Challenges in Compilation of Brook Streaming Programs for FPGAs

Franjo Plavec Zvonko Vranesic Stephen Brown
University of Toronto
Department of Electrical and
Computer Engineering
10 King's College Road,
Toronto, Ontario, Canada
{plavec, zvonko,
brown}@eecg.toronto.edu

ABSTRACT

Streaming languages have recently been proposed as a suitable paradigm for programming large-scale multiprocessor systems based on streaming processors. They have also been found to be suitable for programming traditional multiprocessors and graphics processing units (GPUs). In this paper we discuss suitability of the Brook streaming language for implementation in FPGAs. We present challenges in compiling Brook programs to FPGA logic and show that both data and task-level parallelism can be efficiently exploited.

1. INTRODUCTION

As the digital systems being implemented in FPGAs grow in complexity, there is an increasing need for methodologies that allow designers to implement their systems at a higher abstraction level. As an example, behavioral synthesis compilers can convert a program written in a high-level programming language into an HDL suitable for FPGA implementation. A class of high-level languages that is attracting a lot of attention lately is based on the streaming paradigm. In streaming, data is organized into streams, whose elements are guaranteed to be mutually independent. Kernels are functions used to manipulate streams. A kernel implicitly operates on all elements of its input streams. Depending on the implementation, a kernel can process multiple stream elements in parallel, because the elements are independent. In this manner, data parallelism can be exploited. If there are multiple kernels in the program, the kernels can operate in parallel, thus exploiting task parallelism.

In this paper we discuss how streaming programs can be implemented in FPGA logic in a way that exploits available data and task parallelism. We propose a methodology that converts a program written in the Brook streaming language [1] into C code suitable for behavioral compilation using the C2H compiler from Altera. Kernels are converted into hardware blocks that operate on incoming streams of data. We chose the Brook streaming language from Stanford, because it is an extension of the C programming language, which may aid in its adoption by system designers. We show that a program expressed in Brook can be automatically converted into parallel hardware, which executes significantly faster than software.

The rest of the paper is organized as follows. In section 2 we present basic features of the Brook streaming language. Our approach to implementing Brook programs in FPGAs is described in section 3. Section 4 discusses how parallelism in streaming programs can be exploited.

2. BROOK STREAMING LANGUAGE

Streaming applications are defined through a set of kernels, which define the computation part of an application, and a set of data streams, which define communication. In Brook, kernels are denoted using the *kernel* keyword, while streams are declared similarly to arrays, except that characters "<" and ">" are used instead of square brackets. The following code section demonstrates two different types of kernels supported by Brook.

```
kernel void mul (int a<>, int b<>, out int c<>) {
 c = a*b; }
reduce void sum (int a<>, reduce int r<>) {
 r = r+a; }
```

The first kernel (*mul*) is an ordinary kernel, which multiplies elements of the two input streams to produce an output stream. Although the code refers only to streams, not individual elements, it is understood that all elements should be multiplied. This is only possible if all streams have equal sizes.

The second kernel (*sum*) is a reduction kernel, used to define reduction operations. Reduction operations have to be commutative and associative, so that the ordering in which the elements are reduced does not affect the result. Relative sizes of the input and output streams define how the reduction operation is performed. In the example above, if the output stream (r) has only one element, all elements of the input stream (a) will be added to produce the output. The operation is slightly different if the output stream has more than one element. For instance, if the input stream has 20 elements and the output stream has 4 elements, then 5 consecutive elements of the input stream are added to produce one element of the output stream. This feature imposes some ordering on the operations and has a significant impact on generating hardware for reduction operations.

3. IMPLEMENTING STREAMING PROGRAMS IN FPGAS

We believe that streaming programs can be efficiently implemented in FPGAs, because FPGA's programmable logic blocks are suitable for implementation of parallel computation. Also, FPGAs are easily reprogrammable, so the generated hardware can be tailored to the needs of a specific application.

When mapping streaming applications to FPGA logic, it is a natural choice to map each kernel to a processing node. The processing nodes could be soft-core processors, specialized streaming processors, or custom-generated hardware blocks. We believe that generic soft-core processors are not a good choice for implementing kernels, because they can only communicate with other processors through data bus, which may become a bottleneck if a kernel has many inputs and outputs. Therefore, we believe that specialized streaming processors or custom-generated hardware blocks should be used. Depending on throughput requirements, a processing node could implement several kernels, or one kernel may be distributed over several processing nodes.

Processing nodes need to communicate streams among themselves to implement streaming applications. This communication has to be fast to support high throughput required for many applications. Therefore, off-chip memory should be avoided and FPGA on-chip memory should be used whenever possible. Since the amount of on-chip memory is limited, for many applications it may not be possible to keep all streams in on-chip memory. Instead, we propose using shallow FIFO buffers for passing stream elements between kernels. As new data arrives from outside the FPGA, it is processed by kernels and passed between them through FIFO buffers. Once the last kernel processes the data, it passes the result outside the FPGA. This way, only a limited number of stream elements are kept in on-chip FIFO buffers. FIFO buffers are used instead of simple registers because they provide buffering for cases when execution time of a kernel varies between the elements. If one kernel takes a long time to process one element, the next kernel downstream could become idle if there was just one register between them. Using FIFO buffers, the second kernel can process data from the FIFO. As long as the first kernel delivers the next element before the FIFO buffer becomes empty, the second kernel will not have to stall.

Instead of compiling Brook code directly into an HDL, we take advantage of an existing behavioral synthesis tool (C2H from Altera). We modified an existing Brook compiler from Stanford to generate C code suitable for compilation using C2H. In addition to the C code, our compiler also generates other files which describe the complete SOPC system, including the necessary FIFO buffers and other components. We also generate custom scripts, which are used to automatically produce the programming file for the target FPGA.

4. EXPLOITING PARALLELISM

In streaming programs, communication is explicitly defined through data streams. This makes it easy for compilers to analyze programs, thus allowing parallelism to be exploited. When a programmer specifies that certain data belongs to a stream, this provides a guarantee that the elements of the stream are independent from one another. Kernel computation can be applied to stream elements in any order. In fact, all computation can be performed in parallel, limited only by the available hardware.

FPGA implementation of a streaming program as described in the previous section exploits task parallelism by running kernels in parallel in a pipelined fashion. Data parallelism can be exploited by designing processing elements that can process multiple stream elements in parallel. A simple way to achieve this is to fully replicate the functionality of the kernel. For instance, if a custom hardware block implementing a kernel does not provide sufficient throughput for a given application, two identical blocks could be generated, potentially doubling the throughput. This is possible because stream elements are independent, and because FPGAs are reprogrammable, so we can generate a system specifically for a given application.

The above replication procedure is relatively straightforward for ordinary kernels. However, the situation is more complicated for reduction kernels. To replicate a reduction kernel, the compiler has to build a reduction tree, where the number of elements to be reduced decreases as we progress from leaves to the root of the tree. Previous work has mostly focused on a reduction where the result is only one number. Since Brook also supports reductions where the output has more than one element, this complicates replication of reduction kernels. If such a reduction is to be replicated, the reduction tree has to be carefully designed, and individual replicas have to keep track of the numbers of stream elements processed to perform the reduction correctly. This is challenging because it has to be done automatically by the compiler for streams of arbitrary sizes.

5. CONCLUDING REMARKS

We implemented a source-to-source compiler that converts Brook applications into C code suitable for C2H compilation and we compiled two small applications (FIR filter and Autocorrelation) using this system. The compiler does not yet support full automatic replication, so we replicated critical kernels manually to measure the effect of replication on performance. We found that Autocorrelation benefits from replication and achieves performance improvements close to the theoretical maximum for replication factors 2 and 4. FIR filter also achieves double throughput for replication factor 2, but only 2.4X throughput for replication factor 4. We also found that our replicated implementations performed up to 8.9X better than software running on a soft-core processor and up to 4.3X better than the same software compiled using ordinary C2H flow [2]. We are currently working on providing full support for kernel replication. We also plan to build several large applications to demonstrate usability of our approach for real-world applications.

6. REFERENCES

[1] I. Buck. Brook Spec v0.2, October 2003. Technical Report CSTR 2003-04 10/31/03 12/5/03, Stanford University.

[2] F. Plavec, Z. Vranesic, and S. Brown. Towards Compilation of Streaming Programs into FPGA Hardware. In *Forum on Specification and Design Languages (FDL '08)*, 2008.