A Parallel Hardware Architecture for Fork-Join Parallel Applications

Atakan Doğan, İsmail San
Department of Electrical and Electronics Engineering
Anadolu University
Eskişehir, Turkey
email: atdogan@anadolu.edu.tr, email: isan@anadolu.edu.tr

Kemal Ebcioğlu
Global Supercomputing Corporation
Yorktown Heights, NY, USA
email: kemal.ebcioglu@acm.org

***Abstract*—In order to facilitate the implementation on hardware and improve the performance of a class of fork-join applications that can be modeled by an OpenMP program, a parallel hardware architecture with a specialized memory hierarchy is proposed. Furthermore, three different case studies are provided to show how this model can be employed for the hardware acceleration of such applications.**

Keywords-parallel applications; parallel hardware; hardware thread; caches; NoCs.

I. INTRODUCTION

The OpenMP Application Programming Interface is a well-established standard for parallel programming on shared-memory multiprocessors. OpenMP has adopted the fork-join model of parallel execution. According to this model, an OpenMP program begins as a single thread of execution, called an initial thread. When any thread encounters an OpenMP parallel construct, a team of master and slave threads (this is the fork) is created to execute the code enclosed by the construct. At the end of the construct, only the master thread continues, while all slave threads terminate (this is the join) [1].

In the literature, there are several studies that attempt to generate a parallel hardware from OpenMP applications [2]-[7]. A few High Level Synthesis (HLS) tools, such as Xilinx's SDAccel [8], have support to produce parallel hardware from OpenCL. Finally, fork-join like hardware constructs that are automatically generated from sequential code using compiler dependence analysis is described in [9].

The most recent and similar study in the literature is presented by [9]. However, [9] does not clearly specify how it copes with at least the following issues: (i) How does it achieve an implicit barrier among threads at the end of a parallel region? (ii) How does it perform reduction on hardware? (iii) Is multiple level of fork-join parallelism possible? The parallel hardware architecture model proposed here will be proved to have an answer for these questions that are needed for the acceleration of OpenMP applications.

The rest of the paper is organized as follows: Section II introduces the proposed parallel hardware architecture. Section III shows how this architecture provides support for the fork-join applications using three different case studies. Finally, Section IV concludes the paper.

II. PARALLEL HARDWARE ARCHITECTURE

Motivated by these and other related studies, a generic parallel hardware architecture that can be configured by an OpenMP program for a class of fork-join parallel applications is proposed in this study and illustrated in Figure 1.

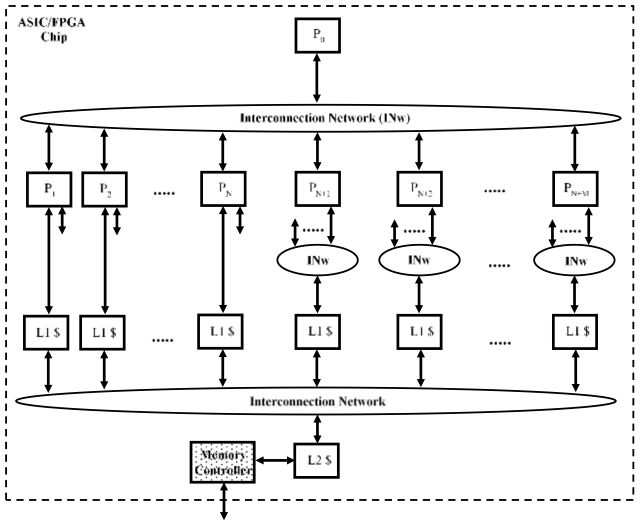

Figure 1. A parallel hardware architecture for parallel applications.

Inside an FPGA (Field Programmable Gate Array) or ASIC (Application Specific Integrated Circuit) chip in Figure 1, there are a few types of components, which include hardware threads, L1 caches (L1 $), single L2 cache (L2 $), and interconnection networks (INw). Each component communicates with messages through its sending FIFO (First-In First-Out) and receiving FIFO interfaces, where an arrow in Figure 1 represents such a bidirectional message communication interface.

A. Hardware Threads

A hardware thread component is a finite state machine that performs either coordination (P_0 in Figure 1) or computation (P_i, $i>0$).

P_0 is the master hardware thread that coordinates/synchronizes the execution of a parallel application among the slave hardware threads. That is, P_0

spawns (forks) new slave threads by sending a start request to each of these slave threads; a barrier synchronization (join) among slave threads is completed once P_0 receives a finish response from each of them.

P_i, $1 \leq i \leq N+M$, are a team of slave hardware threads that really implement the execution of parallel application as follows:

- Waiting for a start request from its parent thread P_0.
- After receiving a start request, working on the task while sending memory load/store requests to L1 cache units. Note that the task, for example, corresponds to the computation due to of #pragma omp parallel for {…}.
- Upon completing the computation, sending a finish response to P_0.

B. Memory Hierarchy

A two-level on-chip memory hierarchy as shown in Figure 1 is proposed to support the parallel hardware acceleration.

L1 $ is a write-back cache that supports *load*, *store*, and *flush* requests coming from the slave hardware threads. In Figure 1, a dedicated L1 cache is instantiated per slave thread that allows each thread to access memory independently for the maximum performance. Note that this model is complaint to the OpenMP shared memory model.

L2 $ is a write-back cache that receives *line read* and *line write* requests from L1 $ components and responds to the requests accordingly. All initial and final data of the parallel application are assumed to be kept in the L2 cache. Furthermore, according to Figure 1, the L2 $ state data is held in an on-chip memory, whereas the application data are kept in an off-chip memory accessed through a memory controller.

C. Interconnection Network

Interconnection Network (INw) is a packet-based network-on-chip network (NoC) that interconnects various components of the architecture [9].

III. CASE STUDIES

A. Matrix-Vector Multiplication

The first case study considers the matrix-vector multiplication of y = A×x, where A is an n×n matrix, and both x and y denote n×1 vectors. The parallel implementation of the matrix-vector multiplication is supported by Figure 1 as follows:

- Each hardware thread P_i, $1 \leq i \leq N$, starts its computation upon receiving a start request from P_0.
- Each P_i, $1 \leq i \leq N$, computes n/N vector elements y[k], where y[k]=A[k,:]×x requires a complete row A[k,:] of the matrix A and the whole vector x.
- The L1 cache directly attached to every P_i ($L1_i$) is loaded with n/N rows of the matrix and the vector x from the L2 cache during the computation.
- Each P_i computes its part of y[k] and stores it into its L1 cache. At the end of its computation, each P_i sends a flush request to $L1_i$ so that all dirty lines of y[k] in $L1_i$ are written back to the L2 cache.
- Each P_i waits for a flush acknowledgement from $L1_i$, and then sends a finish response to P_0. Once P_0 receives N finish responses, the matrix-vector multiplication is completed.

Note that the following components in Figure 1 will not be needed for case A: hardware threads P_i, $N+1 \leq i \leq N+M$, the corresponding interconnection networks and L1 caches. As a result, the matrix-vector multiplication is implemented as a single fork-join paradigm.

B. Vector Inner-Product

The second case study considers the vector inner-product of r = b×x, where b is a 1×n row vector, x denotes an n×1 column vector, and r is a resulting scalar value. The parallelization of the vector inner-product can be accomplished within the framework of Figure 1 as follows:

- Upon receiving a start request from P_0, each P_i, $1 \leq i \leq N$, computes a partial sum scalar value y[i] by means of multiplying its exclusive part of n/N elements of vectors b and x, and then performing n/N sums.
- Since each thread needs n/N elements of both vectors, $L1_i$ is loaded with n/N columns of b and n/N rows of x from the L2 cache.
- After the computation of y[i] is over, each P_i, $1 \leq i \leq N$, writes y[i] into $L1_{N+1}$, and then sends a finish response to P_0.
- After P_0 receives N finish responses, P_0 sends another start request to the thread P_{N+1} so that P_{N+1} can perform the final reduction sum over y[i], $1 \leq i \leq N$, in cache $L1_{N+1}$ and write the result r into $L1_{N+1}$.
- Finally, P_{N+1} sends a flush request to $L1_{N+1}$, waits for a flush acknowledgement from $L1_{N+1}$, and sends a finish response to P_0. Once P_0 receives this final finish response message, the vector inner-product is finished.

The hardware threads P_i, $N+2 \leq i \leq N+M$, the related networks and L1 caches in Figure 1 will not be needed for case B. Thus, the implementation of a vector-inner product requires a fork-join type of parallel execution followed by a reduction operation.

C. Matrix-Matrix Multiplication

Finally, the matrix-matrix multiplication of C = A×B, where each matrix is an n×n dense matrix, is considered as the third case study. Such a matrix multiplication, for example, can be performed as a block-matrix multiplication using (n/Q)x(n/Q) submatrices for Q=2 as shown below:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21},$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22},$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21},$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

The parallel implementation of the block matrix multiplication is supported by Figure 1 as follows:

- Each P_i, $1 \leq i \leq N$, starts its computation upon receiving a start request from P_0.
- Each P_i, $1 \leq i \leq N$, deals with a single submatrix multiplication $Y_{jkl} = A_{jk} \times B_{kl}$ (for example, $Y_{111}=A_{11} \times B_{11}$, $Y_{121}=A_{12} \times B_{21}$, and so on). As a result, each P_i requires the corresponding A_{jk} and B_{kl} submatrices with dimensions of $(n/Q) \times (n/Q)$. There will be $N=Q^3$ submatrix multiplications.
- $L1_i$ is loaded with two submatrices A_{jk} and B_{kl} from the L2 cache during the submatrix multiplication.
- Each P_i completes the submatrix multiplication, stores the result submatrix Y_{jkl} into the cache $L_{N+j+l-1}$, and then sends a finish response to P_0.
- After P_0 receives N finish response messages, P_0 sends a start request to each hardware thread P_{N+i}, $1 \leq i \leq Q^2$, so that P_{N+i} can perform the final sum over Q different submatrices Y_{jkl} kept in cache $L_{N+j+l-1}$ to compute C_{jl}.
- At the end of its computation, each P_{N+i}, $1 \leq i \leq Q^2$, sends a flush request to L_{N+i} so that all dirty lines of C_{jl} in this L1 cache are written back to the L2 cache.
- P_{N+i}, $1 \leq i \leq Q^2$, waits for a flush acknowledgement from its cache, and then sends a finish response to P_0. Once P_0 receives Q^2 finish messages more, the matrix-matrix multiplication is done.

Different from case A and case B, the matrix-matrix multiplication implementation requires the use of all hardware components shown in Figure 1. Furthermore, it features two level of fork-join parallelism where the different number of threads are working on different tasks at each level.

IV. Conclusions

A parallel hardware architecture for a class of parallel applications that can be modeled by a fork-join programming model, such as OpenMP, is introduced. Its features are further highlighted on three different case studies.

Future work involves devising a compiler to generate such parallel hardware from regular OpenMP applications; measuring and reporting the performance that can be attainable by the generated parallel hardware using a set of benchmark OpenMP applications, and making this compiler to support the most of OpenMP constructs.

References

[1] B. Chapman, G. Jost, R. van der Pas, Using OpenMP Portable Shared Memory Parallel Programming. London, UK: The MIT Press, 2008.

[2] J. Choi, St. Brown, and J. Anderson, “From software threads to parallel hardware in high-level synthesis for FPGAs,” International Conference on Field-Programmable Technology (FPT’13), IEEE Press, Dec. 2013, pp. 270-277, doi: 10.1109/FPT.2013.6718365.

[3] Y. Y. Leow, C. Y. Ng, and W.F. Wong, “Generating hardware from OpenMP programs,” IEEE International Conference on Field Programmable Technology, (FPT 2006), IEEE Press, Dec. 2006, pp. 73-80, doi: 10.1109/FPT.2006.270297.

[4] A. Cilardo, L. Gallo, and N. Mazzocca, “Design space exploration for high-level synthesis of multi-threaded applications,” Journal of Systems Architecture, vol. 59, pp. 1171-1183, Nov. 2013, doi: 10.1016/j.sysarc.2013.08.005.

[5] A. Podobas and M. Brorsson, “Empowering OpenMP with automatically generated hardware,” International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), IEEE Press, Jul. 2016, pp. 201-205, doi: 10.1109/SAMOS.2016.7818354.

[6] L. Sommer, J. Korinth, and A. Koch, “OpenMP device offloading to FPGA accelerators,” 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2017), IEEE Press, Jul. 2017, pp. 201-205, doi: 10.1109/ASAP.2017.7995280.

[7] L. Sommer, J. Oppermann and A. Koch, “Synthesis of interleaved multithreaded accelerators from OpenMP loops” International Conference on ReConFigurable Computing and FPGAs (ReConFig), IEEE Press, Dec. 2017, 10.1109/RECONFIG.2017.8279823.

[8] Xilinx SDAccel. [Online]. Available from https://www.xilinx.com/products/design-tools/softwarezone/sdaccel.html/ 2018/06/08.

[9] K. Ebcioglu, E. Kultursay, and M. T. Kandemir, “Method and system for converting a single-threaded software program into an application-specific supercomputer,” US8,966,457B2, 2015.

[10] T. Bjerregaard and S. Mahadevan, “A survey of research and practices of network-on-chip,” ACM Computing Surveys, vol. 38, pp. Jun. 2006, doi: 10.1145/1132952.1132953.