Offloading Python kernels to micro-core architectures

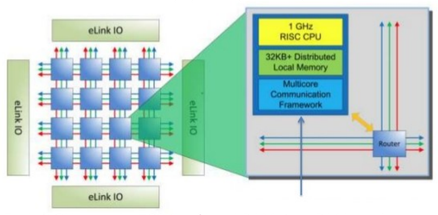
Nick Brown (EPCC at the University of Edinburgh) n.brown@epcc.ed.ac.uk

Micro-core architectures

Combine many *simple*, *low power*, processor cores onto a single package. These are low cost, very energy efficient and leverage high degrees of parallelism. Current manufacturers include:

- Adapteva (16 core Epiphany III and 1024 core Epiphany V)
- Kalray (256 core MPPA)
- Knupath (256 core Hermosa)
- XMOS (32 core xCORE)

The 16 core Epiphany III co-processor



Each 1 GHz core has 32Kb of local RAM, two DMA engines and a fast mesh network for inter-core communications.

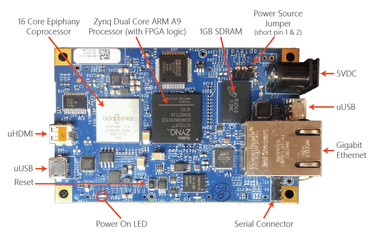
Graphic courtesy of Adapteva.

In many ways we are seeing the embedded and HPC worlds converging, the embedded world now interested in parallelism due to performance, and the HPC community having to consider energy efficiency for exa-scale machines.

Micro-core architectures, providing significant performance for low power are therefore of great interest to both communities.

Parallella: An SBC to experiment with the Epiphany

This is a single board computer, built by Adapteva as a technology demonstrator around the 16 core Epiphany III co-processor. Marketed as *supercomputing for* everyone, with an ARM CPU and 1 GB main memory, running Linux the base version sells for $99. Comes with an Epiphany III co-processor (runs bare metal) that shares 32MB of the main memory.



Programming the Epiphany is technically challenging and time consuming:

- Lack of hardware caching. Whilst 32MB of main memory is addressable by the Epiphany it is very slow to access. Hence either the programmer develops their own software cache approach or limits themselves to 32Kb on-core memory (libc is much bigger than this!)
- Lack of direct IO making the debugging of codes difficult
- Memory must be aligned to specific boundaries or else the core locks up on pointer de-reference

Our challenge: How can we make programming these trivial? It should be possible to go from "zero to hero", writing a simple parallel code, in less than a minute. Experienced programmers should be able to easily experiment with the Epiphany and these architectures for their codes.

Programmability challenge: Python to the rescue!

The technical challenge of writing codes for these architectures has limited their uptake. Python can help here by letting the programmer concentrate on their problem and parallelism rather than the low level, tricky and uninteresting details (for them) of the architecture. The focus of this work is around **education** and **fast prototyping/experimenting** with micro-cores.

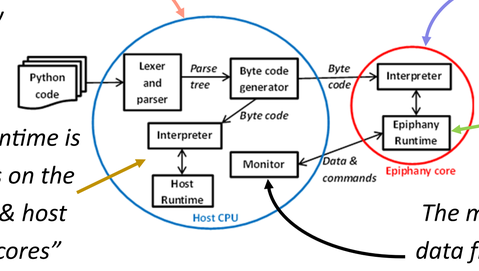
But how can we physically support parallel Python codes running in the very limited resources that these chips provide (i.e. a maximum 32Kb of RAM per core on the Epiphany?)

ePython: Our 24Kb implementation of Python

ePython implements the imperative aspects of Python with full memory management, garbage collection, message passing parallelism, shared memory and task based parallelism.

Do preparation on the host, byte code is generated and copied to the Epiphany

Only a very tight interpreter and runtime (Virtual Machine) runs on the micro-cores. 24Kb in total



Handles aspects such as memory management, parallelism and garbage collection

The interpreter is portable, runtime is architecture specific. Threads on the host can run the interpreter & host runtime to provide "virtual cores"

The monitor listens for commands & data from Epiphany cores, this is how we implement aspects such as I/O

```
from parallel import *
from random import randint
a=reduce(randint(0,100), "max")
print "The highest random number is "+str(a)
```

Run from the Parallella command line and executed on each Epiphany core. This code generates a random number on each core and performs a reduction to determine the global maximum value which is displayed

But with a 24Kb interpreter don't you just have 8Kb left for all the byte code and data?

No—these aspects transparently flow over to the 32MB shared memory (with a runtime penalty)

ePython as an execution engine: Decorators to offload kernels in existing Python codes

We initially focussed on running the entire Python code directly on the micro-cores. But in fact this technology can be more suitably viewed as an accelerator, only offloading specific kernels from the host, providing benefits:

- ✓ Offloading kernels most effectively optimises the use of limited on-core memory.
- ✓ Is an accessible way of encouraging novices to experiment with parallelism and accelerators for education
- ✓ ePython is a subset of the language, offloading kernels would more effectively support large scale existing Python codes. Non-computational aspects on the host could potentially run concurrently with kernels.

What we want to do: Be able to decorate kernels in an existing Python code, run this in any interpreter on the host and during execution for these to be seamlessly offloaded to the micro-cores under ePython.

Offload this function, pass data to the cores as arguments and send back return values to the host

```
def my_kernel(a):
    return a+32

print my_kernel(12)
```

Existing Python code running under any interpreter (such as CPython) on the host

Programmer modifies their code using our simple abstractions

```
from epython import *

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

print my_kernel(12)
```

By adding the offload decorator this seamlessly executes functions on the Epiphany cores

Behind the scenes, functions are launched via message passing from the host to micro-cores. The *taskfarm* ePython module listens for messages.

When code starts up kernels are extracted and run on the Epiphany via ePython.

```
import coprocessor
from taskfarm import *

worker()

def my_kernel(a):
    return a+32
```

Hidden from the programmer, this is automatically generated & runs on the Epiphany cores via ePython

Without additional arguments, by default launches the kernel on all cores and blocks host code for completion. Results are returned as a list, each element per core

```
from epython import * ①

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

print my_kernel(12)
```

```
from epython import * ②

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

handlers=my_kernel(12)
print handlers.wait()
```

Kernel is launched asynchronously (non-blocking)

Handlers are returned which can be tested on and waited for. A scheduler supports queues of kernel launches

The kernel is defined to explicitly run on cores 7, 8 and 9 only but for specific function calls these can be overridden

```
from epython import *

@offload(target=[7,8,9])
def my_kernel(a):
    return a+32

print my_kernel(12, target=1)
```

③ *Run on core 1 only*

```
from epython import *
a=0
define_on_device(a)
copy_to_device("a", 15)

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

modifyA()
print copy_from_device("a")
```

④

Micro-core resident data

Define variable "a" to exist on each Epiphany core and copy an initial value in from the host

The kernel on each Epiphany core accesses this global "a", and modifies it.

It is this modified value that is copied back to the host and displayed

ePython Benchmark: Machine learning for detecting lung cancer in 3D CT scan images

The US cancer moon shot program is looking to make 10 years of progress fighting cancer in just 5 years. The 2017 data science Kaggle competition has teamed up with the National Cancer Institute (NCI) who have made available 1500 labelled 3D lung CT scans.

Current lung cancer detection methods from CT scans are notoriously error prone and result in many false positives. The competition has required, through machine learning, participants to improve lung cancer detection from scans.

- ***Can micro-cores, ePython and our decorators be used to code up a simple neural network suitable for addressing this problem?***
  - ***Can we offload the NN computational kernels to micro-cores with ePython as the execution engine? Is this accessible for education?***

This kernel is offloaded to the micro-cores and performs a forward feed pass through the neural network. It works on its own local neurons and the hidden layer

Individually launch each kernel on a specific Epiphany core with it's own sub-domain of the image. These are all launched in an asynchronous non-blocking manner

Calculate if the neural network determines whether this is cancerous and deduce the reward.

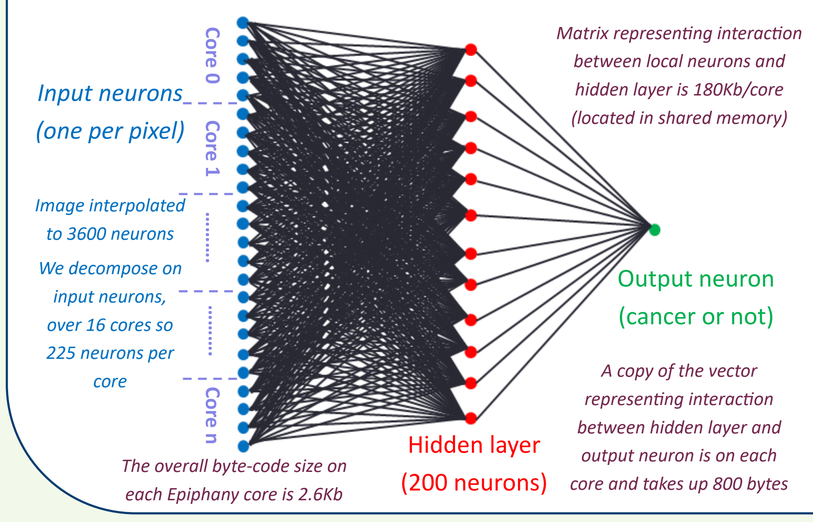
```
@offload
def feed_forward(image_chunk):
    h_data=dotProduct(model_W1, image_chunk)
    return dotProductVector(model_W2, h_data)

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
```

For each training batch, iterate through each image and load this in

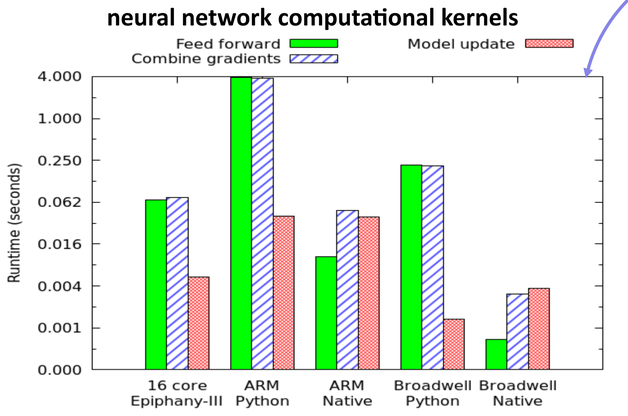
Wait for all kernels to complete executing, sum up all the local contributions towards "logp" and then pass this to the sigmoid function

Combine the gradients and for each training run update the model weights (these form the back propagation and are omitted for brevity.)

In many approaches the input is stored for each image in the batch and back-prop done in one step at the end. We don't have the memory for that hence doing a partial back-prop at each step and updating the neuron weights after the end of the current training batch of images.



Average runtime per execution of the three main neural network computational kernels



ePython is competitive with Python only approaches

- *This is due to the inherent micro-core parallelism that we take advantage of and the lightweight nature of ePython*
- *Using our approach for education, it is fairly trivial to modify Python codes and experiment with parallelism and running kernels on accelerators*
- *But a native approach (BLAS) significantly outperforms ePython*

Conclusions and future work

- *Micro-cores show potential, our approach means people can quickly experiment with their codes on these architectures.*
- *Performance improvements (such as JIT) would be useful*
- *Porting ePython to other micro-core architectures is a next step*

Scan me for the ePython repository, tutorials and more information

Or visit the repository at github.com/mesham/epython