**Microcore 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 128-core Epiphany IV),
- Kalray (124 core MPDK),
- Requisite (256 core hereon),
- IMONS (32 core eCORE).

In every way we are seeing the embedded and HPC worlds converging, the embedded world now interests in parallel due to performance, and the HPC community having to consider energy efficiency for on-scale machines. Microcore architectures, providing significant performance for low power, are essential for green interest in 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, for 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 stacks. While 32MB of main memory is addressable by the Epiphany it is very slow to access. Hence the programmer develops their own software cache approach or limits themselves to 24MB of core memory (big is much larger than that)
- Lack of direct I/O making the debugging of core problematic.
- Memory must be aligned to specific boundaries or else the core locks up on pointer dereference.

**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 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**

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

*What we want to do:* Be able to decorate kernels in an existing Python code, run this in any interpreter on the micro-cores. 24kB in total.

*Do preparation on the host, byte code is generated and copied to the micro-cores.*

Handles aspects such as memory management, garbage collection and parallelism.

- Only very light interpreter and runtime (Virtual Machine) runs on the micro-cores. 24kB in total.
- 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 on the host.

- 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 focused 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.**

- The technical challenge of writing codes for these architectures has limited their uptake.
- The interpreter is portable, runtime is architecture specific. Threads on the host can run the interpreter & host code for batch_ids, batch_cancer in zip(batch_ids, batch_cancer):

*def my_kernel(a):*

*return a+32*  

*From epython import **

*handlers=my_kernel(12)*  

*print handlers.wait()*  

*from epython import **

*my_kernel(12)*  

*print my_kernel(12)*

**Parallella screenshot**

Graphic courtesy of Adapteva.

- Input neurons
- Hidden layer (200 neurons)
- Output neuron (cancer or not)

*Matrix representing interaction between local neurons and hidden layer is 200x128 (shared in memory)*

*An array of the vector representing interaction between hidden layer and output neurons in one core and takes up 600 bytes*

*The overall byte code size on each Epiphany core is 2-3kB*