FPGA Circuit Synthesis of Accelerator Data-Parallel Programs

Barry Bond, Kerry Hammil, Lubomir Litchev
Microsoft
1 Microsoft Way,
Redmond, Washington 98052, USA
{barrybo, khammil, lubol}@microsoft.com

Satnam Singh
Microsoft
7 JJ Thomson Avenue
Cambridge CB3 0FB, United Kingdom
satnams@microsoft.com

***Abstract*— This paper describes the techniques used to describe and synthesize FPGA circuits expressed in a data-parallel domain specific language (DSL) called Accelerator. We identify the subset of data-parallel descriptions that are supported by our system and explain how we track memory access patterns which allow us to generate efficient FPGA circuits.**

I. Introduction

For a significant class of users it is highly desirable to describe data-parallel programs using a regular programming language which also permits the possibility of automatically transforming these programs into FPGA circuits for use in FPGA-based co-processors. Two broad categories of users find such systems appealing. The first category is users that are not digital designers with a detailed knowledge of FPGA architecture but instead are software engineers or scientists (e.g. biologists and physicists) that have a secondary competency in programming and have computationally intensive problems to solve that can benefit from FPGA-based co-processing. The second category is digital designers that need high level techniques to improve design productivity and examples of such users are DSP engineers that currently employ systems like MATLAB/Simulink as a design entry technique for hardware based synthesis flows.

We argue that for certain kinds of data-parallel programming tasks it is possible to use an existing language like C++ and regular compilers like Visual Studio and GNU GCC to describe data-parallel computations and automatically generate VHDL circuit netlists and associated design flow files (e.g. to create instances of floating point cores) without requiring a special compiler. Furthermore, because these data-parallel descriptions are regular C++ programs they can be executed, debugged and analyzed using regular tools found in modern IDEs. The users of such systems do not need to learn a new language or tool because they can reuse their existing knowledge of C++ and its associated compilers and tools.

The data-parallel system that we use as the starting point for generating FPGA circuits was originally designed to target GPGPU programming and is based on a model that dynamically generates code by JIT-ing. This approach to GPU code generation is online because the user does not need to compile the data parallel kernels separately and then link them with the rest of the system. This system was extended to generate SSE4 vector instructions to target X64 multicore processors and this target is also based on online JIT-ing. The FPGA target that we describe in this paper does not JIT and is an offline system because it generates VHDL code and scripts for Xilinx's Core Generator system which require further processing by vendor tools and incorporation into a complete co-processing system before the result of a computation can be obtained. This is unfortunately due to the very slow speed at which circuits are placed and routed which forces us to use an offline approach.

One of our goals is to develop technology to allow us to design, model, implement and verify software and hardware for future heterogeneous manycore processors as suggested by Figure 1.

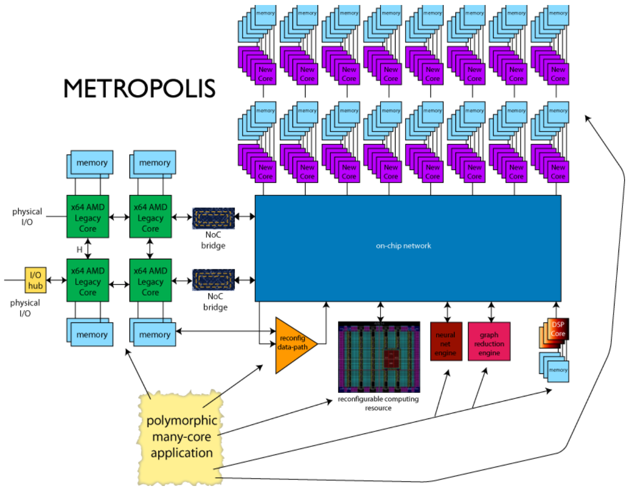

Figure 1. Heterogenous Manycore Processors

A distinctive aspect of our system is that we can compile the same data parallel description to GPGPU code, SIMD SSE4 vector code running on multiple X64 processor cores and FPGA circuits as shown in Figure 2. The starting point for each of the compilation flows in this figure are fairly abstract data parallel descriptions expressed in terms of data-parallel arrays and data-parallel operations over data-parallel arrays. Important information about memory access patterns is also expressed at a high level e.g. accessing memory in order, in reverse order, in a transposed order, with a stride, along columns, along rows etc. This information is exploited by each target to optimize the layout of data in physical memory and to also optimize the instructions and address generators to efficiently stream data without unnecessary re-reading of data-values.

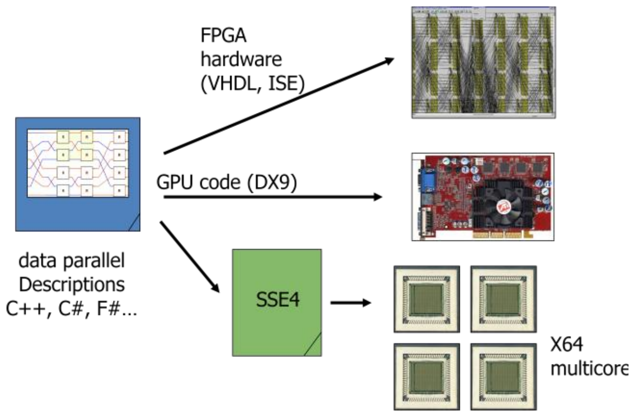

Figure 2. Single description, multiple targets

II. Overview of Accelerator

To help explain the programming model used in Accelerator we show a very simple program and identify key aspects of the Accelerator programming model. The program below is the complete source code in C++ for adding two arrays point-wise using a GPU rather than a regular processor.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "accelerator.h"
#include "DX9Target.h"

#include <iostream>

using namespace ParallelArrays;
using namespace MicrosoftTargets;
using namespace std;

int main()
{
  Target &tgtDX = CreateDX9Target();

  const int size = 5 ;
  float f1[size] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f} ;
  float f2[size] = {0.1f, 0.2f, 0.3f, 0.4f, 0.5f} ;

  FPA x = FPA(f1, size) ;
  FPA y = FPA(f2, size);

  FPA z = x + y ;

  float resultArray[size];
  tgtDX.ToArray(z, resultArray, size);

  for (int i=0 ; i <size; i++)
    cout << resultArray[i] << " " ;
  cout << endl ;

  return 0;
}
```

When executed this program writes out:

```
1.1 2.2 3.3 4.4 5.5
```

which is the expected result of adding point-wise the arrays `x` and `y` which are added to produce the data-parallel array `z`. This program imports the Accelerator system which is simply a library with the namespace import statement `using namespace ParallelArrays`. Data-parallel arrays have types which are distinct from regular arrays in C++ since these arrays represent data-values which can be subject to parallel operations and data-values which do not reside in the memory space of the host program but perhaps elsewhere e.g. on the memory of a GPU card or the BRAMs of an FPGA or DDR2 memory on an FPGA board.

Floating point data-parallel arrays have the type `FPA`. The data-parallel floating point array x is of type `FPA` and it is initialized by instantiating the `FPA` class with a constructor which specifies the size of the data-parallel array to be created and an array of floating point values to be used to populate the data-parallel array x.

The Accelerator system is capable of executing computations on a variety of computing resources which are called “targets”. In the program above a target that supports execution on GPUs via the DirectX9 stack. This program creates a GPGPU target by using an instance called `tgtDX` which is created with a call to `CreateDX9Target()`.

A data-parallel computation is expressed in terms of overloaded data-parallel operations over expressions which have data parallel types like `FPA`. For example, the + operator is overloaded to operate on arrays of type `FPA`:

```
FPA operator+(FPA a1, FPA a2);
```

This operator is defined to build an expression tree node which has two sub-expressions of type `FPA`. An example of an expression tree is shown in Figure 3. Accelerator essentially provides a logical data-parallel language which is embedded in a concrete language e.g. C++. The Accelerator API provides methods which allow a programmer to specify a data-parallel computation in a convenient notation using overloaded operators and calls to static methods. At run-time these descriptions build up an expression tree which is then dynamically compiled into code for execution on a GPU or a x64 multicore processor or in the case of the FPGA target VHDL code is generated.

Accelerator implements an *on-line* phased compilation system for a two level language. In the first phase the control constructs of the host concrete language are eliminated to result in a static graph which represents a data-parallel computation. This phasing restricts us to consider data-parallel computations which do not contain data-dependent loops. This still leaves a very large class of data-parallel computations which can be expressed by our system including stencil-style computations [1][5].

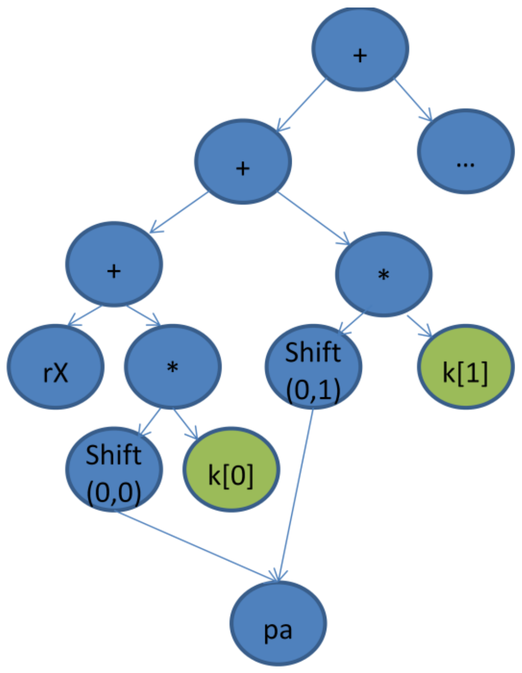

Figure 3. Expression trees using for JIT-ing

The process of JIT-ing the code, sending the input data and generated code to the GPU card, initiating the execution of the GPU code and transferring the result of the computation on the GPU back to memory in the host program is initiated here by calling the `ToArray` method. This method specified the expression graph to be evaluated (here represented by `z`), the memory in host program to which the result data should be copied (here represented by the address of the floating point array `resultArray`) and the size of the result data.

The operations supported by the Accelerator system includes several types of data-parallel arrays (floating point, integer, Boolean and multi-dimensional arrays) and a rich collection of data-parallel operations including addition, subtraction, multiplication, division, min, max etc. abs, ceiling, cosine, sine, square root, reciprocal etc., not, and, or, relational operators, sum, product, dimension changes, section, shift, stretch etc. and linear algebra operations e.g. inner and outer product.

Essentially the same program shown above can be compiled to SSE4 vector code for execution on multiple x64 processor cores:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "accelerator.h"
#include "X64MulticoreTarget.h"

#include <iostream>

using namespace ParallelArrays;
using namespace MicrosoftTargets;
using namespace std;

int main()
{
  Target &tgtMC = CreateX64MulticoreTarget(false);

  const int size = 5 ;
  float f1[size] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f} ;
  float f2[size] = {0.1f, 0.2f, 0.3f, 0.4f, 0.5f} ;

  FPA x = FPA(f1, size) ;
  FPA y = FPA(f2, size);

  FPA z = x + y ;

  float resultArray[size];
  tgtMC.ToArray(z, resultArray, size);

  for (int i=0 ; i <size; i++)
    cout << resultArray[i] << " " ;
  cout << endl ;

  return 0;
}
```

III. An FPGA Target

A circuit which performs the point-wise addition of two streams of inputs can be written in Accelerator as follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "accelerator.h"
#include "FPGATarget.h"

#include <iostream>

using namespace ParallelArrays;
using namespace MicrosoftTargets;
using namespace std;

int main()
{
  Target &tgtFPGA = CreateFPGATarget("adder", Virtex5);

  const int size = 5 ;
  float f1[size] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f} ;
  float f2[size] = {0.1f, 0.2f, 0.3f, 0.4f, 0.5f} ;

  FPA x = FPA(f1, size) ;
  FPA y = FPA(f2, size);

  FPA z = x + y ;

  float resultArray[size];
  tgtFPGA.ToArray(z, resultArray, size);

  return 0;
}
```

This code is very similar to the code for the GPGPU DX9 target and the SSE4 x64 multicore target. However, the FPGA target is *off-line* i.e. calling `ToArray` results in the generation of VHDL source code files plus .XCO files for Xilinx's Core Generator system for the instantiation of floating point cores. For this example each of the input arrays is compiled into BlockRAMs that are initialized with the corresponding binary IEEE floating point representation of the input values. The generated circuit is simply an adder floating point core which reads simultaneously from two such BlockRAMs, performs a point-wise addition and stores the results in an output BlockRAM.

The main reason the FPGA target is off-line is due to the fact that vendor place and route tools take a very long time to execute making a JIT-ing model infeasible. Furthermore details of how data is transmitted from the host computer to the FPGA system (or card) and back are not dealt with by the generic FPGA target and currently need to be dealt with manually by the user. As standard APIs are developed for host to co-processor communication we expect to be able to further abstract FPGA co-processors to the same level at which we have abstracted communication with the GPU subsystem and SSE4 vector code.

The reason we can generate efficient code for FPGA circuits is that we can exploit information in the expression graph about how data sources are access e.g. in order, in reverse, transposed, with a stride etc. Furthermore, operations describing data shifts allow us to retain previously read values for reuse later which avoids unnecessary reads. A combination of such optimizations allows us to build efficient address generation circuits for BlockRAMs or off-chip memory which streams data quickly into our data-paths.

The synthesized data-paths have an architecture and resource utilization which fairly directly maps to the operators used in the source program. We add pipelining registers to balance pipeline delays. A useful extension of our system would be the addition of programs to specify resource duplication or resource sharing.

RELATED WORK

There are several examples of embedded domain specific languages aimed at circuit synthesis or elaboration for implementation on FPGAs. One of the most notable examples is the JHDL system [2] which embeds a parameterized netlist language into Java.

The CUDA system [6] provides an off-line approach for compiling data-parallel descriptions of kernels written in a special language which has to be compiled used a special compiler for NVidia. In contrast, our data-parallel descriptions can be written in any language that has interop with C and compiled with any C or C++ compiler. Furthermore, our model is on-line for the GPGPU and multicore targets whereas CUDA is always off-line.

Another example of an embedded domain specific language for hardware design is Lava [3] which is designed to permit the parameterized structural description of circuits including detailed layout information without the explicit use of Cartesian coordinates.

FUTURE WORK

As future work we hope to add features to allow designers to specify resource usage and in particular to focus on resource sharing and resource duplication. One approach for specifying this information would involve the use of pragmas although this seems rather informal. Another approach would involve devising extra operators to explicitly specify resource usage. Resource usage specific makes to most sense for the FPGA target to guide the synthesis system to instantiate the required number and type of cores. It is less meaningful for the other targets which are more static i.e. they have their core count burned into their architecture.

CONCLUSIONS

For certain kinds of data-parallel descriptions it is possible to devise a logical language of parallel operations and embed this language into a concrete language like C++ and then use a JIT-ing model to dynamically generate code for GPGPU and multicore targets. The same descriptions can also be compiled to FPGA circuits although we have to use an off-line model because vendor place and route tools are too slow.

It is possible to use languages other then C++ to host the logical Accelerator embedded domain specific language. We have successfully used C# and F# [4] and support for other languages is in progress.

The ability to express a data-parallel computation once and then have it automatically compiled to three different targets is a very useful capability for the exploitation of manycore heterogeneous systems.

REFERENCES

[1] R. F. Barret, P. C. Roth, and S. W. Poole, "Finite difference stencils implemented using Chapel." Technical Report TM-2007/119, Caty Inc., 2007.

[2] P. Bellows, B. Hutchings, "JHDL- an HDL for reconfigurable systems." IEEE Symposium on FPGAs for Custom Computing Machines (FCCM). April 1998.

[3] P. Bjesse, K. Claessen, M. Sheeran and S. Singh, "Lava – Hardware Design in Haskell". International Conference on Functional Programming. ACM SIGPLAN, September 1998.

[4] The F# Programming Language Developer Center. http://msdn.microsoft.com/en-gb/fsharp/default.aspx 2009.

[5] M. Lesniak, "PASTHA – Parallelizing Stencil Calculations in Haskell", Declarative Aspects of Muilticore Programming, Madrid, January 2010.

[6] NVIDIA Corporation, "NVIDIA CUDA compute unified device architecture programming guide," http://developer.nvidia.com/cuda, Jan. 2007.

[7] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," Computer Graphics Forum, 26(1):80–113, 2007.

[8] D. Tarditi, S. Puri, J. Oglesby, "Accelerator: using data-parallelsim to program GPUs for genral purpose uses," ASPLOS 2006.