

# **Game Programming Gems 4**

**Edited by Andrew Kirmse**



**CHARLES RIVER MEDIA, INC.**

**Hingham, Massachusetts**

Copyright 2004 by CHARLES RIVER MEDIA, INC.  
All rights reserved.

No part of this publication may be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means or media, electronic or mechanical, including, but not limited to, photocopy, recording, or scanning, without *prior permission in writing* from the publisher.

Publisher: Jenifer Niles  
Series Editor: Mark DeLaura  
Production: Publishers' Design and Production Services, Inc.  
Cover Design: The Printed Image  
Cover Image: "Delivering the Goods" from *MechAssaultIII* © Day 1 Studios 2003. © Microsoft © 2003.

CHARLES RIVER MEDIA, INC.  
10 Downer Avenue  
Hingham, Massachusetts 02043  
781-740-0400  
781-740-8816 (FAX)  
info@charlesriver.com  
www.charlesriver.com

This book is printed on acid-free paper.

Andrew Kirmse. *Game Programming Gems 4*.  
ISBN: 1-58450-295-9

All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

Library of Congress Cataloging-in-Publication Data  
Game programming gems 4 / edited by Andrew Kirmse.

p. cm.  
ISBN 1-58450-295-9 (Hardback with CD-ROM : alk. paper)  
1. Computer games—Programming. I. Kirmse, Andrew.  
QA76.76.C672G364 2004  
794.8'0285—dc22

2003023519

Printed in the United States of America  
04 7 6 5 4 3 2 First Edition

CHARLES RIVER MEDIA titles are available for site license or bulk purchase by institutions, user groups, corporations, etc. For additional information, please contact the Special Sales Department at 781-740-0400.

Requests for replacement of a defective CD-ROM must be accompanied by the original disc, your mailing address, telephone number, date of purchase and purchase price. Please state the nature of the problem, and send the information to CHARLES RIVER MEDIA, INC., 10 Downer Avenue, Hingham, Massachusetts 02043. CRM's sole obligation to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not on the operation or functionality of the product.

# Artificial Neural Networks on Programmable Graphics Hardware

**Thomas Rolfes**

tr@circensis.com

Artificial neural networks mimic biological information processing and are used in a wide range of applications where nonlinear mappings from input to output sets are required. Their evaluation in real-time systems, and the network training involved, are often computationally demanding. Chellapilla and Fogel evolved a small artificial neural network to play checkers at expert level [Chellapilla00]. It took a 400MHz Pentium II over six months to run 840 generations, although no particular effort was made to optimize the program.

The second generation of programmable *graphics processing units* (GPUs) introduced single and half precision floating-point texture formats and floating-point pixel pipelines. Until recently, the simulation of neural networks on consumer graphics hardware was limited to the use of clamped 8-bit blending, and CPU-based implementations were significantly faster. Today, GPUs have taken the vector processing performance lead over standard PC and console CPUs, with instruction and memory throughput being an order of magnitude higher. They are evolving rapidly into fully programmable vector coprocessors with a wide field of applications [GPGPU].

Efficient artificial neural network implementations are often based on numerical linear algebra libraries for scientific computing such as BLAS [Lawson79]. Initial efforts have been made to port subsets of BLAS to GPUs, and further work is in progress. By using GPUs as general-purpose vector processors, programmers can offload vectorizable routines from the CPU, benefit from high GPU performance, and find more opportunities for load balancing.

This article shows how an artificial neural network can be implemented using a GPU-based BLAS level 3 style single-precision general matrix-matrix product (SGEMM) [Dongarra88] and an activation function pixel shader<sup>\*</sup> under DirectX3D, version 9.

---

\* "Fragment shader" in OpenGL terminology

### CPU and GPU Architectures and Systems

Flynn [Flynn72] introduced a taxonomy based on the number of instruction streams and the number of data streams available. Current console and PC CPUs have partially superscalar SISD cores (single instruction stream, single data stream) and some have floating point SIMD (single instruction stream, multiple data streams) vector extensions, such as Intel SSE on the PC and Xbox CPUs, AltiVec on PowerPC (but not on the Gamecube PowerPC 750 CPU), and coprocessor vector units as in the Playstation2. The platforms employ either unified memory architectures (UMA), as on the Xbox, or nonunified (NUMA) with fast DMA paths, as on Gamecube, PS2, and PC in order to link the subsystems.

The programmable components of GPUs are the SIMD vertex and pixel shader units. Vertex shader units read vertex streams and execute vertex programs. Color values, texture coordinates, and other data are then interpolated and passed to pixel shader units, which execute pixel shader programs that can look up texture elements and compute on the other incoming data. Shader programs can be written in assembler and high-level languages. Current high-end GPUs have four vertex shader units and eight pixel shader units, fully working in parallel. The shader units are supported by fast caches and wide buses to the graphics memory.

The Playstation2 vector units can be programmed to work as vector/matrix kernels for artificial neural networks [PS2Neural]. It is also possible to use vertex programs for general-purpose stream processing, and vertex state shaders on the Xbox GPU allow you to persistently modify the constant registers. Future consoles are expected to be well suited for fast general-purpose vector computations.

### Artificial Neural Networks

Feed-forward networks, which are also known as *multilayer perceptrons*, are probably the most common architecture for artificial neural networks used in supervised learning. For an introduction to neural networks, see the primer in *Game Programming Gems* [LaMothe00] and the example in *Game Programming Gems 2* [Manslow01].

The evaluation of an  $n$ -layer feed-forward network with linear basis functions requires the computation of

$$\mathbf{a}_{i+1} = \text{act}(\mathbf{a}_i \cdot \mathbf{W}_i) \quad (4.8.1)$$

where  $\text{act}$  is an activation function and the  $\mathbf{W}_i$  are weight matrices. The vector  $\mathbf{a}_1$  is the input layer and holds the input data nodes and an optional bias node,  $\mathbf{a}_2 \dots \mathbf{a}_{n-1}$  are hidden inner layers, and  $\mathbf{a}_n$  is the output layer. The sizes of the vectors and weight matrices are usually different for each layer.

In the case of a parallel computation of  $m$  input sets, the vector-matrix products become matrix-matrix multiplications

$$\mathbf{A}_{i+1} = \text{act}(\mathbf{A}_i \cdot \mathbf{W}_i) \quad (4.8.2)$$

where the  $n-1$  matrices  $A_i$  have  $m$  rows. The generalized activation function now takes a matrix argument. To archive nonlinear mappings, nonlinear activation functions are used, including:

Gaussian:  $\exp(-a^2\sigma^{-2})$   
 Hardy's multiquadric:  $\sqrt{a^2 + \sigma^2}$   
 Hyperbolic tangent:  $\tanh(a) = (e^a - e^{-a}) / (e^a + e^{-a}) = \tanh_2(a/\ln(2))$   
 Sigmoid:  $(1 + \exp(-a/\sigma))^{-1}$

Efficient matrix-matrix kernels such as the optimized BLAS3 routines in ATLAS [Whaley98] use partitioning into submatrices to improve cache reuse. This technique is known as *block matrix multiplication*.

## Implementation

A number of authors, including Larsen & McAllister [Larsen01], Moravanszky [Moravanszky03], and Krüger & Westermann [Krüger03], show how to implement GPU-based dense and banded matrix multiplication efficiently. The basic concept is to store matrices as textures and to render quadrilaterals via vertex shaders with appropriately chosen vertex and texture coordinates. The interpolated texture coordinates are then passed to the pixel shader units for performing the actual parallel multiply-accumulate operations on the render target. Some implementations require post pixel shader blending, and this feature is not available on all hardware for floating-point buffers. The blending must then be performed in the pixel shader with two textures as input and a third texture as render target. This can be done repeatedly by rotating sources and targets. Sparse matrix multiplication can be achieved via lookup tables, and GPU implementations can benefit here from the high memory bandwidth and low access latency of the graphics systems.

After computing the matrix multiplication, the resulting render target becomes the input texture for the activation function. Again, a quadrilateral is rendered over the entire matrix-surface, or more than one if the matrix size exceeds the allowed texture size. Matrix multiplication and subsequent activation are repeated for each network layer. The pixel shader assembler code of a straightforward implementation of the hyperbolic tangent activation function looks as follows:

```
ps_2_0           // shader version
dcl_2d s0        // texture stage
dcl t0.xy        // texture coordinate

texld r0, t0, s0 // r0 <- texel.xyzw
mad r0, r0, c0.x, c0.y // r0 <- r0*scale+bias

// tanh (base 2)
exp r1.x, r0.x // r1 <- 2^r0
exp r1.y, r0.y
exp r1.z, r0.z
exp r1.w, r0.w
```

```

exp r2.x, -r0.x      // r2 <- 2^(-r0)
exp r2.y, -r0.y
exp r2.z, -r0.z
exp r2.w, -r0.w
add r3, r1, r2      // r3 <- r1+r2
sub r4, r1, r2      // r4 <- r1-r2
rcp r3.x, r3.x      // r3 <- 1/r3
rcp r3.y, r3.y
rcp r3.z, r3.z
rcp r3.w, r3.w
mul r0, r3, r4      // r0 <- r3*r4

// write result to output register
mov oC0, r0

```

While the hyperbolic tangent is naturally bounded to  $-1$  and  $1$ , it can be explicitly clamped by using the `min` and `max` instructions of the pixel shader version 2 language

```

max r0, r0, c0.z    // r0 <- maximum(r0, lower)
min r0, r0, c0.w    // r0 <- minimum(r0, upper)

```

or by using the saturate instruction modifier “`_sat`,” which clamps between 0 and 1 and can be used with any arithmetic instruction except the `fre` and `sincos` instructions. It costs no additional instruction slots.

```
mul_sat r0, r3, r4
```

The sample code to this article is a GPU implementation of Chellapilla’s and Fogel’s checkers position-evaluating feed-forward network. The computation is significantly faster on an ATI Radeon 9700 Pro than the version based on a SSE optimized SGEMM on a 3-GHz Pentium4. For large matrices, recursively applied algorithms such as Strassen [Strassen69] and Winograd [Winograd68] might yield further performance gains when combined with the simple submatrix multiplications.

## Conclusion

GPU implementations can yield good performance gains over CPU solutions. Current graphics hardware allows the processing of networks with capacities on the order of  $10^6$  nodes and  $10^8$  weights for an average connectivity of 100. This is far below the  $10^{11}$  neurons and  $10^{15}$  synapses of a human brain. Simulation hardware is rapidly becoming more powerful, and the true challenges of artificial neural network research will be network organization and learning algorithms.

## References

- Updated links to online versions of papers are available at [www.circensis.com/eg4.html](http://www.circensis.com/eg4.html).
- [Chellapilla00] Chellapilla, K., and D. B. Fogel, “Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved Checkers Program against Commercially

- Available Software," *Proceedings of the 2000 Congress on Evolutionary Computation*, IEEE Press, Piscataway, NJ, pp. 857–863, available online at [www.natural-selection.com/NSIPublicationsOnline.htm](http://www.natural-selection.com/NSIPublicationsOnline.htm).
- [Dongarra88] Dongarra, J.J., J. Du Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.*, Vol. 14 (1988), pp. 1–17.
- [Flynn72] Flynn, M., "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Computers* Vol. 21, 9 (1972), pp. 984–960.
- [GPGPU] "General Purpose Computing Using Graphics Hardware," available online at [www.gpgpu.org](http://www.gpgpu.org).
- [Krüger03] Krüger, J. and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," *SIGGRAPH 2003 conference proceedings*, available online at [www.cg.in.tum.de/Research/Publications/LinAlg](http://www.cg.in.tum.de/Research/Publications/LinAlg).
- [LaMothe00] LaMothe, A., "A Neural-Net Primer," *Game Programming Gems*, Charles River Media, 2000.
- [Larsen01] Larsen, E.S. and D. McAllister, "Fast Matrix Multiplies Using Graphics Hardware," *Super Computing 2001 Conference*, Denver, CO, November 2001. Available online at [www.sc2001.org/papers/pap.pap313.pdf](http://www.sc2001.org/papers/pap.pap313.pdf).
- [Lawson79] Lawson, C., R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for FORTRAN Usage," *ACM Trans. Math. Software* Vol. 5 (1979), pp. 308–371.
- [Manslow01] Manslow, J., "Using a Neural Network in a Game: A Concrete Example," *Game Programming Gems 2*, Charles River Media, 2001.
- [Moravanszky03] Moravanszky, Á., "Dense Matrix Algebra on the GPU," to appear in *ShaderX<sup>2</sup>*, Wordware Publishing, 2003, available online at [www.shaderx2.com/shaderx.pdf](http://www.shaderx2.com/shaderx.pdf).
- [PS2Neural] "PS2 Neural Network Simulator," project site online at <https://playstation2-linux.com/projects/ps2neural>.
- [Strassen69] Strassen, V., "Gaussian Elimination Is Not Optimal," *Numerische Mathematik*, Vol. 13 (1969), pp. 353–356.
- [Whaley98] Whaley, R.C., and J. Dongarra, "Automatically Tuned Linear Algebra Software," *Super Computing 1998 Conference*, Orlando, FL, November 1998.
- [Winograd68] Winograd, S., "A New Algorithm for Inner Product," *IEEE Trans. Computers*, C-17:693–694, 1968.