

Lecture #6: Memory Systems – Data distribution

*Professor: S. Lennart Johnsson**TA: Wei Ding*

1 Memory Systems

In our discussion of vector architectures we noticed that having a memory system that allows for two loads and a store concurrently may significantly enhance the performance. Most of today's microprocessors have only a single port to memory. The first supercomputer, the Cray-1, also had only a single channel between the processor and the main memory, while later Cray models have three memory channels for data and instructions.

The two load and one store channel architectures nicely supports three operand instructions such as $x \leftarrow y \times z$, or the `_AXPY` operation $y \leftarrow y + \alpha \times x$ where α is a constant, or instructions with fewer operands. But, even two load and a store channel to memory is not sufficient for certain operations, such as $y \leftarrow y + x \times z$, which in vectorized form is common for band matrix operations, as we will see later.

If the memory cycle time is slower than the processor cycle time, as is indeed the case today for most commercial computer systems, then the demands on the number of concurrent requests to be served by the memory system increases further. In our description of pipelining we noticed that the memory system for a three operand instruction with a memory cycle time four times longer than the processor cycle time had to support 12 concurrent requests in order to fully support a single processor. In general, for P processors and K operands per instruction, the memory system must support

$$P \times K \times \frac{\text{memory cycle time}}{\text{processor cycle time}}$$

concurrent requests for a design to be balanced with respect to processing power and memory bandwidth. In addition, the memory system must be able to support instruction fetching. For vector architectures, one instruction fetch suffices for each vector instruction. Thus, for a vector instruction set, the bandwidth required for instructions may add in the order of 1% to the bandwidth required for data (one or two words for each instruction covering MVL to 4*MVL data items). For a 10,000 core system with all cores executing three operand instructions and a ratio of memory cycle time to processor cycle time of 10, the memory system must support concurrently the request for 300,000 operands if there is no data reuse. For 64-bit operands, and 48-bit addresses with all addressing being indirect a total of $336 \times 300,000 = 100,800,000$ bits need to be handled concurrently.

The number of concurrent requests that must be supported depends on the ratio between the cycle time for the memory and the processor. For a single processor, a memory cycle time a

few times shorter (K times to be precise) would suffice for a balanced design, while for large scale parallelism the memory cycle time would have to be several orders of magnitude shorter. Today, memory and processors are designed in the same technology, and the difference in cycle time is due to architectural differences between processing logic and memory logic, as well as acceptable power dissipation since there is a power speed tradeoff. One way of measuring the properties (quality) of a design is to compare the cycle time with the switching time of the devices in the technology being used. For a microprocessor, the ratio between the processor cycle time and the device switching time may be about 100 - 200 for an aggressive design, while for a memory chip it is more typical with a ratio of about 1,000. For instance, with transistors with a gate length of 45 nm the delay is about 1 - 2 ps, while processor clock rates are about 2 - 3 GHz or a clock period of about 300 - 500 ps. DRAMs in a corresponding technology may operate at 200 MHz. The dominating technology for both processors and memory today is Metal Oxide Semiconductor (MOS) silicon. In the next section we will briefly discuss MOS technology, in particular with respect to memories.

1.1 MOS Memories

MOS technologies are “charge transfer” technologies. The switching of transistors is controlled by the charge on a gate. Memory cells for storing one bit can be designed in several different ways depending upon the design objective [16, 1, 17]. A very area efficient design makes use of a single transistor and a capacitor. Such memory cell designs are used in Dynamic Random Access Memories, DRAM. Commonly used DRAM chips today have a density of either 1 or 2 Gbits, but 4 Gbit memory chips are also available. The chips are typically mounted as 1, 2 or 4 GB DIMMs. The density of DRAMs has been growing fourfold every three years. One of the drawbacks of DRAMs is that the memory cell does not keep its state very long. The charge is leaking away. Every 1 - 2 msec, the state must be restored through a *refresh* of the entire memory. A refresh implies that each memory cell is read and written again. In fact, DRAM memory cell designs are such that reading the state of a cell destroys it, and the cell must be rewritten after every read operation. This property has led to the notions of *access time* and *cycle time* for memories.

The *memory access time* is the time elapsed between the time a request is issued to the memory and the time when the request is fully serviced.

The *memory cycle time* is the minimum time between two successive requests. Thus, for a DRAM, the cycle time is greater than the access time, since the access time need only cover a read or a write operation, while the cycle time must cover both a read and a write operation.

Addressing every bit in a 4Gbit chip would require 32 pins. Though this is not a very large number of pins compared to what is being used on a processor package, memory chips are considerably smaller to achieve a high yield and low cost which implies a limited space (perimeter) for pins without area expansion. A small package is highly desirable for conservation of space. Thus, already in the 1970s, when the density of chips was three to four orders of magnitude less than today, multiplexing of address pins was introduced to allow for a compact package. The multiplexing corresponds to the internal organization of a memory chip, which is a two-

dimensional array. Thus, a memory cell has a *row address* and a *column address*. Typically, the memory array is square, and the number of rows and columns are the same. A 1 Gbit memory has 32768 rows and columns. Fifteen address pins are required. The row address corresponds to the most significant bits, and is given during the *row-access strobe* or RAS. The least significant bits, corresponding to the column address, is given during the *column-access strobe* or CAS. Typical times for DRAMs are given in Table 1 [8].

Year	Size	Row Access	Column Access	Cycle
1980	64 kbit	150 – 180	75	250
1983	256 kbit	120 – 150	50	220
1986	1 Mbit	100 - 120	25	190
1989	4 Mbit	80 – 100	20	165
1992	16 Mbit	~65 – 85	~15	~ 140
2003	512 Mbit	60	15	~ 120
2007	4Gbit	55	15 - 25	~ 100

Table 1: Access and cycle times in nsec for DRAMs.

Since the access to bits within the same row only requires a column address DRAMs are designed with a buffer for complete rows. Successive accesses to a given row only require that the column address be supplied, and the column access times determines the rate at which data can be accessed within a given row. Therefore, most DRAM designs allow for at least the following three operational modes:

- *Nibble mode*. The DRAM supplies four bits in sequential locations for every access.
- *Page mode*. The row buffer allows random bits within a row to be accessed by supplying only the column address, until the next refresh time.
- *Static column mode*. For accesses to the same column in different rows it suffices to supply the row address only.

Most memory systems takes advantage of the faster addressing capabilities. In *page mode*, the column address time determines the cycle time, which then is about twice the column access time. In Table 2 we have added a column to Table 1 for the cycle time of DRAMs operated in page mode. Operating a DRAM in page mode allows for a cycle time that is a factor of four to five less than the cycle time for full random access (row and column).

When a DRAM is operated in page mode, then the cycle time of the memory system is based on the column access time, and a row access will require several cycles. The need to supply a row address is often referred to as a DRAM *page fault*. A page fault may require four to five cycles.

The refresh time typically corresponds to a full memory cycle, i.e., both a row and column address must be supplied for each row of the DRAM. Thus, the total time for a refresh is

Year	Size	Row Access Time	Column Access Time	Cycle Time	Page Mode
1980	64 kbit	150 – 180	75	250	150
1983	256 kbit	120 – 150	50	220	100
1986	1 Mbit	100 - 120	25	190	50
1989	4 Mbit	80 – 100	20	165	40
1992	16 Mbit	~65 – 85	~15	~ 140	~ 30

Table 2: Access and cycle times in nsec for DRAMs operated in page mode.

approximately equal to the cycle time times the number of rows. The refresh problem is aggravated with denser memories as long as the refresh frequency is not reduced. In 2003, a typical refresh interval for a 512 Mb chip is about 7 microseconds (Samsung 7.6 microseconds).

Though the density of DRAM is growing at an impressive rate, the speed is only growing at a rate of about 7% per year. But, the clock rates of microprocessors have been increasing at a much higher rate: 18 – 35% prior to 1985 and 50 – 100% in the 1985 - 2003 period. Since then the processor clock rates have not increased due to heat problems.

Static Random Access Memories, SRAM, are designed for speed. An SRAM memory cell need not be rewritten after it is read, unlike the DRAM memory cells. Six-transistor cells are typically used for storing one bit of information. The chip area required per bit is about four times that of a DRAM cell [15]. Moreover, an SRAM cell can hold the information for a long time and need not be refreshed. Because of the emphasis on speed for SRAM design, address lines are typically not multiplexed. Thus, since there is no need to rewrite information and no multiplexing of address lines, the cycle time is equal to the access time. And no degradation of performance occurs because of unavailability due to refresh, unlike the case for DRAMs. Thus, SRAMs are 8 to 16 times faster than DRAMs, but at the expense of a density because of a 5 - 10 times larger cell size. The speed of SRAM can match that of a processor. SRAM is used for cache memory, while DRAM is used for main memory.

We have seen that the memory density improves at a very good rate while the speed does not. Before discussing memory organization we briefly review the reasons for these properties. MOS technologies may be viewed as charge transfer technologies [16]. In storing and retrieving data, charges are transferred between the pins of the packet and the memory cell storing the information as gate charges via wires on the chip. The resistance and the capacitance of the wires determine the rate at which the charge on a gate can be drained or accumulated. The size of the gate and the operating voltage of the technology determines the charge that must be transferred.

It is common to measure the size of chip designs in terms of a scalable minimum feature size, 2λ . The minimum separation between different features in the plane is also 2λ . Measured in these units a DRAM cell is about $100\lambda^2$, whereas an SRAM cell is about $400\lambda^2$ [21]. In 0.045μ technology, i.e., a technology with a minimum feature size of 0.045μ , a DRAM cell would be about $0.45 \times 0.45\mu^2$, assuming a square cell. A 1 Gbit DRAM would be about 15×15

mm² in such a technology, which is large for a memory chip. Yield is very important in high volume production. Memory chips are typically a quarter to half this size, which tells us that in commercial memory chip designs the cell size may be about $50\lambda^2$ for DRAM.

The improved density of chips in MOS technologies is due to a steady reduction in the minimum feature size. Over a 15 year period the minimum feature size has been reduced by about a factor of 10, allowing for 100 times more devices in the same physical area. MOS technologies use photolithography to produce chips with the desired functionality. Masks are exposed onto a wafer of silicon covered with photosensitive chemicals followed by etching and diffusion steps to build up transistors, wires, and interconnect. Simple processes may require three to five masks, while complex processes may require about twice this number. A wafer is a disc of silicon with a diameter that has doubled over the same period as the minimum feature size was reduced by a factor of 10. The wafer is subdivided into dies. Upon completion of the manufacturing, wafers are cut into pieces corresponding to individual dies. A die corresponds to a chip. The number of dies per wafer is approximately [8],

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter})^2}{4 \times \text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2} \times \text{Die area}} - \text{test dies per wafer}$$

The second term corrects for the fact that the wafer is round while dies are square. Thus, there is wasted space around the periphery of the wafer. A reasonable estimate for the number of wasted dies is achieved by dividing the perimeter of the wafer with the diagonal of the die, which we assume to be square. Test dies are inserted on the wafer to test for successful processing. Dies still have to be tested to verify correct functionality of individual dies. To get an idea of the number of dies per wafer, consider 1-cm-square dies on a 30-cm wafer. The formula predicts $3.14 \times 900/4 - 3.14 * 30/\sqrt{2} - 5 = 635$ dies per wafer where 5 is the number of test dies per wafer. Quadrupling the die area, corresponding to a large CPU, the die area yields 138 dies per wafer.

The die yield strongly depends upon the die area, since the number of defects is approximately proportional to the area. The *die yield* is a function of wafer yield, die area and the number of defects per unit area.

$$\text{Die yield} = \text{Wafer yield} \times \left\{ 1 + \frac{\text{Defects per unit area} \times \text{Die area}}{\alpha} \right\}^{-\alpha}$$

The wafer yield accounts for completely bad wafers, while the second factor accounts for individual dies on a good wafer. α is a function of the number of masks as well as process characteristics. For simple MOS processes $\alpha = 2.0$. With a wafer yield of 90%, two defects per square centimeter, and a die area of 1-cm-square, the die yield is $0.90 \times (1 + \frac{2 \times 1}{2.0})^{-2.0} = 0.225$. The number of good dies per wafer is $635 \times 0.225 = 143$ for a 30 cm wafer and 1-cm-square dies. Quadrupling the die area would reduce the die yield to 3.6%, and the number of good dies per wafer to 5, or a yield about 30 times smaller than for the 1 cm-square die.

Small dies are also favorable with respect to speed. Since memories typically are organized as a matrix, signals have to propagate a distance approximately equal to the side of a chip. The

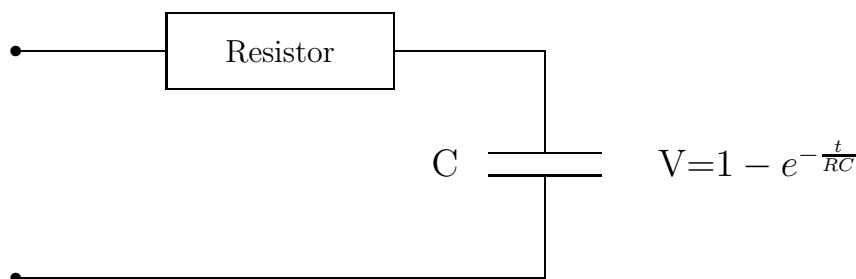


Figure 1: A simple RC -circuit and its response characteristic.

wires transmitting the charge has both a resistance and a capacitance that adds to the gate capacitance determining the charge to be transmitted. The wire capacitance is considerably smaller than the gate capacitance, and is often ignored. But, the wire resistance cannot be ignored. Together, the wire and the gate acts as a simple RC -circuit, see Figure 1. The actual switching time for the transistor is often considerably shorter than the time to move the charges along the wires. For instance, the time required to switch the state of a transistor is about 0.001 nsec in 0.045 μm technology, which is approximately what is used for 1 - 4 Gbit MOS memories. Yet, the access time is about 5 - 10 nsec. Unfortunately, the RC delay does not scale well with reduced feature size. Scaling features on the chip such that the aspect ratio is preserved, implies that reducing the linear dimension by a factor β reduces the cross section of the wire by β^2 . Reducing the gate area by a factor of β^2 reduces the capacitance with the same factor, but reducing the vertical dimension increases the capacitance for a given area by a factor of β . Thus, for a fixed length wire and proportional scaling of all dimensions, the RC -constant actually increases by a factor of β . This simple example shows the difficulty in even maintaining the speed for MOS memories as the feature sizes are reduced. Thus, major innovations in device and process technology is required in order to break the slow growth in the speed of MOS memories.

Given the severe technological challenge to drastically improve the speed of MOS memories, in particular DRAM speeds, architectural innovations are being used instead. Those ideas in essence extends the memory hierarchy onto the DRAM chips by combining SRAM and DRAM technology on the same chip. Some of these are discussed in [4, 5, 6, 7, 10, 18, 20].

Table 3 summarizes some of the developments with respect to performance.

1.2 Memory Organization

We have seen that the speed of SRAM technology is comparable to that of processor designs. One approach would be to use only SRAM for memory. But, even if that was the case, four operand instructions would in fact require that SRAMs were four times as fast as the processor for a balanced design. Using SRAM only, would increase the cost of the memory system by more than one order of magnitude compared to using DRAM for main memory. Thus, using SRAM only is not a realistic solution. The two obvious approaches to the speed problem are

	Enhanced	Cache	Synchr.	Rambus
I/O width, bits	x1,x4	x4	x4,x8,x9	x8,x9
Data rate, MHz	67	50 - 100	50 - 100	500
First-access latency				
Cache/bank hit, nsec	15 - 20	10 - 20	30 - 40	36
Cache/bank miss, nsec	35 - 45	70 - 80	60 - 80	112
Cache-fill bandwidth, MB/s	7314	114	8533	9143
Cache/bank size, bits	2048	8192	4096	8192

Table 3: Some expected performance data for MOS memories.

- increase the width of the memory
- exploit locality of reference by having most references go to a small fast memory, with the appearance of all of memory being as fast as the small memory.

Both these approaches are being used in some combination in most computer systems today. A sufficiently wide memory allows the memory to deliver enough data to keep the processor busy until the memory can deliver a new batch of data. This is the main idea in *interleaved memory* or *banked memory*. A memory hierarchy using registers and so-called caches between the arithmetic-logic-unit, ALU, and main memory follows the second approach. The ATLAS computer built in the 1950s is generally recognized as the first computer to use a memory hierarchy including a cache.

We will now first discuss banked and interleaved memory systems, then memory hierarchies using a cache.

1.2.1 Matching CPU and memory bandwidth

The number of memory banks required to make the *memory bandwidth* (i.e., the rate at which the main memory can deliver instructions and data) match the demands of the processor, is at least equal to the ratio between the memory cycle time and the processor cycle time:

$$\text{Number of Banks} \geq \frac{\text{Memory cycle time}}{\text{CPU cycle time}}$$

As an example, assume that DRAM is used as the technology for main memory. Then, with the DRAM operated in page mode and having a column access time of 10 nsec, 256 8-bit wide memory chips are required to supply a 3.2 GHz core with one 64-bit operand per cycle. Organizing the chips into 64-bit wide memory banks 32 banks would be required. In order to support four operand instructions 128 banks would be required. Such a design is balanced for column accesses. For row accesses, which is about a factor of five slower, a memory organized into 640 banks would be required in order to support four operand instructions at the full speed

Computer	Year	CPU Cycle Time (nsec)	Number of Banks	Memory Technology
Atlas	1961	700	4	Mercury Delay Lines
CDC 6600	1964	100	32	Magnetic Core
Cray-1	1976	12.5	16	Bipolar
Cray-XMP	1983	9.5	16-32	Bipolar
Fujitsu	1983	7.5	64-256	MOS
Cray-2	1985	4.1	1024	MOS
CM-2	1987	143	2048	MOS
Cray-YMP	1987	6.5		Bipolar
CM-200	1989	100	2048	MOS
CM-5	1991	32	$\leq 65,536$	MOS
Cray C-90	1993	4.0		Bipolar

Table 4: Number of memory banks in some high-performance computers.

of a single core with no operand reuse. Thus, though microprocessor based systems have had relatively simple memory systems historically, future high performance microprocessor based systems will require a sophisticated memory system for a sustained performance close to the processor peak performance. This is exaggerated in multi-core systems.

The first supercomputer, the Cray-1, had a processor cycle time of 12.5 nsec, or 80 MHz, while its bipolar memory operated at 20 MHz. On a Cray-YMP (1990), the processor clock frequency had increased to about 160 MHz, while the memory speed (still bipolar) had improved to about 25 MHz. Thus, in about 15 years the processor speed of a Cray doubled, while its memory speed increased by about 25%. The ratio between the memory cycle time and the CPU cycle time for the Cray-1 was four, while on the Cray-YMP it was six.

Banked memory systems have been used for a long time. Table 4 gives some examples. The Cray-1, Cray-XMP, Cray-YMP, Cray C-90 all used bipolar technology for both CPU and memory, while the Cray-2 used MOS memories. The difference in speed of bipolar and MOS technology accounts for most of the difference in the number of memory banks on the Cray-XMP and the Cray-2, which both had 4 CPUs. Except for the Cray-1, the Cray series computers supported concurrent access on two load and one store channel. As a contrast, today's microprocessors that fit on a single die, only has a single port to memory because of pin limitations in the packaging. Currently, the number of memory banks per CPU is typically 1 – 4 for microprocessor systems, while the Cray-2, for example, had 256 banks per CPU.

It is interesting to reflect on the total amount of memory per node that is required for a balanced design, given the memory organization. With eight 64-bit wide memory banks per processor and 1 Gbit 1-bit wide memory chips, a processor will have $8 \times 64 \times 1 \cdot 2^{30}$ bits of memory, or 512 Gbytes. With 8-bit wide memory chips, the memory per processor is reduced to 64 Gbytes, still a substantial amount of memory per processor. Thus, a memory system designed with respect to balancing the memory bandwidth with the processor cycle time will have an increasing amount of memory per CPU as the density of memory chips increases.

Remark 1: A similar phenomenon is true for secondary memory in the form of discs. The disc capacity is increasing at a much higher rate than the bandwidth to a disc.

Both *banked memory systems* and *interleaved memory systems* use memories built out of banks, typically one word wide. The main difference is in how the memory system is operated.

1.2.2 Banked and interleaved memory systems

Memory banks are typically accessed in one of two ways [8]:

- *Synchronized Access*

The banks are synchronized and accessed in parallel with the same local memory address for all banks. The retrieved data is latched for all banks and transmitted one word at a time.

- *Phased Access*

In phased access, the memory banks operate independently. No synchronization between banks is required. After each bank has stored its result, a new access can begin. Different memory banks may use different local addresses.

Interleaved memory systems are typically operated in a synchronized mode. A single local address latch suffices for all banks. *Banked memory systems* typically use the phased access mode. Each bank requires its own address latch, and the control logic is more complex. We will compare the two systems below from a performance point of view.

In a parallel memory system, the address is partitioned into a *bank number* and a *local address* within the bank. The typical way in which the bank number K and local address LA for an index A and B memory banks is determined is as follows:

$$K = A \bmod B$$

$$LA = \left\lfloor \frac{A}{B} \right\rfloor.$$

Thus, successive indices are mapped to successive memory banks in a *cyclic* manner. Usually, the number of memory banks is a power of two. For 2^k memory banks, the k lowest order bits determine the bank number, while the remaining higher order bits determine the local memory address.

Consider a system with eight memory banks and a cycle time of six CPU cycles. Since there are eight banks, the total memory bandwidth is sufficient to deliver one word per CPU cycle for accessing data allocated to successive memory banks. As an example, assume that a vector of length 32, occupying addresses 23 – 54, is to be retrieved. In a banked memory system with phased access, the words are accessed as shown in Figure 2. In the figure, the entries in the

Starting Cycle	Bank Number							
	0	1	2	3	4	5	6	7
5	24/6	25/7	26/8	27/9	28/10	29/11	30/12	23/5
13	32/14	33/15	34/16	35/17	36/18	37/19	38/20	31/13
21	40/22	41/23	42/24	43/25	44/26	45/27	46/28	39/21
29	48/30	49/31	50/32	51/33	52/34	53/35	54/36	47/29

Figure 2: Accessing a vector of length 32 in a phased access memory system.

columns for each bank show the address for which data is retrieved and the cycle during which it is retrieved (the notation is *Address/Cycle*). The first column gives the time when the first data is retrieved for each memory access. Since the starting address is 23, the first word to be retrieved is assigned to bank 7 with a local memory address of 2, assuming memory addresses start from 0. We consider a steady state scenario in which, for simplicity, a request is issued at time 0 and serviced during cycle five. (For a DRAM the access time is shorter than the cycle time, so the result may actually be available earlier, but a new request cannot be issued to the same bank before cycle six.) Thus, the first word is available from the memory latch of bank 7 during cycle five.

Two properties of the phased access to memory banks can be observed from Figure 2. First, since the memory cycle time is six CPU cycles, and there are eight memory banks, in our example requests to the same memory bank are issued once every eight cycles, since the banks are read cyclicly. Second, for every access, the local addresses for banks 0 – 6 are the same, but different from the local address of bank 7. Let the starting address be A_0 , the number of elements retrieved be N , and the memory cycle time be c . Then, the total time to retrieve the N elements is

$$c + N$$

Accessing the elements 23 – 54 in an interleaved memory system with synchronized access is shown in Figure 3. As indicated in the figure, in the synchronized access mode, memory cycles are lost if the starting bank number is greater than the (number of banks) - (memory cycle time). The number of cycles required are

$$c + \max(B - A_0 \bmod B, c) + N - (B - A_0 \bmod B)$$

In our example, the synchronized access system does not have the last item ready until cycle 41, while the phased access memory system has the last data item ready during cycle 36. Thus, the phased access system is about 10% faster in this case.

Note that for the synchronized access scheme, the second request may be issued based on the memory cycle time if the number of banks accessed in the first request is less than the memory cycle time. All subsequent accesses are determined by the number of banks for sequential access.

Starting Cycle	Bank Number							
	0	1	2	3	4	5	6	7
5								23/5
11	24/11	25/12	26/13	27/14	28/15	29/16	30/17	31/18
19	32/19	33/20	34/21	35/22	36/23	37/24	38/25	39/26
27	40/27	41/28	42/29	43/30	44/31	45/32	46/323	47/34
35	48/35	49/36	50/37	51/38	52/39	53/40	54/41	

Figure 3: Accessing a vector of length 32 in an interleaved (synchronized) memory system.

In our example, the phased access mode required one less row access than the synchronized access mode. However, the difference can be much greater. For instance, accessing two elements, one in bank 7 and the next element in bank 0, would require seven cycles in phased memory mode, and 12 cycles in synchronized access mode. In this case, the difference in performance between the two access modes is close to a factor of two. In even worse scenario for a synchronized access mode memory is an access pattern with a *stride* other than one, i.e., the difference in indices between successive accessed elements, equal to the number of banks plus one. In such a case, a banked memory operated in synchronized mode can only deliver one word for each memory cycle, while a system operated in phased access mode can deliver one word per cycle. The performance difference is a factor equal to the memory cycle time in favor of the phased access mode.

Thus, we conclude that phased access mode may offer substantially higher performance than synchronized access mode. All conventional supercomputers use phased access mode at the expense of a more complex control logic.

We notice that both in the phased and synchronized access modes a stride equal to the number of memory banks reduces the performance to the capacity of a single memory bank. So-called *bank conflicts* occur when an attempt is made to access the same memory bank within the memory cycle time.

Memory stalls occur whenever the memory cannot deliver the data at the rate required by the CPU. Stalls are more likely to happen in synchronized mode than in phased mode. We will give additional examples below.

1.3 Data Array Layouts

From our discussion of phased and synchronized access to banked memory systems, it is clear that for indices mapped cyclicly to the banks and sequential access a number of banks equal to the ratio between the memory and CPU cycle time suffice for a balanced design with respect to memory and processor capacity. The demand on the number of banks increases with the number of operands to be fetched concurrently, but may also increase as a function of the memory access pattern. The stride for accessing a single array and the relative location between arrays is important in the case of accessing multiple arrays. The mapping of the index space

Bank Number							
0	1	2	3	4	5	6	7
x(0)	x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	z(0)	z(1)	z(2)	z(3)	z(4)	z(5)
z(6)	z(7)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	y(0)	y(1)	y(2)	y(3)
y(4)	y(5)	y(6)	y(7)	—	—	—	—
—	—	—	—	—	—	—	—

Figure 4: Distribution of three arrays among eight memory banks.

to the memory banks, the data layout, is critical. Alternatives to the cyclic mapping may prove advantageous. Next we will first consider access to three one-dimensional arrays as in a vectorized operation of the form $y \leftarrow x + z$, then consider different accesses to a single multidimensional array.

1.3.1 Multiple one-dimensional arrays

We first consider an example where the array element accesses and distribution over memory banks are ideal, then a more realistic example. In both examples we assume that the memory cycle time is two CPU cycles, and that the CPU is pipelined with a depth of four.

Example 1.

The data distribution for three arrays x , y and z used for the computation $y \leftarrow x + z$ is shown in Figure 4. With a memory cycle time of two CPU cycles and a pipeline latency of four cycles, the memory access pattern is shown in Figure 5. In Figure 5 the retrieving of the x and z vectors both start in cycle 0. Each elements require two cycles to be retrieved. By the skewing of the starting addresses for the two arrays, the x retrieval is chasing the z retrieval without conflict. Moreover, the allocation of y is skewed with respect to x and z in such a way that when the first result y is ready it can be stored without conflict as well, since when the first element of y is ready to be stored in cycle 6, then banks 1 through 4 are not busy.

The data distribution and access pattern are such that there is no conflict in the memory accesses. The pipeline can operate without stalls. The total latency is $2 + 4 + 2 = 8$ cycles, where the first 2 refers to the load latency and the last 2 refers to the store latency. The number of concurrent requests serviced by the memory is six (three times the memory cycle time in CPU cycles, since there are three operands). For a memory cycle time of 10 CPU cycles, the memory system must service at least 30 requests concurrently.

Note that there are several choices for the relative allocation of x , y , and z that allow for conflict free access with the memory cycle time, number of memory banks, and pipeline depth given in

Time	Bank Number								Pipeline stage			
	0	1	2	3	4	5	6	7	1	2	3	4
0	x(0)	—	z(0)	—	—	—	—	—	—	—	—	—
1	x(0)	x(1)	z(0)	z(1)	—	—	—	—	—	—	—	—
2	—	x(1)	x(2)	z(1)	z(2)	—	—	—	0	—	—	—
3	—	—	x(2)	x(3)	z(2)	z(3)	—	—	1	0	—	—
4	—	—	—	x(3)	x(4)	z(3)	z(4)	—	2	1	0	—
5	—	—	—	—	x(4)	x(5)	z(4)	z(5)	3	2	1	0
6	z(6)	—	—	—	y(0)	x(5)	x(6)	z(5)	4	3	2	1
7	z(6)	z(7)	—	—	y(0)	y(1)	x(6)	x(7)	5	4	3	2
8	—	z(7)	—	—	—	y(1)	y(2)	x(7)	6	5	4	3
9	—	—	—	—	—	—	y(2)	y(3)	7	6	5	4
10	y(4)	—	—	—	—	—	—	y(3)	—	7	6	5
11	y(4)	y(5)	—	—	—	—	—	—	—	—	7	6
12	—	y(5)	y(6)	—	—	—	—	—	—	—	—	7
13	—	—	y(6)	y(7)	—	—	—	—	—	—	—	—
14	—	—	—	y(7)	—	—	—	—	—	—	—	—

Figure 5: Conflict free memory accesses for a three operand vector instruction. The CPU pipeline is four cycles deep and the memory cycle time is twice the CPU cycle time.

Bank Number							
0	1	2	3	4	5	6	7
x(0)	x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
z(0)	z(1)	z(2)	z(3)	z(4)	z(5)	z(6)	z(7)
—	—	—	—	—	—	—	—
y(0)	y(1)	y(2)	y(3)	y(4)	y(5)	y(6)	y(7)
—	—	—	—	—	—	—	—

Figure 6: Distribution of three arrays among eight memory banks.

this example. However, there are also several allocations that result in conflicts.

Example 2.

We assume the same memory cycle time as in Example 1, and the same pipeline depth. The only difference is that all arrays start in bank 0, as shown in Figure 6. Then, the timing diagram in Figure 7 results.

In order for the timing diagram in Figure 7 to be valid, a buffering of one of the input operands (x) by the memory cycle time is required. No buffering of the output operand is required in this case. However, should the output operand have started in bank 7, then a buffering equal to three results would have been required and the operation would have required 20 cycles instead of a minimum of 15 cycles.

Time	Bank Number								Pipeline stage			
	0	1	2	3	4	5	6	7	1	2	3	4
0	x(0)	—	—	—	—	—	—	—	—	—	—	—
1	x(0)	x(1)	—	—	—	—	—	—	—	—	—	—
2	z(0)	x(1)	x(2)	—	—	—	—	—	—	—	—	—
3	z(0)	z(1)	x(2)	x(3)	—	—	—	—	—	—	—	—
4	—	z(1)	z(2)	x(3)	x(4)	—	—	—	0	—	—	—
5	—	—	z(2)	z(3)	x(4)	x(5)	—	—	1	0	—	—
6	—	—	—	z(3)	z(4)	x(5)	x(6)	—	2	1	0	—
7	—	—	—	—	z(4)	z(5)	x(6)	x(7)	3	2	1	0
8	y(0)	—	—	—	—	z(5)	z(6)	x(7)	4	3	2	1
9	y(0)	y(1)	—	—	—	—	z(6)	z(7)	5	4	3	2
10	—	y(1)	y(2)	—	—	—	—	z(7)	6	5	4	3
11	—	—	y(2)	y(3)	—	—	—	—	7	6	5	4
12	—	—	—	y(3)	y(4)	—	—	—	—	7	6	5
13	—	—	—	—	y(4)	y(5)	—	—	—	—	7	6
14	—	—	—	—	—	y(5)	y(6)	—	—	—	—	7
15	—	—	—	—	—	—	y(6)	y(7)	—	—	—	—
16	—	—	—	—	—	—	—	y(7)	—	—	—	—

Figure 7: Conflict free memory accesses for a three operand vector instruction. The CPU pipeline is four cycles deep and the memory cycle time is twice the CPU cycle time.

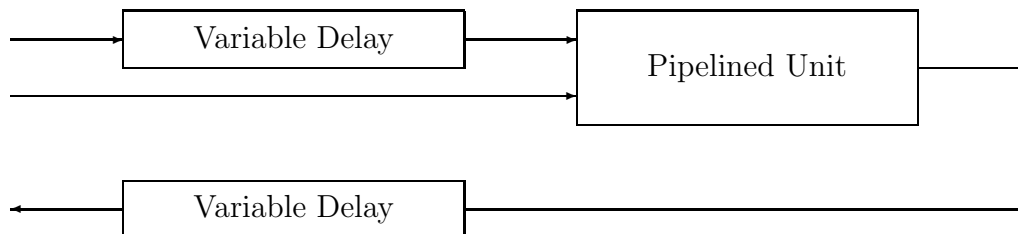


Figure 8: Buffering of input and output operands for continuous operation of pipelines.

The buffering of input and output operands can be arranged as shown schematically in Figure 8.

Though the buffering is conceptually simple, it may neither be simple nor cheap to implement since it must be programmable. The required delays depend on the allocation of the arrays and the access pattern, which need not be the same for all arrays. The setup of the buffering of the operands adds to the setup time of vector instructions (T_{base} in Lecture 2).

A variable delay can be implemented as a tapped delay line or through a memory with independent read and write ports and addresses. A tapped delay line is like a shift register with a read capability at every step of the shift register. The tap, i.e., the read stage, is set according to the desired delay. The depth of the shift register determines the maximum delay. The idea is illustrated in Figure 9.

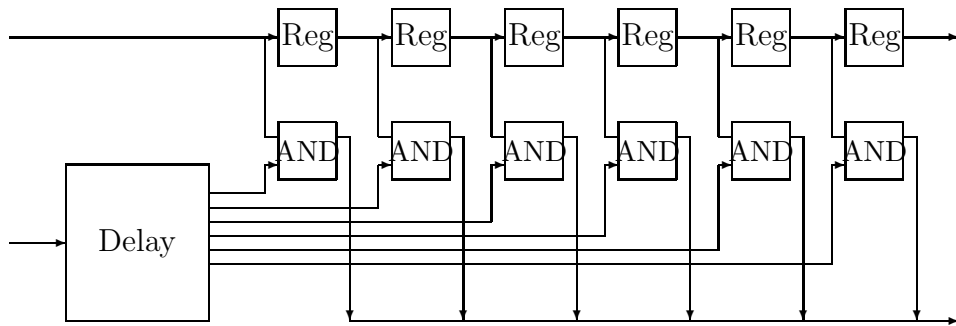


Figure 9: A tapped delay line for buffering of operands.

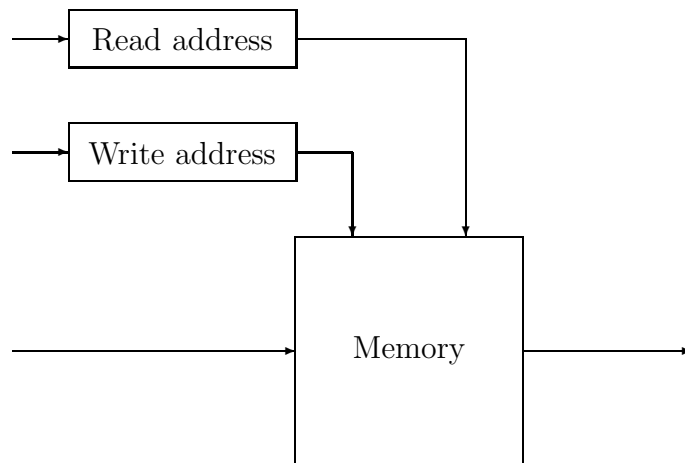


Figure 10: A double ported memory used as a delay line.

The second solution to a variable delay is illustrated in Figure 10. The dual–ported memory has separate write and read address registers. By initially setting the write address to 0 and the read address to $-d$, where d is the desired delay and preventing reads from negative addresses to actually take place, the output stream is identical to the input stream shifted d cycles. By computing addresses modulo N , the size of the memory, it acts as a circular memory. The maximum delay that can be supported is N . The case $d = 0$ may require special handling, since for this case data is routed straight through the memory.

Note that in the dual–ported memory only the cells written, and possibly the cell read changes state, while in the delay line all registers may change state. The delay line approach may consume considerably more power than the dual–ported memory.

Bank Number							
0	1	2	3	4	5	6	7
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
(8,0)	(9,0)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(79,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
(8,1)	(9,1)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(79,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
(8,2)	(9,2)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(79,2)
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—

Figure 11: Allocation of an 80 × 80 data array in column major order to 8 memory banks.

1.3.2 Single Multidimensional arrays

Most languages linearize the address space for multidimensional arrays. Fortran 77 uses *column major* order; most other languages use *row major* order. For a two-dimensional array of $M \times N$ elements, element (i, j) is mapped to index $i + Mj$ in a column major ordering and to index $Ni + j$ in row major ordering, $0 \leq i < M, 0 \leq j < N$. The linearized address space is then mapped onto the memory system. The stride between successive elements in a column is 1 in column major order while in row major order it is N , i.e., the length of a row. Similarly, the stride between successive elements in a row is 1 for row major ordering, and M , the length of a column, for column major ordering.

Traditionally, the linearized address space is mapped to memory banks, whether operated as a banked or interleaved memory system, using the *cyclic* mapping, i.e., address A is mapped to bank

$$K = A \bmod B$$

and to the local address LA is

$$LA = \left\lfloor \frac{A}{B} \right\rfloor$$

The layout of an array of shape 80×80 in a memory system with 8 banks is shown in Figure 11 for column major order. Figure 12 shows the memory layout for row major order.

Bank Number							
0	1	2	3	4	5	6	7
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,8)	(0,9)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(0,79)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(1,8)	(1,9)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(1,79)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(2,8)	(2,9)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(2,79)
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—

Figure 12: Allocation of an 80×80 data array in row major order to 8 memory banks.

The cyclic mapping is good for access to successive elements along the data array axis with stride one. In column major order, columns can be accessed at the full capacity of the memory system. Row accesses are not as favorable. If the number of rows is a multiple of the number of banks, then accesses to elements of a row of the data array implies that all request are to the same memory bank. Similarly, the use of a row major ordering in assigning array elements to memory locations, results in conflict-free access to successive elements within rows, but not to successive elements within columns. We will now give an example that illustrates the significance of this fact.

Example 3.

In Lecture 3, we discussed restructuring of code for vectorization. The restructuring decreased looping overhead, increased vector length, and, as in the case of matrix-vector multiplication, changed the algorithm from an “inner-product” algorithm to an `_AXPY` algorithm. The latter restructuring was accomplished through a change of loop order. For a given array layout, the change of loop order changes the stride in the inner loop, which often is critical for performance, as we have seen above for banked memory systems.

The `_AXPY` version of matrix-vector multiplication had the form

```

FOR I=0 TO M-1 DO
  Y(I)=0
ENDFOR
FOR J=0 TO N-1 DO
  FOR I=0 TO M-1 DO
    Y(I)=Y(I)+A(I,J)*X(J)
  
```

ENDFOR ENDFOR

In column major ordering, the elements of A are accessed with a stride of 1 in the linearized address space, but in row major ordering the stride is N in the linearized address space. Hence, for a row major ordering (used in C) and a banked memory system, an `_AXPY` algorithm may incur a bank conflict for every read, and the performance may be degraded to the speed of a single memory bank. For a row major ordering an inner product algorithm may yield higher performance, despite the fact that inner products for most pipelined architectures are less efficient than `_AXPY` operations with respect to the use of the arithmetic units.

The impact of a stride that is relatively large compared to the DRAM page size is shown in Figure 13 for matrix–vector multiplication in each node of a Connection Machine system CM–200, and for each vector unit of the Connection Machine system CM–5 in Figure 14. On the CM–200 a DRAM page fault implies that a load from memory increases by one cycle. Thus, if every load causes a page fault the memory bandwidth is reduced by a factor of two compared to no page faults. On the CM–5, the reduction is a factor of 3.5 – 5. On the Cray–1 bank conflicts reduces the effective memory bandwidth by a factor of 4, while the reduction on a Cray–YMP is a factor of six. Hence, it is important to match memory access strides (algorithm), i.e., the order in which the index space is traversed, with the data layout. As an example, in the Basic Linear Algebra Subroutines (BLAS) [14] of the Connection Machine Scientific Software Library [22], the loop order is selected at run–time based on the memory layout, axes lengths, loop and pipeline overheads [9].

1.4 Issues in conflict–free accesses in banked memory systems

Whether or not conflicts occur depend upon how arrays are distributed in memory, the access pattern (strides) and the number of memory banks. We have already seen that matrix–vector multiplication may require either row or column access in the inner loop depending on the choice of loop order. Gaussian elimination, discussed in some detail later, requires both row and column access. In straightforward implementations of Gaussian elimination, the dominating operation is an outer product of the pivot row and the pivot column updating a submatrix (defined by the pivot rows and columns). Other algorithms, such as Jacobi’s method for iterative solution of linear systems of equations, or Jacobi’s method for diagonalizing a matrix, require access to the diagonal of a matrix. Some algorithms for matrix transposition require access to the elements on antidiagonals.

Thus, many linear algebra algorithms require access to rows, columns, diagonals and antidiagonals, and blocks. In addition, divide–and–conquer algorithms, such as the FFT, require strides of the form 2^k for some range of nonnegative integers k . Other important examples of a divide–and–conquer algorithms are odd–even and parallel cyclic reduction, both of which also have strides of the form 2^k for a range of nonnegative integers k . The range depends upon the length of the transform axis, and the length of lower ordered axes in multidimensional data arrays.

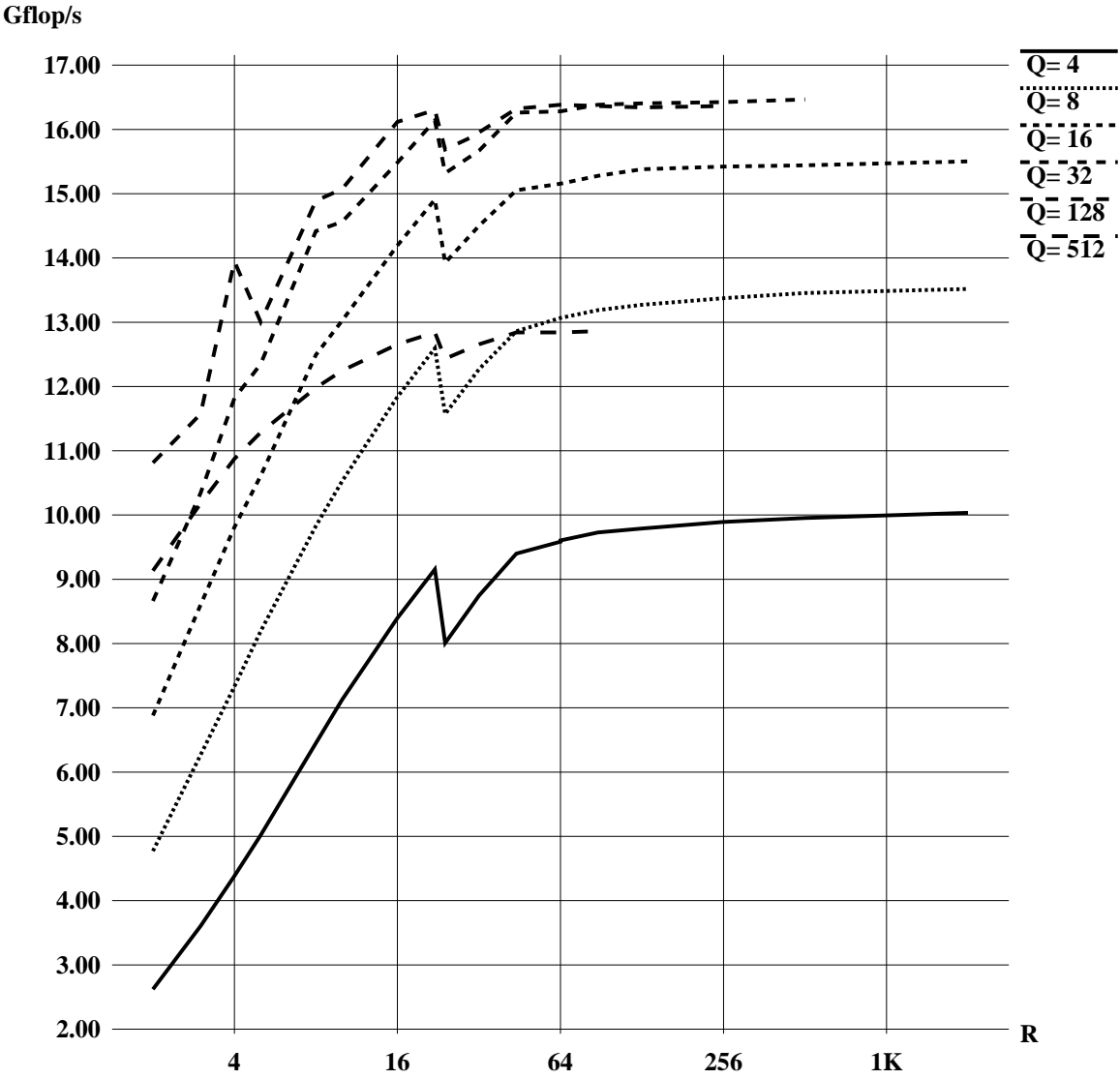


Figure 13: The aggregate performance for vector-matrix multiplication in 64-bit precision on a 2048 processor CM-200. The matrix shape is $Q \times R$.

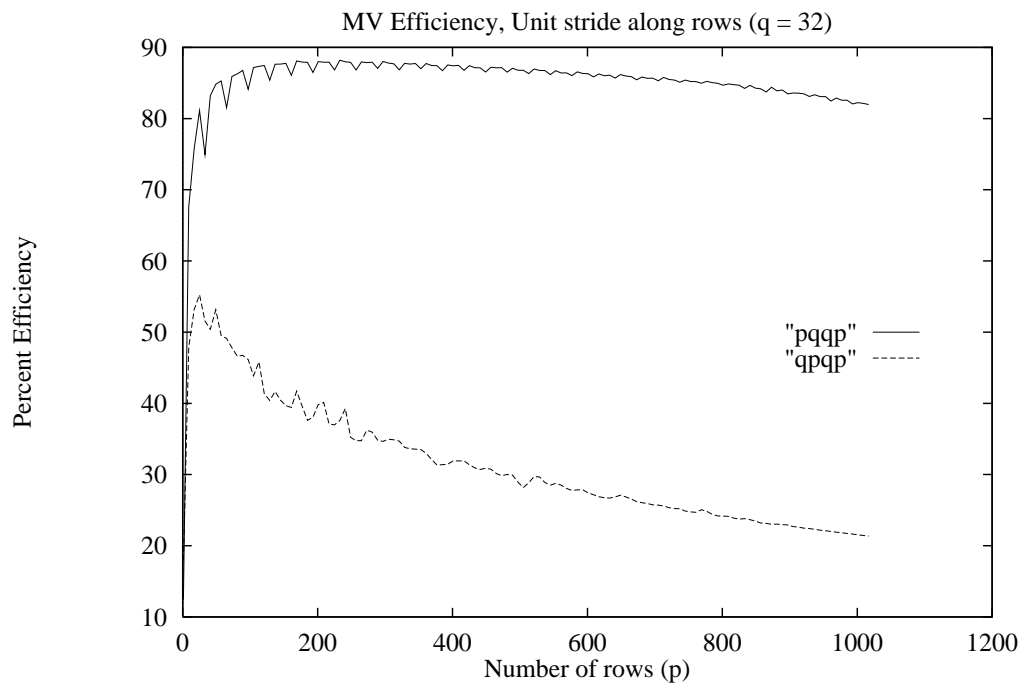


Figure 14: The matrix–vector multiplication performance in a single vector unit of the CM–5 as a function of size (and stride).

We conclude, that for many common operations in scientific computation it is desirable to guarantee conflict free access to

- Rows
- Columns
- Blocks
- Diagonals
- Antidiagonals
- Sets of elements with a stride 2^k , $k \geq 0$.

Below, we will explore a few ideas with respect to memory layout that addresses one or several of the issues above. Is there a data layout that makes all these access patterns possible without bank conflicts?

1.4.1 Data array padding

We have already observed that a number of memory banks B equal to

Stride	Words per cycle
1	$8/8 = 1.00$
2	$4/6 = 0.67$
4	$2/6 = 0.33$
8	$1/6 = 0.17$

Table 5: Performance of a banked memory system with eight banks and a six cycle access time, as a function of stride.

$$B \geq \frac{\text{memory cycle time}}{\text{processor cycle time}}.$$

is sufficient to guarantee one memory access per cycle for a stride of one, but may not be sufficient for a higher stride. For instance, consider the eight bank memory system mentioned earlier, with a memory to CPU cycle time ratio of six. With a stride of two, only four memory banks are used. With a stride of four, only two memory banks are used, and for a stride of eight only one memory bank is used for all accesses. For a stride of two, four words are delivered every six cycles; for a stride of four, only two words per six cycles are delivered. In the worst case only one word for every six CPU cycles is delivered. The memory performance as a function of stride is summarized in Table 5. The performance of the memory system may degrade by as much as a factor of six.

One solution that can be used by the programmer is to pad the array. For instance, suppose we pad the axis with stride one by one element. Then, for instance, instead of storing an 80×80 data array we store an 81×80 array in column major order, or a 80×81 array in row major order. Then, we get the layouts shown in Figures 15 and 16, respectively.

For the padded array successive elements in either a row or a column falls into successive memory banks in either the column or row major ordering if the padding is made on the axis with stride one. Moreover, successive elements of the diagonal of the unpadded array are mapped onto every other memory bank, i.e., banks 0, 2, 4, 6 and 8. The elements of the antidiagonal of the unpadded array is mapped to bank 8. Access to the antidiagonal results in severe bank conflicts.

1.4.2 Conflict-free memory accesses

Padding of the data array allowed both row and column access without bank conflicts at the expense of increasing the memory requirements by one row or column. Moreover, accessing only elements of the unpadded array requires

1. starting address
2. number of elements

Bank Number							
0	1	2	3	4	5	6	7
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
(8,0)	(9,0)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(79,0)
(80,1)	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
(7,1)	(8,1)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(78,1)
(79,1)	(80,1)	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)
(6,2)	(7,2)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(77,2)
(78,2)	(79,2)	(80,2)	(0,3)	—	—	—	—
—	—	—	—	—	—	—	—

Figure 15: Allocation of an 81×80 data array in column major order to 8 memory banks.

Bank Number							
0	1	2	3	4	5	6	7
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(0,8)	(0,9)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(0,79)
(0,80)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(1,7)	(1,8)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(1,78)
(1,79)	(1,80)	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(2,6)	(2,7)	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	(2,77)
(2,78)	(2,79)	(2,80)	(3,0)	—	—	—	—
—	—	—	—	—	—	—	—

Figure 16: Allocation of an 80×81 data array in row major order to 8 memory banks.

Bank Number							
0	1	2	3	4	5	6	7
0	—	—	3	—	—	6	—
—	9	—	—	12	—	—	15
—	—	18	—	—	21	—	—

Figure 17: Bank accesses for a stride of three.

3. stride
4. precision (number of bytes per word)

For the padded array, the number of elements in a vector load is at most equal to the number elements along the axis being accessed. Collapsing of loops to generate longer vectors is in general not possible.

In the unpadded data array either rows or columns, but not both, and diagonals and antidiagonals could be accessed without conflicts. In the padded array, rows and columns can be accessed without bank conflicts, but not antidiagonals. Accessing of diagonals is subject to some conflicts. Thus, the access for some of interesting collections of elements improved, but the accessed to others was made worse. The largest number of elements L that can be accessed with a stride of S without bank conflicts from B banks is determined by the relation

$$L \cdot \text{gcd}(S, B) \leq B$$

where $\text{gcd}(S, B)$ is the greatest common divisor of S and B . This follows from the condition that the bank number for each of the elements of the L -vector must be distinct, i.e., the set $\{(A_0 + S \times I) \bmod B \mid 0 \leq I < L\}$ must contain $L \leq B$ different numbers.

Consider our example with column major ordering and a padded array of shape 81×80 . For column access the stride is one, $\text{gcd}(1, 8) = 1$ and $L = B$ elements can be accessed without conflict. For row access, the stride is 81 and $\text{gcd}(81, 8) = 1$, and conflict free access to $L = B$ elements is again guaranteed. For the diagonal the stride is $81+1=82$ and $\text{gcd}(82, 8) = 2$ and the maximum number of elements that can be accessed along the diagonal is four. For the antidiagonal the stride is $81-1$ and $\text{gcd}(80, 8) = 8$ and only one element can be accessed without conflict.

In general, conflict-free access to a vector of $L = B$ elements with stride S requires that S and B are relatively prime. For instance, a stride of three does not result in any bank conflicts for eight memory banks, as illustrated in Figure 17.

In the following, we limit the discussion to conflict-free access to two-dimensional data arrays of shape $M \times N$ mapped to B memory banks. The generalization of the results to conflict-free access to higher dimensional data arrays is straightforward. Memory addresses are ordered cyclicly, i.e., addresses increases by one from one bank to the next in a wraparound fashion.

For an index k of a one-dimensional array the bank number is $k \bmod B$, and the local address is $\lfloor \frac{k}{B} \rfloor$. Two-dimensional arrays are mapped to a one-dimensional address space through the transformation $s_1i + s_2j$ of the element index (i, j) . For column major ordering, $s_1 = 1$ and $s_2 = M$. For row major ordering, $s_1 = N$ and $s_2 = 1$. Let $\delta_1 = s_1 \bmod B$ and $\delta_2 = s_2 \bmod B$. δ_1 and δ_2 represent the skewing with respect to bank number. The address map $s_1i + s_2j$ is sometimes referred to as a (δ_1, δ_2) skewing scheme.

We note that for a (δ_1, δ_2) skewing scheme, the stride with respect to memory banks, the *bank stride*, for access to elements within a column is δ_1 . For elements within a row the bank stride is δ_2 . The bank stride for diagonals is $\delta_1 + \delta_2$, and the bank stride for the antidiagonals is $\delta_1 - \delta_2$. The requirements for conflict-free accesses to rows, columns, diagonals and antidiagonals are:

$$\begin{array}{lll}
 \text{Row accesses:} & L \cdot \gcd(\delta_1, B) & \leq B \\
 \text{Column accesses:} & L \cdot \gcd(\delta_2, B) & \leq B \\
 \text{Diagonal accesses:} & L \cdot \gcd(\delta_1 + \delta_2, B) & \leq B \\
 \text{Antidiagonal accesses:} & L \cdot \gcd(\delta_1 - \delta_2, B) & \leq B
 \end{array}$$

If access to one element per bank is desired, or $L = B$, then $\gcd(\delta_1, B)$, $\gcd(\delta_2, B)$, $\gcd(\delta_1 + \delta_2, B)$ and $\gcd(\delta_1 - \delta_2, B)$ must all be one. Hence, the listed structures cannot all be accessed conflict-free if B is even. We see this from the fact that for B even, choosing δ_1 and δ_2 to be odd numbers not being factors of B (to satisfy the first two relationships) does not solve the problem. The last two conditions will not be satisfied, because $\delta_1 + \delta_2$ and $\delta_1 - \delta_2$ are both even.

Note that in a column major order with $s_1 = 1$ and $s_2 = M$, rows can be accessed without bank conflicts for B even if M is odd. Similarly, both rows and columns can be accessed without bank conflicts in row major order if N is odd and B even. We used this fact in our example above.

From the analysis of the requirements for accessing rows, columns, diagonals and antidiagonals we conclude that either the number of banks must be odd in order to allow for accessing all banks without conflict for the specified access patterns, or we must accept that the memory system may deliver fewer elements than the number of banks. Let the access pattern define a bank stride of m , i.e., $m = \delta_2$ for row access, $m = \delta_1$ for column access, $m = \delta_1 + \delta_2$ for diagonal access and $m = \delta_1 - \delta_2$ for antidiagonal access. Then, for B even, the largest number of elements that can be guaranteed to be accessed conflict-free for the four considered access patterns is $L = \frac{B}{2}$, or $\gcd(m, B) = 2$ for at least one of the access patterns.

Note that guaranteed access to an L -vector of length $\frac{B}{2}$ is the best possible case for *any* even B . Thus, choosing $B = 2^b$ is as good as any other even B with respect guaranteed access, assuming a suitable skewing scheme can be found. Choosing $B = 2^b$ allows for a very simple calculation of the bank number for a data index. The bank number is simply determined by the b least significant bits of the binary encoding of the linearized index space.

The skewing scheme affects the complexity of the address calculation within the banks and possibly also the memory utilization. Recall that in our example of storing an 80×80 array as an 80×81 in row major ordering, the net effect was a $(1, 1)$ skewing and a waste of memory equal to one column of the array. Another consequence of skewing is that the *elements* of the

L -vector may *appear out of order*. Thus, a reordering of the elements may be required both during load and store operations.

Note further that though any even number of banks yield the same guarantee with respect to row, column, diagonal and antidiagonal access, assuming a suitable skewing scheme can be found, it is not the case for accesses with a stride of $S = 2^k$. For $B = 2^b$ such accesses yield L -vectors with a length of

$$L = \frac{B}{\gcd(2^k, 2^b)} = \begin{cases} 2^{b-k} & k \leq b \\ 1 & k > b \end{cases}$$

Note that for B being twice an odd number, accesses with stride 2^k yields L -vectors of length equal to the odd number, or $B/2$. Hence, though which even number is chosen does not matter with respect to row, column, diagonal and antidiagonal accesses, it is very significant for other common access patterns.

Next we will first propose a skewing scheme that allow for conflict-free access to L -vectors of length at least $\frac{B}{2}$ selected from rows, columns, diagonals, antidiagonals or blocks for $B = 2^b$ banks. Then, we will discuss selecting a suitable odd number of banks for a memory system and associated skewing schemes.

For access to subblocks of a matrix we notice that for B banks, blocks of shape $\delta \times \frac{B}{\delta}$ can be accessed conflict-free by a $(\frac{B}{\delta}, 1)$ skewing scheme. In this scheme, the elements within a row of the block occupies $\frac{B}{\delta}$ successive memory banks. By skewing successive rows by this number of banks, conflict-free access to a block of B elements is clearly guaranteed. In general, for accessing L -vectors and square blocks, a skewing of successive rows of at least $\sqrt{L} = \delta_1$ is desired.

Finally we notice that today's high performance computers have a large amount of primary memory, ranging from a few Gbytes up to hundreds of Gbytes. One of the tradeoffs in memory system design is width versus depth, i.e., the size of individual banks. A wide memory offers a high peak memory bandwidth compared to a narrow memory. The cost for this feature is more complex control logic and potentially longer buses, which may have a negative impact on speed. If the memory system design is made such that one L -vector for every memory cycle matches the demands of the CPU(s), then the peak memory bandwidth exceeds the peak processing capacity by a factor of $\frac{B}{L}$. But, except for relatively small memory systems, the excess peak memory bandwidth may be achieved without increasing the desired total storage. Notice though, that with the increased density of memory chips organized with single bit cells, a 64-bit wide memory bank in 16 Mbit chip technology has 128 Mbytes of storage capacity. A memory system with 32 banks and minimum depth (one chip) has 4 Gbytes storage capacity. Chips are also produced 4 bits wide. In 64-bit, 4 bit wide memory technology a 64-bit wide minimum depth bank would also offer 128 Mbytes of storage and 8192 banks would offer 1 Tbyte of storage. For comparison, a 1024 processor CM-5 has 4096 memory banks. The length of the L -vector determines how many processors can be supported effectively.

In the following we will discuss a few choices of δ_1 , δ_2 and B and the impact on maximum guaranteed length of the L -vectors, the address calculation, memory utilization and alignment.

Bank Number							
0	1	2	3	4	5	6	7
(0,0)	—	(0,1)	—	(0,2)	—	(0,3)	—
(0,4)	—	(0,5)	—	(0,6)	—	(0,7)	—
—	(1,3)	—	(1,0)	—	(1,1)	—	(1,2)
—	(1,7)	—	(1,4)	—	(1,5)	—	(1,6)
(2,1)	—	(2,2)	—	(2,3)	—	(2,0)	—
(2,5)	—	(2,6)	—	(2,7)	—	(2,4)	—
—	(3,0)	—	(3,1)	—	(3,2)	—	(3,3)
—	(3,4)	—	(3,5)	—	(3,6)	—	(3,7)
(4,2)	—	(4,3)	—	(4,0)	—	(4,1)	—
(4,6)	—	(4,7)	—	(4,4)	—	(4,5)	—
—	(5,1)	—	(5,2)	—	(5,3)	—	(5,0)
—	(5,5)	—	(5,6)	—	(5,7)	—	(5,4)
(6,3)	—	(6,0)	—	(6,1)	—	(6,2)	—
(6,7)	—	(6,4)	—	(6,5)	—	(6,6)	—
—	(7,2)	—	(7,3)	—	(7,0)	—	(7,1)
—	(7,6)	—	(7,7)	—	(7,4)	—	(7,5)

Figure 18: Mapping of an 8×8 array by a $(3,2)$ skewing scheme for row major ordering.

1.4.3 Memory systems with $B = 2^b$ banks.

We have already noticed that for an even number of banks L -vectors with a length equal to $\frac{B}{2}$ is the best we can hope for one or more of row, column, diagonal or antidiagonal accesses. In order to support conflict-free accesses to square blocks a skewing of either rows or columns of at least \sqrt{L} is desired. Thus, we let $L = 2^{2l}$ and $B = 2^{2l+1}$. We assume a row major ordering, which now is generalized to mean that the stride within a row is not necessarily one, but lower than the stride within columns. As before, rows are ordered before columns in row major ordering.

There exists several possible skewing schemes that allow for conflict-free access to L -vectors (of length $B/2$) selected as contiguous segments of rows, columns, diagonals, or antidiagonals, or as a block. Here we will show that $(2^l + 1, 2)$ skewing has the desired property. In terms of the array indices (i, j) , this skewing scheme can be achieved through $s_2 = \delta_2 = 2$ and $s_1 = \lceil \frac{\delta_2 N}{B} \rceil B + \delta_1$, where the shape of the data array is $M \times N$. Clearly, $s_1 \bmod B = \delta_1$ and $s_2 \bmod B = \delta_2$. s_1 is determined such that each row of the data array is effectively padded to the smallest multiple of the number of banks sufficiently large to accommodate the entire row. Figure 18 gives an example of $(3, 2)$ skewing for eight banks and a data array of shape 8×8 . Rows, columns, diagonals and antidiagonals can all clearly be accessed without conflict, since with respect to row access $\gcd(2, 8) = 2$, column access $\gcd(3, 8) = 1$, diagonal access $\gcd(5, 8) = 1$ and antidiagonal access $\gcd(1, 8) = 1$ and $L = 4$.

By having an even number of banks, we know that for some of the accesses row, column, diagonal or antidiagonal, the length of the L -vector is at best $\frac{B}{2}$. Choosing the row stride to be two is compatible with this property. Choosing the column bank stride to be odd guarantees that all the other three entities can be accessed conflict-free since $B = 2^b$. In fact, since there

is no common factor between B and the bank access strides for any of the three patterns, L -vectors of length B are possible for column, diagonal and antidiagonal access. The particular choice of odd column skewing is intended to support L -vectors of length $\frac{B}{2}$ for block access.

We will now prove that square blocks of L elements indeed can be accessed conflict-free. The elements of a block $(i + \xi, j + \psi)$, where $i \leq i + \xi < i + 2^l$ and $j \leq j + \psi < j + 2^l$, are mapped to the banks $((i + \xi)(2^l + 1) + (j + \psi)2) \bmod 2^{2l+1}$. Consider two different matrix elements within the block, $(i + \xi_1, j + \psi_1)$ and $(i + \xi_2, j + \psi_2)$, where $0 \leq |\xi_1 - \xi_2| < 2^l$ and $0 \leq |\psi_1 - \psi_2| < 2^l$. The absolute difference in the row and column indices must be less than 2^l since we assume square blocks of size 2^l . If the two elements are mapped to the same bank, then

$$(i + \xi_1)(2^l + 1) + (j + \psi_1)2 \equiv_B (i + \xi_2)(2^l + 1) + (j + \psi_2)2,$$

where \equiv_B means equivalence modulo the number of banks B . We can remove the starting location of element (i, j) from this equivalence relationship

$$\xi_1(2^l + 1) + \psi_1 2 \equiv_B \xi_2(2^l + 1) + \psi_2 2$$

This expression can in turn be rewritten as

$$(\xi_1 - \xi_2)(2^l + 1) \equiv_B (\psi_2 - \psi_1)2$$

The left hand side of this expression is either in the range 0 to $2^{2l} - 1$, or $2^{2l+1} - 2^{2l}$ to $2^{2l+1} - 1$ modulo 2^{2l+1} , while the right hand side is either in the range 0 to $2(2^l - 1)$ or $2^{2l+1} - 2 \cdot 2^l + 1$ to $2^{2l+1} - 1$, since the difference in row and column indices for a block is at most $2^l - 1$. For the equivalence to be true, either

1. $(\psi_2 - \psi_1)2 \bmod B$ is in the range $[0, 2(2^l - 1)]$ and $(\xi_1 - \xi_2)(2^l + 1) \bmod B$ is in the range $[0, 2^{2l} - 1]$, or
2. $(\psi_2 - \psi_1)2 \bmod B$ is in the range $[2^{2l+1} - 2 \cdot 2^l + 1, 2^{2l+1} - 1]$ and $(\xi_1 - \xi_2)(2^l + 1) \bmod B$ is in the range $[2^{2l+1} - 2^{2l}, 2^{2l+1} - 1]$.

For the first case it is easy to see that the only condition for which the equivalence is true is $\xi_1 = \xi_2$ and $\psi_1 = \psi_2$. The same is true for the second condition. Thus, we have proven by contradiction that within a square block of L elements no two elements are mapped to the same bank.

The bank number K for data array element (i, j) and $(2^l + 1, 2)$ skewing is computed as

$$K = ((2^l + 1)i + 2j) \bmod B$$

A relatively simple way of computing the local addresses within a bank is

Bank no.	0	1	2	3	4	5	6	7
Diag. el.	(0,0)	(5,5)	(2,2)	(7,7)	(4,4)	(1,1)	(6,6)	(3,3)

Figure 19: Allocation of diagonal elements to banks for an 8×8 array and $(3, 2)$ skewing.

$$LA = i \times \left\lceil \frac{\delta_2 N}{B} \right\rceil + \left\lfloor \frac{\delta_2 j}{B} \right\rfloor$$

for an $M \times N$ array and B banks, assuming array indices start from zero. Since $\delta_2 = 2$ and $B = 2^b$, shift operations suffice to evaluate the floor and ceiling functions. Integer division can be avoided, but integer multiplication and addition is necessary.

In this mapping each row of the data array starts in a new row in memory, as shown in Figure 18. Moreover, at least half of the memory is unused, since each data array row only use half of the memory banks; $B / \gcd(\delta_2, B) = (8 / \gcd(2, 8)) = 4$. This is apparent from Figure 18 in which an 8×8 matrix is stored in eight memory banks, i.e., $l = 1$. The bank skewing between successive elements in a row is 2, and between successive elements in a column it is $2^1 + 1 = 3$. The bank number is determined by $(3i + 2j) \bmod 8$, and the local address is computed as $i \lceil \frac{2 \times 8}{8} \rceil + \lfloor \frac{2 \times j}{8} \rfloor$.

The memory utilization can be improved by storing another array of the same shape in the “holes”, or by “packing” the matrix into half as many memory rows, since successive elements in a row are allocated to even banks for even rows, and to odd banks for odd rows. The packing is fairly simple to accomplish for this skewing scheme, as illustrated by the local address calculation below:

$$LA = \left\lfloor \frac{i}{2} \right\rfloor \left\lceil \frac{\delta_2 N}{B} \right\rceil + \left\lfloor \frac{\delta_2 j}{B} \right\rfloor$$

With this packing scheme, the waste of memory is limited to at most $B - 2$ words per pair of data array rows.

With respect to ordering of the data it is clear that, modulo B , L -vectors of row and column segments are ordered. However, diagonals are unordered, as seen in our example where the correspondence between diagonal elements and banks are as shown in Figure 19. That the diagonal elements are unordered follows from the fact that the bank stride is $(2^l + 3)$. Thus, accessing an L -vector of 2^{2l} or 2^{2l+1} elements implies several wraparounds, $\frac{2^l+3}{2}$ and $2^l + 3$ respectively, with respect to bank addresses.

The antidiagonals are also unordered, in general. The bank stride is $2^l - 1$, which in our example in Figure 18 yields a bank stride of one. Note, however, that antidiagonals shall be ordered in reverse, while order in Figure 18 is forward. The allocation of the antidiagonal is shown in Figure 20. In general, wrap around $2^l - 1$ times is required for the complete antidiagonal.

Bank no.	0	1	2	3	4	5	6	7
Diag. el.	(2,5)	(3,4)	(4,3)	(5,2)	(6,1)	(7,0)	(0,7)	(1,6)

Figure 20: Allocation of antidiagonal elements to banks for an 8×8 array and $(3, 2)$ skewing.

Bank Number				
0	1	2	3	4
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(0,5)	(0,6)	(0,7)	—	—
(1,3)	(1,4)	(1,0)	(1,1)	(1,2)
—	—	(1,5)	(1,6)	(1,7)
(2,1)	(2,2)	(2,3)	(2,4)	(2,0)
(2,6)	(2,7)	—	—	(2,5)
(3,4)	(3,0)	(3,1)	(3,2)	(3,3)
—	(3,5)	(3,6)	(3,7)	—
(4,2)	(4,3)	(4,4)	(4,0)	(4,1)
(4,7)	—	—	(4,5)	(4,6)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)
(5,5)	(5,6)	(5,7)	—	—
(6,3)	(6,4)	(6,0)	(6,1)	(6,2)
—	—	(6,5)	(6,6)	(6,7)
(7,1)	(7,2)	(7,3)	(7,4)	(7,0)
(7,6)	(7,7)	—	—	(7,5)

Figure 21: Mapping of an 8×8 array by a $(2,1)$ skewing scheme for row major ordering.

It can also be shown that the block elements are unordered.

We will defer the discussion of data alignment until the discussion of the Burroughs Scientific Processor, the BSP.

Finally we notice that the number of admissible memory banks is rather limited in this approach. Successive numbers of admissible banks grows by a factor of four and is 8, 32, 128, 512, 2048, 8192, 65536, etc.

1.4.4 Memory systems with an odd number of banks.

With an odd number of banks, a skewing scheme may contain even bank strides without compromising the length of the L -vector, as long as there is no common factor with the number of banks. Thus, for instance, a $(2^l, 1)$ skewing scheme for $B = 2^{2l} + 1$ allows for both conflict-free access to both row and column segments of length B . Figure 21 shows a $(2, 1)$ skewing applied to a 8×8 array stored in $B = 2^2 + 1 = 5$ banks.

For conflict-free access to diagonals and antidiagonals, it is necessary to show that $\gcd(2^L + 1, 2^{2l} + 1) = 1$ and $\gcd(2^l - 1, 2^{2l} + 1) = 1$, respectively. We will now prove these properties by contradiction.

Assume that $\gcd(2^l + F, 2^{2l} + 1) = C \neq 1$, where $F = \pm 1$. Then,

$$2^{2l} + 1 = CD \quad \text{for some } D$$

and

$$2^l + F = CE \quad \text{for some } E.$$

Substituting $(CE - F)^2$ for 2^{2l} in the first equality yields

$$(CE - F)^2 + 1 = C^2E^2 - 2CEF + F^2 + 1 = CD.$$

Hence $F^2 + 1 = 2$ must be divisible by C . But C must be odd, a contradiction. So C must be 1.

Since the row length for a square block of 2^{2l} elements is 2^l and the bank stride for rows is one and for columns 2^l , it is clear that square blocks can be accessed conflict-free.

Thus, we have showed that L -vectors of at least length $B - 1$ forming segments of rows, columns, diagonals and antidiagonals can be accessed conflict-free by using a $(2^l, 1)$ skewing for $B = 2^{2l} + 1$ banks.

The address computation is no longer as simple as for the number of banks being a power of two. In the case of $B = 2^{2l} + 1$ a modulo operation is required in determining the bank number

$$K = (2^l i + j) \bmod B$$

The local address may be computed as

$$LA = i \left\lceil \frac{N}{B} \right\rceil + \left\lfloor \frac{j}{B} \right\rfloor.$$

Thus, computing the local address requires an integer divide per element, and an integer addition and possibly also an integer multiplication for each element accessed, depending upon the access pattern. Modulo operations and integer division are expensive operations. For the Burroughs Scientific Processor, Lawrie and Vora [13] proposed a scheme reducing the number of multiply-add modulo operations required. Some multiply-add modulo operations are replaced with modulo addition, others are performed using ROM tables. They also proposed to eliminate the need for integer division at the expense of using only L banks per row instead of all $L + 1$ banks. The scheme by Lawrie and Vora for local address computation applied to the $(2, 1)$ skewing scheme yields the addresses

$$LA = i \left\lceil \frac{N}{L} \right\rceil + \left\lfloor \frac{j}{L} \right\rfloor.$$

Since $L = 2^{2l}$ a shift operation suffice to perform the division. The memory allocation is shown in Figure 22.

Bank Number				
0	1	2	3	4
(0,0)	(0,1)	(0,2)	(0,3)	—
(0,5)	(0,6)	(0,7)	—	(0,4)
(1,3)	—	(1,0)	(1,1)	(1,2)
—	(1,4)	(1,5)	(1,6)	(1,7)
(2,1)	(2,2)	(2,3)	—	(2,0)
(2,6)	(2,7)	—	(2,4)	(2,5)
—	(3,0)	(3,1)	(3,2)	(3,3)
(3,4)	(3,5)	(3,6)	(3,7)	—
(4,2)	(4,3)	—	(4,0)	(4,1)
(4,7)	—	(4,4)	(4,5)	(0,6)
(5,0)	(5,1)	(5,2)	(5,3)	—
(5,5)	(5,6)	(5,7)	—	(5,4)
(6,3)	—	(6,0)	(6,1)	(6,2)
—	(6,4)	(6,5)	(6,6)	(6,7)
(7,1)	(7,2)	(7,3)	—	(7,0)
(7,6)	(7,7)	—	—	(7,5)

Figure 22: Mapping of an 8×8 array by a $(2,1)$ skewing scheme and $L = B - 1$ elements per memory row for row major ordering.

	Bank number					Stride
	0	1	2	3	4	
Col	(0,0)	(3,0)	(1,0)	(4,0)	(2,0)	$2^l = 2$
Diag. el.	(0,0)	(2,2)	(4,4)	(1,1)	(3,3)	$2^l + 1 = 3$
Antidiag. el.	(3,4)	(4,3)	(0,7)	(1,6)	(2,5)	$2^l - 1 = 1$

Figure 23: Allocation of columns, diagonals and antidiagonals to banks for an 8×8 array and $(2, 1)$ skewing for $B = 5$ banks.

The amount of wasted memory for $B = 2^{2l} + 1$ banks and the addressing scheme requiring division is at most $B - 1$ elements per data array row. Removing the division by storing only $B - 1$ elements per row adds a waste of one location per memory row, at most. For large B and arrays for which $N \gg B$, the relative memory waste may be very small.

L -vectors for row access are ordered while column vectors are unordered. The ordering of L -vectors for columns, diagonals and antidiagonals are all unordered as shown in Figure 23.

There are many odd numbers that may be used for the number of banks. The choice $2^{2l} + 1$ with $(2^l, 1)$ skewing had very little excess memory bandwidth for the access schemes we considered. But, if a wide memory is desired for some reason, it may be acceptable to use a smaller fraction of the peak memory bandwidth. For instance, choosing $B = 2^{2l+1} - 1$ with a $(2^l, 1)$ skewing yields conflict-free access to 2^{2l} -vectors for rows, columns, diagonals and antidiagonals. It is obvious for rows and columns. The claim that $\gcd(2^{2l} + 1, 2^{2l+1} - 1) = 1$ and $\gcd(2^{2l} - 1, 2^{2l+1} - 1) = 1$ can be proved in a similar way as for $B = 2^{2l} + 1$. The access to blocks is somewhat more

l	$2^{2^l} + 1$	Prime?	$2^{2^{l+1}} - 1$	Prime?
1	5	✓	7	✓
2	17	✓	31	✓
3	65		127	✓
4	257	✓	511	
5	1025		2047	
6	4097		8191	✓
7	16385		32767	

Table 6: Values of $2^{2^l} + 1$ and $2^{2^{l+1}} - 1$ and their primality.

complex.

Choosing $B = 2^{2^l} - 1$, instead of for instance $2^{2^l} + 1$, or $B = 2^{2^{l+1}} + 1$ instead of $2^{2^{l+1}} - 1$ reduces the guaranteed length of the L -vector for conflict-free access to rows, columns, diagonals and antidiagonals with a $(2^l, 1)$ skewing scheme. This can be seen from the fact that the required bank strides are $2^l - 1$, 2^l and $2^l + 1$. By necessity one of these numbers must be a multiple of three. But, since we have shown that $2^{2^l} + 1$ is relatively prime to these three numbers, $2^{2^l} - 1$ must be divisible by three, since 2^{2^l} clearly is not. A similar argument shows that $2^{2^{l+1}} + 1$ is divisible by three.

1.4.5 Memory systems with a prime number of banks.

Prime numbers are odd numbers, except for the number two. But, not all numbers are of the form $2^{2^l} + 1$ or $2^{2^{l+1}} - 1$, neither are all such numbers prime. Table 6 some values of $2^{2^l} + 1$ and $2^{2^{l+1}} - 1$ and marks those which are prime. $B = 2^{2^l} + 1$ is a prime number for $l = 1$, $l = 2$, and $l = 4$, but not for $l = 3$, $l = 5$, $l = 6$ and $l = 7$. $B = 2^{2^{l+1}} - 1$ is a prime number for $l = 1$, $l = 2$, $l = 3$, and $l = 6$, but not for $l = 4$, $l = 5$ and $l = 7$. Prime numbers of the form $2^n - 1$ are known as Mersenne primes.

The reason for considering a prime number of banks is that more general access patterns than discussed above is often desired. The case with array accesses with a stride of the form 2^k , $k \geq 0$, has already been mentioned as typical in many divide-and-conquer algorithms. But, also for other computations access to data along different axis with a stride different from one is often desired.

Assume an array $A(i, j)$ of shape $M \times N$ is accessed with stride S_1 along the first axis, i.e., within columns, and stride S_2 along the second axis, i.e., within rows. Let the array be stored with a (δ_1, δ_2) skewing scheme. The set of indices accessed by an L -vector, starting at indices (i, j) , is $\{(i, j), (i + S_1, j + S_2), (i + 2S_1, j + 2S_2), \dots, (i + (L - 1)S_1, j + (L - 1)S_2)\}$. The memory addresses being accessed are $base_B + (i + rS_1)\delta_1 + (j + rS_2)\delta_2$, or $A_0 + dr$, where $A_0 = (base_B + i\delta_1 + j\delta_2) \bmod B$ and $d = (S_1\delta_1 + S_2\delta_2) \bmod B$. A_0 is the starting memory bank number for the L -vector; d is the memory bank stride.

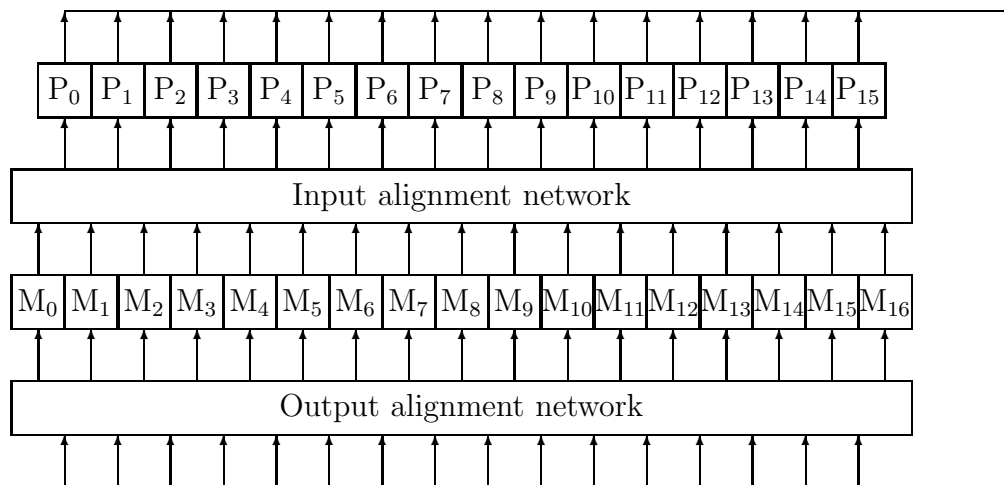


Figure 24: The BSP system architecture.

The requirement for conflict-free access to L elements is $L \leq \frac{B}{\gcd(d, B)}$, as before. But, since $0 \leq S_1 < M$ and $0 \leq S_2 < N$ it is desirable that B be large in order to reduce the number of pairs (S_1, S_2) for which $(S_1\delta_1 + \delta_2 S_2)$ is a multiple of B . Whenever this happens, the bank stride is zero and the memory operates at the speed of a single bank. Moreover, if B can be factored, then the more factors there are the larger the chance is that $(S_1\delta_1 + \delta_2 S_2)$ happens to be a multiple of one of those factors, and the length of the L -vector is reduced by the common factor. Thus, ideally B should be a large prime to minimize the chance that strides may cause short L -vectors (bank conflicts).

1.5 The Burroughs Scientific Processor (BSP)

The Burroughs Scientific Processor (BSP) [11, 12, 13] was designed for 17 memory banks and L -vectors of length 16 (to support 16 processors). The schematic of the system architecture of the BSP is shown in Figure 24. Two important issues had to be addressed for the memory system of the BSP: data alignment and address calculation. We will first discuss data alignment, then address calculation.

We illustrate the alignment problem by considering an 8×8 array stored in column major order with $(1, 1)$ skewing in five memory banks, instead of 17 banks as used in the BSP. The analogous set of issues occur if a row major ordering is assumed. Moreover, we assume that only 16 elements is stored in each memory row in order to avoid an integer divide in the local address arithmetic, as explained above. The array layout is shown in Figure 7.

With $(1, 1)$ skewing and a column major ordering columns are correctly ordered. However, a row access requires reordering, and elimination of one entry since only four elements shall be delivered of the five supplied by the memory. (For the BSP 16 elements out of 17 is delivered by the alignment network.) The alignment for access to the first four elements of row zero is

Bank Number				
0	1	2	3	4
(0,0)	(1,0)	(2,0)	(3,0)	—
(5,0)	(6,0)	(7,0)	—	(4,0)
(2,1)	(3,1)	—	(0,1)	(1,1)
(7,1)	—	(4,1)	(5,1)	(6,1)
—	(0,2)	(1,2)	(2,2)	(3,2)
(4,2)	(5,2)	(6,2)	(7,2)	—
(1,3)	(2,3)	(3,3)	—	(0,3)
(6,3)	(7,3)	—	(4,3)	(5,3)
(3,4)	—	(0,4)	(1,4)	(2,4)
—	(4,4)	(5,4)	(6,4)	(7,4)
(0,5)	(1,5)	(2,5)	(3,5)	—
(5,5)	(6,5)	(7,5)	—	(4,5)
(2,6)	(3,6)	—	(0,6)	(1,6)
(7,6)	—	(4,6)	(5,6)	(6,6)
—	(0,7)	(1,7)	(2,7)	(3,7)
(4,7)	(5,7)	(6,7)	(7,7)	—

Table 7: Mapping of an 8×8 array for $B = 5$, $L = 4$, $K = (A_0 + dr) \bmod B$ and $LA = \lfloor (A_0 + dr)/L \rfloor$.

shown in Figure 25. Note that if instead of elements 0 through 3 of row zero elements 1 through 4 would have been desired, a different alignment is required.

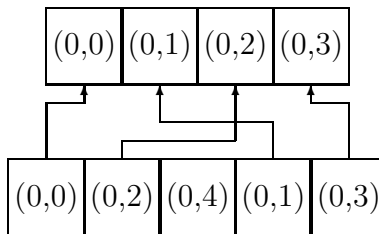
We now consider the computation of the bank number K

$$K = (A_0 + dr) \bmod B$$

and the local address

$$LA = \left\lfloor \frac{A_0 + dr}{L} \right\rfloor.$$

L -vector



Memory

Figure 25: Alignment for the first four elements of row zero of an 8×8 array assigned to 5 banks in column major order and $(1, 1)$ skewing.

With this addressing scheme, the computation of the bank number is difficult while the computation of the local address is relatively simple. However, the simplicity in computing the local address is gained at a loss of memory utilization as discussed before. The fraction of memory waste is:

$$\text{relative memory waste} = \frac{B - L}{B}.$$

The waste is apparent in Figure 7.

In order to compute the local address, A_0 , d , r and L must be known. By letting all L -vectors have the same length, L can be made a constant, independent of access. If fewer than L elements are desired, padding must be performed. A_0 and d are the same for all banks in a given L -vector access. Hence, only r is unique to a bank. But, r can be derived from the bank address K and the stride information d ; r need not be transmitted, neither does the local bank address. It can be computed within the bank, eliminating the need for an addressing network in addition to the alignment network for data. From the relationship $K = (A_0 + dr) \bmod B$ we can solve for r :

$$r = ((K - A_0)d') \bmod B$$

where $d \cdot d' \bmod B = 1$. Then, once r is known it can be used in the local address calculation with no other information unique to the bank.

An interesting idea to simplify the address computation *and* fully utilize the memory is to use a mixed radix residue number system for addresses instead of a binary number system [19]. Let the number of banks be $B = 2^b - 1$, and the local address space be 2^{w-b} , i.e., the size of each memory bank is a power of two. Let B and 2^{w-b} be the radices of the address representation. Moreover, let $B' = 2^b$, and let the address A have the representation $A = \sum_i a_i B'^i$. Then the residue with respect to 2^{w-b} is simply the number represented by the last $w - b$ bits. This residue represents the local address LA . Computing the bank number K is also fairly simple:

$$\begin{aligned} K &= A \bmod B \\ &= \left(\sum_i a_i B'^i \right) \bmod B \\ &= \left(\sum_i a_i (1 + B)^i \right) \bmod B \\ &= \left(\sum_i a_i (1 + \text{multiples of } B) \right) \bmod B \\ &= \left(\sum_i a_i \right) \bmod B. \end{aligned}$$

Modulo B addition is simply 1's complement addition (i.e., an end-around carry needs to be used). Note that the address space used is $2^{w-b}(2^b - 1) = 2^w(1 - 2^b)$, but all memory banks are fully utilized.

The same ideas can also be used for $B = 2^b + 1$. For this case, both addition and subtraction are required in computing the bank number.

1.6 Cache Based Architectures

For a detailed discussion of the design issues in cache based architectures, see [8]. Here we will review some of the issues in memory systems design and briefly discuss the impact on algorithm design/scheduling.

In actual fact, main memory is typically produced from *Dynamic Random Access Memory* or DRAM technology.

1.6.1 Cache Characteristics

The difference in speed between memories and processors has led to the introduction of the *cache*, a relatively small but fast buffer memory between the main memory and the CPU and its register set. The idea is that if most (all) of the references go to the cache, then the memory appears as fast as the cache. However, if the data is not in the cache, it must be retrieved from the memory.

Data transfers between the cache and memory are typically performed as block transfers. The block is often called a *cache line*. Thus, if the data is not in the cache, then retrieving the data from memory may actually be slower than a direct memory access, since a whole cache line is being replaced.

Some cache terminology:

Hit. The desired data is in the cache.

Miss. The desired data is *not* in the cache.

Hit rate. The fraction of references for which the data is in the cache.

Miss rate. The fraction of references for which the data is *not* in the cache.

Hit time. The CPU time required to acquire data from cache.

Miss time. The CPU time required to acquire data when not present in the cache. This time is usually divided into two parts: access time and transfer time. The latter is proportional to the cache line size.

Example

In this example we determine the hit rate required for a cache based system to break even with a direct memory access system. Assume that cache hits require 1 cycle and that for each word that needs to be accessed from memory the following characteristics apply [8]:

Line (block) size	4 – 128 bytes
Hit time	1 – 4 cycles
Miss time	8 - 32 cycles
Access time	(6 - 10 cycles)
Transfer time	(2 - 22 cycles)
Miss rate	1 – 20%
Cache size	1 KB – 256 KB

Table 8: Some typical cache characteristics.

- 1 CPU cycle to send address
- 6 CPU cycles to access each word
- 1 CPU cycle to transfer a word to the cache

For a four word cache line, 32 cycles are required. Thus, with a single word access requiring 8 cycles and a hit rate α , the break even rate is

$$8 = \alpha \times 1 + (1 - \alpha) \times 32$$

or

$$\alpha = 0.774$$

Thus, for a hit rate better than 77.4% the cache based system is faster than the direct memory access system. In the worst case it would be four times slower, and in the best case eight times faster.

In our example, we ignored the impact of cache replacement policies. Loading a new cache line implies that a cache line is overwritten. In a *write-back* policy the replaced cache line is stored in memory, further aggravating the performance penalty for a cache miss. For further details see Chapter 8 of [8].

Some typical cache data from [8] are summarized in Table 8 below:

1.6.2 Locality of Reference

Locality of reference is critical to the value of a cache. We distinguish between two forms of locality of reference:

Temporal Locality. If an item is referenced, it will be referenced again soon.

Spatial Locality. If an item is referenced, nearby items will be referenced soon.

Note that both notions of locality of reference are inherently dependent upon the scheduling of computations.

Example: Matrix–vector multiplication $y = Ax$.

First, we consider the matrix A . In the algorithms we considered earlier, each element of A is only read once. There is no temporal locality of reference for any element of A . Since the address space for single processor systems traditionally is linearized, spatial locality of reference is possible if an algorithm is chosen such that stride in the inner loop is one. For instance, if the stride within a column is one, then a SAXPY (DAXPY) algorithm exhibits locality of reference, while an inner product algorithm will not (for a matrix of a reasonable size). If the stride within a row is one, then the behaviors are reversed.

Note however, that even if the stride within a row is one, then a vectorized and unrolled SAXPY (DAXPY) algorithm may exhibit locality of reference. After a column segment equal to the length of the vector registers is traversed, a new column segment of the next column is traversed. With an unrolling depth equal to a multiple of the cache line size, a cache line will be fully utilized once loaded into the cache.

With respect to the input vector, a SAXPY (DAXPY) algorithm offers maximum temporal locality of reference. With matrix columns processed in order, the spatial locality of reference is also maximal.

Thus, we conclude that some of the restructuring techniques suitable for vector-register architectures may also be excellent for cache based architectures.

1.7 Parallel memories

Determining the bank number through the operation $A \bmod B$, where A is the linearized address, yields a *cyclic* allocation of data to the banks. We notice that the cyclic allocation does not preserve locality of reference well for computations such as the solution of Poisson’s equation. For the two–dimensional problem we discussed earlier, successive grid points along the axis labeled first, say the horizontal axis, are in adjacent memory banks. Thus, in a multiprocessor system in which there is a processor for each bank, the cyclic allocation scheme results in excessive communication. As we will see later, a *consecutive* allocation scheme, is preferable with respect to preserving locality of reference for the solution of Poisson’s equation by Jacobi’s method.

In High Performance Fortran, HPF, designed for multiprocessor, distributed memory computers, both cyclic and consecutive data allocation can be specified, as well as combinations thereof.

In conclusion we notice that there is a tradeoff between preserving locality of reference on a distributed memory architecture, and avoiding bank conflicts in interleaved memory systems.

References

- [1] Marco Annaratone. *Digital CMOS Circuit Design*. Kluwer Academic Publishers, 1986.
- [2] Rajendra V. Boppana and C. S. Raghavendra. Generalized schemes for access and alignment of data in parallel processors with self-routing interconnection networks. *Journal of Parallel and Distributed Computing*, 11(2):97–111, 1991.
- [3] P. Budnik and David J. Kuck. The organization and use of parallel memories. *IEEE Trans. Computer*, 20:1566–1569, December 1971.
- [4] Richard Comerford and George F. Watson. Memory catches up. *IEEE Spectrum*, 29(10):34–35, 1992.
- [5] Mike Farmwald and David Morning. A fast path to one memory. *IEEE Spectrum*, 29(10):50–51, 1992.
- [6] Richard C. Foss and Betty Prince. Fast interfaces for DRAMs. *IEEE Spectrum*, 29(10):54–57, 1992.
- [7] Stein Gjessing, David B. Gustavsson, David V. James, Glen Stone, and Hans Wiggers. A RAM link for high speed. *IEEE Spectrum*, 29(10):52–53, 1992.
- [8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1990.
- [9] S. Lennart Johnsson and Luis F. Ortiz. Local Basic Linear Algebra Subroutines (LBLAS) for distributed memory architectures and languages with an array syntax. *The International Journal of Supercomputer Applications*, 6(4):322–350, 1992.
- [10] Fred Jones. A new era of fast dynamic RAMs. *IEEE Spectrum*, 29(10):43–49, 1992.
- [11] David J. Kuck and Richard A. Stokes. The Burroughs Scientific Processor (bsp). *IEEE Trans. Computers*, C-31(5):363–376, May 1982.
- [12] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. on Computers*, 24(12):99–109, 1975.
- [13] Duncan H. Lawrie and Chandra R. Vora. The Prime Memory System for array access. *IEEE Trans. Computer*, 31:435–442, May 1982.
- [14] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM TOMS*, 5(3):308–323, September 1979.
- [15] Christoffer Lutz, Steve Rabin, Charles L. Seitz, and Donald Speck. Design of the mosaic element. In *Proceedings, Conf. on Advanced research in VLSI*, pages 1–10. Artech House, 1984.
- [16] Carver A. Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

- [17] John Newkirk and Robert Mathews. *The VLSI Designers Library*. Addison-Wesley, 1983.
- [18] Ray Ng. Fast computer memories. *IEEE Spectrum*, 29(10):36–39, 1992.
- [19] Abhiram Ranade. Interconnection networks and parallel memory organization for array processing. In *1985 International Conference on Parallel Processing*. IEEE Computer Society, 1985. Report YALEU/DCS/RR-422.
- [20] Roelof H.W. Salters. Fast DRAMs for sharper TV. *IEEE Spectrum*, 29(10):40–42, 1992.
- [21] Charles L. Seitz. Concurrent VLSI architectures. *IEEE Trans. Comp.*, 33(12):1247–1265, 1984.
- [22] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.0*, 1992.