11th International Workshop on Cellular Neural Networks and their Applications
Santiago de Compostela, Spain, 14-16 July 2008

A Cellular Processor Array Simulation and Hardware Prototyping Tool

David R. W. Barr and Piotr Dudek

School of Electrical & Electronic Engineering
The University of Manchester, PO Box 88, Manchester, M60 1QD, United Kingdom
e-mail: d.barr@postgrad.manchester.ac.uk; p.dudek@manchester.ac.uk

Abstract – **We present a software environment for the efficient simulation of cellular processor arrays (CPAs). This software is used to explore algorithms that are designed for CPAs, neuromorphic arrays, multi-layer neural networks and vision chips. The software (APRON) uses a highly optimised core combined with a flexible compiler to provide the user with tools for the prototyping of new array hardware and the emulation of existing devices. We show that software processor arrays can operate at impressive speeds, with high numerical accuracy. APRON can be configured to use additional processing hardware if necessary, and can even be used as the graphical user interface for new or existing CPA systems.**

I. INTRODUCTION

Cellular processor arrays (CPAs), such as the ones presented in [1-3] implement data processing at a fine-grain level of parallelism. Unlike sequential and coarse-grain parallel processing systems, such as multi-core processors [4], which have large and complex instruction sets, CPAs are often comprised of simpler processors, with specific computational ability. Typically they are used at the lower-level of systems, to perform domain-wide processing, i.e. every processor executes the same instruction, but operates on their own local memories. This is known as the Single Instruction, Multiple Data (SIMD) paradigm. By using simpler processors, there are often high numbers of them on a device, usually arranged in a 2D grid. The massively parallel nature of the device and spatial positioning of the processors give additional performance advantages, in particular with the communication between processors, as each is able to exchange data with its neighbours. A non-parallel system would have to perform many instructions, including memory accesses to accomplish the same task. The power consumption of a CPA, often several orders of magnitude lower than that of an equivalently performing sequential system (such as a desktop PC or workstation), is an appealing property for designers of low power embedded systems.

The highly parallel nature of CPAs has lead to their increased use in a variety of fields, where many simple computations must be performed, and local connectivity would usually be a memory access bottleneck, such as vision chips [2], high-speed image processing [1][5], cellular neural networks [3] and computational neuroscience models [6]. These applications only require simple numerical operations and neighbour communication, but use operations that are applied homogeneously to large data-sets, at high speeds. The fine-grain nature of many CPAs means thousands of local memories can be operated on simultaneously and the simplicity of the processors allow fast execution of instructions.

The design of CPAs presents a set of challenges. It requires difficult trade-offs and design decisions, such as accuracy or speed, analogue or digital, and memory or array size. The drive to achieve a high density of processors in a single chip often makes memory a limited resource and reduces their computational ability. Also, the specialised nature of the hardware often makes it less accessible to potential application developers, while typically used short word-lengths or analogue storage techniques limit their accuracy. Further to this, many CPAs require additional interfaces and control systems which are specifically built for the device, introducing additional design costs and possibly restricting the performance of the CPA device.

This paper presents APRON (Array Processing enviRONment), a high-speed, flexible, virtual CPA implemented entirely in software. It has two objectives: firstly, to provide an extensible, customisable implementation of a general purpose array processing engine, which is a basis for the design and "virtual prototyping" of new CPAs. Secondly to provide a high-speed emulation of any CPA-based device, allowing the user to design pixel-parallel image processing algorithms and explore multi-layer neural-networks, cellular automata and other phenomena related to CPAs, with a variety of tools for data-analysis, performance evaluation and algorithm development. APRON software is written to take advantage of the high-speeds and accuracies of modern desktop processors to provide optimised array processing performance. To this end it is not intended to replace low-power embedded devices, but serves as a fast, accurate substitute in environments where custom hardware is not available. APRON has been designed to provide user-friendly tools and achieve the maximum computational performance, with minimal overheads, without the need for custom low-level implementations.

This paper is organised as follows: Section II presents the APRON simulation core and the integrated development environment (IDE); Section III presents the potential applications/research fields where APRON can be used effectively, and how the system can emulate several types of CPA device simultaneously; Section IV highlights the flexible

978-1-4244-2090-2/08/$25.00 ©2008 IEEE

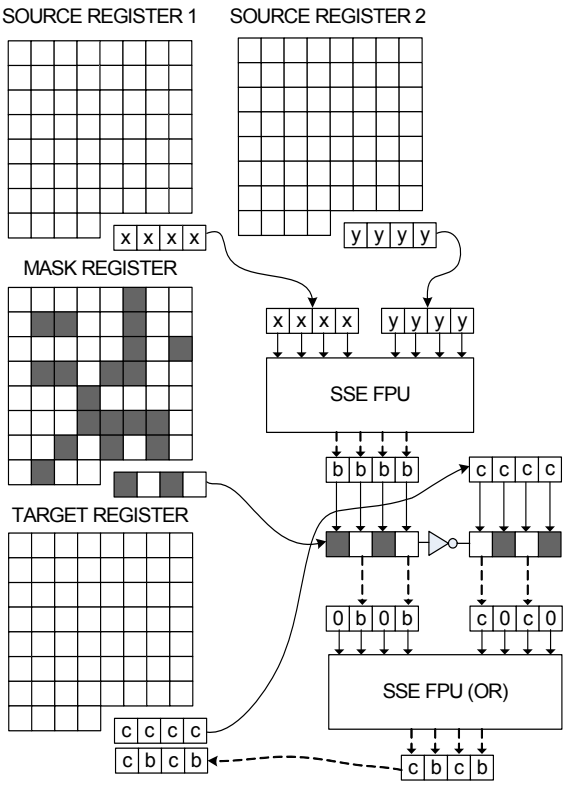


Figure 1 – Disabling processor elements is achieved by restricting writes to the register memory by using the mask in a sequence of logic operations. Dashed lines are memory writes and others are reads.

architecture of the APRON compiler, and illustrates how it can be modified to emulate or prototype other hardware devices; and Section V presents a performance analysis and benchmarks of the APRON system. Throughout the paper, performance figures are presented, which are summarised in the conclusion.

II. APRON SYSTEM OVERVIEW

a) The Simulation Core

The APRON simulation core is a platform-independent module that can be used in many applications for fast array processing in software. The core provides a virtual array processor, operating on three data-types with a flexible instruction set and a virtual micro-controller. The micro-controller controls the program counter which permits non-linear sequencing of algorithm instructions. APRON by default uses 32-bit single precision floating point numeric representations in all operations but it is possible to change this if necessary. The data-types are *registers*, *kernels* and *linkmaps*.

Registers are two dimensional arrays of numeric elements called *cells*, which are 16-byte aligned in memory contiguously, to facilitate cache/page management and to take advantage of the extended x86 intrinsic architectures: SSE (Streaming SIMD Extensions) and SSE2 [7]. These architectures provide an SIMD vector processor in the silicon fabric of the CPU. This vector processor can process four single precision floats in a minimal number of clock-cycles, providing a calculation throughput much faster than that of the standard FPU. SSE also provides cache management facilities, meaning registers can be pre-cached close to the processor, and even read/written straight from/to the main system memory, without polluting the cache if necessary. Registers adopt one of two forms. A *data register* contains numeric data, and a *mask register* contains binary data, which is used to enable and disable the write privileges to a data register, in effect permitting regional register operations (where global register operations are default). For many operations, the target register can also be a source register due to the sequencing of instructions sent to the CPU (e.g. giving x=x+1 functionality). Figure 1 shows how data is sequenced, sent to the vector processor, and combined with a mask register to emulate the effect of disabling a processing element's 'activity flag', providing local autonomy. The contents of the target register should only be updated if the mask register's corresponding bit is set. This requires additional logic when working with a vector of four elements, of which only some of the corresponding mask register bits are set, to form a combined result vector, which is written to the target register.

A kernel is an NxN matrix used in two dimensional convolution/filtering operations and is applied to a register. Kernels come in two forms. A *standard kernel* is used in typical greyscale image filtering operations (convolutions), such as Gaussian blur and gradient-detect. A *template kernel* is a binary kernel used for binary pattern matching, comprising of hits, misses and don't cares. A logic OR operation is performed within the domain of the kernel and the register and the binary result placed in a result register. Unfortunately, SSE2 does not offer a simple way of executing the kernel operation quickly, due to a lack of efficient sideways computation within the vector processor; (however SSE3 does support this, and is becoming standard on modern processors). Instead, APRON ensures all data required is in the highest level of cache, as close to the processor as possible, reducing cache misses (and eliminating paging) and the OS is stopped from swapping out the APRON process for the duration of the operation. The code is written in such a way as to be optimised greatly by the C++ compiler. The disassembly shows 95 instructions are required for a fully filtered pixel using a 3x3 matrix.

Linkmaps are a structured list of all the connections from a single cell. APRON supports local neighbour connectivity by default, so registers can be shifted and rotated in the 4 standard compass directions, but by having no hardware connectivity limitations, it is possible to have any cell connected to any number of cells anywhere within the register. Linkmaps exist for each cell in the register, allowing unlimited connectivity possibilities. Each link is also accompanied by a multiplicative coefficient which is analogous to a synaptic weight (See Figure 2). When a linkmap is used, each source register value is multiplied by each of its connections defined by the linkmap. Each target register value is the accumulated result of all the multiplicative connections that link to that value's location. The APRON IDE allows the algorithmic generation of linkmaps through a built-in Python interpreter, and a set of parametric maps (registers pre-filled with data), which means the

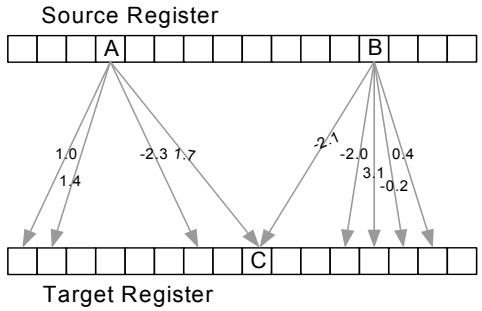


Figure 2 – Linkmaps allow multiplicative connections between the processing cells. Each cell can be connected to any other cell. Here different linkmaps are shown for cells *A* and *B*, where *C=1.7A-2.1B*.

connectivity output from a register's cell can vary in a spatially controlled way. Linkmaps can become large, for a fully specified linkmap of a register size NxN, N^4 connections are required. Often this is not the case, so connections below a specified value are assumed to be neglible (or zero) and are omitted from the linkmap list.

A variation of the linkmap is the *parametric linkmap*, in which the connectivity is described parametrically, for example, all of the connections are based on the same Gaussian equation, but different coefficients can change the shape of that equation. Currently, this is computationally time consuming and is the subject of ongoing research, as the potential of using additional special-purpose hardware to perform this task, could greatly reduce the memory bandwidth requirements of a fully connected linkmap.

The simulation core has been designed to be extensible, so if necessary additional hardware (such as FPGAs or existing processor arrays) can be used to increase the computation speed. Currently the core supports 55 operations/instructions, including mathematical operators, register translation, filtering, linkmap training and external data transfer.

b) Integrated Development Environment (IDE)

To accompany the simulation core, an IDE has been developed. This environment has two parts, the editor and the simulator. The editor is used for creating projects, which describe the algorithms and any constraints, such as maximum number of registers and array size, limitations often faced by hardware developers. Registers can be pre-filled with data algorithmically (using Python script) or loaded with images. A special purpose scripting language called 'APRON-Script' has been developed, which is easy to use and it serves as the underlying description mechanism for all simulations, prototypes and algorithms. All APRON-Script code can be checked for syntax errors and validated automatically. The scripting language does not require variable declaration and operates on a 'one line of code is one hardware instruction' basis. This makes looking for performance bottlenecks and optimisation easier.

$$G_x = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \qquad G_y = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A$$

$$G = \sqrt{G_x^2 + G_y^2}$$

Figure 3 – The Sobel Operator equations, used in APRON benchmarks, where G is the result of a filtered pixel at location (x, y), and A is the unfiltered image.

The simulator provides tools for executing and analysing algorithms. All registers are visually viewable as 2D images, and can be numerically inspected. The user can step through code, or enter a "Run" mode, which executes the algorithms as fast as possible. For long simulations on slower computers, a low-priority run mode is provided. The simulator also provides dynamic input/output. Input can come from data files or captured from an imaging device in real-time. Register contents can be written back as image or data files. The core could be configured to communicate data from/to other sources such as network sockets, allowing the possibility for simulation of large-scale models. An APRON script algorithm can also send interrupts to the simulator to perform tasks, such as update the display, reposition windows, display performance information, issue breakpoints and display run-time error information. Simulations can be terminated at any time. Each instruction is timed and displayed, for identifying performance bottlenecks.

III. APPLICATIONS

a) Generic Processor Array

APRON has been optimised for speed and accuracy. Using just the default APRON-Script instructions, it is possible to implement a wide range of array processing algorithms. Essentially, APRON is still a serial processing system, and so the smaller the registers, the better the performance. The benchmarks in Figure 5 show that for simple numeric operations, global register (128x128) operations occur in several micro-seconds, meaning potentially tens of thousands of register operations can be performed per second. This would imply APRON could potentially be used in both practical and research applications. Algorithms are written in default APRON-Script, which in this scenario can be considered a high-level language. The optimised simulation core allows APRON to outperform popular alternatives such as MATLAB and regular (naïve) C/C++ implementations.

b) Neuromorphic Arrays

The high numerical precision and specific instruction set of APRON enables the implementation of many types of neural network models, using array registers in a way similar to that in [6]. Unlike hardware processor arrays, APRON has fewer limitations on resources used, supporting over 32,000 registers (of a maximum 2048x2048 elements). Neurons are arranged into homogeneous layers the size of a register, however the neuron

parameters need not be homogeneous. For example, a layer of Izhikevich [8] neurons can be implemented using 2 registers to store the states, 1 register to store the spike output, and several temporary registers for calculating the state update. Network connectivity is provided through the use of linkmaps described previously. Linkmaps also provide a built-in learning mechanism akin to Spike Time Dependent Plasticity (STDP), where a large range of learning functions can be used through the transformation of a base function, and through the manipulation of time-decay throughout the simulation. On a 1.8GHz Intel Core 2 Duo processor, an Izhikevich neuron takes approximately 21ns to update, potentially allowing over 45,000 neurons to be updated in real-time with a 1ms time-step. Gaussian projective fields can be implemented either through linkmaps, or by using a Gaussian blurring filter on a register filled with the output of the layer. A single 3x3 blurring filter applied to a 128x128 register takes approximately 170µs.

c) Vision Systems

The APRON simulator allows registers to be filled with data acquired from an imaging device (e.g. a webcam). Therefore live image data can be used during simulation. When operating in "Run" mode, many algorithms execute fast enough to be interactive, and responsive to the visual input. Taking Sobel Edge detection as an example (See Figure 3), APRON can complete this algorithm for a 128x128 array in 0.5ms. Imaging devices vary in capturing/transfer time, but webcams typically take between 10 and 100ms to both capture and transfer an image into APRON. Therefore, a complete capture-process cycle can take 10.5ms, giving a frame-rate of 95fps. Higher quality cameras may bring this down to 2ms, giving 400fps. Vision systems implemented in the APRON virtual array processor can be considered similar to pixel-parallel vision chip systems [9]. For 3x3 filtering of a 128x128 array, APRON has a throughput of 81MP/s (Mega-Pixels/Second) and for simple operations (add/multiply) 958MP/s; for 512x512 this is 51MP/s and 244MP/s respectively. For comparison, in [10], a 3x3 filter on a 3.0Ghz Pentium4 processor gives 9.7MP/s; a Xilinx Virtex-II Pro, 202MP/s; and an nVidia 6800 Ultra GPU, 278MP/s. These results do not take into consideration any other processing or communication requirements, unlike APRON. It may often be the case that actually transferring the data to/from these devices has a substantially negative effect on the performance of the system as a whole.

Combining the 'virtual' vision chip with the neuromorphic array results in a powerful modelling tool suitable for use in retinal and visual-cortical simulations, often in a more controllable environment than that delivered by hardware-oriented systems. This flexibility allows the developer to create entire "system-on-virtual-chip" applications that may execute at speeds suitable for practical application use.

d) Cellular Neural Networks

APRON is easily configured to behave as a CNN (e.g. implementing the classical Chua model[11]). As with the Vision System description previously, an imaging device can be used as input. The performance of the model varies with the array size and the type of algorithm executing (timing and iteration requirements vary). For example, APRON can perform the 'greyscale feedback convolution' task in 180µs, which could be compared with results presented in [3].

IV. HARDWARE PROTOTYPING/EMULATION

One definite benefit a software CPA has over hardware is the relatively unlimited possibilities for processor element complexity, communication strategy and memory resources. APRON provides many optimised operations (here considered low-level) which can be combined to form higher-level more complex macros. This ensures that the CPA designer is using optimised code to implement the functionality of a cell's processing element. APRON provides at the very least the ability to read and write to each cell individually, which is sufficient to implement the most complex and diverse operations.

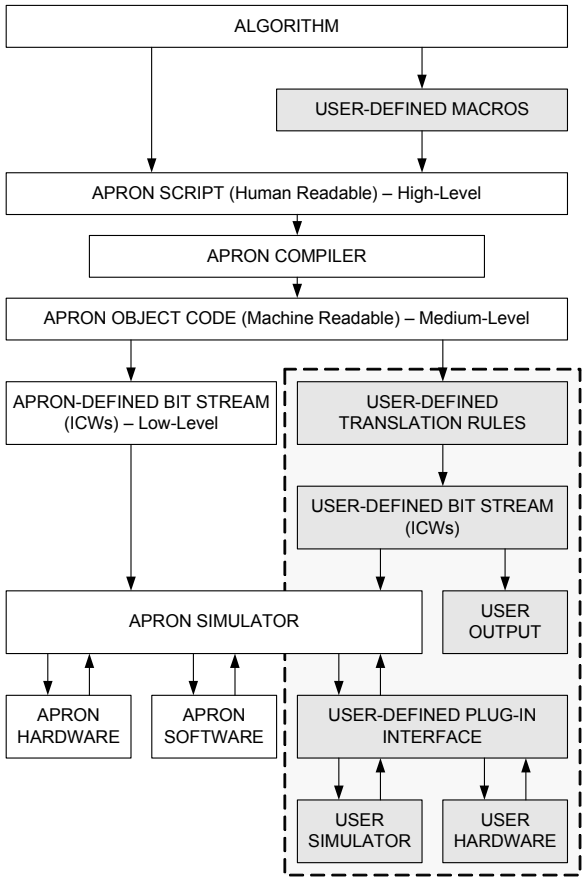


Figure 4 – APRON Compilation/System Diagram. The shaded regions show user configurable components that customize APRON. The boxed region indicates components that need to be programmed and imported as a dynamic library.

Figure 4 shows the data flow through the APRON compiler and simulator. The APRON compiler has three levels of abstraction. The highest is the scripting language itself, which is human readable, and consists of a combination of APRON instructions, and macros. A macro is similar to a function, in that it encapsulates many APRON instructions into what appears to be a single instruction, except it becomes explicitly unrolled into the instruction sequence rather than placed onto the stack when called. Macros have global bodies, i.e. they can access any resource, but have local labels and variables as well, to maintain the encapsulation. For example in the scenario where the same macro is called twice, it is important not to have one macro jump to instructions in the other macro. Macros are defined in an external 'rule-set' file (APRON-Script syntax), which also defines resource name changing, constant declaration, operator overloading and some environmental properties such as Register dimensions. The simulator provides a facility to debug the macros verbosely, to find errors in the rule-set.

Using macros, a cellular processor array designer could specify the complete instruction set of a hardware device, knowing the APRON instructions will deliver repeatable results. This is useful for both the emulation of existing hardware and the prototyping of new hardware. The designer could build several rule-sets, describing the hardware at different levels of abstraction, from an accurate, slower model of the current or bit transfers/operations in the individual cells, to a faster, abstract behavioural model of the device. The APRON IDE already provides the tools for the development and analysis of algorithms, and a fast interactive simulation tool, therefore the designer does not have to create any software himself. The 'rule-set' can be distributed to any user that wishes to develop algorithms for the particular CPA. This approach has been applied successfully to the ASPA vision chip [1], and APRON is used for algorithm development for that device. APRON will also be used for the development of SCAMP devices [2].

The middle-layer of abstraction is APRON object-code, which is a machine readable structure defining each instruction in the APRON-Script algorithm, after all necessary pre-processing (such as macro unrolling) has been applied. This structure breaks down the script instruction into a function name, and a variety of parameters. The object code by default is then turned into a binary Instruction Code Word (ICW) stream. A single object-code instruction can consist of many ICWs. The ICW stream is the lowest-level of abstraction in the APRON system. This stream is delivered to the APRON Core for execution

The module responsible for the translation from object-code to an ICW stream is hot-swappable, with a defined interface, meaning it is possible for designers to create their own ICW streams. This has several benefits. 1) The user can use the APRON environment as a code editor/compiler for a custom CPA; 2) The ICWs can be delivered to other applications, and even hardware devices; 3) The ICWs can be delivered to a custom simulation core, bypassing the APRON Core entirely, providing a highly customised simulation, but utilising the development environment and tools APRON provides; 4) Similar to 3, but actually using a hardware device, and using APRON as a software interface or graphical user interface to the hardware.

Table 1 – Descriptions of the Benchmark Tests performed

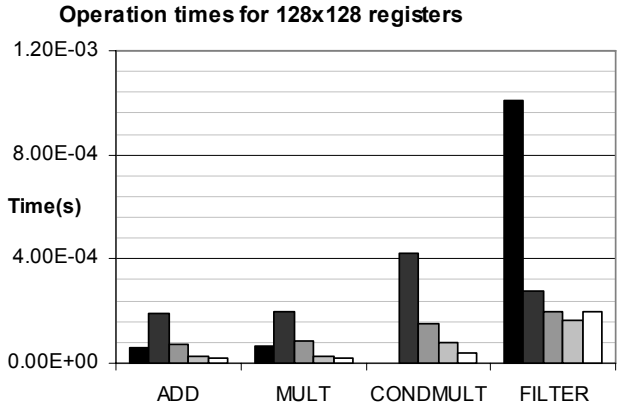
Test	Description
ADD	Two registers were filled with random values. The two registers were added and written to a third separate register 10000 times.
MULT	Same as ADD, but with a multiply operation.
CONDMULT	Same as MULT, but the two registers were multiplied, only where the contents of register 1 are greater than θ, using a mask register
FILTER	One register was filled with random values. A 3x3 Gaussian Blur Kernel was used to filter the register and the result was written to a second separate register, repeated 10000 times.

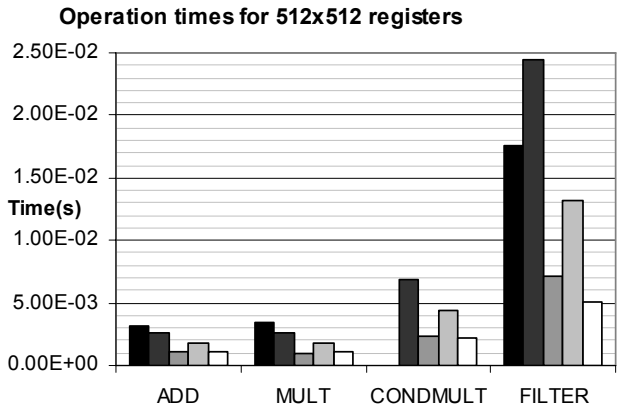
V. BENCHMARKING

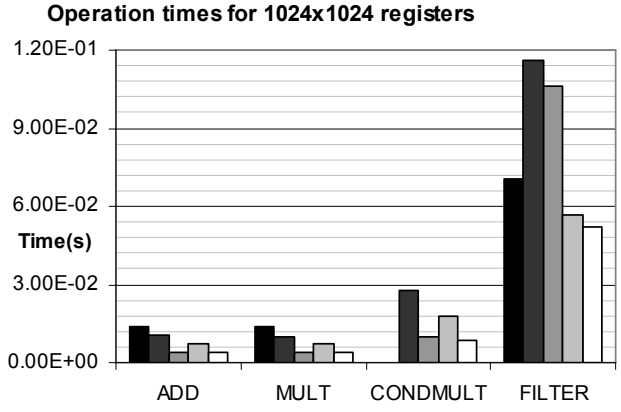
Throughout the development of APRON, a great effort has been made into optimizing the APRON Core, to make it as fast as possible. In order to assess its performance, a set of benchmarks were compiled that measure the execution time of commonly used operations. The tests performed are described in Table 1, and the results shown in Figure 5.

Two test machines were used with similar specifications, a 1.8GHz AMD Opteron 265, running Microsoft Windows XP, and a 1.8GHz Intel Core2 Duo running in Windows Vista. It was not possible to benchmark MATLAB programs on the Intel platform at this time. The benchmark results show that APRON is faster than MATLAB at all tasks, often by a factor of 3, and faster than a naïve C++ approach in almost all cases. The results have shown interesting differences between Intel and AMD processors, with AMD being on the whole slower. This is most likely due to the nature of AMD's implementation of Intel's SSE technology. The operating system's memory/process management system also contributes to the performance and this can be seen in the results of the larger register filtering tests, as SSE is not used for this operation.

APRON performs 128x128 array operations using single precision floats in tens of microseconds, which is suitable for a great range of experiments and applications. Execution times can largely be reduced four-fold if the change to using 8-bit integers is made.







	AMD MATLAB, 1.8GHz Opteron 265, 1GB RAM
	AMD Naïve C++, 1.8GHz Opteron 265, 1GB RAM
	INTEL Naïve C++, 1.8GHz Core2 Duo, 2GB RAM
	AMD APRON, 1.8GHz Opteron 265, 1GB RAM
	INTEL APRON, 1.8GHz Core2 Duo, 2GB RAM

Figure 5 – Benchmarks showing the performance of different array processing operations on different software platforms.

VI. CONCLUSION

This paper introduced an efficient software package used for the simulation and prototyping of cellular processor arrays. This software is useful throughout the life cycle of a cellular processor array based-device, from the initial design, modelling, and prototyping of the hardware; through to being used as an algorithm development platform, simulator and even a hardware interface. APRON software proves to be faster than other software (and some hardware) environments for the tasks of emulating, prototyping and simulating cellular processor arrays, and can even be used as a stand-alone 'array processing' system in many applications.

It is interesting to see that 'virtual' massively-parallel devices created in software are beginning to compete with custom ICs and FPGAs in terms of computational performance, at comparatively lower cost. With commercial CPU manufacturers producing more parallel, lower power devices, perhaps software environments such as APRON will become more appealing to researchers interested in fine-grain processor arrays, computational neuroscience and image processing.

ACKNOWLEDGEMENTS

This work has been supported by the EPSRC; grant number: EP/C516303

REFERENCES

[1A.Lopich and P.Dudek, "Global operations on SIMD cellular processor arrays: towards functional asynchronism", International Workshop on Computer Architectures for Machine Perception and Sensing, CAMPS 2006, pp.18-23, September 2006

[2] P.Dudek, "Implementation of SIMD Vision Chip with 128x128 Array of Analogue Processing Elements", IEEE International Symposium on Circuits and Systems, ISCAS 2005, Kobe, pp.5806-5809 , May 2005

[3] A. Zarandy, M. Foldesy, P. Szolgay, S. Tokes, C. Rekeczky and T. Roska, "Various implementations of topographic, sensory, cellular wave computers" ISCAS 2005. 23-26 May 2005 Page(s):5802 - 5805 Vol. 6

[4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. Shippy, "Introduction to the Cell multiprocessor", IBM 2005, http://www.research.ibm.com/journal/rd/494/kahle.pdf

[5] C.Alonso-Montes, P.Dudek, D.L.Vilarino and M.G.Penedo, "On Chip implementation of a Pixel-Parallel Approach for Retinal Vessel Tree Extraction", European Conference on Circuit Theory and Design, ECCTD 2007, pp.511-514, Seville, Spain, August 2007

[6] D.R.W. Barr, P. Dudek, J. Chambers and K. Gurney, "Implementation of multi-layer leaky integrator networks on a cellular processor array", IJCNN 2007, Orlando, Florida

[7] Intel, AP-809, "Real and Complex FIR Filter Using Streaming SIMD Extensions", version 2.1, 1999, http://cache-www.intel.com/cd/00/00/01/76/17650_rc_fir.pdf

[8] E. M. Izhikevich, "Simple model of spiking neurons," IEEE Transactions on Neural Networks, vol. 14, pp. 1569-72, 2003.

[9] D.R.W. Barr, S.J. Carey, A. Lopich and P. Dudek, "A Control System for a Cellular Processor Array", IEEE International Workshop on Cellular Neural Networks and their Applications, CNNA 2006, pp.176-181

[10] B. Cope, P.Y.K. Cheung, W. Luk, S. Witt, "Have GPUs made FPGAs redundant in the field of video processing?", IEEE International Conference on Field-Programmable Technology, pp. 111-118, 11-14 Dec. 2005

[11] L.O. Chua, L.Yang, "Cellular Neural Networks: Theory and Applications", IEEE Transactions on Circuits and Systems, vol. 35, pp.1257-90, 1988