ePYTHON

An implementation of Python for the many-core Epiphany coprocessor

Nick Brown, EPCC
nick.brown@ed.ac.uk
Epiphany

- Announced by Adapteva in 2012, released in 2014

- The Epiphany is a many core co-processor
  - Most common version is the Epiphany III with 16 RISC cores, 32KB SRAM per core and eMesh interconnect
  - 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
  - Interesting as it has the potential to combine the embedded world with that of HPC and address some of the challenges of exascale
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 the Epiphany 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 many 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 (i.e. the non OO stuff) with full memory management and garbage collection
  • Supports parallelism via the parallel Python module
  • 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
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

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)
Parallel Gauss Seidel example

- Parallel SOR version of Gauss Seidel to solve Laplace’s equation for diffusion in 1D
  - Uses send and receive for halo swapping between cores
  - Reduce for summing up the norm between iterations

- The Python code (in the paper) is 52 lines in total
  - If you understand the algorithm this is very simple and you can clearly see the higher level ideas behind geometric decomposition
  - Apart from the function calls to the parallel module it runs unmodified in any Python interpreter

- Equivalent C code was 266 lines
  - Lots of lower level concerns such as data consistency
Parallel Gauss Seidel performance

- Global size of 1000 elements, solving to 1e-3 with a relaxation factor of 1.3

<table>
<thead>
<tr>
<th>Runtime (s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.61</td>
<td>ePython on 16 Epiphany cores</td>
</tr>
<tr>
<td>1.01</td>
<td>C on 16 Epiphany cores</td>
</tr>
<tr>
<td>52.04</td>
<td>ePython byte code and data in shared memory</td>
</tr>
<tr>
<td>14.71</td>
<td>CPython on host CPU only</td>
</tr>
<tr>
<td>2.23</td>
<td>C on host CPU only</td>
</tr>
</tbody>
</table>
Parallel Gauss Seidel Strong Scaling

- The same experiment running in ePython, varying the number of Epiphany cores
Host interoperability

- Two ways of Python codes running on the Epiphany to interact with the host ARM CPU
  - Can create “virtual cores” which run and communicate exactly like Epiphany cores but are in fact running on the CPU
    ```
    parallella@parallella:~& epython -h 5 -c 16 helloworld.py
    ```
  - The parallel module provides `isdevice` and `ishost` functions

- Support for running “full fat” Python on the host (in any interpreter such as CPython) and this interacting with ePython running on the Epiphany
  - Import the `epython` module in the host code and use this like a “virtual core”, communicating via message passing
Passing functions between cores

- As functions are first class values these too can be communicated between cores

- A taskfarm module is provided which builds on this to provide non-blocking execution of functions on other cores, testing for completion and awaiting return results
  - Works with different numbers and types of function arguments and both scalar and array return values
  - Used for master-worker style parallelism

```python
import parallel

if (coreid() == 0):
    send(functionToRun, 1)
    print recv(1)
elif (coreid() == 1):
    op = recv(0)
    send(op(), 0)

def functionToRun():
    print "Running on core 1"
    return 10
```
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.
Educational uses

• One of the important aspects of ePython, Python and the Parallella is that of education and teaching people how to write and architect parallel codes
  • Yes you could run very many processes on a multi-core desktop, but a machine like the Parallella captures people’s imagination and this also teaches heterogeneous parallelism.

• Blog tutorials about parallel codes
  • Parallel messaging, geometric decomposition, pipelines and task farms

• Lots of examples in the ePython repository
  • Jacobi, Gauss Seidel, Mandelbrot, number sorting, dartboard method to generate PI, master-worker etc....
Epiphany-V

- Adapteva announced the 1024 core Epiphany-V coprocessor last month
  - This is truly many core and has a theoretical power efficiency of 75 GFLOPS/Watt
  - Each core will have 64KB SRAM, which seems cavernous when compared to the Epiphany III, but still very constrained generally.

- Due to ePython the announcement stated that Python is supported on this new chip
  - Which is true, it will be possible to write parallel Python codes that run on the 1024 cores.
  - The Python examples we have seen here should just run without any modification required.
Conclusions and further work

• Writing parallel Python codes for these many core architectures is useful
  • For prototyping and education
• ePython supports this and can be adapted to other many core architectures

• Direct memory sharing (and safety such as mutexes) between cores
• Optimisation on the Epiphany V (we have double the amount of memory!)
• Further work on offloading from the host

[GitHub Link] github.com/mesham/epython