Bridging a Data-Flow Execution Model to a Lightweight Programming Model

Roberto Giorgi and Marco Procaccini

Department of Information Engineering and Mathematics
University of Siena
Siena, Italy
{giorgi,procaccini}@diism.unisi.it

Abstract—Starting from a Data-Flow execution model called “DF-Threads”, we defined a minimalistic API to enable an efficient implementation in the hardware of the distribution of the threads across the cores of a single multi-core system and across the remote cores of a cluster. We aim at proposing this API as a simple programming model in C language that can potentially permit an easy interface between DF-Threads and generic programming models. Clusters are typically programmed with MPI, therefore we evaluated our approach against OpenMPI. If we consider the delivered GFLOPS per core, DF-Threads are also competitive in respect to CUDA. In the basic examples, that we used in this initial investigation, DF-Threads achieve better performance-per-core compared to OpenMPI and CUDA. In particular, OpenMPI has a large portion of OS-kernel activity, which is slowing down its performance.

Index Terms—Performance evaluation, Computer architecture, Computer simulation, Matrix Multiply, Distributed computing, High performance computing, Data-Flow computing

I. INTRODUCTION

The original motivation for research in Data-Flow computing was the possibility of exploiting of its massive parallelism [1], [2]. The performance scaling of the processors has basically followed the path of designing deeper pipelines, increasing clock rates and number of cores in a chip. However, when the single chip performance is not any more sufficient, a distributed architecture becomes an interesting solution. In such case, the full parallelism has not yet been completely exploited due to both execution model and programming model limitations [3], thus creating need for more research.

The Data-Flow execution model is capable of taking advantage of the full parallelism offered by a multi-core and multi-node systems [4]–[13] by introducing a new paradigm, which internally represents applications as a direct graph named program Data-Flow graph. Applications are represented as a set of nodes, where each node may represent an instruction or anything else (according to Data-Flow principles [2]). Directed arcs between nodes represent the data dependencies between the nodes. Whenever all inputs are available, the node is ready for firing. This stands in contrast to the Von Neumann execution model, in which an instruction is only executed when the program counter reaches it, without considering if it can be executed earlier or not. The key advantage is that, in Data-Flow, more than one instruction can be executed at once (in fact superscalar processors exploit internally this Data-Flow principle through the dynamic scheduling of instructions). Thus, if several nodes become ready at the same time, they can potentially be executed in parallel. This simple principle provides an opportunity for massive parallel execution. Past attempts to design an entire machine based on that principle where not successful (e.g., Manchester Data-flow Machine, Explicit Token Store architecture [14], [15]) mainly due to the too fine grain of the approach, i.e., at instruction level. In the context of the AXIOM project [16]–[22], we explored the feasibility of a novel Data-Flow execution model (Data-Flow Threads or DF-Threads [23]) in a heterogeneous and distributed environment, prototyped on our own FPGA-based boards [21]. DF-Threads allows us to offload portions of code to a hardware engine in order to achieve a better scalability than a software scheduler [24]–[27].

In a first instance, we explored the design space by using the HPLabs COTSOn simulator [28] to find the most efficient implementation of the execution model. After that, we tested the DF-Threads on a cluster of AXIOM-Boards (Figure 1) [29]. The AXIOM-Board has several low power cores and an FPGA. Moreover, we can build a complete distributed system, in which AXIOM-Boards can be connected through inexpensive high-speed custom USB-C cables, reaching up to 18 Gb/s per channel (and having four available channels).

Fig. 1: Architecture of the distributed system based on the AXIOM boards. The Distributed System consists of N Nodes based on an FPGA SoC, which includes a Processing System (PS) and Programmable Logic (PL). The nodes of the system are connected through USB-C cables without the need of an external switch.
This paper makes the following contributions:
- It gives an example of how to translate a recursive generic program into Data-Flow programming style.
- It provides an initial quantitative comparison of DF-Threads, OpenMPI and CUDA.

In the Section II we briefly describe the lightweight Data-Flow programming model; in Section III we illustrate the methodology used for the experiments; in Section IV we evaluate our Data-Flow execution model, and finally, we conclude the paper.

II. A LIGHTWEIGHT DATA-FLOW PROGRAMMING MODEL

The DF-Threads execution model relies on the program Data-Flow graph, in which each node of the graph represents a fine-grain thread named DF-Thread. The execution of the DF-Threads follows the producer-consumer paradigm, in which a DF-Thread (consumer) can execute only when all its inputs have been produced by other DF-Threads (producers). The lifetime of a DF-Thread is defined by 4 API calls (potentially instructions) [30], which are briefly recalled in Table I. With additional instructions, it is also possible to subscribe portions of shared memory for collective operations.

For the purpose of easier mapping generic program code, we elevated this API to a lightweight programming model, in which DF-Threads are simple C functions without the need of using the stack for passing parameters and with the addition of Data-Flow semantics. A DF-Threads can therefore be implemented as illustrated in Figure 2 with explicit management of the input frame (where input data is stored) in coordination with the linker. In Table I we give the semantics of the df ldframe and df destroy typically placed respectively at the beginning and end of the DF-Thread:

```c
void a_df_thread (void) {
    df_ldframe ()
    <df_thread_body>
    df_destroy ()
}
```

Fig. 2: A DF-Thread in C language.

As an example, we show here how we can translate the Recursive Fibonacci code into a DF-Thread (Figure 3). In this case, we map the original code (left) into two DF-Threads named fibo and adder (right). The df_schedule defines how many inputs the next instances will receive. The df_write fills up the input frames of next instances. As soon as all inputs of the target thread have been written, the target thread is executable. At the end, the DF-Thread notifies that its metadata is stored) in coordination with the linker. In Table I we give the semantics of the df ldframe and df destroy typically placed respectively at the beginning and end of the DF-Thread:

```c
int fibo (int n) {
    if (n <= 1) return n;
    return fibo(n-1)+fibo(n-2);
}
```

In a preliminary phase, we defined the DF-Threads execution model into a customized version of the HP Labs COTSon Simulator [31]-[34], which permits us to decouple the functional execution from the timing behavior for an easier modeling. Thanks to COTSon simulator, we can model a complete distributed system with many-cores and multi-nodes, in which it is possible to run an off-the-shelf Linux Distribution for a full system simulation [35]. We compared our implementation with: i) OpenMPI, a typically used programming model for clusters and ii) CUDA as it is another widely used solution for performance scaling.

We also needed to design supporting tools to reduce the experimentation time from days/weeks to hours/minutes [36].

In order to have a realistic base for the timing, we also validated our simulations against the timing obtained on the AXIOM-boards, where we gradually migrated the designed Intellectual Property (IP) blocks (Figure 1). The AXIOM platform includes four 64-bit ARM Cortex-A53 cores (at 1.5 GHz); 32 Kib L1 Cache and 1Mib of L2 Cache, programmable logic and fast transceivers.

The Processing Systems (PS) of the AXIOM-board, runs a full Linux Ubuntu 16.04 and it starts the program, while the accelerated portions are offloaded, via the programming
model illustrated in Section II to the Programmable Logic (PL). The PL also includes a Network Interface (NI) that allows the fast communication among the AXIOM-boards. At the hardware level, the soft-IPs are also responsible to distribute the workload among the nodes of the Distributed System and manage the metadata of the DF-Threads. For the CUDA experiments, we used the Tesla-C1060 board, with 240 CUDA core, 610 MHz GPU clock and 4 GiB of RAM.

IV. EXPERIMENTAL RESULTS

For the sake of this initial exploration, we consider two simple benchmarks for the evaluation of the DF-Threads: Recursive Fibonacci and Matrix Multiplication.

- **Recursive Fibonacci** (RFIB) benchmark has been chosen to stress the thread management and quickly evaluate the performance, while there is a need of scheduling many threads. This benchmark takes as input the \( n \) of Fibonacci and a threshold which stops the generation of the parallel recursive calls.

- **Matrix Multiplication** (MM) benchmark involves more memory operations than RFIB and also is a widely used kernel in machine learning. We used a blocked Matrix Multiplication implementation, where a matrix is partitioned in multiple sub-matrices, or blocks, according to the block size that is set. We use square matrices with 448 as size (to avoid a multiple of a power of two, which may cause multiple cache conflicts on a few cache lines) and with 8 as block size. We focus our measurements only on the computational Region of Interest (ROI) of the benchmark as it is the usual practice in comparisons. The Matrix Multiply algorithm used to evaluate OpenMPI and CUDA is the standard available version for such programming models. Results of the benchmarks are checked for correctness at the end of the run. Multiple runs (at least 5) have been also repeated to reduce possible statistical oscillations.

A. **Recursive Fibonacci**

We evaluate the Recursive Fibonacci benchmark with an input size of 35 and 13 as threshold, by varying the number of nodes of the distributed system up to 16. As can be seen in the Figure 4, DF-Threads show a good degree of scalability. The results confirm that the scheduler of the DF-Threads can handle and distribute properly many fine-grain threads among multiple nodes. CUDA and OpenMPI have been not evaluated with the Fibonacci benchmark due to their poor effectiveness with recursive execution on such platforms.

B. **Block Matrix Multiply**

We use the GFLOPS/core metric to compare the performance of the DF-Threads with OpenMPI and CUDA, due to the currently lower number of cores of our Distributed System in respect of CUDA. The matrix size is 448 with 8 as block size. As we can see in the Figure 5, the GFLOPS/core of the DF-Threads outperforms both CUDA and OpenMPI.

OpenMPI is outperformed by a large factor of about 5.5x in the case of 1 node (1N) or about 140x in the case of 16 nodes (16N). This is due to several factors: first of all the OpenMPI runtime library represents a wide middleware layer; secondly, there is the need of invoking system calls that in turn may need a time consuming operating system activity to move content buffer and manage the send and receive operations on the physical media. In the DF-Threads such overheads are reduced, thanks to the simpler interface and the hardware management of the data frames and thread metadata. As depicted in Figure 6, the kernel activity of the DF-Threads is quite limited in comparison with OpenMPI.

CUDA is outperformed by almost a factor of 1.7x among all configurations of the distributed system, due to the efficiency of our scheduling mechanisms. The CUDA platform that we used is one of the first Tesla boards available on the market, but our implementation of DF-Threads is also at the first version and it is not yet optimized. Therefore, the DF-Threads is capable to exploit better the resources of the distributed system, also thanks to the parallelism exposed by the Data-Flow mechanism and the DF-Thread API.

Moreover, we think that there is still much space for further optimization of the DF-Threads. For example, the data locality could be improved and we are investigating a pre-fetching policy, which could load the data into the cache before starting the execution of the DF-Threads.
V. CONCLUSIONS

The Data-Flow execution model is a viable paradigm to be explored today to achieve high degree of parallelism in the modern many-cores multi-nodes architectures. This paper presented how the DF-Threads execution model can bridge the Data-Flow execution to a simple C-based programming model through the DF-Threads API. Our experiments show the capability of the DF-Thread execution model to distribute and manage many fine-grain threads among multiple nodes. We compared the DF-Thread with OpenMPI and CUDA by using the block Matrix Multiplication benchmark and we found that DF-Thread outperform both OpenMPI and CUDA in terms of GFLOPS/core. Future work will expand the capability of automatic translation and demonstrate a larger set of benchmarks.

ACKNOWLEDGMENT

The work of this paper is partly funded by the European Commission on AXIOM H2020 (id. 645496). TERAFLUX (id. 249013), HiPEAC (id. 779656). The authors would like to thank the anonymous reviewers for their helpful comments.

REFERENCES