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, cores into a single processor package. In many ways we are seeing the embedded and HPC worlds converging, the embedded world now interested in parallelism for performance, and the HPC community having to consider energy efficiency for future exa-scale machines. These micro-core architectures, providing performance through significant parallelism at low power cost are therefore of great interest to both communities. There are numerous micro-core architectures (Adapteva's Epiphany [10], Kalray MPPA [6], Knupath Hermosa [8] and XMOS XCore [13]), at varying levels of availability, maturity and cost. This work focuses on Adapteva's 16 core Epiphany III [1], delivering 32 GFLOPs at 2 Watts power draw. Each RISC core comes with its own local, 32Kb of on chip memory and fast inter-core network. The same company also developed the Parallella [2] single board computer, built as a technology demonstrator around the Epiphany co-processor. With a host dual core ARM, 1 GB of RAM and an Epiphany, the based model sold at $99 and is marketed as supercomputing for everyone due to this low price point. However actually programming the Epiphany is technically challenging and, whilst some approaches over and above the low level library provided by Adapteva, such as OpenCL [11], BSP [5], OpenMP [3] and MPI [12] have been ported, these are at different levels of maturity and still require the programmer to explicitly program the chip using C at a low level. Whilst 32MB of the Parallella's main memory is addressable by the Epiphany, there is a significant performance penalty in using this memory. Coupled with the fact that there is no hardware cache support, programmers have to either keep their programs and data within this 32Kb of local memory or design their codes to pre-fetch for reasonable performance. In short it is difficult and time consuming to program these architectures.

1 EPYTHON

ePython [4] is our implementation of a subset of Python for the Epiphany co-processor that comes pre-installed with all Parallella boards. The purposes of ePython is firstly educational and secondly as a research vehicle for fast prototyping applications on, and experimenting with, micro-cores. ePython allows a novice to go from *zero to hero*, i.e. write a simple parallel hello world example running on the Epiphany, in less than a minute. Due to the memory limitations of the Epiphany, our ePython interpreter and runtime (written in C) fits into 24Kb of memory, with the remaining 8Kb used for user byte code, the stack, heap and communications. It is possible for byte code, the stack and heap to overflow into shared memory (for large codes) but there is a performance impact of doing so. ePython supports a rich set of message passing, shared memory and task based parallel abstractions.

Listing 1 illustrates a simple example where each micro-core will generate a random integer and then performs a collective message passing reduction to determine the maximum random number (due to the *"max"* operator) which is then displayed by each core.

```
1 from parallel import *
2 from random import randint
3
4 a=reduce(randint(0,100), "max")
5 print "The highest random number is "+str(a)
```

Listing 1: ePython collective reduction example

To fit within our 24Kb memory limit we designed ePython to do as much preparation of a user's Python code on the host CPU as possible, with only the minimal amount of functionality running on the Epiphany to physically execute the code. This is illustrated in figure 1 where Python code is first preprocessed and parsed on the host to form a parse tree, which is then walked to generate byte code. For memory reasons this byte code is different to Python's standard byte code. Byte code is then copied from shared memory into the local memory of each Epiphany micro-core, which is then executed by the interpreter on every micro-core in parallel.

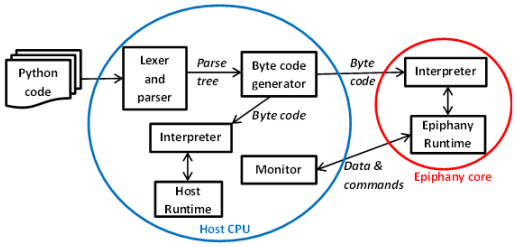

Figure 1: ePython architecture

The interpreter, which is designed to be portable, calls out to the architecture specific runtime for functionality such as allocating memory, communications and IO. A challenge of programming the Epiphany is that it does not support direct IO, which is often useful for debugging codes. To address this (and other limitations) the host will run, in a thread, a monitor which polls a specific area of shared memory for commands and data. For example in order to implement print functionality the Epiphany runtime will write the *print* command, along with a message, into shared memory which is picked up by the monitor and actioned. We adopted the interpreter approach because, whilst the direct compiling of Python code would avoid the need for an interpreter, one would still require the runtime (14Kb of 24Kb) and interpreting makes certain aspects, such as first-class valued functions and overflowing of code, significantly simpler to handle and hence less memory overhead.

2 OFFLOADING FROM FULL PYTHON

Previously the user wrote Python code and directly executed this on the micro-cores. In this poster we detail work done to enable the offloading of kernels to micro-cores in large scale existing Python codes running on the host. Programmers annotate functions to be offloaded with a specific decorator, *@offload*. When these decorated functions are called by the host code then Python will execute the functions using ePython on the micro-cores, passing any input values and sending back return values. This is illustrated in listing 2, which is run by any existing Python interpreter on the host and the *mykernel* function is transferred to, and executed on, every micro-core when called at line 6. The physical offload, along with transfer of arguments to and return values from the micro-cores is entirely abstracted from the programmer. In this example the only modification needed to standard Python code for offloading was importing the *epython* module and decorating the function. We believe this is also useful in education, where novices can quickly offload Python functions to cheap micro-core accelerators and obtain immediate results without worrying about the nitty gritty details initially. Our use of decorators is similar to the approach taken by Numba [9], but their focus is around JIT and GPU acceleration, with the Numba generated code being far too large to fit into the limited memory of the micro-cores. In this work we have developed the appropriate abstractions (decorators and functions) for targetting micro-cores, and then leveraged our existing ePython work as the execution engine.

```
1 from epython import *
2 @offload
3 def mykernel(a):
4   return a+32
5
6 print mykernel(10)
```

Listing 2: Python simple offload example

By default the *@offload* directive will execute the function on every micro-core in a blocking fashion. It is possible to override this, for instance *@offload(async=True, target=[1,3])* would execute the kernel in a non-blocking manner (where control flow continues on the host and a handler is immediately returned which can be tested on or waited for) on micro-cores 1 and 3 only. We also provide the ability (via functions in the *epython* module) to define and manage device resident data which avoids the need to excessively copy common data to and from the micro-cores for every kernel launch. The poster provides labelled code examples of these different facets to our approach.

Executing the kernels, as well as handling the device resident data management functions, is all built upon the fundamental abstraction of message passing functions as first class values to and from the cores. At the ePython interpreter and runtime level no wide spread modifications were required to support this offloading approach, which is desirable due to memory limitations.

3 A MACHINE LEARNING EXAMPLE FOR DETECTING LUNG CANCER

The 2017 Kaggle data science bowl [7] is a challenge around the development of lung cancer detection algorithms. In 2016 the Cancer Moonshot was announced to make a decade's worth of progress in cancer prevention, diagnosis, and treatment in 5 years. The National Cancer Institute (NCI) has made available thousands of high-resolution 3D lung scans and the aim is to develop techniques for determining whether lesions are cancerous or not. This poster is using the NCI's data differently to the competition and asking a separate question. Instead of being concerned with improving the accuracy of these algorithms we are evaluating whether our approach for offload is suitable for this kind of work and more generally whether micro-core architectures and the parallelism that they provide can benefit this area. Accuracy of detection is the primary concern of the competition, but the execution of these algorithms also needs to be realistic. Not only does this involve training the model in a timely fashion, but also employing an architecture which is affordable and utilises a minimal amount of power, which is where micro-core architectures are of interest. ePython aside, the current programmability challenges of micro-cores means that there are no machine learning frameworks or sample neural network code currently developed for the Epiphany or its counterparts. The theory is that our decorator offload approach, and the maturity of ePython, aids programmability and makes micro-cores a viable target for this field.

We have developed a simple neural network where each pixel in the image is an input neuron, a single hidden layer (200 neurons) and an output neuron that determines whether the image was cancerous or not. The input neurons are distributed between the 16 Epiphany cores, with each core working on a separate sub-set of the image. Three kernels are offloaded to the micro-cores; forward-feed for each image, calculating the gradient of the loss function for each image and updating the model with a combined gradient after each training batch has completed. Whilst our code will inevitably be far simpler than the winning algorithms of the data science bowl, the fundamental linear algebra calculations that underlie the forward-feed and back-propagation will be the same and it is these that we are utilising the micro-core accelerator, our offload abstraction and ePython to perform. Educationally the parallelism of this code and our decorators is accessible to, and understandable by, HPC novices.

The poster illustrates and explains our offloaded forward-feed Python kernel code. Performance results against Python and native (BLAS) codes (both CPython 2.7) running on one core of ARM and Intel Broadwell CPUs are also illustrated and it can be seen that using micro-cores for this sort of workload is possible and we are competitive with Python on the CPU only (CPython) but that native code is faster. This not hugely surprising due to the interpreted nature of ePython, and as further work it would be interesting to look at Just In Time (JIT) compilation of ePython code as well as porting ePython to other micro-core architectures.

REFERENCES

[1] Adapteva. E16g301 epiphany 16-core microprocessor. http://www.adapteva.com/docs/e16g301_datasheet.pdf, 2013. [Online; accessed 3-April-2017].

[2] Adapteva. Parallella-1.x reference manual. http://www.parallella.org/docs/parallella_manual.pdf, 2013. [Online; accessed 3-April-2017].

[3] Spiros N. Agathos, Alexandros Papadogiannakis, and Vassilios V. Dimakopoulos. *Targeting the Parallella*, pages 662–674. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[4] Nick Brown. epython: An implementation of python for the many-core epiphany coprocessor. In *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '16, pages 59–66, Piscataway, NJ, USA, 2016. IEEE Press.

[5] Jan-Willem Buurlage, Tom Bannink, and Abe Wits. Bulk-synchronous pseudo-streaming algorithms for many-core accelerators. *CoRR*, abs/1608.07200, 2016.

[6] Benoit Dupont de Dinechin. Kalray mppa®: Massively parallel processor array: Revisiting dsp acceleration with the kalray mppa manycore processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–27. IEEE, 2015.

[7] Kaggle. Data science bowl 2017. https://www.kaggle.com/c/data-science-bowl-2017, 2017. [Online; accessed 3-April-2017].

[8] KNUPATH. Knupath hermosa processors, 2016.

[9] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[10] A. Olofsson. A manycore coprocessor architecture for heterogeneous computing. In *Los Alamos Computer Science Symposium (LACSS) 2009*, 2009.

[11] D Richie. Coprthr api reference. 2013. www.browndeertechnology.com/docs/coprthr_api_ref.pdf.

[12] James A. Ross, David A. Richie, Song Jun Park, and Dale R. Shires. Parallel programming model for the epiphany many-core coprocessor using threaded MPI. *CoRR*, abs/1506.05442, 2015.

[13] XMOS. xcore: Architecture overview. https://www.xmos.com/download/private/xCORE-Architecture%281.2%29.pdf, 2013. [Online; accessed 3-April-2017].