OPTIMIZING THE SOM FOR THE PLAYSTATION 2

Stephen McGlinchey
University of Paisley
Paisley, Scotland
stephen.m@jet5.com

Abstract – *The use of neural networks in computer game AI is a rapidly expanding area of academic research; however, the games industry has been slow to adopt new techniques, and one reason behind this is the scarcity of computational resources in modern games. The Sony PlayStation 2® provides parallel hardware features that can be exploited to implement a highly optimized SOM. This paper summarises the main hardware features of the platform, and presents some optimizations that can be made to the SOM algorithm to tailor it to the hardware. This work makes the SOM a viable option for AI in games, where processing resources are used intensively. The principles presented are also applicable to other modern parallel processors such as "The Cell," and is therefore relevant to the next generation of games consoles and other devices.*

Key words – optimization, games, parallel processing

1 Introduction

For several years now, it has been predicted [1,2] that online learning in games will be the "next big thing" in game AI; however, this prediction has still to be realized. A recent review [3] cited some of the few successful commercial games that have employed online neural network learning as part of the game AI, and also summarized several principal reasons why the industry has been reluctant to use neural networks. One of the perceived problems is that the computational cost of neural networks is prohibitively high, especially since most games use the vast majority of processing resources on the graphics pipeline. Even though the SOM has a relatively low computational cost when compared with some other neural networks (during training and with subsequent use), this is still a factor that has deterred its use in games.

The PlayStation 2, which will hereafter be referred to as the PS2, currently holds greater than 60% share of the home video games market, and offers parallel processing features that are often not fully exploited. This paper shows how the specialist hardware of this platform can be used for the SOM. Firstly, we give a brief introduction to the hardware features of the PS2, and then we describe implementations of low-level operations used with the SOM and its common variants, fine-tuned for the PS2. Finally, we discuss the use of the PS2's DMA controller for caching codebook vectors and input/output data, using the double-buffer used by one of the vector operation processors. This feature allows significant optimizations to be made to the SOM, which are not possible with many other neural networks.

2 Overview of the Emotion Engine

The “emotion engine” (EE) [4] is the CPU of the PlayStation 2, although this unit is actually composed of several processors. Fig 1 shows some of the main components that make up the EE, along with its connection to the Graphics Synthesizer (GS), which is viewed as external to the EE. The EE core is the CPU of the EE, which runs the MIPS III (and partly MIPS IV) instruction set. This unit is also connected to a floating point co-processor (COP1). In addition to instruction and data caches, the EE core has 16kb of scratchpad RAM, which can be used to reduce accesses to main RAM, leaving the main bus for use by the other processors and the DMA controller (DMAC).

Two vector operation processors (VPU0 and VPU1) are provided for processing 4-dimensional vectors and 4x4 matrices. These units are described in more detail in the next section. The Graphics Interface Unit (GIF) is a device that arbitrates between the three data paths from VU1, VIF1 and the main bus, and passes data on to the GS in the form of 2D drawing primitives. The components are connected by a 128-bit main bus, which runs at 300 MHz. The DMAC provides nine DMA channels that transfer data between main RAM and the various other components of the system.

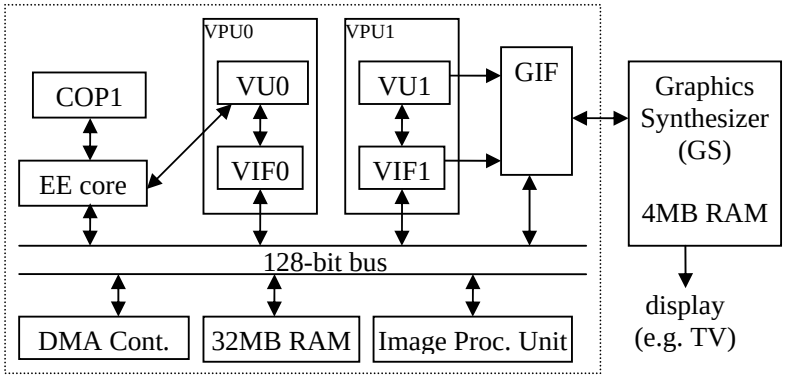

Fig. 1. *The architecture of the emotion engine (EE). (Not all components are shown.)*

2.1 Vector Operation Processors (VPU)

The EE has two specialized processors designed for processing 4-dimensional homogenous co-ordinate data for 3D graphics. These units are composed of a vector unit (VU), which is the vector processor, and a VU interface (VIF). The main purpose of the VIF is to decompress data, so that it can be held in a more compact format in main RAM before being transferred and expanded into VU data memory. Both VUs have 32 128-bit registers, 16 16-bit integer registers, and four floating point ALUs (a.k.a. FMACs), which operate in parallel. The 128-bit registers hold four 32-bit floating point values in fields labelled x, y, z and w. Both VUs can operate in “micro mode” which means that they execute micro-programs stored in their instruction memory, but VU0 can also work in “macro mode”, which is where it behaves as a second co-processor for the EE core. In micro-mode, both vector units have two parallel execution units, allowing simultaneous execution of some instructions. VPU1 has 16k of instruction memory and 16k of data memory, but VU0 has only 4k of each. Unlike VU0, VU1 can send geometry data directly to the GIF/GS without the need to use the main bus.

The VU instruction set caters for conditional branching, integer arithmetic, random number generation, fixed/floating point and integer conversion, view frustum clipping, synchronization and parallel floating point arithmetic including product-sum calculations. VU1 has in addition to these features, an elementary function unit (EFU) for performing trigonometric, exponential and logarithmic operations. Some of the EFU functions take far longer to perform than the basic arithmetic operations; therefore the EFU should not be over-used. It is possible for VU1 to schedule EFU computations, and then continue running other non-EFU instructions whilst waiting for the result.

2.2 Processing Pipeline

With a parallel architecture such as that provided by the EE core, there are various strategies that could be employed to efficiently use the available processors. There is no globally optimum strategy; the most suitable choice is application-specific. Games and 3D graphics engines that use the PS2's hardware efficiently generally employ the same strategy, which is described in table 1 below.

EE core & COP1	Game logic, AI, animation, user input processing, resource management (RAM, textures, etc), culling, scene-graph management, control of EE components.
VU0	Transformations to world and camera space, lighting.
VU1	Transformation to homogeneous and screen space, clipping, particles and other special effects.

Table 1. *Use of the EE's processors in the pipeline of games using 3D graphics.*

Games generally iterate around a main loop, with each iteration producing a rendered frame in the frame buffer. The pipeline is mainly a linear process; most of the tasks outlined in table 1 must be performed in a sequence due to the dependencies of later tasks on earlier tasks. The pipeline can be parallelized by using streaming to simultaneously render frames. Therefore, in rendering a frame, the EE core performs some tasks, and the output is then passed to VU0, which then performs more tasks, the results of which are subsequently passed to VU1. As soon as a processor passes the data for a frame on to the next processor, it can commence processing for the next frame, and it is common to processes two or three frames simultaneously.

3 Using the Vector Units for the SOM

In the previous section, we noted that AI is normally processed in the EE core, which is the most suitable processor for symbolic AI tasks such as tree traversal and searching. However, for computational methods such as the SOM, the vector units offer better facilities for two reasons: firstly, unlike COP1, they support parallel floating point operations (COP1 has only one FMAC), and secondly, the EFU contained in VU1 provides hardware support for mathematical functions that can be used for non-linear distance measures, probability distribution functions and neighbourhood functions. Another feature that can be beneficial is the provision of hardware pseudo-random number generators in both vector units. We will now consider these benefits in more detail.

3.1 Euclidean Distances between Points

As stated before, the VUs are designed for 4-dimensional vectors and 4x4 matrixes, hence the four FMACs in each VU. However, matrices and vectors of any size can be processed, and the following example shows the calculation of the Euclidean distance between an eight-dimensional codebook

vector and input vector. In listing 1, the input vector is supplied in three registers with labels “input1-3,” “input4-6” and “input7-8”. Three registers are required since the w field of these registers is not used because the “sub” instruction that we will use works only on the x, y and z fields. The codebook vector is supplied using the registers with labels “codebook1_1-3,” “codebook1_4-6” and “codebook1_7-8.” Line 1 clears the contents of the register labelled “sum,” which we will use to store some intermediate results and the final result. The second line subtracts the first three dimensions of the input vector from the codebook vector. These are squared and added together in line three, and the result is placed in a special register “P”. The result can be copied from the P register into the x field of the register labelled “sum.” Lines 5-7 and 8-10 perform the same operation for dimensions 4-6 and 7-8 respectively, storing the result in the y and z fields of “sum.” Lines 11 and 12 perform the addition of the x, y and z fields of “sum”, and store the result in the x field of “sum.” Finally, the square root is calculated on line 13, and the final result is placed in the x field of “sum.” The code is written for the VCL pre-processor, which is a tool widely used by PS2 developers to perform register allocation and scheduling of instructions for the dual execution units.

```
 1   add.xyzw   sum, vf00, vf00      ; clear the contents of sum
 2   sub.xyz    diff1_1-3, codebook1_1-3, input1-3
 3   esadd      P, diff1_1-3 ; add the x, y and z components
 4   mfp.x      sum          ; retrieve the result from the P reg
 5   sub.xyz    diff1_4-6, codebook1_4-6, input4-6
 6   esadd      P, diff1_1-3
 7   mfp.y      sum
 8   sub.xy     diff1_7-8, codebook1_7-8, input7-8
 9   esadd      P, diff1_7-8
10   mfp.z      sum
11   esum       sum
12   mfp.x      sum
13   esqrt      P, sum
14   mfp.x      result
```

Listing 1: *Calculation of the Euclidean distance between an input vector and a codebook vector.*

Some variants of the SOM e.g. [4] have used a matrix-vector product instead of the distance measure, and this can take fuller advantage of the parallel features of the VU, using all four FMACs simultaneously. The code below uses three VU instructions for parallel computations: “`mula`”, “`madda`” and “`madd`”. Each of these instructions takes three operands, the first one being the location to store the result, the other two are the inputs to the operation. The “`mula`” instruction multiplies the corresponding fields of the two inputs together, and stores the result in the 128-bit accumulator register. The “`madda`” instruction does the same product, and adds it to the value already held in the accumulator, storing the result in the accumulator. Finally, “`madd`” does the same as “`madda`”, except the result is stored in a floating point register. Following each instruction, the fields that should be used in the parallel calculation are specified (e.g. `xyzw`). The code below shows an example of a 6-D vector multiplied by a 6x5 matrix. The input vector is supplied in two registers: fields x,y,z and w of the register labelled “`input1-4`”, and fields x and y of “`input5-6`.” Each matrix column is held in fields x, y, z and w of registers labelled `col`**`x`**`_1-4`, and field x of `col`**`x`**`_5`. The 5-D result is held in registers with labels `output_1-4` and `output_5`.

```
mula.xyzw  acc, col1_1-4, input1-4.x
madda.xyxw acc, col2_1-4, input1-4.y
madda.xyzw acc, col3_1-4, input1-4.z
```

```
madda.xyzw acc, col4_1-4, input1-4.w
madda.xyzw acc, col5_1-4, input5-6.x
madd.xyzw output_1-4, col6_1-4, input1-6.y
mula.x  acc, col1_5, input1-4.x
madda.x acc, col2_5, input1-4.y
madda.x acc, col3_5, input1-4.z
madda.x acc, col4_5, input1-4.w
madda.x acc, col5_5, input5-6.x
madd.x output_5, col6_5, input1-6.y
```

Listing 2: *Calculation of a matrix-vector product.*

It is also possible to distribute the processing of a matrix-vector product across both vector units using standard parallel algorithms for dense matrices, which are fully described in [5]. This is something that is likely to be more readily exploitable with Sony's next generation console, which is expected to provide 32 separate vector processors.

3.2 Distance and Neighbourhood Functions

The commonly used difference of Gaussian distance function can be quickly calculated on VU1 using the EFU, although this facility is not provided on VU0. The "`eexp`" EFU instruction places the value of e^{-z} in to the P register, where z is the data passed in. Unfortunately, this instruction has a high latency of 54 clock cycles for the instruction to complete, although other operating that do not use the EFU can be performed whilst the result is being generated. Therefore, while the calculation is being performed, the VU processor should be used to perform another useful task, such as calculating the product-sum for other nodes. Another approach is to stream the data between the two vector units. In this strategy, the competition is performed by calculating the distances using the method described in section 3.1, selecting the winner, then passing the results to VU1 which then performs the weight updates using the EFU for the neighbourhood function. At the same time, VU0 can be finding the winning node for the next input vector. A minor drawback to this is that the winner selection for the second input pattern will not be taking into account the changes made to the codebook vectors in response to the first codebook. However, since the codebook holds similar values from one training iteration to a subsequent one, this has a negligible effect on training.

4 Caching of codebook vectors

The memory provided in the vector units for data is 4k and 16k for VPU0 and VPU1 respectively. For many applications, this is very restrictive. If each weight parameter is a single precision floating point value, then VPU1 can store only 4096 parameters. Moreover, this does not take into account any storage space for input vectors and intermediate data. An effective solution to this problem stems from an optimization that was proposed in [8] and has been used in implementations of very large SOMs such as in [9]. The optimization that we propose using is to confine the winner search to within a small neighbourhood of the previous winner. This is acceptable only if we assume that subsequent vectors that are presented to the network are similar, which is the case for many sources of data. In computer game AI, this condition can often be assumed, since the state of a game is similar between successive frames, as was the case in a previous study [10], which applied the SOM to the game of "Pong".

The size of the neighbourhood within which the winner is searched can simply be determined by the amount of VU data memory available. The SOM can be held in main RAM, and a moving window is

imposed on the neighbourhood of the current winner. This window is uploaded to VU Memory, and is searched for the new winner. If the new winner is a different node, then the window is moved along. The codebook vectors that are no longer within the window can be discarded, or if they have been changed, they are transferred back to main RAM. Any codebook vectors that are in the neighbourhood of the new winner, but not the old winner can be uploaded to VU Memory, replacing the vectors that left the neighbourhood.

As introduced in section 2, one of the features of the emotion engine that gives it such high throughput of data is the direct memory access controller (DMAC). When used with VPU1, the DMAC can perform data transfers to VU Memory in parallel with execution of micro-programs, even though they both access the same unit of memory. This is made possible using the technique of double-buffering, which allows data to be DMA transferred to and from one half (upper or lower) of VU memory, whilst the VPU processes data in the other half. This means that the process of swapping codebook vectors that move in and out of the neighbourhood of the winner can be done without halting the progress of training. Double-buffering is also useful for transferring input patterns to the vector unit, and input patterns may be uploaded in "episodes" of several at a time.

3 Conclusions

In this paper, we have given a brief overview of the hardware features of the Sony PlayStation 2, and we have shown how the SOM can be optimized for the specialist processors available. The optimizations include parallelized computation of Euclidean distances between vectors, matrix-vector products, hardware-accelerated calculation of non-linear distance functions and streaming operation between the two vector operation processors. These exploitations will make the SOM a more attractive AI method for use in computer games where processing resources are scarce, in which optimized processing is often essential. Previous studies (e.g. [10]) have shown the SOM to be a useful method for game AI due to its simplicity, its relatively low computational cost, and also its effectiveness in online learning; however, game developers have not adopted the SOM to any significant degree. This work has shown how its processing overhead may be further reduced, making the SOM a more viable option for computer games on the PS2.

As yet, there is little officially known about the next generation of games consoles, although Sony have announced that "The Cell" processor will be used in their forthcoming console. Like the Emotion Engine, this processor also offers independent vector processors: eight in total, each of which have 128k of local RAM, and are interconnected by a DMA controller. The vector processors can also be chained to allow fast streaming of data between a series of units. The optimizations presented in this paper are therefore relevant to the new hardware, which follows on from the success of the emotion engine in using vector processors and high-speed DMA. Future research in this area will focus on tailoring the methods presented for the new processing hardware.

References

[1] S. Rabin (2001) (Ed): *AI Game Programming Wisdom*. Charles River Media

[2] S. Rabin (2003) (Ed): *AI Game Programming Wisdom 2.* Charles River Media

[3] D. Charles, S. McGlinchey (2004) The Past, Present and Future of Artificial Neural Networks in Digital Games. *In proceedings of the Fifth International Conference on Computer Games: Artificial Intelligence, Design and Education* p163–169.

[4] S. McGlinchey & C. Fyfe (1997) An Angular Quantizing Self-Organizing Map for Scale Invariant Classification. *In proceedings of the Workshop on Self-Organizing Maps, pp 91-95* Espoo, Finland

[5] SCEI (2001) *EE User's manual*, 5th Edition, Sony Computer Entertainment Inc.

[6] A. Grama, A. Gupta, G. Karypis, V. Kumar (2003) *Introduction to Parallel Computing* (2nd Edition) Pearson.

[7] T. Kohonen (1997) *Self-Organizing Maps* Springer-Verlag, Berlin

[8] T. Kohonen (1996) *The speedy SOM* Technical Report A33, Helsinki University of Technology, Laboratory of Computer and Information Science, Espoo, Finland.

[9] K. Lagus, S. Kaski, T. Kohonen (2004) *Mining Massive Document Collections by the WEBSOM Method* Information Sciences : an international Journal 163**(3)** p135-156

[10] S. McGlinchey (2003) Learning of AI Players from Game Observation Data in proceedings of the fourth International Conference of Intelligent Games and Simulation, p106-110