The Waverun Programming Model for FPGAs

Abhi D.R. and Ron Sass
Reconfigurable Computing Systems Laboratory
University of North Carolina at Charlotte
{adevalap,rsass}@uncc.edu

1. INTRODUCTION

High-Performance, Heterogeneous platforms, such as those based on FPGA and newer devices (Intel Phi), use their specialized cores to improve the speed, power, and energy efficiency of applications. However, this comes with a significant expense to programmability and efficient utilization of off-chip memory bandwidth. This paper describes a programming model that allows a programmer to write a single program that seamlessly executes across a diverse collection of cores with very different execution models and performance metrics. The runtime system is responsible for scheduling tasks and assigning them to the variety of cores. A memory controller is implemented in the FPGA logic. It uses named memory segments for communication and is designed to hide memory latency and reduce unnecessary data movement. (The runtime and memory subsystems are critical to the proposed programming model but are described elsewhere [1, 4].)

Specifically, the Waverun programming model is designed for a popular parallel programming design pattern called the "wavefront" pattern [2, 3, 5]. This pattern arises in many dense matrix and dynamic programming algorithms.[1] The organizing principle in the Wavefront pattern is to describe the computation in a k-ary, n-dimensional iteration space of computations where each iteration operates on a slice or block of the matrices.

2. HARDWARE PLATFORM

To understand the programming model, a high-level summary of the target system is needed. The computational cores consist of, at minimum, two types of processor cores. The "green" cores are small, energy-efficient, and possibly highly specialized. They operate exclusively out of banked/-multiplexed scratchpad memories and on "bare metal" (no OS). The "white" processor cores are large, fast, and have a traditional cache hierarchy with virtual memory. These cores are optimized for speed. An on-chip network allows many simultaneous core-to-core and core-to-memory transfers. The memory controller "owns" the off-chip memory channels and decides when and where to transfer memory segments based on their names. Tasks do not begin until all of the requested memory is available in local memory and run to completion before another task is scheduled.

[1]Other design patterns such as divide-and-conquer were explored by [1].

H2RC'16, November 14, 2016, Salt Lake City, UT.

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

3. PROGRAMMING MODEL

In the Wavefront design pattern, the programmer creates tasks and a top-level series of **if-then-else** statements for each task that determines where the task is located in the iteration space. For some algorithms, the first row of tasks are different from the first column which is different an interior point. In the proposed model, this top-level structure is replaced with a list of functions (customized for the algorithm and based on its location) where each function becomes a specialized task.

When using MPI, for example, the programmer is also responsible for identifying the task's neighbors based on a rank and communicator so that the tasks can exchange data. In the proposed model, the uniform naming system for memory segments allows the memory controller to match producers and consumers. Hence, the programmer writes every task as a set of input-compute-output functions and then relies on the runtime to schedule and manage the parallelism while the memory subsystem handles communication and consistency across the various heterogeneous cores.

The API consists of six calls to the runtime. The programmer provides a `getFunc` which returns a function list (and, of course, the functions in the list). Inside of each task `match` and `post` request input and produce output memory segments. Finally, a main function calls `prime` to inject the initial memory segments (inputs to the application) into the memory subsystem, `schedule` to start the runtime system, and `gather` to pull the resulting memory segments together (to produce application's output).

4. RESULTS

The proposed system has been implemented in software and hardware. The purely software system was used to explore the feasibility of a range of algorithms including (i) Matrix Multiplication (ii) LU Decomposition (iii) QR Decomposition (iv) Cholesky Decomposition. The hardware system is a Xilinx Zync XC706 with two "white cores" (the ARM processors) and twenty "green" cores. The multiplexed scratchpads, on-chip network, runtime, and memory subsystems have been verified independently but the full end-to-end system has yet to be integrated.

5. REFERENCES

[1] B. Huang, R. Sass, N. DeBardeleben, and S. Blanchard. Harnessing unreliable cores in heterogeneous architecture: The PyDac programming model and runtime. In *2014 Dependable Systems and Networks (DSN)*, pages 744–749, June 2014.

[2] Intel Inc. Design patterns. https://www.threadingbuildingblocks.org/docs/help/hh_goto.htm?index.htm#tbb_userguide/Design_Patterns/Design_Patterns.html Accessed September 23, 2016.

[3] E.-G. Kim and M. Snir. Wavefront pattern. http://snir.cs.illinois.edu/patterns/wavefront.pdf. Accessed September 23, 2016.

[4] Y. Rajasekhar. *Revisiting the memory hierarchy in the many-core era: Computation is cheap, bandwidth is everything*. PhD thesis, University of North Carolina at Charlotte, 2014.

[5] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

APPENDIX

Figure 1 shows the obligatory matrix-matrix multiply kernel using the proposed programming model.

```python
class dgemm(waverun):#Class inherits runtime
  def __init__(self,A,B):
    self.A=A #Input matrix
    self.B=B #Input Matrix
    self.Func={} #List of functions for scheduler
  def Mul(self,data1,data2,citer): #Computation
    immd=0
    if(citer!=0):
       immd= self.match('immd',citer-1)
    result=immd+data1*data2 #Computation
    #Post all the results
    self.post(data1,'vert',citer)
    self.post(data2,'horz',citer)
    self.post(result,'immd',citer)
    self.post(self.match('immd',citer),'M')
    return
  def Corner(self,citer): #(0,0) iterspace
    #Gather all the necessary data
    data1=self.match('data1',citer)
    data2=self.match('data2',citer)
    self.Mul(data1,data2,citer) #Call computation
    return
  def Xedge(self,citer): #(*,0) itersapce
    data1=self.match('data1',citer)
    data2=self.match('horz',citer,pos[0]-1,pos[1])
    self.Mul(data1,data2,citer)
    return
  def Yedge(self,citer): #(0,*) iterspace
    data1= self.match('vert',citer,pos[0],pos[1]-1)
    data2= self.match('data2',citer)
    self.Mul(data1,data2,citer)
    return
  def Interior(self,citer): (*,*) iterspace
    data1=self.match('vert',citer,pos[0],pos[1]-1)
    data2=self.match('horz',citer,pos[0]-1,pos[1])
    self.Mul(data1,data2,citer)
    return
  def getFunc(self): #Method listing all functions
    Func={'00':self.Corner,'$0':self.Xedge,'0$':
    self.Yedge,'$$':self.Interior,'*0':self.Xedge,
    '0*':self.Yedge,'$*':self.Interior,
    '*$':self.Interior,'**':self.Interior}
    return Func
  #Function input to application
  def prime(self):
    l=-1
    for i in range (1,self.A.shape[0]+1):
      k=0
      l+=1
      for j in range (1,self.B.shape[0]+1):
        self.post(self.A[i-1][j-1],'data2',k,1,i)
        self.post(self.B[i-1][j-1],'data1',l,j,1)
        self.post(self.A.shape[0],'maxi','',i,j)
        k+=1
if __name__=='__main__':
   myapp=dgemm(Amat,Bmat) #Create an app object
   wave.schedule(myapp,(row,column),row) #Call runtime
   wave.gather('M') #Gather results
```

Figure 1: Matrix Multiplication