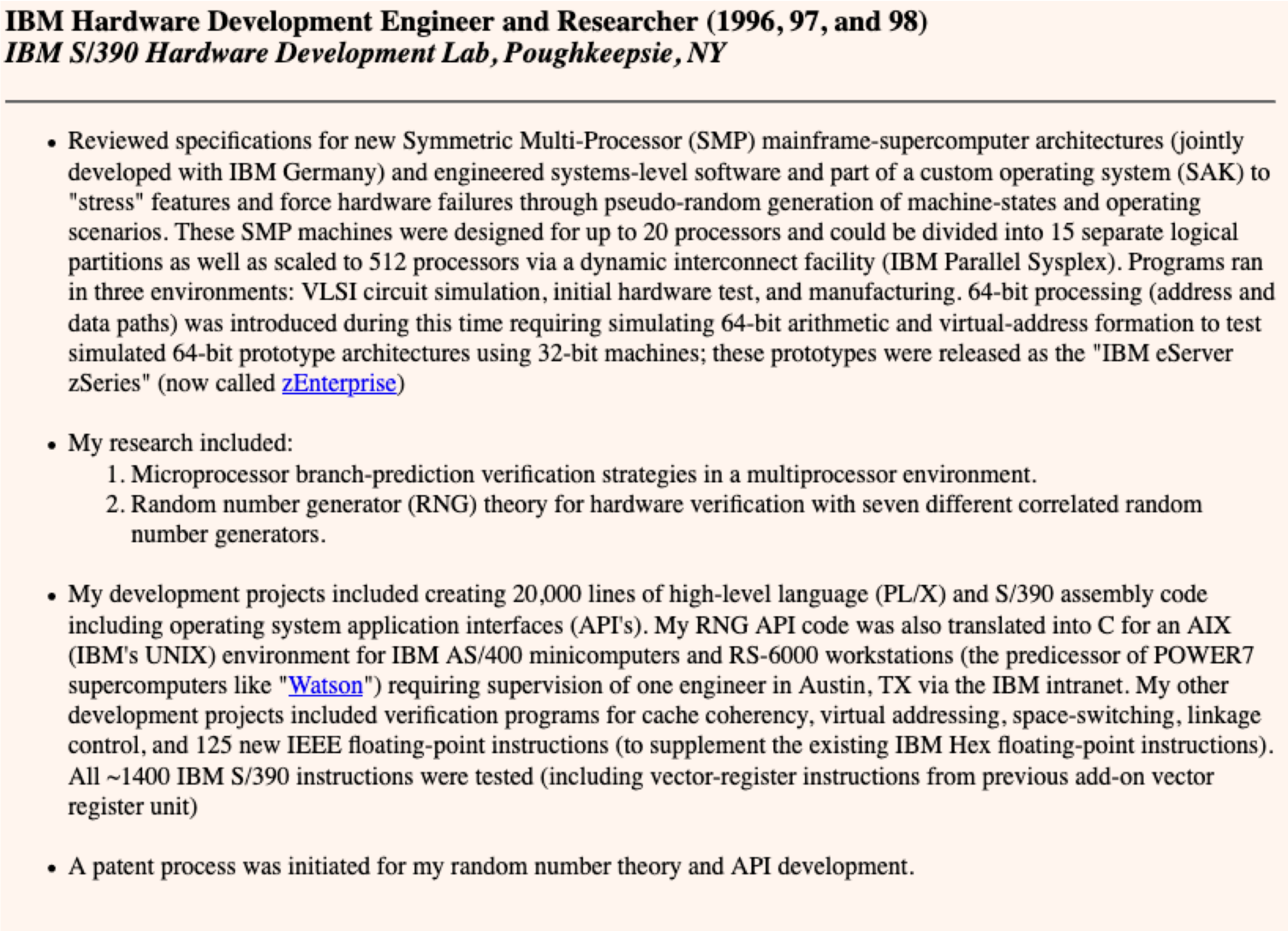
“PROCESSORS” and multi-processors

Excerpt from Hennessey Computer Architecture book; edits by JT Wunderlich PhD

Plus Dr W’s IBM Research & Development:

JT Wunderlich PhD



IBM Hardware Development Engineer and Researcher (1996, 97, and 98)
IBM S/390 Hardware Development Lab, Poughkeepsie, NY

- Reviewed specifications for new Symmetric Multi-Processor (SMP) mainframe-supercomputer architectures (jointly developed with IBM Germany) and engineered systems-level software and part of a custom operating system (SAK) to "stress" features and force hardware failures through pseudo-random generation of machine-states and operating scenarios. These SMP machines were designed for up to 20 processors and could be divided into 15 separate logical partitions as well as scaled to 512 processors via a dynamic interconnect facility (IBM Parallel Sysplex). Programs ran in three environments: VLSI circuit simulation, initial hardware test, and manufacturing. 64-bit processing (address and data paths) was introduced during this time requiring simulating 64-bit arithmetic and virtual-address formation to test simulated 64-bit prototype architectures using 32-bit machines; these prototypes were released as the "IBM eServer zSeries" (now called zEnterprise)

- My research included:
  1. Microprocessor branch-prediction verification strategies in a multiprocessor environment.
  2. Random number generator (RNG) theory for hardware verification with seven different correlated random number generators.

- My development projects included creating 20,000 lines of high-level language (PL/X) and S/390 assembly code including operating system application interfaces (API's). My RNG API code was also translated into C for an AIX (IBM's UNIX) environment for IBM AS/400 minicomputers and RS-6000 workstations (the predicessor of POWER7 supercomputers like "Watson") requiring supervision of one engineer in Austin, TX via the IBM intranet. My other development projects included verification programs for cache coherency, virtual addressing, space-switching, linkage control, and 125 new IEEE floating-point instructions (to supplement the existing IBM Hex floating-point instructions). All ~1400 IBM S/390 instructions were tested (including vector-register instructions from previous add-on vector register unit)

- A patent process was initiated for my random number theory and API development.

IBM S/390 Multiprocessor Hardware at the beginning of Dr. Wunderlich's IBM research
(IBM internal code names "Concerto" and "Monet"):

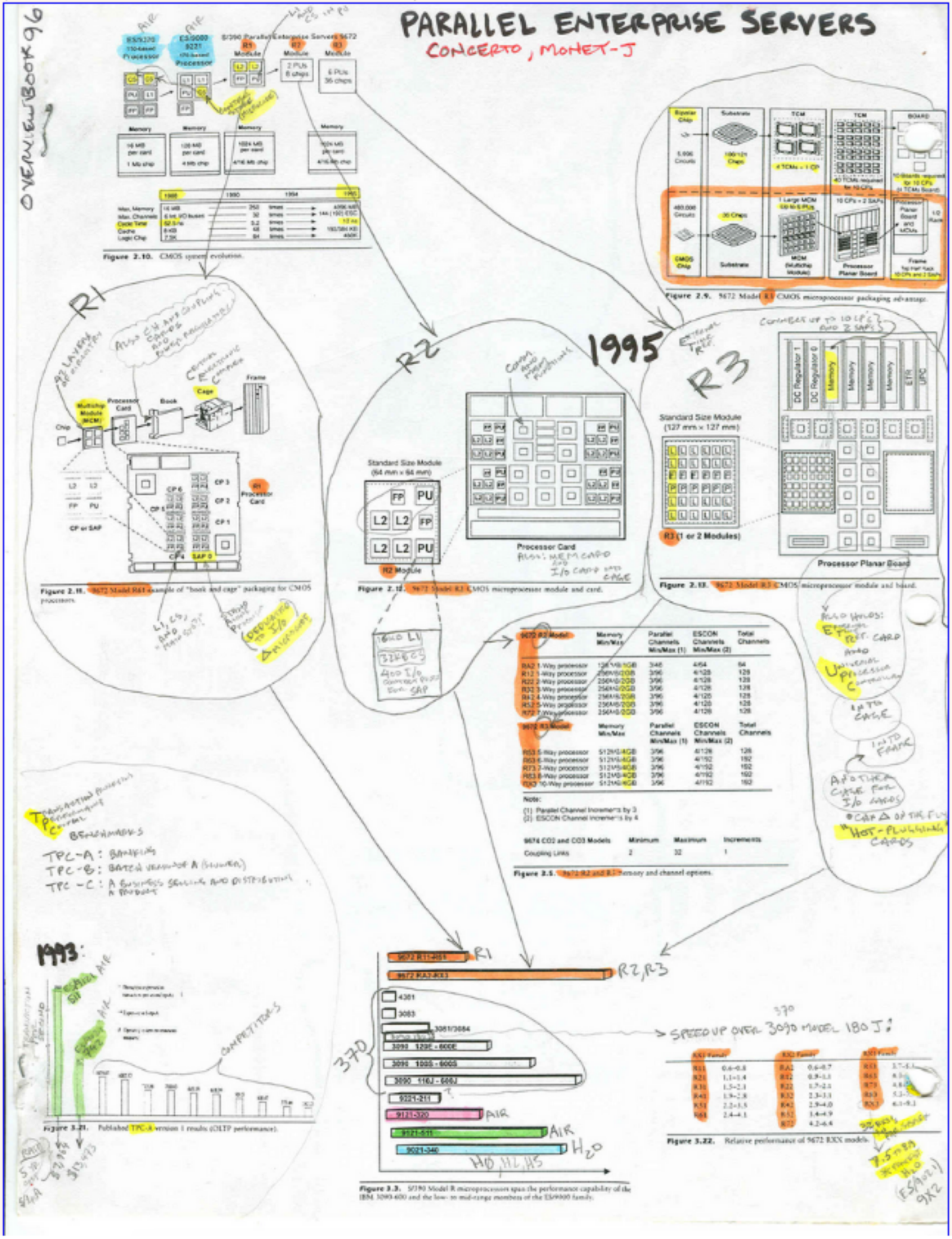

OVERVIEW BOOK 96
PARALLEL ENTERPRISE SERVERS
CONCERTO, MONET-J
RS/6000 110-based Processor
ES/9000 9221 170-based Processor
S/390 Parallel Enterprise Servers 9672
R1 Module
R2 Module
R3 Module
2 PUs 8 chips
6 PUs 36 chips
AIR
L1 AND CS IN PU
Memory
16 MB per card 1 Mb chip
128 MB per card 4 Mb chip
1024 MB per card 4/16 Mb chip
1024 MB per card 4/16 Mb chip
1988
1990
1994
1995
Max. Memory
Max. Channels
Cycle Time
Cache
Logic Chip
16 MB
6 Int. I/O busses
62.5 ns
8 KB
7.5K
256 times
32 times
5.2 times
48 times
64 times
4096 MB
144 (192) ESC
12 ns
192/384 KB
450K
Bipolar Chip
Substrate
TCM
BOARD
5,000 Circuits
100/121 Chips
4 TCMs = 1 CP
40 TCMs required for 10 CPs
10 Boards required for 10 CPs (4 TCMs/Board)
450,000 Circuits
36 Chips
1 Large MCM (10 M-5 PUs)
10 CPs + 2 SAPs
Processor Planar Board and MCMs
I/O Rack
CMOS Chip
Substrate
MCM (Multichip Module)
Processor Planar Board
Frame 1st Half Rack 10 CPs and 2 SAPs
R1
PLUS C4 AND COUPLING CARDS AND POWER REGULATORS
92 LAYERS OF CERAMIC
CENTRAL ELECTRONIC COMPLEX
Multichip Module (MCM)
Processor Card
Book
Cage
Frame
Chip
L2 L2 FP PU
CP or SAP
CP 6 CP 5 CP 4 CP 3 CP 2 CP 1 SAP 0
R1 Processor Card
L1, C4, AND "MAIN CHIP"
STRIP ALONG ALL PANELS
DEDICATED TO I/O PROCESSING
R2
COMM AND MAIN FUNCTIONS
Standard Size Module (64 mm × 64 mm)
FP PU L2 L2 FP L2 L2 PU
R2 Module
Processor Card
PLUS: MEM CARD AND I/O CARD INTO CAGE
16KB L1
32KB C3
400 I/O COMMON PLUS FOR SAP
1995
EXTERNAL TIME REF.
R3
CONNECT UP TO 10 CP'S AND 2 SAPS
Standard Size Module (127 mm × 127 mm)
R3 (1 or 2 Modules)
DC Regulator 1
DC Regulator 0
Memory
ETR
UPC
Processor Planar Board
ALSO HOLDS: EXT. TIME REF. CARD AND UNIVERSAL PROCESSOR CONTROLLER
INTO CAGE
INTO FRAME
ANOTHER CAGE FOR I/O CARDS
CAN Δ ON THE FLY "HOT-PLUGGING" CARDS
TRANSACTION PROCESSING COUNCIL
TPC BENCHMARKS
TPC-A: BANKING
TPC-B: BATCH VERSION OF A
TPC-C: A BUSINESS SELLING AND DISTRIBUTING A PRODUCT
1993:
ES/9000 511 AIR
S/390 9672 AIR
TRANSACTIONS PER SECOND
COMPETITORS
R1
R2, R3
9672 R11-R61
9672 RA2-RX3
4381
3083
3081/3084
3090 120E-600E
3090 100S-600S
3090 110J-600J
9221-211
9121-320
9121-511
9021-340
AIR
AIR
H_2O
H0, H1, H5
370
370
SPEEDUP OVER 3090 MODEL 180 J!
SPEEDUP 38% SPEEDUP
7.5 TO 8.5 SPEEDUP H_2O
(ES/9021) 9X2
5/10 9/1/93
8/1/93

Figure 2.10. CMOS system evolution.

Figure 2.9. 9672 Model R3 CMOS microprocessor packaging advantage.

Figure 2.11. 9672 Model R61 example of "book and cage" packaging for CMOS processors.

Figure 2.12. 9672 Model R2 CMOS microprocessor module and card.

Figure 2.13. 9672 Model R3 CMOS microprocessor module and board.

9672 R2 Model	Memory Min/Max	Parallel Channels Min/Max (1)	ESCON Channels Min/Max (2)	Total Channels
RA2 1-Way processor	128MB/1GB	3/48	4/64	64
R12 1-Way processor	256MB/2GB	3/96	4/128	128
R22 2-Way processor	256MB/2GB	3/96	4/128	128
R32 3-Way processor	256MB/2GB	3/96	4/128	128
R42 4-Way processor	256MB/2GB	3/96	4/128	128
R52 5-Way processor	256MB/2GB	3/96	4/128	128
R72 7-Way processor	256MB/2GB	3/96	4/128	128

9672 R3 Model	Memory Min/Max	Parallel Channels Min/Max (1)	ESCON Channels Min/Max (2)	Total Channels
R53 5-Way processor	512MB/4GB	3/96	4/128	128
R63 6-Way processor	512MB/4GB	3/96	4/192	192
R73 7-Way processor	512MB/4GB	3/96	4/192	192
R83 8-Way processor	512MB/4GB	3/96	4/192	192
RX3 10-Way processor	512MB/4GB	3/96	4/192	192

Note:
(1) Parallel Channel Increments by 3
(2) ESCON Channel Increments by 4

9674 C02 and C03 Models	Minimum	Maximum	Increments
Coupling Links	2	32	1

Figure 3.5. 9672 R2 and R3 memory and channel options.

Figure 3.21. Published TPC-A version 1 results (OLTP performance).

Figure 3.3. S/390 Model R microprocessors span the performance capability of the IBM 3090-600 and the low- to mid-range members of the ES/9000 family.

RX1 Family		RX2 Family		RX3 Family	
R11	0.6–0.8	RA2	0.6–0.7	R53	3.7–4.1
R21	1.1–1.4	R12	0.9–1.1	R63	4.3–[illegible]
R31	1.5–2.1	R22	1.7–2.1	R73	4.8–[illegible]
R41	1.9–2.8	R32	2.3–3.1	R83	5.3–[illegible]
R51	2.2–3.5	R42	2.9–4.0	RX3	6.1–9.1
R61	2.4–4.1	R52	3.4–4.9		
		R72	4.2–6.4		

Figure 3.22. Relative performance of 9672 RXX models.

“PROCESSORS”

Excerpt from Hennessey Computer Architecture book; edits by JT Wunderlich PhD

7.14 Historical Perspective and Further Reading

There is a tremendous amount of history in multiprocessors; in this section we divide our discussion by both time period and architecture. We start with the SIMD approach and the Illiac IV. We then turn to a short discussion of some other early experimental multiprocessors and progress to a discussion of some of the great debates in parallel processing. Next we discuss the historical roots of the present multiprocessors and conclude by discussing recent advances.

SIMD=Single Instruction, Multiple Data

SIMD Computers: Attractive Idea, Many Attempts, No Lasting Successes

> *The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist of recentralizing one of the three major components. . . . Centralizing the [control unit] gives rise to the basic organization of [an] . . . array processor such as the Illiac IV.*
>
> Bouknight, et al. [1972]

The SIMD model was one of the earliest models of parallel computing, dating back to the first large-scale multiprocessor, the Illiac IV. The key idea in that multiprocessor, as in more recent SIMD multiprocessors, is to have a single instruction that operates on many data items at once, using many functional units (see Figure 7.14.1).

Although successful in pushing several technologies that proved useful in later projects, it failed as a computer. Costs escalated from the $8 million estimate in 1966 to $31 million by 1972, despite construction of only a quarter of the planned multiprocessor. Actual performance was at best 15 MFLOPS, versus initial predictions of 1000 MFLOPS for the full system [Hord, 1982]. Delivered to NASA Ames Research in 1972, the computer required three more years of engineering before it was usable.

These events slowed investigation of SIMD, with Danny Hillis [1985] resuscitating this style in the Connection Machine, which had 65,636 1-bit processors.

Real SIMD computers need to have a mixture of SISD and SIMD instructions. There is an SISD host computer to perform operations such as branches and address calculations that do not need parallel operation. The SIMD instructions are broadcast to all the execution units, each of which has its own set of registers. For flexibility, individual execution units can be disabled during an SIMD instruction. In addition, massively parallel SIMD multiprocessors rely on interconnection or communication networks to exchange data between processing elements.

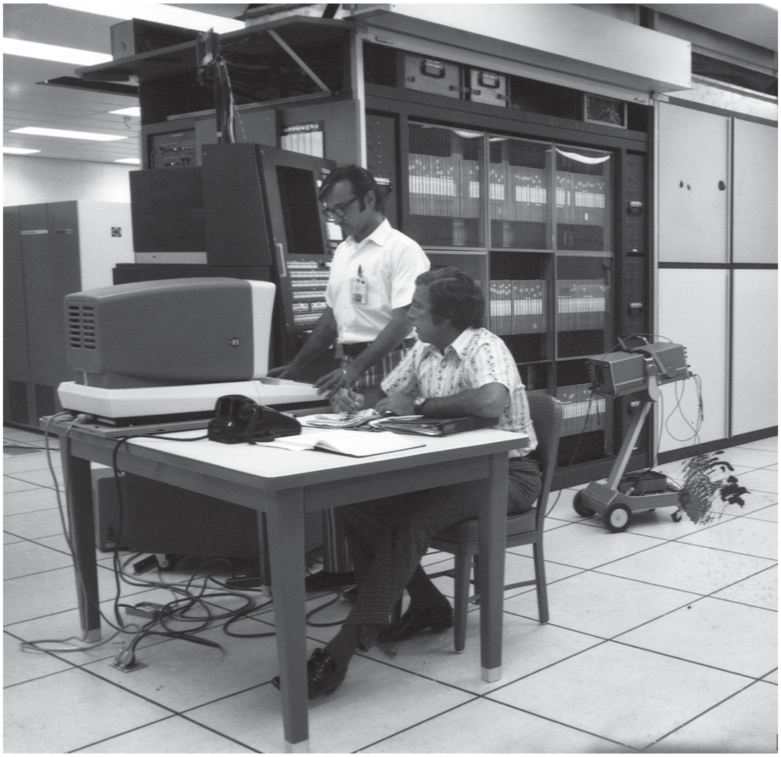


FIGURE 7.14.1 The Illiac IV control unit followed by its 64 processing elements. It was perhaps the most infamous of supercomputers. The project started in 1965 and ran its first real application in 1976. The 64 processors used a 13-MHz clock, and their combined main memory size was 1 MB: 64 × 16 KB. The Illiac IV was the first machine to teach us that software for parallel machines dominates hardware issues. Photo courtesy of NASA Ames Research Center.

SIMD works best in dealing with arrays in for loops. Hence, to have the opportunity for massive parallelism in SIMD, there must be massive amounts of data, or data parallelism. SIMD is at its weakest in case statements, in which each execution unit must perform a different operation on its data, depending on what data it has. The execution units with the wrong data are disabled so that the proper units can continue. Such situations essentially run at 1/*n*th performance, where *n* is the number of cases.

The basic tradeoff in SIMD multiprocessors is performance of a processor versus number of processors. Recent multiprocessors emphasize a large degree of parallelism over performance of the individual processors. The Connection Multiprocessor 2, for example, offered 65,536 single-bit-wide processors, while the Illiac IV had 64 64-bit processors.

After being resurrected in the 1980s, first by Thinking Machines and then by MasPar, the SIMD model has once again been put to bed as a general-purpose multiprocessor architecture, for two main reasons. First, it is too inflexible. A number of important problems cannot use such a style of multiprocessor, and the architecture does not scale down in a competitive fashion; that is, small-scale SIMD multiprocessors often have worse cost performance than that of the alternatives. Second, SIMD cannot take advantage of the tremendous performance and cost advantages of microprocessor technology. Instead of leveraging this low-cost technology, designers of SIMD multiprocessors must build custom processors for their multiprocessors.

Although SIMD computers have departed from the scene as general-purpose alternatives, this style of architecture will continue to have a role in special-purpose designs. Many special-purpose tasks are highly data parallel and require a limited set of functional units. Thus, designers can build in support for certain operations, as well as hardwired interconnection paths among functional units. Such organizations are often called array processors, and they are useful for tasks like image and signal processing.

Multimedia Extensions as SIMD Extensions to Instruction Sets

Many recent architectures have laid claim to being the first to offer multimedia extensions, in which a set of new instructions takes advantage of a single wide ALU that can be partitioned so that it will act as several narrower ALUs operating in parallel. It's unlikely that any appeared before 1957, however, when the Lincoln Lab's TX-2 computer offered instructions that operated on the ALU as either one 36-bit operation, two 18-bit operations, or four 9-bit operations. Ivan Sutherland, considered the Father of Computer Graphics, built his historic Sketchpad system on the TX-2. Sketchpad did in fact take advantage of these SIMD instructions, despite TX-2 appearing before invention of the term SIMD.

Other Early Experiments

It is difficult to distinguish the first MIMD multiprocessor. Surprisingly, the first computer from the Eckert-Mauchly Corporation, for example, had duplicate units to improve availability.

MIMD=Multiple Instruction, Multiple Data

Two of the best-documented multiprocessor projects were undertaken in the 1970s at Carnegie Mellon University. The first of these was C.mmp, which consisted of 16 PDP-11s connected by a crossbar switch to 16 memory units. It was among the first multiprocessors with more than a few processors, and it had a shared memory programming model. Much of the focus of the research in the C.mmp project was on software, especially in the OS area. A later multiprocessor, Cm*, was a cluster-based multiprocessor with a distributed memory and a nonuniform access time. The absence of caches and a long remote access latency made data placement critical. Many of the ideas in these multiprocessors would be reused in the 1980s, when the microprocessor made it much cheaper to build multiprocessors.

Great Debates in Parallel Processing

> *The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. . . . Electronic circuits are ultimately limited in their speed of operation by the speed of light . . . and many of the circuits were already operating in the nanosecond range.*
>
> W. Jack Bouknight, et al.
> The Illiac IV System [1972]

> *. . . sequential computers are approaching a fundamental physical limit on their potential computational power. Such a limit is the speed of light . . .*
>
> Angel L. DeCegama
> The Technology of Parallel Processing, Volume I [1989]

> *. . . today's multiprocessors . . . are nearing an impasse as technologies approach the speed of light. Even if the components of a sequential processor could be made to work this fast, the best that could be expected is no more than a few million instructions per second.*
>
> David Mitchell
> The Transputer: The Time Is Now [1989]

The quotes above give the classic arguments for abandoning the current form of computing, and Amdahl [1967] gave the classic reply in support of continued focus on the IBM 360 architecture. Arguments for the advantages of parallel execution can be traced back to the 19th century [Menabrea, 1842]! Despite this, the effectiveness of the multiprocessor in reducing the latency of individual important programs is still being explored. Aside from these debates about the advantages and limitations of parallelism, several hot debates have focused on how to build multiprocessors.

From today's perspective, it is clear that the speed of light was not the brick wall; it was, instead, the power consumption of CMOS as the clock rates increased.

It's hard to predict the future, yet in 1989 Gordon Bell made two predictions for 1995. We included these predictions in the first edition of the book, when the outcome was completely unclear. We discuss them in this section, together with an assessment of the accuracy of the prediction.

The first was that a computer capable of sustaining a teraFLOPS—one million MFLOPS—would be constructed by 1995, using either a multicomputer with 4K to 32K nodes or a Connection Multiprocessor with several million processing elements [Bell, 1989]. To put this prediction in perspective, each year the Gordon Bell Prize acknowledges advances in parallelism, including the fastest real program (highest MFLOPS). In 1989 the winner used an eight-processor Cray Y-MP to run at 1680 MFLOPS. On the basis of these numbers, multiprocessors and programs would have to have improved by a factor of 3.6 each year for the fastest program to achieve 1 TFLOPS in 1995. In 1999, the first Gordon Bell prize winner crossed the 1 TFLOPS bar. Using a 5832-processor IBM RS/6000 SST system designed specially for Livermore Laboratories, they achieved 1.18 TFLOPS on a shock wave simulation. This ratio represents a year-to-year improvement of 1.93, which is still quite impressive.

What has been recognized since the 1990s is that although we may have the technology to build a TFLOPS multiprocessor, it is not clear that the machine is cost effective, except perhaps for a few very specialized and critically important applications related to national security. We estimated in 1990 that achieving 1 TFLOPS would require a machine with about 5000 processors and would cost about $100 million. The 5832-processor IBM system at Livermore cost $110 million. As might be expected, improvements in the performance of individual microprocessors both in cost and performance directly affect the cost and performance of large-scale multiprocessors, but a 5000-processor system will cost more than 5000 times the price of a desktop system using the same processor. Since that time, much faster multiprocessors have been built, but the major improvements have increasingly come from the processors in the past five years, rather than fundamental breakthroughs in parallel architecture.

The second Bell prediction concerned the number of data streams in supercomputers shipped in 1995. Danny Hillis believed that although supercomputers with a small number of data streams might be the best sellers, the biggest multiprocessors would be multiprocessors with many data streams, and these would perform the bulk of the computations. Bell bet Hillis that in the last quarter of calendar year 1995, more sustained MFLOPS would be shipped in multiprocessors using few data streams (<100) rather than many data streams (>1000). This bet concerned only supercomputers, defined as multiprocessors costing more than $1 million and used for scientific applications. Sustained MFLOPS was defined for this bet as the number of floating-point operations per month, so availability of multiprocessors affects their rating.

In 1989, when this bet was made, it was totally unclear who would win. In 1995, a survey of the current publicly known supercomputers showed only six multiprocessors in existence in the world with more than 1000 data streams, so Bell's prediction was a clear winner. In fact, in 1995, much smaller microprocessor-based multiprocessors (<20 processors) were becoming dominant.

In 1995, a survey of the 500 highest-performance multiprocessors in use (based on Linpack ratings), called the Top 500, showed that the largest number of multiprocessors were bus-based shared memory multiprocessors! By 2005, various clusters or multicomputers played a large role. For example, in the top 25 systems, 11 were custom clusters, such as the IBM Blue Gene system or the Cray XT3, 10 were clusters of shared memory multiprocessors (both using distributed and centralized memory), and the remaining 4 were clusters built using PCs with an off-the-shelf interconnect.

More Recent Advances and Developments

With the primary exception of the parallel vector multiprocessors and more recently of the IBM Blue Gene design, all other recent MIMD computers have been built from off-the-shelf microprocessors using a bus and logically central memory or an interconnection network and a distributed memory. A number of experimental multiprocessors built in the 1980s further refined and enhanced the concepts that form the basis for many of today's multiprocessors.

The Development of Bus-Based Coherent Multiprocessors

Although very large mainframes were built with multiple processors in the 1960s and 1970s, multiprocessors did not become highly successful until the 1980s. Bell [1985] suggests the key was that the smaller size of the microprocessor allowed the memory bus to replace the interconnection network hardware and that portable operating systems meant that multiprocessor projects no longer required the invention of a new operating system. In this paper, Bell defined the terms multiprocessor and multicomputer and set the stage for two different approaches to building larger-scale multiprocessors. The first bus-based multiprocessor with snooping caches was the Synapse N + 1 in 1984.

The early 1990s saw the beginning of an expansion of such systems with the use of very wide, high-speed buses (the SGI Challenge system used a 256-bit, packet-oriented bus supporting up to 8 processor boards and 32 processors) and later the use of multiple buses and crossbar interconnects, for example, in the Sun SPARCCenter and Enterprise systems. In 2001, the Sun Enterprise servers represented the primary example of large-scale (>16 processors), symmetric multiprocessors in active use.

Toward Large-Scale Multiprocessors

In the effort to build large-scale multiprocessors, two different directions were explored: message-passing multicomputers and scalable shared memory multiprocessors. Although there had been many attempts to build mesh and hypercube-connected multiprocessors, one of the first multiprocessors to successfully bring together all the pieces was the Cosmic Cube built at Caltech [Seitz, 1985]. It introduced important advances in routing and interconnect technology and substantially reduced the cost of the interconnect, which helped make the multicomputer viable. The Intel iPSC 860, a hypercube-connected collection of i860s, was based on these ideas. More recent multiprocessors, such as the Intel Paragon, have used networks with lower dimensionality and higher indi- vidual links. The Paragon also employed a separate i860 as a communications controller in each node, although a number of users have found it better to use both i860 processors for computation as well as communication. The Thinking Multiprocessors CM-5 made use of off-the-shelf microprocessors. It provided user-level access to the communication channel, significantly improving communication latency. In 1995, these two multiprocessors represented the state of the art in message-passing multicomputers.

Clusters

Clusters were probably "invented" in the 1960s by customers who could not fit all their work on one computer, or who needed a backup machine in case of failure of the primary machine [Pfister, 1998]. Tandem introduced a 16-node cluster in 1975. Digital followed with VAXclusters, introduced in 1984. They were originally independent computers that shared I/O devices, requiring a distributed operating system to coordinate activity. Soon they had communication links between computers, in part so that the computers could be geographically distributed to increase availability in case of a disaster at a single site. Users log onto the cluster and are unaware of which machine they are using. DEC (now HP) sold more than 25,000 clusters by 1993. Other early companies were Tandem (now HP) and IBM (still IBM). Today, virtually every company has cluster products. Most of these products are aimed at availability, with performance scaling as a secondary benefit.

Scientific computing on clusters emerged as a competitor to MPPs. In 1993, the Beowulf project started with the goal of fulfilling NASA's desire for a 1-GFLOPS computer for less than $50,000. In 1994, a 16-node cluster built from off-the-shelf PCs using 80486s achieved that goal. This emphasis led to a variety of software interfaces to make it easier to submit, coordinate, and debug large programs or a large number of independent programs.

Efforts were made to reduce latency of communication in clusters as well as to increase bandwidth, and several research projects worked on that problem. (One commercial result of the low-latency research was the VI interface standard, which has been embraced by Infiniband, discussed below.) Low latency then proved useful in other applications. For example, in 1997 a cluster of 100 UltraSPARC desktop computers at U.C. Berkeley, connected by 160 MB/sec per link Myrinet switches, was used to set world records in database sort (sorting 8.6 GB of data originally on disk in 1 minute) and in cracking an encrypted message (taking just 3.5 hours to decipher a 40-bit DESkey).

This research project, called Network of Workstations, also developed the Inktomi search engine, which led to a start-up company with the same name.

Google followed the example of Inktomi to build search engines from clusters of desktop computers rather than large-scale SMPs, which was the strategy of the leading search engine, Alta Vista, that Google overtook. In 2008, nearly all Internet services rely on clusters to serve their millions of customers.

Clusters are also very popular with scientists. One reason is their low cost, which enables individual scientists or small groups to own a cluster dedicated to their programs. Such clusters can get results faster than waiting in the long job queues of the shared MPPs at supercomputer centers, which can stretch to weeks.

For those interested in learning more, Pfister [1998] has written an entertaining book on clusters.

Recent Trends in Large-Scale Multiprocessors

In the mid-to-late 1990s, it became clear that the hoped-for growth in the market for ultralarge-scale parallel computing was unlikely to occur. Without this market growth, it became increasingly clear that the high-end parallel computing market was too small to support the costs of highly customized hardware and software designed for a small market. Perhaps the most important trend to come out of this observation was that clustering would be used to reach the highest levels of performance. There are now four general classes of large-scale multiprocessors:

1. Clusters that integrate standard desktop motherboards using interconnection technology, such as Myrinet or Infiniban
2. Multicomputers built from standard microprocessors configured into processing elements and connected with a custom interconnect, such as the IBM Blue Gene
3. Clusters of small-scale shared memory computers, possibly with vector support, including the Earth Simulator

The IBM Blue Gene is the most interesting of these designs, since its rationale parallels the underlying causes of the recent trend towards multicore in uniprocessor architectures. Blue Gene started as a research project within IBM aimed at the protein sequencing and folding problem. The Blue Gene designers observed that power was becoming an increasing concern in large-scale multiprocessors and that the performance/watt of processors from the embedded space was much better than those in the high-end uniprocessor space. If parallelism was the route to high performance, why not start with the most efficient building block and simply have more of them?

Thus, Blue Gene is constructed using a custom chip that includes an embedded PowerPC microprocessor offering half the performance of a high-end PowerPC, but at a much smaller fraction of the area and the power. This allows more system functions, including the global interconnect, to be integrated onto the same die.

The result is a highly replicable and efficient building block, allowing Blue Gene to reach much larger processor counts more efficiently. Instead of using stand-alone microprocessors or standard desktop boards as building blocks, Blue Gene uses processor cores. There is no doubt that such an approach provides much greater efficiency. Whether the market can support the cost of a customized design and special software remains an open question.

In 2006, a Blue Gene processor at Lawrence Livermore with 32K processors held a factor of 2.6 lead in Linpack performance over the third-place system, which consisted of 20 SGI Altix 512-processor systems interconnected with Infiniband as a cluster.

Blue Gene's predecessor was an experimental machine, QCDOD, which pioneered the concept of a machine using a lower-power embedded microprocessor and tightly integrated interconnect to drive down the cost and power consumption of a node.

Looking Further

There is an almost unbounded amount of information on multiprocessors and multicomputers: conferences, journal papers, and even books seem to appear faster than any single person can absorb the ideas. No doubt many of these papers will go unnoticed—not unlike the past. Most of the major architecture conferences contain papers on multiprocessors. An annual conference, Supercomputing XY (where X and Y are the last two digits of the year), brings together users, architects, software developers, and vendors and publishes the proceedings in book, CD-ROM, and online (see www.scXY.org) form. Two major journals, *Journal of Parallel and Distributed Computing* and the *IEEE Transactions on Parallel and Distributed Systems*, contain papers on all aspects of parallel processing. Several books focusing on parallel processing are included in the following references, with Culler, Singh, and Gupta [1999] being the most recent, large-scale effort. For years, Eugene Miya of NASA Ames has collected an online bibliography of parallel processing papers. The bibliography, which now contains more than 35,000 entries, is available online at liinwww.ira.uka.de/bibliography/Parallel/Eugene/index.html.

Asanovic, et al. [2006] recently surveyed the wide-ranging challenges for the industry in this multicore challenge. That report may be a helpful in understanding the depth of the various challenges.

In addition to documenting the discovery of concepts now used in practice, these references also provide descriptions of many ideas that have been explored and found wanting, as well as ideas whose time has just not yet come. Given the move toward multicore and multiprocessors as the future of high-performance computer architecture, we expect that many new approaches will be explored in the years ahead. A few of them will manage to solve the hardware and software problems that have been the key to using multiprocessing for the past 40 years!

List of Intel microprocessors

From https://en.wikipedia.org/wiki/List_of_Intel_microprocessors

64 bit processors: Intel 64 – Skylake microarchitecture

Core i3 (6th Generation)

- **Skylake** (Core i3 6th Generation) – 14 nm process technology
    - 2 physical cores/4 threads
    - 3–4 MB L3 cache
    - Introduced Q3'15
    - Socket 1151 LGA
    - 2-channel DDR3L-1333/1600, DDR4-1866/2133
    - Integrated GPU Intel HD Graphics 530 (only i3-6098P have HD Graphics 510)
    - Variants
        - i3-6098P – 3.60 GHz
        - i3-6100T – 3.20 GHz
        - i3-6100 – 3.70 GHz
        - i3-6300T – 3.30 GHz
        - i3-6300 – 3.80 GHz
        - i3-6320 – 3.90 GHz

Core i5 (6th Generation)

- **Skylake** (Core i5 6th Generation) – 14nm process technology
    - 4 physical cores/4 threads
    - 6 MB L3 cache
    - Introduced Q3'15
    - Socket 1151 LGA
    - 2-channel DDR3L-1333/1600, DDR4-1866/2133
    - Integrated GPU Intel HD Graphics 530
    - Variants
        - i5-6400T – 2.20 GHz/2.80 GHz Turbo Boost
        - i5-6400 – 2.70 GHz/3.30 GHz Turbo Boost
        - i5-6500T – 2.50 GHz/3.10 GHz Turbo Boost
        - i5-6500 – 3.20 GHz/3.60 GHz Turbo Boost
        - i5-6600T – 2.70 GHz/3.50 GHz Turbo Boost
        - i5-6600 – 3.30 GHz/3.90 GHz Turbo Boost
        - i5-6600K – 3.50 GHz/3.90 GHz Turbo Boost

Core i7 (6th Generation)

- **Skylake** (Core i7 6th Generation) – 14nm process technology
    - 4 physical cores/8 threads
    - 8 MB L3 cache
    - Introduced Q3'15
    - Socket 1151 LGA
    - 2-channel DDR3L-1333/1600, DDR4-1866/2133
    - Integrated GPU Intel HD Graphics 530
    - Variants
        - i7-6700T – 2.80 GHz/3.60 GHz Turbo Boost
        - i7-6700 – 3.40 GHz/4.00 GHz Turbo Boost
        - i7-6700K – 4.00 GHz/4.20 GHz Turbo Boost

Latest desktop and mobile processors for consumers [edit J

9th genera tion Core/Coff ee Lake [edit]

Model •	Pr ice (USD) •	c ores /Th r eads •	Base freque ncy (GH z)•	Max turb o f reque n cy (GHI)•	GPU ●	Maxi mumGPUcl ock rate (MHz.) •	L3cache(MB) •	TDP (W) ♦	soc k et •	Re l ease •
i9-9900K	$529	8/16	3.6	5.0	UHD 630	1200	16	95	LGA 1151	04 2018
i7-9700K	$399	818	3.7	4.9	UHD 630	1200	12	95	LGA 1151	0 42018
i5-9600K	$229	616	3.7	4.6	UHD630	1150	9	95	LGA 1151	04 2018

8th generation Core/Coffee Lake/Kaby Lake Refresh [edit]

Des kt op l edit l

M ode l ●	Price (USC) ♦	Cores / Threads +	Base freque ncy (GH z)+	Max t urbo freque n cy (GH z)+	GPU ♦	Max imum GPU cl ock r at e (MHz.) +	L3cache (MB) +	TDP (W) ♦	Socket ♦	Release +
i7-8086K@	$425	6/12	4.0	5.0	UHD 630	1200	12	95	LGA1151	02 2018
i7-8700K@	$359	6/12	3.7	4.7	UHD 630	1200	12	95	LGA1151	Q4 2017
17-8700@	$3{)3	6/12	3.2	4.6	UHD 630	1200	12	65	LGA1151	Q4 2017
15-860 0K@	$257	616	3.6	4.3	UHD 630	1150	9	95	LGA1151	Q4 2017
iS-8500@	$202	6/6	3.0	4.1	UHD 630	1100	9	65	LGA1151	02 2018
i5-8400	$182	6/6	2.8	4.0	UHD 630	105()	9	65	LGA1151	0 4 2017
i3-8350K @	$168	414	4.0	NIA	UHD 630	1150	8	91	LGA1151	Q4 2017
13-810 0@	$117	414	3.6	NIA	UHD 630	1100	6	65	LGA1151	Q4 2017

Mo bil e [edit]

Mo del ■ +	Pr ice (US□)	Cores/ Thr eads +	Basefrequency(GHz) •	Ma x tu rb o freq u ency (GHz) •	GPU	Max imum GPU clock rate (MHz) +	L3cache (MB) •	TDP (W) ♦	Rel ease •
i7-8650U@	$409	418	19	42	UHD 620	115()	8	15	03 2017
i7-85 50U	$409	418	18	4 0	UHD 620	115()	8	15	032017
i5-8350U	5297	418	u	3.6	UHD 620	1100	6	15	0 3 2017
i5-8250U	$297	418	1 6	34	UHD 620	1100	6	15	0 3 2017

7th gene ratioin Cor,e JKaby Lak eJS.k ylake (X-series Proc es sor s)/Apo llo Lak e (.edit)

Desktop- (edi t)

Mo del '	Pri1=e-(USD) •	Gores/Thr eads •	8-asef re-qu enoy(GHz,)	Max turbofre-quency (GHz.,)	GPU •	lila.xim um GPU c l oc:k rate (MIH z) '	L3 cache(MB) '	TDP (WI o	Sooltet •·	Rele-ase,
B-7900:CE	S1933	1813/!	2.6	4.2	NIA	NIA	24.75	165	LGA 2000	00 201JC11
I9-100!))(,o.	:S163S	100 2	2.8	4.2	NIA	NIA	22.00	165	LGA :2000	00 20 11' 11
B-J940XJ....	S1.393	14128	3.1	4.3	NIA	NIA	19.25	165	LGA	OOZ0 1Ji:1l
I9-7S20)(...	S1189	1 4	2.9	4.3	NIA	NIA	16.M	140	LGA 2000	Q3 2017
i9-7900X J....	$999	10/20	3.3	4.3	NIA	NIA	13.15	140	LGA :2000	Q2 2(11]
i7-7320X ,_	$59,	11116	3.6	4.3	NIA	NIA	11.03	140	LGA 2,Cff,	Q2 2017
i7-7&00X ....,	$389	"'1 2	3.5	4.0	NIA	NIA	8.25	140	LGA :2000	02 2017
i7-n40X,_	$3..'-0	418	4.3	4.5	NIA	NIA	8	112	LGA	Q I 2017
i7-n OOK		418	4.2	4.5	HDo:30	11!1.l	8	91	LGA 11.51	QI 2017
u-no0l,	$>12	418	3.6	4.2	HDo:30	ii .00	8	M	LGA H-51	Q i 2017
i7-ITOOT.,	Sl 2	4/8	2.9	3.8	HDo:30	1 !'fl	8	35	LGA n.5	Oi 20 7
i5-7 G40X.L	S242	414	4.0	4.2	NIA	NIA	6	112	LGA 2,Ctx,	QI 201
i5-7&00K_,_	S24J.	414	3.8	4.2	HDo:30	11!1.l	●	91	LGA 11-51	Q I 2017
i5-]00[),1_;.	S224	414	3.5	4.1	HDo:30	n W	●	M	LGA 11-51	Q i 2017
i5-7600T J....	S224	414	2.8	3.1	HDo:30	1100	●	35	LGA 11-51	Q i 2017
i5- 7500.,_,_	S07	414	3.4	3.8	HDo:30	110J		M	LGA 11.51	QI 2017
i5-7500T ,..		414	2.1	3.3	HDo:30	U OJ	6	35	LGA U.51	Q i 20 7
iS-7400 ,_	$1	414	3.0	3.5	HDo:30	1000	6	M	LGA 1151	QI 201
i5-7400T	:Sl:87	414	2.4	3.0	HD630	1000	6	35	LGA 11.51	Q I 2017
il -7J. K ,_	:SH9	214	4.2	NIA	HDo:30	ii .50	4	00	LGA H-51	Qi 2U17
i3-7320:l.-	:Si-57	214	4. i	NIA	HDo:30	ii .10	4	51	LGA H-51	Q i 2017
il -7300.,_,	S147	214	4.0	NIA	HDo:30	11!1.l	4	51	LGA 11.51	QI 2017
i.l-73.00T -t..	$147	214	3.5	NIA	HD630	iiOJ	4	35	LGA 11.51	Q i 2'017
il -7100	Sl 7	214	3.9	NIA	HDo:30	1 03	3	51	LGA H.5	QI 20 7
i3,7100T	$ i17	214	3.4	NIA	HDo:30	i10 J	3	35	LGA 11-51	Q i 2017
i3-71'!HE	$ 117	214	3.9	NIA	HDo:30	110J	3	54	LGA 11-51	0 1 2017
i3-7101TE	:Sil 1	214	3.4	NIA	HDo:30	110J	3	35	LGA 11-51	QI 2017
G.4&20	$3:1	214	3.7	NIA	HDo:30	1100	3	51	LGA 11.51	Q i 2017
G4600<	$!!2	214	3.6	NIA	HD630	ll OJ	3	51	LGA 11.51	QI 2017
G46:>DT	$75	214	3.0	NIA	HDo:30	050	3	35	LGA 5	Q 2(1]
G.4560	W4	214	3.5	NIA	HD 6 10	1050	3	54	LGA H-51	Q i 2017
G4E6DT	W4	214	2.9	NIA	HD 6 10	1050	3	35	LGA 11.51	Q I 2017
G39...'-®•	$52	212	3.0	NIA	HD 610	10.'.-0	2	51	LGA 11-51	0 1 2017
	542	212	2.9	NIA	HD 6 10	10!1.l	2	51	LGA H-51	Qi 2011
GJ9:IOT	$42	212	2.7	NIA	HD610	1000	2	35	LGA 11.51	QI 2011

Model	Price (USD)	Cores/Threads	CPU clock rate (GHz)	CPU Turbo clock rate (GHz)	GPU	ttaximum GPU clock rate (MHz)	Cache (MB)	TOP (W)	Release
i7-7920HQ<-	S!&	418	3.1	4.1	HD830	1100	8	45	Q I 20\7
i7-7820HQ<-	S378	418	2.9	3.9	HD830	1100	8	45	Q I 20\7
i7-7820HK<-	S378	418	2.9	3.9	HD830	HOO	8	45	Q i 20i7
i7-7700HQ<-	S378	418	2.8	3.8	HD830	HOO	6	45	Q i 20i7
i7-7e60U<-	S4!5	2/4	2.5	2.5	Iris?hts640	HOO	4	15	Q| 20i7
i7-7600U<-	S393	2/4	2.8	2.8	H0620	tl!<O	4	15	03 2016
i7-7567U <-	N/A	2/4	3.5	3.5	Iris?lus 6:<>	tl!<)	4	28	03 2016
i7-7560U<-	S415	2/4	2.4	2.4	I ris?lus 640	10!<>	4	15	Q|20\7
i7-7500U<-	S393	2/4	2.7	3.5	HD820	\O!<)	4	15	Q i 20i7
i7-7Y75<-	S393	2/4	i.3	3.6	H0 6 i 5	\O!<)	4	4.5	Q i 20i7
i5-7440HQ<-	S2!<>	414	2.8	3.8	HD830	1 (XX)	6	45	Q i 20i7
i5-7300HQ<-	S2!<>	414	2.5	3.5	HD830	\(XX)	6	45	Q I 20\7
i5-7360U<-	S304	2/4	2.3	2.3	Iris?lus640	\(XX)	4	15	Q I 20\7
i5-7300U<-	S281	2/4	2.6	2.6	H0620	1100	3	15	Q I 20\7
i5-7287U<-	N/A	2/4	3.3	3.3	Iris?hts6:<)	HOO	4	28	Q| 20i7
i5-7267U<-	N/A	2/4	3. i	3.1	Iris?hts6:<)	\O!<)	4	28	Q i 20i7
i5-7260U<-	S304	2/4	2.2	2.2	Iris?hts640	st<>	4	15	Qi20i7
i5-7200U<-	S281	2/4	2.5	2.5	H0620	\(XX)	3	15	03 2016
i5-7Y57<-	S28 1	2/4	1.2	3.3	H0615	st<>	4	45	Q I 20\7
i5-7Y54<-	S28 1	2/4	1.2	3.2	H0615	st<>	4	45	Q I 20\7
il-7100H.e..	S225	2/4	3.0	N/A	HD830	st<>	3	35	Qi20i7
il-7167U<-	N/A	2/4	2.8	N/A	Iris?hts6:<)	\(XX)	3	28	Q i 20i7
il-7130U<-	N/A	2/4	2.7	N/A	HD820	\(XX)	3	15	Q2 20i7
il-7100U<-	S28 1	2/4	2.4	N/A	H0620	\(XX)	3	15	03 2016
ml-7Y32<-	S281	2/4	1.1	3.0	H0 615	900	4	4.5	Q2 20\7
ml-7Y30<-	S281	2/4	1.0	2.6	H0 615	900	4	4.5	03 2016
N4200<-	S16i	414	u	2.5	H0!<>5	700	2 (t2)	6	03 2016
4415U<-	S16i	2/4	2.3	N/A	H0 610	st<>	2	15	Qi20i7
4415Y<-	S16i	2/4	i.6	N/A	H0 6 i 5	8!<)	2	6	Q2 20 i 7
4410Y <-	S16I	2/4	1.5	N/A	H0 615	8!<)	2	6	Q I 20\7
N3450<-	S107	414	1.1	2.2	H0 !.00	6.:<)	2 (t2)	6	03 2016
N3350•	S107	2/2	1.1	2.4	H0 !.00	6.<)	2 (t2)	6	03 2016
3965U•	S107	2/2	2.2	N/A	H0 610	900	2	15	Qi 20i7
W55U•	S107	2/2	i.8	N/A	H0 610	900	2	15	Qi 20i7

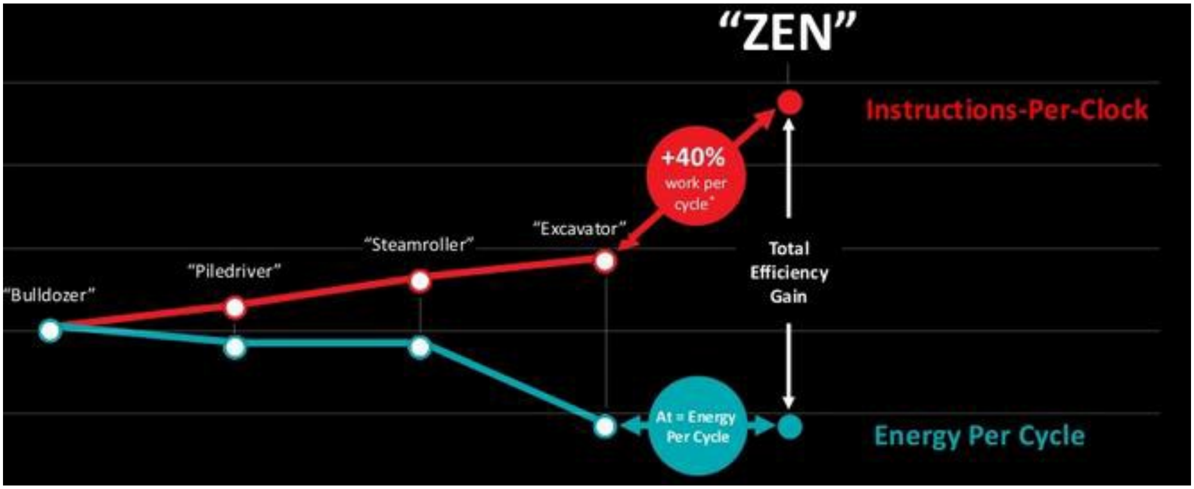
AMD "ZEN" Core

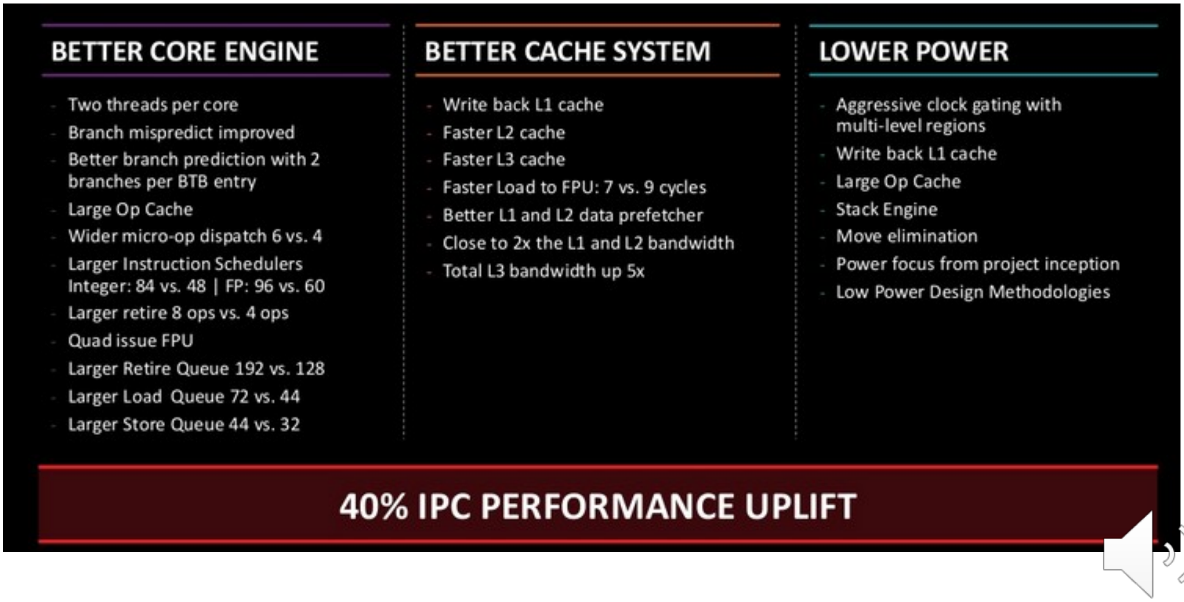
For "Wintel" Machines
(i.e., Windows Operating Systems, Intel x86-FAMILY of Processors)

NOTE: AMD is a competitor of Intel, but adheres to the Machine
Instruction Set of the Intel x86-Processor Family

J Wunderlich PhD 2018 Lecture Notes

SOURCE: http://www.amd.com/en-us/innovations/software-technologies/zen-cpu





BETTER CORE ENGINE

- Two threads per core
- Branch mispredict improved
- Better branch prediction with 2 branches per BTB entry
- Large Op Cache
- Wider micro-op dispatch 6 vs. 4
- Larger Instruction Schedulers Integer: 84 vs. 48 | FP: 96 vs. 60
- Larger retire 8 ops vs. 4 ops
- Quad issue FPU
- Larger Retire Queue 192 vs. 128
- Larger Load Queue 72 vs. 44
- Larger Store Queue 44 vs. 32

BETTER CACHE SYSTEM

- Write back L1 cache
- Faster L2 cache
- Faster L3 cache
- Faster Load to FPU: 7 vs. 9 cycles
- Better L1 and L2 data prefetcher
- Close to 2x the L1 and L2 bandwidth
- Total L3 bandwidth up 5x

LOWER POWER

- Aggressive clock gating with multi-level regions
- Write back L1 cache
- Large Op Cache
- Stack Engine
- Move elimination
- Power focus from project inception
- Low Power Design Methodologies

40% IPC PERFORMANCE UPLIFT

Pure Power

Gooil and quiet processor operation usin gmachine intelligence, sen.sors, and optimized cimuit desi,gin.

■ Monitors temperature, speed and voltage
■ Adaptive control manages reall time for lower power usage
■ Ongoing monitoring guidesother AMD SenseMI features

lP1recisionBoost

lFine-tuned processor performance adjusted in real time *to* rneet the clockspeed deman.ds of your game or app.

■ Works in tandem with Pure Power co:ntrol! loop to optimize performance
■ On-the-fily clock adj ustment without halts or qleue drains
■ High precision tuniing with 25MHz incr ements

ExtendedFrequencyRange

Alutomatic extra perfor man ce boost forenthusiasts wiith premium systems an,d process,or cooling.

■ Permits frequencie.s above and beyond ordinary Prec1sion Boost limits
■ crockspeed scares with cooling solution: air, water, and LN2
■ Fullyautomated; no user intervention r equired

NEURAL NETWORK PREDICTIION

Buillt-in artificial intelligence that primes your processor to tackle your app workload more efficiently.

1. **A true artificial network inside every "'Zen" processor**
2. **Buildss a model of the decisions driven by software code execution**
3. **Anticipates future decisions, pre-load instructions, choose the best path through the CPU**

Smart Prefetch

Leam ing algorithms that predict and pre-road needed data fo:r fast and responsive computing.

■ Ainticipates the rocation of future data, aocesses by appllication code
■ Sophisticated learn ing algorithms model and leam app1icati on data ac,ess. pattems
■ Preifetches vital dat a in o loca,lcache so it 's ready for immediate use

JT Wunderlich PhD in the early 1990's

Masters of Engineering, Engineering Science, Pennsylvania State University, Great Valley 1992

Track: Computer Hardware Design

Thesis: ***"A vector-register neural-network microprocessor design with on-chip learning" (filed Patent Disclosure Document)***

Advisor: Dr. Chung Ho Chen (Unisys R&D, EE and CompEng Professor)

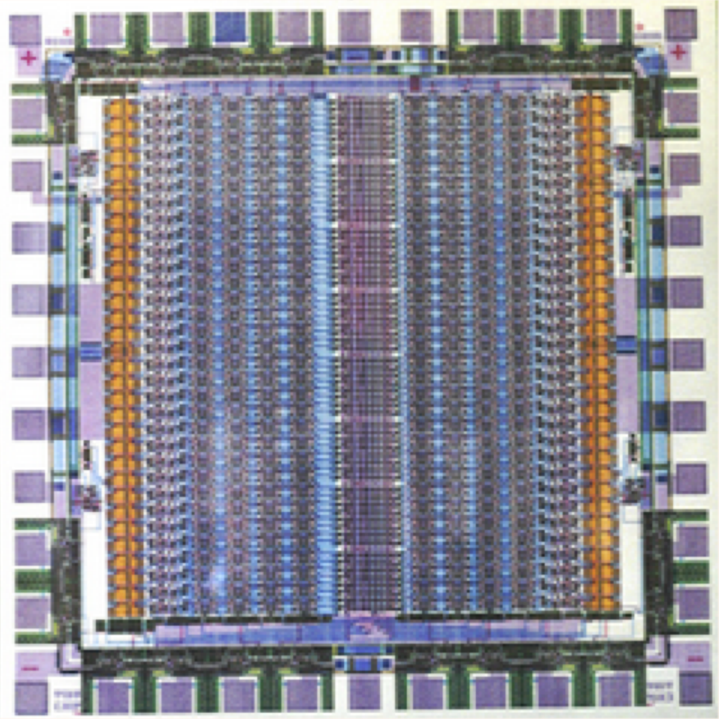
PhD Electrical Engineering, University of Delaware 1996

Research Group: **Computer Engineering**

Dissertation: ***"Optimal kinematic design of redundant and hyper-redundant manipulators for constrained workspaces"***

Advisor: Dr.Charles Boncelet (ECE and CS Professor, ECE Chair, Bell labs, DOD), *Initial PhD research developing a VLSI Neurocomputer*

- **Neurocomputer Design:** University of Delaware and Pennsylvania State University, including new mathematics for machine learning, neural network microprocessor architectures, and simulations for testing theories and architectures



Machine Intelligence
Symbolic AI vs Neural Networks
J Wunderlich PhD

SYMBOLIC AI uses special forms of computer programming to establish rules that lead to outcomes in a more efficient way; this includes using heuristics to prune the search space.

NEURAL NETWORKS use a collection of standardized decision nodes (Neurons), often organized into layers, to collectively generalize to solutions based on being trained with a data set (for supervise learning). The network LEARNS by modifying the strength of the connections between NEURONS to satisfy all of the training set by making small incremental changes in the connection weights over many iterations of reacting to the training set. Then, after learning, the machine can not only rapidly react to input of the exemplars in the training set, but can also react in a desired way to many variations of the inputs.

Example: suppose you have two parents deciding between getting a puppy or a kitten for their baby to play with. So we assign a binary variable to this decision as 0 for a puppy, and 1 for a kitten.

<u>Non- Machine Intelligence case</u>: Parents agree that if either one of them really wants a certain kitten, the spouse will yield to that desire. This would be like a binary OR gate where the parents, assigned variables X and Y, would decide an outcome of 1 (for a kitten), So:

Mom	Dad		
X	Y	Decision	
0	0	0	Puppy
1	1	1	Kitten
2	0	1	Kitten
1	1	1	Kitten

And the decision process, without pruning the search space, would look like this:

1) If XY = 00, then decision equals puppy
2) Else if XY = 01, then decision equals kitten
3) Else if XY = 10, then decision equals kitten
4) Else if XY = 11, then decision equals kitten

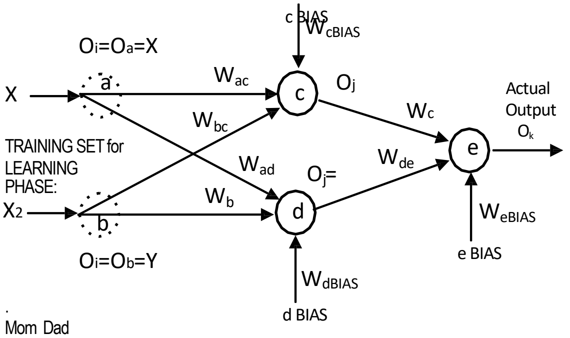
<u>SYMBOLIC AI case</u> :

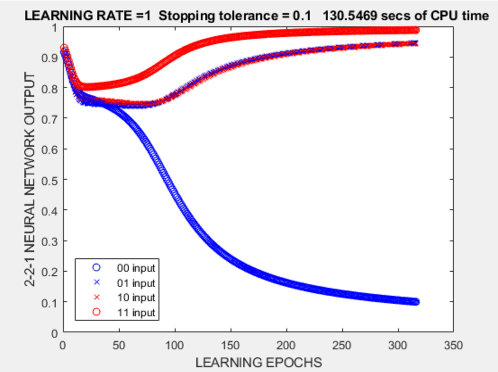
Parents 20% confident in their choice:

Mom	Dad			
X	Y	Decision		
0.2	0.2	0	Puppy	with XX% confidence
0.2	0.8	1	Kitten	with XX% confidence
0.8	0.2	1	Kitten	with XX% confidence
0.8	0.8	1	Kitten	with XX% confidence

The decision process would look like what we discussed previously for an “Expert System” picking a toy for a child: http://users.etown.edu/w/wunderjt//EGR_CS230/PACKET%2020F%20HANDOUT%20CogSci%20HCI%20Lecture%202018.pdf

<u>NEURAL NETWORK case 1</u>: Using same thoughts of the parents as in the Non- Machine Intelligence case





.

	Mom	Dad		
	X	Y	Decision	
Exemplar #1	0	0	0	Puppy
Exemplar #2	0	1	1	Kitten
Exemplar #3	1	0	1	Kitten
Exemplar #4	1	1	1	Kitten

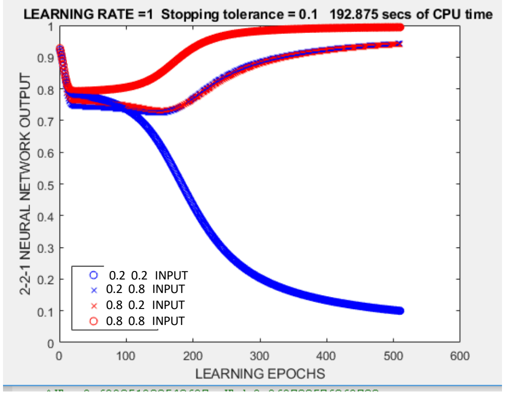
And the <u>LEARNING process</u>:

1) Initialize the inter-neuron connection weights to randomized values
2) Feed the neural network one examplar at a time, each time using the error between desired output in actual output to change connection-weights between neurons
3) Repeat (2) until the output error is within reasonable proximity of desired output for every exemplar (e.g., Decision<=0.1 for puppy, Decision>=0.9 for kitten). Each time you do this with the entire training set is called an "EPOCH". The LEARNING PHASE can take thousands of EPOCHS.
4) After learning is done, the neural network will react instantly to not only binary inputs, but variations of the inputs.

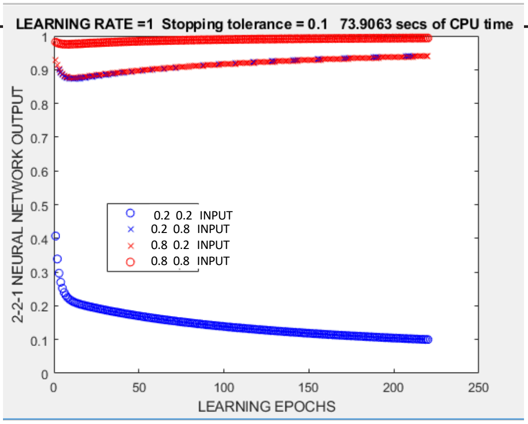
<u>NEURAL NETWORK case 2</u>: Using same thoughts of the parents as in Symbolic AI case

	Mom	Dad		
	X	Y	Decision	
Exemplar #1	**0.2**	**0.2**	0	Puppy
Exemplar #2	**0.2**	**0.8**	1	Kitten
Exemplar #3	**0.8**	**0.2**	1	Kitten
Exemplar #4	**0.8**	**0.8**	1	Kitten

IT STILL LEARNED, BUT IT JUST TOOK LONGER



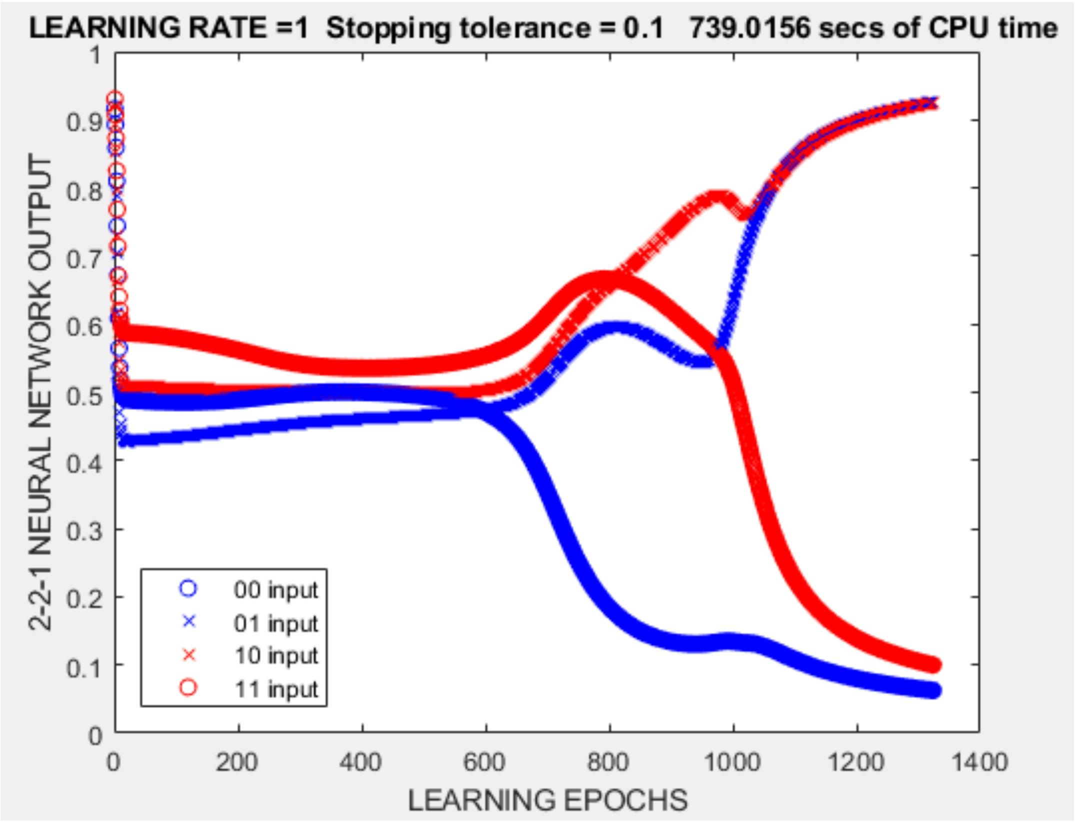
But if we use the weights from the learned “OR” of Neural Network Case 1, **IT LEARNS MUCH FASTER:**



NEURAL NETWORK case 3: Neural Network Case 1 after learning, instant result (no learning needed) for any exemplar input in training set
NEURAL NETWORK case 4: Neural Network Case 1 after learning, instant result (no learning needed) for any exemplar input with a little bit of noise corrupting input (e.g., 5%)

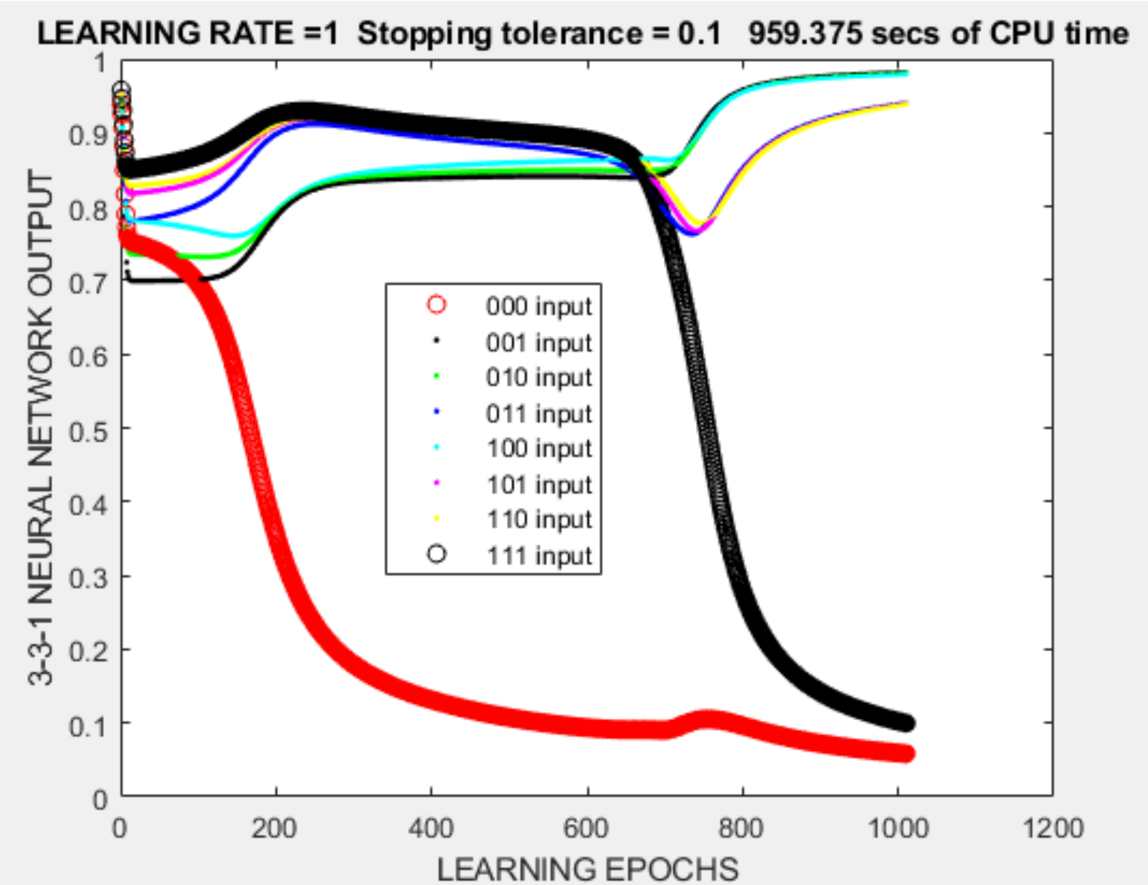
NEURAL NETWORK case 5: For some strange reason, the parents only get a kitten if just one of them wants it, otherwise they get a puppy. This is an XOR function which is harder for a Neural Network to learn because it is a “NONLINEAR SEPARABLE” problem -- think about the logic of what it is trying to solve; if gets no input as in Exemplar #1, nothing comes out, and then if it gets input from either X or Y, it fires, but if both X and Y input 1, nothing comes out – this is counterintuitive – and the Neural Network thinks so too; so it must struggle to solve this, and it therefore takes longer to learn.

	Mom	Dad	
	X	Y	Decision
Exemplar #1	0	0	1 Puppy
Exemplar #2	0	1	2 Kitten
Exemplar #3	1	0	1 Kitten
Exemplar #4	1	1	**0** Puppy



__NEURAL NETWORK case 6__: For some strange reason, the parents only get a puppy if both of them, plus the neighbor, all either agree to get a puppy, or for some really odd reason all want a kitten. This is also a NONLINEAR SEPARABLE problem. We need three inputs, and 1 output, and we choose to have three neurons in the input layer to better facilitate learning.

	Mom	Dad	Neighbor	
	X	Y	Z	Decision
Exemplar #1	0	0	0	**0** Puppy
Exemplar #2	0	0	1	1 Kitten
Exemplar #3	0	1	0	1 Kitten
Exemplar #4	0	1	1	1 Kitten
Exemplar #5	1	0	0	1 Kitten
Exemplar #6	1	0	1	1 Kitten
Exemplar #7	1	1	0	1 Kitten
Exemplar #8	1	1	1	**0** Puppy

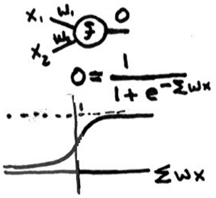


BACKPROPAGATION XOR SIMULATION

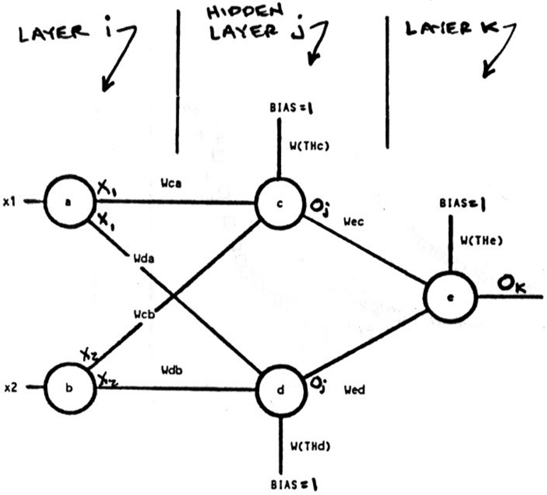
By Dr. Joseph WUNDERLICH

TRAINING SET:

EXEMPLAR #	INPUT X_1	X_2	d_k DESIRED OUTPUT
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0



$$O = \frac{1}{1+e^{-\Sigma wx}}$$



① PICK NETWORK ARCHITECTURE (2-2-1 HERE)

② PICK INITIAL WEIGHTS (RANDOM OK)

③ FEED NETWORK AN EXEMPLAR

④ FIND ERROR
[DESIRED OUTPUT(d_K) − ACTUAL OUTPUT (O_K)]

⑤ USE ERROR TO CHANGE WEIGHTS BETWEEN OUTPUT AND HIDDEN LAYER

$$\Delta W_{jK} = \left[\eta (d_K - O_K) O_K (1-O_K)\right] O_j \qquad \eta = \text{STEP SIZE}$$

⑥ BACKPROPATE ERROR TO CHANGE WT.'s BETWEEN HIDDEN AND INPUT LAYER

$$\Delta W_{ij} = \eta \left[\quad\right] W_{jK} \left(O_j (1-O_j)\right) X_i$$

⑦ REPEAT ③ TO ⑥ FOR ALL EXEMPLARS
⑧ REPEAT ⑦ UNTIL $O_K \approx d_K$ FOR ALL EXEMPLARS

MATLAB CODE:

```
%*****************************************************************************************
% A 2-2-1 or 3-3-1 back-propagation Neural Network
% by Joseph Wunderlich,Ph.D.
%
% 4/19/18:
%  Removed outdated "h=" handle, and input number, in Legend function
% 12/11/09:
%   Added disabling plotting for speed
%   Removed discontinued "flop(s)" function
% 12/10/09:
%   Created new file NN2_2009.m
%   Added 3-3-1 capability
% 3/29/04, fixed NN2.m:
%    WcBIAS=WcBIAS+dWcBIAS;
%    WdBIAS=WcBIAS+dWcBIAS;
%    WeBIAS=WcBIAS+dWcBIAS;
%
% Created original files "NN1.m" and "NN2.m" in 1990's
%***************************************************************************************
```

```
%******************** START TIMER AND INSTRUCTION COUNTER
 startTIME=cputime;
```

```
%******************* PICK AN ARCHITECTURE of 2-2-1 or 3-3-1 ***************************
ARCHITECTURE=1;          %"1" means 2-2-1, "2" means 3-3-1 Network Architecture

%******************** 2-2-1 and 3-3-1 INPUT
*******************************************************************
PLOTTING=1;              %Turn plotting on "1" or off "0" for speed
RATE=1;                  %Learning Rate
EPOCHcountMAX=2000;      %Stop if goal not reached after this many iterations
STOPtolerance=.1;        %How close to get to asymptotes at 0 or 1
 %Training sets of exemplars for each architecture:
EXEMPLAR_221=[0.2  0.2 0;       %input1, input2, and desiredoutput    exemplar #1
              0.2  for
                                                                      exemplar #2
                   0.8 1;       %input1, input2, and desiredoutput
                   for
              0.8  0.2 1;       %input1, input2, and desiredoutput    exemplar #3
                   for
              0.8  0.8 1];      %input1, input2, and desiredoutput    exemplar #4
                   for
 EXEMPLAR_331=[0 0  0 0;    %input1,2,3 and desiredoutput for exemplar #1
               0 0  1 1;    %input1,2,3 and desiredoutput for exemplar #2
               0 1  0 1;    %input1,2,3 and desiredoutput for exemplar #3
               0 1  1 1;    %input1,2,3 and desiredoutput for exemplar #4
               1 0  0 1;    %input1,2,3 and desiredoutput for exemplar #5
               1 0  1 1;    %input1,2,3 and desiredoutput for exemplar #6
               1 1  0 1;    %input1,2,3 and desiredoutput for exemplar #7
               1 1  1 0]; %input1,2,3 and desiredoutput for exemplar   #8
if ARCHITECTURE==1 % START IMPLIMENTING 2-2-1 ARCHITECTURE
%****************************** 2-2-1 INITIALIZATION
***********************************************
 Wac=.5; Wad=.6;
 Wbc=.7; Wbd=.8;
 Wce=.9; Wde=1;
 WcBIAS=1;  WdBIAS=1;    WeBIAS=1;
 %Here's the learned weights for binary "OR" behavior. Try these with
 %non-binary inputs to see Neural Network generalize an answer:
 Wac=2.587599880586697; Wad=3.252993565762999;
 Wbc=2.622851399549637; Wbd=3.263788576863788;
 Wce=3.400823954387610; Wde=4.650702692016342;
 WcBIAS=-1.450905811079590; WdBIAS=-1.774565860598957;    WeBIAS=-3.521403729223980;

 cBIAS=1;   dBIAS=1;     eBIAS=1;
 Exemplar1_OutputLAST=[.5 .5 .5]; %just to get it started
 Exemplar2_OutputLAST=[.5 .5 .5];
 Exemplar3_OutputLAST=[.5 .5 .5];
 Exemplar4_OutputLAST=[.5 .5 .5];
 EPOCHcount=0;
 n=1;
%***************************************************************************************
****
%*************************************   2-2-1 MAIN LOOP
******************************************
%***************************************************************************************
****

while   ((EPOCHcount) < EPOCHcountMAX)& ...
        ((abs(Exemplar1_OutputLAST(3)-EXEMPLAR_221(1,3))> STOPtolerance)|  ...
         (abs(Exemplar2_OutputLAST(3)-EXEMPLAR_221(2,3))> STOPtolerance)|  ...
         (abs(Exemplar3_OutputLAST(3)-EXEMPLAR_221(3,3))> STOPtolerance)|  ...
         (abs(Exemplar4_OutputLAST(3)-EXEMPLAR_221(4,3))> STOPtolerance))

  EPOCHcount=EPOCHcount+1;
  for i=1:4
   Oc=1/(1+exp((-(cBIAS*WcBIAS)- EXEMPLAR_221(i,1)*Wac - EXEMPLAR_221(i,2)*Wbc )));
```

```
Od=1/(1+exp((-(dBIAS*WdBIAS)- EXEMPLAR_221(i,1)*Wad - EXEMPLAR_221(i,2)*Wbd )));
Oe=1/(1+exp((-(eBIAS*WeBIAS)- Oc*Wce - Od*Wde )));
if i==1
   Exemplar1_OutputLAST=[EXEMPLAR_221(i,1) EXEMPLAR_221(i,2) Oe];
   if PLOTTING==1
      figure(1);
      plot(EPOCHcount,Oe,'bo');
      hold on;
   end;
 elseif i==2
    Exemplar2_OutputLAST=[EXEMPLAR_221(i,1) EXEMPLAR_221(i,2) Oe];
    if PLOTTING==1
       figure(1);
       plot(EPOCHcount,Oe,'bx');
       hold on;
    end;
     elseif i==3
        Exemplar3_OutputLAST=[EXEMPLAR_221(i,1) EXEMPLAR_221(i,2) Oe];
        if PLOTTING==1
           figure(1);
           plot(EPOCHcount,Oe,'rx');
           hold on;
        end;
            else
               Exemplar4_OutputLAST=[EXEMPLAR_221(i,1) EXEMPLAR_221(i,2) Oe];
               if PLOTTING==1
                  figure(1);
                  plot(EPOCHcount,Oe,'ro');
                  hold on;
               end;
            end;

            Exemplars_OutputLAST=[EPOCHcount/10000 Exemplar1_OutputLAST; ...
                                  EPOCHcount/10000  Exemplar2_OutputLAST;  ...
                                  EPOCHcount/10000  Exemplar3_OutputLAST;  ...
                                  EPOCHcount/10000 Exemplar4_OutputLAST]

error=EXEMPLAR_221(i,3)-Oe;
errorprop=error*Oe*(1-Oe);

dWeBIAS=RATE*errorprop*eBIAS;
dWce=   RATE*errorprop*Oc;
dWde=   RATE*errorprop*Od;

dWcBIAS=RATE*Oc*(1-Oc)*(errorprop*Wce)*cBIAS;
dWac=   RATE*Oc*(1-Oc)*(errorprop*Wce)*EXEMPLAR_221(i,1);
dWbc=   RATE*Oc*(1-Oc)*(errorprop*Wce)*EXEMPLAR_221(i,2);
dWdBIAS=RATE*Od*(1-Od)*(errorprop*Wde)*dBIAS;
dWad=   RATE*Od*(1-Od)*(errorprop*Wde)*EXEMPLAR_221(i,1);
dWbd=   RATE*Od*(1-Od)*(errorprop*Wde)*EXEMPLAR_221(i,2);

 Wac=Wac+dWac;
 Wad=Wad+dWad;
 Wbc=Wbc+dWbc;
 Wbd=Wbd+dWbd;
 Wce=Wce+dWce;
 Wde=Wde+dWde;
 WcBIAS=WcBIAS+dWcBIAS;
 WdBIAS=WdBIAS+dWdBIAS;
 WeBIAS=WeBIAS+dWeBIAS;
 Wdisplay=[Wac Wad Wbc Wbd Wce Wde WcBIAS WdBIAS WeBIAS];
```

```
   n=n+1;
```

```
end;
end;

EPOCHcount
endTIME=cputime-startTIME

if PLOTTING==1
   figure(1);                    %open figure window #1
%   % axis([-120 335 -50 300]);    %define x and y axis for figure window #1
   title(['LEARNING RATE =',num2str(RATE), '   Stopping tolerance = 
',num2str(STOPtolerance),'    ' ...
   num2str(endTIME), ' secs of CPU time ']);
   xlabel('LEARNING EPOCHS');
   ylabel('2-2-1 NEURAL NETWORK OUTPUT');
   legend('00 input','01 input','10 input','11 input','Location','southwest');
   hold on;
end;
%*****************************************************************************************
%*************************************** END 2-2-1 MAIN LOOP
%*****************************************************************************************


%*************************************** END 2-2-1 ARCITECTURE 
************************************************
%*****************************************************************************************


%*****************************************************************************************
%************************************** BEGIN 3-3-1 ARCITECTURE 
************************************************


elseif ARCHITECTURE==2   % START IMPLIMENTING 3-3-1 ARCHITECTURE
%******************************************* 3-3-1 INITIALIZATION *********************
 %Weight Values
  %A,B,C are input layer neurons
  %D,E,F are hidden layer neurons
  %G is output layer neuron
Wad= .4;    Wae= .45;   Waf= .5;
Wbd= .55;   Wbe= .6;    Wbf= .65;
Wcd= .7;    Wce= .75;   Wcf= .8;
Wdg= .85;   Weg= .9;    Wfg= .63;

%Bias values (MAY BE CHANGED BASED ON a concurrent SITUATION)
dBIAS= 1;   WdBIAS=1;
eBIAS= 1;   WeBIAS=1;
fBIAS= 1;   WfBIAS=1;
gBIAS= 1;   WgBIAS=1;

 Exemplar1_OutputLAST=[.5  .5 .5 .5]; %just to get it started
 Exemplar2_OutputLAST=[.5  .5 .5 .5];
 Exemplar3_OutputLAST=[.5  .5 .5 .5];
 Exemplar4_OutputLAST=[.5  .5 .5 .5];
 Exemplar5_OutputLAST=[.5  .5 .5 .5];
 Exemplar6_OutputLAST=[.5  .5 .5 .5];
 Exemplar7_OutputLAST=[.5  .5 .5 .5];
 Exemplar8_OutputLAST=[.5  .5 .5 .5];
 EPOCHcount=0;
 n=1;
%*****************************************************************************************
%**************************************   3-3-1 MAIN LOOP   *************************
%*****************************************************************************************
```

```
while  ((EPOCHcount) < EPOCHcountMAX)& ...
       ((abs(Exemplar1_OutputLAST(4)-EXEMPLAR_331(1,4))>  STOPtolerance)|  ...
        (abs(Exemplar2_OutputLAST(4)-EXEMPLAR_331(2,4))>  STOPtolerance)|  ...
        (abs(Exemplar3_OutputLAST(4)-EXEMPLAR_331(3,4))>  STOPtolerance)|  ...
        (abs(Exemplar4_OutputLAST(4)-EXEMPLAR_331(4,4))>  STOPtolerance)|  ...
        (abs(Exemplar5_OutputLAST(4)-EXEMPLAR_331(5,4))>  STOPtolerance)|  ...
        (abs(Exemplar6_OutputLAST(4)-EXEMPLAR_331(6,4))>  STOPtolerance)|  ...
        (abs(Exemplar7_OutputLAST(4)-EXEMPLAR_331(7,4))>  STOPtolerance)|  ...
        (abs(Exemplar8_OutputLAST(4)-EXEMPLAR_331(8,4))> STOPtolerance))

  EPOCHcount=EPOCHcount+1;
  for i=1:8
 Od=1/(1+exp((-(dBIAS*WdBIAS)- EXEMPLAR_331(i,1)*Wad - EXEMPLAR_331(i,2)*Wbd -
EXEMPLAR_331(i,3)*Wcd )));
 Oe=1/(1+exp((-(eBIAS*WeBIAS)- EXEMPLAR_331(i,1)*Wae - EXEMPLAR_331(i,2)*Wbe -
EXEMPLAR_331(i,3)*Wce )));
 Of=1/(1+exp((-(fBIAS*WfBIAS)- EXEMPLAR_331(i,1)*Waf - EXEMPLAR_331(i,2)*Wbf -
EXEMPLAR_331(i,3)*Wcf )));
 Og=1/(1+exp((-(gBIAS*WgBIAS)- Od*Wdg - Oe*Weg - Of*Wfg )));

 % 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray',
'lightBlue', 'orange', 'darkGreen'
 if i==1
   Exemplar1_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
   if PLOTTING==1
      figure(1);
      plot(EPOCHcount,Og,'redo');
      hold on;
   end;
   elseif i==2
    Exemplar2_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
    if PLOTTING==1
       figure(1);
       plot(EPOCHcount,Og,'black.');
       hold on;
    end;
    elseif i==3
     Exemplar3_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
     if PLOTTING==1
        figure(1);
        plot(EPOCHcount,Og,'green.');
        hold on;
     end;
     elseif i==4
      Exemplar4_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
      if PLOTTING==1
         figure(1);
         plot(EPOCHcount,Og,'blue.');
         hold on;
      end;
      elseif i==5
       Exemplar5_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
       if PLOTTING==1
          figure(1);
          plot(EPOCHcount,Og,'cyan.');
          hold on;
       end;
       elseif i==6
        Exemplar6_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
        if PLOTTING==1
           figure(1);
           plot(EPOCHcount,Og,'magenta.');
           hold on;
        end;
        elseif i==7
         Exemplar7_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
         if PLOTTING==1
            figure(1);
            plot(EPOCHcount,Og,'yellow.');
            hold on;
         end;
         else
          Exemplar8_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3)
Og];
          if PLOTTING==1
             figure(1);
             plot(EPOCHcount,Og,'blacko');
             hold on;
          end;
          end;

       Exemplars_OutputLAST=[EPOCHcount/10000 Exemplar1_OutputLAST; ...
                             EPOCHcount/10000 Exemplar2_OutputLAST;  ...
                             EPOCHcount/10000 Exemplar3_OutputLAST;  ...
                             EPOCHcount/10000 Exemplar4_OutputLAST;  ...
                             EPOCHcount/10000 Exemplar5_OutputLAST;  ...
                             EPOCHcount/10000 Exemplar6_OutputLAST;  ...
                             EPOCHcount/10000 Exemplar7_OutputLAST;  ...
                             EPOCHcount/10000 Exemplar8_OutputLAST]

   error=EXEMPLAR_331(i,4)-Og;
   errorprop=error*Og*(1-Og);

   dWgBIAS=RATE*errorprop*gBIAS;
   dWdg=   RATE*errorprop*Od;
   dWeg=   RATE*errorprop*Oe;
   dWfg=   RATE*errorprop*Of;

   dWdBIAS=RATE*Od*(1-Od)*(errorprop*Wdg)*dBIAS;
   dWad=    RATE*Od*(1-Od)*(errorprop*Wdg)*EXEMPLAR_331(i,1);
   dWbd=    RATE*Od*(1-Od)*(errorprop*Wdg)*EXEMPLAR_331(i,2);
   dWcd= RATE*Od*(1-Od)*(errorprop*Wdg)*EXEMPLAR_331(i,3);

   dWeBIAS=RATE*Oe*(1-Oe)*(errorprop*Weg)*eBIAS;
   dWae=   RATE*Oe*(1-Oe)*(errorprop*Weg)*EXEMPLAR_331(i,1);
   dWbe=   RATE*Oe*(1-Oe)*(errorprop*Weg)*EXEMPLAR_331(i,2);
   dWce=   RATE*Oe*(1-Oe)*(errorprop*Weg)*EXEMPLAR_331(i,3);

   dWfBIAS=RATE*Of*(1-Of)*(errorprop*Wfg)*fBIAS;
   dWaf=   RATE*Of*(1-Of)*(errorprop*Wfg)*EXEMPLAR_331(i,1);
   dWbf=   RATE*Of*(1-Of)*(errorprop*Wfg)*EXEMPLAR_331(i,2);
   dWcf=   RATE*Of*(1-Of)*(errorprop*Wfg)*EXEMPLAR_331(i,3);

    Wad=Wad+dWad;  Wbd=Wbd+dWbd;  Wcd=Wcd+dWcd;
    Wae=Wae+dWae;  Wbe=Wbe+dWbe;  Wce=Wce+dWce;
    Waf=Waf+dWaf; Wbf=Wbf+dWbf; Wcf=Wcf+dWcf;

    Wdg=Wdg+dWdg; Weg=Weg+dWeg; Wfg=Wfg+dWfg;

    WdBIAS=WdBIAS+dWdBIAS;
    WeBIAS=WeBIAS+dWeBIAS;
    WfBIAS=WfBIAS+dWfBIAS;
    WgBIAS=WgBIAS+dWgBIAS;
    Wdisplay=[Wad Wbd Wcd Wae Wbe Wce Waf Wbf Wcf Wdg Weg Wfg WdBIAS WeBIAS WfBIAS
WgBIAS];

   n=n+1;

end;
end;

EPOCHcount
endTIME=cputime-startTIME

 if PLOTTING==1
   figure(1);                      %open figure window #1
   % axis([-120 335 -50 300]);    %define x and y axis for figure window #1
   title(['LEARNING RATE =',num2str(RATE), '   Stopping tolerance =
',num2str(STOPtolerance),'    ' ...
      num2str(endTIME), ' secs of CPU time ']);
   xlabel('LEARNING EPOCHS');
   ylabel('3-3-1 NEURAL NETWORK OUTPUT');
   legend('000 input','001 input','010 input','011 input','100 input','101 input','110
input','111 input','Location','southwest');
   hold on;
 end;
%****************************************************************************************
%**************************************** END 3-3-1 MAIN LOOP *************************
%****************************************************************************************
%**************************************** END 3-3-1 ARCITECTURE *************************
%****************************************************************************************
***************
end;
```

In 2020 you can you do **deep learning** and **neural networks** using the parallel processing capabilities of graphics cards on simple personal computers! …. Using “CUDA”

A.11 GRAPHICS CARDS, Historical Perspective

(edited by J Wunderlich PhD in 2020)

Graphics Pipeline Evolution

3D graphics pipeline hardware evolved from the large expensive systems of the early 1980s to small workstations and then to PC accelerators in the 1990s, to *$X,000 graphics cards of the 2020's* During this period, three major transitions occurred:

1. Performance-leading graphics subsystems **PRICE** changed from $50,000 in 1980's **down** to $200 in 1990's, then up to $X,0000 in 2020's.
2. **PERFORMANCE** increased from 50 million PIXELS PER SECOND in 1980's to 1 billion pixels per second in 1990''s and from 100,000 VERTICES PER SECOND to 10 million vertices per second in the 1990's. In the 2020's performance is measured more in **FRAMES PER SECOND (FPS)**
3. Hardware **RENDERING** evolved **from WIREFRAME to FILLED POLYGONS, to FULL-SCENE TEXTURE MAPPING**

Fixed-Function Graphics Pipelines

Throughout the early evolution, graphics hardware was configurable, but not programmable by the application developer. With each generation, incremental improvements were offered. But developers were growing more sophisticated and asking for more new features than could be reasonably offered as built-in fixed functions. The NVIDIA GeForce 3, described by Lindholm, et al. [2001], took the first step toward true general shader programmability. It exposed to the application developer what had been the private internal instruction set of the floating-point vertex engine. This coincided with the release of Microsoft's DirectX 8 and OpenGL's vertex shader extensions. Later GPUs, at the time of DirectX 9, extended general programmability and floating point capability to the pixel fragment stage, and made texture available at the vertex stage. The ATI Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel fragment processor programmed with DirectX 9 and OpenGL. The GeForce FX added 32-bit floating-point pixel processors. This was part of a general trend toward unifying the functionality of the different stages, at least as far as the application programmer was concerned. NVIDIA's GeForce 6800 and 7800 series were built with separate processor designs and separate hardware dedicated to the vertex and to the fragment processing. The XBox 360 introduced an early unified processor GPU in 2005, allowing vertex and pixel shaders to execute on the sameprocessor.

Evolution of Programmable Real-Time Graphics

Graphics architecture has evolved from a simple pipline for drawing wireframe diagrams to a highly parallel design consisting of several deep parallel pipelines capable of rendering complex interactive imagery that appears three-dimensional. Concurrently, many of the calculations involved became far more sophisticated and user programmable.

In these **GRAPHICS PIPELINES in GPU (Graphics Processing Unit, on Graphics card, or "Integrated" on Motherboard))**, certain stages do a great deal of floating-point arithmetic on completely independent data, such as transforming the position of triangle vertexes or generating pixel colors. This data independence is a key difference between GPUs and CPUs. A single frame, rendered in 1/60th of a second, might have 1 million triangles and 6 million pixels. The opportunity to use hardware parallelism to exploit this data independence is tremendous.

The specific functions executed at a few graphics pipeline stages vary with rendering algorithms and have evolved to be programmable. Vertex programs map the position of triangle vertices on to the screen, altering their position, color, or orientation. Typically a vertex shader thread inputs a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry programs operate on primitives defined by multiple vertices, changing them or generating additional primitives. Pixel fragment shaders each "shade" one pixel, computing a floating-point red, green, blue, alpha (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. For all three types of graphics shaders, program instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects.

Between these programmable graphics pipeline stages are dozens of fixed- function stages which perform well-defined tasks far more efficiently than a programmable processor could and which would benefit far less from program- mability. For example, between the geometry processing stage and the pixel processing stage is a "rasterizer," a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's bound- aries. Together, the mix of programmable and fixed-function stages is engineered to balance extreme performance with user control over the rendering algorithms. Common rendering algorithms perform a single pass over input primitives and access other memory resources in a highly coherent manner; these algorithms pro- vide excellent bandwidth utilization and are largely insensitive to memory latency. Combined with a pixel shader workload that is usually compute-limited, these characteristics have guided GPUs along a different evolutionary path than CPUs. Whereas CPU die area is dominated by cache memory, GPUs are dominated by floating-point datapath and fixed-function logic. GPU memory interfaces empha- size bandwidth over latency (since latency can be readily hidden by a high thread count); indeed, bandwidth is typically many times higher than a CPU, exceeding 100 GB/second in some cases. The far-higher number of fine-grained lightweight threads effectively exploits the rich parallelism available.

Beginning with NVIDIA's GeForce 8800 GPU in 2006, the three programmable graphics stages are mapped to an array of unified processors; the logical graphics pipeline is physically a recirculating path that visits these processors three times, with much fixed-function graphics logic between visits. Since different rendering algorithms present wildly different loads among the three programmable stages, this unification provides processor load balancing.

Unified Graphics and Computing Processors

By the DirectX 10 generation, the functionality of vertex and pixel fragment shaders was to be made identical to the programmer, and in fact a new logical stage was introduced, the geometry shader, to process all the vertices of a primitive rather than vertices in isolation. The GeForce 8800 was designed with DirectX 10 in mind. Developers were coming up with more sophisticated shading algorithms, and this motivated a sharp increase in the available shader operation rate, particularly floating-point operations. NVIDIA chose to pursue a processor design with higher operating frequency than standard-cell methodologies had allowed to deliver the desired operation throughput as area-efficiently as possible. High-clock-speed design requires substantially more engineering effort, and this favored designing one processor, rather than two (or three, given the new geometry stage). It became worthwhile to take on the engineering challenges of a unified processor (load balancing and recirculation of a logical pipeline onto threads of the processor array) to get the benefits of one processor design.

GPGPU: an Intermediate Step

As DirectX 9–capable GPUs became available, some researchers took notice of the raw performance growth path of GPUs and began to explore the use of GPUs to solve complex parallel problems. DirectX 9 GPUs had been designed only to match the features required by the graphics API. To access the computational resources, a programmer had to cast their problem into native graphics operations. For example, to run many simultaneous instances of a pixel shader, a triangle had to be issued to the GPU (with clipping to a rectangle shape if that's what was desired). Shaders did not have the means to perform arbitrary scatter operations to memory. The only way to write a result to memory was to emit it as a pixel color value, and configure the framebuffer operation stage to write (or blend, if desired) the result to a two-dimensional framebuffer. Furthermore, the only way to get a result from one pass of computation to the next was to write all parallel results to a pixel framebuffer, then use that framebuffer as a texture map as input to the pixel fragment shader of the next stage of the computation. Mapping general computations to a GPU in this era was quite awkward. Nevertheless, intrepid researchers demonstrated a handful of useful applications with painstaking efforts. This field was called "GPGPU" for general purpose computing on GPUs.

GPU Computing

While developing the Tesla architecture for the GeForce 8800, NVIDIA realized its potential usefulness would be much greater if programmers could think of the GPU as a processor. NVIDIA selected a programming approach in which programmers would explicitly declare the data-parallel aspects of their workload.

For the DirectX 10 generation, NVIDIA had already begun work on a high-efficiency floating-point and integer processor that could run a variety of simultaneous workloads to support the logical graphics pipeline. This processor was designed to take advantage of the common case of groups of threads executing the same code path. NVIDIA added memory load and store instructions with integer byte addressing to support the requirements of compiled C programs. It introduced the thread block (cooperative thread array), grid of thread blocks, and barrier synchronization to dispatch and manage highly parallel computing work. Atomic memory operations were added. NVIDIA developed the **CUDA C/C++ compiler, libraries, and runtime software to enable programmers to readily access the new data-parallel computation model and develop applications**.

Scalable GPUs

Scalability has been an attractive feature of graphics systems from the beginning. Workstation graphics systems gave customers a choice in pixel horsepower by varying the number of pixel processor circuit boards installed. Prior to the mid-1990s PC graphics scaling was almost nonexistent. There was one option—the VGA controller. As 3D-capable accelerators appeared, the market had room for a range of offerings. 3dfx introduced multiboard scaling with the original SLI (Scan Line Interleave) on their Voodoo2, which held the performance crown for its time (1998). Also in 1998, NVIDIA introduced distinct products as variants on a single architecture with Riva TNT Ultra (high-performance) and Vanta (low-cost), first by speed binning and packaging, then with separate chip designs (GeForce 2 GTS & GeForce 2 MX). At present, for a given architecture generation, four or five separate GPU chip designs are needed to cover the range of desktop PC performance and price points. In addition, there are separate segments in notebook and workstation systems. After acquiring 3dfx, NVIDIA continued the multi-GPU SLI concept in 2004, starting with GeForce 6800—providing multi-GPU scalability transparently to the programmer and to the user. Functional behavior is identical across the scaling range; one application will run unchanged on any implementation of an architectural family.

CPUs are scaling to higher transistor counts by increasing the number of constant-performance cores on a die, rather than increasing the performance of a single core. At this writing the industry is transitioning from dual-core to quad-core, with eight-core not far behind. Programmers are forced to find fourfold to eightfold task parallelism to fully utilize these processors, and applications using task parallelism must be rewritten frequently to target each successive doubling of core count. In contrast, the highly multithreaded GPU encourages the use of many-fold data parallelism and thread parallelism, which readily scales to thousands of parallel threads on many processors. The GPU scalable parallel programming model for graphics and parallel computing is designed for transparent and portable scalability. A graphics program or CUDA program is written once and runs on a GPU with any number of processors. As shown in Section A.3, a CUDA programmer explicitly states both fine-grained and coarse-grained parallelism in a thread program by decomposing the problem into grids of thread blocks—the same program will run efficiently on GPUs or CPUs of any size in current and future generations as well.

Recent Developments

Academic and industrial work on applications using CUDA has produced hundreds of examples of successful CUDA programs. Many of these programs run the application tens or hundreds of times faster than multicore CPUs are capable of running them. Examples include n-body simulation, molecular modeling, computational finance, and oil and gas exploration data processing. Although many of these use single precision floating-point arithmetic, some problems require double precision. The recent arrival of double precision floating point in GPUs enables an even broader range of applications to benefit from GPU acceleration.

For a comprehensive list and examples of current developments in applications that are accelerated by GPUs, visit CUDAZone:
https://developer.nvidia.com/cuda-toolkit w.nvidia.com/CUDA.

Trends

Naturally, the number of processor cores will continue to increase in proportion to increases in available transistors as silicon processes improve. In addition, GPUs will continue to enjoy vigorous architectural evolution. Despite their demonstrated high performance on data-parallel applications, GPU core processors are still of relatively simple design. More aggressive techniques will be introduced with each successive architecture to increase the actual utilization of the calculating units. Because scalable parallel computing on GPUs is a new field, novel applications are rapidly being created. By studying them, GPU designers will discover and implement new machine optimizations**. In 2020, GPU's have over 2000 CUDA CORES**

Further Reading

Akeley, K. and T. Jermoluk [1988]. "High-Performance Polygon Rendering," *Proc. SIGGRAPH 1988* (August), 239–46.

Akeley, K. [1993]. "RealityEngine Graphics." *Proc. SIGGRAPH 1993* (August), 109–16.

Blelloch, G. B. [1990]. "Prefix Sums and Their Applications." In John H. Reif (Ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Francisco.