Task-Level Parallelism for the Multifrontal Method in Tightly Coupled CPU-FPGA Architectures

Zerong He[1], Zekang Cheng[1], Zhongguang Xu[2], and Xi Jin[1]

[1]*Institute of Microelectronics, Department of Physics, University of Science and Technology of China.*
[2]*School of Microelectronics, University of Science and Technology of China.*
Contact: {hezerong, chengzk}@mail.ustc.edu.cn, {xuxu, jinxi}@ustc.edu.cn

***Abstract*—The multifrontal method is an efficient direct method for solving sparse linear systems. This algorithm transforms the factorization of a large and sparse matrix into a sequence of dense matrix operations. Some of these dense operations can be accelerated with FPGAs, while others are suitable for CPUs. In this paper, we propose a tightly coupled heterogeneous domain-specific architecture (DSA) for the efficient execution of the multifrontal algorithm. This tightly coupled hardware architecture ensures low communication overhead between the CPU and FPGA. Each matrix operation is assigned to the CPU or FPGA based on its characteristics. Furthermore, we apply the task-based scheduling model for multi-core architectures to our heterogeneous DSA. Considering the difference in the storage structure of the CPU and FPGA, we modify the original task graph and propose a data-centric task graph that is more suitable for the FPGA. Based on this scheduling model, we propose two optimizations to further improve performance. Finally, we test the scheduling overhead of the system and determine the finest task granularity accordingly. We evaluate our architecture on Xilinx ZCU102. Compared to MUMPS, for a set of 5 matrices, our implementation can achieve an average of $4.3\times$ performance improvement with just 10% of the computing power.**

***Index Terms*—multifrontal methods, FPGA, heterogeneous architectures, task-level parallelism**

I. Introduction

Solving sparse linear systems plays a pivotal role in numerous scientific and engineering applications [1]–[3]. The two primary approaches employed for this purpose are direct methods and iterative methods. Direct methods can calculate the result in a finite number of steps, as long as a solution to the linear system exists. In contrast, iterative methods tend to converge slowly or even fail to converge for ill-conditioned matrices with large condition numbers. Among direct methods, the multifrontal method [4], [5] has proved to be extremely valuable due to better data locality and its adaptability to parallel computations. Despite its potential for parallelization, existing architectures and scheduling strategies fail to effectively exploit its inherent parallelism.

The multifrontal method transforms the factorization of a sparse matrix into partial factorizations of a series of smaller frontal dense matrices by constructing an elimination tree. The factorization of a frontal matrix comprises three main operators: panel factorization (PANEL), triangular solve with multiple right-hand sides (TRSM), and general matrix-matrix multiply (GEMM) [6]. Since these operators have different dataflow and the topology of the elimination tree is irregular, as far as we know, there is no FPGA-based accelerator [7]–[9] for this algorithm. We propose a hybrid CPU-FPGA architecture and assign each operator to the appropriate components. The operations involved in TRSM and GEMM are almost exclusively matrix multiplication. Although these operations are conceptually straightforward, they exhibit high time complexity (i.e., $O(n^3)$). FPGAs have excellent energy efficiency and can provide the desired arithmetic intensity for these two operators. On the other hand, PANEL has a lower time complexity (i.e., $O(n^2)$), but entails complex dataflow and an indefinite loop structure. The CPU, with its complex control logic, is a good choice for this operator. Furthermore, our architecture is tightly coupled. Compared with other heterogeneous architectures, such as the CPU-GPU approach [10]–[12], the communication latency of the FPGA and CPU is very low.

The potential for parallelism of the multifrontal method stems from the fact that the elimination tree points out the dependencies between the frontal matrices, enabling the parallel execution of independent frontal matrices. Due to the irregular topology of the elimination tree, the size of the workload changes during the execution. Therefore, we propose a task-based execution model to exploit task-level parallelism [13], [14]. In task-based scheduling, the dependencies between tasks are represented by a task graph and tasks with no dependencies can be executed in parallel. Since there is no hierarchical storage structure on FPGAs and data movement is expressed implicitly in the task graph, the input and output data of a task can only be accessed from DRAM. To hide the long latency of DRAM accesses on FPGA platforms, we propose a data-centric task graph that explicitly points out data movement between DRAM and the on-chip buffer. Based on the data-centric task graph, we propose two optimizations for scheduling: Throttle and Immediate Successor. Throttle aims to reduce scheduling overhead and Immediate Successor focuses on minimizing off-chip accesses by exploiting data locality between tasks.

The contributions of this work can be summarized as follows:

- We introduce a tightly coupled heterogeneous DSA designed to accelerate the multifrontal method, leveraging the advantages of both CPUs and FPGAs.

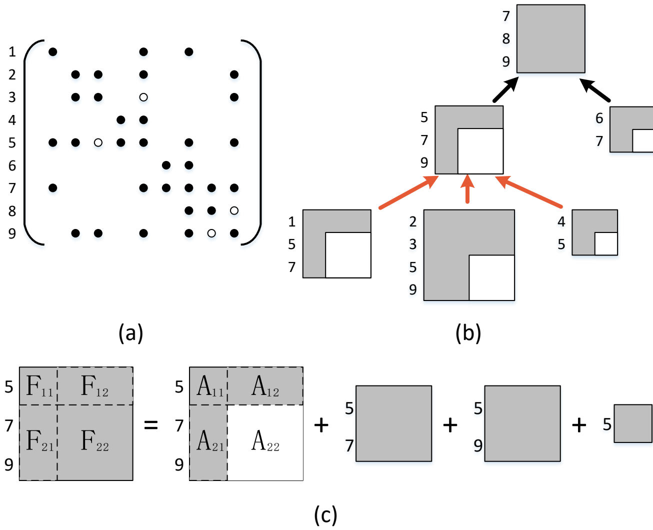

Fig. 1. (a) The nonzero structure of a sparse matrix. The fill-in is represented by a '∘'. (b) The elimination tree of the sparse matrix in (a). (c) Assembling the frontal matrix indicated by the red arrows in (b).

- We propose a data-centric task-based execution model, accompanied by the introduction of two optimizations for scheduling.
- Performance is only improved when task granularity exceeds a certain threshold. We determine the threshold experimentally and merge tasks whose granularity is below this threshold. Then we evaluate the performance of our architecture and compare it with MUMPS [15].

II. Background

Multifrontal methods generally use a three-phase approach to solve a sparse linear system (i.e., $Ax = b$): *Analysis*, *Factorization* and *Solve*.

In *Analysis* phase, as shown in Fig. 1 (a), the structure of the coefficient matrix is analyzed and the fill-in entries (i.e., extra nonzeros generated during Gaussian elimination process) are obtained. Based on the nonzero structure of the sparse matrix, an elimination tree is constructed. To enhance data locality and computational efficiency, the supernodal version of multifrontal methods is commonly employed. A supernode [4], [5] is a contiguous range of columns having the same lower diagonal nonzero structure in the input sparse matrix, such as the root node (i.e., Column 7, 8, 9) in Fig. 1 (b).

In *Factorization* phase, the elimination tree is traversed in a topological order from bottom to top. We refer to the matrix corresponding to each node as the original matrix, and Schur complement generated during factorization of each node as the contribution matrix. The processing of a node τ consists of four steps: 1.

1) As illustrated in Fig. 1 (c), the original matrix and the contribution matrices generated by child nodes of τ are assembled into the frontal matrix F_τ. This operation is also called an extend-add operation.
2) PANEL. Performing LU factorization of the block F_{11}: $F_{11} = L_{11}U_{11}$. Updating the block F_{21}: $F_{21} = F_{21}U_{11}^{-1}$. Although F_{11} in Fig. 1 (c) is an element, it is a block matrix in most cases.
3) TRSM. Updating the block F_{12}: $F_{12} = L_{11}^{-1}F_{12}$.
4) GEMM. Computing the contribution matrix U_τ: $U_\tau = F_{22} - F_{21}F_{12}$.

In *Solve* phase, we obtain the solution by solving two triangular systems (i.e., $Ly = b$ and $Ux = y$). It is experimentally demonstrated that most of the execution time of the multifrontal algorithm is spent on *Factorization* phase [11]. Therefore, our optimization strategies primarily focus on this phase.

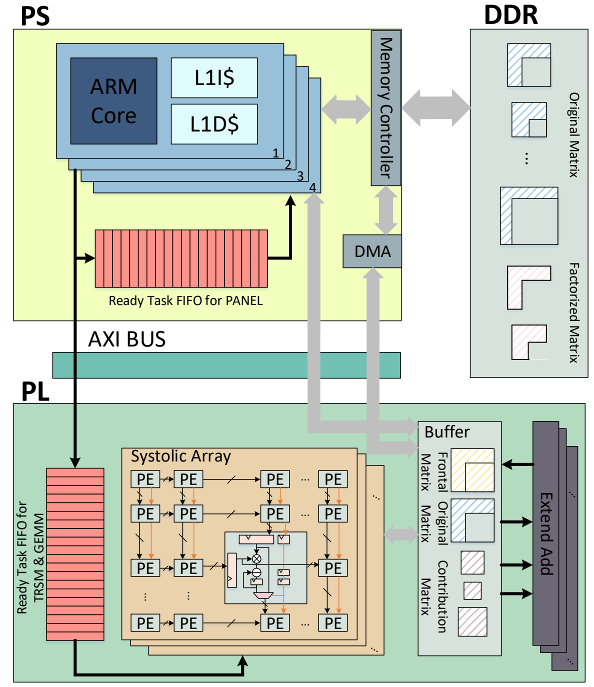

Fig. 2. Heterogeneous architecture for the supernodal multifrontal algorithm.

III. The Tightly Coupled Heterogeneous Architecture

In this section, we design a tightly coupled heterogeneous DSA. We first describe the architecture overview and then present details of the architecture design.

A. Architecture Overview

Fig. 2 illustrates the tightly coupled heterogeneous DSA. The architecture comprises Processing System (PS) and Programmable Logic (PL). PS contains four high-performance ARM cores. In PL, we design Extend Add modules and systolic arrays to accelerate the extend-add operation and matrix multiplication. As mentioned earlier, the algorithm contains three main tasks: PANEL, TRSM and GEMM. PS is responsible for managing all tasks and executing PANEL, while TRSM and GEMM are assigned to PL. PS can directly access Double Data Rate (DDR) memory through Memory Controller. While the data transfer between PL and DDR is accomplished by sending requests to Memory Controller through Direct Memory Access (DMA). PS and PL are

packaged on the same chip. The data transfer between them is achieved through the Advanced eXtensible Interface (AXI) bus, allowing for efficient and low-latency communication.

B. Buffer Management

The multifrontal method produces a large number of intermediate matrices with variable sizes during the execution. Therefore, buffer management is an important issue. Amestoy and Duff, in [16], discuss several memory management schemes in a parallel environment, including garbage collection, fixed block, binary buddy system and combined strategies. These schemes target multi-core architectures. They are either not easy to implement on PL (e.g., buddy system) or not efficient enough (e.g., garbage collection and fixed block).

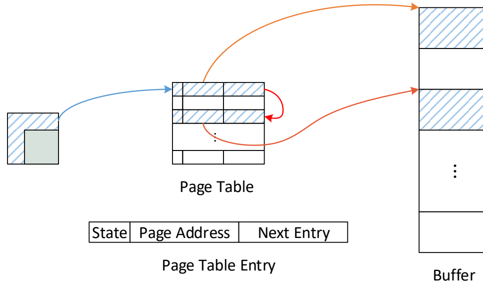

Fig. 3. Buffer management scheme.

In Fig. 3, we introduce a buffer management scheme based on page tables. Each matrix is stored in several discrete pages and these pages are organized in a linked list. Each page table entry consists of three parts: a Status bit indicating whether the page is idle or in use, a Page Address pointing to the first address of the page and a Next Entry pointing to the next item in the linked list. One crucial aspect of our buffer management scheme is the selection of the page size. Choosing an excessively large page size can lead to wastage of storage space, while opting for a small page size can increase management overhead and the number of accesses to the page table. Therefore, the selection of the page size is done skillfully, striking a balance between storage efficiency and management overhead.

In summary, our scheme has the following advantages: 1.

1) No memory fragmentation and no need for garbage collection: The use of discrete pages and the linked list organization prevent memory fragmentation issues.
2) Hardware-based page selection: We employ a hardware encoder to select idle pages in real time. This hardware implementation of buffer management is more efficient compared to software-based approaches.
3) Minimal overhead of accessing the page table: Since the page table is stored directly on the buffer of PL, the overhead of accessing it is minimal.

C. Systolic Array

The classical systolic array can handle GEMM but not TRSM, because TRSM involves the inverse of the lower triangular matrix (i.e., L_{11}^{-1}). In the classical systolic array, one input matrix is fed in a row at a time from the top of the array and is passed down the array. In the execution of TRSM, the data passed down the array is no longer the input matrix itself, but rather the calculation result of the previous row. This requires a modification to the processing pattern of the classical systolic array. As shown in Fig. 4, we add a multiplexer (MUX) and a control signal (represented by the red arrow) to each PE. Switching between the two processing patterns (i.e., TRSM and GEMM) is achieved through the control signal.

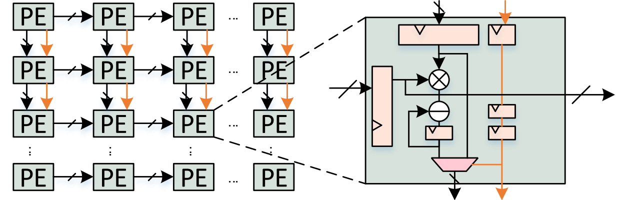

Fig. 4. Systolic array for TRSM and GEMM.

IV. Exploring Task-Level Parallelism

In this section, we first describe our task-based scheduling model without any optimizations. Then we explain the data-centric task graph. Finally, two optimizations for scheduling are introduced.

A. Design of Baseline Execution Model

In the task-based execution model, an algorithm is usually expressed as a Directed Acyclic Graph (DAG) of tasks and an associated dataflow. Fig. 5 (a) illustrates a task graph generated from the example depicted in Fig. 1 (b). The size of the task graph increases with the sparse linear system. To reduce the storage overhead, the task graph is generated dynamically rather than being pre-computed and stored in its entirety. This dynamic generation approach allows for efficient utilization of memory resources while still enabling effective task scheduling and execution.

During execution, a controller consisting of a small number of threads is assigned to generate the task graph and submit tasks. We set up a ready task FIFO to hold the tasks that can be issued. Submitted tasks can enter the FIFO only if their dependencies are released. Otherwise, these tasks are kept in memory and wait for their task predecessor sets to release their dependencies. For example, consider a Read-After-Write (RAW) dependency between tasks A and B, where the output of B is the input of A. The dependency of task A can be released by task B after B has been completed. Accelerators and most threads access the FIFO to fetch ready tasks and execute them. To prevent race conditions arising from simultaneous access to the FIFO, lock-based synchronization mechanisms are employed. However, mutually exclusive access to the FIFO can occasionally become a performance bottleneck. In our architecture, PS and PL are responsible for different tasks. To ensure efficient task management, separate FIFOs are utilized: Ready Task FIFO for PANEL in PS and Ready Task FIFO for TRSM & GEMM in PL. When a task is completed, all tasks

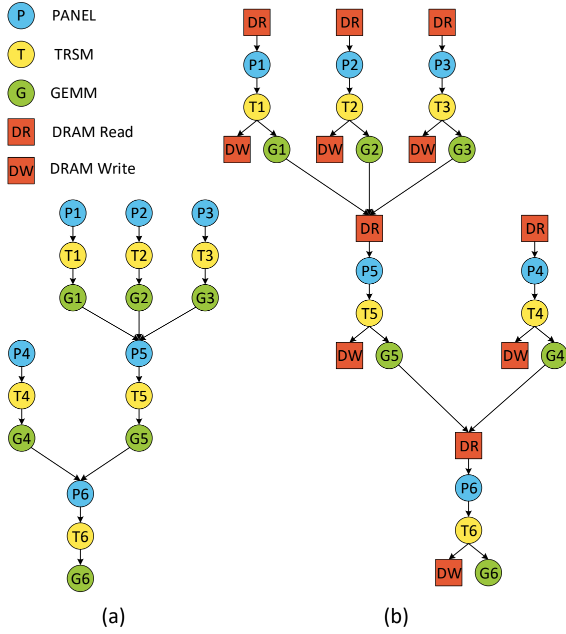

Fig. 5. Task graph for *Factorization* phase. (a) Baseline. (b) Data-centric task graph.

that have data dependencies with it are checked to determine if their dependencies can be released.

B. Data-Centric Task Graph

The task graph can work well on a multi-core architecture, but not on our heterogeneous DSA. CPUs prioritize accessing data from the cache to avoid the long latency associated with DDR accesses, and there is corresponding hardware on CPUs to automatically perform data movement between DDR and the cache. However, FPGAs do not have these mechanisms. Each time PL executes TRSM and GEMM, it needs to read the input of the task from DDR and write the result back to DDR. Considering the data transfer overhead, the utilization of computational resources on PL can be relatively low.

To solve this problem, we introduce data transfer into the task graph. Fig. 5 (b) and Fig. 6 illustrate our data-centric task graph and the corresponding dataflow, respectively. In our approach, we consider the buffer on PL as part of the address space. For the factorization of a frontal matrix, task DR moves the original matrix from DDR to the buffer. Since the contribution matrices are already stored on the buffer, the extend-add operation can be performed immediately. It can be seen from Fig. 6 that each computational task (i.e., PANEL, TRSM and GEMM) can directly access the input data from the buffer and writes the result to the buffer. The factorized matrix is generated after the completion of TRSM. While GEMM updates the contribution matrix using the factorized matrix, task DW writes the factorized matrix on the buffer back to DDR. In Fig. 7, we compare the execution efficiency before and after optimization. Suppose we have one CPU core and two systolic arrays. The data-centric task graph effectively reduces communication with off-chip memory. Since computational tasks no longer need to access off-chip memory, the overall execution efficiency is improved.

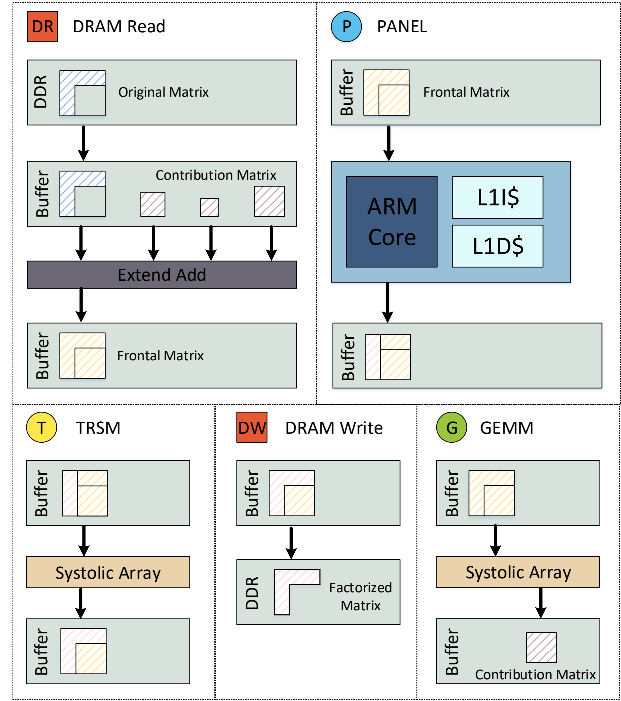

Fig. 6. Dataflow of different tasks.

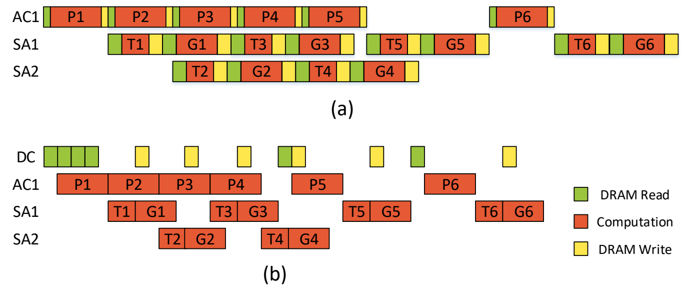

Fig. 7. Comparison of execution efficiency for the task graph in Fig. 5. (a) Baseline. (b) Data-centric task graph. DC: DMA Controller, AC: Arm Core, SA: Systolic Array.

C. Throttle

As mentioned in Section IV-A, our task-based execution model is a centralized scheduling scheme. Typically, each execution unit (i.e., a thread or a systolic array) sends a message to the controller upon completing its current task. Upon receiving the message, the controller releases the dependencies of that task's successor set. However, the controller can only receive messages serially. Furthermore, the process of dependency release involves memory access and is also serialized. As a result, a large number of execution units are waiting for a response from the controller, leading to decreased overall execution efficiency. In the case of fine task granularity, this is exacerbated by an increased number of tasks.

To address this issue, we introduce Throttle, a mechanism that restricts the number of task dependencies that can be

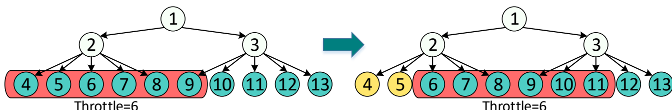

Fig. 8. An example of Throttle. Write nodes are completed tasks, green nodes are tasks that are awaiting dependency release, and yellow nodes are tasks whose dependencies have been released. The threshold is set to 6.

released. We give an example of this mechanism in Fig. 8. Let's assume a threshold value of 6. Without Throttle, upon completing tasks 2 and 3, the dependencies of tasks 4-13 can be released. However, with Throttle, only the dependencies of the six tasks (i.e., tasks 4-9) within the red sliding window can be released. The gain is that the execution unit of task 3 can proceed to execute the next task as soon as the dependency of task 9 is released, without waiting for tasks 9-13. The sliding window is updated periodically, pushing tasks that are awaiting dependency release and pulling tasks whose dependencies have been released. In this way, we can limit the task scheduling overhead to a reasonable range, preventing it from becoming a performance bottleneck.

D. Immediate Successor

In a task graph, some nodes have unique immediate predecessors. In this case, all input data of the node comes from its only immediate predecessor. Prioritizing the execution of such nodes can make full use of on-chip data. We call this mechanism Immediate Successor. Fig. 9 illustrates an example of Immediate Successor. Suppose that task D has been completed. Task A is the only immediate predecessor of task B, whereas task C has other predecessors. After the completion of task A, its output data is temporarily stored on-chip before being sent to memory and cleared. At this point, all input data of task B can be obtained directly from the on-chip. To ensure that task B is prioritized, we send it directly to the local buffer of the execution unit. Since the output data of task D may have already been cleared from the on-chip, prioritizing task C would not provide significant benefits. Therefore, task C enters the ready task FIFO as normal.

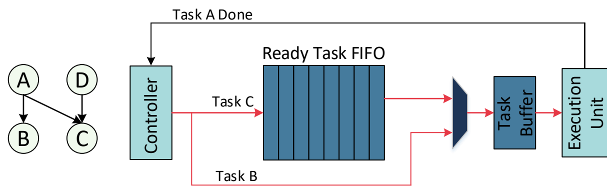

Fig. 9. An example of Immediate Successor.

V. Evaluation Results

We first describe our hardware platforms. Next, we determine the finest task granularity with a semi-quantitative approach. Finally, we compare our architecture to a parallel sparse direct solver MUMPS.

A. Experimental Setup

1) Platform: We implement the accelerator in Verilog HDL on Xilinx ZCU102. This platform features an ARM Cortex-A53 processor. To evaluate the performance of our proposal, we compare our architecture with a parallel sparse direct solver MUMPS that runs on Intel Xeon Gold 6258R Processor. MUMPS utilizes MPI and OpenMP and has been carefully fine-tuned for optimal performance.

2) Benchmark: As illustrated in Tab. I, we evaluate our scheduling model and architecture on five benchmark matrices generated from electromagnetic field simulations for antennas. These matrices correspond to different antenna structures, exhibiting diverse sizes, densities, and element distributions.

TABLE I
Benchmark Overview.

	Dimention	Nonzeros	Matrix Density
case 1	281904	11542476	0.0145%
case 2	618854	25198164	0.00658%
case 3	1373938	56417664	0.00299%
case 4	7702396	320983700	0.000541%
case 5	39863268	1631954894	0.000103%

B. Analysis of Task Granularity

The finer the task granularity, the greater the scheduling overhead relative to the execution overhead [17], [18]. Tasks should be coarse enough to fully leverage the performance benefits offered by the FPGA, considering the scheduling overhead. We set a threshold for the finest allowed task granularity, ensuring that the scheduling overhead remains lower than the execution overhead. The threshold is obtained with a semi-quantitative approach.

We calculate the execution overhead for each computational task (i.e., PANEL, TRSM, GEMM, and Extend Add). Because the input data of each task is a dense matrix and the systolic array has fixed dataflow, the execution overhead of tasks performed on the FPGA is a function of the size of the matrix and the systolic array. For each type of task performed on the CPU, the execution time is proportional to the task size. We obtain the scale factor for each type of task experimentally so that the execution time can be estimated.

We measure the average scheduling

Task granularity is measured in terms of the execution time of each task. Tasks with granularity below the threshold should be merged into larger tasks. To increase the task granularity, we should increase the size of nodes in the elimination tree. Therefore, it is necessary to treat some logical zeros as nonzeros. In this way, nodes/supernodes can be amalgamated into a larger supernode. We repeat this process until the granularity of all tasks surpasses the threshold. Fig. 10 illustrates the distribution of task granularity after merging.

C. Performance Comparison

To achieve optimal performance, the tasks assigned to the CPU and FPGA should match the architectural characteristics. As previously mentioned, PANEL is assigned to the CPU, while GEMM and TRSM are executed by the FPGA. Memory access involved in DR and DW is controlled by DMA. The extend-add operation in DR can be expressed as the addition of multiple sparse matrices. The control logic for this operation is not complex, while the arithmetic intensity is relatively low. As a result, it is theoretically difficult to determine which is

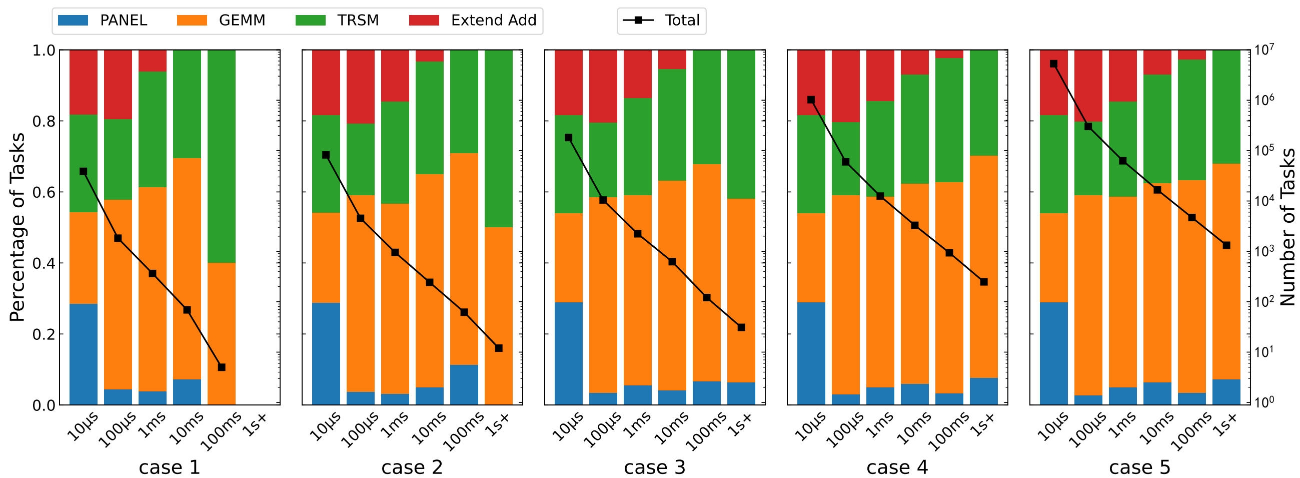

Fig. 10. Task granularity of five benchmark matrices.

better suited to perform this operation, the CPU or the FPGA. To address this, we design two schemes for comparison. In Design 1, the extend-add operation is executed by the FPGA. This requires allocating appropriate resources for the systolic array and Extend Add module. In Design 2, this operation is executed by the CPU. In this case, we can allocate more resources for systolic arrays on the FPGA, but this may increase the workload on the CPU.

In Tab. II, we list the resource consumption of each module. The clock frequency is 200 MHz. The systolic array is designed with a size of 8×8, while the Extend Add module performs 16 addition operations per cycle. Design 2 offloads the extend-add operation to the CPU, allowing for additional systolic arrays. However, since a single systolic array consumes far more resources than an Extend Add module, Design 2 has only one more systolic array than Design 1.

In Fig. 11, we compare the two designs using MUMPS as the baseline. It can be noticed in Fig. 10 that GEMM and TRSM take up most of the execution time. The time complexity of PANEL and Extend Add is relatively much smaller, and the CPU has enough computing power to cope with them. As a result, the performance bottleneck arises from GEMM and TRSM. In Design 2, due to the additional systolic array, the performance bottleneck can be slightly alleviated.

Compared to MUMPS, our approach can achieve significant performance gains for the first three datasets, but not the last two. In Fig. 10, it is obvious that the size of case 1 to case 5 increases sequentially. For large datasets (e.g., case 4 and case 5), the execution is dominated by large dense matrix computations. Consequently, the computing overhead is much

TABLE II
RESOURCE USAGE OF EACH EXECUTION MODULE.

	Systolic Array	Extend Add	Buffer Management
LUT	29686(10.83%)	5085(1.86%)	685(0.25%)
FF	15112(2.76%)	2624(0.48%)	473(0.09%)
BRAM	0(0%)	0(0%)	0(0%)
DSP	256(10.16%)	32(1.27%)	0(0%)
Design 1	8	7	1
Design 2	9	0	1

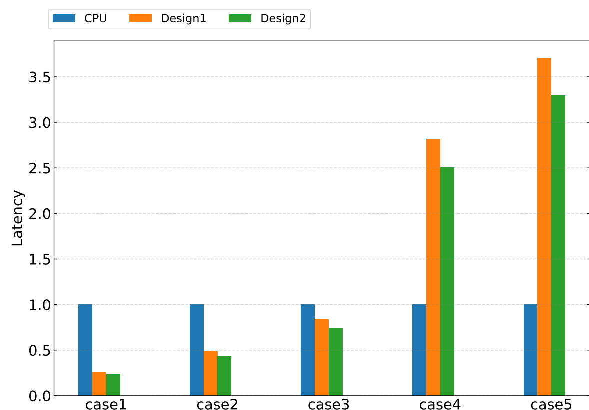

Fig. 11. Performance comparison. Design 1 and Design 2 are our schemes. The extend-add operation is performed by FPGA in Design 1 and by CPU in Design 2.

larger than the scheduling overhead. Additionally, the floating-point arithmetic of our DSA is lower than that of the CPU used for comparison. Therefore, for these two larger datasets, our DSA does not perform as well as the CPU. On the other hand, the first three datasets are smaller in size and can benefit from the schedule of fine-grained tasks. Taking case 1 as an example, our DSA achieves a $4.3\times$ performance improvement with approximately 10% of the computing power used by the CPU.

VI. Conclusion

In this paper, we accelerate the supernodal multifrontal algorithm using a tightly coupled heterogeneous DSA. We study the task-based scheduling model for multi-core architectures and improve it according to the architectural characteristics of FPGAs. Moreover, we experimentally obtain the threshold for the finest allowed task granularity and compare our system with a parallel sparse direct solver MUMPS. One of the strengths of our work is the potential for broader applicability beyond the multifrontal method. Our acceleration scheme can be adapted and reused in other sparse algebraic applications.

REFERENCES

[1] L. W. Nagel, "Spice2: A computer program to simulate semiconductor circuits," Ph.D. dissertation, EECS Department, University of California, Berkeley, May 1975. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1975/9602.html

[2] H. Jasak, A. Jemcov, Z. Tukovic *et al.*, "Openfoam: A c++ library for complex physics simulations," in *International workshop on coupled methods in numerical dynamics*, vol. 1000, 2007, pp. 1–20.

[3] E. Madenci and I. Guven, *The finite element method and applications in engineering using ANSYS®*. Springer, 2015.

[4] J.-Y. L'Excellent, "Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects," Habilitation à diriger des recherches, Ecole normale supérieure de lyon - ENS LYON, Sep. 2012. [Online]. Available: https://theses.hal.science/tel-00737751

[5] J. W. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM review*, vol. 34, no. 1, pp. 82–109, 1992.

[6] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[7] N. Kapre *et al.*, "Breaking sequential dependencies in fpga-based sparse lu factorization," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 60–63.

[8] G. Wu, X. Xie, Y. Dou, J. Sun, D. Wu, and Y. Li, "Parallelizing sparse lu decomposition on fpgas," in *2012 International Conference on Field-Programmable Technology*. IEEE, 2012, pp. 352–359.

[9] E. B. Tavakoli, M. Riera, M. H. Quraishi, and F. Ren, "Fschol: An opencl-based hpc framework for accelerating sparse cholesky factorization on fpgas," in *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2021, pp. 209–220.

[10] R. F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes, "Multifrontal computations on gpus and their multi-core hosts," in *High Performance Computing for Computational Science–VECPAR 2010: 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers 9*. Springer, 2011, pp. 71–82.

[11] D. Y. Chenhan, W. Wang *et al.*, "A cpu–gpu hybrid approach for the unsymmetric multifrontal method," *Parallel Computing*, vol. 37, no. 12, pp. 759–770, 2011.

[12] P. Ghysels and R. Synk, "High performance sparse multifrontal solvers on modern gpus," *Parallel Computing*, vol. 110, p. 102897, 2022.

[13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[14] Barcelona Supercomputing Center, "Ompss-2 specification," 2021, https://pm.bsc.es/ftp/ompss-2/doc/spec/.

[15] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "Mumps: a general purpose distributed memory sparse solver," in *International Workshop on Applied Parallel Computing*. Springer, 2000, pp. 121–130.

[16] P. R. Amestoy and l. S. Duff, "Memory management issues in sparse multifrontal methods on multiprocessors," *The International Journal of Supercomputing Applications*, vol. 7, no. 1, pp. 64–82, 1993.

[17] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault, "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 29–38.

[18] K. Kim and V. Eijkhout, "Scheduling a parallel sparse direct solver to multiple gpus," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1401–1408.