ePYTHON

An Implementation of Python Enabling Accessible Programming of Micro-Core Architectures

Nick Brown, EPCC
n.brown@epcc.ed.ac.uk
Micro-core architectures

- Combine many, low power, low cost, low memory cores onto a single chip
  - Challenging to program (immaturity of tools and lack of standards.)
  - Interesting as it has the potential to combine the embedded world with that of HPC and address some of the challenges of exascale
Epiphany

- Announced by Adapteva in 2012, released in 2014

- The Epiphany is a micro-core co-processor
  - Most common version is the Epiphany III with 16 RISC cores, 32KB SRAM per core and eMesh interconnect
    - A 1024 version has been taped out
  - Has achieved 32 GFLOP/s in tests and can achieve 16 GFLOPS/Watt
  - The cores are designed to be very simple and omit common functionality such as support for hardware caching
Parallella

- A Single Board Computer (SBC) built by Adapteva to allow people to experiment with the Epiphany

- Combines the Epiphany with a dual core ARM A9, 1GB main board RAM and runs Linux
  - The ARM CPU & 1GB RAM is the “host” and the Epiphany is the “device”
  - 32MB of the host RAM is shared between the CPU and Epiphany (this is very slow to access from the Epiphany)

- The base model is sold for less than $100
Programmability

- Programming these architectures is difficult and time consuming, especially for novices
  - Has to be done in C and two executables are built by GCC, one for the host and one for the device
  - Data consistency issues when remotely writing to other core’s memory
  - There is no IO from the Epiphany (makes debugging difficult)
  - When dereferencing pointers that pointer must be aligned to the size of the data (i.e. with ints must be aligned to a 4byte word boundary.)
    - If not the Epiphany core simply locks up until it is reset

- No hardware cache management and 32KB of on core memory is really limiting
  - You could put the executable/libraries in shared memory, but this is very slow and has significant performance impact
  - Means we can’t use libc!
Can Python help here?

- **Yes!** Let the programmer concentrate on their problem and parallelism rather than the low level, tricky and uninteresting details (for them) of the architecture.
  - “Go from zero to hero in one minute”
  - Developing parallel Python codes on the Epiphany for both fast prototyping and educational purposes.

- **What about existing interpreters?**
  - **Memory issue** – CPython is many MBs, Numba is MBs and even MicroPython is hundreds of KBs
  - Don’t address the direct lack of IO etc on the Epiphany
  - Don’t necessarily support the parallelism we want to enable
ePython

- Python implementation designed for low memory micro-core processors
  - The resident in core memory ePython interpreter & runtime is limited to 24KB (in reality means about 20KB for code.)
  - Implements the imperative aspects of Python (the non OO stuff) with full memory management and garbage collection
  - Supports parallelism via special Python modules
  - The interpreter itself is written in C with Python modules to be executed by this interpreter

- Provides aspects such as IO, which the Epiphany itself can not support and handling of this is transparent to the user
- Comes pre-installed on every Parallella
import parallel

print "Hello world from core id "+str(coreid())+" of "+str(numcores())

parallella@parallella:~& epython helloworld.py

[device 0] Hello world from core id 0 of 16
[device 1] Hello world from core id 1 of 16
[device 2] Hello world from core id 2 of 16
[device 3] Hello world from core id 3 of 16
[device 4] Hello world from core id 4 of 16
[device 5] Hello world from core id 5 of 16
[device 6] Hello world from core id 6 of 16
[device 7] Hello world from core id 7 of 16
[device 8] Hello world from core id 8 of 16
[device 9] Hello world from core id 9 of 16
[device 10] Hello world from core id 10 of 16
[device 11] Hello world from core id 11 of 16
[device 12] Hello world from core id 12 of 16
[device 13] Hello world from core id 13 of 16
[device 14] Hello world from core id 14 of 16
[device 15] Hello world from core id 15 of 16
Message passing between cores

```python
import parallel

if coreid()==0:
    send(20, 1)
elif coreid()==1:
    print "Got value " + recv(0) + " from core 0"

from parallel import *

a=bcast(numcores(), 0)
print "The number from core 0 is " + str(a)

from parallel import reduce
from random import randint

a=reduce(randint(0, 100), "max")
print "The highest random number is " + str(a)
```
ePython architecture

- Do as much (preparation) as possible on the host
  - Byte code is designed to be as small as possible
- The host runs a monitor inside a thread which waits for commands & data from Epiphany cores
  - This is how we do IO
- The approach is designed to be as portable as possible, to go from one architecture to another all you need to change is the runtime
Epiphany core view

- Byte code, the stack and heap can transparently overflow into shared memory
  - But this can have a significant performance implication

- The communications area is used for inter-core messaging
  - Works in a post box style, where one core will “post” a message to another core
  - Issues around data consistency here so need to use numeric status bytes to keep track of message versioning to ensure when a message has been sent or a new one received.
Other forms of parallelism

• Direct sharing of memory
  • Form of RMA, where (global) memory address is message passed
  • Hidden in the concurrency Python module supplied

```python
from parallel import coreid, send, recv

if (coreid() == 0):
    send(exampleFn, 1)
    send(22, 1)
elif (coreid() == 1):
    fn=recv(0)
    fn(recv(0))

def exampleFn(a):
    print "Hello from "+str(a)
```

• Passing functions between cores
  • Functions are first class values, so can be message passed
  • Task farm module built to abstract this and enable task based parallelism

• All builds on message passing abstraction
Epiphany as an accelerator

- ePython as an execution engine

Full Python

Kernels and input data

Host CPU

Results

Micro-cores

- ePython
- ePython
- ePython
- ePython
- ePython
- ePython
- ePython
- ePython

EPIPHANY

epcc
Higher level offload approach

- Follows a decorator approach, where functions in Python code running on the host are decorated which seamlessly offloads them to the co-processor
  - Seamless in terms of the low level execution and data movement, programmers will probably still need to optimise these kernels explicitly for the architecture

```python
from epython import *

@offload
def my_kernel(a):
    return a+32

print my_kernel(12)
```

*Imports the epython module in CPython or any other interpreter*

*Denotes that this function will run on the Epiphany*

*From Python running on the host will execute my_kernel on the Epiphany, passing any arguments and returning results*

The critical thing is implementing this without any modifications needed to ePython itself
Kernel launch options

- The default is run the kernel on every core and blocking for completion
- Can provide arguments to the decorator or function call to override this

```python
from epython import *

@offload(async=True)
def my_kernel(a):
    return a+32

df = my_kernel(12)
print df.wait()
print my_kernel(19, async=False, target=1)
```

- A nice thing about Python decorators is that we can create aliases to other decorators with specific arguments (short-cuts)

```python
from epython import *

@offload_multiple
def my_kernel(a):
    return a+32

@offload_single
def my_kernel2(a):
    return a-32
```
Managing device data

- As well as copying data to and from the Epiphany by function arguments and return values, can define global device data

```python
from epython import *

a=0
define_on_device(a)

@offload
def modifyA():
    global a
    a=99

copy_to_device("a", 15)
modifyA()
print copy_from_device("a")
```

- These three functions are provided as pre-defined functions in the coprocessor module (imported into the generated Epiphany code.)

- Therefore no interpreter/runtime changes needed, as device data management is implemented using the same mechanism as our user defined offloaded functions
Detecting lung cancer in 3D CT scans

- 2017 data science bowl working with the Cancer moonshot to make 10 years of cancer progress in five years
- Image assessments currently in use today result in a large number of false positives
- NCI made over 1500 3D CT scans freely available with labelled data indicating the presence of cancer or not
- Can micro-core architectures (and our approach) be used here at all?
Machine learning approach

```python
@offload
def feed_forward(image_chunk):
    h_data = dotProduct(model_W1, image_chunk)
    return dotProductVector(model_W2, h_data)
```

```python
for batch_ids, batch_cancer in zip(trg_imgs, trg_cancer):
    for batch_id, cancer in zip(batch_ids, batch_cancer):
        image = np.load(batch_id + "_.npy").ravel()
        handlers = KernelExecutionHandler()
        for i in range(0, 16):
            handlers.append(feed_forward(image[i*D/16:(i+1)*D/16],
                                         async=True, target=i))
        local_logp = handlers.wait()
        prob = sigmoid(sum(logp))
        action = 1 if np.random.uniform() < prob else 0
        reward = 1 if action == cancer else 0

        combine gradient
        update the model
```

- Matrix representing interactions between input & hidden layer is 2.8MB. Decompose across the Epiphany cores so 180KB/core
- Vector representing interaction between hidden layer and output neuron is 800 bytes
Performance comparison

The byte code size running on the Epiphany is 2.6Kb, with 1.5Kb required for the symbol table (99 entries)
Moving forwards

• Improvements to ePython
  • JIT compilation of code
  • Interfacing with native libraries
  • Port to other micro-core technologies

• General programming challenge
  • Memory hierarchies
  • Low level portability layer
On going FPGA work

- Much of the existing architecture can be applied to FPGAs
- Our byte code is pretty high level, so the theory is we can do lots of interpretation in a single clock cycle

- Implementing the virtual machine in VHDL
  - Existing implementation is a “reference”
Conclusions

• Writing parallel Python codes for these micro core architectures is useful

  • For prototyping, education and moving forwards production codes
  • ePython supports this and can be adapted to other many core architectures

• Our offload directive approach uses ePython as an engine, this is applicable to other micro-cores

github.com/mesham/epython