Having your cake and eating it:

Exploiting Python for programmer productivity and performance on micro-core architectures using ePython

Maurice Jamieson
Nick Brown
Sihang Liu
EPCC at the University of Edinburgh

What are micro-cores?

- Micro-cores have small instructions sets and tiny amounts (c.32KB) of on-chip memory, resulting in very low power consumption
- The low power consumption and simplicity of core mean that a large number can be placed on a chip
- The HPC community is more and more interested in micro-core architectures as they provide massive parallelism on -chip. For example the RISC-V based European Processor Initiative (EPI), and a combination of micro-cores with normal technology for "posits"
- There are lots of hard-processor examples including the Adapteva Epiphany and the PEZY-SC2 (Shoubu system B)
- There is also an increasing number of soft-core examples, such as the GRVI Phalanx

Micro-core memory hierarchies

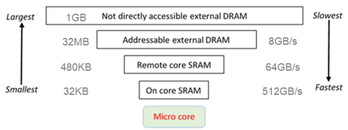
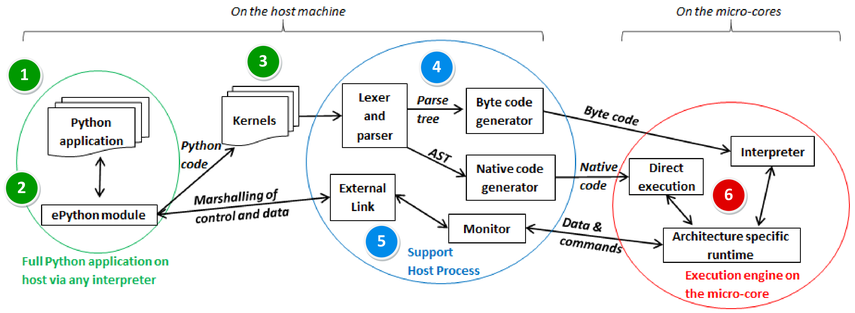


Illustration of memory hierarchy for the Epiphany

- To overcome the extremely limited on-core memory of micro-cores, many architectures support memory hierarchies. As shown for the Epiphany above, the bandwidth increasingly drops for the increasing size of memory away from the core - external DRAM 8GB/s vs on-core SRAM 512GB/s
- Furthermore, on the Epiphany the addressable external DRAM is only 32MB in size. For many data sets, this is insufficient and ePython provides micro-core access to the 1GB of host DRAM as outlined in iii) on the right

ePython overview

1. Standalone ePython scripts, as outlined in i) below, are downloaded to the micro-cores and/or run as threads on the host
2. A module is provided for import into Python code on the host, which provides abstractions and underlying support for handling the offloading of code fragments (*kernels*) to the micro-cores, discussed in ii) below right
3. ePython compiles the code on the host, generating bytecode or native code
4. An ePython support process downloads the code and performs marshalling / control of the micro-cores
5. An execution engine on each of the micro-cores contains an architecture specific runtime, paired with either the ePython interpreter or the execution of native code generated from the programmer's offloaded Python kernels 3



ePython architecture

i) Example parallel ePython code

```
1  from parallel import reduce
2  from random import randint
3
4  def randomise():
5   x = randint(0,100)
6   print "Random number is " + str(x)      A
7   return x
8
9  a = reduce(randomise(), "max")           B
10
11 if coreid()==0:                          C
12   print "The highest random number is " + str(a)
```

```
maurice@parallella: epython example.py
[device 0] Random number is 96
[device 1] Random number is 17
[device 2] Random number is 53
[device 3] Random number is 14
[device 4] Random number is 30
[device 5] Random number is 6
[device 6] Random number is 90
[device 7] Random number is 98
[device 8] Random number is 20
[device 9] Random number is 37
[device 10] Random number is 22
[device 11] Random number is 98
[device 12] Random number is 64
[device 13] Random number is 69
[device 14] Random number is 37
[device 15] Random number is 65
[device 0] The highest random number is 98
```

A Strings from each of the cores are sent back to the host for output

B Randomise function executed on each core and *max* reduction applied to all results to determine maximum - other reduction operators include *min*, *sum* and *prod*

C By checking the core ID we can output the maximum value once

ePython provides inter-core and host communications / control functions e.g. send, receive, broadcast and sync

ii) Offloading kernels

```
1 from epython import offload
2
3 @offload
4 def mykernel(a):
5  print "Hello with " + str(a)
6  return 10
7
8 print mykernel(22)
```

Like the approach taken by Numba, kernel functions to be offloaded are annotated with a decorator, *@offload*. By default, this will be executed on all micro-cores and the host will block until execution has completed. These defaults can be overridden to control the kernel launch and execution by providing explicit arguments to the decorator such as *@offload(async=True)* or by providing options as a named argument to the function call e.g. *mykernel(22, async=True)*

Offload decorator arguments

Async	*Async=True* will execute the kernel in a non-blocking manner and will return a handler of type *KernelExecutionHandler* immediately
Auto	*Auto=n*, where *n* is the number of cores to execute the kernel over and signifies no preference of cores used, but instead to run the kernel on *n* free micro-cores whenever these are available
All	*All=True* will collectively execute the kernel on all available micro-cores
Target	*Target=n*, where *n* is either an integer core id or list of core ids, will guarantee to execute the kernel on those specific cores only, allowing some distinct state or data held by core(s) to be utilised by a kernel
Device	*Device=d*, where *d* is a single type or list of types of micro-core architecture on which to execute the kernel - for programming heterogeneous micro-core systems with a number of micro-cores CPUs of different types, with device types defined for each available micro-core

iii) Memory model

```
1 from epython import offload, memkind
2 import random
3
4 nums1 = memkind.Host(types.int, 1000)
5 nums2 = memkind.Host(types.int, 1000)      D
6
7 ....
8
9 @offload
10 def mykernel(a, b):
11 ....
12
13 print mykernel(nums1, nums2)
```

- In combination with pass by reference and possible prefetching, it is also desirable for the programmer to be able to direct where in the memory hierarchy their data resides. This is supported via memory kinds.
- Micro-core access to a scalar variable or the index of an array, held elsewhere in the memory hierarchy, results in preference being given to any local copy (cached), otherwise a data transfer will be performed from where the data is physically located, copying it to the micro-core and then caching it. The caching policy is *write-through*, where the locally held copy is used for all the reads, and writes are performed on both the local copy of data and also written back to the variable's location elsewhere in the hierarchy.

D In this example, the ePython *memkind* class allocates data on the host but can be anywhere in the memory hierarchy the data belongs. The data type and amount of memory to allocate can be selected. Variables can still be defined using standard Python syntax and the variable will be declared in the level of memory hierarchy that is currently in scope.

Having our cake and eating it

The performance limitations of the ePython interpreter are apparent when we compare an ePython and C version of a benchmark and running on Parallella's host ARM Cortex-A9; the C version executes 80 times faster than the ePython version running on eight cores of the Epiphany!

- The nascent ePython native code generation dramatically closes the C performance gap to between 10% and 40% for a Jacobi benchmark:

CPU	Language	Runtime (s)
Epiphany	C	0.029
Epiphany	ePython (native)	0.031
MicroBlaze	C	0.990
MicroBlaze	ePython (native)	1.377
AMD64	C	0.016
AMD64	ePython (native)	0.019

- This is all underpinned by a dynamic code loading approach, that fetches individual functions and data onto the micro-cores on demand. As such, the minimum executable size is 1.5KB (dynamic bootloader size) with the micro-core memory then caching these code snippets where possible

Now we can fit ePython code into the on-chip micro-core SRAM *and* have the performance we desire

Currently supported Micro-cores

- Adapteva Epiphany III
- Xilinx MicroBlaze (soft-core)
- RISC-V PicoRV32 (soft-core)
- Threads running on the host CPU

Further work

- Upgrade from Python 2.7 to Python 3
- Mature native code generation:
  - Reduce the runtime library footprint from 15KB to 1.5KB
- Extend dynamic loading to third-party libraries

Available on GitHub:
https://github.com/mesham/epython

Contact: maurice.jamieson@ed.ac.uk