Accelerating MPI_Reduce with FPGAs in the Network

Extended Abstract

Joshua Stern
ECE Department, Boston
University

Qingqinq Xiong
ECE Department, Boston
University

Jiayi Sheng
Falcon Computing Solutions, Inc.

Anthony Skjellum
University of Tennessee,
Chattanooga

Martin C. Herbordt
ECE Department, Boston
University

ABSTRACT

MPI collective operations can often be performance killers in HPC applications, especially ones that require both heavy communication and computation such as MPI_Reduce. Using FPGAs, which provide the ability to couple communication and computation, we have designed a hardware accelerator to implement MPI_Reduce in the network. With our design, preliminary results show that we can achieve a 2.5× speedup for long messages and a 10× speedup for short messages over conventional clusters.

KEYWORDS

FPGA, MPI Collectives, offload

1 INTRODUCTION

In most high performance computing applications, collective operations play a vital role in determining overall performance. These collectives along with a myriad of other distributed network operations have all been brought together under the Message Passing Interface (MPI) [2]. As collectives have become central components of the MPI programming model, there has been an abundance of support added at the software level. This includes new algorithms that can improve the performance of collectives such as broadcast and reduction by optimizing them either for low latency with small data sets or for high throughput when dealing with large arrays [17]. On the other hand, this has increasingly complicated the software stack, which we seek to simplify by removing these extra layers and increase performance by implementing MPI collectives in hardware.

Moving these operations into hardware has already shown significant performance benefits such as in the case of the IBM Blue Gene Supercomputer, which contains special hardware support for collectives [8]. Other projects, such as [9] and [7] use InfiniBand multicast support to get considerable improvements in latency reduction and throughput increase. However, one of the drawbacks of their implementations is its lack of flexibility in the types of operations it can perform and data types it can handle. Field Programmable Gate Arrays (FPGAs) provide this flexibility with reconfigurable logic along with low latency, high throughput, and a high degree of parallelism, which makes them ideal for implementing MPI collectives.

The purpose of this project is to design an intelligent network architecture to support MPI_Reduce on FPGAs in the network that can outperform traditional computer clusters. This tight coupling of computation and communication provides great potential in improving the performance of numerous other MPI collectives. We chose to focus first on MPI_Reduce because of its popularity in HPC applications and potential for acceleration. By using FPGAs to take advantage of the time that the data is in the network and computing in flight, our early simulations show up to 10× speedups for MPI_Reduce over the OSU benchmark for small messages. As this project progresses, the speed, flexibility, and portability that FPGAs bring will allow us to extend our design to other MPI operations.

2 MPI_FPGA CONCEPTS

2.1 Software Stack

Our design, which we call MPI_FPGA, assumes complete transparency with MPI middleware. This makes MPI_FPGA portable so that it can be easily integrated into HPC applications without requiring the programmer to have any knowledge of the hardware being used. Instead, constructs automatically access MPI_FPGA capabilities through enhanced middleware. MPI_FPGA makes no assumptions about the types of end systems being used, as it is only affecting the data as it is being routed through the FPGAs in the network. At the MPI_Reduce software layer, we create a new function, MPI_FReduce. This eliminates all of the computation and communication work done in MPI_Reduce and creates a simple reduction message for the network to complete the reduction. By doing so, we avoid a large portion of the software stack that includes MPI point-to-point messages. In terms of the MPICH implementation of MPI middleware [6], all of the functionality of the ADI is maintained. We are currently using MPICH-3.2 to design MPI_FPGA [5]: tasks such as packetizing and computing the predefined reduction operations will be performed identically in our design. At the channel interface of MPICH we add in FPGA communication code that transfers data into the FPGA network. The actual FPGA hardware sits below the channel interface but is responsible for completing...
the computation and communication specified in the above levels, which consist of the reduction algorithm and the underlying point to point message passing (see Figure 1).

2.2 Related Work

In the work presented in [1] on offloading collective operations onto programmable logic, many of concepts are similar to those in our work. However, one of the main differences is that their hardware does not perform any reduction operations until a node receives data from all of its children nodes. In our work, the reduction module begins performing the reduction operation as soon as data is received. This means that when data is received from the final child node, only one operation needs to occur before the final result is ready to be returned to the router. In [3], they also implement a reduction core architecture, but one that is quite different from ours. Their reduction core consists of floating point cores with two input FIFOs, and one output FIFO. The results of the output FIFO can then be looped back as inputs to the reduction core in cases where a node must collect data from multiple children. This architecture is simpler than ours and consumes less FPGA resources, but lacks flexibility in its reduction capabilities. Our design can support any order of multiple reductions occurring simultaneously, whereas their design can only support one reduction at a time. In the reduction hardware support shown in [10], data from all children must be received before a reduction operation can start. This is not the case in our design, in which reduction operations are started as soon as data is received. Also in their design, only one reduction can take place at a time. In [12], they implement their own version of MPI known as TMD_MPI, in which both the hardware logic and software is entirely implemented on FPGAs. Our architecture is designed to be integrated into an existing version of MPI, MPICH-3.2, and can be integrated into other implementations as well.

3 HARDWARE DESIGN

3.1 Implementation

This section describes the different components of the hardware architecture, which are displayed in Figure 2. When packets enter the router from the Multi-Gigabit Transceivers (MGTs), if they are reduction packets, they are transferred to the reduction module. Once a host sends a reduction packet into the router, it is not responsible for any other work other than receiving the final result of the operation. In the reduction module, the reduction instruction decoder receives packets from the router and implements the functionality of the MPI_Reduce operation by making all of the necessary algorithmic and routing decisions.

The two algorithms currently supported are a binary tree algorithm and Rabenseifner’s algorithm, which were chosen because they are the two algorithms implemented in the most recent version of MPICH [5]. For small to medium size reductions, the binary tree algorithm will be used because of its low latency. For large reductions, Rabenseifner’s algorithm is preferred because of its high throughput. These algorithms are both suitable for our architecture because they give us the ability to compute in the network, rather than have every node send data directly to the root node. The two main algorithmic decisions made in this module are about how many children the current node should wait for data from and the rank of the parent node to which it will forward its results to. By essentially performing the software version of MPI_Reduce, we avoid the large software overhead. Results of the decoder are immediately passed to a FIFO, where they reside until the reduction computation unit is ready to read them.

The reduction computation unit consists of a reduction table, which is indexed and capable of supporting multiple reductions simultaneously. Once a reduction packet enters the from the FIFO, the control logic examines the packet header and places the data in the appropriate table slot. If the reduction table slot that matches the incoming packet’s index is empty, then the incoming packet is simply copied into the reduction table slot. For every following packet with the same index that enters the unit, if the reduction table slot that matches the index of the incoming packet is not empty, then the data payload of the incoming packet and the data
already contained in the reduction table are combined, and the result is later fed back into the reduction table entry. This process is displayed in Figure 3. If the incoming packet is supposed to go in a table index that has a corresponding operation currently occurring in the floating point unit, then the computation unit cannot read any new packets from the input FIFO until the corresponding reduction operation is complete. We use Altera IP floating point cores in our design and focus on single precision floating point addition due to its simplicity; performance is virtually identical for double precision (see below). Our design also supports multiplication, maximum, minimum, and is flexible enough to support any other IP cores that Altera offers or that we would construct.

By designing the reduction table to have multiple entry slots, we can support multiple reductions taking place at the same time. If the reduction is of a large data set and needs to be divided into smaller reductions, each will occupy a different index of the reduction table. By having the floating point unit be pipelined, these multiple reductions can all be computed together. This unit can also support separate unrelated reductions, and is flexible enough to allow any reduction packet to be placed in any reduction table entry slot at any time. If the reduction table has 100 entry slots, then this module can support up to 100 different reductions occurring together, all possibly performing different types of reduction operations.

Each reduction table entry slot keeps track of the number of child nodes for any given reduction. Whenever an incoming packet enters the module, the reduction table slot records this and keeps track of the number of children nodes remaining. When a reduction table entry has received packets from all child nodes, then the reduction is complete and new packet is built and returned to the router. The router is then notified that a reduction has been completed and passes the result of the local reduction to the router of the parent node. If the current node is the root node, then rather than being passed to a parent node, the result of the reduction is passed back to the host processor, thus completing the reduction.
In this preliminary communication we simulate 8-, 16-, and 32-node systems with a node from the Novo-G6 cluster. Each target node contains a router that was previously designed by our group [14, 15] along with our new reduction module. We use Multi-Gigabit Transceivers (MGTs), which provide high bandwidth and low latency inter-FPGA interconnections. To simulate their communication latency, we implemented an equivalent communication delay in our design. We simulated our system on a ProceV FPGA board from Gidel with an Altera Stratix V SGSM8. We used a clock set to 150 MHz and ran MPI_Reduce for single precision floating point numbers, with the reduction operation set to MPI_SUM. We use the binary tree algorithm because our tests focus on small to medium sized messages. Since the design is fully pipelined, the operating frequency also represents the throughput, which is therefore independent of the operation being executed and its precision. Note that the 150 MHz is for unoptimized logic; at least double that is likely for a production implementation.

In Figure 5, we show the performance of MPI_FPGA for small to medium array sizes on the 8-node, 16-node, and 32-node systems and compare our results against the OSU benchmark, a well-known benchmark for MPI operations. We see that for small array sizes, MPI_FPGA achieves much better performance. We also see that the MPI_FPGA speedup over the OSU benchmark is much more noticeable in the 16- and 32-node simulations, thus leading us to believe that MPI_FPGA is scalable and would continue to achieve greater speedups as the number of nodes is further increased. In the 8-node simulation, we reached speedups as high as 6× over the benchmark, and this speed-up rises to 10× on the 32-node simulation. As the array size grows, the MPI_FPGA speedup begins to diminish. This is expected because of the low clock rate of FPGAs. The obvious and simple solution is multiple ALUs, which can be added at minimal cost for most operators.

The reduction module without any of the IP cores consumes 12,563 ALMs, which accounts for about 5% logic utilization of the Stratix V FPGA. For the IP cores themselves that are used for the computation, the resources required are typically much less than 1% of the ALMs per core. MPICH-3.2 has 12 predefined reduction operations, so for a full implementation, we would need at most 12 IP cores. This would amount to no more than 5% of additional ALMs needed. The router that we use consumes an additional 21% of the logic resources, bringing our total resource utilization to about 26%. However, a Stratix 10 FPGA has over four times as many logic resources than a Stratix V, meaning that our reduction module together with the router would only consume about 5% of the FPGA resources.

4 SUMMARY AND FUTURE WORK

In this extended abstract, we have described the current state of our efforts to design an accelerator on FPGAs that can be used in the network to improve the performance of MPI_Reduce. Early results from our simulations show significant speedups over the OSU_reduce benchmark. As this is an ongoing project, we are continuously optimizing our reduction module and hope to soon be able to evaluate our design on an actual FPGA cluster. This would allow us to analyze how our design the scalability of our design as the number of nodes and the message size is increased.

We also aim to extend our network architecture towards supporting other popular and computationally heavy MPI collectives such as MPI_A1lreduce over large groups of processes.

REFERENCES