Image Processing Using One-Dimensional Processor Arrays

DAN W. HAMMERSTROM, MEMBER, IEEE, AND DANIEL P. LULICH

Invited Paper

Image processing (IP) is an ideal candidate for specialized architectures because of the sheer volume of data, the natural parallelism of IP algorithms, and the high demandfor IP solutions. The continually increasing circuit density of very large scale integration (VLSI) now makes it possible to integrate a complete parallel computer onto a single piece of silicon, significantly improving cost-performance for IP applications. The widespread use of IP applications such as desktop publishing and multimedia-and the limited performance of even the highest speed microprocessors on these tasks-reveal financial incentives for creating a specialized IP architecture. However, choosing the correct architecture requires the resolution of several complex design trade-offs.

The first half of this paper presents the design rationale for CNAPS™, a specialized one-dimensional (I-D) processor array developed by Adaptive Solutions Inc. In this context, we discuss the problem of Amdahl's law, which severely constrains special-purpose architectures. We also discuss specific architectural decisions such as the kind of parallelism, the computational precision of the processors, on-chip versus off-chip processor memory, and-most importantly-the interprocessor communication architecture. We argue that, for our particular set of applications, a 1 -D architecture gives the best "bang for the buck," even when compared to the more traditional two-dimensional (Z-D) architecture.

The rectangular structure of an image intuitively suggests that IP algorithms map efficiently to a 2-D processor array. Traditional IP architectures hence have consisted of parallel arrays of processors organized in 2-D grids. The configuration is often assumed to be "optimal" when the number of processors is equal to the number of image pixels and when each processor is interconnected with its eight nearest neighbors in a rectangular array. Such one-to-one configurations are almost always too expensive to deploy. The number of processors is typically hundreds to thousands of times less than the number of pixels. Under these conditions, intuitions about optimal mappings and topologies begin to break down. In our application domains, where the number of pixels greatly exceeds the number of processors, and for our target applications a I-D array offers the same performance as a 2-D array, usually at a lower cost.

The second half of this paper describes how several simple algorithms map to the CNAPS array. Our results show that the CNAPS 1 -D array offers excellent performance over a range of IP algorithms. We also briefly look at the performance of CNAPS as a pattern recognition engine because many image processing and pattern recognition problems are intimately related.

I. INTRODUCTION

Since near the beginning of the computer age, engineers have used computers to manipulate and transform digital representations of images. This discipline, called image processing (IP), employs five fundamental classes of operation [1]:

1) image enhancement;
2) image restoration;
3) image analysis;
4) image compression; and
5) image synthesis.

IP [2] then consists of "applying digital techniques to digital images to form digital results, such as a new image or a list of extracted data." IP is used extensively in such diverse fields as medicine, astronomy, and entertainment.

Representing an image requires a large amount of data. Since an image is two-dimensional (2-D), the quantity of data grows as the square, $O(N^2)$, of the image size. Images such as video, often have a third, temporal dimension, which further increases the volume of data. A typical TV signal requires a communication rate of over 20 Mbytes/s, assuming 8 b per color. Performing a modest 50 operations on each pixel in each video frame therefore requires a computation rate of over one billion operations per second to achieve real-time processing rates.

As IP algorithms get more sophisticated and more complex, the performance needs of IP exceed the capability of general-purpose computers. As a result, IP and specialized "acceleration" hardware have a long history together.[1] In addition, many IP applications are generally regular and

Manuscript received September 1, 1995; revised April 19, 1996.

The authors are with Adaptive Solutions, Inc., Beaverton, OR 97006 USA.

Publisher Item Identifier S0018-9219(96)05392-3.

[1] We have limited our definition of IP to operating on 2-D images or on 2-D renditions of 3-D images. Accelerating the rendering of 3-D images requires another set of specialized architectures that are beyond the scope of this paper.

therefore relatively easy to parallelize.[2] IP is a natural *data parallel* problem because it involves applying many identical, independent operations to an array of data, This regularity in the problem lends itself to regular computer architectures, which are similarly straightforward to build and program. IP and specialized hardware therefore also have a long *future* together, especially because IP is becoming more common in large markets such as desktop publishing, multimedia, and image recognition.

As the requirements for image processing have increased, so have the capabilities of integrated circuit technology. The ability to integrate large numbers of transistors onto one piece of silicon and operate them at a very high frequency has increased at a constant rate over the last 20-30 years. This increase, often called *Moore's Law*, was formulated by Gordon Moore, one of the founders of Intel Corporation [3]. Moore's law states that silicon performance and density roughly double every 1.5 years. This continuing increase in silicon performance (coupled to decreases in implementation cost) motivated us to create custom silicon devices for accelerating IP and the related function of pattern recognition. The result of our design efforts is the CNAPS architecture and chip set.

The designer of a high-performance chip architecture must consider some serious pitfalls including the devastating effects of *Amdahl's law* [4], cost-performance trade-offs, and the appropriate range of functionality. We do not claim to have made the correct decision in all cases, and some decisions had better results than others. But we do feel that a discussion of the design rationale and of the applications used by the resulting architecture would be of interest to those who work in IP and beyond.

This paper has two major parts. The first describes the CNAPS™ architecture and the design rationale for its major features. This discussion concentrates on the interprocessor communication design, since that topic offers insight into building large VLSI-based arrays of parallel processors. The second part describes some real-world applications that use CNAPS to show how these applications map to the architecture.

II. CNAPS Architecture-Design Rationale

One goal of CNAPS was to exploit the data parallelism of IP to give a significant cost-performance advantage over existing high-speed microprocessor technology ("killer micros"). Another goal was to make CNAPS fairly easy to program and flexible enough to support a fairly wide range of algorithms. In addition, we wanted it to be small, compact, and inexpensive so that an entire IP engine could fit on one PC card for desktop applications.

The first problem a designer of special-purpose computers runs into is Amdahl's Law.[3] When parallelizing an application, Amdahl says that the part of the application that is not sped up dominates the total speed-up possible. For example, suppose that you have invented a new special-purpose chip that can speed up part of an application up by a factor of 1000. As often happens, the portion of the application your chip accelerate accounts for only a small part of the total application's cycles, say 30%. Consequently, the total application is sped up only by 30%. Unless your new chip is inexpensive and easy to program, few customers will use it, since by the time they port their applications to it, general-purpose processors will be 30% faster anyhow. IP is one of the few areas where specialized architectures can compete with "killer micros" in spite of Amdahl's law, mainly because IP algorithms have a high degree of natural parallelism that makes it relatively easy to create powerful, economical IP engines. Nevertheless, the designer of an IP engine for use with general-purpose applications must be aware of Amdahl's law.

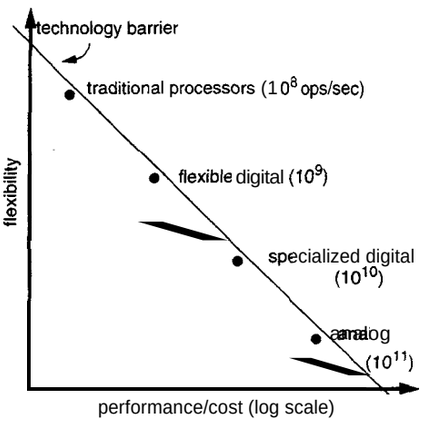


Fig. 1. "Flexibility" versus cost-performance.

The CNAPS solution to Amdahl's law was two-fold. The first objective was to find a specific set of applications that share similar computational characteristics and optimize the architecture for these and nothing else. We make no claims that CNAPS is a general-purpose machine. It was designed for IP and pattern recognition (PR).

The second objective was to create an architecture that was as flexible as possible within the general performance objectives needed for our target markets, another strong argument for a programmable architecture. Fig. 1 demonstrates this trade-off. The horizontal dimension indicates cost-performance,[4] usually measured in operations/second per dollar. The vertical dimension is more subjective, but describes a "flexibility" measure. Although it is difficult to quantify, computer architects agree that there is a clear trade-off between flexibility and performance. When you design a chip for a specific application, you can leverage significant parallel opportunities and forego certain kinds of computation, thus allowing a higher cost-performance than a general-purpose machine that must do everything. Through the middle of the figure is the "technology barrier," which moves to the right over time. The technology banier

[2] This parallelism has been leveraged in several image processing architectures from systolic pipelines to parallel processor arrays. The latter will be the primary focus of this article.

[3] Amdahl's law [4] states simply that an application is only as fast as its slowest component.

[4] We use the colloquial term "cost-performance" to mean performance divided by cost.

results from practical issues: you can put only so many transistors on a chip, they cost so much, and can perform only so much computational work. At the left end of the spectrum are the latest generations of general-purpose processors (i.e., the Intel Pentium™), which give reasonable cost-performance over a wide range of applications. At the other end are the very specialized devices, often built to execute a single function (i.e., a modem chip).

Flexibility is extremely important, because even a simple pattern recognition application must execute a fairly broad range of algorithms. Amdahl tells us that the total system speed up is proportional to the percentage of algorithms that we cannot map to our specialized processor-that is, those algorithms that must be executed by the host. So except for certain high-volume, very specialized chips, Amdahl tends to push specialized architectures to the left end of the scale. An architecture that is too close to the general-purpose processors in cost-performance cannot compete with them. The trick when designing CNAPS was to create enough flexibility to avoid the perils of Amdahl's law, yet have a sufficiently superior cost-performance over existing microprocessor technology to justify its cost in a system.

The next few sections highlight and justify the major architecture decisions in light of the above reasoning.

A. SIMD Parallelism

Most parallel computer architectures fall into two broad categories: multiple instruction stream multiple data stream (MIMD) and single instruction stream and single data stream (SIMD) [5]. In M&ID-parallel machines, each arithmetic logic unit (ALU) unit has its own, independent instruction sequencer. Each ALU therefore can act independently. In SIMD-parallel machines, all ALU's share one sequencer. Each ALU therefore does the same thing at the same time, but on a different piece of data. SIMD computation is often called data parallelism. MIMD is generally more powerful and flexible than SIMD, but SIMD is less expensive. SIMD is also much easier to program because it has one thread of control. MIMD, on the other hand, has multiple, interacting, relatively asynchronous control threads, greatly complicating the programming model.

Almost all custom IP architectures have been SIMD, since IP is very much a data parallel problem [5]. We decided to use SIMD parallelism for the same reasons: our target algorithms consisted of performing the same operation, in parallel, over many pieces of data in large, arrayed computing structures. We were also attracted to the simpler programming model and to the favorable cost-performance achieved through a simple restriction in functionality.

So our first decision was to obtain the needed performance by exploiting the inherent data parallelism in the algorithms used in our target applications, and to implement this parallelism in a SIMD design. Our machine would consist of an array of processor nodes (PN's)[5] operating in lock-step synchrony and controlled by one instruction sequence unit. As with all SIMD architectures, the PN's would be able to conditionally execute an instruction based on some internal logical condition (for example, adder output equal to zero, or negative).

[5] For historical reasons, we use the term processor node (PN) instead of the traditional SIMD term processor element (PE).

B. Arithmetic Precision

The next major architectural question concerned the representation of the data computed by each PN. The alternatives here broadly range from bit-serial to full bit-parallel floating point. Bit-serial representation means that each bit of an arithmetic operation is computed one bit at a time. A fixed-point add of n bits thus requires O(n) clocks, and a multiply requires $O(n^2)$ clocks. Bit-serialism thus allows arbitrary precision (just use more clocks), which is useful in many IP problems. Bit-serial PN's are extremely simple, so many PN's can be placed in one chip. Inter-PN interconnect likewise tends to be serial.

Bit-parallel designs, on the other hand, can perform entire adds or multiplies in one clock cycle. This bit-parallel arithmetic tends to be easier to program and faster than bit-serial arithmetic. A drawback is that bit-parallel PN's are harder to design than bit-serial PN's and also require more silicon area.[6] In addition to "pure" bit-serial or bit-parallel designs, there are also a range of intermediate designs. For example, an ALU could do 4-b arithmetic with the programmer doing higher precision fixed-point operations by serially combining 4-b operations.

All of these options lead to different silicon space/computation time trade-offs. We decided to implement bit-parallel arithmetic because it is easier to program and faster for higher precision. The next question was whether to use a fixed-point (where the programmer statically manages the radix point) or a floating-point representation (where the machine itself dynamically manages the radix point). Floating point is much easier for programmers to use and generally has better arithmetic characteristics such as greater precision and dynamic range. Unfortunately, floating point is expensive, since it requires hardware to perform the various shifting and normalization operations, exponent arithmetic, and bit truncation (usually rounding).

For CNAPS, floating point had an additional expense. Our PN's were small enough to use PN redundancy to improve semiconductor yield (the number of good dice-that is, chips-per wafer), much like the way many memory manufacturers use extra columns and rows to replace faulty rows or columns. Due to the current defect density rates of silicon processes, the use of redundant components can significantly improve yield. It works only if the unit of redundancy is small, the exact size being a complex function of the percentage of nonredundant circuitry (such as global interconnect) and the defect density per unit of

[6] The cost of a chip is generally proportional to its size, since silicon wafers have a fixed cost, and the number of finished chips per wafer depends on the size of the chip. Chip size and wafer yield have a complex. nonlinear relationship, not only do bigger chips take up more space on the wafer, but also each chip is more likely to fail due to manufacturing defects.

wafer area. In our analysis, a 32-b floating-point PN was roughly three to four times larger than the 8116-b fixed-point PN we used,[7] assuming that we implemented the IEEE floating-point standard. This analysis showed that the floating-point PN was too large to effectively employ redundancy, but the smaller 8/16-b PN could do so and double the yield. Extensive simulation showed that the problem domains we have targeted, such as IP, can operate quite effectively at 8 and 16-b precision.

C. *Processor Memory Architecture*

As the PN's compute, they need input data to compute with. In most IP applications, the multiply-accumulate (MAC) operation dominates, which requires two input operands per clock. In most of our computations, the PN's can share one of those operands, but the other is unique to each PN. The question then is, where are these unique operands stored, and how are they moved to each PN's arithmetic unit? The two general techniques are on-chip storage and off-chip storage.

Storing operands off-chip requires moving them onto the chip and into each PN during computation, which in turn requires a separate, independent pathway from an off-chip memory to each PN. An advantage is that each PN can have essentially unlimited memory space, implemented in commercially available memory. A drawback is that the limited bandwidth going on and off the chip (the maximum number of pins, their maximum switching rate, and their power consumption) imposes limits on the speed and/or number of PN's on the chip. Likewise, due to off-chip bandwidth limitations, off-chip memory systems generally use a single address for all PN memory-that is, each PN must access the same location as its brethren.

The other technique is to put a memory next to each PN. One advantage is reducing the off-chip circuitry, pins, and power dissipation. Another is that each PN can now have one clock access to its own local memory, and each PN can access a different location, allowing more complex functionality such as table lookup and per PN indirect addressing. The big disadvantages are the expense, in silicon area, of the memory, and limitations on the total amount of memory available.

The memory design was one of the most complex architecture decisions, since both techniques had comparable advantages and disadvantages. Several factors influenced our final decision to use on-chip per-PN memory. The primary influence was the silicon capability available to us. We were able to use a commercial SRAM process that permitted a large number of dense memory cells on the chip. The first version of the CNAPS architecture was fabricated in a 0.8 μm CMOS SRAM process with four layers of metal and two layer polysilicon. The SRAM cell was 47 μm^2, using four transistors (with poly load pull

[7] It is **possible to have a** 16-b Roatinp-point number, but it is useful only for a limited range of algorithms. For a commercial product, we felt that any floating point representation would have to be the one that people are used to using. The analysis therefore assumed 32-b IEEE standard representation. with an 8-b exponent and 24-b mantissa.

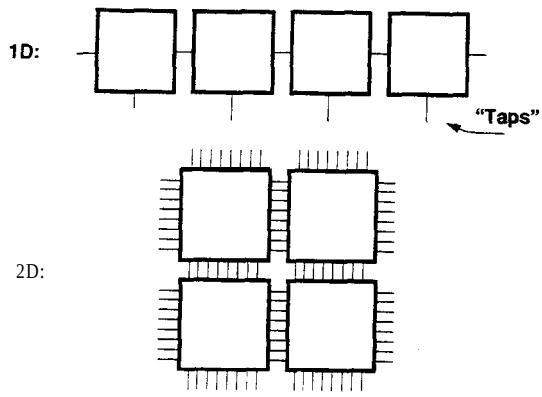


Fig. 2. I-D and 2-D processor arrays. Each "chip" has 64 processors.

ups). The second capability was using redundant components via a proprietary laser fusing technology. This redundancy-developed independently for memory products-coupled with the ability to stitch together multiple mask exposures, let us build a single, large die. The first CNAPS implementation had 64 working PN's on a single chip: 80 were fabricated in a 8 x 10 array, leaving 16 spare for redundancy. This level of integration prevented the use of off-chip memory, but it permitted significant economies of scale, lowering the price per PN. So the final decision was between limited (on-chip) memory per PN, but low per PN cost, or unlimited (off-chip) memory per PN, but high PN cost. We analyzed the mapping of several algorithms to the architecture, such as those shown in the second half of this paper. This analysis is not repeated here, but our conclusion was that the algorithms could adequately execute within a 4 KB PN memory. In retrospect, 4 KB per PN was a sufficiently large "local programmable cache" for a broad range of IP and PR algorithms, even though some applications (general speech recognition being one of the most prominent) could not use CNAPS due to memory constraints.

D. Inter-processor Communication Architecture

The next major decision was how to connect the PN's to each other. Over the years, several interconnection architectures have been developed for parallel processor arrays [5]. Choosing the wrong interconnection architecture could severely impact the cost-performance of CNAPS. In studying our target applications, we realized that they involved almost exclusively global and local communication. We concluded that providing efficient global, intermediate, and local communication was too expensive, so we decided to provide local and global connectivity.

In our target applications, the global interconnect patterns we looked at were dominated by "one to many" and "many to many" style communication. The most efficient way to perform this type of communication is with a simple broadcast bus connecting all PN's. The sequential broadcast of data allows us to perform $O(n^2)$ communication in O(n) clocks. Broadcast is inefficient, however, when loading per-PN data (such as filling PN memories) because each

chip does not have enough pins to give each PN its own path to memory. Our solution was to use "multiple tapped broadcast," where a ***tap*** can broadcast to a subset of PN's, with each tap broadcasting in parallel to its subset. The top of Fig. 2 shows a 1-D array architecture of four chips with a single I/O "tap" per chip. The current implementation of CNAPS offers one tap per chip, which in multichip systems matches the bandwidth of most buses.

We use multiple taps in our 1-D architecture to increase the ratio of external chip I/O paths to PN's. Today's silicon technology provides a particular ratio of pin count (proportional to the length of the chip periphery) to the area for internal logic. These silicon area and power limitations ultimately constrain the ratio of I/O per PN. A 1-D architecture generally has too low a ratio, thereby not effectively using the available I/O capabilities. As we will show below, a 2-D architecture actually has too high a ratio. So, though it is also possible to add multiple taps to a 2-D array, doing so would increase what is already too high of a ratio of I/O to compute.

For outputting data, we have a single "broadcast" like output bus, the OutBus, that allows simultaneous or individual transmission. Several arbitration techniques are available to guarantee that only one PN is using the OutBus at a time. When simultaneous output from two or more PN's, the OutBus performs a logical AND of the various PN's outputs. This lets CNAPS perform global logic operations across all PN's.

The next issue concerned how to support local inter-PN communication, where PN's in a small neighborhood share data with each other. Over the years, several excellent IP architectures have shown great flexibility and speed in their targeted applications [6],[7]. Probably the best example is Martin Marietta's GAPP series [8], which uses a 2-D array of bit-serial processors.

The GAPP and its brethren are 2-D architectures, designed to look like the data structures they compute over. Since images are 2-D, a 2-D grid structure lets the IP programmer assign one processor to each pixel, or to a square or rectangular group of pixels. However, as we progress to higher-density VLSI implementations, 2-D structures start to become unwieldy to implement. Consequently, there have been some 1-D IP architectures [9],[10] that have been proposed through the years. Fig. 2 shows some examples of 1-D and 2-D arrays. We found through our analysis that in our IP and PR applications, we needed only a 1-D structure. Since 1-D is cheaper and our goal was build an effective and inexpensive image processing architecture, we came to the conclusion that for CNAPS and our target applications, the 1-D and 2-D architectures with identical processors (same precision, storage, function, and speed) offered roughly the same performance, and the 1-D case was cheaper to build because it used less silicon area and off-chip bandwidth.

I) ***Scalability:*** The first problem with 2-D architectures is ***scalability.*** In theory, 2-D grid structures are infinitely scaleable; in practice, today's VLSI technology imposes several limits [11], the most serious being off-chip I/O bandwidth, which is expensive in systems cost (pads, traces, packages) and in power dissipation (a growing problem in higher-frequency computing). Consequently, scaling up the number of processors per chip makes the chip and system more expensive.

Several factors influence pin requirements. A 2-D chip with an N x N array requires 4 x N x ***W*** pins, where ***W*** is the word width A 1-D chip requires only 2W pins. In 2-D, adding processors (N) significantly increases the number of pins; in I-D, adding processors leaves the pin count unchanged. While it is sometimes possible to put the entire 2-D grid on one chip, our target applications required more processors than one die can hold. In spite of the high level of VLSI integration, future applications are likely to continue to use multichip processor arrays for some time.

As on-chip clock frequencies get higher and higher-and the difference between on-chip and off-chip frequencies gets larger and larger-the word width grows to avoid becoming a bottleneck. Increasing W adds yet more pins. It is easy to see that 2-D arrays (4 x N x ***W)*** exhaust the pin limits imposed by packaging technology long before 1-D arrays (2W) do. Therefore, as VLSI densities continue to scale, multichip 2-D grids reach bandwidth limits before multichip 1-D grids do.

A 2-D processor grid has on-chip costs as well. Each processor must communicate with four (or eight in some cases) other processors, not with two, forcing additional internal silicon area to be dedicated to busing and control. In addition, a processor typically executes one operation at a time. Therefore, each processor normally can transfer data to only one neighbor per clock cycle. Each I-D processor has essentially one external bus, which it can fully employ at each clock. Each 2-D processor has two buses, so the 2-D array generally underutilizes half of its potential extra bandwidth.

2) ***Redundancy:*** As discussed above, another cost issue was our intended use of redundant processors. As silicon area gets larger, the probability of a defect occurring on a chip gets rapidly larger. While the latest generation of process technology has the lowest defect densities yet, defects are still present. Redundancy improves silicon yields by printing extra copies of certain circuits, then turning off flawed copies in favor of good ones. Redundancy can rescue many dice that would be wasted otherwise [12]. Redundancy has its own cost and ultimately must "pay its way."

It is possible-but difficult and expensive-to employ redundancy in a 2-D array. A 2-D design introduces geometrical constraints that make it difficult to specify a redundancy mechanism that pays for itself by improving yield enough to justify the extra silicon cost. With a 1-D design it is possible to easily bypass any number of processors (within electrical limits). A 1-D array can easily use redundancy to improve yield, decreasing the cost per finished chip.

3) ***I/O Bandwidth:*** A 1-D system has much less I/O bandwidth than a 2-D system (2W pins versus 4 x N x ***W*** pins). In a 2-D design many of those pins are used to

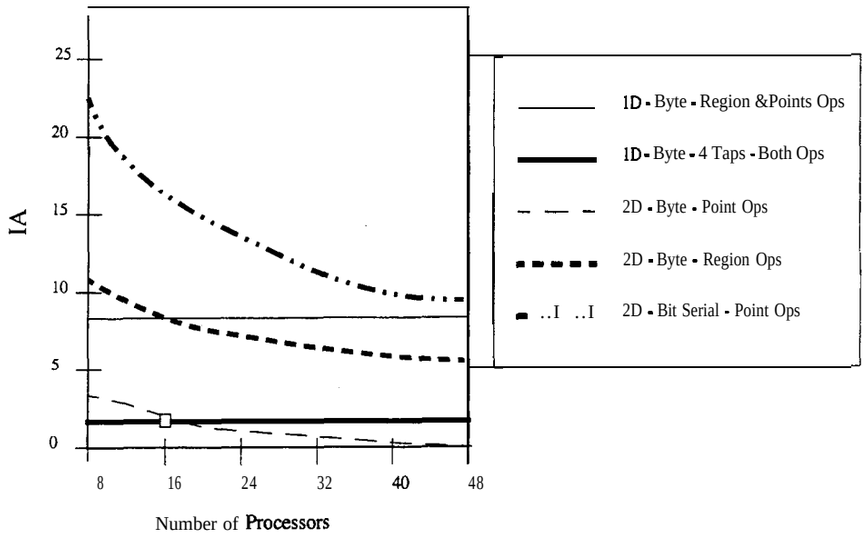


Fig. 3. Data movement rate versus the number of PN's and connection topology.

connect to other chips and not for system I/O. Many 2-D arrays consequently receive input on only one side of the array, since their processors can typically transmit or receive from only one direction per clock.

After the data is in the array, when executing algorithms over 2-D data arrays, a I-D processor array can be programmed to keep all the processors busy doing useful computation for all clocks for most typical IP tasks as we shall demonstrate. Therefore, two chips-one with 1-D interconnect, the other with 2-D interconnect-that have equal numbers of identical processors also have the same computing performance.

4) *Image Architecture Analysis: The* performance of a SIMD image processing architecture, IA, can be expressed as a simple relationship between the time required to process a number of image pixels, *P*, with a number of processors, N, taking into account the time to load and unload pixel data, IO_S (N), the data transfer time required during pixel computation, IO_C, and the time to load the processor instructions, C:

$$IA = P/N + IO_S(N) + IO_C + C.$$

By analyzing the above equation we can see that when the number of processors is much less than the number of image pixels, the performance of a 1-D array approaches that of a 2-D mesh. In theory, a 2-D mesh can outperform a 1-D array even when the number of processors is small. In practice, the opposite can be true. In addition, a cost analysis strongly suggested that for CNAPS, a 1-D array would be superior to a 2-D mesh.

Our analysis begins by noticing that the topology of the array effects only the l/O terms of the above equation. Since a 1-D array is simply an unfolded variant of a 2-D array, the number of processors is unchanged, leaving execution times unaffected. Similarly, the amount of time to load instructions is the same for both interconnection schemes, assuming a global instruction bus for both 1-D and 2-D architectures. Now to examine the IO_S and IO_C terms.

In the 1-D case, two different interconnect schemes will be evaluated: a single input/output bus, and a structure with a small number of taps onto a single input/output bus. A full per processor input/output bus interconnect can be compared only to a similar 2-D interconnect, creating an equivalent performance (IO_S) and cost, and providing no information for our study.

Point-based image processing algorithms, such as adding a constant to every image pixel, are used to study the worst case IO_S time. Point-based algorithms force the interconnect to load and unload every image pixel. The second measure, IO_C, represents the efficiency of algorithms that use the processor interconnect topology. These are neighborhood-based algorithms such as lowpass filtering by convolving with a Gaussian kernel. Here, the interconnect topology increases efficiency because the data required for the computation are available in the local neighborhood of the processor.

Fig. 3 shows the results of our analysis, plotting data for 1-D and 2-D interconnects. For both point-based and neighborhood-based algorithms, we assumed a 2K x 2K pixel image (4 Mbytes). We used a small neighborhood of 3 x 3 pixels in the neighborhood-based analysis. The vertical axis displays the overall performance (IA).

The architectural assumptions for the CNAPS array are a byte wide global broadcast input bus as well as a sequentially arbitrated output bus, and sufficient local memory to hold all pixels required for a neighborhood-based operation. No cost is associated with loading neighborhood data on any processor because this operation is accomplished by using the broadcast bus. Therefore, the system input and output for both point- and neighborhood-based algorithms can be expressed as

$$IO_S = 2P.$$

We also examined the case where the 1-D array has multiple taps on the I/O bus. Such taps enable a concurrent I/O capability that improves the overall IO performance as a function of the number of taps. The 1-D multitap I/O can be compared to a 2-D mesh because it is simply a hybrid I-D case, not a variant of a 2-D mesh. The 1-D hybrid lacks the neighborhood topology of a 2-D array, but overcomes this omission with the judicious use of local memory. Both point-based and neighborhood-based algorithms use the 1-D multitap array in the same way as the single I/O bus. Performance depends on the number of taps; we analyzed a 1-D array with four taps

$$IO_S = \frac{2P}{4}.$$

Point-based algorithms do not benefit from the mesh connected topology of a 2-D array, but inpur and output must still be performed. The number of point-based operations for a 2-D mesh where input and output occur along one side of the array can be written as

$$IO_S = 2\frac{P}{\sqrt{N}}.$$

Neighborhood-based algorithms take advantage of the 2-D interconnect topology to improve IO_C in two ways. First, multiple data paths improve input and output. Second, they also improve communications efficiency because nearby processors can rapidly pass data required for the computation. The following equation expresses this kind of communication efficiency and introduces a new term, R, which is the extent of the local neighborhood required for processing [13]

$$IO_C = [R^2 - 1 + \sqrt{N} + 2[R-1]]\frac{P}{N}.$$

The last case examined, the bit-serial implementation of the 2-D mesh, shows the power of bit parallelism. Widening the bus to a one byte (8-b) width improves IO_S by a factor of eight. Only the IO_S performance of point-based algorithms is plotted. The IO_C can be easily computed as eight times the I/O value of the 2-D neighborhood case. This value was not plotted because the performance along most of the curve was so slow that it was outside the scale of the graph. The difference in performance between the bit-serial bus and byte wide bus is expressed as

$$IO_S = 8\left[2\left[\frac{P}{\sqrt{N}}\right]\right]$$

Referring to Fig. 3, when the number of processors is small, I-D arrays with a byte-wide input bus outperform bit-serial 2-D mesh implementations for both point- and neighborhood-based algorithms. Since bit-serial implementations of the interconnect have so far been the typical choice for mesh connected architectures, our data show that the number of processors must exceed 48 to compete with I-D byte wide arrays.

Although point operations show improvement for small numbers of processors, a 1-D array is more efficient than a 2-D array for neighborhood-based operations. Adding more taps to a 1-D array further improves its efficiency. For example, a 16-processor 1-D system with four taps is equivalent to a 16-processor 2-D mesh where data is input along one side, as marked with a square on Fig. 3.

Notice that a 1-D multitap array is consistently more efficient for neighborhood-based algorithms for configurations up to at least 48 processors. This advantage results from the 1 -D array having a global broadcast bus and using local memory. When the number of processors exceeds 16, the multiple-tap broadcast[8] architecture is roughly equivalent to 2-D, showing better performance in the neighborhood-based operations and slightly worse performance in the point-based operations. Without taking the cost of extra pins into account, it is easy to argue for the I-D architecture in terms of communications performance. When the number of processors is small and multiple taps can be employed, a I-D CNAPS array could offer equal-or in some cases better-performance than a 2-D bit-serial mesh.

To see why better performance is possible, consider the situation where a column of pixels are stored in each PN, and it processes the column one pixel at a time. Horizontal data transfer is required to obtain information from neighboring columns on neighboring PN's. When fetching data in the vertical dimension, however, the access is essentially independent of the number of elements away, since it exists in the memory of the PN itself.

Using pin cost as a metric, we can strengthen our argument by weighing our (IO_S) performance equations by the cost of additional pins. Let us look at the equation for IO_S for point-based operations using a 2-D mesh. We normalize this equation by the cost of the additional interconnect. Since we are inputting data via one side of the array, the 2-D mesh uses $\sqrt{N}$ more pins than a 1-D single bus array. We can state the meaning of this equation as input/output cycles per pin

$$IO_S = 2\frac{P}{\sqrt{N}}\sqrt{N} \text{ or } IO_S = 2P.$$

Upon simplification the performance per pin is exactly the I/O performance of the 1-D case per pin. This result is not really surprising, but drives home the cost-performance argument. Similar normalization can be performed for the remaining I/O equations-the result is the same.

Based on this analysis we concluded that a 1-D CNAPS PN array was sufficient for our application space. We next present a general discussion of the resulting architecture. We find it interesting that Jonker [14] recommended many of our architectural decisions independently. His analysis corroborates our choices of 1-D interconnect, byte, and wordwide buses and arithmetic units, and large local memory for subimage caching.

'Having too many taps makes it difficult to leverage redundancy effectively even for a 1-D architecture. In CNAPS, we have found that using 16 PN's between taps adds reasonable IO_S performance, and redundancy within groups of 16 PN's does not impact yield significantly. Also, as the number of taps increases. the inter-PN interconnect layout cost starts to approach that of 2-D.

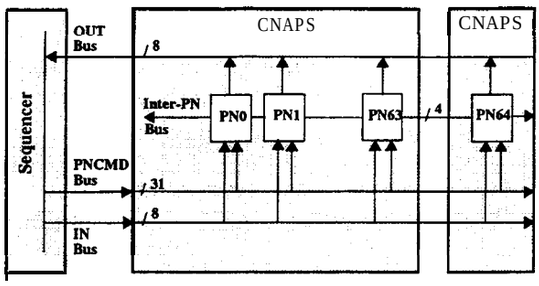


Fig. 4. Overview of the CNAPS architecture.

III. CNAPS Architecture

A. Basic CNAPS Architecture

The CNAPS architecture consists of a 1-D processor array implemented on a custom VLSI chip. Several different processor array chips are available: the 64-Processor Node (PN) CNAPS-1064, the 16-PN CNAPS-1016, and in late 1996 the 32-PN 2032. The CNAPS Sequencer Chip (CSC) governs the execution order of CNAPS instructions, broadcasts input data to the array, and reads results. The sequencer is the external interface to the host computer, called a control processor (CP), which interacts with the sequencer by sending it commands and writing source/destination data pointers. The sequencer in turn signals completion via interrupt of the CP. CNAPS thus operates as a coprocessor to the host CP. Fig. 4 shows an overview of the CNAPS system.

The CNAPS architecture is a single-instruction, multiple-data (SIMD) PN array [15], [16], [17], where all PN's execute the same instruction every clock in synchrony. Its instruction word is 64-b long: 32 b specify the sequencing and I/O control operations and 32 b control the PN operations. These 64-b instructions are stored externally in SRAM, with half the memory feeding the sequencer and half being broadcast to all PN's. The sequencer provides the address to the program memory, since it controls the sequencing.

As is common with SIMD architectures, the PN's can conditionally execute instructions based on the results of internal variables. Likewise, the sequencer can signal all the PN's to ignore an instruction, essentially causing a spin of the PN array, for example when the sequencer is waiting to read input data or to write output data.

The sequencer can drive multiple PN array chips. The size of the array has no architectural limit. The current electrical limit is eight chips-512 PN's with the 64-PN chips, 256 PN's with the 32-PN chips, or 128 PN's with the 16-PN chips. Chip boundaries are invisible to the programmer, who is aware only of the total number of PN's in the system. So, for example, the programmer sees no difference between one 64 PN chip, two 32 PN chips, or four 16 PN chips.

As can be seen in Fig. 4, the PN's are connected into a 1-D array with global broadcast from the sequencer to all PN's (the InBus), and a global output bus (OutBus) from all PN's to the sequencer. Each PN is connected to its nearest neighbor via a 1-D inter-PN bus. The architecture offers several arbitration techniques for the OutBus, the most common being sequential transmission. ((PNn, PNn + 1, etc.) In addition to data transfer, the inter-PN bus is also used to signal the next PN during sequential arbitration. If multiple PN's transmit at once, the array performs a global AND of all PN outputs (the default transmission is all ones). The OutBus goes into the sequencer, which can wrap the data back around to the InBus and/or write it to memory via a DMA channel (StdOut). The sequencer likewise can read data from memory (StdIn) and put it on the InBus for broadcast to the PN array.

In general, we are pleased with the architecture, but it has several weaknesses that we plan to fix in the next generation. These include the 8-b input/output, which should be 16-b, since we have found that many applications need greater I/O bandwidth. Likewise, we need more on-chip support for multiple taps (e.g., buffering). Within the PN's, we found that we need a more powerful, data-driven, barrel shifter. Finally, the next generation will have a simpler, register-register RISC-like instruction set, making it easier for the compiler to generate code. We have found that compiling code for SIMD machines is not that difficult when the appropriate extensions are added to the C language.[9] The biggest problems in compiling C code for CNAPS are that the PN does not easily support the data types required by C and by the complex, microcode-like instruction set, leading to less efficient code generation.

B. Processor Node Architecture

Each PN is a simple 16-b compute engine, similar to a digital signal processor (DSP), as shown in Fig. 5. Each PN has the following internal components, which are connected by two 16-b internal buses:

- 9-b x 16-b multiplier;
- 32-b adder;
- 32 element register file (16-b);
- 16-b shifter/logic operation unit;
- 12-b memory address adder; and
- 4096 byte memory, accessible as either 8- or 16-b values.

The instruction encoding is reasonably horizontal, so it allows several of the these units to operate at the same time. For example, in a typical vector inner product ("multiply-accumulate") operation, a data element is input and sent to one half of the multiplier while the memory simultaneously reads the corresponding element of the multiplicand vector and sends it to the other half of the multiplier. At the same time, the PN computes the address for the next memory location to be accessed and accumulates the result of the previous clock's multiply with the ongoing sum. Likewise, there is hardware for quickly finding which PN has the maximum value of a particular computation.

[9]Adaptive Solutions has extended the C language by adoption of the domain structure. Domains are arrayed structures that allow the programmer to specify the data parallelism of the problem.

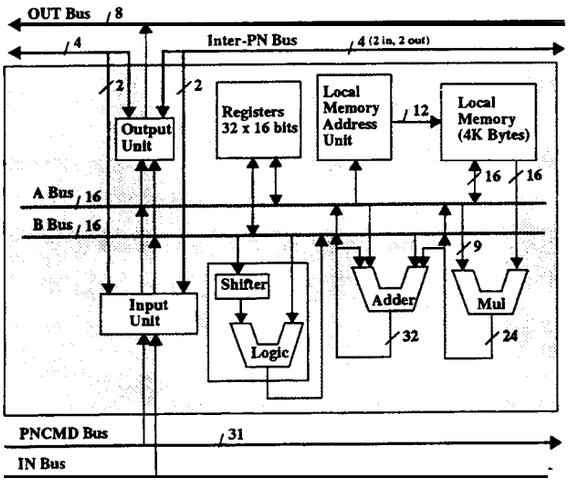


Fig. 5. Internal architecture of a CNAPS PN.

C. Chip Specifications

The 1064 and *1016 are* implemented in 0.8 μm SRAM CMOS with two levels of metal and two levels of polysilicon. The 1064 is 26 mm (roughly 1 in) on a side and contains 14 million transistors. For redundancy, 80 PN's are fabricated on the die in an 8 x 10 array, giving us 16 spare PN's. Each PN has 4 KB of SRAM. For the 1064, that makes a total 256 KB or 2 Mb. Each SRAM cell is a 4 transistor (poly load pull-up), 47 μm^2 cell. The 1064 operates at 25 MHz and offers 3.2 billion arithmetic operations per second. Power dissipation is roughly 6 W worst-case for the 1064. We have found that the redundancy works extremely well and significantly enhances our yield.

The 1016 is a quarter size version of the 1064, 20 PN's are fabricated, 16 need to function for the die to be good. The 1016 allows us to have more parallel taps and to build lower price configurations.

The sequencer chip or CSC is implemented in gate array technology and performs instruction sequencing and I/O processing by feeding and reading the CNAPS array InBus and OutBus.

The newer version of CNAPS, the 2032, is implemented in 0.6 μm SRAM CMOS. The 2032 is roughly 13 mm on a side and operates at 40 MHz. The 2032 requires a different, 40 MHz sequencer, the C2. Like the 1016 and 1064, the 2032 has 4KB of SRAM per PN. The 2032 offers 2.56 billion arithmetic operations per second. Power dissipation is roughly 2 W.

D. Board Configurations

The typical CNAPS board configuration, shown in Fig. 6, consists of a CSC sequencer chip; one to eight 1016 or 1064 PN array chips; local on-board SRAM for the CNAPS program memory; local on-board DRAM for buffering data, constants, or both; plus a bus interface. The newer C2 can drive one to eight 2032 PN array chips.

The CSC state, on-board SRAM, and on-board DRAM are visible to the bus address space. The CP (host) then

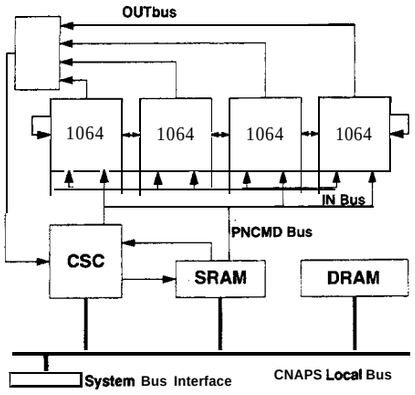


Fig. 6. A typical CNAPS system configuration.

treats the CSC as a coprocessor. It writes system data and control information into the local memory and programs into the SRAM, then sets up the CSC's internal registers and the program's starting address. Finally, the CP writes a "start" command into the CSC command register to begin execution. When it has completed the task or tasks allocated it by the CP, the CSC halts and issues an interrupt to the CP.

E. Convolution Implementation

Now that we have presented the basic structure of the CNAPS architecture, we are ready to discuss two simple, generic operations and their implementations on CNAPS. After describing them, we discuss applications that employ variations on the two basic operations.

The first operation is the spatial filtering of a 2-D image. This filtering is approximate in that it is performed by doing a 2-D spatial convolution over a limited subset of the surrounding (nearest-neighbor) pixels for each pixel. The particular values of the coefficients are unimportant, although some filters take advantage of various types of symmetry to reduce operations. Such optimizations are not discussed here. In general, convolution is described by the following equation

$$f(x,y)*g(x,y) = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,\ n)g(x-m,\ y-n)$$

where $f(\,)$ is the image, g( ) is the filter, and M and N, are the two maximum dimensions of the image.

Let assume that we have a 512 x 512 image with 8-b per pixel, and that we are computing the convolution over a 7 x 7(M = 7, N = 7) mask centered on the pixel. Similarly assume that we have a 512-PN array, which lets us allocate a column of the image to each PN. Although the details are beyond the scope of this paper, the PN architecture has a mechanism which allows each PN to selectively input part of the data stream flowing over the InBus and write that to its local memory. The image is input a scanline (row) at a time. As the row is broadcast, each PN writes only the bytes corresponding to its column. After 512 rows have been scanned, each PN's local memory contains the

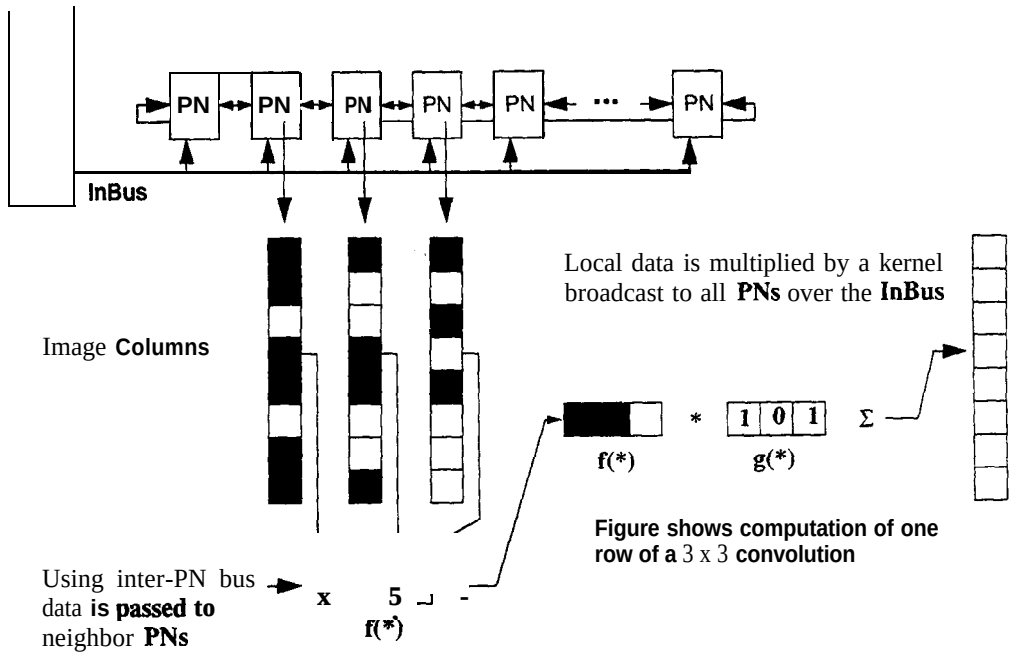


Fig. 7. PN memory allocation for convolution operation.

512-element column vector of its column. This operation takes $\frac{512}{T}$ clocks, where T is the number of taps. A 512 PN system using eight 64-PN chips has eight taps. The CNAPS architecture implements multiple taps only at chip boundaries.

The convolution process begins after the data is in PN memory. The PN's start with the first pixel in the column and computes its new value. This process continues down the column, in parallel for all columns, until all 512 pixels have been computed. The values of the old and new columns easily fit within each PN's 4 KB local memory.

To avoid complicating the description with boundary issues, assume that we are computing the value for a pixel k somewhere in the middle of a PN's column vector. Fig. 7 shows the column allocation for a 3 x 3 convolution. The PN's memory has a buffer containing successive rows of the neighboring pixels' values for the last convolution computation. To compute the next pixel's new value, we need to obtain from the neighboring PN's only the values for the bottom row, since the data of the other six rows are already in the PN's buffer from the previous pixel's computation.

To obtain the data, each PN reads from its local memory the column element $k + 3$ and puts it in the inter-PN transfer register. All PN's then shift the value to the left one PN-that is, each PN gets a copy of the value in the inter-PN register of the PN to its immediate right, $PN + 1$. All PN's write this value to their buffers. The value is then shifted again to the left, to $PN - 1$, and written to memory. This iteration is performed three times to the left. Each PN then reloads the pixel at $k + 3$, and the process is repeated three times to the right.

When all transfers are complete, each PN has a complete copy of all the pixel values under the mask for pixel k in its buffer. At this point, the kernel values are broadcast over the InBus to all the PN's simultaneously, and an inner-product multiply-accumulate is performed between the kernel values, $g(x, y)$, and the buffer values, $f(x, y)$. The kernel values are generally stored in the PN's (one value in each PN) and broadcast to all PN's. The result of the inner product is the new value of pixel k, which (after some scaling) is written into the column of 'hew" pixel values.

The convolution is complete when all pixels in the column have been processed. Additional operations can then be performed on the image, and when all operations are complete, the rows can be scanned out one PN (pixel) at a time per tap back to host memory.

This basic model has several possible variations. For example, in a color image (assuming a three-byte RGB implementation), each color plane would have its own column and convolutions would be performed on each color plane independently. Input and output is slightly more complex, with each PN picking up three bytes at a time during the input scan, and transmitting three bytes at a time during output.

Another variation is to allocate multiple columns to each PN. In the example above, going to 128 total PN's requires each PN to store four columns. In this situation, the column computation is multiplexed, taking four times longer. The movement of neighboring column data also becomes much more complex, since the amount of data moved depends on how many columns the PN is emulating, the size of the kernel, the column currently being computed. However, putting more columns on a PN also reduces the number of inter-PN transfers. Different but similar neighborhood functions, such as morphology, with limited masks could also be performed using the basic data allocation and data processing techniques described above for convolution.

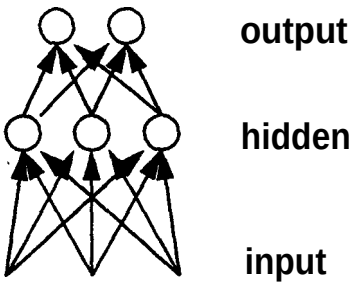


Fig. 8. A three-layer multilayer perceptron (MLP) classifier network.

F. Classifier Implementation

Many IP problems involve back-end processing, such as region or object classification. The ability to efficiently execute classifier algorithms is consequently important in IP applications. This section presents a backpropagation neural network classification algorithm and its mapping to a 1-D PN array. This classifier is based on the most popular and perhaps most powerful neural network technique, the backpropagation of error algorithm [18], [19]. Detail on how the algorithm operates and its derivation is beyond the scope of this paper, but the references include excellent tutorials. Although the number of layers in a network is arbitrary, in this example we assume a simple three-layer network, as shown in Fig. 8. The first layer is consists of just inputs; the hidden and output layers are computational layers.

Backpropagation belongs to a family of algorithms where the coefficients of the network are trained by ***supervision,*** meaning that during training each input is tagged with the known desired output. The network learns to approximate the output when it sees the corresponding input. Backpropagation networks are powerful nonlinear classifiers that can approximate complex nonlinear functions and generalize to noisy or novel input. We will describe both the feedforward and training modes of operation.

Most applications use only feedforward execution. In feedforward mode, the nodes of each layer of the network compute the following function

$$o_{pj} = f\left(\sum_i w_{ji} o_{pi}\right)$$

where o_{pj} is the node output, w_{ji} is the weight value from node i to node j, $f()$ is a smooth, asymptotic, nonlinear function, $f(x) = 1/1 + e^{-x}$, called a sigmoid, and p is the particular training vector being presented to the network.

For ***feedforward*** operation, the inputs are first presented to the network, which computes the above equation for the hidden layer. The hidden-layer outputs are then used as inputs to compute the inner products for the output layer. One can easily see how this equation maps to a 1-D PN, one node per PN.

In computing the inner product, the weight matrix, ***W,*** is assigned a row at a time to the PN to which the node corresponding to that row, or weight vector, is assigned. All that is different here is the multiple-layer aspect of the network, and the nonlinear function. The various layers are time-multiplexed, so the first hidden-layer nodes are allocated to the PN array, one node per PN. Each PN computes the node output for the first hidden-layer node assigned to it.

After all outputs are computed, these outputs are broadcast back to the PN array, while each PN computes the inner product of the output node (or second hidden-layer node in the case where there is more than one hidden layer) assigned to it. If the number of PN's exceeds the number of nodes in the layer, some PN's compute a useless result, which ultimately is not transmitted. If the number of nodes exceeds the number of PN's, the layer is folded, where some PN's compute the outputs for more than one node of that layer. The folded case requires more than one pass of the data to complete all computations on that layer, and more memory per PN for weight storage.

If we have a network with three layers and, say, 256 inputs, 120 hidden nodes, and 20 outputs, then it has 33 120 connections with one weight per connection." A single-processor DSP requires at least 33 120 clocks to emulate one pass through the network, ignoring computation of the nonlinear function. A byte-parallel 1-D PN array, on the other hand, requires 256 clocks for the first layer and 120 clocks for the output layer computation, for a total of 396 clocks. The hidden layer computation uses 120 PN's; the output-layer computation, only 20 PN's. In many limited-precision implementations, the sigmoid function can be a simple table lookup.

The final operation to be discussed is back-propagation learning mode, where the network is presented with an input and a desired output. The network then incrementally adjusts the weights to create an approximation to the function described by the entire training set. When training with the backpropagation model, a feedforward pass is done to get the output for each hidden and output node. Then an ***error delta*** is computed for each output node as follows

$$\delta_{pi} = (t_{pj} - o_{pj}) o_{pj} (1 - o_{pj}).$$

The first term $(t_{pj} - o_{pj})$ is the error term, the difference between the target (desired) output vector and the actual output vector. The next two terms, $o_{pj}(1 - o_{pj})$, constitute the derivative (slope) of the nonlinear sigmoid function at the activation value. Once the output-layer error deltas are known, the error deltas for the last hidden layer can be computed as

$$\delta_{pj_i} = o_{pj}(1 - o_{pj}) \sum \delta_{pi} w_{ij}.$$

The first term is the sigmoid slope again. The second term is an inner product of the output layer error deltas and a ***column*** of the weight matrix. If the network has multiple hidden layers, the error is backpropagated one layer at a time (from output to input) using the same basic formula. After all nodes have computed their error delta values,

[10] Most backpropagation networks also have a bias term for each hidden and output node. This term can be considered as a weight from a constant valued input. The bias terms are ignored here for simplicity.

the weight vector used by that node during feedforward computation can be updated according to"

$$\Delta w_{ji}(n+1) = \eta(\delta_{pj} o_{pj}) + \alpha \Delta w_{ji}(n).$$

When mapping this complex set of computations to a 1-D PN array with limited inter-PN interconnect, the output-layer error delta computation and the weight update formulas are local computations that can easily be performed by the PN's on the nodes they emulate using the data allocation discussed above for feedforward operation. The hidden-layer error delta computation has a problem, however: it sums over columns, but the weight matrix is stored in the PN's by rows. Doing an inner product across PN's for both weights and error deltas would essentially require $O(n^2)$ operations and could not be done in parallel.

The solution is to transpose the matrix, where now each column of the weight matrix (row of the transposed matrix) is allocated to a PN, then do row-wise summation. Over simple broadcast interconnect, the transpose operation is, unfortunately, also $O(n^2)$. While using a full-crossbar inter-PN structure, which allows every PN to talk to every other PN, would solve the transpose problem, it would also greatly increase cost. CNAPS backpropagation consequently stores two versions of the matrix, a regular version for feedforward operation, and a transpose for error backpropagation. This approach doubles the per PN storage requirements of the network and requires two writes when updating memory, but it is a simple solution that operates in linear time.

A well-optimized learning routine takes about five times longer than the feedforward operation by itself. When performing the NetTalk problem [20] which consists of a network of 209 input nodes, (usually) 64 hidden nodes, and 26 output nodes, the entire network fits onto a single 64-PN CNAPS chip and computes at a learning rate of about 160 million connection updates per second. A NetTalk training run of 70k vectors takes about 7 s, as compared to about 15 min on a Sun Microsystems SPARC10. A full 5 12-PN CNAPS system (eight 64-PN chips) computes at over one billion connection updates per second if all PN's are used.

In the next few sections, we briefly describe three applications of the CNAPS architecture. Since most use simple variations of the two algorithms just described, we will go into less detail on the specific implementation of each application.

IV. Applications

A. Desktop Image Processing-Adobe Photoshop Acceleration

Prepress is a common desktop application of IP, and consists of a set of activities that occur between the development of an electronic document and its preparation for printing. At this stage, the entire document can be considered as a complex image. The most complex and difficult-to-prepare parts of that image are photographs, figures, and other kinds of complex subimages. The most popular program for manipulating photographs is Adobe Photoshop@.

The images that most magazines, advertising brochures, and similar publications contain are dense (4k x 4k pixels is not uncommon). Relatively simple IP functions, such as a 2-D spatial filter, can be slow, even on expensive high-end machines. Because the cost-performance of CNAPS greatly exceeds that of traditional desktop systems-and because these kinds of imaging problems can leverage most of that performance-Adaptive Solutions decided to build a Photoshop accelerator card based on CNAPS. One match between the architecture and application is the precision of the images, typically 8-b per pixel per color, which agrees with CNAPS's moderate- to low-precision. The small, simple CNAPS PN arrays and the use of redundancy allows us to offer such performance at an affordable price.

Photoshop offers several filters, such as Unsharp Mask (where a blurred, low pass spatial filtered version of the image is subtracted from the original, sharpening the image by enhancing the higher-frequency components). These filters are implemented more-or-less like the convolution discussed above. Since Photoshop updates the displayed version of the image in the host memory after each operation, the CNAPS card reads an image, a single operation is performed, then the image is written back to main memory. Consequently, a two-level tiled version of the convolution algorithm is used. The image is broken up into large tiles, roughly of a size to fit into the entire PN array. CNAPS processes one of these tiles at a time, sequentially reading a tile, then writing an updated version of the tile back to main memory. The fact that the input tile needs to be slightly larger than the output version to support convolution overlap makes this process somewhat more complex.

When a large tile is written to the PN array, which is done a scan line at time, each PN automatically grabs the subtile that contains the data it needs to compute its pixel's outputs. Photoshop generally represents data in RGB format, which has an 8-b byte for each of three colors. After the subtile is in a PN's memory, however, the actual filter operations are performed on one color plane at a time which permits using the address generation adder to create the appropriate series of offsets into PN memory.

In general, using the 1016 chip set the performance improvements over a PowerMac (PCI bus based) range from factors of 3-5x. For example, a 10 pixel radius convolution filter over a 24-b full color 18 MB image using a 64 PN array is about 8 s, versus 56 s on a PowerMac 8100/100. This total accelerated performance is good, but is below what one would expect from the computational advantages of a 64-PN CNAPS array over a single PowerPC. This difference is due to the need to move an image to the CNAPS card over the PCI bus and back again for each operation this is an example of Amdahl's law placing a fundamental limit on accelerated performance.

"There are numerous weight update formulas; this is one of the most common and generic. The α and η are constants used to control learning rates and stability.

B. Medical Image Classification

An important area of IP and PR is processing and classifying medical images, a field that has significant computing requirements and increasing pressure to decrease costs. Reading and analyzing scanned images-whether MRI scans, optical scans such as Pap smears, or X-rays for suspicious structures such as cancer cells-is a matter of life or death. The process deals with noisy, ambiguous data, and is error prone. R2 Technology in Mountain View, CA, has developed a neural network based classification algorithm for flagging areas of interest in mammograms.

Some of the R2 algorithm is proprietary, but we can give a simplified view of it here to show how the CNAPS 1-D architecture fits the problem. The X-ray images are 4K x 4K pixels, and each pixel is a 12-b value. The basic algorithm consists of performing two sequential convolutions of the image using a 9 x 9 filter. The output of each filter passes through a nonlinear function similar to the sigmoid function used in the back-propagation algorithm discussed above.

As with Photoshop filtering, the basic X-ray image is broken up first into large overlapping tiles. These tiles are read into the PN array. R2 uses the 128-PN PCI card, which constrains the size of the major tiles. Each major tile is then broken into a set of 128 overlapping subtiles. Each PN operates on its own subtile using the data in its memory. For the first convolution, each PN computes the values for an array of pixels that is four units smaller (the overlap area) than the subtile. The final result of the convolution filter is then passed through the nonlinear operator (using table look-up). Then a smaller group of pixels (four units smaller along the edge) are convolved using the outputs of the first pass, and again a nonlinear operation is performed on the output. Finally, the output values of the last convolution for the major tile is output, one scanline at a time to the host memory. Data output is overlapped with the input of the next tile to the PN array.

Because the algorithm employs two successive convolution operations, the overlap between each PN's tile is eight, and there is some redundant computation of first-layer pixels. The tile for which PN is responsible is large enough that the inefficiency of this overlap is minimal.

The PCI board can scan au entire 4K x 4K X-ray for suspicious structures in 14 s, which meets R2's performance requirements.

V. Summary and Conclusions

This paper has described the Adaptive Solutions CNAPS, a 1-D SIMD processor array for IP and PR applications. We discussed the major architecture decisions and motivated these decisions. For interprocessor communication, we have argued that for our target applications a 1-D CNAPS array has greater cost-performance when compared to a 2-D CNAPS array. We showed that if the number of pixels is generally greater than the number of PN's, then a I-D array can be as efficient as a 2-D array in PN utilization. We discussed internal and external I/O cost issues as well as manufacturing issues such as the use of redundant processors, indicating that a 1-D CNAPS scales better and is cheaper to build. We then presented several brief descriptions of some real world applications of a 1-D architecture in image processing and recognition.

We believe that as multimedia applications and the manipulation of images by computers become more widespread, there will be a continual need for inexpensive desktop imaging. We also believe that architectures of the type described here-l-D SIMD arrays with simple, low-precision fixed-point arithmetic and on-chip memory-will provide enough performance improvement to compete favorably against the fastest "super-micros" on the desktop. CNAPS, is one of the first of several chips that are starting to appear which have similar architectures. Other examples include the MPACT from Toshiba and Chromatics, the Media-Engine from MicroUnity, and the Tri-Media from Philips [21]. Even some of the super-micros are beginning to adopt these SIMD techniques internally, i.e., the Multimedia Extensions (MMX) to the Intel Pentium™, the VIS extensions to Sparc™, and the MAX multimedia extensions to the HP PA [22].

References

[1] *G.* A. Baxes, *Digital Image Processing-Principles and Applications. New* York: Wiley, 1994.

[2] Ibid, p. 21.

[3] G. D. Hutcheson and J. D. Hutcheson, '"Technology and economics in the semiconductor industry," *Scientif. Amer.*, pp. 54–62, Jan. 1996.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Palo Alto, CA: Morgan Kaufmann, 1991.

[5] M. J. Quinn, *Parallel Computing: Theory and Practice. New* York: McGraw-Hill, 1994

[6] ——, "The second generation image understanding," *Architecture and Beyond,* pp. 276285, CAMP1993.

[7] P. Kuma, *Parallel Architectures and Algorithms for Image Understanding. New* York: Academic, 1991.

[8] K. Preston, "Sci/industrial image processing: New system benchmark results," *Advanced Imag.*, p. 46, Sept. 1992.

[9] D. Helman and J. JáJá, "Efficient image processing algorithms on the scan line array processor," *IEEE Trans. Patt. Anal. and Mach. Intell., vol. 17*, pp. 47-56, Jan. 1995.

[10] L. A. Schmitt *et al.*, "The AIS-5000 parallel processor," *IEEE Trans. Patt. Anal. and Mach. Intell.*, *vol.* 10, pp. 320–330, May 1988.

[11] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design—A System Perspective.* Reading, MA: Addison-Wesley, 1985.

[12] J. F. McDonald *et al.*, "Yield of wafer-scale interconnections," *VLSI Syst. Des.*, pp. 62-66, Dec. 1986.

[13] V. Cantoni, C. Guerra, and S. Levialdi, "Toward an evaluation of an image processing system," in *Computing Structures for Image Processing*, M. J. B. Duff, Ed. London: Academic, 1983, pp. 43-56.

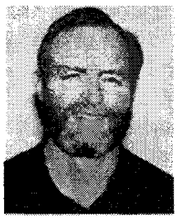
[14] P. P. Jonker, "Why linear arrays are better image processors," in *Proc. 12th IAPR Conf. on Part. Recogn.*, Jerusalem, Israel, vol. 3, ICPR 12; IEEE Computer Soc. Press, 1994, pp. 334-338.

[15] M. Griffin er*al.*, "An 11 million transistor digital neural network execution engine," *IEEE Int. Solid-Stare Circ. Conf.*, *1991*, pp. 180–181.

[16] D. Hammerstrom, W. Henry, and M. Kuhn, "The CNAPS architecture for neural network emulation," in *Parallel Digital Implementations of Neural Networks*, K. W. Przytula and V. K. Prasanna Kumar, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1993, pp. 107–138.

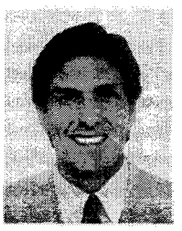
[17] D. Hammerstrom, "A highly parallel digital architecture for neural networks," *VLSI for Artificial Intelligence and Neural Networks*, J. Delgado-Frias and W. Moore, Eds. New York: Plenum, 1991.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations ***by error*** propagation," in ***Parallel Distributed Processing-Vol. I: Foundations,*** D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 3 18–364.
[19] M. Smith, ***Neural Networks for Statistical Modeling. New*** York: Van Nostrand Reinhold, 1993.
[20] T. Sejnowski and C. Rosenberg, "NETtalk: A parallel network that learns to read aloud," Tech Rep. JHU/EECS-86/01, Johns Hopkins Univ. EE and CS Dept., 1986.
[21] "DSP's caught breaking and entering," ***OEM Mug.,*** pp. 52-58, Oct. 1995.
[22] R. B. Lee, "Accelerating multimedia with enhanced microprocessors," ***IEEE Micro,*** pp. 22-32, Apr. 1995.



Dan W. Hammerstrom (Member, IEEE) was born in 1948. He received the B.S. degree (with distinction) from Montana State University, Bozeman, the M.S. degree from Stanford University, Stanford, CA, and the Ph.D. degree from the University of Illinois, Urbana-Champaign, all in electrical engineering, in 1971, 1972, and 1977, respectively.

He is presently an Associate Professor in tbe Computer Science/Engineering Department at the Oregon Graduate Institute and is the founder and Chief Technical Officer of Adaptive Solutions, Inc., where he works on image processing and pattern recognition applications and hardware. From 1980 to 1985 be was with Intel Corp., where he participated in the development and implementation of the iAPX-432, the i960, and the iWarp. From 1977 to 1980, he was an Assistant Professor of Electrical Engineering at Cornell University, Ithaca, NY, and a Computer Systems Design Officer in the U.S. Air Force from 1972 to 1975. He was also a Visiting Researcher at the Royal Institute of Technology in Stockholm, Sweden, in 1979 and 1984. He has been an Associate Editor for the IEEE TRANSACTIONS ON NEURAL NETWORKS, the ***Journal of the International Neural Network*** *Society* (INNS), and the ***International Journal of Neural Networks.*** He has authored over 40 research papers and seven book chapters, and holds seven patents.



Daniel P. Lulich was born in 1953. He received the B.S. degrees in philosophy and psychology from Portland State University, Portland, OR, in 1982. He also received the M.S. degree in computer science and engineering from the Oregon Graduate Institute of Technology, Beaverton, OR.

He is presently an Applications Engineer for Adaptive Solutions Inc., Beaverton, OR. His research interests are image processing, computer vision, neural networks, and the psychophysical study of biological vision systems.

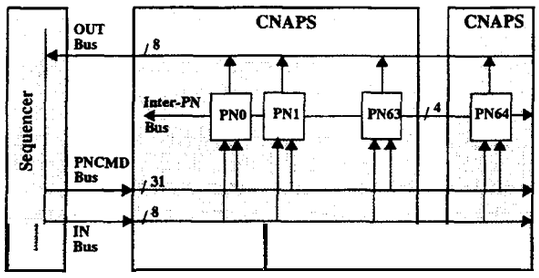


Fig. 4. Overview of the CNAPS architecture.

III. CNAPS Architecture

A. Basic CNAPS Architecture

The CNAPS architecture consists of a I-D processor array implemented on a custom VLSI chip. Several different processor array chips are available: the 64-Processor Node (PN) CNAPS-1064, the 16-PN CNAPS-1016, and in late 1996 the 32-PN 2032. The CNAPS Sequencer Chip (CSC) governs the execution order of CNAPS instructions, broadcasts input data to the array, and reads results. The sequencer is the external interface to the host computer, called a control processor (CP), which interacts with the sequencer by sending it commands and writing source/destination data pointers. The sequencer in turn signals completion via interrupt of the CP. CNAPS thus operates as a coprocessor to the host CP. Fig. 4 shows an overview of the CNAPS system.

The CNAPS architecture is a single-instruction, multiple-data (SIMD) PN array [15], [16], [17], where all PN's execute the same instruction every clock in synchrony. Its instruction word is 64-b long: 32 b specify the sequencing and I/O control operations and 32 b control the PN operations. These 64-b instructions are stored externally in SRAM, with half the memory feeding the sequencer and half being broadcast to all PN's. The sequencer provides the address to the program memory, since it controls the sequencing.

As is common with SIMD architectures, the PN's can conditionally execute instructions based on the results of internal variables. Likewise, the sequencer can signal all the PN's to ignore an instruction, essentially causing a spin of the PN array, for example when the sequencer is waiting to read input data or to write output data.

The sequencer can drive multiple PN array chips. The size of the array has no architectural limit. The current electrical limit is eight chips-512 PN's with the 64-PN chips, 256 PN's with the 32-PN chips, or 128 PN's with the 16-PN chips. Chip boundaries are invisible to the programmer, who is aware only of the total number of PN's in the ***system. So,*** for example, the programmer sees no difference between one 64 PN chip, two 32 PN chips, or four 16 PN chips.

As can be seen in Fig. 4, the PN's are connected into a 1-D array with global broadcast from the sequencer to all PN's (the InBus), and a global output bus (OutBus) from all PN's to the sequencer. Each PN is connected to its nearest neighbor via a 1-D inter-PN bus. The architecture offers several arbitration techniques for the OutBus, the most common being sequential transmission. ($PNn, PNn + 1$, etc.) In addition to data transfer, the inter-PN bus is also used to signal the next PN during sequential arbitration. If multiple PN's transmit at once, the array performs a global AND of all PN outputs (the default transmission is all ones). The OutBus goes into the sequencer, which can wrap the data back around to the InBus and/or write it to memory via a DMA channel (StdOut). The sequencer likewise can read data from memory (StdIn) and put it on the InBus for broadcast to the PN array.

In general, we are pleased with the architecture, but it has several weaknesses that we plan to fix in the next generation. These include the 8-b input/output, which should be 16-b, since we have found that many applications need greater I/O bandwidth. Likewise, we need more on-chip support for multiple taps (e.g., buffering). Within the PN's, we found that we need a more powerful, data-driven, barrel shifter. Finally, the next generation will have a simpler, register-register RISC-like instruction set, making it easier for the compiler to generate code. We have found that compiling code for SIMD machines is not that difficult when the appropriate extensions are added to the C language.[9] The biggest problems in compiling C code for CNAPS are that the PN does not easily support the data types required by C and by the complex, microcode-like instruction set, leading to less efficient code generation.

B. Processor Node Architecture

Each PN is a simple 16-b compute engine, similar to a digital signal processor (DSP), as shown in Fig. 5. Each PN has the following internal components, which are connected by two 16-b internal buses:

- 9-b x 16-b multiplier;
- 32-b adder;
- 32 element register file (16-b);
- 16-b shifter/logic operation unit;
- 12-b memory address adder; and
- 4096 byte memory, accessible as either 8- or 16-b values.

The instruction encoding is reasonably horizontal, so it allows several of the these units to operate at the same time. For example, in a typical vector inner product ("multiply-accumulate") operation, a data element is input and sent to one half of the multiplier while the memory simultaneously reads the corresponding element of the multiplicand vector and sends it to the other half of the multiplier. At the same time, the PN computes the address for the next memory location to be accessed and accumulates the result of the previous clock's multiply with the ongoing sum. Likewise, there is hardware for quickly finding which PN has the maximum value of a particular computation.

[9] Adaptive Solutions has extended the C language by adoption of the ***domain*** structure. Domains are arrayed structures that allow the programmer to specify the data parallelism of the problem.