Executive Summary

This thesis investigates the design of a high-performance parallel SIMD system designed for low-precision integer operations. The hypothesis underlying the research is that a reconfigurable bit-slice approach can be substantially more efficient than conventional bit-serial organizations and more flexible than hardware bit-parallel. Preliminary support for this conclusion has been obtained from a simplified model of hardware and algorithms. The goal of the remaining work is to confirm the conclusion by:

- designing and testing a high performance chip,
- completing a (paper) system-level design that will allow the chips to operate at a high sustained rate,
- implementing a number of SIMD algorithms on a simulator to demonstrate performance on real applications,
- using the instruction traces from the algorithms to make detailed architectural evaluations and comparing simulation results to an analytical model,
- designing a second-generation architecture that builds on the lessons of the initial implementation, with an eye to technology trends.

The organization of the thesis (and of the proposal) essentially follows the outline presented above.

Chapter 1: Introduction and Motivation. Describes the computational requirements of early vision and some parallel processing approaches to achieving these requirements.

Chapter 2: Previous and Related Work. There is a substantial body of previous work relating to the topics of this thesis, from SIMD machines to modern architectural alternatives: the SRC PIM chip, Berkeley's PADDI-2 DSP, and FPGA based computing platforms. After the architectural discussion, this chapter evaluates related analytical performance models.

Chapter 3: Reconfigurable Bit-Parallel Architecture. Describes in detail the motivations for an RBP organization, RBP arithmetic algorithms, and specifics of the Abacus design.

Chapter 4: The Abacus-1 Chip. Describes the implementation of the Abacus-1 chip at a fairly detailed level, as well as chip test results.

Chapter 5: System-Level Issues of a High-Speed SIMD Machine. Describes the issues of a high-speed SIMD machine and presents a relatively detailed design of the Abacus computer, including I/O and control issues.

Chapter 6: Parallel Vision Algorithms. Describes a set of vision and communication algorithms chosen both for application value and for architectural evaluation. These algorithms include

several from the DARPA Image Understanding Benchmark suite. Discusses the performance of Abacus on these algorithms.

Chapter 7: Architectural Tradeoffs. Evaluates architectural tradeoffs such as local memory size, ALU width, off-chip memory bandwidth and network bandwidth based on instruction traces from the parallel algorithms.

Chapter 8: The Next Generation: Abacus-2. Describes a set of modifications that allow a redesigned Abacus element to function in multiple-SIMD mode, as a systolic processor, or to emulate random logic circuits effectively.

Chapter 9: Conclusions. The contents of this chapter will depend (somewhat) on the results of the actual research.

Schedule

Task	Time	Start	End
Test board fabrication	21	Sep 11	Oct 1
Test board assembly	3	Oct 1	Oct 4
Test board/C30 interface	3	Oct 5	Oct 7
Test vector design	5	Oct 8	Oct 12
Chip testing	7	Oct 13	Oct 19
Area Exam	30	Oct 15	Nov 15
Simulator coding	14	Nov 17	Dec 1
Architectural analysis	21	Dec 1	Dec 22
Text writing	30	Dec 22	Jan 12
Thesis defense	7	Jan 13	Jan 20

1 Introduction and Motivation

Computer vision, the understanding of images by a computer, is both an exciting component of artificial intelligence and a daunthing computational task considered as one of the Grand Challenge problems of parallel processing. The design of a parallel computer for vision is an attractive problem for a computer architect, as the problem domain lends itself to optimizations that promise orders of magnitude performance over general-purpose parallel computer.

This thesis proposes to design, implement, and analyze a VLSI-based architecture tailored to the computing demands of early vision. Although the field has received a great deal of academic attention, this work targets a considerably higher speed design than previous efforts, and as a result encounter system-level consequences of a high speed parallel computer. In the body of this section we will the describe the problem domain and the detailed thesis approach.

Computing Requirements of Vision. Computer vision deals with the understanding of imagery by a computer, in contrast to image processing, which generally enhances images for human inspection. The field can be broadly categorized into three levels: low (or early) vision, which extracts physical properties of viewed surfaces, such as distance, orientation, and velocity; middle vision, which manipulates geometric quantities and performs simple recognition; and high level vision, which uses symbolic reasoning to disambiguate objects. Boundaries between each of these levels are rather blurred, especially as recent object recognition research has started to operate at the image level. Of the three, early vision processing is the best understood and therefore the best candidate for acceleration with a specialized architecture. The others are still highly experimental and require flexible general-purpose architectures.

The need for a specialized computer becomes apparent when the computational requirements of early vision are considred. A moderately sized image contains approximately 65,000 pixels, each of which may be used a hundred times in a computation. A single vision algorithm may require ten iterations per fimage, and as many as fifty algorithms may be required to extract essential information such as depth, motion, shapes, and shadows. All of this processing must be repeated thirty times per second to sustain real time response. The aggregate sustained computing power is therefore on the order of 100 billion operations per second (GOPS). It is clear that parallel processing is required to deliver this level of performance.

The question now becomes: what should the parallel system look like? There have been hundreds of proposed parallel system designs. These systems can be broadly categorized as Single-Instruction Multiple Data (SIMD) machines that consist of identical processing elements (PEs) executing the same instructions in lockstep, and Multiple-Instruction Multiple-Data (MIMD) machines, whose PEs execute independent programs and use explicit communication to synchronize operations. Historically, SIMD machines have been built with large numbers of simple PEs, while MIMD machines relied on a small number of complex PEs.

At the time of writing, the most prevalent parallel architectures are MIMD machines consisting of collections of workstation-class RISC processors. Indeed, only one major SIMD vendor, MasPar, still exists, while there are many MIMD vendors, including Cray, Convex, IBM, and Intel. This overwhelming commercial success is due to the constant improvement in microprocessor performance, driven by marketplace pressure and fuelled by enormous investment.

Parallel MIMD machines are excellent general–purpose computing and development platforms, since they come equipped with the software inherited from their workstation heritage. As with the uniprocessors on which they are based, these machines excel at computations with irregular and unpredictable instruction streams. Unfortunately, the price of this flexibility is reduced performance. On problems with a data–parallel, highly regular structure, such as image processing, these processors continue to be unnecessarily flexible at every instruction step. When additional domain constraints such as low-precision arithmetic and 2-dimensional mesh communication are considered, a specialized architecture promises order-of-magnitude improvements over general purpose systems.

The proposed thesis develops a design for Abacus, a SIMD machine specialized for variable-precision integer operations on a dense 2-dimensional mesh, and targeted for a high-speed VLSI implementation. Predicted pperformance of Abacus parts is indeed approximately 50 times higher than that of uniprocessors with comparable silicon resources.

Reconfigurable Bit-Parallel Processing. The Abacus architecture can be classfied as a reconfigurable bit-parallel (RBP) machine. Reconfigurable processors provide a smooth transition between the two extremes of a slow, small bit-serial PE and a complex, fast bit-parallel PE. The tradeoff between the two is that simple one-bit PEs are capable of arbitrary precision and can run at a high clock speed, but require many cycles per operation. Hardware bit-parallel PEs require few cycles, but run slower and waste valuable silicon when computing with data narrower than their ALU width. RBP designs hope to retain bit-serial flexibility and bit-parallel performance.

The key idea of a RBP design is to provide programmable interconnected to link multiple PEs into a single Processing Site (PS) that can operate on sveral bits of a data word simultaneously. Reconfiguration allows precision to be matched to the need of each algorithm. Even more significantly, on-chip memory capacity at each processing site can also be adjusted by changing the number of PEs in a PS. Since off-chip memory references can be a factor of 20 slower than on-chip accesses, decreasing the miss rate at the cost of ALU capacity may lead to substantial performance increases for many algorithms. There is some overhead associated with RBP, such as the time to combine or distribute information from the entire word, as in conditional operations. As with hardware bit-parallel ALUs, operations on data narrower than the PS size (such as boolean flags) are inefficient.

Thesis Approach. This thesis proposes to analyze, quantify, and demonstrate the performance advantages of a high-speed VLSI RBP SIMD architecture on early vision tasks. The overall research approach will be to:

1. Develop an analytical performance model accounting for both architectural and algorithmic effects.
2. Design, implement, and test a VLSI chip containing Abacus processing elements. A working chip will both provide architectural parameters for an analytical model and unearth any inaccurate assumptions about high speed SIMD operations.
3. Implement a simulator and code a number of vision algorithms including a standard image understanding benchmark suite to determine algorithmic parameters.
4. Complete a system-wide (paper) desgin to ensure that there are no unexpected system-level bottlenecks.
5. Use the IC implementation experience and the analytical model to suggest a design for a second generation architecture.

The expected contributions of this thesis can be grouped under three categories: implementational, analytical, and cultural. A successful implementation will demonstrate a SIMD chip with the highest clock rate to date. It will force the development of system interfaces generally ignored in the academic field. This development is absolutely essential as input/output interfaces may prove to be the bottleneck in high speed designs.

The analytical model will allow the selection of an optimal point in implementation space as technology parameters vary. For example, novel high-bandwidth devices such as Rambus memory components may allow reduction in the amount of on-chip memory. The model will allow quantification of these tradeoffs. Also, in the course of obtaining algorithmic parameters, the vision algorithms will be systematically characterized, which may lead to improved intuition about optimizing algorithms for RBP architectures.

Hopefully, the thesis will also have several positive cultural effects. It will to provide a convincing demonstration of the effectiveness of SIMD processing. We believe important commercial applications of SIMD technology are near but may be hampered by the unconvincing performance and architectural deficiencies of existing SIMD processor arrays. Vision-related tasks, such as MPEG coding, OCR, paper document processing, digital picture manipulation, medical imagery, face recognition, vehicle collision avoidance, and automatic vehicle guidance all provide important practical applications which can be solved using SIMD techniques. An existing system design will allow the construction of a cheap tera-op level supercomputer for vision research. Such a platform will allow important image, signal, and text processing applications to be prototyped. Finally, a successful design will encourage research on specialized architectures, which has been decreasing recently due to the success of RISC-based MIMD machines.

2 Previous and Related Work

2.1 Architectural Design

There has been a considerable amount of research on architectures for image processing and image understanding. A variety of processor designs and interconnection networks have been proposed (and far fewer have been implemented). This review concentrates on SIMD designs and on alternate approaches with comparable performance.

Early examples of machines designed for image processing include Goodyear's MPP[1], NCR's GAPP [2], and ICL's DAP. The most widely produced SIMD machine was Thinking Machines' CM-2. A contemporary commercial product was produced by MasPar [3], and recently upgraded to the MP-2 [4].

There are also several research designs which have not been put into commercial production. These designs explored different aspects of SIMD architectures. For example, IBM's polymorphic torus [5] concentrated on adding connection autonomy by the addition of locally reconfigurable network switches. The MCNC Blitzen project [6] updated the original MPP design for VLSI technology by adding on-chip RAM, local modification of addresses, and an X-grid 8-neighbor interconnect. A unique Some/None network intended for associative processing was stressed in the CAAAP architecture [7] developed at the University of Massachusetts at Amherst. As in the polymorphic torus, network switches were locally controlled, but groups of processors could be electrically connected. This feature allowed efficient connected components and global statistics algorithms.

The idea of an RBP architecture is not completely new. Even the first massively parallel machine, the DAP, allowed columns of PEs to be configured into a ripple-carry chain. This capability was not central to the architecture and operated slower than bit-serial operations. The reconfigurable processor array (RPA) [8] uses a mixed approach to RBP. Although each PE processes two bits at a time, and a data word is shared among a PS, considerable hardware resources are devoted to supporting bit-slice operations. As a result, each PE is quite complex for a fine-grained machine.

A number of recent research efforts promise to achieve substantially higher performance than these early designs. A 256-element SIMD array driven by an on-chip RISC-like processor has been designed at the University of Sheffield [9]. The performance is reasonable, given the number of processors, but is limited by the instruction issue rate of the RISC controller, the low I/O bandwidth, and the lack of off-chip memory support. A major US semiconductor manufacturer[1] is readying an as yet unannounced SIMD processor design with 50 MHz clock rates and sixty-four 8-bit processors for use in embedded multimedia applications. The Supercomputer Research Center has developed a processor-in-memory chip which augments a 128 Kbit SRAM array with bit-serial processors at each row [10]. The performance of this design is rather low due to the small number of PEs and modest clock rate.

An alternate processor-in-memory MIMD/SIMD architecture has been developed by IBM [11]. The Execube chip incorporates 8 16-bit PEs, each with 64 KB of DRAM memory, and can be operated in SIMD mode from a global instruction bus. The performance is again rather low due to the long cycle time of DRAM and because instructions are stored in the same memory as data.

An integrated micro-MIMD chip, integrating 48 16-bit processors and an interconnection network on each chip has been designed at UC Berkeley [12]. The IC delivers an impressive 2.4 GOPS on a variety of DSP algorithms and is designed to communicate efficiently with other chips. However, since each PE's instruction memory is only 8 words deep, with no provision for expansion, the architecture is not suitable for more complex algorithms.

A class of DSP chips, most notably the Texas Instruments C40 and the Analog Devices SHARC processors, have been designed to support efficient execution over data arrays and inter-processor communication. These appear more promising than the RISC based systems, but still lack performance and waste power.

Another technology that delivers comparable performance to Abacus and PADDI is the use of reconfigurable logic as exemplified by Field Programmable Gate Arrays. An array of programmable logic can be used to configure applications pecific hardware and thereby obtain excellent performance. For example, research at DEC Paris have implemented algorithms ranging from Laplace filters to binary convolutions [13]. If the per-chip performance is computed as aggregate performance divided by the number of FPGA chips in the system, each Xilinx chip delivers approximately 500 million 16-bit operations per second.

It is unfair to directly compare the 16-bit performance of Abacus compared to the wider-width Execube and MasPar PEs, as they incorporate floating-point support. However, even granting a factor of five in area, their performance is still lagging. Both of these designs make the classic mistake of single-porting their register files. The implication of requiring three cycles to perform a single instruction is that three times the area could have been allocated to triple-porting the memory. And the area in this case refers to the *entire* silicon area, including communication pads and support circuitry.

The only architecture comparable in performance is the Berkeley PADDI-2 chip, and it is not suitable as a general computing element as it can only store eight instructions per PE. Of course, when the computation task can be expressed as piping data through a systolic array, the IC is an ideal high-performance, low chip-count solution.

[1] Psst: Motorola

Machine	Performance 16-bit MOPS	Clock MHz	I/O BW MB/sec	Mem BW Internal	Mem BW External	Tech μ
TMC CM-2	8	8	8	32	16	1.5
SRC TeraSys	20	10	5	80	5	1
IBM EXECUBE	50	25	300	400	0	.8
TI C40	50	50	200	0	200	.8
MasPar MP-2	133	12.5	45	1600	45	1
HP8000	200	200	800	0	960	.5
Sheffield DIP	320	40	160	2560	0	1
Xilinx XC3090	500	20	NA	NA	NA	1
Berkeley PADDI-2	2400	50	800	14400	0†	1
MIT Abacus-1	**2000**	125	1000	32000	512	1

Table 1: Architectural performance comparison

We are not aware of any other research SIMD machine that even approaches the Abacus design in terms of performance per silicon area on low-precision integer operations. Our advantage occurs for a variety of reasons, including an aggressive clock rate of 125 MHz, made possible by careful custom design, high density manual circuit layout, and the inherent advantage of RBP operations.

2.2 Architectural Analysis

Two recent research efforts are most relevant to the proposed work. The performance advantages of RBP architectures due to memory limitations were first evaluated by Audet et al [14], who referred to RBP as the Dynamic Concatenation Approach. The goal of the research was to determine the optimal number of PEs to be grouped into a PS, given particular hardware and algorithmic parameters. This number is called the concatenation factor, w.

Although the critical issue of off-chip memory traffic minimization was addressed, the published work contained some shortcomings. Some of the flaws were:

- Ignored communication time growth with w. For example, on a mesh with square processing sites, time grows as $\sqrt{w}$.
- Scaled execution times by factors based on pipelining effects. The method of deriving these parameters is based on arbitrary assumptions about compilers and hardware implementation. Quantifying pipelining effects applies more to a VLIW-style system where multiple functional units can be triggered by a single instruction. The model appears to apply to multi-cycle, relatively complex functional units, which are not usually present in simple, small SIMD PEs.
- The model assumes a static type of register assignment: the most frequently used variables are placed into registers, and the rest into memory locations. Modern compilers perform dynamic assignments, spilling registers into memory locations when necessary. As the methodology used UNIX-based C profiling tools to determine the often referenced variables, it is not clear whether the need for temporary registers was considered, or whether loop unrolling optimizations were made.
- Virtualization requirements were not considered and program segments appear trivially short. Further, the only program example shown in detail is of a systolic type. Short computation

chains between virtual PE synchronizations could lead to a large number of register save and restore operations.

A simulation approach to SIMD performance evaluation was taken by Herbordt [15]. Traces are generated on an abstract virtual machine and then *trace compiled* for a specific target architecture. The technique can be two orders of magnitude faster than a detailed simulation. This flexibility allows quick evaluations of different architectural parameters, such as register file size or communication latency.

Although the centerpiece of the thesis is the trace compilation methodology, a number of architectural studies were run, and some interesting conclusions resulted. For example, some studies showed that increasing ALU width substantially improved performance only up to a width of 8 bits. After that, performance improved only slightly.

The key differences between the proposed work and Herbordt's approach are:

- Effect of trace compiler quality eliminated through hand-compilation. Since vision algorithms tend to be oblivious, with little data-dependent branching, they can be characterized more exactly than through simulation.
- Algorithms designed with Abacus architectural features in mind. For example, if multiplies are expensive, the implementation may avoid them, or at least specify required precision. Otherwise, unnecessary computations can affect benchmark conclusions.
- Intent is to derive parameters for an analytical model. Possibility of better insight into algorithm structure.
- Use of explicitly allocated off-chip memory rather than cache. This could be approximated by ENPASSANT with a fully associative cache and a LRU replacement policy.

An overall problem is that the simulator simulated algorithms designed for a uniform memory access machine, which is no longer appropriate for today's high-speed designs.

3 Reconfigurable Bit-Parallel Architecture

This section describes the basic idea of RBP, provides an argument of its advantage over a bit-serial organization, describes the Abacus PE architecture, and some RBP arithmetic implementations.

3.1 The Basic Idea of RBP

The basic idea of RBP is the concatenation of several processing elements into a processing site. The advantages of the technique can be viewed from several angles. First, if an algorithm requires many data bits per pixel, then the silicon area of the ALU is small compared with total data area, and therefore adding a few more ALUs will improve performance linearly, with a tiny increase in total area. This interpretation assumes that all data is kept on chip and physically near the ALU.

Alternately, RBP may be viewed as a method of adding the appropriate amount of registers to a processing site to minimize the frequency of expensive off-chip accesses. The improvement in performance is due mostly to decreased stall time rather than additional ALU bits.

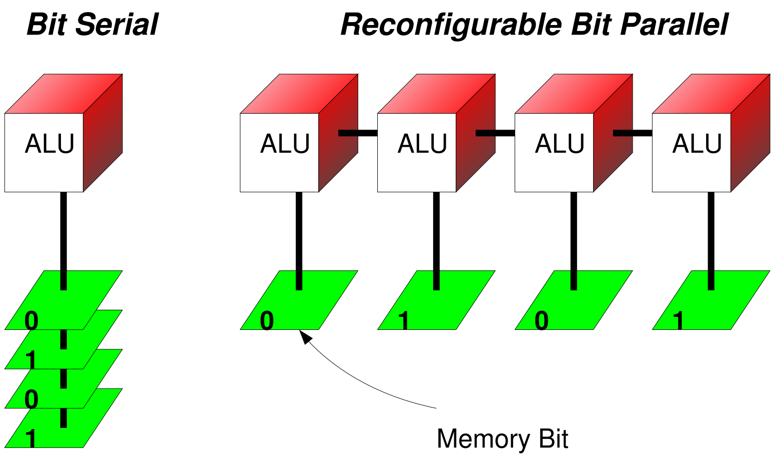

Figure 1: A bit-serial organization compared to a reconfigurable bit-parallel one. Each of the RBP processors is also capable of acting as a bit-serial unit, if appropriate.

3.2 RBP Performance Model

This section quantifies the regime in which the RBP organization is superior to a bit-serial one. We assume that a processing site, be it a bit-serial PE or a group of RBP PEs, requires a certain amount of on-chip storage to operate without frequent off-chip memory accesses. If this requirement is not met, performance will degrade catastrophically due to the limited off-chip bandwidth.

In the following discussion, let B be the number of bits in a data word; W, the number of words in the register file; V, the area of the non-memory overhead (ALU and network) component of the PE; and S, the number of cycles required by the bit parallel operation. Area measurements are normalized to the size of one memory bit. A bit-serial organization requires B cycles to step through each bit of a data word and occupies $V + BW$ area. An RBP PS requires S cycles and occupies $B(V + W)$ area. The silicon efficiency can be expressed as performance per unit area, or simply as $1/AT$. The ratio of efficiency, bit-parallel to bit-serial, is:

$$R = \frac{V + BW}{S(V + W)}$$

For the vision algorithms we are interested in, computations on 8-bit pixels are done with 16 bits to preserve fractions, and as many as 64 words are frequently accessed. In our implementation of the PE, V was approximately 50. Our algorithms have S ranging from 3 to 6, averaging around 4. A graph of R versus W, for relevant values of B, S and V is shown in Figure 2.

The important conclusion is that not only does the RBP approach result in faster (lower-latency) computation, but that it is also more efficient in terms of computational power per silicon area. Of couse, this can been as arguing for hardware bit-parallel.

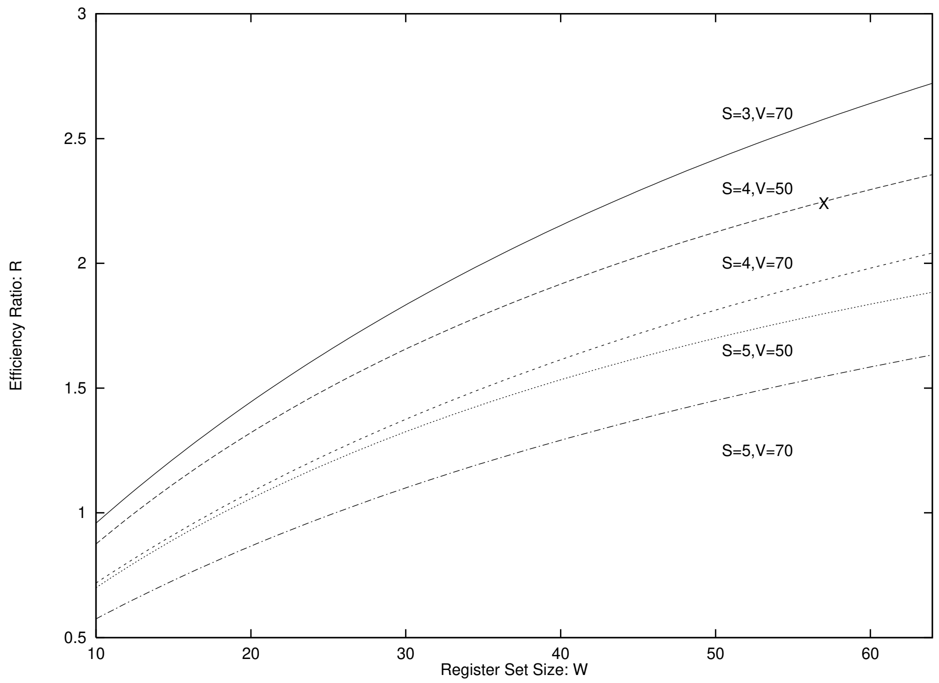

Figure 2: Efficiency ratio R vs register file size W, for various values of non-memory overhead V and algorithmic inefficiency S. The 'X' indicates the design point of our implementation.

3.3 Abacus PE Architecture

The Abacus processing element (PE) consists of 64 1-bit registers organized into two banks. There are two 3-input ALUs, each of which takes two inputs from its memory bank and one input from the other bank. The four available data bits allow complex boolean functions, and two result bits can be written in each cycle, one bit to each bank.

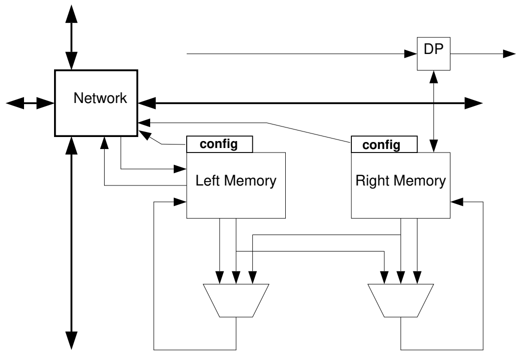

Figure 3: Processing Element Block Diagram.

A PE also has a 1-bit network interface and background I/O interface. Seven of the registers are used for control, leaving 57 general-purpose registers.

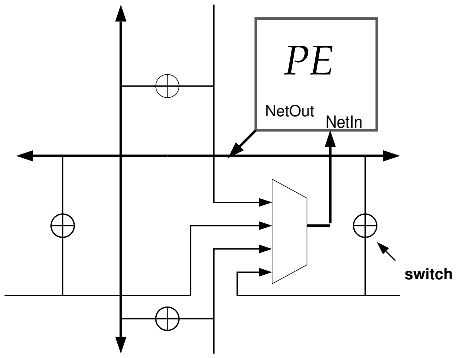

Figure 4: Network

Activity Register Conditional execution of an `if (cond) then ... else ...` sequence on SIMD arrays operates by disabling those PEs for which `cond` is true, executing the true branch, disabling the other set of PEs, and executing the false branch. One of the Abacus PE registers serves as the `activity` bit which, when cleared, disables computation by inhibiting the write of the result to the destination register.

The ability of the ALU to serve as a multiplexer greatly reduces the use of the activity register. For example, the sequence `if` A `then` $B \Leftarrow C$ incurs two cycles of overhead if implemented with the activity register. Alternately, the operation can be expressed as $B \Leftarrow \mathtt{mux}(A, C|B)$, eliminating all overhead.

Network The reconfiguration network serves for both bit-slice interconnection and inter-word communication. The network is a wired-OR reconfigurable mesh. As in a conventional mesh topology, each PE listens to the output node of one its four mesh neighbors. Each PE is provided with a configuration register that specifies which neighbor to listen to. Unlike a mesh, each PE can connect its output node to the selected neighbor's output node. Shorted nodes behave as a wired-OR bus: if any PEs in a connected chain send a 1, all PEs listening to the chain receive a 1.

Background I/O Each PE contains a data plane (DP) register, used for background off-chip data transfers. The DP registers of PEs in a column of the array form a 32-bit shift register. At the edge of the PE array, the 32 shift registers are connected to an off-chip memory port. These registers can be shifted without interfering with PE computation. Although the DP register is not used in arithmetic operations it is essential in hiding the latency of external memory accesses.

3.4 RBP Algorithms

RBP arithmetic algorithms resemble circuits more than bit-serial programs, since they are unrolled in both space and time. During each time step, each PE emulates part of the logic comprising an arithmetic circuit, reconfiguring connections if necessary. In the examples of this section, PSs are shown organized as a line in order to resemble a circuit. In reality, clusters are often arranged as squares or rectangles (a square organization minimizes the average inter-cluster communication distance), and topologically form a ring. In arithmetic circuits, boundary logic cells such as the LSB and MSB are configured differently from the middle cells. Similarly, PEs corresponding to cluster edges are marked as MSB or LSB. In addition, each PE is labeled with its bit position within the cluster. The LSB/MSB bit can be computed from this position, as can network configuration bits.

Logical Shift. The simplest RBP arithmetic operation is the shift. As shown in Figure 5, each PE simply replaces its bit with its neighbor's. For logical shifts, the MSB or LSB is cleared; for arithmetic shifts the MSB is retained.

Sum Accumulation. Accumulating a sum is a very common operation. The sum of a sequence of n numbers can be evaluated in $\Theta(n)$ cycles with a carry-save (CS) algorithm. The CS adder computes the sum and carry bits separately for each addition. The computation is purely local, and the delay of each CS stage is independent of word size. At the end of a summing sequence the carry and sum values must be summed with a full carry-propagation adder, but this delay is amortized

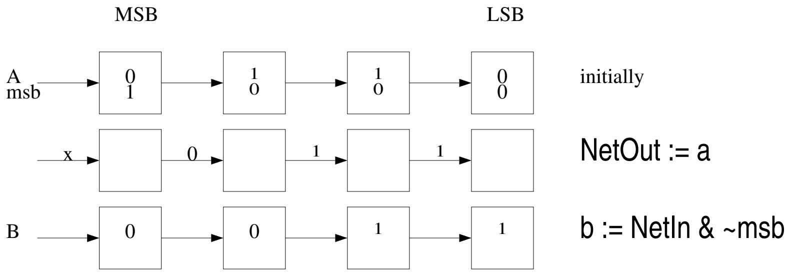

Figure 5: Logical shift right. The number 6 is shifted down to become 3.

over many summations. Many computations usually expressed as additions can be reformulated as accumulations, improving the performance of algorithms such as region summing. It is interesting to note that accumulation is an inherently simpler and faster operation than addition, and yet conventional processors do not make the distinction.

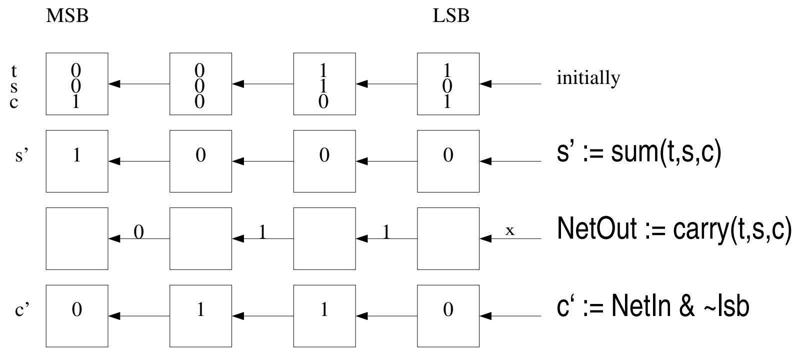

Figure 6: Accumulate. The addition of the number 3 to a sum of 11 represented as 2+9. After the operation, the sum is 14, represented as 6+8. All operations are purely local or nearest neighbor. Steps 1 and 2 can happen during the same cycle if a copy of t is present in both memory banks.

Addition. Fast addition relies on quick computation of the carry bit. A RBP algorithm based on the the well-known logarithmic-time technique was described in [16]. The algorithm "looks-ahead" at the carry by computing the eventual carry into each one-bit adder. It can be shown that the carry c_k into bit k can be expressed as $c_{k+1} = g_k + p_k c_k$, where g_k is the *generate* bit and p_k is the *propagate* bit. Since the computation of successive p and g values can be expressed in terms of a binary associative operator, it can be performed in time logarithmic with the number of bits. This technique forms the basis of many hardware implementation of adder circuits.

The Abacus architecture uses a different approach, one based on the hardware implementation of Manchester carry computation. In this technique, the carry bit propagates electrically down the line of bits unless prevented by a switch controlled by the p signal. Although computation time is linear with the number of bits, the constant factors result in faster operation than software emulation of the logarithmic-time circuit.

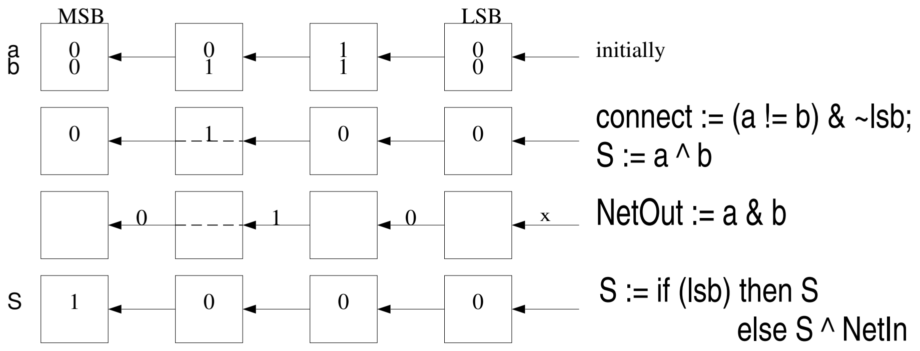

Figure 7: Manchester-carry addition. A NOP cycle occurs after step 2 to allow the carry bit to propagate. Steps 1 and 2 are actually merged into one cycle, and the `S` bit can be computed during the NOP cycle.

Match. The match operation compares two bit patterns for equality. First, all PEs form a wired-OR bus. Then, all PEs write `a != b` to the bus. If any bits differ, the bus becomes a 1; if all bits are identical, the bus remains at 0.

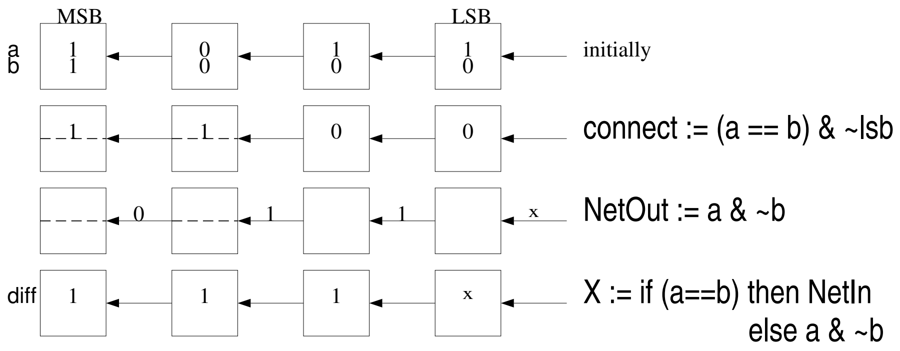

Figure 8: The `compare` operation. After the broadcast, the MSB PE receives A¿B for the rest of the word. It then merges the result with the MSB bit result.

Comparison. The comparison algorithm is based on the observation that the most significant differing bit (MSDB) between two words determines which word is greater. Thus if the MSDB occurs in position i, $A > B$ when $A_i > B_i$, or equivalently $A_i = 1$ and $B_i = 0$. In the first step of the algorithm, all PEs with identical bits bypass themselves and do not participate in the rest of the computation. Then, all PEs send $A \cdot \overline{B}$ towards the MSB PE. Now, the MSB PE knows the value of $A > B$ for the rest of the word, and merges the result with its own data bits to produce the final answer. This entire process is equivalent to performing a Manchester-carry subtraction and testing for the borrow bit at the MSB.

Multiplication. RBP integer multiplication is composed of iterating through a sequence of shifts, broadcasts and carry-save adds. These operations can be partially overlapped so that an iteration requires 7 cycles. Thus, a 16 by 16 multiply requires 112 cycles, plus about 10 cycles of overhead. These numbers are for unsigned multiplication with a double-length result. If only the low word of the result is required, an iteration requires only 6 cycles. Two's complement multiplication is implemented with a software version of the Baugh-Wooley multiplier, and requires one additional iteration. While hardware multipliers frequently use the modified Booth algorithm to halve the number of iterations, this algorithm is not very efficient on the Abacus architecture since it requires the broadcast of three bits.

Mesh Move. In a mesh move each cluster rearranges itself into several shift registers in the direction of movement. The move operation is relatively expensive, as k bits must be transferred through a $\sqrt{k}$-wide bus, requiring $\sqrt{k}$ time steps. Further, unlike arithmetic operations, mesh moves cross chip boundaries and therefore suffer an additional 1-cycle latency cost.

Scan Operations. A set of computational primitives known as *parallel prefix* or *scan* operations have been found to be useful programming constructs across a variety of machine architectures. These operations take a binary associative operator $\oplus$, and an ordered set $[a_0, a_1, \ldots, a_{n-1}]$ and return the ordered set $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots a_{n-1})]$. Common scan operations include *or-*, *and-*, *max-*, *min-*, and *+-scan*. There are also corresponding *reduce* primitives that reduce the values in a vector to a single value. For example, *+-reduce* calculates the sum of a vector, as shown in Figure 9.

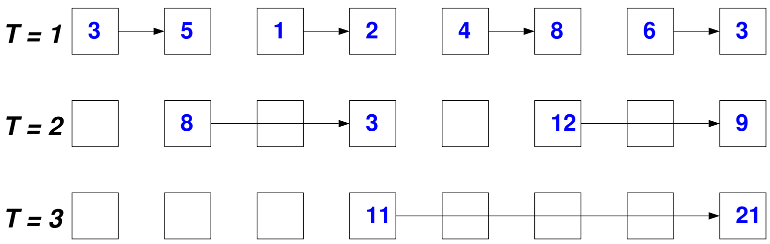

Figure 9: A +-reduce operation.

The bypass capability of the Abacus network can be used to implement fast scan operations. A message can propagate across a chip in two cycles. Crossing a chip boundary requires an additional cycle of latency. The prototype is a 16 by 16 chip array, so that 48 cycles are required for a bit to cross the array. Assuming a 4 by 4 bit PS organization, propagation of a 16-bit cluster requires 200 cycles. Since each stage of a logarithmic scan doubles propagation time, an entire scan operation

	Cycle Count			MOPS		
Operation	8-bit	16-bit	32-bit	8-bit	16-bit	32-bit
Add, Compare	4	4	5	4000	2000	700
Shift	3	3	3	5300	2700	1300
Accumulate	4	4	4	4000	2000	1000
Mesh Move	5	6	8	3200	1300	500
Multiply	66	126	235	242	63	17

Table 2: Single-chip arithmetic performance

requires approximately 500 cycles, or 4 microseconds. This can be optimized further by pipelining bit transmission.

3.5 Performance Summary.

The performance of a single Abacus chip on the operations described in this section is given is in Table 1. The left half of the table gives cycle counts, while the right gives the MOPS rating assuming 1024 PEs per chip and a 125 MHz clock rate.

These ratings are somewhat pessimistic for two reasons. First, they include an additional one-cycle penalty for reconfiguration. For example, if a right shift instruction is followed by an add instruction, PEs must be reconfigured from reading from the MSB direction to reading from the LSB direction. This penalty does not occur during every instruction, and a clever compiler can group instructions with identical configurations to reduce reconfiguration costs.

Second, as discussed earlier, microcode for several arithmetic operations can be overlapped. By the table entries above, the sequence $a = b + 2d + 2c$ performed on 16-bit values appears to require 14 cycles (two shifts, an accumulate, and an add). Yet a handcoded program performs the same operations in only 9 cycles by eliminating three reconfiguration steps, overlapping two broadcasts, and duplicating a data bit in both memory banks.

A high fraction of peak performance has been observed on simulations of vision algorithms.

4 The Abacus-1 Chip Implementation

4.1 PE Core

Processing Element Almost half of the PE's area is consumed by registers, which are implemented as standard 6T SRAM cells and arranged into two banks of eight rows and four columns. The `bit` and `/bit` lines are used single-ended to provide two read ports, allowing four bits to be read and two written in each cycle. Bitline precharging to $V_{dd} - V_t$ is done through the column selects, precharging only bitlines that will be read. Each ALU is implemented as a precharged 8-to-1 multiplexer that uses three data bits to select one of eight instruction bits. During each 8 ns cycle, a PE performs a bitline precharge, SRAM read, ALU evaluate, and SRAM write. The background

DP register can be shifted simultaneously without interfering with the ALU. The simplicity of each PE resulted in a compact, 83μ by 452μ implementation in a 2.6μ minimum metal pitch technology.

Network The on-chip interprocessor communications network consists of a precharged output node and four isolating switches. The node is a simple precharged inverter whose input is driven by the `NetOut` register. The four switches are NMOS gates driven by a 2-4 precharged decoder. The precharge pulse is controlled by software to allow multi-cycle propagation times.

Propagation delay across a series of pass transistors is usually quadratic in the number of devices. Our implementation greatly reduces this delay by a local accelerator circuit at each node which regeneratively pulls a node to ground as soon as the node voltage drops by a transistor threshold. The circuit is a dynamic NORA style circuit and is well suited to the precharged operation of the network. We found it to be several times faster than an implementation based on a complementary inverter. At the nominal process corner, simulations show that a bit propagates through 18 PEs in one 8 ns cycle.

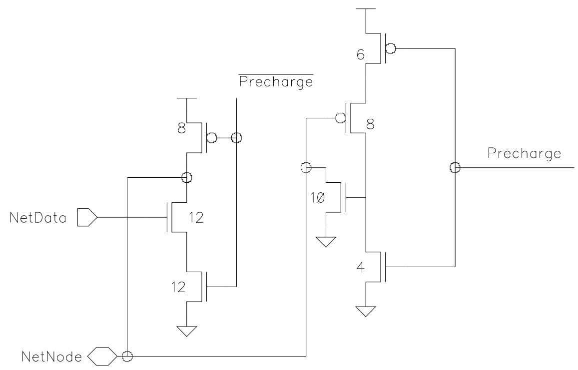

Figure 10: Network accelerator in the NORA circuit style.

This circuit has a noise margin of V_T, so careful layout was done to reduce coupling capacitance to the pre-discharged node. Further, the chip is provided with on-chip bypassing and over 200 power and ground pins to reduce dI/dt noise.

Chip Organization. The chip is organized into three major parts: the PE array, split into two halves; the control signal drivers in the middle; and control logic at the left. The control logic includes address decoders, instruction decode, external memory control, and a scan path controller. The PE core is organized as 16 strips of 64 PEs each. Each strip also contains two sets of 76 buffers that drive control signals outward from the center. Skew is reduced by distributing signals vertically in a lightly-loaded metal-3 bus, buffering horizontally with identical drivers, and equalizing the load on each horizontal control wire.

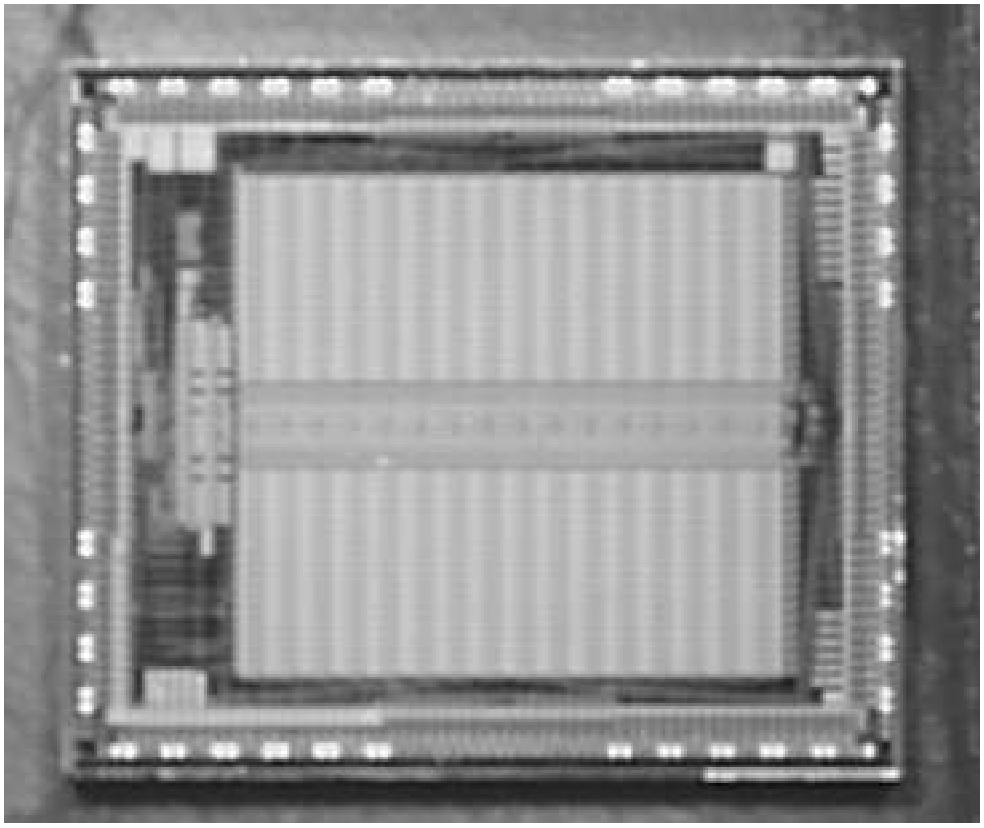

Figure 11: Chip photograph.

Input/Output Interface The Abacus PE chip has a one-bit single-ended ECL input and a one-bit differential ECL output for external data I/O. The aggregate bandwidth of a 256-bit, 125 MHz bus formed by the 256 PE chips is 4 GB/sec, which is almost three orders of magnitude than the 7.7 MB/sec required by 512 by 512 images at 30 Hz. This very high I/O bandwidth makes the architecture well-suited to applications requiring real-time processing of large amounts of data such as video and synthetic aperture radar.

The I/O bandwidth can be scaled down to reduce the interface logic and wiring cost. For example, if only the 16 chips comprising the top row of the 256-chip array are connected, the 4 GB/sec bandwidth would be reduced to 250 MB/sec, which is more than an order of magnitude higher than frame rate.

The Abacus PE chip contains several mechanisms required for high-speed system-level operation: skew-tolerant instruction distribution pads, high-bandwidth local DRAM interface, low voltage swing impedance-matched mesh pads, and low pin-count I/O interface.

4.2 Off-Chip Interfaces

All Abacus off-chip interface circuits were designed by Tom Simon. Their implementation does not constitute part of the thesis.

4.2.1 System Synchronization

Careful synchronization is essential to a high-speed SIMD architecture. On the Abacus PE board the system clock is distributed in differential ECL to each PE chip. Propagation delays are matched by snaking traces so that trace lengths are equalized. To compensate for skew introduced by process variations in the on-chip clock distribution, each chip phase locks its internal clock to the received clock. Phase locking is performed by a delay locked loop (DLL) based on an all-digital delay line. The DLL uses the simple and accurate arbiter circuit shown in Figure 12. The arbiter has a perfectly symmetric 30 ps uncertainty window at nominal process.

The remaining question is: which of the internal clock phases should be locked to the system clock? There are several internal clocks: instruction register, PE register read, network precharge, etc. We choose to synchronize to the clock that drives the mesh output pads. As a result, off-chip communication is synchronized system-wide.

4.2.2 Instruction Distribution

A spatially distributed SIMD system must ensure not only that all clocks are synchronized, but also that all chips are executing the same instruction at the same time. Clock distribution can be controlled by snaking traces to equalize propagation delay, or by using specialized clock distribution parts with programmable delays. Neither technique is feasible for the 30-bit instruction bus running at 250 MHz. Since each PE board is approximately 60 cm on a side, even with signals driven from the center of each board, distribution skew can be as large as 2 ns.

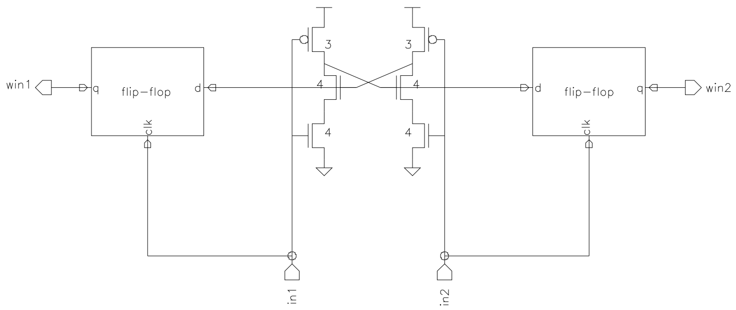

Figure 12: Arbiter used in the delay-locked loop

Another source of instruction skew is the variability of discrete parts in the distribution path. For example, the clock to output delay on a high-performance ECL shift register has an uncertainty of 0.4 ns. A signal traversing two parts can be skewed by a much as 0.8 ns. The combined effect of these skews is that some chips receive an instruction a cycle later than others, even though all chips are supposed to be executing the same instruction at once. Worse yet, bits of an instruction can be mixed with bits of the next instruction due to parts variability.

This skew problem is solved by retiming logic in the instruction pads that synchronizes instruction delivery across all chips. Retiming occurs as part of the system startup process and proceeds in three stages. The first finely adjusts the delay of each bit until the sampling clock occurs in the middle of the data valid period. The second stage lines up all bits within a chip to the same pipeline stage. Finally, pipeline delays are equalized across the system by software. The various stages of the initialization process are controlled through a scan path by the array controller.

4.2.3 Inter-Chip Mesh Communication

The next system level challenge is to provide synchronous high-bandwidth, low-latency mesh communication between PE chips. Due to pin limitations, the edges of the 32 by 32 PE mesh on each chip are time multiplexed onto 16-bit ports. The required clock rate is therefore 250 MHz.

Each of the four mesh ports sends out an escort clock along with the data. Since the off-chip drivers and propagation time are identical for data and clock, the receiving chip can safely use the escort clock (delayed by the setup time of the receiving registers) to sample the data. Metastability problems do not arise as we ensure that data is sampled well before the receiving data needs to use it.

The communication pads are driven with custom 1V, on-chip series terminated drivers and matching receivers in order to minimize power dissipation and transmission line reflections [17]. Impedance calibration for the series termination is performed at system initialization. Each of the four ports on a chip is calibrated independently, since on-board and board-to-board impedances may differ.

In addition to the pipelined, high bandwidth mode of operation, the mesh pads can operate in a flow-through mode that bypasses the on-chip pulldown network with a single wire, allowing fast multi-chip broadcast algorithms. Propagation through a chip is reduced from three clock cycles to half a cycle. However, the time multiplexing is disabled, reducing the effective bandwidth.

4.2.4 Image I/O

While early SIMD designs were concerned with limited image I/O bandwidth, it is not an issue today's high-speed designs. Since I/O is distributed among 256 chips, a single pin on each component results in a very wide bus. The Abacus PE chip has a one-bit single-ended ECL input and a one-bit differential ECL output for external data I/O. The aggregate bandwidth of a 32-byte, 125 MHz bus is 4 GB/sec, which is almost three orders of magnitude than the 7.7 MB/sec required by 512 by 512 images at 30 Hz. The I/O bottleneck is clearly going to occur upstream of the PE array.

The I/O bandwidth can be scaled down to reduce the interface logic and wiring cost. For example, if only the 16 chips comprising the top row of the 256-chip array were connected, the 4 GB/sec bandwidth would be reduced to 250 MB/sec, which is more than an order of magnitude higher than frame rate.

A single PE instruction initiates a burst transfer of 32 bits from the edge of the PE mesh to the output pin (or vice versa). This background transfer frees the array to perform computation, or in the case of the lower cost interface, to use the mesh connections to shift data towards the connected chips.

4.2.5 External Memory Interface

Since many algorithms will exhaust the 64 Kbit on-chip memory, each PE chip is backed by two 1M x 32 specialized DRAM modules. The data interface is 64 bits wide and operates at 16 ns, which matches the 32-bit, 8 ns PE mesh. The 16 ns cycle time is made possible by the Ramtron extended DRAM part, which contains an SRAM cache integrated in the DRAM.

Control signals for the memory chips are synchronized to the PE chips by sending them as part of the instruction stream. Fine control over DRAM timing is made possible by the small instruction cycle time, which is comparable to that of dedicated DRAM controller. A useful side-effect of this approach is the elimination of memory glue logic. A counter in each PE chip generates the external memory addresses. This counter can be loaded from the top row of the PE array, allowing indirect addressing.

5 System-Level Design

This section is empty for now. It will contain relatively detailed design of the entire system, including sequencer, I/O interfaces, and host workstation interfaces. The topics of this section have largely been ignored by academic researchers, with most of the attention going to processing element and interconnection design. However, system interfaces are precisely those elements most likely to cause bottlenecks.

6 Parallel Algorithm Implementation and Performance

This section describes the implementation of several vision and communication algorithms on the Abacus architecture. It will eventually contain images processed by the high-level Abacus simulator, as well as characterizations of the inner loops.

6.1 Basic Vision Algorithms

Edge Detection The widely used Marr-Hildreth edge detection algorithm consists of filtering the image with a Gaussian filter, computing the Laplacian of the filtered image, and locating the zero crossings of the result [18]. Convolution with a Gaussian can be approximated by repeated convolution with a triangular filter, with weights $\frac{1}{4}$, $\frac{1}{2}$, and $\frac{1}{4}$, requiring only arithmetic shifts and accumulates. Since the Gaussian filter is separable, it can be implemented by two one-dimensional convolutions. Further, since zero crossings are independent of absolute magnitude, the expensive scaling operation of the Laplacian kernel is not required.

Figure 13: Edge detection with the $\nabla^2 G$ filter. (a) original image; (b) smoothed with a Gaussian, 2 iterations; (c) smoothed with a Gaussian, 4 iterations; (d) and (e) zero crossings of $\nabla^2 G$.

Surface Reconstruction Surface reconstruction consists of determining the surface shape (height and slope) from a set of potentially sparse and noisy measurements. Harris [19] introduced the coupled depth/slope model for surface reconstruction and developed an iterative solution technique suitable for a mesh-based massively parallel computer. The update equations in each iteration are simple and local and consist of five or six additions and subtractions, followed by a division.

The summation of multiple values at a pixel is computed efficiently by the multi-operand addition algorithm. Division is expensive on a DBP architecture, as it is on a word-parallel bit-serial machine. This algorithm only requires division by a constant, which can be implemented by a special purpose sequence of shifts and adds.

Optical Flow In correlation-based optical flow [20] the original image is displaced in a variety of directions, and at each displacement the difference between the two images is computed, and summed over a window. If the images are processed to produce binary image features, the difference is the exclusive or of the image pixels in the 11 by 11 neighborhood. Finally, each pixel chooses the layer with the smallest difference value in a winner take all step. Images manipulated by the algorithm may consist either of simple brightness values or of more complex features such as edges. This algorithm has been implemented on the simulator for the binary feature case. The edge detection algorithm discussed earlier could be used to obtain the features (in this case edges) from a raw intensity image.

(a) (b)

Figure 14: Optical flow calculation with maximum displacement of 2 pixels and a 5 ×5 summation region. (a) original image. (b) displaced image. The top shape moved up by one pixel and partially off the image; the right shape moved down by one pixel and the lower shape moved three units down and one to the left.

Performance Summary In the vision algorithms table, computation times are shown for a 128 by 128 array processing 256 by 256 pixel images (a virtualization factor of four).

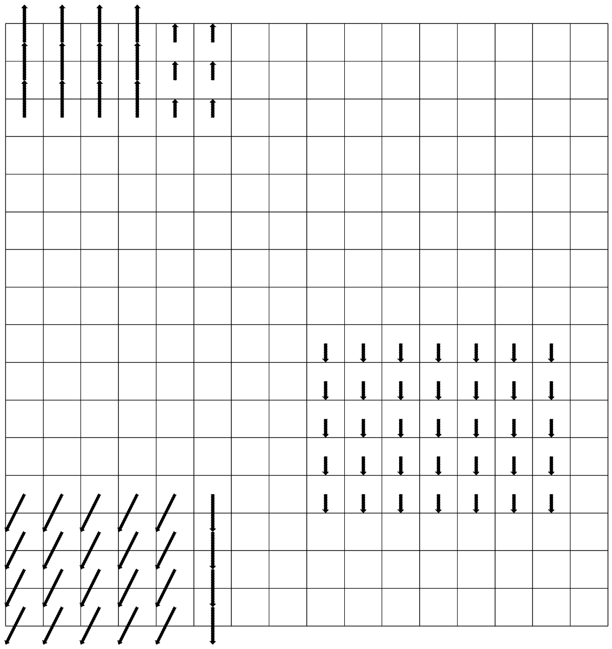

Figure 15: Optical flow field resulting from the raw data in Figure 14. Some confusion is caused by the movement of the top shape off the image, and by the movement of the lower shape by a distance exceeding the maximum displacement layer.

Algorithm	Cycles 1000s	Time (μsec)
Edge Detection $\sigma = 2.0$	1.8	14.4
Optical Flow, $\Delta = 5$, 5 $\times$5 region	40	320
Surface Reconstruction (1 iter)	1.5	12

6.2 Early vision subset of the DARPA IU Benchmark

This section describes the subset of the DARPA Image Understanding Benchmark that falls in the domain of early vision.

Connected Component Labelling: Broadcast A common problem in image analysis involves labeling the connected regions of constant pixel intensity, where unique regions are constituted from a definition of either 4-connectedness (horizontally or vertically adjacent), or 8-connectedness (horizontally, vertically, or diagonally adjacent). This operation is discussed for binary images in the literature ([21], [22], [23], [24], [25]) but has been extended for grayscale values in the following three algorithms. The first approach described here is the most straightforward but requires the most time, the second alternative is faster but requires considerable amounts of memory, and the third is a modification of the second that compresses memory requirements drastically, at the cost of increased computation.

In this algorithm, each pixel of the intensity image is labeled uniquely from its PE row and column. This intensity and its corresponding label (of $O(\log N)$ bits) are broadcast to each of its 8-connected neighbors. At each PE, the neighbors of matching intensity are determined, and the minimum of the their corresponding labels and the current label becomes the new label. This process continues until no more labels in the mesh are updated (checked by a global compare). The number of broadcasting operations required is proportional to the largest "intrinsic diameter" of all connected components in the image, defined as the maximal shortest connected path between any two pixels in the region [22]. For spirals and other high-curvature images, this intrinsic diameter can be as high as $O(N^2)$ in the worst case, resulting in $O(N^2 \log N)$ bit operations.

Connected Component Labelling: Shrinking Levialdi's region shrinking operation [21] provides a iterative method to directionally compress each region down to a single pixel and then remove it entirely without fragmenting or fusing separate regions. If the results of each operation are saved away, the operation can be reversed to generate a unique label when a region consists of only one pixel, and that label can be transmitted to all possible neighbors in the direction of expansion so that they can make a decision should they become connected in the next stage. The third algorithm modifies the basic one by storing only a subset of the shrunk images and reconstructing on the fly by repeating the shrinking operation.

K-curvature Tracking and Corner Detection The connected components map is processed to produce K-curvature values for those pixels on the component borders, which are then smoothed with a Gaussian filter to eliminate multiple peaks near corners. Pixels with smoothed curvature values exceeding a threshold value (peaks) are intersected with the zero-crossings of the first derivative of smoothed curvature to extract candidate corners in the image. Border pixels are defined to be any pixel adjacent (N,E,W,S) to a pixel belonging to another component. K-curvature is defined at each border pixel as the interior angle between the two lines passing through the current pixel and those K border pixels away in either direction along the region's border. See Figure 16.

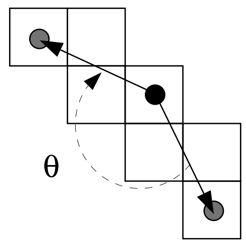

Figure 16: K-curvature definition. In this figure, θ is the K-curvature for K=3. Note that all pixels shown are edge pixels.

Median Filter The median filter is a common image processing operation. Unlike linear filters, each pixel replaced not by a linear combination of its neighbors but rather by the median value. The operation is effective for removing high frequency "speckle" noise without degrading the rest of the image.

Gradient Magnitude By computing the magnitude of a discrete intensity gradient in the image and thresholding the result, strong direction-independent edges in the map can be located. Every pixel intensity in a 3x3 neighborhood about the current PE is multiplied by a weight according to the Sobel X and Y masks and summed to form the Sobel X and Y magnitudes. Since the weights are either 0, 1, or 2, the multiplications are converted to shifts.

The gradient magnitude is the square root of the sum of the squares of the X and Y magnitudes. Results greater than a threshold value are flagged to create a boolean edges map. The operation is constant in space and time with respect to N.

Hough Transform The Hough transform partitions a binary image into discrete bands one pixel thick and oriented at a certain angle, and sums the values within the band to yield a set of projections that can be scanned to locate strong edges in the original image at that angle [26]. Usually transforms are computed for many angles, so that the aggregate data can provide insights into the image properties.

The implemented algorithm partitions the image into bands of constant line offset ρ for a given angle θ according to the equation: $\{(x, y) : x\cos\theta + y\sin\theta = \rho\}$ where (x, y) are PE coordinates and θ is assumed to be in the range $\pi/2 \leq \theta < 3\pi/4$ (the other angles can be accommodated by pre-rotation of the image). A "band total" variable visits all PEs in its assigned band by shifting east, and north if necessary (a result of the angle range is that at most two pixels in the same column belong to the same band). The PEs in the first column are visited first, and then the variable travels eastward across the columns. Since there are many angles to be projected, they are pipelined one column at a time, yielding P projections on an NxN image in $O(N + P)$ time.

6.3 Sorting and Routing Algorithms

Sorting algorithms can be converted into routing algorithms by associating data with the sort key and ensuring that the combined packet is moved together. This allows packets to be routed to a destination. Two types of sorting algorithms were examined: those with a fixed execution time, and those with a data-dependent execution time.

ShearSort and RevSort ShearSort is a $\sqrt{N}(\log N + 1)$ step sorting algorithm [22]. It consists of sorting rows in alternating opposite directions starting at the left edge of the array, then sorting the columns from top to bottom. ShearSort relies on the TranspositionSort [27] procedure to get things sorted. TranspositionSort is the basic linear array sort. It works similar to the uniprocessor BubbleSort: comparing then exchanging (where necessary) alternating pairs of items.

An improvement of ShearSort is called RevSort [28]. It is identical to ShearSort, except CyclicSort is used to sort rows instead of Transposition Sorts. This algorithm finishes in $\sqrt{N}(\log\log N + 3)$ steps and the complexity of the each step is equivalent to each step of ShearSort. CyclicSort is a descendant of the TranspositionSort, and performs as fast as its ancestor. The smallest item is sorted to a chosen processor instead of the leftmost processor.

Mesh Greedy Routing Algorithm A very different packet-based routing algorithm was developed by Herbordt [29]. Each PE emulates two communication channels, one vertical and one horizontal. Packets are moved one step through the X channel until the correct X coordinate for the packet is reached. At that point the packet is moved to the Y channel and proceeds vertically. The X and Y routing steps are interleaved, so that one of each occurs during each iteration. Notice that packets can be blocked from switching to the Y channel if that section of the channel is full. When this occurs, packets behind the blocked PE in the X direction are also blocked. The blocking information propagates backward along the X channel one step at a time. Since each PE contains space for two packets, collisions will not overwrite data, and the blocking information can afford to propagate only one step per cycle.

The key difference from the user's point of view is that this algorithm is not guaranteed to finish in $O(\sqrt{N})$ iterations. In fact, the worst case performance is $O(N)$, where N is the number of PEs. According to Herbordt, the worst case is very unlikely to arise in either a completely random, or typical routing conditions. In fact, the algorithm generally does not need to be iterated more than 2.5 times the optimal case of $2\sqrt{N} - 2$. The paper demonstrates that this property holds for a number of common permutations, such as transpose, bit shuffle, and rotations.

7 Architectural Tradeoffs

In this section we begin to defvelop a performance model for the Abacus computer.

7.1 Slice Optimization

In addition to the usual architectural and algorithmic parameters, the RBP organization allows a compile-time selection of the concatentation factor m. This is the number of PEs grouped into a single PS. We would like to determine the optimal number of bitslices operating for a particular algorithm. Assume the following constants:

Name	Description
	Architectural
D	Number of registers in slice
k	Slice width
B	Bandwidth into each slice (per bit)
	Algorithmic
R	Number of variables in algorithm
W	Bits in data word
N	Number of ops in sequence
Z	Number of answers and initial data items
	Derived
f	Fraction of register misses (function of R,E)
E	Effective registers per pixel
	Compiler Decision
m	Number of slices / pixel

Table 3: Architectural and Algorithmic Parameters

Thus, a processing site consists of m slices of D registers each of k bits. There are W bits in each data word. The number of effective registers per pixel is the total number of bits in a processing site divided by the word length:

$$E = \frac{mkD}{W} \tag{1}$$

The time to load a word/operate on a word is equal to the number of bits to be loaded divided by the cumulative bandwidth/ALU width of the slices doing the loading:

$$T_{load} = \frac{W}{mBk} \quad , \quad T_{alu} = \frac{W}{mk} \tag{2}$$

So we model an operation sequence on a single pixel as loading some initial number Z_{in} of words from off-chip memory, performing N arithmetic operations, and writing Z_{out} result words out, for a total of Z loads and stores. Equation 3 gives the time required for processing a single pixel, assiuming that for every memory fault the processor has to write the previous value to memory and load the requested value. This memory fault is going to occur on some fraction f of the memory accesses.

$$T_{seq} = ZT_{load} + 2Nf(R,E)T_{load} + NT_{alu} \tag{3}$$

Assume that there are P pixels to be processed and M slices to process them. If m slices are allocated to a pixel, M/m pixels are processed in a time step of T_{seq}, so that Pm/M iterations are required. It is convenient to define the quantity $l = Z/N$, the ratio of initialization operations (forced cache misses) to iterative operations (possible cache hits).

$$\begin{aligned} T_{tot} &= \frac{P}{M} m T_{seq} \\ &= \frac{P}{M} m \left[(Z + 2Nf(R,E)) \frac{W}{mBk} + N \frac{W}{mk} \right] \\ &= \frac{PWN}{MBk} [(l + 2f) + B] \\ &\propto (l + 2f) + B \end{aligned}$$

The simplified model presented in this chapter does not consider the disadvantages of RBP, such as the overhead of PEs whose ALUs are unused, or the distribution of conditional information to each PE. Therefore, the obtain optimal concatenation factor w will always be too high. Neverthless, the performance improvement is an upper bound on the advantage of RBP.

So, we would like to choose m to minimize T_{tot}, given the algorithmic constraints of W, N, Z, f; the architectural constraints of B and D. We could evaluate this exactly, except that we do not know the behavior of the miss function $f(R,E)$. So we'll explore some plausible functions.

1. *Case 1: No Cache Misses.* If $R < E$, we have enough local storage and $f = 0$. Q then becomes independent of m. This indicates a bad initial design point: both in the nibble-serial mode and in the nibble-parallel mode, all data fits in memory. Therefore, the memory is too big.

2. *Case 2: All Cache Misses.* If the algorithms steps repeatedly through all the registers, the miss rate is effectively 100%. Again, the number of nibbles allocated is irrelevant.

3. *Case 3: Random Register Usage.* If the register used during each step is selected at random from the logical registers, there is a E/R probability that it is already in the cache. f is therefore
$$f(E,R) = 1 - \frac{E}{R} = 1 - \frac{mkD}{WR} = \frac{WR - mkD}{WR} \tag{4}$$

   So, to maximize performance we want to lower f and therefore increase m.

In conclusion, m should always be maximized.

7.2 Background Loading

A memory access on the Abacus machine has a latency of 40 cycles, but can be initiated before the data is actually required. Only one outstanding memory reference can be pending. We would like to determine the benefit of this feature compared to the worst-case penalty of 40 cycles. The key parameters are the latency of a reference, and a distribution probability on consecutive non-memory references Our first order model assumes a uniform probability p of an off-chip memory reference. Thus, the probability of a sequence of k on-chip references before an off-chip reference is:

$$P(N = k) = (1 - p)^{k-1} p \tag{5}$$

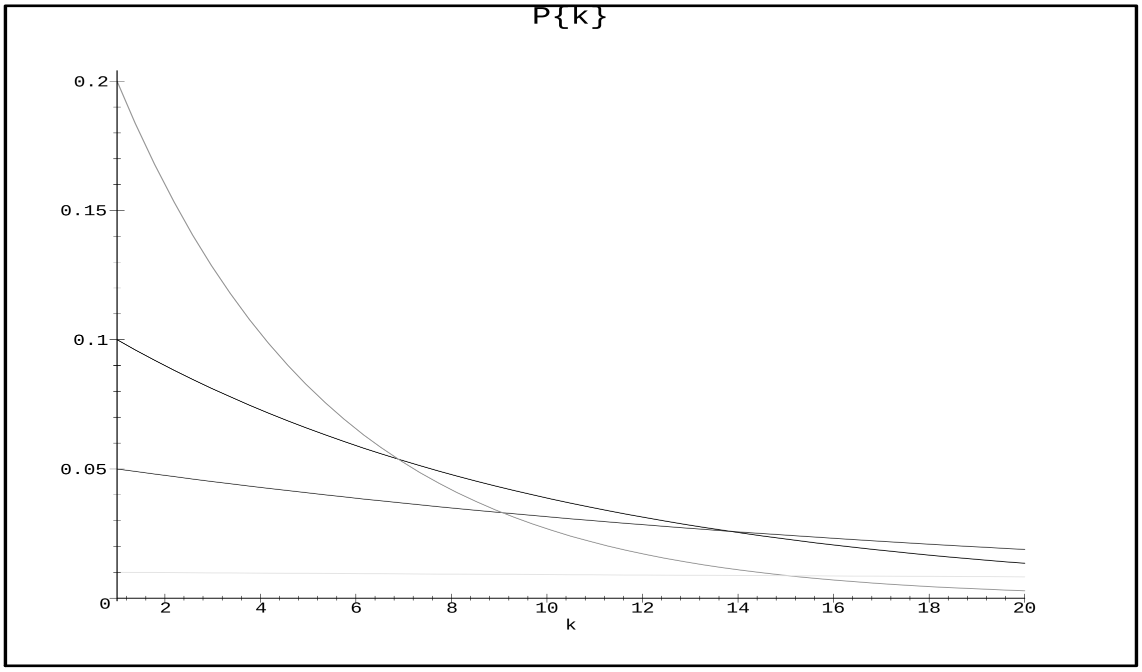

Figure 17: Probability of an on-chip hit sequence of length k for different values of p, the miss probability. Note that for high p almost all the area under the curve is near small values of k, and therefore long load times.

This distribution is shown in Figure 17. Notice that the probability of long sequences is small if p is high. Therefore, we expect background loading to help only in the case of low p or low latency. Since the loading time given a string of k available cycles is $L - k$, the expected loading time is:

$$E[T] = \sum_{k=1}^{L} q^{k-1} p(L - k) = \frac{(1 - p)^L + p\,L - 1}{p} \tag{6}$$

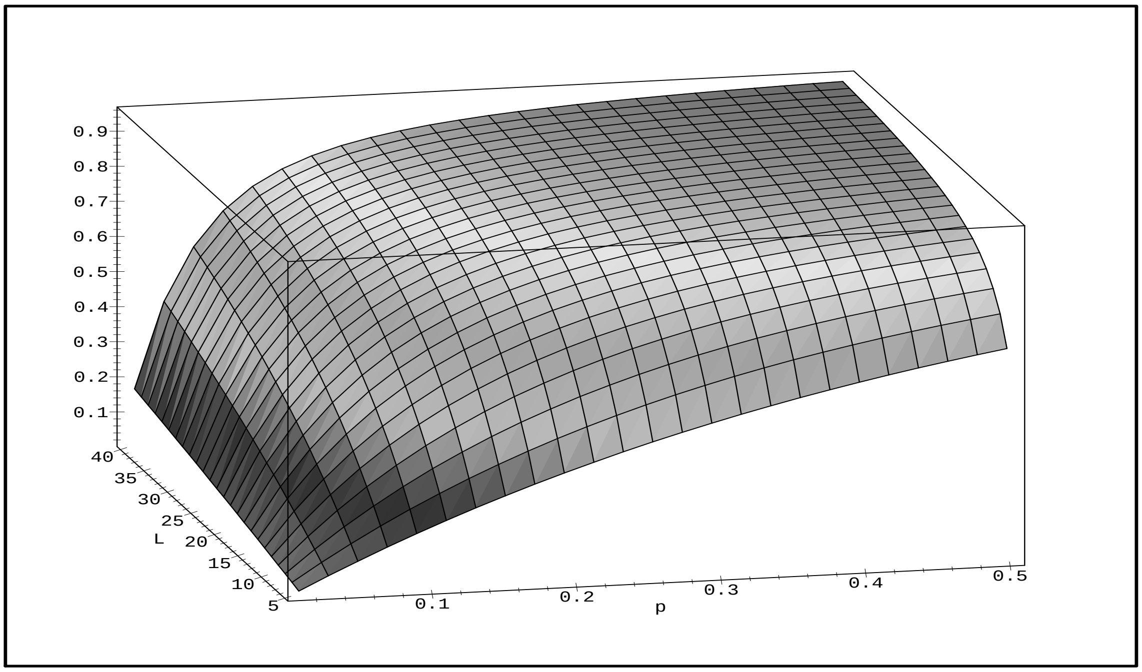

Figure 18: Effective latency vs latency due to background loading as a function of L and p. The advantage of background loading occurs in a relatively narrow zone: where the frequency of loads is very low to begin with, and where latency is low. For low values of L, the ratio may not be really important, since it approaches internal access time.

Still, even for relatively large of L and p^2, we can get a 3X speedup.

7.3 Conclusion

The following plots are for parameter settings of:

Variable	Meaning	Value
Dr	number of registers in slice	50
k	slice width	1
Bw	bandwidth into each slice (per bit)	0.05
R	registers in algorithm	30
Wb	bits per word	16
Z	initial loads/saves	10
N	number of operations in a sequence	30

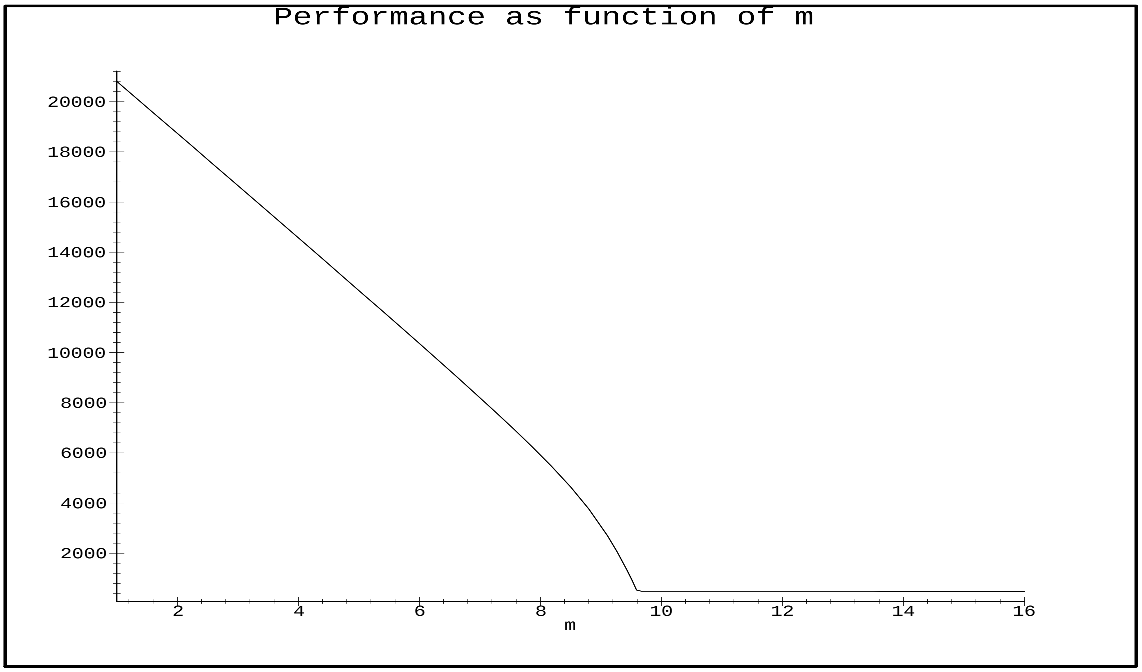

Figure 19: Execution time as function of m. The dip in the middle is the effect of background loading. Without it, the graph would have continued in a straight line. Notice that the dip occurs at a non-useful m value, as it is difficult to evenly partition a 16-bit value.

The algorithm requires 480 bits. At 9.6 slices per pixel, there are no external references and time is constant. Background loading makes a difference only in a narrow zone. Until about $m = 7.8$, The

[2] 40 and 0.1, respectively

linear performance improvement is due to the linearly decreasing hit frequency in our simplified model.

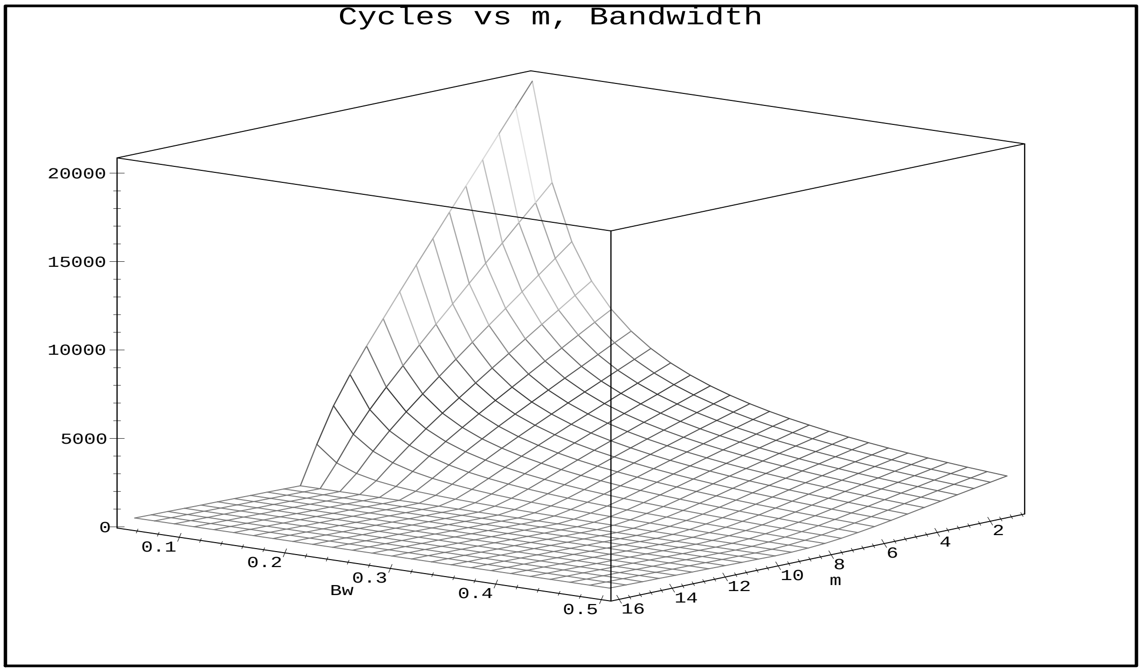

Figure 20: Execution time as function of m and bandwidth

The main conclusion is that external memory references are catastrophic, and that the number of bitslices should be chosen to minimize the number of external loads.

The model can be made more accurate in a number of ways.

1. Accounting for the circuit density improvement of higher-width bit slices.
2. Speed decreases due to larger ALU widths. Of course, this may not matter if the cycle time is dominated by memory access.
3. Slower memory accesses with increasing memory size.
4. Better estimates of cache miss behavior as a function of available registers.

As illustrated by the graphs in this section, the last modification is probably the most significant, since the initial three factors are not likely to be more than a factor of 2 each, while cache misses can result in order of magnitude performance changes.

8 Next Generation: Abacus-2

8.1 Lessons of Abacus-1

We have learned a number of lessons from the implementation of Abacus-1 that we believe will result in a substantial performance improvement in the Abacus-2 design.

1. *Reconfiguration overhead is excessive.* The design of the on-chip interconnection network requires a reconfiguration cycle if the direction of data flow changes. This occurs when, for example, an arithmetic shift right instruction, in which bits flow from left to right is followed by an add instruction, in which the carry bit flows from right to left. The single-cycle reconfiguration still represents a substantial cost.

2. *Arbitrary data formats are expensive.* The one-bit bit slice elements could be grouped in almost arbitrary configurations, as long as each set of PEs responsible for a data word formed a topological circle. As a result, on-chip data could not be accessed by the sequencer without a complex corner-turning reformatting procedure.

3. *Rise/fall times are a substantial fraction of cycle time.* When rise/fall times approach 400 picoseconds, three or four non-overlapping control signal transitions consume as much as 3 ns. When combined with process variation uncertainties, the actual active time shrinks further.

4. *Simple ALUs do insufficient work per cycle.* The very simple ALU operates in approximately 400 picoseconds, a tiny fraction of the overall cycle time.

5. *Lack of pipelining is critical.* Due to the nature of the network design, forwarding is essentially impossible and pipelining the read, execute, and write phases is precluded. At this point it appears that pipelining is sufficiently important to modify the network organization. A pipelined design would allow more time for various execution phases and therefore reduce the size and power dissipation of control line drivers.

6. *Address control wires are expensive.* Although the double-banked register files and twin ALUs (see Approach section) allow two result bits to be written per cycle, the number of control wires doubles compared to a single bank. This leads to unnecessary layout constraints, high capacitance, and difficulty in pitch-matching signal buffers to the PE height. Further, the required board-level instruction bandwidth increases substantially.

7. *ECL components are terrible for integration.* We designed the instruction distribution and I/O paths to use ECL signaling. This was a mistake, since many ECL-based LSI components are required per board, as well as pulldown resistors, bypass capacitors, and a higher number of board layers. The design of a custom multi-function CMOS part would have been preferable.

8. *Burst I/O is overconstraining.* The Abacus chip has two pins for I/O, one for input and one for output. A single I/O instruction initiates a 32-bit burst. This design choice was intended to increase the overlap of computation and I/O, but was a mistake in retrospect. High performance parts are required to handle the 32-bit, 125 MHz bursts, yet they sit idle almost all of the time, as I/O transfers are very rare.

Our preliminary design for the Abacus-2 part addresses these issues, among others. We expect to almost double arithmetic performance while reducing power dissipation and decreasing the required packaging cost. The Abacus-2 chip will fit 256 4-bit PEs on a single 10x10 mm die, and operate at 66 MHz in a 3.3 V (or lower) environment. A single Abacus-2 chip will deliver 4.2 billion 16-bit additions per second and approximately 500 billion 8-bit multiplies per second, in a non-aggressive 1 micron VLSI technology. Implementation in the sub-micron technology recently available through MOSIS would of course lead to higher performance numbers.

The new design is amenable to integration with a simple sequencer and I/O subsystem, allowing scalability in system size, from a 16-chip 64 GigaOps system that can be utilized as a co-processor attached to a conventional RISC-chip based node of a MIMD system, to a standalone 256-chip TeraOp SIMD machine. The Abacus-2 chip will be designed to operate as smart memory, accessible by the controller with low latency. Corner turning will be incorporated on chip.

Scalable I/O should allow the sequencer to access any pixel in the system quickly. This could be done by adding a port to the PE chip or by accessing the external memory. For small systems, the port idea works well, since the access port can be placed on a bus and tristated. For larger systems, discrete tristate parts would have to be added to isolate the bus from capacitive loads. Corner turning can be incorporated on chip, unlike Abacus-1, since the data formats have been restricted somewhat. The only complex part would be the addressing, and that could be taken care of by another set of PALs.

9 Beyond SIMD: Abacus-3

We have a number of preliminary ideas that could lead to new possibilities in performance.

M-SIMD Operation. One way of viewing the Abacus-2 chip is as the integration of a large number of traditional discrete bit-slices (such as the AMD2903) and a flexible interconnection network on a single chip. If a functional version of a discrete sequencer was also integrated, the chip could operate as a uni-processor, ignoring the broadcast SIMD instruction and treating the on-chip memory as registers and the off-chip memory as instruction and data store. The collection of Abacus-2 chips could now operate in a mixed MIMD/SIMD mode, as required by the algorithm. This mode would not be as efficient as pure SIMD, as off-chip memory bandwidth would be reduced, but algorithms more suitable to MIMD operation could be implemented efficiently.

Systolic Operation. A small augmentation of the Abacus-2 PE would allow operation in a systolic mode, where each PE could execute a different instruction. If each PE conditionally latched each bit of the control word under local control (say as a function of the processor ID), then every PE could be programmed to perform a different operation. For example, directing some PEs to multiply their network input by an internal constant and place the result on the network output, while others summed their network inputs, results in a pipelined FIR filter. Of course, there are limitations to this mode, but the research direction seems promising.

SIMD Element or Programmable Logic? Another relatively minor modification allows the Abacus-2 chip chip to emulate programmable logic. If each PE was equipped with local address decoding, and operated in the systolic mode, then different PEs could store different truth tables and therefore emulate programmable logic.

Universal Computing Element? The combination of all three features described above leads to an almost universal computing element, able to operate as a SIMD node, as uniprocessor (with a very wide word), as a collection of systolic computing elements, or as programmable logic. Furthermore, different parts of the chip could be operating in different modes. For instance, some PEs could be serving as logic gates configured as specialized find-first-one hardware co-processors for other SIMD-mode elements.

10 Conclusions

The conclusion section, supported by an actual working device and better algorithm trace analysis, will resemble this section.

First, we tested the limits of the premise that simple one-bit PEs allow a faster overall clock rate. Although this is a common argument in research papers, existing MPP SIMD systems use a clock rate substantially slower than that of commercial bit-parallel microprocessors. The 125 MHz clock of the Abacus chip is the highest of any massively parallel SIMD systems that we're aware of. Further increases in clock speed are limited by instruction bandwidth rather than PE complexity, and therefore transfer the difficulty to off-chip interfaces and printed circuit board design. Retaining the clock speed while increasing the work done per cycle holds more promise as the approach for incorporating smaller and faster VLSI technology.

Second, we determined how effectively a collection of simple one-bit PEs could emulate bit-parallel hardware. As shown in Section 4, a variety of arithmetic circuits can be emulated at a cost of two to five cycles. However, while reconfiguration reduces the silicon cost, it increases the time requirements in two ways. Bit-level reconfiguration precludes hardware pipelining by introducing dependencies on network switch settings. Conventional techniques for dealing with data dependencies (such as forwarding) cannot be used. Additionally, reconfiguration overhead is a significant fraction of arithmetic operation times. We plan to address these issues in our next multi-bit bit-slice design.

Third, we wanted to complete an entire system to see if there are any unexpected bottlenecks and performance hits in high-speed SIMD designs. Consideration of board-level issues led to the development of instruction retiming logic, high-speed mesh signaling, low pin-count data I/O, and software control over DRAM timing. We now have a system framework that allows us to replace details of the PE core and remain plug compatible with the rest of the machine. Since the clock rate is already aggressive, we'll be able to hold it steady for a number of years while researching how to get more work done per cycle.

From a theoretical viewpoint, the Abacus implementation provides a set of realistic constants that can be used to evaluate reconfigurable mesh algorithms. The aggressive (in terms of speed and integration) nature of the design explores physical limitations on architecture that will become more significant in the future. For example, it is becoming obvious that chip boundaries cannot be abstracted away in the interests of software regularity.

Specialized architectures have fallen out of favor as commercial pressure continually drives up the performance of general purpose microprocessors. The current trend in parallel architecture is to construct parallel systems from slightly modified commodity processors. The Abacus project demonstrates that architectures optimized for specific computations can be an order of magnitude faster than such general purpose systems.

A Rough Work

A.1 Scan and Reduce Primitives

Need intro here about scans.

Scans on a mesh with bypass are logarithmic time under the unit time asumption. In practice, the assumption is false, and the network operates as a sublinear mesh.

Assume a 4 by 4 square organization of PEs. Let's look at a reduction operation, say summing the pixel values. Even though this number can overflow, for now we'll do the arithmetic in 16-bit integers. This reflects the real life case of crossing chip boundaries. [3]. [4].

A.1.1 Unpipelined loop

An unpipelined loop algorithm is:

```
for i = 0 to L {
   set_bypass_reg(i)
   configure_horizontal()
   net_out := x
   for j = 1 to n {
      wait( WaitTime(i))
      net_out := net_in            // on the 4th iteration, do Y := in
   }
   do_arithmetic()
}
```

On the ith cycle, 2^i pixels are bypassed (the first iteration, with no bypassing, is not shown). Waiting time as a function of number of PEs, p, bypassed is $p/16$ if $p \leq 32$ and $1.5p/16$ otherwise (chip crossing penalty of 1 cycle). Since $p = 4 \times 2^i = 2^{i+2}$, this can be expressed as 2^{i-2} if $i \leq 3$, and $1.5 \times 2^{i-2}$ otherwise.

[3] Describe model: ie, 2 cycles to cros chip plus one for chip boundary

[4] Note about changing orientation

Let A be the time for the arithmetic operation and other overhead. Say we want to send n bits. Then iteration i requires:

$$T_i = A + n \begin{array}{ll} (1+2^{i-2}) & \text{if } i \leq 3 \\ (1+1.5\times 2^{i-2}) & \text{otherwise} \end{array}$$

The total time $T = \sum_{i=0}^{L} T_i$ is

$$\begin{aligned} T &= L(A+n) + n(1+1+1+2) + 1.5n\sum_{i=3}^{L} 2^{i-2} && (7) \\ &= L(A+n) + 5n + 1.5n\sum_{j=1}^{L-2} 2^{j} && (8) \\ &= AL + n(L+3.5) + 1.5n2^{L-1} && (9) \\ & && (10) \end{aligned}$$

For $L = 6$, $n = 4$, $A = 9$, in this case, $T_i = 54 + 38 + 192 = 284$.

A.1.2 Pipelined Loop

As an aside, given an ideal (transmission line) wire and ideal flip flops with no setup or hold times, how many FFs should be used to pipeline the transmission of n bits? The answer is infinity, if clocking rates can be arbitrarily high.

This requires more iterations but smaller waiting time between iterations.

```
for i = 0 to 6 {
   set_bypass_reg(i)
   configure_horizontal()
   net_out := x
   for j = 1 to NumIter(i) {
      wait(WaitTime(i))
      net_out := net_in            // on the 4th iteration, do Y := in
   }
   do_arithmetic()
}
```

WaitTime(i) is easy: if messages have to cross chip boundaries, it is $n-1$, otherwise it is $\lceil p/16 \rceil$, which is effectively 1. The tricky part now is determining NumIter(i). The number of iterations is the number of bits plus the pipeline length minus 1. If we set up each chip as a pipeline stage, `NumIter(i)` is $N_{\text{chips}} + 3$. The time equation decomposes nicely into two parts: within chip and intra chip.

$$T_i = A + \begin{cases} n & \times \quad (1+1) \quad \text{if } i < 3 \\ n - 1 + 2^{i-3} & \times \quad (1+3) \quad \text{otherwise} \end{cases} \tag{11}$$

Making the substitutions:

$$T_i = A + \begin{cases} 2n & \text{if } i < 3 \\ 4n - 4 + 2^{i-1} & \text{otherwise} \end{cases} \tag{12}$$

$$\begin{array}{rcll} T & = & L(A+1) + 6n + (L-2)(4n-4) & + \sum_{i=3}^{L} 2^{i-1} \\ & & L(A+4n-3) - 2n + 8 & + \sum_{i=3}^{L} 2^{i-1} \\ & & L(A+4n-3) - 2n + 8 & + 2^L - 4 \\ & & L(A+4n-3) - 2n + 4 & + 2^L \end{array} \tag{13}$$

For $L = 6$, $A = 9$, $n = 4$, $T = 6(22) - 8 + 4 + 64 = 192$. or about 30% faster.

Compare with $T = L(A+n) + 3.5n - 4 + \frac{3n}{4}2^L$. The difference is $(\frac{3n}{4} - 1)2^L + 5.5n - 4 + L(3n-3)$, or $2(64) + 22 - 6(9) - 4 = 92$.

A.1.3 Conclusions

Notice that we get only 30% gain. Why? We would naively expect a factor of 3.

A.2 Starting Computing During Background Loading

Start Work Early? A minor modification is to allow computation to start on a nibble as soon as it's read in, and not wait for the entire word. Assuming that it takes longer to load than to compute, the time for a combined load and operate is just the load time plus the time to compute the final slice. As a result, the arithmetic time drops by the number of external loads.

If we assume that the time to process a single slice is 1, then:

$$\begin{aligned} T_{tot} &= Qm\left[(Z + 2Nf)T_{load} + N(1-f)T_{alu} + Nf\right] \\ &= Qm\left[(Z+2Nf)\frac{W}{mB} + N(1-f)\frac{W}{mk} + Nf\right] \\ &= QWN\left[\frac{l+f}{B} + \frac{1-f}{k} + \frac{f}{W}\right] \\ &= \frac{QWN}{Bk}\left[(l+f) + B\left(1 - f + \frac{fk}{W}\right)\right] \end{aligned}$$

Recall that we defined $l = Z/N$. This is faster than the non-overlapped case, but not by much. If we define X as $(l + 2f)$ and Y as k/W then:

$$\begin{aligned} R &= \frac{X + B}{X + B(1 - f + fY)} \\ &= 1 + \frac{f(1-Y)}{\frac{X}{B} + (1 - f(1-Y))} \end{aligned}$$

The increase is at most 20%, even for very high values of f and B, so we'll ignore this effect. Let's assume Y is 1/16, for 1 bit PEs and 16-bit data. Let's say 20 cycles are required to load the chip, so V is 20. Since X ranges between 0.2 and 1.2, the term X/B dominates the denominator, about 4 at the least. So R is approximately $1 + \frac{fB}{X}$, or 1.1 at most. This is a minor effect, so we'll ignore it.

References

[1] J. L. Potter, ed., *The Massively Parallel Processor.* MIT Press Series in Scientific Computation, Cambridge, Massachusetts: The MIT Press, 1985.

[2] E. L. Cloud, "The geometric arithmetic parallel processor," in *Frontiers of Parallel Computation '88*, 1988.

[3] T. Blank, "The maspar mp-1 architecture," in *International Conference on Computer Architecture*, 1990.

[4] R. Tuck and W. Kim, "MasPar MP-2 PE Chip: A Totally Cool Hot Chip," in *IEEE 1993 Hot Chips Symposium*, March 1993.

[5] H. Li and M. Maresca, "Polymorphic-torus architecture for computer vision," *IEEE Trans. Pattern Analysis and Machine Intellegence*, vol. 11, no. 3, pp. 233–243, 1989.

[6] D. W. Blevins, E. W. Davis, R. A. Heaton, and J. H. Reif, "Blitzen: A highly integrated massively parallel machine," in *Frontiers of Parallel Computation '88*, 1988.

[7] D. B. Shu, G. Nash, and C. Weems, *Image Understanding Architecture and Applications*, ch. 9, pp. 297–355. Springer-Verlag, 1989.

[8] A. Rushton, *Reconfigurable Processor-Array: a bit-sliced parallel computer.* Research Monograph in Parallel and Distributed Computing, Pitman, 1989.

[9] N. Thacker, P. Courtney, S. Walker, S. Evans, and R. Yates, "Specification and Design of a General Purpose Image Processing Chip," in *International Conference on Pattern Recognition*, pp. 268–273, IEEE, 1994.

[10] M. Gokhale, B. Holmes, and K. Iobste, "Processing in Memory: The Terasys Massively Parallel PIM Array," *IEEE Computer*, pp. 23–31, April 1995.

[11] P. Kogge, T. Sunaga, H. Miyataka, K. Kitamura, and E. Retter, "Combined DRAM and Logic Chip for Massively Parallel Systems," in *Proceedings of the 16th Conference on Advanced Research in VLSI*, pp. 4–16, IEEE Computer Society Press, March 1995.

[12] A. Yeung and J. Rabaey, "A 2.4 GOPS Data-Driven Reconfigurable Multiprocessor IC for DSP," in *ISSCC Digest of Technical Papers*, pp. 108–110, IEEE, February 1995.

[13] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable active memories: A performance assessment," prl report, DEC Paris Reserch Laboratory, 85, Av. Victor Hugo, 92563 Rueil-Malmaison Cedex, France, June 1992.

[14] D. Audet, Y. Savaria, and J.-L. Houle, "Performance improvements to vlsi parallel systems, using dynamic concatenation of processing resources," *Parallel Computing*, vol. 18, pp. 149–167, 1992.

[15] M. C. Herbordt, *The Evaluation of Massively Parallel Array Architectures.* PhD thesis, University of Massachusetts at Amherst, 1994.

[16] M. Bolotski, R. Barman, J. J. Little, and D. Camporese, "Silt: A distributed bit-parallel architecture for early vision," *International Journal of Computer Vision*, vol. IJCV-11, pp. 63–74, 1993.

[17] A. DeHon, T. F. Knight Jr., and T. Simon, "Automatic impedance control," in *ISSCC Digest of Technical Papers*, pp. 164–165, IEEE, February 1993.

[18] B. K. P. Horn, *Robot Vision.* MIT Electrical Engineering and Computer Science Series, Cambridge, Massachusetts: The MIT Press, 1986.

[19] J. G. Harris, "The coupled depth/slope approach to surface reconstruction," Master's thesis, MIT, May 1986.

[20] J. J. Little and H. H. Bülthoff, "Parallel optical flow using local voting," AI Memo 929, MIT, July 1988.

[21] S. Levialdi, "On Shrinking Binary Picture Patterns," *Communication of the ACM*, vol. 15, Jan 1972.

[22] F. Thomson Leighton, *Introduction to Parallel Algorithms And Architectures: Arrays, Trees, Hypercubes.* Morgan-Kaufmann, 1992.

[23] Sotirious G. Ziavras, "Connected Component Labelling on the BLITZEN Massively Parallel Processor," *Image and Vision Computing*, vol. 11, Dec 1993.

[24] Alok Choudhary and Janak H. Patel, *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*, pp. 69–78. Kluwer Academic Publishers, 1990.

[25] R. E. Cypher and Jorge Sanz and Larry Snyder, "Algorithms for Image Component Labelling on SIMD Mesh-Connected Computers," *IEEE Trans. on Computers*, vol. 39, Feb 1990.

[26] R. E. Cypher and Jorge Sanz, "The Hough Transform Has O(N) Complexity on SIMD MxN Mesh Array Architectures," *IEEE Trans. on Computers*, vol. 39, Feb 1090.

[27] Michael J. Quinn, *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1987.

[28] C. P. Schnorr and A. Shamir, "An Optimal Sorting Algorithm for Mesh Connected Computers," in *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, ACM, 1986.

[29] Martin C. Herbordt and James C. Corbett and Charles C. Weems and John Spalding, "Practical Algorithms for Online Routing on Fixed and Reconfigurable Meshes," *Journal of Parallel and Distributed Computing*, vol. 20, pp. 341–356, 1994.