrix row. It can be easily adapted to another compressed format (e.g. compressed sparse column format). Reading or writing a matrix involves to request access to each row and to request the transfer of rows data between the memory and the compute kernel. Decoupling the access request and the transfer request allows the prefetching of the data into the FPGA memory and thus hides the access latency. Considering that the kernel is developed as a pipeline of stages, which are separated by FIFOs, then the prefetching speed is implicitly limited by the size of the FIFOs. This prevents the FPGA memory from being overloaded due to too early data prefetching. In an ideal case prefetching speed corresponds to the speed of data consumption of the following stages in the pipeline.

Many HPC applications of Finite Element Method (FEM) work on unstructured meshes with triangular elements in 2D and tetrahedral elements in 3D. Typical kernels on such unstructured meshes proceed by mesh updates - updating all elements, nodes or edges of the mesh according to a function of neighbourhood values, applying a convolution or stencil and hence following indirections to both iterate over the mesh and to access neighbourhood information. Consequently, the topology of the mesh and indexing of data has a significant impact on data access locality and therefore application performance. Space-filling curves (SFC) allow to improve data access locality of the mesh [2]. We use this technique to do an efficient chunk partitioning of the mesh. As shown in Fig. 2a, a SFC is drawn in the geometric space of the mesh. Vertices are indexed according the order in which they meet the curve. As illustrated in Fig. 2b, we apply a basic partitioning along the curve, which consists in grouping the values of nodes of consecutive indices in *chunks* of constant size. The elements are numbered in order of the smallest index of their vertices. By following the path of the curve, most of the data of the mesh could be accessed through a sliding window, whose size would not be dependent of the mesh size. Thus, traversal of the mesh would

be done through a continuous flow of data, where the majority of *chunks* would be accessed only once. Only the data of the elements located at the junction zones between the different spaces of the curve would not be accessible through the window. By following the curve, it is possible to identify the corresponding *chunks*. In this way, a *chunk ID* sequence corresponding to these data can be generated. We use these observations to design kernels iterating over unstructured meshes. The sliding window is implemented with an addressable FIFO. Buffers are used to access data not accessible through the sliding window.

3 Data Management Model Validation

To validate the proposed data management model and assess its ability to hide data access latencies, we have conducted experiments with a simulation tool. This tool makes it possible to evaluate the performance of the system from high level modeling without requiring a full FPGA synthesis. The experiments focused on sparse matrix-matrix multiplication and a tsunami simulation code.

3.1 Simulation Methodology

To conduct the experiments, we have chosen to use a simulation tool that we have developed [12]. The objective was to evaluate the performance of the system from a high-level modeling. The behavioral description of the kernels is modeled in *C++*. The irregularity of the applications we are studying and the distributed nature of the system we are targeting imply high and variable data access latencies. Thus, the main objective of this tool is to evaluate the effects of latency on the ability to speed up compute kernels using our data management model. Performance evaluation is based on the generation of data access latencies relating to the activity of the compute kernel. The tool uses a hybrid method: the activity of the compute kernel is generated by a simulation engine and latencies are produced by measuring the real latencies of S-DSM requests executed on the physical architecture, in order to produce faithful latencies. The simulation engine and the S-DSM server can be run on different nodes. This makes it possible to study different topologies associated with different latency profiles. In the rest of the section three topologies are used: *No Latency* which corresponds to the ideal case where all the data is stored in the FPGA memory, *Local* which corresponds to the case where the FPGA is connected by a local bus to the node running the S-DSM server and storing the data, and *Remote* which corresponds to the case where the FPGA and the S-DSM server are on two different nodes and are connected through an Ethernet network. Local node latencies are medium (383 µs for a read request and 207 µs for a write request) and remote node latencies are high (1311 µs for read and 533 µs for write). We have used a Xilinx Virtex VC707 as a reference FPGA to set up the simulation engine. Thus, the clock frequency was set to 200 MHz and the theoretical peak memory bandwidth between the DDR and compute kernels was 12.8 Gb/s. The simulation being non-deterministic, the results presented are the median values of 10 runs.

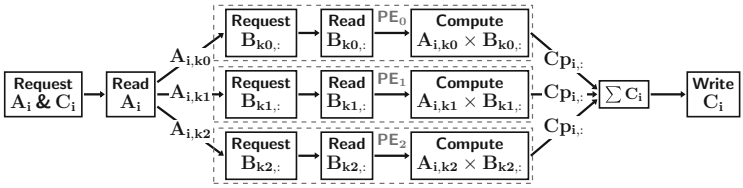


Fig. 3. Dataflow of the SpGEMM compute kernel using 3 PEs.

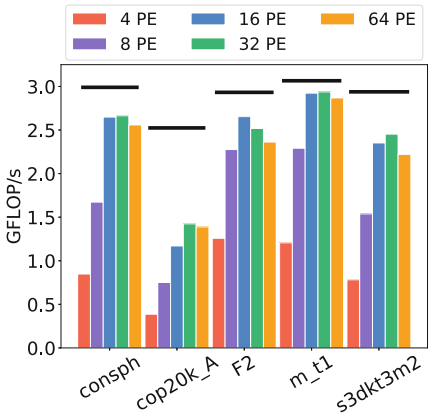
3.2 Case Study 1: General Sparse Matrix-Matrix Multiplication

SpGEMM is widely used to study acceleration methods for sparse linear algebra. This application generates irregular memory access patterns that makes it complex to optimize, with usually a low efficiency in terms of floating point operations per unit of time. We have designed the compute kernel by using the row-wise sparse matrix-matrix multiplication algorithm formulated by Gustavson [10]. Thanks to the row-wise traversal of the matrices, this algorithm is well suited to dataflow processing and is quite straightforward to parallelize. As illustrated in Fig. 3, to parallelize the computations, the kernel is implemented with several processing elements (PEs). The first stages of the kernel access the nonzero elements of the first input matrix and distribute them to the PEs. Each PE multiplies the elements received by the corresponding rows of the second input matrix. Finally, the last stages sum the partial results computed by the PEs and write the result matrix. The indices and the values of the matrix are encoded with 4 bytes (Single precision computations).

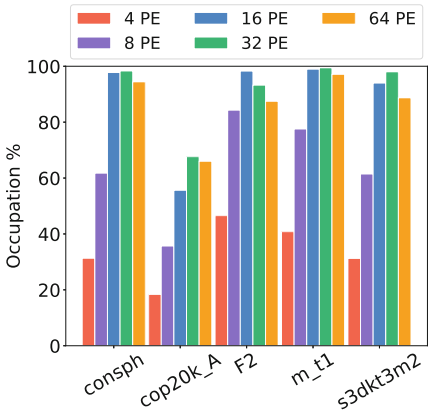
As the arithmetic intensity of SpGEMM is strongly data-dependent, we have chosen matrices, presented in Table 1, with varying sizes, densities and patterns. Thus, the experiments allow to evaluate the capacity of the data management model to adapt to irregularity. In order to limit the simulation time, the memory footprints of the matrices are smaller than a FPGA DRAM. To reproduce a situation where the capacity of the accelerator memory requires to transfer the data during the execution, we have adapted the simulated memory capacity accordingly to the dataset. Thus, the accelerator memory has been configured with 65536 locations of 1 kib (64 Mib). For each matrix, we have defined the theoretical peak computation speed by considering the processing time as the size of data transferred between the memory and the compute kernel divided by

Table 1. Square matrices, from [6], used for simulations. NNZ and density refer to the source matrix. The memory footprint includes the three operand matrices.

Name	Row	NNZ	Density (%)	Memory footprint	Peak GFLOP/s
consph	83334	6010480	0.087	294 Mb	2.99
cop20k_A	121192	2624331	0.018	182 Mb	2.52
F2	71505	5294285	0.10	383 Mb	2.93
m_t1	97578	9753570	0.10	427 Mb	3.07
s3dkt3m2	90449	3753461	0.046	134 Mb	2.94



(a) Computation speed in GFLOP/s (higher is better). The horizontal lines are the theoretical peak computation speeds.



(b) Memory controller activity (occupancy percentage). Close to 100% means saturation.

Fig. 4. Performance according to the number of processing elements (PEs).

the FPGA memory bandwidth. These are represented by the horizontal black bars on Fig. 4a and Fig. 5.

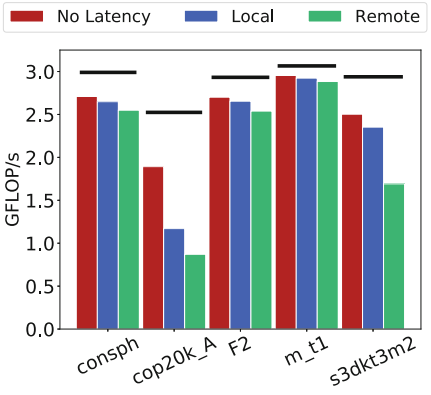


Fig. 5. Computation speed in GFLOP/s according the system topology (higher is better). The horizontal lines are the theoretical peak computation speeds.

For the first experiment, we varied the parallelism level of the compute kernel by implementing between 4 and 64 PEs on the local node. Figure 4a shows the computation speed obtained for this experiment. This shows that the increase in parallelism makes it possible to speed up computations, up to 16 PEs. The speed up obtained between 16 PEs and 32 PEs is low (between 1.04 and 1.22) or even negative. Between 32 and 64 PEs the speed ud is always negative. This efficiency limitation means that the PEs are under-exploited due to an insufficient supply of data (data starvation). The latter can be explained either by a data starvation in FPGA memory (due to excessive latencies), or by a FPGA memory bandwidth bottleneck. Figure 4b illustrates the occupancy rate of the FPGA memory controller. These results show that the controller is saturated for the configuration with 16 PEs. This information highlights that the FPGA memory bandwidth is the bottleneck for this kernel. This bandwidth limit is also one of the explanations for the nonlinear speed up between the configurations with 4 PEs and 16 PEs. The second experiment aimed to study the impact of topology on

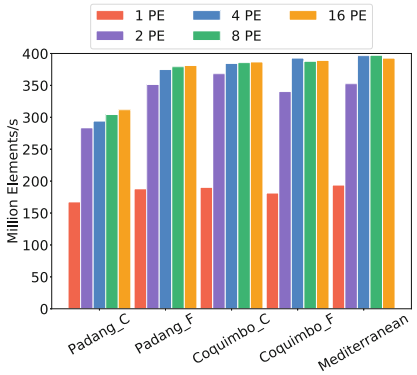
performance. For this we have used a configuration with 16 PEs, able to saturate the memory controller on the local node. The results obtained are illustrated in Fig. 5. It shows that for the matrices *consph*, *F2* and *m_t1* the performance gap with the ideal case for the local node (between 1% and 2%) and the remote node (between 2% and 6%) is very low. This small performance gap is mainly explained by the time to load the first data required to reach the nominal mode of the kernel. This shows the ability of the data management model to hide data access latencies. For the matrices *cop20k_A* and *s3dkt3m2*, the performance gap is larger, but remains relatively small, respectively 38% and 6% for local node and 54% and 32% for remote node. For these two latter, the performance gap with the theoretical peak is also the largest, even for the ideal case. These results highlight a correlation between data density and the ability to speed up computations. Indeed, for the most sparse matrices, the memory accesses at row granularity do not use all the width of the data bus. Therefore, sparsity amplifies the effect of memory bandwidth bottleneck. Moreover, the more the rows are sparse, more processing time is short. This limits the ability to overlay the processing flow with the data transfer flow. For the matrix *cop20k_A*, the low arithmetic intensity per row limits the ability to hide data access latencies.

3.3 Case Study 2: Shallow Water Equation

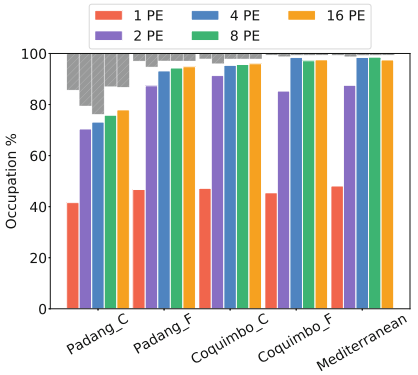
The Shallow Water Equations (SWE) are hyperbolic partial differential equations that describe a layer of fluid below a pressure surface. They can be solved with FEM. The code under study is the TsunAWI simulation code, a production code that implements the SWE with inundation, and whose results are used in the Indonesia Tsunami Early Warning System (InaTEWS), and under real-time constraints in the LEXIS European project [8]. The base data structure is a 2D unstructured mesh. This code has been optimized for performance, especially concerning the mesh ordering [9]. In this code, we have designed a kernel to speed up the calculation of the gradient, one of the operations of the tsunami simulation code. This operation is an interpolation of the sea surface height via barycentric coordinates. The barycentric coordinates are precomputed for each vertex element. The processing of an element requires five floating point operations, resulting in a low FLOP/byte ratio. The kernel is implemented with several independent processing elements (PEs). Each PE process a different part

Table 2. Characteristics of the set of meshes used for the experiments.

Region	Name	# Elements	# Vertices	Memory footprint
Indian ocean	Padang_C	460 119	231 586	14 Mb
	Padang_F	2 470 345	1 242 653	74 Mb
Pacific ocean	Coquibo_C	3 396 755	1 709 506	102 Mb
	Coquimbo_F	9 762 027	4 887 927	293 Mb
Mediterranean Sea	Mediterranean	9 917 645	4 999 404	298 Mb



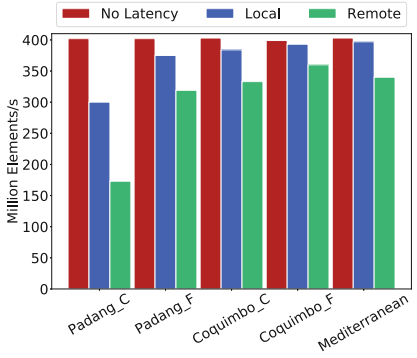
(a) Computation speed in MElements/s (higher is better).



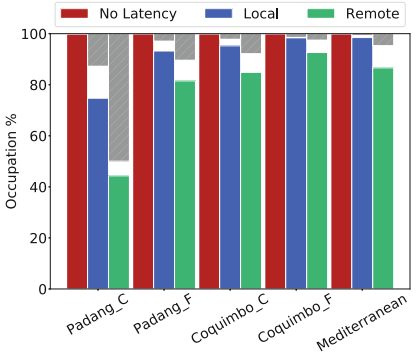
(b) Memory controller activity (occupancy percentage). Gray bars correspond to the loading time of the first data.

Fig. 6. Performance according to the number of processing elements (PEs).

of the mesh. The size of the sliding window is 1024 words. For the experiments, we have used five meshes presented in Table 2. The FPGA memory have been configured with 32 Mib. This was defined according to the maximum number of processing elements implemented (16 PEs), in order to allow each PE to prefetch up to 512 chunks of 1 kib per stream of data. Considering the memory footprint of each mesh and the maximum memory bandwidth of the FPGA, the theoretical peak computation speed is approximately 426 MElements/s for all meshes.



(a) Computation speed in MElements/s (higher is better).



(b) Memory controller activity (occupancy percentage). Gray bars correspond to the loading time of the first chunks.

Fig. 7. Performance according the system topology.

The first experiment aimed to study the speed up efficiency of the kernel according the parallelism level. To do this, the kernel has been implemented with 1 to 16 PEs. The computation speeds obtained are represented in Fig. 6a. It shows that the speed up gain sharply decreases beyond 2 PEs. The occupancy rate of the memory controller illustrated in the Fig. 6b provides a better understanding of these results. In this figure, the colored bars represent the percentage of time the controller is active, the gray bars correspond to the loading time of the first chunks as a percentage of simulation time, and the space between the colored bars and the gray bars represents the part of time during which the controller could have been used more. For the four largest meshes the results show a saturation of the memory controller for configurations with 4 PEs or more. We conclude that the bandwidth of the FPGA memory limits the performance scaling of the kernel. From 2 PEs onward, the controller occupancy rate is too high to efficiently speed up the processing by increasing the parallelism. Additionally, the smaller the mesh, the more the loading time of the first chunks represents a significant part of the total processing time, which reduces the computation speed.

For the second experiment, we have evaluated the kernel performance according to the topology with 4 PEs. Figure 7a and 7b respectively illustrate the computation speed obtained and the associated occupancy rate of the memory controller. These results show that increasing data access latency reduces computation speed and highlight a correlation between the size of the mesh and the slowdown. As shown in Fig. 7b, this effect can be explained by the proportion of the processing time spent to load the first chunks. For large datasets processing on the local node, where the load time is the least impacting, processing speeds almost reach those of the ideal case. For the remote node, where the read access latency is three times greater than on the local node, the slowing down of the computation speed is relatively low for the three largest meshes (from 10% to 17%). Finally, the performance of the *No Latency* configuration is close to the theoretical peak computation speed. The gap is due to the inability to access all nodes from the sliding window, which requires reading several times some chunks. Thus, we conclude that this mesh traversal method is almost ideal, given how small that gap is.

3.4 Discussion

The experiments have evaluated a data management model where accelerator tasks initiate access to distributed data. The experimental scenario used was the most disadvantageous, as the FPGA memory was empty at startup and all data had to be transferred during runtime. The results showed that thanks to prefetching, the programming model can efficiently hide the latencies of distributed data access. Nevertheless, this efficiency depends on the workload of the compute kernel. In practice, the observed workloads are huge. The size of the sparse matrices used in scientific applications can exceed ten gigabytes. The size of the complete datasets used for the tsunami simulation are at least ten times larger than the data subset used for the calculation of the gradient only. For the

complete simulation each element of the mesh involves a hundred floating operation per iteration. Thus, the processing time of the largest meshes on a high end processor can exceed several hours, and this motivates to distribute the processing to an heterogeneous system with FPGAs. This work shows that an S-DSM can simplify the distributed data management thanks to *chunk* partitioning and that the presented data management model solves the data access latency issue. Experiments have shown how in this model an FPGA can be supplied with data. As each accelerator is master of its access to data, this model can be extended to a distributed system integrating several FPGAs.

4 Related Work

Prior work have been done to provide shared memory for distributed systems with accelerators. Willendberg et al. [16] have proposed an FPGA communication infrastructure compatible to GASNet. This enables processing elements implemented on an FPGA to initiate remote direct memory access to remote FPGAs. Unicorn [4] provides a distributed shared memory (DSM) for CPU-GPU clusters. This is achieved with transactional semantics and deferred bulk data synchronization. StarPU [1] uses a DSM to manage data replication for heterogeneous distributed systems, but this DSM is not directly exposed to users. Recent work has studied *chunk* partitioning applied to sparse matrix for acceleration of sparse linear algebra. Winter et al. [17] have proposed an adaptive *chunk*-based SpGEMM for GPU. This approach uses *chunks* to store the partial results of multiplication, then uses the *chunk* metadata for the merge stage. Rubensson and Rudberg [13] have proposed the Chunks and Tasks programming model for parallelization of irregular applications. In this model, matrices are represented by sparse quatrees of chunks. MatRaptor [15] and REAP [14] uses a chunk-based CSR format adaptation and the row-wise product to implement SpGEMM kernel on FPGA. Barrio et al. [3] have proposed an unstructured mesh sorting algorithm to enabling stream processing for finite element method applications. This algorithm was applied to study the acceleration of scientific codes on CPU-FPGA platform.

5 Conclusion

Increasing the energy efficiency of HPC systems has become a major issue. Thanks to their reconfigurable architecture, FPGAs could increase power efficiency for HPC applications with irregular compute kernels. However, due to their complexity of use, FPGAs are underemployed in HPC systems. In this paper we have proposed a data management model for irregular compute kernel acceleration on FPGA integrated in distributed system. This model relies on a S-DSM to allow accelerators to initiate access to distributed data and on *chunk* partitioning to abstract the irregular structure of the datasets. We have shown how this data management model could be applied to compute kernels of sparse linear algebra and finite element method. We have conducted experiments with

a hybrid simulation tool, which exploits the physical system to provide accurate data. These experiments have shown that the data management model enables to efficiently hide high data access latencies. Finally, experiments have shown that memory bandwidth is a bottleneck. This phenomenon is normal since the studied applications are memory bound. High Memory Bandwidth (HBM) technologies as available on current and future FPGAs should help to remove this bottleneck and improving performance of compute kernels.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exp. **23**(2), 187–198 (2011)
2. Bader, M.: Space-Filling Curves: An Introduction with Applications in Scientific Computing, vol. 9. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-31046-1
3. Barrio, P., Carreras, C., López, J.A., Robles, Ó., Jevtic, R., Sierra, R.: Memory optimization in FPGA-accelerated scientific codes based on unstructured meshes. J. Syst. Archit. **60**(7), 579–591 (2014)
4. Beri, T., Bansal, S., Kumar, S.: The unicorn runtime: efficient distributed shared memory programming for hybrid CPU-GPU clusters. IEEE Trans. Parallel Distrib. Syst. **28**(5), 1518–1534 (2017)
5. Cudennec, L.: Software-distributed shared memory over heterogeneous micro-server architecture. In: Euro-Par 2017: Parallel Processing Workshops (2017)
6. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011)
7. Escobar, F.A., Chang, X., Valderrama, C.: Suitability analysis of FPGAs for heterogeneous platforms in HPC. IEEE Trans. Parallel Distrib. Syst. **27**(2), 600–612 (2016)
8. Goubier, T., et al.: Real-time model of computation over HPC/cloud orchestration - the LEXIS approach. In: Barolli, L., Poniszewska-Maranda, A., Enokido, T. (eds.) CISIS 2020. AISC, vol. 1194, pp. 255–266. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-50454-0_24
9. Goubier, T., Rakowsky, N., Harig, S.: Fast tsunami simulations for a real-time emergency response flow. In: 2020 IEEE/ACM HPC for Urgent Decision Making, UrgentHPC@SC 2020, pp. 21–26. IEEE (2020)
10. Gustavson, F.G.: Two fast algorithms for sparse matrices: multiplication and permuted transposition. ACM Trans. Math. Softw. **4**(3), 250–269 (1978)
11. High-Performance Conjugate Gradient (HPCG) Benchmark results, November 2020. https://www.top500.org/lists/hpcg/list/2020/11/
12. Lenormand, E., Goubier, T., Cudennec, L., Charles, H.P.: A combined fast/cycle accurate simulation tool for reconfigurable accelerator evaluation: application to distributed data management. In: 2020 International Workshop on Rapid System Prototyping (RSP) (2020)
13. Rubensson, E.H., Rudberg, E.: Chunks and tasks: a programming model for parallelization of dynamic algorithms. Parallel Comput. **40**(7), 328–343 (2014)
14. Soltaniyeh, M., Martin, R.P., Nagarakatte, S.: Synergistic CPU-FPGA acceleration of sparse linear algebra. CoRR abs/2004.13907 (2020)

15. Srivastava, N.K., Jin, H., Liu, J., Albonesi, D.H., Zhang, Z.: MatRaptor: a sparse-sparse matrix multiplication accelerator based on row-wise product. In: 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 766–780. IEEE (2020)
16. Willenberg, R., Chow, P.: A remote memory access infrastructure for global address space programming models in FPGAs. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 211–220. ACM (2013)
17. Winter, M., Mlakar, D., Zayer, R., Seidel, H.P., Steinberger, M.: Adaptive sparse matrix-matrix multiplication on the GPU. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 68–81. ACM (2019)